RECOMMANDATIONS POUR LA MISE EN ŒUVRE D'UN SITE WEB : MAÎTRISER LES STANDARDS DE SÉCURITÉ CÔTÉ NAVIGATEUR

GUIDE ANSSI

ANSSI-PA-009
28/04/2021

PUBLIC VISÉ :

Développeur **Administrateur** RSSI **DSI** Utilisateur

Informations

Attention

Ce document rédigé par l'ANSSI présente les **« Recommandations pour la mise en œuvre d'un site web : Maîtriser les standards de sécurité côté navigateur »**. Il est téléchargeable sur le site `www.ssi.gouv.fr`.

Il constitue une production originale de l'ANSSI placée sous le régime de la « Licence Ouverte v2.0 » publiée par la mission Etalab [13].

Conformément à la Licence Ouverte v2.0, le guide peut être réutilisé librement, sous réserve de mentionner sa paternité (source et date de la dernière mise à jour). La réutilisation s'entend du droit de communiquer, diffuser, redistribuer, publier, transmettre, reproduire, copier, adapter, modifier, extraire, transformer et exploiter, y compris à des fins commerciales.

Sauf disposition réglementaire contraire, ces recommandations n'ont pas de caractère normatif ; elles sont livrées en l'état et adaptées aux menaces au jour de leur publication. Au regard de la diversité des systèmes d'information, l'ANSSI ne peut garantir que ces informations puissent être reprises sans adaptation sur les systèmes d'information cibles. Dans tous les cas, la pertinence de l'implémentation des éléments proposés par l'ANSSI doit être soumise, au préalable, à la validation de l'administrateur du système et/ou des personnes en charge de la sécurité des systèmes d'information.

Évolutions du document :

VERSION	DATE	NATURE DES MODIFICATIONS
1.0	22/04/2013	Version initiale
1.1	13/08/2013	Corrections et précisions mineures, notamment sur TLS
2.0	28/04/2021	Refonte du guide sous l'angle des standards de sécurité web

Table des matières

1
Introduction

1.1 Objectif du guide

Ce document s'adresse aux administrations, entreprises privées et publiques mettant en œuvre un site web ou faisant usage d'une offre commerciale pour la réalisation ou l'hébergement de celui-ci.

Les recommandations de ce guide concernent la sécurité des contenus présentés par un navigateur web aux utilisateurs. Les sujets abordés se concentrent autour des standards du Web, dont les implémentations côté navigateur requièrent des paramètres à spécifier lors du développement et de l'intégration d'un site ou d'une application web, de façon à en garantir la sécurité. Les pratiques présentées sont à apprécier en fonction de la sensibilité du site web.

Selon le contexte de mise en œuvre du site web, les recommandations sont à présenter au prestataire de services en fonction des engagements contractuels, à l'hébergeur de services ou à l'équipe interne à même d'intervenir. Une population ayant des connaissances en développement web est la plus à même de comprendre et de décliner les recommandations du guide dans leurs contextes d'application.

1.2 Organisation du guide

Après avoir présenté en préambule les menaces et vulnérabilités habituelles dans le cadre des sites et applications web, le guide rappelle les règles d'hygiène de sécurité en les appliquant au contexte.

Le chapitre 4 est dédié à l'utilisation de TLS dans le cas web, qui est un pré-requis à la robustesse de l'ensemble des recommandations abordées au chapitre 5. Ce dernier détaille les principaux standards implémentés par les navigateurs et dont l'utilisation influe sur la sécurité d'un site web. Sont expliqués au fil de l'eau leur fonctionnement, les précautions d'usage et bonnes pratiques liées à leur mise en œuvre vis-à-vis du modèle de sécurité du Web.

Enfin, le dernier chapitre pointe quelques pratiques qui facilitent le maintien en condition de sécurité d'un site ou d'une application web pendant son cycle de vie, ce qui constitue un fil rouge duquel dépend l'efficacité de l'ensemble des recommandations du guide.

L'étude des mécanismes qui ne font pas intervenir d'implémentations côté navigateur, issues de standards, sont hors du périmètre de ce guide. Les considérations qui ne dépendent pas du modèle de sécurité de la plateforme web [1] sont volontairement ignorées.

1. https://en.wikipedia.org/wiki/Web_platform

1.3 Convention de lecture

Pour certaines recommandations de ce guide, il est proposé plusieurs solutions qui se distinguent par le niveau de sécurité qu'elles permettent d'atteindre. Le lecteur a ainsi la possibilité de choisir une solution offrant la meilleure protection en fonction du contexte et de ses objectifs de sécurité.

Ainsi, les recommandations sont présentées de la manière suivante :

R

Recommandation à l'état de l'art

Cette recommandation permet de mettre en œuvre un niveau de sécurité à l'état de l'art.

R -

Recommandation alternative de premier niveau

Cette recommandation permet de mettre en œuvre une première alternative, d'un niveau de sécurité moindre que la recommandation R.

R +

Recommandation renforcée

Cette recommandation permet de mettre en œuvre un niveau de sécurité renforcé. Elle est destinée aux entités qui sont matures en sécurité des systèmes d'information.

La liste récapitulative des recommandations est disponible en page 69.

1.4 Liste des sigles et acronymes

API *Application Programming Interface*
CDN *Content Delivery Network*
CMS *Content Management System*
CORS *Cross-Origin Resource Sharing*
CSP *Content Security Policy*
CSRF *Cross-Site Request Forgery*
CSS *Cascading Style Sheets*
CT *Certificate Transparency*
DOM *Document Object Model*
ECMA *European association for standardizing information and communication systems*, anciennement *European Computer Manufacturers Association*
ES6 *ECMAScript 6*
FTP *File Transfer Protocol*
HSTS *HTTP Strict Transport Security*
HTML *HyperText Markup Language*
HTTP *HyperText Transfer Protocol*
HTTPS *HTTP Secure*
JSON *JavaScript Object Notation*
JSON-P *JSON with Padding*
REST *REpresentational State Transfer*
RGPD Règlement Général sur la Protection des Données
SOP *Same-Origin Policy*
SQL *Structured Query Language*
SQLi *SQL Injection*
SRI *Subresource Integrity*
TCP *Transmission Control Protocol*
TLD *Top-Level Domain*
TLS *Transport Layer Security*
URL *Uniform Resource Locator*
XHR `XMLHttpRequest`
XML *Extensible Markup Language*
XSS *Cross-Site Scripting*

2
Menaces et types d'attaques

Les sites et applications web sont parmi les éléments les plus exposés aux utilisateurs d'un système d'information. Les problèmes de disponibilité (du simple ralentissement à l'interruption de service) ou de protection de l'information (la corruption ou la fuite de données) peuvent avoir des conséquences lourdes tant financièrement qu'en matière de réputation, non seulement pour l'entité en charge de l'application mais aussi pour ses utilisateurs et son hébergeur.

2.1 Menaces et objectifs des attaquants

Les menaces les plus connues pesant sur les sites et applications web sont la compromission des ressources, le vol de données et le déni de service.

- **La compromission des ressources** applicatives est une violation de l'intégrité du contenu, dont une conséquence peut être la défiguration [2] du site. Une telle attaque a pour objectif de modifier le site pour remplacer le contenu légitime par un contenu choisi par l'attaquant. Il s'agit, par exemple, de relayer un message politique, de dénigrer le propriétaire du site ou simplement de revendiquer l'attaque comme preuve d'un savoir-faire. Une perte d'intégrité peut aussi résulter en la conduite d'une attaque par point d'eau (*watering hole*), qui tend un piège aux visiteurs.
- **Le vol de données** est une attaque qui provoque la perte de la confidentialité de certaines données (authentifiants, informations personnelles, informations bancaires, etc.). Elle est réalisée dans un but souvent lucratif et aboutit la plupart du temps à des usurpations d'identité ou à des paiements frauduleux.
- **Le déni de service** [3] a pour objet de rendre indisponible le site attaqué pour ses utilisateurs légitimes que ce soit par l'arrêt ou par un ralentissement considérable du service.

Dans tous les cas, l'impact sur le propriétaire du site est évidemment un déficit d'image et un manque à gagner pour le cas d'un site servant de support à une activité commerciale. Dans certains cas, le préjudice peut s'étendre aux utilisateurs du site web.

Ces exemples représentent les résultats de types d'attaques parmi d'autres. Il ne faut toutefois pas négliger les scénarios d'attaques plus élaborés :

- Il est possible qu'un individu malveillant se serve d'un site web comme d'une porte d'entrée vers le système d'information de l'hébergeur. Un site peut aussi être utilisé comme relais dans

2. Action malveillante aussi appelée brouillage, ou *defacement* en anglais.
3. Abrégé `DoS` pour *Denial of Service* ou `DDoS` pour *Distributed Denial of Service*, voir aussi `https://www.ssi.gouv.fr/guide-ddos/`.

une attaque élaborée vers un système tiers ou comme dépôt de contenus illégaux, ces situations étant susceptibles de mettre l'exploitant légitime du site en difficulté. On parle alors d'une attaque par rebond.

- Dans le cas du point d'eau, l'attaquant vise à tendre un piège aux clients habituels du site (employés, collaborateurs et partenaires du propriétaire), qui vont activer une charge malveillante en le visitant. Les attaques de ce type ont en commun, contrairement à la défiguration ou au déni de service, de rechercher une certaine discrétion et peuvent par conséquent rester insoupçonnées pendant de longues périodes.

2.2 Panorama des classes d'attaques

Pour atteindre l'objectif visé, un attaquant exploite en général une ou plusieurs vulnérabilités du site ou de l'application web. Les moyens techniques employés pour mener à bien une attaque peuvent être regroupés en classes selon le type des vulnérabilités exploitées. Voici un échantillon des classes de vulnérabilités récurrentes :

- **XSS** : une attaque *Cross-Site Scripting* (XSS [4]) consiste en l'injection de données dans une page web dans le but de provoquer un comportement particulier du navigateur qui interprète cette page. Les données injectées ont la forme de langages interprétés par le navigateur tels que `JavaScript` ou HTML. Une attaque XSS cible les utilisateurs du site et vise en général à récupérer des secrets émis ou reçus par ceux-ci (sessions, coordonnées, mots de passe, informations bancaires, etc.), ou bien à effectuer des actions en leur nom ;
- **CSRF** : *Cross-Site Request Forgery* (CSRF [5]) est une classe d'attaques qui force un utilisateur à exécuter, à son insu, des actions privilégiées sur une application tierce sur laquelle il est authentifié. Ce type d'attaques a lieu lors de la navigation sur un site piégé qui émet des requêtes vers un site de confiance, mais vulnérable au CSRF ;
- **SSRF** : le *Server-Side Request Forgery* (SSRF [6]) est l'équivalent, côté serveur, du CSRF. Il s'agit, pour un attaquant, de demander au serveur vulnérable d'effectuer des requêtes vers des destinations choisies par l'attaquant en profitant éventuellement des privilèges du serveur (par exemple, l'accès à un réseau privé) ;
- **SQLi** : l'injection SQL (SQLi [7]) consiste en la transmission de code malveillant parmi les données entrantes qu'un serveur web utilise pour formuler une requête à destination d'une base de données. Cette classe d'attaques occasionne une perte de contrôle sur les données en base, ce qui peut mener à leur exfiltration, altération ou suppression ;
- **LFI/RFI** : *Local/Remote File Inclusion* est une classe de vulnérabilités qui repose sur l'intervention de fichiers dont l'inclusion n'est pas prévue par l'application. Ces fichiers, qu'ils soient locaux à l'application (LFI) ou distants mais accessibles via celle-ci (RFI) peuvent être directement la cible de l'attaque s'ils sont confidentiels, ou bien être utilisés comme moyens d'attaque, dans le cas de l'inclusion de code par exemple ;

4. https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html.
5. https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html.
6. https://cheatsheetseries.owasp.org/cheatsheets/Server_Side_Request_Forgery_Prevention_Cheat_Sheet.html.
7. https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html.

- **XXE** : les injections de type *XML External Entity* (XXE [8]) reposent sur l'utilisation de fonctionnalités dangereuses de la spécification XML, qui permettent le chargement de données externes par l'interpréteur XML. Exploiter une XXE peut donc déboucher sur du LFI/RFI ou bien directement sur de l'exécution de code arbtraire.

Seules les deux premières classes, et principalement le XSS, qui désigne une classe plus riche et plus dangereuse que le CSRF, sont traitées dans ce guide. Les autres types de vulnérabilités sont hors du périmètre du guide, car elles sortent du contexte de la maîtrise des standards de sécurité côté navigateur. Notons que dans le cas d'une attaque faisant intervenir différentes classes de vulnérabilités dont l'exploitation de failles XSS et CSRF, celles-ci se trouvent généralement au début du chemin de compromission. En effet, elles permettent souvent, via l'usurpation de l'identité d'un visiteur, l'accès à une plus grande surface d'attaque. Par exemple, une XSS peut permettre d'orchestrer le navigateur d'un administrateur dans le but de lui faire jouer une SQLi dans un contexte privilégié post-authentification.

La protection contre ces menaces passe à la fois par des mesures préventives de sécurisation des sites web, par leur maintien en condition de sécurité, par la mise en place de mécanismes permettant de détecter les tentatives d'attaques et par l'organisation régulière de tests d'intrusion et d'audits de sécurité.

8. https://cheatsheetseries.owasp.org/cheatsheets/XML_External_Entity_Prevention_Cheat_Sheet.html.

3
Rappel des règles d'hygiène

Les applications web sont un ensemble de briques logicielles qui font intervenir une variété d'acteurs. Elles se divisent, d'une part, en un ensemble d'instructions qui permettent aux navigateurs de s'adresser à un serveur qui implémente, d'autre part, la logique de traitement des requêtes en s'appuyant sur l'infrastructure d'hébergement. La sécurité des sites et applications web est donc à considérer selon plusieurs niveaux :

- **La conception du site ou de l'application**
  La sécurité des sites et applications web est un aspect à prendre en compte en début et non en fin de projet. Il est important de considérer les besoins en sécurité issus de l'analyse des risques, et de veiller à ce que ceux-ci soient traités tout au long des phases de conception. Cette pratique évite de voir les coûts de mise en conformité de sécurité devenir très importants tandis que les échéances de mise en production se rapprochent. De plus, la sécurisation *a posteriori* ne permet généralement pas la correction des mauvaises pratiques (ex. : utilisation d'une dépendance logicielle non maintenue), ni la mise en place de certains principes de sécurité (ex. : moindre privilège, défense en profondeur ou limitation des dépendances).
- **L'intégrité du comportement de l'application côté client**
  L'intégrité du comportement de l'application côté client est souvent ignorée car cet aspect est généralement considéré comme non maîtrisable. Cependant, un fournisseur de service web se doit de protéger ses utilisateurs légitimes des contenus tiers ou utilisateurs malveillants de l'application. Un applicatif ou un site web fait bien souvent appel à des ressources externes difficiles à maîtriser. Une modification de celles-ci peut avoir des conséquences directes sur le comportement de l'application. Un certain nombre de précautions permettent de s'assurer de l'intégrité des données émises depuis et vers les navigateurs, qui constituent une particularité importante de la plateforme web.
- **La configuration de l'infrastructure d'hébergement**
  La configuration de l'infrastructure d'hébergement est bien évidemment très importante. De bonnes pratiques et un certain nombre de recommandations sont à prendre en compte telles que la sécurisation des échanges, le durcissement des configurations, sans oublier l'administration et la supervision, qui ne doivent pas contribuer à augmenter la surface d'attaque.
- **Détection et information**
  En complément des mesures de sécurité et bonnes pratiques, il est recommandé d'être à même de détecter les vulnérabilités et attaques éventuelles. Pour le traitement des vulnérabilités, la planification d'audits de sécurité et de qualité du code doit être prise en compte dès la conception et maintenue durant toute la vie du système. Un audit permet également de vérifier que le système satisfait aux exigences de sécurité définies par un référentiel standard ou issues d'une analyse des risques. Dans le contexte de la détection d'une attaque, la protection contre les menaces à titre préventif seulement n'est pas suffisante. Il est nécessaire de mettre en place des mécanismes permettant de déceler des anomalies, de journaliser les actions illégitimes et de traiter l'information afin d'alerter au plus tôt en cas d'incident de sécurité.

3.1 Défense en profondeur

Le principe général de défense en profondeur consiste à mettre en œuvre plusieurs mesures de protection indépendantes en face de chaque menace envisagée.

Il est plus facile d'appliquer ce principe si le système à sécuriser est composé d'unités distinctes, aux interactions bien définies et possédant leurs propres mécanismes de sécurité. La défense en profondeur demande à ce que soient mises en œuvre les mesures de protection nécessaires et disponibles au niveau de chaque unité. Une mauvaise approche est, par exemple, de concentrer toutes les mesures de sécurité au niveau du point d'entrée du système et de constater que les fonctions internes sont complètement exposées en cas de vulnérabilité en amont, ou encore de ne considérer que le risque sur l'infrastructure en ne plaçant comme élément de sécurité qu'un simple pare-feu périmétrique.

Dans l'idéal, chaque brique logicielle de l'application web participe à la protection de l'ensemble du système. L'architecture logicielle du site web ainsi que l'infrastructure d'hébergement doivent participer à la défense en profondeur.

3.2 Moindre privilège

Ce principe vise à n'octroyer aux éléments et acteurs du système que les permissions strictement nécessaires pour fonctionner, ceci afin de limiter le risque de vol, d'altération ou de destruction de données dans le cas de compromission d'un ou plusieurs éléments.

Exemple

Application du principe de moindre privilège :

- à la conception de l'application, prévoir autant de rôles que de besoins d'accès aux données (lecture seule, écriture, etc.) ;
- limiter les permissions d'accès aux *Application Programming Interface*s (APIs) du navigateur pour une application web ;
- limiter les permissions de l'utilisateur applicatif sur le système de fichiers.

3.3 Réduction de la surface d'attaque

La réduction de la surface d'attaque consiste à ne pas exposer des services, accès et autres points d'entrée s'ils ne sont pas indispensables. Ce principe appliqué au développement logiciel demande que soit limitée la présence de composants logiciels dont l'usage n'est pas strictement nécessaire au fonctionnement nominal du système, afin de réduire son exposition logicielle. En complément, le système doit également chercher à minimiser son exposition réseau, à la conception comme au déploiement.

Information

Le principe de moindre privilège ne se substitue pas à la réduction de la surface d'attaque.

Exemple

Pratiques contribuant à réduire la surface d'attaque, par domaine d'implémentation :

- Au niveau du réseau : filtrage du port TCP nécessaire pour l'administration de l'application ;
- Au niveau du système : désactivation des services non nécessaires dans la configuration par défaut (ex. service FTP) ;
- Au niveau de l'application : exclusion des composants ou modules inutiles susceptibles d'avoir des failles exploitables (ex. module WebDAV, module proxy, bibliothèque de tests unitaires ou autres composants réservés au développement).

3.4 Sécurité des échanges de données

Il est important de garantir que les données soumises ou présentées ne seront ni interceptées, ni modifiées pendant le transport. Si le site ou l'application web présente des données à caractère personnel ce principe revêt un caractère obligatoire au regard de la loi informatique et libertés du 6 janvier 1978 et du Règlement Général sur la Protection des Données (RGPD).

L'objectif est également d'assurer à l'utilisateur que les données transmises ou reçues parviennent ou proviennent effectivement du site consulté et non d'une copie illégitime ou d'un intermédiaire frauduleux.

Dans le contexte du Web, ces garanties de confidentialité, d'intégrité, et d'authenticité lors de l'échange de données découlent de la mise en place du protocole HTTPS, abordée au chapitre 4.

3.5 Conformité du contenu présenté

L'objectif d'assurer la conformité du contenu présenté dans le cas d'un site ou d'une application web est de garantir à l'utilisateur que son navigateur interprète et affiche l'application et son contenu de façon conforme à l'intention du développeur, sans altération malveillante ou imprévue en amont ou en aval de l'échange de données.

