2024

Android Geliştiricileri İçin Güvenlik Rehberi

Murat Yüksektepe

v.1.0.1

https://muratyuksektepe.com

İçindekiler

1. Uygulama İçi Güvenlik 2
1.1. Kod Güvenliği 2
1.1.1. ProGuard ile Kod Karıştırma 2
1.1.2. C ve C++ Kütüphaneleri Kullanımı 4
1.1.3. Debug ve Log'ları Kapatmak 4
1.1.4. Root Erişimi Kontrolü 5
1.2. Veri Güvenliği 6
1.2.1. Geçici Dosyalar 6
1.2.2. Kayıtlı Dosya Korunmuşluk Kontrolü 8
1.2.3. Tek Yönlü Veri Şifreleme 9
1.2.4. EncryptedSharedPreferences 10
1.2.5. API Key Güvenliği 12
1.2.5.1. BuildConfig 12
gradle.properties 13
local.properties 14
apikey.properties 15
build.gradle İçerisinde Oluşturduğumuz Değişkene Ulaşmak 16
1.2.5.2. Android NDK 18
1.2.5.3. Sunucu Tarafında Veri Saklama 18
2. Uygulama - Web Servisleri İletişim Güvenliği 19
2.1. TLS Kullanımı 19
2.2. Özelleştirilmiş Network Ayarları 19
2.2.1. SSL/TLS Pinning 20
Man-in-the-middle Saldırısı 21
Kendinden İmzalı Sertifika İle Çalışmak 23
3. Uygulama - Uygulama İletişim Güvenliği 24
3.1. ContentProvider'a Erişimi Engellemek 24
3.2. Aynı İmza Kontrolü 24
3.3. AES ile Şifrelenmiş Veri Paylaşımı 26
4. Uygulama - Kullanıcı İletişimi Güvenliği 28
4.1. Hassas Veriler Alınırken Dikkat Edilmesi Gerekenler 28
4.1.1. Parolaların Zorluk Seviyesi 28
4.2. Hassas Verilerin Gösterilmesi 28
4.2.1. Parola, Pin, Özel Soru veya Biyometrik Veri ile Doğrulama 29
4.2.2. Kullanıcının Eğitilmesi 29

1. Uygulama İçi Güvenlik

Bu bölümde uygulamamızı geliştirirken ve yayınlarken uygulayabileceğimiz güvenlik adımlarını göreceğiz. Uygulama içi güvenliğin önemini daha iyi anlamak için benim tavsiyem uygulamamızın küçük veya büyük olması, kullanıcı sayının az ya da çok olması fark etmeksizin aklımızın bir kenarında sürekli olarak "kötü amaçlı kişi veya kişilerin uygulamamızı tersine mühendislik ile çözmeye ve uygulamamız için çok önemli verilere ulaşmaya çalışacağı" düşüncesinin olmasıdır. Bu şüphe ile hareket etmek bizlere alabileceğimiz her güvenlik önlemini araştırmak, öğrenmek ve uygulamak için motivasyon sağlamalıdır. Uygulama içi güvenlik konusunda 2 adet önemli başlık bulunmaktadır.

1.1. Kod Güvenliği

Tersine mühendislik ile uygulamanın çözülmesi durumunda ilk görülebilecek dosyaları oluşturan kodlarımızdır. Burada amacımız öncelikle kodların görülmemesini sağlamak olmalıdır. Fakat bugün biliyoruz ki şifreleme teknolojileri ne kadar gelişiyorsa şifre çözme teknolojileri de bir o kadar gelişmektedir. Şu anda bir Android uygulama paket dosyasını (APK, AAB) tersine mühendisliğe tamamen kapatacak bir yöntem bulunmamaktadır. Bu sebeple kodların görülme ihtimaline karşı başvurmamız gereken ve kötü amaçlı kişi veya kişilerin işlerini zorlaştıracak yöntemleri incelememiz faydalı olacaktır.

1.1.1. ProGuard ile Kod Karıştırma

Aslında bu tam anlamıyla bir koruma yöntemi değildir. Kod karıştırma yöntemi kısaca; kullanılmayan kodların derlenme sırasında elenmesi, bizim yazdığımız fonksiyon ve sınıf isimlerini, değişkenleri tanımlarken kullandığımız metinleri mümkün olduğunca kısaltarak uygulama boyutunu küçültmek ve çözülmüş kodun okunabilirliğini zorlaştırmaktır.

Kod karıştırma yöntemi uygulanmış dosyaların tersine mühendislik ile açılmış hallerinde yukarı bahsettiğimiz kısaltılmış isimlerin geri gelmediği ve bıraktığımız yorum satırlarının kaybolduğu görülmektedir. Bu durum kodun okunabilirliğini azaltmasına rağmen Java veya .Net gibi byte-code tabanlı diller için hazırlanmış olan decompiler araçlarında yer alan isimsel ilişkilendirme özelliği, tersine mühendislik sırasında kodun -rahat olmasa bile- okunabilmesini ve akışın takip edilebilmesini sağlar. [1]

Android uygulama geliştirirken kod karıştırma yapılandırması oldukça basittir. Uygulama seviyesindeki `build.gradle` dosyasında yer alan `isMinifyEnabled` değerini `release` ya da `debug` modlar için `true` olarak işaretlemek yeterli olacaktır:

```
android {
  …
    buildTypes {
        release {
            isMinifyEnabled = true
            proguardFiles(getDefaultProguardFile("proguard-android-optimize.txt"), "proguard-rules.pro")
        }
    }
}
```

Android, kod karıştırma işlemi, kullanılmayan kodların elenmesi işlevini gerçekleştiren R8 aracını kullanır. R8 ise fonksiyon, değişken ve sınıf isimlerinin kısaltılması için ProGuard aracını ve bizim özelleştirdiğimiz ProGuard kurallarını kullanır.

R8, bazı durumlarda hangi kodun uygulama içerisinde gerçekten kullanılıp-kullanılmadığı konusunda hata yapabilir.[2] (*Aynı durum uygulama içerisinde kullanılmayan kaynak dosyalarını elemeyle görevli olan `shrinkResource` ayarı içinde mevcuttur)* Bu gibi durumlarda uygulamanın ihtiyacı olan ama R8 tarafından elenen dosyaların ProGuard kurallar dosyasında kod karıştırma işleminden çıkarılması gerekmektedir. Bunu yapmak için uygulamamız içinde ilgili modülün ana dizininde yer

[1] https://www.baeldung.com/cs/code-obfuscation-benefits
[2] https://developer.android.com/build/shrink-code#keep-code

alan `proguard-rules.pro` dosyası içerisinde hata aldığımız dosyanın `-keep` ile işaretlenmesi ya da ilgili kodların başına `@Keep` tanımlamasını eklemek gerekir.

Not: `@Keep`'i bir sınıfın başına eklerseniz bütün sınıf kod karıştırmadan hariç tutulacaktır. Bir değişken ya da fonksiyon için sadece o nesnenin başına eklemelisiniz.

