Projet de programmation système L2 info / math-info

Sujet 2021-2022 : Serveur de web shells

Alexandre Bonlarron, Étienne Lozes ¶

Vidéo de démonstration du résultat final

Le but de ce projet est d'écrire un serveur offrant un service de web shell: vous allez permettre d'exécuter un shell depuis votre navigateur web. Ceci est généralement considéré comme une faille de sécurité. Ici vous placerez ce web shell sur un port "non réservé", c'est donc une vulnérabilité de sécurité difficile à détecter et qu'on tolèrera pour l'exercice (ne faites pas cela plus tard dans votre entreprise). Par précaution, et surtout parce que des erreurs peuvent se produire, et que c'est une bonne habitude, faites des sauvegardes de vos données importantes sur l'ordinateur que vous allez utiliser avant de commencer.

Le projet est découpé en plusieurs étapes qui doivent impérativement être traitées dans l'ordre. Seule petite exception, il suffit pour l'étape 1 que le serveur sache gérer une connection pour pouvoir avancer sur l'étape 2, il est donc possible d'attaquer l'étape 2 en finissant de polir l'étape 1 en parallèle.

Le projet est à rendre dans la boîte de dépôt Moodle avant le dimanche 8 mai 23:59. Chaque jour de retard entrainera deux points de pénalité. Le dossier déposé doit comporter les fichiers suivants:

- `AUTHORS` : la liste des auteurs (seul, en binôme, ou en trinôme) au format suivant: une ligne par auteur, au format `NOM;PRENOM;NUM_ETUDIANT` (par exemple `Etienne;Lozes;2176563`)
- `README` : un fichier texte ou markdown avec vos commentaires éventuels sur ce que vous avez fait
- `serveur.py` : le fichier de l'étape 1
- `traitant1.py`, `traitant2.py`, ... : les fichiers de l'étape 2
- `webshell.py` : le ou les fichier(s) de l'étape 3

Travailler en équipe

L'utilisation de `git` ou d'un autre gestionnaire de versions vous facilitera la tâche, mais si vous ne voulez pas apprendre `git` pour le moment (vous aurez l'occasion de le faire en projet de L3), vous pouvez bien sûr échanger vos fichiers entre vous par mail. Je rappelle aussi la règle suivante: vous ne devez pas échanger de fichiers avec d'autres personnes que votre binôme. Les discussions sur moodle ou discord sont encouragées, c'est très bien si vous posez des questions, et c'est très bien si vous aidez quelqu'un qui a posé une question à la place de l'enseignant (si ce que vous dites n'est pas correct, on corrigera). Mais restez au niveau de la langue française (ou d'un dessin), ne montrez pas de code (occasionnellement, 1 ou 2 lignes de codes peuvent être divulguées si votre question ou votre réponse n'a pas été comprise, mais évitez autant que possible). Enfin, si vous utilisez un dépot dans le cloud (bitbucket, github, etc) paramétrez votre dépot pour qu'il ne soit pas public et que seul votre binôme y ait accès. J'utilise un logiciel de détection de plagiat et j'enlèverai des points aux plagieurs mais aussi aux plagiés (sans chercher à faire de distinction) en cas de plagiat détecté, même partiel.

Librairies autorisées

Il existe des librairies Python qui font des choses proches de celles attendues à certaines étapes du projet. Le but du projet étant de s'approprier les appels systèmes bas niveau, il est interdit de les utiliser.

Plus précisément, les seuls import de modules autorisés sont os, sys, signal, socket, select, atexit, random, time, et json. Vous n'avez pas nécessairement besoin de toutes ces librairies.

L'utilisation d'autres librairies que celles-ci sera pénalisée.

Quantité de travail estimée

L'estimation est toujours forcément subjective et variable d'une personne à l'autre, mais le projet est pensé pour être faisable en 12h, en posé suivi les TDs et les TPs tout au long du semestre. Les 12h incluent les 6h de TP dédiées au projet. Il est recommandé de préparer ces séances dédiées au projet pour poser un maximum de questions.

À titre indicatif, voici la taille des fichiers (en lignes, commentaires et lignes vides comprises) d'une solution:

```
 53 serveur.py
  7 traitant1.py
 26 traitant2.py
 56 traitant3.py
 74 traitant4.py
 78 traitant5.py
164 webshell1.py
177 webshell2.py
213 webshell3.py
```

Barème prévisionnel

Ce barème est donné à titre indicatif et pourra évoluer s'il s'avère trop sévère.

