 The XSS Rat - CAPIE - Certified API Hacking Expert - V1.0

Chapter 1: Introduction to APIs

In today's hyperconnected world, software applications rarely operate in isolation. They constantly interact with other systems, fetching data and exchanging information to provide richer and more efficient user experiences. At the heart of many of these interactions is the **Application Programming Interface (API)**, a contract that defines how different software components should communicate.

What Is an API?

An **API** specifies the rules and protocols by which one piece of software can request and receive services or data from another. In practical terms, you might have an API that allows a mobile application to pull user profile information from a database, or a web service that provides stock quotes. APIs come in various flavors—from **operating system APIs** to **database APIs**—but our primary focus is on **web APIs**. Web APIs handle requests and responses over the internet (most commonly through HTTP) and include popular styles like **REST** and **SOAP**.

Why Are APIs Important?

APIs enable modular, scalable architectures where features can be broken out into separate, interoperable components. They also save development time, allow seamless third-party integrations, and help reuse existing services. However, APIs also introduce potential vulnerabilities if not properly secured. That's why understanding both the functional and security aspects of APIs is crucial—especially for pentesting roles or advanced development practices.

SOAP vs. REST: Two Key Styles of Web APIs

While there are many ways to build APIs, **SOAP** (Simple Object Access Protocol) and **REST** (Representational State Transfer) are two foundational styles you're likely to encounter.

SOAP Basics

SOAP is a protocol that typically relies on XML to format messages. It can operate over several network protocols (HTTP, SMTP, etc.) and has built-in error handling through its *Fault* element. Because SOAP can be stateful, it's often found in enterprise environments or legacy systems requiring rigid standards like ACID transactions or WS-* compliance.

- **Message Format**: Always XML

Example:

xml

```
<soap:Envelope xmlns:soap="http://www.w3.org/2003/05/soap-envelope">
  <soap:Body>
    <m:GetStockPrice xmlns:m="http://www.example.org/stock">
      <m:StockName>GOOG</m:StockName>
    </m:GetStockPrice>
  </soap:Body>
</soap:Envelope>
```

-
- **Error Handling**: Built-in via `<soap:Fault>`

- **Security Considerations**: SOAP supports WS-Security, among other extensions, making it suitable for scenarios that require more standardized security approaches.

- **Use Cases**: Large enterprise systems, banking transactions, and situations that demand strict standards.

REST Fundamentals

REST is an architectural style (rather than a protocol) that uses existing HTTP methods (GET, POST, PUT, DELETE) for communication. RESTful APIs are typically *stateless* and leverage standard HTTP status codes to signal success or failure. They often use JSON, but can also work with XML, HTML, or other formats.

- **Message Format**: Commonly JSON, but flexible

Example:

bash

```
POST /stocks
{
  "stock": "AAPL",
```

```
  "price": "145.00"
}
```

-
- **Error Handling**: Standard HTTP status codes (e.g., 404 for Not Found, 500 for Server Error)
- **Security Considerations**: REST APIs can implement any security mechanism that rides on HTTP (TLS/SSL, OAuth, etc.).
- **Use Cases**: Modern web or mobile apps, microservices, public-facing APIs for third-party integrations.

Key Differences

Aspect	SOAP	REST
Protocol	Strict protocol	Architectural style
State	Can be stateful	Typically stateless
Message	XML	JSON (commonly)
Error	Built-in `<soap:Fault>`	HTTP status codes
Complexity	More “heavyweight”	Generally simpler
Use Cases	Legacy/enterprise apps	Modern web/mobile apps

API Authentication & Authorization

Once you decide on your API style, the next critical aspect is controlling access. **Authentication** verifies someone is who they claim to be, while **Authorization** checks if they are allowed to access a specific resource or perform a certain action.

Common Methods

1. **Basic Authentication**

- Sends a Base64-encoded username and password with every request.
- Simple to implement but insecure without TLS encryption.

2. **API Keys**
    - A unique key provided to an API consumer, often passed in headers or query parameters.
    - Useful for identifying the calling application, but doesn't inherently tie to specific users.
3. **Bearer Tokens**
    - Similar to API keys but often represent a user or client session more explicitly.
    - Tokens can be short-lived or long-lived.
4. **OAuth 2.0**
    - A popular framework for delegated access.
    - Involves *access tokens* and *refresh tokens*, with multiple grant types (Authorization Code, Client Credentials, etc.).
    - Widely used by social logins (e.g., "Sign in with Google").
5. **JSON Web Tokens (JWT)**
    - A token format consisting of Header, Payload, and Signature.
    - Easy to pass around, but must be carefully secured (e.g., use HTTPS, manage signing secrets).
    - Fun fact, pronounced as "jot"
6. **OpenID Connect**
    - Built on top of OAuth 2.0.
    - Provides an *ID token* alongside access tokens, helping verify user identity.

Role-Based and Attribute-Based Access Controls

- **RBAC (Role-Based Access Control)**
  Each user is assigned a role (e.g., Admin, Editor, Viewer). Roles carry permissions, streamlining how access rights are managed.

- **ABAC (Attribute-Based Access Control)**
  Access is determined based on attributes (user attributes, resource attributes, context). Offers more fine-grained control but can be more complex to manage.

Other Considerations

- **Rate Limiting**: Prevents abusive use of APIs by restricting the number of requests within a time window.

- **Common Vulnerabilities**: Token leakage, man-in-the-middle attacks, and insecure direct object references (IDOR).

API Architectures

Beyond individual endpoints and security mechanisms, APIs exist within broader system **architectures** that shape how they're deployed, managed, and scaled.

Monolithic vs. Microservices

- **Monolithic**: A single, unified codebase. Simpler in initial development but can become cumbersome as the application grows.

- **Microservices**: Breaks functionalities into small, independent services. Improves scalability but increases architectural complexity—particularly around communication and security between services.

API Gateway

An **API gateway** acts as an entry point to route requests to the correct microservice or backend. It can handle:

- **Request routing and load balancing**

- **Caching**
- **Rate limiting**
- **Logging and monitoring**

Serverless & FaaS

Platforms like AWS Lambda or Azure Functions run your code in response to events without managing servers. This approach reduces overhead but introduces new security considerations (e.g., function-level privileges, ephemeral runtimes).

GraphQL

An alternative to REST, **GraphQL** allows clients to request exactly the data they need via **queries** and **mutations**. While powerful, poorly designed schemas can be vulnerable to malicious queries (e.g., overfetching, batch attacks).

Stateful vs. Stateless

- **Stateful**: The server keeps track of client session data.
- **Stateless**: All necessary information for processing a request is contained within that request itself. REST typically encourages stateless interactions.

Versioning

APIs evolve over time. **Versioning** strategies (in the URI, headers, or query parameters) help ensure backward compatibility. However, maintaining outdated versions can become a security and maintenance burden.

