EXIN Secure Programming Foundation

Workbook

Tim Hemel
Guido Witmond

Edition: November 2014

SECURE PROGRAMMING FOUNDATION

Van Haren PUBLISHING

EXIN Secure Programming Foundation

Workbook

Tim Hemel
Guido Witmond

Edition: November 2014

SECURE PROGRAMMING FOUNDATION

Colophon

Title	EXIN Secure Programming Foundation – Workbook
Authors	Tim Hemel & Guido Witmond
Editors	Ralph Pisaturo Marianne Hubregtse Eline Kleijer
Publisher	Van Haren Publishing
ISBN Hardcopy	978 94 018 0250 5
ISBN eBook	978 90 820388 6 6
Edition	November 2014

Table of contents

Introduction

Cybercrime, data leaks and information security get more attention than ever in the news. Governments and companies dedicate more and more resources to these areas. However, most of that attention appears to be focused on reactive measures (“How do we catch the cybercriminals?”) instead of on preventive measures (“How do we make our systems secure?”). Although it is hard to measure, research reports indicate that building security in is worth the investment. Key in the software building process is education. If programmers do not understand the security of the software they are building, any additional investment in the process is useless.

The EXIN Secure Programming Foundation exam tests the knowledge of the candidate on the basic principles of secure programming. The subjects of this module are Authentication and Session Management; Handling User Input; Authorization; Configuration, Error Handling and Logging; Cryptography; and Secure Software Engineering.

The exam consists of 40 multiple choice questions. In this workbook you will find several multiple choice sample questions. The exam requirements are specified at the beginning of each Chapter, and the weight of the different exam topics is expressed as a percentage of the total.

Target audience

Everyone who wishes to prepare for the EXIN Secure Programming Foundation exam and everyone interested in the basics of Secure Programming:

- Programmers and software developers who have an interest in developing secure (web) applications.
- Auditors who will work with the Framework Secure Software.

Understanding Secure Programming/Introduction: exam specifications

1. Introduction (10%)

1.1 Security Awareness (2.5%)
The candidate can:
- 1.1.1 Recognize the tension between market demands and security.

1.2 Basic Principles (2.5%)
The candidate can:
- 1.2.1 Explain security jargon and STRIDE.

1.3 Web Security (5%)
The candidate can:
- 1.3.1 Describe HTTP security issues.
- 1.3.2 Explain the Browser Security Model.

1 Understanding Secure Programming

1.1 Introduction

Cybersecurity is getting more attention than ever before (for example: “Data Breach Today,” 2014; Kelly, 2014). Almost every day we read about hacking or a data leak in the news somewhere. Because most of the incidents are hidden, and victims are not likely to publish the damage, there is no accurate way of telling how big the damage caused by cybercrime really is (Mercuri, 2003).

1.1.1 Causes of insecure software

While a lot of attention goes to catching the cybercriminals and detecting cyberattacks, we see relatively little attention paid to the underlying causes of cybercrime: the exploitation of people's ignorance and the exploitation of flawed software. As programmers, we have limited influence over the former, but preventing flawed software is something that we can control. Writing secure software is perhaps the most effective method of preventing most of the cybercrime that exists today (Devanbu & Stubblebine, 2000).

Why does secure software development get so little attention? There are a number of **causes**, some of which you as a programmer may have encountered:

- no time or money
- the customer did not request this
- lack of knowledge
- malicious intent
- nobody would want to attack our software

The speed with which software is developed to get it to the market is incredible and leads to one of the main causes of insecure software. Because buyers cannot distinguish an insecure software product from a secure software product, software manufacturers are more willing to spend resources on extra features or not to spend those resources at all (Rice, 2008). The software that provides the most functionality for the lowest price is more likely to win. This is the *most important* reason for insecure software; a lack of liability for the safety of the software.

