NANYANG TECHNOLOGICAL UNIVERSITY

Lecture 10: Taint Analysis

presented by

Li Yi
Assistant Professor
SCSE

N4-02b-64
yi_li@ntu.edu.sg

Agenda

- Principles of Taint Analysis
- Case Study: PHP
- Data flow analysis – taint analysis for integrity
- Information flow analysis – taint analysis for confidentiality

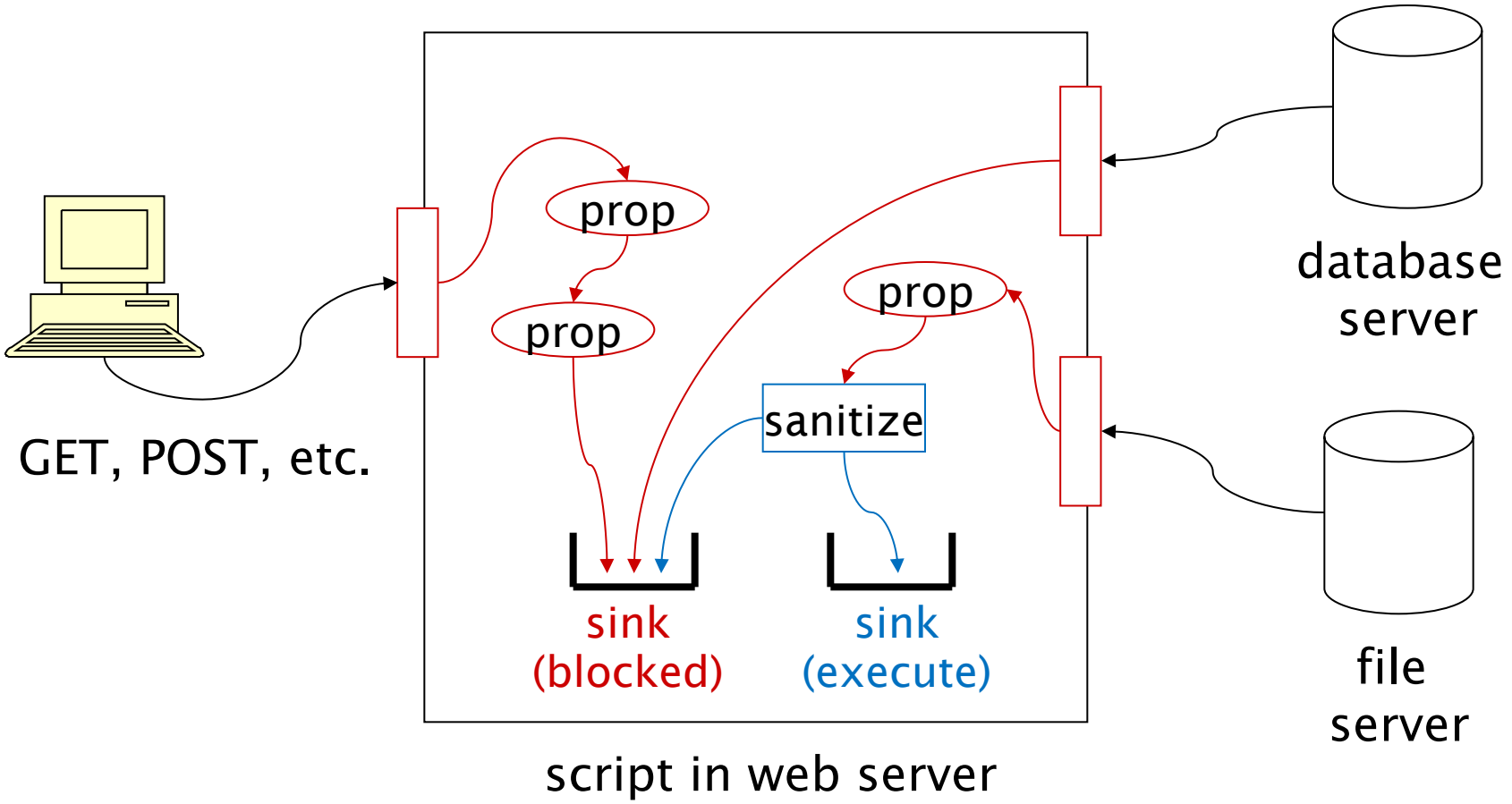
Data Tainting – "Precise Filtering"

