C10 Taint Analysis

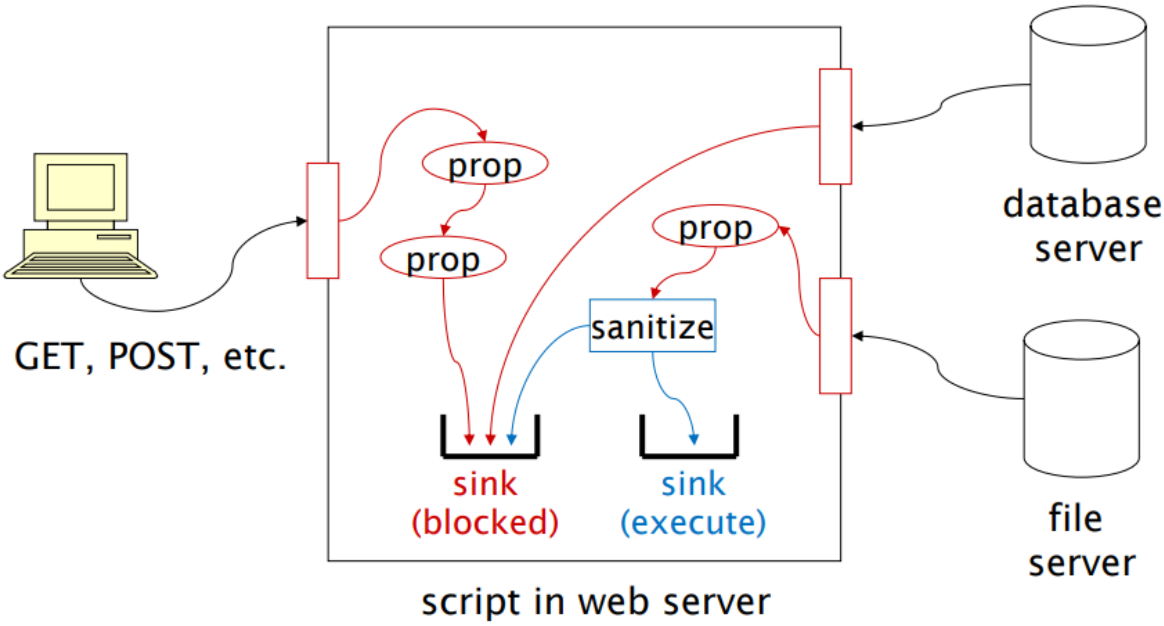
1. Principles of Taint Analysis

Data Tainting

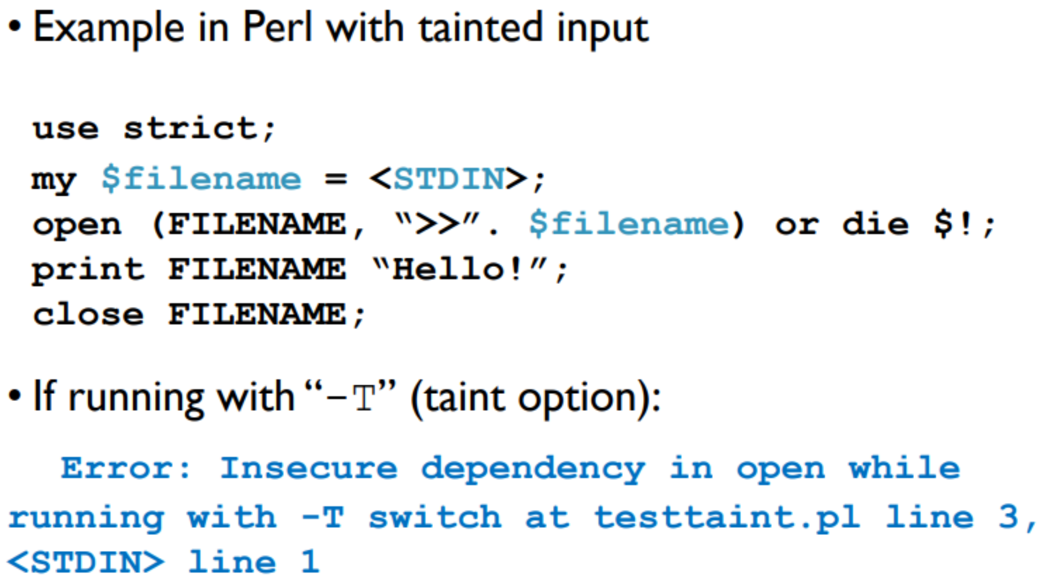
- To filter / encode / escape dangerous characters, there must be information on what is dangerous, and this depends on protocol, programming languages etc.
- Difficult / impossible to have a universal filter that catches all dangerous inputs
    - Can track inputs from the time they enter a module until they are consumed by a 'trust sink'
- **Data tainting** automatically checks whether "tainted" input is passed to a sensitive command without prior sanitization

