Uniwersytet Warszawski
Wydział Matematyki, Informatyki i Mechaniki

Maciej Wachulec
Nr albumu: 371845
Przemysław Jakub Kozłowski
Nr albumu: 371120

Tadeusz Dudkiewicz
Nr albumu: 370782
Wojciech Dubiel
Nr albumu: 371280

SIO2Jail
Narzędzie do nadzorowania wykonania programów zgłaszanych w ramach konkursów algorytmicznych

Praca licencjacka
na kierunku INFORMATYKA

Praca wykonana pod kierunkiem
dr. Robert Dąbrowski

Instytut Informatyki

Maj 2017

Oświadczenie kierującego pracą

Potwierdzam, że niniejsza praca została przygotowana pod moim kierunkiem i kwalifikuje się do przedstawienia jej w postępowaniu o nadanie tytułu zawodowego.

Data Podpis kierującego pracą

Oświadczenie autora (autorów) pracy

Świadom odpowiedzialności prawnej oświadczam, że niniejsza praca dyplomowa została napisana przeze mnie samodzielnie i nie zawiera treści uzyskanych w sposób niezgodny z obowiązującymi przepisami.

Oświadczam również, że przedstawiona praca nie była wcześniej przedmiotem procedur związanych z uzyskaniem tytułu zawodowego w wyższej uczelni.

Oświadczam ponadto, że niniejsza wersja pracy jest identyczna z załączoną wersją elektroniczną.

Data Podpisy autorów pracy

Streszczenie

Streszczenie jak już będziemy mieć całość pracy

Słowa kluczowe

Dziedzina pracy (kody wg programu Socrates-Erasmus)

11.3 Informatyka

Klasyfikacja tematyczna

Social and professional topics
D.127. Computing education
D.127.6. Student asessment

Spis treści

Wprowadzenie

Obecnie systemy Olimpiady Informatycznej, oraz platforma Szkopuł przyjmują co roku setki tysięcy zgłoszeń będących rozwiązaniami zadań programistycznych. Zgłoszone programy są automatycznie kompilowane, uruchamiane na odpowiednich dla zadania danych wejściowych i oceniane. Podstawą dla oceny programu jest jego czas działania, modelowany przez liczbę wykonanych instrukcji, co zapewnia powtarzalność i sprawiedliwość pomiaru. Dodatkowo, na uruchamiane programy nakłada się inne ograniczenia, np: rozmiar dostępnej pamięci, brak możliwości łączenia z siecią, tworzenia procesów potomnym, korzystyania z plików.

Na dzień dzisiejszy do sprawdzania rozwiązań Olimpiada Informatyczna wykorzystuje narzędzie oitimetool. Zostało ono stworzone ponad 6 lat temu przez Szymona Acedańskiego. Narzędzie to korzysta z biblioteki Pin firmy Intel, która "(...)przepisuje kawałki kodu maszynowego przed pierwszym ich wykonaniem, dodając w odpowiednie miejsca elementy instrumentacji, oraz w minimalnym możliwym stopniu modyfikuje oryginalne wywołania."[1]. Dzięki temu możliwe jest zliczanie wykonywanych instrukcji. W oitimetool zapewnienie izolacji ocenianego programu odbywa się poprzez ograniczanie dostępnych wywołań systemowych.

Narzędzie oitimetool nie jest konfigurowalne, zestaw obsługiwanych języków programowania jest mały i dodanie kolejnych wymaga dużego nakładu pracy, co ogranicza zestaw konkursów możliwych do zorganizowania. Ponadto programy uruchamiane pod kontrolą oitimetool dzielą z nim przestrzeń adresową, co jest zagrożeniem bezpieczeństwa i powoduje problemy administracyjne - trudno jest odróżnić błąd oitimetoola od błędu sprawdzanego programu.

W międzyczasie na rynku pojawiło się wiele nowych rozwiązań: technologia perf pozwalająca na sprzętowe liczenie instrukcji, oraz dużo mechanizmów pozwalających na izolację i ograniczanie uprawnień programów. W związku z tym możliwym stało się stworzenie nowego, zmodernizowanego narzędzia rozwiązującego część problemów z którymi boryka się oitimetool. Zaprojektowanie, zaimplementowanie i zintegrowanie z istniejącymi systemami Olimpiady Informatycznej alternatywy jest tematem niniejszej pracy.
Zapraszamy do lektury.

