Przegląd technik filtrowania dostępu do wywołań systemowych w Linuksie

Łukasz Sowa

Wydział Elektroniki i Technik Infomacyjnych
Politechnika Warszawska
L.Sowa@stud.elka.pw.edu.pl

Streszczenie Przedmiotem niniejszego artykułu jest dogłębna analiza zagadnienia kontroli dostępu do wywołań systemowych w systemie operacyjnym *Linux*. Artykuł ma na celu zapoznanie czytelnika z ogólną ideą zastosowania wyżej wspomnianej ochrony. Pierwsza część artykułu zawiera opis potencjalnych korzyści oraz konkretnych przypadków użycia, druga to przegląd istniejących implementacji mechanizmu ochrony usług jądra w *Linuksie*: *ptrace*, *systrace*, *Capabilities*, *seccomp*, *Audit syscall*, *syscalls cgroup*, *LD_PRELOAD*, *Chromium sandbox*, *Patch out* — ich krótkie opisy, zalety oraz wady. Publikacja kończy się podsumowaniem i wnioskami płynącymi z przeprowadzonego badania.

1 Wstęp

Wywołania systemowe w systemach operacyjnych z ochroną pamięci są zasadniczym sposobem interakcji procesów użytkownika z jądrem. Można zatem powiedzieć, że dla wykonującego się programu zbiór dostępnych wywołań definiuje jego możliwości. Naturalnie więc kontrola dostępu do systemowego ABI (*application binary interface*) — tzn. do wspomnianych wywołań systemowych jest istotnym zagadnieniem w tematyce zapewniania bezpieczeństwa. Problem ten jest powszechnie znany w literaturze jako *system call interposition* (co można tłumaczyć jako *przechwytywanie wywołań systemowych*) i doczekał się wielu opracowań [13][14][18][23][25]. Niestety, większość istniejących prac dotyczy systemów innych niż *Linux* — głównie rodziny *BSD*. Ochrona usług jądra w tak popularnym środowisku jak *Linux* jest zatem niezwykle interesującym przedmiotem badań. Wszelkie sukcesy na tym polu mogą znacznie przyczynić się do zwiększenia bezpieczeństwa tego systemu — a więc również do polepszenia jakości usług, które opierają się właśnie na *Linuksie*.

W pierwszej części niniejszego artykułu opisano korzyści płynące z zastosowania mechanizmu ochrony wywołań systemowych. Kolejny punkt wskazuje potencjalne zastosowania zabezpieczenia tego typu. Zasadniczą częścią publikacji jest część trzecia będąca wyczerpującym przeglądem dostępnych rozwiązań. W tej sekcji opisane zostały mechanizmy: *ptrace*, *systrace*, *Capabilities*, *seccomp*, *Audit syscall*, *syscalls cgroup*, *LD_PRELOAD*, *Chromium sandbox* oraz *Patch out*. Artykuł kończy się podsumowaniem i wnioskami z dokonanego przeglądu.

2 Korzyści

Zabezpieczając system komputerowy, zakłada się, iż jądro sytemu operacyjnego jest oprogramowaniem zaufanym — starannie przetestowanym, a więc pozbawionym błędów. Odmiennie, jako niezaufane, traktuje się natomiast aplikacje użytkownika działające wewnątrz systemu. Procesy usługowe są często bardzo złożone, zdarza się także, że są one niedbale napisane czy też niedokładnie zweryfikowane. Te cechy sprzyjają obecności dużej liczby błędów w takim oprogramowaniu, co pociąga za sobą zagrożenie atakiem bądź destabilizacją systemu. Z tego powodu, znaczną część wysiłku wkłada się w zabezpieczenie przed błędami aplikacji, a nie przed błędami jądra systemu operacyjnego.

Zastosowanie mechanizmu kontroli dostępu do systemowego ABI przynosi wiele korzyści. Ograniczenie zbioru dostępnych wywołań systemowych daje możliwość ścisłej kontroli nad tym, co dany proces może w systemie. Jądro systemu *Linux* udostępnia dużą liczbę (w wersji 3.6, dla architektury *x86-64* jest ich ponad 300) dobrze zdefiniowanych wywołań. Możliwe jest zatem łatwe określenie minimalnego, wystarczającego zbioru usług systemowych dla wskazanego procesu. Gdyby ziarnistość wywołań systemowych *Linuksa* była mała — tzn. ABI zawierałoby niewiele funkcji, które spełniałyby wiele ról — mechanizm kontroli dostępu do usług systemowych byłby nieefektywny, gdyż zezwolenie na pewną część zbioru wywołań, dawałoby o wiele więcej możliwości, niż rzeczywiście byłoby potrzebnych. Właściwe zastosowanie opisywanego mechanizmu zapewnia więc działanie w myśl zasady najmniejszego uprzywilejowania — mniej dostępnych funkcji pociąga za sobą mniej potencjalnych szkód po udanym ataku.

Jednym z najczęściej występujących błędów w aplikacjach użytkowych jest przepełnienie bufora (ang. *buffer overflow*). Wykorzystanie takiej podatności pozwala atakującemu zmodyfikować standardową, prawidłową ścieżkę wykonania programu — intruz może zmienić lub wstrzyknąć dodatkowy kod do wykonującego się programu. Atakujący ma więc możliwość wykonania dowolnych instrukcji z prawami użytkownika, do którego należy atakowany proces. Szkody wyrządzone na skutek istnienia błędów przepełnienia bufora mogą być więc większe, jeśli program wykonuje się z uprawnieniami *roota* lub ustawionym bitem *SUID*. Przykładami typowych aplikacji działających w wielu systemach, z wyżej wymienionymi uprawnieniami mogą być odpowiednio — *XServer* oraz *sendmail*. Zastosowanie mechanizmu kontroli dostępu do wywołań systemowych pozwala znacząco złagodzić skutki ataku tego typu. Odpowiednie wydzielenie dostępnych dla procesu usług może ograniczyć lub nawet uniemożliwić wyrządzenie jakichkolwiek szkód. Przykładowo, administrator może zabronić procesowi uruchamiania innych programów, bądź tworzenia procesów potomnych (wzbronienie dostępu do wywołań `fork()`, `vfork()`, `clone()`, `execve()`) — utrudnia to działania atakującego i uniemożliwia wykonanie ataku *DoS* (ang. *Denial of Service*) przy użyciu techniki *fork bomb*. Innym sensownym działaniem może być np. uniemożliwienie zmiany użytkownika (zabronienie `setuid()` i pokrewnych) czy powstrzymanie zmian praw dostępu i własności plików (wstrzymanie wywołań `chmod()`, `chown()` i ich odmian).

