SSTI ve CSTI Zafiyetleri – Ofansif Blog

İçindekiler

SSTI ve CSTI Nedir?

Web uygulamaları geliştirilirken, kullanıcı verilerini dinamik olarak sayfalara yerleştirmek için template engine (şablon motoru) kullanılır. Bu template engine'ler geliştiricilerin kodu daha temiz ve okunabilir yazmasına olanak tanır. Ancak template engine'lerin yanlış kullanılması ciddi güvenlik açıklarına yol açabilir.

Bu güvenlik açıklarının başında SSTI (Server Side Template Injection) ve CSTI (Client Side Template Injection) gelir.

- SSTI, sunucu tarafında çalışan şablon motorlarının kullanıcıdan gelen verileri doğru şekilde filtrelememesi sonucu oluşan bir güvenlik zafiyetidir. Bir saldırgan, template engine üzerinden sunucu üzerinde kod çalıştırabilir veya hassas bilgilere erişebilir.
- CSTI ise istemci (tarayıcı) tarafında çalışan şablon motorlarında oluşan bir açık türüdür. Saldırgan, tarayıcı içinde kötü amaçlı JavaScript kodları çalıştırarak kullanıcı bilgilerini ele geçirebilir veya saldırılar düzenleyebilir.

Bu iki zafiyet de uygulamaların güvenliğini tehlikeye atan ciddi problemler yaratabilir.

Dinamik Veri ve Template Engine Hakkında

Dinamik veri ve template engine kavramlarının şimdiden anlaşılması bu yazıyı anlamayı çok daha kolay hale getirecektir.

Dinamik Veri Nedir?

Dinamik veri bir uygulamanın çalışması sırasında kullanıcı girdisi, sistem olayları veya dış kaynaklardan alınan bilgilerle değişebilen verilerdir. Statik veriler sabitken, dinamik veriler uygulamanın durumu veya kullanıcı etkileşimleri sonucunda güncellenir.

Örneğin: Bir web sitesinde kullanıcı adının her girişte değişmesi (Merhaba, Emir! / Merhaba, Sude!) dinamik veridir.

Bir ürün listesi sayfasında stok durumu güncellenmesi dinamik veridir.

Template Engine Nedir?

Template engine (şablon motoru), geliştiricilerin statik HTML yapılarının içine dinamik veriler yerleştirmesini kolaylaştıran araçlardır. Bunlar HTML gibi sabit içeriklerle veri kaynaklarını birleştirerek son kullanıcıya gösterilecek dinamik sayfalar oluşturur. Örneğin:

```
<h1>Merhaba, {{ username }}!</h1>
```

Sunucu veya tarayıcı tarafında username “Emir” ile değiştirilirse kullanıcıya şu gösterilir:

```
<h1>Merhaba, Emir!</h1>
```

Terim	Tanım	Örnek
Dinamik Veri	Uygulama çalışırken değişebilen veri	Kullanıcı ismi, ürün fiyatı, stok durumu
Template Engine	Statik içerik+dinamik veriyi birleştiren araç	HTML şablonunda {{username}} değişkeni tanımlanması

Template Engine ve Dinamik Veri kavramlarının açıklaması bu şekildedir.

SSTI ve CSTI Zafiyetleri Neden Önemlidir?

SSTI ve CSTI zafiyetleri, saldırganların hedef uygulamalarda kontrolü ele geçirmesine, hassas bilgilere erişmesine ve sistem üzerinde yetkisiz işlemler gerçekleştirmesine olanak tanır.

Özellikle SSTI açıkları, saldırganın sunucu tarafında doğrudan kod çalıştırabilmesine (Remote Code Execution - RCE) kadar ilerleyebilir. Bu durum, sistemin tamamen ele geçirilmesine ve verilerin sızdırılmasına yol açabilir.