Even if the security of a software product was visible to a software buyer or manufacturer, security problems would still be introduced because it turns out that people are generally very bad at estimating risks. Our brains have been optimized to take quick decisions to survive in the savannah, and have not caught up with the modern IT environment. Therefore, we tend to downplay certain risks and exaggerate others (Kahneman, 2002). Because of this, software manufacturers may wrongly decide not to invest in certain security measures, while software buyers may decide that they do not need to be protected against certain threats.

1.1.2 The security jargon

When talking about security, we often use terms such as threat, risk, and attack. These sound very similar, and are often confused. However, there are subtle differences between the terms (Johnstone, 2010). Table 1.1 clarifies the differences between the terms.

Table 1.1 – Security Jargon

Term	Explanation
Attack	An attempt to abuse a system
Exploit	A method of abusing a specific vulnerability
Mitigation	A measure to lower the impact of a threat
Patch	A measure to remove a vulnerability from a system
Risk	The potential loss of value weighed against the gain. Usually expressed as likelihood times impact.
Threat	A potential attack
Vulnerability	A weakness that can actually be abused
Weakness	An erroneous construction that can reduce security

Security is defined as: protected against threats. In order to make software secure, we need to know what threats exist and how to protect against them. However, it is practically impossible to create a complete list of threats for a particular system. Therefore, we cannot promise 100% security. Fortunately, many threats are known threats and we know ways to protect against them.

1.1.3 The STRIDE acronym

One helpful method to gather threats is to apply the **STRIDE acronym** (Hernan, Lambert, Ostwald, & Shostack, 2006). STRIDE describes attack types. Table 1.2 explains the letters in the acronym.

Table 1.2 – The STRIDE Acronym Explained

Term		Explanation
S	Spoofing	Pretend to be someone or something else than you are.
T	Tampering	Modify stored data or data in transit.
R	Repudiation	Deny that you did or did not do something.
I	Information Disclosure	See information that you are not allowed to see.
D	Denial-of-Service	Make a system unavailable.
E	Elevation of Privilege	Do something without permission.

1.1.4 Attack surface

One of the conditions of successful exploitation by an attacker is access to the vulnerable parts of a system. The parts of the system that an attacker has access to, is called the **attack surface**. Different types of attackers can have a different attack surface. Different attackers can access different parts of the attack surface, e.g. an office cleaner might have physical access to the computer system while a cyberattacker can 'only' access via the network.

1.1.5 Trust zones

To determine the attack surface, we can divide a system into so-called **trust zones**, with each zone representing a certain degree of trust. The trust zones are separated by trust boundaries. Trust boundaries are good candidates for an attack surface.

1.2 Web security

A **trust boundary** that we “see” every day is the one between a web-server and a web browser. In the early days of the web, this boundary was clearly visible for programmers, but with the current web application frameworks, this is no longer the case. We use **AJAX calls** from browsers as if it was a call within the browser. Some frameworks even provide what looks like a server-side call, but in reality generate JavaScript code that executes in the browser. If a programmer is not aware of this trust boundary, it is easy to forget to implement certain security measures. Numerous tools exist that let an attacker perform attacks exactly on this trust boundary. The man-in-the-middle (MITM) proxies are the best known attacks of this type.

To become aware of the security implications of ignoring this trust boundary, we have to look at what happens there.

1.2.1 The HTTP protocol

HTTP is a protocol that consists of requests and responses. A typical web request is shown in Figures 1.3 and 1.4.

The HTTP request

The first word is called the **HTTP method** or **HTTP verb**. Then the **request path** is given, followed by the **HTTP version number**. After that, we see the **request headers**. Next follows the **request body**, which is separated from the request headers by an empty line.

The HTTP method

The HTTP method tells the web-server what operation to perform on the resource that is pointed to by the request path. The most well-known methods are `GET` and `POST`, but the HTTP standard defines more methods, such as `OPTIONS`, `PUT`, `DELETE`, `TRACE` and `CONNECT`. A web-server may implement other methods, for example to support the WebDAV protocol.

