2022

JavaScript
Secure Coding Guide

JavaScript 시큐어코딩 가이드

과학기술정보통신부

KISA 한국인터넷진흥원

COTENTS

COTENTS

PART 제3장 부록

1

PART 제1장

개 요

1 개요

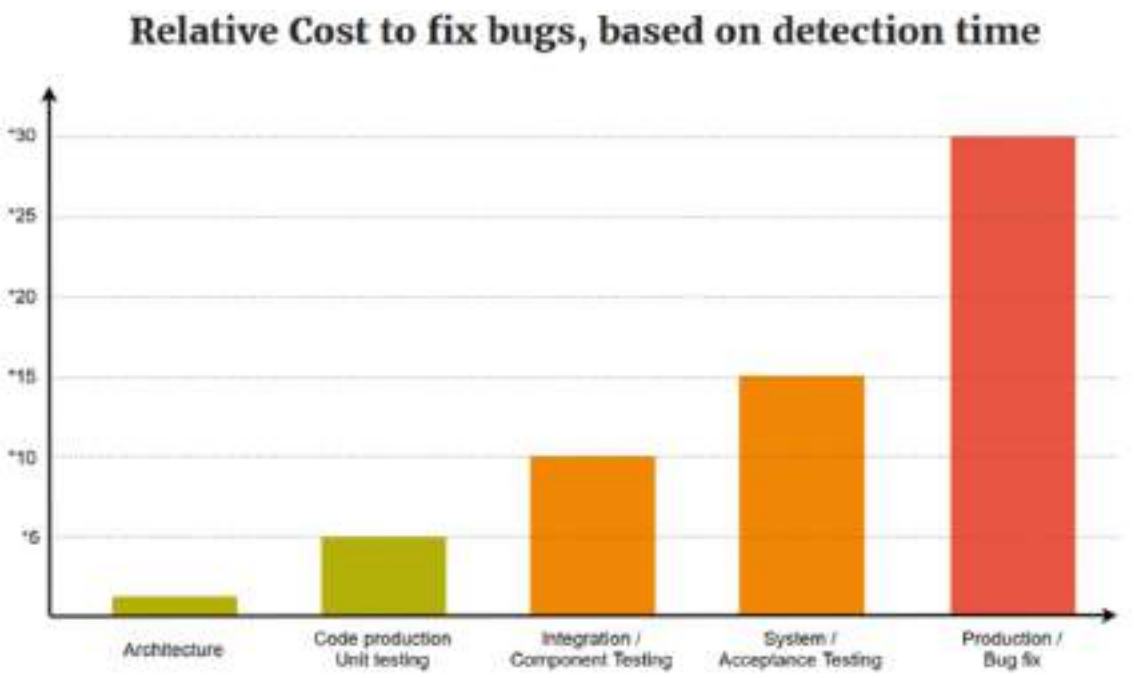
제1절 배경

빠른 실행과 반짝이는 서비스를 앞세운 스타트업들의 성장은 IT 생태계 전반에 큰 영향을 미쳤다. 체계적이고 큰 규모의 프로젝트를 중심으로 자리 잡았던 소프트웨어 개발 문화도 빠른 제품 개발과 고객의 요구사항 대응을 위해 더욱 가볍고 효율적인 방법을 모색하고, 지속적으로 진화하고 있다. 이러한 변화의 흐름의 중심에는 웹 기술의 발전이 큰 부분을 차지한다. 웹은 컴파일 기반의 여타 제품들에 비해 빠르고 가볍게 개발이 가능하며, 별도의 배포 및 설치 없이도 고객의 PC 또는 스마트폰으로 서비스 제공이 가능하다.

컴퓨터 하드웨어와 브라우저 성능이 뒷받침되지 않았던 상황에서, 초창기 웹 서비스는 웹 서버를 중심으로 동작했으며, 클라이언트는 단순히 필요한 정보를 요청하고 응답을 화면에 보여주는 수준에 그쳤다. 하지만 하드웨어 성능의 비약적인 발전과 더불어 웹 생태계를 구성하는 엔진이 고도화되면서 클라이언트의 비중도 함께 커지게 되었다. Angular.js, ReactJS와 같은 프레임워크의 등장으로 변화는 더욱 가속화 되었으며, 하이브리드 웹, 프로그래시브 웹 앱(PWA)의 등장으로 모바일 앱의 영역까지 확장되었다.

별도의 독립 파일 또는 패키지 형태로 배포되고 관리되는 일반적인 소프트웨어와 달리 웹은 브라우저라는 특수한 환경 위에서 동작하는 구조로, 사용자와 맞닿아 있는 인터페이스와 서비스의 핵심 데이터를 제공하는 서버 사이의 거리가 매우 멀고, 동시에 많은 사용자들이 이용한다는 특성을 가진다. 즉, 개발자가 작성한 코드의 많은 부분이 사용자에게 노출되어야 하고, 사용자가 제공하고 요청하는 정보가 서버를 오가기까지 많은 신뢰할 수 없는 구간들을 거쳐야 하는 위험성을 가진다.

초기 웹 서비스는 클라이언트의 비중이 그리 높지 않았으며, 서버의 핵심 자원을 안전하게 보호하는 것에 초점이 맞춰졌다. 하지만 웹 생태계가 거대해지고 복잡해지면서 클라이언트의 역할 또한 커지고 있으며, 안전한 서비스 제공을 위해 고려해야 할 위협 요소도 증가 · 분산되어 위협에 대응하는 것이 더욱 어려워졌다. 지속 가능한 서비스 안정성 및 보안성 보장을 위해선 개발 단계부터 보안을 적용하는 것을 고려해야 한다.



개발 단계별 버그 수정에 필요한 투입 비용 그래프[1)]

소프트웨어 개발보안은 보안 위협에 대응할 수 있는 소프트웨어를 개발하기 위한 일련의 보안 활동으로, 소프트웨어 개발 생명주기(SDLC)의 각 단계별로 요구되는 보안활동을 수행하는 것을 의미한다. NIST에서 공개한 자료에 따르면, 서비스 배포 이후 버그를 수정하는 비용이 설계 단계에서의 버그 수정에 비해 약 30배에 가까운 비용이 소요된다. 여기에 보안 사고로 인한 손실까지 감안하면 초기 단계의 개발보안 적용은 더 큰 손실을 미연에 방지하는 가장 효율적인 보안 대책이라고 볼 수 있다.

소프트웨어 개발보안의 중요성을 이해하고 체계화한 미국의 경우 국토안보부(DHS)를 중심으로 시큐어코딩을 포함한 소프트웨어(SW) 개발 전 과정(설계, 구현, 시험 등)에 대한 보안활동 연구를 활발히 진행하고 있으며, 이는 2011년 발표한 “안전한 사이버 미래를 위한 청사진(Blueprint for a Secure Cyber Future)”에 자세히 언급되어 있다.

국내의 경우 2009년도부터 전자정부 서비스를 중심으로 공공영역에서의 소프트웨어 개발보안 연구 및 정책이 본격적으로 추진되기 시작했다. 2020년 12월 10일에 소프트웨어진흥법 개정법이 시행됨에 따라 민간분야에서의 소프트웨어 개발보안 영역이 확대되었다. 과학기술정보통신부는 정보보호 패러다임 변화에 대응하고 안전하고 신뢰할 수 있는 디지털 안심 국가 실현을 목표로 2021년부터 중소기업 SW 보안약점 진단, 민간 특화 개발보안 가이드 보급, 개발보안 교육 등을 통해 민간 분야의 안전한 디지털 전환을 지원하고 있다. 2022년 2월에는 민간 분야에서 가장 많이 사용되는 파이썬 언어에 대한 시큐어코딩 가이드라인을 공개한 바 있다.

1) 출처: NIST(National Institute of Standards and Technology)

제2절 왜 자바스크립트인가

자바스크립트(Javascript)는 대화형 웹페이지 개발을 위해 만들어진 스크립트 기반 프로그래밍 언어로, 1995년 브랜담 엘크(Brendam Elch)가 넷스케이프 2 브라우저에 사용하기 위해 개발했다. 1997년도에 처음으로 언어 표준인 ECMA-262가 공개되었고 2022년 6월 기준 13번째 버전인 ECMAScript 2022(ES13) 버전이 공개됐다. 초기 자바스크립트는 C, Java와 같이 독립적인 애플리케이션 개발이 가능한 수준의 언어라기보다 웹 상의 데이터 표현을 위한 하나의 규격에 가까웠다. 또한, 자바스크립트는 브라우저 내에 탑재된 인터프리터에 종속되어 있어 확장성에도 제약이 따랐다. 하지만 지속적인 엔진 개선 및 표준화를 거치면서 현재는 명실상부 개발자들에게 가장 많은 사랑을 받는 언어 중 하나로 성장했다.

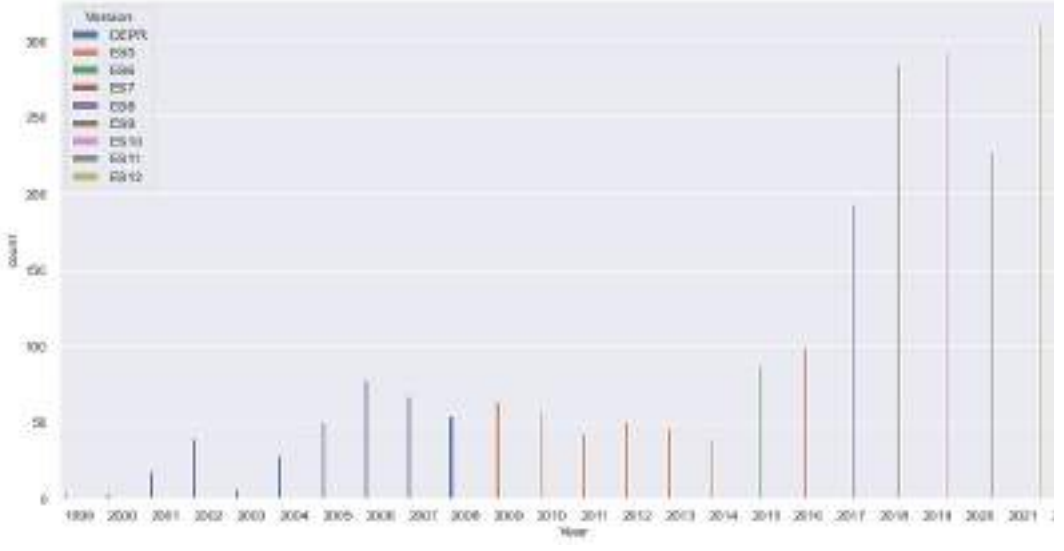
2009년 5월에는 오픈소스 자바스크립트 엔진인 크롬 V8에 비동기 이벤트 처리 라이브러리인 libuv를 결합한 NodeJS 플랫폼이 공개되면서 자바스크립트의 브라우저 종속성 문제가 해결되었고, 독립적인 애플리케이션으로서의 면모를 갖추게 되었다. 언어 자체의 유연함에 확장성까지 더해지면서 많은 개발자들이 자바스크립트 기반 플랫폼을 개발하기 시작했고, 하나의 언어로 풀스택(full-stack) 개발까지 가능한 단계까지 발전했다. 이제 자바스크립트 언어 하나만으로도 클라이언트 프로그램, 애플리케이션 서버, 모바일 앱, 임베디드 프로그램까지 개발이 가능하다.

글로벌 언어 트렌드에서도 자바스크립트의 인기를 확인할 수 있다. 구글 검색량을 기반으로 프로그래밍 언어 트렌드를 집계하는 PYPL에서는 Python, Java에 이어 3위를 차지했고, 전 세계 개발자들의 사랑을 받고 있는 대표적 플랫폼인 스택 오버플로우에서는 자바스크립트가 지난 9년 동안 가장 많이 사용되는 언어로 선정된 바 있다[2].

2) https://insights.stackoverflow.com/survey/2021#most-popular-technologies-language-prof

글로벌 프로그래밍 언어 선호도 트렌드 조사 결과(상: 스택오버플로우 / 하: PYPL)

언어 자체가 급격한 성장을 겪으면서 그만큼 더 높은 보안 위협에 노출되는 위험도 뒤따랐다. 대표적인 공개 취약점 데이터베이스인 CVE 통계치를 살펴보면 자바스크립트가 본격적으로 독립적인 언어의 모습을 갖추기 시작한 2015년경부터 자바스크립트 엔진에서 많은 보안 취약점이 발견되기 시작했고, 해마다 그 수준이 증가하고 있는 추세임을 확인 가능하다. 활용도가 높아지는 만큼 고려해야 할 보안적인 요소도 더욱 많아졌다고 볼 수 있다.



자바스크립트 버전 변화에 따라 발견된 취약점 개수 변화(출처: CVE)

제3절 가이드 목적 및 구성

본 가이드는 소프트웨어 구현 단계에서 발생 가능한 보안약점 제거 기준을 토대로 안전한 자바스크립트 개발을 위한 가이드 및 지침을 제공한다. 언어 엔진 자체의 취약점은 가이드 범위에 포함하지 않으며, 개발자가 통제 가능한 영역 내에서 안전한 코드 작성을 돕는 것을 목표로 한다.

대부분 보안약점 항목은 웹 애플리케이션을 타겟으로 하고 있으며, 자바스크립트로 클라이언트측과 서버측 모두 개발이 가능한 점을 고려해 최대한 풀스택 상황에서의 보안 가이드를 모두 제시하는 것을 목표로 한다. 기본적으로 순수 자바스크립트 언어 예시를 제공하며, 해당되는 경우 클라이언트 측 코드 예시는 ReactJS를, 서버 측 코드 예시는 NodeJS 기반 ExpressJS 예시 코드를 포함했다. 기본적으로 클라이언트 측 코드는 공격자가 충분한 리소스만 투입한다면 분석 및 변조가 가능하므로, 안전한 클라이언트 측 코드가 제시된 항목의 경우에도 반드시 서버측 보안 기능을 반드시 적용해야 함을 주의해야 한다.

구성

- **(1장) 자바스크립트 개발보안 가이드 개발 배경 및 목적**
- **(2장) 자바스크립트 언어 기반 구현단계 보안약점 제거 기준 설명**
  - 구현단계 보안약점 제거 기준 항목(49개) 중 40개에 대해 소개

유형	주요 내용
입력데이터 검증 및 표현	• SQL 삽입, 코드 삽입, 경로 조작 및 자원 삽입 등 13개 항목 ※ (4개 항목 제외) (정수형/메모리 버퍼) 오버플로우, HTTP 응답분할, 포맷 스트링 삽입
보안기능	• 적절한 인증 없는 중요 기능 허용, 부적절한 인가 등 16개 항목
시간 및 상태	• 종료되지 않는 반복문 또는 재귀함수 1개 항목 ※ (1개 항목 제외) 경쟁조건: 검사 시점과 사용 시점(TOCTOU)
에러처리	• 오류 메시지 정보노출, 오류상황 대응 부재 등 3개 항목
코드오류	• Null Pointer 역참조, 부적절한 자원 해제 등 2개 항목 ※ (3개 항목 제외) 해제된 자원 사용, 초기화되지 않은 변수 사용, 신뢰할 수 없는 데이터 역직렬화
캡슐화	• 잘못된 세션에 의한 데이터 정보노출 등 4개 항목
API 오용	• DNS lookup에 의한 보안결정 1개 항목 ※ (1개 항목 제외) 취약한 API 사용

- **(3장) 구현단계 보안약점 제거 기준 및 용어 설명**

2

PART 제2장

시큐어코딩 가이드

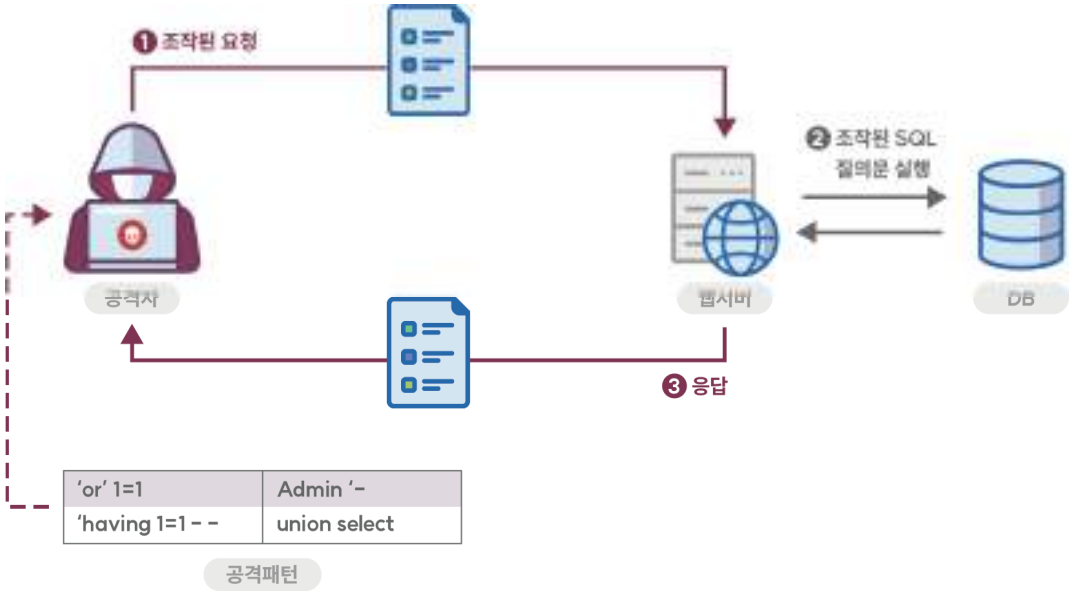
2 시큐어코딩 가이드

제1절 입력데이터 검증 및 표현

프로그램 입력값에 대한 검증 누락 또는 부적절한 검증, 데이터의 잘못된 형식지정, 일관되지 않은 언어셋 사용 등으로 인해 발생되는 보안약점으로 SQL 삽입, 크로스사이트 스크립트(XSS) 등의 공격을 유발할 수 있다.

1. SQL 삽입 VanillaJS - ReactJS - ExpressJS ✓

가. 개요



데이터베이스(DB)와 연동된 웹 응용프로그램에서 입력된 데이터에 대한 유효성 검증을 하지 않을 경우, 공격자가 입력 폼 및 URL 입력란에 SQL 문을 삽입하여 DB로부터 정보를 열람하거나 조작할 수 있는 보안약점을 말한다. 취약한 웹 응용프로그램에서는 사용자로부터 입력된 값을 검증 없이 넘겨받아 동적쿼리(Dynamic Query)를 생성하기 때문에 개발자가 의도하지 않은 쿼리가 실행되어 정보유출에 악용될 수 있다.

자바스크립트에서는 관계형 데이터베이스, NoSQL 등 다양한 유형의 데이터베이스 시스템과 상호작용할 수 있는 라이브러리를 제공하며, 크게 세 가지 유형의 방식을 사용할 수 있다.

- 데이터베이스 드라이버: 클라이언트와 커넥터를 사용해 데이터베이스와 직접 상호작용 ex) mysql
- 쿼리 빌더(Query builder): 데이터베이스 클라이언트보다 한 단계 높은 계층에서 동작하며, 자바스크립트 코드로 쿼리 데이터를 생성하고 데이터베이스와 상호작용 할 수 있음 ex) Knex.js
- ORM(Object Relational Mapping): 개발자가 데이터베이스를 추상화된 객체 형식으로 다룰 수 있게 해주는 데이터베이스 툴킷 ex) Sequelize

데이터베이스 드라이버를 사용할 경우 개발자가 직접 쿼리 문자열을 정의하고, 그 결과를 그대로 데이터베이스에 질의하게 되는데, 이 경우 검증되지 않은 외부 입력값으로 인한 SQL 삽입 공격이 발생할 수 있다. ORM을 사용하는 경우에도 복잡한 조건의 쿼리문 생성 어려움, 성능 저하 등의 이유로 ORM에서 지원하는 원시 쿼리 기능을 사용하면 공격에 취약해질 수 있다. SQL 인젝션 공격은 데이터베이스와 직접 상호작용하는 서버 측 코드에서만 발생할 수 있다.

나. 안전한 코딩기법

사용자 입력값으로 쿼리를 생성하는 경우 쿼리 빌더를 사용해 SQL 인젝션 공격을 방어할 수 있다. 입력값 검증은 사용자가 전달한 값이 쿼리문 구성에 필요한 정보를 가지는지 검증하는 화이트리스트 기반 방식으로, 구현이 간단하지만 특수 문자를 필요로 하는 상황이나 복잡한 동적 쿼리의 경우 데이터베이스 구조에 따라 필터링 규칙을 조금씩 변형해야 하며, 부득이한 경우 공격에 사용되는 문자를 필터링해서는 안 되는 경우가 발생할 수 있다.

데이터베이스 드라이버를 사용하거나 ORM에서 지원하는 원시 쿼리기능 사용 시 인자화된 쿼리를 통해 외부 입력값을 바인딩해서 사용하면 SQL 삽입 공격으로부터 안전하게 보호할 수 있다. 인자화된 쿼리는 사용자가 전달한 입력값을 그대로 쿼리 문자열로 만들지 않고, DB API에서 제공하는 기능을 사용해 쿼리 내에 사용자 입력값을 구성하는 방법을 의미한다. 인자화된 쿼리를 사용하면 사용자가 전달한 값을 그대로 쿼리에 사용하지 않고 미리 컴파일 된 쿼리 템플릿에 값을 삽입하는 방법을 사용해 입력값으로 인한 쿼리 조작이 불가능하게 된다.

다. 코드 예제

가) 데이터베이스 드라이버 사용 예제

다음은 사용자 입력값을 받아 원시 쿼리를 구성해 처리 하는 안전하지 않은 코드 예시를 보여 준다. 클라이언트에서 GET 요청의 인자로 전달된 id 값을 검증 없이 그대로 쿼리에 삽입하므로, 공격자가 정상적인 정수형 id 값 대신 '1 UNION SELECT group_concat(table_name) FROM information_schema.tables WHERE table_schema = database();' 와 같은 공격문을 전달할 경우 데이터베이스 내의 모든 테이블 이름이 조회되고, 이는 곧 공격자가 원하는 모든 쿼리를 실행 가능하다는 의미로 해석될 수 있다.

안전하지 않은 코드 예시

```
1:  const mysql = require("mysql");
2:
3:  // 커넥션 초기화 옵션은 생략함
4:  const connection = mysql.createConnection(...);
5:
6:  router.get("/vuln/email", (req, res) => {
7:    const con = connection;
8:    const userInput = req.query.id;
9:    // 사용자로부터 입력받은 값을 검증 없이 그대로 쿼리에 사용
10:   const query = `SELECT email FROM user WHERE user_id = ${userInput}`;
11:
12:   con.query(query,
13:     (err, result) => {
14:       if (err) console.log(err);
15:       return res.send(result);
16:     }
17:   );
18: });
```

데이터베이스 라이브러리가 인자화된 쿼리 기능을 지원할 경우 해당 기능을 사용해 쿼리를 생성해야 한다. mysql 라이브러리에서는 인자화된 쿼리 기능은 제공하지 않지만 '?' 형식으로 유사한 기능을 구현할 수 있다. 다음은 위 코드를 안전한 코드로 변환한 예시를 보여준다.

안전한 코드 예시

```
1:  const mysql = require("mysql");
2:  ...
3:  router.get("/patched/email", (req, res) => {
4:    const con = connection;
5:    const userInput = req.query.id;
6:    const query = 'SELECT email FROM user WHERE user_id = ?';
7:
8:    // 쿼리 함수에 사용자 입력값을 매개변수 형태로 전달, 이렇게 작성하면 사용자 입력값에
9:    // escape 처리를 한 것과 동일한 결과가 실행
10:   con.query(query, userInput,
11:     (err, result) => {
12:       if (err) console.log(err);
13:       return res.send(result);
14:     }
15:   );
16: });
```

나) ORM 사용 예제

ExpressJS에서 가장 일반적으로 사용하는 ORM 라이브러리인 Sequealize에서는 쿼리 인자화를 사용해 쿼리를 구성하기 때문에 SQL 삽입 공격으로부터 안전하다. 부득이하게 원시 SQL 또는 사용자 정의 SQL을 사용할 경우에도 외부 입력값을 인자화된 쿼리의 바인딩 변수로 사용하면 된다.

다음 코드는 ORM에서 지원하는 인자화된 쿼리 기능을 사용하지 않고 사용자 입력값을 그대로 쿼리에 삽입해 사용하는 예시를 보여 준다. 취약점 발생 원인과 가능한 공격 범위는 앞서 살펴본 mysql 취약 코드 예시와 동일하다.

안전하지 않은 코드 예시

```
1:  const mysql = require("mysql");
2:  const Sequelize = require("sequelize");
3:  const { QueryTypes } = require("sequelize");
4:
5:  // 커넥션 및 ORM 초기화 옵션은 생략함
6:  const connection = mysql.createConnection(...);
7:  const sequelize = new Sequelize(...);
```

```
8:
9:  router.get("/vuln/orm/email", (req, res) => {
10:   const userInput = req.query.id;
11:   // 사용자로부터 입력받은 값을 검증 없이 그대로 쿼리에 사용
12:   const query = `SELECT email FROM user WHERE user_id = ${userInput}`;
13:
14:   sequelize.query(query, { type: QueryTypes.SELECT })
15:     .then((result) => {
16:       return res.send(result);
17:     }).catch((err) => {
18:       console.log(err);
19:     });
20: });
```

다음 코드에서는 원시 코드 실행 시에도 인자화된 쿼리와 인자를 바인딩 후 사용하는 안전한 예시를 보여 준다. 쿼리에 삽입할 값은 $number로 구분한다(예를 들어, 쿼리 구성에 사용하는 사용자 입력값이 두 개인 경우 각각 $1, $2 지시어 사용).

안전한 코드 예시

```
1:  ....
2:
3:  router.get("/vuln/orm/email", (req, res) => {
4:    const userInput = req.query.id;
5:    // 쿼리 내에서 바인딩이 필요한 부분을 $number로 표기
6:    const query = `SELECT email FROM user WHERE user_id = $1`;
7:
8:    // 인자화된 쿼리 기능을 통해 쿼리를 생성 및 실행
9:    sequelize.query(query,
10:     {
11:       bind: [userInput],
12:       type: QueryTypes.SELECT
13:     })
14:     .then((result) => {
15:       return res.send(result);
16:     }).catch((err) => {
17:       console.log(err);
18:     });
19: });
```

라. 참고자료

① CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection'), MITRE, https://cwe.mitre.org/data/definitions/89.html

② SQL Injection Prevention Cheat Sheet, OWASP
https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html

③ The Javascript Guide: Web Application Secure Coding Practices
https://github.com/Checkmarx/JS-SCP/blob/master/dist/js-webapp-scp.pdf

④ JavaScript Secure Coding Standard - Ministry of Transport and Communications
https://compliance.qcert.org/sites/default/files/library/2018-10/MOTC-CIPD_JavaScript_Coding_Standard(US).pdf

⑤ Raw Queries, Sequelize
https://sequelize.org/docs/v6/core-concepts/raw-queries/

⑥ mysql, npm
https://www.npmjs.com/package/mysql

⑦ SQL Expression Language Tutorial, SQLAlchemy
https://docs.sqlalchemy.org/en/14/core/tutorial.html#using-textual-sql

2. 코드 삽입 VanillaJS - ReactJS - ExpressJS ✓

가. 개요

공격자가 소프트웨어의 의도된 동작을 변경하도록 임의 코드를 삽입해 소프트웨어가 비정상적으로 동작하도록 하는 보안약점을 말한다. 코드 삽입은 프로그래밍 언어 자체의 기능에 한해 이뤄진다는 점에서 운영체제 명령어 삽입과 다르다. 프로그램에서 사용자의 입력값 내에 코드가 포함되는 것을 허용할 경우, 공격자는 개발자가 의도하지 않은 코드를 실행해 권한을 탈취하거나 인증 우회, 시스템 명령어 실행 등으로 이어질 수 있다.

자바스크립트에서 코드 삽입 공격을 유발할 수 있는 함수로는 eval(), setTimeOut(), setInterval() 등이 있다. 해당 함수의 인자를 면밀히 검증하지 않는 경우 공격자가 전달한 코드가 그대로 실행될 수 있다. 브라우저 위에서 동작하는 클라이언트측 환경에서는 주요 자원을 포함하는 서버와 달리 코드 삽입 자체만으로는 위협이 되지 않는다.

나. 안전한 코딩기법

동적 코드를 실행할 수 있는 함수를 사용하지 않는다. 필요 시, 실행 가능한 동적 코드를 입력값으로 받지 않도록 외부 입력값에 대해 화이트리스트 기반 검증을 수행해야 한다. 유효한 문자만 포함하도록 동적 코드에 사용되는 사용자 입력값을 필터링 하는 방법도 있다. 또한, 데이터와 명령어를 엄격히 분리해 처리하는 별도의 로직을 구현할 수도 있다.

다. 코드예제

다음은 안전하지 않은 코드로 eval()을 사용해 사용자로부터 입력받은 값을 실행하여 결과를 반환 하는 예제다. 외부로부터 입력 받은 값을 아무런 검증 없이 eval() 함수의 인자로 사용하고 있다. 외부 입력값을 검증 없이 사용할 경우 공격자는 자바스크립트 코드를 통해 악성 기능 실행을 위한 라이브러리 로드 및 원격 대화형 쉘 등을 실행할 수도 있다.

안전하지 않은 코드 예시

```
1:  const express = require('express');
2:
3:  router.post("/vuln/server", (req, res) => {
4:    // 사용자로부터 전달 받은 값을 그대로 eval 함수의 인자로 전달
5:    const data = eval(req.body.data);
6:    return res.send({ data });
7:  });
```

다음은 안전한 코드로 변환한 예제를 보여 준다. 외부 입력값 내에 포함된 특수문자 등을 필터링 하는 사전 검증 코드를 추가하면 코드 삽입 공격 위험을 완화할 수 있다. 만약 부득이하게 클라이언트 요청으로 코드 실행을 해야 하는 경우 화이트리스트 기반 필터링을 통해 실행이 허용된 명령어가 포함된 입력값만 함수의 인자로 전달해야 한다.

