UNIVERSITÀ DEGLI STUDI DELLA BASILICATA

SCUOLA DI INGEGNERIA

CORSO DI LAUREA MAGISTRALE IN INGEGNERIA INFORMATICA

E DELLE TECNOLOGIE DELL'INFORMAZIONE

RELAZIONE PROGETTO TAV

Secure Software Design: Design Pattern per la Sicurezza by Design

Docente:

Ch.mo Prof. Gianvito Summa

Studenti:

Michael Pio Stolfi 68787

Antonella Bonelli 68791

ANNO ACCADEMICO 2024-2025

Sommario

Introduzione

La sicurezza delle applicazioni è un aspetto fondamentale nello sviluppo software, in particolare per le applicazioni che gestiscono dati sensibili e informazioni critiche. Questo report si propone di analizzare l'impatto dei design pattern sulla sicurezza di un'applicazione web, confrontando due versioni della stessa: una sviluppata senza design pattern e un'altra che implementa pattern mirati a mitigare vulnerabilità comuni.

L'analisi è stata condotta attraverso test di sicurezza focalizzati sulle vulnerabilità più critiche identificate dall'OWASP Top 10, tra cui "SQL Injection" e "Broken Access Control". Per condurre i penetration test è stato utilizzato "Burp Suite", una piattaforma avanzata che consente di intercettare e modificare le richieste HTTP per individuare e sfruttare eventuali punti deboli dell'applicazione.

Il progetto oggetto di studio è un sistema di gestione dell'autenticazione del personale aziendale, che prevede la gestione di tesserini digitali e il loro aggiornamento con diversi livelli di accesso.

Lo scopo di questo progetto è di dimostrare come l'adozione di design pattern possa contribuire a migliorare la sicurezza by design, limitando l'esposizione dell'applicazione a vulnerabilità note e migliorando la robustezza del sistema contro attacchi informatici. Attraverso test automatici e manuali, sono stati raccolti dati concreti sulle differenze di sicurezza tra le due implementazioni, permettendo di valutare l'efficacia delle tecniche di protezione adottate.

Nei capitoli successivi verranno presentati i dettagli del progetto, i metodi di testing impiegati e i risultati ottenuti, evidenziando i vantaggi dell'approccio basato sui design pattern nella prevenzione delle vulnerabilità più critiche.

Processo di Sviluppo

Il processo di sviluppo adottato per questo progetto si è focalizzato sulla valutazione dell'impatto dei design pattern sulla sicurezza del software, con particolare attenzione alla mitigazione di alcune delle vulnerabilità indicate da OWASP nella "Top Ten - 2021". A tal fine, sono stati sviluppati due backend distinti utilizzando il linguaggio Java con il framework Quarkus:

- Backend senza Design Pattern: sviluppato senza l'adozione esplicita di pattern architetturali, per osservare le potenziali vulnerabilità in un'applicazione priva di strutture progettuali consolidate;
- Backend con Design Pattern: implementato a partire dal primo aggiungendo una selezione di pattern progettuali mirati a mitigare specifiche vulnerabilità.

Entrambi i backend utilizzano una istanza "containerizzata" di PostgreSQL come database, scelto per analizzare la resilienza dell'applicazione alla SQL Injection. Il frontend, sviluppato in plain JavaScript, è condiviso tra i due backend, per garantire un confronto uniforme delle funzionalità di business e delle vulnerabilità esposte.

Casi d'Uso Sviluppati e Rischi Associati

L'applicazione sviluppata è un sistema per la gestione dell'autenticazione del personale di un'azienda, progettato per permettere l'accesso ai dati dei dipendenti attraverso tesserini digitali. Ogni tesserino contiene informazioni identificative e il livello dell'utente all'interno dell'azienda. L'obiettivo principale dell'applicazione è assicurare la protezione di questi dati sensibili e prevenire accessi non autorizzati.

Il sistema implementa le seguenti funzionalità:

Funzionalità	Descrizione	Vulnerabilità OWASP
Login	L'utente accede al sistema con username e password	• Broken Access Control • SQL Injection • Identification and Authentication Failures

		• Security Logging and Monitoring Failures
Visualizza dati utente nella Dashboard	Mostra i dati personali e il livello dell'utente	• Security Logging and Monitoring Failures
Modifica dati utente nella Dashboard	Gli utenti possono aggiornare solo le proprie informazioni	• Broken Access Control • SQL Injection • Security Logging and Monitoring Failures

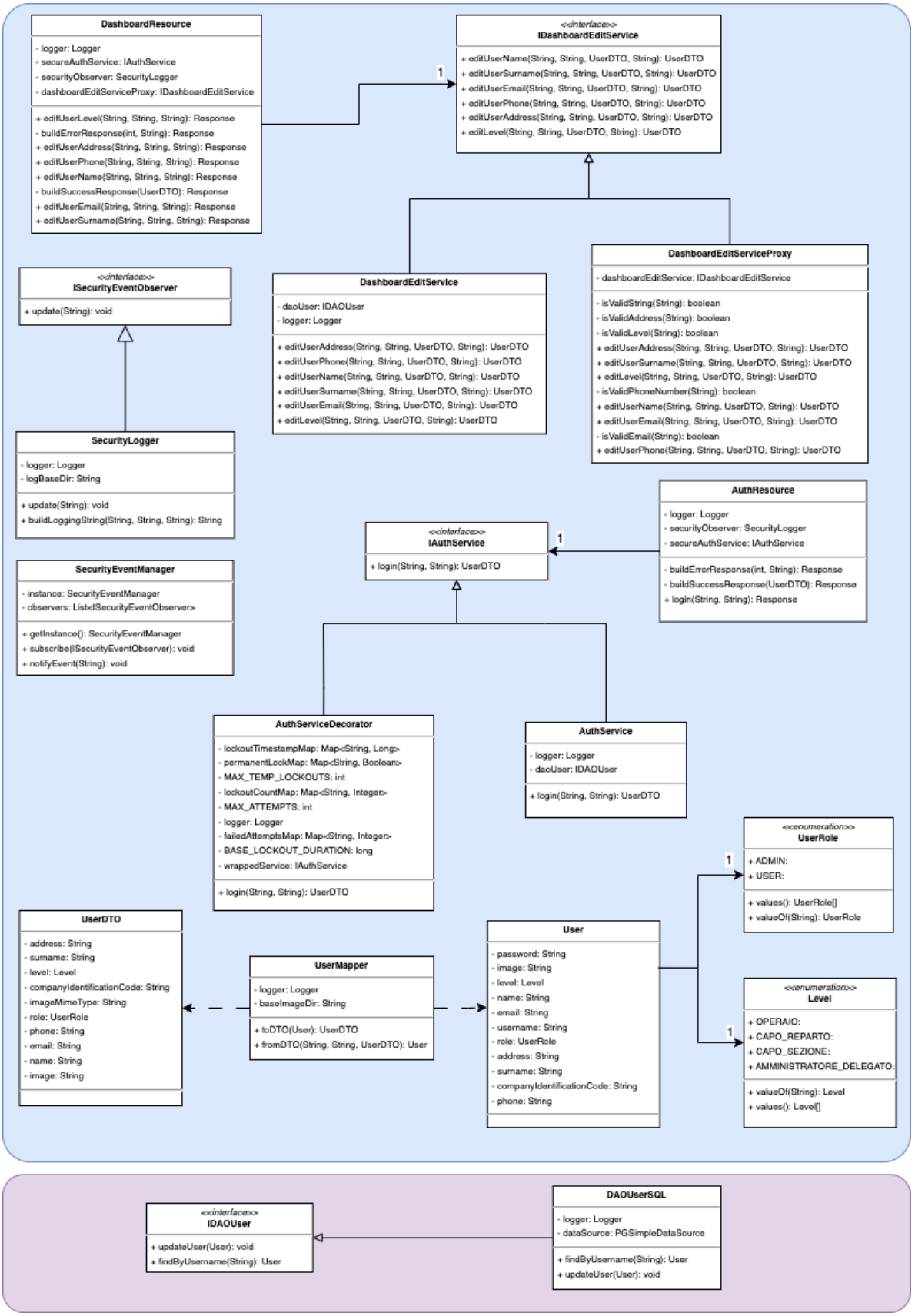
Ogni dipendente può visualizzare il proprio tesserino digitale, che include dati sensibili. Tuttavia, se non gestita correttamente, questa funzionalità può esporre il sistema a vulnerabilità. Uno dei rischi principali è la Sensitive Data Exposure, che si verifica quando dati sensibili vengono accidentalmente esposti nel frontend o nelle risposte API.

