

Ochrona pamięci przy pomocy PaX i Stack Smashing Protector

I nastał pokój

PaX to poprawka dla jądra Linuksa zapewniająca nieco spokoju użytkownikom ceniącym sobie nade wszystko bezpieczeństwo swoich systemów. Peter Busser – programista Adamantix, wyjaśnia podstawowe zasady i zasadność umieszczenia poprawki PaX w tej dystrybucji.

PETER BUSSER

O systemach zabezpieczeń powiedziano już wiele, napisano też mnóstwo artykułów o sposobach zabezpieczania polegających na ograniczaniu dostępu procesów do obiektów systemowych (pliki, urządzenia, pamięć współdzielona, itd.). Generalnie zwiększa to integralność systemu i poufność danych oraz poprawia ogólną dostępność do usług. Wiele przykładów takich rozwiązań można znaleźć wśród poprawek dla jądra Linuksa i programów poziomu użytkownika. Jednym z przykładów jest RSBAC [3], dzięki któremu Adamantix może być uznany za bezpieczny system operacyjny.

O wiele mniej powiedziano do tej pory o ochronie samych procesów w Linuksie. Procesy są jedynym miejscem w całym systemie, gdzie wykonywany jest kod programów. Ponadto, w świecie Linuksa ignoruje się niestety (jest to pewnego rodzaju tradycja) problem ochrony pamięci, w której działają procesy i są przetwarzane dane. Dzięki temu jesteśmy świadkami powstawania wielkiej ilości exploitów, które w głównej mierze należy przypisać błędom programistów niedbale piszących kod zarządzania pamięcią w aplikacjach..



Dlaczego ochrona pamięci jest tak ważna?

W świecie doskonałym oprogramowanie nie zawierałoby żadnych błędów, więc nieautoryzowany dostęp do zapisu/odczytu wewnętrznej pamięci procesu nie byłby możliwy. Niektórzy twierdzą nawet, że moglibyśmy osiągnąć ową doskonałość, gdybyśmy zrezygnowali z programowania w języku C i zaczęli używać Java, C-LISP, Ada, czy dowolnego innego języka programowania, który wydaje się być niezawodny. Gdyby tylko życie było takie proste

Nawet jeżeli takie rozważania są prawdziwe (nie zapominajmy, że Java JDK, Perl, Python i inne języki programowania polegają w dużej mierze na translatorach, które zostały napisane w języku C, przez co stwarzają dodatkowe problemy dotyczące bezpieczeństwa) i zaczniemy zamieniać kod źródłowy z języka C na nasz krańcowo bezpieczny język XYZ, zajmie to i tak wiele lat, a do tego czasu wszystkie systemy będą i tak polegać

na podatnym na ataki kodzie w języku C.

Trzeba zatem przyjąć do wiadomości fakt, że świat nie jest doskonały i nawet najlepsi programiści popełniają czasem błędy. W rezultacie, wewnętrzna pamięć procesu może być narażona na uszkodzenie przez ataki z zewnątrz i jest to wystarczający powód, aby zapewnić jej właściwą ochronę.

Historia

Brak zainteresowania ochroną procesów oraz ogólne niezrozumienie jego znaczenia ma wiele przyczyn. Tradycyjne badania dotyczące bezpieczeństwa systemów skupiają się zwykle na kontroli dostępu i mechanizmach szyfrowania danych. Integralność przetwarzania nie została dostrzeżona jako poważny problem do momentu odkrycia błędów przepełnienia bufora wzrastająca ilość tych błędów nadała integralności przetwarzania nowy wymiar.

Przełomowym wydarzeniem w historii Linuksa była rezygnacja Linusa Torvaldsa

z projektu poprawki OpenWall [1]. Poprawka OpenWall uzbrajała jądro Linuksa w niewykonywalny stos. Stos jest jednak tylko jednym z wielu obszarów pamięci systemowej, który wymaga ochrony. Stanowisko Linusa w tej sprawie (ochrona stosu daje tylko częściowe bezpieczeństwo) było zatem zrozumiałe. Na nieszczęście wiele osób uwierzyło, że niepotrzebne są ŻADNE środki ochrony procesów. Takie myślenie jest równie błędne, co twierdzenie, że Ziemia jest płaska. Niestety taka opinia jest mocno rozpowszechniona i pozostaje żywa do dziś.

