# APP DEFENSE ALLIANCE
# MOBILE APPLICATION SECURITY ASSESSMENT

## Hypr

| | |
|---|---|
| PACKAGE NAME | com.hypr.one |
| APP VERSION | 9.1.0 |
| OPERATING SYSTEM | Android |
| PREPARED FOR | HYPR Corp |
| DATE | 1/18/2024 |

NowSecure™

# TABLE OF CONTENTS

NowSecure™

# DOCUMENT VERSION HISTORY

| Version | Description | Date |
|---|---|---|
| 1 | Initial Report | 12/14/2023 |
| 2 | Platform-1, Code-3 and Code-9 Now Pass | 1/18/2024 |

# 1.0 SUMMARY AND RECOMMENDATIONS

## 1.1 Executive Summary

For this mobile application assessment, NowSecure conducted a thorough evaluation of the application against the 33 App Defense Alliance requirements. These requirements leverage the Open Web Application Security Project® (OWASP) Mobile Application Security Verification Standard (MASVS), an industry recognized standard which establishes baseline security requirements for mobile apps. The App Defense Alliance requires that a mobile application meet certain requirements specified in the MASVS Standard Security Verification Level (L1), which indicates that an app adheres to mobile security best practices and fulfills basic requirements in terms of architecture/design, storage and privacy, cryptography, authentication/session management, network communications, platform interaction, and code quality. NowSecure also tests several optional Level 2 (L2) requirements that, while not required, are recommended for security best practice. This report uses the MASVS v1.4.0 requirements.

## 1.2 Background and Objectives

NowSecure was retained by HYPR Corp to perform an assessment against the OWASP MASVS L1 requirements on the Hypr application for the Android platform(s). The objective for this assessment was to evaluate the security and data exposure risks presented by the use of the application and present them against the requirements laid out in OWASP Mobile Application Security Verification Standard (MASVS) L1 and OWASP Mobile Security Testing Guide (MSTG).

## 1.3 Assumptions and Limitations

- Application testing was conducted on NowSecure instrumented devices that have been jailbroken/rooted. This affords the analysts the greatest level of coverage. While the mobile operating system can offer mitigating controls for application security, in most situations it is best practice for the application to secure its own data as much as possible, making the assumption that it is operating on a compromised device.
- The application's login process is authenticated through a QR Code scan that does not include a username or password.

## 1.4 Testing Methodology

NowSecure implements a five stage process when performing a full scope security assessment. The results of this evaluation are matched against the requirements set forth in the Level 1 requirements of the OWASP MASVS. The process begins by information collection and planning; gathering customer requirements and required test materials. When the assessment begins, the application is used thoroughly in order to observe not only the application's visual UI elements, but to trigger all potential network communications. In addition, the application is internally monitored using debugging and hooking techniques. That information is then used to identify vulnerabilities for users of the application as well as the application owner. Those vulnerabilities are then investigated further to identify exploit potentials to reveal user information, location,

or compromise confidentiality. Finally, that data is compiled into a final report that is delivered to the Customer.


NowSecure's app testing service centers on three main principles:

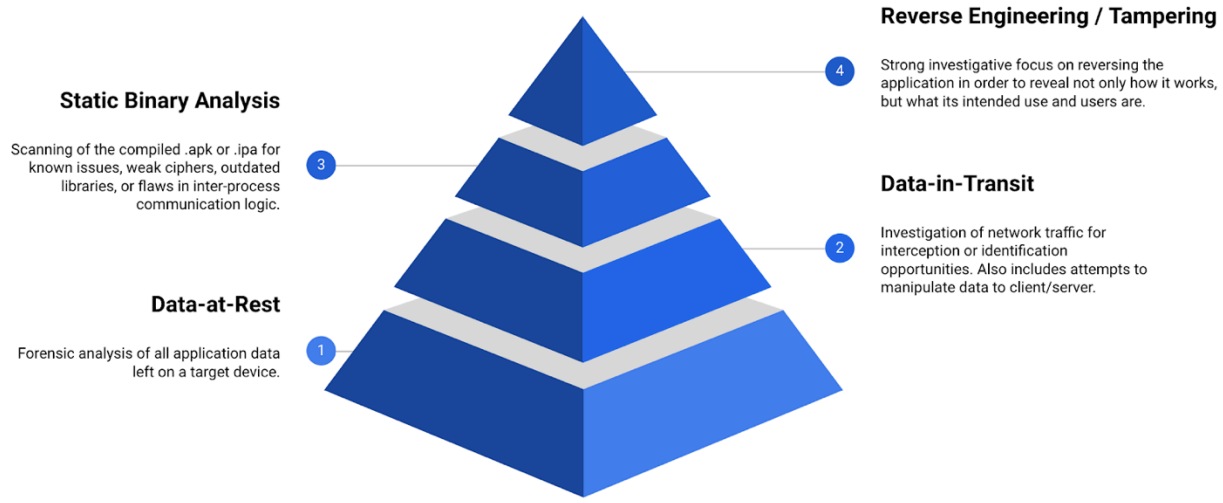
- Real Findings based on realistic application use and objective analysis
- Repeatable process that can be leveraged as a regular part of the development cycle
- Partnership with the client to ensure maximum benefit from NowSecure's expertise

In order to produce real findings, we execute the app assessment on a functional app - ideally pre-release, but it can also be performed in production. All findings are based on actual data we recover. The analysis is not informed by any functional or business considerations – purely objective findings of fact.

NowSecure's assessment includes both authenticated and unauthenticated testing where possible which accurately models the security challenges app providers face. For non-authenticated testing, we mimic the types of analysis and attacks used by cyber criminals. These include reverse engineering the apps, static and dynamic analysis, DNS and Web attacks, man-in-the-middle and more.

Furthermore, NowSecure's skilled forensic investigation uncovers artifacts which may be recoverable after a user has authenticated and utilized the application. These include insecure storage of sensitive data, circumventing passcodes once logged in, 2-factor authentication, and keychain artifacts.

To help illustrate NowSecure's comprehensive testing, the following list is provided for reference. This list may not be exhaustive as NowSecure's continual research and development add new testing criteria on a regular basis, ensuring NowSecure's tests remain up-to-date with the latest exploits and security issues.