3.6 Audit

Les bonnes pratiques de conception évoquées jusqu'ici doivent s'accompagner de processus permettant de détecter les vulnérabilités du site web. La mise en œuvre de ces processus se décline sous la forme d'actions manuelles et automatisées, lors des phases de conception comme d'exploitation de l'application web. Côté automatisation, on peut citer l'ajout de modules d'analyse à la chaîne d'intégration continue (ex. : détection de dépendances vulnérables, outils d'analyses

statique et dynamique). En ce qui concerne les actions organisationnelles, cela passe par l'expression d'une stratégie d'audit qui définit notamment des cadres et jalons (ex. : modification majeure, intervalle de temps). La réalisation de ces audits peut, le cas échéant, être confiée à des prestataires d'audit de la sécurité des systèmes d'information (PASSI) qualifiés pour leur conformité aux exigences de sécurité définies dans le référentiel PASSI [9]. Outre l'audit du code et de la configuration de l'application web, les tests d'intrusion peuvent être complétés par la mise en place d'un *bug bounty*, qui offre un cadre légal aux *hackers* éthiques pour faire remonter aux éditeurs les vulnérabilités de leurs produits en l'échange d'une prime. Sans pouvoir se substituer à un test d'intrusion ni à un audit conventionnel, le *bug bounty* offre une approche complémentaire intéressante par sa continuité dans le temps et par la variété des compétences qu'il est susceptible de mobiliser.

3.7 Journalisation

Les journaux d'événements constituent une brique technique indispensable à la gestion de la sécurité des systèmes d'information. L'activité de journalisation est un moyen de détection des incidents de sécurité et d'analyse du comportement d'un site ou d'une application web. Cette activité concerne la phase de conception de l'application, pour la génération de journaux (ex. traçabiltié d'un changement de privilèges), ainsi que la phase d'exploitation de l'application, pour laquelle l'application des *Recommandations de sécurité pour la mise en œuvre d'un système de journalisation* [2] est nécessaire. En effet, les recommandations portant sur l'horodatage des événements ainsi que sur la synchronisation des horloges entre les composants sont particulièrement pertinentes dans le cas d'une application web, bien souvent décomposée en plusieurs services amenés à générer leurs propres journaux de façon isolée. Il faudra donc parvenir à les corréler pour analyser et comprendre les événements de sécurité remontés par un outil prévu à cet effet. De plus, l'exposition élevée d'une application web fait des journaux d'événements et d'erreurs des moyens exploitables par un attaquant dans le but de réaliser des attaques de déni de service par saturation des journaux, ou encore pour exfiltrer les données sensibles parfois incluses dans certains journaux d'erreur. Il convient donc, malgré le caractère indispensable d'un système de journalisation, de respecter les bonnes pratiques visant à éviter que son utilisation puisse être détournée ou ses données corrompues lors d'une attaque.

9. https://ssi.gouv.fr/passi

4
Utilisation de TLS

La mise en place de HTTPS sur un site ou une application web est une garantie de sécurité qui repose sur TLS pour assurer la confidentialité et l'intégrité des informations échangées, ainsi que l'authenticité du serveur contacté.

L'absence de cette garantie peut entraîner de nombreux abus sans pour autant que l'intention soit malveillante. En exemple, certains *hotspots Wi-Fi* publics insèrent, à l'insu de l'utilisateur, du contenu publicitaire ou des informations annexes dans les pages provenant de sites en HTTP (ex. : horaires de vols dans les aéroports, menus de restaurants, etc.). Il n'en reste pas moins que ce type de pratiques peut être mal perçu par l'utilisateur. Par ailleurs, de nombreux acteurs du Web encouragent fortement l'utilisation de HTTPS : meilleur référencement par les moteurs de recherche, affichage de mises en garde lors de la visite de sites en clair.

La mise en place de HTTPS a pour objectif :

- de garantir, autant que possible, l'authenticité du site consulté ;
- de garantir également l'intégrité et la confidentialité des données échangées en bloquant les attaques de type *Man-In-The-Middle* (écoute, interception ou modification des échanges à la volée par des tiers, à l'insu de l'utilisateur).

Une attaque de type *Man-In-The-Middle* expose les utilisateurs du site web à l'injection de contenu malveillant pouvant conduire, entre autres, à la consultation de pages piégées incitant au téléchargement de codes malveillants ou à des attaques par rebond, que cela soit sur le réseau local de la victime, ou sur des sites tiers sur lesquels la victime a un compte. Ce type d'attaque cible des utilisateurs légitimes d'un site accessible en clair, et ne dépend pas de son contenu. Notamment, ces attaques peuvent toucher les utilisateurs d'un site dynamique comme statique.

La mise en place de HTTPS doit cependant respecter les bonnes pratiques en vigueur. À ce jour, les versions préconisées pour la mise en œuvre d'HTTPS sont TLS`v1.2` et TLS`v1.3`.

R1

Mettre en œuvre TLS à l'état de l'art

Il est nécessaire de mettre en œuvre les *Recommandations de sécurité relatives à TLS [4]* pour tout site même si celui-ci ne traite pas d'informations sensibles.

Pour éviter de perdre du trafic, les sites web acceptent souvent des connexions en HTTP et les redirigent vers HTTPS, laissant ainsi à un attaquant l'opportunité d'intercepter cette communication. L'utilisation de *HTTP Strict Transport Security* (HSTS, [7]) indique au navigateur d'utiliser automatiquement HTTPS pour tous les accès au site web. Il empêche également un utilisateur d'accepter de poursuivre la navigation sur un site non sécurisé en outrepassant les alertes de sécurité (certificat invalide, certificat généré par une autorité non reconnue, etc.) levées par les navigateurs.

R2

Mettre en œuvre HSTS

Il est nécessaire de mettre en œuvre HSTS afin de limiter les risques d'attaque de type *Man-In-The-Middle* dus à des accès non sécurisés générés par les utilisateurs ou par un attaquant.

Attention

Attention, la pérennité de l'accès en HTTPS est un prérequis indispensable à HSTS, qui rendra l'accès en clair impossible.

Le mise en œuvre de HSTS se fait par la transmission d'un en-tête HTTP lors de l'accès au site en HTTPS pour assurer son intégrité. Par défaut, la stratégie HSTS d'un site est enregistrée par le navigateur lorsqu'il est visité pour la première fois (*trust on first use*). Pour combler cette vulnérabilité initiale, le gérant d'un site peut décider de l'inscrire à une « liste préchargée » (voir HSTS *preload* [10]) de sites accessibles seulement en HTTPS, connue à l'avance par les navigateurs.

Exemple

Demander au navigateur d'utiliser exclusivement HTTPS pour se connecter au site visité et à ses sous-domaines, pour une durée d'un an :

```
Strict-Transport-Security: max-age=31536000; includeSubDomains;
```

À la suite de difficultés techniques et organisationnelles constatées auprès de certaines autorités de certification, un protocole nommé *Certificate Transparency* (CT, [8]) a vu le jour. Il permet de surveiller les certificats délivrés afin de s'assurer de leur légitimité et de pouvoir réagir rapidement en cas de problème. Ce protocole s'appuie sur des registres dans lesquels sont ajoutés les certificats lorsqu'ils sont signés par une autorité, les *Certificate Transparency Logs* (CT *logs*). Ces registres ont des propriétés cryptographiques qui les rendent difficiles à falsifier et efficaces à parcourir, ils agissent en preuves cumulatives qu'un certificat donné a bien été émis à un instant `T` par une autorité de certification. CT permet aux différents acteurs du Web de prendre des décisions plus avisées quant aux autorités de certification en lesquelles ils souhaitent avoir confiance et de détecter rapidement un certificat illégitime car absent ou incohérent avec le registre.

R3

Surveiller les CT logs

Il est recommandé que l'hébergeur ou le responsable d'un site web mette en œuvre un processus de surveillance des *Certificate Transparency logs* afin de détecter et révoquer les certificats illégitimes qui correspondent à des domaines sous son contrôle.

10. Inscription et informations sur `https://hstspreload.org`.

5
Description et mise en œuvre des mécanismes de sécurité web

5.1 Stratégie par défaut et évolutions

Huang et al. [6] résument la garantie de sécurité principale offerte par les navigateurs en ces termes : « les utilisateurs doivent pouvoir visiter, sans danger, des sites web arbitraires et exécuter les scripts fournis par ces sites ». Toutes les recommandations abordées dans ce guide reposent sur l'hypothèse que le navigateur est de confiance.

5.1.1 Same-Origin Policy (SOP)

L'`Origin` d'une page est définie par le triplet *protocole*, *destination* et *port* présent dans la barre d'adresse. Les ports 80 et 443 sont implicites, respectivement lors de l'usage de HTTP et HTTPS.

Exemple `Origin`	
URL complète	**`Origin`**
`http://www.exemple.org:8080` `http://www.exemple.org:8080/fichiers/page1.html` `http://www.exemple.org:8080/api/time`	`http://www.exemple.org:8080`
`https://mon.exemple.org` `https://mon.exemple.org/accueil.php` `https://mon.exemple.org/images/logo.jpg`	`https://mon.exemple.org`
`https://www.exemple.org` `https://www.exemple.org:443/accueil.html` `https://www.exemple.org/data/contacts.doc`	`https://www.exemple.org`

L'objectif de *Same-Origin Policy* (SOP, [23]) est de fournir un cadre de contrôle des interactions possiblement effectuées par les éléments embarqués dans une page web. SOP est une contrainte implémentée par tous les navigateurs du marché. Cette contrainte ne signifie pas que toutes les ressources doivent provenir d'une même `Origin`, mais impose des restrictions dans la communication entre composants lorsque ceux-ci ont des `Origins` différentes.

De manière générale, lorsque deux ressources embarquées sont issues d'une même `Origin`, aucune restriction n'est appliquée. À l'inverse, une stratégie de contrôle sera appliquée dans le cas de la communication entre des ressources embarquées d'`Origins` différentes, c'est-à-dire en *Cross-Origin*. Dans ce cas, l'inclusion de contenu, native au navigateur (ex. : afficher une image ou une `iframe`), sera permise, mais l'accès au contenu (de l'`iframe` par exemple) au moyen de scripts sera bloqué.

Les restrictions appliquées par la contrainte SOP se déclinent avec des subtilités en fonction de ce qui est adressé :

- **iframe :** une page ayant pour `Origin` A comportant une `iframe` ayant pour `Origin` B ne peut pas accéder ou intervenir sur le contenu de cette `iframe`. De même, l'`iframe` ne peut pas accéder à l'URL de la page parente ou à son contenu ;
- **XMLHttpRequest et Fetch :** une page web effectuant un appel simple à une URL issue d'un site différent de l'`Origin` appelante est autorisée à émettre la requête mais la réponse sera par défaut refusée par le navigateur au motif du non-respect de la SOP ;
- **Web Storage et IndexedDB :** les bases de données locales `WebStorage` et `IndexedDB` sont strictement propres à une `Origin` ;
- **Cookie :** l'accès à un `cookie` suit une stratégie similaire mais légèrement différente de SOP. Notamment, le chemin qui suit le nom de domaine est contrôlé, mais le port n'est pas pris en compte. En outre, un même `cookie` peut être envoyé sur plusieurs sous-domaines différents.

La limitation par défaut de SOP est contournable pour le cas des APIs `XMLHttpRequest` ou `Fetch` par la mise en œuvre de *Cross-Origin Resource Sharing* (CORS) et par l'utilisation d'une API d'échange de messages (*Cross-document messaging* ou *Web Messaging*) pour les `iframes` et fenêtres issues du `parent`.

Les ressources suivantes sont affichées ou activées peu importe leur `Origin` :

- **JavaScript** : tout script provenant d'une `Origin` différente de l' `Origin` de la page peut accéder aux mêmes données (variables, constantes, *Document Object Model* (DOM)[11], `Cookies`, `WebStorage`, etc.) qu'un script issu de l'`Origin` de la page.
  Exemple de risque : un code `JavaScript` issu d'une `Origin` différente de la page peut surcharger une fonction existante, comme la fonction `alert` de l'objet global `window` dans l'exemple suivant : `window.alert = function(m) { console.log("ALERT " + m); }`
- **CSS** : les *Cascading Style Sheets* (CSS) issues d'une `Origin` différente de la page ont la même influence sur la mise en page que celles en provenance de la même `Origin` que la page.
  Exemple de risque : une feuille de styles CSS issue d'une `Origin` différente de la page peut remplacer un style déjà défini ou charger des images externes.
- **Fichiers multimédia** : ils sont chargés sans différence de traitement selon l'`Origin`. En revanche, l'accès (en lecture ou en écriture) au contenu multimédia (ex. : les pixels d'une image) via un script est bloqué en *Cross-Origin*.

5.1.2 Cross-Origin Resource Sharing (CORS)

Il est parfois nécessaire de contourner la SOP (stratégie de sécurité par défaut du navigateur) afin de permettre l'appel de ressources en dehors de l'`Origin` telles que peuvent en fournir des services web tiers de météo ou d'actualités par exemple. La méthode utilisée dans ce cas est nommée *Cross-Origin Resource Sharing*. Cette méthode est normalisée [22] et vient en remplacement de plusieurs autres techniques jusqu'alors utilisées mais considérées comme dangereuses et limitées telles que la proxyfication ou l'utilisation de *JSON with Padding* (JSON-P).

11. Il s'agit de l'interface structurée de programmation pour les documents HTML.

CORS est un standard qui permet la définition explicite d'un contrat entre le serveur web et le navigateur qui spécifie les conditions d'acceptation d'échanges *Cross-Origin*. La négociation de ce contrat a lieu par l'intermédiaire d'en-têtes HTTP selon la cinématique suivante :

1. une requête *Cross-Origin* présente au serveur destinataire l'en-tête `Origin` fixé par le navigateur avec l'adresse du site en cours de consultation ;
2. le serveur précise dans l'en-tête de réponse `Access-Control-Allow-Origin` l'adresse du site depuis laquelle il accepte de répondre.
   Cette réponse peut être explicite `Access-Control-Allow-Origin: %ORIGINE_ADMISE%` ou implicite `Access-Control-Allow-Origin: *` ;
3. le navigateur reçoit la réponse et analyse l'en-tête `Access-Control-Allow-Origin`.
   Dans le cas général, si celui-ci est `*` ou égal à l'`Origin` courante, la réponse est acceptée. Dans le cas contraire la réponse est bloquée et une exception de sécurité est levée par le navigateur.

Le cas particulier des requêtes authentifiées, présentant les en-têtes `Cookie` ou `Authorization`, est traité dans la section 5.6.2 dédiée à CORS, qui aborde la mise en œuvre du standard avec plus de précision.

5.1.3 Content Security Policy (CSP)

Content Security Policy (CSP, [10]) permet de définir une stratégie de contrôle des accès aux ressources atteignables d'un site web donné par l'application de restrictions sous forme de liste d'autorisations (aussi appelée liste blanche).

La maîtrise de l'ensemble des ressources récupérées par un site web permet de réduire le risque d'apparition et l'exploitabilité de vulnérabilités XSS, abordées plus en détails en section 5.2. La définition de la liste des ressources autorisées peut être effectuée en utilisant :

- l'en-tête HTTP dédié, `Content-Security-Policy`, dans la réponse HTTP ;
- la balise équivalente, `<meta http-equiv="Content-Security-Policy">`, dans la réponse HTML.

Exemple

La stratégie CSP suivante demande au navigateur d'accepter uniquement des ressources servies depuis la même `Origin` que la page actuelle via une connexion sécurisée. En particulier, CSP permet la maîtrise des ressources dont l'activation n'est pas contrainte par la SOP (cf. section 5.1.1).

```
default-src 'self' https: ;
```

La stratégie CSP suivante demande au navigateur d'accepter uniquement des ressources servies depuis la même `Origin` que la page actuelle. Elle autorise l'inclusion de code `JavaScript` *inline*, c'est-à-dire directement dans le HTML ainsi que l'utilisation de fonctions d'évaluation de code (`eval()`, `Function()`, etc.), ce qui n'est pas une bonne pratique.

```
default-src 'self' ; script-src 'unsafe-inline' 'unsafe-eval' ;
```

Une description plus complète du fonctionnement et des bonnes pratiques liées à CSP est disponible à la section 5.3.

5.2 Protection contre les vulnérabilités XSS

Il existe de nombreux scénarios [19] d'attaques mettant en jeu une vulnérabilité XSS. Par exemple, s'il est possible d'injecter du contenu dans une page au travers d'une variable `GET` [12], un attaquant peut inciter (par exemple au moyen d'un courrier électronique trompeur) une victime à cliquer sur un lien fabriqué dans l'objectif d'appeler une page au contenu vulnérable et insérer dans celle-ci un script malveillant. L'attaquant pourra alors contrôler le navigateur de la victime qui pense pourtant visiter un site de confiance. Il est ainsi en mesure, par exemple, de voler la session de la victime et d'usurper son identité sur le site.

On distingue trois causes principales à l'origine des vulnérabilités XSS. Pour chacune d'entre elles, il existe des bonnes pratiques de développement qui permettent de réduire le risque d'une attaque XSS en diminuant la probabilité d'apparition de telles vulnérabilités.

5.2.1 Maîtrise des contextes de composition

Comme tout programme informatique qui permet à un utilisateur de saisir des informations, une application web doit porter une attention particulière à l'encodage des données en entrée. En particulier, côté navigateur il s'agit, lors de la restitution des données, de les afficher en prenant en compte leur contexte d'encodage dans la page web afin d'éviter qu'elles puissent être mal interprétées.

Certaines méthodes `JavaScript` vont directement modifier la source de la page selon la chaîne de caractères passée en paramètre et demander au navigateur de parser à nouveau le code pour y refléter les changements sur la page. Il est donc dangereux de les utiliser en conjonction avec des saisies utilisateur ou autres données non maîtrisées. Les méthodes qui permettent l'injection de code HTML, telles que `document.write()`, `insertAdjacentHTML()`, ou encore les propriétés `.innerHtml`, `.outerHTML`, doivent être utilisées avec précaution. Il est préférable de réaliser les modifications de texte par la propriété `textContent` d'un objet du DOM et l'insertion de contenu via les méthodes d'accès au DOM `document.createTextNode()` et `element.setAttribute()`. Ces propriétés et méthodes ont l'avantage de connaître le contexte cible et d'encoder automatiquement le paramètre donné en fonction de sa nature (du texte dans une page, un attribut au sein d'une balise HTML). Attention, un grand nombre de méthodes et de propriétés de l'API DOM, appelées *sinks* ou points d'injection, permettent, au-delà de la modification du contenu d'une page, l'altération du comportement de l'application web. Leurs paramètres d'entrée doivent donc faire l'objet d'une attention particulière lors du développement. En effet, même une propriété comme `textContent` qui va effectuer un encodage qui empêche son paramètre d'entrée d'être interprété par le navigateur comme du HTML peut se révéler être un *sink* lorsqu'elle est utilisée sur un élément HTML `script`. Pour se faire une idée de la surface d'attaque, il est intéressant de consulter la liste des *sinks* identifiés sur la page dédiée au standard *Trusted Types* [25].

12. Variable transmise directement dans l'URL, sous la forme `http://www.example.org/page?variable=valeur`.

R4

Utiliser l'API DOM à bon escient

Toute intervention sur le contenu client doit être réalisée via l'API DOM. Il est recommandé de ne pas utiliser, ou à défaut de contrôler l'usage de méthodes et propriétés qui effectuent des substitutions ou modifications de contenu dans un contexte à même d'altérer le comportement de l'application web.

Exemple

Dans les 2 exemples suivants, on peut observer que l'utilisation côté client de *Template Strings* ES6 présente une vulnérabilité XSS par rapport à l'utilisation de la balise HTML `<template>`.

```
38  var ligne = `
39    <th>${meteo.city.name}</th>
40    <td>${meteo.weather.temperature}</td>
41  `; // Risque XSS
42  var node = document.createElement("tr");
43  node.innerHTML = ligne; // Risque XSS
44  document.getElementById('display-grid').appendChild(node);
```

Listing 5.1 – Utilisation de *Template Strings* ES6

```
10 <template id="ligne">
11   <tr>
12     <th id="ville">$ville</th>
13     <td id="temperature">$température</td>
14   </tr>
15 </template>
```

```
47  var template = document.querySelector("#ligne");
48  var ligne = template.cloneNode(true);
49  ligne.content.querySelector("#ville").textContent = meteo.city.name;
50  ligne.content.querySelector("#temperature").textContent = meteo.weather.
      temperature;
51  document.getElementById('display-grid').appendChild(ligne.content);
```

Listing 5.2 – Utilisation de HTML `<template>`

En effet, si l'on considère la valeur frauduleusement modifiée de l'un des champs :

```
meteo.city.name = "<iframe src=\"http://autre.site.web\"></iframe>"
```

Ce contenu sera interprété dans le cas de l'utilisation de *Template Strings* ES6. Tandis qu'il sera affiché, et non interprété, dans le cas de l'utilisation de la balise HTML `<template>` avec `.textContent`.

