WHITE PAPER

A deep dive into mobile android apps privacy security controls

Penetration testing report and outcomes of Its mobile apps protection

SecureLayer7
Time and Again, Securing you

Contents

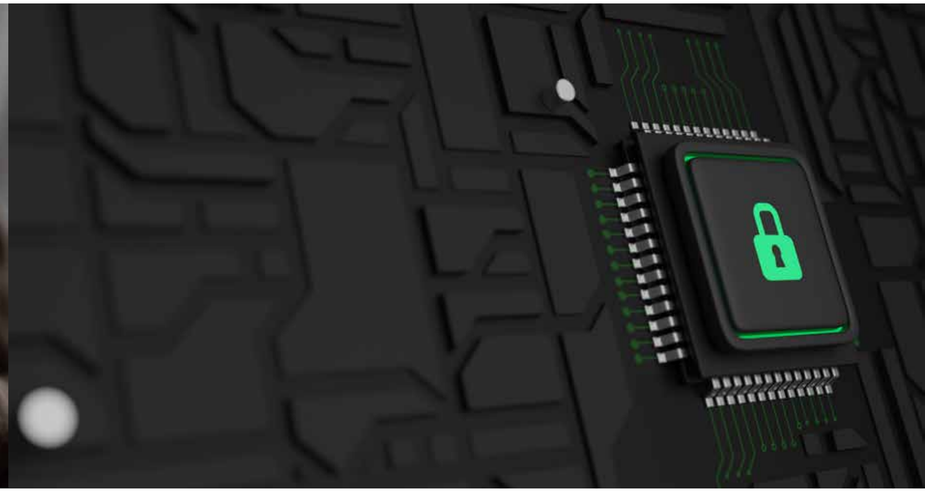
Executive Summary

In the digital landscape, Android mobile applications are increasingly handling sensitive and personally identifiable information, making the implementation of robust privacy security controls a critical imperative.

This necessitates a focused approach towards mitigating a spectrum of vulnerabilities, ranging from unauthorized screen overlays that mislead users, to keyloggers surreptitiously recording keystrokes. Risks inherent in copy/paste functionality and screen mirroring also demand attention, as they pose threats to data confidentiality.

Equally paramount is the protection against advanced rooting techniques like Magisk, and the prevention of remote desktop control attacks and accessibility malware, which exploit system features for nefarious purposes. Robust memory protection strategies are essential to thwart unauthorized data access, emphasizing the multifaceted nature of privacy challenges in Android environments. Each vulnerability presents a distinct avenue for potential privacy breaches, underlining the urgency for comprehensive security controls in Android applications.

SecureLayer7, which is a Austin, Texas based cybersecurity service and product company, reviewed sample Android apps, equipped with Appdome's suite of mobile app privacy protections. This report meticulously documents the findings from the review, focusing on the efficacy and robustness of the privacy control mechanisms within the evaluated sample apps.



Scope and Details

The principal objective of this undertaking was a meticulous scrutiny of the privacy control mechanisms integrated within a selected Android application, in conjunction with the Appdome privacy control implementations. This was a time-boxed investigation of Appdome's mobile apps privacy controls.

SecureLayer7 employed the Fusion Appdome control panel for the purpose of uploading and compiling the Android app's executable file, replete with privacy controls. This process was followed by an in-depth investigation into the potential avenues for circumventing the protective measures established within the sample Android applications. The investigation's central focus was on identifying and overcoming the mobile app privacy protections provided by Appdome.

01 Overlay
02 Keylogger
03 Copy/Paste
04 Screen Mirroring
05 Anti-Root/Magisk
06 Anti-RDC
07 Memory Protection

Test Methodology

In this section, SecureLayer7's methodology is meticulously detailed, describing the approach taken in examining the various components of the app's privacy controls. The documentation provides insight into the thoroughness of the testing coverage and delves into how each aspect of the privacy controls was scrutinized.

Additionally, it offers a deeper analysis of specific areas, particularly focusing on those aspects where significant protection bypasses were not initially evident.

The initial phase of the assignment involved a detailed review of the app's scope, followed by the compilation of the sample Android app's executable file utilizing the Appdome's privacy controls. A key feature of Appdome, highlighted during this process, is its no-code platform capability, which streamlines and simplifies the deployment of such security measures. The SecureLayer7 team adeptly conducted a side-by-side comparative analysis of various compiled files from the application. This process involved reversing the Android app to investigate the privacy control gaps by understanding the app's privacy and security architecture.

The SecureLayer7 team commenced testing for privacy protection bypasses with an emphasis on Anti-App Screen Sharing.

The approach included diverse techniques, extending beyond third-party tools to in-depth code analysis, usage of custom Frida scripts, and other runtime modifications. Initially, the open-source tool scrcpy was employed for screen mirroring, which successfully mirrored the device's screen but displayed a black screen for the protected app. Attempts to bypass this via different video encoding formats in scrcpy and error flag analysis did not yield successful bypasses.

Subsequently, SecureLayer7 employed various tools and frameworks, including LSPOSED and Frida scripts, to modify runtime function values responsible for screen sharing protection. By disabling the FLAG_SECURE using LSPOSED, it was observed that Appdome's Anti-App Screen Sharing protection was bypassed, allowing screen mirroring despite the implemented protection. Conversely, users must activate additional features to enhance protection against screen sharing vulnerabilities.

In the assessment of Copy/Paste protection bypass, SecureLayer7 utilized a multi-faceted approach, incorporating code level analysis and runtime manipulation techniques. The initial test using a browser with Copy/Paste Prevention enabled revealed effective restriction messages. Subsequent static analysis and searches within the codebase for related strings yielded no significant findings.

Despite employing tools like JADX-GUI for decompiling and analyzing the source code, obfuscation levels hindered successful de-obfuscation. Additionally, efforts to locate encryption keys, as suggested in Appdome's documentation for encrypted clipboard data, were fruitless. Advanced methods like using ADB for clipboard service analysis and employing Silent Clipboard Reader also only revealed encrypted data, underscoring the robustness of the implemented privacy feature.

In evaluating the Block App Overlay Attacks protection, SecureLayer7 adopted a multifaceted approach, involving code-level analysis and runtime manipulation techniques. Tests using Tapjacker and Tapjacking-ExportedActivity applications demonstrated effective detection of overlays by the and Tapjacking-ExportedActivity applications demonstrated effective detection of overlays by the protected app, resulting in self-closure. Despite static code analysis and custom Frida script attempts, obfuscation challenges hindered deeper insights.

Additionally, tests with an application designed to check FLAG_WINDOW_IS_PARTIALLY_OBSCURED flags revealed specific vulnerabilities in partially obscured scenarios.