Not 2: ProGuard, `res` klasörü altında yer alan projemizin kaynak dosyalarına kod karıştırma işlemini <u>uygulamaz</u>. Bu yüzden Proguard kullanıyor olsanız bile `strings.xml` içerisine önemli verileri tanımlamak kaçınınız.

1.1.2. C ve C++ Kütüphaneleri Kullanımı

Uygulamanızın içinde kritik rol oynayan işlemler için NDK (Native Development Kit) kullanarak C veya C++ diliyle yazılmış modüller kullanabilirsiniz. Bu dille yazılan dosyalar `.so` uzantılı bir dosya olarak derlenir ve bu dosyanın çözülmesi (decompile) Java'ya göre daha zordur.

1.1.3. Debug ve Log'ları Kapatmak

APK dosyasının sanal bir Android cihaza yüklenmesi ve geliştirme yaparken kullandığımız log çıktılarının görüntülenmesi oldukça kolaydır. Bizler uygulama geliştirme sürecinde sıkça Log çıktıları kullanıyor olabiliriz. Bazen bu log çıktılarında başka kişilerin görmemesi gereken önemli bilgileri de ekleriz.

Uygulamamızın yayınlanabilir sürümünü çıktı almadan önce kullandığımız tüm `log` komutlarını temizlemiş olduğumuzdan ve uygulamamızın hata ayıklama modunu kapattığımızdan emin olmamız gerekir.

Log komutlarını tek tek silebileceğimiz gibi ProGuard'a aşağıdaki kuralı ekleyerek derleme sırasında R8 tarafından silinmesini de sağlayabiliriz. [3]

[3] https://sdks.support.brightcove.com/android/troubleshooting/removing-android-log-messages.html

```
-assumenosideeffects class android.util.Log {
      public static *** v(...);
      public static *** d(...);
      public static *** i(...);
      public static *** w(...);
      public static *** e(...);
}
```

Hata ayıklama modunu kapatmak için ise uygulama seviyesindeki `build.gradle` dosyasında `release` ya da `debug` versiyonlar için `isDebuggable` özelliğini `false` olarak işaretleyebiliriz.

```
buildTypes {
      release {
         isMinifyEnabled = false
         isDebuggable = false
         proguardFiles(getDefaultProguardFile("proguard-android-
optimize.txt"), "proguard-rules.pro")
     }
}
```

1.1.4. Root Erişimi Kontrolü

Fabrika üretiminden çıkan her Android cihaz, güvenilir ve resmi kaynaktan sağlanan bir Android işletim sistemi barındırır. Bu işletim sistemlerinde bazı yetkiler sınırlandırılmıştır. Açık kaynak kodlu olan bu işletim sistemlerine resmi olmayan kişi ya da ekiplerin müdahaleleri sonucu bu kısıtlandırmalar kaldırılarak işletim sisteminin kök yapısına ulaşabilmenin yolu açılabilir. Bu durum kötü niyetli kişi ya da kişilerin cihaz dosyalarında değişim yapabilmesine olanak tanır.

Örneğin biz uygulamamızı geliştirirken hassas bilgilerin gösterildiği ekranlar için işletim sistemine ekran görüntüsü veya ekran kaydı alınmamasını bildiririz. Fakat root erişimi olan bir cihazda bu kısıtlama kaldırılabilir. Bu sebeple uygulamanızın güvenliğini maksimum seviyeye çıkartmak için root erişimine sahip cihazlarda çalışmasını engellemeniz gerekebilir. Bunun için uygulamanız ayağa kalkarken bu kontrolü gerçekleştirmeniz gerekir. Bunu yapmanın bir kaç farklı yöntemi, her yöntemin

ise avantajları ve dejavantajları mevcuttur. Bazı yöntemler çok daha kolay atlatılabilirken bazılarının atlatılması daha zordur. Java veya Kotlin ile bu kontrolü uygulamak için şurayı[4] ve şurayı[5], daha güvenli bir yöntem olan NDK ile bu kontrolü yapmak içinse şurayı[6] inceleyebilirsiniz.

1.2. Veri Güvenliği

Android uygulamalar geliştirirken bizler sık sık veri saklama ihtiyacı duyarız. Bu veriler herkesin görmesine uygun olan açık veriler olabileceği gibi uygulama ile web servislerimiz arasında iletişim kurmak için bizim tanımladığımız API Anahtarları, API linkleri veya kullanıcılarımızın kendi sağladığı TC Kimlik, parola, sicil numarası vb. gibi aşırı önemli ve gizli kalması gereken veriler de olabilir.

Eğer kötü amaçlı birisi tersine mühendislik veya internet trafiğini dinleme yöntemleri ile bizim istek attığımız web servislerin url ve anahtarlarına erişirse bizim sunucularımızdan veri çalabilir ya da sunucularımıza aşırı istek göndererek saldırıda bulunabilir. Saldırı durumlarını sunucu üzerinde yapılacak kontrol ve ayarlarla yönetmek mümkündür ve Android geliştirmenin dışında kalan bir konudur fakat hem url hem de anahtara sahip bir kişinin bu koruma mekanizmasına yakalanmadan veri çalması gayet mümkündür. Bunun için yukarda bahsedilen önemli verileri kötü amaçlı kişilerden olabildiğince korumak adına uygun şifreleme yöntemlerini kullanmamız gerekir.

1.2.1. Geçici Dosyalar

Uygulamamız içerisinde hızlı erişmemiz gereken fakat kalıcı olmasını planlamadığımız bazı veriler oluşturmak istediğimizde şu yöntemi kullanarak geçici dosyalar (cache files) oluşturabiliriz:

---

[4] https://medium.com/@sindee.dev/how-to-add-root-and-emulator-detection-checking-for-android-app-a11da0dc9148
[5] https://stackoverflow.com/a/8097801/3768019
[6] https://stackoverflow.com/a/37237473/3768019

```
class CacheFileUtil(context: Context, fileName: String) {
    private val cacheDir: File = context.cacheDir
    private val myFile = File(cacheDir, fileName)

    fun createAndWriteToCache(data: String, isSuccess: (Boolean) -
> (Unit)) {
        try {
            BufferedOutputStream(FileOutputStream(myFile)).use {
bos ->
                bos.write(data.toByteArray())
                isSuccess(true)
            }
        } catch (e: IOException) {
            e.printStackTrace()
            isSuccess(false)
        }
    }

    fun readFromCache(): String {
        val stringBuilder = StringBuilder()

        try {
            BufferedInputStream(FileInputStream(myFile)).use { bis
->
                val buffer = ByteArray(1024)
                var bytesRead: Int
                while (bis.read(buffer).also { bytesRead = it } !=
-1) {
                    stringBuilder.append(String(buffer, 0,
bytesRead))
                }
            }
        } catch (e: IOException) {
            e.printStackTrace()
        }

        return stringBuilder.toString()
    }


    fun deleteCacheFile() {
        myFile.delete()
    }

}
```

Not: 1 MB altındaki dosyalar için `cacheDir`, 1 MB'ın üstündeki dosyalar için `externalCacheDir` kullanmayı unutmayınız.

```
val context = Context
val fileName = "GeciciDosya.txt"
val cacheFileUtil = CacheFileUtil(context, fileName)

cacheFileUtil.createAndWriteToCache("Dosya İçeriği") { isSuccess -
>
    println("Geçici dosya başarıyla oluşturuldu")
}

cacheFileUtil.readFromCache().let { data ->
    println("Geçici dosya içeriği: $data")
}


cacheFileUtil.deleteCacheFile()
```

Burada, uygulama güvenliği açısından dikkat edilmesi gereken detaylar ise geçici dosyalar içerisine önemli verilerin asla eklenmemesi ve işi bittikten sonra geçici dosyanın silinmesidir.