W czasach, kiedy komputery były naprawdę maszynami osobistymi, problem bezpieczeństwa w ogóle nie istniał. Skoro nie podłączamy naszego komputera do sieci Internet (lub innej dowolnej sieci) i chronimy nasz komputer przed dostępem przez osoby trzecie, nasze dane są w zasadzie bezpieczne. Wielu obecnych programistów dorastało w takim świecie. Wraz ze wzrostem popularności Internetu, sytuacja zmieniła się diametralnie. Do chwili obecnej dużo łatwiej było po prostu ignorować ten problem i mówić użytkownikom, że w zasadzie nic się nie dzieje. Niestety nie możemy dłużej tolerować takiej ignorancji.

PaX przejmie pałeczkę

Podjęto już wiele prób mających na celu stworzenie poprawki jądra lepiej zabezpieczającej pamięć. Na stronie głównej [2] projektu PaX znajdują się odnośniki do kilku z nich. Prace nad większością z tych projektów zostały już porzucone, pozostał w zasadzie wyłącznie PaX. Dzięki wytrwałości autora projektu PaX, użytkownicy Linuksa mogą teraz szczerzyć się najlepszą ochroną pamięci w całym świecie programistycznym.

Prace nad projektem PaX rozpoczęto około 30 miesięcy temu. Po szczegółowej analizie wielu prób ataków systemowych, autor doszedł do wniosku, że jedynym sposobem powstrzymania wielu z nich jest udoskonalenie zabezpieczeń pamięci. Do momentu wykorzystania PaX w poprawce gr-security, nie był on zbyt znany i rozpowszechniony. Gr-security jest bardzo popularna i wiele osób z niej korzysta.

Wiele dystrybucji, jak np. Gentoo, obsługuje gr-security, a więc pośrednio także korzysta z PaX. Spowodowało to znaczne przyspieszenie rozwoju w kierunku ochrony pamięci. Użytkownicy zaczęli wymagać funkcji raportowania błędów i pomocy w implementacji PaX w różnych konfiguracjach. Obecnie

jest to niewielka, ale niezwykle aktywna społeczność użytkowników.

Różnica pomiędzy PaX a innymi poprawkami chroniącymi pamięć jest taka, że PaX nie zabezpiecza przed określonymi atakami z zewnątrz został zaprojektowany bardziej jako ochrona przed pewną generalną klasą ataków, do których dochodzi z wnętrza systemu.

Załóżmy, że wymyślono płot chroniący nas i naszą ziemię przed dzikami. Pomysł całkiem dobry, gdy mamy uprawę, która jest często niszczona przez te zwierzęta. Jednakże ten sam płot nie może powstrzymać zarówno słoni (zbyt słaby) i niewielkich gryzoni (oczka w płocie są zbyt duże). Idealnym płotem chroniłby przed WSZELKIMI zwierzętami biegającymi po ziemi.

Poprawka PaX jest w świecie wolnego oprogramowania właśnie takim płotem. Zapewnia ochronę przed całą klasą ataków. Innymi słowy, chroni nasz przed WSZELKIMI zwierzętami biegającymi po ziemi. Nie chroni nas jednak przed ptakami i owadami. Inne rozwiązania chroniące pamięć mogą zabezpieczać nas przed konkretnymi gatunkami zwierząt. Można powiedzieć, że OpenWall chroni nas przed gryzoniami. Z kolei projekt OpenBSD o nazwie W[^]X [4] oraz exec-shield [5] będą chronić przed wszystkimi dziakami...

Różne klasy ataków

Mówiąc ogólnie, istnieją trzy grupy ataków, pod kątem których tworzone są poprawki wzmacniające ochronę pamięci:

- Wprowadzenie i wykonanie dowolnego kodu (1).
- Wykonanie istniejącego kodu w innej kolejności niż go opracowano (2).
- Wykonanie istniejącego kodu w prawidłowej kolejności, ale ze zmienionymi danymi (3).

Każde możliwe do wykorzystania przekłamanie w obszarze pamięci należy do jednej z tych trzech klas. Przykładowo, wiele popularnych błędów przepełnienia bufora należy do klasy (1). Przykład podany przez Linusa Torvaldsa wykorzystywał tzw. tech-



Rysunek 1. Wielu programistów zaczęło traktować system ochrony pamięci PaX bardziej poważnie. Obecnie jest on dołączany do pakietu GR-Security.

nikę return-to-libc. Należy ona do klasy (2). Błędy klasy (3) występują i są wykorzystywane bardzo rzadko. Po prostu dużo łatwiej jest skorzystać z klasy (1) lub (2).