Un attaccante potrebbe tentare di forzare le credenziali attraverso un attacco brute force, sfruttando la mancanza di limitazioni sui tentativi di login. Un altro rischio rilevante è la SQL Injection, che potrebbe essere sfruttata per manipolare le query al database e ottenere accesso non autorizzato ai dati sensibili.

L'uso di design pattern, nel caso del sistema demo sicuro ha permesso di implementare controlli più robusti e ridurre i rischi sopra elencati

Design Pattern Applicati al sistema Demo Sicuro

Il diagramma UML del sistema demo sicuro rappresenta visivamente l'architettura del sistema, evidenziando come i design pattern selezionati, che verranno presentati di seguito, interagiscono per garantire la sicurezza.


DashboardResource
- logger: Logger
- secureAuthService: IAuthService
- securityObserver: SecurityLogger
- dashboardEditServiceProxy: IDashboardEditService
+ editUserLevel(String, String, String): Response
- buildErrorResponse(int, String): Response
+ editUserAddress(String, String, String): Response
+ editUserPhone(String, String, String): Response
+ editUserName(String, String, String): Response
- buildSuccessResponse(UserDTO): Response
+ editUserEmail(String, String, String): Response
+ editUserSurname(String, String, String): Response
1
«interface»
IDashboardEditService
+ editUserName(String, String, UserDTO, String): UserDTO
+ editUserSurname(String, String, UserDTO, String): UserDTO
+ editUserEmail(String, String, UserDTO, String): UserDTO
+ editUserPhone(String, String, UserDTO, String): UserDTO
+ editUserAddress(String, String, UserDTO, String): UserDTO
+ editLevel(String, String, UserDTO, String): UserDTO
«interface»
ISecurityEventObserver
+ update(String): void
DashboardEditService
- daoUser: IDAOUser
- logger: Logger
+ editUserAddress(String, String, UserDTO, String): UserDTO
+ editUserPhone(String, String, UserDTO, String): UserDTO
+ editUserName(String, String, UserDTO, String): UserDTO
+ editUserSurname(String, String, UserDTO, String): UserDTO
+ editUserEmail(String, String, UserDTO, String): UserDTO
+ editLevel(String, String, UserDTO, String): UserDTO
DashboardEditServiceProxy
- dashboardEditService: IDashboardEditService
- isValidString(String): boolean
- isValidAddress(String): boolean
- isValidLevel(String): boolean
+ editUserAddress(String, String, UserDTO, String): UserDTO
+ editUserSurname(String, String, UserDTO, String): UserDTO
+ editLevel(String, String, UserDTO, String): UserDTO
- isValidPhoneNumber(String): boolean
+ editUserName(String, String, UserDTO, String): UserDTO
+ editUserEmail(String, String, UserDTO, String): UserDTO
- isValidEmail(String): boolean
+ editUserPhone(String, String, UserDTO, String): UserDTO
SecurityLogger
- logger: Logger
- logBaseDir: String
+ update(String): void
+ buildLoggingString(String, String, String): String
AuthResource
- logger: Logger
- securityObserver: SecurityLogger
- secureAuthService: IAuthService
- buildErrorResponse(int, String): Response
- buildSuccessResponse(UserDTO): Response
+ login(String, String): Response
«interface»
IAuthService
+ login(String, String): UserDTO
1
SecurityEventManager
- instance: SecurityEventManager
- observers: List<ISecurityEventObserver>
+ getInstance(): SecurityEventManager
+ subscribe(ISecurityEventObserver): void
+ notifyEvent(String): void
AuthServiceDecorator
- lockoutTimestampMap: Map<String, Long>
- permanentLockMap: Map<String, Boolean>
- MAX_TEMP_LOCKOUTS: int
- lockoutCountMap: Map<String, Integer>
- MAX_ATTEMPTS: int
- logger: Logger
- failedAttemptsMap: Map<String, Integer>
- BASE_LOCKOUT_DURATION: long
- wrappedService: IAuthService
+ login(String, String): UserDTO
AuthService
- logger: Logger
- daoUser: IDAOUser
+ login(String, String): UserDTO
«enumeration»
UserRole
+ ADMIN:
+ USER:
+ values(): UserRole[]
+ valueOf(String): UserRole
1
UserDTO
- address: String
- surname: String
- level: Level
- companyIdentificationCode: String
- imageMimeType: String
- role: UserRole
- phone: String
- email: String
- name: String
- image: String
UserMapper
- logger: Logger
- baseImageDir: String
+ toDTO(User): UserDTO
+ fromDTO(String, String, UserDTO): User
User
- password: String
- image: String
- level: Level
- name: String
- email: String
- username: String
- role: UserRole
- address: String
- surname: String
- companyIdentificationCode: String
- phone: String
«enumeration»
Level
+ OPERAIO:
+ CAPO_REPARTO:
+ CAPO_SEZIONE:
+ AMMINISTRATORE_DELEGATO:
+ valueOf(String): Level
+ values(): Level[]
1
«interface»
IDAOUser
+ updateUser(User): void
+ findByUsername(String): User
DAOUserSQL
- logger: Logger
- dataSource: PGSimpleDataSource
+ findByUsername(String): User
+ updateUser(User): void

Pattern: Proxy

Vulnerabilità mitigata: Broken Access Control

Motivazione: La Broken Access Control si verifica quando gli utenti possono accedere o modificare risorse senza le autorizzazioni appropriate. Un protection proxy aiuta a mitigare questa vulnerabilità perché:

- Centralizza i controlli di accesso, evitando implementazioni distribuite e incoerenti;
- Previene l'accesso non autorizzato verificando il ruolo dell'utente prima di inoltrare le richieste.

Utilizzo: nel sistema demo sicuro il protection proxy viene utilizzato per proteggere le operazioni di modifica dei dati.

Descrizione: il proxy pattern è un pattern strutturale che funge da intermediario tra client e server per controllare l'accesso a una risorsa. Nel contesto della sicurezza, viene utilizzato un protection proxy, un tipo specifico di proxy appartenente alla categoria dei decorator proxy, per intercettare e filtrare le richieste verso risorse sensibili.

Il protection proxy si interpone tra il client e l'oggetto reale, controllando le richieste prima di inoltrarle. In particolare:

- Intercetta la richiesta di accesso a una risorsa o un metodo;
- Verifica i privilegi dell'utente, confrontando il suo ruolo con le autorizzazioni richieste;
- Concede o nega l'accesso, inoltrando la richiesta all'oggetto reale solo se l'utente è autorizzato.

Questo meccanismo è utile per rafforzare i controlli di accesso ed evitare vulnerabilità come la Broken Access Control, identificata come la principale minaccia nell'OWASP Top 10.

Pattern: DAO (Data Access Object) con Prepared Statement

Vulnerabilità ridotta: SQL Injection

Motivazione: la scelta di adottare il DAO Pattern integrato con i prepared statements è motivata da vari fattori:

- Separazione della persistenza;
- Riduzione della vulnerabilità agli attacchi di SQL injection;
- Manutenibilità e scalabilità.

Utilizzo: nel sistema demo sicuro il DAO con prepared statement viene utilizzato per gestire la persistenza delle utenze.

