From AMSI to Reflection 0x0

rxored.github.io/post/csharploader/bypassing-amsi-with-csharp

2021-10-23

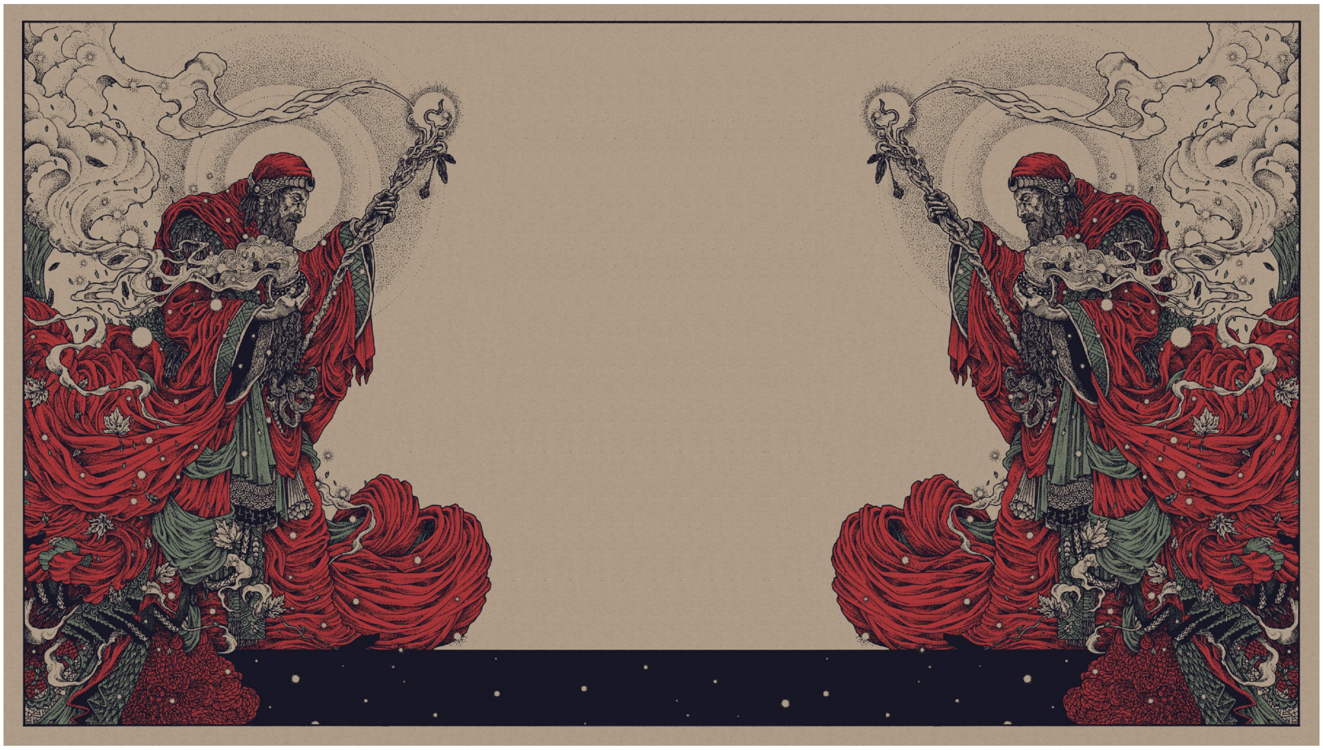


Table of Content

Introduction

In Windows environments, in both initial access and post-exploitation phases, script-based malware plays a major role. Often, hackers utilize microsoft office suite to gain initial access (using droppers, loaders) to the victim and Windows powershell to explore internal network, perform scans... basically to do the post exploitation stuff. (well of course, there are powershell based droppers.)

There is something that is common to both of these tools. Windows scripting engine.

And as a result, Microsoft and antimalware vendors have developed many security mechanisms to deal with those threats that utilize script-based malware. For example, modern anti-malware solutions can statically analyze scripts, binaries and detect whether they are malicious or not using signatures such as strings.

And because of that, malware authors use various techniques to bypass those defense mechanisms. One of the major techniques is code obfuscation.

consider the following example, that I took from MSDN.

```
function displayEvilString
{
    Write-Host 'pwnd!'
}
```

Assuming the above PowerShell snippet is malicious, we can write a signature to detect the malware. this signature can be `Write-Host 'pwnd!'` or simply `'pwnd!'`.

So to avoid signature-based detection, the above snippet can be obfuscated like shown below.

```
function obfuscatedDisplayEvilString
{
    $xorKey = 123
    $code = "LHsJexJ7D3see1Z7M3sUewh7D3tbe1x7C3sMexV7H3tae1x7"
    $byte = [Convert]::FromBase64String($code)
    $newBytes = foreach($byte in $bytes) {
        $byte -bxor $xorKey
    }
    $newCode = [System.Text.Encoding]::Unicode.GetString($newBytes)
}
```

And this is a win for malware authors since this is beyond what anti-malware solutions can emulate or detect until AMSI joins the conversation.

Antimalware Scan Interface

Antimalware Scan Interface, AMSI for short is a standard interface that allows applications to interact with anti-malware products installed on the system. This means is that it provides an API for Application developers. Application developers can use the API to implement security features to make sure that the end-user is safe.

AMSI also enables anti malware vendors to defend againts script based malware.

According to Microsoft, AMSI provides the following features by default.

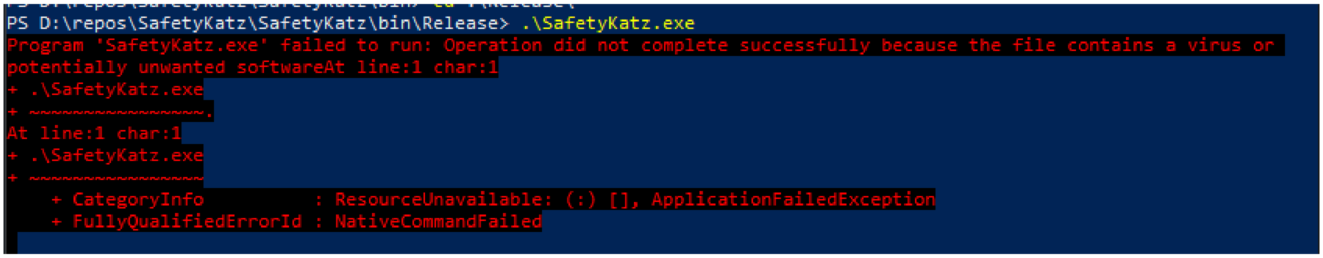
```
-   User Account Control
-   PowerShell
-   Windows Script Host
-   JScript && VBScript
-   Office VBA macros
```

As it is clear from those default features, AMSI specifically provides anti-malware security mechanisms to defend against script-based malware.

AMSI in action

So let’s take Safetykatz as our example.

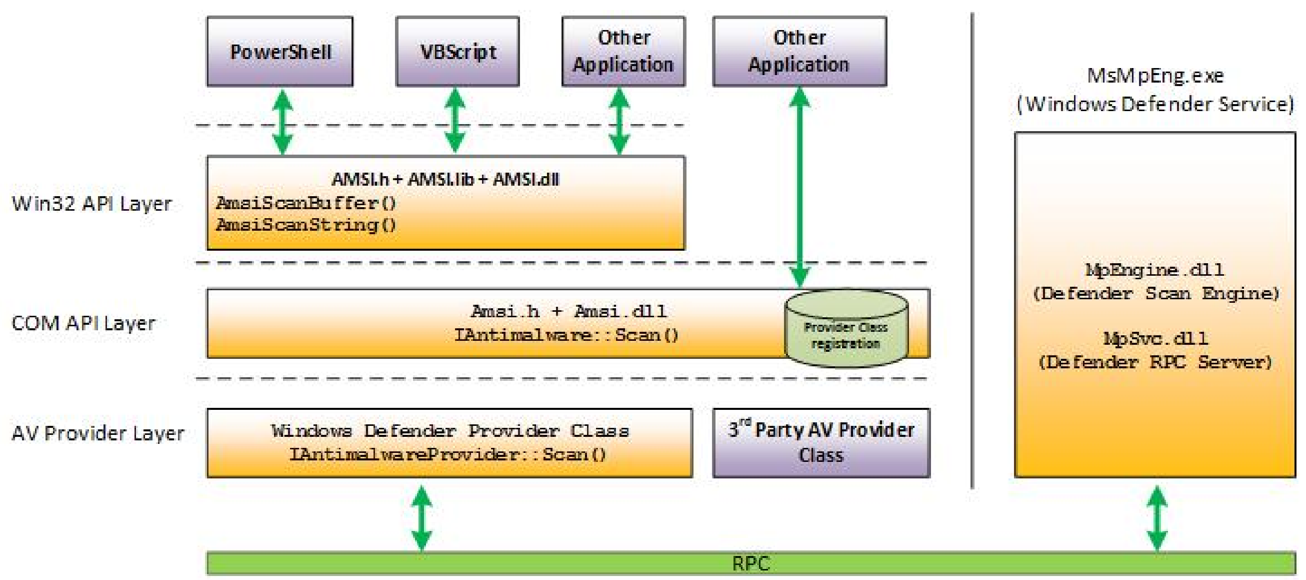
When we run the binary, the result we get is.



See, as we expected, PowerShell stops the execution of the program once it has detected the program is suspicious using AMSI. So, how can we bypass this?, well before that, we have to dive deep into AMSI internals to understand how things work.

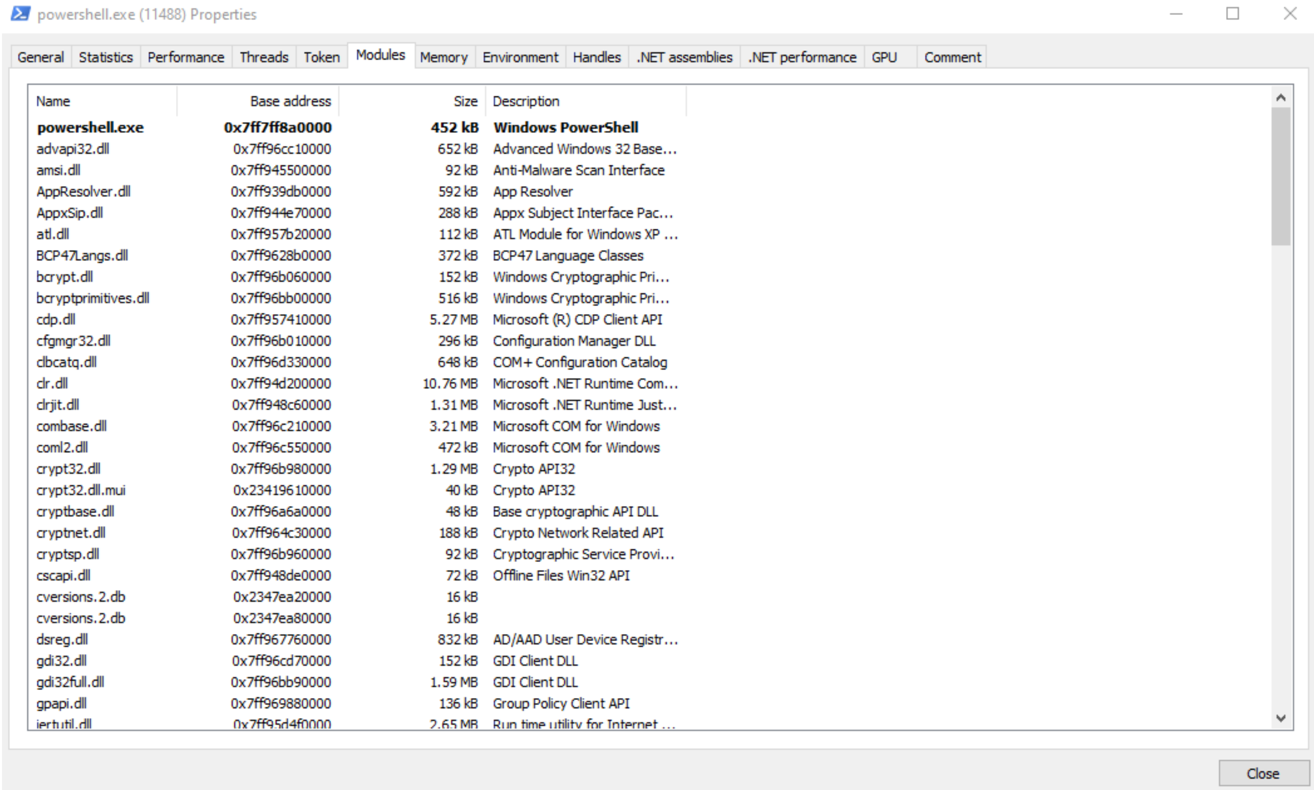
AMSI internals

As I previously mentioned, AMSI enables anti malware vendors to defend againts script based malware. This is done by using AMSI providers. An AMSI provider is basically a COM object that implements `IAntimalwareProvider` COM interface. An anti malware vendor who's willing to implement AMSI interface should then register the COM object by creating a CLSID entry in `HKLM\CLSID` and registering the same CLSID under `HKLM\Software\Microsoft\AMSI\Providers\`.



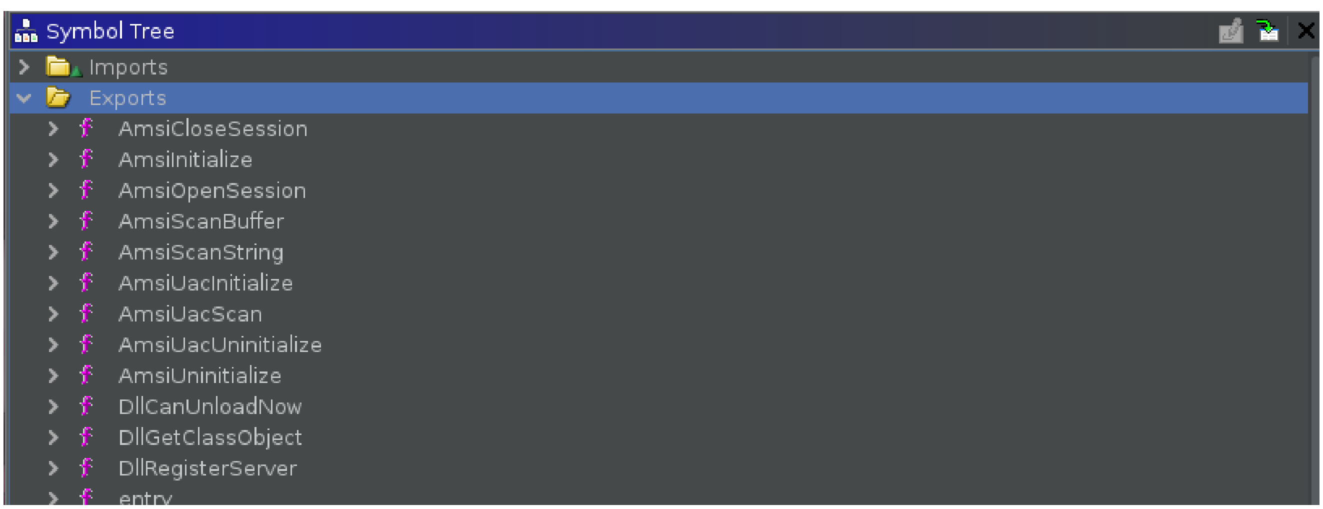
As it is shown in the above diagram, AMSI provides a dll called `amsi.dll` for application developers to interfere with AMSI providers indirectly.

Let's examine PowerShell from process hacker to check whether amsi.dll is loaded.



as we can see, amsi.dll has been loaded into powershell.exe. Now, let's take a look at this dll in-depth and see if we can find anything interesting. Even without looking at the dll, it is possible to think of some techniques to bypass AMSI, Anyway, its time to dig deep.

Before start reading disassembly, let's examine the export table of amsi.dll.



Out of the above exported functions, only two are important to us.

```
-   AmsiInitialize
-   AmsiScanBuffer
-   AmsiScanString
```

Of course there are some other important exports. To name a few, `DllRegisterClass`, `DllGetClassObject` and `AmsiUacScan`.

First we'll go through AmsiScanBuffer.

AmsiScanString

Microsoft documentation does not tell us much about AmsiScanString function. However it gives some basic information about it. Such as,

it's prototype,

```
HRESULT AmsiScanString(
  [in]           HAMSICONTEXT amsiContext,
  [in]           LPCWSTR      string,
  [in]           LPCWSTR      contentName,
  [in, optional] HAMSISESSION amsiSession,
  [out]          AMSI_RESULT  *result
);
```

and parameter information.

According to the documentation, The first parameter this function accepts is `amsiContext`, which is a handle of type `HAMSICONTEXT` that was initially received from AmsiInitialize.

Second and third parameters hold pointers to wide character strings. first one for the string that should be scanned and the latter for the `contentName`.

`contentName` can be either filename, script id, url or similar of the content being scanned.

