AST 와 CFG 에 기반한 Node.js 모듈 취약점 분석

김희연*, 오호균*, 김지훈*, 유재욱**, 신정훈**, 김경곤***
*고려대학교 사이버국방학과
**KITRI 한국정보기술연구원
***고려대학교 정보보호대학원
e-mail : sonysame@naver.com

Node.js Module Vulnerability Analysis: Based on AST and CFG

Hee Yeon Kim *, Ho Kyun Oh*, Ji Hoon Kim*, Jaewook You**, Jeong Hoon Shin**, Kyounggon Kim ***
*Department of Cyber Defense, Korea University
**Korea Information Technology Research Institute(KITRI) Best of the Best
***Graduate School of Information Security, Korea University

요 약

웹어플리케이션의 발전에 따라 자바스크립트 런타임 플랫폼인 Node.js 의 사용도 증가하고 있다. 개발자들은 Node.js 의 다양한 모듈을 활용하여 프로그래밍을 하게 되는데, Node.js 모듈 보안의 중요성에 비하여 모듈 취약점 분석은 충분히 이루어지지 않고 있다. 본 논문에서는 소스코드의 구조를 트리 형태로 표현하는 Abstract Syntax Tree 와 소스코드의 실행 흐름 및 변수의 흐름을 그래프로 나타내는 Control Flow Graph/Data Flow Graph 가 Node.js 모듈 취약점 분석에 효율적으로 활용될 수 있음을 서술하고자 한다. Node.js 모듈은 여러 스크립트 파일로 나누어져 있다는 점과 사용자의 입력이 분명하다는 특징이 있다. 또한 자바스크립트 언어를 사용하므로 선언된 변수들의 타입에 따라 적용되는 범위인 scope 가 다르게 적용된다는 특징이 있다. 본 논문에서는 이러한 Node.js 모듈의 특징을 고려하여 Abstract Syntax Tree 및 Control Flow Graph/Data Flow Graph 을 어떻게 생성하고 취약점 분석에 활용할 것인지에 대한 방법론을 제안하고, 실제 분석에 활용할 수 있는 코드 구현을 통하여 구체화시키고자 한다.

1. 서론

Node.js 는 자바스크립트를 웹브라우저가 아닌 환경에서 실행시켜줄 수 있는 플랫폼으로 웹 서버와 같이 확장성 있는 네트워크 프로그램 제작을 위하여 고안되었다. Node.js 를 통한 개발에 있어서 개발자들은 다양한 기능을 구현하기 위하여 NPM(Node Package Manager)에서 제공하는 모듈들을 다운로드 받아 사용한다. Node.js 는 2009 년 등장 이후, 기하급수적으로 사용량이 증가하고 있다. 2018 년에는 일일 평균 1,010,921 번의 다운로드가 이루어졌으며, NPM 에서 제공되는 모듈 또한 작년에 비해 약 40 만개가 증가하여 현재 약 105 만개의 모듈을 제공하고 있다 [1, 2]. 그러나 이처럼 Node.js 의 사용 및 Node.js 모듈 개발은 확대되고 있음에도 불구하고 모듈 자체의 보안 취약점 분석은 체계적으로 이루어지지 않고 있다.

Node.js 의 각 모듈은 서로 종속성을 갖고 있는 경우가 많다. 웹서버 개발에 사용되는 express 모듈의 경우, 36176 개의 다른 모듈에서 해당 모듈을 호출하여 사용한다 [3]. 이처럼 종속성이 높은 모듈에서 취약점이 발생하게 된다면 다른 모듈에서도 의존적으로 취약점이 발생하므로, 큰 영향을 미치게 된다 [4]. Node.js 모듈 사용자는 이러한 의존성 모듈들의 취약점에 대해서는 고려하지 않고 사용하는 경우가 많기 때문에 Node.js 모듈 보안 취약점 분석은 체계적으로 이루어져야 한다.

현재 사용되고 있는 Snyk, Retire.js, Fortify 와 같은 Node.js 모듈 보안점검 도구들은 대게 모듈 버전 체크를 통하여 해당 버전의 알려진 취약점을 점검해주는 기능을 하고 있다. 그러나 버전 체크를 통한 취약점 점검은 소스코드 자체를 분석하지 않기 때문에 새로운 취약점을 찾을 수 없으므로 한계점이 존재한다.

