Autor: Tomasz Kurpios, nr indeksu 209480

Niniejszy dokument to notatki do prezentacji dotyczącej projektu PaX. Prezentację na ćwiczeniach oparłem na widocznych po lewej stronie slajdach, dodatkowo omawiałem (w pewnym skrócie) to, co znajduje się po prawej stronie slajdów. Wypunktowania w tekście ściśle odpowiadają wypunktowaniom na slajdach.

Czym jest Pax?

- Łatka na jądro Linuxa
- Zabezpiecza przed wykorzystywaniem błędów w programach do zdobycia dostępu do pamięci procesów
- Błędy te najczęściej umożliwiają przepełnienie bufora

PaX (łac. pax – pokój) to łatka na jądro Linuxa, której zadaniem jest zabezpieczenie systemu przed wykorzystywaniem błędów oprogramowania (takich jak przepełnienie bufora, przepełnienie integera, niewłaściwy format łańcucha) do zdobycia kontroli nad procesem i, co może się z tym wiązać, systemem. Powstał w roku 2000, jego autor z nieznanych przyczyn pragnie pozostać anonimowy.

Koncepcja

Zamiast: znajdować błędy w programach i je usuwać

Można: utrudnić życie intruzowi próbującemu dany błąd wykorzystać

Celem PaX-a nie jest tropienie błędów w programach – to jest zadaniem programistów. Skupiono się tutaj natomiast na uniemożliwieniu wykorzystania takich błędów poprzez wprowadzenie pewnych mechanizmów obsługi pamięci procesów, dzięki którym nieuprawnione wtargnięcie do niej staje się bardzo trudne lub niewykonalne, ponadto próby takie są możliwe do wytropienia.

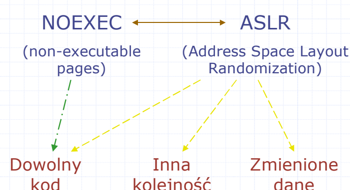
Czego może chcieć intruz?

1) Wprowadzić lub wykonać dowolny kod zamiast istniejącego
2) Wykonać istniejący kod w innym porządku niż go stworzono
3) Wykonać istniejący kod ze zmienionymi danymi

Ponieważ głównym błędem, któremu przeciwdziała PaX jest błąd przepełnienia bufora (bpb), opis ograniczę tylko do tego rodzaju. Intruz może wykorzystać taki błąd na 3 sposoby, wymienione w slajdzie obok.

1) Przykład: shellcode (wykorzystanie błędu w celu uruchomienia nowej powłoki i w ten sposób przejęcia kontroli nad systemem)
2) Tzw. ret2libc – nadpisanie adresu powrotnego funkcji, przez co powrót nastąpi w inne miejsce w pamięci
3) Najrzadziej stosowane, nadpisanie danych umieszczonych w pamięci bezpośrednio za przepełnionym buforem

Jak temu przeciwdziała PaX?



NOEXEC i ASLR to mechanizmy, czy też podejścia PaX-a do radzenia sobie z bpb. Strzałkami przerywanymi oznaczono, który z mechanizmów przeciwdziała któremu ze sposobów wykorzystania błędu. Należy pamiętać, że chociaż są to odrębne mechanizmy, to jednak celem większego zabezpieczenia należy używać ich jednocześnie.

NOEXEC

Założenia:

- Jeśli pewne dane nie muszą być wykonywalne, to nie powinny być wykonywalne
- Jeśli proces nie wymaga dynamicznego generowania kodu, to nie powinien mieć do tego prawa

Ogólnie rzecz biorąc zasada działania NOEXEC (niewykonywalne strony pamięci) jest taka, aby do każdej strony w pamięci można było albo zapisać dane albo wykonać kod w niej umieszczony, jednak nigdy nie

pozwolić na wykonanie tych dwóch rzeczy jednocześnie. W tym celu należy mieć możliwość oznaczania określonych stron pamięci jako niewykonywalne.

Jak zaznaczyć stronę pamięci jako niewykonywalną?

