Input Validation with PHP

Michael Alaimo

DC PHP Monthly Meeting

08/09/2016

You must validate

In programming input validation is very important. When user input is not validated malicious users can exploit security issues in computer applications. Applications can also function incorrectly as a result of bad user input.

Validation is first in Security

Input data validation is the most important security measure in software security. -

Yasuo Ohgaki, a PHP core developer working on updating filter support (http://bit.ly/2b3gl1u).

What to validate

Validate input from all untrusted data sources. Proper input validation can eliminate the vast majority of software vulnerabilities. Be suspicious of most external data sources, including command line arguments, network interfaces, environmental variables, and user controlled files -

Robert C. Seacord, author of Secure Coding in C and C+ who is a computer security specialist and is leads the Secure Coding Initiative at CERT.

CERT

At the CERT Division of the Software Engineering Institute (SEI), we study and solve problems with widespread cybersecurity implications, research security vulnerabilities in software products, contribute to long-term changes in networked systems, and develop cutting-edge information and training to help improve cybersecurity.

You can view the CERT home page at: http://www.cert.org

Top 10 Secure Coding Practices

Top 10 Secure Coding Practices by Robert Seacord (http://bit.ly/2az01Jf)

- **Validate input. Validate input from all untrusted data sources. Proper input validation can eliminate the vast majority of software vulnerabilities. Be suspicious of most external data sources, including command line arguments, network interfaces, environmental variables, and user controlled files [Seacord 05].**
- Heed compiler warnings. Compile code using the highest warning level available for your compiler and eliminate warnings by modifying the code [C MSC00-A, C++ MSC00-A]. Use static and dynamic analysis tools to detect and eliminate additional security flaws.
- Architect and design for security policies. Create a software architecture and design your software to implement and enforce security policies. For example, if your system requires different privileges at different times, consider dividing the system into distinct intercommunicating subsystems, each with an appropriate privilege set.
- Keep it simple. Keep the design as simple and small as possible [Saltzer 74, Saltzer 75]. Complex designs increase the likelihood that errors will be made in their implementation, configuration, and use. Additionally, the effort required to achieve an appropriate level of assurance increases dramatically as security mechanisms become more complex.
- Default deny. Base access decisions on permission rather than exclusion. This means that, by default, access is denied and the protection scheme identifies conditions under which access is permitted [Saltzer 74, Saltzer 75].

Top 10 Secure Coding Practices

- Adhere to the principle of least privilege. Every process should execute with the the least set of privileges necessary to complete the job. Any elevated permission should be held for a minimum time. This approach reduces the opportunities an attacker has to execute arbitrary code with elevated privileges [Saltzer 74, Saltzer 75].
- **Sanitize data sent to other systems. Sanitize all data passed to complex subsystems [C STR02-A] such as command shells, relational databases, and commercial off-the-shelf (COTS) components. Attackers may be able to invoke unused functionality in these components through the use of SQL, command, or other injection attacks. This is not necessarily an input validation problem because the complex subsystem being invoked does not understand the context in which the call is made. Because the calling process understands the context, it is responsible for sanitizing the data before invoking the subsystem.**
- Practice defense in depth. Manage risk with multiple defensive strategies, so that if one layer of defense turns out to be inadequate, another layer of defense can prevent a security flaw from becoming an exploitable vulnerability and/or limit the consequences of a successful exploit. For example, combining secure programming techniques with secure runtime environments should reduce the likelihood that vulnerabilities remaining in the code at deployment time can be exploited in the operational environment [Seacord 05].
- Use effective quality assurance techniques. Good quality assurance techniques can be effective in identifying and eliminating vulnerabilities. Fuzz testing, penetration testing, and source code audits should all be incorporated as part of an effective quality assurance program. Independent security reviews can lead to more secure systems. External reviewers bring an independent perspective; for example, in identifying and correcting invalid assumptions [Seacord 05].
- Adopt a secure coding standard. Develop and/or apply a secure coding standard for your target development language and platform.

