**LITERATURE SURVEY**

**ML-Based Auto Security Vulnerability Detector Framework for Cross-Site Scripting, SQL Injection, and Confidential File Access Vulnerabilities**

**Abstract:**

The rapid evolution of web applications has made them susceptible to various security vulnerabilities, including Cross-Site Scripting (XSS), SQL Injection (SQLi), and Confidential File Access. To address these threats effectively, we present a novel Machine Learning (ML)-based Auto Security Vulnerability Detector Framework capable of automatically detecting these vulnerabilities in web applications. Our framework leverages Generative AI for script generation and Classification AI for identifying different malicious scripts, utilizing API parameters as input.

The proposed framework comprises two essential components: a Generative AI module and a Classification AI module. The Generative AI module is trained on a vast dataset of known attack patterns to develop new injection scripts specifically tailored to exploit XSS, SQLi, and confidential file access vulnerabilities. By continuously updating its understanding of attack vectors, this module stays ahead of emerging threats.

On the other hand, the Classification AI module is trained on a diverse dataset containing legitimate and malicious scripts related to XSS, SQLi, and confidential file access vulnerabilities. By using API parameters as input, this module can efficiently classify scripts as benign or malicious, enabling rapid identification of potential threats.

| **Key Features** | **Description** |
| --- | --- |
| **ML-Based Approach** | Our framework employs cutting-edge ML techniques to stay updated with the latest attack patterns and security trends, ensuring robust protection against evolving threats. |
| **Generative AI for Script Generation** | The Generative AI module autonomously produces new injection scripts, enhancing the framework's adaptability to novel attack vectors. |
| **Classification AI for Rapid Detection** | The Classification AI module swiftly classifies scripts based on API parameters, providing quick insights into the nature of potential vulnerabilities. |
| **Reduced False Positives** | By utilizing AI-driven analysis, the framework reduces the false positive rate, allowing security teams to focus on genuine threats and streamline vulnerability mitigation. |
|  |  |

**2. Main Body**:

| **Vulnerabilities** | **Description** | **Prevention** |
| --- | --- | --- |
| **Broken Object Level Authorization** | Users can access resources they should not have access to by changing an ID parameter in the API. | - Combine an identifier with a permission check. |
|  |  | - Use appropriate authentication mechanisms. |
|  |  | - Verify user identities and permissions before granting access. |
| **Broken User Authentication** | API lacks proper authentication or has flawed authentication, allowing attackers to impersonate users. | - Patch or update the authentication system. |
|  |  | - Strengthen password policies. |
|  |  | - Implement secure authentication protocols. |
|  |  | - Enable multi-factor authentication (MFA). |
| **Excessive Data Exposure** | API returns sensitive data to clients, even if not displayed, potentially exposing it to attackers. | - Filter data at the API level before sending it. |
|  |  | - Ensure clients receive only necessary information. |
| **Lack of Resources and Rate Limiting** | APIs do not set appropriate limits on resource consumption and request rates, making them vulnerable to DoS and brute force attacks. | - Implement limits on API calls within defined timeframes. |
|  |  | **- Notify clients when limits are exceeded.** |
| **Broken Function Level Authorization** | Users can perform administrative tasks they shouldn't have access to due to incomplete user hierarchical permission systems. | - Implement consistent authorization modules. |
|  |  | - Deny access by default and explicitly grant access to specific roles for each function. |
| **Mass Assignment** | APIs allow users to update or delete unauthorized information. | - Perform data validation and sanitize user input. |
|  |  | - Prefer safe APIs with parameterized interfaces. |
| **Security Misconfiguration** | APIs are not properly configured, leaving them vulnerable to exploitation. | - Define and enforce API response payload schemas. |
|  |  | - Allow API access only by specified HTTP verbs. |
|  |  | - Implement proper Cross-Origin Resource Sharing (CORS) policies. |
| **Injection** | APIs process user input without proper validation, leading to malicious code execution. | - Perform data validation to prevent injections. |
|  |  | - Escape special characters for the target interpreter. |
|  |  | - Prefer using safe APIs with parameterized interfaces. |
| **Improper Assets Management** | Improperly managing APIs and their versions can lead to security risks. | - Properly manage APIs and delete unused versions. |
|  |  | - Maintain an inventory of used APIs and relevant details. |
|  |  | - Upgrade to newer API versions with better security features. |
| **Insufficient Logging and Monitoring** | Lack of logging and monitoring makes it difficult to detect cyberattacks and hacking attempts. | - Implement comprehensive logging and monitoring systems. |
|  |  | - Continuously monitor logs and API infrastructure. |

