Rootkit dla dydaktycznego systemu Linux

Karol CELEBI [1], Zbigniew SUSKI [2]

Instytut Teleinformatyki i Automatyki WAT,
ul. Gen. S. Kaliskiego 2, 00-908 Warszawa
[1] karol.celebi@gmail.com
[2] zbigniew.suski@wat.edu.pl

STRESZCZENIE: W artykule opisano projekt, którego celem było opracowanie pakietu *rootkit* dla systemu Linux. Przedstawiono charakterystykę najważniejszych mechanizmów stosowanych w tego typu konstrukcjach, koncepcję rozwiązania i sposób implementacji. Zamieszczono również wybrane wyniki testów opracowanego oprogramowania. Opracowany pakiet przewidziany jest do zastosowania w wybranych maszynach wirtualnych wykorzystywanych w specjalistycznych laboratoriach komputerowych Wydziału Cybernetyki. Stąd przymiotnik „dydaktyczny" w tytule artykułu.

SŁOWA KLUCZOWE: bezpieczeństwo, systemy Linux, *rootkit,* dydaktyka.

1. Wstęp

Rootkit to narzędzie wykorzystywane często w czasie ataków na systemy informatyczne. Jego podstawowe zadanie, oprócz ewentualnej działalności destrukcyjnej, polega na ukrywaniu, wprowadzonych przez intruza, niebezpiecznych obiektów, takich jak np. pliki i procesy. Wspomniane pliki i procesy, zwykle umożliwiają intruzowi utrzymanie kontroli nad systemem, m.in. mogą umożliwić dostęp do zasobu systemu, dla którego wymagane są uprawnienia administratora systemu. Często *rootkit* ukrywa również informację o połączeniach sieciowych wykorzystywanych przez intruza.

Historia rozwoju *rootkitów* sięga lat 80 ubiegłego stulecia. Jednak zyskały one dużą popularność dopiero w ostatnich latach [1]. Oprogramowanie przeciwdziałające zagrożeniom jest już na tyle dopracowane, że konwencjonalne konstrukcje wrogiego oprogramowania nie zapewniają często niewykrywalności w atakowanym systemie. Nazwa *rootkit* wywodzi się ze środowisk unixowych: *root* – domyślna nazwa konta administratora; *kit* – z ang. zestaw narzędzi.

Pierwotne *rootkity* były pakietami oprogramowania zawierającymi zmodyfikowane, kluczowe binaria systemowe przeznaczone dla systemów unixowych. W czasie ataku, zastępowały one pakiety oryginalne. Dzięki modyfikacjom kodu oryginalnego, elementy z *rootkita* blokowały przekazywanie niektórych informacji (np. dotyczących plików lub procesów). Zwykle też, po podaniu specjalnego hasła, umożliwiały logowanie na konto administratora (*root*).

Rootkit infekuje system, a skutkiem jest przede wszystkim „ukrywanie" przed użytkownikiem, również administratorem lub skanerem antywirusowym, siebie samego oraz np. wprowadzonego do systemu procesu konia trojańskiego. Ukrywanie odbywa się najczęściej przez przejęcie wybranych funkcji systemu operacyjnego, służących np. listowaniu procesów lub plików w zadanym katalogu, a następnie „cenzurowaniu" zwracanych przez te funkcje wyników tak, aby informacja o ukrywanych przez *rootkit* obiektach, nie była prezentowana.

Można też spotkać *rootkity*, które nie służą wyrządzaniu szkód, ale mają za zadanie ułatwić realizację niektórych operacji. Przykładem są moduły wirtualnych napędów dyskowych, które pozwalają uzyskać niskopoziomowy dostęp do zasobów systemowych i sprzętowych, czy też elementy oprogramowania do wykrywania ataków czy infekcji (*rootkity* wykrywające inne *rootkity*). Niektóre laptopy zawierają specjalnie przygotowane *rootkity* umieszczone w oprogramowaniu mikroukładowym. Realizują one np. blokadę urządzenia lub awaryjne usuwanie wrażliwych danych po kradzieży sprzętu.

Celem projektu przedstawionego w niniejszym artykule było zbudowanie platformy, która umożliwiłaby „podglądanie" działań studentów w czasie wykonywania ćwiczeń laboratoryjnych. Udostępnia również prowadzącemu zajęcia, narzędzie aktywnego oddziaływania na środowisko ćwiczącego studenta.

Na rynku są dostępne narzędzia o podobnych właściwościach. Wymagają one jednak często zgody na „podglądanie". Istotne było również zapewnienie modyfikowalności rozumianej, jako możliwość praktycznie nieograniczonego rozszerzania właściwości funkcjonalnych. Z tych względów należy potraktować przedstawione rozwiązanie, jako demonstrator wybranych możliwości – swoisty produkt typu *proof of concept*. Podczas realizacji projektu wykorzystano i zweryfikowano wiele pomysłów przedstawionych w literaturze.

2. Charakterystyka mechanizmów *rootkit* w systemach Linux

Najczęściej spotykanym rodzajem *rootkitów* w systemach Linux są *rootkity* jądra systemu. Wykorzystują one modularność jądra systemu Linux,

która pozwala na rozszerzanie funkcjonalności systemu poprzez instalowanie tzw. ładowalnych modułów jądra.

2.1. Ładowalne moduły jądra

Ładowalne moduły jądra (*Loadable Kernel Module* - LKM) są to binaria ładowane przez system do przestrzeni jądra systemu. Ich zadaniem jest najczęściej umożliwienie obsługi urządzeń sprzętowych lub programowych (funkcja sterownika sprzętowego), umożliwienie obsługi systemów plików (funkcja sterownika systemu plików) czy też umożliwienie obsługi nowych funkcji systemowych [2], [3]. Podstawową zaletą i głównym powodem wykorzystywania ładowalnych modułów jądra jest to, iż mogą one rozszerzać funkcjonalność jądra bez potrzeby jego ponownej kompilacji czy też ponownego uruchamiania systemu. Dzięki temu możliwe jest przygotowanie jądra, które nie będzie wrażliwe na zmiany w konfiguracji sprzętowej.

Każdy moduł jądra musi zawierać dwie kluczowe funkcji: *module_init* oraz *module_exit*. Funkcja *module_init* odpowiada za przydzielenie dodatkowej pamięci wymaganej do działania modułu, oprócz pamięci przydzielanej przez jądro w przestrzeni pamięci jądra. Funkcja ta realizuje również powołanie dodatkowych wątków czy procesów. Analogicznie funkcja *module_exit* odpowiada za zwolnienie wcześniej zaalokowanej pamięci, zatrzymanie wątków czy procesów oraz inne operacje niezbędne do usunięcia modułu.