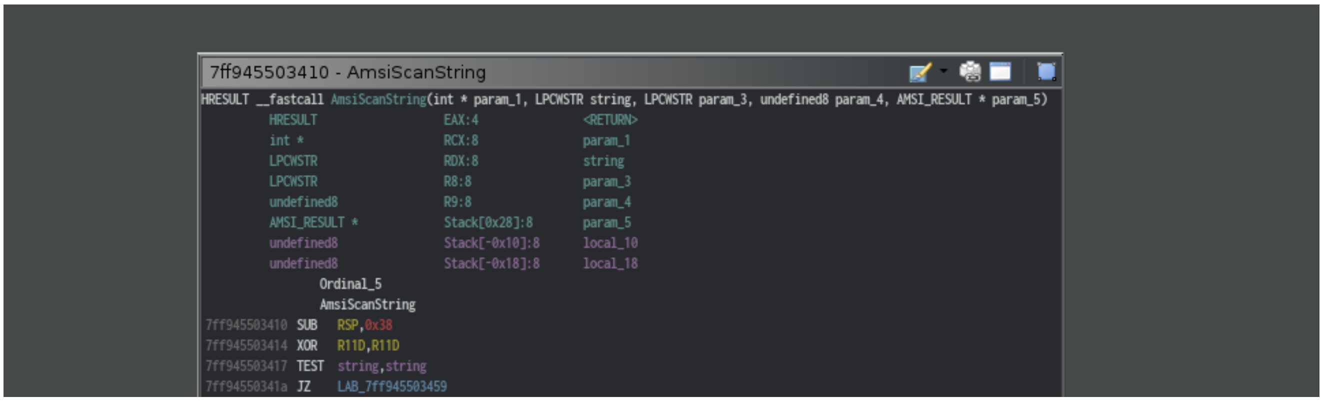
Fourth parameter is marked optional, however if multiple scan requests are to be correlated within a session, this parameter should be set to the handle returned by `AmsiOpenSession` function.

Fifth parameter is an output parameter and this is the one that indicates whether the input string is malicous or not.

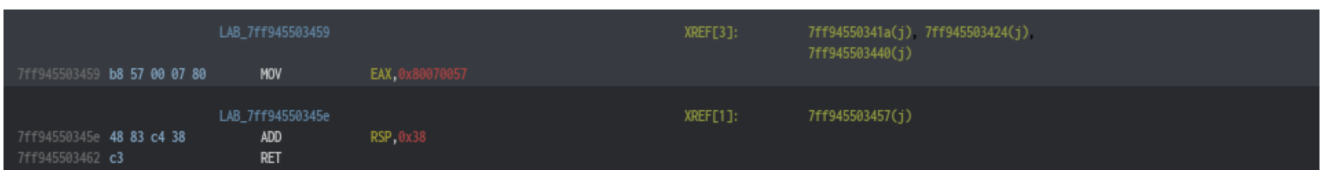
As MSDN says, this function (and AmsiScanBuffer) returns `S_OK` if the call is successful. However, the return value does not indicate whether the buffer is malicious. instead, the function uses fifth parameter of type `AMSI_RESULT` to send the scan results to caller.

```
typedef enum AMSI_RESULT {
    AMSI_RESULT_CLEAN,
    AMSI_RESULT_NOT_DETECTED,
    AMSI_RESULT_BLOCKED_BY_ADMIN_START,
    AMSI_RESULT_BLOCKED_BY_ADMIN_END,
    AMSI_RESULT_DETECTED
} ;
```

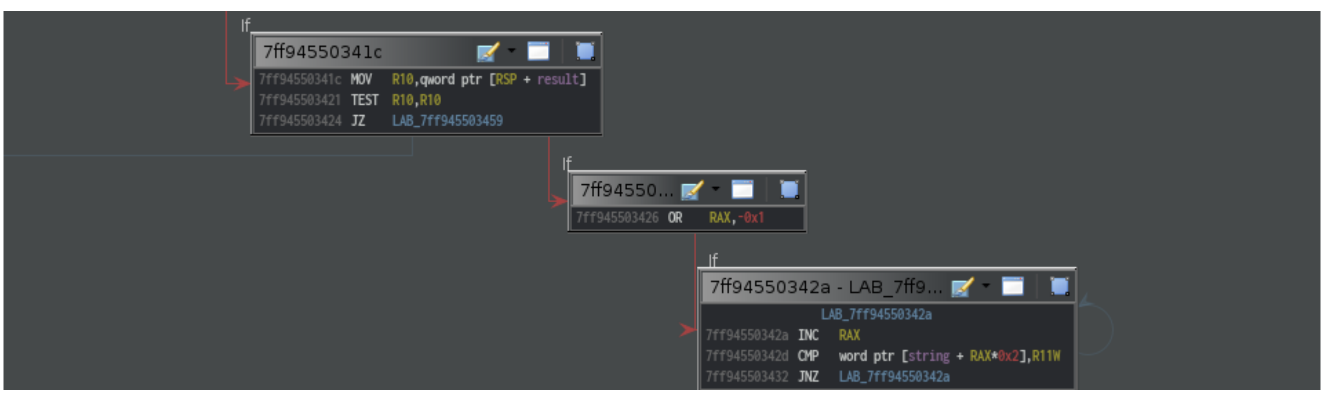
Let's a take a look at `AmsiScanString` in disassembly.



Function allocates some space in the stack and checks if the string is empty or not. If `string` turns out to be empty, it simply returns after loading `0x80070057` into `rax` .

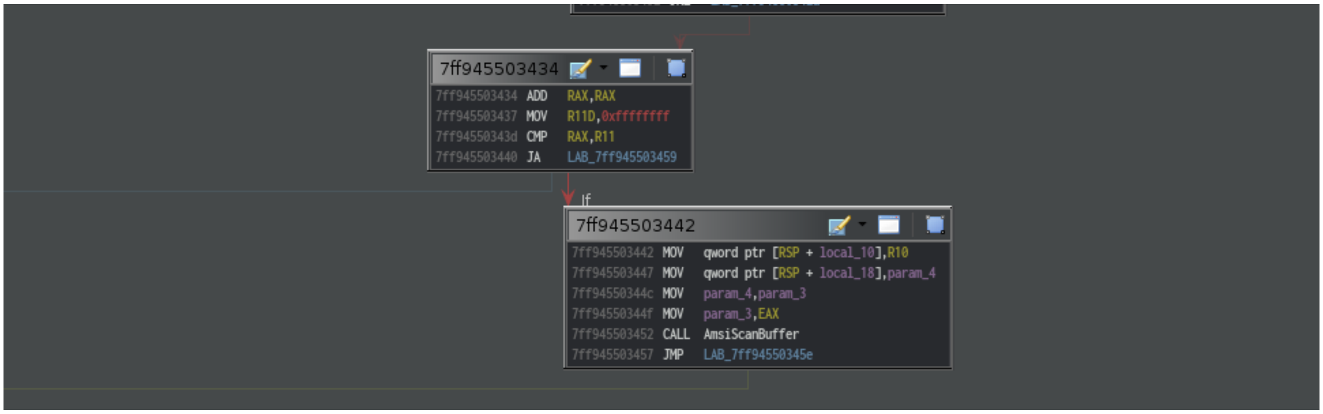


if string to be scanned is not null,



function checks if `result` is null pointer. if so, well the same thing as above, it returns with bad value loaded into `rax` .

else, `result` is valid, it loops through each wide character of the `string` to get the length of it.



After getting the string length, it calls `AmsiScanBuffer` function.

It is clear that this is just a simple wrapper function around `AmsiScanBuffer` .

AmsiScanBuffer

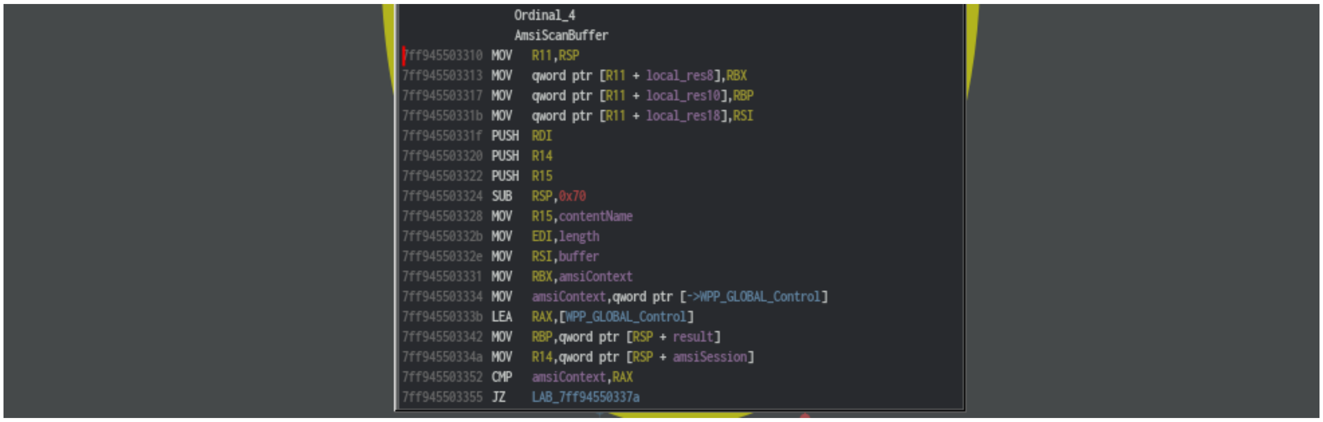
According to the MSDN and as well as the name suggests, the `AmsiScanBuffer` function scans a buffer for malicous content.

here is the function prototype msdn

```
HRESULT AmsiScanBuffer(
  [in]           HAMSICONTEXT amsiContext,
  [in]           PVOID        buffer,
  [in]           ULONG        length,
  [in]           LPCWSTR      contentName,
  [in, optional] HAMSISESSION amsiSession,
  [out]          AMSI_RESULT  *result
);
```

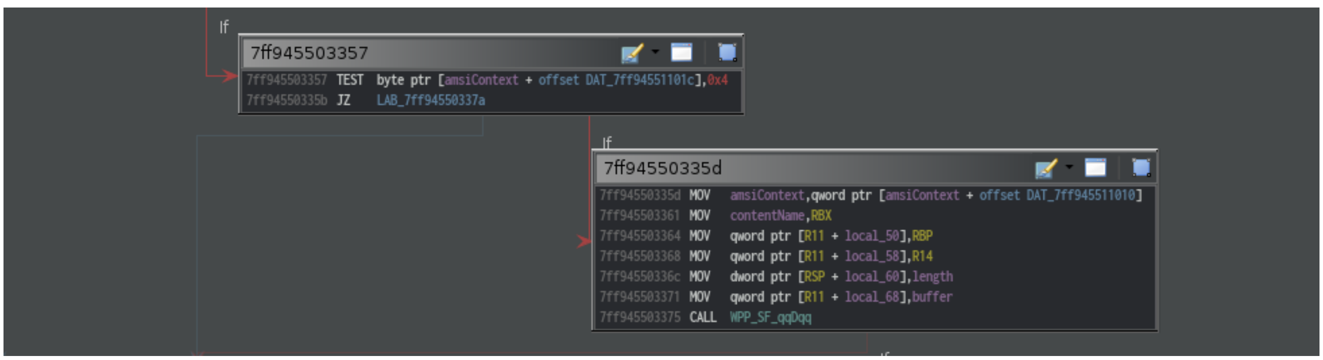
Function takes 6 parameters. One of which is the pointer to the `AMSI_RESULT` enum which i explained above - `*result` . According to MSDN, others include a buffer, which will be scanned by the anti-malware vendor - `buffer` , length of the buffer - `length` , filename, URL, unique script ID - `contentName` and a handler to the session - `HAMSISESSION` structure.

And here’s how this function looks like in disassembly.

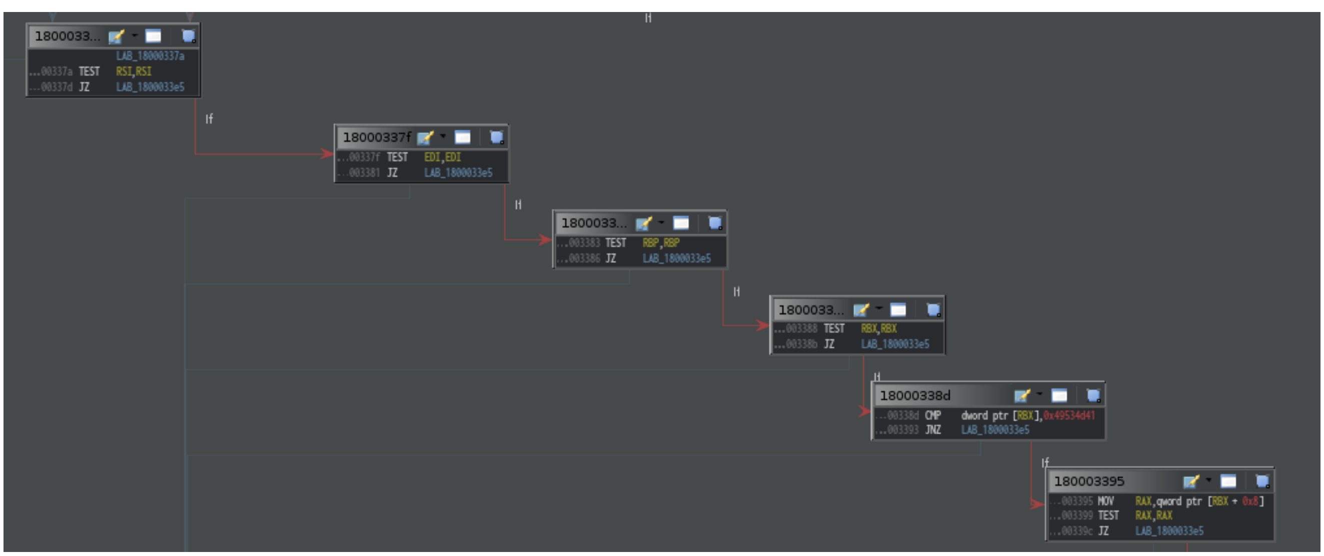


here we can see stack pointer is stored in `r11` register and since this is x64 _stdcall, the first four parameters are stored in rcx, rdx, r8 and r9 registers. Rest are stored in the stack. With that information, we can assume a pointer to the `AMSI_RESULT` enum is stored in the stack.

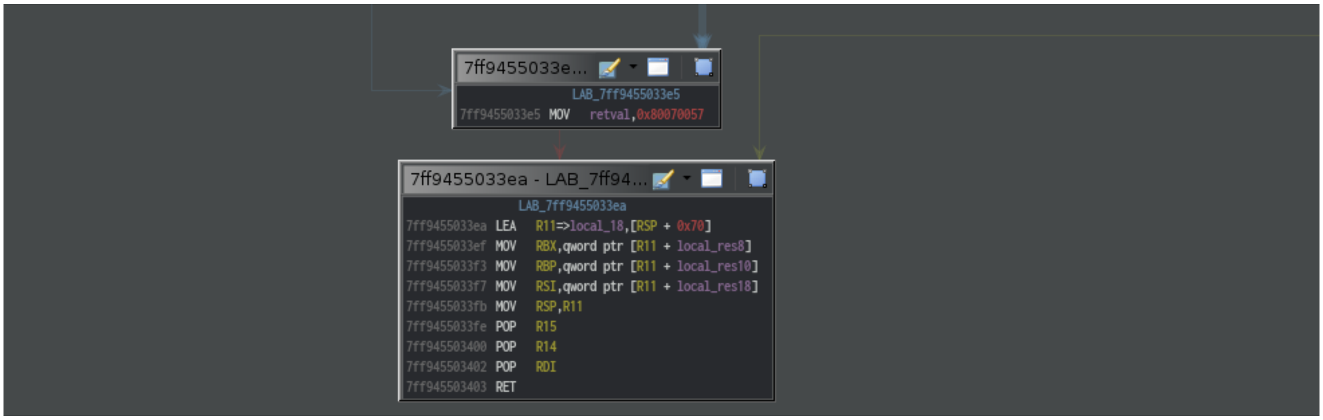
then we can see few comparisons around global data. if the comparisons turns out to be successful, it calls `WPP_SF_qqDqq` function. (windows sofware trace preprocessor).



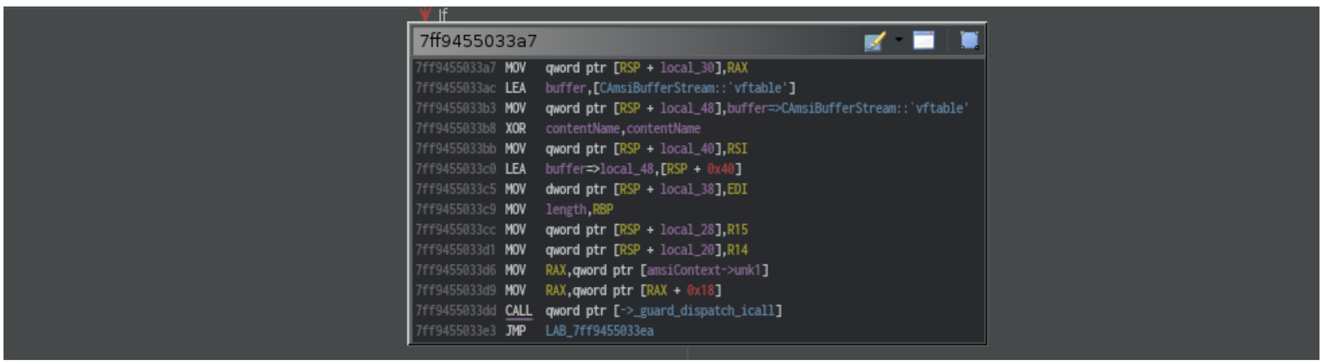
then there is a pretty huge if condition, which is essentially checks if any of the above parameters are invalid



by looking at the comparison, the function won't successfully return if **[rbp]**, which is the first qword of `amsiContext` is not equal to 0x49534d41.