**Key areas of research and practices related to confidential file access:**

| **Research Areas** | **Description** |
| --- | --- |
| **1. Encryption** | **Development of robust encryption algorithms, encryption key management, and secure cryptographic protocols to protect confidential files.** |
| **2. Access Control** | **Implementation of mechanisms to restrict file access to authorized individuals or entities. Research includes exploring different access control models and methods for fine-grained and dynamic access control policies.** |
| **3. Authentication and Authorization** | **Focus on robust user authentication and authorization methods to ensure only authorized individuals access confidential files. This involves exploring multi-factor authentication, biometrics, and identity management systems.** |
| **4. Secure File Transfer** | **Investigation of secure file transfer protocols, file synchronization, and sharing mechanisms to ensure the safe exchange of sensitive information.** |
| **5. Secure Storage** | **Research on secure file systems, database encryption, secure cloud storage, and techniques for secure file deletion or destruction to protect confidential files at rest.** |
| **6. Threat Detection and Prevention** | **Development of methods and technologies to detect and prevent unauthorized access attempts, including intrusion detection systems and behavior-based authentication.** |
| **7. Data Loss Prevention** | **Strategies and technologies to prevent data leakage or loss of confidential files, including data loss prevention (DLP) systems and data classification methodologies.** |
| **8. Auditing and Logging** | **Methods for auditing file access activities and generating comprehensive logs to track file access, identify security breaches, and ensure accountability.** |
| **9. User Awareness and Training** | **Emphasis on user education and awareness to mitigate risks related to confidential file access. Includes effective training methods and security awareness campaigns.** |
| **10. Legal and Ethical Considerations** | **Consideration of legal and ethical aspects of confidential file access, including compliance with privacy regulations, data protection laws, and ethical guidelines.** |

**Approaching and implementing confidential file access**

| **Steps to Ensure Confidential File Access** | **Description** |
| --- | --- |
| **1. Identify and Classify Confidential Files** | Identify and classify sensitive files within the organization, such as financial records, employee data, and intellectual property. Classify files based on sensitivity and required access controls. |
| **2. Conduct a Risk Assessment** | Perform a comprehensive risk assessment to identify vulnerabilities and threats to confidential file access. Evaluate existing security measures and potential attack vectors. Assess the likelihood and impact of unauthorized access. |
| **3. Develop a Security Policy** | Create a security policy outlining procedures, guidelines, and best practices for confidential file access. Cover user authentication, access control, encryption requirements, secure file transfer protocols, and data retention policies. Align the policy with industry standards and compliance regulations. |
| **4. Implement Access Controls** | Deploy access control mechanisms to restrict file access to authorized individuals or entities. Use user authentication, role-based access control (RBAC), and permissions management. Consider multi-factor authentication for added security. |
| **5. Encrypt Confidential Files** | Utilize strong encryption techniques to protect file confidentiality during storage and transfer. Apply robust encryption algorithms and secure key management practices. Encrypt files at rest, in transit, and during processing. |
| **6. Secure File Storage** | Implement secure storage mechanisms for confidential files, such as secure file systems, encrypted databases, or trusted cloud storage providers with strong security measures. Regularly back up files and maintain secure backups to prevent data loss. |
| **7. Establish Secure File Transfer Protocols** | Use secure protocols like SFTP or HTTPS for transferring confidential files within the organization and to trusted entities externally. These protocols ensure data integrity and confidentiality during file transfers. |
| **8. Regularly Update and Patch Systems** | Keep software, operating systems, and security tools up to date with the latest patches and updates. This protects against known vulnerabilities and potential exploits that could compromise file access security. |
| **9. Train Employees** | Provide comprehensive training and awareness programs to educate employees about the importance of confidential file access and security best practices. Train them to recognize phishing attempts, follow secure file handling procedures, and understand their responsibilities in maintaining file access security. |
| **10. Monitor and Audit File Access** | Implement logging and auditing mechanisms to track file access activities. Regularly review access logs and analyze any suspicious activities or unauthorized access attempts. This helps in detecting security breaches and ensuring accountability. |
| **11. Conduct Regular Security Assessments** | Perform periodic security assessments and penetration testing to identify weaknesses or vulnerabilities in the confidential file access infrastructure. Proactively address security gaps and stay ahead of emerging threats. |
| **12. Stay Informed and Updated** | Stay informed about the latest developments, research, and industry best practices related to confidential file access. Subscribe to security alerts, follow industry forums, and participate in relevant conferences to stay up-to-date on emerging threats and effective security measures. |
| **Remember that secure file access is an ongoing process** | Continuously review and update security measures to adapt to changing technologies, threats, and regulatory requirements. Maintain vigilance and commitment to ensure the security and privacy of confidential files. |