Kolejnym zastosowaniem mechanizmu kontroli dostępu do wywołań systemowych jest stworzenie dla procesu piaskownicy — tj. ograniczenia dostępnych usług do minimum (np. do `read()`, `write()` oraz `exit()`). Piaskownica przydaje się podczas uruchamiania kodu nieznanego bądź niepewnego pochodzenia i działania — np. kodu aplikacji webowych wewnątrz przeglądarki. Kod taki, dopóki nie zostanie dokładnie przeanalizowany i zaakceptowany, nie powinien mieć możliwości dokonywania żadnych zmian w systemie operacyjnym.

Dzięki opisywanemu mechanizmowi administrator systemu może globalnie, dla wszystkich procesów, zabronić dostępu do pewnej grupy usług systemowych. Niektóre wywołania są używane tylko w określonych warunkach, w innych natomiast mogą być szkodliwe. Przykładem takiego wywołania systemowego może być `ptrace()` służące do debugowania oraz śledzenia procesów. Usługa ta jest szeroko stosowana jako narzędzie programistyczne w środowiskach rozwojowych. Z drugiej strony, w środowisku produkcyjnym przydatność `ptrace()` jest znikoma, a bogate możliwości tego wywołania mogą posłużyć potencjalnemu intruzowi do skutecznego zakłócenia pracy innych procesów. Zastosowanie globalnych ograniczeń usług systemowych warto także rozważyć w sytuacjach, w których dane wywołanie jest nowym, niedojrzałym jeszcze elementem systemu operacyjnego. Niestety, pomimo skrupulatnego procesu testowania i przeglądów kodu *Linuksa*, zdarzały się sytuacje, w których wywołania systemowe zawierały błędy bezpieczeństwa [5]. Administrator systemu wymagającego wysokiej stabilności i bezpieczeństwa może więc całkowicie wyłączyć możliwość używania niesprawdzonych wywołań.

Mechanizm kontroli ABI daje bogate możliwości, a konkretne jego użycie zależy wyłącznie od specyfiki procesu, na który nakładane są ograniczenia. Dla procesów o niedużym stopniu złożenia oraz o dobrze zdefiniowanej funkcjonalności łatwo skonstruować politykę dostępu do wywołań systemowych i mechanizm ten może być z łatwością zastosowany. Z drugiej strony dla aplikacji o niezbyt klarownej strukturze i funkcjach, zastosowanie odpowiednich ograniczeń może być niewykonalne.

3 Zastosowania

W poprzednim punkcie przedstawiono rozległe korzyści płynące z zastosowania mechanizmu kontroli dostępu do systemowego ABI. W następnym kroku warto więc przeanalizować, w jaki sposób można je wykorzystać w konkretnych sytuacjach. Poniższe przypadki użycia stanowią jedynie przykład eksploatacji opisywanego mechanizmu, gdyż kompletny przegląd jest niemożliwy ze względu na mnogość potencjalnych zastosowań.

3.1 Kontenery procesów

Koncepcja kontenerów [6][17] zapewnia wzajemną izolację grup procesów. Pomimo wydzielenia wielu zasobów poszczególne kontenery, zgodnie z ideą, współdzielą to samo jądro — usługi systemowe stają się jedynym zasobem w pełni

współdzielonym. Zastosowanie mechanizmu ochrony wywołań systemowych dla kontenerów pozwala zatem wprowadzić kolejny poziom izolacji — izolację usług jądra. Stosowanie kontenerów pozwala chronić wzajemnie grupy procesów, natomiast użycie opisywanej techniki umożliwia dodatkowo ochronę systemu operacyjnego przed aplikacjami działającymi wewnątrz środowisk zwirtualizowanych.

Zastosowanie rozważanego mechanizmu w połączeniu z kontenerami ma jeszcze jedną zaletę — łatwość konfiguracji. Procesy działające wewnątrz kontenera spełniają zwykle podobną funkcję — są np. jednostkami wykonawczymi usług pewnego serwera. Dla procesów o podobnym działaniu łatwo stworzyć wspólną politykę dostępu do wywołań systemowych, którą potem można zastosować w całości, dla wszystkich zadań wewnątrz kontenera.

Dla zilustrowania użycia rozważmy dwa kontenery — w pierwszym z nich znajdują się procesy serwera pewnych usług, w drugim natomiast działają aplikacje wspomagające rozwój oprogramowania używane przez programistów. W pierwszym z wymienionych kontenerów z całą pewnością warto zabronić dostępu do niepotrzebnych i potencjalnie niebezpiecznych wywołań takich, jak opisywany wyżej `ptrace()`. Jednocześnie, procesy w drugim kontenerze będą często wykorzystywać śledzenie w celu debugowania i sprawdzania oprogramowania, a więc wywołanie `ptrace()` będzie wykorzystywane — nie powinno być zabronione. Testowanie nowych wersji oprogramowania może być potencjalnie niebezpieczne ze względu na błędy i omyłki — dla drugiego kontenera warto rozważyć zakazanie wywołań typu `setuid()`, aby przypadkowe wykonanie komend z uprawnieniami `roota` nie było możliwe. Zastosowanie mechanizmu ochrony ABI w połączeniu z kontenerami pozwala więc wzmocnić ochronę przed destabilizacją systemu oraz pośrednio zabezpiecza działające wewnątrz niego usługi.

3.2 Serwery

Serwery usług to typ oprogramowania niezwykle powszechnego na komputerach pracujących pod kontrolą systemu *Linux*. Wśród tego typu aplikacji można wyróżnić serwery o bardzo wąskiej specjalizacji np. serwery DNS, serwery czasu NTP, serwery FTP. Oprogramowanie tego typu jest bardzo często zaniedbywane przez administratorów maszyn serwerowych — głównie ze względu na to, iż kolejne wydania bardzo często nie dodają nowej funkcjonalności, a jedynie poprawiają wykryte błędy — nie generują więc widocznej wartości biznesowej. Z tego powodu nieaktualizowane serwery bardzo często są swego rodzaju „furtką” dla intruzów. Atakujący nie musi za bezpośredni cel obierać interesujących go elementów (często dobrze zabezpieczonych). Zamiast tego, aby dostać się do systemu może wykorzystać błędy istniejące w pobocznych usługach, a dopiero później wykonać zasadniczy atak, ponieważ przeprowadzony od środka powinien być stosunkowo łatwy.