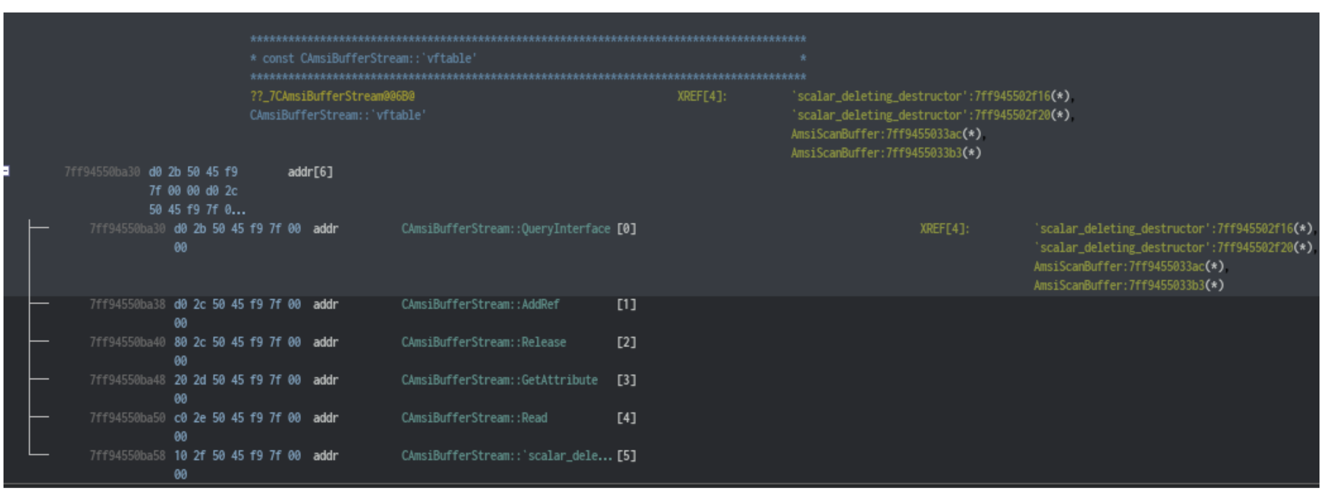


And if parameters invalid, it returns `0x80070057` (which i think is the bad return value)



else, as we can see in the above snippet, `buffer` (rdx register) is now loaded with address of `CAmsiBufferStream::vftable` and stored the value in the stack. This may sound familiar to anyone who has done some C++ reverse engineering since this is a one way to represent constructor calls in assembly (setting vtable to the object's first bytes).

to confirm that we can take a look at `CAmsiBufferStream::vftable`.



as we can see, `CAmsiBufferStream::vftable` is indeed, a virtual function table and what those two instructions doing is creating an object of type `CAmsiBufferStream`. It is also possible to see some member variable intializations too.

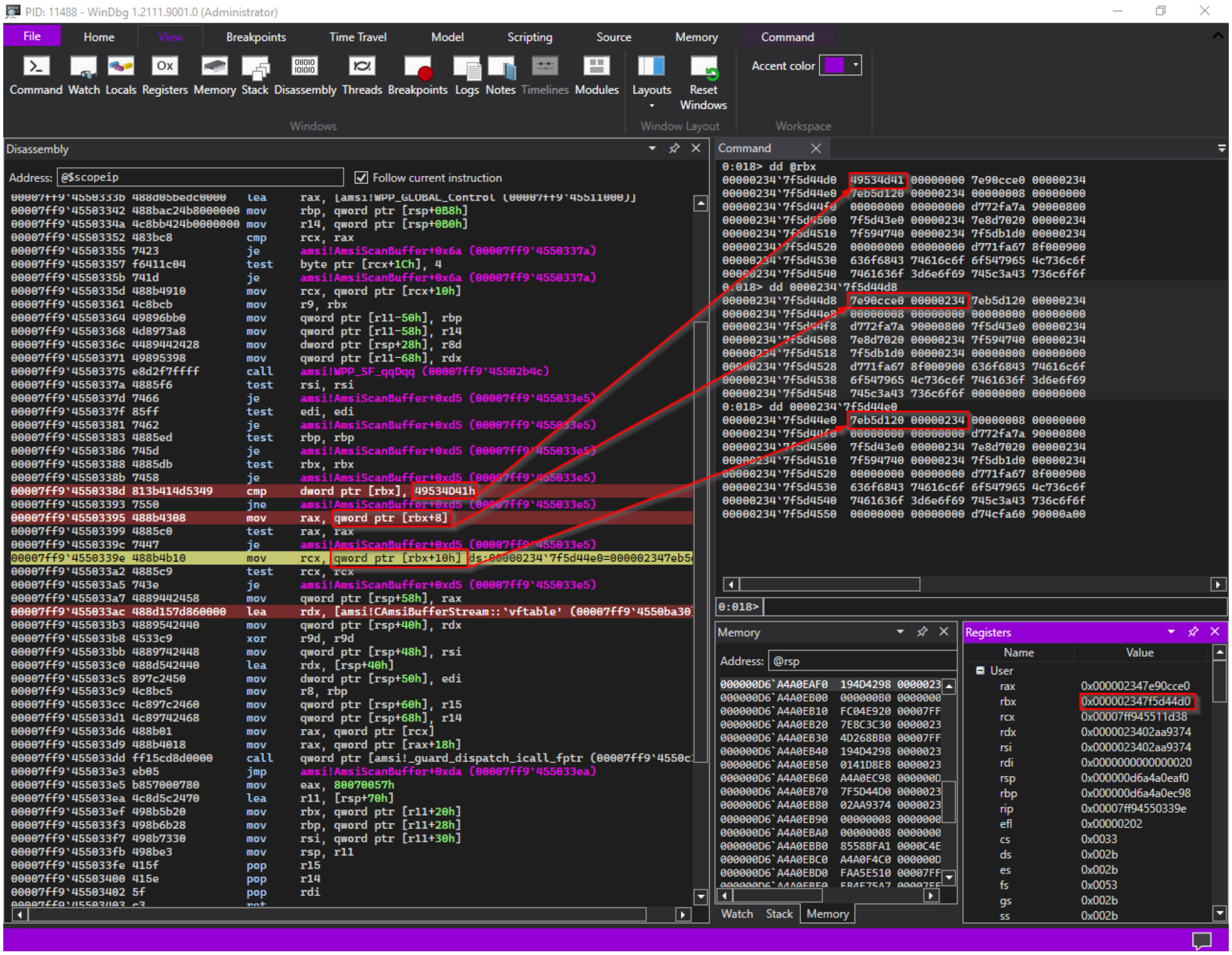
My assumption is that `amsiContext->thirdMember` is somekind of a class that anti-malware vendor has registered to perform scans.

To make sure our assumptions so far are correct, we'll go over this function using windbg.

Since we already know interesting parts of the function, it is easy to place breakpoints.

```
0:018> bl
     0 e Disable Clear  00007ffxxxxx3310     0001 (0001)  0:**** amsi!AmsiScanBuffer
     1 e Disable Clear  00007ffxxxxx338d     0001 (0001)  0: amsi!AmsiScanBuffer+0x7d
     2 e Disable Clear  00007ffx`xxxx3395     0001 (0001)  0:
amsi!AmsiScanBuffer+0x85
     3 e Disable Clear  00007ffxxxx339e     0001 (0001)  0:****
amsi!AmsiScanBuffer+0x8e
     4 e Disable Clear  00007ffxxxxx33ac     0001 (0001)  0:**
amsi!AmsiScanBuffer+0x9c
```

First few breakpoints are placed at locations in assembly where **amsiContext's** member variables are being referenced. Reason being this handle is still unknown to us. Therefore it could be useful to extract every possible information about it. Last breakpoint is placed at the address where **CAmsiBufferStream:vftable** is referenced.



So from the above image, we can assume that the first member of the `amsiContext` is a QWORD but it compares it with a DWORD and second and third members are also QWORDs (8 bytes).

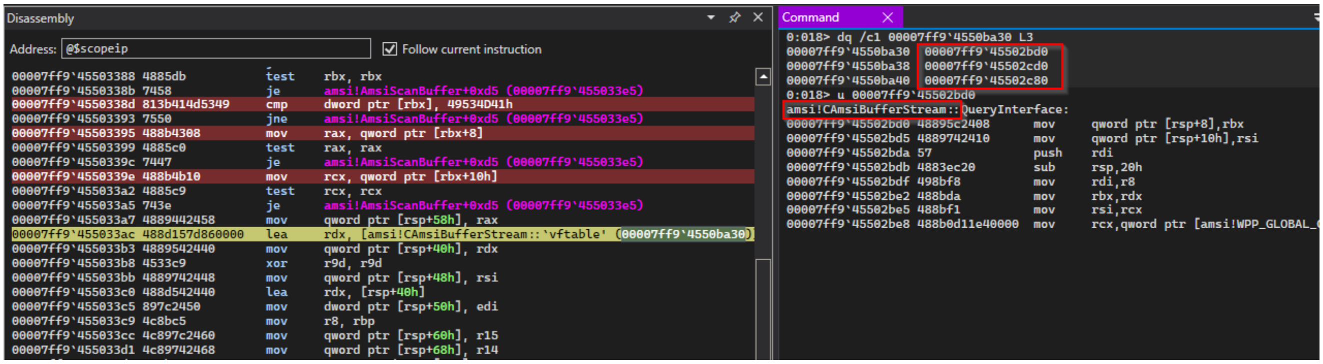
```
0:018> dq /c1 0x000002347f5d44d8 L1
000002347f5d44d8  000002347e90cce0
0:018> dq /c1 0x000002347f5d44e0 L1
000002347f5d44e0  000002347eb5d120
```

We can refer to the memory map to get more information about what those QWORDs are.

∨ 0x2347f550000	Private	1,024 kB	RW	Heap segment (ID 1)	1,016 kB	1,016 kB
0x2347f550000	Private: Commit	1,020 kB	RW	Heap segment (ID 1)	1,016 kB	1,016 kB
0x2347f64f000	Private: Rese…	4 kB		Heap segment (ID 1)		

Now it is clear those two pointers are from heap segment 1. However, we still have no idea about the type of those pointers.

However we already know those are pointers to objects thanks to our previous static analysis.

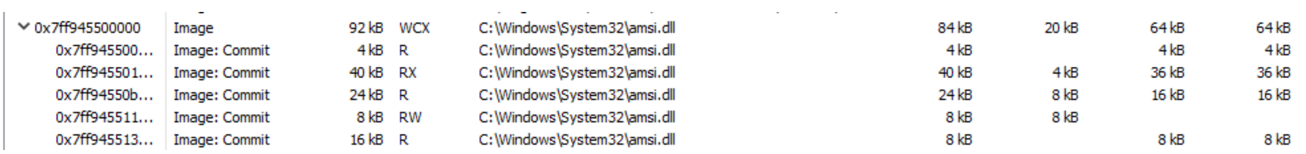


Above screenshot shows the virtual function table of `CAmsiBufferStream`.

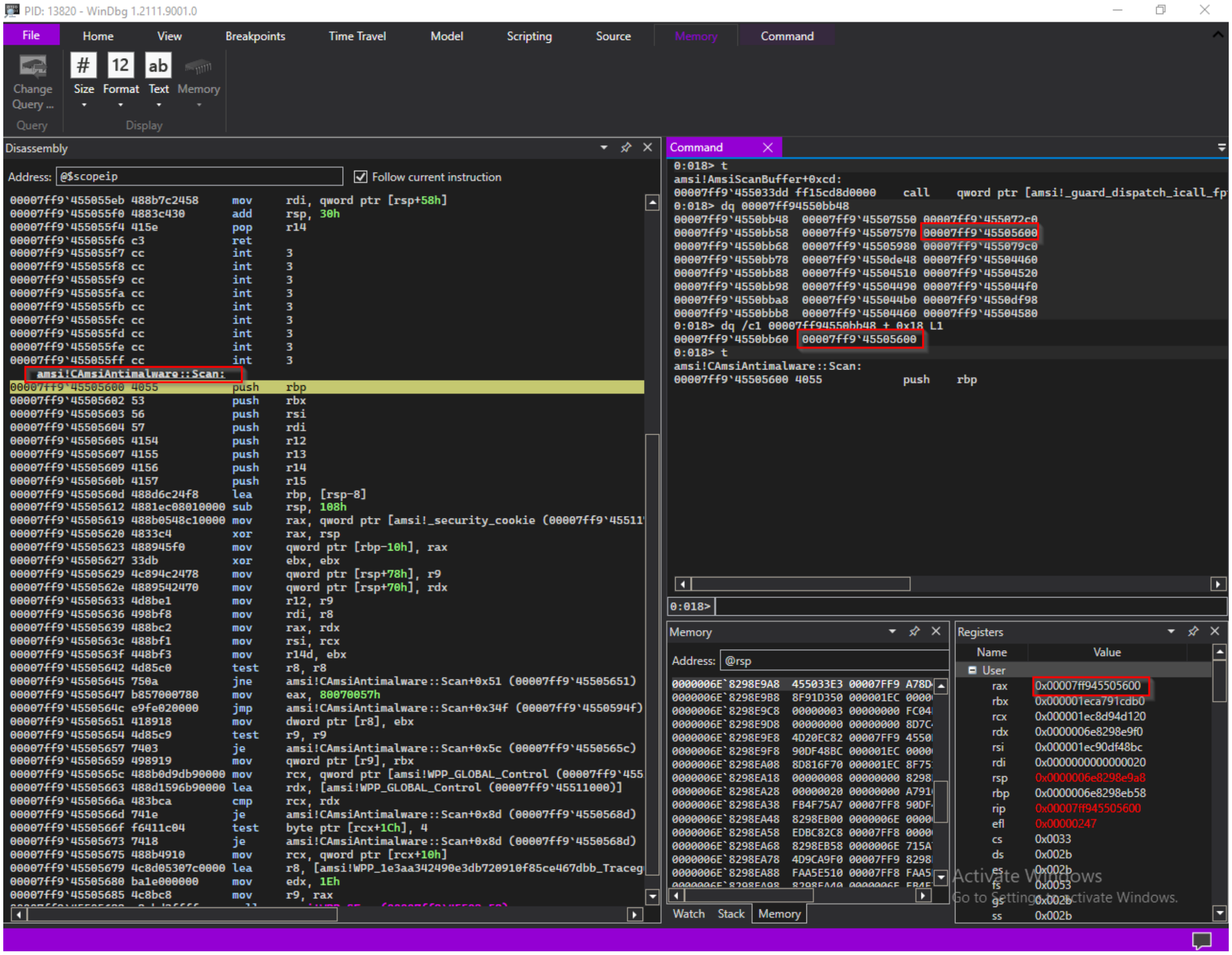
Then the next address where we can find some more information regarding **amsiContext members** is,

```
00007ff9455033d6 488b01          mov     rax, qword ptr [rcx] ds:000002347eb5d120=
{amsi!ATL::CComObject<CAmsiAntimalware>::vftable' (00007ff94550bb48)}
00007ff9455033d9 488b4018        mov     rax, qword ptr [rax+18h]
00007ff9455033dd ff15cd8d0000    call    qword ptr [amsi!_guard_dispatch_icall_fptr
(00007ff9`4550c1b0)]
```

in the above snippet, `rcx` holds one of those pointers we just discussed, `000002347eb5d120` (thirdMember). In the first instruction, 64 bit value at that address is loaded into `rax` register, which, according to the above snippet, is `00007ff94550bb48`. It also specifies that this is a vtable located in .rodata section of the asmi.dll's memory image.



next two instructions retreives address **0x18** offset from the vtable into `rax` register and calls the address stored in `rax`



This proves that our assumption on function pointer extracted from the `HAMSICONTEXT` being a anti-malware vendor's registered function is false and it is a pointer to `amsi!CAmsiAntimalware::Scan` method.

We have uncovered some important details about `HAMSICONETXT` so far. We already know that the first member is a DWORD, and it should be equal to **0x49534d41** in order for scan to be successful. Third member is a pointer to an object of class `CAmsiAntimalware`, which has a virtual function called `amsi!CAmsiAntimalware::Scan`.

And by moving its 0x0 offset `rax` register, we can access it's virtual function table where we can find `Scan` at the 0x18.