1.2.2. Kayıtlı Dosya Korunmuşluk Kontrolü

Eğer uygulamanızın akışı içerisinde bazı bilgileri saklamak için geçici ya da kalıcı dosya oluşturma yöntemini tercih ettiyseniz, dikkat etmeniz gereken bir başka detay daha bulunmaktadır. Bu detay, kaydettiğiniz dosyanın gelecek bir zamanda tekrar okunmadan önce sizin onu oluşturduğunuz şekilde kalıp kalmadığının kontrolünü içermektedir.

İçerisindeki bilgiler önemli olsun veya olmasın, bir dosyanın içeriğinin değiştirilmiş olması uygulamanızın akış sürecinde sorun çıkmasına zemin hazırlar. Bu sebeple cihazın belleğine bir dosya kaydettiğinizde o dosya üzerinden elde edeceğiniz hash kodunu korunaklı bir şekilde saklamanız ve dosyayı tekrar okumadan önce aynı yöntemle bir hash kodu oluşturup bu kodların birbiriyle tamamen eşleştiğinden emin olmanız gerekmektedir. Bunu şu yöntemle yapabilirsiniz:

```
val hash = calculateHash(stream)
// "beklenenHash" kodunu güvenilir bir yerde saklayın.
if (hash == beklenenHash) {
    // Kodlar eşleşti, dosya ile çalışabilirsiniz.
}

// Hash kodunu hesaplamak biraz zaman alabilir bu işlem sırasında
main thread'ı bloklamadığınızdan emin olun.
suspend fun calculateHash(stream: InputStream): String? {
    return withContext(Dispatchers.IO) {
        val digest = MessageDigest.getInstance("SHA-512")
        val digestStream = DigestInputStream(stream, digest)
        while (digestStream.read() != -1) {}
        digest.digest().joinToString(":") { "%02x".format(it) }
    }
}
```

1.2.3. Tek Yönlü Veri Şifreleme

Kullanıcıdan alınan hassas verilerin şifrelenmemiş hâlde saklanması, verilerin kötü amaçlı kişi veya kişilerin eline geçmesi durumunda kullanıcılarımıza zarar vermelerinin önünü açacaktır. Bu sebeple ister uygulama içinde ister uzak sunucuda sakladığımız her hassas veriyi mutlaka şifreleyerek saklamamız gerekir. Parolalar buna en iyi örneklerdir.

Parolalar tek kaynakla doğrulanan verilerdir yani sadece o parola ile bir aksiyon başlatacak kişinin bilmesi gerekir. Başkasına ait bir parolanın bizim tarafımızdan bilinmesinin -eğer kötü bir amacımız yoksa- hiçbir gereği yoktur.

Bir parola verisini şifrelemek için kullanmamız gereken yöntem tek yönlü olmalıdır. Tek yönlü veri şifrelemede, bir verinin bu yöntemle şifrelendikten sonra geri çözülememesi amaçlanmıştır. Sadece aynı saf veri geldiği zaman, aynı yöntemle şifrelenmiş veri oluşturulur ve veri tabanımızdaki kayıt ile eşleştirilir. Eğer eşleşme varsa kullanıcının parolasını doğru yazdığını bilebiliriz ama parolanın ne olduğunu bilemeyiz.

Tek yönlü şifreleme için SHA-256 ve Bcrypt yöntemleri önerilir. Bu yöntemler hakkında daha fazla bilgi için şuraya[7] bakabilirsiniz.

[7] https://medium.com/nerd-for-tech/sha-2-and-bcrypt-encryption-algorithms-e0c0599b0da

***

Örneğin, “Kullanıcı uygulamaya giriş yaptı mı?” veya “kullanıcı arayüzü gece modunda mı kullanıyor?” gibi kısa verileri `key-value` şeklinde saklamak için sıklıkla başvurduğumuz `SharedPreferences` uygun bir yöntemdir. Fakat `SharedPreferences` nihayetinde içerisine kaydedilen bilgilerin açıkça görülebildiği bir XML dosyası oluşturur. Bu dosyaya cihaz belleği içerisinde `/data/data/UYGULAMA.PAKET.ADI/shared_prefs/UYGULAMA.PAKET.ADI_preferences.xml` yolundan kolayca erişmek mümkündür. [8]

Dolayısıyla kötü amaçlı kişi veya kişiler tarafından ulaşılmasını istemediğimiz verileri barındırmak için `SharedPreferences` uygun bir yöntem olmayacaktır. Pektabi `SharedPreferences` kullanırken hassas verileri şifreleme ve şifre çözme işlemlerine tâbi tutabiliriz fakat bu yapıyı kurmak nispeten uzun bir süreç olacaktır.

1.2.4. EncryptedSharedPreferences

`SharedPreferences` ‘in üzerine geliştirilen bu kütüphane AndroidX Security araçlarının altında yer almaktadır ve çalışma prensibi yukarıda anlattığımız esasa dayanır. `EncryptedSharedPreferences` ile bir veriyi saklarken özel bir anahtar ile şifreleme işlemine tâbi tutarız. Böylece oluşan XML dosyasına ulaşılmış olsa bile içerisinde yer alan veriler şifrelenmiş olacaktır.

`EncryptedSharedPreferences` ‘i kullanmak için şu adımları takip edebiliriz.

1. Modül seviyesindeki `build.gradle` dosyamıza gerekli kütüphaneyi ekleyelim:

```
implementation "androidx.security:security-crypto:1.0.0-alpha02"
```

2. Şifreleme işleminde kullanmak üzere bir `MasterKey` oluşturalım:

[8] https://stackoverflow.com/questions/23635644/how-can-i-view-the-shared-preferences-file-using-android-studio#answer-52352741

```
val keyGenParameterSpec = MasterKeys.AES256_GCM_SPEC
val masterKeyAlias = MasterKeys.getOrCreate(keyGenParameterSpec)
```

3. Bu MasterKey'i kullanarak bir `EncryptedSharedPreferences` nesnesi yaratalım. Bunu yapabilmek için `Context`'e ihtiyacımız olacak buna dikkat edelim:

```
val sharedPreferences = EncryptedSharedPreferences.create(
    "shared_preferences_filename",
    masterKeyAlias,
    context,
    EncryptedSharedPreferences.PrefKeyEncryptionScheme.AES256_SIV,
    EncryptedSharedPreferences.PrefValueEncryptionScheme.AES256_GCM
)
```

4. Oluşturduğumuz bu nesne içinde veri saklamak için:

```
sharedPreferences
    .edit()
    .putString("USER_ID", "ABC123")
    .apply()
```

