Retour de l'URN sur mise en place de CAS 6.0.4

Contact : Vincent Bonamy (Vincent.Bonamy at univ-rouen.fr).

Contexte

Eté 2019, l'Université de Rouen Normandie a procédé à une montée de version de son service d'authentification **CAS en 6.0.4**.

Dans un premier temps, le Kit installation CAS V5.2 AMU nous a permis de mettre en place un CAS v5.2.7 fonctionnel comprenant les éléments principaux.

Depuis cette installation, on est ensuite passé en 6.0.3 puis 6.0.4 et on a affiné certaines configurations.

Nous partageons sur cet espace les points qui nous paraissent important à souligner par rapport à cette expérience de **mise en production d'un CAS 6.0.x.**

- Contexte
- ClearPass
- RememberMe
- Throttle
- Spnego / Kerberos
- Agimus
- Endpoints
- Tickets Registry
  - JPA (PostgreSql)
    - Problème
    - Analyse du problème
    - Remarques sur l'implémentation de la persistence JPA
    - Conclusion : le ticketRegistry JPA à éviter
  - Redis
  - Conclusion
- Misc
  - Iframe
  - Chunked transfer encoding
  - pam_cas

ClearPass

On a mis en place le ClearPass pour le faire fonctionner avec esup-filemanager 3.2.0.

Dans cette 3.2.0 d'esup-filemanager, nous avons simplement rajouté une petite modification pour gérer le domaine windows (modification intégrée dans esup-filemanager):
https://github.com/uPortal-contrib/esup-filemanager/pull/58

RememberMe

Nous avons activé le RememberMe dans CAS.

Nous souhaitions le rendre disponible uniquement sur 'mobiles'.

Aussi, pour ne laisser la case à cocher (permettant d'activer cette fonctionnalité) affichée que pour les 'mobiles', dans notre surcharge du footer.html on ajoute le bloc thymeleaf/javascript suivant :

```
<script type="text/javascript" th:unless="${#request.getAttribute('isMobile')}">
if(document.getElementById("rememberMe")) {
 document.getElementById("rememberMe").parentNode.style.display = "none";
}
</script>
```

Throttle

Modification du throttle en mettant :

```
cas.authn.throttle.failure.range-seconds=30
cas.authn.throttle.failure.threshold=12
```

Aussi sur un espace de 10 secondes (10=30/3), l'utilisateur est 'banni' temporairement à la 4ème tentative (12/3=4).

Spnego / Kerberos

Nous avons activé spnego + kerberos avec notre AD Microsoft

A noter que si l'authentification Spnego échoue (PC qui n'est pas dans le domaine ou navigateur non configuré), CAS renvoie le formulaire avec une réponse HTTP 401.

Firefox affiche alors la page de formulaire proposée, mais IE et Chrome réagissent différemment : suite au 401 ils affichent d'abord une popup d'authentifcation BASIC.
En cliquant sur cancel (annuler) l'utilisateur arrive bien sur le formulaire d'authentifcation CAS mais ce comportement n'est malgré tout pas souhaitable.

Nous avons donc activé spnego uniquement pour les navigateurs Firefox :

```
cas.authn.spnego.supportedBrowsers=Firefox
```

Concernant la mise en place de spnego, il faut bien suivre la documentation suivante :

https://apereo.github.io/cas/6.0.x/installation/SPNEGO-Authentication.html

On note que la création d'un compte AD pour le Service Principal Name consiste à créer un compte AD usuel et à appeler dans un powershell exécuté en tant qu'administrateur : domain-account est le nom du compte dans la documentation apereo (à l'URN on a préféré prendre un nom type cas-krb) :

```
Setspn -s HTTP/cas.example.com domain-account
```

On note également que dans cette commande ainsi que dans celle-ci :

```
ktpass /out myspnaccount.keytab /princ HTTP/cas.example.com@REALM /pass * /mapuser domain-account@YOUR.REALM
```

cas.example.com (cad cas) corrrespond **au nom de machine et non à un alias**. Utiliser l'alias et non le nom de machine est une erreur récurrente dans la mise en place de kerberos (erreur que nous avons faite, de manière obstinée ...).
A l'URN on a donc non pas saisi cas.univ-rouen.fr mais cast.univ-rouen.fr dans ces lignes de commande (cast étant le nom de machine hébergeant notre CAS : cf "host cas.univ-rouen.fr").