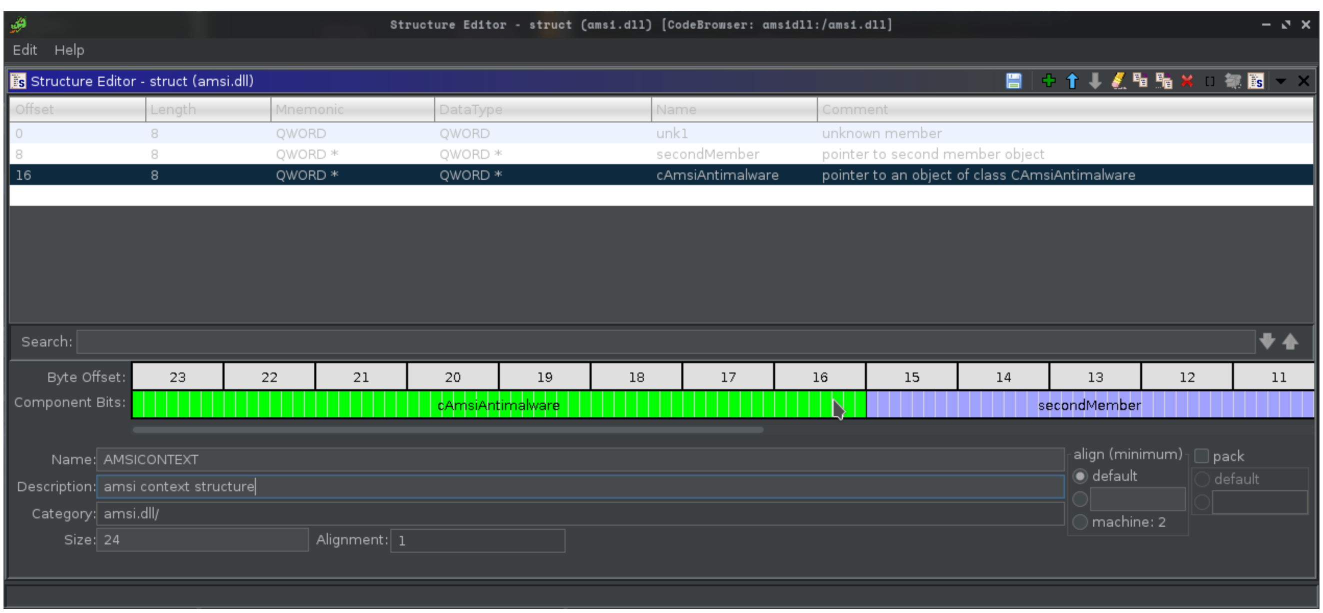
The whole thing can be roughly decompiled down into below C code.

```
class CAmsiAntimalware {
    private:
        [...]

    public:
        virtual Scan(CAmsiBufferStream *, AMSI_RESULT, DWORD);

        [...]
}

typedef HAMSICONTEXT {
    QWORD               unk1;
    QWORD               *secondMember;
    CAmsiAntimalware    *antimalware;

    [...]
};

HRESULT __stdcall AmsiScanBuffer
(
        HAMSICONTEXT amsiContext,
        PVOID buffer,
        ULONG length,
        LPCWSTR contentName,
        HAMSISESSION amsiSession,
        AMSI_RESULT *result
)
{
    auto var;
    if ((WPP_GLOBAL_Control != &WPP_GLOBAL_Control) && (*(WPP_GLOBAL_Control +
0x1c)) != 4))
    {
        WPP_SF_qqDqq(
            *((BYTE*)WPP_GLOBAL_Control + 0x10),
            buffer,
            length,
            amsiContext,
            buffer,
            amsiSession,
            result
        );
    }

    if (
            buffer == NULL ||
            result == NULL ||
            amsiContext == NULL ||
            *((DWORD *)amsiContext) != 0x49534D41 ||
            *((QWORD *)amsiContext + 1) == 0x0 ||
            *((QWORD *)amsiContext+2) == 0x0
        )
```

```
        {
            return 0x80070057;
        }
        else
        {
            CAmsiBufferStream bufferStream = CAmsiBufferStream(
                buffer,
                length,
                amsiContext->secondMember,
                contentName,
                session
            );

            return amsiContext->antimalware->Scan(
                amsiContext->antimalware, // this
                &bufferStream, // CAmsiBufferStream *
                result,
                0
            );
        }
    }
```

We are not done yet. Goal here is to understand how AMSI works. Therefore, our next target is amsi!CAmsiAntimalware::Scan.

But before drill down into it, we need to construct the `HAMSICONTEXT` structure out of the knowlegde we have.

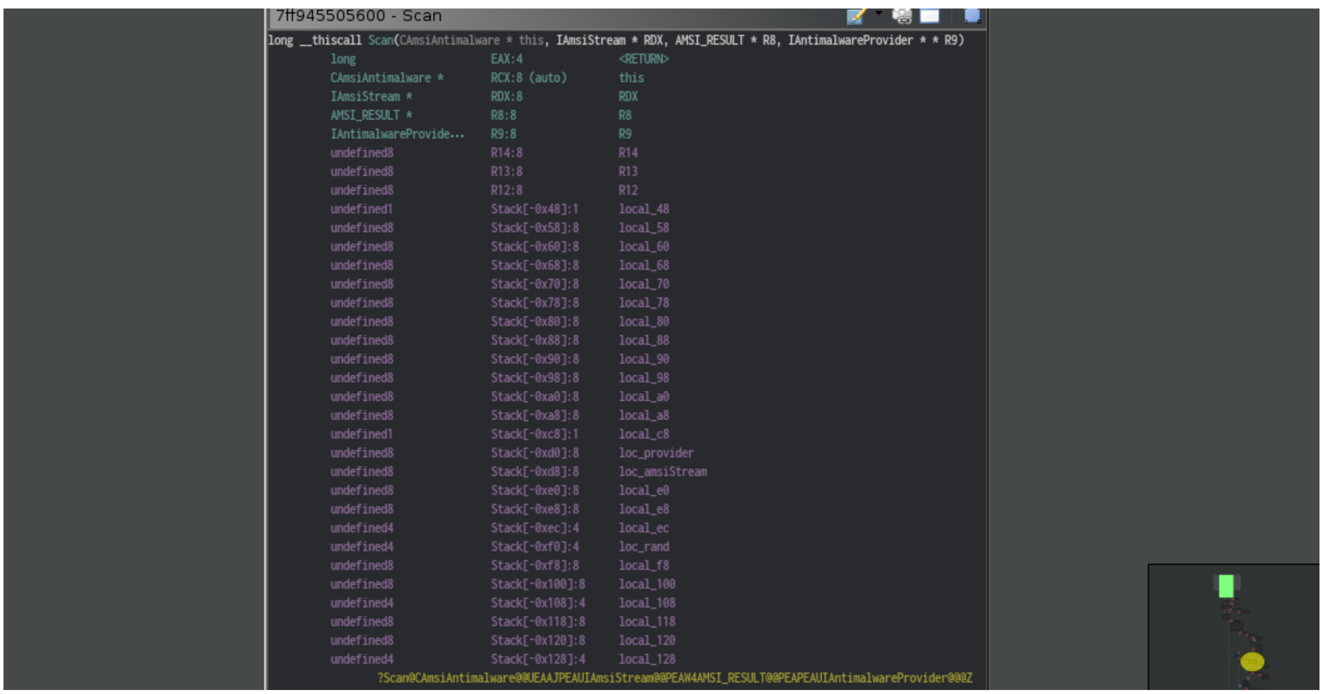


now we can see decompiler output is much more accurate and readable.

```
/* WARNING: Function: _guard_dispatch_icall replaced with injection: guard_dispatch_icall */

HRESULT AmsiScanBuffer(AMSICONTEXT *amsiContext,PVOID buffer,ULONG length,LPCWSTR contentName,
                      undefined8 amsiSession,AMSI_RESULT *result)

{
  HRESULT retval;
  undefined4 in_register_00000084;

                    /* 0x3310  4  AmsiScanBuffer */
  if (((undefined **)WPP_GLOBAL_Control != &WPP_GLOBAL_Control) &&
     ((WPP_GLOBAL_Control[0x1c] & 4) != 0)) {
    WPP_SF_qqDqq(*(undefined8 *)(WPP_GLOBAL_Control + 0x10),buffer,
                 CONCAT44(in_register_00000084,length),amsiContext,(char)buffer,(char)length,
                 (undefined1)amsiSession,(char)result);
  }
  if ((((buffer == (PVOID)0x0) || (length == 0)) || (result == (AMSI_RESULT *)0x0)) ||
     (((amsiContext == (AMSICONTEXT *)0x0 || (*(int *)&amsiContext->unk1 != 0x49534d41)) ||
      ((amsiContext->secondMember == (QWORD *)0x0 || (amsiContext->cAmsiAntimalware == (QWORD *)0x0)
       ))))) {
    retval = -0x7ff8ffa9;
  }
  else {
    retval = (**(code **)((longlong)*amsiContext->cAmsiAntimalware + 0x18))();
  }
  return retval;
}
```

We can also try constructing a `CAmsiAntimalware` class but we dont have enough information to populate member variables.

CAmsiAntimalware::Scan

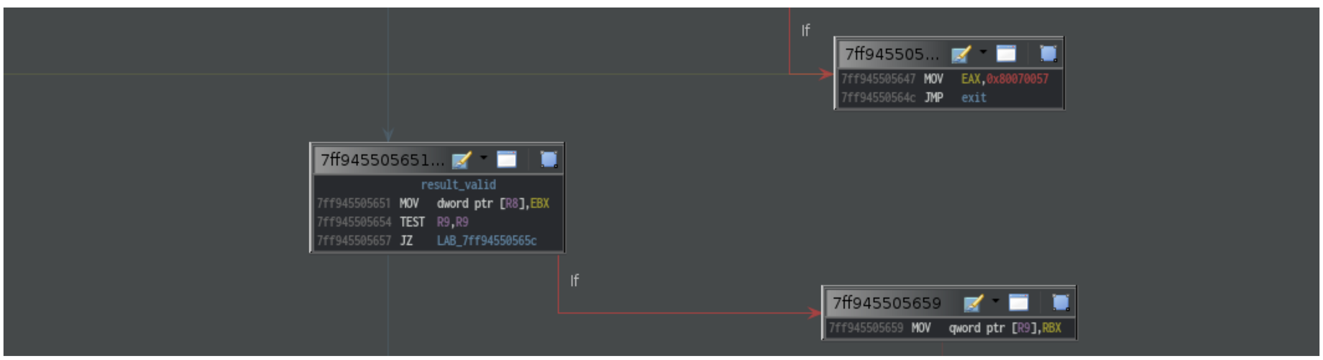


```
7ff945505600 - Scan
long __thiscall Scan(CAmsiAntimalware * this, IAmsiStream * RDX, AMSI_RESULT * R8, IAntimalwareProvider * * R9)
        long                   EAX:4             <RETURN>
        CAmsiAntimalware *     RCX:8 (auto)      this
        IAmsiStream *          RDX:8             RDX
        AMSI_RESULT *          R8:8              R8
        IAntimalwareProvide... R9:8              R9
        undefined8             R14:8             R14
        undefined8             R13:8             R13
        undefined8             R12:8             R12
        undefined1             Stack[-0x48]:1    local_48
        undefined8             Stack[-0x58]:8    local_58
        undefined8             Stack[-0x60]:8    local_60
        undefined8             Stack[-0x68]:8    local_68
        undefined8             Stack[-0x70]:8    local_70
        undefined8             Stack[-0x78]:8    local_78
        undefined8             Stack[-0x80]:8    local_80
        undefined8             Stack[-0x88]:8    local_88
        undefined8             Stack[-0x90]:8    local_90
        undefined8             Stack[-0x98]:8    local_98
        undefined8             Stack[-0xa0]:8    local_a0
        undefined8             Stack[-0xa8]:8    local_a8
        undefined1             Stack[-0xc8]:1    local_c8
        undefined8             Stack[-0xd0]:8    loc_provider
        undefined8             Stack[-0xd8]:8    loc_amsiStream
        undefined8             Stack[-0xe0]:8    local_e0
        undefined8             Stack[-0xe8]:8    local_e8
        undefined4             Stack[-0xec]:4    local_ec
        undefined4             Stack[-0xf0]:4    loc_rand
        undefined8             Stack[-0xf8]:8    local_f8
        undefined8             Stack[-0x100]:8   local_100
        undefined4             Stack[-0x108]:4   local_108
        undefined8             Stack[-0x118]:8   local_118
        undefined8             Stack[-0x120]:8   local_120
        undefined4             Stack[-0x128]:4   local_128
               ?Scan@CAmsiAntimalware@@UEAAJPEAUIAmsiStream@@PEAW4AMSI_RESULT@@PEAPEAUIAntimalwareProvider@@@Z
```

So ghidra has created a nice view of the stack frame for us. And by looking at the parameters, we see the function expects a pointer to an `IAmsiBuffer` object and a pointer to a pointer of `IAntimalwareProvider` object.

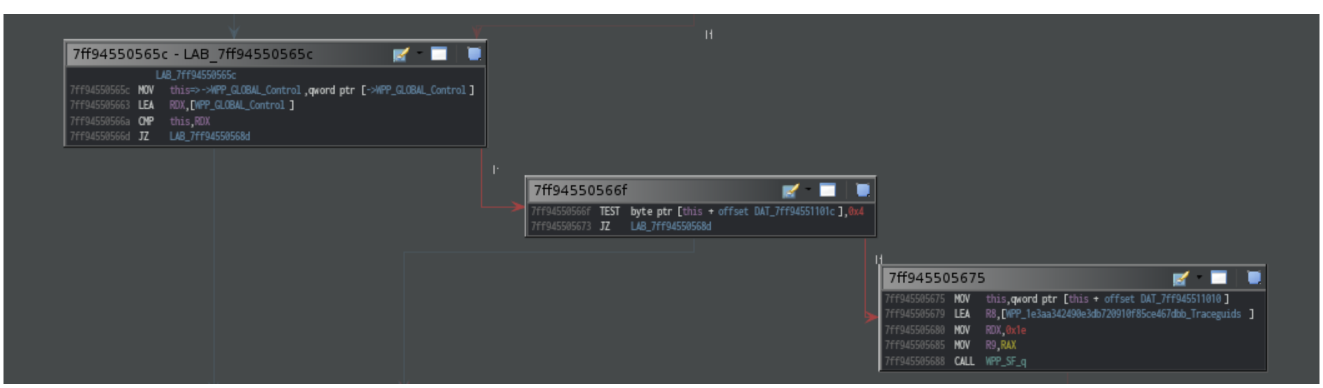
We saw that in the `AmsiScanBuffer` that this value is set to zero.

Then continues to setup all those memory curruption protection machanisms and to check the validity of the input parameters. First it checks if third parameter, `result` is null (remember, result is a pointer to AMSI_RESULT enum).



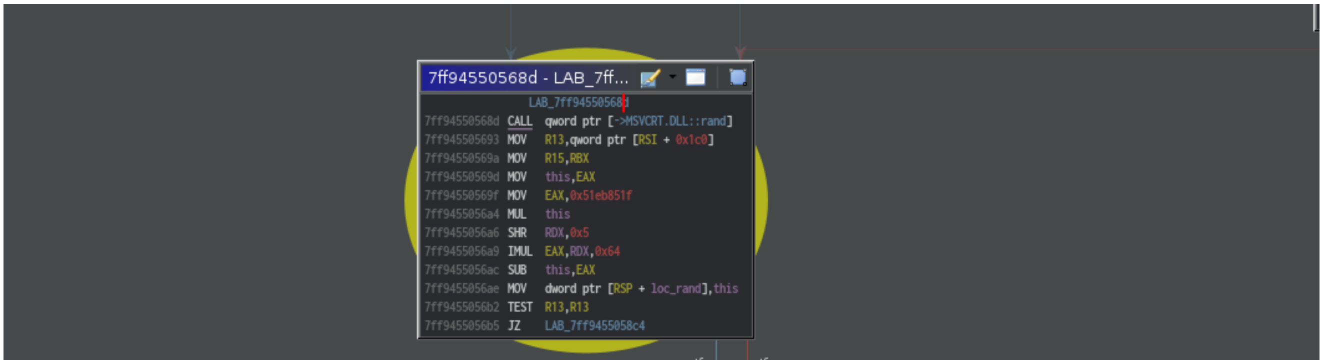
if it is not, it jumps to label `result_valid`. else, it sets `eax` to `0x80070057` and return. In the `result_valid` label, it sets `*result` to `AMSI_RESULT_CLEAN` (0x0). So it looks like the function is clearing the `*result` to not detected state. Which means we can expect value of `result` to change.

It also checks if `provider` is null. If not, it sets value of it to null and continue execution from `LAB_7ff94550565c`. else, it continues the execution from the same location but without setting `*provider` to null.



`LAB_7ff94550565c` does the same thing as `AmsiScanBuffer` did at the block `0x7ffxxxxx335d`. However instead of calling `WPP_SF_qqDqq` it calls `WPP_SF_q`. Also note that above snippet sets `rdx` to either address of `[WPP_GLOBAL_CONTROL]` or `0x1e`.

`LAB_7ff94550568d` looks interesting.



First it calls `rand()` function. In case you dont know, it's pretty common C library function and it generates a psuedo random number and return it. In the next line, it stores a member of `CAmsiAntimalware` class at offset `0x1c0` in `r13` register. Then there are some multipications around the generated value value.

ghidra being ghidra, has renamed registers with the variable names (this is good if we are doing x86 reversing becuase most of calling conventions pass parameters through stack, However, in our case, since parameters are passed through registers, renaming those can cause confusion), So to make it clear, we'll use listing view.

It assigns the return value from `rand()` to `ecx` register and loads `eax` with **0x51eb851f**. then it multiplies random value stored in `ecx` with the value loaded in `eax`. Note that this instruction is capable of changing the value at `edx` register.

Then there's a shift right instruction, which shifts 5 bits from `edx` register. then it multiplies shifted `edx` with 0x64 and stores the value in `eax`.

