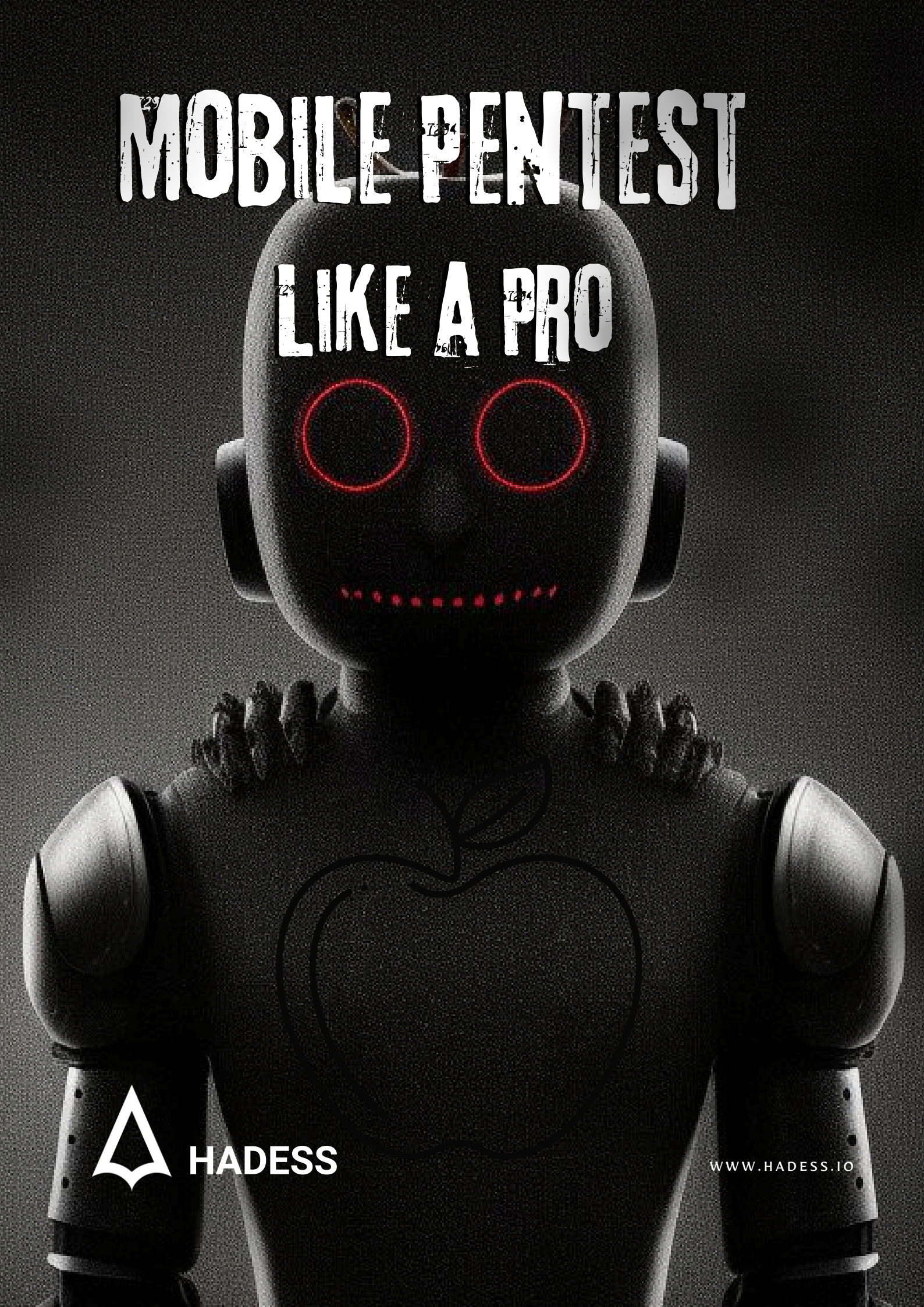
MOBILE PENTEST LIKE A PRO

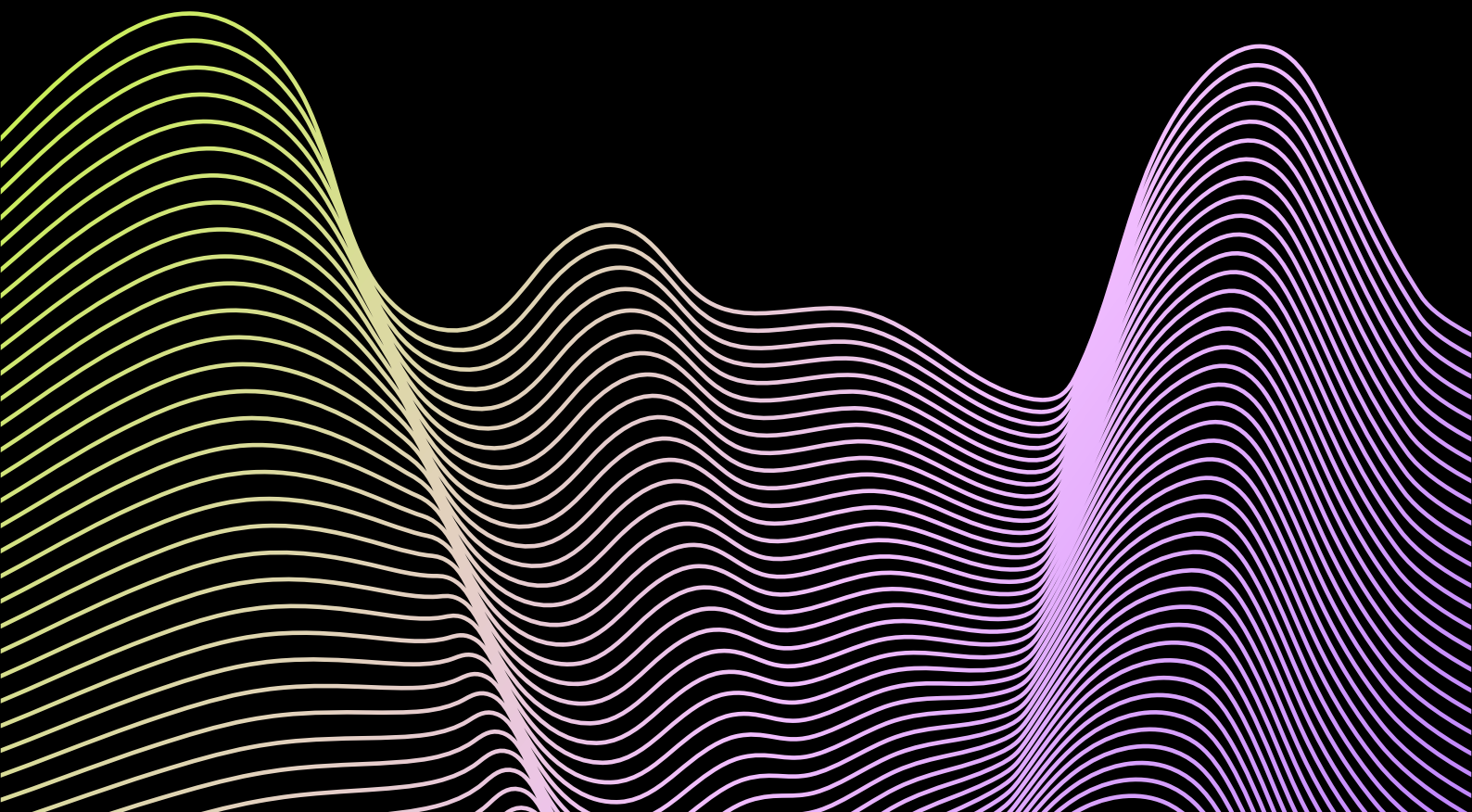


HADESS

WWW.HADESS.IO


“NOTHING IS PERMANENT IN THIS WICKED WORLD – NOT EVEN OUR TROUBLES.“
CHAPLIN

TABLE OF CONTENT



IOS Jailbreak Methods

Jailbreaking an iOS device involves removing the software restrictions imposed by Apple, allowing users to gain root access to the iOS file system and manager. This process enables the installation of apps, themes, and tweaks that are not available through the official App Store. Below are some common jailbreaking methods, including commands and codes, along with their advantages and disadvantages.

ID	Jailbreak Tool	Method	Ease of Use
1	Checkra1n	- Download from official website - Use Terminal commands - Follow on-screen DFU mode instructions	Medium
2	Unc0ver	- Download IPA file - Install via AltStore - Open app and tap "Jailbreak"	Easy
3	Taurine	- Download IPA file - Install via AltStore - Open app and tap "Jailbreak"	Easy
4	Electra	- Download IPA file - Install via Cydia Impactor - Open app and tap "Jailbreak"	Medium
5	3uTools	- Download and install 3uTools on Windows - Connect device and open 3uTools - Navigate to "Flash & JB" tab and select "Jailbreak"	Very Easy
6	Dopamine	- Download IPA file - Install via AltStore - Open app and tap "Jailbreak"	Easy

Common iOS Jailbreaking Methods

