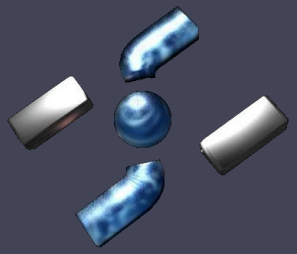


Hispasec

Praktyczne spojrzenie na luki bezpieczeństwa jądra Windows

… o podatnościach ring-0 słów kilka …

Mateusz „j00ru” Jurczyk
SEConference, Kraków 2010

Plan prezentacji

- Tryb jądra – co to jest?
  - Punkt widzenia aplikacji użytkownika
  - Punkt widzenia napastnika
- Bezpieczeństwo jądra
  - Podział pamięci wirtualnej
  - Komunikacja modułów o różnych uprawnieniach
  - Wektory ataku
  - Miejsca występowania potencjalnych luk
- Pytania

Kilka słów o mnie...

Mateusz „j00ru” Jurczyk

- Reverse Engineer @ Hispasec
- Pentester / Bughunter
- Vexillium (http://vexillium.org/)
- Autor bloga technicznego (http://j00ru.vexillium.org)

Tryb jądra – co to jest?

Architektura

- Jeden z poziomów uprzywilejowania kodu – Intel x86
- Dysponuje najwyższymi możliwymi uprawnieniami
  - Wyłączając mechanizmy wirtualizacji
- Wykorzystywany przez rdzeń systemu operacyjnego
  - Obsługa dostępnego sprzętu
  - Tworzenie odpowiedniego środowiska pracy dla aplikacji użytkownika

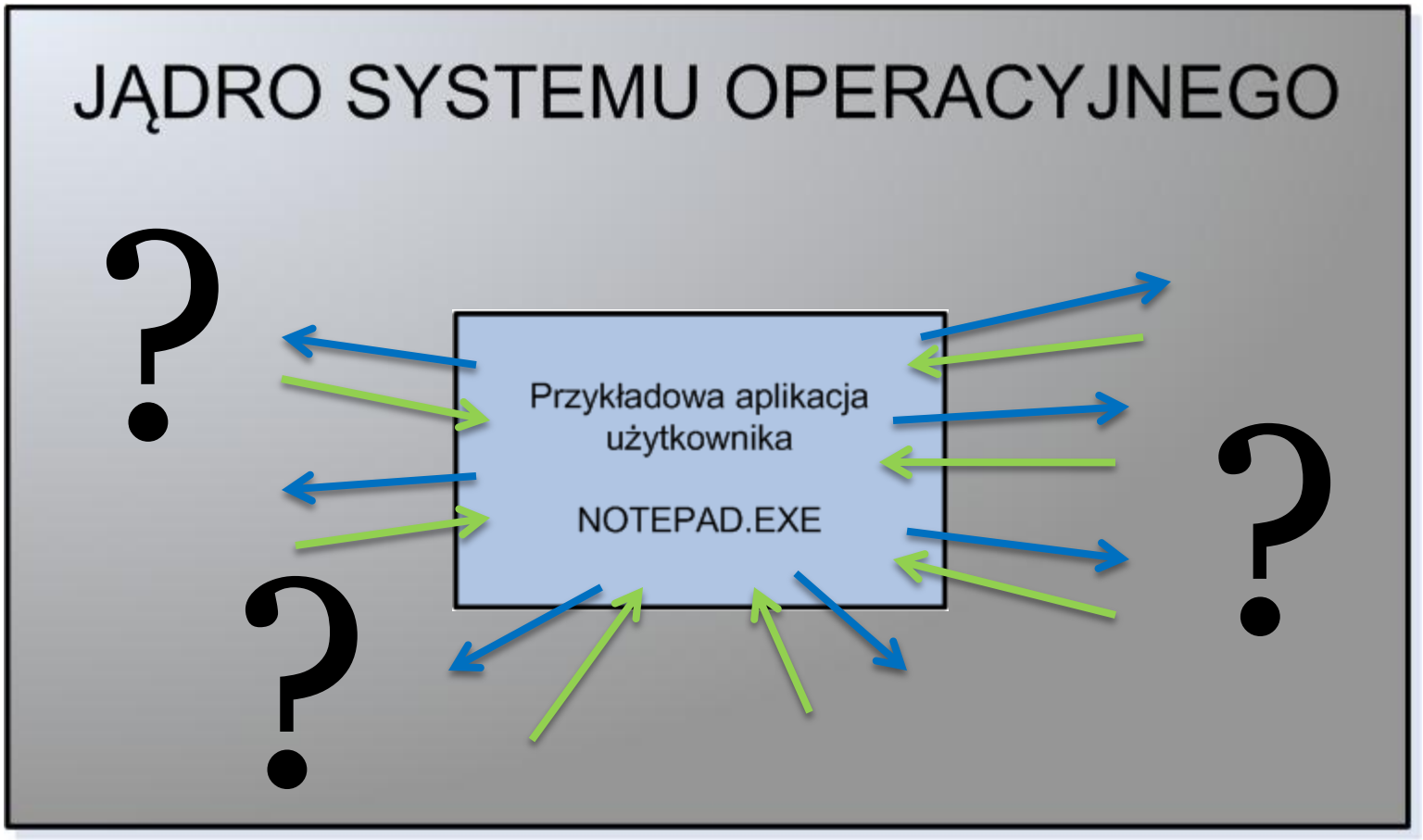
W praktyce – jądro w ring-0

- Zarządza dostępnymi zasobami komputera
  - Planuje i rozdziela czas procesora pomiędzy zadania
  - Zarządza pamięcią fizyczną oraz mechanizmem stronicowania
  - Zapewnia prawidłową obsługę i dostęp do zewnętrznych urządzeń