안전한 코드 예시

```
1:  const express = require('express');
2:
3:  function alphanumeric(input_text) {
4:    // 정규표현식 기반 문자열 검사
5:    const letterNumber = /^[0-9a-zA-Z]+$/;
6:
7:    if (input_text.match(letterNumber)) {
8:      return true;
9:    } else {
10:     return false;
11:   }
12: }
13:
14: router.post("/patched/server", (req, res) => {
15:   let ret = null;
16:   const { data } = req.body;
17:   // 사용자 입력을 영문, 숫자로 제한하며, 만약 입력값 내에 특수문자가 포함되어
18:   // 있을 경우 에러 메시지를 반환
19:   if (alphanumeric(data)) {
20:     ret = eval(data);
21:   } else {
22:     ret = 'error';
23:   }
24:   return res.send({ ret });
25: });
```

라. 참고자료

① CWE-94: Improper Control of Generation of Code ('Code Injection'), MITRE, https://cwe.mitre.org/data/definitions/94.html

② CWE-95: Improper Neutralization of Directives in Dynamically Evaluated Code ('Eval Injection'), MITRE, https://cwe.mitre.org/data/definitions/95.html

③ Code Injection, OWASP, https://owasp.org/www-community/attacks/Code_Injection

④ JavaScript: HTML Form - checking for numbers and letters https://www.w3resource.com/javascript/form/letters-numbers-field.php

⑤ The Javascript Guide: Web Application Secure Coding Practices https://github.com/Checkmarx/JS-SCP/blob/master/dist/js-webapp-scp.pdf

⑥ NodeJs-Secure Code wiki https://securecode.wiki/docs/lang/nodejs/

3. 경로 조작 및 자원 삽입 VanillaJS - ReactJS - ExpressJS ✓

가. 개요

검증되지 않은 외부 입력값을 통해 파일 및 서버 등 시스템 자원(파일, 소켓의 포트 등)에 대한 접근 혹은 식별을 허용할 경우, 입력값 조작으로 시스템이 보호하는 자원에 임의로 접근할 수 있는 보안약점이다. 경로조작 및 자원삽입 약점을 이용해 공격자는 자원 수정·삭제, 시스템 정보누출, 시스템 자원 간 충돌로 인한 서비스 장애 등을 유발시킬 수 있다. 또한, 경로 조작 및 자원 삽입을 통해서 공격자가 허용되지 않은 권한을 획득해 설정 파일을 변경하거나 실행시킬 수 있다.

기본적으로, 클라이언트측 자바스크립트에서는 외부 입력값을 기반으로 하는 호스트 시스템의 프로세스 제어, 파이프 상호작용, 소켓 연결 등을 특별한 경우가 아니면 거의 사용하지 않으며, 사용하는 경우라도 코드 실행 위치가 사용자 시스템이므로 경로 조작이나 자원 삽입과 같은 위협이 큰 의미를 가지지 못한다. NodeJS 기반 서버 프로그램의 경우 fs 및 socket.io 라이브러리를 통해 프로세스 및 소켓 연결을 제어할 수 있다.

나. 안전한 코딩기법

외부로부터 받은 입력값을 자원(파일, 소켓의 포트 등)의 식별자로 사용하는 경우 적절한 검증을 거치도록 하거나, 사전에 정의된 리스트에 포함된 식별자만 사용하도록 해야 한다. 특히, 외부의 입력이 파일명인 경우에는 필터를 적용해 경로순회(directory traversal) 공격의 위험이 있는 문자(/, \, .. 등)를 제거해야 한다.

다. 코드예제

가) 경로 조작 예제

다음은 외부 입력값으로 파일 경로 등을 입력받아 파일을 여는 예시를 보여 준다. 만약 공격자가 '../../../../etc/passwd' 와 같은 값을 전달하면 사용자 계정 및 패스워드 정보가 담긴 파일의 내용이 클라이언트 측에 표시되어 의도치 않은 시스템 정보노출 문제가 발생한다.

안전하지 않은 코드 예시

```
1:  const express = require('express');
2:  const path = require('path');
3:
4:  router.get("/vuln/file", (req, res) => {
5:    // 외부 입력값으로부터 파일명을 입력 받음
6:    const requestFile = req.query.file;
7:
8:    // 입력값을 검증 없이 파일 처리에 사용
9:    fs.readFile(path.resolve(__dirname, requestFile), 'utf8', function(err, data) {
10:     if (err) {
11:       return res.send('error');
12:     }
13:     return res.send(data);
14:   })
15: });
```

외부 입력값에서 경로 조작 문자열(/, \, .. 등)을 제거한 후 파일의 경로 설정에 사용하면 코드를 안전하게 만들 수 있다.

안전한 코드 예시

```
1:  const express = require('express');
2:  const path = require('path');
3:
4:  router.get("/patched/file", (req, res) => {
5:    const requestFile = req.query.file;
6:    // 정규표현식을 사용해 사용자 입력값을 필터링
7:    const filtered = requestFile.replace(/[.\\\/]/gi, '');
8:
9:    fs.readFile(filtered, 'utf8', function(err, data) {
10:     if (err) {
11:       return res.send('error');
12:     }
13:     return res.send(data);
14:   })
15: });
```

나) 자원 삽입 예제

다음은 안전하지 않은 코드 예시로, 외부 입력을 소켓 연결 주소로 그대로 사용하고 있다. 외부 입력값을 검증 없이 사용할 경우 기존 자원과의 충돌로 의도치 않은 에러가 발생할 수 있다.

안전하지 않은 코드 예시

```
1:  const express = require('express');
2:  const io = require("socket.io");
3:
4:  router.get("/vuln/socket", (req, res) => {
5:    try {
6:      // 외부로부터 입력받은 검증되지 않은 주소를 이용하여
7:      // 소켓을 바인딩 하여 사용하고 있어 안전하지 않음
8:      const socket = io(req.query.url);
9:      return res.send(socket);
10:   } catch (err) {
11:     return res.send("[error] fail to connect");
12:   }
13: });
```

다음은 안전한 예제를 보여 준다. 내부 자원에 접근 시 외부에서 입력 받은 값을 소켓 연결 주소와 같은 식별자로 그대로 사용하는 것은 바람직하지 않으며, 꼭 필요한 경우엔 허용 가능한 목록을 설정한 후 목록 내에 포함된 주소만 연결을 허용하도록 코드를 작성해야 한다.

안전한 코드 예시

```
1:  const express = require('express');
2:  const io = require("socket.io");
3:
4:  router.get("/patched/socket", (req, res) => {
5:    // 화이트리스트 내에 속하는 주소만 허용
6:    const whitelist = ["ws://localhost", "ws://127.0.0.1"];
7:    if (whitelist.indexOf(req.query.url) < 0) {
8:      return res.send("wrong url");
9:    }
10:   try {
11:     const socket = io(req.query.url);
12:     return res.send(socket);
13:   } catch (err) {
14:     return res.send("[error] fail to connect");
15:   }
16: });
```

라. 참고자료

① CWE-22: Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal'), MITRE, https://cwe.mitre.org/data/definitions/22.html

② CWE-99: Improper Control of Resource Identifiers ('Resource Injection'), MITRE, https://cwe.mitre.org/data/definitions/99.html

③ Path Traversal, OWASP, https://owasp.org/www-community/attacks/Path_Traversal

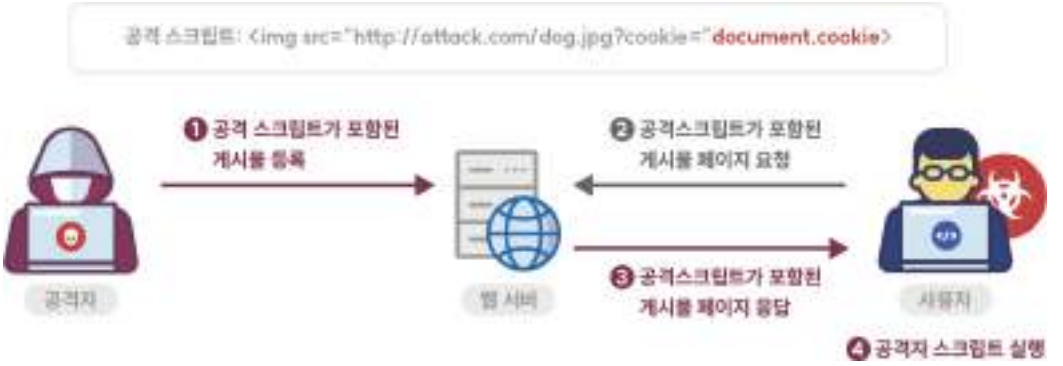
④ Resource Injection, OWASP, https://owasp.org/www-community/attacks/Resource_Injection

⑤ Fortify Taxonomy: Software Security Errors https://vulncat.fortify.com/en/detail?id=desc.dataflow.java.resource_injection#JavaScript%2FTypeScript

⑥ socket.io https://socket.io/docs/v4/

4. 크로스사이트 스크립트(XSS) VanillaJS ✓ ReactJS ✓ ExpressJS ✓

가. 개요



크로스사이트 스크립트 공격(Cross-site scripting Attacks)은 웹사이트에 악성코드를 삽입하는 공격 방법이다. 공격자는 대상 웹 응용프로그램의 결함을 이용해 악성코드(일반적으로 클라이언트 측 자바스크립트 사용)를 사용자에게 보낸다. XSS 공격은 일반적으로 애플리케이션 호스트 자체보다 사용자를 목표로 삼는다.

XSS는 공격자가 웹 응용프로그램을 속여 브라우저에서 실행될 수 있는 형식의 데이터(코드)를 다른 사용자에게 전달할 때 발생한다. 공격자가 임의로 구성한 기본 웹 코드 외에도 악성코드 다운로드, 플러그인 또는 미디어 콘텐츠를 이용할 수도 있다. 사용자가 폼 양식에 입력한 데이터 또는 서버에서 클라이언트 단말(브라우저) 전달된 데이터가 적절한 검증 없이 사용자에게 표시되도록 허용되는 경우 발생한다.

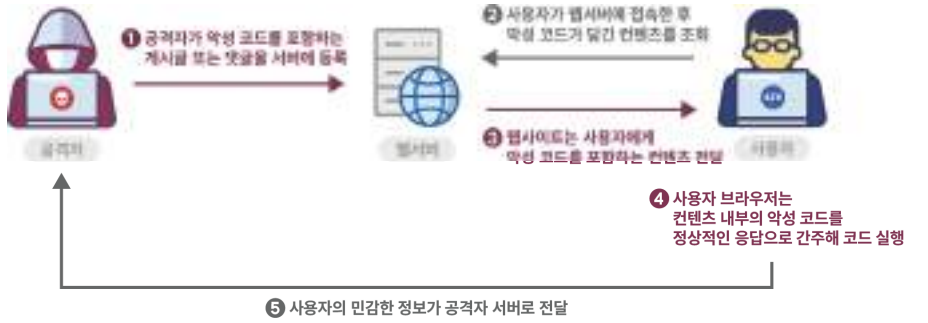
XSS 공격에는 크게 세 가지 유형의 방식이 있다.

• 유형 1 : Reflective XSS (or Non-persistent XSS)

- Reflective XSS는 공격 코드를 사용자의 HTTP 요청에 삽입한 후, 해당 공격 코드를 서버 응답 내용에 그대로 반사(Reflected)시켜 브라우저에서 실행하는 공격 기법이다. Reflective XSS 공격을 수행하려면 사용자로 하여금 공격자가 만든 서버로 데이터를 보내도록 해야 한다. 이 방법은 보통 악의적으로 제작된 링크를 사용자가 클릭하도록 유도하는 방식을 수반한다. 공격자는 피해자가 취약한 사이트를 참조하는 URL을 방문하도록 유도하고, 피해자가 링크를 방문하면 스크립트가 피해자의 브라우저에서 자동으로 실행된다. 대부분의 경우, Reflective XSS 공격 메커니즘은 공개 게시판, 피싱(Phishing) 이메일, 단축 URL 또는 실제와 유사한 URL을 사용한다.

• **유형 2 : Persistent XSS (or Stored XSS)**

- Persistent XSS는 신뢰할 수 없거나 확인되지 않은 사용자 입력(코드)이 서버에 저장되고, 이 데이터가 다른 사용자들에게 전달될 때 발생한다. Persistent XSS는 게시글 및 댓글 또는 방문자 로그 기능에서 발생할 수 있다. 해당 기능을 통해 공격자의 악성 콘텐츠를 다른 사용자들이 열람할 수 있다. 소셜 미디어 사이트 및 회원 그룹에서 흔히 볼 수 있는 것과 같이 공개적으로 표시되는 프로필 페이지는 Persistent XSS의 대표적인 공격 대상 중 하나다. 공격자는 프로필 입력 폼에 악성 스크립트를 주입해 다른 사용자가 프로필을 방문하면 브라우저에서 자동으로 코드가 실행되도록 할 수 있다.



• **유형 3 : DOM XSS (or Client-Side XSS)**

- DOM XSS은 웹 페이지에 있는 사용자 입력값을 적절하게 처리하기 위한 자바스크립트의 검증 로직을 무효화하는 것을 목표로 한다. 공격 스크립트가 포함된 악성 URL을 통해 전달된다는 관점에서 Reflective XSS와 유사하다고 볼 수 있다. 그러나 신뢰할 수 있는 사이트의 HTTP 응답에 페이로드를 포함하는 대신 DOM 또는 문서 개체 모델을 수정해 브라우저와 독립적인 공격을 실행한다는 점에서 차이가 있다.

- 공격자는 DOM XSS 공격을 통해 세션 및 개인 정보를 포함한 쿠키 데이터를 피해자의 컴퓨터에서 공격자 시스템으로 전송할 수 있다. 이 정보를 사용해 특정 웹사이트에 악의적인 요청을 보낼 수 있으며, 피해자가 해당 사이트를 관리 할 수 있는 관리자 권한이 있는 경우 심각한 위협을 초래할 수도 있다. 또한, 신뢰할 수 있는 웹 사이트를 모방하고 피해자가 암호를 입력하도록 속여 공격자가 해당 웹 사이트에서 피해자의 계정을 손상시키는 피싱(Phishing) 공격으로도 이어질 수 있다.

웹 서비스 개발에 사용되는 다른 언어와 달리, 자바스크립트는 클라이언트측 코드와 서버측 코드 모두 개발이 가능한 언어로, XSS 공격 예방을 위해 클라이언트와 서버 양쪽 모두를 고려해야 한다. 서버가 생성한 HTML 응답 데이터 내에 신뢰할 수 없는 데이터가 포함되거나, 위험한 자바스크립트 호출을 통해 DOM을 업데이트 하는데 사용되는 입력값이 전달될 경우 XSS 공격이 발생하게 된다.

나. 안전한 코딩기법

외부 입력값 또는 출력값에 스크립트가 삽입되지 못하도록 문자열 치환 함수를 사용하여 &〈〉*‘’/() 등을 & < > " ' / ( )로 치환하거나, 자바스크립트 라이브러리에서 제공하는 escape 기능을 사용해 문자열을 변환해야 한다. 자바스크립트에서 기본적으로 제공하는 escape() 함수는 deprecated되어 더 이상 사용을 권장하지 않고 있으며, 그 대신 encodeURI() 또는 encodeURIComponent() 함수를 사용하면 된다. HTML 태그를 허용해야 하는 게시판에서는 허용할 HTML 태그들을 화이트리스트로 만들어 해당 태그만 지원하도록 한다.

XSS 방어 코드는 클라이언트측과 서버측에 모두 적용해야 한다. 물론 클라이언트측 코드는 공격자가 중간에 가로채 변조하거나 코드를 수정할 수 있지만 서버로 데이터가 넘어가기 전 1차적인 방어선 역할을 해줄 수 있다. 클라이언트측 코드를 조작하더라도 서버에서 사용자 입력값을 검증하면 XSS 공격을 예방할 수 있다. 마지막으로, 서버로부터 가져온 데이터를 클라이언트측에서 다시 사용자에게 보여줄 때 코드가 포함된 내용일 경우에도 공격에 사용되는 문자열을 필터링해야 한다.

다. 코드예제

가) 클라이언트측 VanillaJS 예제

클라이언트측 코드 내에 사용자가 전달한 입력값을 코드 실행의 일부분으로 사용하거나, 사용자가 작성한 텍스트 내부의 스크립트 코드를 실행할 수 있는 기능이 제공될 경우 XSS 공격이 발생할 수 있다. 다음 예제는 사용자가 폼 또는 에디터로 작성한 내용을 서버에 저장하고, 저장된 내용을 다시 서버로부터 받아와 DOM에 출력하는 예시를 보여 준다. 이 때, 서버로부터 받은 데이터를 검증 없이 그대로 사용할 경우 악성 스크립트가 실행될 수 있다.

안전하지 않은 코드 예시

```
1:  〈html〉
2:  〈body〉
3:    〈script〉
4:      const query = "〈script〉alert('hello world')〈"+"/script〉";
5:      async function req() {
6:        // 사용자가 에디터와 같은 입력 폼에 입력한 데이터를 서버에 저장
7:        const response = await fetch(`/vuln/search?q=${query}`, { method: 'GET' })
8:        const data = await response.text();
9:        // 외부로부터 받은 데이터(HTML 코드)를 아무런 검증 없이 DOM으로 기록
10:       document.write(data);
11:     }
12:     req();
13:   〈/script〉
14: 〈/body〉
15: 〈/html〉
```

자바스크립트에서는 encodeURI(), encodeURIComponent() 함수를 통해 이스케이프 기능을 제공한다. 외부로부터 받은 데이터를 사용해 DOM에 변화를 주는 코드가 실행되기 전에 입력값을 이스케이프 해주면 XSS 공격을 막을 수 있다. 이스케이프 후 decodeURI() 함수를 호출해야 스크립트 실행 방지와 함께 원하는 결과를 DOM에 출력할 수 있다.

encodeURIComponent()만 사용해 html 데이터를 이스케이프 한 결과와(상), 이스케이프 결과를 다시 decodeURI()한 결과(하)

안전한 코드 예시 1 (내장함수 사용)

```
1:  <html>
2:  <body>
3:    <script>
4:      const query = "<script>alert('hello world')<"+"/script>";
5:      async function req() {
6:        const response = await fetch(`/vuln/search?q=${query}`, { method: 'GET' })
7:        const data = await response.text();
8:
9:        // 외부로부터 받은 데이터를 이스케이프 처리 후 사용
10:       document.write(decodeURI(encodeURIComponent(data)));
11:     }
12:     req();
13:   </script>
14: </body>
15: </html>
```

encodeURIComponent 함수는 내장 함수로 사용이 쉽고 가볍다는 장점이 있지만, 모든 XSS 공격 패턴에 대응할 수는 없다. 따라서, XSS 검증을 위한 별도의 필터링 함수를 구현하거나, 개발자 커뮤니티에서 널리 사용되고, 안정적이며, 지속적으로 관리되고 있는 라이브러리를 사용해 문자열을 변환해야 한다. 아래 예시는 xss-filters 라이브러리를 활용한 필터링 사례를 보여 준다.

안전한 코드 예시 2 (라이브러리 사용)

```
<html>
  <head>
    <script src="https://cdn.rawgit.com/yahoo/xss-filters/master/dist/xss-filters.js"></script>
  </head>
  <body>
    <script>
      async function req() {
      ...
        // xss-filters 라이브러리를 사용해 문자열을 이스케이프 처리
        document.write(xssFilters.inHTMLData(data));
      }
      req();
    </script>
  </body>
</html>
```

나) 클라이언트측 ReactJS 예제

ReactJS에서는 JSX(타입스크립트에서는 TSX) 확장자를 가지는 리액트 엘리먼트를 통해 코드를 처리한다. 기본적으로, React DOM은 JSX 렌더링 전에 코드 내의 모든 값을 이스케이프 처리한다. 다시 말해서, 리액트 애플리케이션 내에서 명시적으로 악성 스크립트를 주입하는 것은 불가능하다. ReactJS에서 강제하는 규칙을 따르는 것만으로도 클라이언트측 XSS 공격을 방어할 수 있다.

하지만 ReactJS는 HTML 입력값을 그대로 렌더링할 수 있는 dangerouslySetInnerHTML 함수를 제공한다. 만약 개발자가 사용자 게시글 표현 등을 위해 이 함수를 사용할 경우 XSS 공격에 노출될 수 있다.

안전하지 않은 코드 예시

```
function possibleXSS() {
  return {
    __html:
      '<img
        src="https://upload.wikimedia.org/wikipedia/commons/a/a7/React-icon.svg"
        onload="alert(1)">
      </img>',
    };
  }

const App = ( ) => (
  // XSS에 취약한 함수를 사용해 HTML 코드 데이터를 렌더링
  <div dangerouslySetInnerHTML={possibleXSS()} />
);
ReactDOM.render(<App />, document.getElementById("root"));
```

가급적이면 dangerouslySetInnerHTML 함수를 사용하지 않는 것이 좋겠지만, 서비스 개발을 위해 부득이하게 사용이 필요한 경우 HTML 및 자바스크립트 코드를 직접 이스케이프 처리하는 컴포넌트를 별도로 개발하거나, dompurify와 같이 이스케이프 기능을 제공하는 라이브러리를 사용해 문자열 처리 후 사용해야 한다.

안전한 코드 예시

```
1:  <script src="https://cdnjs.cloudflare.com/ajax/libs/dompurify/2.4.0/purify.min.js"></script>
2:  ...
3:  function possibleXSS() {
4:    return {
5:      __html:
6:        // dompurify 라이브러리를 사용해 입력값을 이스케이프 처리
7:        DOMPurify.sanitize('<img src="https://upload.wikimedia.org/wikipedia/commons/
8:        a/a7/React-icon.svg" onload="alert(1)"></img>'),
9:      };
10: }
11: const App = ( ) => (
12:   <div dangerouslySetInnerHTML={possibleXSS()} />
13: );
14: ReactDOM.render(<App />, document.getElementById("root"));
```

다) 서버측 ExpressJS 예제

클라이언트측에서 사용자 입력값을 검증 및 이스케이프 처리했다고 하더라도, 반드시 서버측에서도 검증 코드를 추가해야 한다. 게시판, 고객 문의와 같이 사용자로부터 다양한 형식의 데이터를 입력받아야 하는 경우 사용자가 입력한 텍스트뿐만 아니라 텍스트를 둘러싸고 있는 HTML 및 자바스크립트 코드도 함께 서버에 저장하게 된다.

이 때 사용자가 입력한 데이터를 서버측에서 검증 없이 데이터베이스에 저장할 경우 해당 게시글 또는 문의글을 열람하는 모든 사용자의 시스템에서 악성 스크립트가 실행될 수 있다. 다음은 사용자로부터 검색 대상을 전달받아 그 결과를 반환하는 취약한 예시를 보여 준다.

안전하지 않은 코드 예시

```
1:  const express = require('express');
2:
3:  router.get("/vuln/search", (req, res) => {
4:    // 사용자로부터 전달 받은 쿼리 데이터로 데이터 조회
5:    const results = selectFromDB(req.query.q);
6:
7:    if (results.length === 0) {
8:      // 검색 결과가 발견되지 않을 경우, '요청한 값'을 찾지 못했다는 메시지를 반환하는데,
9:      // 이 때 정상적인 질의문이 아닌 악성 스크립트가 포함된 데이터를 입력 받은 경우라면
10:     // 클라이언트측에서 악성 스크립트가 실행될 수 있음
11:     return res.send('<p>No results found for "' + req.query.q + '"</p>');
12:   }
13: ...
14: });
```

서버측에서 XSS 공격을 방어하는 방법은 클라이언트측과 동일하다. 사용자 입력값에 HTML 및 자바스크립트 코드가 포함되는 경우 XSS 공격에 사용되는 패턴을 필터링하는 별도의 함수를 구현하거나 라이브러리를 사용해 안전하게 처리한 후 데이터베이스에 저장하거나 사용자에게 다시 반환해 주어야 한다.

안전한 코드 예시

```
1:  const express = require('express');
2:  const xssFilters = require("xss-filters");
3:  ...
4:  router.get("/patched/search", (req, res) => {
5:    const unsafeFirstname = req.query.q;
6:    // xss-filters 라이브러리를 사용해 사용자 입력값을 이스케이프 처리
7:    const safeFirstname = xssFilters.inHTMLData(unsafeFirstname);
8:    const results = selectFromDB(safeFirstname);
9:
10:   if (results.length === 0) {
11:     res.send(util.format("<p>Hello %s</p>", safeFirstname));
12:   }
13: ...
14: });
```

라. 참고자료

① CWE-79: Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting'), MITRE, https://cwe.mitre.org/data/definitions/79.html

② Cross Site Scripting (XSS), OWASP, https://owasp.org/www-community/attacks/xss/

③ encodeURIComponent, mdn web docs https://developer.mozilla.org/ko/docs/Web/JavaScript/Reference/Global_Objects/encodeURIComponent

④ The Javascript Guide: Web Application Secure Coding Practices https://github.com/Checkmarx/JS-SCP/blob/master/dist/js-webapp-scp.pdf

⑤ NodeJs-Secure Code wiki https://securecode.wiki/docs/lang/nodejs/

5. 운영체제 명령어 삽입 VanillaJS - ReactJS - ExpressJS ✓

가. 개요

적절한 검증 절차를 거치지 않은 사용자 입력값이 운영체제 명령어의 일부 또는 전부로 구성되어 실행되는 경우, 의도하지 않은 시스템 명령어가 실행돼 부적절하게 권한이 변경되거나 시스템 동작 및 운영에 악영향을 미칠 수 있다.

명령어 라인의 파라미터나 스트림 입력 등 외부 입력을 사용해 시스템 명령어를 생성 하는 프로그램을 많이 찾아볼 수 있다. 이 경우 프로그램 외부로부터 받은 입력 문자열은 기본적으로 신뢰할 수 없기 때문에 적절한 처리를 해주지 않으면, 공격으로 이어질 수 있다.

NodeJS에서는 child_process 라이브러리를 사용해 코드 내부에서 시스템 명령어를 실행할 수 있다. 만약 시스템 명령어 실행 함수에 사용자가 입력한 값을 전달할 수 있고, 그 값이 적절히 검증되지 않을 경우 패스워드 파일 조회, 시스템 프로세스 강제 종료 등 의도하지 않은 악의적인 명령어 실행으로 이어질 수 있다.

나. 안전한 코딩기법

외부 입력값 내에 시스템 명령어를 포함하는 경우 |, ;, &, :, >, <, `(backtick), \, ! 과 같이 멀티라인 및 리다이렉트 문자 등을 필터링 하고 명령을 수행할 파일명과 옵션을 제한해 인자로만 사용될 수 있도록 해야 한다. 외부 입력에 따라 명령어를 생성하거나 선택이 필요한 경우에는 명령어 생성에 필요한 값들을 미리 지정해 놓고 사용해야 한다.

다. 코드예제

다음 예제는 사용자에게 특정 경로의 파일 목록을 제공하는 프로그램 예시를 보여 준다. 만약 경로값 안에 파이프라인 명령어가 포함될 경우 악의적인 명령어가 실행될 수 있다 (ex) /vuln?path=/usr/app | cat /etc/passwd).

안전하지 않은 코드 예시

```
1:  const express = require('express');
2:  const child_process = require("child_process");
3:
4:  router.get("/vuln", (req, res) => {
5:    // 사용자가 입력한 명령어 인자값을 검증 없이 사용해 의도치 않은 추가 명령어 실행 가능
6:    child_process.exec("ls -l " + req.query.path, function (err, data) {
7:      return res.send(data);
8:    });
9:  });
```

임의의 명령어 실행을 예방하려면 사용자가 입력한 값의 패턴을 검사해 허가되지 않은 패턴이 포함될 경우 기능을 실행하지 않거나, 사용자 입력값 전체를 실행하고자 하는 명령어(예를 들어, 위 코드에서는 /bin/ls)의 인자로 간주해 사용하는 방법이 있다.

안전한 코드 예시

```
1:  const express = require('express');
2:  const child_process = require("child_process");
3:
4:  router.get("/patched", (req, res) => {
5:    const inputPath = req.query.path;
6:    const regPath = /^(\/[\w^]+)+\/?$/;
7:
8:    // 첫 번째 방법, 사용자 입력값 필터링 - 리눅스 경로 지정에 필요한 문자만 허용
9:    if (!inputPath.match(regPath)) {
10:     return res.send('not valid path');
11:   }
12:
13:   // 두 번째 방법, 사용자 입력값이 명령어의 인자로만 사용되도록 하는 함수 사용
14:   child_process.execFile(
15:     "/bin/ls", ["-l", inputPath],
16:     function (err, data) {
17:       if (err) {
18:         return res.send('not valid path');
19:       } else {
20:         return res.send(data);
21:       }
22:     }
23:   );
24: });
```

라. 참고자료

① CWE-78: Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection'), MITRE
https://cwe.mitre.org/data/definitions/78.html

② Command Injection, OWASP
https://owasp.org/www-community/attacks/Command_Injection

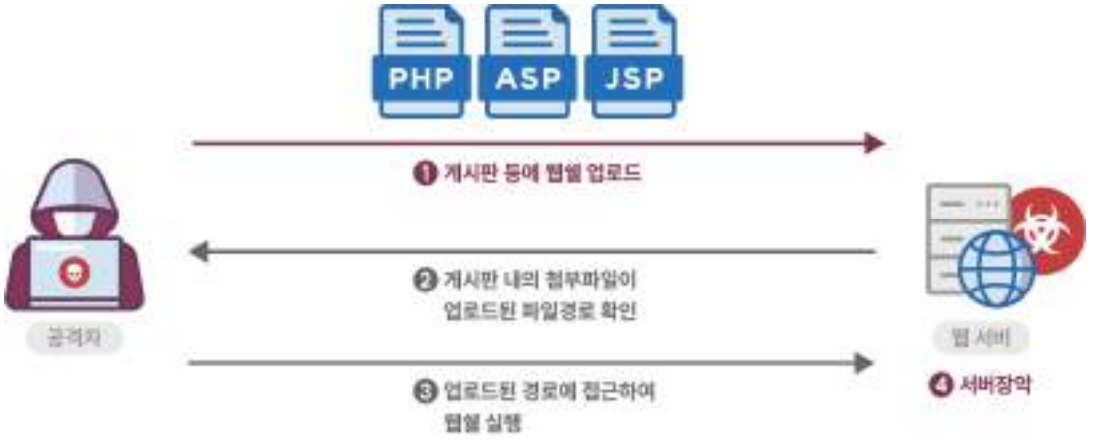
③ OS Command Injection Defense Cheat Sheet, OWASP
https://cheatsheetseries.owasp.org/cheatsheets/OS_Command_Injection_Defense_Cheat_Sheet.html

④ NodeJs-Secure Code wiki
https://securecode.wiki/docs/lang/nodejs/

⑤ Child process, NodeJS
https://nodejs.org/api/child_process.html

6. 위험한 형식 파일 업로드 VanillaJS - ReactJS - ExpressJS ✓

가. 개요



서버 측에서 실행 가능한 스크립트 파일(asp, jsp, php, sh 파일 등)이 업로드 가능하고 이 파일을 공격자가 웹을 통해 직접 실행시킬 수 있는 경우, 시스템 내부 명령어를 실행하거나 외부와 연결해 시스템을 제어할 수 있는 보안약점이다. 서버 사이드 언어(ASP, JSP, PHP)와 달리 NodeJS에서는 파일 업로드가 성공했다고 하더라도 설정파일 수정 빛 반영과 같이 파일 실행을 위해 거쳐야 할 추가 작업들로 인해 웹쉘 공격이 그리 간단하지 않다. 가능성이 낮을 뿐 NodeJS를 기반으로 하는 서버 또한 업로드 공격에 취약하므로, 이에 대비가 필요하다.