1. Checkra1n
2. Unc0ver
3. Taurine
4. Electra
5. 3utools
6. Dopamine

1. Checkra1n

Overview: Checkra1n is based on the checkm8 bootrom exploit and supports iOS devices from iPhone 5s to iPhone X running iOS 12.0 to iOS 14.5.

Commands and Codes:

1. **Download Checkra1n:**
   - Download the latest version from the official website.
   - Transfer the file to your Mac or Linux computer.
2. **Running Checkra1n on macOS:**

```
# Open Terminal
cd /path/to/checkra1n
sudo ./checkra1n
```

3. **Running Checkra1n on Linux:**

```
# Open Terminal
cd /path/to/checkra1n
sudo ./checkra1n
```

4. **Follow On-Screen Instructions:**
   - Connect your iOS device via USB.
   - Follow the instructions to put your device in DFU mode.
   - The jailbreaking process will begin.

2. Unc0ver

Overview: Unc0ver supports a wider range of iOS versions, from iOS 11.0 to iOS 14.3, and works on newer devices compared to Checkra1n.

Commands and Codes:

1. **Download Unc0ver:**
    - Get the IPA file from the official website.
2. **Install Unc0ver via AltStore:**
    - Download and install AltStore on your computer.
    - Connect your device and install AltServer.
    - Use AltStore on your device to install the Unc0ver IPA.
3. **Jailbreaking:**
    - Open the Unc0ver app on your device.
    - Tap "Jailbreak" and wait for the process to complete.