### 1.4.1 Data-at-Rest

Analysts install the application on real devices with the target OS (iOS or Android) and conduct a forensic analysis of the device for specific application or data storage vulnerabilities. The scope of device testing includes:

- Application installation.
- Identify sensitive data stored on the device (in plaintext or reversible hashing/encoding).
- Evaluate common areas of storage for the application and its data files.
- Investigate Keychain/Keystore artifacts.
- Database structure and content evaluation.
- Biometric authentication.
- Sensitive data retained in memory.
- Data encryption.

### 1.4.2 Data-in-Transit

Analysts operate all aspects of the application as a user would and attempt to detect vulnerabilities via network communications. Taking the position of an attacker, analysts will compromise the network in a variety of ways. The scope of network testing includes:

- Identify sensitive data sent over the network (in plaintext or reversible hashing/encoding).
- Evaluate login/logout process.
- Evaluate session management techniques.
- Evaluate the security of the certificate exchange for HTTPS communications.
- Evaluation Multi-factor authentication implementation.
- Fuzzing of client-server API interactions.

Analysts conduct reconnaissance and exploitation of backend services that the mobile application interacts with. The scope of backend testing includes:

- Evaluating server cipher negotiation strength.
- Evaluate API authorization and rate limiting implementation.
- Evaluate session management techniques.

- Investigate user/token discovery and enumeration efficacy.
- Fuzzing of server-client responses.

### 1.4.3 Static Binary Analysis

Analysts use open source and proprietary tools to evaluate the fully compiled binary and discover flaws in the logic that could result in a vulnerability. The scope of static testing includes:

- Check for outdated/vulnerable third-party libraries.
- Evaluate weak cryptography implementations.
- Identify usage of ad or crash reporting SDKs.

### 1.4.4 Reverse Engineering

Analysts take the role of an attacker with limited knowledge of the application and attempt to reverse engineer the application to discover how the application works, determine if any sensitive data can be found in reversed binary source code, and attempt to manipulate the application. Reverse engineering scope includes:

- Source code obfuscation techniques.
- Anti-debug/anti-tamper techniques.
- Investigation of hard-coded secrets or other sensitive information.
- Evaluate weak cryptography implementations.
- Ability to implement biometric authentication bypass.
- Ability to implement certificate pinning bypass.
- Ability to spoof location services.
- Ability to implement jailbreak/root detection bypass.

## 1.5 Testing Platform

NowSecure performed the assessment using the following devices and OS versions:

| Device Hardware | Device Operating System | Version |
|---|---|---|
| Pixel 5a | Android | 13 |

## 1.6 Summary of Results

Below is a table summarizing all requirements and their results. Consult the details section for more information on the analysis results.

| ID | Requirement | Result | Summary |
|---|---|---|---|
| MSTG-STORAGE-1 | System credential storage facilities need to be used to store sensitive data, such as PII, user credentials or cryptographic keys. | Pass | During analysis the application's local storage, shared storage and source code were examined for crypto keys and passwords and none were located. |
| MSTG-STORAGE-2 | No sensitive data should be stored outside of the app container or system credential storage facilities. | Pass | During analysis the shared storage was examined for sensitive data shared by the application and no sensitive data was found. |
| MSTG-STORAGE-3 | No sensitive data is written to application logs. | Pass | During analysis the application's logs were examined in real time while the application was fully exercised. No sensitive data was discovered from the output of those logs. |
| MSTG-STORAGE-5 | The keyboard cache is disabled on text inputs that process sensitive data. | Pass | During analysis the application was exercised to determine if test suggestions were offered. Text suggestions were not located when examining the binary. |
| MSTG-STORAGE-7 | No sensitive data, such as passwords or pins, is exposed through the user interface. | Pass | During analysis it was determined this test control is not applicable as the application does not include a user interface for login. |
| MSTG-STORAGE-12 | The app educates the user about the types of personally identifiable information processed, as well as security | Pass | During analysis it was determined the privacy policy is accessible and accurately discloses what data is used and there are no discrepancies |

| | | | |
|---|---|---|---|
| | best practices the user should follow in using the app. | | between the disclosed data safety section. |
| MSTG-CRYPTO-1 | The app does not rely on symmetric cryptography with hardcoded keys as a sole method of encryption. | Pass | During analysis it was determined the application does not rely solely on symmetric cryptography with hard coded cryptographic keys. |
| MSTG-CRYPTO-2 | The app uses proven implementations of cryptographic primitives. | Pass | During analysis it was determined the application uses well known standard cryptographic algorithms. |
| MSTG-CRYPTO-3 | The app uses cryptographic primitives that are appropriate for the particular use-case, configured with parameters that adhere to industry best practices. | Pass | During analysis it was determined the application does not use weak key generation, deprecated modes for functions or static/predictable IVs. |
| MSTG-CRYPTO-4 | The app does not use cryptographic protocols or algorithms that are widely considered deprecated for security purposes. | Pass | Insecure hashing algorithms MD5 and SHA-1 were discovered during analysis but were not used in sensitive contexts and mostly used by third parties. |
| MSTG-CRYPTO-5 | The app doesn't re-use the same cryptographic key for multiple purposes. | Pass | During analysis it was determined the application does not generate or reuse cryptographic keys. |
| MSTG-CRYPTO-6 | All random values are generated using a sufficiently secure random number generator. | Pass | During analysis it was determined random cryptographic values were generated securely. |
| MSTG-AUTH-1 | If the app provides users access to a remote service, some form of authentication, such as username/password authentication, is performed at the remote endpoint. | Pass | During analysis it was determined the application authenticates through biometrics. Sensitive resources are not present during this process. |

| | | | |
|---|---|---|---|
| MSTG-AUTH-2 | If stateful session management is used, the remote endpoint uses randomly generated session identifiers to authenticate client requests without sending the user's credentials. | Pass | During analysis it was determined random session IDs are generated after successful user authentication and use credentials are not exchanged again within the same session. |
| MSTG-AUTH-3 | If stateless token-based authentication is used, the server provides a token that has been signed using a secure algorithm. | Pass | During analysis it was determined the application does not generate a JWT. |
| MSTG-AUTH-4 | The remote endpoint terminates the existing session when the user logs out. | Pass | During analysis it was determined the application does not timeout as the user needs the application signed in to authenticate on the Hypr website. The application is used strictly as an authenticator. |
| MSTG-AUTH-5 | A password policy exists and is enforced at the remote endpoint. | Pass | During analysis it was determined the application does not have a password policy as it authenticates via biometrics. |
| MSTG-AUTH-6 | The remote endpoint implements a mechanism to protect against the submission of credentials an excessive number of times. | Pass | During analysis it was determined brute prevention is not needed as the application uses biometrics from the local device to authenticate. |
| MSTG-AUTH-7 | Sessions are invalidated at the remote endpoint after a predefined period of inactivity and access tokens expire. | Pass | During analysis it was determined the application does not timeout as the user needs the application signed in to authenticate on the Hypr website. The application is used strictly as an authenticator. |

