Kotlin Guide

Mobile Application Secure Coding Practices

CHECKMARX
choose what developers use

Table of Contents

This book is available for online reading or download in PDF, Mobi and ePub formats.

Introduction

Kotlin Secure Coding Practices is a guide written for anyone using Kotlin for mobile development.

This book is a collaborative effort started by Checkmarx Security Research Team, open sourced for community contributions. Its structure covers the OWASP Mobile Top 10 2016 intended to help developers avoid common mistakes.

Kotlin is a statically typed programming language for modern multiplatform applications 100% interoperable with Java™ and Android™, primarily developed by the team at JetBrains. It is now fully supported by Google as an alternative to the Android standard Java compiler.

Why This Book

Since May 7th 2019, Kotlin is Google's preferred language for Android app development. So, it is important for developers to familiarize with this new language.

Checkmarx Research Team helps educate developers, security teams, and the industry overall about common coding errors, and brings awareness of vulnerabilities that are often introduced during the software development process.

The Audience for this Book

The primary audience of the Kotlin Secure Coding Practices guide is Android developers. This guide can still be used by penetration testers to learn how to identify well-known vulnerabilities on Kotlin applications.

What You Will Learn

The authors of this book mapped the OWASP Mobile Top 10 security weaknesses to Kotlin on a weakness-by-weakness basis while providing examples, recommendations, and fixes to help developers avoid common mistakes and pitfalls. After reading this book and referring to it often, you will learn how to ensure you are developing secure mobile apps using Kotlin.

About Checkmarx

Checkmarx is the Software Exposure Platform for the enterprise. Over 1,800 organizations around the globe rely on Checkmarx to measure and manage software security risk at the speed of DevOps. Checkmarx serves five of the world’s top 10 software vendors, four of the top American banks, and many government organizations and Fortune 500 enterprises, including SAP, Samsung, and Salesforce.com. Learn more at checkmarx.com or follow us on Twitter: @checkmarx.

About OWASP Mobile Security Project

The OWASP Mobile Security Practices is a centralized resource intended to give developers and security teams the resources they need to build and maintain secure mobile applications.

The Mobile Top 10 2016 is the last edition of the top 10 most common mobile security weaknesses.

OWASP itself is "an open community dedicated to enabling organizations to conceive, develop, acquire, operate, and maintain applications that can be trusted. All of the OWASP tools, documents, forums, and chapters are free and open to anyone interested in improving application security".

How to Contribute

To learn how to contribute, please refer to CONTRIBUTING.md.

License

M1: Improper Platform Usage

From the Android documentation: "Content providers are one of the primary building blocks of Android applications, providing content to applications." Content providers are mostly used to share data between Android applications, such as activities, services or receivers. Content providers can have weak permissions or can be exported for all the apps on the device. Such misconfiguration can allow the Android app to leak the data used by the Content provider.

When the Content provider is exported, all the apps can query the Content provider to retrieve or modify the data. On the Goatlin app, we can see that a Content provider is defined with the name `.AccountProvider` and the exported tag is set to true. Here is an extract of the `AndroidManifest` file:

```
<provider
    android:name=".AccountProvider"
    android:authorities="com.cx.goatlin.accounts"
    android:enabled="true"
    android:exported="true" />
```

Looking at the code on the `AccountProvider.kt` class, we can retrieve the Content URI:

```
companion object {
    private val AUTHORITY = "com.cx.goatlin.accounts"
    private val ACCOUNTS_TABLE = "Accounts"
    val CONTENT_URI : Uri = Uri.parse("content://" + AUTHORITY + "/" +
                ACCOUNTS_TABLE)
    private val DATABASE_NAME = "data"
}
```

Then, using the `adb` tool, we can query this provider and even insert data. Here is simple example allowing a query to the provider in order to retrieve the accounts stored:

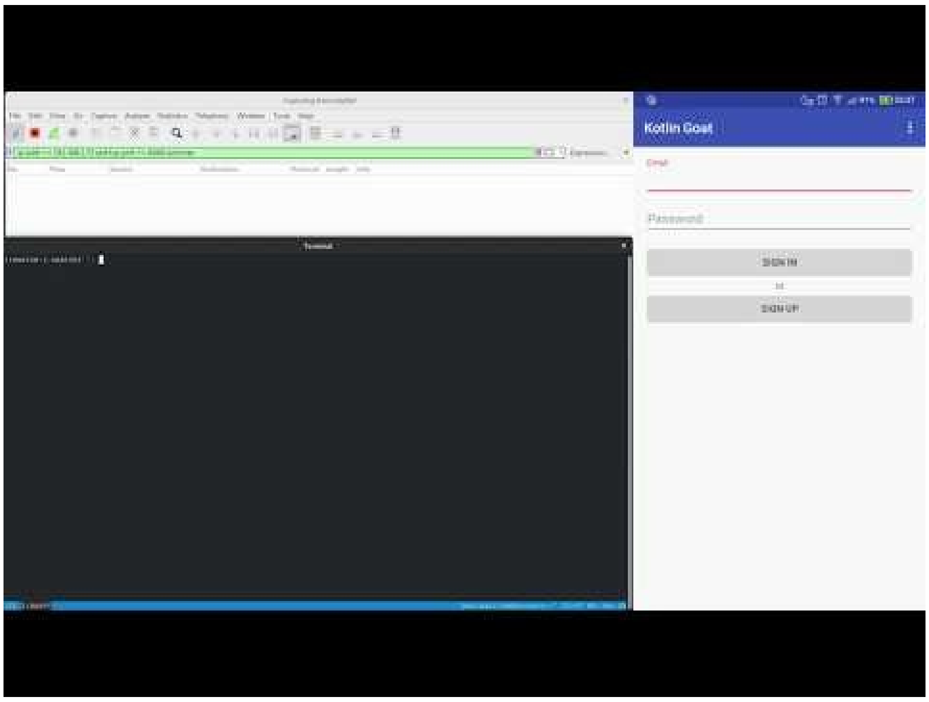
```
$ adb shell content query --uri content://com.cx.goatlin.accounts/Accounts
Row: 0 id=1, username=admin, password=admin
```

We can obtain the admin's credentials in this case.

Here is another example allowing an insertion to add a new account into the Goatlin app:

```
$ adb shell content insert --uri content://com.cx.goatlin.accounts/Accounts  --bind username:s:kotlin --bind pa
ssword:s:goat
$ adb shell content query --uri content://com.cx.goatlin.accounts/Accounts
Row: 0 id=1, username=admin, password=admin
Row: 1 id=2, username=kotlin, password=goat
```

Below you can see this is action



Video link

In the same manner, the app leaks a Content provider named `.NotesProvider` for the notes created by the user.

Here is the definition of the Content provider in the Android manifest:

```
<provider
    android:name=".NotesProvider"
    android:authorities="com.cx.goatlin.notes"
    android:enabled="true"
    android:exported="true"></provider>
```

As shown previously, we can obtain the Content URI by analyzing the `NotesProvider` class:

```
companion object {
    private val AUTHORITY = "com.cx.goatlin.notes"
    private val NOTES_TABLE = "Notes"
    val CONTENT_URI : Uri = Uri.parse("content://" + AUTHORITY + "/" +
                NOTES_TABLE)
    private val DATABASE_NAME = "data"
}
```

Again, with the `adb` tool, we can query this provider and retrieve the notes stored:

```
$ adb shell content query --uri content://com.cx.goatlin.notes/Notes
Row: 0 id=1, title=whvw, content=whvw, createdAt=2019-01-07 14:58:32, owner=1
```

For this provider, we can observe that the notes are not stored in clear text (the title and content of the note should be "test"). This encryption mechanism is further analyzed in the M5: Insufficient Cryptography section.

