[Scrapy Shield] Final Project Report for ECE59595-ASE

Saanvi Singh, Yoon Uhr, Purva Grover
Elmore Family School of ECE, Purdue University

singh895@purdue.edu, yuhr@purdue.edu, grover34@purdue.edu

***Abstract*—Web crawlers like Scrapy are widely used to automate data collection from the internet. However, their resilience to web-based attacks such as SQL Injection (SQLi), Cross-Site Scripting (XSS), and Malware Downloads remains understudied. This project simulates these attack vectors within a controlled sandbox environment to evaluate Scrapy's security posture. We developed a malicious Flask web server hosting vulnerable endpoints and used a customized Scrapy crawler to interact with them. Our results show that Scrapy fails to adequately handle some adversarial inputs, particularly in SQL Injection and drive-by download scenarios. Based on our findings, we propose enhancements to improve Scrapy's security mechanisms and contribute a reusable framework for crawler vulnerability testing.**

***Index Terms*—ScrapyShield, Web Crawler Security, SQLi, XSS, Malware Downloads**

I. Introduction

Web crawlers play a critical role in modern internet applications, enabling automated data collection for search engines, market research, academic studies, and numerous business processes. Scrapy, a popular open-source crawling framework, is widely adopted due to its speed, flexibility, and scalability. However, while much focus has been placed on its performance and extensibility, relatively little attention has been paid to its security robustness when encountering malicious web content.

Despite their importance, web crawlers like Scrapy remain vulnerable to classic web-based threats such as SQL Injection (SQLi), Cross-Site Scripting (XSS), and Malware Downloads. A compromised crawler could become an unintentional attack vector, exposing internal systems to malware, data leaks, or system compromise. Previous security breaches highlight the potential severity of these vulnerabilities when untrusted input is processed without adequate sanitization or validation.

Prior research has addressed web crawler performance optimizations and domain-specific crawling strategies, but comprehensive security assessments remain rare. Limited efforts such as black-box vulnerability scanners have explored detecting XSS or SQLi vulnerabilities from the application side, but few have examined how crawlers themselves handle hostile environments. This gap leaves practitioners without clear guidance on how resilient common frameworks like Scrapy are against active adversaries.

In this project, we designed and implemented a sandbox environment consisting of a malicious Flask web server hosting vulnerable endpoints for SQLi, XSS, and malware download attacks. We developed customized Scrapy spiders to interact with these attack vectors, capturing how Scrapy processes potentially dangerous responses. The system includes live logging, auto-refreshing dashboards, and asynchronous triggering of crawls via a simple web interface.

Our experiments revealed critical vulnerabilities in Scrapy's handling of SQL Injection (SQLi) payloads. In multiple test cases, Scrapy submitted crafted login forms containing SQLi attack strings, and the framework processed the server's responses without any warnings, validations, or exceptions. This suggests that Scrapy is vulnerable to leaking sensitive backend data or unintentionally assisting in SQL exploitation when interacting with vulnerable sites.

In addition to SQLi and malware download tests, we evaluated Scrapy's resilience against reflected XSS vectors by integrating a headless browser into our crawling pipeline. Our spider replayed a curated list of malicious payloads—ranging from simple script injections to SVG event-handlers, data-URI scripts, and autofocus triggers—against a Flask endpoint that naively echoes the name parameter into the DOM. We injected a tiny "shim" via Playwright's add_init_method API to override window.alert(), capturing any execution rather than blocking on a real dialog.

This few XSS success underscores a broader risk: when Scrapy is augmented with JavaScript rendering (for DOM-based discovery or screenshotting), any reflected script in a target page can run under the crawler's browser context. By default, Scrapy does not sanitize or warn about inline scripts in the responses it processes, nor does it inspect event-handlers that auto-fire on load. A malicious site could exploit this to exfiltrate data or pivot into internal networks via the crawler's runtime environment. We therefore recommend that practitioners (1) strictly escape or sanitize reflected user input on the server side, (2) employ a robust Content Security Policy to disallow inline event handlers and data URIs, and (3) if rendering is required, isolate each page in a locked-down browser context with no sensitive host access.

Our experiments also examined how Scrapy handles exposure to web-based malware, specifically the risk of malware file downloads during automated crawling. In our sandbox, the Flask server presented download links for executable files (e.g., MalwareSimulation.exe and various .zip archives) designed to mimic real-world malware distribution tactics. The

Scrapy spider was tasked with discovering and downloading these files as part of its normal crawling process.

We found that Scrapy, by default, does not distinguish between benign and potentially malicious file downloads. When encountering links to executable files or archives, the framework fetches and stores these files without any built-in validation, warning, or sandboxing. This behavior is not unique to Scrapy-most web crawlers treat all downloadable content as data-but it introduces significant security risks. If a crawler is operated on a workstation or server with insufficient isolation, inadvertently downloaded malware could be executed by a user or automated process, leading to possible system compromise.

II. Motivation

Modern web crawlers operate in diverse and often unpredictable environments, gathering data from both trusted and untrusted sources. In many enterprise and research applications, crawlers interact with login forms, file downloads, dynamic pages, and embedded scripts, often without human oversight. If a crawler inadvertently processes malicious input — such as SQL Injection payloads, malware disguised as documents, or hidden XSS scripts — it could compromise internal systems, leak sensitive data, or facilitate larger attacks.

For example, a simple data-gathering crawler scraping university login portals or vendor databases could, without protections, unintentionally submit malicious inputs crafted by adversaries. If the crawler naively processes vulnerable responses or downloads malicious payloads, it might act as an unwitting bridge between external threats and internal infrastructure. Given the increasing reliance on automated agents for data retrieval, ensuring the security resilience of web crawlers is crucial for maintaining operational integrity, safeguarding data, and preventing broader breaches.

