



Rapport Projet Software Security

BEUREL Simon
MAUROIS Quentin
FROMENT Lorenzo

Phase de spécification :

Lors de ce projet de développement de l'application de transfert sécurisé de fichiers SecTrans, une des contraintes imposées était d'utiliser les bibliothèques de Macrohard Corporation. Cependant, l'un des gros problèmes de ces bibliothèques, c'est qu'elles ne possèdent aucune implémentation sécurisée, et par conséquent, aucun concept de cybersécurité ne peut se retrouver dedans.

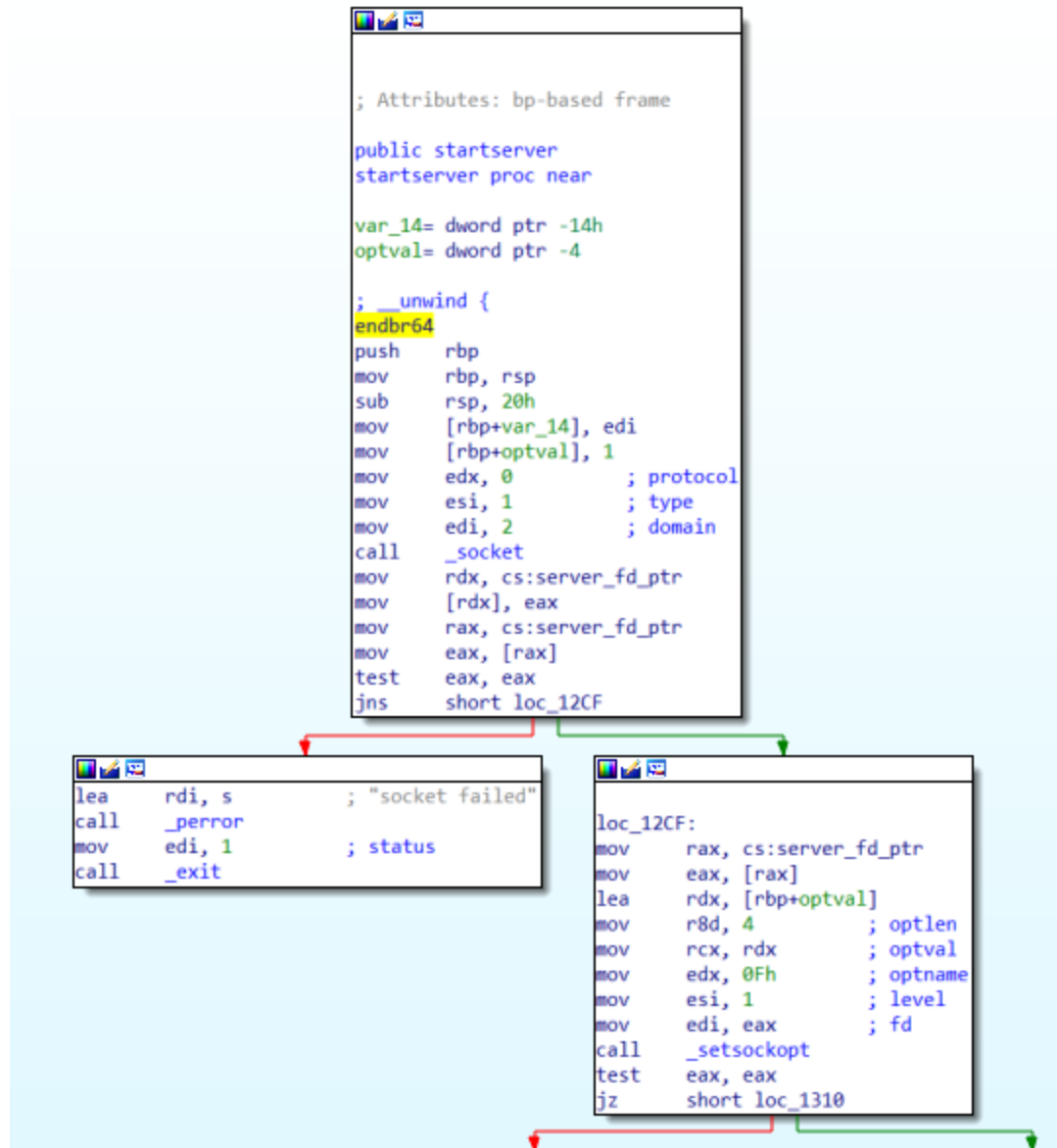
Dans un premier temps, notre équipe a décidé d'implémenter un MVP (Minimum Valuable Product) de l'application SecTrans sans intégrer aucun principe de cybersécurité, puis dans un second temps, nous avons décidé de réaliser une analyse poussée des différentes failles qui pouvaient impacter notre application. Le but de cette analyse est notamment de pouvoir dégager de nombreux principes de cybersécurité importants manquants, et ainsi de les intégrer au MVP de l'application SecTrans dans le but de produire une application sécurisée pour les différents usagers. Voici les différents points que notre étude a révélée :

I - Rétro-conception et analyse statique :

Problème :

L'un des premiers points soulevés par notre étude, est le fait que les différents fichiers notamment fournis par l'entreprise Microhard peuvent être sujets à de la rétro-conception (Reverse Engineering) par des attaquants. Ce problème est très important, car grâce à cette analyse statique, un attaquant va pouvoir analyser le code qui est exécuté pour réaliser le transfert de fichiers, l'ouverture du serveur etc... et ainsi avoir une connaissance approfondie du comportement de notre application.