Mechanizmy ochrony dostępu do wywołań systemowych doskonale nadają się do izolacji serwerów o wysokiej specjalizacji. Przykładowo podstawowy serwer DNS może być zaimplementowany przy użyciu jedynie trzech wywołań systemowych (`recvmsg()`, `sendmsg()`, oraz `write()`) [11] — może być więc ograniczony

tylko do trzech wymaganych usług jądra, stając się mało użytecznym dla atakującego. Mechanizm kontroli dostępu do ABI pozwala skutecznie wzmocnić ochronę przed atakami na te usługi, którym często poświęca się mniej uwagi podczas zabezpieczania. Jest to szczególnie ważne, gdyż jak mówi popularne stwierdzenie: system jest tak bezpieczny, jak bezpieczny jest jego najsłabszy element.

3.3 Obliczenia rozproszone

Wraz z rozwojem sieci komputerowych, a w szczególności Internetu, popularne stało się rozpraszanie obliczeń, które zamiast jednego komputera, wykonuje wiele maszyn, których sumaryczna moc jest często większa od pojedynczych superkomputerów. Wiele ośrodków badawczych stworzyło projekty (np. *SETI@home* [22], *LHC@home* [12]) pozwalające ofiarować część czasu procesora na potrzeby obliczeń naukowych.

Podobny, choć komercyjny, jest projekt *CPUShare* [2], który pozwala odsprzedać część mocy obliczeniowej procesora na rzecz jednostki, która ją wykupiła w serwisie. W związku z tym, że aplikacja kliencka ściąga i wykonuje na komputerze kod nieznanego pochodzenia, istnieją uzasadnione obawy, iż uruchamiany kod może być złośliwy i może służyć np. do tworzenia sieci typu *botnet*. Na potrzeby *CPUShare* zaprojektowano i zaimplementowano więc mechanizm ochrony usług systemowych — *seccomp* (opisany szerzej w 4.4), który ściśle ogranicza zbiór dostępnych wywołań. Dzięki takiemu zastosowaniu mechanizmu ochrony ABI obliczenia rozproszone mogą być w pełni bezpieczne dla komputerów klienckich.

3.4 Aplikacje uprzywilejowane

Wielokrotnie podkreślanym [11][19], lecz wciąż nierozwiązanym, problemem bezpieczeństwa w systemie *Linux* są aplikacje, które muszą być uruchamiane z prawami *roota*, takie jak np. *XServer*. Wysoka podatność na błędy obszernych aplikacji, a także oczywiste korzyści związane z ich przejęciem przez intruzów czynią je częstym celem ataków. Próbą zaradzenia powyższemu problemowi jest mechanizm *Capabilities* (zob. punkt 4.3). Istnieją jednak sytuacje, w których ziarnistość ograniczeń *Capabilities* jest zbyt mała. W takich przypadkach mechanizm ochrony wywołań systemowych daje możliwość uzupełnienia brakujących reguł. Zestawienie tych dwóch mechanizmów nadal nie jest rozwiązaniem idealnym, gdyż aplikacja taka wciąż działa z uprawnieniami o wiele wyższymi niż proces zwykłego użytkownika. Niestety obecnie, poza silną izolacją, nie istnieją alternatywne rozwiązania wyżej wymienionego problemu.

3.5 Sandboxing

Sandboxing jest techniką izolowania procesów, która zyskuje w ostatnim czasie coraz większą popularność. Niewielu programistów dostrzega jednak korzyści

z piaskownic budowanych na bazie mechanizmów ochrony wywołań systemowych — aplikacji wykorzystujących opisywany mechanizm jest bardzo mało.

Jednym z nielicznych wyjątków jest przeglądarka Chrome, która wykorzystuje autorski *sandbox*, opisany szerzej w punkcie 4.8. Część przeglądarki odpowiedzialna za renderowanie (renderer) składa się z wielu złożonych modułów takich jak np. parser czy interpreter, które wykonują duże ilości niezaufanego kodu nieznanego pochodzenia. Proces renderera jest więc wyizolowany — umieszczony w piaskownicy. Z tego powodu wykorzystanie istniejących w kodzie błędów nie umożliwia interakcji z systemem operacyjnym — wszelkie próby zostają przechwycone przez opisywany *sandbox*.

Jak widać, mechanizm kontroli dostępu do systemowego ABI może w skuteczny sposób zostać zastosowany w aplikacjach, które wykonują kod nieznanego pochodzenia. Od wielu lat, programy tego typu (m.in. *Adobe Reader*, *Adobe Flash*, *Microsoft Office*) są z powodzeniem wykorzystywane do ataków na systemy operacyjne właśnie z powodu niekontrolowanego wykonania niezaufanego kodu. Umieszczenie procesów wykonawczych w piaskownicy chroniącej usługi systemowe mogłoby znacząco utrudnić lub wręcz uniemożliwić skuteczny atak tego typu.

4 Istniejące rozwiązania

Tematyka ochrony wywołań systemowych doczekała się wielu opracowań teoretycznych oraz praktycznych implementacji. Wśród znanych inicjatyw badawczych warto wspomnieć o projektach takich jak: *Capsicum* [25], *Janus* [13], *Ostia* [14], *Paladin* [23]. Mnogość istniejących prac dotyczących opisywanego mechanizmu wskazuje, iż zagadnienie nie jest trywialne, a ponadto odgrywa istotną rolę w dziedzinie bezpieczeństwa systemów operacyjnych. Interesujące jest to, że większość implementacji stworzona została dla odmiany systemu *UNIX* — *BSD*. Należy przypuszczać, iż jest to skutek priorytetowego traktowania bezpieczeństwa w systemach tego typu.

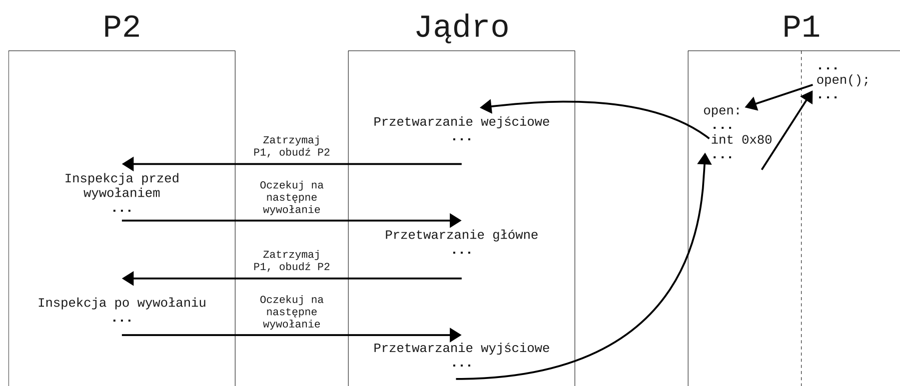
Dla systemu *Linux* również przygotowano wiele mechanizmów, które (często pośrednio) realizują ochronę dostępu do wywołań systemowych. Ze względu na bardzo dynamiczny rozwój tego systemu operacyjnego, nierzadko cechują się one innowacyjnością i różnorodnym podejściem do implementacji. Poniżej zaprezentowano zestawienie wspomnianych mechanizmów: ich sposób działania, zalety oraz wady. Należy wspomnieć, że opis ten nie jest całościowy, a zawiera jedynie popularne oraz aktywnie rozwijane pomysły.