5. Ve sakladığımız veriye ulaşmak için:

```
sharedPreferences.getString("USER_ID", "DEFAULT_VALUE") -> //ABC123
```

Bu işlemler sonrası oluşan XML dosyasını incelediğimiz zaman şöyle bir çıktı görüyor olacağız:

```
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>
<map>
    <string name="ATP1ABa3NIlOap2c7iNkVaUcQmTocrnpkXl0PyI=">AU+p3hwqCgvlDOtIaawFHWVDf4rFsqghM7ivFTEJesrRp19D+zk7tqsqlGZPLAbryHI=</string>
    <string name="__androidx_security_crypto_encrypted_prefs_key_keyset__">12a901802f1a5d2fbc5…</string>
    <string name="__androidx_security_crypto_encrypted_prefs_value_keyset__">128801da6fdef289b2c6e2933c34…</string>
</map>
```

Görüldüğü gibi hiç bir veri açıkca okunabilecek durumda değildir. `SharedPreferences`’a göre bu yöntem, hassas verileri saklarken daha fazla işimize yarayacaktır. Detaylı bilgi için şuraya[9] bakabilirsiniz.

1.2.5. API Key Güvenliği

Yukarıda bahsettiğimiz üzere API anahtarları (API Key) uygulamamız için bizim geliştirme aşamasında sağladığımız ve başkaları tarafından görülmemesi gereken çok önemli verilerdir. Bu gibi verileri saklarken güvenliğini sağlamak üzere başvurabileceğimiz bir kaç yöntem vardır.

1.2.5.1. BuildConfig

Bu yöntemde saklamak istediğimiz veri için ilgili modül seviyesindeki `build.gradle` dosyasına bir değişken oluşturmamız gerekir. Bu değişkeni daha sonra çağırabilmek için yine aynı dosya içerisinde `buildFeatures` blogu içinde yer alan `buildConfig` özelliğini `true` olarak işaretlememiz gerekir:

[9] https://bignerdranch.com/blog/encrypting-shared-preferences-with-the-androidx-security-library/

```
android {
      …
      buildFeatures {
         …
         buildConfig = true
      }
}
```

Şimdi `build.gradle` dosyamız içerisine bir değişken oluşturup API Key verimizi tanımlayabiliriz:

```
android {
      …
      defaultConfig {
            buildConfigField(
                type = "String",
                name = "API_KEY_FROM_BUILD_GRADLE",
                value = "API_KEY_1"
            )
}
```

Not: Bu değişlenlerin uygulama içerisinden nasıl çağıralacağı daha sonra gösterilecektir.

`build.gradle` dosyası bir modülün derlenmesinde önemli rol oynadığı için projemizi paylaşmamız gereken her yerde gereklidir. Bu yüzden Git gibi versiyon kontrol sistemlerine mecburi olarak dahil edilir. Bu ise halka açık (public) projelerde API Key verimizin kolaylıkla görünmesini sağlaması açısından <u>uygun bir yöntem değildir</u>. Bu sebeple versiyon kontrol sistemlerine dahil olması gerekmeyecek bir dosya içerisine verimizi tanımlayıp `build.gradle`'da çağırmamız daha mantılı olacaktır. Bunun için farklı yöntemler kullanabiliriz.

gradle.properties

API Key verimizin hem versiyon kontrol sistemlerinde hem de çözümleme (decompile) işleminde bulunmasını zorlaştırmak `gradle.properties` dosyasında

barınması yöntemine başvurabiliriz. Fakat dikkat edilmesi gereken bir nokta olarak, `gradle.properties` dosyası da tıpkı `build.gradle` dosyası gibi modülün derlenmesinde önemli rol oynar ve paylaşılması gerekir. Bu sebeple bu tanımlamayı yaptıktan sonra hangi dosyaları versiyon kontrol sistemine göndermeyeceğimizi belirlediğimiz `.gitignore` dosyasına eklememiz ve paylaşılmadığına emin olmamız gerekmektedir. Fakat bu `gradle.properties` dosyasında daha sonra yaptığımız değişikliklerin de paylaşılmamasını sağladığı için ilerleyen zamanlarda sorun çıkarabilir. Aynı zamanda `gradle.properties` dosyası tersine mühendislik yöntemleri ile çözülebilir. Bu riskler sebebiyle `gradle.properties` dosyası içerisinde veri saklama tavsiye edilen bir yöntem değildir. Uygulaması ise şöyle yapılır:

1. `gradle.properties` içine `key-value` şeklinde veri eklenir:

```
org.gradle.jvmargs=-Xmx2048m -Dfile.encoding=UTF-8
android.useAndroidX=true
…
API_KEY=API_KEY_2
```

2. Eklediğimiz bu değişken `build.gradle` dosyası içine tanımlanır:

```
android {
      …
      buildConfigField(
          type = "String",
          name = "API_KEY_FROM_GRADLE_PROPERTIES",
          value = property("API_KEY").toString()
      )
}
```

local.properties

Tıpkı yukarıda anlatılan yönteme benzeyen bu yöntem, `local.properties` dosyasının yerel bir dosya olması ve `.gitignore` ‘a otomatik olarak dahil olması sebebiyle versiyon kontrol sistemlerine iletilmemesi açısından bir nebze de olsa daha

mantıklı bir seçenektir. Fakat yine de uygulamanın inşa edilmesi (build) sırasında `build.gradle` için otomatik şekilde oluşturulan `BuildConfig.java` dosyası içerisine eklenmesi sebebiyle tamamen güvenli bir yöntem değildir. Uygulanışı `gradle.properties` yöntemi ile benzerlik gösterir.

1. `local.properties` içine `key-value` şeklinde veri eklenir:

```
sdk.dir=/Users/muratyuksektepe/Library/Android/sdk

…

API_KEY=API_KEY_3
```

2. Eklediğimiz bu değişkene ulaşmak için önce `local.properties` dosyası `inputStream` ile okunur sonra değişkenimiz `build.gradle` dosyası içine tanımlanır:

```
android {
      …
      val localPropertiesFile =
      project.rootProject.file("local.properties")
      val localProperties = Properties()
      localProperties.load(localPropertiesFile.inputStream())

      buildConfigField(
          type = "String",
          name = "API_KEY_FROM_LOCAL_PROPERTIES",
          value = localProperties.getProperty("API_KEY").toString()
      )
}
```

apikey.properties

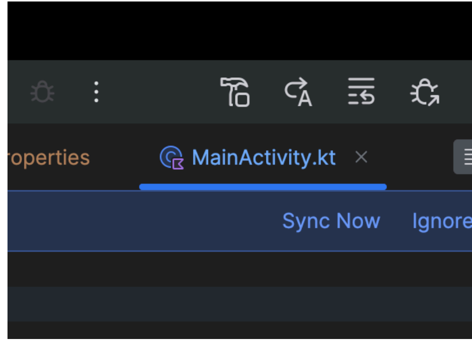
Kendi `properties` dosyanızı oluşturarak uygulayabileceğiniz bu yöntemin çalışma şekli `local.properties` ile tamamen aynıdır.

