New Threat: A Deep Dive Into the Zergeca Botnet

blog.xlab.qianxin.com/a-deep-dive-into-the-zergeca-botnet

Alex.Turing

June 19, 2024

Background

On May 20, 2024, while everyone was happily celebrating the holiday, the tireless `XLab CTIA(Cyber Threat Insight Analysis) system` captured a suspicious ELF file around 2 PM, located at `/usr/bin/geomi`. This file was packed with a modified UPX, had a magic number of 0x30219101, and was uploaded from Russia to VirusTotal, where it was not detected as malicious by any antivirus engine.

Later that evening at 10 PM, another geomi file using the same UPX magic was uploaded to VT from Germany. **The suspicious file path, modified UPX, and multi-country uploads** caught our attention. After analysis, we confirmed that this is a **botnet** implemented in Golang. Given that its C2 used the string "ootheca," reminiscent of the swarming Zerg in StarCraft, we named it `Zergeca`.

Functionally, Zergeca is not just a typical DDoS botnet; besides supporting six different attack methods, it also has capabilities for proxying, scanning, self-upgrading, persistence, file transfer, reverse shell, and collecting sensitive device information. From a network communication perspective, Zergeca also has the following unique features:

- Supports multiple DNS resolution methods, **prioritizing DOH** for C2 resolution.
- Uses the uncommon Smux library for C2 communication protocol, encrypted via XOR.

During the investigation of Zergeca's infrastructure, we found that its C2 IP address, **84.54.51.82**, has been serving at least two Mirai botnets since September 2023. We speculate that the author behind Zergeca accumulated experience operating the Mirai botnets before creating Zergeca.

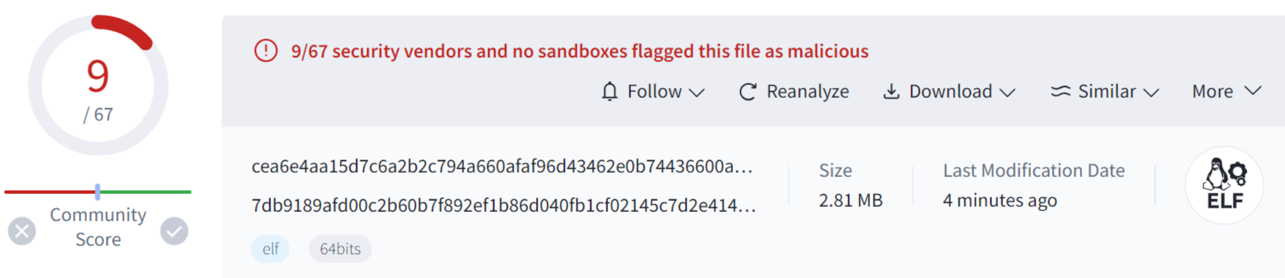
On June 10, `XLab command tracking system` captured a vector 7 DDoS command that the current samples did not support, indicating that Zergeca's author is actively developing and updating, with new samples yet to be discovered. Our persistence paid off when we captured a new sample on the 19th that supports the vector 7. Currently, the detection rates for Zergeca samples and C2 are very low. Considering Zergeca's potential threat in DDoS attacks, we have decided to release this article to share our findings with the community.

Sample & C2 Detection

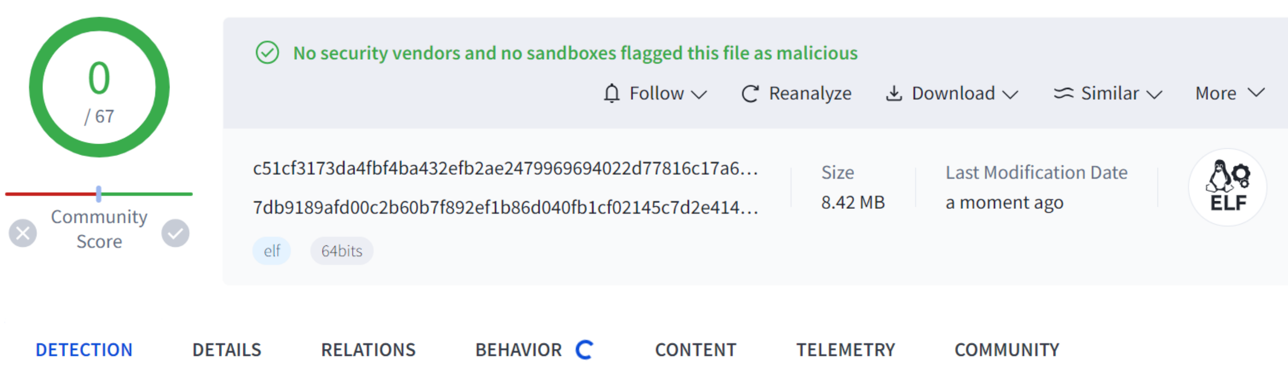
From the sample perspective, we captured a total of 5 Zergeca samples. While their functions are nearly identical, there is a significant discrepancy in their detection rates. How can this anomaly be explained? Most antivirus vendors have categorized the sample `23ca4ab1518ff76f5037ea12f367a469` as **Generic Malware**. We speculate that the detection of Zergeca by antivirus software is based on file hash. Therefore, as long as the hash changes, the detection effectiveness diminishes.

MD5	Detection	First Seen	Telemetry
23ca4ab1518ff76f5037ea12f367a469	28/64	2024.05.20	Russian
9d96646d4fa35b6f7c19a3b5d3846777	0/67	2024.05.20	Germany
d78d1c57fb6e818eb1b52417e262ce59	1/67	2024.05.22	China
604397198f291fa5eb2c363f7c93c9bf	1/66	2024.06.11	France
60f23acebf0ddb51a3176d0750055cf8	0/67	2024.06.18	France

To verify our hypothesis, we appended the 4-byte string "Xlab" to the end of the file 23ca4ab1518ff76f5037ea12f367a469 and re-uploaded it to VirusTotal. The detection rate changed to 9/67, partially confirming our speculation.



Additionally, the current detection is based on the packed samples, after unpacking, the detection rate drops to 0.



From the Domain Perspective, the four samples share two C2 domains that were created on the same day. The samples prioritize using DOH (DNS over HTTPS) for C2 resolution, which obscures the relationship between the samples and the C2 domains to some extent. Because of this, `VirusTotal couldn't even associate the C2 domains with the samples`, resulting in a naturally low detection rate.

