Katedra Informatyki

Wydział Informatyki i Telekomunikacji
Politechnika Krakowska

Programowanie Usług Sieciowych

mgr inż. Michał Niedźwiecki

Uprawnienia

Laboratorium: 09,

system operacyjny: Linux

Kraków, 2022

Spis treści

1. Wiadomości wstępne

Pierwsza część niniejszej instrukcji zawiera podstawowe wiadomości teoretyczne dotyczące uprawnień w systemie Linux, a w szczególności tzw. zdolności (ang. capabilities). Poznanie tych wiadomości umożliwi prawidłowe zrealizowanie praktycznej części laboratorium.

1.1. Tematyka laboratorium

Tematyką laboratorium jest programowanie bezpiecznych aplikacji w oparciu o systemowe mechanizmy kontroli uprawnień (bazujące na tożsamości użytkownika) oraz tzw. zdolności (ang. capabilities).

W systemie Linux, program jest wykonywany z uprawnieniami użytkownika, który ten program uruchomił. Ta zasada z powodzeniem funkcjonuje od ponad 30 lat. Problem pojawia się jednak w przypadku, gdy użytkownik próbuje za pośrednictwem programu wykonać operację, do której nie posiada uprawnień. Z pomocą przychodzi wówczas bit SUID lub mechanizm capabilities. SUID (ang. set user ID) pozwala użytkownikowi na wykonanie programu z uprawnieniami właściciela pliku wykonywalnego, a zatem z uprawnieniami innymi niż jego własne. Jest to potencjalnie niebezpieczna operacja. Jeżeli właścicielem pliku z ustawionym bitem SUID jest administrator (root), zwykły użytkownik może wykorzystać każdy, nawet drobny błąd w programie do zdobycia praw administratora. Są to maksymalne możliwe uprawnienia i często aplikacja, która chce wykonać uprzywilejowaną operację nie potrzebuje aż takiej władzy. Z tego powodu, od wielu lat szukano sposobu na odejście od mechanizmu SUID. W efekcie opracowano zestaw przywilejów (capabilities), które pozwalają indywidualnie zaadresować potrzeby każdej aplikacji.

Część laboratorium poświęcona jest demonom. Demonem nazywamy proces działający w tle i uniezależniony od terminalu kontrolującego. Procesy-demony bardzo często do realizacji powierzonych im funkcji wymagają specjalnych uprawnień.

1.2. Zagadnienia do przygotowania

Przed przystąpieniem do realizacji laboratorium należy zapoznać się z zagadnieniami dotyczącymi:

- zarządzania tożsamością
  (identyfikatory procesów, użytkowników, grup, …) [1 - 3]
- stosowania bitów SUID, SGID [1 - 3]
- funkcji fork() oraz execve() [1]
- demonów [1, 2, 5]
- mechanizmu capabilities [6, 7, 8]
- stosowania narzędzi setcap i getcap [9]
- interfejsu programistycznego biblioteki libcap [10]

Literatura:

[1] W. Richard Stevens, Stephen A. Rago, „Advanced Programming in the UNIX® Environment”, Addison Wesley Professional

[2] W. R. Stevens, „Programowanie Usług Sieciowych”, „API: gniazda i XTI”

[3] MAN (7), „credentials”

[4] MAN (2), „fork”, „execve”

[5] Robert Love, „Linux System Programming”, O’Reilly

[6] MAN (7), „capabilities”

[7] Jinkai Gao, „How Linux Capability Works in 2.6.25”, Syracuse University

[8] S. E. Hallyn, A. G. Morgan, „Linux Capabilities: making them work”, Proceedings of the Linux Symposium, Ottawa, 2008

[9] MAN (8), „setcap”, „getcap”

[10] MAN (3), „cap_init”, „cap_free”, „cap_dup”, „cap_get_proc”, „cap_set_proc”, „cap_clear”, „cap_clear_flag”, „cap_get_flag”, „cap_set_flag”, „cap_from_text”, „cap_from_name”, „cap_to_text”, „cap_to_name”

1.3. Opis laboratorium

1.3.1. Identyfikatory procesów

Każdy proces w systemie Linux posiada następujący zestaw identyfikatorów[3]:

- PID (ang. Process ID),
- PPID (ang. Parent Process ID),
- Process Group ID,
- Session ID,
- Real user ID, real group ID,
- Effective user ID, effective group ID,
- Saved set-user-ID, saved set-group-ID,
- File system user ID, file system group ID,
- Supplementary group IDs.

Specyfikacja POSIX wymaga, aby identyfikatory wymienione w tym podrozdziale były współdzielone przez wszystkie wątki procesu.

PID jest nieujemną liczbą całkowitą, która jest przypisywana do procesu za pomocą funkcji fork(). PID jest reprezentowany przez typ pid_t zdefiniowany w pliku <sys/types.h>. Wywołanie funkcji execve() i nadpisanie obrazu procesu nie zmienia wartości PID. Proces może uzyskać swój identyfikator za pomocą funkcji getpid().

PPID identyfikuje proces rodzica. Podobnie jak PID, PPID jest reprezentowany przez typ pid_t i nie zmienia wartości po wywołaniu funkcji execve(). Proces może uzyskać identyfikator procesu macierzystego za pomocą funkcji getppid().

```
#include <sys/types.h>
#include <unistd.h>
```

```
pid_t getpid(void);
pid_t getppid(void);
```

Każdy proces należy do określonej grupy procesów. Grupa procesów jest kolekcją złożoną z jednego lub więcej procesów, które posiadają ten sam Process Group ID.

Grupa procesów może otrzymywać sygnały od jądra systemu lub innych procesów, tzn. sygnał może zostać wysłany do wszystkich procesów danej grupy. Zazwyczaj procesy w grupie związane są z określonym zadaniem (ang. job). Przykładowo, dla polecenia:

```
$ ls | wc -w
```

powłoka tworzy grupę złożoną z dwóch procesów realizujących jedno zadanie (zliczanie liczby słów).

Grupa może posiadać lidera (ang. process group leader). Lider grupy jest procesem, którego identyfikator jest równy identyfikatorowi grupy.