Descrizione: il DAO pattern (Data Access Object) è un pattern architetturale che si occupa di separare la logica di accesso ai dati dal resto dell'applicazione. Questo approccio consente di isolare le operazioni di persistenza in un componente dedicato, facilitando la manutenzione, il testing e la scalabilità del sistema.

Nel contesto della sicurezza, l'utilizzo di query parametrizzate attraverso i prepared statements all'interno del DAO si rivela fondamentale per mitigare il rischio di SQL Injection. Invece di concatenare dinamicamente le stringhe che compongono le query SQL, i prepared statements permettono di definire le istruzioni SQL con dei segnaposto per i parametri. Quando questi parametri vengono poi associati alla query, il database li tratta esclusivamente come dati, senza interpretarli come parte della logica della query. Questo meccanismo garantisce che input malevoli non possano alterare la struttura della query, riducendo significativamente la possibilità di attacchi di injection.

Pattern: Decorator

Vulnerabilità ridotta: Identification and Authentication Failures

Motivazione: le vulnerabilità legate alla mancata gestione sicura dell'identificazione e dell'autenticazione (ad esempio, rate limiting, logging, meccanismi di lockout dopo tentativi falliti) possono essere mitigate con l'uso del decorator pattern. Questo pattern consente di avvolgere il processo di autenticazione di base con controlli di sicurezza aggiuntivi, senza modificarne l'implementazione centrale.

Utilizzo: nel sistema demo sicuro il decorator pattern viene utilizzato per gestire il blocco delle utenze a seguito di login falliti.

Descrizione: il decorator pattern è un design pattern strutturale che consente di estendere dinamicamente il comportamento di un oggetto senza modificarne il codice sorgente. Questo viene realizzato avvolgendo l'oggetto originale con una serie di classi che implementano la stessa interfaccia e aggiungono funzionalità incrementali. Il pattern si basa sulla composizione piuttosto che sull'ereditarietà, migliorando la flessibilità e la manutenibilità del codice.

Pattern: Observer

Vulnerabilità ridotta: Security Logging and Monitoring Failures

Motivazione: l'adozione del pattern observer in ambito sicurezza è motivata dai seguenti aspetti:

- il sistema di monitoraggio viene centralizzato;
- migliora il disaccoppiamento e la scalabilità.

Utilizzo: nel sistema demo sicuro l'observer viene utilizzato per implementare un sistema di monitoraggio degli accessi.

Descrizione: il pattern prevede due ruoli principali:

- L'oggetto che genera e pubblica eventi, come ad esempio il rilevamento di accessi non autorizzati o modifiche critiche nel sistema.
- Osservatori, componenti sottoscritti all' oggetto, che vengono notificati automaticamente quando si verifica un evento rilevante.

Quando un evento critico si verifica, il soggetto notifica tutti gli osservatori, permettendo così una tracciabilità efficace e una risposta tempestiva agli incidenti. In questo modo, ogni evento di sicurezza viene registrato e analizzato, contribuendo a una visione completa dello stato del sistema.

Strategia di Testing

Dopo lo sviluppo delle due implementazioni, è stata definita una strategia di testing mirata a confrontare la sicurezza dei due approcci. Il processo di testing ha incluso:

- Unit Test, per verificare il corretto funzionamento delle componenti chiave;
- Penetration Test, per simulare attacchi reali e individuare vulnerabilità.

I risultati di questi test, descritti nei paragrafi successivi, hanno fornito un'analisi quantitativa e qualitativa dei miglioramenti ottenuti in termini di security by design grazie all'utilizzo dei design pattern.

Unit Test

Di seguito sono riportati gli esiti degli unit test eseguiti sui due backend messi a confronto.

TestAuthService

Test	Descrizione	Esito demo sicuro	Esito demo insicuro
testLogin_when_user_not_exist	Verifica il comportamento del sistema quando un utente inesistente tenta di accedere. Il test assicura che venga restituito un messaggio di errore appropriato e che non venga concesso l'accesso.	passed	passed
testLogin_when_user_exists_and_password_is_correct	Verifica che un utente esistente con credenziali corrette possa effettuare il login con successo, ricevendo una risposta positiva e ottenendo l'accesso al sistema.	passed	passed

testLogin_when_database_throws_exception	Simula un errore del database durante il processo di login e verifica che il sistema gestisca correttamente l'eccezione, restituendo un messaggio di errore adeguato senza esporre dettagli sensibili.	passed	passed
testLogin_when_password_is_incorrect	Verifica che un utente esistente, ma con una password errata, non possa accedere al sistema e che venga restituito un messaggio di errore appropriato, senza rivelare informazioni sulle credenziali.	passed	passed

TestAuthServiceDecorator

Test	Descrizione	Esito demo sicuro	Esito demo insicuro
testLogin_successful	Verifica che il decorator permetta che un utente con credenziali valide possa effettuare l'accesso con successo, ricevendo una risposta positiva.	passed	-
testLogin_failed_attempt	Simula un tentativo di accesso fallito e verifica che il decorator registri, aggiornando eventualmente il conteggio dei tentativi errati.	passed	-
testLogin_temporary_lockout	Verifica che, dopo un certo numero di tentativi di accesso errati, l'account venga temporaneamente bloccato per un periodo di	passed	-

	tempo definito, impedendo nuovi tentativi fino alla scadenza del blocco.		
testLogin_permanent_Lockout	Verifica che, dopo un numero eccessivo di tentativi falliti, l'account venga bloccato in modo permanente e che ulteriori tentativi di accesso restituiscano un messaggio di errore adeguato.	passed	-
testLogin_resets_after_success	Controlla che, dopo un login riuscito, il contatore dei tentativi falliti venga azzerato, evitando blocchi ingiustificati per futuri accessi validi.	passed	-

TestDAOUserSQL

Test	**Descrizione**	**Esito demo sicuro**	**Esito demo insicuro**
test_findByUsername_when_user_not_exist	Verifica che il metodo findByUsername restituisca un risultato nullo quando l'utente cercato non esiste nel database.	passed	passed
test_findByUsername_when_user_exist	Verifica che il metodo findByUsername restituisca correttamente i dettagli dell'utente quando il nome utente esiste nel database.	passed	passed
test_updateUser_when_user_not_exist	Verifica che il metodo updateUser gestisca correttamente il caso in cui l'utente da aggiornare non	passed	passed

	esiste, non eseguendo alcuna modifica.		
test_multiple_SQL_Injection_In_FindByUsername	Verifica se il metodo findByUsername è vulnerabile a tentativi multipli di SQL Injection, testando payload malevoli.	passed	failed
test_updateUser_when_user_exist	Verifica che il metodo updateUser aggiorni correttamente i dati di un utente esistente nel database.	passed	passed
test_sqlInjection_in_updateUser_updateUser	Verifica se il metodo updateUser è vulnerabile a SQL Injection, testando l'inserimento di input malevoli nei campi aggiornabili.	passed	failed
test_SQL_Injection_In_FindByUsername	Verifica se il metodo findByUsername è vulnerabile a SQL Injection, testando l'inserimento di stringhe malevole nel parametro username.	passed	failed
test_updateUser_with_too_long_values	Verifica il comportamento del metodo updateUser quando vengono forniti valori troppo lunghi, testando la gestione delle limitazioni sui campi del database.	passed	passed
test_updateUser_with_null_or_empty_values	Verifica che il metodo updateUser gestisca correttamente valori nulli o vuoti nei campi evitando ,obbligatori errori o aggiornamenti non validi.	passed	failed

test_multiple_SQL_Injection_In_FindByUsername failed 77 ms

2025-02-26 09:32:39 INFO [it.uni.dao.DAOUserSQL] (Test worker) ResultSet: luigi.bianchi with Password: test
Il risultato atteso è null. Al contrario l'applicazione è vulnerabile alla SQL Injectionorg.opentest4j.AssertionFailedError: Il risultato atteso è null. Al contrario l'applicazione è vulnerabile alla SQL Injection ==> expected: but was:

