Sécuriser un projet web – Étude de cas : Picard

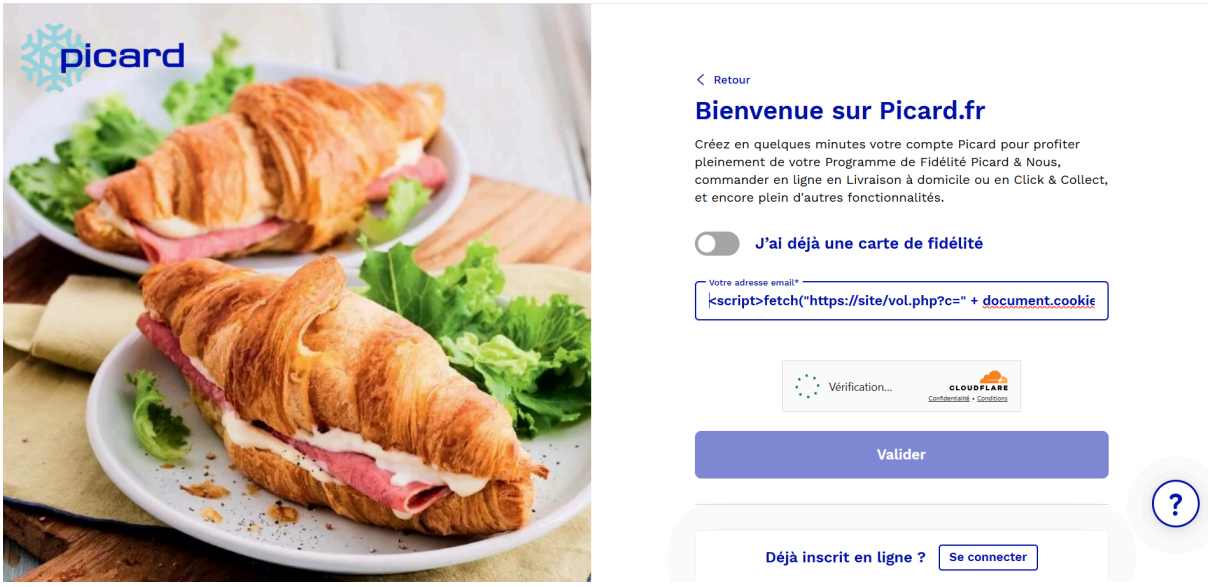
La sécurité web est un enjeu stratégique pour Picard, acteur majeur de la vente de produits surgelés en ligne. Son site, https://www.picard.fr, permet aux utilisateurs de créer un compte, d'ajouter des produits à leur panier, de passer commande et de recevoir des newsletters. Dans ce contexte, trois types d'attaques web peuvent être envisagés. Même si elles ne sont pas nécessairement actives ou exploitables aujourd'hui, elles illustrent des scénarios réalistes de vulnérabilités possibles. Chaque attaque est décrite selon sa méthode, son point d'entrée sur le site, et les contre-mesures à mettre en œuvre.

1ère Attaque **- Injection XSS sur le formulaire d'inscription ou de contact**

Sur le site Picard, plusieurs formulaires sont proposés aux utilisateurs : la création de compte, l'abonnement à la newsletter, ou encore le formulaire de contact. Tous ces formulaires permettent à l'utilisateur de saisir des données comme son prénom, son nom, un message ou une adresse email dans des champs textuels.

Ces champs, précisément, peuvent être utilisés à des fins malveillantes ; ils peuvent être exploités afin d'injecter du code. Si le site affiche ensuite ce champ quelque part (par exemple dans l'espace "Mon compte", ou dans un email de confirmation) sans nettoyer correctement les données, alors ce script s'exécutera dans le navigateur d'un administrateur ou d'un autre utilisateur.

Prenons un exemple concret : si une personne malveillante inscrit dans le champ mail le code suivant.



Imaginons maintenant que ce mail est ensuite affiché tel quel dans le profil ou dans une notification visible côté utilisateur, le script sera automatiquement exécuté par le navigateur. Cela signifie que le script ne sera pas perçu comme du texte, mais bien comme une commande JavaScript. Dans l'exemple ci-dessus, ce script envoie les cookies de la victime à un site externe, ce qui permettrait de voler une session utilisateur et de se connecter à son compte à distance. On parle ici d'une attaque XSS (Cross-Site Scripting), très répandue sur le web, et redoutable si elle n'est pas anticipée.

Pourquoi cette attaque fonctionne-t-elle ?

Cette attaque repose sur le fait que les navigateurs interprètent le HTML : tout ce qui est contenu dans une balise <script> est exécuté comme du code. Si un champ rempli par un utilisateur est

ensuite affiché dans une page sans être traité, il est possible que le navigateur interprète le contenu comme une vraie commande.

