Arab War Games Mobile Challenge Myth - Write - Up

Challenge Overview

The "Myth" challenge from Arab War Games presents a mobile application that promises to reveal ancient secrets when certain conditions are met. The application appears to implement multiple protection mechanisms including emulator detection, and runtime manipulation checks.

Initial Analysis

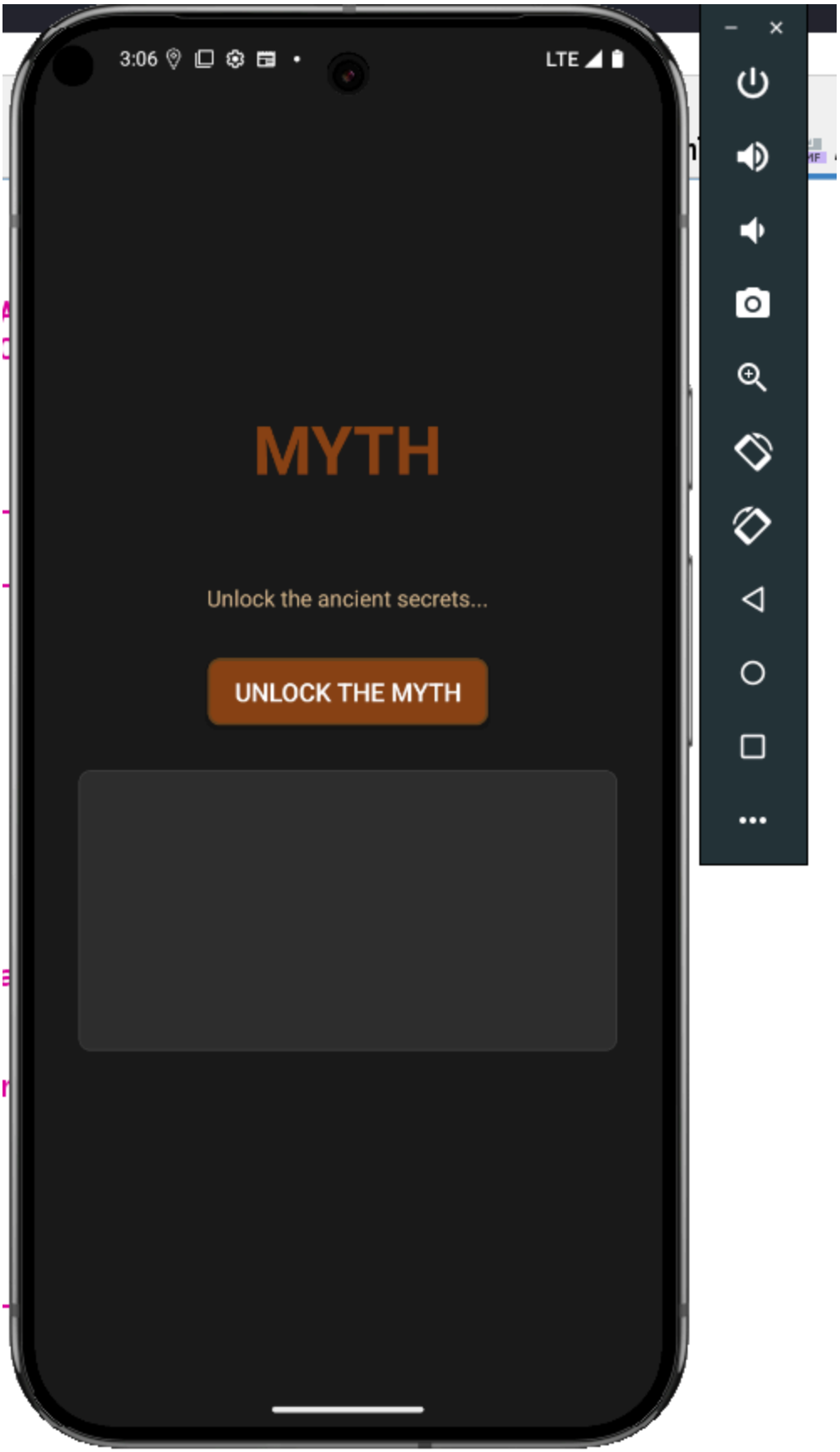
Application Behavior

Upon launching the application, we're presented with a simple interface containing a button labeled "UNLOCK THE MYTH". Clicking this button immediately crashes the application when run in an emulator, suggesting the presence of anti-analysis protections.