1. Proje ana dizinine ismini kendinizin belirlediği, uzantısı ".properties" olan bir dosya oluşturun.
2. Saklamak istediğiniz veriyi `local.properties`'de olduğu gibi `key-value` şeklinde tanımlayın.

3. Oluşturduğunuz yeni dosyayı `.gitignore` dosyasına ekleyin.
4. Oluşturduğunuz yeni dosyayı `build.gradle` dosyasına tanıtın. Burada dosyanın yolunu `root.project()` metodu içerisine doğru şekilde verdiğinizden emin olun.
5. Yeni dosya içerisinde yer alan değişkeninizi `buildConfigField` metodunu kullanarak `build.gradle` içerisine ekleyin.

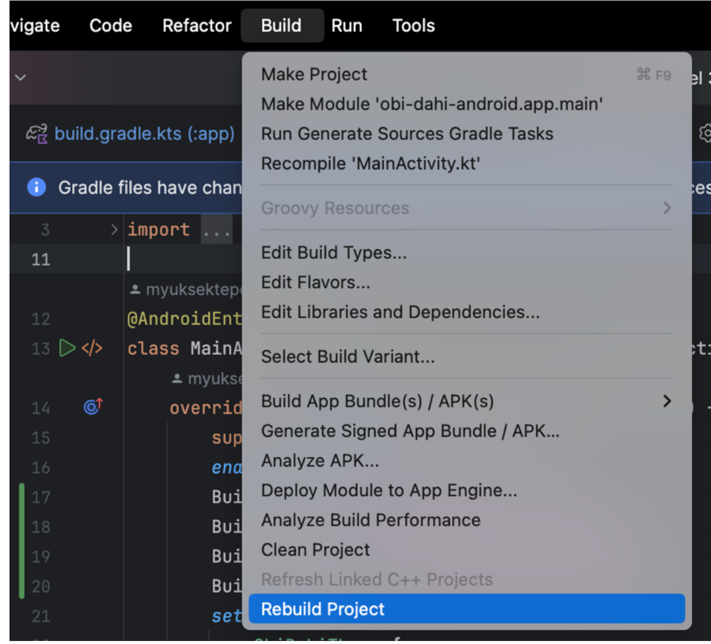
build.gradle İçerisinde Oluşturduğumuz Değişkene Ulaşmak

Yukarıdaki işlemlerden herhangi birini uyguladıysak eğer ilk yapmamız gereken şey gradle'a yaptığımız değişiklikleri uygulaması gerektiğini söylemektir. Android Studio bir değişiklik olduğunu anlayacak ve bize uyarıda bulunacaktır. "**Sync Now**" butonuna tıklayarak bu değişiklikleri uygulayalım.



Şekil 1: Sync Now - Android Studio

Unutmayalım ki yaptığımız bu işlemleri `build.config` dosyası üzerinde gerçekleştirdik. Bu, tanımladığımız değişkenin inşa (build) işlemi sonrası ulaşılabileceği anlamına gelmektedir. Bu yüzden **Build** menüsü altında yer alan **Rebuild Project** işlemini tetiklememiz gerekmektedir.



Şekil 2: Rebuild Project - Android Studio

Build işlemi bittikten sonra artık ilgili modülümüz içerisinde herhangi bir yerden değişkenimize otomatik şekilde oluşan `BuildConfig` objesi altından erişebiliriz.

```
@AndroidEntryPoint
class MainActivity @Inject constructor() : ComponentActivi
    myuksektepe *
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        enableEdgeToEdge()

        BuildConfig.API_KEY_FROM_BUILD_GRADLE
        BuildConfig.API_KEY_FROM_GRADLE_PROPERTIES
        BuildConfig.API_KEY_FROM_LOCAL_PROPERTIES
        BuildConfig.API_KEY_FROM_APIKEYS_PROPERTIES
```

Şekil 3: BuildConfig.API_KEY

Bir kez daha hatırlatalım, API Key'leri saklamak için BuildConfig kullanımı, API Key'in Activity, Fragment ya da ViewModel gibi dosyalar içerisinde statik şekilde saklanmasından daha iyi bir yöntemdir. Kod karıştırma yöntemi ile birlikte kullandığı

zaman bir kademe daha güvenli sayılabilir. Fakat tamamen güvenli bir yöntem demek doğru olmaz. Bu yöntemler API Key'lerinizi, kötü amaçlı kişilerden %100 saklamanızı sağlayamaz.

1.2.5.2. Android NDK

C ve C++ dilinde yazılmış olan kodların çözülme işleminin Java'ya göre daha zor olduğundan daha önce bahsetmiştik. Bu sebeple önemli algoritmalar ve gizli bilgilerin eklenmiş olduğu modülleri NDK (Native Development Kit) olarak projemize ekleyerek güvenlik önlemlerini arttırmış oluruz. Bu yöntemin uygulanma şekli hakkında bilgi almak için şuraya[10], şuraya[11] ve şuraya[12] göz atabilirsiniz.

1.2.5.3. Sunucu Tarafında Veri Saklama

Bir APK ve AAB dosyasının tersine mühendisliğe tam olarak kapatılamadığını, C ve C++ ile yazılmış kodların Java'ya göre daha zor olsa da yine de çözülebildiğini, kod karıştırma yönteminin aslında bir koruma yöntemi olmadığını belirtikten sonra bir API Key'i kötü amaçlı kişilerden saklamanın en iyi yöntemi olarak sunucu taraflı veri saklama yönteminden bahsedebiliriz.

Bu yöntem kısaca API Key ihtiyacı olmayan bir API'e istekte bulunarak sunucu tarafında korunan verinin alınması, uygulama yaşam döngüsü boyunca geçici olarak bellekte barındırılması ve API Key ile çalışan her web servis isteğinde kullanılması mantığına dayanır. Böylece API Key verisi uygulamamız içerisinde hiçbir zaman yer almaz ve kod çözülme işlemleri sonrasında bulunamaz. API Key'in sunucu tarafında korunması Android geliştiriciler olarak bizim yönetimimizde değildir. Fakat bu API Key'e ulaşmak için ilk atılan istek ve bu isteğin cevabının dinlenmesi yine Android uygulamamızın içinde gerçekleşeceği için web servis isteklerinin güvenliği yine bizlerin ilgilenmesi gereken bir konu olacaktır.

---

[10] https://developer.android.com/studio/projects/install-ndk
[11] https://www.youtube.com/watch?v=1xE4mKbNSog
[12] https://techbrainzadda.blogspot.com/2022/11/how-to-secure-api-key-in-android-app.html?m=1

2. Uygulama - Web Servisleri İletişim Güvenliği

Bilindiği üzere, bir uygulama ile web servisler arasındaki iletişim internet trafiği ile sağlanır. Bazı programlar sayesinde bu internet akışını kendi hazırladığımız bir sistem üzerinden geçirerek trafiği kolayca dinlememiz mümkündür. Bu bölümde uygulamamız içerisinde yapılan web servis istekleri ve bu istekler sonrası aldığımız cevapların güvenliğini nasıl sağlayabileceğimizi göreceğiz.