- To filter/encode/escape dangerous characters, we must know what is dangerous
- Depends on protocol, programming language, etc.
- Difficult/impossible to have a universal filter that catches all dangerous inputs
- Alternative: track inputs from the time they enter a module until they are consumed by a "trust sink"
- Data tainting automatically checks whether "tainted" input is passed to a sensitive command without prior sanitization

Principle of Tainting

- Mark data coming from untrusted sources as tainted
- Tainted data may spread across a program through propagation functions (assignments, etc.)
- Certain operations can sanitize (clean up) tainted data; different attacks require different sanitizers
  - A sanitization function may miss some attacks; in such a case additional measures have to be taken
- Sensitive sinks must never use tainted data
- Perform a data flow analysis to check that sensitive operations (sinks) do not receive tainted data

Principle of Tainting (Web Server)



Tainting – Example (Perl)

- Example in Perl with tainted input

```
use strict;
my $filename = <STDIN>;
open (FILENAME, ">>". $filename) or die $!;
print FILENAME "Hello!";
close FILENAME;
```

- If running with "-T" (taint option):

```
  Error: Insecure dependency in open while
running with -T switch at testtaint.pl line 3,
<STDIN> line 1
```

Categories of Operations

- Propagators: functions that propagate tainted data to other variables
  - Same propagators for all attack patterns
- Sanitizers: functions making tainted data safe to use
  - Different attacks require different sanitizers
- Sensitive sinks: functions that access the file system, the database, or output information to the user
  - E.g., create/open/remove file, connect to a server, generate HTML, execute shell or database commands, or change a script's state, e.g., by including a remote file
  - Different attacks have different sensitive sinks

Dynamic & Static Tainting

- Dynamic tainting performed at runtime; necessary checks are normally included by the compiler
  - Can capture data dependent taint propagation
  - Limited to code paths that are actually executed
  - Can significantly reduce performance; each variable access needs special care with regard to tainting
- Static tainting applied to source code at compile time
  - Can protect applications before actually running them; problems can be eliminated before deploying the code
  - Can examine code paths that are rarely executed, but its understanding of those code paths is limited

Data Flow or Information Flow Analysis

- Taint analysis can be done to address injection attacks but also to address leakage of sensitive data
- Code injection attacks → data flow analysis
  - Typical concern for server-side tainting
  - Code-injection attacks may also be an issue on the client-side (DOM-based XSS)
- Leaking of sensitive information → information flow analysis
  - Typical concern for client-side tainting (cookie stealing)
- Sources, propagation functions, and trust sinks will be different, the general principle remains the same

PHP Tainting

Case study

PHP

- Open-source, extensible server-side scripting language for producing dynamic web pages
- Weakly typed: variables do not have an explicit type, can change type, need not be declared before use
- PHP code embedded in HTML documents with an opening <?php and a closing ?> tag
- Works with most operating systems (Windows, Linux, ...), web servers (Apache, IIS, …), database systems (MySQL, PostgreSQL, Firebird, MSSQL, …)

PHP – Inputs to Scripts

- Typically sent from an HTML form; action gives the PHP script, parameters entered in form:
  - `<form action="example.php" method="post">`
    parameters passed in body of POST request
  - `<form action="example.php" method="get">`
    parameters entered in form, passed in URL
- Parameters passed directly in a link:

```
<a href="example.php?var1=value1&var2=value2">
  text in link
</a>
```

- Superglobals arrays: predefined arrays for storing variables from external resources

Superglobals Arrays

`$_GET`	stores all HTTP GET variables received from the web browser
`$_POST`	stores all POST variables received from the form submitted by the client browser
`$_SERVER`	stores information such as headers, paths, script locations; entries created by web server
`$_COOKIE`	associative array of variables passed to the current script via HTTP cookies
`$_FILES`	array of items uploaded to the current script via the HTTP POST method
`$_REQUEST`	all variables from `$_GET`, `$_POST`, `$_COOKIE`
`$_SESSION`	variables associated with the user session

Sources of Tainted Data

- All inputs to the script are tainted; we have to identify all sources of tainted data
- All data in `$_GET`, `$_POST`, `$_COOKIES`, `$_SERVER` superglobals arrays is tainted
- Data from internal sources such as database and files is tainted

Propagation Functions

- Mainly string manipulation and database functions
- PHP works with several databases; every database has a specific set of functions to send, retrieve data

Type	Functions
Functions that return tainted result depending on the input	`substr()`, `str_replace()`, `preg_replace()`, etc
Functions that always return tainted result	`mysql_fetch_array()`, `mysql_fetch_assoc()`, `mysql_fetch_row()`, `file()`, `fread()`, `fscanf()`, etc.

