TP1 : Programme et mémoire

Gilles Menez - DS4H - UCA

September 13, 2024

Contents

1 Variables d'environnement

Les variables d'environnement constituent **un moyen d'influencer le comportement des logiciels** dans un système.

- La plus connue est sans aucun doute la variable "PATH" (utilisée par exemple par les Shells) qui détermine une liste de répertoires parcourus pour "trouver" les fichiers exécutables associés aux commandes.

Vous pouvez visualiser les variables de votre processus Shell courant avec la commande `env` :

```
menez@duke:~/EnseignementsCurrent/Cours_Sys_Prog/Slides$ env
LANG=fr_FR.UTF-8
GDM_LANG=fr_FR.UTF-8
DISPLAY=:1.0
GTK_OVERLAY_SCROLLING=0
COLORTERM=truecolor
XDG_VTNR=2
SSH_AUTH_SOCK=/run/user/1000/keyring/ssh
XDG_SESSION_ID=3
USER=menez
DESKTOP_SESSION=default
PWD=/home/menez/EnseignementsCurrent/Cours_Sys_Prog/Slides
HOME=/home/menez
JOURNAL_STREAM=8:27034
SSH_AGENT_PID=1607
QT_ACCESSIBILITY=1
XDG_SESSION_TYPE=x11
XDG_DATA_DIRS=/usr/share/mate:/usr/local/share/:/usr/share/
MATE_DESKTOP_SESSION_ID=this-is-deprecated
XDG_SESSION_DESKTOP=default
GTK_MODULES=gail:atk-bridge
WINDOWPATH=2
TERM=xterm
SHELL=/bin/bash
VTE_VERSION=4601
XMODIFIERS=emacs
XDG_CURRENT_DESKTOP=MATE
GPG_AGENT_INFO=/run/user/1000/gnupg/S.gpg-agent:0:1
QT_LINUX_ACCESSIBILITY_ALWAYS_ON=1
SHLVL=1
XDG_SEAT=seat0
PYTHONPATH=/home/menez/Dev/GaussianProcess/src
PRINTER=ricoh
WINDOWID=51893435
GDMSESSION=default
LOGNAME=menez
DBUS_SESSION_BUS_ADDRESS=unix:path=/run/user/1000/bus
XDG_RUNTIME_DIR=/run/user/1000
XAUTHORITY=/run/user/1000/gdm/Xauthority
PATH=/home/menez/bin:/usr/local/bin:/usr/bin:/bin:/usr/local/games:/usr/games:
SESSION_MANAGER=local/duke:@/tmp/.ICE-unix/1550,unix/duke:/tmp/.ICE-unix/1550
_=/usr/bin/env
OLDPWD=/home/menez/EnseignementsCurrent/Cours_Sys_Prog/Src
```

L'environnement est transmis à un processus comme **une liste de "chaînes de caractères"** de la forme :

`"IdentDeLaVariable=ValeurDeLaVariable"`

(Je garde les guillemets pour bien montrer qu'il s'agit d'une chaîne)
Un programme qui devient processus hérite la plupart du temps des variables du processus qui l'a lancé.

- Ainsi lorsque vous lancez un programme depuis un Shell, ses (celles du Shell) variables sont communiquées au nouveau processus.

Une des façons de récupérer cet environnement au niveau de votre programme est d'utiliser la variable externe "environ".

- Il y en a d'autres . . . mais celle là va permettre de voir comment vous maîtrisez vos tableaux de chaînes en C ;-)

Dans le programme incomplet qui suit, je donne un exemple de l'utilisation de cette variable :

```c
#include <stdlib.h>
#include <stdio.h>
int main (int argc, char *argv[]){
  extern char** environ;
  int i;
  printf ("Environnement:\n");
  /*
   A remplir !!!!!!!!!!
   */
  exit (0);
}
```

1.1 TODO :

- Est ce que vous pouvez achever ce code pour faire afficher trois colonnes donnant pour chaque variable d'environnement :
  1. Son index
  2. Son adresse
  3. Sa valeur