Pamiętaj, że jest to klasyfikacja możliwych technik wykorzystania błędów, a nie samych błędów oprogramowania. Oznacza to, że daną techniką można wykorzystać wiele różnych błędów, a określony błąd w oprogramowaniu można wywołać przy pomocy kilku technik programistycznych.

Idea poprawki PaX jest ochrona przed całym klasami ataków. Obecnie trwają prace nad klasą (1), klasa (2) jest w przygotowaniu, a klasa (3) zostanie opracowana w najbliższej przyszłości. Cechą odróżniającą PaX od innych poprawek chroniących pamięć jest to, że nie radzą one sobie w pełni z klasą (1), ignorując często klasę (2) i (3).

Klasa (1)

Wprowadzenie i wykonanie dowolnego kodu oznacza, że można:

- zastąpić kod znajdujący się w pamięci innym kodem,
- zastąpić dane znajdujące w pamięci i wykonać je tak, jakby były oryginalnym kodem,
- załadować nowy kod do pamięci i wykonać go.

Jeżeli zatem można zastąpić kod oryginalny, intruz może wprowadzić swój kod do działającego procesu. Tak więc, zamiast wykonywać określone zadania, program będzie robił to, czego oczekuje od niego intruz.

Podobnie jest w przypadku drugiej techniki, gdzie nadpisuje się dane oryginalne, a następnie uruchamia się cały proces, tak jakby był to właściwy kod programu. Trzeba pamiętać, że system inaczej obsługuje kod wy-

konywalny i dane dla tego kodu, ale programiści wolą, dla swojej wygody, traktować kod i dane w ten sam sposób. Oczywiście jest to dla potencjalnych intruzów bardzo dogodnie. Technika ta wykorzystywana jest głównie do ataków przepełnienia bufora.

Dwie pierwsze techniki wymagają wyłącznie dostępu zapisu i możliwości wykonywania w pamięci. PaX radzi sobie z tymi technikami na swój własny sposób. Trzecia technika różni się w tym względzie, że wymaga także dostępu do plików. Podłączenie prądu do naszego płotu znacznie zwiększy jego efektywność. PaX zachowuje się podobnie. Przy pomocy list kontroli dostępu (ACL) i/lub innych mechanizmów kontroli dostępu (np. RSBAC [3]), PaX gwarantuje kompleksową ochronę przed wszystkimi odmianami ataków klasy (1). Jest to jedyna poprawka dla Linuksa chroniąca pamięć w ten sposób.

Klasa (2)

Do klasy (2) należy wcześniej przytoczony przykład Linusa Torvaldsa, rezygnującego z poprawki OpenWall. Ogólnie mówiąc oznacza to, że intruz zastępuje w pamięci adres wykorzystywany do kontroli pracy systemu. Przykładowo, zastąpienie adresu zwrotnego stosu w chwili wykonywania funkcji powrotu spowoduje powrót do miejsca wybranego przez intruza, a nie do miejsca oryginalnie ustalonego w kodzie programu.

Istnieje oczywiście wiele innych miejsc (służących różnym celom), gdzie przechowywane są adresy systemowe. Teoretycznie każde z tych miejsc może być wykorzystane przez potencjalnych intruzów, co umożliwi im uzyskanie wpływu na toczący się proces.

Klasa (3)

Do klasy tej należą ataki, w których istotne dane zastępowane są innymi, umieszczanymi przez potencjalnego intruza. Jeżeli intruz będzie w stanie w jakiś sposób zastąpić istotne dane, program znacznie wykonywać nieprawidłowe procedury i w rezultacie będzie robił nieoczekiwane rzeczy.

Weźmy dla przykładu po-

lecenie mount. Polecenie to można skonfigurować tak, aby umożliwić niektórym użytkownikom montowanie dysków. Uruchamia ono procedurę identyfikacyjną, rozróżniającą, którzy użytkownicy mogą montować dyski, a którzy nie. Jeżeli potencjalny intruz uzyska dostęp do tych procedur, polecenie mount weźmie intruza za autoryzowanego użytkownika i pozwoli mu zamontować dysk (który może być później użyty do kolejnych ataków). Jest to oczywiście czysto hipotetyczny przykład. Trudno jest znaleźć dobre przykłady z rzeczywistości, gdyż jest to klasa, którą najtrudniej wytopić i wykorzystać.