- Wsparcie sprzętowe – NX bit (architektury: alpha, ppc, parisc, sparc, sparc64, amd64 i ia64) – problem trywialny
- Brak wsparcia (ia32) - SEGMEXEC

PaX wykorzystuje dwa sposoby oznaczenia stron pamięci jako niewykonywalne. Sposoby te są zależne od architektury komputera.

1) NX bit (non-executable bit) to bit numer 63 (ostatni) każdego wpisu w tablicy stron, co oznacza, że każdą stronę można łatwo oznaczyć jako niewykonywalną. Jest zatem dokładnie tym, czego potrzebujemy.
2) W jednej z najpopularniejszych architektur, tj. ia32 brakuje takiego bitu, toteż należy zastosować inne podejście. Dawniej podejściem tym był tzw. PAGEEXEC, wykorzystujący logikę stronicowania w celu uzyskania możliwości wykonania potrzebnego nam oznaczenia. Mówiąc w skrócie PAGEEXEC wykorzystywał fakt, że procesory Intela Pentium i późniejsze posiadają odrębne bufory TLB dla danych i kodu. Bardziej wydajnym podejściem okazał się być SEGMEXEC, wykorzystujący z kolei schemat segmentacji.

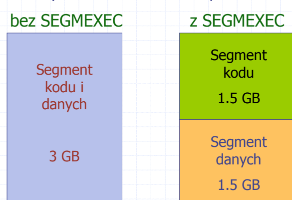
SEGMEXEC

- Emulacja bitu NX poprzez podział wirtualnej przestrzeni pamięciowej użytkownika na 2 połowy:
- Górna połowa – kod (adresy: 0x00000000 – 0x5fffffff)
- Dolna połowa – dane (adresy: 0x60000000 – 0xbfffffff)
- Ponadto: odbicie zawartości górnej połowy w dolnej

SEGMEXEC to pomysł polegający na oznaczeniu stron jako niewykonywalne poprzez podział przestrzeni pamięciowej użytkownika na 2 części (na slajdzie). Wykorzystuje następnie mechanizm PaX-a zwany Virtual Memory Address Mirroring do odzwierciedlenia zawartości górnej połowy w dolnej. Jest to potrzebne dlatego, że mapowania z segmentu kodu mogą również zawierać pewne dane (jak choćby stałe łańcuchy).

SEGMEXEC – c.d.

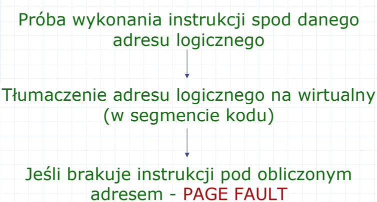
Wirtualna przestrzeń adresowa procesu



Oto rysunek ilustrujący wygląd przestrzeni adresowej procesu bez użycia mechanizmu SEGMEXEC oraz z jego użyciem. 3 GB to domyślnie przyznawana ilość pamięci wirtualnej dla procesu.

SEGMEXEC – c.d.

Działanie prewencyjne:



Ogólny schemat działania SEGMEXEC jest następujący: gdy następuje próba wykonania instrukcji spod danego adresu logicznego, adres ten zostaje przetłumaczony na wirtualny znajdujący się w segmencie kodu. Jeśli pod obliczonym adresem nie ma spodziewanego kodu (co dzieje się właśnie wtedy, gdy kod ten jest w segmencie danych, a znaleźć się tam mógł jedynie w wyniku bpb) podnoszony jest błąd PAGE FAULT, zatem proces zostaje zabity. Ponadto dzięki temu, że każda próba kończy się zgłoszeniem PAGE FAULT, łatwe jest wykrycie w systemie prób wtargnięcia.

SEGMEXEC – implementacja

Serce implementacji – VMA Mirroring
Cel: odbicie zawartości segmentu kodu w segmencie danych
Dodatkowo w katalogu arch/i386/kernel/:
- Nowa tablica GDT (head.s)
- Modyfikacja procedury zamiany kontekstu (__switch_to() w process.c)
- Synchronizacja z drugą tablicą GDT (APM w apm.c, LDT i TSS w traps.c)
- Blokada definiowania własnych deskryptorów segmentu kodu (ldt.c)

W pliku fs/binfmt_elf.c: zmiana w procedurze load_elf_binary() – przygotowanie procesu do wykonania

Implementacja SEGMEXEC w dużej mierze oparta jest na implementacji mechanizmu VMA Mirroring, który dokonuje opisanego wcześniej podziału w procedurze do_mmap() w pliku include/linux/mm.h. Oprócz tego, na slajdzie wymienione są zmiany w jądrze, które dodaje SEGMEXEC.