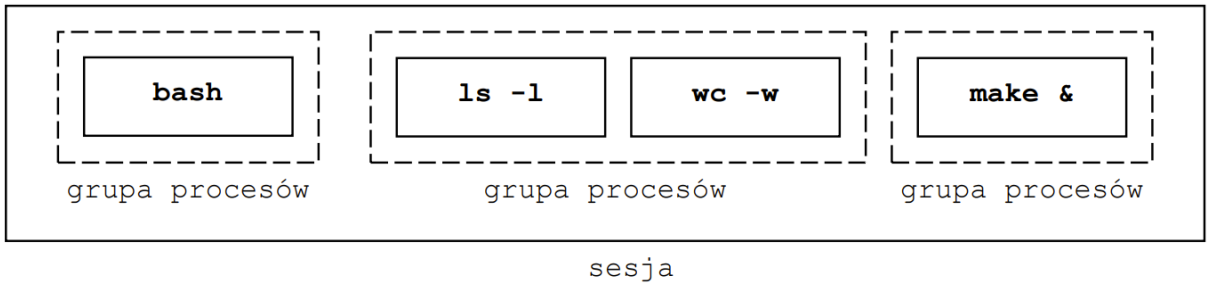
Proces potomny utworzony za pomocą funkcji fork() dziedziczy identyfikator grupy po rodzicu. Identyfikator grupy procesów nie zmienia swojej wartości po wywołaniu funkcji execve().

Sesja jest kolekcją złożoną z jednej lub więcej grup procesów. Sesje związane są z powłokami systemu Linux, a ich głównym zadaniem jest powiązanie aktywności zalogowanego użytkownika z terminalem kontrolującym.

Grupy procesów, które wchodzą w skład sesji są dzielone przez system operacyjny na:

- grupę procesów pierwszoplanowych (ang. foreground),
- jedną lub więcej grup procesów drugoplanowych (ang. background).

W skład sesji może wchodzić tylko jedna grupa procesów pierwszoplanowych. Różnica między procesami pierwszoplanowymi, a procesami drugoplanowymi wiąże się bezpośrednio z zachowaniem powłoki systemu, która w przypadku uruchomionych procesów pierwszoplanowych, musi czekać na ich zakończenie. Innymi słowy, tylko procesy pierwszoplanowe „blokują” powłokę. Rys. 1 przedstawia relacje między procesami, grupami procesów oraz sesją.



Rys. 1. Zależność między sesją, grupami procesów oraz procesami.

Sytuacja przedstawiona na rys.1 mogła zostać osiągnięta przez wydanie w powłoce bash następujących poleceń:

```
$ make &

$ ls -l | wc -w
```

Tylko procesy ls oraz wc tworzą grupę procesów pierwszoplanowych.

Proces potomny utworzony za pomocą funkcji fork() dziedziczy identyfikator sesji po rodzicu. Identyfikator sesji nie zmienia swojej wartości po wywołaniu funkcji execve().

Proces może ustanowić nową sesję za pomocą funkcji setsid():

```
#include <unistd.h>

pid_t setsid(void);
```

Jeżeli proces, który wywołał funkcję setsid() nie jest liderem grupy, to utworzona zostaje nowa sesja. Ponadto:

- Proces staje się liderem sesji (liderem sesji jest proces, który ją utworzył).
- Utworzona zostaje nowa grupa procesów, a proces, który wywołał funkcję setsid() - jedyny proces w grupie - staje się jej liderem.
- Proces jest pozbawiony terminalu kontrolującego.

Funkcja setsid() zwraca identyfikator sesji lub -1 w przypadku błędu. Funkcja zwróci błąd jeżeli proces, który ją wywołał jest liderem grupy. Aby zapobiec tej sytuacji, dobrą praktyką jest wywołanie funkcji fork(), zakończenie procesu macierzystego i wywołanie setsid() w procesie potomnym. Utworzenie nowego procesu za pomocą funkcji fork() gwarantuje, że nie będzie on liderem grupy (identyfikator grupy procesów odziedziczony przez proces potomny jest różny od PID procesu potomnego).

Sesje wiążą użytkownika z terminalem kontrolującym:

- Zamknięcie terminalu przez użytkownika powoduje wysłanie sygnału SIGQUIT do wszystkich procesów w grupie procesów pierwszoplanowych.
- Jeżeli terminal wykryje zamknięcie połączenia sieciowego, to do procesów pierwszoplanowych zostanie wysłany sygnał SIGHUP.
- Wprowadzenie sekwencji przerwania (zazwyczaj CTRL+C) powoduje wysłanie do procesów pierwszoplanowych sygnału SIGINT.

Domyślną dyspozycją dla wymienionych sygnałów jest zakończenie procesu. Utworzenie nowej sesji i pozbycie się zależności od terminalu kontrolującego zapewnia wyizolowane środowisko, które jest wykorzystywane przez demony i ich procesy potomne (podrozdział 1.3.2).

Każdy proces jest powiązany z zestawem identyfikatorów użytkownika i grup. Identyfikatory te są reprezentowane przez typy uid_t oraz gid_t zdefiniowane w pliku <sys/types.h>. W systemie Linux, procesy są związane z następującymi identyfikatorami:

- Real user ID, real group ID

Służą do określenia tożsamości właściciela procesu, tzn. użytkownika, który uruchomił program i grupy, do której ten użytkownik należy. Identyfikatory można pobrać za pomocą funkcji getuid() oraz getgid():

```
#include <unistd.h>
#include <sys/types.h>

uid_t getuid(void);
gid_t getgid(void);
```

- Effective user ID, effective group ID

Efektywne identyfikatory są wykorzystywane przez jądro systemu podczas sprawdzania uprawnień procesu do wykonywania uprzywilejowanych operacji. W systemach Unix (ale nie Linux) identyfikatory te są wykorzystywane również podczas sprawdzania praw dostępu do plików. Identyfikatory można pobrać za pomocą funkcji geteuid() oraz getegid():

```
#include <unistd.h>
#include <sys/types.h>

uid_t geteuid(void);
gid_t getegid(void);
```

Uruchomienie programu z ustawionym bitem SUID/SGID powoduje, że efektywne identyfikatory odpowiadają właścicielowi pliku wykonywalnego, a nie użytkownikowi, który uruchomił program.

- Saved set-user-ID, saved set-group-ID

Te identyfikatory stanowią kopię identyfikatorów efektywnych po wywołaniu funkcji execve(). Programy zwykłych użytkowników mogą zmieniać wartość efektywnych identyfikatorów na wartość identyfikatorów rzeczywistych lub identyfikatorów saved set-user-ID (saved set-group-ID). Identyfikatory można pobrać za pomocą funkcji getresuid() oraz getresgid():

```
#define _GNU_SOURCE
#include <unistd.h>

int getresuid(uid_t *ruid, uid_t *euid, uid_t *suid);
int getresgid(gid_t *rgid, gid_t *egid, gid_t *sgid);
```

Parametr	Opis
`ruid/rgid`	wskaźnik na rzeczywisty identyfikator użytkownika/grupy
`euid/egid`	wskaźnik na efektywny identyfikator użytkownika/grupy
`suid/sgid`	wskaźnik na identyfikator *saved set-user-ID* (*saved set-group-ID*)

- File system user ID, file system group ID

Wykorzystywane w systemie Linux do sprawdzania praw dostępu do pliku. Za każdym razem, gdy zmienia się efektywny identyfikator procesu, jądro systemu zmienia identyfikator systemu plików na tą samą wartość. Wartość identyfikatorów systemu plików można jednak zmieniać niezależnie za pomocą funkcji setfsuid() oraz setfsgid().

