Introduction of Command Injection and Practice

Student Jeongmin Lee

개요

본 보고서는 명령어 삽입 공격(Command Injection)에 대해 다루며, 수업에서 제공된 강의자료 및 유튜브 영상들을 참고하여 작성되었습니다. 부족하다고 판단되는 부분은 추가적으로 인터넷에서 관련 자료를 조사하여 내용을 보완하였습니다.

보고서를 통해서 명령어 삽입 공격의 개념, 유형, 실제 사례, 그리고 이를 방지하기 위한 모범 사례에 대해 심도 있게 이해하는 것을 목표로 합니다.

명령어 삽입 공격이란?

명령어 삽입 공격이란 웹 애플리케이션의 취약점을 악용하여 공격자가 임의의 시스템 명령을 실행할 수 있도록 하는 보안 취약점입니다. 이러한 공격은 사용자 입력이 적절하게 검증되지 않거나 필터링되지 않는 경우에 발생합니다. 공격자는 악의적인 명령어를 입력하여 시스템에서 원하지 않는 작업을 수행하거나 민감한 정보를 탈취할 수 있습니다.

명령어 삽입 공격은 SQL 삽입 공격과 유사합니다. 그러나 주요 차이점은 SQL 삽입이 데이터베이스에 SQL 명령을 삽입하는 반면, 명령어 삽입은 호스트 운영 체제에서 시스템 수준 명령을 실행할 수 있도록 한다는 점입니다. 다시 말해, SQL 삽입은 SQL 쿼리를 조작하여 데이터베이스를 대상으로 하는 반면, 명령어 삽입은 OS 수준에서 임의의 명령 실행을 허용하는 취약점을 악용합니다.

최악의 경우 공격자가 시스템에서 정보를 유출(exfiltrate)하거나 원격 코드 실행(Remote Code Execution, RCE)을 달성할 수 있습니다. 이를 통해 공격자는 시스템에 리버스 쉘(reverse shell)을 추가하여 완전한 제어 권한을 획득하고 웹 서버에서 임의의 명령을 실행할 수 있습니다.

간단한 예시

예를 들어, 사용자가 입력한 IP 주소를 이용해 서버가 다음과 같이 ping 명령을 실행한다고 가정합시다.

- 정상 입력: `ping 127.0.0.1`
- 공격 입력: `127.0.0.1; cat /etc/passwd`
- 실제 실행되는 명령: `ping 127.0.0.1; cat /etc/passwd`

이 경우, 서버는 ping 명령뿐만 아니라 추가로 `cat /etc/passwd` 명령도 실행하여 시스템의 패스워드 파일 내용을 공격자에게 노출할 수 있습니다. 아래는 Python으로 작성된 취약한 코드 예시입니다.

```
user_input = request.args.get("ip")  # e.g., "127.0.0.1; cat /etc/passwd"
cmd = "ping␣" + user_input
os.system(cmd)  # Executes the entire command via shell
```

Listing 1: 취약한 코드 예시 — ping 명령어에 사용자 입력을 직접 결합

명령어 삽입 공격의 유형

명령어 삽입 공격의 유형은 크게 세 가지로 나뉩니다.

첫째, 직접 명령어 삽입**(In-band Command Injection)**은 공격자가 삽입한 명령의 출력이나 결과가 애플리케이션의 응답(Http Response등)에 그대로 포함되어 돌아오는 경우를 가리킵니다. 이 유형의 경우 빠르고 단순하게 정보(디렉터리 목록, 파일 내용 등)을 확인 할 수 있습니다. 앞서 살펴본 간단한 예시가 이에 해당합니다.

둘째, 간접 명령어 삽입**(Blind Command Injection)**은 공격자가 명령을 삽입하여 서버에서 명령이 실행되더라도 명령의 결과(출력)가 HTTP 응답으로 직접 반환되지 않는 경우를 가리킵니다. 공격자는 출력 없이도 다양한 기법으로 명령의 성공/실패 또는 시스템의 정보를 얻어냅니다. 응답 본문에 명령 결과가 없기 때문에 공격자가 "눈으로 직접 보지 못한다"는 의미에서 'blind'라 부릅니다.

셋째, **Out-of-Band(OOB) Command Injection**는 명령의 출력이 공격자가 제어하는 별도 채널(예: DNS, HTTP 콜백, 이메일 등)로 전송되어 공격자가 그 채널에서 응답을 확인하는 경우를 가리킵니다. 일부 자료에서는 Blind의 하위 범주로 보기도 하고, 별도 유형으로 구분하기도 합니다. 저의 보고서에서는 별도 유형으로 구분합니다.

명령어 삽입 취약점의 영향 및 심각성

명령어 삽입 취약점은 시스템과 서비스에 매우 심각한 영향을 미칠 수 있습니다. 주요 영향과 그 심각성은 다음과 같습니다.

- 민감 정보 유출: 공격자는 시스템 명령을 통해 서버 내의 중요한 파일(예: /etc/passwd, 환경설정 파일 등)을 읽거나 외부로 전송할 수 있습니다. 이는 개인정보, 인증 정보, 시스템 설정 등이 노출되는 결과를 초래합니다.
- 원격 코드 실행**(RCE)** 및 권한 상승: 공격자는 임의의 명령을 실행하여 시스템에 악성코드(백도어, 리버스 쉘 등)를 설치하거나, 권한을 상승시켜 서버 전체를 제어할 수 있습니다. 이는 서비스 운영자에게 치명적인 피해를 줄 수 있습니다.
- 서비스 거부**(DoS)** 및 시스템 장애: 공격자가 무한 루프, 대용량 파일 생성, 시스템 자원 소모 명령 등을 실행하면 서비스가 중단되거나 서버가 다운될 수 있습니다.
- 데이터 무결성/가용성/기밀성 손상: 파일 삭제, 데이터 변조, 시스템 설정 변경 등으로 인해 데이터의 무결성, 가용성, 기밀성이 모두 위협받을 수 있습니다.
- 다른 시스템 공격의 발판: 명령어 삽입을 통해 공격자는 서버를 거점 삼아 내부망 또는 외부망의 다른 시스템을 추가로 공격할 수 있습니다. 예를 들어, 네트워크 스캐닝, 추가 악성코드 배포 등이 가능합니다.
- 법적/신뢰도 문제: 정보 유출이나 서비스 장애가 발생하면 기업이나 기관은 법적 책임, 평판 하락, 신뢰도 저하 등 2차 피해도 입을 수 있습니다.

