Implémentation de l'authentification à double facteur dans la solution de SSO AAA LemonLDAP::NG 2.0

Christophe Maudoux

==========

CONSERVATOIRE NATIONAL DES ARTS ET METIERS

PARIS

_____

MÉMOIRE

présenté en vue d'obtenir

le DIPLÔME d'INGÉNIEUR CNAM

SPÉCIALITÉ : INFORMATIQUE

OPTION : Réseaux, Systèmes et Multimédia

par

MAUDOUX, Christophe

_____

Implémentation de l'authentification à double facteur dans la solution de SSO AAA LemonLDAP::NG 2.0

Soutenu le 18/12/2018

_____

JURY

PRESIDENT : Monsieur Tristan Crolard

MEMBRES : Colonel Xavier Guimard – Madame Selma Boumerdassi
Lieutenant-Colonel Sébastien Hamel – Mr Joël Berthelin – Mr Olivier Villin

==========

REMERCIEMENTS

Je tiens tout d'abord à exprimer toute ma reconnaissance à Mme Selma Boumerdassi, ma directrice de mémoire, ainsi qu'à l'ensemble de l'équipe pédagogique du CNAM pour m'avoir encadré, orienté, aidé et conseillé dans la préparation de ce diplôme ou la rédaction de ce mémoire.

J'adresse mes plus sincères remerciements au Colonel Guimard, commandant la Sous-direction des Applications de Commandement et tuteur en milieu professionnel, qui m'a fait confiance dès 2016, soutenu dans mon projet tant personnel que professionnel et aidé à le mener à son terme.

Je remercie particulièrement le Lieutenant-Colonel Hamel, commandant le Bureau du Contrôle Opérationnel des Fichiers, qui m'a soutenu dans mes démarches et permis notamment de suivre les cours nécessaires à l'obtention de ce diplôme.

Je remercie également le Colonel Orvöen, le Capitaine Pratlong et l'Adjudant-Chef Camail pour leurs conseils avisés et m'avoir orienté.

J'adresse mes remerciements au Commandant Kespite, au Capitaine Marcq, commandant la Section Contrôle Technique, et à l'Adjudant-Chef Rosier pour leur accueil au sein de la section, leur soutien, la confiance qu'ils m'ont accordée dans le travail et les connaissances qu'ils m'ont apportées ou transmises.

Je tiens à remercier également Mme Eglantine Mallinger (Ingénieure à la Section Contrôle Fonctionnel) et Mr Vincent Laborie (Ingénieur Principal des SIC, expert en bases de données) pour leur relecture attentive de ce mémoire, leurs conseils et remarques ; sans oublier Mr Clément Oudot pour son éclairage sur le « monde extérieur » ainsi que messieurs Thierry Ramon et Cédric Martin pour l'infographie.

Je terminerai en remerciant mes parents, mon épouse Nadia, mon fils Charles pour leur compréhension, patience, relecture, corrections et leur soutien sans faille ainsi que Barbara et Mike Tinney qui m'ont permis de progresser en anglais et aidé à rédiger ce mémoire.

ABRÉVIATIONS & GLOSSAIRE

2FA / SFA : Authentification à Double Facteur / Second Facteur d'Authentification

AAA : Authentication - Authorization – Accounting

AD : Active Directory

ADER / RIE : ADministrations En Réseau / Réseau Interministériel d'Etat

AngularJS, Jquery, Node.js, Bootstrap : Framework et libraires JavaScript ou CSS

AES : Advanced Encryption Standard

API : Application Programming Interface

CAS : Central Authentication Service

CDA : Cross Domain Authentication

CDN : Content Delivry Network

CGI / FastCGI / uWSGI : Common Gateway Interface – Différents serveurs d'échange

CNIL : Commission Nationale de l'Informatique et des Libertés

CoffeeScript : Langage de programmation qui se compile en JavaScript

CRL / CSP : Certificate Revocation List / Content Security Policy

CSRF ou XSRF / XSS : Cross-Site Request Forgery / Cross Site Scripting

Curasso / Espresso : SSO de la Gendarmerie nationale exposés sur Internet

DMZ / DNS : Demilitarized Zone (Zone Démilitarisée) / Domain Name Service

ETL : Extract – Treatment and Load

EIDAS : Electronic IDentification Authentication and trust Services

FIDO : Fast Identity Online

FI / IdP : Fournisseur d'Identité / Identity Provider

FS / SP : Fournisseur de Service / Service Provider

FTP / IMAP : protocoles de transfert de fichier / messagerie

GNU GPL : GNU General Public License

HMAC : keyed-Hash Message Authentication Code

IE / IIS : Internet Explorer / Serveur Web MicroSoft

IGC / PKI : Infrastructure de Gestion des Clefs / Public Key Infrastructure

IPMS : Infrastructure de Production Mutualisée et Sécurisée

JO : Journal Officiel

JSON / YAML : Format de fichiers

KH : KeyHandle

LCD : Liquid Crystal Display

LDAP / LDIF : Lighweigt Directory Access Protocol / LDAP Data Interchange Format

LLNG : LemonLDAP::NG

LTS / NFC : Long Term Support / Near Free Contact

OS / USB : Operating System / Universal Serial Bus

OTP / TOTP / HOTP : One Time Password / Time-based OTP / HMAC-based OTP

PGS / POD: Plan Global de Secours / Plain Old Documentation

Proxyma / CheopsNG : SSO respectifs de la Gendarmerie et de la Police nationale

PSGI / Plack : Perl Web Server Gateway Interface / API PSGI

PSSI / SI : Politique de Sécurité des Systèmes d'Information – Système d'Information

RGPD : Règlement Général de la Protection des Données

SaaS / SMS : Software As A Service / Short Messages Service

SAML2 (Security Assertion Markup Language) / Shibboleth : Protocoles de fédération des identités

SCF / SCT : Section Contrôle Fonctionnel / Section Contrôle Technique

SGBD : Système de Gestion de Bases de Données

SOAP / REST : Simple Object Access Protocol / REpresentational State Transfer

SSL / TLS : Protocole de chiffrement asymétrique

SSO : Single Sign-On

STIG : Service du Traitement de l'Information de la Gendarmerie

TAP : Test Anything Protocol

U2F : Universal Second Factor (Norme)

VIP / VS / LB : Virtual IP / Virtual Server / Load Balencer

WS / WSDL : Web Service / Web Service Description Layer

XACML : eXtensible Access Control Markup Language

Table des matières

INTRODUCTION

Dans un monde toujours plus connecté, nous utilisons nombre de sites web, de services et d'applications nécessitant d'utiliser des identifiants pour accéder à du contenu ou à des fonctionnalités. Ce constat est valable que ce soit pour les particuliers, les entreprises ou les administrations. Or, le fait de devoir constamment se connecter, se déconnecter et d'être obligé de mémoriser des dizaines d'identifiants est devenu une réelle contrainte.

Outre la contrainte, le fait de devoir se ré-authentifier régulièrement peut nuire à la productivité tant pour les entreprises que pour les administrations. En effet, de nos jours, les utilisateurs doivent accéder à différentes plates-formes dont il est important de maîtriser et faciliter l'administration.

En plus de la productivité, la sécurité est devenue un enjeu majeur et nécessaire dans les Systèmes d'Informations modernes. En effet, les données et les applications ne demeurent plus au sein des organisations mais sont hébergées en « data-centers » communs à plusieurs d'entre-elles et une seule application vulnérable peut suffire à compromettre l'ensemble des ressources.

Enfin, la sécurité d'un SI basée uniquement sur l'utilisation des mots de passe est désormais devenue obsolète. C'est la raison pour laquelle les organisations font très souvent appel à l'authentification avancée, multi-critères ou multi-facteurs. Ces systèmes intègrent le référencement physique ou biométrique (cartes d'accès, analyse des empreintes, etc...) dans le processus de connexion, ce qui renforce la sécurité. Le coût de mise en œuvre d'une méthode d'authentification avancée a considérablement diminué depuis les dernières années. Toutefois, pour qu'elle soit retenue, cette solution doit être efficace, ergonomique et facile à déployer.

La Gendarmerie nationale lance en 2002 un vaste projet informatique d'extension des réseaux locaux et de son Intranet. En effet, jusqu'à présent, seuls les chefs-lieux de département avaient accès à ces services, mais pas les brigades ni les unités isolées ou l'outre-mer. L'Intranet se développant, la plupart des applications de gestion ou métiers, mais aussi les services réseaux tels que la messagerie, deviennent des applications en ligne.

Pour protéger, contrôler et tracer les accès à tous ces services ou applications, la Gendarmerie nationale décide en 2004 de se doter d'un portail d'authentification unique ou webSSO. A l'époque, furent retenues les solutions « SiteMinder » et « OpenSSO ». Après une période de tests, il s'est avéré que « SiteMinder », racheté par la suite par « Computer-Associates » perdait des traces d'accès et que le coût de licence était trop élevé, environ un million d'euros par an pour 120 000 utilisateurs. S'agissant d'« OpenSSO », réellement libre avant de devenir « OpenAM », il fut jugé inefficace.

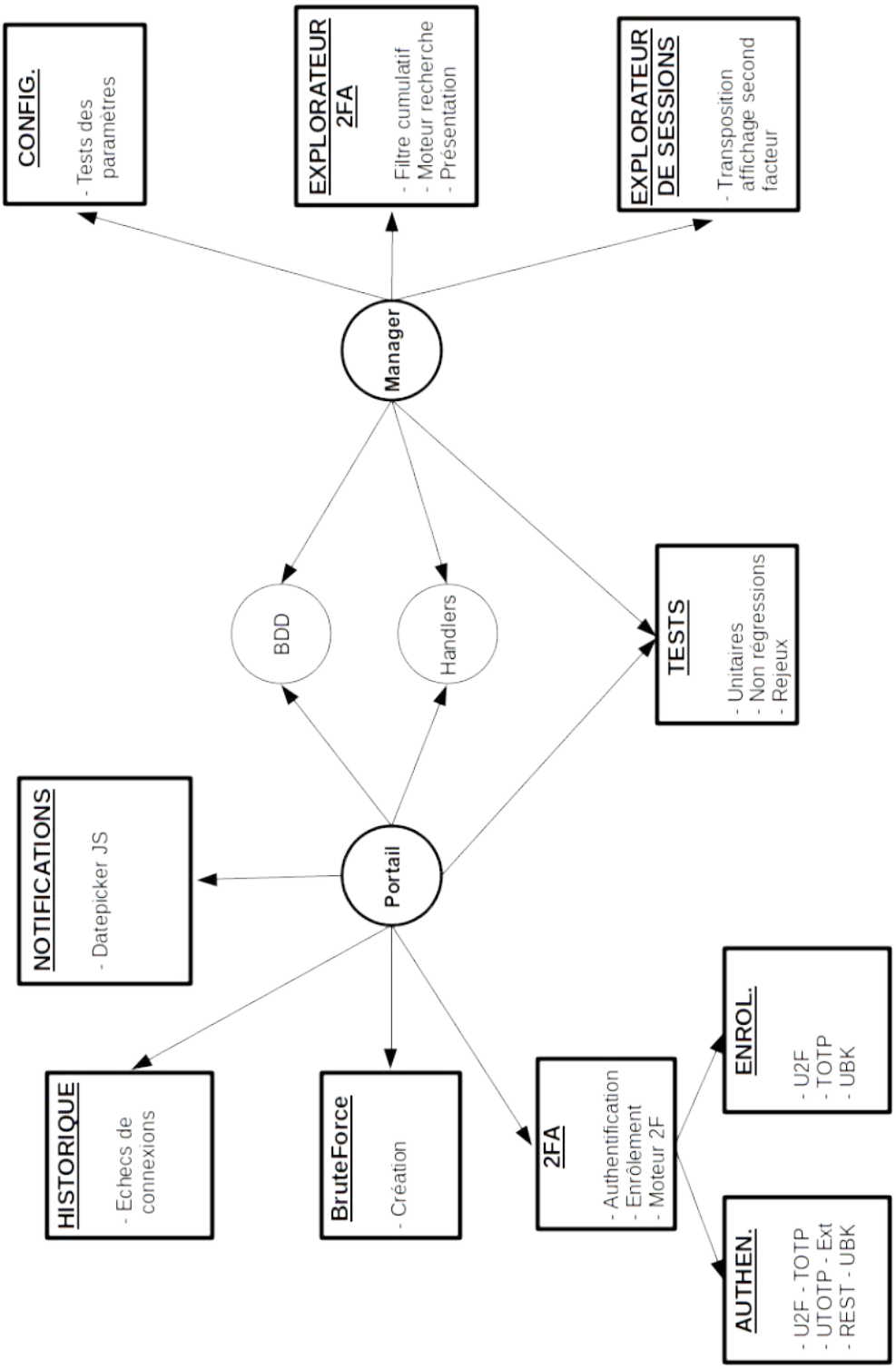

Figure 1 : Composants sur lesquels j’ai travaillé (en traits forts)

Dans la continuité du mouvement initié en 2001 consistant à privilégier une conception d'architecture du système d'information modulaire avec des briques présentant des interfaces standard, le choix des logiciels libres et open source pour les systèmes et applications de la Gendarmerie nationale s'est développé, dès lors que les fonctionnalités sont équivalentes ou approchantes à des offres éditeurs. La gendarmerie nationale a donc pris la décision de reprendre à son compte le projet « LemonLDAP », développé par le Ministère des Finances, pour l'améliorer et l'adapter à ses besoins. Celui-ci sera ensuite renommé « LemonLDAP::NG », porté dès l'origine par le colonel Xavier Guimard qui est encore aujourd'hui un contributeur important.

En août 2016, muté au Service des Technologies et des Systèmes d'Information de la Sécurité Intérieure dépendant de la Direction Générale de la Gendarmerie Nationale, je suis affecté à la Sous-direction des Applications de Commandement commandée par le Colonel Xavier Guimard, Bureau du Contrôle Opérationnel des Fichiers commandé par le Lieutenant-Colonel Hamel, Section Contrôle Technique commandée par le Capitaine Marcq.

Depuis mon affectation au sein de la SCT, je suis chargé avec les autres personnels de la section d'administrer les différentes plateformes SSO de la Sécurité Intérieure plus particulièrement « Proxyma », le SSO de la Gendarmerie nationale ; le « Portail de la Sécurité Intérieure », SSO fédéré pour les différentes forces de la Sécurité Intérieure (Police et Gendarmerie nationale, Douane, Préfecture de Police, Préfectures) et dans une moindre mesure « CheopsNG », le SSO de la Police nationale, « Curasso » et « Espresso », les SSO de la Gendarmerie nationale exposés sur Internet ainsi que les différentes DMZ, les serveurs « Proxy » ou encore la plateforme « Splunk » de collecte et d'analyse des traces d'accès.

Tous les SSO étant basés sur « LemonLDAP::NG » ; depuis 2016, j'ai mis en œuvre, déployé, administré et fait évoluer ses différents composants. Dans le cadre de ce mémoire, j'ai choisi de travailler sur une méthode d'authentification avancée basée sur l'utilisation classique d'un couple identifiant / mot passe et d'un second facteur. Pour ce faire, j'ai proposé à mon tuteur en milieu professionnel, le colonel Xavier Guimard, sous-directeur, d'intégrer l'équipe de développement du logiciel « LemonLDAP::NG », dont il est le fondateur, et d'implémenter l'authentification à double facteur ainsi que le support des différents seconds facteurs existants. Pour ce faire, j'ai développé des modules d'authentification et d'enrôlement spécifiques pour chaque type de second facteur, modifié les principaux composants de LLNG pour les adapter, travaillé tant sur le *Portail* que sur le *Manager* et livré les tests de non-régression correspondants. Pour ce faire, j'ai dû apprendre les langages *Perl*, *CoffeeScript*, *AngularJS*, *jQuery* et maîtriser le protocole HTTP et ses failles. La figure N°1 présente les composants sur lesquels je suis intervenu.

Après quelques rappels théoriques sur les principales technologies, normes et les concepts concernant l'authentification en général, je présenterai et détaillerai dans une seconde partie, la solution d'authentification « LemonLDAP::NG ». Dans la partie C, je ferai un état de l'art des différents seconds facteurs existants et des normes correspondantes. Je terminerai par la description de mon implémentation de l'authentification avec différents seconds facteurs, des problèmes ou difficultés auxquels j'ai été confronté, les solutions apportées et les futures évolutions envisagées.

A – Principaux concepts & Normes

1 – Principes AAA

L'acronyme AAA *(Authentication, Authorization, Accounting)* désigne les trois opérations que doit réaliser un système de gestion des accès.

1.1 – Authentification en général

Alors que l'identification consiste simplement à fournir son identité, l'authentification est le fait de s'assurer que l'entité qui se présente est bien celle qu'elle prétend être. L'authentification permet à l'utilisateur de prouver son identité. En plus de l'authentification, le service peut récupérer des informations sur les utilisateurs, mais ceci n'est pas obligatoire.

1.1.1 – Pour un Système d'Information

D'après, (« Authentification », 2017), l'authentification pour un Système d'Information (SI) est un processus permettant au système de s'assurer de la légitimité de la demande d'accès faite par une entité qui peut être un individu ou un autre système. Ceci a pour but d'autoriser cette entité à accéder à des ressources du SI conformément aux règles d'accès définies. L'authentification permet donc de valider la légitimité de l'accès puis le système attribue à cette entité les données ou attributs d'identité pour cette session. Ces attributs sont détenus par le système ou peuvent être fournis par l'entité lors du processus d'authentification. C'est à partir des éléments issus de ces deux processus que le contrôle d'accès aux ressources du système pourra être paramétré.

Dans le cas d'un individu, l'authentification consiste, en général, à vérifier que celui-ci possède une preuve de son identité ou de son statut. Il existe quatre facteurs d'authentification classiques qui peuvent être utilisés dans le processus d'authentification. Il est possible de demander une information que seul le commettant connaît tel un mot de passe, requérir un objet que seul le commettant possède comme une carte à puce, avoir recours à une information qui caractérise le commettant telle une empreinte ou enfin utiliser une information que seul le commettant peut produire, un geste par exemple. D'autres facteurs d'authentification peuvent parfois être utilisés comme les contraintes temporelles ou les capacités de localisation.

La phase de vérification fait intervenir un protocole d'authentification qui peut être de trois types. Nous distinguons tout d'abord l'authentification simple ne reposant que sur un seul élément ou facteur. Ensuite, il existe l'authentification forte qui requiert au moins deux facteurs différents. Enfin, les architectures informatiques actuelles reposent sur le concept d'authentification unique.

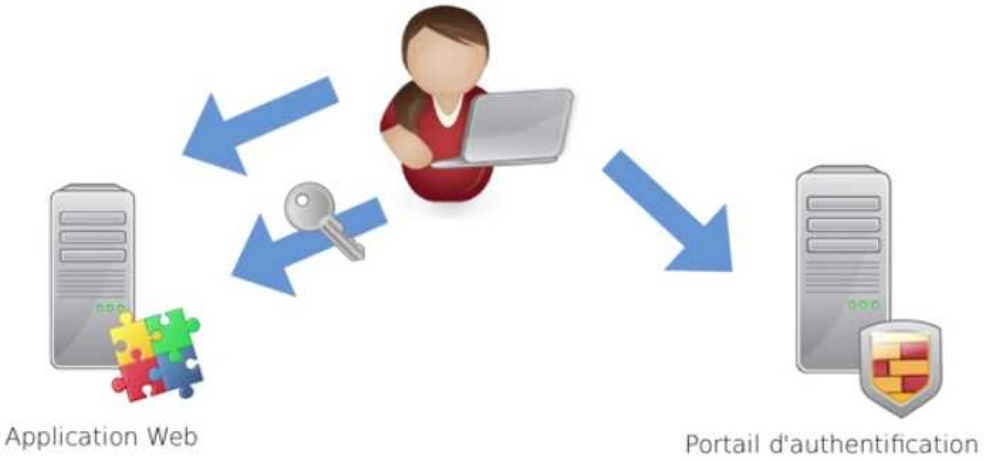

Figure 2 : Authentification unique

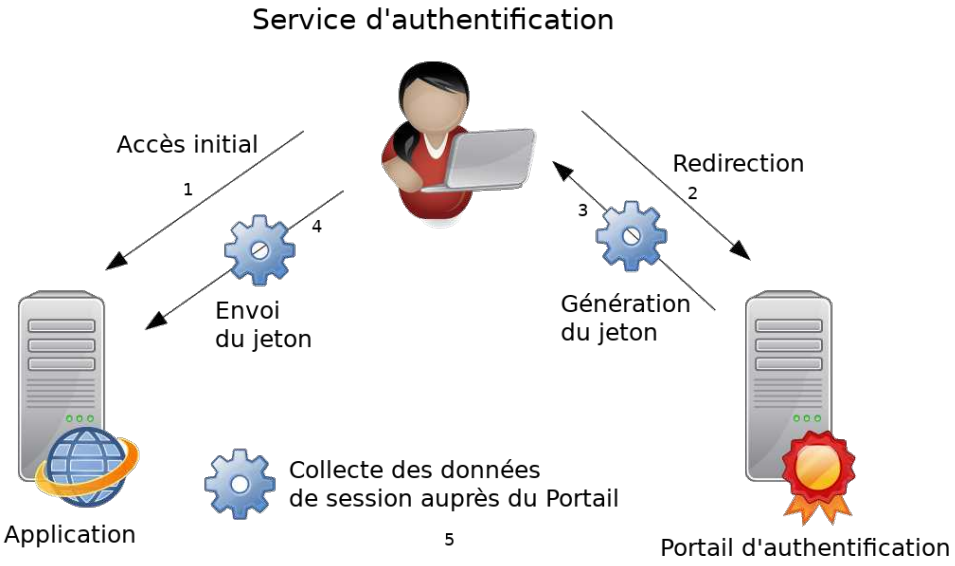

Figure 3 : Service d’authentification

1.1.2 – Le modèle unique

Comme expliqué par (« Authentification unique », 2018), l'authentification unique ou Single Sign-On (SSO) est une architecture composée de plusieurs éléments permettant à une entité de ne s'authentifier qu'une seule fois pour avoir accès à toutes les ressources auxquelles elle a droit que ce soit des applications, des données ou des services. Le principe de base est d'intercepter les requêtes entre le client et le serveur puis d'indiquer au serveur que le client est bien authentifié et autorisé (cf figures 2 & 3 ci-contre).

Les objectifs de ce modèle sont, d'une part, d'offrir aux utilisateurs une meilleure ergonomie en simplifiant la navigation et de faciliter la gestion de l'authentification en ne proposant qu'une seule interface ou serveur d'authentification. D'autre part, d'un point de vue sécuritaire, un SSO permet une gestion centralisée des données personnelles détenues par les différents services en ligne, en les coordonnant par des mécanismes de type méta-annuaire par exemple et d'améliorer la sécurité du SI en simplifiant la définition et la mise en œuvre d'une Politique de Sécurité du Système d'Informations (PSSI).

1.2 – Autorisation

Il s'agit ici de permettre ou d'interdire l'accès à une ressource pour un utilisateur ou un système. Une requête d'autorisation contenant les éléments d'identité obtenus lors de la phase d'authentification est transmise au serveur. Celui-ci, en fonction des règles définies, autorise ou non l'accès à la ressource demandée.

1.3 – Accès & Traçabilité

La traçabilité désigne la situation où l'on dispose de l'information nécessaire et suffisante pour connaître (éventuellement de façon rétrospective) les actions effectuées dans un système d'informations. Elle a pour but de comptabiliser les accès aux ressources, suivre les opérations effectuées en vue d'auditer les performances, les erreurs, détecter d'éventuelles attaques en surveillant des tentatives d'accès non autorisées ou d'imputer les actions aux utilisateurs comme les débuts ou fins de connexions, de transactions.

2 – Authentification Unique (SSO)

2.1 – SSO & WebSSO

On parle de WebSSO quand l'authentification unique concerne uniquement des applications Web. C'est à dire des applications client-serveur dont le client est un navigateur Web comme *Internet Explorer* de *Microsoft*, *Mozilla Firefox* ou *Google Chrome*.

Un SSO permet également de protéger des applications nécessitant un client lourd. Le client lourd est une application spécifique installée sur les postes qui se charge de renseigner l'identifiant et le mot de passe à la place de l'utilisateur. Pour ce faire, l'application peut lire les données de session issues du système d'exploitation ou d'un certificat détenu par l'utilisateur comme une carte professionnelle par exemple.

2.1.1 – Avantages

D'après (« Authentification unique », 2018), les différents avantages de l'authentification unique sont tout d'abord la réduction de la fatigue des mots de passe due au manque de souplesse lié à l'utilisation de différentes combinaisons de nom d'utilisateur et de mot de passe, une réduction du temps passé à saisir le même mot de passe pour le même compte ainsi que la gestion des oublis de mots de passe dans le cadre du support informatique et limite les risques liés au comportement des utilisateurs comme l'écriture sur post-it des mots de passe face à leur multiplicité. Elle permet la centralisation des systèmes d'authentification, la sécurisation à tous les niveaux d'entrée, de sortie et d'accès aux systèmes sans sollicitations multiples des utilisateurs.

En outre, une solution SSO améliore la sécurisation de l'authentification. En effet, les mots de passe ne circulent plus entre les applications et les serveurs applicatifs car ils sont remplacés par des jetons qui peuvent prendre plusieurs formes.

Enfin, un SSO permet d'étendre plus facilement les services offerts aux utilisateurs comme la gestion des mots de passe, la fédération d'identité, le contrôle d'accès ou la transmission d'informations de session. L'utilisateur s'authentifie uniquement auprès du *Portail* qui lui délivre un jeton et crée une session. Les applications contactent le *Portail* pour vérifier la validité du jeton qui peut leur envoyer des informations d'identité ou autres.

2.1.2 – Critiques

Comme un SSO donne accès potentiellement à de nombreuses ressources, une fois l'utilisateur authentifié, les pertes peuvent être lourdes si une personne mal intentionnée a accès à des informations d'identification des utilisateurs. Avec un SSO, il est important de renforcer la sécurité.

Une attention particulière doit donc être prêtée à ces informations et des méthodes d'authentification forte devraient idéalement être combinées telle l'authentification unique à deux facteurs. Il s'agit du principal inconvénient de cette architecture.

Dans certains cas particuliers, comme pour le *webmail* par exemple, c'est l'application qui doit se connecter au serveur. L'application n'ayant pas le mot de passe de l'utilisateur, il lui est impossible de rejouer l'authentification afin d'accéder au service. C'est très bien pour la sécurité mais cela peut parfois être un problème. Dans ce cas, le SSO peut transmettre l'identifiant et le mot de passe de l'utilisateur à l'application mais cela doit rester exceptionnel et différentes méthodes plus ou moins sécurisées existent pour palier cette contrainte.

2.2 – Différentes approches

Comme précisé dans (« Authentification unique », 2018), il existe trois types d'approche pour la mise en œuvre de systèmes d'authentification unique : les approches centralisée, fédérative ou coopérative.

2.2.1 – Centralisée

Le principe de base ici est de disposer d'une base de données globale et centralisée de tous les utilisateurs ou d'un annuaire. Cela permet également de centraliser la gestion de la politique de sécurité. Cette approche est principalement destinée à des services dépendants tous d'une même organisation.

2.2.2 – Fédérative ou Coopérative

Dans ces approches, chaque service gère un ensemble d'utilisateurs (fédérative) ou une partie des données d'un utilisateur (coopérative) mais partage les informations dont il dispose sur l'utilisateur avec les services partenaires. Avec ces approches, l'utilisateur peut donc disposer de plusieurs comptes. Lorsqu'il souhaite accéder à un service offert par un des partenaires, l'utilisateur choisit le fournisseur d'identité sur lequel il souhaite s'authentifier puis les données d'identité sont transmises au partenaire. Elles ont été développées pour répondre à un besoin de gestion décentralisée des utilisateurs, où chaque service partenaire désire conserver la maîtrise de sa propre politique de sécurité. Généralement, chaque utilisateur dépend d'une des entités partenaires. Lorsqu'il cherche à accéder à un service du réseau, l'utilisateur s'authentifie auprès du partenaire dont il dépend.

3 – Principales solutions SSO AAA

Dans ce chapitre, j'ai choisi de présenter les solutions exclusivement de SSO adaptées à un usage par des administrations ou organisations offrant les fonctionnalités d'Authentification, d'Autorisations et de Traçabilité des accès *(Authentication, Authorization & Accounting en anglais)* les plus complètes, proposant de bonnes performances malgré une forte charge et dont les sources sont librement disponibles.

3.1 – OpenAM

D'après (« OpenAM », 2018) Open Access Management est un système de gestion des identités et de contrôle des accès ainsi qu'une plateforme serveur de fédération d'identités. OpenAM est distribué et maintenu par la société « ForgeRock » sous licence commerciale CDDL (« Common Development and Distribution License 1.0 », s. d.)). Ceci signifie qu'OpenAM est en sources ouvertes mais soumis au paiement d'un droit d'utilisation en fonction des services choisis (intégration, support, formation, etc.), du nombre d'utilisateurs ou de connexions par mois.

C'est en 2010 que la société « ForgeRock » a annoncé qu'elle continuerait à développer et supporter le logiciel OpenSSO de l'ancienne société Sun suite à la décision d'Oracle d'abandonner le développement du projet. Elle a rebaptisé le produit OpenSSO en OpenAM car Oracle conserve les droits sur le nom OpenSSO.

Comme expliqué par (« OpenAM - Authorization Guide », s. d.) et (« AM 6 > Quick Start Guide », s. d.) OpenAM est développé en Java et basé sur le moteur Sun Java System Access *Manager*. Il fut récompensé du prix « Sécurité » en 2009. En plus d'implémenter les services AAA, OpenAM offre les fonctionnalités spécifiques suivantes :

- un moteur d'authentification adaptatif : En fonction de critères de risques tels l'adresse IP source, l'utilisation d'un nouvel équipement ou l'heure de connexion, le moteur calcule un niveau de risque et peut imposer l'utilisation d'un facteur d'authentification supplémentaire.
- la fédération d'identités basée sur les protocoles SAML2, OpenId Connect ou CAS. OpenAM peut être utilisé comme relais entre ces différents protocoles
- la haute disponibilité des services par l'utilisation de répartiteurs de charge (LB) et la redondance des serveurs ainsi que des sessions en garantissant l'intégrité de leurs données
- une interface de développement (API) Java, C ou REST permettant d'interroger le serveur via un navigateur web
- des politiques de sécurité permettant de filtrer les URL en utilisant des métacaractères type joker, de filtrer sur les méthodes HTTP (POST, GET, etc...) ou de restreindre les accès en fonction de plages horaires

- des règles écrites dans un langage propriétaire proche d'un langage naturel mixant « verbes » comme du SQL et des opérateurs de comparaison
- la possibilité d'exporter les règles ou les configurations sous différents formats comme JSON, YAML ou XACML (« eXtensible Access Control Markup Language (XACML) Version 3.0 », s. d.)
- des agents disponibles pour les serveurs web Apache, Nginx Plus et IIS

OpenAM étant une application web Java, la solution doit être exécutée sur un serveur de type Apache Tomcat avec une machine Java récente. Le fichier d'installation consiste en un paquet compressé « war » unique emportant toutes les fonctionnalités.

3.2 – LemonLDAP::NG

Le logiciel LemonLDAP::NG (« LemonLDAP », 2017) est issu du projet LemonLDAP dont il est un *fork*, un dérivé. Créé à l'origine par Éric German et basé sur le protocole de gestion d'annuaire LDAP, LemonLDAP fut développé pour le Ministère des Finances français. LemonLDAP fut ensuite repris en 2004 par la Gendarmerie nationale pour l'améliorer et l'adapter à ses besoins.

LemonLDAP::NG (abrégé en LLNG) est distribué sous licence GPLv2 donc gratuit et libre. Il fournit une solution d'authentification unique distribuée avec gestion centralisée des droits. LemonLDAP::NG a obtenu plusieurs distinctions dont le « Lutèce d'Or du Meilleur projet libre réalisé pour une administration » en 2006 et le « OW2 Community Award » en 2014. En plus des « classiques » fonctions AAA et de fédération d'identité, LemonLDAP::NG apporte les fonctionnalités suivantes :

- un système d'authentification unique basé sur des cookies sécurisés ou non, afin de limiter les risques en cas d'interception de cookie
- un menu dynamique des applications
- un dispositif de notification lors de l'ouverture de la session pour notifier un changement de droits ou de politique d'accès par exemple
- un explorateur de sessions permettant de filtrer par adresse IP, date ou type et de supprimer des sessions
- des règles d'accès basées sur les expressions régulières
- une architecture construite suivant le concept MVC (Modèle – Vue – Contrôleur). Le *Portail* peut donc être facilement personnalisable
- la possibilité de restreindre le nombre d'ouvertures de sessions par utilisateur ou adresse IP
- l'accès à toutes les variables de session et l'utilisation de règles comportant des fonctions personnalisées ou étendues comme la date système ou le chiffrement

- toutes les règles ou macros sont exécutées dans une « cage sécurisée ». Il s'agit d'un conteneur allouant de la mémoire, des ressources et un espace de nommage afin d'éviter l'exécution de code malicieux, les injections ou tout simplement les erreurs
- une souplesse d'implémentation permettant l'adaptation à de nombreux systèmes et une très haute disponibilité
- un moteur de rejeu pour compléter des formulaires mais ceci doit rester une solution de dernier recours pour des applications ne pouvant être protégées autrement
- la possibilité d'être déployé selon quatre formes d'architecture différentes pour protéger les applications :

- en mandataire inverse (reverse-proxy), qui concentre en un point unique le *Portail* d'authentification et les agents de contrôle. L'utilisateur n'accède alors jamais directement aux applications mais utilise toujours le mandataire

- par délégation, où les agents de contrôles sont déployés au plus près des applications. Dans ce cas l'utilisateur accède directement aux applications, dont les agents interagissent avec le *Portail* d'authentification pour valider les sessions

- mixte, mélangeant le mandataire inverse et la délégation

- par fédération d'identités, les applications échangent directement avec le *Portail*

3.3 – Synthèse comparative

J'ai choisi de mettre en parallèle et comparer ces deux solutions par rapport à des critères de facilité d'utilisation ou de maintenance, de passage à l'échelle, de sécurité et de disponibilité pour l'infrastructure ou les applications, d'adaptabilité ou d'intégration à l'architecture existante et enfin en terme de coût et de performances globales.

En ce qui concerne la facilité d'utilisation, OpenAM propose énormément d'options. C'est une solution extrêmement complète voire trop. Elle est par conséquent plus difficile à prendre en main ou à aborder et nécessite un passage obligé par l'appropriation de la documentation abondante. LemonLDAP::NG propose un *Manager* clair, organisé par modules qu'il est possible d'activer ou non en modifiant le fichier de configuration. Les menus sont présentés sous forme d'arbre de catégories dépliantes avec une aide contextuelle. En plus des classiques fonctions de sauvegarde et d'import / export de la configuration, il est possible d'afficher les différences entre deux versions de configurations et de naviguer entre-elles pour les restaurer.

S'agissant de la maintenabilité, OpenAM nécessite beaucoup de briques logicielles critiques et complexes issues de différents éditeurs (OpenAM, JRE, Tomcat, Apache, …) ce qui impose de veiller et d'installer fréquemment les mises à jour de sécurité si les éditeurs tiers sont réactifs chacun à leur niveau respectif… En outre, OpenAM est distribué sous forme d'un seul

fichier comportant plus de 2 millions de lignes de code donc pas nécessairement facile à maintenir et potentiellement plus sujet à failles de sécurité ou bug. Pour en terminer sur l'aspect des briques logicielles, OpenAM n'implémente pas rapidement les dernières technologies disponibles.

D'un point de vue passage à l'échelle, LemonLDAP::NG étant entièrement modulaire, il est tout à fait possible de séparer les instances des serveurs hébergeant le *Portail*, des reverse-proxy protégeant les applications. Il suffit d'ajouter des instances pour répondre aux besoins grandissants de ressources. LLNG est capable de servir plus de 200 000 utilisateurs sans difficultés. De plus, cette architecture modulaire est gage d'une haute disponibilité car les services sont fournis par différents équipements (bases de données, annuaires, *Portail*, etc …) eux-mêmes « montés » derrière des répartiteurs de charge (Load Balencer).

Je terminerai sur les trois aspects les plus avantageux qui jouent en la faveur de LemonLDAP::NG que sont le coût de la licence d'exploitation, la facilité d'intégration et les performances.

LemonLDAP::NG étant distribué sous licence GPL, son utilisation est totalement gratuite et il peut être distribué librement. Une licence OpenAM pour une organisation comportant environ 120 000 utilisateurs est de l'ordre de quelques millions d'euros par an ! Pour les quinze dernières années d'exploitation, LLNG aura coûté à la Gendarmerie environ 800 000 euros pour son développement, l'intégration des protocoles SAML et OpenID-Connect ainsi que sa maintenance …

La structure MVC, le côté modulaire de ses composants et les sources ouvertes permettent d'adapter et de personnaliser LemonLDAP::NG assez facilement à l'architecture existante de l'entreprise ou de l'organisation ainsi qu'à des besoins particuliers. En outre, il est possible de faire appel à des sociétés d'intégration. Ce service est moins onéreux que le paiement d'une licence d'exploitation et de prestations annexes.

Enfin, des mesures de performance (« documentation:2.0:performances [LemonLDAP::NG] », s. d.) et (« documentation:2.0:psgi [LemonLDAP::NG] », s. d.) effectuées sur un reverse-proxy Nginx avec et sans l'agent de protection ont montré une différence de seulement 3 millisecondes dans le traitement des requêtes HTTP. Ceci signifie que l'empreinte de l'agent logiciel traitant les requêtes est de 3 millisecondes outre la requête initiale au serveur. Ces tests ont également permis de déterminer les composants les plus adaptés en fonction des bases de données utilisées, des options choisies ou des *Handler*s utilisables en fonction de l'architecture disponible (type de serveur web, fonctionnement en mode ReverseProxy ou direct, serveur d'accès FastCGI, uWSGI ou PSGI).

Ces très bonnes performances sont obtenues notamment par l'utilisation des groupes locaux ou macros dans l'écriture des permissions qui permettent d'éviter les règles d'accès trop complexes à compiler pour le moteur d'expressions régulières. En effet, les macros et groupes locaux sont calculés une seule fois par le *Portail* au moment de la création de la session alors que les règles sont lues, « parsées » par le *Handler* pour chaque requête HTTP. Cette possibilité n'existe pas dans OpenAM.

4 – Normes & Protocoles

Cette section présente quelques normes et protocoles utilisés plus avant dans ce mémoire.

4.1 – Méthodes d’authentification HTTP & Vulnérabilités

4.1.1 – Basique

Il s’agit d’un mécanisme permettant de demander un identifiant et un mot de passe au client avant qu’il n’accède à une page web. Celui-ci est implémenté sur le serveur. Il permet à un ensemble de pages (appelé domaine) d'être protégé par l'authentification.

Ce mécanisme est défini par la spécification HTTP. Il est le plus commun et simple à mettre en œuvre mais aussi le moins sécurisé. Les différentes étapes d’une authentification basique sont :

- le client demande une page web
- le serveur répond avec une erreur (code HTTP 401), demandant une authentification
- le client refait une demande contenant les détails d'authentification dans la requête
- le serveur vérifie les informations et envoie la page demandée, ou une autre erreur (403 par exemple)

Les exemples suivants détaillent ces étapes :

GET /index.html HTTP/1.1

Host: www.example.com

HTTP/1.1 401 Unauthorized

WWW-Authenticate: Basic realm="EXAMPLE"

GET /index.html HTTP/1.1

Host: www.example.com

Authorization: Basic bWF1ZG91eDpzZWNyZXQK== → Base64(maudoux:secret)

4.1.2 – Digest

L'authentification d'accès « Digest » est une méthode qu'un client et un serveur peuvent utiliser pour échanger des informations d'identification sur HTTP. Cette méthode utilise une combinaison du mot de passe et d'autres éléments d'information pour créer un hachage MD5 qui est ensuite envoyé au serveur d'authentification. L'envoi d'un hachage évite les problèmes avec la transmission d'un mot de passe en texte clair, une lacune de l'authentification d'accès basique.

4.1.3 – Cookie

Cette méthode utilise les « cookies » HTTP pour authentifier les utilisateurs et surtout maintenir la session ouverte car l'authentification dépend d'un autre système. Les étapes sont les suivantes :

- Le client envoie une requête d'authentification au serveur
- En cas de succès, le serveur répond en incluant l'entête « Set-Cookie » contenant le domaine de validité du cookie, l'identifiant de session, la date de validité, éventuellement un chemin d'accès à la ressource et des options. Par exemple :

  Set-Cookie: JSESSIONID=abcde12345; Path=/; HttpOnly
- Pour chaque requête suivante, le client doit envoyer son cookie de session

  Cookie: JSESSIONID=abcde12345
- Lors de la déconnexion, le serveur positionne l'entête « Set-Cookie » à vide ce qui a pour effet de supprimer le cookie du client

La taille maximale d'un cookie est de 4 ko. Un cookie est valide jusqu'à la fermeture du navigateur ou jusqu'à une date spécifiée. De plus, il n'est utilisable que sur un domaine ou une ressource donnés. Les cookies peuvent être « HTTPOnly » ou « Secure ». Un cookie « HTTPOnly » ne peut être lu que par un navigateur et non par des applications JavaScript ou Java par exemple. Un cookie « Secure » est valide uniquement avec le protocole HTTPS.

Il est important de noter que la méthode d'authentification par cookie est vulnérable à l'attaque CSRF / XSRF (Cross-Site Request Forgery). Elles nécessite donc de prendre des mesures de sécurité spécifiques comme l'utilisation de jetons CSRF.

4.1.4 – Attaque CSRF & Prévention

D'après (« Cross-site request forgery », 2018) et (« Les attaques de type « cross-site request forgery » – CERT-FR », s. d.), l'objet de cette attaque est de transmettre à un utilisateur authentifié une requête HTTP falsifiée qui pointe sur une action interne au site, afin qu'il l'exécute sans en avoir conscience et en utilisant ses propres droits. L'utilisateur devient donc complice d'une attaque sans même s'en rendre compte. L'attaque étant actionnée par l'utilisateur, un grand nombre de systèmes d'authentification sont contournés. La principale contre-mesure est d'utiliser des jetons de validité dans les formulaires (Synchronizer Token Pattern).

Il s'agit de faire en sorte qu'un formulaire posté ne soit accepté que s'il a été produit quelques minutes auparavant en générant côté client un jeton avec une période de validité. Le jeton anti-CSRF doit être transmis en paramètre par chaque formulaire et vérifié côté serveur ce qui en complexifie la mise en œuvre. Le jeton peut être valide pour une session au lieu d'une requête pour permettre au client l'utilisation de plusieurs onglets dans son navigateur.

4.2 – Quelques rappels sur HTTP

HTTP (« Hypertext Transfer Protocol (HTTP) », 2018) est un protocole de communication client-serveur. La variante HTTPS est sécurisée par l'usage de SSL / TLS. HTTP est un protocole applicatif de la couche ISO 7 (application).

4.2.1 – Versions

La version HTTP/1.0 est définie en 1996. Cette version supporte les serveurs HTTP virtuels, la gestion de cache et l'identification. En 1997, HTTP/1.1 devient le standard de l'IETF. Cette version ajoute le support du transfert en « pipeline » pour permettre le téléchargement en parallèle et la négociation de type de contenu comme le format de données transmises ou de la langue utilisée.

Le protocole HTTP/1.0 prévoit l'utilisation d'en-têtes. La gestion de la connexion est la suivante : le client établit la connexion, envoie une requête, le serveur répond et ferme immédiatement la connexion. Une requête HTTP est un ensemble de lignes envoyé au serveur par le navigateur.

Elle comprend :

- une ligne de requête précisant l’URL du document demandé, la méthode qui doit être appliquée et la version du protocole utilisée par le client.
- les champs d'en-tête de la requête (en-tête) : il s'agit d'un ensemble de lignes facultatives permettant de donner des informations supplémentaires sur la requête ou le client (navigateur, système d'exploitation, etc...). Chacune de ces lignes est composée d'un nom qualifiant le type d'en-tête, suivi de deux points (:) et de la valeur de l'en-tête
- une ligne vide si la requête contient un corps
- le corps de la requête : c'est un ensemble de lignes optionnelles devant être séparées des lignes précédentes par une ligne vide et permettant par exemple un envoi de données par une méthode POST lors de l'envoi de données au serveur par un formulaire

Une requête HTTP présente le format suivant :

Méthode HTTP <Ligne de commande> (Commande, URL, Version de protocole)

En-tête de requête

[Ligne vide]

Corps de requête

Une réponse HTTP présente le format suivant :

Ligne de statut (Version, Code-réponse, Texte-réponse)

En-tête de réponse

[Ligne vide]

Corps de réponse

Exemple de requête HTTP :

```
GET /page.html HTTP/1.1
Host: example.com
Referer: http://example.com/
User-Agent: CERN-LineMode/2.15 libwww/2.17b3
```

Le protocole HTTP/1.1 apporte une meilleure gestion du cache et l'en-tête « Host » devient obligatoire dans les requêtes. Les soucis majeurs des deux premières versions du protocole HTTP sont d'une part le nombre important de connexions lors du chargement d'une page complexe (contenant beaucoup d'images ou d'animations) et d'autre part le temps d'ouverture d'une connexion entre client et serveur. En effet, l'établissement d'une connexion TCP prend un temps triple de latence entre client et serveur. Le client envoie un segment SYN au serveur. Le serveur lui répond par un segment SYN/ACK et le client confirme par un segment ACK. Des expérimentations de connexions persistantes ont cependant été effectuées avec HTTP/1.0 mais cela n'a été définitivement mis au point qu'avec HTTP/1.1.

Par défaut, HTTP/1.1 utilise des connexions persistantes, autrement dit la connexion n'est pas immédiatement fermée après une requête, mais reste disponible pour une nouvelle requête. On appelle souvent cette fonctionnalité *keep-alive*. Il est aussi permis à un client HTTP d'envoyer plusieurs requêtes sur la même connexion sans attendre les réponses. On appelle cette fonctionnalité *pipelining*. La persistance des connexions permet d'accélérer le chargement de pages contenant plusieurs ressources, tout en diminuant la charge du réseau.

4.2.2 – Méthodes courantes

Les méthodes HTTP les plus courantes sont :

- GET : Requête de la ressource située à l'URL spécifiée
- HEAD : Requête de l'en-tête de la ressource située à l'URL spécifiée
- POST : Envoi de données au programme situé à l'URL spécifiée

- PUT : Envoi de données à l'URL spécifiée
- DELETE : Suppression de la ressource située à l'URL spécifiée

La méthode GET est utilisée pour récupérer les données d'une ressource spécifique. Il s'agit de la méthode HTTP la plus utilisée. Une particularité de cette méthode est que les paramètres sont transmis dans l'URL sous la forme « ?nom1=valeur1&nom2=valeur2&... ». Cette liste de valeurs est appelée 'QueryString'. Par exemple : « /test/demo_form.php?name1=value1&name2=value2 ».

Tous les langages de programmation incluent des librairies CGI qui proposent des méthodes pour découper et récupérer ces paramètres qui peuvent être les valeurs transmises par le biais d'un formulaire.

A la différence de GET, la méthode POST permet d'envoyer des données au serveur pour créer ou mettre à jour une ressource. Autre différence avec GET, la méthode POST transmet tous les paramètres dans le corps de la requête.

4.2.3 – URL, URI & QueryString

Toutes les URL HTTP sont conformes à la syntaxe URI (Universal Ressource Identifier). Cette syntaxe consiste en une séquence hiérarchique de 5 éléments :

URI = scheme:[//authority]path[?query][#fragment].

Une URL (*Uniform Ressource Locator*), couramment appelée adresse web, est une référence à une ressource web. Elle permet de spécifier la localisation de cette ressource sur un serveur ou un mécanisme pour la retrouver. En fait, une URL est un type particulier d'URI. Dans la majorité des cas, une URL fait référence à une page web (HTTP) mais peut également être utilisée pour les protocoles FTP (transfert de fichiers), JDBC (accès à des bases de données), etc …

- Le s*chéma* précise le protocole utilisé.
- *authority* et *path* précisent le chemin d'accès à la ressource.
- La q*ueryString* est une partie de l'URL contenant les paramètres transmis par le navigateur au serveur web.
- Le *fragment* correspond à une ancre spécifique de la page.

4.2.4 – Entêtes & Cookies

Les en-têtes sont des valeurs transmises dans la partie en-tête de la requête HTTP. Ils sont transmis sous la forme « Nom:valeur ». Par exemple :

- Host : Spécifie le nom de domaine du site s'il y a plusieurs sites à la même adresse IP.
- Connection : Définit le type de connexion :
    - Close : la connexion est fermée après la réponse
    - Keep-Alive : crée une connexion persistante
- Content-type : Spécifie le type MIME du corps de requête (HTML, Json, etc...)
- Content-Length : Précise la longueur du corps de requête
- Cookie : Permet d'envoyer au serveur les cookies qui ont été enregistrés. Un cookie a toujours un nom et une valeur.

Les types MIME (« Type de médias », 2018) sont des identifiants de format de données sur internet en deux parties. L'IANA (autorité qui attribue les adresses IP, entre autres) définit actuellement les dix types suivants : application, audio, example, font, image, message, model, multipart, text et video, ainsi que les quatre arbres de sous-type suivants : arbre standard (sans préfixe), arbre éditeur (préfixe vnd.), arbre personnel (préfixe prs.) et arbre non enregistré (préfixe x.). Par exemple, le type de médias text/html; charset=UTF-8 est composé du type text, du sous-type html de l'arbre standard et du paramètre optionnel charset=UTF-8.

Le cookie est défini par le protocole de communication HTTP comme étant une suite d'informations envoyée par un serveur HTTP à un client HTTP, que ce dernier retourne lors de chaque interrogation du même serveur HTTP sous certaines conditions.

Les cookies permettent aux développeurs de sites web de conserver des données utilisateur afin de faciliter la navigation et de permettre certaines fonctionnalités. Ils sont envoyés en tant qu'en-tête HTTP par le serveur au navigateur qui le renvoie inchangé à chaque fois qu'il accède au serveur. Un cookie peut être utilisé pour une authentification, une session et pour stocker une information spécifique sur l'utilisateur. Si une durée de validité est précisée, il est stocké dans le cache du navigateur jusqu’à cette date. À défaut, il disparaît lorsque le navigateur est fermé.

4.2.5 – Codes HTTP courants

Les codes HTTP sont classés par type. Ci-dessous, les codes les plus usuels :

- Les codes 20x → Réussite. Ces codes indiquent le bon déroulement de la transaction. Par exemple, « 200 OK : La requête a été accomplie correctement »

- Les codes 30x → Redirection. Ces codes indiquent que la ressource n'est plus à l'emplacement indiqué. Par exemple : « 301 MOVED : Les données demandées ont définitivement été transférées à une nouvelle adresse » et « 302 FOUND : Les données demandées doivent être demandée à une nouvelle URL ».
- Les codes 40x → Erreurs dues au client. Ces codes indiquent que la requête est incorrecte. Par exemple : « 400 BAD REQUEST : La syntaxe de la requête est mal formulée ou est impossible à satisfaire », « 401 UNAUTHORIZED : Le paramètre du message donne les spécifications des formes d'autorisation acceptables. Le client doit reformuler sa requête avec les bonnes données d'autorisation », « 403 FORBIDDEN : L'accès à la ressource est interdit » et « 404 NOT FOUND : Le serveur n'a rien trouvé à l'adresse spécifiée ».
- Les codes 50x → Erreurs dues au serveur. Ces codes indiquent qu'il y a eu une erreur interne du serveur. Par exemple « 500 INTERNAL SERVER ERROR : Le serveur a rencontré une condition inattendue qui l'a empêché de donner suite à la demande ».

4.2.6 – HTTP & TLS / SSL

D'après (« HTTPS », 2018) (« Certificat électronique », 2018) et (« Demande de signature de certificat », 2018), afin de sécuriser les échanges HTTP, une sur-couche SSL / TLS est utilisée pour chiffrer les échanges entre le client et le serveur. Le protocole SSL est basé sur le principe du chiffrement asymétrique, couple clef privée / clef publique. La propriété des algorithmes asymétriques est qu'un message chiffré par une clef privée sera lisible par tous ceux qui possèdent la clef publique correspondante. À l'inverse, un message chiffré par une clef publique n'est lisible que par le propriétaire de la clef privée correspondante. Le problème vient de la transmission de la clef publique. Si celle-ci n'est pas sécurisée, un attaquant peut se positionner entre l'entité et son public en diffusant de fausses clefs publiques puis intercepter toutes les communications, lui permettant d'usurper l'identité du diffuseur des clefs publiques (attaque de type man-in-the-middle). Les certificats résolvent le problème du canal sécurisé grâce à la signature de tiers de confiance.

Un certificat électronique est un ensemble de données contenant au moins une clef publique, des informations d'identification et au moins une signature (clef privée). L'entité signataire (Autorité de Certification) est la seule autorité permettant de prêter confiance (ou non) à l'exactitude des informations du certificat.

Lorsqu'un diffuseur d'information veut diffuser une clef publique, il effectue une demande de signature auprès d'une autorité de certification (CSR). L'autorité de certification reçoit la clef publique et l'identité du diffuseur. Après avoir vérifié la clef publique et l'identité du diffuseur par des moyens conventionnels, elle les place dans un conteneur qu'elle signe en utilisant sa clef privée. Le fichier résultant est le certificat. Il est alors renvoyé au diffuseur qui le conserve pour ses communications (par exemple sur son site internet) avec des utilisateurs. Le protocole HTTPS permet d'utiliser ce mécanisme de certificat pour authentifier le serveur ou le client.

4.3 – Principaux services d'authentification

Comme décrit par (« Clément-Oudot_LDAP », s. d.), les services d'authentification font appel aux notions de cercle de confiance, de fournisseur d'identités (IDP) et fournisseur de services (SP). L'utilisateur qui possède plusieurs identités numériques peut les fédérer au sein d'un cercle de confiance et choisir son fournisseur d'identité. Le résultat visible est l'accès transparent aux fournisseurs de service, mais d'autres avantages existent, comme la déconnexion unique (SLO).