Manifest Examination

Analysis of the AndroidManifest.xml reveals several key details:

- Required permissions:
  - Location access (both fine and coarse)
  - Vibration
  - Internet
- Single activity (`MainActivity`) exported with MAIN/LAUNCHER intent filter


3:06
LTE
MYTH
Unlock the ancient secrets...
UNLOCK THE MYTH

```
SyntheticLambda0 × MainActivity × R × res/xml/backup_rules.xml × res/xml/data_extraction_rules.xml
    <uses-sdk
        android:minSdkVersion="24"
        android:targetSdkVersion="36"/>
    <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"/>
    <uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION"/>
    <uses-permission android:name="android.permission.VIBRATE"/>
    <uses-permission android:name="android.permission.INTERNET"/>
    <permission
        android:name="asc.wargames.myth.DYNAMIC_RECEIVER_NOT_EXPORTED_PERMISSION"
        android:protectionLevel="signature"/>
    <uses-permission android:name="asc.wargames.myth.DYNAMIC_RECEIVER_NOT_EXPORTED_PERMISSION"/>
    <application
        android:theme="@style/Theme.AppCompat.DayNight.NoActionBar"
        android:label="@string/app_name"
        android:icon="@mipmap/ic_launcher"
        android:debuggable="true"
        android:allowBackup="true"
        android:supportsRtl="true"
        android:extractNativeLibs="false"
        android:fullBackupContent="@xml/backup_rules"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:appComponentFactory="androidx.core.app.CoreComponentFactory"
        android:dataExtractionRules="@xml/data_extraction_rules">
        <activity
            android:theme="@style/Theme.AppCompat.DayNight.NoActionBar"
            android:label="@string/app_name"
            android:name="asc.wargames.myth.MainActivity"
            android:exported="true">
            <intent-filter>
                <action android:name="android.intent.action.MAIN"/>
                <category android:name="android.intent.category.LAUNCHER"/>
            </intent-filter>
        </activity>
        <provider
            android:name="androidx.startup.InitializationProvider"
```

Reverse Engineering

MainActivity Analysis

The MainActivity class contains several important elements:

```
/* loaded from: classes3.dex */
public final class MainActivity extends AppCompatActivity {
    private static final String TAG = "Myth";
    private final double SACRED_LAT = 30.043801d;
    private final double SACRED_LON = 31.334688d;
    private TextView resultText;
    private Vibrator vibrator;

    public final native String getFlag(double latitude, double longitude, long currentTime);

    static {
        System.loadLibrary("vault");
    }

    @Override // androidx.fragment.app.FragmentActivity, androidx.activity.ComponentActivity, androidx.core.app.Componen
    protected void onCreate(Bundle savedInstanceState) {
        Vibrator vibrator;
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        if (checkDebugger() || checkEmulator() || checkFrida()) {
            Log.e(TAG, "Debugging environment detected - exiting");
            finish();
            return;
        }
```

Key observations:

1. The application loads a native library called "vault"

2. It implements multiple protection mechanisms:
   - Debugger detection (`checkDebugger()`)
   - Emulator detection (`checkEmulator()`)
   - Frida detection (`checkFride()`)
3. A native method `getFlag()` requires three parameters:
   - Latitude
   - Longitude
   - Current timestamp

UnlockMyth Function

The `unlockMyth()` function (called when pressing the button) performs several operations:

1. Verifies the protection mechanisms
2. Checks location permissions
3. Generates a timestamp using `System.currentTimeMillis() + 2000` (current time plus 2 seconds)
4. Calls the native `getFlag()` function with the parameters

```
*/
public final void unlockMyth() {
    /*
        Method dump skipped, instructions count: 669
        To view this dump change 'Code comments level' option to 'DEBUG'
    */
    throw new UnsupportedOperationException("Method not decompiled: asc.wargames.myth.MainActivity.unlockMyth():void");
}
```