Agimus

On a patché (et proposé ces patchs par PR) cas-server-support-agimus-logs et cas-server-support-agimus-cookie pour qu'ils supportent CAS en 6.0.x :

- https://github.com/EsupPortail/cas-server-support-agimus-cookie/pull/1
- https://github.com/EsupPortail/cas-server-support-agimus-logs/pull/2

Endpoints

En 6.0.x, CAS propose d'utiliser les endpoints à la spring-boot pour interagir avec CAS CAS propose par ce biais des API REST.

Ces possibilités sont notamment décrites ici : https://apereo.github.io/2018/11/06/cas6-admin-endpoints-security/

A l'URN, on a activé ces endpoints pour notre application de gestion de comptes, afin que cette application puisse demander à CAS d'expirer les tickets de comptes compromis.

```
management.endpoints.web.exposure.include=*
management.endpoints.enabled-by-default=true
cas.monitor.endpoints.endpoint.defaults.access=IP_ADDRESS
cas.monitor.endpoints.endpoint.defaults.requiredIpAddresses=192.168.1.22,127.0.0.1
```

On a implémenté la destruction des sessions CAS d'un utilisateur ainsi depuis notre application de gestion de comptes (codée en java/spring) :

```
public class CasService {

	protected Logger log = Logger.getLogger(CasService.class);

	RestTemplate restTemplate;

	String casSsoSessionsUrl;

	String casDestroySsoSessionsUrl;

	public void setRestTemplate(RestTemplate restTemplate) {
		this.restTemplate = restTemplate;
	}

	public void setCasUrl(String casUrl) {
		casSsoSessionsUrl = casUrl + "/actuator/ssoSessions?type=DIRECT";
		casDestroySsoSessionsUrl = casUrl + "/actuator/ssoSessions/{ticketGrantingTicket}";
	}

	public synchronized String destroySsoSessions(String login) {
		String message = "";
		CasSsoSessions casSsoSessions = restTemplate.getForObject(casSsoSessionsUrl, CasSsoSessions.
class);
		for(CasSsoSession casSsoSession : casSsoSessions.activeSsoSessions) {
			if(login.equals(casSsoSession.principal)) {
				log.info(String.format("Call Cas Destroy ticket %s for user %s", casSsoSession.
principal, casSsoSession.ticketGrantingTicket));
				Map<String, String> urlVariables = new HashMap<String, String>();
				urlVariables.put("ticketGrantingTicket", casSsoSession.ticketGrantingTicket);
				CasSsoSessionDestroyResponse casSsoSessionDestroyResponse = restTemplate.
postForObject(casDestroySsoSessionsUrl, null, CasSsoSessionDestroyResponse.class, urlVariables);
				log.info(String.format("CAS response : %s",  casSsoSessionDestroyResponse.
toString()));
				message += casSsoSessionDestroyResponse.toString() + "\n";
			}
		}
		return message;
	}

}
```

Tickets Registry

Cf ci-dessus (paragraphe sur le RememberMe), on souhaite proposer à nos utilisateurs la possibilité de conserver leur session CAS sous mobile durant 2 semaines (14 jours, soit 1209600 secondes).
Sans rememberme de demandé, on propose une conservation de session CAS comme donné par défaut : c'est à dire 8 heures (28800 secondes).

Celà correspond à mettre en oeuvre le remember-me ou long term authentication tel que décrit ici :

https://apereo.github.io/cas/6.0.x/installation/Configuring-LongTerm-Authentication.html

Le TGT peut donc alors être conservé pendant 2 semaines si l'utilisateur active cette fonctionnalité.
Un TGT prenant une "certaine place" à la persistence, il faut alors que le ticket registry, c'est à dire le méanisme de persistence des tickets, permette un tel usage.

La documentation donnée ci-dessus indique :

```
A security policy that requires that long term authentication sessions MUST NOT be terminated prior to their
natural expiration
would mandate a ticket registry component that provides for durable storage, such as the JPA Ticket Registry.
```

Nous nous sommes donc dans un premier temps dirigé vers l'usage d'un ticket registry jpa, en utilisant postgresql.

En test nous n'avons pas eu à nous en plaindre. En production, celà a bien fonctionné dans un premier temps, mais rapidement ce choix s'est avéré problématique, cf ci-dessous.

JPA (PostgreSql)

Nous avons configuré le ticket registry JPA + Postgresql de manière usuelle.

