

# WOJSKOWA AKADEMIA TECHNICZNA

im. Jarosława Dąbrowskiego

---

## WYDZIAŁ CYBERNETYKI



## PRACA DYPLOMOWA

Temat pracy: **UKRYWANIE PROCESÓW W SYSTEMIE  
GNU/LINUX PRZY WYKORZYSTANIU  
MECHANIZMÓW DOSTĘPNYCH  
W PRZESTRZENI UŻYTKOWNIKA**

**KRYPTOLOGIA I CYBERBEZPIECZEŃSTWO**

.....  
(kierunek studiów)

**CYBEROBRONA**

.....  
(specjalność)

Dyplomant:

**Karol BARANOWSKI**

Promotor pracy:

**dr inż. Łukasz STRZELECKI**

---

Warszawa 2020

**OŚWIADCZENIE**

*„Wyrażam zgodę na udostępnianie mojej pracy w czytelni  
Archiwum WAT”.*

Dnia .....

.....  
(podpis)

*Pracę przyjąłem*

*promotor pracy*

*dr inż. Łukasz Strzelecki*

**WOJSKOWA AKADEMIA TECHNICZNA  
WYDZIAŁ CYBERNETYKI  
INSTYTUT TELEINFORMATYKI I AUTOMATYKI**

=====

"AKCEPTUJĘ"  
DZIEKAN WYDZIAŁU CYBERNETYKI

  
dr hab. inż. Kazimierz WORWA

Warszawa, dnia 06.05 2019 r.

**ZADANIE  
do pracy dyplomowej**

Wydane studentowi: **KAROL BARANOWSKI**  
(stacjonarne studia pierwszego stopnia)

- I. Temat pracy:  
**UKRYWANIE PROCESÓW W SYSTEMIE GNU/LINUX PRZY WYKORZYSTANIU  
MECHANIZMÓW DOSTĘPNYCH W PRZESTRZENI UŻYTKOWNIKA**
- II. Treść zadania:
1. Analiza mechanizmów umożliwiających ukrywanie procesów przy wykorzystaniu mechanizmów dostępnych w przestrzeni użytkownika.
  2. Koncepcja działania programu utrudniającego jego wykrycie.
  3. Projekt i implementacja programu demonstracyjnego.
  4. Testy akceptacyjne.
  5. Wnioski.

III. W rezultacie wykonania pracy należy dodatkowo przedstawić:


.....  
.....

IV. Opiekun: .....

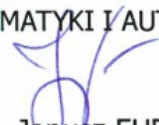
V. Termin zdania przez studenta ukończonej pracy: **31.01.2020 r.**

VI. Data wydania zadania: **30.04.2019 r.**

PROMOTOR  
PRACY DYPLOMOWEJ

  
dr inż. Łukasz STRZELECKI  
(tytuł naukowy, imię i nazwisko)

DYREKTOR INSTYTUTU  
TELEINFORMATYKI I AUTOMATYKI

  
dr inż. Janusz FURTAK  
(tytuł naukowy, imię i nazwisko)

Zadanie otrzymałem dnia 30 IV 2019 .....



Karol BARANOWSKI  
(podpis studenta)

## SPIS TREŚCI

<b>WSTĘP.....</b>	<b>6</b>
<b>ROZDZIAŁ I. ZADANIE PROJEKTOWE.....</b>	<b>7</b>
<b>ROZDZIAŁ II. PROCESY W PRZESTRZENI UŻYTKOWNIKA I W PRZESTRZENI JĄDRA.....</b>	<b>9</b>
<b>ROZDZIAŁ III. MECHANIZMY UŻYWANE DO UKRYCIA PROCESÓW W TRYBIE UŻYTKOWNIKA.....</b>	<b>12</b>
3.1. Zmiana nazwy procesu.....	12
3.2. Zmiana sesji i identyfikatora procesu.....	15
<b>ROZDZIAŁ IV. CEL I ZAŁOŻENIA PROGRAMU DEMONSTRACYJNEGO.....</b>	<b>17</b>
<b>ROZDZIAŁ V. PROJEKT PROGRAMU DEMONSTRACYJNEGO.....</b>	<b>19</b>
<b>ROZDZIAŁ VI. IMPLEMENTACJA PROGRAMU DEMONSTRACYJNEGO.....</b>	<b>22</b>
6.1. Losowanie i ustawianie nowej nazwy procesu.....	22
6.2. <i>Demonizacja</i> procesu.....	25
6.3. Ustawienie parametru czasu dla zmiany sesji.....	28
<b>ROZDZIAŁ VII. TESTY AKCEPTACYJNE.....</b>	<b>32</b>
7.1. Test poprawności działania zmiany nazwy procesu.....	32
7.2. Test poprawności działania zmiany sesji i identyfikatora procesu.....	33
7.3. Test poprawności działania pełnego programu demonstracyjnego.....	34
7.4. Test wykrywalności przy użyciu narzędzia Unhide.....	36
<b>PODSUMOWANIE.....</b>	<b>39</b>
<b>BIBLIOGRAFIA.....</b>	<b>40</b>
<b>WYKAZ KODÓW ŹRÓDŁOWYCH.....</b>	<b>41</b>

## WSTĘP

Tematem pracy dyplomowej jest analiza zagadnień związanych z ukrywaniem procesów w systemach GNU/Linux przy wykorzystaniu mechanizmów dostępnych w przestrzeni użytkownika.

Złośliwe oprogramowanie (często również szpiegujące) ukrywa swoje procesy na rozmaite sposoby, aby utrudnić, bądź wręcz uniemożliwić wykrycie ich w systemie. Jednak ukrywanie procesów nie jest używane tylko w tego typu rozwiązaniach. Często OS GNU/Linux używany jest jako system sieciowy, do którego podłącza się wielu użytkowników. Można ukrywać procesy po to, aby uniemożliwić innym użytkownikom pozyskanie informacji o działających procesach innych osób.

W ramach pracy inżynierskiej poddano analizie mechanizmy umożliwiające ukrycie procesów przy wykorzystaniu narzędzi dostępnych w przestrzeni użytkownika. W celach poglądowych opracowano także program, który wykorzystuje wybrane mechanizmy rozważanego typu do utrudnienia wykrycia jego obecności w systemie operacyjnym – jego implementacja została (fragmentarycznie) omówiona w dalszej części pracy.

## ROZDZIAŁ I.

### ZADANIE PROJEKTOWE

W ramach pracy „Ukrywanie procesów w systemie GNU/Linux przy wykorzystaniu mechanizmów dostępnych w przestrzeni użytkownika” należało zrealizować m.in. poniżej wymienione zadania.

1. Analiza mechanizmów umożliwiających ukrywanie procesów przy wykorzystaniu mechanizmów dostępnych w przestrzeni użytkownika.
2. Koncepcja działania programu utrudniającego jego wykrycie.
3. Projekt i implementacja programu demonstracyjnego.
4. Testy akceptacyjne.
5. Wnioski.

Wszystkie zadania zostały pomyślnie zrealizowane. Kolejne rozdziały dokładnie przedstawiają problematykę każdego z nich oraz sposób ich rozwiązania.

Rozdział drugi „Procesy w przestrzeni użytkownika i w przestrzeni jądra” zawiera informacje związane z działaniem procesów w systemach GNU/Linux, które były podstawą do opracowania większości koncepcji rozwiązań omówionych w niniejszej pracy.

Rozdział trzeci („Mechanizmy używane do ukrycia procesów w trybie użytkownika”) zawiera opis najczęściej spotykanych sposobów ukrywania procesów (np. w narzędziach szpiegujących bądź typu *Remote Administration Tool* - RAT).

Rozdział czwarty został poświęcony przedstawieniu koncepcji rozwiązania realizującego główny cel pracy, a także przyjęte założenia.

Rozdział piąty zawiera słowny schemat działania oraz diagram przepływu danych zaproponowanego rozwiązania.

Rozdział szósty opisuje najważniejsze funkcje zaimplementowanego na potrzeby pracy oprogramowania, którego budowa opiera się na wcześniej przeanalizowanych i zaprojektowanych mechanizmach.

Rozdział siódmy „Testy akceptacyjne” prezentuje wyniki przeprowadzonych testów akceptacyjnych, których celem było praktyczne potwierdzenie poprawności realizacji postawionego zadania (czyli opracowanie oprogramowania ukrywającego fakt jego występowania w zadanym systemie operacyjnym).