4.1 ptrace

`Ptrace()` to wywołanie systemowe pozwalające śledzić (ang. *trace*) wykonanie wskazanego programu. Proces śledzący może sterować wykonaniem procesu śledzonego poprzez odczyt i modyfikację jego pamięci oraz rejestrów. Jedną z możliwości `ptrace()` jest przechwytywanie wywołań systemowych wykonywanych

przez proces, a zatem może ono służyć do implementacji mechanizmu kontroli dostępu do usług jądra.

Projekty (takie jak np. *SUBTERFUGUE* [3][16]) wykorzystujące `ptrace()` do implementacji ochrony wywołań systemowych, działają w architekturze z procesem nadzorującym. W tak skonstruowanym mechanizmie, proces–nadzorca podłącza się za pomocą `ptrace()` do procesu, który ma być śledzony. Od tego momentu kontrolowany proces w chwili przejścia w tryb jądra zostaje każdorazowo wstrzymany, a sterowanie jest przekazywane do procesu nadzorującego. W tej sytuacji nadzorca może odczytać stan procesu, a następnie, zgodnie z zaprogramowaną polityką dostępu, może podjąć odpowiednie działanie — np. zezwolić na kontynuowanie wykonania, gdy wywołanie jest dozwolone albo zakończyć proces, jeśli żądanie jest zabronione.



Rysunek 1. Poglądowy schemat architektury mechanizmu ochrony wywołań systemowych opartej na `ptrace()`. P1 — proces nadzorowany, P2 — proces–nadzorca. Na podstawie: [16].

Zastosowanie `ptrace()` do kontroli dostępu do wywołań systemowych pozwala na dużą elastyczność — śledzone mogą być tylko wybrane procesy, a dla każdego z nich zaprogramowana może być inna polityka ochrony. Ponadto zastosowanie tej techniki nie wymaga modyfikacji kodu programów, które mają być poddane kontroli.

Największą wadą mechanizmów skonstruowanych przy pomocy `ptrace()` jest bardzo wysoki narzut wydajnościowy. Opóźnienia w wykonaniu śledzonych programów wynikają z dwóch czynników: czasu potrzebnego na transfer sterowania pomiędzy procesem–nadzorcą a procesem podlegającym kontroli oraz z czasu wymaganego do sprawdzenia polityki dostępu. Pierwszy, w zależności od tego jak często kontrolowana aplikacja wywołuje usługi jądra, może powodować nawet stukrotny narzut na czas wykonania [26]. Drugi ze składników, dla prostych reguł jest natomiast zaniedbywalnie mały.

Kolejną wadą wywołania `ptrace()` jest niewygodny dla programisty interfejs oraz liczne ograniczenia z nim związane. Niektóre problemy związane z `ptrace()` mają zostać usunięte wraz z wprowadzeniem do jądra modyfikacji *utrace* [4].

4.2 systrace

Systrace to modyfikacja jądra *Linuksa* przygotowana specjalnie w celu kontroli dostępu do wywołań systemowych [18]. Mechanizm ten, poza *Linuksem*, został zaimplementowany dla systemów z rodziny *BSD* oraz *Mac OS X*. *Systrace*, podobnie jak rozwiązania oparte na wywołaniu `ptrace()`, korzysta z dodatkowego procesu — demona przechowującego politykę dostępu. Opisywany mechanizm nie wykorzystuje jednak śledzenia procesów — w niskopoziomowym kodzie przestrzeni jądra dodano sprawdzanie uprawnień procesu do wywołań. W celu minimalizacji narzutu wydajnościowego *systrace* stosuje podejście hybrydowe — proste zasady dostępu typu „zawsze zezwalaj” oraz „zawsze zabraniaj” są przechowywane w pamięci jądra i ich sprawdzenia nie muszą być konsultowane z demonem. Bardziej skomplikowane zasady są z kolei przesyłane do demona (komunikacja odbywa się za pomocą pseudourządzenia `/dev/systrace`) w celu weryfikacji.

Do zalet *systrace*, oprócz eleganckiej architektury, należy zaliczyć możliwość automatycznej generacji (wyuczenia) polityki oraz dodatkowe funkcje takie jak tymczasowe nadawanie uprawnień *roota*, które jest rozszerzeniem idei mechanizmu *Capabilities* [18]. Pomimo tego *systrace* nigdy nie zyskał znacznej popularności i nie został włączony do standardowego jądra *Linuksa*. Do takiego stanu przyczyniły się odkryte w *systrace* błędy typu *time-of-check-to-time-of-use* (*TOCTOU*) [24], których naprawienie jest trudne i znacząco komplikuje architekturę mechanizmu. Wykorzystanie błędów *TOCTOU* polega na zmianie zawartości pamięci pomiędzy zdarzeniami sprawdzenia warunku i użycia rezultatu tej operacji. W przypadku mechanizmów ochrony wywołań systemowych wykorzystanie takiego błędu pozwala na ominięcie sprawdzania polityki dostępu. Kolejnym istotnym mankamentem *systrace* jest wydajność. Zgodnie z badaniami [18][26], w zależności od specyfiki nadzorowanego programu, narzut wydajnościowy może wynosić od kilku, do nawet 30%. W zastosowaniach wymagających wysokiej responsywności jest to efektywność niezadowalająca. Na koniec należy zaznaczyć, że ostatnia wersja *systrace* pochodzi z 2009 roku i nieznane są dalsze losy projektu.

4.3 Capabilities

Capabilities pośrednio są mechanizmem ochrony dostępu do wywołań systemowych. Ustawiane dla procesów możliwości sprowadzają się w praktyce do zezwalania na pewne wywołania systemowe i wzbraniania ich. Specyfika tej techniki polega jednak na tym, iż są to wywołania systemowe dostępne jedynie dla *roota*. Mechanizmu tego można więc używać jedynie do ograniczania wywołań

systemowych dla procesów wykonujących się z uprawnieniami superużytkownika. Kolejnym istotnym problemem bywa niewystarczająca ziarnistość — danej możliwości nie zawsze odpowiada pojedyncze wywołanie systemowe.

