2048 Game Hacking Walkthrough

What is Game Hacking?

Game hacking is the process of modifying a game's behavior to produce unintended effects. This can range from simple cosmetic changes to deep manipulation of game logic, physics, or scoring. Game hacking is used for fun, research, proof-of-concept security work, or in some cases, malicious purposes. As a cybersecurity engineer, understanding how games can be reverse-engineered helps you explore app vulnerabilities, code integrity, and tampering prevention techniques all valuable in real-world application security.

Why Hack 2048 Game

2048 is a simple puzzle game that makes it a perfect target for modding. It uses plain scoring logic which can be easily modified. For this hack, the goal was to change the way the score increased. Instead of only going up when tiles merged, the score should go up by double or quadruple even when nothing meaningful happens. This is a good example of logic tampering. It shows how you can bend a system to behave your way without crashing it.

TO GET FREE ACCESS TO CORELLIUM POST IN THE #mobile-game-hacking village and we will reach out to you for your email to create an account from you.

After creating a quick account, click on snapshots and CLONE the “DO-NOT-USE-CLICK-CLONE-MobileGameHackingSnapshot”.

Please do not USE the snapshot as this is the template others will be using. On the … click on clone.

Setting It All Up on Corellium

Device used: Android VM on Corellium

Game: 2048

Target: Change the score logic

Tools:

- Corellium console
- ADB
- APKTool
- VS Code
- Debug keystore for signing

Deep Dive: Technical

Understanding the Target – 2048 Game Architecture

2048 is a browser-based game ported to Android. It uses **HTML, CSS, and JavaScript** bundled inside a WebView. This structure is common in many mobile games and apps using frameworks like Cordova or PhoneGap. The game logic, especially the score handling, lives in JavaScript files typically inside the /assets/www/js/ folder.

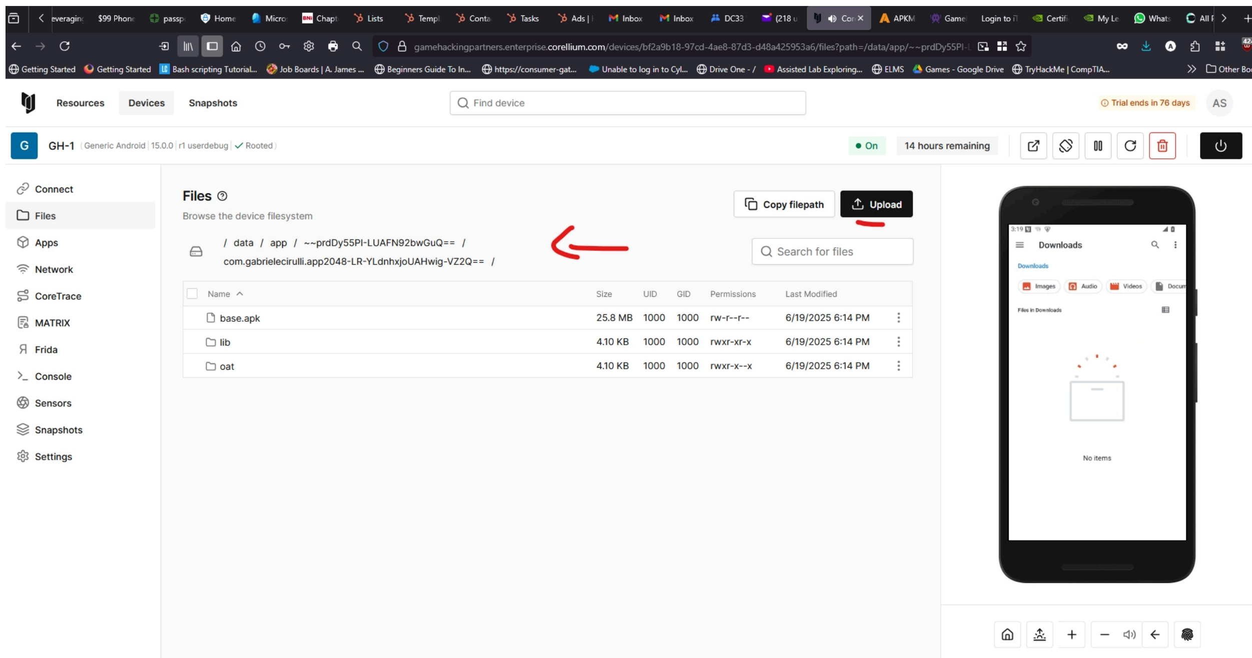
In this project, the logic that handles tile movement and score calculation was stored in a file like main.js or game_manager.js.

We reviewed the code and identified the function responsible for updating the score. It looked something like this: **this.score += points;** We changed it to: **this.score += points * 4;** Or in some cases added lines that update the score even if no tile merged. This was enough to break the integrity of the game logic.

