A PART OF ANTI-VIRUS 2

公開サンプルコードで学ぶ
Windows Antimalware Scan Interface(AMSI)

@KASH1064

A Part of AntiVirus 2

公開コードで学ぶ **Windows Antimalware Scan Interface(AMSI)**

かしわば **(@kash1064)**　著

2025-11-15 版　　発行

本書の内容はすべて、公式ドキュメントやその他の Web サイトに記載の情報、もしくは一般に公開されている書籍やソースコードなどの情報を元に作成しています。技術的な内容について言及する場合は、可能な限り出典を注釈に明記します。また、本書に記載の内容は全て筆者自身の見解であり、所属する企業や組織を代表するものではありません。

まえがき

本書について

本書をお手に取っていただき誠にありがとうございます。かしわば (@kash1064) と申します。

私は主にリバースエンジニアリングやフォレンジックの技術領域に関心を持っており、最近はもっぱら、子育ての傍らさまざまなアプリケーションのソースコードや技術書を読む生活をしています。

また、本業はセキュリティ製品ベンダーのテクニカルサポートエンジニアで、これまでに 2 つのセキュリティ製品ベンダーにて、Windows や Linux 用のアンチマルウェア製品のトラブルシューティングやデバッグを行ってきました。

本書では、前回頒布した「A part of Anti-Virus -サンプルコードで学ぶ Windows AntiVirus とミニフィルタドライバー-」[*1]に続き、公式の Windows classic samples リポジトリ[*2]で公開されている AmsiStream および AmsiProvider のサンプルコードをベースに、AMSI (Windows Antimalware Scan Interface) の概要と仕組みを解説します。

詳しくは 1 章に記載しますが、AMSI とは、従来のディスク上のファイルをスキャンする方法では検知できない動的なスクリプトベースのマルウェアなどへの対策として、Windows 10 から導入された仕組みです。

[*1] A part of Anti-Virus: https://techbookfest.org/product/iFrVq6PX0PPJhivrGzhi32

[*2] Windows-classic-samples： https://github.com/microsoft/Windows-classic-samples/

AMSI は Windows において PowerShell や Office VBA などさまざまなコンポーネントに統合されており、システムにインストールされているアンチマルウェア製品と連動してユーザーとシステムを保護しています。[*3]

このように、特定の脅威の検知のために非常に重要な保護機能として利用されている AMSI ですが、その仕組みのわかりづらさも相まってか、残念ながらほとんどのユーザーや管理者にとってはなじみのない機能であるようです。

実際のところ、AMSI のような OS やアンチマルウェア製品が提供する多くのセキュリティ機能はユーザーの可用性とのトレードオフにより提供されているため、その制限や過検知などによってシステムの運用上の障害が発生する可能性がある点は事実です。

しかし、何らかの問題が発生した際、システムの管理者の各セキュリティ機能の重要性に関する理解が不十分な場合には、リスクに対する検討が不十分なまま機能が無効化され、本来防げたはずの脅威による侵害を許してしまう恐れがあるため、主要なセキュリティ機能の必要性については幅広いユーザーに理解してもらうことが重要と考えております。

そこで、本書では AMSI の概要や利点を紹介するとともに、公開されているサンプルコードからその仕組みを解説することを通じて、AMSI の機能について多くのユーザーに理解してもらうことを目的としています。

本書の内容について

本書では、AMSI の機能や利点について読者の理解を深めることを目的として、各章で以下の内容について解説します。

[*3] AMSI と統合される Windows コンポーネント https://learn.microsoft.com/ja-jp/windows/win32/amsi/antimalware-scan-interface-portal#windows-components-that-integrate-with-amsi

- 1 章： AMSI の概要とその利点について解説します
- 2 章： AmsiStream サンプルコードを使用し、AMSI によるスキャンをアプリケーションから要求するためのクライアントインターフェースについて解説します
- 3 章： AmsiProvider サンプルコードを使用し、アプリケーションから受け取ったスキャン要求を AMSI プロバイダーが処理する方法を解説します
- 4 章： AmsiProvider サンプルコードをカスタマイズし、特定のコンテンツの実行をブロックする AMSI プロバイダーを作成します
- 5 章： 公開されている PowerShell のソースコードから、実際のアプリケーションがどのように AMSI を利用しているのかを解説します

なお、本書ではいわゆる AMSI Bypass と呼ばれる、AMSI による保護を回避して悪意のあるコードを実行する攻撃方法については扱いません。

目次

第 1 章

AMSI (Windows Antimalware Scan Interface) について

1.1 AMSI の概要

Windows 10 から導入された AMSI (Windows Antimalware Scan Interface) とは、一言で表現すると「アプリケーションやサービスをマシン内で稼働するアンチマルウェア製品と統合するための汎用的な標準インターフェース」です。

以下は、簡略化された AMSI のイメージ図です。

AMSI は、このイメージ図のように各アプリケーション側から AMSI インターフェースを通してシステム内で稼働するアンチマルウェア製品にスキャン対象のデータを連携し、そのスキャン結果を受け取ります。

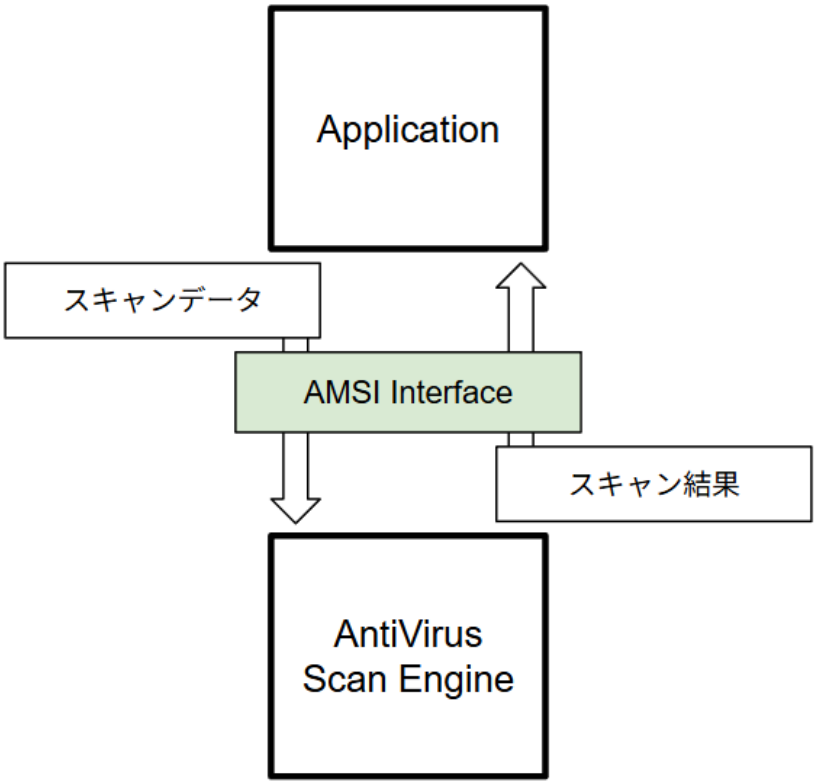

図 1.1: 簡略化された AMSI のイメージ図

上図のように、アプリケーションは AMSI を使用して自身が処理するデータが悪意のあるものか否かを判断するためにシステムに登録されているアンチマルウェア製品を利用することができます。

また、AMSI はアンチマルウェア製品ベンダーには依存しない標準的なインターフェースを提供しており、さまざまな製品ベンダーがデータをスキャ

ンするための AMSI プロバイダーをシステムに登録することができます。[*1]

システムに登録可能な AMSI プロバイダーは、Windows 標準の Microsoft Defender ウイルス対策をはじめとして、多くのアンチマルウェア製品ベンダーにより提供されています。

AMSI プロバイダーを提供している製品ベンダーの一覧については、whoamsi リポジトリから確認することができますが、残念ながら直近数年間は情報が更新されていないため最新情報の網羅性には留意が必要です。[*2]

AMSI を利用するアプリケーション

AMSI は、Windows において例えば以下のようなコンポーネントに統合されています。

- User Account Control - UAC (EXE、COM、MSI の権限昇格や ActiveX インストール)
- PowerShell (スクリプト、対話インターフェース、または動的なコードの評価)
- Windows Script Host (wscript.exe や cscript.exe)
- JavaScript and VBScript
- Office VBA macros

また、上記以外にも例えば SharePoint Server や Exchange Server などのサービスで AMSI による保護を利用することができます。[*3] [*4]

各サービスとアプリケーションは、いずれも AMSI を利用して不正なコー

[*1] Antimalware Scan Interface (AMSI) https://learn.microsoft.com/ja-jp/windows/win32/amsi/antimalware-scan-interface-portal

[*2] whoamsi https://github.com/subat0mik/whoamsi

[*3] Configure AMSI integration with SharePoint Server https://learn.microsoft.com/en-us/sharepoint/security-for-sharepoint-server/configure-amsi-integration

[*4] Exchange Server AMSI integration https://learn.microsoft.com/ja-jp/exchange/antispam-and-antimalware/amsi-integration-with-exchange

ドの実行や悪意のある Web リクエストなどをブロックすることでユーザーを保護しています。

1.2 AMSI による端末の保護

スクリプトベースの脅威について

従来のアンチマルウェア製品は、マルウェアの実行ファイルがディスク上に作成されることを前提としていました。

しかし、AMSI がリリースされた 2015 年までには、PowerShell などのシステムに導入されているツールを使用したスクリプトベースの攻撃が急速に広まっていました。[*5]

このようなスクリプトベースの攻撃は OS 標準のツールを使用します。[*6]

さらに、これらの攻撃は多くの場合メモリ内で悪意のあるコードの作成や実行が行われ、ディスク上にファイルを作成しません。[*7]

また、もしマルウェアがファイルを介して配布される場合でも、スクリプトベースのマルウェアの場合は高度な暗号化または難読化手法を用いることで、従来のシグネチャマッチングによる検知を非常に困難にすることが容易でした。[*8]

[*5] INVESTIGATING POWERSHELL ATTACKS (FireEye,Inc. / 2014 年)

[*6] 「Living off the land」と呼ばれる、システム内にインストールされている正規ツールを悪用する攻撃手法。

[*7] 「ファイルレスマルウェア」と呼ばれる、ファイルの作成やダウンロード/インストールを伴わず、メモリ内で動作することで攻撃を行うマルウェア。

[*8] How the Antimalware Scan Interface (AMSI) helps you defend against malware https://learn.microsoft.com/ja-jp/windows/win32/amsi/how-amsi-helps

```powershell
function Invoke-Evil
{
    $xorKey = 123

    $code = "LHsJexJ7D3see1Z7M3sUewh7D3tbe1x7C3sMexV7H3tae1x7"
    $bytes = [Convert]::FromBase64String($code)
    $newBytes = foreach($byte in $bytes) { $byte -bxor $xorKey }

    $newCode = [System.Text.Encoding]::Unicode.GetString($newBytes)

    Invoke-Expression $newCode
}
Invoke-Evil
```

図 1.2: 難読化されたスクリプトの例 (公開情報より引用)

そのため、上記のような攻撃手法はアンチマルウェア製品の検知を回避して攻撃を成功させる目的で多くの悪意のある攻撃者に用いられることとなり、非常に大きな脅威となっていました。

AMSI による保護の強化

しかし、AMSI のリリースはこのような脅威に対する保護を大幅に強化しました。[*9]

例えば、PowerShell を利用したファイルレスマルウェア攻撃が行われる場合でも、メモリ内に配置されている悪意のある実行コードを AMSI を通してスキャンすることにより、攻撃を未然に防ぎシステムを保護することができます。

また、仮にスクリプトに難読化の手法が用いられている場合でも、難読化解除プロセスを経て最終的にスクリプトエンジンに与えられるプレーンな実行コードを AMSI によりスキャンすることができるため、攻撃者が検知を回避することは非常に困難になりました。

[*9] How the Antimalware Scan Interface (AMSI) helps you defend against malware https://learn.microsoft.com/ja-jp/windows/win32/amsi/how-amsi-helps

AMSI のその他の利点

また、AMSI により強化された保護により対処が可能となるのは、前項で紹介したスクリプトベースの脅威に留まりません。

AMSI によるスキャン要求は PowerShell などのスクリプトエンジン以外にも、さまざまな種類のアプリケーションに組み込むことができます。

そのため、外部から受け取った Web リクエストや、ユーザーが入力フォームから投稿したテキストなど、さまざまなコンテンツを AMSI を通してスキャンし、悪意のある要求やテキストの送信などをブロックすることも可能です。

悪意ある攻撃者との戦い

ここまでに紹介してきた通り、AMSI はそのリリース以降さまざまなアプリケーションに統合され、多くの脅威からユーザーとシステムを保護してきました。

しかし、他のあらゆるセキュリティ製品の例に漏れず、AMSI もより巧妙に検知を回避しようとする攻撃者とのいたちごっこを続けています。

例えば、2018 年には AMSI が Office VBA に新たに統合され、当時から増加していた VBA マクロを使用してアンチマルウェア製品の検知を回避するマルウェアを効果的に検知/ブロックできるようになりました。[*10]

しかし、AMSI による Office VBA マクロを悪用するマルウェアに対する検知能力は向上したものの、Office VBA に統合された AMSI を回避す

[*10] Office VBA + AMSI: Parting the veil on malicious macros https://www.microsoft.com/en-us/security/blog/2018/09/12/office-vba-amsi-parting-the-veil-on-malicious-macros/

るために、1993 年の VBA 導入以前にリリースされた Excel 4.0(XLM) マクロを悪用する手法が増加しました。[*11]

このようなトレンドに対処するため、Microsoft は 2021 年に Excel 4.0(XLM) マクロにも AMSI を統合しました。

※ なお、本書執筆時点では、Excel 4.0(XLM) マクロはユーザーの保護を目的に既定で無効化されています。[*12]

以上のように、AMSI は悪意のある攻撃者により次々と生み出される新しい攻撃手法に対抗するために、さまざまなアプリケーションへの統合を行うことでユーザーとシステムを保護してきました。

もちろん、悪意のある攻撃者が使用する手法は日々進化を続けているため、AMSI による保護は絶対ではありません。

しかし、少なくとも AMSI はスクリプトエンジンや Office マクロを悪用する脅威に対抗するための優れたソリューションの 1 つである点は間違いありません。

そのため、AMSI のような重要なセキュリティ機能をリスクに対する検討が不十分なまま無効化するようなことがないよう、ユーザーやシステムの管理者はこれらの機能がどのような脅威からシステムを保護しているのかについて、十分に把握しておくことが重要であると考えます。

[*11] XLM + AMSI: New runtime defense against Excel 4.0 macro malware https://www.microsoft.com/en-us/security/blog/2021/03/03/xlm-amsi-new-runtime-defense-against-excel-4-0-macro-malware/

[*12] Excel 4.0 (XLM) macros now restricted by default for customer protection https://techcommunity.microsoft.com/blog/excelblog/excel-4-0-xlm-macros-now-restricted-by-default-for-customer-protection/3057905

1.3 アプリケーションが AMSI スキャンを利用する方法

では、このような利点を持つ AMSI スキャンを各アプリケーションがどのように使用するのかを見ていきます。

任意のアプリケーションは、以下の図のように AMSI を利用するためのインターフェースとして用意されている API を使用して、システムに登録されている AMSI プロバイダーにスキャンを要求することができます。

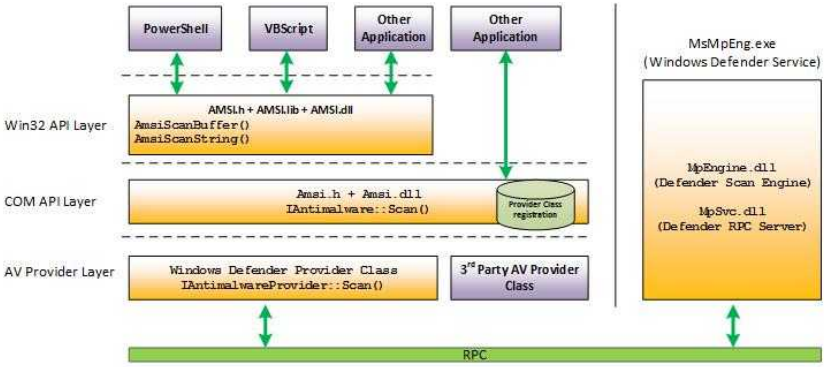

図 1.3: AMSI のイメージ図 (公開情報より引用)

スキャン要求を受け取った AMSI プロバイダーは通常、必要に応じてシステムにインストールされているアンチマルウェアスキャンエンジンを利用し、そのスキャン要求が行われたコンテンツが悪意のあるものであるか否かを確認し、その結果を返します。

なお、既定では MpOav.dll が AMSI プロバイダーとして登録されており、システム内で稼働する Microsoft Defender ウイルス対策が AMSI API

を使用して要求されたデータのスキャンを行います。[*13]

システムに登録されている AMSI プロバイダーは、要求されたスキャンに対するスキャン結果に応じて、通常は以下のいずれかの種類の結果を返します。[*14]

- AMSI_RESULT_CLEAN
- AMSI_RESULT_NOT_DETECTED
- AMSI_RESULT_BLOCKED_BY_ADMIN_START
- AMSI_RESULT_BLOCKED_BY_ADMIN_END
- AMSI_RESULT_DETECTED

詳細については 2 章以降に記載しますが、例えばアプリケーションが行ったスキャン要求に対して AMSI_RESULT_NOT_DETECTED の結果が返された場合、スキャン要求を行ったアプリケーション側でスクリプトの実行ブロックなどの操作を行い、端末を保護することができます。

システムに登録されているプロバイダーを確認する

続けて、システムに登録されている AMSI プロバイダーの情報を確認します。

まず、システムに登録されている AMSI プロバイダーの一覧はレジストリキー HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\AMSI\Providers から確認できます。

既定では、HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\AMSI\Providers 直下に AMSI プロバイダーの ID である {2781761E-28E0-4109-99F

[*13] Evading EDR P.186 (Matt Hand 著 / No Starch Press / 2023 年)

[*14] AMSI_RESULT enumeration https://learn.microsoft.com/ja-jp/windows/win32/api/amsi/ne-amsi-amsi_result