Capabilities są dojrzałym elementem jądra systemu *Linux*, a ich użycie jest bardzo proste i nie powoduje żadnych narzutów wydajnościowych. Pomimo tego, mechanizm ten nie cieszy się dużą popularnością — najprawdopodobniej wynika to z ignorancji bądź przyzwyczajeń administratorów systemu *Linux*.

4.4 seccomp

Mechanizm *seccomp* to narzędzie do filtrowania wywołań systemowych pierwotnie zaprojektowane na potrzeby projektu *CPUShare* (zob. punkt 3.3). W podstawowej wersji (*mode 1*) użycie tego modułu pozwala na umieszczenie procesu w środowisku zezwalającym jedynie na wywołania `exit()`, `sigreturn()`, `read()`, `write()`, a próba wykonania jakiegokolwiek innego wywołania systemowego powoduje zakończenie procesu za pomocą sygnału *SIGKILL*.

Po kilku latach od włączenia *seccomp* do głównej linii *Linuksa*, rozpoczęto prace nad rozszerzeniem omawianego mechanizmu [7]. Ostatecznie, nowy tryb pracy (*mode 2*) został dodany do *Linuksa* w wersji 3.5. Najważniejszym usprawnieniem w tej wersji jest zwiększona ziarnistość — *mode 2* pozwala na filtrowanie dowolnego podzbioru dostępnych usług jądra, a także na wyszczególnienie dozwolonych argumentów dla poszczególnych wywołań systemowych. Dodatkowo rozszerzono możliwość reagowania na naruszenie zdefiniowanej polityki dostępu o opcje pozwalające np. na raportowanie próby wywołania usługi czy anulowanie żądania. *Seccomp* w nowej wersji oferuje zatem funkcjonalność analogiczną do innych, rozbudowanych mechanizmów takich jak *systrace*.

Niezależnie od trybu pracy mechanizm aktywowany jest za pomocą wywołania `prctl()` i działanie to jest nieodwracalne — opuszczenie stworzonego przez proces środowiska jest niemożliwe. W przypadku chęci zainicjowania trybu rozszerzonego konieczne jest również przekazanie struktury definiującej politykę dostępu. W *seccomp mode 2* wywołania systemowe potraktowano analogicznie do pakietów sieciowych — reguły filtrowania definiuje się za pomocą mechanizmu *BPF* (*Berkeley Packet Filter*) znanego z warstwy sieciowej.

Pierwotna wersja modułu nie zyskała znaczącej popularności. Oprócz wspomnianego projektu *CPUShare* posłużył programistom przeglądarki *Google Chrome* do stworzenia dedykowanej dla tego produktu piaskownicy (zob. punkt 4.8). Inne użycia *seccomp* nie były znane. Po dodaniu *mode 2 seccomp* szybko stał się najpopularniejszym mechanizmem filtrowania dostępu do wywołań systemowych i jest obecnie używany w wielu projektach m.in. *OpenSSH* czy *vsftpd*.

Do zalet mechanizmu należy zaliczyć możliwość definiowania ograniczeń na poziomie argumentów dla poszczególnych wywołań, a także zastosowanie znanych i sprawdzonych rozwiązań (baza *seccomp mode 1* oraz *BPF*). Niestety, do implementacji rozwiązania użyto wolnej (śledzonej) ścieżki wywołań systemowych, co powoduje zauważalny narzut wydajnościowy. Dodatkowo, użycie *BPF* do definiowania polityki dostępu może okazać się trudne dla początkujących

użytkowników, jednak ten problem został częściowo rozwiązany poprzez stworzenie dodatkowych bibliotek ułatwiających konfiguracje takich jak *libseccomp* [8], a także rozszerzenie istniejących narzędzi o wsparcie dla *seccomp* (np. w *systemd*).

4.5 Audit syscall

Audit syscall to rejestrator wywołań systemowych będący częścią *Linux Auditing System* — modułu jądra rejestrującego wszystkie istotne (z punktu widzenia bezpieczeństwa systemu) zdarzenia. *Audit syscall* jest narzędziem pozwalającym jedynie odnotowywać (za pomocą demona działającego w trybie użytkownika) fakt wykonania wywołania systemowego — moduł nie umożliwia podjęcia żadnych akcji. W celu wykorzystania informacji dostarczanych przez ten system często łączy się go z implementacjami mechanizmu *MAC* — najczęściej z *SELinux*.

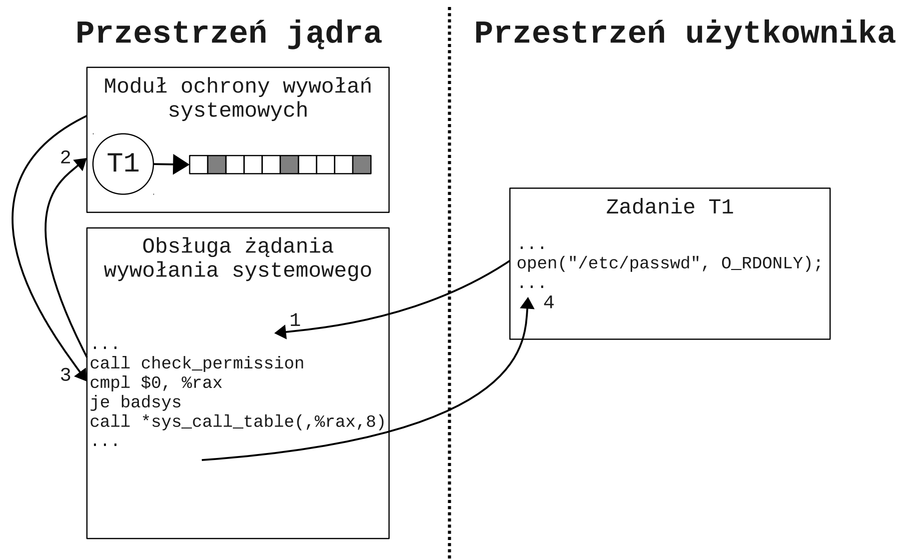
Połączenie *Audit syscall* z *SELinux* pozwala skonfigurować politykę reakcji systemu na wykonane wywołania. Stosowanie opisywanych narzędzi nie umożliwia zatem zabronienia czy też przerwania wywołania systemowego. W kontekście ochrony wywołań systemowych jest to istotna wada tego zestawienia — skutki działania niektórych usług jądra, takich jak np. `unlink()`, czy `truncate()` nie są łatwe do odwrócenia — administratorowi bardziej zależy na wstrzymaniu dostępu do nich niż na niwelowaniu efektów. *Audit syscall* jest wydajnym mechanizmem, który nie powoduje narzutu wydajnościowego dla procesów, dla których audyt nie został skonfigurowany. Istotną jego wadą jest jednak dosyć skomplikowana konfiguracja oparta o tworzenie reguł, która dodatkowo komplikuje się, gdy zachodzi potrzeba współpracy z *SELinux*.

