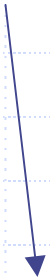
Czym jest Pax?

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

Koncepcja

Zamiast: znajdować błędy
w programach i je usuwać

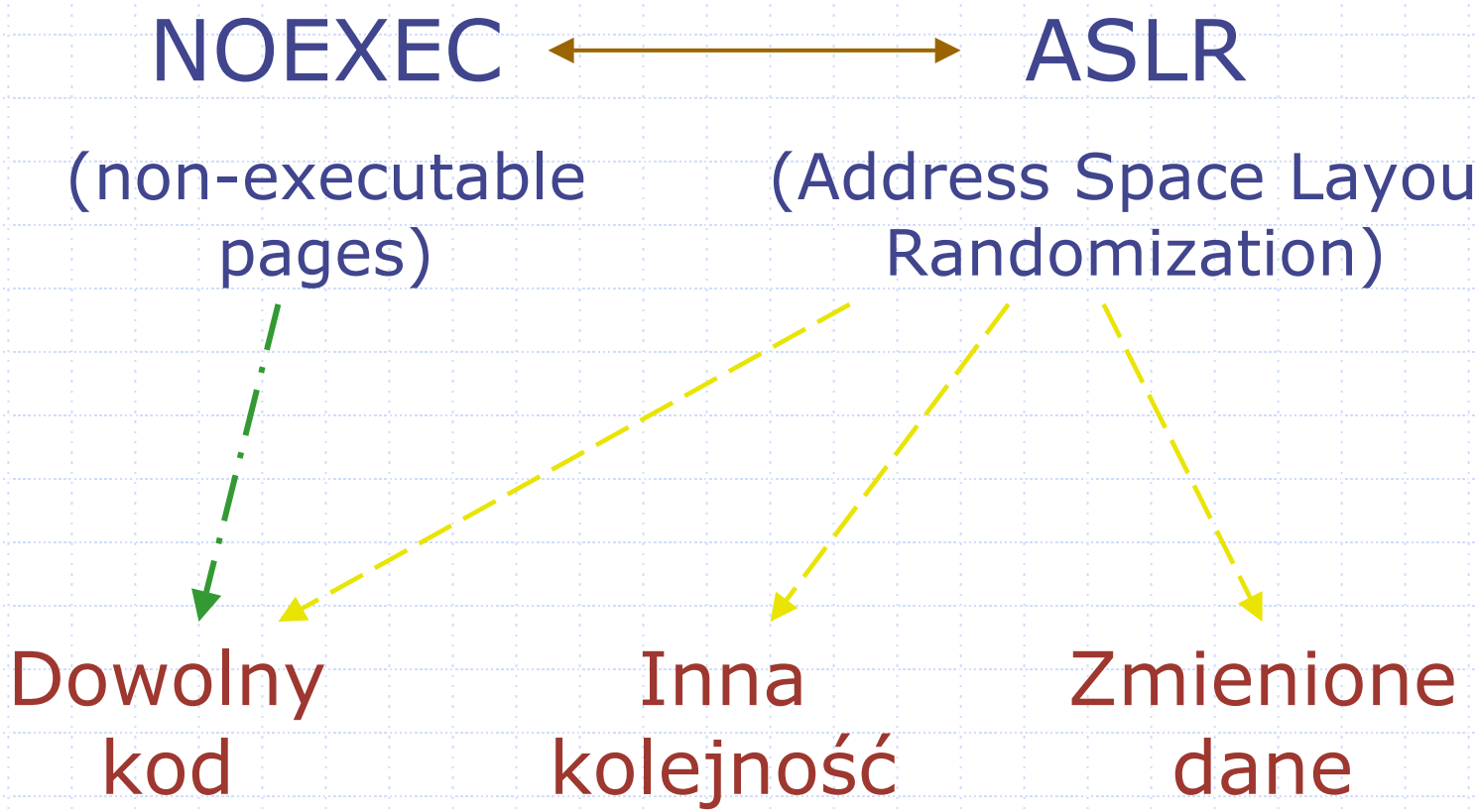


Można: utrudnić życie intruzowi
próbującemu dany błąd
wykorzystać

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

Jak temu przeciwdziała PaX?



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

Jak zaznaczyć stronę pamięci jako niewykonywalną?