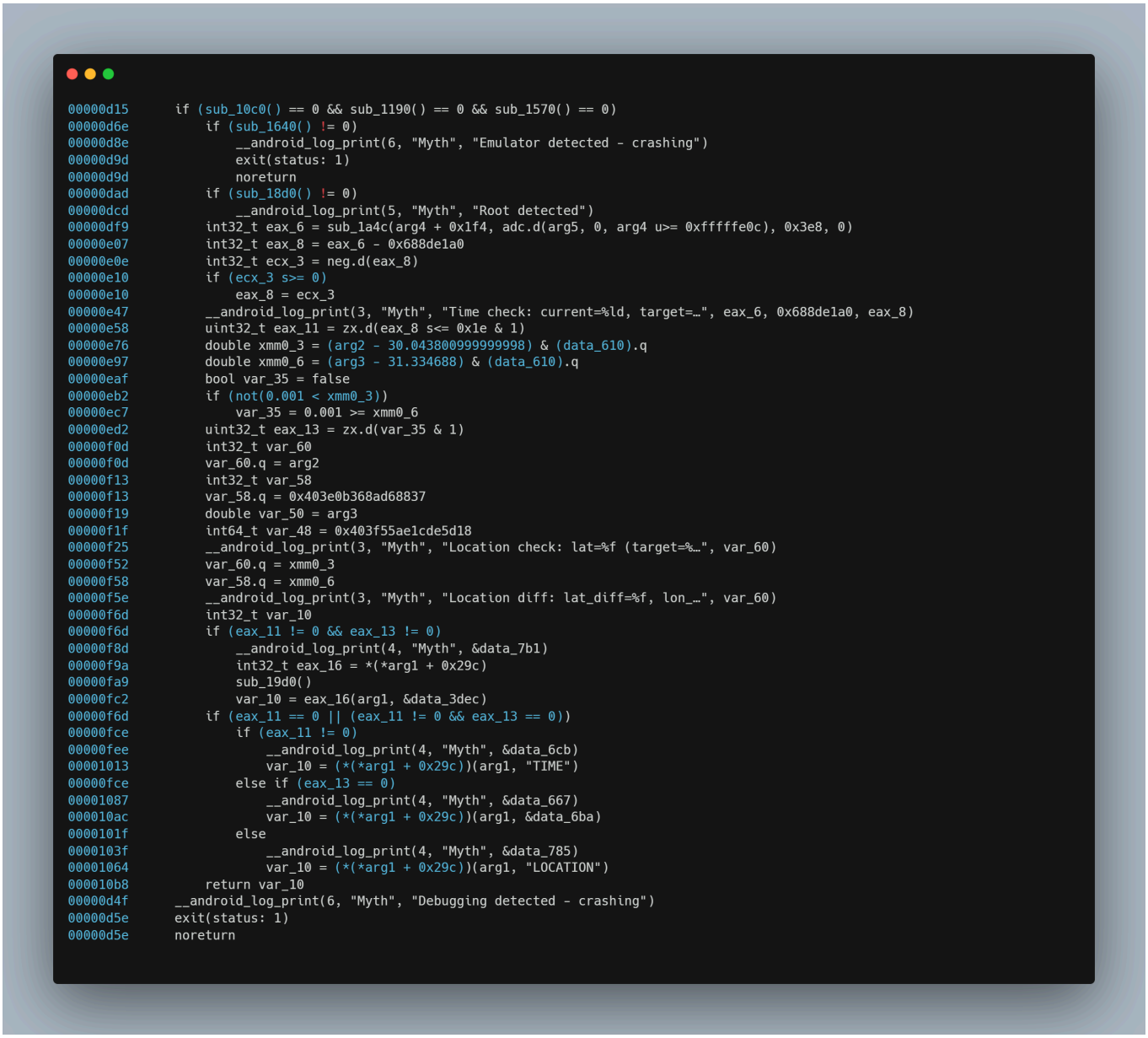
```
invoke-static {}, Ljava/lang/System;->currentTimeMillis()J
move-result-wide v3

const/16 v1, 0x7d0       # decimal 2000
int-to-long v5, v1       # v5 = 2000L

add-long/2addr v5, v3    # currentTime = now + 2000ms (2 seconds)
```