- Supplementary group IDs

Dodatkowe identyfikatory grup są wykorzystywane w celu określenia praw dostępu do plików lub innych zasobów. Wersje jądra systemu Linux przed 2.6.4 ograniczały liczbę grup do 32. Od wersji 2.6.4 proces może być członkiem 65536 grup. Proces może uzyskać listę grup dodatkowych, do których należy za pomocą funkcji getgroups().

Proces potomny utworzony za pomocą funkcji fork() dziedziczy kopie identyfikatorów użytkownika i grup po rodzicu.

Podczas wywołania execve(), proces zachowuje rzeczywiste identyfikatory użytkownika i grupy oraz dodatkowe identyfikatory grup. Identyfikatory efektywne oraz saved set mogą ulec zmianie jeżeli ustawione są bity SUID/SGID.

Identyfikatory użytkownika i grup można modyfikować za pomocą kilku funkcji systemowych.

Funkcja setuid() modyfikuje efektywny identyfikator użytkownika. Jeżeli efektywny identyfikator użytkownika, który wywołał funkcję wynosi zero (root), to modyfikowany jest również identyfikator rzeczywisty i saved set. W tym przypadku (modyfikacja wszystkich trzech identyfikatorów) odzyskanie pełnych uprawnień jest niemożliwe, tzn. nie można ponownie zmienić efektywnego identyfikatora na wartość 0, jeżeli wszystkie identyfikatory mają wartość różną od zera. Odpowiednikiem funkcji setuid() dla efektywnego identyfikatora grupy jest funkcja setgid().

```
#include <sys/types.h>
#include <unistd.h>

int setuid(uid_t uid);
int setgid(gid_t gid);
```

Funkcja seteuid() modyfikuje efektywny identyfikator użytkownika. Procesy nieuprzywilejowane mogą zmieniać efektywny identyfikator na identyfikator rzeczywisty lub saved set. Odpowiednikiem funkcji seteuid() dla efektywnego identyfikatora grupy jest funkcja setegid().

```
#include <sys/types.h>
#include <unistd.h>

int seteuid(uid_t euid);
int setegid(gid_t egid);
```

Najbardziej zaawansowaną spośród funkcji służących do modyfikacji identyfikatorów użytkownika jest setresuid().

```
#define _GNU_SOURCE
#include <sys/types.h>
#include <unistd.h>

int seteuid(uid_t euid);
int setegid(gid_t egid);
```

Funkcja setresuid() definiuje identyfikator rzeczywisty, efektywny oraz saved set dla wywołującego ją procesu. Procesy nieuprzywilejowane mogą zmieniać każdy z wymienionych identyfikatorów na obecną wartość identyfikatora rzeczywistego, efektywnego lub saved set. Procesy uprzywilejowane mogą przypisać identyfikatorom dowolną, poprawną wartość.

1.3.2. Demony

Demon jest procesem wykonywanym w tle i uniezależnionym od terminala kontrolującego. Demony są najczęściej uruchamiane z konta roota lub innego specjalnego użytkownika podczas startu systemu. Konwencją jest, że nazwy demonów kończą się na literę „d", np.: sshd.

Biblioteka glibc implementuje funkcję daemon() odpowiedzialną za przekształcenie procesu w demona.

```
#include <unistd.h>

int daemon(int nochdir, int noclose);
```

Parametr	Opis
`nochdir`	jeżeli parametr ma wartość zero, funkcja zmienia bieżący katalog roboczy na „/"
`noclose`	jeżeli parametr ma wartość zero, funkcja przekierowuje standardowe wejście, standardowe wyjście i standardowe wyjście dla błędów na `/dev/null`

Aby zostać demonem bez wykorzystania funkcji daemon(), proces musi wykonać następujące kroki:

1. Wywołać funkcję fork(). Utworzony przez nią proces potomny będzie demonem.
2. W procesie macierzystym wywołać funkcję exit(). Proces potomny odziedziczył identyfikator grupy procesów po rodzicu i otrzymał nowy PID - nadany przez funkcję fork(). Ponieważ identyfikator grupy procesów jest różny od PID procesu, uzyskana została gwarancja, że proces potomny nie jest liderem grupy. Jest to wymogiem pomyślnego wykonania kolejnego kroku.
3. W procesie potomnym wywołać funkcję setsid(). Funkcja setsid() tworzy nową sesję oraz grupę procesów, do której należy tylko jeden proces. Na tym etapie proces nie posiada terminalu kontrolującego (po utworzeniu nowej sesji proces nie jest związany z żadnym terminalem).
4. Opcjonalnym krokiem jest zmiana katalogu roboczego (ang. working directory) odziedziczonego po procesie macierzystym na „/". Katalog roboczy uniemożliwia

domontowanie systemu plików, na którym się znajduje. Z tego względu jego zmiana na „/” może okazać się pomocna. Zmiany katalogu roboczego można dokonać za pomocą funkcji chdir().

```
#include <unistd.h>

int chdir(const char* path);
```

Parametr	Opis
path	ścieżka bezwzględna lub podawana względem bieżącego katalogu roboczego

5. Przekierować (opcjonalnie) standardowe wejście, standardowe wyjście i standardowe wyjście dla błędów na /dev/null. Gwarantuje to, że wywołania funkcji korzystających z tych deskryptorów w procesie-demonie nie zakończą się błędem. Logowanie błędów lub innych ważnych informacji może odbywać się za pomocą systemowego demona syslogd.

1.3.3. Capabilities

Tradycyjne implementacje systemu UNIX rozróżniają dwie kategorie procesów: procesy uprzywilejowane, których efektywny identyfikator użytkownika wynosi zero, oraz procesy nieuprzywilejowane, o efektywnym UID różnym od zera. Podczas gdy procesy uprzywilejowane pomijają mechanizmy kontroli uprawnień zaimplementowane w jądrze systemu, procesy nieuprzywilejowane są poddawane kontroli na podstawie danych uwierzytelniających (efektywnego UID, efektywnego GID, grup uzupełniających). Procesy nieuprzywilejowane muszą jednak czasem wykonać operacje wymagającą pewnych uprawnień. Przykładowo, zwyczajny użytkownik musi być w stanie zmienić swoje hasło za pomocą programu passwd. Rozwiązaniem jest ustawienie bitu SUID dla pliku wykonywalnego passwd, którego właścicielem jest root. Daje to możliwość uruchomienia programu z uprawnieniami roota przez dowolnego użytkownika. Mechanizm ten posiada zasadniczą wadę: program wymagający minimalnych uprawnień jest uruchamiany z wszystkimi możliwymi uprawnieniami. Błędy w oprogramowaniu mogą doprowadzić do kompromitacji systemu i przejęcia nad nim kontroli. W wersji jądra 2.2 systemu Linux wprowadzono zestaw przywilejów indywidualnie adresujących potrzeby danego programu. Uprawnienia roota zostały podzielone na zbiór przywilejów (ang. capabilities), które mogą zostać przydzielone wątkom niezależnie od siebie. Proszę zwrócić uwagę na fakt, że uprawnienia te są atrybutem wątków, a nie procesów. Implementacja uprawnień (capabilities) w systemie Linux bazuje na projekcie standardu POSIX 1003.1e (prace nad standardem zostały zawieszone w 1998 roku). Opis dostępnych przywilejów można uzyskać na stronie podręcznika systemowego capabilities (7) oraz w pliku nagłówkowym <linux/capability.h>.