Logging & Monitoring

Effective **logging** and **monitoring** are vital for both performance and security. Logs can reveal anomalies, potential breaches, or usage patterns, but must be stored securely to prevent leaks of sensitive data.

API Documentation

 The XSS Rat - CAPIE - Certified API Hacking Expert - V1.0
A well-documented API not only boosts developer productivity but also reduces the risk of misconfiguration and misuse.

Components of Good Documentation

1. **Endpoints & Methods**: List your URLs (e.g., `/users`, `/orders`) and allowed methods (GET, POST, etc.).
2. **Parameters**: Clearly distinguish between required and optional parameters.
3. **Request & Response Examples**: Show real examples, including headers, body, and expected status codes.
4. **Error Handling**: Document possible error codes and messages.
5. **Authentication & Authorization**: Outline how to obtain and use credentials, tokens, or API keys.

OpenAPI & Swagger

- **OpenAPI (formerly Swagger)**: A widely used specification that can generate interactive API documentation.
- **Swagger UI**: Lets developers “try out” endpoints directly from the documentation.

Postman Collections

- Helpful for collaborative API testing.
- Collections can serve as live documentation: each request is stored with descriptions, parameters, headers, and scripts.

Common Pitfalls

- **Outdated or incomplete docs**: Leads to confusion and integration errors.
- **Ambiguities**: Failing to specify required authentication or valid parameter ranges.
- **Security Oversight**: Omitting how to properly handle tokens or lacking rate-limit info.

- **Over-reliance on code-generated Swagger/OpenAPI specs**: If the source code has a bug or omission, the auto-generated documentation will faithfully reproduce that mistake—so any error in the implementation ends up in the docs. This is especially common in low-code systems or when teams lean too heavily on generation tools.

Documentation's Role in Security

Documentation can be a double-edged sword: while it helps legitimate developers integrate responsibly, it also gives potential attackers insights into API structure. Balancing transparency with security best practices is key.

Conclusion & What's Next

APIs stand at the intersection of modern application development and integration, enabling powerful capabilities yet also introducing security challenges. In this introductory chapter, we:

1. Explored **what APIs are** and why they're integral to today's software landscape.
2. Reviewed **SOAP vs. REST**, two foundational API styles with their own use cases and security nuances.
3. Covered a range of **authentication and authorization** mechanisms, from basic auth to OAuth and JWT.
4. Investigated **API architectures**, including monolithic vs. microservices, API gateways, and considerations like GraphQL or serverless models.
5. Emphasized the importance of **API documentation**, which serves as both a development guide and a security checkpoint.

In upcoming chapters, we'll dive deeper into API security testing, examining common vulnerabilities, pentesting methodologies, and real-world scenarios that show how attackers target (and defenders secure) these critical interfaces. Stay tuned for more advanced topics on API pentesting, including **threat modeling**, **secure coding** best practices, and **penetration testing tools** specifically tailored for APIs.

Chapter 2: Tools to interact with an API

Once you understand the fundamentals of APIs—what they are, why they matter, and how they're structured—the next logical step is learning **how to effectively communicate** with them. In this chapter, we explore a range of tools (cURL, Postman, SOAP UI, and Python) that you can use to **send requests** and **examine responses** from APIs in different environments. Whether you're debugging a REST endpoint, testing a SOAP service, or writing automation scripts, these tools are invaluable for day-to-day work.

---

1. cURL: The Command-Line Workhorse

cURL is a powerful command-line utility used for sending and receiving data over various protocols, including HTTP and HTTPS. It's lightweight, cross-platform, and especially popular among developers, QA testers, and **pentesters** because it requires no graphical interface.

1.1 What Is cURL and Why Use It?

- **Definition**: "Client URL," often just called cURL, is a tool for transferring data with URLs.
- **Key Advantage**: It offers fine-grained control over requests (headers, payloads, authentication) directly from the command line, making it ideal for scripting, automation, and quick tests without an IDE or web browser.

1.2 Basic cURL Commands

GET Request:

bash

```
curl https://api.example.com/users
```

-

POST Request (x-www-form-urlencoded data):

bash

```
curl -X POST -d "username=john&password=pass123"
https://api.example.com/login
```

-

Custom Headers:

bash

```
curl -H "Authorization: Bearer <TOKEN>"
https://api.example.com/private-data
```

-

1.3 Handling Authentication

Basic Authentication:

bash

```
curl -u username:password https://api.example.com/auth-endpoint
```

-

Bearer Token:

bash

```
curl -H "Authorization: Bearer <TOKEN>"
https://api.example.com/auth-endpoint
```

-

API Keys:

bash

```
curl -H "x-api-key: <API_KEY>" https://api.example.com/data
```

-

1.4 Sending Various Data Formats

Form Data:

bash

```
curl -d "param1=value1&param2=value2" -X POST
https://api.example.com/form-endpoint
```

-

JSON Data:

bash

```
curl -X POST \
     -H "Content-Type: application/json" \
     -d '{"key1":"value1","key2":"value2"}' \
     https://api.example.com/json-endpoint
```

-

1.5 Additional cURL Features

- **Follow Redirects**:
  `curl -L https://api.example.com/redirect-endpoint`
- **View Headers Only**:
  `curl -I https://api.example.com/resource`
- **Verbose Output**:
  `curl -v https://api.example.com/resource`
- **Cookies**: Storing cookies with `-c cookies.txt` and reading them back with `-b cookies.txt`.

File Upload/Download:

bash

```
# Upload
curl -F "file=@/path/to/local/file.txt" -X POST
https://api.example.com/upload-endpoint

# Download
curl -o savedfile.jpg https://api.example.com/file-endpoint
```

-

cURL's versatility means you can handle everything from **simple GET requests** to **multi-step authenticated sessions**. It's a must-have tool for quickly testing endpoints in real time.

---

2. Postman: A Graphical Interface for API Exploration

Postman provides a user-friendly GUI for making requests, examining responses, and organizing projects into collections. It's popular among developers and QA teams for its ease of use and powerful collaboration features.

2.1 Introduction to Postman

- **What It Is**: An application (desktop or web-based) that simplifies API interactions.
- **Why It's Useful**:
    - No need to memorize command-line flags or syntax
    - Offers built-in support for authentication methods and parameter handling
    - Facilitates team collaboration through collections and shared workspaces

2.2 Making Basic Requests

1. **Create a new request** by selecting the HTTP method (GET, POST, PUT, DELETE, etc.).
2. **Enter the endpoint URL** and any parameters or headers.
3. **Send the request** and view the response status code, headers, and body in the Postman interface.

2.3 Collections & Collaboration

- **Collections**: Group related endpoints into one folder for easier organization.
- **Environments**: Store variables for different setups (e.g., dev vs. production).

- **Collaboration**: Share collections and environments with team members, reducing setup time and ensuring consistency.

