

# Analyzing the Severity and Exploit Complexity of Server-Side Web Application Vulnerabilities

---

## Abstract

This paper investigates key vulnerabilities in web applications, focusing on server side critical security flaws identified in the OWASP Top 10 and implementations on them: Server-Side Request Forgery (SSRF), Broken Access Control, Server-Side Template Injection (SSTI), and Insecure Deserialization. These vulnerabilities, often found in modern web applications, pose significant risks, including the potential for remote code execution (RCE), unauthorized access, and account takeover. Through a detailed analysis of each vulnerability, this research examines how they are exploited in real-world scenarios and the potential consequences of their exploitation. The paper also discusses effective mitigation strategies to safeguard against these attacks, aiming to provide actionable insights for developers and security professionals to enhance web application security and prevent common exploit techniques.

**Keywords:** Web Application Security, OWASP Top 10, Server-Side Request Forgery (SSRF), Broken Access Control, Server-Side Template Injection (SSTI), Insecure Deserialization, Remote Code Execution (RCE), Account Takeover, API Vulnerabilities, Exploitation, Mitigation Strategies

## Introduction:

Web application vulnerabilities are chosen as a focus area due to their high frequency and significant security impact. According to the 2023 Verizon Data Breach Investigations Report (DBIR), over 70% of breaches involve web applications, highlighting their critical role in modern attack surfaces[1]. Additionally, the National Vulnerability Database (NVD) shows that a large proportion of reported vulnerabilities relate to web technologies, emphasizing their persistent exposure to threats. The dynamic nature of web applications, frequent updates, third-party integrations, and direct user interactions create a broad attack surface that is often exploited. Studying these vulnerabilities is essential to understand real-world security risks and to develop practical defense strategies:

**Server-Side/API Vulnerabilities:** Many modern web applications rely heavily on server-side logic and APIs to process data, often exposing critical endpoints to external users. Vulnerabilities such as broken access control, SSRF, and insecure deserialization in APIs can allow attackers to bypass authentication, access internal services, or manipulate system behavior, leading to severe security risks.

Such as, **RCE (Remote Code Execution), Access Control Management, and Account Takeover:** Remote Code Execution (RCE) vulnerabilities, often caused by issues like SSRF and SSTI, allow attackers to execute arbitrary code on the server, potentially gaining full control over the application environment. Furthermore, broken access control mechanisms and improper session handling can enable attackers to escalate privileges, take over user accounts, and access sensitive data, undermining the security of the entire application.

## Analysis of Complexity and Severity of the flaws

Broken access control occurs when a web application fails to enforce its intended authorization policies, allowing attackers to access sensitive pages or perform unauthorized actions. Unlike traditional input-based vulnerabilities such as SQL Injection or Cross-Site Scripting, access control vulnerabilities are challenging to detect because they often flow through legitimate control paths without obvious abnormal inputs, making them difficult to identify during normal development and testing phases[2]. Recent vulnerability data further demonstrates the practical significance of broken access control. Fig 1 summarizes ten recently published CVEs linked to access control weaknesses. All recorded vulnerabilities exhibit **low attack complexity** but **medium severity**, highlighting how easily attackers can exploit authorization flaws once discovered.

CVE number	Attack Complexity	Severity	CVSS 4.0 score
<a href="#">CVE-2025-3855</a>	Low	Medium	5.3
<a href="#">CVE-2025-31406</a>	Low	Medium	4.3
<a href="#">CVE-2025-31285</a>	Low	Medium	4.6
<a href="#">CVE-2025-28131</a>	Low	Medium	4.6
<a href="#">CVE-2025-2686</a>	Low	Medium	6.9
<a href="#">CVE-2025-26393</a>	Low	Medium	5.4
<a href="#">CVE-2025-2202</a>	Low	Medium	6.9
<a href="#">CVE-2025-0843</a>	Low	Medium	7.3
<a href="#">CVE-2024-9298</a>	Low	Medium	5.3
<a href="#">CVE-2024-7476</a>	Low	Medium	4.3

Fig 1.[3]