test_SQL_Injection_In_FindByUsername failed 47 ms

2025-02-26 09:32:39 INFO [it.uni.dao.DAOUserSQL] (Test worker) ResultSet: luigi.bianchi with Password: test
Il risultato atteso è null. Al contrario l'applicazione è vulnerabile alla SQL Injectionorg.opentest4j.AssertionFailedError: Il risultato atteso è null. Al contrario l'applicazione è vulnerabile alla SQL Injection ==> expected: but was:

test_updateUser_with_null_or_empty_values failed 41 ms

2025-02-26 09:32:39 INFO [it.uni.dao.DAOUserSQL] (Test worker) ResultSet: mario.rossi with Password: test
Il risultato atteso è una stringa non vuotaorg.opentest4j.AssertionFailedError: Il risultato atteso è una stringa non vuota ==> expected: but was:

test_sqlInjection_in_updateUser failed 71 ms

2025-02-26 09:32:39 INFO [it.uni.dao.DAOUserSQL] (Test worker) ResultSet: mario.rossi with Password: test
Il risultato atteso è la password utente inalterata. Al contrario l'applicazione è vulnerabile alla SQL Injectionorg.opentest4j.AssertionFailedError: Il risultato atteso è la password utente inalterata. Al contrario l'applicazione è vulnerabile alla SQL Injection ==> expected: but was:

TestDashboardEditService

Test	Descrizione	Esito demo sicuro	Esito demo insicuro
testEditLevel	Verifica che un amministratore possa modificare correttamente il livello e che le modifiche vengano salvate e applicate correttamente nel sistema.	passed	passed
testEditUserName	Controlla che un utente possa aggiornare il proprio nome e che la modifica venga registrata nel database correttamente.	passed	passed
testEditUserSurname	Controlla che un utente possa aggiornare il proprio cognome e che la modifica venga registrata nel database correttamente.	passed	passed
testEditUserEmail	Controlla che un utente possa aggiornare la propria mail e che la modifica venga registrata nel database correttamente.	passed	passed

testEditUserPhone	Controlla che un utente possa aggiornare il proprio numero di telefono e che la modifica venga registrata nel database correttamente.	passed	passed
testEditUserAddress	Controlla che un utente possa aggiornare il proprio indirizzo e che la modifica venga registrata nel database correttamente.	passed	passed

TestUserMapper

Test	Descrizione	Esito demo sicuro	Esito demo insicuro
test_toDTO_when_user_is_null	Verifica il comportamento del sistema quando si tenta di convertire un oggetto utente nullo in un DTO. Il test assicura che venga gestito correttamente il caso di valore nullo senza causare errori imprevisti.	passed	passed
test_fromDTO_when_DTO_is_valid	Verifica che il sistema converta correttamente un DTO valido in un oggetto utente, assicurandosi che i dati siano mappati correttamente e che l'operazione avvenga senza errori.	passed	passed
test_toDTO_when_user_is_valid	Verifica che un oggetto utente valido venga correttamente convertito in un DTO, con tutti i campi mappati in modo accurato e senza perdere informazioni.	passed	passed

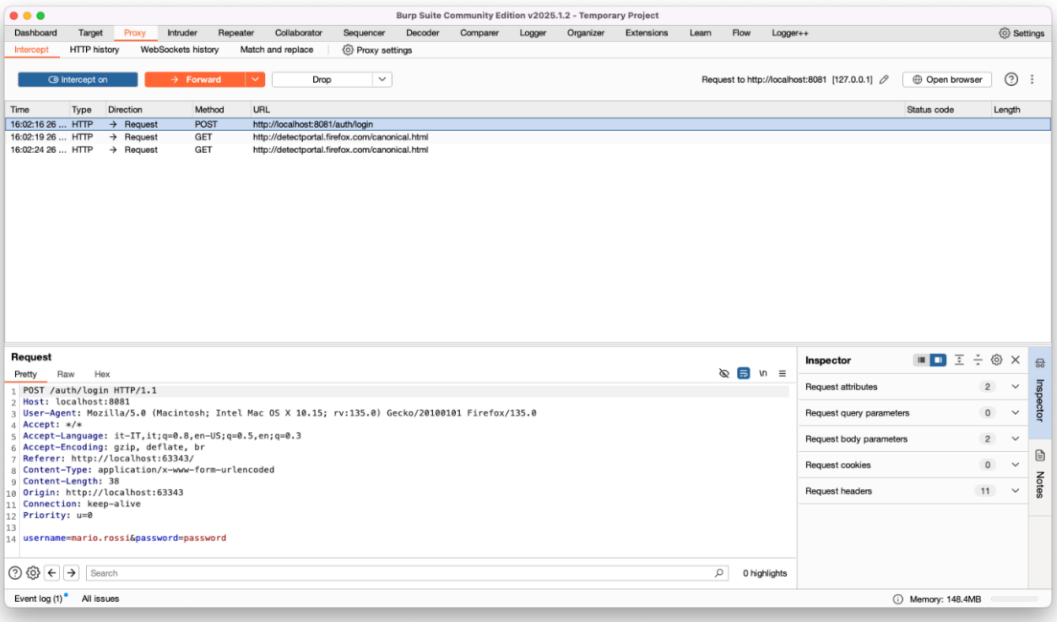
test_fromDTO_with_image	Verifica che il sistema gestisca correttamente il caso in cui un DTO contenga un'immagine, assicurando che venga convertita correttamente nel formato desiderato.	passed	passed
test_fromDTO_when_DTO_is_null	Verifica che il sistema gestisca correttamente il caso di un DTO nullo, evitando errori e garantendo una gestione appropriata della situazione senza interruzioni nel flusso del programma.	passed	passed

Penetration Test

Per l'analisi della sicurezza dell'applicazione è stata scelta Burp Suite, una piattaforma avanzata per il penetration testing di applicazioni web. Burp Suite agisce come un proxy intermedio tra il browser e il server web. Questo permette di intercettare, modificare e analizzare tutte le richieste e risposte HTTP scambiate tra il client e l'applicazione target.

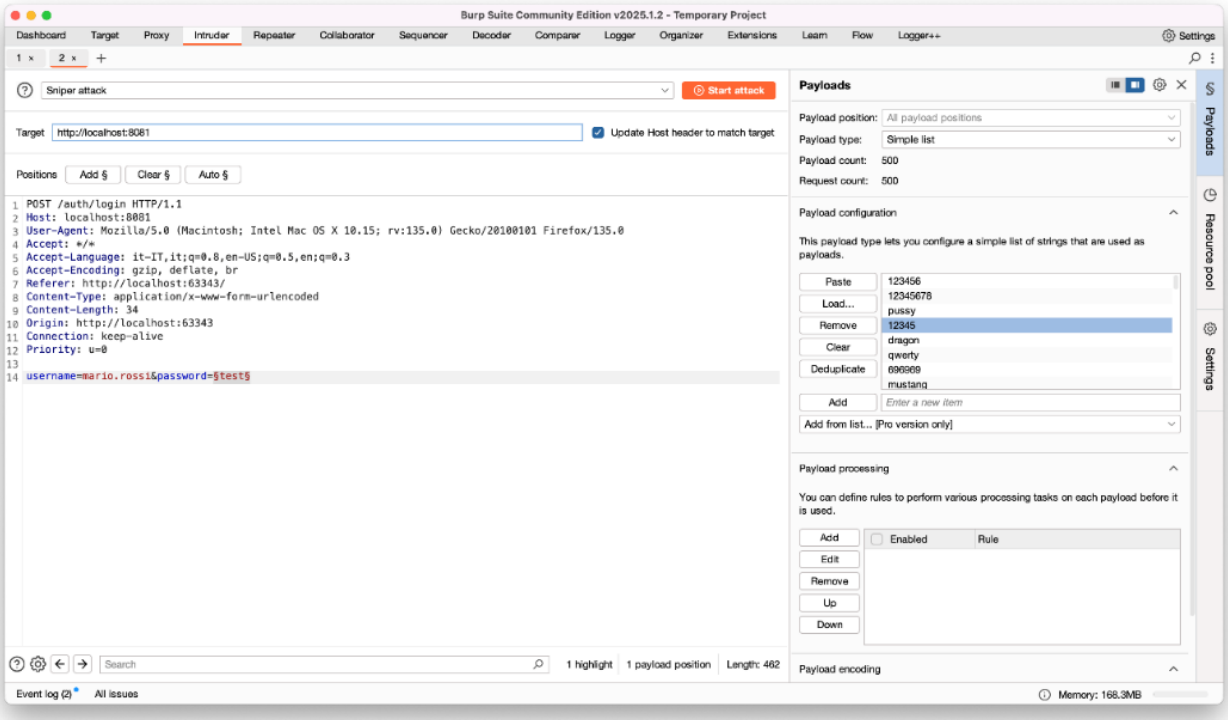
L'obiettivo principale dei test è stato quello di individuare e dimostrare la presenza di vulnerabilità, concentrandosi in particolare su tre tipologie di attacco: Broken Access Control, SQL Injection e Identification and Authentication Failures. I test sono stati condotti sia manualmente che in modo automatizzato, sfruttando le funzionalità offerte dallo strumento