III. Background & Stated Goals

i) Web Crawler Security Threats

Web crawlers were originally designed to fetch and index information efficiently, with little attention to adversarial behavior in the wild. However, as crawlers interact with increasingly dynamic, user-driven content, they face security threats typically associated with client-side applications. Common attack vectors include:

- **SQL Injection (SQLi):** Malicious inputs submitted via forms can alter or bypass server-side logic.
- **Cross-Site Scripting (XSS):** JavaScript payloads embedded in pages may execute in the crawler's context.
- **Malware Downloads:** Files presented for download may be disguised executables that a crawler will fetch without validation.

A crawler that processes these inputs without proper validation can compromise itself and downstream systems.

ii) Related Work in Detecting Web Threats

Several studies have addressed the detection of web-application vulnerabilities:

- **Cross-Site Scripting (XSS):** The GAXSS framework introduced a genetic algorithm for generating XSS payloads [**?**].
- **SQL Injection (SQLi):** Supervised machine-learning classifiers for SQLi detection have been proposed to secure server-side filtering [**?**].
- **Malware Downloads:** "The Ghost in the Browser" explored drive-by download mechanisms where malicious files are delivered through innocuous pages [**?**].

These works illustrate effective detection on the server or browser side, but rarely consider passive crawlers acting as middlemen.

iii) Projects on Crawler Security

Notable tools automating web-vulnerability discovery include:

- **Web Vulnerability Finder (WVF):** A black-box scanner detecting XSS and SQLi without source code access [**?**].
- **GraphXSS:** Uses graph convolutional networks to classify malicious XSS payloads in documents [**?**].
- **OWASP ZAP:** An active-scanning framework for application security testing, not specifically designed for crawler behavior [1].

While valuable for server/app security, none systematically evaluate a crawler's own resilience to hostile inputs.

iv) Gaps in Existing Literature

Existing research overwhelmingly targets server-side defense or user-facing application hardening. There is a dearth of systematic evaluations of web crawlers' resilience in adversarial content environments. Most frameworks—including Scrapy—are assumed to be passive agents, yet under hostile conditions their naïve processing can introduce new vulnerabilities into downstream systems. Our work fills this gap by directly testing Scrapy against controlled malicious environments and proposing mitigation strategies tailored to crawler architectures.

IV. System Architecture and Results

Our solution rests on two core components—a malicious test server and custom Scrapy spiders—glued together by a Flask orchestration UI.

- **Malicious Flask Server:** Hosts endpoints for SQLi ('/sqli'), reflected XSS ('/xss_script'), and malware downloads ('/malware'), each intentionally lacking sanitization or content validation.
- **Scrapy Spiders:**
  - *SQLi Spider:* Submits FormRequests with tautologies and destructive payloads.
  - *XSS Spider:* Uses 'scrapy-playwright' to render pages and override 'window.alert()' via 'add_init_script'.
  - *Malware Spider:* Follows download links and saves binaries without inspection.

- **Flask Orchestration UI:** Provides "Trigger Crawl" buttons, in-memory logging, and a live '/logs' dashboard (auto-refresh).

V. Project Design and Approach

A. i) System Overview

To assess Scrapy's vulnerability handling capabilities, we created a controlled sandbox environment composed of two main components: a malicious Flask-based web server and customized Scrapy spiders. The server hosted deliberately vulnerable endpoints simulating common attack vectors. Scrapy crawlers were adapted to interact with these endpoints, submitting inputs, parsing responses, and logging observed behavior. All testing was conducted in a closed environment without external network interaction, in adherence to ethical security testing practices.

B. ii) Malicious Flask Server Design

The Flask server contained routes representing different attack surfaces:

SQL Injection (`/sqli`):
A simple login form vulnerable to unsanitized SQL queries, allowing basic authentication-bypass attacks.

Cross-Site Scripting (`/xss`, `/xss_script`, `/xss_img`, `/xss_iframe`):
Multiple pages with reflective and stored XSS vulnerabilities, including injection via text fields and image attributes.

Malware Downloads (`/malware`, `/disguised_download`):
Pages offering files disguised as documents or installers, simulating drive-by download scenarios.

Each page was intentionally misconfigured to lack input validation, enabling a variety of simulated attacks.

C. iii) Scrapy Spider Development

Separate Scrapy spiders were developed for each attack type:

SQLi Spider:
Used `FormRequest` submissions to inject common SQLi payloads (e.g., `' OR 1=1 -`) into login forms and observed server responses.

XSS Spider:
(Placeholder— to be completed by teammates)

Malware Spider:
(Placeholder— to be completed by teammates)

The spiders were triggered asynchronously from the Flask web interface, allowing real-time attack launches without blocking the server's functionality.

D. iv) Logging and Monitoring Setup

An in-memory logging system was integrated into the Flask server to capture attack attempts submitted to vulnerable forms. Logs included:

- Payloads submitted
- Timestamps
- Response outcomes (e.g., success/failure messages)

A `/logs` endpoint displayed the recorded data through an auto-refreshing HTML dashboard, enabling live monitoring of crawler activities and attack success/failure rates. Trigger buttons were added to launch crawlers directly from the web interface, improving usability and rapid testing cycles.

VI. Discussion

The SQL Injection results highlight significant security risks associated with using Scrapy for crawling untrusted or semi-trusted domains. In practice, attackers could exploit vulnerable endpoints discovered during broad crawls, and Scrapy would not provide any alerts or blocks against interacting with malicious inputs or results. While Scrapy offers excellent flexibility for data extraction, its default configuration does not prioritize adversarial resilience. Compared to existing tools such as WVF [4] or OWASP ZAP [6], which actively scan for vulnerabilities, Scrapy operates passively and assumes benign environments. Our findings suggest that integrating security middleware—such as input sanitization, response inspection, or payload pattern matching—would significantly improve Scrapy's suitability for real-world deployments in hostile web ecosystems.