Shiyi Wei 의 연구에서는 웹사이트에서 사용되고 있는 자바스크립트 소스코드에 초점을 맞추어 JSBAF(자바스크립트 Blended Analysis Framework)툴을 제작하여 취약점 분석방안을 제시하였다 [5]. 또한 Murthy 는 특허를 통하여 동적인 특성을 지니는 자바스크립트코드 분석에 CFG(Control Flow Graph)를 활용하는 방안을 제안하였다 [6]. Quinlan 의 연구에서는 C 와 C++언어

로 구현된 소스코드에서 AST(Abstract Syntax Tree)를 이용하여 코드패턴 매칭을 통하여 취약점을 분석하는 방안을 제시하고 있다 [7]. 이러한 기존 연구들의 대부분은 웹 상에서 사용되는 자바스크립트 코드 취약점 분석에 초점이 맞추어져 있었으며, Node.js 모듈과 같은 광범위한 타겟에 대해서는 연구된 바가 없다. 본 논문에서는 Node.js 모듈 취약점 분석에 AST 와 CFG 분석 방식을 적용하고자 한다. 이 때, Node.js 모듈 사용의 특성을 고려하여 AST 와 CFG 를 효율적으로 활용하는 모듈 보안 취약점 분석 방법론을 제안하고자 한다.

본 논문의 구성은 다음과 같다. 2 장에서는 Node.js 모듈의 보안 취약점을 분석하기 위한 방법을 제시한다. 3 장에서는 2 장에서 제시한 방법이 실제 Node.js 모듈 보안 취약점 분석에 적용될 수 있음을 보이며 분석 방법을 구체화한다. 4 장에서는 결론을 서술하며 본 논문을 마친다.

2. Node.js 모듈 보안 취약점 분석 방법

일반적인 자바스크립트 클라이언트 코드 및 Node.js 서버 어플리케이션에 대한 취약점 분석은 코드상의 취약점 발생 부분까지 사용자 입력이 도달할 수 있는 지를 확인하는 것에 초점이 맞추어져 있다. 이는 곧 공격자가 해당 결함에 대한 접근이 가능한지 검증하는 것과 같은 의미를 지니며, 본 연구는 이러한 접근 방식을 Node.js 모듈에 대해서도 동일하게 적용하여, 모듈에서 외부에 노출되는 인터페이스에 대한 입력이 코드상의 취약 부분까지 도달할 가능성을 분석하고자 한다. 분석 단계는 다음과 같이 나뉜다.

2.1 소스코드 패킹

일반적인 자바스크립트 코드는 하나의 스크립트로 동작하며, 이는 본 연구에서 활용하고자 하는 AST 및 CFG 생성에 이점으로 작용한다. 그러나 Node.js 모듈의 경우에는 스크립트를 여러 자바스크립트 파일로 나누어서 작성하는 경우가 많으며, 외부 모듈을 require 함으로써 발생하는 의존성도 존재한다. 따라서 Node.js 모듈 분석에 있어서 여러 자바스크립트 코드를 하나의 파일로 합치는 과정인 패킹이 선행되어야 한다. 이를 통하여 외부 모듈과의 종속성으로 인해 발생하는 코드 전체 분석의 어려움도 해결할 수 있다.

2.2 AST 생성

AST 는 Abstract Syntax Tree 로 프로그래밍 언어로 작성된 소스코드의 추상 구문 구조의 트리이다. 트리의 각 노드는 소스코드에 등장하는 모든 구조체이며 모든 연산자와 피연산자, 분기문과 반복문, 선언문 등을 각각 하나의 노드로 표현한다. AST 를 이용하면 소스코드의 문장들을 계층구조로 나타낼 수 있다. 예를 들어 if(x==1)이라는 자바스크립트 소스코드 문장은 AST 로 (그림 1)과 같이 나타낼 수 있다.

```
"type": "IfStatement",
"test": {
        "type": "BinaryExpression",
        "operator": "==",
        "left": {
                "type": "Identifier",
                "name": "x"
        },
        "right": {
                "type": "Literal",
                "value": 1,
                "raw": "1"
        }
}
```

(그림 1) Example of AST - if(x==1)

2.1 에서 하나의 Node.js 모듈을 구성하고 있는 여러 스크립트들을 하나의 파일로 합치고 난 뒤, 위와 같이 AST 를 생성하면 해당 모듈의 전체 코드를 하나의 트리 구조로 표현할 수 있다.

