SQL injection

bad code practices e conseguenze

Francesco James Fanti

9 maggio 2024

Indice

1 Di cosa si tratta?

Il programma fornito è una simulazione di login-page di una webapp (si noti che, per ovvie ragioni, viene omessa la gestione sicura delle password che non sono criptate e vengono passate in chiaro nelle richieste http; anche la componente di sicurezza relativa ai protocolli di rete è omessa) tramite questa *Very Secure WebApp* è possibile tentare di violare delle proprietà della CIA Triad.

1.1 Riguardo i riferimenti e le immagini

Ho provvisto di link tutte le citazioni nel testo che contengono riferimenti a persone; software; utility. Il libro di testo [1] da cui sono integrate le nozioni su cui si basa l'esperimento verrà citato opportunamente come fonte. Le immagini - tutte di mia personale produzione - usate per descrivere i passaggi dell'esperimento, sono poste in un capitolo a parte e consultabili tramite comodi collegamenti ipertestuali. Questa scelta è stata necessaria per ottimizzare la gestione dello spazio automatica di LaTeX ed evitare un documento poco piacevole alla vista.

1.2 Tipo di attacco

Tenteremo di minare la *Confidentiality*, superando la pagina di login senza conoscere le credenziali; potenzialmente anche l'*Integrity* potendo decidere di modificare i dati a cui si ha accesso; successivamente vogliamo violare la *Availability* andando a eliminare la tabella degli utenti nel database. Il risultato dell'attacco sarà un server non funzionante.

1.3 Ambiente dell'attacco

L'applicazione web è costruita utilizzando:

- Go per far girare un server minimale e un'handler API, come backend.
- Vue.js per la UI lato utente (la pagina di login effettiva).

Sfrutteremo una serie di bug nel codice del backend e nel modo in cui si interfaccia con il frontend per eseguire un attacco SQLi. Per mantenere consistenza e portabilità su più macchine, il tutto viene eseguito tramite container Docker. In particolare utilizzeremo due immagini docker, una per il backend e una per il frontend. L'immagine di backend utilizza fa girare il server in go tramite net/http che ascolta le richieste del frontend su `:3000`, l'immagine di frontend utilizza un'istanza Nginx che viene mappata su `:8080` (invece dello standard di Nginx che è `:80`).

1.3.1 Motivazioni riguardo la scelta di questo ambiente

Seguendo il corso di Web And Software Architecture (WASA) mi sono trovato a dover costruire un'API REST in Go+Vue. Durante la scrittura del backend

sono finito sulla documentazione di Go riguardo l'uso dei database, notando con interesse la sezione riguardante il modo corretto di gestire le query per evitare SQL injection. Di fatto, una query costruita dinamicamente dovrebbe essere costruita tramite placeholder:

```
rows, err := db.Query("SELECT * FROM user WHERE id = ?", id)
```
(in Go il placeholder è il `?`)

In questa maniera, una stringa malevola inserita al posto del placeholder non dovrebbe essere interpretata come uno statement, ma soltanto come una stringa (venendo sempre trattata come se fosse tra virgolette, insomma). Ma cosa accadrebbe se invece decidessimo - per qualsiasi motivo (fosse anche solo distrazione o ignoranza rispetto alle best practices) - di costruire le query tramite una funzione di formattazione del testo?
In `sqli-demo` la query del login è costruita così:

```
query := fmt.Sprintf(‘SELECT id, username
                        FROM users
                        WHERE username = ’\%s’
                        AND password = ’\%s’;‘,
username, password) }
```

questo significa che l'input dell'utente non riceve nessun controllo di validità né *sanitization*, permettendoci di utilizzare il form di login per iniettare comandi malevoli direttamente nel database.

1.3.2 Note riguardandi il progetto

Essendo stato ispirato all'utilizzo di Go durante il lavoro svolto per il corso WASA del professor Panizzi, La struttura del progetto è costruita in maniera simile a Fantastic coffee (decaffeinated). Tuttavia non ho usato direttamente il template, ma ho ricostruito una struttura di directory simile, oltre che a un metodo di distribuzione quasi 1:1 rispetto a quello utilizzato per WASA.

2 Codice

Abbiamo visto brevemente la struttura della query vulnerabile, descriveremo ora alcuni punti interessanti del codice del progetto, per chiarimenti ulteriori la repo è documentata e commentata in inglese.
Il **main** si occupa di poche cose:

- Controlla se il file del database esiste:
  - se esiste si limita ad aprire la connessione col DB.
  - altrimenti lo crea e inizializza dei dati demo al suo interno.
- Incapsula il puntatore al DB in una struttura e lo passa ad un handler per il login.

- Mette in ascolto il server su `:3000`.