Insecure deserialization refers to a critical security flaw where untrusted serialized data is processed without proper validation, allowing attackers to construct object chains (gadget chains) that invoke dangerous behaviors during the deserialization phase. This vulnerability is particularly severe in languages like Java, where dynamic features such as polymorphism and reflection create a massive and unpredictable attack surface. Research shows that constructing exploit chains often relies on Property-Oriented Programming (POP), where adversaries craft nested object structures with specific property values to navigate application logic and reach sensitive methods like `Method.invoke()`. Traditional detection tools relying on static taint analysis face high false-positive rates due to Java's runtime polymorphism and complex method overriding hierarchies. Recent vulnerability disclosures underline the real-world risks associated with insecure deserialization[2]. As shown in Fig 2, ten newly reported CVEs related to deserialization flaws highlight the wide severity spectrum, ranging from medium to critical. Vulnerabilities such as **CVE-2025-20124** and **CVE-2025-25940** achieved **CVSS 4.0 scores of 9.9 and 9.8**, marking them as critical threats that enable remote code execution with minimal attack complexity. Even lower-complexity vulnerabilities like **CVE-2025-32375** demonstrate critical impact levels, emphasizing how relatively simple exploit chains can have devastating effects. These

results reinforce the importance of precise detection, secure coding practices, and automated mitigation strategies for handling serialized data safely.

CVE number	Attack Complexity	Severity	CVSS 4.0 score
<a href="#">CVE-2025-43708</a>	Low	Low	3.1
<a href="#">CVE-2025-32375</a>	Low	Critical	9.8
<a href="#">CVE-2025-27925</a>	High	High	8.5
<a href="#">CVE-2025-27816</a>	Low	Critical	9.8
<a href="#">CVE-2025-27520</a>	Low	Critical	9.8
<a href="#">CVE-2025-27218</a>	Low	Medium	5.3
<a href="#">CVE-2025-25940</a>	Low	Critical	9.8
<a href="#">CVE-2025-20124</a>	Low	Critical	9.9
<a href="#">CVE-2025-0586</a>	Low	High	7.2
<a href="#">CVE-2024-8316</a>	Low	High	7.8

Fig 2.

**Server-Side Template Injection (SSTI)** is a high-risk vulnerability that emerges when user-supplied inputs are incorrectly embedded into server-side templates, enabling unauthorized code execution. Recent analysis shows that many popular template engines, including Jinja2, Twig, and Velocity, inherently allow access to dangerous functionalities through introspection, exposing paths to Remote Code Execution (RCE). Template engines were categorized into four types of RCE exposure: direct code execution within templates, execution through special tags or functions, introspective traversal of object attributes, and exploitation of specific bugs in the engine itself. The significance of SSTI vulnerabilities is reinforced by recent CVE reports[4]. As shown in Fig 3, multiple vulnerabilities associated with SSTI demonstrate medium to critical severity levels, with several vulnerabilities such as CVE-2025-23211 and CVE-2025-25362 reaching CVSS 4.0 scores of 10 and 9.8, indicating critical security risks. Despite generally low attack complexity, the potential for complete server compromise remains high.

CVE number	Attack Complexity	Severity	CVSS 4.0 score
<a href="#">CVE-2025-25768</a>	Low	Medium	5.4
<a href="#">CVE-2025-25362</a>	Low	Critical	9.8
<a href="#">CVE-2025-23211</a>	Low	Critical	10
<a href="#">CVE-2025-2040</a>	Low	Medium	5.3
<a href="#">CVE-2025-1040</a>	Low	High	8.8
<a href="#">CVE-2024-7899</a>	Low	Medium	5.1
<a href="#">CVE-2024-6947</a>	Low	Medium	5.1
<a href="#">CVE-2024-6936</a>	Low	Medium	5.1
<a href="#">CVE-2024-6469</a>	Low	Medium	5.1
<a href="#">CVE-2024-57177</a>	Low	High	7.3

Fig 3.