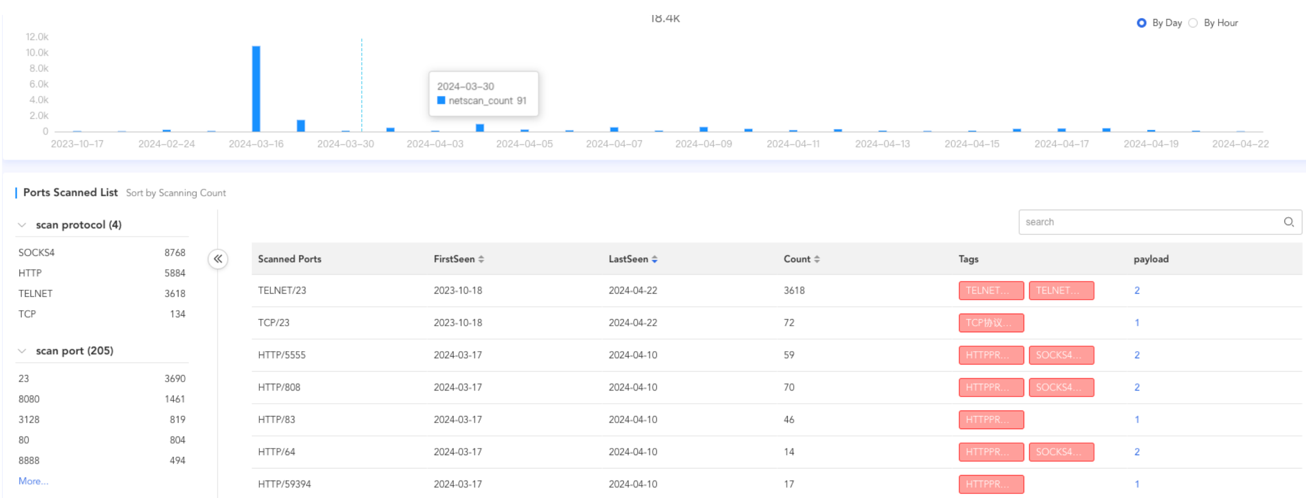
Domain	Detection	Create date
ootheca.pw	1/93	2024.04.28
ootheca.top	1/93	2024.04.28

Profile of 84.54.51.82

The two C2 servers of Zergeca point to the same IP address, 84.54.51.82. According to our data, this IP has been in use since September 2023, serving a variety of roles. During this period, it has acted as a Scanner, Downloader, Mirai botnet C2, and Zergeca botnet C2.

Scanner

Starting from September 18, 2023, scanning activities commenced, primarily targeting protocols such as Telnet, HTTP, and socks4. The main ports scanned include`23, 8080, 3128, 80, and 8888`.



Mirai Downloader&C2

From September and October 2023 to April 2024, 84.54.51.82 was primarily used as the Loader IP and Downloader IP for the Mirai botnet.

- 2023.09 - 2023.10, it was used as the Loader and Downloader IP to implant the following related samples.

```
 #Downloader

http://84.54[.51.82/jaws
http://84.54[.51.82/bin
http://84.54[.51.82/596a96cc7bf9108cd896f33c44aedc8a/db0fa4b8db0333367e9bda3ab68
b8042.x86
http://84.54[.51.82/596a96cc7bf9108cd896f33c44aedc8a/db0fa4b8db0333367e9bda3ab68
b8042.spc
http://84.54[.51.82/596a96cc7bf9108cd896f33c44aedc8a/db0fa4b8db0333367e9bda3ab68
b8042.sh4
http://84.54[.51.82/596a96cc7bf9108cd896f33c44aedc8a/db0fa4b8db0333367e9bda3ab68
b8042.ppc
http://84.54[.51.82/596a96cc7bf9108cd896f33c44aedc8a/db0fa4b8db0333367e9bda3ab68
b8042.mpsl
http://84.54[.51.82/596a96cc7bf9108cd896f33c44aedc8a/db0fa4b8db0333367e9bda3ab68
b8042.mips
http://84.54[.51.82/596a96cc7bf9108cd896f33c44aedc8a/db0fa4b8db0333367e9bda3ab68
b8042.m68k
http://84.54[.51.82/596a96cc7bf9108cd896f33c44aedc8a/db0fa4b8db0333367e9bda3ab68
b8042.i686
http://84.54[.51.82/596a96cc7bf9108cd896f33c44aedc8a/db0fa4b8db0333367e9bda3ab68
b8042.arm7
http://84.54[.51.82/596a96cc7bf9108cd896f33c44aedc8a/db0fa4b8db0333367e9bda3ab68
b8042.arm6
http://84.54[.51.82/596a96cc7bf9108cd896f33c44aedc8a/db0fa4b8db0333367e9bda3ab68
b8042.arm5
http://84.54[.51.82/596a96cc7bf9108cd896f33c44aedc8a/db0fa4b8db0333367e9bda3ab68
b8042.arm
http://84.54[.51.82/596a96cc7bf9108cd896f33c44aedc8a/db0fa4b8db0333367e9bda3ab68
b8042.arc

#CC

mirai://bot.hamsterrace.space:59666
```

- 2024.04, it was used as the Loader IP to implant the following related samples.

```
#Downloader
http://145.239[.108.150/Fantazy.sh
http://145.239[.108.150/Fantazy/Fantazy.arm5
http://145.239[.108.150/Fantazy/Fantazy.arm6
http://145.239[.108.150/Fantazy/Fantazy.mpsl
http://145.239[.108.150/Fantazy/Fantazy.sh4
http://145.239[.108.150/Please-Subscribe-To-My-YT-Channel-VegaSec/1isequal9.x86
http://145.239[.108.150/cache

# CC

mirai://145.239.108.150:63645
```

Zergeca C2

Starting from April 29, 2024, 84.54.51.82 began being used as the C2 server for Zergeca. The relevant C2 domains and their resolution records are as follows:

Resolution Records

Domain Name	FirstSeen	LastSeen	Count	Tags
ootheca.pw	2024-04-29 22:23:32	2024-06-13 19:13:26	9120	Zergeca...
ootheca.top	2024-05-23 04:33:06	2024-06-13 19:12:57	9235	Zergeca...
bot.hamsterrace.space	2023-09-18 04:34:25	2023-10-12 19:23:06	153	僵尸网络 Mirai cc