Połączone mechanizmy ochrony

Ochrona tylko przed jedną klasą ataków jest bardzo potrzebna, ale jest niezbyt efektywna. Linus wykorzystał w poprawce OpenWall przykład klasy (2) do ominięcia zabezpieczenia przed klasą ataków (1). Jest to reguła dla wszystkich opisanych tu klas możemy obejść zabezpieczenie przed jedną z klas, stosując połączenie kilku technik.

Użytkownicy często dostrzegają tylko jedną klasę ataków i zapominają o pozostałych. Co gorsza, rezygnują z ochrony przeciwko jednej z klas, gdyż do pełnej ochrony niezbędne są dodatkowe mechanizmy zabezpieczające ich przed atakami. Jest to błąd, który już wcześniej popełnił Linus Torvalds. Jediną właściwą drogą jest połączenie mechanizmów ochrony przeciwko wszystkim trzem grupom ataków.

Stack Smashing Protector (przełącznik ochrony stosu)

Jednym z powodów umieszczenia Stack Smashing Protector (SSP) w dystrybucji Adamantix była chęć połączenia mechanizmów obronnych chroniących przed różnymi klasami ataków. SSP [7] (znany także pod nazwą ProPolice) to poprawka dla kompilatora GCC zabezpieczająca system przed określoną grupą ataków klasy (1), (2) i (3), znanych pod nazwą błędów przepełnienia stosu.

Do osiągnięcia zamierzonych celów SSP wykorzystuje dwa mechanizmy:

- W chwili wykrycia potencjalnie niebezpiecznej funkcji dołącza „minę-pułapkę” do stosu (ponieważ niestety mechanizm detekcji nie jest jeszcze niezawodny).
- Zmienia kolejność lokalnych zmiennych w taki sposób, aby podejrzane zmienne znalazły się obok „miny-pułapki”, zwiększając w ten sposób prawdopodobieństwo wykrycia błędu.

Mina-pułapka często określana jest w żargonie mianem kanarka (w kopalniach kanarki wykrywały kiedyś obecność śmiertelniego tlenku węgla gaz zabijał kanarka zanim zabił górników). Kanarkiem w naszym przypadku jest losowa liczba umieszczona na stosie. Atak przepełnienia stosu zastępuje tę liczbę, która następnie jest wykrywana przez SSP przed wykonaniem kodu wykorzystującego ten błąd. Jeżeli SSP wykryje zastąpienie liczby losowej inną liczbą, dokonuje zapisu komunikatu w dzienniku systemu i przerywa pracę programu.

Taki sposób kontroli jest niezwykle kosztowny, jeśli chodzi o czas procesora, szczególnie przy używaniu niewielkich funkcji. SSP próbuje wykryć funkcje, które mogą być podatne na przepełnienie stosu i dokonuje kontroli tylko w zagrożonych elementach. Mechanizm detekcji nie jednak idealny, tak więc trzeba się liczyć z sytuacją, w której SSP nie doda funkcji kontrolnych tam, gdzie będą one faktycznie potrzebne, a umieszcza je w miejscu, w którym będą zupełnie zbędne.

SSP może być wykorzystany do kompilacji jądra, dołączając w ten sposób ochronę przepełnienia stosu do funk-

```

uwolf@nix: ~ - Konsole
Session Edit View Settings Help
PaXtest - Copyright(c) 2003 by Peter Buser <peter@adamantix.org>
Released under the GNU Public Licence version 2 or later

Executable anonymous mapping      : Killed
Executable bss                   : Killed
Executable data                  : Killed
Executable heap                  : Killed
Executable stack                 : Killed
Executable anonymous mapping (mprotect) : Killed
Executable bss (mprotect)       : Killed
Executable data (mprotect)      : Killed
Executable heap (mprotect)      : Killed
Executable shared library bss (mprotect) : Killed
Executable shared library data (mprotect) : Killed
Executable stack (mprotect)     : Killed
Anonymous mapping randomisation test : 16 bits (guessed)
Heap randomisation test (ET_EXEC) : 13 bits (guessed)
Heap randomisation test (ET_DYN)  : 25 bits (guessed)
Main executable randomisation test (ET_EXEC) : No randomisation
Main executable randomisation test (ET_DYN) : 17 bits (guessed)
Shared library randomisation test : 16 bits (guessed)
Stack randomisation test (SEGMEEXEC) : 23 bits (guessed)
Stack randomisation test (PAGEEXEC) : 23 bits (guessed)
Return to function (strcpy)       : Vulnerable
Return to function (strcpy, RANDEXEC) : Vulnerable
Return to function (memcpy)       : Vulnerable
Return to function (memcpy, RANDEXEC) : Vulnerable
Executable shared library bss     : Killed
Executable shared library data    : Killed
Writable text segments           : Killed
[uwolf@nix uwolf]$