Step-by-Step Guide

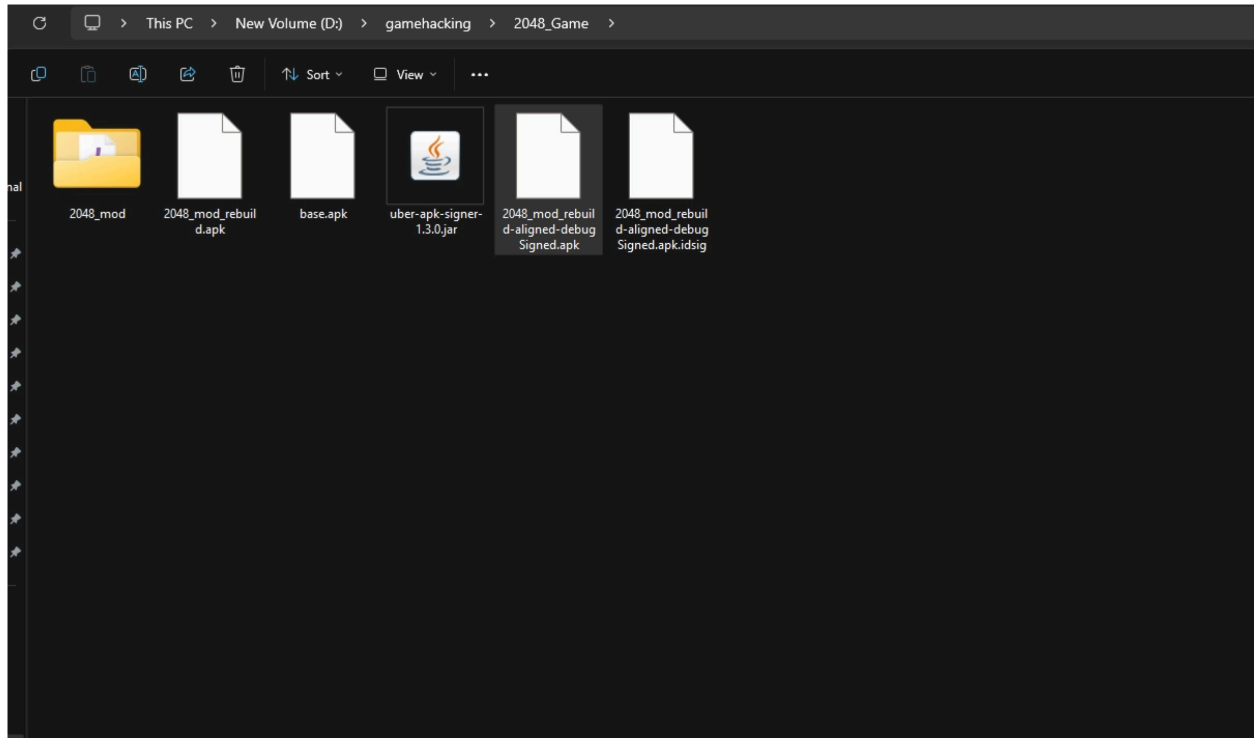
Step 1: Download and Decompile the Game

The 2048 game was downloaded and placed into a working folder. APKTool was used to decompile the APK so we could access its assets, JavaScript, and Small files.



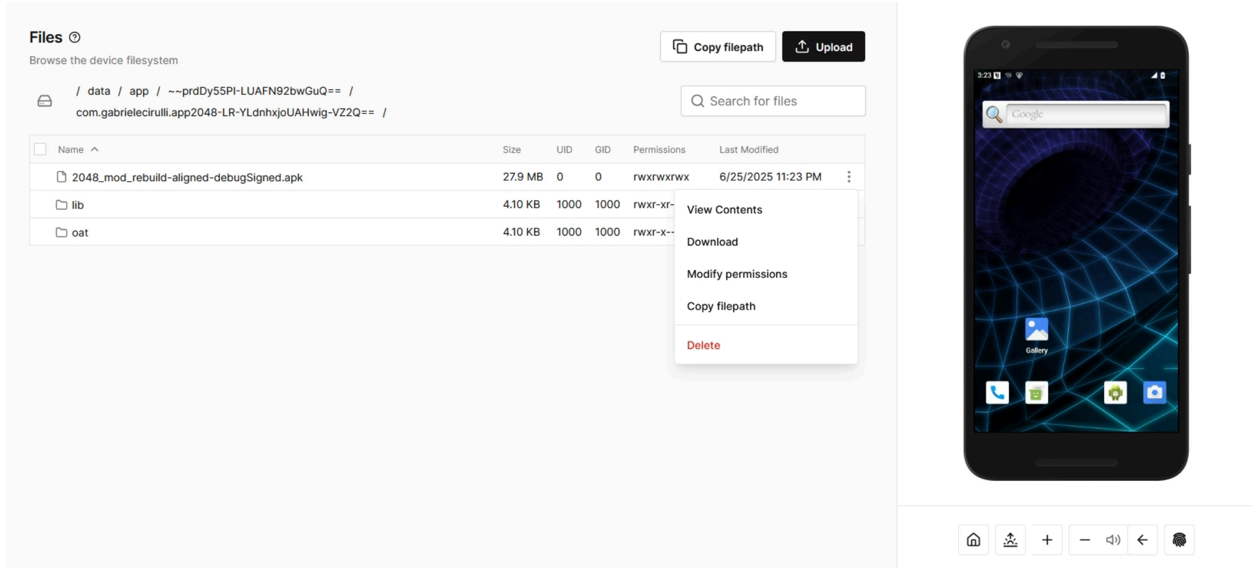
Step 2: Locating and Downloading the Game APK

We downloaded the 2048 APK from APKMirror and moved it into a working folder on the Windows system. The APK file was downloaded to the Windows system so that it could be decompiled and modified.