2.1. TLS Kullanımı

Eğer uygulamanız internet ile bir web servis bağlantısı kuracaksa (örn: API Call), bağlantı kurmak istediğiniz web sunucusunun güvenilir bir kaynaktan sağlanmış ve geçerli olan bir güvenlik sertifikasına sahip olduğuna emin olun. Android 10 ile birlikte tüm HTTP isteklerinde bağlantıların TLS 1.3 üzerinden yapılması varsayılan olarak belirlenmiştir. Kısacası eğer özel bir ayar yapmıyorsanız uygulamanız içinden yapacağınız her web servis isteği HTTPS ile başlayan, güvenilir bir url üzerinden yapılmalıdır.

Ek Bilgi: TLS (Transfer Layer Security), SSL'in bazı güvenlik açıklarının düzeltilmiş, güncel versiyonudur. [13]

2.2. Özelleştirilmiş Network Ayarları

Uygulamanızın, TLS/SSL kullanmıyor olsa bile hangi bağlantılara güvenebileceğini belirlemenizin bir yöntemi vardır. Örneğin uygulamanız içerisinde yapacağınız tüm web isteklerinde TLS/SSL zorunluluğunu kaldırmak için `AndroidManifest.xml` dosyası içerisinde şu basit ayarı yapabilirsiniz:

```
<application android:usesCleartextTraffic="true"> … </application>
```

[13] https://aws.amazon.com/tr/compare/the-difference-between-ssl-and-tls

Bu yöntem doğal olarak önerilmez. Eğer uygulamanız içinde güvensiz dahi olsa istekte bulunacağınız linkler belli ise bunu şu yöntemle yapmanız daha doğru olacaktır:

- `res/xml/` yoluna `network_security_config.xml` isimli bir dosya oluşturun.
- Dosyanın içeriğini isteğidiniz ayarları içerecek şekilde yapılandırın:

```
<network-security-config>
    <domain-config cleartextTrafficPermitted="true">
        <domain includeSubdomains="true">website.com</domain>
    </domain-config>
</network-security-config>
```

cleartextTrafficPermitted: Bu domain için yapılacak network isteklerinin güvenli yöntem (SSL/TLS) üzerinden yapılıp-yapılmamasını belirlediğimiz alandır. HTTPS ile başlayan url'ler için güvenli istekleri kullanmak üzere `false`, HTTP ile başlayan güvenli olmayan istekler için `true` olarak işaretleyin.

includeSubdomains: Yaptığınız ayalara, bu domain'e ait alt alan adlarının da dahil olup-olmamasını seçebileceğiniz alandır.

- Oluşturduğunuz bu ayar dosyasını uygulamanıza `AndroidManifest.xml` dosyası aracılığıyla tanıtın:

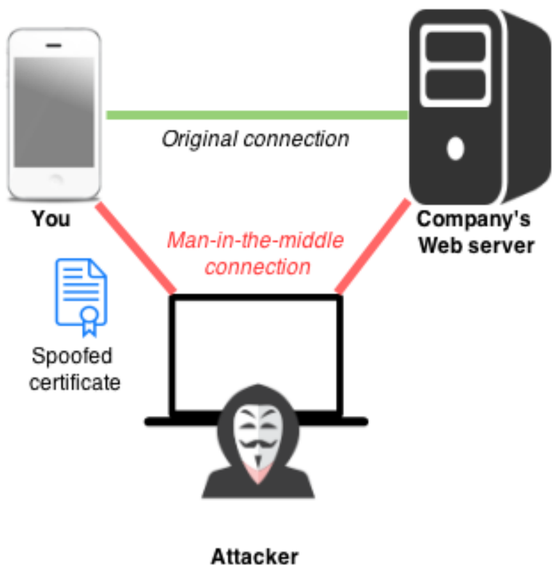
```
<application
    android:networkSecurityConfig="@xml/network_security_config">
    ...
</application>
```

2.2.1. SSL/TLS Pinning

Hazır özelleştirilmiş network ayarlamalarından bahsediyorken, uygulamamız ile web servislerimiz arasındaki iletişim güvenliğini artıracak yöntemlerden bahsetmeye devam edelim. Yukarıda hazırladığımız `network_security_config.xml` dosyası bizim için güvensiz bağlantı ayarı yapmamıza olanak sağladığı gibi bağlantılarımızın güvenliğini güçlendirmek için de olanak sağlar.

Man-in-the-middle Saldırısı

Bir uygulama ile web servisleri arasındaki iletişimin internet üzerinden sağlanırken kötü amaçlı kişilerin bu internet trafiğine dahil olarak istek ve cevapları dinleyebildiğinden bahsetmiştik.



Şekil 4: Man-in-the-middle şeması (Görsel: Wikipedia)

Bu, Man-in-the-middle (MiTM) saldırısının sadece bir kısmıdır. MiTM saldırısı yapan bir kişi dinlediği trafiği manipüle ederek hem uygulamamıza gelen cevabı hem de uygulamamızdan sunucuya giden isteğin muhtevasını değiştirebilir. Bu durum iki taraf içinde istenmeyen sonuçlar doğracaktır. Bu duruma engel olmak için ek önlemler almamız gerekir. Bu önlemlerden en önemlisi SSL/TLS Pinning (Sertifika Sabitleme)'dir.

***

Sertifika pinleme işlemi, uygulamamız içerisinde istekte bulanacağımız linkleri daha önce oluşturduğumuz `network_security_config.xml` dosyasına eklerken bu web servisin sahip olduğu sertifikanın bilgilerini de ekleyerek gerçekleştirdiğimiz bir yöntemdir. MiTM saldırısı yapan biri uygulamanızın kabul edebileceği düzeyde güvenli fakat sahte bir sertifika sağlayarak trafik akışını kendi üzerine kolayca alabilir.

Yaptığımız bu işlem ise uygulamamıza “*web servis isteği atarken; sadece linkin doğru olduğuna, HTTPS ile başladığına ve geçerli bir güvenlik sertifikasına sahip olduğuna güvenme aynı zamanda bu sertifikayı şu veriler ile doğrula*” anlamına gelmektedir. Yani uygulamamız içindeki web servis isteklerini sadece belirli bir ya da bir grup sertifikaya güvenerek yapmaya zorlamamız demektir.

```
<network-security-config>
    <domain-config cleartextTrafficPermitted="false">
        <domain
includeSubdomains="true">guvenliwebsite.com</domain>
          <pin-set>
              <pin digest="SHA-256">SERTIFIKA_PIN</pin>
              <pin digest="SHA-256">YEDEK_SERTIFIKA_PIN</pin>
          </pin-set>
    </domain-config>
</network-security-config>
```

pin digest: Güvenli bağlantı yapmak istediğiniz alan adına ait sertifikanın SHA-256 yöntemi ile şifrelenmiş benzersiz anahtar değerini eklemek için kullanılan ayardır. Bir sertifikanın pin değerini bulmak için şurayı[14] inceleyebilirsiniz.

***

Bir ya da bir grup sertifika için pin koruması eklediğimiz zaman en çok dikkat etmemiz gereken konular sertifikanın değişmesi durumu ve geçerlilik süresidir. Sertifikanın değişmesi durumu için her zaman bir yedek pin eklemeniz önerilir. Aksi takdirde uygulamanızda yeni pini içeren bir değişiklik yaparak uygulama mağazasına güncelleme göndermeniz gerekecektir.