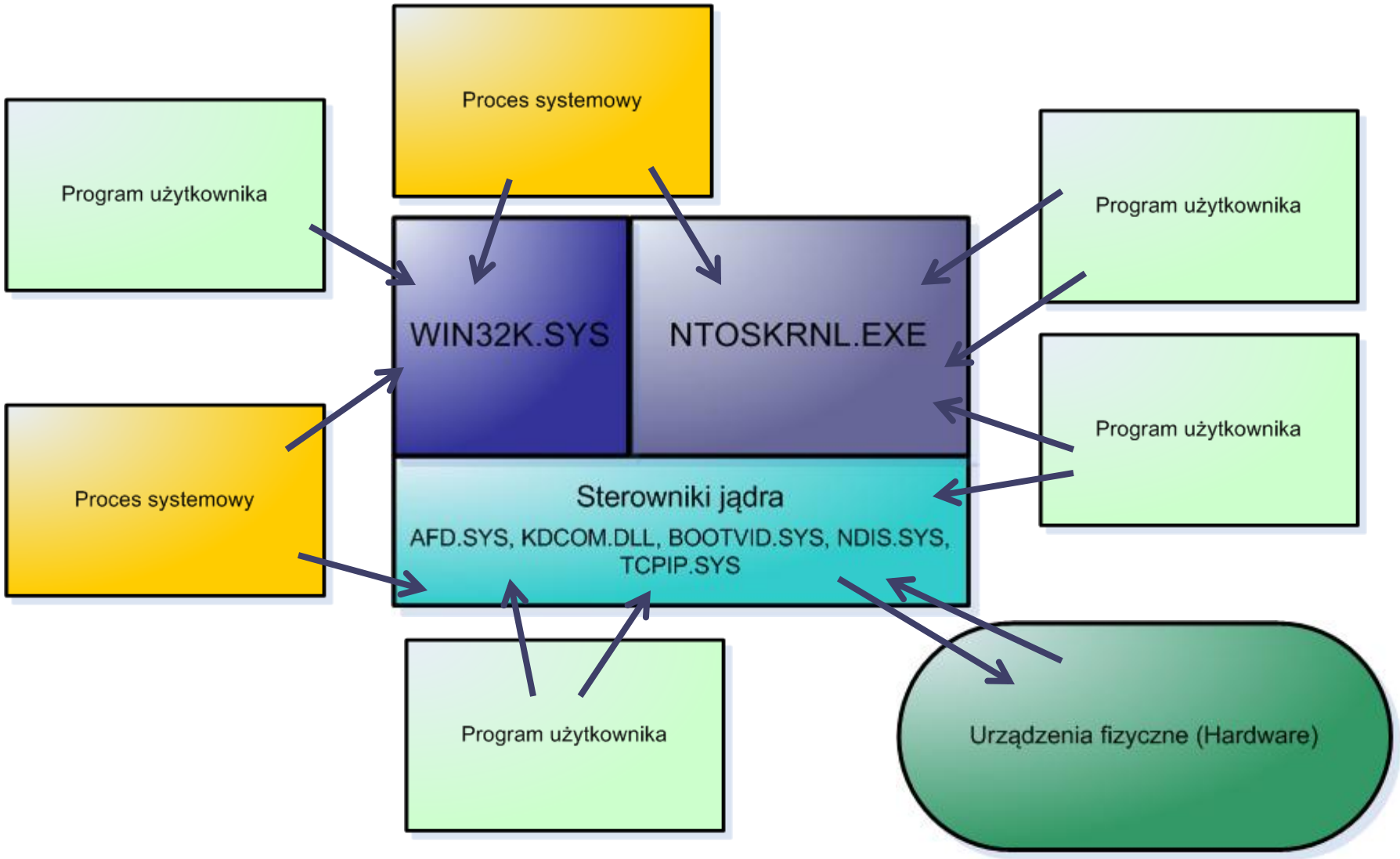
W praktyce – jądro w ring-0

- Udostępnia interfejs komunikacji jądra z zadaniami
  - Pozwala programom na wykonywanie określonych przez jądro operacji w systemie
  - Nadzoruje każdy istotny ruch, wykonywany przez aplikację
  - Operuje na ogromnej ilości danych dostarczanych przez uruchomione aplikacje oraz aktywne urządzenia

