Université de Caen Normandie
UFR des Sciences
Département Informatique

2ème année de master informatique
Spécialité Sécurité des Systèmes d'Information

Préindustrialisation d'un module collaboratif de lutte contre le phishing : sécurité et scalabilité

Stage

UFR des Sciences Caen Normandie

Simon DRIEUX

Encadrant: Emmanuel GIGUET

Jury: Marc SPANIOL & Gaétan RICHARD

Année universitaire : 2021 / 2022

Soutenu le Lundi 5 Septembre 2022

Table des matières

1 Introduction

Le phishing est une menace récurrente dans notre quotidien. Jeu du chat et de la souris, les mails frauduleux deviennent de plus en plus sophistiqués, authentiques et plus difficiles à détecter. De plus en plus de personnes tombent dans les pièges tirés par les personnes malveillantes. Des outils d'analyse sont développés et constamment mis à jour pour combattre le phishing. Aujourd'hui, il y a plusieurs méthodes pour protéger les utilisateurs : analyse des mails, détection de liens suspects, pare-feu, signalements... À l'Université de Caen, la D.S.I. (Direction des Systèmes d'Information) est souvent ciblée par des attaques de phishing pour obtenir des informations sensibles sur le personnel et les étudiants, pour la revente ou pour se faire passer pour eux. La plateforme de mail Zimbra, utilisée par l'université, ne dispose pas d'outils de détection ou de signalement de mails.

Dans ce rapport, nous expliquerons les mises en oeuvre pour améliorer Hamsa en terme de performance, de sécurité et de fiabilité. Ce rapport explique les problèmes existants, ses multiples pistes d'améliorations et les solutions apportées. Nous soulèverons aussi les problèmes rencontrés lors du stage, pendant le développement.

Ce rapport est distingué en 7 parties sur le travail réalisé, des parties sont dédiés à la présentation de l'entreprise et de mon cadre de travail :

— Un résumé du rapport de stage en français et en anglais
— Présentation du laboratoire GREY'C et son environnement de travail
— Le sujet du stage, fonctionnement de l'application, architecutre initiale et une analyse critique.
— Les nouvelles architectures pour l'intégration d'un système d'authentification par CAS ou par proxy de Zimbra
— Modifications et améliorations apportées sur l'application Hamsa : Comment transformer un prototype en une application stable capable de gérer du trafic élevé.
— L'application Zimbra et son Zimlet : Notre point de départ des communications, les données et échanges nécessaires pour minimiser le trafic tout en protégeant l'utilisateur.
— Les résultats de ces changements et retours de la D.S.I. sur le projet

Remerciements

Je tiens à remercier toutes les personnes qui ont contribué, de loin comme de près, à la réussite de mon stage et qui m'ont aidé tout au long.

J'adresse d'abord mes remerciements à mon maître de stage, Emmanuel GIGUET qui a su me faire confiance sur mon travail, mon autonomie et les décisions prises tout au long du stage. Je remercie notamment toute l'équipe SAFE du GREYC pour leur accueil et moments passés de détente.

Je tiens surtout à remercier Davy GIGAN et Pierre BLONDEAU de la D.S.I. qui ont été un très grand soutien tout au long du stage, que ce soit par mails ou aux réunions hebdomadaires. Ils ont été présents activement lorsque j'avais besoin d'aide. Sans eux, le stage n'aurait pas été satisfait et le projet, incomplet. Ils m'ont apporté tous les outils dont j'avais besoin pour travailler dans des conditions agréables.

Resumé du rapport de stage

Français

J'ai choisi de faire ce stage de 5 mois au laboratoire du GREYC dans l'objectif d'améliorer l'outil de signalement collaboratif Hamsa, projet prometteur de la promotion Master 2 S.S.I 2021-2022. Lors de ce stage, beaucoup de modifications sur le projet ont eu lieu, sur l'application Hamsa et sur Zimbra. Du côté de l'application Hamsa, une API stockage de mails de signalement, j'ai sécurisé l'ensemble des méthodes et fiabilisé les communications entrantes et sortantes. De l'autre côté, Zimbra et sa Zimlet ont reçu des améliorations de stabilité et de nouvelles fonctionnalités, notamment sur son service de proxy. J'ai mis en place deux protocoles d'authentification distinct, par serveur CAS ou par proxy de Zimbra afin de laisser une liberté quant à l'utilisation de l'application Hamsa. Chacun a ses avantages et inconvénients, la D.S.I. de l'Université de Caen pourra choisir le plus adapté à la situation.

Des réunions toutes les semaines avec Davy GIGAN, Pierre BLONDEAU et Emmanuel GIGUET ont été mise en place pour faire un point sur les avancées et les objectifs pour la semaine suivante. C'est un projet qui servira à protéger le personnel de l'Université et les étudiants, mais aussi à construire une base d'apprentissage pour une I.A dédié à la détection automatique de mails. Je suis fier d'être acteur et de contribuer à ce projet en appliquant mes connaissances et de le voir en oeuvre.

Anglais

I chose to do an internship at GREYC Laboratory in order to upgrade the collaborative reporting tool Hamsa, promising prototype created at the beginning of the first semester 2021-2022 by the Master 2 INFOSEC. During this internship, I made a lot of changes, both Hamsa and Zimbra. On Hamsa, I secured the API endpoints and made both methods and communications reliable. On Zimbra and its Zimlet, both got reliability updates and new features added, especially on internal proxy, service provided by Zimbra. On the whole scale, I made 2 new distinct authentication protocols, one with CAS integrated server and one with internal proxy from Zimbra. I made both of these because we want to leave a choice, depending on the situation. Both have their pros and cons and it'll be up to the I.T. department to decide which one to use.

Each week, there is meetings with Davy GIGAN, Pierre BLONDEAU and Emmanuel GIGUET to check on what is done and the objectives for the next week. It's a project that will protect university staff and students but also provide a training database for A.I. specialized in dangerous mail detection. Overall, I am very proud to be an actor, a contributor to this project by applying my knowledge and see it in work.

2 Présentation du laboratoire GREYC

Mon stage s'est déroulé au sein du laboratoire GREYC à Caen, dans l'équipe de cybersécurité SAFE.

Le GREYC, Groupe de Recherche en Informatique, Image, et Instrumentation de Caen, est un laboratoire de recherche en sciences du numérique dont les savoirs-faire sont centrés sur l'informatique et l'électronique. Il est dirigé par Christophe Rosenberger. Implanté en Normandie, le laboratoire, est associé au CNRS, à l'Université de Caen Normandie et à l'École Nationale Supérieure d'Ingénieurs de Caen.

Le GREYC est reconnu sur la scène nationale et internationale pour ses contributions scientifiques originales, ainsi que pour ses réalisations matérielles et logicielles. Près de 200 chercheurs, enseignants, ingénieurs, techniciens, post-doctorants, doctorants, et personnels administratifs contribuent à l'avancée des connaissances au travers de projets de recherche fondamentale et appliquée.

Les collaborations pluridisciplinaires avec les sciences humaines et sociales, les mathématiques et les sciences de l'ingénieur, ainsi que les partenariats industriels, sous forme de contrat ou de création de startups, contribue à son rayonnement.

Le GREYC est structuré en 6 équipes de recherche, dont 3 sont physiquement situées dans les locaux de l'Université de Caen, les 3 autres étant situées dans un bâtiment de l'ENSICAEN :

— l'équipe AMACC, dont l'expertise porte sur l'algorithmique, les modèles de calcul, l'aléa, la cryptographie, et la complexité (AmacC)
— l'équipe CoDaG, centrée sur les systèmes à base de contraintes, la fouille de données, les ontologies, et l'analyse à base de graphes
— l'équipe MAD travaille sur la mise au point de modèles pour le raisonnement, aux systèmes multi-agents et à la prise de décision
— l'équipe Image, spécialisée en traitement d'image et reconnaissance des formes
— l'équipe Électronique, spécialisée dans l'étude des capteurs et du bruit à basse fréquence,
— l'équipe SAFE, spécialisée en sécurité informatique, sciences de l'investigation et biométrie

C'est au sein de cette dernière équipe, SAFE, que mon stage s'est déroulé.