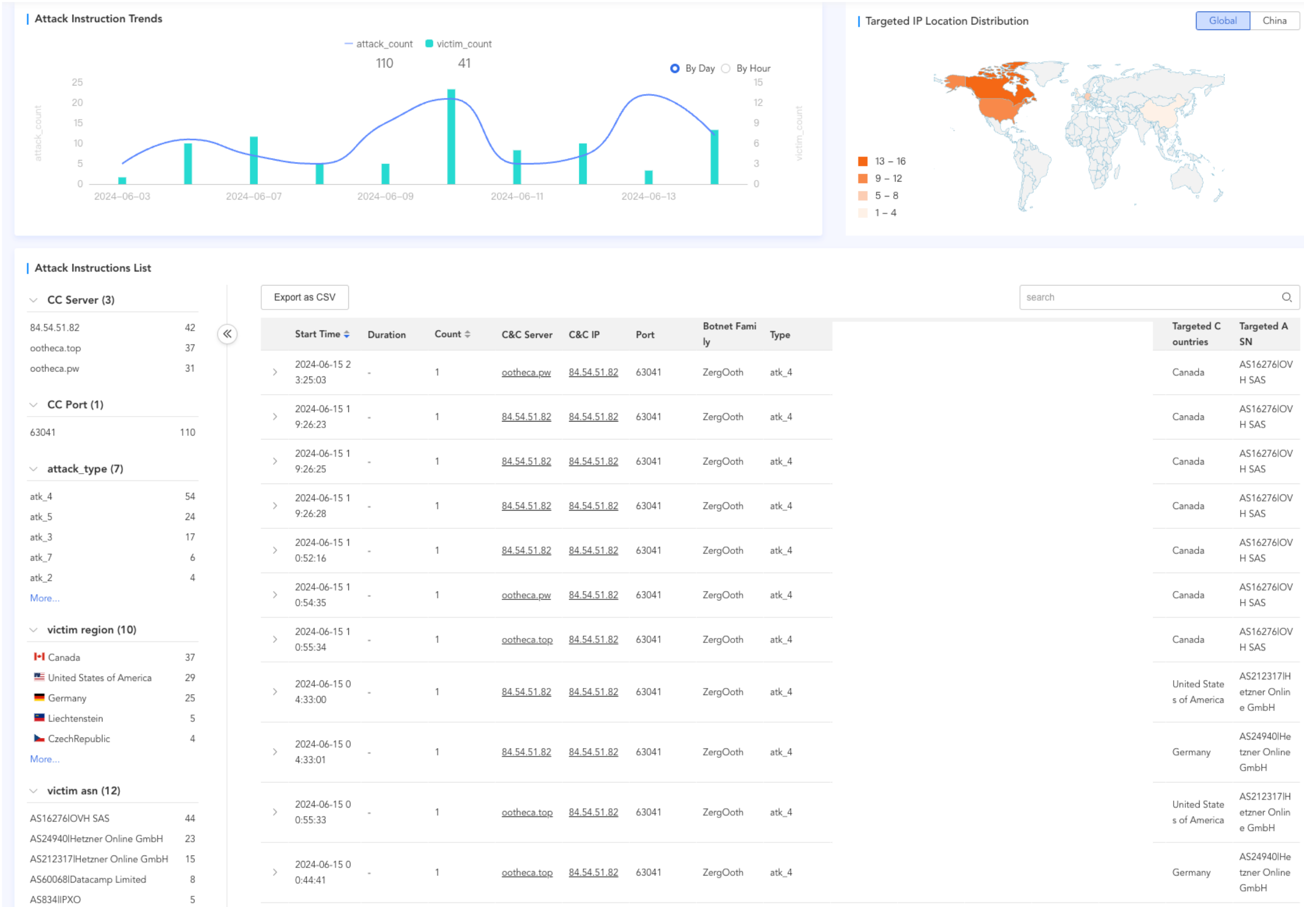
Exploits

In our observation, the primary methods used by 84.54.51.82 to propagate samples are Telnet weak passwords and certain known vulnerabilities. The relevant vulnerability identifiers are as follows:

```
Telnet Weak Password
CVE-2022-35733
CVE-2018-10562
CVE-2018-10561
CVE-2017-17215
CVE-2016-20016
```

DDoS Statistics

From early to mid-June 2024, the Zergeca botnet primarily targeted regions such as **Canada, the United States, and Germany**. The main type of attack was ackFlood (atk_4), with victims distributed across multiple countries and different ASNs.



Reverse Analysis

The four Zergeca samples in our observation are all designed for the x86-64 CPU architecture and target the Linux platform. The presence of strings like "android," "darwin," and "windows" in the samples, along with Golang's inherent cross-platform capabilities, suggests that the author may eventually aim for full platform support.

This article focuses on the earliest captured sample for detailed analysis. The sample is packed with UPX and has a magic number of 0x30219101. For this type of modified UPX packer, simply changing the magic back to the standard "UPX!" allows for unpacking with the command `upx -d`.

```
MD5:23ca4ab1518ff76f5037ea12f367a469
Mgaic:ELF 64-bit LSB executable, x86-64, version 1 (SYSV), statically linked,
corrupted section header size
Packer: UPX
Version:0.0.01c
```

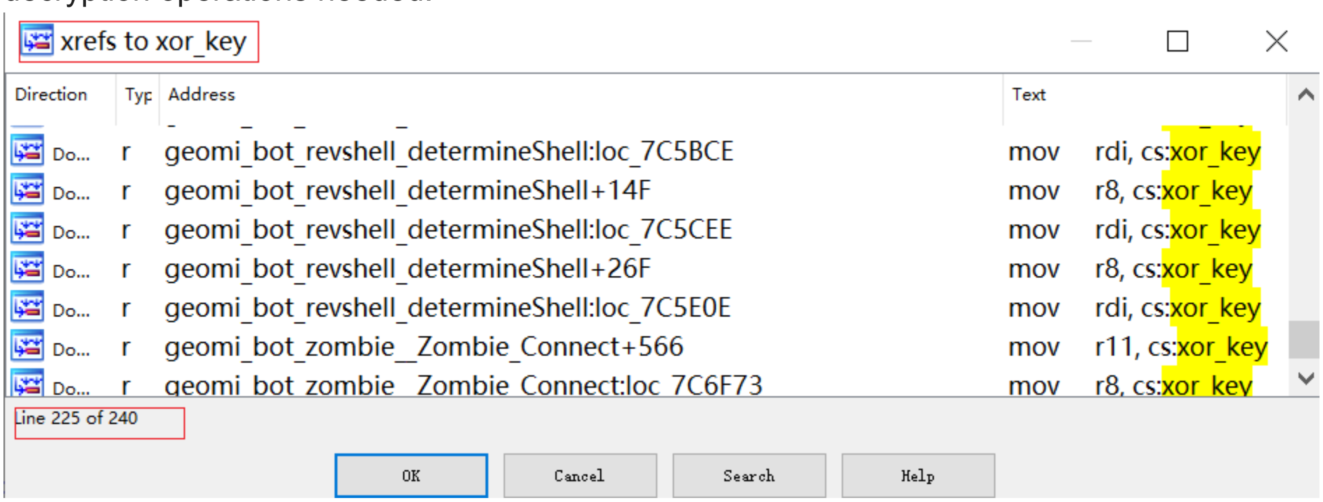
After unpacking, it becomes evident that Zergeca is a botnet implemented in Go language. The symbols are not obfuscated, making reverse analysis relatively straightforward.

```
geomi_bot_silivaccine_Sibling();
geomi_bot_silivaccine_Start();
geomi_bot_persistence_service();
geomi_bot_proxy_Start();
geomi_bot_zombie_New();
```

The figure above shows a code snippet of the main_main function. Functionally, it can be broken down into four distinct modules. The persistence and proxy modules are self-explanatory, with the former ensuring persistence and the latter handling proxying. The silivaccine module is used to remove competing malware, ensuring exclusive control over the device. The most crucial module is zombie, which implements the full botnet functionality. It reports sensitive information from the compromised device to the C2 and awaits commands from the C2, supporting six types of DDoS attacks, scanning, reverse shell, and other functions.