CSTI zafiyetleri ise istemci tarafında kötü amaçlı JavaScript kodlarının çalıştırılmasına (Cross-Site Scripting - XSS gibi) sebep olarak kullanıcıların oturum bilgilerinin çalınmasına veya çeşitli sosyal mühendislik saldırılarına zemin hazırlayabilir.

Gerçek dünyada, template injection açıkları hem bireysel kullanıcılar hem de kurumlar için ciddi güvenlik riskleri oluşturmakta ve veri sızıntısı, servis kesintisi veya maddi kayıplar gibi sonuçlara neden olabilmektedir. Bu nedenle, SSTI ve CSTI zafiyetlerinin anlaşılması ve önlenmesi, güvenli yazılım geliştirme süreçlerinin kritik bir parçasıdır.

Zafiyet	Risk Seviyesi	Potansiyel Sonuçlar
SSTI	Acil-Kritik	Sunucu kontrolü, Veri sızıntısı
CSTI	Orta-Yüksek	Oturum hırsızlığı, XSS saldırıları

SSTI (Server Side Templete Injection) Nedir?

SSTI (Server-Side Template Injection), bir web uygulamasının kullanıcıdan aldığı verileri sunucu tarafında çalışan bir template engine içerisine doğrudan ve güvenlik önlemi alınmadan yerleştirmesi sonucu oluşan bir güvenlik zafiyetidir.

Template engine’ler, dinamik içerik üretmek için geliştirilmiş araçlardır ve Python (Jinja2) ve PHP (Smarty) gibi birçok dilde yaygın olarak kullanılmaktadır. Eğer kullanıcı girdisi şablonun bir parçası haline getirilirken doğru şekilde filtrelenmezse, saldırganlar kendi kodlarını bu şablonlara enjekte edebilir.

Bu durum, template engine'in doğrudan saldırgan tarafından manipüle edilmesine ve sunucu üzerinde kötü amaçlı komutlar çalıştırılmasına (Remote Code Execution - RCE) kadar varabilecek ciddi güvenlik risklerine yol açar.

Benzer Durum Örneği:

Bir Flask uygulamasında kullanıcıdan alınan "name" değişkeni, doğrudan bir şablona gömülürse:

```
name = request.args.get('name')
render_template_string('Hello {{ ' + name + ' }}!')
```

Saldırgan, name parametresine template engine'in anlayacağı bir ifade ({{ 7*7 }} gibi) göndererek, sunucuda işlem yapılmasını sağlayabilir (burada 7*7 aslında komut çalıştırılabileceğinin bir kanıtı olarak işlev görür, eğer çalışmazsa bir zafiyet olmadığı anlamına gelmez farklı payloadlar (yükler) denenebilir). Sonuç olarak "Hello 49!" gibi bir çıktı üretilir. Daha ileri seviyelerde, sistemde komut çalıştırma gibi daha tehlikeli işlemler de gerçekleştirilebilir (reverse shell vs.).

SSTI zafiyetleri, çoğunlukla küçük bir ihmal sonucu ortaya çıkar ve saldırganlara çok yüksek seviyede yetki kazandırabileceği için son derece tehlikeli kabul edilir.

CSTI (Client-Side Template Injection) Nedir?

CSTI (Client-Side Template Injection), istemci tarafında (genellikle tarayıcıda) çalışan şablon motorlarının kullanıcıdan alınan verileri doğrudan ve yeterli kontrol olmadan işlemeleri sonucunda oluşan bir güvenlik zafiyetidir.

Modern web uygulamaları, dinamik içerik oluşturmak için sıklıkla client-side şablon motorları kullanmaktadır. Bu motorlar, kullanıcı girdilerini işleyerek sayfa üzerinde dinamik olarak veri gösterimi sağlar. Ancak, kullanıcı verileri doğrudan template engine'e aktarılırken uygun güvenlik önlemleri alınmazsa, saldırganlar kendi şablon kodlarını enjekte ederek tarayıcıda kötü amaçlı JavaScript kodları çalıştırabilir.