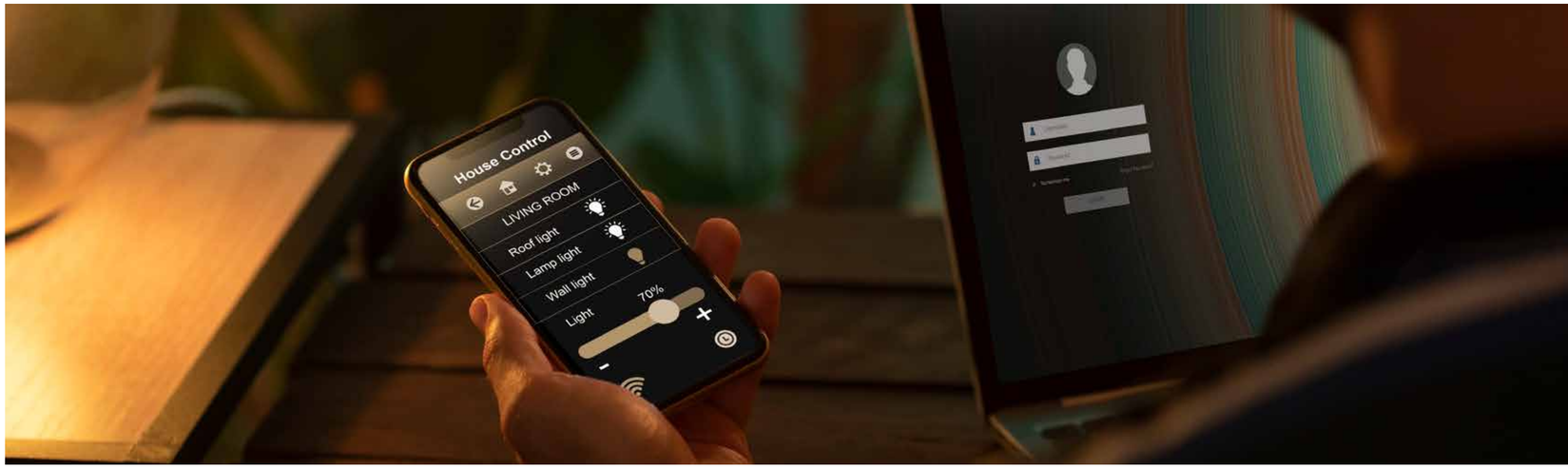
Test Methodology

In assessing the Anti Remote Desktop Control (RDC) protection, SecureLayer7 applied a blend of techniques including code analysis and runtime manipulation. Tests with TeamViewer indicated effective RDC detection by the protected app, displaying a specific warning. Subsequent code analysis did not reveal direct references to TeamViewer or similar apps. Further tests with RustDesk, which uses Accessibility services, initially bypassed the protection but were thwarted by enabling 'Block Suspicious Accessibility Services, confirming the effectiveness of this additional security layer.

In the keylogging protection bypass assessment, SecureLayer7 employed a methodology encompassing code analysis and runtime techniques. Using LokiBoard, a keylogger, revealed effective detection by the protected app, which displayed a specific warning against untrusted keyboards. Further analysis did not find the exact toast message in the codebase. Attempts to input via LokiBoard were blocked, and no evidence of captured keystrokes was found in the expected storage directory, indicating robust keylogging protection.

In assessing Memory Protection, SecureLayer7 applied a methodology that included code-level analysis and other techniques. The use of the open-source tool MobSF, a security assessment framework, facilitated the static analysis of the protected application. The analysis revealed robust memory protection mechanisms, evidenced by the presence of Stack Canary and Full RELRO in all shared libraries. Further search for hardcoded keys or secrets within the decompiled source code yielded no results, indicating a lack of easily identifiable vulnerabilities in data encryption.

In the Anti-Root bypass assessment, SecureLayer7's methodology included code analysis and runtime techniques. The BrowserAndroid application's Root Detection was tested, revealing a toast notification on rooted devices. Static analysis with Jadx-gui4 did not find related strings in the code. Attempts using Objection and Frida gadget to patch the app resulted in it not progressing past the icon screen. Additional tests for emulator detection bypass using various tools and scripts were unsuccessful due to strong obfuscation in the app's code, highlighting the robustness of the Anti-Root protection.

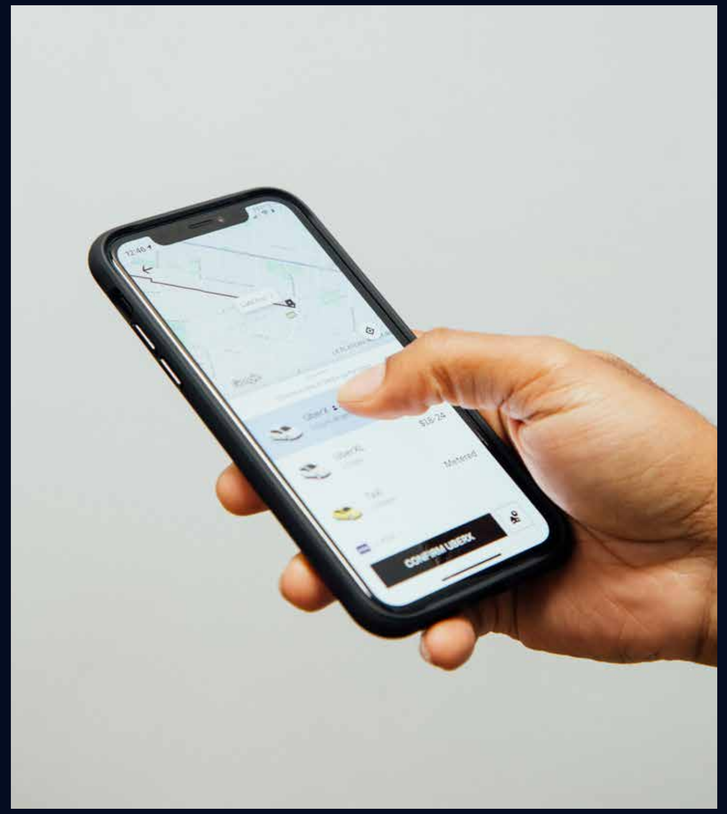


Observations

The Observation section of the white paper provides a status of Appdome's various privacy protection measures in an Android environment.

Testcase	Description	Result
Appdome Overlay Protection Bypass in Android Sample App	Explored the effectiveness of Appdome's Overlay Protection in an Android app, revealing potential bypass methods through advanced testing techniques. It adds additional layer of protection using Appdome fusion panel block suspicious accessibility services for workaround	Not Bypassed
Appdome Keyloggers Protection Bypass in Android Sample App	Assessed the resilience of Appdome's Keylogger Protection in an Android environment, identifying possible circumvention strategies.	Not Bypassed
Appdome Screen Mirroring Safeguards Bypass in Android Sample App	Analysed the robustness of Appdome's Screen Mirroring Protection, testing for vulnerabilities that could allow unauthorised mirroring.	Not Bypassed
Appdome Anti-Root Magisk Protection Bypass in Android Sample App	Investigated the strength of Appdome's Anti-Root Magisk Protection, focusing on potential bypass methods in a rooted Android scenario.	Not Bypassed

Testcase	Description	Result
Appdome Remote Desktop Control (RDC) Protection Bypass in Android App	Examined the effectiveness of Appdome's RDC Protection against remote access attempts, identifying possible loopholes. It adds additional layer of protection using Appdome fusion panel enable detecting hooking framework for workaround	It adds additional layer of protection using
Appdome Accessibility Malware Protection Bypass in Android App	Tested the capability of Appdome's Accessibility Malware Protection to withstand sophisticated malware attacks exploiting accessibility features.	Not Bypassed
Appdome Memory Protection Protection Bypass in Android App	Investigated data within the RAM, especially while the app is actively in use, ensuring that sensitive information remains secure from unauthorized access or manipulation.	Not Bypassed



Conclusion

In conclusion, SecureLayer7's assessment of Appdome's privacy protections in an Android environment yielded mixed results. While the majority of protections, such as Keylogger, Anti-Root Magisk, Accessibility Malware, and Memory Protection, were not bypassed, indicating their effectiveness, vulnerabilities were identified in the Remote Desktop Control (RDC) and Screen Mirroring protections.

These findings suggest that while Appdome's mechanisms are robust against common attacks and effectively hinder advanced analysis, there are still areas that could be exploited by determined attackers. Despite this, the time and resources required to fully bypass these protections are significant, making them sufficient for deterring most threat models. This level of security demonstrates Appdome's commitment to thwarting common bypass methods and underscores the need for continuous enhancement of security measures.

Appendix

Attack Narrative

This document delineates the methodologies employed in circumventing Appdome's privacy controls within a sample app, with the objective of extracting sensitive data.
The purpose was to gauge the effectiveness of Appdome's protections against common attack strategies and to furnish the Appdome team with insights into potential bypasses.

The following sections detail the scope and extent of the testing coverage achieved. It elaborates on the methodologies applied in scrutinizing different components under the purview of Appdome's Privacy Protection bypass.