Przykładowe przywileje opisane w projekcie standardu POSIX przedstawione są w poniższej tabeli.

Nazwa	Opis
CAP_CHOWN	zmiana właściciela pliku oraz grupy, do której należy plik
CAP_DAC_OVERRIDE	pominięcie sprawdzania uprawnień przy dostępie do pliku (włączając uprawnienia dostępu ACL – *Access Control Lists*); prawa do wykonania pliku są przyznawane, jeżeli co najmniej jeden z trzech bitów odpowiedzialnych za prawo wykonywalności jest ustawiony; DAC jest akronimem od *Discretionary Access Control*
CAP_DAC_READ_SEARCH	odczyt i przeszukiwanie katalogów; prawo odczytu plików
CAP_SETUID	pozwala na modyfikację identyfikatorów użytkownika za pomocą `set*uid()`

Przykładowe przywileje specyficzne dla systemu Linux zostały przedstawione w następującej tabeli.

Nazwa	Opis
CAP_NET_BIND_SERVICE	umożliwia powiązanie uprzywilejowanych portów (*well-known* ports) z gniazdem
CAP_NET_BROADCAST	umożliwia transmisję rozgłoszeniową (ang. *broadcast*) oraz nasłuchiwanie na transmisję typu *multicast*
CAP_NET_ADMIN	konfiguracja sieci, interfejsów, filtrowania pakietów, tablic routingu, definiowanie TOS, korzystanie z trybu *promiscuous*
CAP_NET_RAW	korzystanie z gniazd typu `SOCK_RAW` lub domeny komunikacyjnej (rodziny protokołów) `AF_PACKET`; strony podręcznika systemowego: `raw(7)`, `pf_packet(7)`
CAP_SYS_MODULE	ładowanie i usuwanie modułów jądra, modyfikacja jądra
CAP_SYS_ADMIN	szeroki zakres uprawnień administracyjnych: konfiguracja nazwy domenowej, montowanie systemów plików, włączanie i wyłączanie pamięci wymiany, usuwanie semaforów, …
CAP_SYS_BOOT	prawo użycia `reboot()`

Przywileje (capabilities) są wspierane przez wirtualny system plików (ang. Virtual File System) począwszy od wersji jądra 2.6.24. We wcześniejszych wersjach systemu Linux, możliwe było określenie przywilejów tylko dla uruchomionych procesów (za pomocą programów z pakietu libcap1). Było to rozwiązanie niepełne i tymczasowe. W systemach z wsparciem dla capabilities ze strony wirtualnego systemu plików, wykorzystuje się pakiet libcap2. Zawiera on narzędzia do odczytu i zapisu uprawnień dla plików: getcap i setcap. Niniejsza podrozdział poświęcony jest implementacji z wersji 2.6.25, w której ograniczający zbiór przywilejów (ang. capability bounding set) nie jest atrybutem globalnym, a atrybutem wątku[6].

Każdy wątek posiada 3 zbiory przywilejów:

- efektywny (ang. effective)
  Zbiór wykorzystywany przez jądro systemu do sprawdzania uprawnień wątku (zawiera uprawnienia wykorzystywane przez wątek w danej chwili). Zbiór effective jest podzbiorem zbioru permitted.
- dozwolony (ang. permitted)
  Jest to zbiór przywilejów jakie wątek może wykorzystać w zbiorze effective. Zbiór permitted ogranicza również przywileje jakie można dodać do zbioru inheritable, jeżeli wątek nie posiada uprawnienia CAP_SETPCAP w zbiorze effective. Jeżeli wątek usunie uprawnienie ze zbioru permitted, nie będzie w stanie go odzyskać (wyjątek od tej reguły może wystąpić podczas wywołania execve()[6]).
- dziedziczny (ang. inheritable)
  Zbiór przywilejów zachowywany po wywołaniu execve(). Przywileje w zbiorze inheritable są również wykorzystywane do określenia przywilejów w zbiorze permitted nowego procesu (wątku).

Wątek w procesie potomnym utworzonym za pomocą funkcji fork() dziedziczy zbiory przywilejów rodzica.

Począwszy od wersji 2.6.24 jądra systemu Linux, zbiory przywilejów mogą być powiązane z plikami wykonywalnymi za pomocą polecenia setcap. Przywileje te są określane jako tzw. file capabilities. Zbiory przywilejów plików są przechowywane w rozszerzonym atrybucie o nazwie security.capability. Rozszerzone atrybuty (ang. extended attributes) są wspierane przez większość systemów plików, w tym ext2, ext3, ext4, ReiserFS i XFS. Modyfikacja atrybutu security.capability przez wątek wymaga, aby posiadał on uprawnienie CAP_SETFCAP.

Plik wykonywalny może posiadać – podobnie jak wątek – trzy zbiory uprawnień:

- dozwolony (ang. permitted), znany też jako wymuszony (ang. forced)
- dziedziczny (ang. inheritable), określany wcześniej mianem dopuszczalnego (ang. allowed)
- efektywny (ang. effective), który jest zaimplementowany jako pojedynczy bit (ang. legacy bit).

Zbiory uprawnień przypisane plikom wykonywalnym, w połączeniu ze zbiorami uprawnień wątku determinują uprawnienia nowego wątku po wywołaniu execve(). Podczas wywołania execve() jądro systemu określa uprawnienia nowego wątku wg poniższej formuły[6]:

```
T’(inheritable) = T(inheritable)
T’(permitted) = [B & F(permitted)] | [T(inheritable) & F(inheritable)]
T’(effective) = F(effective) ? T’(permitted) : 0
```

gdzie:

- T – oznacza zbiór uprawnień wątku przed wywołaniem execve(),
- T’ – oznacza zbiór uprawnień wątku po wywołaniu execve(),
- F – oznacza zbiór uprawnień pliku wykonywalnego.
- B – oznacza zbiór ograniczający (ang. bounding set)

Uprawnienia ze zbioru F(permitted) są przyznawane nowemu wątkowi, jeżeli znajdują się w zbiorze ograniczającym B (równanie 2). Z reguły zbiór F(permitted) jest podzbiorem zbioru B. Z tego względu zbiór F(permitted) nazywa się zbiorem wymuszonym (ang. forced) – wszystkie jego uprawnienia znajdą się w zbiorze T'(permitted) nowego wątku.