2.2. Funkcje systemowe

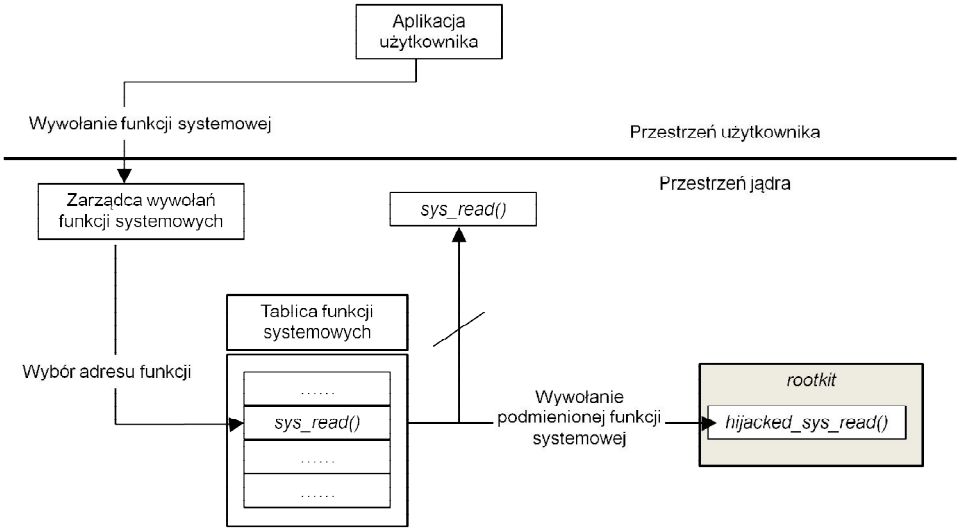
Funkcje systemowe (*system calls*), nazywane czasami wywołaniami systemowymi, są elementarnym mechanizmem jądra systemu Linux jak i innych systemów operacyjnych [4]. Każde wywołanie jakiejkolwiek funkcji w aplikacji trybu użytkownika, pociąga za sobą szereg wywołań funkcji systemowych na poziomie jądra systemu. Możliwe jest to dzięki zarządcy wywołań funkcji systemowych (*system calls handler*), który na potrzeby każdego wywołania, wyznacza adresy funkcji systemowych, wykorzystując tablicę funkcji systemowych (*system calls table*). Na podstawie adresu funkcji uzyskanego od zarządcy, jądro jest w stanie uaktywnić kod funkcji a następnie zwrócić wynik jej wykonania do funkcji, która ją wywołała [5], [6].

Funkcje systemowe są niewidoczne dla procesów działających w przestrzeni użytkownika. Właściwość ta jest szeroko stosowana przez *rootkity*, które podmieniając adresy funkcji systemowych mogą wykonać swój własny, często szkodliwy kod.

2.3. Tablica funkcji systemowych

Tablica funkcji systemowych [4], [7], nazywana również tablicą wywołań systemowych, jest strukturą przechowującą adresy kodu poszczególnych funkcji systemowych, w obszarze pamięci jądra systemu. Funkcje są identyfikowane za pomocą numerów i na podstawie numeru funkcji systemowej można ustalić jej adres w pamięci a następnie zainicjować wykonywanie. Począwszy od wersji 2.6.x jądra systemu Linux, adres tablicy funkcji systemowych przestał być eksportowany do biblioteki *syscalls.h*. Miało to na celu utrudnienie jawnego dostępu do wspomnianej tablicy i jej edycję.

Aby edytować zawartość tablicy funkcji systemowych, *rootkity* wykorzystują różne metody uzyskania jej adresu. Rysunek 1 przedstawia typowy schemat procesu podmiany funkcji systemowej poprzez edycję jej adresu w tablicy funkcji systemowych.

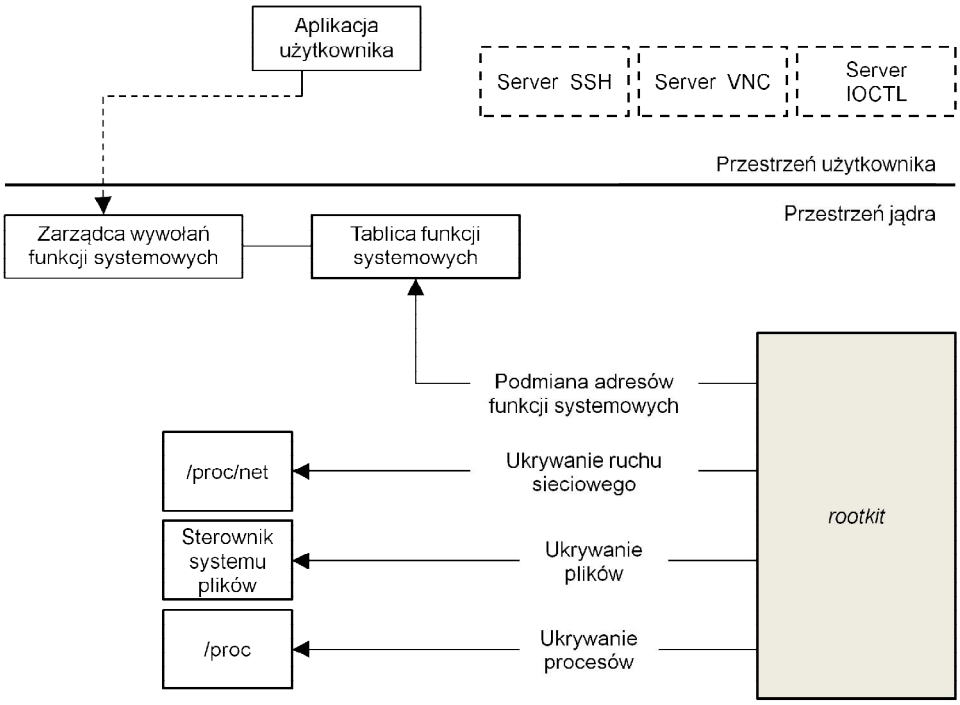


Rys. 1. Schemat podmiany funkcji systemowej przez *rootkit*

3. Koncepcja rozwiązania

Opracowując koncepcję rozwiązania przyjęto, że należy zastosować model modułowy. Poszczególne elementy *rootkita* powinny wejść w skład ładowalnego modułu jądra, który będzie komunikował się z serwerami usług uruchamianymi w przestrzeni użytkownika. Takie rozwiązanie zapewnia łatwiejsze zarządzanie kodem aplikacji, zwiększa możliwości w zakresie

potencjalnej rozbudowy, zmniejsza możliwość wykrycia przez programy antywirusowe. Ideę rozwiązania przedstawiono na rysunku 2.