**Server-Side Request Forgery (SSRF)** is a serious vulnerability in web applications where attackers manipulate server-side requests to access unauthorized internal systems or third-party resources. Modern PHP applications are particularly vulnerable due to dynamic features such as magic methods, variable classes, and variable functions, which complicate static analysis. Existing static detection tools often suffer from both false positives and false negatives because they fail to model application-specific sources and sinks, misinterpret PHP's dynamic behavior, and apply imprecise taint propagation rules. Recent advances proposed more accurate detection techniques by integrating precise taint analysis and large language models (LLMs), successfully identifying 24 SSRF vulnerabilities across 13 real-world PHP applications, including four newly discovered CVEs [5]. As summarized in Fig 4, the CVSS 4.0 scores for these SSRF-related vulnerabilities range from 4.9 to 7.5, with CVE-2025-32948 categorized as high severity at 7.5, while most others maintain medium severity levels. Notably, several vulnerabilities, despite having high attack complexity, still pose significant risk due to their potential internal network exploitation capabilities. These results highlight that SSRF remains a stealthy yet impactful attack vector that demands improved detection accuracy and stricter input validation controls in web platforms.

CVE number	Attack Complexity	Severity	CVSS 4.0 score
<a href="#">CVE-2025-46531</a>	High	Medium	4.9
<a href="#">CVE-2025-46511</a>	Low	Medium	6.4
<a href="#">CVE-2025-46503</a>	High	Medium	4.9
<a href="#">CVE-2025-46443</a>	High	Medium	4.9
<a href="#">CVE-2025-3787</a>	Low	Medium	5.1
<a href="#">CVE-2025-3691</a>	Low	Medium	5.1
<a href="#">CVE-2025-32948</a>	Low	High	7.5
<a href="#">CVE-2025-32691</a>	Low	Medium	4.9
<a href="#">CVE-2025-32675</a>	Low	Medium	6.3
<a href="#">CVE-2025-3254</a>	Low	Medium	5.3

Fig 4.

## General Overview about vulnerabilities complexity and severity:

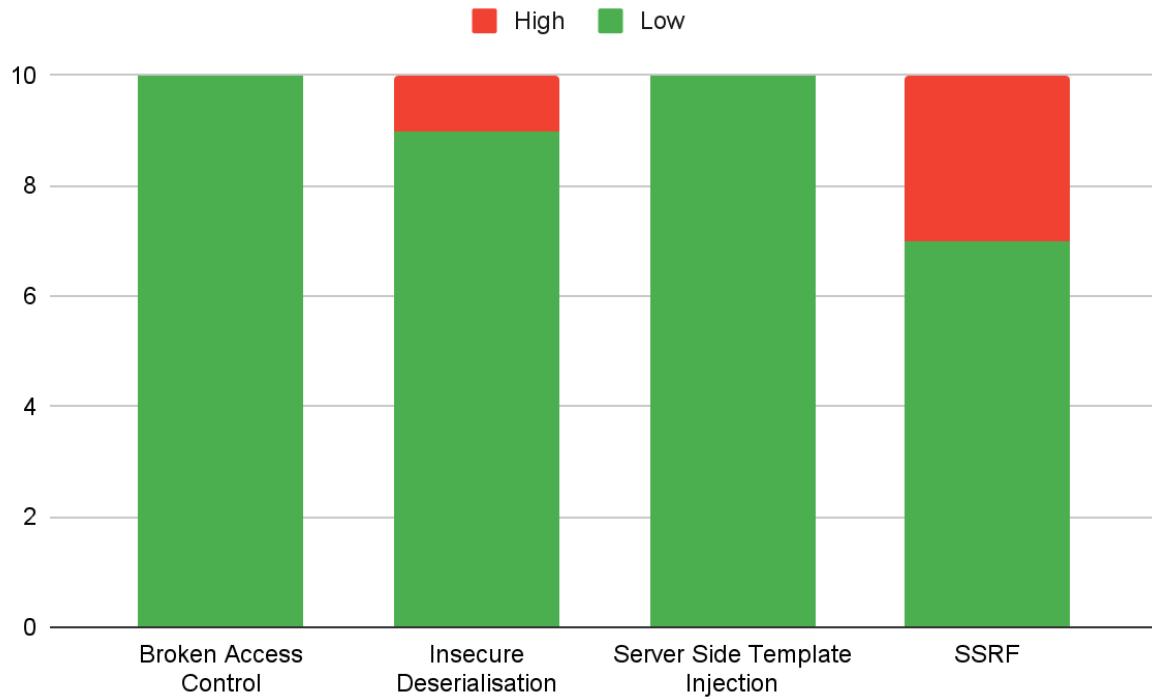


Fig 5.