E-B9D127C57AFE} が登録されていることを確認できます。

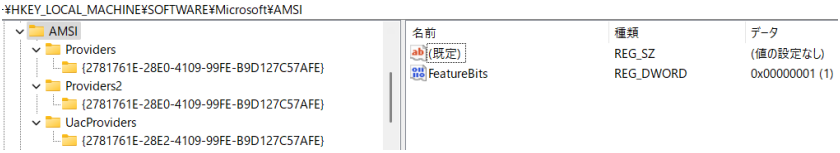

図 1.4: 既定で登録されている AMSI プロバイダーの CLSID

また、この ID は COM サーバーとして登録されている AMSI プロバイダーの CLSID に一致します。

そのため、レジストリキー HKEY_LOCAL_MACHINE\SOFTWARE\Classes\CLSID\{CLSID} から一致する CLSID のレジストリキーを確認すると、以下のスクリーンショットのように HKEY_LOCAL_MACHINE\SOFTWARE\Classes\CLSID\{2781761E-28E0-4109-99FE-B9D127C57AFE} のレジストリキーが登録されていることを確認でき、ここから Microsoft Defender のコンポーネントに関する情報を参照できることがわかります。

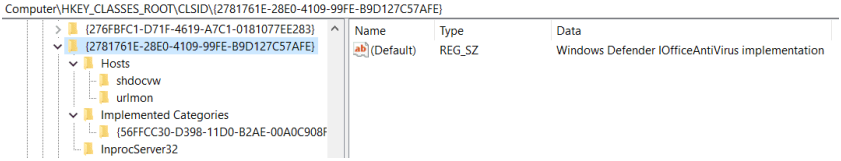

図 1.5: 既定で登録されている AMSI プロバイダー

なお、上記の例ではシステムに既定でインストールされている Microsoft Defender IOfficeAntivirus のみが AMSI プロバイダーとして登録されていますが、サードパーティのアンチマルウェア製品をインストールしている場合には、サードパーティ製の AMSI プロバイダーが登録されている場合があります。

1.4 AMSI によりスクリプトの実行をブロックする

続いて、実際に AMSI により PowerShell や VBScript のスキャンが行われ、検知される動作を確認してみます。[*15]

また、今回は単にスクリプトの実行がブロックされることを確認するだけでなく、ETW プロバイダー `Microsoft-Antimalware-Scan-Interface` から出力されるイベントを確認することで、実際に AMSI によりスキャンされ、検知されたコンテンツを特定してみることにします。

事前準備

AMSI による検知テストと、検知時の ETW トレースを取得するため、仮想マシンなどにインストールした Windows 端末で事前に以下の操作を実施します。

1. logman を使用して ETW イベントトレースセッションを開始する。

まずは、管理者権限で起動したコマンドプロンプトで以下のコマンドを実行し、ETW プロバイダー `Microsoft-Antimalware-Scan-Interface` から出力されるイベントを取得するための ETW イベントトレースセッションを開始します。

```
logman start AMSITrace -p Microsoft-Antimalware-Scan-Interf
ace Event1 -bs 64 -nb 16 256 -o %USERPROFILE%\Downloads\AMS
ITrace.etl -ets
```

[*15] AMSI demonstrations with Microsoft Defender for Endpoint https://learn.microsoft.com/ja-jp/defender-endpoint/mde-demonstration-amsi

このコマンドにより、logman ツールを使用して `Microsoft-Antimalware-Scan-Interface` という ETW プロバイダーが出力する情報をキャプチャできるようになります。[*16]

キャプチャしたイベント情報は、上記のコマンド実行時のユーザープロファイル直下の Downloads フォルダに AMSITrace.etl ファイルとして保存されます。

ファイルを特定のフォルダに配置したい場合には、`-o` オプションで指定している出力先のファイルパスを変更できます。

開始した ETW イベントトレースセッションを停止してイベントのキャプチャを停止するには、以下のコマンドを実行します。

```
logman stop AMSITrace -ets
```

なお、注釈に記載している "Better know a data source: Antimalware Scan Interface" などの多くのセキュリティベンダーの公開ブログなどでは、以下のように `-bs 64 -nb 16 256` ETW トレースセッションのバッファサイズに関するオプションを含まないコマンド例が紹介されています。[*17]

```
logman start AMSITrace -p Microsoft-Antimalware-Scan-Interf
ace Event1 -o AMSITrace.etl -ets
```

[*16] logman https://learn.microsoft.com/ja-jp/windows-server/administration/windows-commands/logman

[*17] Better know a data source: Antimalware Scan Interface https://redcanary.com/blog/threat-detection/better-know-a-data-source/amsi/

しかし、 十分なバッファサイズを指定せずに ETW イベントトレースセッションを開始した場合には、サイズの大きなイベントがドロップされ、保存した ETL ファイルに含まれない場合があるので注意が必要です。[*18]

なお、指定したバッファサイズより大きな ETW イベントがロストしたかどうかを確認したい場合は、logman ではなく Windows Performance Toolkit (WPT) に含まれる Xperf を使用します。 [*19]

```
# トレースセッションの開始
xperf.exe -start AMSITrace -on Microsoft-Antimalware-Scan-I
nterface -BufferSize 64 -MinBuffers 16 -MaxBuffers 256 -f %
USERPROFILE%\Downloads\AMSITrace.etl

# トレースセッションの終了
xperf.exe -stop AMSITrace
```

Xperf を使用すると、ETW イベントのロストが発生した場合は、`xperf: warning: Session "AMSITrace" lost 1 events.` のような警告を表示してくれます。

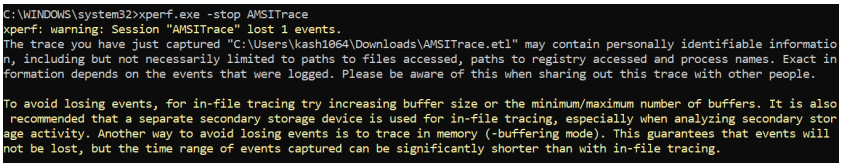

図 1.6: イベントのロストが発生した場合の警告

[*18] logman create trace https://learn.microsoft.com/ja-jp/windows-server/administration/windows-commands/logman-create-trace

[*19] Windows Performance Toolkit https://learn.microsoft.com/ja-jp/windows-hardware/test/wpt/

2. Get-AMSIEvent コマンドレットをインポートする

ETW イベントトレースを取得できるようになったら、次は取得したイベントを解析するためのツールをインストールします。

取得した ETW トレースイベントは、標準の Get-WinEvent コマンドレットで解析することができますが、このコマンドレットを使用する場合、取得した `AMSI/Debug` イベントの各フィールドを上手く解釈することができません。[*20]

そのため、今回は前述の "Better know a data source: Antimalware Scan Interface" という記事の中で提供されている AMSITools.psm1 を使用することにします。

このツールを使用するため、まずはフォークした AMSITools.psm1 のソースコードを以下のリポジトリからダウンロードし、マシンの任意のパスに配置します。

https://gist.github.com/kash1064/7e752a71f6ef48ab32e1323d764b8774

続いて、取得した AMSITools.psm1 を以下のコマンドレットでインポートし、イベント解析のためのヘルパー関数 Get-AMSIEvent を使用できるようにします。

[*20] Microsoft.PowerShell.Diagnostics Get-WinEvent https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.diagnostics/get-winevent

```
Import-Module C:\Tools\AMSI\AMSITools.psm1
```

ツールのインポートが完了すると、以下のコマンドで保存した ETL ファイルを解析できます。

```
Get-AMSIEvent "$($env:USERPROFILE)\Downloads\AMSITrace.etl"
```

AMSI で PowerShell スクリプトの実行をブロックする

Microsoft-Antimalware-Scan-Interface から出力されるイベントを取得するための準備ができたので、まずは PowerShell スクリプトの実行を AMSI でブロックする動作を確認します。

AMSI のテストには、テスト用に公開ドキュメントで提供されている以下のスクリプトを使用します。[*21]

```
# Save this sample AMSI powershell script as AMSI_PoSh_scri
pt.ps1
$testString = "AMSI Test Sample: " + "7e72c3ce-861b-4339-87
40-0ac1484c1386"
Invoke-Expression $testString
```

上記のスクリプトを evil.ps1 などのファイル名で端末に保存して PowerShell で実行すると、以下のようにスクリプトの実行がブロックされること

[*21] AMSI demonstrations with Microsoft Defender for Endpoint https://learn.microsoft.com/ja-jp/defender-endpoint/mde-demonstration-amsi

を確認できます。

```
PS C:\Users\kash1064\Downloads> .\evil.ps1
Invoke-Expression : At line:1 char:1
+ AMSI Test Sample: 7e72c3ce-861b-4339-8740-0ac1484c1386
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
This script contains malicious content and has been blocked by your antivirus software.
At C:\Users\kash1064\Downloads\evil.ps1:3 char:1
+ Invoke-Expression $testString
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : ParserError: (:) [Invoke-Expression], ParseException
    + FullyQualifiedErrorId : ScriptContainedMaliciousContent,Microsoft.PowerShell.Commands.InvokeExpressionCommand
```

図 1.7: PowerShell スクリプトの実行ブロック

また、このテスト用スクリプトがブロックされる際に取得した ETW トレースを `Get-AMSIEvent "$($env:USERPROFILE)\Downloads\AMSITrace.etl"` コマンドで解析してみると、以下のようにスクリプト全体が評価された際には検知が発生しなかった (`AMSI_RESULT_NOT_DETECTED`) ものの、その後に `$testString` として連結された文字列 `AMSI Test Sample: 7e72c3ce-861b-4339-8740-0ac1484c1386` が評価された際には、AMSI による検知が行われた (`AMSI_RESULT_DETECTED`) ことを確認できます。

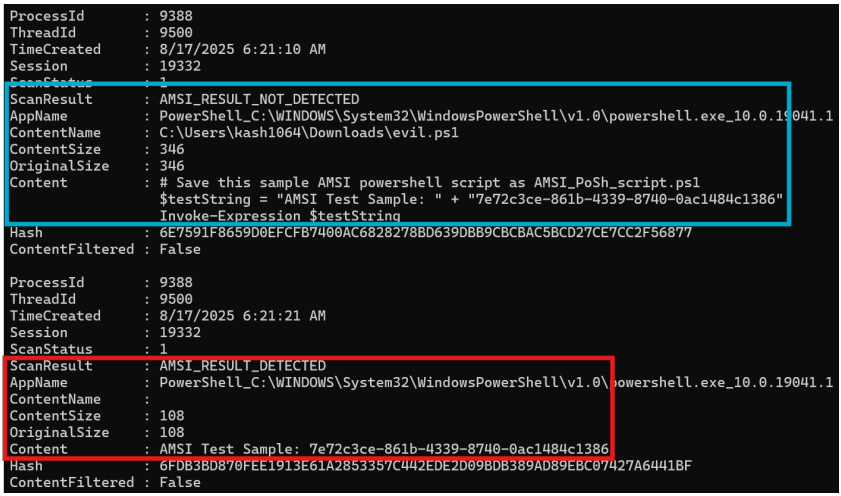

図 1.8: AMSI イベントの解析結果

また、テスト検知用の文字列 `AMSI Test Sample: 7e72c3ce-861b-4339-8740-0ac1484c1386` を以下のように難読化したスクリプトでもテストを実施してみます。[*22]

```
$base64 = "FHJ+YHoTZ1ZARxNgUl5DX1YJEwRWBAFQAFBWHgsFAlEeBwAA
Ch4LBAcDHgNSUAIHCwdQAgALBRQ="
$bytes = [Convert]::FromBase64String($base64)
$string = -join ($bytes | %{ [char] ($_ -bxor 0x33) })
iex $string
```

このスクリプトを実行した場合も、難読化されたデータの復号処理を行っ

[*22] AMSI demonstrations with Microsoft Defender for Endpoint https://learn.microsoft.com/ja-jp/defender-endpoint/mde-demonstration-amsi

た後、'AMSI Test Sample: 7e72c3ce-861b-4339-8740-0ac1484c1386' という文字列が評価されたタイミングで AMSI による検知が行われることを確認できます。

このように、PowerShell に統合された AMSI を使用することで、実行コードが高度に難読化されている場合でも復号処理後の実行コードをスキャンでき、脅威が検知されることがわかります。

AMSI で VBScript スクリプトの実行をブロックする

続いて、VBScript でもスクリプトの実行が AMSI によりブロックされることを確認してみます。

VBScript についても、公開ドキュメントで提供されている以下のテストスクリプトを使用します。

```
Dim result
result = eval("AMSI Test Sample:" + "7e72c3ce-861b-4339-874
0-0ac1484c1386")
WScript.Echo result
```

このスクリプトを evil.vbs として保存して実行すると、以下のようなエラーウィンドウが表示され、スクリプトの実行がブロックされたことを確認できます。

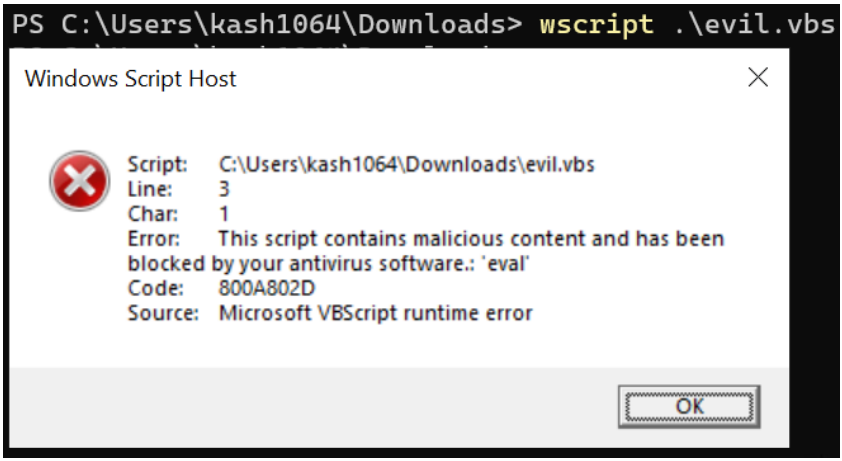

図 1.9: VBScript の実行ブロック

また、この時取得した ETW イベントトレースを解析してみると、PowerShell の場合と同じく検知用のテスト文字列が評価されたタイミングで AMSI による検知が発生したことを確認できます。

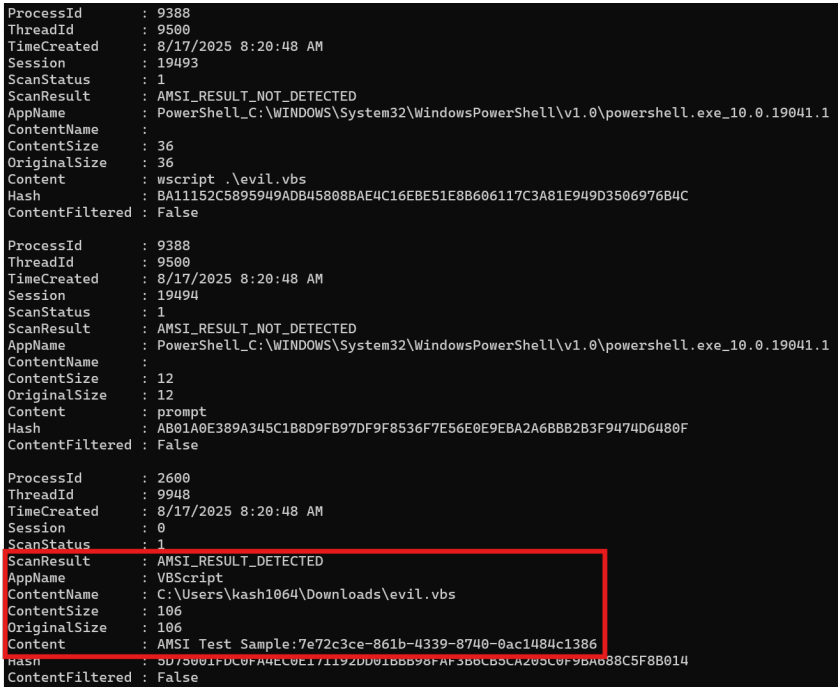

図 1.10: AMSI イベントの解析結果

1.5 1 章のまとめ

本章では、AMSI の概要やその保護機能を使用するメリット、また実際に AMSI による検知が行われる場合のイベントの確認方法などを紹介しました。

次章からは、サンプルコードを使用し、実際に AMSI がアプリケーションにどのように統合されているか、また AMSI によるスキャン要求をプロバイダーがどのように処理するかについて解説していきます。

第 2 章

AMSI クライアントインターフェース

本章では、公開されているサンプルコードを使用して、AMSI スキャン要求を行うクライアントアプリケーションの実装について解説します。

本章で使用するサンプルコードは、https://github.com/microsoft/Windows-classic-samples/releases/tag/MicrosoftDocs-Samples にて amsistream-sample.zip として配布されています。

このサンプルプログラム AmsiStream は、単一ファイル AmsiStream.cpp のみで実装されています。

AmsiStream.cpp は 300 行程度のプログラムですので、比較的容易に実装を把握することができます。

ダウンロードした ZIP ファイルを解凍し、AmsiStream.sln ファイルを Visual Studio で開くことでサンプルコードをビルドできるようになります。

2.1 AmsiStream の構成要素

今回使用するサンプルプログラム AmsiStream は、主に以下のクラスにより実装されています。

- CAmsiStreamBase クラス

- CStreamScanner クラス
- CAmsiFileStream クラス
- CAmsiMemoryStream クラス

以下に、各クラスの概要をまとめます。

CAmsiStreamBase クラス

CAmsiStreamBase クラスは、CAmsiFileStream クラスと CAmsiMemoryStream クラスに継承される基底クラスであり、以下のメソッドが定義されています。

- SetContentName メソッド
- BaseGetAttribute メソッド
- CopyAttribute メソッド

SetContentName メソッドでは、スキャンするファイルの名前などの情報をメモリに保存するメソッドです。

また、BaseGetAttribute メソッドと CopyAttribute メソッドは、後述する Attribute(属性) と呼ばれる AMSI スキャンにおいて重要な要素を処理する機能を定義しています。

これらのメソッドは、スキャン対象がファイルの場合に使用する CAmsiFileStream クラスと、スキャン対象がメモリ内のデータの場合に使用する CAmsiMemoryStream クラスの両方で使用されます。

CStreamScanner クラス

CStreamScanner クラスは、amsi.dll で実装されている AMSI インターフェースの COM オブジェクトを作成し、直接的に AMSI スキャン要求を行う機能を持つクラスです。