L'équipe SAFE

L'équipe SAFE, dirigée par Christophe Charrier et co-directeur de mon stage, mène des activités de recherche en sécurité informatique suivant trois axes qui traitent d'aspects à la fois théoriques et applicatifs de la sécurité :

— l'axe *Biométrie*
— l'axe *Architecture et modèles de sécurité*
— l'axe *Science de l'investigation numérique* (Forensique)

C'est au sein de l'axe Science de l'investigation numérique, coordonné par Emmanuel Giguet, directeur de mon stage, que j'ai mené mes travaux de stage. Cet axe de recherche s'intéresse plus particulièrement à la mise au point et à l'évaluation de méthodes et d'outils au service de l'investigation criminelle numérique.

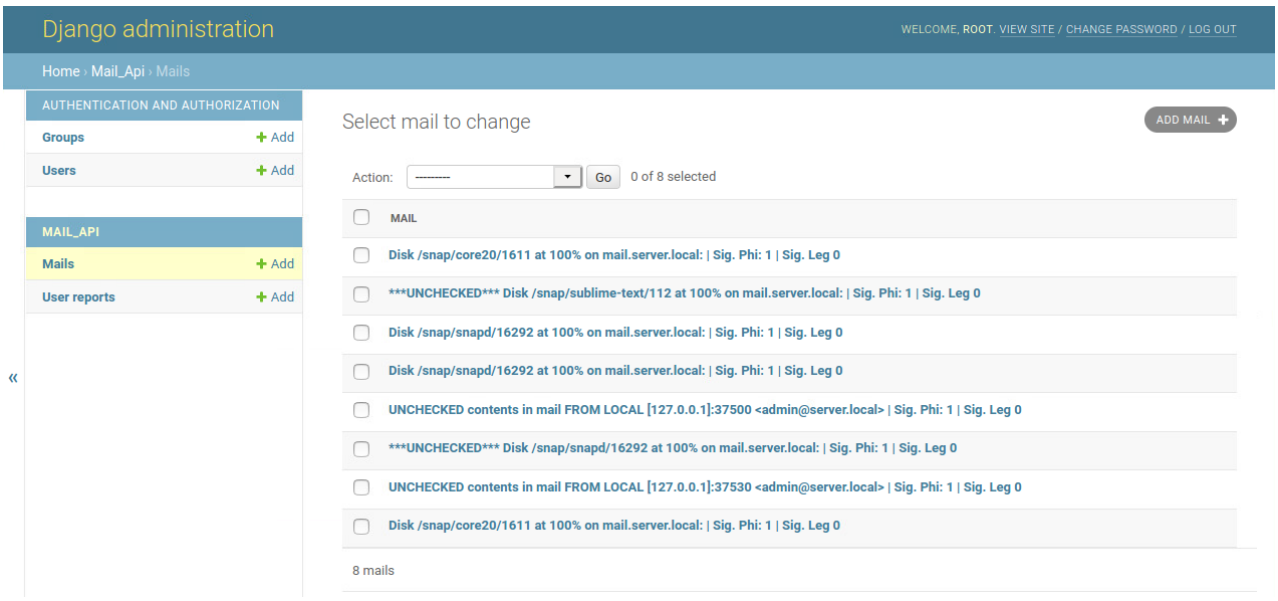
Les techniques d'investigation numérique trouvant des applications naturelles en criminalistique, l'équipe SAFE entretient des relations avec l'IRCGN ou la Section de Recherches de Caen. Mon stage, qui porte sur la détection de falsification vidéo, trouve des applications dans ce cadre.

3 Sujet de stage

1 Fonctionnement de l'application Hamsa

Dans le cadre d'un projet annuel, la promotion Master 2 S.S.I. (Sécurité des Systèmes d'Information) 2021-2022 de l'Université de Caen a été chargée de développer un outil contre les mails frauduleux pour le serveur de mail Zimbra. Deux pistes : Analyse des mails par intelligence artificielle et plateforme de signalement collaboratif. En fin de projet, la plateforme de signalement a été retenue, appelé Hamsa. À l'issue du projet annuel, le projet Hamsa est seulement un prototype fonctionnel qui a besoin de nombreuses modifications pour qu'il soit déployé sur l'infrastructure de l'université de Caen.

Hamsa est une application Web basée sur le framework Django et le module DRF (Django Rest Framework). Il s'agit d'une API qui reçoit des requêtes et renvoie une réponse appropriée. Les API peuvent servir de relais entre une base de donnée et un autre site web existant. Par exemple, un site web peut demander une information sur la météo dans un lieu avec simplement une requête vers l'API gérant les données météo. Dans notre cas, l'application Hamsa sert de stockage de mails signalés par les usagers. Afin de réaliser un signalement d'un mail suspect sur Zimbra, il faut récupérer les informations importantes du mail, construire le message et l'envoyer à l'API Hamsa. Afin de réaliser ce traitement en un clic par l'utilisateur, Zimbra permet l'intégration de modifications sur l'interface utilisateur de Zimbra (ou extension) appelés Zimlets.

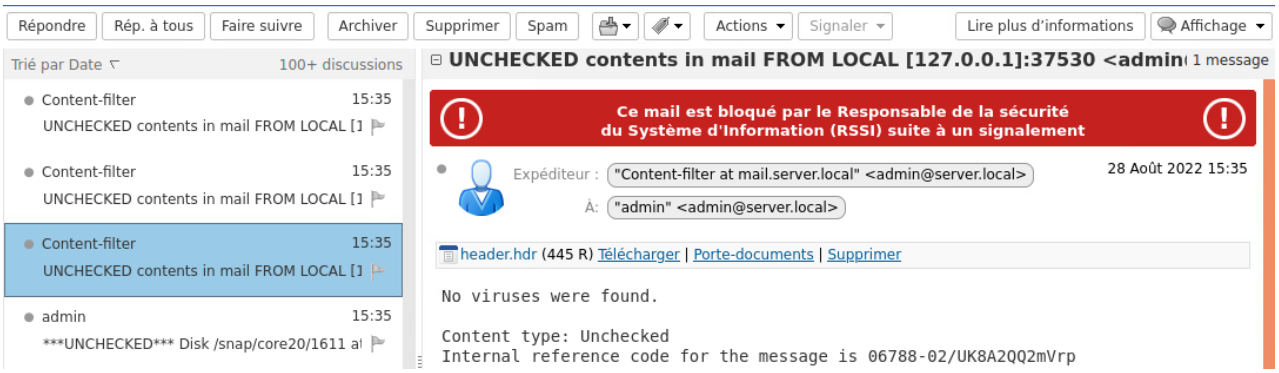


Page administrateur de Hamsa avec plusieurs mails signalés

Un Zimlet est un fichier Javascript qui permet de réaliser des modifications graphiques, ajouter des widgets ou récupérer des informations présentes sur la page web en temps réel. Grâce à cette Zimlet, on peut récupérer les informations nécessaires d'un mail ouvert telles que le sujet, le contenu, le destinataire, l'envoyeur et autres données pour le signalement. Afin que l'utilisateur puisse signaler un mail, un bouton a été intégré sur l'interface web de Zimbra. Il est possible de signaler un mail frauduleux, mais aussi un mail considéré légitime, comme faux positif.

Premièrement, à chaque ouverture de mail, le Zimlet se charge d'envoyer une requête vers Hamsa pour savoir si le mail en question est présent dans la base de donnée. Donc un mail présent dans la base de donnée est un mail qui a été signalé comme suspect auparavant par un utilisateur. Hamsa vérifie donc si les informations reçues correspondent à un mail signalé puis renvoie une réponse positive ou négative à la Zimlet. Lorsque la réponse est reçue, si positive, une bannière d'alerte s'affiche au-dessus du mail ouvert pour avertir l'utilisateur qu'il s'agit d'un mail suspect, potentiellement dangereux. Il existe deux types de bannière, chacune correspondant à un degré de danger.

Le premier degré de danger est un simple avertissement lorsque le mail a été signalé quelques fois comme un mail suspect ou alors lorsque le mail a déjà été signalé comme frauduleux, mais avec des signalements de faux positif. L'utilisateur est donc avertit qu'il s'agit potentiellement d'un mail frauduleux grâce à une bannière d'alerte orange demandant d'être vigilant et peut confirmer ou non à l'aide de son signalement. Le second degré de danger est celui d'un mail frauduleux confirmé par la D.S.I ou signalé un certain nombre au-delà d'un seuil défini par le Zimlet. Par exemple, les mails suspects ayant plus de 50 signalements comme mail suspect ou alors vérifié par la D.S.I sont dans cette catégorie. Les utilisateurs sont avertis visuellement par une bannière rouge pour les mails classés comme dangereux et déconseille fortement d'accéder aux liens présent dans le mail.

