Intelligent Cyber-Security
Case studies: Injection attacks

Dr. Alexandru ARCHIP

Gheorghe Asachi Technical University of Iasi

Artificial Intelligence (MSc., second year) – 2025 – 2026

alexandru.archip@academic.tuiasi.ro

In brief...

1. The attack pattern

2. SQL Injection
   - Weaknesses and vulnerabilities
   - Classic mitigation techniques
   - AI/ML defensive approaches

3. Cross-Site Scripting
   - Weaknesses and vulnerabilities
   - Classic mitigation techniques
   - AI/ML defensive approaches

4. Shell command injection
   - Weaknesses and vulnerabilities
   - Classic mitigation techniques
   - AI/ML defensive approaches

Outline

Definitions and examples

Code injection [1]

Code injection is an attack type which consists of injecting code that is interpreted/executed by the target application.

Effect: cause target application to behave out-of-scope with respect to its intended functionality.

Command injection [1]

Command injection is an attack in which the goal is the execution of arbitrary commands on the host operating system.

Effect: arbitrary shell commands/applications are executed on the target host.

Shared vulnerability

Improper input sanitisation: lack of input data validation (e.g., allowed character set/allowed values), lack of input data format validation, lack input size validation.

Definitions and examples

SQL Injection assume <user, pass> are the input fields that should supply user name and user password for an authentication step (Web form input fields);

sample code for sql query:

```
select * from authData where username=' + $user + '
and password=' + $pass + '
```

sample attack input (pass field): `' OR '1'='1'; --`

effect: authenticate as any user!

Definitions and examples

Cross-Site Scripting (XSS) assume a web page accepts URI query parameters for different scenarios;

sample URI: http://example.com/articles?id={articleID}

sample attack input (adapted from [2]):
`id=%3Cscript%3Ealert%281%29%3C%2Fscript%3E`

effect: client browser displays an alert box with the value 1.

Definitions and examples

Shell command injection a web application executes a C application to display the content of a file;

sample vulnerable C app: see Listing 1

sample attack input: `"; rm -rf /"`

effect: varies, but could recursively delete the root partition!

```
1 int main(int argc, char** argv) {
2     char cmd[CMD_MAX] = "/usr/bin/cat␣";
3     strcat(cmd, argv[1]); system(cmd);
4     return 0;
5 }
```

Listing 1: Vulnerable C example demonstrating command injection

Outline

Weaknesses and vulnerabilities

Potential weaknesses:

- **SQL** is a **declarative**, **domain-specific language**;
  - the formal grammar of SQL is defined by the SQL standard;
  - database management systems (DBMS) (e.g. MariaDB, MySQL) may implement the standard differently;
  - different DBMS products may provide additional, more powerful constructs;
  - DBMS generally execute any query string that conforms to their supported SQL dialect.
- Ensuring **correct SQL syntax** and the **use of parameterised queries** is the responsibility of the client/application;
  - this enables unsafe application practices if developers concatenate untrusted input into queries.

Weaknesses and vulnerabilities

Vulnerabilities

- The main vulnerability is the **improper handling of user input**; SQL *executable* statements are **constructed directly** from user input:
  - use of string concatenation instead of parameterised queries;
  - insufficient validation of user input;
  - inadequate escaping of special characters.
- DBMS and host OS **configuration policies** may also introduce severe vulnerabilities:
  - excessive SQL user privileges for client/application interactions;
  - extensive DBMS functionalities that may permit execution of OS/shell commands;
  - inappropriate OS privileges for the account running the DBMS.

Mitigation and prevention techniques

Application level (developer's perspective) [3]:

- use access libraries that support prepared statements and parameterised queries;
- use properly constructed stored procedures (*only if stored procedures do not increase the attack surface*);
- validate and sanitise user input.

DBMS level (SQL server) [3]:

- disable unnecessary SQL features (e.g., `load_file()`-like functions should not be enabled by default);
- adopt the *least privilege* security principle for SQL accounts;
- restrict access to SQL `root` and application-specific `DB Admin` accounts;
- ensure errors forwarded to client applications contain minimal information, exposing only what is strictly required.