2.4 Advanced Features

- **Authentication**: Built-in support for Basic Auth, Bearer Tokens, OAuth 2.0, etc.
- **Pre-request Scripts & Tests**: Automate steps before and after requests using JavaScript.
- **Runner & Newman**: Execute an entire collection in sequence or integrate with CI/CD pipelines via **Newman** (Postman's command-line runner).
- **Mock Servers**: Create mock endpoints to simulate server responses for front-end testing.

With Postman, you can **rapidly prototype** and **document** your APIs, while also automating certain tasks such as integration testing or performance checks.

---

3. SOAP UI: Dedicated Testing for SOAP & REST

Although it started as a SOAP-centric tool, **SOAP UI** now handles both SOAP and REST services. It excels at **project-based test organization**, **data-driven testing**, and **mocking** services to replicate real-world environments.

3.1 Introduction

- **History & Role**: Initially focused on SOAP-based web services, SOAP UI has grown to support REST, making it a comprehensive testing tool for many protocols.

3.2 Creating SOAP Projects

- **Import a WSDL**: SOAP UI reads the WSDL (Web Services Description Language) to automatically generate request templates.
- **SOAP Envelope Essentials**: Understand how the envelope, header, and body structure your SOAP message.

- **Assertions**: Validate parts of the XML response (e.g., checking for certain tags or values).

3.3 REST Projects

- **Setup**: Add a new REST project, specify endpoints, methods (GET, POST, etc.), and parameters.
- **Assertions & Test Cases**: Similar to SOAP, you can set up **REST** test steps and assertions to ensure the response meets expectations.

3.4 Advanced Testing & Features

- **Data-Driven Tests**: Pull data from files or databases to loop through multiple scenarios.
- **Property Transfer**: Share data (like tokens or IDs) between test steps.
- **Groovy Scripting**: Extend functionality with custom scripts for logic or advanced assertions.
- **Security Testing**: Basic scans for SQL injection, XSS, and other vulnerabilities.
- **Mock Services**: Simulate endpoints and responses for local or offline testing.

SOAP UI's structured approach makes it well-suited for **comprehensive regression testing** or **continuous integration** environments—especially where both SOAP and REST services coexist.

---

4. Python: Scripting and Automation for APIs

Python is a high-level language widely used for automation, data analysis, and (of course) **API interactions**. Its readability and extensive library ecosystem (especially the `requests` library) make it a top choice for both quick scripts and larger projects.

4.1 Introduction to Python for API Communication

- **Why Python?**

- Beginner-friendly syntax
- Rich ecosystem of libraries for HTTP requests, JSON handling, and more
- Great for one-off scripts or production-level applications

4.2 Making Basic GET & POST Requests

GET Example:

python

```
import requests

response = requests.get('https://api.example.com/data')
print(response.status_code)
print(response.json())
```

POST Example:

python

```
data = {'key': 'value'}
response = requests.post('https://api.example.com/post', data=data)
print(response.text)
```

4.3 Handling Headers, Auth, and JSON

python

```
# Custom header & Bearer token
headers = {
    'Authorization': 'Bearer <TOKEN>',
    'Content-Type': 'application/json'
}
response = requests.get('https://api.example.com/protected',
headers=headers)
```

```
# JSON data in a POST request
json_data = {'key1': 'value1', 'key2': 'value2'}
response = requests.post('https://api.example.com/post',
json=json_data)
```

4.4 Advanced Topics

Query Parameters:

python

```
params = {'search': 'python', 'page': 2}
response = requests.get('https://api.example.com', params=params)
```

-

Exception Handling:

python

```
try:
    response = requests.get('https://api.example.com', timeout=5)
    response.raise_for_status()
except requests.exceptions.RequestException as e:
    print("An error occurred:", e)
```

-

File Operations:

python

```
# Upload
files = {'file': open('filename.txt', 'rb')}
response = requests.post('https://api.example.com/upload',
files=files)

# Download
with open('downloaded.jpg', 'wb') as f:
```

```
f.write(response.content)
```

-
- **Session Management**: Persist cookies, headers, and other settings across multiple requests using `requests.Session()`.

4.5 When to Use Python

- **Automation**: Scheduled scripts for data retrieval or monitoring.
- **Integration**: Glue code for microservices or third-party APIs.
- **Data Processing**: Pull data from an API, transform it, and store it in a database.
- **Testing**: Write automated test scripts for continuous integration pipelines.

Conclusion & What's Next

With **cURL**, **Postman**, **SOAP UI**, and **Python**, you have a robust toolkit for nearly any API testing scenario. Here's a quick recap:

- **cURL**: Powerful command-line utility for quick requests and automation scripts in shell environments.
- **Postman**: Intuitive GUI for organizing requests, collaborating with teams, and quickly testing various endpoints.
- **SOAP UI**: Comprehensive testing environment specializing in SOAP (but also supporting REST) with features like data-driven tests and security scans.
- **Python**: Ideal for scripting and automation, leveraging the popular `requests` library for straightforward HTTP interactions.

In practice, teams often use a combination of these tools depending on their workflows. For example, a developer might prototype a request in Postman, then automate it later with Python. A QA engineer might set up SOAP UI for structured regression tests and still use cURL to quickly test server responses.

Going Forward:

- You can now **confidently interact** with both RESTful and SOAP services.
- Next steps might include exploring **best practices** for each tool (e.g., advanced scripting in Postman, groovy scripts in SOAP UI) or delving deeper into **performance testing** and **security validation** with these utilities.
- As you progress, consider how to incorporate these tools into your broader **CI/CD pipelines** to ensure ongoing quality and security checks for your APIs.

By mastering these core technologies, you'll be better positioned to **debug**, **test**, and **secure** APIs, ensuring seamless integration and robust performance across diverse software ecosystems.

 The XSS Rat - CAPIE - Certified API Hacking Expert - V1.0

Chapter 3: The OWASP API Top 10 - 2019

APIs are the connective tissue of modern software, linking mobile clients, web applications, IoT devices, and backend services. With APIs taking center stage in digital transformation, vulnerabilities targeting them have also risen sharply. In 2019, the Open Web Application Security Project (OWASP) released its **OWASP API Top 10**, outlining the most critical API weaknesses frequently exploited by attackers. Below is an expanded look at each category, illustrating how these vulnerabilities manifest, how they can be tested, and which remediation strategies help mitigate them.

1. Broken Object Level Authorization

Description

APIs often expose endpoints for specific resources—"objects" like user profiles, shopping carts, or content items. **Broken Object Level Authorization (BOLA)** arises when an API does not correctly confirm that the requesting user owns or is permitted to access a specific object. As a result, an attacker can modify object identifiers (IDs) in requests to access unauthorized data or resources.

Real-World Example

- **Manipulating Query Parameters**: An attacker changes `GET /api/orders/123` to `GET /api/orders/124` and gains another user's order details.