나. 안전한 코딩기법

파일 업로드 공격을 방지하기 위해서 특정 파일 유형만 허용하도록 화이트리스트 방식으로 파일 유형을 제한해야 한다. 이때 파일의 확장자 및 업로드 된 파일의 Content-Type도 함께 확인해야 한다. 또한 파일 크기 및 파일 개수를 제한하여 시스템 자원 고갈 등으로 서비스 거부 공격이 발생하지 않도록 제한해야 한다. 업로드 된 파일을 웹 루트 폴더 외부에 저장해 공격자가 URL을 통해 파일을 실행할 수 없도록 해야 하며, 가능하면 업로드 된 파일의 이름은 공격자가 추측할 수 없는 무작위한 이름으로 변경 후 저장하는 것이 안전하다. 또한 업로드 된 파일을 저장할 경우에는 최소 권한만 부여하는 것이 안전하고 실행 여부를 확인하여 실행 권한을 삭제해야 한다.

다. 코드예제

업로드 대상 파일 개수, 크기, 확장자 등의 유효성 검사를 하지 않고 파일 시스템에 그대로 저장할 경우 공격자에 의해 악성코드, 쉘코드 등 위험한 형식의 파일이 시스템에 업로드 될 수 있다.

안전하지 않은 코드 예시

```
1:  const express = require('express');
2:
3:  router.post("/vuln", (req, res) => {
4:    const file = req.files.products;
5:    const fileName = file.name;
6:
7:    // 업로드 한 파일 타입 검증 부재로 악성 스크립트 파일 업로드 가능
8:    file.mv("/usr/app/temp/" + fileName, (err) => {
9:      if (err) return res.send(err);
10:     res.send("upload success");
11:   });
12: });
```

아래 코드는 업로드 하는 파일의 개수, 크기, 파일 확장자 등을 검사해 업로드를 제한하고 있다. 파일 타입 확인은 MIME 타입을 확인하는 과정으로 파일 이름에서 확장자만 검사할 경우 변조된 확장자를 통해 업로드 제한을 회피할 수 있어 파일 자체의 시그니처를 확인하는 과정을 보여 준다.

안전한 코드 예시

```
1:  const express = require('express');
2:
3:  router.post("/patched", (req, res) => {
4:    const allowedMimeTypes = ["image/png", "image/jpeg"];
5:    const allowedSize = 5242880;
6:
7:    const file = req.files.products;
8:    const fileName = file.name;
9:
10:   // 업로드 한 파일 타입 검증을 통해 악성 스크립트 파일 업로드 방지
11:   if (allowedMimeTypes.indexOf(file.mimetype) < 0) {
12:     res.send("file type not allowed");
13:   } else {
14:     file.mv("/usr/app/temp/" + fileName, (err) => {
15:       if (err) return res.send(err);
16:       res.send("upload success");
17:     });
18:   }
19: });
```

라. 참고자료

① CWE-434: Unrestricted Upload of File with Dangerous Type, MITRE,
https://cwe.mitre.org/data/definitions/434.html

② Unrestricted File Upload, OWASP,
https://owasp.org/www-community/vulnerabilities/Unrestricted_File_Upload

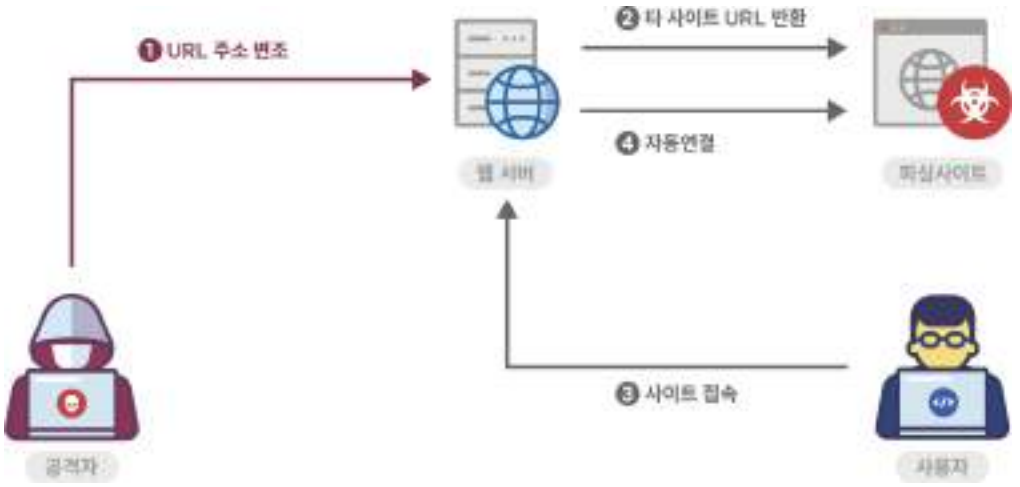
③ express-fileupload, npm
https://www.npmjs.com/package/express-fileupload

④ NodeJs-Secure Code wiki
https://securecode.wiki/docs/lang/nodejs/

⑤ The Javascript Guide: Web Application Secure Coding Practices
https://github.com/Checkmarx/JS-SCP/blob/master/dist/js-webapp-scp.pdf

7. 신뢰되지 않은 URL주소로 자동접속 연결 VanillaJS - ReactJS - ExpressJS ✓

가. 개요



사용자가 입력하는 값을 외부 사이트 주소로 사용해 해당 사이트로 자동 접속하는 서버 프로그램은 피싱(Phishing) 공격에 노출되는 취약점을 가진다. 클라이언트에서 전송된 URL 주소로 연결하기 때문에 안전하다고 생각할 수 있으나, 공격자는 정상적인 폼 요청을 변조해 사용자가 위험한 URL로 접속할 수 있도록 공격할 수 있다.

나. 안전한 코딩기법

리다이렉션을 허용하는 모든 URL을 서버 측 화이트리스트로 관리하고 사용자 입력값을 리다이렉트 할 URL이 존재하는지 검증해야 한다.

만약 사용자 입력값이 화이트리스트로 관리가 불가능하고 리다이렉션 URL의 인자 값으로 사용되어야만 하는 경우는 모든 리다이렉션에서 프로토콜과 host 정보가 들어가지 않는 상대 URL(relative URL)을 사용 및 검증해야 한다. 또는 절대 URL(absoute URL)을 사용할 경우 리다이렉션을 실행하기 전에 사용자 입력 URL이 https://myhompage.com/ 처럼 정상 서비스 중인 URL로 시작하는지 확인해야 한다.

다. 코드예제

ExpressJS에서는 redirect() 함수를 시용해 사용자 요청을 다른 페이지로 리다이렉트할 수 있다. 다음은 사용자로부터 입력받은 url 주소를 검증 없이 redirect 함수의 인자로 사용해 의도하지 않은 사이트로 접근하도록 하거나 피싱(Phishing)공격에 노출되는 예시를 보여 준다.

안전하지 않은 코드 예시

```
1:  const express = require('express');
2:
3:  router.get("/vuln", (req, res) => {
4:    const url = req.query.url;
5:    // 사용자가 전달한 URL 주소를 검증 없이 그대로 리다이렉트 처리
6:    res.redirect(url);
7:  });
```

다음은 안전한 코드 예제로 사용자로부터 주소를 입력받아 리다이렉트하고 있는 코드로 위험한 도메인이 포함될 수 있기 때문에 화이트리스트로 사전에 정의된 안전한 웹사이트에 한하여 리다이렉트 할 수 있도록 한다.

안전한 코드 예시

```
1:  const express = require('express');
2:
3:  router.get("/patched", (req, res) => {
4:    const whitelist = ["http://safe-site.com", "https://www.example.com"];
5:    const url = req.query.url;
6:
7:    // 화이트리스트에 포함된 주소가 아니라면 리다이렉트 없이 에러 반환
8:    if (whitelist.indexOf(url) < 0) {
9:      res.send("wrong url");
10:   } else {
11:     res.redirect(url);
12:   }
13: });
```

라. 참고자료

① CWE-601: URL Redirection to Untrusted Site ('Open Redirect'), MITRE, https://cwe.mitre.org/data/definitions/601.html

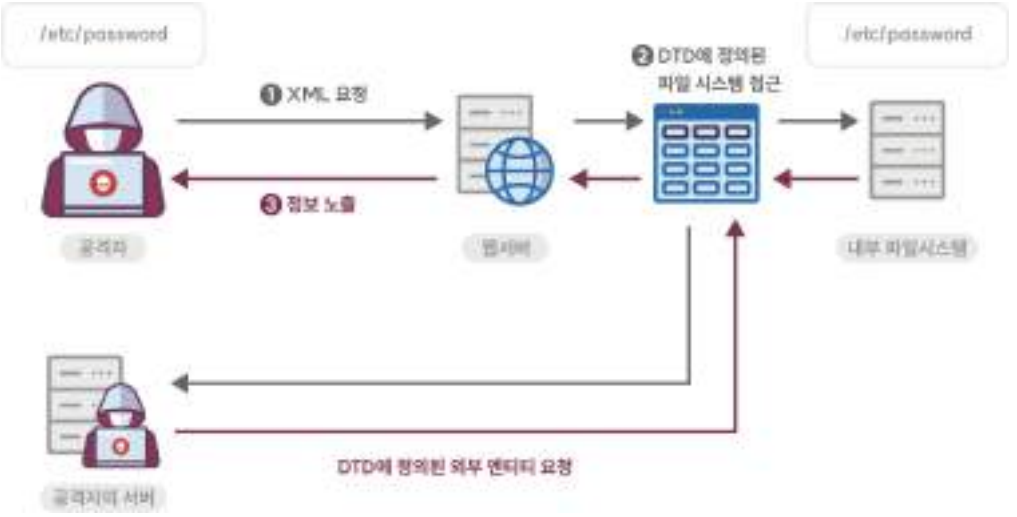
② Unvalidated Redirects and Forwards Cheat Sheet, OWASP https://cheatsheetseries.owasp.org/cheatsheets/Unvalidated_Redirects_and_Forwards_Cheat_Sheet.html

③ JavaScript Secure Coding Standard - Ministry of Transport and Communications https://compliance.qcert.org/sites/default/files/library/2018-10/MOTC-CIPD_JavaScript_Coding_Standard(US).pdf

④ express Documentation - StrongLoop, IBM https://devdocs.io/express/index#res.redirect https://devdocs.io/express/index#req.query

8. 부적절한 XML 외부 개체 참조 VanillaJS - ReactJS - ExpressJS ✓

가. 개요



XML 문서에는 DTD(Document Type Definition)를 포함할 수 있으며, DTD는 XML 엔티티(entity)를 정의한다. 부적절한 XML 외부개체 참조 보안약점은 서버에서 XML 외부 엔티티를 처리할 수 있도록 설정된 경우에 발생할 수 있다. 취약한 XML parser가 외부값을 참조하는 XML을 처리할 때, 공격자가 삽입한 공격 구문이 동작되어 서버 파일 접근, 불필요한 자원 사용, 인증 우회, 정보 노출 등이 발생할 수 있다.

NodeJS에서는 내장 XML 파싱 엔진을 지원하지 않으며, 별도의 라이브러리를 사용해야 한다. 어떠한 라이브러리를 사용해도 무방하지만 반드시 외부 엔티티 파싱 기능을 비활성화 하는 옵션을 설정해 주어야 한다.

나. 안전한 코딩기법

로컬 정적 DTD를 사용하도록 설정하고, 외부에서 전송된 XML 문서에 포함된 DTD를 완전하게 비활성화해야 한다. 비활성화를 할 수 없는 경우에는 외부 엔티티 및 외부 문서 유형 선언을 각 파서에 맞는 고유한 방식으로 비활성화 한다. 외부 라이브러리를 사용할 경우 기본적으로 외부 엔티티에 대한 구문 분석 기능을 제공하는지 확인하고, 제공이 되는 경우 해당 기능을 비활성화 할 수 있는 방법을 확인해 외부 엔티티 구문 분석 기능을 비활성화 한다.

많이 사용하는 XML 파서의 한 종류인 lxml의 경우 외부 엔티티 구문 분석 옵션인 resolve_entities 옵션을 비활성화 해야 한다. 또한 외부 문서 조회 시 네트워크 액세스를 방지하는 no_network 옵션이 활성화(True) 되어 있는지도 확인해야 한다.

다. 코드예제

다음 예제는 XML 소스를 읽어와 분석하는 코드를 보여 준다. 공격자는 아래와 같이 XML 외부 엔티티를 참조하는 xxe.xml 데이터를 전송하고, 서버에서 해당 데이터 파싱 시 /etc/passwd 파일 내용이 사용자에게 전달될 수 있다.

```
// xxe.xml
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE foo [
  <!ELEMENT foo ANY >
  <!ENTITY xxe1 SYSTEM "file:///etc/passwd" >
]>
<foo>&xxe1;</foo>
```

안전하지 않은 코드 예시

```
1:  const express = require('express');
2:  const libxmljs = require("libxmljs");
3:
4:  router.post("/vuln", (req, res) => {
5:    if (req.files.products && req.files.products.mimetype == "application/xml") {
6:      const products = libxmljs.parseXmlString(
7:        req.files.products.data.toString("utf8"),
8:        // 외부 엔티티 파싱 허용 설정(미설정 시 기본값은 false)
9:        { noent: true }
10:     );
11:     return res.send(products.get("//foo").text());
12:   }
13:   return res.send("fail");
14: });
```

libxmljs 라이브러리에서는 기본적으로 외부 엔티티 파싱 기능이 비활성화 되어 있지만, 명시적으로 비활성 선언을 해주는 것이 좋다.

안전한 코드 예시

```
1:  const express = require('express');
2:  const libxmljs = require("libxmljs");
3:
4:  router.post("/patched", (req, res) => {
5:    if (req.files.products && req.files.products.mimetype == "application/xml") {
6:      const products = libxmljs.parseXmlString(
7:        req.files.products.data.toString("utf8"),
8:        // 외부 엔티티 파싱을 허용하지 않도록 설정
9:        // 미설정 시 기본값은 false이지만, 명시적으로 선언을 해 주는 것이 좋음
10:       { noent: false }
11:     );
12:     return res.send(products.get("//foo").text());
13:   }
14:   return res.send("fail");
15: });
```

라. 참고자료

① CWE-611: Improper Restriction of XML External Entity Reference, MITRE, https://cwe.mitre.org/data/definitions/611.html

② XML External Entity (XXE) Processing, OWASP, https://owasp.org/www-community/vulnerabilities/XML_External_Entity_(XXE)_ Processing

③ XML External Entity Prevention Cheat Sheet, OWASP, https://cheatsheetseries.owasp.org/cheatsheets/XML_External_Entity_Prevention_Cheat_Sheet.html

④ JavaScript Secure Coding Standard - Ministry of Transport and Communications https://compliance.qcert.org/sites/default/files/library/2018-10/MOTC-CIPD_JavaScript_Coding_Standard(US).pdf

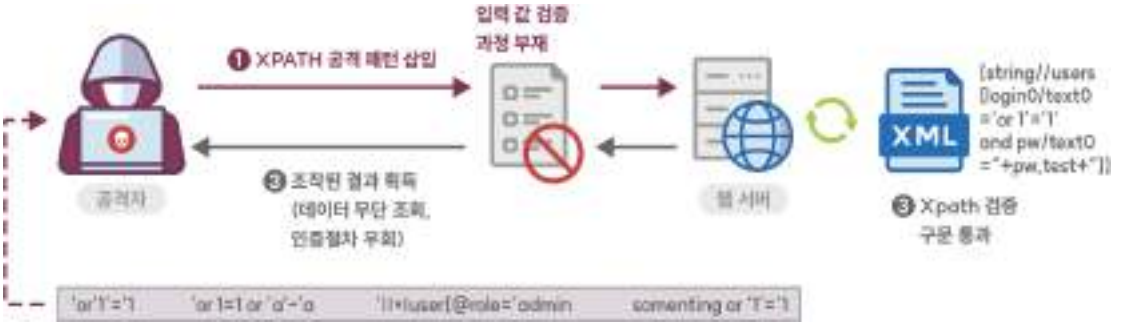
⑤ NodeJs-Secure Code wiki https://securecode.wiki/docs/lang/nodejs/

⑥ Sonar Rules[3] https://rules.sonarsource.com/javascript/RSPEC-2755

3) 해당 자료는 Sonar 사의 소유로 LPGL 3.0 라이선스로 보호되며, 본문에서 제시하는 코드는 sonarsource 사이트에서 제공하는 룰셋의 일부를 인용했다. 상세 문서는 다음 주소에서 확인 가능: https://www.sonarsource.com/license/

9. XML 삽입 VanillaJS - ReactJS - ExpressJS ✓

가. 개요



검증되지 않은 외부 입력값이 XQuery 또는 XPath 쿼리문을 생성하는 문자열로 사용되어 공격자가 쿼리문의 구조를 임의로 변경하고 임의의 쿼리를 실행해 허가되지 않은 데이터를 열람하거나 인증절차를 우회할 수 있는 보안약점이다.

나. 안전한 코딩기법

XQuery 또는 XPath 쿼리에 사용되는 외부 입력 데이터에 대하여 특수문자 및 쿼리 예약어를 필터링 하고 인자화된 쿼리문을 지원하는 XQuery를 사용해야 한다.

다. 코드예제

다음은 자바스크립트 xpath 패키지를 사용해 XML 데이터를 처리하는 예시를 보여 준다. 사용자 이름과 패스워드가 XML 내의 값과 일치할 경우 사용자 홈 디렉터리를 반환하는 예시로, 인자화된 쿼리문을 사용하지 않고 XPath 쿼리문을 생성하고 있어 공격자가 요청문을 조작해 패스워드 검증 로직을 우회할 수 있다.

안전하지 않은 코드 예시

```
1:  const express = require('express');
2:  const xpath = require('xpath');
3:  const dom = require('xmldom').DOMParser;
4:
5:  const xml = `<users>
6:    <user>
7:      <login>john</login>
8:      <password>abracadabra</password>
9:      <home_dir>/home/john</home_dir>
10:   </user>
```

```
    <user>
      <login>cbc</login>
      <password>1mgr8</password>
      <home_dir>/home/cbc</home_dir>
    </user>
    </users>`;

router.get("/vuln", (req, res) => {
    const userName = req.query.userName;
    const userPass = req.query.userPass;

    const doc = new dom().parseFromString(xml);

    // 조작된 입력값(/vuln?userName=john' or ''='&userPass=' or ''=')을 통해
    // 패스워드 검사 로직을 우회할 수 있음
    const badXPathExpr = "//users/user[login/text()='" + userName
                         + "' and password/text() = '" + userPass + "']/home_dir/text()";
    const selected = xpath.select(badXPathExpr, doc);

    try {
      const userPath = selected[0].toString();
      return res.send(`userPath = ${userPath}`);
    } catch {
      return res.send('not found');
    }
});
```

xpath에서는 parse 및 select 함수를 사용해 인자화된 쿼리를 생성하고, 쿼리문에 필요한 변수값을 전달할 수 있다. 만약 xpath가 아닌 다른 패키지를 사용할 경우 반드시 인자화된 쿼리를 지원하는지 확인해야 한다. 다음은 인자화된 쿼리를 통해 사용자가 조작한 입력값이 실행되지 않도록 하는 안전한 예시를 보여 준다.

안전한 코드 예시

```
1:  const express = require('express');
2:  const xpath = require('xpath');
3:  const dom = require('xmldom').DOMParser;
4:
5:  const xml = `<users>
6:    ...
7:    </users>`;
8:
9:  router.get("/patched", (req, res) => {
10:   const userName = req.query.userName;
11:   const userPass = req.query.userPass;
12:   const doc = new dom().parseFromString(xml);
13:
14:   // 인자화된 쿼리 생성
15:   const goodXPathExpr = xpath.parse("//users/user[login/text()=$userName and password/text()
16:   =$userPass]/home_dir/text()");
17:   // 쿼리문에 변수값 전달 및 XML 조회
18:   const selected = goodXPathExpr.select({
19:     node: doc,
20:     variables: { userName: userName, userPass: userPass }
21:   });
22:   try {
23:     const userPath = selected[0].toString();
24:     return res.send(`userPath = ${userPath}`);
25:   } catch {
26:     return res.send('not found');
27:   }
28: });
```

라. 참고자료

① CWE-643: Improper Neutralization of Data within XPath Expressions ('XPath Injection'), MITRE, https://cwe.mitre.org/data/definitions/643.html

② XPATH Injection, OWASP, https://owasp.org/www-community/attacks/XPATH_Injection

10. LDAP 삽입 VanillaJS - ReactJS - ExpressJS ✓

가. 개요

외부 입력값을 적절한 처리 없이 LDAP 쿼리문이나 결과의 일부로 사용하는 경우, LDAP 쿼리문이 실행될 때 공격자는 LDAP 쿼리문의 내용을 마음대로 변경할 수 있다. 이로 인해 프로세스가 명령을 실행한 컴포넌트와 동일한 권한(Permission)을 가지고 동작하게 된다. 자바스크립트에서는 LDAP 데이터 처리를 위한 다양한 패키지가 존재하며, 그 중 ldapjs가 가장 사용자들이 선호하는 패키지로 알려져 있다.

나. 안전한 코딩기법

다른 삽입 공격들과 마찬가지로 LDAP 삽입에 대한 기본적인 방어 방법은 적절한 유효성 검사를 적용하는 것이다.

- 올바른 인코딩(Encoding) 함수를 사용해 모든 변수 이스케이프 처리
- 화이트리스트 방식의 입력값 유효성 검사
- 사용자 패스워드와 같은 민감한 정보가 포함된 필드 인덱싱
- LDAP 바인딩 계정에 할당된 권한 최소화

다. 코드예제

사용자의 입력을 그대로 LDAP 질의문에 사용하고 있으며 이 경우 권한 상승 등의 공격에 노출될 수 있다.

안전하지 않은 코드 예시

```
1:  const express = require('express');
2:  const ldap = require('ldapjs');
3:
4:  const config = {
5:    url: 'ldap://ldap.forumsys.com',
6:    base: 'dc=example,dc=com',
7:    dn: 'cn=read-only-admin,dc=example,dc=com',
8:    secret: 'd0accf0ac0dfb0d0fd...',
9:  };
10:
11: async function searchLDAP (search) {
12: ...
13: }
14:
15: router.get("/vuln", async (req, res) => {
16:   // 사용자의 입력을 그대로 LDAP 질의문으로 사용해 권한 상승 등의 공격에 노출
17:   const search = req.query.search;
18:   const result = await searchLDAP(search);
19:
20:   return res.send(result);
21: });
```

사용자의 입력 중 LDAP 질의문에 사용될 변수를 이스케이프 하여 질의문 실행 시 공격에 노출되는 것을 예방할 수 있다. ldapjs 패키지에서는 사용자 입력값을 이스케이핑 할 수 있는 parseFilter 함수를 제공한다.

안전한 코드 예시

```
1:  const express = require('express');
2:  const ldap = require('ldapjs');
3:  const parseFilter = require('ldapjs').parseFilter;
4:
5:  const config = {
6:    url: 'ldap://ldap.forumsys.com',
7:    base: 'dc=example,dc=com',
8:    dn: 'cn=read-only-admin,dc=example,dc=com',
9:    secret: 'd0accf0ac0dfb0d0fd...',
10: };
```

```
11:
12:  async function searchLDAP (search) {
13:  ...
14:  }
15:
16:  router.get("/patched", async (req, res) => {
17:
19:    let search;
20:
21:    // 사용자의 입력에 필터링을 적용해 공격에 사용될 수 있는 문자 발견 시
22:    // 사용자에게 잘못된 요청값임을 알리고, 질의문을 요청하지 않음
23:    try {
24:      search = parseFilter(req.query.search);
25:    } catch {
26:      return res.send('잘못된 요청값입니다.');
27:    }
28:
29:    const result = await searchLDAP(search);
30:    return res.send(result);
31:  });
```

라. 참고자료

① CWE-90: Improper Neutralization of Special Elements used in an LDAP Query ('LDAP Injection'), MITRE, https://cwe.mitre.org/data/definitions/90.html

② LDAP Injection Prevention Cheat Sheet, OWASP, https://cheatsheetseries.owasp.org/cheatsheets/LDAP_Injection_Prevention_Cheat_Sheet.html

③ ldapjs Filters API
http://ldapjs.org/filters.html

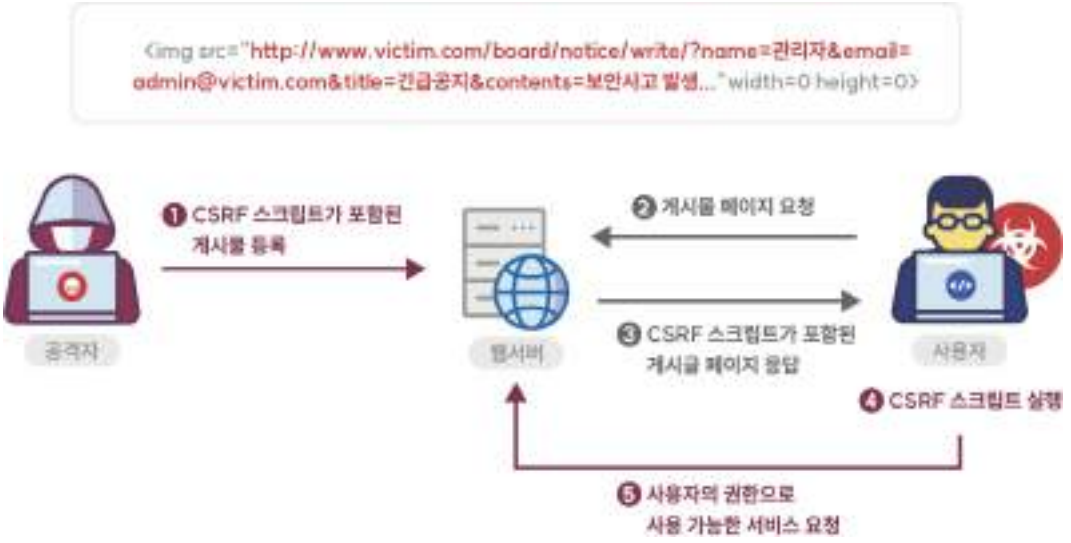
참고: 자바스크립트 기반 LDAP 데이터 처리 함수

```
// (참고) ldapjs 패키지를 사용해 ldap 데이터를 처리하는 코드 예시로, 앞서 제시한 두 코드에서
// 생략된 searchLDAP 함수의 전체 코드
const ldap = require('ldapjs');

const config = {
  url: 'ldap://ldap.forumsys.com',
  base: 'dc=example,dc=com',
  dn: 'cn=read-only-admin,dc=example,dc=com',
  password: 'password',
};
async function searchLDAP (search) {
  const opts = {
    filter: `(&(objectClass=${search}))`,
    attributes: ['sn', 'cn', 'mail', 'telephonenumber', 'uid'],
    scope: 'sub',
  };
  const users = [];
  const client = ldap.createClient({ url: config.url });
  return new Promise((resolve, reject) => {
    client.bind(config.dn, config.password, (err) => {
      if (err) {
        console.log('LDAP bind error - ', err);
      } else {
        client.search(config.base, opts, (err, res) => {
          res.on('searchEntry', (entry) => {
            users.push(entry.object);
          });
          ...
          res.on('end', (result) => {
            console.log('status: ' + result.status);
            resolve(users);
          });
        });
      }
    });
  });
}
```

11. 크로스사이트 요청 위조(CSRF) VanillaJS ✓ ReactJS ✓ ExpressJS ✓

가. 개요



특정 웹사이트에 대해 사용자가 인지하지 못한 상황에서 사용자의 의도와는 무관하게 공격자가 의도한 행위(수정, 삭제, 등록 등)를 요청하게 하는 공격을 말한다. 웹 응용프로그램이 사용자로부터 받은 요청이 해당 사용자가 의도한 대로 작성되고 전송된 것인지 확인하지 않는 경우 발생 가능하다. 특히 사용자가 관리자 권한을 가지는 경우 사용자 권한관리, 게시물 삭제, 사용자 등록 등 관리자 권한으로만 수행 가능한 기능을 공격자의 의도대로 실행시킬 수 있게 된다. 공격자는 사용자가 인증한 세션이 특정 동작을 수행해도 계속 유지되어 정상적인 요청과 비정상적인 요청을 구분하지 못하는 점을 악용한다.

자바스크립트의 대표적인 웹 애플리케이션 프레임워크인 ExpressJS에서는 기본적으로 CSRF 토큰 기능을 지원하지 않으며, csurf와 같은 별도의 패키지를 사용해 CSRF 토큰을 적용할 수 있다. csurf는 많은 시큐어 코딩 문서 및 가이드에서 CSRF 공격 대응에 사용 가능한 것으로 알려져 있다. 초기에는 안전한 것으로 알려졌으나, 해당 패키지 적용만으로 대응이 어려운 다양한 유형의 보안 취약점 발견 및 구현 자체의 제약(취약한 해쉬함수 사용)으로 인해 공식 리파지토리에서는 더 이상 유지되지 않은 deprecated 패키지로 공지되어 있다.

비록 패키지 자체만으로 완전한 CSRF 공격 대응이 불가능하다는 단점이 있지만, 이를 대체할 수 있는 것으로 검증된 패키지가 아직까지 등장하지 않았고, 추가 옵션 적용을 통해 보안성을 강화할 수 있다. 토큰 값 세션 저장, 토큰 헤더 키 추가와 같은 방법을 사용 가능하다.

나. 안전한 코딩기법

해당 요청이 정상적인 사용자가 절차에 따라 요청한 것인지 구분하기 위해 세션별로 CSRF 토큰을 생성하여 세션에 저장하고, 사용자가 작업 페이지를 요청할 때마다 hidden 값으로 클라이언트에게 토큰을 전달한 뒤, 해당 클라이언트의 데이터 처리 요청 시 전달되는 CSRF 토큰값을 체크하여 요청의 유효성을 검사하도록 한다.

다. 코드예제

템플릿 엔진을 사용해 클라이언트측 코드를 렌더링 할 경우, 서버로 데이터를 전송하거나 기능을 호출하는 부분에 csrf 토큰을 포함하지 않을 경우 CSRF 공격에 취약하다.