Host OS configurations:

- reduce host OS privileges for the DBMS account (e.g., limit file access rights, disable access to critical commands).

Classic detection techniques

- A firewall-like solution is deployed in front of the public-facing application (e.g., a Web Application Firewall placed before the web server hosting the protected application).
- Common approaches rely on parameter inspection stages:
  - normalise input — decode and/or canonicalise encoded data;
  - employ signature-based detection — inspect incoming requests for known SQL keywords embedded in parameters.
- Dynamic solutions include anomaly or behavioural control:
  - establish a baseline of legitimate request patterns;
  - flag and/or drop deviations.

AI/ML defensive approaches

- Most solutions focus on parameter analysis and attempt to develop classification models that detect anomalous or attack patterns.
- Common training datasets are derived from known attacks.
  - specialised sites such as Kaggle include various such datasets [4];
  - penetration-testing distributions (e.g. Kali Linux) include tools that allow simulations of SQL injection attacks (e.g. `sqlmap`); these tools may be used to generate new datasets.
- Practical model application usually follows two main directions:
  - embed the trained model within a WAF-like application;
  - use model results to dynamically update signature-based detection systems.

AI/ML defensive approaches – examples

Irungu *et al.* [5] tested three well-known classifiers: *Support Vector Machine* (SVM), *K-Nearest Neighbours* (KNN), and *Random Forest* (RF).

Features	statistical data such as query length and query length variations, special character frequencies, and known patterns (e.g. tautologies, UNION-type or piggyback patterns).
Training and validation	classic 70%/30% split.
Results	SVM achieved the highest accuracy — 98%;
	SVM achieved the highest weighted precision — 99%;
	SVM achieved the highest weighted recall — 99%.

AI/ML defensive approaches – examples

Wang *et al.* [6] address SQL Injection detection through semantic analysis of SQL queries. They evaluate the performance of an SVM classifier when the feature set is constructed using Word2Vec embeddings.

Features	Word2Vec (CBOW) embeddings computed over the *words* in each SQL query (a "word" is defined as any string between two consecutive white spaces)
Training and validation	classic 70%/30% split
Results	recall – 95.76%; precision – 97.83%; strength of the solution: improved detection of obfuscated or highly unstructured queries

AI/ML defensive approaches – examples

Gabriela Ursachi [7] proposed a novel approach to SQL Injection: she developed an ML-based application firewall for the protected DBMS server.