Rozdział 1

Dostępne technologie

1.1. perf

Nowoczesne procesory wyposażone są w sprzętowe liczniki zdarzeń występujących podczas wykonywania programu[2]. Dostępne są między innymi:

- liczba cykli podczas których procesor nie spał
- liczba wykonanych instrukcji
- liczba odwołań do pamięci
- liczba trafień i nietrafień w cache procesora, z podziałem na poziomy

Perf jest to API jądra Linuxa umożliwiającym odczytywanie tych liczników przez programy w przestrzeni użytkownika. Pozwala ono pobrać liczbę zdarzeń które zaszły w kontekście wybranego procesu, lub grupy procesów (zdefiniowanej za pomocą mechanizmu `cgroup`). Można odróżnić zdarzenia które wystąpiły gdy proces był w przestrzeni użytkownika, od tych które wystąpiły gdy był w przestrzeni jądra.

1.2. seccomp-bpf

Seccomp[3] to dostępny w Linuxie mechanizm filtrowania wywołań systemowych. Pozwala on przygotować i przekazać do jądra program zapisany w kodzie BPF[1], który decyduje o dozwolonych dla danego procesu wywołaniach systemowych. Filtr ten jest sprawdzany bardzo wcześnie podczas obsługi wywołań systemowych, jeszcze zanim sterowanie trafi do kodu odpowiedzialnego za konkretne wywołanie systemowe. Pozwala to ograniczyć interakcje wykonywanego programu z jądrem i w konsekwencji zmniejszyć powierzchnię ataku na jądro. Dodatkową zaletą seccompa jest jego szybkość, filtry wykonują się w całości w przestrzeni jądra, więc unikamy narzutu spowodowanego przełączeniem kontekstu.

1.3. ptrace

Jądro linuxa umożliwia śledzenie i ingerowanie w wykonanie programu. Proces śledzący ma możliwość zatrzymywania procesu śledzonego przy wystąpieniu różnych zdarzeń (m.in.

[1] BPF - Berkeley Packet Filter, jest rodzajem kodu bajtowego, pierwotnie służącego do filtrowania pakietów sieciowych. Przykładowo, do niego kompilują się filtry programu Wireshark.

rozpoczęcie wywołania systemowego, powrót z wywołania systemowego, wysłanie sygnału `TRAP`, zwrócenie odpowiedniej wartości przez filtr seccomp), pobierania danych z rejestrów i przestrzeni adresowej, oraz ich modyfikację. Ptrace jest używany przez debuggery (gdb) i nie służy do izolacji procesu.

1.4. namespaces

Namespaces to grupa mechanizmów izolacji w jądrze Linuxa pozwalających na wydzielenie niektórych zasobów jądra widocznych z punktu widzenia wybranej grupy procesów. Składa się ona z następujących mechanizmów:

mount namespaces pozwala zmienić dostępne dla procesu drzewo plików. Proces widzi inny katalog główny, oraz inne punkty montowania. Jest to rozwiązanie podobne do wywołania `chroot`, ale od niego bardziej elastyczne i bezpieczne.

PID namespaces pozwala odizolować widoczne drzewo procesów oraz ich numery. Procesy wewnątrz namespace'u nie widzą procesów z zewnątrz, nie mogą na nie oddziaływać i otrzymują identyfikatory z niezależnej puli od 1 wzwyż, jak gdyby były jedynymi procesami w systemie. Są one jednak widoczne dla procesów z zewnątrz, które mogą pobierać o nich informacje.

network namespaces pozwala prezentować procesom wewnątrz namespace'u inne interfejsy sieciowe niż te dostępne na zewnątrz. Można w ten sposób uzyskać efekt taki, jakby wnętrze namespace'u było osobnym hostem z punktu widzenia protokołów sieciowych, na przykład takim, który nie posiada żadnych interfejsów sieciowych.

UTS namespaces pozwala zaprezentować wybranym procesom inną nazwę hosta.