1) Nowa tablica GDT (Global Descriptor Table) jest potrzebna z dwóch powodów: z jednej strony upraszcza implementację, z drugiej uniemożliwia wykorzystaniu instrukcji (takich jak retf) do przeniesienia wykonywania z segmentu kodu do segmentu danych.
2) Ponieważ każdy proces korzystający z SEGMEXEC posiada inne tablice GDT, należy odpowiednio je przeładować podczas zamiany kontekstu i przed początkiem wykonywania procesu (odpowiednio: __switch_to() i load_elf_binary())
3) Należy dokonać pewnych synchronizacji między tablicami GDT (APM – Advanced Power Management, LDT – Local Descriptor Table, TSS – Task State Segment)
4) Aby uniemożliwić tworzenia własnych deskryptorów segmentu kodu, co rujnowałoby koncepcję PaX-a, jako że pozwalałoby na wyłamanie się z segmentu chronionego, wykonywane są dodatkowe sprawdzenia w procedurze write_ldt() w wymienionym na slajdzie pliku

MPROTECT

Cel: *powstrzymać wprowadzanie nowego (wykonywalnego) kodu do przestrzeni adresowej procesu*

Środek: *ograniczenie możliwości funkcji systemowych: mmap() i mprotect()*

Mając już semantykę niewykonywalności stron (zapewnioną przez SEGMEXEC lub NX bit), zastosowana zostaje ona do powstrzymania wprowadzenia kodu do przestrzeni adresowej procesu. W tym celu zmienione (ograniczone) zostają funkcje systemowe mmap() (czyli mapowanie plików/urządzeń do pamięci) oraz mprotect() (czyli zmiana uprawnień do przydzielonych stron)

MPROTECT – c.d.

Czemu zapobiegają te ograniczenia?
- Tworzeniu anonimowych, wykonywalnych mapowań
- Tworzeniu wykonywalnych i zapisywalnych mapowań plików
- Nadawaniu takich praw już istniejącym mapowaniom

Ograniczenia te zapobiegają wymienionym na slajdzie rodzajom mapowań.

1) Są to mapowania związane ze stosem oraz ze stertą poprzez funkcje: brk() (zmiana rozmiaru segmentu danych) oraz mmap(). Jądro pozwala, żeby takie mapowania były wykonywalne i zapisywalne jednocześnie
2) Tak jak powyżej, takie mapowania bez zmian w jądrze pozwalają na ich jednoczesną wykonywalność i zapisywalność.
3) Każde mapowanie, które było kiedykolwiek zapisywalne, nigdy nie może się stać wykonywalne. Każde mapowanie, które jest wykonywalne, nie może się stać zapisywalne.

MPROTECT – implementacja

Dla jakich stanów niemożliwe jest wprowadzenie nowego, wykonywalnego kodu?
Są to tzw. „dobre stany”:

- VM_WRITE
- VM_MAYWRITE
- VM_WRITE | VM_MAYWRITE
- VM_EXEC
- VM_MAYEXEC
- VM_EXEC | VM_MAYEXEC

Stany niedopuszczalne przez jądro

Stany mapowań określone są w strukturze vma (pole vm_flags) i określają, czy dany obszar (i co za tym idzie strony jemu przydzielone) jest wykonywalny/zapisywalny (stany VM_EXEC, VM_WRITE) lub czy może taki się stać po użyciu funkcji mprotect() (VM_MAYEXEC, VM_MAYWRITE).
PaX definiuje stany „dobre” jako takie, które uniemożliwiają wprowadzenie nowego, wykonywalnego kodu do danego obszaru. Stany te wymienione są na slajdzie. Zauważmy, że pierwsze trzy stany pozwalają na zapis do określonego obszaru. Uniemożliwione jest natomiast jego wykonanie. Ostatnie trzy z kolei, pozwalają co prawda na wykonanie, jednak nie można w trakcie wykonywania procesu niczego do tych obszarów zamapować. Stany zaznaczone na czerwono są niedopuszczalne przez jądro Linuxa, ponieważ flaga VM_WRITE pociąga za sobą VM_MAYWRITE, analogicznie VM_EXEC. Pozostają zatem cztery „dobre” stany.