In reverse engineering endeavors, it's crucial to establish clear objectives. Given the impracticality of dissecting every function in an application, the focus is directed towards pivotal areas linked to root detection. This involves strategic instrumentation and investigation into segments that are likely integral to the targeted functionalities.

The examination began with the intent to identify and bypass the Anti-App Screen Sharing protection. This involved a multifaceted approach, not just limited to the usage of third-party tools and hardware but also including an in-depth analysis of base code level changes.

Custom Frida scripts and various other scripts were developed and employed to facilitate runtime alterations and challenge the existing privacy protection mechanisms.

The initial phase of testing utilized "scrcpy", an open-source utility tool designed for screen sharing of Android devices connected to a computer with USB debugging enabled. Upon execution, SecureLayer7 noted that while "scrcpy" mirrored the Android device as intended, it encountered a significant limitation when applied to the app with Anti-App Screen Sharing protection, resulting in a blank black screen. Efforts to circumvent this protection included experimenting with different video encoding formats within "scrcpy". Despite these attempts, the protection remained effective, showing no signs of being compromised.

Further investigative measures involved re-launching "scrcpy" with an emphasis on error detection, using flags like ERROR to analyze if any specific errors were generated when initiating the privacy-protected application. However, this approach did not yield any notable errors. Similar observations were recorded when deploying other flags, such as WARN, indicating the robustness of the protection against these testing methods.

Code Snippet 1: Monitoring filesystem access

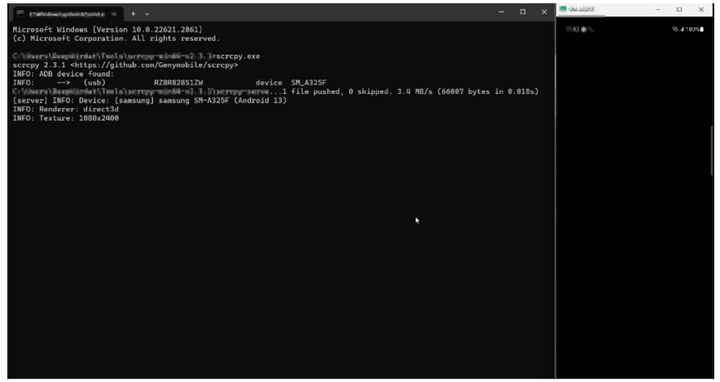


Figure #1: Bypassing using scrcpy open-source tool

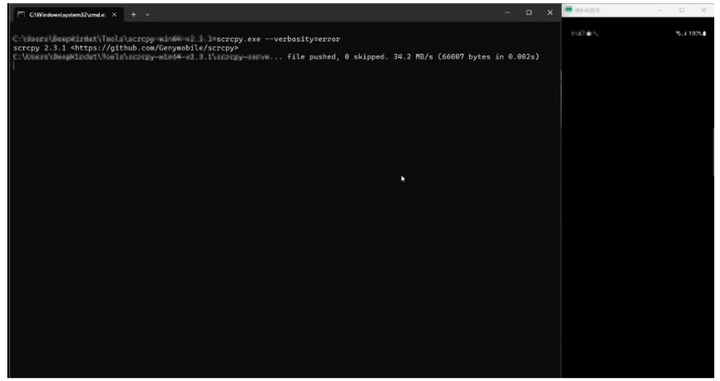


Figure #2: Bypassing using scrcpy open-source tool with verbosity flag

Appendix

Subsequently, the team employed ApowerMirror, another third-party application for screen sharing. This tool, akin to scrcpy, successfully mirrored screens of standard applications. However, it encountered a similar limitation with the Anti-App Screen Sharing protected application, displaying only a black screen. This further affirmed the effectiveness of the Appdome's screen sharing privacy protection in safeguarding against external screen mirroring attempts.

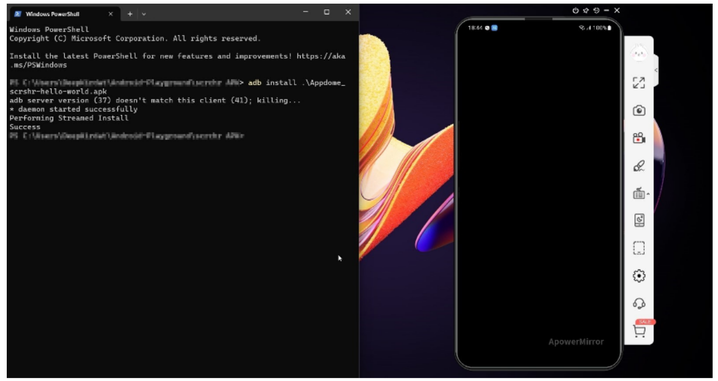


Figure #3: Bypassing using 3rd party screen mirroring tool ApowerMirror

Recognizing the necessity to delve deeper, SecureLayer7 shifted focus to dissecting the custom code within the Appdome application. The objective was to pinpoint specific functions and classes responsible for the Anti-App Screen Sharing protection.

For this purpose, SecureLayer7 harnessed open-source utilities APKTOOL and JADX-GUI. APKTOOL was instrumental in decompiling the zipped APK, facilitating the analysis of the embedded custom code. Complementing this, the meld tool was utilized to execute a diff operation, enabling an effective comparison across various code files.

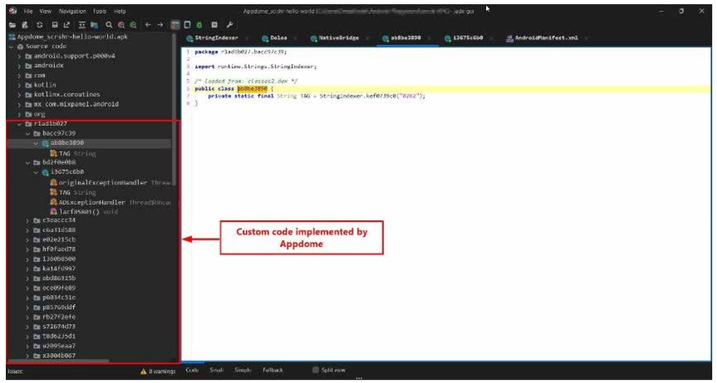


Figure #4: Custom code within the application visible after decompiling the APK

The analysis using apktool, Jadx-gui, and meld involved a comparative study between two APK versions: one without Anti-App Screen protection (APK-1) and the other with the protection implemented (APK-2). This comparison revealed that upon implementing the Anti-App Screen protection, Appdome introduces a custom, obfuscated Java code, characterized by alpha-numeric filenames and function names like r1ad1b027. An example of this can be seen in the AndroidManifest.xml file, which illustrates one of many instances where Appdome employs custom code to prevent app screen sharing.

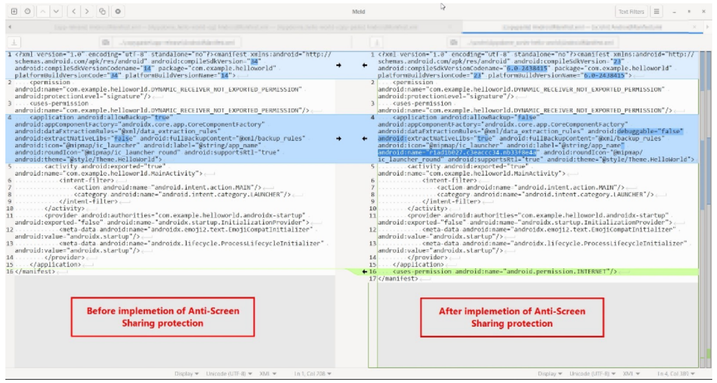


Figure #5: Diff of AndroidManifest.xml with and without Appdome's Privacy Protection

