Security

Chapter 15

- Many of the examples we've created have been very insecure.
- The Internet is a rather hostile place.
- Hacking a website is not as easy as it looks on television, nor is it all that difficult.
- Even if you deem your website unimportant, if you store any kind of personal data, it is important to secure it.

There are many reasons someone may want to exploit your website.

- **Read private data**
- **Change data**
- **Spoofing**
- **Damage or shut down site**
- **Damage reputation or credibility of a site**
- **Spread viruses or other malware**

- **Brute Forcing** - attempt to guess passwords, file names, directory names. . .
- **Cross-Site Request Forgery (CSRF)** - leading a browser into making requests from a malicious page to another site, one for which the browser has previously stored a cookie, that perform malicious actions.
- **Cross-Site Scripting (XSS) or HTML Injection** - inserting HTML or scripts into a web page to be viewed or executed by others.
- **Denial of Service (DoS)** - make a resource unavailable to others by saturating the server with too many requests.
- **Exploits** - taking advantage of security problems in the underlying web software.

- **Improper error handling** - showing innappropriately specific error messages to the user.
- **Information Leakage** - allowing an unauthorized user to look at data, files, etc that he or she shouldn't be allowed to see (directory browsing). Discovering other URLs, files, etc is called *resource discovery*.
- **Malicious file exception** - causing arbitrary file content to be executed as a server-side program.
- **Man-in-the-middle** - many different problems with placing a machine between the client and the server.
- **Physical access attacks** - allowing an unauthorized user to have physical access to your server.

- **Privilege Escalation** - code that is run as a "privileged" user when it shouldn't be.
- **Session Hijacking** - unauthorized use of another user's session cookie.
- **Social Engineering** - tricking a user into willingly compromising the security of a site, for example, phishing.
- **SQL Injection** - inserting malicious SQL queries into a web site to reveal sensitive database information.

- An important part of web development (and programming in general) is to prevent attacks from happening.
- **Prevention** includes employing security practices like secure connections, firewalls, reduced user access/privileges, disabling unneeded web services and secure programming.
- No system is 100% secure.
- Another aspect of security is detecting a problem and responding to it.

- You need to first view your website as an attacker would view it.
- Let's say you have a web page to transfer money from one account to another:
  http://www.someWebsite.com/bank/transfer.php?
  srcAcctNum=12345&destAcctNum=12345&
  src=checking&dest=savings&amount=100.00
- What would be some reasonable attacks on this web site?
- Standard HTML error codes may also give information to an attacker. If you try to find a document called bob-jones.doc and you get a 403 error, how is this useful?

- URL-mangling is a simple attack.
- Let's assume we know a user's login but not the password. The website takes us to *login-failure.php*. What happens if we try and go to *login-success.php*, does it work?
- A URL that contains parameters is ripe for attacking. Can we omit the parameters? Can we change them to something we want? Can we add parameters?
- As of August 1, 2014, I can buy a rather expensive piece of software for almost nothing: http://www.allvirtualware.com/languages/reverso.htm

- An attacker can view the pages being requested by Ajax or something similar.
- If a requested URL is deposit.php, maybe we can go to that URL directly and see what happens.
- Helpful error messages in debugging can be dangerous when the site is live. You do not want an attacker to know your username and where a database is located.
- Other error messages might also contain information about the tables in a database, variables that aren't initialized in PHP and user names. Error messages to the public should be brief and non-descriptive. A simple "Something went wrong..." is good enough.

- A large number of attacks (84%) in 2007 were XSS attacks. http://eval.symantec.com/mktginfo/enterprise/white_papers/b-whitepaper_exec_summary_internet_security_threat_report_xiii_04-2008.en-us.pdf
- Cross-site scripting is inserting malicious code into a web page to be viewed by others.
- Most XSS attacks involve submitting HTML or JavaScript into a form and causing the attack to appear on the response page.