MPROTECT – implementacja c.d.

Rodzaj mapowania		Stan bez MPROTECT	Stan z MPROTECT
Mapowania anonimowe (sterta przez brk() i mmap() oraz stos)		W | MW | X | MX	W | MW
Mapowania pamięci dzielonej		W | MW	W | MW
Mapowania plików	mmap() z PROT_WRITE	MW | MX | ...	W lub W | MW
	mmap() bez PROT_WRITE	MW | MX | ...	X | MX

Tabelka przedstawia sposób, w jaki MPROTECT przeciwdziała powstawaniu „złych” stanów. O ile są one normalnie dopuszczalne przez jądro, o tyle użycie MPROTECT powoduje ich modyfikację.
Zastosowałem skróty: M – MAY, W – WRITE, X – EXEC i pominąłem prefix VM_ celem większej czytelności tabelki.

1) Opisane wcześniej mapowania anonimowe otrzymują od jądra status W| MW | X | MX, który jest „zły”. MPROTECT odbiera takim mapowaniom prawo wykonywania.
2) MPROTECT nie zmienia niczego jeśli chodzi o mapowania pamięci współdzielonej przez procesy, co wynika z tego, że jądro samo przyznaje takim mapowaniom „dobry” stan
3) W przypadku mapowań plików (tutaj jądro również przyznaje „zły” stan), nadawany jest im status X | MX, chyba że jawnie poproszono o możliwość zapisu (flaga PROT_WRITE w mmap()), wówczas kosztem realizacji tej prośby odebrana jest możliwość wykonania.

MPROTECT – implementacja c.d.

Co z aplikacjami, które wykorzystują „złe” stany?
Należy rozszerzyć interfejs mmap(), tzn. dodać nowe flagi: PROT_MAY*

Jak zapewnić zgodność z MPROTECT?
Zadanie dla programisty:

1) mmap(..., PROT_READ | PROT_WRITE | PROT_MAYREAD | PROT_MAYEXEC, ...)
2) generowanie kodu do powyższego obszaru
3) mprotect(..., PROT_READ | PROT_EXEC)

Istnieją jednak aplikacje, które wymagają możliwości zapisu i wykonania danego obszaru pamięci (są to takie aplikacje, które dynamicznie generują kod). PaX stosuje specjalny zabieg, aby to umożliwić, wymaga to jednak także „współpracy” programisty piszącego takie aplikacje. Metoda ta jest przedstawiona na slajdzie. Należy zwrócić uwagę, że choć dozwolone jest wykonanie mmap() z flagą ... | PROT_WRITE | PROT_MAYEXEC | ... , to aby dokonać PROT_EXEC należy wykonać mprotect() po zakończeniu generowania kodu do zamapowanego obszaru pamięci.

MPROTECT – implementacja c.d.

Zmiany w jądrze:

- Zapobieganie tworzeniu niewłaściwych mapowań: w pliku mm/mmap.c do_mmap_pgoff() i do_brk()
- Zapobieganie próbom niewłaściwych zmian uprawnień do mapowanych obszarów: w pliku mm/mprotect.c sys_mprotect()

Implementacja MPROTECT znajduje się w plikach wymienionych na slajdzie. Zmiany te odbywają się wewnątrz funkcji systemowych mmap() i mprotect(), a opisane zostały wcześniej.

Na slajdzie przedstawiony jest układ przestrzeni adresowej procesu niewykorzystującego oraz wykorzystującego mechanizm NOEXEC. Warto zwrócić uwagę, że przy wykorzystaniu NOEXEC żaden obszar pamięci nie jest jednocześnie wykonywalny i zapisywalny.