OWASP

The Open Web Application Security Project - (http://bit.ly/2aJcoOe)

- OWASP thrives to be the global community that drives visibility and evolution in the safety and security of the world’s software.
- They provide information on PHP security practices including input validation.
- You can find your local chapter here: http://bit.ly/2bfM9VL

OWASP Secure Coding Practices Quick Reference Guide

Input Validation:

- Conduct all data validation on a trusted system (e.g., The server)
- Identify all data sources and classify them into trusted and untrusted. Validate all data from untrusted sources (e.g., Databases, file streams, etc.)
- There should be a centralized input validation routine for the application
- Specify proper character sets, such as UTF-8, for all sources of input
- Encode data to a common character set before validating (Canonicalize)
- All validation failures should result in input rejection
- Determine if the system supports UTF-8 extended character sets and if so, validate after UTF-8 decoding is completed

OWASP Secure Coding Practices Quick Reference Guide

- Validate all client provided data before processing, including all parameters, URLs and HTTP header content (e.g. Cookie names and values). Be sure to include automated post backs from JavaScript, Flash or other embedded code
- Verify that header values in both requests and responses contain only ASCII characters
- Validate data from redirects (An attacker may submit malicious content directly to the target of the redirect, thus circumventing application logic and any validation performed before the redirect)
- Validate for expected data types
- Validate data range
- Validate data length
- Validate all input against a "white" list of allowed characters, whenever possible

OWASP Secure Coding Practices Quick Reference Guide

- If any potentially hazardous characters must be allowed as input, be sure that you implement additional controls like output encoding, secure task specific APIs and accounting for the utilization of that data throughout the application . Examples of common hazardous characters include: < > " ' % ( ) & + \ \' \"
- If your standard validation routine cannot address the following inputs, then they should be checked discretely
- Check for null bytes (%00)
- Check for new line characters (%0d, %0a, \r, \n)
- Check for “dot-dot-slash" (../ or ..\) path alterations characters. In cases where UTF-8 extended character set encoding is supported, address alternate representation like: %c0%ae%c0%ae/ (Utilize canonicalization to address double encoding or other forms of obfuscation attacks)

OWASP Secure Coding Practices Quick Reference Guide

Output Encoding:

- Conduct all encoding on a trusted system (e.g., The server)
- Utilize a standard, tested routine for each type of outbound encoding
- Contextually output encode all data returned to the client that originated outside the application's trust boundary. HTML entity encoding is one example, but does not work in all cases
- Encode all characters unless they are known to be safe for the intended interpreter
- Contextually sanitize all output of un-trusted data to queries for SQL, XML, and LDAP
- Sanitize all output of un-trusted data to operating system commands

OWASP Secure Coding Practices Quick Reference Guide

You can download the full OWASP Secure Coding Practices Quick Reference Guide here:

http://bit.ly/2aEGIKg

Input Validation can prevent

Overflows

Injections

OpenBSD and W^X overflow protection

W^X first appeared in OpenBSD 3.3, released May 2003. Similar features are available for other operating systems, including the PaX and Exec Shield patches for Linux, and NetBSD 4+'s implementation of PaX.

Although this feature has only protected userland programs for most of its existence, in late 2014 and early 2015, Mike Larkin made W^X protect the OpenBSD kernel itself on the AMD64 architecture, with Theo de Raadt aiding the effort.

Windows uses Data Execution Protection (DEP)

(http://bit.ly/2beXt0Z)

Types of Overflows

- arithmetic overflow - a condition that occurs when a calculation produces a result that is greater than what a given register can store or represent
- integer overflow - a condition that occurs when an integer calculation produces a result that is greater than what a given register can store or represent
- Buffer overflow - a situation whereby the incoming data size exceeds that which can be accommodated by a buffer
- heap overflow – A type of buffer overflow occuring in the heap data area
- stack overflow - A type of buffer overflow in which a computer program makes too many subroutine calls and its call stack runs out of space

Types of Injections

- SQL Injection – Modify an SQL query
- LDAP Injection – Modify an LDAP query
- XML Injection - Modify an XML application
- Xpath Injection – Modify an Xpath query
- Shell Injection – Modify execution of a shell application
- Code Injection – Modify an object in the applications runtime
- Cross Site Scripting (XSS) – Add script to the web application
- Cross Site Request Forgery (CSRF) – Exploit a web application from a trusted account

National Vulnerability Database

NVD is the U.S. government repository of standards based vulnerability management data represented using the Security Content Automation Protocol (SCAP). This data enables automation of vulnerability management, security measurement, and compliance. NVD includes databases of security checklists, security related software flaws, misconfigurations, product names, and impact metrics.