| | | | |
|---|---|---|---|
| MSTG-NETWORK-1 | Data is encrypted on the network using TLS. The secure channel is used consistently throughout the app. | Pass | During analysis no HTTP requests related to the application were discovered. |
| MSTG-NETWORK-2 | The TLS settings are in line with current best practices, or as close as possible if the mobile operating system does not support the recommended standards. | Pass | During analysis it was determined the application uses TLS v1.3 which is in line with current best practices. |
| MSTG-NETWORK-3 | The app verifies the X.509 certificate of the remote endpoint when the secure channel is established. Only certificates signed by a trusted CA are accepted. | Pass | During analysis MITM attacks were attempted and no certificate validation or hostname verification issues were found. |
| MSTG-PLATFORM-1 | The app only requests the minimum set of permissions necessary. | Pass | During analysis it was determined the application does not request excessive permissions for the application's functionality. |
| MSTG-PLATFORM-2 | All inputs from external sources and the user are validated and if necessary sanitized. This includes data received via the UI, IPC mechanisms such as intents, custom URLs, and network sources. | Pass | During analysis it was determined the application exported multiple content providers (3) but they were not used for sensitive functions. Context-registered broadcast receivers without permissions were examined. Permissions were not required for the data being handled. |
| MSTG-PLATFORM-3 | The app does not export sensitive functionality via custom URL schemes, unless these mechanisms are properly protected. | Pass | The deep links used by the application did not expose sensitive data. During analysis WebViews were not located. |
| MSTG-PLATFORM-4 | The app does not export sensitive functionality through IPC facilities, unless these | Pass | During analysis it was determined the application exported multiple content |

| | | | |
|---|---|---|---|
| | mechanisms are properly protected. | | providers (3) but they were not used for sensitive functions. Context-registered broadcast receivers without permissions were examined. Permissions were not required for the data being handled. |
| MSTG-CODE-1 | The app is signed and provisioned with a valid certificate, of which the private key is properly protected. | Pass | During analysis it was determined the application uses the v2 signing scheme. |
| MSTG-CODE-2 | The app has been built in release mode, with settings appropriate for a release build (e.g. non-debuggable). | Pass | During analysis the AndroidManifest.xml was examined to determine if the android:debuggable="true" directive was present and no debuggable results were found. |
| MSTG-CODE-3 | Debugging symbols have been removed from native binaries. | Pass | During analysis libraries and architectures were examined for false stripped results to determine if debug symbols are present. The stripped statuses for the application are set to true. |
| MSTG-CODE-4 | Debugging code and developer assistance code (e.g. test code, backdoors, hidden settings) have been removed. The app does not log verbose errors or debugging messages. | Pass | During analysis the application was exercised thoroughly and debug messages and debug code were not found during examination of the logs. |
| MSTG-CODE-5 | All third party components used by the mobile app, such as libraries and frameworks, are identified, and checked for known vulnerabilities. | Pass | During analysis third party libraries were examined and it was determined no known vulnerable libraries were located. |
| MSTG-CODE-9 | Free security features offered by the toolchain, such as byte-code minification, stack | Pass | During analysis it was determined no libraries had stack canary set to false. |

| | | | |
|---|---|---|---|
| | protection, PIE support and automatic reference counting, are activated. | | Proper stack smashing protection is in place. |

## 1.7 Additional Security Best Practice Recommendations

In addition to the App Defense Alliance requirements, several MASVS L2 requirements pertain to guidance currently under review by NIST. Please note that these are **not required** for App Defense Alliance certification, but should be considered security best practice.

### (Optional) MSTG-ARCH-10: Security is addressed within all parts of the software development lifecycle.

Even after the software has been released, a critical part of the maintenance phase of the SDLC is to continually monitor for potential bugs, defects, or security vulnerabilities. This includes third-party components that comprise an application. Users of the application have a reasonable expectation that the application is monitored for new vulnerabilities and will address them until the application is no longer updated or is considered end-of-life. It's important that application users understand when an expectation of security maintenance expires or the application ultimately becomes end-of-life. Ideally the application developer will display an end-of-life policy on their website or within the app itself. New draft NIST guidance (NIST SP-800-216) states "The product or service owner should assist stakeholders in dealing with vulnerabilities until a product has reached the end of service."

While *not required* for App Defense Alliance certification, we encourage all developers to create an end-of-life policy that states the timelines by which security updates will no longer be made. Ideally, the developer will notify users of the application at least 1 year prior to application end-of-life.

**Result:** Pass

There is a public security end of life policy at https://www.hypr.com/trust-center/terms-of-service

(Optional) MSTG-ARCH-11: A responsible disclosure policy is in place and effectively applied.

Both NIST SP-800-216 and ISO 29147 on which it references provide requirements and recommendations to vendors on the disclosure of vulnerabilities in products and services. Vulnerability disclosure enables users to perform technical vulnerability management as specified in both documents. The goal of vulnerability disclosure is to reduce the risk associated with exploiting vulnerabilities. Coordinated vulnerability disclosure is especially important when multiple vendors are affected, such as common in third-party components frequently shared in applications.

While *not required* for App Defense Alliance certification, we encourage all developers to define an appropriate vulnerability disclosure policy, which provides:

- information on how the developer receives and reviews reports about potential vulnerabilities;
- information on how to submit a vulnerability
- guidelines on how vulnerability remediations will be disclosed
- any terms and conditions associated with the submission of a vulnerability

**Result:** Pass

There is a public VDP program at https://www.hypr.com/trust-center/security-advisories

# 2.0 REQUIREMENT DETAILS

## MSTG-STORAGE-1: System credential storage facilities need to be used to store sensitive data, such as PII, user credentials or cryptographic keys.