Rys. 2. Idea rozwiązania

Przyjęty model ułatwia osiągnięcie zamierzonych rezultatów w dość prosty sposób. Wykorzystując obiekty jądra systemu można uzyskać bezpośredni, niskopoziomowy dostęp do pamięci, urządzeń zewnętrznych i innych składników systemu. Używając serwerów usług pracujących w trybie użytkownika, zyskuje się możliwość używania języków programowania wysokiego poziomu oraz gotowych bibliotek czy interfejsów programistycznych (API). Główny moduł *rootkita* pełni rolę medium synchronizującego i zarządzającego pracą pozostałych komponentów. Monitoruje on pracę serwerów usług. Może je w przypadku awarii lub na polecenie atakującego (zarządzającego[1]) włączyć, wyłączyć czy uruchomić ponownie (zrestartować). Zadaniem głównego modułu jest również „nadpisanie" zawartości tablicy

[1] Biorąc pod uwagę cel prezentowanego projektu, mianem zarządzającego określa się nauczyciela prowadzącego zajęcia, który ma wgląd w czynności wykonywane przez ćwiczącego studenta.

funkcji systemowych. Pełna lista funkcji realizowanych przez moduł główny obejmuje:

- odszukanie tablicy funkcji systemowych,
- dynamiczną[2] podmianę funkcji systemowych,
- dynamiczne ukrywanie procesów o zadanym identyfikatorze PID,
- dynamiczne ukrywanie plików i katalogów o zadanej nazwie,
- dynamiczne ukrywanie sieciowych kanałów transportowych, wykorzystujących określone porty, realizowanych z wykorzystaniem protokołów UDP oraz TCP,
- uzyskiwanie pełnych uprawnień administratora (*root*) bez jawnego uwierzytelnienia.

Proponowany pakiet obejmuje następujące moduły usług:

- Serwer SSH (*Secure Shell*) – moduł obsługujący zdalne połączenie terminalowe, zapewniający szyfrowanie przesyłanych danych, uruchamiany i zamykany po otrzymaniu specjalnego, spreparowanego pakietu *ICMP Echo Request*. Umożliwia aktywne oddziaływanie na środowisko użytkownika pracującego w zainfekowanym systemie.
- Serwer VNC (*Virtual Network Computing*) – moduł przekazywania obrazu z wirtualnego, bądź fizycznego środowiska graficznego. Umożliwia podgląd zawartości ekranu, bez możliwości interakcji - aby nie wzbudzać podejrzeń użytkownika zainfekowanej maszyny. Uruchamiany jest na żądanie atakującego (zarządzającego).
- *Keylogger* – moduł monitorujący wykorzystanie klawiatury. Realizuje zapis informacji o wciskanych klawiszach klawiatury do ukrytego pliku dziennika. Jest uaktywniany w czasie ładowania *rootkita* do pamięci systemu.
- Server IOCTL (*input/output control*) – moduł obsługi protokołu IOCTL. Umożliwia sterowanie pracą *rootkita* poprzez przechwytywanie pakietów IOCTL zawierających komendy i flagę autoryzacji. Współpracuje z dedykowanym programem klienta IOCTL, który umożliwia wysyłanie wspomnianych pakietów.

4. Implementacja

Projekt został zaimplementowany w postaci zbioru modułów programowych, obejmujących:

[2] W tym przypadku pod terminem „dynamiczna/dynamiczne", należy rozumieć „realizowana/realizowane" na bieżąco, na żądanie - w czasie rzeczywistym.

- ładowalny moduł jądra systemu Linux,
- aplikację pełniącą rolę serwera VNC,
- aplikację pełniącą rolę serwera SSH,
- aplikację służącą do sterowania pracą modułu jądra wykorzystującą interfejs IOCTL.

Wszystkie wykorzystane moduły projektu, zostały zaimplementowane w języku C. Dla każdego zostały utworzone pliki konfiguracji kompilacji *Makefile*. Do kompilacji wykorzystano kompilator *GNU gcc* w wersji 4.6.3. Jako systemu testowego użyto dystrybucji *Elementary OS* (jest to pochodna dystrybucji *Ubuntu* z jądrem w wersji 3.2.0-74).

4.1. Ładowalny moduł jądra

Ładowalny moduł jądra został zaimplementowany zgodnie ze specyfikacją ładowalnych modułów jądra systemu Linux [4]. Dwie podstawowe, umieszczone w nim funkcje to *module_init* oraz *module_exit*. W realizowanym projekcie wykorzystano elementy kodu programowego dostępnego w [8], [9].

Funkcja *module_init* powoduje usunięcie modułu jądra z listy uruchomionych modułów, przyznanie dla modułu uprawnień administratora, wyszukanie adresu początku tablicy funkcji systemowych, podmienienie wybranych funkcji systemowych, uruchomienie procesu serwera SSH oraz wątku obsługi podsłuchu klawiatury, ukrycie odpowiednich plików, procesów oraz sieciowych kanałów transportowych wykorzystywanych przez *rootkit*.

Funkcja *module_exit* powoduje zatrzymanie procesu serwera SSH oraz wątku obsługi podsłuchu klawiatury, przywraca widoczność wszystkich plików, procesów, sieciowych kanałów transportowych, oraz oryginalne adresy funkcji systemowych.

4.1.1. Manipulowanie kodem funkcji systemowych

Uzyskanie adresu tablicy funkcji systemowych wymaga przeanalizowania mechanizmu funkcjonowania zarządcy wywołań funkcji systemowych. Procesy wywołujące funkcję systemową generują przerwanie programowe 0x80. Przeglądając tablicę wektorów przerwań można odnaleźć adres procedury obsługi tego przerwania i rozpocząć jej 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 1. Wynik deasemblacji kodu obsługi przerwania programowego 0x80

Analizując rozkazy w ramach procedury obsługi przerwania 0x80 (kod 1) można stwierdzić, że do najważniejszych wykonywanych operacji można zaliczyć:

- zapis wartości rejestrów procesora na stosie (rozkazy *push*),
- kopiowanie rejestrów (rozkazy *mov*),
- skoki warunkowe (rozkazy *jae*, *jne*),
- wywołanie procedury spod konkretnego adresu, z przesunięciem o wartość rejestru EAX (rozkaz *call*).

Szczególnie interesująca jest ostatnia wymieniona operacja. Jest to wywołanie procedury konkretnej funkcji systemowej. Jej numer jest zawarty w rejestrze EAX. Zatem widoczny adres jest adresem tablicy funkcji systemowych.

Przedstawiony schemat postępowania jest bardzo popularny. Wykorzystuje go m.in. funkcja *find_sys_call_table*, która umożliwia odnalezienie adresu tablicy funkcji systemowych [9], [10].

