Process Mockingjay: Echoing RWX In Userland To Achieve Code Execution

securityjoes.com/post/process-mockingjay-echoing-rwx-in-userland-to-achieve-code-execution

Security Joes June 27, 2023

- Security Joes
-
  - Jun 27
  - 14 min read

Our research team is committed to continuously identifying potential security vulnerabilities and techniques that threat actors may exploit to bypass existing security controls. In this blog post, our team is detailing on a comprehensive research specifically focused on process injection techniques utilized by attackers to deceive robust security products integrated into the security stack, such as EDRs and XDRs.

Throughout the blog post, we will delve into various process injection techniques employed by attackers to bypass security controls and gain unauthorized access. We will address the challenges presented by EDRs and XDRs, emphasizing the importance of understanding and mitigating the risks associated with process injection. Additionally, we will share insights gained from our research, including the development of our own process injection technique. We will highlight the capabilities and implications of this technique, shedding light on its effectiveness, potential impact and detection opportunities for defenders.

By definition, process injection refers to the different methods employed to inject malicious code into the memory space of a process. This technique enables attackers to hide the injected code and evade detection. To accomplish this in a Windows environment, attackers rely on a combination of Windows APIs, each serving a specific purpose and following a specific order in the injection process.

Security solutions are well aware of these techniques and have developed mechanisms to detect and block such operations based on the patterns executed in the operating system during an infection. To monitor such behaviors, EDR (Endpoint Detection and Response) software often sets hooks on these Windows APIs within the memory space of every launched process. These hooks intercept and capture the parameters passed to these functions, enabling the EDR to identify potentially malicious actions associated with these Windows APIs.

Considering that EDR systems commonly target several of the Windows APIs required to perform the process injections techniques publicly documented, our research aimed to *discover alternative methods to dynamically execute code within the memory space of Windows processes, without relying on the monitored Windows APIs.*

We explored trusted Windows libraries that already contain sections with default protections set as RWX (Read-Write-Execute). By misusing these libraries, we were able to successfully inject code into various processes and eliminate the need to execute several Windows APIs usually monitored by security solutions. This approach reduces the likelihood of detection by defense software, as our application does not directly invoke Windows APIs typically associated with process injection techniques.

We call this technique "Mockingjay" as it is rather smoother than other techniques and requires smaller number of steps to achieve. The injection is executed without space allocation, setting permissions or even starting a thread. The uniqueness of this technique is that it requires a vulnerable DLL and copying code to the right section. As the copy echoes back in the read/write/execute, we considered the name Mockingjay suitable.

The article in nutshell:

```
[1] Our research team has recently developed and tested a new process injection
technique called "Mockingjay".
[2] This technique has demonstrated its capability to successfully inject code into
Windows processes even in the presence of security software like EDRs.
[3] Our unique approach, which involves leveraging a vulnerable DLL and copying code
to the appropriate section, allowed us to inject code without memory allocation,
permission setting, or even starting a thread in the targeted process.
```

Security Joes is a multi-layered incident response company strategically located in nine different time-zones worldwide, providing a follow-the-sun methodology to respond to any incident remotely. Security Joes' clients are protected against this threat.

Contact us at response@securityjoes.com for more information about our services and technologies and get additional recommendations to protect yourself against this kind of attack vector.

Process Injection

Process injection is a well-known technique utilized by malicious software and attackers to insert and execute code within the memory space of a process. This injection can occur either on the same process performing the operation (self-injection) or on an external process. In the case of injecting an external process, attackers typically target legitimate ones, such as running applications or system processes, aiming to achieve unauthorized access, manipulate the process's behavior, or conceal the injected code from security tools and defenders.

To inject and execute code in memory, whether within the same process or a remote process, attackers employ a combination of Windows APIs that serve distinct purposes within the injection logic. The number of function calls and the specific Windows APIs employed can vary depending on the technical foundation of the chosen code injection method.

Over the years, multiple methods have been developed to achieve code injection and execution within the memory space of Windows processes. As reference, in the following table, the most common Process Injection methods are described and compared based on the Windows API calls required to successfully implement them.

Name	Description	Required Windows APIs
Self Injection	Technique commonly found in malware packers. This technique does not impact any external process; rather, the process executing the injection is the same process that receives the injected payload.	• VirtualAlloc, LocalAlloc, GlobalAlloc • VirtualProtect
Classic DLL Injection	Technique used to inject a malicious DLL into the memory space of another process. In this case, the malicious sample must first identify the specific process it intends to target, allocate a portion of memory within it and create a thread to start the execution of the malicious DLL from disk.	• VirtualAllocEx • WriteProcessMemory • LoadLibrary • CreateRemoteThread, NtCreateThreadEx or RtlCreateUserThread
PE Injection	Technique that maps an entire Portable Executable (PE) file into the memory space of a running process. It allocates a new memory section within the target process, which will serve as the destination for the payload. The contents of the payload are then dynamically mapped onto this memory section using its relocation descriptors and the absolute address of the section, imitating the functionality of the Windows' Loader.	• VirtualAllocEx • WriteProcessMemory • CreateRemoteThread