- Assume you have a textbox inside some form:
  <input type="text" name="something" />
- If your resulting PHP page does the following, it is not secure:

```
<?php
$theText = $_POST["something"];
?>
<p>You submitted <?= $theText ?></p>
```

- The problem is the query parameter was directly output.
- If the user enters in:
  <script type="text/javascript">/* anything here */ </script>
- If JavaScript can be written, an attacker can insert code that will run when others visit the website.

- The proper way to deal with XSS is to not trust anything the user types in.
- Should never output user generated input without making sure it is not HTML, JavaScript, etc.
- We could disallow < or & but they might be valid input.
- Several PHP functions accept a string and return a safe HTML-encoded version. Each potentially bad character, e.g. <, is replaced with the equivalent HTML character entity reference, e.g. <. This is called **HTML-encoding** a string.

```
<?php
$theText = htmlspecialchars($_POST["something"]);
$evenSafer = htmlentities($_POST["something"]);
//but usually unnecessary
?>
<p>You submitted <?= $theText ?></p>
```

- Other attacks involved submitting malicious or invalid input to a server.
- We want to make sure the data being submitted is valid.
- **Validation** is ensuring that user-provided data meets certain requirements. Examples include:
  - Preventing blank values.
  - Ensuring the type is correct.
  - Ensuring the data is in range.
  - Ensuring certain combinations are valid.
- Validation can be done on the client side, server side, or both.

- **Client-side validation** is the process of verifying data before it is sent to the web server.
- **Implicit Validation** - uses the web browser to enforce certain limitations. Mechanisms include hidden form fields, maxlength properties, checkboxes, radio buttons...
- Client-side validation relies on:
  1. the user is using the web browser to send data to the server.
  2. the user's web browser has not been modified to send unexpected data.
- Neither assumption is always true. An attacker can use the tools we've been using to debug to change values or other properties of the web page.
- Hidden fields are not secure fields.

- **Explicit Validation** - inspects each piece of data and evaluates whether or not it is valid.
- Functions can be written to validate input in JavaScript.
- This is important to give a better user experience. Don't need to keep sending information back and forth from client to server.
- However, this client-side validation is not secure. An attacker can use the JavaScript console to redirect functions.
- If we have this:

```
document.getElementById("myForm").onsubmit = validate;
```

- An attacker can open up the console and type:

```
document.getElementById("myForm").onsubmit = undefined;
```

- **Server-side Validation** - process of checking input parameters on the server receiving the HTTP request.
- This is done by simply writing PHP code to check if some value is valid.
- Does a zip code have 5 numbers? If possible, can you validate a zip code and the city?
- If you made the user enter in an e-mail address twice, does it match?
- If the data is invalid, you should not simply redirect back to the original form.
- Redisplay the form with the correct values and clear the incorrect values. Place some error message at the top of the screen or highlight the fields that need to be fixed, or both.

- **Regular Expressions** - also know as regex, is an encoded representation of a *pattern* of text characters.
- You can write regular expressions to describe almost any string you want. For example, only 5 digit strings, an e-mail address, words that start with "cat", words that contain the substring "cat" and many more.
- Regular expressions are not really a security feature but a useful tool to help with security.
- A simple tutorial can be found at http://regexone.com/

- Regular expressions typically start and end with a / characters. Between the slashes you write the sequence of characters you wish to match.
- Most regular expressions look for substrings. /hi/ would match all strings that contain hi.
- By default, the characters must appear in the same order and have the same capitalization. To ignore case, you would write an i after the expression. For example: /hi/i would match "CHIP."
- The . dot character is generally a wildcard and will match any single character. /h./i would match Hi, ha, HE, help!

- The | pipe operator indicates an "or." For example, /cs|3.6/ will match any string that contains cs or 3 followed by any character, followed by a 6.
- Parenthesis group as expected. For example, /ch(i|o)p/ matches any string that contains chip or chop.
- As with most languages, the backslash is an escape character. If you want to match a period, you would use: /\./
- If you want to match an entire string, you can use an anchor.^ at the beginning indicates the pattern must start with the string. $ at the end indicates the string must end with the string.

