Web application security

Some examples and solution techniques

Read the book

"Foundations of Security: What Every Programmer Needs To Know“ by

Neil Daswani, Christoph Kern, and Anita Kesavan

Goals of an attacker

- Read private data: user names, passwords, credit card numbers, grades
- Change data: grades, prices of products, passwords
- Spoof: pretend to be someone they are not
- Damage or shut down the web site
- Harm the reputation
- Spread malware

These goals can be achieved by

- Cross-site scripting or HTML injection
- Denial of service (DoS): saturating the server
- Session hijacking
- SQL injection
- …

The security mindset

Assume nothing. Trust no one

RESOURCE DISCOVERY

Resource discovery

- Result of ordering a movie:

http://www.example.com/movies/2010/the-dark-knight?prepaid=true&price=4.99

- Try:

http://www.example.com/movies/2010/

http://www.example.com/movies/2010/avatar?prepaid=true&price=4.99

- Bad programming example:

Lecture: http://csci.viu.ca/~barskym/WP2013/NODE_JS/NodeLab.pdf

- Try:

http://csci.viu.ca/~barskym/WP2013/NODE_JS/

Guessing the file name

- We try to login, we know the name but not the password
- We try one guess and are forwarded to *login-failure.php*

- Try

manually change the URL to *login-success.php*, *login-ok.php*, maybe this will let us in

Changing GET parameters

http://www.example.com/videos/movies/2010/the-dark-knight?prepaid=true&price=4.99

- Try

Change value for price

Omit parameter

Pass other parameters: debug=true or admin=on

Examining page source and resource files

- We know how to see file directory
- We can see the source of the page, which supposedly requires authentication

```
<?php
// User name and password for authentication
  $username = 'rock';
  $password = 'roll';
  if (!isset($_SERVER['PHP_AUTH_USER']) ||
         !isset($_SERVER['PHP_AUTH_PW']) ||
         ($_SERVER['PHP_AUTH_USER'] != $username) ||
         ($_SERVER['PHP_AUTH_PW'] != $password)) {
// The user name/password are incorrect so send the authentication headers
         header('HTTP/1.1 401 Unauthorized');
         header('WWW-Authenticate: Basic realm="Guitar Wars"');
         exit('Sorry, you must enter a valid user name and password to access this
page.');
```

Generating and examining error messages

- Helpful (to a developer) error messages are dangerous security pitfalls

- Try to cause the page to generate an error and get interesting information from the error message
- Example: login page
- Enter invalid characters for user name, and get:

MANIPULATING FORM INPUT

Input validation

- Web applications often accept input from their users.
- To be secure, *web applications should not trust clients*, and should *validate* all input received from clients.

<input type="hidden" value="?">

- Hidden values in HTML forms are not directly shown to the user in the web browser's GUI.
- However, these hidden values can be easily manipulated by malicious clients.

- Data submitted from hidden form fields should be considered input and validated just like all other input, even though the server typically generates information that is stored in hidden form fields.

Example: pizza confirmation form

<FORM ACTION="submit_order" METHOD="GET">

The total cost is 5.50.

Are you sure you would like to order?

<INPUT TYPE="**hidden**" NAME="price" VALUE="5.50">

<INPUT TYPE="submit" NAME ="pay" VALUE="yes">

<INPUT TYPE="submit" NAME ="pay" VALUE="no">

Example: pizza confirmation form

<FORM ACTION="submit_order" METHOD="GET">

The total cost is 5.50.

Are you sure you would like to order?

<INPUT TYPE="**hidden**" NAME="price" VALUE="5.50">

<INPUT TYPE="submit" NAME ="pay" VALUE="yes">

GET /submit_order?price=5.50&pay=yes

- Once the web server receives the HTTP request, it then sends a request to a credit card payment gateway to charge $5.50.
- Once the credit card payment gateway accepts the charge, the web server can dispatch the delivery person.

Getting pizza for 0.01 $

- To change the value of the transaction: view the source code of the HTML form in a text editor, and change the value in the hidden form field from 5.50 to 0.01.
- Simply save the modified HTML to disk, reopen it with a browser, and submit the form with the modified price to the server!