MPROTECT – furtka dla intruzów

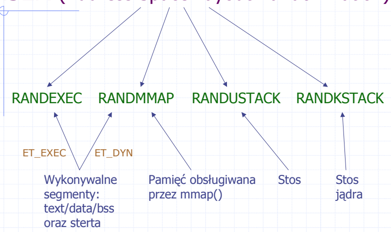
Sposób na ominięcie zabezpieczenia MPROTECT:

Zmapowanie do pamięci pliku żądając jednocześnie prawa do wykonania (PROT_EXEC)

Jedyne wymaganie:

Intruz musiał mieć możliwość tworzenia i zapisywania do pliku w systemie

Istnieje sposób na ominięcie zabezpieczeń oferowanych przez MPROTECT, którego świadomi są jego projektanci i przed nim przestrzegają. Ponieważ zgodnie z przedstawioną kilka slajdów wcześniej tabelką można zamapować z flagą wykonywalności do pamięci plik, to jeśli intruz miał możliwość wcześniejszego utworzenia i zapisania do pliku w systemie, to umożliwione mu będzie wykonanie potencjalnego kodu znajdującego się w tymże pliku. Radzenie sobie z taką sytuacją wykracza jednak poza ramy projektu PaX. Zadanie to zostawia się innym systemom ochrony dostępu.



ASLR (losowość układu przestrzeni adresowej) to drugi (po NOEXEC) mechanizm walki z intruzami wykorzystującymi bpb. Polega on na wprowadzeniu losowości w dotychczas sztywne rozłożenie poszczególnych obiektów w przestrzeni adresowej. Powoduje to, że możemy stwierdzić z pewnym mierzalnym, bardzo wysokim prawdopodobieństwem, że pewne techniki wykorzystywania bpb się nie powiodą. Losowość ta jest wprowadzana inaczej dla różnych rodzajów obiektów w pamięci. Sposoby wprowadzania zaznaczone są na slajdzie kolorem zielonym, zaś niebieskim rodzaje obiektów im odpowiadających. Dodatkowo należy zwrócić uwagę na różnice dla plików wynikowych ELF w formacie ET_DYN (stworzone z kodu PIC - niezależnego od położenia w pamięci), jak i ET_EXEC (kod statycznie położony).

ASLR - implementacja

Zmiany w jądrze:

Inicjalizacja trzech zmiennych (delta_exec, delta_mmap, delta_stack) losowymi wartościami przy tworzeniu procesu

Gdzie:

load_elf_binary() w fs/binfmt_elf.c

Ponadto:

Dodatkowe zmiany dla każdej składowej ASLR

Implementacja ASLR jest podzielona na cztery składowe wymienione na poprzednim slajdzie. Dodatkowo przeprowadzane są zmiany widoczne na tym slajdzie. load_elf_binary() to funkcja wywoływana bezpośrednio przed rozpoczęciem działania procesu.

RANDEXEC - działanie

- Randomizacja adresów mapowań plików ET_EXEC
- Dwa mapowania plików, będące wzajemnymi odbiciami
- Przy próbie wykonania instrukcji spod oryginalnego adresu podnoszony jest PAGE FAULT, który jest następnie obsługiwany:
  - Jeśli próba ta była uzasadniona – wykonanie jest przeniesione do odbicia instrukcji
  - Jeśli nie (atak ret2libc) – proces jest zabijany

Celem działania RANDEXEC jest wprowadzenie losowości w adresy mapowań plików ELF zakładających statyczne położenie kodu w pamięci (ET_EXEC). Realizowane jest to poprzez wykonanie dodatkowego odbicia takiego mapowania w losowym miejscu pamięci (wykorzystywany jest ten sam mechanizm, co przy SEGMEXEC, czyli VMA Mirroring). Ponadto oryginalne mapowanie oznaczone zostaje jako niewykonywalne. Zauważmy, że w związku z tym każda próba wykonania instrukcji spod oryginalnego adresu zostanie zakończona zgłoszeniem błędu PAGE FAULT, wymagana jest więc dokładna obsługa tego błędu. Polega ona na sprawdzeniu, co znajduje się bezpośrednio pod wskaźnikiem stosu procesu (ESP–4). Jeśli jest tam błędny adres, potraktowane zostaje to jako atak ret2libc (opisany wcześniej) i proces zostaje zabity. W przeciwnym razie następuje przekierowanie do obszaru odbitego. Ponieważ adres może zostać uznać za błędny również w przypadku „niewinnego” procesu, zdarza się, że przez RANDEXEC są one zabijane, co należy uznać za pewien minus tego mechanizmu.