PHP Strings

- PHP: single and double quotes as string delimiters
- Variable names in single quoted strings are interpreted as strings
- Variables in double quoted strings are evaluated; variables are replaced by their values

```
<?php
  $num = 1;
  echo 'Display the number $num';
  echo "Display the number $num";
  // Result 1: "Display the number $num"
  // Result 2: "Display the number 1"
?>
```

Propagation in Strings

- Taint can propagate through double quoted strings

```
$num = $_GET['num'];
$str = "The number is $num";
```

- String between the double-quotes is evaluated; variable **$num** will be replaced with the value from **$_GET['num']**; the result is then also tainted

Propagation in Strings

- Taint propagation through other PHP string manipulation functions:
  - **`substr()`** (substring): if the input of this function is tainted, the result is also tainted
  - **`str_replace()`**: replaces all occurrences of the search string with the replacement string
  - String concatenation by "." : **`$str = $str1 . $str2;`**

    if one string is tainted, the concatenation is also tainted

Propagation Functions

- Functions that always return tainted result:
  - E.g., **`mysql_fetch_assoc()`** fetches a result row from a SQL query as an associative array
  - Example next slide
- Functions that retrieve data from the file system:
  - E.g., function **`file()`** reads an entire file into an array; each array element represents a line in the file; in this case, each element in this array is tainted

Database Propagation Functions

```
$sql = "SELECT article_name,
        article_content
        FROM articles WHERE id = 1";
$result = mysql_query($sql);
$row = mysql_fetch_assoc($result);
$article_name = $row["article_name"];
$article_content = $row["article_content"];
```

- Retrieves an article from a database and outputs its name and content
- **`$article_name`** and **`$article_content`** are tainted as they depend on input from the database

Sanitization Functions

- Clean up input data, return untainted results
- Sanitization functions specific for different attacks

Attack	Sanitization functions
XSS	**`htmlspecialchars()`**, **`htmlentities()`**, **`strip_tags()`**
Shell Command Injection	**`escapeshellcmd()`**, **`escapeshellarg()`**
SQL Injection	int type cast, **`mysql_escape_string()`**, **`mysql_real_escape_string()`**,
Code Injection	No filter function that makes all data safe as input for **`eval()`**, **`include()`**

XSS Sanitizers

- **`htmlentities()`** converts characters that have special meaning in HTML to HTML entities
  - Prevents user-supplied text from containing HTML markup, such as in a message board or guest book application
  - User-supplied HTML markup can still be preserved
- **`strip_tags()`** explicitly strips tags from HTML markups
  - Strips all HTML and PHP tags from a string

SQL Injection Sanitizers

- **`mysql_escape_string()`, `mysql_real_escape_string()`**

  add backslash in front of single quotes, double quotes, and other characters that may be used to break out of a user input

- Applying **`mysql_escape_string()`** to the input

  `"john'; DELETE FROM users; --",`

  adds a backslash in front of the single quote

  `"john\'; DELETE FROM users; --",`

Vulnerable Code – Example

```
<?php
   $customer_id= $_GET['c_id'];
   $sql="SELECT * FROM customers WHERE
            customer_id=".$customer_id;
   mysql_query($sql);
?>
```

- Input `c_id = 1; DELETE FROM customers`

  gives `SELECT * FROM customers WHERE customer_id=1;`

  `DELETE FROM customers;`

Type Conversion as Sanitizer

```
<?php
    $customer_id= $_GET['c_id'];
    $sql="SELECT * FROM customers WHERE
        customer_id=".(int)$customer_id;
    mysql_query($sql);
?>
```

- Type cast of **`$customer_id`** will keep the `1`, but remove the string

Shell Command Sanitizers

- Must be invoked before arguments are passed to system calls like **`system()`**, **`exec()`**, **`passthru()`**
- Sanitizers remove harmful characters from user input that is passed as argument to a system command
- **`escapeshellarg()`** for strings used as shell arguments; adds single quotes around the string and escapes (adds a backslash in front of) single quotes within the string
- **`escapeshellcmd()`** applied on a complete shell command; escapes characters that have a special meaning to the underlying operating system, e.g. the pipe character `" | "`

