CTF Writeup: so-AIRMES

Insomnihack 2026 · Orange Cyberdefense Switzerland · Mobile / Reverse Engineering

`soairmes-19d57e84105a9ea0fa7da0646b2b47f853cb4ba366026d88c7c13dee0de6dbb5.apk`

Anthony Bondu (Fawlen) · Oterihack · Symbiotic Security

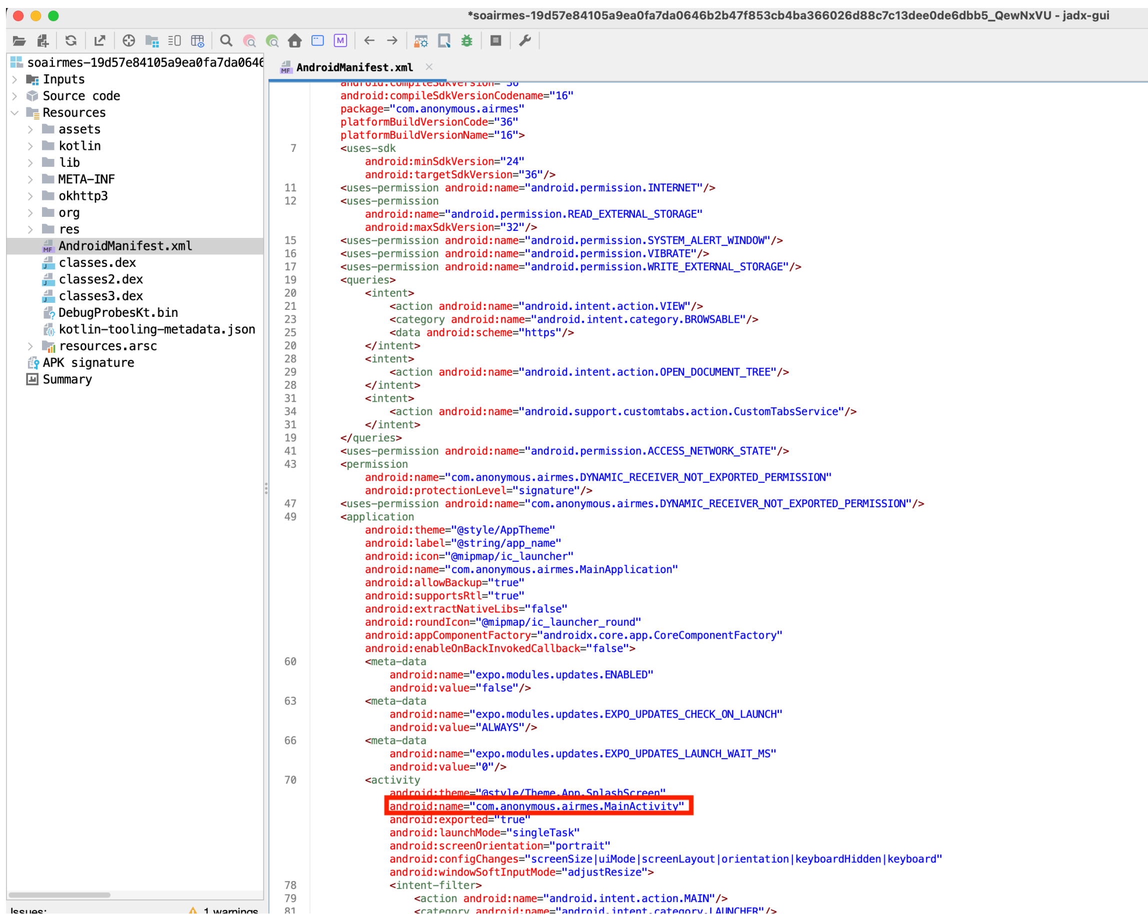
Initial Recon

We are given an APK file. Unfortunately, I couldn't do any dynamic analysis during the CTF because the internet connection was too slow to download the Android tools needed (Android SDK, emulator, etc.). So this was done entirely through static analysis.

The first reflex for any Android reverse challenge is to open it in **JADX**.