Benzer Durum Örneği:

Bir web sayfasında kullanıcı adı doğrudan bir şablon ifadesi içine yerleştiriliyorsa:

```
<div>
    {{username}}
</div>
```

Ve username değeri doğru şekilde filtrelenmiyorsa, bir saldırgan {{constructor.constructor('alert(1)')()}} gibi bir payload göndererek kullanıcının tarayıcısında zararlı JavaScript kodlarının çalışmasını sağlayabilir. Bu durum, klasik bir XSS (Cross-Site Scripting) saldırısına dönüşebilir.

CSTI zafiyetleri, saldırganlara kullanıcının oturum bilgilerini çalma, phishing saldırıları oluşturma veya kullanıcı tarayıcısında istenmeyen eylemler gerçekleştirme gibi imkanlar sunar. Bu nedenle istemci tarafı şablon motorlarının kullanımında da kullanıcı verilerinin dikkatle işlenmesi kritik öneme sahiptir.

SSTI Örnek Uygulama

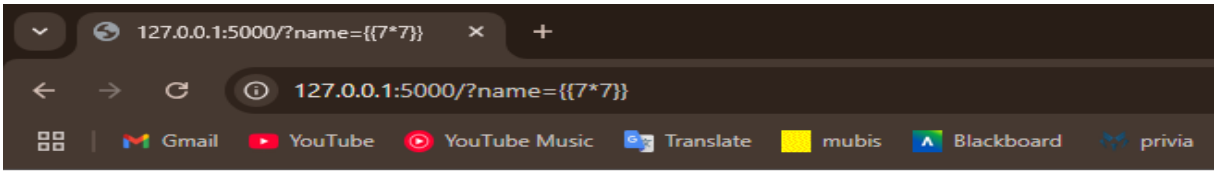
Flask ile SSTI

```
from flask import Flask, request, render_template_string

app = Flask(__name__)

@app.route('/')
def home():
    name = request.args.get('name', '')
    template = 'Hello ' + name
    return render_template_string(template)
```

```
if __name__ == '__main__':
    app.run(debug=True)
```

Kullanıcı, URL'ye name adında bir parametre vererek uygulamaya veri gönderiyor. Bu veri, doğrudan render_template_string() fonksiyonuna aktarılıyor. Güvenlik kontrolü yapılmadığı için, gönderilen veri template engine tarafından yorumlanıyor. Uygulama çalıştıktan sonra URL de kod execute edebiliyoruz. Örneğin:

http://127.0.0.1:5000/?name={{7*7}}



Hello 49

Sonuç:

Bu basit örnek, kullanıcı girdisinin doğrudan şablona işlenmesinin nasıl ciddi güvenlik açıklarına yol açabileceğini göstermektedir.

Gerçek hayatta SSTI zafiyetlerinin kötüye kullanılması, veri sızıntısından sunucu ele geçirmeye kadar pek çok ciddi sonuca sebep olabilir.

CSTI Örnek Uygulama

HTML ile CSTI

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>CSTI Example</title>
</head>
<body>

<h1>Client Side Template Injection Örneği</h1>

<p id="output"></p>

<script>
    // Kullanıcıdan gelen veri
    const params = new URLSearchParams(window.location.search);
    const name = params.get('name');

    // Şablon içine doğrudan yerleştiriliyor (zafiyet burada!)
    const template = `Merhaba, ${name}!`;
    document.getElementById('output').innerHTML = template;
</script>