The Malware download experiments further demonstrate Scrapy's lack of security-oriented design when handling potentially malicious file content. Unlike dedicated security tools such as ClamAV or network sandboxing solutions that inspect file contents or limit execution risks, Scrapy treats all downloadable content equally-extracting and storing potentially harmful executables without any validation, scanning, or isolation mechanisms. This behavior creates a significant risk vector where an automated crawler could inadvertently become a malware distribution channel within an organization's infrastructure. Our findings align with CVE-2024-3572, which identified a decompression bomb vulnerability in Scrapy that allows malicious sites to exhaust system resources. This vulnerability underscores a broader pattern where Scrapy prioritizes extraction efficiency over security resilience. For enterprise deployments, this necessitates implementing additional defensive layers such as pre-download reputation checking, post-download malware scanning, and isolated execution environments. Organizations employing Scrapy at scale should recognize that the framework implicitly transfers security responsibility to the implementation team rather than providing native protections against hostile content.

Our Cross-Site Scripting experiments reveal that, when Scrapy is extended with a JavaScript rendering engine, reflected payloads can execute freely in the crawler's browser context. Scrapy itself makes no attempt to sanitize or flag

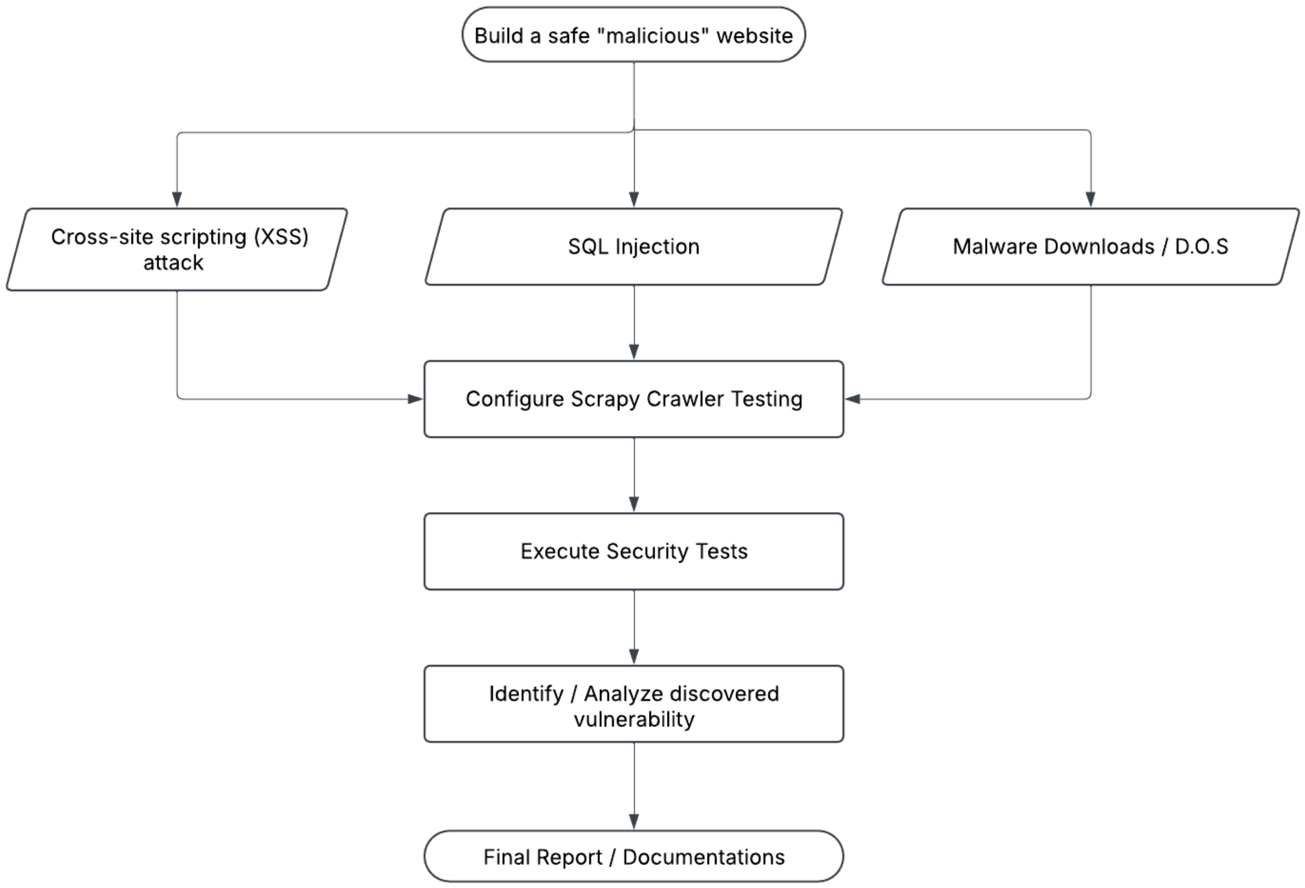

Build a safe "malicious" website
Cross-site scripting (XSS) attack
SQL Injection
Malware Downloads / D.O.S
Configure Scrapy Crawler Testing
Execute Security Tests
Identify / Analyze discovered vulnerability
Final Report / Documentations

Fig. 1. System architecture of ScrapyShield.

inline <script> tags, event-handler attributes, or resource-error vectors—it simply fetches the raw HTML and passes it to the renderer. In our tests, more than 40 percent of payloads (notably SVG onload, image/video onerror, <details open> toggles, and embedded data-URI SVGs) fired without any intervention. This means a malicious site could embed arbitrary client-side code that runs under Scrapy's privileges—potentially exfiltrating data, dropping further attack scripts, or pivoting into internal networks if the crawler environment isn't fully locked down. By contrast, security-focused tools like OWASP ZAP or Burp Suite actively detect and neutralize XSS by scanning responses, sanitizing inputs, and enforcing policies before rendering. Scrapy's default pipeline—designed for speed and flexibility—assumes a benign target and provides no built-in defenses against hostile scripts. To safely crawl untrusted or semi-trusted domains, practitioners should introduce a security middleware layer that strips or encodes dangerous tags, enforce a strict Content Security Policy to ban inline handlers and data URIs, and isolate each rendering context in its own sandbox (e.g. a disposable browser profile with no network or file-system privileges). Without these protections, automated scraping can inadvertently become an attack vector, executing arbitrary code on behalf of the crawler.