Prima di procedere con i test, è stato necessario configurare Burp Suite e il browser per intercettare il traffico dell'applicazione. Ciò è stato ottenuto impostando un proxy locale, che ha permesso di catturare e analizzare ogni richiesta HTTP effettuata dall'applicazione. Grazie a questa configurazione, è stato possibile ricostruire la mappa del sito (site map), ottenendo una panoramica delle richieste inviate dal client al server. L'analisi delle chiamate eseguite durante un normale flusso utente ha permesso di identificare i punti potenzialmente vulnerabili.



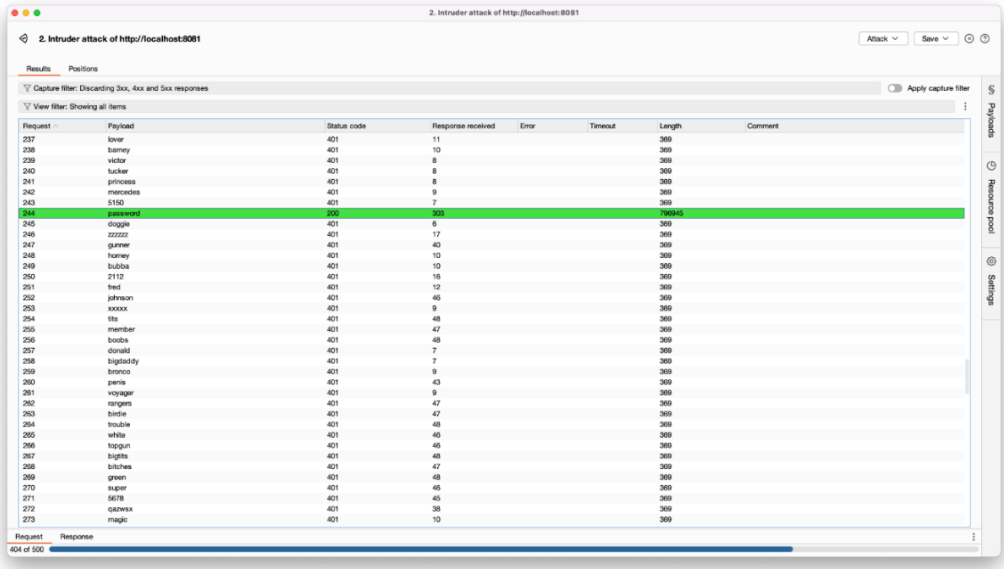
Uno dei primi test eseguiti ha riguardato la verifica della robustezza del meccanismo di autenticazione. La chiamata POST relativa al login è stata analizzata in dettaglio e si è rivelata suscettibile ad attacchi di brute force, poiché non presentava limitazioni sul numero di tentativi falliti.

Per testare questa vulnerabilità, è stata utilizzata la funzionalità Intruder di Burp Suite, che ha permesso di inviare automaticamente centinaia di tentativi di login con credenziali diverse. L'attacco è stato effettuato utilizzando un dizionario di 500 password comunemente utilizzate, simulando un attacco realistico da parte di un attaccante.

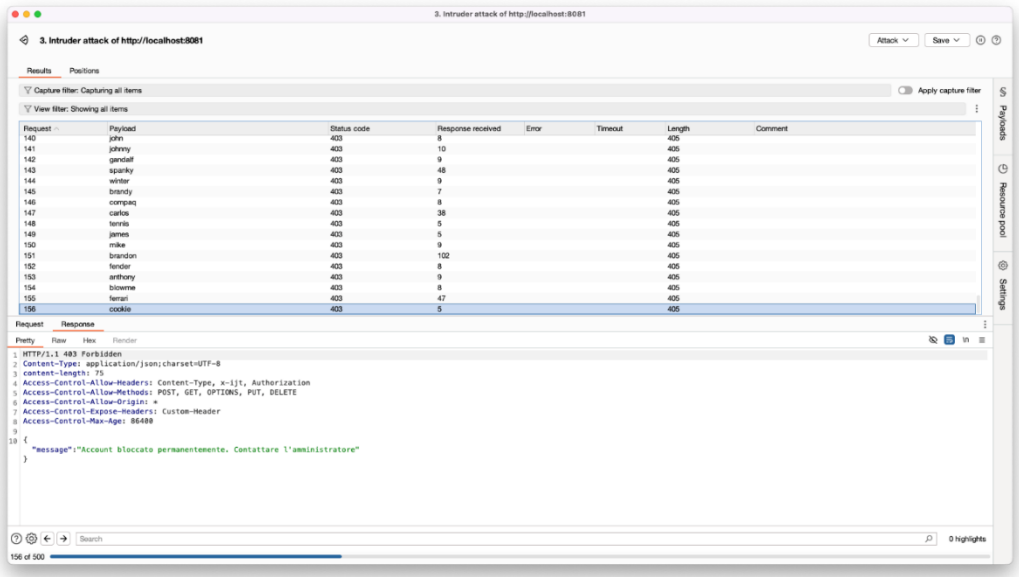


I risultati hanno mostrato una netta differenza tra i due approcci implementati:

- Nel progetto senza design pattern, l'attacco è stato eseguito con successo. Dopo un numero sufficiente di tentativi, una delle password nel dizionario ha permesso l'accesso alla dashboard aziendale, confermando l'assenza di protezioni efficaci contro attacchi di tipo brute force.

