Aite PARTNER. ADVISOR. CATALYST.

# In Plain Sight: The Vulnerability Epidemic in Financial Mobile Apps

APRIL 2019

**Prepared for:**

# TABLE OF CONTENTS



# LIST OF FIGURES



# LIST OF TABLES

# EXECUTIVE SUMMARY

*In Plain Sight: The Vulnerability Epidemic in Financial Mobile Apps*, commissioned by Arxan and produced by Aite Group, examines the perceived security of financial mobile apps. It highlights the findings from a research campaign that Aite Group conducted over a six-week period to analyze financial institutions' (FIs') mobile apps across every vertical in financial services. The quantity and severity of the vulnerabilities discovered across the mobile apps clearly identify a systemic problem: a widespread absence of application security controls and secure coding, such as technology that implements application shielding, detection, and response capabilities.

Application shielding is a process in which the source code of an app is obfuscated, preventing adversaries from analyzing (aka decompiling) it to find vulnerabilities in the mobile app or repackage it to distribute it with malware. Application shielding also provides other enhanced security, such as application binding, repackaging detection, tamper detection, data-at-rest encryption, and key protection through white-box encryption. App-level threat detection identifies and alerts on exactly how and when apps are attacked at the code level in exercises such as the one Aite Group performed. And threat response can trigger immediate actions, such as shutting down an application, sandboxing a user, revising business logic, and repairing code.

In this research, the mobile applications were decompiled, meaning we reversed the app back to its original source code to assess vulnerabilities. When an app is capable of being decompiled, it allows adversaries to access sensitive information inside the source code—such as application programming interface (API) keys, API secrets, private certificates, and URLs that the app communicates with (which would allow an adversary to then target the APIs of the back-end servers)—recompile it to insert malware for later redistribution, and gain an understanding of how it detects jailbroken/rooted phones (Figure 1) so they can circumvent those checks and disable mandatory code signing and sandboxing.

**Figure 1: Checks for Jailbroken Device**

```
public static boolean isPhoneRooted() {
    String[] arrstring = Build.TAGS;
    if (arrstring != null && arrstring.contains("test-keys")) {
        return true;
    }
    for (String string2 : new String[]{"/sbin/", "/system/bin/", "/system/xbin/", "/data/local/xbin/", "/data/local/bin/", "/sys
tem/sd/xbin/", "/system/bin/failsafe/", "/data/local/"}) {
        StringBuilder stringBuilder = new StringBuilder();
        stringBuilder.append(string2);
        stringBuilder.append("su");
        if (!new File(stringBuilder.toString()).exists()) continue;
        return true;
    }
    return false;
}
```

*Source: Aite Group*

All of these threats stemming from the ability to decompile the app may lead to a range of exploits against FIs or their customers, including account takeovers, synthetic identity fraud, credit application fraud, identity theft, gift-card cracking, and credential stuffing attacks. Other adversarial campaigns can follow suit, such as reward abuse on retail sites and money laundering via online banking. There is no shortage of anecdotal evidence that hackers are actively seeking to leverage those vulnerabilities, such as the recent discovery in the wild of mobile malware that leveraged Android's accessibility features to copy the finger taps required to send money out of an individual's PayPal account. The malware was posted on a third-party app store disguised as a battery optimization app. This mobile banking trojan was designed to wire US$1,000 out of an individual's PayPal account within three seconds, despite PayPal's additional layer of security using multifactor authentication.

Some quick facts follow:

- The app category that implemented the most security controls was the cryptocurrency category.
- The app category that implemented the fewest security controls was the HSA bank category.
- The app category with the greatest number of critical vulnerabilities was the retail banking category
- The app category with the lowest number of critical vulnerabilities was the HSA bank category.
- The total number of critical vulnerabilities discovered across all app categories was 180.
- The FIs tested in the United States represented 85% of the vulnerabilities.
- The FIs tested in Europe represented 15% of the vulnerabilities. The total number of European company apps tested was five of the 30 apps tested.
- The three app categories with the highest number of critical vulnerabilities that can compromise the confidentiality, integrity, or availability of the user or FI were retail banking, retail brokerages, and auto insurance companies.
- The three app categories with the fewest critical vulnerabilities were HSA banks, health insurers tied with mobile payment apps, and credit card issuers.
- The app category with the most severe findings was auto insurance, as these apps contained the most number of hard-coded private keys, API keys, and API secrets in their code.