La vulnérabilité XSS dépend de la méthode utilisée pour présenter les contenus aux utilisateurs.

Dans un objectif de contenir les vulnérabilités XSS, les pages HTML ne devraient pas être générées dynamiquement par le serveur et les contenus à inclure devraient l'être sur le client par consommation de services web après encodage *JavaScript Object Notation* (JSON) et envoi avec l'en-tête `Content-Type: application/json` pour les données, ou autres encodages sans ambiguïté pour le contenu multimédia.
Enfin, les pages HTML ne devraient pas contenir de code CSS (*inline styles*), ou `JavaScript` (*inline scripts*), mais faire référence à des fichiers externes dédiés.

R5

Dissocier clairement la composition des pages web

Il est recommandé de dissocier clairement les données (JSON), la structure (HTML), le style (CSS) et la logique (`JavaScript`) d'une page web afin de réduire le risque d'occurrence de vulnérabilités XSS.

Information

Le contrôle du respect de cette recommandation est réalisable par la mise en œuvre de CSP (cf. section 5.3).

Attention

De manière générale, la méthode consistant à générer dynamiquement une page sur le serveur par inclusion d'un modèle et fusion avec des données présente plus de risques que d'effectuer le rendu directement sur le navigateur, puisque cela ouvre la porte à des injections côté serveur lors du parsing du *template*. Il n'est pas recommandé de transmettre un contenu tiers ou de faible confiance directement dans la page HTML ou de contourner, côté serveur, les mécanismes de sécurité implémentés par les navigateurs.

R6

Expliciter la nature d'une ressource avec l'en-tête Content-Type

L'application de la recommandation R5 permet aussi de spécifier de manière explicite la nature d'un contenu et donc le contexte dans lequel le navigateur peut l'utiliser. Spécifier un `Content-Type` approprié contribue à réduire le risque qu'une ressource soit interprétée de manière inattendue et exploitée par un attaquant.

Un autre cas rencontré fréquemment est celui du XSS stocké. L'attaquant dépose les éléments de l'attaque de manière permanente sur le site, par exemple au moyen d'un système de commentaires (*forum web*), afin que tous les visiteurs ultérieurs du site soient touchés par le code malveillant. Que le code malveillant soit stocké sur le serveur (XSS stocké), résultat d'un lien piégé (XSS reflété) ou qu'il réside uniquement dans le navigateur (XSS DOM), une bonne pratique de conception permet d'augmenter sa protection contre ce type d'attaques : l'encodage contextuel ou « échappement ».

Il est important de s'assurer que toutes les données [13] issues de sources externes incluses dans la page web soient protégées, c'est-à-dire aient subi un traitement qui empêche leur interprétation dans le contexte où elles sont employées. Cette pratique est courante au niveau des contrôles réalisés côté serveur, mais il est aussi nécessaire de faire effectuer ce contrôle par le navigateur, qui est l'un des garants de l'intégrité des contenus présentés aux utilisateurs légitimes.

R7

Vérifier l'échappement des contenus inclus

Les données externes employées dans quelque partie que ce soit de la réponse envoyée au navigateur doivent avoir fait l'objet d'un « échappement » adapté au contexte d'interprétation.

13. On entend ici par données : les paramètres, les en-têtes, fichiers, et saisies réalisées par l'utilisateur ou d'autres sources de données externes que l'application utilise en paramètres d'entrée, un webservice par exemple.

Exemple

L'exemple ci-après montre l'utilisation de fonctions d'échappement :

```
async function printUsername(username) {
  try {

    // Utilisation de encodeURIComponent pour échapper le contenu non fiable (contexte URL)
    const res = await fetch('https://my-api.com/user/' + encodeURIComponent(username));
    const jsonData = await res.json();

    // Utilisation de textContent pour échapper le contenu non fiable (contexte HTML)
    document.getElementById('username').textContent = jsonData.username;

  } catch (err) {
    // Gestion de l'erreur à implémenter
  }
}
```

Listing 5.3 – fetch-and-escape.js

```
<!doctype html>
<html>
  <body>
    <h1>Bienvenue <span id="username"> </span> !</h1>
    <script src="fetch-and-escape.js"></script>
  </body>
</html>
```

Listing 5.4 – welcome.html

R8

Vérifier la conformité des données issues de sources externes

Il est recommandé de vérifier, chaque fois que c'est possible, que les données ont bien la forme attendue. Lorsque cela est possible, une approche par liste d'autorisations est recommandée : par exemple une donnée censée être numérique ne doit être composée que de chiffres.

L'en-tête non-standard `X-XSS-Protection` n'est plus préconisé pour se prémunir des vulnérabilités XSS. Cet en-tête permettait la configuration d'une heuristique de filtrage XSS implémentée par les navigateurs. L'augmentation de la surface d'attaque entraînée par l'implémentation de ce filtre, l'introduction de nouvelles vulnérabilités et l'existence de contournements font que le support de ce mécanisme est en cours de retrait par certains navigateurs et a déjà été retiré par d'autres.

La mesure de sécurité standardisée à mettre en œuvre est *Content Security Policy* (CSP, voir section 5.3), qui offre une protection supérieure contre ce type d'attaques. L'usage de `X-XSS-Protection` ne présente d'intérêt que pour les navigateurs anciens qui ne supporteraient pas CSP.

Attention

Afin de désactiver le filtrage implicitement réalisé par `XSS-Protection`, il est nécessaire de placer la valeur d'en-tête suivante `X-XSS-Protection: 0`. En l'absence d'une CSP stricte ou pour le support de navigateurs anciens, la valeur `X-XSS-Protection: 1; mode=block` est tolérée.

5.2.2 Maîtrise de l'évaluation de code

La génération de code JavaScript à la volée est parfois utilisée afin d'intégrer des données en entrée transmises sous forme de chaînes de caractères. Cette mauvaise pratique est une opportunité pour un attaquant d'exploiter la vulnérabilité XSS introduite par l'usage de la fonction JavaScript `eval()`. En effet, l'utilisation de la fonction de désérialisation `eval()` avec une chaîne de caractères forgée permet à un attaquant de modifier le contenu et le comportement du site.

L'usage de la fonction `eval()` est donc à proscrire au profit, pour l'intégration de données structurées, de l'utilisation du format JSON et de la méthode associée `JSON.parse()`.

Exemple

Exemple d'utilisation de la fonction `JSON.parse()` à la place de la fonction `eval()` :

```
1 var raw_string = '{ "nom" : "dylan", "prenom" : "bob" }';
2 var forged_string = raw_string + ',alert()';
3 var parsed = eval(`(${raw_string})`);
4 console.log(parsed.prenom); // Affiche "bob"
5 var danger = eval(`(${forged_string})`); // alert() interprété, pop-up XSS !
```

Listing 5.5 – Utilisation incorrecte de `eval()`

```
1 var raw_string = '{ "nom" : "dylan", "prenom" : "bob" }';
2 var forged_string = raw_string + ',alert()';
3 var parsed = JSON.parse(raw_string);
4 console.log(parsed.prenom); // Affiche "bob"
5 var safe = JSON.parse(forged_string); // Exception levée, pas de XSS
```

Listing 5.6 – Utilisation correcte de `JSON.parse()`

R9

Proscrire l'usage de la fonction eval()

La fonction `eval` est dédiée à la transformation de chaîne de caractères en code JavaScript. L'usage de cette fonction doit être proscrit.

Information

Les fonctions JavaScript permettant la modification de contenu par insertion de fragment interprétable sont des fonctions susceptibles d'injecter du code malveillant. L'usage de ces fonctions dans un site ou une application web augmente à la fois la probabilité d'apparition et l'impact d'une vulnérabilité XSS.

Les fonctions `setTimeout()` et `setInterval()` sont des *timers* JavaScript pouvant être utilisés avec, en premier argument, un morceau de code sous forme de chaîne de caractères, ou une fonction. Le morceau de code sera compilé et exécuté à l'issue du *timer*, la fonction sera appelée comme un *callback* JavaScript classique.

Toute méthode permettant l'insertion de contenu interprétable dans son contexte d'exécution est susceptible de permettre une injection de code. Ainsi, toute insertion non contrôlée de chaîne de caractères dans une page HTML induit une exposition à une vulnérabilité XSS.

Exemple

Utilisation de la fonction `setInterval()`. Dans l'hypothèse où la variable `message` est contrôlée par un attaquant (ex. : saisie utilisateur), le premier exemple comporte une vulnérabilité XSS.

```
var message = 'Bonne année"),alert("';
var count = 3;
var ival = setInterval(` // setInterval + chaîne de caractères
  if (count === 0) clearInterval(ival), console.log("${message}");
  else console.log(count--);
`, 1000); // 3 2 1 Bonne année + pop-up XSS
```

Listing 5.7 – Utilisation incorrecte de `setInterval()`

```
var message = 'Bonne année"),alert("';
var count = 3;
var ival = setInterval(() => { // setInterval + lambda javascript
  if (count === 0) clearInterval(ival), console.log(message);
  else console.log(count--);
}, 1000); // 3 2 1 'Bonne année"),alert("' -> pas d'interprétation
```

Listing 5.8 – Utilisation correcte de `setInterval()`

R10

Proscrire l'usage de constructions basées sur l'évaluation de code

Interdire l'usage des constructions `JavaScript` dont l'interprétation des paramètres peut aboutir sur de l'exécution de code arbitraire. Des exemples de telles constructions sont `setInterval` et `setTimeout` avec une chaîne de caractères en paramètre, le constructeur `Function('code')`, ou encore la méthode `.constructor('code')` du prototype d'une fonction.

5.2.3 Maîtrise de l'intégrité des ressources

L'organisation du code en fichiers de ressources séparés CSS et `JavaScript` est encouragée (R5) mais cette pratique doit être complétée par l'utilisation d'un mécanisme permettant d'assurer l'intégrité des ressources utilisées par le navigateur.

Dans le cas d'un site utilisant des ressources internes (fichiers CSS et `JavaScript`) pour son propre usage, la mise en œuvre d'un mécanisme de vérification d'intégrité permet de contrôler la non-altération des fichiers de ressources présents sur le serveur.

Il est fréquent que les ressources utilisées, CSS, fontes ou `JavaScript`, soient externes au site consulté et hébergées par des *Content Delivery Network*s (CDNs) afin d'améliorer la disponibilité du site et d'économiser de la bande passante. Cette pratique présente cependant des risques car elle étend la surface d'attaque jusqu'aux CDNs. En effet, par défaut une ressource corrompue ne sera pas détectée et se diffusera sur tous les sites qui en font usage. L'utilisation d'un mécanisme de vérification de l'intégrité des ressources issues d'un CDN, tel que *Subresource Integrity* (SRI, [12]), permet de s'assurer que les fichiers de ressources actifs correspondent bien à ceux qui ont été audités et validés en phase d'intégration logicielle. Une meilleure maîtrise du contenu présenté aux utilisateurs limite la vulnérabilité au XSS en cas, par exemple, de compromission d'un CDN.

SRI permet d'exposer, via l'attribut dédié `integrity`, le résultat attendu d'un *hash* réalisé sur un fichier de ressource dite *active*, c'est-à-dire issue d'une balise `<link>` ou `<script>`. Pour chaque empreinte déclarée, le navigateur va comparer le *hash* de la ressource à charger avec le *hash* attendu, et bloquer la ressource si les empreintes sont différentes. La mise en place de SRI n'a d'intérêt que dans le cadre d'un transport HTTPS qui garantit l'intégrité de ce champ, dont l'empreinte doit avoir été calculée après vérification de la conformité de la ressource.

Exemple

Exemple de mise en œuvre de SRI pour le contrôle des ressources `bootstrap` et `jquery`.

```
<!DOCTYPE html>
<html>
<head>
<link rel="stylesheet"
  href="https://netdna.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css"
  integrity=
  "sha384-BVYiiSIFeK1dGmJRAkycuHAHRg32OmUcww7on3RYdg4Va+PmSTsz/K68vbdEjh4u"
  crossorigin="anonymous">
<script type="text/javascript"
  src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/js/bootstrap.min.js"
  integrity=
  "sha384-Tc5IQib027qvyjSMfHjOMaLkfuWVxZxUPnCJA7l2mCWNIpG9mGCD8wGNIcPD7Txa"
  crossorigin="anonymous">
</script>
<script type="text/javascript"
  src="https://code.jquery.com/jquery-3.2.1.min.js"
  integrity=
  "sha384-xBuQ/xzmlsLoJpyjoggmTEz8OWUFM0/RC5BsqQBDX2v5cMvDHcMakNTNrHIW2I5f"
  crossorigin="anonymous">
</script>
...
</head>
<body>
...
```

Le calcul de l'empreinte des fichiers de ressources peut être réalisé par un simple script :

```
#!/bin/sh
METH=$1
URL=$2
RES=`curl -s $URL | openssl dgst -$METH -binary | openssl enc -base64 -A`
echo "integrity=\"$METH-$RES\""
```

Lancement :

```
./sri.sh sha384 https://code.jquery.com/jquery-3.2.1.min.js
```

Remarque : la mise en œuvre de SRI (calcul des empreintes et définition des attributs `integrity` correspondants) peut être automatisée par l'utilisation de *bundlers* tels que `webpack` combinés à des modules dédiés comme `webpack-subresource-integrity`. Attention, il est essentiel d'auditer la dépendance afin de s'assurer qu'elle ne présente pas de risque pour la sécurité de l'application avant d'en calculer l'empreinte, y compris en cas de mise à jour de la dépendance.

Contrairement au traitement CSP, la spécification SRI ne permet pas (à ce jour) la définition d'URL de *reporting* pouvant alerter lors d'un échec du contrôle d'intégrité. En cas d'échec, seul un défaut de rendu ou d'exécution de la page sera constaté par l'utilisateur. Une trace de sécurité sera générée

mais celle-ci ne sera visible que dans les journaux du navigateur.

R11

Contrôler l'intégrité des contenus internes

Il est recommandé de mettre en œuvre SRI pour les ressources `JavaScript` et CSS internes.

R12

Contrôler l'intégrité des contenus tiers

Dans le cas d'un site en HTTPS, il est recommandé de mettre en œuvre systématiquement le contrôle de l'intégrité des ressources via SRI afin de réduire le risque de vulnérabilité XSS, en particulier pour les contenus issus d'un CDN.

Attention

- La spécification SRI est limitée aux contrôles des ressources de types CSS et `JavaScript`. Elle ne s'applique pas au contenu multimédia ou aux pages HTML.
- SRI permet de vérifier l'intégrité d'une ressource `JavaScript` téléchargée mais ne permet pas de vérifier l'intégrité des dépendances appelées par celle-ci.
- La mise en œuvre de SRI implique une parfaite maîtrise de la configuration des en-têtes de cache générés par le serveur HTTP, afin d'éviter les faux positifs.
- La définition des attributs `integrity` implique en général l'utilisation de ressources dont l'URL comporte un numéro de version. En effet, dans le cas d'une ressource hébergée par un CDN par exemple, une URL non versionnée correspondra souvent à la dernière version disponible de la ressource, et non à la version utilisée pour le calcul initial de l'empreinte lors de la définition de SRI.

Information

Le standard *Trusted Types* [25], mentionné à la section 5.2.1, propose un mécanisme natif au navigateur permettant de verrouiller les points d'injection abordés jusqu'ici (ex. : `innerHTML`, `eval`, etc.) sans en proscrire l'usage. Cette spécification, en brouillon à l'heure actuelle, force l'utilisation de valeurs typées et maîtrisées à la place de chaînes de caractères. Dans le cas où une refonte du code ne permet pas de se passer de l'utilisation de tels points d'injection, il peut être intéressant de suivre les évolutions de *Trusted Types* et de ses implémentations.

5.3 Mise en œuvre de Content Security Policy (CSP)

Comme introduit en section 5.1.3, *Content Security Policy* (CSP) est un standard [10], implémenté par tous les navigateurs modernes, à mettre en place lors de l'intégration d'un site ou d'une application web. Il s'agit d'une contre-mesure très efficace contre le XSS qui vient en complément des bonnes pratiques de conception et de développement. CSP est une mesure de défense en profondeur forte, mais elle ne se substitue pas à la correction des vulnérabilités identifiées dans l'application web.

5.3.1 Principe de liste d'autorisations CSP

CSP permet de définir une stratégie de restriction des ressources accessibles par le navigateur lors de la navigation sur un site web. Ces restrictions sont effectuées par le biais d'une liste d'autorisations (aussi appelée liste blanche) spécifique au site ou à la page. Ainsi, les ressources (scripts, styles ou images) issues d'origines non déclarées dans la CSP, mais présentes dans les pages, sont bloquées ou non exécutées par le navigateur dans l'objectif de mettre en œuvre le principe de moindre privilège.

R13

Restreindre les contenus aux ressources fiables

Il est recommandé de mettre en œuvre CSP afin de présenter aux navigateurs une liste des sites reconnus comme présentant des ressources fiables et ainsi contribuer au principe de moindre privilège en réduisant le risque potentiel de vulnérabilité XSS.

5.3.2 Mise en œuvre de CSP

Le complément apporté par l'utilisation de CSP à la stratégie de sécurité par défaut *Same-Origin Policy* est, quant à lui, à l'initiative du site web et doit être déclaré explicitement par celui-ci aux navigateurs au moyen d'un en-tête de réponse HTTP, ou par l'utilisation d'une balise HTML `<meta>` dans le contenu de la page.

Que la CSP soit déclarée via en-tête ou balise, celle-ci est efficace seulement si le site ou l'application web utilise un transport HTTPS. Sans utilisation de HTTPS, l'intégrité des en-têtes ou du corps de la réponse ne peut être garantie.

La mise en œuvre de CSP par les en-têtes HTTP permet l'expression d'un plus grand nombre de stratégies que via balise HTML (`frame-ancestors`, `sandbox`). En complément, ce positionnement autorise la mise en œuvre d'une URL de collecte des violations de CSP (`report-uri`) ainsi qu'un mode de déploiement progressif (`report-only`) notifiant des erreurs sans effectuer de blocage.

R14

Mettre en œuvre CSP par en-tête HTTP

Il est recommandé de privilégier la mise en œuvre de CSP par l'utilisation de l'en-tête HTTP `Content-Security-Policy`.

La mise en œuvre par en-tête HTTP peut être réalisée :

- en intervenant sur la configuration de la chaîne de *reverse-proxies* [14] en place ;
- par une demande spécifique auprès de l'hébergeur ;
- ou par la configuration des paramètres de sécurité du CMS [15] ou *framework* utilisé.

14. Un *reverse-proxy* est un serveur mandataire inverse qui se situe habituellement en frontal d'un serveur applicatif et peut assurer des fonctions de sécurité, telles que le filtrage des requêtes, pour celui-ci.
15. *Content Management System* ou système de gestion du contenu, une application dédiée à la conception et à la mise à jour dynamique s'un site web et qui en simplifie l'édition du contenu. Exemple : *WordPress* est un CMS.

Information

L'application de plusieurs CSP pour une même page ne peut aller que dans le sens du durcissement de la stratégie de restriction :

- un serveur d'application peut choisir de positionner un ou plusieurs en-tête CSP, puis un *reverse proxy* peut faire de même. Les stratégies définies par chaque en-tête sont prises en compte dans l'ordre de réception des en-têtes par le navigateur, seulement dans le sens de l'affermissement de la stratégie globale ;
- si une stratégie CSP est déjà définie par un en-tête HTTP, une balise HTML `<meta>` CSP peut uniquement venir en complément, sans l'affaiblir ;
- une page web peut comporter plusieurs balises `<meta>` CSP. Dans ce cas, elles sont prises en compte dans leur ordre de déclaration, là encore uniquement dans le sens du durcissement ;
- si une page web présente deux en-têtes CSP et deux `<meta>` CSP, les ressources chargées par le navigateur doivent satisfaire à l'ensemble des directives présentes, en conservant la directive la plus stricte en cas de conflit lors de la composition.

L'application d'une série de contraintes de CSP complémentaires peut être réalisée directement au niveau de la page au moyen d'une balise HTML `<meta>` CSP si l'on souhaite s'assurer de l'absence de scripts tiers (publicité, analyseur de trafic, etc.) lors de l'affichage ou de la saisie de données sensibles (mots de passe, informations bancaires, etc.).