GET and POST

It is worth looking at the `GET` and `POST` requests. These two methods were invented as part of the CGI (Common Gateway Interface) standard that allowed processing of user input and dynamically generated responses by web applications. `GET` is intended for "read-only" operations, whereas `POST` will modify the application's state. From a security point of view it is important to look at the way that input is passed to the web application. In a `GET` request the input parameters are included in the request path, in a `POST` request they are included in the request body. The part of the request path containing the CGI input parameters is called the query string.

Anything in the request path can end up in the browser history, in server-side logs, in proxy logs, and in the so-called **'Referer' header**. This header is sent by the browser anytime you click a link or submit a form on a webpage. The header contains the originating webpage. In the early days of the web, system administrators could see who was linking to their webpage.

If the query string contains sensitive data, this data can thus be disclosed in several places.

Figure 1.3 - An example of a `GET` request

```
GET Http://localhost/insecure/public/Login.jsp?loginadmin&passadmin HTTP/1.1
Host: localhost
User-Agent: Mozilla/5.0 Gecko/20100101 Firefox/14.0.1
Accept: text/html,application/xhtml+xml,application/xml;q0.9,*/*;q0.8
Accept-Language: en-us,en;q0.5
Proxy-Connection: keep-alive
Referer: http://localhost/insecure/public/Login.jsp
Cookie: JSESSIONID73A4496C2B6C846B5E8D8C9DDDBD9D24
```

Figure 1.4 - An example of a `POST` request

```
POST http://localhost/account/login HTTP/1.1
Host: localhost
User-Agent: Mozilla/5.0 Gecko/20100101 Firefox/14.0.1
Accept: text/html,application/xhtml+xml,application/xml;q0.9,*/*;q0.8
Accept-Language: en-us,en;q0.5
Proxy-Connection: keep-alive
Referer: http://localhost/
Cookie: _session_id5bd5199e505bce65cffdc4c5ef02b617
Content-Type: application/x-www-form-urlencoded
Content-Length: 46

user_loginadmin&user_passwordadmin&x45&y20
```

Request headers

The request headers give the server additional information about the HTTP connection, the browser, the preferred language and encoding of the response, etcetera. For example, AJAX requests send the following header:

```
X-Requested-With: XmlHttpRequest
```

Accessing the request data

An application can access the input data through functions that the web application framework provides. For example, the `HttpServletRequest` class in Java provides methods to access `GET` or `POST` parameters, request headers and so forth. .NET provides a similar class, but in PHP one has to use the special arrays `$_GET`, `$_POST` and the function `apache_request_headers()`.

As a convenience, some frameworks provide a way to get a parameter regardless of its source. In .NET for example, the `HttpRequest.Params[name]` property will find the value of the parameter "name" by first searching the query string (`GET parameters`), then the form (`POST parameters`), then cookies and finally server variables (the server's configuration settings). Such convenience functions can lead to security problems. Suppose that an application relies on the server variable `REMOTE_USER` and accesses it through `HttpRequest.Params["REMOTE_USER"]`. If an attacker adds the `GET` parameter `REMOTE_USER=Administrator` to the query string, the application will think that the user that is logged in to the application is Administrator. An attacker can thus spoof certain values.

The HTTP standard is unclear on certain aspects. For example, it is not specified how to handle parameters, cookies or headers that occur more than once. This can lead to ambiguities, such as the situation where a web application firewall module will only see the first occurrence and the application uses the last occurrence. This problem is called **HTTP parameter pollution**.

Figure 1.5 – A typical HTTP response

```
HTTP/1.1 200 OK
Content-Type: text/html;charsetutf-8
Content-Length: 1744
X-Xss-Protection: 1; modeblock
X-Content-Type-Options: nosniff
X-Frame-Options: SAMEORIGIN
Server: WEBrick/1.3.1 (Ruby/1.9.3/2013-11-22)
Date: Mon, 04 Aug 2014 10:57:38 GMT
Connection: close

<!DOCTYPE html>
<html>
<head>
  <title> Home</title>
      ...
```

1.2.2 The HTTP Response

A typical **HTTP response** is shown in Figure 1.5. The response starts with HTTP and the version number of the HTTP protocol, then a status code, followed by a status description. Then the response headers are sent, and finally the response body. Headers and body are separated by an empty line.

