Finding Vulnerabilities in Web Applications

Christopher Kruegel
Secure Systems Lab, Technical University Vienna
• The past few years have witnessed a significant increase in the number of deployed web applications (and web services) as well as attacks targeting these systems.
Web Applications

- **Pervasive**
  - deployed by virtually all companies, institutions, and organizations
- **Critical**
  - access/manage sensitive information
- **Open**
  - widely accessible through firewalls
- **Dynamic**
  - change frequently
- **Vulnerable**
  - untrained developers, time to market pressure
Web Application Attacks

• Significant fraction of reported security flaws
  – 26% of entries in CVE database
  – Snort 2.3, 1006 of 2564 signatures

• Large variety of attack vectors
  – cross-site scripting, SQL injection, command injection
  – weak cookies or session management
  – bypassing client-side validation

• Bug finding is difficult and tedious
  – automated solutions are necessary
Taint-Style Vulnerabilities

• Tainted data
  – potentially malicious data enters the program at specific points
  – data propagated through program
  – may reach sensitive sink, where it can lead to security problem

• Models important class of security flaws
  – cross-site scripting (XSS)
    • tainted data might enter through GET or POST requests
    • reaches an output function (which returns data to the user's browser)
  – SQL injection
  – command injection

• Attack can be prevented using sanitization functions
Web Application Attacks

- Cross-site scripting (XSS)
  - attacker injects HTML or Javascript into application output which is displayed and executed in victims browser
  - reflected vs. stored XSS variants
  - stealing of user data (cookies, credentials…)
  - redirecting login form to hackers web server
  - create exploit URLs and send phishing mails
Web Application Attacks

- SQL Injection
  - no validation before using input in database query
  - dynamically built SQL query problematic
    
    \[ q = "select * from user where
    mail="' + mail + "' and pw="' + pw + "'\"
    
  - attacker sends value(s) with SQL keywords
    
    mail: ` or 1=1--    pw: ` or 1=1--

  - changes semantics of query
    
    \[ q = "select * from user where mail=' or 1=1--' and pw=' or 1=1--'\"

  - actual query
    
    \[ q = "select * from user"\]
Bug Finding

- **Black box approach**
  - send input to system and evaluate response
  - every bug found is real
  - large number of programs can be checked (language-independent)
  - can only cover a certain number of input (test cases)

- **White box approach**
  - analyze source code
  - better precision, more bugs can be found
  - false positives
  - limited to supported programming languages
Black Box Solution

Where to start
- locate vulnerable application(s)
- use user-defined list
- crawl pages, start from seed address, follow (on-site) links
- identify forms

What to do
- attack each form parameter
- user-selectable attack plug-ins
- analysis of the responses
- decision whether form / page is vulnerable
- possible generation of exploit code
Secure Systems Lab

Attack Plug-ins

• SQL injection
  – inject single quote ‘ into form parameter
  – assumption
    • unfiltered input leads to syntactically incorrect SQL statement
  – response analyzed for occurrence of suspicious keywords
    • indication for errors, exceptions, …
  – keywords receive weights
  – total confidence factor calculated
Attack Plug-ins

• Cross-site scripting
  – works only for reflected attacks
  – inject simple script
  – assumption
    • unfiltered input is included in response
  – response analyzed for presence of injected script
    • also make sure that script is executable
  – different scripts injected that make use of script obfuscation
    • lowercase/uppercase characters, different character encodings, …
Results

- Evaluation run
  - seeded with Google results for "login" search
  - 25,064 crawled sites
  - 21,627 web forms

- 4 attack plug-ins used
  - SQL Injection: 6.63%
  - Simple XSS: 4.30%
  - Enhanced XSS: 5.60%
  - Form-Red. XSS: 5.52%

- Some vulnerable sites
  - eBay, Austrian Finance Ministry, Geizhals (price management),
  - Crit.org (security associated content), Apple (developer access)
Results

• Query vulnerable site using \texttt{Whois} service

• Notify site administrator
  – send one mail per subdomain
  – 591 mails sent (239 default office@...)