Shell Command Sanitizers – Example

- Send the following value to **`shell_exec()`**:

  `/usr/bin/wc /dev/null | cat /etc/shadow`

- **`escapeshellcmd()`** will add a backslash in front of "|", so the attack cannot obtain information from **`/etc/shadow`**

Input Filtering

- Set of standard filter functions for validating and sanitizing user supplied data
- Validation filters for integers, boolean, float, email addresses, etc; return a Boolean value to indicate whether an input value is valid
- Sanitization filters return a value that complies with the filter rules, not a Boolean value

Input Filtering – Examples

- Function **`filter_var()`** filters a single variable; **`FILTER_VALIDATE_INT`** defines an integer filter

```
<?php
    $product_id = $_GET['product_id'];
    if(filter_var($product_id,
                  FILTER_VALIDATE_INT))
    echo $product_id;
?>
```

- Integer filter validating **`$product_id`** retrieved from HTTP GET array; if it is a valid integer, **`echo()`** will output the variable

Sanitizing Filters

- Constant **`FILTER_SANITIZE_NUMBER_INT`** specifies the integer sanitizing filter as the parameter

```
<?php
   $product_id = $_GET['product_id'];
   echo filter_var($product_id,
        FILTER_SANITIZE_NUMBER_INT);
?>
```

- Filter returns a sanitized integer

Sensitive Sinks

- Functions that access the file system or the database system, or output information to the user
- Sensitive sinks depend on mode of attack

Attack Type	Sensitive sinks
XSS	`echo()`, `print()`, `printf()`, `mysql_query()`, etc
Shell Command Injection	`system()`, `exec()`, `passthru()`, `proc_open()`, `shell_exec()`
SQL Injection	`mysql_query()`, `mysqli_query()`
Code Injection	`include()`, `require()`, `eval()`, `preg_replace()`

Sensitive Sinks

- **`echo()`,`print()`,`printf()`** output data to client
  - XSS attacks can send malicious code from a database to client through these functions
- **`system()`**, **`exec()`**, **`passthru()`** execute operating system commands from within PHP scripts
  - Could allow an attacker to execute commands that access private files and information
- **`mysql_query()`** to insert or retrieve data from a DB
- **`include()`**, **`require()`** to include files in a script
- **`eval()`**, **`preg_replace()`** can evaluate a string and execute the string as a PHP code

Propagation Issues

Taint Propagation

- Tracking taint is "easy" with the taint propagation functions discussed so far
- We will now discuss some of the more challenging situations when tracking taint

Flow Sensitivity

- Variables declared in a script may be used several times; taint analysis must consider each program point

```
<?php
    $var = 'var1';
    echo $var;
    $var = $_GET['var'];
    echo $var;
?>
```

- **`$var`** is first initialized locally in the script with the string **`'var1'`** so **`$var`** is untainted
- Later **`$var`** is re-assigned a value from **`$_GET['var']`**, an external source; now **`$var`** must be marked as tainted

Context Sensitivity

- Two calls to `foo()` in the code snippet:

```
<?php
    $var_a = foo($_GET['var_a']);
    echo $var_a;
    $var_b = foo('ok');
    echo $var_b;
    function foo($tmp) {
        return $tmp;
    }
?>
```

- `foo()` first called with a tainted parameter that is returned and assigned to `$var_a`, whose use in `echo` should be flagged
- Second call of `foo()` uses a harmless value; should be allowed

Alias Analysis

- An alias of a variable (defined with the operator "=&") is a variable that refers to the same memory location
- Assigning a value to a variable writes this value to the variable's memory location, thus affecting all aliases of this variable

```
<?php
    $var = 'ok';
    $r_var =&$var;
    $var = $_GET['var'];
    echo $r_var;
?>
```

- Variables **$var** and **$r_var** are aliases; when **$var** is assigned a tainted value this value is also assigned to **$r_var**
- Hence, **echo** should be flagged as a vulnerability

File Inclusion

- PHP code may be split into several files merged at runtime with inclusion statements ("include","require")
- Included file may contain vulnerabilities, so tainting must automatically resolve inclusions
- Static file inclusion:

```
<?php // in file_a.php
    $x = 'ok';
    include('file_b.php');
    // in file_b.php
    // $x = $_GET['x'];
    echo $x;
?>
```

- Vulnerability, since **`$x`** gets tainted in **`file_b.php`**

Dynamic File Inclusion

- Dynamic file inclusion: included file can only be determined at runtime

```
<?php
    $name = 'file_b';
    $ext = '.php';
    include($name . $ext);
    echo $x;
?>
```

- Not sufficient to check the include statement; we also must know the values held in variables **$name**
- More complicated when string values are propagated across functions, defined constants, global variables, etc.