This issue was fixed on Goatlin: you can find it on feature/m1-improper-platform-usage branch.

More information about how to test improper platform usage can be found on the OWASP Mobile Testing Guide and especially on the Testing Platorm Interaction section.

Resources

Tools

- adb
- apktool
- jadx
- jd-gui

Readings

- Android documentation: Content Provider
- Testing Platform Interaction
- OWASP Mobile Testing Guide
- OWASP Mobile Top 10 2016: M1 - Improper Platform Usage

M2: Insecure Data Storage

The Android ecosystem provides several ways to store data for an app. The kind of storage used by developers depends on the kind of data stored, the usage of the data, and also whether the data should be kept private or shared to other apps.

The following solutions are provided by Android:

- **Internal file storage**: Files stored on the device and only available for the app.
- **External file storage**: Files usually stored on the SDCard (or any removable device). Files are available for everyone.
- **Databases**: Internal SQLite databases only available for the app.
- **Shared Preferences**: XML files mostly used as key-pair values to store configuration parameters.

Unfortunately, it is very common to find sensitive information stored in clear text. For instance, it is frequent to find API keys, passwords, Personally Identifiable Information (PII) stored on the Shared Preferences or databases used by the app.

In the case of the Goatlin app, when a user performs a sign up, the credentials are stored locally inside the database. Here is the extract of the SignupActivity class showing the creation of the account and how it is stored into the database:

```
/**
 * Attempts to create a new account on back-end
 */
private fun attemptSignup() {
    val name: String = this.name.text.toString()
    val email: String = this.email.text.toString()
    val password: String = this.password.text.toString()
    val confirmPassword: String = this.confirmPassword.text.toString()

    if (confirmPassword != password) {
        this.confirmPassword.error = "Passwords don't match"
        this.confirmPassword.requestFocus()
        return;
    }

    val account: Account = Account(name, email, password)

    val call: Call<Void> = apiService.signup(account)
    call.enqueue(object: Callback<Void> {
        override fun onFailure(call: Call<Void>, t: Throwable) {
            Log.e("SingupActivity", t.message.toString())
        }

        override fun onResponse(call: Call<Void>, response: Response<Void>) {
            val emailField: AutoCompleteTextView = findViewById(R.id.email)
            var message:String = ""

            when (response.code()) {
                201 -> {
                    if (createLocalAccount(account)) {
                        val intent = Intent(this@SignupActivity, LoginActivity::class.java)

                        startActivity(intent)
                    } else {
                        message = "Failed to create local account"
                    }
                }
                409 -> {
                    message = "This account already exists"
                    emailField.error = message
                    emailField.requestFocus()
                }
                else -> {
                    message = "Failed to create account"
                }
            }
        }
    }
}
```

When the app receives a response from the back-end with the HTTP code 201, the function `createLocalAccount()` is called. As shown below, this function only adds the username and password into the database:

```
/**
 * Creates local account
 */
private fun createLocalAccount(account: Account): Boolean {
    return DatabaseHelper(applicationContext).createAccount(account.email,
       account.password)
}
```

Here is the code of the `createAccount()` function provided by the `DatabaseHelper` class:

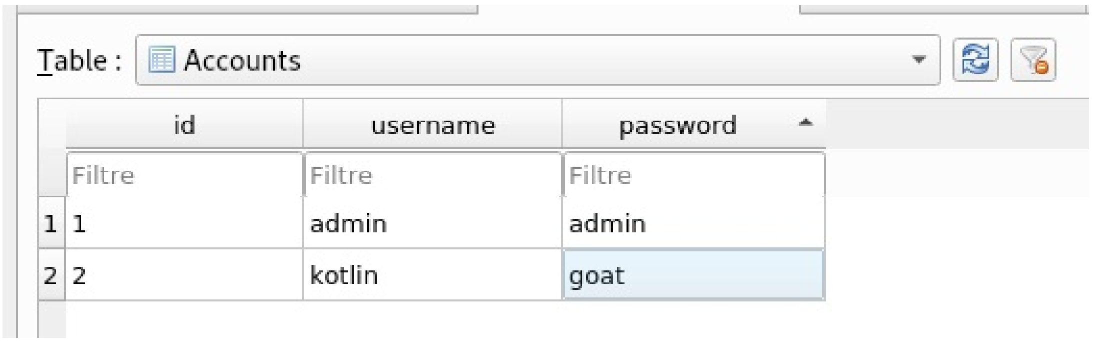
```
public fun createAccount(username: String, password: String) : Boolean {
    val db: SQLiteDatabase = this.writableDatabase
    val record: ContentValues = ContentValues()
    var status = true

    record.put("username", username) record.put("password", password)

    try {
        db.insertOrThrow(TABLE_ACCOUNTS, null, record)
    } catch (e: SQLException) {
        Log.e("Database signup", e.toString())
        status = false
    } finally {
        return status
    }
}
```

In the same manner, we can observe that the credentials are checked locally by retrieving the credentials stored inside the database. Here is an extract of the `LoginActivity` class where the check is made:

```
override fun doInBackground(vararg params: Void): Boolean? {
    if ((mUsername == "Supervisor") and (mPassword == "MySuperSecretPassword123!")){
        return true
    } else {
        try {
            val account: Account = DatabaseHelper(applicationContext).getAccount(mUsername)

            if (mPassword == account.password) {
                val prefs: SharedPreferences = applicationContext.getSharedPreferences(
                    applicationContext.packageName, Context.MODE_PRIVATE)
                val editor: SharedPreferences.Editor = prefs.edit()

                editor.putInt("userId", account.id).apply()
                editor.putString("userEmail", mUsername).apply()
            }

            return account.id > -1
        } catch(e: Exception) {
            return false
        }
    }
}
```

The app retrieves the account stored in the database by using the `getAccount()` function. If the password stored in the database matches the password provided by the user, the authentication is successful.

An attacker able to access the database of the app (rooting the device, backup of the app, etc.) can retrieve the credentials of the different users using the app. Using sqlitebrowser, it is easy to inspect the content of an SQLite database. Here is the content of the `Accounts` table used by the app:



	id	username	password
	Filtre	Filtre	Filtre
1	1	admin	admin
2	2	kotlin	goat

As discussed before, we can confirm that the passwords are stored in clear text without using any encryption mechanism.

