An Efficient Black-box Technique for Defeating Web Application Attacks

R. Sekar
Stony Brook University

(Research supported by DARPA, NSF and ONR)
Example: SquirrelMail Command Injection

- Attack: use maliciously crafted input to exert unintended control over output operations.
- Detect “exertion of control”
  - Based on “taint” degree to which output depends on input.
- Detect if control intended:
  - Requires policies
    - Application-independent policies are preferable.

Incoming Request
(Untrusted input)

Program

Outgoing Request/Response
(Security-sensitive operations)
(To databases, backend servers, command interpreters, files, ...)

\$send_to_list = 
\$_GET['sendto']

\$command = “gpg 
-r \$send_to_list
2>&1”

\$command="gpg –r
nobody; rm–rf *
2>&1”

popen($command)

popen($command)

Attack: Removes files

Send to = “nobody; rm–rf ”

Removes files

\$send_to_list = 
$_GET['sendto']

Program

Incoming Request
(Untrusted input)
Attack Space of Interest (CVE 2006-07)

- SQL injection: 14%
- Command injection: 18%
- Cross-site scripting: 19%
- Directory traversal: 4%
- Input validation/DoS: 9%
- Others: 24%
- Config/Race errors: 1%
- Memory errors: 10%
- Format string: 1%
- Generalized Injection Attacks
Drawbacks of Taint-Tracking and Motivation for Our Approach

- **Intrusive instrumentation**
  - Transform every statement in target application
  - Can potentially impact stability and robustness
- **High performance overheads**
  - Often slow down programs by 2x or more
- **Language dependence**
  - E.g., they apply either to Java or C/C++
Approach Overview

- Efficient, language-neutral, and non-intrusive
- Consists of
  - **Taint-inference**: Black-box technique to infer taint by observing inputs and outputs of protected apps
  - **Syntax- and Taint-aware policies** for detecting unintended use of tainted data
Syntax Analysis: Input Parsing

- **Inputs:**
  - Parse into components
    - Request type, URL, form parameters, cookies, ...
    - Exposes more of protocol semantics to other phases
    - All information mapped to (name, value) pairs
  - Normalize formats to avoid effect of various encoding schemes
    - To cope with evasion techniques
    - To ensure accuracy of taint-inference
  - Our implementation uses ModSecurity code
Syntax Tree Construction

- **Outputs:**
  - Pluggable architecture to parse different output languages
    - HTML, SQL, Shell scripts, ...
  - Use “rough” parsing, since accurate parsers are:
    - time-consuming to write
    - may not gracefully handle:
      - errors (especially common in HTML), or
      - language extensions and variations (different shells, different flavors of SQL)
  - Map to a language-neutral representation
  - Implemented using standard tools (Flex/Bison)
Taint Inference

- Infer taint by observing inputs and outputs
- Allow for simple transformations that are common in web applications
  - Space removal (or replacement with “_”)
  - Upper-to-lower case transformation, quoting or unescaping, ...
- Other application-specific changes
  - SquirrelMail, when given the “to” field value "alice, bob; touch /tmp/a" produces an output "-r alice@ -r bob; touch /tmp/a"
- Solution: use approximate substring matching
Taint Inference Algorithm

- Standard approximate substring matching algorithms have quadratic time and space complexity
  - Too high, since inputs and outputs can be quite large
- Our contribution
  - A linear-time “coarse-filtering” algorithm
    - More expensive edit-distance algorithm invoked on substrings selected by coarse-filtering algorithm
    - The combination is effectively linear-time
  - Ensures taint identification if distance between two strings is below a user-specified threshold $d$
    - Contrast with biological computing tools that provide speed up heuristics, but no such guarantee
Coarse-filtering to speed up Taint Inference

- **Definition of taint:**
  - A substring $u$ of $t$ is tainted if $ED(s, u) < d$
  - Here, $ED$ denotes the edit-distance

- **Key idea for coarse-filtering:**
  - Approximate $ED$ by $ED^\#$, defined on length $|s|$ substrings of $t$
  - Let $U$ (and $V$) denote a multiset of characters in $u$ (resp., $v$)
  - $ED^\#(u, v) = \min(|U-V|, |V-U|)$
  - Slide a window of size $|s|$ over $t$, compute $ED^\#$ incrementally
  - Prove: $ED(s, r) < d \Rightarrow ED^\#(s, r) < d$ for all substrings $r$ of $t$

- **Result:**
  - $O(|s|^2)$ space in worst-case
  - performs like a linear-time algorithm in practice
Overview of Syntax+Taint-aware Policies

- Leverage structure+taint to simplify/generalize policy
  - Policy structure mirrors that of syntax trees
    - And-Or “trees” (possibly with cycles)
  - Can specify constraints on values (using regular expressions) and taint associated with a parse tree node

1. Policy for detecting XSS
Injection attacks and Syntax-aware policies

- (2) SpanNodes policy: captures “lexical confinement”
  - tainted data to be contained within a single tree node
- (3) StraddleTrees policy: captures “overflows”
- Both are “default deny” policies
  - Tainted data begins in the middle of one syntactic structure (subtree), then flows into next subtree
Further Optimization: Pruning Policies

- Most inputs are benign, and cannot lead to violation of policies
  - Policies constrain tainted content, which comes from input
  - Thus, policies implicitly constrain inputs

