Łamanie haseł

Zakres ćwiczenia

Celem ćwiczenia jest podniesienie umiejętności w zakresie stosowania dotąd nabytej wiedzy. Poznasz kilka nowych funkcji biblioteki PVM oraz napiszesz samodzielnie program, w którym samodzielnie zadecydujesz, w jaki sposób podzielić obliczenia między procesy.

Przedstawienie problemu

Naszym celem dzisiaj będzie złamanie zakodowanych haseł. Napiszesz program, który będzie otrzymywał zaszyfrowany (za pomocą algorytmu DES) łańcuch tekstowy i który będzie jaka była pierwotna postać tego łańcucha. Osiągniemy to za pomocą metody brutalnego ataku. Będziemy po kolei szyfrowali kolejne łańcuchy tekstowe i sprawdzali, czy po zaszyfrowaniu pasują one do otrzymanego wzorca. Jeżeli tak, to zaszyfrowany łańcuch jest szukanym „hasłem". Jeżeli nie, będziemy szukać dalej.

Aby uprościć problem, wprowadzimy dodatkowe ograniczenie na długość i rodzaj „hasła" – umówmy się, że jego długość wynosi na pewno 5 znaków i wszystkie znaki są małymi literami. Tak więc możliwe hasła są z zakresu od „aaaaa" do „zzzzz". W metodzie naszej będziemy więc kodować ciągi znaków „aaaaa", „aaaab", „aaaac"... i sprawdzać, czy po zakodowaniu są identyczne jak podane zaszyfrowane hasło. Jeżeli podana zakodowana postać hasła wyglądać będzie na przykład „aaXYZxyz", a kodując tekst „xyzxy" otrzymamy również łańcuch „aaXYZxyz", to oznacza, że poszukiwanym hasłem jest właśnie „xyzxy".

W drugiej kolejności zajmiemy się problemem wykrycia zakończenia. Przedstawimy najprostsze możliwe rozwiązanie. Zagadnienie do rozwiązania przedstawia się następująco: może zdarzyć się sytuacja, w której jeden z procesów odgadnie już wcześniej hasło dużo wcześniej przed wszystkimi pozostałymi. Proces-*master* powinien w takim wypadku powiadomić wszystkie pozostałe procesy *slave*, że powinny zakończyć natychmiast pracę, tak by nie marnować niepotrzebnie zasobów maszyny wirtualnej.

Zadanie do samodzielnego wykonania

Hasła podawane do naszego programu będą kodowane za pomocą funkcji `crypt` (tak samo jak hasła użytkowników systemu Linuks). Funkcja ta posiada dwa parametry: łańcuch do zaszyfrowania, oraz zarodek. Dla ułatwienia zarodek będziemy zawsze podawać jako „aa" – zresztą, łatwo zauważyć, że w zakodowanym haśle zarodek jest określony przez dwa pierwsze znaki (i dlatego w rzeczywistych zastosowaniach nie jest przechowywany). Można więc przyjąć, że hasło zostało zakodowane za pomocą następującego fragmentu kodu:

```
char * zakodowane_haslo=crypt(haslo, "aa");
```

Postać wykorzystywanego przez nas pliku `def.h` zmieni się odrobinę.

Plik def.h

```
1. #ifndef PUTCRYPTH
2. #define PUTCRYPTH
3. #define SLAVENUM 10
4. #define MSG_RES 11
5. #define MSG_END 12
6. #define MSG_SLV_END 12
7. #define _XOPEN_SOURCE
8. #include <unistd.h>
9. #include <stdio.h>
10. #include "pvm3.h"
11. #endif
```

W celu uzyskania dostępu do prawidłowej deklaracji zapowiadającej funkcji crypt, należy przed załączeniem pliku `unistd.h` zdefiniować stałą `_XOPEN_SOURCE` (linijka 7). Jest to bardzo ważne, w przeciwnym razie zostanie załączona nieprawidłowa postać nagłówka tej funkcji, co może prowadzić do błędów.

Szkielet programu *mastera* przedstawiony jest poniżej.