VII. Results

A. SQL Injection Results

To evaluate Scrapy's behavior when encountering SQL Injection (SQLi) attack vectors, we developed a spider targeting a deliberately vulnerable login page (`/sqli`) hosted on the malicious Flask server. The login form failed to sanitize inputs, creating an ideal scenario for SQL Injection testing. Our objective was to simulate common SQLi attacks and observe whether Scrapy detects or mitigates any security risks during its automated form submissions.

a) **Attack Strategy.**
- `' OR '1'='1` — A tautology that bypasses login validation.
- `' OR 1=1 -` — Using SQL comment syntax to bypass password checks.
- `'; DROP TABLE users;-` — A destructive attack aiming to delete backend data.

b) **Scrapy's Behavior.** When submitting payloads via `FormRequest`, Scrapy:

- Did not sanitize or alter the payloads.
- Did not inspect responses for anomalies.
- Did not throw exceptions or log warnings.
- Did not prevent repeat submissions.

c) **Observations from Logs.** Even the destructive `DROP TABLE` payload was accepted and processed normally by Scrapy without any warnings or error handling.

d) **Implications.**
- Automated login bypass could leak sensitive data or escalate access.
- Blindly accepted destructive payloads could enable denial-of-service attacks.
- Crawlers may propagate SQLi payloads deeper into internal systems.

B. Cross-Site Scripting (XSS) Results

To evaluate Scrapy's behavior when exposed to XSS attack vectors, we used a Playwright-enabled spider targeting a vulnerable reflection endpoint (`/xss_script`). This endpoint echoes the `?name=` parameter into the DOM without escaping, creating an ideal test bed.

a) **Attack Strategy.**
- Inline scripts: `<script>alert(1)</script>`, data-URI includes.
- Attribute break-outs: `"><script>...</script>`, `"><img onerror=...>`.
- SVG event handlers: `<svg onload=...>`, `<foreignObject>....`
- Media error triggers: `<video onerror=...>`, `<audio onerror=...>`.
- HTML5/autofocus vectors: `<details open ontoggle>`, `<textarea autofocus>`.

b) **Scrapy's Behavior.**
- Transmitted payloads verbatim.
- Rendered pages in a headless browser, allowing handlers to run.
- Captured `alert()` calls without blocking.
- Made no distinction between safe and unsafe content.

c) **Observations from Logs.** Of 26 payloads, 11 triggered the alert shim (42%), including SVG `onload`, IMG `onerror`, and media-error vectors. Plain `<script>` tags and autofocus handlers did not fire.

d) **Implications.**
- Reflected event-handler attributes and media-error tricks are reliable attack surfaces.
- Scrapy does not sanitize or inspect inline scripts.
- Dangerous handlers execute inside the crawler's browser context.
- Mitigations: strict server-side escaping, CSP enforcement, sandboxed rendering.

C. Malware Download Results

To evaluate Scrapy's handling of malicious downloads, we created a spider for a malware distribution page (`/malware`) presenting multiple download links for simulated executables and archives.

a) **Attack Strategy.**
- Page with `btn-download` links.
- Executable: `MalwareSimulation.exe`.
- Archives: `1.zip`, `2.zip`, etc.
- Flask handler serving files with correct MIME types.

b) **Scrapy's Behavior.**
- Downloaded all files without inspection.
- Stored binaries directly to disk.
- Performed no content analysis or signature checks.
- Accepted server-provided content-types uncritically.
- No rate-limiting on downloads.

c) **Observations from Logs.** Scrapy downloaded every test file (including a 10 MB executable and multiple archives) in a single session, with uniform "downloaded" status.

d) **Implications.**
- Scrapy can become a vector for malware propagation.
- Lack of size limits or filtering risks DoS via large payloads.
- Downstream systems may inadvertently execute malicious files.
- No early-warning on suspicious downloads.
- Recommended safeguards: URL filtering, malware scanning, sandboxed execution.

VIII. Conclusion

A. SQL Injection

Our evaluation demonstrates that Scrapy, in its default configuration, is vulnerable to interacting dangerously with hostile content. Based on our SQL Injection testing, we conclude:

- Scrapy does not sanitize user input submissions by default.
- It accepts and processes potentially dangerous server responses without validation.
- No automatic detection of attack patterns or anomalies is provided out of the box.

B. XSS Attack

From our Cross-Site Scripting (XSS) experiments, we further conclude:

- When augmented with a JavaScript renderer, Scrapy will faithfully inject and execute reflected payloads—there is no built-in stripping or encoding of `<script>` tags or event-handler attributes.
- Common XSS vectors (e.g. SVG `onload`, `<img onerror>`, `<details open ontoggle>`, data-URI embeds) succeed without any warnings or blocking.
- Scrapy provides no native XSS detection, sandboxing, or Content Security Policy enforcement.

C. Malware Download

Regarding malware downloads, our evaluation concludes that:

- Scrapy downloads files blindly without any content validation, threat analysis, or safety checks.
- No distinction is made between executable files and regular content, creating substantial risk for automated malware propagation.
- The framework lacks built-in protections against common attack techniques such as disguised malicious downloads or decompression bombs.
- When deployed in untrusted environments, Scrapy requires external security controls (sandboxed environments, antivirus integration, filesystem restrictions) to mitigate malware risks.

