Static Detection of Web Application Vulnerabilities

Vitaly Shmatikov
Reading Assignment

Pixy

[Jovanovic, Kruegel, Kirda]

- Uses static analysis to detect cross-site scripting and SQL injection vulnerabilities in PHP apps
  - Same ideas apply to other languages

- Basic idea: identify whether “tainted” values can reach “sensitive” points in the program
  - Tainted values: inputs that come from the user (should always be treated as potentially malicious)
  - Sensitive “sink”: any point in the program where a value is displayed as part of HTML page (XSS) or passed to the database back-end (SQL injection)
Example of Injection Vulnerabilities

```php
function postcomment($id, $title) {
    ...
    $title = urldecode($title);    // tainted
    ...
    echo $title;                   // sensitive sink
    ...
}
```

```php
if (...) {
    $entry = $_GET['entry'];
    ...
    $temp_file_name = $entry;
    ...
} else {
    ...
    $temp_file_name = 
        stripslashes($_POST['file_name']);
    ...
}
```

Main Static Analysis Issues

◆ Taint analysis
  • Determine, at each program point, whether a given variable holds unsanitized user input

◆ Data flow analysis
  • Trace propagation of values through the program

◆ Alias analysis
  • Determine when two variables refer to the same memory location (why is this important?)

◆ Pixy: flow-sensitive, context-sensitive, interprocedural analysis (what does this mean?)
Handling Imprecision

- Static data flow analysis is necessarily imprecise (why?)
- Maintain a lattice of possible values
  - Most precise at the bottom, least precise ($\Omega$) at the top
- Example from the paper
  ```
  v = 3;
  if (some condition on user input)
      v = 3;
  else
      v = 4;
  ```
Annotated Control-Flow Graph

Carrier lattice
Data Flow Analysis in PHP

- PHP is untyped; this makes things difficult
- How do we tell that a variable holds an array?
  - Natural: when it is indexed somewhere in program
  - What about this code?
    ```php
    $a[1] = 7;  $b = $a;  $c = $b;  echo $c[1];
    ```
- Assignments to arrays and array elements
  - `$a = $b;  // ... where $a is an array`
  - `$a[1][2][3] = ...`
  - `$a[1][$b[$i]] = ...`
Other Difficulties

- Aliases (different names for same memory loc)
  
  $a = 1; \; \; b = 2; \; \; b = &a; \; \; a = 3; \; \; \// \; b == 3, \; too!

- Interprocedural analysis
  
  - How to distinguish variables with the same name in different instances of a recursive function?

```php
1:  function f1() {
2:    // when entering this function, the local variables $a and $b
3:    // do NOT point to the same memory location
4:    $a; $b;
5:    // after the following statement, $a and $b DO point to the same memory location,
6:    // but this must not affect $a and $b in other incarnations of this function
7:    $a = &b;
8:    //
9: 10: if (...) f1();
11: }
```

What is the depth of this recursion?
Modeling Function Calls

◆ Call preparation
  • Formal parameter ← actual argument
    - Similar to assignment
  • Local variables ← default values

◆ Call return
  • Reset local variables
  • For pass-by-reference parameters,
    actual argument ← formal parameter
    - What if the formal parameter has an alias inside function?
  • What about built-in PHP functions?
    - Model them as returning Ω, set by-reference params to Ω
Taint Analysis

◆ Literal – always untainted
◆ Variable holding user input – tainted
  • Use data flow analysis to track propagation of tainted values to other variables
◆ A tainted variable can become untainted
  • $a = \langle\text{user input}\rangle; \ a = \text{array}();$
  • Certain built-in PHP functions
    – htmlentities(), htmlspecialchars() – what do they do?
False Positives in Pixy

- Dynamically initialized global variables
  - When does this situation arise?
  - Pixy conservatively treats them as tainted

- Reading from files
  - Pixy conservatively treats all files as tainted

- Global arrays sanitized inside functions
  - Pixy doesn’t track aliasing for arrays and array elements

