Node.js 취약점 분석 방법론에 대한 연구

이범진*, 문시우**, 유재욱***, 신정훈***, 김경곤****
*전북대학교 IT 응용시스템공학과
**선린인터넷고등학교 정보통신과
***KITRI 차세대 보안리더 양성 프로그램 (Best of the Best)
****고려대학교 정보보호대학원
e-mail : posix.lee@gmail.com

A Study on the Vulnerability Analysis Methodology of Node.js

Beomjin Lee*, Siwoo Mun**, Jaewook You***, Jeonghoon Shin***, Kyounggon Kim****
*Department of IT Application Systems Engineering, Chonbuk National University
**Department of Information and Communication, Sunrin Internet High School
***Korea Information Technology Research Institute(KITRI) Best of the Best
****Graduate School of Information Security, Korea University

요 약

Node.js 는 자바스크립트 런타임으로 확장성 있는 웹 애플리케이션 개발에 주로 사용되는 플랫폼이다. 자바스크립트는 수 년에 걸쳐 많은 발전을 이뤘으며 현재는 Node.js 로 서버 사이드까지 자리 잡아 웹이라는 생태계에서 빼놓을 수 없는 요소가 되었다. 본 논문은 Node.js 에 관한 전반적인 이론을 포함하여 자바스크립트 특성으로 인하여 Node.js 에서 자주 발견되는 취약점의 탐색 방법을 모듈 관점에서 다뤄보고자 한다. 또한 취약점이 발생하는 다양한 패턴과 해당 패턴을 막기 위하여 적용된 Mitigation 을 실제 모듈을 예시로 살펴보며 분석해보고자 한다.

1. 서론

객체 기반 프로그래밍 언어인 자바스크립트는 현대 웹 애플리케이션에서 분리할 수 없는 요소가 되었다. 1995 년 12 월에 발표된 이래로 25 년 동안 웹 API 의 표준으로써 작용되어왔을 뿐만 아니라 브라우저에 훌륭한 동적 인터페이스를 제공함으로써 누구나 웹 어플리케이션을 편하게 접할 수 있게 되었다. Client-side 언어로 자리매김한 자바스크립트는 13 년에 이르러 새롭게 Node.js 라는 이름의 서버 사이드 언어로 사용범위를 넓혀 나갔고 2019 년 현재 REST API 를 뒷받침하는 Server-side 언어로서 주목을 받게 되었다 [3]. 브라우저에 쓰일 것을 고려하여 설계된 언어 한계를 극복하기 위해서 자바스크립트가 범용 언어로서 사용될 수 있도록 CommonJS 명세가 채택되었다. Server-side 언어로서 필요한 기능(I/O Framework)들을 모듈의 형태로 위치시켜 사용될 수 있도록 하였으며 편리한 개발 환경을 마련하여 수많은 개발자들이 간편하게 모듈을 제작해 업로드 할 수 있도록 하였다 [4].

Node.js 모듈 배포 환경의 대표적인 예시인 npm 은 모듈간 의존성을 관리하여 개발을 더욱 편리하게 만들어 줄 뿐만 아니라 버그 리포트 시스템 및 모듈 보안성 관리의 역할까지 수행해 주고 있다. 현재 시점에서 npm 에 등록된 Node.js 모듈의 수는 1,080,000 개를 넘었으며 Node.js 서버 구성에 사용자 모듈 사용은 필수적인 일이 되었다. 본 연구는 이러한 Node.js 언어의 보안에 영향을 끼칠 수 있는 특징들을 언급하고, 모듈을 개발하거나 사용함에 있어 취약점을 유발할 수 있는 부분을 정리하는 것을 목적으로 진행되었다. Server-side 언어의 관점에서 Node.js 모듈이 가지는 특징과 발생할 수 있는 주요 취약점 패턴을 소개하고 그러한 취약점을 탐색할 수 있는 방법론을 제시한다.