R14 -

Mettre en œuvre CSP par balise meta dans les pages HTML

Si cela n'est pas possible via en-tête, ou dans des cas particuliers d'affermissement d'une stratégie, il est recommandé de mettre en œuvre CSP dans les pages HTML par l'utilisation de la balise HTML `<meta>`.

Attention

La restriction définie par une stratégie CSP issue d'une balise `<meta>` ne s'applique pas aux contenus qui la précèdent dans le DOM. Il est donc nécessaire de la placer le plus tôt possible dans la page HTML. Cela explique aussi pourquoi il est préférable de déclarer une CSP au moyen de l'en-tête HTTP dédié.

Les *plugins* cités dans les exemples suivants ne sont fournis qu'à titre indicatif et n'ont fait l'objet d'aucune vérification dans le cadre par exemple d'un visa de sécurité.

Exemple

Sans devoir intervenir directement sur la chaîne de *reverse-proxies* de l'hébergeur, il est possible de mettre en œuvre CSP au sein d'une infrastructure `WordPress` par l'utilisation d'un *plugin* tel que :

- **gd-security-headers**
  `https://wordpress.org/plugins/gd-security-headers/`
  Ce *plugin* est dédié à la définition des en-têtes de sécurité. Son usage ne se limite donc pas à CSP. Il permet la définition d'autres en-têtes de sécurité abordés dans

ce guide, tels que HSTS ou `Referrer-Policy`.

`Drupal` propose un *plugin* permettant la définition de stratégies de sécurité HTTP par configuration :

- **Security Kit**
  `https://www.drupal.org/project/seckit/`
  Ce *plugin* est dédié à la définition de tous les en-têtes de sécurité. Son usage ne se limite donc pas à la définition de l'en-tête `Content-Security-Policy` mais couvre également `Strict-Transport-Security` (`HSTS`).

5.3.3 Directives CSP par type de ressource

Une stratégie CSP est constituée d'un ensemble de directives qui permettent de définir des restrictions pour un certain nombre de types de ressources. En voici un échantillon :

- `script-src`, `style-src`, `img-src`, `media-src`, `object-src`, `font-src` : origines des `JavaScript`, CSS, des images, des éléments audio et vidéo, des contenus embarqués type `PDF` et des fontes ;
- `child-src` et `frame-ancestors` : origines des *workers* et *frames* enfants (*child*) et parents (*ancestors*) ;
- `form-action` et `connect-src` : origines vers lesquelles il est acceptable d'envoyer des formulaires et d'initier des connexions asynchrones (XHR et `Fetch`, `WebSockets`, `EventSource`) ;
- `default-src` : indication au navigateur du comportement à adopter vis-à-vis d'une ressource en l'absence de directive spécifique. `default-src` ne s'applique qu'aux directives omises et peut être définie en complément d'autres directives plus ciblées ;
- d'autres directives sont globales à tous les types de ressources : `upgrade-insecure-requests`, et `block-all-mixed-content`, qui maîtrisent le chargement de ressources en clair au sein d'une page en HTTPS, ou `sandbox` par exemple, qui permet l'application de restrictions de fonctionnalités du navigateur (voir section 5.7.3.2).

La majorité des directives CSP (à l'exception, notamment, des dernières de la liste précédente, qui ne se terminent pas par `-src`) acceptent pour valeur une liste de sources de contenu séparées par des espaces. Selon les directives, une source de contenu peut être décrite par :

- tout ou partie d'une origine : `https:` ou `domaine.fr` ou `https://domaine.fr:8443` ;
  - cette syntaxe accepte le caractère générique « étoile » pour le protocole, les sous-domaines, le port, ou la source entière :
    - `*://*.domaine.fr:*` autorise tous les sous-domaines, peu importe le protocole ou le port utilisé,
    - définir une directive à « `*` » autorise toute source de contenu ;
- le mot-clé `'none'` désactive la prise en charge du type de ressource en question ;
- le mot-clé `'self'` correspond à l'`origin` de la page actuelle ;
- le mot-clé `'unsafe-inline'` s'applique uniquement aux directives `script-src` et `style-src`, il active la prise en charge de contenu *inline*, c'est-à-dire du code `JavaScript` ou CSS figurant au sein d'une page HTML ;

- le mot-clé `'unsafe-eval'` s'applique uniquement à la directive `script`, il active la prise en charge des fonctions d'évaluation de code ;
- une autorisation spécifique à un `script` ou à un `style` *inline*, donnée sous la forme d'une empreinte (ex. : `'sha256-B2yPHKaXnvFWtRC[...]F8='`) ou d'un *nonce* repris lors de l'inclusion de la ressource (ex. : `nonce-2726c7f26c`) et renouvelé à chaque transmission de la CSP ;
- d'autres mots-clés existent et peuvent être ajoutés selon l'évolution du standard, comme par exemple le mot-clé `strict-dynamic`, introduit par CSP niveau 3, qui propage une autorisation spécifique aux ressources chargées récursivement.

Information

Sauf mention contraire avec `'unsafe-inline'` ou `'unsafe-eval'`, la définition d'une CSP désactive implicitement l'interprétation de code *inline* et des fonctions d'évaluation de code.

Attention

En l'absence de `default-src`, ne pas définir une directive revient à autoriser toutes les sources de contenu pour chaque type de ressource omis. L'absence de directive est donc identique à la définition du caractère générique « `*` » comme origine pour celle-ci (sauf pour le cas particulier des mots-clés `unsafe-*`, qui ne désignent pas des origines).

5.3.4 Contrôle des bonnes pratiques

L'adoption de CSP contribue à la réduction des vulnérabilités XSS en bloquant par convention :

- les ressources en base64 `<img src="...38==" />` ;
- la génération de code `JavaScript` par évaluation ;
- l'utilisation de CSS et de `JavaScript` *inline*, directement dans les pages HTML.

CSP incite à mettre en œuvre les bonnes pratiques de séparation entre les ressources CSS, `JavaScript` et HTML (R5) et de non-évaluation de code (R9 et R10). Même si, au titre de la rétrocompatibilité et pour faciliter l'adoption du standard, CSP met à disposition des mots-clés qui permettent de contourner le comportement par défaut, il est préférable de les ignorer.

R15

Interdire des contenus inline

Les contraintes CSP ne doivent pas présenter les mots-clés suivants : `data:`, `'unsafe-eval'` ou `'unsafe-inline'`.

Notons que, par l'interdiction de l'inclusion de code `JavaScript` et CSS *inline* (*i.e.* directement dans le HTML), CSP contribue à la réduction de la surface d'attaque et donc de la probabilité d'apparition d'une vulnérabilité XSS. En outre, son fonctionnement par déclaration des domaines atteignables réduit également l'exploitabilité de telles vulnérabilités (*i.e.* le *post-XSS*), puisqu'un attaquant ne pourra pas demander au navigateur de récupérer du code malveillant ou de divulguer une information si cela requiert l'accès à un site qui ne figure pas dans la liste d'autorisations.

Information

Dans un contexte de transition, afin de tendre vers l'absence de contenu *inline* et pour faciliter l'application de la recommandation R15, il est possible d'ajouter à une CSP des autorisations spécifiques (*hash* ou *nonce*) pour certains contenus *inline* de confiance, tout en bloquant ceux qui ne seront pas explicitement déclarés ou dépendants (au sens `strict-dynamic` [16]) des premiers.

D'une version du standard à l'autre, et d'une implémentation navigateur à l'autre, le nombre de directives CSP disponibles est amené à varier. Afin de faciliter le suivi de ces évolutions, mais aussi pour limiter les risques d'oubli lors de la mise en place de CSP, il est essentiel de définir un comportement par défaut.

R16

Définir la directive default-src

Lors de l'élaboration d'une CSP, il est recommandé de veiller à ce qu'elle contienne au moins la directive `default-src`, et que celle-ci ne soit pas simplement positionnée à « `*` ».

Information

Lorsque les besoins de sécurité sont élevés pour une page ou un site en particulier (ex. : une mire d'authentification), il peut être intéressant de définir `default-src 'none'`, puis d'inclure dans la liste d'autorisations chaque type de ressource utile.

Exemple

Exemples d'en-têtes CSP mettant en œuvre les bonnes pratiques vues jusqu'ici :

- toutes les ressources doivent provenir de l'origine actuelle, à l'exception des images, pour lesquelles nous faisons aussi confiance à `my-cdn.fr`, en HTTPS :

```
Content-Security-Policy: default-src 'self'; img-src 'self' https://my-cdn.fr;
```

- la page ne doit charger aucune ressource, mais peut afficher une `iframe` provenant d'un sous-domaine de confiance :

```
Content-Security-Policy: default-src 'none'; child-src https://ifr.domaine.fr;
```

Dans les deux cas, la prise en charge des contenus *inline* est à son mode par défaut lors de la définition d'une CSP, c'est-à-dire désactivée.

5.3.5 Protection contre le clickjacking

Le détournement de clic, ou *clickjacking*, est un type d'attaque dans lequel une page web trompeuse incite un utilisateur légitime à cliquer sur du contenu en apparence légitime qui le mène en réalité à effectuer des actions, à son insu, sur d'autres sites. Ces attaques sont en général mises en œuvre au moyen d'une page piégée incluant des cadres (`iframes`) invisibles qui pointent vers des sites légitimes sur lesquels l'utilisateur piégé a ouvert une session.

La protection contre ce type d'attaques consiste en la mise en place de mécanismes qui interdisent au navigateur l'inclusion du site à protéger dans un site tiers au travers d'une `frame` ou `iframe`.

16. Utilisation de `strict-dynamic` (CSP 3) : https://w3c.github.io/webappsec-csp/#strict-dynamic-usage.

La directive `frame-ancestors` de CSP est une contre-mesure au détournement de clic car elle permet de spécifier une liste d'origines depuis lesquelles le navigateur a le droit exclusif de charger le site protégé dans un cadre.

Exemple

- la page ne peut en aucun cas être chargée dans un cadre (`frame` ou `iframe`) :

```
Content-Security-Policy: default-src 'self'; frame-ancestors 'none';
```

- la page peut être chargée dans un cadre seulement si l'`Origin` appelante est identique à l'`Origin` appelée :

```
Content-Security-Policy: default-src 'self'; frame-ancestors 'self';
```

- la page peut être chargée dans un cadre seulement si l'`Origin` appelante est *a.fr* ou *b.fr* :

```
Content-Security-Policy: default-src 'self'; frame-ancestors a.fr b.fr;
```

Attention

La directive `frame-ancestors` n'est pas supportée par l'implémentation en balise HTML de CSP. La protection contre le *clickjacking* avec CSP implique sa mise en œuvre au moyen de l'en-tête HTTP dédié.

Pour des raisons de compatibilité avec certains navigateurs, il est aussi possible de recourir à l'en-tête non standard `X-Frame-Options`, rendu obsolète par CSP, qui permet de préciser la stratégie d'affichage des contenus du site.

Exemple

- `X-Frame-Options: deny` est équivalent à `frame-ancestors 'none';`
- `X-Frame-Options: sameorigin` est équivalent à `frame-ancestors 'self';`
- `X-Frame-Options: allow-from https://site.fr` est équivalent à `frame-ancestors https://site.fr;`
- contrairement à CSP, `X-Frame-Options` n'accepte qu'un seul paramètre pour la directive `allow-from`.

R17

Utiliser CSP contre le clickjacking

Pour l'intégralité d'un site ou au minimum pour les pages présentant une sensibilité particulière (changement de mot de passe, connexion, virements, etc.), il est recommandé de mettre en place une protection contre le détournement de clic en définissant l'attribut CSP `frame-ancestors` à une liste d'autorisations minimale.

R18

Utiliser X-Frame-Options contre le clickjacking

Au titre de la défense en profondeur, il est recommandé de mettre en place une protection complémentaire contre le détournement de clic en définissant un en-tête

`X-Frame-Options` strict.

5.3.6 Collecte des rapports de violation

La directive `report-uri` de CSP permet de définir une URL de collecte des rapports de violations à une stratégie CSP. Lorsqu'elle est définie, le navigateur d'un visiteur enverra au `report-uri` des informations complémentaires en cas de blocage d'une non-conformité à la CSP.

Exemple

Exemples de définition d'une URL de collecte :

- CSP niveau 2, utilisation de la directive `report-uri` :

```
Content-Security-Policy: default-src: 'self'; report-uri: https://csp.my.fr;
```

- CSP niveau 3[17], : utilisation de la *Reporting API* [24] à la place de `report-uri`, rendu obsolète :

```
1 Reporting-Endpoints: my-csp-endpoint="https://csp.my.fr";
2 Content-Security-Policy: default-src: 'self'; report-to: my-csp-endpoint;
```

Lors de l'élaboration d'une stratégie CSP, il est possible d'utiliser l'en-tête `Content-Security-Policy-Report-Only` à la place de `Content-Security-Policy`, pour activer l'envoi de rapports sans imposer de blocage des ressources côté navigateur. Cet en-tête est utile lorsque l'on cherche à définir une CSP sur un site existant, afin de localiser et corriger les non-conformités sans provoquer l'interruption du service.

Si la collecte de ces rapports est utile dans le cadre de la mise en place de CSP, l'idée de s'en servir comme une mesure de détection des tentatives de XSS doit être accompagnée d'un certain nombre de précautions. En effet, la compromission du *endpoint* renseigné, tout comme l'interception du trafic généré par l'envoi des rapports, peuvent dégrader la sécurité de l'application web :

- Considérations relatives à la sécurité :
  - un rapport CSP contient des informations précises qui peuvent indiquer la présence d'une vulnérabilité sur le site. Ce risque augmente avec l'utilisation de l'expression `'report-sample'`, qui augmente la verbosité du rapport ;
  - le standard de *reporting* [24] soulève également le problème de sécurité lié à la fuite d'informations sur les *Capability URLs* [9], pages pour lesquelles on préférera désactiver l'envoi de rapports. En effet, de telles URLs peuvent contenir des jetons dont on souhaite préserver la confidentialité (par exemple, un jeton de remise à zéro d'un mot de passe) ;
- Considérations relatives à la confidentialité :
  - Un rapport CSP contient des données relatives à la navigation de l'utilisateur, telles que l'URL de la page visitée et son `Referer`, comme cela sera abordé à la section 5.4 ;
- Enfin, la définition d'un point d'entrée supplémentaire augmente la surface d'attaque de l'application web. Il n'existe pas de mécanisme permettant de garantir l'authenticité d'un rapport reçu. Le système de collecte des rapports constitue donc une ressource qu'il faut héberger, maîtriser

17. CSP niveau 3 est, à l'heure actuelle, un brouillon et non une recommandation.

et sécuriser au même titre que le site principal. L'utilisation d'un domaine et d'un serveur dédiés et distincts de ceux du site principal permet de limiter la contagion en cas de déni de service ou de vulnérabilité applicative.

R19

Etudier les risques liés à la collecte de rapports CSP

Lors de la mise en œuvre d'un système de collecte des rapports CSP, il est recommandé d'étudier la sensibilité des informations présentes dans chaque page du site et de procéder au cas par cas à l'activation de la fonctionnalité en fonction du risque envisagé.

5.3.7 Maîtrise des requêtes silencieuses

Certaines fonctionnalités de la spécification HTML [23] permettent de demander au navigateur d'émettre des requêtes silencieuses sans passer par l'exécution de code `JavaScript` ou CSS. Comme tout comportement qui conduit le navigateur d'une victime à initier une connexion de manière silencieuse, ces requêtes sont potentiellement indésirables et présentent des risques allant de la fuite d'informations jusqu'à l'exploitation de failles CSRF en passant par la réalisation d'attaques par déni de service distribué (`DDoS`).

Un premier exemple de ce genre de fonctionnalités est l'attribut `ping`. Une balise HTML `<a>` peut, si celle-ci présente un attribut `href`, comporter un attribut `ping` en complément. L'attribut `ping` contient alors une liste d'URLs vers lesquelles seront réalisées des requêtes `POST` lorsque le lien sera cliqué. Les URLs définies par l'attribut `ping` peuvent se situer en dehors de l'`Origin` et son utilisation relève généralement du *tracking* publicitaire.

L'attribut `ping` est implémenté par le navigateur et ne met pas en jeu l'utilisation de `JavaScript` pour émettre des requêtes silencieuses. Sa mise en œuvre augmente donc le risque CSRF, parce que les contre-mesures qui visent à interdire la transmission d'un `cookie` de session lors d'une requête issue d'un `<script>` ne s'appliquent pas. Par ailleurs, `ping` permet de démultiplier efficacement le nombre de requêtes émises lors du clic sur un lien, et de piéger des victimes afin de distribuer une attaque par déni de service, puisque la cible du lien peut être légitime sans que la présence ou la valeur de l'attribut `ping` soit visible par un utilisateur non spécialiste.

Exemple

Dans les exemples suivants, les lignes 7 et 8 vont afficher un lien piégé qui tentera une attaque CSRF sur *banque.fr*, tandis que les lignes 9 et 10, proposent un lien avec un attribut `ping` qui effectuera l'effacement d'un article sur le site actuellement visité si l'utilisateur (victime) en a l'autorisation.

```
6  <body>
7   <a href="/actualites"
8      ping="http://banque.fr/virement?vers=bob">Liens vers les actualités</a> |
9   <a href="/meteo"
10     ping="/article/123/delete">Liens vers la météo</a>
11 ...
```

Listing 5.9 – Exemples de liens avec l'attribut `ping`

Un autre exemple dans lequel la spécification HTML met en jeu l'utilisation de requêtes silencieuses est le standard *Resource Hints*. Il définit les attributs *dns-prefetch*, *preconnect*, *prefetch* et *prerender*, utilisables au sein d'une balise `<link>` ou de l'en-tête HTTP `Link` équivalent. Ce standard vise à optimiser la performance en lançant en avance de phase des connexions potentiellement utiles par la suite (chargement d'une image, d'un document, résolution d'un nom de domaine, etc.). Le développeur peut ainsi déclarer au navigateur son intention de charger du contenu avant son affichage effectif. Là encore, ce genre de mécanisme peut être détourné pour demander au navigateur d'une victime d'initier des connexions vers les cibles d'une attaque, sans pour autant que celles-ci correspondent à du réel contenu à afficher par la suite et sans recourir au *scripting*.

Exemple

Dans les exemples suivants, la ligne 4 va initier une connexion à l'avance vers un CDN afin que, lorsqu'une ressource y fera appel, son chargement soit accéléré. La ligne 5 demande au navigateur de télécharger, avec une faible priorité, une image dont l'apparition est jugée probable.

```
3 <head>
4   <link rel="preconnect" href="//mon-cdn.fr">
5   <link rel="prefetch" href="/image-lourde.bmp" as="image">
```

Listing 5.10 – Exemples de l'utilisation de *Resource Hints*

CSP peut être utilisé afin de définir une stratégie qui maîtrise les domaines vers lesquels le navigateur peut effectuer une connexion. CSP peut donc être mis en œuvre afin de limiter les mauvais usages qui peuvent être faits de l'attribut `ping` ou des *Resource Hints* sur les `Origins` en dehors de la liste d'autorisations. Ici aussi, on préférera la mise en place par en-tête HTTP à la méthode par balise `<meta>`, notamment pour traiter le cas de la déclaration de ressources via l'en-tête `Link`.

Exemple

Exemples de CSP qui limitent l'impact des requêtes silencieuses :

- utilisation de la directive `connect-src` pour `<a> ping` entre autres :

```
Content-Security-Policy: default-src 'self' a.fr; connect-src 'self';
```

- utilisation de la directive `prefetch-src` pour les `Resource Hints` :

```
Content-Security-Policy: default-src 'self' a.fr; prefetch-src 'self';
```

- en l'absence de directive spécifique, `default-src` aura le même effet :

```
Content-Security-Policy: default-src 'self';
```

R20

Réduire l'impact des requêtes silencieuses via CSP

Il est recommandé de définir une CSP limitant les `Origins` atteignables par le navigateur dans le but de bloquer l'émission de requêtes silencieuses difficiles à maîtriser à cause de la nature de la spécification HTML.

Information

- Depuis sa version 5, la spécification HTML est un *living standard* [23], c'est-à-dire un standard évolutif non figé, qui peut suivre ou précéder les implémentations des différents acteurs du Web. Par exemple, si le standard prévoit que les requêtes issues de `ping` ne devraient pas être silencieuses, les implémentations actuelles le sont. Cela renforce l'utilité des mesures de sécurité par liste d'autorisations exclusives telles que CSP, dont on peut espérer que les implémentations suivent l'évolution du standard, comme c'est le cas pour la directive `prefetch-src`, ajoutée en réponse aux *Resource Hints*.
- En complément de CSP, une spécification en cours de rédaction, appelée *Permissions Policy* [18], vise à offrir un contrôle comparable aux permissions que l'on trouve aujourd'hui sur les applications mobiles, mais pour les APIs du navigateur.