- Étape 1 : 8 points
  - 3 points : serveur capable de lancer un processus traitant pour traiter une connection
  - 2 points : plusieurs connections simultanées possibles
  - 2 points : gestion du signal SIGINT et terminaison propre
  - 1 point : limitation du nombre de connections
- Étape 2 : 6 points
  - étape 2.1 : 1 point
  - étape 2.2 : 1 point
  - étape 2.3 : 2 points
  - étape 2.4 : 1 point
  - étape 2.5 : 1 point
- Étape 3 : 6 points
  - étape 3.1 : 2 points
  - étape 3.2 : 2 points
  - étape 3.3 : 2 points

Étape 1 : Serveur générique

> usage: `./serveur.py traitant port`

avec

- `traitant` qui est le nom d'un fichier exécutable
- `port` est un entier > 2000 (un port non réservé). Dans les exemples ci-dessous, j'utiliserai 3765, mais vous pouvez mettre ce que vous voulez.

Le serveur écoute sur le port spécifié en argument et accepte jusqu'à 4 connections simultanées. Au-delà, le serveur attend qu'une connection ait terminé avant d'en lancer une nouvelle.

Le serveur implémente son propre pare-feu: si la demande de connection a été effectuée d'une machine autre que localhost, la connection est fermée immédiatement (considérez les informations renvoyées par la méthode `accept`, cf doc ici)

Si la connection acceptée a bien été initiée par un client sur localhost, le serveur réalise les étapes suivantes:

1. un processus fils est créé pour gèrer cette connection (`os.fork`). Le père se met en attente de la prochaine connection (`accept`) ou de la fin d'un fils (`os.wait`) si le nombre connections maximal a été atteint.
2. Le fils redirige l'entrée et la sortie standard de ce processus (`os.dup2`, ou `os.close` / `os.dup`) sur la socket liée à cette connection (méthode fileno).
3. Le fils charge ensuite l'exécutable spécifié en argument (`os.execvp`)

Si le serveur reçoit le signal SIGINT (Ctrl+C), il s'arrête proprement en envoyant un signal SIGINT à tous les processus fils encore vivants et en attendant la terminaison de chacun.

▶ **Indications (cliquer pour lire)**

▶ **Test (cliquer pour lire)**

Étape 2 : Le traitant de requête

Le but est d'écrire un petit utilitaire `./traitant.py` à utiliser avec le serveur de la question précédente (vous lancerez donc `./serveur.py ./traitant.py PORT` pour tester).

Cet utilitaire lit une requête http GET sur l'entrée standard et y répond sur sa sortie standard, puis ferme la sortie standard et termine.

On détaille les principales sous-étapes pour y arriver ci-dessous. À la fin de chaque sous-étape, conservez le travail que vous avez fait dans un fichier nommé `traitant`x`.py` (en remplaçant x par 1, 2, 3, etc). Ces fichiers intermédiaires seront à rendre.

Étape 2.1 : la première requête

Commencez par écrire un traitant simple qui lit jusqu'à 100 Ko sur l'entrée standard, affiche sur sa **sortie d'erreur** ce qu'il a lu, et termine.

Une fois que c'est fait, lancez `./serveur.py ./traitant.py 3765`, puis dans votre navigateur web tapez `http://localhost:3765`. Vous devriez voir dans le terminal du serveur s'afficher la requête du client (votre navigateur), avec quelque chose comme:

```
GET / HTTP/1.1
Host: localhost:3765
Upgrade-Insecure-Requests: 1
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/605.1.15 (KHTML, like Gecko) Version/15.
4 Safari/605.1.15
Accept-Language: fr-FR,fr;q=0.9
Accept-Encoding: gzip, deflate
Connection: keep-alive
```

Il s'agit d'une requête GET dans le protocole http version 1.1.

Pour terminer l'étape 1, ajoutez le comportement suivant à votre traitant: si la requête n'est pas un GET et n'est pas sur le protocole http 1.1, le traitant affiche `request not supported` sur la sortie d'erreur et termine immédiatement.