본 논문의 구성은 다음과 같다. 2 장과 3 장에서는 Node.js 의 보안과 연관되는 특성에 대해 설명한다. 4 장에서는 2 장과 3 장의 내용을 토대로, 2 장에 제시된 특성이 Node.js 모듈 보안 취약점 분석에 어떻게 적용될 수 있는지 연관 지어 설명한다. 5 장에서는 결론을 서술하며 본 논문을 마친다.

2. Characteristics of Node.js

1) Single-Thread Long-Running Process

Node.js 는 Google Chrome 내부에서 사용하는 V8 엔진을 채택하여 사용하고 있으므로, 자바스크립트 본연의 특성을 그대로 가지고 있다. 기존의 Server-side 언어(e.g. PHP, JSP)는 클라이언트의 연결에 따라 별도로 스레드를 분할하거나 프로세스를 생성하여 요청에 대응하는 경우가 일반적이지만, 자바스크립트는 모든 응답을 단일 스레드로 처리하기 때문에 타 언어에서의 멀티 스레드, 멀티 프로세스의 특성으로 발생하는 취약점(e.g. Deadlock, Race Condition)은 Node.js 에서 일어날 가능성이 비교적 낮다.

본 특성으로 인해 자바스크립트는 자원 사용에 제약을 가지지만 외부 프로그램과의 연결성을 지속적으로 유지할 수 있다는 장점 또한 가지고 있다. 대신 Prototype pollution 또는 적절하지 않은 로직에 의해 전역 변수, 함수 등의 값이 변조된다면, 서버가 재 실행되기 전까지 그대로 유지되어 공격자의 취약점 활용성을 증대시키는 원인이 되기도 한다.

2) Asynchronous

비동기성은 자바스크립트의 핵심적인 특성 중 하나로, 어떤 명령이 끝나지 않아도, 이어서 다음 작업을 병렬적으로 수행할 수 있도록 한다.

```
setTimeout(function() {
    console.log('Second');
}, 1000);
console.log('First');
```

(그림 1) asynchronous basic

(그림 1)은 특정 함수를 setTimeout 함수에 등록하여 임의의 시간 후에 실행하도록 작성한 비동기성을 설명하는 간단한 예제다. 실제 실행해보면 “First”가 출력된 후에 “Second”가 출력되는 것을 볼 수 있다. 자바스크립트에서 함수의 결과값은 언제 반환될지 알 수 없기 때문에, 동기적인 처리가 필요한 상황(e.g. 외부 URI 참조) 에서는 콜백함수를 등록해 주는 것으로 동기적인 처리를 해 줄 수 있다.

```
result = {}

function func() {
    setTimeout(function() {
        result = $.get('/');
    });
    return result;
}

console.log(func());
```

(그림 2) bad ordered call

```
result = {}

function func(callback) {
    setTimeout(function() {
        result = $.get('/');
        callback();
    });
}

func(function() {
    console.log(result);
});
```

(그림 3) synchronous with callback wrapping

(그림 2)는 현재 접속중인 서버의 루트 페이지를 대상으로 jQuery 의 get 메소드 요청을 통해 정보를 가져오는 코드이다. 그러나 비동기 특성으로 인해 $.get 함수가 무언가 정보를 반환하기 전에 result 를 반환한다. 따라서 result 가 반환하는 것은 처음 정의해 두었던 빈 객체가 된다. (그림 3)는 그러한 문제점을 해결하기 위해 callback 함수로 wrapping 하여 후처리 되어야 할 명령을 $.get 이 값을 반환하여 result 에 대입하고 나서 수행할 수 있도록 한 것이다. 이외에 es6 문법에서 지원하는 async, await, promise 기능을 통해서 동기 처리를 해 줄 수도 있다.