Resources

Tools

- apktool
- jadx
- jd-gui
- sqlitebrowser

Readings

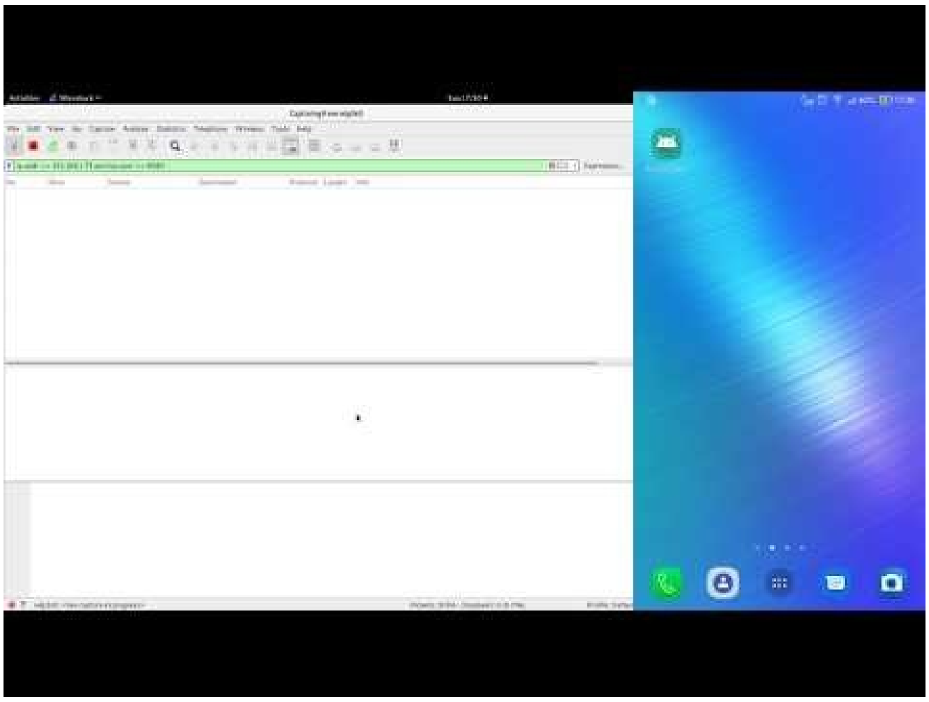
- Android documentation: Data and file storage overview
- OWASP Mobile Top 10 2016: M2 - Insecure Data Storage
- OWASP Mobile Security Testing guide - Test Data Storage

M3: Insecure Communication

Currently, most mobile applications exchange data in a client-server fashion at some point. When these communications happen, data traverses either the mobile carrier's network or between some Wi-Fi network and the internet.

Although exploiting the mobile carrier's network is not an impossible task, exploiting a Wi-Fi network is usually much easier. If communications lack SSL/TLS, then an adversary will be able not only to steal the data, but also to execute Man-in-the-Middle (MitM) attacks.

The following video demonstrates Insecure Communications exploitation on Kotlin Goat mobile application. The movie shows network monitoring, what gives an adversary access to exchanged data nevertheless, due to insecure communication, Man-in-the-Middle (MitM) would also be possible.



Video link

Now that we have seen the exploitation taking place, it's time to go back to the application source code and fix this issue. We will add SSL/TLS to all client-server communications and also implement Certificate Pinning to remove the "conference of trust" to no longer depend on Certificate Authorities or third-party agents regarding decisions on a server's identity.

To enable SSL/TLS we will need certificates to be available in the server. Nowadays you can get free certificates with Let's Encrypt - a free, automated and open Certificate Authority. You'll get the certificates deployed easily by following the documentation.

On Goatlin we'll go with a self-signed certificate. While this is a common practice during the development stage, it is not recommended for production systems. How to generate the certificate is out of scope for this guide.

With the certificate in hand, we should make a few changes on our back-end API to make it use HTTPS instead of HTTP

- Put `server.key` and `server.crt` under the `ssl` directory

- Replace `http` package with `https` one
- Load `server.key` and `server.crt`

```
const https = require('https');
const fs = require('fs');
const path = require('path');

// ...

/**
 * Create HTTP server.
 */
const sslDirectory = path.join(__dirname,'..','ssl');
const privateKey = fs.readFileSync(path.join(sslDirectory, 'server.key'), 'utf8');
const certficate = fs.readFileSync(path.join(sslDirectory, 'server.crt'), 'utf8');
const credentials = {key: privateKey, cert: certficate};

var server = https.createServer(credentials, app);

// ...
```

The following command line outputs our certificate fingerprint so that we can pin it on Goatlin:

```
openssl x509 -in server.crt -pubkey -noout | openssl pkey -pubin -outform der | openssl dgst -sha256 -binary |
openssl enc -base64
```

Now we have to modify the `create` method of our API service `Client` interface as shown below:

```
interface Client {
    @POST("accounts")
    fun signup (@Body data: Account): Call<Void>

    companion object {
        fun create(): Client {
            val certificatePinner = CertificatePinner.Builder()
                    .add("192.169.1.87:8080", "sha256/5Kl14sIBRoArZ8ujwNLWoLOI1QmsvE58nmXTO/9GSJw=")
                    .build()

            val client: OkHttpClient = OkHttpClient.Builder()
                    .certificatePinner(certificatePinner)
                    .build()

            val retrofit = Retrofit.Builder()
                    .addCallAdapterFactory(RxJava2CallAdapterFactory.create())
                    .addConverterFactory(GsonConverterFactory.create())
                    .baseUrl("https://192.168.1.87:8080")
                    .client(client)
                    .build()

            return retrofit.create(Client::class.java)
        }
    }
}
```

You can test Certificate Pinning by switching to feature/m3-insecure-communication branch. Replacing the back-end API certificates or the fingerprint on Goatlin source code will break the signup feature.

Resources

Tools

- Wireshark
- Burp Suite
- Let's Encrypt
- OkHttp

Readings

- Certificate Pinning
- OkHttp Certificate Pinning
- OWASP Mobile Top 10 2016: M3 - Insecure Communication

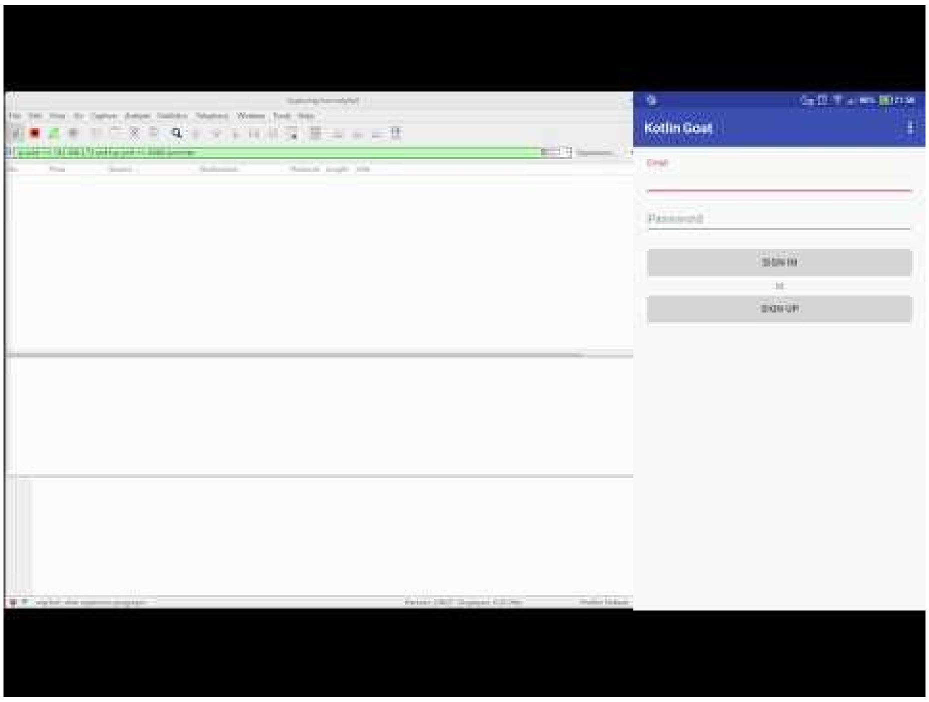
M4: Insecure Authentication

Weak authentication for mobile applications is fairly prevalent due to mobile devices' input factor: 4-digit PINs are a great example of it. Either a weak password policy due to usability requirements or authentication based on features like TouchID, make your application vulnerable. Contrary to what you may think, unlike passwords, you may be forced to give up your fingerprint.