GET /submit_order?price=0.01&pay=yes

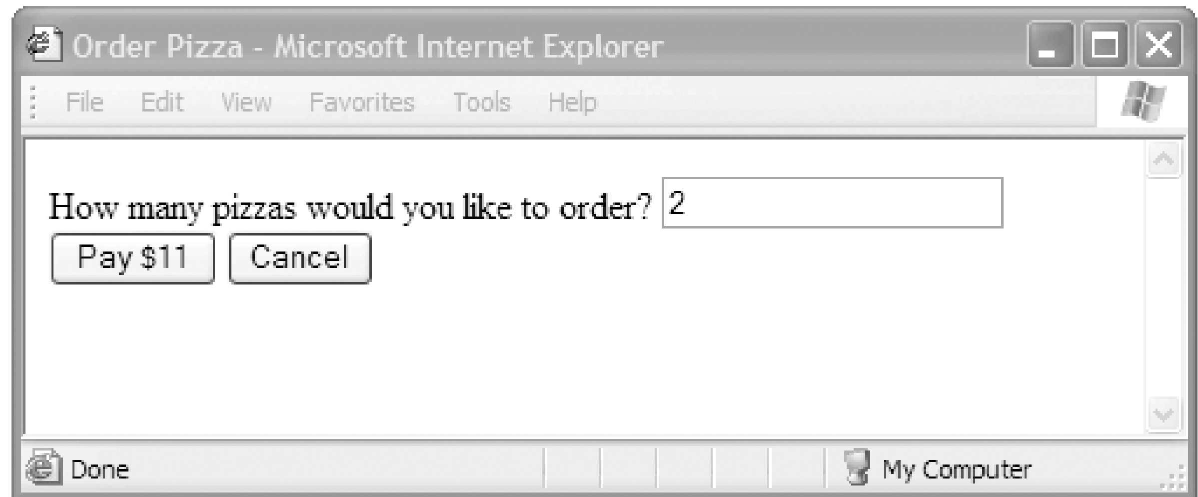
Hidden fields: conclusion

- Hidden form fields are only visually hidden from the user, but are effectively sent “in the clear” from a security standpoint.
- As such, they can be easily accessed and manipulated by malicious clients.

- Do not use hidden fields to maintain state of the application

JavaScript used to compute values of hidden inputs on client

```
<SCRIPT>
   function computePrice() {
      f.price.value = 5.50 * f.qty.value;
      f.order.value = "Pay $" + f.price.value
   }
</SCRIPT>
```



Bypassing JavaScript

- Again, simply save the HTML page to disk, delete the JavaScript from the HTML page, substitute 10000 for the quantity and 0 for the price, and submit the form.
- Alternatively, just submit an HTTP request such as

GET /submit_order?qty=1000&price=0&Order=Pay

- **Note for the programmer**: Do the price computation on the server, and charge the user the price that is computed by the server.

Ordering pizzas for all my friends

- Use tools such as curl (http://curl.haxx.se) or Wget (www.gnu.org/software/wget). These are open source, command-line tools that can be used to generate HTTP and other types of requests in an automated fashion.

curl https://www.deliver-me-pizza.com/submit_order?price=0.01&pay=yes

- Switching to POST would not help very much.

curl -dprice=0.01 -dpay=yes https://www.deliver-me-pizza.com/submit_order

wget --post-data 'price=0.01&pay=yes' https://www.deliver-me-pizza.com/submit_order

For the programmer

- There is no reason that the web server should trust any of its clients: By sending the transaction state back to the client in response to the order and confirmation forms, it gives the client the ability to tamper with that state.

- Possible solutions:

1. Keeping an authoritative copy of the session state in a database at the server.
2. Sending the authoritative state back to the client, but with a “signature” that will alert the server of any potential tampering with the state.

Solution 1:
Authoritative state stays at server

- The price of the transaction is not sent back to the client.
- In response to filling out an order form, the server randomly generates a new 128-bit session-id
- The server keeps a table of session-ids and the corresponding prices for client transactions.
- It sends it back as a hidden field in the confirmation form

```
<INPUT TYPE="hidden" NAME="session-id"
        VALUE="3927a837e947df203784d309c8372b8e">
```

- When submits, issues:

```
GET /submit_order?session-id=3927a837e947df203784d309c8372b8e&pay=yes
```

Now orders are less vulnerable, but still …

- Our only chance for a free pizza is to guess valid session-ids.
- If we are lucky, we could issue HTTP requests for these session-ids with pay=yes
- In a real application, some additional state is kept in the database: the customer's address, the quantity of pizzas, the user's credit card number, and other transaction details.
- We may be able to modify an existing order to include additional pizzas to be sent to our own address, but have the legitimate customer's credit card charged for the transaction!