このクラスでは、以下の 2 つのメソッドが定義されています。

- Initialize メソッド
- ScanStream メソッド

これらのメソッドのうち、ScanStream メソッドが AMSI インターフェースの Scan メソッドを使用してコンテンツのスキャンを行い、その結果を表示する役割を担っています。

なお、今回使用するサンプルプログラムは PowerShell などのように何らかのコードを実行する機能は持たないため、AMSI スキャンを実施した結果は、コンソール上に表示されるだけの動作となります。

CAmsiFileStream クラス

CAmsiFileStream クラスは、AmsiStream プログラムにおいてファイルのスキャン要求を行う場合に使用するクラスです。

このクラスは、COM コンポーネントを実装するために使用できる Windows Runtime C++ Template Library(WRL) を利用して実装されています。

このクラスでは、以下の 3 つのメソッドが定義されています。

- RuntimeClassInitialize メソッド
- GetAttribute メソッド
- Read メソッド

これらのメソッドのうち、RuntimeClassInitialize メソッドは MakeAndInitialize 関数を使用して WRL クラスのオブジェクトを初期化するために使用する関数です。[*1]

CAmsiFileStream クラスは、エントリポイントの wmain 関数から直接呼び出される ScanArguments 関数内で MakeAndInitialize 関数により初

[*1] RuntimeClass Class https://learn.microsoft.com/ja-jp/cpp/cppcx/wrl/runtimeclass-class

期化されます。

他の 2 つのメソッドである GetAttribute メソッドと Read メソッドは、それぞれ IAmsiStream インターフェースのメソッドとして宣言されている処理を実装しています。

`IAmsiStream::GetAttribute` メソッドはストリームから要求された属性を返す役割を持ち、`IAmsiStream::Read` メソッドは AMSI プロバイダーからの要求に対して、指定されたバッファいっぱいのコンテンツを返す役割を持ちます。

CAmsiFileStream クラスの場合、スキャン対象がファイルになるので、Read メソッドはスキャン対象のファイルから読み出したデータを返すように実装されています。

CAmsiMemoryStream クラス

CAmsiFileStream クラスは、AmsiStream プログラムにおいてメモリコンテンツのスキャン要求を行う場合に使用するクラスです。

このクラスも、COM コンポーネントを実装するために使用できる Windows Runtime C++ Template Library(WRL) を利用して実装されており、CAmsiFileStream クラスと同じく以下の 3 つのメソッドが定義されています。

- RuntimeClassInitialize メソッド
- GetAttribute メソッド
- Read メソッド

これらのメソッドの用途はいずれも CAmsiFileStream クラスと同じですが、それぞれファイルではなくメモリコンテンツをスキャンする場合に使用できるように異なる処理が行われます。

例えば、Read メソッドはファイルから読み出したコンテンツではなく、

ハードコードされてサンプルプログラムのメモリ内にロードされているデータを返すように実装されています。

2.2 サンプルプログラムを実行する

サンプルプログラムをビルドする

サンプルプログラム AmsiStream の詳しい実装を確認する前に、まずはサンプルコードをビルドしてプログラムを実行してみることにします。

サンプルコードのビルドのため、ダウンロードした iamsistream-sample.zip を解凍した後、ソリューションファイル AmsiStream.sln を Visual Studio で開きます。(本書では Visual Studio 2022 を使用します。)

ソリューションファイルの起動時に以下のようなプロジェクトの再ターゲットに関するウインドウが表示された場合は、デフォルトの設定のまま [OK] をクリックして問題ありません。

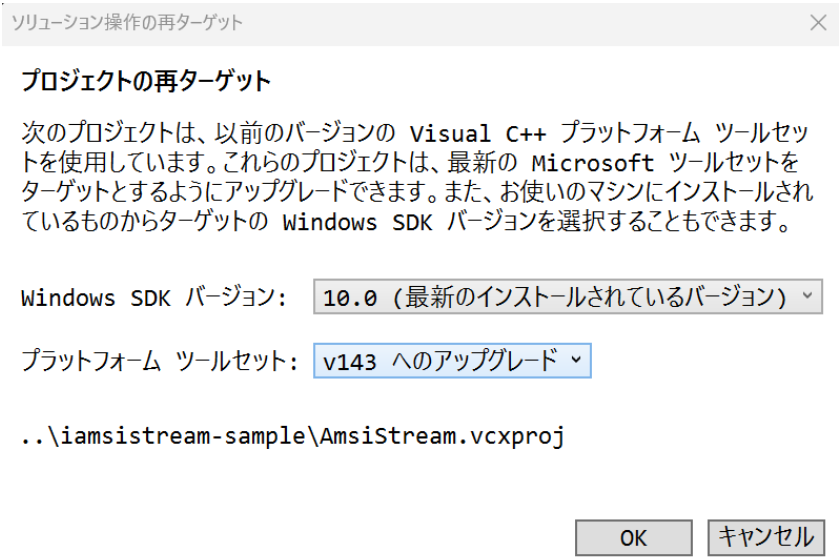

図 2.1: プロジェクトの再ターゲット

Visual Studio でソリューションファイルを開いたら、画面上部のメニューから [ソリューションのリビルド] をクリックしてサンプルコードをビルドできます。

後でデバッグを行うため、ビルド構成は [Debug] を選択したままにしておきます。

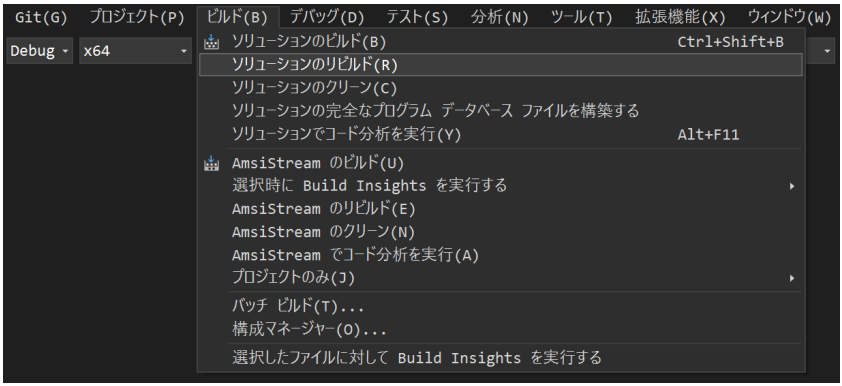

図 2.2: サンプルコードをビルドする

ビルドが正常に完了すると、`.\iamsistream-sample\x64\Debug` の直下に AmsiStream.exe と AmsiStream.pdb の 2 つのファイルが作成されたことを確認できます。

サンプルプログラムでメモリコンテンツをスキャンする

ビルドした AmsiStream.exe をコマンドライン引数無しで実行すると、以下のようにスキャンが行われたことを示す情報がコマンドライン上に出力されます。

```
PS C:\Users\kash1064\Downloads> .\AmsiStream.exe
Creating memory stream object
Calling antimalware->Scan() ...
GetAttribute() called with: attribute = 0, bufferSize = 1
GetAttribute() called with: attribute = 0, bufferSize = 68
GetAttribute() called with: attribute = 1, bufferSize = 1
GetAttribute() called with: attribute = 1, bufferSize = 38
GetAttribute() called with: attribute = 2, bufferSize = 8
GetAttribute() called with: attribute = 3, bufferSize = 8
GetAttribute() called with: attribute = 4, bufferSize = 8
GetAttribute() called with: attribute = 6, bufferSize = 8
GetAttribute() called with: attribute = 8, bufferSize = 8
GetAttribute() called with: attribute = 2, bufferSize = 8
GetAttribute() called with: attribute = 3, bufferSize = 8
GetAttribute() called with: attribute = 4, bufferSize = 8
GetAttribute() called with: attribute = 0, bufferSize = 1
GetAttribute() called with: attribute = 0, bufferSize = 68
GetAttribute() called with: attribute = 1, bufferSize = 1
GetAttribute() called with: attribute = 1, bufferSize = 38
Scan result is 1. IsMalware: 0
Provider display name: Microsoft Defender Antivirus
Leaving with hr = 0x0
```

図 2.3: AMSI でメモリスキャンを実行する

この時、サンプルプログラム AmsiStream はメモリ内に保存されている文字列 `Hello, world` を AMSI によりスキャンしています。

実際に AMSI により文字列 `Hello, world` がスキャンされているか否かについては、1 章に記載した手順で ETW イベントトレースを取得することで確認することができます。

ただし、1 章で使用した解析ツール Get-AMSIEvent を使用すると、`[Text.Encoding]::Unicode.GetString($_.Properties[7].Value)` によりスキャンコンテンツを UTF-16 LE でエンコードしてしまうので、文字列 `Hello, world` がスキャンされたことを確認することができません。[*2]

[*2] Encoding.Unicode Property https://learn.microsoft.com/ja-jp/dotnet/api/system.text.encoding.unicode

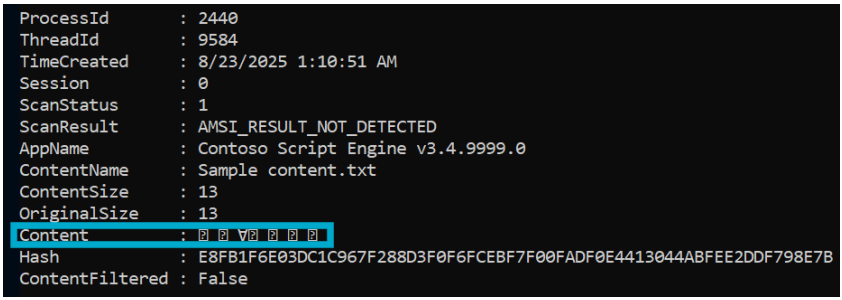

図 2.4: Get-AMSIEvent の出力結果

そこで、今回は標準の Get-WinEvent コマンドレットを使用して ETL ファイルからスキャン対象の AMSI コンテンツを確認してみます。

AMSI によりスキャンされたデータを確認するため、まずは保存された AMSITrace.etl を以下のコマンドで `$Events` として取得します。

```
$Events = Get-WinEvent -Path "$($env:USERPROFILE)\Downloads
\AMSITrace.etl" -Oldest
```

上記のコマンドで取得したイベントは EventRecord クラスのオブジェクトとして保存されました。[*3]

[*3] EventRecord Class https://learn.microsoft.com/ja-jp/dotnet/api/system.diagnostics.eventing.reader.eventrecord

```
PS C:\> $Events[1].GetType()

IsPublic IsSerial Name                                     BaseType
-------- -------- ----                                     --------
True     False    EventLogRecord                           System.Diagnostics.Eventing.Reader.EventRecord


PS C:\> $Events[1].ProviderName
Microsoft-Antimalware-Scan-Interface
```

図 2.5: 保存されたイベントの確認

続いて、このイベントの中から AMSI でスキャンされたコンテンツを抽出します。

1 章で使用した解析ツール Get-AMSIEvent では、Get-WinEvent コマンドレットで取得したイベントの `Properties[4].Value` を ContentName、および、`Properties[7].Value` を AMSIContent として取り出しています。

そこで、取得したイベントに対して以下のコマンドをそれぞれ実行することで、ContentName が `Sample content.txt` であり、かつスキャンされたデータが確かに `Hello, world` であることを確認できました。

```
# ContentName
$Events[2].Properties[4].Value

# AMSIContent
[Text.Encoding]::UTF8.GetString($Events[2].Properties[7].Va
lue)
```

```
PS C:\> $Events[2].Properties[4].Value
Sample content.txt
PS C:\> [Text.Encoding]::UTF8.GetString($Events[2].Properties[7].Value)
Hello, world
```

図 2.6: AMSI でスキャンされたコンテンツ

サンプルプログラムでファイルをスキャンする

続いて、サンプルプログラム AmsiStream によるファイルのスキャンを試します。

サンプルプログラム AmsiStream では、実行時のコマンドライン引数にファイルパスを指定した場合、メモリ内のデータではなくファイルに保存されているデータを AMSI でスキャンできます。

以下は、AMSI 検知テスト用の文字列 `AMSI Test Sample:7e72c3ce-861b-4339-8740-0ac1484c1386` が保存された sample1.txt と、無害なテキストが保存された sample2.txt を同時に AmsiStream でスキャンした場合の結果です。

この実行結果から、テスト検知用の文字列を含むファイルである sample1.txt のみが、AMSI スキャンによりマルウェアと判定されたことがわかります。

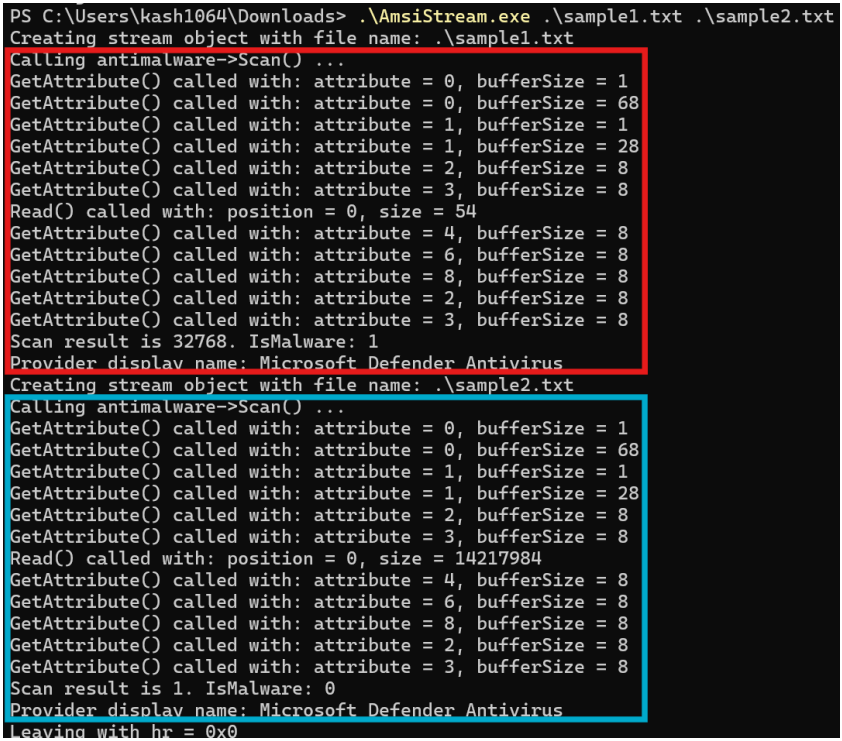

図 2.7: AMSI でファイルのスキャンを実行する

2.3 AMSI クライアントインターフェースの実装

サンプルプログラム AmsiStream の動作を確認できたので、詳しい実装を解説していきます。

以下のステップは、サンプルプログラム AmsiStream の動作を簡略化したものです。

1. エントリポイントである wmain 関数が呼び出され、CoInitializeEx

関数による初期化を行う
2. 初期化に成功した場合、コマンドライン引数とともに ScanArguments 関数を呼び出す
3. ScanArguments 関数にて、初めに CStreamScanner クラスの初期化を行う
4. ファイルパスを指定するためのコマンドライン引数が存在している場合には CAmsiFileStream クラス、存在していない場合には CAmsiMemoryStream クラスのオブジェクトを初期化する
5. ScanStream メソッドを使用してファイルもしくはメモリコンテンツのスキャンを行う

本章では、上記の実行ステップに沿ってプログラムの実行コードを追っていくことにします。

エントリポイントの実装

エントリポイントとなる wmain 関数は、以下のように小さな関数として実装されています。

```
int __cdecl wmain(_In_ int argc, _In_reads_(argc) WCHAR **a
rgv)
{
  HRESULT hr = \
    CoInitializeEx(nullptr, COINIT_MULTITHREADED);

  if (SUCCEEDED(hr)) {
    hr = ScanArguments(argc, argv);
    CoUninitialize();
  }
  wprintf(L"Leaving with hr = 0x%x\n", hr);
  return 0;
}
```

この中ではまず、CoInitializeEx 関数を呼び出して COM ライブラリの初期化を行います。[*4]

その後、各種クラスの初期化やスキャン要求を行う ScanArguments 関数がコマンドライン引数とともに呼び出され、スキャンの完了後に CoUninitialize 関数によりリソースが解放されます。

ScanArguments 関数の実装

ScanArguments 関数は、以下のように wmain 関数からコマンドライン引数を受け取って呼び出されます。

```
HRESULT ScanArguments(_In_ int argc, _In_reads_(argc) wchar
_t** argv)
```

この ScanArguments 関数内ではまず、CStreamScanner クラスのオブジェクトである scanner の初期化を行います。

CStreamScanner クラスは、前述した通り AMSI インターフェースの Scan メソッドを使用してコンテンツのスキャンを行い、その結果を表示する役割を担う ScanStream メソッドを定義しているクラスです。

```
CStreamScanner scanner;
HRESULT hr = scanner.Initialize();
if (FAILED(hr))
{
  return hr;
}
```

[*4] CoInitializeEx function https://learn.microsoft.com/ja-jp/windows/win32/api/combaseapi/nf-combaseapi-coinitializeex

scanner の初期化の完了後、ScanArguments 関数ではコマンドライン引数にファイルパスが含まれるか否かによって、CAmsiFileStream クラス、もしくは CAmsiMemoryStream クラスの初期化を行います。

コマンドライン引数無しでプログラムを実行した場合、サンプルプログラム AmsiStream はメモリコンテンツをスキャンするために CAmsiMemoryStream クラスの初期化を行います。

```
// Scan a single memory stream.
wprintf(L"Creating memory stream object\n");

ComPtr<IAmsiStream> stream;
hr = MakeAndInitialize<CAmsiMemoryStream>(&stream);
if (FAILED(hr)) {
   return hr;
}
```

上記のコードでは、Windows ランタイムクラスを初期化するための MakeAndInitialize 関数を使用して、CAmsiMemoryStream クラスのオブジェクト stream を初期化しています。[*5]

そして、CAmsiMemoryStream クラスのオブジェクトの初期化が完了した後、`scanner.ScanStream(stream.Get())` によりスキャン要求を実施しています。

```
hr = scanner.ScanStream(stream.Get());
if (FAILED(hr))
{
   return hr;
}
```

[*5] MakeAndInitialize Function https://learn.microsoft.com/ja-jp/cpp/cppcx/wrl/makeandinitialize-function