RANDEXEC + SEGMEXEC

Mapowania sekcji text plików ET_EXEC:

- Do segmentu danych i segmentu kodu (SEGMEXEC)
- Mapowanie z segmentu danych rozbite na dwa (RANDEXEC)

Wystarczą dwa mapowania sekcji text (SEGMEXEC)

Zastanówmy się, jak wygląda przestrzeń wirtualna procesu przy jednoczesnym użyciu RANDEXEC i SEGMEXEC. Sekcja text (wykonywalna) plików ET_EXEC znajduje się wówczas w trzech miejscach (opis na slajdzie). Ponieważ system segmentacji zapewnia wystarczające bezpieczeństwo, a także aby uprościć implementację VMA Mirroring odpowiedzialnego za dokonywanie takich odbić, dla tego szczególnego pominięto randomizację wykonywaną przez RANDEXEC.

RANDEXEC - implementacja

Sercem RANDEXEC jest VMA Mirroring

Inne zmiany w jądrze:

- Bezpośrednie wywołanie do_mmap_pgoff() zamiast elf_map() (load_elf_binary() w fs/binfmt_elf.c)
- Obsługa wyjątku PAGE FAULT i ewentualne przekierowanie (do_page_fault(), pax_do_page_fault(), pax_handle_fetch_fault() w arch/i386/mm/fault.c)

Na slajdzie wypisane są zmiany w jądrze implementujące RANDEXEC.

1) Ponieważ elf_map() jest otoczką wokół do_mmap(), gdzie wykonywane jest odbicie na użytek SEGMEXEC, wywoływany jest zamiast elf_map() do_mmap_pgoff(), która to funkcja z kolei jest normalnie wywoływana z do_mmap().
2) Tutaj odbywają się dodatkowe sprawdzenia, stwierdzające, czy można dokonać przekierowania

RANDMMAP – działanie i implementacja

Randomizacja obszarów pamięci obsługiwanych przez do_mmap()

Zmiany w jądrze:

- Na podstawie delta_mmap ustawiane są bity 12-27 stałej TASK_UNMAPPED_BASE w mm/mmap.c
- Zmiana wartości stałej ELF_ET_DYN_BASE:
  0x08048000 + delta_exec
  w fs/binfmt_elf.c

Adres bazowy ET_EXEC

RANDMMAP to druga ze składowych ASLR. Sam mechanizm przydziału pamięci dla pozostaje niezmieniony w stosunku do tego, co oferuje jądro, tzn. wyszukiwany jest pierwszy wystarczająco duży obszar pamięci, rozpoczynając od adresu określonego przez TASK_UNMAPPED_BASE. RANDMMAP powoduje wprowadzenie losowości do tejże stałej, przez co zmienia miejsce rozpoczęcia szukania i, co za tym najczęściej idzie, przyczynia się do przydzielenia innego obszaru pamięci niż normalnie. Losowość ta polega na ustawieniu odpowiednich bitów tej stałej na podstawie delta_mmap (jest to jedna z trzech stałych inicjalizowanych dla każdego procesu, co było opisane kilka slajdów wcześniej). Z kolei dla plików ET_DYN zmieniana jest stała ELF_ET_DYN_BASE zgodnie z opisem na slajdzie.

RANDUSTACK – działanie i implementacja

Randomizacja adresów stosu wykonana podczas tworzenia procesu

Zmiany w jądrze:

- Losowość w pamięci jądra w zakresie adresów, które będą przeznaczone dla procesu na stos (bity 2-11) (do_execve() w fs/exec.c)
- Losowość w adresie stosu zmapowanym do przestrzeni użytkownika (używając delta_stack do bitów 12-27 stałej STACK_TOP (include/asm-i386/a.out.h)

RANDUSTACK wprowadza losowość w adres stosu dla każdego procesu. Wykonywane jest to dwustopniowo:

1) Odbywa się w przestrzeni jądra podczas alokacji stron na stos dla procesu
2) Odbywa się w przestrzeni procesu, podczas mapowania do niej obszarów przydzielonych w punkcie 1)

Sposób wprowadzenia tej losowości opisany jest na slajdzie.