**Result:** Pass **MASVS Reference:** MASVS 2.1

**Supporting Information:**
In general, sensitive data stored locally on the device should always be at least encrypted, and any keys used for encryption methods should be securely stored within the Android Keystore or iOS Keychain. These files should also be stored within the application sandbox. If achievable for the application, sensitive data should be stored off device or, even better, not stored at all. iOS offers secure storage APIs, which allow developers to use the cryptographic hardware available on every iOS device. If these APIs are used correctly, sensitive data and files can be secured via hardware-backed 256-bit AES encryption. When storing personally identifiable information, or other sensitive data such as cryptographic keys, using secure (often hardware backed) facilities provided by the platform is typically the best practice.

**Analyst Details:**
During analysis the application's local storage, shared storage and source code were examined for cryptographic keys and passwords and none were located.

## MSTG-STORAGE-2: No sensitive data should be stored outside of the app container or system credential storage facilities.

**Result:** Pass **MASVS Reference:** MASVS 2.2

**Supporting Information:**
In general, sensitive data stored locally on the device should always be at least encrypted, and any keys used for encryption methods should be securely stored within the Android Keystore or iOS Keychain. These files should also be stored within the application sandbox. If achievable for the application, sensitive data should be stored off device or, even better, not stored at all. iOS offers secure storage APIs, which allow developers to use the cryptographic hardware available on every iOS device. If these APIs are used correctly, sensitive data and files can be secured via hardware-backed 256-bit AES encryption. When storing personally identifiable information, or other sensitive data such as cryptographic keys, using secure (often hardware backed) facilities provided by the platform is typically the best practice.

**Analyst Details:**
During analysis the shared storage was examined for sensitive data shared by the application and no sensitive data was found.

## MSTG-STORAGE-3: No sensitive data is written to application logs.

**Result:** Pass **MASVS Reference:** MASVS 2.3

**Supporting Information:**
As a general recommendation to avoid potential sensitive application data leakage, logging statements should be removed from production releases unless deemed necessary to the application or explicitly identified as safe. Developers should take care to remove logging statements that write credentials, cryptographic information, or other sensitive data related to the application or user.

**Analyst Details:**
During analysis the application's logs were examined in real time while the application was fully exercised. No sensitive data was discovered from the output of those logs.

## MSTG-STORAGE-5: The keyboard cache is disabled on text inputs that process sensitive data.

**Result:** Pass **MASVS Reference:** MASVS 2.5

**Supporting Information:**
When users type in input fields, the software automatically suggests data. This feature can be very useful for messaging apps. However, the keyboard cache may disclose sensitive information when the user selects an input field that takes this type of information.

**Analyst Details:**
During analysis the application was exercised to determine if test suggestions were offered. Text suggestions were not located when examining the binary.

## MSTG-STORAGE-7: No sensitive data, such as passwords or pins, is exposed through the user interface.

**Result:** Pass **MASVS Reference:** MASVS 2.7

**Supporting Information:**
Often, application functionality may warrant entering sensitive data directly into the app's UI. This data may be financial information such as credit card data or user account passwords. However, this data may be exposed if the app doesn't properly mask it while it is being typed. The most common issue this attempts to mitigate is risks such as shoulder surfing. Sensitive data should not be exposed unless explicitly required. Masking is typically done by showing asterisks or dots instead of clear text.

**Analyst Details:**
During analysis it was determined this test control is not applicable as the application does not include a user interface for login.

## MSTG-STORAGE-12: The app educates the user about the types of personally identifiable information processed, as well as security best practices the user should follow in using the app.

**Result:** Pass **MASVS Reference:** MASVS 2.12

**Supporting Information:**
Mobile apps handle all kinds of sensitive user data, from identification and banking information to health data. There is an understandable concern about how this data is handled and where it ends up. A privacy policy should be readily accessible in both the application and the developer website. That policy should declare what data is being collected, used, and how. Over the last year both Google Play and the App Store introduced Nutrition Labels / Data Safety Labels to help users understand how their data is being collected, handled and shared. It is vital that these labels are accurate in order to provide user assurance and mitigate developer abuse.

- App Store Nutrition Labels (since 2020).
- Google Play Data Safety Labels (since 2021).

**Analyst Details:**
During analysis it was determined the privacy policy is accessible and accurately discloses what data is used and there are no discrepancies between the disclosed data safety section.

## MSTG-CRYPTO-1: The app does not rely on symmetric cryptography with hardcoded keys as a sole method of encryption.

**Result:** Pass **MASVS Reference:** MASVS 3.1

**Supporting Information:**
Cryptography plays an especially important role in securing the user's data - even more so in a mobile environment, where attackers having physical access to the user's device is a likely scenario. This test case focuses on hardcoded symmetric cryptography where best practice dictates that symmetric keys shall not be hardcoded in the application and that this cryptographic function is not used as the only method of encryption.
Encryption algorithms convert plaintext data into cipher text that conceals the original content. Plaintext data can be restored from the cipher text through decryption. Encryption can be symmetric (secret-key encryption) or asymmetric (public-key encryption).
Symmetric-key encryption algorithms use the same key for both encryption and decryption. This type of encryption is fast and suitable for bulk data processing. Since everybody who has access to the key is able to decrypt the encrypted content, this method requires careful key management. As such, hardcoding the key into the mobile application means that all users of the application will use the same key to encrypt and decrypt data.

**Analyst Details:**
During analysis it was determined the application does not rely solely on symmetric cryptography with hard coded cryptographic keys.

## MSTG-CRYPTO-2: The app uses proven implementations of cryptographic primitives.

**Result:** Pass **MASVS Reference:** MASVS 3.2

**Supporting Information:**
This test case focuses on the implementation and use of cryptographic primitives. Cryptographic primitives are well-established cryptographic algorithms that are frequently used to build cryptographic protocols. A single encryption algorithm will provide no authentication mechanism, nor any explicit message integrity checking. Only when combined in security protocols, will more than one security requirement be addressed; ultimately making up the cryptographic protocol. Using primitives that are unproven may compromise the entire cryptographic protocol. For more information, see CWE 1240: Use of a Cryptographic Primitive with a Risky Implementation.

**Analyst Details:**
During analysis it was determined the application uses well known standard cryptographic algorithms.

## MSTG-CRYPTO-3: The app uses cryptographic primitives that are appropriate for the particular use-case, configured with parameters that adhere to industry best practices.

**Result:** Pass **MASVS Reference:** MASVS 3.3

**Supporting Information:**
This test case focuses on the implementation and use of cryptographic primitives. Cryptographic primitives are well-established cryptographic algorithms that are frequently used to build cryptographic protocols. A single encryption algorithm will provide no authentication mechanism, nor any explicit message integrity checking. Only when combined in security protocols, will more than one security requirement be addressed; ultimately making up the cryptographic protocol. Common configuration issues may include insufficient key length, weak key generation functions, weak random number generators, and incorrect/weak cipher modes.