An analysis of the collected CVE data across four vulnerability categories—Broken Access Control, Insecure Deserialization, Server-Side Template Injection (SSTI), and Server-Side Request Forgery (SSRF)—reveals significant patterns in both attack complexity and severity distribution. As illustrated in Figure 5, most vulnerabilities in Broken Access Control, SSTI, and Insecure Deserialization exhibit **low attack complexity**, meaning they can be exploited without requiring sophisticated attacker capabilities. SSRF, however, shows a comparatively higher share of **high-complexity attacks**, indicating that while SSRF is widespread, its successful exploitation can demand more precise targeting or specific application behaviors.

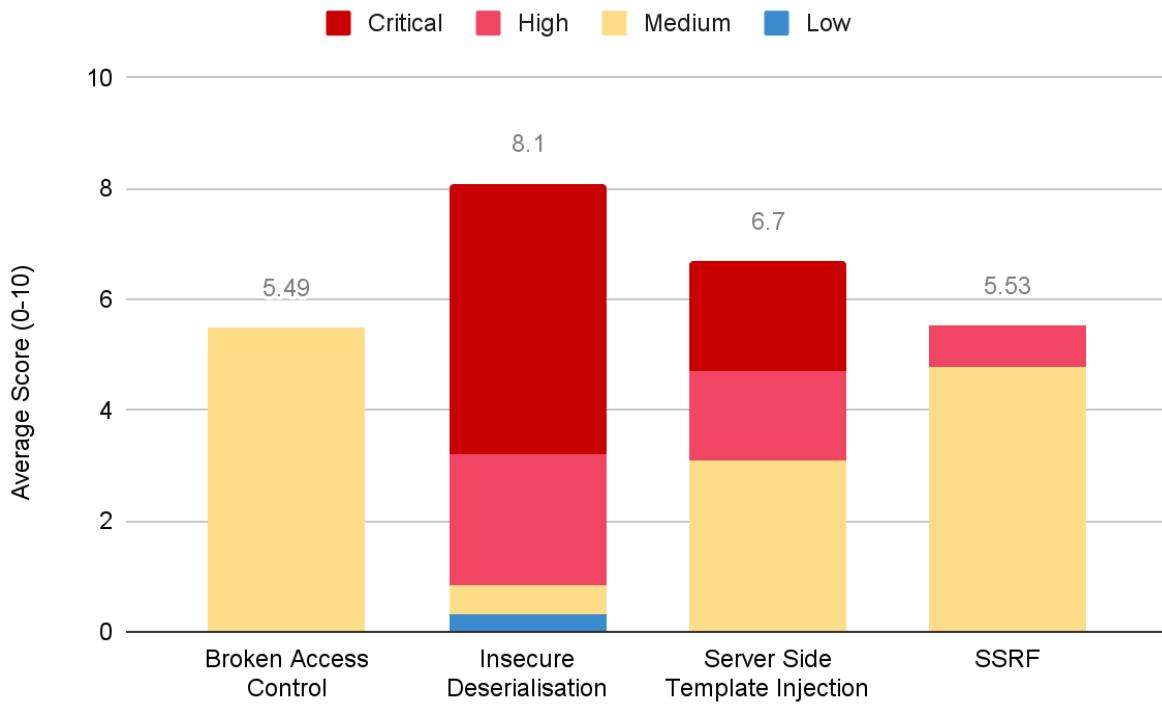


Fig 6.