Pentesting Focus

- Enumerate and test object IDs in URL paths, body parameters, or query strings.
- Attempt horizontal privilege escalation (e.g., normal user accessing another user's data).
- Try vertical escalation (e.g., user accessing admin-only resources).

Remediation

- Enforce **server-side authorization** checks whenever accessing or modifying an object.
- Use **non-guessable identifiers** like UUIDs to reduce ID enumeration risks.

- Implement a consistent, centralized authorization mechanism.

---

2. Broken User Authentication

Description

Authentication mechanisms ensure that only legitimate users can log in and obtain valid session tokens. **Broken User Authentication** vulnerabilities occur when these mechanisms fail—whether via weak password policies, improper session management, inadequate token expiration, or missing multi-factor authentication (MFA).

Real-World Example

- **Unsecured Session IDs**: Session tokens remain valid long after a user logs out, allowing attackers to reuse them for unauthorized access.

Pentesting Focus

- Evaluate login endpoints for potential brute force or credential stuffing attacks (if permitted).
- Verify whether session tokens are invalidated upon logout.
- Check for missing MFA or any insecure password recovery flows.

Remediation

- Implement **strong password** requirements and lockouts for repeated failures.
- Use **JWT** or secure session tokens with short lifetimes and robust invalidation.
- Enforce **multi-factor** authentication for high-risk operations.

---

3. Excessive Data Exposure

 The XSS Rat - CAPIE - Certified API Hacking Expert - V1.0

Description

Sometimes APIs inadvertently return more data in their responses than the client actually requires. **Excessive Data Exposure** can reveal sensitive information like passwords, internal system fields, or debug data. Even if the front-end does not display these hidden fields, they remain accessible to anyone inspecting the raw response.

Real-World Example

- **Leaked PII**: A mobile banking application's JSON response contains user addresses, phone numbers, and account status fields, even though only the user's first name and balance are needed by the front-end.

Pentesting Focus

- Intercept and review responses carefully for sensitive fields.
- Observe if debug or error responses reveal stack traces or database schemas.

Remediation

- Filter and **whitelist** the exact fields the client needs.
- Use **DTOs (Data Transfer Objects)** to explicitly define what data is returned.
- Review logs for potential leaks from verbose error messages.

---

4. Lack of Resource & Rate Limiting

Description

APIs can be exploited if they allow **unlimited requests** or do not regulate the **size** of those requests. Attackers exploit this to launch denial-of-service (DoS) attacks, brute force user credentials, or overwhelm system resources.

Real-World Example

- **Bulk Request Spam**: Automated scripts send thousands of requests per second to an e-commerce API, crippling performance.

Pentesting Focus

- Attempt repeated login attempts to see if the service imposes lockouts or rate-limits.
- Send large payloads (e.g., multiple megabytes) to detect any enforced size threshold.

Remediation

- Implement **rate-limiting** or **throttling** policies at the application or gateway level.
- Set **payload size limits** and define graceful failure scenarios for oversized requests.
- Use **CAPTCHA** or MFA for repeated authentication attempts.

---

5. Broken Function Level Authorization

Description

Unlike Object Level Authorization, which focuses on specific data objects, **Function Level Authorization** deals with permissions to execute certain actions. If the API incorrectly validates user roles (e.g., admin vs. standard user), attackers could trigger privileged operations using a low-privilege account.

Real-World Example

- **Regular User Invokes Admin Endpoint**: Accessing `/api/admin/deleteUser` while logged in as a non-admin user, due to missing role checks.

Pentesting Focus

- Map the API's available functions and endpoints, identifying those exclusive to certain roles.

- Attempt calls to privileged endpoints with lower-level credentials.

Remediation

- Use **RBAC (Role-Based Access Control)** or **ABAC (Attribute-Based Access Control)** to restrict sensitive endpoints.
- Ensure **consistent** role/permission checks in both the application and any microservices or gateway layers.

---

6. Mass Assignment

Description

Modern frameworks often provide auto-binding of request data to server-side models. **Mass Assignment** exploits this by **injecting extra parameters** into the request that map to fields a user shouldn't control—potentially changing user roles, payment statuses, or other critical fields.

Real-World Example

- **Privilege Escalation**: A user includes `"role": "admin"` in the JSON body while signing up, and the server automatically sets the new user's role to admin.

Pentesting Focus

- Add extra or guessed parameters to requests to see if they modify restricted fields.
- Look for hints in responses (Excessive Data Exposure) that reveal internal field names.

Remediation

- Use **whitelists** for allowable fields.
- Adopt **DTOs** that explicitly define the properties permitted in incoming data.

- Validate each field server-side, rejecting unknown or privileged attributes.

---

7. Security Misconfiguration

Description

APIs—and the servers hosting them—rely on numerous components: web servers, databases, frameworks, or cloud services. **Security Misconfiguration** arises when default or weak settings persist, unneeded ports remain open, or verbose error messages give away internal details.

Real-World Example

- **Exposed Admin Interfaces**: A test environment with default admin credentials or open debugging endpoints.

Pentesting Focus

- Check for unused features or misconfigured HTTP methods (e.g., PUT, DELETE left open).
- Scan for open ports, default passwords, or directory listings.
- Inspect error handling for sensitive data in debug logs.

Remediation

- Harden servers: remove defaults, disable unused services, and set least privilege.
- Configure frameworks securely: restrict directories, enforce HTTPS, sanitize error logs.
- Maintain and regularly review your deployment environment.

---

8. Injection

 The XSS Rat - CAPIE - Certified API Hacking Expert - V1.0

Description

Injection vulnerabilities occur when untrusted data is fed into an interpreter—like SQL, NoSQL, OS commands, or LDAP—without proper sanitization or parameterization. This can lead to unauthorized data access, data manipulation, or even remote code execution.

Real-World Example

- **SQL Injection**: `SELECT * FROM users WHERE username = 'admin' OR 1=1; --' AND password = 'anyvalue'` returns all user records, bypassing login.

Pentesting Focus

- Provide malicious inputs in request parameters and observe if the server returns unusual errors or behaviors.
- Test for time-based or blind injections by measuring response times or side effects.

Remediation

- Always use **parameterized queries** or prepared statements.
- Validate and sanitize all incoming data.
- Employ secure coding practices in database or command-line interactions.

---

9. Improper Assets Management

Description

Many organizations deploy multiple API versions, testing endpoints, or staging environments. If they fail to **document** and secure these assets, attackers can exploit older, unpatched endpoints or discover new features early in development.

Real-World Example

- **Deprecated API**: An outdated version (v1) remains functional, containing unpatched vulnerabilities from a year ago.

Pentesting Focus

- Enumerate subdomains, versioned URLs, or hidden endpoints (`/v1`, `/dev`, etc.).
- Attempt connections to these endpoints and see if they are less secure.

Remediation

- Implement an **API lifecycle** policy: deprecate, retire, or patch old endpoints.
- Maintain an accurate **inventory** of all environments (production, staging, test).
- Use **strict access** controls for non-production services.