`Program master.c`

```
1. #include "crypt.h"
2.
3. int main(int argc, char **argv)
4. {
5.         int tids[SLAVENUM];
6.         char res[64]={0};
7.         int i;

8.         if (argc<2)
9.         {
10.              printf("Za malo argumentow. Podaj zakodowane \
                                                    haslo");
11.              exit(0);
12.        }

13.        pvm_catchout(stdout);
14.        int nproc=pvm_spawn("slave", argv, \
                       PvmTaskArch, "LINUX64", SLAVENUM, tids);
```

Jak widać, szkielet ten nie jest specjalnie zaawansowany. Jeżeli użytkownik nie poda zakodowanego hasła (liczba argumentów < 2; `argc` zawsze jest równe co najmniej jeden, gdyż, jak wiemy, `argv[0]` to nazwa programu) pojawia się komunikat o błędzie i następuje koniec programu.

W linijce 13 pojawia się dotąd nie omawiana funkcja `pvm_catchout`. Jest ona użyteczna wtedy, gdy uruchamiamy program nie z poziomu konsoli PVM za pomocą komendy `spawn`, ale z linii poleceń i pełni ona funkcję analogiczną jak opcja -> dla komendy `spawn`.

W linijce 14 *master* uruchamia `SLAVENUM` procesów potomnych. Linijka ta przedstawia także sposób wykorzystania trzeciego argumentu funkcji `pvm_spawn`. Flaga `PvmTaskArch` mówi, że chcemy uruchomić zadania na konkretnej architekturze, a nie na dowolnym węźle. Kolejny argument precyzuje, że chodzi nam o architekturę `LINUX64`.

Programy korzystające z funkcji crypt muszą być kompilowane razem z odpowiednią biblioteką w następujący sposób:

```
gcc master.c –lcrypt –lpvm3 –o master
gcc slave.c –lcrypt –lpvm3 –o slave
```

Program uruchom za pomocą komendy konsoli `spawn`:

```
spawn -> master aaWXLYPWXXQLs
```

Alternatywnym sposobem jest uruchomienie programu za pomocą wydania komendy z zwykłej linii poleceń powłoki, przebywając oczywiście w odpowiednim katalogu w którym znajduje się plik wykonywalny `master`. Jak pamiętamy z jednych z poprzednich ćwiczeń, w przypadku, gdy nasza aplikacja się zablokuje, niektóre wiadomości wysyłane na standardowe wyjście możemy nie zobaczyć. Czasami uruchamiając program w ten sposób uzyskujemy większą szansę na ujrzenie takich wiadomości:

```
./master aaWXLYPWXXQLs.
```

Wskazówki do zadania

Porównywanie dwóch łańcuchów tekstowych można dokonać za pomocą funkcji `strcmp`. Zwraca ona wartość 0, jeżeli dwa łańcuchy podane jako argumenty dla funkcji są identyczne. Można więc wykorzystać następujący fragment kodu

```
if ((strcmp(haslo,lancuch))==0)
/* jeżeli łańcuchy "haslo" i "lancuch" są identyczne.. */
```

Nie musisz szukać tablicy kodów ASCII w celu tworzenia kolejnych słów z zakresu od „aaaaa" do „zzzzz". Jeżeli zdefiniujesz tablicę sześcioznakową (szósty znak konieczny, w nim znajdzie się zero terminujące łańcuch) jako `char lancuch={0}` to możesz użyć następującej pętli:

```
for (j=0;j<'z'-'a';j++)
{
      lancuch[1]='a'+j;
```

Do podziału zadania między procesy być może będziesz im chciał wysyłać liczby określające, którą część zadania mają one wykonywać. Możesz zamiast tego skorzystać z numeru instancji w grupie. Skorzystanie z mechanizmu grup pozwoli ci skorzystać z rozgłaszania w grupie. Jak być może pamiętasz, numer instancji dowolnego procesu możesz pobrać za pomocą funkcji biblioteki PVM o nazwie `pvm_getinst`.