3. Taurine

Overview: Taurine, developed by the Odyssey Team, supports iOS 14.0 to iOS 14.3 and is known for its speed and reliability.

Commands and Codes:

1. **Download Taurine:**
    - Get the IPA file from the official website.
2. **Install Taurine via AltStore:**
    - Follow the same steps as with Unc0ver to use AltStore for installation.
3. **Jailbreaking:**
    - Open the Taurine app on your device.
    - Tap "Jailbreak" and follow the on-screen instructions.

4. Electra

Overview: Electra is an older tool, suitable for jailbreaking iOS 11.0 to iOS 11.4.1.

Commands and Codes:

1. **Download Electra:**
   - Obtain the IPA file from the official website.
2. **Install Electra via Cydia Impactor:**
   - Download Cydia Impactor and connect your device.
   - Drag the Electra IPA into Cydia Impactor and follow the installation steps.
3. **Jailbreaking:**
   - Open the Electra app on your device.
   - Tap "Jailbreak" and let the process complete.

5. 3uTools

Overview: 3uTools is a comprehensive tool for iOS devices, providing features for flashing firmware, managing files, and jailbreaking. It's a user-friendly tool that integrates several jailbreaking methods, making it easier for users to perform various tasks without needing multiple tools.

Method:

1. **Download and Install 3uTools:**
   - Download the latest version of 3uTools from the official website.
   - Install 3uTools on your Windows computer.
2. **Launch 3uTools and Connect Your Device:**
   - Open 3uTools and connect your iOS device using a USB cable.
3. ****Jailbreaking with 3uTools:****
   - Navigate to the "Flash & JB" tab.
   - Select "Jailbreak."
   - 3uTools will automatically detect the iOS version and provide the appropriate jailbreak tool (e.g., Checkra1n, Unc0ver).
4. **Follow On-Screen Instructions:**
   - Follow the prompts to enter DFU mode (if required).
   - The jailbreaking process will begin and complete automatically.

6. Dopamine

Overview: Dopamine is a modern jailbreak tool for iOS, primarily supporting newer iOS versions and devices. It's designed for ease of use and reliability.

Method:

1. **Download Dopamine:**
    - Obtain the latest Dopamine IPA from the official website.
2. **Install Dopamine via AltStore:**
    - Download and install AltStore on your computer (available for both macOS and Windows).
    - Connect your iOS device and open AltServer.
    - Use AltStore on your device to install the Dopamine IPA.
3. **Jailbreaking:**
    - Open the Dopamine app on your device.
    - Tap "Jailbreak" and follow the on-screen instructions.

Commands and Codes:

- **Installing AltStore:**
    - **macOS:**

```
brew install --cask altserver
```

- Open AltServer and follow the instructions to install AltStore on your device.
    - **Windows:**
        - Download the AltServer installer from the official website and follow the installation steps.
- **Using AltStore:**
    - Connect your device via USB.
    - Open AltStore on your device and select "Install AltStore."
    - Select the Dopamine IPA file and follow the prompts to install it.

Android Root Methods

ID	Root Tool	Method	Ease of Use
1	KingoRoot	- Download KingoRoot APK - Install and run the APK - Tap "One Click Root"	Very Easy
2	One Click Root	- Download One Click Root software - Install on PC - Connect device via USB - Follow on-screen instructions	Very Easy
3	Magisk	- Install Magisk Manager - Flash Magisk zip via custom recovery (e.g., TWRP) - Reboot and manage root with Magisk Manager	Easy
4	SuperSU	- Download SuperSU zip - Flash SuperSU zip via custom recovery (e.g., TWRP) - Reboot device	Easy
5	Odin (for Samsung)	- Download Odin and CF-Auto-Root - Boot device into Download Mode - Connect to PC - Use Odin to flash CF-Auto-Root	Medium
6	Xposed or EdXposed	- Install Magisk - Install Riru module - Install EdXposed module - Reboot device - Manage modules with EdXposed Manager	Medium
7	DFT Pro	- Install DFT Pro on PC - Connect device via USB - Use DFT Pro software to root device	Medium
8	Chimera	- Install Chimera Tool - Connect device to PC - Use Chimera Tool to root device	Medium
9	Global Unlocker Pro	- Install Global Unlocker Pro - Connect device to PC - Use software to root device	Medium
10	Pandora Box	- Install Pandora Box - Connect device to PC - Use Pandora Box to root device	Medium
11	Infinity CM2 Dongle	- Install Infinity CM2 Dongle software - Connect device to PC - Use software to root device	Medium

1. Magisk

Overview: Magisk is a popular tool that allows you to root your device systemlessly, meaning it doesn't modify the system partition. This makes it easier to hide the root status from apps that detect it, like banking apps.

Commands and Codes:

1. **Download Magisk:**
   - Download the latest Magisk zip and Magisk Manager APK from the official website.
2. **Install Magisk:**
   - Boot your device into custom recovery (e.g., TWRP).
   - In TWRP, select "Install" and choose the Magisk zip file.
   - Swipe to confirm the flash.
3. **Install Magisk Manager:**
   - After rebooting, install the Magisk Manager APK.

2. SuperSU

Overview: SuperSU was one of the first widespread rooting solutions, modifying the system partition to grant root access.

Commands and Codes:

1. **Download SuperSU:**
   - Download the SuperSU zip file from the official website.
2. **Install SuperSU:**
   - Boot your device into custom recovery (e.g., TWRP).
   - In TWRP, select "Install" and choose the SuperSU zip file.
   - Swipe to confirm the flash.