National Vulnerability Database

The URL is: http://nvd.nist.gov/

Examples

Security issues found by using t he NVD system and are located in Common Vulnerabilities and Exposures (CVE)

CSRF

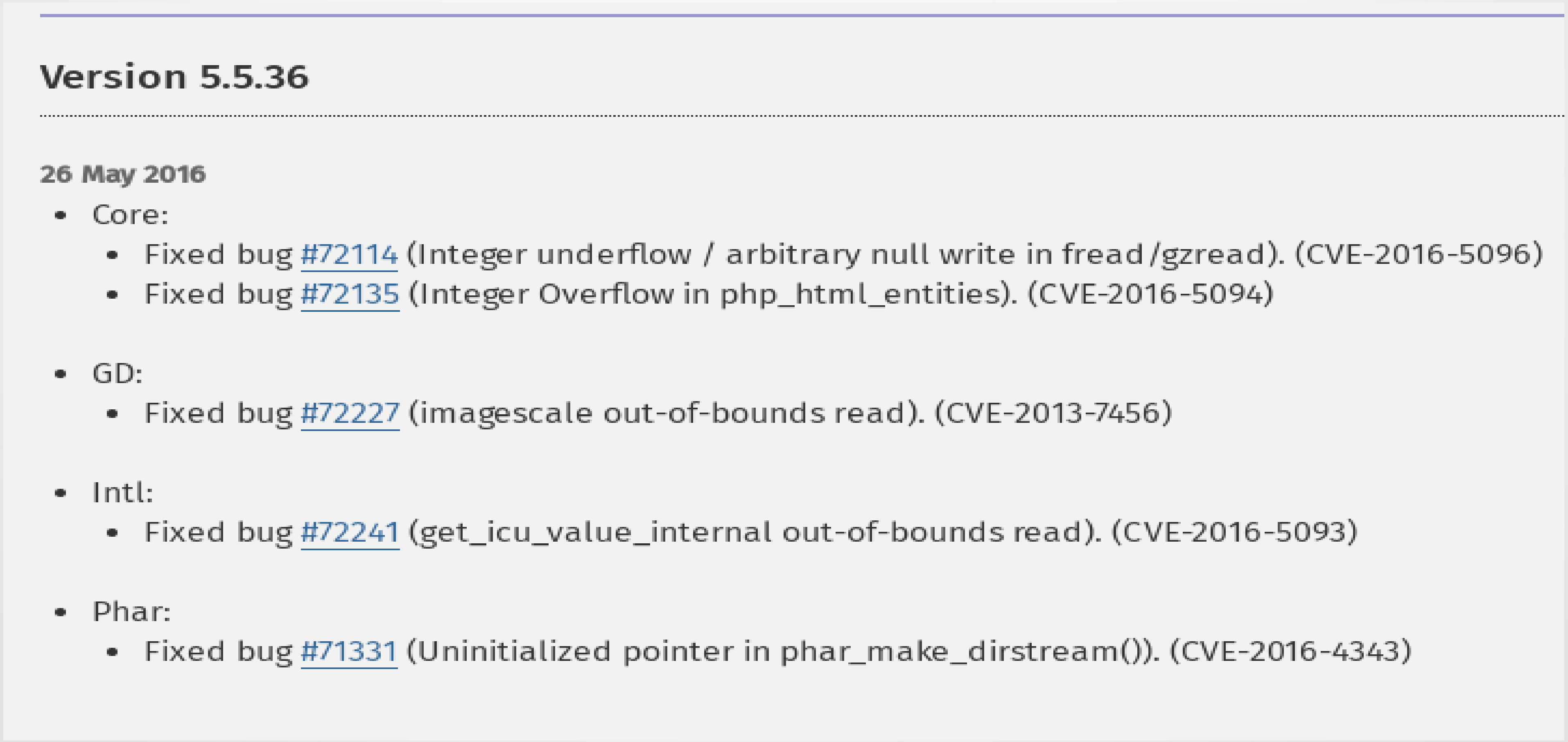
- WordPress 4.5 contained a CSRF attack security issue
- CVE-2016-6635 - http://bit.ly/2b8kGou
- They security fix involved adding a check to ensure that the nonce or security token is valid during an AJAX compression test
- This was corrected by Ronni Skansing - http://bit.ly/2b8k2aq

Denial of service, integer overflow

- PHP before 5.5.36 and 5.6.x before 5.6.22 contain an issue with htmlspecialchars and filter functionality
- CVE-2016-5094 - http://bit.ly/2b6oZPS
- Here an extra validation of max input size was added to ensure that the use does not add data larger than the system can handle