L'unico handler presente in `api/handlers.go` si occupa di gestire la richiesta `POST` per l'endpoint `/login`. Vengono abilitate le richieste Cross Origin per i metodi `POST` e `OPTION` in modo da garantire il preflight e l'effettiva richiesta per il login. La funzione handler passa username e password (non hashata per motivi di demo, ma supponiamo un real use case in cui è gestita decentemente almeno la password) alla funzione del package database che si occupa delle query.

2.1 Major bugs nel codice del database package

La funzione LogIn del package database presenta le vulnerabilità maggiori che andremo a sfruttare.

- La query è costruita **senza placeholder** venendo assemblata direttamente:

```
fmt.Sprintf(‘SELECT id, username
              FROM users
              WHERE username = ’\%s’
              AND password = ’\%s’;‘,
username, password)
```

- Sia che torni un errore generico sia nel caso in cui la query non dovesse tornare risultati, passa un errore all'handler **che lo scrive direttamente nella risposta http**

Il ruolo di punto di accesso per questo attacco è quindi quello della funzione LogIn, ma la pratica - sbagliatissima, ma allettante per un programmatore frettoloso o non conscio dei pericoli per avere un log diretto degli errori sul frontend - di scrivere un errore non filtrato e verboso nella risposta http è il punto cruciale che permette ad un attaccante di capire che può sfruttare un attacco SQLi. L'altra fondamentale vulnerabilità (e design scadente) è che i dati prelevati dalla query di login sono inseriti in una stringa all'interno di un ciclo che itera sul risultato della query e poi passati al frontend. Questo design fa sì che dal frontend si possa avere log di più di un risultato per query, esponendo potenzialmente dati sensibili.

3 Attacco

3.1 Funzionamento atteso del programma

Al momento della creazione il database contiene quattro utenti di prova e le rispettive password:

id	username	password
1	piero	s3cret
2	giorgio	prova!
3	susan	l33tPasswd!
4	brian	cann0tfind!

Possiamo infatti notare che inserendo delle credenziali valide tra queste, riusciremo a loggare (è naturalmente possibile leggere la password in chiaro nel form eliminando il campo HTML `type="password"` da ispeziona elemento).

Ad esempio possiamo eseguire un login legittimo con `username = piero` e `password = s3cret`. Come visibile da [1].

3.2 Fase preliminare

Cosa succede però se proviamo a loggare con un utente che non esiste? Logghiamo ad esempio con `username = admin` e `password = admin`. Analizzando i risultati in [2] possiamo vedere come il log ci dia un interessante suggerimento: "`user admin not found`". Facilmente ipotizziamo come la ricerca dello user avviene su una tabella `users` (cosa piuttosto banale e comune, d'altronde). Lo terremo a mente per la seconda parte dell'attacco.

3.3 Payload

Andiamo ora a vedere cosa succede con l'inserimento di query malevole. Sarà integrato l'utilizzo combinato del *end-of-line comment* anche in altri payload per tagliare fuori dalla query il controllo della password o altri parametri.

3.3.1 Tautology

Il primo attacco sarà di tipo **tautology**, otterremo l'accesso se lo username è uguale a un placeholder o se $1 = 1$ (sempre vero). Il comando verrà iniettato tramite il box username del form di login, è indifferente quello che viene inserito nel box password dato che la commenteremo via dalla query. Inserendo il payload:

```
_' OR 1 = 1 --
```

Siamo effettivamente riusciti a superare la pagina di login senza conoscere credenziali di accesso [3]. La query che è stata eseguita dal DBMS ha questa forma:

```
SELECT id, username FROM users
                    WHERE username = '_'
                    OR 1 = 1 --'
                    AND password = '';
```

Abbiamo potenzialmente violato la *Confidentiality* bypassando i controlli di accesso, ma effettivamente non siamo loggati su nessun account con dati sensibili, possiamo fare di meglio.

3.3.2 EOL comment

Notiamo che i console log del frontend hanno due comportamenti diversi a seconda del login di successo che effettuiamo [4]. Questo ci dà accesso agli username degli utenti reali (si noti che probabilmente su un sito gli username degli utenti sono comunque visualizzabili in una qualche maniera, rendendo questa debolezza estrema del programma - inserita, ovviamente ad hoc - probabilmente non così necessaria ai fini del nostro attacco).

Eseguiamo un secondo attacco, questa volta di tipo **End-of-line comment** questa volta tentando di loggare come l'utente `susan`, non conosciamo la sua password, ma possiamo sfruttare caratteristiche della query utilizzata in precedenza. Se nel campo username inseriamo:

```
susan' --
```

Inserendo questo payload, la query finale eseguita dal DBMS ha questa forma:

```
SELECT id, username FROM users
                    WHERE username = 'susan'
                    --' AND password = '';
```

Abbiamo richiesto un utente valido e tolto il campo password dalla query rendendolo un commento. Otterremo la schermata di successo e il log "corretto" dal DB che ha selezionato il campo id, e username richiesti dal DB. Per la logica (erronea) del nostro programma abbiamo eseguito con successo l'accesso come `susan`[5] . Stiamo violando sia la *Confidentiality* che potenzialmente l'*Integrity* considerando che possiamo sia accedere a dati riservati senza autorizzazione, sia - volendo - manipolarli a nostro piacere (ovviamente sulla nostra demo non ci sono dati da manipolare).