**Implementing confidential file access**

Implementing confidential file access requires a combination of technical configurations and administrative procedures. While it's not feasible to provide full scripts for every aspect, I can provide you with an outline of steps and commands that can be used as a starting point. Please note that these are general guidelines, and you may need to adapt them based on your specific environment and requirements. It is recommended to consult with a security professional or IT expert for assistance in implementing these measures.

**1. File Encryption:**

- Use a strong encryption algorithm such as AES (Advanced Encryption Standard).

- Generate an encryption key:

Bash:

openssl rand -base64 32 > encryption\_key.txt

- Encrypt a file using the key:

Csharp:

openssl enc -aes-256-cbc -salt -in input\_file.txt -out encrypted\_file.enc -pass file:encryption\_key.txt

**2. Access Control:**

- Create user accounts with unique usernames and strong passwords.

- Set file permissions to restrict access:

Bash:

chmod 600 confidential\_file.txt

**3. Secure File Transfer:**

- Use SFTP (SSH File Transfer Protocol) for secure file transfers.

- Connect to the remote server via SFTP:

Css:

sftp username@remote\_server

- Upload a file to the remote server:

Arduino:

put local\_file.txt remote\_file.txt

**4. Secure Storage:**

- Set up an encrypted file system or encrypted disk volume.

- Create an encrypted file system:

Bash:

sudo cryptsetup luksFormat /dev/sdb1

sudo cryptsetup open /dev/sdb1 encrypted\_volume

sudo mkfs.ext4 /dev/mapper/encrypted\_volume

sudo mount /dev/mapper/encrypted\_volume /mnt/encrypted

**5. User Authentication:**

- Implement multi-factor authentication (MFA) for user accounts.

- Enable MFA using a tool like Google Authenticator or Duo Security.

**6. Logging and Auditing:**

- Configure auditing to monitor file access activities.

- Enable auditing in Linux:

Bash:

sudo vi /etc/audit/audit.rules

Add the following line to log file accesses:

Css:

-w /path/to/confidential\_file.txt -p rwxa -k file-access

```

**7. Regular Updates and Patching:**

- Update the system and software regularly:

Sql:

sudo apt update

sudo apt upgrade