## ROZDZIAŁ II.

### PROCESY W PRZESTRZENI UŻYTKOWNIKA

#### I W PRZESTRZENI JĄDRA

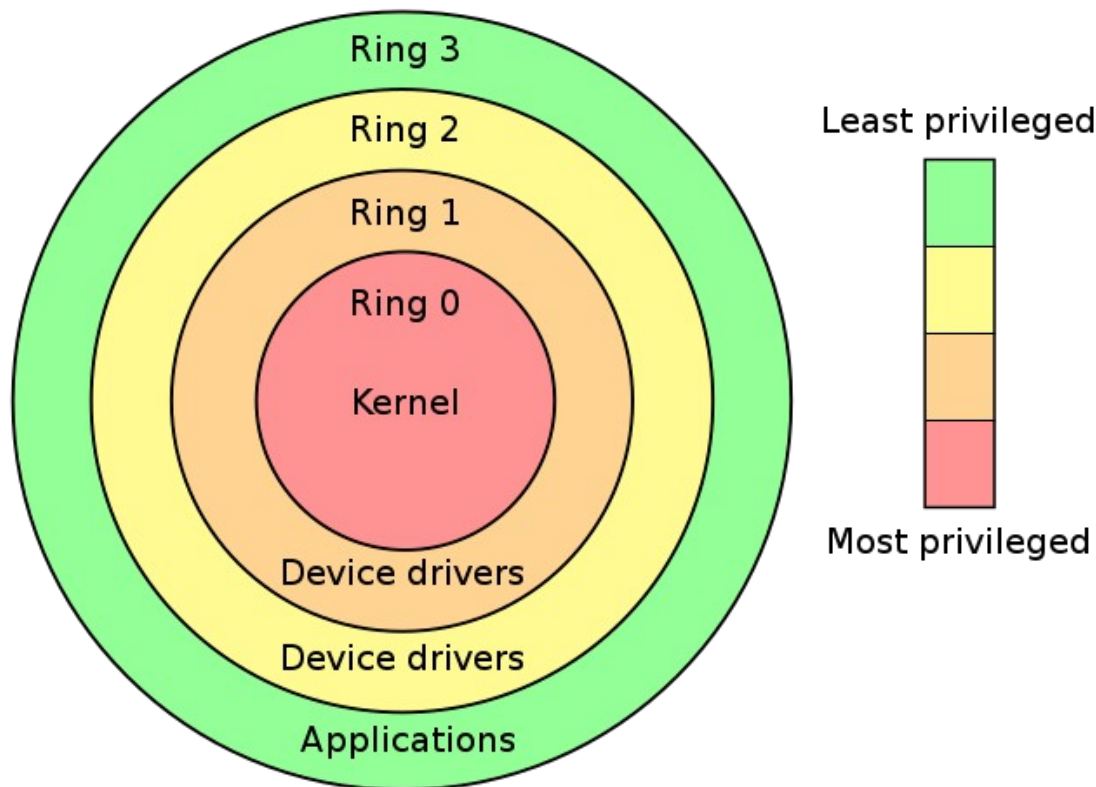
Proces w systemach GNU/Linux to w uproszczeniu aktualnie wykonywany przez procesor ciąg poleceń. Każdy proces charakteryzuje się pewnymi atrybutami (m.in. przestrzeń adresowa, deskryptory pliku, dane, zależności systemowe itp.).

Pamięć operacyjna w GNU/Linux jest podzielona między dwie odseparowane od siebie przestrzenie – zostały one omówione poniżej:

1. Przestrzeń użytkownika (tzw. *user mode*) lub tryb użytkownika – zakres przestrzeni adresowej, w której działają procesy użytkowników. Rolą jądra jest takie zarządzanie procesami w tej przestrzeni, aby nie odnosiły się do nienależących do nich przestrzeni adresowych (np. innych aplikacji oraz procesów jądra).
2. Przestrzeń jądra (*kernel mode*) lub tryb jądra - miejsce przechowywania i wykonywania kodu jądra.

Procesy działające w przestrzeni użytkownika mają dostęp tylko do ograniczonej części pamięci podczas, gdy procesy jądra mają dostęp pełny. Procesy przestrzeni użytkownika mogą uzyskać dostęp tylko do małej części jądra za pośrednictwem interfejsu udostępnianego przez jądro - wywołań systemowych. Jeśli proces wykonuje wywołanie systemowe, do jądra wysyłane jest przerwanie programowe, które wymusza wykonanie powiązanej z nim procedury obsługi przerwania (po zakończeniu obsługi przerwania następuje powrót do poprzednio realizowanych zadań).

Kod przestrzeni jądra może być ładowany w trybie użytkownika, który (w typowej architekturze x86) nazywa się kodem wykonywanym pod pierścieniem 0. Zazwyczaj w architekturze x86 są 4 pierścienie ochrony, co poglądowo przedstawiono na rysunku 2.1.



Rysunek 2.1 Schemat pierścieni uprawnień dla architektury x86

Jądro systemu GNU/Linux używa tylko pierścienia 0 dla trybu jądra i pierścienia 3 dla trybu użytkownika. Układ procesora jest tak fizycznie zbudowany, że pierścień 0 jest najbardziej uprzywilejowany i ma dostęp do wszystkich instrukcji urządzenia. Pierścień 3 jest najmniej uprzywilejowany i aplikacje w jego ramach działające mają dostęp tylko do podzbioru instrukcji procesora. Na przykład aplikacja taka, jak przeglądarka internetowa nie może używać instrukcji *lgdt* (języka *assembly x86*) do ładowania globalnej tabeli deskryptorów lub *hlt* w celu zatrzymania procesora. Inne aspekty dotyczące działania w ramach pierścienia 3 zostały wymienione poniżej.

1. Nie można zmienić własnego pierścienia. W przeciwnym razie można byłoby ustawić dowolną wartość pierścienia, co w praktyce uniemożliwiłoby spełnianie przez ten mechanizm swojego zadania. Innymi

słowy, nie można modyfikować bieżącego deskryptora segmentu, który określa bieżący pierścień.

2. Nie można modyfikować tabel stron, czyli nie można modyfikować rejestru *CR3*. Także samo stronicowanie uniemożliwia taką modyfikację. Zapobiega to przed dostępem jednego procesu do pamięci innych procesów.
3. Nie można rejestrować obsługi przerwań.
4. Nie można bezpośrednio wykonywać instrukcji *IO*, takich jak *in* i *out*, a tym samym mieć nieograniczony dostęp do sprzętu. W przeciwnym razie np. uprawnienia do plików byłyby bezużyteczne, ponieważ każdy proces mógłby bezpośrednio odczytywać dane z dysku.

Podsumowując, tryb użytkownika jest dużo bardziej ograniczony pod względem możliwości (dostępnych funkcji) oprogramowania działającego w jego ramach, niż tryb jądra. Z tego powodu programy, które ukrywają procesy pisze się głównie dla trybu jądra, którego możliwości są znacznie większe. Nie mniej, mimo iż ukrywanie procesów w trybie użytkownika jest mniej popularne, wciąż jest możliwe i stanowi przedmiot niniejszej pracy.

## ROZDZIAŁ III.

### MECHANIZMY UŻYWANE DO UKRYCIA PROCESÓW W TRYBIE UŻYTKOWNIKA

Do podstawowych technik ukrycia procesu w trybie użytkownika należą:

1. zmiana nazwy procesu,
2. zmiana sesji procesu (tzw. *demonizacja* – w połączeniu z zamknięciem deskryptorów dla standardowego wejścia, wyjścia i wyjścia diagnostycznego),
3. zmiana identyfikatora procesu.

Pojedynczo techniki te nie pozwalają skutecznie ukryć procesu. Jednakże połączenie wszystkich trzech metod w jednym programie, niesie możliwość skutecznego zakamuflowania procesu. Poniżej opisano każdą technikę.

#### 3.1. Zmiana nazwy procesu

W większości dystrybucji systemów GNU/Linux informacje o procesach, w tym ich nazwy umieszczone są w wirtualnym systemie plików *procfs*, który jest standardowo zamontowany w folderze */proc* (administrator może zmienić lokalizację montowania tego systemu plików). Folder ten zawiera pliki z informacjami systemowymi takimi jak konfiguracja sprzętu, zamontowane urządzenia, pamięć systemu itp. Zawartość każdego pliku generowana jest przez jądro systemu podczas próby jego odczytu, co sprawia, że przekazywane w ten sposób informacje są zawsze aktualne. Wiele systemowych narzędzi odwołuje się do plików w tym folderze, np. uruchomienie *lsmod* to synonim dla wywołania polecenia *cat /proc/modules*. Poprzez modyfikację plików, które się tu znajdują można nawet zmieniać parametry działania jądra systemu. Poza tym wszystkie pliki umieszczone w */proc* mają rozmiar 0 (z wyjątkiem *kcore*, *mtrr* i *self*), ponieważ ze względu na generowanie ich zawartości podczas próby ich odczytania nie jest

możliwe bieżące wskazywanie ich rozmiaru. Na rysunku 3.1 przedstawiono przykładową zawartość folderu `/proc`. Każdy folder z numerem w nazwie to katalog zawierający informacje o procesie z identyfikatorem procesu odpowiadającym numerowi z tej nazwy.

```
$ ls -la /proc
total 4
dr-xr-xr-x. 234 root    root          0 Dec 20 14:49 .
dr-xr-xr-x.  18 root    root          4096 Nov 11 10:46 ..
dr-xr-xr-x.   9 root    root           0 Dec 20 14:49 1
dr-xr-xr-x.   9 root    root           0 Dec 20 14:49 10
dr-xr-xr-x.   9 root    root           0 Dec 20 14:49 11
dr-xr-xr-x.   9 dnsmasq dnsmasq       0 Dec 20 14:49 1150
dr-xr-xr-x.   9 root    root           0 Dec 20 14:50 1151
dr-xr-xr-x.   9 root    root           0 Dec 20 14:49 12
dr-xr-xr-x.   9 root    root           0 Dec 20 14:50 1220
dr-xr-xr-x.   9 root    root           0 Dec 20 14:50 1242
dr-xr-xr-x.   9 root    root           0 Dec 20 14:49 13
dr-xr-xr-x.   9 root    root           0 Dec 20 14:49 132
dr-xr-xr-x.   9 root    root           0 Dec 20 14:49 133
dr-xr-xr-x.   9 root    root           0 Dec 20 14:49 134
dr-xr-xr-x.   9 root    root           0 Dec 20 14:49 135
dr-xr-xr-x.   9 root    root           0 Dec 20 14:49 136
dr-xr-xr-x.   9 root    root           0 Dec 20 14:49 137
dr-xr-xr-x.   9 root    root           0 Dec 20 14:49 138
dr-xr-xr-x.   9 root    root           0 Dec 20 14:49 139
dr-xr-xr-x.   9 colord  colord        0 Dec 20 14:50 1393
dr-xr-xr-x.   9 root    root           0 Dec 20 14:49 14
dr-xr-xr-x.   9 root    root           0 Dec 20 14:49 140
dr-xr-xr-x.   9 root    root           0 Dec 20 14:49 141
```