PHP Changelog

http://bit.ly/2aLGYKt



Version 5.5.36

26 May 2016

- Core:
  - Fixed bug #72114 (Integer underflow / arbitrary null write in fread/gzread). (CVE-2016-5096)
  - Fixed bug #72135 (Integer Overflow in php_html_entities). (CVE-2016-5094)
- GD:
  - Fixed bug #72227 (imagescale out-of-bounds read). (CVE-2013-7456)
- Intl:
  - Fixed bug #72241 (get_icu_value_internal out-of-bounds read). (CVE-2016-5093)
- Phar:
  - Fixed bug #71331 (Uninitialized pointer in phar_make_dirstream()). (CVE-2016-4343)

Code Injection, Remote Code Execution

- Drupal had a recent issue where session data truncation can lead to unserialization of user provided data
- CVE-2016-3171 - http://bit.ly/2aVY061

Upgrading PHP to 5.4.45, 5.5.29, 5.6.13 can mitigate the issue

This was fixed by:

- David Jardin of the Joomla Security Team
- Damien Tournoud of the Drupal Security Team - http://bit.ly/2b8pf1T
- Heine Deelstra of the Drupal Security Team - http://bit.ly/2aMtLQt

Code Injection

It is possible to alter the function of an application who implements __wakeup or __desctruct methods

- The injection occurs during unserialization or object destruction
- http://bit.ly/2aLTNXN

- The code would function normally, but a malicious user modified a parameter of the serialized object.
- When the object was unserlialized the wakeup method is called and when the object is destroyed the descruct method is called.
- It is possible that a parameter may have been modified external to the application and will now cause code injection

Code Injection

Code injection is possible using the REQUEST methds like GET, POST, PUT, DELETE and COOKIE

- For example a user could store serialized data in a cookie which could be modified on the client side
- When the object is unserlized on the server side parameters of the object may have changed to adversley effect the system during unserialization
- The best option is to not use serialized data from a request object

Code Injection

For example, it may make sense to store a small object into a HTTP cookie, but that will leave your code open to code injection

HMAC Cookies

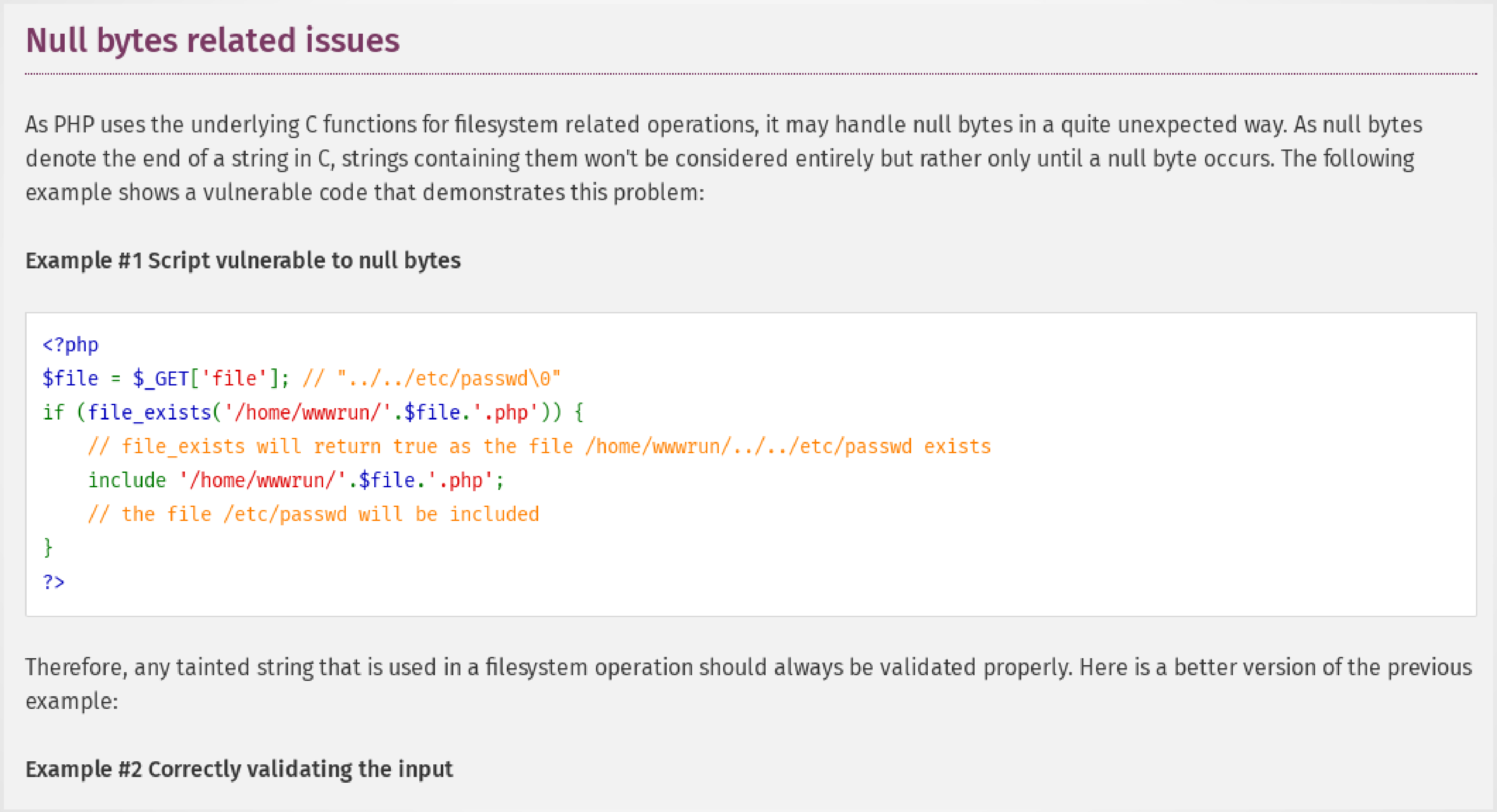
- It is possible to store a hash of your cookie data and the raw cookie data into a cookie
- You can then hash the raw data on the server side and test it against the hash that was sent with your cookie
- If the values match then you can feel that the data was not modified