W prezentowanym projekcie zdecydowano się jednak na użycie innej metody podmiany funkcji systemowych. Jak stwierdzono powyżej, zaprezentowana metoda jest bardzo popularna i wiele prostych programów wykrywających *rootkity* jest w stanie ją zdemaskować.

Wykorzystano metodę, którą można by określić, jako „podmianę w miejscu wywołania". Zamiast zmieniać adres funkcji systemowej w tablicy funkcji systemowych, co jest łatwo wykrywalne, modyfikowany jest początek kodu oryginalnej funkcji w taki sposób, aby wywołanie zostało zrealizowane przez spreparowany kod [11].

Ze względów bezpieczeństwa, zapis do pamięci obszaru jądra jest zablokowany, więc nie była by możliwa modyfikacja kodu funkcji systemowych. Aby można było tego dokonać, niezbędne jest wcześniejsze wyłączenie wspomnianej blokady poprzez zmianę wartości rejestru CR0 procesora. W tym celu wykorzystuje się dwie funkcje: *disable_wp* służącą do wyłączenia blokady oraz *restore_wp*, przywracającą blokadę [10].

Podmiana kodu funkcji systemowych jest realizowana przy pomocy zestawu opracowanych funkcji [9], [10]. Najważniejsze z nich, to:

- *Hijack_start* – funkcja podmienia pierwsze 6 bajtów kodu oryginalnej funkcji systemowej, wprowadzając do nich kod rozkazów *push $addr* oraz *ret*, gdzie *$addr* jest adresem spreparowanej funkcji systemowej. Dzięki takiemu zabiegowi, sterowanie zostanie przekazane do funkcji wskazywanej przez *$addr*. Oryginalny kod, adres początku wywołania oraz podmieniany kod są zapamiętywane.
- *Hijack_stop* – funkcja przywraca pierwsze 6 bajtów oryginalnego kodu funkcji systemowej, do stanu pierwotnego.

Jak już powiedziano, zaletą zastosowanego rozwiązania jest to, iż nie zostaje w żaden sposób zmodyfikowana tablica funkcji systemowych. Programy służące do detekcji *rootkitów* w systemie, bardzo często porównują zawartość tablicy funkcji systemowych umieszczonej w pamięci z oryginalną, zapisaną na dysku lokalnym, w katalogu */boot*. W przypadku zaproponowanego rozwiązania nie wykryją one żadnych różnic i nie spowodują alarmu.

Dodatkową kwestią, na którą należy zwrócić uwagę jest wielowątkowość systemu. Może zdarzyć się sytuacja, w której nastąpi przełączenie zadania podczas podmiany funkcji systemowej, na jakieś inne zadanie, również wywołujące daną funkcję systemową. W takiej sytuacji wystąpiłby błąd jądra systemu. Aby temu zapobiec należy zapewnić czasowe blokowanie przełączania zadań procesora przy pomocy funkcji systemowej *preempt_disable*.

4.1.2. Nadawanie uprawnień administratora

Do przyznawania procesom uprawnień administratora (*root*), wykorzystano mechanizm zarządzania poświadczeniami systemu Linux. Przyznanie uprawnień sprowadza się do przypisania procesowi, wartości identyfikatora użytkownika oraz wartości identyfikatora grupy, odpowiadających identyfikatorom grupy administratorów (w systemach Linux jest to 0 [12].

4.1.3. Ukrywanie plików i procesów

Wypisanie listy plików zawartych w określonym katalogu realizuje się za pomocą polecenia *ls*. Wypisanie listy procesów aktualnie uruchomionych w systemie realizuje się za pomocą polecenia *ps.* Korzystając z programu *strace* [13] można przeprowadzić analizę wywołań funkcji systemowych wykorzystywanych przez programy *ls* oraz *ps*. Uzyskane wyniki wskazują, że każda operacja listowania plików czy procesów wymaga iteracyjnego uruchomienia funkcji systemowych operujących na wirtualnym systemie plików VFS (*virtual filesystem*). Funkcja systemowa odwołująca się do wirtualnego systemu pliku wywołuje z kolei szereg innych funkcji, w zależności od tego, jakiego systemu plików dotyczą działania (partycja dyskowa, system procesów *procfs* czy inny).

Począwszy od jądra systemu Linux w wersji 2.6.31, adresy wykorzystywanych funkcji *filldir* czy *readdir* można pozyskać ze struktury *file_operations* związanej z każdym węzłem informacyjnym pliku (*i-node*). W strukturze tej zapisane są wskaźniki do funkcji powiązanych z danym typem pliku (plik zwyczajny, plik katalogu, *procfs*) [7].

Po utworzeniu list przechowujących nazwy ukrywanych plików i katalogów (*hidden_files*) oraz identyfikatorów PID ukrywanych procesów (*hidden_procs*), możliwe stało się ukrywanie wskazanych plików, katalogów i procesów.

Zadanie to jest wykonywane przez zmodyfikowane wersje funkcji systemowych: *proc_iterate*, *root_iterate*, *proc_filldir*, *root_filldir* [7], [9], [10].

4.1.4. Pozyskiwanie identyfikatorów procesów z przestrzeni użytkownika

Przestrzenie jądra oraz użytkownika są odseparowane od siebie i działają w innym kręgu (pierścieniu) uprawnień. Przestrzeń jądra to pierścień 0, a przestrzeń użytkownika – pierścień 3. Wobec tego nie jest możliwe powoływanie nowych procesów w przestrzeni użytkownika, z poziomu jądra, w sposób znany z przestrzeni użytkownika. Aby uruchomić proces użytkownika z poziomu jądra systemu, należy posłużyć się specjalnym interfejsem *usermode_helper,* a konkretnie wchodzącą w jego skład funkcją *call_usermodehelper* [12]. Wspomniany interfejs posiada jednak, z punktu widzenia konstruowania *rootkitów,* pewną wadę. Nie udostępnia informacji o identyfikatorze PID uaktywnionego procesu. Struktura *subprocess_info,* opisująca proces nie zawiera pola identyfikatora procesu (PID).

Wobec tego wykorzystano zmodyfikowaną wersję struktury *subprocess_info*. Rozszerzono implementację funkcji systemowych *call_usermode_helper* oraz *call_usermode_helper_setup* wprowadzając

kopiowanie identyfikatora PID do zmiennej statycznej w obszarze pamięci *rootkita*. Adresy tych funkcji należało odnaleźć w pliku */proc/kallsyms,* wykorzystując funkcję *kallsyms_lookup_name*. Zwraca ona adres funkcji na podstawie jej nazwy użytej w wywołaniu. Dzięki temu możliwe było podmienienie oryginalnych funkcji systemowych. Zostało to następnie wykorzystane przy powoływaniu procesów serwerów użytkowych (SSH, VNC). Informacja o identyfikatorach tych procesów jest niezbędna do ich ukrycia.