---

10. Insufficient Logging & Monitoring

Description

Poor or nonexistent logging can let intrusions slip by undetected, delaying or preventing incident response. **Insufficient Logging & Monitoring** means the system doesn't keep track of critical actions—like login attempts, data access, or suspicious request patterns.

Real-World Example

- **Failed Login Flood**: Attackers attempt thousands of password guesses, and the system does not log or alert admins.

Pentesting Focus

- Generate suspicious activity (invalid tokens, repeated login failures, etc.) and see if any **alerts** are triggered.

- Check for the presence of logs that detail errors, warnings, and usage patterns.

Remediation

- Log **all critical events** (auth changes, new user creations, admin actions).
- Ensure logs are **tamper-proof** and kept in a secure location.
- Integrate logs into **monitoring** and **alerting** systems, like SIEM solutions.

Conclusion

From insufficient authorization checks to insecure defaults, each vulnerability in the **OWASP API Top 10 (2019)** highlights a distinct weakness that adversaries routinely exploit. By understanding and mitigating these issues, teams can establish a robust baseline of **API security**. While this list isn't exhaustive, it provides a vital starting point for **pentesters**, **developers**, and **security engineers** aiming to build, test, or maintain safer APIs. Going forward, continuous **monitoring**, **testing**, and **compliance checks** are essential in keeping pace with evolving threats—and ensuring that APIs remain the reliable backbone of modern applications.

 The XSS Rat - CAPIE - Certified API Hacking Expert - V1.0

Chapter 4: The OWASP API Top 10 - 2023

As APIs increasingly serve as the backbone of modern applications—from mobile apps to IoT devices—the **attack surface** they create continues to expand. The **OWASP API Security Top 10 (2023)** provides vital insights into the most critical API security risks plaguing organizations today. Below, you'll find each category explained, along with common exploit scenarios and mitigation strategies.

1. Broken Object Level Authorization (BOLA)

What It Is

Often regarded as one of the most severe API flaws, BOLA occurs when an API endpoint fails to verify that a user is allowed to access or modify a particular object. Attackers exploit predictable or easily manipulated object identifiers—e.g., changing `/api/orders/123` to `/api/orders/124`—to gain unauthorized access.

Exploit Example

- A malicious user enumerates possible user or order IDs and accesses sensitive data belonging to other accounts.

Mitigation

- Enforce **server-side authorization** checks on every request.
- Use **non-guessable identifiers** such as UUIDs.
- Consistently validate object ownership or permissions.

---

2. Broken Authentication

What It Is

APIs rely on robust authentication systems (passwords, tokens, multi-factor mechanisms) to validate users or clients. When these fail—through poor password policies, insecure token

management, or missing session timeouts—attackers can log in as other users or hijack active sessions.

Exploit Example

- Weak password requirements allow brute-forcing, or session tokens stay valid even after logout, enabling attackers to reuse them.

Mitigation

- Implement **strong credentials** and enforce secure password reset flows.
- Use **short-lived tokens** (e.g., JWT) with proper revocation on logout.
- Apply **multi-factor authentication** for critical operations.

---

3. Broken Object Property Level Authorization

What It Is

Similar to BOLA but focused on **object properties**. Some frameworks auto-bind user inputs to model objects, allowing attackers to set properties they shouldn't control (often referred to as "Mass Assignment" in older classifications). For instance, sending `{"role":"admin"}` in a JSON payload can grant elevated privileges if the API lacks property-level checks.

Exploit Example

- An API automatically maps any JSON field to the database model. The attacker includes extra fields (`"isAdmin": true`) to escalate privileges or modify protected data.

Mitigation

- Whitelist only allowable properties in incoming JSON or form data.
- Validate each field server-side (reject unknown or sensitive fields).

- Use **Data Transfer Objects (DTOs)** to separate internal models from external inputs.

---

4. Unrestricted Resource Consumption

What It Is

APIs that fail to limit the **quantity** or **frequency** of requests and the **size** of incoming data can face denial-of-service (DoS) conditions or brute-force attacks. Attackers exploit lack of rate limiting or payload size checks to overwhelm the service or systematically guess credentials.

Exploit Example

- Sending massive JSON payloads repeatedly, consuming excessive memory or CPU on the server, leading to degraded performance or downtime.

Mitigation

- Implement **rate limiting** (e.g., request throttling) at the API gateway or server.
- Enforce **payload size constraints** and reject overly large requests.
- Set timeouts and concurrency limits to prevent resource exhaustion.

---

5. Broken Function Level Authorization

What It Is

While BOLA covers object access, **Broken Function Level Authorization** deals with permissions to specific **actions** or **endpoints**. An attacker with lower privileges can invoke high-privilege functions if the API fails to verify user roles thoroughly.

Exploit Example

- A standard user calls `/admin/deleteUser` or `/admin/reporting` endpoints by crafting direct requests, bypassing any front-end restrictions.

Mitigation

- Enforce **role-based or attribute-based** checks for sensitive functions.
- Validate user privileges on every endpoint, not just the front-end interface.
- Keep a well-documented list of which roles can call which API methods.

---

6. Unrestricted Access to Sensitive Business Flows

What It Is

Some APIs expose high-impact actions or “business flows” (e.g., financial transactions, bulk data exports) without controlling or monitoring usage. Attackers exploit these to perform large-scale actions, often without hitting typical user-oriented rate limits.

Exploit Example

- A malicious user calls a special endpoint for “bulkDeleteAccounts” or “bulkExportRecords,” which lacks any additional checks because it’s rarely used in normal user flows.

Mitigation

- Mark business-critical endpoints and apply **stricter authentication/authorization** or additional **verification steps** (e.g., MFA).
- Enforce separate **rate limiting** or usage tracking for high-impact operations.
- Monitor logs for unusual spikes in business flow usage.

---

7. Server Side Request Forgery (SSRF)

What It Is

SSRF arises when an API endpoint fetches a resource (URL, file, etc.) based on user input but fails to validate or sanitize it. Attackers can manipulate these requests to target internal services, access internal metadata, or pivot deeper into the network.

Exploit Example

- An endpoint `POST /api/fetchUrl` that retrieves external URLs is fed `http://localhost:8080/secure-admin`, allowing the attacker to read internal files or internal admin panels.

Mitigation

- Restrict outbound requests to a **whitelist** of allowed domains or IP ranges.
- Validate or sanitize user-supplied URLs, disallowing internal or link-local addresses.
- Segment network architecture so that API servers have limited internal resource access.

---

8. Security Misconfiguration

What It Is

APIs run on complex stacks—web servers, frameworks, containers, microservices. **Security Misconfiguration** covers default or weak configurations, leaving debug endpoints, open ports, or verbose error messages accessible to attackers.