PHP Security Manual

- http://bit.ly/2aSILO5
- Contains information related to PHP and security including input validation

- Null Byte Security
- SQL Injections
- Register Globals
- Magic Quotes

Null Byte Security



Null bytes related issues

As PHP uses the underlying C functions for filesystem related operations, it may handle null bytes in a quite unexpected way. As null bytes denote the end of a string in C, strings containing them won't be considered entirely but rather only until a null byte occurs. The following example shows a vulnerable code that demonstrates this problem:

Example #1 Script vulnerable to null bytes

```php
<?php
$file = $_GET['file']; // "../../etc/passwd\0"
if (file_exists('/home/wwwrun/'.$file.'.php')) {
    // file_exists will return true as the file /home/wwwrun/../../etc/passwd exists
    include '/home/wwwrun/'.$file.'.php';
    // the file /etc/passwd will be included
}
?>
```

Therefore, any tainted string that is used in a filesystem operation should always be validated properly. Here is a better version of the previous example:

Example #2 Correctly validating the input

SQL Injections

A way for an attacker to obtain or modify database data

- PDO – use quote or a prepared statement
- MySQLi – use real_escape_string or a prepared statement
- PGSQL – use pg_escape_* functions or the prepare functionality

- Using the database quoting and escaping functionality is the correct way to stop an SQL Injection
- SQL injection attacks are arguably the most common way PHP websites get exploited. The importance can not be overstated. - http://bit.ly/2aLYsWQ

Register Globals and Magic Quotes

- These features are now deprecated in PHP
- They have been items that made code injections and sql injections something that could be easily overlooked during development

Register Globals

- When enabled it is possible to create an SQL or code injection by overwriting a global variable which was not intended to be overwritten from an HTTP request

Magic Quotes

- Adds slashes to data similar to addslashes PHP function which does not prevent an SQL injection
- Instead use the database escape functionality

Methods on Validating Data with PHP

- PCRE Regular Expression - http://bit.ly/2bgKTS2
- PHP Filter functions - http://bit.ly/2aZCH5T
- Type Juggling - http://bit.ly/2b6J1d7
- Strip Tags - http://bit.ly/2aM1pa2
- Encoding Data

PCRE Regular Expression Example

```
If (preg_match('/^(\d+)$/', $_GET['id'], $matches) === 1) {
$id = $matches[1];
} else {
Echo 'Input is not an integer';
}
```