AndroidManifest.xml

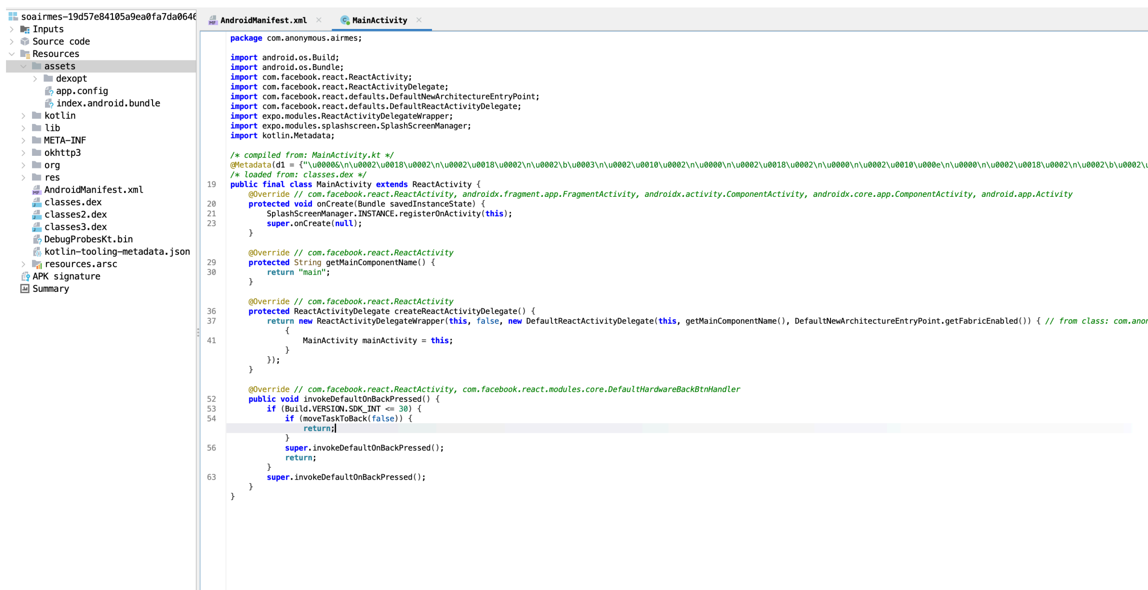
The first thing we check is the `AndroidManifest.xml` because it's the quickest way to find the application's entry point.



We can read the package name (`com.anonymous.airmes`) and, scrolling down to the `<activity>` tag with the `MAIN`/`LAUNCHER` intent filter, we find the main activity: `com.anonymous.airmes.MainActivity`.

Main Activity

Clicking on `com.anonymous.airmes.MainActivity` in JADX, we immediately see it extends `ReactActivity`:



This confirms we're dealing with a **React Native** application. The Java/Kotlin layer is just a thin shell and the real application logic lives in JavaScript, bundled inside the APK.

Exploring the Assets

Since it's React Native, we navigate to `assets/` in the APK and find `index.android.bundle`. This is **not** plain JavaScript, it's compiled **Hermes bytecode**. Hermes is Meta's JavaScript engine for React Native that precompiles JS into bytecode for faster startup. Opening it in a text editor shows binary data, not readable JS.

At this point, we extract the full APK to work with the files directly:

```
1  soairmes.apk
2  ├── assets/
3  │   └── index.android.bundle        <- Hermes bytecode (compiled JS)
4  ├── lib/
5  │   ├── arm64-v8a/
6  │   │   └── libairmesnative.so      <- Native C++ module
7  │   ├── armeabi-v7a/
8  │   │   └── libairmesnative.so
9  │   ├── x86/
10 │   │   └── libairmesnative.so
11 │   └── x86_64/
12 │       └── libairmesnative.so
13 ├── classes.dex
14 └── AndroidManifest.xml
```

We note the presence of `libairmesnative.so` compiled for four architectures, but we don't know its role yet. Let's first analyze the Hermes bytecode to understand the application flow. The native library will come into play later.

Hermes Bytecode Analysis with hermes-decomp

Since the JavaScript is compiled to Hermes bytecode, we need a specialized tool to analyze it. Standard JS deobfuscators won't work here. We use **hermes-decomp** by Symbiotic Security, a Rust-based decompiler for Hermes bytecode that can disassemble, decompile, and perform cross-reference analysis on `.hbc` files. This challenge actually served as one of the real-world test cases for the tool.

Bundle Overview

```
1 $ hermes-decomp info index.android.bundle
```

```
~/Desktop/writeup_solve » /Users/antho/Documents/hermes_decompiler
android.bundle
Hermes Bytecode Info
  Version: 96
  Layout: Legacy
  Function header layout: Legacy16
  Functions: 10176
  Strings: 9438
  Identifiers: 5249
  RegExp: 163
  CJS Modules: 0
  BigInt: 0
  Function sources: 38
  Instruction offset: 412028
```

10,176 functions, 9,438 strings. That's a full React Native app with all its dependencies bundled in. We need a strategy to find the challenge logic in this haystack.