The HTTP Status

The **HTTP status codes** are used to communicate how the server executed the request. For example:

- `Code 200` means everything went ok
- `Code 404` means that the resource was not found
- `Code 403` means that access was forbidden

Some codes influence how the browser behaves:

- `Code 302` will redirect the browser to the URL specified in the Location header
- `Code 401` will display a login window to authenticate to the web server

The HTTP response headers

Table 1.6 lists and explains some response headers. Some of the response headers are purely informational, such as the Server header. In certain cases, these headers can provide that an attacker can use to execute an attack more effectively. Detailed information on which web platform is used can be used to find specific vulnerabilities in that platform. Other headers influence browser behavior, sometimes in combination with a certain status code. Some headers are relevant to security, such as the Cache-Control header, that can be used to prevent sensitive information from being (unintentionally) stored on a caching server. There are even special headers that influence the security behavior of the browser, but a discussion about these headers is outside the scope of this workbook.

Table 1.6 – The meaning of a few response headers

Response header	Meaning
`Content-Type`	Provides a MIME-type, telling the browser how to handle the response body. Some browsers, most notably Internet Explorer, ignore this header and inspect the content to determine the type (content sniffing).
`X-Content-Type-Options`	With the value `nosniff`. Prevents content-sniffing and uses the Content-Type header.
`Content-disposition`	Tells the browser to either display the content in the browser (with value inline), or delegate processing to an external program or save to disk (with value attachment).
`Location`	With a 302 status code, the browser will open the URL specified by this header.
`WWW-Authenticate`	Allows the browser to ask for authentication credentials.
`Set-Cookie`	Stores information in a browser cookie.
`Cache-control`	Tells whether and how the information is allowed to be cached. *`no-cache`* means that the content is always revalidated with the server, *`no-store`* means that data is not allowed to be stored on non-volatile storage.

HTTP header injection

If a response header's value is constructed based on user input, an attacker may not only be able to influence this value, but also add extra response headers or even a complete response body. If an attacker adds a carriage return followed by a line feed character (CRLF), a new line is started in the response, which can be followed by an arbitrary response header. A double CRLF will include an empty line, after which the response body can be given. Such an attack is called a **header injection attack**. The attack will only work if the CRLF characters are passed as-is into the HTTP response, some web frameworks prevent this. In the past, injecting a complete body in a response allowed one to confuse the synchronization of web proxies, resulting in information leakage and manipulation of other people's HTTP responses. That attack is known as **HTTP response splitting**.

1.2.3 The Browser security model

With the advent of dynamic content in the browser, the need for security within the browser increased. This resulted in a **browser security model** (Horst & Nordhoff, 2008). One specific part of this model, the Same Origin Policy, describes under which conditions scripts and programs on websites can access content on other websites.

The basic principle behind this same origin policy states that scripts in one site's context are not allowed to access elements in another site's context. This applies to accessing document elements (DOM elements), cookies, performing `XmlHttpRequests` and accessing HTML5's local storage. The details of this policy differ per browser and even per browser version. Simple Object Access Protocol (SOAP) is a protocol that sends XML-messages over HTTP.

Recently some standards have been proposed that implement a more fine grained control over browser security with the help of several HTTP headers. As these standards are still being developed, we leave them outside of this workbook's scope.

Exam preparation: Chapter 1

To help prepare for the exam, we have included multiple choice questions (the answer key can be found at the end of this workbook). Additionally, you are provided with an overview of terms with which you should be familiar at the end of this chapter.

Sample questions

1 / 40
Attackers and defenders are two players within the field of security.

Why is the attacker at an advantage?