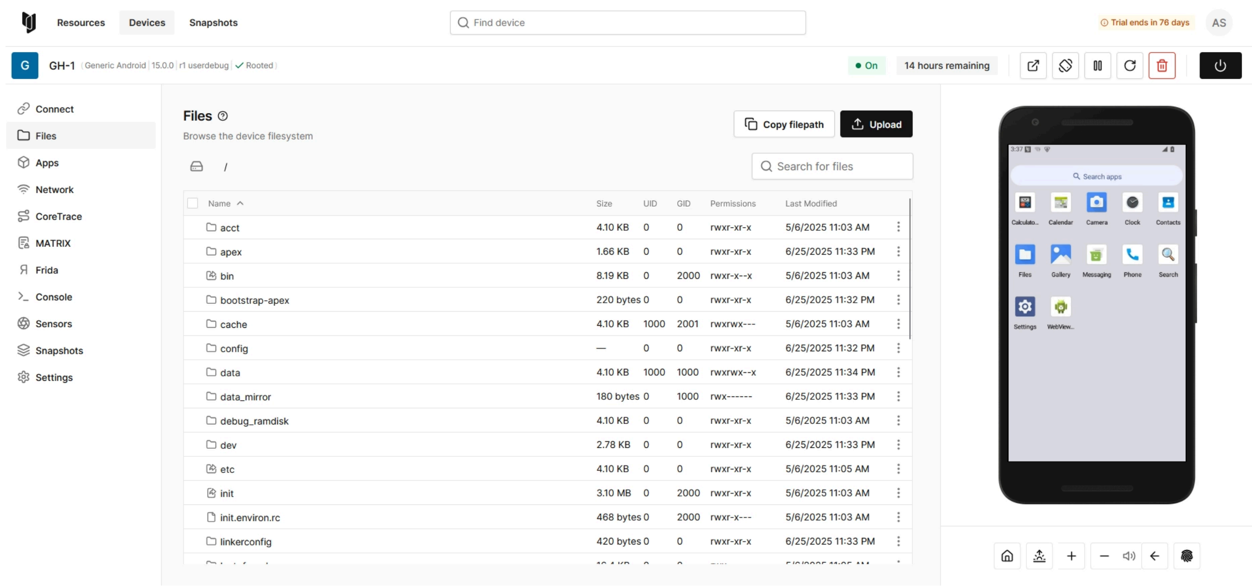
Step 3: Use APKTool to Decompile the APK

We used the apktool JAR file from the Windows Command Prompt to decompile the APK. This extracted the app's internal file structure and gave access to the JavaScript files used for logic. It displays the output folder after APKTool finishes the decompilation.



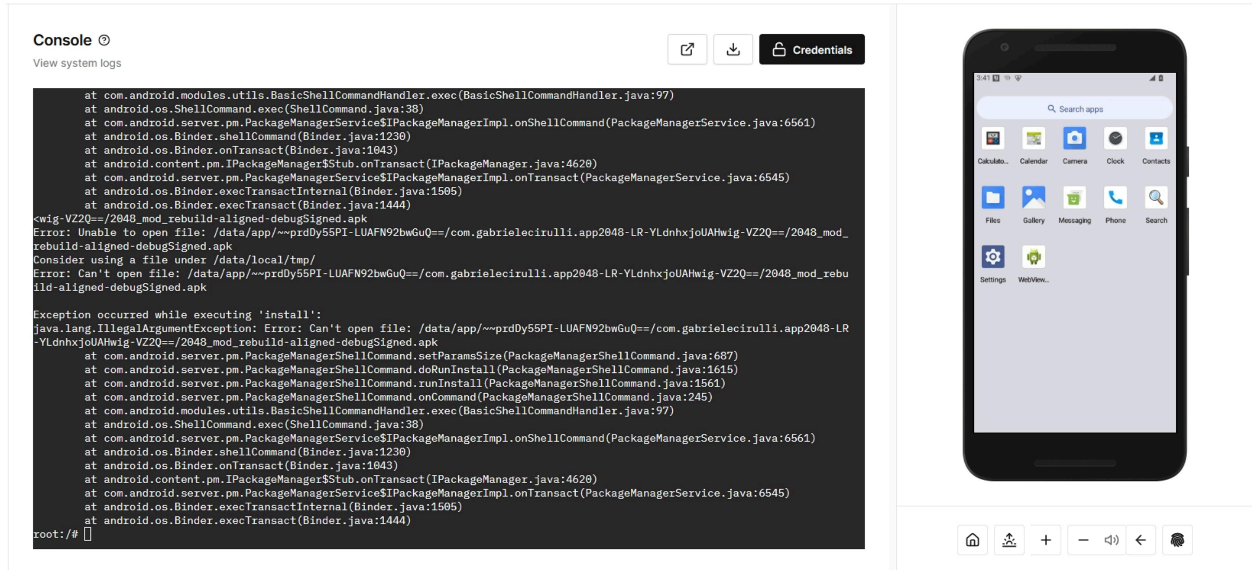
Step 4: Navigating to JavaScript Files

We went inside the decompiled folders and found the main.js file where the game's score logic was defined. This file was the main target for modifying the behavior. It shows the file path where the JavaScript files are located.