4.3.1 – CAS

D'après (« BENGLIA_BENHAMMOUDA.pdf », s. d.), CAS est un protocole d'authentification centralisé fournissant un service de SSO pour les applications Web, inspiré de Kerberos et basé sur le protocole HTTP(S). Il s'agit d'un des plus anciens modèles SSO. CAS pour Central Authentication Service a été développé par l'université de Yale aux Etats-Unis. Il s'agit d'un serveur d'authentification basé sur des routines java fonctionnant sur des moteurs de type Apache-Tomcat.

CAS ne traite pas les besoins liés aux autorisations (droits applicatifs), aux fédérations d'identités (pas de connecteur SAMLv2 ou OpenIdConnect) et au transport d'attributs. CAS ne permet pas de protéger des applications dites « sur étagère ». En outre, la base d'authentification est locale, au niveau de l'établissement. Les aspects inter-établissements ne sont donc pas pris en compte. Les mots de passe utilisateurs ne circulent qu'entre le navigateur et le serveur d'authentification à travers un canal sécurisé HTTPS. L'utilisation de cookies exclusivement privés dans CAS (passage de tickets entre serveur d'authentification et applications uniquement sous forme de paramètres de GET HTTP) permet à CAS d'être opérationnel sur des serveurs situés dans des domaines DNS différents.

Un module Apache (mod_cas) permet d'utiliser CAS pour protéger l'accès à des documents web statiques, les librairies clientes ne pouvant être utilisées dans ce cas. Un module PAM (pam_cas) permet de « CAS-sifier » des services non web, tels que FTP, IMAP. L'architecture de CAS est basée sur trois composants principaux : le serveur CAS, le navigateur web et le client CAS.

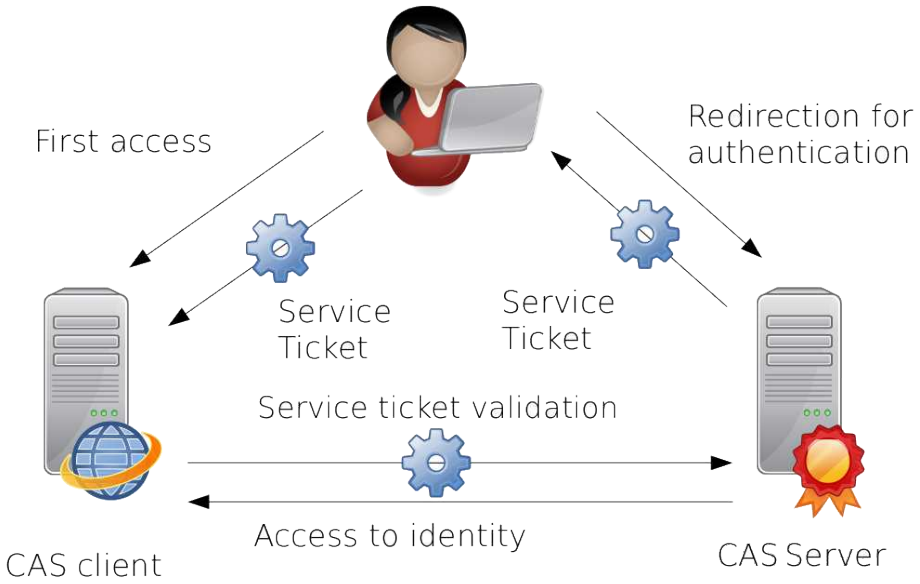

Figure 4 : Cinématique CAS

Exemples de requêtes :

- Requête ticket de service CAS :

https://auth.example.com/cas/login?service=http://auth.example.com/cas.pl

- Réponse ticket de service CAS :

http://auth.example.com/cas.pl?ticket=ST-6096f5d3ddb33df6fd79529e2d626a6d

- Requête validation ticket CAS :

https://auth.example.com/cas/serviceValidate?service=http://auth.example.com/cas.pl&ticket=ST-6096f5d3ddb33df6fd79529e2d626a6d

- Réponse validation ticket CAS version 2 :

```
<cas:serviceResponse xmlns:cas='http://www.yale.edu/tp/cas'>
  <cas:authenticationSuccess>
    <cas:user>maudoux</cas:user>
  </cas:authenticationSuccess>
</cas:serviceResponse>
```

L'authentification est centralisée sur un serveur CAS. Celui-ci est le seul acteur du mécanisme CAS qui relaye les mots de passe des utilisateurs à l’annuaire centralisé LDAP. Il a pour rôle d’authentifier les utilisateurs et de certifier l'identité de la personne authentifiée aux clients CAS par son ticket de service.

Le client CAS est une application Web embarquant une bibliothèque cliente ou un client CAS intégré. Il ne délivre les ressources qu'après s’être assuré que le navigateur se soit authentifié auprès du serveur CAS. Le client s’assure également à intervalle régulier de la validité de la session CAS.

Le fonctionnement de base est le suivant (cf. figure 4 ci-contre) :

- Authentification d’un utilisateur : Un utilisateur non authentifié, ou dont l’authentification a expiré, accède au serveur CAS. Ce dernier lui propose un formulaire d’authentification, dans lequel il est invité à saisir son identifiant et son mot passe.
- Si les informations d’authentification sont correctes, le serveur renvoie au navigateur un cookie appelé TGC (Ticket Granting Cookie). Le TGC est un cookie privé et sécurisé transmis uniquement au serveur CAS avec une durée de vie limitée à celle de la session du navigateur.
- Accès à une ressource protégée après l’authentification : Le navigateur tente d'accéder à une ressource protégée sur un client CAS. Le client redirige le navigateur vers le serveur CAS afin d'authentifier l'utilisateur. Le navigateur, précédemment authentifié auprès du serveur CAS, lui présente le TGC. Le serveur CAS délivre au navigateur un Service Ticket (ST). Il s’agit d’un ticket ne comportant aucune information personnelle. Il n'est utilisable que par le « service » (l'URL) qui en a fait la demande. Dans le même temps, le serveur CAS redirige le navigateur vers le service demandeur en passant le Service Ticket. Le Service Ticket est alors validé auprès du serveur CAS par le client CAS qui délivre la ressource au navigateur. Toutes les redirections sont transparentes pour l’utilisateur.
- Accès à une ressource protégée sans authentification préalable : Si l’utilisateur ne s’est pas authentifié auprès du serveur CAS avant d'accéder à une ressource web, son navigateur est redirigé vers le serveur CAS qui lui propose alors un formulaire d'authentification. Ensuite, suivent les mêmes étapes présentées précédemment.

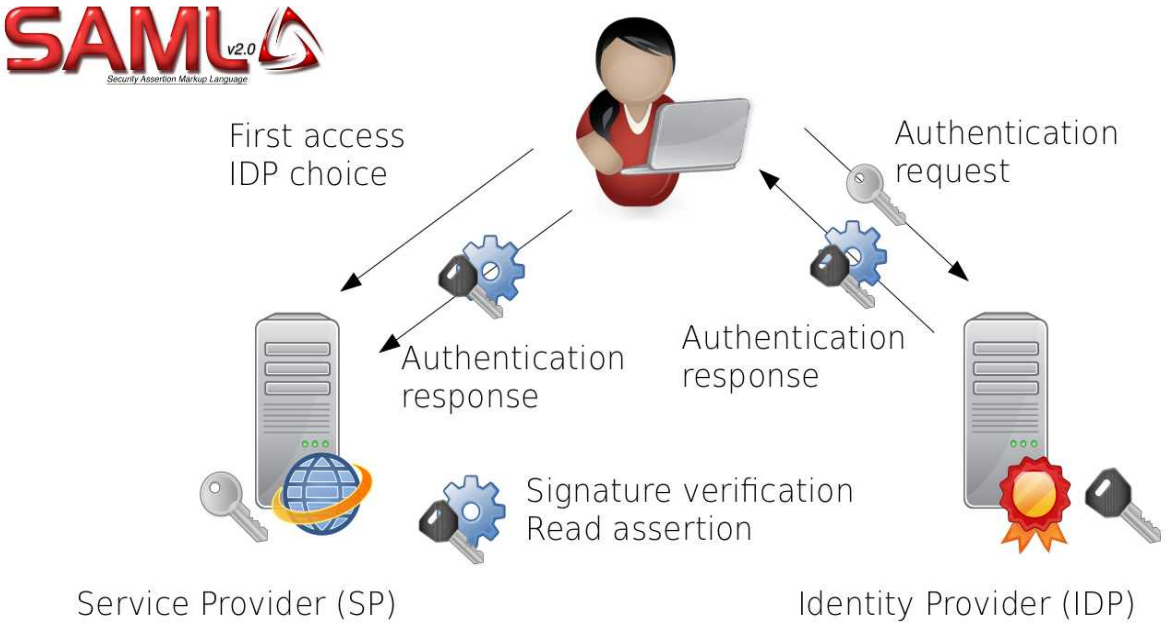

Figure 5-1 : Cinématique SAML2

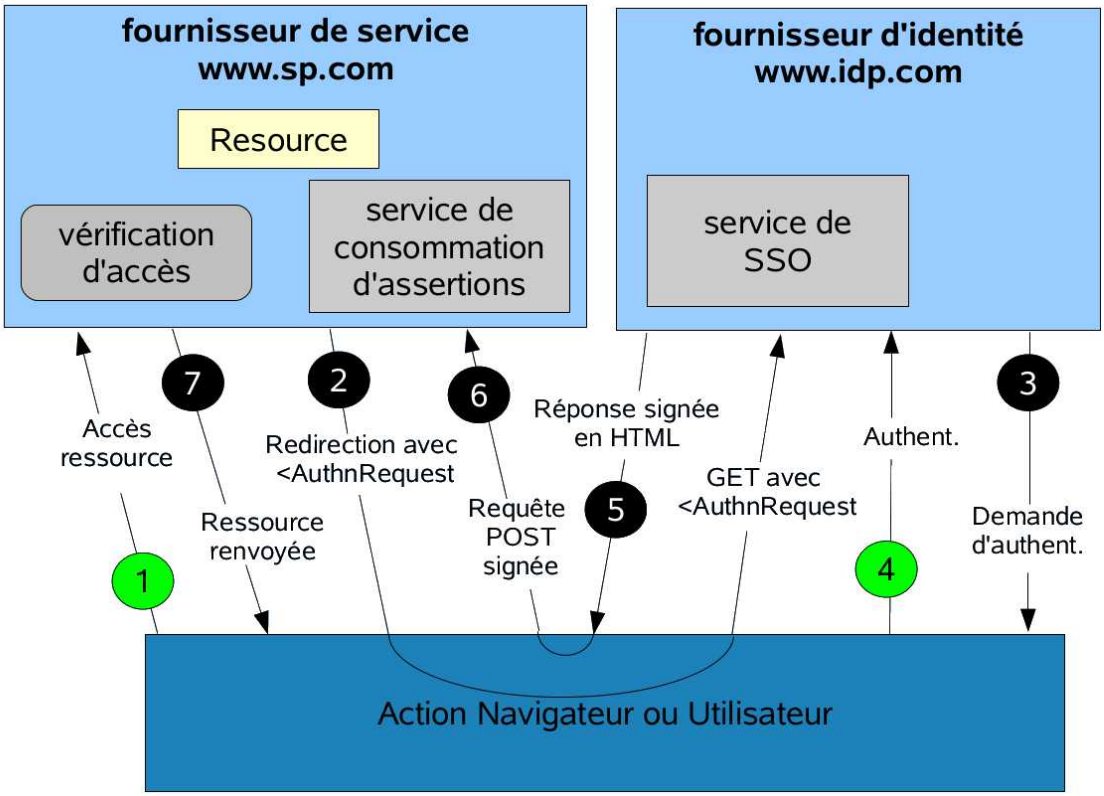

Figure 5-2 : Cinématique SAML2 DÉTAILLÉE

4.3.2 – SAML2 / Shibboleth

(« SAML et fédération d'identité », s. d.) Le protocole SAML a été créé par l'organisation OASIS en 2002. L'OASIS (Organization for the Advancement of Structured Information Standards) est un consortium mondial qui travaille pour la standardisation de formats de fichiers ouverts basés notamment sur XML. En 2002, les protocoles SAML, Shibboleth et ID-FF (Liberty Alliance) ont fusionné pour devenir SAML2. Il a pour objectifs de permettre l'authentification et l'échange d'informations d'identités de façon sécurisée. Il autorise la fédération entre fournisseurs de services et d'identités situés dans des organisations différentes.

SAML est basé sur la notion de cercle de confiance. Il impose l'enregistrement préalable des fournisseurs de services (Service Provider) et des fournisseurs d'identités (Identity Provider) par l'échange mutuel des métadonnées entre les partenaires. SAML autorise plusieurs méthodes : GET, POST, Artefact GET ou Artefact POST. Les méthodes « Artefact » permettent de passer des jetons en lieu et place de la requête. De plus, SAML permet de gérer des conditions ou des contextes d'authentification. Les applications peuvent également imposer des contraintes sur l'identification.

Le principe de fonctionnement est le suivant (cf figures 5-1 & 5-2). L'application initiale, le fournisseur de service, délègue l'authentification de l'utilisateur aux fournisseurs d'identités. Un fournisseur de services (FS) peut faire appel à plusieurs fournisseurs d'identités (FI). Lors de la phase suivante, l'utilisateur s'authentifie auprès du FI de son choix puis est redirigé vers le FS d'origine. Le FI transmet au FS une assertion SAML contenant l'ensemble des attributs nécessaires.

D'un point de vue technique, SAML définit la structure des échanges de messages au format XML, appelés les assertions. Il décrit également les cas d'utilisation en précisant les échanges des messages ainsi que les paramètres envoyés et reçus. De manière complémentaire, les relations de confiance techniques entre les fournisseurs s'appuient sur des certificats qui permettent ainsi de garantir des communications sécurisées entre eux. Les assertions SAML sont basées sur les couches SOAP, XML Encryption et XML Signature.

SOAP est le protocole d'encapsulation standard des messages XML, utilisé principalement par les Web services.

XML Encryption est le protocole standard de chiffrement des messages XML. Il a la particularité de pouvoir chiffrer la globalité du message ou simplement un sous-ensemble précis. Cela permet d'avoir par exemple un document XML en clair avec des valeurs d'attributs chiffrées.

XML Signature est le protocole standard de signature des messages XML. Tout comme *XML Encryption*, il permet de cibler l'élément à signer. Cela permet à plusieurs intervenants de signer chacun une partie différente du document XML.

SP et IdP sont deux entités qui ont connaissance chacune l'une de l'autre en termes d'identifiant et de certificat. Les messages XML qui transitent sur le réseau sont donc chiffrés par la clef publique du destinataire, seul capable de déchiffrer le message avec sa clef privée. L'émetteur signe ses assertions avec sa clef privée permettant au destinataire de vérifier sa provenance.

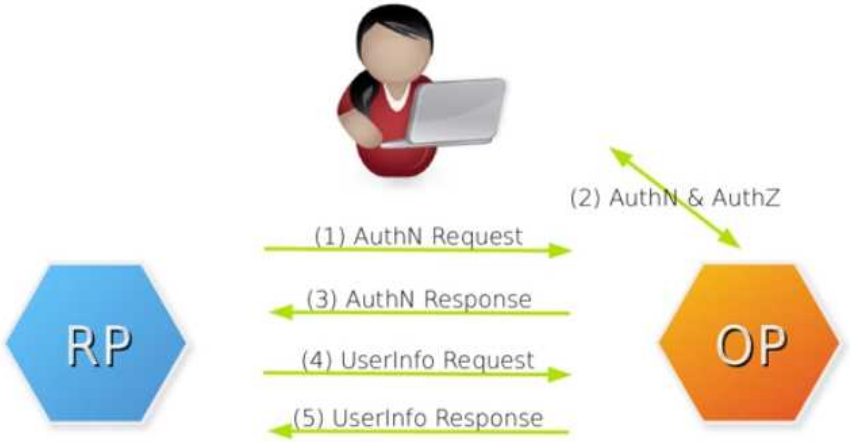

Figure 6.1 : Demande des informations utilisateur

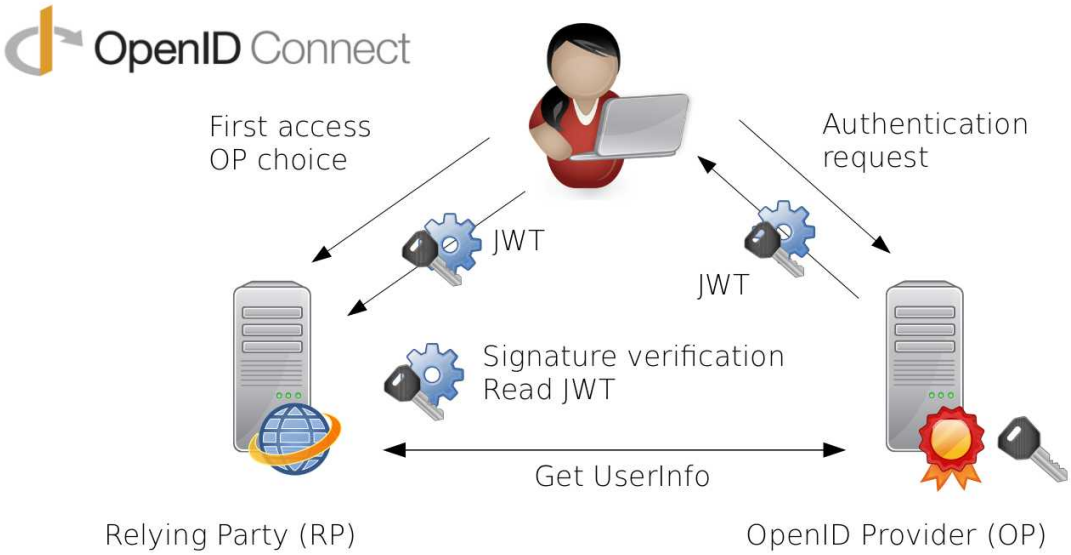

Figure 6.2 : Cinématique OpenID-Connect

Figure 6.3 : Json Web Token

4.3.3 – OpenID Connect

OpenID Connect *(OIDC)* (« OpenID Connect », 2018) est un protocole récent porté par la fondation OpenID. Il a été créé en 2014. Ce standard est adopté par de nombreux acteurs, comme par exemple Google qui l'utilise pour son authentification en remplacement de OpenID 2.0. L'État Français a lui aussi fait le choix de OpenID Connect pour sa nouvelle plate-forme France-Connect.

OpenID Connect fait appel aux notions de Relying Party et d'OpenID Provider. Le RP, l'application, se connecte à l'OP pour authentifier l'utilisateur. L'utilisateur s'authentifie et l'OP retourne la requête confirmant la réussite. Ensuite, le RP envoie une requête à l'OP pour demander les données de l'utilisateur et l'OP les lui retourne (cf. figure 6.1). Le protocole prévoit de notifier l'utilisateur sur les données transmises et d'obtenir son consentement.

OpenID Connect est une couche d'identification basée sur le protocole OAuth 2.0 et utilisant les dernières normes telles JSON en lieu et place de XML ou JWT (Json Web Token) encodé en base64 pour récupérer les informations utilisateurs (cf. figure 6.2). Le JWT est en fait constitué de trois JSON encodés en base64 concaténés avec des points : l'entête précisant la méthode de signature utilisée, les données utilisateur et la signature (cf. figure 6.3). OIDC autorise les clients à vérifier l'identité d'un utilisateur final en se basant sur l'authentification fournie par un serveur d'autorisation, en suivant le processus d'obtention d'une simple information du profil de l'utilisateur final. Ce processus est basé sur un dispositif interopérable de type REST.

OpenID Connect permet à un éventail de clients, y compris web, mobiles et JavaScript, de demander et recevoir des informations sur la session authentifiée et l'utilisateur final car il permet l'échange d'attributs d'identité. Cet ensemble de spécifications est extensible, supporte des fonctionnalités optionnelles telles le chiffrement des données d'identité, la découverte dynamique de fournisseurs OpenID et la gestion de sessions.

4.3.4 – Synthèse

CAS fût le deuxième protocole de SSO disponible sur le protocole HTTP après *Kerberos*. Il ne fournit qu'un service d'authentification et ne permet pas l'échange d'attribut. CAS est basé sur l'utilisation de tickets de service dans l'URL. Les applications doivent être préalablement « CAS-sifiées » pour joindre le serveur CAS. Ensuite, toutes les applications « CAS-sifiées » ont accès au serveur. Celui-ci ne les « connaît » pas car elles n'ont pas à être déclarées (mode ouvert). Ceci est très pratique à déployer mais pas très sécurisé. Il est plutôt adapté et réservé à un usage interne.

SAML autorise le partage d'identités entre organismes et l'échange d'attributs. La notion de cercle de confiance impose aux fournisseurs de service et d'identité de s'enregistrer mutuellement par échange de leurs métadonnées. Les assertions SAML transmises sont chiffrées ou signées à l'aide des clefs publiques publiées dans les métadonnées. Il s'agit d'un protocole complexe de fédération très sécurisé et répandu dans les administrations. Il est également très utilisé dans les

solutions SaaS. SAML est orienté sécurité mais travaille en mode connecté. Il faut utiliser un navigateur ou une application web pour le mettre en œuvre.

OpenID Connect est un protocole basé sur les dernières innovations. Il utilise le même principe de fonctionnement que SAML mais avec le format JSON en lieu et place du XML. Les messages JSON peuvent être signés ou chiffrés par clefs publiques (publiées dans le JWKS) ou symétriques par échange d'un secret. Il permet, comme pour SAML, de récupérer des attributs sur l'utilisateur au format JSON grâce à une API. Il est plutôt tourné vers les applications grand public. Comme pour SAML, les parties RP et OP doivent s'échanger leurs métadonnées mais il offre la possibilité de l'auto-enregistrement. OIDC permet de travailler en mode déconnecté. Les applications peuvent conserver un « refresh token » et demander un nouveau jeton de session. C'est une fonctionnalité surtout utilisée par les applications mobiles. De plus, OIDC expose une API REST permettant la connexion avec des jetons. De ce fait, le couple identifiant / mot de passe n'est jamais transmis aux applications.

4.4 – Règlement EIDAS

4.4.1 – Présentation & Objectifs

Le règlement européen « eIDAS » n°910/2014 du 23 juillet 2014 a pour ambition d'accroître la confiance dans les transactions électroniques au sein du marché intérieur. Il établit un socle commun pour les interactions électroniques sécurisées entre les citoyens, les entreprises et les autorités publiques.

Il concerne principalement les organismes du secteur public et les prestataires de services de confiance établis sur le territoire de l'Union européenne. Il instaure un cadre européen en matière d'identification électronique et de services de confiance, afin de faciliter l'émergence du marché unique numérique.

Pour ce faire, le règlement formule des exigences relatives à la reconnaissance mutuelle des moyens d'identification électronique ainsi qu'à celle des signatures électroniques, pour les échanges entre les organismes du secteur public et les usagers.

Afin de pouvoir bénéficier de cette reconnaissance mutuelle, un moyen d'identification électronique doit :

- Avoir été délivré conformément à un schéma d'identification électronique notifié par l'Etat membre concerné et figurant sur la liste publiée par la Commission. Un schéma d'identification électronique est un système pour l'identification électronique en vertu duquel des moyens d'identification électronique peuvent être délivrés à des personnes physiques ou morales.

- Avoir un niveau de garantie égal ou supérieur à celui requis par l’organisme du secteur public concerné pour accéder à ce service en ligne, à condition que ce niveau soit substantiel ou élevé.

Les exigences applicables aux différents niveaux de garantie qui sont prévus par le règlement sont détaillées dans le règlement d’exécution n°2015/1502 du 8 septembre 2015. D’après le règlement (« CELEX_32015R1502_FR_TXT.pdf », s. d.), ces niveaux sont accordés en fonction du respect de spécifications, normes et procédures minimales. Il définit trois niveaux de garantie :

- Faible : l’objectif est simplement de réduire le risque d’utilisation abusive ou d’altération d’identité
- Substantiel : l’objectif est de réduire substantiellement le risque d’utilisation abusive ou d’altération d’identité
- Élevé : l’objectif est d’empêcher l’utilisation abusive ou l’altération de l’identité

4.4.2 – Niveaux d’authentification

J’aborderai dans ce chapitre uniquement les conditions relatives aux personnes physiques et les moyens d’identification utilisables sur un SI.

Trois pré-requis sont indispensables avant de pouvoir générer et délivrer un moyen d’identification électronique. Il faut s'assurer que le demandeur soit informé des conditions associées à l'utilisation du moyen d'identification électronique, des précautions de sécurité recommandées relatives à ce moyen et recueillir les données d'identité pertinentes requises pour la preuve et la vérification de son identité.

Les différents niveaux eIDAS exigent des éléments nécessaires à la preuve et à la vérification de l’identité. Pour atteindre le niveau :

- eidas1 : niveau faible (exemple : authentification par identifiant / mot de passe)
  La personne peut être présumée en possession d'un élément d'identification

- eidas2 : niveau substantiel (exemple : authentification à double facteur)

Doit être de niveau faible et il a été vérifié que la personne est en possession d'un second élément d'identification. Celui-ci doit être unique, faire l'objet d'une vérification visant à déterminer son authenticité et uniquement délivré à son propriétaire. Celui-ci repose sur le principe d'une authentification dynamique : OTP, challenge et réponse correspondante.

L’authentification avec second facteur peut, en fonction du type, être qualifiée de niveau substantiel ou fort. Il est nécessaire pour cela de prendre en compte la notion de non-répudiation réalisée uniquement avec un système basé sur des clefs privées et publiques ce qui n’est pas le cas d’un OTP par exemple comme précisé par (« Authentification forte », 2018a).

- eidas3 : niveau fort (exemple : utilisation de certificats ou de cartes à puce qui sont également un second facteur mais dont le mot de passe ne circule pas)

  Doit être de niveau substantiel et soit :

1. il a été vérifié que la personne est en possession d'un élément d'identification biométrique ou photographique et le demandeur est identifié comme ayant l'identité alléguée par comparaison d'une ou de plusieurs caractéristiques physiques de la personne auprès d'une source faisant autorité

   ou

2. lorsque des moyens d'identification électronique sont délivrés sur la base d'un moyen d'identification électronique notifié valide ayant le niveau de garantie élevé et des mesures sont prises pour prouver que les résultats de cette précédente procédure de délivrance d'un moyen d'identification électronique notifié demeurent valides (IGC, PKI).

5 – Synthèse

Dans cette première partie, j'ai introduit les notions d'authentification et les niveaux normalisés eIDAS. Pour ce faire, j'ai présenté les enjeux liés à l'authentification unique, comparé deux solutions SSO complètes AAA puis rappelé quelques principes de base.

Depuis maintenant une quinzaine d'années, la Gendarmerie nationale a mis un place un système d'authentification unique pour permettre à ses personnels d'accéder de façon sécurisée aux nombreuses ressources (applications, fichiers ou services, Internet) disponibles sur son Système d'Information mais également de maîtriser et contrôler les actions des utilisateurs**,** de manière à assurer une traçabilité permettant d'imputer des actions à leurs auteurs. Cette démarche s'inscrit dans un cadre très strict en termes de contraintes légales, de performances et de limitation des coûts.

Pour ce faire, la Gendarmerie Nationale a choisi de déployer LemonLDAP::NG puis a été imitée par beaucoup d'autres ministères ou administrations. LemonLDAP::NG est devenu le produit SSO AAA prédominant au sein de l'état français.

Je présenterai et détaillerai donc cette solution dans la seconde partie de ce mémoire.

B – LemonLDAP::NG

Figure 7 : Logo LemonLDAP::NG

Au 15 septembre 2018, la dernière version stable de LemonLDAP::NG est la 1.9.17. La version 2.0.0, prévue pour le second semestre 2018, est actuellement en cours de finalisation et disponible en version « bêta2 » pour tests. Il est important de préciser qu'entre la 1.9.17 et la 2.0.0, le *Portail* et une partie du *Manager* ont été réécrits ainsi que les différents agents logiciels.

N'ayant travaillé que sur LLNG 2.0, la suite de ce mémoire sera consacrée uniquement à cette version sauf cas particuliers explicitement précisés.

1 – Le projet

1.1 – Présentation

Lemonldap::NG est un projet d'infrastructure. Le seul aspect visible par les utilisateurs, et ce uniquement quelques secondes, est le Portail auprès duquel ils s'authentifient. LLNG est une solution essentiellement Web-SSO mais pouvant également servir des applications avec client lourd. LLNG peut fonctionner en mode mandataire inverse (reverse-proxy), applicatif, fédération d'identités ou mixte.

LLNG est compatible avec la plupart des protocoles de fédération d'identité qu'ils soient modernes (OpenId Connect), courants (SAML, CAS) ou dépréciés (OpenId, Shibboleth). Il peut être fournisseur de services ou d'identités. Une autre particularité spécifique et peu courante est la possibilité d'utiliser LLNG comme relais protocolaire *(Protocol Relaying Proxy)* entre ces différents systèmes de fédération d'identités très hétérogènes ou entre deux instances de LemonLDAP::NG quelle que soit la version..

LLNG gère et propose les services d'authentification et d'autorisation. Il fournit également des en-têtes HTTP aux applications protégées pour qu'elles puissent exploiter des informations d'identité ou autres. Ces en-têtes permettent également de contrôler les accès et d'imputer les actions aux utilisateurs. LLNG est donc une solution AAA complète.

Les autorisations d'accès sont construites par association d'une expression régulière et d'une règle. Les expressions régulières utilisent le moteur interne de Perl (PCRE) et sont appliquées à chaque requête HTTP (URL). La règle est calculée pour déterminer si l'utilisateur est autorisé ou non à accéder à la ressource demandée.

Toutes les demandes d'ouvertures de sessions et les accès aux applications protégées sont tracés et journalisés.

LLNG est entièrement écrit en Perl et JavaScript (avec les framework AngularJS et Jquery). Il utilise des briques logicielles et est distribué uniquement sous licence GPL. LLNG fonctionne sous environnement UNIX avec les serveurs web Apache, Nginx (protocoles FastCGI / uWSGI) ou compatibles Plack (Starman, Corona, etc …).

1.2 – Equipe de développement

Fondé par Xavier Guimard en 2004, LemonLDAP::NG est principalement co-développé par Clément Oudot et David Coutadeur. En outre, plusieurs contributeurs occasionnels ont également participé à son développement.

Le Colonel Xavier Guimard, Ingénieur diplômé de l'Ecole Polytechnique, est sous-directeur au sein du Service des Technologies et des Systèmes d'Information de la Sécurité Intérieure depuis 2014 et chef de 80 ingénieurs, techniciens, gendarmes et policiers en charge du contrôle d'accès, des référentiels, de la cartographie opérationnelle, de l'architecture générale, de la mobilité, du traitement des appels d'urgence et des salles de crises et de commandement.

Clément Oudot, Ingénieur diplômé de l'école Télécom SudParis, est expert dans le domaine de la gestion et de la fédération des identités. Chargé du développement de l'agence Linagora Lyon puis expert infrastructure et sécurité chez Savoir-Faire Linux, il est actuellement en poste chez Worteks. David Coutadeur est Ingénieur, expert LDAP, chez Linagora depuis 2010.

1.3 – Historique & Versions majeures

(« Tous les contenus taggés avec lemonldap - LinuxFr.org », s. d.)

1.3.1 – Version 1.0 du 1er décembre 2010

Il s'agit de la première version stable publiée de LemonLDAP::NG. Son point fort est l'intégration des protocoles SAML, OpenID et CAS en complément des fonctionnalités SSO existantes. Cette version 1.0 constitue une avancée unique dans l'interopérabilité des SSO. LLNG peut simultanément assurer les fonctions de SSO, de client SAML, OpenID, CAS, Twitter et fournisseur d'identité avec SAML, OpenID et CAS.

Les principales fonctionnalités de cette version 1.0 sont :

- SSO avec contrôle d'accès basé sur l'association entre expressions régulières et expressions booléennes
- En-têtes HTTP personnalisables par site
- Dispositif de notification : les messages sont affichés à la connexion de l'utilisateur
- Explorateur des sessions actives
- Gestionnaire de configuration entièrement graphique
- Authentification sur Twitter, OpenID, SAML, CAS ou un autre LemonLDAP::NG
- Authentification sur ces protocoles par règle ou au choix de l'utilisateur
- Fournisseur d'identité SAML, OpenID, CAS activables simultanément au choix de l'administrateur
- Intégration simple de nombreuses applications : Mediawiki, Zimbra, Sympa, OBM, Tomcat
- SAML basé sur la librairie Lasso (publiée sous licence GPL par Entr'ouvert) (« Lasso - Free Liberty Alliance Single Sign On », s. d.)

1.3.2 – Version 1.4 du 30 juin 2014

Les principaux apports de cette nouvelle version sont tout d'abord un nouveau thème graphique du *Portail* basé sur *Bootstrap*. Celui-ci étant *responsive* (adaptatif), il permet d'utiliser LLNG depuis tout type de périphérique (téléphone, tablette, poste de travail). En outre, de nombreux thèmes sont disponibles sur Internet permettant d'en changer l'apparence, simplifiant ainsi la personnalisation de l'interface Web de LLNG. *Bootstrap* facilite également la gestion des feuilles de styles CSS par les développeurs.

Ensuite, cette version 1.4 propose plusieurs fonctionnalités comme la possibilité de réinitialiser son mot de passe lorsqu'il a été perdu ou de pouvoir créer un compte si l'on n'en possède pas. La démarche est simple, calquée sur la plupart des services similaires proposés par des sites en ligne :

1. L'utilisateur remplit un formulaire avec nom, prénom et adresse mail
2. Il reçoit un message avec un lien sur lequel cliquer
3. Après avoir cliqué, il reçoit un message lui confirmant la création de son compte, avec ses informations de connexion

Bien entendu cette fonctionnalité a peu d'intérêt en entreprise ou au sein d'administrations, où les utilisateurs sont déclarés dans un annuaire central et ne peuvent choisir de s'inscrire ou non au service. Toutefois, cela offre désormais la possibilité de protéger des services en ligne par LemonLDAP::NG.

Autre modification majeure de la 1.4 est l'utilisation du module Perl Mouse. « Mouse » apporte une sur-couche orientée-objet à Perl permettant une manipulation haut niveau des instances. Ce module est utilisé à partir de cette version de LemonLDAP::NG, ce qui a entraîné beaucoup de réécriture de code, avec un objectif de meilleure maintenabilité. Ces changements sont visibles essentiellement des développeurs, mais ils ont tout de même permis quelques améliorations comme :

- une gestion unifiée du cache des sessions quel que soit le type (SSO, CAS, SAML, etc...)
- la centralisation des valeurs par défaut des attributs de configuration
- la simplification des appels aux *Handler*s dans la configuration Apache se réduisant à une simple directive d'appel dans chaque hôte virtuel

Les dernières modifications concernent le fonctionnement de LLNG. Le principe du SSO est que l'accès aux différentes applications est possible à l'aide d'un jeton, en l'occurrence pour LemonLDAP::NG, une clef de session stockée dans un cookie. Par défaut, les identifiants de session sont générés par la fonction de hashage SHA1 avec une taille de 32 caractères. Cela peut poser des problèmes de sécurité, en particulier si un attaquant tente de découvrir un identifiant de session par force brute. Cette fragilité n'est pas spécifique au SSO. Ce type d'attaque est possible sur toute application utilisant un cookie de session. Toutefois dans le cas du SSO, la découverte de la session permet l'accès à toutes les applications autorisées. A partir de la 1.4, LLNG offre la possibilité de choisir son propre module de génération d'identifiants de sessions et propose un module basé sur SHA256, créant des clefs sur 64 caractères.

S'agissant du SAML, le mode SSO initié par le fournisseur d'identité (SSO IDP initiated) est désormais disponible. Il permet de générer des URLs sur le *Portail* LLNG (agissant comme IDP SAML) qui transmettent directement une réponse SAML à un fournisseur de service (SP). Cela permet une authentification en deux étapes (authentification sur l'IDP puis accès au SP) au lieu des trois étapes du processus standard (accès au SP, authentification sur l'IDP puis retour sur le SP). En outre, la 1.4 permet également une configuration plus fine des dates utilisées dans les messages SAML (durée de validité des assertions et des sessions).

Pour terminer, c'est à partir de cette version que LLNG propose un agent (*Handler*) pour le serveur web Nginx. Depuis ses origines, LemonLDAP::NG a été conçu pour fonctionner avec Apache. Une première implémentation du *Handler* en LUA a été proposée mais cette implémentation ne permet pas de couvrir la totalité des fonctionnalités offertes par l'agent standard Apache (gestion des expressions régulières, de l'interception des déconnexions, etc...).

1.3.3 – Version 1.4.6 du 9 octobre 2015

Cette version contient des correctifs importants et quelques nouvelles fonctionnalités. Elle apporte notamment la gestion du temps d'inactivité. Le code du *Handler* Apache a été réécrit. Une de ses nouvelles fonctionnalités est la mise à jour d'un attribut de la session afin de mesurer l'activité d'un utilisateur. Si l'utilisateur est inactif trop longtemps, sa session peut être fermée et ainsi l'obliger à se ré-authentifier.

Ensuite, la version 1.4.6 propose le chaînage des modules d'authentification. LLNG propose de nombreux modules d'authentification (LDAP, Radius, OpenID, SAML, CAS, etc…) qui peuvent être chaînés entre eux : si l'un échoue, le suivant est testé. Il est possible de chaîner plusieurs modules du même type (plusieurs annuaires LDAP par exemple), dans ce cas un label est ajouté au nom du module choisi (LDAP#labelA ; LDAP#labelB). Le nom des modules contenant le label est désormais conservé dans des attributs de sessions différents de ceux utilisés pour connaître le module d'authentification utilisé pour se connecter, afin que le label ne provoque pas d'erreur lors de l'appel à certaines fonctions.

Un autre apport de cette version est le stockage des groupes en session. LLNG permet de rechercher dans un annuaire LDAP les groupes auxquels appartient un utilisateur, et ce de manière récursive (gestion des groupes de groupes). Il est même possible de stocker en session plusieurs attributs du groupe (son identifiant, son nom, sa description, etc.). Cependant, si l'un de ces attributs était multivalué, seule une valeur était conservée en session. Ce comportement a été modifié, introduisant une nouvelle représentation des groupes dans la session, sous forme de hash Perl. La nouvelle variable de session se nomme $hGroups et coexiste avec la variable historique $groups qui conserve une syntaxe de chaîne de caractères. Cette nouvelle syntaxe permet une écriture plus élégante des règles d'accès. Par exemple, pour autoriser uniquement les utilisateurs du groupe « admin », il est possible d'écrire : defined $hGroups{'admin'}. Pour rechercher une valeur particulière d'un attribut du groupe, la fonction « groupMatch » a été ajoutée.

Enfin, la politique des mots de passe avec *Active Directory* (AD) a été améliorée. LLNG utilise l'AD comme un annuaire LDAP pour authentifier les utilisateurs. L'intégration avec Active Directory s'améliore avec la possibilité de notifier les utilisateurs en cas d'expiration proche de leur mot de passe. Cela s'ajoute à la gestion déjà existante de la réinitialisation du mot de passe à la prochaine connexion. Ainsi le *Portail* LLNG peut tout à fait permettre aux utilisateurs de gérer leur mot de passe AD depuis leur navigateur web, sans être raccordés au domaine.

1.3.4 – Version 1.9.0 du 2 mars 2016

Cette version majeure offre le support du protocole OpenID Connect et permet donc d'utiliser LemonLDAP::NG pour s'authentifier sur ces services. En plus du rôle de client (Relying Party), LLNG a aussi le rôle de serveur (OpenID Provider) ce qui lui permet d'authentifier toute application compatible avec ce protocole.

L'autre principale nouveauté de la 1.9 concerne l'interface d'administration, appelée *Manager*. Elle a été réécrite en AngularJS afin d'améliorer son ergonomie et son utilisation. Cette nouvelle interface permet la navigation entre les différentes versions de la configuration, l'import ou l'export de configurations (au format JSON) et la duplication d'hôtes virtuels. L'interface propose toujours un explorateur de sessions mais ajoute la possibilité de parcourir les sessions persistantes (sessions conservées entre chaque connexion d'un utilisateur) et conserve le gestionnaire de notifications.

Ensuite, et peut-être le plus important apport de cette version est le support du serveur web Nginx. La réécriture du code de l'agent passant par l'implémentation d'une API a permis d'offrir un support du protocole FastCGI. Un petit serveur FastCGI est livré avec LemonLDAP::NG et permet de contacter le *Portail*, le *Manager* et le *Handler* depuis le serveur Nginx.

Enfin, la 1.9 est compatible en partie avec le protocole CAS 3.0 et offre la possibilité de partager des attributs dans les réponses de validation CAS. Cela permet en particulier d'utiliser l'API PHP-CAS et sa fonction getAttributes(). Ces Web-Services permettent de s'authentifier (et donc de créer la session SSO correspondante), consulter et modifier les sessions existantes, consulter et modifier la configuration. Avec cette version, MongoDB est utilisable pour le stockage des configurations et des sessions. Il s'ajoute à la liste des nombreux backends disponibles : fichier (JSON), DBI (MySQL, PgSQL, SQLite, …), LDAP, Redis, etc …

1.3.5 – Version 2.0 prévue le second semestre 2018

Cette nouvelle version apportera, entre autres nouveautés, le support de l'authentification à double facteurs (clefs U2F, TOTP, Yubikey ou externes comme l'envoi de SMS). LLNG proposera un moteur pour la gestion de l'authentification et des seconds facteurs entièrement configurables. En effet, l'administrateur pourra choisir d'activer ou non un type particulier de second facteur pour l'ensemble des utilisateurs ou pour un utilisateur en particulier ou encore d'autoriser l'auto-enregistrement via le *Manager*. De plus, il sera possible, par exemple, d'imposer l'enrôlement d'une clef U2F avant de pouvoir enregistrer un TOTP. Un module spécifique permettra de gérer les sessions avec double facteur.

Ensuite, elle fournira un *Handler* pour les applications Node.js ainsi qu'un agent « DevOps » orienté « SSO as a Service ». LLNG pourra également fonctionner sur des serveurs web PSGI comme Starman ou Corona.

Enfin, cette version 2.0 inaugure un nouveau système de journalisation qui offrira une grande souplesse dans la gestion des logs. Celui-ci permettra d'utiliser différents modules de journalisation comme *log4perl*, *sentry*, *syslog* et de séparer les journaux techniques des journaux d'utilisation en cinq niveaux.

1.4 – Fonctionnalités AAA

LLNG garantit l'indépendance des mécanismes d'authentification et de recherche des données de l'utilisateur.

La phase d'authentification permet de valider la création de la session SSO. Elle peut être effectuée par toute méthode d'authentification disponible.

La phase de recherche des données correspond à la récupération des informations propres à l'utilisateur (nom, mail, etc...) qui serviront d'une part à valider les règles d'accès et d'autre part à approvisionner les applications protégées.

1.4.1 – Authentification

Le processus d'authentification est composé des étapes suivantes :

- Contrôle l'URL demandée pour prévenir les attaques XSS (injection de code) (« Cross-site scripting », 2018) et les mauvaises redirections
- Vérifie si une session est déjà existante et application des éventuelles contraintes (1 session par utilisateur, par adresse IP, etc ...)
- Récupère les informations de connexion : couple identifiant / mot de passe, certificat ou variables d'environnement en fonction du module d'authentification choisi
- Collecte les informations de l'utilisateur en interrogeant la base de données
- Demande le second facteur si activé
- Calcule les macros
- Interroge la base de données pour déterminer les groupes
- Calcule les groupes locaux
- Interroge la base de données des utilisateurs pour vérifier l'identité
- Vérifie si l'utilisateur est autorisé à ouvrir une session SSO (plage horaire, IP source, …)
- Sauvegarde les informations de l'utilisateur dans la base de données des sessions
- Construit un cookie de session contenant l'Id de session
- Redirige l'utilisateur vers l'application protégée ou vers le *Portail* et le menu des applications

1.4.2 – Autorisation

Les autorisations sont vérifiées par l'agent pour chaque requête HTTP. Elles sont définies par une règle d'accès pour une URL ou pour toutes, par défaut. Les autorisations sont renseignées pour chaque hôte virtuel (VH) et ne s'appliquent qu'au VH dans lequel elles sont déclarées. Il n'y a pas d'autorisation globale hormis la règle d'ouverture de session sur le *Portail*.

LemonLDAP::NG peut transmettre des en-têtes HTTP aux applications protégées. Un en-tête (*header*) est défini par un nom et une valeur. Comme pour les autorisations, les en-têtes sont définis pour chaque hôte virtuel et ne s'appliquent qu'au VH dans lequel ils sont positionnés. Il n'y a pas d'en-tête global.

Les différentes règles d'accès peuvent être :

- *accept* : tous les utilisateurs authentifiés sont autorisés à accéder
- *deny* : interdit pour tout le monde
- *skip* : aucun contrôle ni envoi d'en-tête
- *unprotect* : tout le monde peut accéder mais les utilisateurs authentifiés sont vus comme tels. Envoi des en-têtes pour les utilisateurs authentifiés uniquement.
- *logout_sso, logout_app, logout_app_sso* : afin d'intercepter les requêtes de déconnexion
- une expression Perl : du code Perl retournant la valeur '0' ou '1'

1.4.3 – Accès

Tous les accès sont journalisés et peuvent être exploités par une application tierce.

- Journalisation des accès au *Portail* : le *Portail* génère une trace dans les journaux du serveur web pour tout accès, tentative d'accès et déconnexion
- Journalisation des accès aux applications protégées : l'agent LLNG informe le serveur web qu'un utilisateur a accédé à l'application et le serveur enregistre l'accès ainsi que le paramètre «_whatToTrace », clef unique configurée par l'administrateur pour identifier les utilisateurs. «_whatToTrace » est lié, « mappé » à un attribut comme « uid, nom, mail » par exemple. Historiquement, c'est ce paramètre qui était utilisé par le serveur web Apache pour journaliser les connexions. Les transactions et actions effectuées au sein de l'application doivent être gérées par l'application elle-même
- Journalisation des actions dans l'application protégée : LLNG fournit aux applications les éléments d'identité à insérer dans leurs traces

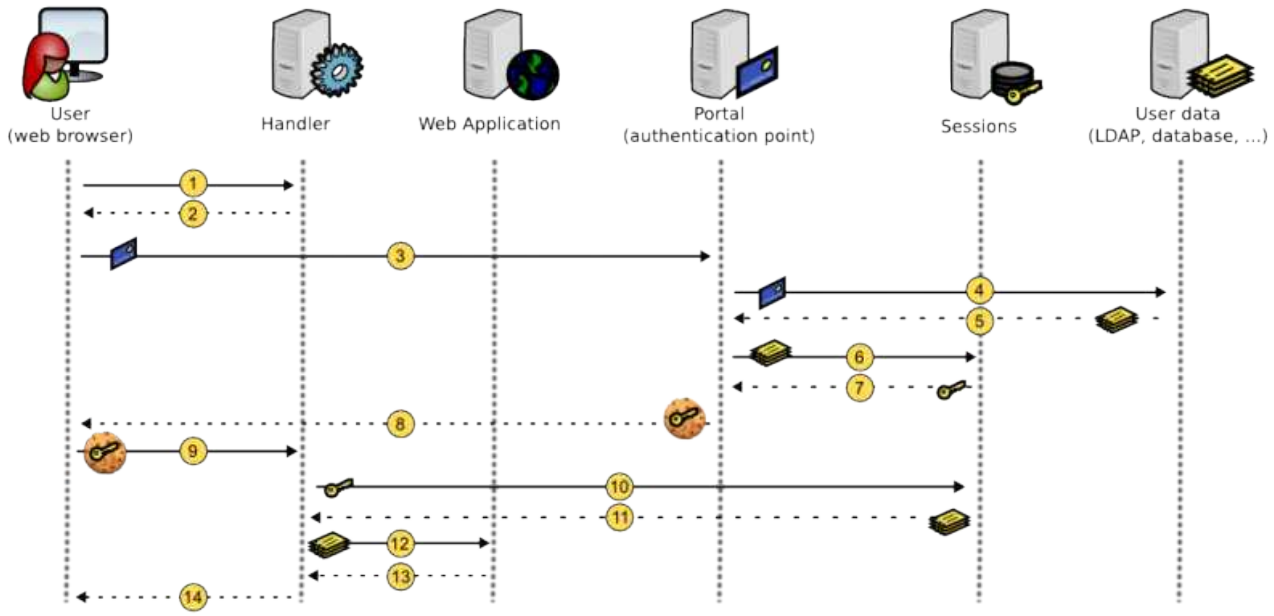

Figure 8 : Cinématique du contrôle d’accès

2 – Cinématique & Contrôle d'accès

Les informations ci-dessous sont issues de la documentation en ligne : (« LemonLDAP::NG - Open Source Web Single Sign On », s. d.)

2.1 – Schéma

Cf. figure N°8 ci-contre.

2.2 – Déroulé

1. Un utilisateur tente d'accéder à une application protégée. Sa requête est interceptée par le *Handler*
2. Si le *Handler* ne détecte pas de cookie SSO, il redirige l'utilisateur vers le *Portail* afin qu'il s'authentifie
3. L'utilisateur s'authentifie auprès du *Portail*
4. Le *Portail* vérifie l'authentification
5. Si l'utilisateur s'authentifie avec succès, le *Portail* collecte les informations sur l'utilisateur auprès de la base des utilisateurs
6. Le *Portail* crée ensuite une session pour sauvegarder les informations sur l'utilisateur
7. Le *Portail* récupère la clef de session
8. Le *Portail* crée un cookie SSO contenant uniquement la clef de session obtenue précédemment
9. L'utilisateur est redirigé vers l'application protégée muni de son cookie de session
10. Le *Handler* récupère la clef de session transmise par le cookie SSO et collecte les informations utilisateur
11. Le *Handler* sauvegarde dans son cache (15s) les données de l'utilisateur
12. Le *Handler* vérifie les règles d'accès et transmet les informations de l'utilisateur à l'application
13. L'application envoie la réponse au *Handler*
14. Le *Handler* transmet la réponse de l'application à l'utilisateur

Ensuite, le *Handler* vérifiera le cookie SSO et le droit d'accès à chaque requête HTTP.

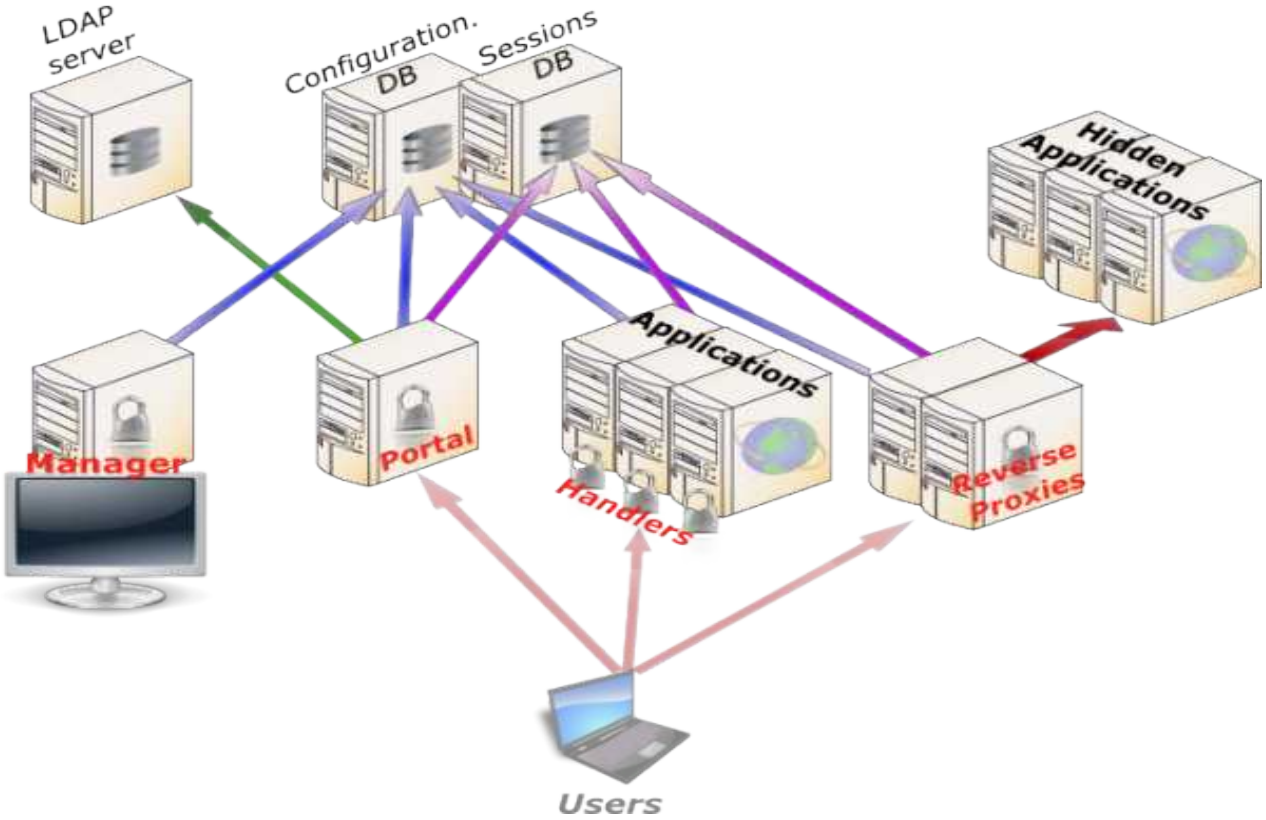

Figure 9 : Architecture globale des composants

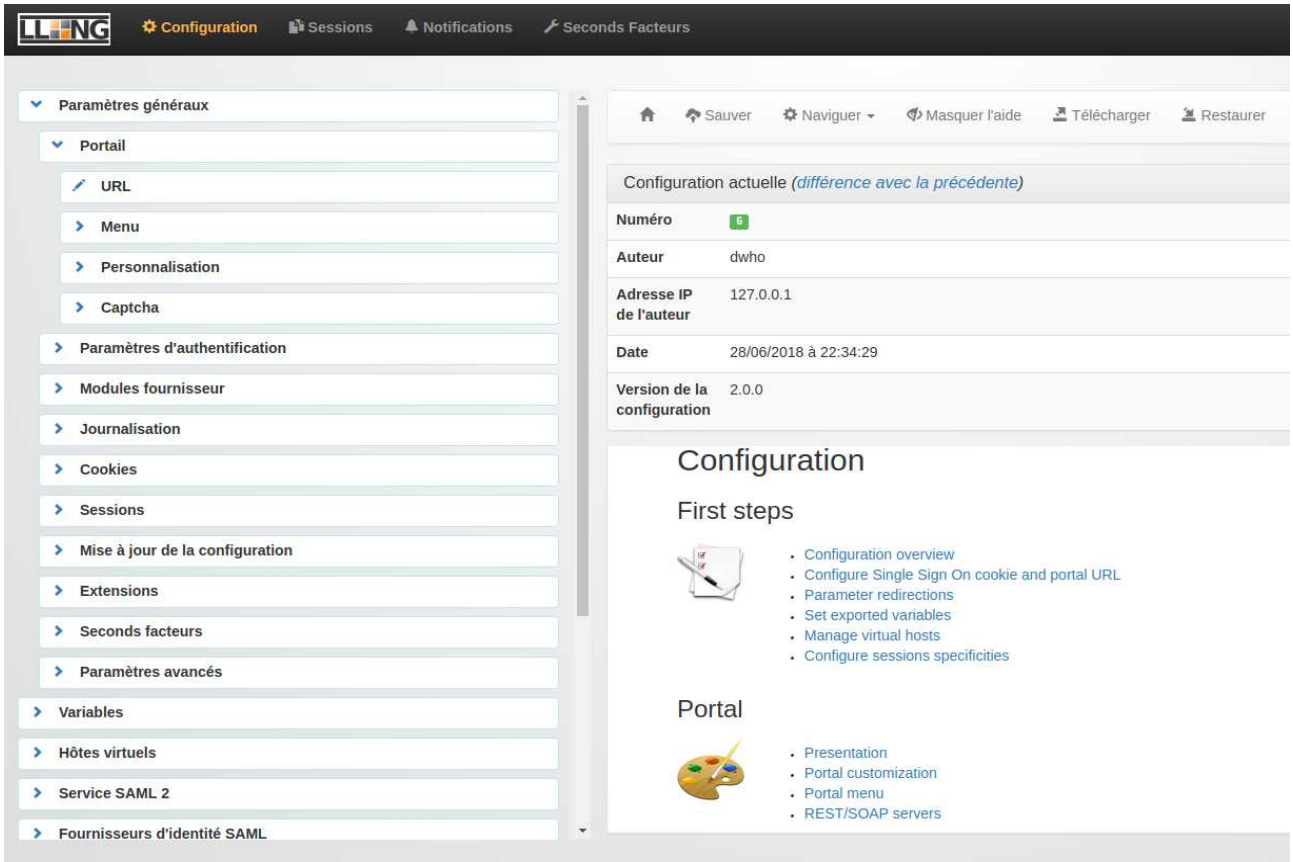

Figure 10 : *Manager*

3 – Principaux composants

3.1 – Implémentation

Cf. figure N°9 ci-contre.