A. An attacker only needs to find one flaw; defenders need to consider all possible flaws.
B. An attacker is more skilled and determined than a defender.
C. An attacker abuses new technologies that a defender has to install.
D. An attacker has more computing power that he can use for performing all kind of attacks.

2 / 40
Some well-known *security principles* are used when designing secure systems. One of them is an application design that prevents single points of failure with security redundancies and layers of defense.

What principle is used to accomplish this design?

A. Defend in depth
B. Fail securely
C. Grant least privilege
D. Separate privileges

3 / 40
The following authorization header is sent by the browser to the server in response to a "`401 Authorization Required`" response:

```
Authorization: Basic bmFtZTpwYXNzd29yZA==
```

Is it safe to send this header using the HTTP protocol?

A. Yes, because the value is encrypted and cannot be reversed.
B. Yes, because the value is used only once and changes with every request.
C. No, because the value is encrypted using a weak algorithm.
D. No, because the value can be sniffed and reversed to valid credentials.

4 / 40

The term *SOP* most commonly refers to the mechanism that controls access for JavaScript and other scripting languages to the DOM properties and methods across domains.

Which conditions have to be satisfied to grand access between two interacting pages that do NOT use the document.domain property?

A. Protocol, IP number and - for browsers other than Microsoft Internet Explorer - port number.
B. Protocol, domain name and - for browsers other than Microsoft Internet Explorer - port number.
C. Protocol, PTR record and - for browsers other than Microsoft Internet Explorer - port number.
D. Protocol, FQDN and - for browsers other than Microsoft Internet Explorer - port number.

Exam Terms

Causes	Tampering	GET and POST
Hacking	Threat	HTTP method
Attack	Vulnerability	HTTP verb
Denial-of-Service	Weakness	HTTP version number
Exploit	AJAX calls	'Referer' header
Information Disclosure	Attack surface	Request body
Mitigation	Brute force attack	Request path
Patch	Core dump leaks	HTTP parameter pollution
Repudiation	Malware	Request headers
Risk	Phishing	HTTP status codes
Security	Timing attack	Header injection attack
Spoofing	Trust boundary	HTTP response splitting
STRIDE acronym	Trust zones	Simple Object Access Protocol (SOAP)

Authentication and Session Management: exam specifications

2. Authentication and Session Management (15%)

2.1 Passwords (5%)
The candidate can:
- 2.1.1 Identify problems involved in password usage.
- 2.1.2 Apply principles of password management.

2.2 Session Management (7.5%)
The candidate can:
- 2.2.1 Explain how Session Management works.
- 2.2.2 Recognize problems in Session Management.
- 2.2.3 Recognize best solutions for problems in Session Management.

2.3 Cross-Site Request Forgery (CSRF/XSRF) and Clickjacking (2.5%)
The candidate can:
- 2.3.1 Recognize problems and solutions of CSRF and Clickjacking.

2 Authentication and Session Management

2.1 Authentication and Session Management

2.1.1 Authentication

Authentication is a measure to determine the identity of a party with which we communicate. Examples are: passwords, biometric methods such as fingerprint recognition, or a smart card containing a client certificate. After initial authentication, usually a so-called session is created during which the server reliably knows the identity of the other party. In order to assure this, secure **session management** is important.

Passwords

Passwords are a secret that is shared between two parties, for example a user and a web server. One party presents the password, while the other verifies it. It is presumed that both parties keep the password secret. If it leaks, it can no longer be relied on for authentication: anyone who possesses the password can spoof the identity of the party that must present it, and the server cannot tell the difference.

Protecting the password in transit

The best way to **protect a password** from being eavesdropped is by not sending it. There are protocols to perform password authentication without transmitting the password to the server, sometimes called zero-knowledge proofs or challenge response authentication. If implemented correctly, a third party who sniffs or tampers with the connection between the user and server will not be able to spoof someone's identity. In web applications, such protocols are not deployed widely. This means that for most web applications, a password will be sent to the server.