Important Folders & Files

ID	Title	Path	Type of File
1	Application Databases	/data/data/app_name/databases/*.sqlite, .db	SQLite Database Files
2	Shared Preferences	/data/data/app_name/shared_prefs	XML Files
3	SMS and MMS Database	/data/com.android.providers.telephony /databases/mmssms.db	SQLite Database Files
4	MMS Attachments	/data/user_de/0/com.android.providers. telephony/app_parts	Media Files
5	Samsung Messaging Database	/data/com.samsung.android.messaging /databases/messages_content.db	SQLite Database Files
6	iOS Application Databases	/Applications/.../Library/Database	SQLite Database Files
7	iOS Application Preferences	/Applications/.../Library/Preferences	Property List (.plist) Files

important files and folders in iOS and Android systems, focusing on common directories and databases that store critical application and system data. This includes paths, file types, and key tables within databases.

Important Files and Folders in Android

Application Data

- Path: `/data/data/app_name/databases/*.sqlite` , `.db`
  - **Description:** Stores SQLite database files used by apps.
  - **Example Files:**
    - `app_name.db`
    - `user_data.sqlite`

Shared Preferences

- Path: `/data/data/app_name/shared_prefs`
  - **Description:** Stores XML files for app-specific shared preferences.
  - **Example Files:**
    - `settings.xml`
    - `user_preferences.xml`

SMS and MMS Data

- Path: `/data/com.android.providers.telephony/databases/mmssms.db`
  - Description: Database for SMS and MMS messages.
  - Tables of Interest:
    - `addr` : Stores addresses related to messages.
    - `sms` : Contains SMS message data.
    - `mms` : Contains MMS message data.
    - `part` : Stores parts of MMS messages, like text and attachments.
    - `pdu` : Stores protocol data units of MMS messages.
    - `threads` : Contains threads of conversations.
    - `canonical_addresses` : Maps phone numbers to IDs.

Attachments and Parts

- Path: `/data/user_de/0/com.android.providers.telephony/app_parts`
  - Description: Stores parts of MMS messages, such as images and other attachments.

Samsung Messaging

- Path: `/data/com.samsung.android.messaging/databases/messages_content.db`
  - Description: Database specific to Samsung's messaging app.
  - Tables of Interest: Similar to `mmssms.db` but specific to Samsung's implementation.

Important Files and Folders in iOS

Application Data

- Path: `/Applications/.../Library/Database`
  - Description: Stores SQLite databases for iOS applications.
  - Example Files:
    - `app_data.sqlite`
    - `user_info.db`

Example Detailed Paths and Files:

Android

- App Data:
  - `/data/data/com.example.myapp/databases/app_data.db`
  - `/data/data/com.example.myapp/shared_prefs/settings.xml`
- Telephony Data:
  - `/data/com.android.providers.telephony/databases/mmssms.db`
  - `/data/user_de/0/com.android.providers.telephony/app_parts/attachment.jpg`
- Samsung Messaging:
  - `/data/com.samsung.android.messaging/databases/messages_content.db`

iOS

- App Data:
  - `/Applications/com.example.myapp/Library/Database/app_data.sqlite`
  - `/Applications/com.example.myapp/Library/Preferences/settings.plist`

Example Database Tables and Their Significance:

Android

- `mmssms.db` :
  - `addr` : Stores message addresses.
  - `sms` : Stores text messages.
  - `mms` : Stores multimedia messages.
  - `part` : Stores individual parts of MMS messages.
  - `pdu` : Stores protocol data units for MMS.
  - `threads` : Manages conversation threads.
  - `canonical_addresses` : Maps contact numbers to internal IDs.

iOS

- Example Database (`app_data.sqlite`):
  - `users` : Stores user information.
  - `messages` : Stores messages data.
  - `settings` : Stores application settings.

Commands for SQLite Database Management:

Android

Example Database Tables and Their Significance:

Android

- `mmssms.db` :
  - `addr` : Stores message addresses.
  - `sms` : Stores text messages.
  - `mms` : Stores multimedia messages.
  - `part` : Stores individual parts of MMS messages.
  - `pdu` : Stores protocol data units for MMS.
  - `threads` : Manages conversation threads.
  - `canonical_addresses` : Maps contact numbers to internal IDs.

iOS

- Example Database (`app_data.sqlite`):
  - `users` : Stores user information.
  - `messages` : Stores messages data.
  - `settings` : Stores application settings.

Commands for SQLite Database Management:

Android

```
adb shell
sqlite3 /data/data/com.example.myapp/databases/app_data.db
```

Useful SQLite Commands:

```
.headers on
.tables
SELECT * FROM table_name;
```

iOS

Accessing SQLite Databases on a Jailbroken Device:

```
ssh root@<device-ip>
sqlite3 /Applications/com.example.myapp/Library/Database/app_data.sqlite
```

Static Analytics

Static analysis of iOS and Android applications involves examining the code and resources of an app without executing it. Important elements to analyze include the Global Offset Table (GOT), which holds addresses of functions and variables used in the application. Here's an overview of static analytics for both platforms, including methods and relevant commands:

iOS Static Analytics

Method:

1. **Decompile the App:**
   - Use tools like Hopper, IDA Pro, or Ghidra to decompile the iOS application binary (Mach-O file).
2. **Analyze the Global Offset Table (GOT):**
   - Locate the GOT in the disassembled code.
   - Identify important addresses, such as function calls, library references, and variable access points.
3. **Extract Strings and Resources:**
   - Extract strings and resources from the binary for further analysis.
   - Look for sensitive information, API endpoints, and hardcoded values.