Unless there's a functional requirement, mobile applications do not require a back-end server to which they should be authenticated in real-time. Even when such back-end servers exist, usually users are not required to be online at all times. This poses a great challenge on mobile applications' authentication. Whenever authentication has to happen locally, then it can be bypassed on jailbroken devices through runtime manipulation or modification of the binary.

Insecure Authentication is not only about guessable passwords, default user accounts, or data breaches. Under certain circumstances, the authentication mechanism can also be bypassed and the system will fail to identify the user and log its (malicious) activity. Usually in this scenario, user's will gain access to sensitive functionalities, since the system will also fail to validate its role, highlighting problems with the authorization controls as well.

The movie below shows an Insecure Authentication exploitation on Kotlin Goat



Video link

Now it is time to improve the application by establishing a strong password policy and storing authentication data safely. We will keep authentication data locally, since not all applications have a back-end server to handle it. When such a back-end exists, the password policy should be the same on both sides. Optionally, password "strength" validation can be delegated to the back-end.

The PasswordHelper object implements OWASP recommendations for Password Strength:

```
package com.cx.goatlin.helpers

object PasswordHelper {
    /**
     * Performs given password validation according to OWASP proper password strength
     * @link https://www.owasp.org/index.php/Authentication_Cheat_Sheet#Implement_Proper_Password_Strength_Cont
rols
     */
    fun strength (password: String): Boolean {
        var complexityRulesMatches: Int = 0

        if (!length(password)) {
            return false
        }

        // Password must meet at least 3 out of the following 4 complexity rules
        if (hasAtLeastOneUppercaseLetter(password)) {
            complexityRulesMatches++
        }

        if (hasAtLeastOneLowercaseLetter(password)) {
            complexityRulesMatches++
        }

        if (hasAtLeastOneDigit(password)) {
            complexityRulesMatches++
        }

        if (hasAtLeastOneSpecialChar(password)) {
            complexityRulesMatches++
        }

        if (complexityRulesMatches < 3) {
            return false
        }
        //

        if (!noMoreThanTwoIdenticalCharsInARow(password)) {
            return false
        }

        return true
    }
    // ...
}
```

`PasswordHelper.strength()` is then called from `signupAttempt()` on Goatlin `SignupActivity` (source):

```
package com.cx.goatlin
// ...
class SignupActivity : AppCompatActivity() {
    // ...
    private fun attemptSignup() {
        val name: String = this.name.text.toString()
        val email: String = this.email.text.toString()
        val password: String = this.password.text.toString()
        val confirmPassword: String = this.confirmPassword.text.toString()

        // test password strength
        if (!PasswordHelper.strength(password)) {
            this.password.error = """|Weak password. Please use:
                                |* both upper and lower case letters
                                |* numbers
                                |* special characters (e.g. !"#$%&')
                                |* from 10 to 128 characters sequence""".trimMargin()
            this.password.requestFocus()
            return;
        }
        // ...
    }
    // ...
}
```

Although the passwords are now stronger, they're still stored as clear text on a database. Someone with access to the device is still able to retrieve and manipulate database records. To address this issue, we will store a salted version of `username` and `password`.

In the case of password storage, OWASP recommends the following algorithms: bcrypt, PDKDF2, Argon2 and scrypt. These can enable hashing and salting passwords in a robust way.

We'll use `bcrypt`, which should be satisfactory for most situations. The advantages of `bcrypt` is that it's simpler to use. Therefore, it is less error-prone.

After adding jBCrypt as a dependency to have access to a `bcrypt` implementation, we just need to make two small changes to Goatlin. First is the `attemptSignup()` method of `SignupActivity` so that passwords are stored as a salted hash (source):

```
package com.cx.goatlin
// ...
class SignupActivity : AppCompatActivity() {
    // ...
    /**
     * Attempts to create a new account on back-end
     */
    private fun attemptSignup() {
        //...
        // hashing password
        val hashedPassword: String = BCrypt.hashpw(password, BCrypt.gensalt())
        val account: Account = Account(name, email, hashedPassword)
        // ...
    }
}
```

And the second one is the `UserLoginTask` `doInBackground()` method to compare a provided password with the stored one using `Bcrypt.checkpw()` method (source):

```
package com.cx.goatlin
// ...
class LoginActivity : AppCompatActivity(), LoaderCallbacks<Cursor> {
    // ...
    inner class UserLoginTask internal constructor(private val mUsername: String, private val mPassword: String
) : AsyncTask<Void, Void, Boolean>() {
        override fun doInBackground(vararg params: Void): Boolean? {
            if ((mUsername == "Supervisor") and (mPassword == "MySuperSecretPassword123!")){
                return true
            }
            else {
                val account:Account = DatabaseHelper(applicationContext).getAccount(mUsername)
                if (BCrypt.checkpw(mPassword, account.password)) {
                    // ...
                }
                // ...
            }
        }
    }
}
```

Keep in mind that this is just a brief overview of Insecure Authentication. Especially if you're doing local authentication, we highly recommend that you carefully read sections M8: Code Tampering and M9: Reverse Engineering.

Resources

Tools

- jBCrypt

Reading

- OWASP Authentication Cheat Sheet: Implement Proper Password Strength Controls
- BCrypt
- PBKDF2
- Argon2
- Scrypt
- OWASP Mobile Top 10 2016: M4 - Insecure Authentication

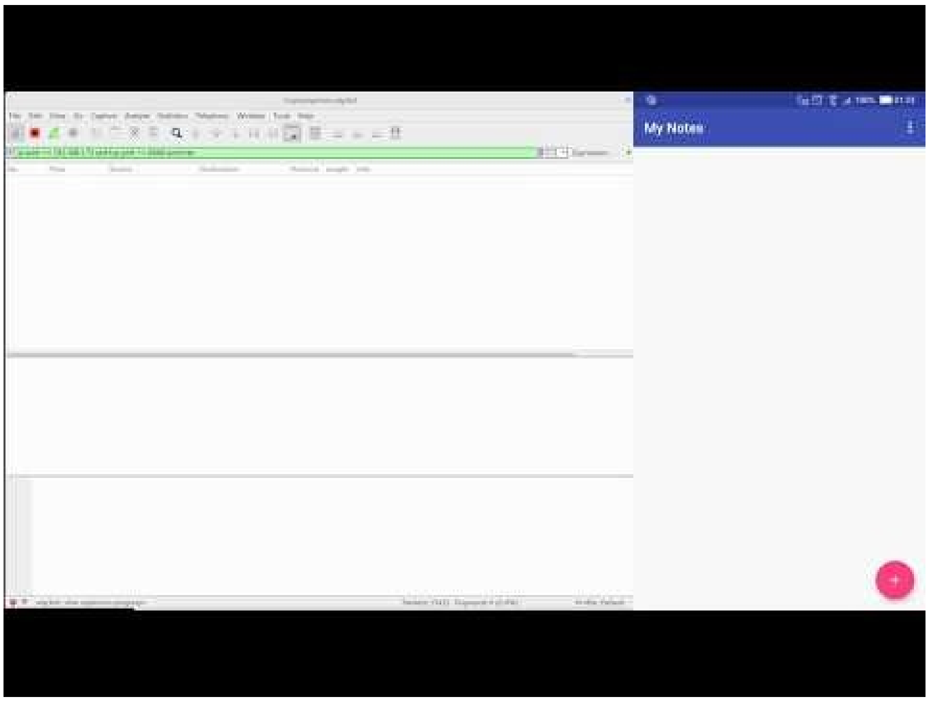
M5: Insufficient Cryptography

Let's assume that an application collects some Personal Identifiable Information (PII) which should be stored locally. Due to data relevance, it is encrypted. Now let's think about an adversary "physically attaining" the mobile device where such data is stored. The adversary will have access to all third-party application directories; therefore, they'll also have access to the stored data. In this scenario, whenever the adversary is able to return the encrypted data to its original unencrypted form, your cryptography was insufficient.