1.1 Principles

- Data coming from untrusted sources should be marked as tainted, and may spread across a program through propagation functions.
- Certain operations can sanitise tainted data but different attacks require different sanitisers and may require additional measures if attacks are missed
- Sensitive sinks must never use tainted data, checked using a data flow analysis



- **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
```

1.2 Categories of Operations

- **Propagators** - functions that propagate tainted data to other variables
- **Sanitisers** - functions making tainted data safe to use
- **Sensitive sinks** - functions that access the file system / database / output information to the user

1.3 Dynamic & Static Tainting

- Dynamic tainting performed at runtime; necessary checks are normally included by the compiler
    - 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, eliminating problems before deployment of code
    - Can examine code paths that are rarely executed but understanding will be limited

1.4 Data / Information flow analysis

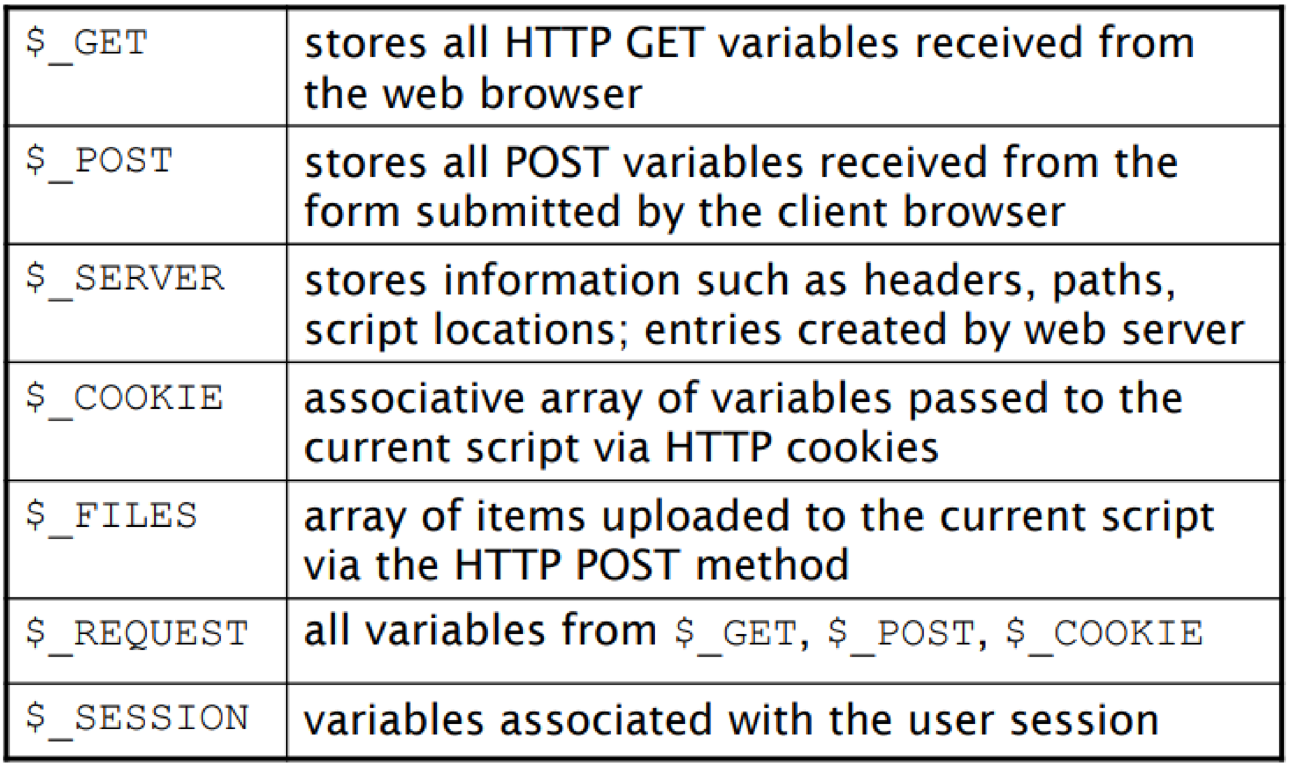
- Taint analysis can be done to address injection attacks as well as address leakage of sensitive data
- Code injection attacks → **data flow analysis**
    - Concern for server-side tainting
    - May be an issue on client side (DOM-based XSS)
- Leaking of sensitive data → **information flow analysis**
    - Concern for client-side tainting (cookie stealing)