안전하지 않은 코드 예시

```
1:  <html>
2:  ...
3:  <body>
4:    <form method='post' action='/api/vuln'>
5:      <!-- CSRF 토큰 없이 폼 데이터를 서버에 전달 -->
6:      <fieldset>
7:        <legend>Your Profile:</legend>
8:        <label for='username'> Name:</label>
9:        <input name='username' class='username' type='text'> <br><br>
10:       <label for='email' > Email:</label>
11:       <input name='email' class='useremail' type='email'> <br><br>
12:       <button type='submit'>UPDATE</button>
13:     </fieldset>
14:   </form>
15: </body>
16: </html>
```

다음은 CSRF 토큰 처리가 없는 서버측 라우터 예시를 보여 준다. 만약 공격자가 사용자 업데이트 페이지를 모방해 사용자가 해당 페이지를 통해 서버로 업데이트 요청을 보내도록 할 경우, 서버는 정상적인 세션을 가지는 클라이언트의 요청을 아무런 검증 없이 실행하게 된다.

안전하지 않은 코드 예시

```
1:  const express = require('express');
2:
3:  router.post("/api/vuln", (req, res) => {
4:    const userName = req.body.username;
5:    const userEmail = req.body.useremail;
6:
7:    // 사용자 업데이트 요청이 정상 사용자로부터 온 것이라고 간주하고,
8:    // 사용자로부터 받은 값을 그대로 내부 함수에 전달
9:    if (update_user(userName, userEmail)) {
10:     return res.send('update completed');
11:   } else {
12:     return res.send('update error');
13:   }
14: });
```

다음은 csurf 패키지를 사용해 CSRF 공격에 대응하는 예시를 보여 준다. 앞서 설명한 것처럼, csurf를 포함해 어떠한 패키지를 사용하더라도 다양한 변종 CSRF 패턴을 모두 방어하는 것은 불가능하다. 다만, 다음 예시에서 보는 것처럼 csurf 기능의 한계를 인지한 상태에서 세션에 정보 저장, 토큰 접두어 사용 등의 부가적인 보안 대책을 적용해 방어 수준을 높일 수 있다.

안전한 코드 예시

```
1:  const csrf = require('csurf');
2:  const express = require('express');
3:  const session = require("express-session");
4:
5:  const app = express();
6:  // 토큰을 쿠키가 아닌 세션에 저장
7:  app.use(session({
8:    secret: process.env.COOKIE_SECRET,
9:    cookie: { path: '/', secure: true, httpOnly: true, sameSite: 'strict' },
10:   saveUninitialized: false,
11:   resave: false
12: }));
```

```
13:
14:  // CSRF 토큰 이름 앞에 __Host- 접두어 추가
15:  const csrfProtection = csrf({
16:    key: '__Host-token',
17:  });
18:
19:  // 클라이언트측 프레임워크(ReactJS)를 위한 토큰 값 제공
20:  router.get("/getCSRFToken", csrfProtection, (req, res) => {
21:    res.json({ csrfToken: req.csrfToken() });
22:  });
23:
24:  // 템플릿 페이지를위한 토큰 값 제공
25:  router.get("/page", csrfProtection, (req, res) => {
26:    res.render('csrf', { csrfToken: req.csrfToken() });
27:  });
28:
29:  // 라우터 데이터 처리 전 클라이언트의 csrf 토큰을 검사
30:  router.post("/api/patched", csrfProtection, (req, res) => {
31:    const userName = req.body.username;
32:    const userEmail = req.body.useremail;
33:
34:    if (update_user(userName, userEmail)) {
35:      return res.send('update completed');
36:    } else {
37:      return res.send('update error');
38:    }
39:  });
```

클라이언트측 코드 내의 form action 또는 axios와 같은 서버 기능 호출 부분에도 csrf 토큰을 추가해야 한다. 템플릿 엔진에서 폼 형태로 데이터를 전달하는 경우 hidden 필드에 토큰 값을 할당하면 된다.

안전한 코드 예시

```
...
  <form method='post' action='/api/vuln'>
    <!-- 서버로부터 제공 받은 CSRF 토큰을 폼 데이터에 hidden 값으로 포함해 전달 -->
    <input type='hidden' name='_csrf' value='<%= csrfToken  %>'>
    <fieldset>
      <legend>Your Profile:</legend>
      <label for='username'> Name:</label>
      <input name='username' class='username' type='text'> <br><br>
      <label for='email' > Email:</label>
      <input name='email' class='useremail' type='email'> <br><br>
      <button type='submit'>UPDATE</button>
    </fieldset>
  </form>
...
```

클라이언트측에서 렌더링을 처리하는 ReactJS의 경우 서버 기능 호출 전 서버로부터 토큰을 받아와 요청 헤더 내에 삽입해야 한다.

안전한 코드 예시

```
const App = () => {
  const getData = async () => {
    // 서버에 기능 호출 전 먼저 CSRF 토큰을 받아와 헤더에 저장
    const response = await axios.get('getCSRFToken');
    axios.defaults.headers.post['X-CSRF-Token'] = response.data.csrfToken;

    // CSRF 토큰이 설정된 상태에서 서버 기능 호출
    const res = await axios.post('api/patched', {
      username: 'hello_user',
      useremail: 'test@email.com',
    });
    document.write(res.data);
  };
  React.useEffect(() => { getData(); }, []);
  return <div>react-test</div>;
};
ReactDOM.render(<App />, document.getElementById("root"));
```

라. 참고자료

① CWE-352: Cross-Site Request Forgery (CSRF), MITRE,
https://cwe.mitre.org/data/definitions/352.html

② Cross Site Request Forgery (CSRF), OWASP,
https://owasp.org/www-community/attacks/csrf

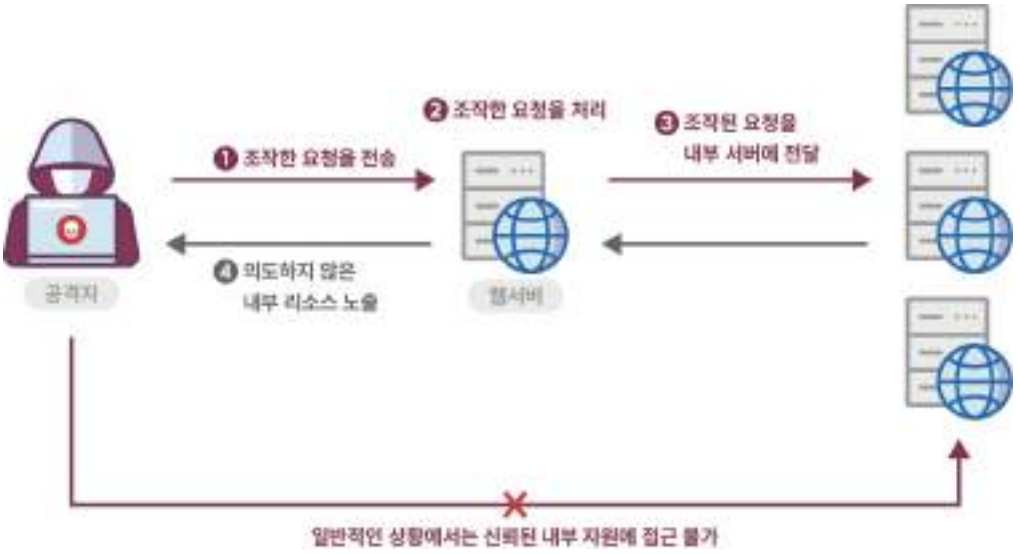
③ Cross-Site Request Forgery Prevention Cheat Sheet, OWASP
https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html

④ NodeJS Secure Code Wiki - Payatu
https://securecode.wiki/docs/lang/nodejs

⑤ Disabling CSR protections is security-sensitive
https://rules.sonarsource.com/javascript/RSPEC-4502

12. 서버사이드 요청 위조 VanillaJS - ReactJS - ExpressJS ✓

가. 개요



적절한 검증 절차를 거치지 않은 사용자 입력값을 내부 서버간의 요청에 사용해 악의적인 행위가 발생할 수 있는 보안약점이다. 외부에 노출된 웹 서버가 취약한 애플리케이션을 포함하는 경우 공격자는 URL 또는 요청문을 위조해 접근통제를 우회하는 방식으로 비정상적인 동작을 유도하거나 신뢰된 네트워크에 있는 데이터를 획득할 수 있다.

나. 안전한 코딩기법

식별 가능한 범위 내에서 사용자의 입력값을 다른 시스템의 서비스 호출에 사용하는 경우, 사용자의 입력값을 화이트리스트 방식으로 필터링한다.

부득이하게 사용자가 지정하는 무작위의 URL을 받아들여야 하는 경우라면 내부 URL을 블랙리스트로 지정하여 필터링 한다. 또한 동일한 내부 네트워크에 있더라도 기기 인증, 접근권한을 확인하여 요청이 이루어질 수 있도록 한다.

다. 코드예제

〈참고 : 삽입 코드의 예〉

설명	삽입 코드의 예
내부망 중요 정보 획득	http://sample_site.com/connect?url=http://192.168.0.45/member/list.json
외부 접근 차단된 admin 페이지 접근	http://sample_site.com/connect?url=http://192.168.0.45/admin
도메인 체크를 우회하여 중요 정보 획득	http://sample_site.com/connect?url=http://sample_site.com:x@192.168.0.45/member/list.json
단축 URL을 이용한 Filter 우회	http://sample_site.com/connect?url=http://bit.ly/sdjk3kjhkl3
도메인을 사설IP로 설정해 중요정보 획득	http://sample_site.com/connect?url=http://192.168.0.45/member/list.json
서버 내 파일 열람	http://sample_site.com/connect?url=file:///etc/passwd

다음 예제는 안전하지 않은 코드를 보여 준다. 사용자로부터 입력된 URL 주소를 검증 없이 사용하면 의도하지 않은 다른 서버의 자원에 접근할 수 있게 된다.

안전하지 않은 코드 예시

```
1:  const request = require('request');
2:  const express = require('express');
3:
4:  router.get("/vuln", async (req, res) => {
5:    const url = req.query.url;
6:
7:    // 사용자가 입력한 주소를 검증하지 않고 HTTP 요청을 보낸 후
8:    // 그 응답을 그대로 사용자에게 전달
9:    await request(url, (err, response) => {
10:     const resData = response.body;
11:     return res.send(resData);
12:   });
13: });
```

다음과 같이 안전한 코드를 작성하면 사전에 정의된 서버 목록에서 정의하고 매칭되는 URL만 사용할 수 있으므로 URL 값을 임의로 조작할 수 없다.

안전한 코드 예시

```
1:  const request = require('request');
2:  const express = require('express');
3:
4:  router.get("/patched", async (req, res) => {
5:    const url = req.query.url;
6:    const whiteList = [ 'www.example.com', 'www.safe.com'];
7:
8:    // 사용자가 입력한 URL을 화이트리스트로 검증한 후 그 결과를 반환하여
9:    // 검증되지 않은 주소로 요청을 보내지 않도록 제한
10:   if (whiteList.includes(url)) {
11:     await request(url, (err, response) => {
12:       const resData = response.body;
13:       return res.send(resData);
14:     });
15:   } else {
16:     return res.send('잘못된 요청입니다');
17:   }
18: });
```

라. 참고자료

① CWE-918: Server-Side Request Forgery (SSRF), MITRE
https://cwe.mitre.org/data/definitions/918.html

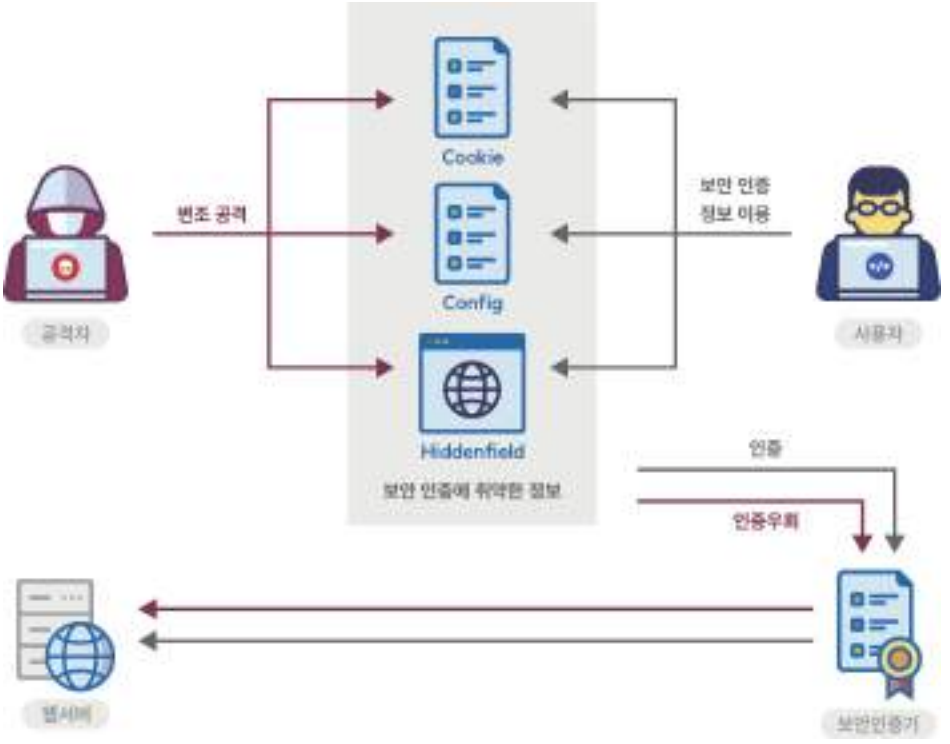
② Server Side Request Forgery, OWASP
https://owasp.org/www-community/attacks/Server_Side_Request_Forgery

③ Server-Side Request Forgery Prevention Cheat Sheet, OWASP
https://cheatsheetseries.owasp.org/cheatsheets/Server_Side_Request_Forgery_Prevention_Cheat_Sheet.html

④ Server-side requests should not be vulnerable to forging attacks, Sona Rules

13. 보안기능 결정에 사용되는 부적절한 입력값 VanillaJS - ReactJS - ExpressJS ✓

가. 개요



응용 프로그램이 외부 입력값에 대한 신뢰를 전제로 보호 메커니즘을 사용하는 경우 공격자가 입력값을 조작할 수 있다면 보호 메커니즘을 우회할 수 있게 된다.

흔히 쿠키, 환경변수 또는 히든필드와 같은 입력값이 조작될 수 없다고 가정하지만 공격자는 다양한 방법을 통해 이러한 입력값들을 변경할 수 있고 조작된 내용은 탐지되지 않을 수 있다. 인증이나 인가와 같은 보안 결정이 이런 입력값(쿠키, 환경변수, 히든필드 등)에 기반을 두어 수행되는 경우 공격자는 입력값을 조작해 응용프로그램의 보안을 우회할 수 있다. 따라서, 충분한 암호화, 무결성 체크를 수행하고 이와 같은 메커니즘이 없는 경우엔 외부 사용자에 의한 입력값을 신뢰해서는 안 된다.

나. 안전한 코딩기법

상태 정보나 민감한 데이터 특히 사용자 세션 정보와 같은 중요 정보는 서버에 저장하고 보안확인 절차도 서버에서 실행한다. 보안설계 관점에서 신뢰할 수 없는 입력값이 응용 프로그램 내부로 들어올 수 있는 지점을 검토하고, 민감한 보안 기능 결정에 사용되는 입력값을 식별해 입력값에 대한 의존성을 없애는 구조로 변경 가능한지 분석한다.

다. 코드예제

다음은 안전하지 않은 코드로 쿠키에 저장된 권한 등급을 가져와 관리자인지 확인 후에 사용자의 패스워드를 초기화 하고 메일을 보내는 예제다. 쿠키에서 등급을 가져와 관리자 여부를 확인한다.

안전하지 않은 코드 예시

```
1:  const express = require('express');
2:
3:  router.get("/admin", (req, res) => {
4:    // 쿠키에서 권한 정보를 가져옴
5:    const role = req.cookies.role;
6:    if (role === "admin") {
7:      // 쿠키에서 가져온 권한 정보로 관리자 페이지 접속 처리
8:      return res.render('admin', { title: '관리자 페이지' } ));
9:    } else {
10:     return res.send("사용 권한이 없습니다.");
11:   }
12: });
```

중요 기능 수행을 결정하는 데이터는 위변조 가능성이 높은 쿠키보다 세션에 저장하도록 한다.

안전한 코드 예시

```
1:  const express = require('express');
2:  const session = require("express-session");
3:
4:  app.use(session({
5:    secret: 'test',
6:    resave: true,
7:    saveUninitialized: true,
8:    store: new MemoryStore({checkPeriod: 60 * 60 * 1000})
9:  }));
10:
11: router.get("/patched", (req, res) => {
12:   // 세션에서 권한 정보를 가져옴
13:   const role = req.session.role;
14:   if (role === "admin") {
15:     // 세션에서 가져온 권한 정보로 관리자 페이지 접속 처리
16:     return res.render('admin', { title: '관리자 페이지' } ));
17:   } else {
18:     return res.send("사용 권한이 없습니다.");
19:   }
20: });
```

라. 참고자료

① CWE-807: Reliance on Untrusted Inputs in a Security Decision, MITRE, https://cwe.mitre.org/data/definitions/807.html

② express-session, npm
https://www.npmjs.com/package/express-session

제2절 보안기능

보안기능(인증, 접근제어, 기밀성, 암호화, 권한관리 등)을 부적절하게 구현 시 발생할 수 있는 보안약점에는 적절한 인증 없는 중요기능 허용, 부적절한 인가 등이 있다.

1. 적절한 인증 없는 중요 기능 허용 VanillaJS - ReactJS - ExpressJS ✓

가. 개요

서버측에서는 NodeJS에서 제공되는 기본 기능을 사용해 인증 기능을 구현할 수도 있으며, ExpressJS나 nest.js와 같은 대표적인 서버 프레임워크에서 제공하는 기능을 사용하는 방법도 있다. 클라이언트측에서도 다양한 라이브러리를 사용해 인증 기능을 구현할 수 있지만, 궁극적으로는 서버측에서 안전한 인증이 지원되지 않으면 보안 문제가 발생할 수 있다.

나. 안전한 코딩기법

클라이언트의 보안 검사를 우회하여 서버에 접근하지 못하도록 설계하고 중요한 정보가 있는 페이지는 재인증을 적용한다. 또한 안전하다고 검증된 라이브러리나 프레임워크를 사용해야 한다.

다. 코드예제

다음은 패스워드 수정 시 수정을 요청한 패스워드와 DB에 저장된 사용자 패스워드 일치 여부를 확인하지 않고 처리하고 있으며 패스워드의 재확인 절차도 생략되어 취약한 코드 예시를 보여 준다.

안전하지 않은 코드 예시

```
1:  const express = require('express');
2:  const crypto = require("crypto");
```

```
3:
4:  router.post("/vuln", (req, res) => {
5:    const newPassword = req.body.newPassword;
6:    const user = req.session.userid;
7:    const hs = crypto.createHash("sha256")
8:    const newHashPassword = hs.update(newPassword).digest("base64");
9:
10:   // 현재 패스워드와 일치 여부를 확인하지 않고 업데이트
11:   updatePasswordFromDB(user, newHashPassword);
12:
13:   return res.send({message: "패스워드가 변경되었습니다.", userId, password, hashPassword});
14: });
```

DB에 저장된 사용자 패스워드와 변경을 요청한 패스워드의 일치 여부를 확인하고, 변경 요청한 패스워드와 재확인 패스워드가 일치하는지 확인 후 DB의 패스워드를 수정해 안전하게 코드를 적용할 수 있다.

안전한 코드 예시

```
1:  const express = require('express');
2:  const crypto = require("crypto");
3:
4:  router.post("/patched", (req, res) => {
5:    const newPassword = req.body.newPassword;
6:    const user = req.session.userid;
7:    const oldPassword = getPasswordFromDB(user);
8:    const salt = crypto.randomBytes(16).toString('hex');
9:
10:   const hs = crypto.createHash("sha256")
11:   const currentPassword = req.body.currentPassword;
12:   const currentHashPassword = hs.update(currentPassword + salt).digest("base64");
13:
14:   // 현재 패스워드 확인 후 사용자 정보 업데이트
15:   if (currentHashPassword === oldPassword) {
16:     const newHashPassword = hs.update(newPassword + salt).digest("base64");
17:     updatePasswordFromDB(user, newHashPassword);
18:     return res.send({ message: "패스워드가 변경되었습니다." });
19:   } else {
20:     return res.send({ message: "패스워드가 일치하지 않습니다." });
21:   }
22: });
```

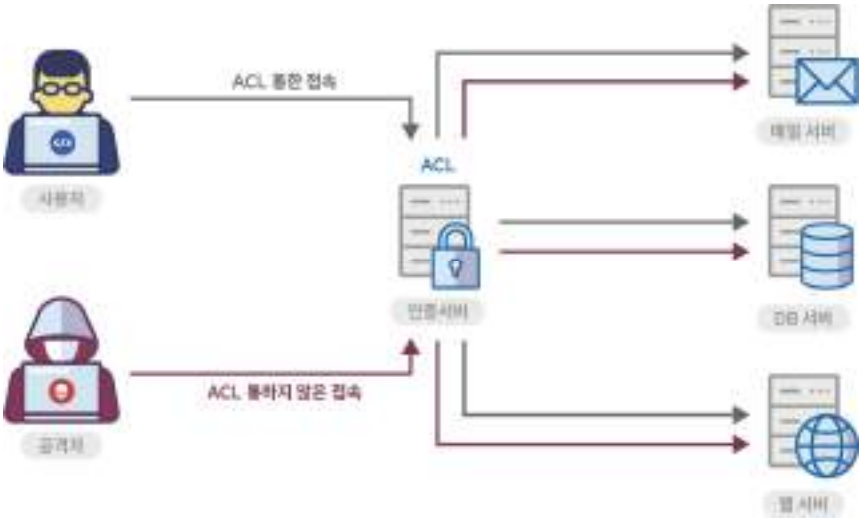
라. 참고자료

① CWE-306: Missing Authentication for Critical Function, MITRE, https://cwe.mitre.org/data/definitions/306.html

② Access Control, OWASP, https://www.owasp.org/index.php/Access_Control_Cheat_Sheet

2. 부적절한 인가 VanillaJS - ReactJS - ExpressJS ✓

가. 개요



사용자가 접근 가능한 모든 실행 경로에 대해서 접근 제어를 정확히 처리하지 않거나 불완전하게 검사하는 경우, 공격자는 접근 가능한 실행경로를 통해 정보를 유출할 수 있다.

나. 안전한 코딩기법

응용 프로그램이 제공하는 정보와 기능이 가지는 역할에 맞게 분리 개발함으로써 공격자에게 노출되는 공격 노출면(Attack Surface)을 최소화하고 사용자의 권한에 따른 ACL(Access Control List)을 관리한다. 공격 노출면은 공격자가 진입할 수 있거나 공격에 영향을 줄 수 있는 시스템 경계선 지점, 시스템 요소 또는 환경을 의미한다.

다. 코드예제

다음은 사용자의 권한 확인을 위한 별도의 통제 없이 사용자 입력값에 따라 삭제 작업을 수행하는 예시를 보여 준다.

안전하지 않은 코드 예시

```
1:  const express = require('express');
2:
```

```
3:  function deleteContentFromDB(contentId) {
4:    ...
5:  }
6:
7:  router.delete("/vuln", (req, res) => {
8:    const contentId = req.body.contentId;
9:    // 작업 요청을 하는 사용자의 권한 확인 없이 삭제 작업 수행
10:   deleteContentFromDB(contentId);
11:
12:   return res.send("삭제가 완료되었습니다.");
13: });
```

세션에 저장된 사용자 정보를 통해 해당 사용자가 수행할 작업에 대한 권한이 있는지 확인한 후, 권한이 있는 경우에만 작업을 수행하도록 해야 한다.

안전한 코드 예시

```
1:  const express = require('express');
2:
3:  function deleteContentFromDB(contentId) { ... }
4:
5:  router.delete("/patched", (req, res) => {
6:    const contentId = req.body.contentId;
7:    const role = req.session.role;
8:
9:    // 삭제 기능을 수행할 권한이 있는 경우에만 삭제 작업 수행
10:   if (role === "admin") {
11:     deleteContentFromDB(contentId);
12:     return res.send("삭제가 완료되었습니다.");
13:   } else {
14:     return res.send("권한이 없습니다.");
15:   }
16: });
```

라. 참고자료

① CWE-285: Improper Authorization, MITRE,
https://cwe.mitre.org/data/definitions/285.html

② Authorization, OWASP,
https://cheatsheetseries.owasp.org/cheatsheets/Authorization_Cheat_Sheet.html

3. 중요한 자원에 대한 잘못된 권한 설정 VanillaJS - ReactJS - ExpressJS ✓

가. 개요

응용프로그램이 중요한 보안관련 자원에 대해 읽기 또는 수정하기 권한을 의도하지 않게 허가할 경우, 권한을 갖지 않은 사용자가 해당 자원을 사용하게 된다. NodeJS에서는 fs.chmodSync 함수를 사용해 파일 생성, 수정 및 읽기 권한을 설정할 수 있다.

나. 안전한 코딩기법

설정 파일, 실행 파일, 라이브러리 등은 관리자에 의해서만 읽고 쓰기가 가능하도록 설정하고, 설정 파일과 같이 중요한 자원을 사용하는 경우 허가 받지 않은 사용자가 중요한 자원에 접근 가능한지 검사한다.

다. 코드예제

다음 예제는 /root/system_config 파일에 대해서 모든 사용자가 읽기, 쓰기, 실행 권한을 가지는 상황을 보여 준다.

안전하지 않은 코드 예시

```
1:  const fs = require("fs");
2:
3:  function writeFile() {
4:    // 모든 사용자가 읽기, 쓰기, 실행 권한을 가지게 됨
5:    fs.chmodSync("/root/system_config", 0o777);
```

```
6:     fs.open("/root/system_config", "w", function(err,fd) {
7:       if (err) throw err;
8:     });
9:     fs.writeFile("/root/system_config", "your config is broken", function(err) {
10:      if (err) throw err;
11:      console.log('write end');
12:    });
13:  }
```

주요 파일에 대해서는 최소 권한만 할당해야 한다. 구체적으로, 파일의 소유자라고 하더라도 기본적으로 읽기 권한만 부여해야 하며, 부득이하게 쓰기 권한이 필요한 경우에만 제한적으로 쓰기 권한을 부여해야 한다.

안전한 코드 예시

```
1:  const fs = require("fs");
2:
3:  function writeFile() {
4:    // 소유자 이외에는 권한을 가지지 않음
5:    fs.chmodSync("/root/system_config", 0o700);
6:    fs.open("/root/system_config", "w", function(err,fd) {
7:      if (err) throw err;
8:    });
9:    fs.writeFile("/root/system_config", "your config is broken", function(err) {
10:     if (err) throw err;
11:     console.log('write end');
12:   })
13: }
```

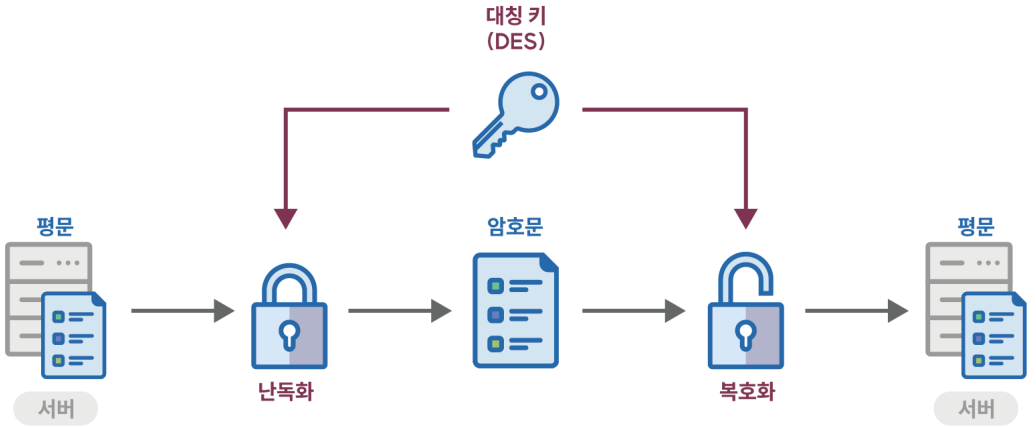
라. 참고자료

① CWE-732: Incorrect Permission Assignment for Critical Resource, MITRE, https://cwe.mitre.org/data/definitions/732.html

② File System, NodeJS
https://nodejs.org/api/fs.html#file-system

4. 취약한 암호화 알고리즘 사용 VanillaJS ✓ ReactJS ✓ ExpressJS ✓

가. 개요



개발자들은 환경설정 파일에 저장된 패스워드를 보호하기 위해 간단한 인코딩 함수를 이용해 패스워드를 감추는 방법을 사용하기도 한다. 하지만 base64와 같은 지나치게 간단한 인코딩 함수로는 패스워드를 제대로 보호할 수 없다.

정보보호 측면에서 취약하거나 위험한 암호화 알고리즘을 사용해서는 안 된다. 표준화되지 않은 암호화 알고리즘을 사용하는 것은 공격자가 알고리즘을 분석해 무력화시킬 수 있는 가능성을 높일 수도 있다. 몇몇 오래된 암호화 알고리즘의 경우는 컴퓨터의 성능이 향상됨에 따라 취약해지기도 해서, 예전에는 해독하는데 몇 십 억년이 걸릴 것이라고 예상되던 알고리즘이 며칠이나 몇 시간 내에 해독되기도 한다. RC2, RC4, RC5, RC6, MD4, MD5, SHA1, DES 알고리즘이 여기에 해당된다.

나. 안전한 코딩기법

자신만의 암호화 알고리즘을 개발하는 것은 위험하며, 학계 및 업계에서 이미 검증된 표준화된 알고리즘을 사용해야 한다. 기존에 취약하다고 알려진 DES, RC5 등의 암호알고리즘을 대신하여, 3TDEA, AES, SEED 등의 안전한 암호알고리즘으로 대체하여 사용한다. 또한, 업무관련 내용, 개인정보 등에 대한 암호 알고리즘 적용 시, 안전한 암호화 알고리즘을 사용해야 한다.