0x00: String Decryption

Zergeca uses XOR encryption for many sensitive strings. Using IDA, we found that the XOR key is referenced 240 times across various functions. Each decryption involves two uses of the XOR key: `one for initialization and one for decryption`. So there are 120 decryption operations needed.

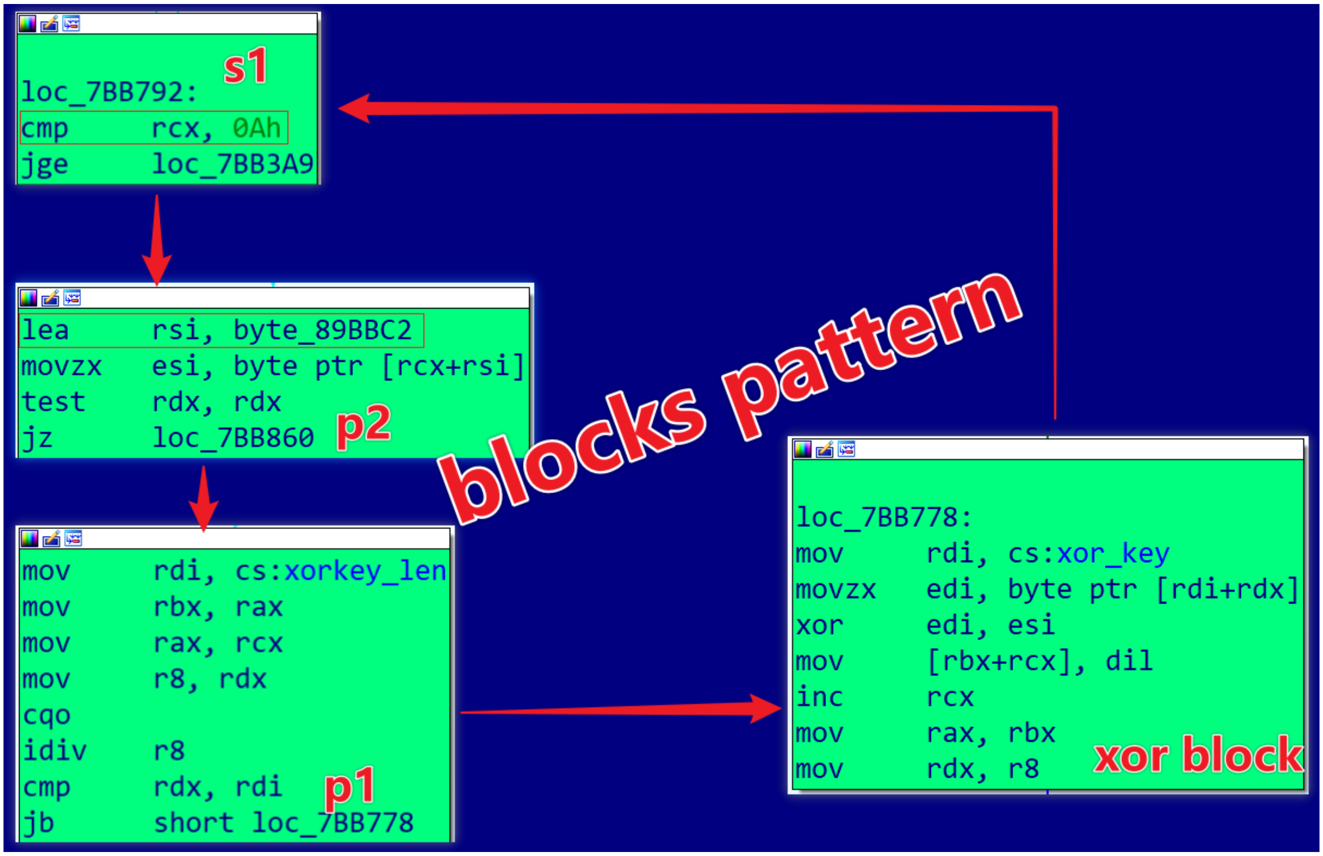


The XOR key is initially set to `EC 22 2B A9 F3 DD DF 1C CD 46 AC 1E`, but only the first six bytes (`EC 22 2B A9 F3 DD`) are used.

```
for ( i = 0LL; i < 5; ++i )
{
  if ( !v11 )
    runtime_panicdivide();
  v13 = v11;
  v14 = i % v11;
  if ( v14 >= xorkey_len )
    runtime_panicIndex();
  *(_BYTE *)(v10 + i) = geomi[i] ^ *((_BYTE *)xor_key + v14);
  v11 = v13;
}
```

Manually decrypting 120 times is impractical. Although the decryption process isn't confined to a single function, CFG analysis revealed a specific pattern in most decryption-related code blocks:

1. The XOR block has one predecessor and one successor.
2. The predecessor block's first instruction is `mov`, with the first operand being an address pointing to the original length of the XOR key.
3. The successor block's first instruction is `cmp`, with the first operand being a number indicating the ciphertext's length.
4. The predecessor block's predecessor's first instruction is `lea`, with the first operand being an address pointing to the ciphertext's starting address.



By identifying these patterns, we can automate the decryption process and restore all encrypted strings efficiently.We implemented IdaPython decryption script in the Appendix with the following results: 111 successful decryptions and 9 mismatches.

```
geomi_bot_silivaccine_init 0x722bc6 matched, ciphertext at 0x895a0f <----> b'kaiten'
geomi_bot_silivaccine_init 0x722cf8 matched, ciphertext at 0x89b8ac <----> b'kdevtmpfsi'
geomi_bot_silivaccine_init 0x722e2a matched, ciphertext at 0x896b33 <----> b'kinsing'
geomi_bot_silivaccine_init 0x722f66 matched, ciphertext at 0x899c71 <----> b'kthreaddi'
geomi_bot_silivaccine_init 0x723098 matched, ciphertext at 0x89b8b6 <----> b'meminitsrv'
geomi_bot_silivaccine_init 0x7231ca matched, ciphertext at 0x895a15 <----> b'minerd'
geomi_bot_silivaccine_init 0x723306 matched, ciphertext at 0x895a1b <----> b'mozi.a'
geomi_bot_silivaccine_init 0x723438 matched, ciphertext at 0x895a21 <----> b'Mozi.a'
geomi_bot_silivaccine_init 0x72356a matched, ciphertext at 0x895a27 <----> b'mozi.m'
geomi_bot_silivaccine_init 0x7236a6 matched, ciphertext at 0x895a2d <----> b'Mozi.m'
geomi_bot_silivaccine_init 0x7237d8 matched, ciphertext at 0x895a33 <----> b'Nbrute'
geomi_bot_silivaccine_init 0x72390a matched, ciphertext at 0x8981a7 <----> b'pdenferd'
geomi_bot_silivaccine_init 0x723a46 matched, ciphertext at 0x894e9b <----> b'srv00'
geomi_bot_silivaccine_init 0x723b78 matched, ciphertext at 0x895a39 <----> b'start_'
geomi_bot_silivaccine_init 0x723caa matched, ciphertext at 0x8981af <----> b'startapp'
```