- Approach:
  - Define “pruning policies” that make these implicit constraints explicit
  - Pruning policies identify subset of inputs that can possibly lead to policy violation
  - For other inputs, we can skip taint inference as well as policy checking algorithms
### Evaluation: Applications and Policies

<table>
<thead>
<tr>
<th>Application</th>
<th>Language</th>
<th>LOC (Size)</th>
<th>Environment</th>
<th>Attacks</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>phpBB</td>
<td>PHP/C</td>
<td>34K</td>
<td>Apache or IIS w/MySQL</td>
<td>SQL inj</td>
<td>Popular real-world apps. Exploits from the wild.</td>
</tr>
<tr>
<td>SquirrelMail</td>
<td>PHP/C</td>
<td>35K/42K</td>
<td>Apache or IIS</td>
<td>Shell command inj, XSS</td>
<td>Samples from the wild.</td>
</tr>
<tr>
<td>XMLRPC (library)</td>
<td>PHP/C</td>
<td>2K</td>
<td>Apache or IIS</td>
<td>PHP command inj</td>
<td></td>
</tr>
<tr>
<td>Apps from gotocode.com</td>
<td>Java/C</td>
<td>30K</td>
<td>Apache+Tomcat w/MySQL</td>
<td>SQL inj (21K attacks, 4K legitimate)</td>
<td>Attacks by [Halfond et al]</td>
</tr>
<tr>
<td>WebGoat</td>
<td>Java/C</td>
<td></td>
<td>Tomcat</td>
<td>command inj, HTTP response splitting</td>
<td></td>
</tr>
<tr>
<td>DARPA RedTeam App</td>
<td>PHP</td>
<td>2K</td>
<td>Apache</td>
<td>SQL inj</td>
<td>App developed by Red Team</td>
</tr>
</tbody>
</table>

- We used the 3 policies described earlier in the talk.
False Negatives (and Detection Results)

- Occur due to
  - Complex application-specific data transformations
    - Protocol/language-specific transformations handled
  - Second-order attacks (data written into persistent store, read back subsequently, and used in security-sensitive operations)
    - A limitation common to taint-based approaches

- Experimental results:
  - Detected all attacks in experiments with the exception of a single second-order injection attack in Red Team evaluation
    - Shell and PHP command injections and XSS on
    - ~21K SQL injection attacks on 5 moderate-size JSP applications (AMNESIA [Halfond et al] dataset)
    - HTTP response splitting on WebGoat
False Positives

- Result of coincidental matches (in taint-inference)
  - Can be controlled by setting the distance threshold $d$ based on the desired false positive probability
  - Likelihood small even for short strings
  - No false positives reported in experiments

- Implication
  - Can use large distances for moderate-size strings ($\text{len} > 10$), thus tolerating significant input transformations

![Graph showing the relationship between distance threshold and false positive rate]
Taint inference overhead

- Coarse filtering optimization
  - 10x to 20x improvement in speed in experiments
  - 50x to 1000x reduction in space
  - Time spent in coarse filtering (linear-time algorithm) exceeds time spent inside edit-distance algorithm
  - Performance decreases with large values of distance
    - When coincidental probability increases beyond $10^{-6}$
Overhead of different phases

- 60% spent in taint inference
  - After coarse-filtering optimization
- 20% in parsing
- 20% in policy checking
- Overhead of interposition not measured
  - but assumed to be relatively small because of reliance on library interposition
End-to-end Performance Overhead

- Measured using AMNESIA [Halfond et al] dataset on utility applications from gotocode.com
- Performance measured with pruning filters deployed
  - ~5x performance improvement due to pruning

<table>
<thead>
<tr>
<th>Application</th>
<th>Size (LOC)</th>
<th># of Requests</th>
<th>Response time (sec)</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bookstore</td>
<td>9552</td>
<td>605</td>
<td>20.7</td>
<td>1.7%</td>
</tr>
<tr>
<td>Empldir</td>
<td>3028</td>
<td>660</td>
<td>17.3</td>
<td>3.4%</td>
</tr>
<tr>
<td>Portal</td>
<td>8775</td>
<td>1080</td>
<td>31.7</td>
<td>5.1%</td>
</tr>
<tr>
<td>Classifieds</td>
<td>5726</td>
<td>576</td>
<td>18.0</td>
<td>4.3%</td>
</tr>
<tr>
<td>Events</td>
<td>3805</td>
<td>900</td>
<td>23.0</td>
<td>3.1%</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>30886</strong></td>
<td><strong>3821</strong></td>
<td><strong>110.7</strong></td>
<td><strong>3.5%</strong></td>
</tr>
</tbody>
</table>
Related Work

- Su and Wasserman [2006]
  - Focus on formal characterization of SQL injection
  - Our contributions
    - A robust, application-independent technique to infer taint propagation
    - Policies decoupled from grammar
      - Applicable to many languages

- Dataflow anomaly detection [Bhatkar et al 2006]
  - Flow inference algorithms tuned for simpler data (file names, file descriptors, ...)

- Program transformations for taint-tracking
  - And related approaches (AMNESIA, CANDID, ...)
  - Require deep analysis/instrumentation of applications
Summary

- A black-box alternative for taint-tracking on web applications
- A simple, language-neutral policy framework
- Ability to detect a wide range of exploits across different languages (Java, C, PHP, ...) and platforms (Apache, Tomcat, IIS, ...) with just a few general policies
- Low performance overheads (below 5%)