3.3.3 Piggyback query

Abbiamo potuto intuire, grazie alla disattenzione del programmatore inesperto, che esiste una tabella `users`. Possiamo sfruttare l'assenza di controlli ulteriori sulla query (e il modo erroneo in cui viene costruita dal backend) per apporre una seconda query all'interno di quella di login. Eseguiremo un attacco di tipo **piggyback query** per cancellare dal DB la tabella degli utenti. Inietteremo nella query, sempre dal box dello username, questa stringa:

```
_'; DROP TABLE users --
```

così facendo verrà tentato un login per un utente che non esiste, lo statement della query sarà terminato dal punto e virgola e verrà eseguita un'altra query per eliminare la table degli utenti, alla fine di questo statement abbiamo inserito

un EOL comment per bypassare il controllo originale della password. La query eseguita ha questa forma:

```
SELECT id, username FROM users
                    WHERE username = '_';
DROP TABLE users --' AND password = '';
```

Questo ci porta a una serie di risultati interessanti, il login da errore [6], ce lo aspettiamo, non stiamo provando ad accedere. Cosa accade però se riproviamo a loggare come `piero` (che sia tramite la sua password oppure tramite EOL comment). Il tentativo di login legittimo come utente `piero` e come qualsiasi altro utente restituisce questo log al frontend [7]. Abbiamo eliminato la tabella `users` con successo. Siamo riusciti a violare la *Availability* rendendo il login non funzionante e potenzialmente eliminando i dati degli utenti (o anche solo il loro punto d'accesso) minando indirettamente di nuovo anche l'*Integrity*.

4 Conclusioni

Tramite la nostra dimostrazione abbiamo potuto constatare l'importanza dello scrivere codice sicuro, seguendo sempre le best practices di sicurezza. Una scrittura frettolosa e disattenta del codice che interfaccia un database o la disattenzione nella lettura della documentazione (come dicevo, la doc ufficiale di Go ci spiega esplicitamente come NON fare errori di questo tipo nelle query) può esporre la nostra applicazione a SQL injection. Oltre i tre payload - banali - che ho utilizzato, è facile reperirne, studiarne e testarne moltissimi semplicemente esplorando repository dedicate alla cybersecurity. Risulta evidente che la gestione sicura dei database sia un punto cruciale nell'ambito della sicurezza delle applicazioni che scriviamo. La SQL injection oltre ad essere un attacco molto comune è anche un attacco potenzialmente molto severo, di fatto abbiamo visto come può minare le tre proprietà cardine della CIA Triad.

5 Se volessi riprodurre l'esperimento?

Una copia della repo di `sqli-demo` (al momento privata, ma in futuro pubblica sul mio github) viene allegata insieme al presente documento. Per eseguire la webapp abbiamo bisogno di un'installazione di Docker, possibilmente in ambiente `gnu+linux`. Builderemo le immagini docker con:

```
$ docker build -t sqliback:latest -f Dockerfile.backend .
$ docker build -t sqlifront:latest -f Dockerfile.frontend .
```

e le avvieremo da due istanze della nostra shell con:

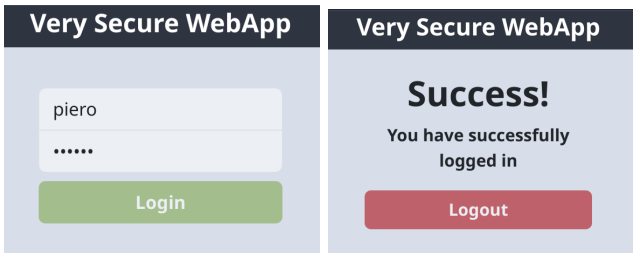
```
$ docker run -it --rm -p 3000:3000 sqliback:latest
$ docker run -it --rm -p 8080:80 sqlifront:latest
```

(Il flag `--rm` in particolare ci permette di far sì che il container venga svuotato a fine esecuzione, ricostruendo il database e fornendo un ambiente vergine, pronto ai test,

ad ogni avvio. Probabilmente il flag `-it` è superfluo a meno che non si voglia aprire un'altra shell e navigare tra le directory per controllare il file del database ecc.)

