Esame di Computer Security

Relazione di Sicurezza Android su "Progetto Tesi di Laurea Triennale – MyUniversity"

Professori: Armando Alessandro
Verderame Luca

Studenti: De Luca Jacopo - matricola 3675883
Rossi Riccardo - matricola 3689831

Possibili problemi di sicurezza presi in considerazione:

- SQL Injection
- Content Provider
- Man-in-the-middle (protocollo HTTP)

I. SQL Injection

Cos'è e come funziona

SQL Injection (SQLi) è un injection attack che permette di inserire codice SQL malevolo in campi input (es. LogIn, Caselle di testo ecc.) che sarà poi eseguito.
Questo attacco sfrutta la superficialità, inesperienza o incapacità del programmatore il quale non si preoccupa di filtrare gli statement SQL per evitare che vengano eseguite query dannose (solitamente si tratta di applicazioni web).
Questa tecnica consente all'attacker di recuperare, modificare, cancellare record presenti nelle tabelle del database oppure renderle inaccessibili o addirittura diventare amministratore dello stesso database.

Tipologie

SQL injection sfrutta alcune vulnerabilità sempre con lo stesso scopo di manomettere i dati di un qualsiasi database. In seguito considereremo le più comuni:

CARATTERI NON FILTRATI CORRETTAMENTE

In questo caso l'attacco si verifica quando non viene filtrato l'input dell'utente dai caratteri di escape.

Esempio

```
"SELECT * FROM accessi WHERE nome = '" +username+ "';";
```

Dato questo codice SQL, se un malintenzionato dovesse scrivere la variabile *username* in un certo modo, questo statement potrebbe fare qualcosa di inaspettato non previsto dal programmatore.

Per esempio, se impostassimo la variabile *username*:

' OR ' 1 ' = ' 1

Potremmo forzare la selezione di tutti i campi della tabella *accessi* piuttosto che un singolo *username* come era stato specificato dal codice.

GESTIONE NON CORRETTA DEL TIPO

Questo attacco invece si verifica quando non vengono controllati vincoli sul tipo.

Esempio

```
"SELECT * FROM accessi WHERE id = '" +variabile+ "';";
```

In questo caso il campo id, sottointeso come numero intero, potrebbe essere composto da una stringa: un malintenzionato potrebbe impostare *variabile* in questo modo

1; DROP TABLE accessi

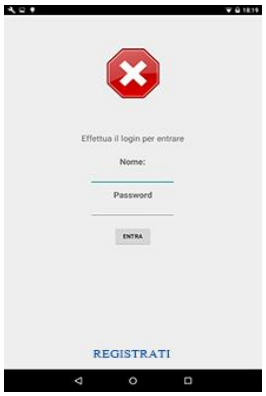
eliminando dal database la tabella *accessi.*

Android e "MyUniversity"

Anche se SQL Injection viene utilizzato per lo più nel mondo web, con il progredire della tecnologia, questo tipo di attacco può essere esteso alle applicazioni mobile.
In particolare ci occuperemo di Android, che può fare uso anche di SQLite (un linguaggio SQL semplificato proprio per questo OS), e anche in questo caso l'attacco ovviamente consiste nell'inserimento di stringhe di codice direttamente in input per manomettere i dati nelle tabelle.
L'attacco quindi consiste nel terminare prematuramente la stringa e concatenarci un altro comando, in maniera che quest'ultimo venga riconosciuto (ed eseguito) prima della fine dell'esecuzione.
Da notare che il comando dannoso aggiunto deve concludersi con la sintassi di commento (--) evitando così di considerare eventuali altri comandi successivi.

Consideriamo la nostra app *MyUniversity*.
Questa applicazione è stata sviluppata con lo scopo di facilitare lo studente durante la propria carriera universitaria. Mediante questa è dunque possibile per lo studente inserire una votazione in caso di superamento di un esame oltre che consentire all'università di creare il piano di studi, lezioni e appelli d'esame con relativi orari, dati che lo studente potrà gestire autonomamente scegliendo quali corsi frequentare e quali appelli sostenere. Quindi all'avvio dell'applicazione si presenta una schermata di LogIn per autorizzare l'accesso agli utenti registrati; si distinguono infatti, come detto, due tipi di utenti con diversi permessi: *studente* e *università*.
E' evidente l'utilizzo di un database, che contiene le seguenti tabelle:

- *piano* – (Piano di studi)
- *sessione* – (Sessioni di esami)
- *orario* – (Orario delle lezioni)
- *seguiti* – (Corsi seguiti)
- *prenotati* – (Appelli prenotati)
- *coordinate* – (Coordinate aule)
- *accessi* – (Username, password e tipo di permesso: *admitted* e *denied*)



Schermata di LogIn

ORARI DI ESAME

Codice	Materia	Aula	Lun	Mar	Mer	Gio	Ven	Cfu
60523	Fisica	G1	08-11		11-13			12
81345	PAA	INFAL1	11-13			8-11		9
73432	Teoria dei sistemi	E1		8-11		14-17		12
80075	Inglese	E0		11-13			10-13	3

Esempio di tabella del database: orari di esame

VULNERABILITA' E SOLUZIONI

Sulla base di quanto detto finora, la nostra applicazione è ovviamente vulnerabile ad un attacco SQL Injection in quanto viene chiesto all'utente di immettere dati input di tipo stringa in vari campi.

Consideriamo quindi l'inserimento di dati nelle varie tabelle.
In particolare nella figura di fianco si può notare che potrebbe avvenire un injection per manomettere il database.
Per quanto riguarda i campi *Codice* e *CFU*, per evitare problemi (tra cui proprio un SQL injection) abbiamo deciso di

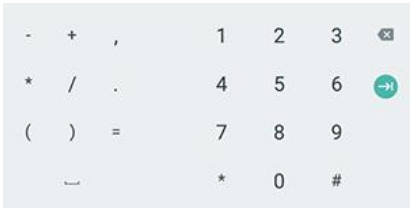
Esempio di inserimento nel database: piano di studi

limitare l'inserimento a soli caratteri numerici; al momento della creazione del database, SQL permette di specificare quale tipo di dato input sia valido per quel campo:

```
sql += "codice INTEGER NOT NULL,";
sql += "materia STRING NOT NULL,";
sql += "cfu INTEGER NOT NULL,";
```

Inoltre quando l'utente dovrà immettere i valori, come ulteriore controllo abbiamo deciso di limitare l'input da tastiera proprio a soli caratteri numerici:

```
<EditText
    android:layout_width="150dp"
    android:layout_height="wrap_content"
    android:inputType="number"
    android:id="@+id/codice_piano"/>
```



Il campo *Materia* invece è più problematico, poiché accetta una stringa che può essere manipolata da un attaccante. Per prevenire quanto detto, si potrebbe filtrare la stringa input in maniera che i caratteri speciali (tra cui ' " – ; ecc.), che potrebbero essere sinonimo di un attacco, vengano riconosciuti e sostituiti codificando la stringa, oppure rifiutando l'input e invitando l'utente ad inserirne un altro valido:

```
String codifica_materia = "null";
try{
    codifica_materia = URLEncoder.encode(materia_piano.getText().toString(), "UTF-8");
}catch(Exception e){}
values.put("materia", codifica_materia);
```

Esempio di codifica Unicode UTF-8 della stringa

Ovviamente in questo modo si evita un SQL Injection attack, ma la stringa codificata (innocua) verrà comunque inserita nel database anche se priva di significato. Si potrebbe pensare quindi di effettuare un altro tipo di controllo sulla stringa tramite l'utilizzo di *Espressioni regolari*, le quali, oltre a convalidare dati, permettono di eseguire una ricerca all'interno del testo che non verrà considerato nel caso di formato non valido.

Come precedentemente accennato, per una maggiore sicurezza, ci è sembrata una buona idea l'implementazione di un sistema di autenticazione (LogIn) per evitare che utenti non autorizzati possano accedere ai dati del database:

```
String sql = "SELECT * FROM accessi WHERE user = " + username.getText().toString() +
        " AND password = " + password.getText().toString() + " AND permesso = admitted";
```

Facendo però un'analisi di sicurezza, anche il LogIn è vulnerabile ad un SQL injection, in quanto un intruso potrebbe servirsi degli espedienti già considerati sopra per forzare l'accesso.
Per rendere sicura l'applicazione, è uso comune l'utilizzo di *compileStatement*, i quali però non garantiscono la totale prevenzione da attacchi:

```
SQLiteStatement stmt = db.compileStatement("SELECT * FROM accessi WHERE" +
        " user = ? AND password = ? AND permesso = ?");
stmt.bindString(1, username.getText().toString());
stmt.bindString(2, password.getText().toString());
stmt.bindString(3, "admitted");
stmt.execute();
```

Dopo la creazione del database, per passare le query in una selezione si utilizzeranno *compileStatement* ('?') e *bind variables* per nascondere i parametri dello statement: questo previene l'alterazione input di un malintenzionato.