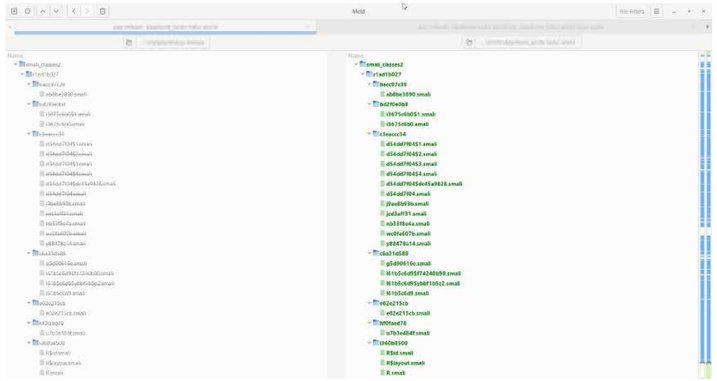


Figure #6: Diff of AndroidManifest.xml with and without Appdome's Privacy Protection

Appendix

In their pursuit to identify the specific class and function responsible for the Anti-App Screen protection, SecureLayer7 embarked on an in-depth code analysis. The presence of obfuscated Java code added a significant challenge. Efforts were made to de-obfuscate this code using multiple open-source tools, including JDO, Java-deobfuscate, and Java-deobfuscate-gui. Despite these attempts, the obfuscation proved resilient, as none of the tools were successful in translating the obfuscated Java code into a plain text, human-readable format, highlighting the complexity and robustness of Appdome's code protection mechanisms.

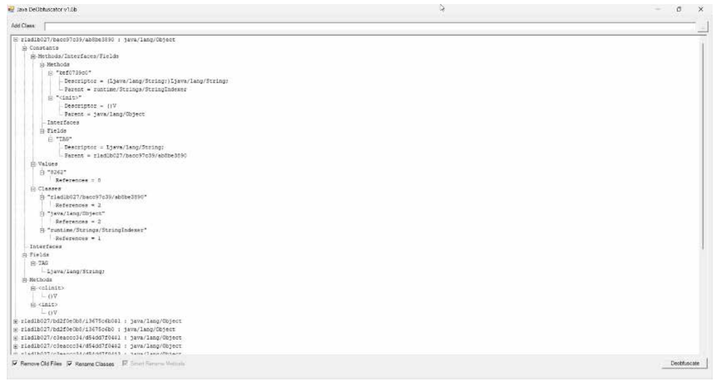


Figure #7: Obfuscated Code from Protection

Progressing further in the analysis, SecureLayer7 employed an array of tools and frameworks, including LSPOSED and various Frida scripts, tailored to modify runtime function values associated with screen sharing protection.

This technique aimed to potentially bypass these protections. The initial tests using LSPOSED to disable the FLAG_SECURE setting revealed a vulnerability in Appdome's Anti-App Screen Sharing protection, allowing screen mirroring despite the protection being active. This finding underscores the need for continual enhancements in security measures to address evolving bypass techniques.

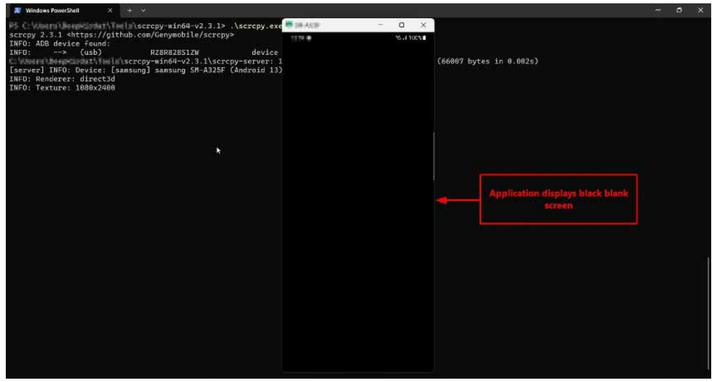


Figure #8: Black screen before enabling the LSPOSED's FLAG_SECURE module

When LSPOSED's FLAG_SECURE is activated, it results in the bypassing of the protection. However, the Appdome team suggests an additional security layer: blocking suspicious accessibility services during the app's compilation in the Appdome fusion panel. This recommendation indicates an approach for reinforcing the app's defense mechanisms against such bypass techniques.

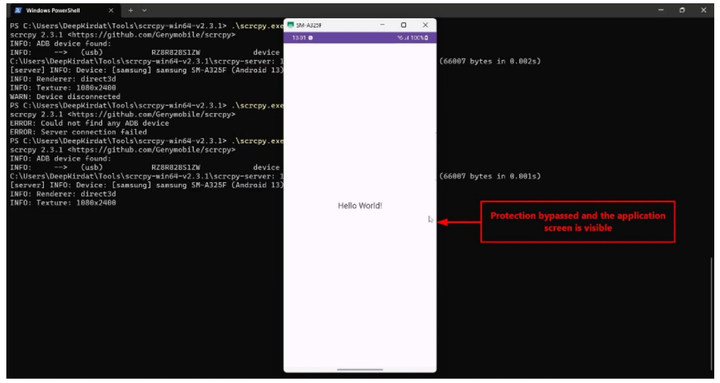


Figure #9: Protection bypassed after enabling the LSPOSED's FLAG_SECURE module

In the assessment of the Copy/Paste Prevention feature, SecureLayer7 utilized a comprehensive methodology encompassing both code level analysis and runtime manipulation. This approach involved comparing the application's code before and after the implementation of Copy/Paste Prevention to identify any changes or newly implemented security flags.

The testing began with the utilization of a Browser Android application in which Copy/Paste Prevention was enabled. Upon launching the app and attempting to copy the text 'test' entered into the URL address field, it was observed that the clipboard displayed a message indicating the prevention of copying and pasting from the app. This initial test demonstrated the effective functioning of the Copy/Paste Prevention feature.

Appendix

To further understand the underlying mechanisms, the team conducted static analysis, searching for the specific string displayed by the clipboard throughout the application's code base. However, this analysis did not yield matching results, suggesting the complexity and effectiveness of the implemented Copy/Paste Prevention measures.

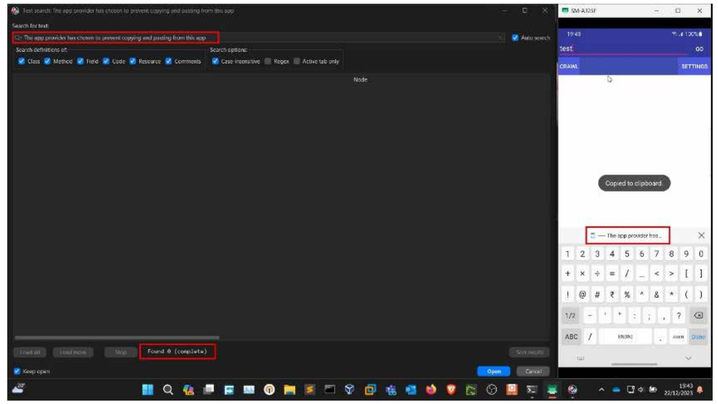


Figure #10: Looking for Copied Text

In the pursuit of identifying the functions and classes behind the Copy/Paste Prevention feature, SecureLayer7 employed JADX-GUI for decompiling the APK to analyze the source code. Despite this effort, the high level of obfuscation in the code posed a significant challenge. Subsequent attempts using various open-source tools like JDO and Java-deobfuscate also proved unfruitful, as these tools either malfunctioned or were incapable of de-obfuscating the complex code structure, hindering further in-depth analysis of the underlying mechanisms.

