Compte rendu Sécurité Nanvix

Erwan Poncin / Maxime Bossant / Maxence Maury

Lien du git:

https://github.com/erwanponcin/AS.git

Création de deux utilisateurs user1 et user2

Pour ajouter plusieurs users, on change le paramètre `MULTIUSERS` dans le fichier `config.h` puis on ajoute ces deux lignes dans le fichier `nanvix/tools/build/build-img.sh:`

```
bin/useradd $file user1 mdp 1 1
bin/useradd $file user2 md2 2 2
```

Crash système par une saturation des structures internes du noyau

Dans `AS/nanvix/src/sbin/test.c`, on rajoute un test qui ajoute une forkbomb, dans lequel on crée une boucle qui initialise des processus à l'infini, ce qui monopolise le processeur et fait planter Nanvix

```
int forkbomb_test(){
	while (1)
		fork();
	return 0;
}
```

Le but est donc de limiter le nombre de processus qu'un utilisateur qui n'est pas root peut créer. Pour cela, on ajoute:

- Dans `nanvix/include/nanvix/pm.h`:

```
#define MAX_PROC_PER_USER 10
#define MAX_USER 5
EXTERN int nb_proc_user[MAX_USER];
```

On définit 3 macros:

`MAX_PROC_PER_USER` défini le nombre max de process qu'un user peut créer.

`MAX_USER` défini le nombre max de user qu'il peut y avoir.

`nb_proc_user` définit le tableau du nombre de processus pour chaque user.

- Dans `nanvix/src/kernel/pm/pm.c`:

```
int nb_proc_user[MAX_USER];
```

On définit le tableau de processus par user.

- Dans `nanvix/src/kernel/pm/die.c` (fonction bury()):

```
nb_proc_user[proc→euid]--;
```

On décrémente le nombre de processus du user à la mort du processus.

- Dans `nanvix/src/kernel/sys/fork.c` :

```
nb_proc_user[curr_proc→euid]++;
   if ((!IS_SUPERUSER(curr_proc)) && (nb_proc_user[curr_proc→euid] > MAX_PROC_PER_USER))
         return (-EAGAIN);
```

On incrémente le nombre de processus du user actuel et on vérifie qu'il n'a pas atteint son max (s'il est pas root).

Après test: on obtient bien un blocage à 10 processus (`MAX_PROC_PER_USER`):

```
1 process created
2 process for user 0

10 process for user 1

0 process for user 2

0 process for user 3

0 process for user 4
```

Print Users

On affiche la liste des utilisateurs au lancement de nanvix sur la page de login (comme sur un OS classique où on a la liste des utilisateurs auxquels on peut se connecter). Le code dans `nanvix/src/ubin/login/login.c` suit cette logique:

- On ouvre le fichier `etc/password` : `(file = open("/etc/passwords", O_RDONLY))`
- On parcourt le fichier avec la taille du type `account`
  - On décrypte le nom: `account_decrypt(a.name, USERNAME_MAX, KERNEL_HASH);`
  - On l'affiche: `printf("username: %s uid: %d gid: %d\\n", a.name , a.uid, a.gid);`

Crack password

On cherche ici à écrire un programme qui a comme paramètre un nom d'utilisateur et va essayer de trouver le mot de passe de cet user par brut force

```
char *crack(char *username)
{
   char *password = malloc(PASSWORD_MAX);
   char *alphabet = "abcdefghijklmnopqrstuvwxyz";

   for (int i = 0; i < 26; i++)
   {
```

```
        password[0] = alphabet[i];

        for (int j = 0; j < 26; j++)
        {
            password[1] = alphabet[j];

            for (int k = 0; k < 26; k++)
            {
                password[2] = alphabet[k];
                if (authenticate(username, password))
                    return password;
            }
        }
    }
    return NULL;
}
```

Pour tester les différents password, on essaye chaque combinaison de lettres de l'alphabet pour chaque place d'un password de taille 3. Possibilité d'augmenter la taille du password testé en rajoutant une boucle mais le temps est grandement augmenté.

Si on arrive à s'authentifier avec le password parcouru, on retourne le password trouvé.

Sinon, on renvoi `NULL` ce qui signifie que le password n'a pas pu être trouvé.

Résultat :

Pour `user1` ayant le mot de passe `mdp` on obtient le résultat suivant (~ 6 secondes):

```
% crack user1
Password found: mdp
```

Pour `user2` ayant le mot de passe `md2` on obtient le résultat suivant (~15 secondes):

```
% crack user2
Password not found
```

Su command

Le but est de créer une nouvelle commande: `su` qui permet de changer le `euid` du processus courant (mais pas le `uid`). Pour cela, on crée le fichier `nanvix/src/ubin/su/su.c` tel que:

A l'appel de la fonction `su <user>` , le mot de passe est demandé et s'il est bon, le changement se fait avec la fonction `seteuid()` :

```
PUBLIC int sys_seteuid(uid_t uid){
    curr_proc->euid = uid;
    return (0);
}
```

- Fonctionnement de `su.c` :
    - On récupère le user en argument de la fonction

- On demande à l'utilisateur de donner le mot de passe
- On utilise la fonction `authenticate()` pour vérifier que le mot de passe est bon
- Si oui, on change le `euid`

```
int main (int argc, char ** argv) {
   if (argc < 2) {
         printf("USAGE : su [user]\n");
         return 0;
   }
   char *name = argv[1];
   char password[PASSWORD_MAX];
   fgets(password, PASSWORD_MAX, stdin);
   if (authenticate(name, password)) {
         seteuid(uid);
         printf("Success\nYour new euid is %d\n", geteuid());
         return 0;
   }
   printf("Authentication failed\n");
   return 1;
}
```