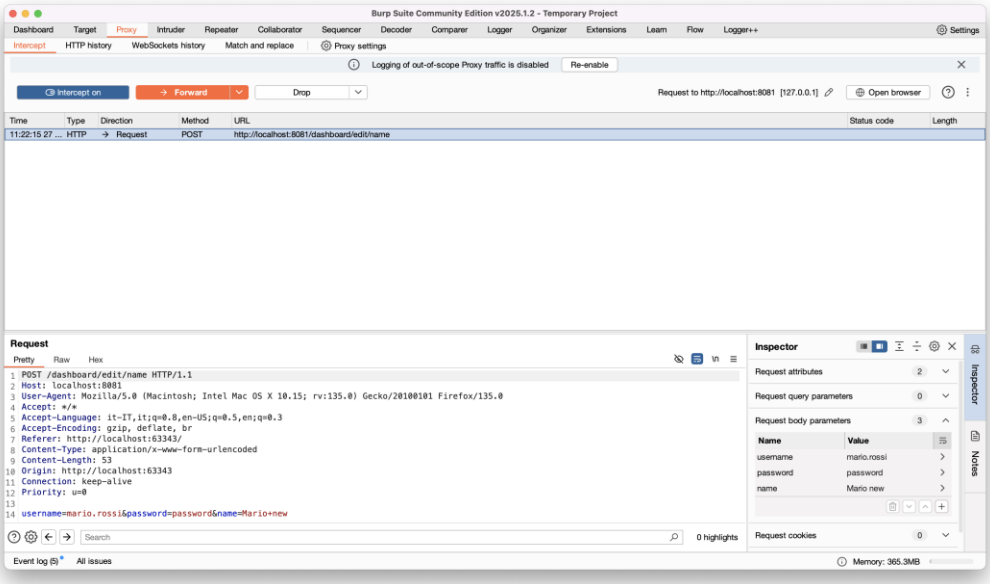


- Nel progetto con design pattern, invece, il meccanismo di protezione implementato ha bloccato l'utenza dopo un numero limitato di tentativi falliti. Grazie all'utilizzo del decorator, è stato introdotto un sistema di account lockout, che impedisce nuovi tentativi di autenticazione dopo un certo numero di errori, riducendo così l'efficacia di un attacco brute force.



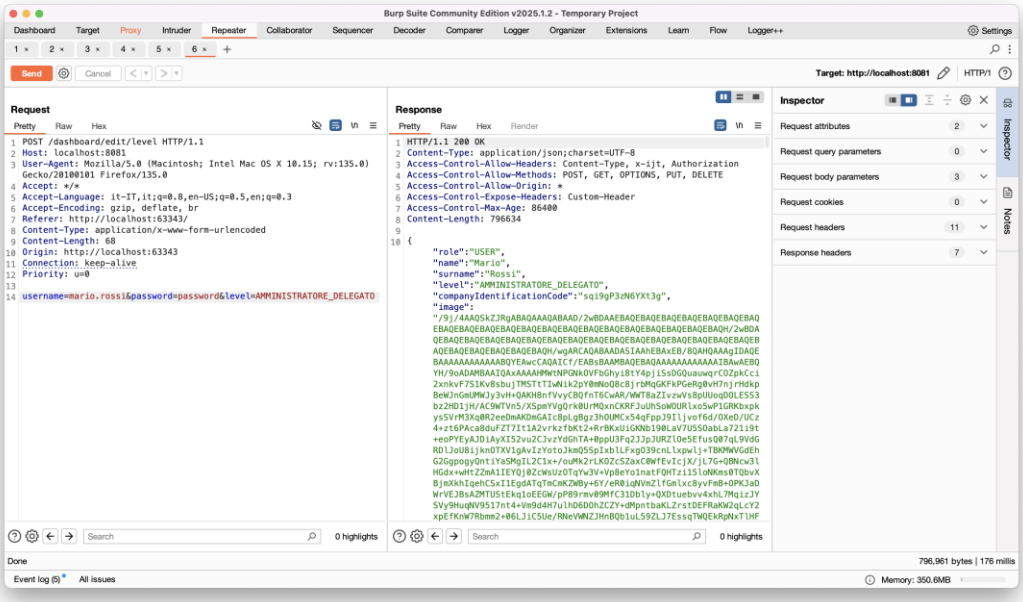
Per testare la vulnerabilità Broken Access Control, è stata utilizzata la funzionalità intercept di Burp Suite, che ha permesso di intercettare e modificare la richiesta di aggiornamento dati inviata da un utente con livello "user". In particolare, è stato alterato il parametro relativo al livello

di accesso nel body della richiesta, per il quale ha i permessi di modifica solamente un utente con livello “admin”. L'immagine seguente mostra la richiesta intercettata.



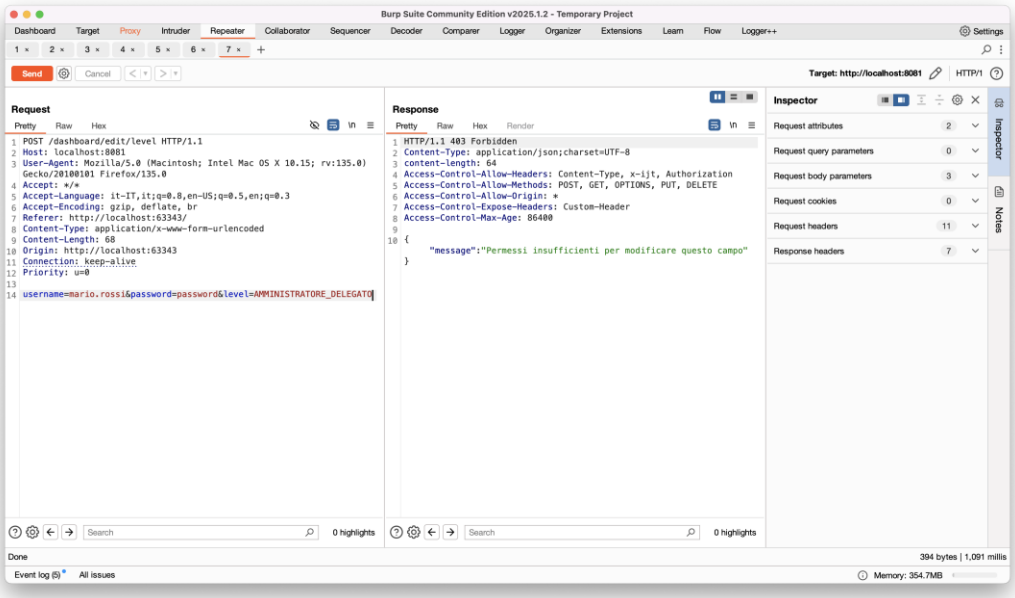
I risultati ottenuti evidenziano quanto segue:

- Nel progetto senza design pattern, l'attacco è stato eseguito con successo. La richiesta modificata è stata accettata dal server, restituendo un codice “200 OK” e aggiornando il livello di accesso dell'utente, nonostante questi non avesse i permessi necessari.



- Nel progetto con design pattern, invece, il meccanismo di protezione implementato ha impedito la modifica, restituendo un

messaggio di errore appropriato e mantenendo inalterati i privilegi dell'utente.

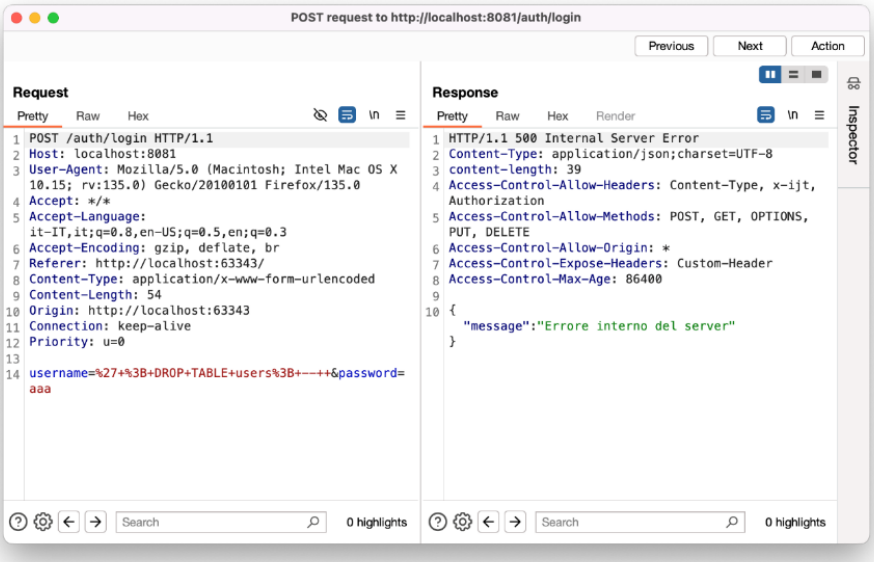


Un altro aspetto critico analizzato è stato il rischio di SQL Injection, poiché ritenuta una delle vulnerabilità più pericolose secondo la OWASP Top 10. Il test si è concentrato in particolare su due funzionalità dell'applicazione:

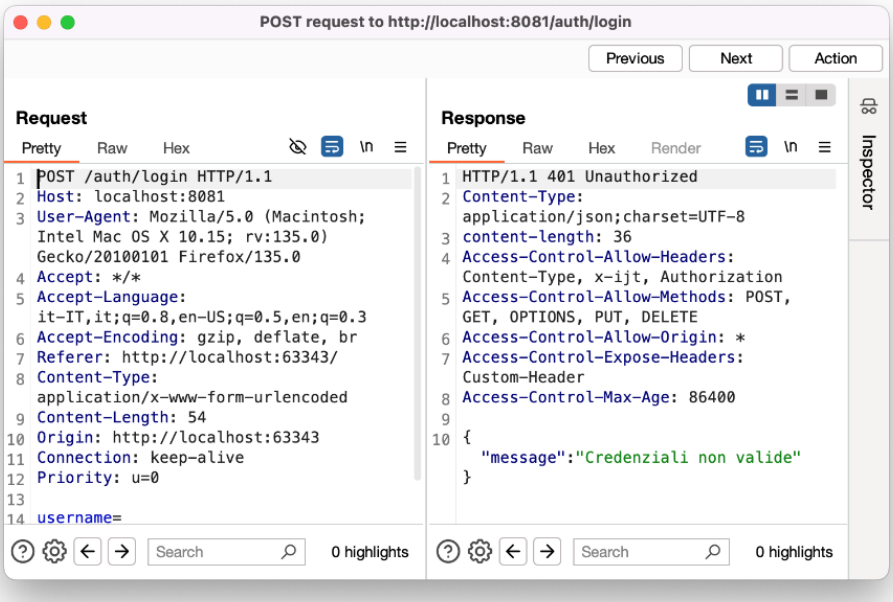
- Il form di login, che prende in input username e password e li invia al backend.
- La funzionalità di modifica dati utente, che permette agli utenti di aggiornare informazioni personali.

Nel progetto senza design pattern, entrambe le funzionalità sono risultate vulnerabili.

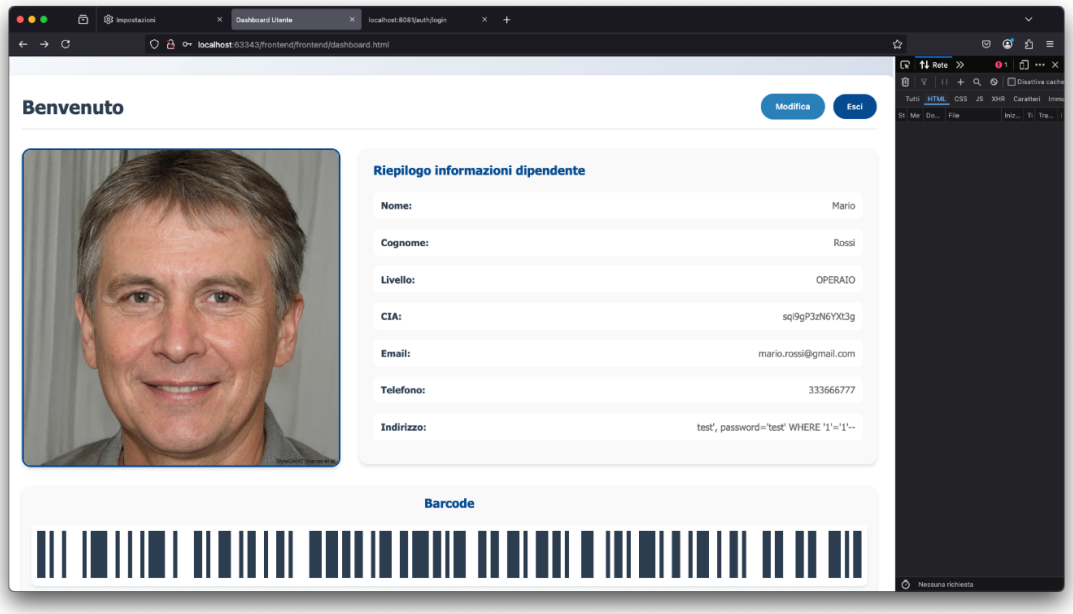
Ad esempio, inserendo la seguente stringa SQL malevola nel campo username del form di login utente: ' ; DROP TABLE users; -- il backend ha restituito errore 500 e l'intera tabella users del database è stata eliminata.



Nel progetto con design pattern, invece, ha gestito correttamente l'input utente come una semplice stringa e non come un comando.

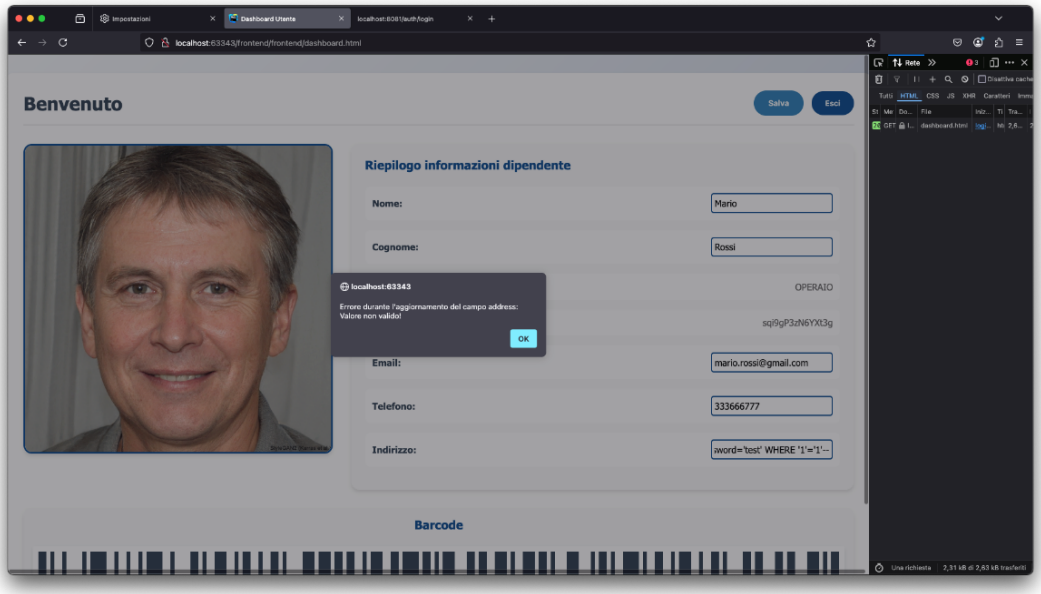


Per quanto riguarda la modifica dei dati, inserendo la seguente stringa SQL malevola nel campo indirizzo del form di modifica utente: "test', password='test' WHERE '1'='1'--", il backend ha interpretato il codice come parte della query SQL, portando alla modifica delle password di tutti gli utenti registrati nel database.



Questo è stato possibile a causa della concatenazione diretta dei parametri utente all'interno della query SQL, senza alcuna validazione o utilizzo di prepared statements. L'assenza di misure di protezione ha permesso di manipolare le query ed eseguire comandi arbitrari sul database.

Nel progetto con design pattern, invece, la vulnerabilità è stata mitigata grazie all'uso di prepared statements, che impediscono l'iniezione di codice SQL malevolo. L'applicazione ha restituito un messaggio di errore coerente e ha impedito la modifica non autorizzata dei dati.



Analisi dei Risultati

L'analisi dei test eseguiti ha evidenziato differenze notevoli tra il backend sviluppato senza l'uso di design pattern e quello che invece implementa pattern progettuali mirati a migliorare la sicurezza. Sebbene gli unit test abbiano confermato il corretto funzionamento delle principali funzionalità dell'applicazione, mostrano al contempo le vulnerabilità presenti nel progetto insicuro.