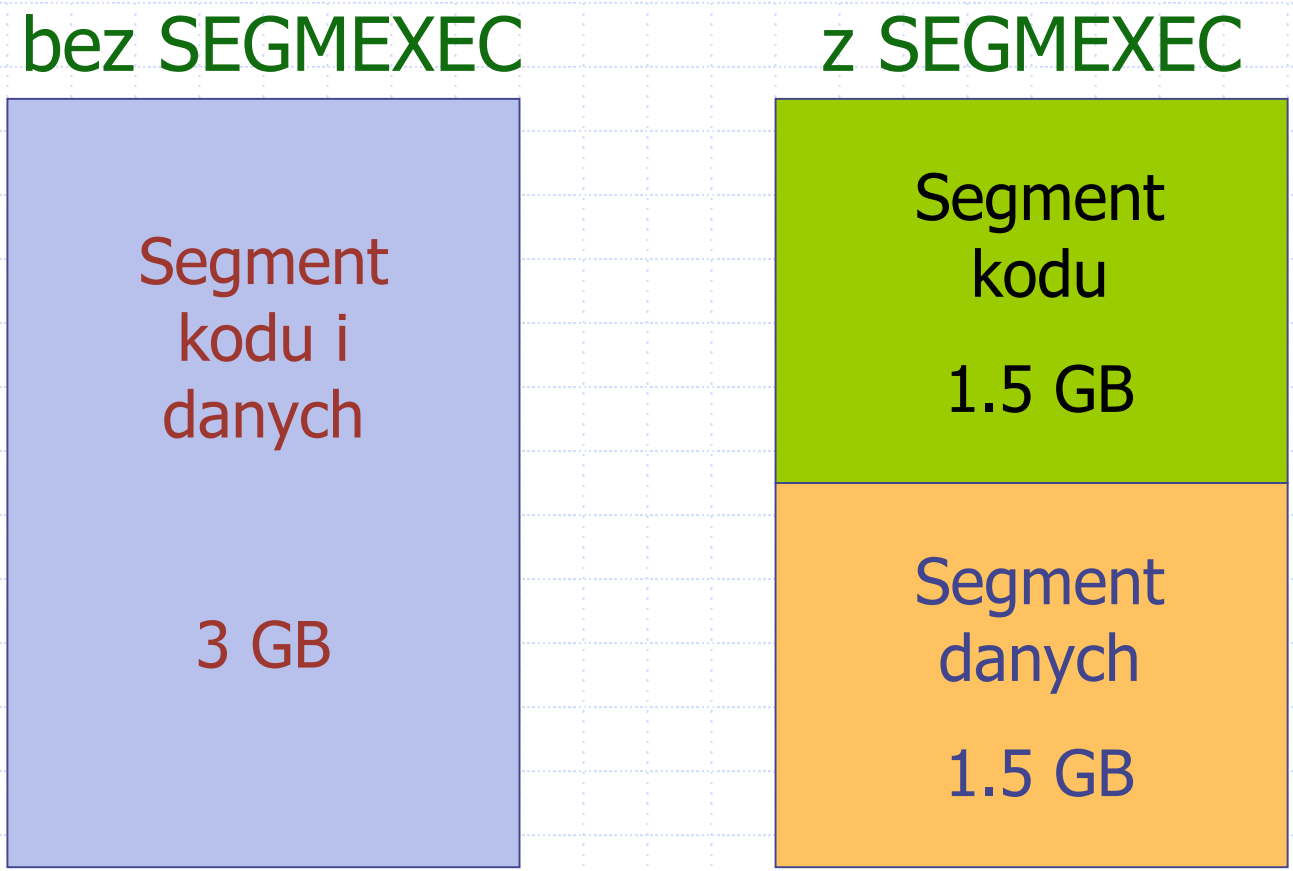
- Wsparcie sprzętowe – NX bit (architektury: alpha, ppc, parisc, sparc, sparc64, amd64 i ia64) – problem trywialny
- Brak wsparcia (ia32) - SEGMEXEC

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 – c.d.

Wirtualna przestrzeń adresowa procesu



SEGMEXEC – c.d.