Par exemple, comme le montre cette capture d'écran, il est très simple de pouvoir désassembler le code dans un outil comme IDA ce qui nous donne par exemple pour la fonction `startserver()` de la bibliothèque `libserver.so` ceci (la capture d'écran est coupée car sinon ce serait trop long) :



Mais également, on peut encore aller plus loin car on peut générer le code original et ainsi retrouver encore plus précisément le comportement de la fonction. Par exemple, avec le site <https://dogbolt.org/> nous pouvons dé-compiler les différents fichiers et ainsi retrouver le code original des différents fichiers .so, ainsi par exemple, pour la fonction startserver() nous pouvons en conclure que son code original est :

```
void startserver(uint16_t param_1)
{
    int iVar1;
    undefined4 local_c;
    local_c = 1;
    server_fd = FUN_001011b0(2,1,0);
    if (server_fd < 0) {
        perror("socket failed");
        // WARNING: Subroutine does not return
        exit(1);
    }
    iVar1 = setsockopt(server_fd,1,0xf,&local_c,4);
    if (iVar1 != 0) {
        perror("setsockopt");
        // WARNING: Subroutine does not return
        exit(1);
    }
    address = 2;
    DAT_001040b4 = 0;
    DAT_001040b2 = htons(param_1);
    iVar1 = bind(server_fd,(sockaddr *)&address,0x10);
    if (iVar1 < 0) {
        perror("bind failed");
        // WARNING: Subroutine does not return
        exit(1);
    }
    iVar1 = listen(server_fd,3);
    if (iVar1 < 0) {
```

```
        perror("listen");  
        // WARNING: Subroutine does not return  
        exit(1);  
    }  
    return;  
}
```

Solution :

Pour pouvoir résoudre ce problème, une des techniques utilisées est l'obfuscation de code. Ce terme technique désigne le fait que l'on va envoyer notre fichier source .c en entrée d'un algorithme, et que ce dernier va nous donner en sortie un nouveau fichier .c dont le code sera devenu illisible pour un être humain normal. Grâce à cette technique, si un attaquant arrive à décompiler notre exécutable, il pourra avoir de grosses difficultés pour retrouver le code original. De plus, l'obfuscation de code peut également générer des branchements inutiles dans le code, c'est derniers apportent une grosse plus value car comme nous l'avons montré précédemment, il est possible grâce à des logiciels comme IDA de pouvoir générer un arbre qui montre les différentes branchements réalisés dans notre exécutable. Ainsi, grâce à la technique de l'obfuscation, il est donc possible de "fausser" la vision de l'attaquant.

Malheureusement ici nous n'avons pas réussi à l'implémenter dans notre projet SecTrans, mais nous avons trouvé des outils des intéressants :

<https://github.com/scrt/avcleaner>

<https://github.com/weak1337/Alcatraz>

<https://github.com/obfuscator-llvm/obfuscator>

Une alternative envisageable consiste à utiliser l'argument -O3 lors de la compilation et de la génération de nos fichiers exécutables. En spécifiant cet argument, nous indiquons au compilateur notre souhait d'obtenir un fichier compact tout en améliorant son efficacité. Cette option rend également le reverse engineering plus complexe.

Il est également intéressant de rappeler que l'un des points absolument cruciaux pour éviter qu'un attaquant puisse dérober des informations en réalisation du reverse engineering, est le fait de ne laisser aucun mot de passe etc... écrits en clair dans notre

code, car si ce dernier arrive à retrouver le code original, alors cela pourrait compromettre la sécurité de l'entreprise.

II - Les attaques DDOS :

Problème :

Les attaques DDOS peuvent être très endommageantes pour notre application. Dans notre cas, l'application SecTrans se base sur un principe de requête entre le client et le serveur. Si un attaquant veut tester le comportement de l'application, il pourrait avec l'aide d'une armée de bots, envoyer plusieurs millions/milliards de requêtes à notre serveur, ce qui pourrait potentiellement le mettre hors service.

Il est donc très important de prendre en compte cette attaque, et de se préparer à cette éventualité, car notamment si le serveur venait à s'éteindre, cela pourrait compromettre les conditions de travail des différents employés utilisant l'application SecTrans.

Solution:

Malheureusement, nous ne pouvons pas avoir accès à l'infrastructure dans laquelle notre application sera mise en place, c'est pour cela que pour essayer de contrer les attaques DDOS, nous avons défini un nombre limité de requêtes qu'un utilisateur peut réaliser par minute. Cette limite, qui est de 10 requêtes autorisées par minute, permet de contrer les attaques DDOS, mais ce n'est pas la seule chose que l'on peut mettre en place. En effet, cela sort du domaine du développement de SecTrans, mais il serait judicieux de mettre en place une analyse continue des requêtes effectuées sur l'application pour observer si une activité anormale se déroule, pour ainsi essayer de détecter ces attaques DDOS. Par exemple, si on détecte que chaque jeudi à 12h, le logiciel SecTrans traite en moyenne 2 000 requêtes, et que un jeudi particulier il en traite 150 000, alors on peut penser qu'une attaque DDOS est en cours, et qu'il est nécessaire de réaliser une sauvegarde des données présentes sur le serveur.

III - Cryptographie :

Problème :

L'un des points les plus importants en ce qui concerne la sécurité de l'application SecTrans est la cryptographie. En effet, comme nous l'avons dit précédemment, notre application est une application de **transfert sécurisé de fichiers** entre un client et un serveur. Ainsi, un des points très importants qu'il faut implémenter est un système permettant de crypter les données et de les décrypter pour éviter de potentielles attaques. Voyons quelques scénarios d'attaques qui pourraient arriver dans notre MVP de SecTrans :

- 1) Actuellement, aucun protocole de cryptographie n'est utilisé. Les fichiers sont donc stockés en clair dans la base de données de notre serveur. Imaginons que Monsieur le président Emmanuel Macron envoie sur le serveur un fichier texte comportant les codes de la bombe nucléaire, ces codes seront donc stockés en clair dans notre base de données. Malheureusement, si un attaquant arrive à avoir accès à notre base de données, il pourra donc lire tous les fichiers présents et comme ces derniers sont stockés en clair, il aura donc accès à toutes les données et connaîtra donc les codes de la bombe nucléaire...
- 2) Dans ce deuxième scénario, nous allons encore prendre l'exemple du président de la république qui envoie les codes de la bombe nucléaire sur le serveur. Cette fois, nous allons établir l'hypothèse que le président envoie les codes en clair, mais qu'une fois ces derniers arrivés sur le serveur, ils se font chiffrés et sont donc illisibles dans la base de données. Cependant, une attaque possible peut consister à avoir un attaquant qui analyse toutes les données envoyées par le président au serveur. Si notre attaquant arrive à avoir accès aux échanges réseau entre le président et le serveur, il pourra analyser les trames réseau avec un logiciel comme Wireshark, et comme le président envoie en clair ses données, il pourra donc récupérer les données en clair avant que celles-ci soient cryptées.