3.2 – Manager

Cf. figure N°10 ci-contre.

Le *Manager* est l'interface d'administration graphique de LemonLDAP::NG. Il permet d'éditer la configuration générale ainsi que les droits d'accès et les en-têtes spécifiques à chaque application protégée. Le *Manager* permet d'accéder à l'explorateur de sessions qui offre une vue hiérarchique des sessions (par identifiant ou adresse IP) et affiche le contenu de chacune d'elles. Depuis l'interface du *Manager*, il est également possible d'ouvrir le gestionnaire de notifications et l'explorateur dédié aux sessions avec seconds facteurs. La *Manager* étant une application web à part entière, il peut être également géré par le WebSSO. A la différence des autres applications, il n'est pas nécessaire de déclarer le *Handler* dans le VH du *Manager* car il hérite de la classe du *Handler*. Le *Manager* est donc une application auto-protégée. En outre, le *Manager* est une application à page unique (SPA) écrite en Perl compatible PSGI car basée sur Plack et utilisant AngularJS afin d'en améliorer son ergonomie.

Contrairement aux anciennes version de LLNG, il n'est plus nécessaire de cliquer sur "Appliquer" à chaque changement de valeur et il est possible d'éditer tous les en-têtes ou toutes les règles dans un seul et même écran.

LLNG propose également une Interface en Ligne de Commande (CLI) afin de déployer des installations par scripts. Ceci est plus particulièrement destiné aux intégrateurs.

3.3 – Portail

Cf. figure N° 11.

Le *Portail* est le composant principal de LLNG. Il est également écrit en Perl/PSGI et JavaScript (Jquery). Il est construit autour de plusieurs modules et fournit différents services.

3.3.1 – Services offerts

- Pour les utilisateurs utilisant un client web :

  - interrogation d'une base de données ou d'annuaire (LDAP, SQL, …)

  - système d'authentification fourni par le serveur web (certificat, HTTP basique, …)

  - chaînage des modules d'authentification

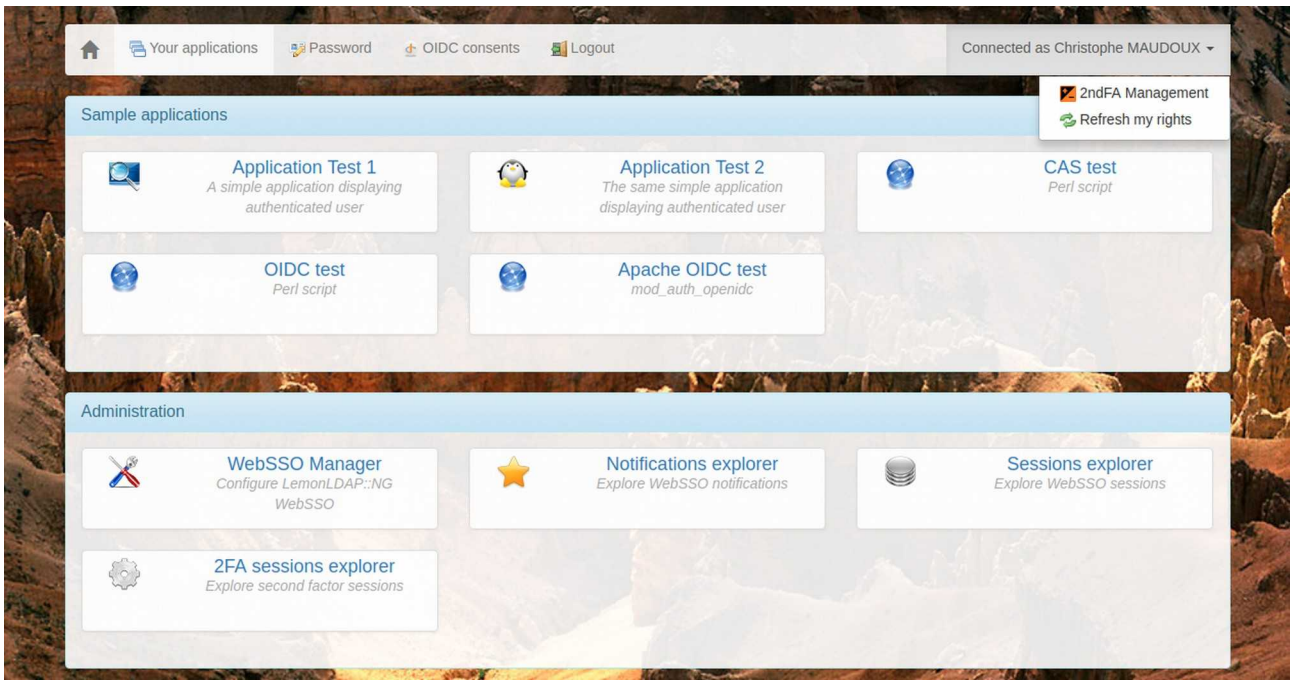

Figure 11 : *Portail*

- Serveurs SOAP, REST et d'API
- Fournisseur d'identités ou de services SAML, OpenID Connect, CAS, etc ...
- Proxy protocolaire entre différents protocoles de fédération d'identités
- Services de gestion des mots de passe pour les utilisateurs :

  - Formulaire de changement

  - Réinitialisation par mail

  - Imposer un renouvellement de mot de passe
- Menu dynamique des applications autorisées
- Service de notification d'informations aux utilisateurs

3.3.2 – Architecture

Le *Portail* est construit autour de quatre modules qui peuvent être désactivés :

- Authentification des utilisateurs (Auth) : comment vérifier l'identité des utilisateurs
- Base de données des utilisateurs (userDB) : où collecter les données utilisateurs
- Base de données des mots de passe (Pwd) : où changer les mots de passe utilisateurs
- Fournisseur d'identité (IdP) : vers qui transférer l'identité *(protocoles de fédération SAML, OpenID-Connect,…)*

3.3.3 – Cinématique

Lorsqu'un client tente d'accéder à une application protégée, le *Portail* :

- Vérifie la validité de l'URL demandée
- Contrôle si l'utilisateur est déjà authentifié. S'il n'est pas authentifié ou si l'authentification est forcée, le *Portail* recherche l'utilisateur (module UserDB) et l'authentifie (module Auth). En cas de succès, il crée la session, demande l'éventuel second facteur, calcule les groupes, macros et enregistre la session. LLNG peut également imposer la saisie d'un « capcha ».
- Modifie le mot de passe si nécessaire (module Pwd)
- Fournit l'identité si demandée (module IdP)
- Construit le cookie SSO contenant la clef de session
- Redirige l'utilisateur vers l'URL demandée ou affiche le menu

3.4 – Agents (Handlers) & Protection des applications

Les *Handler*s sont les agents logiciels chargés de protéger les applications. Ils ont pour rôle d'intercepter toutes les requêtes provenant ou à destination des clients et des applications. LLNG fournit des *Handler*s pour quatre types de serveurs web que sont Apache2.X, Nginx, Node.js *(serveur web basé sur JavaScript)* ou des serveurs web de type PSGI, basés sur Plack par exemple, tels Corona, Starman ou Twiggy. S'agissant de la protection des applications, elle peut être mise en œuvre sous quatre modes différents.

3.4.1 – Modes de protection

* Le mode direct consiste à installer les *Handler*s sur les serveurs web applicatifs. Les *Handler*s sont embarqués par les applications. Ce mode permet de protéger des applications hébergées sur les quatre types possibles de serveurs web mais impose aux développeurs d'incorporer le code des *Handler*s dans leurs applications.

* Le second mode est le fonctionnement en mandataire inverse (reverse-proxy). L'avantage ici est qu'il permet de « connecter » au SSO les applications à protéger sans modification. En revanche, ceci peut créer un goulet d'étranglement car toutes les requêtes passent par ce point unique. Actuellement, seuls les serveurs web Apache et Nginx peuvent être utilisés comme reverse-proxy (rvprx).

(« Proxy inverse », 2017) Un reverse-proxy est un serveur habituellement placé en frontal de serveurs web. Le reverse-roxy sert ainsi de relais pour les utilisateurs souhaitant accéder à une application web en lui transmettant indirectement les requêtes (cf. figure 12). Grâce au reverse-proxy, le serveur web est protégé des interrogations directes, ce qui renforce la sécurité et peut permettre de répartir la charge sur plusieurs serveurs équivalents servant la même application (Load Balancer). La ré-écriture programmable des URL permet de masquer et de contrôler l'architecture d'un site web interne. Mais cette architecture permet surtout le filtrage en un point unique des accès aux ressources web ou optimiser la compression du contenu des sites.

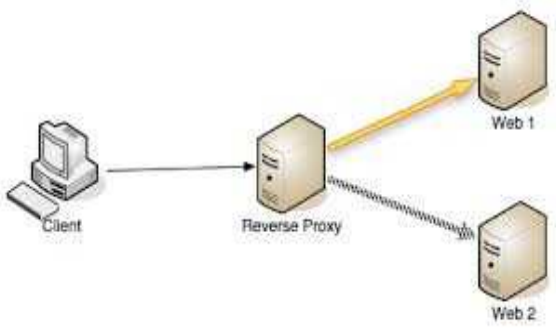

Figure 12 : Mandataire Inverse (*ReverseProxy*)

Le fonctionnement en mode reverse-proxy impose de prendre quelques mesures de sécurité. Il est important de veiller à ce que les applications n'autorisent des requêtes HTTP issues uniquement du pool de reverse-proxies afin d'éviter le contournement du SSO en interrogeant directement les applications. Ensuite, il peut être judicieux de mettre en place une connexion SSL via les proxy inverses pour limiter les risques au cas où une application serait piratée.

Sous ce mode, les *Handler*s doivent être déclarés dans chaque hôte virtuel servi par le reverse-proxy afin d'activer la protection. Il est tout à fait envisageable d'utiliser différents types ou technologies de *Handler* sur le même reverse-proxy sous certaines conditions que je détaillerai plus loin dans ce mémoire. Par exemple, avec Nginx, il est possible de protéger une application par l'intermédiaire d'un VH utilisant le *Handler* « ServiceToken » de type Node.js et une autre application avec un *Handler* « AuthBasic » utilisant FastCGI.

* Le troisième mode est en fait un mode hybride servant des applications protégées soit directement soit derrière un mandataire inverse.

* Le dernier mode de protection est l'utilisation par les applications elle-mêmes d'un protocole de fédération d'identité. L'application peut échanger directement avec le *Portail* en SAML par exemple.

3.4.2 – Types de Handlers

En fonction du mode de protection, de dialogue supporté par les clients ou du but recherché par les applications, il est possible d'utiliser plusieurs types de *Handler*s pour répondre aux différents besoins. Il faut noter que les services de fédération d'identités sont gérés uniquement par le *Portail* et non par les différents *Handler*s.

L'agent standard (*Main*) est basé sur une authentification par cookie. Le *Portail* délivre un cookie SSO au client qui le transmet avec chaque requête au serveur (en-tête cookie). Le *Handler* vérifie la présence du cookie et autorise l'accès. Le cookie SSO n'est jamais transmis aux applications, uniquement les en-têtes HTTP.

L'agent basique (*AuthBasic*) impose une authentification basique au client en envoyant un en-tête « WWW-Authenticate » lors de la première demande d'accès. L'application cliente transmet le couple identifiant / mot de passe au *Handler* qui interroge le *Portail* en utilisant le service web REST pour jouer l'authentification. Lorsque la session a été créée par le *Portail* et un cookie SSO délivré, le *Handler* contrôle les accès de façon standard. Généralement, il est utilisé pour mettre en place une authentification entre applications ou de serveur à serveur.

L'agent de développement (*DevOps*) offre un service SSO aux développeurs, il s'agit du « SSO as a Service ». Le SSOaaS permet aux développeurs de définir au sein de leurs applications web les règles d'accès et les en-têtes qu'ils souhaitent recevoir du SSO. Dans ce cas, en-têtes et règles ne sont plus renseignés dans le *Manager* mais dans un fichier au format JSON qui doit être enregistré à la racine du site.

L'agent sécurisé (*SecureToken*) permet de transmettre un jeton dans un en-tête avec chaque requête aux applications protégées. L'identifiant de l'utilisateur est enregistré par un serveur de cache et les applications peuvent interroger le cache pour récupérer l'identité de l'utilisateur. Ce mécanisme permet à une application de rejouer l'authentification auprès d'un autre serveur comme un web-mail par exemple. Ce *Handler* est déprécié car le jeton peut être utilisé pour adresser toutes applications connectées derrière le SSO. Le *Handler* de service est à privilégier.

L'agent de service (*ServiceToken*) est une évolution du *Handler* précédent. Le *Handler* Service Token permet de spécifier et donc de restreindre les URL pour lesquelles il est valide. De plus, le jeton de service permet de définir une période de validité, fixée à 30 secondes par défaut.

Enfin, l'agent *DevOpsST* combine le *SSOaaS* ainsi que le *ServiceToken*. Le *Handler* « Zimbra » utilise le protocole de pré-authentification Zimbra pour accéder à leurs services (mail, calendrier, etc...). Le cas du *Handler Cross-Domain Authentication* (CDA) permettant d'étendre le SSO sur des domaines de confiance différents est détaillé dans le paragraphe suivant.

3.4.3 – SSO étendu à plusieurs domaines (CDA)

Pour des raisons de sécurité, un cookie n'est valable que sur le domaine pour lequel il a été délivré. Pour étendre leur portée et pouvoir mettre en place un SSO valable sur plusieurs domaines, LLNG implémente un agent de type Cross-Domain Authentication (CDA).

La cinématique *CDA* est la suivante :

1. L'utilisateur possède un cookie valable sur le domaine principal
2. L'utilisateur essaye d'accéder à une application hébergée sur un autre domaine
3. Le *Handler* ne peut pas lire le cookie SSO car le *Handler* et le cookie ne sont pas du même domaine. Le *Handler* redirige donc l'utilisateur vers son *Portail* d'origine
4. Le *Portail* identifie l'utilisateur avec son cookie SSO et constate qu'il provient d'un autre domaine (en-tête *referer*)
5. Le *Portail* redirige l'utilisateur vers l'application protégée hébergée sur l'autre domaine en passant un jeton en paramètre dans l'URL. Le jeton est une référence vers une session contenant la vraie clef de session
6. Le *Handler* détecte le paramètre passé dans l'URL, récupère la clef de session, supprime le jeton de l'URL et crée un cookie valable dans son propre domaine contenant la clef de session

3.5 – Bases de données & Annuaires

3.5.1 – Présentation

Pour fonctionner, LLNG requiert l'utilisation de différentes bases de données. Par base de données, j'entends tout système permettant de lire ou écrire des données. Il peut s'agir d'une base de données en tant que telle, d'un fichier à plat ou d'un annuaire.

Les bases de données nécessaires peuvent être réparties en deux catégories. Les bases de données internes uniquement utilisable par LLNG et les bases externes non gérées par LLNG.

Pour le processus d'authentification et de gestion des mots de passe utilisateurs, en plus des protocoles LDAP et ActiveDirectory, LLNG est également capable d'utiliser plusieurs types de base de données via un connecteur Perl spécifique (classique type RDBI ou compact tel CDBI) pour des bases SQL (MySQL ou PostGres), MongoDB, Redis, etc...

LLNG est compatible avec la politique des mots de passe des annuaires LDAP. C'est le cas d'OpenLDAP avec « l'overlay policy » (Liefooghe, s. d.). Par défaut, dans OpenLDAP, le mot de passe utilisateur est stocké sans cryptage. Si l'on modifie le mot de passe via un fichier LDIF (« LDAP Data Interchange Format », 2017), on aura donc le mot de passe en clair dans l'annuaire, ce qui n'est pas sécurisé. Il est possible également d'appliquer un algorithme de cryptage en amont (dans l'application cliente) et de le transmettre à OpenLDAP qui le stockera tel quel. Cette « couche » intermédiaire de politique des mots de passe permet à l'utilisateur de savoir que son compte est bloqué ou a expiré mais également de changer son mot de passe en respectant les critères fixés dans l'annuaire (taille minimale, présence dans l'historique, etc...)

Les configurations et sessions peuvent être stockées sous forme de fichiers à plat, dans une base de données, par SOAP ou dans un serveur de cache comme « memcached ». (« Memcached », 2017) est un système d'usage général servant à gérer la mémoire cache distribuée. Il est souvent utilisé pour augmenter la vitesse de réponse des sites web utilisant des bases de données. Il gère les données et les objets en RAM de façon à réduire le nombre de fois qu'une même donnée stockée dans un périphérique externe est lue. Il fonctionne sous Unix, Windows et MacOS et est distribué selon les termes d'une licence libre dite permissive. Ce choix permet de mettre en place très rapidement des solutions de haute-disponibilité. En outre, en cas de configuration stockée sur un système distant, un mécanisme de cache permet de limiter le transfert des données sur le réseau.

3.5.2 – Internes

Les bases de données ou tables internes nécessaires à LLNG sont utilisées pour stocker la configuration, les sessions, les notifications et le cache.

La base des configurations permet de stocker la configuration courante de LLNG et d'en conserver l'historique. La configuration de LLNG n'inclut pas la configuration du serveur web qui n'est pas gérée par le *Manager*.

La base des sessions est utilisée pour stocker les sessions SSO des utilisateurs créées par le *Portail*. La session SSO contient l'ensemble des attributs et groupes LDAP de l'utilisateur mais également le résultat des macros et des groupes calculés, définis dans le *Manager*. En plus des sessions SSO, LLNG utilise un autre type de session nommé « sessions persistantes ». Comme leur nom l'indique, ces sessions sont persistantes en base. Une session persistante est créée pour chaque utilisateur lors de sa première connexion au *Portail*. Les sessions persistantes permettent de conserver l'historique des connexions, les différents seconds facteurs de l'utilisateur et ses consentements concernant l'échange d'informations entre le fournisseur d'identité qu'est LLNG et les différents fournisseurs de services.

La base de cache permet de stocker la configuration et les sessions SSO pendant 10 minutes. Ce mécanisme est utilisé pour augmenter la vitesse de réponse en réduisant le nombre de connexions à la base de données. Il est important de préciser qu' un utilisateur dont la session a expiré aura toujours accès le temps de la mise à jour du cache.

La dernière base interne est celle des notifications. Les notifications sont des messages qui peuvent être présentés à l'utilisateur lors de sa connexion au *Portail* pour l'informer d'un changement de politique d'accès ou de l'expiration imminente de son mot de passe par exemple.

3.5.3 – Externes

Les principales bases de données externes sont les bases des identités, des données utilisateurs et des mots de passe.

La base des identités est la base permettant d'authentifier les utilisateurs. Elle contient les identités des utilisateurs.

La base des données utilisateurs contient les différents attributs des utilisateurs comme le mail, l'adresse, le grade, sa fonction, etc... Ces données peuvent ensuite être utilisées dans le *Manager* pour écrire les règles d'accès, calculer des macros ou des groupes locaux.

Enfin, la base des mots de passe est la base contenant le mot de passe des utilisateurs.

LLNG permet dans sa configuration de chaîner ces différentes bases. Si la donnée n'est pas trouvée dans la première base, le *Portail* interroge la suivante.

3.5.4 – Bases de données & Connecteurs

Pour accéder à la base de données des configurations,, si le type de base choisi est de type SQL, LLNG utilise le module Perl DBI (DataBase Interface) (« DBI - search.cpan.org », s. d.). Il s'agit d'une interface de programmation permettant de manipuler des bases de données en langage de programmation Perl. DBI a été spécifié par Tim Bunce, en collaboration avec d'autres, à partir de 1994. Perl DBI est couramment maintenu à travers Internet comme un module CPAN conformément au modèle Open Source. DBD (DataBase Driver) est une couche d'abstraction qui permet aux programmeurs d'utiliser dans leurs applications du code SQL aussi indépendant que possible de la base de données utilisée. LLNG peut interroger des bases de données SQL au format

RDBI avec des contraintes sur les tuples ou CDBI plus compact (modèles créés par Xavier Guimard). L'utilisation de ce module permet de générer une clef primaire incrémentielle pour référencer les configurations.

Pour les lire et écrire les données de sessions, LLNG accède à la base à l'aide du module Apache::Session (« Apache::Session - search.cpan.org », s. d.) ou ses dérivés Apache::Session::Browseable (« Xavier Guimard / Apache-Session-Browseable », s. d.). Apache::Session est un framework permettant de stocker et rechercher des données de session dans des bases PostgreSQL, MySQL, Oracle, Redis, etc... Ce module génère l'identifiant de session qui sera transmis aux clients via le cookie de session. A l'origine, il fut écrit pour le serveur Apache mais il peut fonctionner avec d'autres serveurs. L'inconvénient majeur de celui-ci est qu'il ne permet pas d'effectuer de recherche. Par exemple, il est impossible de connaître le nombre de sessions existantes en base. Les dérivés « browseable », écrits par Xavier Guimard, permettent d'optimiser les accès pour l'explorateur de sessions et les restrictions d'ouverture de sessions grâce à l'ajout d'index de recherches et de méthodes spécifiques. En effet, sans ces index, LLNG doit récupérer toutes les sessions présentes en base puis rechercher les sessions nécessaires. Avec l'index, LLNG téléchargera uniquement les sessions voulues.

Les tests de performance réalisés ont permis de constater les résultats suivants. Les serveurs d'annuaire LDAP, surtout optimisés en lecture, présentent de très mauvaises performances pour stocker les sessions. Il faut éviter de les utiliser pour des systèmes très sollicités. Les bases de données avec SGBD MySQL ou MariaDB ont de meilleures performances en lecture qu'en écriture.

Il est conseillé d'utiliser des bases PostgresSQL en utilisant le paramètre UNLOGGED pour la table des sessions afin de ne pas écrire dans les journaux de transactions (WAL). Ne pas utiliser ce paramètre pour la table des sessions persistantes

Le WAL (« Write-Ahead Logging (WAL) », s. d.) est un mécanisme permettant de maintenir la cohérence de la base de données en écrivant les actions dans un journal avant de les exécuter. Ceci permet de rejouer (REDO) les transactions en cas de crash du système. Pour des installations soumises à forte charge, la meilleure solution est d'utiliser une base PostgreSQL avec le module Browseable::Postgres, PgHstore ou PgJSON.

Les meilleures performances sont obtenues par des bases de données NoSQL avec le Système de Gestion de Base de Données (SGBD) Redis.

3.5.5 – SQL & NoSQL

(Bruchez, 2013) Les bases de données relationnelles sont souvent structurées autour d'un seul point d'entrée, la clef, et sont susceptibles de croître très rapidement. Par ailleurs, certains cas d'utilisation exigeant des temps d'accès très courts défient également les capacités des moteurs transactionnels. C'est pour répondre à ces différentes problématiques que sont nées les bases de données NoSQL (Not Only SQL), sous l'impulsion de grands acteurs du Web comme Facebook ou Google, qui les avaient développées à l'origine pour leurs besoins propres. Grâce à leur flexibilité et

leur souplesse, ces bases non relationnelles permettent en effet de gérer de gros volumes de données hétérogènes sur un ensemble de serveurs de stockage distribués, avec une capacité de montée en charge très élevée. Elles peuvent aussi fournir des accès de paires clef-valeur en mémoire avec une très grande célérité.

Le choix d'une base SQL ou NoSQL peut être orienté par l'utilisation du théorème CAP ou CDP en français (« Théorème CAP », 2018). Celui-ci aussi connu sous le nom de théorème de Brewer énonce qu'il est impossible sur un système informatique de calcul distribué de garantir en même temps (c'est-à-dire de manière synchrone) les trois contraintes suivantes :

- Cohérence (*Consistency*) : tous les nœuds du système voient exactement les mêmes données au même moment
- Disponibilité (*Availability*) : garantie que toutes les requêtes reçoivent une réponse
- Tolérance au partitionnement (*Partition Tolerance*) : quel que soit le nombre de serveurs, toute requête doit fournir un résultat correct

D'après ce théorème, un système de calcul distribué ne peut garantir à un instant T que deux de ces contraintes mais pas les trois simultanément. En fonction des contraintes que l'on souhaite garantir, il est possible de choisir différents SGBD NoSQL. Les bases de données NoSQL les plus courantes actuellement sont Redis, MongoDB et Cassandra (cf. figure 13-1).

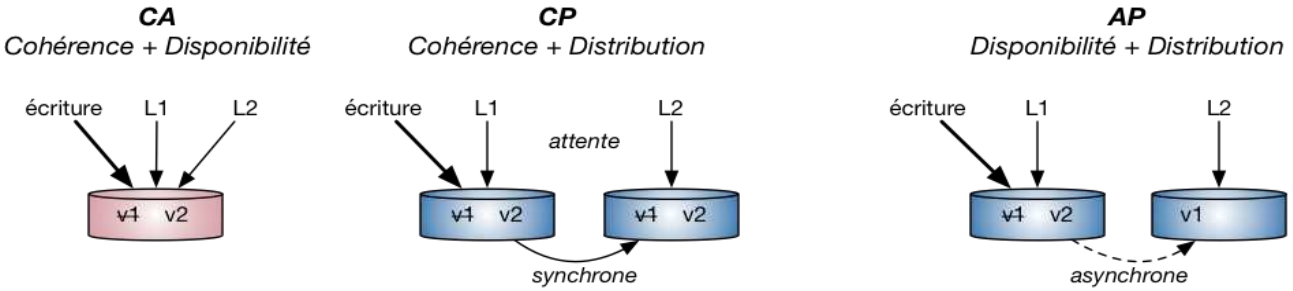

Figure 13 : Modèles

CA : Lors d'opérations concurrentes sur une même donnée, les requêtes L1 et L2 retournent la nouvelle version (v2) et sans délai d'attente. Cette combinaison n'est possible que dans le cadre de bases de données transactionnelles.

CP : Distribuer les données sur plusieurs serveurs en garantissant la tolérance aux pannes (réplication). En même temps, il est nécessaire de vérifier la cohérence des données en garantissant la valeur retournée malgré des mises à jour concurrentielles.

AP : Fournir un temps de réponse rapide tout en distribuant les données et les réplicas. De fait, les mises à jour sont asynchrones sur le réseau, et la donnée est potentiellement consistante.

Dans le cadre de LLNG, il est tout à fait envisageable d'utiliser une base NoSQL pour stocker les sessions. Les sessions étant par définition temporaires, leur perte aurait simplement pour conséquence d'obliger l'ensemble des utilisateurs connectés à se ré-authentifier.

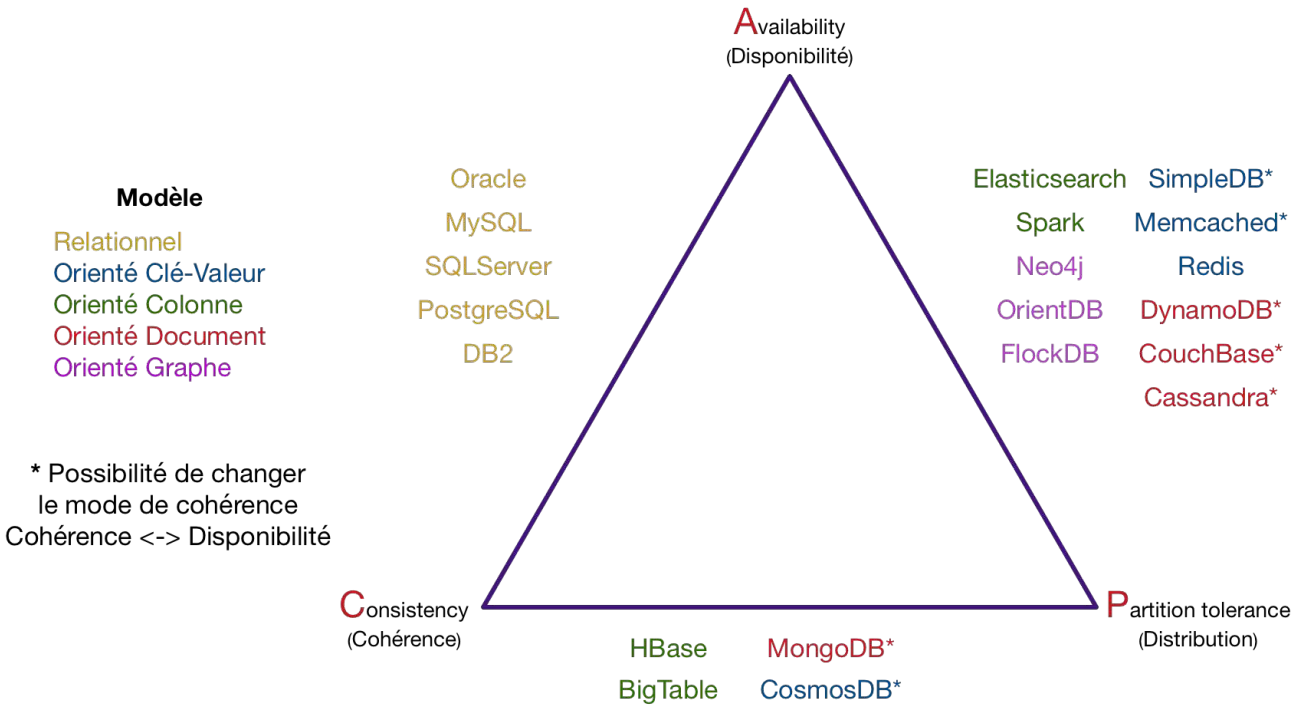

Figure 13-1 : Triangle du théorème CAP

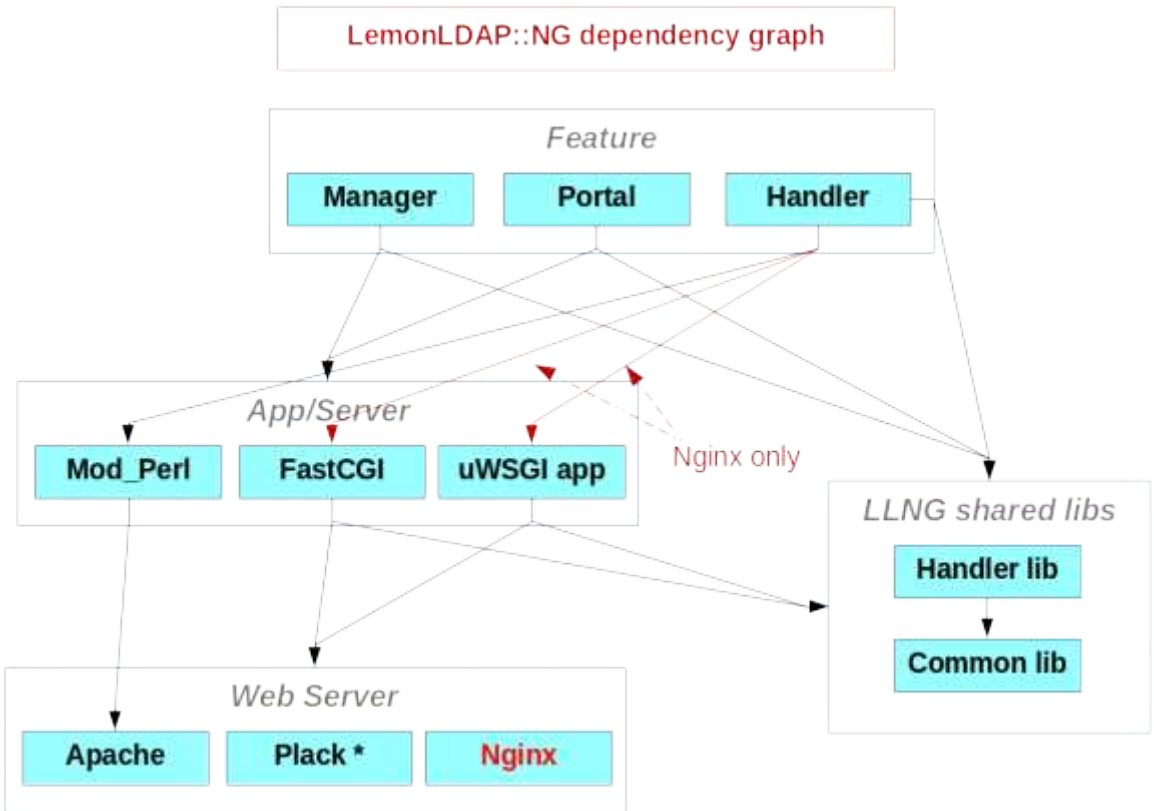

Figure 14 : Schéma des Plateformes & Dépendances

4 – Plateforme & Dépendances

4.1 – Architecture & Installation de LLNG

En fonction de la charge que doit supporter le système à protéger (nombre d'utilisateurs et de connexions), il est recommandé d'utiliser les configurations suivantes. Pour des plateformes soumises à de faibles ou moyennes charges, il est à privilégier des serveurs Nginx avec le serveur LLNG FastCGI par défaut ou des serveurs Apache avec le moteur mpm_prefork. Pour des installations fortement sollicitées, il est préférable d'avoir recours à des serveurs Nginx et d'adapter le moteur FastCGI aux usages.

(« Apache HTTP Server », 2018) Le choix du moteur d'Apache change les performances du serveur HTTP. Historiquement, Apache fonctionne en *prefork*, ce qui signifie qu'un processus père, lancé avec des droits étendus (root), démarre des processus enfants qui traiteront chacun un certain nombre de requêtes des clients. Cependant, sous Linux, la multiplication des processus provoque une augmentation de consommation de ressources (mémoire, descripteurs de fichiers). En mode *worker*, Apache lance des threads qui géreront les demandes entrantes. La différence est qu'il s'agit d'un mode plus pré-emptif dans lequel le processus père prépare les ressources pour ses « threads ». Des modules ou des librairies utilisées par ces modules peuvent ne pas être prévus pour fonctionner dans un environnement multi-thread. Ils pourront en ce cas être source de dysfonctionnements si on les utilise avec le mode *worker*. Depuis la version 2.4, Apache supporte le module *event*. C'est un fonctionnement dérivé du mode *worker* à ceci près que les threads ne desservent pas seulement une connexion client mais peuvent réaliser plusieurs tâches indépendamment de la connexion. Ainsi, les notions de « Keep-Alive » sont mieux gérées dans le sens où un thread n'attend plus que la connexion soit terminée pour en desservir une autre. Plus clairement, le thread dessert une requête et non pas une connexion.

LemonLDAP::NG est hautement adaptable et configurable. Il peut fonctionner sur plusieurs types de serveurs, fournir des services de protection sous différentes formes et utiliser beaucoup de composants différents. La contre partie à cette souplesse est le grand nombre d'implémentations possibles (cf. figure 14) présentées dans le paragraphe suivant.

4.2 – Portail & Manager

A compter de la version 2.0, le *Portail* et le *Manager* sont des applications FastCGI construites avec le module Plack (« PSGI/Plack - Perl Superglue for Web Frameworks and Web Servers », s. d.) qui est une implémentation du protocole PSGI (« PSGI », 2018). *Portail* et *Manager* peuvent donc être installés sur des serveurs web compatibles.

Leur installation sur Apache est possible avec un moteur FastCGI fourni par les modules « mod_fcgid » ou « mod_fastcgi ».

Nginx est nativement compatible avec les protocoles FastCGI et uWSGI. Il est possible d'utiliser le *Manager* et le *Portail* avec Nginx par l'intermédiaire d'un serveur FastCGI ou uWSGI qui peut être servi par différents moteurs.

Une solution intéressante et efficace est d'utiliser des serveurs web de type PSGI comme *Starman* qui permettent de faire fonctionner *Portail* et *Manager* directement. En revanche, ceux-ci ne sont pas adaptés pour servir du contenu. Il sera nécessaire d'utiliser une VIP chiffrante pour mettre en place du HTTPS ou un serveur dédié pour le contenu statique (CDN).

4.3 – Handlers : Interception des requêtes

Tous les *Handler*s sont écrits en programmation de classes avec Perl et basés sur le module Plack implémentant le protocole PSGI sauf le *Handler* Node.js, écrit en JavaScript. La programmation de classes est un modèle particulier permettant de coder sans créer d'objet donc pratiquement sans empreinte mémoire. Le « choix » de Perl et de la programmation de classes sont des héritages du serveur web Apache pour lequel avait été écrit LemonLDAP::NG à l'origine. Celui-ci n'autorisait que ce type de programmation et n'offrait que les modules Mod_Perl (« mod_perl », 2018) ou le langage C pour manipuler les requêtes HTTP. Le choix a donc été fait d'utiliser Mod_Perl, plus pratique à utiliser.

L'interception des requêtes par les *Handler*s fonctionne de la façon suivante. Le serveur web reçoit les requêtes HTTP provenant du client (navigateur) et les « passe », les transmet au(x) *Handler*(s) via une interface de type CGI (Common Gateway Interface) (« Common Gateway Interface », 2016). En effet, au lieu d'envoyer le contenu d'un fichier, le serveur HTTP exécute un programme, ici le *Handler*, puis retourne le contenu généré. CGI est le standard industriel qui indique comment transmettre la requête du serveur HTTP au programme, et comment récupérer la réponse générée.

La norme RFC3875 (Robinson <drtr@apache.org>, s. d.) impose un format de nommage des en-têtes HTTP. En effet, le serveur HTTP utilise un ensemble de variables d'environnement pour transmettre des informations aux scripts CGI. Certaines de ces variables d'environnement sont utilisées pour communiquer certains aspects de la requête HTTP, comme le type de contenu, le port TCP, le nom d'hôte ou la méthode de requête (par exemple GET ou POST). Il y a un ensemble de variables d'environnement différent, que la norme appelle « Meta-Variables spécifiques au protocole », qui servent à transmettre les valeurs d'en-têtes HTTP au script CGI. La mise en correspondance des en-têtes HTTP avec les variables d'environnement suit une procédure standard : le nom d'en-tête HTTP est converti en majuscules, « - » est remplacé par « _ », et « HTTP_ » est le préfixe. Par exemple, l'en-tête « Cookie » est traité avec la variable d'environnement « HTTP_COOKIE ».

De plus, comme les *Handler*s sont écrits en respectant les standards PSGI (Plack) et FastCGI, plusieurs implémentations sont possibles avec les serveurs web actuels.

4.3.1 – Serveurs de type PSGI (Starman, Corona, Twiggy, …)

Le serveur passe les requêtes directement au *Handler*. Celui-ci « lit » la clef de session transmise par le cookie SSO. Il récupère les informations utilisateur puis les sauvegarde dans son cache local (15 secondes). Ensuite, il vérifie les règles d'accès et initialise les en-têtes HTTP à transmettre à l'application, en-têtes qui ont été déclarés dans le *Manager*. Enfin, le *Handler* construit l'en-tête de la requête en y incluant les en-têtes HTTP et retourne la requête construite au serveur web qui la transmet au client.

4.3.2 – Serveur Nginx

(« nginx », 2018) Nginx est un serveur Web ainsi qu'un proxy inverse écrit par le mathématicien Igor Sysoev, dont le développement a débuté en 2002 pour les besoins d'un site russe à très fort trafic.

Nginx est un système asynchrone par opposition aux serveurs synchrones où chaque requête est traitée par un processus dédié. Au lieu d'exploiter une architecture parallèle et un multiplexage temporel des tâches par le système d'exploitation, Nginx utilise les changements d'état pour gérer plusieurs connexions en même temps. Le traitement de chaque requête est découpé en de nombreuses mini-tâches et permet ainsi de réaliser un multiplexage efficace entre les connexions. Afin de tirer parti des ordinateurs multiprocesseurs, plusieurs processus peuvent être démarrés. Ce choix d'architecture se traduit par des performances très élevées, mais également par une charge et une consommation de mémoire particulièrement faibles comparativement aux serveurs HTTP classiques, tels qu'Apache. Nginx supporte nativement les protocoles FastCGI et uWSGI qui sont deux implémentations optimisées d'une CGI.

D'après (« FastCGI », 2017), FastCGI a été créée en 1996 pour gérer les applications dynamiques des applications web. Avec CGI, chaque requête lance une nouvelle instance de CGI qui appellera le programme à exécuter. Le binaire CGI recrée à chaque appel le contexte de l'environnement d'exécution et ne permet pas de limiter le nombre de processus simultanés. Ce nombre sera donc dépendant du nombre de processus simultanés du serveur web. En revanche, avec FastCGI, une variable est introduite permettant de déterminer le nombre minimum et maximum de processus CGI à exécuter, indépendamment du nombre de processus HTTP maximum.

uWSGI, comme FastCGI, est un protocole permettent à un serveur web de déléguer des traitements à un autre programme. A l'origine WSGI et uWSGI avaient été développés pour servir des applications *Python*.

Nginx reçoit les requêtes HTTP et les transmet à un serveur d'échange FastCGI ou uWGSI emportant le module PSGI. LLNG fournit un serveur FastCGI par défaut utilisant le moteur FCGI. Celui-ci peut être remplacé par d'autres implémentations : FCGI::Engine::ProcManager, FCGI::Async ou autres. Le serveur FastCGI / uWSGI passe les requêtes au *Handler* pour traitement qui les retourne en suivant le parcours inverse.

Le *Handler* Nginx a pour particularité de ne pas pouvoir ajouter directement les en-têtes HTTP aux requêtes transmises à l'application protégée. Deux solutions sont possibles, soit ajouter les en-têtes manuellement dans les « vHosts » soit utiliser un script LUA qui ajoute les directives. Le script fournit par LLNG (nginx-lua-headers.conf) est en fait une « simple boucle » qui permet de transmettre au maximum dix en-têtes HTTP par défaut. Il peut facilement être modifié pour transmettre plus d'en-têtes.

LUA (« Lua », 2018) est un langage de script libre, réflexif et impératif. Créé en 1993, il est conçu de manière à pouvoir être embarqué au sein d'autres applications afin d'étendre celles-ci. L'interpréteur Lua est écrit en langage C ANSI strict. De ce fait, il est compilable avec une grande variété de systèmes. Il est également très compact et est souvent utilisé dans des systèmes embarqués.

4.3.3 – Serveur Apache 2.X (2.2 ou 2.4)

(« Apache HTTP Server », 2018) Ce serveur web était le plus populaire sur le web. Apache est conçu pour prendre en charge de nombreux modules lui donnant des fonctionnalités supplémentaires : interprétation du langage Perl, PHP, Python et Ruby, serveur proxy, Common Gateway Interface, réécriture d'URL, négociation de contenu, protocoles de communication additionnels, etc... Néanmoins, il est à noter que l'existence de nombreux modules Apache complexifie la configuration du serveur web et de nombreuses failles de sécurité affectant uniquement les modules sont régulièrement découvertes.

Dans le cas du serveur Apache, les requêtes sont transmises directement au *Handler* via le module *Mod_Perl*. Le *Handler* Apache est en réalité constitué de deux modules : le *Handler* lui-même et un module « PerlHeaderParser*Handler* » qui ajoute les en-têtes HTTP aux requêtes. Il existe également une implémentation FastCGI pour Apache (Mod_fstcgi). En revanche, ce module n'offre pas la possibilité de déléguer l'autorisation d'accès à un processus externe ou de modifier l'en-tête des requêtes. Il permet uniquement de générer du contenu dans le corps de la requête. De ce fait, l'utilisation de *Mod_Perl* reste donc obligatoire.

4.3.4 – Node.js

(« Des-applications-ultra-rapides-avec-node-js.pdf », s. d.) & (« Node.js », 2018) Node.js est une plateforme logicielle libre et événementielle écrite en JavaScript orientée vers les applications réseau qui doivent pouvoir monter en charge. Elle utilise la machine virtuelle V8 développée par Google. Le moteur V8 est un moteur javascript libre écrit en C++. Parmi les modules natifs de Node.js, on retrouve « HTTP » qui permet le développement de serveur HTTP. Node.js est un environnement bas niveau permettant l'exécution de JavaScript côté serveur.

Node.js dispose de nombreux modules comme « Express.js » permettant d'étendre ses fonctionnalités notamment un module FastCGI. C'est par son intermédiaire que le serveur web Node.js transmet les requêtes au *Handler*. De plus, le *Handler* Node.js nécessite un module spécifique comme « nodedbi.js » pour pouvoir interroger une base de données pour le stockage du cache.

Ce *Handler* étant développé en *JavaScript*, les règles doivent être écrites sous cette forme dans le *Manager* : $uid eq « dwho » devient $uid === « dwho » et les applications qu'il protège doivent être explicitement déclarées dans la section Node*Handler* du fichier de configuration de LLNG : *lemonldap-ng.ini*.

5 – SSO & Sécurité Intérieure

5.1 – Le STSISI

(« Service des technologies et des systèmes d'information de la Sécurité intérieure », s. d.) Le Service des Technologies et des Systèmes d'Informations de la Sécurité Intérieure est un service français dépendant du ministère de l'Intérieur créé le 1er septembre 2010. Il est issu de la fusion de la Sous-Direction des Télécommunications et de l'Informatique (SDTI), créée le 1er février 1985, dépendante de la Direction Générale de la Gendarmerie Nationale (DGGN), et du Service des Technologies et des Systèmes d'Information, dépendant de la Direction Générale de la Police Nationale (DGPN).

Rattaché organiquement à la DGGN, il est commandé par le général de corps d'armée Bruno Poirier-Coutansais, par le contrôleur général Christophe Fichot son adjoint et placé sous l'autorité conjointe des deux directeurs généraux. Le STSISI ou ST(SI)² fait partie des toutes premières mutualisations entre Police et Gendarmerie. Le STSISI est situé au sein de la direction générale de la Gendarmerie nationale sise à Issy-les-Moulineaux.

Ce service a pour mission première la maîtrise d'ouvrage des modernisations technologiques des forces de la Sécurité Intérieure pour la Police Nationale et de la Gendarmerie Nationale. Il assure le pilotage du système d'information de la Police et de la Gendarmerie, la mise en œuvre des architectures, systèmes et outils informatiques d'aide à l'enquête, le développement de partenariats et l'animation d'un réseau avec les entreprises. Il assure en outre des missions de recherche, d'expertise et d'essais pour la Police et d'autres forces de sécurité, et développe de nouveaux équipements. Il assure également la gestion du parc des équipements et le pilotage des supports de proximité, soit environ 2 700 policiers et gendarmes sur l'ensemble du territoire pour plus de 245 000 utilisateurs.

Le STSISI est subdivisé en quatre sous-directions qui sont la Sous-Direction du Soutien Opérationnel, la Sous-Direction des Systèmes d'Information, la Sous-Direction des Réseaux Radio et la Sous-Direction des Applications de Commandement (cf. figure 14-a).

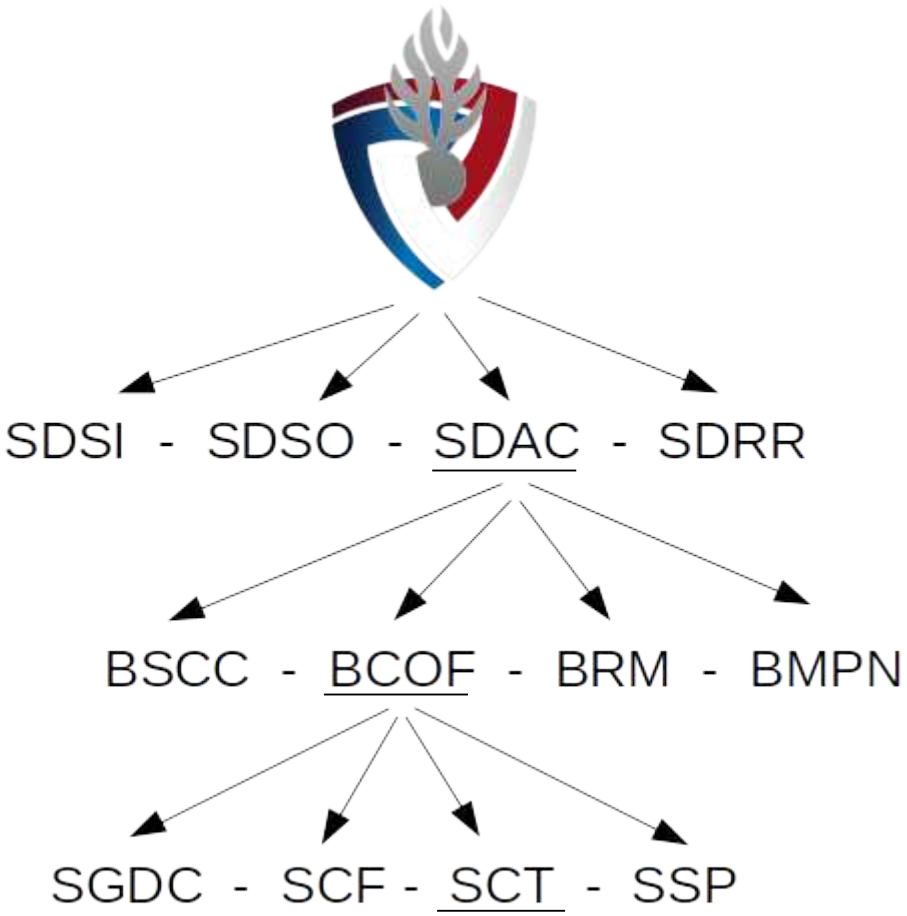

Figure 14-a : Arbre hiérarchique du STSISI

5.2 – Mise en œuvre

La Sous-Direction des Applications de Commandement (SDAC) se compose de quatre bureaux et renforce son rôle opérationnel en assumant le suivi de projets majeurs pour la sécurité intérieure tels la mobilité, la proximité numérique et le traitement des appels d'urgence (projets MCIC pour la Police et BDSP pour la Gendarmerie nationale), la fonction de contrôle opérationnel des fichiers (authentification de la sécurité intérieure) et de suivi des référentiels métiers.

Au sein de la SDAC, le Bureau de Contrôle Opérationnel des Fichiers, commandé par le Lieutenant-Colonel Hamel, est chargé, entre autres activités, de gérer et mettre en œuvre le contrôle d'accès aux applications et aux fichiers. Deux sections, commandées par le Capitaine Marcq, assurent cette mission.

5.2.1 – Section Contrôle Fonctionnel

Cette section a pour rôle de mettre en œuvre les règles d'accès aux applications ou fichiers en concordance avec les différents arrêtés ministériels publiés au Journal Officiel (JO) et de s'assurer de la conformité avec les déclarations CNIL. Pour ce faire, la Section Contrôle Fonctionnel (SCF) travaille en collaboration avec les délégués à la Protection et Gouvernance des Données, qui s'assurent de la conformité au RGPD et avec les déclarations CNIL, ainsi que les maîtrises d'ouvrages qui proposent les créations et modifications des politiques d'accès selon la doctrine d'emploi en vigueur. Deux personnels sont ainsi responsables du bon accès "fonctionnel" et du respect du droit d'en connaître de quelques 250 000 agents de la police et de la gendarmerie nationales à une quarantaine de fichiers administratifs ou judiciaires.

5.2.2 – Section Contrôle Technique

La Section Contrôle Technique (SCT), où je suis affecté, est armée par 5 personnels. Elle a pour mission de construire les accès aux différents fichiers ou applications qui peuvent être hébergés en interne (IPMS) ou par d'autres ministères et services extérieurs.

Cette section est plus particulièrement chargée de la gestion et de la mise en œuvre des architectures réseaux : administrations des DMZ, des proxy, des connexions vers les autres réseaux, de la fédération d'identités avec les partenaires extérieurs et plus généralement des SSO de la Sécurité Intérieure ainsi que la mise à disposition des traces de connexions et d'outils d'analyse aux différents échelons de commandement et Inspections Générales.

Les différents SSO de la Sécurité Intérieure sont tous basés sur la solution LemonLDAP::NG. Pour chaque SSO, plusieurs déclinaisons existent afin de répondre aux différents besoins d'interconnexion ou de la vie d'un projet (production, formation, pré-production, développement).