Key takeaways from the study include the following:

- Comprehensive mobile application shielding that includes code hardening, threat detection, and encryption capabilities is an important security layer ignored by many large and midsize financial institutions.

- The FIs with the most critical vulnerabilities in this research are the retail banking apps—a surprising finding considering the highly sensitive nature of checking accounts.
- The smallest number of vulnerabilities was found in the HSA apps.
- Several mobile banking apps hard-coded private certificates and API keys into their apps. Hackers could exploit this by copying the private certificates to their computers and running any number of free password-cracking programs against them. Should the hackers successfully crack the private key, they would be able to decrypt all communication between the back-end servers and mobile devices, among other things. The API keys allow an adversary to then begin targeting the FI's API servers, gaining them access to data in the back-end databases.
- Many of the apps contained hard-coded SQL statements that gave adversaries the ability to employ SQL injection attacks, such as modifying an existing SQL query or inserting a new SQL query in a man-in-the-middle attack that allows them to download all of the data in the database, delete data, or modify it.
- An API key uniquely identifies an application or user. The API key also acts as a secret token for authentication and also can define what the app or user is authorized to access. The API key should very much be considered somewhat akin to a password used for authentication. In this research study, it was discovered that several FIs and fintechs hard-coded API keys and secrets into their mobile apps for not only their services but also the API servers of third parties, such as cloud service providers and payment processors.
- Surprisingly, the smaller companies had the most secure development hygiene, while the larger companies produced the most vulnerable apps.

# INTRODUCTION

The purpose of this study is to highlight the systemic problem across the financial services industry of FIs' failure to properly secure their mobile apps at the code level through application shielding, encryption, and threat detection and response. When a company fails to implement proper application security technology for its app, it opens up the app to be easily reverse-engineered, potentially leading to account takeovers, data spills, and fraud. As a result, the company could experience significant financial losses and damage to brand, customer loyalty, and shareholder confidence as well as government penalties.

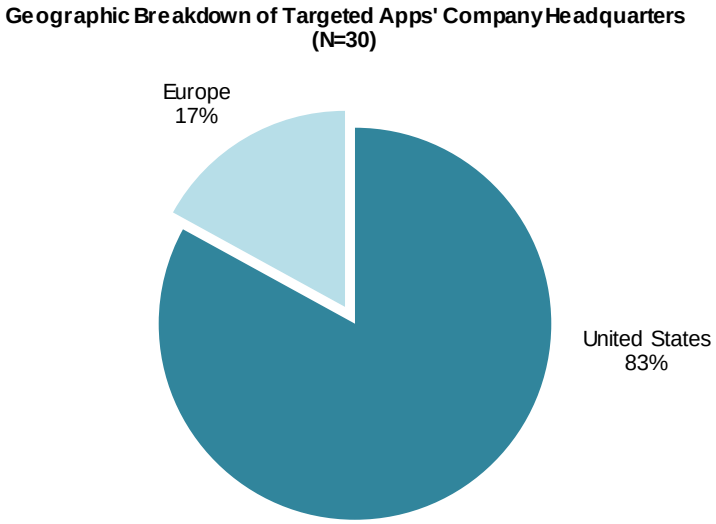
While the findings in this report are specific to these companies, many of them are systemic across all of the mobile apps tested, and other types of companies should use them as a guide for securing their mobile apps. The specific company name for each app and unique identifiers in screenshots of source code have been redacted.

## METHODOLOGY

This white paper discloses the vulnerabilities found in each mobile app, identifying a monumental assailable attack surface for these financial services companies that few of them, if any, are paying attention to. This paper details the specific vulnerabilities and consequences, accentuating the prodigious size of this problem to chief information security officers, chief privacy officers, chief risk officers, and senior business leaders across all of these financial services companies.