**Analyst Details:**
During analysis it was determined the application does not use weak key generation, deprecated modes for functions or static/predictable IVs.

## MSTG-CRYPTO-4: The app does not use cryptographic protocols or algorithms that are widely considered deprecated for security purposes.

**Result:** Pass **MASVS Reference:** MASVS 3.4

**Supporting Information:**
Application developers should make sure that the app does not use cryptographic algorithms and protocols that have significant known weaknesses or are otherwise insufficient for modern security requirements. Algorithms that were considered secure in the past may become insecure over time. Vulnerable algorithms include outdated block ciphers (such as DES and 3DES), stream ciphers (such as RC4), hash functions (such as MD5 and SHA1), and broken random number generators (such as Dual*EC*DRBG and SHA1PRNG). Algorithms with known weaknesses should be replaced with more secure alternatives. Alternatives should be up to date and in-line with industry standards, have a key length as recommended by industry standards, and are reasonable for the operation intended. Some generally accepted recommendations are:

- Confidentiality algorithms: AES-GCM-256 or ChaCha20-Poly1305
- Integrity algorithms: SHA-256, SHA-384, SHA-512, Blake2, the SHA-3 family
- Digital signature algorithms: RSA (3072 bits and higher), ECDSA with NIST P-384
- Key establishment algorithms: RSA (3072 bits and higher), DH (3072 bits or higher), ECDH with NIST P-384

**Analyst Details:**
Insecure hashing algorithms MD5 and SHA-1 were discovered during analysis but were not used in sensitive contexts and mostly used by third parties.

## MSTG-CRYPTO-5: The app doesn't re-use the same cryptographic key for multiple purposes.

**Result:** Pass **MASVS Reference:** MASVS 3.5

**Supporting Information:**
The security implications of cryptographic key reuse are substantial. Attacks leveraged by a key that's reused may allow an attacker to gain access to sensitive information like administrator credentials which can be used in further attacks or personal sensitive information. The private key in public key cryptography should be exclusively used for signing and the public key only for encryption. Symmetric keys should not be reused for multiple purposes. A new symmetric key should be generated if it's used in a different context.

**Analyst Details:**
During analysis it was determined the application does not generate or reuse cryptographic keys.

## MSTG-CRYPTO-6: All random values are generated using a sufficiently secure random number generator.

**Result:** Pass **MASVS Reference:** MASVS 3.6

**Supporting Information:**
Pseudo-random number generators (RNG) compensate for the inability to truly make deterministic randomness by producing a stream of pseudo-random numbers - a stream of numbers that appear as if they were randomly generated. The quality of the generated numbers varies with the type of algorithm used. Cryptographically secure RNGs generate random numbers that pass statistical randomness tests, and are resilient against prediction attacks (e.g. it is statistically infeasible to predict the next number produced). This test case focuses on random values used by the application.

**Analyst Details:**
During analysis it was determined random cryptographic values were generated securely.

## MSTG-AUTH-1: If the app provides users access to a remote service, some form of authentication, such as username/password authentication, is performed at the remote endpoint.

**Result:** Pass **MASVS Reference:** MASVS 4.1

**Supporting Information:**
Authentication and authorization problems are prevalent security vulnerabilities that consistently rank in the most common risks identified. Most mobile apps implement some kind of user authentication. It's important that the remote services that an app connects to are protected by some form of authentication.

**Analyst Details:**
During analysis it was determined the application authenticates through biometrics. Sensitive resources are not present during this process.

## MSTG-AUTH-2: If stateful session management is used, the remote endpoint uses randomly generated session identifiers to authenticate client requests without sending the user's credentials.

**Result:** Pass **MASVS Reference:** MASVS 4.2

**Supporting Information:**
Stateful (or "session-based") authentication is characterized by authentication records on both the client and server. When sessions are improperly managed, they are vulnerable to a variety of attacks that may compromise the session of a legitimate user, allowing the attacker to impersonate the user. This may result in lost data, compromised confidentiality, and illegitimate actions. It's important to ensure the consistent enforcement of authorization. The backend service must verify the user's session ID or token and make sure that the user has sufficient privileges to access the resource. If the session ID or token is missing or invalid, the request must be rejected.

**Analyst Details:**
During analysis it was determined random session IDs are generated after successful user authentication and use credentials are not exchanged again within the same session.

## MSTG-AUTH-3: If stateless token-based authentication is used, the server provides a token that has been signed using a secure algorithm.

**Result:** Pass **MASVS Reference:** MASVS 4.3

**Supporting Information:**
Token-based authentication is implemented by sending a signed token (verified by the server) with each HTTP request. The most commonly used token format is the JSON Web Token, defined in RFC7519. A JWT may encode the complete session state as a JSON object. Therefore, the server doesn't have to store any session data or authentication information. JWTs consist of three Base64Url-encoded parts separated by dots. The Token structure is as follows:

```
base64UrlEncode(header).base64UrlEncode(payload).base64UrlEncode(signature)
```

The header typically consists of two parts: the token type, which is JWT, and the hashing algorithm being used to compute the signature, such as follows where `alg` defines the algorithm used to sign or encrypt the JWT.

```
{"alg":"HS256","typ":"JWT"}
```

Most JWTs in the wild are just signed. The most common algorithms are:

- HMAC + SHA256
- RSASSA-PKCS1-v1_5 + SHA256
- ECDSA + P-256 + SHA256

It's important to remember when the token is protected using an HMAC based algorithm, the security of the token is entirely dependent on the strength of the secret used with the HMAC. If an attacker can obtain a valid JWT, they can then carry out an offline attack and attempt to crack the secret using tools such as John the Ripper or Hashcat.