Process Hollowing / Run PE	In this technique, the original code and resources of the target process are replaced or removed, leaving behind only the bare process framework. The hollowed process then becomes a host for the injected malicious code, allowing it to execute under the guise of a legitimate process.	• CreateProcess • NtUnmapViewOfSection • VirtualAllocEx • WriteProcessMemory • SetThreadContext • ResumeThread
Thread Execution Hijacking	Technique used to gain control of the execution flow within a process by redirecting the execution of a target thread to arbitrary code. It allows an attacker to manipulate the behavior of a running process without creating a new process or modifying the underlying code.	• OpenThread • SuspendThread • VirtualAllocEx • WriteProcessMemory • SetThreadContext
Mapping Injection	By utilizing *NtMapViewOfSection*, the malicious code is mapped into the target process from an existing section, controlled by the attacker. This approach eliminates the requirement to explicitly allocate RWX sections and avoids the need for separate payload content copying. The malicious code indirectly becomes part of the target process's memory space, allowing it to execute within the context of a genuine module.	• OpenProcess, CreateProcess • NtCreateSection • NtMapViewOfSection • RtlCreateUserThread
APC Injection and Atombombing	This technique manipulates the Asynchronous Procedure Call (APC) mechanism in the Windows operating system to inject and execute malicious code in a target process.	• OpenThread • QueueUserAPC

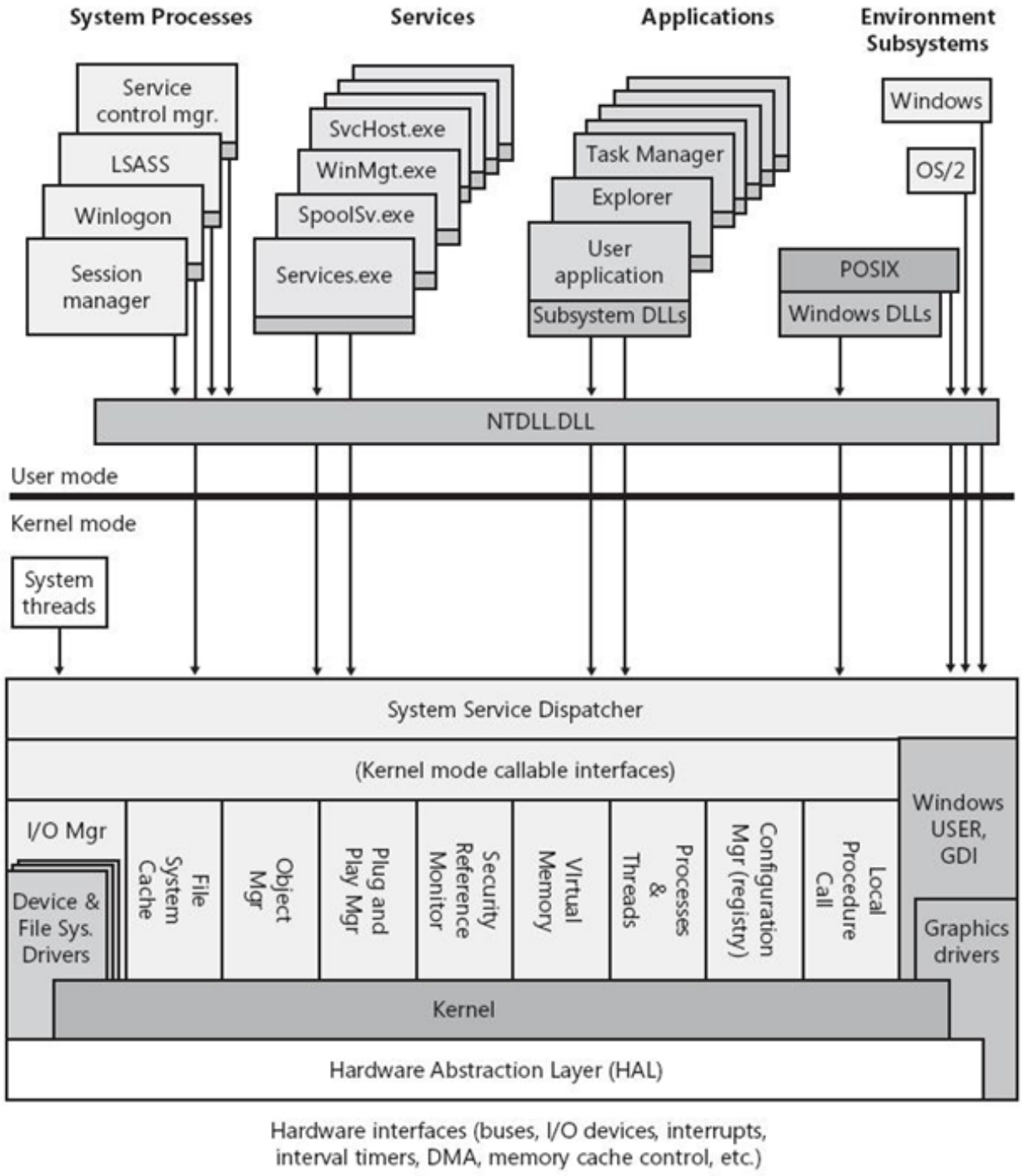
Process Doppelganging	Technique used in malware development to disguise malicious processes by creating a process with a legitimate appearance. It involves utilizing transactional NTFS (TxF) and Windows process loading mechanisms to create a new process that looks like an existing, legitimate process, but runs malicious code instead.	• CreateTransaction • CreateFileTransacted • RollbackTransaction) • NtCreateProcessEx