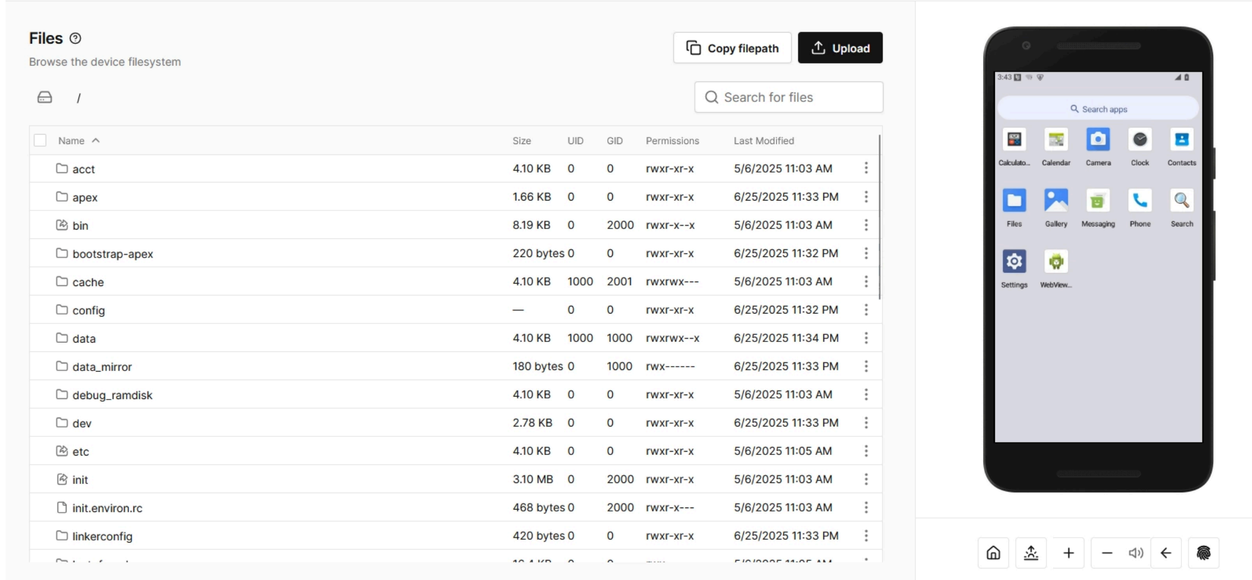
Step 5: Editing the Score Logic

We opened the file in VS Code and found the scoring function. By analyzing the logic, we added or modified the code so that the score gets multiplied. The changes were saved carefully to avoid breaking the app. Highlights the lines in main.js that were changed to multiply the score.



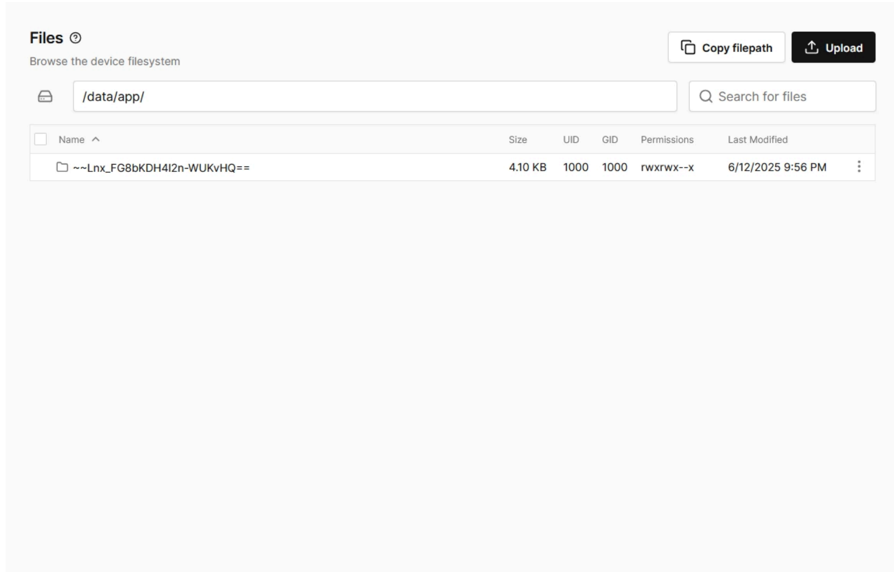
Step 6: Rebuilding the APK

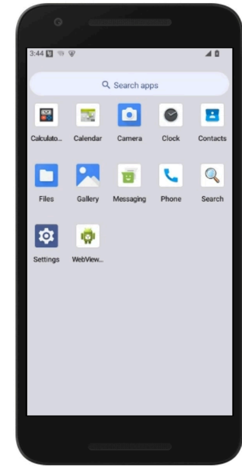
After making changes, we repacked the app using APKTool to rebuild the APK with the new logic. It shows the output folder after building the APK again.