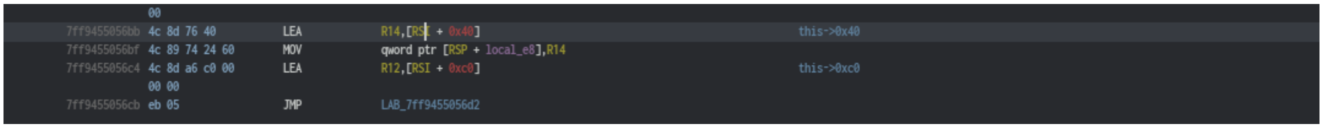
sub instruction substracts `eax`, by `ecx`. what this whole thing does is similar to below expression

```
rand() % 0x64;
```

value of `ecx` is then stored in a local variable `loc_rand` and function checks if `r13`, which holds the value of `this->0x1c0` is 0/null. If yes, it jumps to `LAB_7ff9455058c4`. else, it continues exection from next address.

Now we got two control paths to follow. but first, I'm not gonna take the jump.

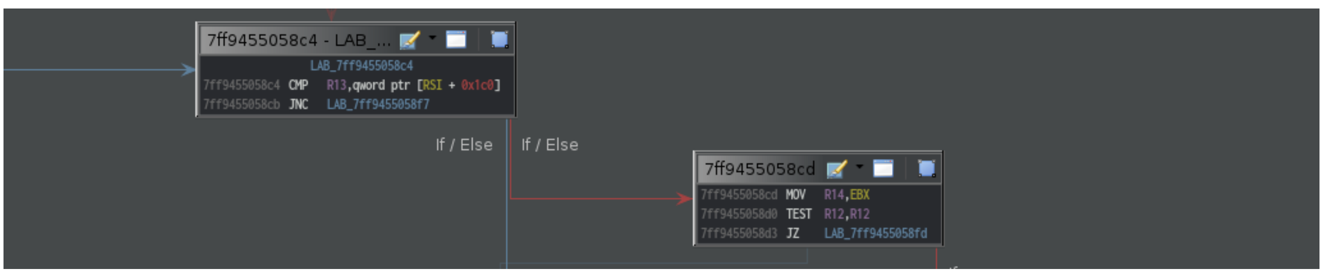
Control flow path 1



`0x7ffxxxxx56bb`, address of `this->0x40` gets loaded into `r14`, which then gets stored in a local variable. Next instruction loads `this->0xc0` into `r12` register.

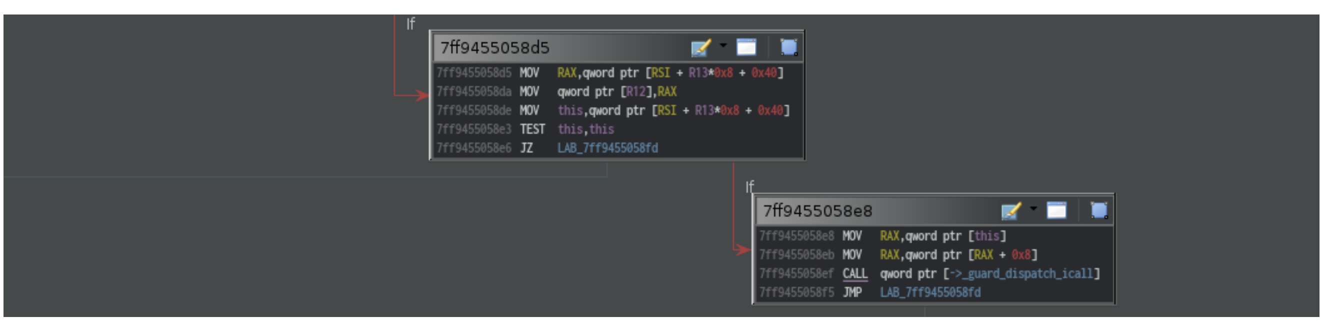
Then there's an unconditional jump and this one jumps directly into a loop. so Im gonna save that part for a debugging session and continue with the other control flow path.

Control flow path 2



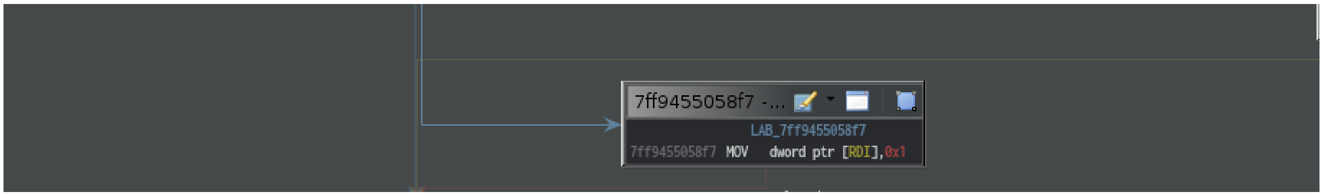
`LAB_7ff9455058c4` starts with a comparison of `r13` (this->0x1c0 but as a local variable) with `this->0x1c0`. The comparison checks if `r13` is less than `this->0x1c0`. if it is, control flow is directed to address `0x7ffxxxxx58cd`. else, control flow is directed to label `LAB_7ff9455058f7`.

First instruction at `0x7ffxxxxx58cd` sets `r14` to zero (rbx is xored by itself at the begining of the function). Next two instructions checks if `r12` is null.

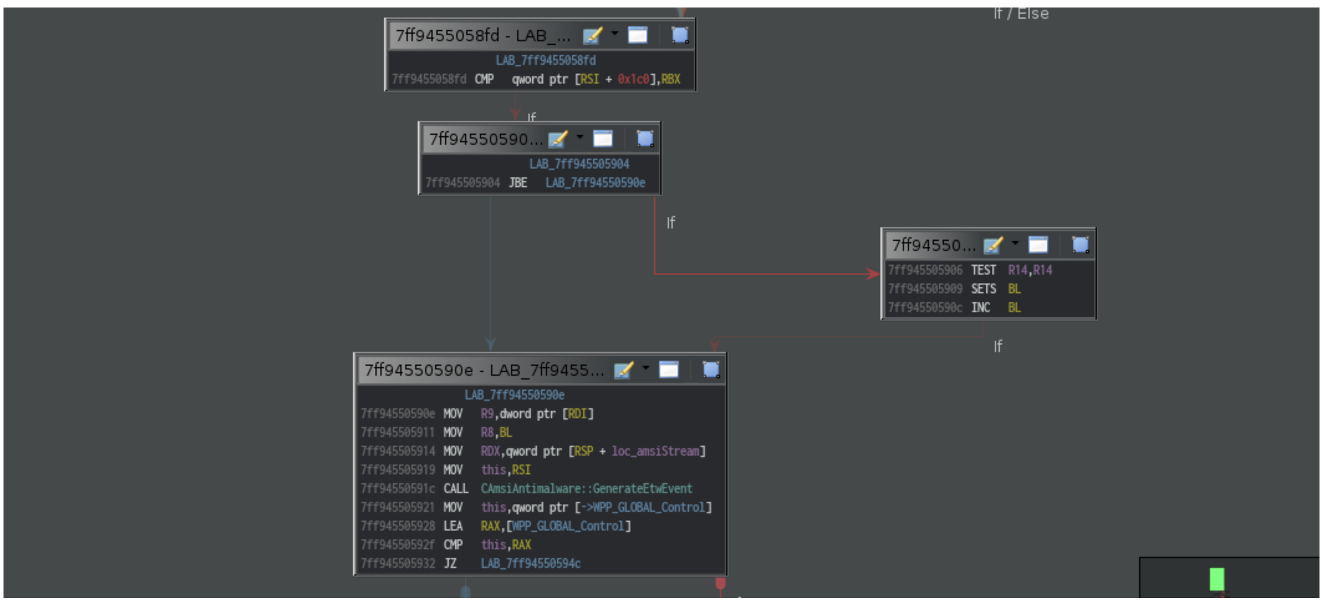


if not, value at address `r12` is set to `[RSI + r13*0x8 + 0x40]`. Then it checks if `rcx` is null. If we assume the jump to `LAB_7ff9455058c4` taken from `0x7ffxxxxx56b5`, then `rcx` would be the remainder of `rand() % 0x64` thing. if `rcx` is null, jump is taken to label `LAB_7ff9455058fd`. else, it loads value at `(*(rcx) + 0x8)` to `rax` and calls it through `_guard_dispatch_icall`.

if `r12` is null, jump is also taken to label `LAB_7ff9455058fd`.



on the other hand, `LAB_7ff9455058f7` also jumps to `LAB_7ff9455058fd` after moving 0x1 into `[rdi]`. We already know that `rdi` is pointing to `AMSI_RESULT` enum. Constant 1 means `AMSI_RESULT_NOT_DETECTED`.



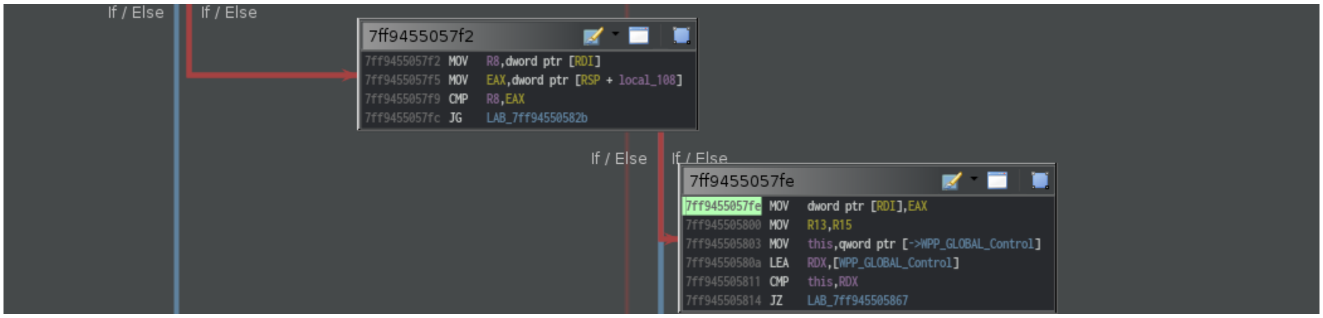
this simply checks if `this->0x1c0` is null, if it is, it jumps to label `LAB_7ff94550590e` else, it continues exection from address `0x7ffxxxxx5906`.

block starting at `0x7ffxxxxx5906` basically checks if `R14` is null. it sets `bl` if previous comparison has caused sign flag to be 1. The operation may look like this in pseudocode.

```
bl = (r14 < 0) + 1;
```

as you can see in the above control flow graph, code is finally directed towards `LAB_7ff94550590e`. What this snippet does is, call `CAmsiAntimalware::GenerateEtwEvent` method. it passes `this` and `amsiStream` and `bl` through `rcx`, `rdx` and `r9` registers as first three arguments. fourth and the last one is passed through `r9` and this is basically the `AMSI_RESULT`.

Now Im going to find where `AMSI_RESULT` is being modified. We already know `rdi` is a pointer to the enum.



In the above snippet, `rdi` (result) is assigned to value of `eax` . if we go up in the control flow, we can see `eax` is assigned with `local_108` .

Now we know some interesting places to place breakpoints and analyze, it is time to get into a windbg session.

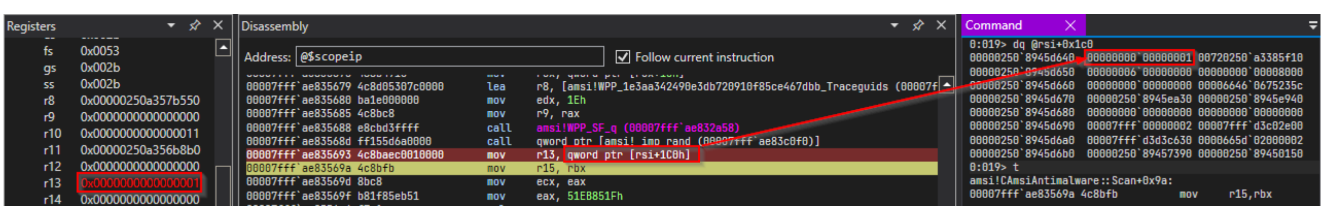
First, Im gonna place a break point at address at place where `provider` is checked.

```
0:018> bp 0x7ffxxxxx5654
0.018> g

[...]

0.018> r r9
r9=0000000000000000
```

As it is clear from the above snippet, `r9` register which holds a pointer to a pointer of `IAntimalwareProvider` class is set to zero. We saw this earlier in `AmsiScanBuffer` function.

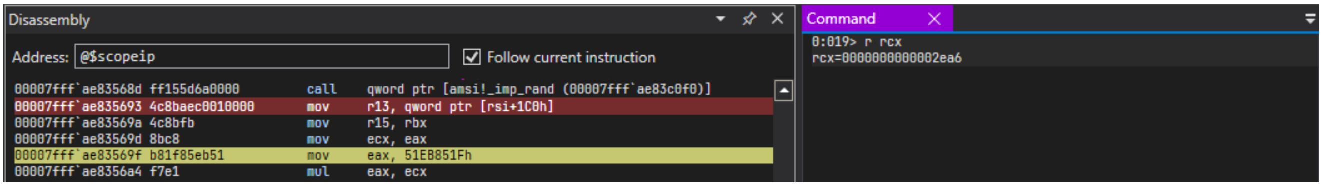
Even if some value is passed down through this register, `CAmsiAntimalware::Scan` will set it to zero.

the next important piece for us is where `this` is being accessed.



above diagram shows exection has been stopped just after the instruction where function accessess `this->0x1c0` .

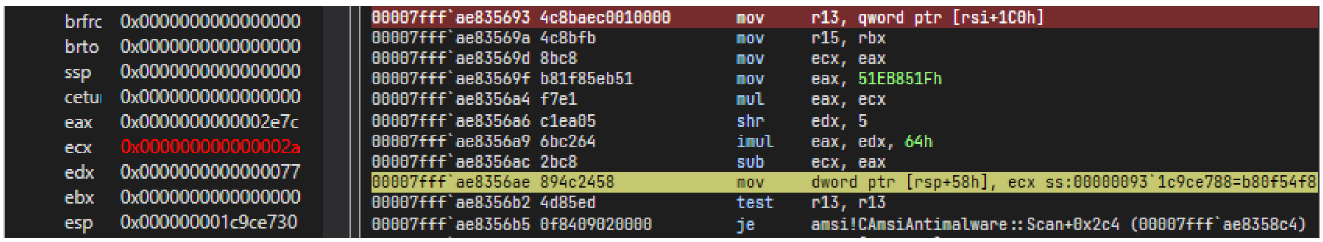
And the value at that address is set to 0x1. This gives us a hint that this member might be numerical value rather than a pointer.



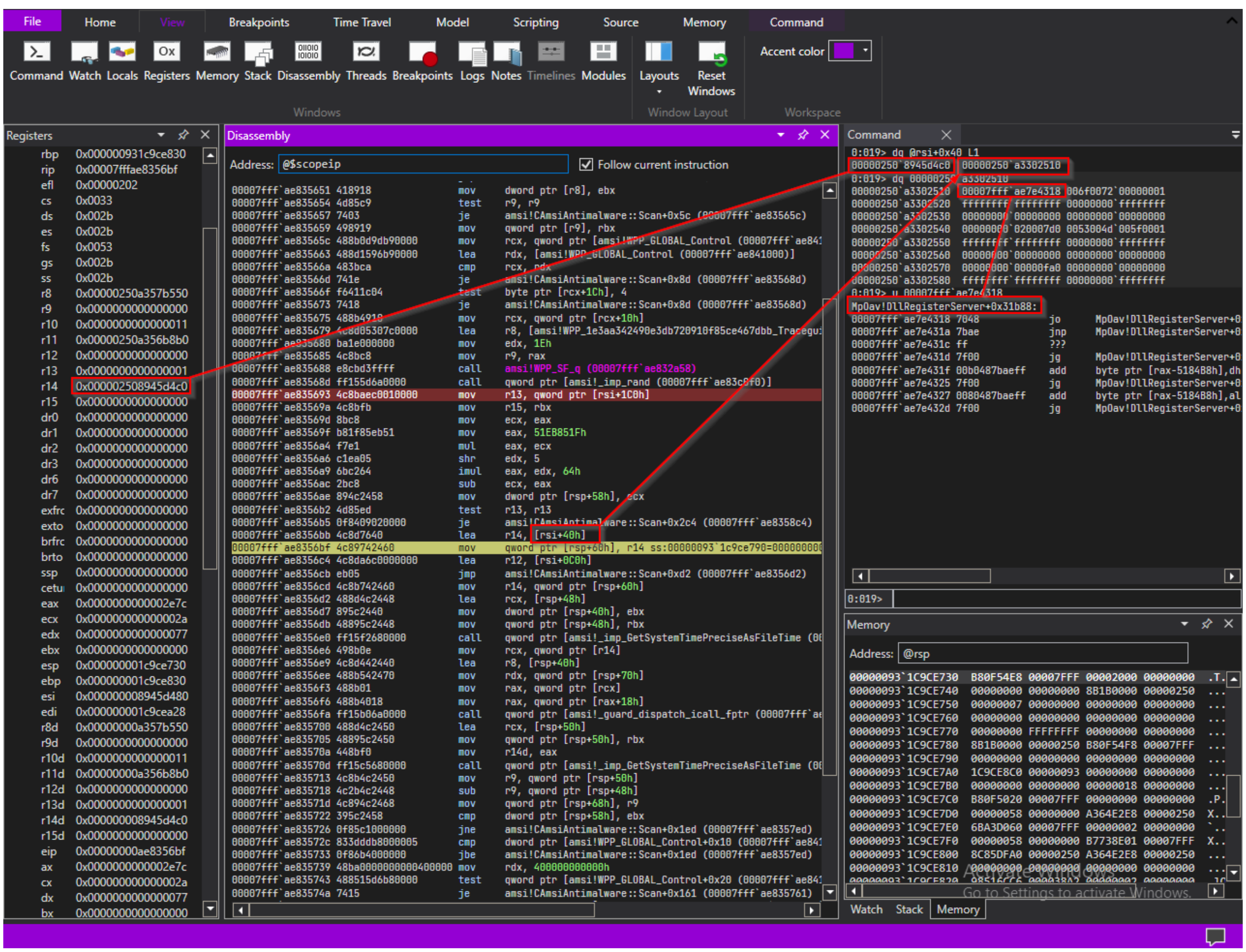
A little below that, we can the random number generated by `rand()` being stored in `ecx` register and that value is `0x2ea6`.