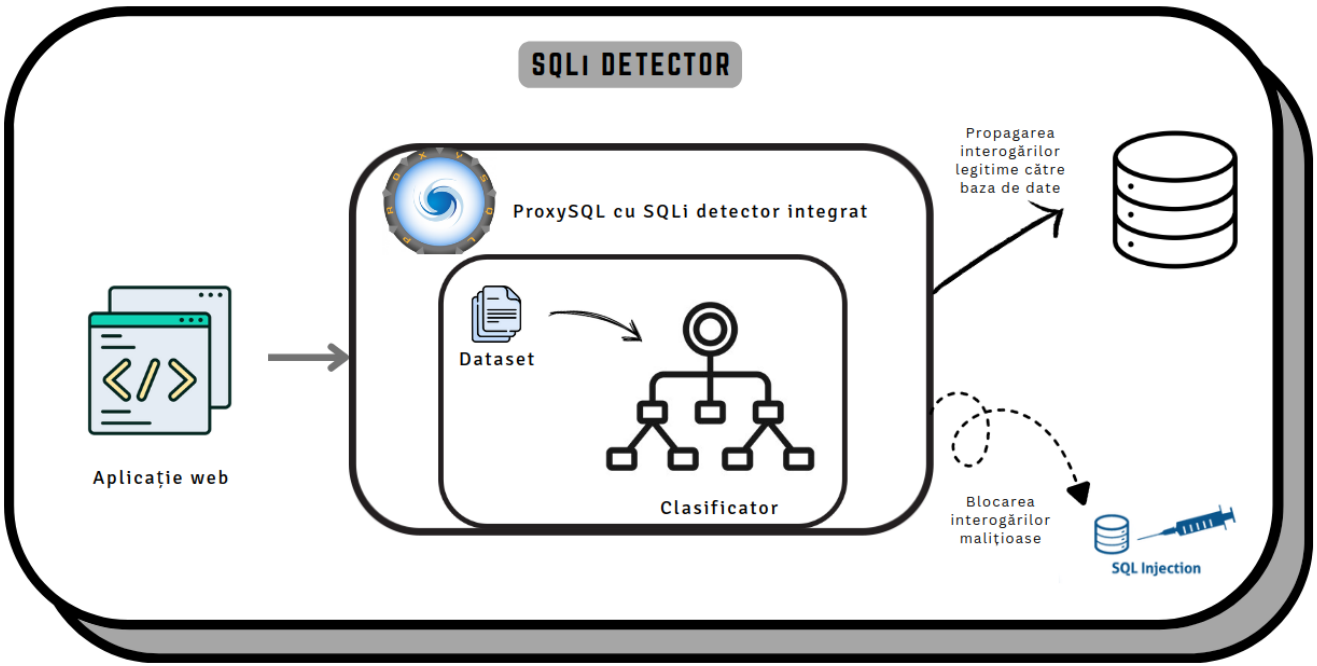

Figure 1: Application overview (image taken from [7])

AI/ML defensive approaches – examples

Classifier architecture stacking classifier using Logistic Regression, SVM, and a neural-network-based classifier.

Features query length; counts of OR, AND, UNION, and NULL; number of comments; number of empty-style lines; semicolon counts; number of hexadecimal values; number of character-encoding function calls; Shannon entropy of the query string; number of CONCAT() calls; presence of tautologies; presence of unbalanced quotes.

Training and validation k-fold cross-validation ($k = 5$) for individual classifiers; `sqlmap`-generated attacks for the overall solution.

Results 99.77% precision and recall in training and validation stages; 99.47% recall and 99.46% precision in the `sqlmap` testing stage.

Outline

Weaknesses and vulnerabilities

Potential Weaknesses

- **HTML** is **not a strict** markup language, having a **permissive** parsing model:
  - *HTML* syntax allows *javascript* components almost everywhere (e.g. in `<script>` tags or in tag attributes);
  - storage and rich formatting capabilities enable the dynamic embedding of raw HTML code within existing pages;
  - *browser* applications are implemented to accept malformed or mixed markup and to automatically correct/hide HTML components.
- **JavaScript** (JS) is a **first-class** programming language and is **executed by default**:
  - *browsers* are programmed to run Javascript code regardless of its source, provided that security mechanisms such as *Same-Origin Policy* (SOP) or *Content Security Policy* (CSP) are not configured;
  - JS is *inherently tied* to HTML: there is an API for each modifiable HTML element;
  - this significantly increases the attack surface!

Weaknesses and vulnerabilities

Vulnerabilities

- Websites and web pages often rely on **misconfigured security policies** (SOP/CORS or CSP):
  - allowing the execution of inline scripts;
  - exchanging sensitive cookies or HTTP headers without proper safeguards (e.g., `HttpOnly` or `SameOrigin`).
- **Improper handling of user input** generates additional vulnerabilities:
  - external or third-party content loaded without sufficient validation;
  - inadequate validation of user input.
- **Application developers** may inadvertently **increase** the attack surface:
  - use of insufficiently validated external or third-party JavaScript libraries;
  - acceptance of raw HTML or rich content features;
  - erroneous character set settings when changing scope or application context.

Types of XSS [8]

Reflected XSS the user input is immediately returned by the victim web application;

- it usually requires an initial request to be sent by the client's user agent, and is therefore observable on the server side;
- it affects only a single client (i.e., the one that performed the request).