**Analyst Details:**
During analysis it was determined the application does not generate a JWT.

## MSTG-AUTH-4: The remote endpoint terminates the existing session when the user logs out.

**Result:** Pass **MASVS Reference:** MASVS 4.4

**Supporting Information:**
Many mobile apps don't automatically log users out. There can be various reasons, such as: because it is inconvenient for customers, or because of decisions made when implementing stateless authentication. The application should still have a logout function, and it should be implemented according to best practices, destroying all locally stored tokens or session identifiers.
If session information is stored on the server, it should also be destroyed by sending a logout request to that server. In case of a high-risk application, tokens should be invalidated. Not removing tokens or session identifiers can result in unauthorized access to the application in case the tokens are leaked. Note that other sensitive types of information should be removed as well, as any information that is not properly cleared may be leaked later, for example during a device backup.
Failing to destroy the server-side session is one of the most common logout functionality implementation errors. This error keeps the session or token alive, even after the user logs out of the application. An attacker who gets valid authentication information can continue to use it and hijack a user's account.

**Analyst Details:**
During analysis it was determined the application does not timeout as the user needs the application signed in to authenticate on the Hypr website. The application is used strictly as an authenticator.

## MSTG-AUTH-5: A password policy exists and is enforced at the remote endpoint.

**Result:** Pass **MASVS Reference:** MASVS 4.5

**Supporting Information:**
Password strength is a key concern when passwords are used for authentication. The password policy defines requirements to which end users should adhere. A password policy typically specifies password length, password complexity, and password topologies. A "strong" password policy makes manual or automated password cracking difficult or impossible.
Some considerations for password strength:

- Password Length: Per NIST SP800-63B, passwords shorter than 8 characters are considered to be weak.
- Maximum password length should not prohibit users from creating passphrases.
- Passwords should not be silently truncated if length exceeds the maximum length.
- All unicode and whitespace characters should be acceptable.

**Analyst Details:**
During analysis it was determined the application does not have a password policy as it authenticates via biometrics.

## MSTG-AUTH-6: The remote endpoint implements a mechanism to protect against the submission of credentials an excessive number of times.

**Result:** Pass **MASVS Reference:** MASVS 4.6

**Supporting Information:**
Ideally, to prevent login brute force attempts, the remote endpoint should employ some sort of throttling to prevent automated attacks. It may be something as simple as a counter for logins attempted in a short period of time with a given user name and a method to prevent login attempts after the maximum number of attempts has been reached. After an authorized login attempt, the error counter should be reset. A five-minute account lock is commonly used for temporary account locking.

**Analyst Details:**
During analysis it was determined brute force prevention is not needed as the application uses biometrics from the local device to authenticate.

## MSTG-AUTH-7: Sessions are invalidated at the remote endpoint after a predefined period of inactivity and access tokens expire.

**Result:** Pass **MASVS Reference:** MASVS 4.7

**Supporting Information:**
Minimizing the lifetime of session identifiers and tokens decreases the likelihood of successful account/session hijacking. A Session Hijacking attack consists of an attack that compromises the session token by stealing or predicting a valid session token to gain unauthorized access to the remote service. A familiar form of this attack is known as cross-site scripting, where the attacker tricks the user's computer into running code which is treated as trustworthy because it appears to belong to the server, allowing the attacker to obtain a copy of the session token or perform other operations.

**Analyst Details:**
During analysis it was determined the application does not timeout as the user needs the application signed in to authenticate on the Hypr website. The application is used strictly as an authenticator.

## MSTG-NETWORK-1: Data is encrypted on the network using TLS. The secure channel is used consistently throughout the app.

**Result:** Pass **MASVS Reference:** MASVS 5.1

**Supporting Information:**
One of the core mobile app functions is sending/receiving data. If that data is not properly protected in transit, an attacker with access to any part of the network infrastructure (e.g., a Wi-Fi access point) may intercept, read, and/or modify it. This is why plaintext network protocols are rarely advisable. TLS is the currently accepted standard by which the unencrypted HTTP protocol is wrapped in an encrypted connection. Even when sensitive data is not being exchanged, it's prudent to still communicate via that encrypted channel. Most modern third party services also offer HTTPS (HTTP over TLS) connections to their endpoints.

**Analyst Details:**
During analysis no HTTP requests related to the application were discovered.

## MSTG-NETWORK-2: The TLS settings are in line with current best practices, or as close as possible if the mobile operating system does not support the recommended standards.

**Result:** Pass **MASVS Reference:** MASVS 5.2

**Supporting Information:**
One of the core mobile app functions is sending/receiving data. If that data is not properly protected in transit, an attacker with access to any part of the network infrastructure (e.g., a Wi-Fi access point) may intercept, read, and/or modify it. This is why plaintext network protocols are rarely advisable. TLS is the currently accepted standard by which the unencrypted HTTP protocol is wrapped in an encrypted connection. Even when sensitive data is not being exchanged, it's prudent to still communicate via that encrypted channel. Most modern third party services also offer HTTPS (HTTP over TLS) connections to their endpoints. Ensuring proper TLS configuration on the server side is also important. The SSL protocol (the predecessor to TLS) is deprecated and should no longer be used. Also TLS v1.0 and TLS v1.1 have known vulnerabilities and their usage is deprecated in all major browsers. TLS v1.2 and TLS v1.3 are considered best practice for secure transmission of data. If a mobile application connects to a specific server, its networking stack can be tuned to ensure the highest possible security level for the server's configuration. Lack of support in the underlying operating system may force the mobile application to use a weaker configuration.

**Analyst Details:**
During analysis it was determined the application uses TLS v1.3 which is in line with current best practices.

## MSTG-NETWORK-3: The app verifies the X.509 certificate of the remote endpoint when the secure channel is established. Only certificates signed by a trusted CA are accepted.

**Result:** Pass **MASVS Reference:** MASVS 5.3

**Supporting Information:**
One of the core mobile app functions is sending/receiving data. If that data is not properly protected in transit, an attacker with access to any part of the network infrastructure (e.g., a Wi-Fi access point) may intercept, read, and/or modify it. This is why plaintext network protocols are rarely advisable. TLS is the currently accepted standard by which the unencrypted HTTP protocol is wrapped in an encrypted connection. Even when sensitive data is not being exchanged, it's prudent to still communicate via that encrypted channel. Most modern third party services also offer HTTPS (HTTP over TLS) connections to their endpoints. Encrypting communication between a mobile application and its backend API is not trivial. Developers often decide on simpler but less secure solutions (e.g., those that accept any certificate) to facilitate the development process, and sometimes these weak solutions make it into the production version, potentially exposing users to interception attacks.
The application should always verify that a certificate comes from a trusted source, i.e. a trusted CA (Certificate Authority) and determine whether the endpoint server presents the right certificate.

**Analyst Details:**
During analysis MITM attacks were attempted and no certificate validation or hostname verification issues were found.