CAmsiMemoryStream クラスは `MakeAndInitialize<CAmsiMemoryStream>(&stream)` により Windows ランタイムクラス `ComPtr<IAmsiStream>` として初期化されているため、`stream.Get()` により `Microsoft::WRL::ComPtr<IAmsiStream>::Get` が呼び出され、この ComPtr と紐づくインターフェース (IAmsiStream インターフェース) へのポインタが返されます。[*6]

`scanner.ScanStream()` により実行されるスキャン動作については、`CStreamScanner::ScanStream` メソッドの項で後述します。

続いて、コードを少し戻し、サンプルプログラムがコマンドライン引数により指定されたファイルパスと共に実行された場合の動作を確認します。

サンプルプログラムがファイルパスを引数として実行された場合、ScanArguments 関数は CAmsiMemoryStream クラスではなく CAmsiFileStream クラスの初期化を行います。

CAmsiFileStream クラスの初期化を行う際には、コマンドライン引数から取得したファイルパスが引数として渡されます。

また、CAmsiMemoryStream クラスを使用する場合と同様、`stream.Get()` で取得したインターフェースへのポインタを使用して `scanner.ScanStream()` を呼び出しスキャンを実行しています。

```
// Scan the files passed on the command line.
for (int i = 1; i < argc; i++)
{
  LPWSTR fileName = argv[i];

  wprintf(L"Creating stream object with file name: %s\n",
          fileName
        );
```

[*6] ComPtr Class https://learn.microsoft.com/ja-jp/cpp/cppcx/wrl/comptr-class

```
  ComPtr<IAmsiStream> stream;
  hr = MakeAndInitialize<CAmsiFileStream>(&stream,
                                          fileName
                                         );
  if (FAILED(hr)) {
    return hr;
  }

  hr = scanner.ScanStream(stream.Get());
  if (FAILED(hr))
  {
    return hr;
  }
}
```

CStreamScanner クラスの実装

CStreamScanner クラスは、サンプルプログラムにてメモリコンテンツの AMSI スキャン要求を行うために使用するクラスです。

このクラスでは、以下の 2 つのメソッドが定義されています。

- Initialize メソッド
- ScanStream メソッド

Initialize メソッドは、ScanArguments 関数内で `scanner.Initialize()` として呼び出される初期化用のメソッドであり、以下の通り定義されています。

```
HRESULT Initialize()
{
  return CoCreateInstance(
    __uuidof(CAntimalware),
    nullptr,
```

```
        CLSCTX_INPROC_SERVER,
        IID_PPV_ARGS(&m_antimalware));
}
```

この Initialize メソッドの中で呼び出されている CoCreateInstance 関数は、指定した CLSID と紐づくクラスのオブジェクトを作成して初期化します。[*7]

この中で引数に使用されている CAntimalware は、ヘッダファイル amsi.h で以下の通り定義されている CLSID です。

そのため、この CLSID と紐づくクラスのオブジェクトを作成して初期化することで、サンプルプログラムからの AMSI スキャン要求を行うことができるようになります。

ちなみに、この CLSID と対応する COM クラスは `%windir%\system32\amsi.dll` に含まれています。

```
class DECLSPEC_UUID("fdb00e52-a214-4aa1-8fba-4357bb0072ec")
 CAntimalware;
```

また、CoCreateInstance 関数で作成されたインターフェースのポインタは `ComPtr<IAntimalware> m_antimalware;` として定義されているプライベートメンバである m_antimalware に保存されます。

なお、`__uuidof` オペレーターや `IID_PPV_ARGS` マクロについては本書のスコープからは外れるので詳しくは扱いませんが、いずれもより効果的に COM を使用するプログラムを実装するために使用しています。[*8]

[*7] CoCreateInstance function https://learn.microsoft.com/ja-jp/windows/win32/api/combaseapi/nf-combaseapi-cocreateinstance

[*8] COM Coding Practices https://learn.microsoft.com/ja-jp/windows/win32/learnwin32/com-coding-practices

CStreamScanner クラスで定義されているもう 1 つのメソッドである ScanStream メソッドは、IAmsiStream インターフェイスのオブジェクトを引数 stream として受け取り、IAntimalware インターフェースの Scan メソッドを呼び出して AMSI スキャン要求を行います。

このメソッドは以下の通り実装されています。

```
HRESULT ScanStream(_In_ IAmsiStream* stream)
{
  wprintf(L"Calling antimalware->Scan() ...\n");

  ComPtr<IAntimalwareProvider> provider;
  AMSI_RESULT r;
  HRESULT hr = m_antimalware->Scan(stream, &r, &provider);
  if (FAILED(hr)) {
    return hr;
  }

  wprintf(L"Scan result is %u. IsMalware: %d\n",
           r,
           AmsiResultIsMalware(r)
         );

  if (provider) {
      PWSTR name;
      hr = provider->DisplayName(&name);
      if (SUCCEEDED(hr)) {
        wprintf(L"Provider display name: %s\n",
                 name
               );
        CoTaskMemFree(name);
      }
      else
      {
        wprintf(L"DisplayName failed with 0x%x", hr);
      }
  }
```

```
  return S_OK;
}
```

以下のコードでは、Initialize メソッドにより初期化した IAntimalware インターフェースのポインタが保存されている m_antimalware を使用し、引数として受け取った stream と共に Scan メソッドを呼び出しています。

```
ComPtr<IAntimalwareProvider> provider;
AMSI_RESULT r;
HRESULT hr = m_antimalware->Scan(stream, &r, &provider);
if (FAILED(hr)) {
  return hr;
}
```

この時呼び出される IAntimalware インターフェースの Scan メソッドは以下の通り 3 つの引数を取ります。[*9]

```
HRESULT Scan(
  [in]  IAmsiStream          *stream,
  [out] AMSI_RESULT          *result,
  [out] IAntimalwareProvider **provider
);
```

第 1 引数には、IAmsiStream インターフェイスのオブジェクトである stream を使用します。

また、第 2 引数には、スキャン結果 (AMSI_RESULT) を受け取る出力先のアドレスを使用します。

[*9] IAntimalware::Scan method https://learn.microsoft.com/ja-jp/windows/win32/api/amsi/nf-amsi-iantimalware-scan

そして、第 3 引数では、アンチマルウェア製品のプロバイダー情報を示す IAntimalwareProvider インターフェースのオブジェクトを受け取ります。

AMSI スキャンが完了すると、ScanStream メソッドは受け取ったスキャン結果をコンソールに出力します。

ここで使用している AmsiResultIsMalware マクロは、受け取ったスキャン結果 (AMSI_RESULT) から、対象のコンテンツをブロックすべきかどうかを判断します。[*10]

```
wprintf(L"Scan result is %u. IsMalware: %d\n",
         r,
         AmsiResultIsMalware(r)
       );
```

このマクロは、公開されているヘッダファイル amsi.h の中で AMSI_RESULT と共に以下のように定義されており、スキャン結果の値が AMSI_RESULT_DETECTED = 32768 以上の値の場合に True を返すことがわかります。

```
enum AMSI_RESULT
{
    AMSI_RESULT_CLEAN   = 0,
    AMSI_RESULT_NOT_DETECTED    = 1,
    AMSI_RESULT_BLOCKED_BY_ADMIN_START  = 0x4000,
    AMSI_RESULT_BLOCKED_BY_ADMIN_END    = 0x4fff,
    AMSI_RESULT_DETECTED    = 32768
}   AMSI_RESULT;

#define AmsiResultIsMalware(r) ((r) >= AMSI_RESULT_DETECTED
)
```

[*10] AmsiResultIsMalware macro https://learn.microsoft.com/en-us/windows/win32/api/amsi/nf-amsi-amsiresultismalware

つまり、AMSI プロバイダーとして登録されるアンチマルウェア製品は、スキャンしたコンテンツをマルウェアと判定する場合にはスキャン結果を AMSI_RESULT_DETECTED = 32768 以上の値として返す必要があります。

登録されている AMSI プロバイダーがスキャン結果を返却する仕組みについては 3 章で後述します。

CAmsiStreamBase クラスの実装

CStreamScanner クラスの ScanStream メソッドは、CAmsiFileStream クラスと CAmsiMemoryStream クラスのメソッドからメモリコンテンツやファイルデータのスキャンを行うために呼び出されます。

本項では、これらのクラスの実装を解説する前に、2 つのクラスに継承される基底クラスである CAmsiStreamBase クラスの実装を解説します。

前述の通り、CAmsiStreamBase クラスには以下の 3 つのメソッドが定義されています。

- SetContentName メソッド
- BaseGetAttribute メソッド
- CopyAttribute メソッド

SetContentName メソッドは、以下の通り定義されており、スキャンするファイルの名前などの情報をメモリに保存します。

```
HRESULT SetContentName(_In_ PCWSTR name)
{
    m_contentName = _wcsdup(name);
    return m_contentName ? S_OK : E_OUTOFMEMORY;
}
```

例えば、CAmsiFileStream クラスのメソッドから呼び出される場合には、

SetContentName(fileName) のようにファイル名を含む文字列が引数として与えられます。

また、CAmsiMemoryStream クラスのメソッドから呼び出される場合には、SetContentName(L"Sample content.txt") と、ハードコードされている文字列が引数として与えられます。

ここで保存される m_contentName は、あとで AMSI プロバイダーから属性情報 AMSI_ATTRIBUTE_CONTENT_NAME を要求された際にクライアントアプリケーションが返却する値としてに使用されます。

次に、CAmsiStreamBase クラスで実装されている BaseGetAttribute メソッドを解説します。

このサンプルプログラムにおいて、BaseGetAttribute メソッドは要求された AMSI 属性情報を返すために使用されるメソッドである IAmsiStream インターフェースの GetAttribute メソッドから呼び出される形で利用されます。[*11]

この BaseGetAttribute メソッドでは、引数として要求された AMSI 属性の情報を受け取り、その値に応じて異なる属性情報を出力先のバッファに書き込みます。

```
HRESULT BaseGetAttribute(
  _In_ AMSI_ATTRIBUTE attribute,
  _In_ ULONG bufferSize,
  _Out_writes_bytes_to_(bufferSize, *actualSize) PBYTE buff
er,
  _Out_ ULONG* actualSize)
  //
  // Return Values:
```

[*11] IAmsiStream::GetAttribute method https://learn.microsoft.com/ja-jp/windows/win32/api/amsi/nf-amsi-iamsistream-getattribute

```
    //   S_OK: SUCCESS
    //   E_NOTIMPL: attribute not supported
    //   E_NOT_SUFFICIENT_BUFFER: need a larger buffer, requi
  red size in *retSize
    //   E_INVALIDARG: bad arguments
    //   E_NOT_VALID_STATE: object not initialized
    //
```

この時に要求される、AMSI スキャン時に使用する属性情報には以下の種類があります。

属性	データ
AMSI_ATTRIBUTE_APP_NAME	LPWSTR からコピーされた呼び出し元アプリケーションの名前/バージョン、または GUID 文字列
AMSI_ATTRIBUTE_CONTENT_NAME	LPWSTR からコピーされたコンテンツのファイル名、URL、一意のスクリプト ID、またはそれに類するもの
AMSI_ATTRIBUTE_CONTENT_SIZE	ULONGLONG で表現された入力のサイズ
AMSI_ATTRIBUTE_CONTENT_ADDRESS	コンテンツが完全にメモリにロードされている場合のメモリアドレス
AMSI_ATTRIBUTE_SESSION	セッションは、スキャン対象のコンテンツが同一の元スクリプトに属している場合など、異なるスキャン呼び出しを関連付けるために使用されるものであり、コンテンツが自己完結している場合は nullptr を使用する

図 2.8: AMSI スキャン時に使用する属性情報

また、このサンプルプログラムの中では、要求された属性に対してそれぞれ以下の値を返します。

要求された属性	サンプルプログラムが返す値
AMSI_ATTRIBUTE_APP_NAME	グローバル変数として定義されたアプリ名である Contoso Script Engine v3.4.9999.0
AMSI_ATTRIBUTE_CONTENT_NAME	SetContentName メソッドで保存された m_contentName の値
AMSI_ATTRIBUTE_CONTENT_SIZE	スキャン対象のメモリコンテンツのデータサイズもしくは GetFileSizeEx 関数で取得したファイルサイズ
AMSI_ATTRIBUTE_CONTENT_ADDRESS	(メモリスキャンの場合のみ)コンテンツがロードされている場合のメモリアドレス
AMSI_ATTRIBUTE_SESSION	サンプルプログラムの場合、常に nullptr を返す

図 2.9: サンプルプログラムが返す属性情報

以下が、BaseGetAttribute メソッドにて各属性情報をバッファに書き込むためのコード部分です。

AMSI_ATTRIBUTE_CONTENT_ADDRESS は、メモリコンテンツのスキャン時にのみ使用されるため、BaseGetAttribute メソッドではなく CAmsiMemoryStream クラスの GetAttribute メソッド内で定義されています。

```
*actualSize = 0;

switch (attribute)
{
  case AMSI_ATTRIBUTE_CONTENT_SIZE:
    return CopyAttribute(&m_contentSize,
                         sizeof(m_contentSize),
                         bufferSize,
                         buffer,
                         actualSize
                        );

  case AMSI_ATTRIBUTE_CONTENT_NAME:
    return CopyAttribute(m_contentName,
                         (wcslen(m_contentName) + 1) \
                                     * sizeof(WCHAR),
```

```
                          bufferSize, buffer,
                          actualSize
                         );

    case AMSI_ATTRIBUTE_APP_NAME:
      return CopyAttribute(AppName,
                           sizeof(AppName),
                           bufferSize,
                           buffer,
                           actualSize
                          );

    case AMSI_ATTRIBUTE_SESSION:
      // no session for file stream
      constexpr HAMSISESSION session = nullptr;
      return CopyAttribute(&session,
                           sizeof(session),
                           bufferSize,
                           buffer,
                           actualSize
                          );
  }

  return E_NOTIMPL; // unsupport attribute
```

上記のコード部分を見るとわかる通り、実際に要求された AMSI 属性情報と対応する値をバッファに書き込む操作は、同じく CAmsiStreamBase クラスのメンバである CopyAttribute メソッドで行われます。

このメソッドの実装は非常にシンプルで、引数として受け取ったデータから、バッファのサイズのチェックと memcpy_s 関数によるデータの書き込み操作を行います。[*12]

[*12] memcpy_s https://learn.microsoft.com/ja-jp/cpp/c-runtime-library/reference/memcpy-s-wmemcpy-s

```
HRESULT CopyAttribute(
  _In_ const void* resultData,
  _In_ size_t resultSize,
  _In_ ULONG bufferSize,
  _Out_writes_bytes_to_(bufferSize, *actualSize) \
                                        PBYTE buffer,
  _Out_ ULONG* actualSize)
{
  *actualSize = (ULONG)resultSize;
  if (bufferSize < resultSize)
  {
    return E_NOT_SUFFICIENT_BUFFER;
  }
  memcpy_s(buffer, bufferSize, resultData, resultSize);
  return S_OK;
}
```

CAmsiMemoryStream クラスの実装

CAmsiMemoryStream クラスは、メモリコンテンツのスキャンを行う場合に使用されるクラスです。

このクラスは AMSI の仕組みにおいて属性情報やスキャンコンテンツの取得に使用される IAmsiStream インターフェースの実装として利用されます。[*13]

また、前項で解説した基底クラス CAmsiStreamBase を継承しています。

```
class CAmsiMemoryStream : public RuntimeClass<RuntimeClassF
lags<ClassicCom>, IAmsiStream>, CAmsiStreamBase
```

[*13] IAmsiStream interface https://learn.microsoft.com/ja-jp/windows/win32/api/amsi/nn-amsi-iamsistream

この CAmsiMemoryStream クラスは IAmsiStream インターフェースの実際の実装であるため、GetAttribute メソッドと Read メソッドの 2 つのメソッドが定義されています。

IAmsiStream インターフェースの GetAttribute メソッドは、スキャン時に要求された AMSI 属性情報と対応する値を返す必要のあるメソッドとして定義されています。[*14]

GetAttribute メソッドが要求する各引数の用途は以下の通りです。

パラメータ	用途
[in] attribute	要求された AMSI 属性の種類
[in] dataSize	出力バッファのサイズ
[out] data	要求された属性を受け取るバッファ
[out] retData	バッファに返されるバイト数(GetAttribute メソッドが E_NOT_SUFFICIENT_BUFFER を返す場合、retData には要求に必要なバイト数が書き込まれる)

図 2.10: GetAttribute メソッドのパラメータ情報

CAmsiMemoryStream クラスにおいてこのメソッドは以下の通り定義されており、AMSI_ATTRIBUTE_CONTENT_ADDRESS 以外の属性に関する要求を受け取った場合には、前項で確認した基底クラスの BaseGetAttribute メソッドに処理を任せるように実装されています。

また、要求された属性が AMSI_ATTRIBUTE_CONTENT_ADDRESS の場合には、グローバル定数 SampleStream として定義されているスキャン対象の文字列のアドレスを返します。

[*14] IAmsiStream::GetAttribute method https://learn.microsoft.com/ja-jp/windows/win32/api/amsi/nf-amsi-iamsistream-getattribute

```
STDMETHOD(GetAttribute)(
  _In_ AMSI_ATTRIBUTE attribute,
  _In_ ULONG bufferSize,
  _Out_writes_bytes_to_(bufferSize, *actualSize) \
                                        PBYTE buffer,
  _Out_ ULONG* actualSize)
{
  HRESULT hr = BaseGetAttribute(attribute,
                                bufferSize,
                                buffer,
                                actualSize
                               );
  if (hr == E_NOTIMPL)
  {
    switch (attribute)
    {
    case AMSI_ATTRIBUTE_CONTENT_ADDRESS:
      const void* contentAddress = SampleStream;
      hr = CopyAttribute(&contentAddress,
                         sizeof(contentAddress),
                         bufferSize,
                         buffer,
                         actualSize
                        );
    }
  }
  return hr;
}
```

次に、Read メソッドの実装を確認します。

IAmsiStream インターフェースの Read メソッドは、受け取ったバッファがいっぱいになるまでスキャン対象のコンテンツを返す必要のあるメソッドとして定義されています。[*15]

このメソッドが受け取る引数の用途は以下の通りです。

[*15] IAmsiStream::Read method https://learn.microsoft.com/ja-jp/windows/win32/api/amsi/nf-amsi-iamsistream-read