Stored XSS the user input is stored on the target server (e.g., a comment on a discussion forum stored in a database);

- it is observable on the server side (at least during the initial delivery);
- it affects *all* clients requesting the altered web resource.

DOM-Based XSS the attack payload is executed directly by the client's user agent as a result of modifications to the page's DOM environment;

- it may not involve server interaction and therefore may not be observable on the server side;
- example: browser user agents do not usually forward the `#fragment` component of a URI.

Mitigation and prevention techniques [9]

Web application level (Reflected and Stored XSS):

- protect application variables: validation and sanitisation techniques *must* be applied to all application variables (most modern JavaScript frameworks implement this step natively);
- sanitise HTML results and encode output:

  HTML tags — use HTML entity encoding (e.g. the `textContent` attribute);

  HTML attributes — use standard HTML syntax (i.e., attribute values are enclosed in " or '); use `setAttribute()`;

  JavaScript elements — encode variable values using the `\xHH` format (e.g. via the `EncodeForJavaScript()` function);

  CSS elements — use variables only for CSS properties (e.g. `style.property = value`);
- *do not alter a sanitised HTML output*;
- apply correct encoding for URI components: first encode the URI variables, then apply HTML encoding techniques over the resulting URI.

Mitigation and prevention techniques [10]

DOM-based XSS:

- apply HTML escaping techniques, then JavaScript escaping when generating HTML content;
- use JavaScript encoding before inserting untrusted data into HTML attributes;
- validate data before adding it to event handlers;
- use JavaScript encoding before inserting untrusted data into CSS attributes;
- first encode URIs, then apply JavaScript encoding when working with web links;
- use safe JavaScript functions or properties when modifying HTML (e.g. `innerText` or `textContent`).

Classic detection techniques

Non-ML/AI detection and prevention techniques are similar to those used against SQL Injection attacks.

- Web Application Firewalls are typically deployed in front of the web-facing application;
- the focus is on URI and body parameter inspection:
  - input is decoded and normalised;
  - values are then inspected for known XSS patterns, calls to known vulnerable JavaScript functions, syntactic anomalies, and known exploits;
  - most approaches employ signature-based detection methods.
- it is worth noting that this type of solution is generally effective only against *Reflected* and *Stored XSS* attempts;
- mitigation of *DOM-based XSS* relies heavily on the client-side techniques mentioned previously and on browser security configurations.

AI/ML defensive approaches

Kaur *et al.* [11] (2023) published a comprehensive review on ML/AI applications for XSS detection and prevention:

- a wide variety of ML algorithms have been evaluated on XSS attack scenarios, including decision trees (Random Forest, ADTree, J48), support vector machines, KNN, and Naïve Bayes;
- most approaches focus on syntactic features and known vulnerable JavaScript functions;
  - the main drawback lies in the quality of the datasets — the vast majority are built on *Reconnaissance*-phase data (see CKC in Lecture 1);
- as with other code injection contexts, integrating such models into real-world WAF-like applications can be time-consuming if not carefully managed.

AI/ML defensive approaches

Babaey and Ravindran [12] (2025) describe an interesting approach:

- they studied the potential of Generative AI (GenAI) (GPT-4o and Google Gemini Pro) in synthesising new, syntactically valid and highly obfuscated XSS payloads, with a focus on payloads that evaded WAF rules;
- successful payloads were then clustered using TF-IDF vectorisation with hierarchical agglomerative clustering, and sequence-based similarity with DBSCAN;
- features extracted from the clusters were fed back into the GenAI tools to derive new detection rules for the ModSecurity plugin for the Apache web server;
- **results**: 15 new rules, providing an estimated 86% increase in detection effectiveness for the studied WAF.

AI/ML defensive approaches

Maria-Gabriela Fodor [13] proposed a novel approach to XSS detection and mitigation, employing stacking classifiers integrated into an HTTP forwarder application.

