Web technologies

Sanitizing and validating client inputs

David Jelenc

May 2017

Index

1. Sanitizing and validating inputs
   - Web applications can be accessed by anyone
   - SQL injection
   - Validating and sanitizing data
   - Pitfalls of $_SERVER["PHP_SELF"]

Index

1. Sanitizing and validating inputs
   - Web applications can be accessed by anyone
   - SQL injection
   - Validating and sanitizing data
   - Pitfalls of $_SERVER["PHP_SELF"]

Web applications can be accessed by anyone

- Web applications (running on public addresses) can be accessed by anyone – anyone can send HTTP requests (and data) to them.

Web applications can be accessed by anyone

- Web applications (running on public addresses) can be accessed by anyone – anyone can send HTTP requests (and data) to them.
- Your application should **never, never, never** trust data that is sent to it by the the client. Any client. Any time.

Web applications can be accessed by anyone

- Web applications (running on public addresses) can be accessed by anyone – anyone can send HTTP requests (and data) to them.
- Your application should **never, never, never** trust data that is sent to it by the the client. Any client. Any time.
- You application should **always, always, always** verify data for correctness:

Web applications can be accessed by anyone

- Web applications (running on public addresses) can be accessed by anyone – anyone can send HTTP requests (and data) to them.
- Your application should **never, never, never** trust data that is sent to it by the the client. Any client. Any time.
- You application should **always, always, always** verify data for correctness:
  - Expecting an email address from the user? Check the received string if it is in a valid format – does it contain symbol @?

Web applications can be accessed by anyone

- Web applications (running on public addresses) can be accessed by anyone – anyone can send HTTP requests (and data) to them.
- Your application should **never, never, never** trust data that is sent to it by the the client. Any client. Any time.
- You application should **always, always, always** verify data for correctness:
  - Expecting an email address from the user? Check the received string if it is in a valid format – does it contain symbol @?
  - Expecting a number? Check the received data if it is a number.

Web applications can be accessed by anyone

- Web applications (running on public addresses) can be accessed by anyone – anyone can send HTTP requests (and data) to them.
- Your application should **never, never, never** trust data that is sent to it by the the client. Any client. Any time.
- You application should **always, always, always** verify data for correctness:
  - Expecting an email address from the user? Check the received string if it is in a valid format – does it contain symbol @?
  - Expecting a number? Check the received data if it is a number.
  - In general, validation and sanitization are difficult.

Web applications can be accessed by anyone

- Web applications (running on public addresses) can be accessed by anyone – anyone can send HTTP requests (and data) to them.
- Your application should **never, never, never** trust data that is sent to it by the the client. Any client. Any time.
- You application should **always, always, always** verify data for correctness:
  - Expecting an email address from the user? Check the received string if it is in a valid format – does it contain symbol @?
  - Expecting a number? Check the received data if it is a number.
  - In general, validation and sanitization are difficult.
- What can go wrong?

Web applications can be accessed by anyone

- Web applications (running on public addresses) can be accessed by anyone – anyone can send HTTP requests (and data) to them.
- Your application should **never, never, never** trust data that is sent to it by the the client. Any client. Any time.
- You application should **always, always, always** verify data for correctness:
  - Expecting an email address from the user? Check the received string if it is in a valid format – does it contain symbol @?
  - Expecting a number? Check the received data if it is a number.
  - In general, validation and sanitization are difficult.
- What can go wrong? **Effects range between nothing to loss of customer data and breaches into computer systems in general.**

Which data?

Which data are we talking about?

Which data?

Which data are we talking about?
Anything that can come with an HTTP request:

Which data?

Which data are we talking about?
Anything that can come with an HTTP request:

- GET request parameters (contents of `$_GET`)

Which data?

Which data are we talking about?
Anything that can come with an HTTP request:

- GET request parameters (contents of `$_GET`)
- POST request parameters (contents of `$_POST`)

Which data?