To protect the password in transit, the connection must be encrypted. Usually this requires the use of the HTTPS protocol. The details of HTTPS will be discussed in the chapter on cryptography.

Password entropy

Instead of eavesdropping on a password, an attacker could simply guess the password. If the password is short and simple, it will be easy to guess. In an *online* password guessing attack, the attacker sends password guesses to the server and sees which ones are accepted.

To prevent an attacker from trying all combinations there are some mitigations:

- Limit the number of authentication attempts. This can be permanent or temporary.
- Implement a deliberate delay in the authentication process, to slow down an attacker.
- Lock the account after a certain number of tries; limit the total number of tries.

The last mitigation, locking the account, is very effective against a brute force attempt, but introduces an easy Denial-of-Service attack: an attacker can lock an arbitrary account by executing enough invalid login attempts for that account.

As computers get more powerful each year, **cracking passwords** gets easier. To counter that threat, password strength needs to increase.

Two methods for making passwords stronger exist:

- increase the character set for the passwords
- increase the length of the password

To prevent people from choosing easy to guess passwords, a certain complexity can be enforced in a password policy. The complexity of the password is called **password entropy** (Ma, Campbell, Tran, & Kleeman, 2010).

However, passwords that are more complex are more difficult to remember. It is not always easy to find the right balance. Password manager software shifts the balance to the more complex passwords and prevents people from reusing passwords at multiple sites.

Secure password storage on the server

Storing passwords on the server can pose a huge risk. If an attacker gets access to the server and reads the password store, the attacker has access to all passwords. As some users will choose the same password for more than one service, the problem of leaked passwords may be bigger than just a compromised web application. Hashing and salting increases the security of stored passwords.

To prevent the passwords from being readable, they must never be stored as clear text. Instead, it is better to store not the passwords themselves, but a so-called hash of the passwords. A **hash** is a value that is easy to compute given a password. However, it is extremely difficult to compute the password from a hash. At the same time, the chance that two users have the same hash value for different passwords (a so-called collision) is extremely low. By computing the hash of the password that is entered, and comparing it to the stored hash, the server can tell with confidence whether the password was correctly entered.

Hashes have the property that the same input leads to the same hash-value. That means that if two people choose the same password, this will show in the password file. To protect against this, a so-called **salt** is used. This is a random value that is concatenated with the password before it is hashed. This has two benefits. Not only will two users with the same passwords (very probably) have a different hash, but also the cracking of password hashes will be made more difficult.

Cracking password hashes is a method for guessing passwords that is performed offline. An attacker can spend as much time and resources as his budget allows for. Recently, the so-called Rainbow Tables have become very popular. This is basically a pre-computed list of password hashes for a given list of passwords. By comparing the password hashes to the pre-computed hashes, an attacker can discover passwords that are on this list. By salting the passwords, the hashes would need to be pre-computed for every salt value, which increases the length of the list significantly.

In addition to implementing hashes and salts, the leaking of password hashes must be prevented.

Password reset procedures

An additional problem with passwords is that people tend to forget them. In order to give users a new password, the identity of the person needs to be re-verified. How this needs to be done

depends on the application for which authentication is needed. A banking website may have stricter identification requirements than a forum to discuss trout fishing techniques.

For some applications, email addresses can be used as a substitute identity, assuming that the email address can only be accessed by the user.

One method is to ask for an email address at account creation time. When a user indicates that they lost their password, the application sends a link to a password reset form.

Another method is to generate a new password for the user and send it to the user's known email address. This has the benefit that the server can determine the strength of the password, and prevents people from using the same password at multiple sites. The downside is that the email containing the password may leak, or gets stored for long. Another disadvantage is that a user will not remember the password.

Given that many people have anywhere between 20 and 100 password protected accounts or places, it can be expected that most people either write down passwords, or use the same password more than once (Ives, Walsh, & Schneider, 2004). Password manager software is an excellent help to let people manage their passwords in a reasonably secure way.