Punkt widzenia aplikacji użytkownika

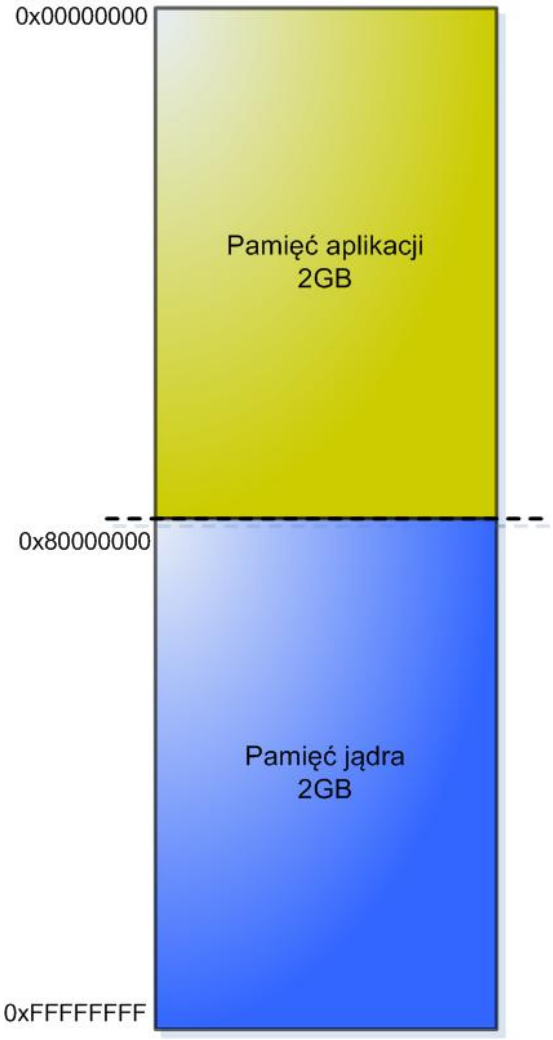


Punkt widzenia napastnika



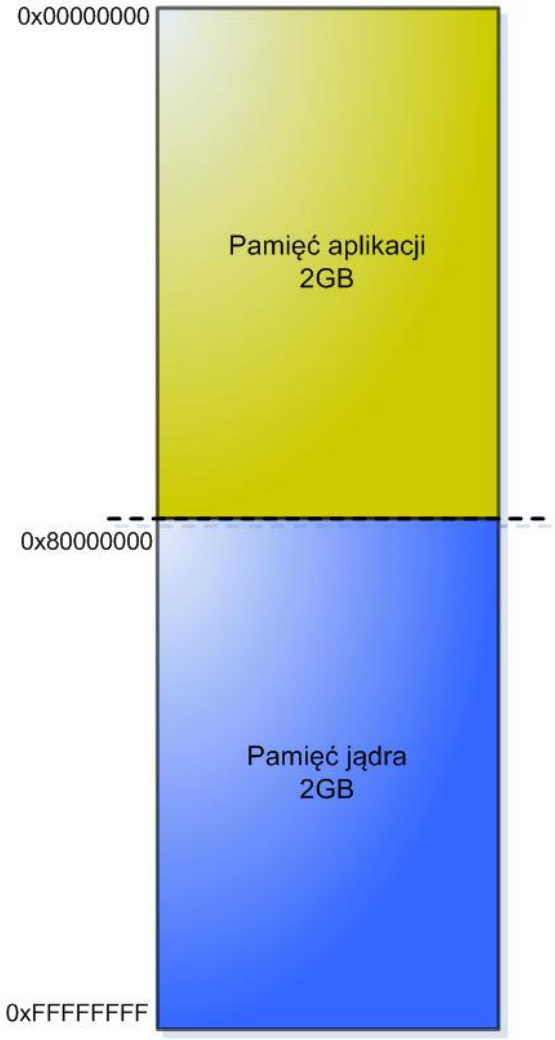
Bezpieczeństwo jądra

Podział pamięci wirtualnej



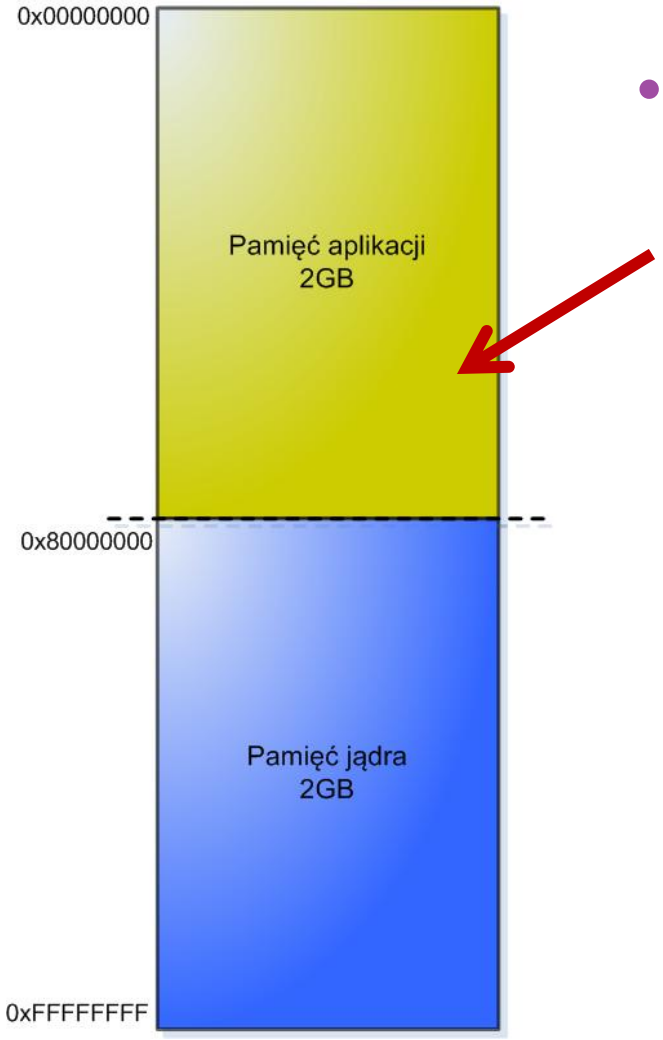
- Całkowita powierzchnia 4GB podzielona na dwie części

Podział pamięci wirtualnej



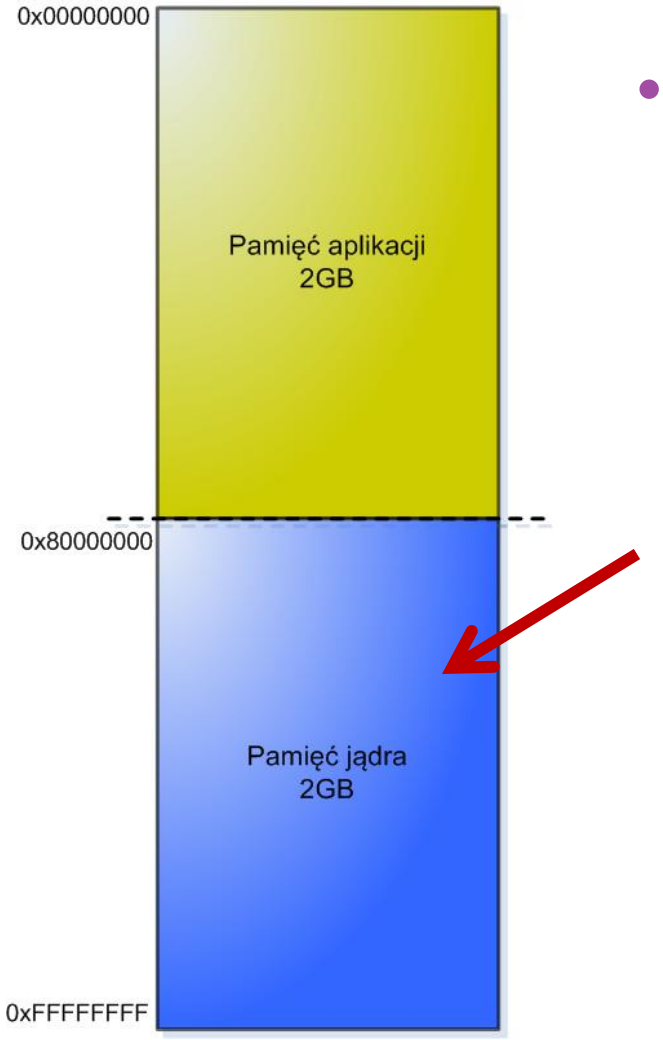
- Dolne obszary pamięci
  - Przechowują dane (kod, informacje) danego procesu
  - Regularnie zmieniają swoją postać (*context switch*)
  - Dostępne z każdego poziomu uprzywilejowania
  - Mogą zostać tymczasowo zapisane na dysk twardy (plik stronicowania)