Étape 2.2 : la première réponse

Vous allez maintenant former un paquet http contenant la réponse à la requête et l'envoyer (i.e. l'écrire sur la sortie standard). Voici un exemple de paquet réponse:

```
HTTP/1.1 200
Content-Type: text/html; charset=utf-8
Connection: close
Content-Length: 125

<!DOCTYPE html>
<head>
    <title>Hello, world!</title>
</head>
<body>
Bonjour le monde! <br>
Coma va, Nizza?
</body>
</html>
```

Quelques explications sur le paquet http (vous référer à votre cours de réseau, ou lire par exemple ceci, ceci, ou ceci pour en savoir plus):

- la ligne vide sépare la partie en-tête (*header*) de la partie utile (*payload*) du paquet.
- l'entête spécifie le code de réponse (ici 200, on a pu répondre à la requête), le format de la partie utile (du html utilisant le jeu de caractères utf 8), la gestion de la connection (elle est fermée immédiatement après l'envoi de la réponse), et la longueur de la partie utile, en octets (ici il y a 125 octets entre <!DOCTYPE et html>)
- la partie utile est du code html. Vous vous référerez à votre cours de Techno web pour en savoir plus.

Commencez par modifier le traitant pour qu'il envoie le paquet donné en exemple ci-dessus. Vérifiez que le navigateur web affiche bien *Bonjour le monde!* et *Coma va, Nizza?* sur deux lignes.

Pour terminer cette étape, modifiez le paquet réponse pour qu'il affiche la requête initiale dans le navigateur web (à la place de *Bonjour etc*).

Voir aussi : vidéo de démo de fin de l'étape 2.2

Etape 2.3: la deuxième requête et sa réponse

Le but est de réaliser le comportement suivant: la page web contient une zone de saisie et un bouton pour valider. Quand on presse le bouton, la page est **rechargée** (pour les petits malins, javascript est interdit, on veut tout faire côté serveur) et ce qui a été saisi s'affiche au-dessus de la zone de saisie.

Voir aussi: vidéo de démo

Indications:

- vous pourrez utiliser le code suivant entre les balises <body> de la page générée pour définir un formulaire

```
<form action="ajoute" method="get">
        <input type="text" name="saisie" value="Tapez quelque chose" />
        <input type="submit" name="send" value="&#9166;">
    </form>
```

- Faites afficher la requête reçue par le traitant. Lorsque l'utilisateur clique sur le bouton, où se trouve ce qu'il a saisi?
- Il vous faudra décoder la saisie. Les espaces sont remplacés par des `+`, et d'autres caractères sont représentés par leur code utf8 en hexadécimal avec le caractère d'échappement `%` devant chaque octet. Par exemple, `a "` est codé en `a+%22`. Vous pourrez utiliser la fonction ci-dessous pour décoder la saisie trouvée dans le header et reconstituer ce que l'utilisateur a tapé dans le navigateur.

```python
def escaped_utf8_to_utf8(s):
    res = b'' ; i = 0
    while i < len(s):
        if s[i] == '%':
            res += int(s[i+1:i+3], base=16).to_bytes(1, byteorder='big')
            i += 3
        else :
            res += bytes(s[i])
            i += 1
    return res.decode('utf-8')
```

  Par exemple,

```python
saisie_codee = 'a+%22'
saisie = escaped_utf8_to_utf8(saisie_codee.replace('+', ' '))
print(saisie)
```

  affiche `a "`.
- vous ne devez pas recoder la saisie en l'échappant pour l'envoyer. Dans le paquet de réponse vous avez déclaré que vous utilisez le jeu de caractère UTF-8, donc utilisez-le.

Étape 2.4 : historique des saisies

Le but est de réaliser le comportement suivant: la page web contient toujours une zone de saisie et un bouton pour valider en bas de page. En haut de page s'affichent les saisies déjà réalisées: chaque nouvelle saisie s'ajoute aux précédentes.

vidéo de démo de l'étape 2.4

Cette étape est plus difficile que les autres, parce qu'on a besoin de conserver l'historique entre deux requêtes (vérifiez que ce n'est pas le cas avec ce que vous avez fait à l'étape 3). Le problème, c'est que chaque requête est traitée par un processus traitant différent: on ne peut donc pas sauver l'historique dans une variable du processus, cet historique doit persister après la fin du processus traitant. Nous nous proposons de résoudre ce problème de la façon suivante: une copie de l'historique va être maintenue par les divers processus traitant dans un fichier `/tmp/historique.txt` : chaque traitant va donc possiblement écrire et lire ce fichier. On rappelle l'existence du flag `O_APPEND` pour `os.write`.