- Custom sanitization
  - PhpNuke: remove double quotes from user-originated inputs, output them as attributes of HTML tags – is this safe? why?
Wassermann-Su Approach

- Focuses on SQL injection vulnerabilities
- Soundness
  - Tool is guaranteed to find all vulnerabilities
  - Is Pixy sound?
- Precision
  - Models semantics of sanitization functions
  - Models the structure of the SQL query into which untrusted user inputs are fed
  - How is this different from tools like Pixy?
"Essence" of SQL Injection

◆ Web app provides a template for the SQL query
◆ Attack = any query in which user input changes the intended structure of SQL query
◆ Model strings as context-free grammars (CFG)
  • Track non-terminals representing tainted input
◆ Model string operations as language transducers
  • Example: `str_replace("'" ', "'" , $input)`

\[
\begin{array}{c}
1 \xrightarrow{\varepsilon} \quad 1 \xrightarrow{'} \quad 1 \xrightarrow{'} \quad 1 \\
\quad 2 \xrightarrow{'} \quad 2 \xrightarrow{'} \quad 2 \\
\quad 3 \\
\end{array}
\]

A matches any char except "'"
Phase One: Grammar Production

Generate annotated CFG representing set of all query strings that program can generate

Example code with an SQLCTIV.
String Analysis + Taint Analysis

- Convert program into static single assignment form, then into CFG
  - Reflects data dependencies

- Model PHP filters as string transducers
  - Some filters are more complex:
    - `preg_replace("/a([0-9]*)b/", "x\1\1y", "a01ba3b")` produces "x0101yx33y"

- Propagate taint annotations
Phase Two: Checking Safety

◆ Check whether the language represented by CFG contains unsafe queries

  • Is it syntactically contained in the language defined by the application's query template?

```
query → query1'
query1 → query2 userid
query2 → query3 WHERE userid='
query3 → SELECT * FROM `unp_user`
userid → GETuid
GETuid → Σ* [0–9] Σ*

direct = {GETuid} indirect = {}
```

This non-terminal represents tainted input

For all sentences of the form $\sigma_1 \text{GETUID} \sigma_2$ derivable from query, GETUID is between quotes in the position of an SQL string literal (means what?)

Safety check:
Does the language rooted in GETUID contain unescaped quotes?
Tainted Substrings as SQL Literals

- Tainted substrings that cannot be syntactically confined in any SQL query
  - Any string with an odd # of unescaped quotes (why?)
- Nonterminals that occur only in the syntactic position of SQL string literals
  - Can an unconfined string be derived from it?
- Nonterminals that derive numeric literals only
- Remaining nonterminals in literal position can produce a non-numeric string outside quotes
  - Probably an SQL injection vulnerability
  - Test if it can derive DROP WHERE, --, etc.
Taints in Non-Literal Positions

- Remaining tainted nonterminals appear as non-literals in SQL query generated by the application
  - This is rare (why?)

- All derivable strings should be proper SQL statements
  - Context-free language inclusion is undecidable
  - Approximate by checking whether each derivable string is also derivable from a nonterminal in the SQL grammar
    - Variation on a standard algorithm
Evaluation

- Testing on five real-world PHP applications
- Discovered previously unknown vulnerabilities, including non-trivial ones
  - Vulnerability in e107 content management system: a field is read from a user-modifiable cookie, used in a query in a different file
- 21% false positive rate
  - What are the sources of false positives?
Example of a False Positive

```php
isset($_GET['newsid']) ?
    $getnewsid = $_GET['newsid'];
    $getnewsid = false;
if (($getnewsid !== false) &&
    (!preg_match('/^[\d]+$/', $getnewsid)))
{
    unp_msg('You entered an invalid news ID.');
    exit;
}
...
if (!$showall && $getnewsid)
{
    $getnews = $DB->query("SELECT * FROM `unp_news`
    ."WHERE `newsid`='{$getnewsid}'`
    ."ORDER BY `date`DESC LIMIT 1");
}
```