## MSTG-PLATFORM-1: The app only requests the minimum set of permissions necessary.

**Result:** Pass **MASVS Reference:** MASVS 6.1

**Supporting Information:**
Because each Android app operates in a process sandbox, apps must explicitly request access to resources and data that are outside their sandbox. They request this access by declaring the permissions they need to use system data and features. Depending on how sensitive or critical the data or feature is, the Android system will grant the permission automatically or ask the user to approve the request. Android permissions are classified into four different categories on the basis of the protection level they offer:

- Normal: This permission gives apps access to isolated application-level features with minimal risk to other apps, the user, and the system.
- Dangerous: This permission usually gives the app control over user data or control over the device in a way that impacts the user.
- Signature: This permission is granted only if the requesting app was signed with the same certificate used to sign the app that declared the permission.

iOS makes all third-party apps run under the non-privileged mobile user with each app being sandboxed. Access to protected resources or data (some also known as app capabilities) is possible, but it's strictly controlled via special permissions known as entitlements. For most, the user will be explicitly asked the first time the app attempts to access a protected resource, such as for Bluetooth peripherals, Calendar data, Location, Camera, etc. Apple asks developers to be very clear on how they use the permissions they ask for, but the end-result is not always obvious.
In both cases, this test case evaluates if the permissions requested align with the needs of the application. Permissions that do not appear necessary or are unused after the execution of the application may be flagged.

**Analyst Details:**
During analysis it was determined the application does request excessive permissions for the application's functionality. More information is needed about why the application needs to record audio.

***Note**: The developer has resolved this issue by providing the reasoning why the application requires access to record audio.*

*"Record Audio permission is required because one of our Authentication mechanisms allows the user to register and authenticate with voice. This functionality has been deprecated for new users, however, any existing users who had this type of authentication registered before deprecation can still use it."*

## MSTG-PLATFORM-2: All inputs from external sources and the user are validated and if necessary sanitized. This includes data received via the UI, IPC mechanisms such as intents, custom URLs, and network sources.

**Result:** Pass **MASVS Reference:** MASVS 6.2

**Supporting Information:**
Android apps can expose functionality through custom URL schemes (which are a part of Intents). They can expose functionality to other apps via IPC mechanisms, such as Intents, Binders, Android Shared Memory, or BroadcastReceivers), or the user via the user interface. None of the input from these sources should be trusted and must be validated and/or sanitized. Validation ensures processing of data that the app is expecting only. If validation is not enforced, any input can be sent to the app, which may allow an attacker or malicious app to exploit app functionality. iOS apps expose similar functionality through URL schemes and Apple's recommended Universal Links.

**Analyst Details:**
During analysis it was determined the application exported multiple content providers (3) but they were not used for sensitive functions. Context-registered broadcast receivers without permissions were examined. Permissions were not required for the data being handled.

## MSTG-PLATFORM-3: The app does not export sensitive functionality via custom URL schemes, unless these mechanisms are properly protected.

**Result:** Pass **MASVS Reference:** MASVS 6.3

**Supporting Information:**
Android apps can expose functionality through custom URL schemes (which are a part of Intents). They can expose functionality to other apps via IPC mechanisms, such as Intents, Binders, Android Shared Memory, or BroadcastReceivers), or the user via the user interface. None of the input from these sources should be trusted and must be validated and/or sanitized. Validation ensures processing of data that the app is expecting only. If validation is not enforced, any input can be sent to the app, which may allow an attacker or malicious app to exploit app functionality. iOS apps expose similar functionality through URL schemes and Apple's recommended Universal Links. As a developer, you should carefully validate any URL before calling it. You can allow only certain applications which may be opened via the registered protocol handler. Prompting users to confirm the URL-invoked action is another helpful control.

**Analyst Details:**
The deep links used by the application did not expose sensitive data. During analysis WebViews were not located.

## MSTG-PLATFORM-4: The app does not export sensitive functionality through IPC facilities, unless these mechanisms are properly protected.

**Result:** Pass **MASVS Reference:** MASVS 6.4

**Supporting Information:**
Android apps can expose functionality through custom URL schemes (which are a part of Intents). They can expose functionality to other apps via IPC mechanisms, such as Intents, Binders, Android Shared Memory, or BroadcastReceivers), or the user via the user interface. iOS apps expose similar functionality through URL schemes and Apple's recommended Universal Links.

**Analyst Details:**
During analysis it was determined the application exported multiple content providers (3) but they were not used for sensitive functions. Context-registered broadcast receivers without permissions were examined. Permissions were not required for the data being handled.

## MSTG-CODE-1: The app is signed and provisioned with a valid certificate, of which the private key is properly protected.

**Result:** Pass **MASVS Reference:** MASVS 7.1

**Supporting Information:**
Code signing your app assures users that the app has a known source and hasn't been modified since it was last signed. Before an iOS app can integrate app services, be installed on a device, or be submitted to the App Store, it must be signed with a certificate issued by Apple.
Android requires all APKs to be digitally signed with a certificate before they are installed or run. The digital signature is used to verify the owner's identity for application updates. This process can prevent an app from being tampered with or modified to include malicious code. When an APK is signed, a public-key certificate is attached to it. This certificate uniquely associates the APK with the developer and the developer's private key. In both cases, using the latest developer guidance for signing is a must. As vulnerabilities are detected in signing schemes or advancements made in cryptography, updates to best practices are common.

**Analyst Details:**
During analysis it was determined the application uses the v2 signing scheme.

```
Signature verification succeeded
Valid APK signature v2 found

    Signer 1

        Type: X.509
        Version: 3
        Serial number: 0x23db6907
        Subject: O=HYPR Corp
        Valid from: Wed Jan 31 10:26:58 CST 2018
        Valid until: Thu Jan 19 10:26:58 CST 2068

        Public key type: RSA
        Exponent: 65537
        Modulus size (bits): 2048
        Modulus: 188559427627377284031310222265774984340256885110236029915445121988334059065979504174767580
        Signature type: SHA256withRSA
        Signature OID: 1.2.840.113549.1.1.11

        MD5 Fingerprint: FC AB 85 80 83 B4 93 27 7E 23 70 C2 56 98 D9 85
        SHA-1 Fingerprint: CC 6B 08 B1 88 7C 20 88 59 94 4F B8 99 0B D0 50 3B 5A B6 EC
        SHA-256 Fingerprint: 8B 56 5D F3 59 C6 72 41 99 5D 85 5E 47 F5 2B 03 22 8E 1A 8B 1B C2 5C 3F B4 70 9
```

## MSTG-CODE-2: The app has been built in release mode, with settings appropriate for a release build (e.g. non-debuggable).