- By choosing a 128-bit randomly generated session-id, our probability of success is *n* / 2128, where *n* is the number of session-ids in the server's database.

For the programmer: additional techniques

- You can have sessionids “**timeout**,” or expire after some time period: anyone who starts ordering a pizza should be able to complete their order within a *k*-minute period

If the user does not complete the order in *k* minutes, you have the right to just delete it from the database

- You can have the session-id be the “hash” of a **pseudo-random number and the IP address** that the web server reports the client is connected from.

If you use this technique, an attacker not only needs to guess a valid session-id, but also needs to spoof the IP address of the client in order to use the session-id.

Downsides of solution 1

- Your server-side infrastructure is no longer stateless.
- Every time an HTTP request arrives at your web server, a database lookup needs to be done, and could turn the database access into a performance bottleneck.
- If the database lookup takes nontrivial computational resources, an attacker could issue many such requests with random session-ids as part of a DoS (denial of service) attack.

Solution 2: signed state sent to client

- The authoritative state is returned to the client—but to prevent a client from tampering with the state, a “signature” of the transaction state is also sent.
- The server possesses a cryptographic key known only to the server which it uses to produce the signature.
- The client will not be able produce modified signatures to match the altered state, because it does not know the server’s key.

Signed confirmation form

- When a client fills out an order form, the server sends back a form that includes all the parameters of the transaction (including the price) and a signature:

```
<INPUT TYPE="hidden" NAME="item-id" VALUE="1384634">
<INPUT TYPE="hidden" NAME="qty" VALUE="1">
<INPUT TYPE="hidden" NAME="address" VALUE="123 Main St, .. ">
<INPUT TYPE="hidden" NAME="credit_card_no" VALUE="5555 ..">
<INPUT TYPE="hidden" NAME="exp_date" VALUE="1/2012">
<INPUT TYPE="hidden" NAME="price" VALUE="5.50">
<INPUT TYPE="hidden" NAME="signature"
        VALUE="a2a30984f302c843284e9372438b33d2">
```

Message Authentication Code (MAC)

- The signature was generated by computing a message authentication code (MAC) over all the other parameters of the transaction, including the item-id, quantity, address, credit card number, expiration date, and price.
- The MAC is a function of a cryptographic key only known to the server.
- If the client attempts to change the price or any of the other parameters, the client will not be able to recompute a corresponding signature because it does not know the key.

Cost of Solution 2 vs. Solution 1

- By using the signature-based approach, the server does not need to keep track of sessionids.
- It can continue to be stateless at the expense of having to compute MACs when processing HTTP requests and having to stream state information to and from the client.
- For state-intensive applications, the amount of extra bandwidth required to stream state may be more costly than the server-side storage required for user data in a session-id–based solution.

Solution 2 caveats

- The *entire* transaction state must be signed—not just part of it (such as the price).
- Otherwise, an attacker can conduct (part of) a legitimate transaction to coerce the server into generating a signature for her, and she can then conduct an illegitimate transaction by pasting in parameters of her choice that are not included in the signature.

For instance, if only the price is signed, the attacker can go through the order process having selected a cheap item to obtain a signature on the price, and then submit that signature and price in an HTTP request to purchase a more expensive item.

INPUT VALIDATION

Client-side input validation

- Verifying data on client before sending it to server
- Decreases load on server, decreases response time, but opens ways to invalid input

- Implicit validation: automatic enforcement of certain constraints
- Explicit validation: JavaScript code executed before *submit* event

Implicit client validation example

- Quiz application: uses hidden form inputs to keep track of user’s score

```
<input type=“hidden” name=“word” value=“<?= $answer ?>”/>
<input type=“hidden” name=“total” value=“<?= $total ?>”/>
<input type=“hidden” name=“correct” value=“<?= $correct ?>”/>
```

- Assumption: with the next submission, the values do not change

Getting high score

- Open page inspection tools and change the values to arbitrary numbers
- Submit the form

- Get result:

Your score: 100/100!