5.4 Mise en œuvre de Referrer-Policy

5.4.1 Introduction à Referrer-Policy

Pour chaque requête émise, le navigateur positionne automatiquement l'en-tête `Referer`, qui indique l'URL de la ressource depuis laquelle l'URL cible a été obtenue. Cela a lieu lors de la navigation d'une page web à une autre (ex. : clic sur un lien), mais aussi lors du chargement de ressources (ex. : images, scripts, etc.) depuis une page web existante.

Il peut être utile pour un site web de récupérer l'URL de la page web précédemment visitée pour l'analyse statistique, l'optimisation du cache, la journalisation et d'autres raisons légitimes. Si le standard empêche l'émission de l'en-tête `Referer` d'un contexte HTTPS vers un contexte HTTP, ce comportement par défaut peut tout de même nuire à la confidentialité comme le montre l'exemple suivant :

Exemple

Problème de confidentialité dans la stratégie par défaut de `Referrer-Policy`.
Si l'on considère le serveur HTTPS du site `www.site.fr` ne définissant pas de `Referrer-Policy` et la page chargée à partir de l'URL :
`https://www.site.fr/main?login=jdoe&email=john.doe@here.com`

```
<!DOCTYPE html>
<html lang="fr">
<head>
  <meta charset="utf-8">
  <link rel="stylesheet"
    href="https://cdn.bootstrap.com/bootstrap/3.3.7/css/bootstrap.min.css">
  <title>Sample page</title>
</head>
<body>
<a href="https://www.autre-site.fr/accueil.html">Autre site</a>
</body>
</html>
```

Listing 5.11 – Exemple de page sans `Referrer-Policy`

La valeur de l'en-tête `Referer`
(`https://www.site.fr/main.html?login=jdoe&email=john.doe@here.com`)
pourra être récupérée par :

> - `https://cdn.bootstrap.com` lors du `GET` sur le fichier CSS ;
> - `https://www.autre-site.fr/accueil.html` si l'utilisateur clique sur le lien ;
> - un `JavaScript` s'exécutant sur la page `https://www.autre-site.fr` via l'expression `document.referrer`.
>
> Cela est d'autant plus préoccupant lorsque le `Referer` est une *Capability URL* [9].

Le standard *Referrer Policy* [15] permet la définition d' une stratégie de fonctionnement du `Referer` [18] au niveau :

- du serveur HTTP par la définition de l'en-tête `Referrer-Policy` ou
- au sein de chaque page web via la balise HTML `<meta> referrer`.

La spécification présente plusieurs options de configuration :

`Referrer-Policy`	
Option	**Stratégie appliquée**
`no-referrer`	Aucune information n'est transmise via l'en-tête `Referer`.
`no-referrer-when-downgrade`	L'URL complète est utilisée en tant que `Referer` sauf dans le cas du passage de HTTPS à HTTP. Remarque : il s'agit du comportement par défaut du standard en l'absence de `Referrer-Policy`.
`origin`	L'`Origin` est utilisée en tant que `Referer` et non l'URL complète.
`same-origin`	Aucune information n'est transmise via l'en-tête `Referer`, sauf dans le cas d'accès au site courant où l'URL complète est utilisée.
`strict-origin`	L'`Origin` est utilisée en tant que `Referer` uniquement lorsque la communication s'effectue vers une destination de sécurité au moins équivalente ou supérieure : HTTP vers HTTP, HTTP vers HTTPS ou HTTPS vers HTTPS.
`origin-when-cross-origin`	L'URL complète est utilisée en tant que `Referer` pour le site courant, l'`Origin` est utilisée pour les autres cas.
`strict-origin-when-cross-origin`	L'URL complète est utilisée en tant que `Referer` pour le site courant, l'`Origin` est utilisée pour les autres cas à la condition que la sécurité de la destination soit au moins équivalente ou supérieure (cf. `strict-origin`). Dans le cas contraire aucun `Referer` ne sera transmis. Remarque : il s'agit du comportement par défaut dans certaines implémentations navigateur.
`unsafe-url`	L'URL complète est utilisée en tant que `Referer`, y compris lors de la navigation d'une page HTTPS vers une page HTTP.

18. À la suite d'une erreur dans l'orthographe du mot "referrer" lors de la rédaction de la RFC 1945 (HTTP/1.0) `https://tools.ietf.org/html/rfc1945#section-10.13` , l'en-tête HTTP utilise le mot "Referer".

R21

Définir la stratégie de construction de l'en-tête Referer

Il est recommandé de définir une stratégie de construction de l'en-tête de requête HTTP `Referer` au travers de l'en-tête de réponse HTTP `Referrer-Policy`.
La stratégie de construction de l'en-tête `Referer` par défaut ne doit pas être conservée et l'option `unsafe-url` ne doit pas être utilisée.

Attention

Avant la parution du standard *Referrer Policy*, la spécification CSP proposait une directive `referrer` qui offrait des capacités similaires. Celle-ci est désormais obsolète, non standard et n'est plus implémentée par les navigateurs.

Information

Quelle que soit la configuration du `Referrer-Policy`, l'en-tête `Referer` ne présente jamais les informations potentiellement sensibles d'une *Uniform Resource Locator* (URL) que sont le `#fragment` et les authentifiants qui peuvent précéder le nom de domaine. Par exemple, `https://user:password@domain.com/path?query#fragment` donne `https://domain.com/path?query`.

5.4.2 Modification ponctuelle du Referrer-Policy

Il peut arriver que la stratégie de composition de l'en-tête `Referer` appliquée à un site ou à une page ne convienne pas à certains liens vers des sites externes. Par exemple, si l'auteur d'un article ne souhaite pas indiquer au site destination qu'il fait une référence à celui-ci, il peut définir l'attribut `referrerpolicy` sur les liens sortants.

L'attribut `referrerpolicy` peut avoir pour valeur les différentes stratégies présentées dans la section 5.4.1 et s'applique à un certain nombre d'éléments permettant le chargement de ressources : `<a>`, `<area>`, `<img>`, `<iframe>` et `<link>`.

Exemple

```
1 <a href="http://blog.autre-site.fr/news/23" referrerpolicy="origin">Article</a>
```

Listing 5.12 – Utilisation de l'attribut `referrerpolicy`

Dans cet exemple, la modification du `referrerpolicy` sur le lien est destinée à limiter la diffusion de l'information de `Referer` à l'`Origin` pour empêcher la fuite de l'URL complète vers l'article en cours. Il est également possible d'empêcher le navigateur de positionner l'en-tête `Referer` par la définition de l'attribut de relation (`rel`) d'un élément à la valeur `noreferrer`. Cependant, cet attribut ne permet pas de définir d'autres stratégies de `Referer` et ne s'applique que sur les balises `<a>`, `<link>` et `<area>`.

R22

Modifier ponctuellement l'en-tête Referer

Il est recommandé d'utiliser l'attribut `referrerpolicy` afin de définir des modifications de stratégie de `Referrer` sur des éléments spécifiques.

Attention

La balise `script` ne gère pas l'attribut `referrerpolicy`. En revanche, les requêtes issues de code `JavaScript` peuvent se voir attribuer une stratégie de `Referrer` au cas par cas en utilisant l'`API Fetch` (cf. section 5.6.3).

5.5 Mise en œuvre des Web Storage, IndexedDB et Cookies

5.5.1 Précaution d'usage des bases de données de type Web Storage

Les bases de stockage de données hors-ligne `localStorage` et `sessionStorage` sont des bases de données de type clé / valeur permettant l'enregistrement de données côté client par `Origin` respectivement persistantes et non persistantes. Elles font partie du standard HTML [23].

Les bases de données locales respectent la contrainte de *Same-Origin Policy*. Les données manipulées au sein d'une `Origin` ne sont visibles et accessibles que par cette `Origin`. Il s'agit cependant de la seule mesure implémentée permettant le contrôle d'accès à ces données. Au sein d'une même `Origin`, tous les scripts `JavaScript` accèdent aux mêmes bases `localStorage` et `sessionStorage`.

Exemple

- la page d'authentification d'un site web peut accéder aux données du `localStorage` de la connexion précédente ;
- un code `JavaScript` injecté peut accéder aux données du `sessionStorage` même si l'utilisateur a mis fin à sa session.

R23

Ne pas stocker des informations sensibles dans les bases de données locales

Les bases de données (dites hors ligne) `localStorage` et `sessionStorage` ne doivent être utilisées que pour le stockage de données non sensibles et pour lesquelles la perte ou la divulgation sera sans conséquence, telles que les préférences utilisateur.

R23 -

Éviter de stocker des informations sensibles dans les bases de données locales

La décision du stockage d'informations sensibles dans les bases de données locales doit être prise en compte dans le cadre d'une analyse des risques, du fait de la faiblesse du mécanisme de contrôle d'accès à ces bases.

5.5.2 Précaution d'usage des bases de données de type IndexedDB

Les bases de données `IndexedDB` [16] sont des bases d'enregistrements persistants permettant la recherche et le stockage clé / objet dans le navigateur de l'utilisateur. À la différence de `LocalStorage` et `SessionStorage`, `IndexedDB` expose une API asynchrone et un modèle transactionnel.

Les bases de données `IndexedDB` respectent la contrainte de *Same-Origin Policy*. Les données manipulées au sein d'une `Origin` ne sont visibles et accessibles que par cette `Origin`.

Information

À la différence des `cookies` et des bases de données *Web Storage* (`LocalStorage` et `SessionStorage`), il est possible d'utiliser l'API `IndexedDB` dans un *Web Worker*. L'`Origin` de rattachement considérée pour la base de données sera alors l'`Origin` ayant instanciée le *Web Worker*, c'est d'ailleurs l'un des moyens de transmettre des données depuis et vers un *Web Worker*.

Le contrôle d'accès y étant identique, les bases de données `IndexedDB` présentent les mêmes risques que les bases de données locales.

R24

Ne pas stocker des informations sensibles dans les bases de données IndexedDB

Les bases de données `IndexedDB` ne doivent être utilisées que pour le stockage de données non sensibles et pour lesquelles la perte ou la divulgation sera sans conséquence, pour mettre en cache l'état d'une application web par exemple.

R24 -

Éviter de stocker des informations sensibles dans les bases de données IndexedDB

La décision du stockage d'informations sensibles dans les bases de données IndexedDB doit être prise en compte dans le cadre d'une analyse des risques, du fait de la faiblesse du mécanisme de contrôle d'accès à ces bases.

Une autre déclinaison des APIs vues ici, nommée *Web SQL Database* [5], permettait l'interrogation en SQL d'une base locale au navigateur. Son utilisation est aujourd'hui à proscrire car la spécification n'est plus maintenue et son implémentation disparaît des navigateurs.

R25

Proscrire l'usage de l'API Web SQL Database

Interdire l'usage de l'API *Web SQL Database*, désormais obsolète.

5.5.3 Précaution d'usage des cookies

Les `cookies` [17] permettent de conserver des informations sur le navigateur pour une durée déterminée. Le volume d'un `cookie` est très réduit : 0 à 4 ko. C'est un couple clé / valeur émis entre le

navigateur et le serveur lors de chaque cycle requête / réponse au moyen des en-têtes HTTP `Cookie` (requête cliente) et `Set-Cookie` (réponse serveur). Cela fait du `cookie` un bon moyen pour maintenir un état entre un client donné et le serveur tout en reposant sur un protocole sous-jacent sans état, comme HTTP. Les `cookies` sont souvent la cible des attaquants, qui visent à modifier ou récupérer cet état.

R26

Ne pas stocker d'informations sensibles dans les cookies

Dans le cadre de la défense en profondeur et à l'exception des jetons de session, il est recommandé de ne pas stocker des informations sensibles dans les `cookies`. Leur utilisation n'est souhaitable que pour le stockage temporaire d'informations de faible volume, pour lesquelles la perte ou la divulgation sera sans conséquence.

Sous réserve de l'application des recommandations qui vont suivre, le `cookie` est un bon candidat pour le stockage des jetons de session. À l'inverse des bases de données côté navigateur vues précédemment, les `cookies` disposent d'attributs de cloisonnement dédiés à cet usage et qui visent à limiter leur exposition. L'en-tête `Cookie` est automatiquement positionné par le navigateur lors de l'émission de requêtes en fonction des attributs de chaque `cookie`, définis via `Set-Cookie` :

- `Domain` : domaine parent vers lequel le `cookie` sera envoyé. Implicitement, déclarer cet attribut autorise l'envoi du `cookie` vers les sous-domaines également. Si cet attribut est vide, il prendra la valeur du domaine ou sous-domaine depuis lequel il a été émis ;
- `Path` : préfixe des chemins vers lesquels le `cookie` sera envoyé. Si cet attribut est vide, il prendra la valeur « `/` » et sera envoyé peu importe le chemin ;
- `Max-Age` : anciennement `Expires`, durée de validité du `cookie` en secondes. Si cet attribut est vide, il s'agit d'un `cookie` de session navigateur, dont la vie s'arrête à la fermeture du navigateur (sauf mécanismes de restauration de session) ;
- `HttpOnly` : si mentionné, le `cookie` ne sera pas accessible depuis un contexte `JavaScript` ;
- `Secure` : si mentionné, le `cookie` ne sera envoyé que via un transport sécurisé HTTPS. Le standard est en cours d'évolution sur ce point, si bien que les navigateurs modernes empêchent également un site en HTTP de déclarer ou modifier des `cookies` avec l'attribut `secure`. Pour plus de maîtrise sur la protection en écriture des `cookies`, voir *Cookie Prefixes* [11] ;
- `SameSite` : stratégie d'envoi du `cookie` en *cross-site*, contribue à la protection contre les attaques CSRF. Trois valeurs sont possibles : `Strict` pour que le `cookie` ne soit envoyé que dans un contexte *same-site*, `None` pour que cet attribut ne soit pas pris en compte lors de la composition de la stratégie d'envoi, `Lax` pour que le `cookie` soit envoyé en *same-site* et lors des requêtes `cross-origin` considérées sûres [19]. Selon les implémentations, l'absence de cet attribut sera équivalente à `Lax` (navigateurs modernes) ou à `None` (navigateurs anciens).

Les `cookies` obéissent à une stratégie de contrôle différente de la *Same-Origin Policy*, reposant sur les attributs ci-dessus. Notamment, si héberger deux sites ayant le même nom de domaine sur deux ports différents garantit une forme d'isolation au regard de la *Same-Origin Policy*, cela ne sera pas le cas pour les `cookies`.

19. Cela concerne les requêtes de navigation (donc hors chargement de ressources externes type images) dites « sûres » au sens de la RFC 7231, c'est-à-dire dont l'émission ne devrait pas changer l'état de l'application comme peuvent le faire, par convention, les verbes HTTP `POST`, `PUT`, `DELETE` par exemple. Plus d'informations dans le standard [17].

Exemple

Pour mieux comprendre la notion de *same-site* introduite par l'attribut éponyme, voici une illustration des différences entre *same-origin*, *same-site* et *cross-site* :

- `https://www.site.fr/articles` et `https://www.site.fr/contact` ont le même triplet *protocole*, *hôte* et, implicitement, *port*. Ces URL sont *same-origin* ;
- `https://blog.site.fr/` et `https://forum.site.fr/` ont le même *protocole* et le même *Top-Level Domain* (TLD). Ces URL sont *same-site*, non *same-origin* ;
- `https://www.site.fr/` et `https://www.exemple.fr` sont *cross-site*, et donc `cross-origin` également.

R27

Cloisonner les sessions au moyen de noms de domaine distincts

Afin d'éviter qu'un `cookie` ne soit envoyé par correspondance involontaire sur l'attribut `Domain` avec le domaine ou sous-domaine en question, il est recommandé de répartir les périmètres de responsabilité d'une application web sur des domaines différents.

Exemple

Scénario de correspondance involontaire sur l'attribut `Domain` d'un `cookie` :

- L'administrateur d'un *Content Management System* (CMS) se rend sur `https://cms.fr/admin` afin de rédiger un article. Un `cookie` de session est positionné lorsqu'il se connecte :

```
Set-Cookie: Domain=cms.fr; sessionId=abc123; Secure; HttpOnly; SameSite=Lax
```

- `https://cms.fr/` fournit également un service de proxy web, dont le périmètre de responsabilité est différent, accessible sur `https://cms.fr/webproxy` ;
- Lorsque l'administrateur visite `https://cms.fr/webproxy?url=externe.fr`, son `cookie` de session permettant l'administration du CMS sera envoyé par le navigateur, par correspondance sur l'attribut `Domain` ;
- En revanche, puisque les périmètres de responsabilité sont différents, il est possible que l'administrateur du service de proxy web ait décidé de transférer l'ensemble des en-têtes HTTP, pour des raisons de compatibilité ;
- Dans ce cas, l'administrateur du site `externe.fr` recevrait le `cookie` et serait en mesure d'usurper l'identité de l'administrateur du CMS sur `cms.fr`.

Les CMS ne proposant pas nécessairement la définition de restrictions sur l'attribut `Path` du `cookie`, une bonne pratique ici serait d'héberger l'accès en administration du contenu au sein d'un sous-domaine distinct, tel que `https://admin.cms.fr/`, ce qui aura pour effet de limiter l'exposition du `cookie` ainsi que d'activer les mécanismes d'isolation garantis par la SOP.

Information

En général, ne pas spécifier l'attribut `Domain` est une bonne pratique lors de la définition d'un `cookie`, parce que le navigateur va automatiquement le positionner au

sous-domaine émetteur et non à un domaine parent.

Dans le cas de l'hébergement de plusieurs applications sur un même site web, la valeur du `path` doit être définie strictement afin de limiter la visibilité des `cookies` des applications les unes par rapport aux autres. Notons que cela offre une protection amoindrie vis-à-vis de la séparation par noms de domaine, parce que la *Same-Origin Policy* ne vérifie pas le `path`.

R28

Définir le path d'un cookie

Il est recommandé de restreindre la portée des `cookies` en suivant le principe de moindre privilège. Le `path` de chaque `cookie` doit être ajusté au découpage hiérarchique du site web et à la sensibilité du `cookie`.

Exemple

Exemple d'ajustement du `path` au découpage hiérarchique du site :

- Un site peut définir, pour tous ses utilisateurs, un `cookie` de session global, ayant pour `path` « `/` » ;
- Pour accéder aux pages relatives à l'administration fonctionnelle du site, l'administrateur se verra positionner un cookie supplémentaire, ayant le `path` « `/admin` », ce qui limitera l'exposition d'une session privilégiée lors de la consultation de pages de moindre confiance au sein du même domaine.

Attention

Il ne faut pas considérer l'usage des attributs `path` et `domain` comme une mesure suffisante contre la lecture. En effet tout code `JavaScript` s'exécutant dans le même contexte que la page mais provenant d'une URL différente du site d'origine sera autorisé à lire tous les `cookies` présents, ce qui peut aboutir à un vol de session.

R29

Maîtriser l'accès aux cookies en JavaScript

Dès lors qu'un `cookie` n'a d'usage que pour le serveur d'applications ou n'a pas la nécessité d'être traité par un code exécuté sur le navigateur, l'attribut `HttpOnly` doit être utilisé afin de limiter le risque de vol par un code `JavaScript`.

R30

Proscrire l'accès en JavaScript à un cookie de session

Pour un `cookie` de session, il est nécessaire de positionner l'attribut `HttpOnly`.

L'attribut `Secure` permet d'autoriser l'envoi du `cookie` uniquement lorsque la communication avec le serveur est considérée sécurisée (HTTPS). Ce point permet de limiter le vol de `cookie` (voire de session) et assure qu'un `cookie` transmis sur un canal sécurisé ne transitera pas ultérieurement sur un canal non sécurisé (HTTP), y compris involontairement lors d'une redirection par exemple.

R31

Limiter le transit des cookies aux flux sécurisés

Dès lors que des `cookies` sont nécessaires et que le site ou l'application n'est accessible qu'en HTTPS, le flag `Secure` doit être utilisé.

Enfin, l'attribut `SameSite` permet de réduire drastiquement la vulnérabilité aux attaques CSRF, dans la mesure où il permet d'indiquer au navigateur dans quelles conditions il est acceptable d'émettre un `cookie` en fonction du contexte de navigation en cours.