IPC namespaces pozwala oddzielić obiekty komunikacji międzyprocesowej System V oraz kolejki komunikatów POSIX widoczne dla wybranych procesów

user namespaces pozwala przedstawić wybranej grupie procesów inne identyfikatory użytkowników i grup. W szczególności użytkownik który utworzył namespace może otrzymać wewnątrz UID równy zero (być w nim użytkownikiem root). Otrzymuje on również pełne uprawnienia (`capabilities`) do zarządzania obiektami utworzonymi w ramach tego namespace'u. Na przykład nieuprzywilejowany użytkownik może utworzyć user namespace, w którym utworzy mount namespace. W nim będzie mu wolno montować systemy plików w dowolnym miejscu (jakby miał uprawnienia roota). Nie będzie to jednak miało wpływu na procesy na zewnątrz user namespace'u, oraz znajdujące się w innych mount namespace'ach. User namespace'y tworzą hierarchię drzewa, ukorzenioną w tzw. `init user namespace` utworzonym przez jądro systemu. Użytkownik root w tym namespac'ie jest "prawdziwym rootem", tzn. jego uprawnienia dotyczą wszystkich obiektów jądra, w tym wszystkich podrzędnych namespace'ów, oraz dostępu do prawdziwych urządzeń sprzętowych.

Wszystkie namespace'y oprócz user namespace'ów mogą być tworzone tylko przez procesy uprzywilejowane (wymagane jest `CAP_SYS_ADMIN`). Procesy mogą znaleźć się w namespac'ie jedynie dziedzicząc go po rodzicu, lub samemu do niego wchodząc. Wejście do już istniejącego namespace'u (a więc być może ucieczka z namespace'u obecnego do nadrzędnego) wymaga posiadania otwartego deskryptora pliku odnoszącego się do namespace'u docelowego. Dzięki temu proces tworzący namespace nie może złośliwie umieścić w nim innych już istniejących

procesów, a proces utworzony wewnątrz (bez deskryptora pliku odnoszącego się do namespace'u na zewnątrz) nie jest w stanie uciec na zewnątrz.

1.5. rlimit

Standard POSIX określa mechanizm ograniczania maksymalnego zużycia zasobów przez każdy proces. W Linuxie z jego pomocą można ustawiać górne ograniczenia między innymi na rozmiar przestrzeni adresowej, stosu, czy czas działania programu.

1.6. cgroup

Mechanizm cgroup w jądrze Linuxa pozwala grupować procesy i ograniczać rozmiar zużywanych przez nie zasobów (takich jak czas procesora, pamięć operacyjna, czy liczba operacji wejścia/wyjścia na sekundę). W przeciwieństwie do rlimit, procesy są łączone w grupy, a limity dotyczą sumarycznego zużycia zasobów przez wszystkie procesy w grupue. Podobnie jak w przypadku namespace'ów, przynależność do cgroup'y jest automatycznie dziedziczona przez procesy potomne, a same cgroup'y tworzą drzewo.

Rozdział 2

Implementacja

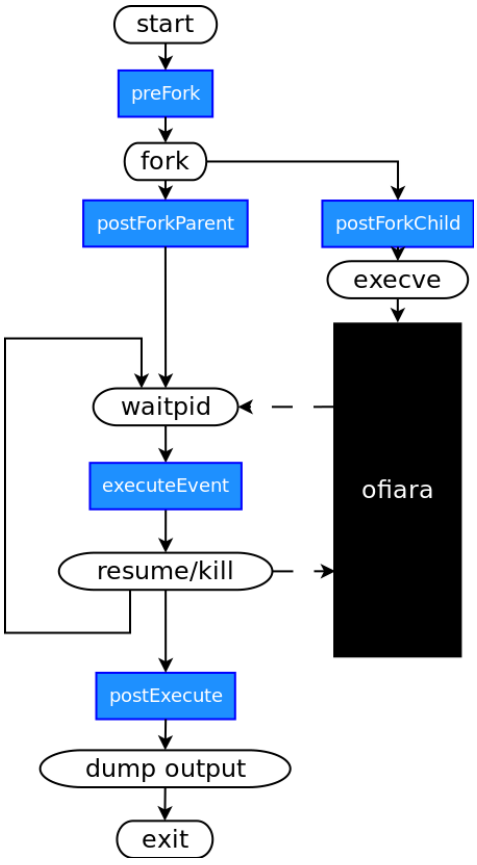
Do napisania narzędzia wybrany został język `C++`, ze względu na łatwość wykorzystania niskopoziomowego API jądra Linuxa, oraz możliwość wygodnego budowania abstrakcji.