The 9 mismatched codes are distributed across six functions. Among them, the `packets__Cursor` Read/WriteString functions handle network packet encryption/decryption and can be ignored.

```
gomi_bot_zombie__Zombie_Connect
geomi_common_utils_init_0_func1,
geomi_bot_discovery_Run,
geomi_common_packets__Cursor_WriteString,
geomi_common_packets__Cursor_ReadString,
geomi_common_utils_RandomUserAgent
```

For the remaining four functions, the issue was that the ciphertexts were arrays rather than single entries, causing the pattern match to fail. For example, in the `RandomUserAgent` function, the `user_agent_list` contains 1000 encrypted user agents.

```
data:0000000000C56FA0 user_agent_list dq offset off_C668C0
data:0000000000C56FA0
data:0000000000C56FA8 qword_C56FA8     dq 1000
data:0000000000C56FA8
data:0000000000C56FB0                  dq 1000
```

For such cases, we can use the `manual_decode` function, where the first parameter is the starting address of the ciphertext array and the second parameter is the number of array elements.

```
ey=b"\xEC\x22\x2B\xA9\xF3\xDD"

def manual_decode(base,cnt):
    for i in range(cnt):
        start=idc.get_qword(base)
        addr=idc.get_qword(start+i*16)
        size=idc.get_qword(start+8+i*16)
        buff=idc.get_bytes(addr,size)
        out=bytearray()
        for k,v in enumerate(buff):
            out.append(v ^ key[k%6])
        print(out.decode())

manual_decode(0x000000000C56FA0,1000)  #user agent
manual_decode(0x0000000000C56F80,0xc)  #opennic dns
manual_decode(0x000000000C56C40,2) # c2
```

Decrypted examples include various user agents, OpenNIC DNS server, and C2s.

```
Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:37.0) Gecko/20100101 Firefox/37.0
Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:38.0) Gecko/20100101 Firefox/38.0
Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:39.0) Gecko/20100101 Firefox/39.0
Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:40.0) Gecko/20100101 Firefox/40.0
Mozilla/5.0 AppleWebKit/999.0 (KHTML, like Gecko) Chrome/99.0 Safari/999.0
168.235.111.72
152.53.15.127
194.36.144.87
80.152.203.134
217.160.70.42
178.254.22.166
81.169.136.222
185.232.68.212
207.89.102.10
185.181.61.24
137.220.52.23
51.158.108.203
ootheca.pw:63041
ootheca.top:63041
```

With all strings successfully decrypted, we can now begin reverse-engineering Zergeca's various functionalities.

0x01: Persistence Module

Zergeca achieves persistence on compromised devices by adding a system service `geomi.service`. This service ensures that the Zergeca sample automatically generates a new `geomi` process if the device restarts or the process is terminated.

```
[Unit]
Description=
Requires=network.target
After=network.target
[Service]
PIDFile=/run/geomi.pid
ExecStartPre=/bin/rm -f /run/geomi.pid
ExecStart=/usr/bin/geomi
Restart=always
[Install]
WantedBy=multi-user.target
```

Experiment A

When running the Zergeca sample on a virtual machine and restarting the device, `geomi.service` automatically launches the Zergeca sample. The resulting process named `geomi` had a PID of 897. Terminating this process with `kill -9 897` immediately spawned a new `geomi` process with PID 8460.

```
┌──(root㉿kali)-[/home/kali]
└─# netstat -tpn
Active Internet connections (w/o servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State       PID/Program name
tcp        0      0 192.168.96.129:59744    84.54.51.82:63041       ESTABLISHED 897/geomi

┌──(root㉿kali)-[/home/kali]
└─# kill -9 897

┌──(root㉿kali)-[/home/kali]
└─# netstat -tpn
Active Internet connections (w/o servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State       PID/Program name
tcp        0      0 192.168.96.129:37066    34.117.186.192:443      ESTABLISHED 8460/geomi
tcp        0      1 192.168.96.129:55234    104.16.248.249:443      SYN_SENT    8460/geomi
tcp        0      0 192.168.96.129:47770    104.26.9.44:80          ESTABLISHED 8460/geomi
tcp        0      0 192.168.96.129:53770    172.67.69.226:443       ESTABLISHED 8460/geomi
tcp        0      0 192.168.96.129:59744    84.54.51.82:63041       TIME_WAIT   -
tcp        0      1 192.168.96.129:38616    8.8.4.4:443             SYN_SENT    8460/geomi
tcp        0      0 192.168.96.129:60940    34.117.186.192:80       ESTABLISHED 8460/geomi
```

When network administrators discover a `geomi` process and suspicious traffic on a device, they can attempt the following cleanup steps:

1. Delete `/etc/systemd/system/geomi.service`
2. Delete the sample file referenced by the `ExecStart` parameter
3. Terminate the `geomi` process

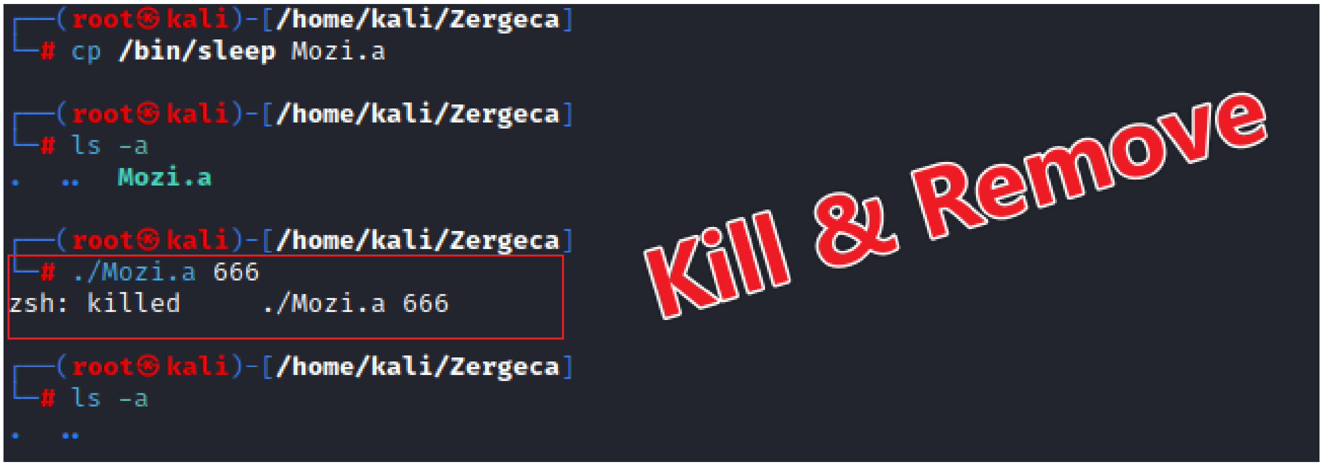
0x2: Silivaccine Module

To monopolize the device, Zergeca includes a list of competitor threats, covering miners, backdoor trojans, botnets, and more. Some familiar names on the list include mozi, kinsing, and various mining pools. Zergeca continuously monitors the system and terminates any process whose name or runtime parameters match those on the list, deleting the corresponding binary files.