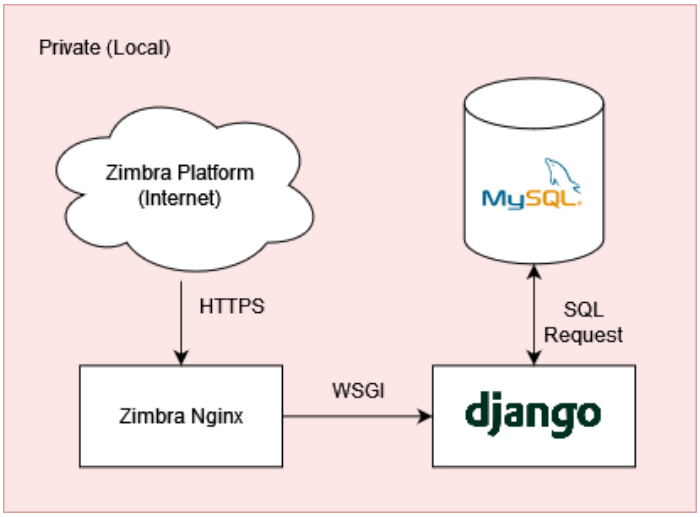


Exemple de bannière orange pour signaler l'utilisateur d'un mail suspect

Lorsqu'un mail est signalé par l'utilisateur via le Zimlet, une requête est envoyée à l'API Hamsa, contenant plus d'informations que la requête précédente qui vérifie l'existence du mail dans la base de données de signalement de Hamsa. Pour réaliser un signalement, il suffit d'appuyer sur le bouton de signalement et choisir si c'est un signalement de mail suspect ou un signalement de mail légitime. Du côté de l'application Hamsa, on récupère les données et on les valide pour créer une nouvelle entrée dans la base de données. On retourne une réponse positive si correctement enregistrée.

2 Architecture initiale de l'infrastructure

L'architecture à la fin du projet et au début de mon stage est de la forme suivante : l'application Hamsa connectée a une base de données MySQL. La communication se fait entre la Zimlet et Hamsa via des requêtes HTTP. Les données transmises sont au format JSON.



Architecture initiale de Django et Zimbra

3 Analyse critique

Nous avons donc un projet fonctionnel qui répond à la demande de la D.S.I. qui est de proposer une plateforme de signalement pour Zimbra. Mais si on regarde sur chaque composant de ce projet, beaucoup de problèmes sont identifiables :

1. Une infrastructure Django pas à jour, des patchs de sécurités, résolution de bug et ajout de nouvelles fonctionnalités. La version de Django sur le projet est la version 2.2 LTS (Long Term Support) qui n'est plus maintenue depuis avril 2022 par l'équipe responsable du développement. De plus, il n'y a aucune vérification de l'intégrité du message, impossible de savoir si le message a été forgé intentionnellement pour polluer ou se faire passer pour un autre mail légitime et crédible. Le système d'authentification de base n'est pas du tout adapté à notre demande et n'est donc pas utilisé. Il faut trouver un nouveau moyen qui permet à l'utilisateur de s'authentifier avec le minimum d'interaction avec Hamsa.
2. La Zimlet manque de robustesse, peu de feedback pour l'utilisateur et pour le développeur, le code n'est pas lisible. Il y a donc quelques travaux à réaliser sur la partie Zimlet
3. L'architecture générale du projet fonctionne bien en local avec quelques personnes communiquant les informations. Mais l'Université de Caen peut potentiellement atteindre les centaines voir milliers d'utilisateurs connectés simultanément. De plus, le projet lors de ses tests de performance n'a pas atteint la demande en termes de requêtes traités par secondes. De plus, l'architecture n'est pas disposée d'un serveur web adapté pour du gros trafic, tel que nginx ou Apache, dont les fonctionnalités pourraient nous intéresser (Load balancer, HTTPS, ...).
4. L'application Hamsa, elle-même n'est pas adaptée à un environnement en production, les fichiers de configurations sont prévues pour un environnement en local.

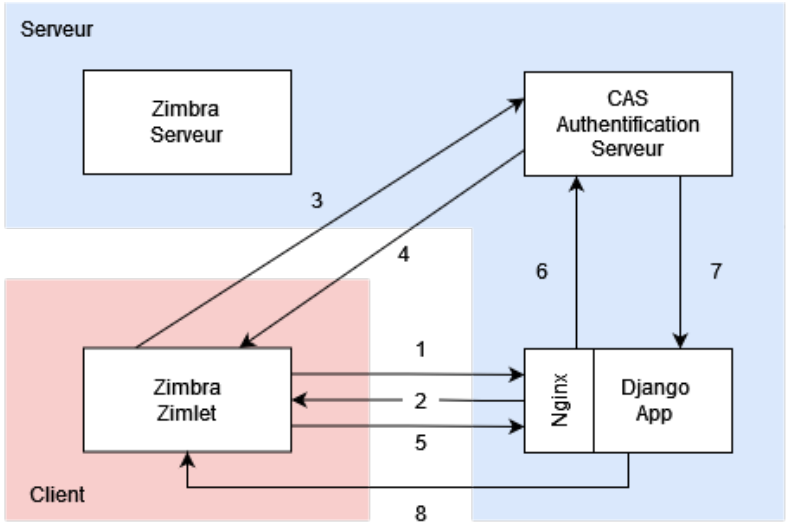
Il y a donc beaucoup de points négatifs à résoudre pour permettre la mise en production de cette application et il faut assurer sa robustesse ainsi sa performance face aux attaques (Surcharge, Forgeage, ...).

4 Systèmes d'authentification

1 Authentification par server CAS dans Hamsa

1.1 Architecture CAS

Un serveur CAS (Central Authentification Service) est un système d'authentification unique pour le web. Il est largement utilisé dans les universités et organismes du monde. Lorsque quelqu'un s'identifie sur un site web grâce au serveur CAS, il est automatiquement authentifié par tous les autres sites web utilisant le même serveur CAS.



1. La Zimlet envoie une requête API Ajax à Hamsa.
2. Django renvoie une redirection au le client vers le serveur CAS pour s'authentifier
3. Le client s'authentifie
4. Le CAS renvoie une redirection vers l'application Django avec token d'authentification si succès.
5. Le client est redirigé vers Django avec le token d'authentification
6. Django vérifie le token auprès du serveur CAS
7. CAS renvoie une réponse succès et Django crée une session pour le client
8. Django finalise le traitement et répond à la requête API initiale.

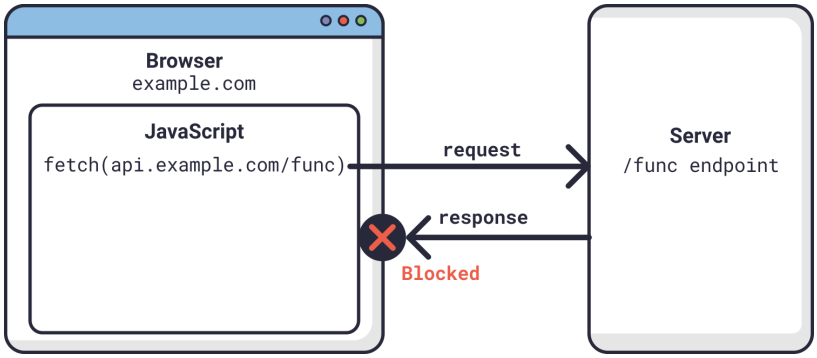
Architecture générale avec intégration du serveur CAS avec détails de chaque requête numéroté par ordre chronologique

La première architecture se repose sur une authentification par serveur CAS. L'utilisateur s'identifie grâce au serveur CAS pour accéder à Zimbra et aura accès à Hamsa automatiquement. L'intégration de CAS dans notre projet apporte une couche de sécurité robuste supplémentaire pour les requêtes de Zimlet vers Hamsa. Cela permet de protéger Hamsa des communications anonymes extérieure souhaitant polluer la base de données en forgeant des messages similaires à ceux que le Zimlet envoie. Afin de communiquer avec Hamsa, l'utilisateur est obligé de s'authentifier sur le CAS s'il n'est pas authentifié sur Zimbra. Mais avec le CAS, si l'utilisateur est authentifié sur Zimbra (donc légitime), il est automatiquement authentifié sur l'application Hamsa.