Which data are we talking about?
Anything that can come with an HTTP request:

- GET request parameters (contents of `$_GET`)
- POST request parameters (contents of `$_POST`)
- Cookies (contents of `$_COOKIE`)

Which data?

Which data are we talking about?
Anything that can come with an HTTP request:

- GET request parameters (contents of `$_GET`)
- POST request parameters (contents of `$_POST`)
- Cookies (contents of `$_COOKIE`)
- Even some values accessible via `$_SERVER` super-global

SQL injection

- SQL injection is an attack procedure (an *attack vector*) where the attacker injects the vulnerable application with a query parameter that modifies the actual SQL query.

SQL injection

- SQL injection is an attack procedure (an *attack vector*) where the attacker injects the vulnerable application with a query parameter that modifies the actual SQL query.
- SQL injection occurs when the web application passes unverified data to the RDMS.

SQL injection

- SQL injection is an attack procedure (an *attack vector*) where the attacker injects the vulnerable application with a query parameter that modifies the actual SQL query.
- SQL injection occurs when the web application passes unverified data to the RDMS.
- Successful SQL injection attacks can read sensitive data from the database, modify database data, execute administration operations on the database, and in some cases issue commands to the operating system.

SQL injection

- SQL injection is an attack procedure (an *attack vector*) where the attacker injects the vulnerable application with a query parameter that modifies the actual SQL query.
- SQL injection occurs when the web application passes unverified data to the RDMS.
- Successful SQL injection attacks can read sensitive data from the database, modify database data, execute administration operations on the database, and in some cases issue commands to the operating system.
- *Log-in example*

Defending against SQL injection

- **Always use prepared statements and bind parameters**

Defending against SQL injection

- **Always use prepared statements and bind parameters – ALWAYS!**

Defending against SQL injection

- **Always use prepared statements and bind parameters – ALWAYS!**
- If using `mysql_*` functions (that do not support prepared statements), filter inputs by using function `mysql_real_escape_string()`, and use the filtered values to construct queries

Defending against SQL injection

- **Always use prepared statements and bind parameters – ALWAYS!**
- If using `mysql_*` functions (that do not support prepared statements), filter inputs by using function `mysql_real_escape_string()`, and use the filtered values to construct queries
- `$safe = mysql_real_escape_string($unsafe)`

Defending against SQL injection

- **Always use prepared statements and bind parameters – ALWAYS!**
- If using `mysql_*` functions (that do not support prepared statements), filter inputs by using function `mysql_real_escape_string()`, and use the filtered values to construct queries
- `$safe = mysql_real_escape_string($unsafe)`
- If possible, always use PDO and bind parameters

Defending against SQL injection

- **Always use prepared statements and bind parameters – ALWAYS!**
- If using `mysql_*` functions (that do not support prepared statements), filter inputs by using function `mysql_real_escape_string()`, and use the filtered values to construct queries
- `$safe = mysql_real_escape_string($unsafe)`
- If possible, always use PDO and bind parameters
- `http://php.net/security.database.sql-injection.php`