이처럼 명령어 삽입 취약점은 단순한 정보 유출을 넘어 시스템 전체의 보안과 안정성, 조직의 신뢰도까지 심각하게 위협할 수 있으므로 반드시 예방 및 신속한 대응이 필요합니다.

예방 및 방어 전략

명령어 삽입(Command Injection) 취약점을 예방하는 핵심은 운영 체제의 셸 명령을 직접 호출하지 않는 것입니다. 즉, 가능한 한 애플리케이션 로직에서 셸을 거치지 않고, 안전한 API나 전용 함수만을 사용하는 것이 최우선 방어책입니다. 그 외의 보완적 방어 기법은 아래와 같습니다.

- 입력 검증 및 정규화**(Input Validation and Sanitization)**: 사용자 입력을 엄격히 검증하고 예상치 못한 메타문자가 포함되지 않도록 정규화합니다. 특히, 정규 표현식이나 스키마를 통해 허용되는 문자 집합을 제한합니다.
  - 화이트리스트 기반 검증: 허용된 값이나 패턴만 입력으로 받도록 합니다. 예: IPv4 주소는 숫자와 점(.)만 허용.

```
import re, subprocess

def is_valid_ipv4(ip):
    pattern = r"^([0-9]{1,3}\.){3}[0-9]{1,3}$"
    return re.match(pattern, ip) is not None

user_input = request.args.get("ip")
if is_valid_ipv4(user_input):
    subprocess.run(["ping", user_input], check=True)  # shell=False
else:
    raise ValueError("Invalid␣IP␣address")
```

Listing 2: Python 예시: IPv4 입력값 검증 및 안전 실행

- 안전한 **API** 및 라이브러리 함수 사용: 시스템 호출을 직접 문자열로 조합하기보다, 의도한 작업만 수행할 수 있는 안전한 함수 호출을 사용합니다. 예를 들어, 디렉터리 생성은 `os.mkdir()`와 같은 전용 API를 사용합니다.

```
import os

# Vulnerable way (allows command injection)
user_input = "new_directory;␣rm␣-rf␣/"  # Malicious input
os.system("mkdir␣" + user_input)

# Safe way (only creates directory)
os.mkdir(user_input)
```

Listing 3: 안전하지 않은 방식 vs. 안전한 방식

- 최소 권한 원칙**(Principle of Least Privilege)**: 애플리케이션을 불필요하게 높은 권한(예: root)으로 실행하지 않도록 하고, 실행 계정의 권한을 최소화하여 공격 성공 시에도 피해 범위를 줄입니다.
- 보안 옵션 및 프레임워크 활용: Python의 `subprocess` 모듈에서 `shell=False` 옵션을 사용하거나, 다른 언어에서도 인자 리스트 기반 실행 방식을 활용해 셸 해석을 회피합니다. 또한, 최신 보안 프레임워크나 라이브러리를 적용하여 입력 검증, 로깅, 오류 처리 등을 체계적으로 관리합니다.

```
import re
import subprocess

def is_valid_ipv4(ip: str) -> bool:
    pattern = r"^([0-9]{1,3}\.){3}[0-9]{1,3}$"
    return re.match(pattern, ip) is not None

user_input = request.args.get("ip", "")

if not is_valid_ipv4(user_input):
    raise ValueError("Invalid␣IP")

subprocess.run(["ping", "-c", "4", user_input], check=True, shell=False)
```

Listing 4: Python: subprocess.run 사용 (shell=False). 입력 검증 포함

- 출력 인코딩**(Output Encoding)**: 명령 실행 결과나 에러 메시지가 HTML, JSON 등으로 사용자에게 반환될 때는 반드시 적절히 인코딩합니다. 이는 Command Injection이 XSS 등 다른 취약점과 결합되는 것을 방지하는 추가적인 보안 계층입니다.

실제 적용

실제로 명령어 삽입 공격을 실습해보았습니다. 실습은 `https://portswigger.net/web-security/all-labs#os-command-injection` 에서 진행하였습니다.

실습을 진행하며, 위에서 설명하지 않은 부분들에 대해서 추가 설명을 덧붙이겠습니다.

Lab: OS command injection, simple case

문제 설명 : 이 랩은 상품 재고 조회기(product stock checker) 애플리케이션의 운영체제 명령어 삽입(OS command injection) 취약점을 다룹니다.

핵심 포인트
- 애플리케이션은 사용자가 입력한 상품 ID와 매장 ID를 포함한 셸 명령을 실행합니다.
- 셸 명령의 원시 출력(raw output)이 HTTP 응답에 그대로 반환됩니다.
- 목표: 취약점을 이용해 `whoami` 명령을 실행하고, 현재 사용자의 이름을 확인하세요.

Lab: Blind OS command injection with time delays

Lab: Blind OS command injection with output redirection

Lab: Blind OS command injection with out-of-band interaction

Lab: Blind OS command injection with out-of-band data exfiltration