2.1. Architektura

Działanie narzędzia dzieli się na kilka faz. W pierwszej z nich następuje przygotowanie środowiska do uruchomienia ofiary. Następnie program forkuje się. Proces potomny finalizuje przygotowanie środowiska, a następnie przekazuje sterowanie do programu ofiary (wywołanie `execve`). Proces rodzica również finalizuje przygotowanie środowiska, a następnie wchodzi w pętle obsługi zdarzeń z procesu potomnego. Po zakończeniu się procesu potomnego proces rodzica zbiera wyniki, wypisuje je i kończy swoje działanie.

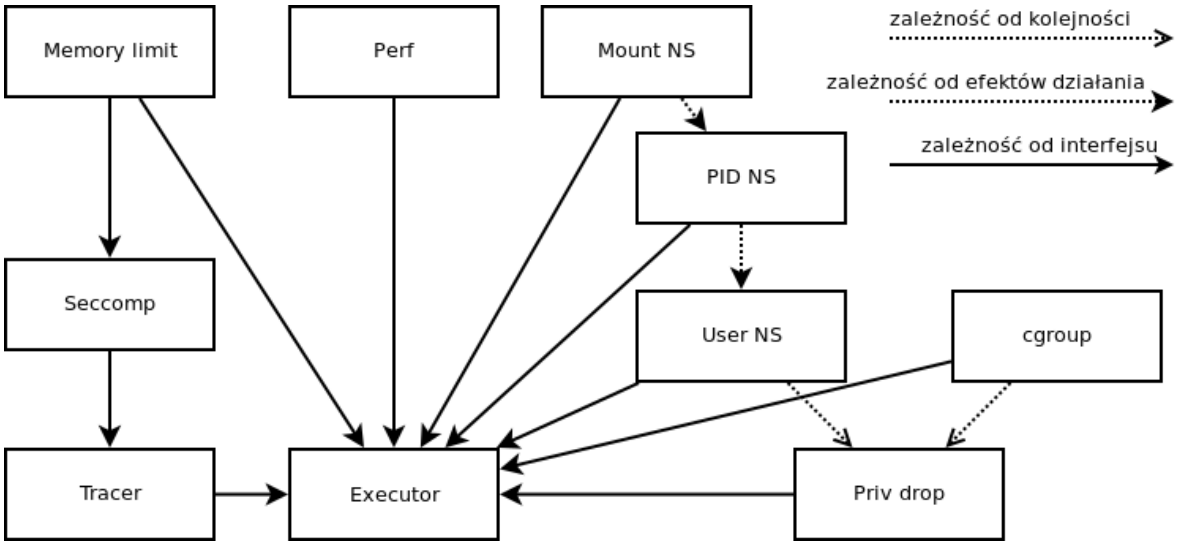
Ponieważ wykorzystane przez nas technologie wymagają wykonania różnych operacji w różnych etapach tego cyklu, wyróżniliśmy w nim kilka momentów i dla każdego stworzyliśmy listę funkcji, które mają się w nim wykonać. Każdy moduł programu definiuje które momenty go interesują i jakie funkcje chce w nich wykonać. W ten sposób uzyskaliśmy modularność i czytelność naszego rozwiązania[1]. Zdefiniowaliśmy:

- preFork – przygotowanie środowiska
- postForkChild – finalizacja środowiska w procesie potomnym
- postForkParent – finalizacja środowiska w procesie rodzica
- executeEvent – reagowanie na zdarzenia związane z procesem potomnym, w trakcie wykonywania się programu ofiary
- postExecute – zakończenie się procesu potomnego



Rysunek 2.1: Diagram sterowania

[1] Pisać takie rzeczy w tej pracy?