Command and Codes (Using Hopper Disassembler):

1. **Decompile Binary:**

```
hopper disassemble -64 -o <output_directory> <app_binary_path>
```

2. **Analyze GOT:**
   - Use Hopper's GUI to navigate to the Global Offset Table section and examine the addresses.
3. **Extract Strings:**
   - Hopper provides a built-in feature to extract strings from the disassembled code.

Android Static Analytics

Method:

1. **Decompile the APK:**
    - Use tools like JADX, apktool, or JADX-GUI to decompile the Android APK file.
2. **Inspect Smali Code:**
    - Explore the smali code to understand the application's structure and functionality.
    - Locate the Global Offset Table (GOT) equivalent in Android, which contains method and class references.
3. **Extract Resources:**
    - Extract resources, such as XML files, layouts, and strings, for analysis.

Command and Codes (Using JADX):

1. **Decompile APK:**

```
jadx -d <output_directory> <apk_file>
```

3. **Inspect Smali Code:**
    - Navigate to the `smali` directory in the output directory to explore the decompiled smali files.
    - Search for relevant sections related to the GOT.
3. **Extract Resources:**
    - Use JADX's built-in feature to extract XML files, layouts, and strings for further analysis.

Important Global Offset Table Addresses

- **iOS:**
  - In iOS, the Global Offset Table (GOT) is part of the Mach-O binary format.
  - Addresses in the GOT typically point to external functions and variables.
  - Example: `0x10001000` - Address pointing to a function in a shared library.
- **Android:**
  - In Android, the GOT equivalent is represented in the smali code.
  - Addresses point to method and class references used by the application.
  - Example: `Lcom/example/MainActivity;->onCreate(Landroid/os/Bundle;)V` - Reference to the `onCreate` method of the `MainActivity` class.

Hooking

Dynamic analysis of iOS and Android applications involves inspecting and manipulating the behavior of the app during runtime. Hooking important functions and using Frida and objection scripts are common techniques for dynamic analytics. Here's an overview, including methods, important scripts, and their applications:

iOS Dynamic Analytics

Hooking Important Functions:

1. **Using Cydia Substrate or Substitute:**
   - Hook into important functions using Substrate or Substitute, allowing you to intercept and modify behavior.
   - Common hooks include method swizzling, function interception, and dynamic class modification.
2. **Frida Scripting:**
   - Use Frida to dynamically hook into iOS applications, enabling real-time function interception and manipulation.

Important Frida and Objection Scripts:

1. **Frida Script to Bypass Jailbreak Detection:**

```
// JavaScript code to bypass jailbreak detection
Interceptor.attach(Module.findExportByName(null, "open"), {
    onEnter: function(args) {
        this.log("Bypassing jailbreak detection...");
        args[0] = Memory.allocUtf8String("/private/var/lib/apt/");
    }
});
```

Frida Script to Trace Objective-C Method Calls:

```
// JavaScript code to trace Objective-C method calls
function traceObjC(className) {
    var hook = ObjC.classes[className];
    for (var methodName in hook.$ownMethods) {
        console.log("Tracing: " + className + "." + methodName);
        try {
            hook[methodName].implementation =
ObjC.implement(hook[methodName], {
                implementation: function() {
                    console.log(className + "." + methodName + " called
with arguments: " + JSON.stringify(arguments));
                    return this[methodName].apply(this, arguments);
                }
            });
        } catch (e) {
            console.error("Error tracing " + className + "." + methodName +
": " + e);
        }
    }
}
```

Android Dynamic Analytics

Hooking Important Functions:

1. Using Xposed Framework:
    - Create Xposed modules to hook into Android applications and modify their behavior.
    - Hooks can intercept method calls, access private variables, and manipulate UI elements.
2. Frida Scripting:
    - Utilize Frida to dynamically hook into Android applications, providing real-time function interception and manipulation.

Important Frida and Objection Scripts:

Frida Script to Bypass SSL Pinning:

```
// JavaScript code to bypass SSL pinning
Java.perform(function() {
    var CertificatePinner = Java.use("okhttp3.CertificatePinner");
    CertificatePinner.check.overload('java.lang.String',
'java.util.List').implementation = function() {
        console.log("[*] Bypassing SSL pinning");
    };
});
```

Frida Script to Trace Java Method Calls:

```
// JavaScript code to trace Java method calls
Java.perform(function() {
    var targetClass = Java.use("com.example.MainActivity");
    targetClass.onCreate.implementation = function() {
        console.log("onCreate() called");
        this.onCreate();
    };
});
```