```

Rysunek 2. Dane uzyskane z programu PaXtest pokazują, że jądro wzmocnione poprawką PaX jest niewrażliwe na większość ataków potencjalnych intruzów.

jonalności jądra. A dzięki optymalizacji wykonanej przez SSP, wydajność takiego jądra jest całkiem wysoka.

Istnieją także bardziej złożone techniki kompilacji, dzięki którym programy napisane w języku C stają się bezpieczniejsze (ang. full bounds checking). Kontrola tego typu oznacza sprawdzenie każdego dostępu do danych. Dzięki temu poziom bezpieczeństwa znacznie wzrasta – niestety kosztem innych czynników. Prędkość byłaby porównywalna z przetwarzaniem języka Java. Jest to znaczne pogorszenie wydajności systemu, na które niewiele osób może sobie pozwolić.

Randomizacja (losowość)

Jedną z cech poprawki PaX, i innych poprawek ochrony pamięci, jest randomizacja układu przestrzeni adresowej, czyli w skrócie ASLR (Address Space Layout Randomization). ASLR powoduje, że do różnych miejsc w pamięci systemu ładowane są różne części programu. Układ pamięci jest zmieniany przy każdym uruchomieniu programu.

Nie jest to mechanizm ochronny – nie zapewnia on żadnego mechanizmu kontroli. Dzięki niemu możemy jednak znacznie utrudnić wykorzystanie dziur w naszym systemie zabezpieczeń. Intruz będzie miał duże trudności z ustaleniem odpowiedniego układu pamięci. Poprawki tego typu różnią się między sobą pod względem ich stopnia randomizacji. Generalnie, im wyższy stopień randomizacji, tym trudniej wykonać brutalny atak na nasz system. Obecnie poprawka PaX zapewnia użytkownikom najlepszy dostępny stopień randomizacji w systemie Linux, a jednocześnie jest dużo lepsza niż w systemie OpenBSD.

Kompatybilność

Aby korzystać z naszych programów na jądrze z poprawką PaX, nie musimy ich kompilować ponownie. Większość programów będzie działać prawidłowo bez żadnych modyfikacji. Większość problemów stwarzają biblioteki. Z własnego doświadczenia wiemy, że Debian Wody ma pewne problemy z kilkoma bibliotekami, ale za to bardzo istotnymi (np. biblioteka zlib). Potwierdzają to także użytkownicy starszych wersji Red Hat-a. Tak więc poddaliśmy Debiana Sarge pobieżnym testom i okazało się, że nie było z nim żadnych problemów na jądrze z poprawką PaX (wyjątkiem było XFree86).

Większość programów dobrze współpracuje z poprawką PaX, nawet po włączeniu najbardziej restrykcyjnych ustawień (takich jak w pakietach Adamantixa). Z kilkuset pakietów pro-

gramów, które zostały do tej pory zaadaptowane dla systemu Adamantix, tylko kilka z nich stwarza problemy przy współpracy z PaX. Większość z nich można z kolei bardzo łatwo usunąć. Najczęściej wymagane poprawki to zmiana flag kompilacji, szczególnie dla bibliotek. Czasami wymaga to od nas użycia języka C zamiast asemblera (np. dla bibliotek zlib i gnupg). Niekiedy też konieczna jest zmiana kodu źródłowego (z powodu sposobu pracy kompilatora C). Pozostaje tylko garstka programów, których nie można uruchomić. Dzieje się tak dlatego, że są one uzależnione od tworzenia kodu wykonywalnego w pamięci. Dobrym przykładem będzie tutaj SUN Java Runtime Environment (JRE). Aby zdefiniować wyjątki dla programów tego typu, można użyć programu chpax. Jego ustawienia zachowywane są w pliku wykonywalnym. Po uruchomieniu pliku PaX wykrywa te ustawienia i wyłącza niektóre z testów kontrolnych. Najtrudniejsze przypadki programów mogą być przygotowane w taki sposób, aby działały bez zmiany kodu.