Solution :

Pour pouvoir corriger ce problème, nous avons mis en place un système de chiffrement des données grâce à un protocole de chiffrement symétrique : **AES 256 CBC**. Le principe mis en place est très simple :

- Quand vous créez une connexion, une clé est automatiquement générée (au format AES 256). La génération de cette clé s'effectue grâce au périphérique spécial `"/dev/urandom"` qui s'assure de fournir un flux de données pseudo-aléatoire. Si vous souhaitez envoyer un fichier sur le serveur, alors SecTrans va d'abord chiffrer le contenu du fichier en fonction de votre clé, puis il va envoyer le fichier sur le serveur ainsi que la clé correspondante.
- Si vous souhaitez récupérer un fichier présent sur le serveur, alors le serveur vous enverra le fichier chiffré, mais également la clé correspondante permettant de pouvoir déchiffrer le fichier.

Grâce à cette implémentation, on peut garantir le fait que les fichiers présents sur le serveur soient chiffrés ce qui empêche par conséquent qu'un attaquant ayant accès au serveur puisse comprendre le contenu des fichiers, mais également on peut garantir le fait que si une personne intercepte un échange réseau, alors il ne puisse pas lire le contenu du fichier.

Il est important de noter qu'ici, nous n'avons intégré que cette solution dans ce projet, mais il existe d'autres solutions, comme utiliser une fonction de hachage pour garantir à l'utilisateur qui télécharge un fichier, que le contenu de ce dernier est bien celui qui a été envoyé originellement. Il est également possible d'implémenter les échanges de clés à l'aide du protocole TLS (Transport Layer Security) qui pourrait permettre un échange sécurisé entre le serveur et le client.

IV - Buffer Overflows et gestion des entrées :

Problème :

Ce nouveau point découvert par notre étude et qui n'est pas négligeable est la vérification des différentes entrées réalisées par le client. En effet, selon la librairie de MicroHard, le client fournit des entrées de taille 1024 et le serveur retourne des sorties

également de taille 1024, il faut donc s'assurer qu'un client ne puisse pas réaliser un buffer overflow ce qui pourrait compromettre la sécurité de l'application.

De plus, il faut également partir du principe qu'un attaquant pourrait essayer de réaliser de l'exécution de code malveillant quand il voudra envoyer un message. Il est donc nécessaire de bien vérifier les entrées en utilisant les bonnes fonctions C pour éviter que ce code malveillant soit exécuté

Solution :

Pour pouvoir résoudre ce problème, nous nous sommes intéressés réaliser deux implémentations :

- La première, consiste à rajouter des vérifications sur chacune des entrées envoyées par l'utilisateur, dans le but de vérifier si les entrées ont une taille comprise entre 0 et 1024. Cette vérification est nécessaire car dans la librairie de Microhard, les fonctions permettant d'envoyer et de recevoir des messages prennent en paramètres des tableaux de 1024 caractères. Ainsi, grâce à cette vérification, les attaques de type Buffer Overflow sont beaucoup plus dure à réaliser pour un attaquant.
- La deuxième, consiste à vérifier chacune des entrées envoyées par l'utilisateur en l'analysant grâce à une regex. Cette regex autorise les entrées qui correspondent **SEULEMENT** à des commandes connues du système comme "sectrans -list" etc... Ainsi, grâce à cette vérification grâce à la regex, on établit une whitelist d'entrée, c'est-à-dire une liste d'entrées "valides" pour notre système.

Il est important de noter que pour réaliser la gestion des entrées, nous aurions pu créer une blacklist, c'est-à-dire une liste d'entrées "non valides" pour notre système. Or, ces blacklists sont souvent très peu fiables car elles nécessitent de devoir se faire remettre à jour de manière régulière, à contrario de la whitelist.

V - Gestion des accès :

Problème :

Un des derniers points notés par notre analyse de failles présentes dans ce projet, est notamment le fait que le serveur devrait être capable de pouvoir gérer les différents

accès aux fichiers qu'il contient. Par exemple, un étudiant de l'école Polytech Nice Sophia ne devrait pas pouvoir avoir accès aux codes nucléaires sur le serveur du président de la République...

Solution :

Pour résoudre ce problème, la solution la plus logique (et qui est celle que nous avons implémentée) est de réaliser un système d'authentification au système. Ainsi, quand un client lance le programme, on lui demande son nom d'utilisateur et son mot de passe. Une fois ces derniers rentrés, le système regarde si une correspondance existe, et si c'est le cas, le client peut donc utiliser le programme, sinon, son exécution s'arrête.

Notre système d'authentification implémenté est assez simple, mais théoriquement nous pouvons imaginer de nouveaux ajouts pour pouvoir renforcer la sécurité comme :

- Créer une vraie base de données contenant les paires nom d'utilisateur / mot de passe
- Stocker les mots de passe et noms d'utilisateur en utilisant un chiffrement symétrique comme AES dans le but de rendre ces données non lisibles pour un humain.
- Attribuer à chaque compte, une autorisation de lecture des fichiers présents sur l'application. Cela permettrait notamment de pouvoir déployer SecTrans à grande échelle dans plusieurs entreprises, sans forcément que l'employé de l'entreprise A puisse avoir accès aux fichiers de l'entreprise B.

Un des derniers points qui pourrait être implémenté pour garantir l'accès à l'application serait de déployer notre serveur sur le réseau privé de l'entreprise, et de lire les messages en provenance de clients que si ces derniers sont présents sur le réseau de l'entreprise. Ainsi, grâce à cette solution, on s'assure que seuls les membres de l'entreprise sont capables de pouvoir accéder au serveur. **Cette solution fonctionne seulement si l'on souhaite restreindre l'utilisation de SecTrans aux employés présents dans l'entreprise.**

VI - Conclusion :

Après cette analyse, nous avons donc pu conclure que les 5 points cités précédemment sont trois points très importants qui peuvent compromettre la sécurité de notre

application SecTrans. Il est donc important d'intégrer au cœur de notre MVP, différentes techniques pour sécuriser et contrer ces 3 points.

Nous avons également pu voir les différentes solutions que nous avons apportées pour essayer de rendre l'application SecTrans la plus sécurisée possible.