- /John/ matches all strings that contain John.
- /^John/ matches all strings that start with John.
- /John$/ matches all strings that end with John.
- /^John$/ matches all strings that are exactly John.
- /^John.McClane$/i matches all strings that start with John, are followed by 1 character and end with McClane (case-sensitive).

Quantifiers

- **Quantifiers** - allow you to match a pattern multiple times.
- The + plus operator means you want to match a pattern 1 or more times. For example: /^hi+/ means match all strings that start with h and are followed by 1 or more i's.
- If you want to match multiple characters, use parenthesis. /^(hi)+/ matches strings that start with 1 or more occurrences of hi.
- The * star operator means you want to match a pattern 0 or more times. /^hi*/ matches all strings that start with h and are followed by any number of i's.
- The ? question mark operator means you want to match a pattern 0 or 1 times exactly. For example: /^hi?/ means match all strings that start with h or start with hi.

Quantifiers

- It is possible to be more specific about the number of times to match a string. Sometimes 0 or more isn't specific enough.
- The {} quantifier accomplishes this.
- /c(at){2,3}/ matches strings that contain catat and catatat. The lower bound or upper bound can be omitted. /c(at){,3}/ says to match up to 3. /c(at){3,}/ says to match at least 3. /c(at){3}/ says to match exactly 3, it's identical to /catatat/.

Character Sets

- When you want to match one out of a given group of characters, you can use a **character set**.
- /Grade:[ABCDF]/ matches strings that contain Grade:A or Grade:D but doesn't match Grade:E.
- /[ABCDF]/ is equivalent to /(A|B|C|D|F)/.
- /[ABC]*/ matches AAB, BBBAAAAA, the empty string and CBACABBACCAB. Be careful, [ABC]* matches everything.
- Character sets can be shortened with a hyphen: [a-zA-Z]. For example, /[0-9]{5}/ matches exactly 5 digits.

Character Sets

- If you want anything but a set of characters, you can use the ^ symbol. For example, this matches anything but a lowercase letter: [^a-z]
- Several character sets are so common that they have shortcuts:
  - \d == 0-9
  - \w == a-zA-z0-9
  - \s matches any whitespace character
  - \D == ^0-9
  - \W == ^(a-zA-z0-9)
  - \S matches any non-whitespace character
- If you want to match a real number with exactly 1 digit after the decimal point: /^-?\d+\.\d$/

Back References

- A back reference is when you refer to a previous portion of the regular expression.
- Back references are not simply copies of the previous pattern but must match the previous pattern in the actual text.
- /\w+ \w+/ matches 2 words. "cat fish" would match this expression. So would "cat cat"
- If we want to force the 2 words to be the same, we must use a back reference. /(\w+) \1/ matches "cat cat" but does not match "cat fish"

- Most HTML form elements consist of input tags.
- HTML5 has some attributes for *client-side* validation.
- A **required** attribute must be filled out:
  <input type="number" name="zip" required="required" />
- A regular expression attribute is also available:
  <input type="number" name"zip" pattern="\d{5}" title="Zip code must be exactly 5 digits" />

- CSS3 pseudo-classes related to form validation are useful.
- http://www.w3schools.com/cssref/css_selectors.asp contains a complete list.
- For example, if we want our form to have a green background with valid fields and a red background for invalid fields, we can add the following to our CSS file:

```
:valid {
    background-color: green;
}
:invalid {
    background-color: red;
}
```

- If HTML handles validation, then why use JavaScript?
- The String object contains a match method and a replace method to handle regular expressions.
- replace and match do not affect the original string. If you want it modified, you must store it back into itself.

```
var str = document.getElementById("someId").innerHTML;
//assume str == "I like computers"
if(str.match(/[a-z]+/)){
    str = str.replace(/[aeiou]/, "*"); //I l*ke computers
    str = str.replace(/[aeiou]/g, "*"); //I l*k* c*mp*t*rs
    str = str.replace(/[^*]+/g, ""); //*****
}
```

- To actually validate data, we must do it on the server side.
- PHP has a few built in functions that use regular expressions to search and manipulate strings.
- PHP represents regular expressions as strings that start and end with a / symbol.
- http://php.net/manual/en/ref.pcre.php contains a full list of functions.