〈 암호알고리즘 검증기준 ver3.0 〉

분류		암호 알고리즘
최소 안전성 수준		• 112비트
블록암호 (운영모드)	ARIA	• 운영모드 – 기밀성(ECB, CBC, CFB, OFB, CTR) – 기밀성/인증(CCM, GCM)
	SEED	• 운영모드 – 기밀성(ECB, CBC, CFB, OFB, CTR) – 기밀성/인증(CCM, GCM)
	LEA	• 운영모드 – 기밀성(ECB, CBC, CFB, OFB, CTR) – 기밀성/인증(CCM, GCM)
	HIGHT	• 운영모드 – 기밀성(ECB, CBC, CFB, OFB, CTR)
해쉬함수	SHA-2	• SHA-224/256/384/512
	LSH	• LSH-224/256/384/512/512-224/512-256
	SHA-3	• SHA-3-224/256/384/512
메시지 인증	해쉬함수 기반	• HMAC
	블록암호 기반	• CMAC, GMAC
난수발생기	해쉬함수 기반	• Hash_DRBG, HMAC_DRBG
	블록암호 기반	• CTR_DRBG
공개키 암호	RSAES	• 공개키 길이 : 2048, 3072 • 해쉬함수 : SHA-224, SHA-256
전자서명	RSA-PSS	• 공개키 길이 : 2048, 3072 • 해쉬함수 : SHA-224, SHA-256
	KCDSA	• (공개키 길이, 개인키 길이) : (2048, 224), (2048, 256) • 해쉬함수 : SHA-224, SHA-256
	EC-KCDSA	• p-224, p-256, B-233, B-283, K-233, K-283 • 해쉬함수 : SHA-224, SHA-256
	ECDSA	• p-224, p-256, B-233, B-283, K-233, K-283 • 해쉬함수 : SHA-224, SHA-256
키 설정	DH	• (공개키 길이, 개인키 길이) : (2048, 224), (2048, 256)
	ECDH	• p-224, p-256, B-233, B-283, K-233, K-283
키 유도	KBKDF	• HMAC, CMAC
	PBKDF	• HMAC

다. 코드예제

다음 예제는 취약한 DES 알고리즘으로 암호화하는 예시다. DES 이외에 2TDEA, Blowfish 등의 취약한 알고리즘을 사용해선 안 된다.

안전하지 않은 코드 예시

```
1:  const crypto = require("crypto");
2:
3:  function getEncText(plainText, key) {
4:    // 취약한 알고리즘인 DES를 사용하여 안전하지 않음
5:    const cipherDes = crypto.createCipheriv('des-ecb', key, '');
6:
7:    const encryptedData = cipherDes.update(plainText, 'utf8', 'base64');
8:    const finalEnctypedData = cipherDes.final('base64');
9:
10:   return encryptedData + finalEnctypedData;
11: }
```

취약한 DES 알고리즘 대신 안전한 AES 암호화 알고리즘을 사용해야 한다. 블록 암호화에서 운영 모드를 ECB(Electronic Codebook) 모드로 사용할 경우 한 개의 블록만 해독되면 나머지 블록도 해독이 되는 단점이 있다. CBC(Cipher Block Chaining) 모드는 평문의 각 블록이 XOR 연산을 통해 이전 암호문과 연산이 되기 때문에 같은 평문이라도 암호문이 서로 다르다. 이러한 특성으로 보안성이 ECB 모드보다 높다.

안전한 코드 예시

```
1:  const crypto = require("crypto");
2:
3:  function getEncText(plainText, key, iv) {
4:    // 권장 알고리즘인 AES를 사용하여 안전함
5:    const cipherAes = crypto.createCipheriv('aes-256-cbc', key, iv);
6:
7:    const encryptedData = cipherAes.update(plainText, 'utf8', 'base64');
8:    const finalEnctypedData = cipherAes.final('base64');
9:
10:   return encryptedData + finalEnctypedData;
11: }
```

다음 예제는 취약한 MD5 해쉬 함수를 사용하는 예시다. 암호 알고리즘과 마찬가지로, 해쉬 함수도 수학적으로 취약한 것으로 확인된 MD5와 같은 함수를 사용하면 역계산은 불가하나 레인보우테이블을 이용하는 등의 방법으로 평문을 알아낼 수 있다.

안전하지 않은 코드 예시

```
1:  const crypto = require("crypto");
2:
3:  function makeMd5(plainText) {
4:    // 취약한 md5 해쉬함수 사용
5:    const hashText = crypto.createHash('md5').update(plainText).digest("hex");
6:
7:    return hashText;
8:  }
```

아래 코드처럼 수학적으로 안전하다고 알려진 sha-256 해쉬 함수 등을 적용해야 한다.

안전한 코드 예시

```
1:  const crypto = require("crypto");
2:
3:  function makeSha256(plainText) {
4:    const salt = crypto.randomBytes(16).toString('hex');
5:
6:    // 안전한 sha-256 해쉬함수 사용
7:    const hashText = crypto.createHash('sha256').update(plainText + salt).digest("hex");
8:
9:    return hashText;
10: }
```

라. 참고자료

① CWE-327: Use of a Broken or Risky Cryptographic Algorithm, MITRE,
https://cwe.mitre.org/data/definitions/327.html

② Approved Security functions FIPS 140 Annex a, NIST
http://csrc.nist.gov/publications/fips/fips140-2/fips1402annexa.pdf

③ Crypto, NodeJS
https://nodejs.org/api/crypto.html

5. 암호화되지 않은 중요정보

VanillaJS ✓ ReactJS ✓ ExpressJS ✓

가. 개요

많은 응용 프로그램은 메모리나 디스크 상에서 중요한 정보(개인정보, 인증정보, 금융정보 등)를 처리한다. 이러한 중요 정보가 제대로 보호되지 않을 경우, 보안 문제가 발생하거나 데이터의 무결성이 깨질 수 있다. 특히 사용자 또는 시스템의 중요 정보가 포함된 데이터를 평문으로 송·수신 또는 저장 시 인가되지 않은 사용자에게 민감한 정보가 노출될 수 있다.

나. 안전한 코딩기법

개인정보(주민등록번호, 여권번호 등), 금융정보(카드번호, 계좌번호 등), 패스워드 등 중요정보를 저장하거나 통신채널로 전송할 때는 반드시 암호화 과정을 거쳐야 하며 중요정보를 읽거나 쓸 경우에 권한인증 등을 통해 적합한 사용자만 중요정보에 접근하도록 해야 한다.

가능하다면 SSL 또는 HTTPS 등과 같은 보안 채널을 사용해야 한다. 보안 채널을 사용하지 않고 브라우저 쿠키에 중요 데이터를 저장하는 경우, 쿠키 객체에 보안속성을 설정해(Ex. secure = True) 중요 정보의 노출을 방지할 수 있다.

다. 코드예제

• 중요정보 평문저장

아래 예제는 사용자로부터 전달받은 패스워드 암호화를 누락한 경우이다.

안전하지 않은 코드 예시

```
1:  function updatePass(dbconn, password, user_id) {
2:    // 암호화되지 않은 비밀번호를 DB에 저장하는 경우 위험함
3:    const sql = 'UPDATE user SET password=? WHERE user_id=?';
4:    const params = [password, user_id];
5:
6:    dbconn.query(sql, params, function(err, rows, fields){
7:      if (err) console.log(err);
8:    })
9:  }
```

아래는 해쉬 알고리즘을 이용하여 단방향 암호화 이후에 패스워드를 저장하는 예시를 보여 준다. 이 때, 해쉬 함수 또한 SHA256과 같이 안정성이 검증된 알고리즘을 사용해야 한다.

안전한 코드 예시

```
1:  const crypto = require("crypto");
2:
3:  function updatePass(dbconn, password, user_id, salt) {
4:    // 단방향 암호화를 이용하여 비밀번호를 암호화
5:    const sql = 'UPDATE user SET password=? WHERE user_id=?';
6:    const hashPw = crypto.createHash('sha256').update(password + salt, 'utf-8').digest('hex');
7:    const params = [hashPw, user_id];
8:    dbconn.query(sql, params, function(err, rows, fields){
9:      if (err) console.log(err);
10:   })
11: }
```

• 중요정보 평문전송

아래 예제는 인자값으로 전달 받은 패스워드를 암호화 없이 네트워크를 통해 전송하는 예시를 포함한다. 전달 받은 패스워드가 암호화가 되어 있지 않을 경우 패킷 스니핑을 통하여 패스워드가 노출될 수 있다.

안전하지 않은 코드 예시

```
1:  const { io } = require("socket.io-client");
2:  const socket = io("http://localhost:3000");
3:
4:  function sendPassword(password) {
5:    // 패스워드를 암호화 하지 않고 전송하여 안전하지 않음
6:    socket.emit("password", password);
7:  }
8:
9:  socket.on("password", function(data) {
10:   if (data === 'success') {
11:     console.log("\nSuccess to send a message to a server \n")
12:   }
13: });
```

아래는 네트워크를 통해 전달되는 패스워드가 노출되지 않도록 암호화하여 전송하는 예시를 보여 준다.

안전한 코드 예시

```
1:  const { io } = require("socket.io-client");
2:  const crypto = require("crypto");
3:  const socket = io("http://localhost:3000");
4:  const PASSWORD = getPassword();
5:
6:  function aesEncrypt(plainText) {
7:    const key = getCryptKey();
8:    const iv = getCryptIV();
9:    const cipherAes = crypto.createCipheriv('aes-256-cbc', key, iv);
10:
11:   const encryptedData = cipherAes.update(plainText, 'utf8', 'base64');
12:   const finalEncryptedData = cipherAes.final('base64');
13:
14:   return encryptedData + finalEncryptedData;
15: }
16:
17: function sendPassword(password) {
18:   // 패스워드 등 중요정보는 암호화하여 전송하는 것이 안전함
19:   const encPassword = aesEncrypt(password);
20:   socket.emit("password", encPassword);
21: }
22:
23: socket.on("password", function(data) {
24:   if (data === 'success') {
25:     console.log("\nSuccess to send a message to a server \n")
26:   }
27: });
```

라. 참고자료

① CWE-312: Cleartext Storage of Sensitive Information, MITRE, https://cwe.mitre.org/data/definitions/312.html

② CWE-319: Cleartext Transmission of Sensitive Information, MITRE, https://cwe.mitre.org/data/definitions/319.html

③ Password Plaintext Storage, OWASP, https://owasp.org/www-community/vulnerabilities/Password_Plaintext_Storage

6. 하드코드된 중요정보 VanillaJS ✓ ReactJS ✓ ExpressJS ✓

가. 개요

프로그램 코드 내부에 하드코드된 패스워드를 포함하고, 이를 이용해 내부 인증에 사용하거나 외부 컴포넌트와 통신을 하는 경우 관리자의 정보가 노출될 수 있어 위험하다. 또한 하드코드된 암호화 키를 사용해 암호화를 수행하면 암호화된 정보가 유출될 가능성이 높아진다. 암호키의 해쉬를 계산해 저장하더라도 역계산이 가능해 무차별 공격(Brute-Force)공격에는 취약할 수 있다.

나. 안전한 코딩기법

패스워드는 암호화 후 별도의 파일에 저장하여 사용한다. 또한 중요 정보 암호화 시 상수가 아닌 암호화 키를 사용하도록 하며, 암호화가 잘 되었더라도 코드 내부에 상수 형태의 암호화 키를 주석으로 달거나 저장하지 않도록 한다.

다. 코드예제

소스코드에 패스워드 또는 암호화 키와 같은 중요 정보를 하드코딩 하는 경우, 중요 정보가 노출될 수 있어 위험하다.

안전하지 않은 코드 예시

```
1:  const express = require('express');
2:  const mysql = require("mysql");
3:  const crypto = require("crypto");
4:
5:  const dbQuery = "SELECT email, name FROM user WHERE name = 'test'";
6:
7:  router.get("/vuln", (req, res) => {
8:    // 데이터베이스 연결에 필요한 인증 정보가 평문으로 하드코딩되어 있음
```

```
 9:    const connection = mysql.createConnection({
10:      host: 'localhost',
11:      user: 'root',
12:      password: 'root1234',
13:      database: 'javascript',
14:      port: 3306,
15:    });
16:
17:    connection.query(
18:      dbQuery, (err, result) => {
19:        if (err) return res.send(err);
20:          return res.send(result);
21:    })
22:  });
```

패스워드와 같은 중요 정보는 안전한 암호화 방식으로 암호화 후 별도의 분리된 공간(파일)에 저장해야 하며, 암호화된 정보 사용 시 복호화 과정을 거친 후 사용해야 한다.

안전한 코드 예시

```
 1:  const express = require('express');
 2:  const mysql = require("mysql");
 3:  const crypto = require("crypto");
 4:
 5:  const dbQuery = "SELECT email, name FROM user WHERE name = 'test'";
 6:  const key = getCryptKey(); // 32bytes
 7:  const iv = getCryptIV(); // 16bytes
 8:
 9:  router.get("/patched", (req, res) => {
10:    // 설정파일에 암호화 되어 있는 user, password 정보를 가져와 복호화 한 후 사용
11:    const decipher = crypto.createDecipheriv('aes-256-cbc', key, iv);
12:    const user = decipher.update(process.env.USER, 'base64', 'utf8') + decipher.final('utf8');
13:    const decipher2 = crypto.createDecipheriv('aes-256-cbc', key, iv);
14:    const password = decipher2.update(process.env.PASSWORD, 'base64', 'utf8') + decipher2.final
15: ('utf8');
16:
17:    // DB 연결 정보도 설정파일에서 가져와 사용
18:    const connection = mysql.createConnection({
19:      host: process.env.DB_HOST,
20:      user: user,
21:      password: password,
22:      database: process.env.DB_NAME,
23:      port: process.env.PORT,
24:    });
25:   ...
```

라. 참고자료

① CWE-259: Use of Hard-coded Password, MITRE,
https://cwe.mitre.org/data/definitions/259.html

② CWE-321: Use of Hard-coded Cryptographic Key, MITRE,
https://cwe.mitre.org/data/definitions/321.html

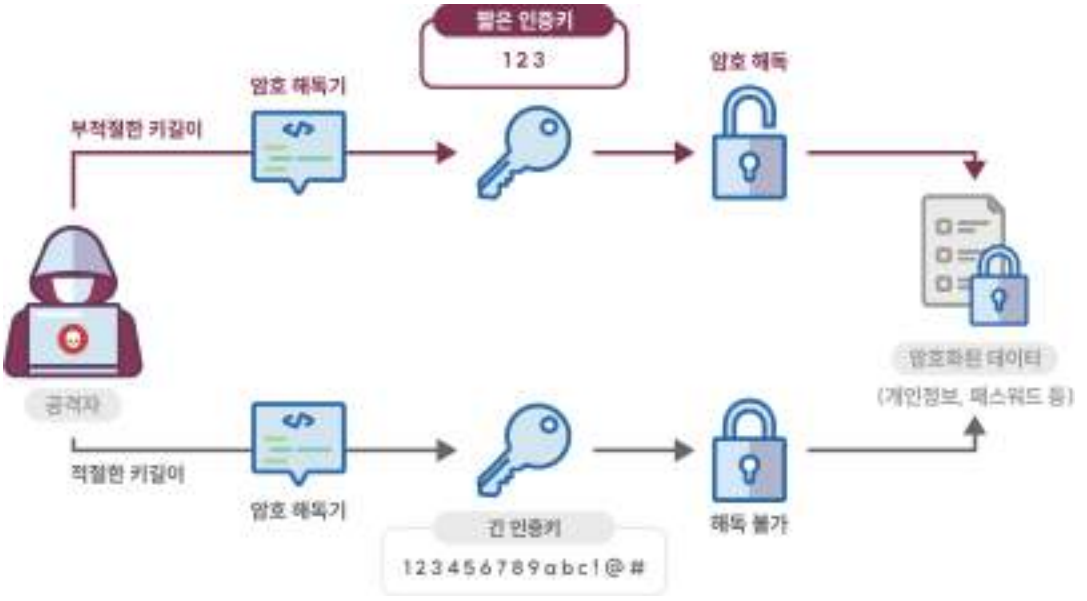
③ Use of hard-coded password, OWASP,
https://owasp.org/www-community/vulnerabilities/Use_of_hard-coded_password

④ Password Management Hardcoded Password, OWASP,
https://owasp.org/www-community/vulnerabilities/Password_Management_Hardcoded_Password

⑤ Hard-coded credentials are security-sensitive, sona-rules
https://rules.sonarsource.com/javascript/RSPEC-2068

7. 충분하지 않은 키 길이 사용 VanillaJS ✓ ReactJS ✓ ExpressJS ✓

가. 개요



짧은 길이의 키를 사용하는 것은 암호화 알고리즘을 취약하게 만들 수 있다. 키는 암호화 및 복호화에 사용되는데, 검증된 암호화 알고리즘을 사용하더라도 키 길이가 충분히 길지 않으면 짧은 시간 안에 키를 찾아낼 수 있고 이를 이용해 공격자가 암호화된 데이터나 패스워드를 복호화 할 수 있게 된다.

암호 알고리즘 및 키 길이 선택 시, 암호 알고리즘의 안전성 유지기간과 보안강도별 암호 알고리즘 키 길이 비교표를 기반으로 암호 알고리즘 및 키 길이를 선택해야 한다.

〈 보안강도별 암호 알고리즘 비교표 〉

보안강도	대칭키 암호 알고리즘 (보안강도)	해시함수 (보안강도)	공개키 암호 알고리즘				암호 알고리즘 안전성 유지기간 (년도)
			인수분해 (비트)	이산대수		타원곡선 암호(비트)	
				공개키(비트)	개인키(비트)		
112 비트	112	112	2048	2048	224	224	2011년에서 2030년까지
128 비트	128	128	3072	3072	256	256	2030년 이후
192 비트	192	192	7680	7680	384	384	
256비트	256	256	15360	15360	512	512	

나. 안전한 코딩기법

RSA 알고리즘은 적어도 2,048 비트 이상의 길이를 가진 키와 함께 사용해야 하고, 대칭 암호화 알고리즘(Symmetric Encryption Algorithm)의 경우에는 적어도 보안 강도 112비트 이상을 지원하는 알고리즘을 사용해야 한다.

다. 코드예제

보안성이 강한 RSA 알고리즘을 사용하는 경우에도 키 크기를 작게 설정하면 프로그램의 보안약점이 될 수 있다.

안전하지 않은 코드 예시

```
1:  const crypto = require("crypto");
2:
3:  function vulnMakeRsaKeyPair() {
4:    // RSA키 길이를 1024 비트로 설정하는 경우 안전하지 않음
5:    const { publicKey, privateKey } = crypto.generateKeyPairSync('rsa',
6:    {
7:      modulusLength: 1024,
8:      publicKeyEncoding: { type: "spki", format: 'pem' },
9:      privateKeyEncoding: { type: "pkcs8", format: 'pem' }
10:   });
11:   return { PRIVATE: publicKey, PUBLIC: privateKey }
12: }
13:
14: function vulnMakeEcc() {
15:   // ECC 키 길이가 224비트 이하이면 안전하지 않음
16:   const { publicKey, privateKey } = crypto.generateKeyPairSync('ec', {
17:     namedCurve: 'secp192k1',
18:     publicKeyEncoding: { type: 'spki', format: 'der' },
19:     privateKeyEncoding: { type: 'pkcs8', format: 'der' }
20:   });
21:   return { PRIVATE: publicKey.toString('hex'), PUBLIC: privateKey.toString('hex') }
22: }
```

RSA, DSA의 경우 키의 길이는 적어도 2048 비트를, ECC의 경우 224 비트 이상으로 설정해야 안전하다. 다음은 안전한 키 길이를 사용해 RSA 및 ECC 키를 생성한 예제다.

안전한 코드 예시

```
1:  const crypto = require("crypto");
2:
3:  function vulnMakeRsaKeyPair() {
4:    // RSA키 길이를 2048 비트로 설정해 안전함
5:    const { publicKey, privateKey } = crypto.generateKeyPairSync('rsa',
6:    {
7:      modulusLength: 2048,
8:      publicKeyEncoding: { type: "spki", format: 'pem' },
9:      privateKeyEncoding: { type: "pkcs8", format: 'pem' }
10:   });
11:   return { PRIVATE: publicKey, PUBLIC: privateKey }
12: }
13:
14: function vulnMakeEcc() {
15:   // ECC 키 길이를 256 비트로 설정해 안전함
16:   const { publicKey, privateKey } = crypto.generateKeyPairSync('ec', {
17:     namedCurve: 'secp256k1',
18:     publicKeyEncoding: { type: 'spki', format: 'der' },
19:     privateKeyEncoding: { type: 'pkcs8', format: 'der' }
20:   });
21:   return { PRIVATE: publicKey.toString('hex'), PUBLIC: privateKey.toString('hex') }
22: }
```

라. 참고자료

① CWE-326: Inadequate Encryption Strength, MITRE,
https://cwe.mitre.org/data/definitions/326.html

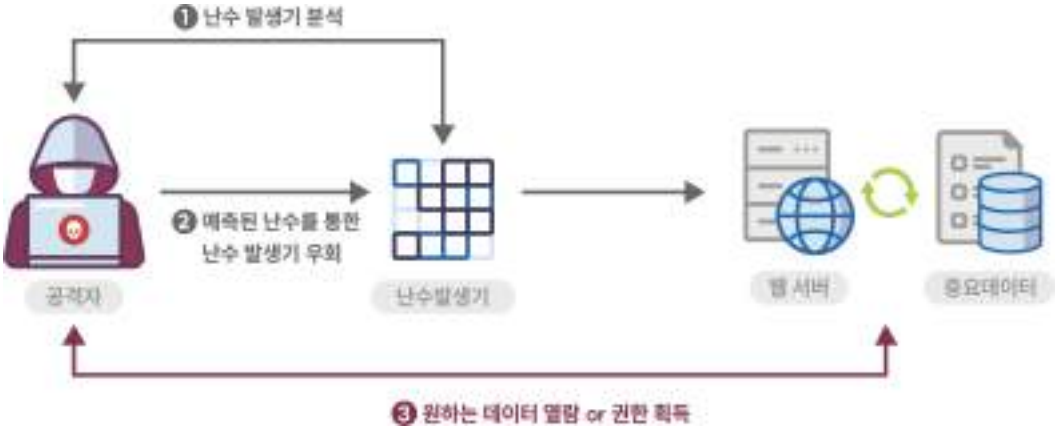
② FEDERAL INFORMATION PROCESSING STANDARDS PUBLICATION (FIPS PUB 186-4), NIST
https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf

③ 암호 알고리즘 및 키 길이 이용 안내서, KISA,
https://www.kisa.or.kr/2060305/form?postSeq=5&lang_type=KO#fnPostAttachDownload

8. 적절하지 않은 난수 값 사용

VanillaJS ✓ ReactJS ✓ ExpressJS ✓

가. 개요



예측 불가능한 숫자가 필요한 상황에서 예측 가능한 난수를 사용한다면, 공격자가 생성되는 다음 숫자를 예상해 시스템을 공격할 수 있다.

나. 안전한 코딩기법

난수 발생기에서 시드(Seed)를 사용하는 경우에는 고정된 값을 사용하지 않고 예측하기 어려운 방법으로 생성된 값을 사용한다.

자바스크립트 NodeJS 엔진에서는 crypto.getRandomBytes를 사용해 암호학적으로 안전한 의사 난수 바이트를 생성할 수 있다. 만약 이렇게 생성한 바이트를 숫자로 변환하게 되면 안정성이 깨질 수도 있다는 점을 주의해야 한다. 브라우저에서는 RandomSource.getRandomValues를 사용해 암호학적으로 안전한 의사 랜덤 숫자를 생성할 수 있다.

다. 코드예제

random 데이터 생성 시에는 반드시 유추하기 어려운 seed 값을 이용하여 난수를 생성해야 하며, 이렇게 생성된 난수라 하더라도 강도가 낮기 때문에 주요 보안 기능을 위한 난수 이용 시에는 안전하지 않다. 아래는 안전하지 않은 코드 예제로 암호학적으로 안전하지 않은 Math.random 함수를 사용해 난수를 생성하고 있다.

안전하지 않은 코드 예시

```
1:  function getOtpNumber() {
2:    let randomStr = '';
3:    // Math.random 라이브러리는 보안기능에 사용하면 위험함
4:    for (let i = 0; i < 6; i++) {
5:      randomStr += String(Math.floor(Math.random() * 10))
6:    }
7:    return randomStr;
8:  }
```

다음 코드는 NodeJS 환경에서 crypto 라이브러리를 사용해 암호학적으로 안전한 난수 값을 생성하는 예제를 보여 준다.

안전한 코드 예시

```
1:  const crypto = require("crypto");
2:
3:  function getOtpNumber() {
4:    // 보안기능에 적합한 난수 생성용 crypto 라이브러리 사용
5:    const array = new Uint32Array(1);
6:    // 브라우저에서는 crypto 대신에 window.crypto를 사용
7:    const randomStr = crypto.getRandomValues(array);
8:    let result;
9:    for (let i = 0; i < randomStr.length; i++) {
10:     result = array[i];
11:   }
12:   return String(result).substring(0, 6);
13: }
```

라. 참고자료

① CWE-330: Use of Insufficiently Random Values, MITRE,
https://cwe.mitre.org/data/definitions/330.html

② Insecure Randomness, OWASP,
https://owasp.org/www-community/vulnerabilities/Insecure_Randomness

③ Math.random(), mdn web docs
https://developer.mozilla.org/ko/docs/Web/JavaScript/Reference/Global_Objects/Math/random

④ getRandomValues(), mdn web docs
https://developer.mozilla.org/en-US/docs/Web/API/Crypto/getRandomValues

9. 취약한 패스워드 허용 VanillaJS ✓ ReactJS ✓ ExpressJS ✓

가. 개요

강한 패스워드 조합 규칙을 사용하도록 강제하지 않으면 패스워드 공격으로부터 사용자 계정이 위험에 빠질 수 있다. 안전한 패스워드를 생성하기 위해서는 「패스워드 선택 및 이용 안내서」에서 제시하는 패스워드 설정 규칙을 적용해야 한다.

나. 안전한 코딩기법

패스워드 생성 시 강한 조건 검증을 수행한다. 패스워드(패스워드)는 숫자와 영문자, 특수문자 등을 혼합하여 사용하고, 주기적으로 변경하여 사용하도록 해야 한다.

다. 코드예제

사용자가 입력한 패스워드에 대한 복잡도 검증 없이 가입 승인 처리를 수행하고 있다.

안전하지 않은 코드 예시

```
1:  const express = require('express');
2:  const mysql = require("mysql");
3:  const connection = mysql.createConnection( ... );
4:
5:  router.post("/vuln", (req, res) => {
6:    const con = connection;
7:    const { email, password, name } = req.body;
8:    // 패스워드 생성 규칙 검증 없이 회원 가입 처리
9:    con.query(
10:     "INSERT INTO user (email, password, name) VALUES (?, ?, ?)",
11:     [email, password, name],
12:     (err, result) => {
13:       if (err) return res.send(err);
14:       return res.send("회원가입 성공");
15:     });
16: });
```

사용자 계정 보호를 위해 회원가입 시 패스워드 복잡도와 길이를 검증 후 가입 승인처리를 수행해야 한다. 코드 내의 특수문자('!@#$%^&*')는 기업 내부 정책에 따라 변경하여 사용하면 되며, 패스워드를 숫자로만 10자리로 구성할 경우 취약할 수 있으니 사용자가 안전한 패스워드로 변경할 수 있도록 안내해야 한다.

안전한 코드 예시

```
1:  const express = require('express');
2:  const mysql = require("mysql");
3:  const connection = mysql.createConnection(...);
4:
5:  router.post("/patched", (req, res) => {
6:    const con = connection;
7:    const { email, password, name } = req.body;
8:
9:    function checkPassword(password) {
10:     // 2종 이상 문자로 구성된 8자리 이상 비밀번호 검사 정규식
11:     const pt1 = /^(?=.*[A-Z])(?=.*[a-z])[A-Za-z\d!@#$%^&*]{8,}$/;
12:     const pt2 = /^(?=.*[A-Z])(?=.*\d)[A-Za-z\d!@#$%^&*]{8,}$/;
13:     const pt3 = /^(?=.*[A-Z])(?=.*[!@#$%^&*])[A-Za-z\d!@#$%^&*]{8,}$/;
14:     const pt4 = /^(?=.*[a-z])(?=.*\d)[A-Za-z\d!@#$%^&*]{8,}$/;
15:     const pt5 = /^(?=.*[a-z])(?=.*[!@#$%^&*])[A-Za-z\d!@#$%^&*]{8,}$/;
16:     const pt6 = /^(?=.*\d)(?=.*[!@#$%^&*])[A-Za-z\d!@#$%^&*]{8,}$/;
17:     // 문자 구성 상관없이 10자리 이상 비밀번호 검사 정규식
18:     const pt7 = /^[A-Za-z\d!@#$%^&*]{10,}$/;
19:
20:     for (let pt of [pt1, pt2, pt3, pt4, pt5, pt6, pt7]) {
21:       console.log(pt.test(password));
22:       if (pt.test(password)) return true;
23:     }
24:     return false;
25:   }
26:
27:   if (checkPassword(password)) {
28:     con.query(
29:       "INSERT INTO user (email, password, name) VALUES (?, ?, ?)",
30:       [email, password, name],
31:       (err, result) => {
32:         if (err) return res.send(err);
33:         return res.send("회원가입 성공");
34:       }
35:     );
36:   } else {
37:     return res.send("비밀번호는 영문 대문자, 소문자, 숫자, 특수문자 조합 중 2가지 이상 8자리이거나
38:     문자 구성 상관없이 10자리 이상이어야 합니다.");
39:   }
40: });
```

라. 참고자료

① CWE-521: Weak Password Requirements, MITRE,
https://cwe.mitre.org/data/definitions/521.html

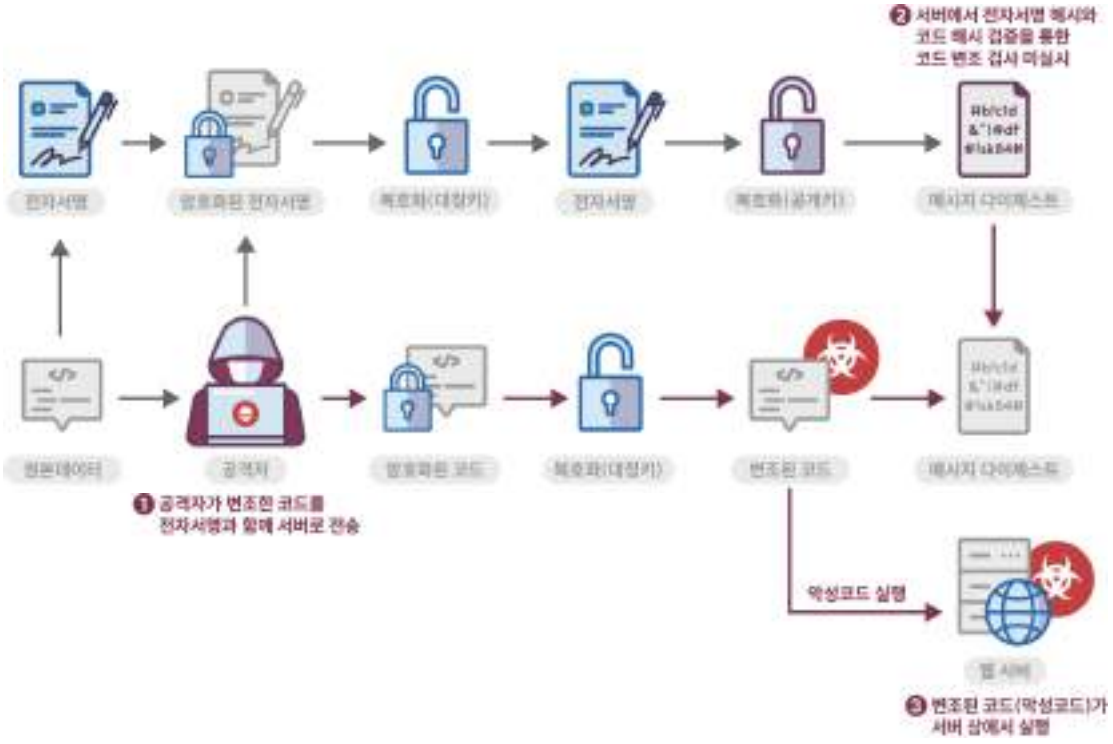
② Authentication Cheat Sheet, OWASP,
https://cheatsheetseries.owasp.org/cheatsheets/Authentication_Cheat_Sheet.html

③ JavaScript-mdn-web-docs - MDN contributors
https://devdocs.io/javascript/global_objects/regexp

④ 패스워드 선택 및 이용 안내서
https://seed.kisa.or.kr/kisa/Board/53/detailView.do

10. 부적절한 전자서명 확인 VanillaJS - ReactJS - ExpressJS ✓

가. 개요



프로그램, 라이브러리, 코드의 전자서명에 대한 유효성 검증이 적절하지 않아 공격자의 악의적인 코드가 실행 가능한 보안약점으로, 클라이언트와 서버 사이의 주요 데이터 전송, 파일 다운로드 시 발생할 수 있다. 데이터 전송 또는 다운로드 시 함께 전달되는 전자서명은 원문 데이터의 암호화된 해쉬 값으로, 수신측에서 이 서명을 검증해 데이터 변조 여부를 확인할 수 있다. 단순히 해쉬 기반 검증만 사용할 경우 해쉬 자체를 변조해 악성코드를 전달할 수 있지만, 전자서명을 사용하게 되면 원문 데이터에 대한 해쉬 자체도 안전하게 보호할 수 있다.