As described previously, each of these injection techniques requires a set of specific Windows APIs, which generate characteristic patterns that can be leveraged by defenders and security software for detection and mitigation purposes.

Windows APIs Monitoring

To enable effective analysis, detection, and mitigation of suspicious behaviors, Endpoint Detection and Response (EDR) software often utilizes specific hooks on select Windows APIs within the memory space of every process launched on a device. The exact Windows APIs monitored may vary depending on the EDR vendor, but they typically include the most commonly used functions associated with malicious techniques. By implementing these hooks, EDRs can intercept and capture the parameters passed to these functions. This capability allows the EDR to run the required checks during the normal execution of any program and identify potential malicious actions associated to the usage of these Windows APIs and respond accordingly.

When an application is launched, the EDR receives a notification about the new process creation and attaches its own dynamic library. Once executed, the EDR modifies specific functions within the in-memory copy of **NTDLL.DLL** by altering the byte code. EDRs prioritize hooking **NTDLL.DLL** due to its role as the intermediary layer between user applications and the Windows kernel, as depicted in the accompanying image.



As an example, in the code below, we can observe this kind of modifications by comparing the in-memory copy of the *NtProtectVirtualMemory* function in the **NTDLL.DLL** executed by a process running in a system with a commercial EDR solution installed and the copy of such function on-disk. In this case, the EDR replaced the original instruction *mov eax, 0x50*(**in red**) with a *jmp* instruction (**in blue**). This *jmp* instruction performs an unconditional jump to a memory location where inspections for *NtProtectVirtualMemory* are performed by the EDR. If any malicious activity is detected during the inspection, the EDR promptly halts the process execution.

```
Original Function On-Disk:
----------------------

mov     r10, rcx
mov     eax, 50h
test    byte ptr [0x7FFE0h], 1
jne     0x17e76540ea5
syscall
ret
```

```
EDR Hooked Function In-Memory:
----------------------

mov     r10, rcx
jmp     0x7ffaeadea621
test    byte ptr [0x7FFE0h], 1
jne     0x17e76540ea5
syscall
ret
```

Meet the Mockingjay!

Acknowledging the fact that EDR systems often monitor system calls commonly used for code injection into a process's memory, *our objective was to find a way to minimize reliance on these API calls or reduce their usage as much as possible*. This approach aimed to mitigate the risk of detection by defensive software, as our application would execute its injection logic without invoking those monitored system calls.

Upon analyzing various publicly available techniques for injecting code into the memory space of Windows processes, we observed that two fundamental operations were consistently present. These operations involved allocating memory space and setting the protections of this memory section to Read-Write-Execute (RWX), enabling the injected code to be written and executed seamlessly. However, both of these operations were performed using the Syscalls *NtWriteVirtualMemory* and *NtProtectVirtualMemory*, which are closely monitored by EDRs.

Considering the significance of these Syscalls and the monitoring hooks implemented by EDRs on such functions, we decided to take a different approach to inject and execute our code in memory. Our strategy involved searching for pre-existing PE files within the Windows OS that contained default RWX sections in their structure. By leveraging these existing sections, we could eliminate the need to allocate space and set protections on separate sections.

To accomplish this, we divided the research into two distinct stages:

- To identify a vulnerable DLL that possessed a default Read-Write-Execute (RWX) memory section.

- To implement the process injection technique abusing the RWX memory section already present in the DLL previously found.

Each of these stages and their technical details are presented in the following subsections.

