Hello everyone, my name is **Pritish Ravindra Borkar** , and welcome to my cybersecurity capstone demonstration based on Damn Vulnerable Web Application, also known as DVWA. In this video, I will walk through a structured, real-world web penetration testing process covering three major vulnerabilities: **SQL Injection**, **Stored Cross-Site Scripting**, and **Cross-Site Request Forgery**, all inside an isolated testing environment. I will also simulate a short **Incident Response** process to show how organizations should react to such attacks. This project is completed as part of Task-5, dedicated to hands-on exploitation and security validation on a vulnerable web service.

Let me start with the foundation of my environment. Everything shown here is performed entirely inside a **Kali Linux virtual machine**, so there is no risk to production systems. I am running Apache as the web server and MariaDB as the database. DVWA is installed locally and accessible through the URL: **http://localhost/**. I used the default DVWA login credentials: *username admin* and *password password*. This keeps the demonstration simple and easy to replicate on any academic lab setup.

As part of the initial setup, I ensured that Apache and MariaDB services were running correctly, and that the DVWA configuration file was updated with the correct database username, password, and host entries. After that, I opened the DVWA setup page in the browser, clicked **Create / Reset Database**, and confirmed that the required tables were successfully created inside the DVWA database. All evidence in this video is also documented inside my project folder and GitHub repository.

Once DVWA was fully operational, I logged in and navigated to the **DVWA Security** tab. Since this video focuses on learning exploitation techniques, I temporarily set security to **Low**. This lowers protections intentionally to help demonstrate how vulnerabilities originate and behave in unsafe applications. Later in the video, we will raise defenses and highlight improvements as part of the incident response and remediation stage.

Now, let's begin our first major vulnerability: **SQL Injection**.

SQL Injection, often abbreviated as SQLi, is one of the most dangerous and common vulnerabilities in web applications. It happens when websites **fail to validate user input**, and the input gets directly executed as a database command. That means, if a text box expects an ID number, but instead receives malicious SQL code such as ' OR '1'='1', the website might reveal hidden data or even allow modification of the entire database.

In DVWA, the SQL Injection test page accepts an "ID" parameter. Normally, this value fetches a single user record. So first, I type the ID as **1** and confirm that the application shows information about the first user. But then I submit **1'** with a single apostrophe. This breaks the original SQL query, proving that **no input validation** exists. This is the first major clue that the page is vulnerable.

To escalate further, I type:
1 OR 1=1

This condition is logically true for every row in the table. As expected, the application suddenly displays **multiple user records** where only a single result should be allowed. This successfully demonstrates SQL Injection through **manual exploitation**.

Now, to move into real-world automation, I use a professional penetration testing tool named **sqlmap**. sqlmap is designed to test database-driven web applications and automatically extract

sensitive information. I run the sqlmap command with the URL of the injection-point and specify that the parameter id is vulnerable.

sqlmap analyzes server responses, detects the vulnerability, starts fingerprinting the database type, and then requests permission to dump database content. In this demonstration, sqlmap successfully enumerates the **dvwa** database and retrieves tables including **users**. The extracted contents reveal usernames and password hashes. Even though these hashes are stored, they are still sensitive information — once an attacker has them, cracking is only a matter of time, depending on hash strength.

This exploitation clearly illustrates the **data-theft risk** associated with SQLi. Attackers can potentially escalate to dump secret configuration tables, modify stored data, or even escalate privileges to gain system-level control. Therefore, SQL Injection holds a **High severity rating**.

Mitigation is straightforward but must be implemented consistently. Developers should always use **parameterized queries** or **prepared statements** so that user input is treated strictly as data, not as part of the command. Strong input validation should escape or reject harmful characters. Least-privilege access should be set for the database user, meaning the web application should not use an administrator-level account for database connections. Defensive filtering through WAFs — Web Application Firewalls — can add additional protection. And finally, regular security scanning tools like sqlmap should be part of security testing cycles.

With that, real-world SQL Injection exploitation has been demonstrated.

Next, let's move to our second major vulnerability: **Stored Cross-Site Scripting**, commonly known as Stored XSS.

While SQL Injection attacks a system from the server-side, XSS attacks target users in their browsers. Stored XSS occurs when an application **stores user input** and **later displays it to others without sanitizing it**. This allows attackers to store malicious JavaScript inside the target website. When another user views the page, the JavaScript executes silently in their browser. This may steal cookies, redirect visitors to phishing pages, or perform unauthorized actions.

In DVWA, the page where users can leave a message is vulnerable. Instead of typing a normal message, I input a script tag such as:

<script>alert('XSS_TEST')</script>

After submitting the form, the page reloads and stores this input inside the database. Now, when any user — including the attacker — visits the same page again, the script executes, displaying a popup box containing the text "**XSS_TEST**". This simple alert proves execution of arbitrary JavaScript code.