The size of the targeted companies differed in market capitalizations (from small companies to middle-market companies to companies with more than US$10 billion in market capitalization) across eight financial services sectors: retail banking, credit card, mobile payment, cryptocurrency, HSA, retail brokerage, health insurance, and auto insurance. Surprisingly, the companies with the largest market caps had the most vulnerable code, while the smaller companies produced apps with the fewest vulnerabilities. The apps tested are from companies with as few as 72 employees and from large multinational corporations with over 250,000 employees. The mobile apps that were tested were produced by companies headquartered in the U.S. and Europe (Figure 2).

**Figure 2: Headquarters of Companies That Published Targeted Apps**




*Source: Aite Group*

Empirical research leveraged the analysis of the mobile applications' static code. The apps were accessed via the Google Play store and downloaded using an LG G Pad X2 8.0 Plus Android tablet running Android version 7.0, patch level April 1, 2017, Kernel Version 3.18.31. APK Extractor was used to extract the mobile apps' Android application package (APK) files to a cloud drive, from which they were downloaded to the analyst's workstation for analysis.

Apktool version 2.3.4 was used to reverse-engineer Android binaries. Static code analysis was performed using MobSF version 1.0.5 Beta. Network interdiction for analysis of network traffic was performed using Burp Suite.

Table A and Table B list a decomposition of all the vulnerabilities found across all of the FI apps.

**Table A: Vulnerabilities Found Across All Financial Sectors**

| Type of app | Lack of binary protections | Insecure data storage | Unintended data leakage | Client-side injection | Weak encryption | Implicit trust of all certificates |
|---|---|---|---|---|---|---|
| **Retail bank** | ● | ● | ● | ◑ | ● | ◔ |
| **Credit card issuer** | ◕ | ● | ● | ◕ | ● | ○ |
| **Mobile payments** | ● | ● | ● | ◕ | ● | ○ |
| **HSA bank** | ● | ◕ | ◕ | ◑ | ◕ | ○ |
| **Retail brokerage** | ◕ | ◕ | ● | ◑ | ◕ | ◔ |

| Type of app | Lack of binary protections | Insecure data storage | Unintended data leakage | Client-side injection | Weak encryption | Implicit trust of all certificates |
|---|---|---|---|---|---|---|
| **Health insurer** | ● | ◑ | ◕ | ◔ | ◕ | ○ |
| **Auto insurer** | ● | ● | ● | ◑ | ◕ | ○ |
| **Cryptocurrency** | ◕ | ◕ | ◕ | ◔ | ◕ | ◔ |

*Source: Aite Group*
*Legend: ○ = 0% of the apps tested exhibited the vulnerability; ◔ = 25% of the apps tested exhibited the vulnerability; ◑ = 50% of the apps tested exhibited the vulnerability; ◕ = 75% of the apps tested exhibited the vulnerability; ● = 100% of the apps tested exhibited the vulnerability*

**Table B: Vulnerabilities Found Across All Financial Sectors (Continued)**

| Type of app | Execution of activities as root | World readable/ writable files and directories | Private key exposure | Exposure of database parameters and SQL queries | Insecure random number generation |
|---|---|---|---|---|---|
| **Retail bank** | ◑ | ◔ | ◔ | ● | ● |
| **Credit card issuer** | ◑ | ○ | ○ | ◕ | ● |
| **Mobile payments** | ○ | ○ | ○ | ● | ◑ |
| **HSA bank** | ◕ | ○ | ◔ | ◑ | ◑ |
| **Retail brokerage** | ◔ | ○ | ◔ | ◑ | ◕ |
| **Health insurer** | ◕ | ○ | ○ | ◑ | ◕ |
| **Auto insurer** | ◔ | ○ | ◑ | ◕ | ◕ |
| **Cryptocurrency** | ◕ | ○ | ◕ | ◑ | ◕ |

*Source: Aite Group*
*Legend: ○ = 0% of the apps tested exhibited the vulnerability; ◔ = 25% of the apps tested exhibited the vulnerability; ◑ = 50% of the apps tested exhibited the vulnerability; ◕ = 75% of the apps tested exhibited the vulnerability; ● = 100% of the apps tested exhibited the vulnerability*

# THE PROBLEM

Our analysis of the FIs' mobile apps highlights 11 types of vulnerabilities (Figure 3):

- Lack of binary protections
- Insecure data storage
- Unintended data leakage
- Client-side injection
- Weak encryption
- Implicit trust of all certificates
- Execution of activities using root
- World readable/writable files and directories
- Private key exposure
- Exposure of database parameters and SQL queries
- Insecure random number generation