Pour implémenter ce que nous souhaitons faire, nous avons mis les propriétés suivantes pour les temps de conservation des tickets :

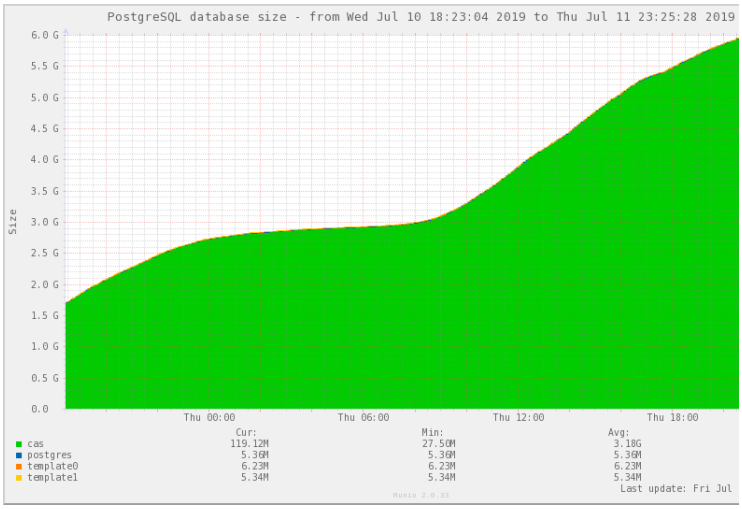
```
cas.ticket.tgt.maxTimeToLiveInSeconds=28800
cas.ticket.tgt.timeToKillInSeconds=28800
cas.ticket.tgt.rememberMe.timeToKillInSeconds=1209600
```

Passé en production, pas de problème de temps de réponse, du point de vue de l'utilisateur tout fonctionne parfaitement.

Problème

Mais nous avons constaté que :

- chaque TGT prenait 'beaucoup' de place, d'une certaine manière celà nous a conforté dans un premier temps aux choix de JPA et donc de la base de données pour la persistence des tickets.
- la base de données ne faisait qu'augmenter ... et donc la purge des tickets ne se faisait vraissemblablement pas correctement - cf ce graphe munin de 24 heures d'exploitation en production :



En regardant les logs CAS, on trouve l'erreur récurrente suivante (ceci est un extrait : on n'a conservé que les lignes 'intéressantes') :

```
2019-07-10 03:32:58,876 ERROR [org.hibernate.engine.jdbc.batch.internal.BatchingBatch] - <HHH000315: Exception
executing batch [org.hibernate.StaleStateException: Batch update returned unex\
pected row count from update [0]; actual row count: 0; expected: 1], SQL: update TICKETGRANTINGTICKET set
NUMBER_OF_TIMES_USED=?, CREATION_TIME=?, EXPIRATION_POLICY=?, EXPIRED=?, LAST_TIME_\
USED=?, PREVIOUS_LAST_TIME_USED=?, AUTHENTICATION=?, DESCENDANT_TICKETS=?, PROXIED_BY=?,
PROXY_GRANTING_TICKETS=?, SERVICES_GRANTED_ACCESS_TO=?, ticket_Granting_Ticket_ID=? where ID=?>
2019-07-10 03:32:58,876 ERROR [org.apereo.cas.ticket.registry.DefaultTicketRegistryCleaner] - <Batch update
returned unexpected row count from update [0]; actual row count: 0; expected: 1>
javax.persistence.OptimisticLockException: Batch update returned unexpected row count from update [0]; actual
row count: 0; expected: 1
.......
        at org.apereo.cas.ticket.registry.JpaTicketRegistry.deleteTicketGrantingTickets(JpaTicketRegistry.java:
198) ~[cas-server-support-jpa-ticket-registry-6.0.4.jar:6.0.4]
.......
        at com.sun.proxy.$Proxy152.deleteTicket(Unknown Source) ~[?:?]
        at org.apereo.cas.ticket.registry.DefaultTicketRegistryCleaner.cleanTicket(DefaultTicketRegistryCleaner.
java:78) ~[cas-server-core-tickets-api-6.0.4.jar:6.0.4]
.......
        at org.apereo.cas.ticket.registry.DefaultTicketRegistryCleaner.cleanInternal
(DefaultTicketRegistryCleaner.java:65) ~[cas-server-core-tickets-api-6.0.4.jar:6.0.4]
        at org.apereo.cas.ticket.registry.DefaultTicketRegistryCleaner.clean(DefaultTicketRegistryCleaner.java:
45) ~[cas-server-core-tickets-api-6.0.4.jar:6.0.4]
.......
```

Suivi du message :

```
2019-07-10 03:32:58,877 ERROR [org.apereo.cas.config.CasCoreTicketsSchedulingConfiguration] - <Transaction
silently rolled back because it has been marked as rollback-only>
org.springframework.transaction.UnexpectedRollbackException: Transaction silently rolled back because it has
been marked as rollback-only
```

Les considérations qui suivent font suite à une analyse du code et des recherches sur internet (dont forums CAS).