Eğer sunucuda yer alan sertifikanın geçerlilik süresi dolmuşsa veya sertifika yenilenmişse, uygulamanız içerisindeki pinler eski kalmış olacak ve bağlantı sorunları yaşanacaktır. Bunun üstesinden gelmek için her pin setine bir geçerlilik süresi eklemek mümkündür. Bu sayede pinleriniz kontrol edilmeden önce geçerlilik süresinin dolup-

[14] https://techmusings.optisolbusiness.com/unveiling-the-sha-256-fingerprint-using-ssl-labs-ssl-pinning-76d37743e18c

dolmadığına bakılır. Geçerlilik süresi dolmuş pinler için pin eşleştirilmesi yapılmaksızın belirttiğiniz link güvenli sayılır:

```
<network-security-config>
    <domain-config cleartextTrafficPermitted="false">
        <domain
includeSubdomains="true">guvenliwebsite.com</domain>
          <pin-set expiration="2024-08-17">
              <pin digest="SHA-256">SERTIFIKA_PIN</pin>
              <pin digest="SHA-256">YEDEK_SERTIFIKA_PIN</pin>
          </pin-set>
    </domain-config>
</network-security-config>
```

Kendinden İmzalı Sertifika İle Çalışmak

Bazı durumlarda halka açık olmayan, sadece şirket içinde kullanılması için hazırlanmış ve güvenilir bir kaynak yerine kendi kendine imzalanmış sertifikaya sahip bir linke istek atmanız gerekebilir. Bu sunucu tarafındaki sertifikanın çok uzun süre -belki de hiç- değişmeyeceği anlamına gelir. Bu gibi durumlarda en güvenli yöntem ise sunucu tarafında yer alan sertifikanın uygulama içerisine dahil edilerek tam eşleşme sağlanmasıdır. Uygulamanız bu linklere bir istek gönderdiğinde kendisinde yer alan sertifika ile karşı taraftaki sertifikayı kıyaslar ve sadece tamamen aynı olduğu durumlarda isteği gerçekleştirir. Bunun için sunucu tarafındaki sertifika dosyasını `.pem` ya da `.der` uzantılı şekilde projemiz de `res/raw` klasörü altına eklemek ve `network_security_config.xml` dosyasını şu şekilde düzenlemek gerekir:

```
<network-security-config>
    <domain-config>
        <domain includeSubdomains="true">ozelwebsite.com</domain>
        <trust-anchors>
            <certificates src="@raw/KAYDEDILMIS_SERTIFIKA"/>
        </trust-anchors>
    </domain-config>
</network-security-config>
```

3. Uygulama - Uygulama İletişim Güvenliği

Uygulamamız için düşünmemiz gereken bir diğer güvenlik alanı da bizim uygulamamız ile yine bizim tarafımızdan geliştirilmiş veya tamamen farklı bir uygulama ile olan iletişiminin güvenliğidir. Eğer iki uygulama arasında planlanan bir iletişim söz konusu değilse kötü amaçlı iletişimlere karşı uygun önlemleri almamız elzemdir.

3.1. ContentProvider'a Erişimi Engellemek

Eğer kendi uygulamanız ile -size ait olsun veya olmasın- diğer uygulamalar arasında bir veri alış-verişi planlamıyorsanız, diğer uygulamaların sizin uygulamanıza ait olan ContentProvider'lara (uygulama içinde kullanıcıdan aldığınız veya kendi oluşturduğunuz verileri barındırdığınız ortak alan) erişimine izin vermeyerek güvenlik sağlayabilirsiniz:

```
<manifest
xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.myapp">
    <application ... >
        <provider
            android:name="android.support.v4.content.FileProvider"
            android:authorities="com.example.myapp.fileprovider"
            …
            android:exported="false">
            …
        </provider>
        …
    </application>
</manifest>
```

3.2. Aynı İmza Kontrolü

Size ait olan birden fazla uygulama olduğunu, bu uygulamaların aynı imza dosyası ile imzalandığını ve kullanıcı verileri, bir form ile toplanmış veriler veya ödeme öncesi kart bilgileri gibi bazı önemli bilgileri birbiriyle paylaşması gerektiğini düşünelim. Bu durumda A uygulamasının `AndroidManifest.xml` dosyası içerisinde

özelleştirilmiş bir izin tanımlarken, bu izne diğer uygulamalar tarafından hangi yöntemle ulaşılacağını belirttiğimiz koruma seviyesi özelliğini (`protectionLevel`), aynı imzayı paylaşan uygulamalar (`signature`) olarak seçebilirsiniz:

A Uygulaması

AndroidManifest.xml içinde:

```
<permission android:name="com.example.appA.ACCESS_SHARED_DATA"
  android:protectionLevel="signature" />
```

Activity veya Servis içerisinde:

```
fun performSensitiveOperation() {
      enforceCallingOrSelfPermission(
            "com.example.appA.SHARED_DATA",
            "Access Denied"
      )
}
```

B Uygulaması

AndroidManifest.xml içinde :

```
<permission android:name="com.example.appA.ACCESS_SHARED_DATA"/>
```

Veriye ulaşmak istediğimiz zaman:

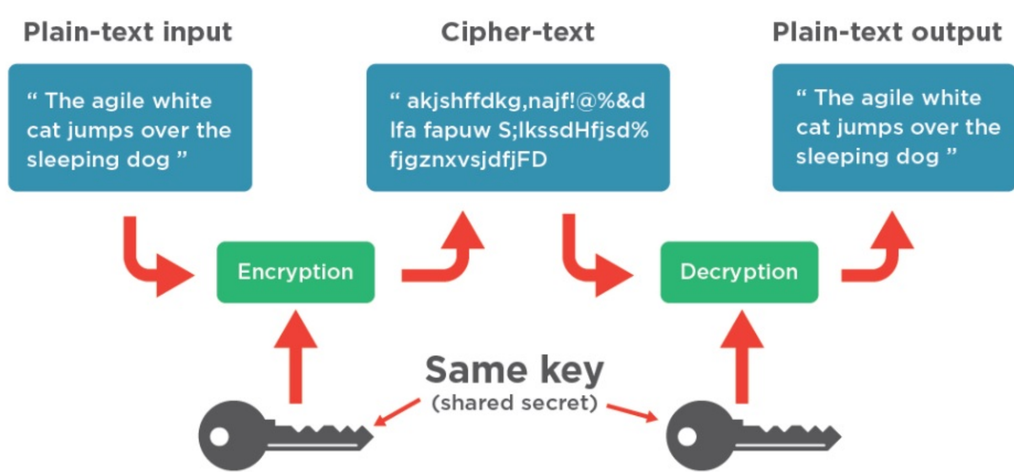
```
fun accessProtectedData() {
    try {
        val appAContext = createPackageContext("com.example.appA",
0)
    } catch (e: SecurityException) {
        // Yetki hatası
    }
}
```

Önemli Notlar:

- Bu yöntemde iki uygulama arasındaki iletişim imzalardan kaynaklandığı için imza dosyalarının korunması/kaybedilmemesi çok önemlidir.
- Tüm özel izinler imza seviyesiyle korunamaz. Bazı hassas bilgiler için daha yüksek seviye koruma yöntemleri kullanmak gerekir.
- İmza seviyesinde koruma yöntemini sadece gerektiği zaman kullanmak gerekir. Her izin için bu seviyenin kullanılması ilerleyen zamanlarda uygulamamızın diğer uygulamalar ile iletişim kurmasının engelleyen bir faktör olabilir.