Full Decompilation First

Rather than guessing which function to look at, we start by decompiling **everything** and extracting all Metro modules to get an overview of the application structure:

```
1 $ hermes-decomp decompile index.android.bundle --output full_decompile.js
2 $ hermes-decomp extract index.android.bundle --output modules/
3 $ hermes-decomp dump --kind strings index.android.bundle > strings_dump.txt
```

This gives us 956 extracted Metro modules and a full string table. Searching through the strings immediately reveals the interesting ones:

```
1 [342]  "so-AIRMES"
2 [941]  "REVEAL FLAG"
3 [2692] "Native module missing: AirmesCrypto.reveal()"
4 [4528] "insomnihack 2026 • Orange Cyberdefense Switzerland"
5 [6011] "AirmesCrypto"
```

Now we use `xref` to trace where these strings are referenced:

```
1 $ hermes-decomp xref index.android.bundle --query "AirmesCrypto"
2   Function 10138 () at offset 00d0: GetById        <- module import
3   Function 10143 (?anon_0_) at offset 0057: LoadConstString  <- error message
```

```
~/Desktop/writeup_solve » /Users/antho/Documents/hermes_decompiler/target/release/hermes-decomp xref /Users/antho/Desktop/writeup_solve/challenges/so-AIRMES/apk/extracted/assets/index.android.bundle --query "AirmesCrypto"
Found 2 cross-references for 'AirmesCrypto':
  Function 10138 () at offset 00d0: GetById
  Function 10143 (?anon_0_) at offset 0057: LoadConstString
```

This tells us exactly where to look: **Function 10138** is the module that imports `AirmesCrypto`, and **Function 10143** uses it. Now we can dive into these specific functions.

Understanding the Application Flow

The Main Module (F10138)

```
1 $ hermes-decomp decompile index.android.bundle --function 10138
```

hermes-decomp produces the following decompiled output (truncated for brevity):

```
1  // Function 10138 - Decompiled
2  function f10138(arg0, arg1, arg2, arg3, arg4, arg5, arg6, arg7) {
3    closure_0 = arg1;
4    closure_1 = arg6;
5    fn = /* F10139 */ function _interopDefault() { ... };
6    closure_9 = /* F10140 */ function Index() { ... };
7    closure_10 = /* F10146 */ function statusDotStyle() { ... };
8    // ...
9    closure_7 = tmp2.NativeModules.AirmesCrypto;
10   StyleSheet = tmp2.StyleSheet;
11   obj2 = { root: { flex: 1, backgroundColor: "#050505", ... }, ... };
12   closure_8 = StyleSheet.create(obj2);
13 }
```

This is the Metro module (module 950) that initializes the challenge. The key line is `closure_7 = tmp2.NativeModules.AirmesCrypto`, which stores a reference to the native module. We can also see it registers a React component `Index` (F10140) and a `statusDotStyle` helper.

The UI Component: Index (F10140)

```
1 $ hermes-decomp decompile index.android.bundle --function 10140
```

```
1  // Function 10140 - Decompiled
2  function Index(arg0) {
3    tmp13 = undefined;
4    defaultResult = closure_3.default(closure_4.useState("READY"), 2);
5    tmp2 = defaultResult[0];
6    closure_0 = defaultResult[1];
7    defaultResult1 = closure_3.default(closure_4.useState(""), 2);
8    [r10020, closure_1] = defaultResult1;
9    // ...
10   obj = { style: closure_8.title, children: "so-AIRMES" };
11   items[1] = closure_6.jsx(closure_5.Text, obj);
12   obj3 = { style: closure_8.subtitle, children: "Challenge" };
13   items[2] = closure_6.jsx(closure_5.Text, obj3);
14   // ...
15   obj6 = { style: closure_8.buttonText, children: "REVEAL FLAG" };
16   items[4] = closure_6.jsx(closure_5.Pressable, obj5);
17   // ...
```

```
18    obj10.children = "insomnihack 2026 • Orange Cyberdefense Switzerland";
19  }
```

The decompiled output clearly shows a React component. We can read the UI structure directly: a title `"so-AIRMES"`, a `"REVEAL FLAG"` button wired to `onPressReveal()`, and status indicators. Following the closure chain, the button handler is created in **F10141**, which wraps an async generator **F10142/ F10143**.