Działanie prewencyjne:

Próba wykonania instrukcji spod danego adresu logicznego

↓

Tłumaczenie adresu logicznego na wirtualny (w segmencie kodu)

↓

Jeśli brakuje instrukcji pod obliczonym adresem - PAGE FAULT

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

MPROTECT

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

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

MPROTECT – c.d.

Czemu zapobiegają te ograniczenia?

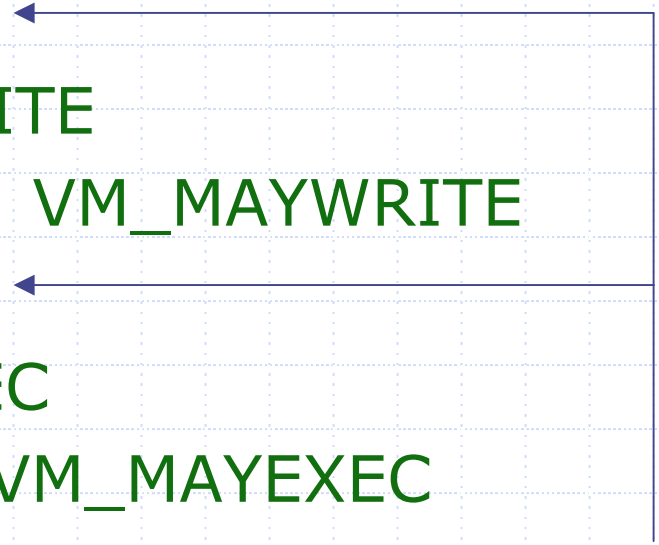
- Tworzeniu anonimowych, wykonywalnych mapowań
- Tworzeniu wykonywalnych i zapisywalnych mapowań plików
- Nadawaniu takich praw już istniejącym mapowaniom

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

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

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?

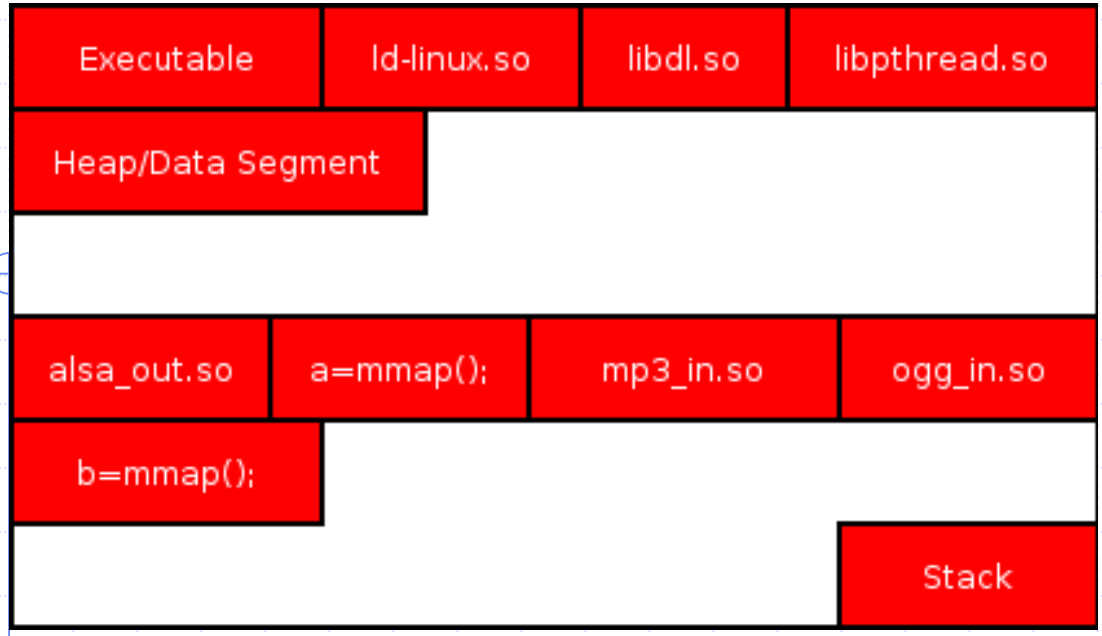
Zadanie dla programisty:

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

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()