• Administrator response
  – 306 “recipient unknown“
  – 48 detail inquiries
  – eBay gave us a call :)

• General impression
  – large scale probing for vulnerable services quite possible
  – at the time, few people were really concerned
White Box Solution

• Data flow analysis (DFA)
  – static analysis technique
  – operates on program’s control flow graph
  – defines properties
    • lattice of elements (what values can properties take)
    • transfer functions (how do operations change properties)
  – for each program point and variable
    • what values of property are possible (over all program paths)

• Allows to answer the following question
  – Is it possible that tainted data reaches a sensitive sink?
Pixy

- **Pixy**
  - static analysis engine for detecting taint-style vulnerabilities
  - runs on PHP scripts
  - currently, XSS detection implemented
  - high precision
  - fast
  - open source

- **Interesting challenges**
  - weak typing of scripting languages
  - use of aliases (instead of pointers)
  - *scalability despite precise analysis*
Pixy’s Taint Analysis

• Taint Analysis
  – at the core
  – determine which variables might be tainted
    (for each program point)

• Effects of built-in functions modeled in a configuration file
  – specification of the returned taint value depending on the input parameters
  – e.g., \( \text{nl2br}($in) \): returns tainted if $in is tainted
  – of course, not application-specific (nothing to do for users of Pixy)
  – default: return tainted (\( \rightarrow \) safe)
Pixy’s Alias Analysis

- Problem with stand-alone taint analysis
  - no information about alias relationships
    - explicit aliasing: \$a = \& \$b
    - call-by-reference parameters
    - access to global variables (\texttt{global} keyword)
  - can lead to false positives and false negatives

→ support taint analysis with additional alias analysis

- First application of alias analysis to scripting languages ever
Pixy’s Literal Analysis

• Additional precision through literal analysis
  – path pruning
  – variable file inclusions

• Potential applications
  – variable variables
  – variable array indices
  – variable function calls
Resolving Includes

• File inclusions in scripting languages
  – can be tricky, because
    dynamic, conditional, and even recursive

• Solution
  – use iterative algorithm consisting of two steps
  – first stage
    transitively resolve static includes
  – second stage
    if there are any dynamic includes, resolve them with literal analysis
  – repeat as long as there are includes left
  – fast, yields good results, easy to implement
Analysis Precision

• All applied analyses are
  – flow-sensitive
  – inter-procedural
  – context-sensitive

→ high precision
  (in addition to mutual support between the analyses)

• Limitations
  – object-oriented features are treated in an optimistic way
  – no support for aliases between arrays
Results

• Scanned six real-world applications

<table>
<thead>
<tr>
<th>Program</th>
<th>Lines of Code</th>
<th>Vulnerabilities</th>
</tr>
</thead>
<tbody>
<tr>
<td>PhpNuke 6.9</td>
<td>17,479</td>
<td>24</td>
</tr>
<tr>
<td>PhpMyAdmin 2.6.0</td>
<td>89</td>
<td>9</td>
</tr>
<tr>
<td>Gallery 1.3.3</td>
<td>3,529</td>
<td>3</td>
</tr>
<tr>
<td>Simple PHP Blog 0.4.5</td>
<td>20,792</td>
<td>8</td>
</tr>
<tr>
<td>Serendipity 0.8.4</td>
<td>6,588</td>
<td>2</td>
</tr>
<tr>
<td>Yapig 0.95b</td>
<td>5,128</td>
<td>5</td>
</tr>
</tbody>
</table>

• Vulnerabilities: 51 (15 previously unknown)
• False positives: 33
• Less than a minute for every scanned file
Results

• Breakdown of false positives
  13: conservative treatment of file reads
  7: aliasing between arrays
  6: partial sanitization (double quotes)
  5: syntactically incomplete branches
  2: custom regular expression sanitization
Conclusions

• Web application vulnerabilities
  – many widely-deployed and accessible applications
  – many vulnerabilities

• Bug finding is important
  – black box and white box approaches

• Black box approach
  – language independent
  – few false positives

• White box approach
  – high precision