There are two fundamental mistakes in the development process leading to Insufficient Cryptography: either the encryption/decryption process relies on a flawless underlying process/library or the application may implement or leverage a weak encryption algorithm.

Keep in mind that encryption depends on secrets (keys) and even the best encryption algorithm will be useless if your application fails to keep its secrets by making the keys available to the attacker.

In the movie below you'll see how Goatlin cryptography fails by enabling the adversary to get the unencrypted version of stored data.



Video link

To address Insufficient Cryptography, we will replace the encryption algorithm by the AES - Advanced Encrypt Standard (Rijndael). As many other symmetric ciphers, AES can be implemented in different modes. In this case, we will use the GCM (Galoi Counter Mode). GCM is preferable to most popular CBC/ECB modes because the former is an authenticated cipher mode; meaning that after the encryption stage, an authentication tag is added to the ciphertext, which will then be validated prior to message decryption and ensuring the message has not been tampered with.

All major changes were done in the CryptoHelper class which was given two new methods: `createUserKey()` and `getUserKey()` . `encrypt()` and `decrypt()` methods were also changed to receive a `usernane` argument:

```kotlin
package com.cx.goatlin.helpers
// ...
class CryptoHelper {
    companion object {
        fun createUserKey(username: String) { /* ... */ }
        private fun getUserKey(username: String): SecretKey? { /* ... */ }
        fun encrypt(original: String, username: String): String { /* ... */ }
        fun decrypt(message: String, username: String): String { /* ... */ }
    }
}
```

As previously stated, encryption depends on secrets (keys), which should be handled carefully. In this case, on successful signup, a random key is created and persisted in Android Keystore. This key is user specific (see SignupActivity) and it is used to encrypt/decrypt a user's notes only.

Every time encryption/decryption is required, the `username` should be provided to the appropriate `CryptoHelper` method, since it is used as an alias to locate the user's key in Android Keystore (see CryptoHelper.getUserKey()):

```kotlin
package com.cx.goatlin.helpers
// ...
class CryptoHelper {
    companion object {
        private fun getUserKey(username: String): SecretKey? {
            val ks: KeyStore = KeyStore.getInstance("AndroidKeyStore").apply {
                load(null)
            }
            val entry = ks.getEntry(username, null) as? KeyStore.SecretKeyEntry
            // @todo handle null entry
            return entry?.secretKey
        }
    }
}
```

There is another implementation detail worth mentioning, since it may prove challenging. AES GCM encryption requires an Initialization Vector (IV). By default this is a random value. The value used during encryption should then be used on the corresponding decryption operation. Although randomness can be disabled (see `setRandomizedEncryptionRequired()`), replacing random IV by a constant value will reduce encryption security.

In our implementation we kept IV random, prepending it to the encrypted message. Then, while decrypting, the first 12 bytes correspond to the IV and the rest corresponds to the message. Note that IV is not secret.

Resources

Readings

- Android Keystore System
- Using the Android Keystore system to store and retrieve sensitive information
- Securely Storing Secrets in an Android Application
- OWASP Mobile Top 10 2016: M5 - Insufficient Cryptography

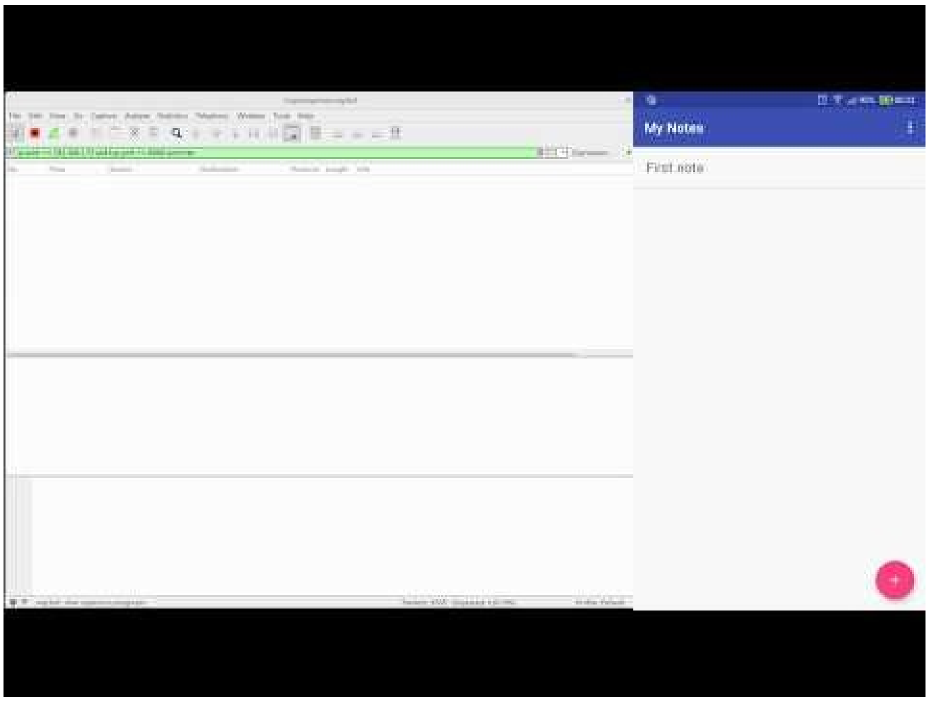
M6: Insecure Authorization

It is important to distinguish between authentication and authorization: the former is the act of identifying an individual whereas the later is the act of checking that the identified individual has the necessary permissions to perform the act. To exploit Insecure Authorization, adversaries usually log in to the application as a legitimate user first, then they typically force-browse to a vulnerable endpoint.

Because authentication precedes authorization, if an application fails to identify an individual before making an API request, then it automatically suffers from Insecure Authorization.

Usually Insecure Authorization is greatly associated with IDOR - Insecure Direct Object Reference but it is also found on hidden endpoints that developers assume will be accessed only by someone with the right role. If the mobile application sends the user role or permissions to the back-end as part of the request, it is likely vulnerable to Insecure Authorization.

The movie below demonstrates how Insecure Authorization can be exploited on Goatlin.



Video link

Insecure Authorization in Goatlin is clearly a back-end issue. Although API routes include **authentication** middleware when appropriate, no permissions (authorization) are validated:

```
router.put('/accounts/:username/notes/:note', auth, async (req, res, next) => {
    // ...
});

router.get('/accounts/:username/notes', auth, async (req, res, next) => {
    // ...
});
```

In this case, resources can be managed only by their owner. This is the validation that our authorization middleware will be responsible for:

```
function ownership (req, res, next) {
    if (req.params.username !== req.account.username) {
        res.statusMessage = "Unauthorized"
        return res.status(403).end();
    }

    next();
}
```

This is how API routes look like when they include both authentication and authorization middlewares:

```
router.put('/accounts/:username/notes/:note', [auth, ownership], async (req, res, next) => {
    // ...
});

router.get('/accounts/:username/notes', [auth, ownership], async (req, res, next) => {
    // ...
});
```

Resources

Readings

- Insecure Direct Object Reference Prevention Cheat Sheet
- Testing for Insecure Direct Object References (OTG-AUTHZ-004)
- Using middleware - Express
- OWASP Mobile Top 10 2016: M6 - Insecure Authorization

M7: Client Code Quality

This category includes code-level issues like a buffer overflow in C, or a DOM-based XSS in a Webview mobile app. It's usually something that requires code changes to be fixed, since it is caused by an improper API or language constructs usage.