Step 7: Signing the APK

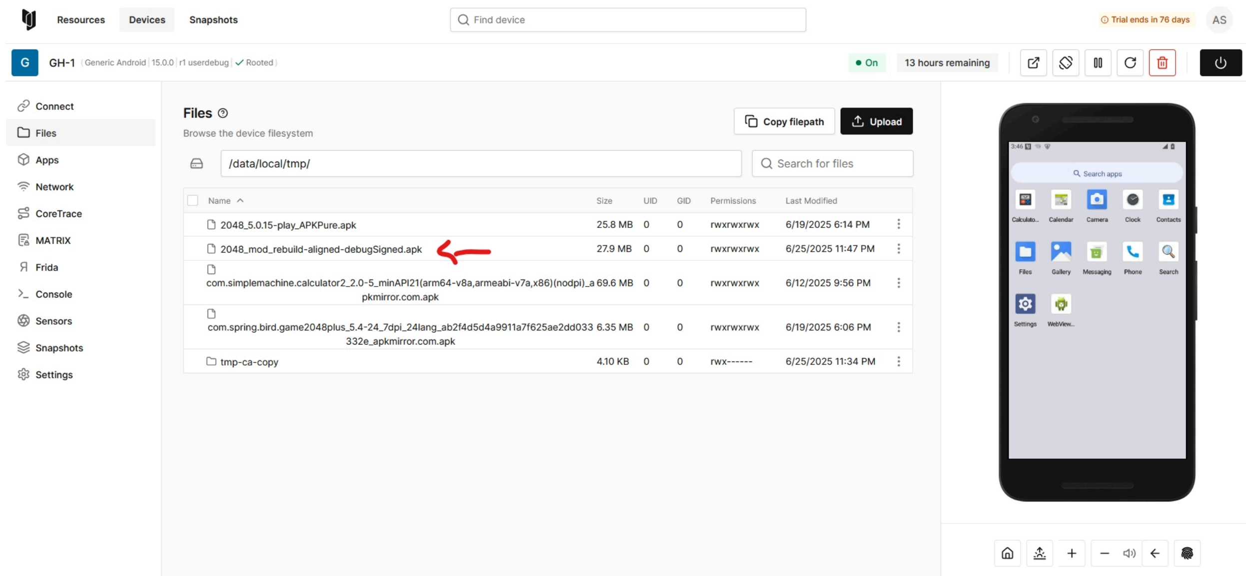
We signed the APK with a debug keystore. This was necessary to install it on the Corellium Android phone. Unsigned APKs are usually rejected by the system. It displays the terminal after the signing process using jarsigner or apksigner.

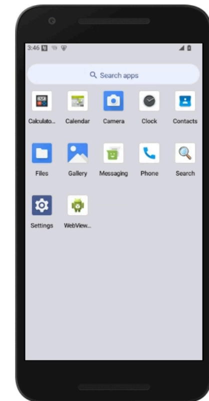




Step 8: Installing the Modified APK

We uploaded the modified APK to Corellium and installed it using the ADB shell or file manager inside Corellium's UI. It shows the APK being installed successfully.





Step 9: Running the Game

After installation, the app ran normally. But this time, the score started increasing in unusual ways. Even swiping without merging tiles caused the score to rise. It captures the game with a much higher score than possible under normal logic.

Console
View system logs

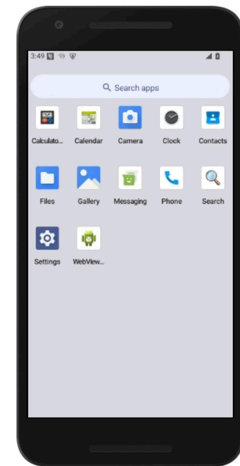
```
<d-aligned-debugSigned.apk" /data/local/tmp/2048.apk
cp: bad '/data/app/~~prdDy55PI-LUAFN92bwGuQ==/com.gabrielecirulli.app2048-LR-YLdnhxjoUAHwig-VZ2Q==/2048_mod_rebuild-aligned-debugSigned.apk': No such file or directory
</local/tmp/2048_mod_rebuild-aligned-debugSigned.apk
Failure [INSTALL_FAILED_UPDATE_INCOMPATIBLE: Existing package com.gabrielecirulli.app2048 signatures do not match newer version; ignoring!]
root:/#
```

Step 10: Validating the Hack

We verified that the score logic had been altered correctly. The game responded exactly as expected based on the changes made in the JavaScript code. It displays proof of the score increasing rapidly without meaningful moves.