Rysunek 3.1 Przykładowy widok zawartości folderu `/proc`

Poniżej, na rysunku 3.2 przedstawiono przykładową zawartość folderu odpowiadającemu jednemu z procesów.

```
arch_status  comm          fdinfo      maps        numa_maps   root         stack       timerslack_ns
attr         coredump_filter  gid_map     mem         oom_adj     sched        stat        uid_map
autogroup   cpuset           io          mountinfo  oom_score   schedstat    statm       wchan
auxv        cwd             latency     mounts     oom_score_adj sessionid     status
cgroup      environ         limits     mountstats pagemap     setgroups    syscall
clear_refs  exe            loginuid   net        personality smaps        task
cmdline     fd             map_files  ns         projid_map  smaps_rollup timers
```

Rysunek 3.2 Przykładowa zawartość folderu procesu

Występuje tu wiele plików zawierających informacje o atrybutach procesu, w tym opisane poniżej.

1. *cmdline* – plik tylko do odczytu zawierający pełną komendę z wiersza poleceń dla danego procesu, w tym jego nazwę, która jest pierwszym argumentem polecenia.
2. *status* – Plik zawierający informacje z plików *stat* i *statm* w formacie czytelnym dla człowieka. Przechowywane jest w nim wiele informacji o parametrach procesu m.in. nazwa procesu, stan, identyfikator procesu, identyfikator procesu *rodzica*, identyfikator użytkownika, identyfikator grupy, czas rozpoczęcia i wiele innych. Nazwa procesu znajduje się w pierwszym wierszu tego pliku.

Wiedząc, że nazwy procesów przechowywane są we wcześniej omówionych plikach, większość programów pobiera je właśnie stąd. Na tej podstawie można napisać program, który będzie zmieniał nazwę w wymienionych zasobach na inną. Dla przykładu, w języku C do zmiany nazwy procesu w pliku *status* można użyć funkcji :

```
prctl(PR_SET_NAME, „nowa_nazwa”)
```

Uwzględniając, że w pliku *cmdline* nazwą jest pierwszy argument z uruchomienia programu poprzez wiersz poleceń, czyli `argv[0]`, można go zastąpić innym napisem np. z pomocą funkcji `strncpy()`. Należy pamiętać, że nowa nazwa nie może być dłuższa od nazwy początkowej, ponieważ długość pierwszej nazwy jest liczona i tylko tyle miejsca w pamięci jest na nią zarezerwowane. Z tego powodu warto wybrać dla pierwotnego programu długą nazwę, co zarezerwuje odpowiednią ilość pamięci, aby zawsze móc ustawić nazwę o długości maksymalnej odpowiadającej długości nazwy początkowej (którą w przypadku nazwy krótszej zawsze można uzupełnić znakami 0x00). Na końcu nowej nazwy należy dodać znak sterujący NULL („\0”) informujący o końcu napisu.

Na podstawie powyższych informacji można przyjąć, że jest możliwe napisanie programu działającego w trybie użytkownika, który w pętli będzie zmieniał swoją nazwę np. co określony, bądź losowy czas. Aby zakamuflować proces skutecznie, nazwa powinna wyglądać autentycznie. By to osiągnąć można

np. iterować po folderze */proc*, i na podstawie zamieszczonych tam danych stworzyć listę wszystkich procesów działających w systemie, a następnie wylosować jedną z nich i zastąpić aktualną nazwę procesu. Tego typu podejście zostało dokładniej opisane w rozdziale dotyczącym implementacji programu demonstracyjnego.

### 3.2. Zmiana sesji i identyfikatora procesu

W chwili wylogowywania się użytkownika z systemu, jądro musi zakończyć wszystkie uruchomione przez niego procesy (w przeciwnym razie użytkownicy zostawiliby aktywne procesy czekające na dane wejściowe, które nigdy nie mogłyby nadejść). Aby uprościć to zadanie, procesy są zorganizowane w sesję. Identyfikator sesji jest taki sam jak identyfikator procesu, który utworzył sesję za pomocą wywołania systemowego `setsid()` (proces ten zostaje liderem sesji). Wszyscy potomkowie tego procesu są wówczas członkami tej sesji chyba, że wyraźnie się z niej usuną za pomocą funkcji `setsid()`. Funkcja ta nie przyjmuje żadnych argumentów i zwraca nowy identyfikator sesji. Nowo powstała sesja nie jest powiązana z żadnym terminalem.

Aby skutecznie ukryć proces i zmienić jego sesję należy przeprowadzić proces tzw. *demonizacji*. *Demony* to w uproszczeniu usługi działające w tle systemu. Ich procesem macierzystym jest proces *init* (pierwszy uruchamiany proces po starcie systemu, po którym dziedziczą wszystkie inne). Dodatkowo nie korzystają one z terminala (nie używają standardowego wyjścia, wejścia i strumienia błędów). *Demonizacja* zwykle składa się z poniżej wymienionych czynności.

1. Uruchomienie procesu potomnego w tle i zakończenie procesu macierzystego. W ten sposób proces *init* stanie się automatycznie *rodzicem* „osieroconego” procesu, któremu zostanie przydzielony nowy identyfikator procesu.

2. Utworzenie nowej sesji. Proces zostanie liderem nowej sesji i grupy procesów i w ten sposób odłączy się od powiązanego z nim pierwotnie terminala.
3. Obsługa sygnałów. W zależności od potrzeb można napisać kod obsługujący i/lub ignorujący dany sygnał (poza sygnałami SIGKILL i SIGSTOP, ponieważ nie można zmienić ich obsługi).
4. Ponowne uruchomienie procesu *dziecka* w tle i zakończenie procesu macierzystego. Nowo powstały proces nie będzie teraz liderem sesji (identyfikator procesu będzie inny niż identyfikator sesji) i jako niesesyjnemu liderowi grupy, nigdy nie będzie można przywrócić mu terminala kontrolującego.
5. Zamknięcie wszystkich otwartych deskryptorów pliku, które mogły zostać odziedziczone po procesie macierzystym.

Dodatkowo w niektórych przypadkach stosowana jest również zmiana katalogu roboczego procesu oraz maski uprawnień. Zmianę katalogu stosuje się, aby upewnić się, że proces nie utrzymuje go w użyciu. Nieprzestrzeganie tego może spowodować, że np. nie będzie możliwe usunięcie katalogu (w zależności od systemu plików) bądź odmontowanie systemu plików. Zmiana maski umożliwia kontrolę uprawnień do nowo tworzonych przez proces plików. Nie wiadomo z jaką maską proces macierzysty zostanie uruchomiony, a procesy potomne domyślnie ją dziedziczą. Pożądaną sytuacją jest, aby proces mógł czytać, pisać i wykonywać utworzone przez siebie pliki.



## ROZDZIAŁ IV.

### CEL I ZAŁOŻENIA PROGRAMU DEMONSTRACYJNEGO

Celem rozwiązania demonstracyjnego było udowodnienie, że jest możliwe ukrycie procesu przy wykorzystaniu mechanizmów dostępnych w przestrzeni użytkownika. W realnych scenariuszach, programy ukrywające swoje procesy są często złośliwym oprogramowaniem. W tej sytuacji – w celu osadzenia tego typu oprogramowania na docelowym hoście - atakujący musiałby uzyskać dostęp do wybranej maszyny i samodzielnie uruchomić program. Innym rozwiązaniem mogłoby być wysłanie oferty programu pocztą elektroniczną i nakłonienie jej do włączenia przy wykorzystaniu technik z zakresu socjotechniki. Program korzystający z ukrywania procesu, mógłby również być używany w firmach, a jego celem byłoby monitorowanie działań pracowników.

Projekt i implementacja zostały przedstawione w dalszej części pracy, natomiast przyjęte założenia zostały przedstawione poniżej.