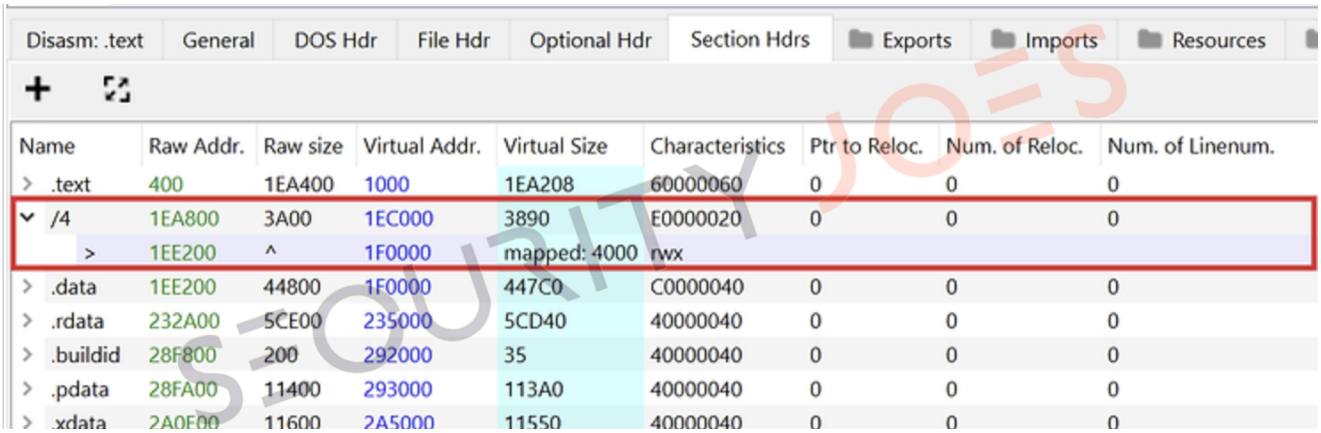
Finding the Vulnerable DLL

In our pursuit of injecting code into pre-existing memory sections that possessed Read-Write-Execute (RWX) permissions, we embarked on a comprehensive exploration of libraries within the Windows OS. To facilitate this endeavor, we developed a dedicated tool that systematically traversed the entire file system, meticulously searching for DLLs that exhibited the specific characteristic of default RWX sections.

Through this systematic exploration, our application thoroughly examined each DLL encountered, evaluating their memory sections to determine if any default RWX sections were present. This approach enabled us to identify potential candidates that could serve as suitable targets for code injection without triggering the attention of defensive software.

As depicted in the aforementioned figure, our comprehensive search across the file system successfully led us to the discovery of the DLL *msys-2.0.dll (***in green**), inside Visual Studio 2022 Community.

Remarkably, this DLL possesses a *default RWX section* that could potentially be exploited to load malicious code. The validity of this finding was substantiated when we opened the DLL using PE-Bear, as shown in the figure below.



With a generous 16 KB of available RWX space, this DLL presents an ideal location for injecting and executing our code. By leveraging this pre-existing RWX section, we can take advantage of the inherent memory protections it offers, effectively bypassing any functions that may have already been hooked by EDRs. This approach not only circumvents the limitations imposed by userland hooks but also establishes a robust and reliable environment for our injection technique.

Injection Mechanisms

After identifying the vulnerable DLL that contains a default Read-Write-Execute (RWX) section on disk, we conducted several tests to explore two different methods that could leverage this misconfiguration to execute code in memory. Our objective was to validate whether this RWX section in the DLL could be utilized to minimize the number of Windows APIs required for injecting code into the same process's memory space, as well as injecting code into the memory space of other processes. Both methods exploit the presence of the RWX section in the vulnerable DLL to optimize code injection, thereby increasing the attack's efficiency and potentially evading detection. The details of each method are outlined below.

Self Injection & EDR Unhooking

In this approach, our custom application loaded the vulnerable DLL directly using the Windows API *LoadLibraryW* and *GetModuleInformation* . The RWX section within the DLL was utilized to store and execute the injected code. The code snippet below illustrates this process:

```
int main(int argc, char *argv[])
{
    // Load the vulnerable DLL
    HMODULE hDll = ::LoadLibraryW(L"path_to_vulnerable_dll");

    if (hDll == nullptr) {
        // fail
    }

    MODULEINFO moduleInfo;
    if (!::GetModuleInformation(
      ::GetCurrentProcess(),
      hDll,
      &moduleInfo,
      sizeof(MODULEINFO))
      ) {
        // fail
    }

    // Access the default RWX section (Vulnerable DLL address + offset)
    LPVOID rwxSectionAddr = (LPVOID)((PBYTE)moduleInfo.lpBaseOfDll +
RWX_SECTION_OFFSET);

    // Write the injected code to the RWX section
    WriteCodeToSection(rwxSectionAddr , injectedCode);

    // Execute the injected code
    ExecuteCodeFromSection(rwxSectionAddr);
```

By directly loading the vulnerable DLL into the memory space of our custom application, we gain direct access to the default RWX section. This allows us to write and execute our desired code without the need for additional memory allocation or permission setting.