パラメータ	用途
[in] position	読み取り対象のコンテンツの 0 から始まるインデックス
[in] size	要求された読み取りサイズ
[out] buffer	読み取りコンテンツを返すバッファ
[out] readSize	バッファに読み込まれたバイト数

図 2.11: Read メソッドのパラメータ情報

以下は、CAmsiMemoryStream クラスにおける Read メソッドの実際の実装です。

```
STDMETHOD(Read)(
    _In_ ULONGLONG position,
    _In_ ULONG size,
    _Out_writes_bytes_to_(size, *readSize) PBYTE buffer,
    _Out_ ULONG* readSize)
{
  wprintf(L"Read() called with: position = %I64u, size = %u
\n",
          position,
          size
        );

  *readSize = 0;
  if (position >= m_contentSize)
  {
    wprintf(L"Reading beyond end of stream\n");
    return HRESULT_FROM_WIN32(ERROR_HANDLE_EOF);
  }

  if (size > m_contentSize - position) {
```

```
    size = static_cast<ULONG>(m_contentSize - position);
  }

  *readSize = size;
  memcpy_s(buffer, size, SampleStream + position, size);
  return S_OK;
}
```

上記のコードの通り、CAmsiMemoryStream クラスを使用する場合にはグローバル定数 SampleStream として定義されているスキャン対象の文字列を memcpy_s 関数でバッファに書き込む操作を行うことで、IAmsiStream インターフェースの Read メソッドの要件を満たしています。

CAmsiFileStream クラスの実装

最後に、AMSI によりファイルコンテンツのスキャンを行う CAmsiFileStream クラスの実装を解説します。

```
class CAmsiFileStream : public RuntimeClass<RuntimeClassFla
gs<ClassicCom>, IAmsiStream>, CAmsiStreamBase
```

CAmsiFileStream クラスは CAmsiMemoryStream と同じく、AMSI の仕組みにおいて属性情報やスキャンコンテンツの取得に使用される IAmsiStream インターフェースの実装として利用されます。

このクラスの初期化時にはまず、スキャン対象のファイルパスを文字列として保存した fileName が基底クラスの SetContentName メソッドにより m_contentName としてコピーされます。

```
HRESULT hr = S_OK;
hr = SetContentName(fileName);
if (FAILED(hr))
{
  return hr;
}
```

続いて、CreateFileW 関数によりスキャン対象のファイルに対する読み取りアクセスようのファイルハンドルを作成し、それをプライベートメンバ m_fileHandle にアタッチしています。

この m_fileHandle とは、WRL で定義されているファイルハンドルを扱うための FileHandleTraits のオブジェクトであり、ファイルハンドルを安全に管理するために利用されます。

```
m_fileHandle.Attach(
  CreateFileW(
    fileName,
    GENERIC_READ,          // dwDesiredAccess
    0,                     // dwShareMode
    nullptr,               // lpSecurityAttributes
    OPEN_EXISTING,         // dwCreationDisposition
    FILE_ATTRIBUTE_NORMAL,// dwFlagsAndAttributes
    nullptr
  )
);                         // hTemplateFile
```

スキャン対象のファイルのファイルハンドルを取得したら、GetFileSizeEx 関数により対象ファイルのサイズを取得し、ULONGLONG 型の変数 m_contentSize として保存します。

これは、前述の通り AMSI 属性 `AMSI_ATTRIBUTE_CONTENT_SIZE` として返される値は ULONGLONG 型で表現されると定義されているためです。

```
LARGE_INTEGER fileSize;
if (!GetFileSizeEx(m_fileHandle.Get(), &fileSize))
{
  hr = HRESULT_FROM_WIN32(GetLastError());
  wprintf(L"GetFileSizeEx failed with 0x%x\n", hr);
  return hr;
}

m_contentSize = (ULONGLONG)fileSize.QuadPart;
```

次に、CAmsiFileStream クラスの GetAttribute メソッドを確認します。

CAmsiFileStream クラスでは CAmsiMemoryStream クラスとは異なり、基底クラスの BaseGetAttribute メソッドに完全に依存する形で GetAttribute メソッドが定義されています。

```
STDMETHOD(GetAttribute)(
      _In_ AMSI_ATTRIBUTE attribute,
_In_ ULONG bufferSize,
_Out_writes_bytes_to_(bufferSize, *actualSize) PBYTE buffer
,
_Out_ ULONG* actualSize)
{
  return BaseGetAttribute(attribute, bufferSize, buffer, ac
tualSize);
}
```

最後に、CAmsiFileStream クラスの Read メソッドを確認します。

CAmsiMemoryStream クラスと同じく、CAmsiFileStream クラスの場合も Read メソッドは IAmsiStream インターフェースの実際の実装であり、受け取ったバッファがいっぱいになるまでスキャン対象のコンテンツを返す必要のあるメソッドとして定義されています。

```
OVERLAPPED o = {};
o.Offset = LODWORD(position);
o.OffsetHigh = HIDWORD(position);

if (!ReadFile(m_fileHandle.Get(),
              buffer,
              size,
              readSize,
              &o
             )
    )
{
  HRESULT hr = HRESULT_FROM_WIN32(GetLastError());
  wprintf(L"ReadFile failed with 0x%x\n", hr);
  return hr;
}
```

ただし、上記のコード部分の通り、CAmsiMemoryStream クラスの場合とは違い、スキャン対象のファイルから ReadFile 関数で読み出したデータを出力先のバッファに書き込む点に大きな違いがあることがわかります。

2.4 2 章のまとめ

本章では、サンプルプログラム AmsiStream が AMSI を使用してメモリコンテンツとファイルコンテンツのスキャン要求を行う仕組みを解説しました。

サンプルプログラムの実装から、クライアントアプリケーションに AMSI を統合してコンテンツスキャンを行うためには、端的に以下の操作を行うインターフェースを実装すればよいことがわかりました。

- IAmsiStream インターフェースのオブジェクトを定義し、各 AMSI 属性情報やスキャンするコンテンツを返すための GetAttribute メソッドと Read メソッドを実装する
- amsi.dll で定義されている COM クラスのオブジェクト (IAntimal-

ware インターフェース) を作成し、Scan メソッドを呼び出して AMSI スキャンを要求する
- Scan メソッドの結果 (`AMSI_RESULT`) を元に、コンテンツのブロックなどの処理を行う

次の 3 章では、IAntimalware インターフェースの Scan メソッドの呼び出し後に、システムに登録されている AMSI プロバイダーがどのようにコンテンツの取得やスキャンを行うのかを解説します。

第 3 章

AMSI プロバイダー

本章では、アプリケーションから要求されたスキャンを行うための AMSI プロバイダーについて解説します。

本章で使用するサンプルコードは、https://github.com/microsoft/Windows-classic-samples/releases/tag/MicrosoftDocs-Samples にて iantimalwareprovider-sample.zip として配布されています。

AMSI プロバイダーのサンプルコードも、単一ファイル AmsiProvider.cpp により実装されており、コード量も 300 行未満と、容易に実装を把握できるようになっています。

また、2 章で扱ったサンプルコードと同じく、ダウンロードしたソリューションファイルを Visual Studio で開くことで、サンプルコードをビルドできるようになります。

3.1 AmsiProvider の構成要素

本章で扱うサンプルプロバイダー AmsiProvider は、基本的には IAntimalwareProvider インターフェースを継承する SampleAmsiProvider クラスにより実装されており、IAntimalwareProvider インターフェースに含まれる以下の 3 つのメソッドの実際の動作を定義しています。[*1]

[*1] IAntimalwareProvider interface https://learn.microsoft.com/en-us/windows/win32/api/amsi/nn-amsi-iantimalwareprovider

- Scan メソッド
- DisplayName メソッド
- CloseSession メソッド

また、SampleAmsiProvider クラス以外にも、ビルドした DLL を AMSI プロバイダーとして登録するために必要ないくつかの操作や、イベントを ETW トレースセッションとして出力するために使用する関数などが実装されています。

3.2 サンプルプロバイダーを実行する

今回使用するサンプルプロバイダー SampleAmsiProvider は、システムに AMSI プロバイダーとして登録された後、受け取ったスキャン要求に対して以下の操作を行います。

- スキャンの開始後、取得した AMSI 属性情報を ETW トレースセッションに出力する
- スキャン対象として受け取ったデータのすべてのバイトを XOR し、その結果を ETW トレースセッションに出力する
- すべてのスキャン要求に対して `AMSI_RESULT_NOT_DETECTED` を返却する

本項では、実際にサンプルプロバイダーをビルドし、仮想マシン内で実行することで上記の動作となることを確認します。

サンプルプロバイダーを登録する

まずは、2 章と同じ手順でビルドしたファイルを仮想マシンに配置した後、管理者権限で起動したコマンドプロンプトで以下のコマンドを実行することで AMSI プロバイダーとして登録します。

```
regsvr32 AmsiProvider.dll
```

このコマンドにより、HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\AMSI\Providers の直下に新しく SampleAmsiProvider の ID である {2E5D8A62-77F9-4F7B-A90C-2744820139B2} が登録されました。

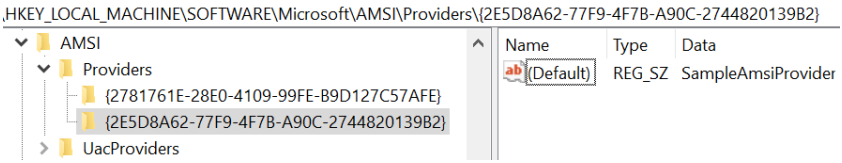

図 3.1: SampleAmsiProvider の登録

さらに、HKEY_LOCAL_MACHINE\SOFTWARE\Classes\CLSID\{2E5D8A62-77F9-4F7B-A90C-2744820139B2} には、COM サーバーとして登録されている SampleAmsiProvider の CLSID と DLL へのパスが書き込まれたことを確認できます。

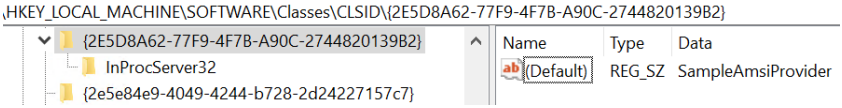

図 3.2: SampleAmsiProvider の登録

サンプルプロバイダーで AMSI スキャン要求を処理する

サンプルプロバイダー SampleAmsiProvider をシステムに登録できたので、登録した SampleAmsiProvider の動作を確認します。

このサンプルプロバイダーでは、GUID {00604c86-2d25-46d6-b814-cd149bfdf0b3} にて ETW トレースログの出力を行っています。

そのため、この GUID と対応するイベントを取得することで、SampleAmsiProvider の動作を確認できます。

1 章で紹介した通り、ETW トレースログは logman や Xperf などのツールで取得できますが、本章では Windows WDK に含まれる traceview.exe を使用します。

traceview.exe を使用すると、GUI 上で簡単に ETW トレースログの取得と参照を行うことができるので便利です。

WDK をインストール済みの環境の場合、`C:\Program Files (x86)\Windows Kits\10\bin\<バージョン>\x64\traceview.exe` を実行することで traceview.exe を起動できます。[*2]

traceview.exe を起動したら、SampleAmsiProvider のイベントトレースを取得するため、GUID {00604c86-2d25-46d6-b814-cd149bfdf0b3} を指定して新しいトレースセッションを開始します。

[*2] Download the Windows Driver Kit (WDK) https://learn.microsoft.com/ja-jp/windows-hardware/drivers/download-the-wdk

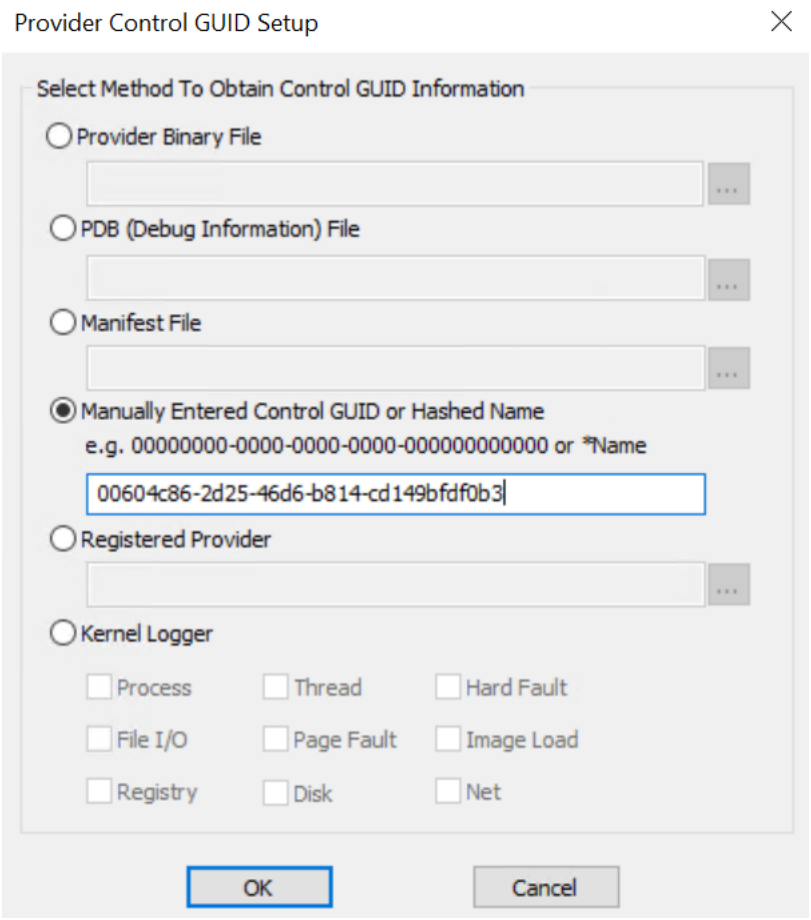

図 3.3: ETW プロバイダーの GUID を指定する

この GUID は、SampleAmsiProvider のコード内で以下の通り定義されています。

```
// Define a trace logging provider:
// 00604c86-2d25-46d6-b814-cd149bfdf0b3
TRACELOGGING_DEFINE_PROVIDER(
  g_traceLoggingProvider, "SampleAmsiProvider",
  (0x00604c86, 0x2d25, 0x46d6, 0xb8, 0x14, 0xcd, 0x14, 0x9b
, 0xfd, 0xf0, 0xb3)
);
```

SampleAmsiProvider のイベントトレース取得を開始したら、2 章で使用したサンプルプログラム AmsiStream を実行して AMSI スキャン要求を行います。

その結果、SampleAmsiProvider による複数のイベント出力が行われることを、Trace View の GUI 上で確認することができます。

ID	Msg#	Name	Process	Thread	CPU	Sequence	System Time	Message
0	3	SampleAmsiProvider	7072	6104	2	0	09\07\202...	{"meta":{"provider":"SampleAmsiProvider","event":"Loaded","time":"2025-09-07T03:24:32.036","cpu":2,"pid":7072,"tid":6104,"channel":11,"level":5}}
	4	SampleAmsiProvider	7072	6104	2	0	09\07\202...	{"requestNumber":1,"meta":{"provider":"SampleAmsiProvider","event":"Scan Start","time":"2025-09-07T03:24:32.057","cpu":2,"pid":7072,"tid":6104,"channel":11,"level":5}}
	5	SampleAmsiProvider	7072	6104	2	0	09\07\202...	{"requestNumber":1,"App Name":"Contoso Script Engine v3.4.9999.0","Content Name":"Sample content.txt","Content Size":13,"Session":0,"Content Address":"0x7FF79C43FBF8","meta"
	6	SampleAmsiProvider	7072	6104	2	0	09\07\202...	{"requestNumber":1,"result":44,"meta":{"provider":"SampleAmsiProvider","event":"Memory xor","time":"2025-09-07T03:24:32.058","cpu":2,"pid":7072,"tid":6104,"channel":11,"level":5}}
	7	SampleAmsiProvider	7072	6104	2	0	09\07\202...	{"requestNumber":1,"meta":{"provider":"SampleAmsiProvider","event":"Scan End","time":"2025-09-07T03:24:32.058","cpu":2,"pid":7072,"tid":6104,"channel":11,"level":5}}

図 3.4: SampleAmsiProvider のトレースログを取得する

まず最初に記録されたのは、`"Loaded"` というテキストのみが含まれるイベントでした。

```
{
  "meta": {
    "provider": "SampleAmsiProvider",
    "event": "Loaded",
    /* 省略 */
  }
}
```

続いて、`"requestNumber": 1` という情報と共に、テキスト `"Scan St`

art" を含むイベントが記録されていました。

これは、SampleAmsiProvider によりスキャンが開始されたことを示すイベントであり、requestNumber には SampleAmsiProvider で管理されている、要求されたスキャンの回数を示す情報が記録されます。

```
{
  "requestNumber": 1,
  "meta": {
    "provider": "SampleAmsiProvider",
    "event": "Scan Start",
    /* 省略 */
  }
}
```

スキャンが開始すると、スキャン対象の AMSI 属性情報に関するイベントが記録されます。

以下のイベントでは、サンプルプログラム AmsiStream から返された AMSI_ATTRIBUTE_APP_NAME や AMSI_ATTRIBUTE_CONTENT_ADDRESS などの各情報が記録されています。

```
{
  "requestNumber": 1,
  "App Name": "Contoso Script Engine v3.4.9999.0",
  "Content Name": "Sample content.txt",
  "Content Size": 13,
  "Session": 0,
  "Content Address": "0x7FF79C43FBF8",
  "meta": {
    "provider": "SampleAmsiProvider",
    "event": "Attributes",
    /* 省略 */
  }
}
```

最後に、AMSI スキャンにより受け取ったデータをバイトごとに XOR した結果を result として出力しています。

```
{
  "requestNumber": 1,
  "result": 44,
  "meta": {
    "provider": "SampleAmsiProvider",
    "event": "Memory xor",
    /* 省略 */
  }
}
```

この結果は、確かにサンプルプログラム AmsiStream がメモリスキャン時に使用する文字列 `Hello, world` の各バイトを XOR した値と一致することを以下の Python スクリプトで確認できます。

```
scan_conten = "Hello, world"
result = 0

for byte_data in scan_conten:
    result ^= ord(byte_data)

print(result) # 44
```

これで、サンプルプロバイダー SampleAmsiProvider が AMSI スキャン要求を受け取り、コンテンツの処理を行っていることを確認することができました。