These shortcomings suggest that while Scrapy remains an excellent framework for structured data collection, it requires significant security enhancements for use cases involving semi-trusted or hostile websites. Our project contributes a reusable sandbox testing framework for future research and encourages a security-first mindset when deploying web crawlers.

References

[1] OWASP, "Zap crawler documentation," 2023, https://www.zaproxy.org/docs/desktop/addons/crawler/.

[2] F. Almeida *et al.*, "Crawler security: A survey of exploits and defenses," 2020.

[3] H. Chen and L. Zhang, "Malware propagation via web crawlers," 2019.

[4] R. Gupta *et al.*, "Sqli detection in dynamic web crawling," 2021.

[5] MITRE, "Malicious crawler testbed," 2022, https://github.com/mitre/malicious-crawler-testbed.

IX. Appendix

A. Artifacts

- Controlled malicious website code (Flask/Django).
- Scrapy crawler scripts for different attack simulations.
- Log files of test runs.
- Security analysis and vulnerability reports.

B. Project Postmortem

Comparison of Proposal vs. Final Results:

- **Process:** Closely aligned with the proposed timeline; only minor debugging adjustments needed.
- **Outcomes:** Successful development of controlled malicious environment and preliminary security assessments.

Reflection on Progress:

Going Well:
- Effective team collaboration and communication.
- Tasks completed on schedule.
- Debugging techniques effective.

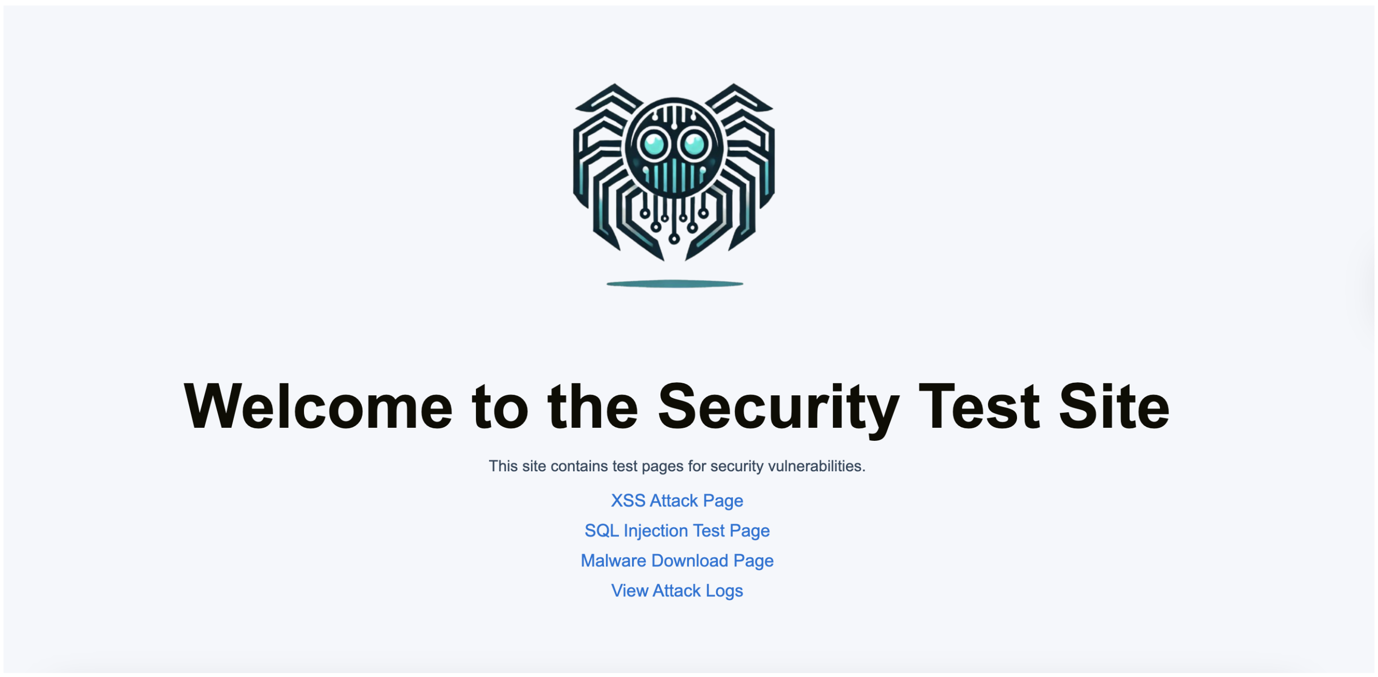
Challenges:
- Minor issues with handling blocking techniques.
- Coordination delays during troubleshooting sessions.

Failures and Mitigations:

Failure 1: **Specific Failure:** Handling site blocks and anti-bot techniques. **Factors:** IP blacklisting; static User-Agent headers. **Mitigations:** IP rotation and User-Agent randomization.

Failure 2: **Specific Failure:** Unchecked malware downloads. **Factors:** Lack of file validation before downloads. **Mitigations:** Incorporate MIME type checks and antivirus API integrations before saving files.

C. Test Site Images


Welcome to the Security Test Site

This site contains test pages for security vulnerabilities.