Analyse du problème

Le DefaultTicketRegistryCleaner est en charge de nettoyer les tickets expirés.

Pour ce faire il lance une méthode clean qui est transactional et qui va donc se charger dans une seule et même transaction de regarder tous les tickets pour nettoyer ceux qui sont expirés.

Le nettoyage d'un ticket ne consiste pas seulement à supprimer un ticket, il consiste aussi à supprimer les tickets potentiellement enfants ou/et à spécifier que le ticket est expiré ou/et à mettre à jour les tickets parents pour indiquer qu'un enfant est expiré, etc.

Entre nos chainages de proxycas, de clearpass, ... on comprend du log ci-dessus que pour un ticket, le nettoyage ne passe pas : on tente de mettre à jour un ticket qui est en fait supprimé.
Cependant si l'opération de suppression est effectivement dans la même transaction, ça doit passer malgré tout, on en déduit 2 choses distinctes :

- que celà fait suite sans doute à un bug antérieur, notre base n'est plus consistante
- le fait que DefaultTicketRegistryCleaner n'utilise qu'une seule et même transaction doit permettre qu'un update sur un ticket sur lequel on a appelé un delete avant ne pose justement pas de pb (pure spéculation ici, l'usage d'une seule transaction pour nettoyer n tickets laisse perplexe).

Vu qu'il y a une erreur sur un nettoyage d'un ticket, c'est toute la transaction qui est avortée.
Et donc aucun ticket ne peut être nettoyé.
D'où le nombre de tickets qui ne fait qu'augmenter en base.

Remarques sur l'implémentation de la persistence JPA

En tentant de débloquer la situation, on a remarqué les choses suivantes :

- la persistence d'un tgt en jpa correspond à une table dont 6 colonnes sont des blobs, et donc des Large Object dans PostgreSql et donc des fichiers sur le système de fichiers
- malgré celà, la suppression d'un ticket n'entraine pas la suppression des Large Object : pour y remédier, il faudrait alors mettre en place un trigger ou lancer rgulièrement un vacuumlo

A y regarder de plus près, ces stockages sous forme de blob correspondent une logique de programmation où on stocke de la donnée non structurée ~ 'no-sql'.

L'implémentation DefaultTicketRegistryCleaner.java nous conforte dans celà : pour récupérer les tickets qui doivent être supprimés, on récupère tous les tickets puis on regarde ceux qui doivent être expirés.

Dans une logique sql, on récupèrerait directement par un select uniquement les tickets que l'on doit supprimer.

Enfin la taille de la base de données, et la taille prise donc par la persistence d'un ticket dans postgresql est justifiée par cet usage des Large Object ; du coup celà prend une place relativement conséquente ; l'ordre de grandeur est environ 3GB pour 30000 TGT ...

Conclusion : le ticketRegistry JPA à éviter

Celà nous amène à conclure que l'usage d'un ticket registry CAS en JPA/Postgresql est à éviter.

C'est ce qui est +/- indiqué dans la documentation ici : https://apereo.github.io/cas/6.0.x/ticketing/JPA-Ticket-Registry.html :

```
Usage Warning!
Using a relational database as the back-end persistence choice for ticket registry state management is a fairly
unnecessary and complicated process.
Unless you are already outfitted with clustered database technology and the resources to manage it, the
complexity is likely not worth the trouble.
```

Redis

Redis est une solution permettant de persister des données simples clef/valeurs. En ce sens elle correspond à cette logique no-sql qu'on a pu observer ci-dessus.

C'est a priori le ticket registry recommandé, et c'est celui que le script de l'AMU donnait par défaut.

Aussi lors de notre passage en production sur CAS 6.0.4, ayant quelques doutes avec le ticket registry en jpa/postgresql, nous avions préservé les configurations pour rebasculer sur du Redis en cas de problème, ce qui a donc été fait 2 jours après la mise en production de notre CAS 6.0.4.

Le passage sur Redis a donc été immédiat.
En production, on constate que le stockage (complètement en RAM ... avec un fichier de dump pour préserver les tickets lors des redémarrages) est en fait peu gourmand.