Podział pamięci wirtualnej



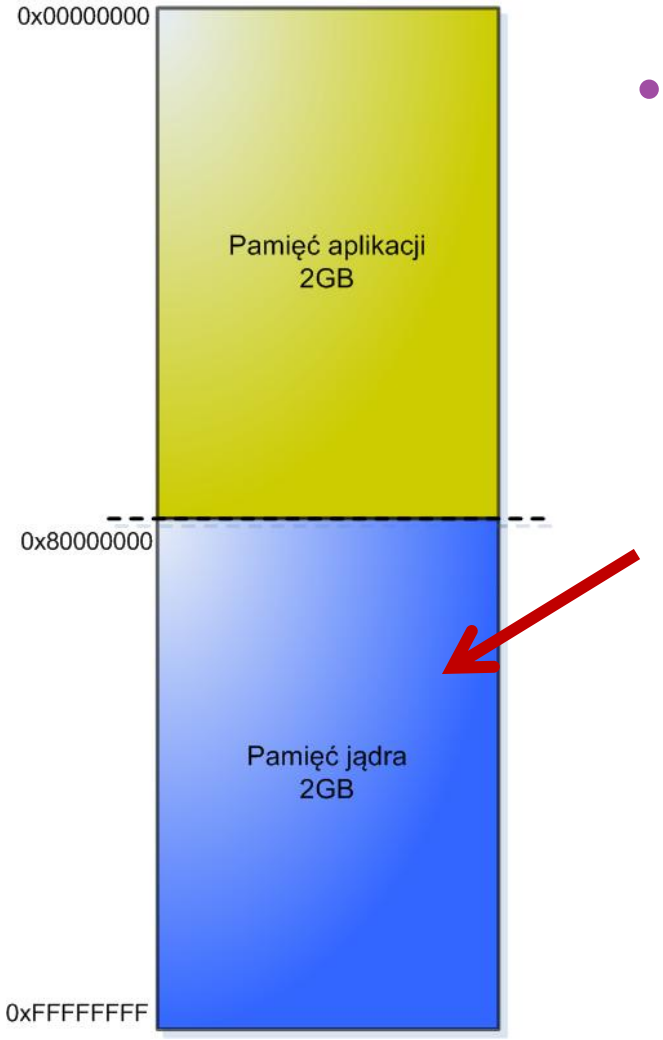
- Pamięć aplikacji zawiera
  - Obraz wykonywalny programu
  - Obrazy wykonywalne bibliotek zewnętrznych
  - Stosy oraz sterty procesu
  - Struktury systemowe
    - *Thread Environment Block*
    - *System Environment Block*
    - *KUSER_SHARED_DATA*
  - Prywatne dane programu

Podział pamięci wirtualnej



- Wysokie obszary pamięci
  - Nie ulegają zmianie w kontekście różnych procesów
  - Mogą, ale nie muszą zostać oznaczone jako PAGEABLE
  - Są współdzielone pomiędzy wszystkimi modułami jądra w systemie
  - Chronione przed odczytem i zapisem z poziomu *user-mode* (całkowity brak dostępu)

Podział pamięci wirtualnej



- Pamięć jądra zawiera:
  - Obraz wykonywalny jądra
  - Obrazy wykonywalne pozostałych sterowników
  - Stosy kernel-mode
  - Sterty (kernel memory pools)
  - Wewnętrzne listy i struktury jądra, opisujące obecny stan systemu

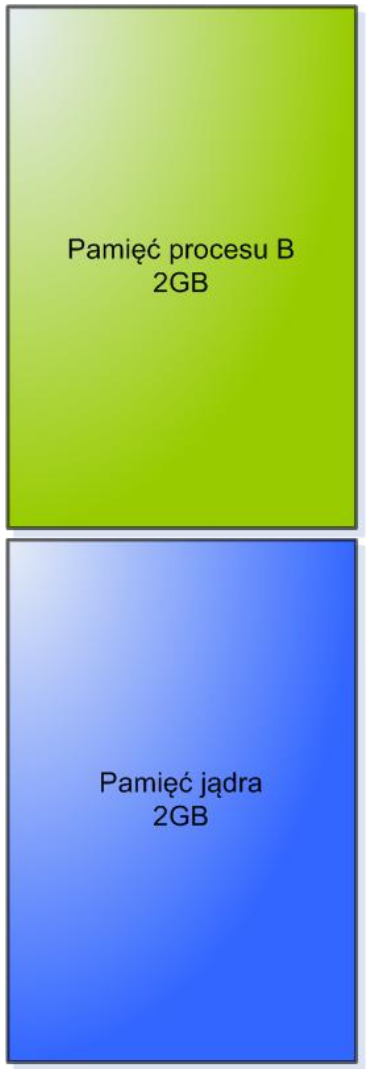
Podział pamięci wirtualnej

- Konkluzje
  - Kod z uprawnieniami ring-3 nie ma bezpośredniego dostępu do wysokich obszarów
  - Jeśli system jest bezpieczny, kod użytkownika nie nadpisze krytycznych struktur jądra
  - Wniosek – kod działający w *kernel-mode* może bezgranicznie ufać strukturom, umieszczonym w pamięci jądra

Podział pamięci wirtualnej

- Jedyny sposób, aby nadpisać pamięć trybu jądra z *user-mode* – zrobić to za pomocą aktywnego modułu jądra
- Stąd problem bezpieczeństwa – luki pozwalają na wykonywanie niedozwolonych operacji przez sterowniki, w imieniu aplikacji użytkownika