After calculating the address of the RWX section, we created a structure called *SectionDescriptor* to store both the initial and final addresses of the RWX section. By encapsulating the starting and ending addresses within this structure, we ensure easy access and efficient management of the RWX section throughout the process.

```
// Create SectionDescriptor structure
SectionDescriptor descriptor = SectionDescriptor {
    rwxSectionAddr,
    (LPVOID)((PBYTE)rwxSectionAddr + RWX_SECTION_SIZE)
};
```

Since the purpose of this PoC is solely to demonstrate the usage of misconfigured RWX sections, we opted to implement the Hell's Gate EDR unhooking technique. Hell's Gate technique involves creating a system call stub within the main executable. It searches for

and extracts the system call numbers from a clean **NTDLL.DLL** module. These extracted numbers are then passed to the system call stub, enabling the execution of the desired system call. Hell's Gate is a reliable method, but it relies on having an unaltered version of **NTDLL.DLL** for accurate system call number retrieval, which involves loading a pristine copy of **NTDLL.DLL** directly from disk and extracting the desired syscall numbers by parsing the PE file. While this approach may not be ideal for Red Team scenarios, none of the conducted tests detected this behavior by any EDR.

After loading the vulnerable RWX section into the memory space of our custom application, our next step involved generating the necessary stubs within it. To achieve this, we implemented the following code snippet.

```
LPVOID createSyscallStub(SectionDescriptor &descriptor, LPVOID testLocation, uint32_t
syscallNumber)
{
   BYTE stub[] = {
        0x49, 0x89, 0xca,                          // mov r10, rcx
        0xb8, 0x00, 0x00, 0x00, 0x00,              // mov eax, 0x0
        0xFF, 0x25, 0x00, 0x00, 0x00, 0x00,        // jmp qword ptr [rip]
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00
    };
    ULONG stubSize = sizeof(stub);

    if (((PBYTE)descriptor.nextStubAddr + stubSize) > descriptor.endSectionAddr) {
        // fail
    }

    memcpy((uint32_t *)&stub[4], &syscallNumber, sizeof(uint32_t));
    memcpy((PVOID *)&stub[14], &testLocation, sizeof(PVOID));

    auto stubAddr = descriptor.nextStubAddr;
    memcpy((PBYTE *)stubAddr, &stub, stubSize);

    printf("[!] Stub created at 0x%p\n", stubAddr);

    descriptor.nextStubAddr = ((PBYTE)descriptor.nextStubAddr) + stubSize;

    return stubAddr;
}
```

It starts by checking if there is enough space available to accommodate a new stub. If sufficient space is present, the code proceeds to copy the syscall numberfrom the untouched NTDLL.DLL and the address of the test instruction following the jmp added by the EDR. For the sake of brevity, the specific data points obtained through parsing the **NTDLL.DLL** are not included here.

By employing this strategy of leveraging the loaded **NTDLL.DLL** file from disk, we dynamically re-constructed the original syscall number for each of the functions modified by the EDR. Consequently, all the EDR hooks present within the process were effectively eliminated.

The complete process to achieve this can be described as follows:

1. Custom application loads vulnerable DLL using *LoadLibraryW*.
2. Location of the RWX section is resolved using the base address of the DLL and the offset of section.
3. A clean copy of NTDLL.DLL is loaded from the disk, and the system call numbers for the desired syscalls are obtained.
4. The addresses of the test instructions after the jmp added by the EDR are retrieved from the NTDLL.DLL in-memory copy (hooked by the EDR).
5. Using the addresses of the test instructions and the syscall numbers, we assemble our stubs in the RWX area of the vulnerable DLL.
6. When the stub is executed, it prepares the syscall number in the EAX register, as usual, and immediately jumps to the address of the corresponding test instruction for the chosen system call, bypassing the EDR verification step.

After conducting the necessary validations, our method has proven to be a successful solution for injecting and executing code. In this case, we were able to inject our own shellcode into the memory space of our custom application **nightmare.exe**, without relying on Windows APIs such as NtWriteVirtualMemory and NtProtectVirtualMemory. This complete removal of dependency on Windows APIs not only reduces the likelihood of detection but also enhances the effectiveness of the technique. Notably, the injected shellcode successfully removed all EDR inline userland hooks without triggering any detection.

Remote Process Injection

The second method we explored involved leveraging the RWX section in the vulnerable DLL to perform process injection in a remote process. This requires identifying non-malicious binaries that depend on the DLL msys-2.0.dll for their operation.

During our research, we noticed that the msys-2.0.dll library is commonly utilized by applications that require POSIX emulation, such as GNU utilities or applications not originally designed for the Windows environment. We found relevant binaries with these characteristics within the Visual Studio 2022 Community subdirectory.

For our proof of concept, we selected the **ssh.exe** process located within the Visual Studio 2022 Community directory as the target for injecting our payload. To accomplish this, we initiated the **ssh.exe** process as a child process of our custom application using the Windows API CreateProcessW, as presented in the code below. It is worth noting that the arguments passed to the **ssh.exe** binary does not need to reference real machines; it is solely used to trigger the execution logic within the **ssh.exe** binary.

For our proof of concept, we specifically chose the **ssh.exe** process located in the Visual Studio directory to inject our payload. It is important to note that in this injection method, there is no need to explicitly create a thread within the target process, as the process automatically executes the injected code. This inherent behavior makes it challenging for Endpoint Detection and Response (EDR) systems to detect this method.