- Sources, propagation functions, trust sinks are different but general principle remains the same

2. PHP

- Scripting language for producing dynamic web pages
- Weakly typed: variables do not have explicit type (need not be declared before use and can change type)
- Embedded in HTML documents with `<?php [php script] ?>`

Inputs to Scripts

- Typically sent from HTML form which gives the PHP script, parameters entered in form:
  - `<form action="example.php" method="post">` for parameters passed in body of POST request
  - `<form action="example.php" method="get">` for 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>`
- **Superglobal arrays** are predefined to store variables from external resources



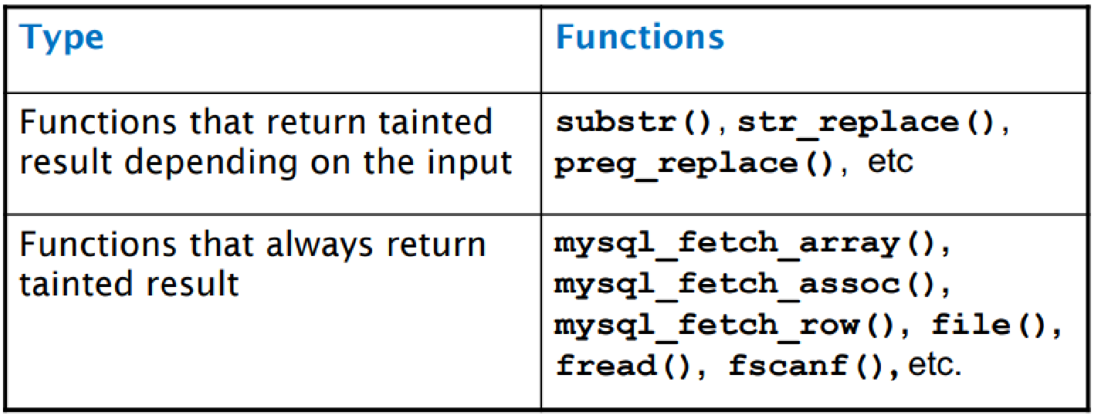
`$_GET`	stores all HTTP GET variables received from the web browser
`$_POST`	stores all POST variables received from the form submitted by the client browser
`$_SERVER`	stores information such as headers, paths, script locations; entries created by web server
`$_COOKIE`	associative array of variables 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

- All inputs to the script are tainted and have to be identified
  - `$_GET, $_POST, $_COOKIES, $_SERVER` superglobal arrays + data from internal sources (database and files) are tainted

Propagation Functions

- For string manipulation and database functions

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



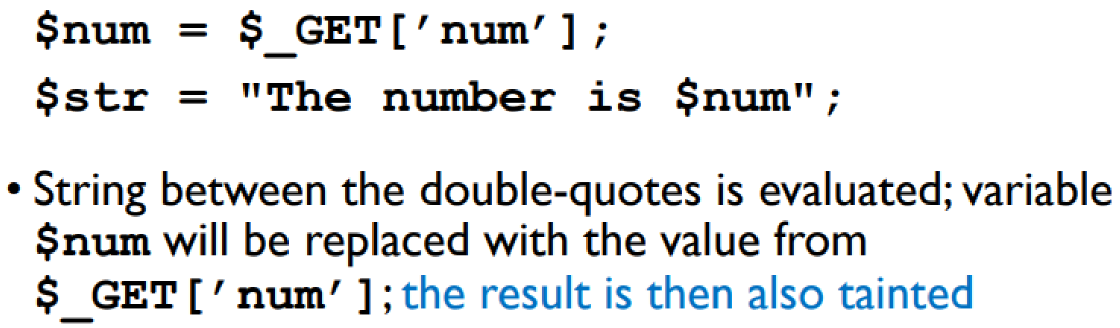