R32

Définir une stratégie stricte d'envoi des cookies en cross-site

Dès qu'un `cookie` n'a pas de raison d'être émis lors de la navigation depuis un site web extérieur, définir l'attribut `SameSite` à `Strict`. Dans le cas contraire, utiliser la valeur `Lax` si le `cookie` n'autorise pas d'action privilégiée via la méthode HTTP `GET`.

R33

Définir une stratégie stricte d'envoi des cookies de session en cross-site

Pour un `cookie` de session, l'attribut `SameSite` doit être défini et ne doit pas être positionné à `None`.

Au même titre qu'une donnée de formulaire ou un paramètre de requête, un `cookie`, comme la plupart des en-têtes HTTP, est à considérer comme une entrée maîtrisée par l'utilisateur et doit suivre le même circuit de validation. Notamment, un sous-domaine corrompu peut définir un `cookie` sur un domaine voisin ou parent, il faut donc vérifier la cohérence de l'information transmise par le client et, dans le cas d'un `cookie` de session, forcer une ré-authentification en cas de doute. Il existe contre-mesures standard, implémentées par les navigateurs, qui offrent une protection en écriture sur les `cookies`, à savoir les *Cookie Prefixes* [11] et la *Public Suffix List* [20].

Attention

Tous les moyens de stockage d'information côté navigateur abordés dans cette section ont une persistance limitée dans la mesure où les actions utilisateur du type « *effacer les données de navigation* » vont vider, entre autres, `LocalStorage`, `SessionStorage`, et les `cookies`. Il convient donc de considérer ces informations comme perdables et de proposer à l'utilisateur un moyen de les récupérer ou de les renouveler. Un mécanisme d'effacement similaire, *Clear Site Data* [14], est également en cours de standardisation et expose ces fonctionnalités aux développeurs d'applications web.

Information

Si le `cookie` est une solution robuste de maintien d'une session entre un utilisateur et un service web, l'établissement de cette session nécessite une étape d'authentification qui met généralement en jeu un couple *login* / mot de passe ou un fédérateur d'identité, dont les implémentations sont variées. Le standard *Credential Management API* [20] ainsi que son extension *WebAuthn* [26] proposent de doter les navigateurs

20. Un navigateur refusera d'enregistrer un `cookie` dont la portée est globale à un domaine inscrit dans cette liste : `https://publicsuffix.org/`.

web d'une interface unifiée qui encadre la création, l'utilisation, le stockage et plus généralement l'interaction avec des authentifiants utilisateurs tels que des mots de passe, ou des biclés dans le cas de *WebAuthn*.

5.6 XMLHttpRequest (XHR) et Cross-Origin Resource Sharing (CORS)

REpresentational State Transfer (REST) est une approche sémantique du protocole HTTP fréquemment employée pour la mise en œuvre de *WebServices*. Souvent utilisée en combinaison avec le format JSON, REST est une solution répandue d'échanges de données structurées entre un client `JavaScript` et un serveur.

L'API `JavaScript` `XMLHttpRequest` (XHR, [27]) permet d'émettre des requêtes HTTP à partir d'une page web s'exécutant dans un navigateur. Le nom de cette classe est trompeur, car celle-ci n'est pas limitée à la récupération de données XML. Le contenu texte d'une réponse à ce type de requête est généralement analysé et transformé dans le format attendu (généralement JSON ou XML).

Exemple

```
function processData(data) {
  var temperature = data.temp + '°C';
  var pression    = data.pressure + 'millibars';
  //...
}
function handler() {
  if (this.status === 200 && this.responseText !== null) {
    try{  processData(JSON.parse(this.responseText)); }
    catch(e) { displayErrorMessage(e); }
  } else { displayStatus(this.status); }
}
var client = new XMLHttpRequest();
client.onload = handler;
client.open('GET', '/api/city/Paris');
client.send();
```

Listing 5.13 – Exemple XHR - Conditions météo sur Paris

La fonction `open` peut comprendre les paramètres suivants :

- la méthode HTTP à utiliser ;
- l'URL destinataire de la requête ;
- si la requête doit être faite en synchrone (`false`) ou asynchrone (`true`).

Information

`JavaScript` est un langage dans lequel il est possible d'omettre les paramètres d'une fonction. Il est fréquent de voir la fonction `open` avec seulement 2 paramètres. Cette méthode s'exécutant sur le fil d'exécution principal (sauf *worker*), la rendre bloquante (ou synchrone) aurait pour conséquence de figer la page. Il n'est pas erroné d'utiliser la méthode `open` avec seulement 2 paramètres car elle sera asynchrone par défaut.

5.6.1 Utilisation de XMLHttpRequest (XHR)

Placer un contenu HTML dans une réponse `XMLHttpRequest` sous-entend une utilisation de la méthode `JavaScript` `innerHTML` pour inclure ce fragment dans la page courante. Cette pratique permet l'injection de code `JavaScript` (vulnérabilité XSS). Il est donc préférable d'utiliser un format d'encodage et non un fragment HTML.

R34

Encoder les réponses XMLHttpRequest

Le contenu d'une réponse de requête XHR doit être formaté par le serveur sous un format de données non exécutable par le client (ex. : JSON ou XML).

Exemple

En exemple de vulnérabilité XSS avec XHR, certaines bibliothèques `JavaScript` comportent des fonctions d'autocomplétion qui consomment un service web de suggestions pour les afficher sous forme de liste. Ces fonctions attendent une réponse au format liste HTML composée par le serveur (`<ul><li>un</li><li>deux</li><li>etc.</li></ul>`).
Cette réponse préformatée est directement incluse au sein de la page HTML au moyen de la fonction `innerHTML`, qui est un *sink* (cf. section 5.2.1) permettant l'injection de code et donc la modification du comportement de la page par un service web corrompu.

Le choix de la méthode HTTP d'une API n'est pas neutre même dans le cas d'un site ou d'une application sécurisée par TLS. L'exposition des données transférées dépend du verbe HTTP utilisé.

R35

Choisir une API selon sa méthode HTTP

Il est recommandé de vérifier que le niveau de confidentialité de la donnée manipulée est compatible avec la méthode HTTP proposée par l'API. Dans le cas contraire, il est recommandé de ne pas utiliser l'API proposée et de demander son évolution à son concepteur.

La méthode `GET` véhicule les données des formulaires directement dans l'URL, cette dernière est conservée dans l'historique du navigateur, dans les journaux (*logs*) des serveurs d'applications, mais aussi dans ceux de tous les *proxies* d'interception, *reverse-proxies* ou autres points de terminaison TLS traversés par la requête. En outre, les réponses à ce type de requêtes sont sujettes à la mise en cache par les équipements. L'usage de la méthode `GET` n'est donc pas adapté à toutes les situations.

R36 -

Utiliser XHR avec la méthode GET sous certaines conditions

Les requêtes XHR `GET` peuvent être utilisées uniquement :

- si elles comportent seulement des données publiques dans leurs URLs ;
- à des fins de récupération de données non sensibles (conservables dans un cache) ;
- à la condition de ne provoquer aucun traitement persistant ou changement d'état côté serveur (idempotence).

La méthode POST, en l'absence de données dans l'URL, ne présente pas les défauts de confidentialité et de mise en cache de la méthode GET.

R36

Utiliser XHR avec la méthode POST

Il est recommandé d'utiliser la méthode POST pour les requêtes XHR pour éviter les risques de fuite de données.

La méthode PUT n'a pas les inconvénients de la méthode GET et présente un intérêt de sécurité par rapport à la méthode POST. En effet, le *preflight* (ou test préalable, voir section 5.6.2) est systématique dans le cas d'un appel en *cross-origin* avec la méthode PUT. Cela limite le risque de fuite des données et la vulnérabilité au CSRF, sauf dans le cas d'une configuration CORS explicitement trop permissive.

R36 +

Utiliser XHR avec la méthode PUT

Afin de limiter le risque CSRF et de contrôler le risque de fuite d'information, il est recommandé d'utiliser la méthode PUT afin de limiter le risque CSRF et contrôler le risque de fuite d'informations lors d'appels XHR.

R37

Compléter la mise œuvre de XHR par une configuration CSP

Afin de limiter le risque d'exfiltration de données réalisable par un attaquant qui serait parvenu à remplacer les XHR par des appels CORS valides, il est recommandé de mettre en œuvre une stratégie *Content Security Policy* bloquant les appels XHR en dehors de l'Origin.

Exemple

Limiter les appels XHR en dehors de l'Origin :

```
Content-Security-Policy: default-src 'self' a.fr; connect-src 'self';
```

Information

La directive connect-src ne se limite pas à la définition de restrictions uniquement pour XHR, mais va contrôler les domaines accessibles par un certain nombre d'API similaires : XHR, Fetch, EventSource, WebSocket. En l'absence de directive spécifique connect-src, ces appels seront tout de même protégés par la directive default-src.

Les appels XHR peuvent, au même titre que les autres requêtes HTTP, véhiculer des attaques CSRF.

R38

Protéger les appels XHR par un contrôle anti-CSRF

Il est recommandé d'ajouter un contrôle anti-CSRF aux appels XHR à l'aide d'un CSRF-Token. Celui-ci doit contenir une valeur aléatoire générée à l'aide d'une fonction utilisant un générateur d'aléa cryptographique et ayant une entropie minimale de 128 bits. Cette taille peut être atteinte en générant aléatoirement une chaîne de 22 caractères ASCII imprimables (A à Z, a à z et 0 à 9).

De multiples solutions sont possibles pour communiquer le `CSRF-Token` à la partie cliente :

- placement dans l'en-tête dédié d'une réponse HTTP ;
- placement dans un `meta` tag HTML.

5.6.2 Fonctionnement de Cross-Origin Resource Sharing (CORS)

Cross-Origin Resource Sharing (CORS) est un standard du Web [22] permettant de dépasser la contrainte de *Same-Origin Policy* par l'établissement d'un contrat entre le serveur de destination et le navigateur, négocié via des en-têtes HTTP.

CORS est à mettre en œuvre en remplacement des techniques précédemment utilisées :

- JSON-P : cette technique est doublement risquée car elle combine l'évaluation de code et le passage de paramètres par l'URL via un `GET`, ce qui induit souvent une vulnérabilité XSS ;
- Proxyfication XHR par le serveur : cette technique a pour conséquence de transmettre les `cookies` et autres en-têtes du client (`Authorization`, `X-Auth-Token`) au domaine destinaire.

CORS concerne les requêtes silencieuses *cross-origin*, c'est-à-dire les requêtes XHR, `Fetch`, ou autre mécanisme d'émission de requête silencieuse dont le contenu peut être lu en `JavaScript` et dont l'`Origin` de l'URL destination est différente de l'`Origin` de la page.

CORS classe les appels XHR en 2 catégories (voir annexe A) :

- les appels par « méthodes simples avec en-têtes simples », qui correspondent généralement aux requêtes émissibles sans l'aide de `JavaScript` ;
- et les autres appels.

Si la requête est catégorisée « méthode simple avec en-têtes simples », elle sera envoyée inconditionnellement et ce n'est qu'à la réception de la réponse HTTP que le navigateur décidera si son contenu est accessible ou non, d'où un risque de fuite de données. Cette décision se fait sur la base des en-têtes CORS suivants, présents dans la réponse :

- `Access-Control-Allow-Origin` et `Access-Control-Allow-Methods`.

Si la requête n'entre pas dans la catégorie « méthode simple avec en-têtes simples », une requête HTTP préalable, nommée `preflight` dans le standard, sera envoyée au serveur de destination. Cette requête préalable utilise la méthode HTTP `OPTIONS`. Elle précise la méthode et les en-têtes particuliers qui seront utilisés par la requête réelle à venir au moyen des en-têtes CORS suivants :

- `Access-Control-Request-Method` et `Access-Control-Request-Headers`.

À la réception de la réponse HTTP au `preflight`, le navigateur la met en cache et décide s'il envoie ou non la requête véritable en fonction du contenu des en-têtes reçus :

- `Access-Control-Allow-Origin`, `Access-Control-Allow-Methods`, `Access-Control-Allow-Headers`, `Access-Control-Allow-Credentials`.

Si ces en-têtes de réponse ne correspondent pas aux caractéristiques de la requête qui a requis un `preflight`, l'échange *cross-origin* sera annulé et tracé en erreur par le navigateur. Les paramètres potentiellement sensibles présents dans la requête initiale n'auront jamais été transmis par le navigateur.

Dans un souci de confidentialité, il est important de vérifier que le niveau de sensibilité de la donnée manipulée est compatible avec l'API CORS proposée (la manipulation de données sensibles devrait requérir un `preflight`). Dans le cas contraire, il est recommandé de ne pas utiliser l'API CORS proposée et de demander son évolution.

R39

Mettre en œuvre un preflight lors des appels CORS

Si les données transmises par un appel CORS présentent un caractère sensible, il est recommandé qu'un `preflight` soit prévu côté serveur et forcé côté client afin de limiter le risque de fuite d'informations. Un `preflight` peut être forcé par la présence, à vérifier, d'un en-tête non standard dans chaque requête CORS.

Information

Les appels CORS mettant en œuvre la méthode `POST` et un `Content-Type application/json` impliquent un `preflight`, de même que tout appel avec la méthode `PUT` ou tout appel nécessitant une authentification.

R40

Vérifier la valeur de l'Origin lors de la réception d'une requête CORS

L'en-tête `Origin`, dont la falsification est empêchée par le navigateur, doit être contrôlé par l'application avec une liste d'`Origins` autorisées pour réduire le risque CSRF via CORS.

Attention

L'utilisation du caractère générique « `*` » comme valeur pour l'en-tête de réponse `Access-Control-Allow-Origin` est dangereuse dans le cas d'une application d'intranet. En effet, cela revient à autoriser tout site web à requêter cette application, et permet donc l'utilisation du navigateur d'un employé comme relais pour attaquer une application interne depuis internet.
Pour cette raison entre autres, une ressource HTTP qui déclare supporter des requêtes authentifiées (`Access-Control-Allow-Credentials` à `true`) ne peut pas positionner `Access-Control-Allow-Origin` à « `*` », car cela fera échouer le `preflight`.
Une requête authentifiée désigne ici une requête qui comporte des identifiants utilisateur, sous la forme de `cookies`, en-tête `Authorization`, ou certificat client.

La granularité du décloisonnement offert par CORS se situe, encore une fois, au niveau de l'`Origin`. Il n'est donc pas possible d'héberger au sein de la même `Origin` deux applications à cloisonner, puisque si l'une met en place CORS, cela affectera également l'autre.

R41

Cloisonner les services web au moyen de noms de domaines distincts

Lors de la mise en place de plusieurs `WebServices` indépendants, il est recommandé de dédier un domaine à chacun d'entre eux.

Il n'est pas rare que les sites mettant en place des appels CORS proposent au téléchargement les bibliothèques clientes `JavaScript` simplifiant leur utilisation. Comme pour toute inclusion de bibliothèques externes dans une application ou un site web, il conviendra d'être très prudent avant

de les utiliser. En effet, toute bibliothèque `JavaScript` présente dans les ressources d'une page web peut accéder aux éléments de celle-ci. L'objet ici étant de dépasser, via CORS, la contrainte de *Same-Origin Policy*, le risque d'exfiltration de données par l'inclusion d'une bibliothèque non maîtrisée est à considérer.

R42

Éviter l'usage de bibliothèques publiques effectuant des appels CORS.

Une bibliothèque `JavaScript` dont le code est obscurci afin de bloquer son analyse, mais effectuant des appels CORS ne doit pas être incluse dans les ressources d'une application web.

R42 -

Isoler l'utilisation de bibliothèques publiques effectuant des appels CORS.

A défaut de pouvoir contrôler le code `JavaScript` d'une bibliothèque effectuant un appel CORS, celle-ci doit être isolée du reste de l'application via un *Web Worker* ou, à défaut, une `iframe`.

En complément du déclenchement automatique de CORS dans les conditions évoquées ci-dessus, ce mécanisme peut être explicitement invoqué lors du chargement de certaines ressources via l'utilisation de l'attribut HTML `crossorigin`. Cet attribut, valide sur les balises de média, de styles et de scripts, active CORS pour la ressource et permet d'exercer un contrôle explicite sur l'envoi des données d'authentification lors de la récupération de la ressource. En effet, même en présence d'un en-tête `Access-Control-Allow-Credentials` à `true` lors de la récupération d'une ressource, il est possible de maintenir la confidentialité de l'utilisateur en fixant la valeur de l'attribut `crossorigin` à `anonymous`. Inversement, pour permettre l'accès en `JavaScript` à une ressource qui requiert une authentification, la valeur `use-credentials` peut être utilisée pour `crossorigin`.

R43

Anonymiser le chargement des ressources en cross-origin

Dans le but de limiter l'exposition des authentifiants et pour préserver la confidentialité des utilisateurs, il est recommandé de positionner l'attribut `crossorigin` à `anonymous` pour les ressources dont la récupération ne nécessite pas d'authentification.

Exemple

Exemple de chargement anonyme d'une ressource :

```
<img src="https://trackme-cdn.com/image.png" crossorigin="anonymous" />
```

5.6.3 Utilisation de l'API Fetch

L'API `Fetch` [22] se présente comme une alternative plus flexible à l'utilisation de XHR notamment par l'utilisation de promesses (*Promises*) `JavaScript`.

Exemple

```
const url = '/api/ville/Paris';
const params = {
  method: 'GET',
  mode: 'same-origin',
  credentials: 'omit',
  cache: 'default',
  referrerPolicy: 'no-referrer',
  redirect: 'error',
  integrity: 'sha256-abcdef1234567890'
};

fetch(url, params)
  .then( res => res.json() )
  .then( data => handleMeteoData(data) );

// avec async/await, équivalent à :

(async () => {
  const meteoDataRaw = await fetch(url, params);
  const meteoDataJson = await meteoDataRaw.json();
  handleMeteoData(meteoDataJson);
})();
```

Listing 5.14 – Exemple API Fetch - Conditions météo sur Paris

Le second paramètre de la méthode `fetch` constitue la configuration de la requête à réaliser.

L'option `mode` permet de préciser le type attendu de la requête. Les valeurs possibles de `mode` et les comportements associés sont :

- **no-cors :** contrôle que la requête est catégorisée « méthode simple avec en-têtes simples » ;
- **cors :** active le `preflight` si nécessaire et contrôle les en-têtes nécessaires ;
- **same-origin** contrôle que la requête est effectuée sur la même `Origin`.

L'option `credentials` permet de contrôler l'émission de `cookies`, certificats clients, et autres données d'authentification HTTP. Par défaut, sa valeur est `same-origin`. Les autres valeurs possibles sont `omit` et `include`.

L'option `redirect` permet de décrire le fonctionnement à adopter si l'URL cible propose une redirection (301, 302, 303, 307 ou 308).

L'API `Fetch` offre une bonne séparation des usages avec les objets distincts *Headers*, *Request* et *Response*. Elle facilite la manipulation de données de plus bas niveau (ex. `blobs`), et l'utilisation des API *Cache*, *Referrer Policy* ou encore SRI, déjà évoquées dans ce document. `Fetch` présente un avantage en termes de simplicité et de sécurité par rapport à XHR car elle permet au développeur de spécifier, par configuration, son contexte d'utilisation légitime.

Exemple

Un appel déclaré avec l'option `"mode: same-origin"` ne sera pas envoyé si l'URL cible ne correspond pas à l'`Origin` appelante.

La réponse d'un appel déclaré avec l'option `"mode: no-cors"` ne sera acceptée que

si la méthode HTTP est « simple » : `GET`, `HEAD` et si l'appel n'est effectivement pas un appel CORS.

R44

Préférer l'utilisation de l'API Fetch à XMLHttpRequest

Dans la mesure du possible, l'utilisation de l'API `Fetch` est recommandée par rapport à `XMLHttpRequest`.

Information

- L'ensemble des recommandations abordées à la section 5.6.1 sont aussi applicables dans le cadre de l'utilisation de l'API `Fetch`.
- Des travaux sont en cours pour automatiser l'ajout de métadonnées par le navigateur lors de l'émission de requêtes. Le standard *Fetch Metadata Request Headers* [21] introduit quatre nouveaux en-têtes HTTP, complémentaires à `Origin`, qui offrent au serveur des informations enrichies sur le contexte d'émission de la requête afin d'en évaluer la légitimité avec plus de certitude en pré-traitement.