- The functions return the results, they don't modify the argument passed in. For example:

```
$orig = "I like computers";
$updated = preg_replace("/[aeiou]/i", "*", $orig);

print $orig; //I like computers
print $updated; //* l*k* c*mp*t*rs

//make sure zip code is 5 digits
$zip = $_POST["zip"];
if(!preg_match("/^\d{5}$/", $zip)){
    print "bad zip code";
} else{
    //do something with good zip code
}
```

- **SQL Injection** - inserting malicious SQL queries into a web site that reveal sensitive information or make unwanted modifications.
- Some websites receive hundreds of SQL injection attacks a day: www.imperva.com/docs/HII_Web_Application_Attack_Report_Ed4.pdf
- Ironically, mysql.com was compromised by a SQL injection attack: http://seclists.org/fulldisclosure/2011/Mar/309

- SQL injection attacks are pretty easy to create.
- What happens if the user searches for: it's

```
$db = new PDO("mysql:dbname=mydb", "user", "pword");
$rows = $db->query("SELECT * FROM someTable WHERE
                    someCol='" . $_GET["col"] . "'");
```

- When the user searches for it's, this SQL command is created: SELECT * FROM someTable WHERE someCol='it's
- This is not valid syntax and an error message would be displayed.
- The user was able to introduce a single quote which is valid SQL syntax.

```
$db = new PDO("mysql:dbname=mydb", "user", "pword");
$rows = $db->query("SELECT * FROM someTable WHERE
                    someCol='" . $_GET["col"] . "'");
```

Exposing Data

- What happens if the user enters this: ' OR '1'='1
  Our SQL command becomes:
  SELECT * FROM someTable WHERE someCol='' OR '1'='1'
- The WHERE clause is true for every tuple in someTable. Every tuple is returned.
- It would be bad if this table contained usernames, passwords, credit card information, SSN...

```
$db = new PDO("mysql:dbname=mydb", "user", "pword");
$rows = $db->query("SELECT * FROM someTable WHERE
                    someCol='" . $_GET["col"] . "'";);
```

- SQL injections allow the attacker to query and modify the table not originally intended to be queried.
- A SQL command called **UNION** can be placed between two SQL queries that return the same number of columns.

```
$db = new PDO("mysql:dbname=mydb", "user", "pword");
$rows = $db->query("SELECT * FROM someTable WHERE
                    someCol='" . $_GET["col"] . "'"
                    UNION
                    SELECT * FROM someTable WHERE
                    someCol='" . $_GET["col2"] . "'");
```

- What happens if the attacker enters this: ' UNION SELECT password FROM someOtherTable WHERE username='admin
- Even worse, the attacker could modify or delete data easily with the following: '; DROP TABLE users; –
- Renaming tables and accounts to something obscure is not sufficient.

```
$db = new PDO("mysql:dbname=mydb", "user", "pword");
$rows = $db->query("SELECT * FROM someTable WHERE
                    someCol='" . $_GET["col"] . "'");
```

- Never put user input directly into a SQL query.
- Always encode/sanitize the input.
- It is quite easy to sanitize the query. Use the quote method from the PDO object.

```
$db = new PDO("mysql:dbname=mydb", "user", "pword");
$better = $db->quote($_GET["col"]);
$rows = $db->query("SELECT * FROM someTable WHERE
                    someCol='" . $better . "'");
```

- It may be hard to verify you've sanitized all user input.
- Prepared statements are a more reliable way to prevent attacks.
- **Prepared Statement** - submit a (mostly) complete query to the database and have the server perform preprocessing steps ahead of time.
- Allows you to secure the structure of the query so it can't be tampered with.

```
$db = new PDO("mysql:dbname=mydb", "user", "pword");
$stmt = $db->prepare("SELECT * FROM someTable WHERE
                      someCol=? AND otherCol=?");
$stmt->execute(array($_GET["col"], $_GET["other"]));
```