Un peu comme cela :

```
Environnement:
0 (0bffff83c): USER=menez
1 (0bffff840): LOGNAME=menez
2 (0bffff844): HOME=/users/menez
3 (0bffff848): PATH=/bin:/usr/bin:/usr/local ...
4 (0bffff84c): MAIL=/var/mail/menez
5 (0bffff850): SHELL=/bin/zsh
6 (0bffff854): SSH_CLIENT=::ffff:134.59.1
7 (0bffff858): SSH_CONNECTION=::ffff:134.
8 (0bffff85c): SSH_TTY=/dev/pts/0
9 (0bffff860): TERM=cygwin
...
11 (0bffff868): PWD=/users/menez
```

```
...
38 (0bffff8d4): JAVAC=/opt/blackdown-jdk-1.4.1/bin/javac
39 (0bffff8d8): DISPLAY=vtr.i3s.unice.fr:0
40 (0bffff8dc): _=/users/menez/./a.out
```

2 Segments et variables

On commence avec le programme C le plus vide qui soit :

```c
#include <stdio.h>

int main(void) {
    return 0;
}
```

La commande Shell "size" montre la taille des segments du programme compilé :

```
> size -G a.out
text       data        bss      total filename
320        1452          8       1780 a.out
```

On sait ainsi que l'exécutable a.out occupe

- 327 octets de code,
- 1452 octets de données globales/statiques initialisées
- 8 octets de données globales/statiques non initialisées

2.1 QUIZZ :

- Pourquoi la taille de la pile et celle du tas ne font pas partie des sorties de la commande size ?
- Ce programme "est vide" ... et pourtant il occupe(ra) de la mémoire. Pourquoi ?
- Essayez de compiler avec l'option static et regarder la size :

  ```
  > gcc -static vide0.c
  > size -G a.out
  text       data        bss      total filename
  513001     177965      22440     713406 a.out
  ```

  Qu'est ce qui se passe ?

2.2 TODO :

Vous interprétez les tailles de segments dans les cas suivants :

- Quels segments voient leurs tailles modifiées ?
- Pourquoi ces évolutions ?

1.
```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    return 0;
}
```

2.
```c
#include <stdio.h>

int global;

int main(void) {
    return 0;
}
```

3.
```c
#include <stdio.h>

int global;

int main(void) {
    static int var;
    return 0;
}
```

4.
```c
#include <stdio.h>

int global;

int main(void) {
    static int var = 10;
    return 0;
}
```

5.
```c
#include <stdio.h>

int global = 200;

int main(void) {
    static int var = 10;
    return 0;
}
```

3 Segments et fonctions

3.1 TODO :

- Pour illustrer le mécanisme de passage de paramètre en pile, vous réalisez une fonction récursive de calcul de la représentation binaire d'un nombre entier et vous faîtes exécuter cela sous

  `https://pythontutor.com`

  N'hésitez pas à faire appel à cet outil pour comprendre ce comment la mémoire est gérée.

4 Buffer overflow

En reprenant les notes de cours et l'exemple :

4.1 TODO :

1. Expérimenter le buffer overflow
2. Pourquoi quand on utilise le caractère "x" pour déborder, la variable "access_granted" prend cette valeur (celle dans le terminal de la figure du cours)?
3. Résoudre le problème de débordement de tampon vu en cours.

5 Shellcode

Il manque encore quelques notions pour développer "son" Shellcode et notamment la notion d'appel système mais cela va venir ...

5.1 TODO :

Plus tard vous ferez mais pour l'instant,

1. Reproduire l'exécution de l'exemple du cours.

   N'oubliez de compiler avec la commande :

   ```
   gcc -Wall -z execstack hello.c -o hello
   ```

   Il fait quoi ce Shellcode ?
2. Selon vous quel est l'appel système qui est à la base de ce test ?
3. D'ailleurs à quoi sert l'option de compilation "-z execstack" ?

   - `https://en.wikipedia.org/wiki/Executable-space_protection`
   - `https://linux.die.net/man/8/execstack`
4. Et si on faisait du code une variable globale (au lieu d'une variable dynamique) ?

   Pourquoi ce résultat ?