In terms of severity (Figure 6), Insecure Deserialization stands out sharply, with a notably higher concentration of **critical and high-severity vulnerabilities**, and an average CVSS 4.0 score of **8.1**, the highest among all categories. SSTI vulnerabilities also display elevated risk, reflected by a significant portion classified as critical and an average severity score of **6.7**. In contrast, Broken Access Control and SSRF, while still impactful, maintain predominantly **medium severity** profiles, with average scores around **5.5**. These findings suggest that although exploitation difficulty remains relatively low across most categories, vulnerabilities like insecure deserialization and SSTI have far more damaging consequences when exploited, emphasizing the need for stricter input validation, better serialization practices, and stronger templating security in modern applications.

### The common mistakes, Good and Bad code styles:

#### Bad code style:

```

name = request.GET.get('name', '')

# Vulnerable to SSTI: Directly using user input in the template string
template_str = f'{name}'

template = Template(template_str)
    
```

```
rendered_name = template.render(name=name)
```

### Mistake:

The code directly incorporates **user input (name)** into the template string, which allows attackers to inject malicious template expressions, potentially leading to **remote code execution** when user input is reachable

### Good Code Style:

Where the user input is whitelisted:

```
name = request.GET.get('name', '')

# Sanitize user input to prevent template injection
sanitized_name = re.sub(r'^\w\s', '', name) # Allow only alphanumeric
and spaces

template_str = f'{sanitized_name}'
template = Template(template_str)
rendered_name = template.render(name=sanitized_name)
```

### Access Control issues:

Bad code example ,

```
@api_view(['POST'])
def change_user_role(request):
    user = basic_auth(request)
    if not user:
        return JsonResponse({'error': 'Authentication failed'}, status=403)

    target_user = request.data.get('target_user')
    new_role = request.data.get('new_role')

    if new_role not in ['admin', 'normal']:
        return JsonResponse({'error': 'Invalid role. Role must be either
"admin" or "normal".'}, status=400)

    try:
        target = User.objects.get(username=target_user)
        target_profile = UserProfile.objects.get(user=target)
        target_profile.role = new_role
        target_profile.save()
        return JsonResponse({'message': f'{target_user} is now {new_role}'})
    except User.DoesNotExist:
        return JsonResponse({'error': 'User not found'}, status=404)
```

### Mistake:

In the above code, there is **no access control check** implemented to ensure that the user performing the role change has the necessary permissions (e.g., only an admin should be allowed to change roles). As a result, any authenticated user can potentially change other users' roles, leading to **privilege escalation** and **unauthorized access**.