3.3 AMSI プロバイダーの実装

サンプルプロバイダー SampleAmsiProvider の動作を確認できたので、詳しい実装を解説していきます。

以下のステップは、システムに登録されたサンプルプロバイダー SampleAmsiProvider の動作を簡略化したものです。

1. システムに登録されている AMSI プロバイダーの DLL が AMSI インターフェースによりロードされる
2. DLLMain 関数内にて ETW プロバイダーの登録や COM コンポーネント初期化の処理を行う
3. クライアントアプリケーションからの要求を受けて Scan メソッドを実行する
4. クライアントアプリケーションから AMSI 属性情報を取得し、ETW トレースログとして出力する
5. クライアントアプリケーションから読み込んだスキャンデータの各バイトを XOR した結果を ETW トレースログ出力する

本章では、IAntimalwareProvider インターフェースを継承する SampleAmsiProvider クラスの実装を中心に、サンプルプロバイダーの実行コードを追っていくことにします。

なお、amsi.dll にて実装されているインターフェースがシステムに登録されている AMSI プロバイダーの DLL をロードする際の動作については、公開されているサンプルコードの実装には含まれないため本書では扱いません。

しかし、上記のような動作については、本書でも参考情報として使用している Evading EDR の 10 章で詳しく紹介されているため、ご興味があれば

一読いただくことをおすすめします。[*3]

SampleAmsiProvider クラスの実装

本章で解説するサンプルプロバイダーの中心となる SampleAmsiProvider クラスは以下のコードで実装されています。

```
class
  DECLSPEC_UUID("2E5D8A62-77F9-4F7B-A90C-2744820139B2")
  SampleAmsiProvider : \
    public RuntimeClass<RuntimeClassFlags<ClassicCom>,
           IAntimalwareProvider,
           FtmBase>
{
public:
  IFACEMETHOD(Scan)(_In_ IAmsiStream* stream,
                    _Out_ AMSI_RESULT* result) override;
  IFACEMETHOD_(void, CloseSession) \
                   (_In_ ULONGLONG session) override;
  IFACEMETHOD(DisplayName) \
                   (_Outptr_ LPWSTR* displayName) override;

private:
  // We assign each Scan request
  // a unique number for logging purposes.
  LONG m_requestNumber = 0;
};
```

上記のコードの通り、SampleAmsiProvider クラスは IAntimalware-Provider インターフェースを継承する COM コンポーネントとして定義されており、IAntimalwareProvider インターフェースに必要な 3 つのメソッド (Scan、CloseSession、DisplayName) をオーバーライドしています。

また、先ほど確認したサンプルプロバイダーの CLSID である {2E5D8A6

[*3] Evading EDR P.189 (Matt Hand 著 / No Starch Press / 2023 年)

2-77F9-4F7B-A90C-2744820139B2} が

なお、プライベートメンバとして定義されている m_requestNumber は、ETW トレースログに requestNumber として記録されるスキャン要求のカウンターとして使用されます。

SampleAmsiProvider クラスでオーバーライドされる Scan メソッドは、IAntimalwareProvider インターフェースの定義通り以下の引数を受け取り、スキャンの結果 (AMSI_RESULT) を返すメソッドです。[*4]

```
HRESULT SampleAmsiProvider::Scan(
          _In_ IAmsiStream* stream,
          _Out_ AMSI_RESULT* result
        )
```

このメソッドが呼び出されると、まず始めに InterlockedIncrement 関数によりインクリメントされた m_requestNumber の値と共に、スキャンの開始を ETW トレースログに記録します。

```
LONG requestNumber = InterlockedIncrement(&m_requestNumber)
;

TraceLoggingWrite(g_traceLoggingProvider,
                  "Scan Start",
                  TraceLoggingValue(requestNumber)
                );
```

続いて、GetStringAttribute 関数と GetFixedSizeAttribute 関数により、クライアントアプリケーションから各種 AMSI 属性情報を取得します。

[*4] IAntimalwareProvider::Scan method https://learn.microsoft.com/ja-jp/windows/win32/api/amsi/nf-amsi-iantimalwareprovider-scan

AMSI 属性情報の取得に使用している GetStringAttribute 関数と GetFixedSizeAttribute 関数の実装については後述しますが、いずれも IAmsiStream インターフェースの GetAttribute メソッドを使用してクライアントアプリケーションから AMSI 属性情報を取得するラッパー関数として動作します。

```
auto appName = GetStringAttribute(
                 stream,
                 AMSI_ATTRIBUTE_APP_NAME
               );

auto contentName = GetStringAttribute(
                    stream,
                    AMSI_ATTRIBUTE_CONTENT_NAME
                  );

auto contentSize = GetFixedSizeAttribute<ULONGLONG>(
                     stream,
                     AMSI_ATTRIBUTE_CONTENT_SIZE
                   );

auto session = GetFixedSizeAttribute<ULONGLONG>(
                 stream,
                 AMSI_ATTRIBUTE_SESSION
               );

auto contentAddress = GetFixedSizeAttribute<PBYTE>(
                        stream,
                        AMSI_ATTRIBUTE_CONTENT_ADDRESS
                      );
```

上記の処理で取得した AMSI 属性情報は、以下のコード部分にて ETW トレースログとして出力されます。

```
TraceLoggingWrite(
  g_traceLoggingProvider,"Attributes",
  TraceLoggingValue(requestNumber),
  TraceLoggingWideString(appName.Get(),"App Name"),
  TraceLoggingWideString(contentName.Get(),"Content Name"),
  TraceLoggingUInt64(contentSize,"Content Size"),
  TraceLoggingUInt64(session,"Session"),
  TraceLoggingPointer(contentAddress,"Content Address")
);
```

続いて、SampleAmsiProvider は要求されたスキャンコンテンツの処理を行います。

この時の動作は、AMSI 属性情報 AMSI_ATTRIBUTE_CONTENT_ADDRESS と対応するスキャンコンテンツが保存されたメモリアドレスを取得できたかどうかによって変化します。

SampleAmsiProvider がスキャンコンテンツが保存されたメモリアドレス (contentAddress) を取得できた場合、Scan メソッド内では CalculateBufferXor 関数を呼び出してメモリ内のデータをバイトごとに XOR した結果をイベントに書き込みます。

```
// The data to scan is provided in the form of a memory buf
fer.
auto result = CalculateBufferXor(
                contentAddress,
                contentSize
              );

TraceLoggingWrite(g_traceLoggingProvider, "Memory xor",
   TraceLoggingValue(requestNumber),
   TraceLoggingValue(result));
```

サンプルプロバイダー SampleAmsiProvider の場合は実際のマルウェ

アスキャンは行わないため、スキャンコンテンツに対して行われる処理はCalculateBufferXor 関数で実装されています。

以下の通り、CalculateBufferXor 関数は単純にバッファ内のデータを 1 バイトずつ取り出して XOR した結果を返す処理のみを行います。

```
BYTE CalculateBufferXor(
  _In_ LPCBYTE buffer,
   _In_ ULONGLONG size
) {
  BYTE value = 0;
  for (ULONGLONG i = 0; i < size; i++)
  {
    value ^= buffer[i];
  }
  return value;
}
```

なお、実際の AMSI プロバイダーの場合は、CalculateBufferXor 関数の代わりにインストールされているアンチマルウェア製品のスキャンエンジンを使用してバッファ内のデータのスキャンを行います。

一方、以下は SampleAmsiProvider がスキャンコンテンツが保存されたメモリアドレス (contentAddress) を取得できなかった場合に実行されるコード部分です。

スキャンコンテンツを CalculateBufferXor 関数で XOR した結果を ETW トレースログに書き込む点については共通ですが、処理するデータについては IAmsiStream インターフェースの Read メソッドを呼び出すことで取得している点が異なります。

```
// Provided as a stream. Read it stream a chunk at a time.
BYTE cumulativeXor = 0;
BYTE chunk[1024];
ULONG readSize;

for (ULONGLONG position = 0;
     position < contentSize;
     position += readSize
    )
{
  HRESULT hr = stream->Read(
                position,
                sizeof(chunk),
                chunk,
                &readSize
              );

  if (SUCCEEDED(hr))
  {
    cumulativeXor ^= CalculateBufferXor(chunk, readSize);
    TraceLoggingWrite(g_traceLoggingProvider,"Read chunk",
      TraceLoggingValue(requestNumber),
      TraceLoggingValue(position),
      TraceLoggingValue(readSize),
      TraceLoggingValue(cumulativeXor));
  }
  else
  {
    TraceLoggingWrite(g_traceLoggingProvider,"Read failed",
      TraceLoggingValue(requestNumber),
      TraceLoggingValue(position),
      TraceLoggingValue(hr));
    break;
  }
}
```

Scan メソッドと同じく SampleAmsiProvider でオーバーライドされて

いる DisplayName メソッドは以下の通りシンプルに実装されています。[*5]

このメソッドは、要求に対して AMSI プロバイダー名を返すように実装されています。

```
HRESULT SampleAmsiProvider::DisplayName(
  _Outptr_ LPWSTR *displayName
) {
  *displayName = const_cast<LPWSTR>(L"Sample AMSI Provider"
);
  return S_OK;
}
```

上記のメソッドは、2 章で解説したサンプルプログラムでは以下のように使用されており、Scan メソッドのアウトプットとして受け取った AMSI プロバイダーの名前の取得を行っています。

```
PWSTR name;
hr = provider->DisplayName(&name);

if (SUCCEEDED(hr)) {
  wprintf(L"Provider display name: %s\n",
          name
         );
 CoTaskMemFree(name);
}
```

最後の CloseSession メソッドは、`"Close session"` という文字列を ETW トレースログに出力するだけのメソッドとして実装されています。

[*5] IAntimalwareProvider::DisplayName method https://learn.microsoft.com/ja-jp/windows/win32/api/amsi/nf-amsi-iantimalwareprovider-displayname

```
void SampleAmsiProvider::CloseSession(_In_ ULONGLONG sessio
n)
{
  TraceLoggingWrite(
    g_traceLoggingProvider,
    "Close session",
    TraceLoggingValue(session)
  );
}
```

実際には、CloseSession メソッドは指定の AMSI セッションを閉じるためのメソッドとして定義されているため、プロバイダーの実装によっては何らかのリソースの解放などを担う可能性があります。[*6]

GetStringAttribute 関数の実装

GetStringAttribute 関数は、前項で解説した Scan メソッドの中で、`AMSI_ATTRIBUTE_APP_NAME` と `AMSI_ATTRIBUTE_CONTENT_NAME` の属性情報の取得に使用されていた関数です。

この関数は、以下の通りスキャンメソッドが受け取る IAmsiStream インターフェースのオブジェクトと、取得したい AMSI 属性情報を指定する値を引数として取ります。

```
HeapMemPtr<wchar_t> GetStringAttribute(
  _In_ IAmsiStream* stream,
  _In_ AMSI_ATTRIBUTE attribute
)
```

なお、戻り値として指定されている HeapMemPtr は、ヒープの割り当てと解放を行うためにサンプルプロバイダー内で定義されているクラスです。

[*6] IAntimalwareProvider::CloseSession method https://learn.microsoft.com/ja-jp/windows/win32/api/amsi/nf-amsi-iantimalwareprovider-closesession

HeapMemPtr では、API 関数 HeapAlloc を使用したメモリの割り当てを行います。

GetStringAttribute 関数は、以下の通り実装されています。

```
HeapMemPtr<wchar_t> result;

ULONG allocSize;
ULONG actualSize;
if (
  stream->GetAttribute(
            attribute,
            0,
            nullptr,
            &allocSize
          ) == E_NOT_SUFFICIENT_BUFFER &&
  SUCCEEDED(result.Alloc(allocSize)) &&
  SUCCEEDED(stream->GetAttribute(
            attribute,
            allocSize,
            reinterpret_cast<PBYTE>(result.Get()),
            &actualSize
           )
          ) && actualSize <= allocSize)
{
  return result;
}
return HeapMemPtr<wchar_t>();
```

上記のコード内では、要求に必要なメモリサイズを特定するために、まずあえて dataSize と対応する第 2 引数に 0 を指定した上で GetAttribute メソッドの呼び出しを行い、その結果が E_NOT_SUFFICIENT_BUFFER エラーとなることを確認しています。

```
stream->GetAttribute(
          attribute,
          0,
          nullptr,
          &allocSize
        ) == E_NOT_SUFFICIENT_BUFFER
```

これは、2 章の「図 2.10: GetAttribute メソッドのパラメータ情報」に記載の通り、GetAttribute メソッドが E_NOT_SUFFICIENT_BUFFER を返す場合に、第 4 引数として受け取る retData に要求に必要なバイト数が書き込まれることを利用しています。

上記のように GetAttribute メソッドで要求に必要なサイズを確認したら、HeapMemPtr<wchar_t> のオブジェクトである result を使用し、result.Alloc(allocSize) にて結果を取得するために必要なメモリ領域の割り当てを行います。

その後、再度以下のように GetAttribute メソッドを呼び出すことで、result.Get() で取得した出力先のバッファにて要求した AMSI 属性情報を受け取ります。

```
stream->GetAttribute(
           attribute,
           allocSize,
           reinterpret_cast<PBYTE>(result.Get()),
           &actualSize
         )
```

GetFixedSizeAttribute 関数の実装

前項で解説した GetStringAttribute 関数とは異なり、GetFixedSizeAttribute 関数は AMSI_ATTRIBUTE_CONTENT_SIZE や AMSI_ATTRIBUTE_SE

SSION および AMSI_ATTRIBUTE_CONTENT_ADDRESS のように、取得する情報のサイズが固定されている AMSI 属性情報の取得を行います。

そのため、関数の実装は GetStringAttribute 関数と比較してよりシンプルであり、単純に GetAttribute メソッドにて指定した属性の値をバッファで受け取る関数となっています。

```
template<typename T>
T GetFixedSizeAttribute(
  _In_ IAmsiStream* stream,
  _In_ AMSI_ATTRIBUTE attribute
) {
    T result;

    ULONG actualSize;
    if (SUCCEEDED(stream->GetAttribute(
                    attribute,
                    sizeof(T),
                    reinterpret_cast<PBYTE>(&result),
                    &actualSize
                  )
                ) &&
        actualSize == sizeof(T))
    {
        return result;
    }
    return T();
}
```

3.4 3 章のまとめ

本章では、サンプルプロバイダー AmsiProvider が AMSI インターフェースを使用して要求されたスキャンコンテンツを受け取り、またそれを処理する仕組みを解説しました。

サンプルプロバイダーの実装から、AMSI を通して要求されたスキャンを

行うためには、以下の実装が必要であることがわかりました。

- システムに AMSI プロバイダー を登録する
- IAntimalwareProvider インターフェースにて定義された各メソッドをオーバーライドする
- AMSI インターフェースを通して Scan 要求を受けた際に、各種 AMSI 属性情報とスキャンコンテンツの取得を行う
- 取得したスキャンコンテンツに対して、アンチマルウェアスキャンなどの任意の処理行い、結果 (`AMSI_RESULT`) を返す

なお、本章で解説したサンプルプロバイダー AmsiProvider は、すべてのスキャン要求に対して `AMSI_RESULT_NOT_DETECTED` を返却してしまうため、実際に作成した AMSI プロバイダーの結果によりコンテンツがブロックされる動作を確認することができませんでした。

そこで、次の 4 章では、本書で解説したサンプルコードをカスタマイズすることで、自身が登録した AMSI プロバイダーによりコンテンツがブロックされることを確認するとともに、より AMSI スキャンの動作に対する理解を深めることに挑戦します。

第 4 章

サンプルプログラムのカスタマイズ

本章では、2 章と 3 章で解説したサンプルプログラムのカスタマイズを通して、より AMSI スキャンに対する理解を深めることに挑戦します。

具体的に、本章ではサンプルプログラムに対してそれぞれ以下のカスタマイズを行います。

- サンプルプロバイダー
  - 要求されたスキャンコンテンツを出力する
  - 簡易的なコンテンツスキャナーを実装し、動作ブロックを行う
- サンプルプログラム
  - 入力として受け取った文字列をメモリコンテンツとして保存し、AMSI によりスキャンする

なお、カスタマイズ後のサンプルコードについては、下記リポジトリにて公開しております。

https://github.com/kash1064/AMSI-Samples

4.1 AMSI プロバイダーのカスタマイズ

まずは、3 章で解説した AMSI プロバイダーのサンプルコードをカスタマイズします。

既に解説した通り、AMSI プロバイダーのサンプルは実際には要求されたデータのスキャンは行わず、すべての要求に対して AMSI_RESULT_NOT_DETECTED を返却します。

そのため、今回はサンプルプロバイダーに簡易的なスキャナーを実装し、コンテンツを不正と判定する機能を追加していきます。

サンプルプロバイダーにスキャン機能を追加する

3 章で確認した通り、SampleAmsiProvider がスキャンコンテンツが保存されたメモリアドレス (contentAddress) を取得できた場合には、Scan メソッド内の以下のコード部分にて CalculateBufferXor 関数を呼び出してメモリ内のデータをバイトごとに XOR した結果をイベントに書き込む操作を行います。

```
// The data to scan is provided in the form of a memory buf
fer.
auto result = CalculateBufferXor(
                contentAddress,
                contentSize
              );

TraceLoggingWrite(g_traceLoggingProvider, "Memory xor",
   TraceLoggingValue(requestNumber),
   TraceLoggingValue(result));
```

今回は、上記のコードをすべてコメントアウトし、新たに作成した Mem-

oryContentScan 関数を呼び出すように書き換えました。

```
*result = MemoryContentScan(contentAddress, contentSize, ap
pName.Get());
```

MemoryContentScan 関数は、以下の通りスキャン対象のデータが含まれるバッファとそのサイズ、また AMSI スキャン要求を行ったアプリケーションから取得した属性情報である AMSI_ATTRIBUTE_APP_NAME の 3 つを引数に取ります。

この関数の中では、スキャン要求がサンプルプログラム AmsiStream からのものであるか、もしくは PowerShell からのものであるかを確認し、処理を分岐します。

```
AMSI_RESULT MemoryContentScan(
  _In_ LPCBYTE buffer,
  _In_ ULONGLONG size,
  _In_ PCWSTR appName)
{

    if (wcsncmp(
          appName,
          L"Contoso Script Engine",
          wcslen(L"Contoso Script Engine")
        ) == 0)
    {
        /* サンプルプログラムからのスキャン要求を処理 */
    }
    else if (wcsncmp(
              appName,
              L"PowerShell",
              wcslen(L"PowerShell")
            ) == 0)
    {
```

```
        /* PowerShell からのスキャン要求を処理 */
    }

    return AMSI_RESULT_NOT_DETECTED;

}
```