4.1.5. Ukrywanie kanałów transportowych

Informacje o kanałach transportowych (połączeniach sieciowych) można pozyskać za pomocą polecenia *netstat* [14]. Wykorzystując wspomniany wcześniej program *strace*, można uzyskać informację o tym, jakie funkcje systemowe wykorzystywane są przez program *netstat*. Okazuje się, że wszystkie informacje odczytywane są z plików w wirtualnym systemie plików: /*proc/net/tcp, /proc/net/tcp6, /proc/net/udp* oraz /*proc/net/udp6*. Sposób jest podobny do tego, jaki miał miejsce w przypadku odczytu procesów, plików lub katalogów [15].

Utworzono cztery listy, które przechowują numery ukrywanych portów, odpowiednio dla protokołów TCP i UDP, korzystających z adresowania IPv4 jak i IPv6. Wykorzystano również zmodyfikowane funkcje systemowe *tcp4_seq_show, tcp6_seq_show, udp4_seq_show, udp6_seq_show* [9], [10].

Do zarządzania listami ukrytych portów utworzono po cztery funkcje typu *hide* oraz *unhide*, które dodają lub usuwają numery portów z konkretnej listy związanej z typem kanału transportowego.

4.2. Podsłuch klawiatury

Do implementacji modułu podsłuchu klawiatury wykorzystano wbudowany w jądro systemu Linux, mechanizm *keyboard_notifier*. Wobec tego nie jest wymagana żadna podmiana funkcji systemowych, a jedynie rejestracja własnej procedury obsługi zdarzeń generowanych przez klawiaturę.

W funkcji inicjującej podsłuch klawiatury tworzony jest nowy wątek, którego zadaniem jest cykliczne zapisywanie bufora odczytanych klawiszy do pliku dziennika. W funkcji inicjowania, realizowana jest również rejestracja procedury obsługi zdarzeń klawiatury.

Procedura obsługi zdarzeń powodowanych przez klawiaturę ustala, jakiego typu zdarzenie wystąpiło. Naciskanie klawiszy przez użytkownika powoduje zdarzenie typu KBD_KEYSYM. Wówczas następuje sprawdzenie, jakiego rodzaju klawisz został wciśnięty (odczyt bitów 8÷12). W zależności od

typu klawisza uruchamiane są dalsze funkcje dekodujące kod wciśniętego klawisza. Ostatecznie kod odczytanego klawisza zapisywany jest do bufora, a po zapełnieniu bufora do pliku dziennika.

4.3. Aplikacja IOCTL

Mechanizm IOCTL jest szeroko stosowanym mechanizmem komunikacji między aplikacjami użytkownika a urządzeniami (sterownikami) jądra systemu [4]. Dzięki temu mechanizmowi możliwe jest np. sterowanie konfiguracją urządzeń sieciowych. Realizacja takich działań wymaga zdefiniowania odpowiedniego gniazda, które pozwoli na przesyłanie danych pomiędzy aplikacją a urządzeniem (komendy wraz z parametrami).

Ze względu na to, iż *rootkit* nie powinien zostawiać śladów swojej egzystencji w systemie, wykorzystano mechanizm IOCTL w nieco odmienny sposób niż w przypadkach typowych.

Wszystkie pakiety IOCTL są odbierane. W wyniku podmiany kodu funkcji systemowej *inet_ioctl*, w treści odbieranych komunikatów, w polu polecenia, wyszukiwana jest unikalna wartość (0xDEADC0DE). Parametr tego „polecenia" określa, które z wymienionych niżej działań należy zrealizować:

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

Jeżeli zawartość pola polecenia w pakiecie IOCTL jest różna od wartości 0xDEADC0DE, uruchamiana zostaje oryginalna wersja funkcji systemowej *inet_ioctl*.

Aby wykorzystać zaimplementowane funkcje interfejsu IOCTL, opracowano również program klienta IOCTL. Klient umożliwia wysyłanie wszystkich komend obsługiwanych przez *rootkit* oraz pustej komendy testowej.

Aby utrudnić wykrycie klienta, on sam wysyła żądanie ukrycia własnego procesu. Program klienta IOCTL umożliwia, w ramach sesji SSH, sterowanie pracą *rootkita* (ukrywanie plików, procesów, połączeń).

4.4. Serwer SSH

SSH (*Secure Shell*) to protokół komunikacyjny, który umożliwia m.in. uruchomienie zdalnej sesji terminalowej, kopiowanie plików czy też tunelowanie ruchu sieciowego innych usług. Protokół ten udostępnia mechanizm uwierzytelnienia wykorzystujący kryptografię asymetryczną oraz szyfrowanie transmisji wykorzystujące szyfry symetryczne. Protokół ten jest szeroko stosowany w środowiskach pracujących pod kontrolą SO Linux. Np. prace administracyjne są realizowane poprzez zdalną sesję terminalową.

Jednym z celów każdego *rootkita* jest uzyskanie dostępu, z uprawnieniami administratora, do zasobów na komputerze ofiary. W tym celu można wykorzystać wywoływanie komend, jako administrator (*root*), w czasie sesji SSH.

Do zaimplementowania serwera SSH użyto funkcji, udostępnianych w bibliotece *libssh*. Proces serwera zostaje ukryty w wyniku wysłania, przy użyciu interfejsu IOCTL, odpowiedniego żądania do *rootkita*.

Proces serwera SSH jest uruchamiany przez moduł główny, w wyniku użycia wspomnianego wcześniej (punkt 4.1.4) interfejsu programistycznego *usermode_helper*. Interfejs ten został zmodyfikowany w taki sposób, aby możliwe było pozyskanie identyfikatora PID procesu uruchomionego w przestrzeni użytkownika. Dodatkową zaletą uruchamiania procesu przez moduł główny jest to, iż uzyskuje on te same uprawnienia co moduł główny, czyli uprawnienia administratora systemu.

Długotrwałe pozostawianie serwera SSH w stanie aktywności, ułatwiałoby jego wykrycie. Dawałoby to również możliwość włamania się do systemu innej osobie niż „właściciel" *rootkita*. Z tego względu serwer jest uruchamiany tylko wtedy, gdy odebrany zostanie specjalnie spreparowany pakiet ICMP ECHO REQUEST. W sekcji danych, pakiet ten zawiera klucz autoryzacyjny (0xDEADC0DE).

Przechwytywanie pakietów ICMP zrealizowano poprzez użycie usługi *netfilter*. Podobnie jak wspomniana wcześniej usługa *keyboard_notifier* (punkt 4.2) pozwala ona zarejestrować własną funkcję obsługi zdarzeń określonego typu, w tym przypadku odebrania pakietu ICMP [16].