II. Content Provider

Cos'è e come funziona

In Android esiste la possibilità di una comunicazione tra diverse applicazioni permettendo ad una di accedere al database di un'altra. Questo meccanismo di condivisione è rappresentato dai *Content Provider*, i quali possono essere immaginati come un "archivio" di dati; il modo in cui questi sono immagazzinati non è rilevante, è importante, invece, come l'applicazione esterna riesca ad accedere ai dati.
Un *Content Provider* ha un comportamento molto simile a quello di un database: possiamo interrogarlo, modificarlo, aggiungere o cancellare contenuti. Comunque sia, si possono usare modi differenti per immagazzinare i dati: in database, in file o anche in rete.
Per creare un *Content Provider* basterà inizializzare alcune costanti che servono per il matching dell'*URI* interrogato da parte del *contentResolver* e fare l'override dei metodi (*getType(), onCreat(), query(), insert(), delete(), update()*).

Prenderemo in considerazioni vulnerabilità derivanti dall'uso dei *Content Provider* e cercheremo di trovare appropriate soluzioni, tuttavia, all'interno della nostra applicazione *MyUniversity*, non è stato implementato questo meccanismo.

Vulnerabilità e soluzioni

Come accennato precedentemente, a causa delle somiglianze tra database e *Content Provider*, i problemi incontrati finora si ripresentano nuovamente. Infatti un'applicazione esterna può fare richieste al nostro database interno tramite *Content Provider*, di cui un malintenzionato potrebbe servirsi per manomettere i dati mediate SQL injection.
Durante un'analisi di sicurezza, l'attenzione cade sui parametri del metodo *query*:

- *uri* : rappresenta l'indirizzo da verificare per il matching delle tabelle (o delle singole righe) all'interno di un certo *Content Provider* (assimilabile a una *FROM*)
- *projection* : rappresenta le colonne selezionate da una query (*SELECT*) elaborate come array di stringa
- *selection* : rappresenta ciò che segue una clausola *WHERE* in una tipica query SQL
- *selectionArgs* : array di stringhe contenente i parametri da passare alla query (sostituendo '?')

Quando verrà invocato il *contentResolver* con l'*URI* corrispondente al contenuto ricercato, il *Content Provider* dovrà restituire il risultato della richiesta. L'injection, quindi, sarà effettuata invocando il metodo *query* con il parametro *selection* impostato con valori non previsti: in questo modo si otterrà la più classica delle injection sulla clausola *WHERE* e si avrà accesso a tutti i dati relativi alla nostra richiesta.
Si potrebbe andare oltre, accedendo anche a colonne non previste dalla query, ma non a tabelle non previste dal *Conten Provider*, in quanto vengono elaborate *on-the-fly* dal *contentResolver* in base all'*URI* che verrà dato in ingresso al metodo *query*.
Per aggirare il problema dell'injection basterebbe validare in maniera corretta i parametri dei metodi utilizzati. In particolare, per quanto riguarda la query potrebbe essere sufficiente utilizzare un costrutto che fissi a priori il parametro *selection*, consentendo alle applicazioni esterne di modificare unicamente *selectionArgs*.

III. Man-in-the-middle (protocollo HTTP)

Cos'è e come funziona

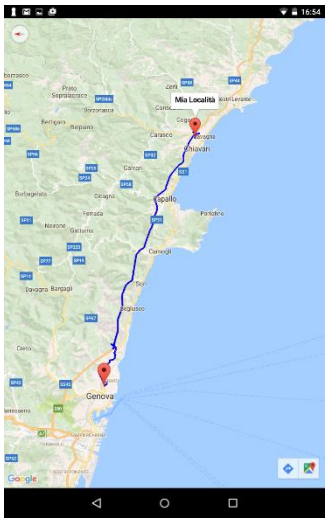
Man-in-the-middle è un attacco in cui un intruso si intromette segretamente in una "conversazione" tra due parti che credono di comunicare tra loro, al fine di recuperare dati confidenziali ed eventualmente ritrasmettere ed alterare le informazioni. Questo tipo di attacco può funzionare solo se entrambe le parti non sono a conoscenza dell'intruso che compromette il collegamento.