이러한 특성으로 인하여 자바스크립트의 Root Context 에서는 어떠한 명령이 지연되더라도 다음 명령을 병렬 연산하기에 프로그램이 완전히 정지해 버리는 일은 일어나지 않는다. 하지만 Callback Wrapping 을 통해 동기성을 부여한 것과 같은 원리로 자바스크립트에서의 Event Listener 는 콜백으로 구성되어 있기에 Event Listener 는 동기적인 프로세스로 수행된다. 따라서 Event Listener 함수가 값을 반환하기 전까지는 다음 이벤트를 수행할 수 없게 되는데, Node.js 서버구성에 자주 사용되는 http, express 모듈 등은 사용자 요청에 대한 응답 생성을 Event Listener 를 통해 대응하는 방식을 취하고 있으므로 클라이언트의 요청에 대한 응답은 모두 동기적으로 처리된다. 따라서 http, express 모듈에서 프로그램 수행이 지연된다면 서버 전체를 정지시킬 수 있게 된다.

3) Module Dependent

Node.js 는 Python, Ruby 와 비슷하게 많은 기능을 모듈에 의존한다. 따라서 서버에서 사용하는 모듈을 참조하는 모듈에서 취약점이 발견된 경우 해당 서버에도 영향을 끼칠 수 있으며 이러한 취약성은 모듈 사용 패턴에서 추적할 수 있다.

3. Characteristics of Node.js Modules

1) Development Environment

Node.js 애플리케이션 개발 환경 구성에 있어, NPM 을 사용하며 디렉토리 별로 각각 다른 모듈이 설치되도록 프로젝트를 구성할 수 있다. NPM 은 package.json 파일을 통해 모듈 의존성을 관리하며 install, npm-check, outdated 등의 하위 명령어를 사용해 package.json 에 정의된 모듈을 관리할 수 있다.

2) Dependency Management

서버에서 참조하는 서로 다른 모듈에서 중복된 참조를 가질 때, 참조되는 모듈에 대한 충돌을 방지하기 위해서 node_modules 디렉토리에 모듈을 통합하여 보관하며, 저장된 각각의 모듈에 대해 package.json 를 별도로 관리한다. 따라서 Node.js 모듈 사용에 있어서, 사용자는 의존성 충돌 문제를 고민할 필요가 없다.

4. Module Vulnerability & Analysis Methodology

1) Prototype Pollution

본 취약점은 자바스크립트 객체 구조의 특이성으로

인해 성립된다. 따라서 Node.js 등 자바스크립트 엔진을 기반으로 동작하는 프로그램에서만 발생할 수 있는 특수한 취약점이다. 자바스크립트 실행 환경에 있어 객체의 중요 값들을 조작하여 실행 흐름을 바꾸는 등의 행위가 가능하며 Node.js 와 같은 서버 사이드에서는 상황에 따라 임의 코드 실행이 가능해지므로 심각한 보안 문제를 유발할 수 있다. 자바스크립트의 자료형은 prototype 속성을 포함하는데 특정 객체 / 클래스 / 함수의 prototype 데이터의 변조가 가능하다면 이후 그 객체 / 클래스 / 함수의 생성자를 통해 새로운 인스턴스를 생성하면 prototype 에 지정하였던 속성 등이 그대로 유지된다 [2].

```
var obj2 = new Object();
obj2.__proto__.polluted = true;
var obj = new Object;
console.log(polluted); // true
```

(그림 5) prototype pollution basic

(그림 5)를 통해 알 수 있듯, 객체 클래스 인스턴스의 __proto__ 속성을 부여해 주는 것으로 다음 생성될 인스턴스의 속성에 영향을 끼칠 수 있다.