**Result:** Pass **MASVS Reference:** MASVS 7.2

**Supporting Information:**
Before final distribution, the app should be built in its final release mode. Debug features, verbose logging, etc should be removed or minimized. When debugging, it may be necessary to report detailed information to the programmer. However, if the debugging code is not disabled when the application is operating in a production environment, then this sensitive information may be exposed to attackers.

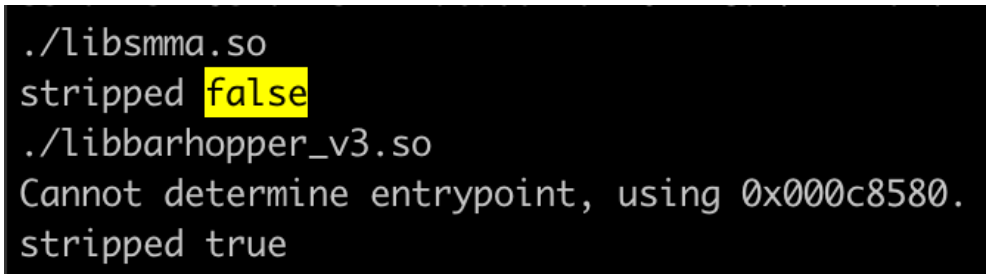
**Analyst Details:**
During analysis the AndroidManifest.xml was examined to determine if the android:debuggable="true" directive was present and no debuggable results were found.

## MSTG-CODE-3: Debugging symbols have been removed from native binaries.

**Result:** Pass **MASVS Reference:** MASVS 7.3

**Supporting Information:**
Generally, an app should have compiled code with as little explanation as possible. Some metadata, such as debugging information, line numbers, and descriptive function or method names, make the binary or bytecode easier for the reverse engineer to understand, but these aren't needed in a release build and can therefore be safely omitted without impacting the app's functionality.

**Analyst Details:**
During analysis libraries and architectures were examined for false stripped results to determine if debug symbols are present. The stripped results for the application are set to false for the library ./libsmma.so. More information is needed to determine if this library is used by a third party or native to the application.




***Note**: The developer has resolved this issue and debugging symbols have been removed from all native libraries.*

## MSTG-CODE-4: Debugging code and developer assistance code (e.g. test code, backdoors, hidden settings) have been removed. The app does not log verbose errors or debugging messages.

**Result:** Pass **MASVS Reference:** MASVS 7.4

**Supporting Information:**
To speed up verification and get a better understanding of errors, developers often include debugging code, such as verbose logging statements about responses from their APIs and about their application's progress and/or state. Furthermore, there may be debugging code for "management-functionality", which is used by developers to set the application's state or mock responses from an API. Reverse engineers can easily use this information to track what's happening with the application. Therefore, debugging code should be removed from the application's release version.

**Analyst Details:**
During analysis the application was exercised thoroughly and debug messages and debug code were not found during examination of the logs.

## MSTG-CODE-5: All third party components used by the mobile app, such as libraries and frameworks, are identified, and checked for known vulnerabilities.

**Result:** Pass **MASVS Reference:** MASVS 7.5

**Supporting Information:**
Apps often make use of third party libraries which accelerate development as the developer has to write less code in order to solve a problem. It's especially advantageous to reuse industry accepted cryptography libraries. However, third party libraries may contain vulnerabilities, incompatible licensing, or malicious content. It can be quite difficult for organizations and developers to manage application dependencies, including monitoring library releases and applying available security patches.

**Analyst Details:**
During analysis third party libraries were examined and it was determined no known vulnerable libraries were located.

## MSTG-CODE-9: Free security features offered by the toolchain, such as byte-code minification, stack protection, PIE support and automatic reference counting, are activated.

**Result:** Pass **MASVS Reference:** MASVS 7.9

**Supporting Information:**
In Android, decompiling Java classes is trivial. Because of this, applying some basic obfuscation to the release byte-code is recommended. ProGuard offers an easy way to shrink and obfuscate code and to strip unneeded debugging information from the byte-code of Android Java apps. It replaces identifiers, such as class names, method names, and variable names, with meaningless character strings. This is a type of layout obfuscation, which is "free" in that it doesn't impact the program's performance. R8 is the new code shrinker from Google and was introduced in Android Studio 3.3 beta. By default, R8 removes attributes that are useful for debugging, including line numbers, source file names, and variable names. R8 is a free Java class file shrinker, optimizer, obfuscator, and pre-verifier and is faster than ProGuard.
In iOS, Xcode enables all binary security features by default. However the following features may be unintentionally turned off:

- ARC (Automatic Reference Counting): A memory management feature that adds retain and release messages when required
- Stack Canary / Stack-smashing protection: Helps prevent buffer overflow attacks by means of having a small integer right before the return pointer. The value of the canary is always checked to make sure it has not changed before a routine uses the return pointer on the stack.
- PIE (Position Independent Executable): Enables full ASLR for the executable binary (not applicable for libraries).

It's important to know that the Stack Canary, if applied using `-fstack-protector` or `-fstack-protector-strong`, is heuristically applied by the compiler. This means that analysis of any native libraries may result in findings claiming that the canary was excluded, even though build settings are properly configured. This may be resolved by applying `-fstack-protector-all` which forces a canary on each function regardless of need, or a statement may be provided to be included in this report stating the build settings, or performance reasons why the canary was excluded.

**Analyst Details:**
During analysis it was determined the application has one library with a canary status set to false implying f-stack protection is not enabled for this library. More information is needed to determine if the library is native or third party.

```
./libsmma.so
canary   true
pic      true
./libbarhopper_v3.so
Cannot determine entrypoint, using 0x000c8580.
canary   true
pic      true
./libFido2Authenticator.so
canary   false
pic      true
./libdgrt.so
```

***Note**: The developer has resolved this issue and no libraries have stack canary set to false. Proper stack smashing protection is in place.*

# RELEASE INFORMATION

| | |
|---|---|
| Katie Bochnowski<br>SVP, Customer Success & Services<br>kbochnowski@nowsecure.com | Michael Krueger<br>Sr. Director, Application Security<br>mkrueger@nowsecure.com |