1. Program stanowi jedynie demonstrator możliwości wykorzystania technik ukrywania procesów w przestrzeni użytkownika i jako taki nie realizuje żadnych funkcji złośliwego lub szpiegowskiego oprogramowania oraz nie dodaje się po uruchomieniu do *autostartu*.
2. Program w całości zostanie napisany w języku C.
3. W celu kompilacji rozwiązania użyto kompilatora GCC w wersji 9.2.1 20190827.
4. Program powinien poprawnie działać pod kontrolą 64-bitowego systemu *Linux Fedora 31* (z jądrem w wersji 5.3.8-300.fc31.x86\_64).
5. Oprogramowanie powinno być uruchamiane przez użytkownika w terminalu. Może mieć on dowolne uprawnienia, jakkolwiek od nich zależy dostęp do poszczególnych informacji o procesach w */proc*. W większości konfiguracji systemów, standardowy użytkownik będzie mógł tylko przeczytać nazwy procesów, których jest właścicielem. Program miałby dostęp do informacji

o wszystkich procesach, gdyby uruchomiono go z konta *root*, ponieważ to konto widzi procesy wszystkich użytkowników w */proc* i może odczytać o nich każdą informację.

6. W przypadku wystąpienia błędów program powinien w kontrolowany sposób zakończyć swoje działanie.
7. Proces powinien uniemożliwiać predykcję kolejnych operacji przezeń wykonywanych (np. okres zmiany nazwy).
8. Podczas zmiany nazwy procesu – jako nowe możliwości – powinny być brane pod uwagę nazwy aktualnie działających w systemie procesów.

## ROZDZIAŁ V.

### PROJEKT PROGRAMU DEMONSTRACYJNEGO

W ramach zadania pracy inżynierskiej należało zaprojektować oraz zaimplementować demonstracyjny program utrudniający jego wykrycie z wykorzystaniem standardowych narzędzi systemowych (i przynajmniej niektórych specjalizowanych). Podczas działania, program będzie korzystał z wywołań systemowych, aby komunikować się z jądrem systemu m.in. w celach:

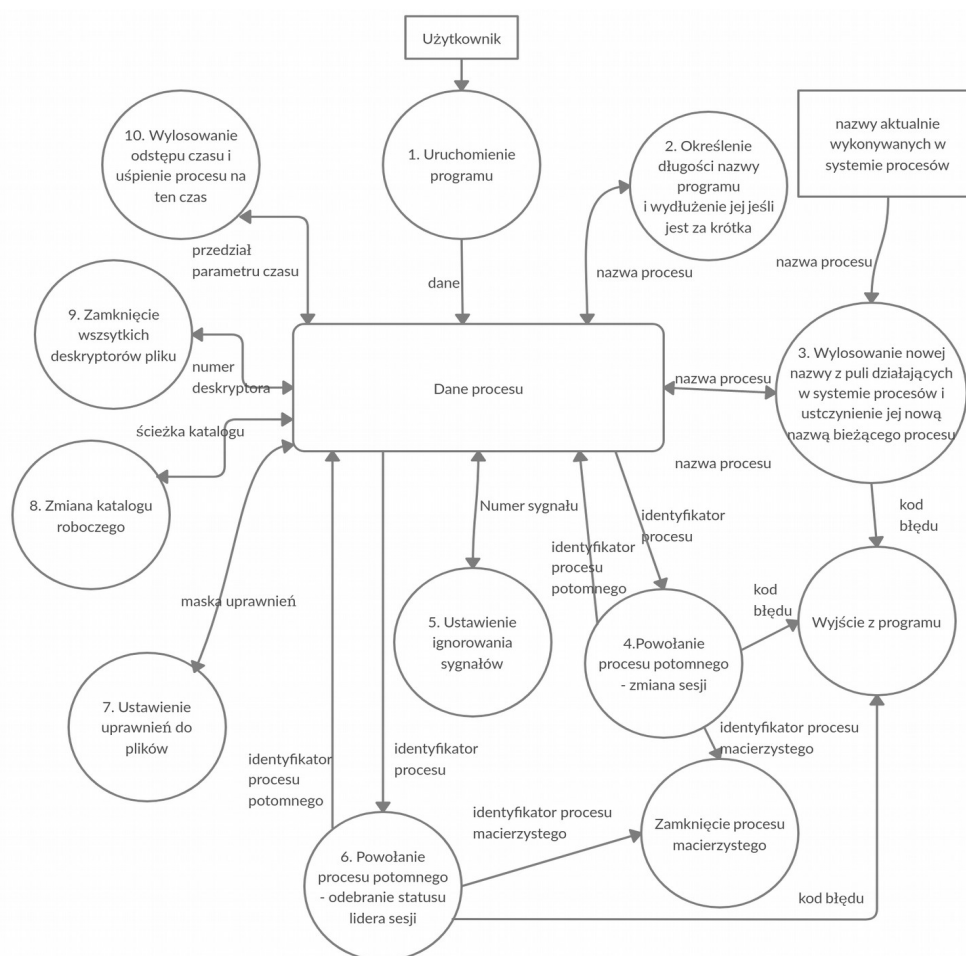
1. zarządzania pamięcią,
2. zapisu i odczytu danych (informacje o procesach),
3. otwierania i zamykania plików.

Zakładany sposób działania rozwiązania wypunktowano poniżej.

1. Program po uruchomieniu, określa długość swojej nazwy i rezerwuje adekwatną ilość pamięci na następne nazwy, które będą zastępować aktualną. Jeżeli pierwotna nazwa jest krótka (w zależności od ustawionych w programie parametrów), program uruchamia się sam jeszcze raz z dłuższą nazwą.
2. Sprawdzenie czy zamontowany jest system plików */proc*. Jeżeli nie, program jest kończony.
3. Określenie ilości procesów poprzez iterację po folderach zawierających informacje o procesach w */proc*.
4. Losowany jest jeden z procesów. Następuje pobranie jego nazwy z pliku */proc/<identyfikator procesu>/status*.
5. Sprawdzenie, czy długość pobranej nazwy jest równa lub krótsza długości nazwy początkowej. Jeśli nie, wróć do punktu 4.
6. Ustawienie pobranej nazwy jako nowej nazwy programu.
7. Powołanie procesu potomnego i zamknięcie procesu macierzystego – zmiana sesji.

8. Ignorowanie wszystkich sygnałów wysyłanych do procesu, oprócz SIGKILL oraz SIGSTOP, ponieważ nie można zmienić ich obsługi.
9. Ponowne powołanie procesu potomnego i zamknięcie procesu macierzystego – odebranie statusu lidera sesji procesów.
10. Ustawienie nowych praw do plików.
11. Zmiana katalogu roboczego.
12. Zamknięcie wszystkich otwartych deskryptorów plików.
13. Odczekanie losowego czasu.
14. Powrót do kroku 2.

Opisany powyżej schemat działania może zostać przedstawiony również z wykorzystaniem diagramu przepływu typu DFD<sup>1</sup> - został on przedstawiony na rysunku 5.1.



Rysunek 5.1 Diagram przepływu danych

<sup>1</sup> Arciuch A., Liderman K.: *Projektowanie systemów komputerowych*, Warszawa 2001, s. 166-173.

Odnosząc się do diagramu przedstawionego na rysunku 5.1 warto zwrócić uwagę na zmianę nazwy procesu oraz losowanie odstępu czasu pomiędzy kolejnymi zmianami sesji, z którymi mogą wiązać się poniżej omówione problemy.

1. Pierwotna nazwa powinna być jak najdłuższa, aby było można ją zastąpić nazwą dowolnego procesu działającego w systemie. Jeżeli program zostanie uruchomiony z krótką nazwą zawierającą np. 3 znaki – nie będzie mógł kopiować nazw wszystkich procesów. W takiej sytuacji program powinien móc uruchamiać się ponownie, tym razem przyjmując dłuższą nazwę.
2. Odpowiedni parametr czasu pomiędzy zmianami sesji musi spełniać wiele warunków. W celu nieprzewidywalnego ukrywania się, powinien być losowany z szerokiego przedziału. Wartość dolna tego przedziału nie może być zbyt mała, gdyż proces może nie zdążyć zrealizować swojego prawdziwego celu w systemie oraz jego identyfikator po zmianie sesji (którego nie można zmienić programowo na dowolny w trybie użytkownika), będzie bliski identyfikatorowi procesu macierzystego. Wartość górna tego przedziału nie może być za duża, ponieważ gdyby działanie procesu zostało wykryte przez użytkownika, to miałby on wystarczająco dużo czasu, aby znaleźć odpowiedzialny za to działanie proces i go zakończyć.

Propozycja rozwiązania powyżej opisanych komplikacji została przedstawiona w dalszej części pracy.

## ROZDZIAŁ VI.

### IMPLEMENTACJA PROGRAMU DEMONSTRACYJNEGO

Z pośród samodzielnie zaimplementowanych funkcji, których używa program, z całą pewnością najważniejsze są:

1. losowanie nowej nazwy procesu,
2. ustawienie nowej nazwy procesu,
3. *demonizacja* procesu.

Zostały one szczegółowo omówione w kolejnych podrozdziałach.

#### 6.1. Losowanie i ustawianie nowej nazwy procesu

Operacja zmiany nazwy procesu rozpoczyna się od ustalenia liczby znaków nazwy początkowej programu, która przechowywana jest w pierwszym argumencie wywołania programu - `argv[0]`. Ważne jest, aby nazwa ta była długa np. 200 znaków, ponieważ w dalszej części programu nowa nazwa będzie losowana pośród istniejących procesów, których nazwa jest krótsza lub równa nazwie początkowej programu. Maksymalna długość większości procesów w pliku `/proc/<identyfikator procesu>/status`, z którego kopiowana będzie wylosowana nazwa, to 15 znaków. Im nazwa pierwotna jest dłuższa, tym większa liczba nazw procesów w losowaniu. Aby zapewnić losowanie spośród wszystkich dostępnych nazw, jeżeli program uruchomiono z nazwą krótszą niż 200 znaków, wywołuje on na początku swojego działania funkcję

```
exec1(argv[0], new_name, (char *)NULL),
```

która uruchamia proces jeszcze raz z nową nazwą i kończy obecny. Zmienną `new_name` uzupełniono znakiem „.” i dużą liczbę spacji – 200. Parametr, do którego przypisano liczbę znaków, nazwano `space`.