Plusieurs modules pour l'intégration du CAS sont disponibles sur Internet, chacun ont leurs avantages et inconvénients : gestion de serveurs multiples, degré de customisation, etc ... J'ai donc opté pour un module simple d'utilisation et facilement réglable via les paramètres de Django : django-cas-ng. Pour ajouter un module sur Django, il faut l'installer via pip et ajouter son nom dans les réglages Django. Il n'y a qu'un paramètre obligatoire à préciser : l'adresse du serveur CAS. Dans notre cas, le serveur CAS de l'université de Caen est "https ://cas.unicaen.fr". Après cet ajout, les modifications nécessaires à la base de données ont été réalisées automatiquement pour stocker les tokens d'authentification et gérer les connexions.

1.2 Protocole CORS

Le protocole CORS (Cross-Origin Ressource Sharing) est un protocole qui rajoute des en-têtes HTTP afin de permettre à navigateur web, appelé client, d'accéder à des ressources (images, scripts, données d'une API) d'un serveur situé sur un autre domaine différent du domaine d'origine. Ce protocole est déployé pour autant de libertés que les requêtes de mêmes origine tout en sécurisant la communication. Pour information, les en-têtes HTTP sont des informations sur la requête, son type, provenance, protocole utilisé et autres.



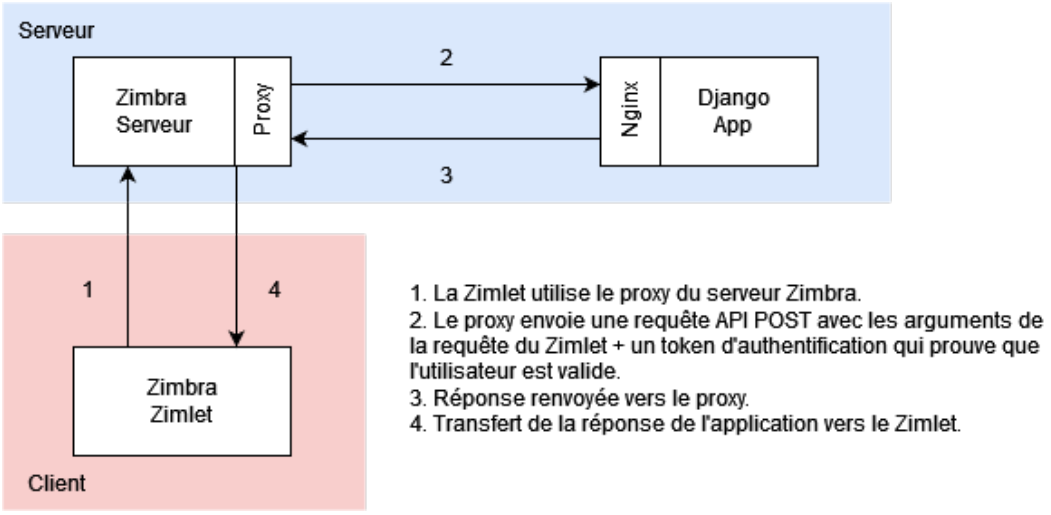
Protocole CORS refusant une requête sur un serveur qui n'a pas renvoyé une réponse avec les en-têtes requis - Source : Victoria Lo

Notre service mail Zimbra et l'application Hamsa, possèdent chacun un nom de domaine différent (webmail.unicaen.fr et hamsa-dev.greyc.fr), ils ne peuvent donc pas communiquer de données entre eux, car le protocole CORS est mis en place automatiquement. Cela fonctionne de la manière suivante : dans la requête initiale, nous avons un en-tête HTTP spécifique appelé "Origin" suivi du domaine sur lequel la requête a été réalisée (Dans notre cas, Zimbra). Lorsque la requête parvient au serveur, il fait son traitement et renvoie une réponse contenant l'en-tête "Access-Control-Allow-Origin" suivi d'un astérisque (tous les domaines autorisés) ou d'un domaine spécifique. Si le domaine présent dans l'en-tête de la requête "Origin" n'est pas inclus dans l'en-tête "Access-Control-Allow-Origin" de la réponse, alors l'échange ne peut aboutir.

Pour notre application, il s'agissait d'un problème difficile à résoudre. Le manque d'information au début nous a fait perdre beaucoup de temps, et il a fallu retracer le chemin de la requête, comparer les échanges valides, la différence des en-têtes, etc. Il existe une solution simple pour le serveur Hamsa : le paquet django-cors-headers. Il s'agit d'un paquet Django qui ajoute automatiquement les en-têtes CORS nécessaires, configurable. Nous avons jugé ce paquet comme une sur-couche inutile que nous devions simplifier au maximum pour la performance. Les réponses du serveur Django passent par plusieurs interfaces avant de terminer sur le client Zimbra, il y a donc la possibilité de rajouter les en-têtes CORS dans Nginx ou dans l'application Hamsa. Nous avons choisi Hamsa pour les placer afin de ne pas altérer les performances optimales de Nginx.

2 Authentification par proxy de Zimbra

La second architecture possible utilise sur une fonctionnalité déjà existante dans Zimbra : le proxy. En effet, Zimbra propose un proxy interne (sous l'URL .../service/proxy par défaut) pour que les Zimlets puissent communiquer avec les sites web en dehors de Zimbra afin de résoudre automatiquement les problèmes engendrés par le protocole CORS. On sait que l'utilisateur doit être authentifié pour utiliser le proxy de Zimbra mais, Hamsa ne sait pas de quel utilisateur il s'agit. Il y a donc une modification à réaliser sur le code source de Zimbra afin que l'application Hamsa puisse savoir qui est l'utilisateur qui a signalé le mail.



Architecture générale avec intégration du proxy avec détails de chaque requête numéroté par ordre chronologique

Pour que l'utilisateur puisse utiliser le proxy de Zimbra et communiquer avec l'application par le biais du proxy, il doit être authentifié sur Zimbra. Au lieu de faire une requête directement à l'application Hamsa, elle est dirigée vers l'URL du proxy qui va retransmettre ensuite la requête à Hamsa. Mais l'API n'a aucun moyen de savoir qui est l'utilisateur à l'origine de cette requête vu qu'il n'y a pas les informations permettant de l'identifier. Cette architecture est efficace, car elle part du principe que les requêtes envoyées depuis le proxy de Zimbra sont forcément des utilisateurs connus de l'Université qui se sont déjà authentifié. Pour résoudre le problème d'identification, nous avons intégré dans la requête du proxy, un en-tête customisé qui contient l'adresse mail de l'utilisateur authentifié. Ainsi, l'application Hamsa n'à qu'a vérifier l'en-tête et valider la requête. Il est possible de modifier l'en-tête, lui donner un nom de variable différent et de l'utiliser pour d'autres API qui utiliseraient un système similaire à celui de Hamsa.

Mais ce n'est pas suffisant, avec cette architecture, une personne pourrait forger une requête avec le bon format de données et l'en-tête customisé afin qu'elle soit acceptée auprès de l'application Hamsa. Afin de contrer ce problème, une vérification avec liste blanche a été mise en place : l'origine de chaque requête est vérifiée si elle correspond à une adresse IP ou à une URL d'un site web. Cette liste blanche donne donc l'autorisation de communiquer avec l'application.

Par rapport à l'architecture précédente, avec le serveur CAS, celle-ci est indépendante, il suffit de mettre à jour le Zimbra pour intégrer la fonctionnalité, faire les réglages nécessaires du côté de Hamsa et de Zimbra afin d'assurer la bonne communication.

5 Application Hamsa : Un travail de stabilisation, sécurisation et scalabilité.

1 Professionnalisation et refactorisation des points d'accès API de Django

Mettre à jour Django est l'une des première actions que j'ai faite lorsque j'ai commencé mon stage. Cela assure une stabilité de Django, une sécurité supplémentaire contre les failles de sécurités dans le framework Django et de l'optimisation des performances. Hamsa a été construit sur une version de Django qui n'est plus maintenu depuis avril 2022, il fallait donc mettra à jour vers une version récente et maintenu jusqu'à un certain temps. Notre candidat idéal est la version 3.2 LTS de Django qui sera maintenu jusqu'en 2024, il s'agit d'une version stable et ne recevra que des patchs de sécurité et d'intégrité. Suivant la mise à jour de Django, ses modules tels que DRF ou django-cas-ng sont mis à jour eux aussi.