Since we already know what this snippet does, we can perform the calculation by ourself.

```
>>> hex(0x2ea6 % 0x64)
'0x2a'
```



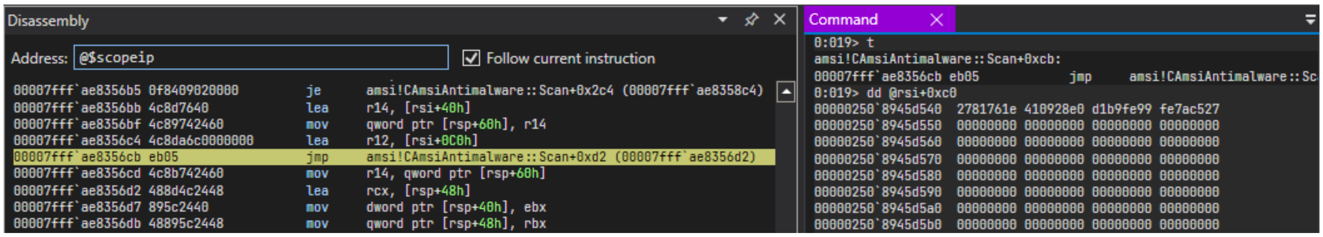
Above diagram conludes that.



Above diagram shows where the function retreives address of `this->0x40` into `r14` register.

When `this->0x40` is printed, it also looks like an address that pointed at heap.

Value at `*this->0x40` looks like a function pointer and when disasseble that address, windbg prints disassembly of `MpOav!DllRegisterServer` (another dll ? we'll see)but disassembly starts from the middle of the function. This might not be a function pointer after all.



here is another place where a member of `CAmsiAntimalware` class has been referenced. this time as we've discussed when doing static analysis, stores address `this->0xc0`.

It doesnt provide us with imformation about type of data even if we take a look at the data at that address,

Control flow path 1 continued

Now we are at the instruction in disassembly where that loop begins.

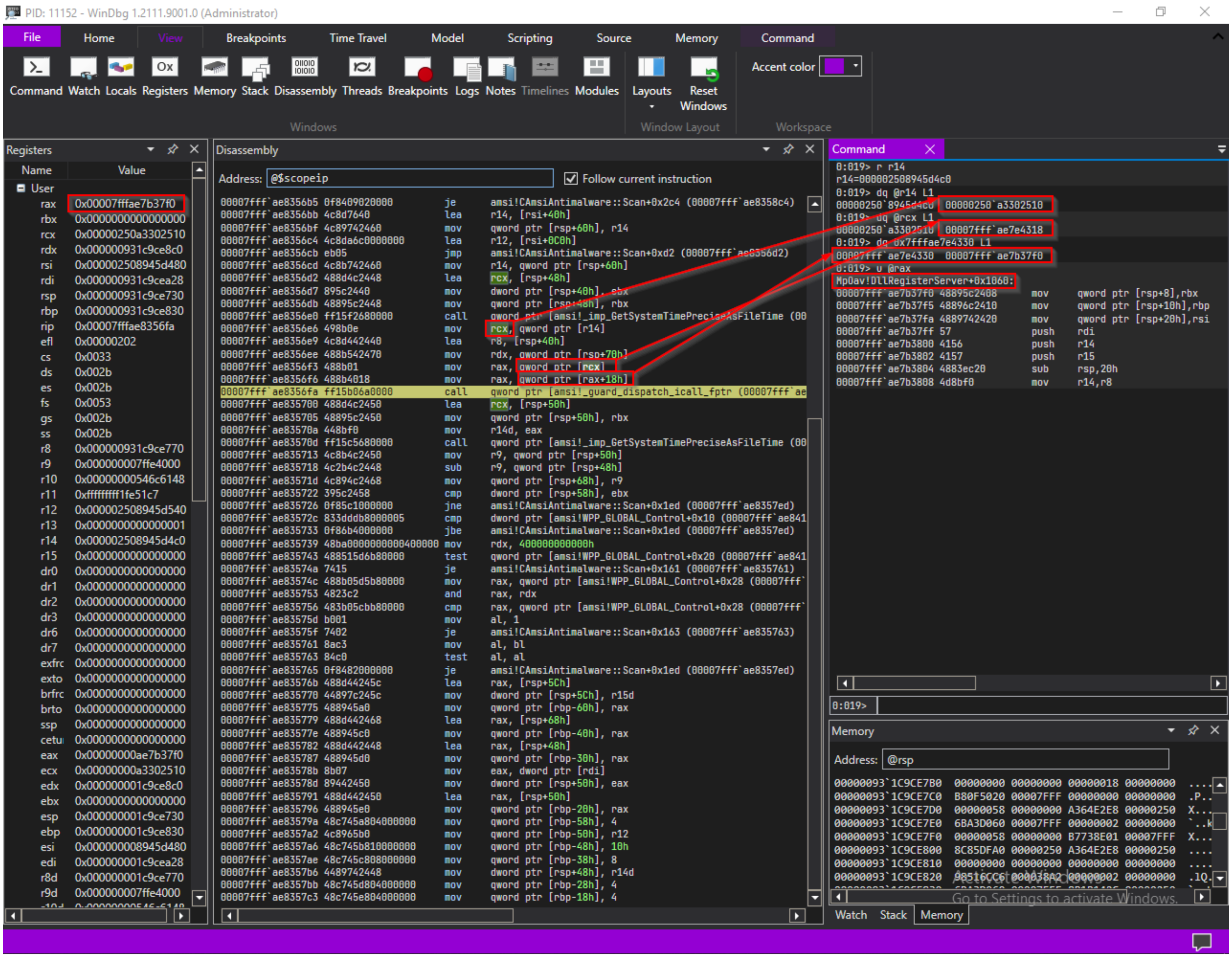
```
00007fffae8356d2 488d4c2448           lea     rcx, [rsp+48h]
00007fffae8356d7 895c2440             mov     dword ptr [rsp+40h], ebx
00007fffae8356db 48895c2448           mov     qword ptr [rsp+48h], rbx
00007fffae8356e0 ff15f2680000         call    qword ptr
[amsi!_imp_GetSystemTimePreciseAsFileTime (00007fff`ae83bfd8)]
```

We see that in the above image, first instruction loads address of `rsp+0x48` into `rcx` register and calls `GetSystemTimePreciseAsFileTime`, which is used to retrieve the current system date and time with the highest possible level of precision in UTC format.

before the call instruction it also initialize `rsp+0x40` and `rsp+0x48` with 0x0.

Then value at address `r14` gets stored in `rcx` register. if you remember, `r14` register stores `&this->0x40` so `rcx` would be value of `this->0x40`.

Then can see some manipulations around that value.



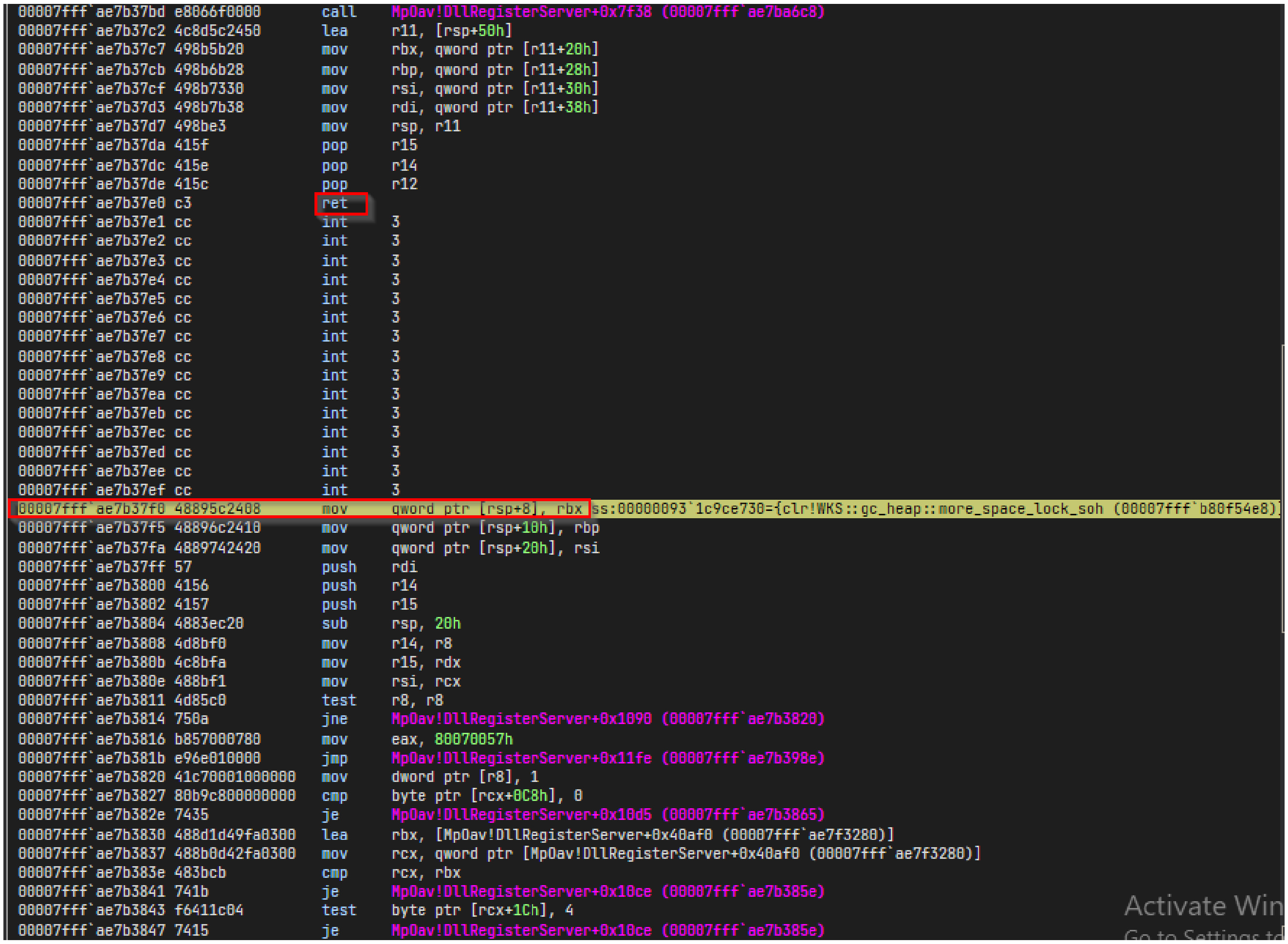
`mov rax, qword ptr [rcx]` stores value at `*this->0x40` in `rax` register. Next instruction takes 0x18 th offset of it and stores it back in `rax` register. Then that address is called using a `gaurd_dispatch_icall_fptr`.

With that information it is clear that `this->0x40` is a pointer to an object of an unknown class. `rcx` now points to that object and `rax` holds one of function pointers in the object's vftable. Well my guess is that this is the windows defender's AMSI COM interface.

The first argument passed to the function is `this->0x40`. Second, third and fourth are passed through `rdx` and `r8` registers. we can see that in the disassembly `rdx` being set to `rsp+0x70` (amsiBuffer) and `r8` being initialized to the address of `rsp +0x40` (who's value is 0).

Weird thing is, the function is jumping into the middle of a function.

Let's try following it.

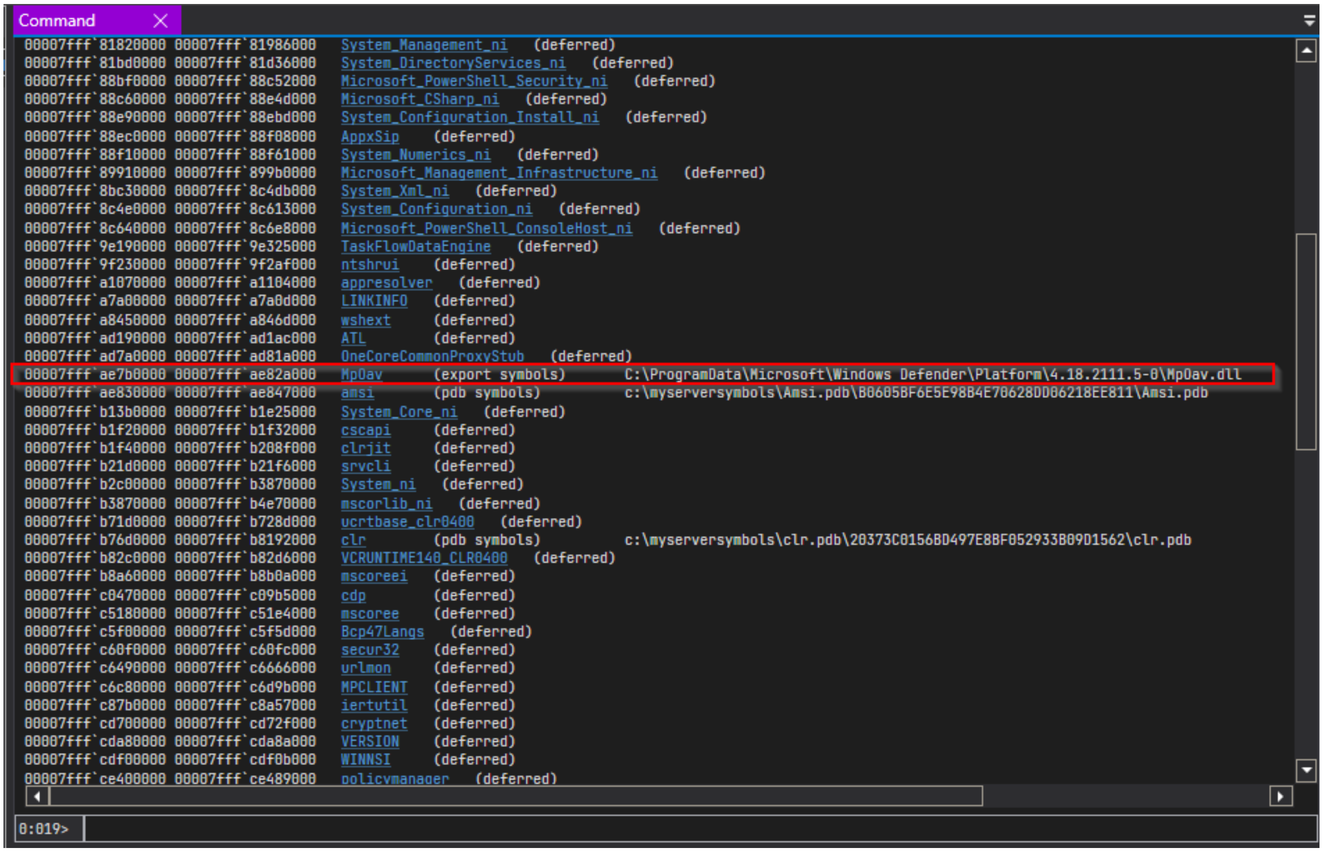


```
00007fff`ae7b37bd e8066f0000        call    MpOav!DllRegisterServer+0x7f38 (00007fff`ae7ba6c8)
00007fff`ae7b37c2 4c8d5c2450        lea     r11, [rsp+50h]
00007fff`ae7b37c7 498b5b20          mov     rbx, qword ptr [r11+20h]
00007fff`ae7b37cb 498b6b28          mov     rbp, qword ptr [r11+28h]
00007fff`ae7b37cf 498b7330          mov     rsi, qword ptr [r11+30h]
00007fff`ae7b37d3 498b7b38          mov     rdi, qword ptr [r11+38h]
00007fff`ae7b37d7 498be3            mov     rsp, r11
00007fff`ae7b37da 415f              pop     r15
00007fff`ae7b37dc 415e              pop     r14
00007fff`ae7b37de 415c              pop     r12
00007fff`ae7b37e0 c3                ret
00007fff`ae7b37e1 cc                int     3
00007fff`ae7b37e2 cc                int     3
00007fff`ae7b37e3 cc                int     3
00007fff`ae7b37e4 cc                int     3
00007fff`ae7b37e5 cc                int     3
00007fff`ae7b37e6 cc                int     3
00007fff`ae7b37e7 cc                int     3
00007fff`ae7b37e8 cc                int     3
00007fff`ae7b37e9 cc                int     3
00007fff`ae7b37ea cc                int     3
00007fff`ae7b37eb cc                int     3
00007fff`ae7b37ec cc                int     3
00007fff`ae7b37ed cc                int     3
00007fff`ae7b37ee cc                int     3
00007fff`ae7b37ef cc                int     3
00007fff`ae7b37f0 48895c2408        mov     qword ptr [rsp+8], rbx ss:00000093`1c9ce730={clr!WKS::gc_heap::more_space_lock_soh (00007fff`b80f54e8)}
00007fff`ae7b37f5 48896c2410        mov     qword ptr [rsp+10h], rbp
00007fff`ae7b37fa 4889742420        mov     qword ptr [rsp+20h], rsi
00007fff`ae7b37ff 57                push    rdi
00007fff`ae7b3800 4156              push    r14
00007fff`ae7b3802 4157              push    r15
00007fff`ae7b3804 4883ec20          sub     rsp, 20h
00007fff`ae7b3808 4d8bf0            mov     r14, r8
00007fff`ae7b380b 4c8bfa            mov     r15, rdx
00007fff`ae7b380e 488bf1            mov     rsi, rcx
00007fff`ae7b3811 4d85c0            test    r8, r8
00007fff`ae7b3814 750a              jne     MpOav!DllRegisterServer+0x1090 (00007fff`ae7b3820)
00007fff`ae7b3816 b857000780        mov     eax, 80070057h
00007fff`ae7b381b e96e010000        jmp     MpOav!DllRegisterServer+0x11fe (00007fff`ae7b398e)
00007fff`ae7b3820 41c70001000000    mov     dword ptr [r8], 1
00007fff`ae7b3827 80b9c800000000    cmp     byte ptr [rcx+0C8h], 0
00007fff`ae7b382e 7435              je      MpOav!DllRegisterServer+0x10d5 (00007fff`ae7b3865)
00007fff`ae7b3830 488d1d49fa0300    lea     rbx, [MpOav!DllRegisterServer+0x40af0 (00007fff`ae7f3280)]
00007fff`ae7b3837 488b0d42fa0300    mov     rcx, qword ptr [MpOav!DllRegisterServer+0x40af0 (00007fff`ae7f3280)]
00007fff`ae7b383e 483bcb            cmp     rcx, rbx
00007fff`ae7b3841 741b              je      MpOav!DllRegisterServer+0x10ce (00007fff`ae7b385e)
00007fff`ae7b3843 f6411c04          test    byte ptr [rcx+1Ch], 4
00007fff`ae7b3847 7415              je      MpOav!DllRegisterServer+0x10ce (00007fff`ae7b385e)
```


Well this makes it bit clear. First of all we not jumping into the middle of a function, See that `ret` instruction up there? What this tells us is, we jumped into a function but it is not labelled correctly.