Quinlan 의 연구는 패턴 매칭을 통해서 C/C++로 작성된 소스코드 분석이 가능함을 보여주었다 [7]. 취약한 소스코드가 가지는 특성을 AST 를 이용하여 계층적으로 특정할 수 있기 때문에 이는 취약점 패턴 분석에도 사용될 수 있다. 취약한 코드의 특성을 패턴화 시킬 수 있다면, 코드의 전체 구조를 트리 형태로 나타낸 AST 를 생성함으로써 코드의 취약한 부분을 식별할 수 있다. 또한 코드에서 사용자의 입력을 받을 수 있는 부분도 AST 를 생성하여 식별 가능하기 때문에 AST 는 코드에서 취약한 패턴의 존재 여부 및 사용자의 입력 위치를 찾는 과정에서 효과적으로 활용 가능하다. 따라서 사용자의 입력이 분명하다는 Node.js 의 특징을 이용하여, AST 는 Node.js 모듈 취약점 분석에 매우 효율적으로 사용될 수 있다.

2.3 CFG 생성

CFG 는 Control Flow Graph 로 프로그램의 모든 가능한 실행 흐름을 그래프로 표현한 것이다. 주로 정적 분석 도구와 컴파일러에서 사용되며 프로그램의 실행 흐름을 나타낼 수 있다. CFG 는 노드와 엣지로 구성되는데, 점프가 없는 기본 블록을 노드로 하고 해당 블록에서 다른 블록으로 실행 가능한 흐름을 엣지로 나타낸다.

2.2 에서의 AST 는 패턴 매칭에 효율적으로 사용될 수 있다. 그러나 실제 코드 실행 단계에서의 실행 흐름을 알 수 없다는 한계점을 가진다. 따라서 프로그램 실행 단계에서 공격자의 입력이 코드의 취약한 부분까지 도달 가능한지에 대한 여부는 실행 흐름을 그래프로 나타낸 CFG 로 식별할 수 있다.

또한 변수의 실행 흐름을 파악하는 과정이 필요하다. Source 를 취약한 입력으로 정의하고 Sink 를 취약점이 존재하는 부분으로 정의했을 때, Source 가 Sink 로 도달하는 것은 Sink 에 들어가는 입력의 변수 흐름을 역추적하면 알아낼 수 있다. 이 때 우선 어떠한 변수들이 존재하는지 파악할 필요가 있다. 자바스크립트에는 var, let, const 및 전역 변수와 같은 다양한

변수 선언 방법이 있다. 선언하는 타입에 따라 변수의 적용 범위인 scope 가 서로 다르게 지정되는데, 이는 타 언어와는 다른 자바스크립트의 고유한 특징이다 [8]. 따라서 Node.js 모듈을 분석하는 경우, 변수들의 scope 를 명확히 구분하는 과정이 필요하다.

```
1 function a() {
2   {
3     var b = 1;
4   }
5   return b;
6 }
7 console.log(a()); // 1
```

(그림 2) Example of function-level scope

```
1 function a() {
2   {
3     let b = 1;
4   }
5   return b;
6 }
7 console.log(a());
8 // ReferenceError
```

(그림 3) Example of block-level scope

(그림 2)와 (그림 3)은 각각 function-level scope, block-level scope 의 예제이다. var 로 선언된 변수의 경우, scope 계산이 function 단위로 이루어진다. 따라서 (그림 2)에서는 함수 내 전체 블록에서 변수 b 에 대한 접근이 가능하다. 반면 let 의 경우 scope 계산이 블록 단위로 이루어지므로, (그림 3)에서 function a 내부이더라도 변수 b 가 선언된 블록의 외부에서 변수 b 에 대한 접근을 시도하면 ReferenceError 가 일어난다. 자바스크립트에서는 변수에 적용되는 scope 에 따라 데이터 흐름이 달라질 수 있다. 따라서 Node.js 모듈 분석의 경우, 사용되는 각 변수의 scope 를 구분하는 작업이 필요하다.

3. Node.js 모듈 보안 취약점 분석 구체화

(그림 4)는 Arbitrary Code Execution 취약점이 존재한다고 알려진 Node.js 의 fast-redact 모듈의 소스코드 중 일부이다. 본 절에서는 fast-redact 모듈을 예시로 활용하여 2 장에서 제안한 AST 와 CFG 에 기반한 모듈 취약점 분석 방법을 구체화시키고자 한다.

```
2  if (/O/.test(s)) throw Error()
3  const proxy = new Proxy({}, {get: () => proxy, set: () => { throw Error() }})
4  const expr = s.replace(/^\*/, 'O').replace(/\.\*/g, '.O').replace(/\[\*\]/g, '[O]')
5  if (/\/\*/.test(expr)) throw Error()
6  /* eslint-disable-next-line */
7  new Script(`
8    o.${expr}
9    if ([o.${expr}].length !== 1) throw Error()
10 `).runInNewContext({o: proxy, O: null})
```

(그림 4) fast-redact 소스코드 중 일부

3.1 소스코드 패킹 구현

Webpack 은 현대 자바스크립트 어플리케이션을 위한 정적 모듈 번들러이다 [9]. Webpack 을 이용하여 코드에 대한 의존성을 분석한 후 원하는 파일들을 합쳐 하나의 코드로 생성해낼 수 있다. 따라서 본 연구에서는 모듈의 Javascript 코드를 하나로 합쳐주는 스크립트를 작성하여 원하는 버전의 모듈의 소스를 하나의 파일로 만들어 이용하였다.