3.3. AES ile Şifrelenmiş Veri Paylaşımı

Bilindiği üzere bir çok veri şifreleme yöntemi mevcuttur, eğer sizin tarafınızdan geliştirilen ve birbiriyle iletişim halinde olacak birden fazla uygulamanız varsa AES (Advantage Encryption Standard) şifreleme algoritması sizin için en mantıklı seçenek olabilir. Bu yöntemde veriler şifrelenme ve şifre çözülme işlemlerine maruz kalırken önceden tanımlanmış bir anahtar metin üzerinden işlem görürler.



Şekil 5: AES Şifreleme Şeması (Görsel: https://us.transcend-info.com/embedded/technology/aes-encryption)

Böylece, MiTM saldırısına maruz kalsanız ve iki uygulama arasında akan veri kötü amaçlı birinin eline geçmiş bile olsa, önceden belirlediğiniz anahtar metin olmadan

şifreyi çözüp saf veriye ulaşamaz. İhtiyacınız olan anahtar metnin saklanması da son derece önemlidir bu yüzden eğer henüz okumadıysanız **Uygulama İçi Güvenlik** konusunu okumanızı tavsiye ederiz.

AES'in bir çok modu bulunmaktadır. Basit bir uygulama için şu kodlara göz atabilirsiniz:

```
import javax.crypto.Cipher
import javax.crypto.spec.SecretKeySpec
import kotlin.io.encoding.Base64
import kotlin.io.encoding.ExperimentalEncodingApi

@OptIn(ExperimentalEncodingApi::class)
fun encryptAES(plainText: String, key: String): String {
    val cipher = Cipher.getInstance("AES/ECB/PKCS5Padding")
    val secretKeySpec = SecretKeySpec(key.toByteArray(), "AES")
    cipher.init(Cipher.ENCRYPT_MODE, secretKeySpec)
    val encryptedBytes = cipher.doFinal(plainText.toByteArray())
    return Base64.encode(encryptedBytes)
}

@OptIn(ExperimentalEncodingApi::class)
fun decryptAES(encryptedText: String, key: String): String {
    val cipher = Cipher.getInstance("AES/ECB/PKCS5Padding")
    val secretKeySpec = SecretKeySpec(key.toByteArray(), "AES")
    cipher.init(Cipher.DECRYPT_MODE, secretKeySpec)
    val encryptedBytes = Base64.decode(encryptedText)
    val decryptedBytes = cipher.doFinal(encryptedBytes)
    return String(decryptedBytes)
}

fun main() {
    val key = "ABCDEFGHIJLKMNOP" // 16, 24, or 32 bytes for AES-
128, AES-192, or AES-256
    val plainText = "Hello, AES in CBC!"

    val encryptedText = encryptAES(plainText, key)
    println("Encrypted: $encryptedText")

    val decryptedText = decryptAES(encryptedText, key)
    println("Decrypted: $decryptedText")
}
```

Diğer modları ve uygulama şekillerini incelemek için şuraya[15] göz atabilirsiniz.

[15] https://medium.com/@mobiledev4you/implementation-of-aes-encryption-in-android-dca250525b4

4. Uygulama - Kullanıcı İletişimi Güvenliği

Bu bölümde uygulamamız ile kullanıcı arasındaki iletişimin güvenliğini sağlamak ve kullanıcının uygulamamıza olan güvenini oluşturmak ve korumak üzerine olan konulara değineceğiz.

4.1. Hassas Veriler Alınırken Dikkat Edilmesi Gerekenler

Uygulamamızın kullanıcı ile olan iletişimi sırasında kullanıcıdan hassas ve/veya kişisel bilgiler talep ettiğimiz durumlar olabilir. Bu gibi durumlarda hem kullanıcının uygulamayla arasında güven oluşması hem de uygulamanın kötü niyetli kişi ya da kişiler tarafından çözülmeye çalışılmasında alabileceğimiz bazı önlemler mevcuttur.

4.1.1. Parolaların Zorluk Seviyesi

Bir kullanıcının uygulamamıza kayıt olması gerektiği durumlarda ondan e-mail adresi veya kullanıcı adı gibi benzersiz bir veri ve bir parola isteriz. Bu noktada istediğimiz parolanın kabul edilebilir zorluk seviyesi önem teşkil eder. En az bir büyük ve bir küçük harf, bir rakam ve bir özel karakter içeren ve en az 8 karakter uzunluğa sahip parolaların çözülmesi bu şartları sağlamayanlara göre çok daha kolay olacaktır. Ayrıca günümüzde -yaygın kullanımı sayesinde- kullanıcıların alıştığı bu uygulamanın dışında basit parolalar istemek kullanıcının gözünde uygulamamıza karşı bir güvensizlik şüphesi başlatabilir.

4.2. Hassas Verilerin Gösterilmesi

Uygulamamız içerisinde kullanıcıya özel hassas verileri gösterirken öncesinde o kullanıcıya ait bazı özel veri ve aksiyonların doğrulayıcı görevinde kullanılması kötü amaçlı kişi veya kişilerin o verileri görmesine engel olacaktır.

4.2.1. Parola, Pin, Özel Soru veya Biyometrik Veri ile Doğrulama

Kullanıcının önceden tanımladığı özel bir soruyu gösterip cevabını istemek, parola veya önceden tanımladığı kısa pin bilgisini sormak gibi yöntemler kullanılabilir. Eğer bu aşamada kesinliği arttırmak ve korumayı daha üst seviyelere çıkarmak istiyorsak parmak izi ve yüz tanıma gibi biyometrik veriler içeren yöntemlere başvurabiliriz. Biyometrik doğrulama ile ilgili detaylar için şuraya[16] göz atabilirsiniz.

4.2.2. Kullanıcının Eğitilmesi

Uygulamamızın akışı içerisinde hem kullanıcının kendi güvenliği hem kullandığı cihazın güvenliği hem de uygulamamızın güvenliğini sağlayacak ipuçları ve önerilerin yer aldığı tanıtım ekranlarını kullanıcıya uygun şekil ve zamanlarda göstererek kullanıcıyı bu konularda bilgilendirmiş ve uygulamamıza olan güvenini sağlamasına yardımcı olmuş oluruz.

Kullanıcıya gösterilecek her bilginin olabildiğince kısa ve açıklayıcı olması, uygun görsel materyallerle desteklenmesi kullanıcının dikkatini çekmesine ve az zaman ayırarak verilmek istenen mesaja en hızlı şekilde ulaşmasına yardımcı olacaktır.

[16] https://developer.android.com/identity/sign-in/biometric-auth