Następnie w wiecznej pętli uruchamiana jest funkcja przypisująca procesowi nową nazwę:

```
set_new_process_name(char *argv[], size_t space)
```

przy czym między każdą zmianą nazwy mija losowy czas uśpienia procesu, co ma na celu uniemożliwienie predykcji momentu zmiany nazwy (realizuje to m.in. cel zmniejszenia przewidywalności działań maskujących procesu ukrytego). Podana funkcja w swoim obrębie wywołuje procedurę losującą nową nazwę, która została opisana w dalszej części pracy (strona 24). Po wylosowaniu pasującej nazwy, znaki z `argv[0]`, których pozycja przekracza dostępną długość nowej nazwy zamieniane są na NULL przy użyciu funkcji

```
memset(argv[0], '\0', space),
```

aby po podmianie nazwy w pliku `/proc/<identyfikator procesu>/cmdline` poprzez

```
strncpy(argv[0], new_process_name, space - 1),
```

po jej ostatnim znaku wystąpił znak NULL, oznaczający koniec napisu. Do zmodyfikowania nazwy w pliku `/proc/<identyfikator procesu>/status` użyto funkcji `prctl(PR_SET_NAME, new_process_name)`.

Funkcję przypisującą procesowi nową nazwę przy uwzględnieniu opisanych powyżej operacji przedstawiono na poniższym listingu (kod źródłowy 6.1).

```
void set_new_process_name(char *argv[], size_t
space){
    new_process_name = get_rand_process_name();
    if(strlen(new_process_name)>space-1)
        sleep(1);
    set_last_char_to_null_term(new_process_name);
    memset(argv[0], '\0', space);
    strncpy(argv[0], new_process_name, space - 1);
    prctl(PR_SET_NAME, new_process_name);
}
```

*Kod źródłowy 6.1 Funkcja ustawiająca nową nazwę procesu*

Wykorzystana powyżej funkcja `get_rand_process_name()` zwraca adres pierwszego elementu napisu, który będzie nowo wylosowaną nazwą. Jej logika opiera się na iteracji przez elementy folderu `/proc`. Elementy te są folderami z identyfikatorem procesu w nazwie. Poniżej został przytoczony fragment omawianej funkcji (kod źródłowy 6.2), odpowiedzialny za policzenie procesów i zapis do zmiennej `proc_amount`.

```
char *get_rand_process_name()
{
    while ((dp = readdir(dir)) != NULL)
    {
        if(is_numeric(dp->d_name)!=0)
            proc_amount += 1;
    }
    closedir(dir);
}
```

*Kod źródłowy 6.2 Funkcja losująca nową nazwę procesu – liczenie procesów*

W dalszej części tej funkcji, losowana jest pozycja z pomiędzy zera i bieżącej liczby procesów pomniejszonej o 1. Dla wylosowanej pozycji pobierany jest identyfikator procesu, który użyty jest do pozyskania odpowiadającej mu nazwy procesu z pliku `/proc/<identyfikator procesu>/status`. Wylosowana pozycja prawie zawsze odpowiada pozycji któregoś z procesów. Wyjątkiem jest sytuacja, gdy pomiędzy policzeniem liczby procesów, a wyborem wylosowanej pozycji, zmieniła się liczba aktualnie wykonywanych procesów na mniejszą, niż liczba przypisana wcześniej do zmiennej `proc_amount`. Dzieje się to bardzo rzadko, ponieważ instrukcje odpowiedzialne za te czynności, są wykonywane w kodzie zaraz po sobie, czyli w rzędzie milisekund. Jednakże, aby do tego nie dopuścić, podczas iteracji przez elementy `/proc`, w celu pobrania nazwy elementu odpowiadającego wylosowanej pozycji, zapisywana jest nazwa każdego iterowanego elementu (identyfikator procesu). Jeżeli pozycja nie została odnaleziona (liczba procesów jest



mniejsza niż liczba określająca wylosowaną pozycję), pobierany jest ostatnio zapisany identyfikator procesu. Używając takiego podejścia nie trzeba zapisywać nigdzie w programie listy identyfikatorów procesów. Po zamknięciu deskryptora folderu */proc* zwracana jest wylosowana nazwa. Uproszczony fragment funkcji przedstawiono poniżej w kodzie źródłowym 6.3.

```
while ((dp = readdir(dir)) != NULL){
    if(is_numeric(dp->d_name)!=0){
        incr += 1;
        memset(name, '\0', sizeof(name));
        strncpy(name, dp->d_name, sizeof(name));
        if(incr==ran)
            break;
    }
}
closedir(dir);
pid_int = atoi(name);
sprintf(command, "ps -p %d -o comm=", pid_int);
FILE * ps = popen(command, "r");
memset(name, '\0', strlen(name));
while (fgets(name, sizeof(name), ps) != 0) {}
pclose(ps);
return name;
```

*Kod źródłowy 6.3 Funkcja losująca nową nazwę procesu – wybór nazwy*

## 6.2. Demonizacja procesu

Według opisu *demonizacji* z podrozdziału 3.2. „Zmiana sesji i identyfikatora procesu” zaimplementowano funkcję `daemonize()`.

Najpierw tworzony jest nowy proces (działający w tle) przy użyciu funkcji systemowej `fork()` służącej do duplikowania procesów. Proces, z którego została

wywołana funkcja zostaje tzw. *rodzicem*, natomiast nowy proces tzw. *dzieckiem*. Jeżeli jej wywołanie powiodło się, funkcja w procesie *dziecka* zwraca „0”, a w procesie *rodzica* identyfikator procesu *dziecka*. W przypadku implementacji *demon*a, proces *rodzica* jest kończony, po czym *rodzicem* nowego procesu zostaje proces *init*. Jeżeli wywołanie funkcji `fork()` nie powiodło się, co sygnalizowane jest zwrotem wartości mniejszej od „0” – następuje wyjście z programu z błędem zgodnie z założeniem dotyczącym obsługi błędów (kod źródłowy 6.4).

```
pid_t pid = fork();
if (pid < 0)
    exit(EXIT_FAILURE);
if (pid > 0)
    exit(EXIT_SUCCESS);
```

*Kod źródłowy 6.4 Obsługa funkcji fork()*

Kolejnym etapem jest zmiana sesji. W tym celu użyto funkcji `setsid()`, która ustawia proces liderem nowej sesji, jak i liderem nowo powstałej grupy procesów w tej sesji. Po jej wywołaniu proces nie jest powiązany z terminalem. Jeżeli nie udało się pomyślnie wykonać wywołania, zwracany jest jedyny możliwy błąd `EPERM`. Oznacza on, że proces wywołujący `setsid()` jest już liderem grupy procesów w nowo powstałej sesji, co znaczyłoby, że sesja się nie zmieniła. Jeżeli funkcja zwróciła ten błąd, po sekundzie nastąpi ponowna próba jej wykonania (kod źródłowy 6.5). W przypadku powtórnego błędu program zakończy się zgodnie z założeniem dotyczącym obsługi błędów.

```
if (setsid() < 0){
    sleep(1);
    if (setsid() < 0)
        exit(EXIT_FAILURE);
}
```

*Kod źródłowy 6.5 Zmiana sesji*

W kolejnym kroku ponownie wywoływana jest instrukcja `fork()`, a proces *rodzica* jest kończony. Zapewnia to usunięcie procesu będącego liderem sesji, co z kolei powoduje, że identyfikator procesu potomnego będzie różnił się od identyfikatora sesji.

W dalszym etapie zaimplementowano obsługę sygnałów. W tym przypadku jest to ich ignorowanie (kod źródłowy 6.6), jednakże obsługa sygnałów mogłaby być inna w zależności od właściwego celu działania ukrywającego się procesu.

```
struct sigaction act;
act.sa_handler = SIG_IGN;
for(int i = 1 ; i < 65 ; i++){
    if((i != SIGKILL) && (i != SIGSTOP) && (i != 32)
    && (i != 33))
        assert(sigaction(i, &act, NULL) == 0);
}
```

*Kod źródłowy 6.6 Obsługa sygnałów*

Dalej realizowana jest zmiana praw dostępu do pliku oraz zmiana folderu, w którym aktualnie pracuje proces. W implementacji wyzerowano maskę praw dostępu, a jako katalog roboczy ustawiono folder główny *root* „/”(kod źródłowy 6.7). Katalog jest zmieniany, aby proces nie utrzymywał w użyciu folderu, z którego został powołany, co zostało dokładniej omówione we wcześniejszej części pracy.

```
umask(0);
chdir("/");
```

*Kod źródłowy 6.7 Zmiana maski i folderu*

Następnie należało zamknąć wszystkie otwarte deskryptory plików, które mogły zostać odziedziczone z procesu *rodzica*. Pewnie było to zrobić iteracją po wartościach zwracanych z funkcji `sysconf(_SC_OPEN_MAX)`, która zwraca liczbę

otwartych przez proces deskryptorów plików. W ten sposób oprócz zamknięcia deskryptorów wejścia, wyjścia i wyjścia diagnostycznego zamknięte zostaną wszystkie inne deskryptory, które mogłyby być otwarte dla procesu (kod źródłowy 6.8).

```
for (int x = sysconf(_SC_OPEN_MAX); x>=0; x--)  
    close (x);
```

*Kod źródłowy 6.8 Zamknięcie deskryptorów plików*

### 6.3. Ustawienie parametru czasu dla zmiany sesji

Funkcja realizująca proces *demonizacji* wywoływana jest w pętli co losowy kwant czasu. W programach wykorzystywanych w praktyce, przed uśpieniem na losowy czas, powinna być realizowana procedura wykonująca właściwy cel ukrywającego swój proces programu. Złośliwe, bądź szpiegujące oprogramowanie mogłoby m.in. powoływać ukryty proces będący *keyloggerem* czy osiągalnym serwerem TCP, do którego zdalnie łączyłby się użytkownik innego hosta.