### Good Code Example (Access Control Check):

```
@api_view(['POST'])
def change_user_role(request):
    user = basic_auth(request)
    if not user:
        return JsonResponse({'error': 'Authentication failed'}, status=403)

    # Ensure the user is an admin before allowing role change
    if user.role != 'admin':
        return JsonResponse({'error': 'Unauthorized: Admin privileges required'}, status=403)

    target_user = request.data.get('target_user')
    new_role = request.data.get('new_role')

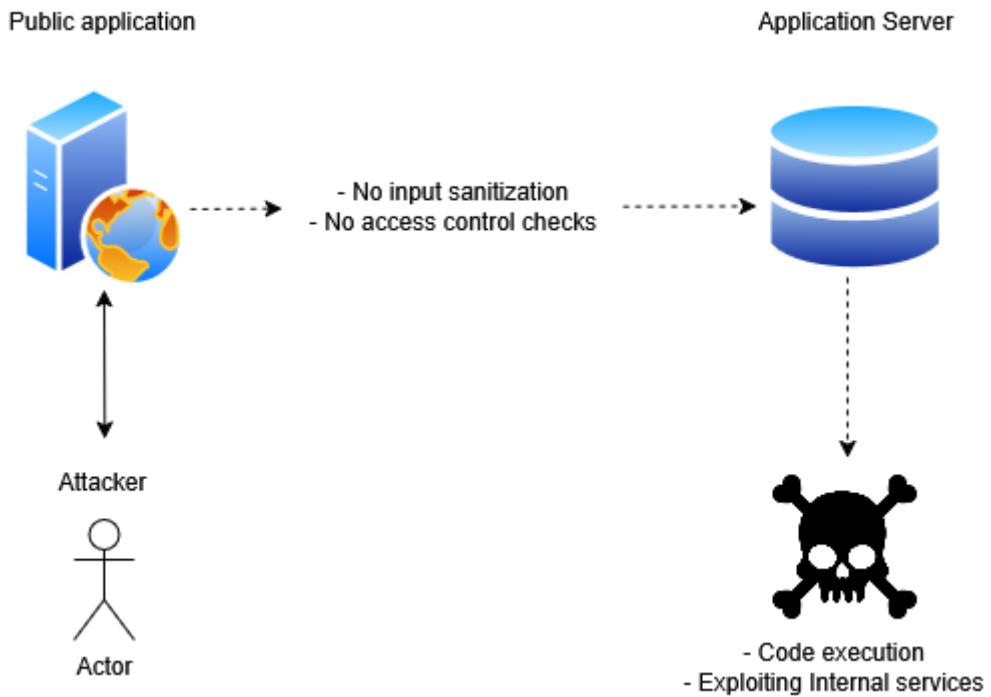
    if new_role not in ['admin', 'normal']:
        return JsonResponse({'error': 'Invalid role. Role must be either "admin" or "normal".'}, status=400)

    try:
        target = User.objects.get(username=target_user)
        target_profile = UserProfile.objects.get(user=target)
        target_profile.role = new_role
        target_profile.save()
        return JsonResponse({'message': f'{target_user} is now {new_role}'})
    except User.DoesNotExist:
        return JsonResponse({'error': 'User not found'}, status=404)
```

### Improvement:

The **access control check** ensures that only users with the `admin` role can modify other users' roles. If the logged-in user does not have the necessary permissions, the request is denied with a `403 Unauthorized` response. This mitigates the risk of unauthorized role changes and privilege escalation.

## Root Cause Analysis Of The Vulnerabilities:



Fig

**User Input Not Sanitized:** Insecure deserialization and Template injection often arise from failing to sanitize user input, allowing attackers to manipulate data and execute arbitrary code.

**Lack of Awareness:** Developers often overlook vulnerabilities like SSRF and broken access control due to insufficient security training, leading to misconfigurations that expose sensitive systems.

**No Secure Coding Policies:** The absence of secure coding standards contributes to vulnerabilities like SSTI and insecure randomness, where developers bypass security best practices for convenience.

## References:

- [ 1 ] Verizon. (2023). *2023 Data Breach Investigations Report (DBIR)*. Verizon.  
<https://inquest.net/wp-content/uploads/2023-data-breach-investigations-report-dbir.pdf>
- [ 2 ] Cao, S., Zhou, Y., Zhai, H., Xu, X., & Duan, L. (2023). *ODDFuzz: Discovering Java deserialization vulnerabilities via structure-aware directed greybox fuzzing*. In *Proceedings of the 2023 IEEE Symposium on Security and Privacy (SP)* (pp. 2726–2731). IEEE.  
<https://doi.org/10.1109/SP46215.2023.00177>

[3] The MITRE Corporation. (n.d.). *CVE – Common Vulnerabilities and Exposures*. <https://www.cve.org/>

[ 4 ] Pisu, L., Maiorca, D., & Giacinto, G. (2024). *A survey of the overlooked dangers of template engines*. arXiv. <https://arxiv.org/abs/2405.01118>

[ 5 ] Ji, Y., Dai, T., Tang, Y., & He, J. (2024, October). *Whether we are good enough to detect server-side request forgeries in PHP-native applications?* Poster presented at the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS '24), Salt Lake City, UT, USA. Retrieved from [https://tingdai.github.io/files/ssrf\\_CCS24.pdf](https://tingdai.github.io/files/ssrf_CCS24.pdf)

[ 6 ] Otojon. (2024). *Vulnerable application: Simulated web security flaws for educational use [Computer software]*. GitHub. <https://github.com/Otojon/vulnerable-application>