Z wyżej wymienionych powodów *Audit syscall* jest nieefektywnym mechanizmem ochrony wywołań systemowych. Z drugiej strony, gdy istotne jest jedynie rejestrowanie wykonanych usług jądra, mechanizm ten jest aktualnie najwydajniejszym i najbardziej niezawodnym narzędziem dla tej klasy problemów.

4.6 syscalls cgroup

Syscalls cgroup to mechanizm ochrony dostępu do wywołań systemowych oparty na popularnym podsystemie grup kontrolnych (ang. *control groups, cgroups*). Rozwiązanie to wykorzystuje hierarchiczną organizację procesów w grupach kontrolnych — z każdą grupą skojarzona jest wydajna struktura bitmapowa przechowująca informacje o dostępnych dla przynależnych zadań wywołaniach systemowych. Procesy mają dostęp wyłącznie do wywołań dozwolonych w ich grupie z uwzględnieniem uprawnień grupy nadrzędnej — żądanie usługi niedozwolonej kończy się zwróceniem standardowego błędu `ENOSYS`.

Zarządzanie mechanizmem odbywa się w sposób typowy dla *cgroups* — poprzez system plików. Nadawanie i odbieranie uprawnień polega na zapisaniu numeru wybranego wywołania systemowego odpowiednio do plików `syscalls.allow` albo `syscalls.deny`, które znajdują się w katalogu reprezentującym daną grupę kontrolną. Konfiguracja mechanizmu jest zatem bardzo prosta, wymaga jednak



Rysunek 2. Poglądowy schemat architektury *syscalls cgroup*. 1 — skok do kodu obsługi żądania wywołania systemowego, 2 — sprawdzenie uprawnień zadania do żądanego wywołania na podstawie bitmapy wywołań skojarzonej z grupą kontrolną, 3 — powrót z modułu, 4 — powrót do wykonania zadania. Na podstawie: [21].

interwencji administratora (superużytkownika), ponieważ proces nie ma możliwości modyfikowania swoich uprawnień. Dzięki zintegrowaniu rozwiązania z podsystemem *cgroups* doskonale sprawdza się ono przy organizacji kontenerów (zob. punkt 3.1), co należy zaliczyć jako istotną zaletę. Mechanizm *syscalls cgroup* ze względu na swoją budowę oferuje ziarnistość na poziomie pojedynczych wywołań systemowych. Nie ma możliwości filtrowania wywołań na podstawie przekazanych argumentów, jak np. w *seccomp mode 2*. Podjęcie takiej decyzji projektowej podyktowane było dążeniem do uzyskania możliwie najmniejszego narzutu wydajnościowego przy jednoczesnym zachowaniu dosyć szerokich możliwości. Można powiedzieć, że cel ten udało się osiągnąć — *syscalls cgroup* jest najwydajniejszym rozwiązaniem w swojej klasie. Szczegółowe badania tego zagadnienia, a także dokładny opis procesu projektowania i implementacji mechanizmu można odnaleźć w [21].

Pomimo tego, że *syscalls cgroup* został dostrzeżony w społeczności programistów *Linuksa* [10], nie został on włączony do głównej linii i funkcjonuje jedynie jako nieoficjalna łata na jądro. Zasadniczym powodem podjęcia takiej decyzji było wybranie mechanizmu *seccomp* jako oficjalnego rozwiązania problemu ochrony dostępu do ABI w *Linuksie* głównie ze względu na jego szersze możliwości.

4.7 LD_PRELOAD

LD_PRELOAD to zmienna środowiskowa pozwalająca na wymuszenie na konsolidatorze czasu uruchomienia (ang. *runtime linking process*) załadowania obiektu współdzielonego (ang. *shared object*), zanim załadowane zostaną jakiekolwiek inne biblioteki dynamiczne. Zastosowanie tej zmiennej umożliwia więc przesłonięcie dowolnych symboli bibliotek dołączanych przy starcie programu. Typowo jedną z bibliotek ładowanych przy uruchamianiu programu jest biblioteka standardowa języka C — *glibc*. Wykorzystując *LD_PRELOAD* można zatem stworzyć „opakowania” (ang. *wrapper functions*) funkcji (z *glibc*) realizujących wywołania systemowe. Funkcje opakowujące mogą zawierać kod, który pozwala na śledzenie i kontrolę dostępu do usług jądra, czyli implementację mechanizmu ochrony wywołań systemowych. Dla aplikacji z dostępnym kodem źródłowym możliwe jest również całkowite podmienienie biblioteki *glibc* już na etapie kompilacji, co pozwala na eliminację konieczności użycia zmiennej *LD_PRELOAD* przy uruchamianiu procesu. Technika ta jest wykorzystywana do implementacji mechanizmu ochrony wywołań systemowych w projektach takich jak *Plash* [20], czy *Ostia*.

Wykorzystanie *LD_PRELOAD* do implementacji mechanizmu ochrony ABI ma kilka istotnych zalet. Przede wszystkim rozwiązanie takie jest w pełni elastyczne. W zależności od konkretnych potrzeb kod funkcji opakowującej może po prostu zabraniać danego wywołania lub wręcz przeciwnie — decyzja o dopuszczeniu wykonania usług może być podjęta na podstawie złożonej polityki opartej np. na wartościach argumentów. Kolejną zaletą jest to, że narzut wydajnościowy tak skonstruowanego mechanizmu zależy wyłącznie od złożoności procesów decyzyjnych, a dla wywołań niemodyfikowanych jest on zerowy. Warto też zaznaczyć, iż zastosowanie *LD_PRELOAD* nie wymaga modyfikacji kodu aplikacji, która będzie kontrolowana.

Najpoważniejsza wada rozwiązania wynika wprost z uproszczonego podejścia do zagadnienia — zmieniane są wyłącznie wywołania biblioteczne, a nie kod jądra — mechanizm ten zatem w żaden sposób nie chroni przed niskopoziomowymi wywołaniami kodu systemowego (np. bezpośrednie użycie instrukcji *int 0x80*). Technika ta nie jest więc zabezpieczeniem przed atakami polegającymi na wstrzyknięciu kodu do działającej aplikacji, takimi jak przepełnienie bufora. Ładowanie kodu kontrolującego wyłącznie przy uruchomieniu uniemożliwia podłączenie mechanizmu do już działających aplikacji, niemożliwe jest również wyłącznie mechanizmu bez ponownego uruchamiania procesu. Ostatnią istotną wadą zastosowania zmiennej *LD_PRELOAD* jest to, że działa ona wyłącznie dla aplikacji, do których biblioteka *glibc* jest linkowana dynamicznie.