- Trace d'exécution:

```
% su user2
md2
Success
Your new euid is 2
```

Création d'une nouvelle commande passwd

Le but est de créer une nouvelle commande: `passwd` qui permet de changer le mot de passe du processus courant. Pour cela, on crée le fichier `nanvix/src/ubin/passwd/passwd.c` tel que:

A l'appel de la fonction `passwd <user>` , le mot de passe est demandé et s'il est bon, le nouveau mot de passe est demandé puis une confirmation. Si les deux essais correspondent, le mot de passe est changé.

- Fonctionnement de `passwd.c` :
  - Initialisation des variables pour lire les entrées clavier + authentification:

```
char new[PASSWORD_MAX];
 char new2[PASSWORD_MAX];
char *name = argv[1];
char password[PASSWORD_MAX];
fgets(password, PASSWORD_MAX, stdin);
if (!authenticate(name, password))
{
      return 1;
   }
```

- Récupération des nouveaux mot de passe + vérification de la correspondance:

```
fgets(new, PASSWORD_MAX, stdin);
   fgets(new2, PASSWORD_MAX, stdin);

   if (strcmp(new, new2) != 0)
   {
      printf("passwords do not match\n");
      return 1;
   }
```

- On ouvre le fichier contenant les passwords, on le parcourt jusqu'à trouver le username correspondant à celui dont le password a été modifié. Lorsqu'on a trouvé le bon utilisateur, on change son mot de passe par celui entré précédemment (`strcpy(a.password, new)`). Une fois le mot de passe changé, on ré-encrypte le username et le password afin de les écrire par la suite dans le fichier `/etc/passwords` au bon emplacement.

```
   /* Search in the  passwords file. */
   while (read(file, &a, sizeof(struct account)))
   {
         account_decrypt(a.name, USERNAME_MAX, KERNEL_HASH);
      /* No this user. */
      if (strcmp(name, a.name) != 0 )
            continue;

      strcpy(a.password, new);
      account_encrypt(a.name, USERNAME_MAX, KERNEL_HASH);
      account_encrypt(a.password, PASSWORD_MAX, KERNEL_HASH);

      // changer mot de passe
      lseek(file, -sizeof(a), SEEK_CUR);
      write(file, &a, sizeof(a));
      printf("Password succesfully updated.\n");
      return 0;
   }
```

- Trace d'exécution:

```
% passwd user2
md2
new_mdp
new_mdp
Password succesfully updated.
```

Utilisation d'un dictionnaire

Pour implémenter la vérification des mots de passe trop courants, nous avons :

- **Création un fichier dictionnaire**

  Ajout d'un fichier `popular` contenant 150 mot de passe courants (en français/anglais) dans `tools/` :

```
123456
password
azerty
qwerty
...
```

- **Intégration dans** `build-img.sh`

  Nouvelle fonction `popular()` qui copie le fichier dictionnaire dans `/etc/` (comme déjà implémenté pour `passwords` :

```
function popular
{
   file="popular"

   # chmod 666 tools/$file
   # Let's care about security...
   if [ "$EDUCATIONAL_KERNEL" == "0" ]; then
      chmod 600 tools/$file
   fi

   bin/cp.minix $1 tools/$file /etc/$file $ROOTUID $ROOTGID

   # House keeping.
   rm -f $file
}
```

1. **Vérification dans** `passwd.c`

   Ajout de 2 fonctions :

   - `remove_newline()` : retire les `\n` des saisies utilisateur car nous avions des problèmes lors de la lecture des lignes dans le dictionnaire car chaque passwords finissaient par `\n` .

```
void remove_newline(char *str)
{
   size_t len = strlen(str);
   if (len > 0 && str[len - 1] == '\n')
      str[len - 1] = '\0';
}
```

   - `verify_dictionary()` : Vérifie si le mot de passe existe dans `/etc/popular`

```
static int verify_dictionary(const char *password)
{
   FILE *dict = fopen("/etc/popular", "r") ;
   char line[PASSWORD_MAX];
   if (dict  == NULL)
   {
      fprintf(stderr, "password dictionary unavailable at %s\n", "/etc/popular");
      return 0;
   }
```

```
    int i = 0;

    while (fgets(line, sizeof(line), dict))
    {
        remove_newline(line);
        if (strcmp(line, password) == 0)
        {
            fclose(dict);
            return 1;
        }
        i++;
    }

    fclose(dict);
    return 0;
}
```

Cette fonction vérifie si un mot de passe est présent dans un fichier de mots de passe courants (`/etc/popular`). Elle parcourt chaque ligne du fichier, enlève le caractère de nouvelle ligne, et compare avec le mot de passe donné. Si une correspondance est trouvée, elle retourne `1`, sinon `0`.

Appel avant de retaper le mot de passe à nouveau :

```
if (verify_dictionary(new)) {
    fprintf(stderr, "Password rejected: too common\n");
    return 1;
}
```

Résultat :

```
% passwd user1
mdp
marseille
Password rejected: too common (found in dictionary)
% passwd user1
mdp
valid
valid
Password succesfully updated.
```

`marseille` est bien présent dans le fichier `popular` mais pas `valid` .