Native Library Analysis

getFlag Function Requirements

Analysis of the native library revealed that the `getFlag()` function requires:

1. Location coordinates matching:
   - Latitude ≈ 30.043801
   - Longitude ≈ 31.334688
2. Current time must be between 9:00 and 20:00 (local time)



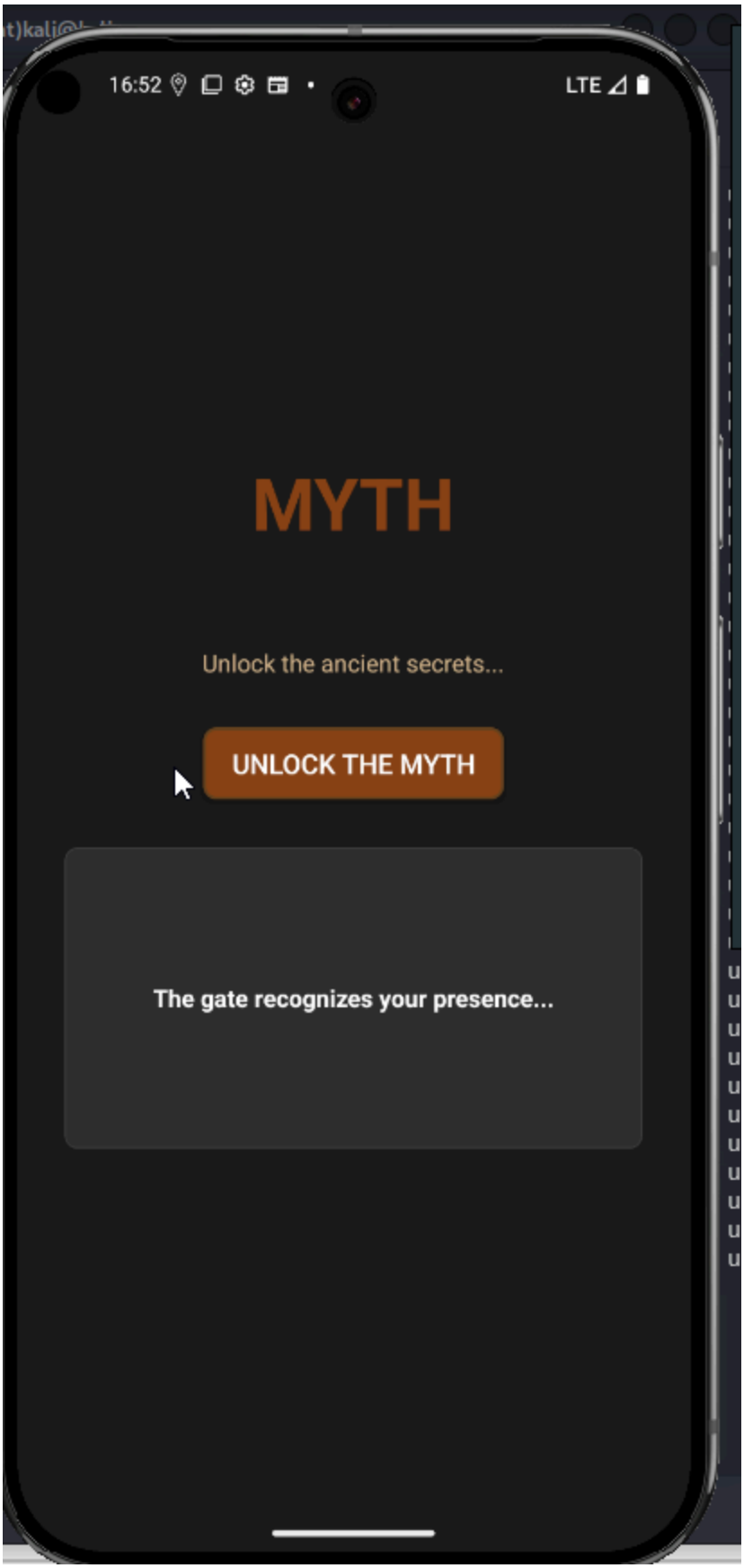
```
00000d15      if (sub_10c0() == 0 && sub_1190() == 0 && sub_1570() == 0)
00000d6e          if (sub_1640() != 0)
00000d8e              __android_log_print(6, "Myth", "Emulator detected - crashing")
00000d9d              exit(status: 1)
00000d9d              noreturn
00000dad          if (sub_18d0() != 0)
00000dcd              __android_log_print(5, "Myth", "Root detected")
00000df9          int32_t eax_6 = sub_1a4c(arg4 + 0x1f4, adc.d(arg5, 0, arg4 u>= 0xfffffe0c), 0x3e8, 0)
00000e07          int32_t eax_8 = eax_6 - 0x688de1a0
00000e0e          int32_t ecx_3 = neg.d(eax_8)
00000e10          if (ecx_3 s>= 0)
00000e10              eax_8 = ecx_3
00000e47          __android_log_print(3, "Myth", "Time check: current=%ld, target=…", eax_6, 0x688de1a0, eax_8)
00000e58          uint32_t eax_11 = zx.d(eax_8 s<= 0x1e & 1)
00000e76          double xmm0_3 = (arg2 - 30.043800999999998) & (data_610).q
00000e97          double xmm0_6 = (arg3 - 31.334688) & (data_610).q
00000eaf          bool var_35 = false
00000eb2          if (not(0.001 < xmm0_3))
00000ec7              var_35 = 0.001 >= xmm0_6
00000ed2          uint32_t eax_13 = zx.d(var_35 & 1)
00000f0d          int32_t var_60