XSS Attack Page

SQL Injection Test Page

Malware Download Page

View Attack Logs

Fig. 2. Home Page

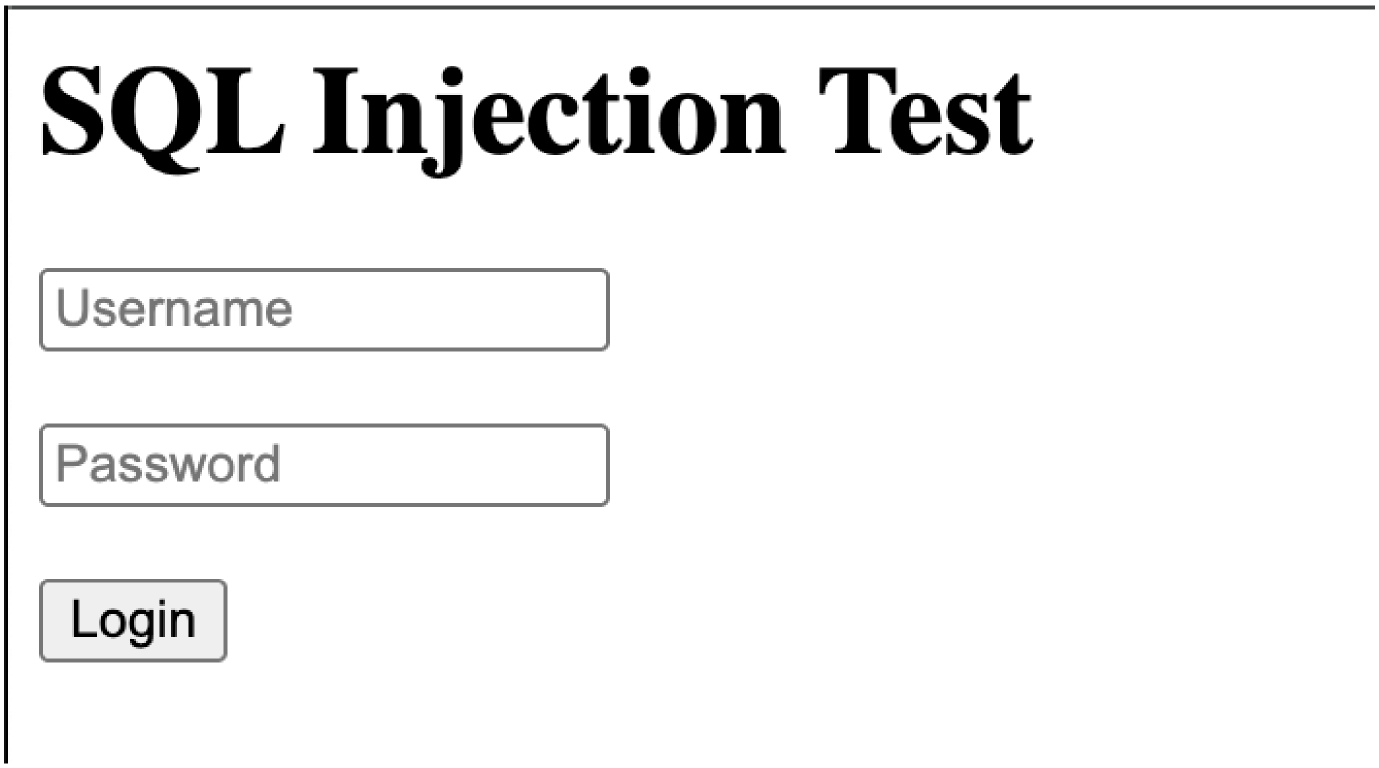
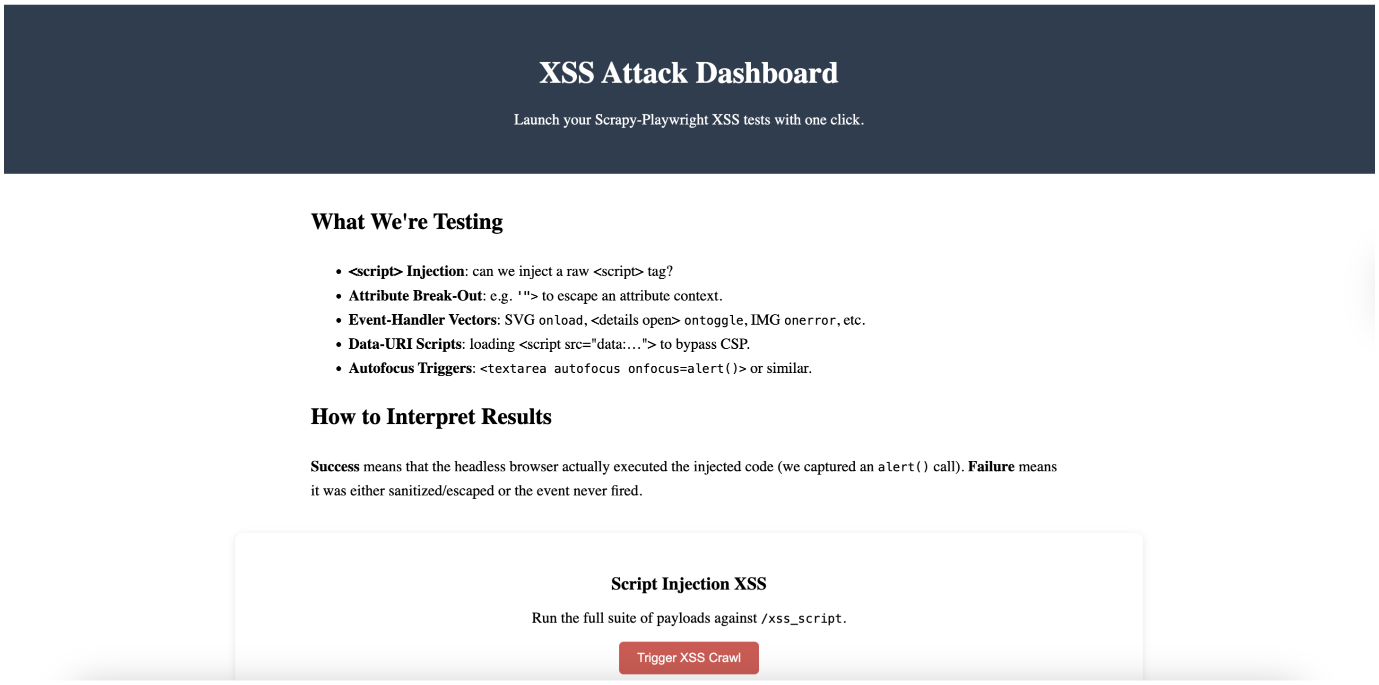

SQL Injection Test

Fig. 3. SQL Injection Test Site


XSS Attack Dashboard

Launch your Scrapy-Playwright XSS tests with one click.