Funkcja, która realizuje obsługę zdarzenia polegającego na odebraniu pakietu ICMP, analizuje pakiety i wyszukuje te, które zawierają klucz autoryzacyjny. Nie może ona uruchomić procesu serwera SSH z poziomu jądra,

gdyż zakończyłoby się to wystąpieniem krytycznego błędu jądra systemu. Wobec tego należało powołać osobny watek jądra, odpowiedzialny za uruchomienie, ukrycie i zakończenie działania procesu serwera SSH.

Do wysyłania pakietu ICMP, którego odebranie powoduje uaktywnienia procesu serwera SSH, wykorzystany został skrypt napisany w języku Python.

4.5. Serwer VNC

VNC (*Virtual Network Computing*) to system przekazywania obrazu z wirtualnego lub fizycznego środowiska graficznego oraz sterowania tym środowiskiem. W projekcie zaimplementowano tylko część odpowiedzialną za przekazywanie obrazu, jako iż zdalne przesuwanie kursora myszy bądź wprowadzanie znaków za pośrednictwem klawiatury mogłoby wzbudzić podejrzenia u operatora zainfekowanego systemu.

Do implementacji wykorzystano bibliotekę *libVNCServer* oraz biblioteki obsługi serwera wyświetlania obrazu systemu Linux – X11 (X.Org). Ponieważ serwer X11 uruchamiany jest w środowisku konkretnego użytkownika i nie jest możliwe administrowanie nim 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 nie został wykryty, moduł główny ukrywa połączenia z nim związane (port 5900). Proces serwera VNC zostaje usunięty z listy widocznych procesów, w wyniku obsługi stosownego żądania IOCTL (ukrycie procesu o zadanym identyfikatorze).

5. Wyniki testowania

Opracowany *rootkit* testowano w wirtualizowanym środowisku systemu *Elementary OS*. Jest to jedna z wersji dystrybucji *Ubuntu*, który z kolei jest wersją dystrybucji systemu *Debian*. Wykorzystano jądro w wersji 3.2.0-74. Wszystkie biblioteki systemu oraz wymagane aplikacje przed rozpoczęciem testów zostały zaktualizowane do najnowszych wersji dostępnych w repozytoriach z oprogramowaniem. Jako pomocnicze środowisko testowe służące do wykonywania połączeń SSH oraz VNC posłużono się maszyną działającą pod kontrolą systemu Windows 8.1 Pro.

W celu zapewnienia możliwości śledzenia pracy *rootkita*, w szczególnie istotnych miejscach dodano wywołania funkcji *printk*, która zapisuje komunikaty do dziennika jądra. Może on być następnie odczytany przy pomocy polecenia *dmesg*. W niniejszym opracowaniu, ze względu na konieczność ograniczenia jego wielkości, przedstawiono jedynie wyniki wybranych testów.

```
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,
```

Rys. 3. Fragment dziennika jądra systemu z informacjami dotyczącymi testowanego modułu rootkita

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

Rys. 4. Wynik testowania funkcji ukrywania modułu rootkita

```
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:/$
```

Rys. 5. Zawartość katalogu głównego uzyskana przy pomocy polecenia *ls*, przed zrealizowaniem procedury ukrywania plików

Pierwszy test dotyczył poprawności ładowania modułu jądra. Należało odpowiedzieć na pytane: czy moduł jest ładowany prawidłowo, czy nie powoduje krytycznych błędów jądra oraz czy jest wyświetlany na liście załadowanych modułów systemowych. Wyniki potwierdzające poprawność ładowania przedstawia rysunek 3. Moduł nosi nazwę *rooty*. Na rysunku 4 można zauważyć, że nie jest on prezentowany na liście załadowanych modułów jądra systemu.

Kolejny test dotyczył funkcji ukrywania plików, katalogów oraz procesów. Należało sprawdzić, czy rzeczywiście pliki, katalogi oraz procesy nie są widoczne w dostępnych raportach. W celu ukrycia wybranych plików, katalogów i procesów użyto opisanego wcześniej interfejsu IOCTL zaimplementowanego w *rootkicie*.

Przebieg i skutki realizacji procedury ukrywania plików przedstawiono na rysunkach 5-7. Ukrywaniu podlegały katalogi /*usr*, /*home* oraz /*lib*. Należy zwrócić uwagę, że na rysunku 7 wspomniane wcześniej katalogi nie są widoczne.

```
test@test-virtual-machine:~$ ioctl
Usage: ioctl CMD [ARG]
        CMD     Description:
        ----    ------------
        0       Give root privilages
        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:~$
```

Rys. 6. Raport z realizacji procedury ukrywania plików (katalogów /usr, /home, /lib)

```
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-g
eneric-pae
test@test-virtual-machine:/$
```

Rys. 7. Zawartość katalogu głównego uzyskana przy pomocy polecenia *ls*, po zrealizowaniu procedury ukrywania plików

Do przetestowania poprawności ukrywania procesów wykorzystano aplikację przeglądarki internetowej Chrome. Przeglądarka ta tworzy dla każdej otwartej karty czy zainstalowanego rozszerzenia osobny proces. Prezentuje to rysunek 8.

```
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:~$
```

Rys. 8. Lista uruchomionych procesów przeglądarki Chrome

Do ukrywania procesów również użyto interfejsu IOCTL (rys. 9). Po zastosowaniu procedury ukrywania, żaden proces przeglądarki internetowej Chrome nie był widoczny (rys. 10).

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

Rys. 9. Raport z realizacji procedury ukrywania procesów przeglądarki Chrome

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

Rys. 10. Lista procesów po zrealizowaniu procedury ukrywania procesów przeglądarki Chrome

Kolejny test dotyczył funkcji ukrywania połączeń sieciowych. W tym przypadku również wykorzystano interfejs IOCTL. W systemie otwartych było kilka połączeń TCP na porcie 80 oraz 443 (rys. 11). Rysunek 12 przedstawia raport z przebiegu procedury ukrywania portów 80 i 443.

Połączenia sieciowe TCP związane z portem 80 i 443 zostały ukryte i nie są widoczne (rys. 13). Postępując analogicznie możliwe jest ukrycie każdego połączenia sieciowego bez względu na numer portu, rodzaj połączenia (TCP czy UDP) oraz sposób adresowania (IPv4 czy IPv6).