Type	Functions
Functions that return tainted result depending on the input	`substr()`, `str_replace()`, `preg_replace()`, etc
Functions that always return tainted result	`mysql_fetch_array()`, `mysql_fetch_assoc()`, `mysql_fetch_row()`, `file()`, `fread()`, `fscanf()`, etc.

2.1 PHP Strings

- Variables in double quoted strings `""` are evaluated (replaced by values) and variables in single quoted strings `''` are read as strings

Propagation in strings

- Taint can propagate through double quoted strings



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

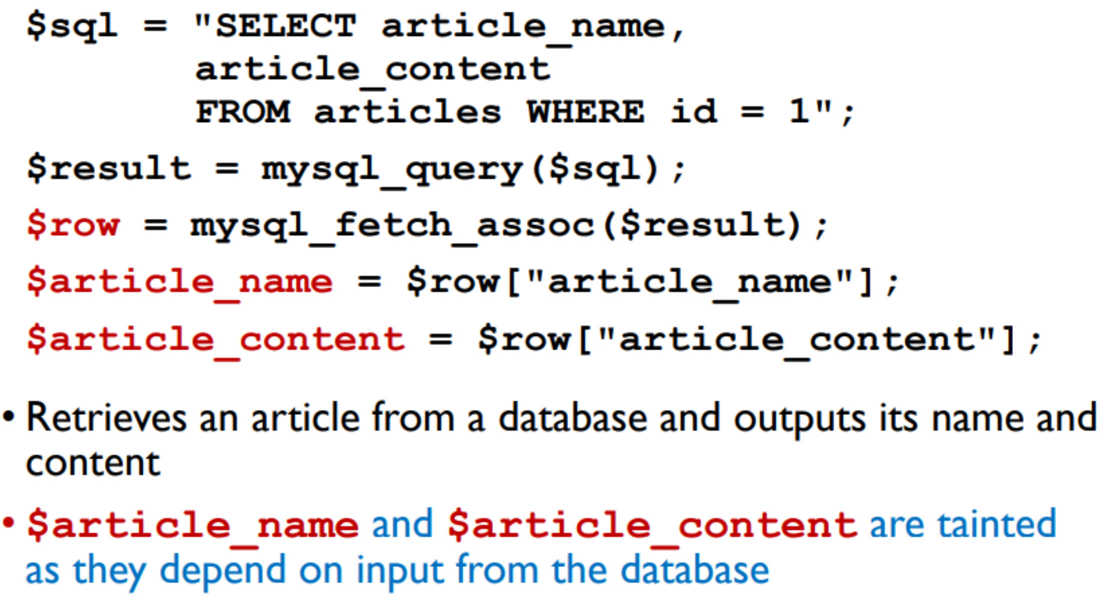
- String between the double-quotes is evaluated; variable `$num` will be replaced with the value from `$_GET['num']`; the result is then also tainted

- `substr()` - if input is tainted, result is also tainted
- `str_replace()` - replaces all occurrences of the search string with replacement string
- `$str = $str1 . $str2` (concatenation string) - if one string is tainted, LHS is also tainted

2.2 Propagation Functions

- Always return tainted results

  - `mysql_fetch_assoc()` fetches a result row from a SQL query as an associative array



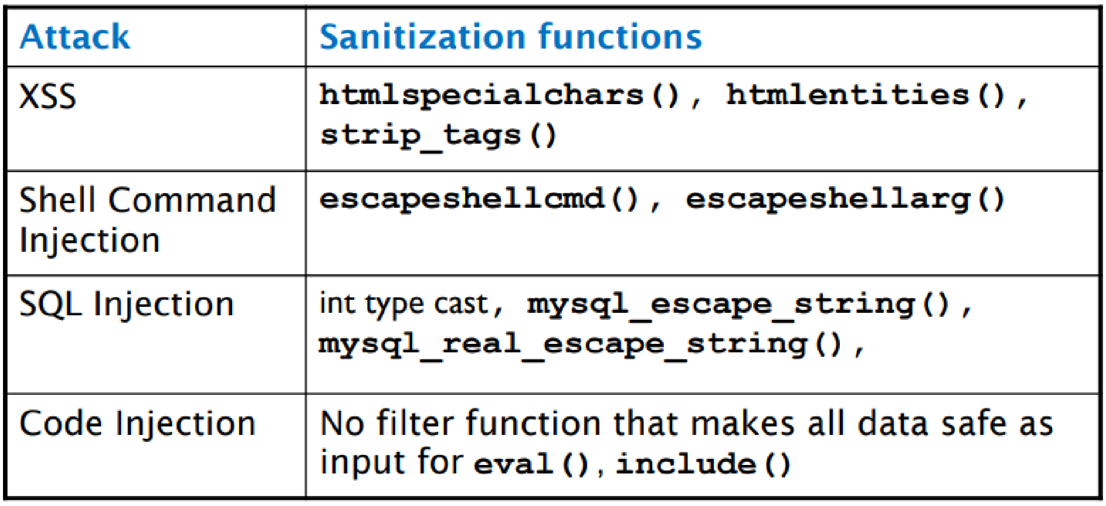
```
$sql = "SELECT article_name,
        article_content
        FROM articles WHERE id = 1";
$result = mysql_query($sql);
$row = mysql_fetch_assoc($result);
$article_name = $row["article_name"];
$article_content = $row["article_content"];
```

- 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