Phase d'analyse de sécurité :

Maintenant que la phase de développement du projet à été réalisée, et que nous avons détecté puis implémenté des solutions de sécurité sur notre projet, nous pouvons réaliser la phase d'analyse de sécurité de notre projet. Lors de cette phase d'analyse, le but sera de se mettre dans la peau d'un utilisateur malveillant souhaitant compromettre la sécurité de notre application. Lors de cette analyse, nous réaliserons 3 points :

- Une analyse statique : Nous allons dans cette partie essayer de réaliser un "reverse engineering" sur notre application pour essayer de voir si nous arrivons à comprendre le fonctionnement de notre application
- Une analyse réseau : Dans cette partie, nous allons essayer d'analyser les échanges réseau réaliser par le client et le serveur, pour détecter si nous arrivons à intercepter des messages
- Une analyse dynamique : Le but de cette partie sera de réaliser une analyse dynamique notamment à l'aide du fuzzing, pour détecter si certaines entrées sont compromettantes pour notre application.

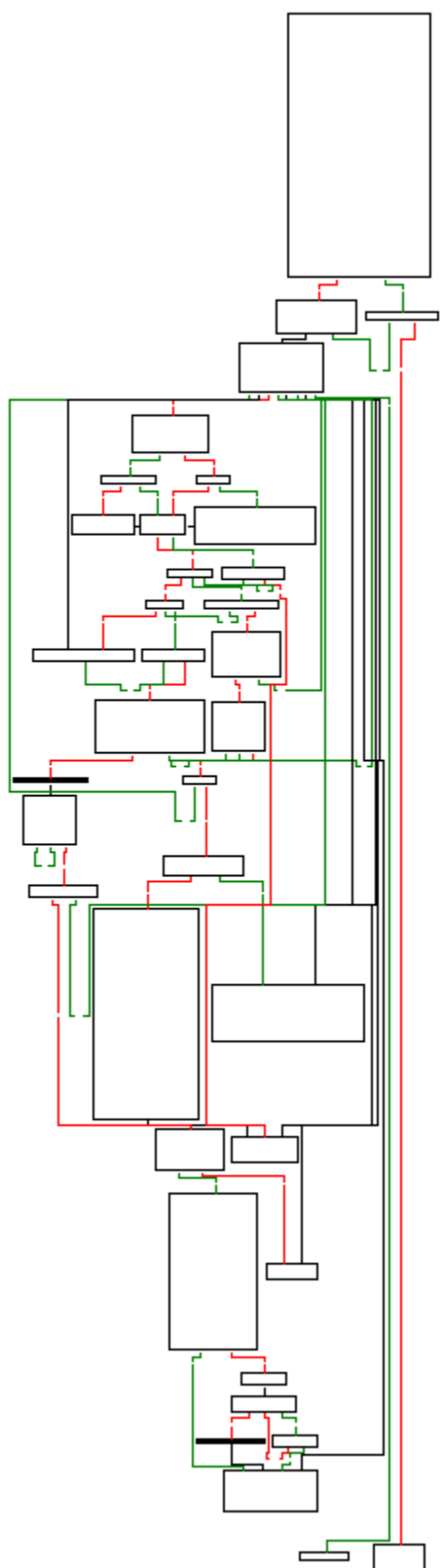
I - Analyse statique :

Comme mentionné dans la partie précédente, il est possible de réaliser une analyse statique de notre programme, notamment grâce au "reverse engineering". Le but ici, va être de travailler seulement sur l'exécutable, et d'oublier tout le travail de développement réalisé en amont. Nous devons nous mettre dans la peau d'une personne malveillante souhaitant attaquer le service SecTrans.

Pour cela, nous allons utiliser le logiciel IDA, qui est un désassembleur très connu dans le domaine du reverse-engineering. Nous allons d'abord nous intéresser à l'exécutable correspondant au Client.

Analyse du code de notre Client :

Tout d'abord, l'une des premières étapes importante dans l'analyse de notre Client est d'analyser le graphique affiché par le logiciel IDA :



Comme nous pouvons le voir ici, ce graphique qui correspond à l'exécution de notre programme (fonction main et donc aux différentes branches/appels réalisés) est assez imposant, et également il possède plusieurs branchements qui font que la compréhension de ce dernier est très difficile. Cependant, certaines parties de ce graphique peuvent être comprises, par exemple sur l'image ci-dessous, on remarque que beaucoup de branchements arrivent à nouveau au point de départ, on peut donc facilement en conclure que cela correspond à une boucle while(1) qui va analyser les entrées de l'utilisateur à chaque fois et que l'on reviendra à ce point de départ après chaque commande écrite.



Maintenant, nous pouvons également essayer d'analyser le pseudo-code généré par IDA pour cette fonction main. Une fois la génération du pseudo-code réalisée, on se rend compte que malgré que le code soit assez compacte et dès fois difficilement compréhensible comme sur la capture d'écran ci-dessous,

```

}
if ( *(_QWORD *)v37 == 0x736E617274636573LL && *(_DWORD *)&v37[8] == 1886727456 )
    break;
if ( *(_QWORD *)v37 == 0x736E617274636573LL && *(_QWORD *)&v37[6] == 0x6E776F642D20736E )
{

```

il est quand même possible de pouvoir comprendre certaines instructions car le nom des fonctions sont des noms connus. Par exemple, sur cette capture d'écran :

```

v24 = generate_key();
v25 = generate_key();
v23 = EVP_CIPHER_CTX_new();
v3 = EVP_aes_256_cbc();
EVP_EncryptInit_ex(v23, v3, 0LL, v24, v25);

```

On peut comprendre que ce bout de code nous donne beaucoup d'éléments sur les principes cryptographiques appliqués. Par exemple, la fonction `EVP_aes_256_cbc()` est une fonction connue de la bibliothèque OpenSSL (https://www.openssl.org/docs/man1.1.1/man3/EVP_aes_256_cbc.html) et par conséquent, un attaquant peut donc comprendre que le chiffrement se fait grâce à l'algorithme de chiffrement symétrique AES-256-CBC.