3.2 AST 생성 구현

모듈의 Script 함수에서 사용자의 입력이 매개변수로 사용될 경우 공격자의 악의적인 입력을 통해 원격 실행공격까지 이어질 수 있다. fast-redact 모듈의 경우 해당 취약점이 존재하며, AST 를 이용하여 이러한 취약점을 일으킬 수 있는 패턴이 분석하고자 하는 코드에 존재하는지 식별할 수 있다.

<알고리즘 1> Algorithm for detecting vulnerable scripts

Algorithm 1 Algorithm for detecting vulnerable scripts

```
1: Esprima = require('esprima')
2: ast = Esprima.parseScript(this.code, {range: true})
3: traverse(ast, function (node) {
4:     if node.callee then
5:         if node.callee.type = 'Identifier' and node.callee.name = 'Script' then
6:             if node.arguments[0].type ≠ 'Literal' then
7:                 console.log('Script Vulnerable');
8:             end if
9:         end if
10:    end if
   })
```

알고리즘 1 은 본 연구에서 구현한 AST 를 이용하여 해당 패턴의 코드를 탐지할 수 있는 알고리즘이다. ECMAScript Parser 인 Esprima 를 이용하여 AST 를 생성하였다 [10]. AST 상에서 Script 라는 이름의 Identifier 를 찾은 뒤, 매개변수로 상수(Literal)가 아닌 값이 전달되는 경우를 탐지한다 (알고리즘 1 의 line 5, line 6). 알고리즘 1 을 통하여 fast-redact 모듈의 코드에서 'new Script(...' (그림 4 의 line 7) '부분에서 취약점이 존재함을 AST 를 통해 탐지할 수 있다.

3.3 CFG 생성 구현

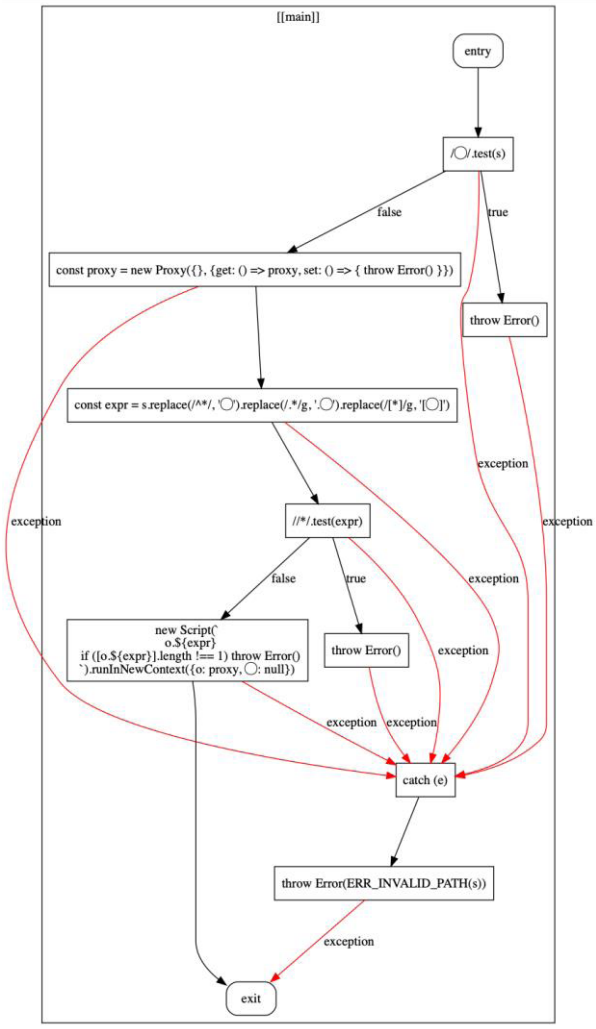
Esprima 를 이용하여 분석하고자 하는 코드의 AST 를 생성한 뒤에는, Esgraph 를 이용하여 CFG 를 생성할 수 있다 [11]. 알고리즘 2 는 Esprima 와 Esgraph 를 이용하여 CFG 를 생성하는 알고리즘이다.