Per quanto riguarda il servizio di autenticazione TestAuthService, entrambi i backend hanno superato i test relativi alla gestione degli accessi per utenti inesistenti, credenziali corrette e gestione delle eccezioni a livello di database. Tuttavia, nel progetto sicuro, il decorator ha introdotto protezioni aggiuntive come il blocco temporaneo e permanente degli account dopo un numero predefinito di tentativi falliti, evitando così attacchi di brute force, vulnerabilità invece presente nel progetto senza pattern.

I test eseguiti sul livello di accesso ai dati TestDAOUserSQL hanno mostrato la vulnerabilità del progetto insicuro alle SQL Injection. In particolare, il metodo findByUsername e il metodo updateUser risultano suscettibili a manipolazioni malevole, consentendo di eseguire query arbitrarie. Questo è stato confermato dagli unit test che hanno fallito in presenza di input contenenti payload malevoli. Nel progetto sicuro, invece, l'uso di prepared statements ha impedito tali attacchi, garantendo che gli input venissero trattati come semplici stringhe e non come parte della query SQL.

Per quanto riguarda la gestione delle modifiche utente TestDashboardEditService, entrambi i backend hanno superato i test relativi all'aggiornamento dei dati personali. Tuttavia, il progetto insicuro non ha gestito correttamente valori nulli o troppo lunghi in alcuni campi, portando a errori di database o aggiornamenti errati. Il progetto sicuro ha invece implementato controlli adeguati, impedendo operazioni anomale.

I penetration test, condotti con Burp Suite, hanno ulteriormente confermato le vulnerabilità riscontrate nei test unitari. Il progetto insicuro

è risultato vulnerabile a tentativi di brute force, mentre il progetto sicuro ha bloccato gli accessi dopo un numero limitato di tentativi falliti. Anche le vulnerabilità a SQL Injection sono state verificate, dimostrando che nel progetto insicuro era possibile manipolare il database tramite input malevoli, mentre nel progetto sicuro tali tentativi sono stati respinti grazie all'uso di query parametrizzate.

Infine, il test di sniffing della richiesta di aggiornamento utente, condotto sempre con Burp Suite, ha messo in evidenza una grave vulnerabilità nel progetto insicuro legata al Broken Access Control. Intercettando e modificando la richiesta HTTP di aggiornamento dati, un utente senza privilegi amministrativi è stato in grado di **modificare il** proprio livello di accesso, violando i controlli di autorizzazione. Nel progetto sicuro, invece, i controlli implementati hanno impedito la modifica non autorizzata, restituendo un messaggio di errore appropriato. Questo test ha confermato l'importanza di una corretta gestione dei permessi e della validazione lato server per prevenire escalation di privilegi non autorizzate.

Linee Guida per l'Applicazione dei Design Pattern

Di seguito vengono presentate le linee guida per applicarle in maniera efficace i pattern applicati nel progetto, evidenziando come ciascuno contribuisca a mitigare specifiche vulnerabilità.

Proxy Pattern (Protection Proxy)

É indicato utilizzare il proxy pattern per controllare l'accesso alle risorse sensibili. In particolare, il protection proxy si interpone tra il client e l'oggetto reale, intercettando le chiamate ai metodi per verificare i privilegi dell'utente prima di inoltrare la richiesta, per mitigare le vulnerabilità legate al Broken Access Control.
Linee Guida:

- Identificare le operazioni e le risorse critiche che necessitano di controlli di accesso;
- Implementare il proxy come un intermediario trasparente, che non alteri la logica di business se l'utente è autorizzato;
- Centralizzare le verifiche di autorizzazione, garantendo che ogni richiesta venga sottoposta a validazione.

DAO Pattern con Prepared Statements

Il DAO pattern isola la logica di accesso ai dati dal resto dell'applicazione, facilitando la manutenzione e aumentando la sicurezza. L'impiego dei prepared statements all'interno del DAO garantisce che i parametri vengano trattati in modo sicuro, prevenendo attacchi di SQL Injection.
Linee Guida:

- Centralizzare tutte le operazioni di persistenza dei dati in componenti dedicati (DAO), separando la logica di accesso dai processi di business;
- Utilizzare query parametrizzate per garantire che gli input siano gestiti come dati e non possano alterare la struttura delle query SQL;
- Effettuare controlli e validazioni sugli input prima di passarli ai DAO, rafforzando ulteriormente la sicurezza.

Decorator Pattern

Il decorator offre un approccio flessibile per estendere dinamicamente le funzionalità di un componente, in questo caso il processo di autenticazione, senza modificare il suo codice sorgente. In ambito sicurezza, l'obiettivo è integrare controlli come il rate limiting e i meccanismi di lockout per mitigare le vulnerabilità legate alle Identification and Authentication Failures.

Linee Guida:

- Creare decorator specifici per la sicurezza: sviluppare classi decorator che implementano la stessa interfaccia, avvolgendo il componente di base e aggiungendo funzionalità di sicurezza. Ad esempio un decorator per il lockout che blocca l'utenza dopo un numero predefinito di login falliti;
- Progettare i decorator in modo che possano essere applicati in cascata, permettendo di combinare più controlli di sicurezza senza modificare la logica di base dell'autenticazione. Questo favorisce un'architettura flessibile e a basso accoppiamento.

Observer Pattern

L' observer è impiegato per implementare un sistema di monitoraggio in tempo reale per eventi di sicurezza, come accessi non autorizzati.

Linee Guida:

- Definire un soggetto che rileva gli eventi di sicurezza e mantenga una lista di osservatori (come sistemi di logging e monitoraggio);
- Assicurarsi che ogni evento rilevante venga notificato immediatamente a tutti i componenti interessati, garantendo una tracciabilità completa.
- Progettare il sistema in modo che l'aggiunta di nuovi osservatori sia semplice e non implichi modifiche al codice del soggetto, favorendo un'architettura scalabile e con un basso accoppiamento.

Conclusioni

L'analisi condotta ha evidenziato come l'uso dei design pattern possa migliorare significativamente la sicurezza di un'applicazione web, mitigando vulnerabilità come Broken Access Control, SQL Injection, Identification and Authentication Failures e Security Logging and Monitoring Failures . Il confronto tra il progetto senza design pattern e quello con un'implementazione sicura ha dimostrato che l'adozione di pattern strutturati permette di prevenire attacchi e ridurre le superfici di rischio.

Tuttavia, è importante sottolineare che i pattern da soli non garantiscono la sicurezza. Devono essere integrati all'interno di un approccio più ampio di security by design e defense in depth, che preveda una gestione attenta delle credenziali, logging sicuro, monitoraggio continuo e aggiornamenti costanti. Solo attraverso un'architettura ben progettata e una strategia di sicurezza completa è possibile garantire un sistema robusto e resiliente agli attacchi.

In sintesi, l'utilizzo dei design pattern rappresenta una best practice fondamentale nello sviluppo software sicuro, ma deve essere affiancato da una consapevolezza continua sulle minacce emergenti e da una strategia di protezione che combini più livelli di difesa.

Bibliografia

I. OWASP. “OWASP Top Ten: 2021”. https://owasp.org/Top10/
II. AGID. “Linee guida per lo sviluppo del software sicuro”. https://www.agid.gov.it/it/sicurezza/cert-pa/linee-guida-sviluppo-del-software-sicuro
III. Microsoft. “Microsoft Security Development Lifecycle (SDL)”. https://learn.microsoft.com/it-it/compliance/assurance/assurance-microsoft-security-development-lifecycle
IV. Port Swigger. “Authentication vulnerabilities”. https://portswigger.net/web-security/authentication
V. Port Swigger. “SQL injection”. https://portswigger.net/web-security/sql-injection#what-is-sql-injection-sqli