Although Client Code Quality issues are prevalent, the exploitation often requires low-level knowledge. The typical primary goal is to execute foreign code within the mobile code's address space.

Consistent coding patterns and coding style guidelines broadly accepted in the organization will help to improve code quality. Since these issues are not easily detected on code review, using a static analysis tool usually provides the results. Buffer overflows and memory leaks should be top priorities over other code quality issues yet to be solved.

Detekt is one of such static code analysis tools for Kotlin licensed under the Apache License 2.0. It provides several integration mechanisms such as a Gradle plugin and a SonarQube integration, but it can also run standalone.

Running Detekt on Goatlin source code, results as bellow:

```
$ java -jar detekt-cli/build/libs/detekt-cli-1.0.0-RC12-all.jar -r txt:/tmp/goatlin.txt -i goatlin/packages/cli
ents/android/

Overall debt: 7h

Complexity Report:
        - 1230 lines of code (loc)
        - 912 source lines of code (sloc)
        - 610 logical lines of code (lloc)
        - 87 comment lines of code (cloc)
        - 145 McCabe complexity (mcc)
        - 47 number of total code smells
        - 9 % comment source ratio
        - 237 mcc per 1000 lloc
        - 77 code smells per 1000 lloc

Project Statistics:
        - number of properties: 149
        - number of functions: 75
        - number of classes: 18
        - number of packages: 5
        - number of kt files: 17
```

The report summary highlights several issues grouped by ruleset. Below are just the most relevant issues found:

```
Ruleset: complexity - 40min debt
        TooManyFunctions - 15/11 - [DatabaseHelper] at goatlin/packages/clients/android/app/src/main/java/com/c
x/goatlin/helpers/DatabaseHelper.kt:16:1
        ComplexMethod - 15/10 - [showProgress] at goatlin/packages/clients/android/app/src/main/java/com/cx/goa
tlin/LoginActivity.kt:126:5
Ruleset: exceptions - 1h 40min debt
        TooGenericExceptionCaught - [exception] at goatlin/packages/clients/android/app/src/main/java/com/cx/go
atlin/helpers/DatabaseHelper.kt:46:18
        TooGenericExceptionThrown - [installDatabaseFromAssets] at goatlin/packages/clients/android/app/src/mai
n/java/com/cx/goatlin/helpers/DatabaseHelper.kt:47:13
        TooGenericExceptionThrown - [getAccount] at goatlin/packages/clients/android/app/src/main/java/com/cx/g
oatlin/helpers/DatabaseHelper.kt:91:13
        TooGenericExceptionThrown - [getNote] at goatlin/packages/clients/android/app/src/main/java/com/cx/goat
lin/helpers/DatabaseHelper.kt:165:13
```

```
        TooGenericExceptionCaught - [e] at goatlin/packages/clients/android/app/src/main/java/com/cx/goatlin/Ed
itNoteActivity.kt:67:20
Ruleset: style - 4h 10min debt
        MagicNumber - [lowerBoundary] at goatlin/packages/clients/android/app/src/main/java/com/cx/goatlin/help
ers/CryptoHelper.kt:12:63
        WildcardImport - [LoginActivity.kt] at goatlin/packages/clients/android/app/src/main/java/com/cx/goatli
n/LoginActivity.kt:20:1
```

Resources

Tools

- Detekt
- SonarQube

Readings

- Kotlin Coding Conventions
- Kotlin style guide
- Idiomatic Kotlin. Best Practices.
- OWASP Mobile Top 10 2016: M7 - Client Code Quality

M8: Code Tampering

Once a mobile application is delivered and installed on a device, both the code and data will be available there. This gives the adversary the chance to directly modify the code, manipulate memory content, change or replace system APIs or simply modify application's data and resources. This is known as **Code Tampering**.

Rogue mobile applications play an important role in fraud-based attacks, becoming even more prevalent than malware. Typically attackers exploit code modification via malicious types of applications, tricking users to install the app via phishing attacks.

Notice that Kotlin has no advantage over plain Java when it comes to avoiding reverse engineering.

Technically, all mobile applications are vulnerable to code tampering but some are historically more targeted (e.g. mobile games) than others. Deciding whether or not to address this risk is a matter of business impact that can range from revenue loss to reputational damage.

OWASP Reverse Engineering and Code Modification Prevention Project is a great reference on how to detect and prevent Reverse Engineering and Code Modification. Generally speaking applications should be able to detect at runtime whether code was added or removed based upon what they know about their integrity at compile time.

To address this weakness on Goatlin we followed OWASP recommendation on Android Root detection. The `RootDetectionHelper` class implements a few techniques such as:

- Whether the kernel was signed with custom keys generated by a third-party developer:

```
private fun detectDeveloperBuild(): Boolean {
  val buildTags: String = Build.TAGS

  return buildTags.contains("test-keys")
}
```

- OTA certificates are available:

```
private fun detectOTACertificates(): Boolean {
  val otaCerts: File = File("/etc/security/otacerts.zip")

  return otaCerts.exists()
}
```

- Well-known applications to gain root access on Android devices are installed:

```
private fun detectRootedAPKs(ctx: Context): Boolean {
  val knownRootedAPKs: Array<String> = arrayOf(
      "com.noshufou.android.su",
      "com.thirdparty.superuser",
      "eu.chainfire.supersu",
      "com.koushikdutta.superuser",
      "com.zachspong.temprootremovejb",
      "com.ramdroid.appquarantine"
  )
  val pm: PackageManager = ctx.packageManager

  for(uri in knownRootedAPKs) {
      try {
          pm.getPackageInfo(uri, PackageManager.GET_ACTIVITIES)
```

```
            return true
        } catch (e: PackageManager.NameNotFoundException) {
            // application is not installed
        }
    }

    return false
}
```

- `su` binary is available:

```
private fun detectForSUBinaries(): Boolean {
    var suBinaries: Array<String> = arrayOf(
        "/system/bin/su",
        "/system/xbin/su",
        "/sbin/su",
        "/system/su",
        "/system/bin/.ext/.su",
        "/system/usr/we-need-root/su-backup",
        "/system/xbin/mu"
    )

    for (bin in suBinaries) {
        if (File(bin).exists()) {
            return true
        }
    }

    return false
}
```

- Attempt to run `su` and check the id of current user:

To prevent the application to run on a Rooted environment, the `RootDetectionHelper.check()` method, which combines all the described techniques, is called on our main activity (Login). If a Rooted environment is detected then the user is presented a dialog and the application is forced to close:

```
package com.cx.goatlin
// ...
class LoginActivity : AppCompatActivity(), LoaderCallbacks<Cursor> {
    // ...
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_login)

        if (RootDetectionHelper.check(applicationContext)) {
            forceCloseApp()
        }
        //  ...
    }
    // ...
    private fun forceCloseApp() {
        val dialog: AlertDialog.Builder = AlertDialog.Builder(this)

        dialog
                .setMessage("The application can not run on rooted devices")
                .setCancelable(false)
                .setPositiveButton("Close Application", DialogInterface.OnClickListener {
                    _, _ -> finish()
                })

        val alert: AlertDialog = dialog.create()

        alert.setTitle("Unsafe Device")
        alert.show()
    }
    //...
}
```

Resources

Readings

- OWASP Reverse Engineering and Code Modification Prevention Project
- Android Root Detection Techniques
- OWASP Mobile Top 10 2016: M8 - Code Tampering