Des optimisations ont été apportées au niveau des requêtes SQL, notamment pour le traitement de l'existence du mail signalé dans la base de données. Vu qu'il s'agit d'une requête SQL qui va être exécutée de nombreuses fois en un cours laps de temps, il est important de pouvoir l'optimiser en intégrant la notion d'index SQL. Un index n'est pas un objet et se situe au niveau de la donnée, son but est d'indexer certaines colonnes d'une table et d'optimiser le parcours des données d'une table. Toutefois, un index utilise de l'espace, de la mémoire lors de son utilisation et nécessite un coût supplémentaire lors de la suppression, modification ou ajout d'une donnée dans la table. Dans notre cas, le signalement d'un mail sera beaucoup moins sollicitée qu'une vérification donc ajouter un index à la table est intéressant. De plus, Django intègre cette notion d'index de manière simple en écrivant explicitement quels sont les champs qui sont seront des index ou index liés :

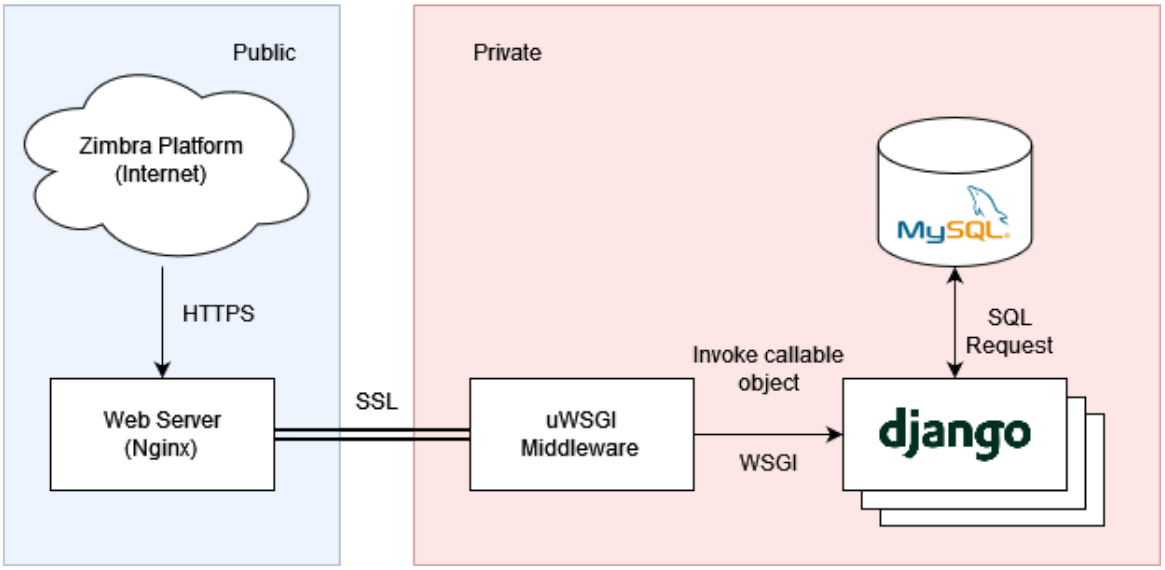
```
1 indexes = [
2     models.Index(fields=['mail_id'], name="mail_id_idx"),
3     models.Index(fields=['mail_subject', 'mail_content_hash'], name="
    subj_conthash_idx")]
```

Code sur l'index

En plus de cela, un travail a été réalisé sur les méthodes, appelées "points d'accès" qui se chargeront de recevoir les requêtes HTTP, la traiter et finalement renvoyer une réponse. Si ces méthodes sont trop complexes, le temps de traitement pourrait être considérablement long en cas de trafic élevé. On procède de la manière suivante : on vérifie l'intégrité du contenu, si altéré, on retourne une erreur puis on vérifie la validité des données (dans le bon format, protection contre les potentielles injections SQL), enfin, on réalise le traitement, la vérification du mail dans la base de données ou sa création. Des sécurités sont mises en place pour couvrir le maximum de situations, telle que la situation ou deux personnes signalent un mail en même temps, il y a une chance que deux entrées identiques soient créée. Dans le cas normal, la méthode de signalement procède à une vérification si le mail existe dans la base, si il existe, on incrémente le compteur de signalement. S'il y a deux mail

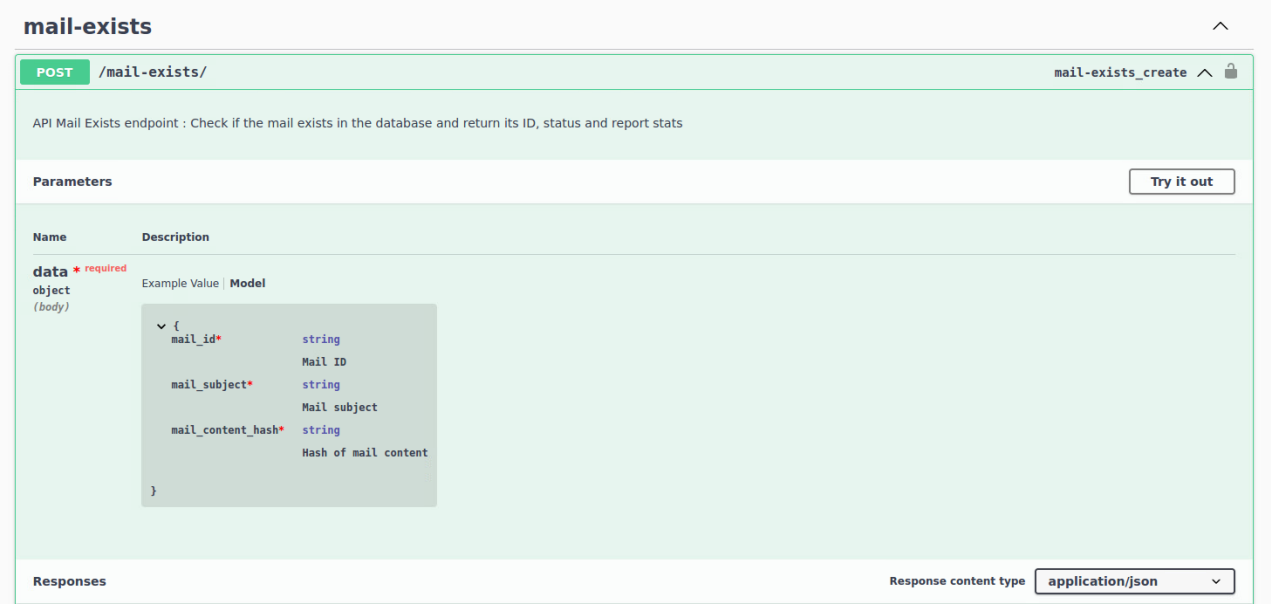
identiques, on supprime donc le/les plus anciens et on incrémente le plus récent, normalement unique.

A côté de cela, des couches de sécurités ont été ajoutées pour préparer la mise en production. Il y a d'abord nginx qui servira de proxy pare-feu. Il permettra d'autoriser la communication HTTPS configurée par mes soins. nginx est un serveur web simple d'utilisation et clair : on peut paramétrer facilement sur quels ports les requêtes doivent communiquer et surtout bloquer les requêtes HTTP et n'autoriser que les requêtes HTTPS. Forcer le protocole HTTPS garanti une protection des données des utilisateurs, vu comme un gage de sécurité. Les données échangées sont cryptées grâce au certificat SSL, donc un tier entre le client et le serveur ne peux pas comprendre l'échange de données. J'ai pu installer le protocole HTTPS sur le nginx de ma machine sans soucis, il faut seulement générer un certificat SSL et ajouter des paramètres dans le fichier de configuration nginx. A savoir qu'un certificat SSL doit être renouvelé tous les ans pour la sécurité.



Trajet d'une communication HTTPS entre le navigateur jusqu'à Hamsa

Hamsa est une API REST qui a besoin que les données reçues sont sous le bon format avec les paramètres à envoyer et leur type de donnée. La personne ne connaissant pas le code ne peut pas savoir ce qu'il doit construire et envoyer à l'API. L'ajout d'une documentation est donc primordial pour le bon fonctionnement d'une API. On appelle cela une spécification API : elle standardise les communications entre le client et le serveur en fournissant une documentation claire avec la donnée attendue et ce qu'elle retourne en cas de succès ou d'échec. Après quelques recherches, nous avons trouvé un module Django qui répond à notre demande : drf-yasg, "DRF-Yet Another Swagger Generator". C'est un module simple d'utilisation, on lui fournit un URL dédié que l'on va appeler .../docs/ qui sera accessible aux développeurs de l'Université de Caen, et le module se charge de créer une interface interactive avec les points d'accès de l'API. Afin de construire cette documentation, j'ai créé un ficher Python appelé "schema.py" contenant toutes les informations sur les requêtes et réponses de chaque point d'accès de l'API.