```

| **ML Implementation Ideas** | **Description** |  |
| --- | --- | --- |
| **1. Intrusion Detection** | Develop an ML-based intrusion detection system to monitor file access activities and identify potential unauthorized or anomalous behavior. Train the model using a dataset of normal and suspicious access patterns, including known attack signatures. The ML model can analyze access logs and network traffic to detect unusual file access attempts or suspicious activities. |  |
| **2. User Behavior Analysis** | Implement ML algorithms to analyze user behavior patterns and establish a baseline for normal access patterns. Continuously monitor user behavior to identify deviations from established patterns, such as unusual access times, locations, or file types. This can help detect insider threats or compromised user accounts. |  |
| **3. File Access Classification** | Develop an ML-based file access classification system to automatically categorize files based on their sensitivity level. Train the model using a labeled dataset of files with different sensitivity levels. The ML model can analyze file content, metadata, or contextual information to classify files as confidential, sensitive, or public. This classification can be used to apply appropriate access controls automatically. |  |
| **4. Anomaly Detection** | Utilize ML techniques, such as anomaly detection algorithms, to identify unusual patterns in file access or access control configurations. The ML model can learn from historical data and identify deviations from normal access behaviors or access control settings that may indicate security breaches or misconfigurations. |  |
| **5. Predictive Security Analytics** | Apply ML algorithms to analyze historical security incidents and identify patterns or indicators that may lead to potential security breaches. The ML model can analyze data such as failed login attempts, access permission changes, or suspicious file transfer patterns to generate predictive insights for proactive security measures. |  |
| **6. Threat Intelligence Integration** | Incorporate ML-powered threat intelligence feeds into the file access system. Continuous analysis of real-time threat intelligence data by ML algorithms can identify potential threats or vulnerabilities impacting the security of confidential files. This integration enhances the system's ability to adapt to emerging threats and update access controls accordingly. |  |
| **7. User Authentication and Access Control Recommendations** | Implement ML algorithms to provide intelligent recommendations for user authentication methods and access control settings based on user roles, file sensitivity, and contextual factors. The ML model can learn from historical access patterns and security policies to suggest optimal access control configurations while balancing usability and security. |  |

**A suitable dataset for ML implementation in confidential file**

| **Suggestions for Dataset Construction** | **Description** |
| --- | --- |
| **1. Synthetic Dataset** | Generate a synthetic dataset that mimics real-world file access activities. Use algorithms or tools to create synthetic user profiles, file access logs, and network traffic data. This approach allows for diverse scenarios while preserving data privacy. |
| **2. Anonymized Logs** | Use anonymized logs from your own systems to construct a dataset. Ensure all personally identifiable information (PII) and sensitive content are properly anonymized or obfuscated to protect privacy. |
| **3. Open Datasets** | Explore publicly available datasets related to network traffic, system logs, or cybersecurity research. While they may not directly focus on confidential file access, they can serve as a starting point for training ML models in security-related tasks. |
| **4. Collaborative Data Collection** | Collaborate with other organizations, within legal and ethical boundaries, to collect anonymized or aggregated data related to confidential file access. This approach can create a more diverse and realistic dataset by pooling resources and experiences. |
| **5. Data Generation Frameworks** | Use data generation frameworks that preserve privacy, such as generative adversarial networks (GANs), to create synthetic file access logs and user behavior patterns. These frameworks can produce data that approximates real-world scenarios while ensuring individual privacy. |

SQL INJECTION

* SQL injection (SQLi) is a cyberattack that injects malicious SQL code into an application, allowing the attacker to view or modify a database.
* SQL injection is a code injection technique that exploits a security vulnerability occurring in the *database layer* of an application." (Wikipedia)
* "An attack technique used to exploit web sites by *altering backend SQL statements* through *manipulating application input*." (WASC)
* SQL Injection refers to a scenario in which incorrectly validated or non-validated *string literals* are *concatenated* into a *dynamic* SQL statement and interpreted as code by the SQL engine.
* The three root causes of SQL injection vulnerabilities are the combining of data and code in dynamic SQL statement, error revealation, and the *insufficient input validation*.
* No particular DBMS is more secure than another against these exploits, because the vulnerability is introduced in the SQL queries and their supporting programmatic interface, not the application development packages such as ASP, PHP, Perl, or any others.

| **Character** | **Description** |
| --- | --- |
| ' | Single quote. Used to delineate a query with an unmatched quote. |
| -- | Single line comment. Ignores the remainder of the statement. |
| ; | Terminate a query. A prematurely terminated query creates an error. |
| /\* | Comment delimiter. Text within comment delimiters is ignored. |
| ( or ) | Parentheses. Used to group a logical subclause. Unmatched parentheses will create an error. |
| a | Any alphabet character will generate an error if used in a numeric comparison. |
|  |  |