Rysunek 2.2: Diagram zależności

Dla każdej technologii wydzieliliśmy moduł (ptrace, perf, user namespace, pid namespace, mount namespace, seccomp, memory limit, time limit), powstał też moduł do wykonania całego cyklu i do zrzekania się uprawnień. Ponadto każdy moduł może dodawać informacje do danych wynikowych, które pod koniec wykonania programu są wypisywane w jednym z dostępnych formatów.

Moduły odpowiadające za ptrace i seccomp definiują własne zdarzenia wykonania programu, na które inne moduły mogą reagować:

- ptrace – opakowuje zdarzenie wykonania, rozszerzając je o dodatkowe informacje o procesie, umożliwia też ingerowanie w wykonywany proces, w tym, zakończanie jego działania
- seccomp – opakowuje te zdarzenia ptrace, które polegały na wykonaniu się wywołania systemowego, dodając informacje o tym wywołaniu

Pomiędzy modułami występują nietrywialne zależności. Kolejność w jakiej moduły wykonują swoje funkcje w danym momencie cyklu jest istotna.

	preFork	postForkChild	postForkParent	executeEvent	postExecute
memory limit		setlimit		read	
trace		traceme	ptrace init	zdarzenia trace	
perf		perf open			read stats
user NS	enter NS				
pid NS	enter NS				
mount NS		enter NS			cleanup
privilege drop		drop	drop		
seccomp	prepare filter	load filter		zdarzenia seccomp	

Rysunek 2.3: Etapy wykorzystywane przez moduły

2.2. Moduły

2.2.1. limit pamięci

Zużycie pamięci przez proces pobieramy z pliku `/proc/$pid/status` z wartości `vmPeak`. Jest to wartość, która odpowiada maksymalnemu rozmiarowi przestrzeni adresowej od początku działania programu. W celu ograniczenia zużycia pamięci przez program ustawiamy przy pomocy wywołania systemowego `rlimit` maksymalny rozmiar przestrzeni adresowej. Gdy proces przekroczy ten limit przez stos, otrzyma sygnał `SIGSEGV` i zostanie zabity, natomiast gdy zrobi to z pomocą wywołań `brk` albo `mmap` (z którego korzystają np, funkcja `malloc`) otrzyma błąd `ENOMEM` (funkcja `malloc` zwraca wtedy wartość `NULL`). Program użytkownika mógłby więc 'testować' limit pamięci, ukrywając ten fakt przed naszym narzędziem. Żeby temu zapobiec zdecydowaliśmy się:

- ustawiać twardy limit pamięci o pewną ilość x MB większy niż ten podany jako argument, w ten sposób jeżeli program przekroczył limit z pomocą małych alokacji (mniejszych niż x MB), zużycie pamięci większe niż limit zostanie zapisane w `vmPeak`. Wtedy po zakończeniu programu stwierdzimy niepoprawne wykonanie.
- z pomocą filtrów seccomp generować zdarzenia ptrace dla alokacji pamięci większych niż x MB. Takich alokacji program nie wykonuje dużo, więc narzut na ich obsługę nie jest odczuwalny. Narzędzie może sprawdzić aktualne zużycie pamięci i w razie próby przekroczenia limitu zakończyć działanie procesu potomnego.

2.2.2. trace

2.2.3. perf

2.2.4. user namespaces

Z modułów odpowiedzialnych za namespace-y jako pierwszy wykonuje się ten odpowiedzialny za user namespaces. Dzieje się tak, ponieważ w przypadku, gdy sio2jail jest uruchamiany jako użytkownik nieuprzywilejowany, utworzenie innych namespace-ów wymaga wcześniejszego utworzenia user namespace'a i uzyskania uprawnień pseudo-roota wewnątrz niego. Dlatego jest jako pierwszy na liście funkcji do wywołania przed forkiem (`preFork`)

Moduł ten najpierw tworzy nowy user namespace za pomocą wywołania `unshare(CLONE_NEWUSER)`, a następnie wypełnia pliki `/proc/self/uid_map`, `/proc/self/gid_map` i `/proc/self/setgroups`,

wpisując do nich odwzorowanie między identyfikatorami użytkowników i grup wewnątrz namespace'u, a tymi na zewnątrz. Użytkownikowi, jako który został uruchomiony sio2jail, przypisuje wewnątrz namespace'u UID zero, tj. roota . Do pliku `setgroups` zapisywany jest napis `deny`, co uniemożliwa zapis do `gid_map` kosztem zablokowania wywołania systemowego `setgroups`. (Jądro wymaga tego po to, aby nieuprzywilejowane procesy, tworząc user namespace, nie mogły zrzec się grup ze swojej listy grup suplementarnych. Jest to podyktowane tym, że bycie członkiem grupy bywa czasami używane jako sposób obniżenia uprawnień, a nie ich podniesienia.) Następnie odwzorowanie numerów grup jest wypełniane w sposób analogiczny do odwzorowania numerów uzytkowników.