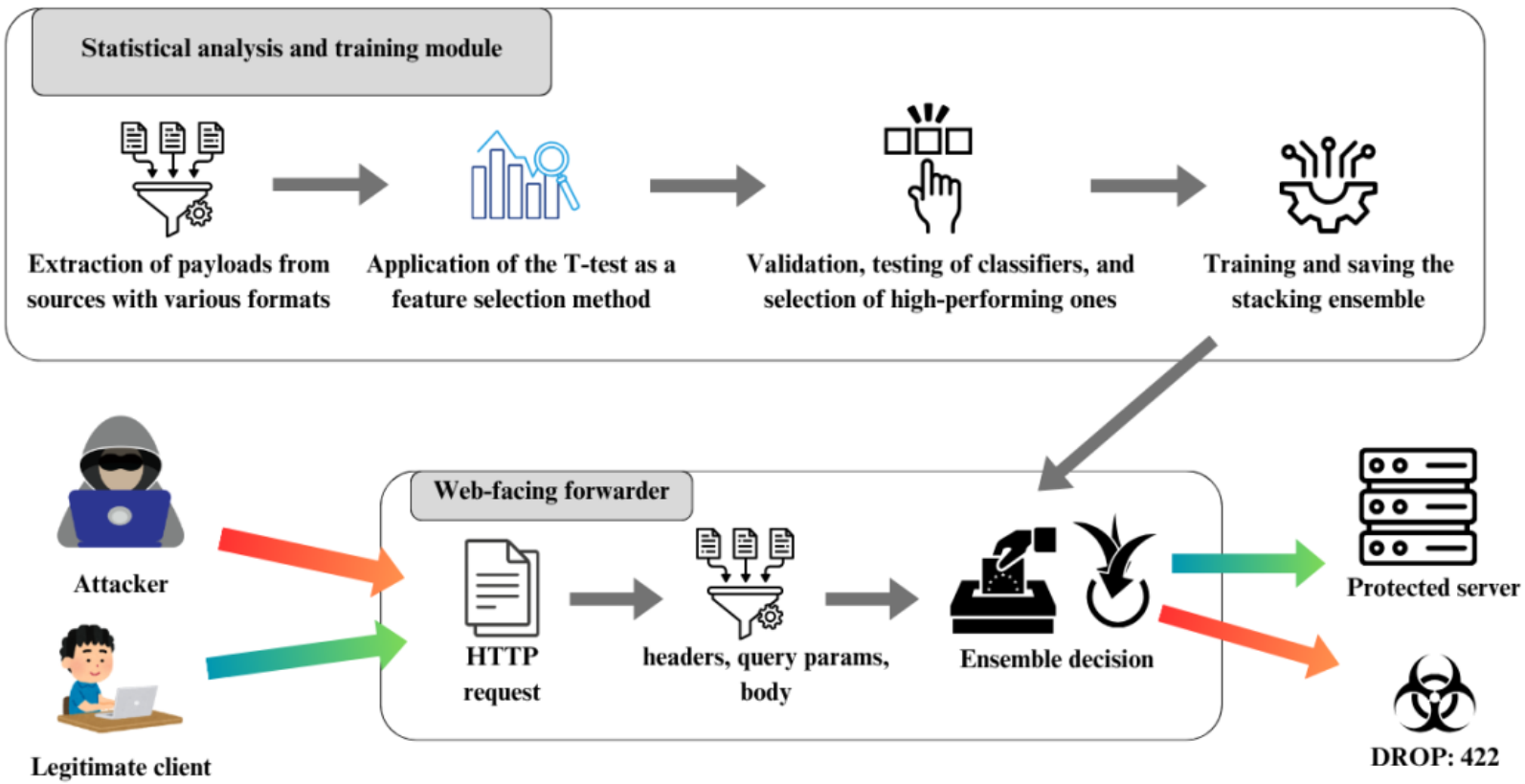

Figure 2: Application overview (image adapted by Maria from [13])

AI/ML defensive approaches

Classifier architecture Logistic Regression combined with Random Forest and Multinomial Naive Bayes;