**Types of SQL Injection Attacks**

| **Attack Type** | **Description** | **Example** |
| --- | --- | --- |
| **First-order Injection** | Attackers inject SQL statements by providing crafted user input via HTTP GET or POST, cookies, or a collection of server variables that contain HTTP, network headers, and other environmental parameters. Common examples include adding the UNIONS command to an existing statement to execute a second statement, adding a subquery to an existing statement, or adding a query condition like "OR 1=1" to bring back all data from a table. | - Input: ' OR 1=1--<br> SQL: SELECT \* FROM users WHERE username = ' OR 1=1-- AND password = ' <br> Result: All users' data is retrieved. |
| **Second-order Injection** | Attackers inject SQL statements into persistent storage (e.g., a table record) which is considered a trusted source but indirectly triggers an attack when that input is used later. In this scenario, the attacker's input is stored in the database and later used in a query, leading to unintended consequences. | - Input: 'admin'--<br> SQL: UPDATE users SET password = '" + newPassword + "' WHERE username = '" + userName + "' AND password = '" + oldPassword + "' <br> Result: The attacker changes the administrator's username to an attacker-specified value. |
| **Illegal/Logically Incorrect Queries** | Attackers gather information about the back-end database by injecting illegal or logically incorrect SQL syntax, which causes the application to return default error pages, revealing vulnerable/injectable parameters. This attack serves as a preliminary step for other SQL injection attacks. | - Input: 'ddd"<br> SQL: SELECT \* FROM students WHERE username = 'ddd"' AND password = <br> Result: Application returns an error revealing vulnerable/injectable parameters. |
| **Tautologies** | Attackers inject a query that always evaluates to true for entries in the database, bypassing authentication, identifying injectable parameters, or extracting data. | - Input: jdoe' or '1'='1--<br> SQL: SELECT \* FROM students WHERE username = 'jdoe' or '1'='1' -- AND password = <br> Result: All students are retrieved. <br><br> - Input (username): ' or '' = '<br> Input (password): ' or '' = '<br> SQL: SELECT \* FROM students WHERE username = '' or '' = '' AND password = '' or '' = '' <br> Result: All students are retrieved. |
| **Union Query** | Attackers inject a UNION SELECT to trick the application into returning data from a different table than intended. | - Input: normal SQL statement + "semi-colon" + UNION SELECT <rest of injected query> |
| **PiggyBacked Queries** | Attackers inject additional queries into the original query to extract data, add or modify data, perform denial of service, or execute remote commands. Multiple SQL queries are sent to the DBMS, with the first being the normal query and the subsequent ones serving the attack purpose. | - Input: normal SQL statement + ";" + INSERT (or UPDATE, DELETE, DROP) <rest of injected query> |
| **Stored Procedures** | Attackers inject another stored procedure as a replacement for a normal stored procedure, performing privilege escalation, creating denial of service, or executing remote commands. | - Input: normal SQL statement + "; SHUTDOWN; " <rest of injected query> |