Paxtest

Kiedy twórcy Adamantixa zdecydowali się na użycie poprawki w ich jądrze, przekompilowałem ponownie programy dla PaX ASLR. Wszystko działało na tyle dobrze, że zacząłem wątpić, czy użycie poprawki PaX ma jakikolwiek sens. Przed rozpoczęciem instalacji oczekiwaliśmy wielu problemów do pokonania, a tymczasem nie wydarzyło się nic poważnego. Jak to możliwe? Być może poprawka PaX w ogóle nie działa. Nie chcąc opierać się na spekulacjach chciałem mieć dowód działania PaX i dlatego zacząłem tworzenie programu paxtest.

Paxtest to zestaw programów, z których każdy sprawdza jeden z funkcjonalnych aspektów PaX. Jeden z nich zapisuje kod maszynowy w ciągu znakowym, a następnie próbuje wykonać ten kod. Prawidłowo działająca konfiguracja z Pax powinna wykryć problem i zatrzymać wykonywanie programu. Inne programy sprawdzają kolejno wszystkie inne funkcje ochronne poprawki PaX.

W pakiecie znajdują się także testy sprawdzające randomizację ASLR. Dzięki temu otrzymujemy w miarę pełny obraz poziomu zabezpieczeń. Im więcej testów spowoduje wstrzymanie działania programu, tym lepiej. Wykorzystując dane pochodzące z pakietu Paxtest dowiedzieliśmy się, że poprawka PaX doskonale współpracuje z jądrem Adamantixa, a brak poważniejszych problemów był po prostu oznaką, że PaX pracuje znacznie lepiej niż się spodziewaliśmy.

Porównanie: PaX a inne poprawki

Dodatkową zaletą pakietu Paxtest jest możliwość przetestowania także innych poprawek zapewniających ochronę pamięci. Paxtest można pobrać ze strony internetowej znajdującej się pod adresem [2], skompilować go, a następnie uruchomić i porównać wyniki (można użyć polecenia `apt-get install paxtest`, jeżeli korzystamy z Adamantixa).

Po uruchomieniu Paxtestu na systemie z łąką exec-shield otrzymamy ciekawe wyniki (uzależnione od wersji Paxtestu). Występujące w starszych wersjach Paxtestu błędy mylnie przekonały niektórych użytkowników, że exec-shield radzi sobie znacznie lepiej niż w rzeczywistości. Ogólnie rzecz biorąc, ochrona przy zastosowaniu exec-shield jest niewystarczająca. Byłoby dobrze, gdyby komuś udało się przenieść paxtest do systemu OpenBSD i tam przeprowadzić testy. Przypuszczamy, że wyniki modułu W^X z systemu OpenBSD nie byłyby zbyt zachęcające.

Na zakończenie

Ochrona pamięci przetwarzania jest bardzo ważna, ale jak na razie nie opracowano rozwiązania chroniącego systemy przed wszystkimi trzema klasami ataków. Najstarszą poprawką ochrony pamięci dla jądra, a jednocześnie najlepszą w chwili pisania tego artykułu, jest poprawka PaX. Przy pomocy list dostępu (ACL) i innych mechanizmów kontroli dostępu, gwarantuje ona ochronę przed atakami klasy (1).

Aby ochronić nasze systemy przed atakami z klasy (2) i (3), niezbędne są kolejne mechanizmy obronne, takie jak SSP. Koszt wdrożenia zabezpieczeń tego typu jest relatywnie niski. Wszystkie dystrybucje Linuksa, dla których bezpieczeństwo danych jest istotną sprawą, powinny w niedługim czasie wyposażać się w taki rodzaj zabezpieczeń. ■

INFO

- [1] <http://old.lwn.net/1998/0806/allinus-noexec.html>
- [2] <http://pageexec.virtualave.net/>
- [3] <http://www.rsac.org/>
- [4] <http://archives.n.eohapsis.com/archives/openbsd/2003-04/1362.html>
- [5] <http://people.redhat.com/mingo/exec-shield/>
- [6] <http://www.adamantix.org/>