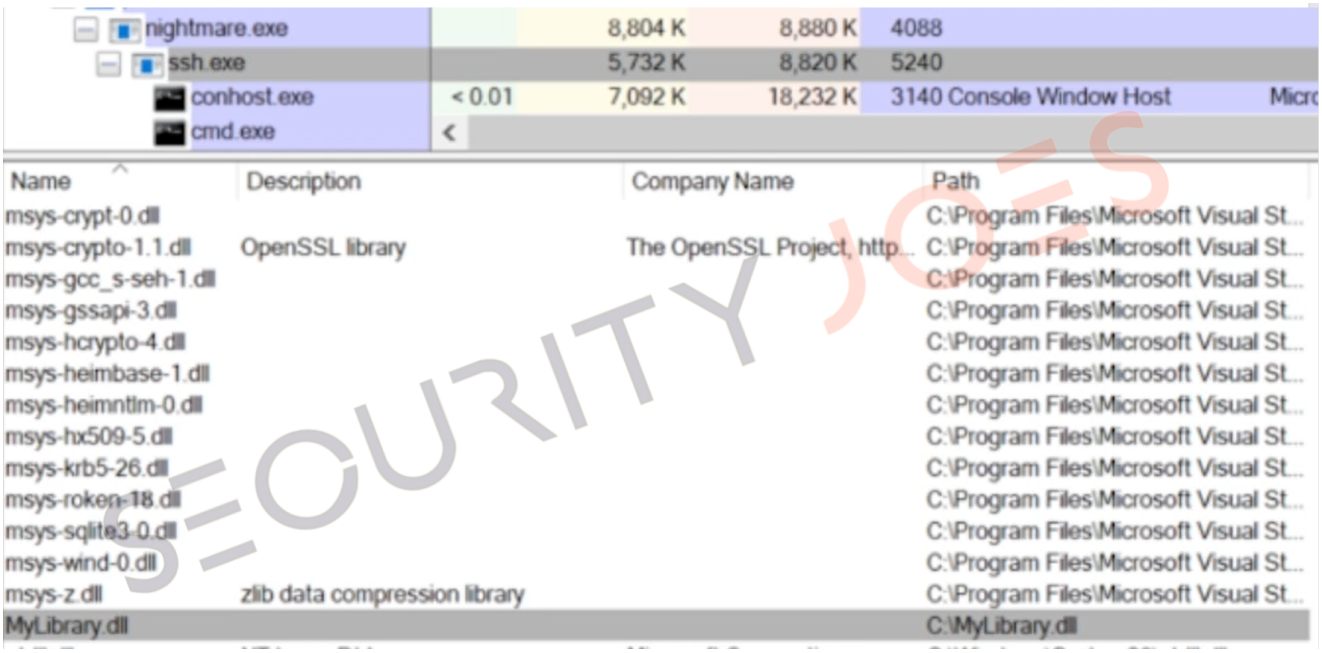
```
LPTSTR command = L"C:\\Program Files\\Microsoft Visual
Studio\\2022\\Community\\Common7\\IDE\\CommonExtensions\\Microsoft\\TeamFoundation\\Te
 Explorer\\Git\\usr\\bin\\ssh.exe";
    LPTSTR args = L"ssh.exe decoy@decoy.dom";
    STARTUPINFOW si;
    PROCESS_INFORMATION pi;
    ZeroMemory(&si, sizeof(si));
    si.cb = sizeof(si);
    ZeroMemory(&pi, sizeof(pi));
    DWORD dwCreationFlags = 0;
    BOOL success = ::CreateProcessW(
        command,
        args,
        nullptr,
        nullptr,
        FALSE,
        dwCreationFlags | DEBUG_ONLY_THIS_PROCESS | DEBUG_PROCESS,
        nullptr,
        nullptr,
        &si,
        &pi);
```