Aperçu de la spécification API avec format de requête et réponses prévues, la méthode mail-exists/ et ses paramètres attendus

Pendant le développement, je n'ai pas rencontré de difficultés sur les modifications précédentes, Django est un outil puissant et largement utilisé donc en cas de problème, la solution était souvent disponible sur Internet. L'intégration du protocole HTTPS et de la mise en place de Nginx etait quelque chose de nouveau pour moi, il y a eu donc plusieurs heures de recherches et de compréhension sur son fonctionnement.

2 Tests de charge sur différentes architectures logicielles et matérielles

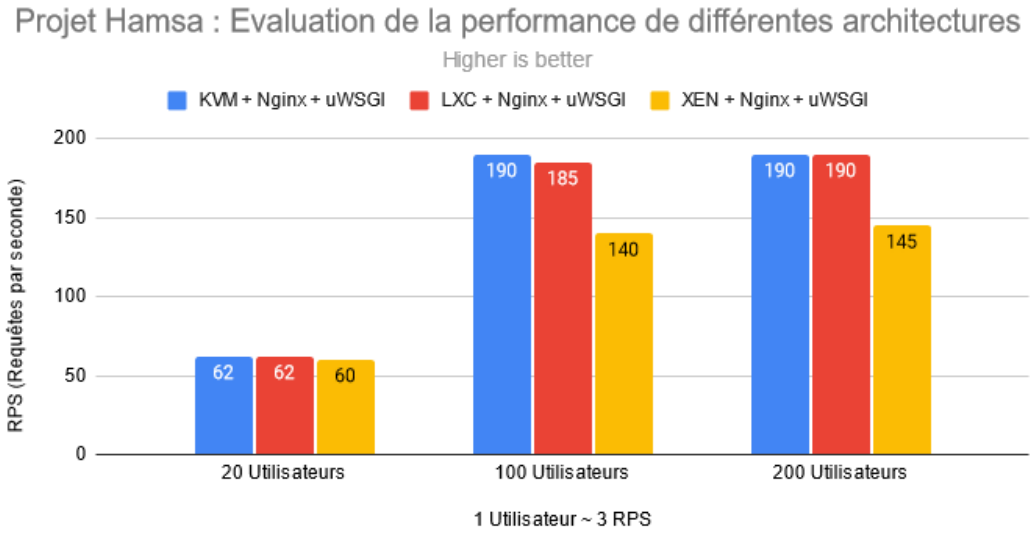
Afin de tester la performance de l'application Django, nous avons utilisé des outils de tests de charge tels que Locust ou Apache Benchmark. Les tests de charge réalisés pendant le projet de Master ne sont pas convaincants et ne reflètent pas la réalité. En effet, tout a été réalisé en local et sur une machine virtuelle, les communications ne sont donc pas les mêmes qu'en réalité, avec une architecture web-client. Donc notre objectif sur les tests de charge est de voir quelles sont les réelles performances de Django, observer son comportement et potentiellement trouver des "goulots d'étranglement" ou "bottlenecks", c'est-à-dire des performances réduites à cause d'un composant matériel ou logiciel. Les deux tests de charge suivants ont été réalisés sur un serveur dédié à Django, dans un environnement 1 coeur CPU et 4Go de RAM puis 4 coeurs CPU et 8 Go de RAM.

Exemple de visualisation Locust, en haut, le nombre de requêtes par secondes. En bas, temps de réponse moyen

Nous avons commencé par tester 3 architectures de machines virtuelles différentes afin de savoir quelle sera la plus performante :

- KVM (Kernel-based Virtual Machine) est une technologie de virtualisation open-source intégrée à Linux
- LXC est système de virtualisation, utilisant l'isolation comme méthode de cloisonnement au niveau du système d'exploitation, ou simplement appelé "containers"
- XEN est une hypervirtualisation qui fonctionne directement sur la couche matérielle en créant une couche de virtualisation entre la machine physique et la machine virtuelle.

Chacune de ses architectures a son propre moyen de créer un environnement dédié à Hamsa. Nous avons choisi pour chacune les mêmes puissances matérielles, soit 1 coeur CPU et 4 Go de RAM. Sur le graphe ci-dessous, on observe une différence de performance entre KVM et LXC qui sont à 190 requêtes par seconde en moyenne contre XEN qui est à 140 requêtes par seconde en moyenne.



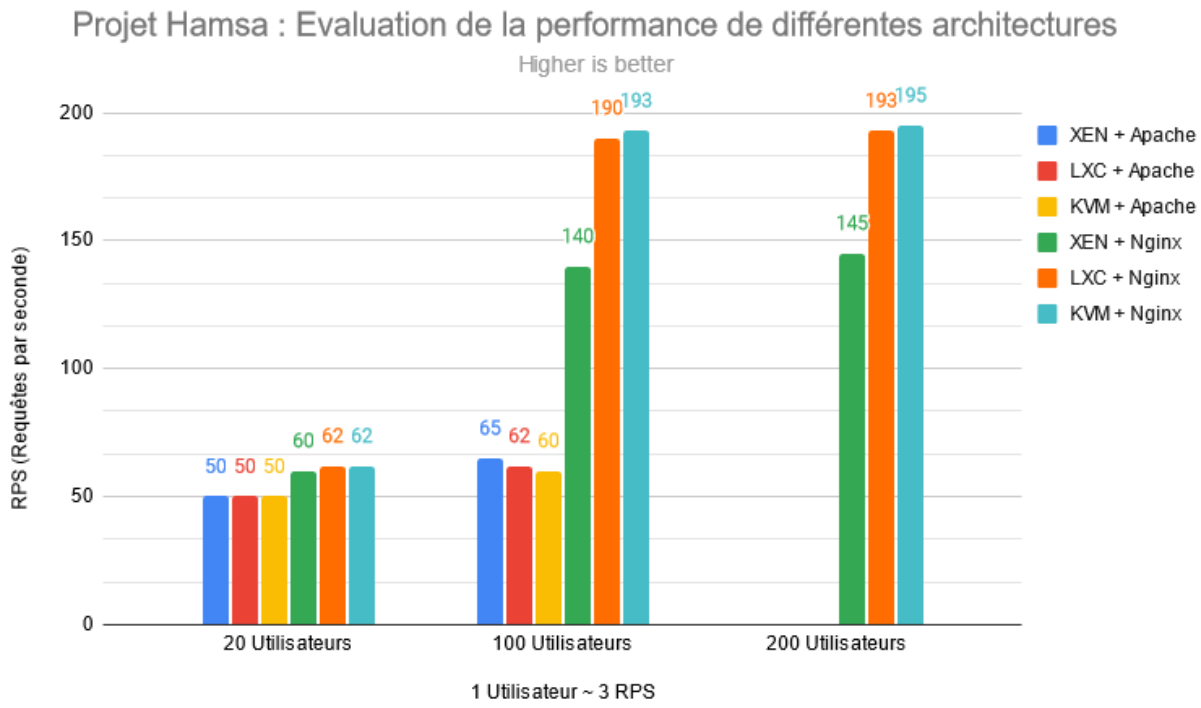
Comparaison des performances entre les 3 architectures de machine virtuelles KVM, LXM, XEN

Nous avons ensuite testé sur chaque architecture la performance de serveur web, deux candidats pour notre test de performance :

— la gestion des les connexions nginx se repose sur un algorithme de traitement basé sur les evènements.

— le serveur web Apache est basé sur une architecture basé sur les processus.

Chaque serveur est configuré de la même manière, toutes les ressources sont allouées de manière identique et les serveurs sont paramétrés sur du multi-threading. Nous avons fait les tests sur les 3 architectures différentes pour chaque candidat afin de vérifier s'il y a des performances notables. On remarque clairement que le serveur nginx est plus performant que le serveur Apache sur ces tests de charge.