Information Flow Analysis

Information Flow Analysis

- With SQLI and XSS, taint analysis checks whether user-supplied data can be sent to sensitive sinks; there is no intention to protect sensitive user data
- Tainting can be used to prevent sensitive user data from being leaked to a third party, e.g., cookie stealing
  - Sometimes known as client-side tainting, but client-side tainting may also be used to detect code injection
- Similarities between tainting for injection and tainting for leakage exist, but there are also differences with respect to the sources of tainted data, the propagation functions, and the sensitive sinks

Client-Side Tainting

- First introduced in Netscape Navigator 3.01
- Another line of defence against XSS
  - Attacker's script passed by the server to the client; client tries to stop the script from leaking sensitive data to attacker
  - Script may use sensitive data only within the HTML page
- Sources of tainted inputs differ between tainting for injection attacks and tainting for extraction attack
  - Injection: tainted sources are the users-supplied data;
  - Extraction (leakage): tainted sources are data holding information about users
- Main sources: cookies, URL of visited web page, etc.

Sensitive Data Sources

Objects	Tainted Properties and Methods
Document	cookie, domain, forms[], lastModified, links[], location, referrer, title, URL
Form	action
All Form input elements: Button, Submit, Checkbox, FileUpload, Password, Radio, Hidden, Reset, Select, Text, Textarea	checked, defaultChecked, defaultValue, name, selectedIndex, toString(), value
History	current, next, previous, toString(), all array elements
Location, Link, Area	hash, host, hostname, href, pathname, port, protocol, search, toString()
Option	defaultSelected, selected, text, value
Window	defaultStatus, status

Sensitive Data Sources

- Each object in the table above represents an HTML element in the HTML document (DOM)
- Document object contains array properties specifying information about the contents of the document
  - Properties represent the cookie, links, anchors, HTML forms, applets, and embedded data contained in the document
- Form object represents an HTML form; users interact with web applications via form submissions
  - Action property stores URL the form has been submitted to
  - Contains input elements such as Text Fields, Checkbox, Dropdown list and Buttons etc.

Sensitive Data Sources

- Option object represents an option in a dropdown list in an HTML form
- History object stores the web browsers' history; contains the methods to navigate to previous or next pages the web browser had opened
- Location object represents current URL of document; changing this property redirects the page to another URL

Taint Propagation

- Values derived from tainted data elements are also tainted
- If a tainted value is passed to a function, the return value of the function is tainted
- If a string is tainted, any substring of the string is also tainted
- If a script examines a tainted value in an "`if`", "`for`", or "`while`" statement, the script itself becomes tainted
- Traditional concern in information flow security

Taint Propagation – Assignments

- In an assignment operation, if the right-hand side variable of the assignment is tainted, then the left-hand side variable is also tainted
- If the left-hand side variable is an array element and the right-hand side variable is tainted, then the whole array object is tainted
- If a property of an object is set to a tainted value, then the whole object is tainted

Taint Propagation – ALOs

Arithmetic and logic operations (+, −, &, etc.):

- In tainting for integrity, the result of a numeric operation is untainted since the result is a number which is not harmful to the system
  - With ternary operations (e.g., `c = (a > b) ? a : b`), only when a tainted value is assigned to the left-hand side variable then the variable is tainted
- Tainting for confidentiality: if one operand is tainted, then the result is tainted for all arithmetic operations, be they unary (e.g., `a++`), binary (e.g. `a*b`) or ternary

Conditional Expressions

- If the condition of a control structure (`if`, `while`, `switch`) contains the test of a tainted value, then the entire control structure is a tainted scope
  - All operations & assignments result in this scope are tainted
  - A variable is dynamically tainted when its value is modified inside a scope during program execution

```
if (document.cookie[0] == 'a'){
     x = true;
}
```