Pamięć wirtualna – context switch



1. Wątek 01 procesu A zostaje uruchomiony
2. Wątek 02 procesu B zostaje uruchomiony – zmiana kontekstu pamięci
3. Wątek 01 procesu B zostaje uruchomiony – brak zmiany kontekstu

- Wniosek
  - *Context switch* następuje jedynie w obrębie pamięci użytkownika, jeśli aktualny i kolejny wątek należą do różnych procesów

O wyjątkach słów kilka

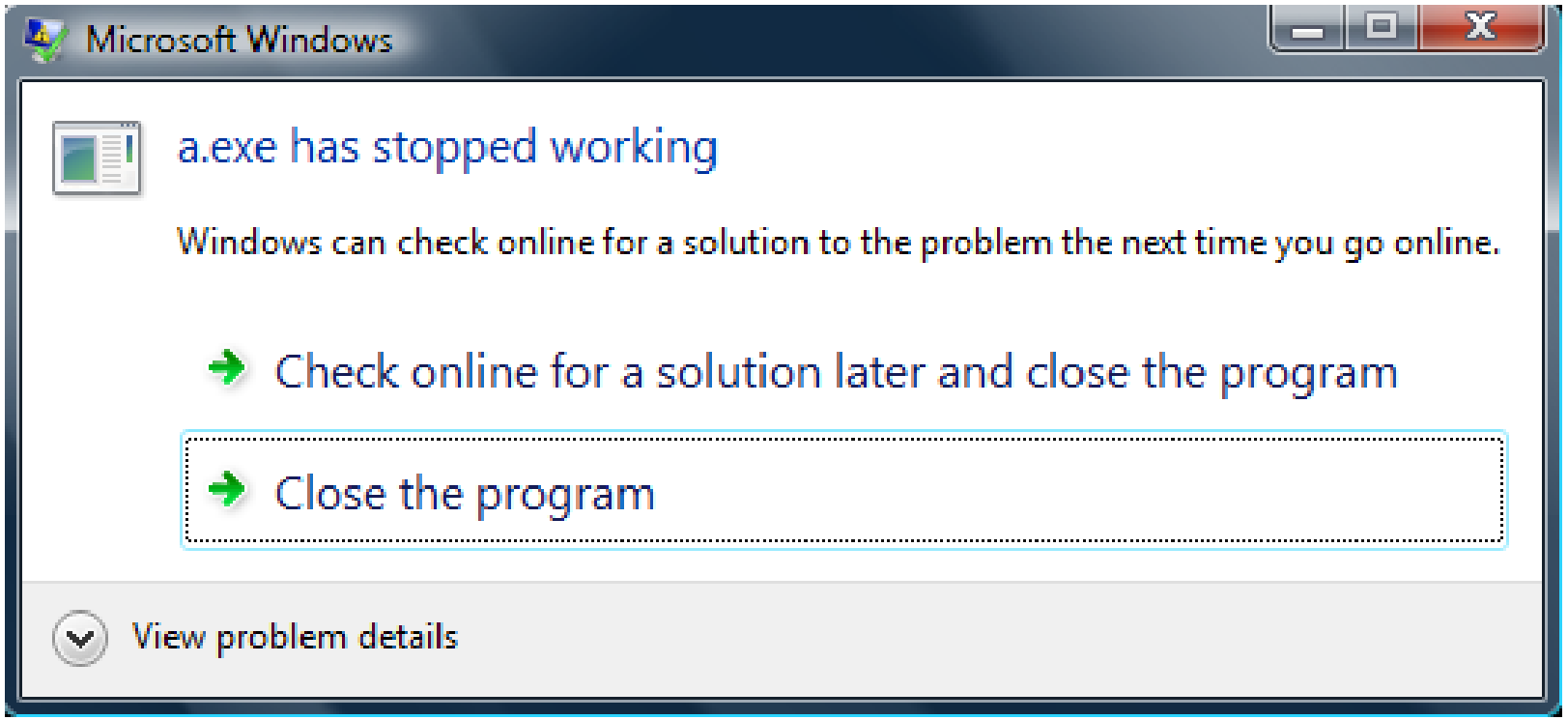
Tryb użytkownika

- Dwa mechanizmy obsługi wyjątków
  - *Structured Exception Handling*
  - *Vectored Exception Handling*
- Nieobsłużony wyjątek – *crash* aplikacji
- Brak wpływu na stabilność całego systemu