Pamiętaj, że nie ma sensu tworzyć zbyt dużej liczby procesów, o ile nie dodasz równocześnie wystarczająco dużej ilości węzłów. Liczbę węzłów oraz inne informacje na temat maszyny wirtualnej możesz uzyskać za pomocą funkcji `pvm_config`.

Zadanie do samodzielnego wykonania – detekcja zakończenia

Jeżeli udało ci się wykonać pierwszą część dzisiejszego ćwiczenia, możemy teraz zająć się kolejnym problemem. Jeżeli hasło będzie brzmiało „aaaaa", to zostanie błyskawicznie odgadnięte przez pierwszy proces, podczas gdy pozostałe będą kontynuowały pracę, niepotrzebnie zajmując czas procesora węzłów. Należałoby więc powiadomić pozostałe procesy *slave*, że mogą przerwać łamanie haseł i zakończyć się.

W stosunku do zagadnień znanych ci z wykładu, jest to wyjątkowo trywialny problem i nazywanie go detekcją zakończenia jest być może nawet pewnym nadużyciem, gdyż posiadamy tutaj znany z góry, wyróżniany proces, który czeka na zakończenie tylko dokładnie jednego procesu (jeżeli prawidłowo zaimplementujesz algorytm, tylko jeden proces odgadnie hasło).

Rozwiązaniem byłoby wysyłanie do *mastera* specjalnej wiadomości mówiącej o tym, że proces *slave* odgadł poprawnie hasło. Po odebraniu tej wiadomości, *master* powinien natychmiast wysyłać specjalną wiadomość do pozostałych procesów, tak by one również opuściły środowisko PVM (za pomocą `pvm_exit`) i zakończyły pracę (za pomocą `exit`).

```
1.  pvm_recv( -1, MSG_HASLO );
2.  pvm_upkstr( res );
3.  printf( "Haslo zlamane: %s \n", res );
4.  pvm_initsend( PvmDataRaw );
5.  pvm_bcast("grupa",MSG_END);
```

Pojawia się jedynie problem, w jaki sposób pozostałe procesy mają odbierać wiadomość żądającą zakończenia pracy. Po pierwsze, mogłyby one przetwarzać dane blokami i po zakończeniu przetwarzania danego bloku mogłyby wysyłać do *mastera* prośbę o kolejne dane. Proces *master* odsyłałby im albo kolejną porcję danych (na przykład, pojedynczą liczbę oznaczającą, od której litery w alfabecie należy budować słowa do szyfrowania) albo specjalny znacznik oznaczający „koniec przetwarzania". Jeżeli ten znacznik byłby zawarty nie w komunikacie (umieszczony tam za pomocą `pvm_pkint`) tylko byłby *typem* komunikatu, to proces *slave* musiałby używać funkcji `pvm_recv` z wartością –1 dla filtra możliwych znaczników (oznaczających, że chce odbierać wiadomości wszystkich typów) i następnie sprawdzać jaką wiadomość (jakiego typu) otrzymał korzystając z funkcji `pvm_bufinfo`.

Z samego opisu tej metody widać, że jest ona dość skomplikowana i łamie wcześniej poznaną zasadę „im mniej komunikacji tym lepiej”. Lepszym rozwiązaniem byłoby żądanie, by proces co jakiś czas sprawdzał, czy nadszedł do niego komunikat mówiący o potrzebie zakończenia przetwarzania. Nie można w tym celu oczywiście korzystać z blokującej funkcji odebrania wiadomości. Na szczęście biblioteka PVM zawiera szereg funkcji przydatnych do naszego celu.