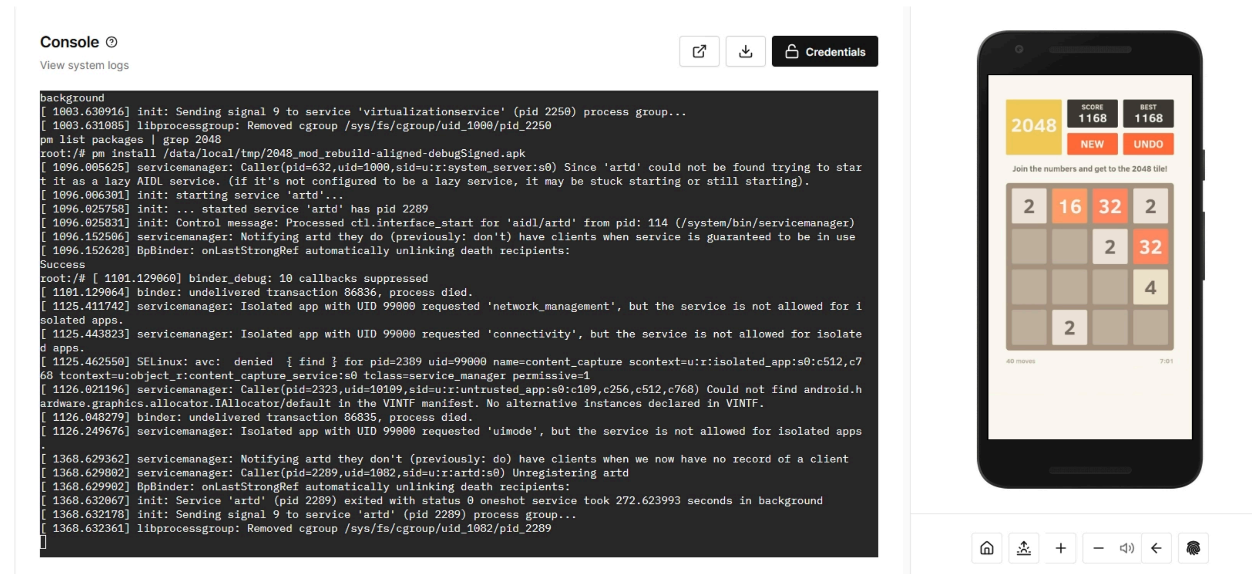
Console
View system logs

```
[  989.176087] binder: 632:715 transaction async to 1612:0 failed 86845/29189/-3, size 148-0 line 3544
[  989.176280] binder_alloc: 1291: binder_alloc_buf, no vma
[  989.177761] binder: 632:763 ioctl c00c620f 6e627f7fcc returned -22
[  989.178248] binder: 632:763 ioctl c00c620f 6e627f7fcc returned -22
root:/# [  989.183518] binder_alloc: 1538: binder_alloc_buf, no vma
[  989.183637] binder_alloc: 1556: binder_alloc_buf, no vma
[  989.309926] servicemanager: Caller(pid=632,uid=1000,sid=u:r:system_server:s0) Since 'android.system.virtualizationmaintenance' could not be found trying to start it as a lazy AIDL service. (if it's not configured to be a lazy service, it may be stuck starting or still starting).
[  989.310669] init: starting service 'virtualizationservice'...
[  989.312253] init: ... started service 'virtualizationservice' has pid 2250
[  989.312336] init: Control message: Processed ctl.interface_start for 'aidl/android.system.virtualizationmaintenance' from pid: 114 (/system/bin/servicemanager)
[  989.676301] servicemanager: Caller(pid=2250,uid=1000,sid=u:r:virtualizationservice:s0) Could not find android.hardware.security.secretkeeper.ISecretkeeper/default in the VINTF manifest. No alternative instances declared in VINTF.
[  989.676965] servicemanager: Caller(pid=2250,uid=1000,sid=u:r:virtualizationservice:s0) Could not find android.hardware.security.keymint.IRemotelyProvisionedComponent/avf in the VINTF manifest. VINTF declared instances: default.
[  989.677307] servicemanager: Notifying android.system.virtualizationmaintenance they do (previously: don't) have clients when service is guaranteed to be in use
[  989.681213] BpBinder: onLastStrongRef automatically unlinking death recipients:
[ 1003.629317] servicemanager: Notifying android.system.virtualizationmaintenance they don't (previously: do) have clients when we now have no record of a client
[ 1003.629627] servicemanager: Caller(pid=2250,uid=1000,sid=u:r:virtualizationservice:s0) Unregistering android.system.virtualizationmaintenance
[ 1003.629744] BpBinder: onLastStrongRef automatically unlinking death recipients:
[ 1003.629842] servicemanager: Caller(pid=2250,uid=1000,sid=u:r:virtualizationservice:s0) Unregistering android.system.virtualizationservice
[ 1003.629936] BpBinder: onLastStrongRef automatically unlinking death recipients:
[ 1003.630820] init: Service 'virtualizationservice' (pid 2250) exited with status 0 oneshot service took 14.319000 seconds in background
[ 1003.630916] init: Sending signal 9 to service 'virtualizationservice' (pid 2250) process group...
[ 1003.631085] libprocessgroup: Removed cgroup /sys/fs/cgroup/uid_1000/pid_2250
```



Step 11: Outcome and Final Score

The final score was much higher than a legitimate session. This validated the mod and demonstrated how simple logic hacks can compromise a game's integrity. Final score screen showing successful scoreboard manipulation.



Why This Matters in Cybersecurity

This exercise was not just about increasing a score. It is about understanding the risks of poorly protected code and how attackers or modders can manipulate app behavior. As cybersecurity engineers, knowing how attackers think helps us build stronger defenses. Game hacking is an entry point into reverse engineering, obfuscation bypassing, and runtime patching skills. These are all valuable when protecting Android apps from tampering or piracy.

That section discusses why understanding game hacking is important in cybersecurity. It's the perfect transition into **advanced hacking techniques like Frida**.

Runtime Hooking with FRIDA

After demonstrating how static logic manipulation through JavaScript editing can impact the game behavior, I took it further by exploring runtime hooking using **Frida**. Unlike static modification, Frida allows us to **inject code at runtime** into the memory of a running app. This gives more flexibility to change the behavior without modifying the APK.

What is Frida?

Frida is a powerful dynamic instrumentation toolkit that lets us hook into functions, modify return values, or even overwrite memory behavior on the fly. It works great on Android (especially in platforms like Corellium) and is widely used in security research and reverse engineering.