The danger is real

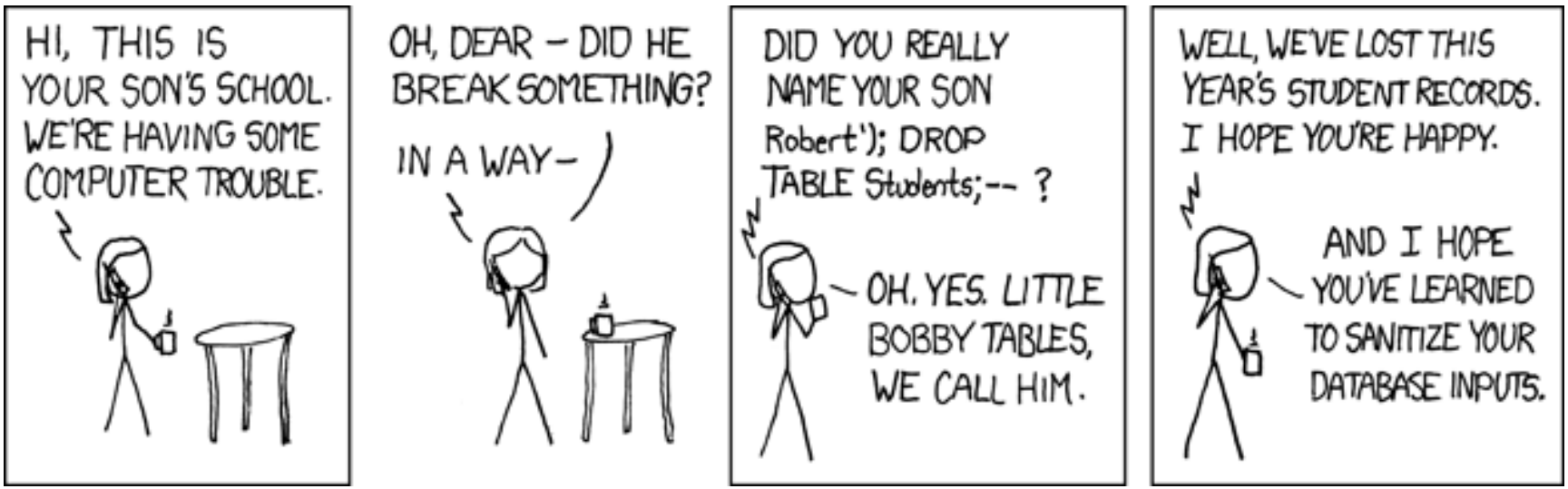

Figure: `http://xkcd.com/327`

Validating and sanitizing data

- Any client data that is not verified is a potential security threat.

Validating and sanitizing data

- Any client data that is not verified is a potential security threat.
- What are we most afraid of?

Validating and sanitizing data

- Any client data that is not verified is a potential security threat.
- What are we most afraid of?
  - Executing JavaScript in places that we did not plan, placed there by the attackers.

Validating and sanitizing data

- Any client data that is not verified is a potential security threat.
- What are we most afraid of?
  - Executing JavaScript in places that we did not plan, placed there by the attackers.
  - If an attacker can send a JavaScript snippet that gets *reflected* from our application into browsers of other clients, the attacker can run arbitrary code in browsers of those clients.

Validating and sanitizing data

- Any client data that is not verified is a potential security threat.
- What are we most afraid of?
  - Executing JavaScript in places that we did not plan, placed there by the attackers.
  - If an attacker can send a JavaScript snippet that gets *reflected* from our application into browsers of other clients, the attacker can run arbitrary code in browsers of those clients.
  - This is called a **cross-site scripting attack (XSS)**

Validating and sanitizing data

- Any client data that is not verified is a potential security threat.
- What are we most afraid of?
  - Executing JavaScript in places that we did not plan, placed there by the attackers.
  - If an attacker can send a JavaScript snippet that gets *reflected* from our application into browsers of other clients, the attacker can run arbitrary code in browsers of those clients.
  - This is called a **cross-site scripting attack (XSS)**
- We prevent such unwanted cases by validating data that is sent by clients. The validation can take place

Validating and sanitizing data

- Any client data that is not verified is a potential security threat.
- What are we most afraid of?
  - Executing JavaScript in places that we did not plan, placed there by the attackers.
  - If an attacker can send a JavaScript snippet that gets *reflected* from our application into browsers of other clients, the attacker can run arbitrary code in browsers of those clients.
  - This is called a **cross-site scripting attack (XSS)**
- We prevent such unwanted cases by validating data that is sent by clients. The validation can take place
  - on the **client side** (in the browser) or

Validating and sanitizing data

- Any client data that is not verified is a potential security threat.
- What are we most afraid of?
  - Executing JavaScript in places that we did not plan, placed there by the attackers.
  - If an attacker can send a JavaScript snippet that gets *reflected* from our application into browsers of other clients, the attacker can run arbitrary code in browsers of those clients.
  - This is called a **cross-site scripting attack (XSS)**
- We prevent such unwanted cases by validating data that is sent by clients. The validation can take place
  - on the **client side** (in the browser) or
  - on the **server side** (in the web application).