W celu skutecznego zakamuflowania procesu i utrudnienia jego detekcji, procedurę *demonizacji* wykonuje się co pewien czas, wylosowany z odpowiednio dobranego przedziału. Na poniższym rysunku 6.1, przedstawiono wynik polecenia *ps tree*, dla trzech kolejnych zmian sesji i identyfikatora procesu w odstępie 5 sekund.

```

-ibus-x11(1708)---{ibus-x11}(1721)
                  {ibus-x11}(1722)
-nautilus(2183)---{nautilus}(2185)
                  {nautilus}(2186)
                  {nautilus}(2193)
                  {nautilus}(2194)
-pulseaudio(1554)---{pulseaudio}(1585)
                   {pulseaudio}(1588)
-vmtoolsd(1781)---{vmtoolsd}(1968)
                  {vmtoolsd}(1970)
                  {vmtoolsd}(96594)
-xdg-permission-(1630)---{xdg-permission-}(1631)
                       {xdg-permission-}(1633)
-zmieniam_sesje(106287)

-ibus-x11(1708)---{ibus-x11}(1721)
                  {ibus-x11}(1722)
-nautilus(2183)---{nautilus}(2185)
                  {nautilus}(2186)
                  {nautilus}(2193)
                  {nautilus}(2194)
-pulseaudio(1554)---{pulseaudio}(1585)
                   {pulseaudio}(1588)
-vmtoolsd(1781)---{vmtoolsd}(1968)
                  {vmtoolsd}(1970)
                  {vmtoolsd}(96594)
-xdg-permission-(1630)---{xdg-permission-}(1631)
                       {xdg-permission-}(1633)
-zmieniam_sesje(106298)

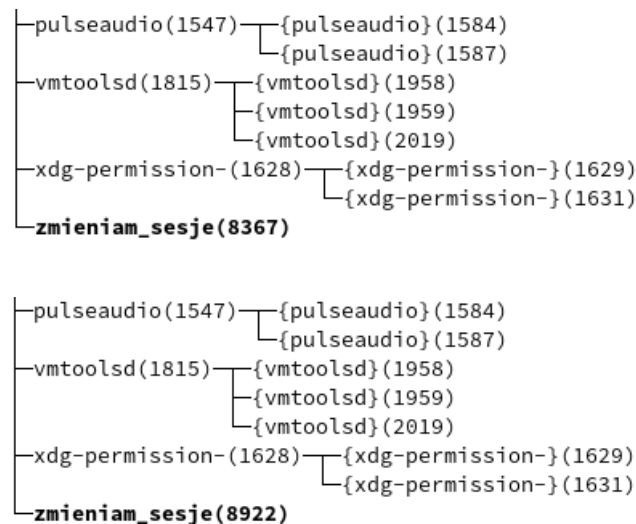
-ibus-x11(1708)---{ibus-x11}(1721)
                  {ibus-x11}(1722)
-nautilus(2183)---{nautilus}(2185)
                  {nautilus}(2186)
                  {nautilus}(2193)
                  {nautilus}(2194)
-pulseaudio(1554)---{pulseaudio}(1585)
                   {pulseaudio}(1588)
-vmtoolsd(1781)---{vmtoolsd}(1968)
                  {vmtoolsd}(1970)
                  {vmtoolsd}(96594)
-xdg-permission-(1630)---{xdg-permission-}(1631)
                       {xdg-permission-}(1633)
-zmieniam_sesje(106311)

```

Rysunek 6.1 Wynik polecenia *ps*tree dla trzech następujących po sobie zmian sesji i identyfikatora w odstępie 5 sekund

Warto zwrócić uwagę, że jeśli sesja jest zmieniana zbyt często, to każdy nowy identyfikator procesu, jest bardzo bliski do wartości poprzedniego. Prowadzi to do konkluzji, iż czas ponownej operacji *demonizacji* i zmiany nazwy powinien być roztropnie dobrany. Identyfikator procesu powinien zmieniać się o jak największą wartość. Z drugiej strony, czas nie może być zbyt długi, aby utrudnić skojarzenie

prawdziwego działania procesu z jego identyfikatorem i wyłączenie go sygnałem SIGKILL, który nie może zostać zignorowany. Dodatkowe funkcje procesu powinny wykonać się w obrębie tego czasu. Poniżej na rysunku 6.2 przedstawiono zmianę sesji i nazwy dla parametru czasu ustawionego na 3 minuty.



Rysunek 6.2 Wynik polecenia *ps tree* dla trzech następujących po sobie zmian sesji i identyfikatora w odstępie 3 minut

Po odczekaniu 3 minut, identyfikator procesu zwiększył się o ponad 500. Poprzez taką różnicę, ciężko byłoby wydedukować, że pierwszy proces był *rodzicem* drugiego (w szczególności, gdyby nazwa też była zmieniona). Liczba ta będzie różnić się w zależności od liczebności powołanych w tym czasie innych procesów w systemie. Im więcej ich będzie tym większy identyfikator zostanie przydzielony procesowi po zmianie sesji.

Proces *rodzic* przed powołaniem procesu *potomnego* powinien zostać uśpiony z jeszcze jednego, najważniejszego powodu. Otóż podczas pierwszego uruchomienia programu, jego proces przybiera najwyższy identyfikator jako najnowszy proces w systemie. Gdyby dziedziczące po sobie procesy w pętli, zawsze przyjmowały najwyższe identyfikatory – ułatwiałoby to predykcję zachowania procesów i wykrycie, że próbują się ukryć. Czas wylosowany z przedziału od 2 do 4 minut między kolejnymi zmianami sesji, zapewnia pojawienie się w systemie od

kilkudziesięciu do kilkuset nowych aktywnych procesów, których identyfikator będzie większy niż identyfikator procesu *dziecka* omawianego procesu.

## ROZDZIAŁ VII.

### TESTY AKCEPTACYJNE

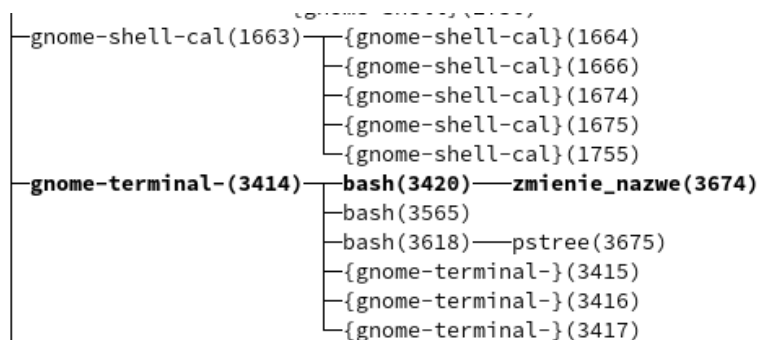
W ramach testów poprawności działania programu oddzielnie przetestowano funkcję zmiany nazwy i funkcję zmiany sesji i identyfikatora procesu używając polecenia *ptree*, które wyświetla działające procesy w strukturze drzewa, którego korzeniem jest proces *init*. Stanowi to wizualną alternatywę dla poleceń takich jak *ps*, które wyświetla listę uruchomionych procesów lub *top*, które również wyświetla procesy, ale na bieżąco aktualizuje swój wynik. Następnie wykonano test całego programu używając rejestrowania działania procesu w systemowym dzienniku zdarzeń.

W celu zweryfikowania możliwości identyfikacji ukrywającego się procesu posłużono się popularnym narzędziem dla systemów GNU/Linux: *Unhide*. Podczas przeprowadzonych testów wykorzystano autorski program demonstracyjny, który (nie wykonując żadnych niepożądanych działań) próbuje zataić swoje funkcjonowanie przy wykorzystaniu technik opisanych w niniejszej pracy. Uwzględniając powyższe, uzyskane wyniki będzie można uznać za adekwatną ocenę skuteczności powziętych działań w zakresie ukrywania procesów.