2.1.2 Session Management

By nature, HTTP(S) is a stateless protocol; each request is independent of another. This would require a user to type their username and password at every click. That would be unworkable.

To tie a series of requests together, HTTP sessions have been invented. At the start of the session, a session identifier is issued. This session identifier is then sent with every request and is used to access server side **session data** during the session. At the end of a session, the data on the server can be cleaned up and the session identifier no longer points to that data.

To use sessions in combination with authentication, a number of guidelines must be observed.

After **logging** in, the session is created and the server generates a session identifier. This session identifier must be difficult to predict, as knowing the session identifier would give an attacker access to the session data of a particular user. It is worth mentioning that if the session identifier is sent via an unencrypted communications channel, an attacker can eavesdrop on it. The encryption is not only needed for the authentication step to protect the password, but for the complete session.

At the end of the session, the session should be invalidated. This means that the session identifier will no longer point to the session data, and that the session ID will become invalid upon ending a session. Because a user may forget to explicitly log out, the session should be invalidated after a certain period of inactivity.

Because the session identifier is something that should be kept secret, it is good practice not to re-use the session identifier for another session.

For the same reason, a session identifier should never be used as part of the query string of a request. Using cookies instead of CGI parameters will ensure this. Cookies are automatically sent with every request and can be revoked. This makes them appropriate for session management.

So far we have focused on the secrecy of the session identifier. However, an attacker's main goal is getting access to the actual session, whether that happens through learning the session identifier or not. What if an attacker, instead of guessing the correct session identifier, could simply determine what the session identifier will be? In some cases, this is possible. One method is by using a feature that is provided by some frameworks, called cookie-less sessions. Despite the fact that using CGI parameters for session identifiers is not a good idea, some frameworks support this for browsers that do not support cookies. In such a case, an attacker can let somebody visit a URL such as:

https://bank.com/;jsessionid=ABCDEFG1234567

In certain circumstances, which are beyond the scope of this workbook, the application may accept the session ID value as the session identifier that will be used. Since many web applications already start a session before the user logs in, this session will have been created. The attacker waits until the user logs in and can then access the session data with the above mentioned session identifier. This attack is called **session fixation**. A simple mitigation to this problem, although not treating the root cause, is to generate a fresh session identifier immediately after logging in, and to make sure that the earlier session identifier does not point to confidential session data.

Summarizing:

- The session ID should be difficult to predict. Generate a large random token to ensure this.
- After logging out, the session should (immediately) be invalidated.
- The session should be invalidated after a certain time of inactivity.
- The session ID should not be reused. Generating a large random token makes this very unlikely.
- Immediately after logging in, the session ID should be refreshed and the old session ID should be invalidated.
- Cookies should be used, and not CGI parameters.
- Session cookies should be sent via an encrypted channel.

Other measures can mitigate the problem of an attacker accessing the session, but they are either not very effective or lead to other problems. One could for example limit the number of simultaneous sessions, or tie the session to a certain IP-address. This introduces problems when a user's browser crashes or people get a different IP-address because of network roaming.

2.1.3 CSRF and clickjacking

The security issues in this chapter have thus far taken place on the HTTP protocol level. They involve spoofing or hijacking. On the browser level, similar attacks exist: cross-site request forgery and clickjacking.

Cross Site Request Forgery

Cross-Site Request Forgery (CSRF or XSRF) abuses the fact that one can refer to content on another website and let the browser make a request to get that content, without needing the explicit consent of the user. For example an attacker could embed the following external image on his website:

```
<img src="https://e-
voting.com/vote.cgi?candidate=George+Washington">
```

When someone visits the attacker's website, the browser will automatically request the above URL and vote for George Washington. Because the web-server cannot tell whether the request was explicitly requested by the user or was loaded because it was cross-linked from another website, it cannot reject the request.

Since the request is completely predictable, it is easy to execute this attack. Therefore, the key to mitigating this attack is to make the request unpredictable. This is done as follows:

- The application stores a random value in the session data. This value is called a **nonce** or **CSRF-token**.
- The CSRF-token is then included in every request.
- When the request is made, the application will first verify whether the token in the request matches that in the session data, before it will execute the request.