Client-side validation

- We use HTML elements and/or JavaScript that limit what the user can write into HTML forms.

Client-side validation

- We use HTML elements and/or JavaScript that limit what the user can write into HTML forms.
- `http://www.w3schools.com/html/html_form_attributes.asp`
- `http://www.w3schools.com/js/js_validation.asp`

Client-side validation

- Client-side validation **does not really work and is insufficient**, because attackers can trivially circumvent it.

Client-side validation

- Client-side validation **does not really work and is insufficient**, because attackers can trivially circumvent it.
- Client-side validation is meant to *help honest users* to provide valid data: to provide the correctly formatted email addresses or to remind them to fill in all fields etc.

Client-side validation

- Client-side validation **does not really work and is insufficient**, because attackers can trivially circumvent it.
- Client-side validation is meant to *help honest users* to provide valid data: to provide the correctly formatted email addresses or to remind them to fill in all fields etc.
- **Remember:** HTTP requests can be sent without the browser (recall telnet, curl, Requests library).

Client-side validation

- Client-side validation **does not really work and is insufficient**, because attackers can trivially circumvent it.
- Client-side validation is meant to *help honest users* to provide valid data: to provide the correctly formatted email addresses or to remind them to fill in all fields etc.
- **Remember:** HTTP requests can be sent without the browser (recall telnet, curl, Requests library).
- **We must always validate the data on the server.**

Server-side validation

- We use various PHP functions to validate and filter data before doing any additional processing.

Server-side validation

- We use various PHP functions to validate and filter data before doing any additional processing.
  - If we require a number withing specified range, we make sure that the data is a number and of appropriate size.

Server-side validation

- We use various PHP functions to validate and filter data before doing any additional processing.
  - If we require a number withing specified range, we make sure that the data is a number and of appropriate size.
  - Sometimes we apply filters to data and then proceed with processing: a filter simply strips or converts all illegal characters from the data.

Server-side validation: Useful functions

- `htmlspecialchars($input)` converts special characters to HTML entities:
  `<p>Text</p>` → `&lt;p&gt;Text&lt;/p&gt;`

Server-side validation: Useful functions

- `htmlspecialchars($input)` converts special characters to HTML entities:
  `<p>Text</p>` → `&lt;p&gt;Text&lt;/p&gt;`
- `is_numeric($input)` checks if a given value is numeric

Server-side validation: Useful functions

- `htmlspecialchars($input)` converts special characters to HTML entities:
  `<p>Text</p>` → `&lt;p&gt;Text&lt;/p&gt;`
- `is_numeric($input)` checks if a given value is numeric
- Casting comes in helpful

Server-side validation: Useful functions

- `htmlspecialchars($input)` converts special characters to HTML entities:
  `<p>Text</p>` → `&lt;p&gt;Text&lt;/p&gt;`
- `is_numeric($input)` checks if a given value is numeric
- Casting comes in helpful
- `strip_tags($input)` removes HTML tags; can be insecure if used improperly

Function filter_input()

- Functions `filter_input()` and `filter_input_array()` offer a nicer and more unified API to handle validation and sanitization of inputs

Function filter_input()

- Functions `filter_input()` and `filter_input_array()` offer a nicer and more unified API to handle validation and sanitization of inputs
  - `http://php.net/manual/en/function.filter-input.php`

Function filter_input()

- Functions `filter_input()` and `filter_input_array()` offer a nicer and more unified API to handle validation and sanitization of inputs
  - `http://php.net/manual/en/function.filter-input.php`
  - `http://php.net/manual/en/function.filter-input-array.php`

Function filter_input()

- Functions `filter_input()` and `filter_input_array()` offer a nicer and more unified API to handle validation and sanitization of inputs
  - `http://php.net/manual/en/function.filter-input.php`
  - `http://php.net/manual/en/function.filter-input-array.php`
- See *add book* example

Pitfalls of $_SERVER["PHP_SELF"]

- $_SERVER["PHP_SELF"] is a super-global that tells us the server-relative URL of the script that is currently being executed.