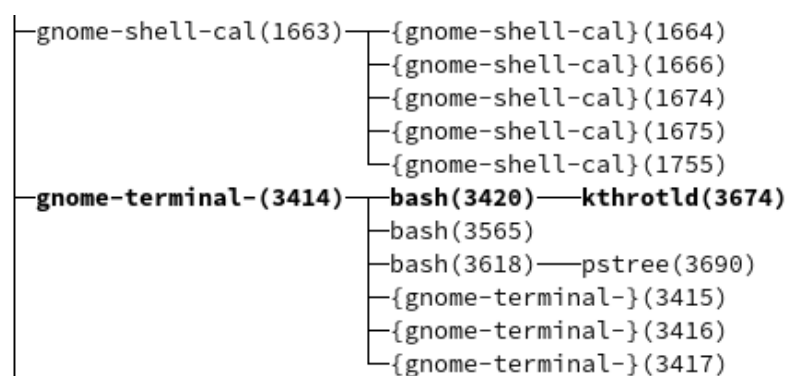
#### 7.1. Test poprawności działania zmiany nazwy procesu

W celu przetestowania poprawności działania funkcji zmieniającej nazwę procesu użyto systemowego narzędzia *ptree*. Na rysunkach 7.1 i 7.2 przedstawiono wynik polecenia *ptree* z zaznaczeniem identyfikatora procesu przed i po zmianie nazwy przez proces.





Rysunek 7.1 Proces przed zmianą nazwy

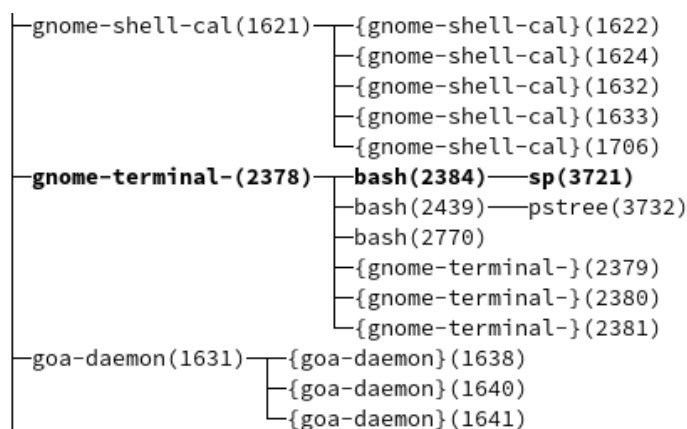


Rysunek 7.2 Proces po zmianie nazwy

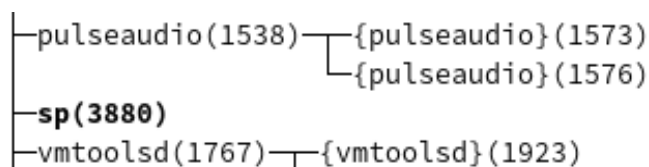
Nazwa została poprawnie zamieniona z *zmienie\_nazwe* na wylosowaną *kthrotld*, a identyfikator procesu oraz sesja (*rodzicem* procesu na obu rysunkach jest proces *bash*, a jego *rodzicem* terminal *gnome*) pozostały niezmienione.

## 7.2. Test poprawności działania zmiany sesji i identyfikatora procesu

Na rysunkach 7.3 i 7.4 przedstawiono wyniki polecenia *ps tree*, na których uwidoczniiono przykładową zmianę sesji i identyfikatora dla procesu wykorzystującego nazwę *sp*.



Rysunek 7.3 Proces przed zmianą sesji i identyfikatora



Rysunek 7.4 Proces po zmianą sesji i identyfikatora

Na rysunku 7.3 liderem sesji jest terminal *gnome*, którego procesem potomnym jest proces *bash*, będący rodzicem dla procesu *sp*. Po *demonizacji*, proces *sp* znajduje się w nowej sesji i posiada nowy identyfikator procesu, który - dzięki ponownemu użyciu funkcji `fork()` - nie jest identyfikatorem sesji.

### 7.3. Test poprawności działania pełnego programu demonstracyjnego

W celu przetestowania poprawności działania programu łączącego wszystkie opisane i zaimplementowane techniki, konieczna była implementacja zapisu informacji o powołaniu procesu *demon*a np. do systemowego dziennika zdarzeń (zasobu, w którym notowane są różnorodne zdarzenia, takie jak komunikaty o błędach systemu, uruchamianie i zamykanie systemu, zmiana konfiguracji sieci itp.). Do realizacji opisanego działania użyto funkcji `syslog()`. Należy zaznaczyć, że skorzystanie z możliwości zapisu do systemowego dziennika zdarzeń zostało dodane do programu demonstracyjnego jedynie w celach testowych, gdyż

w rzeczywistości zapisywanie jakichkolwiek informacji o ukrytym procesie może posłużyć do jego wykrycia.

W systemie Fedora, na którym testowano działanie programu, aktywność procesów zapisywana jest w dzienniku `/var/log/messages`. Jest to domyślny dziennik zdarzeń dla nieskonkretyzowanych strumieni informacji. Każdy wiersz w tym pliku to pojedynczy komunikat odnotowany przez program lub usługę. Linie są podzielone na pięć części:

1. data i godzina wiadomości,
2. nazwa hosta, z którego pochodzi komunikat,
3. nazwa (procesu lub) usługi, która wygenerowała komunikat,
4. identyfikator procesu (zawarty w nawiasach kwadratowych) programu, który wygenerował komunikat,
5. treść komunikatu.

Program skompilowano i zapisano jako *wersja\_demonstracyjna*, a następnie uruchomiono. Nazwę usługi, która generuje komunikat zdefiniowano jako *log\_ukryty\_proces*, a jako jego treść ustawiono nazwę procesu. Następnie filtrując zawartość pliku `/var/log/messages`, aby wyświetlone zostały tylko linie zawierające tę nazwę, otrzymano historię zmian działań przedmiotowego procesu. Listing przedstawiono na rysunku 7.5, na którym proces rozpoczął działanie z nazwą *./wersja\_demonstracyjna* i identyfikatorem *118817*, a następnie co losowy czas zmieniał swoją nazwę i identyfikator procesu.

```
$ sudo strings /var/log/messages | grep "log_ukryty_proces"
Jan  8 01:27:22 fedora log_ukryty_proces[118817]: ./wersja_demonstracyjna
Jan  8 01:27:49 fedora log_ukryty_proces[118897]: tracker-store
Jan  8 01:28:16 fedora log_ukryty_proces[118957]: rcu_sched
Jan  8 01:28:43 fedora log_ukryty_proces[119011]: gsd-screensaver
Jan  8 01:29:10 fedora log_ukryty_proces[119057]: mpt_poll_0
Jan  8 01:29:37 fedora log_ukryty_proces[119128]: mpt/0
Jan  8 01:30:04 fedora log_ukryty_proces[119169]: ModemManager
Jan  8 01:30:31 fedora log_ukryty_proces[119208]: kdmflush
Jan  8 01:30:58 fedora log_ukryty_proces[119262]: gnome-session-b
Jan  8 01:31:25 fedora log_ukryty_proces[119313]: gnome-shell
Jan  8 01:31:53 fedora log_ukryty_proces[119348]: scsi_eh_0
Jan  8 01:32:20 fedora log_ukryty_proces[119389]: sssd
Jan  8 01:32:47 fedora log_ukryty_proces[119438]: ssh-agent
Jan  8 01:33:14 fedora log_ukryty_proces[119482]: udisksd
Jan  8 01:33:41 fedora log_ukryty_proces[119521]: gsd-power
Jan  8 01:34:08 fedora log_ukryty_proces[119569]: gvfs-afc-volume
Jan  8 01:34:35 fedora log_ukryty_proces[119624]: at-spi-bus-laun
Jan  8 01:35:02 fedora log_ukryty_proces[119670]: irq/36-pciehp
Jan  8 01:35:29 fedora log_ukryty_proces[119715]: atd
Jan  8 01:35:56 fedora log_ukryty_proces[119768]: sssd_be
Jan  8 01:36:23 fedora log_ukryty_proces[119820]: dbus-broker-lau
Jan  8 01:36:50 fedora log_ukryty_proces[119896]: irq/46-pciehp
```

Rysunek 7.5 Wyświetlenie informacji o ukrytym procesie w dzienniku zdarzeń

## 7.4. Test wykrywalności przy użyciu narzędzia Unhide

Program *Unhide* jest narzędziem skanującym system w celu wykrycia procesów, które ukrywają w nim swoją obecność. Przeprowadza on wiele elementarnych testów np. porównanie zawartości */proc* i informacji zebranych na wskutek wywołanych funkcji systemowych z wynikiem polecenia *ps*. Na rysunku 7.6 przedstawiono wynik działania programu *Unhide* dla systemu, w którym został uruchomiony proces ukrywający swą obecność przy wykorzystaniu opisanych wcześniej technik.

```

$ sudo unhide -d -m -v proc procall procfs quick reverse sys brute
[sudo] password for K4r01:
Unhide 20130526
Copyright © 2013 Yago Jesus & Patrick Gouin
License GPLv3+ : GNU GPL version 3 or later
http://www.unhide-forensics.info

NOTE : This version of unhide is for systems using Linux >= 2.6

Used options: verbose brutesimplecheck morecheck
[*]Searching for Hidden processes through /proc stat scanning

[*]Searching for Hidden processes through /proc chdir scanning

[*]Searching for Hidden processes through /proc opendir scanning

[*]Searching for Hidden thread through /proc/pid/task readdir scanning

Warning : Cannot open /proc/135285/task directory ! ! Skipping process 135285.
[*]Searching for Hidden processes through getpriority() scanning

[*]Searching for Hidden processes through getpgid() scanning

[*]Searching for Hidden processes through getsid() scanning

[*]Searching for Hidden processes through sched_getaffinity() scanning

[*]Searching for Hidden processes through sched_getparam() scanning

[*]Searching for Hidden processes through sched_getscheduler() scanning

[*]Searching for Hidden processes through sched_rr_get_interval() scanning

[*]Searching for Hidden processes through kill(..,0) scanning

[*]Searching for Hidden processes through comparison of results of system call