Node.js 모듈에서 prototype pollution 공격이 성립되려면, 최소한 아래와 같은 방식의 이중 참조 정의가 이루어져야 한다.

```
obj[a][b] = value
```

(그림 6) double reference definition

공격자는 a 값을 __proto__로 정의하고 b 값을 임의로 변조함으로써 앞으로 새롭게 정의될 모든 객체에 영향을 줄 수 있다.

```
obj[a][b][c] = value
```

(그림 7) triple reference definition

삼중 참조 역시 동작한다. a 값을 constructor 로 정의하고 b 에 해당하는 값을 prototype 으로 지정하여, 앞으로 생성될 객체에서의 c 속성 값을 임의로 지정할 수 있게 된다. 하지만 일반적으로 삼중 참조 할당이 일어나기 위해서는 상당히 복잡한 할당 로직을 필요로 하기 때문에 일반적으로 일어나기 쉬운 상황은 아니다.

```
class A { }
var obj = new A();
obj.__proto__.__proto__.polluted = true;
console.log(polluted);
```

(그림 8) prototype pollution for user-defined class

만약 Pollution 가능한 객체의 클래스가 Object 가 아니더라도 __proto__ 속성을 여러 차례 참조함으로써 최종적으로 Object Prototype 을 가리키도록 할 수 있다.

```
class A { }
class B extends A { }

var obj = new B();
obj.__proto__.__proto__.__proto__.polluted = t
rue;
console.log(polluted);
```

(그림 9) prototype pollution for user-defined class 2

만약 대상 클래스가 타 클래스를 상속하는 형태를 띄고 있다면 참조의 단계는 더욱 깊어져야 한다. 일반적으로 prototype pollution 은 재귀적으로 객체의 값을 참조, 할당하거나 직접적으로 다중 참조를 거칠 때 발생할 수 있다. 모듈 내에서 발생할 수 있는 일반적인 패턴으로는 속성 정의, 객체 병합, 객체 복사 등의 객체 내부 값을 제어하는 기능 구현에서 사용자가 인자를 자유롭게 설정할 수 있고, 인자로 전달되는 객체의 key 값을 검증하는 로직이 없다면 해당 모듈은 취약하다고 판단한다.

```
function merge(target, source) {
    keys = Object.keys(source);
    keys.forEach((key)=>{
        value = source[key];
        if (typeof value == 'object') merge(ta
rget[key], value);
        else target[key] = value;
    })
    return target
};
```

(그림 10) deep merge

(그림 10)은 prototype pollution 취약점이 발생하는 deep merge 알고리즘의 일반적인 경우이다. 검증 로직이 부재함을 확인할 수 있다.

```
var obj = {};
merge({}, JSON.parse('{ "__proto__" : { "pollu
ted" : true } }'));
console.log(this.polluted);
```

(그림 11) deep merge to prototype pollution

따라서 함수의 source 인자로 __proto__ 속성이 포함된 객체를 전달하면 prototype pollution 이 가능하다. 이러한 유틸리티 코드의 취약성 판단을 위해서는, source 에 사용자 입력이 주어질 수 있는지 확인하는 과정이 필요하다.

```
if (key === 'constructor'
&& typeof object[key] === 'function') return;
if (key == '__proto__') return;
```

(그림 12) prevent prototype pollution

(그림 12)는 lodash 모듈에 추가된 검증 로직이다. 모듈 내부에서 속성 정의, 객체 병합, 객체 복사 등의 동작을 수행할 때 이처럼 인자로 전달된 객체에 대한 검증을 수행한다면 취약점 발생을 방지할 수 있을 것이다.

2) Denial of Service

비동기를 사용하는 자바스크립트 특성상, 일반적인 상황에서는 작업이 지연되어도 프로세스가 멈추는 일은 발생하지 않는다. 하지만, Event Loop 내에서 실행되는 작업에서 지연이 발생되면, 서버 전체가 정지될 수 있다. Node.js 모듈에서 탐색 가능한 Denial of Service 취약점은 주로 3 가지로 분류할 수 있다.

① Infinite Loop

```
var arr = { "len" : "Infinity" };
var new_arr = { };
new_arr.keys = Object.keys(arr);
new_arr.len = new_arr['keys'].length;

for (let i = 0; i < new_arr.len; ++i) {
    new_arr[new_arr.keys[i]] =
        arr[new_arr.keys[i]];
}
```

(그림 13) double reference definition

Node.js 의 프로그램 흐름은 싱글 스레드의 Event Loop 에 의해서 관리되므로, 만약 반복문 종료 조건을 조작하는 것이 가능하다면, 프로그램 전체가 마비되는 상황에 빠질 수 있다.