The Reveal Handler (F10143): The Core Logic

```
1  $ hermes-decomp decompile index.android.bundle --function 10143
```

```
~/Desktop/writeup_solve » /Users/antho/Documents/hermes_decompiler/target/release/hermes-decomp decompile /Users/antho/Desktop/writeup_solve/challenges/so-AIRMES/apk/extracted/assets/index.android.bundle --f
unction 10143
function f10143(arg0) {
  tmp7 = undefined;
  tmp8 = closure_0(undefined, "WORKING");
  tmp9 = closure_1(undefined, "");
  try {
    tmp11 = null;
    tmp12 = undefined;
    if (closure_7 == null) {
      if (reveal) {
        try {
          tmp17 = closure_7;
          tmp18 = yield closure_7.reveal(closure_7);
          obj = tmp18;
          tmp19 = closure_1;
          tmp20 = closure_1(tmp7, tmp18);
          if (tmp18 === "ACCESS DENIED") {
            tmp27 = closure_0;
            str8 = "DENIED";
            tmp28 = closure_0(tmp7, "DENIED");
          } else if (obj === "NOT PROVISIONED") {
            tmp25 = closure_0;
            str7 = "PROVISION";
            tmp26 = closure_0(tmp7, "PROVISION");
          } else {
            tmp21 = obj;
            str4 = "INS{";
            tmp22 = closure_0;
            if (obj.startsWith(obj, "INS{")) {
              str6 = "GRANTED";
              tmp22Result = tmp22(tmp7, "GRANTED");
            } else {
              str5 = "ERROR";
              tmp22Result1 = tmp22(tmp7, "ERROR");
            }
          }
        } catch (tmp6) {
        }
      } else {
        tmp13 = closure_0;
        str2 = "ERROR";
        tmp14 = closure_0(tmp7, "ERROR");
        tmp15 = closure_1;
        str3 = "Native module missing: AirmesCrypto.reveal()";
        tmp16 = closure_1(tmp7, "Native module missing: AirmesCrypto.reveal()");
        return tmp7;
      }
    } else {
      reveal = tmp10.reveal;
    }
  } catch (tmp6) {
    _String = closure_0;
    _String = undefined;
    if (__exception == null) {
      str = _String;
      if (_String != tmp5) {
        _String = str;
        str = globalThis;
        str = HermesInternal;
        str = "ERROR: ";
        str = "ERROR: " + "ERROR: ";
        str = tmp2(tmp, str);
      } else {
        _String = globalThis;
        _String = String;
        str = String(tmp, tmp3);
        break;
      }
    } else {
      _String = tmp6.message;
    }
  }
  return;
```

F10143 is a **generator function** (async/await). **hermes-decomp** handles the generator state machine and produces the following decompiled output:

```
1   function f10143(arg0) {
2     tmp7 = undefined;
3     tmp8 = closure_0(undefined, "WORKING");
4     tmp9 = closure_1(undefined, "");
5     try {
6       tmp11 = null;
7       tmp12 = undefined;
8       if (closure_7 == null) {
9         if (reveal) {
10          try {
11            tmp17 = closure_7;
```

```
12            tmp18 = yield closure_7.reveal(closure_7);
13            obj = tmp18;
14            tmp19 = closure_1;
15            tmp20 = closure_1(tmp7, tmp18);
16            if (tmp18 === "ACCESS DENIED") {
17              tmp28 = closure_0(tmp7, "DENIED");
18            } else if (obj === "NOT PROVISIONED") {
19              tmp26 = closure_0(tmp7, "PROVISION");
20            } else {
21              if (obj.startsWith(obj, "INS{")) {
22                tmp22Result = closure_0(tmp7, "GRANTED");
23              } else {
24                tmp22Result1 = closure_0(tmp7, "ERROR");
25              }
26            }
27          } catch (tmp6) { }
28        } else {
29          closure_0(tmp7, "ERROR");
30          closure_1(tmp7, "Native module missing: AirmesCrypto.reveal()");
31          return tmp7;
32        }
33      } else {
34        reveal = tmp10.reveal;
35      }
36    } catch (tmp6) { ... }
37    return;
38  }
```

The decompiled output is very readable. We can directly follow the logic:

1. `closure_0("WORKING")` sets status to WORKING
2. `closure_1("")` clears the output
3. Check if `closure_7` (which is `AirmesCrypto`) exists and has a `reveal` method
4. `result = yield closure_7.reveal()` calls the native method (yield = await in generator context)
5. If result starts with `"INS{"` → status `"GRANTED"`, **this is the flag**
6. Otherwise: `"ACCESS DENIED"` → `"DENIED"`, `"NOT PROVISIONED"` → `"PROVISION"`, else → `"ERROR"`

This tells us everything we need to know on the JS side:

- The flag comes from `AirmesCrypto.reveal()`, a **native JNI call**
- The flag format is `INS{...}` (confirmed by the `startsWith` check)
- The JS layer is just UI, all the real crypto happens in native code

Back to JADX: Finding the Native Bridge

Now that we know the JavaScript calls `NativeModules.AirmesCrypto.reveal()`, we go back to JADX and search for `AirmesCrypto`. We find `com.anonymous.airmes.crypto.AirmesCryptoModule`, a Kotlin class that:

- Loads the native library with `System.loadLibrary("airmesnative")`

- Declares an `external fun nativeReveal(): String`, the JNI bridge
- Exposes `reveal()` to JavaScript, which simply calls `nativeReveal()`