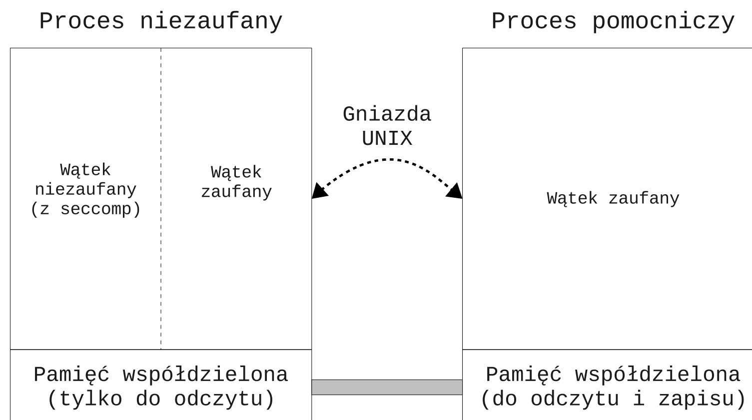
4.8 Chromium sandbox

Chromium sandbox [9] to piaskownica zaprojektowana specjalnie na potrzeby przeglądarki *Google Chrome* w celu izolacji procesów renderujących. Pomimo tak wyspecjalizowanego zastosowania, budowa tego mechanizmu jest na tyle generyczna, iż może on być z powodzeniem użyty do ochrony innych aplikacji —

na bazie *Chromium sandbox* powstało kilka projektów (np. *seccomp nurse* [1]) umożliwiających zastosowanie tego narzędzia z innymi programami.

Każdy proces znajdujący się w opisywanej piaskownicy posiada dwa wątki: niezaufany oraz zaufany. Wątek niezaufany wykonuje zasadnicze obliczenia i działa w trybie *seccomp* (zob. punkt 4.4) — zbiór bezpośrednio dostępnych wywołań systemowych jest bardzo wąski. Pozostałe usługi jądra mogą być wywoływane na rzecz wątku niezaufanego jedynie za pomocą zdalnych wywołań procedur RPC skierowanych do wątku zaufanego poprzez gniazda IPC. Zadaniem wątku zaufanego jest weryfikacja żądanych usług oraz ewentualne ich wykonanie. W tym miejscu warto zauważyć, że wątek zaufany działa w potencjalnie wrogim środowisku, gdyż współdzieli pamięć z wątkiem niezaufanym — w związku z tym została podjęta decyzja, aby wątek zaufany wykorzystywał jedynie rejestry procesora (które są chronione sprzętowo) i nie wykorzystywał wspólnej pamięci.

Jednym z istotnych problemów implementacji takiego rozwiązania jest zamiana wywołań systemowych na wywołania RPC. Zastosowanie podmienionej biblioteki jest niebezpieczne ze względów opisany w punkcie 4.7. Kolejną możliwością jest modyfikacja kodu procesu renderującego, lecz takie rozwiązanie jest wyjątkowo uciążliwe w utrzymaniu i konserwacji. Ostatecznie została więc podjęta decyzja o stworzeniu prostego deasemblera, który ma odczytywać binarny kod wątku niezaufanego i zamieniać wszelkie wywołania systemowe na wywołania RPC.



Rysunek 3. Poglądowy schemat architektury *Chromium sandbox*. Na podstawie: [1].

Ostatnim problemem implementacyjnym jest kwestia błędów typu *TOCTOU* (zob. punkt 4.2). Argumenty wywołań systemowych przekazywane poprzez wskaźniki mogą ulec zmianie w czasie pomiędzy sprawdzeniem dostępu a rzeczywistym wykonaniem usługi. W celu rozwiązania tego problemu dla wszystkich wątków mechanizmu renderującego tworzony jest pojedynczy, zaufany proces pomocniczy, który współdzieli kilka stron pamięci z zaufanymi wątkami. Wszelkie

wywołania systemowe, które nie mogą być zweryfikowane przez wątek zaufany — albo z powodu zbyt wysokiego stopnia komplikacji, albo z powodu odwołań do pamięci — są delegowane do procesu pomocniczego. Proces pomocniczy podczas obsługi żądania kopiuje wszystkie argumenty do swojej pamięci tak, by nie mogły być one zmienione przez niezaufany wątek. Wyniki pracy procesu pomocniczego są następnie umieszczane w pamięci współdzielonej do odczytu dla wątku zaufanego.

Chromium sandbox jest mechanizmem niezwykle wyrafinowanym, ale także bardzo złożonym, przez co może być trudny w zastosowaniu i rozwoju. Oczywistą wadą tego rozwiązania jest również niska wydajność. Mnogość zastosowanych mechanizmów IPC sprawia, iż rozwiązanie to jest jednym z najwolniejszych wśród wszystkich przeanalizowanych.

4.9 Patch out

Patch out to prymitywna technika polegająca na usunięciu z kodu źródłowego jądra fragmentów odpowiedzialnych za niechciane w systemie wywołania. Została ona zaproponowana, lecz jeszcze niewdrożona do systemu *Chrome OS*. Wedle założeń [15] z kodu jądra powinny zostać usunięte wywołania nieprzydatne (jak np. `ptrace()`) oraz takie, które uznawane są za niedojrzałe bądź niedostatecznie przetestowane (m.in. `vmsplice()`).

Istotną zaletą tego rozwiązania jest stuprocentowa skuteczność — kod, który nie istnieje, z całą pewnością nie zostanie wykonany. Oczywistą wadą opisywanego podejścia jest natomiast całkowity brak elastyczności — usunięcie wywołania dotyczy wszystkich procesów, a ponowne udostępnienie usługi wymaga rekompilacji jądra. Technika ta jest zatem bardzo kłopotliwa w zastosowaniu.

Prezentowane rozwiązanie można wykorzystać również w inny sposób. Wielokrotnie kompilując kod jądra z różnymi podzbiorami dostępnych wywołań, można wytworzyć wiele binarnych wersji systemu, które mogą odpowiadać różnym profilom wykonywanych aplikacji. Następnie, za pomocą projektu *User Mode Linux*, możliwe jest uruchomienie wspomnianych wielu wersji *Linuksa* jako procesów w systemie pełniącym rolę nadzorcy. W ten sposób uzyskuje się wiele zwirtualizowanych środowisk uruchomieniowych dla procesów. Podejście to, kosztem większej komplikacji zastosowania i trudności w konfiguracji, daje daleko większą elastyczność niż zastosowanie pojedynczego jądra z usuniętymi wywołaniami.