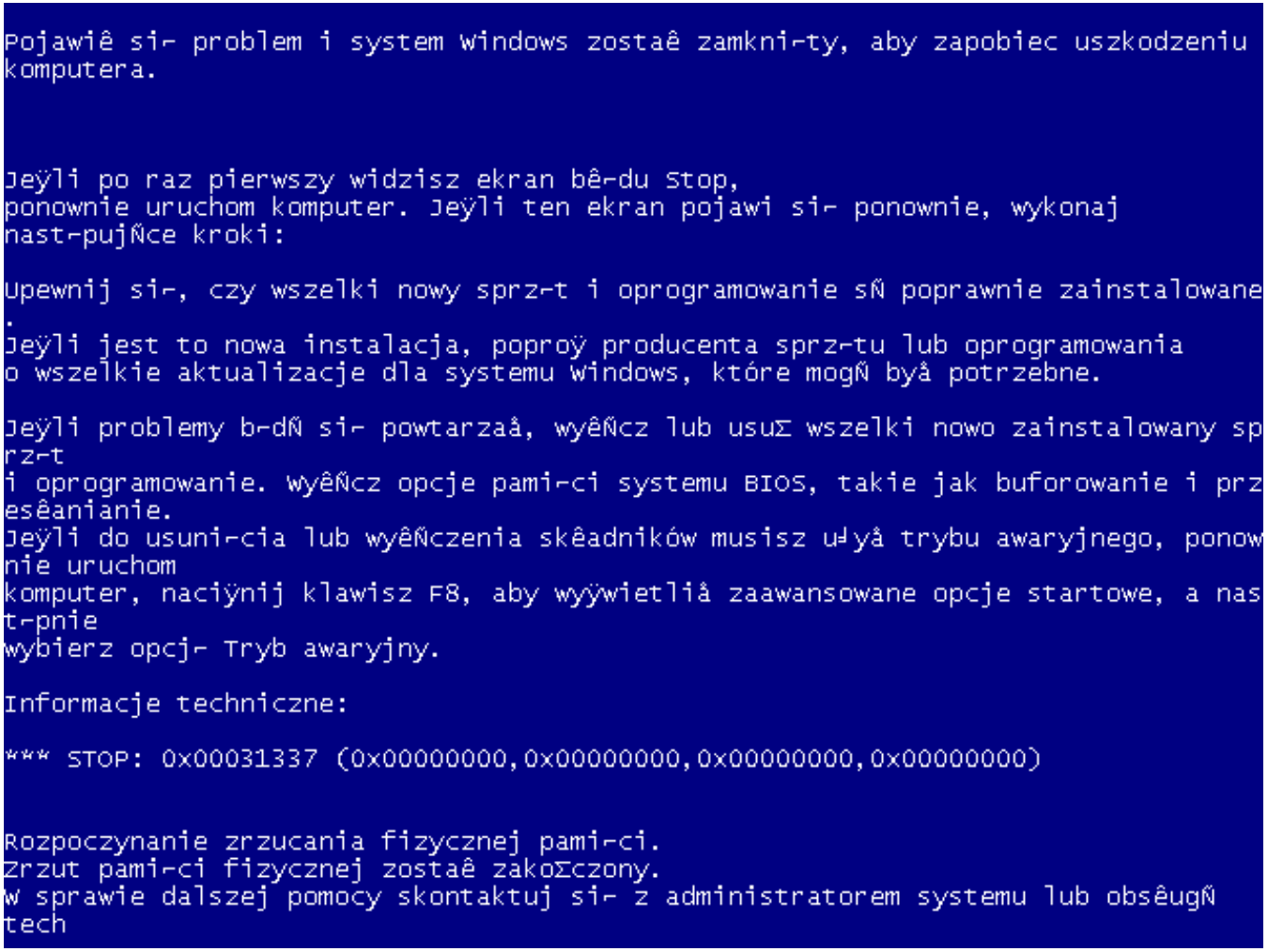
Tryb jądra

- Jeden mechanizm obsługi wyjątków
  - *Structured Exception handling*
- Nieobsłużony wyjątek – *Blue Screen of Death* oraz załamanie pracy systemu
- Bardziej ogólnie – luka typu *Denial of Service* (najczęściej lokalna)

Wyjątek aplikacji



Wyjątek jądra



Komunikacja z jądrem

Komunikacja z modułami jądra

- Wywołania systemowe (ang. *system calls*)
- Przerwania systemowe (ang. *interrupts*)
- IOCTL (ang. *Device Input and Output Control*)
- Urządzenia fizyczne – hardware

Wywołania systemowe

- Zbiór kilkuset funkcji jądra systemu, wywoływane z *user-mode*
- Odpowiedzialne za wykonywanie (prawie) wszystkich operacji w imieniu programu
- Obsługiwane przez dwa moduły
  - ntoskrnl.exe (wywołania bazowe)
  - win32k.sys (wywołania graficzne)
- Przykładowe funkcjonalności
  - Pliki
  - Rejestr
  - Procesy, wątki
  - Informacje o systemie

Wywołania systemowe

- Często *hookowane* (!)
  - Malware
  - „Uprawnione” programy AV
  - Cel – monitorowanie i filtracja danych przepływających pomiędzy aplikacją a jądrem
- Dwie metody wywołania
  - Stary – *INT 0x2e (wciąż wspierany!)*
  - Nowy – instrukcja *SYSENTER / SYSCALL*

ntoskrnl.exe vs win32k.sys

Jądro systemu

- Blisko 300 wywołań systemowych
- Numery identyfikacyjne <= 0x1000
- Dobrze usystematyzowane nazewnictwo funkcji
- Używane przez każdy istniejący w systemie proces

Sterownik graficzny

- Ponad 600 wywołań systemowych
- Numery identyfikacyjne > 0x1000
- Słaba systematyka nazw
- Używane wyłącznie przez procesy korzystające z trybu graficznego

Przerwania systemowe

- Zbiór pięciu przerwań, których może użyć aplikacja
  - KiGetTickCount (0x2a)
  - KiCallbackReturn (0x2b)
  - KiRaiseAssertion (0x2c)
  - KiDebugService (0x2d)
  - **KiSystemService (0x2e)**
- Wywoływane instrukcją *int*, parametryzowane za pomocą rejestrów ogólnego przeznaczenia

IRP / IOCTL

- Udokumentowane metody komunikacji ze sterownikami
  - Wyłącznie posiadającymi globalną nazwę, np. *\Global\MyDriver*
- Pozwalają na wykonywanie zapytań i odbieranie odpowiedzi w formie buforów we/wy

	I/O request packet	Device Input and Output Control
Otwieranie	CreateFile	CreateFile
Odczyt	ReadFile	DeviceIoControl
Zapis	WriteFile	

Wnioski

- Zarówno rdzeń systemu, jak większość dodatkowych sterowników operuje na dużej ilości danych „z zewnątrz”
- Co więcej, istnieje wiele sposobów komunikacji z jądrem – a więc wiele potencjalnych luk

Wnioski

Wszystkie moduły jądra mają równe uprawnienia

- Zachwianie bezpieczeństwa dowolnego z nich umożliwia wykonanie ataku na cały system

Pułapki trybu jądra

Pułapki trybu jądra

- Brak zaufania do otrzymywanych danych
  - Adresy wirtualne (wskaźniki)
  - Struktury kontrolowane przez użytkownika
  - Wartości zwracane przez wewnętrzne funkcje
- Zasada – nie zakładać nic o wartościach, nad którymi nie mamy kontroli

Pułapka pierwsza – pamięć wirtualna

- Dostęp do pamięci
  - Konieczność weryfikacji zakresu adresu
    - Funkcje *ProbeForRead* i *ProbeForWrite*
  - Obszar *user-mode*
    - Nieistniejąca strona (*Access Violation*)
    - Nieprawidłowe uprawnienia strony (*Access Violation*)
    - Błędne wyrównanie (?)
    - Dynamiczna zawartość pamięci (*Race Condition*)