M9: Reverse Engineering

One of the first steps when performing security assessments on Android applications is to perform static analysis of the app in order to understand its internals such as:

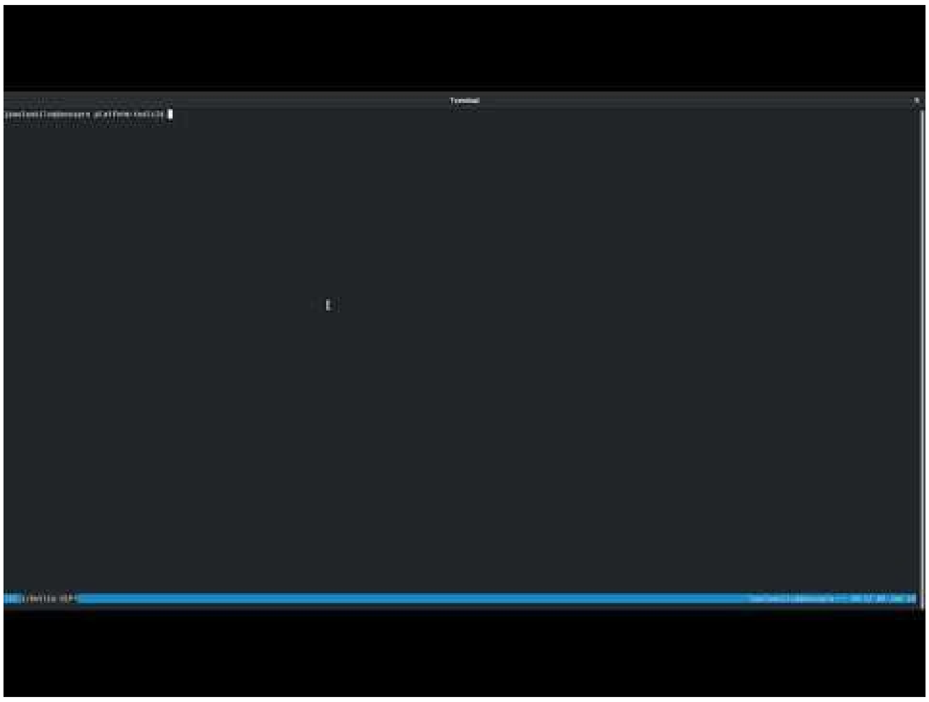
- How is it working?
- What kind of communications are established?
- Which libraries are used?

Several tools are available to perform this task such as apktool, jadx or jd-gui.

Apktool is a decompiling tool, allowing to obtain the smali bytecode of the app, which is executed by the Dalvik virtual machine. The code obtained is very low level, but it permits a deep overview of the app internals. Here is an example of smali bytecode:

```
# static fields
.field static final synthetic $$delegatedProperties:[Lkotlin/reflect/KProperty;

# instance fields
.field private _$_findViewCache:Ljava/util/HashMap;
.field private final apiService$delegate:Lkotlin/Lazy;
.field private listView:Landroid/widget/ListView;

# direct methods
.method static constructor <clinit>()V
    .locals 5
    const/4 v0, 0x1
    new-array v0, v0, [Lkotlin/reflect/KProperty;
    new-instance v1, Lkotlin/jvm/internal/PropertyReference1Impl;
    const-class v2, Lcom/cx/goatlin/HomeActivity;
    invoke-static {v2}, Lkotlin/jvm/internal/Reflection;->getOrCreateKotlinClass(Ljava/lang/Class;)Lkotlin/refl
ect/KClass;
    move-result-object v2
    const-string v3, "apiService"
    const-string v4, "getApiService()Lcom/cx/goatlin/api/service/Client;"
    invoke-direct {v1, v2, v3, v4}, Lkotlin/jvm/internal/PropertyReference1Impl;-><init>(Lkotlin/reflect/KDecla
rationContainer;Ljava/lang/String;Ljava/lang/String;)V
    invoke-static {v1}, Lkotlin/jvm/internal/Reflection;->property1(Lkotlin/jvm/internal/PropertyReference1;)Lk
otlin/reflect/KProperty1;
    move-result-object v1
    check-cast v1, Lkotlin/reflect/KProperty;
    const/4 v2, 0x0
    aput-object v1, v0, v2
    sput-object v0, Lcom/cx/goatlin/HomeActivity;->$$delegatedProperties:[Lkotlin/reflect/KProperty;
    return-void
.end method
```

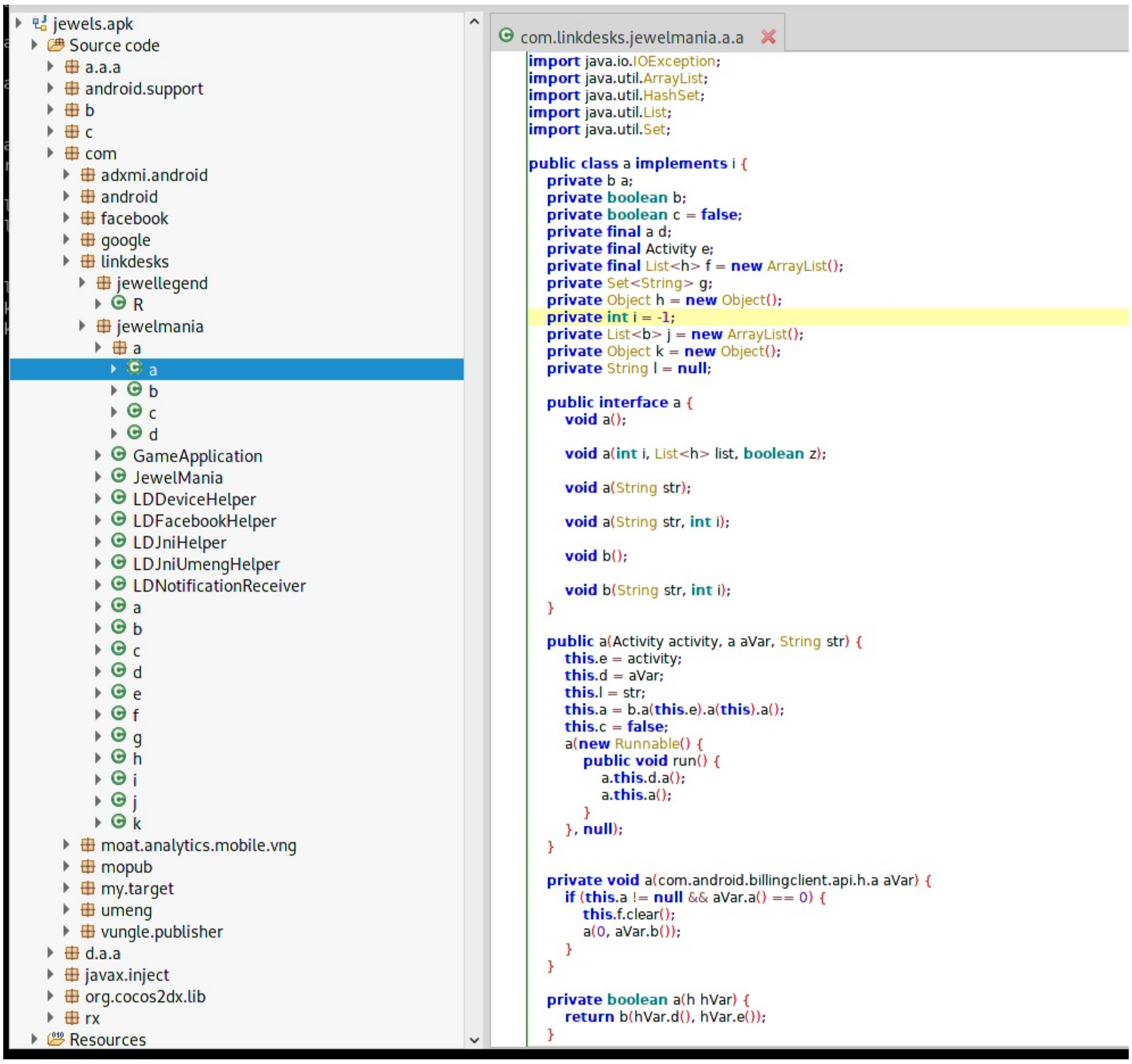
In order to obtain Java code, an attacker can use jadx or jd-gui to decompile the app. The video below demonstrates it using jadx on Kotlin Goat



Video link

As you can see, there is no obfuscation at all on the Goatlin. An attacker is able to easily analyze the app in order to understand the inner mechanisms.