Cela se produit lorsque le développeur réinjecte directement les données saisies dans le DOM avec des fonctions comme innerHTML, ou dans un email HTML généré dynamiquement, sans aucune sanitisation ou échappement du contenu.

Autrement dit, l'application fait confiance aux données entrées par l'utilisateur et les affiche sans filtre, ce qui ouvre la porte à l'injection de code.

Comment s'en protéger efficacement ?

Pour empêcher les attaques XSS, il faut changer complètement la façon dont le site gère l'affichage des données utilisateur. La première règle fondamentale est de ne jamais faire confiance à ce que l'utilisateur entre, même si cela paraît anodin comme un prénom ou un message.

La méthode la plus robuste est d'échapper systématiquement les caractères spéciaux avant d'afficher une donnée. Cela signifie que si l'utilisateur écrit <script>, le navigateur n'interprétera pas cela comme une balise, mais affichera le texte brut <script>. Cela suffit à bloquer l'exécution du script.

Pour cela, il est fortement conseillé d'utiliser des bibliothèques spécialisées comme DOMPurify en JavaScript, ou des fonctions d'échappement côté serveur (PHP, Node, Python, etc.). En front-end, il faut aussi éviter les méthodes dangereuses comme innerHTML, qui insèrent directement du code dans la page. Il est préférable d'utiliser textContent ou innerText, qui se contentent d'afficher du texte sans l'interpréter comme du code HTML.

Mais la sécurisation ne s'arrête pas là. Il est également recommandé de renforcer les politiques de sécurité dans les headers HTTP. Par exemple, en définissant une Content-Security-Policy, on peut indiquer au navigateur de bloquer l'exécution de tout script non autorisé. De même, d'autres headers comme X-Content-Type-Options ou X-Frame-Options ajoutent une couche de protection contre les interprétations erronées ou le clickjacking.

Enfin, il faut sécuriser les cookies contenant les informations de session. Si jamais un script parvenait malgré tout à s'exécuter, il ne devrait pas pouvoir accéder à ces cookies. Pour cela, ils doivent être marqués comme :

- HttpOnly : le cookie ne peut pas être lu par du JavaScript
- Secure : transmis uniquement via HTTPS
- SameSite=Strict : empêchant leur envoi en cas de requête externe

---

2ème Attaque **- Webhook non sécurisé simulé sur le système de commandes**

Chaque commande déclenche une série d'actions automatisées : confirmation de commande, mise à jour des stocks, création de facture ou déclenchement de l'expédition. Ces actions passent souvent par des webhooks, c'est-à-dire des points d'entrée exposés sur le site, qui attendent des notifications venant d'un service externe.

Par exemple, lorsqu'une commande est validée sur Picard, un service partenaire pourrait envoyer une requête HTTP vers une URL dédiée de type : https://www.picard.fr/api/webhook/order-received

Cette URL est censée recevoir des informations fiables (comme un identifiant de commande, la liste des produits, la quantité, etc.). Le problème, c'est que si cette adresse est connue ou devinée, et qu'aucune vérification d'authenticité n'est mise en place, un attaquant peut simuler ces appels sans jamais passer par l'interface du site.

Pourquoi cette attaque fonctionne-t-elle ?

Cette attaque repose sur un principe simple : un webhook est une URL publique.
Par défaut, le serveur ne sait pas qui envoie la requête. Il ne fait que recevoir des données et les traiter. Sans vérification, n'importe qui peut imiter un service légitime et envoyer une fausse commande.

Un attaquant peut, par exemple, utiliser un outil comme Postman pour envoyer une requête POST vers une URL qu'il pense être un webhook de traitement de commande, en construisant un corps json qui ressemble à une vraie commande :

```
{
  "commande_id": "1234",
  "produits": [
    { "ref": "kiwi", "quantite": 99 }
  ]
}
```

Si le serveur accepte cette requête, alors le stock peut être mis à jour, des actions internes peuvent être déclenchées, et des données fausses peuvent polluer la base (factures fantômes, rapports faussés..).

Ce type d'attaque est d'autant plus dangereux qu'il est silencieux : l'attaquant n'a pas besoin d'avoir un compte, de se connecter ou de compromettre l'interface. Il s'attaque directement à la couche serveur.

Comment s'en protéger efficacement ?

La seule manière fiable de protéger un webhook est de vérifier que les requêtes reçues viennent bien de la source attendue. Cela passe par la mise en place d'un système de signature cryptographique. Dans la majorité des cas, on utilise une signature HMAC-SHA256 basée sur un secret partagé entre le service émetteur et le serveur Picard. Cette vérification doit être réalisée avant tout traitement, y compris avant de parser le JSON, afin d'éviter tout comportement anormal en cas d'entrée malformée.

En complément, il est conseillé d'ajouter un timestamp dans chaque requête, afin de rejeter les appels trop anciens. Cela empêche les attaques par relecture (replay attacks), où un attaquant rejouerait une requête interceptée.