Étape 2.5 : notion de session

On veut maintenant pouvoir taper `http://localhost:3765` dans plusieurs onglets du navigateur, ou dans plusieurs navigateurs, et développer des historiques de saisies qui ne se mélangent pas.

vidéo de démo de l'étape 2.5

Indication

Au lieu d'un seul fichier `/tmp/historique.txt`, vous pourrez utiliser plusieurs fichiers `/tmp/historique_sessionXXX.txt` où XXX est un identifiant de session. Chaque requête initiale (celle envoyée par le navigateur quand vous tapez `localhost:PORT` dans la barre de navigation) génère un nouvel identifiant de session (par exemple le pid du traitant appelé pour la requête initiale).

Le formulaire généré par le traitant en réponse à la requête initiale reprend cet identifiant: le champ `action` devient par exemple `ajoute_dans_session_XXX`. Les requêtes envoyées par le client lorsqu'on presse le bouton contiendront donc l'identifiant de session et permettront à un traitant de savoir dans quelle session s'inscrit la requête qu'il a à traiter.

Étape 3 : Web shell

Le but est de réaliser le programme `./webshell.py` suivant.

> usage: `./serveur.py ./webshell.py PORT`

Lance un serveur de web shells. On peut ensuite ouvrir un shell dans un navigateur web en demandant la page `localhost:PORT`.

De même que pour l'étape 2, chaque sous-étape peut donner lieu à un rendu de fichier différent (`webshell1.py`, `webshell2.py`, etc).

Étape 3.1 : exécution de commandes dans des shells distincts

Pour cette première étape, le but est de pouvoir saisir et exécuter n'importe quelle commande, mais avec quelques limitations.

1. Tout d'abord, chaque commande sera traitée par un shell différent à chaque fois. Par exemple la commande `cd` sera sans effet pour les commandes suivantes.
2. Ensuite, on va supposer que les commandes ne lisent rien sur l'entrée standard. Si l'utilisateur lance une commande qui fait une lecture sur l'entrée standard, la page web va se figer.

Ces deux limitations seront levées aux étapes suivantes. Il restera encore au moins une limitation importante par rapport à un shell qui s'exécute dans un terminal: notre web shell ne sera pas en mesure de supporter les processus qui exploitent les fonctionalités de terminal, comme `less` ou `man`... Ce serait intéressant, mais cela nous amènerait beaucoup plus loin... et il y a déjà pas mal de chemin à parcourir.

Lorsque le shell attend une nouvelle commande, une invite (*prompt*) est affichée avec l'heure courante. Lorsque la commande a fini de s'exécuter, la page web est actualisée et affiche le résultat de la commande ainsi qu'un nouveau prompt.

Indication: usage d'un shell et son option `-c`

Pour traiter une commande complexe, vous pouvez demander à un shell de l'exécuter. La plupart des shells (`sh`, `bash`, `csh`, `zsh`, ...) ont une option `-c` qui permet de spécifier une commande à faire exécuter par le shell. Par exemple, le programme python ci-dessous

```python
import os
commande = 'ls -l | grep -e "hello word" || echo "repertoire vide"'
os.execvp('sh', ['sh','-c', commande])
```

fait un recouvrement avec un shell qui exécute la commande `ls -l | grep -e "hello word" || echo "repertoire vide"`.

Il vous faudra faire ce recouvrement dans un processus fils (pas directement dans le processus du traitant de requête) pour pouvoir faire des actions "après" le recouvrement. Il faudra en plus créer un tube anonyme (`os.pipe`) entre le père et le fils: le fils (qui exécutera la commande) aura ses sorties standards et d'erreur redirigées vers le tube. Le père (le traitant de requête) n'aura qu'à lire dans le tube pour récupérer la sortie de la commande, et l'utiliser pour former le paquet de réponse à envoyer au navigateur.

Indication: commandes qui n'affichent rien

À un moment vous aurez quelque chose qui gère bien des commandes comme `ls` qui affichent quelque chose, mais qui bloquent pour des commandes qui n'affichent rien comme `touch toto.txt`. Le problème, c'est que sans doute votre traitant est bloqué en lecture sur le tube où le shell est censé renvoyer la sortie de la commande. Il y a plusieurs approches possibles pour résoudre le problème:

- passer en mode non bloquant (cf os.setblocking et exercice du TD 6 où on calcule la capacité d'un tube), faire un appel à `time.sleep` pour laisser un peu de temps au fils pour remplir le tube, puis faire un read non bloquant.
- insérer un marqueur de fin dans la sortie de la commande:
  1. remplacer la commande `cmd` à exécuter par `cmd ; echo MARKER` où MARKER est une chaîne de caractères que la commande a peu de chance de produire elle-même (par exemple quelque chose de calculé de manière aléatoire à partir de l'identifiant de session).
  2. vider le tube caractère par caractère jusqu'à avoir lu MARKER
  3. effacer le marqueur de fin de ce qui a été lu pour reconstituer la sortie de la commande `cmd` initiale

Test fin d'étape 3.1

Saisissez `rm -f /tmp/toto.txt`, vérifiez que vous récupérez bien la main, saisissez `touch /tmp/toto.txt`, vérifiez que vous récupérez bien la main, et enfin saisissez `ls -l /tmp/ |grep toto`, et vérifiez que `/tmp/toto.txt` s'affiche bien.

Extras optionnels

Ce n'est pas à proprement parler nécessaire, mais vous pouvez remplacer le fichier historique au format texte de l'étape 2 par un fichier au format json qui contiendra une structure de donnée qui modélise "l'état" de la session. Personnellement je m'en suis servi pour distinguer facilement les parties de l'historique qui étaient des saisies utilisateurs de celles qui étaient des résultats de commande. Les plus curieux regarderont ce qu'on peut faire avec le module json.

Autre idée bonus hors barème (ca n'apporte pas de fonctionalité particulière, et ce n'est pas compliqué, mais c'est toujours bien de savoir faire): au lieu de fixer en dur quel shell vous allez utiliser (par exemple `sh` dans l'exemple ci-dessus), vous pouvez consulter la variable d'environnement `SHELL` pour voir quel est le shell qui a été utilisé pour lancer le serveur, et utiliser le même shell pour exécuter vos commandes.

Étape 3.2 Plusieurs commandes au sein d'un même shell

Pour le moment, si vous saisissez `cd /tmp` puis `pwd`, vous ne verrez pas `/tmp` s'afficher (sauf si vous avez lancé le serveur depuis `/tmp`...). La seule chose qui marche, c'est de grouper les deux en une seule commande (`cd /tmp ; pwd`)... mais c'est limité.

L'objectif de cette étape est que chaque commande "fasse effet" sur les suivantes. Lorsqu'on saisira `cd /tmp` puis `pwd`, on verra bien s'afficher `/tmp` comme attendu dans un vrai shell.

Pour cela, vous allez utiliser le même processus shell pour interpréter les différentes commandes. Ce processus shell va donc devoir "persister" entre deux requêtes d'une même session. En revanche, chaque requête va créer un nouveau traitant qui va devoir :

1. établir une connection avec ce processus shell persistant
2. lui indiquer quelle est la prochaine commande à exécuter
3. récupérer le résultat, construire le paquet réponse http, l'envoyer au navigateur web,
4. et mourir en fermant la connection avec shell, sans mettre fin au shell.

Il faudra donc lancer ce shell persistant à la première requête d'une session (celle qui correspond au moment où l'utilisateur tape `localhost:PORT` dans la barre du navigateur) et mettre en place des tubes nommés identifiables à partir de l'identifiant de session. Vous aurez deux tubes nommés à utiliser (ainsi qu'un troisième à la prochaine étape). Le premier tube permettra au traitant d'envoyer la commande au shell, et le deuxième permettra au shell d'envoyer le résultat au traitant.

Pour comprendre l'idée, voici une petite manipulation à faire:

1. ouvrez un terminal et tapez

   ```
   cd /tmp ; mkfifo shell_vers_traitant traitant_vers_shell
   ```

   puis tapez

   ```
   sh traitant_vers_shell &> shell_vers_traitant
   ```

   Notez dans cette dernière commande que `traitant_vers_shell` n'est pas utilisé pour faire une redirection, mais comme argument de sh. Quand un shell a un fichier en argument, il le traite comme un "fichier de script" et lit les commandes à exécuter dedans. Ici c'est un fichier de script "en cours d'écriture" que les traitants vont écrire au fur et à mesure, et que le shell va exécuter comme n'importe quel fichier de script.
2. ouvrez deux autres terminaux, et tapez dans chacun respectivement

   ```
   cat > /tmp/traitant_vers_shell
   ```

   et

   ```
   cat < /tmp/shell_vers_traitant
   ```

   Dans le premier des deux terminaux, vous pouvez désormais saisir des commandes dont la sortie s'affichera dans le second terminal.