SSL Pin

ID	Function/Library Name	Popularity
1	okhttp3.CertificatePinner.check (Android)	High
2	javax.net.ssl.X509TrustManager.checkServerTrusted (Android)	High
3	javax.net.ssl.SSLContext.init (Android)	High
4	android.webkit.WebViewClient.onReceivedSslError (Android)	High
5	javax.net.ssl.HttpsURLConnection.setSSLSocketFactory (Android)	High
6	NSURLSessionDelegate.URLSession:didReceive Challenge:completionHandler: (iOS)	High
7	NSURLConnectionDelegate.connection:didReceive AuthenticationChallenge: (iOS)	High
8	CFReadStreamSetProperty (iOS)	High
9	NSURLRequest.allowsAnyHTTPSCertificateForHost (iOS)	High
10	SecTrustEvaluateWithError (iOS)	High
11	org.apache.http.impl.client.DefaultHttpClient (Android)	Medium
12	CFNetwork (iOS)	Medium
13	javax.net.ssl.HostnameVerifier.verify (Android)	Medium
14	okhttp3.OkHttpClient.Builder.sslSocketFactory (Android)	Medium
15	java.net.URL.openConnection (Android)	Medium
16	NSURLRequest.allowEgress (iOS)	Medium
17	SSLContext.getInstance (Android)	Medium
18	NSURLProtectionSpace.authenticationMethod (iOS)	Low
19	SSLContext.getInstance("TLS") (Android)	Low
20	NSURLSession:canAuthenticateAgainstProtectionSpace: (iOS)	Low

commonly hooked functions and libraries used to bypass SSL pinning in both Android and iOS applications with Frida and objection:

Top Functions and Libraries for SSL Pinning Bypass

Android:

1. OkHttp (Android):
    - Library: `okhttp3`
    - Functions to Hook:
        - `CertificatePinner.check`
        - `SSLContext.getInstance`
2. TrustManager (Android):
    - Library: `javax.net.ssl`
    - Functions to Hook:
        - `X509TrustManager.checkServerTrusted`
3. HttpClient (Android):
    - Library: `org.apache.http.impl.client.DefaultHttpClient`
    - Functions to Hook:
        - `SSLContext.init`
4. WebView (Android):
    - Library: `android.webkit`
    - Functions to Hook:
        - `WebViewClient.onReceivedSslError`
5. HttpsURLConnection (Android):
    - Library: `javax.net.ssl`
    - Functions to Hook:
        - `HttpsURLConnection.setSSLSocketFactory`

iOS:

1. NSURLSession (iOS):
    - Library: `Foundation`
    - Functions to Hook:
        - `-[NSURLSessionDelegate URLSession:didReceiveChallenge:completionHandler:]`
2. NSURLConnection (iOS):
    - Library: `Foundation`
    - Functions to Hook:
        - `-[NSURLConnectionDelegate connection:didReceiveAuthenticationChallenge:]`
3. CFNetwork (iOS):
    - Library: `CFNetwork`
    - Functions to Hook:
        - `CFReadStreamSetProperty`
4. NSURLRequest (iOS):
    - Library: `Foundation`
    - Functions to Hook:
        - `-[NSURLRequest allowsAnyHTTPSCertificateForHost:]`
5. SecTrustEvaluate (iOS):
    - Library: `Security`
    - Functions to Hook:
        - `SecTrustEvaluateWithError`

Frida and Objection scripts to hook the `CertificatePinner.check` method in the OkHttp library, commonly used for SSL pinning in Android applications:

```
// Android SSL Pinning Bypass with Frida

// Define the target class and method to hook
var className = "okhttp3.CertificatePinner";
var methodName = "check";

// Hook into the CertificatePinner.check method
Interceptor.attach(Module.findExportByName(null, "Java_" +
className.replace(/\./g, "_") + "_" + methodName), {
    onEnter: function(args) {
        console.log("[*] CertificatePinner.check() hooked!");
        // Log the arguments passed to the method
        console.log("[+] Hostname: " + args[1].readUtf8String());
        console.log("[+] Certificate: " + args[2].readUtf8String());
        // Modify the behavior if needed
        // args[0] = Memory.allocUtf8String("modified_certificate");
        // args[1] = Memory.allocUtf8String("example.com");
    }
});
```

Sample Objection Command:

```
# Start the objection tool and spawn a Frida gadget script to bypass SSL
pinning
objection --gadget "com.example.app" explore --script
"path/to/frida_script.js"
```

Root Detection

ID	Function/Library Name	Popularity
1	java.lang.Runtime.exec (Android - Magisk Detection)	High
2	java.io.File.exists (Android - Superuser.apk Detection)	High
3	com.scottyab.rootbeer.RootBeer.isRooted (Android - RootBeer Library)	High
4	java.io.File.canExecute (Android - Su Binary Detection)	High
5	java.io.File.exists (Android - BusyBox Detection)	High
6	Foundation.fileExistsAtPath (iOS - Jailbreak Detection)	High
7	Foundation.fileExistsAtPath (iOS - Cydia Detection)	High
8	Foundation.fileExistsAtPath (iOS - SSH Daemon Detection)	High
9	Foundation.fileExistsAtPath (iOS - MobileSubstrate Detection)	High
10	Foundation.sandboxed (iOS - App Sandbox Integrity Check)	High
11	java.lang.System.getenv (Android - Environment Variable Detection)	Medium
12	android.os.Build.PRODUCT (Android - Build Properties Detection)	Medium
13	java.lang.Class.forName (Android - Class Loading Detection)	Medium
14	com.scottyab.rootbeer.RootBeer.checkForDangerousProps (Android - RootBeer Library)	Medium
15	android.os.Debug.isDebuggerConnected (Android - Debugger Detection)	Medium
16	java.io.File.listRoots (Android - Root Filesystem Detection)	Medium
17	Foundation.fileExistsAtPath (iOS - Symbolic Link Detection)	Medium
18	System.loadLibrary (Android - Native Library Loading Detection)	Medium
19	com.scottyab.rootbeer.RootBeer.checkForSuBinary (Android - RootBeer Library)	Low
20	com.scottyab.rootbeer.RootBeer.detectTestKeys (Android - RootBeer Library)	Low

top 10 libraries and function names commonly hooked to bypass root detection using Frida and objection in both Android and iOS applications:

Top Libraries and Functions for Root Detection Bypass

Android:

1. Magisk Detection (Android):
    - Library: `java.lang.Runtime`
    - Function: `exec`
2. Superuser.apk Detection (Android):
    - Library: `java.io.File`
    - Function: `exists`
3. RootBeer Library (Android):
    - Library: `com.scottyab.rootbeer.RootBeer`
    - Function: `isRooted`
4. Su Binary Detection (Android):
    - Library: `java.io.File`
    - Function: `canExecute`
5. BusyBox Detection (Android):
    - Library: `java.io.File`
    - Function: `exists`

iOS:

1. Jailbreak Detection (iOS):
    - Library: `Foundation`
    - Function: `fileExistsAtPath`
2. Cydia Detection (iOS):
    - Library: `Foundation`
    - Function: `fileExistsAtPath`
3. SSH Daemon Detection (iOS):
    - Library: `Foundation`
    - Function: `fileExistsAtPath`
4. MobileSubstrate Detection (iOS):
    - Library: `Foundation`
    - Function: `fileExistsAtPath`
5. App Sandbox Integrity Check (iOS):
    - Library: `Foundation`
    - Function: `sandboxed`

Insecure Logging

insecure logging in Android and iOS applications, including both native and third-party logging frameworks:

Android:

1. **Logcat (Native Android Logging):**
   - Command: `adb logcat`

2. **Filtering by Priority Level:**

- **Command:** Filter logs by priority level (V, D, I, W, E, F, S).
- **Example:** Show logs with priority level higher than or equal to `W` (Warning) and tag `MyApp`

```
adb logcat *:W MyApp:*
```

3. **Filtering by Keyword:**

- **Command:** Filter logs by a specific keyword in the message.
- **Example:** Show logs containing the keyword `error`

```
adb logcat | grep -i error
```

4. **Filtering by Application Name:**

- **Command:** Filter logs by the name of the application package.
- **Example:** Show logs for the application with package name `com.example.app`

```
adb logcat | grep com.example.app
```

iOS:

1. **NSLog (Native iOS Logging):**
    - Command: Not applicable (logging output visible in Xcode Console or retrieved from the device)
2. **NSLog (Native iOS Logging):**
    - Command: Not applicable (logging output visible in Xcode Console or retrieved from the device)

```
NSLog(@"Message: %@", variable);
```

3. **OSLog (Native iOS Logging):**

- Command: Not applicable (logging output visible in Xcode Console or retrieved from the device)

```
os_log(OS_LOG_DEFAULT, "Message: %@", variable);
```

Insecure Storage

insecure storage in Android and iOS applications, including both native and common third-party libraries:

Android:

1. **Insecure SharedPreferences (Native Android):**

   - **Command:** Not applicable (accessed programmatically)
   - **Code:** Accessing SharedPreferences without encryption

```
SharedPreferences prefs = getSharedPreferences("my_prefs",
Context.MODE_PRIVATE);
String secret = prefs.getString("secret_key", "");
```

2. **Insecure SQLite Database (Native Android):**

- **Command:** Not applicable (accessed programmatically)
- **Code:** Creating an SQLite database without proper encryption

```
SQLiteDatabase db = dbHelper.getWritableDatabase();
```

3. **Insecure File Storage (Native Android):**

- **Command:** Not applicable (accessed programmatically)
- **Code:** Writing sensitive data to external storage

```
File file = new File(Environment.getExternalStorageDirectory(), "data.txt");
FileOutputStream fos = new FileOutputStream(file);
```

iOS:

1. Insecure UserDefaults (Native iOS):

- **Command:** Not applicable (accessed programmatically)
- **Code:** Accessing UserDefaults without encryption

```
let defaults = UserDefaults.standard
let secret = defaults.string(forKey: "secret_key") ?? ""
```

2. Insecure Keychain (Native iOS):

- **Command:** Not applicable (accessed programmatically)
- **Code:** Storing sensitive data in Keychain without proper protection:

```
let keychain = Keychain(service: "com.example.myapp")
try keychain.set("password", key: "user_password")
```

3. Insecure File Storage (Native iOS):

- **Command:** Not applicable (accessed programmatically)
- **Code:** Writing sensitive data to local file system

```
let data = "sensitive_data".data(using: .utf8)!
try data.write(to: URL(fileURLWithPath: "/path/to/file.txt"))
```

Content Provider

Content Providers can lead to unauthorized access and data leakage in both Android and iOS applications. Here are the top 20 commands for potential attacks on Content Providers in Android and iOS:

Android:

SQL Injection Attack (Query Injection) (Native Android):

```
- **Command:** Inject malicious SQL code into a query.
```

```
String maliciousSelection = "1 OR 1=1";
Cursor cursor = getContentResolver().query(Uri, projection,
maliciousSelection, selectionArgs, sortOrder);
```

Unauthorized Data Access (Native Android):