Mozi.a	com.ufo.miner	kinsing	kthreaddi
kaiten	srv00	meminitsrv	.javae
solr.sh	monerohash	minexmr	c3pool
crypto-pool.fr	f2pool.com	xmrpool.eu	....

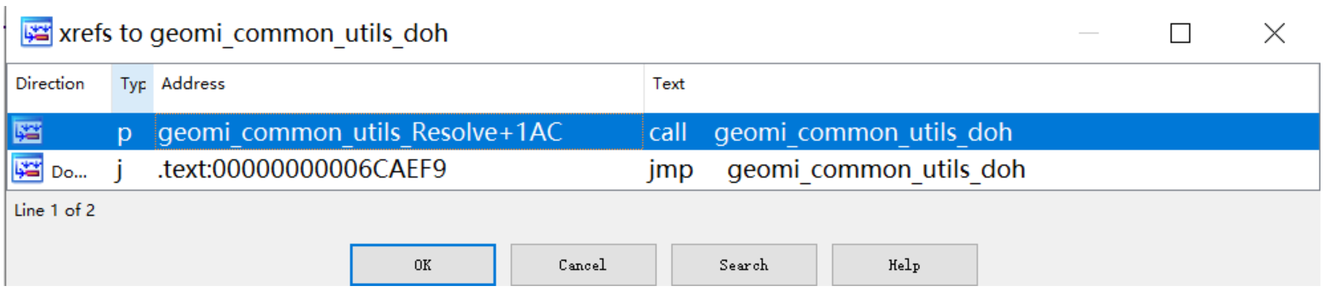
Experiment B

We renamed the system program `/bin/sleep` to `Mozi.a` and ran it. The `Mozi.a` process was killed, and the corresponding binary file was deleted.



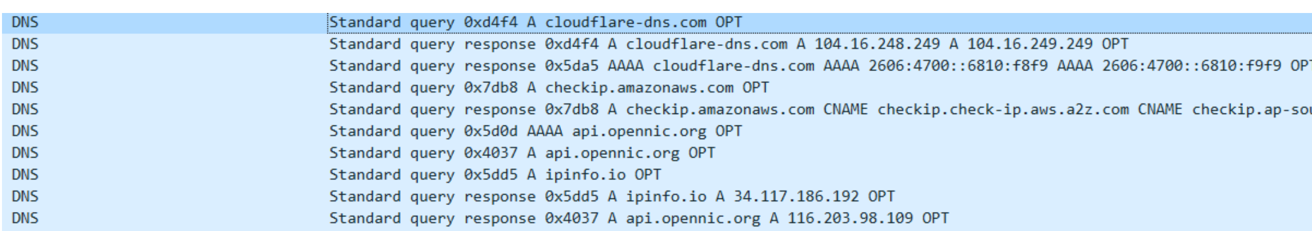
0x3: Zombie Module

Zergeca resolves the C2 IP address using the `geomi_common_utils_Resolve` function, which supports four resolvers: Public DNS, Local DNS, DoH (DNS over HTTPS), and OpenNIC.



Zergeca prioritizes two DoH resolvers, masking C2 domain resolution in DNS traffic.

```
https://cloudflare-dns.com/dns-query
https://dns.google/resolve
```



After obtaining the C2 IP, the bot reports device sensitive information encapsulated in a `DeviceInfo` structure, including details like "country, public IP, OS, user groups, runtime directory, and reachability".

```
struct DeviceInfo
{
Country string
PlucAddress byte[]
MAC string
OS string
ARCH string
Name string
MachineId string
Numcpu uint32
CPUMODEL string
username string
uid string
gid string
Users []string
Uptime time.Duration
PID     uitn32
Path string
checksum []uint8
version string
Reachable bool
}
```

The bot then awaits commands from the C2, processing them with different handlers.

geomi_bot_proxy_Handle	.text	000000000071E400
geomi_bot_filetransfer_Handle	.text	00000000007C57A0
geomi_bot_revshell_Handle	.text	00000000007C5B40
geomi_bot_zombie__Zombie_HandleAttack	.text	00000000007C6340
geomi_bot_zombie_HandleUpdate	.text	00000000007C68A0
geomi_bot_zombie_handleDiscovery	.text	00000000007C6B40

The supported functions are as follows:

ID	Task
0x01	Proxy
0x02	Reverse Shell
0x03	FileTransfer
0x05	Self-update
0xa0	DDoS
0xb0	Stop Discovery
0xb1	Start Discovery