5.2.3 – STIG & IPMS

(« Service du traitement de l'information de la Gendarmerie », 2015) Le STIG assure la production, l'exploitation et l'intégration des applications informatiques sur le *DataCenter* de la Gendarmerie Nationale. Situé en Ile de France, il est le premier centre de services informatiques de l'État français à être certifié ISO 20000 et ISO 27001. Les personnels du STIG sont des experts de la production, de l'exploitation et de l'intégration dans des environnements de « DataCenter ».

Le STIG dispose actuellement d'une plate-forme de très haute disponibilité appelée IPMS, pour « infrastructure de production mutualisée et secourue », répartie sur deux sites distants en région parisienne, ce qui permet notamment d'assurer la continuité et la reprise d'activité. L'IPMS est une plateforme issue du projet PGS (Plan Global de Secours) qui a pour objectif de :

- garantir la continuité de service des applications en fonction de la criticité
- fournir, une solution de secours sur site distant en cas de sinistre majeur et de reprendre l'activité
- donner à ce nouveau système la possibilité d'évoluer facilement
- réorganiser la production informatique et le support utilisateur en s'appuyant sur la mise en place de processus et pratiques industriels

L'IPMS est une infrastructure efficiente, hautement disponible, capable de rétablir le services aux utilisateurs très rapidement en cas de sinistre majeur. Les clients du STIG sont notamment le $ST(SI)^2$ et divers ministères ou entités ministérielles.

5.2.4 – Les différents SSO

Pour sécuriser et tracer les accès aux différentes applications et fichiers, le STSISI a en charge la mise en œuvre, l'administration et le bon fonctionnement de plusieurs SSO sur son Intranet mais également sur l'Internet. Tous sont basés sur la solution LemonLDAP::NG et fonctionnent soit en mode ReverseProxy soit en fédération d'identité avec d'autres administrations ou ministères. LLNG est l'élément commun à toutes les applications sensibles qui garantie pour la CNIL la confidentialité et la traçabilité des accès. « Proxyma » est le SSO dédié à la Gendarmerie nationale. Celui de la Police nationale est baptisé « CheopsNG ». Pour gérer les applications ou fichiers communs aux différentes forces de sécurité intérieure, le STSISI a mis en place le « Portail de la Sécurité Intérieure ». Enfin, le SSO exposé sur Internet est dénommé « Curasso ». Tous sont déclinés sous différentes plateformes : Production, Pré-Production, Formation et Développement.

5.3 – Méthodes d'authentification

5.3.1 – Faible : Couple Identifiant / Mot de Passe (EIDAS1)

L'authentification par couple Identifiant / Mot de passe correspond au niveau 1 de la norme EIDAS. C'est une méthode facile à implémenter avec un coût de mise en œuvre très faible. Pour en améliorer la robustesse, il est possible d'imposer aux utilisateurs une politique des mots de passe définissant le nombre ou le type de caractères, leur durée de vie.

5.3.2 – Forte : Carte professionnelle (EIDAS3)

Le niveau 3 de la norme EIDAS correspond ici à l'authentification par carte à puce. La carte professionnelle emporte un certificat contenant l'identifiant de son porteur. Pour être lu, il faut posséder un lecteur compatible et la clef de déchiffrement (code secret du porteur). Les certificats, qualifiés au standard RGS 2*, sont générés par une autorité de certification centrale habilitée et peuvent être révoqués. Le certificat est utilisé par le client pour établir une connexion sécurisée HTTPS avec le serveur.

Cette solution apporte le niveau de protection le plus élevé mais engendre des coûts importants. En effet, elle nécessite l'achat des lecteurs de carte, la fabrication des cartes en elles-mêmes, leur gestion (génération et révocation des certificats), fournir une application pour le changement du code secret et mettre en place une Infrastructure de Gestion des Clefs (IGC) comme EJBCA par exemple.

5.3.3 – Nouveaux besoins

Afin de limiter les risques d'attaques liés à un niveau d'authentification trop faible sur Curasso, il a été décidé de mettre en place un niveau d'authentification plus fort (eIDAS 2 ou 3). L'utilisation de la carte professionnelle étant trop complexe et onéreuse à mettre en place (lecteurs de carte, middleware spécifique, CRL), l'utilisation d'un second facteur lors de l'authentification s'est imposée.

La solution initialement envisagée fut l'utilisation de la plateforme d'envoi de SMS proposée par l'hébergeur Internet de « Curasso ». Une première estimation du coût de la prestation basée uniquement sur les connexions réussies au cours de l'année antérieure avoisinait les 200 000 euros par an. Or, en tenant compte des échecs d'authentification, des SMS envoyés inutilement et de l'augmentation future de la population ayant à avoir accès à « Curasso », le coût grimpait à environ un million d'euros par an.

Le choix a donc été fait d'implémenter l'authentification à double facteur directement dans LemonLDAP::NG. Cette implémentation est l'objet de ce mémoire et constitue le cœur du travail que j'ai réalisé depuis plusieurs mois maintenant. Cette solution avait pour avantages de ne pas nécessiter d'équipement particulier à installer, un coût raisonnable de mise en œuvre (développement, achat de matériels) et de permettre un niveau d'authentification eIDAS 2 voire 3 en fonction du type ou du mode d'enrôlement du second facteur d'authentification.

6 – Synthèse

Dans cette seconde partie, j'ai présenté et détaillé le projet LemonLDAP::NG. Ensuite, j'ai abordé les différents SSO implémentés par la Sécurité Intérieure pour protéger et tracer les accès aux différents services et applications disponibles sur son Système d'Information. Enfin, j'ai mis en exergue la nécessité de mettre en œuvre une politique de sécurité basée sur l'utilisation d'un second facteur d'authentification pour sécuriser les accès depuis notre SSO hébergé sur Internet.

C – LemonLDAP::NG & Seconds Facteurs d'Authentification

LemonLDAP::NG est un projet d'infrastructure. Il s'agit d'une « brique de sécurité » critique et majeure du Système d'Information qui vient se placer entre les utilisateurs et les applications. Cet élément commun aux applications permet de garantir la confidentialité et la traçabilité des actions ainsi qu'une gestion centralisée des autorisations qui sont des conditions nécessaires pour une homologation par la CNIL suite à un audit de sécurité.

Le *Portail* est la seule partie visible aux utilisateurs lors de la procédure d'authentification. Mon projet a consisté en l'implémentation de l'authentification à double facteur permettant d'atteindre le niveau d'authentification substantiel. Pour ce faire, j'ai créé, côté *Portail*, les différents modules permettant de gérer tous les types de seconds facteurs lors des phases d'enrôlement et d'authentification ainsi que le « Gestionnaire 2FA » permettant aux utilisateurs d'afficher ou supprimer leurs différents seconds facteurs. Côté *Manager*, j'ai créé l'« Explorateur de sessions 2FA » et modifié l'« Explorateur de sessions » afin que des « administrateurs » puissent afficher les sessions persistantes avec seconds facteurs et supprimer les seconds facteurs des utilisateurs via le *Manager*.

Tableau I : Synthèse des travaux réalisés

	COTE CLIENT (CoffeeScript)	**COTE SERVEUR** (Perl Orienté Objet)	**TESTS** Unitaires / Non régression	**TEMPS** Dév. + Tests (jours)
Portail (jQuery)	• Gestionnaire 2FA • Formulaires d'Enrôlement / Authentification	• Modules (9) • Option pour imposer enrôlement 1ère cnx • Moteur chargement modules 2FA • Modules Brute Force + Historique	• 550	45 + 25
Manager Applications à page unique (AngularJS)	• Explorateur de sessions 2FA • Explorateur de session	• Modules (2) • Tests de la configuration (nouvelles options)	• 200 • Moteur de rejeux	25 + 15

Tableau II : Différents modules créés

	U2F	TOTP	UBK	Ext2F (OTP)	REST (OTP)	UTOTP	
Enrôlement	X	X	X	BDD utlisateurs	BDD utilisateurs		**3**
Authentification	X	X	X	X	X	X	**6**

1 – Etude fonctionnelle

1.1 – PSSI

La Politique de Sécurité des Systèmes d'Information actuellement mise en place au sein de la Gendarmerie nationale ne prévoit que deux niveaux d'authentification. Le niveau faible (EIDAS1) correspond à une authentification simple par couple Identifiant / Mot de Passe. Celui-ci est requis pour accéder à des fonctionnalités jugées non sensibles ou si une authentification forte n'est plus possible (perte de la carte professionnelle ou révocation de son certificat). Le niveau d'authentification fort (EIDAS3), imposé pour accéder à des applications sensibles, est atteint grâce à l'utilisation de la carte professionnelle et la saisie du code personnel associé. La carte professionnelle est homologuée EIDAS3 car il s'agit d'un second facteur remis à la personne par un tiers de confiance en face à face après vérification de son identité. Puis, la carte est activée par la saisie du demi code valideur et du demi code porteur reçu précédemment par courrier.

Les inconvénients majeurs de l'authentification par carte professionnelle sont le recours obligatoire à un périphérique extérieur (lecteur de carte et pilotes du périphérique), la vérification de la validité du certificat présent sur la carte (liste de révocation) et le coût de déploiement s'il était envisagé d'offrir cette fonctionnalité quel que soit le poste client (internet ou personnel).

L'intégration à LemonLDAP::NG de l'authentification à double facteur (2FA) basée sur un équipement nativement supporté par les navigateurs et ne nécessitant donc aucune installation spécifique apporte à la PSSI un niveau d'authentification intermédiaire voire fort. Tout d'abord, celle-ci permet d'améliorer le niveau de confiance de l'authentification depuis des équipements pour lesquels l'ajout d'un périphérique extérieur est impossible du fait de limitations techniques (tablettes ou smartphones dépourvus de port USB par exemple) ou logicielles (OS propriétaire et verrouillé). Dans le cas d'un auto-enrôlement par l'utilisateur, le double facteur permettrait d'atteindre le niveau EIDAS2. En outre, pour bénéficier d'un niveau EIDAS3, il est tout à fait envisageable d'imposer à l'utilisateur d'enrôler son second facteur depuis un poste professionnel (Intranet) via une application nécessitant une authentification forte. Une autre solution serait qu'une tierce personne enrôle le second facteur de l'utilisateur et le lui remette après vérification de son identité.

En fonction du mode d'authentification utilisé, l'utilisateur se verra attribué un niveau d'authentification valable pour la session en cours. Pour accéder à leurs applications, les chefs de projets peuvent imposer un niveau d'authentification minimum (*$_authLevel*) qui sera utilisé dans le *Manager* pour calculer les règles et autoriser ou non l'accès à la ressource demandée.

Enfin, il faudra modifier la PSSI pour prendre en compte ce niveau d'authentification supplémentaire. Il pourrait être intéressant d'imposer ou proposer aux personnels occupant des postes à responsabilité ou de commandement jugés comme importants d'enregistrer un second facteur en vue de n'autoriser que les niveaux EIDAS2 ou 3. En outre, il sera important de définir les critères de la population habilitée à enrôler ou supprimer le second facteur des utilisateurs ainsi que la procédure de saisine et les protocoles à mettre en œuvre.

1.2 – Population cible

L'implémentation de l'authentification avec second facteur (2FA) dans la version 2.0 de LLNG est une fonctionnalité très attendue et rendue nécessaire au vu des menaces de plus en plus présentes sur Internet. Il est devenu indispensable d'imposer une authentification de niveau substantiel aux personnels qui ont besoin d'un accès depuis l'extérieur du réseau. En 2018, la population éventuellement concernée par l'authentification à double facteur est de l'ordre de plusieurs dizaines de milliers d'utilisateurs, chiffre de plus en croissance.

De plus, la 2FA permettra de proposer aux différentes Directions d'Application un niveau d'authentification intermédiaire pour accéder à leurs ressources.

1.3 – Etat de l'art des seconds facteurs d'authentification

L'authentification à double facteur repose sur l'utilisation de matériels spécifiques. La difficulté de conception de ces objets est de parvenir à trouver une méthode permettant de prouver que l'on est bien détenteur de l'objet au moment de l'authentification. On ne peut évidemment pas se baser sur des choses aussi simple qu'un numéro de série inscrit sur l'objet qui serait transmis par un moyen ou un autre. En effet, il suffirait de récupérer ce numéro une seule fois pour pouvoir s'authentifier sans posséder réellement l'objet.

La plupart des systèmes d'authentification de ce type repose donc sur des clefs de chiffrement (symétriques ou asymétriques) dont la clef va être en écriture seule sur le matériel et à laquelle personne ne pourra avoir accès sinon le fabricant et généralement pas même son détenteur. Les différents 2FA actuels sont décrits dans l'article (« Authentification forte », 2018b).

1.3.1 – Codes à usage unique (OTP)

Lors de la création de son compte, l'utilisateur renseigne soit un numéro de téléphone mobile soit une adresse de messagerie électronique. A la connexion, le service envoie un SMS ou un e-mail contenant un OTP (One Time Password) voire un lien valide quelques minutes. La validation du lien ou la saisie du code à usage unique est obligatoire pour finaliser le processus d'authentification.

SMS et e-mails sont devenus très courants, pratiques et faciles d'usage. Quasiment tout le monde possède un téléphone capable d'interpréter ce type de support et leur réception est presque instantanée. Même en cas de perte, il est toujours possible de transférer son numéro de téléphone ou de consulter ses e-mails depuis un autre équipement.

Le principal inconvénient est la nécessité d'être dans une zone couverte par le réseau de l'opérateur. De plus, il a été prouvé qu'il était possible d'usurper le numéro de la ligne. Celui-ci n'étant pas lié au terminal, il est possible pour des pirates d'intercepter les messages sans que l'utilisateur s'en aperçoive. En outre, cette solution peut devenir onéreuse en fonction du nombre de tentatives de connexion.

Dans le cadre du projet de sécurisation de *Curasso*, l'envoi de SMS aurait été facturé 30 centimes d'euro par message pour un coût annuel estimé à un million d'euros par an.

1.3.2 – Codes à usage unique basés sur le temps (TOTP)

Il s'agit d'une version améliorée du principe de l'OTP. Le TOTP peut être généré par une clef matérielle équipée d'un afficheur LCD (OTP C200 par exemple) ou par une application comme *Google Authenticator* ou *FreeOTP*. Dans le cas de la clef matérielle, il faut renseigner la clef secrète fournie lors de l'achat au moment de la création du compte d'accès au service ou lors de la phase d'enrôlement. Pour les solutions logicielles, la clef secrète est générée par le serveur lors de la phase d'enrôlement. Après avoir installé une des applications, il faut enregistrer la clef sur l'équipement via l'application en flashant le QR code fourni par le service. Cette clef secrète est utilisée par l'application ou l'équipement ainsi que le serveur comme secret partagé. Celui-ci servira de base ou *seed* pour générer d'autres clefs à usage unique. Le QR code étant dépendant de l'URL du serveur, il est possible d'avoir une clef partagée différente pour chaque service ce qui n'est pas possible avec les accessoires OTP.

Ensuite, l'application ou le matériel TOTP calcule un mot de passe à l'aide de cette clef partagée et le temps actuel. Le temps permet de "saler" le secret, c'est-à dire qu'il sert de base pour le calcul du code à générer. Cela donne un mot de passe à usage unique basé sur le temps (Time-based One Time Password) que seules les deux entités, étant synchronisées, peuvent connaître et donc permettre l'authentification de l'utilisateur. Cette méthode est sûre tant que l'équipement du propriétaire du compte n'est pas compromis.

Elle a pour avantage qu'il n'est pas nécessaire d'être sous la couverture réseau de l'opérateur et qu'il est impossible d'intercepter le secret partagé lors d'une phase d'authentification. De plus, des sites Internet comme « https://totp.danhersam.com/ » permettent de générer des TOTP après avoir renseigné une clef à partager, le nombre de chiffres composant le TOTP et sa période de validité. Cette solution a donc également un coût de mise en œuvre pratiquement nul soit en utilisant un smartphone avec *FreeOTP ou Google-Authenticator* soit via un site Internet pour les

utilisateurs qui n'en possèdent pas. La solution matérielle, quant à elle, nécessite l'achat d'un accessoire valant environ 30 euros.

En revanche, il peut arriver que les horloges de l'équipement et du serveur soient désynchronisées et, de ce fait, générer des codes erronés. Autre revers, si un attaquant parvient à récupérer le secret partagé, il peut générer des codes valides sans être repéré.

1.3.3 – Codes à usage unique basés sur un compteur (HOTP)

Cette méthode est dérivée du TOTP. Elle consiste à utiliser un compteur en lieu et place du temps pour « saler » la clef partagée. Après chaque génération d'un HOTP (HMAC-based One Time Password), le compteur est incrémenté côté client et côté serveur.

Les principaux avantages de cette méthode sont qu'un HOTP n'a pas de limite de validité et qu'il n'est pas nécessaire que serveur et client soient synchronisés hormis s'il y a un « raté ». Dans ce cas, il faut resynchroniser le compteur des deux parties.

En revanche, un HOTP est considéré comme étant moins sécurisé qu'un TOTP car plus facile à casser. Dans les deux cas, comme cette technologie utilise un secret partagé, ces systèmes ne sont pas capables d'offrir la non-répudiation. Il n'est donc pas possible d'affirmer que les données ont bien été envoyées par le serveur et reçues par le client.

1.3.4 – Clefs Yubikey

Une *Yubikey* est une petite clef USB qui est vue par le Système d'Exploitation (OS) comme un clavier. Elle est équipée d'une puce contenant une clef AES en écriture seule (pas de lecture possible de la clef), connue également du fabriquant *Yubico*.

Le principe se rapproche du "TOTP" mais au lieu de rentrer un code généré par un équipement, celui-ci est calculé directement par la clef. Elle est obligatoirement connectée à l'ordinateur lors de la phase d'authentification. L'utilisateur doit toucher la clef lorsqu'il y est invité par le navigateur. Le calcul du code est fait instantanément puis celui-ci est transmis au serveur. Cette méthode d'authentification est relativement sécurisée car elle impose à l'utilisateur souhaitant s'authentifier de posséder un objet et génère un code unique à chaque utilisation.

Un autre intérêt d'une *Yubikey* est qu'elle possède deux entrées internes appelées *slots*. Le *slot* 1 est appelé par un appui court et le 2 par un appui long. On peut donc utiliser un *slot* pour un serveur Yubico et l'autre pour son serveur personnel par exemple. Il est également possible d'écrire dans un des *slots* un mot de passe standard classique mais constitué de 64 caractères aléatoires !!!

Enfin, une *Yubikey* peut également être utilisée comme une clef U2F standard et supportant le protocole NFC. Celle-ci doit également être enrôlée avant utilisation.

Le seul inconvénient pourrait être le coût d'achat compris entre 40 et 60 euros.

1.3.5 – Accessoires U2F / FIDO Alliance

Ils reposent sur un standard ouvert nommé U2F (Universal Second Factor) développé par Google, Yubico et NXP, société à l'origine des puces NFC. Le standard U2F est à présent maintenu par l'alliance FIDO ("Fast IDentity Online") qui regroupe plusieurs entreprises et organismes. Les clefs de sécurité U2F peuvent également être utilisées comme 2FA avec les services compatibles tels que Google, Dropbox, GitLab, etc...

Différents modèles de clefs ou matériels U2F existent comme des périphériques NFC ou des cartes à puce dans une plage de prix allant de 10 à 50 euros. L'utilisateur doit simplement connecter sa clef dans un port USB, approcher son périphérique NFC ou toucher sa carte à puce.

Le principal avantage d'une clef U2F par rapport à un OTP est qu'il s'agit d'un véritable second facteur physique, facile à mettre en œuvre et utilisable uniquement avec les sites où elle a été enrôlée. En outre, elle ne peut pas être interceptée à la différence des SMS. Il s'agit d'une des 2FA la plus sécurisée actuellement.

Les seuls reproches à faire seraient éventuellement son coût compris entre 9 euros pour une clef basique et 40 euros pour une clef NFC ainsi que les risques de perte ou de panne.

En outre, si la Gendarmerie devait acheter et fournir des clefs U2F à l'ensemble des utilisateurs se connectant à *Curasso*, le coût resterait de toute façon inférieur à l'utilisation de la plateforme de SMS proposée par l'hébergeur et serait amorti dès la première année d'exploitation (quelques dizaines de milliers de clefs U2F à 10 euros).

1.3.6 – Solutions avancées

La catégorie des solutions avancées regroupe les systèmes de reconnaissance faciale, vocale ou de lecture d'empreintes digitales ou rétiniennes. Tous ces systèmes requièrent l'utilisation d'un périphérique extérieur et des pilotes spécifiques. L'énorme avantage de ceux-ci est qu'ils sont très difficiles à contourner ou pirater. En revanche, s'ils le sont, ils deviennent inutilisables car impossibles à modifier. De plus, la voix ou les empreintes digitales peuvent varier en fonction de l'humeur ou de conditions extérieures. En outre, ces systèmes restent très onéreux à déployer.

1.3.7 – Synthèse

Quel est le meilleur second facteur ? Cela dépend du but recherché.

Si les critères de sécurité et de protection de la vie privée sont prédominants, il faut privilégier les clefs U2F ou *Yubikey*, accessoires fiables et robustes. Elles ne peuvent pas être interceptées et sont basées sur un chiffrement asymétrique. Avec cette solution, le coût n'est pas négligeable mais reste raisonnable.

Si l'on privilégie l'aspect pratique, l'emploi des SMS est un procédé simple, efficace et relativement sécurisé. En revanche, de récentes études ont montré qu'ils peuvent être interceptés, détournés et imposent d'être sous une couverture réseau.

La solution équilibrée entre sécurité et facilité est probablement l’utilisation des TOTP. Mais, il s’agit d’une solution utilisant un chiffrement symétrique. Il est donc impératif de veiller à sécuriser la clef privée tant côté client que serveur et de ne pas perdre ou casser l’équipement utilisé pour générer le code à usage unique. Dans le cadre de ce mémoire, il a donc été arrêté par Xavier Guimard et moi-même de n’implémenter que ces types de seconds facteurs car jugés ayant un niveau de sécurité suffisant, le plus faible d’entre-eux étant le TOTP.

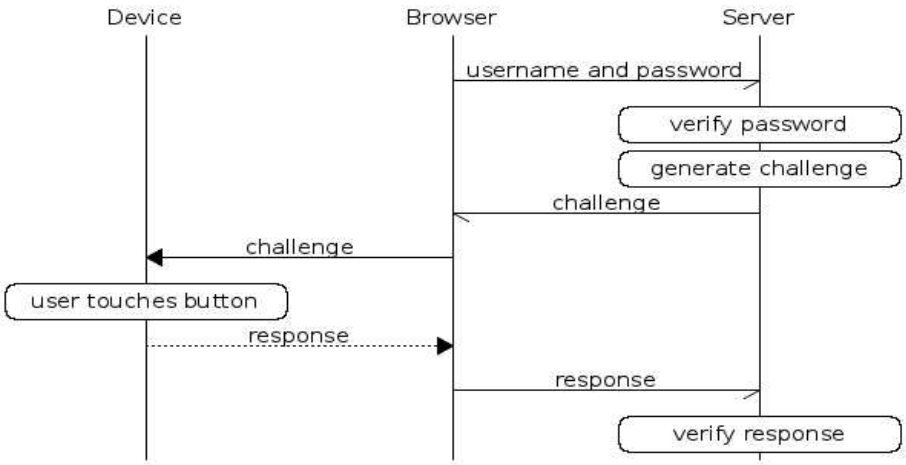

Figure 15 : Cinématique Challenge - Réponse

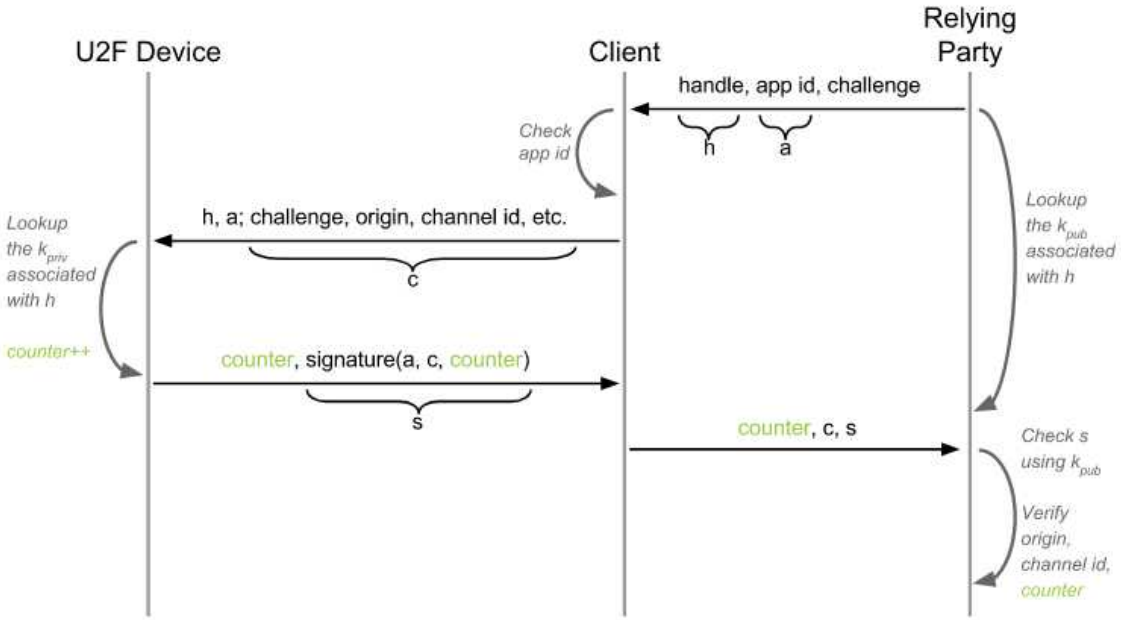

Figure 16 : Cinématique U2F détaillée

- Recherche du KH associé à l'équipement puis envoie du challenge, KH et Id du service (RP)

- Client (navigateur) vérifie l'Id, génère *HashServeur* (origin, channel) avec les données vues du navigateur et transmet l'ensemble (chaîne) à l’équipement U2F

- Équipement U2F détermine clef privée avec le KH, incrémente le compteur et signe la chaîne

- RP décode la chaîne avec la clef publique puis vérifie les données et le compteur

2 – Etude technique : Normes & Analyse

2.1 – Clefs U2F

Ces clefs étant des équipements USB, elles communiquent avec le système hôte grâce au protocole HID. La spécification HID (« Human interface device », 2018) est une couche d'abstraction plus élevée que le protocole USB utilisée pour simuler un clavier. De ce fait, les clefs U2F ne nécessitent pas l'installation de pilotes spécifiques pour être reconnues par l'OS. Pour ce qui est de leur exploitation logicielle, la norme U2F est à présent nativement supportée par les navigateurs tels que Chrome à compter de la version 38, Opéra 40, IE 10 et Firefox 57. Seules les clefs de marque « Keydo » produites par la société marseillaise « NeoWave » ont nécessité une configuration spécifique détaillée par le fabriquant pour fonctionner sous OS Linux (cf Annexe 1). Les clefs U2F sont protégées contre la copie et la rétro-ingénierie par l'utilisation des puces de chiffrement avancé.

Une fois la connexion établie entre le navigateur et la clef, le navigateur envoie à la clef une requête d'authentification basée sur le principe du challenge-réponse en utilisant une méthode de chiffrement asymétrique (clef publique / clef privée). La clef publique est utilisée pour chiffrer un challenge que seule la clef privée peut décoder (cf. figure 15).

D'après les spécifications décrites dans le document (« Universal 2nd Factor (U2F) Overview.pdf », s. d.) dont un extrait est joint en annexe 2, les clefs et le protocole U2F se doivent de garantir la vie privée et la sécurité. Pour ce faire, la clef U2F génère une biclef privée / publique pour chaque fournisseur de service ou site (cf. figure 16). Le périphérique transmet ensuite au service une clef publique et un *KeyHandle* (KH) qui lui est spécifique via le navigateur lors de la phase d'enrôlement (*registration*). Plus tard, quand l'utilisateur essaye de s'authentifier, le site retourne le *KeyHandle* à l'accessoire U2F via le navigateur. Grâce au KH, l'équipement U2F est capable de recalculer la clef privée associée au serveur et génère une signature qu'il lui retourne pour valider la présence du périphérique U2F (cf annexe 2).

Par conséquent, le KH est juste l'identifiant d'une clef privée particulière de l'accessoire U2F. La paire de clef générée durant la phase de *registration* est spécifique au fournisseur de service. Pour calculer ce KH, le périphérique U2F se base sur un *HashServer* transmis par le navigateur qui est une combinaison du protocole utilisé, du nom du serveur et du port vus par le navigateur. En outre, l'accessoire U2F encode l'URL du serveur dans le KH.

Lors de la phase d'authentification, le serveur retourne le KH au navigateur qui le transmet à son tour avec le *HashServer* au périphérique U2F. Grâce au *HashServer*, l'accessoire U2F s'assure que le KH a bien été transmis depuis le serveur correspondant avant de jouer la phase de signature. Le serveur vérifie également que la clef publique et le KH ont bien été transmis par le périphérique U2F et non par un autre site.

Pour la phase de signature, le navigateur transmet à l'accessoire U2F, en plus du KH et du *hashServer*, un *HashClient* constitué d'un challenge aléatoire et du nom du serveur. Avec ces trois éléments, l'accessoire U2F vérifie l'origine du serveur grâce au *hashServer*, signe le *hashClient* avec sa clef privée associée (déterminée par le KH) et le retourne au service. Le service vérifie la signature en déchiffrant le challenge avec la clef publique associée au compte (« La signature numérique - Fonctionnement », s. d.)

En outre, les phases d'enrôlement et d'authentification sont soumises à un test de présence de l'utilisateur. Celui-ci est invité par le navigateur à toucher l'équipement pour déclencher l'action. Ceci a pour but d'éviter qu'une action ne soit initiée à l'insu de l'utilisateur, par un logiciel malveillant par exemple.

La norme U2F et son API (« Balfanz et al. - FIDO U2F JavaScript API.pdf », s. d.) (cf. annexe 3) exposent côté client les deux méthodes suivantes. La première, « Register », qui génère pour un service une nouvelle clef utilisée pour l'enrôlement. La seconde, « sign », pour signer le challenge afin de valider la clef publique précédemment enregistrée. Il est recommandé d'encoder la clef publique générée en base64 pour pouvoir la sauvegarder plus facilement.

Côté serveur, pour faciliter l'appel en Perl des méthodes U2F, nous avons choisi d'utiliser le module (« Crypt::U2F::Server::Simple », s. d.) dans sa version corrigée par Xavier Guimard (0.43). En effet, les fonctions manipulant les clefs publiques étant écrites en langage C, celles-ci tronquaient la valeur de la clef si elle contenait un '0' car ce caractère correspond au marqueur de fin de chaîne en C. Ce module fournit donc les quatre fonctions suivantes : registrationChallenge(), registrationVerify(), authenticationChallenge(), authenticationVerify().

2.2 – Clefs Yubikey

Yubico implémente un protocole propriétaire permettant de déléguer l'authentification à un service intermédiaire (« Yubikey de Yubico », 2017)

À chaque appui sur le bouton de la clef, celle-ci va émettre une chaîne composée avec son identifiant constitué de 12 caractères, d'un compteur de session qui s'incrémente à chaque branchement de la clef, d'un compteur d'horloge s'incrémentant 8 fois par seconde et d'un compteur d'utilisation qui s'incrémente à chaque appui sur le bouton. La chaîne complète est chiffrée avec la clef AES 128 bits interne puis est envoyée à l'application qui souhaite vous authentifier.

L'application cliente contacte alors le serveur de Yubico et lui soumet la chaîne chiffrée. Le serveur possédant lui aussi la clef privée, déterminée grâce à l'identifiant de la clef, il est en mesure de déchiffrer les données et de les vérifier. S'il est capable de déchiffrer les données et que l'identifiant d'utilisateur correspond bien à celui associé à la clef, la clef présentée est valide. De plus, si les valeurs de tous les compteurs sont bien strictement supérieures à celles de la dernière chaîne validée, on est en présence d'un nouvel identifiant et non d'un rejeu d'une chaîne interceptée.

La dernière propriété consistant à vérifier les compteurs est assez intéressante. Même si un pirate parvient à subtiliser une clef, à l'utiliser pour générer un ensemble de chaînes puis à la restituer à son propriétaire, la prochaine utilisation de la clef invalidera l'intégralité de toutes les chaînes enregistrées. En effet, le compteur de session ou d'utilisation doit être inférieur au compteur interne de la clef.

Une fois vérifiée, le serveur Yubico répond à l'application intermédiaire via l'utilisation d'un chiffrement asymétrique si la chaîne est valide ou pas. L'application autorise alors ou non l'accès à la ressource protégée.

Tous les logiciels nécessaires à l'intégration d'une clef Yubikey sont libres et intégrés nativement à la plupart des distributions GNU/Linux. Il est également possible de créer un serveur de validation privé indépendant de celui de Yubico mais peu d'applications supportent un serveur personnalisé. L'enrôlement de l'application intermédiaire nécessite de se connecter à l'interface web suivante : https://upgrade.yubico.com/getapikey/. Il faut ensuite renseigner une adresse email puis s'authentifier en générant un OTP à l'aide d'une Yubikey pour obtenir un identifiant client et une clef secrète.

Le principal inconvénient concernant l'implémentation de ce protocole est l'obligation d'avoir accès à une connexion Internet si l'on ne souhaite pas recourir à un serveur de validation privé. De plus, il s'agit d'un protocole propriétaire moins sécurisé que la norme U2F.

2.3 – TOTP

TOTP et HTOP sont basés sur le même algorithme. Dans le cas d'un HTOP, le salage utilise un compteur incrémentiel quant au TOTP , il s'agit d'un compteur temporel. L'algorithme de génération est expliqué dans l'article suivant : (cez40, 2014)

Le compteur (TC) est un compteur basé sur le temps exprimé sous la forme « epoch Posix » correspondant au nombre de secondes écoulées depuis le 01 janvier 1970 minuit. K est la clef secrète partagée (chiffrement symétrique) entre le client et le serveur.

* La première étape consiste à générer un HMAC-SHA1 qui nous retourne 20 octets (160 bits, taille d'un hash SHA1), en appliquant la formule suivante (« Keyed-hash message authentication code », 2018) :

$$HMAC(K,TC) = SHA1(K \text{ xor } 0x5c5c\ldots5c \ || \ SHA1((K \text{ xor } 0x3636\ldots36) \ || \ TC))$$

avec :

- K est converti en hexadécimal
- TC = UnixTime / TS où TS représente la durée de validité du TOTP, généralement 30s, convertie en 8 octets

Un HMAC (keyed-hash message authentication code) est un type de code d'authentification de message (CAM) calculé en utilisant une fonction de hachage cryptographique en combinaison avec une clef secrète. Ici, il est utilisé pour vérifier simultanément l'intégrité des données et l'authenticité d'un message. Les fonctions de hachage cryptographique telles MD5, SHA1 ou SHA256 permettent d'obtenir l'empreinte numérique d'un fichier.

* Cette deuxième étape consiste à extraire une chaîne de 31 bits (4 octets – 1 bit) à partir du HMAC calculé lors de l'étape 1 en utilisant la fonction *DynamicTruncation* qui prend en paramètre une chaîne de caractères : DT(String)

- String, le HMAC calculé à l'étape 1, est une chaîne composée de 20 octets (String[0] à String[19])
- On extrait les 4 bits de poids faible de l'octet 19. La valeur est contenue dans OffsetBits
  - OffsetBits est donc compris entre 0 et 15 (4 bits) car 4 + 15 = 19 max
  - OffsetBits donne la position dans String à partir duquel on extrait 4 octets (32 bits)
  - Puis ET logique avec le masque : 7FFFFFFF pour supprimer le bit de signe (31 Bits)

Exemple avec 'y = 0101b = 5d' :

```
 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19
XX XX XX XX XX XX XX XX XX XX XX XX XX XX XX XX XX XX XX XY
               AA BB CC DD
               ^^ ^^ ^^ ^^                                 ^
```

Sbits = DT(HMAC(K,TC))

* Enfin, il reste à générer un token (T) à partir de la chaîne « Sbits » calculée à l'étape précédente :

- Convertion de Sbits en décimal = Snum
- T = Snum mod 10^Digits avec Digits le nombre de chiffres du Token, généralement 6.

La clef partagée K est calculée par un générateur de nombres pseudo-aléatoires (« Générateur de nombres pseudo-aléatoires », 2018) puis encodée en base32. Elle est associée avec l'URL du portail et proposée à l'utilisateur par le navigateur sous forme d'un QRCode généré à l'aide de la bibliothèque *JavaScript* « QRious » (Mercer, 2011/2018).

2.4 – Autres OTP (API REST ou Externes)

Pour permettre l'utilisation de seconds facteurs personnalisés ou nécessitant de se connecter à des services tiers, LLNG offre deux interfaces supplémentaires que sont les modules « REST » et « Ext2F ».

Le module « Ext2F » peut être activé pour exécuter une commande ou un script externe indépendamment du langage utilisé (bash, C, Python, etc...). Le script et ses arguments sont séparés, placés dans un tableau puis exécutés par un appel système Perl (« system - perldoc.perl.org », s. d.). Le fait de passer un tableau comme argument à la fonction Perl « system » permet de se prémunir des attaques par injection de code.

L'interface « REST » permet de faire appel à un web service externe via une API pour soumettre et valider le second facteur de l'utilisateur au moment de sa connexion. L'API, pour Application Programming Interface (« Qu'est-ce qu'une api REST ? », s. d.), est la partie du programme qu'on expose officiellement au monde extérieur pour manipuler celui-ci. Cette dernière permet d'entrer des données et de les récupérer à la sortie d'un traitement. Initialement, une API regroupe un ensemble de fonctions ou méthodes, leurs signatures et ordre d'usage pour obtenir un résultat. Une API REST se doit d'être sans état. La communication entre le client et le serveur ne doit pas dépendre d'un quelconque contexte provenant du serveur. Ainsi, chaque requête doit contenir l'ensemble des informations nécessaires à son traitement. Cela permet de traiter indifféremment les requêtes de plusieurs clients via de multiples instances de serveurs. Les actions que nous souhaitons effectuer grâce à l'API doivent être associées à des méthodes HTTP (POST, GET, PUT ou DELETE). Pour chaque réponse renvoyée par l'API, un code HTTP doit être retourné (200, 400, 403, etc …). Ce code correspond à l'état de la requête et dépend de la réussite ou non de celle-ci. Les différents paramètres sont transmis à l'API sous le format JSON et l'API doit retourner le code de résultat sous la forme d'un JSON contenant une clef unique de la forme « result » : « true/false ».

Contrairement aux modules *U2F*, *TOTP* ou *Yubikey*, *REST* et *Ext2F* ne permettent pas l'auto-enrôlement du second facteur par l'utilisateur. En effet, ceux-ci permettent d'interconnecter LLNG avec des services tiers comme une plateforme de SMS ou une application extérieure.

3 – Environnement de développement

3.1 – Licence & Droits d’auteur

3.1.1 – GNU

(« GNU », 2017) est à l’origine un système d’exploitation libre créé en 1983 par Richard Stallman, maintenu par le projet GNU. Son nom est un acronyme récursif qui signifie en anglais « GNU’s Not UNIX » (littéralement, « GNU n’est pas UNIX »). Il reprend les concepts et le fonctionnement d’UNIX. Les logiciels composant GNU sont généralement utilisés en association avec des logiciels libres issus d'autres projets tels que le noyau Linux (« Noyau Linux », 2018) crée par Linus Torvald.

La licence de ce projet est la GNU/GPL *(GNU General Public License)*. Elle est aujourd’hui très utilisée pas les développeurs de logiciels libres pour protéger leurs travaux.

Le strict respect de cette licence et du droit d’auteur m’a valu quelques « rappels à l’ordre » de la part de mon tuteur, notamment en ce qui concerne l’utilisation de contenu disponible sur Internet. Par conséquent, j’ai pris soin de n’importer et d’utiliser que du contenu, comme des images par exemple, issu uniquement de la *CommonWikimedia* et de référencer l’utilisation, la source et l’auteur dans le fichier « COPYING » situé à la racine du projet.

3.1.2 – GPL

LLNG est distribué sous licence GPL (« Licence GNU », s. d.). La « GNU General Public Licence » pose quatre grands principes de base garantissant que personne ne doit être limité par les logiciels qu'il utilise. Elle définit quatre libertés fondamentales que tout utilisateur doit posséder :

- la liberté d'utiliser le logiciel à n'importe quelle fin
- la liberté de modifier le programme pour répondre à ses besoins
- la liberté de redistribuer des copies
- la liberté de partager avec d'autres les modifications faites

Quand un programme offre à ses utilisateurs toutes ces libertés, il est qualifié de logiciel libre. Les développeurs qui écrivent des logiciels peuvent les publier sous les termes de la GNU GPL. Ce faisant, leur logiciel sera libre et le restera, indépendamment de qui modifiera et distribuera ce logiciel. Cette notion est le *copyleft.* Le logiciel est bien soumis au droit d'auteur, mais plutôt que d'utiliser ces droits pour restreindre l'utilisation que peuvent en faire les utilisateurs, ils sont utilisés pour garantir que tous les utilisateurs auront ces libertés.

3.1.3 – DFSG

Les principes du logiciel libre selon Debian « Debian Free Software Guidelines » sont un ensemble de caractéristiques définissant les critères d'appréciation qui permettent de déclarer si un logiciel est « libre» pour les développeurs de la distribution GNU/Linux Debian. Ces principes ont été repris par de nombreuses organisation et sont aujourd'hui la référence principale pour déterminer si une licence logicielle est libre.

Pour Debian, un logiciel libre doit respecter 9 principes fondamentaux :

- Redistribution libre et gratuite : la licence d'un composant de Debian ne doit pas empêcher quiconque de vendre ou de donner le logiciel
- Code source : le programme doit inclure le code source
- Applications dérivées : la licence doit autoriser les modifications
- Intégrité du code source : la licence peut défendre de distribuer le code source modifié
- Aucune discrimination de personne ou de groupe
- Aucune discrimination de champ d'application
- Distribution de licence
- La licence ne doit pas être spécifique à Debian
- La licence ne doit pas contaminer d'autres logiciels

Le simple fait qu'une licence soit libre ne signifie pas nécessaire qu'elle soit conforme au DFSG. En effet, par exemple, Debian considère que la licence CreativeCommon est compatible avec ses principes très stricts uniquement à partir de la version 3.0.

Debian a défini 3 « tests » pour vérifier la conformité au DFSG d'une licence :

- le test du dissident : ne pas imposer de publier ou déclarer les modifications des sources ou simplement l'utilisation
- le test de l'île déserte : pouvoir modifier et publier ses modifications sans contraintes
- le test des tentations du Diable : la licence ne doit pas permettre de restreindre les principes fondamentaux même ultérieurement

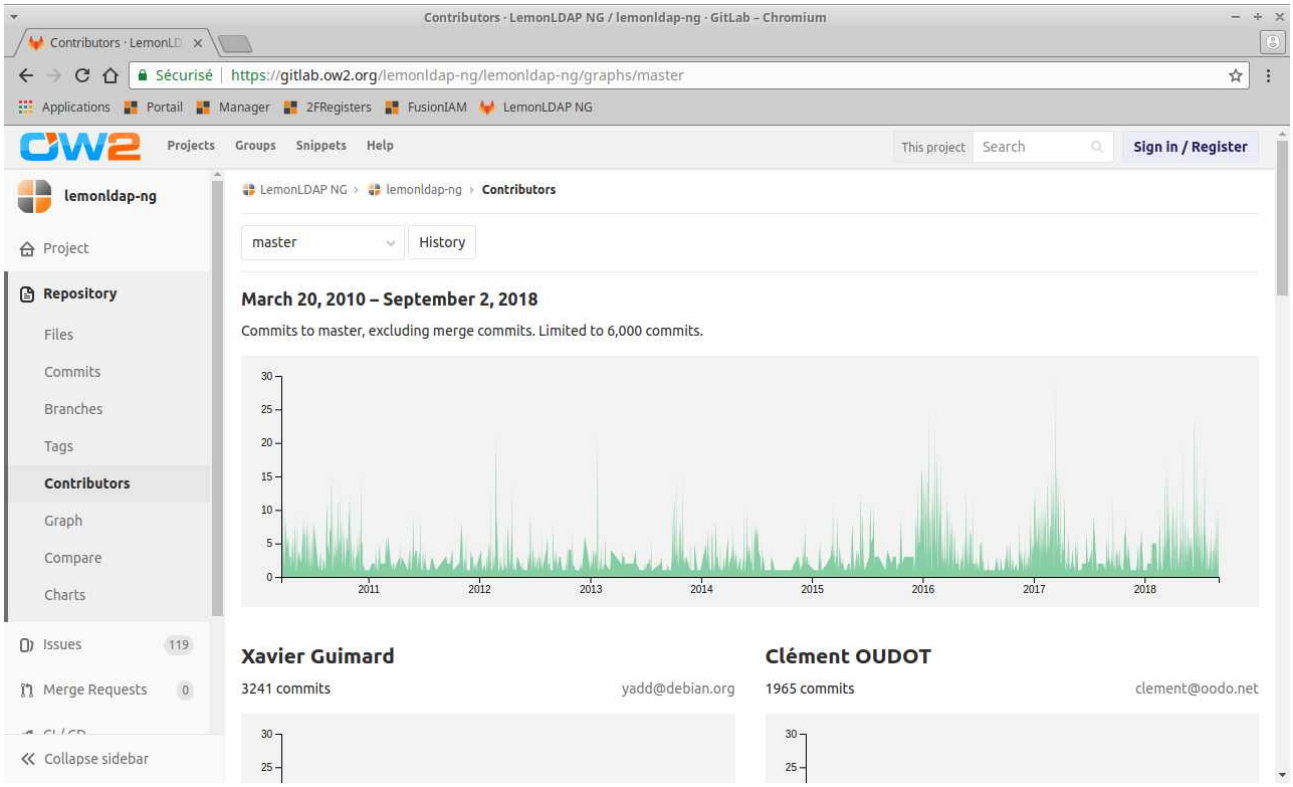

Figure 17 : Plateforme GitLab

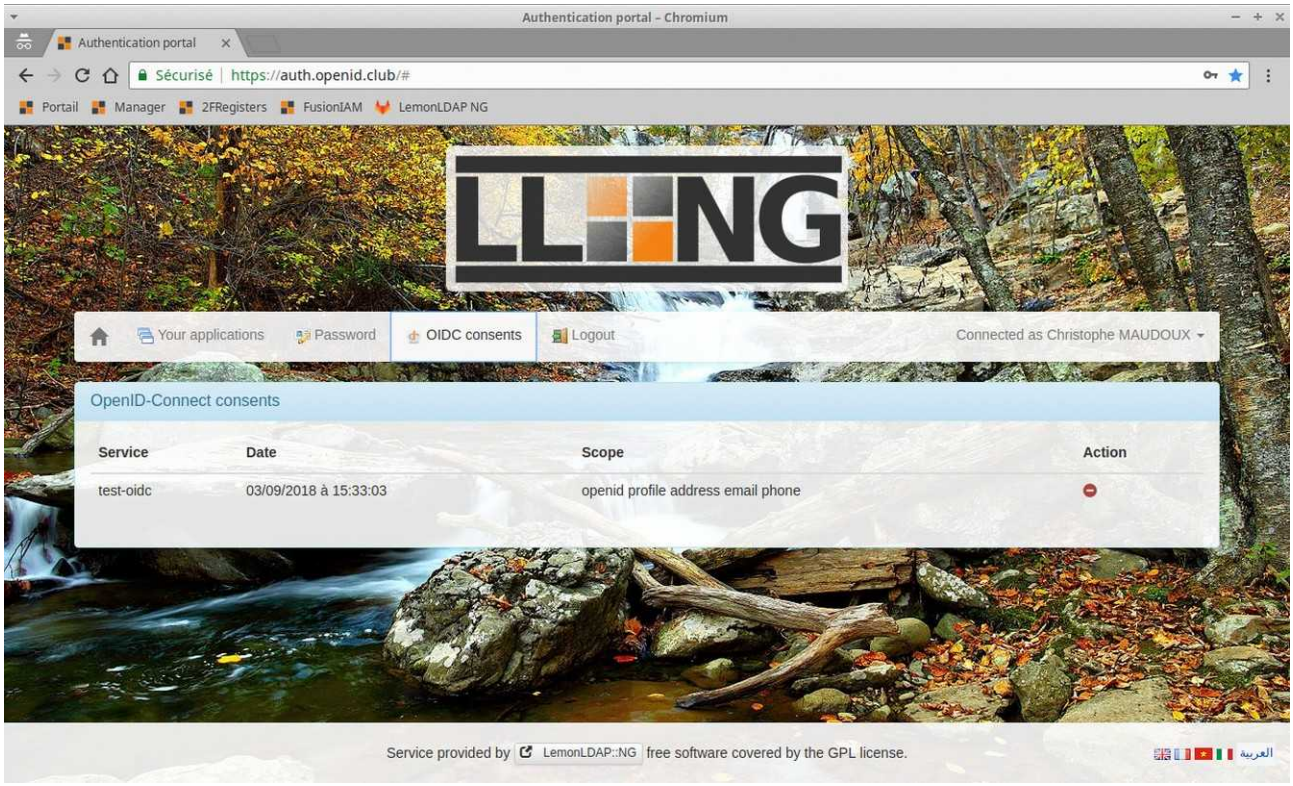

Figure 18 : Plateforme FusionIAM

3.2 – Plateformes

3.2.1 – GitLab

Le développement de LLNG est basé sur le modèle collaboratif et les sources sont disponibles sur la plateforme GitLab (cf. figure 17). Il m'a donc fallu commencer par apprendre à utiliser cet outil. Il s'agit d'une forge logicielle apportant les fonctionnalités suivantes :

- Tout d'abord, elle permet de gérer des dépôts Git ainsi que les utilisateurs et leurs droits d'accès aux dépôts.

- Elle offre une authentification pouvant utiliser deux facteurs et la connexion à un annuaire LDAP.

- Elle permet de gérer l'accès par branche à un dépôt, d'effectuer des examens de code et de renforcer la collaboration avec les demandes de fusion.

- Enfin, chaque projet possède un outil de ticket et un wiki.

La fonction de gestion des versions est fournie par le logiciel Git (« git », 2018) également créé par Linus Torval. Git ne repose pas sur un serveur centralisé mais utilise un système de connexion pair à pair (HTTP ou SSH). Le code informatique développé est stocké non seulement sur l'ordinateur de chaque contributeur du projet, mais il peut également l'être sur un serveur dédié. C'est un outil de bas niveau qui se veut simple et performant, dont la principale tâche est de gérer l'évolution du contenu d'une arborescence. Git indexe les fichiers d'après leur somme de contrôle calculée avec la fonction de hachage SHA-1. Quand un fichier n'est pas modifié, la somme de contrôle ne change pas et le fichier n'est stocké qu'une seule fois. En revanche, si le fichier est modifié, les deux versions sont stockées sur le disque.

Après avoir créé mon compte en février 2018, j'ai *« cloné » (dupliqué)* la branche 2.0 de LLNG en local et « fourché » pour commencer à travailler sur ma propre branche. Les premiers mois, n'ayant qu'un profil de « contributeur », je devais faire une requête de fusion vers la branche principale « master » après chaque évolution majeure. Cette « merge request » était vérifiée puis validée par mon tuteur ou Clément Oudot avant fusion. Après quelques mois de pratique, nécessaires pour apprendre à utiliser la forge, j'ai été très fier d'être promu « propriétaire » en avril et « mainteneur » fin mai, statut me permettant de publier directement mes modifications sur la branche *master*, de m'attribuer des tickets ou fusionner des branches. Ceci étant, en cas de doute, je n'hésitais pas à soumettre mes travaux à l'équipe pour vérification et validation.

Une autre particularité de la forge *GitLab* est qu'elle permet l'intégration continue et que la batterie de tests de non régression est exécutée après chaque publication de modifications (*git push*). En cas d'échec, la construction nocturne des paquets spécifiques à chaque distribution n'est pas effectuée et le développeur fautif reçoit une notification d'échec de construction.

3.2.2 – FusionIAM

En plus de la forge *GitLab*, Clément Oudot nous a mis à disposition une plateforme Internet de développement basée sur la solution dont il est co-développeur, « FusionIAM » (cf. figure 18). Cette plateforme est particulièrement utile pour tester les fonctionnalités de fédération d'identités.

Le projet *FusionIAM* est une solution « clef en main » de gestion des identités et de contrôle d'accès emportant LLNG, un annuaire LDAP et des outils spécifiques d'administration LDAP.

Au delà de ce mémoire, cela fait maintenant six mois que j'ai intégré l'équipe de développement de LLNG et contribue à la version 2.0. Je participe également à maintenir les autres branches comme la 1.9 notamment le rétro-portage de correctifs.

3.3 – Projet collaboratif

3.3.1 – Travail en équipe

Le développement collaboratif de LLNG implique d'échanger ou d'éventuellement consulter les autres membres de l'équipe pour l'ajout, la modification ou la correction des différentes fonctionnalités. Un avantage majeur de ce mode de fonctionnement est également de pouvoir, en cas de besoin, demander de l'aide ou solliciter un conseil.

En outre, les décisions ou solutions sont prises ou choisies de façon collégiale ce qui est très gratifiant et rassurant pour les nouveaux membres. Ce mode de fonctionnement est très enrichissant mais impose le respect de règles ou conventions tant sur la forme que sur le fond.

Les échanges avec l'équipe, en particulier avec Clément Oudot qui a eu l'occasion d'installer LLNG dans différentes administrations ou entreprises tant en France qu'à l'étranger, m'ont permis d'aborder LLNG de façon plus large, avec une vue plus globale. En effet, je n'avais qu'une vision très « gendarmerie » de la manière dont était installé et utilisé le SSO, tout particulièrement en ce qui concerne le second facteur. Celui-ci n'étant pas encore en place au sein de la Gendarmerie nationale, je ne voyais pas comment il pouvait être mis en œuvre. La politique la plus répandue dans les autres organisations est la mise en place de TOTP ou de clefs U2F. L'emploi d'un second facteur y est imposé pratiquement à tous les personnels car ceux-ci doivent se connecter au réseau de l'entreprise depuis des postes connectés à Internet.

3.3.2 – Documentations

Tous d'abord, la politique *Debian*, très stricte au sujet de la fuite de données, interdit de référencer des ressources externes directement dans le code. Ceci nous a donc imposé de compiler la documentation en ligne dans des paquets spécifiques en plus de la rédaction des fichiers POD (Plain Old Documentation) (Lieuze, s. d.) présentant les différentes fonctions, leurs architecture et utilisation.