```

*Rysunek 7.6 Wynik skanowania systemu programem Unhide podczas działania procesu ukrywającego się*

Narzędzie wyświetliło jedno ostrzeżenie, przy użyciu metody *readdir scanning*, która w uproszczeniu najpierw pobiera listę procesów z */proc*, po czym sprawdza czy istnieje folder *proc/<identyfikator procesu>/task* dla każdego procesu z tej listy. W przypadku procesu z identyfikatorem *135285*, który faktycznie był ukrywającym się procesem – treść ostrzeżenia zawierała zapis, że nie można otworzyć tego folderu. Po kilkukrotnym uruchomieniu *Unhide* zauważono, że ostrzeżenie nie pojawia się zawsze. Stąd konkluzja, że było ono wyświetlane gdy ukrywający się proces zmienił swoją sesję i zamknął proces *rodzica* między

pobranie przez program *Unhide* listy procesów, a przed przeczytaniem katalogu *proc/<identyfikator procesu>/task*. Taki scenariusz wyjaśniałby przyczynę pojawienia się zaistniałego ostrzeżenia. W celu sprawdzenia tej możliwości uruchomiono jeszcze raz tą metodę skanowania, a podczas jej działania włączano i wyłączano naprzemiennie przeglądarkę internetową. Zaproponowane wyżej wyjaśnienie zostało w ten sposób potwierdzone, ponieważ wynik narzędzia zwrócił to samo ostrzeżenie dla zakończonych procesów przeglądarki (rysunek 7.7).

```
$ sudo unhide -d -v procall
[sudo] password for K4r01:
Sorry, try again.
[sudo] password for K4r01:
Sorry, try again.
[sudo] password for K4r01:
Unhide 20130526
Copyright © 2013 Yago Jesus & Patrick Gouin
License GPLv3+ : GNU GPL version 3 or later
http://www.unhide-forensics.info

NOTE : This version of unhide is for systems using Linux >= 2.6

Used options: verbose brutesimplecheck
[*]Searching for Hidden processes through /proc stat scanning

[*]Searching for Hidden processes through /proc chdir scanning

[*]Searching for Hidden processes through /proc opendir scanning

[*]Searching for Hidden thread through /proc/pid/task readdir scanning

Warning : Cannot open /proc/4293/task directory !! Skipping process 4293. [No such file or directory]
Warning : Cannot open /proc/5445/task directory !! Skipping process 5445. [No such file or directory]
Warning : Cannot open /proc/5498/task directory !! Skipping process 5498. [No such file or directory]
Warning : Cannot open /proc/5646/task directory !! Skipping process 5646. [No such file or directory]
Warning : Cannot open /proc/5675/task directory !! Skipping process 5675. [No such file or directory]
Warning : Cannot open /proc/5694/task directory !! Skipping process 5694. [No such file or directory]
Warning : Cannot open /proc/5875/task directory !! Skipping process 5875. [No such file or directory]
```

*Rysunek 7.7 Wynik skanowania systemu programem Unhide podczas włączania i wyłączania przeglądarki internetowej.*

Osoba skanująca system tą metodą, mogłaby uznać ostrzeżenie za sprawą ukrywającego się procesu za pomyłkę (tzw. *false positive*), ponieważ może ono również znaczyć, że jakiś proces zakończył się podczas skanowania systemu przez program *Unhide*.

## PODSUMOWANIE

W pracy tej przedstawiono i omówiono podstawowe mechanizmy dostępne w przestrzeni użytkownika i wykazano (poprzez implementację własnego rozwiązania), że programy działające w tym trybie mogą skutecznie ukrywać działanie swoich procesów. Ich możliwości są jednak mniejsze od procesów używających mechanizmów przestrzeni jądra systemu. Z tego względu są one rzadziej stosowane w rzeczywistych scenariuszach. W zaimplementowanym rozwiązaniu ukrycie procesu opierało się na zmianie jego nazwy, sesji oraz identyfikatora procesu.

Przeprowadzone testy akceptacyjne programu udały się, ponieważ realizował on swoje funkcje prawidłowo oraz wykryła go tylko jedna technika narzędzia *Unhide*. Należy jednak zwrócić uwagę, że program realizował wyłącznie ukrycie swojego procesu bez żadnych złośliwych funkcji, które w zależności od ich działania i sposobu implementacji mogłyby zostać wykryte.

Kolejnym krokiem rozbudowy programu mogłoby być zaimplementowanie dodatkowego działania ukrywającego się procesu. Mógłby on być serwerem, który utrzymywałby zdalne połączenie z innym hostem (np. wykorzystując protokół UDP w celu nieujawniania się poprzez widoczną rezerwację portu TCP) lub procesem wykorzystującym moc obliczeniową zainfekowanego hosta do wydobywania kryptowalut.

Analiza, projektowanie i implementacja technik ukrycia procesu w trybie użytkownika wymagały dokładnego przestudiowania wielu zagadnień związanych z systemami GNU/Linux takimi, jak montowanie i działanie systemów plików, czy mechanizm zarządzania procesami i ich komunikacja z jądrem systemu za pomocą wywołań systemowych. Wszystkie założenia i zadania udało się pomyślnie zrealizować.

## BIBLIOGRAFIA

1. Arciuch A., Liderman K.: *Projektowanie systemów komputerowych*, Warszawa 2001
2. Harbison III S. P., Steele Jr. G. L., *C: A Reference Manual*, Upper Saddle River, NJ 2002
3. Rochkind M., *Programowanie w systemie UNIX® dla zaawansowanych*, Warszawa 2007
4. Celebi K., Suski Z., *Rootkit dla dydaktycznego systemu Linux*, Przegląd teleinformatyczny nr 1-2, Instytut Teleinformatyki i Automatyki WAT, Warszawa, 2016.
5. [https://en.wikipedia.org/wiki/Linux\\_kernel](https://en.wikipedia.org/wiki/Linux_kernel)
6. <https://www.tecmint.com/linux-process-management/>
7. <https://pl.wikibooks.org/wiki/Linux/Procesy>
8. <https://www.maketecheasier.com/hide-processes-from-other-users-linux/>
9. [https://en.wikipedia.org/wiki/Protection\\_ring](https://en.wikipedia.org/wiki/Protection_ring)
10. <https://www.tldp.org/LDP/Linux-Filesystem-Hierarchy/html/proc.html>
11. <https://linuxinsomnia.wordpress.com/2010/12/04/daemon-process/>
12. <https://block-chain.com/news/hidden-cryptocurrency-mining-essence-harm-and-ways-of-protecting>



## WYKAZ RYSUNKÓW

Rysunek 2.1 Schemat pierścieni uprawnień dla architektury x86.....	10
Rysunek 3.1 Przykładowy widok zawartości folderu /proc.....	13
Rysunek 3.2 Przykładowa zawartość folderu procesu.....	13
Rysunek 5.1 Diagram przepływu danych.....	20
Rysunek 6.1 Wynik polecenia pstree dla trzech następujących po sobie zmian sesji i identyfikatora w odstępie 5 sekund.....	29
Rysunek 6.2 Wynik polecenia pstree dla trzech następujących po sobie zmian sesji i identyfikatora w odstępie 3 minut.....	30
Rysunek 7.1 Proces przed zmianą nazwy.....	33
Rysunek 7.2 Proces po zmianie nazwy.....	33
Rysunek 7.3 Proces przed zmianą sesji i identyfikatora.....	34
Rysunek 7.4 Proces po zmianą sesji i identyfikatora.....	34
Rysunek 7.5 Wyświetlenie informacji o ukrytym procesie w dzienniku zdarzeń.....	36
Rysunek 7.6 Wynik skanowania systemu programem Unhide podczas działania procesu ukrywającego się.....	37
Rysunek 7.7 Wynik skanowania systemu programem Unhide podczas włączania i wyłączania przeglądarki internetowej.....	38

## WYKAZ KODÓW ŹRÓDŁOWYCH

Kod źródłowy 6.1 Funkcja ustawiająca nową nazwę procesu.....	23
Kod źródłowy 6.2 Funkcja losująca nową nazwę procesu – liczenie procesów.....	24
Kod źródłowy 6.3 Funkcja losująca nową nazwę procesu – wybór nazwy.....	25
Kod źródłowy 6.4 Obsługa funkcji fork().....	26
Kod źródłowy 6.5 Zmiana sesji.....	26
Kod źródłowy 6.6 Obsługa sygnałów.....	27
Kod źródłowy 6.7 Zmiana maski i folderu.....	27
Kod źródłowy 6.8 Zamknięcie deskryptorów plików.....	28