Pierwszą z funkcji możliwych do użycia jest `pvm_probe` o znaczeniu argumentów identycznym jak dla `pvm_recv`, która sprawdza, czy jest dostępna jakaś wiadomość pasująca do zadanych filtrów. Można też użyć nieblokującej funkcji odbioru komunikatów `pvm_nrecv`, która zwraca 0 w przypadku, gdy nie ma żadnej wiadomości do odebrania i wartość większą od zera (numer bufora komunikacyjnego, który potem można wykorzystać np. dla funkcji `pvm_bufinfo`) gdy udało się odebrać wiadomość. Wreszcie można skorzystać z funkcji `pvm_trecv`, której trzeci argument określa jak długo ma proces czekać na nadejście oczekiwanej wiadomości.

```
1. /* przykład użycia funkcji pvm_probe */
2. if (pvm_probe( pvm_parent(), MSG_END )
3. pvm_recv( pvm_parent(), MSG_END );
4. /* przykład użycia funkcji pvm_trecv */
5. struct timeval tv;
6. tv.tv_sec = 1;
7. if (pvm_trecv( -1, MSG_END, (struct timeval *)&tv)
8.         pvm_upkint( &number, 1, 1 );
```

Którą z tych funkcji wybierzesz, zależy od ciebie. Zmodyfikuj teraz swój program tak, by zakończenie pracy przez jednego z procesów *slave* było wykrywane przez proces *master* i powodowało zatrzymanie całego przetwarzania.

Wynik działania ostatecznej wersji programu (dla 10 procesów typu *slave*) może wyglądać tak, jak jest to pokazane poniżej, przy czym treść złamanego hasła zastąpiono łańcuchem tekstowym XXXXX, a niektóre komunikaty pominięto:

```
pvm> spawn -> crypt-master aaWXLYPWXXQLs
[1]
1 successful
t8000f
[1:t8000f] [t40022] BEGIN
[1:t8000f] [tc0016] BEGIN
[1:t8000f] [t40023] BEGIN
[1:t8000f] [t100016] BEGIN
[1:t8000f] [t180017] BEGIN
[1:t8000f] [t140016] BEGIN
[1:t8000f] [tc0017] BEGIN
[1:t8000f] [t180018] BEGIN
[1:t8000f] [t140017] BEGIN
[1:t8000f] [t100017] BEGIN
[1:t8000f] Czekam na odpowiedz od 10 procesow
[1:t8000f] [tc0016] Czekam na barierze aaWXLYPWXXQLs
[1:t8000f] [tc0016] Ok, zaczynam prace 6
[1:t8000f] [t140016] Czekam na barierze aaWXLYPWXXQLs
[1:t8000f] [t140016] Ok, zaczynam prace 8
[1:t8000f] [t100016] Czekam na barierze aaWXLYPWXXQLs
[1:t8000f] [t100016] Ok, zaczynam prace 4
[1:t8000f] [t40022] Czekam na barierze aaWXLYPWXXQLs
[1:t8000f] [t40022] Ok, zaczynam prace 0
[1:t8000f] [t180017] Czekam na barierze aaWXLYPWXXQLs
[1:t8000f] [t180017] Ok, zaczynam prace 3
[1:t8000f] [t40023] Czekam na barierze aaWXLYPWXXQLs
[1:t8000f] [t40023] Ok, zaczynam prace 1
[1:t8000f] [tc0017] Czekam na barierze aaWXLYPWXXQLs
[1:t8000f] [tc0017] Ok, zaczynam prace 7
[1:t8000f] [t140017] Czekam na barierze aaWXLYPWXXQLs
[1:t8000f] [t140017] Ok, zaczynam prace 9
[1:t8000f] [t100017] Czekam na barierze aaWXLYPWXXQLs
[1:t8000f] [t100017] Ok, zaczynam prace 5
[1:t8000f] [t180018] Czekam na barierze aaWXLYPWXXQLs
[1:t8000f] [t180018] Ok, zaczynam prace 2
[1:t8000f] Haslo zlamane:
[1:t8000f] [t40022] ..Udalo sie: XXXXX aaWXLYPWXXQLs
aaWXLYPWXXQLs 0
[1:t8000f] [t40022] EOF
[1:t8000f] [t40023] ..1 Nie znalazlem od c do d
[1:t8000f] [t40023] EOF
[1:t8000f] [t180018] ..Znacznik konca od procesu master!
[1:t8000f] [t180018] EOF
[1:t8000f] [t100017] ..5 Nie znalazlem od k do l
[1:t8000f] [t100017] EOF
[1:t8000f] [tc0017] Znacznik konca od procesu master!
[1:t8000f] [tc0017] EOF
[1:t8000f] [t140017] ..9 Nie znalazlem od s do t
[1:t8000f] [t140017] EOF
[1:t8000f] [t180017] .Znacznik konca od procesu master!
[1:t8000f] [t180017] EOF
[1:t8000f] [t100016] .. Znacznik konca od procesu master!
[1:t8000f] [t100016] EOF
[1:t8000f] [tc0016] ..6 Nie znalazlem od m do n
 1:t8000f] [tc0016] EOF
[1:t8000f] EOF
[1] finished
```