Zbiór F(inheritable) nazywa się dopuszczalnym (ang. allowed) lub opcjonalnym (ang. optional). Nowy wątek uzyska jego uprawnienia, tylko jeżeli znajdują się one w zbiorze T(inheritable) wątku, który wywołał execve().

Jeżeli bit F(effective) jest ustawiony dla pliku wykonywalnego, to efektywny zbiór uprawnień nowego wątku jest kopią T'(permitted). W przeciwnym wypadku zbiór efektywny jest pusty (równanie 3).

Zbiór B - capability bounding set – ogranicza przywileje jakie może uzyskać wątek podczas wywołania execve(). Ponadto, wątek może dodać do zbioru inheritable tylko uprawnienia, które znajdują się w zbiorze B. Począwszy od wersji 2.6.25 jądra systemu Linux, zbiór ograniczający jest atrybutem wątku. Zbiór ograniczający jest dziedziczony przez wątek procesu potomnego podczas wywołania funkcji fork(), a wywołanie execve() nie modyfikuje zbioru w żaden sposób. Proces init, przodek wszystkich procesów posiada wszystkie możliwe uprawnienia w zbiorze ograniczającym. Wątki mogą usuwać uprawnienia ze zbioru ograniczającego za pomocą funkcji prctl(), ale nie mogą dodawać uprawnień do zbioru.

```
#include <sys/prctl.h>

int prctl(int option, unsigned long arg2, unsigned long arg3,
    unsigned long arg4, unsigned long arg5);
```

Pierwszy parametr funkcji prctl() określa typ przeprowadzanej operacji. Możliwe operacje są zdefiniowane w pliku <linux/prctl.h>. Znaczenie pozostałych parametrów różni się w zależności od parametru option. Dla wartości PR_CAPBSET_DROP znaczenie ma tylko parametr arg2, który określa uprawnienie (capability) do usunięcia ze zbioru ograniczającego. Nazwy uprawnień i odpowiadające im numery zostały zdefiniowane w pliku <linux/capability.h>. Wątek, który wywołuje funkcję prctl z opcją PR_CAPBSET_DROP musi posiadać uprawnienie CAP_SETPCAP.

W celu zachowania kompatybilności z tradycyjnym modelem uprawnień systemu Unix, obowiązują następujące reguły podczas wywołania execve():

- Jeżeli rzeczywisty UID wątku jest równy zero lub efektywny UID wątku wynosi zero, to zbiory F(inheritable) oraz F(permitted) posiadają wszystkie uprawnienia. Taka sytuacja ma miejsce, gdy wątek wywołujący execve() wykonuje się w kontekście użytkownika root (rzeczywisty UID równy 0) lub został uruchomiony program z ustawionym bitem SUID (efektywny UID równy 0).
- Jeżeli efektywny UID wątku utworzonego po wywołaniu execve() wynosi zero, efektywny bit (zwany jako legacy bit) pliku jest ustawiany. Ta reguła w połączeniu z poprzednią powoduje, że wątek posiada wszystkie uprawnienia (poza maskowanymi przez zbiór ograniczający) w zbiorach T'(effective) i T'(permitted).

Dodatkowe reguły obowiązują podczas zmian identyfikatorów użytkownika za pomocą funkcji setuid(), setresuid() i podobnych:

- Jeżeli jeden lub więcej z identyfikatorów: rzeczywisty UID, efektywny UID, saved set UID, file system UID był zerem i wszystkie wymienione identyfikatory zostały zmienione na wartość różną od zera, to wszystkie uprawnienia są usuwane ze zbioru dozwolonego (permitted) oraz efektywnego (effective) wątku. Aby temu zapobiec, wątek może wywołać funkcję prctl() z opcją PR_SET_KEEPCAPS.
- Jeżeli efektywny UID został zmieniony z zera na inną wartość, to wszystkie uprawnienia są usuwane ze zbioru efektywnego wątku.
- Jeżeli efektywny UID został zmieniony z wartości różnej od zera na inną, to wszystkie uprawnienia ze zbioru dozwolonego (permitted) wątku są kopiowane do zbioru efektywnego.
- Jeżeli file system UID został zmieniony z zera na inną wartość, to następujące uprawnienia są usuwane ze zbioru efektywnego wątku: CAP_CHOWN, CAP_DAC_OVERRIDE, CAP_DAC_READ_SEARCH, CAP_FOWNER, CAP_FSETID, CAP_MAC_OVERRIDE. Jeżeli file system UID został zmieniony z wartości różnej od zera na zero, to wymienione uprawnienia są ustawiane w zbiorze efektywnym wątku, jeżeli istnieją w zbiorze dozwolonym.

Zbiory uprawnień wątku mogą być modyfikowane za pomocą wywołania systemowego capset() lub funkcji z biblioteki libcap. Interfejs libcap jest obecnie preferowaną metodą zarządzania uprawnieniami wątków. W systemach ze wsparciem dla mechanizmu capabilities ze strony wirtualnego systemu plików, wątek może modyfikować tylko własne uprawnienia.

Funkcje biblioteki libcap nie operują bezpośrednio na uprawnieniach wątku, ale na ich kopii roboczej (ang. working storage). Kopia robocza zawiera wszystkie trzy zbiory uprawnień wątku: efektywny, dozwolony, dziedziczny. Użytkownik biblioteki libcap posiada m.in. możliwości:

- utworzenia kopii roboczej uprawnień (na podstawie istniejących uprawnień wątku lub od podstaw),
- zarządzania uprawnieniami kopii roboczej,
- zastąpienia uprawnień wątku przez uprawnienia przechowywane w kopii roboczej.

Podczas modyfikacji uprawnień wątku należy kierować się następującymi zasadami:

- Zbiór efektywny (effective) musi być podzbiorem zbioru dozwolonego (permitted).
- Wątek nie może dodać uprawnień do zbioru dozwolonego (permitted).
- Jeżeli wątek nie posiada uprawnienia CAP_SETPCAP, do zbioru dziedzicznego (inheritable) mogą zostać dodane tylko uprawnienia ze zbioru dozwolonego (permitted).
- Począwszy od wersji 2.6.25 jądra systemu Linux, do zbioru dziedzicznego (inheritable) mogą zostać dodane tylko uprawnienia ze zbioru ograniczającego (bounding).