나. 안전한 코딩기법

주요 데이터 전송 또는 다운로드 시 데이터에 대한 전자서명을 함께 전송하고, 수신측에서는 전달 받은 전자서명을 검증해 파일의 변조 여부를 확인해야 한다.

다. 코드예제

다음은 송신측이 데이터와 함께 전달한 전자서명을 수신측에서 별도로 처리하지 않고 데이터를 그대로 신뢰해 데이터 내부에 포함된 자바스크립트 코드가 실행되는 취약한 예시를 보여 준다.

안전하지 않은 코드 예시

```
1:  const express = require('express');
2:  const crypto = require("crypto");
3:  const fs = require('fs');
4:
5:  router.post("/vuln", (req, res) => {
6:    // 클라이언트로부터 전달받은 데이터(전자서명을 수신 처리 하지 않음)
7:    const { encrypted_msg } = req.body;
8:    let secret_key;
9:    fs.readFile('/keys/secret_key.out', 'utf8', (err, data) => {
10:     if (err) {
11:       console.error(err);
12:       return;
13:     }
14:     secret_key = data;
15:   }
16:
17:   // 대칭키로 클라이언트가 전달한 코드 복호화
18:   // (decrypt_with_symmetric_key 함수는 임의의 함수명으로 세부적인 복호화 과정은 생략함)
19:   const decrypted = decrypt_with_symmetric_key(encrypted_msg, secret_key);
20:
21:   // 클라이언트로부터 전달 받은 코드 실행
22:   eval(decrypted);
23:   res.send('요청한 코드를 실행했습니다');
24: });
```

중요한 정보 또는 기능 실행으로 연결되는 데이터를 전달하는 경우, 반드시 전자서명을 함께 전송해야 하며 수신측에서는 전자서명을 확인해 송신측에서 보낸 데이터의 무결성을 검증해야 한다. 만약 송수신측 언어가 다른 경우 사용한 암호 라이브러리에 따라 데이터 인코딩 방식에 차이가 있으니 반드시 코드 배포 전 서명 검증에 필요한 복호화 과정이 정상적으로 잘 처리되는지 검증해야 한다.

안전한 코드 예시

```
1:  // 전자서명 검증에 사용한 코드는 의존한 패키지 및 송신측 언어에 따라
2:  // 달라질 수 있으며, 서명을 제공한 서버의 공개키로 복호화한 전자서명과 원본 데이터 해쉬값의
3:  // 일치 여부를 검사하는 코드를 포함
4:  const express = require('express');
5:  const crypto = require("crypto");
6:
7:  router.post("/patched", (req, res) => {
8:
9:    const { encrypted_msg, encrypted_sig, client_pub_key } = req.body;
10:   let secret_key;
11:   fs.readFile('/keys/secret_key.out', 'utf8', (err, data) => {
12:     if (err) {
13:       console.error(err);
14:       return;
15:     }
16:     secret_key = data;
17:   }
18:   // 대칭키로 클라이언트 데이터 및 전자서명 복호화
19:   const decrypted_msg = decrypt_with_symmetric_key(encrypted_msg);
20:   const decrypted_sig = decrypt_with_symmetric_key(encrypted_sig);
21:
22:   // 전자서명 검증에 통과한 경우에만 데이터 실행
23:   if (verify_digit_signature(decrypted_msg, decrypted_sig, clint_pub_key)) {
24:     eval(decrypted_msg);
25:     res.send('요청한 코드를 실행했습니다');
26:   } else {
27:     res.send('[!] 에러 - 서명이 올바르지 않습니다.');
28:   }
29: });
```

라. 참고자료

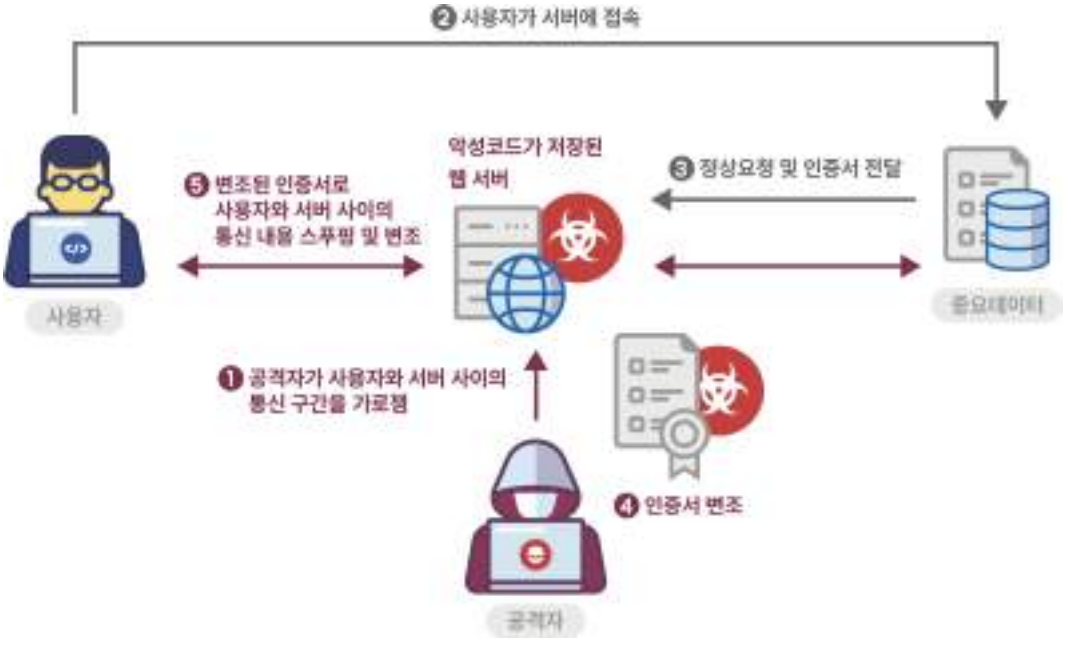
① CWE-347: Improper Verification of Cryptographic Signature, MITRE, https://cwe.mitre.org/data/definitions/347.html

② Security Consideration for Code Signing, NIST, https://csrc.nist.gov/CSRC/media/Publications/white-paper/2018/01/26/security-considerations-for-code-signing/final/documents/security-considerations-for-code-signing.pdf

③ Class: verify, NodeJS https://nodejs.org/api/crypto.html#class-verify

11. 부적절한 인증서 유효성 검증 VanillaJS - ReactJS - ExpressJS ✓

가. 개요



인증서가 유효하지 않거나 악성인 경우, 공격자가 호스트와 클라이언트 사이의 통신 구간을 가로채 신뢰하는 엔티티인 것처럼 속일 수 있다. 이로 인해 대상 호스트가 신뢰 가능한 것으로 믿고 악성 호스트에 연결하거나 신뢰된 호스트로부터 전달받은 것처럼 보이는 스푸핑 된(또는 변조된 데이터)를 아무런 의심 없이 수신하는 상황이 발생할 수 있다.

나. 안전한 코딩기법

데이터 통신에 인증서를 사용하는 경우 송신측에서 전달한 인증서가 유효한지 검증한 후 데이터를 송수신해야 한다. 언어에서 기본으로 제공되는 검증 함수가 존재하지 않거나 일반적이지 않은 방식으로 인증서를 생성한 경우 암호화 패키지를 사용해 별도의 검증 코드를 작성해야 한다.

다. 코드예제

다음은 SSL 기반 웹 서버 연결 예시로, 클라이언트 측에서 통신 대상 서버를 인증하지 않고 접속하는 상황을 보여 준다. 이 경우, 서버를 신뢰할 수 없으며 클라이언트 시스템에 영향을 주는 악성 데이터를 수신할 수 있다.

안전하지 않은 코드 예시

```
const https = require('https');

const getServer = () => {

  const options = {
    hostname: "dangerous.website",
    port: 443,
    method: "GET",
    path: "/",
    // 유효하지 않은 인증서를 가지고 있어도 무시하는 옵션으로 안전하지 않음
    rejectUnauthorized: false
  };
  https.request(
    options, (response) => {
    console.log('response - ', response.statusCode);
    }
  );
});
```

rejectUnauthorized 옵션을 true로 설정하면 정상적이지 않은 인증서를 가진 서버 연결 시 예외가 발생하고 연결이 수립되지 않아 악성 서버로부터 클라이언트를 안전하게 보호할 수 있다.

안전한 코드 예시

```
const express = require('express');
const https = require('https');

const getServer = () => {

  const options = {
    hostname: "dangerous.website",
    port: 443,
    method: "GET",
    path: "/",
    // 유효하지 않은 인증서 발견 시 예외 발생
    rejectUnauthorized: true
  };
  const hreq = https.request(
    options, (response) => {
    console.log('response - ', response.statusCode);
    }
  );
  hreq.on('error', (e) => {
    console.error('에러발생 - ', e);
  });
});
```

라. 참고자료

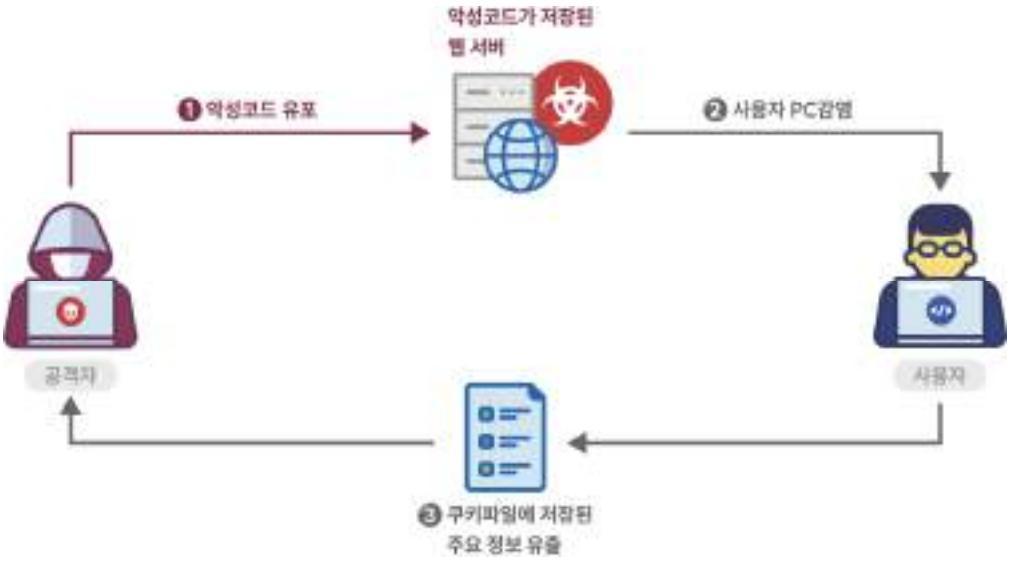
① CWE-295: Improper Certificate Validation, MITRE,
https://cwe.mitre.org/data/definitions/295.html

② Identification and Authentication Failures, OWASP,
https://owasp.org/Top10/A07_2021-Identification_and_Authentication_Failures/

③ https, NodeJS,
https://nodejs.org/api/https.html#httpsrequestoptions-callback

12. 사용자 하드디스크에 저장되는 쿠키를 통한 정보 노출 VanillaJS - ReactJS - ExpressJS ✓

가. 개요



대부분의 웹 응용프로그램에서 쿠키는 메모리에 상주하며, 브라우저가 종료되면 사라진다. 개발자가 원하는 경우, 브라우저 세션에 관계없이 지속적으로 쿠키 값을 저장하도록 설정할 수 있다. 이 경우 정보는 디스크에 기록되고, 다음 브라우저 세션 시작 시 메모리에 로드 된다. 개인정보, 인증 정보 등이 이와 같은 영속적인 쿠키(Persistent Cookie)에 저장된다면, 공격자는 쿠키에 접근할 수 있는 보다 많은 기회를 가지게 되며, 이는 시스템을 취약하게 만든다.

나. 안전한 코딩기법

쿠키의 만료시간은 세션 지속 시간을 고려하여 최소한으로 설정하고 영속적인 쿠키에는 사용자 권한 등급, 세션 ID 등 중요 정보가 포함되지 않도록 한다.

다. 코드예제

다음과 같이 쿠키의 만료시간을 과도하게 길게 설정하면 사용자 하드 디스크에 저장된 쿠키가 도용될 수 있다.

안전하지 않은 코드 예시

```
1:  const express = require('express');
2:
3:  router.get("/vuln", (req, res) => {
4:    // 쿠키의 만료 시간을 1년으로 과도하게 길게 설정하고 있어 안전하지 않다
5:    res.cookie('rememberme', '1', {
6:      expires: new Date(Date.now() + 365*24*60*60*1000)
7:    });
8:    return res.send("쿠키 발급 완료");
9:  });
```

만료 시간은 해당 기능에 맞춰 최소로 설정하고 영속적인 쿠키에는 중요 정보가 포함되지 않도록 한다. HTTPS를 통해서만 쿠키를 전송하도록 secure 속성값을 True(기본값은 False)로 사용할 수 있다. 클라이언트 측에서 자바스크립트를 통해 쿠키를 접근하지 못하도록 제한하고자 할 경우엔 httpOnly 속성을 True(기본값은 False)로 설정한다. 다음은 쿠키 만료 시간을 1시간으로 설정한 예시다.

안전한 코드 예시

```
1:  const express = require('express');
2:
3:  router.get("/patched", (req, res) => {
4:    // 쿠키의 만료 시간을 적절하게 부여하고 secure 옵션을 활성화
5:    res.cookie('rememberme', '1', {
6:      expires: new Date(Date.now() + 60*60*1000),
7:      secure: true,
8:      httpOnly: true
9:    });
10:   return res.send("쿠키 발급 완료");
11: });
```

라. 참고자료

① CWE-539: Use of Persistent Cookies Containing Sensitive Information, MITRE, https://cwe.mitre.org/data/definitions/539.html

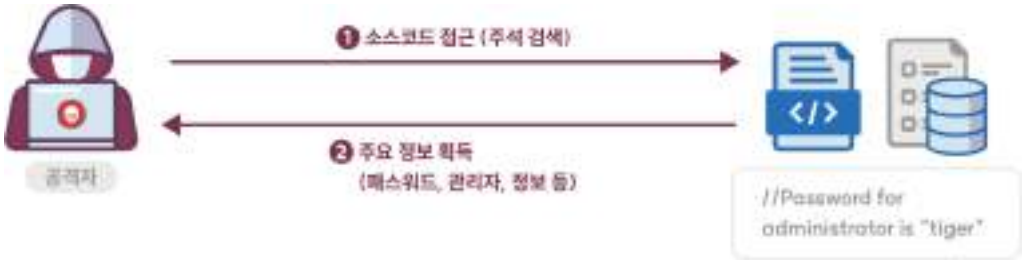
② Expire and Max-Age Attributes, OWASP, https://cheatsheetseries.owasp.org/cheatsheets/Session_Management_Cheat_Sheet.html#expire-and-max-age-attributes

③ Express - StringLoop, IBM, https://devdocs.io/express/index#res.cookie

13. 주석문 안에 포함된 시스템 주요정보

VanillaJS ✓ ReactJS ✓ ExpressJS ✓

가. 개요



소프트웨어 개발자가 편의를 위해서 주석문에 패스워드를 적어둔 경우, 소프트웨어가 완성된 후에는 그것을 제거하는 것이 매우 어렵게 된다. 만약 공격자가 소스코드에 접근할 수 있다면 시스템에 손쉽게 침입할 수 있다.

나. 안전한 코딩기법

주석에는 아이디, 패스워드 등 보안과 관련된 내용을 기입하지 않는다.

다. 코드예제

편리성을 위해 아이디, 패스워드 등 중요정보를 주석문 안에 작성 후 지우지 않는 경우 정보 노출 보안약점이 발생한다.

안전하지 않은 코드 예시

```
1:  const express = require('express');
2:
3:  router.post("/vuln", (req, res) => {
4:    // 주석문에 포함된 중요 시스템의 인증 정보
5:    // id = admin
6:    // password = 1234
7:    const result = login(req.body.id, req.body.password);
8:    return res.send(result);
9:  });
```

프로그램 개발 시에 주석문 등에 남겨놓은 사용자 계정이나 패스워드 등의 정보는 개발 완료 후 확실하게 삭제해야 한다.

안전한 코드 예시

```
1:  const express = require('express');
2:
3:  router.post("/vuln", (req, res) => {
4:    // 주석문에 포함된 민감한 정보는 삭제
5:    const result = login(req.body.id, req.body.password);
6:    return res.send(result);
7:  });
```

라. 참고자료

① CWE-615: Inclusion of Sensitive Information in Source Code Comments, MITRE, https://cwe.mitre.org/data/definitions/615.html

14. 솔트 없이 일방향 해쉬 함수 사용 VanillaJS - ReactJS - ExpressJS ✓

가. 개요

패스워드와 같이 중요정보를 저장할 경우, 가변 길이 데이터를 고정된 크기의 해쉬값으로 변환해주는 일방향 해쉬 함수를 이용해 저장할 수 있다. 만약 중요정보를 솔트(Salt)없이 일방향 해쉬 함수를 사용해 저장한다면, 공격자는 미리 계산된 레인보우 테이블을 이용해 해쉬값을 알아낼 수 있다.

나. 안전한 코딩기법

패스워드와 같은 중요 정보를 저장할 경우, 임의의 길이인 데이터를 고정된 크기의 해쉬값으로 변환해주는 일방향 해쉬 함수를 이용하여 저장한다. 또한, 솔트값은 사용자별로 유일하게 생성해야 하며, 이를 위해 사용자별 솔트 값을 별도로 저장하는 과정이 필요하다.

자바스크립트 NodeJS에서는 crypto 패키지를 사용해 해쉬값 생성 및 솔트(randomBytes)를 생성할 수 있다.

다. 코드예제

다음은 솔트 없이 길이가 짧은 패스워드를 해쉬 함수에 전달해 원문이 공격자에 의해 쉽게 유추되는 예시를 보여 준다.

안전하지 않은 코드 예시

```
1:  const crypto = require("crypto");
2:
3:  function getHashFromPwd(pw) {
4:    // salt가 없이 생성된 해쉬값은 강도가 약해 취약
5:    const hash = crypto.createHash('sha256').update(pw).digest('hex');
6:    return hash;
7:  }
```

짧은 길이의 패스워드로 강도 높은 해쉬값을 생성하기 위해서는 반드시 솔트 값을 함께 전달해야 한다.

안전한 코드 예시

```
1:  const crypto = require("crypto");
2:
3:  function getHashFromPwd(pw) {
4:    // 솔트 값을 사용하면 길이가 짧은 패스워드로도 고강도의 해쉬를 생성할 수 있음
5:    // 솔트 값은 사용자별로 유일하게 생성해야 하며, 패스워드와 함께 DB에 저장해야 함
6:    const salt = crypto.randomBytes(16).toString('hex');
7:    const hash = crypto.createHash('sha256').update(pw + salt).digest('hex');
8:
9:    return { hash, salt };
10: }
```

라. 참고자료

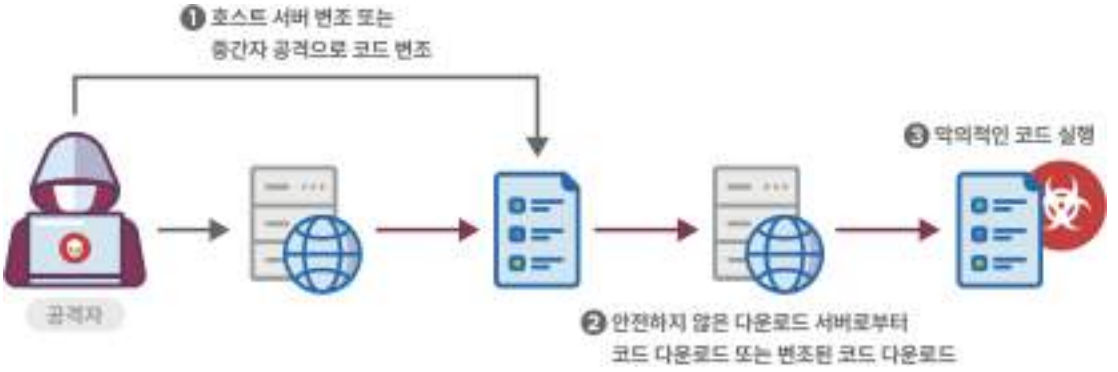
① CWE-759: Use of a One-Way Hash without a Salt, MITRE,
https://cwe.mitre.org/data/definitions/759.html

② Password Storage Cheat Sheet - Salting, OWASP,
https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html#salting

③ createHash, NodeJS,
https://nodejs.org/api/crypto.html#cryptocreatehashalgorithm-options

15. 무결성 검사없는 코드 다운로드 VanillaJS - ReactJS - ExpressJS ✓

가. 개요



원격지에 위치한 소스코드 또는 실행 파일을 무결성 검사 없이 다운로드 후 이를 실행하는 프로그램을 종종 발견할 수 있다. 이러한 프로그램은 호스트 서버의 변조, DNS 스푸핑(Spoofing) 또는 전송 시의 코드 변조 등의 방법을 이용해 공격자가 악의적인 코드를 실행하는 위협에 노출시킬 수 있다.

파일(및 해당 소프트웨어) 무결성을 확인하는 두 가지 주요 방법으로는 암호화 해쉬 및 디지털 서명이 있다. 무결성을 보장하기 위해 해쉬를 사용하고 가능하면 적절한 코드 서명 인증서를 사용하고 확인하는 것이 더 안전하다.

나. 안전한 코딩기법

DNS 스푸핑(Spoofing)을 방어할 수 있는 DNS lookup을 수행하고 코드 전송 시 신뢰할 수 있는 암호 기법을 이용해 코드를 암호화한다. 또한, 다운로드한 코드는 작업 수행을 위해 필요한 최소한의 권한으로 실행하도록 한다.

소스코드는 신뢰할 수 있는 사이트에서만 다운로드해야 하고 파일의 인증서 또는 해쉬값을 검사해 변조되지 않은 파일인지 확인하여야 한다.

다. 코드예제

다음 예제는 http.get을 통해 원격에서 파일을 다운로드한 뒤 파일에 대한 무결성 검사를 수행하지 않아 파일 변조 등으로 인한 피해가 발생하는 사례를 보여 준다. 이 경우 공격자가 악의적인 코드를 실행할 수 있다.

안전하지 않은 코드 예시

```
1:  const express = require('express');
2:  const fs = require("fs");
3:  const http = require("http");
4:
5:  router.get("/vuln", (req, res) => {
6:    // 신뢰할 수 없는 사이트에서 코드를 다운로드
7:    const url = "https://www.somewhere.com/storage/code.js";
8:
9:    // 원격 코드 다운로드
10:   http.get(url, (res) => {
11:     const path = "./temp/sample1.js"
12:     const writeStream = fs.createWriteStream(path);
13:
14:     res.pipe(writeStream);
15:     writeStream.on("finish", () => {
16:       writeStream.close();
17:     });
18:   });
19:
20:   // 무결성 검증 없이 파일 사용
21:   fs.readFile("./temp/sample1.js", "utf8", function (err, buf) {
22:     res.end(buf);
23:   })
24: });
```

안전한 코드 실행을 위해 다운로드한 파일로 계산한 해쉬값과 파일과 함께 전송된(또는 이미 저장된) 해쉬값 비교를 통한 무결성 검사를 거친 후 코드를 실행해야 한다.

안전한 코드 예시

```
1:  const express = require('express');
2:  const fs = require("fs");
3:  const http = require("http");
4:  const crypto = require("crypto");
5:
6:  router.get("/patched", async (req, res) => {
7:    const url = "https://www.somewhere.com/storage/code.js";
8:    const codeHash = req.body.codeHash;
9:
```

```
10:     http.get(url, (res) => {
11:       const path = "./temp/sample1.js"
12:       const writeStream = fs.createWriteStream(path);
13:       res.pipe(writeStream);
14:       writeStream.on("finish", () => {
15:         writeStream.close();
16:       });
17:     });
18:
19:     const hash = crypto.createHash('sha256');
20:     const input = fs.createReadStream("./temp/sample1.js");
21:
22:     let promise = new Promise ((resolve, reject) => {
23:       input.on("readable", () => {
24:         const data = input.read();
25:           if (data) {   hash.update(data); }
26:           else {   resolve();   }
27:        })
28:     });
29:
30:     await promise;
31:     const fileHash = hash.digest('hex');
32:
33:     # 무결성 검증에 통과할 경우에만 파일 사용
34:     if (fileHash === codeHash) {
35:       fs.readFile("./temp/sample1.js", "utf8", function (err, buf) {
36:         res.end(buf);
37:       })
38:     } else {
39:       return res.send("파일이 손상되었습니다.")
40:     }
41:   });
```

라. 참고자료

① CWE-494: Download of Code Without Integrity Check, MITRE, https://cwe.mitre.org/data/definitions/494.html

② Top 25 Series - Download of Code Without Integrity Check, SANS, https://www.sans.org/blog/top-25-series-rank-20-download-of-code-without-integrity-check/

16. 반복된 인증시도 제한 기능 부재

VanillaJS ✓ ReactJS ✓ ExpressJS ✓

가. 개요

일정 시간 내에 여러 번의 인증 시도 시 계정 잠금 또는 추가 인증 방법 등의 충분한 조치가 수행되지 않는 경우, 공격자는 인증에 성공할 가능성이 높은 계정과 패스워드들을 사전(Dictionary)으로 만들고 무차별 대입(brute-force)하여 로그인 성공 및 권한 획득이 가능하다.

악의적인 사용자의 반복된 인증 시도 자체를 클라이언트측에서 단독으로 제한하거나, 서버와의 연계를 통해 어느 정도 차단 효과를 기대할 수 있지만, 궁극적으로는 안전한 서버측 코드가 구현되어야만 무작위 대입 공격에 대응할 수 있다.

나. 안전한 코딩기법

최대 인증시도 횟수를 적절한 횟수로 제한하고 설정된 인증 실패 횟수를 초과할 경우 계정을 잠금 하거나 추가적인 인증 과정을 거쳐서 시스템에 접근이 가능하도록 한다. 코드 상에서 인증 시도 횟수를 제한하는 방법 외에 CAPTCHA나 Two-Factor 인증 방법도 설계 시부터 고려해야 한다.

다. 코드예제

다음 예제는 사용자 로그인 시도에 대한 횟수를 제한하지 않는 코드를 보여 준다.

안전하지 않은 코드 예시

```
1:  const express = require('express');
2:  const crypto = require('crypto');
3:
4:  router.post("/vuln", (req, res) => {
5:    const id = req.body.id;
6:    const password = req.body.password;
7:
8:    const hashPassword = crypto.createHash("sha512").update(password).digest("base64");
9:    const currentHashPassword = getUserPasswordFromDB(id);
10:
11:   // 인증 시도에 따른 제한이 없어 반복적인 인증 시도가 가능
12:   if (hashPassword === currentHashPassword) {
13:     return res.send("login success");
14:   } else {
15:     return res.send("login fail")
16:   }
17: });
```

다음은 사용자 로그인 시도에 대한 횟수를 제한하여 무차별 공격에 대응하는 방법을 보여 준다.

안전한 코드 예시

```
1:  const express = require('express');
2:  const crypto = require('crypto');
3:
4:  const LOGIN_TRY_LIMIT = 5;
5:
6:  router.post("/patched", (req, res) => {
7:    const id = req.body.id;
8:    const password = req.body.password;
9:    // 로그인 실패기록 가져오기
10:   const loginFailCount = getUserLoginFailCount(id);
11:
12:   // 로그인 실패횟수 초과로 인해 잠금된 계정에 대한 인증 시도 제한
13:   if (loginFailCount >= LOGIN_TRY_LIMIT) {
14:     return res.send("account lock(too many failed)")
15:   }
16:
```

```
17:     // 해시 생성시 솔트를 사용하는 것이 안전하나, 코드의 복잡성을 피하기 위해 생략
18:     const hashPassword = crypto.createHash("sha512").update(password).digest("base64");
19:     const currentHashPassword = getUserPasswordFromDB(id);
20:
21:     if (hashPassword === currentHashPassword) {
22:       deleteUserLoginFailCount(id);
23:       return res.send("login success");
24:     } else {
25:       updateUserLoginFailCount(id);
26:       return res.send("login fail")
27:     }
28:   });
```

라. 참고자료

① CWE-307: Improper Restriction of Excessive Authentication Attempts, MITRE, https://cwe.mitre.org/data/definitions/307.html

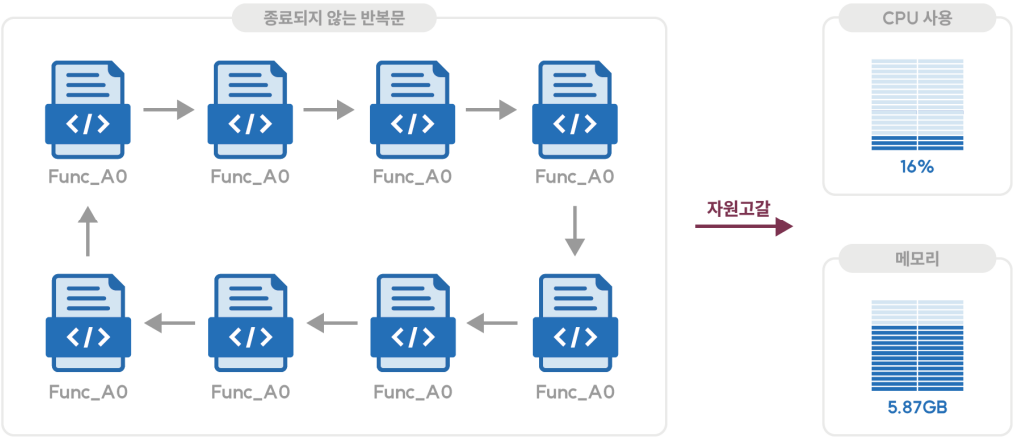
② Blocking Brute Force Attacks, OWASP, https://owasp.org/www-community/controls/Blocking_Brute_Force_Attacks

제3절 시간 및 상태 VanillaJS ✓ ReactJS ✓ ExpressJS ✓

동시 또는 거의 동시에 여러 코드 수행을 지원하는 병렬 시스템이나 하나 이상의 프로세스가 동작되는 환경에서 시간 및 상태를 부적절하게 관리하여 발생할 수 있는 보안약점이다.