- Retrieve data from file system
  - `file()` reads an entire file into an array; each array element represents a line in the file, tainting each element

2.3 Sanitization Functions

- Clean up input data,, return untainted results



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()`

XML Sanitizers

- `htmlspecialchars()` - convert characters that have special meaning in HTML to HTML entities
  - Prevents user-supplied text from containting HTML markup, such as in a message board or guest book application
- `strip_tags()` - strip tags from HTML markups

SQL Injection Sanitizers

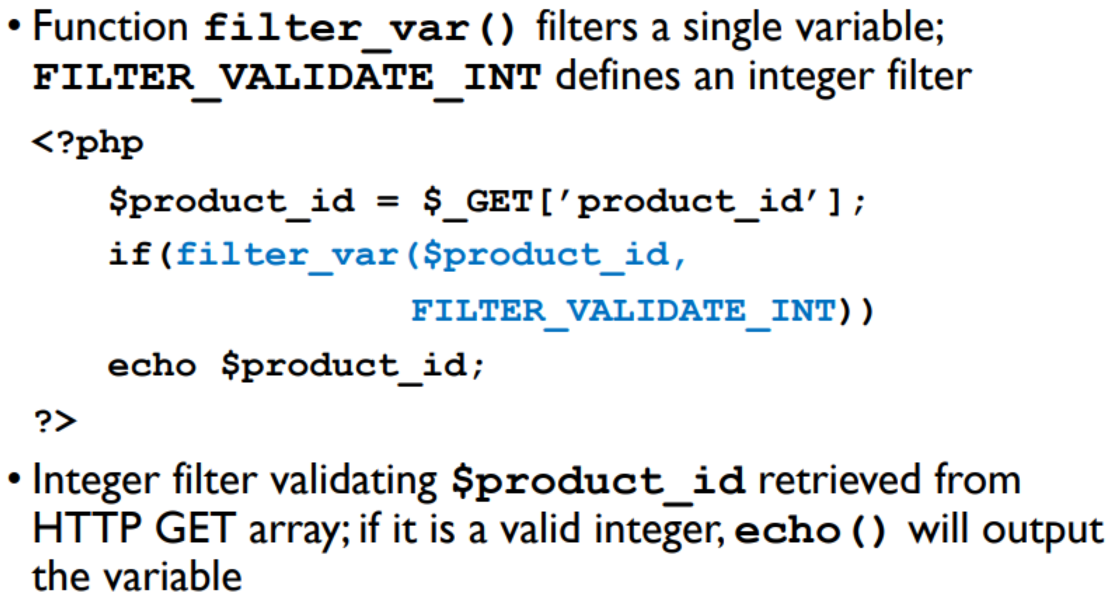
- `mysql_escape_string()`, `mysql_real_escape_string()` - adds backslash in front of single / double quotes and other characters that may be used to break out of a user input

Shell Command Sanitizers

- Invoked before arguments are passed to system calls like `system(), exec(), passthru()`
- 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 single quotes within the string
- `escapeshellcmd()` used on complete shell command; escapes characters that have a special meaning to the underlying operating system

2.4 Input Filtering

- Set of filter functions for validating and sanitizing user supplied data
- **Validation filters** returns a boolean value to indicate whether input is valid

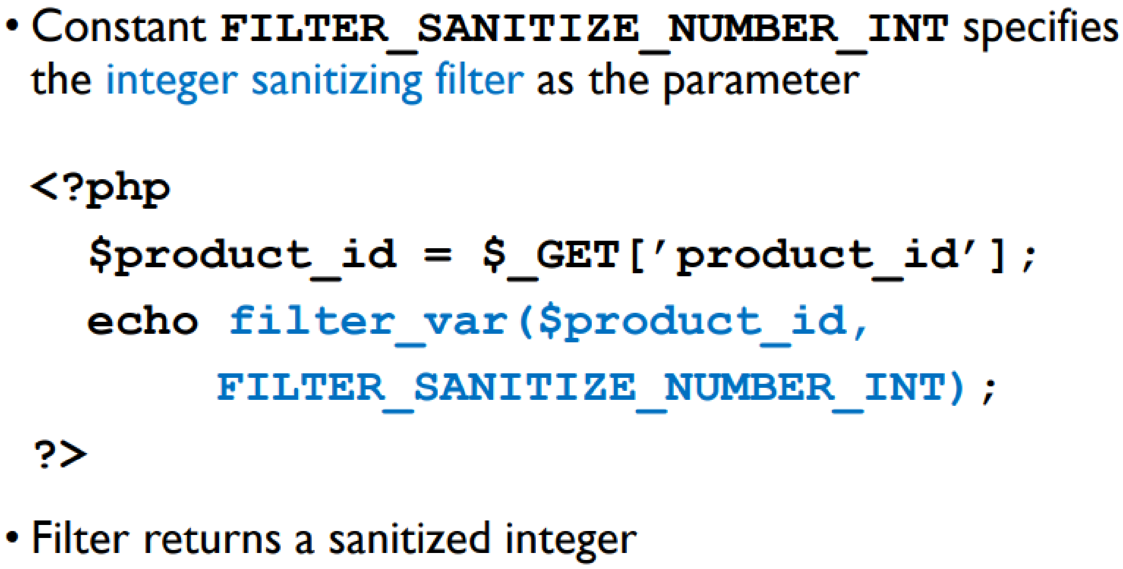


- Function `filter_var()` filters a single variable; `FILTER_VALIDATE_INT` defines an integer filter