However if you try to goto this address from a disassembler, it will fail. Indicating that this a function from another dll.

here's the memory map.



See? It seems like this dll is the COM dll that implements `IAmsiAntimalware` interface for windows defender.

To confirm that, let's check the registry.

// registry

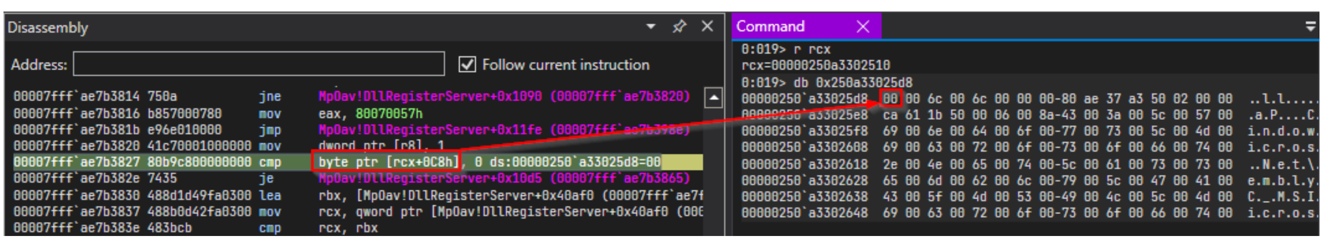
Now it is confirmed, let's go through this function.

```
MpOav!DllRegisterServer+0x1060:
00007fffae7b37f0 48895c2408      mov     qword ptr [rsp+8],rbx
00007fffae7b37f5 48896c2410      mov     qword ptr [rsp+10h],rbp
00007fffae7b37fa 4889742420      mov     qword ptr [rsp+20h],rsi
00007fffae7b37ff 57              push    rdi
00007fffae7b3800 4156            push    r14
00007fffae7b3802 4157            push    r15
00007fffae7b3804 4883ec20        sub     rsp,20h
00007fffae7b3808 4d8bf0          mov     r14,r8
00007fffae7b380b 4c8bfa          mov     r15,rdx
00007fffae7b380e 488bf1          mov     rsi,rcx
00007fffae7b3811 4d85c0          test    r8,r8
00007fffae7b3814 750a            jne     MpOav!DllRegisterServer+0x1090
(00007fffae7b3820)

MpOav!DllRegisterServer+0x1086:
00007fffae7b3816 b857000780      mov     eax,80070057h
00007fffae7b381b e96e010000      jmp     MpOav!DllRegisterServer+0x11fe
(00007fffae7b398e)
```

First it does some work on the stack frame and moves `0x80070057` to `rax` register if third parameter is null (pointer to a stack variable of CAmsiAntimalware::Scan method), And we know this is `E_INVALIDARG`. And then function jumps to the epilogue. So this is basically a small sanity check.

```
00007fffae7b3820 41c70001000000 mov     dword ptr [r8], 1
ds:000000931c9ce770=00000000
00007fffae7b3827 80b9c800000000 cmp     byte ptr [rcx+0C8h], 0 ds:00000250a33025d8=00
```

then it moves 1 or `AMSI_RESULT_NOT_DETECTED` into third parameter and checks if first parameter (rcx) + 200 is 0. We know that first parameter (rcx) passed down to this function is `CAmsiAntimalware->0x40`. (yes doesnt make much sense.)

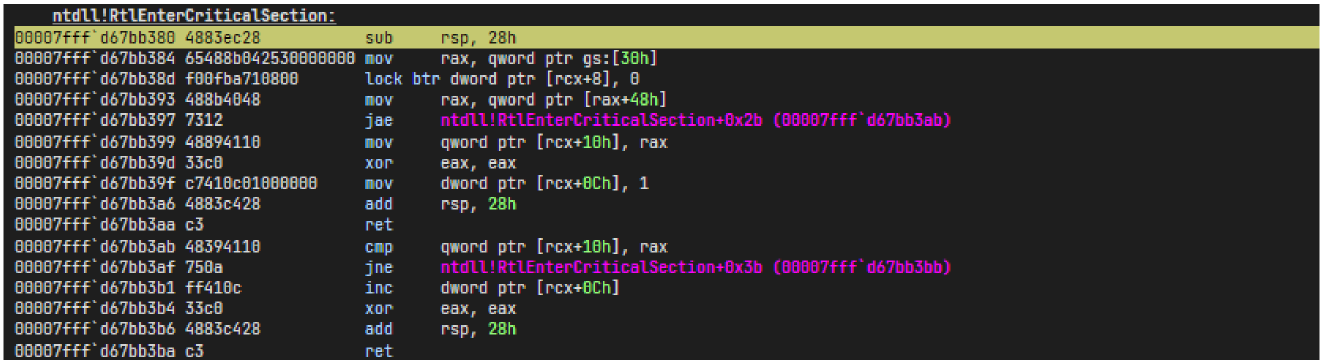


In our case, comparison turns out to be true.

A little below that, there's a call to another fuction from this dll.

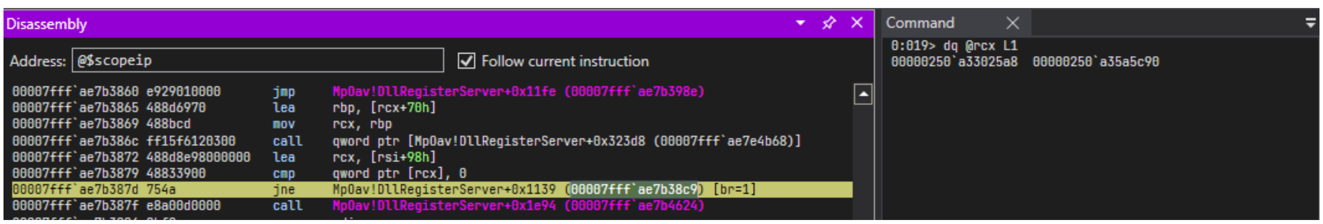
```
MpOav!DllRegisterServer+0x10d5:
00007fffae7b3865 488d6970        lea     rbp, [rcx+70h]
00007fffae7b3869 488bcd          mov     rcx, rbp
00007fffae7b386c ff15f6120300    call    qword ptr [MpOav!DllRegisterServer+0x323d8
(00007fffae7e4b68)]
```

it seems to take only one argument and it is `&rcx+0x70` .

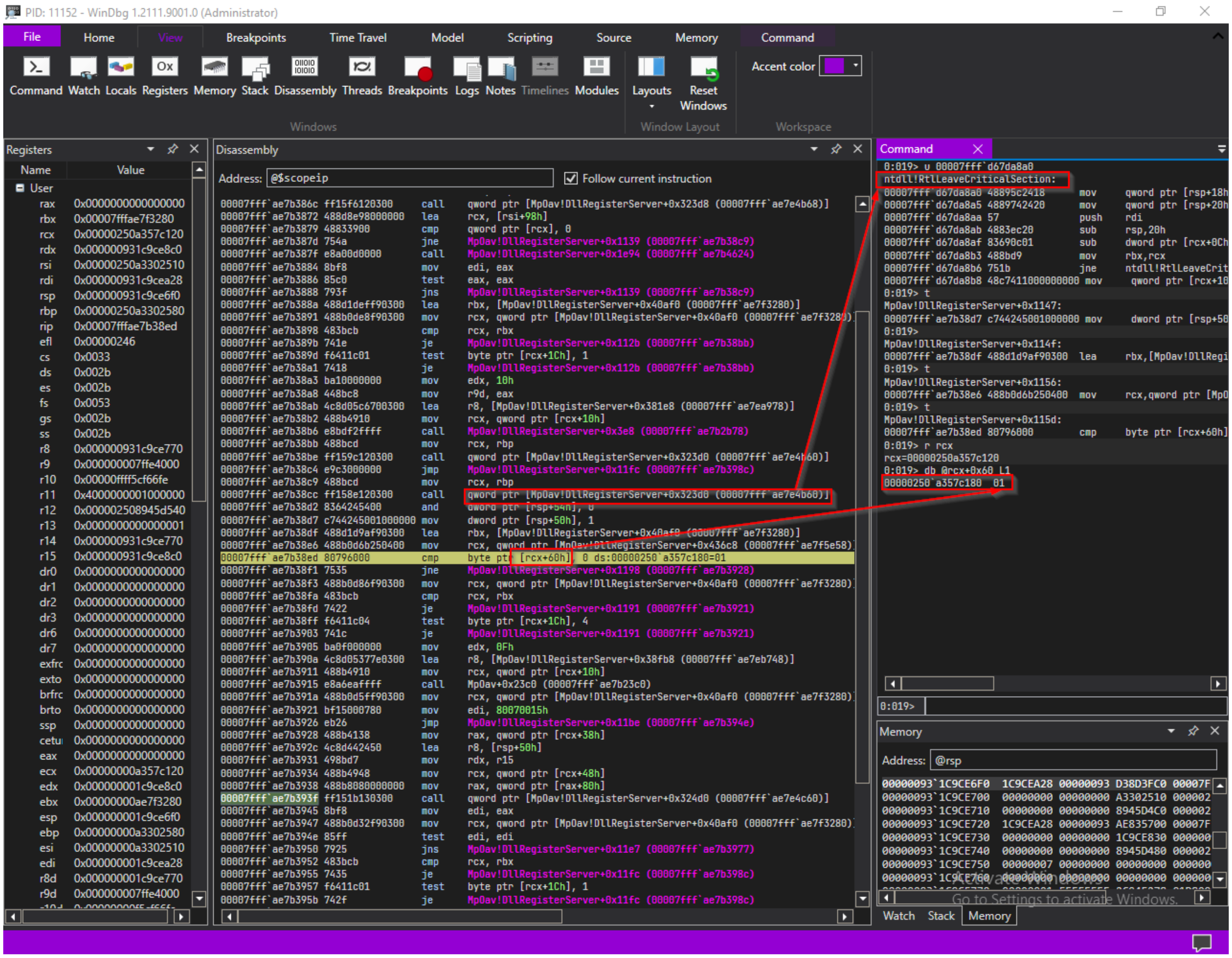


if we step into it, windbg indentifies function as `RtlEnterCriticalSection` from ntdll. According to msdn, `EnterCriticalState` function waits for ownership of the specified critical section object. The function returns when the calling thread is granted ownership. function accepts a single parameter and it is of `LPCRITICAL_SECTION` .

In this case, critical section that this function waits for is `rcx+0x70` .



next few instructions compare `rsi+0x98` with 0 (both rsi and rcx pointed to same address but since rcx now points to rcx+0x70, rsi is used). if comparison fails, it jumps to another location disassembly where `LeaveCriticalState` is being called.



as shown in the above diagram, function loads `rbp`, which points to the critical section (rsi->0x70) into `rcx`. Then `LeaveCriticalState` function is called.

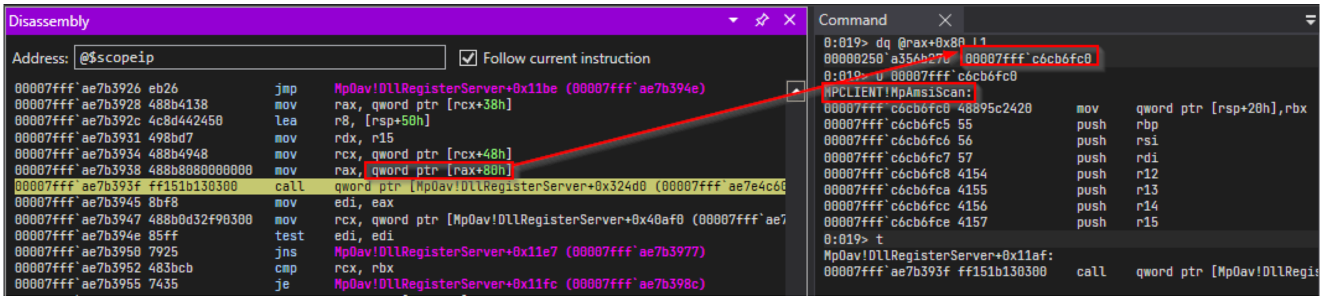
then two local variables, rsp+0x54 and rsp+0x50, get initialized to 0x0 and 0x1, following a `mov` instruction which loads a global variable into `rcx`. then it does a comparison of `rcx+0x60` with 0.

In our case, comparision fails and for that reason, jump will be taken.

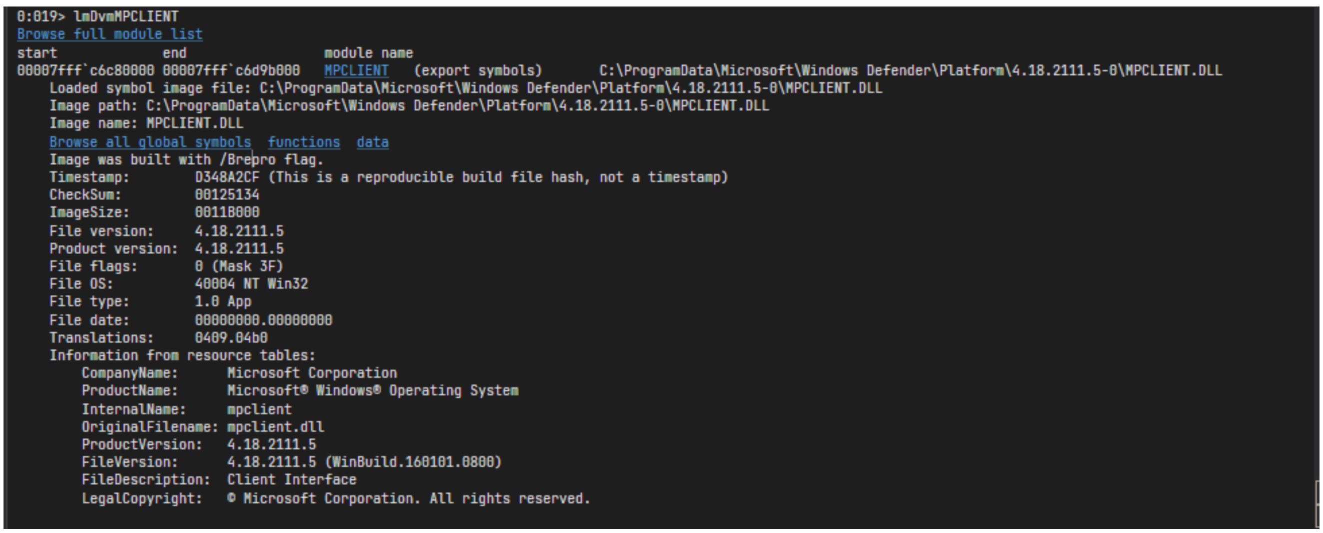
```
MpOav!DllRegisterServer+0x1198:
00007fffae7b3928 488b4138        mov     rax, qword ptr [rcx+38h]
ds:00000250a357c158=00000250a356b1f0
00007fffae7b392c 4c8d442450      lea     r8, [rsp+50h]
00007fffae7b3931 498bd7          mov     rdx, r15
00007fffae7b3934 488b4948        mov     rcx, qword ptr [rcx+48h]
```

here we can see another call.