2.2.5. pid namespaces

Moduł odpowiedzialny za PID namespace-y wykonuje wywołanie systemowe `unshare(CLONE_NEWPID)`. Niestety, zmiana PID namespace-u do którego należy istniejący proces nie jest możliwe, i powyższe wywołanie zmienia jedynie PID namespace procesów potomnych, które w przyszłości powstaną z obecnego procesu. Dlatego ten moduł wykonuje się przed forkiem (moment `preFork`), aby proces potomny utworzony przez fork był pierwszym procesem w nowym PID namespace-sie.

2.2.6. mount namespaces

Moduł odpowiedzialny za mount namespace-y w momencie inicjalizacji tworzy tymczasowy katalog, który stanie się katalogiem głównym dla procesu-ofiary.

Większość funkcjonalności tego modułu wykonuje się po forku, w procesie potomnym (moment `postForkChild`).

Najpierw wykonywane jest wywołanie systemowe `unshare(CLONE_NEWNS)`, które tworzy nowy mount namespace, i umieszcza w nim obecny proces. Następnie za pomocą wywołania `mount` z flagami `MS_PRIVATE|MS_REC` typ wszystkich punktów montowania jest rekurencyjnie zmieniany na prywatny. Wyłącza to propagację zmian punktów montowania z wewnątrz namespace-u na zewnątrz. Od tego momentu montowanie i odmontowywanie systemów plików wewnątrz namespace-u nie wymaga posiadania uprawnień na zewnątrz, ponieważ działania te nie wpływają na procesy poza obecnym namespacem.

Kolenymi krokami jest utworzenie mountpointu za pomocą bind-mounta z katalogu który ma się stać katalogiem głównym (`/`) z punktu widzenia programu-ofiary, oraz wykonanie dodatkowych bind-mountów w celu umieszczenia niezbędnych katalogów i plików wewnątrz tego katalogu, m.in. pliku wykonywalnego programu-ofiary. Potem wchodzimy do tego katalogu wywołaniem `chdir`, oraz robimy z niego katalog główny wywołaniem `pivot_root`. Zgodnie z dokumentacją wywołania `pivot_root` nie jest wiadomo, czy katalog główny obecnego procesu się zmieni, i konieczne jest wywołanie `chroot(.)` tuż po nim, co też robimy.

Jeśli włączona została opcja montowania procfs-u wewnątrz namespace-a, moduł tworzy katalog `/proc` a następnie montuje w nim procfs. Procfs daje dostęp do procesów z punktu widzenia tego PID namespace-u, w którym był proces, który go zamontował. Dlatego musi on zostać zamontowany z PID namespace-u, w którym chcemy uruchomić proces-ofiarę. A ponieważ proces potomny utworzony przez fork jest pierwszym procesem w naszym PID namespace-sie, moduł odpowiedzialny za mount namespace-y musi wykonać sie po forku.

Sposób w jaki użyliśmy `pivot_root` powoduje, że poprzedni katalog główny nadal jest zamontowany pod ścieżką `/`, w tym samym miejscu co nowy katalog główny. Odmontowujemy go za pomocą wywołania systemowego `umount("/", MNT_DETACH)`. Flaga `MNT_DETACH` jest

konieczna, ponieważ odmontowywany system plików nadal jest w użyciu poza obecnym mount namespacem.

Ostatnim krokiem jest przemontowanie nowego katalogu głównego jako read-only.

Ponadto, po zakończeniu działania programu-ofiary (`postExecute`), utworzony wcześniej katalog tymczasowy jest usuwany.