```
<?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

- **Sanitization filters** returns a value that complies with filter rules



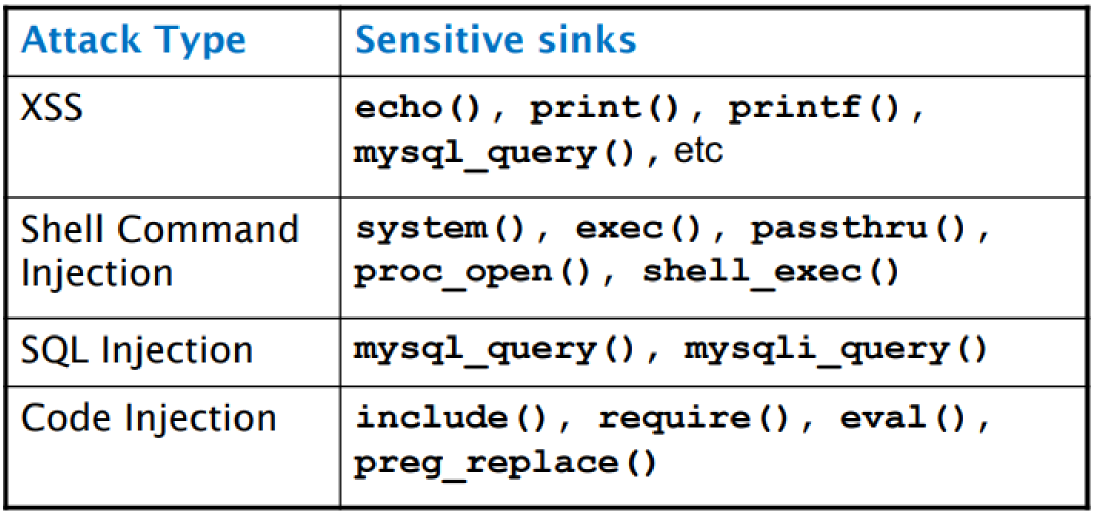
- Constant `FILTER_SANITIZE_NUMBER_INT` specifies the integer sanitizing filter as the parameter

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

- Filter returns a sanitized integer

2.5 Sensitive Sinks

- Functions that access the file / database system / output information to user



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()`

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

3. Data flow analysis - TA for integrity

3.1 Propagation Issues

Flow Sensitivity

- Variables declared in a script may be used several times, which must be considered by taint analysis at each program point

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

- `$var` first initialised locally in script with constant `var1` so it is untainted initially
  - `$var` is reassigned a value from `$_GET['var']` which is an external source, tainting `$var`