</body>
</html>
```

Kullanıcı, URL'ye name adında bir parametre vererek uygulamaya veri gönderir.

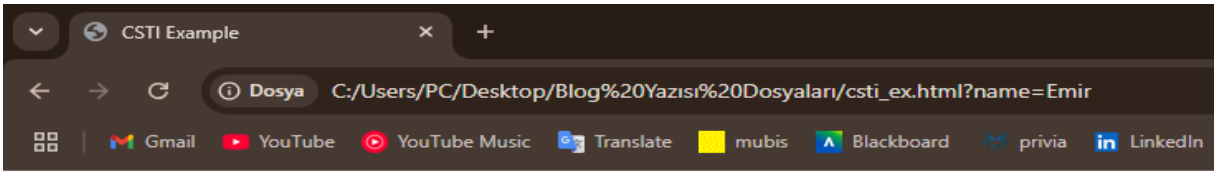
Bu veri doğrudan HTML içerisine (innerHTML) yerleştirilir.

Güvenlik önlemi alınmadığı için, kullanıcı tarafından sağlanan veri doğrudan tarayıcıda işlenir.

CSTI Zafiyetinin Sömürülmesi

HTML dosyasını tarayıcıda açtıktan sonra aşağıdaki gibi bir URL kullanarak normal bir kullanım sağlanabilir:

file:///C:/.../csti_example.html?name=Emir

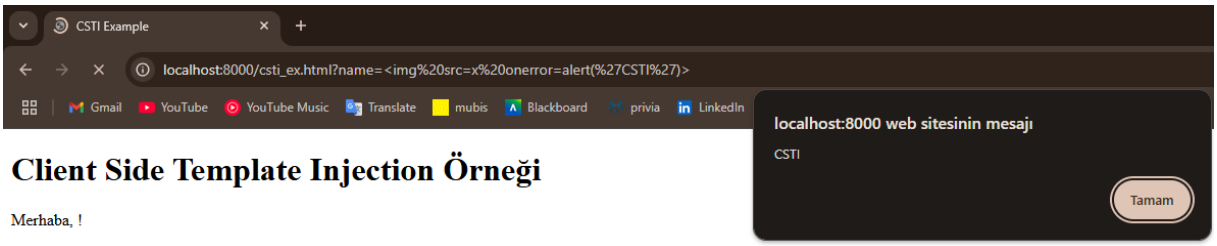


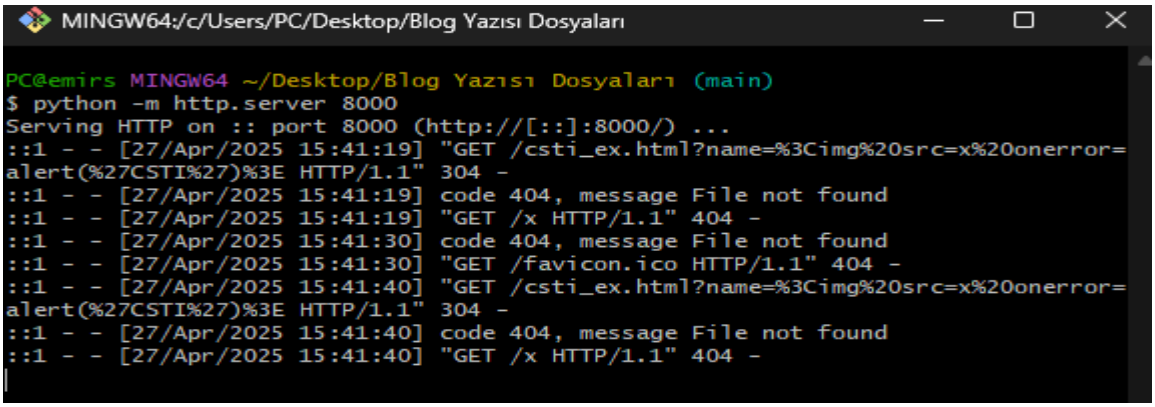
Bu ekranda Merhaba, Emir! Gibi bir çıktı versede saldırgan zararlı bir payload gönderebilir. Örneğin:

Dosyaların bulunduğu dizinde şu komutu çalıştırmanız ve sırasıyla verilen url'leri kullanmanız adım adım bu zafiyeti gösterecektir

python -m http.server 8000 (port isteğe bağlı olarak değiştirlebilir) --> Laboratuvar ortamı için küçük bir web sunucu ayağa kaldırılır.

http://localhost:8000/csti_ex.html?name=<img src=x onerror=alert('CSTI')> --> Zafiyetli URL





Bu durumda tarayıcıda bir JavaScript alert penceresi açılır. Bu bize kullanıcının tarayıcısında alert fonksiyonu ile javascript çalıştırdığımızı kanıtlar. Ayrıca bu, kullanıcıdan alınan verinin doğrudan şablon içerisine yerleştirildiği ve kötü niyetli kodların çalıştırılabildiği anlamına gelir.

Bu örnek, kullanıcı girdilerinin doğrudan şablon motorlarına veya HTML yapısına aktarılmasının nasıl güvenlik riskleri oluşturabileceğini göstermektedir.

Sonuç:

Bu yazıda SSTI ve CSTI zafiyetlerinin ne olduğunu, nasıl ortaya çıktıklarını ve basit örneklerle nasıl sömürülebileceklerini inceledik.

Bu tür güvenlik açıkları hem server tarafında hem de client tarafında ciddi tehditler oluşturabilir. Geliştiricilerin template engine kullanırken her zaman kullanıcı girdilerini güvenli bir şekilde işlediğinden ve template engine'lerin doğru yapılandırıldığından emin olmaları gerekmektedir.