Il est aussi important de :

- Comparer les signatures avec une méthode résistante au timing (timingSafeEqual)
- Ne jamais exposer ou documenter publiquement l'URL du webhook
- Journaliser tous les appels entrants, même ceux rejetés, pour surveiller les tentatives malveillantes

La défense doit être globale : même un webhook bien signé ne doit pas avoir un impact critique s'il est abusé. Le système métier doit vérifier la cohérence des données reçues (commande connue, client légitime, etc.) avant de déclencher des actions irréversibles.

3ème Attaque **- Brute force sur la page de connexion**

Nous allons maintenant nous intéresser à la page de connexion (https://www.picard.fr/mon-compte-picard). Celle-ci présente un formulaire classique : un champ pour l'adresse e-mail, un champ pour le mot de passe, et un bouton de connexion. C'est une interface courante, mais aussi l'une des plus ciblées.

Lorsqu'un utilisateur entre ses identifiants, le site vérifie que ceux-ci correspondent à un compte existant et, si c'est le cas, crée une session pour lui donner accès à son espace personnel.

Ce mécanisme est simple mais vulnérable, notamment face à des attaques par bruteforce, consistant à tester des milliers de combinaisons d'e-mails et de mots de passe jusqu'à trouver une correspondance valide.

Pourquoi cette attaque fonctionne-t-elle ?

L'attaque repose sur le fait que la page de connexion permet à n'importe qui de tenter sa chance. Si aucun système ne limite le nombre de tentatives, un attaquant peut utiliser un script automatisé pour générer ou récupérer des listes d'adresses e-mail et tester pour chacune d'elles des mots de passe courants. Ce type d'attaque est souvent mené par des bots.

Le problème ne vient donc pas du système en lui-même du formulaire, mais du manque de contrôle sur un usage intensif de ce formulaire. Dans un scénario simple, un attaquant pourrait entrer une adresse e-mail connue (ex. : contact@picard.fr), tester 5000 mots de passe à la suite et finir par trouver une combinaison valide, accédant au compte de la victime.
Plus le site a d'utilisateurs, plus la probabilité de succès de ce genre d'attaque augmente.

Comment s'en protéger efficacement ?

La première étape est d'empêcher l'automatisation massive des tentatives de connexion. Cela ne passe pas par une mesure unique, mais par un ensemble de mécanismes complémentaires.

- **Limiter le nombre de tentatives (Rate limiting)**

Le serveur doit être capable de détecter les comportements anormaux, comme 10 tentatives échouées en moins d'une minute depuis la même adresse IP.
Dans ce cas, l'IP doit être temporairement bloquée (ou ralentie), et la tentative signalée.
Cela peut être fait via un middleware côté backend (express-rate-limit en Node.js, par exemple), ou par un pare-feu applicatif comme Cloudflare.

- **Bloquer temporairement un compte après X échecs**

Pour éviter que les bots ne ciblent une adresse e-mail précise, il est important de désactiver temporairement les connexions sur un compte après un certain nombre d'échecs successifs, même si ceux-ci viennent d'IP différentes. Cela empêche les attaques réparties (distributed brute force).

- **Ajouter une vérification humaine (Captcha)**

Après 2 ou 3 tentatives infructueuses, un captcha (comme reCAPTCHA de Google) doit apparaître, obligeant l'utilisateur à prouver qu'il n'est pas un robot. Cela ralentit considérablement les attaques automatiques.

- **Utiliser un système de hash sécurisé pour les mots de passe**

Les mots de passe doivent être hachés avec un algorithme robuste comme bcrypt ou argon2, avec un salt unique par utilisateur. Cela garantit que, même si la base de données est compromise, les mots de passe ne peuvent pas être récupérés en clair.
Il est aussi essentiel de ne jamais afficher de message d'erreur précis (ex. : "email incorrect" ou "mot de passe incorrect") afin de ne pas aider l'attaquant à valider l'existence d'un compte.

- **Notifier l'utilisateur en cas de comportement suspect**

Un e-mail automatique peut être envoyé lorsqu'une connexion a lieu depuis un appareil inconnu ou lorsqu'un certain nombre de tentatives ont échoué.
Cela permet à l'utilisateur de réagir rapidement en cas de tentative réelle d'intrusion.

- **Surveiller et journaliser toutes les tentatives**

Une bonne politique de sécurité impose de logguer toutes les tentatives de connexion, qu'elles soient réussies ou non, afin de détecter les schémas inhabituels (ex. : 300 tentatives en une heure depuis un même pays).

Toutes ces mesures visent à décourager les attaquants par des mécanismes visibles et invisibles qui ralentissent la progression, apportant risques et difficulté.

L'attaque n'est donc pas impossible, mais ces mesures en augmentent considérablement le coût et la complexité.