5.7 HTML5 et JavaScript

L'omniprésence de `JavaScript` dans des pages web de plus en plus complexes a contribué à une mise en œuvre massive de fonctionnalités qui demandent au navigateur de faire l'interface entre un nombre croissant de sites, pages, et composants de niveaux de confiance et de maîtrise hétérogènes. Cette section aborde les bonnes pratiques qui permettent d'exercer un contrôle plus fin sur ces interfaces et de mieux les comprendre lorsqu'elles sont abstraites par un *framework*.

5.7.1 Précaution dans l'ouverture de fenêtres

Il est commun d'utiliser l'attribut `target` afin de faire ouvrir une page dans une nouvelle fenêtre ou un nouvel onglet sans fermer la page parente. Son utilisation sans complément réalise l'instanciation d'une nouvelle page en conservant une affiliation avec la page initiale. L'expression `window.opener` donne à la page appelée une référence vers la page appelante.

Dans le cas de l'utilisation de l'attribut `target="_blank"` sur un lien vers une page dans le même site, l'accès à la référence `window.opener` est équivalent à la capacité d'exécuter du code `JavaScript` arbitraire sur la page appelante. En revanche, au titre de la *Same-Origin Policy*, si le lien mène vers un site différent, des restrictions s'appliquent et la plupart des objets du DOM parent ne seront pas accessibles. Cependant, même dans le cas *cross-site*, un site potentiellement malveillant peut modifier l'attribut `location` de la référence `window.opener` et ainsi remplacer silencieusement l'URL du site appelant par un site contrefait. Une autre exception à la *Same-Origin Policy* dans ce cadre est la possibilité d'utiliser l'API `postMessage` pour récupérer et émettre des informations depuis et vers la page appelante `window.opener`. La valeur `noopener` de l'attribut de relation `rel` a été introduite pour empêcher toute communication avec la page appelante.

R45

Sécuriser l'ouverture de nouvelles fenêtres

Il est recommandé de sécuriser systématiquement l'ouverture de nouvelles fenêtres :

- en HTML par l'utilisation de l'attribut `rel="noopener"` dès lors que l'attribut `target` est utilisé pour un lien ;
- en `JavaScript` par l'utilisation de l'option `noopener` dès lors que le second paramètre `target` est utilisé dans la commande `window.open`.

Exemple

```
<a href="page.html" target="onglet1" rel="noopener">Lien noopener</a><br>
<button id="bt">Bouton noopener</button>
```

```
document.getElementById("bt").onclick = () =>
  window.open('page.html', 'onglet1', 'noopener');
```

Listing 5.15 – Mise en œuvre de `noopener`

Information

- Les mises en œuvre légitimes de `window.opener`, telles que les séquences d'approbation `OAuth`[21]qui mettent en jeu des *pop-ups*, doivent être implémentées dans des pages minimalistes dédiées à cet usage, qui ne présentent que les liens hypertexte strictement nécessaires ;
- L'utilisation de la valeur `noreferrer` pour l'attribut `rel` désactive aussi implicitement la référence `window.opener`.

Si l'attribut `noopener` est à l'initiative de l'appelant et garantit une protection lors de l'ouverture d'un site tiers dans une nouvelle fenêtre, une mesure de sécurité similaire existe et traite également le cas inverse, c'est-à-dire lorsque l'application web à isoler est ouverte par un site tiers. L'en-tête de réponse HTTP `Cross-Origin-Opener-Policy` permet de contrôler l'isolation du contexte de navigation à la fois pour les navigations sortantes (ouverture d'une fenêtre vers un tiers) et entrantes (ouverture dans une fenêtre depuis un tiers). Les contrôles d'accès et l'isolation mémoire induits par cet en-tête sont cruciaux dans le cas de la définition d'une API *Web Messaging* (`postMessage`) en écoute globale sur la page (ex. : `window.addEventListener('message')`).

R46

Définir une stratégie d'ouverture en cross-origin

Il est recommandé de définir une stratégie *Cross-Origin-Opener-Policy* stricte afin de garantir l'isolation du contexte de navigation lors de l'ouverture de fenêtres : `Cross-Origin-Opener-Policy : same-origin`

21. `OAuth` est un protocole d'autorisation dans lequel l'utilisateur consent à partager des informations via des boîtes de dialogue du type « *Autorisez-vous l'application X à accéder aux données Y de l'application Z ?* ».

5.7.2 Sécurité des développements JavaScript

5.7.2.1 Utilisation du mode strict

Le mode `strict` [22] de `JavaScript` permet de désactiver certains comportements jugés trop souples lors de l'interprétation, dans des domaines tels que la syntaxe ou encore les types. Une fois le mode `strict` activé, toute erreur ignorée dans le mode par défaut sera considérée comme une exception et interrompra le fil d'exécution `JavaScript`. Le mode `strict` oblige le développeur `JavaScript` à plus de rigueur et désactive également les fonctionnalités en voie d'obsolescence, dont l'implémentation n'est valable qu'au titre de la rétrocompatibilité. Il peut être conservé en phase de production afin de contrarier une tentative d'injection de code `JavaScript` (vulnérabilité XSS).

Ce mode se déclare au niveau d'un fichier ou au niveau d'une fonction par l'ajout de la chaîne de caractères `"use strict";` [23]. Dans les deux cas, cette chaîne de caractères doit être la première directive rencontrée. Attention, dans le cas de la déclaration au niveau du fichier, l'utilisation d'outils de minification de code peut rendre la directive globale à tous les fichiers. Au contraire, si les outils de génération de code ne positionnent pas `"use strict";` au début du fichier généré mais après d'autres instructions, la directive sera ignorée pour l'ensemble du fichier.

Exemple

```
// déclaration au niveau du fichier
"use strict";
// contexte strict
function chuck(parms) {
  // contexte strict également
}
```

```
// déclaration au niveau d'une fonction
// contexte par défaut
function strictFunc(params) {
  "use strict";
  // contexte strict
}
```

R47

Utiliser le mode strict

Il est recommandé de déclarer l'utilisation du mode `strict` en utilisant la directive `"use strict";` au début de chaque fonction `JavaScript`. Pour couvrir l'ensemble du code, l'utilisation de fonctions auto-invoquées est préférable à la mise en œuvre du mode `strict` au niveau du fichier.

Information

Lors des développements, il peut être intéressant d'utiliser un langage de programmation qui offre un comportement par défaut plus strict que `JavaScript` au niveau du typage et de la syntaxe afin de limiter les erreurs humaines. Attention, comparablement à ce que peuvent apporter des messages d'avertissement lors de la compi-

22. https://www.ecma-international.org/ecma-262/5.1/#sec-14.1
23. Attention la concaténation n'est pas acceptée. `"use " + "strict";` ou l'utilisation d'une variable ne fonctionnera pas.

lation d'un binaire, les vérifications supplémentaires introduites par ces langages ne seront plus présentes au *runtime* si le code a été transpilé en `JavaScript` pour mise en production.
Exemples : `TypeScript`, `CoffeeScript`, `Dart`.

5.7.2.2 Utilisation de Tag sur les Template Strings d'ES6

Les *Template Strings*, ou *Template Literals* [24] , sont des littéraux ou gabarits de chaînes de caractères qui permettent d'intégrer des expressions. La spécification ECMAScript 2015 (ES6) propose l'utilisation d'un `tag` afin de spécifier un comportement lors de l'interprétation des expressions d'un *template string*. Le `tag` est une fonction `JavaScript` utilisée en préfixe du *template* et invoquée lors de l'utilisation de celui-ci.

Si les *Template Strings* sont par nature vulnérables au XSS (voir exemple 5.2.1), il est cependant possible de se protéger en utilisant un `tag` et sa fonction associée.

Exemple

Protection d'un *Template String* ES6 contre une vulnérabilité XSS dans un contexte HTML (fonction d'échappement volontairement succincte, donnée à titre d'exemple uniquement) :

```
function safeTag(strings, ...values) {
  var out = "";
  for (i in strings) {
    out += strings[i];
    if (values[i]) {
      out += values[i].replace(/</g, "&lt;")
                      .replace(/>/g, "&gt;")
                      .replace(/&/g, "&amp;");
    }
  }
  return out;
}

const badVar1 = '<img src="ok" onerror="alert(1)"/>bad';

const myStringTemplate = safeTag`
Bonjour ${badVar1} ! <br>
Le multiligne est très <b>pratique</b>.
`;

document.body.innerHTML = myStringTemplate;
// sans safeTag, le navigateur afficherait l'alert(1)
```

Listing 5.16 – Exemple d'implémentation de la recommandation R7

5.7.3 Techniques de cloisonnement JavaScript

Lors de la composition d'une page web, il est de plus en plus fréquent de faire appel à des ressources et de consommer des services web mis à disposition par des tierces parties. Le cloisonnement de code `JavaScript` permet d'utiliser un composant non maîtrisé en isolant celui-ci du reste de l'application.

24. http://www.ecma-international.org/ecma-262/6.0/#sec-template-literal-lexical-components.

5.7.3.1 Mise en œuvre des Web Workers

Un *Web Worker* est un fragment `JavaScript`, issu de la même `Origin` que la page courante, mais s'exécutant en tâche de fond et dans un contexte différent (*event loop*, *stack* et *execution context* distincts). Il s'agit d'une API de *multithreading* côté navigateur dont les contraintes de conception en font un bon outil d'isolation entre deux contextes de navigation, notamment en *same-origin*. Un *Web Worker* s'exécute dans un environnement contraint ne lui permettant pas d'accéder au DOM. En contrepartie, le fil d'exécution principal de la page ne sera pas bloqué ou ralenti par un traitement effectué au sein d'un *Web Worker*. Celui-ci peut toutefois mettre en œuvre des requêtes XHR et `Fetch`, des `EventSources`, des `WebSockets` et utiliser des bases de données `IndexedDB`. Les *Web Workers* font partie du standard HTML [23].

L'API `postMessage` permet le transit de messages, avec ou sans sérialisation, entre le *worker* et son contexte d'instanciation. Une variante des *Web Workers*, appelée *Shared Workers*, permet également l'utilisation de cette API pour autoriser différents contextes de navigation à échanger des informations avec un même *Shared Worker*. Par exemple, deux onglets issus de la même `Origin` peuvent s'adresser au même *Shared Worker*.

Exemple

```
var colors = ['red', 'grey', 'blue', 'orange', 'yellow', 'white'];
function loop() {
  var n = Math.floor(Math.random() * colors.length);
  var msg = { color: colors[n] };
  postMessage(JSON.stringify(msg));
  setTimeout(loop, 2000);
}
loop();
```

Listing 5.17 – Fragment d'un Web Worker JavaScript : worker.js

```
<h1 id="color">Web Worker</h1>
<button id="bt">Run/Stop</button>
<script src="worker-wrapper.js"></script>
```

Listing 5.18 – HTML appelant

```
var worker = null;
function start() {
  worker = new Worker("worker.js");
  worker.addEventListener('message', (event) => {
    var data = JSON.parse(event.data);
    document.body.style.background = data.color;
    document.getElementById('color').textContent = data.color;
  });
}
function stop() {
  worker.terminate();
  worker = null;
}
document.getElementById('bt')
  .addEventListener('click', () => (worker === null) ? start() : stop());
```

Listing 5.19 – Instanciation du worker : worker-wrapper.js

Les *Web Workers* constituent un élément important de la défense en profondeur en permettant le confinement de code `JavaScript` dans un contexte dédié, sans accès au DOM, aux `cookies` et

stockages locaux (`LocalStorage` et `SessionStorage`), ce qui en réduit la surface d'attaque.

R48

Isoler les traitements par Web Workers

Les traitements utilisant des ressources `JavaScript` non maîtrisées, externes au site ou des appels CORS doivent si possible être isolés dans des *Web Workers* afin d'empêcher leurs accès aux DOM, `cookies`, `localStorage` et `sessionStorage` et de réduire leur impact sur le fil d'exécution principal.

Un *Web Worker* peut charger des ressources `JavaScript` externes.

Exemple

```
importScripts("weather-libs.js");
```

Listing 5.20 – Chargement d'une ressource JavaScript dans un Web Worker

Attention

- La fonction `importScripts` permet le chargement de ressources d'`Origins` différentes, et la mise en œuvre de SRI à l'intérieur d'un *Web Worker* n'est pas possible. L'intégrité d'une ressource `JavaScript` utilisée à l'intérieur d'un *Web Worker* ne peut donc pas être contrôlée via ce mécanisme.
- Les requêtes silencieuses émises par un *worker* seront, par défaut, émises en provenance de l'`Origin` de la page parente. Elles ne bénéficient donc d'aucun cloisonnement au titre de la SOP.

R48 +

Isoler les traitements par Web Worker et Origin « data : »

Il est recommandé d'instancier un *worker* de faible confiance à l'aide d'une URL de type « `data:` » dans le but d'isoler celui-ci de l'`Origin` de confiance.

Information

Les *Web Workers* autorisent l'utilisation d'`IndexedDB`, ce qui en fait un vecteur de contamination possible d'un *worker* non sûr vers des contextes jugés maîtrisés mais dont l'exécution dépend d'informations issues de l'`IndexedDB`. La recommandation R48+ permet un cloisonnement de l'`IndexedDB` et des activités réseau du *worker* au titre de la *Same-Origin Policy*.

Exemple

```
let worker;
fetch('worker.js')
.then( res => res.text() )
.then( txt =>
  worker = new Worker('data:text/plain;charset=utf-8;base64,' + btoa(txt))
).catch( errorHandler )
```

Listing 5.21 – Instanciation d'un worker avec origin « `data:` » distincte

Dans le cadre de l'utilisation d'un *Web Worker* pour isoler un contenu de moindre confiance, la spécification d'une interface de communication stricte avec le *worker* permet de réduire l'impact d'une vulnérabilité dans son contenu sur la page appelante.

R49

Formaliser les échanges en utilisant l'API de Message

Il est recommandé de définir avec précision les formats des messages acceptés en provenance d'un *worker*. L'API *Web Messaging* (`postMessage`) est à favoriser par rapport à l'utilisation de l'`IndexedDB`, potentiellement polluée. La mise en œuvre du format JSON et des fonctions associées `JSON.parse` et `JSON.stringify` est recommandée afin d'éviter que les échanges de messages ne débouchent sur de l'exécution de code.

Attention

Certains outils `JavaScript` très répandus ne peuvent pas être utilisés à l'intérieur d'un *Web Worker*. Lorsque l'accès au DOM est indispensable, l'isolation du traitement peut passer par des `iframes`.
Par exemple, `JQuery` réalise un accès au DOM immédiatement après avoir été téléchargé. Le confinement d'un code `JavaScript` utilisant `JQuery` ne peut donc pas être réalisé par un *Web Worker*.

5.7.3.2 Mise en œuvre des iframes

Une `iframe` est un morceau de code HTML imbriqué dans la page HTML courante. Les utilisations de la balise `<iframe>` sont très nombreuses :

- inclusion de contenu externe (publicité, carte, vidéo, liens de partages, etc.) ;
- présentation de contenu segmenté (carte météo, code avec prévisualisation, etc.) ;
- expérience de navigation sans rechargement de la page principale.

L'intérêt des `iframes` est de permettre la présentation de contenus dont on ne maîtrise pas le code, tout en contrôlant les interactions avec le contexte de la page courante.

À la différence d'un *Web Worker*, une `iframe` peut directement charger du contenu en provenance d'`Origins` arbitraires et manipuler l'API DOM, ce qui en augmente la surface d'attaque par rapport à un *worker*. Une `iframe` est un contexte de navigation aussi riche qu'un onglet ou qu'une fenêtre, qui possède son propre DOM, contexte d'exécution `JavaScript` indépendant (*event loop*, *stack* et *execution context* distincts), et accède aux stockages locaux propres à sa source. L'exécution d'une `iframe` est considérée sans impact sur les performances de la page parente.

R50

Cloisonner les traitements dans des iframes

Les traitements utilisant des ressources externes non maîtrisées, mais nécessitant la présence d'un DOM devraient être isolés dans une `iframe` afin d'interdire leurs accès aux DOM, `cookies`, `localStorage` et `sessionStorage` de la page parente.

L'attribut `sandbox` de l'élément `iframe` propose un certain nombre d'options permettant d'activer des autorisations de fonctionnement. La présence seule de l'attribut `sandbox` sur une `iframe` signifie que toutes les restrictions sont activées. Ce principe de liste d'autorisations permet aussi à

l'`iframe` de bénéficier automatiquement des évolutions du standard.
Ainsi, à partir d'une `iframe` dans une *sandbox*, il sera impossible :

- de soumettre un formulaire (dérogation via `allow-forms`) ;
- d'exécuter du code `JavaScript` (dérogation via `allow-scripts`) ;
- d'accéder aux éléments protégés par la SOP : `cookies`, `LocalStorage`, appel REST ou CORS (dérogation via `allow-same-origin`) ;
- d'ouvrir de nouvelles fenêtres (dérogation via `allow-popups`, `allow-modals`) ;
- d'obtenir des informations sur le déplacement de la souris (dérogation via `allow-pointer-lock`) ;
- d'accéder au DOM de la page parente (dérogation via `allow-top-navigation`).

D'autres restrictions peuvent voir le jour au fur et à mesure que le standard et les fonctionnalités du navigateur évoluent.

Exemple

Dans cet exemple, l'`iframe` est incluse dans la page courante sans utiliser l'attribut `sandbox` et présente une URL relative.

```
35 <iframe name="weatherFrame_same_origin"
36         src="weather-frame.html"
37         width="0" height="0"></iframe>
```

Listing 5.22 – Inclusion d'une `iframe` sans *sandbox* et issue de la même `Origin`

L'`Origin` de cette `iframe` est identique à celle de la page parente. Le `JavaScript` présent dans cette `iframe` peut avoir accès aux mêmes `cookies` et `LocalStorage` que la page parente. De plus, le DOM de la page parente est exposé à l'`iframe`. Dans cet exemple, il n'y a donc aucune isolation entre la page parente et l'`iframe`.

Exemple

L'attribut `sandbox` peut être utilisé pour cloisonner la page `weather-frame.html` de l'exemple précédent, pourtant chargée depuis la même `Origin` que la page appelante.

```
38 <iframe sandbox
39         name="weatherFrame_same_origin_sandboxed"
40         src="weather-frame.html"
41         width="0" height="0"></iframe>
```

Listing 5.23 – Inclusion d'une `iframe` avec *sandbox* et issue de la même `Origin`

En plus des restrictions en fonctionnalités imposées par la présence de l'attribut `sandbox`, son implémentation par le navigateur va résulter en l'attribution d'une `Origin` nulle pour cette `iframe`, ce qui aura pour effet d'activer toutes les contraintes relatives à la *Same-Origin Policy* comme si le contenu provenait d'une autre `Origin`.

Exemple

Dans cet exemple, l'`iframe` est incluse dans la page courante à partir d'un site différent de celui de la page parente. De plus l'attribut `sandbox` est présent avec les options `allow-scripts` et `allow-same-origin`.

```
42 <iframe sandbox="allow-scripts allow-same-origin"
43         name="weatherFrame_alien"
44         src="http://api.site.fr:8081/frames/weather-frame.html"
45         width="0" height="0"></iframe>
46 <script>
```

Listing 5.24 – Inclusion d'une `iframe` avec *sandbox* et issue d'une `Origin` différente

L'exécution de code `JavaScript` est autorisée pour cette `iframe`.

L'`Origin` de celle-ci étant différente de celle de la page parente. Le code `JavaScript` de l'`iframe` ne pourra pas accéder aux `cookies`, `LocalStorage` et DOM de la page parente.

Malgré l'option `allow-same-origin`, la page parente ne pourra pas accéder directement au DOM de l'`iframe` en raison des différences d'`Origin`.

L'option `allow-same-origin` est cependant nécessaire afin de pouvoir utiliser l'API `Message` en précisant directement l'`Origin` du destinataire et ainsi éviter l'utilisation de `weatherFrame_alien.postMessage(..., "*")`, ce qui aurait pour effet d'autoriser une `Origin` non maîtrisée à recevoir le message par inclusion de l'`iframe`.

Comme pour les *Web Workers*, la communication entre l'`iframe` et la fenêtre parente s'effectue par l'utilisation de l'API *Web Messaging* (`postMessage`).

Information

L'utilisation de l'API *Web Messaging* en précisant l'`Origin` de l'`iframe` destinataire implique les options `allow-same-origin` et `allow-scripts` sur l'attribut `sandbox`.