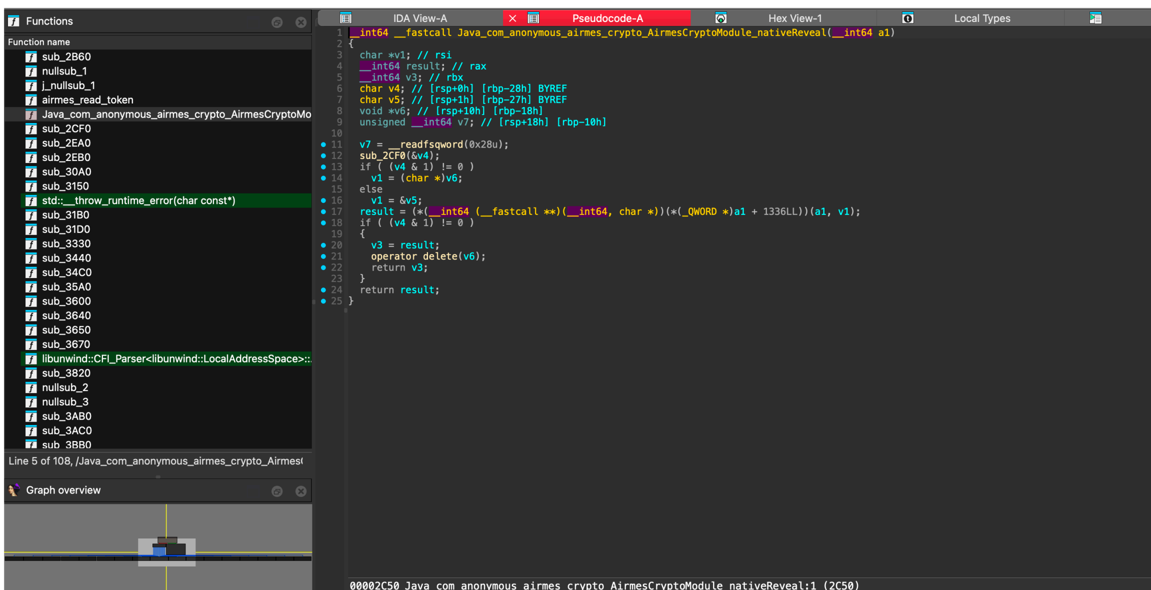
So this is where `libairmesnative.so` comes in. The Kotlin layer is a thin pass-through: when JS calls `AirmesCrypto.reveal()`, it goes straight to native C++ code via JNI.

Reversing the Native Library in IDA

Now we switch to IDA to reverse engineer `libairmesnative.so`. We use the **x86_64 version** for cleaner pseudocode output.

Identifying Entry Points

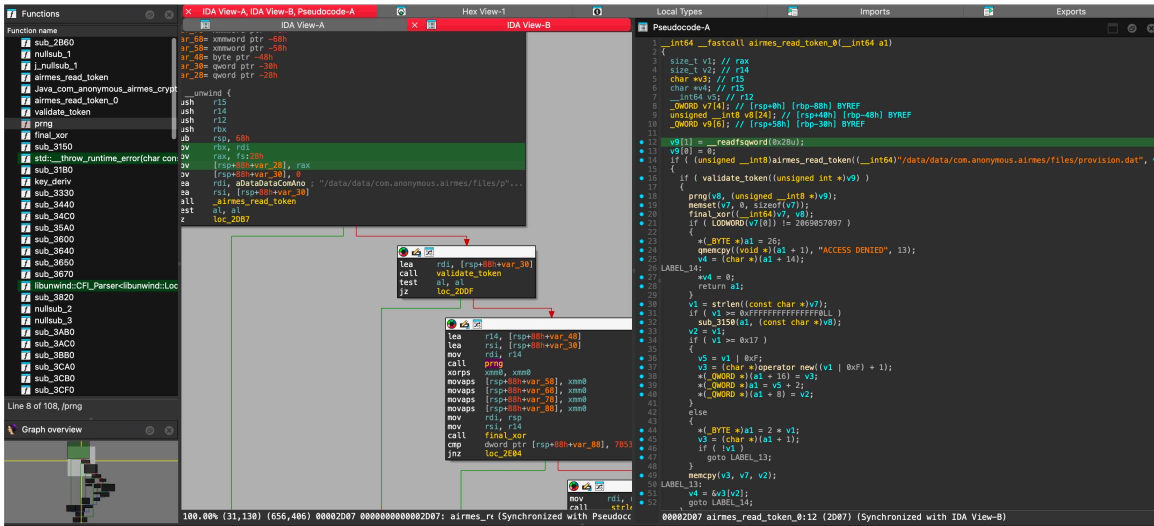
The function list in IDA immediately shows two named symbols: `airmes_read_token` and `Java_com_anonymous_airmes_crypto_AirmesCryptoModule_nativeReveal`. The rest are unnamed `sub_XXXX`.