② ReDoS

자바스크립트에서 정규식을 통한 검색은 Event Loop 를 통해 수행된다. 따라서 정규식 연산이 지연되면 서버가 마비될 수 있다. ReDoS 는 그러한 자바스크립트의 특성을 이용해 Denial of Service 를 일으키는 취약점이다 [1]. ReDoS 는 두가지 방법을 통해 발생시킬 수 있는데 정규식에 대응되는 문자열을 조절 가능하며 독립되는 마지막 문자가 7 번째 순서(서버 성능에 의존적임) 이후에 위치하는 대상 문자열을 예측할 수 있는 경우, 그리고 정규식이 어느정도 취약하고 검색에 사용되는 대상 문자열을 조작 가능한 경우가 그것이다.

```
function dos(str) {
    regex = new RegExp(' *, *');
    regex.exec(str);
}
dos(' '.repeat(1e5) + 'x')
```

(그림 14) ReDos with evil regex

(그림 14)는 취약한 정규식을 통해 ReDoS 을 유발시키는 예시이다. 100,000 자 이상의 입력을 필요로 하지만 서버로 운용되고 있는 상황에서 100,000 자 입력은 어렵지 않게 주어질 수 있으므로 공격자가 반복하여 주입한다면 적은 량의 패킷으로도 서버를 지속하여 마비시킬 수 있을 것이다.

```
function dos(regex_str) {
    var regex = new RegExp(regex_str);
    regex.exec('deadbeef');
}
dos("((((((((((([^f]+)+)+)+)+)+)+)+)+)+)+$");
```

(그림 15) ReDoS with regex control

(그림 15)는 정규식 생성에 관여할 수 있는 경우에 ReDoS 를 발생시키는 예시이다. 독립적으로 존재하여 특정 가능한 마지막 문자가 7 번째 이상의 순서에 위치한다면 이를 통해 ReDoS 를 유발할 수 있다.

③ Memory Exhaustion

```
for (let i = 0; i < 100; ++i) {
    this[`a${i}`] = 'a'.repeat(1e8);
    this[`a${i}`].substr(0,1);
};
```

(그림 16) Node.js Memory Exhaustion

(그림 16)은 메모리 과할당을 통해 Denial of Service 를 발생시키는 예시이다. 적절하지 못한 메모리 관리로 인해 메모리 누수, 과할당 등을 통한 메모리 고갈이 일어날 경우 서버가 완전히 종료될 수 있다.

5. 결론

본 논문은 Node.js 에 관한 전반적인 개념을 포함하여 자바스크립트의 특성으로 인해 Node.js 에서 일어날 수 있는 취약점의 탐색 방법을 모듈 관점에서 살펴봄으로써 Node.js 개발자가 일반적인 Node.js 모듈 취약점에 대해 이해하고 탐색할 수 있도록 방법을 제시하는 것을 목표로 하고 있다. 이러한 연구 결과는 Node.js 모듈을 개발하고 사용함에 있어서 보안적인 측면의 안전성을 높이는 계기가 될 것이다. 또한 국내에서 Node.js 자체 특성을 이용한 취약점의 발생 원인을 다루는 연구가 부족한 실정이므로, 본 논문이 국내 Node.js 보안 생태계 발전에 기여할 수 있을 것으로 기대한다.

참고문헌

[1] Cristian-Alexandru Staicu, Michael Prade “Freezing the Web: A Study of ReDoS Vulnerabilities in JavaScript-based Web Servers”
[2] Olivier Arteau “Prototype pollution attack in Node.js application”
[3] Doglio, Fernando “REST API Development with Node.js”
[4] Jeong-Hwan. (2019). WebFrameworks.kr – “Create API Server using NodeJS”