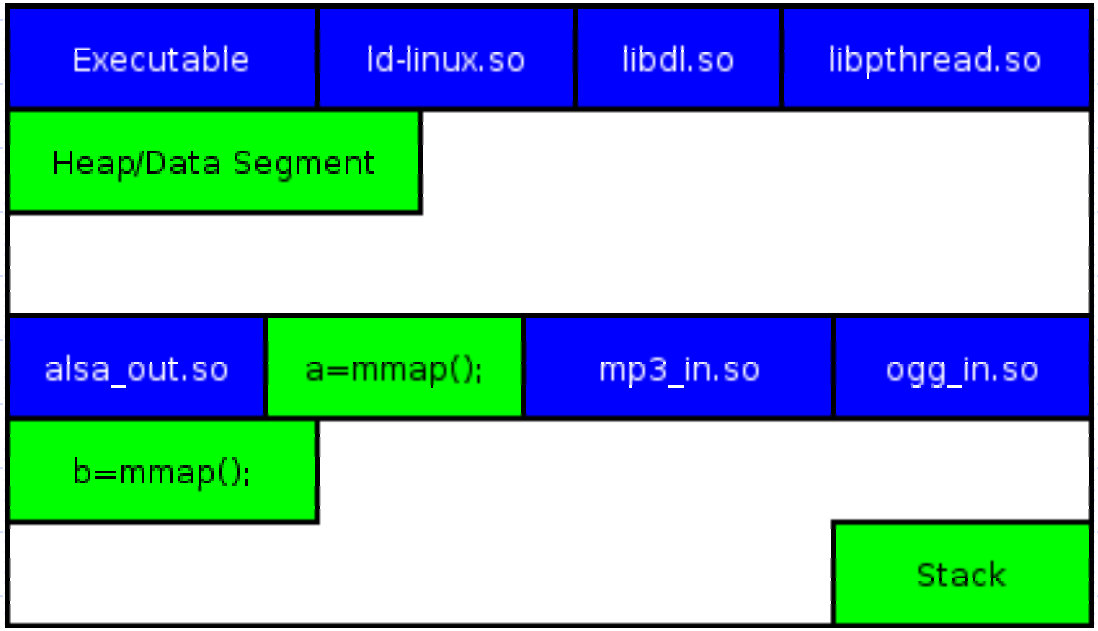
Bez NOEXEC

Można zapisywać i wykonywać

Z NOEXEC

Można zapisywać

Można wykonywać



MPROTECT – furtka dla intruzów

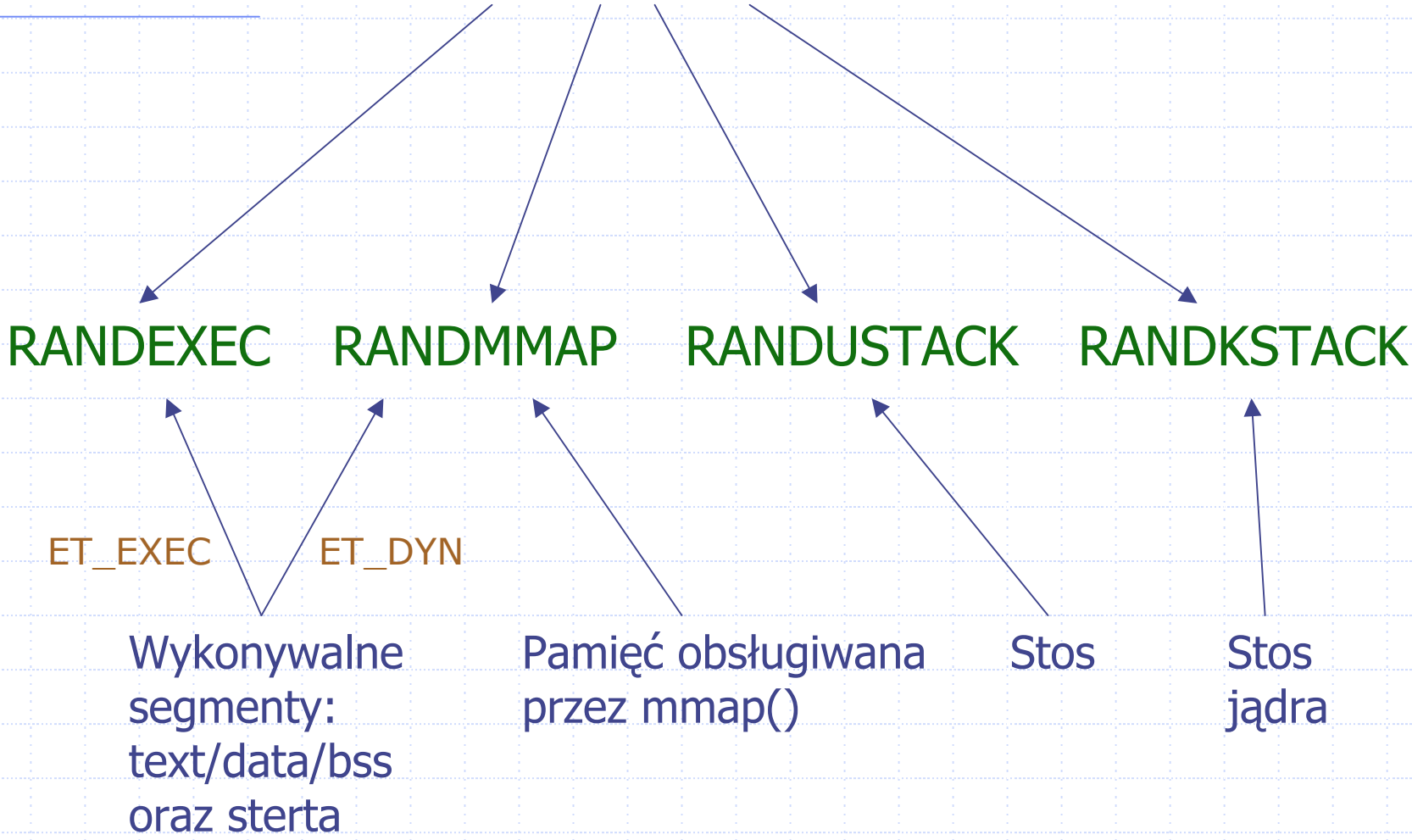
Sposób na ominięcie zabezpieczenia MPROTECT:

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

Jedyne wymaganie:

Intruz musiał mieć możliwość tworzenia i zapisywania do pliku w systemie

ASLR (Address Space Layout Randomization)



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

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

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)

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)

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

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)

RANDKSTACK – działanie i implementacja

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

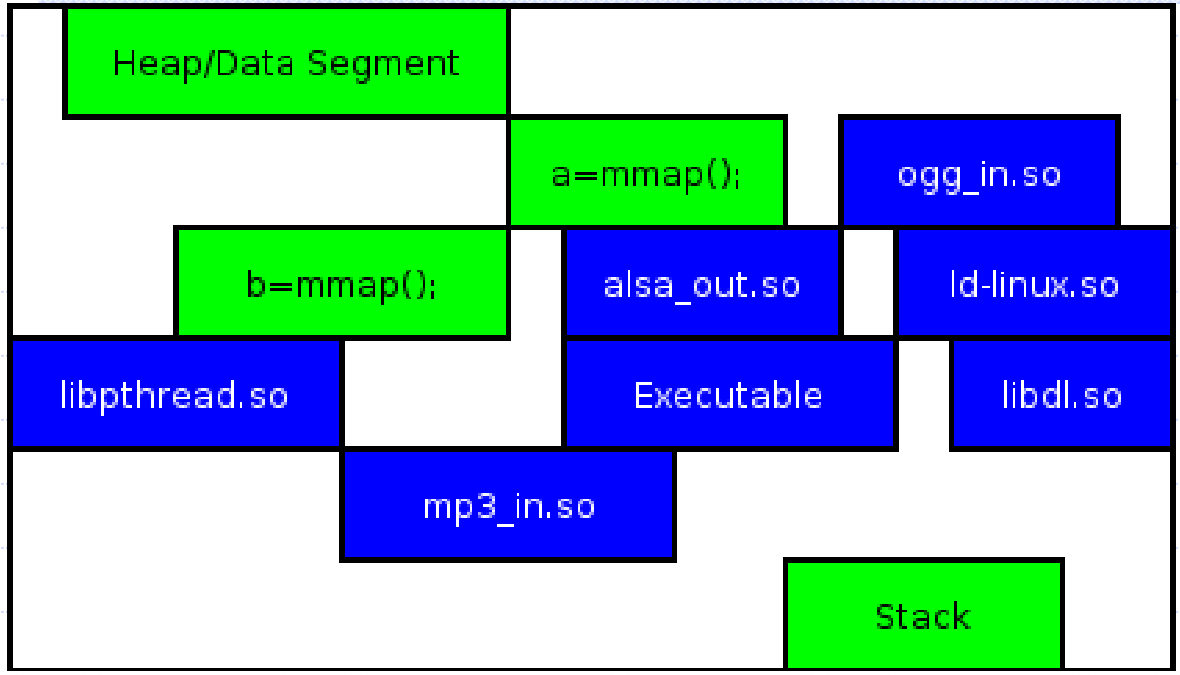
Zmiany w jądrze:

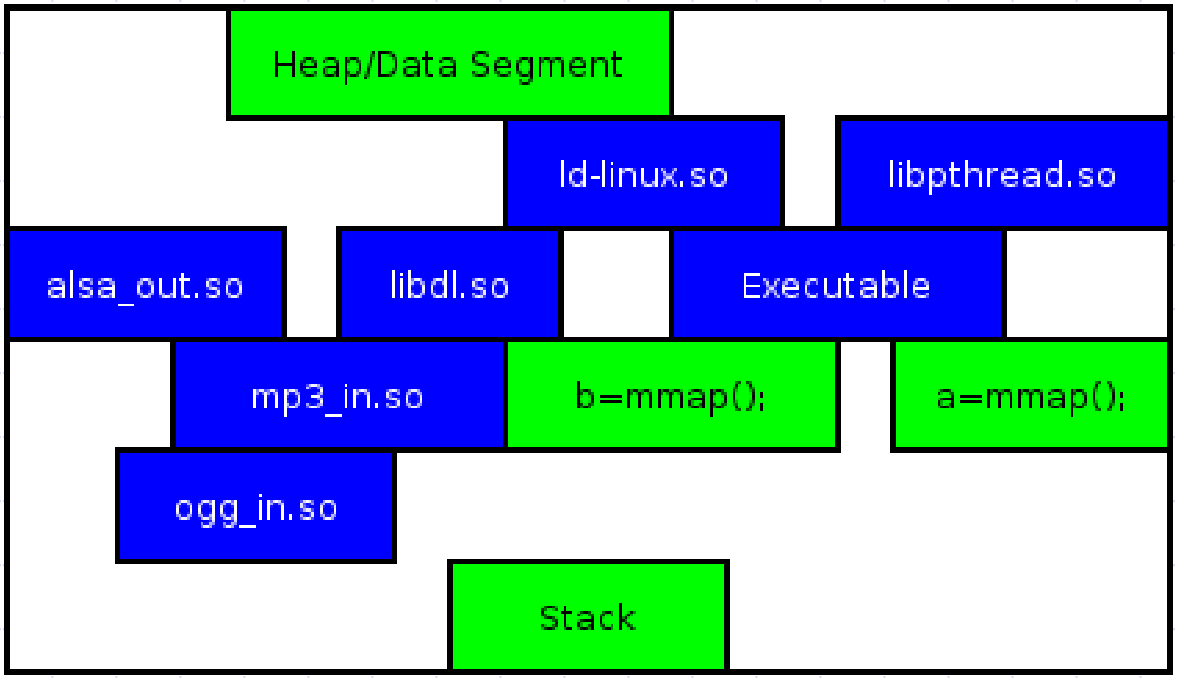
- Funkcja systemowa pax_randomize_kstack() w arch/i386/kernel/process.c
  - Losowość bitów 2-6 wskaźnika stosu jądra
  - W dwóch miejscach:
    - tss->esp0 (wskaźnik stosu w Task State Segment)
    - current->thread.esp0 (do przeładowania powyższego przy przełączaniu kontekstu)

ASLR

+

NOEXEC


Heap/Data Segment
a=mmap();
ogg_in.so
b=mmap();
alsa_out.so
ld-linux.so
libpthread.so
Executable
libdl.so
mp3_in.so
Stack


Heap/Data Segment
ld-linux.so
libpthread.so
alsa_out.so
libdl.so
Executable
mp3_in.so
b=mmap();
a=mmap();
ogg_in.so
Stack

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

Pax – upadek projektu

1.04.2005

Oficjalne zamknięcie projektu PaX

Przyczyna:

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