Architecture

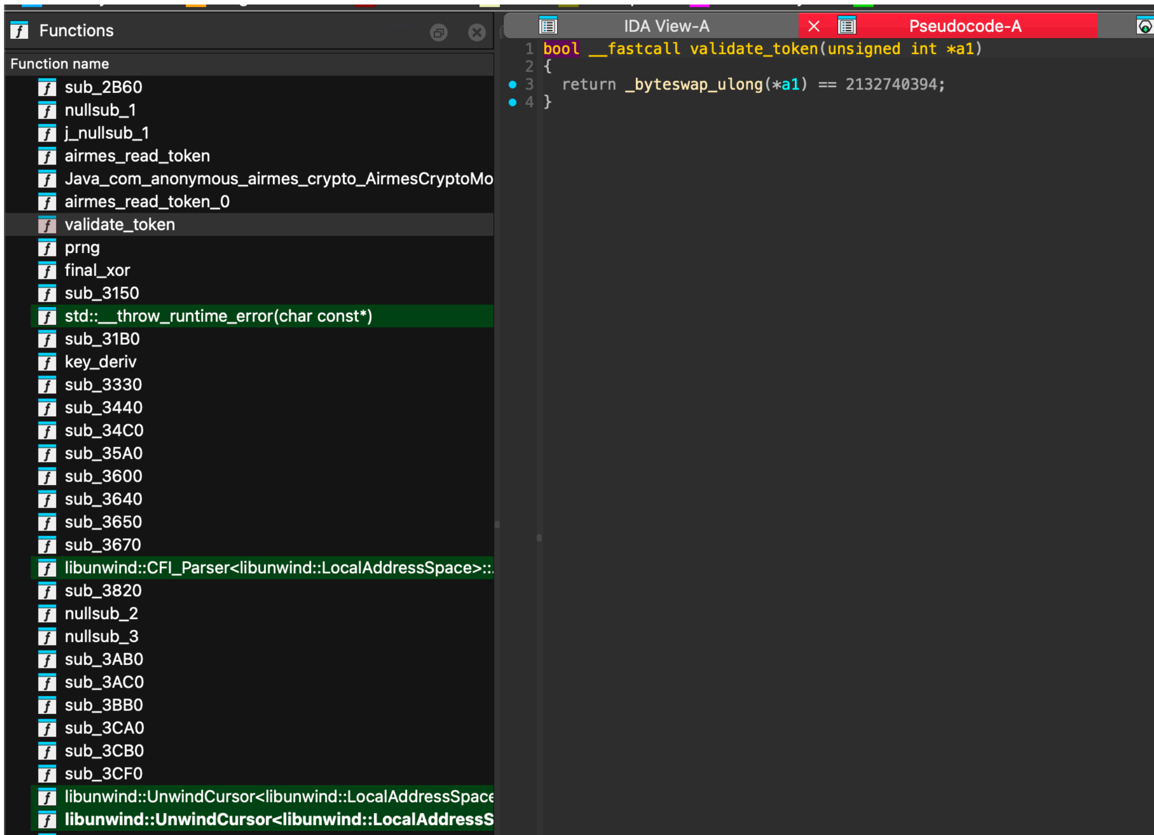
```
1  Java_...nativeReveal (0x2C50)
2       └── sub_2CF0 (internal wrapper)
3           ├── airmes_read_token()      reads 8 bytes from provision.dat
4           ├── sub_2EA0()               validates magic bytes (bswap check)
5           ├── sub_2EB0()               djb2 hash + xorshift32 PRNG
6           ├── sub_31D0()               key derivation (ROL4 mixing)
7           └── sub_30A0()               XOR keystream with constants → flag
```

airmes_read_token



This function opens `/data/data/com.anonymous.airmes/files/provision.dat`, reads exactly 8 bytes into a buffer, and returns a boolean indicating success. If the file doesn't exist or has fewer than 8 bytes, it returns false, and `nativeReveal` returns `"NOT PROVISIONED"`.

`sub_2EA0`: Token Validation



IDA decompiles this to one line:

```
1 bool __fastcall sub_2EA0(unsigned int *a1)
2 {
```

```
3     return _byteswap_ulong(*a1) == 2132740394;
4 }
```

`2132740394` in hex is `0x7F1F092A`. So the first 4 bytes of the token must be `0x2A, 0x09, 0x1F, 0x7F` (little-endian). If validation fails, `nativeReveal` returns `"ACCESS DENIED"`.