スキャンの要求元がサンプルプログラム AmsiStream である場合には、バッファ内から取り出したスキャン対象の ASCII 文字列を ETW トレースログとして出力した後、データ内に Malicious もしくは Clean のいずれかの文字列が含まれるかをスキャンします。

```
LPSTR ascii_text = (LPSTR)malloc(
                           strlen((const char*)buffer) + 1
                         );
if (!ascii_text) return AMSI_RESULT_NOT_DETECTED;
memcpy(ascii_text, buffer, strlen((const char*)buffer));
ascii_text[strlen((const char*)buffer)] = '\0';

TraceLoggingWrite(
  g_traceLoggingProvider,
  "Scan Contoso Script Engine Content",
  TraceLoggingString(ascii_text, "Buffer")
);

static const char malicious[] = "Malicious";
static const char clean[] = "Clean";
size_t len_malicious = strlen(malicious);
size_t len_clean = strlen(clean);

for (size_t i = 0; i <= strlen(ascii_text); i++) {
  if (i <= strlen(ascii_text) - len_malicious) {
    if (memcmp(
          &ascii_text[i],
          malicious,
          len_malicious
```

```
        ) == 0)
    {
      return AMSI_RESULT_DETECTED;
    }
  }

  if (i <= strlen(ascii_text) - len_clean) {
    if (memcmp(&ascii_text[i], clean, len_clean) == 0) {
      return AMSI_RESULT_CLEAN;
    }
  }
}
```

ここで、データ内に文字列 Malicious が含まれる場合には、サンプルプロバイダーはスキャン結果として AMSI_RESULT_DETECTED を返します。

一方で、データ内に文字列 Clean が含まれる場合、サンプルプロバイダーはスキャン結果として AMSI_RESULT_CLEAN を返します。

また、上記のどちらの文字列も含まれない場合には、AMSI プロバイダーは最終的に AMSI_RESULT_NOT_DETECTED を結果として返します。

もし、スキャンの要求元が PowerShell である場合には、サンプルプロバイダーは受け取ったスキャンコンテンツを ETW トレースログに出力した後、Malicious-Script または Clean-Script のいずれかのワイド文字列が含まれるかをスキャンします。

```
TraceLoggingWrite(
  g_traceLoggingProvider,
  "Scan PowerShell Content",
  TraceLoggingCountedWideString(
    (PCWSTR)buffer,
    wcslen((PCWSTR)buffer),
    "Buffer"
  )
```

```
);

static const wchar_t malicious[] = L"Malicious-Script";
static const wchar_t clean[] = L"Clean-Script";
size_t len_malicious = wcslen(malicious);
size_t len_clean = wcslen(clean);

for (size_t i = 0; i <= size; i++) {
  if (i <= size - len_malicious) {
    if (wcsncmp(
         (PCWSTR)&buffer[i],
         malicious,
         len_malicious
        ) == 0)
    {
        return AMSI_RESULT_DETECTED;
    }
  }

  if (i <= size - len_clean) {
    if (wcsncmp(
         (PCWSTR)&buffer[i],
         clean,
         len_clean
        ) == 0)
    {
        return AMSI_RESULT_CLEAN;
    }
  }
}
```

スキャンの結果は要求元がサンプルプログラムの場合と同様に評価され、ここで、データ内にワイド文字列 Malicious-Script が含まれる場合には AMSI_RESULT_DETECTED が、ワイド文字列 Clean-Script が含まれる場合には AMSI_RESULT_CLEAN がスキャンの結果として返されます。

4.2　サンプルプログラムのカスタマイズ

次に、2 章で解説したサンプルプログラム AmsiStream に任意の文字列を入力する機能を追加します。

サンプルプログラムに入力機能を追加する

2 章で解説した通り、サンプルプログラム AmsiStream では実行時にコマンドライン引数を指定しない場合、グローバル定数 SampleStream としてハードコードされた文字列を AMSI によりスキャンします。

そこで、まずはグローバル変数 SampleStream を以下のように再定義し、後で標準入力から受け取った文字列を保存するバッファとして使用します。

```
#define BUF_SIZE 256
char SampleStream[BUF_SIZE];
```

次に、ScanArguments 関数内で CAmsiMemoryStream クラスの初期化を行う直前に、標準入力から受け取った文字列をグローバル変数 SampleStream に書き込むコードを追加します。

```
// Scan a single memory stream.
wprintf(L"Creating memory stream object\n");
wprintf(L"Please input for memory stream scan\n");

if (fgets(SampleStream, BUF_SIZE, stdin) != NULL) {
  SampleStream[strcspn(SampleStream, "\n")] = '\0';
}

printf("Input text: %s\n", SampleStream);
```

これにより、標準入力から受け取った任意の文字列をスキャンすることができるようになりました。

カスタマイズしたプログラムを実行する

次に、カスタマイズしたサンプルプログラムとサンプルプロバイダーをそれぞれ再ビルドし、仮想マシンにインストールすることで動作確認を行います。

もし、すでにサンプルプロバイダーを AMSI プロバイダーとして登録済みの場合は、事前に以下のコマンドでサンプルプロバイダーのアンロードを行った後に、DLL ファイルを差し替える必要があります。

```
regsvr32 /u AmsiProvider.dll
```

再ビルドした AmsiProvider.dll を仮想マシンに配置したら、3 章と同じく管理者権限で起動したコマンドプロンプトで以下のコマンドを実行し、更新したサンプルプロバイダーをシステムに登録します。

```
regsvr32 AmsiProvider.dll
```

次に、こちらも 3 章と同じ手順で Windows WDK に含まれる traceview.exe を起動し、サンプルプロバイダーの GUID である {00604c86-2d25-46d6-b814-cd149bfdf0b3} を指定して新しいトレースセッションを開始します。

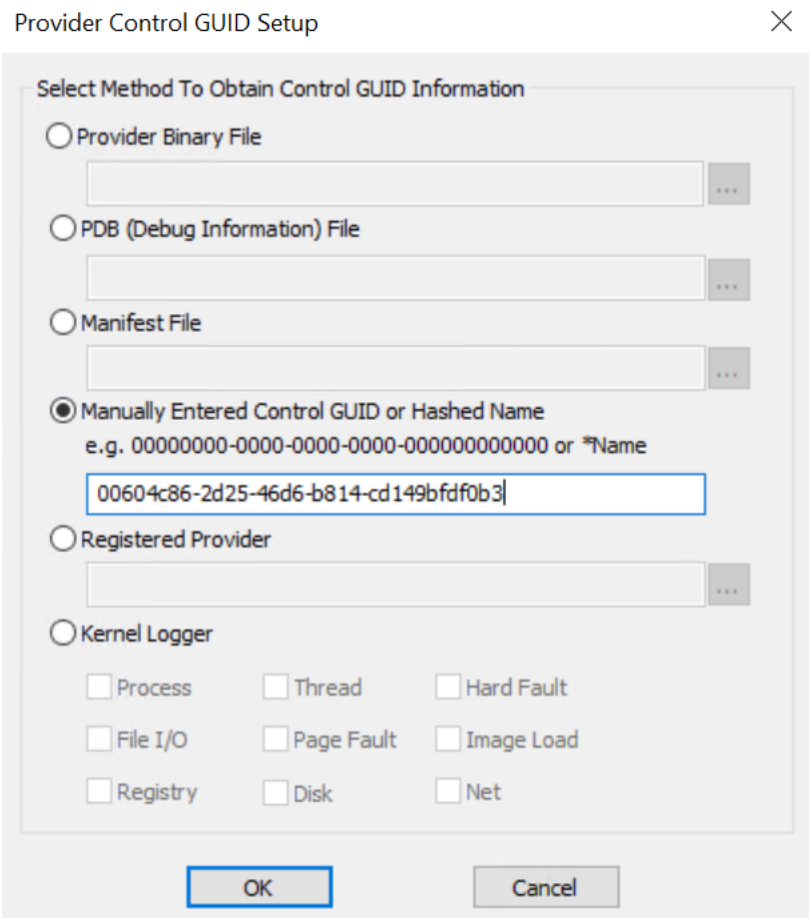

図 4.1: ETW プロバイダーの GUID を指定する

すべての準備が完了したら、まずは再ビルドしたサンプルプログラム AmsiStream を起動し、標準入力から受け取った任意の文字列をスキャン対象に指定できること、また、文字列 `Malicious` を含むテキストがサンプルプロバイダーにより検知されることを確認できます。

```
C:\Users\kash1064\Downloads\AMSI>AmsiStream.exe
Creating memory stream object
Please input for memory stream scan
Test: Malicious content
Input text: Test: Malicious content
Calling antimalware->Scan() ...
GetAttribute() called with: attribute = 0, bufferSize = 1
GetAttribute() called with: attribute = 0, bufferSize = 68
GetAttribute() called with: attribute = 1, bufferSize = 1
GetAttribute() called with: attribute = 1, bufferSize = 38
GetAttribute() called with: attribute = 2, bufferSize = 8
GetAttribute() called with: attribute = 3, bufferSize = 8
GetAttribute() called with: attribute = 4, bufferSize = 8
GetAttribute() called with: attribute = 6, bufferSize = 8
GetAttribute() called with: attribute = 8, bufferSize = 8
GetAttribute() called with: attribute = 0, bufferSize = 0
GetAttribute() called with: attribute = 0, bufferSize = 68
GetAttribute() called with: attribute = 1, bufferSize = 0
GetAttribute() called with: attribute = 1, bufferSize = 38
GetAttribute() called with: attribute = 2, bufferSize = 8
GetAttribute() called with: attribute = 4, bufferSize = 8
GetAttribute() called with: attribute = 3, bufferSize = 8
GetAttribute() called with: attribute = 2, bufferSize = 8
GetAttribute() called with: attribute = 3, bufferSize = 8
GetAttribute() called with: attribute = 4, bufferSize = 8
GetAttribute() called with: attribute = 0, bufferSize = 1
GetAttribute() called with: attribute = 0, bufferSize = 68
GetAttribute() called with: attribute = 1, bufferSize = 1
GetAttribute() called with: attribute = 1, bufferSize = 38
Scan result is 32768. IsMalware: 1
Provider display name: Sample AMSI Provider
```

図 4.2: 文字列 Malicious のスキャン結果

さらに、traceview.exe で取得した上記のスキャン時の ETW トレースログ上でも、確かにサンプルプロバイダーが `Test:  Malicious content` という文字列をスキャンしたことを確認できます。

```
{
  "Buffer": "Test: Malicious content",
  "meta": {
    "provider": "SampleAmsiProvider",
    "event": "Scan Contoso Script Engine Content",
    /* 省略 */
  }
}
```

続けて、以下の PowerShell コマンドを実行した場合にもサンプルプロバイダーによる検知が行われることを確認します。

```
$testString = "Sample Provider: " + "Malicious" + "-" + "Sc
ript"
Invoke-Expression $testString
```

上記のコマンドを実行した場合、文字列 Malicious-Script は分割により難読化されているため、1 行目のコマンド実行はサンプルプロバイダーによる検知は回避されます。

しかし、2 行目の `Invoke-Expression $testString` にて分割された文字列を結合した上で評価する際には、サンプルプロバイダーにより不正なワイド文字列 Malicious-Script が検知され、コマンドの実行が AMSI によりブロックされることを確認できます。

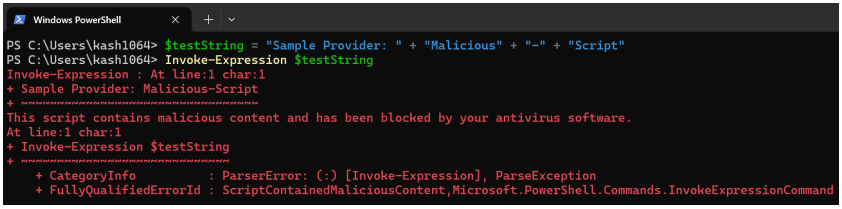

図 4.3: PowerShell コマンドをサンプルプロバイダーでブロックする

また、この時にサンプルプロバイダーの出力する ETW トレースログを確認すると、PowerShell からサンプルプロバイダーが不正なワイド文字列 Malicious-Script を受け取るまでの一連のスキャンコンテンツを追跡できます。

図 4.4: PowerShell コマンドブロック時のトレースログ

4.3 4 章のまとめ

本章では、これまでに解説したサンプルコードのカスタマイズを行い、サンプルプロバイダーとサンプルプログラムに以下の機能を追加しました。

- サンプルプロバイダー
  - 要求されたスキャンコンテンツを出力する

  - 簡易的なコンテンツスキャナーを実装し、動作ブロックを行う
- サンプルプログラム
  - 入力として受け取った文字列をメモリコンテンツとして保存し、AMSI によりスキャンする

最後の 5 章では、AMSI が実際のプロダクトにどのように統合されているかについて、OSS としてソースコードが公開されている PowerShell を例に解説します。

第 5 章

PowerShell に統合された AMSI

本章では、2 章で解説したクライアントアプリケーションへの AMSI 統合が実際のプロダクトでどのように使用されているかを紹介します。

本書では、実際のプロダクトの例として、OSS でソースコードが公開されている PowerShell を使用します。

なお、下記のイメージ図の通り PowerShell は 2 章で解説したサンプルプログラムとは異なり、COM API レイヤーの IAmsiStream::Scan メソッドではなく、Win32 API レイヤーでラップされた AmsiScanBuffer 関数などを使用して、システムに登録されている AMSI プロバイダーにスキャンを要求します。

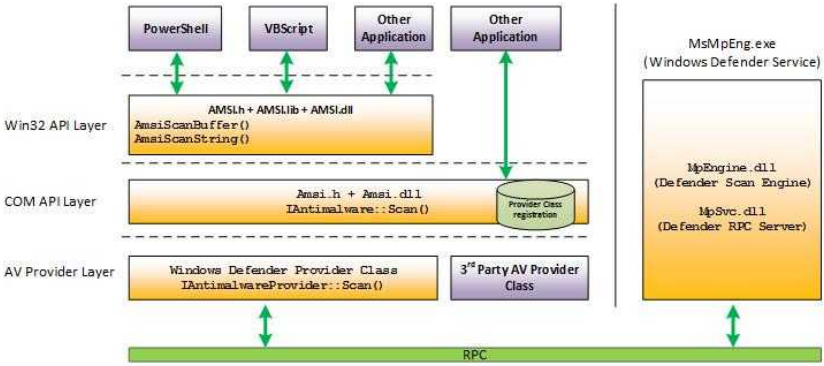

図 5.1: AMSI のイメージ図 (公開情報より引用)

しかし、Win32 API レイヤーの関数を使用する場合も、AMSI スキャン時の流れは基本的には 2 章で解説したサンプルプログラムと同等です。

本章では、公開されている PowerShell のソースコードから AMSI スキャン要求を行い、その結果を元にコードの実行可否を決定する部分の動作を解説します。

なお、PowerShell の詳しい動作や実装については本書のスコープ外であるため、詳しい解説は行いません。

5.1 PowerShell のソースコードを取得する

OSS として提供されている PowerShell のソースコードは、GitHub リポジトリ https://github.com/PowerShell/PowerShell から取得できます。

本章では、https://github.com/PowerShell/PowerShell/tree/v7.5.0 のソースコードを利用します。

Git を使用可能な場合は、以下のコマンドを順に実行することでも同様のソースコードを参照できるようになります。

```
# GitHub リポジトリの Clone
git clone https://github.com/PowerShell/PowerShell.git

# Clone したフォルダに移動
cd PowerShell

# v7.5.0 tag の Checkout
git checkout v7.5.0
```

5.2 PowerShell による AMSI スキャン要求

まずは、PowerShell がどのように AMSI スキャン要求を行っているのかを特定します。

冒頭でも紹介した以下のイメージ図の通り、PowerShell は Win32 API レイヤーの AmsiScanBuffer 関数を使用して AMSI スキャンを要求します。

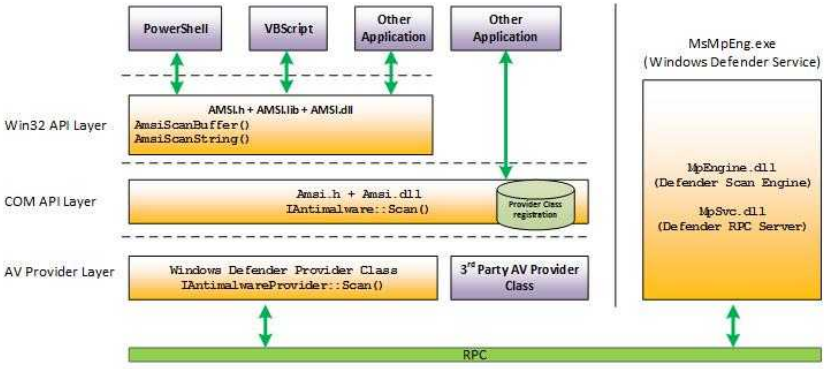

図 5.2: AMSI のイメージ図 (公開情報より引用)

本章では、PowerShell が AmsiScanBuffer 関数を使用してスキャンを行う際の動作を確認していきます。

AmsiScanBuffer 関数について

AmsiScanBuffer 関数とは、AMSI を使用してバッファ内のコンテンツスキャンを AMSI プロバイダーに要求するために使用できる Win32 API レイヤーの関数です。[*1]

AmsiScanBuffer 関数は以下通り HAMSICONTEXT 型のハンドルオブジェクトやスキャン対象のデータが含まれるバッファなどを引数として呼び出されます。

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

この AmsiScanBuffer 関数は amsi.dll にて実装されています。

また、amsi.dll は Microsoft からパブリックシンボルが配布されているモジュールであるため、比較的容易にデバッグを行うことができます。

5.3 AmsiScanBuffer 関数呼び出し時のコールスタックを調査する

まずは WinDbg を使用して AmsiScanBuffer 関数呼び出し時のコールスタックを取得します。

[*1] AmsiScanBuffer function https://learn.microsoft.com/ja-jp/windows/win32/api/amsi/nf-amsi-amsiscanbuffer

WinDbg を使用して `amsi!AmsiScanBuffer` にブレークポイントをセットした後に、デバッグ中の PowerShell プロンプトで適当なコマンドを実行しようとすると、以下のように AmsiScanBuffer 関数呼び出し時のコールスタックを取得できるようになります。