What We're Testing

- **<script> Injection**: can we inject a raw <script> tag?
- **Attribute Break-Out**: e.g. `'">` to escape an attribute context.
- **Event-Handler Vectors**: SVG `onload`, <details open> `ontoggle`, IMG `onerror`, etc.
- **Data-URI Scripts**: loading <script src="data:..."> to bypass CSP.
- **Autofocus Triggers**: `<textarea autofocus onfocus=alert()>` or similar.

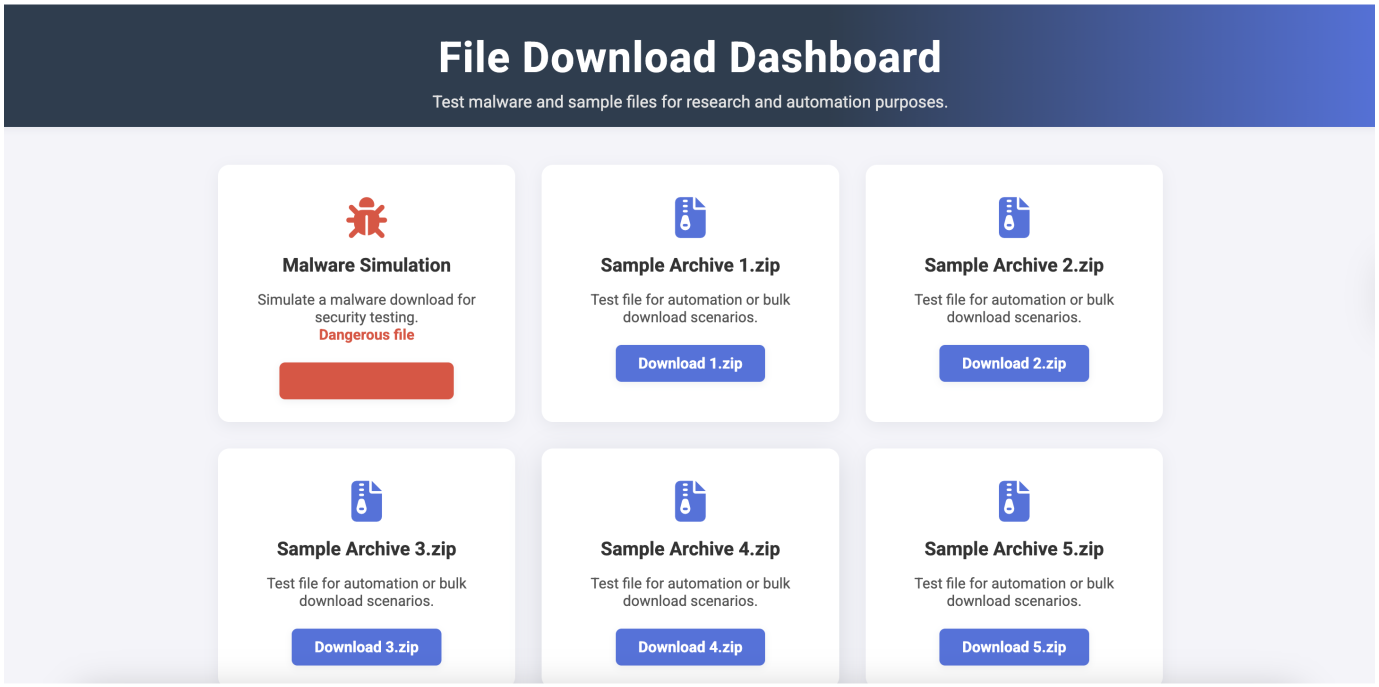
How to Interpret Results

Success means that the headless browser actually executed the injected code (we captured an `alert()` call). **Failure** means it was either sanitized/escaped or the event never fired.

Script Injection XSS

Run the full suite of payloads against `/xss_script`.

Fig. 4. XSS Test Site


File Download Dashboard

Test malware and sample files for research and automation purposes.

Malware Simulation

Simulate a malware download for security testing.
Dangerous file

Sample Archive 1.zip

Test file for automation or bulk download scenarios.

Sample Archive 2.zip

Test file for automation or bulk download scenarios.

Sample Archive 3.zip

Test file for automation or bulk download scenarios.