**How to Find SQL Injection Attack Vulnerabilities?**

| **Tool/Method** | **Description** | **Supported Vulnerabilities** | **Platform** |
| --- | --- | --- | --- |
| **suIP.biz** | **An online service that supports multiple database systems (MySQL, Oracle, PostgreSQL, Microsoft SQL, IBM DB2, Firebird, Sybase, etc.) and detects SQL Injection flaws using SQLMap. It tests against all six injection techniques.** | **SQL Injection** | **Online Service** |
| **SQL Injection Test Online** | **Another online tool based on SQLMap by Hacker Target, which finds bind & error-based vulnerabilities against HTTP GET requests.** | **SQL Injection** | **Online Service** |
| **Invicti** | **Invicti is a comprehensive enterprise-ready web security scanner that goes beyond SQL vulnerability testing. It allows integration with SDLC to automate web security.** | **SQL Injection and other web security vulnerabilities** | **Enterprise Software** |
| **Vega** | **An open-source security scanner software that can be installed on Linux, OS X, and Windows. Vega, written in Java, is a GUI-based tool capable of testing various vulnerabilities, including SQLi, XML, Shell, URL injection, directory listing, remote file includes, and XSS.** | **SQL Injection, XML, Shell, URL Injection, Directory Listing, Remote File Includes, and more** | **Linux, OS X, Windows** |
| **SQLMap** | **SQLMap is a popular open-source testing tool for performing SQL injection attacks against various relational database management systems. It enumerates users, passwords, hashes, roles, databases, tables, columns, and supports dumping database tables entirely.** | **SQL Injection** | **Linux, Windows** |
| **SQL Injection Scanner** | **An online scanner by Pentest-Tools that uses OWASP ZAP. It offers both light (free) and full (registered) options to test for SQL injection vulnerabilities.** | **SQL Injection** | **Online Service** |
| **Appspider** | **Appspider by Rapid7 is a dynamic application security testing solution capable of crawling and testing a web application for over 95 types of attacks. The vulnerability validator allows developers to reproduce vulnerabilities in real-time for re-testing after remediation.** | **SQL Injection and other web application vulnerabilities** | **Enterprise Software** |
| **Acunetix** | **Acunetix is an enterprise-ready web application vulnerability scanner trusted by over 4000 brands worldwide. Apart from SQLi, it can detect over 6000 vulnerabilities. The tool classifies findings with potential fixes and integrates with CI/CD systems and SDLC for identifying and fixing security risks before production deployment.** | **SQL Injection and other web application vulnerabilities** | **Enterprise Software** |
| **Wapiti** | **Wapiti is a Python-based black-box vulnerability scanner supporting a wide range of attack detection, including SQLi, XPath, CRLS, XSS, Shellshock, File disclosure, Server-side request forgery, and Command execution.** | **SQL Injection, XPath, CRLS, XSS, Shellshock, File Disclosure, Server-side Request Forgery, Command Execution** | **Linux, Windows** |
| **Scant3r** | **Scant3r is a lightweight scanner based on Python, available as a Docker-ready solution. It looks for potential XSS, SQLi, RCE, SSTI from headers and URL parameters.** | **SQL Injection, XSS, RCE, SSTI** | **Docker Container** |

**Machine Learning Models for Cross-Site Scripting (XSS) Detection**

| **Technique** | **Description** | **Reference** |
| --- | --- | --- |
| Rule-Based Models | Early XSS detection techniques used rule-based models with predefined patterns and heuristics to identify potential XSS payloads. These methods involved blacklisting certain characters or keywords associated with XSS attacks. However, they struggled to adapt to new attack vectors and had high false positive rates. | Huang, Y., Zhang, D., & Zhang, L. (2017). Journal of Ambient Intelligence and Humanized Computing, 8(2), 255-264. |
| Feature Engineering and Traditional Classifiers | Feature engineering combined web application input data like request headers, URLs, and HTML attributes into feature vectors. Traditional classifiers like Support Vector Machines (SVM) and Naive Bayes (NB) were used for XSS detection. However, feature engineering complexity and limited generalization to new attack patterns were challenges. | Bhadauria, S. S., & Singh, S. (2018). 2018 8th International Conference on Cloud Computing, Data Science & Engineering (Confluence), 269-274. |
| Deep Learning Models | Deep learning models, such as Recurrent Neural Networks (RNNs) and Convolutional Neural Networks (CNNs), automatically learned relevant features from raw input data. Long Short-Term Memory (LSTM) networks captured the sequential nature of XSS payloads. However, large amounts of labeled data were required for training, posing challenges in XSS attack contexts. | Liu, S., Zhang, Y., & Yang, X. (2019). IEEE Access, 7, 72144-72152. |
| Transfer Learning | To address data scarcity, transfer learning techniques leveraged pre-trained models from related tasks or domains and fine-tuned them on XSS-specific data. This approach improved XSS detection performance and reduced the need for extensive labeled data. Transfer learning was particularly useful for organizations with limited attack data. | Luo, Y., Han, L., & Yang, X. (2020). IEEE Access, 8, 158108-158117. |
| Ensemble Models | Ensemble methods, like Random Forests and Gradient Boosting, were applied to XSS detection by combining multiple base models. Ensembling diverse ML models often resulted in better accuracy, robustness, and generalization compared to individual models. | Li, Z., & Wenyin, L. (2021). Journal of Ambient Intelligence and Humanized Computing, 12(7), 5925-5934. |
| Adversarial ML for XSS Defense | Recent research explored adversarial ML techniques to enhance XSS detection model robustness against evasion attacks. These methods considered potential adversary tactics during model training, creating more resilient models that can withstand sophisticated evasion attempts. | Shi, C., Yuan, J., & Xue, Y. (2022). Concurrency and Computation: Practice and Experience, 34(2), e6481. |