Goal

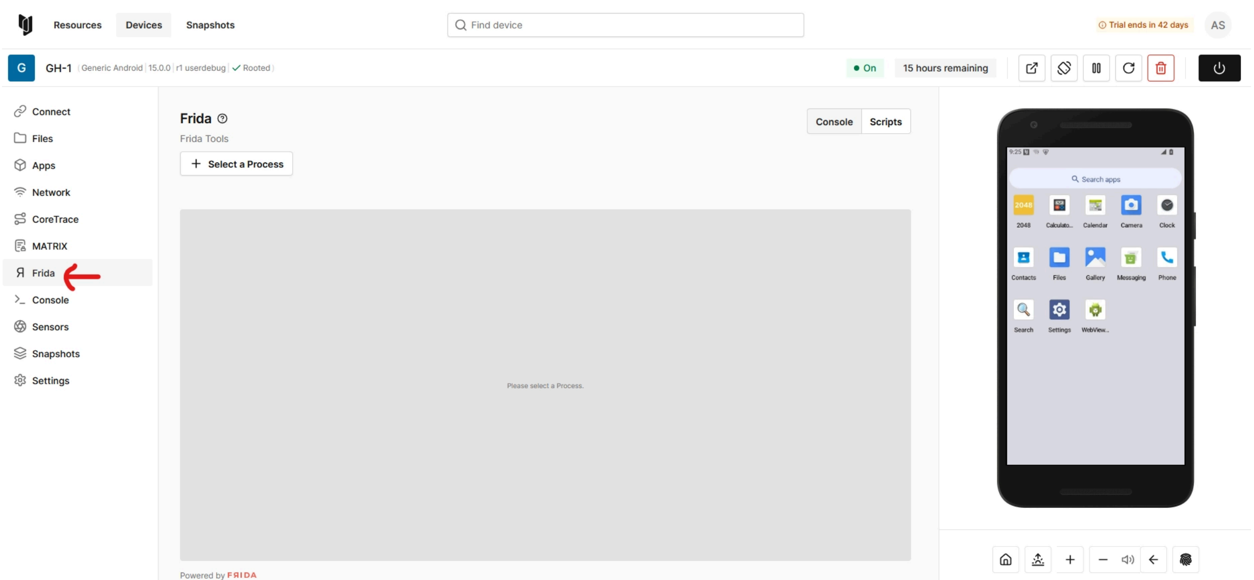
My goal was to hook into the score function of the 2048 Android game and **multiply the score in real-time**. This helped prove how runtime patching works without rebuilding the APK.

Objective & Setup Summary

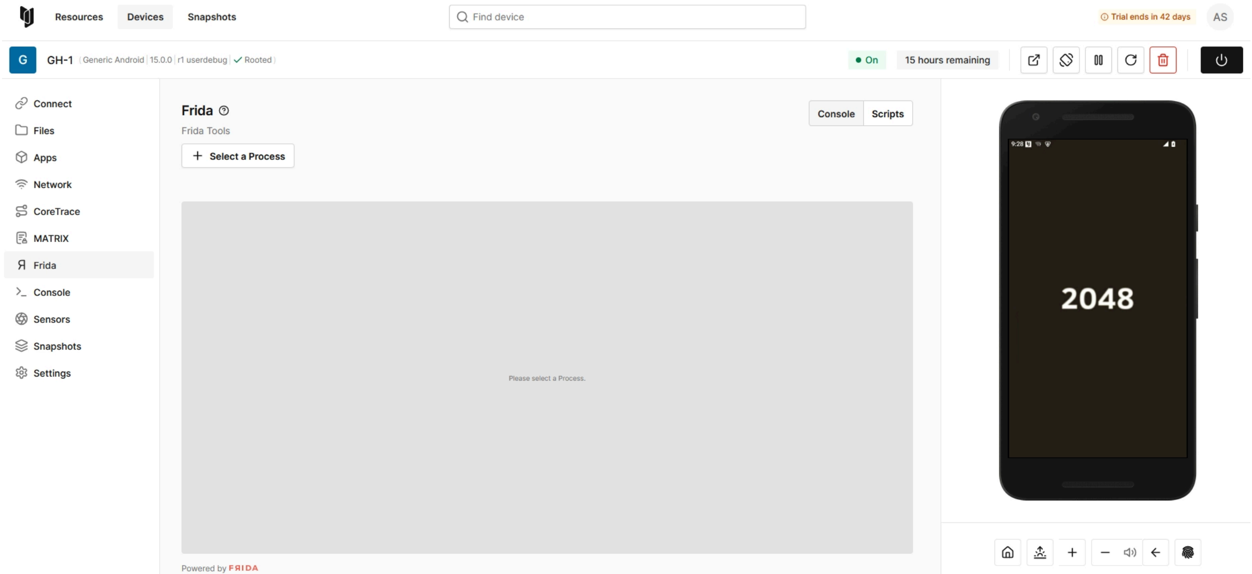
We are targeting the score calculation logic of the 2048 game. The goal is to hook the function that updates the score and multiply the value (quadruple it). All of this is done live without rebuilding or resigning the APK.