1. 종료되지 않는 반복문 또는 재귀 함수

가. 개요



재귀 함수의 순환 횟수를 제어하지 못해 할당된 메모리나 프로그램 스택 등의 자원을 개발자가 의도한 범위를 과도하게 초과해 사용하면 위험하다. 대부분의 경우, 기본 케이스(Base Case[4])가 정의되어 있지 않은 재귀 함수는 무한 루프에 빠져들게 되고 자원고갈을 유발함으로써 시스템의 정상적인 서비스를 제공할 수 없게 한다.

재귀함수가 과도하게 누적되면 함수 반환을 위한 콜스택이 가득 차게 되고 이로 인해 정상적으로 동작되어야 할 코드까지 영향을 미칠 수 있다. 자바스크립트에서는 과도한 재귀함수 호출 시 'too much recursion(파이어폭스)', 'Maximum call stack size exceeded(크롬, 사파리)' 예외를 발생시키며 프로그램이 비정상 종료된다.

나. 안전한 코딩기법

모든 재귀 호출 시 호출 횟수를 제한하거나 재귀 함수 종료 조건을 명확히 정의해 호출을 제어해야 한다.

4) 기본 케이스(Base Case)는 재귀 호출을 하지 않고 반환하는 방법을 의미 한다

다. 코드예제

다음 코드 예시의 factorial 함수는 함수 내부에서 자신을 호출하는 함수로 재귀문을 빠져 나오는 조건을 정의하고 있지 않아 시스템 장애를 유발할 수 있다.

안전하지 않은 코드 예시

```
1:  function factorial(x) {
2:    # 재귀함수 탈출조건을 설정하지 않아 동작 중 에러 발생
3:    return x * factorial(x - 1);
4:  });
```

특정 조건 또는 횟수에 따라 재귀 코드 실행을 중단해 프로그램이 무한 반복에 빠지지 않도록 한다.

안전한 코드 예시

```
1:  function factorial(x) {
2:    # 재귀함수 사용 시에는 탈출 조건을 명시해야 한다
3:    if ( x === 0 ) {
4:      return;
5:    }
6:    else {
7:        return x * factorial(x - 1);
8:    }
9:  }
```

라. 참고자료

① CWE-674: Uncontrolled Recursion, MITRE,
https://cwe.mitre.org/data/definitions/674.html

② CWE-835: Loop with Unreachable Exit Condition ('Infinite Loop'), MITRE,
https://cwe.mitre.org/data/definitions/835.html

③ InternalError: moo much recursion, MDN contributors
https://devdocs.io/javascript/errors/too_much_recursion

제4절 에러처리

에러를 처리하지 않거나, 불충분하게 처리하여 에러 정보에 중요정보(시스템 내부정보 등)가 포함될 때, 발생할 수 있는 보안약점이다.

1. 오류 메시지 정보노출 VanillaJS - ReactJS ✓ ExpressJS ✓

가. 개요

응용 프로그램이 실행환경, 사용자 등 관련 데이터에 대한 민감한 정보를 포함하는 오류 메시지를 생성해 외부에 제공하는 경우, 공격자의 악성 행위로 이어질 수 있다. 예외발생 시 예외 이름이나 추적 메시지(traceback)를 출력하는 경우, 프로그램 내부 구조를 쉽게 파악할 수 있기 때문이다.

ExpressJS에서는 HTTP 오류 코드가 있는 요청을 처리하기 위한 errorHandler 미들웨어를 제공한다.

나. 안전한 코딩기법

오류 메시지는 정해진 사용자에게 유용한 최소한의 정보만 포함하도록 한다. 코드 내에서 예외 상황은 내부적으로 처리하고 사용자에게 시스템 내부 정보 등 민감한 정보를 포함하는 오류를 출력하지 않고 미리 정의된 메시지를 제공하도록 설정한다.

Express 프레임워크에서는 미들웨어 방식으로 에러 페이지 핸들러를 정의할 수 있다.

다. 코드예제

사용자 요청을 정상적으로 처리할 수 없는 경우 에러 페이지에 디버그 정보 또는 서버의 정보가 노출될 수 있다. 발생한 에러 내용을 그대로 반환해선 안 되며, 내부적으로 정의된 기준에 따라 최소한의 정보만 반환해야 한다.

안전하지 않은 코드 예시

```
1:  const express = require('express');
2:  const fs = require("fs");
3:
4:  router.get("/vuln", (req, res) => {
5:    const filePath = "./file/secret/password";
6:    fs.readFile(filePath, (err, data) => {
7:      if (err) {
8:        // 서버 내부에서 발생한 에러 메시지를 그대로 사용자에 전달해 주요 정보 노출 위험
9:        return res.status(500).send(err);
10:     } else {
11:       return res.send(data);
12:     }
13:   })
14: });
```

서버 내부에서 에러 발생 시 그 정보를 그대로 클라이언트에 전달하지 않고, 자체적인 기준에 따라 제한된 정보만 클라이언트에 반환해야 한다.

안전한 코드 예시

```
1:  const express = require('express');
2:  const fs = require("fs");
3:
4:  router.get("/vuln", (req, res) => {
5:    const filePath = "./file/secret/password";
6:    fs.readFile(filePath, (err, data) => {
7:      if (err) {
8:        // 에러 내용을 그대로 전달하지 않고 필터링 처리
9:        return res.status(500).send({ message: "잘못된 요청입니다" });
10:     } else {
11:       return res.send(data);
12:     }
13:   })
14: });
```

Express에서 제공하는 errorHandler 미들웨어를 정의해 에러 발생 유형별로 미리 정의된 규칙에 맞게 에러 메시지를 일관되게 전달할 수 있다.

안전한 코드 예시

```
1:  const express = require("express");
2:  const app  = express();
3:
4:  // errorHandler 미들웨어는 다른 app.use() 및 라우터 정의 후 가장 마지막에 정의해야 하며,
5:  // errorHandler 미들웨어를 커스텀하여 에러별 에러페이지를 반환할 수 있다
6:
7:  app.use(clientErrorHandler);
8:  app.use(errorHandler);
9:
10: function clientErrorHandler (err, req, res, next) {
11:     if (req.xhr) {
12:       res.status(500).send({ error: 'Something failed!' });
13:     } else {
14:       next(err);
15:     }
16: }
17:
18: function errorHandler (err, req, res, next) {
19:     if (err.name = "ValidError") {
20:         res.status(400)
21:         res.render('400error', { error: err })
22:     } else if (err.name = "AuthError") {
23:         res.status(401)
24:         res.render('401error', { error: err })
25:     } else if (err.name = "ForbiddenError") {
26:         res.status(403)
27:         res.render('403error', { error: err })
28:     } else if (err.name = "DBError") {
29:         res.status(500)
30:         res.render('500error', { error: err })
31:     }
32: }
```

라. 참고자료

① CWE-209: Generation of Error Message Containing Sensitive Information, MITRE
https://cwe.mitre.org/data/definitions/209.html

② Improper Error Handling, OWASP,
https://owasp.org/www-community/Improper_Error_Handling

③ Error, MDN contributors
https://devdocs.io/javascript/global_objects/error

④ Error Handling in Express, StrongLoop, IBM,
https://devdocs.io/express/guide/error-handling

2. 오류상황 대응 부재 VanillaJS ✓ ReactJS ✓ ExpressJS ✓

가. 개요

오류가 발생할 수 있는 부분을 확인하였으나, 이러한 오류에 대해 예외 처리를 하지 않을 경우, 공격자는 오류 상황을 악용해 개발자가 의도하지 않은 방향으로 프로그램이 동작하도록 할 수 있다.

예외처리는 코드를 견고하게 만들고 프로그램 제어 실패로 인해 의도치 않은 중단으로 이어지는 잠재적인 오류를 방지하는데 도움이 된다.

나. 안전한 코딩기법

오류가 발생할 수 있는 부분에 대하여 제어문(try-catch)을 사용해 적절하게 예외 처리 한다.

다. 코드예제

다음 예제는 try 블록에서 발생하는 오류를 포착(catch)하고 있지만, 그 오류에 대해서 아무 조치를 하지 않는 상황을 보여준다. 아무 조치가 없으므로 프로그램이 계속 실행되기 때문에 개발자가 의도하지 않은 방향으로 프로그램이 동작할 수 있다.

안전하지 않은 코드 예시

```
1:  const express = require('express');
2:  const crypto = require("crypto");
3:
4:   const staticKeys = [
5:       { "key" : "a6823ecf34012b1ca5ca7889f4eabb51", "iv" : "e79ce03b4563647a" },
6:       { "key" : "9e7c30e899f296a1daca7d9a0f92e71c", "iv" : "ab39941053fb5f6a" }
7:    ];
8:
9:  router.post("/vuln", (req, res) => {
10:    let statickKey = { "key" : "00000000000000000000000000000000", "iv" : "0000000000000000" };
11:
12:    const inputText = req.body.text;
13:    const keyId = req.body.id;
14:
15:    try {
16:      staticKey.key = staticKeys[keyId].key;
17:      staticKey.iv = staticKeys[keyId].iv;
18:    } catch (err) {
19:      // key 선택 중 오류 발생 시 기본으로 설정된 암호화 키인
20:      // '0000000000000000' 으로 암호화가 수행됨
21:      console.log(err);
22:    }
23:
24:    const cipher = crypto.createCipheriv('aes-256-cbc', statickKey.key, statickKey.iv);
25:    const encryptedText = cipher.update(inputText, 'utf8', 'base64') + cipher.final('base64');
26:
27:    return res.send(encryptedText);
28: });
```

예외상황 발생 시에 프로그램이 개발자의 의도와 다르게 동작하지 않도록 반드시 예외 처리 구문을 추가해야 한다.

안전한 코드 예시

```
1:  const express = require('express');
2:  const crypto = require("crypto");
3:
4:   const staticKeys = [
5:       { "key" : "a6823ecf34012b1ca5ca7889f4eabb51", "iv" : "e79ce03b4563647a" },
6:       { "key" : "9e7c30e899f296a1daca7d9a0f92e71c", "iv" : "ab39941053fb5f6a" }
7:    ];
8:
9:  router.post("/vuln", (req, res) => {
10:    let statickKey = { "key" : "00000000000000000000000000000000", "iv" : "0000000000000000"
11: };
12:
13:    const inputText = req.body.text;
14:    const keyId = req.body.id;
15:
16:    try {
17:      staticKey.key = staticKeys[keyId].key;
18:      staticKey.iv = staticKeys[keyId].iv;
19:    } catch (err) {
20:      // 키 선택 중 오류 발생 시 랜덤으로 암호화 키를 생성하도록 설정
21:      staticKey.key = crypto.randomBytes(16).toString('hex');
22:      staticKey.iv = crypto.randomBytes(8).toString('hex');
23:    }
24:
25:    const cipher = crypto.createCipheriv('aes-256-cbc', statickKey.key, statickKey.iv);
26:    const encryptedText = cipher.update(inputText, 'utf8', 'base64') + cipher.final('base64');
27:
28:    return res.send(encryptedText);
29: });
```

라. 참고자료

① CWE-390: Detection of Error Condition Without Action, MITRE,
https://cwe.mitre.org/data/definitions/390.html

② The JavaScript Guide[5)]
https://github.com/Checkmarx/JS-SCP/blob/master/dist/js-webapp-scp.pdf

5) 해당 자료는 Creative Commons Attribution Share Alike 4.0 International 저작권을 따르며, 저작권 원본 주소는 다음과 같음:
https://github.com/Checkmarx/JS-SCP/blob/master/LICENSE

3. 부적절한 예외 처리 VanillaJS ✓ ReactJS ✓ ExpressJS ✓

가. 개요

프로그램 수행 중에 함수의 결과 값에 대한 적절한 처리 또는 예외 상황에 대한 조건을 적절하게 검사 하지 않을 경우, 예기치 않은 문제를 야기할 수 있다.

나. 안전한 코딩기법

값을 반환하는 모든 함수의 결과값을 검사해야 한다. 결과값이 개발자가 의도했던 값인지 검사하고, 예외 처리를 사용하는 경우에 광범위한 예외 처리 대신 구체적인 예외 처리를 수행한다.

다. 코드예제

다음 예제는 다양한 예외가 발생할 수 있음에도 불구하고 광범위한 예외 처리로 예외상황에 따른 적절한 조치가 부적절한 사례를 보여 준다.

안전하지 않은 코드 예시

```
1:  const express = require('express');
2:
3:  router.post("/vuln", (req, res) => {
4:    const password = req.body.password;
5:    const user = {
6:      user: "user",
7:      password: password,
8:    };
9:    function getUser() {
10:     return new Promise((resolve, reject) => {
11:       resolve(user);
12:     });
13:   }
14:   function getUserPassword(user) {
15:     return new Promise((resolve, reject) => {
16:       if (user.password === undefined) {
17:         reject(new Error("login fail"));
18:       } else {
19:         resolve(user);
20:       }
21:     });
22:   }
23:
24:
```

```
25:     function checkPassword(user) {
26:       return new Promise((resolve, reject) => {
27:         if (user.password == "abc") {
28:           resolve(user);
29:         } else {
30:           reject(new Error("login fail"));
31:         }
32:       });
33:     }
34:     return getUser()
35:       .then(getUserPassword)
36:       .then(checkPassword)
37:       .then((result) => res.send(result))
38:       .catch((err) => {
39:         // 서로 다른 예외 상황에서도 동일한 에러 메시지만 출력
40:         res.send(err.message);
41:       });
42: });
```

다음은 발생 가능한 예외를 세분화한 후 예외상황에 따라 적합한 처리 예시를 보여 준다.

안전한 코드 예시

```
1:   const express = require('express');
2:
3:  router.post("/patched", (req, res) => {
4:     const password = req.body.password;
5:     const user = {
6:       user: "user",
7:       password: password,
8:     };
9:     function getUser() {
10:      return new Promise((resolve, reject) => {
11:        resolve(user);
12:      });
13:    }
14:    function getUserPassword(user) {
15:      return new Promise((resolve, reject) => {
16:        if (user.password === undefined) {
17:          reject(new Error("password required"));
18:        }
19:        resolve(user);
20:      });
21:    }
```

```
22:     function checkPassword(user) {
23:       return new Promise((resolve, reject) => {
24:         if (user.password == "abc") {
25:           resolve(user);
26:         } else {
27:           reject(new Error("wrong password"));
28:         }
29:       });
30:     }
31:     return getUser()
32:       .then(getUserPassword)
33:       .then(checkPassword)
34:       .then((result) => res.send(result))
35:       .catch((err) => {
36:         // 예외 상황에 맞는 처리 결과 메시지를 로그에 기록(내장 함수 log가 있다고 가정)
37:         log(err.messsage);
38:         // 클라이언트에는 에러 발생 여부만 전달
39:         res.send('login failed');
40:       });
41:   );
```

라. 참고자료

① CWE-754: Improper Check for Unusual or Exceptional Conditions, MITRE, https://cwe.mitre.org/data/definitions/754.html

② The JavaScript Guide https://github.com/Checkmarx/JS-SCP/blob/master/dist/js-webapp-scp.pdf

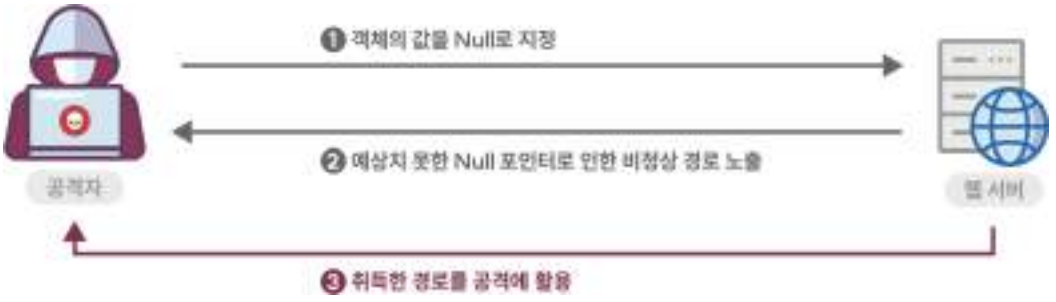
제5절 코드오류

타입 변환 오류, 자원(메모리 등)의 부적절한 반환 등과 같이 개발자가 범할 수 있는 코딩 오류로 인해 유발되는 보안약점이다.

1. Null Pointer 역참조

VanillaJS ✓ ReactJS ✓ ExpressJS ✓

가. 개요



널 포인터(Null Pointer) 역참조는 '일반적으로 그 객체가 널(Null)이 될 수 없다'라고 하는 가정을 위반했을 때 발생한다. 공격자가 의도적으로 널 포인터 역참조를 발생시키는 경우, 공격자는 그 결과로 발생하는 예외 상황을 이용해 추후 공격 계획에 활용할 수 있다.

자바스크립트에서는 Null pointer dereference가 발생하지 않는다. 자바스크립트에서는 Null 객체가 사용되지 않으며 정의되지 않은 변수는 undefined로, null 개체는 null 값을 가지며 '정의되지 않음'과 '없음'을 다르게 처리한다.

나. 안전한 코딩기법

변수의 값을 검사할 때 undefined 상태와 null 상태를 구분해서 처리하지 않으면 기대하지 않은 결과값이 나올 수 있다. undefined 또는 null이 될 수 있는 데이터를 참조하기 전에 해당 데이터의 값을 정확히 검사하여 시스템 오류를 줄일 수 있다.

다. 코드예제

자바스크립트에서는 포인터를 사용하지는 않지만 데이터에 대한 적절한 검사를 수행하지 않을 경우 Null pointer와 유사한 None 값 참조 오류를 범할 수 있다.

안전하지 않은 코드 예시

```
1:  const express = require('express');
2:
3:  router.get("/vuln", (req, res) => {
4:    const id = req.query.id;
5:    // 사용자가 id 값을 전달하지 않을 경우(undefined) 에러 발생
6:    return res.send("length = " + id.length);
7:  });
```

참조하고자 하는 자원을 호출 시에는 반드시 개체가 undefined 또는 null이 아닌지 검증해야 한다.

안전한 코드 예시

```
1:  const express = require('express');
2:
3:  router.get("/patched", (req, res) => {
4:    const id = req.query.id;
5:    if (id === undefined) {
6:      // id 값이 전달되지 않을 경우 사용자에게 요구하는 예외 코드 추가
7:      return res.send("id is required");
8:    } else {
9:      return res.send("length = " + id.length);
10:   }
11: });
```

라. 참고자료

① CWE-476: NULL Pointer Dereference, MITRE,
https://cwe.mitre.org/data/definitions/476.html

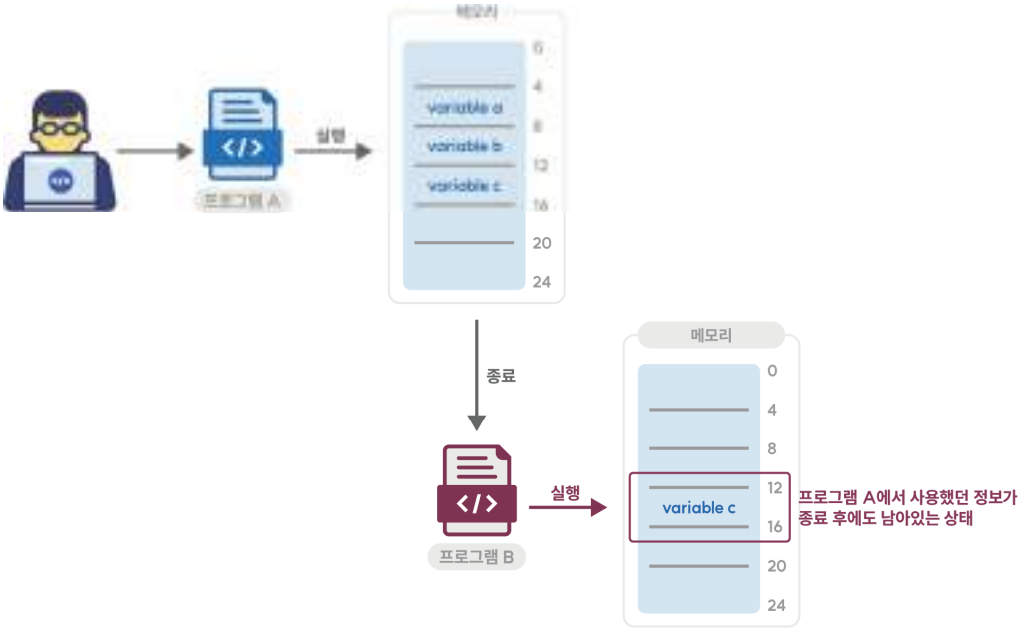
② Null Dereference, OWASP,
https://owasp.org/www-community/vulnerabilities/Null_Dereference

③ undefined, MDN contributors,
https://devdocs.io/javascript/global_objects/undefined

④ Properties of variables with "null" or "undefined" values should not be accessed, Sona Rules,
https://rules.sonarsource.com/javascript/RSPEC-2259

2. 부적절한 자원 해제 VanillaJS ✓ ReactJS ✓ ExpressJS ✓

가. 개요



프로그램의 자원, 예를 들면 열려 있는 파일 식별자(Open File Descriptor), 힙 메모리(Heap Memory), 소켓(Socket) 등은 유한한 자원이다. 이러한 자원을 할당 받아 사용을 마치고 더 이상 사용하지 않는 경우에는 적절히 반환해야 하는데, 프로그램 오류 또는 에러로 사용이 끝난 자원을 반환하지 못하는 경우에 문제가 발생할 수 있다.

나. 안전한 코딩기법

자원을 획득하여 사용한 다음에는 반드시 자원을 해제 후 반환한다.

다. 코드예제

다음은 try 구문 내의 코드 실행 중 오류가 발생할 경우, close() 메소드가 실행되지 않아 사용한 자원이 반환되지 않는 경우를 보여 준다.

안전하지 않은 코드 예시

```
1:  const express = require('express');
2:  const fs = require('fs');
3:
4:  router.get("/vuln", (req, res) => {
5:    const configPath = './config.cfg';
6:    let fid = null;
7:    let fdata = null;
8:
9:    try {
10:     fid = fs.openSync(configPath, 'r');
11:     fdata = fs.readFileSync(fid, 'utf8');
12:     // 예외 발생 상황 가정: 유효하지 않은 파일 디스크립터로 파일 읽기 시도
13:     fdata = fs.readFileSync('100', 'utf8');
14:
15:     // try 절에서 할당된 자원이 반환(close)되기 전에 예외가 발생하면
16:     // 할당된 자원이 시스템에 바로 반환되지 않음
17:     fs.close(fid, err => {
18:       if (err) {
19:         console.log('error occured while file closing');
20:       } else {
21:         console.log('file closed');
22:       }
23:     });
24:   } catch (e) {
25:     console.log('error occured!' , e);
26:   }
27:   return res.send(fdata);
28: });
```

예외 상황이 발생하여 함수가 종료될 때, 예외의 발생 여부와 상관없이 항상 실행되는 finally 블록에서 할당 받은 모든 자원을 반환해야 한다.

안전한 코드 예시

```
1:  const express = require('express');
2:  const fs = require('fs');
3:
4:  router.get("/patched", (req, res) => {
5:    const configPath = './5_code_error/40_improper_resource_shutdown/config.cfg';
6:    let fid = null;
7:    let fdata = null;
8:
9:    try {
10:     fid = fs.openSync(configPath, 'r');
11:     fdata = fs.readFileSync(fid, 'utf8');
12:     fdata = fs.readFileSync(100, 'utf8');
13:
14:   } catch (e) {
15:     console.log('error occured!' , e);
16:
17:   // try 절에서 할당된 자원은 finally 절에서 시스템에 반환을 해야 함
18:   } finally {
19:     fs.close(fid, err => {
20:       if (err) {
21:         console.log('error occured while flle closing');
22:       } else {
23:         console.log('file closed');
24:       }
25:     });
26:   }
27:   return res.send(fdata);
28: });
```

라. 참고자료

① CWE-404: Improper Resource Shutdown or Release, MITRE, https://cwe.mitre.org/data/definitions/404.html

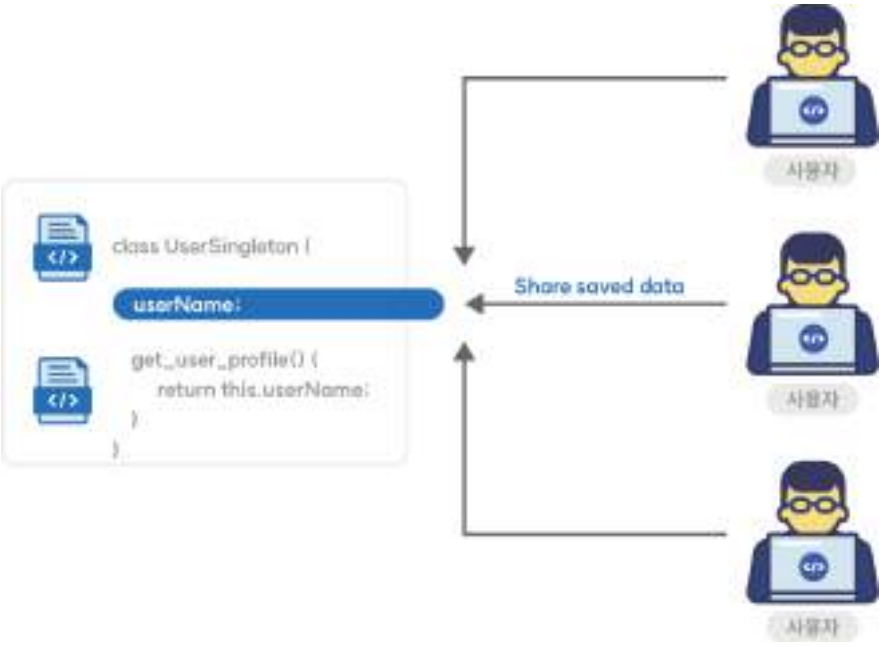
② Unreleased Resource, OWASP, https://owasp.org/www-community/vulnerabilities/Unreleased_Resource

제6절 캡슐화

중요한 데이터 또는 기능을 불충분하게 캡슐화하거나 잘못 사용함으로써 발생하는 보안약점으로 정보노출, 권한 문제 등이 발생할 수 있다.

1. 잘못된 세션에 의한 데이터 정보 노출 VanillaJS - ReactJS - ExpressJS ✓

가. 개요



다중 스레드 환경에서는 싱글톤(Singleton) 객체 필드에 경쟁조건(Race Condition)이 발생할 수 있다. 따라서, 다중 스레드 환경에서는 정보를 저장하는 전역 변수가 포함되지 않도록 코드를 작성해 서로 다른 세션에서 데이터를 공유하지 않도록 해야 한다.

나. 안전한 코딩기법

싱글톤 패턴을 사용하는 경우, 변수 범위(Scope)에 주의를 기울여야 한다. 특히 다중 스레드 환경에서 클래스 변수의 값은 하위 메소드와 공유되므로 필요한 경우 인스턴스 변수로 선언하여 사용한다.

다. 코드예제

서버 라우터 코드에서 전역 변수로 정의한 값을 공용으로 사용할 경우 서로 다른 세션 간에 데이터가 공유되어 의도하지 않은 데이터가 전달될 수 있다.

안전하지 않은 코드 예시

```
let instance;

class UserSingleton {
  constructor() {
    this.userName = 'testUser';
      if (instance) {
        return instance;
      }
    instance = this;
  }
  getUserProfile() {
    return this.userName;
  }
};

router.get("/vuln", (req, res) => {
    const user = new UserSingleton();
    const profile = user.getUserProfile();
    // 서로 다른 세션에서 공유되는 값을 사용하는 경우 다른 세션에 의해 데이터가 노출될 수 있음
    return res.send(profile);
});
```

공유가 금지된 변수는 싱글톤 패턴이 아닌 일반 클래스 안에서 선언하여 세션 간에 공유되지 않도록 한다.

안전한 코드 예시

```
1:  class User {
2:    constructor() {
3:      this.userName = 'testUser';
4:    }
5:    getUserProfile() {
6:      return this.userName;
7:    }
8:  };
9:
10: router.get("/patched", (req, res) => {
11:     const user = new User();
12:     const profile = user.getUserProfile();
13:     // 서로 다른 세션에서 공유되지 않는 클래스 정의를 사용해 안전함
14:     return res.send(profile);
15: });
```

라. 참고자료

① CWE-488: Exposure of Data Element to Wrong Session, MITRE, https://cwe.mitre.org/data/definitions/488.html

② CWE-543: Use of Singleton Pattern Without Synchronization in a Multithreaded Context, MITRE, https://cwe.mitre.org/data/definitions/543.html

2. 제거되지 않고 남은 디버그 코드 VanillaJS ✓ ReactJS ✓ ExpressJS ✓

가. 개요

디버깅 목적으로 삽입된 코드는 개발이 완료되면 제거해야 한다. 디버그 코드는 설정 등의 민감한 정보 또는 의도하지 않은 시스템 제어로 이어지는 정보를 담고 있을 수 있다. 만일 디버그 코드가 남겨진 채로 배포될 경우, 공격자가 식별 과정을 우회하거나 의도하지 않은 정보 노출 문제가 발생할 수 있다.