00000f0d          var_60.q = arg2
00000f13          int32_t var_58
00000f13          var_58.q = 0x403e0b368ad68837
00000f19          double var_50 = arg3
00000f1f          int64_t var_48 = 0x403f55ae1cde5d18
00000f25          __android_log_print(3, "Myth", "Location check: lat=%f (target=%…", var_60)
00000f52          var_60.q = xmm0_3
00000f58          var_58.q = xmm0_6
00000f5e          __android_log_print(3, "Myth", "Location diff: lat_diff=%f, lon_…", var_60)
00000f6d          int32_t var_10
00000f6d          if (eax_11 != 0 && eax_13 != 0)
00000f8d              __android_log_print(4, "Myth", &data_7b1)
00000f9a              int32_t eax_16 = *(*arg1 + 0x29c)
00000fa9              sub_19d0()
00000fc2              var_10 = eax_16(arg1, &data_3dec)
00000f6d          if (eax_11 == 0 || (eax_11 != 0 && eax_13 == 0))
00000fce              if (eax_11 != 0)
00000fee                  __android_log_print(4, "Myth", &data_6cb)
00001013                  var_10 = (*(*arg1 + 0x29c))(arg1, "TIME")
00000fce              else if (eax_13 == 0)
00001087                  __android_log_print(4, "Myth", &data_667)
000010ac                  var_10 = (*(*arg1 + 0x29c))(arg1, &data_6ba)
0000101f              else
0000103f                  __android_log_print(4, "Myth", &data_785)
00001064                  var_10 = (*(*arg1 + 0x29c))(arg1, "LOCATION")
000010b8          return var_10
00000d4f      __android_log_print(6, "Myth", "Debugging detected - crashing")
00000d5e      exit(status: 1)
00000d5e      noreturn
```