Weryfikacja adresu

- Aplikacja użytkownika **nigdy** nie powinna używać adresu jądra jako argumentu wywołania
- W przeciwnym wypadku:
  - Bufor wejściowy → *Memory Disclosure*, potraktowanie sekretnych danych jądra jako informacji wejściowych
  - Bufor wyjściowy → *Write-what-where*, wykonanie zapisu pod adres jądra, zawierający krytyczne dane

Weryfikacja adresu

```
NTSTATUS
IOCTLHandler(
  PVOID InBuffer,
  PVOID OutBuffer,
  ULONG InBufferSize,
  ULONG OutBufferSize
)
{
  struct OBJECT obj;

  if((struct OBJECT*)InBuffer->dwSize > 8)
    return STATUS_INFO_LENGTH_MISMATCH;

  /* … */

  memcpy(&obj,InBuffer,InBufferSize);
  return STATUS_SUCCESS;
}
```

Weryfikacja adresu

Sprawdzenie rozmiaru bufora
(*buffer overflow*)

Weryfikacja adresu

```
{
  struct OBJECT obj;

  if(InBufferSize != sizeof(struct OBJECT))
    return STATUS_INVALID_PARAMETER;

  if((struct OBJECT*)InBuffer->dwSize > 8)
    return STATUS_INFO_LENGTH_MISMATCH;

  /* … */

  memcpy(&obj,InBuffer,InBufferSize);
  return STATUS_SUCCESS;
}
```

Weryfikacja adresu

Wstępne wywołanie *ProbeForRead* / *ProbeForWrite* (zakres adresu)

Weryfikacja adresu

```
{
  struct OBJECT obj;

  if(InBufferSize != sizeof(struct OBJECT))
    return STATUS_INVALID_PARAMETER;

  ProbeForRead(InBuffer,InBufferSize,sizeof(BYTE));

  if((struct OBJECT*)InBuffer->dwSize > 8)
    return STATUS_INFO_LENGTH_MISMATCH;

  /* … */

  memcpy(&obj,InBuffer,InBufferSize);
  return STATUS_SUCCESS;
}
```

Weryfikacja adresu

Ograniczenie kodu operującego na wskaźniku znacznikami

try{} except(){} (race condition)

Weryfikacja adresu

```
{
  struct OBJECT obj;

   __try
   {
     if(InBufferSize != sizeof(struct OBJECT))
       return STATUS_INVALID_PARAMETER;

     ProbeForRead(InBuffer,InBufferSize,sizeof(BYTE));

    if((struct OBJECT*)InBuffer->dwSize > 8)
      return STATUS_INFO_LENGTH_MISMATCH;

    /* … */

    memcpy(&obj,InBuffer,InBufferSize);
  } except(EXCEPTION_EXECUTE_HANDLER)
  {
    return GetExceptionCode();
  }
  return STATUS_SUCCESS;
}
```

Weryfikacja adresu

Kopiowanie zawartości bufora do pamięci jądra
(race conditon)

Weryfikacja adresu

```
{
  struct OBJECT obj;

   __try
   {
     if(InBufferSize != sizeof(struct OBJECT))
       return STATUS_INVALID_PARAMETER;

    ProbeForRead(InBuffer,InBufferSize,sizeof(BYTE));

    memcpy(&obj,InBuffer,InBufferSize);
    if(obj.dwSize > 8)
      return STATUS_INFO_LENGTH_MISMATCH;

    /* … */
  } except(EXCEPTION_EXECUTE_HANDLER)
  {
    return GetExceptionCode();
  }
  return STATUS_SUCCESS;
}
```

Pamięć wirtualna – c.d.

- Niezainicjalizowane wskaźniki zagrożeniem?
  - Domyślnie wskazują na pierwszą stronę pamięci użytkownika (NULL = 0x00000000)
  - Nieświadome użycie adresu zerowego powoduje pobranie danych zaufanych danych od użytkownika

Nieinicjalizowane wskaźniki

- *Microsoft Windows Kernel NULL Pointer Dereference Local Privilege Escalation Vulnerability*
  - BID: 36939 (Windows 2000-Vista)
  - BID: 36624 (Windows 2000-Vista)
- *Microsoft Windows SMB Null Pointer Remote Denial of Service Vulnerability*
  - BID: 38051 (Windows XP – Seven)

Błędne wskaźniki – c.d.

```
NTSTATUS(*DispatchRoutine)(VOID);

NTSTATUS
IOCTLHandler()
{
  DispatchRoutine = RealFunction;
  return STATUS_SUCCESS;
}

NTSTATUS
SecondHandler()
{
  return DispatchRoutine();
}
```

Błędne wskaźniki po raz ostatni

- Czy adres zerowy może wystąpić w innej sytuacji, niż niezainicjalizowana zmienna?
- **TAK!**

- Przykład:
  - *ExAllocatePool* zwracająca *NULL* w przypadku błędu alokacji
  - Brak weryfikacji zwróconego adresu → nieświadome odwołania do pamięci użytkownika

Błędne wskaźniki po raz ostatni

- Kiedy alokacja się nie powiedzie?
  - Rozmiar pożądanego obszaru pamięci przekracza rozmiar obszaru jądra, np. *ExAllocatePool(oxDEADBEEF);*
  - Jądro nie dysponuje wystarczającą ilością wolnego miejsca – sytuacja „zapełnionej” sterty
  - Został wyczerpany limit pamięci dla konkretnego procesu – *pool quota*

Błędne wskaźniki po raz ostatni

- Wniosek
  - Nigdy nie zakładajmy, że dynamiczna alokacja kończy się powodzeniem – każdy wynik rezerwacji miejsca powinien być weryfikowany
  - W przeciwnym wypadku operujemy na obszarach pamięci kontrolowanych przez użytkownika