R51

Cloisonner les traitements avec une sandbox

Lors de l'utilisation d'une `iframe` à des fins de cloisonnement, il est recommandé de paramétrer l'attribut `sandbox`, qui permet une maîtrise accrue du confinement de l'`iframe`.

Dans le cas d'une `iframe` de même `Origin` que la page appelante, la définition des attributs `allow-scripts` et `allow-same-origin` simultanément octroie suffisamment de droits à l'`iframe` pour simplement retirer l'attribut HTML `sandbox`. En outre, les protections d'`Origin` offertes par une *sandbox* déclarée par la page appelante seront caduques si un attaquant arrive à charger le contenu en dehors de l'`iframe`. La déclaration d'une *sandbox* au moyen de CSP permet de traiter ce risque en retirant la dépendance sur le contexte appelant.

R52

Favoriser la déclaration de sandbox via CSP

Lorsque l'hébergement du contenu à isoler est maîtrisé, il est recommandé de mettre en place la *sandbox* via une stratégie *Content Security Policy* plutôt que par l'attribut HTML équivalent.

Une mesure de sécurite moins sujette aux erreurs de configuration consiste à héberger le contenu à isoler sur une `Origin` différente.

R52 +

Cloisonner les traitements par une iframe sur une seconde Origin

Lorsque cela est possible, il est recommandé de mettre en œuvre une seconde `Origin` destinée à héberger le traitement afin d'isoler celui-ci de l'`Origin` proposant du contenu.

Cette pratique consiste à mettre en œuvre pour un site une seconde `Origin` afin d'isoler le traitement du contenu et de profiter ainsi du cloisonnement offert par la *Same-Origin Policy* et par l'attribut `domain` d'un `cookie`. Notons que cette pratique est compatible avec la définition d'une *sandbox* complémentaire.

Un cas d'usage intéressant se présente lorsque la page courante utilise une `Origin` donnée et met en œuvre une `iframe` chargée à partir de la seconde `Origin`. Cette seconde `Origin` héberge le traitement. La communication entre la page parente et le traitement utilise l'API de messages et un format d'échange défini (exemple : JSON). Ce modèle revient à avoir un service web à la fois exposé et consommé par le navigateur.

Attention

Cette pratique n'est correcte qu'à la condition de mettre en place les mécanismes de contrôle des messages échangés entre l'`iframe` et la page parente.

R53

Formaliser les échanges en utilisant l'API de Message

Il est recommandé de définir avec précision les formats des messages acceptés en provenance d'une `iframe` via l'API *Web Messaging* (`postMessage`). La mise en œuvre du format JSON et des fonctions associées `JSON.parse` et `JSON.stringify` est recommandée afin d'éviter que les échanges de messages ne débouchent sur de l'exécution de code.

L'utilisation de l'API de messages entre `frames` revient à effectuer une dérogation explicite à la *Same-Origin Policy* pour activer la communication inter-domaines. L'ouverture de ce canal de communication expose le site ou l'application web à un certain nombre de risques :

- changement de référence d'une `iframe` ;
- blocage de la diffusion de messages ;
- interception et falsification de messages ;
- émission de messages d'`Origin` étrangère ;
- partage d'espace mémoire favorisant l'exploitation de vulnérabilités de type *Spectre* [25].

Par rapport à l'API accessible aux *Web Workers*, l'API de messages portée par l'objet global `window` prend un paramètre `origin`, qui permet de mieux maîtriser ces communications.

25. Plus d'informations sur les mécanismes d'isolation mémoire mis à disposition des développeurs web par les navigateurs : `https://w3c.github.io/webappsec-post-spectre-webdev/`

R54

Définir l'Origin lors de l'utilisation de l'API de Message

Il est recommandé d'invoquer la méthode `postMessage` sur une instance d'`iframe` en précisant l'`Origin` destinatrice.

R55

Contrôler l'Origin lors de l'utilisation de l'API de Message

Il est recommandé de contrôler l'`Origin` de l'émetteur ainsi que le format de message lors de la réception de message dans une `iframe`.

Le contrôle d'`Origin` en `JavaScript` doit être complété par des mesures plus bas niveau basées sur la définition d'en-têtes HTTP. La directive CSP `frame-ancestors`, introduite au paragraphe 5.3.5 dans le contexte du détournement de clic, est également importante dans le cas de la définition d'une API de messages. Elle permet de restreindre les origines autorisées à ouvrir la ressource dans une `iframe`, et donc à utiliser l'API `postMessage`. En complément, la directive `child-src` permet d'effectuer le contrôle inverse, vis-à-vis des `iframes` que l'application peut ouvrir. Si `frame-ancestors` constitue la première ligne de défense contre tout site malveillant essayant d'ouvrir des `iframes` vers les destinations de son choix, `child-src` se présente davantage comme une mesure de défense en profondeur, puisque l'initiative de l'ouverture est prise par l'application qui déclare la CSP. Elle maîtrise donc a priori, sauf vulnérabilité, compromission ou erreur de logique, les ressources qu'elle appelle.

R56

Compléter la déclaration d'une API de messages par la définition d'une CSP

Il est recommandé de compléter la déclaration d'une API de messages par la définition des directives CSP `frame-ancestors` et `child-src` visant à contrôler les contextes d'inclusion d'une ressource sous forme d'`iframe`.

De façon similaire à l'en-tête de sécurité `Cross-Origin-Opener-Policy` abordé à la section 5.7.1, les en-têtes HTTP `Cross-Origin-Embedder-Policy` et `Cross-Origin-Resource-Policy` permettent de déclarer dans quel contexte (*cross-origin* ou *same-origin*) une ressource donnée est autorisée à être embarquée. Cela permet également au navigateur de minimiser le partage d'espace mémoire entre les contextes (ex. : vulnérabilités de type *Spectre*).

Outre l'utilisation de l'API de messages entre deux `iframes` d'`Origins` différentes, il existe d'autres techniques de relâchement de la *Same-Origin Policy*. L'utilisation de certaines d'entre elles présente des risques inhérents.

Un premier exemple est le mécanisme de *domain relaxation*, qui permet à deux sous-domaines de se mettre d'accord pour faciliter les échanges entre eux en définissant la variable `JavaScript` `document.domain` à un domaine parent commun. Cependant, il s'agit d'une fonctionnalité historique qui offre peu de granularité et augmente le risque de fuite de données en cas de sous-domaine corrompu ou bien dans un contexte d'hébergement mutualisé. Ce mécanisme est en cours de retrait de la spécification HTML [23].

R57

Proscrire l'écriture de document.domain

Il est recommandé de proscrire l'utilisation du mécanisme de *domain relaxation*, c'est-à-dire de ne pas modifier la valeur de `document.domain`. Pour ces cas d'usage, le mécanisme à favoriser est l'API *Web Messaging* (`postMessage`) pour une communication sûre entre les `Origins`.

Un second exemple est une technique appelée *JSON with Padding* (JSON-P). Il s'agit d'une façon de charger dynamiquement du code `JavaScript` en *cross-origin* par l'utilisation de balises `<script>` dédiées au chargement de *callbacks*. Cela met généralement en jeu de la génération de `JavaScript` côté serveur, en fonction de paramètres de requête. L'authenticité de ce type de ressources dynamiques est difficile à maîtriser et une vulnérabilité dans leur contenu aboutit directement à de l'exécution de code. JSON-P est une astuce à considérer comme obsolète depuis l'implémentation de CORS par les navigateurs.

R58

Proscrire l'usage de JSON-P

Il est recommandé de proscrire l'utilisation de la technique JSON-P. La solution à privilégier pour consommer des ressources en *cross-origin* est le *Cross-Origin Resource Sharing* (CORS, cf. 5.6.2).

6
Maintien en conditions opérationnelle et de sécurité

Les vulnérabilités liées aux contenus d'un site web dépendent directement de l'obsolescence des composants mis en œuvre pour les générer, les manipuler ou les afficher. Un composant dont la mise à jour a été négligée peut devenir un vecteur d'attaque. Si la sécurité des contenus est l'objet de ce guide, la sécurisation de l'infrastructure nécessaire à leur mise à disposition a été abordée en détails dans d'autres publications, telles que les *Recommandations relatives à l'administration sécurisée des systèmes d'information [3]* et le guide d'*Externalisation et sécurité des systèmes d'information [1]*. Ce chapitre est dédié aux pratiques relatives à la maîtrise des contenus et composants, dans l'objectif de sécuriser leur mise en production et de faciliter leur maintien en conditions opérationnelles et de sécurité.

6.1 Maîtrise des contenus

Il est important que des informations sensibles, personnelles ou relatives au fonctionnement du site web n'apparaissent pas lorsqu'une erreur se produit. Il est fréquent que durant les phases de développement, soient créées des pages spécifiques de débogage permettant de retrouver rapidement une information et l'origine d'un problème.

L'utilisation de plusieurs profils d'exécution ou cibles de compilation permet de différencier le comportement de l'applicatif en fonction du contexte et d'adapter les messages d'erreurs éventuels au public cible. Ainsi le développeur, qui utilise le profil « `dev` » ou « `debug` », aura une pleine visibilité sur l'erreur et sa cause tandis que l'utilisateur final, qui utilise l'applicatif déployé en mode « `prod` » ou « `release` », se verra présenter un message d'aide ou une erreur simplifiée, qui seront des informations plus difficiles à exploiter dans le cadre d'une attaque.

R59

Définir des profils de déploiement spécifiques aux contextes

Il est recommandé de mettre en place une gestion des erreurs spécifique par profil d'utilisation. Ainsi, selon que l'application est en développement, test fonctionnel ou production, la gestion des erreurs sera différente et adaptée au contexte.

R60

Empêcher le déploiement d'un profil non adapté au contexte

L'application de la recommandation R59 doit s'accompagner de la mise en place de contrôles automatisés qui empêchent le déploiement d'un profil non durci (développement, test, etc.) dans un contexte de production.

6.2 Maîtrise des composants

Les sites web utilisent généralement un grand nombre de composants logiciels : système d'exploitation et serveur web mais aussi système de gestion de contenu et ses greffons, bibliothèques `Java`, `Node.js` ou `PHP`, système de gestion de base de données, modules ou extensions du serveur web, etc.

Les composants logiciels tiers (bibliothèques, greffons, *frameworks*, etc.) sont des briques offrant des fonctionnalités développées par des tierces parties. De nombreuses attaques sur les sites web aboutissent en exploitant des vulnérabilités connues dans les composants logiciels tiers et non dans le code spécifique au site. Un tel scénario est d'autant plus regrettable lorsque le composant exploité est présent mais non utilisé : il convient de réduire la surface d'attaque en supprimant par exemple les greffons inutiles des systèmes de gestion de contenu.

Exemple

Le CMS `WordPress` est livré en standard avec 3 thèmes et 2 *plugins* désactivés mais pré-installés. La présence de code inutilisé sur le serveur augmente la surface d'attaque.

Le serveur `Tomcat` est par défaut installé avec quelques applications d'exemple (`jsp-examples`, `servlet-examples`) et de gestion (`host-manager`, `manager`). Si la présence des applications d'exemple augmente la surface d'attaque du serveur, celle des applications de gestion augmente drastiquement les chances de succès d'une attaque.

R61

Limiter les composants logiciels tiers

La liste des composants applicatifs tiers employés doit être limitée au strict nécessaire. Les composants non nécessaires doivent faire l'objet d'une suppression. Si leur suppression n'est pas envisageable, il est recommandé de les désactiver.

R62

Maintenir à jour les composants logiciels tiers utilisés

Les composants applicatifs tiers employés doivent être recensés et maintenus à jour. Cela impose que les composants sélectionnés pour une production soient évalués sur leur pérennité lors des phases de conception et que les vulnérabilités publiées soient suivies pour chacun d'eux.

R63

Ne pas modifier le cœur des composants logiciels tiers utilisés

Pour faciliter leur mise à jour, il est recommandé de ne pas modifier le cœur des composants logiciels tiers utilisés. Toutes les modifications doivent se faire par des greffons ou par l'utilisation d'un composant adapté aux besoins.

Les produits de gestion de contenus web proposent pour la plupart la possiblité d'ajouter ou de réaliser des greffons (extensions ou *plugins*) afin d'améliorer le fonctionnement de tel ou tel point

du produit. Comme pour toute dépendance logicielle tierce, la mise en production d'un nouveau greffon ne doit être réalisée qu'à la condition d'avoir étudié et testé le greffon en question sur les quatre aspects suivants :

- fonctionnement : la dépendance fonctionne correctement, elle est compatible avec l'écosystème et la configuration en place, et apporte une vraie plus-value ;
- origine : la dépendance est connue et a une bonne réputation, le code est téléchargeable et référencé sur le site officiel de l'éditeur du CMS (greffon) ou site dédié (bibliothèque) ;
- sécurité : la dépendance ne présente pas de problèmes de sécurité connus et non résolus. La dépendance n'interfère pas avec les mécanismes de sécurité mis en place par l'application et son volume n'augmente pas outre mesure la surface d'attaque ;
- pérennité : la dépendance existe depuis plusieurs versions, continue d'évoluer et d'être mise à jour. Le greffon est maintenu et ne bloque pas longtemps la mise à jour du CMS. L'installation initiale et la mise à jour de la dépendance sont des opérations maîtrisées et facilement reproductibles.

Information

La prise en compte des quatre points précédents est valable pour un greffon dans le cas d'un CMS, mais aussi pour le choix d'une bibliothèque ou d'un *framework*. L'utilisation d'un *framework*, lorsque l'on comprend les mécanismes qu'il met en œuvre et que l'on maîtrise les interfaces qu'il fournit, permet de s'abstraire de certaines considérations de développement pour se concentrer sur la logique applicative. Par exemple, des *frameworks* plébiscités tels que *React*, *Angular* ou *Vue.js* [26] constituent une base solide d'abstractions documentées et, dans une certaine mesure, auditées par la communauté.

26. https://stateofjs.com/

Annexe A
Cas d'application du preflight CORS

Tout appel CORS ne respectant pas un ou plusieurs des critères suivants n'est pas considéré simple et se voit appliquer la stratégie du test préalable (requête `preflight`).

Requête avec méthode et en-tête simple		
Utilise l'une des 3 méthodes	**N'utilise que les en-têtes HTTP**	**Utilise l'un des Content-Type**
GET HEAD POST*	Cache-Control Content-Language Content-Type Expires Last-Modified Pragma	multipart/form-data application/x-www-form-urlencoded text/plain

* Le téléversement de fichiers par la méthode `POST` n'entre pas dans la catégorie des appels avec en-tête et méthode simple.

Exemples :

Méthode	**En-têtes présents**	**Content-Type**
POST	Content-Type, Cache-Control	application/x-www-form-urlencoded
Requête avec en-têtes et méthode simple		→ Pas de "Preflight"
GET	Cache-Control, Authorization	
Requête avec en-tête "Authorization"		→ "Preflight"
POST	Content-Type, Cache-Control	application/json
Requête avec Content-Type "application/json"		→ "Preflight"
POST	Content-Type, Cache-Control	multipart/form-data
Requête avec en-têtes et méthode simple		→ Pas de "Preflight"
POST	Content-Type, Cache-Control, X-Auth-Token	multipart/form-data
Requête avec en-tête "X-Auth-Token"		→ "Preflight"
PUT	Content-Type, Cache-Control	multipart/form-data
Requête avec méthode "PUT"		→ "Preflight"

Liste des recommandations

Bibliographie

[1] *Maîtriser les risques de l'infogérance. Externalisation des systèmes d'information*.
Guide Version 1.0, ANSSI, décembre 2010.
https://www.ssi.gouv.fr/infogerance.

[2] *Recommandations de sécurité pour la mise en œuvre d'un système de journalisation*.
Note technique DAT-NT-012/ANSSI/SDE/NP v1.0, ANSSI, décembre 2013.
https://www.ssi.gouv.fr/journalisation.

[3] *Recommandations relatives à l'administration sécurisée des systèmes d'information*.
Guide ANSSI-PA-022 v2.0, ANSSI, avril 2018.
https://www.ssi.gouv.fr/securisation-admin-si.

[4] *Recommandations de sécurité relatives à TLS*.
Guide ANSSI-PA-035 v1.2, ANSSI, mars 2020.
https://www.ssi.gouv.fr/nt-tls.

[5] *Standard Web SQL Database*.
Spécification obsolète, W3C, novembre 2010.
https://www.w3.org/TR/webdatabase/.

[6] *Talking to Yourself for Fun and Profit*.
Rapport de recherche, Huang L-S., Chen E., Barth A., Rescorla E., et Jackson C., mai 2011.
http://www.adambarth.com/papers/2011/huang-chen-barth-rescorla-jackson.pdf.

[7] *Standard HTTP Strict Transport Security*.
Rfc 6797, IETF, novembre 2012.
https://tools.ietf.org/html/rfc6797.

[8] *Standard Certificate Transparency*.
Rfc expérimentale 6962, IETF, juin 2013.
https://tools.ietf.org/html/rfc6962.

[9] *Bonnes pratiques pour les Capability URLs*.
Bonnes pratiques, W3C, février 2014.
https://www.w3.org/TR/capability-urls/.

[10] *Standard Content Security Policy Level 2*.
Recommandation, W3C, décembre 2016.
https://www.w3.org/TR/CSP2/.

[11] *Standard Cookie Prefixes*.
Brouillon pour rfc 6265, IETF, juin 2016.
https://tools.ietf.org/html/draft-west-cookie-prefixes/.

[12] *Standard Subresource Integrity*.
Recommandation, W3C, juin 2016.
https://www.w3.org/TR/SRI/.

[13] *Licence ouverte / Open Licence v2.0*.
Page web, Mission Etalab, 2017.
https://www.etalab.gouv.fr/licence-ouverte-open-licence.

[14] *Standard Clear Site Data*.
Brouillon de travail du w3c, W3C, novembre 2017.
https://www.w3.org/TR/clear-site-data/.

[15] *Standard Referrer Policy*.
Recommandaton candidate, W3C, janvier 2017.
https://www.w3.org/TR/referrer-policy/.

[16] *Standard Indexed Database API 2.0*.
Recommandation, W3C, janvier 2018.
https://www.w3.org/TR/IndexedDB/.

[17] *Standard Cookies*.
Brouillon pour rfc 6265, IETF, décembre 2020.
https://tools.ietf.org/html/draft-ietf-httpbis-rfc6265bis/.

[18] *Standard Permissions Policy (anciennement Feature Policy)*.
Brouillon de l'éditeur, W3C, décembre 2020.
https://w3c.github.io/webappsec-permissions-policy/.

[19] *Scénarios d'attaques XSS*.
Document communautaire, OWASP, mars 2021.
https://github.com/OWASP/www-community/blob/master/pages/xss-filter-evasion-cheatsheet.md.

[20] *Standard Credential Management Level 1*.
Brouillon de l'éditeur, W3C, février 2021.
https://w3c.github.io/webappsec-credential-management/.

[21] *Standard Fetch Metadata Request Headers*.
Brouillon de l'éditeur, W3C, mars 2021.
https://w3c.github.io/webappsec-fetch-metadata/.

[22] *Standard Fetch, protocole CORS*.
Standard évolutif, WHATWG, mars 2021.
https://fetch.spec.whatwg.org/#cors-protocol.

[23] *Standard HTML, Same-Origin, Web Storage, Web Workers*.
Standard évolutif, WHATWG, mars 2021.
https://html.spec.whatwg.org/.

[24] *Standard Reporting API*.
Brouillon de l'éditeur, W3C, février 2021.
https://w3c.github.io/reporting/.

[25] *Standard Trusted Types, intégrations*.
Brouillon de l'éditeur, W3C, mars 2021.
https://w3c.github.io/webappsec-trusted-types/dist/spec/#integrations.

[26] *Standard Web Authentication*.
Brouillon de l'éditeur, W3C, février 2021.
https://w3c.github.io/webauthn/.

[27] *Standard XMLHttpRequest*.
Standard évolutif, WHATWG, mars 2021.
https://xhr.spec.whatwg.org/.

ANSSI-PA-009
Version 2.0 - 28/04/2021

AGENCE NATIONALE DE LA SÉCURITÉ DES SYSTÈMES D'INFORMATION

ANSSI - 51, boulevard de La Tour-Maubourg, 75700 PARIS 07 SP
www.ssi.gouv.fr / conseil.technique@ssi.gouv.fr

Liberté • Égalité • Fraternité
RÉPUBLIQUE FRANÇAISE
Premier ministre
SGDSN