After creating the process and allowing it to execute the provided command line arguments, the next step is to open the process and move the payload directly into the Read-Write-Execute (RWX) section. This can be accomplished using the OpenProcess and WriteProcessMemory Windows APIs, as demonstrated in the code snippet below:

```
/*unsigned char buf[] =
    "\xfc\x48\x81\xe4\xf0\xff\xff\xff\xe8\xd0\x00\x00\x00\x41"
    "\x51\x41\x50\x52\x51\x56\x48\x31\xd2\x65\x48\x8b\x52\x60"
    "\x3e\x48\x8b\x52\x18\x3e\x48\x8b\x52\x20\x3e\x48\x8b\x72"
    "\x50\x3e\x48\x0f\xb7\x4a\x4a\x4d\x31\xc9\x48\x31\xc0\xac"
    "\x3c\x61\x7c\x02\x2c\x20\x41\xc1\xc9\x0d\x41\x01\xc1\xe2"
    "\xed\x52\x41\x51\x3e\x48\x8b\x52\x20\x3e\x8b\x42\x3c\x48"
    "\x01\xd0\x3e\x8b\x80\x88\x00\x00\x00\x48\x85\xc0\x74\x6f"
    "\x48\x01\xd0\x50\x3e\x8b\x48\x18\x3e\x44\x8b\x40\x20\x49"
    "\x01\xd0\xe3\x5c\x48\xff\xc9\x3e\x41\x8b\x34\x88\x48\x01"
    "\xd6\x4d\x31\xc9\x48\x31\xc0\xac\x41\xc1\xc9\x0d\x41\x01"
    "\xc1\x38\xe0\x75\xf1\x3e\x4c\x03\x4c\x24\x08\x45\x39\xd1"
    "\x75\xd6\x58\x3e\x44\x8b\x40\x24\x49\x01\xd0\x66\x3e\x41"
    "\x8b\x0c\x48\x3e\x44\x8b\x40\x1c\x49\x01\xd0\x3e\x41\x8b"
    "\x04\x88\x48\x01\xd0\x41\x58\x41\x58\x5e\x59\x5a\x41\x58"
    "\x41\x59\x41\x5a\x48\x83\xec\x20\x41\x52\xff\xe0\x58\x41"
    "\x59\x5a\x3e\x48\x8b\x12\xe9\x49\xff\xff\xff\x5d\x49\xc7"
    "\xc1\x00\x00\x00\x00\x3e\x48\x8d\x95\xfe\x00\x00\x00\x3e"
    "\x4c\x8d\x85\x0b\x01\x00\x00\x48\x31\xc9\x41\xba\x45\x83"
    "\x56\x07\xff\xd5\x48\x31\xc9\x41\xba\xf0\xb5\xa2\x56\xff"
    "\xd5\x48\x65\x6c\x6c\x6f\x2c\x20\x4a\x4f\x45\x53\x21\x00"
    "\x41\x6c\x65\x72\x74\x00";

    HANDLE hProcess = OpenProcess(PROCESS_ALL_ACCESS, FALSE,
    pi.dwProcessId);
    if (hProcess == nullptr) {
        printf("[x] ReadProcessMemory failed with status 0x%x\n",
        ::GetLastError());
        return 1;
    }

    if (!DebugActiveProcess(4236)) {
        printf("[x] DebugActiveProcess failed with status 0x%x\n",
        ::GetLastError());
        ::CloseHandle(hProcess);  return 1;
    }  SIZE_T bytesWritten = 0;

    if (!WriteProcessMemory(hProcess, (LPVOID)0x21022D120, buf,
    sizeof(buf), &bytesWritten)) {
        printf("[x] WriteProcessMemory failed with status 0x%x\n",
        ::GetLastError());
    }
```

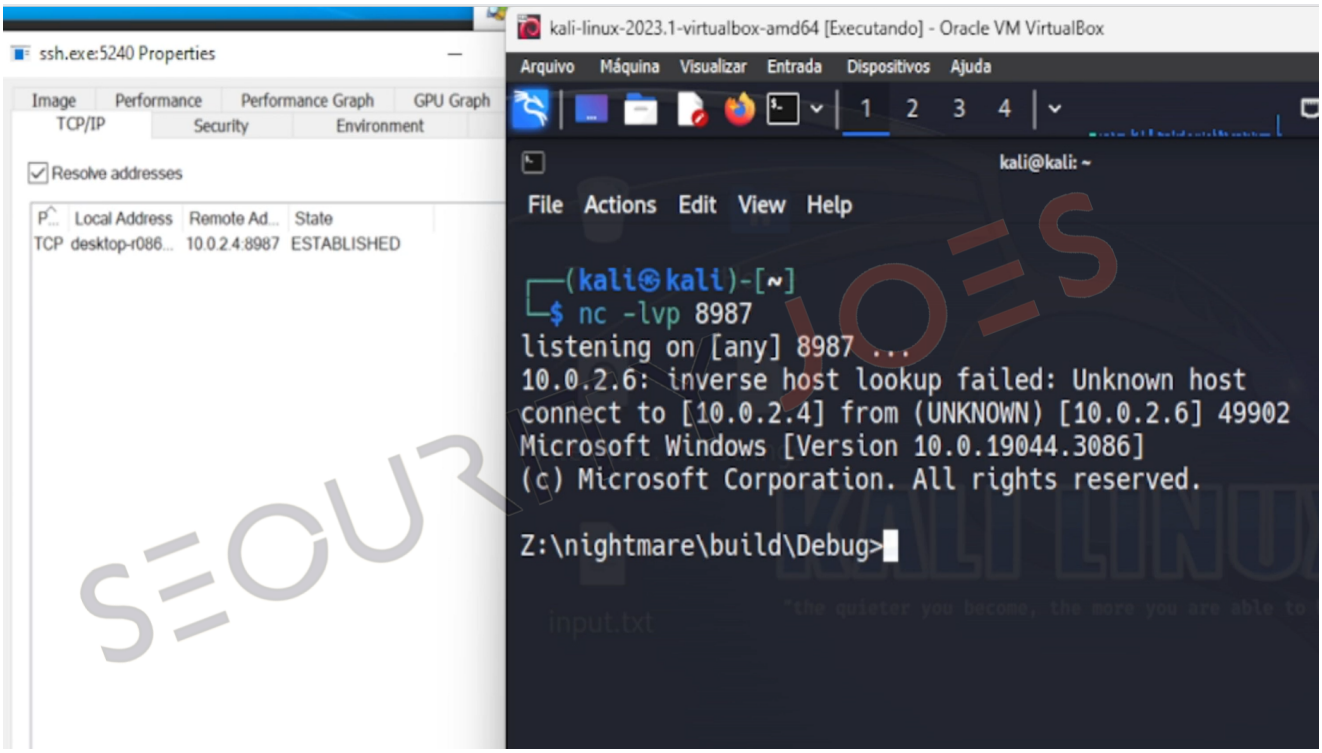
Once our code has been successfully written into the Read-Write-Execute (RWX) section of the ssh.exe process's memory, we can patiently wait for the execution flow of ssh.exe to naturally reach the RWX section and execute our injected code. It is important to note that in this specific case, the targeted DLL does not employ Address Space Layout Randomization (ASLR), which means the address of the RWX section remains consistent and can be hardcoded in the payload. Thus, there is no need to dynamically resolve the address of the targeted section during runtime.