Ensuite, toutes les caractéristiques de la solution LemonLDAP::NG sont détaillées dans la documentation disponible sur le site du projet https://lemonldap-ng.org/documentation/. J'ai consacré pratiquement tout le mois d'avril à lire et relire la documentation existante pour appréhender, comprendre et rédiger la deuxième partie de ce mémoire. Cet important travail de lecture et relecture de cette documentation a été un échange gagnant-gagnant tout comme les cours d'anglais que j'ai pu suivre sur mon temps de travail. En effet, la documentation ayant presque été intégralement rédigée en anglais par le fondateur du projet lui-même, des points qui peuvent paraître évidents ne le sont pas forcément pour un lecteur extérieur. Cette phase a donc également été l'occasion de la valider ou de la faire évoluer. Après plusieurs semaines d'interrogation et d'apprentissage, j'ai même pu être en mesure de relever et corriger quelques incohérences ! J'en ai profité également pour rédiger ou mettre à jour les pages relatives aux différents seconds facteurs d'authentification. Toute cette documentation ainsi que les menus et les échanges sur la plateforme de développement sont rédigés en anglais. Seuls les menus des interfaces sont traduits vers une dizaine de langues telles l'espagnol, le vietnamien, l'italien ou l'arabe. Les fichiers de langue sont mis à disposition de la communauté pour traduction sur la plateforme *Transifex*, un espace collaboratif de traduction.

Enfin, il est très important d'inclure des commentaires dans le code source et d'utiliser des constructions claires et détaillées pour permettre aux autres développeurs de reprendre ou poursuivre le code. J'ai donc pris soin de respecter cette règle et d'ajouter un maximum de messages de « debug » pour permettre de suivre les différentes étapes d'exécution, notamment pour permettre la correction d'erreurs.

3.3.3 – Règles, Conventions & Architecture globale

Pour faciliter la lecture et améliorer la maintenabilité du code, des règles d'usage et des conventions d'écriture m'ont été imposées lors de mon intégration à l'équipe. En effet, j'ai eu à apprendre à utiliser des outils comme (« PerlTidy », 2018) pour formater le code Perl et « yuiCompressor » pour minifier le code *JavaScript*.

De plus, ce qui m'a posé le plus de difficultés avec *AngularJS*, a été l'apprentissage du langage (« CoffeeScript », 2018). Il m'aura fallu environ deux semaines de pratique et lire le guide suivant : (Burnham, 2015). *CoffeeScript* est un langage de programmation, qui se compile en *JavaScript*. Celui-ci ajoute du « sucre syntaxique » le rendant plus agréable à écrire comme à lire afin d'améliorer la brièveté et la lisibilité du JavaScript, tout en lui ajoutant des fonctionnalités. Les particularités de *CoffeeScript* sont l'absence de caractère pour délimiter les blocs de code mais l'utilisation de l'indentation comme en « Python » et la déclaration des paramètres des fonctions avant la fonction elle-même ce qui est un peu perturbant au début. Le résultat est compilé de façon prévisible en JavaScript, et les programmes peuvent être écrits avec moins de code (typiquement un tiers de lignes en moins) sans effet sur la vitesse d'exécution.

L'architecture du projet LLNG est modulaire et m'a demandé une dizaine de jours pour bien l'appréhender. La structure de son arborescence peut être vue comme quatre composants indépendants, correspondants aux répertoires *Portail*, *Manager*, *Handler* et *Common*. Chacun de ces répertoires est basé sur le même schéma et contient, entre autres, les sous-répertoires « lib », « site » et « t » ainsi que les fichiers « MANIFEST », « bower.json », « README » et « debian/control ».

Les répertoires « lib » contiennent l'ensemble du code et des modules exécutables. Le nom de chaque module correspond à sa position dans l'arborescence. Dans « site », nous trouvons tous les fichiers concernant l'interface web notamment tous les *CoffeeScript*, les *JavaScript* générés, les *templates*, le CSS et les fichiers de langues pour la traduction de l'interface utilisateur. Je terminerai avec les répertoires « t » qui contiennent l'ensemble des tests de non régression. Ces tests représentent environ 40 % de l'ensemble du code de LLNG et leur temps d'exécution est d'environ 6 minutes ce qui commence à être important mais nécessaire. Ces tests de non régression ont pour but de s'assurer que toute nouvelle évolution ou modification ne « casse » pas une fonctionnalité existante.

Pour chaque répertoire correspondant à un composant de LLNG (Portail, Manager, etc.), le « MANIFEST » décrit l'ensemble des fichiers qu'il contient ce qui permet d'en vérifier l'intégrité. L'utilitaire « bower » est un outil logiciel de gestion des paquets. Il est utilisé pour vérifier et télécharger les mises à jour des librairies nécessaires à LLNG. Celles-ci sont décrites dans le fichier « bower » qui est lu par l'utilitaire éponyme. Le fichier « README » est un guide succinct d'installation spécifique à chaque module. Enfin, chaque arborescence de module contient un fichier « debian/control ». Il contient les informations les plus importantes sur le paquet source et sur les paquets binaires qui seront créés. Il permet surtout de préciser les dépendances. Il s'agit d'une des caractéristiques les plus puissantes du système de paquets Debian. Les paquets peuvent être liés entre eux et être dépendants, recommandés ou suggérés.

3.4 – Tests & Intégration continue

En plus d'être lancés par la plateforme de développement continu, les tests peuvent et doivent être exécutés localement par le développeur avant la publication de ses révisions.

Comme précisé dans la deuxième partie, tous les composants de LLNG sont construits avec le module PLACK implémentant le protocole PSGI. Il est donc possible de construire et passer des requêtes HTTP aux modules pour tester l'ensemble des fonctionnalités. La batterie de tests compte actuellement 192 tests pour la partie *Common*, 166 pour les *Handlers*, 4119 pour le *Portail* et 3854 pour le *Manager.* Ces tests permettent de vérifier si les codes HTTP répondus correspondent bien à ceux attendus (200, 302 en particulier), de contrôler le contenu et l'intégrité des pages reçues en testant la présence d'une balise spécifique (un script par exemple) ou de valider l'envoi de formulaire en vérifiant la présence des champs requis.

Tableau I : Synthèse des attaques

Attaques	Principes	Cibles	Protections
Injection (XSS)	Insertion code malicieux	Formulaires + SQL + Entêtes, Règles & Macros	Expressions régulières + Préparation / Exécution (échappement) + Confinement cage sécurisée
Interception (MitM)	Vol jeton par écoute trames	Trafic réseau	HTTPS
Rejeu (CSRF)	Actions insu et au nom utilisateur	Requêtes soumission formulaires	Jeton usage unique
Contenu malicieux	Insertion images / codes	Portail	Politique Sécurité Contenu
Brute force	Cassage MdP	Portail	Délai reconnexion
Inclusion	iFrame	Site malicieux	X-Frame-Option = DENY
Déni de service	Submerger de requêtes	Serveurs applicatifs	Pare-feu + ReverseProxies + Sondes type IPS
Contournement + Surcharge entêtes	Viser directement les applications	SSO en ReverseProxy	Pare-feu + Règles de filtrage adresses IP sources

Suite à l'implémentation de la 2FA, j'ai été amené à ajouter ou à modifier des tests unitaires pour les adapter aux nouvelles fonctionnalités. Les tests d'enrôlement ou de validation des clefs U2F sont basés sur le module Perl (« Authen::U2F::Tester - FIDO/U2F Authentication Test Client », s. d.) qui permet de simuler un équipement compatible U2F. La fonctionnalité TOTP est testée en enrôlant un code TOTP puis en soumettant un formulaire HTTP de connexion contenant le code. Enfin, les connexions avec second facteur REST ou Externe sont testées grâce à un client PSGI, un serveur PSGI veillant deux URL « verify » et « init » ou deux scripts *Perl* simulant la plateforme externe et une connexion avec un compte de test.

En plus des tests de non régression, LLNG dispose d'un moteur de rejeu basé sur « Protractor » et le framework « Jasmine ». (« Protractor - end-to-end testing for AngularJS », s. d.) est un moteur permettant de tester une interface utilisateur fonctionnant sous *AngularJS*. *Protractor* exécute les tests en utilisant un vrai navigateur, ici « Chrome », pour interagir avec l'application comme le ferait un utilisateur réel. En revanche, seul le *Manager* peut être testé complètement par ce moteur de rejeu. Tout comme la partie *Handler*, le *Portail* est testé sommairement car écrit en *Perl* ou basé sur le framework *jQuery*. (« Jasmine (JavaScript testing framework) », 2018) est utilisé pour tester les fonctionnalités JavaScript. Il permet une syntaxe plus simple et fournit des outils pour vérifier facilement le retour des fonctions tels la valeur, le type et un ensemble de fonctionnalités de tests.

3.5 – Sécurité : Scénarios d'attaques & Contre-mesures

La sécurité a également été améliorée dans cette version 2.0. La contre-partie est une complexité supplémentaire à comprendre et à prendre en compte lors des phases de conception et de développement.

3.5.1 – Content Security Policy ou CSP

Dans son fonctionnement par défaut, LLNG délivre des cookies de session contenant juste un « Id » et sans date de validité. La session est gérée par le *Portail*. Ces cookies ne sont valables que pour le domaine du SSO d'où l'implémentation d'un mécanisme de *CDA*.

En plus de restreindre le domaine de validité des cookies ou les domaines de confiance vers lesquels l'utilisateur peut être redirigé depuis le *Portail* après authentification, la version 2.0 de LLNG implémente une Politique de Sécurité du Contenu (CSP). Il s'agit d'un mécanisme de sécurité permettant de restreindre l'origine du contenu dans une page web (tel qu'un script *Javascript*, une feuille de style, etc...) à certains sites autorisés. Cela permet de mieux se prémunir d'une éventuelle faille XSS.

Le site envoie la liste des URL ou sources autorisées sous forme de liste de noms de domaine via l'en-tête HTTP « *Content-Security-Policy* ». Les navigateurs ne supportant pas cette spécification ignorent simplement l'en-tête, cela est donc transparent pour le visiteur.

Page index.php :

```
<html>

   <form method="post" action="login.php">
      <input type="texte" name="pseudo" />
      <input type="submit" value="Connexion" />
   </form>

</html>
```

Page login.php :

```
<?php

   echo "Bonjour ".$_POST['pseudo']." !"

?>
```

* Premier test :

Saisi de la chaîne → Christophe

Résultat → Bonjour **Christophe** !

* Deuxième test :

Saisi de la chaîne → <script>alert('Il y a une faille XSS')</script>

Résultat →

En plus de limiter les sources des images, des scripts ou encore les URL autorisées à transmettre des formulaires, cette *CSP* interdit d'embarquer du code *JavaScript* directement dans les pages *HTML*. A plusieurs reprises, j'ai dû modifier leurs structures notamment lors de la mise en place de la colorisation du lien correspondant au module actif dans le bandeau du *Manager*. Les différents paramètres de la CSP sont modifiables via le *Manager* sauf la directive « form-action » qui est générée dynamiquement.

3.5.2 – Cross-Site Request Forgery ou CSRF / XSRF

L'objet de cette attaque est de transmettre à un utilisateur authentifié une requête HTTP falsifiée qui pointe sur une action interne au site, afin qu'il l'exécute sans en avoir conscience et en utilisant ses propres droits. Par exemple, la requête peut contenir une pseudo-image. L'URL de l'image est un lien vers le script malicieux. L'utilisateur devient donc complice d'une attaque sans même s'en rendre compte. L'attaque étant actionnée par l'utilisateur, un grand nombre de systèmes d'authentification, notamment basés sur les sessions, sont contournés.

Le principe de protection est de s'assurer que l'application requiert une authentification qui n'est pas utilisée automatiquement par le navigateur. L'ajout du jeton permet de s'assurer que la requête ne sera considérée comme légitime que si elle contient cet élément aléatoire. En effet, les attaques *CSRF* ne sont possibles que si les requêtes sont prédictibles.

Tous les formulaires HTTP sont sécurisés par l'emport et la vérification d'un jeton de validité pour se prémunir de ces attaques. Celui-ci est généré aléatoirement par le serveur puis passé dans un attribut caché du formulaire lors de son envoi. Ensuite, il est vérifié par le serveur après soumission pour éviter toute interception et réutilisation. L'attaque *CSRF* est dirigée contre les visiteurs d'un site et non contre le site lui-même. Les moyens de protection pour les développeurs servent donc à protéger leurs utilisateurs.

3.5.3 – Injection de code ou XSS (Cross-Site Scripting)

En revanche, il est important de noter que cette protection contre les attaque *CSRF* ne fonctionne pas si le site est vulnérable aux attaques par injections de code *JavaScript* ou *XSS*. En effet, il est alors possible au site attaquant d'accéder aux données renvoyées par le site vulnérable notamment les cookies ou les captchas.

L'attaque *XSS* consiste à injecter du contenu dans une page, provoquant ainsi des actions sur les navigateurs web visitant la page. Les possibilités des *XSS* sont très larges puisque l'attaquant peut utiliser tous les langages pris en charge par le navigateur (*JavaScript, Java, Flash…*). Il est par exemple possible de rediriger vers un autre site ou encore de voler la session en récupérant les cookies. Voir l'exemple de la page ci-contre.

Pour cette raison, lors du processus d'authentification, le *Portail* vérifie le formatage des URL demandées et recherche la présence de caractères interdits. De plus, toutes les valeurs saisies dans les champs des formulaires HTTP sont échappées et testées avec des expressions régulières pour interdire tous les caractères spéciaux. Les scripts externes et leurs paramètres sont passés pour exécution via des objets de type tableau.

En outre, pour se prémunir de l'injection de code *SQL* consistant à utiliser une zone de saisie ou un paramètre pour exécuter du code arbitraire au travers de l'applicatif, toutes les requêtes *SQL* sont réalisées en deux étapes : préparation puis exécution. L'étape de préparation permet d'échapper automatiquement toutes les variables et de vérifier la validité de la requête. Ensuite, celle-ci est interprétée lors de l'étape d'exécution.

S'agissant du *Manager*, toutes les macros ou règles ainsi que les entêtes sont évalués dans une cage sécurisée. Il s'agit d'un espace d'exécution ou container à l'intérieur duquel les fonctions utilisables doivent être explicitement déclarée afin d'isoler ou confiner les applications et limiter les risques d'injection.

3.5.4 – Interception ou Man-in-the-middle

Comme précisé et imposé par la norme, l'usage des clefs U2F est soumis à la mise en place d'une connexion HTTPS entre le client et le serveur. Pour les tests effectués localement sur les postes de développement, l'utilisation d'un certificat auto-signé est suffisant. S'agissant de la plateforme internet *FusionIAM*, nous avons choisi d'utiliser un certificat fourni et signé par l'autorité de certification (AC) (« Lets Encrypt », 2018). Cette autorité fournit des certificats gratuits au format X.509 (.pem) pour le protocole cryptographique TLS au moyen d'un processus automatisé destiné à se passer du processus complexe actuel impliquant la création manuelle, la validation, la signature, l'installation et le renouvellement des certificats pour la sécurisation des sites internet.

Le protocole HTTPS permet d'authentifier le serveur et le client pour se prémunir de l'intrusion d'un tiers pendant les échanges. Le protocole HTTPS permet également de se prémunir du vol des cookies de session.

3.5.5 – Force brute

L'attaque par force brute est une méthode utilisée pour trouver un mot de passe ou une clef. Il s'agit de tester, une à une, toutes les combinaisons possibles. Cette attaque permet de casser tout mot de passe en un temps fini indépendamment de la protection utilisée, mais le temps augmente avec la longueur du mot de passe. Cette méthode est souvent combinée avec l'attaque par dictionnaire (liste de mots de passe les plus courants) pour trouver le secret plus rapidement.

Pour protéger le *Portail* de ce type d'attaque, j'ai implémenté le module « Plugins ::BruteForceProtection ». Celui-ci consiste à imposer un temps d'attente avant une nouvelle tentative d'authentification après plusieurs échecs consécutifs dans un laps de temps restreint.

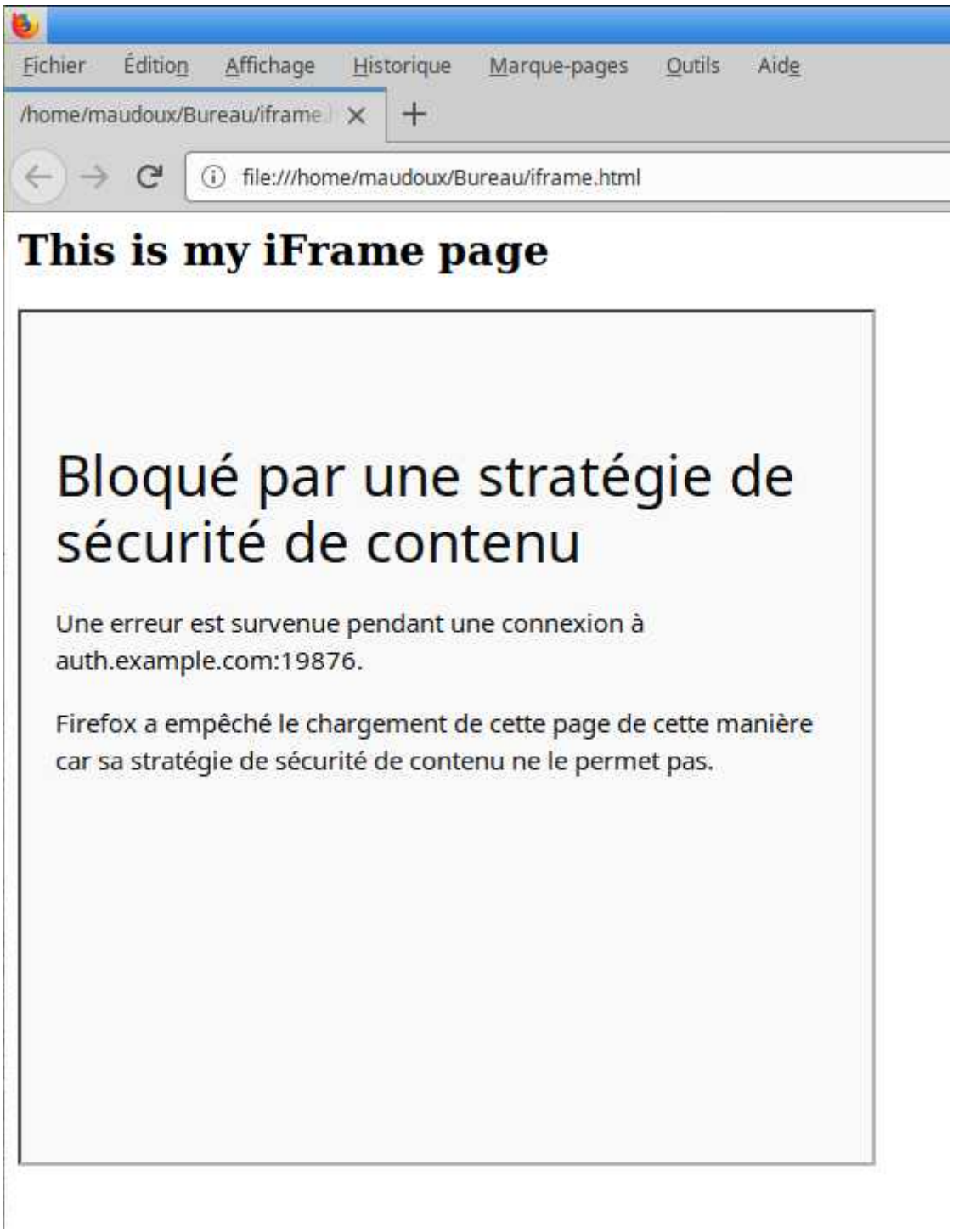

Figure 19 : Protection anti-iFrame

Une autre solution est la mise en œuvre sur les serveurs web de l'application *Fail2Ban* qui permet d'interdire un adresse IP après plusieurs échecs de connexion par la mise en place de règles *ipTable*. Fail2Ban permet de surveiller plusieurs services (ssh, HTTP, FTP, etc...).

3.5.6 – iFrame invisible

(« iFRAME, fonctionnement et protection – CERT-FR », s. d.) Contrairement à la balise *frame* utilisée pour diviser une page HTML en différentes pages organisées de manière logique et toutes stockées sur le même serveur, la balise *iFrame* permet d'afficher au sein d'une même page des informations stockées sur des serveurs différents.

Introduite en 1997 par *IE*, la balise *iFrame* signifie « inline frame » (cadre en ligne). Un *iFrame* est un document HTML intégré à un autre. Il peut être utilisé pour insérer des bandeaux publicitaires hébergés sur des serveurs dédiés ou afficher du contenu provenant d'un domaine externe comme des images, des vidéos intégrées ou des élément décoratifs.

L'utilisation malveillante des *iFrames* peut être classée en deux catégories. Soit le pirate compromet un site légitime en y insérant des *iFrames* rendus invisibles en réduisant leurs tailles (width=0 height=0 frameborder=0) ou en bloquant leur affichage. Soit il insère un lien vers le site légitime dans un *iFrame* de mêmes dimensions. Dans ce cas, l'internaute croit visiter le site légitime sans remarquer que celui-ci est contenu dans un *iFrame* inclus dans une page mère.

La connexion de l'utilisateur au serveur hébergeant le contenu de l'*iFrame* s'effectuant à son insu, il est facile pour un individu malveillant d'exploiter cette propriété afin de compromettre sa victime. L'objectif de l'utilisation de la balise *iFrame* par une personne malintentionnée est bien souvent la propagation de codes malveillants ou intercepter les données saisies. En effet, les *iFrames* permettent de contourner le cloisonnement d'exécution des codes comme JavaScript. Le visiteur, se rendant sur la page d'un site a priori de « confiance », établit alors, à son insu, une connexion vers un site et télécharge un code malveillant. Ce code, pour s'exécuter, exploite des vulnérabilités du navigateur et s'installe sur la machine de la victime.

La protection mise en œuvre dans la version 2.0 de LLNG permet d'interdire l'affichage du *Portail* si celui-ci est inclus dans un *iFrame* (cf. figure 19). Celle-ci est basée sur l'ajout de l'entête HTTP « X-FRAME-OPTIONS = deny » dans la CSP.

3.5.7 – Déni de service ou DoS / Distributed DoS

(« Attaque par déni de service », 2018) Une attaque par déni de service a pour but de rendre indisponible un service, d'empêcher les utilisateurs légitimes d'un service de l'utiliser. La grande majorité de ces attaques se fait à partir de plusieurs sources (BotNet ou réseaux Zombies). On parle alors d'attaque par déni de service distribuée ou DdoS. Il peut s'agir de submerger un réseau afin d'empêcher son fonctionnement, de perturber les connexions entre deux machines pour empêcher l'accès à un service particulier ou d'envoyer énormément de requêtes à un serveur pour le faire tomber. L'attaque par déni de service peut ainsi bloquer un serveur de fichiers, rendre impossible l'accès à un serveur web ou empêcher la distribution de courriel dans une entreprise.

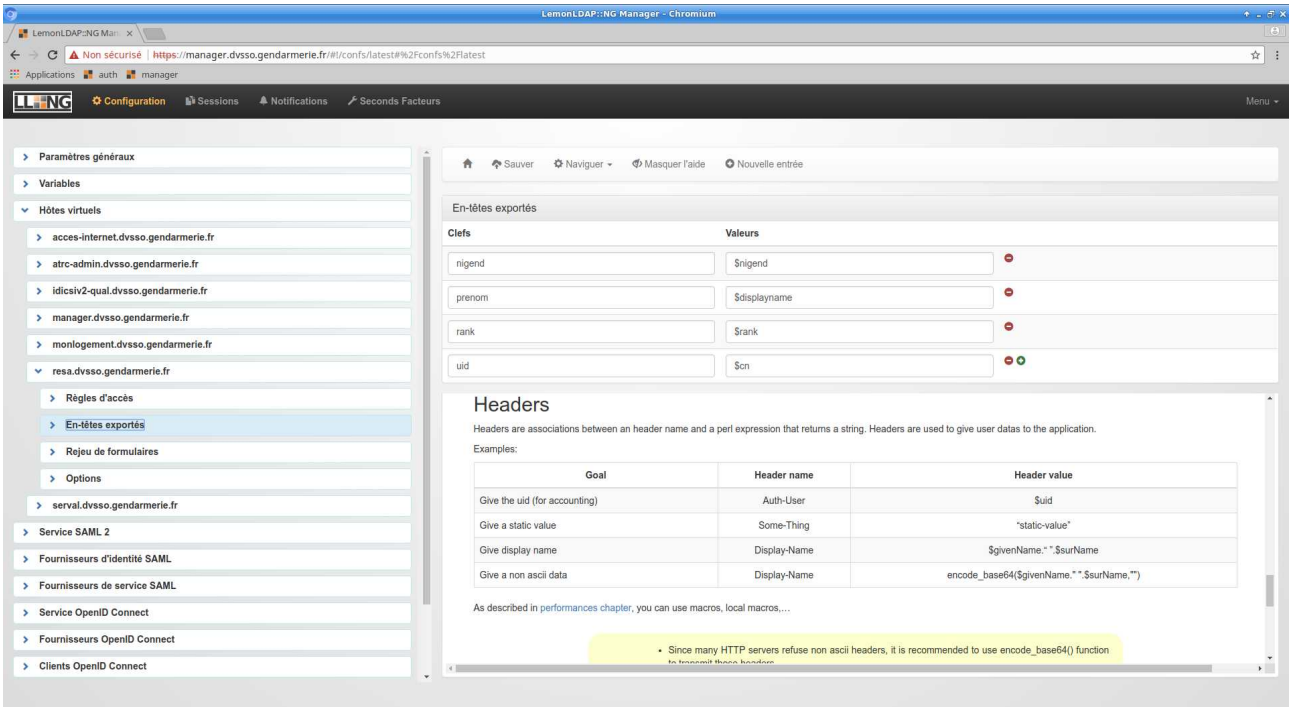

Figure 20 : Configuration des entêtes dans le *Manager*

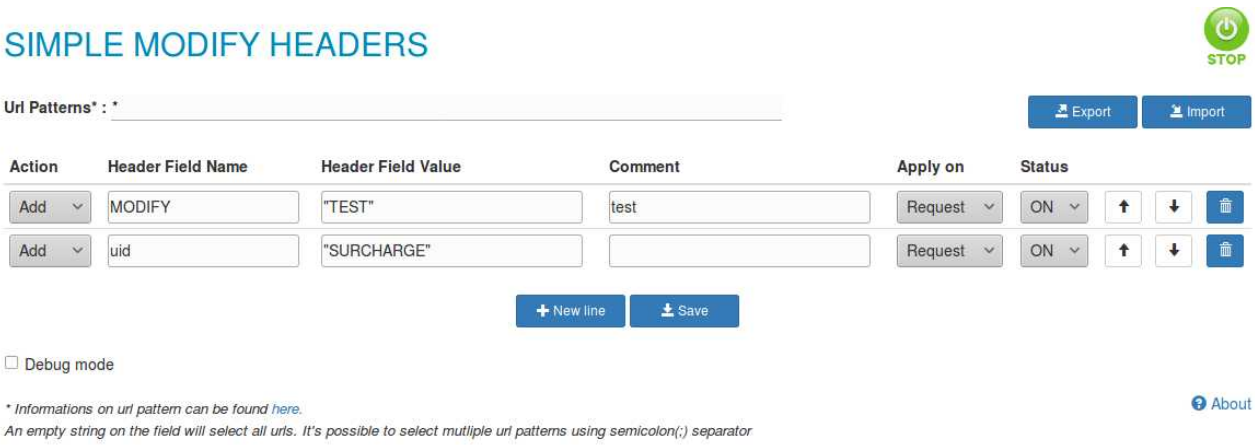

Figure 21 : Entêtes surchargés avec l'extension « ModifyHeaders »

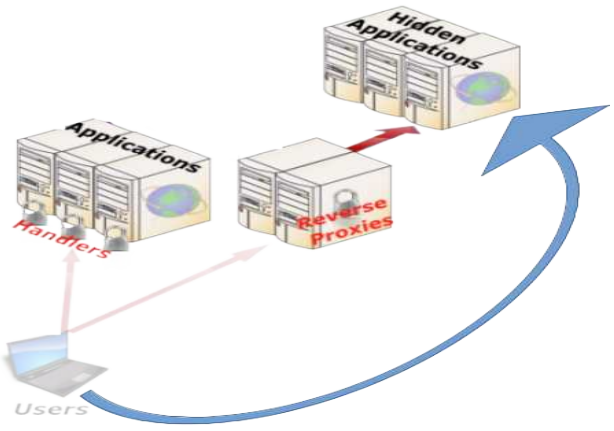

Figure 21-1 : Attaque par contournement

L'attaquant n'a pas forcément besoin de matériel sophistiqué. Ainsi, certaines attaques DoS peuvent être exécutées avec des ressources limitées contre un réseau de taille plus importante et plus moderne. On appelle parfois ce type d'attaque « attaque asymétrique » en raison de la différence de ressources entre les protagonistes.

Les seules parades existantes pour se prémunir de ce type d'attaques sont la mise en place de pare-feu « intelligents » capables d'identifier des trames illégitimes ou de bloquer des adresses IP. Il est également possible de disposer en frontal des serveurs web un pool de ReverseProxies Nginx permettant d'absorber la charge et de limiter le nombre de requêtes par seconde.

Cette protection doit donc être assurée par l'architecture réseau mise en place et non LLNG.

3.5.8 – Contournement SSO & API

Ce type d'attaque par contournement du SSO est possible uniquement contre des applications protégées en mode *ReverseProxy*. Celles-ci sont « cachées » derrière un pool de *ReverseProxies* embarquant les *Handlers*.

Le principe ici est de contacter directement les serveurs applicatifs sans passer par le pool de mandataires inverses(cf figure 21-1). Les requêtes n'étant pas interceptées par les *Handlers*, aucun contrôle d'accès n'est effectué. Ensuite, le pirate peut usurper une identité en forgeant lui-même ses propres entêtes HTTP en utilisant par exemple l'extension pour FireFox « ModifyHeaders ».

Dans ce cas, la contre-mesure à mettre en place est de configurer les serveurs applicatifs afin qu'ils n'acceptent que les connexions en provenance des *ReverseProxies* SSO grâce à des règles de filtrage sur les adresses IP dans les serveurs web (directives 'deny - allow') ou par l'utilisation d'équipements actifs comme des « BigIP F5 » ou des pare-feux.

L'utilisation seule de l'extension FireFox « ModifyHeaders » n'est pas suffisante car les *Handlers* écrasent les éventuels entêtes HTTP homonymes (cf. figures 20 à 23).

En ce qui concerne les services Web API ou REST, il est préférable de mettre en place un *Portail* dédié à ces services.

3.5.9 – Failles logicielles

L'attaque ici consiste à mettre en œuvre des exploits. Un exploit est un élément de programme permettant à un individu ou à un logiciel malveillant d'exploiter une faille de sécurité dans un système informatique. Que ce soit à distance ou sur la machine sur laquelle cet exploit est exécuté, le but de cette manœuvre est de s'emparer des ressources d'un ordinateur ou d'un réseau, d'accroître le privilège d'un logiciel ou d'un utilisateur sur la machine-cible, ou encore d'effectuer une attaque par déni de service.

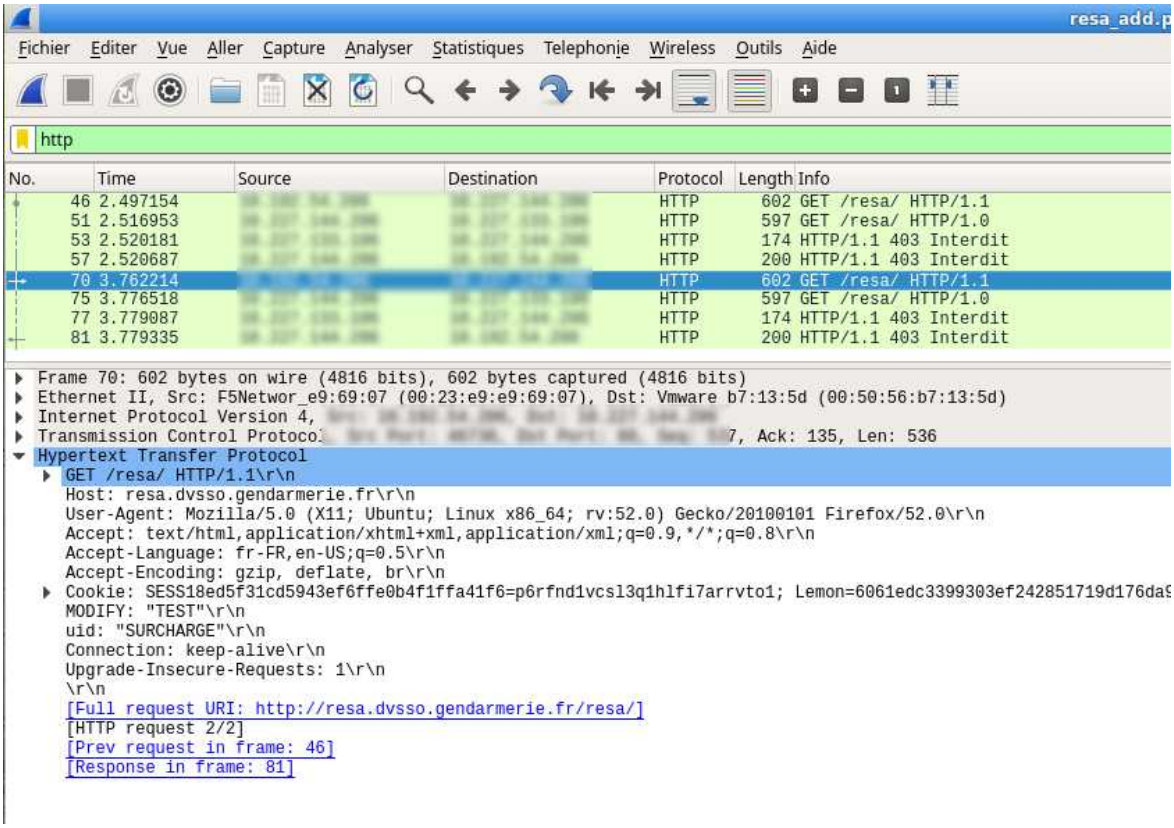

Figure 22 : Entêtes transmis par le navigateur avec « ModifyHeaders »

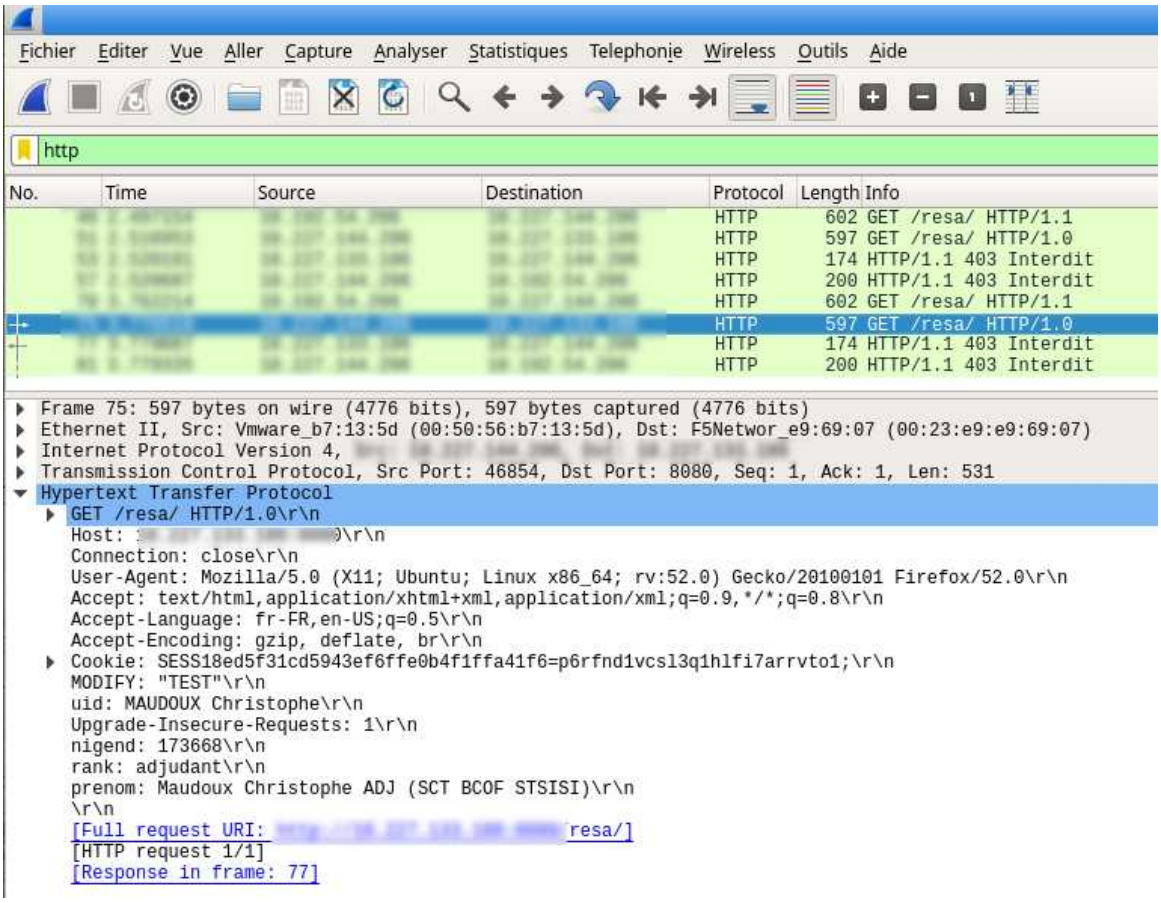

Figure 23 : Entêtes transmis par le *ReverseProxy*

Ces failles sont bien souvent révélées par les éditeurs de logiciels eux-mêmes lors de la publication des correctifs de sécurité. En effet, ceux-ci sont analysés et les pirates s'attaquent aux logiciels non corrigés. Les systèmes d'exploitation et les logiciels doivent être mis à jour régulièrement. Pour ce faire, nous utilisons *bower* pour faciliter la montée en version des librairies logicielles *JavaScript* et ce de préférence en version LTS (Long Term Support) comme c'est le cas pour *AngularJS*, actuellement en version 1.7 LTS.

En revanche, ces montées en version ont parfois eu des « effets de bord » et ont nécessité de modifier le code comme, par exemple, la syntaxe des URL pour *AngularJS* (/#/), la fonction « encode64 », la création des cookies ou encore le « datepiker » *JavaScript* du module *Notifications*.

4 – Implémentation initiale

4.1 – Historique du développement

J'ai commencé à travailler sur la version 2.0 du projet LemonLDAP::NG début février et sollicité ma première fusion de branche mi février 2018. J'ai choisi de commencer par implémenter les clefs U2F, le TOTP et les clefs Yubikey.

Ensuite, régulièrement entre mars et juin, j'ai soumis une vingtaine de demandes de fusions dont un peu moins de la moitié pour le *Portail*. Début mars, j'ai ajouté la possibilité pour l'utilisateur de supprimer son second facteur et travaillé essentiellement sur le *Manager*. La première version du module permettant de gérer les sessions persistantes avec second facteur (module *2ndFA*) est créée. Mi-mars, j'ai ajouté la fonctionnalité de suppression du second facteur d'un utilisateur depuis le *Manager* via des boutons spécifiques à chaque type de second facteur. Fin mars, j'ai réussi à intégrer au module *2ndFA* le moteur de recherche des sessions persistantes par « uid ». Puis, j'ai ajouté le filtre d'affichage sur les différents types de seconds facteurs et supprimé un niveau de recherche dans l'explorateur.

Début avril, après discussion avec l'équipe de développement, j'ai modifié le module *Session* existant afin de masquer les attributs secrets associés aux seconds facteurs et modifier leur présentation. Puis, courant avril, nous avons créé avec Xavier Guimard le moteur de gestion des seconds facteurs afin de ne charger et présenter les modules correspondants uniquement s'ils sont activés et de ne proposer à l'authentification que les seconds facteurs enrôlés. Fin avril, j'ai modifié la structure de sauvegarde des seconds facteurs en session persistante pour permettre l'enrôlement de plusieurs équipements U2F et créé le gestionnaire permettant aux utilisateurs d'administrer leurs différents seconds facteurs.

En mai, j'ai ajouté au menu de *Portail* un lien permettant d'accéder à la page de gestion des seconds facteurs uniquement si l'un d'entre eux est activé. J'ai également apporté des modifications pour limiter le nombre maximum de seconds facteurs qu'il est possible d'enregistrer, la longueur du nom associé, pour permettre à l'administrateur d'afficher ou non le TOTP existant d'un utilisateur, autoriser ou non la suppression ou l'enrôlement par l'utilisateur d'un second facteur. En effet, ceux-ci peuvent être créés et enregistrés en sessions persistantes par un processus extérieur comme un ETL ou gérés par une autre application d'où l'implémentation des modules « REST » et « Ext2F »

Ma dernière demande de fusion a été ouverte le 7 juin puis j'ai continué à apporter régulièrement des modifications ou améliorations directement sur la branche « master ». J'ai ajouté la fonctionnalité de présentation de l'historique des connexions après une authentification avec double facteur et créé ou modifié les tests de non régression associés.

En plus de ces tests unitaires, j'ai ajouté des tests lors de la sauvegarde de la configuration pour éviter soit que le *Manager* devienne inaccessible suite à un mauvais paramétrage soit d'activer des options contradictoires ou nécessitant des modules *Perl* optionnels absents ce qui entraînerait une erreur interne du serveur.

Portail et *Manager* sont deux éléments qui peuvent être vus comme deux applications distinctes. En effet, ils ont été conçus avec deux années d'écart et les technologies utilisées sont différentes. N'ayant pas eu à modifier la partie *Handler*, je ne l'aborderai pas dans ce mémoire.

La réalisation de ce mémoire a également été pour moi l'occasion de prendre en compte et traiter une quinzaine de tickets concernant des problèmes ou des demandes d'évolution ouvertes par des utilisateurs sur « GitLab ».

4.2 – Portail

4.2.1 – Structure & Technologies utilisées

Tout d'abord, lors de la procédure d'authentification, nous pouvons distinguer pour un utilisateur trois états différents : *non-authentifié*, *authentifié* et *autorisé*. Lorsqu'un utilisateur se connecte au *Portail* sans présenter de cookie SSO, il est dans l'état *non-authentifié*. Il est alors invité à saisir son couple Identifiant / Mot de Passe. S'il est reconnu par le *Portail*, il passe dans un état transitoire dit *authentifié* et ses données de session persistante sont collectées. Le *Portail* applique les conditions d'ouverture de session, vérifie si une notification doit être validée puis demande le second facteur si celui-ci est activé ou requis. Une fois le second facteur validé, le *Portail* affiche l'historique de connexion s'il a été demandé ou redirige l'utilisateur vers le menu. L'utilisateur est alors dans l'état *autorisé*, c'est-à-dire que sa session temporaire SSO a été créée par le *Portail*.

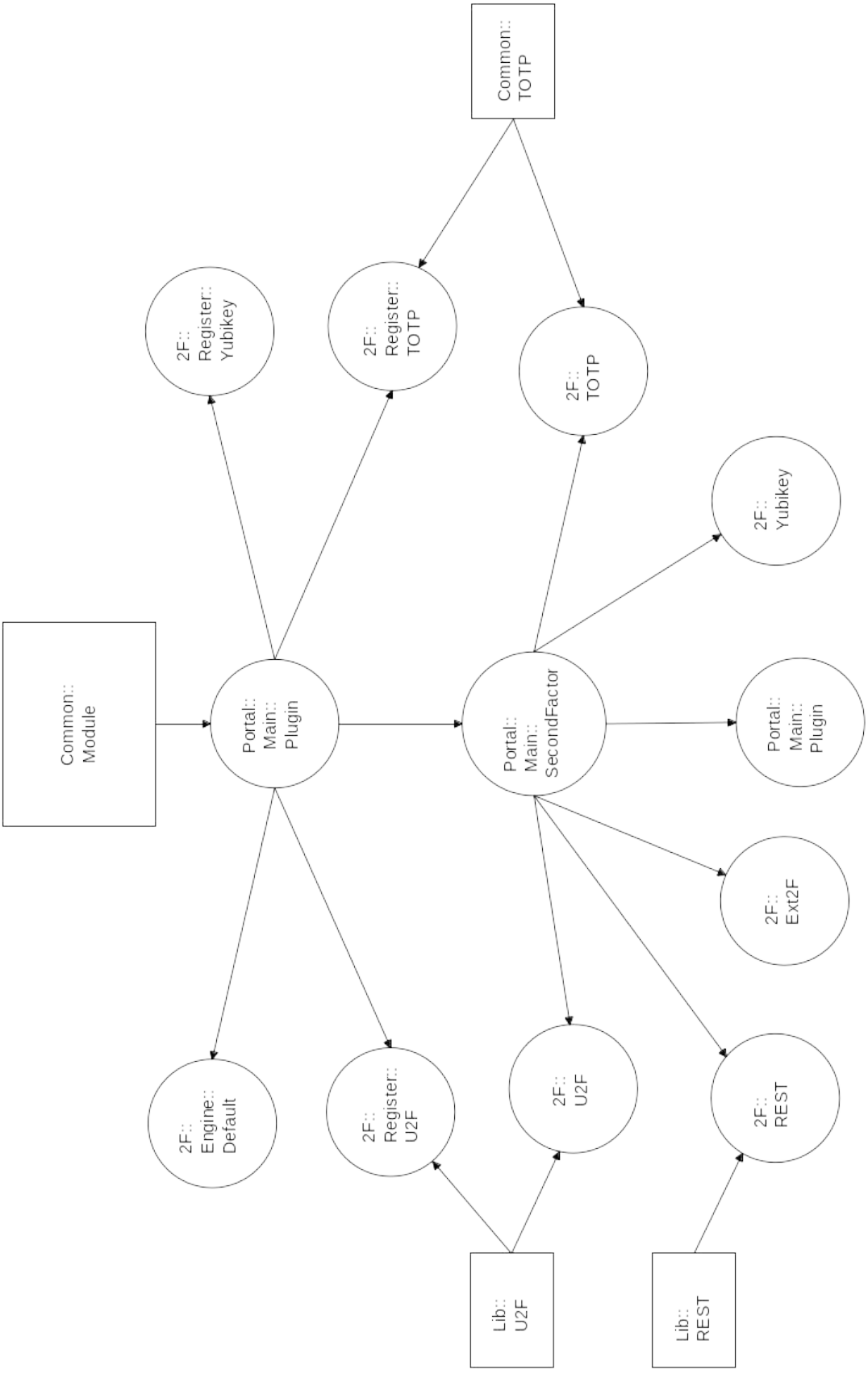

Figure 24 : Graphe d’héritage des modules 2FA

Ensuite, à compter de la version 2.0 de LLNG, la structure des différents modules Perl est modulaire, hiérarchique et basée sur le principe d'héritage issu de la programmation orientée objet (cf. figure 24). Celui-ci est mis en œuvre et simplifié par l'utilisation du module (« Mouse - Moose minus the antlers », s. d.). La lecture de cet article (Feed 32up, s. d.) m'a été très utile pour en comprendre la logique et la syntaxe. Pour chaque module sont définies des constantes accessibles en lecture, écriture ou les deux qui pourraient être assimilées aux propriétés en langage objet. Celles-ci fournissent des accesseurs qui permettent de lire ou initialiser les « propriétés ». Nous trouvons également des fonctions qui pourraient être vues comme des méthodes. Tous les modules du *Portail* héritent de l'objet « p ». Cette structure a pour avantage qu'ils disposent tous de leur propre espace de nommage et de leur propre copie de la configuration. Pour chaque requête HTTP interceptée, le *Handler* initialise un objet « req » grâce auquel il est possible d'accéder entre-autres au corps de la requête, aux paramètres ou aux données de session persistante ou temporaire d'un utilisateur *authentifié*, objet « userData ».

J'ai créé cinq modules correspondants aux différents types de seconds facteurs implémentés dans LLNG permettant d'authentifier les utilisateurs à savoir : U2F, TOTP, Yubikey, REST et Ext2F. Pour les seconds facteurs qui peuvent être enrôlés par l'utilisateur : U2F, TOTP et Yubikey, nous avons créé les trois modules correspondants sous le répertoire «/Register». Tous ces modules héritent des fonctions de « Portal::Main::SecondFactor » ou d'autres modules spécifiques grâce à la directive « extend » de *Mouse*.

Pour chacun des cinq modules de contrôle d'accès, nous trouvons les trois fonctions obligatoires suivantes : « init » , « run » et « verify ». « init » est exécutée au chargement du module pour vérifier si le second facteur correspondant est activé et autorise ou non l'auto-enrôlement. Si oui, je vérifie si l'utilisateur a enregistré un équipement du même type. La fonction « run » est appelée pour vérifier si l'utilisateur est authentifié, récupérer ses données de session persistante et initialiser les variables. « verify » est chargée de vérifier et valider le second facteur présenté. Pour les trois modules d'auto-enrôlement, seules « init » et « run » sont nécessaires. « init » permet de s'assurer que le module a bien été chargé en retournant la valeur « 1 » et « run » expose les différentes fonctions nécessaires à l'enrôlement.

Par défaut, les pages HTML sont mises en forme en utilisant le framework *Bootstrap (*« Bootstrap (framework) », 2018) et des (« Feuilles de style en cascade (CSS) », 2018) que j'ai également été amené à modifier. Les deux principaux avantages de *Bootstrap* sont sa facilité à être personnalisé en téléchargeant des thèmes et qu'il adapte dynamiquement la mise en page (principe de la grille) en fonction du terminal ou de la résolution utilisée (*responsive*). En outre, il gère très mal les tableaux HTML ce qui pose des problèmes avec les terminaux mobiles.

Nombre de fonctions JavaScript utilisées pour créer le *Portail* sont fournies par la librairie *Jquery* (« jQuery », 2018). N'ayant que les connaissances de bases en JavaScript, après trois semaines de lecture de l'ouvrage (Allain, 2011) et du guide (« AJAX », s. d.) suivi d'un rapide apprentissage de *CoffeeScript*, j'ai pu mettre en œuvre l'architecture AJAX (Asynchronous JavaScript And XML***)*** (« Ajax (informatique) », 2018) nécessaire aux fonctions d'enrôlement et de suppression des seconds facteurs ainsi qu'à la modification du DOM (Document Objet Model) (« Document Object Model », 2018) pour la présentation des dates. En effet, LLNG étant une application web modulaire, *Portail*, *Manager* et clients peuvent être installés et répartis sur des serveurs ou fuseaux horaires différents. Par conséquent, toutes les dates sont générées côté serveur au format *epoch-Posix* puis affichées par le navigateur dans le format de langue et au fuseau horaire définis par l'utilisateur.

Les différentes pages HTML sont construites dynamiquement par les modules Perl en chargeant des *templates* auxquels sont passés des paramètres au format JSON convertis en chaîne de caractères. D'une part, ces différents *templates* importent les structures d'entête « header.tpl » et de pied de page « footer.tpl ». « header.tpl » est important ici car il nous permet de charger les fonctions « getValues » et « translatePage » incluses dans *Portal.js*. D'autre part, ils sont constitués de structures conditionnelles ou de boucles, de formulaires ou de balises HTML classiques et de balises spécifiques comme « trspan » et « application/init ». Les balises « trspan » sont recherchées par la fonction « translatePage » pour gérer les traductions de l'interface et les balises « application/init » sont traitées par « getValues » pour initialiser les variables utilisées par les fonctions JavaScript. Ces deux fonctions utilisent la méthode *Jquery* « each » pour rechercher dans le code HTML les balises spécifiques. Les dix fichiers de langue dont le français, l'arabe, l'italien ou le vietnamien sont situés dans le répertoire « static/languages ». Si une traduction n'est pas disponible, c'est l'anglais qui est utilisé par défaut.

Pour terminer cette présentation de la structure du *Portail* et des principes de fonctionnement de LLNG nécessaires à la réalisation de mes travaux, j'aborderai les notions de routes authentifiées et non-authentifiées. A leur chargement, ces *templates* importent les codes *JavaScript* minifiés, générés à partir des fichiers *CoffeeScript*. Les formulaires HTML et les codes *JavaScript* permettent d'exécuter des appels de fonctions pour répondre à des événements comme la soumission de formulaires HTTP ou les clics sur des boutons. Pour déterminer la fonction qui doit être exécutée lors de l'appel, nous utilisons les notions d'URL et de méthodes HTTP. Lors de l'initialisation des modules, nous déclarons des routes qui peuvent être accessibles à des utilisateurs non-authentifiés « addUnauthRoute » ou réservées uniquement à des utilisateurs authentifiés « addAuthRoute ».

L'ensemble de ce travail de découverte, d'apprentissage, de questionnement à propos de LLNG, de compréhension de l'ensemble de la structure et des technologies utilisées représente, cumulé, environ quatre mois d'investissement personnel. Ceci était très frustrant au début car il me fallait dix jours pour pouvoir écrire une ligne de code ! De surcroît, je n'ai pas encore abordé tous les concepts notamment toute la partie *Handler* ou analyse de la configuration (*parser*).

4.2.2 – Modules Main::SecondFactor & Lib::U2F

* « Lib::U2F » vérifie la présence, charge et teste la librairie U2F. Ce module hérite de « Crypt::U2F::Server::Simple ».

* « Main::SecondFactor » hérite de «Lib::OneTimeToken » et expose les fonctions suivantes :

- Sa fonction « init » initialise les routes non-authentifiées nécessaires à l'appel de la fonction « verify » de chacun des modules :

```
$self→addUnauthRoute( $self->prefix . '2fcheck' => '_verify', ['POST'] );
```

- « _verify » contrôle la présence et la validité du jeton anti-CSRF et appelle la fonction « verify » du module correspondant à l'URL.