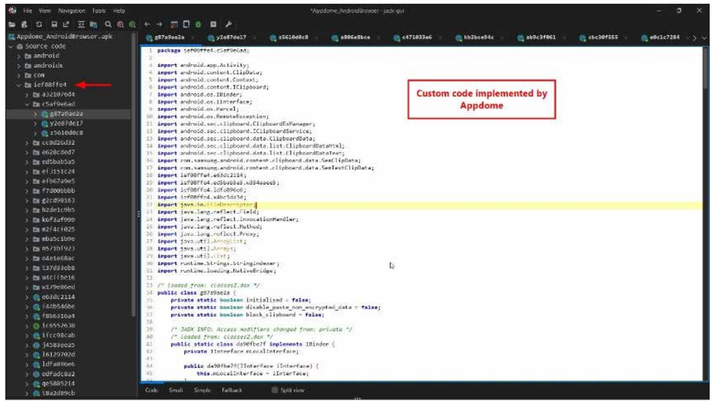


Figure #11: Custom code within the application visible after decompiling the APK

In line with Appdome's documentation on Copy/Paste protection, the team explored the possibility of data encryption during runtime, especially focusing on scenarios where copied data gets encrypted. The objective was to locate any encryption keys within the app's source code that might be used for encrypting and decrypting the copied data. Despite thorough analysis, this search did not result in the discovery of any encryption key references in the code.

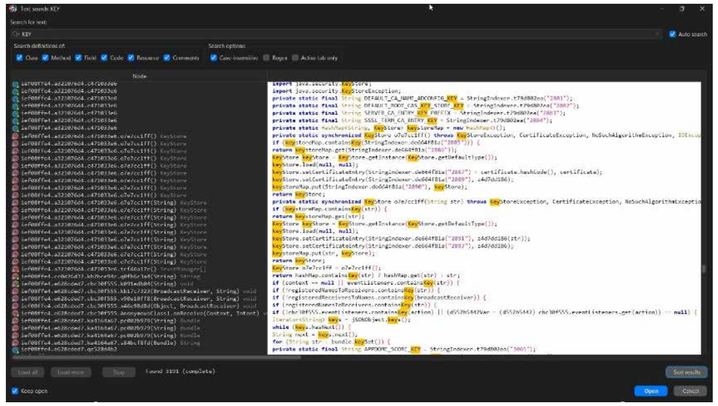


Figure #11: Identifying the encryption keys within the source code of the decompiled APK

Utilizing ADB, the team gained shell access to further investigate the clipboard service's functionality. This allowed for an examination of the clipboard service's contents, revealing user and application IDs, along with the process name. An attempt was made to dump the clipboard service data while having superuser access. Despite this, the analysis did not reveal any copied text in the dumped data, indicating the effectiveness of the copy/paste protection in securing clipboard contents.

Appendix

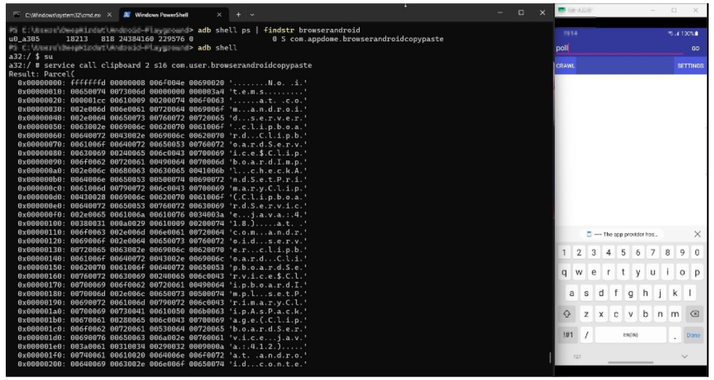


Figure #12: Extracting the clipboard contents via ADB

In the continuation of the assessment, SecureLayer7 utilized an open-source APK named Silent Clipboard Reader, designed to replicate and display clipboard data. The observation revealed that even with this tool, the copied data from the clipboard was in an encrypted format. This finding further underscores the effectiveness of the copy/paste protection in safeguarding data against unauthorized access or replication.

Code Snippet 1: Monitoring filesystem access

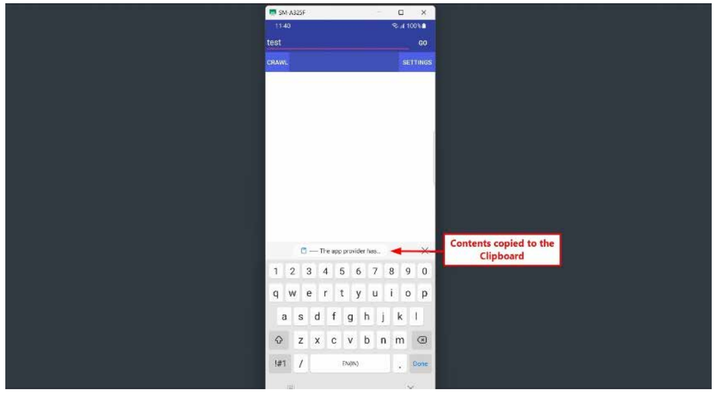


Figure #13: bypass using Silent Clipboard Reader -1

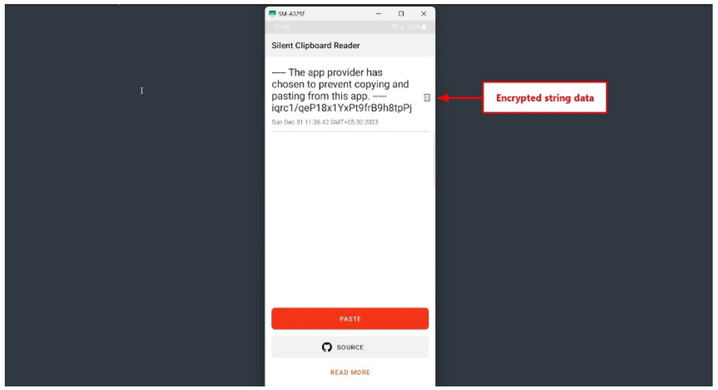


Figure #14: bypass using Silent Clipboard Reader -2

In assessing the Block App Overlay Attacks protection, SecureLayer7 employed a comprehensive approach, which included code-level analysis, observation of differences, and other runtime manipulation techniques. The testing utilized the BrowserAndroid application. An open-source Android application called Tapjacker, used for demonstrating Android tapjacking attacks, was a key tool in this assessment.

With Tapjacker, the user selects the application package and activity to display an overlay screen. When Tapjacker was used on the BrowserAndroid app with Appdome's overlay protection enabled (com.appdome.browserandroidoverlay and its MainActivity), it triggered a protective response. The app displayed a toast message indicating the detection of screen overlay usage and subsequently closed, demonstrating the effectiveness of Appdome's overlay attack protection.