6 Galleria immagini

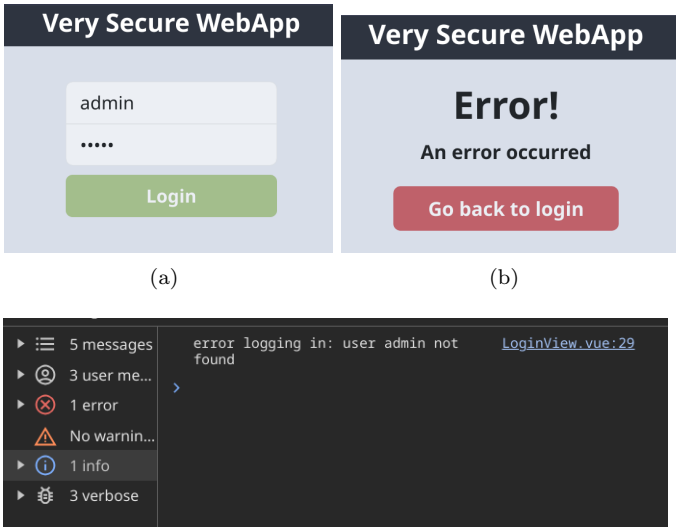
Le immagini sono poste a fondo relazione per lasciare a LaTeX la possibilità di ottimizzare lo spazio al meglio, sono consultabili tramite riferimenti e forniscono etichette per tornare al punto del testo da cui si è arrivati.



(a) (b)

Figura 1: (a) inserisco le credenziali (b) riesco a loggare

torna su

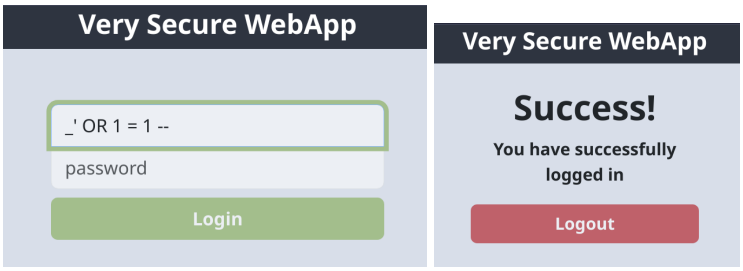


(a) (b)

(c)

Figura 2: (a) inserisco le credenziali (b) mi avverte del fallimento (c) c'è un log decisamente interessante e verboso

torna su

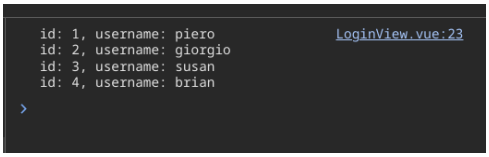


(a) (b)

Figura 3: (a) inserisco il payload (b) riesco a loggare

torna su

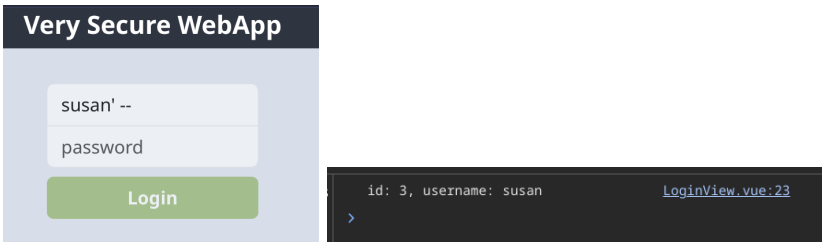
(a)



(b)

Figura 4: (a) i log di un login legittimo (b) log dell'accesso con tautologia, espongono tutti gli username

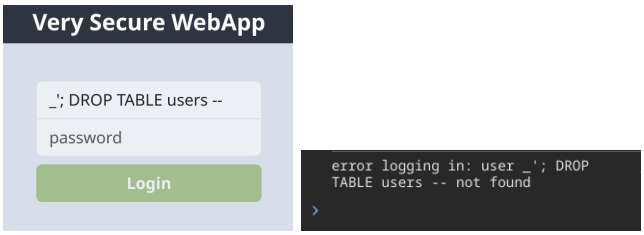
torna su



(a) (b)

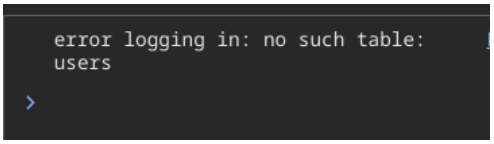
Figura 5: (a) query malevola per il login di susan (b) log dell'accesso con EOL comment

torna su



(a) (b)

Figura 6: (a) query malevola per piggyback (b) log dell'errore dopo la query
torna su



(a)

Figura 7: (a) query malevola per piggyback
torna su

Riferimenti bibliografici

[1] William Stallings and Lawrie Brown. *Computer Secutiry*. Pearson, fourth edition, 2018.