Pitfalls of $_SERVER["PHP_SELF"]

- $_SERVER["PHP_SELF"] is a super-global that tells us the server-relative URL of the script that is currently being executed.
- It is often used to set the form action attribute or to make links.

Pitfalls of $_SERVER["PHP_SELF"]

- $_SERVER["PHP_SELF"] is a super-global that tells us the server-relative URL of the script that is currently being executed.
- It is often used to set the form action attribute or to make links.

```
<form method="post"
  action="<?= $_SERVER["PHP_SELF"] ?>">
  <!-- form elements  -->
</form>
```

Pitfalls of $_SERVER["PHP_SELF"]

- $_SERVER["PHP_SELF"] is a super-global that tells us the server-relative URL of the script that is currently being executed.
- It is often used to set the form action attribute or to make links.

```
<form method="post"
  action="<?= $_SERVER["PHP_SELF"] ?>">
  <!-- form elements  -->
</form>
```

- Seems secure, since the value is set by the interpreter and not the client ...

Pitfalls of $_SERVER["PHP_SELF"]

- $_SERVER["PHP_SELF"] is a super-global that tells us the server-relative URL of the script that is currently being executed.
- It is often used to set the form action attribute or to make links.

```
<form method="post"
  action="<?= $_SERVER["PHP_SELF"] ?>">
  <!-- form elements  -->
</form>
```

- Seems secure, since the value is set by the interpreter and not the client ...
- **Not completely true.**

Pitfalls of $_SERVER["PHP_SELF"]

- What will be the value of `$_SERVER["PHP_SELF"]`, if instead of the valid URL, such as
  - `http://localhost/script.php`

Pitfalls of $_SERVER["PHP_SELF"]

- What will be the value of `$_SERVER["PHP_SELF"]`, if instead of the valid URL, such as
  - `http://localhost/script.php`
    the user requests the following URL

Pitfalls of $_SERVER["PHP_SELF"]

- What will be the value of `$_SERVER["PHP_SELF"]`, if instead of the valid URL, such as
  - `http://localhost/script.php`
    the user requests the following URL
  - `http://localhost/script.php/arbitrary/content?`

Pitfalls of $_SERVER["PHP_SELF"]

- What will be the value of `$_SERVER["PHP_SELF"]`, if instead of the valid URL, such as
  - `http://localhost/script.php`
    the user requests the following URL
  - `http://localhost/script.php/arbitrary/content?`
- Answer: `/script.php/arbitrary/content`

Pitfalls of $_SERVER["PHP_SELF"]

- What will be the value of `$_SERVER["PHP_SELF"]`, if instead of the valid URL, such as
  - `http://localhost/script.php`
    the user requests the following URL
  - `http://localhost/script.php/arbitrary/content?`
- Answer: `/script.php/arbitrary/content`
- Instead of `/arbitrary/content` an attacker can modify the link to contain JavaScript code snippet that will be reflected from the web application and loaded into the client's browser if your application prints the content of the `$_SERVER["PHP_SELF"]`. This is a XSS attack.

Alternatives to $_SERVER["PHP_SELF"]

- We sanitize the variable with function `htmlspecialchars()`

Alternatives to $_SERVER["PHP_SELF"]

- We sanitize the variable with function `htmlspecialchars()`
- We use an alternative constants, like

Alternatives to $_SERVER["PHP_SELF"]

- We sanitize the variable with function `htmlspecialchars()`
- We use an alternative constants, like
  - `basename(__FILE__)`

Alternatives to $_SERVER["PHP_SELF"]

- We sanitize the variable with function `htmlspecialchars()`
- We use an alternative constants, like
  - `basename(__FILE__)`
  - `basename($_SERVER["SCRIPT_FILENAME"])`