`sub_31D0`: Key Derivation

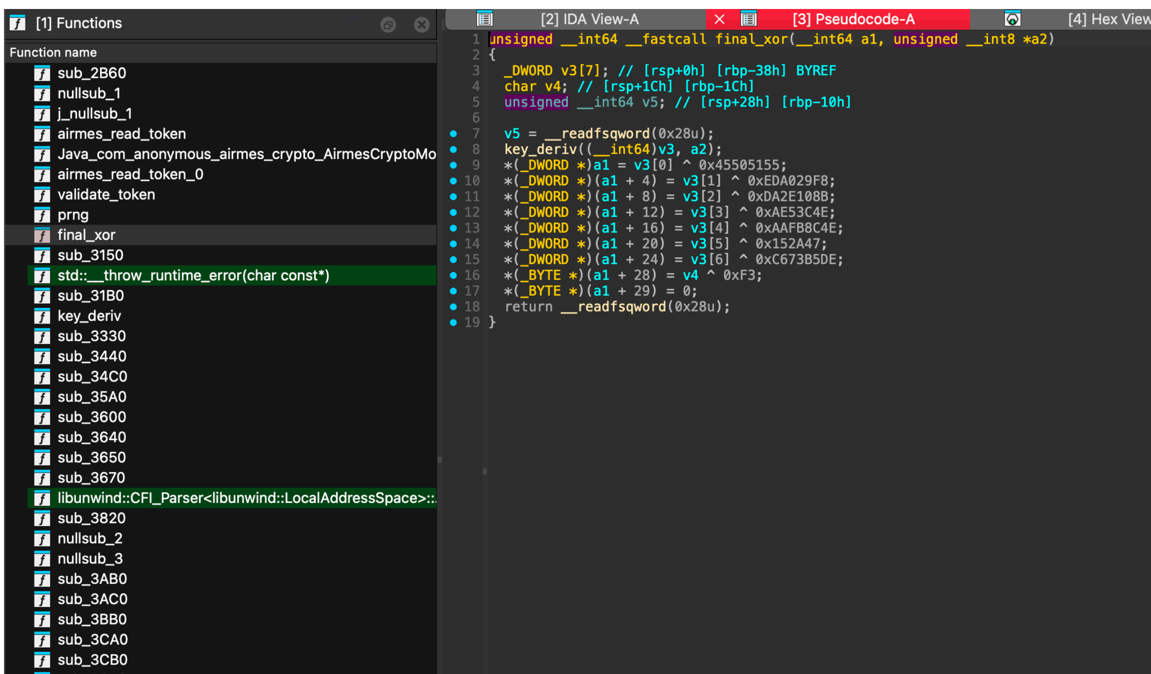
IDA shows a massive nested expression of `__ROL4__` (rotate left 32-bit) operations. The constant `1640531527` is the unsigned representation of `-0x9E3779B9` (the golden ratio constant, commonly used in hash functions like TEA/XTEA).

The function takes 16 PRNG bytes and mixes them through:

- Processing bytes in groups of 4 with variable shifts (`<< 8`, `<< 16`, `<< 24`)
- Subtracting `0x9E3779B9` at each round
- Nested `ROL4` with rotation count of 5

The result is a single **32-bit seed** that feeds into a final xorshift32 to produce **29 bytes of keystream**.

`sub_30A0`: XOR Final



The IDA pseudocode for this function is clean and we can read the constants directly:

```
1  unsigned __int64 __fastcall sub_30A0(__int64 a1, unsigned __int8 *a2)
2  {
3      _DWORD v3[7];
4      sub_31D0(v3, a2);                       // key derivation -> keystream
5
6      *(_DWORD *)a1        = v3[0] ^ 0x45505155;
7      *(_DWORD *)(a1 + 4)  = v3[1] ^ 0xEDA029F8;
```

```
8       *(_DWORD *)(a1 + 8) = v3[2] ^ 0xDA2E108B;
9       *(_DWORD *)(a1 + 12) = v3[3] ^ 0x0AE53C4E;
10      *(_DWORD *)(a1 + 16) = v3[4] ^ 0xAAFB8C4E;
11      *(_DWORD *)(a1 + 20) = v3[5] ^ 0x00152A47;
12      *(_DWORD *)(a1 + 24) = v3[6] ^ 0xC673B5DE;
13      *(_BYTE *)(a1 + 28) = v4 ^ 0xF3;
14      *(_BYTE *)(a1 + 29) = 0;              // null terminator
15  }
```

The 29-byte keystream is XORed against hardcoded constants to produce the flag string directly.

From IDA Pseudocode to C Solver

The key insight is understanding how to translate IDA's pseudocode into a working solver. Let's walk through the process.

Identifying the xorshift32 pattern

Inside `sub_31D0`, after the `__ROL4__` mixing produces a 32-bit seed, IDA shows a loop that repeatedly applies three operations to a state variable:

```
1 v2 ^= (v2 << 13);
2 v2 ^= (v2 >> 17);
3 v2 ^= (v2 << 5);
```

This is the classic **xorshift32** PRNG by George Marsaglia. Each iteration produces one pseudo-random 32-bit value, and the low byte (`v2 & 0xFF`) is stored as one byte of keystream. The loop runs 29 times to produce 29 keystream bytes.