Exploit Example

- Leaving debugging endpoints like `/debug/metrics` open, revealing internal server data or logs with sensitive credentials.

Mitigation

- Regularly **audit configurations** and remove unnecessary services or endpoints.
- Set minimal privileges for each component (principle of least privilege).
- Keep environments consistent with **Infrastructure as Code** (IaC) and automated checks.

---

9. Improper Inventory Management

What It Is

Organizations often operate multiple API versions, staging servers, or leftover endpoints from previous releases. **Improper Inventory Management** means failing to maintain a complete record of these assets, leaving older, insecure endpoints online.

Exploit Example

- Attackers discover `/api/v1/` or `/test/` endpoints that still exist in production and contain unpatched vulnerabilities.

Mitigation

- Track all public-facing APIs using **inventory tools** or **API gateways**.
- Deprecate or patch old versions promptly; retire them fully when no longer needed.
- Implement **strict access controls** on non-production environments.

---

10. Unsafe Consumption of APIs

What It Is

APIs often **consume** data or functionalities from third-party services. If those external APIs are insecure, or if the integration fails to handle unexpected responses gracefully, attackers can exploit these dependencies to compromise your system.

Exploit Example

- An internal microservice calls a partner API for order processing. The partner's API is compromised and sends malicious data that triggers a downstream injection in the consuming service.

Mitigation

- Validate and sanitize **all external data**—even from trusted vendors.
- Implement **version pinning** and robust input checks for third-party libraries or services.
- Use **zero-trust principles** when designing integrations, limiting the blast radius if an external service is compromised.

Chapter 5: API Pentesting Documentation

\Modern API pentesting goes beyond the technical act of finding vulnerabilities. It also involves **planning** the test, **documenting** results, and **coordinating** with stakeholders. This chapter covers three critical aspects of that workflow:

1. **Test Plan** – Outlining how the pentest will be conducted.
2. **Test Report** – Detailing the vulnerabilities discovered and how to fix them.
3. **Test Debrief Meeting** – Reviewing the findings, remediations, and lessons learned with all relevant parties.

---

1. The Test Plan

1.1 Definition and Importance

A **Test Plan** is a structured document that clarifies the **objectives**, **scope**, and **methodology** of a pentest. By setting clear goals and detailing the resources and tools to be used, the Test Plan ensures **completeness** and **consistency** throughout the engagement.

1.2 Preparing for an API Pentest

- **NDA and Documentation**: Securely share all relevant information (e.g., Postman collections, Swagger files) under a Non-Disclosure Agreement.
- **Grey-Box or Source Code-Assisted**: Decide on the testing approach. *Grey-box* testing can provide insights into API functionality (e.g., partial internal architecture knowledge), improving the accuracy and speed of the pentest.
- **OWASP Web Security Testing Guide**: Leverage established standards to identify common weaknesses and best practices.

1.3 Example Test Plan

A typical pentest plan for an organization like **labs.hackxpert.com** might include:

- **Objective**: Identify and remediate potential security issues across all APIs.

- **Scope**: Every endpoint on `labs.hackxpert.com`; third-party systems out of scope.
- **Methodology**: Grey-box testing, referencing OWASP guidelines for thorough coverage.
- **Execution**: Evaluate authentication flows, test for SQL Injection, XSS, and other vulnerabilities.
- **Reporting**: Document findings with risk ratings and remediation steps.
- **Post-Testing**: Conduct a lessons-learned session to refine future assessments.

By aligning the team on what will be tested (and how), the Test Plan minimizes confusion and ensures you can measure success objectively once the engagement is complete.

---

2. The Test Report

2.1 Purpose and Essential Components

A **Test Report** is the main deliverable summarizing the **pentest methodology**, **findings**, **recommendations**, and any supporting details. It provides:

1. **Executive Summary**: High-level overview for non-technical stakeholders (risk exposure, key takeaways).
2. **Methodology**: Detailed explanation of the testing approach, tools, and scope.
3. **Findings**: Categorized vulnerabilities with severity, proof of concept, and potential impact.
4. **Recommendations**: Concrete steps to remediate each issue.
5. **Appendices**: Technical logs, references, or additional data (e.g., request/response payloads).

2.2 Example Test Report Structure

For a pentest against **labs.hackxpert.com**:

- **Executive Summary**: Summarize objectives, highlight total discovered vulnerabilities (e.g., 3 critical, 5 high).
- **Methodology**: Mention grey-box approach, OWASP guidelines.
- **Findings**:
    - **SQL Injection (Critical)** at `/api/v1/users`. Evidence includes query manipulation retrieving all records.
    - **Cross-Site Scripting (High)** at `/api/v1/comments`. Show the injected script payload.
    - **Insecure Direct Object Reference (Medium)** at `/api/v1/files/{fileId}`. Show unauthorized file access.
- **Recommendations**: Input validation with prepared statements, output encoding to prevent script injection, and strong authorization checks for file endpoints.
- **Appendices**: Full logs, testing screenshots, references to library documentation.

A strong test report not only **exposes** vulnerabilities, but also **guides** the development or security teams in **fixing** them.

---

3. The Test Debrief Meeting

3.1 Purpose and Objectives

A **Test Debrief Meeting** brings together pentesters, developers, security teams, and relevant stakeholders to:

- **Review** the findings in detail.
- **Discuss remediation strategies** and timelines.
- **Extract lessons learned** to improve future tests.
- **Plan** additional security steps or retests.

3.2 Key Participants

- **Pentest Team**: Presents the report and clarifies technical details.
- **Security Team**: Evaluates remediation feasibility and risk considerations.
- **Development Team**: Learns about code or configuration fixes, clarifies implementation.
- **Stakeholders**: Executive sponsors or project managers who manage resources and deadlines.

3.3 Typical Agenda

1. **Welcome & Introduction**: State the meeting goals, introduce participants.
2. **Presentation of Findings**: Summarize high-impact vulnerabilities (e.g., SQL Injection, XSS).
3. **Remediation Discussion**: Propose fixes, gather feedback on feasibility and risk.
4. **Lessons Learned**: Assess the test plan's coverage; identify missed areas or new insights.
5. **Planning for Next Steps**: Set schedules for patching, retesting, and future pentests.
6. **Closing Remarks**: Confirm action items and responsibilities.

When well-structured, the Test Debrief Meeting encourages **clear communication** among all roles, ensuring the discovered issues are understood, prioritized, and resolved.

Chapter 6: API Firewalls – Essentials, Installation, and Bypass Techniques

1. Introduction to API Firewalls

1.1 What Is an API Firewall?

 The XSS Rat - CAPIE - Certified API Hacking Expert - V1.0

An **API firewall** is a security layer that inspects incoming and outgoing traffic to an Application Programming Interface (API). Similar to a traditional web application firewall (WAF), its purpose is to **block malicious requests**, detect abnormal patterns, and **enforce security policies** at the API layer. Unlike general WAFs, API firewalls typically provide deeper inspection of **REST**, **SOAP**, and **GraphQL** traffic, allowing more granular rules.