Błędne wskaźniki po raz ostatni

```
NTSTATUS
IOCTLHandler(
  PVOID InBuffer,
  PVOID OutBuffer,
  ULONG InBufferSize,
  ULONG OutBufferSize
)
{
  BYTE* Pointer = ExAllocatePool(PagedPool,InBufferSize);
  memcpy(Pointer,DecryptionKey,16); // klucz prywatny
  return DecryptData(InBuffer,InBufferSize,Pointer);
}
```

Problemy z liczbami całkowitymi

- Integer overflow
  - Jedna z najczęściej spotykanych klas błędów
  - Polega na „przepełnieniu” wartości zmiennej całkowitoliczbowej
  - Niezwykle istotne w przypadku obliczania rozmiaru bufora (alokacja dynamiczna)
  - Przyczyna: potencjalnie dowolna operacja arytmetyczna

Problemy z liczbami całkowitymi

- Przykład:

```
UINT a = 0x20000000;
UINT b = 0x38000000;
BYTE* Buffer = ExAllocatePool(a+b*4);
for( UINT i=0;i<a;i++ )
  Buffer[i] = 'A';
```

Problemy z liczbami całkowitymi

- Funkcja *ExAllocatePool* została wywołana a z argumentem 0 (zero)
- Alokacja powiodła się – został zarejestrowany najmniejszy możliwy rozmiar alokacji – 8 bajtów
- W pętli wypełniającej bufor danymi następuje przepełnienie

Problemy z liczbami całkowitymi

- Wariacje popularnych błędów
  - *8/16-bit integer wrap* – alokacja przy użyciu dolnych bitów rozmiaru

Problemy z liczbami całkowitymi

```
NTSTATUS
IOCTLHandler()
{
  const DWORD Size = 0x123456;
  BYTE* Pointer;

  Pointer = ExAllocatePool(PagedPool,(WORD)Size);
  if(Pointer == NULL)
      return STATUS_INSUFFICIENT_RESOURCES;

  memset(Pointer,0,Size);
  return STATUS_SUCCESS;
}
```

Problemy z liczbami całkowitymi

- Wariacje popularnych błędów
  - *8/16-bit integer wrap* – alokacja przy użyciu dolnych szesnastu bitów rozmiaru
  - *Off-by-one*

Problemy z liczbami całkowitymi

```
NTSTATUS
IOCTLHandler()
{
  static BYTE Buffer[256];
  UINT i;

  for( int i=0;i<=sizeof(Buffer);i++ )
    Buffer[i] = 'A';

  return STATUS_SUCCESS;
}
```

Problemy z liczbami całkowitymi

- Wariacje popularnych błędów
  - *8/16-bit integer wrap* – alokacja przy użyciu dolnych szesnastu bitów rozmiaru
  - *Off-by-one*
  - Wyjątki procesora – działania arytmetyczne
    - Dzielenie przez zero
    - Dzielenie INT_MIN przez -1
  - Brak weryfikacji poprawności indeksu tablicy

Problemy z liczbami całkowitymi

```
NTSTATUS
IOCTLHandler(
  LPDWORD InBuffer
  ULONG InBufferSize)
{
  DWORD Table[4096];
  UINT i;

  /* walidacja bufora wejściowego */

  for( i=0;i<InBufferSize/sizeof(DWORD);i++ )
  {
      Table[InBuffer[i]]++;
  }

  return STATUS_SUCCESS;
}
```

Przepełnienia bufora

- Oparte o pamięć stosu
  - W dzisiejszych czasach – praktycznie niespotykane
  - Powody
    - Zabezpieczenia stosu (/GS)
    - Używanie bezpiecznych funkcji obsługi tekstu
      - strcpy_s, wcscpy_s, _mbscpy_s
  - Chociaż kiedyś się zdarzały ☺
    - *Microsoft Windows Kernel Message Handling Buffer Overflow Vulnerability*
    - BID: 7370 (Windows 2000 – Windows XP SP1)

Przepełnienie bufora

- Oparte o pamięć sterty
  - Bardzo często spotykane
  - Powody
    - Łatwość wykorzystania – brak ochrony sterty
    - Wewnętrzna struktura podobna do sterty *user-mode*
    - Konsekwencja popularnego *integer overflow*

Przepełnienie bufora

- Oparte o pamięć sterty c.d.
  - Poprawne wykorzystanie → 4-*byte write-what-where condition*
  - Po nadpisaniu → *arbitrary code execution*

Przepełnienie bufora

- Oparte o pamięć sterty c.d.
  - Dlaczego tak łatwo?
    - Całkowity brak zabezpieczeń w mechanizmie *Kernel Pools* – włącznie z Windows Vista
    - Windows 7 – po raz pierwszy wprowadzono weryfikację o nazwie *Safe Unlinking*, zapobiegającej większości ataków

Uwagi końcowe

Uwagi końcowe

- Jądro Windows – cel ataku
  - Możliwość przejęcie całkowitej kontroli nad systemem
  - Mnogość metod wymiany informacji – potencjalnych furtek dla napastnika
  - Różnorodność możliwych ataków
    - Poziom weryfikacji danych i wskaźników wejściowych
    - Poziom właściwej synchronizacji
    - Poziom logiki konkretnych mechanizmów jądra

Uwagi końcowe

- Dziesiątki modułów otwartych na komunikację z aplikacją użytkownika
  - Każdy z nich potencjalnym zagrożeniem…
  - … a zwłaszcza …
  - STEROWNIKI FIRM TRZECICH

Uwagi końcowe

- Pojawienie się niespodziewanego wyjątku w trybie *ring-0* – załamanie pracy systemu
  - A więc prosty atak typu DoS
  - Szczególne niebezpieczne w przypadku systemów serwerowych

Uwagi końcowe

- Konieczna ostrożność przy każdej operacji
  - Działanie arytmetyczne
  - Odwołanie do pamięci użytkownika
  - Odwołanie do tablicy o niekontrolowanym indeksie
- Sprawdzanie zwracanych wartości
  - Alokacja pamięci – *ExAllocatePool*
- Brak jednego z powyższych – wysoce prawdopodobne zagrożenie dla systemu

Pytania

Dziękuję za uwagę!

Q & A

E-mail: j00ru.vx@gmail.com
Tech blog: http://j00ru.vexillium.org/