**Figure 3: Vulnerability Findings Across All FI Apps**

| Lack of binary protections | | | | | | | |
|---|---|---|---|---|---|---|---|
| Retail bank | Credit card issuer | Mobile payments | HSA bank | Retail brokerage | Health insurer | Auto insurer | Crypto-currency |
| **Insecure data storage** | | | | | | | |
| Retail bank | Credit card issuer | Mobile payments | HSA bank | Retail brokerage | Health insurer | Auto insurer | Crypto-currency |
| **Unintended data leakage** | | | | | | | |
| Retail bank | Credit card issuer | Mobile payments | HSA bank | Retail brokerage | Health insurer | Auto insurer | Crypto-currency |
| **Client-side injection** | | | | | | | |
| Retail bank | Credit card issuer | Mobile payments | HSA bank | Retail brokerage | Health insurer | Auto insurer | Crypto-currency |
| **Weak encryption** | | | | | | | |
| Retail bank | Credit card issuer | Mobile payments | HSA bank | Retail brokerage | Health insurer | Auto insurer | Crypto-currency |
| **Implicit trust of all certificates** | | | | | | | |
| Retail bank | Retail brokerage | Crypto-currency | | | | | |
| **Execution of activities using root** | | | | | | | |
| Retail bank | Credit card issuer | HSA bank | Retail brokerage | Health insurer | Auto insurer | Crypto-currency | |
| **World readable/writable files and directories** | | | | | | | |
| Retail bank | | | | | | | |
| **Private key exposure** | | | | | | | |
| Retail bank | HSA bank | Retail brokerage | Auto insurer | Crypto-currency | | | |
| **Exposure of database parameters and SQL queries** | | | | | | | |
| Retail bank | Credit card issuer | Mobile payments | HSA bank | Retail brokerage | Health insurer | Auto insurer | Crypto-currency |
| **Insecure random number generation** | | | | | | | |
| Retail bank | Credit card issuer | Mobile payments | HSA bank | Retail brokerage | Health insurer | Auto insurer | Crypto-currency |

*Source: Aite Group*

## LACK OF BINARY PROTECTIONS

Ninety-seven percent of the apps tested suffered from a lack of binary protection, making it possible to decompile the apps and review the source code. Additionally, all of the FI apps tested failed to implement application security that would have obfuscated the source code of the apps, making it possible to decompile them. This provided all of the sensitive API URLs, API keys, and API secrets hard-coded into the apps, and some of the URLs included nonstandard port numbers and development servers used by developers for testing and QA, which were reachable at the time of the testing. By decompiling the binaries, it was also possible to discover several private keys hard-coded into their files and located in subdirectories of the app, making it possible to crack the private key passwords offline.

Additional findings included the ability to execute client-side code in an app's WebView; raw SQL queries embedded in the source code, yielding database schema information and the ability to perform SQL injection; the creation and storage of sensitive data into temp files on the mobile device or clipboard memory; and hard-coded public and private keys.

In addition to a better understanding of the FI's back end, decompiling the binary into its raw source code gives adversaries the ability to inject malware and repackage the app as a rogue/pirated app hosted in a third-party app market, such as TweakBox, Aptoide, and TutuApp, or send it to victims via smishing (SMS phishing). Decompiling the app also allows an adversary to understand how the app detects jailbroken mobile devices, which, once vulnerabilities (such as API keys, private keys, and credentials) are found in the source code, results in theft of money through banking trojans, username/password theft or account takeover using overlay screens, and the theft of confidential data.

An overlay screen attack occurs when mobile malware overlays HTML5 pages on top of a legitimate app to collect usernames and passwords or financial information, such as the case with the PayPal malware discovered in November 2018. The malware downloaded overlay screens for Gmail, Viber, Skype, WhatsApp, and Google Play in all of the screens, prompting users to enter their credit card numbers and credentials. One such overlay screen was discovered being used by malware for bank logins.

Many of these findings at the application layer would have been more difficult to make if application security at the code level (including code obfuscation) had been implemented, as the source code would have been unreadable.