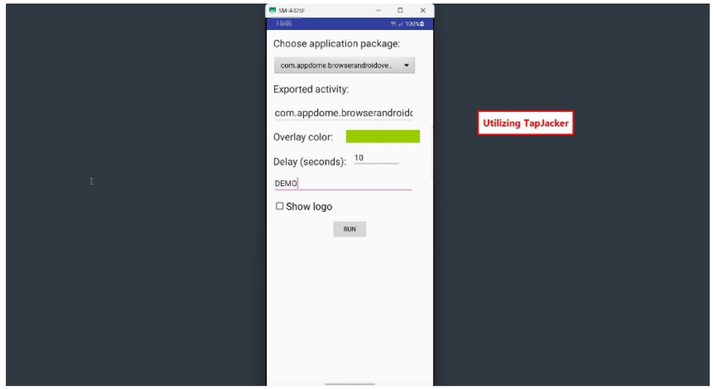


Figure #15: bypassing using TapJacker tool -1

Appendix

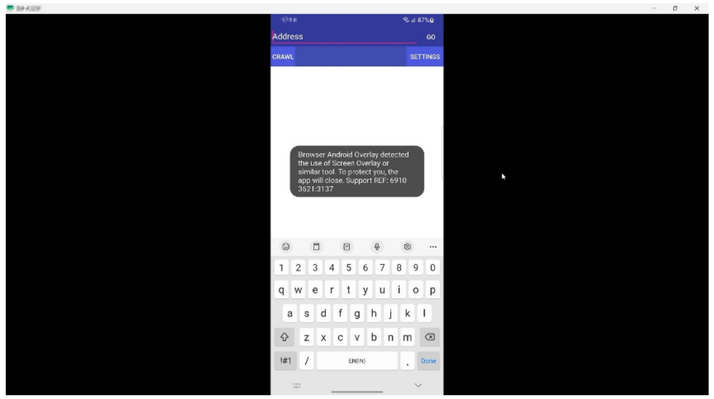


Figure #15: bypassing using TapJacker tool -2

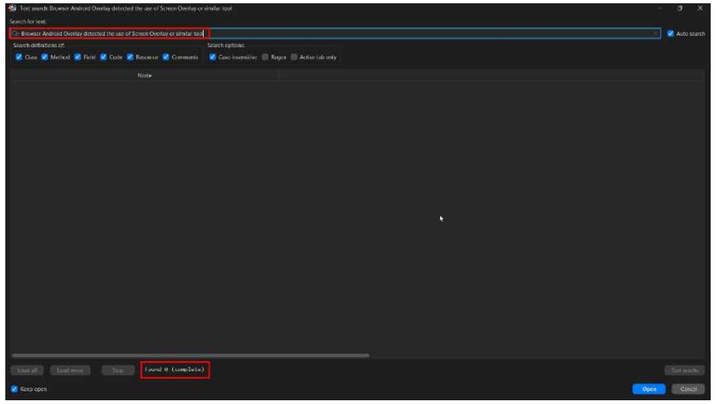


Figure #16: Searched for the displayed toast via static analysis

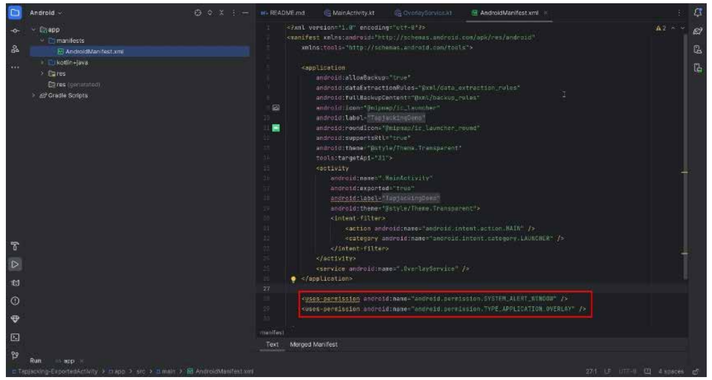


Figure #17: Bypass using Tapjacking-ExportedActivity open-source tool

Following the overlay attack testing, SecureLayer7 conducted a static analysis by examining the decompiled APK source code of the application with implemented protection. This analysis aimed to locate the specific toast message triggered during the overlay protection mechanism. However, no corresponding results were found in the code base.

SecureLayer7 utilized another open-source Android application named Tapjacking-ExportedActivity for further testing. This application leverages SYSTEM_ALERT_WINDOW and TYPE_APPLICATION_OVERLAY permissions to create an overlay.

To test the Tapjacking-ExportedActivity application, the SecureLayer7 team input specific identifiers: the package name "com.appdome.browserandroidoverlay" and the main activity name "com.appdome.browserandroidoverlay. MainActivity".

Appendix

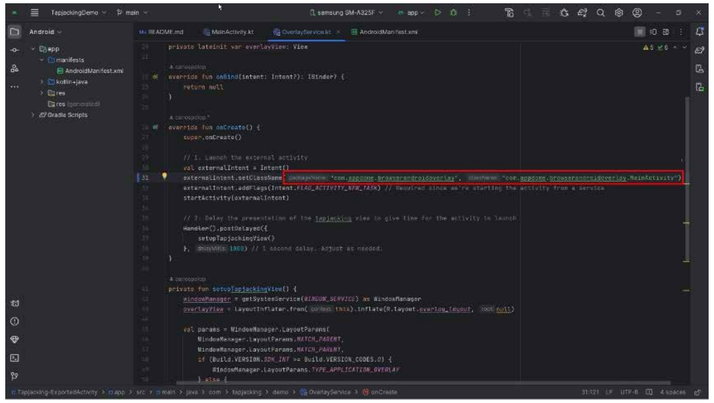


Figure #18: Bypass using Tapjacking-ExportedActivity open-source tool

Upon executing the Tapjacking-ExportedActivity application, it successfully created an overlay on the app equipped with Appdome's protection. When interaction occurred with this overlay screen, the protected application displayed a toast message similar to previous observations and then proceeded to close itself.

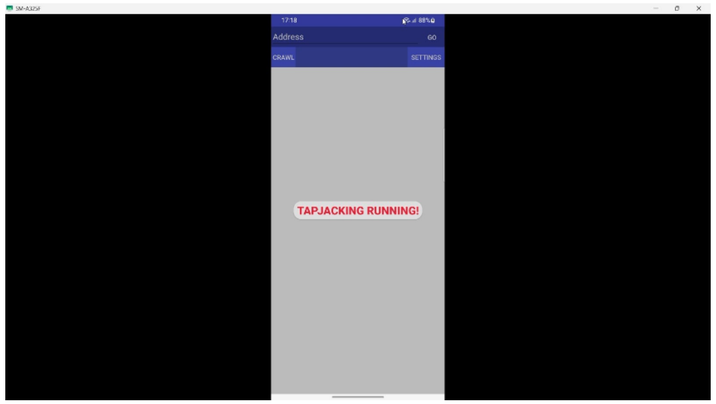


Figure #19: Bypass using Tapjacking-ExportedActivity open-source tool

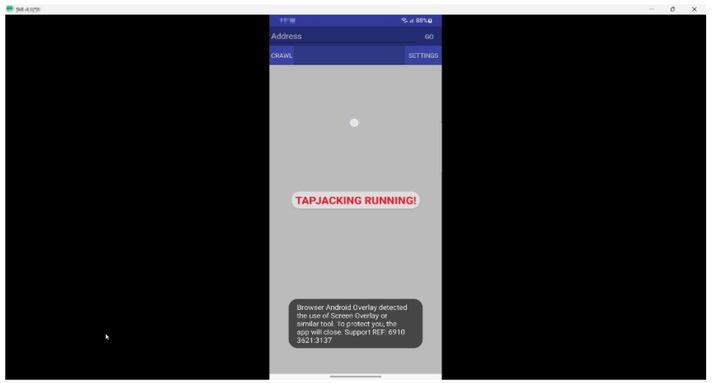


Figure #20: Bypass using Tapjacking-ExportedActivity open-source tool

In their continued analysis, SecureLayer7 delved into the static code of the application and identified a specific class, s79af6adb.fe51a4e9a.fe51a4e9a, which appears to be linked to the implementation of the Overlay protection.

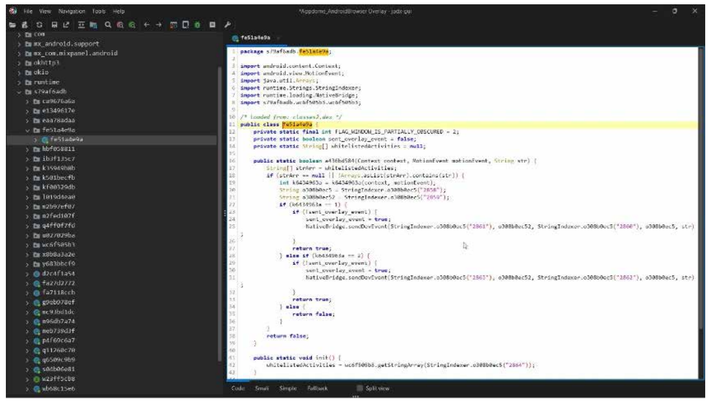


Figure #21: Located the code responsible for protection

In an effort to further understand the Overlay protection, SecureLayer7 developed a custom Frida script intended to hook into specific functions and return a Boolean value, thereby manipulating the app's runtime behavior. However, they encountered a challenge as the function did not hook properly. This issue was attributed to the obfuscation present in the source code, which added a layer of complexity to the analysis and thwarted the effectiveness of the script.