4.2.3 – Modules U2F & Register::U2F

L'authentification avec périphériques U2F s'articule principalement autour de deux modules : « 2F::U2F » pour le contrôle d'accès et « 2F::Register::U2F » pour gérer l'enrôlement. « 2F::U2F.pm » fait appel au *template* « u2fcheck.tpl » et au *JavaScript* généré à partir de « u2fcheck.coffee » et « 2F::Register::U2F » utilise « u2fregister.tpl » et « u2fregistration.js ». Ces deux modules héritent de « Lib::U2F » et « Main::SecondFactor » ou « Main::Plugin »

```
extends 'Lemonldap::NG::Portal::Main::SecondFactor',
'Lemonldap::NG::Portal::Lib::U2F';
```

Le module « Main ::SecondFactor » permet également la gestion du jeton anti-CSRF. Lors de la phase d'enrôlement, l'utilisateur a été authentifié et autorisé par le *Portail*. Le *Handler* a donc initialisé l'objet *userData* permettant d'accéder aux données de session. Pour sécurisé la phase d'authentification, lorsque l'utilisateur est dans l'état *authentifié,* les données de session persistante sont collectées grâce à l'*Id* passé dans le jeton anti-CSRF. c'est la fonction « getToken », héritée de «Lib::OneTimeToken », qui instancie l'objet *$session* utilisé par les fonctions « verify » des modules de contrôle d'accès.

* Le fonctionnement de « 2F::U2F » est le suivant :

- La fonction « init » vérifie s'il est activé ou si la fonction d'auto-enrôlement est autorisée. S'il n'est pas activé, le module n'est pas chargé et la 2FA avec équipement U2F n'est pas disponible. S'il est simplement activé, le second facteur est chargé depuis la session persistante et requis à la connexion. Dans ce cas, le second facteur est provisionné en session persistante par un

moyen extérieur. S'il est activé avec une règle spécifique (requis pour un utilisateur en particulier ou pour une plage d'adresses IP spécifiques par exemple), la règle est appliquée. S'il est activé avec la fonction d'auto-enrôlement, la règle d'activation est modifiée afin que l'équipement U2F ne soit requis par le *Portail* que si un équipement U2F a été précédemment enregistré, donc si une entrée U2F existe dans la session persistante :

```
if (  $self→conf→{u2fSelfRegistration} and $self->conf->{u2fActivation} eq '1' )
      {   $self->conf->{u2fActivation} = '$_u2fSecret =~ /\w+/';   } ;
return 0   unless ( $self->Lemonldap::NG::Portal::Main::SecondFactor::init()
      and $self->Lemonldap::NG::Portal::Lib::U2F::init() );
```

La dernière étape d'« init » est l'appel des fonctions « init » des modules parents « Lib::U2F » pour l'héritage des fonctions spécifiques à l'U2F et « Main::SecondFactor » pour l'initialisation des routes :

```
$self->addUnauthRoute( $self->prefix . '2fcheck' => '_verify',  ['POST']  );
$self->addUnauthRoute(  $self->prefix . '2fcheck' => '_redirect', ['GET'] );
```

- La fonction « run » permet de vérifier si l'utilisateur est authentifié, de collecter ses données persistantes dont le *secret* et le *KeyHandle* correspondants à l'équipement U2F enregistré et de générer un *challenge* qui sera utilisé lors la phase d'authentification. Ces étapes sont réalisées grâce aux fonctions fournies par le module « Lib::U2F » qui hérite de « Crypt::U2F::Server::Simple ». La dernière étape de « run » est l'appel via la fonction « sendHTML » du *template* « u2fcheck.tpl » avec passage du jeton anti-CSRF et des paramètres au script *JavaScript* dans le corps de la requête.

- La fonction « verify » contrôle si le client a bien retourné le *challenge* ainsi que le *challenge signé* et vérifie la signature grâce à la clef publique. Les différentes erreurs possibles sont l'absence de *challenge* ou de *signature* dans la réponse due à un équipement défectueux, que le *challenge* retourné par le client ne corresponde pas au *challenge* transmis par le serveur (modification par le client ou problème réseau) ou que la signature ne soit pas reconnue (équipement non enrôlé ou incompatible).

* En ce qui concerne « 2F::Register::U2F » :

- « init » charge les routes authentifiées via la fonction « init » parente et retourne la valeur « 1 » si le module a bien été chargé.

```
return 0 unless $self→SUPER::init;  return 1;
```

- La fonction « run » permet de gérer les deux actions nécessaires à l'enrôlement à savoir « register, registration». Lors de l'appel de l'action « register » par « u2fregistration.js » grâce à la route authentifiée "#{portal}2fregisters/u/register", un *challenge* est retourné puis soumis par le navigateur à l'équipement U2F. Après que la signature ai été autorisée par une action de l'utilisateur, « u2fregistration.js » envoie au module serveur « 2F::Register ::U2F », les données générées par l'équipement U2F ainsi que le *challenge* d'origine via la route authentifiée "#{portal}2fregisters/u/registration". « registration » vérifie les données grâce à la fonction « registrationVerify » héritée de « Lib::U2F » et sauvegarde en session persistante le *KeyHandle* et la clef publique associée :

```
$self->p->updatePersistentSession( $req,
        { _u2fKH => to_json($_KH), _u2fSecret => to_json($_secret) } );
```

Les options nécessaires à renseigner dans le *Manager* sont donc l'activation du module ($u2fActivation), l'auto-enrôlement ($u2fSelfRegistration) et le niveau d'authentification atteint avec ce second facteur ($u2fAuthnLevel).

4.2.4 – Modules TOTP & Register::TOTP

J'ai utilisé la même structure pour le TOTP à savoir un module de contrôle d'accès « 2F::TOTP » et un module dédié à l'enrôlement « 2F::Register::TOTP » avec les *templates* et les *JavaScript* associés. Ceux-ci héritent de « Main::SecondFactor » ou de « Common::TOTP » :

```
extends 'Lemonldap::NG::Portal::Main::SecondFactor',
  'Lemonldap::NG::Common::TOTP';
```

* Comme pour les modules gérant l'U2F, les actions gérées par « 2F::TOTP » sont :

- la fonction « init » qui vérifie si le module est activé et si la fonction d'auto-enrôlement est autorisée, modifie la règle d'activation si besoin et appelle la fonction « init » du module parent pour l'initialisation des routes non-authentifiées :

```
if    ( $self→conf→{totp2fSelfRegistration}
        and $self->conf->{totp2fActivation} eq '1' )
{  $self->conf->{totp2fActivation} = '$_totpSecret =~ /\w+/'; }
        return $self→SUPER::init();
```

Il n'y a pas d'ambiguïté ici car « Common ::TOTP » n'expose pas de fonction « init ».

- « run » appelle le *template* « totp2fcheck.tpl » avec la fonction « sendHTML » et transmet le jeton anti-CSRF. Contrairement à l'authentification U2F, il n'y a pas de *challenge* ou autre paramètre à passer.

- « verify » contrôle la présence du code TOTP dans la réponse HTTP, récupère le secret précédemment enregistré par l'utilisateur en session persistante via l'objet *$session* créé grâce à l'Id de session transmis avec le jeton anti-CSRF. Avec le secret, « verify » calcule le TOTP en utilisant la fonction « verifyCode » héritée de « Common::TOTP » et compare avec le TOTP saisi par l'utilisateur. Si le TOTP calculé correspond au TOTP saisi par l'utilisateur, celui-ci est *autorisé*.

* Le modules Register/TOTP.pm est constitué comme suit :

- « init » charge les routes authentifiées et retourne la valeur « 1 » si le module a bien été chargé.

- La fonction « run » permet de gérer les deux actions nécessaires à l'enrôlement à savoir « verify, getKey ». Lors de l'appel de l'action « getkey » par « totp2fregistration.js » via la route authentifiée "#{portal}2fregisters/totp/getkey », une clef symétrique est calculée par la fonction « newSecret » héritée de « Common::TOTP ». La fonction « getkey » est exécutée soit au chargement de la page soit sur demande de l'utilisateur en cliquant sur le bouton permettant d'obtenir une nouvelle clef secrète. Celle-ci est retournée avec d'autres paramètres (nombre de chiffres du TOTP, période de validité, URL par défaut du portail ou l'URL précisé dans la configuration et le nom de l'utilisateur) à la libraire *Qrious* pour génération puis affichage au format chaîne de caractères de la clef secrète et du *QRCode* correspondant. Le secret est transmis dans le corps de la requête et conservé également grâce à un jeton anti-CSRF.

Le contrôle du TOTP lors de la phase d'enrôlement est effectué par l'action « verify », appelée par « totp2fregistration.js » lors du clic sur le bouton de validation via la route authentifiée "#{portal}2fregisters/totp/verify". « verify » est utilisée ici pour s'assurer que l'utilisateur dispose d'une application TOTP valide. « verify » récupère le TOTP saisi par l'utilisateur dans le corps de la requête et le secret de l'utilisateur correspondant au jeton anti-CSRF. Le TOTP est calculé en faisant appel à la fonction « verifyCode » héritée de « Common::TOTP ». Le fait de récupérer le secret via

le jeton anti-CSRF permet de s'assurer que le secret utilisé est bien celui fourni par le serveur et non un secret re-généré côté client. Si le TOTP calculé correspond au TOTP saisi par l'utilisateur, le secret partagé est sauvegardé en session persistante :

```
$self->p->updatePersistentSession( $req,
                        { _totpSecret => to_json($_secret) } );
```

Les options nécessaires à renseigner dans le *Manager* sont donc l'activation du module ($totp2fActivation), l'activation de l'auto-enrôlement ($totp2fSelfRegistration), l'intervalle de temps ($interval), le nombre de chiffres du TOTP ($digits), le nombre d'intervalles à tester ($range) et le niveau d'authentification atteint avec ce second facteur ($totp2fAuthnLevel).

4.2.5 – Modules Yubikey & Register::Yubikey

Les modules « 2F::Yubikey » et « 2F::Regiter::Yubikey » héritent de « Main::SecondFactor » ou directement de « Main::Plugin » pour les fonctions d'initialisation des routes ou d'appel des *templates*.

* Le module « 2F::Yubikey » hérite uniquement de « Main::SecondFactor » et fonctionne suivant la structure suivante :

- Tout d'abord, « init » teste si le client Web Yubikey est installé :

```
eval { require Auth::Yubikey_WebClient };
```

puis vérifie si le module est activé et si la fonction d'auto-enrôlement est autorisée. Si oui, la règle d'activation est modifiée :

```
if ( $self->conf->{yubikey2fSelfRegistration}
   and $self->conf->{yubikey2fActivation} eq '1' )
{   $self->conf->{yubikey2fActivation} =   '$_yubikeySecret =~ /\w+'; }
```

Ensuite, « init » instancie un client Yubikey en passant les paramètres de configuration :

```
$self->yubi(   Auth::Yubikey_WebClient->new(
        {  id    => $self->conf->{yubikey2fClientID},
           api   => $self->conf->{yubikey2fSecretKey},
           nonce => $self->conf->{yubikey2fNonce},
           url   => $self→conf→{yubikey2fUrl}   } ) ;
```

et appelle la fonction « init » parente pour l'initialisation des routes : $self → SUPER::init();

- « run » vérifie si l'utilisateur a enrôlé une Yubikey et appelle le *template* « yubikey2fcheck.tpl » avec la fonction « sendHTML ». Comme pour l'authentification par TOTP, seul le jeton anti-CSRF est transmis.

- « verify » récupère la partie publique de la clef en session persistante grâce à l'Id transmis via le jeton anti-CSRF et l'OTP envoyé comme paramètre par le formulaire dans le corps de la requête par la Yubikey. Ensuite, « verify » interroge le serveur Yubico en utilisant une instance du client Web : $self → yubi → otp($code).

* Le module « 2F::Register::Yubikey » hérite directement de « Main::Plugin » et est construit avec :

- « init » qui appelle le *template* « yubikey2fRegister.tpl » et retourne la valeur « 1 ».

- « run » expose l'action « register » qui est appelée lors de la soumission du formulaire HTTP (form action="/2fregisters/yubikey/register"). « register » récupère, via le formulaire, l'OTP généré par la *Yubikey*, conserve uniquement la partie publique de celui-ci (12 chiffres par défaut) :

```
my $key = substr( $otp, 0, $self→conf→{yubikey2fPublicIDSize}
```

vérifie que cette clef publique n'ai pas déjà été enrôlée et la sauvegarde en session persistante.

Les options nécessaires à renseigner dans le *Manager* sont donc l'activation du module ($yubikey2fActivation), l'activation de l'auto-enrôlement ($yubikey2fSelfRegistration), l'URL du serveur à interroger ($yubikey2fUrl), l'identifiant du client Web ($yubikey2fClientID), le secret du client ($yubikey2fSecretKey), l'éventuelle clef de salage ($yubikey2fNonce), la partie publique de l'OTP ($_yubikeySecret), la longueur de la partie publique de l'OTP (yubikey2fPublicIDSize) et le niveau d'authentification atteint avec ce second facteur ($yubikey2fAuthnLevel).

4.2.6 – Modules 2F::REST & 2F::Ext2F

Les deux modules « 2F::REST » et « 2F::Ext2F » ne permettent pas l'auto-enrôlement et héritent également de « Main::SecondFactor » ou « lib::REST ». Ils permettent de déléguer l'authentification à double facteur à un service extérieur à LLNG comme une plateforme de SMS ou un service d'OTP propriétaire par exemple.

* « 2F::REST » expose les trois fonctions obligatoires suivantes :

- « init » contrôle si l'URL du service REST de vérification est renseignée puis teste tous les arguments d'initialisation et de vérification afin de s'assurer qu'il n'y ait pas de caractères spéciaux pour se prémunir des injections de code : `$attr =~ /^\w+$/`. Ensuite, la fonction « init » parente est appelée.

- « run » appelle l'URL d'initialisation si elle a été renseignée avec les arguments associés en utilisant la méthode « RestCall » héritée de « lib::REST », teste la réponse et charge le *template* « rest2fcheck.tpl ».

- « verify » appelle l'URL de vérification avec ses arguments, teste le résultat et redirige l'utilisateur *authorisé* vers le portail.

Les options nécessaires à renseigner dans le *Manager* sont l'activation du module ($rest2fActivation), l'URL d'initialisation ($rest2fInitUrl) et ses arguments ($rest2fInitArgs), l'URL de vérification ($rest2fVerifyUrl) et ses arguments ($rest2fVerifyArgs) et le niveau d'authentification atteint avec ce second facteur ($rest2fAuthnLevel).

* Ext2F.pm expose également :

- « init » contrôle la présence des commandes « Send » et « Validate » et appelle la fonction « init » parente.

- « run » exécute la commande « Send » grâce à la fonction « launch », vérifie le code retour et appelle le *template* « ext2fcheck.tpl » via la fonction « sendHTML ». « send » permet de soumettre le second facteur.

- « verify » exécute la commande « Validate », teste le code retour et redirige l'utilisateur *authorisé* vers le portail. « verify » permet de valider le second facteur renseigné par l'utilisateur.

- La fonction « launch » est la suivante :

```
my ( $self, $session, $command, $code ) = @_;  my @args;
foreach ( split( /\s+/, $command ) ) {
if ( defined $code ) {    s#\$code\b#$code#g;      }
s#\$(\w+)#$session->{$1} // ''#ge;  (global et eval, // → defined-or-op)
push @args, $_;  } ;    return system @args;
```

Par exemple :

```
/usr/local/bin/sendSMS –uid $uid
/usr/local/bin/verifySMS –uid $uid –code $code
```

A la différence de l'opérateur | | (OU logique) qui teste si la variable est vraie, l'opérateur / / teste si la variable est définie : `EXPR1 // EXPR2 <=> defined(EXPR1) ? EXPR1 : EXPR2`

« launch » place chaque élément de la commande et ses arguments dans un tableau pour éviter les injections de code et exécute un appel système (« system - perldoc.perl.org », s. d.) en passant le tableau. Le premier élément est la commande, les autres éléments étant considérés comme ses arguments. A la différence de « exec », « system » exécute la commande dans un processus fils (« Lancer des processus à partir de Perl », s. d.).

Les options nécessaires à renseigner dans le *Manager* sont l'activation du module ($ext2fActivation), la commande de soumission ($ext2FSendCommand), la commande de validation, vérification ($ext2FValidateCommand) et le niveau d'authentification atteint avec ce second facteur ($ext2fAuthnLevel).

4.2.7 – Tests de non-régression

Pour chaque fonctionnalité implémentée dans LLNG, il existe un ou plusieurs tests de non-régression. Ceux-ci permettent de s'assurer que toute nouvelle modification n'altère pas le fonctionnement actuel ou qu'une erreur corrigée ne puisse pas réapparaître. De plus, ces tests offrent une image du comportement d'un code et peuvent permettre de mesurer l'avancement du travail. L'ensemble de ces tests est situé dans le répertoire : « lemon-ldap-ng-portal/t ».

Tous ces tests de non-régression utilisent, entre autres, les modules Perl (« Test::More - perldoc.perl.org », s. d.) qui est une extension du module de base « Test::Simple » et (« Test::Harness - Run Perl standard test scripts with statistics », s. d.). Ceux-ci sont basés sur le protocole TAP (« Test Anything Protocol », 2018). La lecture de cet article (« Les tests en Perl - Présentation et modules standards », s. d.) m'a permis d'en comprendre l'écriture.

Les tests suivent la structure suivante : un plan décrivant le nombre de tests à réaliser s'il est prédictible et les tests à réaliser qui peuvent être : OK pour tester une valeur booléenne, IS / ISNT pour vérifier que deux valeurs sont respectivement égales ou différentes, ISA_OK permet de vérifier si un objet ou une référence est du type escompté, etc … Le résultat du test est présenté sous la forme : état (OK ou NOT_OK), le numéro du test (compteur), la description du test et une directive (TODO ou SKIP pour passer le test). Tous ces tests sont pilotés par le module « Test::Harness » qui permet de recueillir le maximum d'informations, de les analyser et les interpréter.

Le principal intérêt du protocole TAP est la séparation entre le programme de test proprement dit, qui génère la suite de « ok / not_ok », et le programme d'analyse de la sortie du premier, l'interpréteur TAP. En effet, il suffirait de stocker les informations dans un objet et d'afficher le résultat à la fin de l'exécution du test ; mais que se passe-t-il si le programme de test plante et meurt ? Rien, aucune sortie n'est générée puisque le code en charge de l'affichage des résultats a été purgé avec le reste du processus. C'est là que la séparation entre le générateur et l'interpréteur TAP devient très intéressante : comme c'est l'interpréteur qui lance l'exécution du générateur, il n'est pas affecté si ce dernier meurt et peut ainsi détecter que le générateur a été quitté inopinément. Il peut même indiquer quel test est à l'origine de l'arrêt, simplifiant ainsi le travail de recherche du code fautif. Autre avantage, l'interpréteur peut préparer l'environnement du programme de test afin de positionner certaines variables et vérifier la présence des bibliothèques ou modules nécessaires. Enfin, générateur et interpréteur peuvent être écrits dans des langages différents.

Suite à l'implémentation de la 2FA, les cinq tests suivants : U2F.t, TOTP.t, TOTP_8.t, REST.t et Ext.t ont été ajoutés. Le module « Yubibey » n'est pas testé car il n'existe pas de client permettant de simuler la présence d'une *Yubikey.* De plus, du fait de la mise à jour d'un compteur interne d'utilisation, il est impossible de réutiliser un OTP Yubikey.

* Le test « U2F.t » est constitué de 17 tests unitaires et ses principales étapes sont :

- Vérification de la présence des dépendances « U2 ::Server » et « U2F ::Tester ».

- Modification de la configuration pour activer la 2FA avec clef U2F et l'auto-enrôlement :

```
my $client = LLNG::Manager::Test→new( {
    ini => { logLevel   => 'error',
    u2fSelfRegistration => 1,
    u2fActivation       => 1  }  }  );
```

- Tests de connexion au *Portail* avec le compte de test 'dwho', récupération du cookie SSO, demande de la page d'enrôlement U2F (GET '2fregisters') et redirection vers la page '/2fregisters/u' (recherche dans l'URL grâce à une expression régulière) et vérification de la page retournée par le serveur (recherche dans le corps de la requête du script 'u2fregistration.min.js' chargé en fin de page) :

```
my $id = expectCookie($res);

expectRedirection( $res, qr#/2fregisters/u$# );

ok( $res->[2]->[0] =~ /u2fregistration\.(?:min\.)?js/, 'Found U2F js' );
```

- Simulation de la demande de *challenge*, envoi du *challenge* par le serveur, réception du *challenge* par le client et création d'une connexion HTTPS par le module de test :

```
Register challenge: { "challenge":
"6fUQ6WasRvlcPA7V3UF2ORWOMpfczwU3VvzqICVpZbw", "version": "U2F_V2",
"appId": "https:\/\/auth.example.com:19876" }
```

- Création et envoi du KH et des données d'enrôlement par le testeur au serveur et sauvegarde du KH et de la clef publique associée en session persistante.

```
my $r = $tester->register( $data->{appId}, $data->{challenge} );

ok( $r->is_success, ' Good challenge value' ) or diag( $r->error_message );

my $registrationData = JSON::to_json(    {
        clientData      => $r->client_data,
        errorCode       => 0,
        registrationData => $r->registration_data,
        version         => "U2F_V2"       }    );
```

```
Get registration data
{"registrationData":"BQR4IKisaeRFlfYgz.....","challenge":"6fUQ6WasRvlcPA7
V3UF2ORWOMpfczwU3VvzqICVpZbw","version":"U2F_V2","clientData":"eyJ0e
XAiOiJuYXZpZ2F0b3IuaWQuZmluaXNoRW5yb2xsbWVudCIsImNoYWxsZW5nZSI6
IjZmVVE2V2FzUnZsY1BBN1YzVUYyT1JXT01wZmN6d1UzVnZ6cUlDVnBaYnciLCJv
cmlnaW4iOiJodHRwczovL2F1dGguZXhhbXBsZS5jb206MTk4NzYiLCJjaWRfcHVia2
V5IjoidW51c2VkIn0"}
```

clientData contient le challenge d'origine, l'URL du serveur et le type de requête encodés en base64 :

```
echo
'eyJ0eXAiOiJuYXZpZ2F0b3IuaWQuZmluaXNoRW5yb2xsbWVudCIsImNoYWxsZW
5nZSI6IjZmVVE2V2FzUnZsY1BBN1YzVUYyT1JXT01wZmN6d1UzVnZ6cUlDVnBaY
nciLCJvcmlnaW4iOiJodHRwczovL2F1dGguZXhhbXBsZS5jb206MTk4NzYiLCJjaWRf
cHVia2V5IjoidW51c2VkIn0' | base64 -d
```

retourne :

```
{"typ":"navigator.id.finishEnrollment","challenge":"6fUQ6WasRvlcPA7V3UF2OR
WOMpfczwU3VvzqICVpZbw","origin":"https://
auth.example.com:19876","cid_pubkey":"unused"}
```

- Déconnexion et reconnexion au *Portail* avec le compte 'dwho', envoi du challenge et du KH par le *Portail* puis retour du *challenge* et du *challenge* signé par le testeur U2F :

```
$client->logout($id);
$r =  $tester->sign( $data->{appId}, $data->{challenge},
$data->{registeredKeys}->[0]->{keyHandle} );
```

- Vérification de la présence du cookie SSO pour s'assurer que l'utilisateur est bien *autorisé*.

* Les tests « TOTP.t » et « TOTP_8.t » sont constitués de 16 et 18 tests unitaires et leurs principales étapes sont :

- Vérification des dépendances « Convert::Base32 » et « Authen::OATH » pour le test avec TOTP à 8 chiffres et modification de la configuration de base pour activer la 2FA avec TOTP à 8 ou 6 chiffres par défaut et l'auto-enrôlement :

```
my $client = LLNG::Manager::Test->new( {
   ini                        => {  logLevel   => 'error',
   totp2fSelfRegistration => 1,
   totp2fActivation        => 1,
   totp2fDigits              => 8,  (optionnel)
 }    }    );
```

- Authentification avec le compte de test 'dwho', appel de la page '2fregisters' et test de la redirection vers '/2fregisters/totp' par recherche du script « totpregistration.min.js » :

```
ok( $res->[2]->[0] =~ /totpregistration\.(?:min\.)?js/, 'Found TOTP js' );
```

- Demande de génération et envoi en base32 par le serveur d'un secret partagé par l'appel de l'URL '/2fregisters/totp/getkey', décodage du secret et vérification de la présence du jeton anti-CSRF :

```
ok( $key  = $res->{secret}, 'Found secret' );
ok( $token = $res->{token},  'Found token' );
$key = Convert::Base32::decode_base32($key);
```

- Calcule du TOTP constitué de 6 ou 8 chiffres, vérification du nombre de chiffres puis envoi au serveur du TOTP et du jeton par appel de l'URL '/2fregisters/totp/verify' :

```
ok( $code = Lemonldap::NG::Common::TOTP::_code( undef, $key, 0, 30,
6 ou 8 ), 'Code' );
ok( $code =~ /^\d{6 ou 8}$/, 'Code contains 6 or 8 digits' );
my $s = "code=$code&token=$token";
```

- Calcul par le serveur du TOTP correspondant au secret puis sauvegarde du secret en session persistante.

- Déconnexion et reconnexion avec le compte de test 'dwho', vérification de l'envoi par le serveur du formulaire de connexion avec le TOTP et le jeton anti-CSRF, calcul du TOTP correspondant au secret généré précédemment par la fonction interne « _code » pour les deux tests. De plus, je calcule également le TOTP en utilisant une application extérieure pour le test du TOTP à 8 chiffres. J'utilise, pour valider la formule interne de LLNG, la fonction fournie par le module « Authen::OATH » puis soumission du formulaire et vérification de la présence du cookie SSO :

```
$client→logout($id);
expectForm( $res, undef, '/totp2fcheck', 'token' );
ok( $totp = Lemonldap::NG::Common::TOTP::_code( undef, $key, 0, 30,
6 ou 8 ), 'Code' );
```

```
my $oath = Authen::OATH->new( digits => 8 );
ok( $code = $oath->totp( $key ),   'Ext. App Code' );
ok( $code == $totp, 'Both TOTP match')
    or explain( [ $code, $totp ], 'LLNG and Ext. App TOTP mismatch');
$client->_post('/totp2fcheck', IO::String→new($query), length =>
length($query),)
$id = expectCookie($res);
```

J'ai rajouté le test du calcul du TOTP avec une application externe suite à la découverte d'une erreur dans l'algorithme initial de la fonction « _code ».

* Le test « REST.t » est constitué de 4 tests unitaires et ses principales étapes sont :

- Modification de la configuration pour activer le module REST et déclaration des URL d'initialisation et de vérification avec leurs arguments respectifs :

```
my $client = LLNG::Manager::Test->new(    {
ini => {  logLevel            => 'error',  rest2fActivation => 1,
          rest2fInitUrl       => 'http://auth.example.com/init',
          rest2fInitArgs      => { name => 'uid' },
          rest2fVerifyUrl     => 'http://auth.example.com/vrfy',
          rest2fVerifyArgs  => { code => 'code' }     }    });
```

- Création d'un serveur REST qui répond aux deux URL 'init' et 'vrfy'. Celui-ci attend respectivement et successivement comme contenu l'identifiant 'dwho' et le code '1234' puis redirige l'utilisateur autorisé vers le *Portail*.

```
if ( $req->path_info eq '/init' ) { ok( $req->content eq '{"name":"dwho"}')}
  elsif ( $req->path_info eq '/vrfy' ) { ok( $req->content eq '{"code":"1234"}')
```

- Authentification avec le compte de test 'dwho', test de l'envoi par le serveur du formulaire '/rest2fcheck' contenant le jeton anti-CSRF et le champ 'code', renseignement du code dans le corps de la requête avec '1234' puis soumission du formulaire :

```
expectForm( $res, undef, '/rest2fcheck', 'token', 'code' );
$query =~ s/code=/code=1234/;
$res = $client->_post( '/rest2fcheck', IO::String→new($query))
```

- Appel de l'URL 'vrfy' par le serveur avec passage du code dans le paramètre 'code', réception par le service REST, contrôle du code et réponse du service REST. Vérification par le test de la présence du cookie SSO.

```
my $id = expectCookie($res) ;
```

* Pour pouvoir tester le module « Ext2F », j'ai simulé le service extérieur à l'aide des deux scripts 'sendOTP.pl' et 'vrfyOTP.pl'. Ceux-ci retournent simplement la valeur '0' si l'identifiant correspond à 'dwho' et le code à '123456' ou '1' dans le cas contraire. Exemple pour 'vrfyOTP.pl' :

```
Exit   ! ( $swt1 eq '-uid' && $user eq 'dwho' && $swt2 eq '-code'
          && $code eq '123456' )
```

Le test « Ext.t » est constitué de 3 tests unitaires et ses principales étapes sont :

- Modification de la configuration de base pour activer la 2FA extérieure, déclaration des commandes d'envoi et de vérification avec leurs paramètres respectifs :

```
my $client = LLNG::Manager::Test->new(    {
      ini => {  logLevel    => 'error',  ext2fActivation => 1,
      ext2FSendCommand  => 't/sendOTP.pl -uid -$uid',
       ext2FValidateCommand   =>  't/vrfyOTP.pl -uid -$uid -code $code' } } );
```

- Authentification avec le compte de test 'dwho', test de l'envoi par le serveur du formulaire '/ext2fcheck' contenant le jeton anti-CSRF et le champ 'code', vérification de la présence de la balise 'input' pour le saisie du code, renseignement du code dans le corps de la requête avec '123456' puis soumission du formulaire :

```
expectForm( $res, undef, '/ext2fcheck', 'token', 'code' );
ok( $res→[2]→[0] =~ qr%<input name='code' value='' class='form-control'
id='extcode'>%)
$query =~ s/code=/code=123456/;
$res = $client->_post( '/ext2fcheck', IO::String→new($query))
```

- Exécution de la commande 'ValidateCommand' par le serveur avec passage de l'uid et du code dans le paramètre '$code' puis vérification par le test de la présence du cookie SSO dans la réponse.

```
my $id = expectCookie($res) ;
```

Les méthodes spécifiques comme « expectForm, expectRedirection, expectCookie, explain » sont fournies par la librairie « test-lib.pm » qui utilise le module (« Data::Dumper - perldoc.perl.org », s. d.) pour afficher le contenu des variables Perl.

* Les fichiers de langues et les fichiers .POD font également l'objet d'un test de non-régression spécifique. Le test des langues consiste à compter le nombre de lignes contenu dans chacun d'entre-eux et à vérifier s'il est identique à celui de référence qui est l'anglais. Le test des fichiers '.POD' utilise le module (« Test::Pod - check for POD errors in files - metacpan.org », s. d.) pour vérifier la structure de chacun d'entre-eux qui doit respecter un formalisme particulier (balises HEAD, ITEM, etc...).

Le test des fichiers de langues est constitué comme suit : ouverture du répertoire et chargement de tous les fichiers '.JSON' excepté l'anglais, vérification de la structure, tri des entrées, comptage et comparaison avec le fichier anglais de référence. Ensuite, l'ensemble des *templates* est analysé pour rechercher les balises « trspan » et vérifier qu'une entrée correspondante à chaque balise existe dans les fichiers de langues.

4.2.8 – Synthèse

Après environ dix semaines cumulées de travail collaboratif, les différents modules de contrôle d'accès et d'enrôlement ainsi que les tests de non-régression correspondants sont réalisés.

A la fin de cette première phase, l'ensemble de l'authentification avec double facteur U2F, TOTP, Yubikey ou externes est fonctionnelle côté *Portail*. Les utilisateurs peuvent enrôler leurs seconds facteurs et celui-ci est requis si le module correspondant est activé. Il reste encore à réaliser toute l'implémentation côté *Manager.*

4.3 – Manager

4.3.1 – Structure & Technologies utilisées

La principale caractéristique du *Manager* est qu’il s’agit d’une SPA (« Application web monopage », 2018) C’est une application web accessible via une page web unique. Le but est d'éviter le chargement d'une nouvelle page pour chaque action demandée et de fluidifier ainsi l'expérience utilisateur. Deux méthodes existent pour implémenter ce mécanisme. Soit l'ensemble des éléments de l'application est chargé (contenu, images, CSS et JavaScript) dans un unique fichier HTML, soit les ressources nécessaires sont récupérées et affichées dynamiquement en fonction des actions de l'utilisateur. Le terme a été introduit par Steve Yen en 2005.

L'enregistrement en local de la page définissant une application web monopage et la possibilité de continuer à l'exécuter localement est l'une des propriétés importantes des SPA qui les distingue des applications web standards qui reposent sur l'existence d'un serveur HTTP avec lequel elles échangent données, continuations applicatives et interfaces.

Quand les SPA gèrent des données et permettent de les modifier, pour conserver ces données modifiées, il est nécessaire que ces applications modifient leur code, c'est-à-dire : elles doivent être capable de se modifier pour que la sauvegarde locale de l'état modifié de l'application (dont les données) soit persistant.

Pour mettre en œuvre cette structure, le *Manager* est réalisé avec le framework *AngularJS* développé par Google (« AngularJS », 2018). Une page web conçue avec *AngularJS* suit le patron d'architecture MVC (« Modèle-vue-contrôleur », 2018). Ce patron a pour avantage de proposer un couplage faible entre la présentation, les données, et les composants métier. Dans un langage web, cette séparation permet de diminuer l'importance des manipulations DOM et d'améliorer la testabilité du code. Dans *AngularJS*, la partie "vue" est déclarée dans une version étendue du HTML traditionnel, qui comporte de nouvelles balises et attributs. Ce HTML étendu est utilisé pour déclarer une liaison de données bidirectionnelle entre les modèles et les vues. Les données sont synchronisées automatiquement et moins d'éléments ont besoin d'être définis en *Javascript*. Les modèles sont composés de plusieurs couches appelées « scopes ». Les contrôleurs dans *AngularJS* permettent de prototyper des actions en code *JavaScript*. Les modèles sont les classes assurant la gestion des données, les vues correspondent à la manière d'afficher les informations à l'utilisateur et les contrôleurs réagissent aux actions des utilisateurs, ils vont chercher les données et les mettent à disposition des vues.

Le page du *Manager* est constituée des trois parties principales suivantes : le menu en haut de page listant les modules actifs et le choix des langues de l’interface, la partie gauche constituant l’arbre de navigation dans les différentes options de la configuration, enfin, la partie centrale utilisée pour l’affichage de l’aide contextuelle et le détail des options (cf. figure 9). La présentation sous forme d’arbre utilise la librairie « angular-ui-tree.js » et le CSS associé.

4.3.2 – Approche de développement

La structure particulière du *Manager* m'a imposé une méthode de développement que j'ai qualifiée de « circulaire ». Chaque module Perl constituant le menu est situé dans le répertoire 'manager'. Chaque module fait appel à un *template* situé dans 'site/templates' et un fichier *CoffeeScript* dans 'site/coffee'. Ces trois éléments interagissent entre-eux et nécessitent donc d'être développés en parallèle.

Début mars, pour créer le module explorateur de sessions avec second(s) facteur(s) qui deviendra par la suite le gestionnaire des sessions avec 2FA nommé '2ndFA', j'ai commencé par dupliquer les trois éléments relatifs à l'explorateur de session « classique » à savoir « ::sessions » et déclarer le module « ::2ndFA » dans le fichier 'lemonlap-ng.ini'. Celui-ci est chargé au démarrage de LLNG et permet d'initialiser les paramètres de base obligatoires : répertoire des *templates*, mode de protection, type et emplacement des bases de données, langues disponibles et les modules à charger qui sont par défaut « conf, sessions, notifications ». 'lemonlap-ng.ini' peut également être utilisé pour surcharger la configuration paramétrée via le *Manager* ou configurer LLNG sans passer par le *Manager* ce qui peut-être utile pour déployer par scripts.

Ensuite, par étape, j'ai supprimé tout le code dont je n'avais pas besoin pour obtenir les « squelettes » de mes trois fichiers relatifs au module *2ndFA*. Le plus difficile pour développer côté *Manager* est la maîtrise d'*AngularJS* et de *CoffeeScript*. J'ai commencé par modifier l'explorateur pour ne présenter que les sessions avec second(s) facteur(s) avec un niveau de recherche et un seul niveau d'affichage par identifiant, paramètre « whatToTrace ». En effet, les utilisateurs ne pouvant avoir qu'une seule session persistante, l'arbre de l'explorateur 2ndFA ne comporte que deux niveaux contrairement à l'explorateur de sessions classique qui en compte trois. Cette modification apparemment simple m'a demandé une quinzaine de jours afin de comprendre le fonctionnement ! Puis, j'ai ajouté le filtre cumulatif par type de 2FA et terminé par le moteur de recherche par identifiant de session. Au fur et à mesure des modifications, et ce régulièrement durant deux mois, je me suis documenté sur ce *framework* notamment en lisant les ouvrages (Ollivier & Gury, 2015) et (Panda, 2014) ainsi que l'article (« AngularJS — Superheroic JavaScript MVW Framework », s. d.).

Enfin, pour les tests et la résolution des erreurs, j'ai beaucoup utilisé la console d'erreurs de *Chrome*, le module *FireBug* pour *FireFox* et les journaux systèmes de LLNG. De plus, une fois les fichiers assemblés, les différents modules sont exécutés dans le répertoire « blib ». Celui-ci a la même structure que « lib » et permet de modifier le code à la « volée » et de tester sans avoir à tout reconstruire. En revanche, pour tester LLNG dans un autre environnement plus proche de la réalité fonctionnant sous *Nginx*, avec des bases de données externes et un annuaire LDAP, j'ai utilisé la plateforme de Clément Oudot, *FusionIAM*. Celle-ci étant en configuration de production, il me fallait à chaque fois, cloner la dernière version de LLNG, l'installer puis modifier et recharger les *templates* afin de ne pas utiliser les codes minifiés impossibles à analyser !

4.3.3 – Module Manager::2ndFA

Les différents modules du Manager héritent de «Common::Conf::AccessLib » permettant d'accéder à la configuration de LLNG (lecture des paramètres comme « whatToTrace » par exemple et « Common::Session::REST » pour échanger avec le *Portail* (lecture ou modification des sessions par exemple).

Les modules activés dans 'lemonlap-ng.ini' sont chargés par « ::*Manager »* qui appelle également la fonction *addRoutes* de chacun d'entre-eux. La *Manager* étant une application auto-protégée par héritage du *Handler*, toutes les routes déclarées sont implicitement des routes authentifiées, accessibles uniquement aux utilisateurs *autorisés*. En outre, la CSP autorise uniquement la soumission de formulaire émis avec l'URL du portail 'form-action = self' et l'utilisation des jetons anti-CSRF n'est pas nécessaire.

* Le module *2ndFA.pm* :

- permet d'initialiser avec la méthode 'GET' la route par défaut « {portail}/2ndfa.html » chargeant le *template* et la route « sfa/{ ':sessionType' } » appelant la fonction 'sfa' :

```
$self->addRoute( '2ndfa.html', undef, ['GET'] )
      ->addRoute(   sfa => { ':sessionType' => 'sfa' },   ['GET']   )
```

- expose la fonction 'sfa'. Celle-ci appelle la fonction 'session' héritée du serveur REST si un identifiant de session persistante ou temporaire est passé dans l'URL de la route :

```
if ($session) {    return $self->session( $req, $session, $skey );    }
```

Dans le cadre de ce module, les seconds facteurs étant sauvegardés en sessions persistantes, seul ce type de session sera appelé par *2ndFA.js* mais j'ai choisi de conserver la structure d'origine du module 'sessions'. Si la requête ne contient pas d'identifiant de session, le traitement par le module Perl se poursuit : recherche et tri des sessions persistantes par paramètre « whatToTrace » passé dans la variable « _session_uid » puis initialisation des paramètres du filtre par type de second facteur passés via la *QueryString* de l'URL par la fonction 'search2FA' fournie par *2ndFA.js* :

```
{portail}/manager/sfa/persistent?
_session_uid=*&U2FCheck=1&TOTPCheck=1&UBKCheck=1
foreach (qw(TOTP U2F UBK)) {
   $self->{ $_ . 'Check' } = delete $params->{ $_ . 'Check' }
    if ( defined $params->{ $_ . 'Check' } ); }
```

Puis, seules les sessions avec second(s) facteur(s) (filtre sessions 2FA) et correspondant au filtre par type sont conservées. L'implémentation des deux niveaux de filtrage est réalisée à l'aide de deux boucles 'foreach' recherchant en session persistante les clefs correspondantes à chaque type de second facteur (totpSecret, yubikeySecret et u2fSecret). La liste des sessions persistantes restantes est triée puis passée dans le corps de la requête sous forme de tableau :

```
$res = [
        sort { $a->{date} <=> $b->{date} }
        map { { session => $_, userId => $res->{$_}->{_session_uid} } }
        keys %$res  ];
return $self→sendJSONresponse(  $req,  {  result => 1,
        count  => scalar(@$res),
        total    => $total,  values => $res } );
```

* Concernant le fichier *2ndfa.tpl*, l'implémentation du filtre par type côté client est construit avec un formulaire HTTP constitué de balises « input » de type « checkbox ». Lors de la soumission de celui-ci, la fonction *JavaScript* 'search2FA()' est appelée :

```
<form name="filterForm">
<input type="checkbox" ng-model="U2FCheck" class="form-check-input"
ng-true-value="2" ng-false-value="1" ng-change="search2FA()"/>
<label class="form-check-label" for="U2FCheck">U2F</label>
<input type="checkbox" ng-model="TOTPCheck" class="form-check-input"
ng-true-value="2" ng-false-value="1" ng-change="search2FA()"/>
<label class="form-check-label" for="TOTPCheck">TOTP</label>
<input type="checkbox" ng-model="UBKCheck" class="form-check-input" ng-
true-value="2" ng-false-value="1" ng-change="search2FA()"/>
<label class="form-check-label" for="UBKCheck">UBK</label>  </form>
```

Le moteur de recherche est réalisé avec un formulaire et une balise « input » de type « text », baptisée 'searchString'. Il appelle également la fonction 'search2FA()' mais en passant le contenu du champ 'text' dans la variable « _session_uid » concaténé avec un méta-caractère 'glob' ou 'wildcard', ici '*' qui servira de filtre de recherche :

```
<form class="navbar-form" role="search">
<input class="form-control" placeholder="{{translate('search')}}" type="text"
ng-model="searchString" ng-init="" ng-keyup="search2FA()"/>
<button class="btn btn-default" ng-click="search2FA(1)">
<i class="glyphicon glyphicon-search"></i></button> </form
```

Le contenu du champ 'text' est réinitialisé en appelant la fonction 'search2FA()' avec le paramètre '1'.

* Côté client, les différentes actions sont gérérs par le script « *2ndFA.js* ». Celui-ci est écrit en suivant le formalisme imposé par *AngularJS* qui introduit notamment les notions de « contrôleur » qui va gérer les événements et de « scope » ou contexte de travail des objets. Le contrôleur réagit aux différents clics et « déplie » les différents nœuds constituant l'arbre de l'explorateur de sessions. Pour ce faire, des requêtes HTTP sont émises pour récupérer les données par la fonction 'search2FA()'. Celle-ci envoie au serveur les paramètres de filtrage grâce à la méthode AJAX suivante :

```
$http.get("#{scriptname}sfa/#{sessionType}?
_session_uid=#{$scope.searchString}*&groupBy=substr(_session_uid,
{scope.searchString.length})&U2FCheck=#{$scope.U2FCheck}&TOTPCheck
=#{$scope.TOTPCheck}&UBKCheck=#{$scope.UBKCheck}")
```

* Côté serveur, c'est le module Perl « ::*2ndFA* » qui traite les requêtes, récupère les différents paramètres de filtrage et retourne le résultat. Celui-ci est constitué d'un JSON avec le total de sessions correspondant au filtre et le code résultat puis, si la requête contient un identifiant de session, un JSON avec toutes les paires clef/valeur contenues dans la session persistante :

Requête 1 : manager.fcgi/sfa/persistent?
_session_uid=*&groupBy=substr(_session_uid,0)&U2FCheck=1&TOTPCheck=1
&UBKCheck=1

Réponse 1 : {"values":[{"value":"d","count":1}],"count":1,"total":"1","result":1}

Requête 2 : /manager.fcgi/sfa/persistent/5efe8af397fc3577e05b483aca964f1b

Réponse 2 : {"_utime":"1529010508","_loginHistory":{"successLogin":
[{"_utime":1529010553,"ipAddr":"127.0.0.1"},
{"_utime":1529010508,"ipAddr":"127.0.0.1"}]},"_updateTime":"201806282231
21",.....

4.3.4 – Configuration & Structure

L'implémentation de l'authentification à double facteur a nécessité l'ajout de nombreuses entrées dans la configuration de LLNG. J'ai donc dû analyser sa structure et comprendre comment elle est générée pour pouvoir la modifier et accéder aux données.

La structure du fichier de configuration est construite par le script *jsongenerator.pl* à partir des trois modules : « Build::Tree », « Build::CTree » et « Buil ::Attributes » situés sous le répertoire « Manager ». « Build::Tree » constitue l'arbre de base avec son tronc, les branches et sous-branches. C'est lui qui est lu par l'explorateur de configuration pour afficher le menu. « Build::CTree » est l'arbre complémentaire permettant de greffer des branches supplémentaires. Pour chaque ajout d'un *Vhost*, d'un partenaire SAML, OIDC, CAS, etc …, une branche supplémentaire est créée, téléchargée et greffée dynamiquement à l'arbre de base au fur et à mesure que le menu est déplié, parcouru. Cette mécanique a énormément fluidifié le téléchargement de la configuration par le *Manager* par rapport aux versions antérieures de LLNG. Enfin, toujours par analogie avec l'arbre, le fichier « Buil ::Attributes » contient l'ensemble des feuilles, c'est-à-dire les options de configuration.

« Build::Tree », « Build::CTree » sont tous les deux basés sur le même principe. Chaque option doit être déclarée et rattachée à une branche, un menu. Pour chaque menu ou sous-menu, il faut préciser le nom qui sera utilisé par les fichiers de traduction lors de l'affichage, une URL pointant vers l'aide contextuelle et le format d'affichage de l'option (champ texte, sélection, puces, etc …). Chaque format d'affichage doit correspondre à un *template* défini dans le répertoire « static/form ». Exemple pour le second facteur :

```
title => 'plugins',
nodes => [
                    title => 'secondFactors',
                    nodes => [
                     title => 'u2f',
                     help  => 'u2f.html',
                     form  => 'simpleInputContainer',
                     nodes => [   'u2fActivation', 'u2fSelfRegistration',
                                  'u2fAuthnLevel', 'u2fUserCanRemoveKey',     ]
                     title => 'external2f',
                     help  => 'external2f.html',
                     form  => 'simpleInputContainer',
```

```
nodes => [   'ext2fActivation', 'ext2FSendCommand',
                'ext2FValidateCommand', 'ext2fAuthnLevel',    ]
```

Le module « Buil ::Attributes » est utilisé pour décrire chacune des options de configuration en précisant son type, sa valeur par défaut, un commentaire et un test facultatif (format adresse IP ou mail par exemple). Les types disponibles sont précisés au début du fichier. Après chaque modification, il est nécessaire de reconstruire l'arbre de la configuration afin qu'elle soit prise en compte en exécutant le script « jsongenerator.pl » ou via la commande 'make json'. Exemple de déclaration pour le TOTP :

```
totp2fActivation => {  type   => 'boolOrExpr',   default   => 0,
                        documentation => 'TOTP activation',  },
totp2fSelfRegistration => {   type     => 'boolOrExpr',  default  => 0,
                    documentation => 'TOTP self registration activation',  },
 totp2fAuthnLevel => { type => 'int',  test   => qr/^\d+$/,  documentation =>
      'Authentication level for users authentified by password+TOTP'  }
```

4.3.5 – Tests de configuration

Une fois les modifications apportées à la configuration, celle-ci doit être sauvegardée en base en cliquant sur le bouton « Sauver ». Après cette étape de soumission et avant son envoi pour sauvegarde, un ensemble de tests est réalisé par le module « Conf::Test ». L'ensemble de ces tests sont des fonctions retournées dans une table de hachage, exploitée par le module « Conf::Parser ». Si l'ensemble des tests réussit, la nouvelle configuration est sauvegardée. En cas d'échec, le test peut retourner '1'avec un message d'avertissement ou '0' avec un message d'erreur ce qui interdit la sauvegarde de la configuration.

Suite à l'implémentation de la 2FA, j'ai ajouté les tests suivants :

* Vérification de l'installation des dépendances nécessaires, à savoir « Convert::Base32" pour les TOTP, « Crypt::U2F::Server::Simple » pour la norme U2F et « Auth::Yubikey_WebClient » pour les *Yubikey.* Exemple pour le TOTP :

```
if (   $conf→{totp2fActivation} or $conf->{utotp2fActivation} )
  {   eval "use Convert::Base32";
  return ( 1, "Convert::Base32 module is required to enable TOTP" ) if ($@); } }
```

* Avertir si le nombre de chiffres constituant le TOTP est différent de 6 ou 8. En effet, l'application *GoogleAuthenticator* n'autorise que les TOTP à 6 chiffres tandis que *FreeOTP* accepte les TOTP à 6 ou 8 chiffres :

```
return (  1,  (  ( $conf->{totp2fDigits} == 6 or $conf->{totp2fDigits} == 8 )  ?
'  '  : ' TOTP should be 6 or 8 digits long '   )  );
```

* S'assurer que les paramètres obligatoires pour l'utilisation de certains seconds facteurs soient définis et vérifier leur valeur. Exemple pour les Yubikey :

```
return ( 0, "Yubikey client ID and secret key must be set" )
        unless ( defined $conf->{yubikey2fSecretKey}
          && defined $conf->{yubikey2fClientID} );
 return ( 1,  (  ( $conf->{yubikey2fPublicIDSize} == 12 )  ?
'  '  : 'Yubikey public ID size should be 12 digits long'   )  );
```

* Les routes sont définies par des URL relatives à l'URL du *Portail*, déclarées dans le *Manager* et sauvées dans la variable $conf→{portal}. Il est impératif que l'URL du *Portail* termine par le caractère '/' afin que les URL résultantes soient correctes. J'ajoute donc un '/' s'il est absent et je supprime les '/' inutiles s'ils existent :

```
portalURL => sub {
  $conf->{portal} .= '/'  unless ( $conf->{portal} =~ qr#/$# );
  my $regex = qr#/+$#;$conf->{portal} =~ s/$regex/\//;   return 1;   },
```

4.3.6 – Synthèse

Après deux mois de travail, l'explorateur de sessions avec second(s) facteur(s) est fonctionnel ainsi que l'ensemble des tests de la configuration.

L'explorateur comporte les fonctionnalités suivantes : affichage uniquement des sessions 2FA, filtre d'affichage cumulatif par type de second facteur et moteur de recherche par identifiant de session « _whatToTrace ».

Le plus difficile a été de comprendre et maîtriser le framework *AngularJS* et la syntaxe *CoffeeScript*. Ayant déjà développé en *JavaScript* auparavant, il m'a parfois fallu analyser le code généré pour comprendre la syntaxe *CoffeeScript*.

4.4 – Module Common::TOTP

Pour valider ou vérifier le TOTP saisi par l'utilisateur, les deux modules « 2F::TOTP » et « Register::TOTP » font appel à la fonction « verifyCode ». Celle-ci utilise la méthode privée « _code » qui est une implémentation Perl (Gray, 2015/2018) de la formule décrite dans le paragraphe C – 2.3 à savoir :

- sub verifyCode {

  my ($self, $interval, $range, $digits, $secret, $code) = @_;

  my $s = eval { decode_base32($secret) };

  if ($@) { for (0 .. $range) {

  if ($code eq $self->_code($s, $_, $interval, $digits)) {return 1; }}
  return 0; }

- sub _code {

  my ($self, $secret, $r, $interval, $digits) = @_;

  my $hmac = hmac_sha1_hex(

  pack('H*',sprintf('%016x', int((time - $r * $interval) / $interval))),
  $secret);

  return sprintf('%0' . $digits . 'd', (hex(substr($hmac,
  hex(substr($hmac, -1)) * 2, 8)) & 0x7fffffff) % 10 ** $digits);}

J'ai choisi d'implémenter une fonction spécifique de calcul du TOTP pour pouvoir moduler l'intervalle de temps et le nombre d'intervalles à tester, ces deux paramètres étant configurables via le *Manager,* ce qui n'est pas possible avec la méthode fournie par le module « Authen::OATH ». Celle-ci permet uniquement de choisir le nombre de chiffres et le type de hachage, par défaut 6 et SHA1.

Pour gérer le secret partagé, le module « Register::TOTP » utilise la fonction « newSecret » permettant de générer un code encodé en base32. Ce codage utilise uniquement les caractères de 'A' à 'Z' et '2' à '7' pour éviter que le '0' soit confondu avec 'O', '1' avec 'I' et '8' avec 'B'. Le code de « newSecret » est le suivant :

- sub newSecret { my ($self) = @_;

  my @chars = ('a' .. 'z', 2 .. 7); (plage des caractères disponibles)

  return join('', @chars[map { int(rand(32)) } 1 .. 32]); }
  (sélection aléatoire)

5 – Implémentation définitive

5.1 – Analyse critique

Après quatre mois de travail, l'authentification à double facteur est fonctionnelle côté *Portail* et l'explorateur des sessions 2FA est disponible via le *Manager*. Cette première implémentation comporte trois défauts majeurs.

Tout d'abord, les utilisateurs ne peuvent pas supprimer leurs seconds facteurs. En effet, une fois enrôlé, celui-ci est systématiquement demandé à la connexion si le module correspondant est activé. La seule solution est la suppression par un administrateur de la session persistante de l'utilisateur depuis l'explorateur de sessions du *Manager* ce qui, en outre, supprime tous les seconds facteurs ainsi que l'historique de connexion. Il me faut donc créer une interface spécifique permettant aux utilisateurs de gérer leurs différents seconds facteurs voire les renommer.

Ensuite, la structure de sauvegarde des seconds facteurs en session persistante interdit l'enrôlement de plusieurs équipements de même type ce qui est très pénalisant notamment pour les accessoires U2F. En effet, seul le dernier enrôlé est conservé. De plus, l'utilisation d'une clef en session persistante par type de second facteur et par paramètre rend la présentation des sessions et leur affichage complexes à lire. En effet, une session persistante avec les trois types de second facteur enregistrés comporte 4 entrées dédiées : 2 pour le type U2F, 1 pour le TOTP et 1 pour une *Yubikey*. Le format de sauvegarde en session persistante des seconds facteurs doit être modifié.

Enfin, il est impossible pour les administrateurs de déléguer la gestion du second facteur à une équipe d'exploitation ou un secrétariat par exemple sans donner l'accès au *Manager* ce qui n'est pas une solution très sécurisante. Un module dédié doit être développé.

En outre, cette implémentation initiale souffre de quelques problèmes d'ergonomie.