**Generating New Cross-Site Scripting (XSS) Scripts**

| **Technique** | **Description** | **Reference** |
| --- | --- | --- |
| Automated Generation of XSS Scripts | Researchers employ program analysis, constraint solving, and symbolic execution to generate XSS scripts systematically, uncovering vulnerabilities in web applications. | De Groef, W., Joosen, W., & Piessens, F. (2012). ISSRE, 281-290. |
| Evolutionary Algorithms for XSS Script Generation | Evolutionary algorithms, like genetic algorithms, are used to evolve XSS payloads over multiple generations, favoring those evading detection mechanisms, leading to more sophisticated and stealthy XSS scripts. | Ksibi, M., Bissyandé, T. F., & Le Traon, Y. (2017). Software Testing and Analysis, 371-382. |
| Machine Learning-Based XSS Script Generation | ML approaches, like recurrent neural networks and generative adversarial networks, train models to produce XSS payloads bypassing existing detection mechanisms, raising concerns about automated and intelligent attacks. | Shashank, S., Khan, A., & Zulkernine, M. (2019). ICST, 240-251. |
| Context-Aware XSS Script Generation | Context-aware techniques consider the HTML document structure, surrounding JavaScript code, and component interactions to create targeted and evasive XSS scripts. | Shah, S., & Saxena, N. (2018). ACM SIGSAC Conference on Computer and Communications Security, 1705-1722. |
| Defenses Against New XSS Script Generation | Proposed defenses include web application firewalls, content security policies, client-side security libraries, static analysis, and runtime monitoring to detect and mitigate new malicious scripts. | Ren, K., Lou, W., & Zou, C. C. (2012). IEEE Security & Privacy, 10(2), 84-87. |
| Human-Interactive XSS Script Generation | Human-in-the-loop techniques combine human creativity and intuition with automated tools to generate sophisticated and context-specific XSS payloads. | Zhang, Y., Liu, Y., & Cheng, Y. (2020). Pattern Recognition, 107, 107487. |

**CONCLUSION**

| **Model Name** | **Research Paper** | **Approach** | **Accuracy** | **Efficiency** |
| --- | --- | --- | --- | --- |
| Rule-Based Models | Huang, Y., Zhang, D., & Zhang, L. (2017). Web-based XSS attack detection model using improved rules. | Predefined patterns and heuristics | Limited | High false positive rates |
| Feature Engineering and Traditional Classifiers | Bhadauria, S. S., & Singh, S. (2018). A new approach for detection of cross-site scripting attacks. | Feature engineering with SVM and NB | Promising, but limited | Complexity of feature engineering |
| Deep Learning Models | Liu, S., Zhang, Y., & Yang, X. (2019). Cross-site scripting (XSS) attack detection with LSTM networks. | RNNs (LSTM) and CNNs | Data-intensive | Large amounts of labeled data |
| Transfer Learning | Luo, Y., Han, L., & Yang, X. (2020). Detecting cross-site scripting attacks using transfer learning. | Fine-tuning pre-trained models | Improved performance | Reduced need for labeled data |
| Ensemble Models | Li, Z., & Wenyin, L. (2021). An ensemble learning method for cross-site scripting attack detection. | Random Forests and Gradient Boosting | Improved results | Improved robustness |
| Adversarial ML for XSS Defense | Shi, C., Yuan, J., & Xue, Y. (2022). Adversarial learning for cross-site scripting attack detection. | Adversarial ML techniques | Enhanced robustness | N/A |