In order to slow down the process of reverse engineering, developers use various techniques to obfuscate the code such as renaming the variables and name functions with weird names or using a non-Latin charset. Here is an example where all the variables and method names were renamed:



```
jewels.apk
  Source code
    a.a.a
    android.support
    b
    c
    com
      adxmi.android
      android
      facebook
      google
      linkdesks
        jewellegend
          R
        jewelmania
          a
            a
            b
            c
            d
          GameApplication
          JewelMania
          LDDeviceHelper
          LDFacebookHelper
          LDJniHelper
          LDJniUmengHelper
          LDNotificationReceiver
          a
          b
          c
          d
          e
          f
          g
          h
          i
          j
          k
      moat.analytics.mobile.vng
      mopub
      my.target
      umeng
      vungle.publisher
    d.a.a
    javax.inject
    org.cocos2dx.lib
    rx
  Resources
```

com.linkdesks.jewelmania.a.a

```java
import java.io.IOException;
import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.Set;

public class a implements i {
    private b a;
    private boolean b;
    private boolean c = false;
    private final a d;
    private final Activity e;
    private final List<h> f = new ArrayList();
    private Set<String> g;
    private Object h = new Object();
    private int i = -1;
    private List<b> j = new ArrayList();
    private Object k = new Object();
    private String l = null;

    public interface a {
        void a();

        void a(int i, List<h> list, boolean z);

        void a(String str);

        void a(String str, int i);

        void b();

        void b(String str, int i);
    }

    public a(Activity activity, a aVar, String str) {
        this.e = activity;
        this.d = aVar;
        this.l = str;
        this.a = b.a(this.e).a(this).a();
        this.c = false;
        a(new Runnable() {
            public void run() {
                a.this.d.a();
                a.this.a();
            }
        }, null);
    }

    private void a(com.android.billingclient.api.h.a aVar) {
        if (this.a != null && aVar.a() == 0) {
            this.f.clear();
            a(0, aVar.b());
        }
    }

    private boolean a(h hVar) {
        return b(hVar.d(), hVar.e());
    }
```

The most well known tool to perform code obfuscation is Proguard. Quoting Wikipedia: "ProGuard is an open source command-line tool that shrinks, optimizes and obfuscates Java code. It is able to optimize bytecode as well as detect and remove unused instructions."

The Android documentation provides guidance on how to shrink your code and resources. In this article, you can find explanations on how to enable Proguard with Android Studio.

Resources

Tools

- apktool
- jadx
- jd-gui
- Proguard

Readings

- Android documentation: Shrink your code and resources

- Enabling Proguard for Android
- OWASP Mobile Top 10 2016: M9 - Reverse Engineering

M10: Extraneous Functionality

As the name suggests, extraneous functionality are functions or secrets hidden inside the app. Those functionalities allow an attacker to perform unintended actions such as:

- Accessing administrative or debug functions
- Retrieving hidden secrets (API keys, account credentials, personal information, etc)
- Discovering hidden back-end endpoints

In the case of the Goatlin app, a backdoor account is hardcoded into the code as shown below:

```
inner class UserLoginTask internal constructor(private val mUsername: String,
        private val mPassword: String) : AsyncTask<Void, Void, Boolean>() {

    override fun doInBackground(vararg params: Void): Boolean? {
        if ((mUsername == "Supervisor") and (mPassword == "MySuperSecretPassword123!")){
            return true
        }
        else {
            val account:Account = DatabaseHelper(applicationContext).getAccount(mUsername)
            if (mPassword == account.password) {
                // ...
            }
            // ...
        }
        // ...
    }
    // ...
}
```

When performing static analysis using apktool, jadx, or jd-gui, an attacker is able to retrieve those credentials and then use them to obtain access to the application.

Using apktool, it is possible to discover those credentials. On the `LoginActivity$UserLoginTask.smali` file, an attacker can identify the hardcoded account:

```
# virtual methods
.method protected varargs doInBackground([Ljava/lang/Void;)Ljava/lang/Boolean;
    .locals 7
    .param p1, "params"    # [Ljava/lang/Void;
        .annotation build Lorg/jetbrains/annotations/NotNull;
        .end annotation
    .end param
    .annotation build Lorg/jetbrains/annotations/Nullable;
    .end annotation
    const-string v0, "params"
    invoke-static {p1, v0}, Lkotlin/jvm/internal/Intrinsics;->checkParameterIsNotNull(Ljava/lang/Object;Ljava/l
ang/String;)V
    .line 218
    iget-object v0, p0, Lcom/cx/goatlin/LoginActivity$UserLoginTask;->mUsername:Ljava/lang/String;
    const-string v1, "Supervisor"
    invoke-static {v0, v1}, Lkotlin/jvm/internal/Intrinsics;->areEqual(Ljava/lang/Object;Ljava/lang/Object;)Z
    move-result v0
    iget-object v1, p0, Lcom/cx/goatlin/LoginActivity$UserLoginTask;->mPassword:Ljava/lang/String;
    const-string v2, "MySuperSecretPassword123!"
    invoke-static {v1, v2}, Lkotlin/jvm/internal/Intrinsics;->areEqual(Ljava/lang/Object;Ljava/lang/Object;)Z
    move-result v1
```

Another way is to use the jadx tool. Then, when looking at the `LoginActivity` class, we can find the following decompiled code:

```
@Nullable
protected Boolean doInBackground(@NotNull Void... params) {
    Intrinsics.checkParameterIsNotNull(params, "params");
    boolean z = true;
    if ((Intrinsics.areEqual(this.mUsername, (Object) "Supervisor") & Intrinsics.areEqual(this.mPassword, (Obje
ct) "MySuperSecretPassword123!")) != 0) {
        return Boolean.valueOf(true);
    }
    // ...
}
```

Resources

Tools

- apktool
- jadx
- jd-gui

Readings

- OWASP Mobile Top 10 2016: M10 - Extraneous Functionality

Final Notes

The Checkmarx Research team is confident that this Kotlin Secure Coding Practices Guide provided value to you. We encourage you to refer to it often, as you're developing Android apps written in Kotlin. The information found in this guide can help you develop more-secure apps and avoid the common mistakes and pitfalls that lead to vulnerable applications. Understanding that exploitation techniques are always evolving, new vulnerabilities might be found in the future, based on dependencies that may make your application vulnerable.

Since the OWASP Mobile Top 10 changes every few years, we recommend staying abreast of the following:

- OWASP Mobile Top 10 2016
- OWASP Mobile Testing Guide
- Check OWASP Cheat Sheet Series