De plus, on peut observer sur cette capture d'écran que les différentes entrées utilisateur sont vérifiées par une fonction `check_input()` :

```

fgets(v37, 1024, stdin);
v37[strcspn(v37, "\n")] = 0;
if ( (unsigned int)check_input(v37) == 1 )
    exit(0);

```

En analysant le pseudo-code généré de cette fonction grâce à IDA, on observe quelque chose de très intéressant :







```

v1 = pcre_compile(
    "^sectrans -list$|^sectrans -up [a-zA-Z0-9_./]*$|^sectrans -down [a-zA-Z0-9_./]*$",
    0LL,
    &v7,
    &v6,
    0LL);
if ( v1 )
{
    v2 = v1;
    v3 = strlen(s);
    v4 = pcre_exec(v2, 0LL, s, v3, 0LL, 0LL, v8, 120LL);
    if ( v4 < 0 )
    {
        if ( v4 == -1 )
            puts(aLEntr);
        else
            printf("Erreur de correspondance de la regex %d\n", (unsigned int)v4);
    }
}

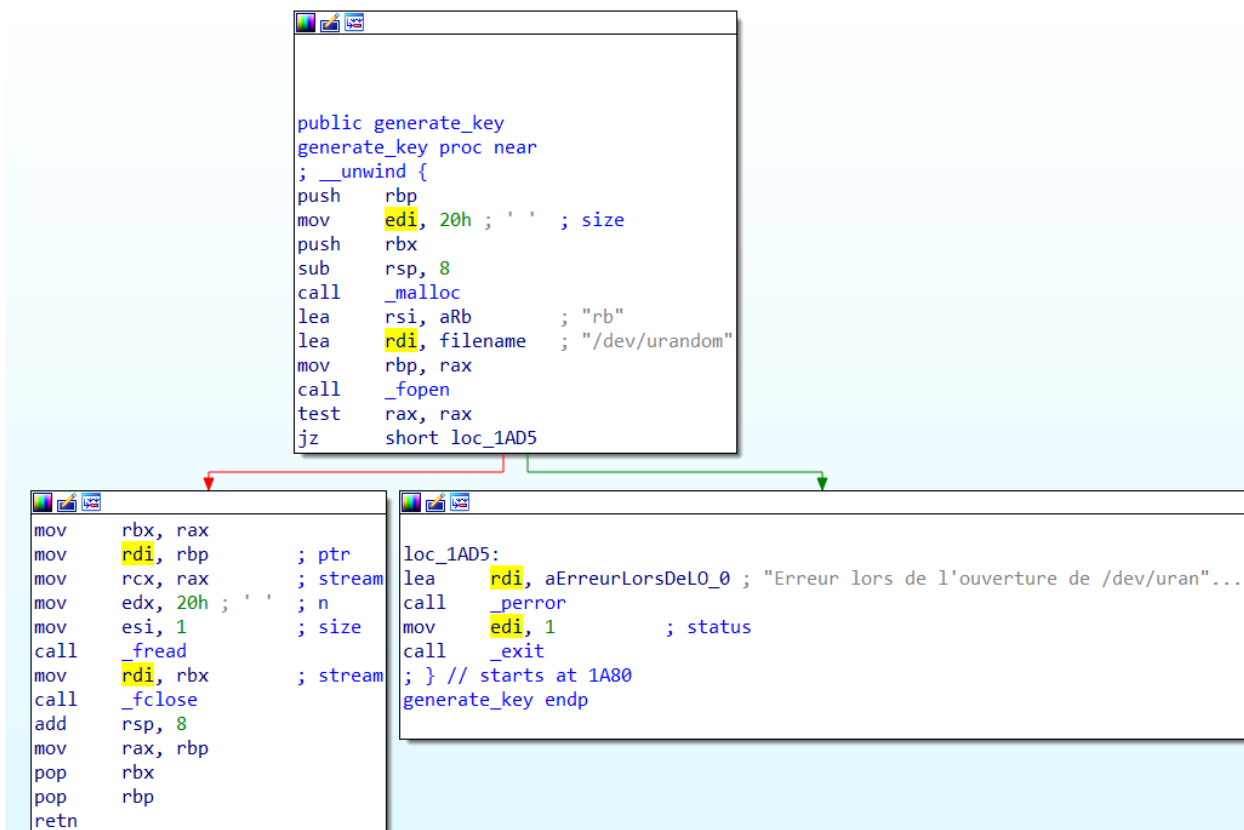
```

Et oui, vous l'aurez compris, en tant qu'attaquant, on peut donc savoir que les différentes entrées utilisateur sont vérifiées grâce à une regex présente dans la fonction *check_input()* de notre programme !

La dernière étape pour analyser le code de notre Client, est d'analyser les différents strings présente car ce tableaux peut nous permettre de vérifier notamment si par exemple des clés/mots de passe n'ont pas été laissés en clair dans le code. En analysant le tableau, on remarque quelque chose de très particulier :

	.rodata:0000... 00000013	C	me se met en pause
	.rodata:0000... 00000055	C	Erreur lors de l'ouverture du fichier, votre fichier n'existe pas dans le bon chemin
	.rodata:0000... 0000000D	C	/dev/urandom
	.rodata:0000... 00000007	C	Erreur
	.rodata:0000... 0000000F	C	sectrans -list
	.rodata:0000... 0000000D	C	sectrans -up

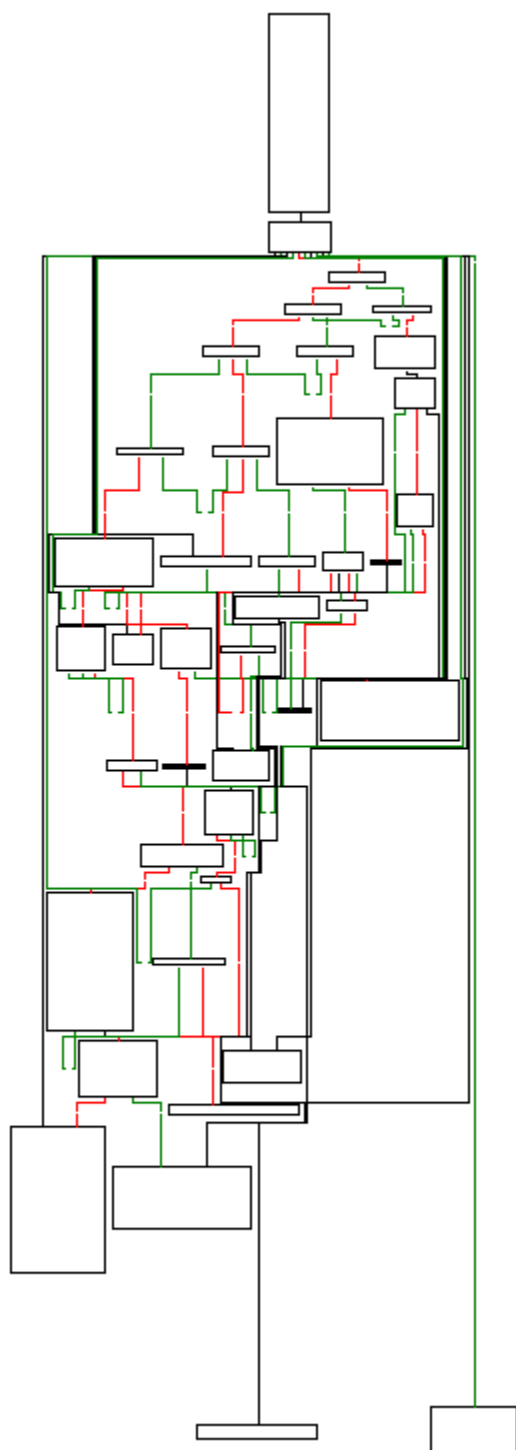
Pourquoi l'adresse */dev/urandom* est-elle mentionnée ? En nous intéressant de plus près on se rend compte que cette adresse est utilisée dans une fonction qui s'appelle *generate_key()* qui est la fonction qui va générer une clé pour l'AES-256. Ainsi, on peut donc comprendre que pour essayer de garantir un aléatoire dans la génération de clé, les développeurs ont décidé d'utiliser ce chemin.