1.2 Why API Firewalls Matter

Modern APIs handle critical business logic and sensitive data. Consequently, they are prime targets for attacks involving:

- **Injection** (SQL, NoSQL, command, etc.)
- **Broken authentication or authorization**
- **Data exposure**
- **Business logic exploitation**

An API firewall operates as the “first line of defense,” filtering out suspicious traffic or requests violating specified constraints—such as malformed JSON or unauthorized HTTP methods. By analyzing each request at the **application layer**, an API firewall can enforce schema validations, rate limits, and even detect advanced threats, thereby minimizing risk.

---

2. Installing Your Own API Firewall

2.1 Prerequisites

Before deploying an API firewall, gather:

- **Documentation** on your APIs (OpenAPI specs, Postman collections).
- **Infrastructure details** (which servers, containers, or microservices host the APIs).
- **Security policies** or compliance requirements (e.g., PCI-DSS, HIPAA).

2.2 Common Deployment Models

1. **Reverse Proxy**: The firewall sits in front of your API, receiving incoming requests first and then forwarding them to the backend if they pass inspection.

2. **Sidecar/Service Mesh**: In microservices environments, the firewall logic can run alongside each service (e.g., using Envoy or Istio).

3. **Inline Appliance**: Dedicated hardware or VM-based solutions typically used by large enterprises.

2.3 Configuration Essentials

- **Rules and Signatures**: Enable out-of-the-box protections for known attacks (e.g., injection patterns), then tailor them to your API's unique endpoints.

- **Schema Validation**: Apply strict JSON schema or XML schema checks to ensure only valid request structures are accepted.

- **Rate Limiting**: Enforce request-per-second thresholds to block brute-force or DoS attempts.

- **Authentication Checks**: Optionally integrate with OAuth, JWT verification, or API key validation.

- **Logging**: Record all blocked or suspicious requests for later analysis.

2.4 Testing the Installation

After installation:

1. **Send valid requests** to confirm the API works as expected behind the firewall.

2. **Try malicious payloads** (e.g., SQL injection strings, overly large requests) to verify the firewall blocks them.

3. **Review logs** to ensure you have clarity around which requests are accepted, rejected, or flagged for follow-up.

---

3. Bypassing an API Firewall (Ethical Perspective)

3.1 Ethical Disclaimer

Any discussion on bypassing security measures must be conducted **responsibly**. Engaging in firewall evasion should occur **only** with explicit authorization, such as:

- **Penetration tests** sanctioned by the organization.
- **Bug bounty programs** where the business invites ethical hackers to find vulnerabilities.

The goal is always to **improve** defenses, not to exploit them for harm.

3.2 Common Bypass Techniques

1. **Input Validation Evasion**
    - **Description**: Attackers use encoding (URL encoding, Base64, double-encoding) or unusual character sets to pass malicious payloads undetected.
    - **Example**: Obscuring `<script>` tags with partial HTML encoding like `<script>`.
    - **Mitigation**: Normalize and decode inputs before applying validation rules, ensuring multiple layers of inspection.
2. **HTTP Verb Tampering**
    - **Description**: Exploiting endpoints that accept unexpected HTTP methods (PUT, PATCH, DELETE) when they're meant to only allow GET or POST.
    - **Example**: Attempting `curl -X DELETE https://api.example.com/users/12345` to remove resources if the firewall doesn't filter this method.
    - **Mitigation**: Strictly define and allow only necessary HTTP methods. Monitor unusual method usage.
3. **Header Manipulation**
    - **Description**: Attackers alter or add **HTTP headers** to confuse or override firewall logic.

- **Example**: Setting `X-Original-URL` or `X-Forwarded-For` to trick upstream routing, or smuggling malicious data in lesser-known headers.
- **Mitigation**: Validate header names and values, discard unknown or unexpected headers, and adopt zero-trust principles in proxy configurations.

4. **Advanced Bypass Tactics**
   - **Request Smuggling**: Crafting requests such that the firewall and backend parse them differently, leading to partial validation or incorrect routing.
   - **Protocol Downgrade**: Trying older protocols or subprotocols not covered by the firewall's rules.
   - **Multipart Mischief**: Hiding malicious payloads in unconventional multipart/form-data boundaries.

3.3 Real-World Examples

- **Encoding Bypasses**: Attackers used double-encoded SQL injection payloads to slip past basic WAF filters.
- **Smuggling Attacks**: High-profile breach where request smuggling led to partial checks from the firewall, ultimately exposing user sessions.

3.4 Staying Ahead

- **Continuous Updates**: Firewalls should be patched regularly to handle new bypass techniques.
- **Regular Pentests**: Encourage authorized testers to evaluate your firewall's resilience.
- **Security Culture**: Share findings across dev, ops, and security teams so everyone can detect new bypass methods early.

Chapter XTRA 001: HTTP Request Methods

In RESTful and other web-based APIs, the HTTP request method (sometimes called the "verb") conveys the desired action on a resource. Understanding each method's semantics, idempotence, and safety is key to designing and consuming APIs correctly.

 The XSS Rat - CAPIE - Certified API Hacking Expert - V1.0

1. GET

- **Purpose**: Retrieve a representation of a resource.
- **Safety**: Safe (does not modify server state).
- **Idempotence**: Idempotent (multiple identical requests have the same effect as one).
- **Use Case**: Fetching user profiles, listing items.

Example:

http

```
GET /api/users/42 HTTP/1.1
Host: api.example.com
Accept: application/json
```

-

2. POST

- **Purpose**: Create a new resource or trigger server-side processing.
- **Safety**: Not safe (can change state).
- **Idempotence**: Not idempotent by default (multiple requests may create multiple resources).
- **Use Case**: Submitting forms, uploading files, initiating complex operations.

Example:

http

```
POST /api/articles HTTP/1.1
Content-Type: application/json

{ "title": "Hello World", "body": "..." }
```

-

3. PUT

- **Purpose**: Replace an existing resource or create it if it does not exist.
- **Safety**: Not safe.
- **Idempotence**: Idempotent (repeating the same PUT yields the same state).
- **Use Case**: Updating entire resource representations.

Example:

http

```
PUT /api/users/42 HTTP/1.1
Content-Type: application/json

{ "username": "jdoe", "email": "jdoe@example.com" }
```

-

4. PATCH

- **Purpose**: Apply partial modifications to a resource.
- **Safety**: Not safe.
- **Idempotence**: Can be idempotent if designed that way (depends on patch document).
- **Use Case**: Updating one or two fields (e.g., changing a password, toggling a flag).