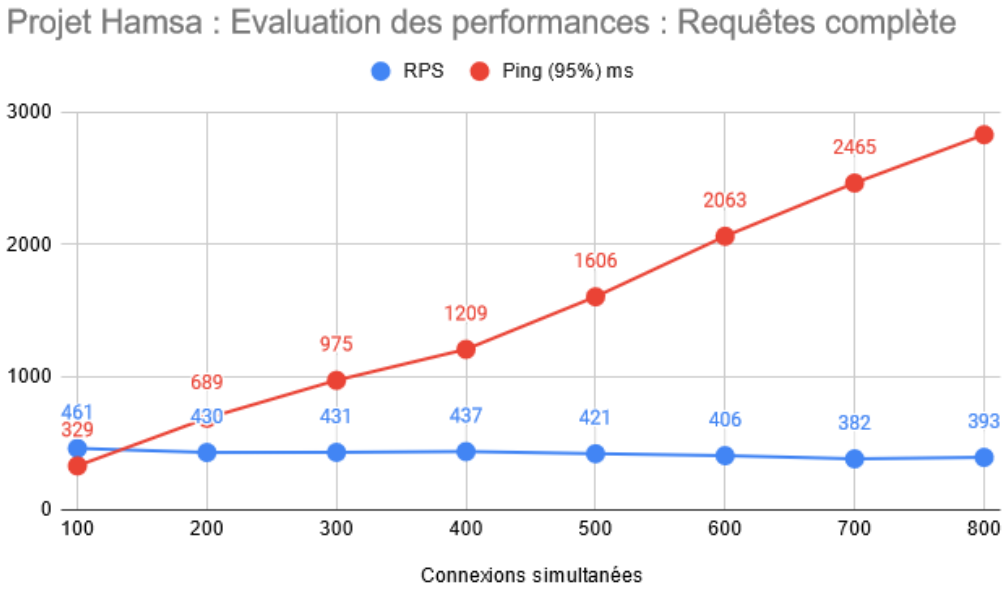


Comparaison des performances entre Apache et nginx

Après quelques suspicions sur la performance de Locust, j'ai opté pour un autre outil de test de charge, : Apache Benchmark, outil simple qui s'exécute en ligne de commande dans un terminal. C'est un outil simple et rapide d'utilisation, mais limité dans ce que l'on peut faire. Par exemple, on ne peut pas trier les requêtes non abouties et les requêtes d'erreurs aboutis qui est pris en compte dans le nombre de requêtes par secondes. Après les précédents tests, nous avons sélectionné le serveur khamsa2 avec l'architecture KVM avec nginx et uwsgi pour le prochain test, jugé comme la plus performante. La commande utilisée pour exécuter le test de performance sur un candidat :

```
$ ab -p post.txt -l -T application/json -c 100 -n 20000 https://
 khamsa2.greyc.fr/mail-exists/
```

Le fichier post.txt contient un JSON avec des informations de mails adapté à l'API afin de tester toute la chaîne complète de Hamsa, du nginx jusqu'à la base de donnée SQL. On paramètre Apache-Benchmark pour exécuter 20000 requêtes au total pour 100 communications concurrentes à l'adresse khamsa2.greyc.fr. Ainsi, on peut savoir s'il y a un goulot d'étranglement dans notre application Hamsa. Le graphe ci-dessous réalise des tests de performances avec un nombre de connexions simultanés croissant, allant de 100 connexions jusqu'à 800. On remarque une moyenne de 400 requêtes par seconde avec un temps de réponse qui augmente au fur et à mesure. A savoir qu'une meilleure configuration matérielle a été utilisé sur serveur khamsa2, 4 coeurs CPU et 8 Go de RAM.



Tests de charge d'une requête en situation réelle

Après ces tests, nous avons pu déterminer l'architecture et la configuration la plus optimale pour notre application et voir quelles étaient le maximum de requêtes par secondes elle pouvait traiter. En résumé, notre application pourrait supporter des charges plus de 400 requêtes par seconde. Pour un parc informatique avec parfois 1000 utilisateurs connectés simultanément sur tous les services, avoir une capacité de 400 requêtes par seconde semble raisonnable.

6 Zimbra : Le Zimlet et contribution au proxy

1 Le Zimlet module de signalement

La Zimlet est une extension/module capable d'ajouter de nouvelles interactions et fonctionnalités pour améliorer l'expérience de l'utilisateur. Par exemple, on peut intégrer des fenêtres de dialogues pour un meilleur feedback sur les actions réalisés ou encore ajouter des boutons pour automatiser des tâches. A la fin du projet de Master, une Zimlet a été utilisée pour créer le bouton de signalement et faire une vérification automatique à l'ouverture d'un mail pour savoir s'il s'agit d'un mail dangereux ou non. Zimlet fonctionnelle, mais qui manque de retour ou même un retour incorrect : un signalement "réussi" alors que la requête n'a pas pu aboutir par exemple. Ma première tâche a été donc de résoudre ces problèmes et de structurer le code. Lors du développement, pour vérifier les modifications faites sur la Zimlet, il faut créer un paquet .zip contenant le fichier JavaScript et le fichier de configuration XML de la Zimlet puis exécuter des commandes pour installer ce paquet sur Zimbra, puis rafraîchir le cache de Zimbra pour appliquer les changements. Le fichier de configuration XML est important, car il permet d'ajouter des interfaces tels que boutons, donner des informations sur la Zimlet, sa version, description et les fichiers JavaScript à exécuter pour que la Zimlet soit fonctionnelle.

Afin d'assurer une intégrité entre Zimbra et l'application Hamsa, un paramètre supplémentaire dans le message a été ajouté, contenant le hash SHA-256 du message. Lorsque l'API va recevoir la requête, le hash SHA-256 va être comparé avec le message reçu, et si la comparaison n'est pas valide, alors la requête n'est pas traitée. Du coté de Zimbra, il a fallu intégrer un fichier Javascript avec la méthode pour hasher un message en SHA-256. Le fichier Javascript doit être dans le paquet .zip pour que le zimlet puisse utiliser la fonction de hashage.

Lorsqu'une modification a été faite sur l'application Hamsa, il est important de vérifier que le Zimlet puisse toujours communiquer avec, sans erreurs. Par exemple, si nous avons changé un nom de variable ou que la réponse renvoyée a été améliorée au niveau de l'API, il faut prévoir ce changement sur le JavaScript de la Zimlet aussi. J'ai amélioré le feedback et le traitement des réponses avec des messages plus structurés du type Status/Message. Ainsi, on peut savoir rapidement d'où vient l'erreur, de la Zimlet ou de l'application Hamsa.

2 Le proxy interne de Zimbra

Lors de nos réflexions sur les architectures d'authentification qui pourrait être appliquées à l'application Hamsa, nous avons eu l'idée d'exploiter le proxy de Zimbra pour servir d'authentification. En effet, pour utiliser le proxy de Zimbra, l'utilisateur doit être authentifié sur la plateforme Zimbra. Nous allons donc partir de cet avantage pour communiquer à Hamsa le fait que l'utilisateur qui a fait le signalement est déjà authentifié auprès d'un service de l'Université.

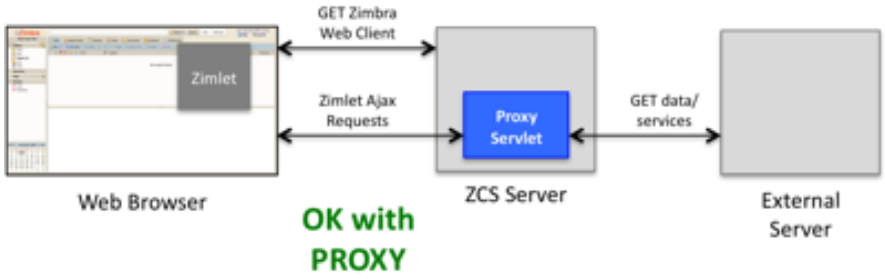
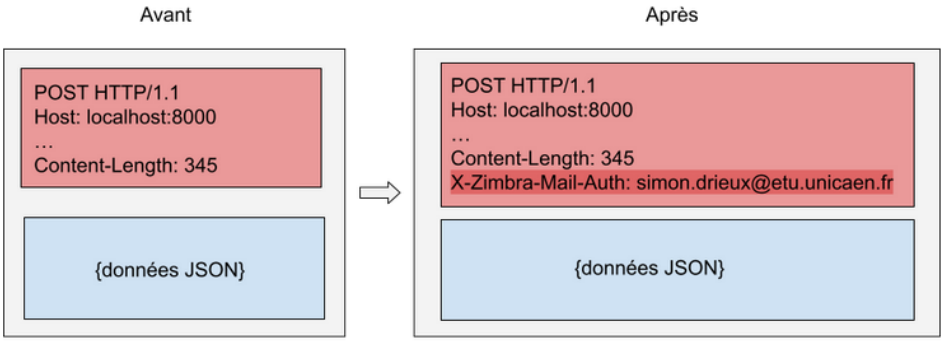