Operowanie uprawnieniami wątku odbywa się za pomocą funkcji cap_get_proc() oraz cap_set_proc(). Pierwsza z nich alokuje kopię roboczą aktualnych uprawnień wątku i zwraca wskaźnik do kopii roboczej (cap_t) lub NULL w przypadku błędu.

```
#include <sys/capability.h>

cap_t cap_get_proc(void);
```

Programista jest odpowiedzialny za zwolnienie pamięci zaalokowanej na rzecz kopii roboczej uprawnień. W tym celu należy wykorzystać funkcję cap_free():

```
#include <sys/capability.h>

int cap_free(void* obj_d);
```

Argumentem obj_d może być zmienna typu cap_t (wskaźnik do kopii roboczej uprawnień). W przypadku wystąpienia błędu, cap_free() zwróci wartość -1.

Funkcja cap_set_proc() zastępuje uprawnienia wątku (3 zbiory) uprawnieniami zawartymi w kopii roboczej (parametr cap_p):

```
#include <sys/capability.h>

cap_t cap_get_proc(cap_t cap_p);
```

Jeżeli którekolwiek z uprawnień kopii roboczej nie może zostać przypisane do wątku, wywołanie funkcji zakończy się niepowodzeniem i stan uprawnień wątku pozostanie niezmieniony.

Do tworzenia kopii roboczej uprawnień można wykorzystać funkcje cap_init() i cap_dup().

Funkcja cap_init() zwraca kopią roboczą, w której wszystkie flagi uprawnień są wyzerowane. Innymi słowy zwrócona kopia robocza zawiera puste zbiory uprawnień.

```
#include <sys/capability.h>

cap_t cap_init(void);
```

Funkcja cap_dup() zwraca duplikat kopi roboczej wskazywanej przez argument cap_p. Dla duplikatu alokowana jest pamięć, więc jest on całkowicie niezależny od kopii roboczej wskazywanej przez cap_p.

```
#include <sys/capability.h>

cap_t cap_dup(cap_t cap_p);
```

W przypadku wystąpienia błędu, funkcje cap_init() i cap_dup() zwracają wartość NULL. Programista jest odpowiedzialny za zwolnienie zaalokowanej pamięci (za pomocą cap_free()).

Zarządzanie uprawnieniami kopii roboczej odbywa się za pomocą funkcji cap_clear(), cap_clear_flag(), cap_get_flag() oraz cap_set_flag().

Funkcja cap_clear() zeruje wszystkie uprawnienia przechowywane w kopii roboczej wskazywanej przez argument cap_p.

```
#include <sys/capability.h>

int cap_clear(cap_t cap_p);
```

Funkcja cap_clear_flag() usuwa wszystkie uprawnienia ze zbioru określonego przez parametr flag.

```
#include <sys/capability.h>

int cap_clear_flag(cap_t cap_p, cap_flag_t flag);
```

Parametr flag może przyjmować wartości odpowiadające zbiorom uprawnień wątku: CAP_EFFECTIVE, CAP_INHERITABLE, CAP_PERMITTED.

Funkcja cap_get_flag() pobiera aktualną wartość uprawnienia z określonego zbioru uprawnień w kopii roboczej.

```
#include <sys/capability.h>

int cap_get_flag(cap_t cap_p, cap_value_t cap, cap_flag_t flag,
    cap_flag_value_t *value_p);
```

Parametr	Opis
`cap_p`	wskaźnik na kopię roboczą uprawnień
`cap`	identyfikuje uprawnienie, np.: `CAP_CHOWN`, którego stan chcemy określić
`flag`	identyfikuje zbiór uprawnień, np.: `CAP_EFFECTIVE`
`value_p`	wskaźnik na zmienną, która po wywołaniu funkcji będzie określać stan uprawnienia: `CAP_SET` (1) lub `CAP_CLEAR` (0)

Ostatnią z omawianych funkcji biblioteki libcap jest cap_set_flag(). Funkcja cap_set_flag() definiuje stan uprawnień z tablicy caps na wartość value.

```
#include <sys/capability.h>

int cap_set_flag(cap_t cap_p, cap_flag_t flag, int ncap,
    cap_value_t *caps, cap_flag_value_t value);
```

Parametr	Opis
cap_p	wskaźnik na kopię roboczą uprawnień
flag	identyfikuje zbiór uprawnień, np.: CAP_EFFECTIVE
ncap	określa liczbę uprawnień w tablicy wskazywanej przez caps
caps	wskaźnik na tablicę uprawnień, których stan zostanie zmodyfikowany przez funkcję
value	stan, na który zostaną zmodyfikowane uprawnienia w tablicy caps dla zbioru określonego przez parametr flag; możliwe wartości to: CAP_SET (1) lub CAP_CLEAR (0)

W przypadku powodzenia, funkcje: cap_clear(), cap_clear_flag(), cap_get_flag() i cap_set_flag() zwracają wartość zero. Wystąpienie błędu jest sygnalizowane przez zwrócenie wartości -1 i ustawienie zmiennej errno.

Typowy scenariusz dodania uprawnień do zbioru efektywnego wątku przedstawiony jest poniżej.

```
/* Wskaźnik (uchwyt) na kopię roboczą uprawnień wątku. */
cap_t caps;

/* Lista uprawnień, które zostaną dodane do zbioru efektywnego: */
cap_value_t cap_list[2];

/*
 * Pobranie aktualnych uprawnień wątku. Funkcja alokuje pamięć dla
 * kopii roboczej i zwraca wskaźnik do niej:
 */

caps = cap_get_proc();
if (caps == NULL) {
    /* Obsługa błędu. */
}

/* Określenie uprawnień dodawanych do zbioru efektywnego: */
cap_list[0] = CAP_FOWNER;
cap_list[1] = CAP_SETFCAP;

/*
 * Zdefiniowanie stanu uprawnień z tablicy „cap_list” dla
 * zbioru efektywnego:
 */
if (cap_set_flag(caps, CAP_EFFECTIVE, 2, cap_list, CAP_SET) == -1) {
    /* Obsługa błędu. */
}

/* Zastąpienie uprawnień wątku przez uprawnienia z kopii roboczej: */
```

```
if (cap_set_proc(caps) == -1) {
    /* Obsługa błędu. */
}

/* Zwolnienie pamięci zaalokowanej na rzecz kopii roboczej: */
if (cap_free(caps) == -1) {
    /* Obsługa błędu. */
}
```

Dla zwykłych użytkowników systemu Linux, rozwiązaniem stosowanym w zastępstwie capabilities jest ustawienie bitu SUID dla pliku wykonywalnego. Przykładowo, program ping, który tworzy gniazdo surowe, do poprawnego funkcjonowania wymaga uprawnienia CAP_NET_RAW. Obecność bitu SUID powoduje uruchomienie programu z efektywnym UID roota, co gwarantuje posiadanie wszystkich uprawnień, ale nie jest w pełni bezpieczne.