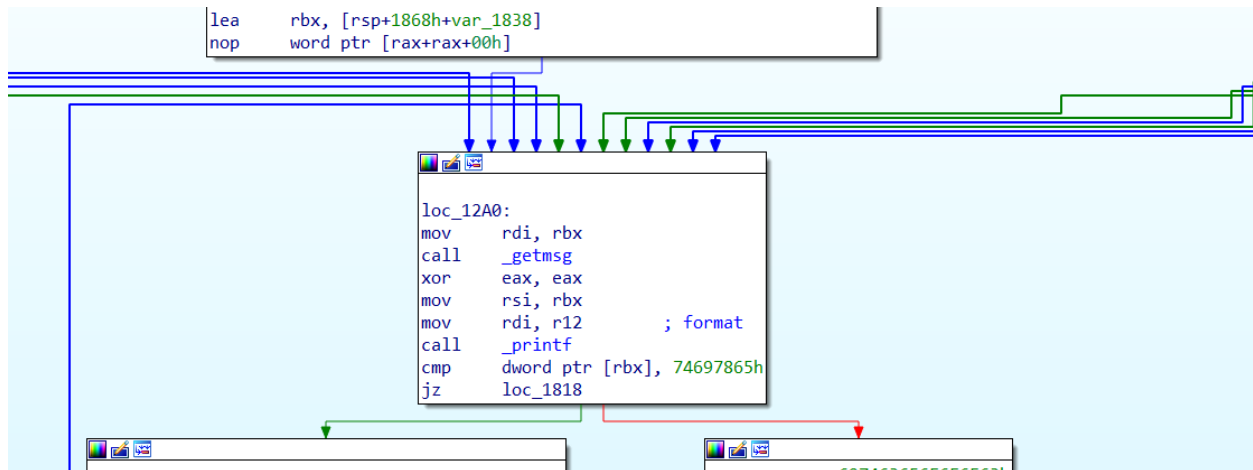
Maintenant, intéressons- nous à l'analyse statique de l'exécutable de notre Serveur.

Analyse du code de notre Serveur:

Comme pour l'analyse du Client, nous allons dans un premier temps générer le graphique correspondant à l'exécution du main de notre exécutable. Voici le résultat généré :



Comme vous pouvez le constater, ce graphe est assez complexe est le nombre de chemins possibles est très vaste. Cependant, comme vu lors de l'analyse du Client, on observe un "point de départ" central dans ce graphique, ce qui pourrait correspondre à une boucle while(1) qui s'exécute quand le serveur est activé, pour lire les différentes entrées envoyées par un client :



Maintenant, nous allons réaliser la génération du pseudo-code de cette fonction main et vérifier si des éléments sont compromettants pour la sécurité de notre application. Une fois cette génération réalisée, nous remarquons une faille assez importante.

```
v33 = (const char **)malloc(0x18uLL);
v4 = strdup("admin");
*v33 = v4;
s = v4;
v33[1] = strdup("admin");
```

Cette capture d'écran montre qu'au début de ce code, nous allouons de la mémoire pour créer un objet donc les attributs sont "admin" "admin", cela correspond donc probablement au compte administrateur écrit en dur dans notre code, ce qui est une très grosse faille !

!!\ ATTENTION /\ Cette faille à été laissé volontairement car elle est nécessaire pour le système d'authentification, dans l'idéal, il faudrait implémenter une vraie base de données ainsi que d'autres pratiques plus sécurisées (voir V - Gestion des accès à la page 8 du rapport)

Globalement, le pseudo-code généré par IDA est difficilement compréhensible, on comprend seulement que le serveur tourne selon ce principe :




- s'il reçoit la commande "sectrans -list", alors il récupère le nom de tous les fichiers et les envoie au client
- s'il reçoit la commande "sectrans -up XXX" alors il récupère et écrit les données envoyées et récupère également la clé associée au fichier pour le déchiffrer
- s'il reçoit la commande "sectrans -down XXX", alors il envoie le fichier et la clé correspondante au client.

Certaines parties de code peuvent-être assez compliquées pour un attaquant (notamment un attaquant de notre niveau) à déchiffrer, comme par exemple celle-ci :

```
*(__m128i *)v43 = _mm_load_si128((const __m128i *)&xmmword_2280);
*(__m128i *)&v43[16] = _mm_load_si128((const __m128i *)&xmmword_2290);
*(__m128i *)&v43[32] = _mm_load_si128((const __m128i *)&xmmword_22A0);
```

Mais il ne faut pas oublier, comme mentionné précédemment dans le rapport, que l'utilisation d'un outil d'obfuscation peut être très utilisé notamment pour pouvoir rendre ce code encore plus difficile à lire.

Lorsque nous affichons le tableau des strings présentes dans cet exécutable, nous retrouvons bien l'erreur "admin admin" mentionnée auparavant, mais également le chemin vers la base de donnée :

	.rodata:0000... 0000000F	C	sectrans -list
	.rodata:0000... 0000000B	C	./database
	.rodata:0000... 0000000D	C	sectrans -up

Encore une fois, ceci est dangereux mais par manque de temps nous n'avons pas eu le temps de réaliser une vraie base de données sur un vrai serveur, cela est un point d'amélioration si l'on veut mettre en production l'application dans le monde réel.