Simple d'usage, on peut interroger avec redis-cli le serveur redis et observer le fonctionnement : les tickets (clef/valeur) sont poussés par CAS avec une date d'expiration, cette date d'expiration étant ensuite gérée en interne par redis lui-même.

Cette durée / date d'expiration correspond au pramètre cas.ticket.tgt.maxTimeToLiveInSeconds - paramètre que nous n'avions en fait pas compris lors de la mise en place de CAS avec JPA / Postgresql (une base SQL comme postgresql ne propose pas ce mécanisme d'expiration de la donnée).

Ainsi nous mettons finalement comme péramétrage :

```
cas.ticket.tgt.maxTimeToLiveInSeconds=1209600
cas.ticket.tgt.timeToKillInSeconds=28800
cas.ticket.tgt.rememberMe.timeToKillInSeconds=1209600
```

Avec ce paramétrage :

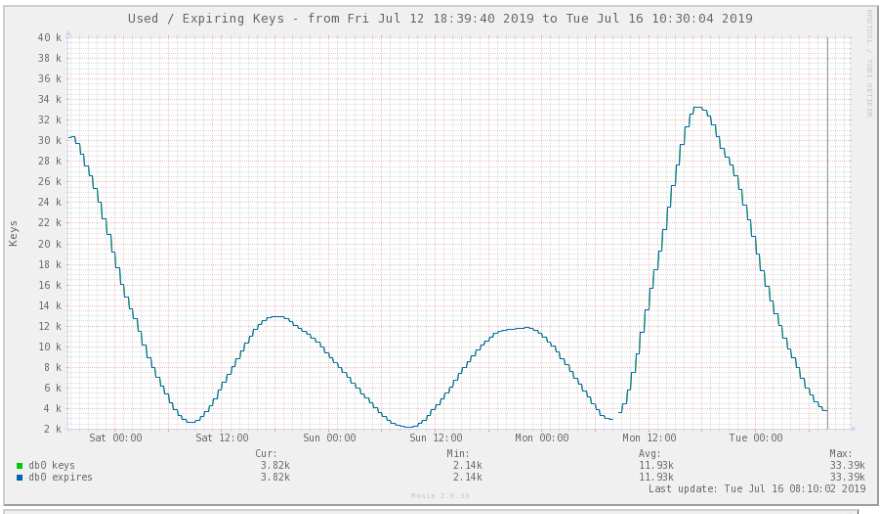
- tous les TGT expirent via les mécanismes redis internes au bout de 1209600 secondes (2 semaines)
- le DefaultTicketRegistryCleaner quant à lui va passer sur tous les tickets pour focer la suppression des tgt en fonction des paramètres timeToKillInSeconds au bout de 2 semaines ou 8 heures.

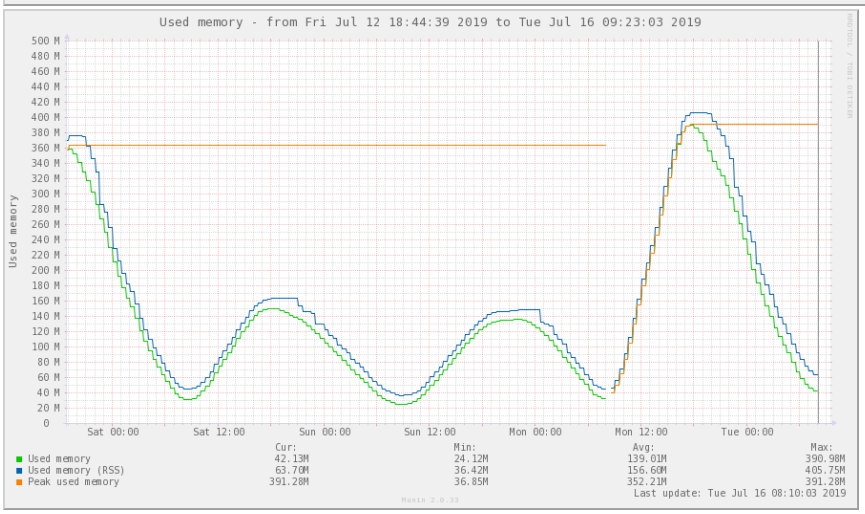
La souplesse et tolérance des commandes passées par redis font que ça ne bug pas, il n'y a pas de transaction, la récupération de l'ensemble des données pour un tri a postériori convient également à cet usage redis.

De même le stockage en ram comme en fichier (fichier de dump) requiert donc finalement peu d'espace. Là où nous avions besoin d'environ 3GB pour 30.000 TGT, c'est 300MB de RAM qu'il nous faut.