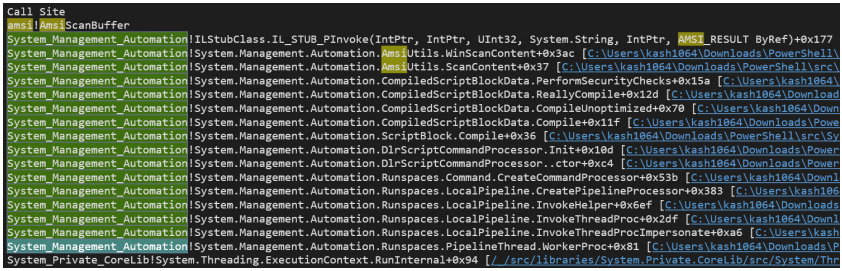

図 5.3: AmsiScanBuffer 関数呼び出し時のコールスタック

このコールスタックを見ると、System.Management.Automation.dll で実装されている AmsiUtils.ScanContent メソッドがさらに AmsiUtils.WinScanContent メソッドを呼び出し、最終的に Win32 API レイヤーの AmsiScanBuffer 関数の実行に繋がっていることがわかります。

また、この呼び出しは同じく System.Management.Automation.dll で実装されている CompiledScriptBlockData.PerformSecurityChecks メソッドから行われていることを確認できます。

本書では PowerShell の詳しい実装については扱いませんが、この PerformSecurityChecks メソッドは PowerShell で実行されるスクリプトが実行前に最終的にコンパイルされる際に呼び出される System.Management.Automation の ReallyCompile メソッドから実行されていることがわかります。[*2]

[*2] https://blogs.blackberry.com/en/2018/04/how-to-implement-anti-malware-scanning-interface-provider

上記の PerformSecurityChecks メソッドが呼び出される際の操作は src/System.Management.Automation/engine/runtime/CompiledScriptBlock.cs にて以下の通り定義されています。

```
private void ReallyCompile(bool optimize)
{
/* 省略 */

  PerformSecurityChecks();

  Compiler compiler = new Compiler();
  compiler.Compile(this, optimize);

/* 省略 */
}
```

また、AmsiUtils.WinScanContent メソッドを呼び出して直接的に AMSI スキャン要求を開始する PerformSecurityChecks メソッドも、同じく CompiledScriptBlock.cs にて定義されています。

PerformSecurityChecks メソッド

以下は、PowerShell によるコード実行時に直接的に AMSI スキャン要求を行う PerformSecurityChecks メソッドのうち、AMSI スキャンを要求し、結果が AMSI_RESULT_DETECTED であった場合に実行エラーを返す部分の抜粋です。

```
private void PerformSecurityChecks()
{
  /* 省略 */
```

```
    var scriptExtent = scriptBlockAst.Extent;
    var scriptFile = scriptExtent.File;

    /* 省略 */

    // Call the AMSI API to determine
    // if the script block has malicious content
    var amsiResult = AmsiUtils.ScanContent(
                          scriptExtent.Text,
                          scriptFile
                     );

    if (amsiResult == \
       AmsiUtils.AmsiNativeMethods.AMSI_RESULT.AMSI_RESULT_DE
  TECTED
    )
    {
      var parseError = new ParseError(
            scriptExtent,
            "ScriptContainedMaliciousContent",
            ParserStrings.ScriptContainedMaliciousContent
          );
          throw new ParseException(new[] { parseError });
    }

    /* 省略 */
  }
```

このメソッドの中では、`AmsiUtils.ScanContent(scriptExtent.Text, scriptFile);` を実行し、その結果を amsiResult として受け取っています。

その後、受け取ったスキャン結果 (amsiResult) を評価し、結果が `AMSI_RESULT_DETECTED` の場合には、エラー ID「ScriptContainedMaliciousContent」とともに ParseError を返し、コード実行をブロックします。

こうして、AMSI によって不正な PowerShell スクリプトの実行がブロックされ、以下のようなエラー画面が表示されることになります。

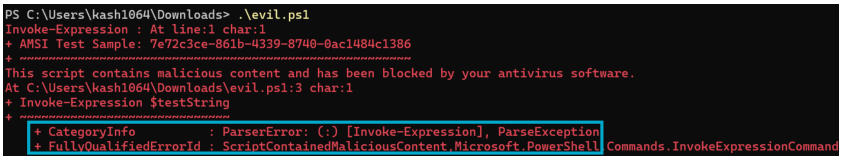

図 5.4: PowerShell スクリプトの実行ブロック

ここからは、PowerShell 内で AmsiUtils.ScanContent メソッドが呼び出され、AMSI を通して行ったスキャン結果を受け取るまでの流れを解説していきます。

AmsiUtils.ScanContent メソッド

ScanContent メソッドは、PowerShell が AMSI を使用して文字列バッファのスキャン要求を行うために呼び出されるメソッドであり、`src/System.Management.Automation/security/SecuritySupport.cs` で定義されています。

このメソッドは、PerformSecurityChecks メソッドから呼び出される際に、スキャン対象の文字列バッファである content と、実行されたスクリプトファイル名などを含む sourceMetadata の 2 つの引数を受け取り、それをそのまま使用して WinScanContent メソッドを呼び出します。

```
internal static AmsiNativeMethods.AMSI_RESULT
    ScanContent(
        string content,
        string sourceMetadata
    )
{
#if UNIX
  return AmsiNativeMethods.AMSI_RESULT.AMSI_RESULT_NOT_DETE
CTED;
```

```
#else
  return WinScanContent(
              content,
              sourceMetadata,
              warmUp: false
          );
#endif
}
```

ソースコードが公開されている PowerShell の場合には、デバッグ用のバイナリを自分でビルドすることでプライベートシンボルを使用したデバッグが可能になるため、以下のように関数呼び出し時の引数情報などを簡単に確認することができます。

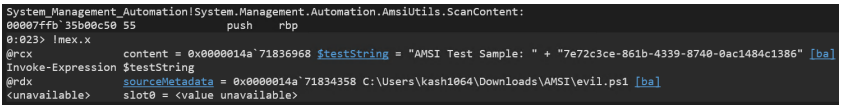

図 5.5: ScanContent メソッド呼び出し時の引数

なお、もちろんプライベートシンボル無しでも関数呼び出し時の引数に含まれる文字列情報を確認することはできますが、プライベートシンボルを使用した場合の方がスムーズかつ確実にデバッグ情報を確認できるのは明白です。

```
0:018> du @rcx+0xc
00000222`deb1f84c  "$testString = "AMSI Test Sample:"
00000222`deb1f88c  " " + "7e72c3ce-861b-4339-8740-0a"
00000222`deb1f8cc  "c1484c1386"..Invoke-Expression $"
00000222`deb1f90c  "testString"
0:018> du @rdx+0xc
00000222`deb1a8d4  "C:\Users\kash1064\Downloads\AMSI"
00000222`deb1a914  "\evil.ps1"
```

図 5.6: ScanContent メソッド呼び出し時の引数 (プライベートシンボル無し)

AmsiUtils.WinScanContent メソッド

WinScanContent メソッドは、ScanContent メソッドと同じく `src/System.Management.Automation/security/SecuritySupport.cs` で定義されている内部メソッドです。

前項で確認した通り、WinScanContent メソッドは ScanContent メソッドから以下の引数と共に呼び出されます。

```
internal static AmsiNativeMethods.AMSI_RESULT WinScanConten
t(
    string content,
    string sourceMetadata,
    bool warmUp)
```

引数 warmUp は AMSI コンポーネントのウォームアップ中であるかを指示する値であり、True が与えられた場合には実際には何のスキャンも行われず `AMSI_RESULT_NOT_DETECTED` が返されます。

そのため、通常のスキャン時に ScanContent メソッドから WinScanCon-

tent メソッドが呼び出される場合には、引数 warmUp の値はハードコードされている False が強制されます。

WinScanContent メソッドが呼び出し直後の箇所には、以下のような Eicar テストマルウェアの文字列をチェックするコード部分が存在しています。

しかし、これは恐らくデバッグ用途のフラグである InternalTestHooks.UseDebugAmsiImplementation が True の場合にのみ実行されるコードなので今回は無視します。

```
const string EICAR_STRING = "X5O!P%@AP[4\\PZX54(P^)7CC)7}$E
ICAR-STANDARD-ANTIVIRUS-TEST-FILE!$H+H*";
if (InternalTestHooks.UseDebugAmsiImplementation)
{
  if (content.Contains(EICAR_STRING, StringComparison.Ordin
al))
  {
    return AmsiNativeMethods.AMSI_RESULT.AMSI_RESULT_DETECT
ED;
  }
}
```

なお、余談ですが上記の検知動作については PowerShell プロンプト上で以下のコマンドを順に実行し、UseDebugAmsiImplementation を True に変更した上で Eicar テストマルウェアの文字列を含むコマンドを実行することでテストが可能です。

```
# UseDebugAmsiImplementation フラグを True に変更する
[System.Management.Automation.Internal.InternalTestHooks]::
SetTestHook('UseDebugAmsiImplementation', $true)
```

```
# Eicar テストマルウェアの文字列を含むコマンドを実行する
[ScriptBlock]::Create('X5O!P%@AP[4\PZX54(P^)7CC)7}$EICAR-ST
ANDARD-ANTIVIRUS-TEST-FILE!$H+H*') | Out-Null
```

以下は、実際に上記のコマンドを実行した場合の実行結果です。

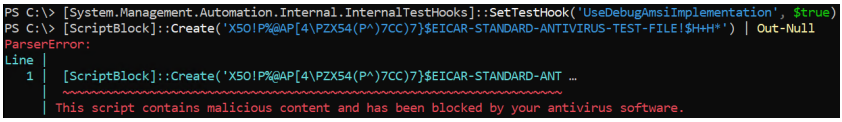

図 5.7: UseDebugAmsiImplementation を有効化する

続くコード部分では、AMSI スキャンのためにいくつかのチェックが行われます。

まず、以下では AMSI コンポーネントの初期化が成功しているかをチェックし、失敗している場合にはスキャンを要求せずに AMSI_RESULT_NOT_DETECTED を返すよう実装されています。

```
// If we had a previous initialization failure,
// just return the neutral result.
if (s_amsiInitFailed)
{
  PSEtwLog.LogAmsiUtilStateEvent(
          "ScanContent-InitFail",
          $"{s_amsiContext}-{s_amsiSession}"
      );
  return AmsiNativeMethods.AMSI_RESULT.AMSI_RESULT_NOT_DETE
CTED;
}
```

また、AMSI コンポーネントの初期化の失敗以外にも、前項で解説したように warmUp の値が True に設定されている場合などにも、WinScanContent

メソッドはスキャンを要求せずに AMSI_RESULT_NOT_DETECTED を返します。

これらのチェックにパスした場合、WinScanContent メソッドは冒頭で解説した Win32 API レイヤーの API 関数である AmsiScanBuffer を呼び出してコンテンツのスキャン結果を取得します。

```
AmsiNativeMethods.AMSI_RESULT result = \
    AmsiNativeMethods.AMSI_RESULT.AMSI_RESULT_CLEAN;

// Run AMSI content scan
int hr;
unsafe
{
  fixed (char* buffer = content)
  {
    var buffPtr = new IntPtr(buffer);
    hr = AmsiNativeMethods.AmsiScanBuffer(
            s_amsiContext,
            buffPtr,
            (uint)(content.Length * sizeof(char)),
            sourceMetadata,
            s_amsiSession,
            ref result
        );
    }
}
```

なお、この時呼び出される AmsiScanBuffer 関数は、AmsiNativeMethods クラスのメソッドとして呼び出されていますが、実際には amsi.dll で実装されている AmsiScanBuffer 関数を外部メソッドとして実行しています。

```
[DefaultDllImportSearchPaths(DllImportSearchPath.System32)]
[DllImport("amsi.dll", EntryPoint = "AmsiScanBuffer", Calli
ngConvention = CallingConvention.StdCall)]
internal static extern int AmsiScanBuffer(
System.IntPtr amsiContext,
  System.IntPtr buffer,
  uint length,
  [In][MarshalAs(UnmanagedType.LPWStr)] string contentName,
  System.IntPtr amsiSession,
  ref AMSI_RESULT result
);
```

AmsiScanBuffer 関数呼び出し後のスキャン動作

残念ながら、amsi.dll で実装されている AmsiScanBuffer 関数はソースコードが公開されていないため、本書では詳しい挙動は解説しません。

ただし、幸いなことに amsi.dll は Microsoft によりパブリックシンボルが公開されているため、比較的容易にデバッグを行うことができます。

実際に AmsiScanBuffer 関数呼び出し後のコールスタックを取得してみると、以下のように AmsiScanBuffer 関数から CAmsiAntimalware::Scan が呼び出されており、その後システムに登録されている各 AMSI プロバイダーがオーバーライドしている IAntimalwareProvider インターフェースの Scan メソッドが順に実行されたことを確認できます。

```
# Microsoft Defender AntiVirus の呼び出し
MpOav!CComMpOfficeAV::Scan
amsi!CAmsiAntimalware::Scan+0xf0
amsi!AmsiScanBuffer+0xd3
/* 省略 */

# SampleAmsiProvider の呼び出し
```

```
AmsiProvider!SampleAmsiProvider::Scan
amsi!CAmsiAntimalware::Scan+0xf0
amsi!AmsiScanBuffer+0xd3
/* 省略 */
```

上記の挙動から、AmsiScanBuffer 関数によりスキャン要求が行われた場合でも、3 章および 4 章で解説した流れでシステムに登録されている AMSI プロバイダーによるスキャンが実施されていることがわかります。

また、上記の結果から、要求されたデータはシステムに登録されているすべての AMSI プロバイダーによりスキャンされていることがわかります。

実際に、その結果として、Microsoft Defender AntiVirus のプロバイダーでのみ検知される `AMSI Test Sample: 7e72c3ce-861b-4339-8740-0ac1484c1386` という文字列と、4 章で実装したサンプルプロバイダー用の検知テスト文字列 `Sample Provider: Malicious-Script` がどちらも AMSI により検知されることを確認できます。

```
PS C:\> echo "AMSI Test Sample: 7e72c3ce-861b-4339-8740-0ac1484c1386"
ParserError:
Line |
   1 |  echo "AMSI Test Sample: 7e72c3ce-861b-4339-8740-0ac1484c1386"
     |  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
     | This script contains malicious content and has been blocked by your antivirus software.
PS C:\> echo "Sample Provider: Malicious-Script"
ParserError:
Line |
   1 |  echo "Sample Provider: Malicious-Script"
     |  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
     | This script contains malicious content and has been blocked by your antivirus software.
```

図 5.8: 各プロバイダーによる検知テスト

5.4 5 章のまとめ

本章では、実際に PowerShell に統合されている AMSI によりコンテンツスキャンが実施される動作の流れを解説しました。

本章で解説した通り、もし高度な手法によりスクリプトが難読化されており、ファイルスキャンによる検知がバイパスされている場合でも、最終的に実行コードがコンパイルされるまでのロジックの中に AMSI によるスキャンが組み込まれていることで、アンチマルウェアエンジンは難読化解除後のペイロードをスキャンし、脅威を実行前にブロックすることができます。

あとがき

本書を最後までお読みいただき誠にありがとうございました。

今回は、前回頒布した「A part of Anti-Virus -サンプルコードで学ぶ Windows AntiVirus とミニフィルタドライバー-」[*3]に続き、公式の AmsiStream および AmsiProvider のサンプルコードをベースに、AMSI (Windows Antimalware Scan Interface) の概要としくみを解説しました。

前著で解説したアンチマルウェアのリアルタイムファイルスキャン機能については、「マルウェア (ウイルスを含む)」という明確な実体のある脅威をスキャンし排除するという、ある種シンプルな仕組みが一般に広く認知されています。

そのため、多くの場合セキュリティや IT に対する知識を持たない一般のユーザーであっても、端末にファイルスキャンが可能なアンチマルウェア製品をインストールする必要性を認識しており、場合によってはライセンス料を支払い、有料のアンチマルウェア製品を購入していることすら珍しくありません。

一方で、AMSI は非常に強力な保護機能を提供する仕組みであり、多くの悪意ある攻撃者にとっては非常に邪魔な存在であるにも関わらず、その保護の仕組みやセキュリティ上の利点についてはあまり認知されていないためか、一般のユーザーにはファイルスキャン機能と比較して軽視されており、安易に無効化されてしまう傾向があると感じています。

AMSI のようなセキュリティ機能はユーザーの可用性とのトレードオフ

[*3] A part of Anti-Virus: https://techbookfest.org/product/iFrVq6PX0PPJhivrGzhi32

により提供されていることもあり、一般のユーザーにとっては単なる邪魔な機能としか認識されていない場合がある点は否定できません。

しかし、AMSI のような強力なセキュリティ機能について、リスクに対する検討が不十分なまま機能が無効化され、本来防げたはずの脅威による侵害を許してしまう事例を少しでも減らすためにも、本書が AMSI の利点や必要性について少しでも多くのユーザーに理解してもらう一助となることを願っています。

なお、本書は AMSI をテーマに執筆しましたが、今後もさまざまな AntiVirus の仕組みについて紹介していきたいと考えています。

もし本書を通して AntiVirus ソフトウェアについて関心を持ってくれる方がいましたら、ぜひ次回作にもご期待ください。

改めて、本書をお読みいただき誠にありがとうございました。

著者紹介

かしわば (@kash1064) です。

本業はセキュリティ製品ベンダーのテクニカルサポートエンジニアで、これまでに 2 つのセキュリティ製品ベンダーにて、Windows や Linux 向けの複数の AntiVirus ソフトウェアのトラブルシューティングやデバッグを行ってきました。

情報セキュリティに強い関心があり、特にリバースエンジニアリングやフォレンジックについて日々学習を続けています。

また、CISSP や OSCP などの情報セキュリティ関連資格も保有しています。

普段は趣味で Capture The Flag (CTF) と呼ばれるセキュリティコンテストに 0nePadding というチームで参加しており、特に Windows プログラムのデバッグやシステムダンプファイルの解析を好んで行っています。

A Part of AntiVirus 2

公開コードで学ぶ Windows Antimalware Scan Interface(AMSI)

2025 年 11 月 15 日　初版第 1 刷　発行

著　者　かしわば (@kash1064)