## INSECURE DATA STORAGE

Eighty-three percent of the apps tested stored data insecurely, meaning outside of a sandbox and in the device's local file system, in external storage, or copied to the clipboard (Figure 4), allowing other apps to access it. Storing data in an insecure area allows users to access sensitive data that the app could have stored in temporary files or logs.

A systemic issue across all of the apps was the common practice among the FIs of depending on the mobile device's local or external storage for housing application data, including sensitive data inputted by the user. The storage on the mobile device is not a sandbox environment in which it's possible to carve that data off the storage system. In the event of an acquisition of the

mobile device by an adversary, this data is easily accessed, manipulated, and used to further penetrate the back-end servers of the FI through its APIs.

The threat facing APIs is a new attack surface being targeted by adversaries. Examples include the recent API breaches at Google, McDonald's, and Facebook, exposing customer names, email addresses, phone numbers, home addresses, and social media links.

To address this threat, FIs should ensure developers aren't writing code that stores sensitive data in insecure areas and instead are using sandboxes to store any temporary files or data where it can't be seen by the file system or other apps on the mobile device without being decrypted.

**Figure 4: Data Access via Clipboard**


*Source: Aite Group*

## UNINTENDED DATA LEAKAGE

Ninety percent of the apps tested shared services with other apps on the device, leaving the data from the FI's app accessible to any other application on the device. This allows adversaries to potentially leak data from the FI's app to apps within their control.

The data is stored in an area of the mobile device that is easily accessible by other apps or users, which could result in the breach of user privacy and lead to unauthorized use of data. Common leakage monitoring is not being performed across many of the apps analyzed, such as monitoring cache, logs, application background, HTML5 data storage, and browser cookie objects.

## CLIENT-SIDE INJECTION

Forty-three percent of the apps tested were vulnerable to client-side injection. This type of vulnerability refers to the execution of malicious code on the client side of the mobile device via the mobile app. This was discovered in the WebView component of the mobile apps, where the app could execute user-supplied code in the WebView window. During processing, the code forced a context switch, and the framework reinterpreted the data as executable code. In many cases, the code ran within the scope of the user's access permissions, and in some cases, the app supported the ability to run the code as the root/superuser account.

Client-side attacks create holes through which various functions of the mobile device can be accessed. An injection vulnerability of this magnitude might even allow adversaries to adjust trust settings on the device for the apps, even breaking out of sandboxes put in place as a security control.

## WEAK ENCRYPTION

Eighty percent of the apps tested implemented weak encryption algorithms, such as MD5, or the incorrect implementation of a strong cipher. This enables adversaries to decrypt sensitive data to its original form and manipulate or steal it as needed. This can result in complete dependence on built-in encryption processes, use of custom encryption protocols, use of insecure algorithms, etc. Several apps implemented poor key management, storing keys in easily accessible locations or hard-coding private keys within the app.

A weak cipher renders the encryption useless and provides adversaries full access to see or modify data in transit that would have been otherwise encrypted and not possible to crack.

## IMPLICIT TRUST OF ALL CERTIFICATES

Ten percent of the apps tested implicitly trust certificates presented to them without checks, making them vulnerable to man-in-the-middle attacks whereby a hacker sits between the mobile device and the FI's servers, presents a fake certificate to the user, and intercepts the user's traffic to the server and forwards it, thereby pretending to be both sides of the conversation. This allows an adversary to not only intercept and decrypt data but also modify data in transit between the user and FI, for example, modifying the bank account numbers or the transfer amount relevant to a money transfer.

## EXECUTION OF ACTIVITIES AS ROOT

Forty percent of the apps tested execute activities on the mobile device as the root user account, a superuser privilege level that grants the application access to all files, executables, and data on the device. The mobile application performs an operation at a privilege level higher than the minimum level required, creating potential for an adversary to disable the normal security checks that the operating system or surrounding environment performs.

This vulnerability allows an adversary to gain access to any resource allowed by the extra privilege—in the case of root, everything. As a result, the adversary could execute code, disable services, and read restricted data.

## WORLD READABLE/WRITABLE FILES AND DIRECTORIES

Three percent of the apps tested have directories or files configured with world readable and writable permissions, allowing any process or user to read and write to and from the files and directories.