Kolejnym przeprowadzanym testem był test funkcji podsłuchu klawiszy na klawiaturze (*keylogger*). Jak wcześniej napisano, funkcja ta uruchamiana jest automatycznie w momencie ładowania modułu *rootkita* do pamięci. Informacja o wciskanych klawiszach zapisywana jest w ukrytym pliku dziennika wciśniętych klawiszy (*/.keylog*). Dostęp do niego ma jedynie użytkownik z prawami administratora. Rysunek 14

prezentuje zawartość dziennika wciśniętych klawiszy. Na jego podstawie można stwierdzić, że podsłuch klawiatury działa zgodnie z oczekiwaniami. Rejestrowane są wciśnięcia wszystkich przycisków klawiatury: zarówno alfanumerycznych jak i klawiszy kursorów czy klawiszy funkcyjnych.

```
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:/$
```

Rys. 11. Wykaz otwartych połączeń sieciowych uzyskany, przed zrealizowaniem procedury ukrywania połączeń sieciowych

Następnym etapem testowania było sprawdzenie poprawności funkcjonowania serwera SSH. Serwer SSH uruchamiany jest dopiero po otrzymaniu specjalnie spreparowanego pakietu ICMP ECHO REQUEST. Dlatego w tym przypadku została użyta druga maszyna, z której taki pakiet został wysłany. Zrealizowano to zostało za pomocą skryptu *ping.py* napisanego w języku Python (rys. 15). Następnie z tej samej zdalnej maszyny, wykonana została próba otwarcia zdalnej sesji terminalowej SSH. Sprawdzono również, czy proces serwera SSH oraz wszystkie procesy uruchamiane w ramach sesji terminalowej są ukrywane i nie figurują na liście aktywnych procesów.

```
test@test-virtual-machine:~$ ioctl
Usage: ioctl CMD [ARG]
        CMD     Description:
        ----    ------------
        0       Give root privilages
        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:~$
```

Rys. 12. Raport z realizacji procedury ukrywania połączeń sieciowych na portach 80 i 443

```
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:/$
```

Rys. 13. Wykaz otwartych połączeń sieciowych uzyskany, po zrealizowaniu procedury ukrywania połączeń sieciowych

Wysłany pakiet został prawidłowo przechwycony i proces serwera SSH został uruchomiony (rys. 16). Nie jest on widoczny na liście procesów (rys. 17). Nie są również widoczne połączenia sieciowe na porcie 22 (rys. 18), typowe dla nasłuchującego serwera SSH. Dokonano próby połączenia przy użyciu nazwy

logowania i hasła „zaszytego” w konfiguracji serwera SSH. Sesja terminalowa SSH użytkownika *root* została nawiązana, ale nie jest widoczna w liście procesów obserwowanych przez użytkownika (rys. 19).

```
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:/$
```

Rys. 14. Zawartość pliku dziennika /.keylog

```
D:\Dropbox\Studia\INŻ\rooty>python ping.py 192.168.233.154

D:\Dropbox\Studia\INŻ\rooty>
```

Rys. 15. Raport z uruchomienia skryptu *ping.py*

```
[ 1498.297314] rooty: IOCTL->Hiding PID 15490
[ 1498.297343] rooty: IOCTL->Hiding PID 12064
[ 1503.120387] rooty: IOCTL->Hiding PID 15809
[ 1503.120434] rooty: IOCTL->Hiding PID 12068
[ 1507.206213] rooty: IOCTL->Hiding PID 15810
[ 1507.206259] rooty: IOCTL->Hiding PID 12075
[ 2175.139434] rooty: IOCTL->Hiding PID 16994
[ 2190.698893] rooty: IOCTL->Hiding PID 16995
[ 2190.698925] rooty: IOCTL->Hiding TCPv4 port 80
[ 2198.398273] rooty: IOCTL->Hiding PID 16996
[ 2198.398303] rooty: IOCTL->Hiding TCPv4 port 443
[ 2945.470887] rooty: ICMP packet: payload_size=4, magic=deadc0de, ip=0, port=0
[ 2945.470914] rooty: Starting SSHD server
[ 2945.486281] rooty: SSHD pid 17121
test@test-virtual-machine:/$
```

Rys. 16. Raport prezentujący uruchomienie serwera SSH

```
test@test-virtual-machine:/$ ps -A | grep sshd
test@test-virtual-machine:/$
```

Rys. 17. Lista procesów po zrealizowaniu procedury uruchamiania serwera SSH

```
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:/$
```

Rys. 18. Lista aktywnych połączeń sieciowych po zrealizowaniu procedury uruchamiania serwera SSH

Dysponując aktywną sesją terminalową można było zdalnie uruchomić serwer VNC. Aby tego dokonać, należało sprawdzić, który użytkownik uruchomił serwer wyświetlania obrazu X.Org (X11) a następnie przy użyciu poświadczeń tego użytkownika uruchomić serwer VNC.

Proces serwera X.org został uruchomiony w ramach sesji terminalowej *tty7*. Domyślnie skonfigurowany serwer X11 jest uruchamiany w ramach takiej właśnie sesji terminalowej. Z tego można wnioskować, że mamy do czynienia z konfiguracją standardową.

W następnym kroku należało sprawdzić, który użytkownik jest zalogowany w tej sesji terminalowej. Okazało się, że użytkownikiem tym jest *test*. Wobec tego, w kolejnym kroku zalogowano się na konto tego użytkownika. Jak widać nie było potrzebne podawanie hasła użytkownika. Ostatecznie można było uruchomić serwer VNC. Opisany tok postępowania został przedstawiony na rysunku 20.

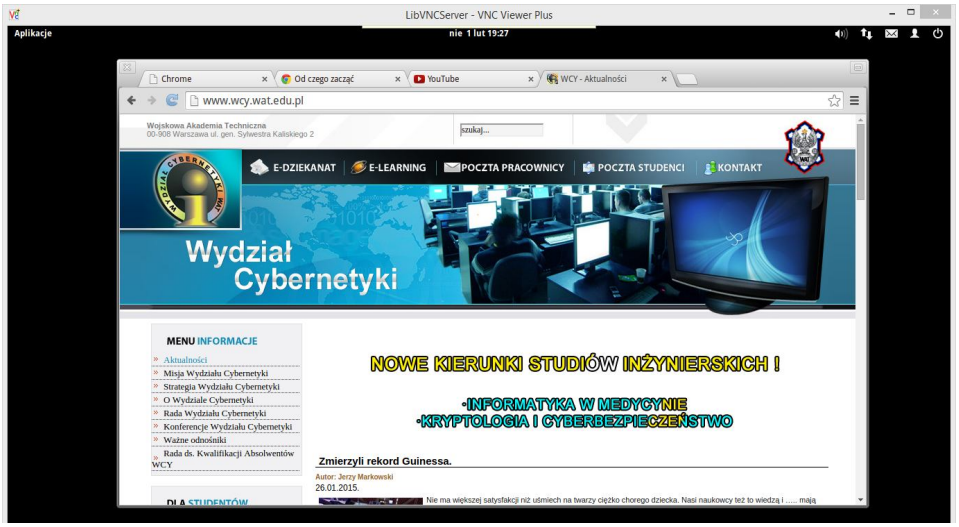
```
login as: root
root@192.168.233.154's password:
root@test-virtual-machine:/# uname -a
Linux test-virtual-machine 3.2.0-74-generic-pae #109-Ubuntu SMP Tue Dec 9 17:00:
00 UTC 2014 i686 i686 i386 GNU/Linux
root@test-virtual-machine:/# whoami
root
root@test-virtual-machine:/# who
test     tty7     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

Rys. 19. Raport (po stronie klienta) z przebiegu testowej sesji SSH