Example:

http

```
PATCH /api/users/42 HTTP/1.1
Content-Type: application/json-patch+json
```

```
[ { "op": "replace", "path": "/email", "value": "new@example.com" } ]
```

-

5. DELETE

- **Purpose**: Remove a resource.
- **Safety**: Not safe.
- **Idempotence**: Idempotent (deleting a non-existent resource still yields the same state).
- **Use Case**: Deleting user accounts, removing items from a cart.

Example:

http

```
DELETE /api/users/42 HTTP/1.1
```

-

6. HEAD

- **Purpose**: Like GET, but only fetches headers (no body).
- **Safety**: Safe.
- **Idempotence**: Idempotent.
- **Use Case**: Checking resource existence or metadata (e.g., Content-Length) before downloading.

Example:

http

```
HEAD /api/files/report.pdf HTTP/1.1
```

-

7. OPTIONS

- **Purpose**: Describe available communication options for the target resource (including allowed methods and CORS policies).
- **Safety**: Safe.
- **Idempotence**: Idempotent.
- **Use Case**: Preflight CORS requests, discovering API capabilities.

Example:

http

```
OPTIONS /api/users HTTP/1.1
```

-

8. Other Methods

- **CONNECT**: Establishes a tunnel (e.g., for HTTPS through a proxy).
- **TRACE**: Echoes the received request for diagnostic purposes.
- **WebDAV Extensions**: PROPFIND, PROPPATCH, MKCOL, etc., for remote content management.

Choosing the Right Method

1. **Semantics**: Pick the method whose intended action matches your operation (e.g., use DELETE for removals, not POST).
2. **Idempotence Guarantees**: If clients must safely retry without risk (e.g., in unreliable networks), prefer idempotent methods (GET, PUT, DELETE, HEAD, OPTIONS).
3. **Payload Size & Caching**: GET requests can be cached; bodies in GET aren't standardized—send large payloads with POST or PUT.

4. **Security**: Ensure side-effects only occur on non-safe methods; use CSRF protection on state-changing endpoints (POST, PUT, PATCH, DELETE).

By adhering to HTTP method semantics, you build APIs that are intuitive, interoperable, and resilient

 The XSS Rat - CAPIE - Certified API Hacking Expert - V1.0

Chapter XTRA 002: CVSS Scoring and Calculation

The Common Vulnerability Scoring System (CVSS) is a standardized framework for assessing the severity of software vulnerabilities. A consistent CVSS score helps security teams prioritize remediation efforts and communicate risk to stakeholders. This chapter explains CVSS v3.1 metrics, demonstrates the scoring formulas, and provides a worked example.

---

1 Overview of CVSS

CVSS produces a numerical score ranging from 0.0 to 10.0, along with qualitative severity ratings (None, Low, Medium, High, Critical). The score aggregates three metric groups:

1. **Base Metrics** (immutable properties of a vulnerability)
2. **Temporal Metrics** (characteristics that evolve over time)
3. **Environmental Metrics** (organization-specific factors)

The **Base Score** is most commonly used for initial prioritization. Temporal and Environmental scores refine that baseline.

---

2 Base Metrics

The Base Metric group comprises two subcategories:

7.2.1 Exploitability Metrics

- **Attack Vector (AV)**: How the vulnerability is exploited (Network = 0.85, Adjacent = 0.62, Local = 0.55, Physical = 0.2).
- **Attack Complexity (AC)**: Conditions beyond attacker control (Low = 0.77, High = 0.44).
- **Privileges Required (PR)**: Privileges needed (None, Low, High, with distinct values depending on Scope).
- **User Interaction (UI)**: Whether a user must perform an action (None = 0.85, Required = 0.62).
- **Scope (S)**: Whether a vulnerability affects resources beyond its security scope (Unchanged or Changed).

7.2.2 Impact Metrics

 The XSS Rat - CAPIE - Certified API Hacking Expert - V1.0

- **Confidentiality (C)**, **Integrity (I)**, **Availability (A)**: Each rated None (0.0), Low (0.22), High (0.56).

Base Score is calculated via a formula combining Exploitability and Impact:

```
Unset
Impact = 1 - [(1 - C) × (1 - I) × (1 - A)]

If Scope = Unchanged:

    BaseScore = round_up(min((Impact + Exploitability), 10))

Else (Scope = Changed):

    BaseScore = round_up(min(1.08 × (Impact + Exploitability),
10))
```

where Exploitability = 8.22 × AV × AC × PR × UI.

---

3 Temporal Metrics

These adjust the Base Score based on current exploitability and remediation:

- **Exploit Code Maturity (E)**: Not Defined, High, Functional, Proof-of-Concept, Unproven.
- **Remediation Level (RL)**: Not Defined, Unavailable, Workaround, Temporary Fix, Official Fix.
- **Report Confidence (RC)**: Not Defined, Confirmed, Reasonable, Unknown.

Temporal Score = Base Score × E × RL × RC (all metric values ≤ 1.0).

---

4 Environmental Metrics

Environmental metrics tailor the score to an organization's context:

- **Modified Base Metrics**: Adjust any Base metric (e.g., Modified Attack Vector).
- **Security Requirements (CR, IR, AR)**: Importance of Confidentiality, Integrity, Availability (Low = 0.5, Medium = 1.0, High = 1.5).

 The XSS Rat - CAPIE - Certified API Hacking Expert - V1.0

Steps:

1. Replace Base metrics with Modified values.
2. Compute **Modified Impact**:

```
Unset
```

3.
```
ModifiedImpact = min(1 - [(1 - C × CR) × (1 - I × IR) × (1 - A × AR)], 0.915)
```

4. Compute **Modified Exploitability** with modified Exploitability metrics.
5. **Environmental Score** = round_up((Impact + Exploitability) × E × RL × RC).

---

5 Worked Example

A network-accessible SQL injection yields:

- AV = Network (0.85)
- AC = Low (0.77)
- PR = None (0.85)
- UI = None (0.85)
- Scope = Unchanged
- C = High (0.56), I = High (0.56), A = High (0.56)

1. **Exploitability** = 8.22 × 0.85 × 0.77 × 0.85 × 0.85 ≈ 3.74
2. **Impact** = 1 - (0.44 × 0.44 × 0.44) ≈ 0.915
3. **Base Score** = min(3.74 + 0.915, 10) = 4.655 → rounded up = 4.7 → **Medium**

If a functional exploit exists (E = 0.95) and no official fix (RL = 0.96), Report Confidence = Reasonable (RC = 0.96):

- **Temporal Score** = 4.7 × 0.95 × 0.96 × 0.96 ≈ 4.10

An environment that considers confidentiality critical (CR = High = 1.5) but integrity and availability medium (1.0) would slightly raise the score when recomputed via Modified metrics.

---

6 Best Practices

- Always document chosen metric values and justification.

- Recompute scores when exploit code or patches emerge.
- Use environmental scoring for high-value assets.
- Leverage calculators (e.g., the official CVSS website) to avoid manual errors.

By mastering CVSS calculation, security teams can prioritize fixes objectively and communicate risk effectively.