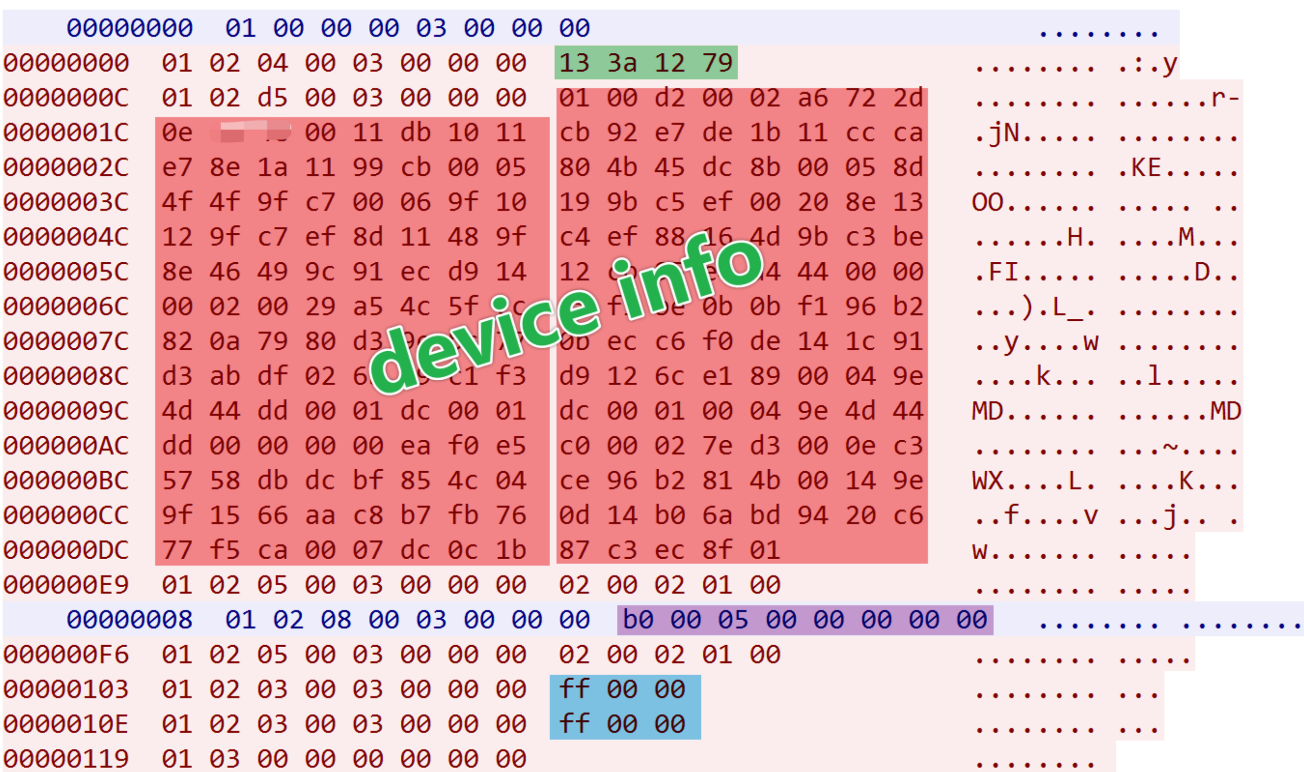
The DDoS functionality supports the following seven attack vectors:

Sub-ID	Attack Vector
1	minecraft
2	httpPPS
3	synFlood
4	ackFlood
5	pushFlood
6	rstFlood
7	pushOVHFlood

Communication Protocol

Zergeca uses smux for Bot-C2 communication. Smux(Simple MUltipleXing) is a Golang multiplexing library that relies on underlying connections like TCP or KCP for reliability and ordering, providing stream-oriented multiplexing. Smux packets feature an 8-byte header: `VERSION(1B) | CMD(1B) | LENGTH(2B) | STREAMID(4B) | DATA(LENGTH)`.

From an analysis perspective, only the `LENGTH` and `DATA` fields are of primary concern. The captured traffic includes various messages such as online status, device information reporting, command 0xb0, and heartbeat messages.



Online Message:

- Length: 0x04 bytes
- Content: Hardcoded `13 3a 12 79`

Device Info Report:

- Length: 0xd5 bytes (varies by device)
- Content (excluding IP): XOR encrypted with key `EC 22 2B A9 F3 DD`
- Decrypted DeviceInfo as follows

```
pos: 0x4 len: 0x2 <----> b'JP'
pos 0x7 len: 4 <----> 45.14.XX.XX
pos: 0xc len: 0x11 <----> b'72:ba:29:e9:b8:08'
pos: 0x1f len: 0x5 <----> b'linux'
pos: 0x26 len: 0x5 <----> b'amd64'
pos: 0x2d len: 0x6 <----> b's22262'
pos: 0x35 len: 0x20 <----> b'b19642a3c672d4f20cbdb5b1569bf98f'
pos: 0x5b len: 0x29 <----> b'Intel(R) Xeon(R) CPU E5-2678 v3 @ 2.50GHz'
pos: 0x86 len: 0x4 <----> b'root'
pos: 0x86 len: 0x4 <----> b'root'
pos: 0xa2 len: 0x2 <----> b'\x92\xf1'
pos: 0xa6 len: 0xe <----> b'/usr/bin/geomi'
pos: 0xb6 len: 0x14 <---->
b'r\xbd>\xcfY\x15[\xd9]\xa4\xe7m\x86\x9f\xbf\x895\xaa\x19\xe8'
pos: 0xcc len: 0x7 <----> b'0.0.01c'
```

Command 0xb0 Message:

- Length: 0x08 bytes
- Function: Stop scanning

Heartbeat Message:

- Length: 0x03 bytes
- Content: `ff 00 00`

Let's take a look at the DDoS-related packets. The format is `cmd (1 byte) + length (2 bytes) + sub_cmd (1 byte) + target_info (length-1)`, where `cmd` is `0xa0`, indicating a DDoS command, and `sub_cmd` is `0x4`, indicating an ACK flood attack. The `target_info` field focuses on the first 4 bytes, which represent the target IP. For example, `1f 06 10 21` corresponds to the IP address 31.6.16.33.

```
00000128  01 02 2d 00 03 00 00 00  a0 00 2a 04 1f 06 10 21   ..-..... ..*....!
00000138  00 00 00 0a df 13 05 9f  dd ec da 0c 18 9a 00 01   ........ ........
00000148  c3 00 00 00 00 00 00 0f  a0 00 00 00 00 00 00 00   ........ ........
00000158  01 00 00 01 f4                                     .....
```

When the Bot receives the aforementioned command, the resulting attack traffic aligns perfectly with our analysis.

No.	Time	Destination	Protocol	Info
1599…	1942.409651	31.6.16.33	TCP	1752 → 38238 [ACK] Seq=1 Ack=1 Win=32847 Len=0
1599…	1942.409699	31.6.16.33	TCP	51880 → 24101 [ACK] Seq=1 Ack=1 Win=1080 Len=0
1599…	1942.409757	31.6.16.33	TCP	30306 → 22145 [ACK] Seq=1 Ack=1 Win=1162 Len=0
1599…	1942.409785	31.6.16.33	TCP	5495 → 16618 [ACK] Seq=1 Ack=1 Win=1137 Len=0
1599…	1942.409805	31.6.16.33	TCP	11323 → 48861 [ACK] Seq=1 Ack=1 Win=473 Len=0
1599…	1942.409823	31.6.16.33	TCP	59824 → 20129 [ACK] Seq=1 Ack=1 Win=122 Len=0
1599…	1942.409854	31.6.16.33	TCP	41203 → 26981 [ACK] Seq=1 Ack=1 Win=1030 Len=0
1599…	1942.409873	31.6.16.33	TCP	14417 → 49039 [ACK] Seq=1 Ack=1 Win=1686 Len=0
1599…	1942.409924	31.6.16.33	TCP	24953 → 38610 [ACK] Seq=1 Ack=1 Win=53 Len=0
1599…	1942.409950	31.6.16.33	TCP	38929 → 35511 [ACK] Seq=1 Ack=1 Win=1162 Len=0
1599…	1942.409968	31.6.16.33	TCP	31741 → 22459 [ACK] Seq=1 Ack=1 Win=131 Len=0
1599…	1942.409985	31.6.16.33	TCP	56153 → 11803 [ACK] Seq=1 Ack=1 Win=467 Len=0
1599…	1942.410012	31.6.16.33	TCP	36535 → 17117 [ACK] Seq=1 Ack=1 Win=5011 Len=0
1599…	1942.410022	31.6.16.33	TCP	41579 → 17043 [ACK] Seq=1 Ack=1 Win=1773 Len=0
1599…	1942.410035	31.6.16.33	TCP	5563 → 59525 [ACK] Seq=1 Ack=1 Win=2060 Len=0
1599…	1942.410050	31.6.16.33	TCP	48399 → 25736 [ACK] Seq=1 Ack=1 Win=521 Len=0
1599…	1942.410056	31.6.16.33	TCP	59117 → 41806 [ACK] Seq=1 Ack=1 Win=3898 Len=0