So from the Java code and assembly i easily determined the location needs to be around these coordinates

Latitude ≈ 30.043801

Longitude ≈ 31.334688

and the only piece is missing now is the time



Flag Generation

When all conditions are met:

1. The application performs an XOR operation on the flag bytes
2. The operation is reversible with a simple Python script

Solution Approach

Bypassing Protections

Several options were available:

1. Patch the smali code to bypass protection checks
2. Use Frida hooks to bypass runtime detections
3. Reverse engineer the native library to understand the requirements

Final Solution

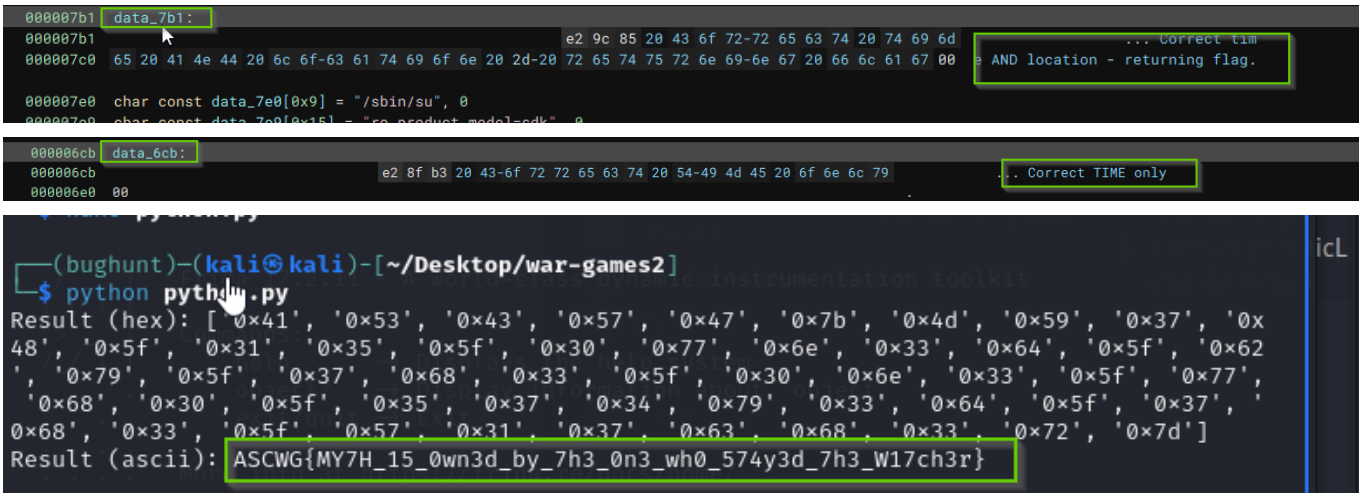
The most efficient approach was to:

1. Extract the hardcoded coordinates (30.043801, 31.334688)
2. Determine the time window requirement (9:00-20:00)
3. Reverse the XOR operation from the native library rather than attempting to satisfy all runtime conditions

```
000019d0  int32_t sub_19d0()

000019fa      for (int32_t i = 0; i u< 0x36; i = i + 1)
00001a1f          *(i + &data_3dec) = *(i + 0x8e1) ^ *(i + 0x917)
00001a3a      data_3e22 = 0
00001a4b      return &data_3dec
```

if any of parameters not true

```
var_58.q = xmm0_6
__android_log_print(3, "Myth", "Location diff: lat_diff=%f, lon_…  ", var_60)
int32_t var_10
if (eax_11 != 0 && eax_13 != 0)
    __android_log_print(4, "Myth", &data_7b1)
    int32_t eax_16 = *(*arg1 + 0x29c)
    sub_19d0()
    var_10 = eax_16(arg1, &data_3dec)
if (eax_11 == 0 || (eax_11 != 0 && eax_13 == 0))
    if (eax_11 != 0)
        __android_log_print(4, "Myth", &data_6cb)
        var_10 = (*(*arg1 + 0x29c))(arg1, "TIME")
    else if (eax_13 == 0)
        __android_log_print(4, "Myth", &data_667)
        var_10 = (*(*arg1 + 0x29c))(arg1, &data_6ba)
    else
        __android_log_print(4, "Myth", &data_785)
        var_10 = (*(*arg1 + 0x29c))(arg1, "LOCATION")
```



Conclusion

The challenge demonstrated several common mobile security concepts:

- Anti-analysis techniques (debugger, emulator, and Frida detection)
- Native code protection
- Environmental checks (location and time requirements)
- Obfuscation through simple cryptographic operations (XOR)

The solution highlighted the importance of:

1. Comprehensive static analysis
2. Understanding native code components
3. Choosing the most efficient path to solution (in this case, reversing the flag generation rather than satisfying all runtime checks)