Appendix

```
Java.perform(function () {
    const fe51a4e9aClass = Java.use('s79af6adb.fe51a4e9a.fe51a4e9a');

    fe51a4e9aClass.a436bd584.implementation = function (context, motionEvent, str) {
        console.log('Intercepted a436bd584 method. Returning false.');
        return false;
    };
});

```

Figure #22: Located the code responsible for protection

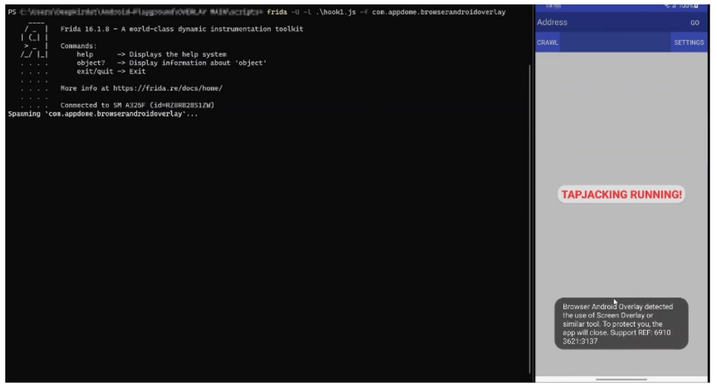


Figure #23: Failed to attach and manipulate the output

The final phase of SecureLayer7's analysis involved using an open-source application named Android-overlay-detection, which utilizes SYSTEM_ALERT_WINDOW and HIDE_OVERLAY_WINDOWS. This was in response to the code analysis finding that the app checks for the FLAG_WINDOW_IS_PARTIALLY_OBSCURED.

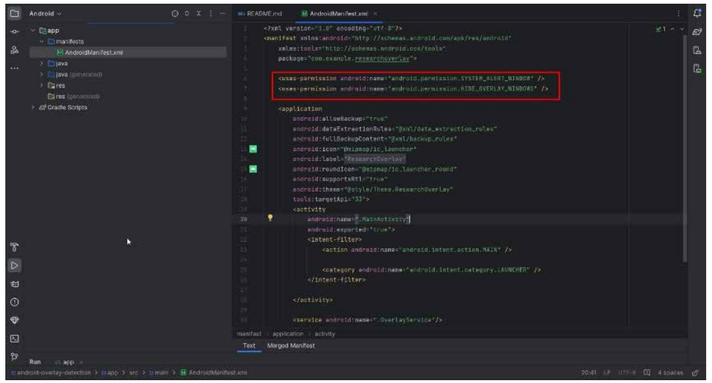


Figure #24: bypass using Android-overlay-detection open-source tool

In the final testing phase with the Android-overlay-detection application, SecureLayer7 focused on the FLAG_WINDOW_IS_OBSCURED and FLAG_WINDOW_IS_PARTIALLY_OBSCURED flags.

The application created an overlay to test these flags. It was observed that tapping the partially obscured area of the screen resulted in no response from the application. This was evidenced by the app not opening the keyboard when the address bar, located in the obscured area, was tapped.

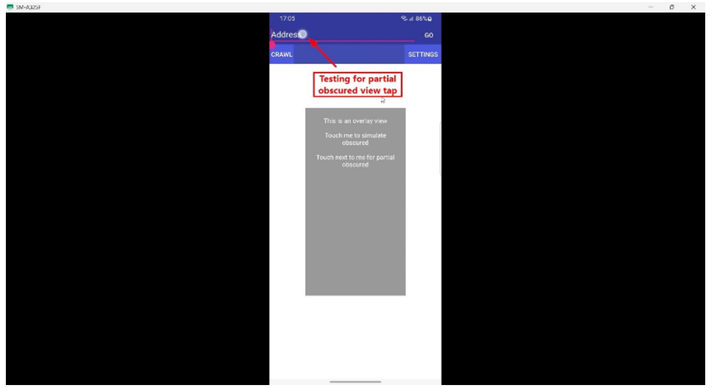


Figure #25: Bypass using Android-overlay-detection open-source tool

Additionally, when interacting with the overlay screen created by Android-overlay-detection, the protected application displayed a consistent toast message: Browser Android Overlay detected the use of Screen Overlay or similar tool. To protect you the app will close.

Appendix

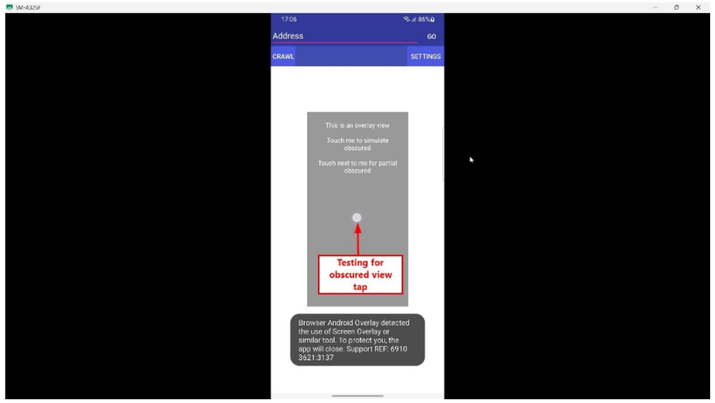


Figure #26: Bypass using Android-overlay-detection open-source tool

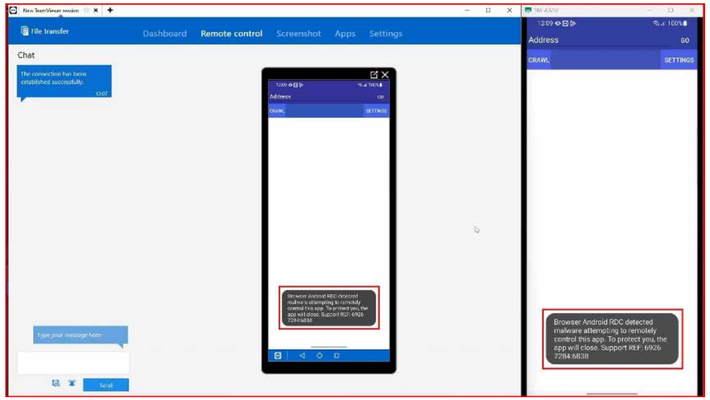


Figure #27: Utilized TeamViewer to check for the protection

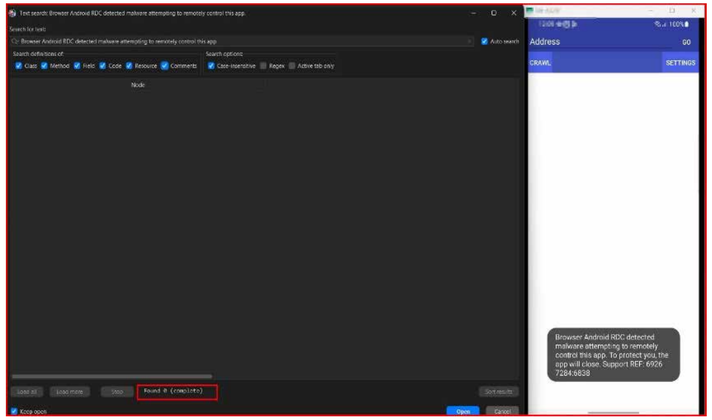


Figure #28: Utilized TeamViewer to check for the protection

In the evaluation of Anti Remote Desktop Control (RDC) protection, SecureLayer7 implemented various techniques including code-level analysis and runtime manipulation. The TeamViewer application, a tool for remote access and control, was used for this assessment. When operated on an Android device linked to a desktop, it was noted that launching an app with Anti RDC protection triggered a warning toast message. This indicated the app's capability to detect and respond to unauthorized remote control attempts.

After conducting the initial tests with TeamViewer, SecureLayer7 proceeded with a static analysis of the app's decompiled source code. This analysis aimed to locate the specific toast message triggered by the Anti Remote Desktop Control protection. However, no corresponding results were found within the code base, suggesting a sophisticated implementation of the warning mechanism that was not directly identifiable in the analyzed code.