<알고리즘 2> Algorithm for detecting vulnerable scripts

Algorithm 2 Algorithm for detecting vulnerable scripts

```
1: Esprima = require('esprima')
2: Esgraph = require('esgraph')
3: ast = Esprima.parseScript(this.code, {range: true})
4: cfg = Esgraph(ast)
```

(그림 5)는 fast-redact 모듈의 소스코드에 대하여 생성한 CFG 이다. 해당 CFG 를 분석하여 3.1 에서 AST 로 식별한 취약점 패턴으로 실행 흐름이 도달하기 위해서는 실행 중 어떠한 분기문들을 거쳐야 하는 지 파악할 수 있다. fast-redact 모듈의 경우 소스코드에서 나오는 두 번의 분기문(그림 4 의 line 2, line 5)에서 false 분기로 실행 흐름이 진행되어야 3.1 에서의 AST 로 식별한 코드의 취약한 부분에 도달할 수 있음을 알아낼 수 있다. 또한 데이터 흐름에 대한 분석을 통하여 두 번의 조건문이 false 의 결과가 나오기 위한 데이터 흐름 조건을 파악할 수 있다.



(그림 5) fast-redact 모듈의 control flow graph

4. 결론

Node.js 를 사용하는 개발자들은 계속해서 증가하고 있으며, Node.js 의 보안 역시 그 중요성이 부각되고 있다. 그러나 현재 Node.js 모듈의 취약점을 점검할 수 있는 도구는 이미 알려진 취약점에 대한 통계적 분석을 통해 사용중인 모듈이 취약한 버전인지 검사해주는 정도에 머무른다.

본 논문에서는 AST 와 CFG 를 적용하여 Node.js 모듈의 취약점을 효율적으로 분석할 수 있는 방법론을 제안한다. 우선 모듈 소스 분석의 용이함과 의존성 해결을 위하여 소스 코드를 하나로 합치는 패킹 작업을 수행한다. 이후 소스 코드를 파싱하여 AST 와 CFG 를 생성한다. 마지막으로 미리 정의된 Source 와 Sink 패턴을 탐지하여 Source 입력이 Sink 에 도달하는지를 판별하는 작업을 통해 취약점 여부를 판단한다. 또한 전체적인 과정을 fast-redact 모듈의 기존 취약점에 대해서 테스트해봄으로써 위의 기법이 효과적임을 보였다. 본 논문은 AST 와 CFG 를 조합하여 기존의 툴과 연구문헌으로 찾지 못했던 취약점을 보다 포괄적으로 찾을 수 있는 방법으로 하나의 예시를 바탕으로 분석하였다. 이러한 취약점 패턴 기법을 전체 Node.js 모듈에 대해 자동화 수행 시 상당히 많은 취약점 패턴들이 검출될 것이라 예상되며, 이는 향후 과제이다. 이를 통해 Node.js 모듈 사용의 소프트웨어 안전성에 기여하고자 하는 것이 본 연구의 궁극적인 목표이다. 보다 정확한 분석을 위해서 DFG(Data Flow Graph)를 이용한 데이터 흐름 분석과 취약점 존재 여부 검증을 위한 동적 분석에 대한 연구가 필요하다.

참고문헌

[1] Node by Numbers 2018 — NodeSource. (2019). Retrieved 23 September 2019, from https://nodesource.com/node-by-numbers

[2] Modulecounts. (2019). Retrieved 23 September 2019, from http://www.modulecounts.com

[3] express. (2019). Retrieved from https://www.npmjs.com/package/express

[4] Teixeira, P. (2012). *Professional Node. js: Building 자바스크립트 based scalable software*. John Wiley & Sons.

[5] Wei, S., & Ryder, B. G. (2013, July). Practical blended taint analysis for 자바스크립트. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis* (pp. 336-346). ACM.

[6] Murthy, P. K., & Rajan, S. P. (2014). *U.S. Patent No. 8,656,370*. Washington, DC: U.S. Patent and Trademark Office.

[7] Quinlan, D. J., Vuduc, R. W., & Misherghi, G. (2007, July). Techniques for specifying bug patterns. In *Proceedings of the 2007 ACM workshop on Parallel and distributed systems: testing and debugging* (pp. 27-35). ACM.

[8] Zakas, N. C. (2016). *Understanding ECMAScript 6: the definitive guide for 자바스크립트 developers*. No Starch Press.

[9] Clow, M. (2018). Introducing Webpack. In *Angular 5 Projects* (pp. 133-137). Apress, Berkeley, CA.

[10] Hidayat, A. (2017). Esprima: Ecmascript parsing infrastructure for multipurpose analysis.

[11] esgraph. (2017). Retrieved from https://www.npmjs.com/package/esgraph