Il reste un petit problème à résoudre: lorsque vous tuez le processus `cat > /tmp/traitant_vers_shell`, vous devriez observer que cela met fin aussi au shell et au `cat < /tmp/shell_vers_traitant`.

C'est le problème qu'il reste à gérer, puisque le traitant va mourrir et qu'un autre traitant réouvrira le tube plus tard. Il y a une solution, mais il faut bien comprendre comment marche un tube.

Rappelons que

- l'ouverture d'un tube nommé en lecture est bloquante: tant qu'il n'y a pas au moins une ouverture en écriture du même tube, on attend.
- c'est une synchronisation par rendez-vous (on parle aussi de barrière de synchronisation): l'ouverture en écriture est aussi bloquante en attente d'une ouverture en lecture
- une fois que la synchronisation a eu lieu, un canal est "alloué". Il reste alloué tant qu'il y a au moins un lecteur et au moins un écrivain. De nouveaux lecteurs et écrivains peuvent s'ajouter en ouvrant le tube nommé, d'autre peuvent partir en fermant leur descripteur de fichier, mais c'est toujours le même canal qui sera utilisé.
- Cependant, si à un moment il n'y a plus aucun lecteur (ou plus aucun écrivain), le canal est désalloué (le tube nommé, lui, reste visible dans le système de fichier, mais "ce n'est qu'un nom"). Il faudra une nouvelle synchronisation avec au moins deux open sur le tube nommé pour réallouer un nouveau canal.

Ceci étant dit, vous devriez être en mesure de comprendre l'astuce ci-dessous pour que les canaux associés aux tubes nommés ne soient pas fermés à la mort du traitant. Reprennez la petite manip avec les trois terminaux décrite ci-dessus, mais cette fois-ci tapez

```
sh traitant_vers_shell 3<> traitant_vers_shell &> shell_vers_traitant 4< shell_vers_traitant
```

Soyez sûrs de comprendre ce que veut dire cette commande! Au besoin, relisez comment fonctionnent les redirections.

Attention, l'ordre dans lequel les redirections sont faites est important pour éviter les interblocages...

Vérifiez que maintenant si vous tuez la commande cat d'un des deux autres terminaux et que vous la relancez, vous pouvez reprendre le shell comme si de rien n'était.

Mieux: vous pouvez maintenant faire la manip avec uniquement deux terminaux: celui du shell, et l'autre dans lequel vous faites un cat pour envoyer la commande au shell, vous tuez le processus, vous faites un cat pour lire le résultat, vous tuez le processus, vous faites un cat pour envoyer une deuxième commande au shell, etc. Faites ce test avec uniquement deux terminaux pour vérifier que tout marche bien et qu'il n'y a pas d'interblocage, et identifiez quels sont les risques d'interblocages.

Ensuite il ne vous reste plus qu'à mettre cette idée en pratique dans votre code...

Étape 3.3 Gestion de l'entrée standard

Le but de cette étape est de permettre d'exécuter des commandes qui lisent l'entrée standard. Typiquement, on veut pouvoir exécuter la commande read sans que ça bloque, et vérifier que la saisie a bien été prise en compte avec `echo $REPLY`.

```
PROMT $ read⏎
hello l'entree standard⏎
PROMT $ echo $REPLY⏎
hello l'entree standard
PROMT $
```

Lorsque le processus lancé par une commande semble bloqué en attente d'une saisie clavier (il n'a rien produit en sortie durant la dernière seconde), le traitant envoie une page au navigateur qui affiche ce que le processus a généré sur la sortie standard ou la sortie d'erreur jusque là, ainsi que la boite de saisie habituelle, mais sans le prompt. À la requête suivante, le prochain traitant enverra la saisie non pas dans le tube `traitant_vers_shell`, mais dans un autre tube, relié à l'entrée standard du shell.

Il vous faudra gérer ce nouveau tube, adapter l'astuce précédente pour qu'il soit persistant lui aussi, mettre en place un méchanisme pour sortir de l'attente bloquante sur le tube `shell_vers_traitant` au bout d'une seconde (par exemple avec `select.select`, ou en utilisant `signal.alarm`, entre autre, vous avez le choix...), et mettre en place quelque chose pour que le traitant sache dans quel contexte l'utilisateur était au moment de la saisie (saisie d'une ligne de commande ou saisie sur l'entrée standard) et dans quel tube envoyer la saisie qu'il a reçue.

À vous de jouer!