Capabilities związane z plikami wykonywalnymi pozwalają nadać programom minimalne wymagane uprawnienia. Proszę zwrócić uwagę, że stosowanie tego rozwiązania nie wymaga wprowadzenia zmian w kodzie źródłowym programu:

```
$ ls -l /bin/ping
-rwsr-xr-x root root 33168 2007-12-10 20:03 /bin/ping

$ sudo chmod u-s /bin/ping

$ ls -l /bin/ping
-rwxr-xr-x root root 33168 2007-12-10 20:03 /bin/ping

$ ping mars.iti.pk.edu.pl
ping: icmp open socket: Operation not permitted

$ sudo setcap cap_net_raw=ep /bin/bing

$ getcap /bin/ping
/bin/ping = cap_net_raw+ep

$ ping mars.iti.pk.edu.pl
PING mars.iti.pk.edu.pl (149.156.146.210) 56(84) bytes of data.
64 bytes from mars.iti.pk.edu.pl (149.156.146.210): icmp_seq=1 ...
```

Polecenie:

```
$ sudo setcap cap_net_raw=ep /bin/bing
```

powoduje dodanie uprawnienia CAP_NET_RAW do zbioru dozwolonego (permitted) oraz ustawienie bitu efektywnego. Ustawienie bitu sprawia, że podczas uruchomienia programu ping, uprawnienia dozwolone (w tym przypadku CAP_NET_RAW) są kopiowane do zbioru uprawnień efektywnych wątku.

Jeżeli aplikacja potrafi zarządzać uprawnieniami wątku (capability aware application), np. za pomocą biblioteki libcap, to plikowi wykonywalnemu można przyznać tylko uprawnienia dozwolone (permitted):

```
$ sudo setcap cap_net_raw=p /bin/bing
```

W takiej sytuacji wątek może modyfikować uprawnienia efektywne dla poszczególnych fragmentów programu, tzn. przyznawać uprawnienia tylko fragmentom, które tego wymagają.

Poniżej przedstawiony jest schemat aplikacji dynamicznie zarządzającej uprawnieniami[8]:

```
/* Tablica uprawnień (posiada tylko 1 element): */
const cap_value_t cap_vector[1] = { CAP_NET_RAW };

/*
 * Utworzenie kopii roboczej „privilege_dropped”, która posiada
 * wszystkie uprawnienia wyzerowane:
 */
cap_t privilege_dropped = cap_init();

/* Utworzenie duplikatu kopii roboczej „privilege_dropped”: */
cap_t privilege_off = cap_dup(privilege_dropped);

/*
 * Kopia „privilege_off” posiada uprawnienie CAP_NET_RAW tylko
 * w zbiorze dozwolonym. Zbiór efektywny i dziedziczny są puste:
 */
cap_set_flag(privilege_off, CAP_PERMITTED, 1, cap_vector, CAP_SET);

/* Utworzenie duplikatu kopii roboczej „privilege_off”: */
cap_t privilege_on = cap_dup(privilege_off);

/*
 * Kopia „privilege_on” posiada uprawnienie CAP_NET_RAW w zbiorze
 * dozwolonym oraz w zbiorze efektywnym:
 */
cap_set_flag(privilege_on, CAP_EFFECTIVE, 1, cap_vector, CAP_SET);

/* Przypisanie wątkowi uprawnień z kopii „privilege_on”: */
cap_set_proc(privilege_on);

/*
 * (...)
 * Wykonanie operacji uprzywilejowanej (wymagającej CAP_NET_RAW). */
 */

/*
```

```
 * Przypisanie wątkowi uprawnień z kopii „privilege_off”.
 * Wykonanie operacji wymagającej CAP_NET_RAW nie będzie możliwe,
 * ponieważ CAP_NET_RAW zostanie usunięte ze zbioru efektywnego:
 */
cap_set_proc(privilege_off);

/*
 * W tym momencie możliwe jest przywrócenie uprawnienia CAP_NET_RAW
 * z kopii „privilege_on”, ponieważ wątek posiada uprawnie CAP_NET_RAW
 * w zbiorze dozwolonym.
 */

/*
 * Porzucenie wszystkich uprawnień w zbiorze efektywnym i dozwolonym.
 * Zdobycie uprawnień przez wątek po wywołaniu funkcji będzie
 * niemożliwe:
 */
cap_set_proc(privilege_dropped);

/* Zwolnienie zaalokowanej pamieci: */
cap_free(privilege_on);
cap_free(privilege_off);
cap_free(privilege_dropped);
```

1.4. Cel laboratorium

Celem laboratorium jest zapoznanie się z tradycyjnym modelem uprawnień funkcjonującym w systemach UNIX oraz z modelem, który opiera się na tzw. zdolnościach (capabilities). Podczas realizacji tego laboratorium zapoznasz się z:

- identyfikatorami procesu,
- zastosowaniem bitu SUID,
- demonami,
- zarządzaniem uprawnieniami wątków i plików wykonywalnych,
- podstawowymi funkcjami biblioteki libcap.

2. Przebieg laboratorium

Druga część instrukcji zawiera zadania do praktycznej realizacji, które demonstrują zastosowanie technik z omawianego zagadnienia.

2.1. Zadanie 1. Bit SUID

Zadanie polega na analizie kodu przykładowego programu. Program tworzy gniazdo nasłuchujące dla protokołu TCP i oczekuje na nadejście połączenia. Argumentem wywołania programu jest numer portu, który zostanie powiązany z gniazdem nasłuchującym. Nawiązanie połączenia przez klienta TCP powoduje zakończenie programu.

W celu uruchomienia programu należy wykonać następujące czynności:

1. Przejść do katalogu ze źródłami (rozpakować je w razie konieczności).
2. Skompilować program źródłowy do postaci binarnej:

   ```
   $ gcc -o suidserver suidserver.c
   ```

3. Uruchomić program z konta zwykłego użytkownika podając numer portu mniejszy od 1024:

   ```
   $ ./suidserver <numer portu>
   ```

4. Zmienić właściciela pliku na użytkownika root i ustawić bit SUID dla pliku wykonywalnego:

   ```
   $ sudo chown root: suidserver
   $ sudo chmod u+s suidserver
   ```

5. Ponownie uruchomić program z konta zwykłego użytkownika podając numer portu mniejszy od 1024:

   ```
   $ ./suidserver <numer portu>
   ```

6. W oddzielnym terminalu wydać polecenie:

   ```
   $ ps -C suidserver -o comm,pid,uid,euid,suid,fsuid
   ```

7. Jakim rezultatem zakończy się uruchomienie programu z konta roota?

2.2. Zadanie 2. Uruchamianie programu z konta roota

Celem zadania jest modyfikacja poprzedniego programu. Program ma być uruchamiany z konta roota i po wykonaniu czynności wymagających specjalnych uprawnień, powinien zmienić identyfikatory: UID, EUID, saved set-UID na identyfikator zwykłego użytkownika. Identyfikator użytkownika ma być podawany w argumencie wywołania programu.