- preg_match will return 1 when a match is found and 0 when no match is found
- Matches will be contained in the third parameter of the preg_match function

PHP Filter Example

```
If (($var = filter_input(INPUT_GET, 'id', FILTER_VALIDATE_INT,
array('options' => array('min_range' => 5, 'max_range' => 10))
=== FALSE) {
    echo 'Input is not an integer';
}
```

- You can use filter_input for REQUEST variables
- You can use filter_var for regular variables
- There are both sanitize and validate filters
- Options exist for many filters

Strip Tags

- Striping tags can prevent Cross Site Scripting (XSS) and it also removes the null byte charater.
- Htmlspecialchars can prevent XSS as well, but leaves in html entities

Encoding Data

- Using filter functionality to encode ASCII data
- Using urlencode to encode url parameters
- Using utf8_encode when working with XML
- Using http_build_query when creating a url
- Using htmlentities or htmlspecialchars

PHP Casting Example

```
$id = (int)$_GET['id'];
```

- Variable $id now contains an integer
- (int), (integer) - cast to integer
- (bool), (boolean) - cast to boolean
- (float), (double), (real) - cast to float
- (string) - cast to string
- (array) - cast to array
- (object) - cast to object
- (unset) - cast to NULL (PHP 5)
- (binary) casting and b prefix forward support was added in PHP 5.2.1

Size Checking

- Checking the size of data can be important too
- Size checking can prevent overflows.
- SQL Fields can be tested to be within the ranges of the database fields. This includes text and numerical data and prevents data loss through truncation or query failure during inserts and updates.
- Testing the size of inputs can be important when dealing with numbers, text and files.
- Testing input size can prevent a script from not working due to running out of memory, filling up the hard disk with uploaded files or even filling server log files resulting in a denial of service.

Size Checking

- You can use the PHP comparison operators to check numerical sizes
- PHP FILTER_VALIDATE_INT can be used to test min and max size
- The php strlen function can determin a strings length

Validating data for creating requests

- Sending data to someone else's server is important
- This goes for Web Servers, SQL servers and file servers
- It is important to ensure that the data you send to the servers whether they are yours or someone elses is validated, sanitized and properly encoded

Input Validation Pointers

- If you use JavaScript or client side validation, you must also use serverside validation with something like PHP
- Read Internet Resources such as OWASP and the php|architect’s Guide to PHP Security
- Validate all user data including data read from files

OWASP Validation Strategy

- Exact Match (Constrain)
- Known Good (Accept)
- Reject Known bad (Reject)
- Encode Known bad (Sanitise)
- http://bit.ly/2aGXNTM
- http://bit.ly/2bgO0It

Exact Match

- In addition, there must be a check for maximum length of any input received from an external source, such as a downstream service/computer or a user at a web browser.
- Rejected Data must not be persisted to the data store unless it is sanitized. This is a common mistake to log erroneous data, but that may be what the attacker wishes your application to do.
- **Exact Match**: (preferred method) Only accept values from a finite list of known values.
- e.g.: A Radio button component on a Web page has 3 settings (A, B, C). Only one of those three settings must be accepted (A or B or C). Any other value must be rejected.

Known Good

- **Known Good**: If we do not have a finite list of all the possible values that can be entered into the system, we use the known good approach.
- e.g.: an email address, we know it shall contain one and only one @. It may also have one or more full stops ".". The rest of the information can be anything from [a-z] or [A-Z] or [0-9] and some other characters such as "_ "or "–", so we let these ranges in and define a maximum length for the address.

Reject Known Bad

- **Reject Known bad**: We have a list of known bad values we do not wish to be entered into the system. This occurs on freeform text areas and areas where a user may write a note. The weakness of this model is that today known bad may not be sufficient for tomorrow.

Encode Known Bad

- **Encode Known Bad**: This is the weakest approach. This approach accepts all input but HTML encodes any characters within a certain character range. HTML encoding is done so if the input needs to be redisplayed the browser shall not interpret the text as script, but the text looks the same as what the user originally typed.
- HTML-encoding and URL-encoding user input when writing back to the client. In this case, the assumption is that no input is treated as HTML and all output is written back in a protected form. This is sanitisation in action
- Sanitize data with PHP filter to encode unwanted characters

Conclusion

Input validation is important for security and correct functioning of the application. It can improve the user experience and prevent malicious users from using the application in a way other than it was intended.