Upon further analysis of the application's source code, Secure-Layer7 discovered that the app includes a check for the status of Accessibility services. To explore this aspect, RustDesk3, an open-source remote access and control software that utilizes Accessibility services to establish remote connections, was used. This approach aimed to test the application's response to remote access attempts involving Accessibility services.

Appendix

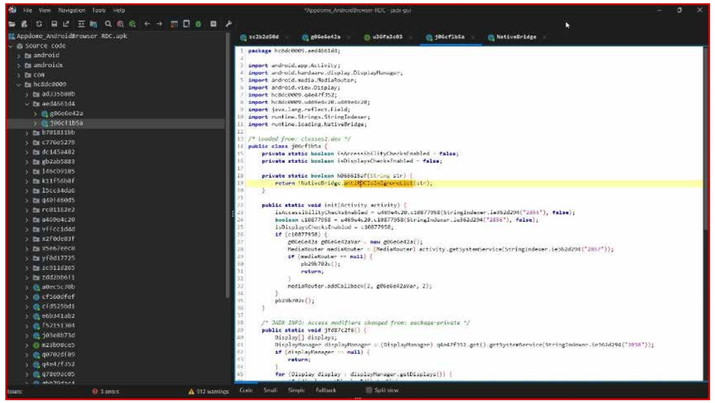


Figure #29:Analyzed the code which checks for Accessibility service

SecureLayer7 further tested the 'Block Suspicious Accessibility Services' protection by enabling it. It was observed that when the application with the above-mentioned protection is launched using remote access software such as RustDesk which utilizes Accessibility services, the application displayed a warning toast with the following "Browser Android w a11y detected malware using Accessibility Services not permitted with this app. To protect you, the app will close". Thus, rendering this attack vector useless when Block Suspicious Accessibility Services' protection is enabled.

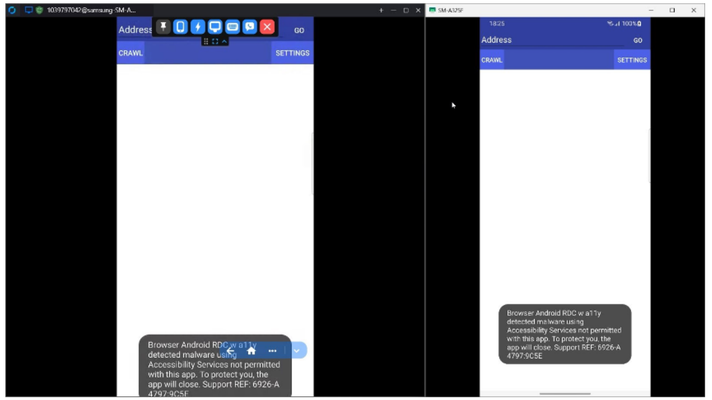


Figure #30: Failed to bypass the Block Suspicious Accessibility Services protection

In evaluating Appdome's Keylogging protection, SecureLayer7 employed a methodology incorporating code analysis, observation of changes, and runtime manipulation. An open-source keylogger, LokiBoard, was used to test this protection. LokiBoard replaces the device's default keyboard with a malicious one, recording keystrokes. When the protected application was launched with LokiBoard active, it detected the untrusted keyboard and displayed a warning toast, advising the use of an approved or built-in keyboard, thereby indicating the effectiveness of the keylogging protection.

In their static analysis aimed at uncovering the code responsible for keylogging protection, SecureLayer7 searched for the specific warning toast triggered by using a malicious keyboard. However, this analysis of the decompiled source code of the application with keylogging protection did not yield any results for the specific toast message string, suggesting a more complex or obfuscated implementation of this security feature.

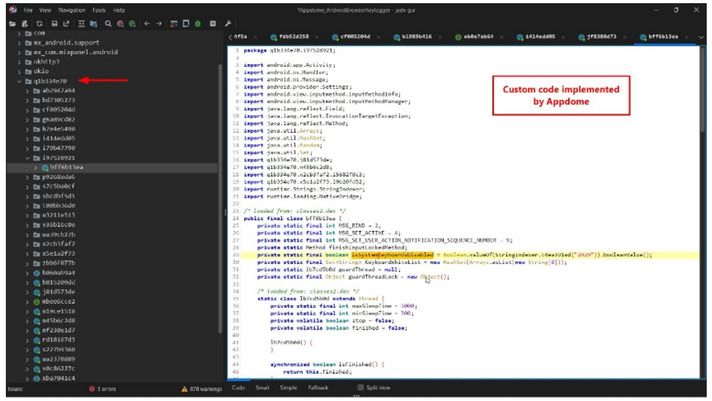


Figure #31: Failed to bypass the Block Suspicious Accessibility Services protection

Appendix

In further testing with the LokiBoard keylogger, SecureLayer7 observed that the application with keylogging protection did not recognize or accept keystrokes from the third-party keyboard. This was evident as no input was registered in the address bar of the Browser Android application when attempted with the malicious keyboard, showcasing the effectiveness of the app's keylogging protection mechanism.

The static analysis was conducted by uploading the APK with implemented memory protection to MobSF. The analysis revealed that in all shared libraries, Stack Canary and RELRO settings were enabled, with Full RELRO applied, indicating robust memory protection measures in place.

SecureLayer7's examination extended to the decompiled source code for any hardcoded keys or secrets that might be used for data encryption. Despite a thorough analysis, no such elements were found, indicating an absence of easily identifiable encryption mechanisms within the memory protection framework.

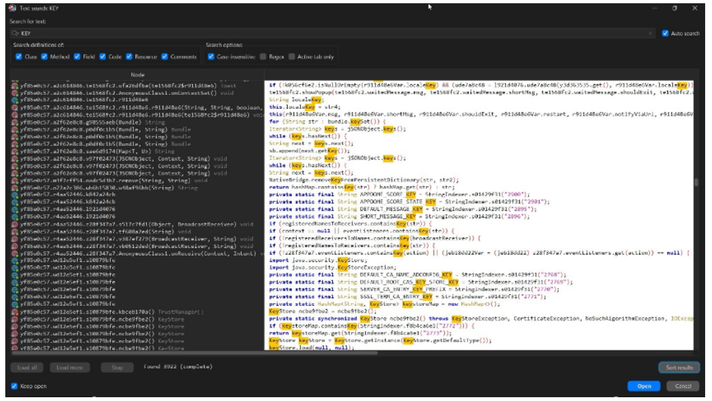


Figure #32: No encryption keys within the source code of the decompiled APK

SecureLayer7 encountered challenges while attempting to bypass emulator detection using the BrowserAndroid app. The application, executed with protection mechanisms enabled, displayed a blank screen on the Android emulator. Efforts to circumvent this included employing tools like Frida scripts and searching for lsposed/exposed modules, but these were detected and neutralized by the application. Further attempts using jd-gui and other deobfuscation tools to analyze the source code were also hindered due to the application's complex obfuscation techniques, complicating the reverse engineering process.

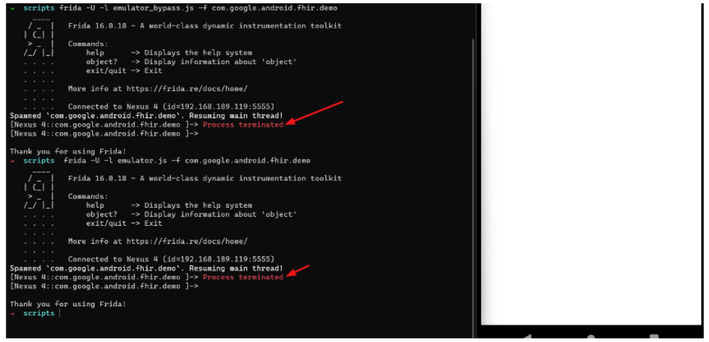


Figure #33: Tested for Emulator bypass using Frida scripts