II - Analyse réseau :

Pour réaliser cette analyse réseau, nous allons utiliser un outil très connu et très puissant dans le domaine du réseau : Wireshark (<https://www.wireshark.org/>). Le but de

cette analyse sera d'essayer de récupérer les échanges réalisés entre un client et un serveur, et de regarder s'il est possible de pouvoir lire le contenu des fichiers envoyés. Si le contenu est impossible à lire, cela veut dire que le travail de l'équipe de développement a été bien fait 😊

Notre projet étant assez simple, nous avons le serveur qui est présent sur le port 8080 de notre machine, et notre client qui est présent sur le port 8081 de notre machine. On doit donc "écouter" les messages que s'échangent ces deux ports. Tout d'abord, la première étape est d'exécuter la commande tcpdump pour capturer tous les échanges réalisés et les sauvegarder dans un fichier pour ensuite ouvrir ce dernier avec Wireshark.

```
(s1o@PC-LENOVO)-[~]
$ sudo tcpdump -i lo 'port 8080 or port 8081' -w output.pcap
tcpdump: listening on lo, link-type EN10MB (Ethernet), snapshot length 262144 bytes
^C132 packets captured
264 packets received by filter
0 packets dropped by kernel
```

Avec Wireshark, on détecte rapidement les échanges entre le client et le serveur (ici quand le client envoie *sectrans -up empty_file.txt*) :

87 30.145835	127.0.0.1	127.0.0.1	TCP	93 58450 → 8080 [PSH, ACK] Seq=6 Ack=1 Win=65536 Len=27 TSval=730672619 TSecr=730672619 [TCP segment of a reassembled PDU]
88 30.145837	127.0.0.1	127.0.0.1	TCP	66 8080 → 58450 [ACK] Seq=1 Ack=33 Win=65536 Len=0 TSval=730672619 TSecr=730672619
89 30.145842	127.0.0.1	127.0.0.1	TCP	66 58450 → 8080 [FIN, ACK] Seq=33 Ack=1 Win=65536 Len=0 TSval=730672619 TSecr=730672619
90 30.145912	127.0.0.1	127.0.0.1	TCP	66 8080 → 58450 [FIN, ACK] Seq=1 Ack=34 Win=65536 Len=0 TSval=730672619 TSecr=730672619
▶ Frame 87: 93 bytes on wire (744 bits), 93 bytes captured (744 bits) ▶ Ethernet II, Src: Xerox_00:00:00 (00:00:00:00:00:00), Dst: Xerox_00:00:00 (00:00:00:00:00:00) ▶ Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1 ▶ Transmission Control Protocol, Src Port: 58450, Dst Port: 8080, Seq: 6, Ack: 1, Len: 27				
0000	00 00 00 00 00 00 00 00	00 00 00 00 00 00 45 00E:	
0010	00 4f 11 ba 40 00 00 06	2a ed 7f 00 00 01 7f 00	.n..@..*.....	
0020	00 01 e4 52 1f 90 9f ea	20 c2 31 32 c3 95 80 18	...R....12....	
0030	02 00 fe 43 00 00 01 01	08 0a 2b 8d 2d eb 2b 8d	...C....+...+	
0040	2d eb 73 65 63 74 72 61	6e 73 20 2d 75 70 20 65	--sectra ns -up e	
0050	6d 70 74 79 5f 66 69 6c	65 2e 74 78 74	mpty_fil e.txt	

Après analyse du fichier, on arrive enfin à retrouver le paquet contenant les données envoyées par le client. Le résultat est très convaincant car les données sont bien chiffrées et elles sont illisibles pour quelqu'un externe à l'application :

Frame 97: 124 bytes on wire (992 bits), 124 bytes captured (992 bits) Ethernet II, Src: Xerox_00:00:00 (00:00:00:00:00:00), Dst: Xerox_00:00:00 (00:00:00:00:00:00) Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1 Transmission Control Protocol, Src Port: 58458, Dst Port: 8080, Seq: 6, Ack: 1, Len: 58 [2 Reassembled TCP Segments (63 bytes): #95(5), #97(58)] [Frame: 95, payload: 0-4 (5 bytes)] [Frame: 97, payload: 5-62 (58 bytes)] [Segment count: 2] [Reassembled TCP length: 63] [Reassembled TCP Data: 3130323500889ad20a9448a7661713f015882f91dc6d7c0dee19339a40edf276a1756876113f9b841ffee3b30dc25860bf7323]				
0000	00 00 00 00 00 00 00 00	00 00 00 00 08 00 45 00E:	
0010	00 6e a2 8d 40 00 00 06	99 fa 7f 00 00 01 7f 00	.n..@..*.....	
0020	00 01 e4 5a 1f 90 fe b7	17 cf 22 3c 7f 30 80 18	...Z...."<0...	
0030	02 00 fe 62 00 00 01 01	08 0a 2b 8d 2d ec 2b 8d	...b....H...f.../	
0040	2d ec 88 9a d2 0a 94 48	a7 66 17 13 f0 15 88 2f	...m]...3 @...v:uh	
0050	91 dc 6d 7c 0d ee 19 33	9a 40 ed f2 76 a1 75 68	v-?-...-X`s#	
0060	76 11 3f 9b 84 1f fe e3	b3 0d c2 58 60 bf 73 23	}6-Q... 2..	
0070	7d 26 89 be 51 e3 83 c8	c7 32 ce f0		

On peut également que les données capturées sont bien celles qui sont stockées dans la base de données, car si on regarde de plus près, on peut observer la suite de caractères "Hf/ml", que l'on retrouve également quand on essaye de lire le fichier depuis un IDE quelconque sur notre machine :

```

HfETBDC3NAK/m|
EM3@vuhvDC1?US
X`s#}&Q2

```

III - Analyse dynamique :

Pour conclure sur l'analyse de la sécurité de notre projet, nous allons réaliser une analyse dynamique de ce dernier. Lors de l'analyse statique, nous avons étudié notre programme quand ce dernier était à "l'arrêt", c'est-à-dire que nous étudions notamment son code source, etc... Ici, le but de l'analyse dynamique est d'étudier le comportement du programme quand ce dernier est "actif".