W celu uruchomienia programu należy wykonać następujące czynności:

1. Skompilować program źródłowy do postaci binarnej:

   ```
   $ gcc -o rootserver rootserver.c
   ```

2. Uruchomić serwer z konta roota podając numer portu mniejszy od 1024 oraz identyfikator użytkownika, w kontekście którego będzie wykonywał się program:

   ```
   $ ./rootserwer <numer portu> <identyfikator użytkownika>
   ```

   Identyfikatory użytkowników można uzyskać wydając polecenie:

   ```
   $ cat /etc/passwd
   ```

3. W oddzielnym terminalu proszę wydać polecenie:

```
$ ps -C rootserver -o comm,pid,uid,euid,suid,fsuid
```

2.3. Zadanie 3. Przekształcanie procesu w demona

Zadanie polega na analizie kodu przykładowego programu. Program przekształca proces serwera TCP w demona. Proszę zwrócić szczególną uwagę na funkcję daemonize(), która realizuje etapy omówione w podrozdziale 1.3.2.

W celu uruchomienia programu należy wykonać następujące czynności:

1. Przejść do katalogu ze źródłami.
2. Skompilować program źródłowy do postaci binarnej:

```
$ gcc -o daemon daemon.c
```

3. Uruchomić serwer podając numer portu większy od 1024:

```
$ ./daemon <numer portu>
```

4. Wydać polecenie:

```
$ ps -C daemon -o comm,uid,euid,suid,fsuid,tty,pid,pgrp,session
```

5. Jaką zależność wykazują: PID, identyfikator grupy procesów oraz identyfikator sesji?
6. Proszę nawiązać połączenie z demonem za pomocą programu nc:

```
$ nc localhost <port demona>
```

7. Nawiązanie połączenia powinno zakończyć działanie demona. Proszę zweryfikować to za pomocą polecenia ps (punkt 4).

2.4. Zadanie 4. Uprawnienia plików wykonywalnych

Proszę zmodyfikować program z zadania 1 w taki sposób, aby możliwe było uruchomienie serwera TCP na porcie uprzywilejowanym, tj. o numerze mniejszym od 1024, z konta normalnego użytkownika. Modyfikacja polega na usunięciu wywołań funkcji odpowiedzialnych za zarządzanie identyfikatorami użytkownika, a nie na dodaniu nowej funkcjonalności. Aby możliwe było uruchomienie serwera na porcie uprzywilejowanym, należy nadać plikowi wykonywalnemu odpowiednie uprawnienia (CAP_NET_BIND_SERVICE).

W celu uruchomienia programu należy (z konta zwykłego użytkownika) wykonać następujące czynności:

1. Skompilować program źródłowy do postaci binarnej:

```
$ gcc -o fserver fserver.c
```

2. Nadać plikowi wykonywalnemu uprawnienie CAP_NET_BIND_SERVICE:

   ```
   $ sudo setcap cap_net_bind_service=ep fserver
   ```

3. 3. Za pomocą programu getcap zweryfikować uprawnienia pliku:

   ```
   $ getcap fserver
   ```

4. Uruchomić serwer podając uprzywilejowany numer portu:

   ```
   $ ./fserver <numer portu>
   ```

5. Proszę w oddzielnym terminalu wydać polecenie:

   ```
   $ ps -C fserver -o comm,pid,uid,euid,suid,fsuid
   ```

2.5. Zadanie 5. Zarządzanie uprawnieniami wątku

Proszę zmodyfikować program z zadania 4 w taki sposób, aby uruchomienie serwera TCP na porcie uprzywilejowanym wymagało uprawnienia CAP_NET_BIND_SERVICE tylko w zbiorze dozwolonym pliku wykonywalnego, a nie w zbiorze efektywnym. Program ma być uruchamiany z konta zwykłego użytkownika i po powiązaniu numeru portu z gniazdem, powinien zrzucić wszystkie uprawnienia. Do zmiany uprawnień wątku proszę wykorzystać bibliotekę libcap.

W celu uruchomienia programu należy (z konta zwykłego użytkownika) wykonać następujące czynności:

1. Skompilować program źródłowy do postaci binarnej:

   ```
   $ gcc -o capserver -lcap capserver.c
   ```

2. Nadać plikowi wykonywalnemu uprawnienie CAP_NET_BIND_SERVICE:

   ```
   $ sudo setcap cap_net_bind_service=p capserver
   ```

3. Za pomocą programu getcap zweryfikować uprawnienia pliku:

   ```
   $ getcap capserver
   ```

4. Uruchomić serwer podając uprzywilejowany numer portu:

   ```
   $ ./capserver <numer portu>
   ```

5. W oddzielnym terminalu wydać polecenie:

   ```
   $ ps -C capserver -o comm,pid
   ```

6. Zweryfikować uprawnienia z jakimi wykonuje się serwer. W tym celu proszę wydać polecenie:

   ```
   $ cat /proc/<PID>/status
   ```

   gdzie <PID> jest identyfikatorem uruchomionego procesu (krok 5), a następnie odszukać etykiety: CapInh, CapPrm, CapEff oraz CapBnd.

3. Opracowanie i sprawozdanie

Realizacja laboratorium pt. „Uprawnienia” polega na wykonaniu wszystkich zadań programistycznych podanych w drugiej części tej instrukcji. Wynikiem wykonania powinno być sprawozdanie w formie wydruku papierowego dostarczonego na kolejne zajęcia licząc od daty laboratorium, kiedy zadania zostały zadane.

Sprawozdanie powinno zawierać:

- opis metodyki realizacji zadań (system operacyjny, język programowania, biblioteki, itp.),
- algorytmy wykorzystane w zadaniach (zwłaszcza, jeśli zastosowane zostały rozwiązania nietypowe),
- opisy napisanych programów wraz z opcjami,
- trudniejsze kawałki kodu, opisane tak, jak w niniejszej instrukcji,
- uwagi oceniające ćwiczenie: trudne/łatwe, nie/realizowalne, nie/wymagające wcześniejszej znajomości zagadnień (wymienić jakich),
- wskazówki dotyczące ewentualnej poprawy instrukcji celem lepszego zrozumienia sensu oraz treści zadań.