In the real world, attackers would replace the alert with malicious logic. For example, they could send cookies off to a remote system to hijack sessions. They could deface websites, install crypto-mining scripts, or automatically perform administrative actions using the victim’s session.

To protect against Stored XSS, developers need to ensure strict **output encoding**. This means that input like <script> should display literally as text, not as executable code. Implementing a strong **Content Security Policy (CSP)** helps block unauthorized scripts. Cookies should be flagged with **HttpOnly** and **Secure** attributes, preventing JavaScript access and requiring HTTPS. Finally, input validation using allow-lists should prevent harmful symbols in user-generated fields.

Thus, Stored XSS is classified as a **Medium severity vulnerability** but can reach high severity depending on its target and the victim's privileges.

Our third exploitation is **Cross-Site Request Forgery**, known as **CSRF**.

This vulnerability tricks an authenticated user into performing an action they did not intend, simply by visiting a malicious webpage while they are logged in to their account on another site. The site receives a valid request since the browser automatically sends cookies, allowing the attacker to change passwords, send messages, or perform admin-level actions.

Inside DVWA, the password change feature does not validate whether the request originated legitimately from the application. To exploit this flaw, I create a small HTML page named csrf_attack.html and place it inside the server's webroot. This page contains a hidden form that automatically submits a password change request to DVWA using JavaScript.

Then I log into DVWA as the admin user using the original credentials. Now, simply opening that malicious csrf_attack.html causes the browser to perform an unwanted action: the admin's password gets changed silently to **hacked123**, without the admin clicking any button or verifying the action.

This kind of attack can completely compromise a user's account, especially privileged accounts. Many well-known historical data breaches leveraged forms of CSRF.

Preventing CSRF is based on authenticating intent. Websites must generate random **anti-CSRF tokens** that are tied to each user session and must be included in every sensitive request. If a malicious page submits a request, lacking this token, the server rejects it. Configuring cookies with the **SameSite** attribute reduces risk by blocking cookie sharing in cross-origin requests. Particularly sensitive operations like password changes should require users to re-enter credentials.

Thus, CSRF is classified as **Medium severity** but can escalate to critical depending on which features it affects.

Now that exploitation has been demonstrated, let's shift to the security professional's other job: **Incident Response**.

Incident Response is a structured approach a company follows when suspicious activities are detected on its systems. In this simulation, I act as the defender responding to the detected attacks.

The first step is **Detection**. I inspect Apache access logs and identify malicious patterns such as SQL keywords, script tags, or suspicious POST requests. These log entries confirm that both SQL Injection and XSS payloads were executed, as well as unauthorized password resets.

The second step is **Containment**. If this were a real active attack, I would temporarily cut off the attacker's access. In my demonstration, I show a firewall rule using **iptables**, blocking the attacker's IP address entirely. This prevents ongoing automated exploitation.

The third step is **Eradication**. Here, I rotate compromised credentials, update the web application configuration, fix vulnerable code, and remove any malicious artifacts such as the CSRF attack page used in the demo. This is also where patches, secure coding upgrades, and input validation mechanisms must be implemented.

The fourth step is **Recovery**. The goal here is to restore normal operations while maintaining security. I increase DVWA Security level from Low to High, reflecting security upgrades and confirming that exploitation attempts now fail. Logs should continue to be monitored for suspicious behavior.

Finally, we record **Lessons Learned**. These include: always validate and sanitize user input, escape output to protect user browsers, enforce anti-CSRF tokens, use database accounts with limited privileges, enable secure cookie flags, and integrate monitoring tools to detect intrusions early.

Incident Response strengthens an organization's ability to protect and recover from attacks, completing the cycle of defensive cybersecurity.

Before we conclude, I would like to highlight the **ethical and professional standards** followed in this project. All exploitation was carried out **only on intentionally vulnerable systems inside a controlled laboratory environment**. These techniques must *never* be used against real systems without legal authorization. Cybersecurity is always built upon ethical responsibility and respect for data privacy.

To summarize the capstone:

✅ DVWA was installed and configured successfully
✅ SQL Injection was exploited manually and using sqlmap, proving database disclosure
✅ Stored XSS was triggered to demonstrate cross-browser execution of script payloads
✅ CSRF was executed to show unauthorized password change using hidden submissions
✅ A practical Incident Response sequence was performed: detect, contain, eradicate, recover
✅ Remediation steps were discussed for each vulnerability
✅ All findings, commands, and evidence files were documented and published safely in a GitHub repository

This project has deepened my understanding of **web application security**, both from the attacker and defender perspective. I gained practical experience with vulnerability exploitation, automation tools, secure configuration, remediation planning, and cybersecurity reporting — all skills essential for my future career in the cybersecurity industry.

Thank you very much for watching my demonstration. I appreciate your time and interest. If you would like to learn more about the techniques displayed here, collaborate on cybersecurity research, or provide feedback on my work, please feel free to reach out or visit my GitHub repository for the full project documentation.