Özellikle veri doğrulama ve güvenli template engine kullanımı gibi temel güvenlik pratikleri, bu zafiyetlerin önlenmesinde kritik rol oynamaktadır. Günümüzde, küçük bir şablon hatası bile büyük veri ihlallerine ve ciddi itibar kayıplarına yol açabileceğinden, SSTI ve CSTI gibi zafiyetlere karşı farkındalık sahibi olmak ve gerekli önlemleri almak son derece önemlidir.

Nasıl Önlenebilir?

SSTI ve CSTI zafiyetlerini önlemek için geliştiricilerin hem server tarafı hem de client tarafı güvenlik önlemlerini bütünsel bir yaklaşımla uygulamaları gerekmektedir.

Özellikle kullanıcı girdilerinin işlenmesi, template engine yapılandırılması ve browser güvenlik politikalarının uygulanması kritik öneme sahiptir.

Başlıca önleme yöntemleri şunlardır:

Kullanıcı Girdilerini Doğrulama, Filtreleme ve Temizleme: Kullanıcıdan alınan veriler uygulamaya dahil edilmeden önce mutlaka sıkı doğrulamalardan geçirilmelidir. Kabul edilen veri formatları net bir şekilde tanımlanmalı, gereksiz karakterler temizlenmeli ve yalnızca beklenen veri tiplerine izin verilmelidir. Template engine'e aktarılacak verilerde özellikle şablon ifadelerini tetikleyebilecek karakterler ({}, $, | gibi) dikkatle filtrelenmelidir.

Güvenli Template Engine Konfigürasyonu: Kullanılan template engine'ler güvenli çalışma modlarında yapılandırılmalı ve autoescaping (otomatik kaçış) özelliği aktif edilmelidir. Şablon içerisinde doğrudan kod yürütülmesini mümkün kılan fonksiyonlardan (eval, exec, Function constructor'ları gibi) kaçınılmalıdır. Ayrıca kullanıcı verileri doğrudan şablon kodları içine entegre edilmemeli, binding (veri bağlama) işlemleri güvenli API'lar kullanılarak yapılmalıdır.

İçerik Güvenlik Politikası (Content Security Policy - CSP) Uygulaması: Client tarafında çalışan template engine'lerde, kötü niyetli JavaScript kodlarının yürütülmesini önlemek amacıyla güçlü bir CSP uygulanmalıdır. CSP politikası ile sadece belirli kaynaklardan gelen script çalıştırmaya izin verilmeli (script-src kısıtlaması) ve inline script kullanımına mümkün olduğunca izin verilmemelidir. Ek olarak, Content-Type başlıklarının doğru yapılandırılması ve tarayıcı güvenlik mekanizmalarının desteklenmesi sağlanmalıdır.