As a result the attacker cannot predict the correct value to put in the request, hence a successful attack is blocked.

Clearly, the CSRF-token should be kept secret. In `GET` requests, this could be a problem, but since the CSRF-token has a limited validity, this may be acceptable. Ideally, every form should have its own CSRF-token, but this may lead to usability problems when people open multiple browser tabs or press the browser's 'back' button. In practice, the CSRF-tokens are unique per login session.

Some frameworks add automatic CSRF-tokens to all forms (`POST` requests), but not to `GET` requests. Because we want to protect the unauthorized execution of operations, which are usually implemented as a `POST` request and not a `GET` request, this may make sense. A `GET` request that only shows information without further state changes, may be of no use to an attacker, since the attacker will not be able to see that information (after all, it is requested and rendered by the user's browser). Unfortunately, this is not entirely true. Requests that return JSON and JSONP data can be read by cleverly loading them inside of a script tag:

```
<script src="...">
```

This technique is called JSON hijacking or JavaScript hijacking (Haack, 2009). Keep in mind that `GET` requests may need CSRF protection too.

Similarly, AJAX requests must be protected as well.

Clickjacking

Thanks to CSS and JavaScript we are able to change the look and feel of user interface elements in our webpage. We can even stack elements on top of each other in layers. Clickjacking abuses these techniques to perform an attack on the user interface layer.

The basic **clickjacking attack** is done as follows. An attacker embeds the target website in a frame, and overlays it on the current window. With CSS, he sets the opacity to 0%, which makes the frame completely transparent. A person, who visits that website, will load both the attacker's and the target websites, but will only see the attacker's website. The clicks and key presses however, are all caught by the top layer - the invisible target site. This way, an attacker can convince somebody

to click buttons on a page that is invisible, and therefore let that person execute operations on a certain webpage.

Clickjacking is an attack where an attacker uses CSS and or JavaScript to make a victim click on something that the victim does not intend to.

Variations on clickjacking attacks exist where a user is asked to double click, but after the first click the application rapidly displays a new window containing the target webpage with a button at the same location, just in time to receive the second click. It is very difficult to defend against this last particular attack. However, the basic clickjacking attack can be prevented.

Modern browsers support the X-Frame-Options HTTP header. By setting the header to the value `SAME-ORIGIN` or `DENY`, a browser will refuse that page to be embedded as a frame into another website.

Older browsers sometimes use JavaScript code to detect if they are loaded in a frame. This code is called **frame-busting code**. However, its effectiveness is limited.

Exam preparation: Chapter 2

To help prepare for the exam, we have included multiple choice questions (the answer key can be found at the end of this workbook). Additionally, you are provided with an overview of terms with which you should be familiar at the end of this chapter.

Sample questions

5 / 40
An application allows a user who is logged in to change his/her password. This function is only available for authenticated users and uses the HTTPS protocol to send the data.

What is considered BEST PRACTICE to perform the password change?

A. Ask the user for the login name, old password, new password and confirmation of this password.
B. Ask the user for the new password and confirmation of this password.
C. Ask the user for the old password, new password and confirmation of this password.
D. Use the session data to identify the user and ask for the new password and confirmation of this password.

6 / 40
When storing passwords in a file or a database, what is the BEST approach?

A. Store the hashed value of the password that the user has chosen including a random salt.
B. Store the hashed value of the password that the user has chosen including a fixed salt.
C. Store the plain text value of the password that the user has chosen.
D. Store the encrypted value of the password that the user has chosen including the initialization vector.

7 / 40
HTTP sessions are used to keep state between several requests and use a session ID for identification.

What is the MOST important practice in regard to the session ID?

A. The session ID should be kept secret at all times.
B. The session ID should change with every POST request.
C. The session ID should be at least ten characters long.
D. The session ID should be encrypted using a strong algorithm.