Step-by-Step Breakdown

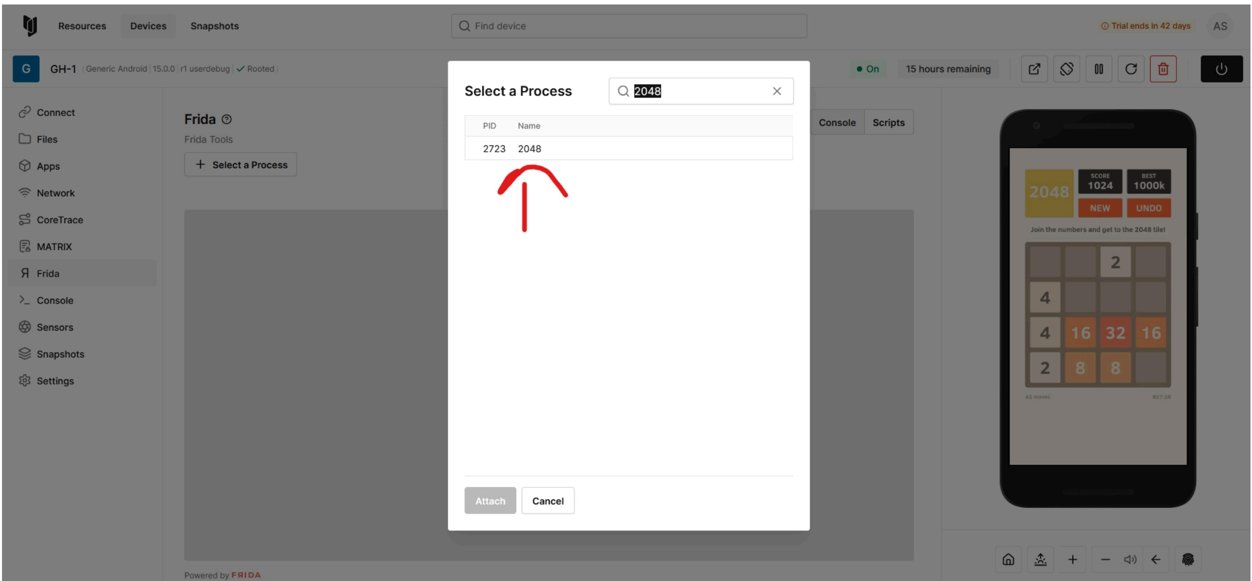
1. Launched Frida from the built-in Corellium terminal



2. Loaded the 2048 game APK



Resources
Devices
Snapshots
Find device
Trial ends in 42 days
AS
G
GH-1
Generic Android | 15.0.0 | r1 userdebug | Rooted
On
15 hours remaining
Connect
Files
Apps
Network
CoreTrace
MATRIX
Frida
Console
Sensors
Snapshots
Settings
Frida
Frida Tools
Select a Process
Console
Scripts
Please select a Process.
Powered by FRIDA
2048



Resources
Devices
Snapshots
Find device
Trial ends in 42 days
AS
G
GH-1
Generic Android | 15.0.0 | r1 userdebug | Rooted
On
15 hours remaining
Connect
Files
Apps
Network
CoreTrace
MATRIX
Frida
Console
Sensors
Snapshots
Settings
Frida
Frida Tools
Select a Process
Console
Scripts
Select a Process
2048
PID
Name
2723
2048
Attach
Cancel
Powered by FRIDA
2048
SCORE
1024
BEST
1000k
NEW
UNDO
Join the numbers and get to the 2048 tile!

3. Navigated to the **scripts** tab inside Frida

4. Edited and saved a script file named `hook_java.js` with the following code:

Java.perform(function () { Java.enumerateLoadedClasses({ onMatch: function (className) { if (className.indexOf("com.gabrielecirulli.app2048") !== -1 && (className.toLowerCase().includes("score") |className.toLowerCase().includes("game"))) { console.log("[+] Found Class: " + className); } }, onComplete: function () { console.log("[*] Class search complete."); } }); });

5. Loaded and confirmed Frida hook.

Script execution: %load /data/vendor/corellium/frida/scripts/hook_java.js

Result

Once the Frida hook was successfully loaded using the script, the game logic was modified in real time. We specifically targeted the part of the game responsible for updating the score. By altering that logic, every point increase in the game was multiplied by four instead of the usual increment. As a result, I was able to score much faster than normal gameplay would allow. For example, a single tile merge that would usually give 4 points now resulted in 16. This showed that our Frida hook worked exactly as intended it intercepted and altered the in-game logic dynamically. This simple tweak made the game behave in a completely unfair way, proving how easily game mechanics can be manipulated if the underlying logic isn't properly secured. It was a hands-on demonstration of why mobile apps, especially games, need to implement strong protections against runtime modification.

Summary

This project was not just about hacking a game but understanding how vulnerable mobile apps can be when logic is exposed and not protected. By modifying a single line in JavaScript, we turned a fair game into a rigged one. This shows how even casual or non-malicious attackers can manipulate logic when proper security practices are not followed.

This kind of exercise builds skills in:

- Reverse engineering
- Secure mobile app development
- Threat modeling
- Building countermeasures against tampering and piracy

Defense Recommendations

To protect against such attacks in a real production app:

1. **Obfuscate the JavaScript files** – Use tools like JavaScript Obfuscator to make code harder to understand
2. **Minify and hash assets** – Prevent editing and add integrity checks
3. **Use certificate pinning and APK signature verification** – Detect tampering
4. **Validate game logic on the server-side** – Do not trust client calculations for scores or purchases
5. **Use anti-tampering SDKs** – Tools like DexGuard, AppSealing, or Licel can protect against runtime patching or rooted devices
6. **Detect rooted or emulated environments** – Prevent apps from running on platforms like Corellium or Magisk-rooted phones