## PRIVATE KEY EXPOSURE

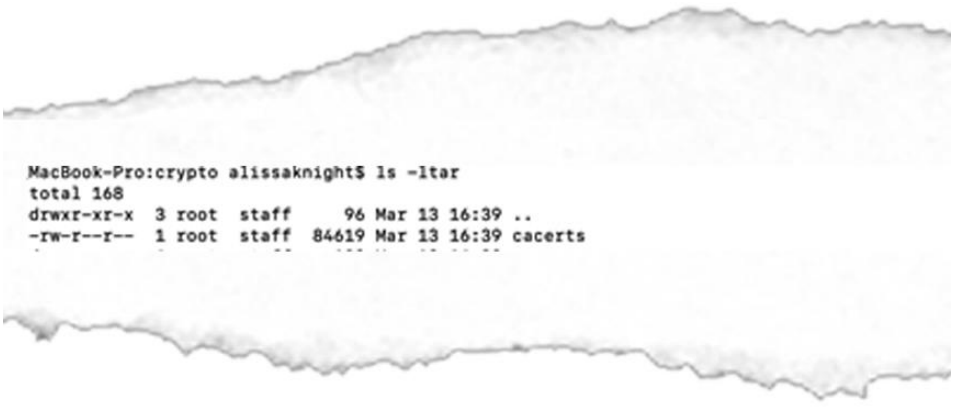
Twenty-seven percent of the apps tested hard-coded the API keys and private certificates in the apps or stored them in files on the file system.

This was found in the retail banking, HSA bank, retail brokerage, auto insurance, and cryptocurrency apps, which stored the private keys or API keys in a file on the mobile device (Figure 5, Figure 6, and Figure 7). Should a hacker gain access to this key file, offline cracking of the private key's password will be possible. Once the password is recovered, the hacker can use it to decrypt encrypted sessions between the app and back end after importing the key into the hacker's system keychain, giving the hacker access to sensitive financial data being encrypted in transit or to login credentials.

## Figure 5: Private Key File


*Source: Aite Group*

## Figure 6: File Containing All Private Keys




*Source: Aite Group*

**Figure 7: API Keys Hard-Coded in App**


*Source: Aite Group*

## EXPOSURE OF DATABASE PARAMETERS AND SQL QUERIES

Sixty percent of the apps tested exposed sensitive database parameters, configurations, and SQL queries in their code, allowing for SQL database query manipulation and injection (Figure 8).

**Figure 8: SQL Statements Inside Code**

```
    private static int \u02c8 = 0;
    private static final String \u02ca = "CREATE TABLE tbl_enhanced_token_info (_id INTEGER PRIMARY KEY AUTOINCREMENT, vPanEnrollmen
tID TEXT, vProvisionedTokenID TEXT, token_requester_id TEXT, encryption_metadata TEXT, tokenStatus TEXT, payment_instrument_last4 TE
XT, payment_instrument_expiration_month TEXT, payment_instrument_expiration_year TEXT, token_expirationDate_month TEXT, token_expira
tionDate_year TEXT, appPrgrmID TEXT, static_params TEXT, dynamic_key BLOB, mac_left_key BLOB, mac_right_key BLOB, enc_token_info BLO
B, dynamic_dki TEXT, token_last4 TEXT, token_par TEXT, mst_cvv TEXT, mst_svc_code TEXT, oda_data TEXT, icc_priv_key BLOB, pub_key_ex
piry INTEGER, token_type INTEGER, nic INTEGER, locate_sad_offset INTEGER, sdad_sfi INTEGER, sdad_rec INTEGER, sdad_offset INTEGER, s
dad_length INTEGER, car_data_offset INTEGER, decimalized_crypto_data BLOB, bouncy_submarine BLOB, UNIQUE (vProvisionedTokenID) ON CO
NFLICT REPLACE);";
    private static final String \u02cb;
    private static final String \u02ce = "CREATE TABLE tbl_data_group(_id INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT, token_key INTE
GER, dgi TEXT NOT NULL, dgi_data BLOB NOT NULL, FOREIGN KEY(token_key) REFERENCES tbl_enhanced_token_info (_id) ON DELETE CASCADE, U
NIQUE (token_key,dgi) ON CONFLICT REPLACE);";
    private static final String \u02cf = "CREATE TABLE tbl_tvl (_id INTEGER PRIMARY KEY AUTOINCREMENT, token_key INTEGER, timestamp
 NUMERIC, unpredictable_number TEXT, atc INTEGER, transaction_type TEXT, api TEXT, cryptogram TEXT, UNIQUE (atc,token_key) ON CONFLI
CT REPLACE, FOREIGN KEY(token_key) REFERENCES tbl_enhanced_token_info (_id) ON DELETE CASCADE );";
    private static final int \u037a = -1;
```