- **Command:** Access Content Provider without proper permissions.

```
Cursor cursor = getContentResolver().query(Uri, projection, selection,
selectionArgs, sortOrder);
```

Excessive Data Leakage (Native Android):

- **Command:** Query all data without proper filtering.

```
Cursor cursor = getContentResolver().query(Uri, null, null, null, null);
```

iOS:

Unauthorized Data Access (Native iOS):

```
- **Command:** Access CNContactStore without proper permissions.
```

```
let store = CNContactStore()
let keysToFetch = [CNContactGivenNameKey]
let predicate = CNContact.predicateForContacts(matchingName: "John")
do {
    let contacts = try store.unifiedContacts(matching: predicate,
keysToFetch: keysToFetch)
} catch {
    print("Error accessing contacts: \(error)")
}
```

Excessive Data Leakage (Native iOS):

- **Command:** Query all contacts without proper filtering.

```
let store = CNContactStore()
let keysToFetch = [CNContactGivenNameKey]
let allContacts = CNContactFetchRequest(keysToFetch: keysToFetch)
do {
    let contacts = try store.unifiedContacts(matching: allContacts)
} catch {
    print("Error accessing contacts: \(error)")
}
```

Exported Content Provider (Info.plist) (Native iOS):

- **Command:** Check if CNContactStore usage description is provided

```
<key>NSContactsUsageDescription</key>
<string>We need access to contacts for better user experience</string>
```

Inadequate Supply Chain Security

Inadequate supply chain security can lead to various attacks on Android and iOS applications, compromising the integrity and security of the software distribution process. Here are the top 20 commands for potential attacks related to inadequate supply chain security:

Android:

Dependency Hijacking (Android):

```
- **Command:** Replace or inject malicious dependencies into the project.
```

```
implementation 'com.example:malicious-library:1.0.0'
```

Malicious SDK Integration (Android):

- **Command:** Integrate a malicious SDK into the application.

```
implementation 'com.example:malicious-sdk:1.0.0'
```

Compromised Build Systems (Android):

- **Command:** Compromise build systems to inject malicious code during the build process.

```
echo "echo 'Malicious code executed!'" >> build.gradle
```

Code Injection via Build Scripts (Android):

- **Command:** Inject malicious code into build scripts to modify the application behavior.

```
exec {
    commandLine 'bash', '-c', 'malicious_command'
}
```

iOS:

Dependency Hijacking (iOS):

- **Command:** Replace or inject malicious dependencies into the project

```
pod 'MaliciousLibrary'
```

Malicious Framework Integration (iOS):

- **Command:** Integrate a malicious framework into the application.

```
target 'MyApp' do
    pod 'MaliciousFramework'
end
```

Compromised Build Systems (iOS):

- **Command:** Compromise build systems to inject malicious code during the build process.

```
echo "echo 'Malicious code executed!'" >> Podfile
```

Code Injection via Build Scripts (iOS):

- **Command:** Inject malicious code into build scripts to modify the application behavior.

```
post_install do |installer|
    system 'malicious_command'
end
```

Static Scanner

ID	Tool	Usage Features	Popularity
1	MobSF (Mobile Security Framework)	Static and dynamic analysis, API testing, SSL/TLS verification, data storage security checks, and more.	High
2	QARK (Quick Android Review Kit)	Static analysis for Android applications, vulnerability detection, insecure logging, cryptography issues, and more.	High
3	AndroBugs Framework	Static analysis tool for Android applications, identifying security vulnerabilities such as SQL injection, insecure data storage, and more.	Medium
4	Appknox	Automated security testing for Android and iOS apps, identifying vulnerabilities like OWASP Top 10, insecure data storage, and more.	Medium
5	Checkmarx	Static application security testing (SAST), identifying security vulnerabilities in code, including OWASP Top 10, insecure data storage, and more.	High
6	Veracode Mobile Security Testing	Automated security testing for Android and iOS applications, including static and dynamic analysis, identifying vulnerabilities, and more.	High
7	NowSecure Lab	Automated mobile app security testing, identifying vulnerabilities, insecure data storage, SSL/TLS issues, and more.	Medium
8	Fortify Static Code Analyzer	Static analysis tool for identifying security vulnerabilities in code, including OWASP Top 10, insecure data storage, and more.	High
9	Kryptowire	Automated security testing for Android and iOS applications, including static and dynamic analysis, identifying vulnerabilities, and more.	Medium
10	ZAP (Zed Attack Proxy)	Dynamic application security testing (DAST), identifying vulnerabilities, SSL/TLS issues, insecure data storage, and more.	High

Resources

- https://owasp.org/www-project-mobile-top-10/2023-risks/
- SANS FOR585
- SANS GMOB

cat ~/.hadess

"Hadess" is a cybersecurity company focused on safeguarding digital assets and creating a secure digital ecosystem. Our mission involves punishing hackers and fortifying clients' defenses through innovation and expert cybersecurity services.

Website:
WWW.HADESS.IO

Email
MARKETING@HADESS.IO

To be the vanguard of cybersecurity, Hadess envisions a world where digital assets are safeguarded from malicious actors. We strive to create a secure digital ecosystem, where businesses and individuals can thrive with confidence, knowing that their data is protected. Through relentless innovation and unwavering dedication, we aim to establish Hadess as a symbol of trust, resilience, and retribution in the fight against cyber threats.