VotD: SQL Injection

Attack Pattern: See CAPEC-66 (https://capec.mitre.org/data/definitions/66.html)

Weakness/Vulnerability: See CWE-89 (https://cwe.mitre.org/data/definitions/89.html)

Examples: sql-injection.zip (https://uncw.instructure.com/courses/16302/files/351744/download?wrap=1)

- Unzip contents to a directory. The zip file contains a Java class demonstrating the vulnerability and an in-memory SQL database called H2.
- First run `make` from the terminal to compile the code.
- The `SQLInjection.java` class has two methods that mimic checking if a username/password combination is in the "Users" table.
    - `auth` - does not sanitize the user input. It simply puts in whatever the user typed into the SQL query.
    - `safe` - uses a PreparedStatement. The PreparedStatement automatically sanitized any inputs corresponding to `?`s in the query string.
- Run `make run` in the terminal. This will prompt you to type a username and password. The only user in the system is user=Bobby, password=tables. Any other username and password combination should fail.
- Now let's attack. Type `make run` again and type in exactly the following for the username: `bobby\n' or TRUE --`. Type in whatever you want for the password. You should now be authenticated.
    - Looking at the Java code, we can see that this results in executing the SQL query `SELECT * FROM Users WHERE Username='bobby\n' or TRUE --' AND Password='<whatever>'`.
    - The -- indicates the beginning of an inline comment in SQL, so the -- and everything after it will be ignored. So, the database effectively executes the query `SELECT * FROM Users WHERE Username='bobby\n' or TRUE`
    - The "or TRUE" dutifully selects ALL the rows in the table. The Java code checks to see if any rows were returned by the database and, if so, logs the user in.
- Now edit the line `System.out.println(auth(user, password, conn));` to call the `safe` method instead of the `auth` method.
- Try the attack again. You will be unsuccesful as the PreparedStatement will strip out all special SQL syntax characters.

DVWA and Fuzzer:

- Now that you've had a sense for how SQL Injection works in general, try it on the SQL Injection page of the DVWA (https://uncw.instructure.com/courses/16302/pages/activity-vulnerable-web-application-setup) . Remeber to the set the security level to Low.
- SQL Tutorial (https://www.w3schools.com/sql/sql_wildcards.asp) - Everything up through SQL Wildcards will be useful. Also jump to the very last tutorial: SQL Comments.
- Our original attack won't work because MySQL doesn't understand the 'or TRUE' part. Instead, let's give it a statement that evaluates to TRUE, e.g., `bobby\n' or '0'='0`.

- Click the 'View Source' button to piece together what's happening. Feel free to 'View Help' as well.
- Try elevating the security level. Can you hack it?
- Add any successful hacks to your Web Application Vulnerabilities google doc you created when playing with XSS.
- Now give it a try on the SQL Injection (Blind) page. Many of the same attacks will work, only you won't get the same level of feedback from the server on what went wrong. This can make life more difficult for the fuzzer because it is less obvious if an attack failed or succeeded.

Mitigations:

- The **only** acceptable mitigation are properly-used prepared statements. No string concatenation to form SQL queries should be used. Don't turn user input into executable code without proper sanitization.
- All programming languages support Prepared Statements or have libraries that enable them for SQL databases.
- Escaping characters has proven to be a poor substitute, as changing character sets (e.g., from US to Russian) makes this a moving target and quite hard.

Notes

- One of the more prevalent and costly vulnerabilities in history.
- **ARS Technica** (http://arstechnica.com/information-technology/2016/10/how-security-flaws-work-sql-injection/) has a very nice writeup of the history and consequences of SQL-injection
- See this **classic XKCD** (http://xkcd.com/327/).