Côté *Manager*, les clefs relatives aux seconds facteurs sont présentées avec les autres clefs systèmes sans distinction spécifique tant avec l'explorateur classique qu'avec l'explorateur 2FA. Elles sont « noyées » avec les autres clefs car présentées par ordre alphabétique. La mise en page des clefs relatives aux seconds facteurs sera à modifier. En outre, l'explorateur 2ndFA affiche l'ensemble des clefs de session et non pas uniquement les clefs relatives aux seconds facteurs. De plus, la barre de menu ne permet pas de distinguer le module actif, en cours d'affichage parmi les autres modules disponibles.

Coté *Portail*, les utilisateurs ne peuvent pas nommer leurs seconds facteurs, vérifier si leur équipement est correctement enregistré ou savoir lequel est enrôlé. De plus, le fonctionnement actuel ne permet pas de gérer la défaillance ou la perte d'un équipement en imposant, comme certains services tel Google par exemple, l'enrôlement d'un TOTP pour pouvoir enregistrer une clef U2F. Ensuite, si l'authentification avec second facteur est activée, la demande par l'utilisateur de l'affichage de son historique de connexion n'est pas prise en compte car le paramètre correspondant n'est pas transmis entre les étapes d'authentification et d'autorisation. Autre point signalé par un utilisateur via la plateforme GitLab, l'affichage du TOTP existant lors de la phase d'enrôlement devrait pouvoir être désactivé afin d'éviter le vol du secret partagé par un tiers.

L'ergonomie générale est à revoir et souffre d'un manque de cohérence, d'intégration globale tant dans l'organisation des différents seconds facteurs car ayant été développée au fur et à mesure et indépendamment, que dans la structure de sauvegarde des données utilisateur en session persistante interdisant plusieurs seconds facteurs du même type. Avec Xavier Guimard, il a été décidé de créer une page de gestion unique pour l'ensemble des seconds facteurs enrôlés par un utilisateur lui permettant de lister, supprimer ou ajouter, avec un nom, l'équipement. Celui-ci permettra une meilleure intégration afin d'améliorer l'ergonomie de la gestion tant pour l'utilisateur que pour les administrateurs particulièrement en termes de maintenabilité et d'évolutivité facilitant l'ajout de futurs types de seconds facteurs.

5.2 – Structure de données cohérente

Pour l'enregistrement des différents seconds facteurs en session persistante, la structure initiale nécessitait deux clefs pour les équipement U2F, une clef pour les TOTP et une clef pour les *Yubikey*. Par exemple :

```
$session→{_totpSecret}      = '123456'
$session→{_yubikeySecret} = '123456'
$session→{_u2fKH}           = 'abcdef'
$session→{_u2fSecret}       = '123456'
```

Avec la structure définitive, chaque second facteur est enregistré dans un tableau permettant de n'avoir qu'une seule clef en session persistante pour sauver l'ensemble d'entre-eux. Ensuite, chaque second facteur est implémenté en Perl grâce à une table de hachage ou *hash* constitué de plusieurs couples clef – valeur ce qui permet d'enregistrer tous les paramètres d'un second facteur dans le même objet. Pour pouvoir différencier les seconds facteurs, il n'était pas envisageable de se baser sur le nom saisi par l'utilisateur car sans certitude d'être unique. J'ai donc choisi d'ajouter une clef 'epoch' avec pour valeur la date et l'heure d'enrôlement au format 'epoch-POSIX'. En outre, cette clef 'epoch' est utilisée par le navigateur pour afficher la date d'enrôlement du second facteur au bon format et dans le fuseau horaire de l'utilisateur. Par convention, les fonctions et variables précédées du caractère '_' sont considérées comme privées bien que cette notion n'existe pas en Perl. La structure définitive est donc un tableau de *hash*. Par exemple :

```
$session→{_2fDevices} =
[{type => 'U2F', name => 'myU2FKey', epoch => '123456789',
                _userKey => '123456', _keyHandle => 'abcdef'} ,
{ type => 'TOTP', name => 'myTOTP', epoch => '123456789',
                    _secret => '123456'} ,
{ type => 'UBK',   name => 'myYubikey', epoch => '123456789',
                    _secret => '123456'}]
```

Pour sauvegarder cette structure complexe en session persistante donc en base au sens large, il est nécessaire de transformer, encoder ce tableau de *hash* en un JSON encapsulé dans une chaîne de caractères. Pour cela, nous utilisons la fonction « to_json » fournie par le module Perl (« JSON (JavaScript Object Notation) encoder/decoder », s. d.). La fonction réciproque, de décodage est « from_json ». Par exemple :

```
'_2fDevices' =>
'[{"type":"U2F","_keyHandle":"654321","epoch":1535302744,"_userKey":"1234
56","name":"MyU2FKey"},
{"_secret":"123456","epoch":1535302744,"type":"TOTP","name":"MyTOTP"},
{"epoch":1535302744,"_secret":"123456","type":"UBK","name":"MyYubikey"}]'
```

5.3 – Portail

5.3.1 – Evolutions & Paramètres de configuration

Pour prendre en compte la nouvelle structure de sauvegarde des seconds facteurs en session persistante, j'ai repris l'ensemble du code des différents modules et mis en place des mécanismes de limitations. De plus, après étude de la norme U2F, j'ai pu implémenter la possibilité pour l'utilisateur d'enrôler plusieurs clefs U2F. En effet, le paragraphe 5.2.1 de l'API (Balfanz, Birgisson, & Lang, s. d.) précise qu'il est possible de passer à la fonction « verify », avec le challenge, un tableau contenant plusieurs *keyHandle*. Si l'un des *KH* est reconnu, l'authentification est validée :

```
foreach (@u2fs) {
    $kh = $_->{_keyHandle};  $uk = $_->{_userKey};
     my $c = $self->crypter( keyHandle => $kh, publicKey => $uk );
     if ($c) {  push @crypters, $c;   }
```

```
$req->data->{crypter} = \@crypters;  my @rk;
foreach ( @{ $req->data->{crypter} } ) {
        my $k = push @rk,
        { keyHandle => $_->{keyHandle}, version => $data->{version} }; }
```

- La nouvelle structure de sauvegarde permet théoriquement l'enregistrement d'un nombre illimité de seconds facteurs. Pour éviter qu'un utilisateur n'enrôle plusieurs fois le même accessoire, je vérifie si la partie publique de la clef *Yubikey* n'est pas déjà connue et j'ai choisi de n'autoriser l'enrôlement que d'un seul TOTP. En outre, un accessoire U2F génère une clef publique différente à chaque enrôlement. J'ai donc décidé de limiter à 10, le nombre de seconds facteurs par utilisateur : $conf→{max2FDevices} = 10.

```
my $SameUBKFound = 0;
foreach (@$_2fDevices) {
        if ( $_->{_yubikey} eq $key ) { $SameUBKFound = 1; last; } }
my @keep = ();
while (@$_2fDevices) {
        my $element = shift @$_2fDevices;
        push @keep, $element unless ( $element->{type} eq "TOTP" ); }
my $size    = @$_2fDevices;
my $maxSize = $self->conf->{max2FDevices};
if ( $size >= $maxSize ) {
  return $self->p->sendError( $req, 'maxNumberof2FDevicesReached', 400 ); }
```

- Afin que l'affichage de la liste des seconds facteurs et de leurs noms reste lisible, j'ai limité la longueur du nom à 20 caractères ($conf→{max2FDeviceNameLength} = 20). De plus, pour éviter d'éventuelles attaques par injection de code, le nom des seconds facteurs ne peut contenir que des lettres, des chiffres ou le caractère « _ » (\w). Par défaut, la date au format epoch-POSIX est utilisée comme nom.

```
my $keyName = $req->param('keyName');
my $epoch   = time(); $keyName ||= $epoch;
unless ( $keyName =~ /^[\w]+$/ ) {
      return $self->p->sendError( $req, 'badName', 200 ); }
$keyName=substr($keyName, 0, $self->conf->{max2FDevicesNameLength} );
```

'max2FDevices' et 'max2FDeviceNameLength' ne sont pas renseignés via le *Manager*. Il s'agit de deux des paramètres renseignés directement dans la configuration de LLNG et pouvant être surchargés grâce au fichier 'lemonldap-ng.ini'.

- L'affichage ou non du secret TOTP existant est déterminé par la valeur de la variable « $conf→{totp2fDisplayExistingSecret} ». Si cette variable est fausse, le secret n'est pas transmis sauf si l'utilisateur demande un nouveau secret et s'il est autorisé à le changer « $self->conf->{totp2fUserCanChangeKey} » ou s'il n'en possède pas :

```
if ( ( $req->param('newkey') and $self->conf->{totp2fUserCanChangeKey} )
      or not $secret )    {  $secret = $self->newSecret;   $nk = 1;  }
elsif ( $req->param('newkey') ) {
      return $self->p->sendError( $req, 'notAuthorized', 200 );   }
elsif ( $self->conf->{totp2fDisplayExistingSecret} ) {
      $self->logger->debug("User secret = $secret");  }
else { return $self->p->sendError( $req, 'totpExistingKey', 200 );   }
```

- Pour gérer la suppression des seconds facteurs, j'ai ajouté à chaque module d'enrôlement l'action 'delete'. Celle-ci consiste en une boucle parcourant et recopiant le tableau des seconds facteurs de l'utilisateur sauf celui dont les paramètres 'type' et 'epoch' correspondent à ceux passés dans la requête appelant l'action. De plus, cette boucle n'est exécutée que si l'utilisateur a été autorisé à supprimer ce type de second facteur ($conf →{*2fUserCanRemoveKey} :

```
unless ( $self->conf->{u2fUserCanRemoveKey} ) {
        return $self->p->sendError( $req, 'notAuthorized', 200 );  }
my @keep = ();
while (@$_2fDevices) { my $element = shift @$_2fDevices;
      push @keep, $element unless ( $element->{epoch} eq $epoch ); }
```

- Pour gérer la demande par l'utilisateur de l'affichage de son historique de connexion, j'ai ajout dans les formulaires un paramètre de type « hidden » permettant de passer la valeur de la « checkbox » entre les étapes de connexion :

```
<input type="hidden" id="checkLogins" name="checkLogins"
value="<TMPL_VAR NAME="CHECKLOGINS">">
```

- Enfin, j'ai modifié la règle d'activation des modules pour correspondre à la nouvelle structure de sauvegarde. Par exemple pour l'U2F :

```
self->conf->{u2fActivation} = '$_2fDevices
                                && $_2fDevices =~ /"type":\s*"U2F"/s';
```

5.3.2 – Module 2F::Engines::Default - Moteur & Gestionnaire 2FA

Ce module est « le cœur de la mécanique » implémentant l'authentification avec seconds facteurs. Il gère côté *Portail* le chargement des différents modules liés au second facteur ainsi que la redirection des utilisateurs vers les pages d'enrôlement ou de connexion adéquates. Il s'agit sans aucun doute de la partie la plus complexe de la 2FA. Le module « 2F::Engines::Default » ou moteur 2FA, conçu et écrit essentiellement par Xavier Guimard, utilise les *templates* « 2fchoice.tpl » et « 2fregisters.tpl ».

Succinctement, les différentes étapes de la phase d'authentification sont définies dans un tableau initialisé dans la requête par la fonction 'login'. Les principales étapes sont le contrôle de l'URL, le processus d'authentification (authProcess), la génération de la session (sessionData) et la validation de la session (validSession). 'authProcess' interprète le formulaire de connexion, collecte les données de l'utilisateur dans la base de données des utilisateurs et vérifie le mot de passe. 'sessionData' calcule les macros (setMacros), les groupes (setGroups) et initialise le processus d'authentification avec second facteur (secondFactor). 'validSession' met à jour l'historique de connexion et construit le cookie SSO. Il existe des points d'entrée entre les principales étapes, à disposition des développeurs, pour intercaler des étapes supplémentaires (beforeAuth, afterData, ...).

Au démarrage du *Portail*, « Portal::Main:Init » charge, entre autres, le moteur 2FA puis « Portal::Main:Process » exécute l'étape 'secondFactor' :

```
$self->{_sfEngine} = $self->loadPlugin( $self->conf->{sfEngine} );
sub secondFactor {
    my ( $self, $req ) = @_;
    return $self→_sfEngine→run($req); }
```

Par défaut, la variable de configuration système 'sfEngine' contient, la valeur '::2F::Engines::Default'.

* Le fonctionnement du module « Engines::Default » est le suivant :

- Après avoir été chargé, le moteur 2FA exécute la fonction « init ». Celle-ci utilise les variables systèmes 'available2FSelfRegistration' et 'available2F' contenant respectivement 'TOTP, U2F, Yubikey' et 'UTOTP, TOTP, U2F, REST, Ext2F, Yubikey' pour déterminer quels sont les seconds facteurs activés et quels sont ceux dont l'auto-enrôlement est autorisé. Pour ce faire, le moteur 2FA teste les variables '*2fSelfRegistration' et '*2fActivation' où '*' vaut successivement les valeurs contenues dans les variables systèmes 'available*' citées précédemment puis charge les modules correspondants. Ensuite, « init » crée la route non-authentifiée '2fchoice' associée à la méthode POST qui appelle la fonction « _choice » si au moins deux modules d'authentification sont activés et les routes authentifiées '2fregisters' qui appelle « _register » pour GET et « _displayRegister » pour POST si au moins un module avec auto-enrôlement est activé.

```
if ( @{ $self->{sfModules} } > 1 ) {
    $self->addUnauthRoute( '2fchoice' => '_choice',  ['POST'] );
    $self->addUnauthRoute( '2fchoice' => '_redirect', ['GET'] );    }
if ( @{ $self->{sfRModules} } ) {
    $self->addAuthRoute( '2fregisters'  => '_displayRegister', ['GET'] );
    $self->addAuthRoute( '2fregisters'  => 'register', ['POST'] ); }
```

- La fonction 'run' est exécutée systématiquement lorsqu'un utilisateur souhaite s'authentifier sur le *Portail*. Une fois le formulaire de connexion soumis, la fonction sauvegarde l'état du paramètre de demande d'affichage de l'historique de connexion puis redirige l'utilisateur vers le menu du *Portail* si aucun module de second facteur n'est chargé donc activé. Dans le cas contraire, les règles d'activation sont analysées pour déterminer si un second facteur doit être fourni par l'utilisateur lors de la connexion. Les données de session de l'utilisateur sont sauvegardées sur le jeton anti-CSRF.

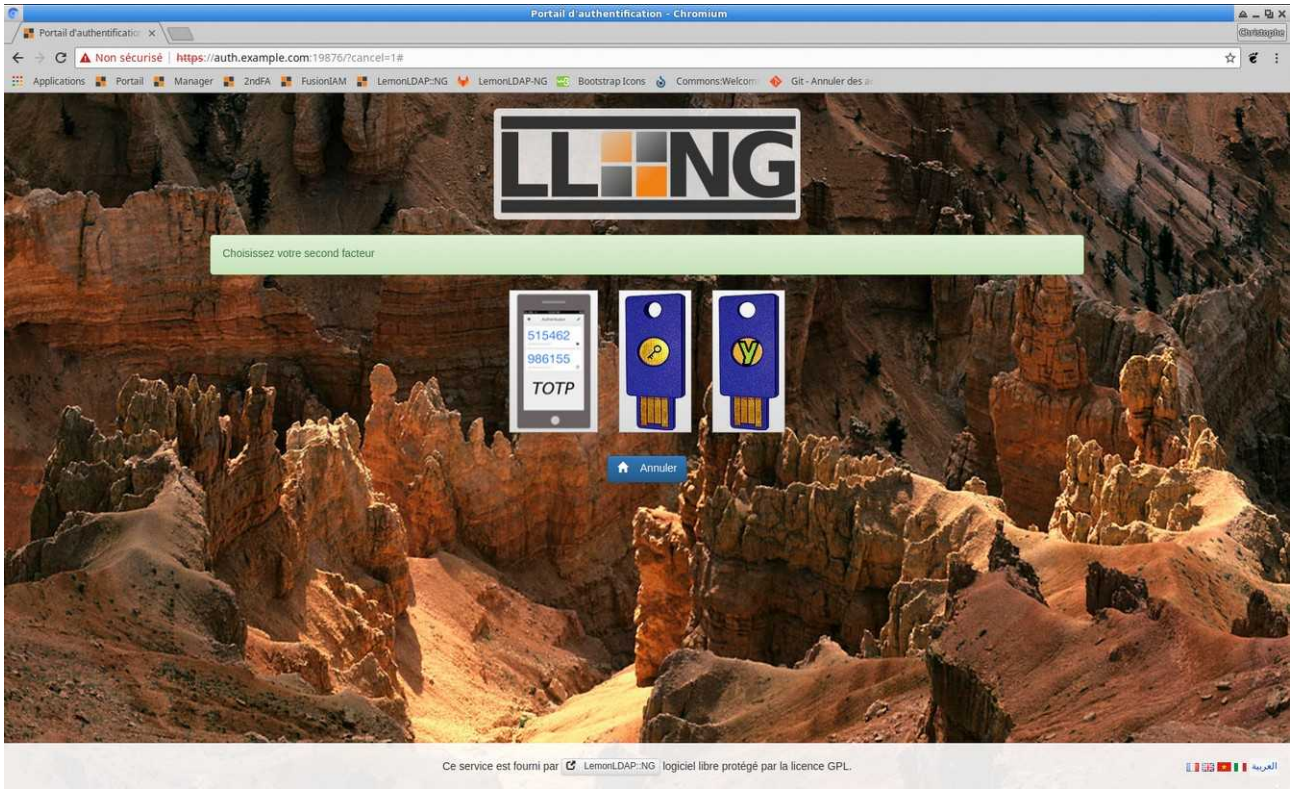

Figure 25 : Page de connexion avec choix

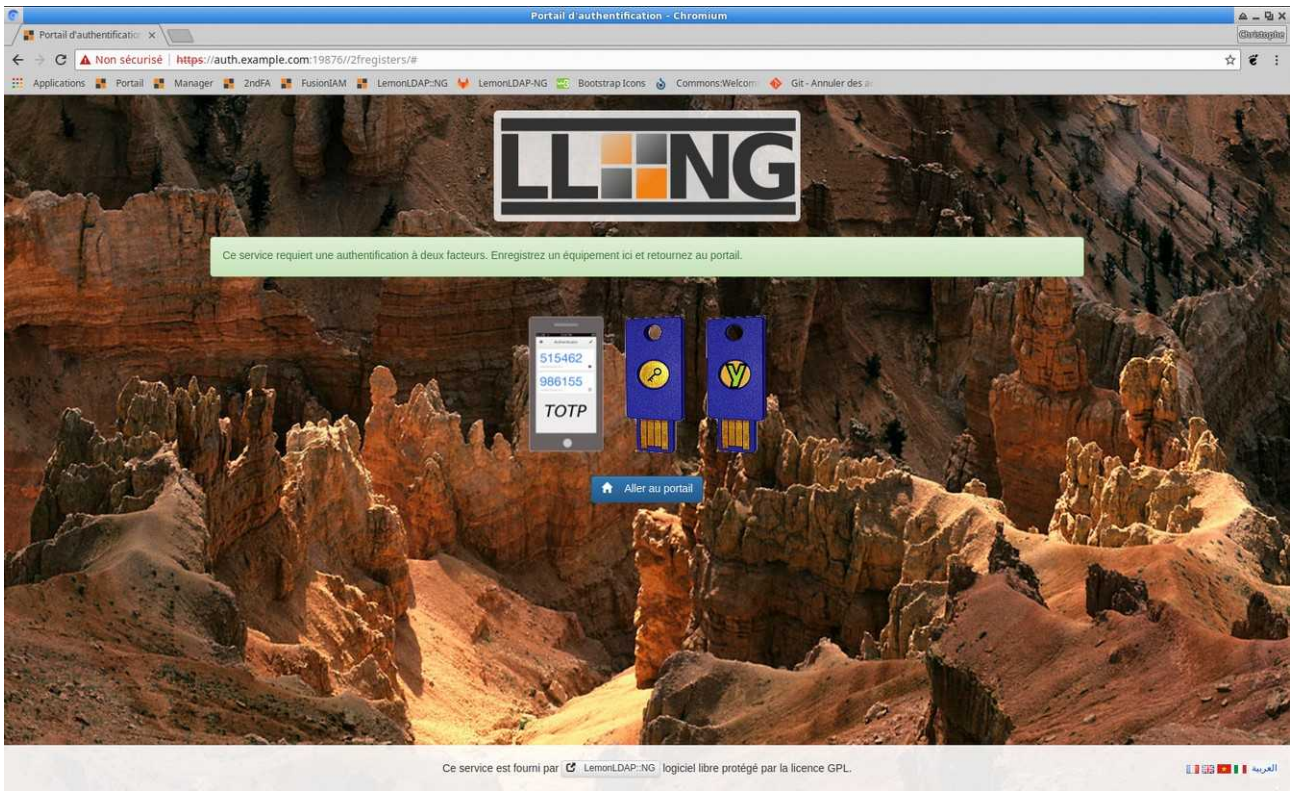

Figure 26 : Page d'enrôlement avec second facteur requis

Si plusieurs seconds facteurs sont activés, la page de choix est chargée en appelant le *template* « 2fchoice.tpl » (cf. figure 25) avec passage de la liste des seconds facteurs autorisés et le paramètre d'affichage de l'historique sinon la page de l'unique second facteur est présentée. Si l'option 'sfRequired' est activée, un second facteur doit obligatoirement être enrôlé par l'utilisateur s'il n'en possède pas. Dans ce cas, un code HTTP 302 est retourné par le serveur et l'utilisateur est redirigé vers la page d'enrôlement.

```
my $tpl  =  $self→p→sendHtml( $req,  '2fchoice', params => {
  SKIN        => $self→conf→{portalSkin},  TOKEN   => $token,
  MODULES => [ map { { CODE => $_->prefix, LOGO => $_->logo } } @am ],
  CHECKLOGINS => $checkLogins  } );
```

- « _choice » est appelée par le formulaire lorsque l'utilisateur clique sur le second facteur de son choix (<form action="/2fchoice" method="POST">). Elle vérifie la présence et la validité du jeton anti-CSRF, le reconstruit et exécute les quatre étapes suivante : appel du *template* correspondant au second facteur choisi par l'utilisateur avec passage du jeton anti-CSRF. Si l'utilisateur est autorisé, alors il y a contrôle de l'URL demandée, appel des étapes « afterData » pour affichage des éventuelles notifications ou de l'historique de connexion si demandé par l'utilisateur et construction du cookie SSO sinon, redirection vers le *Portail*.

```
return $self->p->do(
   $req, [sub { $res }, 'controlUrl', 'buildCookie',@{ $self->p->afterData }   ] );
```

- « displayRegister » affiche le *template* spécifique du second facteur s'il est précisé dans la requête (lien par exemple). Si un seul module est activé alors il y affichage de la page correspondante sinon la page de gestion des seconds facteurs est chargée par l'appel du *template* « 2fregisters.tpl » avec passage de la liste des seconds facteurs enregistrés dans le paramètre 'SFDEVICES'. De plus, le paramètre 'REG_REQUIRED' est transmis pour activer ou non l'affichage d'un bandeau spécifique pour indiquer à l'utilisateur que l'enrôlement d'un second facteur est obligatoire s'il n'en possède pas encore (cf. figure 26).

```
return $self→p→sendHtml( $req,  '2fregisters', params => {
      SKIN  => $self->conf->{portalSkin},
       MODULES      => \@am, (liste des 2FA disponibles)
       SFDEVICES    => $_2fDevices, (liste des 2F existants)
       REG_REQUIRED => $req→data→{sfRegRequired},  }  ); (2F requis)
```

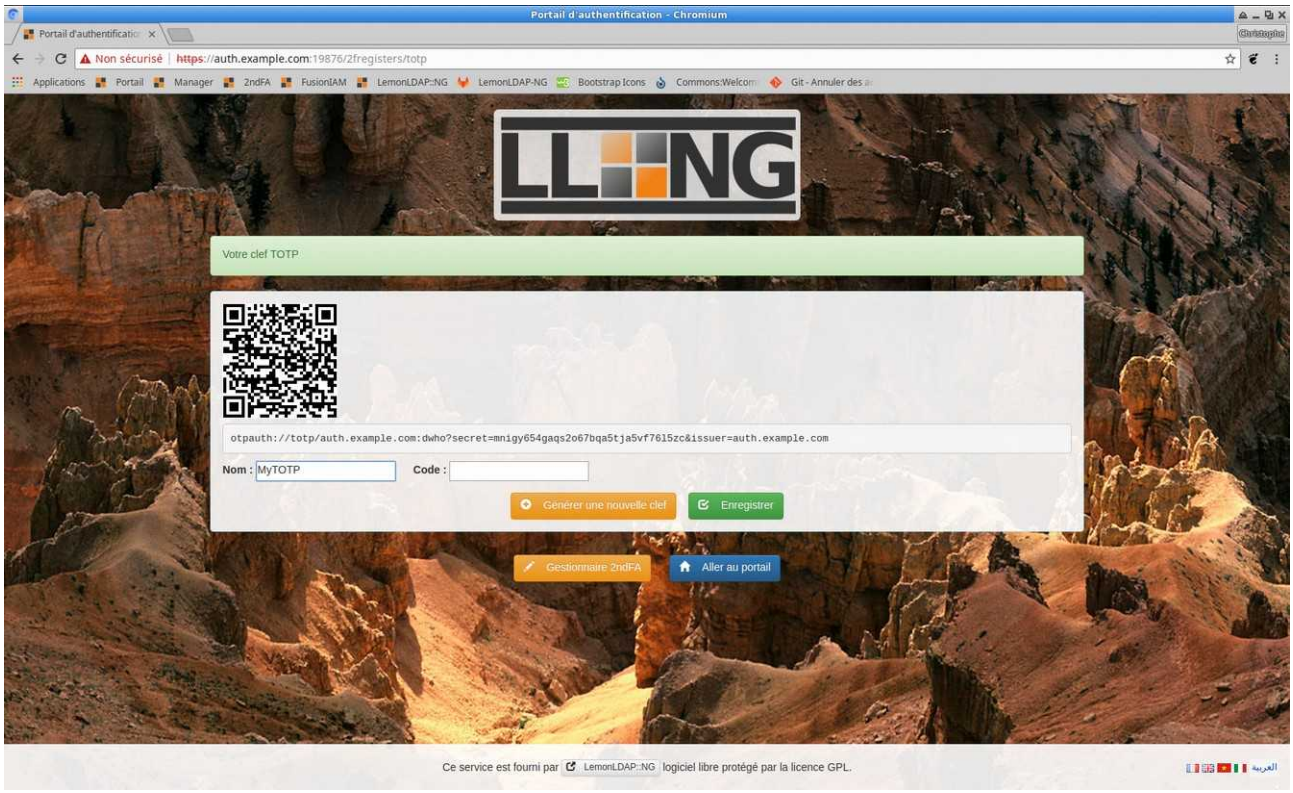

Figure 27 : Formulaire d'enrôlement TOTP

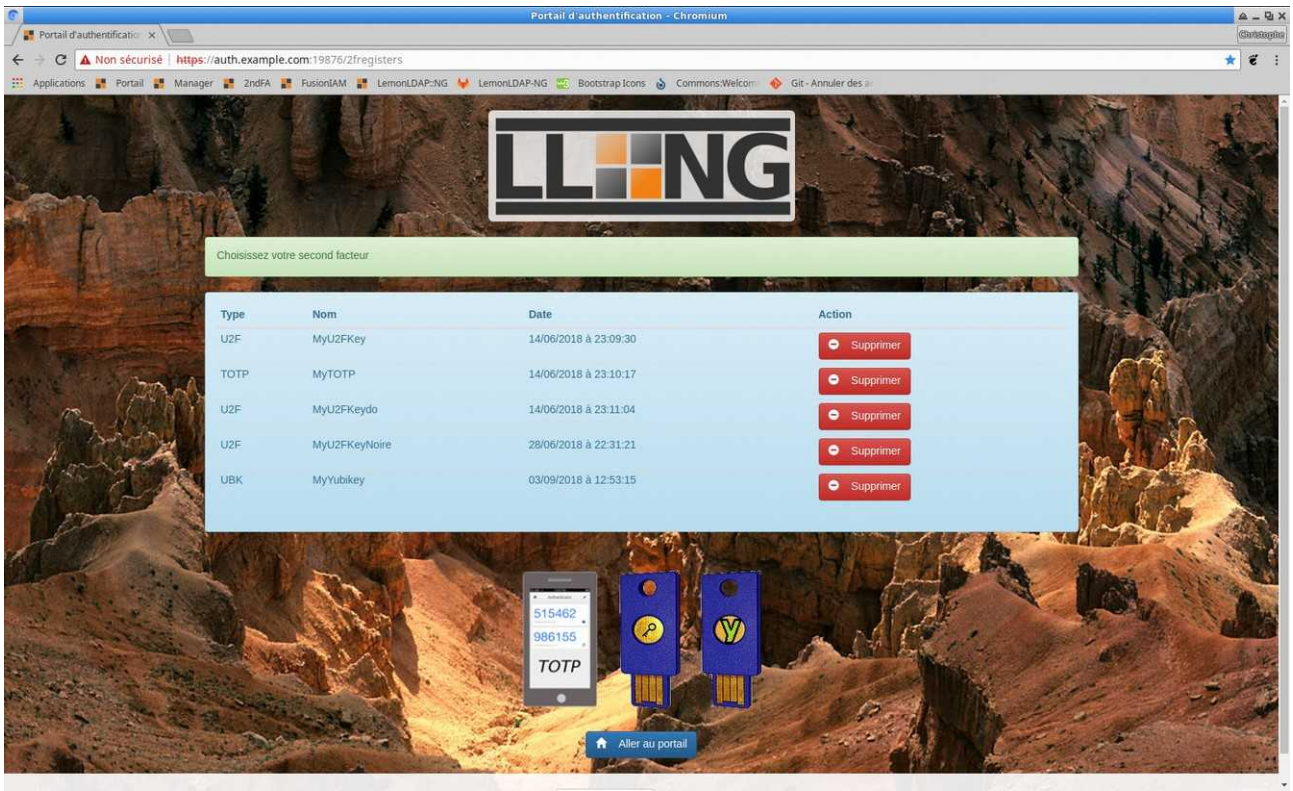

Figure 28 : Gestionnaire 2ndFA

- La fonction « register », appelée lors du clic par l'utilisateur sur l'image du second facteur qu'il souhaite enregistrer, charge la page d'enrôlement correspondante en passant l'action « register » dans l'URL (cf. figure 27). Si celle-ci n'est pas valide, la fonction retourne la liste des modules disponibles pour cet utilisateur :

```
return $m->{m}->run( $req, @args );
```

- « display2fRegisters » retourne « 1 » si un module avec auto-enrôlement est activé. Celle-ci est utilisée pour afficher ou non le lien vers la page de gestion des seconds facteurs dans le menu du *Portail.*

* Le *template* « 2fregisters.tpl » constitue le « gestionnaire 2FA » (cf. figure 28) permettant à l'utilisateur de lister et supprimer ses seconds facteurs. Il est constitué d'un bandeau activé par la variable 'REG_REQUIRED', d'une table utilisant le paramètre 'SFDEVICES' pour la présentation des seconds facteurs enrôlés et d'une boucle exploitant 'MODULES' pour afficher les seconds facteurs pouvant être enregistrés. Lors du clic sur le bouton de suppression, la fonction JavaScript 'delete2F', fournie par « 2fregistration.js », est exécutée.

* « 2fregistration.js » permet de gérer l'action d'effacement du second facteur et la conversion d'affichage des dates exprimées au format epoch-POSIX. Lors du clic sur le bouton de suppression, la fonction capture l'Id de la ligne qui est en fait la date d'enrôlement au format epoch-POSIX et le type du second facteur puis appelle via l'URL l'action 'delete' du module Perl correspondant :

```
$(".data-epoch").each ->
                    myDate = new Date($(this).text() * 1000)
                    $(this).text(myDate.toLocaleString())
$('body').on 'click', '.btn-danger', () -> delete2F ( $(this).attr 'device' ),
                                                    ( $(this).attr 'epoch' )
$.ajax type: "POST"
        url: "#{portal}2fregisters/#{device}/delete"
                    data: epoch: epoch
                    url: "#{portal}2fregisters/#{device}/delete"
```

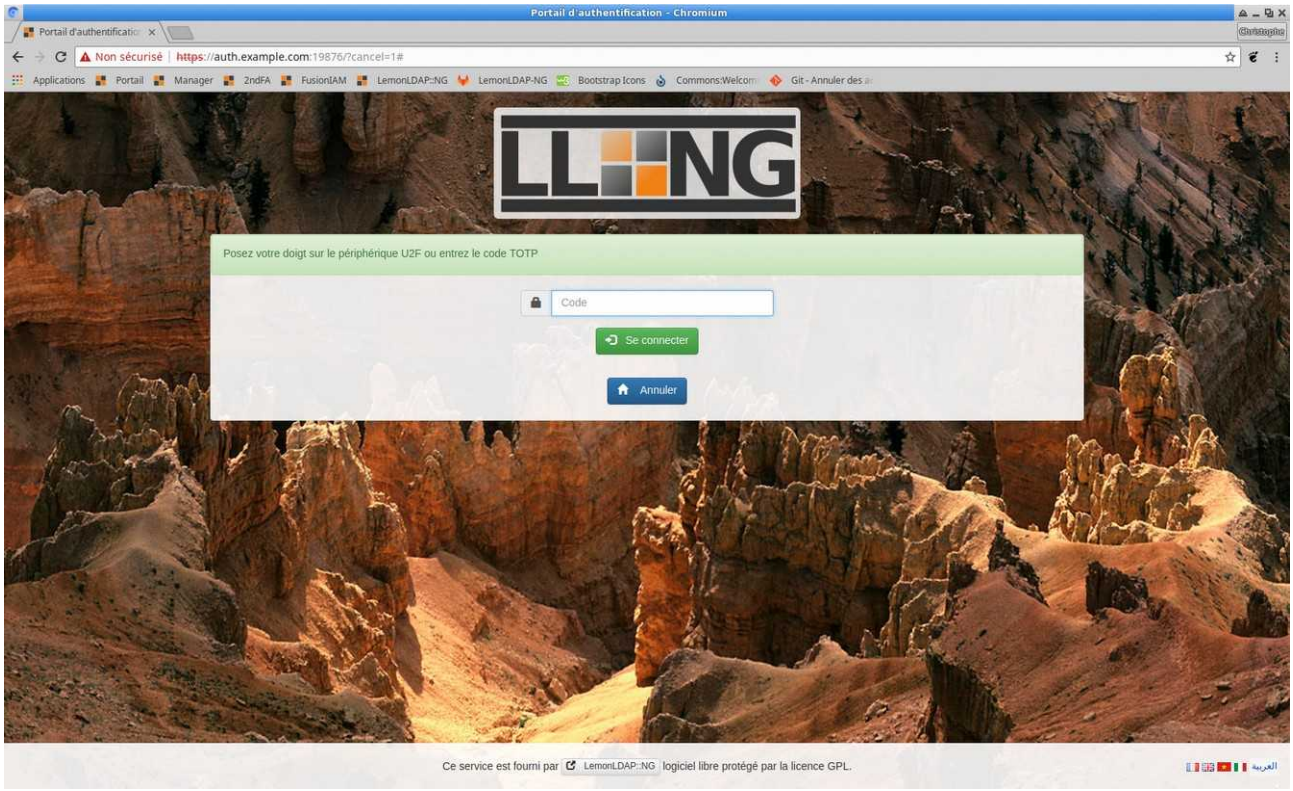

Figure 29 : Page de connexion UTOTP

5.3.3 – Module 2F::UTOTP

Pour pallier les risques de perte ou de panne du second facteur, entraînant de fait l'impossibilité pour l'utilisateur d'accéder à son compte, des fournisseurs de services comme *GitLab* ou *Google* imposent aux utilisateurs l'enregistrement de deux seconds facteurs différents, généralement un TOTP et un accessoire U2F voire, imposent l'enrôlement d'un TOTP pour pouvoir enregistrer un équipement U2F. Mais, dans un souci d'ergonomie de l'étape de connexion, le même formulaire est présenté à l'utilisateur lui permettant d'utiliser l'un ou l'autre.

Afin de calquer ce mode de fonctionnement, le module UTOP a été implémenté (cf. figure 29). En outre, nous avons choisi de ne pas imposer l'enregistrement d'un TOTP avant l'accessoire U2F. Grâce aux règles basées sur les expressions régulières, LLNG « impose » sa mécanique mais laisse la possibilité à l'intégrateur de choisir le fonctionnement de l'authentification à second facteur.

A la différence des modules U2F et TOTP, UTOTP ne permet pas l'enrôlement. Le TOTP ou les équipements U2F doivent être enrôlés via les modules « Register » correspondants. En revanche, les modules U2F et TOTP ne doivent pas être activés en même temps que UTOTP.

* Tout comme les autres modules, UTOTP dispose des fonctions :

- « init » qui modifie la règle d'activation et charge les modules U2F et TOTP.

```
$self->conf->{utotp2fActivation} = '$_2fDevices && $_2fDevices =~ /"type":\s*"(?:TOTP|U2F)"/
```

- Tout d'abord, « run » sauvegarde le paramètre d'affichage de l'historique. Ensuite, la fonction vérifie si l'utilisateur a enrôlé un équipement U2F. Si oui, « run » génère un challenge, charge la liste des équipements U2F et appelle le *template* « utotp2fcheck.tpl » en passant l'ensemble des paramètres dans la variable 'DATA'. Si 'DATA' est définie, le *template* charge le 'u2fcheck.js' et propose d'utiliser l'accessoire U2F (paramètre signature) ou le TOTP (paramètre code) sinon seulement le TOTP :

```
<TMPL_IF NAME="DATA">
<span trspan="touchU2fDeviceOrEnterTotp"> </span>
<TMPL_ELSE> <span trspan="enterTotpCode"> </span>
```

- Lors de la soumission du formulaire, la fonction « verify » teste si le paramètre 'signature' est renseigné. Si oui, la fonction « verify » du module U2F est exécutée. Si 'code' est renseigné alors « verify » de « TOTP est appelée sinon le message d'erreur de formulaire vide est retourné :

```
if ( $req->param('signature') ) { $self->u2f->verify( $req, $session  ) };
if ( $req->param('code') )      { $self->totp->verify( $req, $session ) };
```

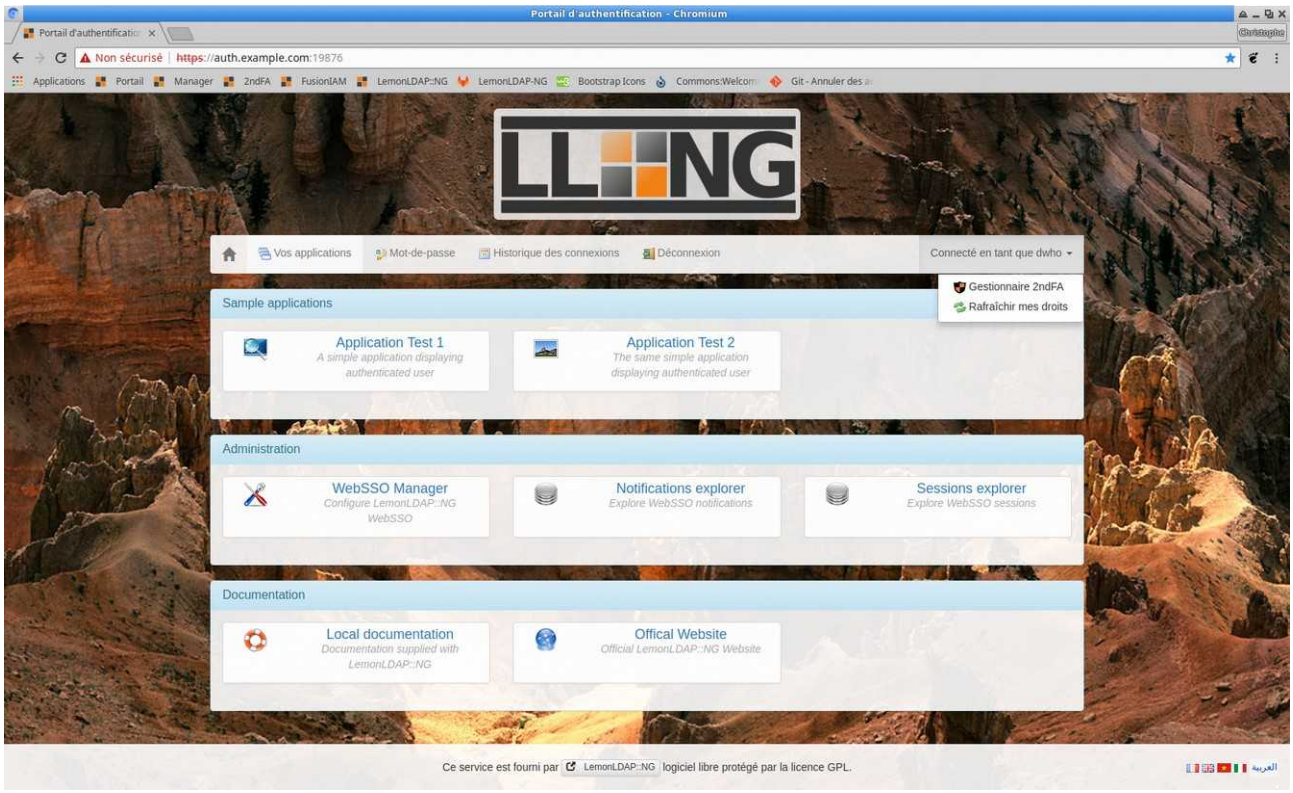

Figure 30 : Portail et lien vers le Gestionnaire 2ndFA

5.3.4 – Expérience utilisateur

Pour améliorer l'utilisation et l'esthétique du *Portail*, j'ai utilisé la fonction « display2fRegisters » pour afficher ou non le lien vers la page de gestion des seconds facteurs (cf. figure 30). Afin de respecter le DFSG, j'ai pris soin, sur conseil de mon tuteur ;-), de télécharger une image distribuée sous licence Creative Common 3.0 que j'ai mis aux couleurs de LLNG et dont j'ai précisé l'origine dans le fichier COPYING.

Le menu est construit par le module « Main::Menu » en utilisant le *template* « menu.tpl » :

```
$res{sfaManager} =
$self->p->_sfEngine->display2fRegisters( $req, $req->userData );

<TMPL_IF NAME="sfaManager">
<li><a href="/2fregisters">
<img src="<TMPL_VAR
NAME="STATIC_PREFIX">common/icons/sfa_manager.png"
width="16" height="16" alt="refresh" />
<span trspan="sfaManager">sfaManager</span> </a></li>
</TMPL_IF>
```

5.3.5 – Tests de non-régression

Suite à l'ajout des options permettant de demander l'affichage de l'historique de connexion, d'imposer l'enrôlement d'un second facteur, j'ai créé les tests correspondants. Pour ce faire, j'ai dupliqué puis modifié les différents tests relatifs aux seconds facteurs pour tester la présence et la validité de ces différents paramètres.

* La fonction 'expectForm' permet de vérifier que les paramètres sont bien transmis lors de la soumission du formulaire. Puis, je teste la présence de la balise « trspan="lastLogins" » :

```
my ( $host, $url, $query ) =
expectForm( $res, undef, '/totp2fcheck', 'token', 'checkLogins' );
ok( $res->[2]->[0] =~ /trspan="lastLogins"/, 'History found' )
```

* En activant uniquement le TOTP et l'option 'sfRequired', un utilisateur n'ayant pas de second facteur est redirigé directement vers la page d'enrôlement. La fonction 'expectRedirection' permet de tester l'envoi par le serveur d'un code 302 de redirection ainsi que l'URL retournée :

```
my $client = LLNG::Manager::Test→new( {   ini => { logLevel    => 'error',
                totp2fSelfRegistration => 1,
                totp2fActivation         => 1,
                sfRequired                => 1, } } ) ;
expectRedirection( $res, qr'http://auth.example.com/+2fregisters/?' );
ok( $res->[2]->[0] =~ m#/2fregisters/totp#, 'Found TOTP link' );
```

* Pour valider le fonctionnement du gestionnaire 2FA, j'ai écrit le test *75-2F-Registers.t* :

- J'active l'authentification avec TOTP et U2F ainsi que l'auto-enrôlement :

```
 my $client = LLNG::Manager::Test→new(  {  ini => { logLevel  => 'error',
            totp2fSelfRegistration  => 1,
            totp2fActivation          => 1,
            u2fSelfRegistration      => 1,
            u2fActivation              => 1, } } );
```

- Je me connecte au *Portail*, enrôle un TOTP. Puis, je me déconnecte, me ré-authentifie et appelle la page du gestionnaire 2FA :

```
ok(  $res = $client→_get('/2fregisters', cookie => "lemonldap=$id",
                                accept => 'text/html', ), 'Form 2fregisters' );
ok( $res->[2]->[0] =~ /2fregistration\.(?:min\.)?js/, 'Found 2f registration js' );
```

- Ensuite, je vérifie la réponse à la recherche des images 'u2f.png' et 'totp.png' ainsi que les liens vers les pages d'enrôlement :

```
ok( $res->[2]->[0] =~
qr%<img src="/static/bootstrap/totp.png" alt="totp2F" title="totp2F" /> )
ok( $res->[2]->[0] =~ qr%<a href="/2fregisters/totp" class="nodecor">% );
```

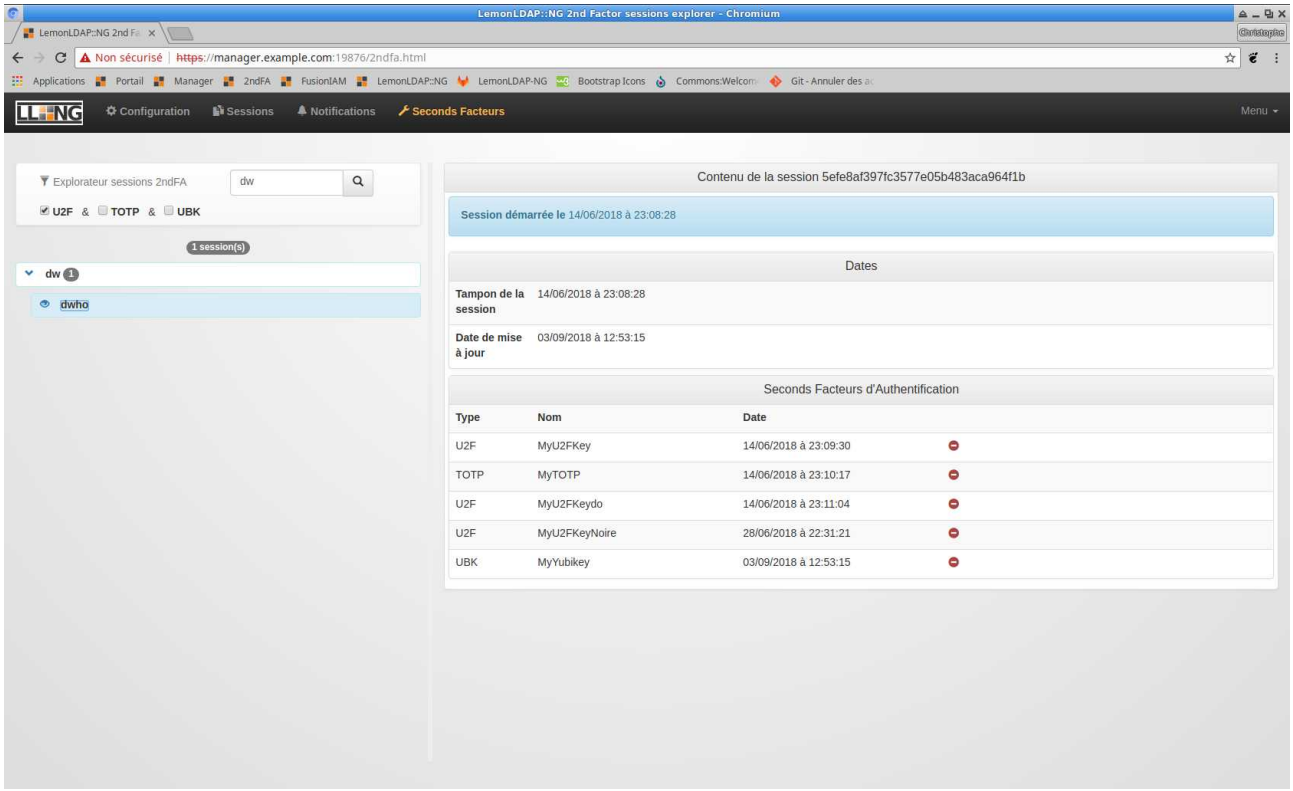

Figure 31 : Explorateur de sessions 2ndFA

- Enfin, je capture la balise 'epoch' du TOTP, appelle la fonction de suppression, recharge la page et vérifie que la ligne du TOTP n'est plus affichée :

```
ok(  $res->[2]->[0] =~
qr%<td class="align-middle" >TOTP</td><td class="align-middle">
(\d{10})</td><td class="data-epoch">\d{10}</td>%,"TOTP epoch $1 found")
ok(  $res = $client→_post( '/2fregisters/totp/delete',
        IO::String->new("epoch=$1"),
        length => 16,  cookie => "lemonldap=$id" ),  'Delete TOTP query' );
ok( $res->[2]->[0]   !~
qr%<td class="align-middle" >TOTP</td><td class="align-middle">
(\d{10})</td><td class="data-epoch">\d{10}</td>%, "TOTP deleted" );
```

5.4 – Manager

5.4.1 – Module Manager::2ndFA : Explorateur de sessions 2ndFA

Tout comme l'explorateur de notifications ou de sessions, l'explorateur « 2ndFA » est un module indépendant (cf. figure 31). De fait, son accès peut être autorisé à une catégorie particulière d'utilisateurs par la mise en place d'une règle sans pour autant donner l'accès au module de configuration du *Manager*. Par exemple : $unite =~ /\bITOps\b/.

Tout d'abord, j'ai modifié le ficher « 2ndFA.coffee » pour présenter dans une catégorie spécifique et sous forme de tableau les seconds facteurs enregistrés en session persistante.

```
if session[attr].toString().match(/"type":\s*"(?:TOTP|U2F|UBK)"/)
    subres.push
    title: "type"
    value: "name"
    epoch: "date"
    array = JSON.parse(session[attr])
```

Ensuite, j'ai ajouté la possibilité de supprimer les seconds facteurs d'un utilisateur depuis cet explorateur par l'ajout d'un bouton et la création d'une fonction exposée par « 2ndFA.js ». Lors du clic, la fonction capture l'identifiant de la ligne construit avec la date au format epoch-POSIX du second facteur ainsi que son type et appelle l'URL « https://portail/manager/sfa/persistente/id?type&epoch » avec la méthode DELETE et les deux paramètres passés dans la *QueryString* :

```
$http['delete']("#{scriptname}sfa/#{sessionType}/
#{$scope.currentSession.id}?type=#{type}&epoch=#{epoch}")
```

Dans le module « 2ndFA », cette URL correspond à une route authentifiée qui appelle la fonction 'del2F'. Celle-ci vérifie la présence et la valeur des paramètres puis appelle la méthode 'delete2F' fournie par le serveur REST de « Common ». En effet, contrairement au *Portail*, le *Manager* ne peut pas accéder directement aux sessions.

```
if ( $type =~ /\b(?:U2F|TOTP|UBK)\b/ ) {
    return $self->delete2F( $req, $session, $skey ); }
```

5.4.2 – Tests de non-régression

Tous les tests relatifs au *Manager* sont situés sous le répertoire « /t ». Ceux correspondants aux attributs ou aux modules « notifications, sessions» ayant déjà été écrits par Xavier Guimard, je me suis basé sur « *sessions.t* » pour écrire *« 2ndFA.t »*.