Experiment C

Based on our network protocol analysis, we implemented a fake C2 to control the Bot and observe its behavior upon receiving different commands. In this experiment, we sent the Bot a `0xb1` command, which is to "start scanning."

```
000000EC  01 02 05 00 03 00 00 00  02 00 02 00 00              ........ .....
    00000008  01 02 08 00 03 00 00 00  b1 00 05 00 00 00 00 00    ........ ........
000000F9  01 02 05 00 03 00 00 00  02 00 02 00 01              ........ .....
00000106  01 02 03 00 03 00 00 00  ff 00 00                    ........ ...
```

Upon receiving this command, the Bot immediately began scanning 16 ports on randomly generated IP addresses.

Destination	Protocol	Destination Port	Info
49.47.[redacted]	TCP	37215	34688 → 37215 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM TSval=1733349856 TSecr=0 WS=128
49.47.[redacted]	TCP	5522	36796 → 5522 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM TSval=1733349857 TSecr=0 WS=128
49.47.[redacted]	TCP	22	55680 → 22 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM TSval=1733349857 TSecr=0 WS=128
49.47.[redacted]	TCP	222	52336 → 222 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM TSval=1733349858 TSecr=0 WS=128
49.47.[redacted]	TCP	2222	50834 → 2222 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM TSval=1733349858 TSecr=0 WS=128
49.47.[redacted]	TCP	2333	48962 → 2333 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM TSval=1733349862 TSecr=0 WS=128
49.47.[redacted]	TCP	2375	37970 → 2375 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM TSval=1733349864 TSecr=0 WS=128
49.47.[redacted]	TCP	2376	35178 → 2376 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM TSval=1733349865 TSecr=0 WS=128
49.47.[redacted]	TCP	2275	34046 → 2275 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM TSval=1733349866 TSecr=0 WS=128
49.47.[redacted]	TCP	9922	43600 → 9922 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM TSval=1733349866 TSecr=0 WS=128
49.47.[redacted]	TCP	23	37312 → 23 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM TSval=1733349867 TSecr=0 WS=128
49.47.[redacted]	TCP	2323	41322 → 2323 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM TSval=1733349868 TSecr=0 WS=128
49.47.[redacted]	TCP	2735	32860 → 2735 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM TSval=1733349868 TSecr=0 WS=128
49.47.[redacted]	TCP	2380	55208 → 2380 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM TSval=1733349887 TSecr=0 WS=128
49.47.[redacted]	TCP	8291	45342 → 8291 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM TSval=1733349888 TSecr=0 WS=128
49.47.[redacted]	TCP	2736	56984 → 2736 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM TSval=1733349888 TSecr=0 WS=128

Summary

Through reverse analysis, we gained initial insights into Zergeca's author. The built-in competitor list shows familiarity with common Linux threats. Techniques like modified UPX packing, XOR encryption for sensitive strings, and using DoH to hide C2 resolution demonstrate a strong understanding of evasion tactics. Implementing the network protocol with Smux showcases their development skills. Given this combination of operational knowledge, evasion tactics, and development expertise, encountering more of their work in the future would not be surprising.

This is our basic intelligence of Zergeca. We welcome unique insights from other companies, such as Init Access. And readers can contact us on Twitter for more details.

IOC

Sample

```
23ca4ab1518ff76f5037ea12f367a469
9d96646d4fa35b6f7c19a3b5d3846777
d78d1c57fb6e818eb1b52417e262ce59
604397198f291fa5eb2c363f7c93c9bf

f68139904e127b95249ffd40dfeedd21
d7b5d45628aa22726fd09d452a9e5717
6ac8958d3f542274596bd5206ae8fa96

pathced with "xlab" at the end of file
980cad4be8bf20fea5c34c5195013200

sample captured on 2024.06.19, support ddos vector 7
60f23acebf0ddb51a3176d0750055cf8
```

Domain

```
ootheca.pw
ootheca.top
bot.hamsterrace.space
```

IP

```
84.54.51.82     The Netherlands|None|None       AS202685|Aggros Operations Ltd.
```

Appendix

IdaPython Script

```
# Test script, only for 23ca4ab1518ff76f5037ea12f367a469
# Modidy keyaddr,sizeaddr in your case

def decode(buf):
    key=b"\xEC\x22\x2B\xA9\xF3\xDD"
    out=bytearray()
    for i in range(len(buf)):
        out.append(buf[i]^key[i%6])
    return out

count=0
notcount=0
failedfunc=[]
successedfunc=[]

keyaddr=0x0000000000C56FC0
sizeaddr=0x0000000000C56FC8

refs=XrefsTo(keyaddr, flags=0)
for ref in refs:
    f_blocks = idaapi.FlowChart(idaapi.get_func(ref.frm), flags=idaapi.FC_PREDS)
    for blk in f_blocks:
        if blk.start_ea!=ref.frm:
            continue
        if len(list(blk.preds()))!=1 and len(list(blk.succs()))!=1:
            continue
        predblk=list(blk.preds())[0]
        succsblk=list(blk.succs())[0]

        if idc.get_operand_value(predblk.start_ea,1)!=sizeaddr:

            continue
        if idc.get_operand_type(succsblk.start_ea,1)!=0x5:
            print(idc.get_func_name(ref.frm),hex(ref.frm),"not matched")
            notcount+=1
            failedfunc.append(idc.get_func_name(ref.frm))
            continue
        ppredblk=list(predblk.preds())
        if len(ppredblk)!=1:
            continue
        addr=idc.get_operand_value(ppredblk[0].start_ea,1)
        size=idc.get_operand_value(succsblk.start_ea,1)
        buf=idc.get_bytes(addr,size)
        out=decode(buf)
        count+=1
        print(idc.get_func_name(ref.frm),hex(ppredblk[0].start_ea),"matched,
ciphertext at", hex(addr), "<---->",bytes(out))
        successedfunc.append(idc.get_func_name(ref.frm))

print("\n--------------------Statistic--------------------")
print(f'Success:{count},Failed:{notcount}\n')
print("---------Success Function---------")
```

```
print(set(successedfunc),'\n')
print("---------Failed Function---------")
print(set(failedfunc),'\n')
```