Les graphes suivants montrent sur 3 jours l'usage redis en nombre de tickets et RAM utilisée.

A noter que le lundi (15 juillet) on recense environ 15000 visites sur le CAS pour 10000 visiteurs différents et un total de 170000 tickets créés.
On a en fait usuellement 2 fois plus de fréquentation un jour hors période de vacances scolaires.
Rappel également pour donner un ordre de grandeur : notre université accueille environ 30.000 étudiants.





En imaginant que le service soit effectivement utilisé par quelques milliers de personnes depuis leurs mobiles (quasimenet 1 visite sur 2 sur notre CAS se fait actuellement depuis un mobile), on peut imaginer des pics de consommation de RAM de 3GB de RAM pour Redis.

Proposer 14 jours de persistence de TGT sur les mobiles pour les utilisateurs qui choissent l'option 'se souvenir de moi' est à portée avec le ticket registry sous Redis.

Conclusion

- Le ticket registry via Redis donne toute satisfaction.
- Le ticket registry via jpa/postgresql a peut-être été un choix possible dans les anciennes versions de CAS ; aujourd'hui celà parait être un choix très risqué ... il est à déconseiller dans le sens où il ne correspond pas à la logique de développement actuel de CAS.

Misc

Via des versions à jour de spring-boot et spring-security, CAS en 6.0.x amène l'usage des dernières technologies web, notamment en matière de sécurité (csp, xss, csrf, xsrf ...) et de protocole (chunked transfer encoding).

Iframe

Ainsi l'authentification d'une de nos applications ne fonctionnait plus après le passage sur notre nouveau CAS car celle-ci intègre le CAS dans une iframe. En attendant de corriger celà côté de l'application, on a ajusté (temporairement donc) la chose pour que ça passe à nouveau : suppression de l'entête http X-Frame-Options en positionnant simplement la règle suivante sur notre Apache qui fait office de proxy avec notre Tomcat CAS :

```
Header unset X-Frame-Options
```

Chunked transfer encoding

On conserve à l'URN une version ancienne 2.x.y de notre webmail SOGo (en plus de la nouvelle version SOgo 4 qui n'a posé aucun problème avec le nouveau CAS) qui est cassifié.

Cf https://sogo.nu/bugs/view.php?id=2408&nbn=22 cette version est sensible aux réponses CAS et s'attend à avoir dans son dialoguqe ProxyCAS, une réponse HTTP proposant simplement le contenu XML avec un Content-Length correspondant.

Ici, le nouveau CAS utilise maintenant le 'Chunked transfer encoding' qui n'est pas du tout apprécié par cette ancienne version de SOGo ; pour continuer de faire fonctionner le proxy-cas avec ce vieux SOGo (en fin de vie à l'URN malgré tout), on a reproduit un proxy (cgi python sur el apache de CAS) spécifique pour ce SOGo pour l'accès aux urls CAS en /proxy et /serviceValidate :

```
#!/usr/bin
/python


import os
import cgi
import urllib2

def pageget(url):
    req = urllib2.Request(url)
    rep = urllib2.urlopen(req)
    data = rep.read()
    return data

if __name__ == '__main__':
     url = os.environ["REQUEST_URI"]
     content = pageget('https://cas.univ-ville.fr' + url)
     print "Status: 200"
     print "Content-Length: " + str(len(content)+1)
     print
     print content
```

pam_cas

La cassification de notre serveur imap (proxy cas du webmail) utilise esup_pam_cas.

Nous avons paramétré notre CAS pour qu'il propose aux applications cassifiées certains attributs dans l'attribut multi-valué memberOf. Utilisateurs de grouper, le memberOf est conséquent pour certains utilisateurs.

De fait nous nous sommes retrouvés avec des erreurs sur le pam_cas du type :

```
Unexpected read error or response too large from ....
```

Cf https://github.com/EsupPortail/esup-pam-cas/blob/master/sources/cas_validator.c la réponse xml du cas dépassant 4096 caractères pour certains utilisateurs qui sont membres de nombreux groupes et ont donc un memberOf conséquent.

Aussi nous avons limité les attributs donnés dans cas.authn.attributeRepository.defaultAttributesToRelease pour ne présenter le memberOf qu'aux applications en http/https (de notre domaine) excluant le serveur imap qui n'a en fait besoin que de l'uid de l'utilisateur pour authentifier ce dernier par proxy-cas.