- document.cookie[0] used in `if` condition is tainted; variable **`x`** is thus related to document.cookie[0] and is tainted

Taint Propagation – eval()

- If a tainted parameter is passed to a function, then the return value of the function is tainted (as in tainting for integrity)
- Functions defined inside a tainted scope are tainted; all the expressions and assignments result returned by the function are also tainted
- When tainting for integrity (SQLI, XSS), `eval()` is a sensitive sink for code injection
- When tainting for confidentiality (cookie stealing), `eval()` is a propagator; if invoked in a tainted scope or if its argument is tainted, then the result is tainted

Sensitive Sinks

- Tainting for integrity: sensitive sinks are points where tainted data is inserted into the database or displayed to users
- Tainting for confidentiality: sensitive sinks are points where sensitive data is transferred to a site under the attacker's control
- Transfer can be achieved using a variety of methods; we list the main methods used for data transfer

Sinks – Transfer Methods

- Change location of the current web page:
  - Changing the `document.location` object value will make the web browser navigate to another web page; attackers can thus trick the victim to submit sensitive data to another URL
- Change source of an image in the web page
  - JavaScript can manipulate the source of an image object to dynamically change the picture in the view; attacker can assign the source of an image object with a predefined URL and append the sensitive data as a query parameter
- Automatically submitting a form in the web page
  - JavaScript can submit a form object in the HTML document; attacker can either embed sensitive data in the form or append them to the URL as query parameters

Sensitive Sinks

- Expression property in CSS
  - The expression property of CSS allows developers to assign a JavaScript expression to a CSS property; attacker can use this property to transfer data to other website
- Special objects, such as `XMLHttpRequest`
  - `XMLHttpRequest` object provides a way to communicate with a server after a web page has been loaded; script can send / retrieve data between client and server in the background

Dynamic Data Tainting

- Dynamic data tainting at client implemented by modifying the JavaScript engine of the browser
- JavaScript engine tracks information flow of sensitive data; when an attempt to relay such information to a third party is detected, the user is warned and given the possibility to stop the transfer
- Taint analysis for information flow applies taint to variables, but not to the data in the variables
  - Checks whether tainted data is sent out to another website
  - The value of tainted data is not checked

Information Flow

- Dynamic tainting tracks the flow of sensitive values through data dependencies, but it is not sufficient to detect all kinds of control dependencies

```
<?php
    $x = false;
    $y = false;
    if (document.cookie == "abc")
         { $x = true; }
    else { $y = true; }
    if ($x == false) { ... }
    if ($y == false) { ... }
?>
```

Information Flow

- Variables **`$x`** and **`$y`** are initialized to `false`
- First `if` condition uses `document.cookie` (tainted)
- If the condition is true, variable **`$x`** is assigned true; **`$x`** is modified inside a tainted block so gets tainted
- **`$y`** is not modified and thus remains untainted, as are the operations in the third block, which thus could leak information about `document.cookie`
- Dynamic tainting misses the vulnerability because it only tracks the branch which is actually executed
- Note: observing that something has NOT happened may leak information

Information Flow

- In our example, the else branch of the first if block is not executed, thus variable `$y` remains untainted, although its value depends on a sensitive input
- Static analysis can consider every branch in the control flow that depends on a tainted input
- No matter whether a branch in the control flow is executed or not, all variables that are assigned values within the control flow must be tainted
- In the example, both `$x` and `$y` will be tainted
  - P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna: Cross-Site Scripting Prevention with Dynamic Data Tainting and Static Analysis, NDSS 2007

Conclusion

Summary

- Tainting tracks potentially dangerous inputs to the point where they are used in sensitive operations
  - Challenge: identify all sources of tainted input
  - Challenge: identify all modes of propagation
  - Challenge: identify all sensitive sinks
  - Challenge: design of sanitizers, understand their limitations
- Answers to these challenges may depend on specific features of operating systems, web servers, or database systems, and on the given threat model
- Implementation challenge: where to store taint and how to propagate taint in the runtime environment

Summary

- For injection attacks, perform data flow taint analysis
  - Taint analysis for DOM-based XSS tracks data flows in client
- For leakage attacks, perform information flow taint analysis
  - Information can flow to an area without data flowing there!
  - No data flow can imply information flow
- Limitation of tainting: sanitization functions are not always perfect; an attack might then pass through the sanitization function and reach sensitive sinks