```
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
```

Rys. 20. Raport prezentujący uruchamianie serwera VNC w ramach sesji SSH

Bez problemów zestawiono połączenie protokołu VNC (rys. 21). Dodatkowo proces serwera VNC nie był widoczny dla użytkownika (rys. 22). Jest to powodowane wysłaniem przez niego żądania IOCTL ukrycia własnego procesu.



Rys. 21. Obraz ekranu przedstawiający udane połączenie VNC przy użyciu programu *VNC Viewer PLUS*

```
test@test-virtual-machine:/$ ps -A | grep vncd
test@test-virtual-machine:/$
```

Rys. 22. Lista aktywnych procesów po uruchomieniu serwera VNC

6. Podsumowanie

W artykule opisano projekt, którego celem było opracowanie pakietu *rootkit* dla systemu Linux. Przedstawiono charakterystykę najważniejszych mechanizmów stosowanych w tego typu konstrukcjach, koncepcję rozwiązania i sposób implementacji. Zamieszczono również wybrane wyniki testów opracowanego oprogramowania. Opracowany pakiet przewidziany jest do zastosowania w wybranych maszynach wirtualnych wykorzystywanych w specjalistycznych laboratoriach komputerowych Wydziału Cybernetyki. Stąd przymiotnik „dydaktyczny" w tytule artykułu.

Uzyskane wyniki pozwalają stwierdzić, że podstawowy cel został osiągnięty. Zbudowano platformę, która umożliwia „podglądanie" działań studentów, w czasie wykonywania ćwiczeń laboratoryjnych. Udostępnia również prowadzącemu zajęcia, narzędzie aktywnego oddziaływania na środowisko ćwiczącego studenta.

Jak stwierdzono we wstępie, istotne było również zapewnienie modyfikowalności rozumianej, jako możliwość praktycznie nieograniczonego rozszerzania właściwości funkcjonalnych. Ten cel również został osiągnięty.

W najbliższej przyszłości podjęte zostaną prace mające na celu wykorzystanie opracowanego narzędzia w dydaktyce. Po ich zakończeniu, wyniki zostaną opublikowane w postaci oddzielnego opracowania.

W artykule nie podjęto problematyki wykrywania zainstalowanego *rootkita*. Wynika to bezpośrednio z jego przeznaczenia. W środowisku laboratoryjnym, użytkownik (student) nie będzie miał możliwości zainstalowania oprogramowania wykrywającego *rootkity*. W czasie ćwiczeń właściwie jedynie wykorzystuje dostarczony mu system. Na niektórych zajęciach realizuje co prawda czynności związane z instalowaniem dodatkowego oprogramowania. Są one jednak bardzo ograniczone i dotyczą oprogramowania dostarczonego przez prowadzącego zajęcia.

Literatura

[1] BUNTEN A., *UNIX and Linux based Rootkits Techniques and Countermeasures*. 16th Annual First Conference on Computer Security Incident Handling, Budapest, 2004, https://www.researchgate.net/publication/28356110_UNIX_and_Linux_based_Kernel_Rootkits (dostęp 10.07.2015).

[2] MOLLOY D., *Linux Kernel Programming – Writing A Linux Kernel Module – Part 1: Introduction*. http://derekmolloy.ie/writing-a-linux-kernel-module-part-1-introduction (dostęp 14.05.2015).

[3] HENDERSON B., *Linux Loadable Kernel Module HOWTO*. http://www.tldp.org/HOWTO/pdf/Module-HOWTO.pdf, (dostęp 11.06.2015).

[4] RUBINI A., CORBET J., KROAH-HARTMAN G., *Linux Device Drivers*. O'Reilly Media Inc., Sebastopol, 2005.

[5] BAR M., *Linux System Calls*. Linux Journal 75/2000, http://www.linuxjournal.com/article/4048, (dostęp 13.06.2015).

[6] DRYSDALE D., *Anatomy of system call, part 2*. https://lwn.net/Articles/604515, (dostęp 21.04.2015).

[7] BORLAND T., *Modern Linux Rootkits*. http://turbochaos.blogspot.com/2013/09/linux-rootkits-101-1-of-3.html, (dostęp 23.06.2015).

[8] SKLYAROV I., *Programming Linux Hacker Tools Uncovered*. A-LIST LLC, Wayne, 2007.

[9] COPPOLA M., Repozytorium bloga Github. https://github.com/mncoppola/suterusu, (dostęp 23.03.2015).

[10] CELEBI K., *Mechanizmy typu rootkit w systemach Linux*. Praca dyplomowa, Wydział Cybernetyki, Wojskowa Akademia Techniczna, Warszawa, 2015.

[11] REDDY A., O'NEILL R., GARCIA D., R., *Rootkit Analytics*. http://www.rootkitanalytics.com/kernelland, (dostęp 28.03.2015).

[12] MCNALLY C., *maK_it: Linux Rootkit. Technical Specification*. Dublin City University, Dublin, 2014, http://r00tkit.me/maK_it-Linux-Rootkit.pdf, (dostęp 28.05.2015).

[13] *Linux man page – strace(1).* https://linux.die.net/man/1/strace, (dostęp 18.04.2015).

[14] *Linux man page – netstat(8).* https://linux.die.net/man/8/netstat, (dostęp 18.04.2015).

[15] CASE A., *MoVP 1.5 KBeast Rootkit, Detecting Hidden Modules, and sysfs*. https://volatility-labs.blogspot.com/2012/09/movp-15-kbeast-rootkit-detecting-hidden.html, (dostęp 15.05.2015).

[16] SCAMBRAY J., MCCLURE S., KURTZ G., *Hacking Exposed: Network Security Secrets & Solutions*. McGraw Hill, Berkeley, 2012.

The *rootkit* for didactic Linux system

ABSTRACT: The paper describes a project, whose aim was to develop a package rootkit for Linux. Characteristics of the most important mechanisms used in these types of structures, a solution concept, and a method of implementation are presented. The paper also contains selected results of tests of the developed software. The developed package is provided for the use in selected virtual machines that are used in specialized computer laboratories at the Faculty of Cybernetics.

KEYWORDS: security, Linux systems, rootkit, didactics.

Praca wpłynęła do redakcji: 16.09.2016 r.