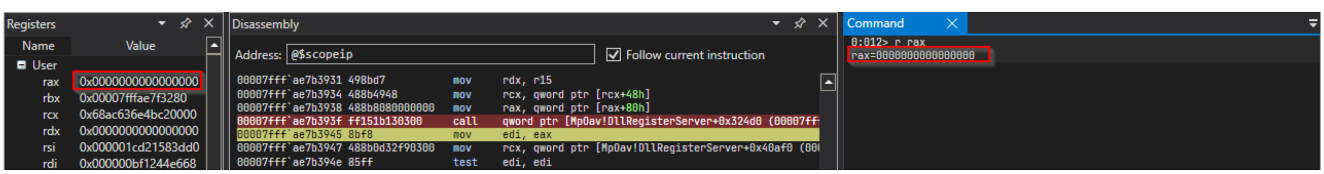
`rcx` is set to `[rcx+0x48]` and `rdx` is loaded with `amsiBuffer` meanwhile `r8`, third argument is loaded with address `rsp+0x50`.



as we can see in the above diagram, this call is to `MPCLIENT!MpAmsiScan` function. This is basically a function exported by windows defender's MPCLIENT.dll. So this means we have reached our destination.



Let's step over this function and inspect the return value since it is out of scope of this article to reverse engineer windows defender internals.

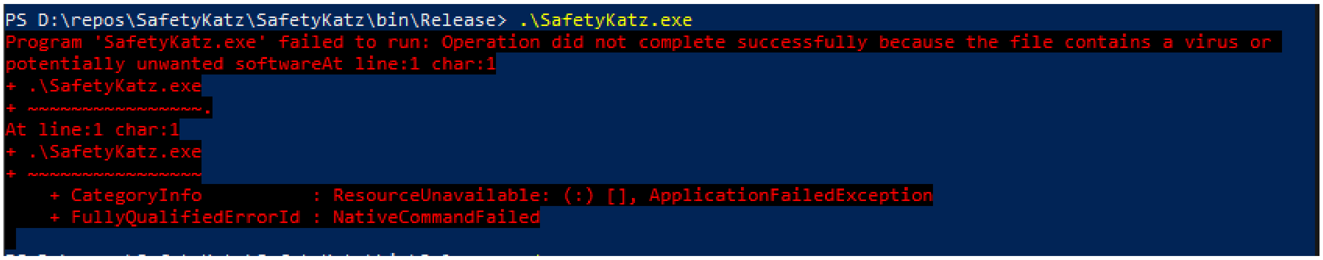


According the above diagram, the return value we get is 0x0. And there's no way to determine whether this is a indication of detection or not because windows documentation does not provide imformation about `MpAmsiScan`

Therefore we have to try some tricky methods to identify it.

First, im going to continue the exection.

as expected the result is,



Then we can place a breakpoint at the address where `MpAmsiScan` return and send some non-malicous input.

Weirdly enough, return value is same. So this function must be using an output parameter to pass the result of the scan, just like `AmsiScanBuffer` .

Can you remember that the third parameter to `MpAmsiScan` is a pointer to a local variable? Just in case, keep it's address in mind.

Somewhere down below, before the program generates an event saying safetykatz is malicious, return value or output parameter of `MpAmsiScan` must be accessed in order determine whether it's detected by windows defender or not.

Back to where we left off,

return value of `MpAmsiScan` is stored in `edi` register and function compares it with 0 after moving some value to `rcx` register.

```
00007fffae7b3945 8bf8              mov     edi, eax
00007fffae7b3947 488b0d32f90300    mov     rcx, qword ptr
[MpOav!DllRegisterServer+0x40af0 (00007fffae7f3280)]

MpOav!DllRegisterServer+0x11be:
00007fffae7b394e 85ff              test    edi, edi
00007fffae7b3950 7925              jns     MpOav!DllRegisterServer+0x11e7
(00007fffae7b3977) [br=1]
```

if return value (edi) is greater than or equal to zero,

```
MpOav!DllRegisterServer+0x11e7:
00007fffae7b3977 837c245401        cmp     dword ptr [rsp+54h], 1
00007fffae7b397c 0f94c0            sete    al
00007fffae7b397f 8886c8000000      mov     byte ptr [rsi+0C8h], al
00007fffae7b3985 8b442450          mov     eax, dword ptr [rsp+50h]
00007fff`ae7b3989 418906            mov     dword ptr [r14], eax
```

it sets value of third parameter (pointed by r14) to 1 and simply returns. Also note that return value is set to `edi` .

else if return value of `MpAmsiScan` (edi) is less than 0,

```
MpOav!DllRegisterServer+0x11c2:
00007fffae7b3952 483bcb          cmp     rcx,rbx
00007fffae7b3955 7435            je      MpOav!DllRegisterServer+0x11fc
(00007fffae7b398c)

MpOav!DllRegisterServer+0x11c7:
00007fffae7b3957 f6411c01        test    byte ptr [rcx+1Ch],1
00007fffae7b395b 742f            je      MpOav!DllRegisterServer+0x11fc
(00007fffae7b398c)

MpOav!DllRegisterServer+0x11cd:
00007fffae7b395d ba11000000      mov     edx,11h
00007fffae7b3962 448bcf          mov     r9d,edi
00007fffae7b3965 4c8d050c700300  lea     r8,[MpOav!DllRegisterServer+0x381e8
(00007fffae7ea978)]
00007fffae7b396c 488b4910        mov     rcx,qword ptr [rcx+10h]
00007fffae7b3970 e803f2ffff      call    MpOav!DllRegisterServer+0x3e8
(00007fffae7b2b78)
00007fffae7b3975 eb15            jmp     MpOav!DllRegisterServer+0x11fc
(00007fff`ae7b398c)
```

it checks validity of some data and calls a function and then returns after setting return value to that of `MpAmsiScan` stored in `edi` register, just like the previous one.

```
00007fffae7b398c 8bc7             mov     eax, edi
00007fffae7b398e 488b5c2440       mov     rbx, qword ptr [rsp+40h]
00007fffae7b3993 488b6c2448       mov     rbp, qword ptr [rsp+48h]
00007fffae7b3998 488b742458       mov     rsi, qword ptr [rsp+58h]
00007fffae7b399d 4883c420         add     rsp, 20h
00007fffae7b39a1 415f             pop     r15
00007fffae7b39a3 415e             pop     r14
00007fffae7b39a5 5f               pop     rdi
00007fff`ae7b39a6 c3               ret
```

Because the return value we got from `MpAmsiScan` is 0x0, execution path will be the first one we've discussed above.

There is something interesting that we havent discussed about that control flow path. There is a comparison of `rsp+0x54` and 1. if that comparison is able to set zero flag, next instruction sets `al` register to 1.

in our case, `rsp+0x54` is not equal to 1.

```
0:018> dd @rsp+0x54 L1
00000015`8864e564  00000000
```

which means, `al` wont be set to 1. If you can remember, `rsp+0x54` is only accessed once, just after the call to `LeaveCriticalState` and that that is the only instruction that sets `rsp+0x54` to 0x0. My guess is that this checks if function has entered the

`LeaveCriticalSection` block. It then sets `[rsi+0C8h]` (rsi == first parameter) to the value of `al`. Note that `rsi+0xc8` should be set to zero in order for this function to be sucessful. We discussed rest of this block earlier.

after the function returns, we'll end up back at `CAmsiAntimalware::Scan`. Good news is, we dont need to read every instruction since we already know what we are looking for.

```
GetSystemTimePreciseAsFileTime(&local_100);
pAVar8 = local_108;
uVar2 = (**(code **)(**R14 + 0x18))();
puVar3 = &local_f8;
local_f8 = 0;
GetSystemTimePreciseAsFileTime();
```

Above image shows how the call looks in decompiled pseudo code. return value of the callee is stored in local variable `uVar2`. However, we know this is not accurate because caller need to pass three args to the callee (we see none). That's not important to us though.

```
if ((loc_rand == 0) && (5 < (uint)DAT_7ff945511010)) {
  if (((_DAT_7ff945511020 & 0x400000000000) == 0) ||
     (bVar1 = true, (DAT_7ff945511028 & 0x400000000000) != DAT_7ff945511028)) {
    bVar1 = false;
  }
  if (bVar1) {
    local_a8 = &local_ec;
    local_ec = (undefined4)i;
    local_88 = &local_e0;
    local_78 = &local_100;
    local_f8 = local_f8 & 0xffffffff00000000 | (ulonglong)*R8;
    local_68 = &local_f8;
    local_a0 = 4;
    local_90 = 0x10;
    local_80 = 8;
    local_100 = local_100 & 0xffffffff00000000 | (ulonglong)uVar2;
    local_70 = 4;
    local_60 = 4;
    local_98 = R12;
    _TlgWrite(puVar3,&DAT_7ff94550e5fa,pAVar8,local_e0,7,local_c8);
  }
}
```

Here, the if confition only evaluate true when `loc_rand()` is equal to zero and a global variable is less than 5. `loc_rand` is basically the local variable where the random number was stored. Therefore this block is not going to execute.

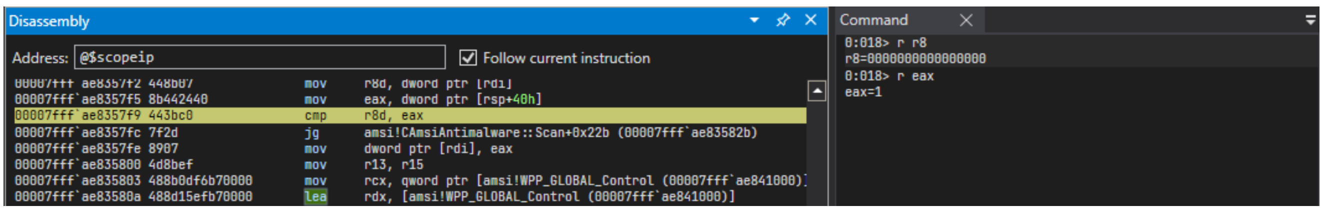
```
if (uVar2 == 0) {
  pAVar8 = (AMSI_RESULT *)(ulonglong)*R8;
  if ((int)local_108[0] < (int)*R8) {
    if (((undefined **)WPP_GLOBAL_Control != &WPP_GLOBAL_Control) &&
       ((WPP_GLOBAL_Control[0x1c] & 4) != 0)) {
      uVar4 = 0x20;
      goto LAB_7ff945505853;
    }
  }
  else {                                         <---- r8 <= eax
    *R8 = local_108[0];
    R13 = i;
    if (((undefined **)WPP_GLOBAL_Control != &WPP_GLOBAL_Control) &&
       ((WPP_GLOBAL_Control[0x1c] & 4) != 0)) {
      uVar4 = 0x1f;
LAB_7ff945505853:
      WPP_SF_PDDI(*(undefined8 *)(WPP_GLOBAL_Control + 0x10),uVar4,pAVar8,i);
    }
  }
```

Above if condition checks if return value (stored in r14) is zero. In our case it is. we know that the third argument passed to the collee is the address of `rsp+0x40` and was passed through `r8`.

below image shows disassembly of the above snippet

```
amsi!CAmsiAntimalware::Scan+0x1ed:
00007fffae8357ed 4585f6          test    r14d,r14d
00007fffae8357f0 757f            jne     amsi!CAmsiAntimalware::Scan+0x271
(00007fffae835871)

amsi!CAmsiAntimalware::Scan+0x1f2:
00007fffae8357f2 448b07          mov     r8d,dword ptr [rdi]
00007fffae8357f5 8b442440        mov     eax,dword ptr [rsp+40h]
00007fffae8357f9 443bc0          cmp     r8d,eax
00007fffae8357fc 7f2d            jg      amsi!CAmsiAntimalware::Scan+0x22b
(00007fffae83582b)
```

As shown above, `mov r8d, dword ptr[rdi]` moves value at address stored in `rdi` into `r8` register. `rdi` stores the address of `AMSI_RESULT` enum passed down to `CAmsiAntimalware::Scan` method. it then moves `rsp+0x40`, output paramater we discussed earlier into `eax` register.



comparison instruction and jump instruction checks if value in `r8` (result) is greater than that of in `eax` (output parameter). jump wont be taken and execution will directed to the next mov instruction.

This is basically checking if current scan's result is greater than that of previous one.

```
amsi!CAmsiAntimalware::Scan+0x1fe:
00007fffae8357fe 8907           mov     dword ptr [rdi],eax
00007fffae835800 4d8bef         mov     r13,r15
00007fffae835803 488b0df6b70000 mov     rcx,qword ptr [amsi!WPP_GLOBAL_Control
(00007fffae841000)]
00007fffae83580a 488d15efb70000 lea     rdx,[amsi!WPP_GLOBAL_Control
(00007fffae841000)]
00007fffae835811 483bca         cmp     rcx,rdx
00007fffae835814 7451           je      amsi!CAmsiAntimalware::Scan+0x267
(00007fffae835867)

amsi!CAmsiAntimalware::Scan+0x216:
00007fffae835816 f6411c04       test    byte ptr [rcx+1Ch],4
00007fffae83581a 744b           je      amsi!CAmsiAntimalware::Scan+0x267
(00007fffae835867)

amsi!CAmsiAntimalware::Scan+0x21c:
00007fffae83581c 4c894c2430     mov     qword ptr [rsp+30h],r9
00007fffae835821 418d561f       lea     edx,[r14+1Fh]
00007fffae835825 89442428       mov     dword ptr [rsp+28h],eax
00007fffae835829 eb28           jmp     amsi!CAmsiAntimalware::Scan+0x253
(00007fff`ae835853)
```

In the above snippet it loads `eax` into `[rdi]` , and value of `r15` into `r13` and compare some global variables related to `WPP` .

According to the decompiled snippet, this checks some global variables related to WPP tracer and if checks are valid, it jumps to a location in disassembly after setting `rdx` register to the address `r14 + 1f` . Well this has nothing to do with addresses eventhough the instruction is `lea` . `r14` is 0x0. therefore what this does is, it loads `0x1f` into `rdx` register.

However, if we step through each instruction, `cmp rcx, rdx` will evaluate to 0x0 and the jump will be taken.

```
amsi!CAmsiAntimalware::Scan+0x267:
00007fffae835867 813f00800000   cmp     dword ptr [rdi],8000h
00007fffae83586d 7d50           jge     amsi!CAmsiAntimalware::Scan+0x2bf
(00007fffae8358bf)

amsi!CAmsiAntimalware::Scan+0x26f:
00007fffae83586f eb34           jmp     amsi!CAmsiAntimalware::Scan+0x2a5
(00007fff`ae8358a5)
```

in the above snippet, dword value at address stored in `rdi` is compared to hex 0x8000, decimal 32768. Aand this is exactly the same value msdn specifies in their documentation for `AMSI_RESULT` enum. quoting msdn,

```
    'Any return result equal to or larger than 32768 is considered malware, and the
content
    should be blocked. An app should use AmsiResultIsMalware to determine if this is
the case.'
```

next instruction is a `jge` and it essentially takes the jump if dword at address stored in `rdi` (AMSI_RESULT) is greater than or equal to 0x8000. if it is, it breaks from the loop.

In our case, value at address stored in `rdi` is less than 0x8000 so the jump won't be taken. Instead control flow will be redirected to

```
amsi!CAmsiAntimalware::Scan+0x2a5:
00007fffae8358a5 488344246008    add     qword ptr [rsp+60h],8
00007fffae8358ab 49ffc7          inc     r15
00007fffae8358ae 4983c410        add     r12,10h
00007fffae8358b2 4c3bbec0010000  cmp     r15,qword ptr [rsi+1C0h]
00007fffae8358b9 0f820efeffff    jb      amsi!CAmsiAntimalware::Scan+0xcd
(00007fffae8356cd)
```

`r15` is incremented by 1 and it is then compared to `this->0x1c0`, whose value is 1. if `r15` is below that value, it will jump to the address where the loop begins.

Possibly, the loop is going through every registered anti-malware vendor's COM interface. Since I dont have any anti malware services installed in the VM, its going to loop only once. This also uncovers some details about `CAmsiAntimalware` class members. The loop terminates after loop iterator veriable being compared to `this->0x1c0`. Therefore `this->0x1c0` is the value that indicates number of registered anti malware services or AMSI providers.

Now the question is, we just executed a malicous program and it just got flagged as `AMSI_RESULT_NOT_DETECTED`. But we still see powershell produces that red ugly output saying that it detected a malicous program.

And suprisingly, there's no call to `AmsiResultIsMalware`.

```
00007fffae8358c4 4c3baec0010000  cmp     r13,qword ptr [rsi+1C0h]
00007fffae8358cb 732a            jae     amsi!CAmsiAntimalware::Scan+0x2f7
(00007fffae8358f7)

amsi!CAmsiAntimalware::Scan+0x2cd:
00007fffae8358cd 448bf3          mov     r14d,ebx
00007fffae8358d0 4d85e4          test    r12,r12
00007fffae8358d3 7428            je      amsi!CAmsiAntimalware::Scan+0x2fd
(00007fff`ae8358fd)
```

First if condition checks if `r13` register is less than the number of providers (this->0x1c0). We saw that `r15`, which acts as the counter loaded into `r13` previously. What this is checking is that if anything malicous detected before going through all the providers.

Now it is time to conclude our assumptions on AmsiInitialize.

AmsiInitialize

The End

So yeah that's it for now… we explored AMSI in-depth in this article. In the next one, We will go through some common AMSI bypass techniques.

#Spread Anarchy!