5 Podsumowanie i wnioski

Na podstawie przeprowadzonego badania widać, iż mechanizm kontroli dostępu do wywołań systemowych jest istotnym i uznanym środkiem zapewniania bezpieczeństwa systemów operacyjnych. Mnogość potencjalnych i już istniejących zastosowań wskazuje, że tematyka nie jest jeszcze wyczerpana, a w zagadnieniu tym tkwi potencjał.

Przeanalizowane w pracy implementacje prezentują odmienne podejścia do postawionego problemu — wynika to z tego, że projektanci poszczególnych rozwiązań różne właściwości uznawali za priorytetowe. Wśród cech, które mogą

podlegać ocenie w każdym z rozważanych przypadków, można wyróżnić: niezawodność, wydajność, konfigurowalność, łatwość użycia oraz prostotę budowy. Stworzenie mechanizmu idealnego — tj. takiego, który łączy i realizuje wszystkie powyższe postulaty — jest niemożliwe. Istnieje zatem konieczność dokonywania wyborów i pójścia na związane z nimi ustępstwa: rozwiązanie o bogatych możliwościach konfiguracji traci na wydajności, łatwość użycia często wyklucza prostą budowę, a techniki niezawodne często mają ograniczony obszar zastosowania.

Do niedawna problem ochrony wywołań systemowych w *Linuksie* pozostawał nierozwiązany [11] — nie istniał mechanizm, który spełniałby oczekiwania społeczności. Każdy projekt zainteresowany tego typu funkcjonalnością zmuszony był zatem stosować autorską kombinację różnych technik. Wydaje się, że wraz z publikacją jądra w wersji 3.5 zawierającego *seccomp mode 2* sytuacja ta się zmienia. Jest to pierwszy mechanizm tego typu, który szybko doczekał się szerokiego wdrożenia i jest w kręgu zainteresowania wielu autorów projektów. Należy jednak pamiętać, iż jest to rozwiązanie bardzo świeże, a co za tym idzie — niedojrzałe. Zanim uzna się *seccomp* za mechanizm w pełni wystarczający, trzeba najpierw uważnie śledzić rozwój tego modułu i dokładnie analizować komentarze ze strony aktywnych użytkowników. Trudno bowiem wydać słuszny i ostateczny werdykt w dziedzinie tak dynamicznej, a zarazem delikatnej, jak bezpieczeństwo.

Literatura

1. Nicolas Bareil. seccomp nurse homepage http://www.chdir.org/nico/seccomp-nurse/.
2. Caca Labs. CPUShare homepage http://caca.zoy.org/wiki/CPUShare.
3. Mike Coleman. SUBTERFUGUE homepage http://subterfugue.org/.
4. Jonathan Corbet. Introducing utrace. LWN.net, March 2007 http://lwn.net/Articles/224772/.
5. Jonathan Corbet. vmsplice(): the making of a local root exploit. LWN.net, February 2008 http://lwn.net/Articles/268783/.
6. Jonathan Corbet. Notes from a container. LWN.net, October 2007 http://lwn.net/Articles/256389/.
7. Jonathan Corbet. Yet another new approach to seccomp. LWN.net, January 2012 https://lwn.net/Articles/475043/.
8. Jake Edge. A library for seccomp filters. LWN.net, April 2012 https://lwn.net/Articles/494252/
9. Jake Edge. Google's Chromium sandbox. LWN.net, August 2009 http://lwn.net/Articles/347547/.
10. Jake Edge. Limiting system calls via control groups? LWN.net, October 2011 http://lwn.net/Articles/463683/.
11. Jake Edge. LSS: The kernel hardening roundtable. LWN.net, September 2011 http://lwn.net/Articles/458805/.
12. European Organization for Nuclear Research. LHC@home homepage http://lhcathome.web.cern.ch/LHCathome/.
13. Tal Garfinkel. Traps and pitfalls: Practical problems in in system call interposition based security tools. Proc. Network and Distributed Systems Security Symposium, February 2003 http://www.stanford.edu/~talg/papers/traps/traps-ndss03.pdf.
14. Tal Garfinkel, Ben Pfaff, Mendel Rosenblum. Ostia: A delegating architecture for secure system call interposition. Proc. Network and Distributed Systems Security Symposium, February 2004 http://benpfaff.org/papers/ostia.pdf.
15. Google Inc. Chromium OS Design Docs, 2011 http://dev.chromium.org/chromium-os/chromiumos-design-docs.
16. K. Jain, R. Sekar. User-Level Infrastructure for System Call Interposition: A Platform for Intrusion Detection and Confinement. In Proc. Network and Distributed Systems Security Symposium, 1999 http://seclab.cs.sunysb.edu/seclab/pubs/ndss00.pdf.
17. Paul B. Menage. Adding Generic Process Containers to the Linux Kernel http://www.kernel.org/doc/ols/2007/ols2007v2-pages-45-58.pdf.
18. Niels Provos. Improving Host Security with System Call Policies. In Proceedings of the 12th Usenix Security Symposium, strony 257–272, 2002 http://research.google.com/pubs/archive/33459.pdf.
19. Joanna Rutkowska. The Linux Security Circus: On GUI isolation. The Invisible Things Lab's blog, April 2011 http://theinvisiblethings.blogspot.com/2011/04/linux-security-circus-on-gui-isolation.html.
20. Mark Seaborn. Plash Wiki http://plash.beasts.org/wiki/.
21. Łukasz Sowa, Nowoczesne mechanizmy bezpieczeństwa w systemie Linux, 2012.
22. University of California, Berkeley. SETI@home homepage http://setiathome.berkeley.edu/.

23. Jeffrey A. Vaughan, Andrew D. Hilton. Paladin: Helping Programs Help Themselves with Internal System Call Interposition, 2010 http://www.cs.ucla.edu/~jeff/docs/paladin.pdf.
24. Robert N. M. Watson. Exploiting Concurrency Vulnerabilities in System Call Wrappers, 2007 http://www.watson.org/~robert/2007woot/2007usenixwoot-exploitingconcurrency.pdf.
25. Robert N. M. Watson, Jonathan Anderson, Ben Laurie. Capsicum: practical capabilities for UNIX. August 2010 http://www.cl.cam.ac.uk/research/security/capsicum/papers/2010usenix-security-capsicum-website.pdf.
26. Jörg Zinke. System call tracing overhead, October 2009 http://www.linux-kongress.org/2009/slides/system_call_tracing_overhead_joerg_zinke.pdf.