기본적으로, 자바스크립트에서는 console.log 또는 console.error 함수를 사용해 디버그 메시지를 출력한다. 개발 완료 후에 디버깅용으로 작성한 코드를 오버라이드(override) 해 애플리케이션 동작 시 콘솔 로그가 출력되지 않도록 한다. 또한, 기본적으로 ExpressJS는 디버그 모드가 비활성화 된 상태에서 동작하며, 서버 구동 시 'DEBUG=' prefix 설정을 하지 않는 한 서버 디버깅 메시지는 출력하지 않는다.

나. 안전한 코딩기법

소프트웨어 배포 전 반드시 디버그 코드를 확인 및 삭제한다. 코드 에디터에서 console.log 또는 console.error를 모두 삭제하거나, 서비스 배포 모드에서만 console.log가 동작하지 않도록 설정할 수 있다

다. 코드예제

가) ReactJS 예제

개발이 끝난 코드 배포 전 불필요한 디버그 코드를 모두 삭제해야 하며, 개발자의 실수로 인한 미삭제 로그 대응을 위해 아래와 같은 코드를 추가해 로그가 출력되지 않도록 한다.

안전한 코드 예시

```
1:  // 애플리케이션 진입 코드인 Index.js에 다음 코드를 추가하면
2:  // 코드 내에 삭제되지 않은 모든 console.log 코드로 인해 아무런 내용도 출력되지 않음
3:  if (process.env.NODE_ENV === 'production' && typeof window !== 'undefined') {
4:    console.log = () => {};
5:  }
```

나) ExpressJS 예제

마찬가지로, 개발이 끝난 코드 배포 전 불필요한 디버그 코드를 모두 삭제해야 하며, 개발자의 실수로 인한 미삭제 로그 대응을 위해 아래와 같은 코드를 추가해 로그가 출력되지 않도록 한다.

안전한 코드 예시

```
1:  // 서버 진입 코드인 index.js 내의 서버 구동 부분에 다음 코드 추가
2:  app.listen(80, () => {
3:    if (!process.env.DEBUG) {
4:      console.log = function(){}
5:    }
6:  });
```

라. 참고자료

① CWE-489: Active Debug Code, MITRE,
https://cwe.mitre.org/data/definitions/489.html

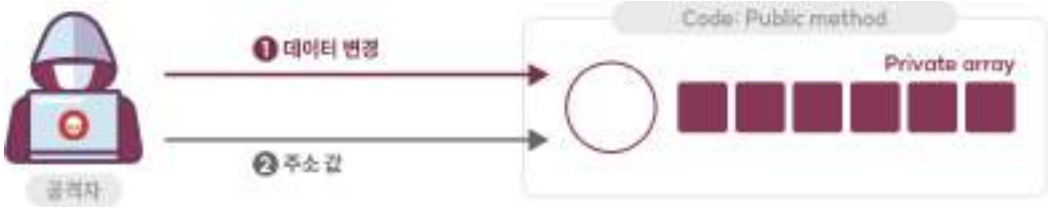
② Debugging Express,
https://expressjs.com/en/guide/debugging.html

③ Production best practices: performance and reliability,
https://expressjs.com/en/advanced/best-practice-performance.html

3. Public 메소드로부터 반환된 Private 배열

VanillaJS ✓ ReactJS ✓ ExpressJS ✓

가. 개요



자바스크립트는 ES 버전별로 클래스를 정의하는 방법에 조금씩 차이가 있다. ES6 이전에는 프로토타입을 클래스처럼 정의해서 사용했고, ES6부터 클래스 문법이 정식으로 도입되어 보다 명확한 구현이 가능해졌다.

초기 클래스의 경우 private 속성을 지원하지 않았으며, 언더스코어 프리픽스(_) 코드 컨벤션을 사용해 private 속성을 처리했다. 후에 '#' 기호를 프리픽스로 사용해 private 속성을 정의할 수 있게 되었다. public으로 선언된 메소드에서 배열을 반환하면, 해당 배열의 참조 객체가 외부에 공개되어 외부에서 배열 수정과 객체 속성 변경이 가능해진다. 이러한 속성은 배열뿐만 아니라 변경 가능한(mutable) 모든 객체에 해당된다.

나. 안전한 코딩기법

private로 선언된 배열을 public으로 선언된 메소드로 반환하지 않도록 한다. private 배열에 대한 복사본을 반환하도록 하고 배열의 원소에 대해서는 Object.assign() 메소드를 통해 복사된 원소를 저장하도록 해서 private 선언된 배열과 객체 속성에 대한 의도치 않은 수정을 방지한다.

다. 코드예제

다음 예제는 private 변수를 생성하고 이를 반환하는 public 메소드를 사용하는 예시를 보여 준다. 이 경우 외부에서 클래스 내에 숨겨져 있는 private 배열 값에 접근할 수 있는 문제점이 발생한다.

안전하지 않은 코드 예시

```
1:  class UserObj {
2:    #privArray = [];
3:
4:    // private 배열을 리턴하는 public 메소드를 사용하는 경우 취약
5:    get_private_member = () => {
6:      return this.#privArray;
7:    }
8:  }
```

아래 예제는 내부와 외부의 배열이 서로 참조되는 것을 예방하기 위해 새로운 객체를 생성하여 값을 반환하고 있다.

안전한 코드 예시

```
1:  class UserObj {
2:    #privArray = [];
3:
4:    // private 배열을 반환하는 경우 복사본을 사용해 외부와 내부의
5:    // 배열이 서로 참조되지 않도록 해야 함
6:    get_private_member = () => {
7:      const copied = Object.assign([], this.#privArray);
8:      return copied;
9:    }
10: }
```

라. 참고자료

① CWE-495: Private Data Structure Returned From A Public Method, MITRE, https://cwe.mitre.org/data/definitions/495.html

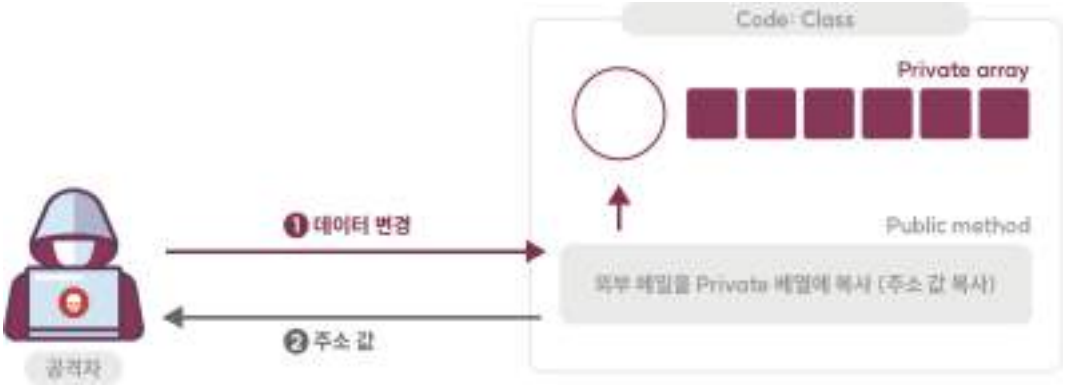
② Do not return references to private mutable class members, CERT, https://wiki.sei.cmu.edu/confluence/display/java/OBJ05-J.+Do+not+return+references+to+private+mutable+class+members

③ Private class fields, MDN web docs, https://developer.mozilla.org/ko/docs/Web/JavaScript/Reference/Classes/Private_class_fields

4. Private 배열에 Public 데이터 할당

VanillaJS ✓ ReactJS ✓ ExpressJS ✓

가. 개요



public으로 선언된 메소드의 인자가 private로 선언된 배열에 저장되면 private 배열을 외부에서 접근하여 배열 수정과 객체 속성 변경이 가능해진다.

나. 안전한 코딩기법

public으로 선언된 메소드의 인자를 private로 선언된 배열에 저장하지 않도록 한다. 사용자가 전달한 값으로 클래스 외부에서 private 값을 변경해서는 안 되며, 필요한 경우 별도의 인스턴스 변수로 정의하거나, 의도한 기능이라면 전달된 값의 정상 여부를 검증한 후 적용해야 한다.

다. 코드예제

다음 예제는 private 배열을 생성하고 외부값을 대입하는 public 메소드를 사용하는 예시를 보여 준다. 이 경우 특정 배열 타입에 따라 외부에서 private 배열을 변조할 수 있는 문제를 내포하고 있다.

안전하지 않은 코드 예시

```
1:
2:  class UserObj {
3:    #privArray = [];
4:
5:    // private 배열에 외부 값을 바로 대입하는 public 메소드를 사용하는 경우 취약
6:    set_private_member = (input_list) => {
7:      this.#privArray = input_list;
8:    }
9:  }
```

다음은 public 메서드의 인자를 private가 아닌 public 배열에 저장하는 안전한 예시를 보여 준다.

안전한 코드 예시

```
1:  class UserObj2 {
2:    #privArray = [];
3:
4:    // 사용자가 전달한 값을 private가 아닌 public 배열로 저장
5:    set_private_member = (input_list) => {
6:      this.userInput = input_list;
7:    }
8:  }
```

라. 참고자료

① CWE-496: Public Data Assigned to Private Array-Typed Field, MITRE, https://cwe.mitre.org/data/definitions/496.html

② Object.assign(), MDN web docs, https://developer.mozilla.org/ko/docs/Web/JavaScript/Reference/Global_Objects/Object/assign

③ Private class fields, MDN web docs, https://developer.mozilla.org/ko/docs/Web/JavaScript/Reference/Classes/Private_class_fields

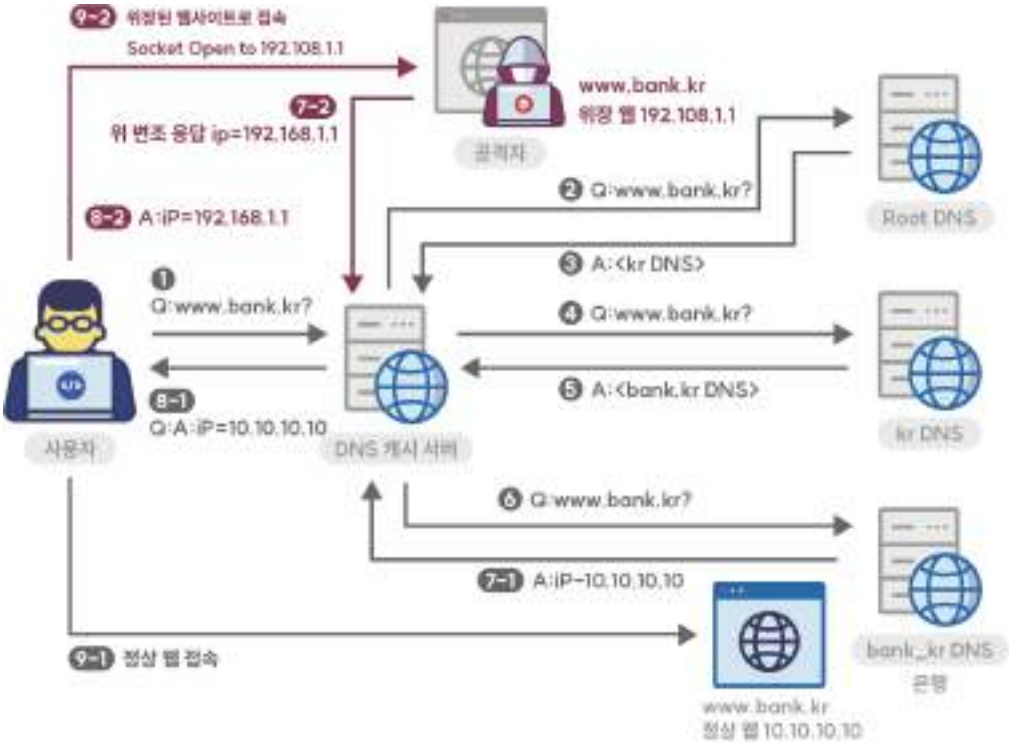
제7절 API 오용

의도된 사용에 반하는 방법으로 API를 사용하거나, 보안에 취약한 API를 사용하여 발생할 수 있는 보안약점이다.

1. DNS lookup에 의존한 보안결정

VanillaJS - ReactJS - ExpressJS ✓

가. 개요



공격자가 DNS 엔트리를 속일 수 있으므로 도메인명에 의존해서 보안결정(인증 및 접근 통제 등)을 하지 않아야 한다. 만약, 로컬 DNS 서버의 캐시가 공격자에 의해 오염된 상황이라면, 사용자와 특정 서버 간의 네트워크 트래픽이 공격자를 경유하도록 할 수도 있다. 또한, 공격자가 마치 동일 도메인에 속한 서버인 것처럼 위장할 수도 있다.

나. 안전한 코딩기법

보안결정에서 도메인명을 이용한 DNS lookup을 하지 않도록 한다.

다. 코드예제

다음의 예제는 도메인명을 통해 해당 요청을 신뢰할 수 있는지를 검사하는 예시로, 공격자는 DNS 캐쉬 등을 조작해서 쉽게 이러한 보안 설정을 우회할 수 있다.

안전하지 않은 코드 예시

```
1:  const express = require('express');
2:  const dns = require("dns");
3:
4:  router.get("/vuln", (req, res) => {
5:    let trusted = false;
6:
7:    const trustedHost = "www.google.com";
8:    const hostName = req.query.host;
9:
10:   // 공격자에 의해 실행되는 서버의 DNS가 변경될 수 있으므로
11:   // 안전하지 않음
12:   if (hostName === trustedHost) {
13:     trusted = true;
14:   }
15:   return res.send({trusted});
16: });
```

도메인명을 이용한 비교를 하지 말고 IP 주소를 직접 비교하도록 수정해 코드를 안전하게 만들 수 있다.

안전한 코드 예시

```
1:  const express = require('express');
2:  const dns = require("dns");
3:
4:  router.get("/patched", async (req, res) => {
5:    let trusted = false;
6:
7:    const trustedHost = "142.250.207.100";
8:    // 실제 서버의 IP 주소를 비교하여 DNS 변조에 대응
9:
```

```
10:     async function dnsLookup() {
11:       return new Promise((resolve, reject) => {
12:         dns.lookup(req.query.host, 4, (err, address, family) => {
13:           if (err) reject(err);
14:           resolve(address);
15:         })
16:       })
17:     }
18:
19:     const hostName = await dnsLookup();
20:
21:     if (hostName === trustedHost) {
22:       trusted = true;
23:     }
24:     return res.send({trusted});
25:   });
```

라. 참고자료

① CWE-350: Reliance on Reverse DNS Resolution for a Security-Critical Action, MITRE, https://cwe.mitre.org/data/definitions/350.html

3

PART 제3장

부 록

3 부록

제1절 구현단계 보안약점 제거 기준

1. 입력데이터 검증 및 표현

번호	보안약점	설명
1	SQL 삽입	SQL 질의문을 생성할 때 검증되지 않은 외부 입력값을 허용하여 악의적인 질의문이 실행가능한 보안약점
2	코드 삽입	프로세스가 외부 입력값을 코드(명령어)로 해석·실행할 수 있고 프로세스에 검증되지 않은 외부 입력값을 허용한 경우 악의적인 코드가 실행 가능한 보안약점
3	경로 조작 및 자원 삽입	시스템 자원 접근경로 또는 자원제어 명령어에 검증되지 않은 외부 입력값을 허용하여 시스템 자원에 무단 접근 및 악의적인 행위가 가능한 보안약점
4	크로스사이트 스크립트	사용자 브라우저에 검증되지 않은 외부 입력값을 허용하여 악의적인 스크립트가 실행 가능한 보안약점
5	운영체제 명령어 삽입	운영체제 명령어를 생성할 때 검증되지 않은 외부 입력값을 허용하여 악의적인 명령어가 실행 가능한 보안약점
6	위험한 형식 파일 업로드	파일의 확장자 등 파일형식에 대한 검증없이 파일 업로드를 허용하여 공격이 가능한 보안약점
7	신뢰되지 않는 URL 주소로 자동접속 연결	URL 링크 생성에 검증되지 않은 외부 입력값을 허용하여 악의적인 사이트로 자동 접속 가능한 보안약점
8	부적절한 XML 외부 개체 참조	임의로 조작된 XML 외부개체에 대한 적절한 검증 없이 참조를 허용하여 공격이 가능한 보안약점
9	XML 삽입	XQuery, XPath 질의문을 생성할 때 검증되지 않은 외부 입력값을 허용하여 악의적인 질의문이 실행 가능한 보안약점
10	LDAP 삽입	LDAP 명령문을 생성할 때 검증되지 않은 외부 입력값을 허용하여 악의적인 명령어가 실행 가능한 보안약점
11	크로스사이트 요청 위조	사용자 브라우저에 검증되지 않은 외부 입력값을 허용하여 사용자 본인의 의지와는 무관하게 공격자가 의도한 행위가 실행 가능한 보안약점
12	서버사이드 요청 위조	서버 간 처리되는 요청에 검증되지 않은 외부 입력값을 허용하여 공격자가 의도한 서버로 전송하거나 변조하는 보안약점
13	HTTP 응답분할	HTTP 응답헤더에 개행문자(CR이나 LF)가 포함된 검증되지 않은 외부 입력값을 허용하여 악의적인 코드가 실행 가능한 보안약점
14	정수형 오버플로우	정수형 변수에 저장된 값이 허용된 정수 값 범위를 벗어나 프로그램이 예기치 않게 동작 가능한 보안약점

번호	보안약점	설명
15	보안기능 결정에 사용되는 부적절한 입력값	보안기능(인증, 권한부여 등) 결정에 검증되지 않은 외부 입력값을 허용하여 보안기능을 우회하는 보안약점
16	메모리 버퍼 오버플로우	메모리 버퍼의 경계값을 넘어서 메모리값을 읽거나 저장하여 예기치 않은 결과가 발생하는 보안약점
17	포맷 스트링 삽입	str.format등 포맷 스트링 제어함수에 검증되지 않은 외부 입력값을 허용하여 발생하는 보안약점 * 포맷 스트링: 입·출력에서 형식이나 형태를 지정해주는 문자열

2. 보안기능

번호	보안약점	설명
1	적절한 인증 없는 중요 기능 허용	중요정보(금융정보, 개인정보, 인증정보 등)를 적절한 인증없이 열람(또는 변경) 가능한 보안약점
2	부적절한 인가	중요자원에 접근할 때 적절한 제어가 없어 비인가자의 접근이 가능한 보안약점
3	중요한 자원에 대한 잘못된 권한 설정	중요자원에 적절한 접근 권한을 부여하지 않아 중요정보가 노출·수정 가능한 보안약점
4	취약한 암호화 알고리즘 사용	중요정보 (금융정보, 개인정보, 인증정보 등)의 기밀성을 보장할 수 없는 취약한 암호화 알고리즘을 사용하여 정보가 노출 가능한 보안약점
5	암호화되지 않은 중요정보	중요정보(패스워드, 개인정보 등) 전송 시 암호화 또는 안전한 통신채널을 이용하지 않거나, 저장 시 암호화 하지 않아 정보가 노출 가능한 보안약점
6	하드코드된 중요정보	소스코드에 중요정보(패스워드, 암호화키 등)를 직접 코딩하여 소스코드 유출 시 중요정보가 노출되고 주기적 변경이 어려운 보안약점
7	충분하지 않은 키 길이 사용	암호화 등에 사용되는 키의 길이가 충분하지 않아 데이터의 기밀성·무결성을 보장할 수 없는 보안약점
8	적절하지 않은 난수 값 사용	사용한 난수가 예측 가능하여, 공격자가 다음 난수를 예상해서 시스템을 공격 가능한 보안약점
9	취약한 패스워드 허용	패스워드 조합규칙(영문, 숫자, 특수문자 등) 미흡 및 길이가 충분하지 않아 패스워드가 노출 가능한 보안약점
10	부적절한 전자서명 확인	프로그램, 라이브러리, 코드의 전자서명에 대한 유효성 검증이 적절하지 않아 공격자의 악의적인 코드가 실행 가능한 보안약점
11	부적절한 인증서 유효성 검증	인증서에 대한 유효성 검증이 적절하지 않아 발생하는 보안약점
12	사용자 하드디스크에 저장되는 쿠키를 통한 정보노출	쿠키(세션 ID, 사용자 권한정보 등 중요정보)를 사용자 하드디스크에 저장하여 중요정보가 노출 가능한 보안약점
13	주석문 안에 포함된 시스템 주요정보	소스코드 주석문에 인증정보 등 시스템 주요정보가 포함되어 소스코드 노출 시 주요정보도 노출 가능한 보안약점
14	솔트 없이 일방향 해쉬 함수 사용	솔트를 사용하지 않고 생성된 해쉬 값으로부터 공격자가 미리 계산된 레인보우 테이블을 이용하여 해쉬 적용 이전 원본 정보를 복원가능한 보안약점 *솔트: 해쉬 적용하기 전 평문인 전송정보에 덧붙인 무의미한 데이터
15	무결성 검사 없는 코드 다운로드	소스코드 또는 실행파일을 무결성 검사 없이 다운로드 받아 실행하는 경우, 공격자의 악의적인 코드가 실행 가능한 보안약점
16	반복된 인증시도 제한 기능 부재	인증 시도 수를 제한하지 않아 공격자가 반복적으로 임의 값을 입력하여 계정 권한을 획득 가능한 보안약점

3. 시간 및 상태

번호	보안약점	설명
1	경쟁조건 : 검사 시점과 사용 시점	멀티 프로세스 상에서 자원을 검사하는 시점과 사용하는 시점이 달라서 발생하는 보안약점
2	종료되지 않는 반복문 또는 재귀함수	종료조건 없는 제어문 사용으로 반복문 또는 재귀함수가 무한히 반복되어 발생할 수 있는 보안약점

4. 에러처리

번호	보안약점	설명
1	오류 메시지 정보노출	오류메시지나 스택정보에 시스템 내부구조가 포함되어 민감한 정보, 디버깅 정보가 노출 가능한 보안약점
2	오류상황 대응 부재	시스템 오류상황을 처리하지 않아 프로그램 실행정지 등 의도하지 않은 상황이 발생 가능한 보안약점
3	부적절한 예외처리	예외사항을 부적절하게 처리하여 의도하지 않은 상황이 발생 가능한 보안약점

5. 코드오류

번호	보안약점	설명
1	Null Pointer 역참조	변수의 주소 값이 Null인 객체를 참조하는 보안약점
2	부적절한 자원 해제	사용 완료된 자원을 해제하지 않아 자원이 고갈되어 새로운 입력을 처리할 수 없는 보안약점
3	해제된 자원 사용	메모리 등 해제된 자원을 참조하여 예기치 않은 오류가 발생하는 보안약점
4	초기화되지 않은 변수 사용	변수를 초기화하지 않고 사용하여 예기치 않은 오류가 발생하는 보안약점
5	신뢰할 수 없는 데이터의 역직렬화	악의적인 코드가 삽입·수정된 직렬화 데이터를 적절한 검증 없이 역직렬화하여 발생하는 보안약점 * 직렬화: 객체를 전송 가능한 데이터형식으로 변환 * 역직렬화: 직렬화된 데이터를 원래 객체로 복원

6. 캡슐화

번호	보안약점	설명
1	잘못된 세션에 의한 데이터 정보노출	잘못된 세션에 의해 인가되지 않은 사용자에게 중요정보가 노출 가능한 보안약점
2	제거되지 않고 남은 디버그 코드	디버깅을 위한 코드를 제거하지 않아 인가되지 않은 사용자에게 중요정보가 노출 가능한 보안약점
3	Public 메서드로부터 반환된 Private 배열	Public으로 선언된 메소드에서 Private로 선언된 배열을 반환(return)하면 Private 배열의 주소값이 외부에 노출되어 해당 Private 배열값을 외부에서 수정 가능한 보안약점
4	Private 배열에 Public 데이터 할당	Public으로 선언된 데이터 또는 메소드의 인자가 Private로 선언된 배열에 저장되면 Private 배열을 외부에서 접근하여 수정 가능한 보안약점

7. API 오용

번호	보안약점	설명
1	DNS lookup에 의존한 보안결정	도메인명 확인(DNS lookup)으로 보안결정을 수행할 때 악의적으로 변조된 DNS 정보로 예기치 않은 보안위협에 노출되는 보안약점
2	취약한 API 사용	취약한 함수를 사용해서 예기치 않은 보안위협에 노출되는 보안약점

제2절 용어정리

● Developer Economics State of the Developer Nation, 20th Edition

developernation.net에서 매년 165개국 30,000명 이상의 개발자들을 대상으로 설문조사를 하여 제공하고 있다. 웹, 모바일, 데스크톱, 클라우드, 산업용 IoT, 소비자 전자 제품, 임베디드소프트웨어, AR 및 VR등 다양한 분야의 설문을 실시하고 있다.

● AES(Advanced Encryption Standard)

미국 정부 표준으로 지정된 블록 암호 형식으로 이전의 DES를 대체하며, 미국 표준 기술 연구소 (NIST)가 5년의 표준화 과정을 거쳐 2001년 11월 26일에 연방 정보처리표준(FIPS 197)으로 발표하였다.

● DES 알고리즘

DES(Data Encryption Standard)암호는 암호화 키와 복호화키가 같은 대칭키 암호로 64비트의 암호화키를 사용한다. 전수공격(Brute Force)공격에 취약하다.

● HMAC(Hash-based Message Authentication Code)

해쉬 기반 메시지 인증 코드, MD5, SHA-1 등 반복적인 암호화 해쉬 기능을 비밀 공용키와 함께 사용하며, 체크섬을 변경하는 것이 불가능하도록 한 키 기반의 메시지 인증 알고리즘이다.

● HTTPS(Hypertext Transfer Protocol over Secure Socket Layer)

WWW(월드 와이드 웹) 통신 프로토콜인 HTTP의 보안이 강화된 버전이다.

● LDAP(Lightweight Directory Access Protocol)

TCP/IP 위에서 디렉토리 서비스를 조회하고 수정하는 응용 프로토콜이다.

● SHA(Secure Hash Algorithm)

해쉬알고리즘의 일종으로 MD5의 취약성을 대신하여 사용한다. SHA, SHA-1, SHA-2(SHA-224, SHA-256, SHA-384, SHA-512) 등의 다양한 버전이 있으며, 암호 프로토콜인 TLS, SSL, PGP, SSH, IPSec 등에 사용된다.

● umask

파일 또는 디렉토리의 권한을 설정하기 위한 명령어이다.

● 개인키(Private Key)

공개키 기반구조에서 개인키란 암·복호화를 위해 비밀 메시지를 교환하는 당사자만이 알고 있는 키이다.

● 공개키(Public Key)

공개키는 지정된 인증기관에 의해 제공되는 키값으로서, 이 공개키로부터 생성된 개인키와 함께 결합되어, 메시지 및 전자서명의 암·복호화에 효과적으로 사용될 수 있다. 공개키를 사용하는 시스템을 공개키 기반구조(Public Key Infrastructure, PKI)라 한다.

● 경로순회(directory traversal)

상대경로 참조 방식(“./”,“../”등)을 이용해 다른 디렉토리의 중요파일에 접근하는 공격방법으로 경로 추적이라고도 한다.

● 동적 SQL(Dynamic SQL)

프로그램의 조건에 따라 SQL문이 다르게 생성되는 경우, 프로그램 실행 시에 전체 쿼리문이 완성되어 DB에 요청하는 SQL문을 말한다.

● 동적 쿼리(Dynamic Query)

컬럼이나 테이블명을 바꿔 SQL 쿼리를 실시간 생성해 DB에 전달하여 처리하는 방식이다.

● 소프트웨어 개발보안

소프트웨어 개발과정에서 개발자 실수, 논리적 오류 등으로 인해 소프트웨어에 내재된 보안취약점을 최소화하는 한편, 해킹 등 보안위협에 대응할 수 있는 안전한 소프트웨어를 개발하기 위한 일련 의 과정을 의미한다. 넓은 의미에서 소프트웨어 개발보안은 소프트웨어 생명주기의 각 단계별로 요구되는 보안활동을 모두 포함하며, 좁은 의미로는 SW개발과정에서 소스코드를 작성하는 구현 단계에서 보안약점을 배제하기 위한 '시큐어코딩(Secure Coding)'을 의미한다.

● 소프트웨어 보안약점

소프트웨어 결함, 오류 등으로 해킹 등 사이버공격을 유발할 가능성이 있는 잠재적인 보안취약점을 말한다.

● 싱글톤 패턴(Singleton Pattern)

하나의 프로그램 내에서 하나의 인스턴스만을 생성해야만 하는 패턴이다. Connection Pool, Thread Pool과 같이 Pool 형태로 관리되는 클래스의 경우 프로그램 내에서 단 하나의 인스턴트로 관리해야 하는 경우를 말함.

● 정적 쿼리(Static Query)

동적 쿼리와 달리 프로그램 소스코드에 이미 쿼리문이 완성된 형태로 고정되어 있다.

● 해쉬함수

주어진 원문에서 고정된 길이의 의사난수를 생성하는 연산기법이며, 생성된 값은 '해쉬값'이라고 한다. MD5, SHA, SHA-1, SHA-256 등의 알고리즘이 있다.

● 화이트 리스트(White List)

블랙리스트(Black List)의 반대개념으로 신뢰할 수 있는 사이트나 IP주소 목록을 말한다.

● 파싱(Parsing)

일련의 문자열을 의미 있는 token(어휘 분석의 단위)으로 분해하고 그것들로 이루어진 Parse tree를 만드는 과정이다. 어떤 문장을 분석하거나 문법적 관계를 해석하는 행위를 말한다.

● 파서(Parser)

컴파일러(compiler)의 일부로 컴파일러나 인터프리터(Interpreter)에서 원시 프로그램을 읽어 들여 그 문장의 구조를 알아내는 parsing(구문 분석)을 행하는 프로그램을 말한다.

● XML(eXtensible Markup Language)

W3C에서 개발되었으며, 다른 특수한 목적을 갖는 마크업 언어를 만드는데 사용된다. 인터넷에 연결된 시스템끼리 데이터를 쉽게 주고받을 수 있어 HTML의 한계를 극복할 목적으로 만들어졌다.

● DTD(Document Type Definition)

문서 타입 정의(DTD)는 XML 문서의 구조 및 해당 문서에서 사용할 수 있는 적법한 요소와 속성을 정의한다.

● 공개 키 인증서(Public Key Certificate)

공개키의 소유권을 증명하는데 사용되는 전자 문서이다. 키에대한 정보, 소유자의 신원에 대한 정보, 발급자의 디지털 서명이 포함되어 있다.

● 솔트(salt)

솔트는 해싱 처리 과정 중 각 패스워드에 추가되는 랜덤으로 생성된 유일한 문자열을 의미한다.

〈비매품〉

JavaScript 시큐어코딩 가이드

인 쇄 2022년 12월
발 행 2022년 12월

발행처 과학기술정보통신부
세종특별자치시 가름로 194

한국인터넷진흥원
전라남도 나주시 진흥길 9

JavaScript
시큐어코딩 가이드

과학기술정보통신부
KISA 한국인터넷진흥원