Binary JSON : Insecurity in Implementing Serialization

[One Step Ahead]

Aditya K Sood
http://www.secniche.org

Abstract

The article describes serialization attack based on binary Javascript Object Notation. The infection vector encompasses the manipulation of objects like arrays. The binary JSON enhances the working functionality of Javascript request-response mechanism by speeding up processing. The binary JSON is designed to handle serializing operations in an efficient manner. The J SON provides centralized concept of designing server request handling. All the data is undertaken as a string in serialization concept. The point of talk is the serialization base is vulnerable to web attacks. It is possible through object infection. The binary JSON comprise of JSON fused with binary concepts for string handling mechanism. We will delve into the infection spots of the binary JSON and enumerate the impact of wrong serialization on web applications. A saying :

Success in warfare is gained by carefully accommodating ourselves to the enemy's purpose.

The security aspect work on this paradigm.

Why Binary JSON?

The binary JSON is the result of fusibility of binary concepts in the Java script object notation. The binary JSON concept sticks complex serialization. The serialization is a methodology to transform data from raw object notation to a well formed output having defined constraints. The notation used in object designing is based on the Javascript standards. The server dynamically processes the request and serializes the data in the execution path. This dynamic nature is provided for better functionality It also ensures that transformation takes place on the defined standards. It is a functional peculiarity of Javascript object notation is to convert not to transform. The form of data is changed but not converted to the requisite layout. The binary JSON handles the complex request comprised of variables of high length or nested statements.

1. The BISON stands for the Binary interchange and structure object notation. It is a protocol used for the structural transformation of objects through HTTP undertaking the concept of XMLHTTP request ie Ajax standard. The base comes from the JSON framework and it strictly adheres to the web2.0.

2. The BISON entirely relates to the concept of binary transformation which includes integers, strings, arrays, multibyte values etc. The BISON is termed as Binary JSON. Remember the BISON favors only the multibyte values that are converted to Little Endian format prior to have some functions defined on it. It means before designing any object for request, the binary variables or multibyte values should be in little Endian format.

3. The concept of NULL Byte is highly critical because for an effective usage , the strings must end with null byte ['\0'].If this is not undertaken then web applications are prone to insecurity and exploitation realm. It suggests that string operations should be handled in definite manner to avoid complexities.

4. The integers should be used in a signed way for the applications to work exactly in the same manner across the platforms. The testing against unsigned integers will cause the application to work in a stringent way and create a lot of problems. Further a specification has been made about the two's compliment use of the integers. The integers must inherit the properties of two's compliment.

5. The concept of Magic numbers is the sole point of the message format which is sent through the HTTP i.e. every single message must start with the magic number.

The BISON is used mostly where the serialization is complex and the transformation of objects is very critical. This sets an element of flexibility in the serialization and deserialization of the variables placed in objects.

Insecure Vectors in BISON

The specification and requirements have already been analyzed. Before digging deeper I would like to discuss about the security weakness or infection vector that dismantles the working of the BISON.

1. The concept of NULL byte ending has been introduced. It is a prime point of concern because if a string is subjected to null byte , it will truncate the string at the first prime byte leaving the other bytes in a random layout. It is a contrary part of web application with serialization and creates a lot of problems in the context in which it is applied. To overcome this issue, the byte should inherit the escape element. It makes the concept

very complex and harder to implement because a coder has to take care of everything in mind to overcome the issue. It is a very critical way and can lead to some web application problems.

2. The server is able to process to the strings that are passed as ['']. The element passed in quotes treated as an object by the arrays. The request is treated and handled in a same manner as the normal request like HTML/XML by the server. It means if a link is embedded with definitive parameters it get processed and serialized with the application structure. The data is dynamically serialized. The object is infected very easily because server treats object as a string. The arrays can be annotated or nested. The most stringent point is that there is no specific limit on the size of nested arrays. So, this means as the nested index increases so as the infection .It makes the web application rogue and deeply infected All this leads to infected serialization.

Array: [temp, ret, true,"Hello World", [2, 3, 4, 5], 4]]
The array can be infected by placing string with rogue request in the above model.

3. The designing of object is the core of all object notation language. The objects are processed and then executed through the serialization. The infection through the objects is possible. The object has a member element and a value attached to it. Objects can be nested. It makes the infection possible at a primary level ,secondary level and so on. The nested object concept works fine but can be manipulated very easily. The depth of nested object is not defined, so you can think how much the vector gets traversed deep. The object member value is stored as null articulated string. The number of member objects is of limited size. The interoperability issues are undertaken by enforcing the naming conventions. The object simulated as

```
Object: {
 NestedObject: {
            AnotherNestedObject: {
                        name: "Blue",
                        age: 32,   }
},
```

4. The base of this specific protocol is HTTP context, so it is possible to implement methods like get and post. It becomes convenient for the attacker or user to extract the BISON document from the server directly by crafting a specific request. This functionality is inherited from the basic HTTP protocol. That's why the fusibility is applicable.

POST /service/ HTTP/1.1
Host: www.host.com
Content-Type: application/bison
Content-Length: 410
[BMF Message]

HTTP/1.1 200 OK
Content-Type: application/bison
Content-Length: 321
[BMF Message]

5. The data can be easily encoded or decoded into hexadecimal format. It favors the attacker to design an infected request and serialize it after encoding it in hexadecimal. In this

manner, it becomes hard for the security elements to undertake an issue efficiently. You can look at the desired layout as:

{ PI: 3.14159265, }

This is a simple declaration of element and when it is converted to hexadecimal as by the inbuilt BISON request as:

00000 70 77 6C 3B 2B 2A 7A 73 2A 37 04 39 73 6A

That's how the conversion occurs through the inbuilt request.