Da HTTP a HTTPS

Il protocollo HTTP è un sistema per la trasmissione di informazioni da un web server al browser in uso. La connessione è teoricamente intercettabile e vulnerabile ad un attacco *Man-in-the-middle*, quindi utilizzare tale protocollo per visitare una pagina non è sicuro, nonostante la maggior parte dei siti web venga consultata in suddetta maniera. Questo perché generalmente si suppone che HTTP non venga utilizzato per passare informazioni sensibili che necessiterebbero una totale sicurezza.
Visitando invece una pagina con protocollo HTTPS (ad esempio il sito di una banca), sulla barra degli indirizzi verrà visualizzata l'icona con un lucchetto verde, a conferma che la connessione è autenticata e cifrata, quindi protetta da tentativi di intercettazione e attacchi *Man-in-the-middle*.
Esistono anche pagine a "contenuti misti", cioè accessibili tramite protocollo HTTPS, ma con porzioni HTTP di contenuti che possono essere potenzialmente intercettati, rendendo la pagina parzialmente cifrata, e anche se sembra sicura in realtà non lo è completamente.

Android e "MyUniversity"

Come tutti i dispositivi, ovviamente anche gli Android Device possono stabilire una connessione internet. Consideriamo la nostra app *MyUniversity.*
Come detto precedentemente, questa applicazione permette all'università di creare gli elementi necessari per la carriera dello studente (piano di studi, corsi, orari, esami ecc), il quale potrà gestirli con l'aiuto della stessa. In questo paragrafo introdurremo un'altra funzionalità di *MyUniversity*, grazie al quale l'utente potrà ricevere notizie relative alle attività imminenti. Innanzitutto lo studente dovrà prenotare un evento di suo interesse (corsi o appelli d'esame) e l'applicazione lo avviserà, con i tempi necessari (un paio d'ore prima), tramite notifica: oltre alla funzione di promemoria, tale notifica condurrà, al click, in un'altra *Activity* contenente una *Google Map*. Su tale mappa saranno segnati due *marker* relativi alla nostra posizione attuale e alla destinazione (aula di un esame/corso) e sarà calcolato il percorso da intraprendere mediante decodifica dell'opportuna *polyline*.
Per avere queste informazioni, la nostra applicazione avrà quindi bisogno di stabilire una connessione internet tramite opportune funzioni android.



Mappa con percorso

```
URL url = new URL(stringUrl);
HttpURLConnection httpconn = (HttpURLConnection) url.openConnection();
if (httpconn.getResponseCode() == HttpURLConnection.HTTP_OK) {
    BufferedReader input = new BufferedReader(
            new InputStreamReader(httpconn.getInputStream()),
            8192);
    String strLine = null;

    while ((strLine = input.readLine()) != null) {
        response.append(strLine);
    }
    input.close();
}
```

Per prima cosa, come si può notare dal codice sopra, passiamo al costruttore dell'oggetto *URL* la stringa contenente l'indirizzo che sarà utilizzato in seguito per stabilire la connessione. Così fatta, la stringa effettuerà una richiesta ai server Google, i quali risponderanno con un oggetto *JSON* contenente tutte le informazioni sul percorso. A questo punto, se la connessione sarà andata a buon fine, verrà letto il contenuto del *JSON* e verrà salvato in una stringa. Questa sarà poi passata alla funzione *decodePoly* che mi consentirà di "disegnare" il percorso sulla mappa tra le due posizioni.

E’ evidente lo scambio di informazioni più o meno personali (la nostra posizione) con i server Google, quindi, per quanto detto precedentemente, si potrebbe utilizzare il protocollo HTTPS per prevenire un attacco *Man-in-the-middle* al momento della richiesta ai server.

```
String stringUrl = "https://maps.googleapis.com/maps/api/directions/json?origin="
        + STARTING_POINT.latitude
        + ","
        + STARTING_POINT.longitude
        + "&destination="
        + END_POINT.latitude
        + ","
        + END_POINT.longitude + "&sensor=false";
```

Conclusione

Nonostante i problemi considerati e le soluzioni riportate, la sicurezza informatica (in questo caso Android Security) è un campo ampio e molto complesso in cui entrano in gioco innumerevoli fattori.
Bisogna considerare che con il progredire della tecnologia, aumentano anche i metodi di attacco ai sistemi informatici, che, per quanto sicuri, sono il risultato di una progettazione umana, quindi imperfetta.
La nostra app *MyUniversity* è stata sviluppata in maniera semplice ed essenziale, di conseguenza anche i problemi e le relative soluzioni sono immediati e basilari.

“The only truly secure system is one that is powered off […]”
K. David Mitnick

De Luca Jacopo – matricola 3675883
Rossi Riccardo – matricola 3689831