HEIG-VD — SLH

Laboratoire 2 – Rapport

Loïc Herman

13 décembre 2024

1 Questions

> Les passkeys, comme l'authentification unique (SSO), permettent toutes deux à un utilisateur d'utiliser un seul mécanisme de sécurité pour tous ses services, sans s'exposer à une attaque de « credentials stuffing ». Mais quelle est la différence en termes de risque ?

Dans le cas du SSO, les informations d'authentification sont centralisées sur les serveurs du fournisseur d'identité. Ce modèle présente plusieurs implications sécuritaires :

Tout d'abord, cette centralisation crée un point unique de défaillance. Si les serveurs du fournisseur SSO sont compromis, tous les services connectés deviennent potentiellement vulnérables. En outre, les fournisseurs SSO deviennent des cibles très attractives pour les attaquants en raison de la concentration de données sensibles.

En revanche, les passkeys adoptent une approche décentralisée du stockage. Les clés privées sont stockées localement sur les appareils des utilisateurs. Cette architecture présente un profil de risque différent :

Le stockage décentralisé signifie qu'il n'existe pas de point unique de défaillance global. Un attaquant devrait compromettre chaque appareil individuellement pour obtenir accès aux passkeys. Par exemple, même si un attaquant parvient à compromettre un appareil contenant une passkey, les autres appareils et services de l'utilisateur restent sécurisés.

Cependant, cette décentralisation introduit ses propres défis. La perte ou le vol d'un appareil peut potentiellement compromettre l'accès aux services si l'utilisateur n'a pas configuré de dispositif de récupération. Il est crucial de noter que la sécurité globale du système est déterminée par son maillon le plus faible. Dans le contexte de ce laboratoire, où la récupération des passkeys repose sur un email envoyé à l'utilisateur, la sécurité effective du système ne peut pas dépasser celle de la boîte email. Ainsi, même si les passkeys offrent une forte protection cryptographique, un attaquant qui parviendrait à compromettre l'email de l'utilisateur pourrait commencer une procédure de récupération et contourner complètement les protections offertes par les passkeys.

En conclusion, bien que les deux approches présentent des vulnérabilités potentielles, leurs modèles de stockage créent des profils de risque distincts. Le SSO concentre le risque sur un point unique mais bénéficie généralement d'une sécurité professionnelle, tandis que les passkeys distribuent le risque mais dépendent de la sécurité des appareils individuels. Le choix entre les deux dépendra donc du contexte d'utilisation et des menaces spécifiques auxquelles l'organisation fait face.

> Concernant la validation d'entrées pour les images, quelles sont les étapes que vous avez effectuées ? Et comment stockez vous les images ?

Dans notre implémentation, nous avons mis en place un processus de validation des images en plusieurs étapes, organisées selon un principe de « fail fast » pour optimiser les performances. Cette approche nous permet de rejeter rapidement les fichiers non conformes sans effectuer d'opérations coûteuses inutilement.

La première validation est effectuée automatiquement par le framework Axum, qui impose une limite de taille de 2 Mo sur les fichiers mis en ligne. Cette restriction intégrée nous dispense d'implémenter notre propre vérification de taille, et évitera les problèmes de stockage mémoire quand le contenu de la requête est stocké dans un tableau de bytes.

Ensuite, nous procédons à une vérification de l'extension du fichier. Nous n'acceptons que les extensions `.jpg` et `.jpeg`, rejetant immédiatement tout fichier ne correspondant pas à ces critères. L'extension est récupérée avec l'aide de la librairie standard Rust permettant d'éviter des erreurs de parsing en garantissant une certaine robustesse.

La troisième étape consiste à vérifier le « magic header » du fichier à l'aide de la crate `image`. Cette vérification est cruciale, car elle permet de détecter les tentatives de déguisement de fichiers malveillants par simple renommage de l'extension. Si le magic header ne correspond pas à celui d'un fichier JPEG, le fichier est rejeté à ce stade.

La dernière étape de validation, la plus intensive en ressources, consiste à tenter de parser l'intégralité du fichier en tant qu'image JPEG. Cette opération permet de détecter d'éventuelles corruptions ou malformations dans le fichier qui n'auraient pas été identifiées lors des étapes précédentes. Du fait de son coût en termes de performances, cette vérification n'est effectuée qu'en dernier recours, une fois que toutes les autres validations ont réussi.

Pour le stockage, nous avons adopté une approche sécurisée basée sur l'indirect object reference. Plutôt que de conserver le nom original du fichier, qui pourrait contenir des caractères problématiques ou être utilisé pour des attaques de type path traversal, nous générons un UUID unique pour chaque image. Ce UUID sert d'identifiant pour le fichier stocké et est enregistré dans la base de données en association avec le post correspondant.

> Que pensez-vous de la politique de choix des noms de fichiers uploadés ? Y voyez-vous une vulnérabilité ? Si oui, suggérez une correction.

Effectivement, la politique actuelle de gestion des noms de fichiers présente une vulnérabilité significative de type path traversal. Le code utilise directement le nom de fichier fourni dans la requête multipart sans validation appropriée. Bien que dans un usage normal via un navigateur web ce champ soit rempli automatiquement, un attaquant peut facilement modifier la requête pour exploiter cette vulnérabilité.

En utilisant des séquences « `../` » dans le nom du fichier, un attaquant pourrait potentiellement accéder et écraser n'importe quel fichier sur le système auquel l'utilisateur du serveur a accès. Cette situation est particulièrement dangereuse, car elle pourrait compromettre l'intégrité du système.

Comme mentionné ci-avant, nous avons donc utilisé le principe d'indirect object reference pour stocker les fichiers. Un UUID est généré pour remplacer le nom et l'extension est définie en dur selon le format qui est attendu. Le contenu du fichier doit quand même être validé sémantiquement pour s'assurer qu'il n'y ait pas d'incohérence entre l'extension employée et le contenu binaire du fichier stocké.

```rust
let filename = format!("{}.jpg", Uuid::new_v4());
```