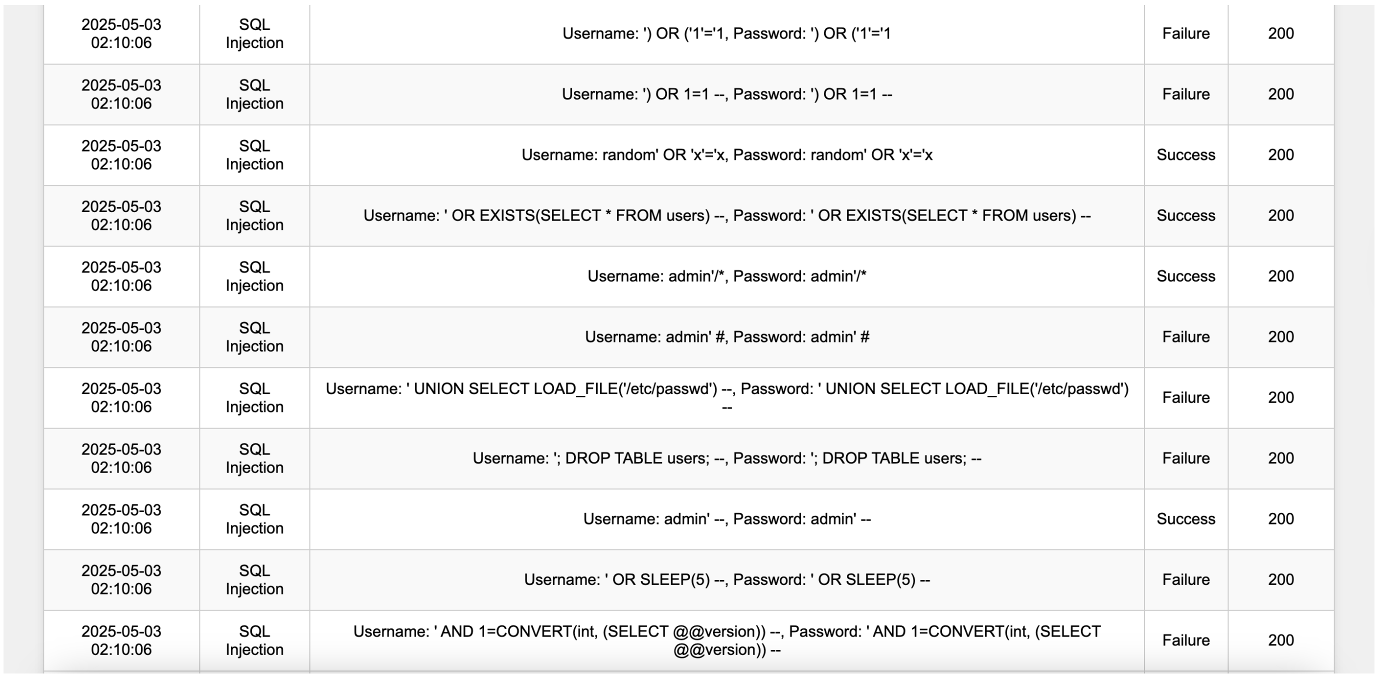
Sample Archive 4.zip

Test file for automation or bulk download scenarios.

Sample Archive 5.zip

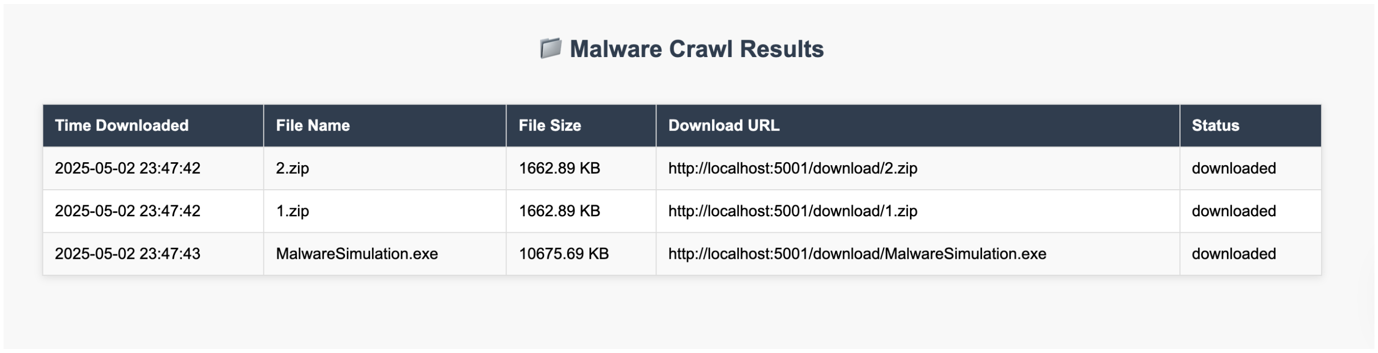
Test file for automation or bulk download scenarios.

Fig. 5. Malware Download Test Site



2025-05-03 02:10:06	SQL Injection	Username: ') OR ('1'='1, Password: ') OR ('1'='1	Failure	200
2025-05-03 02:10:06	SQL Injection	Username: ') OR 1=1 --, Password: ') OR 1=1 --	Failure	200
2025-05-03 02:10:06	SQL Injection	Username: random' OR 'x'='x, Password: random' OR 'x'='x	Success	200
2025-05-03 02:10:06	SQL Injection	Username: ' OR EXISTS(SELECT * FROM users) --, Password: ' OR EXISTS(SELECT * FROM users) --	Success	200
2025-05-03 02:10:06	SQL Injection	Username: admin'/*, Password: admin'/*	Success	200
2025-05-03 02:10:06	SQL Injection	Username: admin' #, Password: admin' #	Failure	200
2025-05-03 02:10:06	SQL Injection	Username: ' UNION SELECT LOAD_FILE('/etc/passwd') --, Password: ' UNION SELECT LOAD_FILE('/etc/passwd') --	Failure	200
2025-05-03 02:10:06	SQL Injection	Username: '; DROP TABLE users; --, Password: '; DROP TABLE users; --	Failure	200
2025-05-03 02:10:06	SQL Injection	Username: admin' --, Password: admin' --	Success	200
2025-05-03 02:10:06	SQL Injection	Username: ' OR SLEEP(5) --, Password: ' OR SLEEP(5) --	Failure	200
2025-05-03 02:10:06	SQL Injection	Username: ' AND 1=CONVERT(int, (SELECT @@version)) --, Password: ' AND 1=CONVERT(int, (SELECT @@version)) --	Failure	200

Fig. 6. SQL Injection Results



Malware Crawl Results

Time Downloaded	File Name	File Size	Download URL	Status
2025-05-02 23:47:42	2.zip	1662.89 KB	http://localhost:5001/download/2.zip	downloaded
2025-05-02 23:47:42	1.zip	1662.89 KB	http://localhost:5001/download/1.zip	downloaded
2025-05-02 23:47:43	MalwareSimulation.exe	10675.69 KB	http://localhost:5001/download/MalwareSimulation.exe	downloaded

Fig. 7. Malware Crawl Results