*Source: Aite Group*

## INSECURE RANDOM-NUMBER GENERATION

Seventy percent of the apps tested use an insecure random-number generator. These mobile apps use insufficiently random numbers or values in a security context that depends on unpredictable numbers.

# SOLUTION

To lower the risk of these vulnerabilities being identified and ultimately exploited, FIs must adopt a comprehensive approach to application security—including app shielding, encryption, and threat analytics—and ensure their developers receive adequate secure programming training and implement security in the software development life cycle when writing the code.

Comprehensive application security should protect and detect against reverse engineering threats, malware, debugging, emulators/fake execution environments, device cloning, root/jailbreak detection, code injection (runtime library injection), hooking frameworks, repackaging, system and user screenshots, keylogging using untrusted keyboards and screen scraping through untrusted screen readers, native code hooks, external screen sharing, man-in-the-app scenarios, man-in-the-middle scenarios, asset integrity checks, and overlay detection, as well as perform data-at-rest and data-in-transit encryption with secure key storage.

# CONCLUSION

- Despite the growing threat of bad actors targeting financial services businesses, FIs are still failing to write secure code and apply adequate application security technology, such as app shielding with code obfuscation, encryption, and threat analytics capabilities, to their mobile apps.
- The amount of sensitive data surrounding the FIs' API servers contained in these apps is alarming. This information should be obfuscated with an application security solution that includes white-box cryptography to protect API keys.
- FIs are still using rampant insecure coding when developing mobile apps and are hard-coding private keys and API secrets.
- FIs' mobile apps rarely use sandboxing to store sensitive data in secured/encrypted memory space; instead, they store data on external memory or the device itself.
- While only one of the FIs analyzed implemented API security, none of the FIs implemented any control mechanisms to detect whether the app was being reverse-engineered, which can be remediated by applying an application security solution with real-time threat detection to stop app-level attacks and generate alerts when they occur.

# ABOUT AITE GROUP

Aite Group is a global research and advisory firm delivering comprehensive, actionable advice on business, technology, and regulatory issues and their impact on the financial services industry. With expertise in banking, payments, insurance, wealth management, and the capital markets, we guide financial institutions, technology providers, and consulting firms worldwide. We partner with our clients, revealing their blind spots and delivering insights to make their businesses smarter and stronger. Visit us on the web and connect with us on Twitter and LinkedIn.

## AUTHOR INFORMATION

**Alissa Knight**
+1.206.765.7434
aknight@aitegroup.com

## CONTACT

For more information on research and consulting services, please contact:

**Aite Group Sales**
+1.617.338.6050
sales@aitegroup.com

For all press and conference inquiries, please contact:

**Aite Group PR**
+1.617.398.5048
pr@aitegroup.com

For all other inquiries, please contact:

info@aitegroup.com

# ABOUT ARXAN TECHNOLOGIES

Arxan, a global trusted leader providing the industry's most comprehensive application protection solutions, works with organizations looking to protect applications and to securely deploy and manage business-critical apps to the extended enterprise. Arxan currently protects more than one billion application instances across many industries, including financial services, mobile payments, medical devices, automotive, gaming, and entertainment. Unlike legacy security solutions that rely on perimeter-based barriers to keep bad actors out or that require device management controls, Arxan products protect at the application level from the inside out. This approach protects the source and binary code to expand the corporate perimeter of trust to the new endpoint—the application. Arxan provides a broad range of patented security capabilities, such as a dynamic app policy engine, code hardening, obfuscation, white-box cryptography and encryption, threat analytics, and rapid app protection deployment designed for DevOps processes. Founded in 2001, Arxan is headquartered in North America with global offices in EMEA and APAC. For more information, please visit our website or follow us on Twitter.