2.2.7. privilege drop

Po utworzeniu i przygotowaniu namespace-ów, nie potrzebujemy już uprawnień, które uzyskaliśmy tworząc user namespace. Z pewnością nie chcemy też, aby te uprawnienia miał program nadzorowany. Zrzeknięcie się tych uprawnień jest rolą modułu privilege drop.

Moduł ten wykonuje się po forku oraz wszystkich modułach odpowiedzalnych za namespace-y. Wykorzystuje do tego etapy `postForkChild` oraz `postForkParent`, aby zrzec się uprawnień zarówno w procesie potomnym jak i w rodzicu. W obu przypadkach robi to w ten sam sposób.

Pierwszym krokiem jest wyzerowanie wszystkich bitów tzw. capability bounding set, czyli maski bitowej ograniczającej które `capabilities` proces może uzyskać w przyszłości. Bitów tej maski po wyzerowaniu nie można spowrotem ustawić na 1. Zerowanie bitów tej maski wykonuje się za pomocą wywołania systemowego `prctl` z argumentem `PR_CAPBSET_DROP`, co wymaga przywileju `CAP_SETPCAP`.

Drugim krokiem jest ustawienie securebits wywołaniem systemowym `prctl` z argumentem `PR_SET_SECUREBITS`. To również wyamaga `CAP_SETPCAP`. Ustawiane są bity `SECBIT_NOROOT` oraz `SECBIT_NOROOT_LOCKED`. Pierwszy z nich powoduje, że w przyszłości jakakolwiek zmiana efektywnego UIDu tego procesu lub procesów potomnych z nie-zerowego na zerowy, ani wykonanie `execve` mając zerowy UID nie spowoduje otrzymania żadnych przywilejów (`capabilities`). Dotyczy to również zmiany wywołanej wykonaniem programu z flagą setuid. Ustawienie tego bitu jest konieczne jeśli korzystamy z user namespace'u i wewnątrz niego zarówno sio2jail jak i proces nadzorowany mają zerowy uid. Drugi z ustawianych bitów powoduje, że żadnego z tych dwóch bitów nie będzie można już zmienić.

W przypadku działania na kernelu 4.3 lub nowszym, dodatkowo ustawiane są bity `SECBIT_NO_CAP_AMBIEN` i `SECBIT_NO_CAP_AMBIENT_RAISE_LOCKED`. Pierwszy z nich uniemożliwia dodawanie przywilejów do zbioru ąmbient", który został wprowadzony w Linuxie 4.3. Teoretycznie, proces nie posiadający żadnych przywilejów w pozostałych zbiorach (konkretnie w inheritable i permitted) nie powinien móc nic dodać do zbioru ambient, ale zablokowanie tego secbitem może zmniejszyć powierzchnię ataku na jądro, a na pewno nie powinno zaszkodzić. Drugi z bitów, tak jak poprzednio uniemożliwia zmianę któregokolwiek z nich w przyszłości.

Potem moduł ustawia flagę `no_new_privs` za pomocą `prctl` z argumentem `PR_SET_NO_NEW_PRIVS`. Flagi tej po ustawieniu nie da się wyzerować. Powoduje ona, że proces, ani jego procesy potomne, nigdy nie otrzymają nowych uprawnień (w tym zmiany UIDu) w skutek wykonania `execve`. Umożliwa ona też korzystanie przez proces z pewnych mechanizmów w jądrze, które byłyby niebezpieczne gdyby `execve` podnosiło uprawnienia. Seccomp, którego używamy w innym module, jest jednym z tych mechanizmów.

Na koniec usuwane są wszystkie przywileje (`capabilities`) ze zbiorów inheritable, permitted i effective, za pomocą wywołania systemowego `capset`.

2.2.8. seccomp

Rozdział 3

Zarządzanie projektem

TODO

Rozdział 4

Testowanie

TODO

Rozdział 5

Wdrożenie

TODO

Rozdział 6

Podsumowanie

TODO

Bibliografia

[1] Szymon Acedański, *Wykorzystanie sprzętowych liczników zdarzeń do oceny wydajności algorytmów* (2009).

[2] Intel, *64 and IA-32 Architectures Software Developer's Manual June 2016*

[3] *Linux Kernel Documentation, Seccomp BPF (SECure COMPuting with filters)*