Features validated using the t-statistical test and the Pearson correlation coefficient;

include a `redirect` counter as a feature (uncommon among most classifiers);

Training and validation k-fold cross-validation ($k = 5$) for all classifiers during the training phase;

XSStrike and XSSer were used to simulate attacks for the overall solution;

Results 96.58% recall, 98.66% precision, and 97.61% F1-score during training and validation;

96.87% recall, 99.17% precision, and 98.01% F1-score during the attack simulation stage;

HTTP forwarder performance 54 request-per-second for normal traffic and 131 request-per-second for malicious requests; the application achieved this throughput in a *single-threaded configuration*.

Outline

Bibliography

1. Open Worldwide Application Security Project (OWASP). (n.d.). *Attacks*. In *OWASP Community Pages*. OWASP Foundation. https://owasp.org/www-community/attacks/
2. Bright Security Inc. (2020). *The Ultimate Beginner's Guide to XSS Vulnerability*. In *Bright Security Blog*. Bright Security Inc. https://brightsec.com/blog/cross-site-scripting-xss/
3. OWASP Foundation. (n.d.). *SQL Injection Prevention Cheat Sheet*. In *OWASP Cheat Sheet Series*. OWASP Foundation. https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html
4. Kaggle. (n.d.). *SQL Injection datasets*. In *Kaggle Datasets*. Kaggle. https://www.kaggle.com/datasets?search=sql+injection

Bibliography

5. Irungu, J., Graham, S., Girma, A., & Kacem, T. (2023). *Artificial Intelligence Techniques for SQL Injection Attack Detection*. In *Proceedings of the 2023 8th International Conference on Intelligent Information Technology (ICIIT '23)*. Association for Computing Machinery, New York, NY, USA, 38-45. https://doi.org/10.1145/3591569.3591576
6. F. Wang, G. Zhang, Q. Kong, L. Fang, Y. Xiao and G. Wang. (2023). *Semantic-Based SQL Injection Detection Method*. In *Proceedings of the 2023 5th International Conference on Artificial Intelligence and Computer Applications (ICAICA)*, Dalian, China, 519–524. IEEE. https://ieeexplore.ieee.org/document/10405528
7. Gabriela Ursachi. (2025). *Artificial Intelligence Models Used in the Detection and Prevention of SQL Injection Attacks* (Master's Dissertation Thesis). Iași, Romania. Supervisor: Assistant Professor Alexandru Archip, PhD.

Bibliography

8. OWASP Foundation. (n.d.). *Types of Cross-Site Scripting*. In *OWASP Community Wiki*. OWASP Foundation. https://owasp.org/www-community/Types_of_Cross-Site_Scripting
9. OWASP Foundation. (n.d.). *Cross-Site Scripting Prevention Cheat Sheet*. In *OWASP Cheat Sheet Series*. OWASP Foundation. https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html
10. OWASP Foundation. (n.d.). *DOM based XSS Prevention Cheat Sheet*. In *OWASP Cheat Sheet Series*. OWASP Foundation. https://cheatsheetseries.owasp.org/cheatsheets/DOM_based_XSS_Prevention_Cheat_Sheet.html
11. Kaur, J., Garg, U., & Bathla, G. (2023). *Detection of cross-site scripting (XSS) attacks using machine learning techniques: a review*. *Artificial Intelligence Review*. Springer. https://link.springer.com/article/10.1007/s10462-023-10433-3

Bibliography

12. Vahid Babaey and Arun Ravindran. (2025). *GenXSS: an AI-Driven Framework for Automated Detection of XSS Attacks in WAFs*. In *Proceedings of SoutheastCon 2025*, pp. 1519-1524. IEEE. https://doi.org/10.1109/SoutheastCon56624.2025.10971558
13. Fodor, Maria-Gabriela. (2024). *Detection and prevention of XSS attacks using machine learning techniques*. Bachelor's thesis, Iași, Romania. Supervisor: Assistant Professor Alexandru Archip.