- Pour ce test, je commence par créer une session temporaire pour valider l'accès et quatre sessions persistantes dont seules trois comportent un ou plusieurs seconds facteurs différents :

```
$ids[1] = newSession( 'msmith', '127.10.0.1', 'Persistent', []);
$sfaDevices = [   {   "type"    => "U2F",    "epoch"   => $epoch   },
                  {    "type"   => "UBK",  "epoch"   => $epoch  }];
$ids[2] = newSession( 'rtyler', '127.10.0.1', 'Persistent', $sfaDevices );
$sfaDevices = [   {  "type"   => "U2F",   "epoch"   => $epoch   },
                  {  "type"   => "TOTP",     "epoch"   => $epoch  },
                  { "name" => "MyYubikey", "type"  => "UBK",
                    "_secret" => "123456", "epoch"   => $epoch } ];
```

- Ensuite, via l'explorateur de sessions, je vérifie l'existence des cinq sessions persistantes :

```
for ( my $i = 1 ; $i < 6 ; $i++ ) {
   $res = &client->jsonResponse("/sessions/persistent/$ids[$i]");
  ok( ( $res→{uid} and $res->{uid} =~ /^(?:dwho|rtyler|msmith|davros|tof)$/)}
```

- Puis, je teste le filtrage par type de second facteur avec différentes combinaisons et analyse la réponse du serveur :

```
$res = &client->jsonResponse( '/sfa/persistent',
'uid=*&groupBy=substr(uid,0)&U2FCheck=1&TOTPCheck=1&UBKCheck=1' );
(tout)
'uid=*&groupBy=substr(uid,0)&U2FCheck=2&TOTPCheck=2&UBKCheck=1');
(U2F & TOTP)
ok( $res->{result} == 1,      'Search "uid"=* & UBK & TOTP - Result code = 1' );
ok( $res->{values}->[0]->{value} && $res->{values}->[0]->{value} eq 'd') ;
ok( $res->{values}->[0]->{count} == 2, 'Found 2 sessions starting with "d" &
U2F & TOTP' );
```

- et vérifie les réponses fournies par le moteur de recherche avec plusieurs requêtes et les filtrent par type :

```
$res = &client->jsonResponse( '/sfa/persistent',
'uid=da*&groupBy=substr(uid,2)&U2FCheck=1&TOTPCheck=1&UBKCheck=2' )
ok( $res->{result} == 1, 'Search "uid"=da* & UBK - Result code = 1' );
ok( $res->{count} == 0,  'Found 0 session with "da" & UBK' )
```

- enfin, je teste la suppression de chaque type de second facteur et vérifie qu'il n'ait plus de session affichée par l'explorateur 2ndFA :

```
foreach ( 3 .. 4 ) {
        ok(  $res = &client->_del( "/sfa/persistent/$ids[$_]",
        "type=TOTP&epoch=$epoch" ) );
        ok( $res->[0] == 200, 'Result code is 200' ); }
$res = &client->jsonResponse( '/sfa/persistent',
         'groupBy=substr(uid,1)&U2FCheck=1&TOTPCheck=1&UBKCheck=1' );
ok( $res->{result} == 1, 'Result code = 1' );
ok( $res->{count} == 0,  'Found 0 session with 2F device' ) ;
```

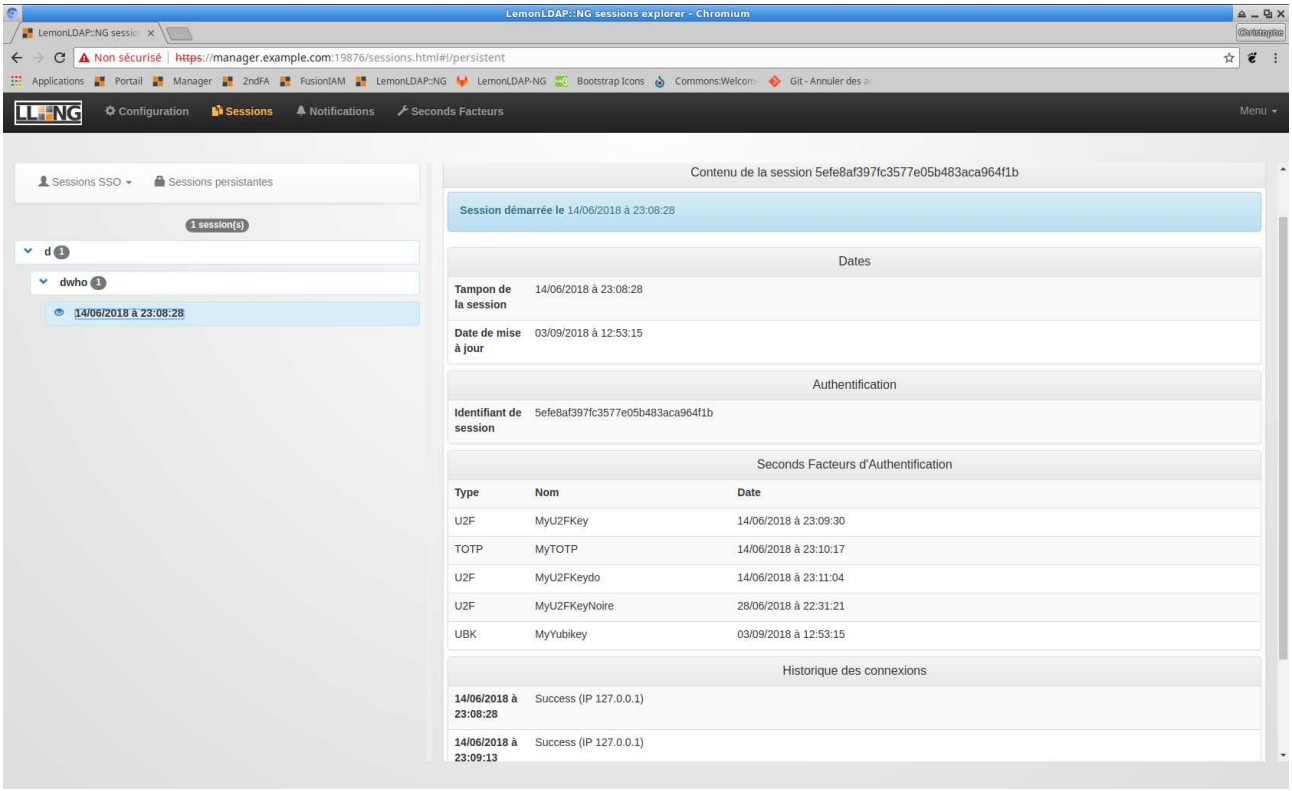

Figure 32 : Explorateur de sessions avec entrées 2FA

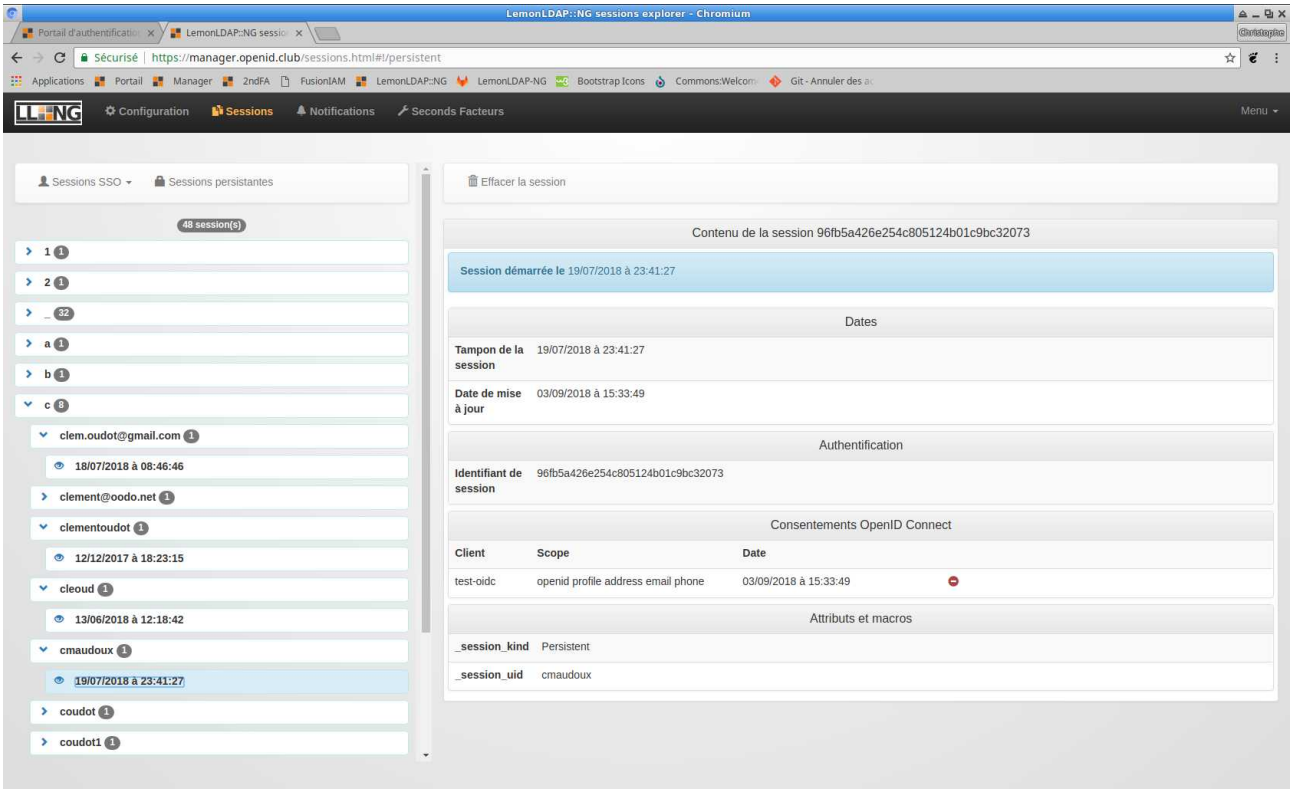

Figure 33 : Explorateur de sessions avec consentements OIDC

5.4.3 – Expérience utilisateur

Pour améliorer l'interface du *Manager*, j'ai souhaité coloriser le module actif dans la barre du menu. Après deux semaines de recherches et tests, je suis parvenu au résultat escompté. Le plus difficile était de déterminer le module actif lors du chargement initial du menu. En effet, les modules sont présentés dans l'ordre de chargement déclaré dans le fichier 'lemonldap-ng.ini'.

Pour ce faire, j'ai modifié le *template* « menubar.tpl » et les *JavaScript* spécifiques à chaque module :

- Initialisation au chargement du *JavaScript* par le *template* de la variable spécifiant le module actif et la couleur du lien :

```
$scope.activeModule = "2ndFA"
$scope.myStyle = {color: '#ffb84d'}
```

- Au chargement, si la variable 'activeModule' correspond au lien, le style 'color' est valide. Lors de l'événement clic 'mousedown', je change le style du lien pour modifier sa couleur :

```
<li ng-mousedown="clickStyle={color: '#ffb84d'}"><strong>
<i ng-if="activeModule == l.title"  ng-style="myStyle" class="glyphicon
glyphicon-{{l.icon}}"></i></li>
```

De plus, j'ai reporté dans l'explorateur de session la mise en page des seconds facteurs développée pour l'explorateur 2ndFA (cf. figure 32) et utilisé la même implémentation pour présenter et supprimer les consentements OpenID Connect (cf. figure 33). Cette évolution a été l'occasion d'un débat avec Clément Oudot qui souhaitait pouvoir également supprimer les seconds facteurs depuis l'explorateur de sessions. Mais, j'ai estimé que cette fonctionnalité faisait double emploi et qu'un module dédié aux seconds facteurs permettait de déléguer leur gestion à un groupe spécifique de personnels autre que nécessairement administrateurs. Ceci reste dans l'esprit de LLNG qui est de ne rien imposer mais d'offrir un maximum de souplesse afin d'adapter LLNG au fonctionnement de l'organisation et non l'inverse mais ceci au prix d'un plus gros travail d'intégration.

5.4.4 – Tests de validité de la configuration

L'implémentation de ces nouvelles fonctionnalités a nécessité l'ajout de tests complémentaires pour s'assurer la validité de la configuration générale avant sa sauvegarde en base pour améliorer la robustesse de l'application.

* Les modules U2F ou TOTP ne pouvant être activés avec UTOP simultanément, j'ai ajouté le test correspondant. Celui-ci vérifie la valeur des paramètres « $conf->{*2fAvtivation} » pour déterminer s'ils sont activés simultanément :

```
utotp => sub {
    return 1 unless ( $conf->{utotp2fActivation} );   my $w = "";
    foreach ( 'totp', 'u' ) {
    $w .= uc($_) . "2F is activated twice \n"
    if ( $conf->{ $_ . '2fActivation' } eq '1' ); }  return ( 1, ( $w ? $w : () ) ); }
```

* A présent, il est possible d'imposer l'enrôlement d'un second facteur à la connexion grâce à l'option « $conf->{sfRequired ». Pour ce faire, il faut au minimum un second facteur avec auto-enrôlement activé :

```
my $msg = '';  my $ok  = 0;
foreach (qw(u totp yubikey)) {
    $ok ||= $conf->{ $_ . '2fActivation'} && $conf->{ $_.'2fSelfRegistration'};
    last if ($ok);}
$ok   ||= $conf->{'utotp2fActivation'} && ( $conf→{'u2fSelfRegistration'}
    ||   $conf->{'totp2fSelfRegistration'} );
$msg = "A registrable module should be enabled to require 2FA" unless ($ok);
return ( 1, $msg )
```

* Pour éviter de bloquer l'accès au *Manager*, j'ai interdit la sauvegarde de la configuration si la durée de validité du jeton anti-CSRF est inférieure à 30 secondes :

```
formTimeout => sub {
      return 1 unless ( defined $conf->{formTimeout} );
      return ( 0, "XSRF form token TTL must be higher than 30s" )
      unless ( $conf->{formTimeout} > 30 );
      return ( 1, "XSRF form token TTL should not be higher than 2mn" )
      if ( $conf->{formTimeout} > 120 ); }
```

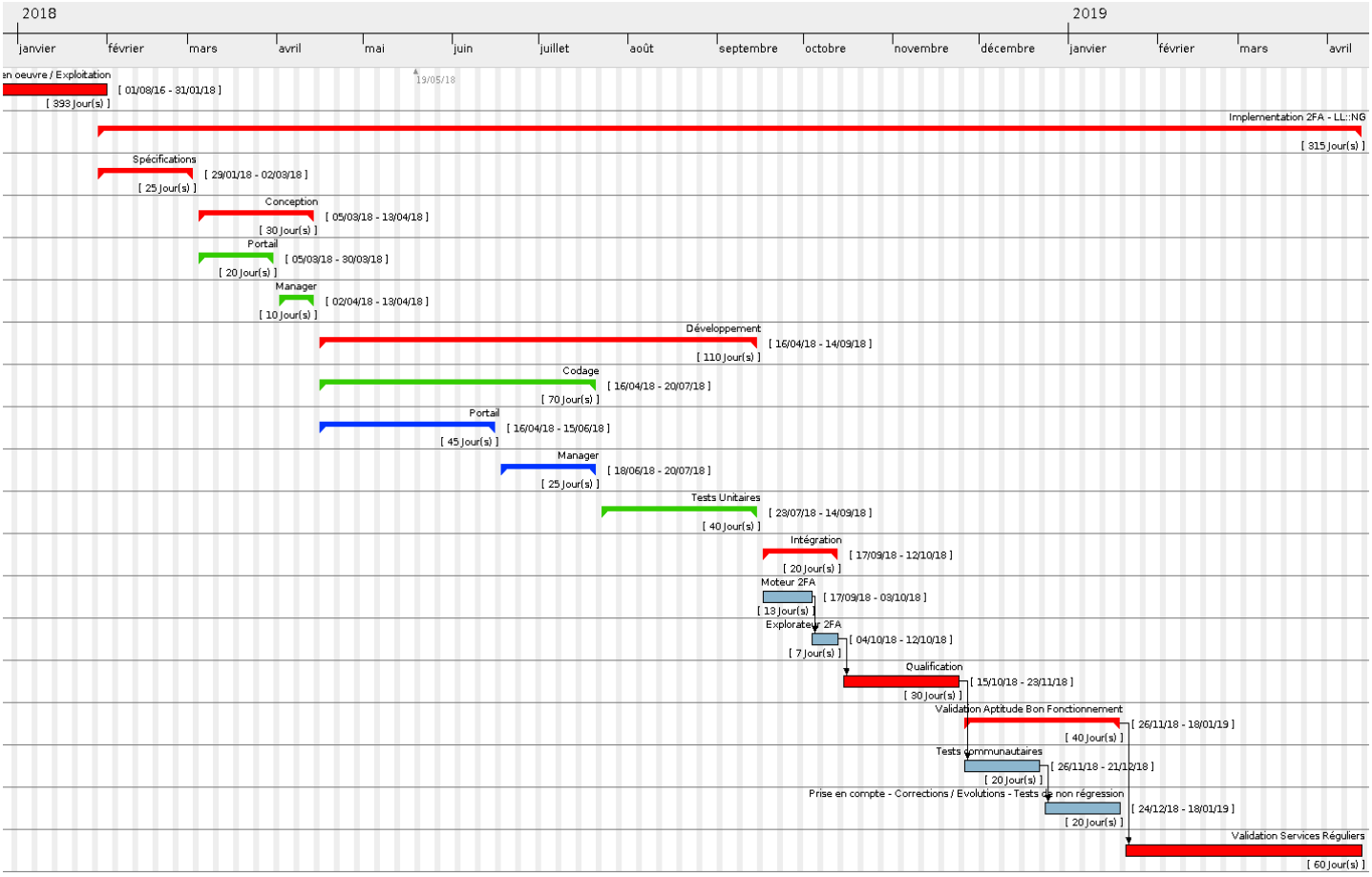

Figure 34 : Diagramme de Gantt de mon projet

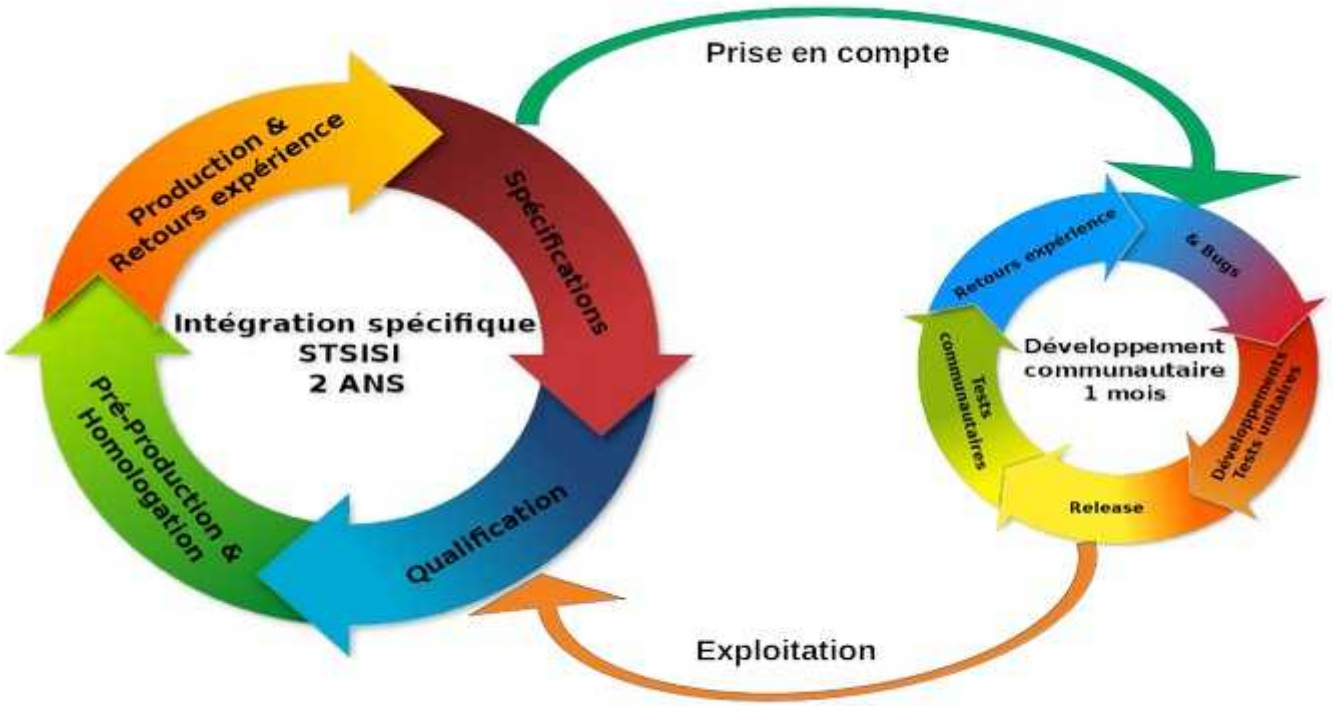

Figure 35 : Cycle de développement de LLNG

5.5 – Module Common::Session::REST

Pour supprimer le second facteur en session persistante, le *Manager* doit passer par le serveur REST de LLNG. Pour ce faire, j'ai écrit la fonction 'delete2F' fournie par le module « Common::Session::REST ». Tout, d'abord, je vérifie la méthode HTTP et la présence de l'Id de session. Ensuite, je charge la session persistante correspondante puis les seconds facteurs existants. Enfin, je conserve tous les seconds facteurs sauf celui devant être supprimé (type et epoch passés dans la requête) puis mets à jour la session en base :

```
my $mod = $self→getMod($req) ; my $id = $req→params('sessionId') ;
my $type   = $params→{type} ;   my $epoch = $params->{epoch}
$_2fDevices = eval { from_json( $session->data->{_2fDevices} );
push @keep, $element unless ( ( $element->{type} eq $type )
                              and ( $element->{epoch} eq $epoch ) );
$session->data->{_2fDevices} = to_json( \@keep );
$session->update( \%{ $session->data } );
```

5.6 – Synthèse

Cette implémentation définitive est le fruit de nombreux cycles courts de développement, d'analyse du service rendu puis de modifications pour prendre en compte les remarques ou demandes d'évolution. Cette manière de travailler peut tout à fait être assimilée à la méthode AGILE, utilisée par des entreprises de services du numérique.

En plus de tous les tests de non-régression ou de rejeu, cette version 2.0 de LLNG a fait l'objet de nombreux tests de bon fonctionnement (qualification). De plus, la mise à disposition en temps réel via la plateforme GitLab des dernières versions alpha puis bêta, nous a permis de faire tester les évolutions par des intégrateurs externes sur différentes plateformes (VABF) et de répondre aux différents besoins des utilisateurs ou intégrateurs en ajoutant de nouvelles fonctionnalités comme le « monitoring » par MRTG ou le support du format YAML remontées via la forge GitLab (cf figure 34). Le diagramme de Gantt (figure 34) est une représentation non chronologique des phases du projet. J'ai travaillé parallèlement sur les différents modules (*Portail*, *Manager*).

Parallèlement au développement communautaire mensuel, le STSISI a mis en place son propre cycle de déploiement de LemonLDAP::NG dont la période est de deux années. Celui-ci est constitué de la qualification d'une version de LLNG réputée stable, suivie par une mise en pré-production permettant l'homologation de l'infrastructure du SSO. Après homologation, la version est mise en production et les éventuels « bugs » sont signalés sur la plateforme de développement GitLab pour correction par la communauté (cf figure 35).

CONCLUSION

Mon intégration à l'équipe de développement de LemonLDAP::NG et la réalisation de ce mémoire ont été des étapes très enrichissantes à plusieurs points de vue.

Sur le plan personnel tout d'abord, l'apprentissage de toutes les technologies utilisées puis la compréhension de l'architecture et du fonctionnement de LLNG m'ont demandé beaucoup de travail de recherche, de documentation et d'investissement personnel. J'ai passé beaucoup de temps à faire des recherches sur Internet, à consulter des ouvrages, à analyser ou modifier le code pour en comprendre le fonctionnement et poser beaucoup de questions tant à mon tuteur qu'à mes collègues. Comme j'aime à le rappeler : « le ticket d'entrée est élevé ». Ce n'est pas tant la complexité des technologies ou concepts utilisés mais le nombre et la très grande quantité de connaissances ou de notions à acquérir et maîtriser. Les très bons rapports entretenus avec les autres membres de l'équipe toujours disponibles, en particulier mon tuteur qui a toujours eu la patience et pris le temps de répondre à mes questions, m'ont permis d'apprendre, progresser et avancer.

D'un point de de vue Sécurité des Systèmes d'Information, j'ai eu à tenir compte et implémenter un panel très large de contre-mesures pour lutter contre les attaques courantes qui sévissent de plus en plus sur Internet. Aujourd'hui, la sécurité est un enjeu majeur pour les organisations ainsi que pour l'ensemble des acteurs qui l'entourent. En plus de ces aspects purement sécuritaires, il m'a fallu également travailler pour garantir la disponibilité, l'intégrité et la confidentialité de LemonLDAP::NG en plus des classiques concepts 3A (Authentification, Autorisation et traces d'Accès).

Du point de vue de l'implémentation et des technologies, les développements les plus complexes furent les modules U2F et l'explorateur 2ndFA en *AngularJS*. Il m'a fallu comprendre et maîtriser les interactions entre les différents concepts, langages, outils et normes. La structure modulaire de LemonLDAP::NG permet de ne charger que les modules nécessaires pour en optimiser le fonctionnement général et de développer de nouvelles fonctionnalités comme le support des futurs seconds facteurs sans avoir à modifier le code principal mais au prix d'une difficulté de conception accrue. Toutes ces technologies s'imbriquent et dépendent les unes des autres ce qui impose d'aborder le problème de façon globale, d'avoir une approche générale du concept. Il n'est pas possible de concevoir séparément le côté serveur avec ses modules et la partie cliente avec son code *JavaScript* et ses *templates*. En outre, une très bonne connaissance des réseaux, de la sécurité et du protocole HTTP est indispensable. Le modèle Client / Serveur impose de mettre en place des artifices (OTT, cookies, données spécifiques dans les requêtes) pour conserver le contexte d'une requête à l'autre. J'ai apprécié la partie implémentation du TOTP pour son approche un peu plus mathématique qui m'a obligé à comprendre la formule de génération des codes. La mise en œuvre de l'authentification à double facteur basée sur les modules Yubikey, REST et Ext2F n'a pas posé de difficulté particulière. J'ai apprécié pouvoir également développer côté *Manager* ce qui change du *Portail*. J'ai dû aborder les choses différemment. Partir d'un module existant, l'épurer et ajouter des fonctionnalités au fur et à mesure. De plus, j'ai pris plaisir à concevoir les différents tests de configuration nécessaires à la protection du système ainsi que les

tests de non-régression qui, de surcroît, ont parfois révélé des cas d’erreurs auxquels je n’avais pas pensé et dont j'ai tenu compte ensuite.

La partie la plus complexe à concevoir a été sans aucun doute le moteur 2FA. Celui-ci gère le chargement des modules actifs et les redirections des utilisateurs vers les pages de connexion ou d’enrôlement. Ce moteur m’a demandé un gros travail de réflexion pour en comprendre le fonctionnement puis le modifier notamment pour la mise en œuvre de l’option ‘2fRequired’ et écrire les différents tests de non-régressions des différents modules d’authentification et d’enrôlement.

L’écriture de ces tests de non-régression m’a réellement permis de progresser et comprendre le fonctionnement de LLNG. De plus, la fourniture des tests correspondant aux nouvelles fonctionnalité m’a permis de découvrir le concept de l'intégration continue. Celle-ci consiste à vérifier à chaque modification de code source que le résultat des modifications ne produit pas de régression dans l'application développée. Le principal but de cette pratique est de détecter les problèmes d'intégration au plus tôt lors du développement. De plus, elle permet d'automatiser l'exécution des suites de tests et de voir l'évolution du développement du logiciel. L'intégration continue est de plus en plus utilisée afin d'améliorer la qualité du code et du produit final.

Après huit mois de travail, il m’est intéressant de constater que le temps passé à « coder » est très faible comparativement au temps nécessaire d’analyse et de compréhension. Au début, cette situation était même frustrante pour moi car j’avais l’impression de ne pas avancer. Mais, tous ces efforts n'ont pas été vains ou inutiles et m'ont permis d'aboutir. Je suis particulièrement satisfait et fier d'avoir apporté ma contribution à cette version 2.0 de LLNG très attendue notamment pour le support de l'authentification à double facteur nécessaire pour le pour le SSO de la gendarmerie nationale exposé sur internet. La version 2.0 est sortie le 01 décembre 2018.

En effet, cette version va être déployée en production début 2019 au sein du Ministère de l’Intérieur. La Gendarmerie nationale dispose, avec « Proxyma », de l’un des plus gros déploiements de LemonLDAP::NG avec 120 000 utilisateurs pour environ 25 millions d’accès par jour et 350 applications protégées !!! L’arrivée d’applications en mode *WebSocket* oblige à passer de Apache à Nginx, de monter de version de LemonLDAP::NG pour utiliser la plus récente (version 2.0) et migrer vers des bases de données PostGreSQL. Cette évolution concernera l’ensemble des SSO gérés par le STSISI. Avec l’Adjudant-chef Rosier, nous avons préparé et installé les plateformes de Production et de Développement puis fourni les scripts de création des bases de données correspondantes au STIG (exploitation).

L’ensemble des tests de non-régression et d’intégration continue comme l’évaluation en Pré-Production de la version 1.9.17 de LLNG témoignent d’un très haut niveau de robustesse. Quant aux tests de charge, les applications seront migrées au fur et à mesure avec retour arrière possible sur l’architecture actuelle. LLNG étant « scalable », des machines virtuelles supplémentaires peuvent être ajoutées si besoin. Parallèlement à ce déploiement en Production au Ministère de l’Intérieur, de nombreuses entreprises ou organisations en France comme à l’étranger utilisent déjà,

au vu des nombreuses requêtes et commentaires reçus via les moyens de communication de la communauté, la solution LemonLDAP::NG comme Nantes Métropole, Orange Business Service, la région Basse Normandie ou les universités de Limoges ou Namur ainsi que la plupart des ministères français. Une administration envisage même de quitter « OpenAM » pour migrer vers LLNG pour réduire ses coûts de licence…

Pour terminer, le développement de LemonLDAP::NG respecte intégralement les cycles et protocoles mis en place par les grands éditeurs de logiciels professionnels. Cette version 2.0 est l'aboutissement de quinze années de travail et d'évolutions. LLNG est une solution très fiable, robuste et performante. Les prochaines étapes sont la ré-écriture complète du *Manager* pour le faire évoluer vers une conception calquée sur celle du *Portail* , la fin de l'explorateur sous forme d'arbre, inadapté aux équipements mobiles (tablette ou smartphones) et le remplacement *d'AngularJS* dont le support de la version 1.7 LTS se terminera en 2022, très probablement par V*ue.js*. S'agissant du *Portail*, l'abandon progressif de « Bootstrap » pour migrer vers « Materialize » est prévu prochainement. J'envisage également d'ajouter à l'explorateur de sessions un moteur de recherche calqué sur l'explorateur de sessions 2FA et j'aimerais créer un moteur permettant de rechercher des clefs dans la configuration.

A moyen terme, avec l'arrivée de solutions « clef en main » comme « KeyCloak », il nous faut travailler sur la simplification ou l'automatisation d'une installation de base adaptée pour de petites organisations ne nécessitant pas forcément de la haute disponibilité ou des performances élevées.

De plus, la prochaine fonctionnalité à intégrer dans LemonLDAP::NG sera la prise en charge de la nouvelle API libre d'authentification portée par la FIDO Alliance et le W3C baptisée « WebAuthn ». Elle a pour but de trouver une alternative plus sûre aux traditionnels mots de passe. Une API qui serait à la fois plus sécurisée, mais aussi libre, de façon à ce que n'importe quelle société, mais aussi développeur, puisse l'utiliser pour son site web ou ses services en ligne. Elle est d'ores et déjà intégrée dans la dernière version de *Firefox* et devrait arriver d'ici quelques mois sur *Chrome*. Le principe de ce nouveau protocole de sécurité repose sur un « système authentificateur » physique. Il peut s'agir aussi bien d'une clef U2F, du capteur d'empreinte digitale d'un smartphone ou encore d'un système de reconnaissance faciale d'une caméra. Pour se connecter à un site web qui supportera « WebAuthn », il suffira alors simplement d'utiliser l'un ou l'autre des seconds facteurs disponibles. Outre l'absence de mot de passe, « WebAuthn » possède de nombreux avantages au niveau de la sécurité. Le site web ou service ne stockera ainsi aucun mot de passe. Il se contentera simplement de stocker une clé de sécurité publique envoyée par l'utilisateur et au moment de se connecter, son smartphone enverra au site un message d'authentification signé avec une clé privée, stockée uniquement sur son téléphone (cf. figure 36). Pirater le compte d'un utilisateur devrait donc s'avérer bien plus compliqué que de lui soutirer son mot de passe via un système de « phishing ». Il faudra pour cela accéder à l'appareil « authentificateur », puis trouver sa clef de sécurité privée (« WebAuthn », 2018).

Un protocole de sécurité plus facile à utiliser côté utilisateur, certes, mais qui nécessite qu'il soit systématiquement équipé d'un appareil physique pour s'authentifier. De plus, le mot de passe n'étant plus demandé, il ne s'agit plus d'un second facteur mais d'une authentification faible.

A plus long terme, je vais poursuivre le développement de LemonLDAP::NG et travailler sur une version 3.0 peut-être écrite en *Javascript* pour *Node.js*. En outre, j'envisage d'intégrer la communauté Debian et participer à la maintenance des paquets *Perl* ou *JavaScript*.

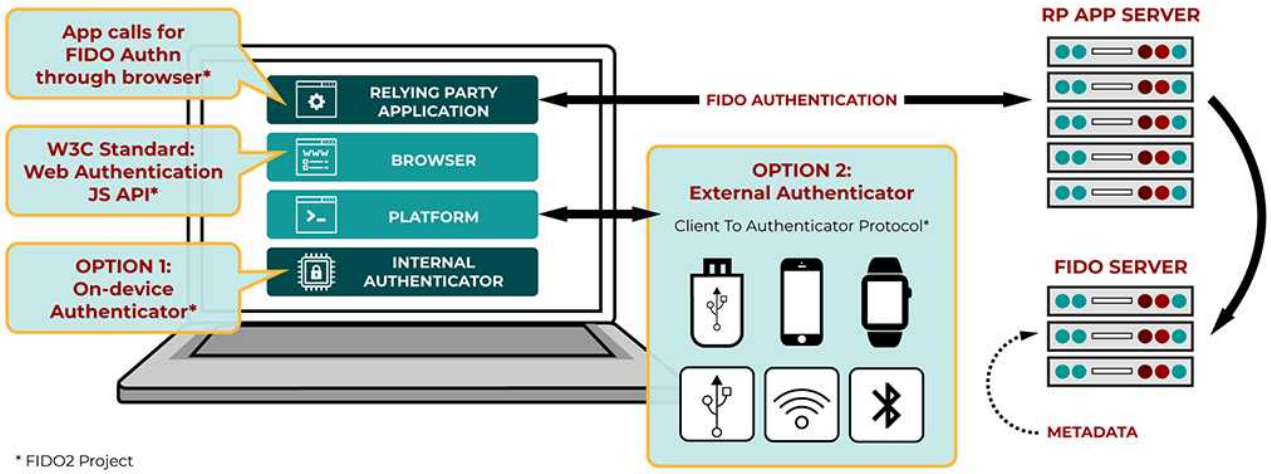

Figure 36 : Norme WebAuthn FIDO-2

ANNEXES

Annexe 1 : Fiche technique NeoWave

Annexe 2 : Protocole & Cinématique détaillée U2F

Annexe 3 : Norme U2F – API JavaScript

BIBLIOGRAPHIE

AJAX. (s. d.). Consulté à l'adresse http://projet.eu.org/pedago/sin/1ere/5-AJAX.pdf

Ajax (informatique). (2018). In *Wikipédia*. Consulté à l'adresse https://fr.wikipedia.org/w/index.php?title=Ajax_(informatique)&oldid=150775204

Allain, G. (2011). *Jquery*. MA Editions.

AngularJS. (2018). In *Wikipédia*. Consulté à l'adresse https://fr.wikipedia.org/w/index.php?title=AngularJS&oldid=150691254

AngularJS — Superheroic JavaScript MVW Framework. (s. d.). Consulté 14 mars 2018, à l'adresse https://angularjs.org/

Apache HTTP Server. (2018). In *Wikipédia*. Consulté à l'adresse https://fr.wikipedia.org/w/index.php?title=Apache_HTTP_Server&oldid=148265587

Apache::Session - search.cpan.org. (s. d.). Consulté 10 juin 2018, à l'adresse http://search.cpan.org/~chorny/Apache-Session-1.93/lib/Apache/Session.pm

Application web monopage. (2018). In *Wikipédia*. Consulté à l'adresse https://fr.wikipedia.org/w/index.php?title=Application_web_monopage&oldid=148626533

Attaque par déni de service. (2018). In *Wikipédia*. Consulté à l'adresse https://fr.wikipedia.org/w/index.php?title=Attaque_par_d%C3%A9ni_de_service&oldid=150838665

Authentification. (2017). In *Wikipédia*. Consulté à l'adresse https://fr.wikipedia.org/w/index.php?title=Authentification&oldid=142160576

Authentification forte. (2018a). In *Wikipédia*. Consulté à l'adresse https://fr.wikipedia.org/w/index.php?title=Authentification_forte&oldid=146863959

Authentification forte. (2018b). In *Wikipédia*. Consulté à l'adresse https://fr.wikipedia.org/w/index.php?title=Authentification_forte&oldid=150855514

Authentification unique. (2018). In *Wikipédia*. Consulté à l'adresse https://fr.wikipedia.org/w/index.php?title=Authentification_unique&oldid=145368582

Authen::U2F::Tester - FIDO/U2F Authentication Test Client. (s. d.). Consulté 8 août 2018, à l'adresse https://metacpan.org/pod/Authen::U2F::Tester

Balfanz, D., Birgisson, A., & Lang, J. (s. d.). FIDO U2F JavaScript API, 7.

Balfanz et al. - FIDO U2F JavaScript API.pdf. (s. d.). Consulté à l'adresse https://fidoalliance.org/specs/fido-u2f-v1.2-ps-20170411/fido-u2f-javascript-api-v1.2-ps-20170411.pdf

BENGLIA_BENHAMMOUDA.pdf. (s. d.).

Bootstrap (framework). (2018). In *Wikipédia*. Consulté à l'adresse https://fr.wikipedia.org/w/index.php?title=Bootstrap_(framework)&oldid=148248680

Bruchez, R. (2013). *Les bases de données NoSQL: Comprendre et mettre en oeuvre*. Editions Eyrolles.

Burnham, T. (2015). *CoffeeScript: Accelerated JavaScript Development*. Pragmatic Bookshelf.

CELEX_32015R1502_FR_TXT.pdf. (s. d.).

Certificat électronique. (2018). In *Wikipédia*. Consulté à l'adresse https://fr.wikipedia.org/w/index.php?title=Certificat_%C3%A9lectronique&oldid=147724522

cez40. (2014, mai 25). HOTP et TOTP. Consulté 2 août 2018, à l'adresse https://blog.cybercod.com/hotp-et-totp/

Clément-Oudot_LDAP. (s. d.).

CoffeeScript. (2018). In *Wikipédia*. Consulté à l'adresse https://fr.wikipedia.org/w/index.php?title=CoffeeScript&oldid=144371352

Common Development and Distribution License 1.0. (s. d.). Consulté 23 mai 2018, à l'adresse https://opensource.org/licenses/CDDL-1.0

Common Gateway Interface. (2016). In *Wikipédia*. Consulté à l'adresse https://fr.wikipedia.org/w/index.php?title=Common_Gateway_Interface&oldid=133012809

Cross-site request forgery. (2018). In *Wikipédia*. Consulté à l'adresse https://fr.wikipedia.org/w/index.php?title=Cross-site_request_forgery&oldid=147972406

Cross-site scripting. (2018). In *Wikipédia*. Consulté à l'adresse https://fr.wikipedia.org/w/index.php?title=Cross-site_scripting&oldid=149028333

Crypt::U2F::Server::Simple. (s. d.). Consulté 4 août 2018, à l'adresse https://metacpan.org/pod/Crypt::U2F::Server::Simple

Data::Dumper - perldoc.perl.org. (s. d.). Consulté 21 août 2018, à l'adresse https://perldoc.perl.org/Data/Dumper.html

DBI - search.cpan.org. (s. d.). Consulté 10 juin 2018, à l'adresse http://search.cpan.org/dist/DBI/DBI.pm#General

Demande de signature de certificat. (2018). In *Wikipédia*. Consulté à l'adresse https://fr.wikipedia.org/w/index.php?title=Demande_de_signature_de_certificat&oldid=147622089

Des-applications-ultra-rapides-avec-node-js.pdf. (s. d.). Consulté à l'adresse https://data.brains-master.com/pdf/762576-des-applications-ultra-rapides-avec-node-js.pdf

Document Object Model. (2018). In *Wikipédia*. Consulté à l'adresse https://fr.wikipedia.org/w/index.php?title=Document_Object_Model&oldid=150078334

documentation:2.0:performances [LemonLDAP::NG]. (s. d.). Consulté 25 mai 2018, à l'adresse https://lemonldap-ng.org/documentation/2.0/performances?&#apachesession_performances

documentation:2.0:psgi [LemonLDAP::NG]. (s. d.). Consulté 25 mai 2018, à l'adresse https://lemonldap-ng.org/documentation/2.0/psgi

eXtensible Access Control Markup Language (XACML) Version 3.0. (s. d.). Consulté 23 mai 2018, à l'adresse http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.html

FastCGI. (2017). In *Wikipédia*. Consulté à l'adresse https://fr.wikipedia.org/w/index.php?title=FastCGI&oldid=141745409

Feed 32up, 31 May 2018 Ruth Holloway. (s. d.). Mouse: A time-saving object system for smaller projects. Consulté 14 août 2018, à l'adresse https://opensource.com/article/18/5/mouse-oo-class-tool-perl

Feuilles de style en cascade (CSS). (2018). In *Wikipédia*. Consulté à l'adresse https://fr.wikipedia.org/w/index.php?title=Feuilles_de_style_en_cascade&oldid=150643916

Générateur de nombres pseudo-aléatoires. (2018). In *Wikipédia*. Consulté à l'adresse https://fr.wikipedia.org/w/index.php?title=G%C3%A9n%C3%A9rateur_de_nombres_pseudo-al%C3%A9atoires&oldid=148218505

git. (2018). In *Wikipédia*. Consulté à l'adresse https://fr.wikipedia.org/w/index.php?title=Git&oldid=150831557

GNU. (2017). In *Wikipédia*. Consulté à l'adresse https://fr.wikipedia.org/w/index.php?title=GNU&oldid=143519892

Gray. (2018). *Two Factor Authentication Perl code*. Perl. Consulté à l'adresse https://github.com/j256/perl-two-factor-auth (Original work published 2015)

HTTPS. (2018). In *Wikipedia*. Consulté à l'adresse https://en.wikipedia.org/w/index.php?title=HTTPS&oldid=845589241

Human interface device. (2018). In *Wikipedia*. Consulté à l'adresse https://en.wikipedia.org/w/index.php?title=Human_interface_device&oldid=839118503

Hypertext Transfer Protocol (HTTP). (2018). In *Wikipédia*. Consulté à l'adresse https://fr.wikipedia.org/w/index.php?title=Hypertext_Transfer_Protocol&oldid=148330944

iFRAME, fonctionnement et protection – CERT-FR. (s. d.). Consulté 24 septembre 2018, à l'adresse https://www.cert.ssi.gouv.fr/information/CERTA-2008-INF-001/

Jasmine (JavaScript testing framework). (2018). In *Wikipedia*. Consulté à l'adresse https://en.wikipedia.org/w/index.php?title=Jasmine_(JavaScript_testing_framework)&oldid=830762474

jQuery. (2018). In *Wikipédia*. Consulté à l'adresse https://fr.wikipedia.org/w/index.php?title=JQuery&oldid=150642243

JSON (JavaScript Object Notation) encoder/decoder. (s. d.). Consulté 26 août 2018, à l'adresse https://metacpan.org/pod/JSON#to_json

Keyed-hash message authentication code. (2018). In *Wikipédia*. Consulté à l'adresse https://fr.wikipedia.org/w/index.php?title=Keyed-hash_message_authentication_code&oldid=148642467

La signature numérique - Fonctionnement. (s. d.). Consulté à l'adresse http://www-igm.univ-mlv.fr/~dr/XPOSE2006/depail/fonctionnement.html

Lancer des processus à partir de Perl. (s. d.). Consulté 20 août 2018, à l'adresse http://articles.mongueurs.net/magazines/linuxmag55.html

Lasso - Free Liberty Alliance Single Sign On. (s. d.). Consulté 21 juin 2018, à l'adresse http://lasso.entrouvert.org/

LDAP Data Interchange Format. (2017). In *Wikipédia*. Consulté à l'adresse https://fr.wikipedia.org/w/index.php?title=LDAP_Data_Interchange_Format&oldid=134809105

LemonLDAP::NG. (2017). In *Wikipédia*. Consulté à l'adresse https://fr.wikipedia.org/w/index.php?title=LemonLDAP::NG&oldid=133218398

LemonLDAP::NG - Open Source Web Single Sign On. (s. d.). Consulté 15 mars 2018, à l'adresse https://lemonldap-ng.org/welcome/

Les attaques de type « cross-site request forgery » – CERT-FR. (s. d.). Consulté 11 août 2018, à l'adresse https://www.cert.ssi.gouv.fr/information/CERTA-2008-INF-003/

Les tests en Perl - Présentation et modules standards. (s. d.). Consulté 21 août 2018, à l'adresse http://articles.mongueurs.net/magazines/linuxmag88.html

Lets Encrypt. (2018). In *Wikipédia*. Consulté à l'adresse https://fr.wikipedia.org/w/index.php?title=Lets_Encrypt&oldid=150604558

Licence GNU. (s. d.). Consulté 4 août 2018, à l'adresse http://www.gnu.org/licenses/gpl.html

Liefooghe, V. (s. d.). Gestion des mots de passe avec OpenLDAP |. Consulté 10 juin 2018, à l'adresse https://www.vincentliefooghe.net/content/gestion-des-mots-passe-avec-openldap

Lieuze, F. (s. d.). Documentez vos modules Perl avec Pod. Consulté 6 août 2018, à l'adresse http://woufeil.developpez.com/tutoriels/perl/pod/

Lua. (2018). In *Wikipédia*. Consulté à l'adresse https://fr.wikipedia.org/w/index.php?title=Lua&oldid=149305926

Memcached. (2017). In *Wikipédia*. Consulté à l'adresse https://fr.wikipedia.org/w/index.php?title=Memcached&oldid=142145344

Mercer, A. (2018). *qrious: JavaScript library for QR code generation*. JavaScript. Consulté à l'adresse https://github.com/neocotic/qrious (Original work published 2011)

Modèle-vue-contrôleur. (2018). In *Wikipédia*. Consulté à l'adresse https://fr.wikipedia.org/w/index.php?title=Mod%C3%A8le-vue-contr%C3%B4leur&oldid=144497730

mod_perl. (2018). In *Wikipédia*. Consulté à l'adresse https://fr.wikipedia.org/w/index.php?title=Mod_perl&oldid=146404855

Mouse - Moose minus the antlers. (s. d.). Consulté 14 août 2018, à l'adresse https://metacpan.org/pod/Mouse

nginx. (2018). In *Wikipédia*. Consulté à l'adresse https://fr.wikipedia.org/w/index.php?title=Nginx&oldid=147140635

Node.js. (2018). In *Wikipédia*. Consulté à l'adresse https://fr.wikipedia.org/w/index.php?title=Node.js&oldid=149036611

Noyau Linux. (2018). In *Wikipédia*. Consulté à l'adresse https://fr.wikipedia.org/w/index.php?title=Noyau_Linux&oldid=150534982

Ollivier, S., & Gury, P.-A. (2015). *AngularJS: Développez aujourd'hui les applications web de demain*. Editions ENI.

OpenAM. (2018). In *Wikipedia*. Consulté à l'adresse https://en.wikipedia.org/w/index.php?title=OpenAM&oldid=838646973

OpenAM - Authorization Guide. (s. d.). Consulté 23 mai 2018, à l'adresse https://backstage.forgerock.com/docs/am/6/authorization-guide/index.html#chap-authz-implementation

OpenAM - Quick Start Guide. (s. d.). Consulté 23 mai 2018, à l'adresse https://backstage.forgerock.com/docs/am/6/quick-start-guide/

OpenID Connect. (2018). In *Wikipédia*. Consulté à l'adresse https://fr.wikipedia.org/w/index.php?title=OpenID_Connect&oldid=146186765

Panda, S. (2014). *AngularJS: Novice to Ninja*. Sebastopol, US: O'Reilly.

PerlTidy. (2018). In *Wikipedia*. Consulté à l'adresse https://en.wikipedia.org/w/index.php?title=PerlTidy&oldid=836425905

Protractor - end-to-end testing for AngularJS. (s. d.). Consulté 11 août 2018, à l'adresse https://www.protractortest.org/#/

Proxy inverse. (2017). In *Wikipédia*. Consulté à l'adresse https://fr.wikipedia.org/w/index.php?title=Proxy_inverse&oldid=139299591

PSGI. (2018). In *Wikipédia*. Consulté à l'adresse https://fr.wikipedia.org/w/index.php?title=PSGI&oldid=147322559

PSGI/Plack - Perl Superglue for Web Frameworks and Web Servers. (s. d.). Consulté 2 juillet 2018, à l'adresse https://plackperl.org/

Qu'est-ce qu'une api REST ? (s. d.). Consulté 4 août 2018, à l'adresse https://www.supinfo.com/articles/single/5642-qu-est-ce-qu-une-api-rest-restful

Robinson <drtr@apache.org>, D. (s. d.). The Common Gateway Interface (CGI) Version 1.1. Consulté 23 juin 2018, à l'adresse https://tools.ietf.org/html/rfc3875#section-6.3

SAML et fédération d'identité. (s. d.). Consulté 14 mai 2018, à l'adresse https://www.journaldunet.com/developpeur/xml/analyse/la-federation-d-identite-au-travers-de-saml.shtml

Service des technologies et des systèmes d'information de la Sécurité intérieure. (s. d.). Consulté 22 juin 2018, à l'adresse https://fr.wikipedia.org/wiki/Service_des_technologies_et_des_syst%C3%A8mes_d%27information_de_la_S%C3%A9curit%C3%A9_int%C3%A9rieure

Service du traitement de l'information de la Gendarmerie. (2015). In *Wikipédia*. Consulté à l'adresse https://fr.wikipedia.org/w/index.php?title=Service_du_traitement_de_l%27information_de_la_Gendarmerie&oldid=113353407

system - perldoc.perl.org. (s. d.). Consulté 4 août 2018, à l'adresse https://perldoc.perl.org/functions/system.html

Test Anything Protocol. (2018). In *Wikipedia*. Consulté à l'adresse https://en.wikipedia.org/w/index.php?title=Test_Anything_Protocol&oldid=847601170

Test::Harness - Run Perl standard test scripts with statistics. (s. d.). Consulté 21 août 2018, à l'adresse https://metacpan.org/pod/Test::Harness

Test::More - perldoc.perl.org. (s. d.). Consulté 21 août 2018, à l'adresse https://perldoc.perl.org/Test/More.html

Test::Pod - check for POD errors in files - metacpan.org. (s. d.). Consulté 22 août 2018, à l'adresse https://metacpan.org/pod/Test::Pod

Théorème CAP. (2018). In *Wikipédia*. Consulté à l'adresse https://fr.wikipedia.org/w/index.php?title=Th%C3%A9or%C3%A8me_CAP&oldid=144541442

Tous les contenus taggés avec lemonldap - LinuxFr.org. (s. d.). Consulté 9 mai 2018, à l'adresse https://linuxfr.org/tags/lemonldap/public

Type de médias. (2018). In *Wikipédia*. Consulté à l'adresse https://fr.wikipedia.org/w/index.php?title=Type_de_m%C3%A9dias&oldid=149203873

Universal 2nd Factor (U2F) Overview.pdf. (s. d.). Consulté à l'adresse https://fidoalliance.org/specs/fido-u2f-v1.2-ps-20170411/fido-u2f-overview-v1.2-ps-20170411.pdf

WebAuthn : le futur du web sans mot de passe se rapproche. (2018, avril 12). Consulté 26 septembre 2018, à l'adresse https://www.journaldugeek.com/2018/04/12/webauthn-futur-web-de-passe-se-rapproche/

Write-Ahead Logging (WAL). (s. d.). Consulté 22 juin 2018, à l'adresse https://docs.postgresql.fr/8.0/wal.html

Xavier Guimard / Apache-Session-Browseable. (s. d.). Consulté 10 juin 2018, à l'adresse http://search.cpan.org/dist/Apache-Session-Browseable/

Yubikey de Yubico. (2017, avril 4). Consulté 29 juillet 2018, à l'adresse https://blog.arnaudminable.net/yubikey-de-yubico-cest-plus-rigolo/

LISTE DES FIGUES

LISTE DES TABLEAUX

RÉSUMÉ & MOTS-CLEFS

RÉSUMÉ :

De nos jours, nombre d'entreprises et d'organisations fournissent à leurs employés ou clients des services ou des applications web. Ceci oblige les utilisateurs à mémoriser de plus en plus de mots de passe parfois complexes. Une solution est l'utilisation de l'authentification unique (SSO) qui permet à un utilisateur d'avoir accès à l'ensemble des ressources réseaux auxquelles il a droit en n'ayant à s'authentifier qu'une seule fois. L'utilisateur disposera d'une session SSO valide pour l'ensemble des applications ou ressources protégées. De plus, un SSO améliore la sécurité, facilite la gestion des comptes utilisateurs et l'ergonomie du système d'information.

Or, l'authentification par couple identifiant – mot de passe est devenue obsolète et trop vulnérable aux attaques. Pour en améliorer la sécurité, elle peut être associée à l'utilisation d'un second facteur d'authentification (2FA). L'authentification à double facteur consiste à demander à l'utilisateur, après s'être authentifié, de fournir un élément qu'il possède, qu'il connaît ou qui le caractérise. L'utilisation d'un 2FA permet d'accroître significativement la robustesse de l'authentification.

Dans un premier temps, j'introduis les notions d'Authentification – Autorisation et d'Accès (AAA), le modèle SSO en comparant les solutions LemonLDAP::NG et OpenAM, les niveaux d'authentification eIDAS et quelques rappels sur le protocole HTTP. Puis, je présente le WebSSO LemonLDAP::NG (LLNG) et en détaille l'architecture ainsi que le fonctionnement. Je termine par la description de l'implémentation de l'authentification à double facteur dans LLNG, notamment le moteur permettant de gérer et proposer différents types et propose quelques fonctionnalités à implémenter ou à améliorer dans la version 3.0 de LemonLDAP::NG

MOTS-CLEFS :

Implémentation de l'authentification à double facteur dans la version 2.0 de la solution SSO LemonLDAP::NG :

Sécurité – WebSSO – Authentification unique – Autorisation – Contrôle d'Accès – Second facteur – TOTP – U2F – LemonLDAP::NG – Identité

ABSTRACT :

Nowadays, more and more corporations and organisations provide on-line applications or services to their customers and staff. Memorising complex passwords for each of those systems is a challenge many users face every day. Single Sign On, also known as SSO, may be the solution to this issue. It allows users to have access to multiple applications by signing in using only one account to different systems and resources. It also has other important features in regards to simple management, security, ease of use and seamlessness.

In case of complex SSO architecture, multiple usernames and passwords not only cause frustration but are also a serious security weakness. Basic username and password combinations are becoming more and more vulnerable to theft. To manage this risk, you can use security-enhancing methods like second factor authentication (2FA). 2FA works by confirming a user's claimed identity during login by running an extra verification check on the user attempting to log in with their username and password. With 2FA, a user will enter their username and password as normal. But, to prove it's really the account owner trying to log in, the user will then have to provide the "second factor", which can be based on the following : something you know, something you have or something you are. Enabling just one of these additional factors on top of the usual login credentials will significantly improve the security of your account.

In this paper, I first introduce Authentication – Authorization and Accounting (AAA) concepts, SSO in general terms by comparing LemonLDAP::NG and Open AM solutions, HTTP protocol and eIDAS authentication levels. Afterwards, I describe LemonLDAP::NG WebSSO solution and explain how it works. Finally, I present my LLNG 2FA implementation. I developed a 2FA engine able to, depending on context, to provide 2FA services or manage different devices. To conclude I propose some features to improve or include in LLNG 3.0.

KEYWORDS :

Second factor authentication implementation in webSSO LemonLDAP::NG 2.0 :

Single Sign-On – Authentication – Authorization – Access Control – Second Factor – TOTP U2F – LemonLDAP::NG – Security – Identity