Schéma du proxy de Zimbra en fonctionnement sur un exemple simple

Afin que Hamsa puisse identifier l'utilisateur à l'origine de la requête, il doit récupérer une information en provenance du proxy. Zimbra est une application open-source donc il est possible de modifier le code et de proposer des améliorations ou bugfix aux mainteneurs. Notre idée est d'ajouter l'adresse mail de la personne authentifiée dans un en-tête customisé qui va être ajouté dans la requête du proxy. Notre modification est simple et reste totalement optionnelle, donc mettre à jour l'application avec les paramètres initiaux n'impactera pas son fonctionnement. Afin de paramétrer cet ajout et l'utiliser pour Hamsa, il faut ajouter une nouvelle variable dans l'annuaire LDAP (Lightweight Directory Access Protocol). Il s'agit d'un annuaire où sont stockés de nombreuses variables servant par exemple de configuration pour Zimbra. La variable que l'on doit ajouter est : zimbraProxySendMailAddrHeader avec le format "domain,header", "domain" qui correspond au nom de domaine sur lequel le proxy est autorisé à communiquer et "header", son nom d'en-tête à envoyer avec l'adresse mail.



Objectif de notre modification : Ajouter un en-tête customisé dans la requête du proxy

Donc la première étape du développement de la nouvelle fonctionnalité est de créer ce nouvel attribut pour l'annuaire LDAP de Zimbra pouvant être accessible depuis le code Java de Zimbra. Il faut modifier le ficher XML listant tous les attributs LDAP avec leur type de donnée, sa cardinalité (mono ou multi) et une description. Après avoir ajouté notre attribut, il faut générer les getters et setters pour que le code Java puisse accéder et modifier cet attribut. L'outil de compilation "Ant" utilisé pour compiler le code Java de Zimbra dispose de commandes pré-faites pour diverses demandes comme la compilation complète, le déploiement, les tests unitaires, le nettoyage de dossier de sortie ainsi que de générateur de getters setters pour les attributs LDAP, celui qu'on va donc exécuter. Lorsque la génération est terminée, il est maintenant possible d'ajouter notre modification dans le code Java.

```
<attr
    id="3077" name="zimbraProxySendMailAddrHeader" type="string"
    cardinality="multi" optionalIn="account,cos" since="8.8.15">
  <desc>
      Specify allowed domain and custom header name for sendMailId in
    proxy request body forauthentication.
      Following format : domain,Header-Name
      Example : *.zimbra.com,Mail-Auth
  </desc>
</attr>
```

Nouvel attribut qui sera généré automatiquement

Il y a plusieurs étapes à réaliser avant de pouvoir ajouter notre en-tête à la requête du proxy. Premièrement, il faut récupérer l'attribut zimbraProxySendMailAddrHeader et ses valeurs depuis l'annuaire LDAP. On utilise donc les méthodes d'accès qui ont été générées auparavant pour récupérer ses valeurs. Si l'attribut est vide, qu'il n'a pas été configuré ou sous un mauvais format, on ne retourne pas de valeur et on n'ajoute donc pas l'en-tête. Ensuite, il nous faut récupérer l'adresse mail de la personne connectée à Zimbra, qui se fait de manière simple en allant récupérer dans la classe Account la méthode d'accès à l'adresse mail. Nous avons donc notre en-tête et notre adresse mail à envoyer que nous allons ajouter dans la requête pendant sa construction.

```
private Set<String> getProxySendMailAddrHeaders(AuthToken auth)
   throws ServiceException {
    Provisioning prov = Provisioning.getInstance();
    Account acct = prov.get(AccountBy.id, auth.getAccountId(), auth);
    Cos cos = prov.getCOS(acct); // Getters de l'annuaire LDAP
    Set<String> sendMailAddrHeaders = cos.getMultiAttrSet(
   Provisioning.A_zimbraProxySendMailAddrHeader);
    return sendMailAddrHeaders; }

private String getProxySendMailAddr(AuthToken auth) throws
   ServiceException {
    Provisioning prov = Provisioning.getInstance();
    Account acct = prov.get(AccountBy.id, auth.getAccountId(), auth);
    String mailAddr = acct.[...].getAddress();
    return mailAddr; }
```

Code Java pour obtenir la valeur de l'attribut et l'adresse mail de l'utilisateur

Notre proxy envoie l'adresse mail dans un en-tête customisé, il faut donc implémenter un "middleware" qui vérifie la présence de cet en-tête dans la requête et qui provient d'une origine placée sur liste blanche. Cette liste blanche est configurable dans le fichier de paramètre Django, adresses IP et URLs sont acceptés. Ainsi, les administrateurs peuvent accéder aux méthodes et au panel d'administration protégé par mot de passe pour gérer les mails signalés. Dans le cas ou la personne qui tente de communiquer avec Hamsa n'est pas sur la liste blanche, elle recevra automatiquement une réponse 405 Not Allowed. Ainsi, on assure une sécurité en limitant grandement toutes les communications possibles avec les points d'accès.

Cette modification du proxy m'a demandé 1 mois et demi voire 2 de travail. En effet, travailler sur des logiciels open-source n'est pas tâche aisée. Il faut appréhender l'architecture, comprendre comment l'application fonctionne et savoir à quel endroit il faut faire la modification. C'est avant tout un travail de recherche et il est important d'avoir une documentation claire et récente pour les développeurs vu qu'il s'agit d'un logiciel open-source. Par exemple, j'ai eu des soucis sur la compilation de Zimbra qui utilisait Java 8 alors que Zimbra utilisait la version 11 pour tourner son serveur de mail. Il était facile de faire des manipulations dangereuses qui pourrait détruire l'environnement de travail. Vu que je travaillais sur une machine virtuelle avec un environnement dédié à Zimbra, il était possible de créer des sauvegardes à un moment donné en cas de problème. Le manque de documentation sur la compilation et le manque de connaissances sur le code Zimbra fait que le développement était difficile. De plus, il fallait respecter les règles de code, noms de variable, indentations, ... qui ne sont inscrites nulle part publiquement. C'est la première fois que je travaille sur un logiciel open-source en ajoutant une modification, j'ai beaucoup appris sur son fonctionnement et le développement open-source en général. Je sais quelles sont les règles à suivre et comment procéder en cas de problèmes ou manque de documentation.

7 Conclusion

L'application, plateforme de signalement de mails dangereux, Hamsa est maintenant prête à sa mise en production grâce aux améliorations de performance, de sécurité et d'intégrité. Les communications sont devenues sûres entre Zimbra et Hamsa. Nous avons pensé cette application pour qu'elle soit générale et utilisable par d'autres universités qui sont dans le besoin d'une protection contre les mails frauduleux. Pour l'Université de Caen, la modification apportée au code source de Zimbra est un bénéfice et rend le fonctionnement de Hamsa indépendant de services tiers d'authentification. Je peux considérer le projet comme la version 1 qui recevra encore son lot de modifications dans le but d'améliorer la sécurité et d'optimiser les performances. Les nouvelles fonctionnalités pourraient être développées sur la détection automatique de mails qui pourrait faire le signalement automatique. L'objectif principal de Hamsa est de protéger les usagers, mais aussi de réunir les mails frauduleux afin de construire une base de données qui pourra servir à l'entraînement d'une Intelligence Artificielle sur la détection de mails.

Ce projet m'a permis d'acquérir une expérience sur la création d'une API, de la conception jusqu'à la mise en production. J'ai appris à me servir de nginx, utiliser le protocole HTTPS, les différentes manières de sécuriser une application. Grâce aux tests de charges, j'ai découvert que l'architecture des machines virtuelles peut avoir un impact sur les performances d'une application. J'ai pu surmonter les difficultés posées par Zimbra et le code source en allant pas à pas dans son fonctionnement. Les erreurs commises pendant le développement, les décisions prises aux réunions et les recherches approfondies sur des nouvelles technologies m'ont permis d'acquérir de nouvelles compétences et une professionnalisation.