- Another way to specify parameters is with names preceded by colons.
- This way is less error prone and is recommended.

```
$db = new PDO("mysql:dbname=mydb", "user", "pword");
$stmt = $db->prepare("SELECT * FROM someTable WHERE
                      someCol=:first AND
                      otherCol=:second");
$stmt->execute(array(":second" => $_GET["other"],
                     ":first" => $_GET["col"]));
```

- One final way is to bind the parameters before calling execute.

```
$db = new PDO("mysql:dbname=mydb", "user", "pword");
$stmt = $db->prepare("SELECT * FROM someTable WHERE
                      someCol=:first AND
                      otherCol=:second");
$stmt->bindParam(":first", $_GET["col"]);
$stmt->bindParam(":second", $_GET["other"]);
$stmt->execute();
```

- Another defense mechanism is to limit the permissions on the database.
- Set the relevant user to have access to only the tables it needs access to.
- **Principle of least privilege** - any action should be performed in a user context with the minimal set of permissions able to perform the action.
- If a user only needs to SELECT and INSERT, then only allows those permissions. The way to deny permissions in mysql is:
  DENY UPDATE ON dbName.tableName TO user;

- If you are storing passwords in a database, you must always hash the password. **NEVER** store plaintext passwords anywhere.
- The book recommends using MD5 to hash your passwords. It's better than not hashing at all but is still a bad idea.
- **password_hash(pword, algo)** - the preferred way to encrypt a password. http://php.net/manual/en/function.password-hash.php contains a manual explaining the details.

```
//this is how you encrypt the password
$password = $_POST["pword"];
$storeMe = password_hash($password, PASSWORD_BCRYPT);

//this is how you decrypt the password
//assume the hash is stored in $storeMe
password_verify($otherPassword, $storeMe);
```

- Sessions are very useful for remembering login information, cart products, etc.
- **Session Hijacking** - occurs when an attacker is able to acquire a valid session ID.
- Session hijacking can occur in the following scenario:
  - Attacker inserts JavaScript code into http://insecure.com/example.php using XSS.
  - That JavaScript code runs on your machine when you visit that page.
  - The JS code examines your document.cookie looking for PHPSESSID.
  - JS code inserts an img element into the page with a src attribute of http://stealingwebsite.com/takeID.php?id=$PHPSESSID
  - Your browser attempts to connect to that image and the bad URL is contacted.
  - The bad website then logs the session id.
- If interested, a full JavaScript example is found on page 616 of the book.

- A way to combat session ID's from being stolen and used is to encrypt traffic from client to server.
- You should use HTTPS if at all possible while sensitive information is being sent back and forth, not just on logins or checkout pages.
- A Firefox extension called Firesheep allows you to view and hijack sessions on your local network.

- **Cross-site Request Forgery** - tricks your browser by taking advantage of the implicit trust a session ID creates between browser and server.
- Attacker causes an insecure or compromised site to coerce your browser into making an HTTP request to a different site.
  - Suppose you log into yourbank.com
  - Any future HTTP request will include that cookie.
  - An attacker discovers how transfers are made at yourbank.com and creates a URL to transfer money out of your account.
  - The URL can be embedded in an image on badsite.com. If you were logged into your bank and visited that site, the URL would work.
  - The attacker can find a website you visit often with a vulnerability or non HTML-encoded forums and encode an image.

- If you have sensitive pages, how many pages lead to that page? It's likely only 1 or maybe a handful.
- Let's assume the sensitive page is sensitive.php and presense.php is the page leading up to it.
  - Create a unique token on presense.php.
  - Temporarily store that token in the user's session data as a session variable.
  - Embed that token inside presense.php's form to be sent back to the server on submission.
  - When the user loads up sensitive.php, make sure the token sent matches the one created initially.
- Many web development frameworks offer features to do this automatically as it is a bit cumbersome.