Context Sensitivity

```php
<?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 tainted parameter `$_GET` and assigned to `$var_a` which is displayed to user using `echo`, should be flagged
- `foo()` second call involves a harmless parameter, should be allowed

Alias analysis

- An alias, defined with `=&` is a variable that refers to the same memory location; assigning a value to the variable writes the value to the variable's memory location, affecting all aliases of the variable.
- When any variable (main / aliases) are modified using tainted functions, all of them should be flagged

File inclusion

- PHP code may be split into several files merged at runtime with inclusion statements (`include`, `require`); included files may contain vulnerabilities and **must be resovled automatically by tainting**

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

- `$x` gets tainted in file_b, causing a vulnerability

Dynamic File Inclusion

- Included file can only be determined at runtime

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

- There is a need to know the values held in variables `$name`, which becomes complicated when string values are propagated across functions, defined constants, global variables etc.

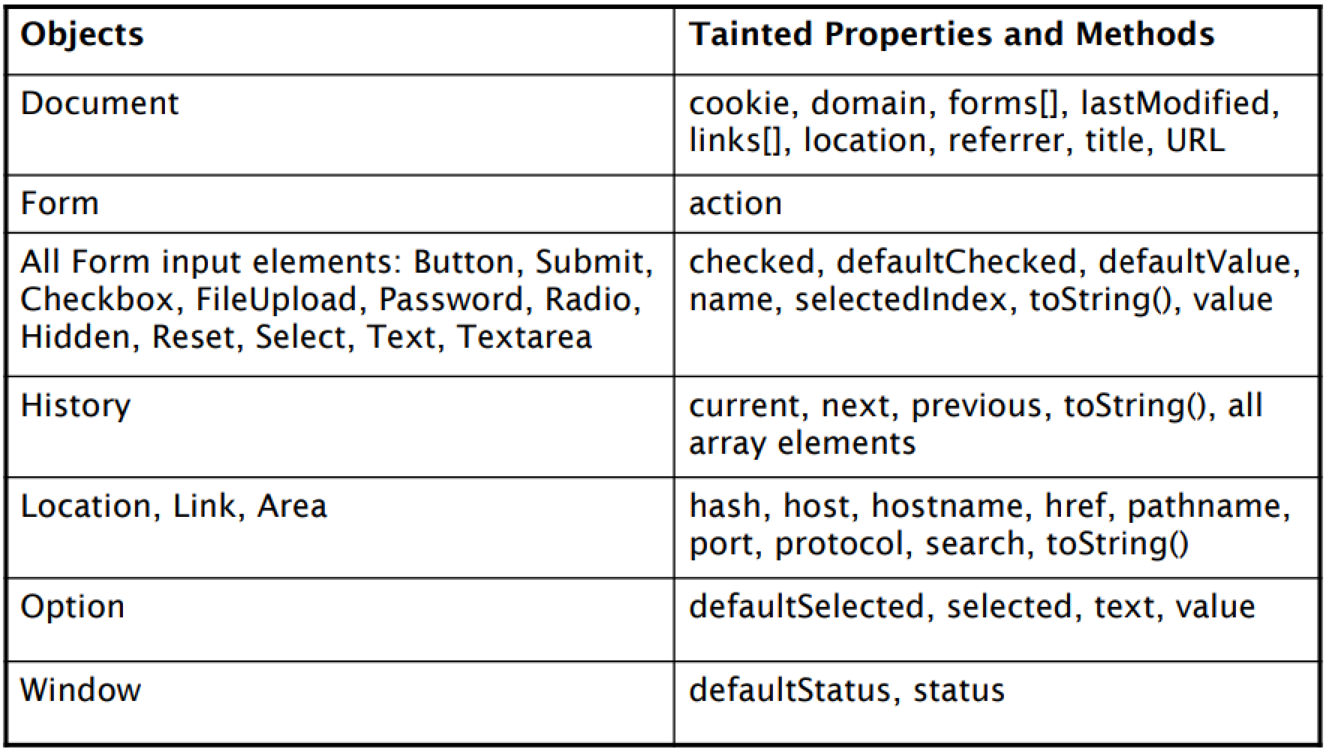
4. Information flow analysis - TA for confidentiality

- With SQL Injection and XSS, taint analysis checks whether user-supplied data can be sent to sensitive sinks; there is no intention to protect sensitive data
- Tainting can be used to prevent sensitive user data from being leaked to a third party (client-side tainting - can also be used to detect code injection)
- Tainting for injection and leakage are different with respect to the sources of tainted data, propagation functions and sensitive sinks.

4.1 Client-Side Tainting

- 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
  - Tainted sources for **Injection Attacks** are user-supplied data
  - Tainted sources for **Extraction (leakage)** are data holding information about users
- Main sources are cookies, URL

4.2 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

(each object represents a HTML element in DOM)

- **Document** - contains array properties specifying information about the contents of the document
  - Cookie, links, anchors, HTML forms, applets, embedded data
- **Form** - represents a HTML form which users use to interact with a web application
  - Action stores URL the form is submitted to
  - Contains elements like Text Fields, Checkbox, Dropdown list, buttons etc
- **Option** - represents an option in a dropdown list in HTML form
- **History** - stores the web browser's history; contains methods to navigate to previous or next pages the web browser has visited
- **Location** - represents current URL of document

4.3 Taint Propagation

- Values derived from tainted data elements are also tainted. When passed to a function, return value of the function will also be tainted.
- If a string is tainted, its substrings are also tainted
- If a script examines a tainted value in a conditional statement, the script becomes tainted