For the purposes of the proof of concept, the injected shellcode in the ssh.exe process attempts to load an additional DLL named MyLibrary.dll into memory. This DLL was designed to establish a back connect shell session with an external machine. However, it's important to note that the functionality of this DLL can be customized to execute any desired operation based on the developer's intention.



After conducting extensive tests, our method has proven to be a highly successful solution for injecting and executing code in a remote process that uses the DLL msys-2.0.dll. In this case, we were able to inject our own code into the memory space of the ssh.exe process without being detected by the EDR.

The uniqueness of this technique lies in the fact that there is no need to allocate memory, set permissions or create a new thread within the target process to initiate the execution of our injected code. This differentiation sets this strategy apart from other existing techniques and makes it challenging for Endpoint Detection and Response (EDR) systems to detect this method.



Notably the injected shellcode seamlessly loaded an additional DLL and created a remote shell within the ssh.exe process, as shown in the previous image, all while evading detection mechanisms. This successful demonstration highlights the effectiveness and discretion of our method in achieving the desired outcome without raising suspicion.

The complete process to achieve this can be summarized as follows:

1. Custom application is executed.
2. Trusted application (ssh.exe) using DLL msys-2.0.dll is launched as a child process.
3. Custom application opens a handle to the target process (ssh.exe).
4. Code to be injected is copied into the RWX section of msys-2.0.dll.
5. Trusted application executes the injected code during its normal execution flow.
6. Additional DLL MyLibrary.dll is loaded by the shellcode injected in the RWX section.
7. Back connect shell session is stablished.

PoC Video

Watch Video At: https://youtu.be/155OXwnnAyw

For more videos, visit our YouTube Channel.

Security Implications

As mentioned earlier in this research paper, attackers may utilize this technique to circumvent detection by EDRs or antivirus software by evading userland hooks and injecting malicious code into the process space of trusted software. In the proof of concept demonstrated in this blog post, the vulnerable DLL that was exploited for injection was msys-2.0.dll. However, it is important to note that there are potentially numerous other DLLs that share similar characteristics, making the detection of this behavior even more challenging.

To effectively counteract such attacks, security solutions need to employ a comprehensive and proactive approach that goes beyond static monitoring of specific DLLs or system calls. Behavioral analysis, anomaly detection, and machine learning techniques can enhance the ability to identify process injection techniques and detect malicious activities within the memory space of trusted processes.

Detection Opportunities

We provide the following set of ideas to the infosec community to help hunting this threat on your environments, feel free to test them and improve them if required.

1. Look for GNU utilities launched by suspicious or uncommon processes.
2. Look for network connections to non-standard ports from processes such as **ssh.exe** or any other GNU utility.

3. Maintain a database of DLLs that exhibit such characteristics and identify any loading attempts made by non-legitimate processes.
4. Employ reputation-based systems that assign trust levels to DLLs, considering factors such as the source of the DLL, digital certificates, historical behavior, and the characteristics of its sections.

Conclusions

This research has provided valuable insights into the utilization of legitimate DLLs with Read-Write-Execute (RWX) sections as an effective method for evading userland hooks and injecting code into remote processes. By leveraging trusted libraries with these attributes, threat actors can bypass the need to allocate RWX memory, set permissions, or even create new threads in the target process. These findings underscore the critical importance of implementing comprehensive defense strategies to combat such advanced evasion techniques. Organizations should employ dynamic analysis techniques to detect and analyze runtime behaviors, leverage behavioral analysis to identify anomalous activities, utilize signature-based detection for known malicious patterns, implement reputation-based systems to flag suspicious files or activities, and establish robust memory protection mechanisms to prevent unauthorized code execution.