Explicit client validation

```
window.onload = function()
{
 document.getElementById(“frm”).onsubmit
                    =validate;
}
function validate ()
{
 if (value!=“one” && value!=“two”
                              && value!=“three”)
  {
    alert (“Freeze, hacker”);
    return false;
  }
 return true;
}
```

- In browser console set form.onsubmit=undefined
- Submit the form with invalid data

Rules of validation

Any client-side computation or validation should be repeated on the server

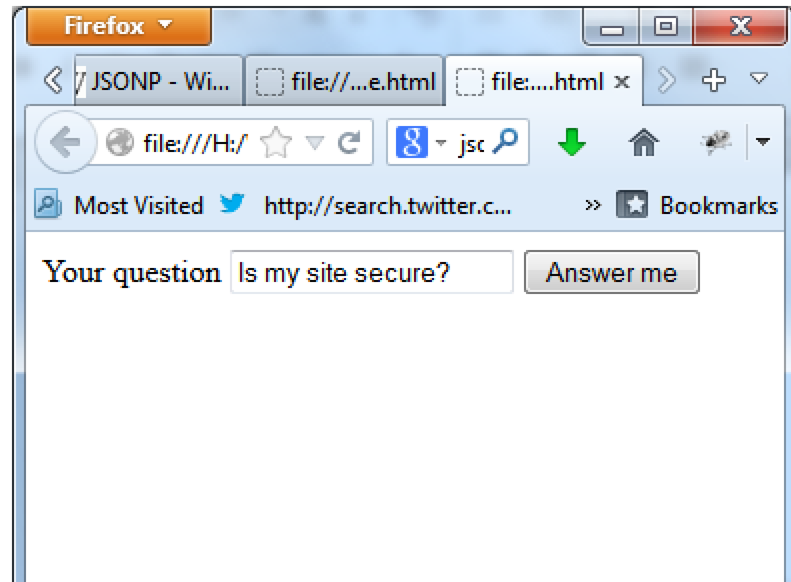
DANGEROUS INPUTS

Cross-Site Scripting (XSS)

- Submitting HTML or JavaScript code inside form fields and causing this code to appear on other pages

XSS example (link)

```
<FORM action="answer.php" method="post">
Your question <INPUT type="text" name ="question">
<INPUT type="submit" value="Answer me">
</FORM>
```



Changing HTML output

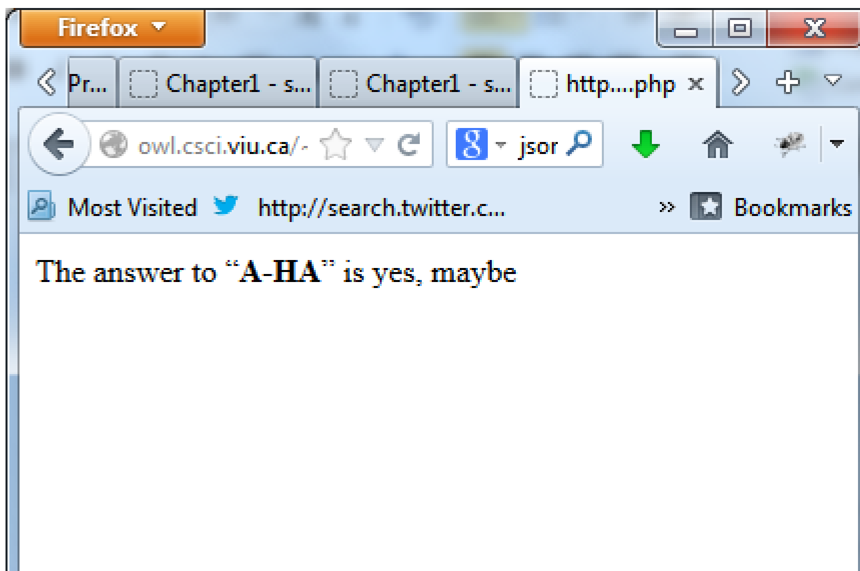
- Try input:

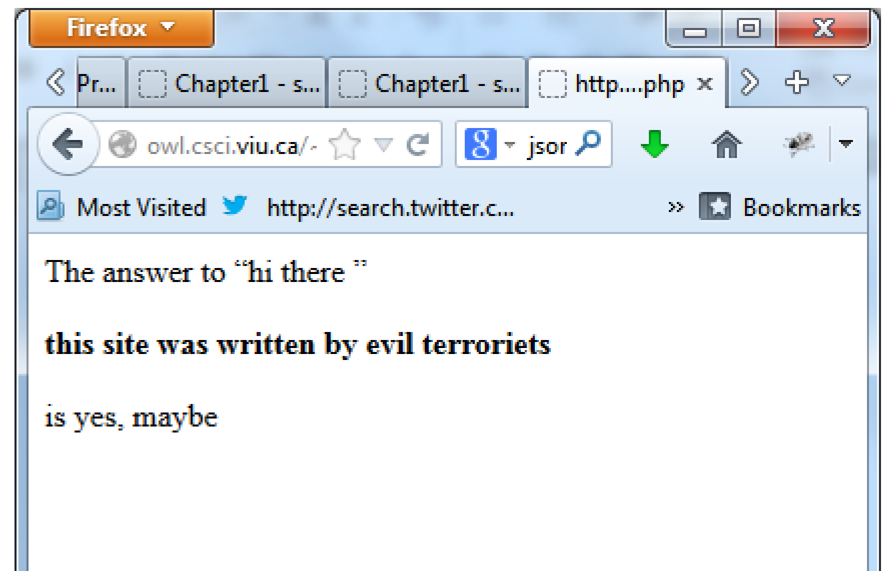
A-HA

<em style="border-bottom: 3px dotted green">HM-M

<script>alert("hello");</script>

hi there <p>this site was written by evil terrorists





Defending your text inputs: escaping special characters (PHP)

- htmlspecialchars (s) –Replaces some tags with HTML character entities
- htmlentity (s) – Replaces all characters that have an equivalent HTML entity with this entity
- htmlspecialchars_decode, htmlentity_decode – converts entities back to a plain text

```
$text = “<p> hi 2 u & me </p>
$text = htmlspecialchars ($text)
// @lt;p&gt; hi 2 u &amp; me @lt;/p&gt
```

SQL injection

- SQL injection is the act of inserting malicious SQL queries into an input field to reveal sensitive private data and make unwanted modifications

SQL injection examples

- Getting all data from the table

SELECT *fieldlist*

FROM *table*

WHERE *field* = '**anything' OR 'x'='x**';

- Discovering table names

SELECT *fieldlist*

FROM *table*

WHERE *field* = '**x' AND 1=(SELECT COUNT(*) FROM *tabname*); --**';

- Getting personal data

SELECT email, passwd, login_id, full_name

FROM members

WHERE email = '**x' OR full_name LIKE '%Bob%**';

And more examples

- Adding a new member

SELECT email, passwd, login_id, full_name FROM members

WHERE email = **'x'; INSERT INTO members ('email','passwd','login_id','full_name') VALUES ('steve@unixwiz.net','hello','steve','Steve Friedl');--'**;

- Mail me a “forgotten” password

SELECT email, passwd, login_id, full_name

FROM members

WHERE email = **'x'; UPDATE members SET email = 'steve@unixwiz.net' WHERE email = 'bob@example.com'**;

Preventing SQL injections

- Validate and escape submitted data
- Use PDO and bound variables

```
$db->quote ("oh no, 'quotes'!")
//returns "oh no, \'quotes\'!“

$stmt = $db->prepare("SELECT * FROM user WHERE name=:name AND password=:pwd");
$stmt ->bindParam (":name", $_POST["name"]);
$stmt ->bindParam (":pwd", $_POST[“password"]);
$stmt->execute();
```

AJAX AND JSONP

Examining Ajax requests

- You can see Ajax requests issued in JavaScript
- You can also see the results of Ajax requests using developer tools
- Then you can issue similar requests from your own pages

- Conclusions: no Ajax requests for money transfers for bank accounts

JSON-P: injecting JavaScript

- Including script tags from remote servers allows the remote servers to inject *any* content into a website.
- An effort is underway to define a safer strict subset definition for JSON-P that browsers would be able to enforce on script requests with a specific MIME type such as "application/json-p".
- If the response didn't parse as strict JSON-P, the browser would throw an error or just ignore the entire response.
- For the moment however the correct MIME type is "application/javascript" for JSONP.

JSON-P: possible scenarios

- A malicious page can request and obtain JSON data belonging to another site. This will allow the JSON-encoded data to be evaluated in the context of the malicious page, possibly divulging passwords or other sensitive data if the user is currently logged into the other site.
- Your page is requesting data from a malicious site: use your own callback function, you also know JSON format that you expect and can check that JSON objects passed as parameters do not contain any methods, just data
- Malicious RESTful services can potentially ignore your callback function and execute their own JavaScript functions, which can make use of private values of the page form inputs

JSONP - conclusions

- The JSON-encoded data should not contain sensitive information
- You should carefully treat results of JSONP requests in your own callback functions
- Use RESTful services only from the trusted sources