So we have undertaken many of the insecure parameters and explored the infections. . Now we will look into the request designing in BISON which is inherited from the Javascript object notation. It will clarify the BISON approach.

Evaluating Strings And Serialization

```
var bison = new Bison();
function stringToBison(str) {
var obj;
try {
obj = eval("(" + str + ")");
return bison.serialize(obj);
        } catch (e) {
alert("Error in expression: " + e.message);
return false;
        }
}
```

Request Generation Based On XMLHTTP Object:

```
XmlHttpRequest = {

getInstance: function() {
        var instance = false;
        if (typeof XMLHttpRequest != "undefined") {
        instance = new XMLHttpRequest();
 }

if (!instance) {
    try {
        instance = new ActiveXObject("Msxml2.XMLHTTP");
    } catch(e) {
        try {
            instance = new ActiveXObject("Microsoft.XMLHTTP");
        } catch(e) {
            instance = false;
        }
                                    }
        }
```

```
return instance;
            }
}
```

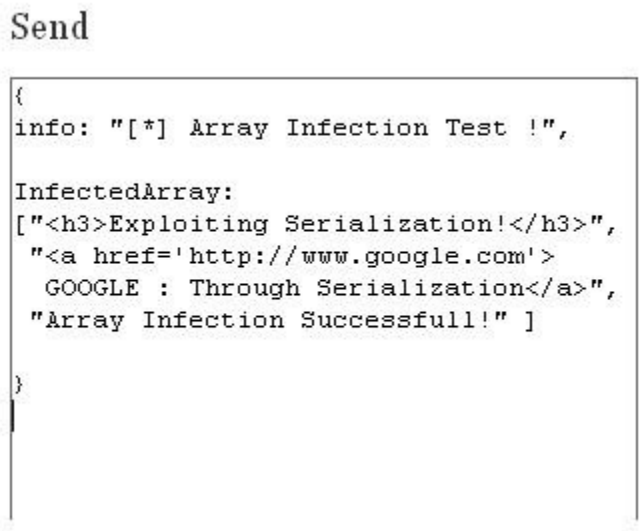
Sending Data To The Server:

```
function send() {

var bisonStr = stringToBison(document.getElementById("send-box").value);
if (bisonStr)
{
var xmlHttp = XmlHttpRequest.getInstance();
xmlHttp.open("POST", "./bisonserver.php", true);
xmlHttp.onreadystatechange = function() {
      if (xmlHttp.readyState == 4) {
            var obj = bison.deserialize(xmlHttp.responseText);
            document.getElementById("receive-box").innerHTML = dump(obj);
             }
            }
            xmlHttp.send(bisonStr);
             }
            }
```

You can look clearly how exactly the request is generated and sent to the server for processing and serialization of data. Now we look at the way of how attackers exploit the serialization concept based on the BISON through array infection, object infection etc. The point is to look at the injection of rogue parameters. Lets analyze the infection and the serialization effect:

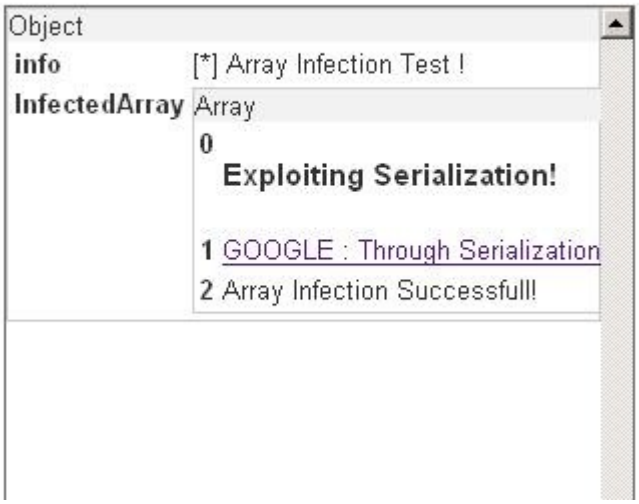
Demonstration : Infecting Arrays.



The array is constructed as an infected array underlined to test. We will design an array in send window and will notice the output in receive window.

The output subjected out to be:

Receive



The link gets easily embedded after serialization which shows that an array gets infected.

Infecting Notation Objects:

In this we will look how the notational objects are infected and result after serialization.

Send

```
{
  Member :
  "<a href='http://www.google.com'>
  Google Object</a>"

}
```

The receive window will throw output as:

Receive

The object is getting infected according to the concept. The hexadecimal of this test is

The next test will show you the third party connection is also possible.

Embedding Telnet Protocol Request Through Serialization

In this a simple object is defined in which a telnet protocol request is embedded. This is done to check whether the connection is passed by the serialization of data or not.

Send

```
{
  Telnet:
  "<a href='telnet://203.197.219.33:23'
   Telnet Test Through Serialization!
  </a>"

}
```

The output is:

Receive

Object	
Telnet	Telnet Test Through Serialization!

The test was successful when you click on the required link: Let's see what happen

When user launches application, a command shall appear asking for remote credentials for the server as shown above.

Conclusion

No doubt with the advent of new web technologies, the developer's task has become very easy but, the insecurity has increased. The concern should be to restrict the insecure element and reduce infections in web application. The serialization concept works in both ways. The approach of designing objects and fusibility of links should be scrutinized properly. At the last the protection is all yours.