Assignments

- If RHS of assignment is tainted, the LHS will also be tainted
- If LHS is an array element that has been tainted, the whole array object becomes tainted
- If a property of an object is set to a tainted value, then the whole object is tainted

Arithmetic and Logic Operations

- **Tainting for Integrity:** result of a numeric operation is untainted since the result is a number which is not harmful to the system
    - Variable is only tainted when a tainted value is assigned to the LHS variable in a ternary operation (`c = (a > b)? a : b`)
- **Tainting for confidentiality:** if one operand is tainted, then the result is tainted for all arithmetic operations

Conditional Expressions

- If the condition of a control structure contains the test of a tainted value, then the entire control structure is a tainted scope
    - All operations and assignment results in the scope are tainted
    - A variable is dynamically tainted if its value is modified inside a scope during program execution

eval()

- Functions defined inside a tainted scope are tainted, together with all expressions and assignment result returned by the function
- When **tainting for integrity (SQLI, XSS)**, `eval()` is a sensitive sink for code injection
- When **tainting for confidentiality**, `eval()` is a propagator; if invoked in a tainted scope or if its argument is tainted, then result is tainted

4.4 Sensitive Sinks

- **Tainting for Integrity** - Sensitive sinks are points where tainted data is inserted into the database or displayed to the users
- **Tainting for Confidentiality** - Sensitive sinks are points where sensitive data is transferred to a site under the attacker's control

Transfer Methods

- Change location of current web page
    - Changing the `document.location` object value will make the web browser navigate to another web page
- 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 webpage
    - JavaScript can be used to 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
- Expression Property in CSS

    - Allows developers to assign a JavaScript expression to a CSS property; attacker can use this property to transfer data to other website

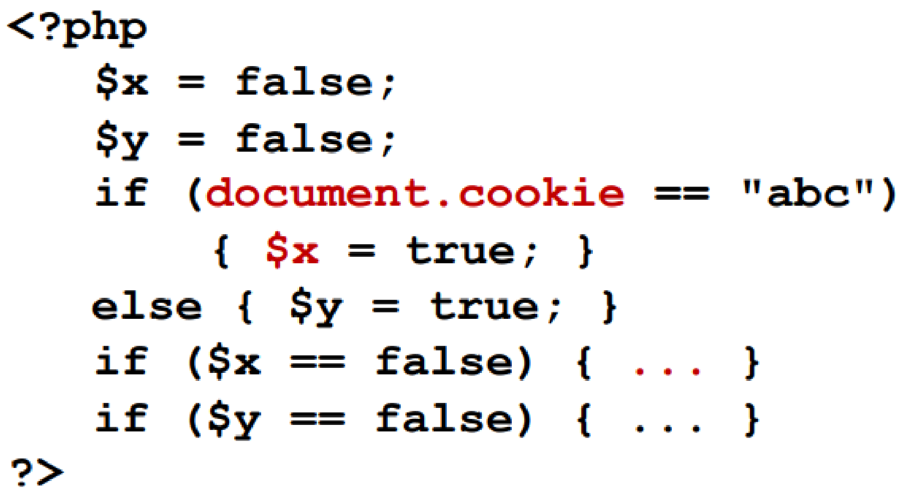
- Special objects (e.g. `XMLHttpRequest`)
    - `XMLHttpRequest` provides a way to communicate with a server after a web page has been loaded; script can send / retrieve data between client and server in the background

4.5 Dynamic Data Tainting

- Implemented by modifying JS engine of the browser; JS 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
    - 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) { ... }
?>
```

- Variables `$x` and `$y` are initialised to `false`
- First `if` condition uses `document.cookie` which is tainted, and if true, `$x` gets assigned `true`, causing it to become tainted
    - `$y` is not modified and 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
    - Observing that something has not happened may leak information
- Static analysis can consider every branch in the control flow that depends on 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