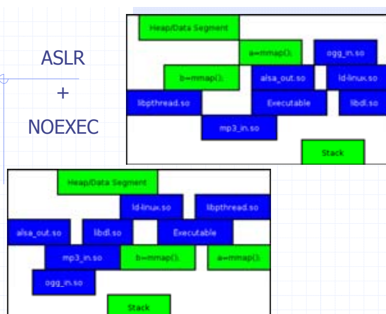
RANDKSTACK – działanie i implementacja

Randomizacja adresów stosu jądra dla procesów (przed każdym powrotem z wywołania funkcji systemowej)

Zmiany w jądrze:

- Funkcja systemowa pax_randomize_kstack() w arch/i386/kernel/process.c
  - Losowość bitów 2-6 wskaźnika stosu jądra
  - W dwóch miejscach:
    - tss->esp0 (wskaźnik stosu w Task State Segment)
    - current->thread.esp0 (do przeładowania powyższego przy przełączaniu kontekstu)

RANDKSTACK wprowadza losowość w adres stosu jądra dla każdego procesu. Liczba losowa generowana jest na podstawie instrukcji rdtsc (read time stamp counter). Obliczanie wskaźnika stosu następuje poprzez zaaplikowanie do jego bitów 2-6 (xor) tejże liczby losowej.



Na slajdzie przedstawiono dwa przykłady możliwych układów przestrzeni adresowych procesów wykorzystujących mechanizmy ASLR i NOEXEC (czyli wszystko, o czym mówiliśmy). Jak widać wszystkie obiekty w pamięci są albo wykonywalne albo zapisywalne, ponadto każdy z nich znajduje się w losowym (choć nie zupełnie dowolnym) miejscu tej przestrzeni.

Podsumowanie możliwości

Czego może chcieć intruz?
1) Wprowadzić lub wykonać dowolny kod zamiast istniejącego
2) Wykonać istniejący kod w innym porządku niż go stworzono
3) Wykonać istniejący kod ze zmienionymi danymi

Czemu przeciwdziała PaX?
NOEXEC + MPROTECT (1) – z drobnym wyjątkiem
ASLR (1) (2) (3) – jeśli intruz musi znać strukturę adresów w infekowanym procesie i nie może jej poznać

Podsumowując, PaX chroni przed wykorzystaniem bpb przez intruza w następującym zakresie:

- przeciw (1), jeśli intruz nie ma możliwości utworzenia i zapisania pliku w atakowanym systemie
- przeciw (2) i (3) z bardzo wysokim prawdopodobieństwem, jeśli intruz potrzebuje znajomości układu przestrzeni adresowej atakowanego procesu oraz nie ma możliwości wydobycia tej informacji (np. poprzez jakiś błąd wycieku)

Pax – upadek projektu

1.04.2005
Oficjalne zamknięcie projektu PaX

Przyczyna:
VMA Mirroring podatne na privilege elevation vulnerability (podniesienie przywilejów) – problem krytyczny

Jak się okazało, projekt PaX nie jest wystarczającym zabezpieczeniem. W marcu 2005 roku odkryto, że mechanizm VMA Mirroring jest podatny na atak privilege elevation. Powoduje on podniesienie przywilejów (user -> superuser), co w konsekwencji sprawia, że kontrola nad systemem może zostać przejęta przez intruza. Twórcy PaX-a próbowali jeszcze radzić sobie z tym przypadkiem wydając poprawioną łatkę, jednak nie sprostano problemowi. Z racji tego, że wykrycie tej słabości całkowicie zrujnowało koncepcję, wedle której tworzony był PaX, projekt został oficjalnie zamknięty.

Bibliografia:

http://pax.grsecurity.net/docs/
http://en.wikipedia.org/wiki/PaX
http://www.grsecurity.net/PaX-presentation.ppt
http://www.linux-magazine.pl/issue/02/CoverStory_PAX.pdf