Sicurezza: SQL Injection

In questo esempio, dimostreremo una vulnerabilita di SQL injection usando il progetto di WASA (Web and software architecture), con opportune modifiche per renderlo vulnerabile a questo tipo di attacco.

La SQL Injection è un tipo di vulnerabilità di sicurezza che consente a un utente malintenzionato di eseguire comandi SQL arbitrari sul database di un'applicazione. Questo avviene quando i dati forniti dall'utente vengono concatenati direttamente alle query SQL senza un'adeguata sanitizzazione.

Struttura del progetto

Il progetto WASAPhoto su cui proveremo queste query e' un esempio di social media, che permette ad utenti di condividere immagini, seguire profili a cui sono interessati, e avere un feed personale di post. Nella versione originale del progetto, tutti gli input vengono sanitizzati per mitigare possibili vulnerabilita'. A scopo didattico, queste misure di sicurezza sono state rimosse dalla pagina di login, permettendo ad utenti malintenzionati di svolgere SQL injection.

Esempio pratico

Analizziamo la funzione *CreateUser* presente nel nostro database:

```go
/*
CreateUser creates a new user in the database.

If the user already exists, it returns the user schema.
Otherwise, it creates it and returns the user schema.

NOTE: This function is vulnerable to SQL Injection attacks.
*/
func (db *appdbimpl) CreateUser(user User) (User, error) {
	// Vulnerable code: Directly inserting user input into the SQL query
	query := fmt.Sprintf(`INSERT OR IGNORE INTO Users(username) VALUES ('%s');`,
				user.Username)

	// Execute the query
	_, err := db.c.Exec(query)
	if err != nil {
		return User{}, err
	}

	// Fetch the user details using a similar vulnerable method
	query = fmt.Sprintf(`SELECT * FROM Users WHERE username = '%s';`, user.Username)
	row := db.c.QueryRow(query)

	err = row.Scan(&user.UserId, &user.Username)
	if err != nil {
		return User{}, err
	}

	return user, nil
}
```

Piuttosto che usare una query parametrizzata, in questo caso stiamo concatenando direttamente lo username otenuto dalla schermata di login. Un attaccante puo' quindi inserire delle query formattate per includere comandi aggiuntivi, ad esempio una *DROP TABLE* per cancellare la tabella *Users*.

Esempi di attacchi

Consideriamo la seguente tabella *Users*:

	userId	username
1	1	Mark
2	2	Zoe

Ogni volta che un nuovo utente si registra, viene aggiunto in una nuova riga.

Eliminazione di una Tabella

Un utente malintenzionato può inserire come username una stringa del tipo:
Bob'); DROP TABLE Users;--.

Questa stringa manipolerà la query SQL in modo tale da eseguire un comando per eliminare la tabella *Users*, come mostrato nella query risultante:
INSERT OR IGNORE INTO Users(username) VALUES ('Bob'); DROP TABLE Users;--');

Modifica dei Dati di Altri Utenti

In alternativa, un attaccante può anche modificare i dati di altri utenti. Ad esempio, utilizzando la seguente stringa:
Alice'); UPDATE Users SET username='Mark' WHERE username='Zoe';--.

Questa input viene interpretata come:
INSERT OR IGNORE INTO Users(username) VALUES ('Alice'); UPDATE Users SET username='Mark' WHERE username='Zoe';--');

In questo caso, l'attaccante può modificare il nome utente di *Zoe* in *Mark*. Questo dimostra come la vulnerabilità di SQL Injection possa essere sfruttata non solo per cancellare dati, ma anche per alterare le informazioni presenti nel database.

Tecnica del Piggybacking

La tecnica di *piggybacking*, utilizzata in questi esempi, consiste nell'agganciare una query SQL aggiuntiva a una query già esistente attraverso un input fornito dall'utente. Questo avviene quando l'input dell'utente non è correttamente sanitizzato e viene inserito direttamente nella query SQL. Il risultato è che la query aggiuntiva viene eseguita insieme alla query originale, permettendo all'attaccante di eseguire operazioni non autorizzate sul database.

Conclusione

Questa vulnerabilita' permette all'attaccante di compromettere l'**integrita'** dei dati (poiche' puo modificarli o cancellarli liberamente) e la loro **confidenzialita'** (possiamo iniettare una *SELECT* per leggere l'intera tabella, ottenendo gli *userId* che dovrebbero essere inaccessibili da un utente).

La mitigazione di questo tipo di attacchi e' relativamente semplice grazie alle teconologie odierne. Esistono diversi strumenti di analisi (ad esempio sqlmap) capaci di individuare query vulnerabili, permettendoco di agire in modo preventivo e svolgere test mirati su un determinato database.