Poznane funkcje biblioteki PVM

`int info = pvm_catchout(FILE *ff)`

Domyślnie PVM zapisuje standardowe wyście *stdout* i wyjście diagnostyczne *stderr* do specjalnego pliku zwanego logiem o lokalizacji /tmp/pvml/<*uid*>. Funkcja `pvm_catchout` powoduje przechwytywanie wyjścia procesów uruchomionych za pomocą `pvm_spawn`. Znaki umieszczane w strumieniach *stdout* i *stderr* przez zadania .dzieci. zbierane są i przesyłane w komunikatach kontrolnych do procesu rodzica, gdzie każda linia opatrzona zostaje identyfikatorem procesu i wyświetlona na standardowym wyjściu. W przypadku wywołania funkcji `pvm_exit` przez proces rodzica przechwytującego wyjścia dzieci proces ten zostanie wstrzymany aż do chwili, gdy wszystkie procesy potomne opuszczą maszynę wirtualną.

`int bufid = pvm_nrecv(int tid, int tag)`

Funkcja analogiczna do funkcji `pvm_recv`, z tą tylko różnicą , że jest nieblokująca. Jeśli nie ma komunikatu, który mógłby zostać odebrany zwracana jest wartość 0.

`int bufid = pvm_probe(int tid, int msgtag)`

Jeśli żądany komunikat nie nadszedł `pvm_probe` zwraca 0. W przeciwnym przypadku zwraca identyfikator bufora, ale nie odbiera komunikatu. Funkcja ta wykorzystywana jest do określania czy komunikat już nadszedł.

`int bufid = pvm_trecv(int tid, int msgtag, struct timeval *timeout)`

PVM dostarcza również funkcję blokującego odbiory zaopatrzoną w mechanizm *timeout-u*. Czekanie na nadejście komunikatu ogranicza się tylko do zdefiniowanego przez użytkownika periodu czasowego, po którym zwracana jest informacja o braku komunikatu o ile ten wcześniej nie nadszedł. Jeśli użytkownik zdefiniuje długi czas oczekiwania na komunikat funkcja ta upodabnia się do `pvm_recv`. Jeśli czas będzie krótki bądź 0 to możemy mówić o funkcjonalności zbliżonej do polecenia `pvm_nrecv`. Jak widać jest to więc funkcja pośrednia pomiędzy blokującym a nieblokującym odbiorem.

`int info = pvm_config(int *nhost, int *narch, struct pvmhostinfo **hostp)`

Funkcja `pvm_config` zwraca informację o konfiguracji maszyny wirtualnej włączając w to ilość węzłów, oraz różnych architektur. `hostp` jest wskaźnikiem na tablicę struktur `pvmhostinfo`. Każda z tych struktur zawiera *TID* demona, nazwę węzła, rodzaj architektury i relatywną prędkość maszyny, na której działa demon.

Podsumowanie

W wyniku obecnych ćwiczeń napisałeś samodzielnie rozproszony program łamiący hasła metodą brutalnego ataku oraz rozwiązałeś bardzo prostą postać problemu detekcji zakończenia (w niezbyt elegancki sposób).

Co powinieneś wiedzieć:

- Jakie funkcje można zastosować do nieblokującego odbierania wiadomości (`pvm_nrecv`, `pvm_trecv`) oraz jak sprawdzić, czy jest dostępna jakaś wiadomość gotowa do odbioru (`pvm_probe`)