Avant de commencer, il est important de noter que pour pouvoir assurer une sécurité sur les entrées réalisées par les utilisateurs, ces dernières sont analysées grâce à une regex, et ainsi nous pouvons vérifier si ces dernières sont correctes. Cela nous permet donc d'établir une White List des différentes entrées.

```

int check_input(char *input){
    const char *regex_pattern = "^sectrans -list$|^sectrans -up [a-zA-Z0-9_./]*$|^sectrans -down [a-zA-Z0-9_./]*$";
}

```

La solution naïve pour réaliser cette analyse, est d'utiliser un outil de fuzzing comme celui vu en cours : *zzuf* car, grâce à cet outil, nous pouvons générer des entrées aléatoires et par conséquent tester notre application avec des entrées diverses et variées, et potentiellement trouver une entrée qui pourrait provoquer des dégâts sur notre application. Cependant, notre application est assez complexe à tester de manière dynamique, car nous devons d'abord nous connecter puis ensuite nous pouvons envoyer les différentes commandes. De plus, notre application possède une protection anti-DDOS avec un nombre de requêtes autorisées par utilisateur. Cela complique donc l'utilisation d'un fuzzing.

Pour réaliser notre analyse dynamique, nous allons donc utiliser un script bash dont le but sera de générer des entrées "aléatoires" permettant de suivre la regex, et ainsi de tester notre application. Voici le script bash utilisé pour réaliser ceci :

```
#!/bin/bash

chemin_client="./client"

function fuzzing {
    regex="^sectrans -list|^sectrans -up [a-zA-Z0-9_./]*|^sectrans -down [a-zA-Z0-9_./]*$"
    while true
    do
        random_data=$(head -c 50 /dev/urandom | tr -dc 'a-zA-Z0-9_./')
        for input in "sectrans -list" "sectrans -up" "sectrans -down" "sectrans"
        do
            fuzz_input="$input $random_data"
            #echo "$fuzz_input"
            if [[ $fuzz_input =~ $regex ]]; then
                # $chemin_client "$fuzz_input"
                echo -e "admin\nadmin\n$fuzz_input\nexit\n" | $chemin_client
            else
                echo "Entrée non conforme à la regex: $fuzz_input"
            fi
        done
        sleep 1
    done
}

fuzzing
```

Le but de ce script est de réaliser une boucle infinie dans laquelle :

- On génère une entrée aléatoire
- On réalise une connexion
- On envoie la commande avec l'entrée aléatoire
- On quitte l'application.

Une fois le script lancé, nous pouvons observer ces résultats côté Client :

```
[+] Bienvenue ! Veuillez rentrer votre commande
Veuillez rentrer votre identifiant
Veuillez rentrer votre mot de passe
Connexion réussie, vous pouvez rentrer vos commandes
Erreur lors de l'ouverture du fichier, votre fichier n'existe pas dans le bon chemin
L'entrée ne correspond pas au modèle attendu.
[+] Bienvenue ! Veuillez rentrer votre commande
Veuillez rentrer votre identifiant
Veuillez rentrer votre mot de passe
Connexion réussie, vous pouvez rentrer vos commandes
Reception dun fichier
Erreur : Le fichier demandé n'existe pas
L'entrée ne correspond pas au modèle attendu.
Entrée non conforme à la regex: sectrans PqzOZqJCzcb
```

Mais ce qui est surtout important, c'est de vérifier que l'on traite bien les commandes également côté serveur :

```
Message reçu :connection
Connexion réussie
Message reçu :sectrans -down 2jBzp4.nWsJQd
Erreur lors de l'ouverture du fichier pour l'envoi
Message reçu :connection
Connexion réussie
Message reçu :sectrans -up Pqz0ZqJCzcB
Reception dun fichier
Erreur lors de la reception du fichier
```

Après avoir laissé tourner ce script pendant plusieurs dizaines de minutes, on remarque qu'à chaque fois le serveur traite correctement les messages et qu'aucune erreur n'est relevée dans les différents logs. Il est important de noter que seule l'analyse dynamique ne garantit pas à elle seule la sécurité optimale de notre projet, c'est la combinaison de plusieurs analyses (dont celle dynamique) qui permet de nous positionner sur la sécurité de notre application.

IV - Conclusion :

Suite à ces 3 analyses, nous pouvons conclure que beaucoup de failles de sécurité ont pu être évitées notamment grâce à la partie de spécification réalisée au début de ce rapport. Cependant, il est important de noter que certains points sont très importants et nécessitent une correction dans le futur, comme la création d'une vraie base de données que ce soit pour les comptes, pour les fichiers, mais également l'introduction d'obfuscation de code dans le but de pouvoir rendre le code encore plus difficile à lire pour un attaquant réalisant du reverse engineering.

De plus, il peut être conseillé de faire appel à une entreprise Red Team pour réaliser un audit sur l'application SecTrans dans le but de découvrir d'autres failles non trouvées.

Dans l'optique de réaliser d'autres projets et d'en apprendre plus sur les bonnes pratiques de développement, il peut être conseillé de réaliser une formation pour les

développeurs en suivant les bonnes pratiques de la fondation OWASP :
<https://owasp.org/>

Utilisation du projet :

Cette partie à été rajoutée dans le but de servir de guide d'instructions pour la prise en main du projet, de la compilation à l'exécution.

Pour compiler ce projet, voici les étapes à suivre :

- 1: Veuillez vous rendre à la racine du projet, et exécutez la commande "make".
- 2: Dans un terminal, rendez vous dans le dossier Client/ et exécutez la commande "export LD_LIBRARY_PATH=D_LIBRARY_PATH:./"
- 3: Dans un deuxième terminal, répétez cette action mais cette fois dans le dossier Server/

Il est important de réaliser notamment la commande "make" car la librairie nécessaire à la vérification de la regex doit être installée sur la machine de l'utilisateur, car cette dernière peut être manquante.

Pour exécuter ce projet, voici les étapes à suivre :

- Assurez vous que les ports 8080 et 8081 de votre machine soient libres, ce sont les ports utilisés pour l'exécutable Client et l'exécutable Server
- Dans le terminal où vous avez réalisé l'étape 3), exécutez la commande "./server"
- Dans le terminal où vous avez réalisé l'étape 3), exécutez la commande "./client"
- Rentrez vos identifiants. Par défaut, les identifiants "admin" "admin" sont déjà présents dans l'application.
- Félicitations ! Vous pouvez utiliser les différentes commandes comme :
 - sectrans -up simon.txt
 - sectrans -list
 - sectrans -down simon.txt