The same xorshift32 pattern also appears earlier in `sub_2EB0` (the PRNG generation step), where it runs 16 times after a djb2 hash of the token.

Translating the XOR constants

In `sub_30A0`, IDA gives us the XOR operation as `_DWORD` (32-bit) writes. For example:

```
1 *(_DWORD *)a1 = v3[0] ^ 0x45505155;
```

This means the first 4 bytes of the flag are computed as `keystream[0..3] XOR 0x45505155`. Since x86 is little-endian, the bytes are stored as `[0x55, 0x51, 0x50, 0x45]`. In the solver, we need to XOR byte-by-byte with the constant split into its individual bytes:

```
1 flag[i*4]   = ks[i*4]   ^ (c & 0xFF);         // low byte
2 flag[i*4+1] = ks[i*4+1] ^ ((c >> 8) & 0xFF);
3 flag[i*4+2] = ks[i*4+2] ^ ((c >> 16) & 0xFF);
4 flag[i*4+3] = ks[i*4+3] ^ ((c >> 24) & 0xFF); // high byte
```

The bruteforce shortcut

The entire chain (token → djb2 → xorshift32 → ROL4 mixing → xorshift32 → XOR) ultimately depends on a single 32-bit value: the seed that enters the final xorshift32 loop. Since we know the flag starts with `"INS{"`, we can compute what the first 4 keystream bytes must be, and bruteforce all 2^{32} possible seeds. With early rejection (checking one byte at a time), this runs in seconds.

Reconstructed Algorithm

Looking at the full chain, everything collapses into a simple dependency:

```
token[8] -> djb2 -> xorshift32 x16 -> 16 bytes PRNG -> key_derivation (ROL4 mixing) ->
seed2 (32-bit) -> xorshift32 x29 -> 29 bytes keystream -> XOR constants -> FLAG
```

The entire algorithm depends on a single 32-bit seed. We don't need `provision.dat`, the token, or even the intermediate steps. We can bruteforce all 2^{32} possible seed values for `seed2`.

Known Constraint

The flag starts with `"INS{"` = `[0x49, 0x4E, 0x53, 0x7B]`.

The first 4 keystream bytes must therefore satisfy:

```
keystream[0] = 0x49 ^ 0x55 = 0x1C
keystream[1] = 0x4E ^ 0x51 = 0x1F
keystream[2] = 0x53 ^ 0x50 = 0x03
keystream[3] = 0x7B ^ 0x45 = 0x3E
```

Each byte constraint eliminates 255/256 of candidates. Four constraints reduce the search space to approximately **1 valid seed**.

Solver

```c
#include <stdio.h>
#include <stdint.h>

static const uint32_t xor_consts[] = {
    0x45505155, 0xeda029f8, 0xda2e108b, 0x0ae53c4e,
    0xaafb8c4e, 0x00152a47, 0xc673b5de
};

static inline uint32_t xorshift32(uint32_t x) {
    x ^= (x << 13); x ^= (x >> 17); x ^= (x << 5);
    return x;
}

int main() {
```

```
15      uint8_t target[4] = {0x49^0x55, 0x4e^0x51, 0x53^0x50, 0x7b^0x45};
16      for (uint64_t s = 0; s < 0x100000000ULL; s++) {
17          uint32_t st = (uint32_t)s;
18          st = xorshift32(st); if ((st&0xFF) != target[0]) continue;
19          st = xorshift32(st); if ((st&0xFF) != target[1]) continue;
20          st = xorshift32(st); if ((st&0xFF) != target[2]) continue;
21          st = xorshift32(st); if ((st&0xFF) != target[3]) continue;
22
23          uint8_t ks[29]; st = (uint32_t)s;
24          for (int i = 0; i < 29; i++) {
25              st = xorshift32(st); ks[i] = st & 0xFF;
26          }
27
28          uint8_t flag[30];
29          for (int i = 0; i < 7; i++) {
30              uint32_t c = xor_consts[i];
31              flag[i*4]=ks[i*4]^(c&0xFF);
32              flag[i*4+1]=ks[i*4+1]^((c>>8)&0xFF);
33              flag[i*4+2]=ks[i*4+2]^((c>>16)&0xFF);
34              flag[i*4+3]=ks[i*4+3]^((c>>24)&0xFF);
35          }
36          flag[28] = ks[28] ^ 0xf3; flag[29] = 0;
37          if (flag[0]=='I' && flag[27]=='}')
38              printf("FLAG: %s (seed=0x%08x)\n", flag, (uint32_t)s);
39      }
40  }
```

```
1  $ cc -O3 -o solver solver.c && time ./solver
2  FLAG: INS{r3v_n4tiv3_cpp_w311_d0n3} (seed=0x060dce7d)
3  ./solver  6.78s user 0.06s sys
```

Flag

INS{r3v_n4tiv3_cpp_w311_d0n3}