# Securing Web Applications

Lecture 09

# Learning Outcomes

After completing this lesson you should be able to

- Explore different web  security vulnerabilities
- Explain best practices for securing web application
- Explore mechanisms available in Go to ensure web application security

# Security Attack Categories

**Spoofing** impersonating something or someone else

**Tampering** modifying something you're not supposed to modify. It can include packets on the wire (or wireless), bits on disk, or the bits in memory

**Repudiation** claiming you didn't do something

**Denial of Service** attacks designed to prevent a system from providing service, including by crashing it, making it unusably slow, or filling all its storage

# Security Attack Categories

**Information Disclosure** exposing information to people who are not authorized to see it

**Elevation of Privilege** when a program or user is technically able to do things that they're not supposed to do

# Threat Mitigation Approach

What can be done to prevent these attacks?

| Threat Type | Property Violated | Mitigation Approach |
|---|---|---|
| Spoofing | Authentication | |
| Tampering | Integrity | |
| Repudiation | Non-repudiation | |
| Information Disclosure | Confidentiality | |
| Denial of Service | Availability | |
| Elevation of Privilege | Authorization | |

# Threat Mitigation Approach

| Threat Type | Property Violated | Mitigation Approach |
|---|---|---|
| **Spoofing** | **Authentication** | Passwords, Multi-Factor Authentication, Digital Signature |
| **Tampering** | **Integrity** | Permissions/ACLs, Digital Signature |
| **Repudiation** | **Non-repudiation** | Secure Logging and Auditing, Digital Signature |
| **Information Disclosure** | **Confidentiality** | Encryption, Permissions/ACLs |
| **Denial of Service** | **Availability** | Quotas, Permissions/ACLs |
| **Elevation of Privilege** | **Authorization** | Permissions/ACLs, Input Validation |

# Securing Go Web Application

# Input Validation

In validation checks, the user input is checked against a set of conditions in order to guarantee that the user is indeed entering the expected data

**IMPORTANT:** If the validation fails, the input must be rejected

This is important not only from a security standpoint but from the perspective of data consistency and integrity, since data is usually used across a variety of systems and applications

# Input Validation

Any part of an application that allows user input is a potential security risk

Problems can occur

**from threat actors** that seek a way to compromise the application

**from erroneous input** caused by human error (statistically, the majority of the invalid data situations are usually caused by human error)

# Input Validation

## Handling Strings

`strconv` package handles string conversion to other data types

```
Atoi, ParseBool, ParseFloat, ParseInt
```

`strings` package contains functions that handle strings and its properties

```
Trim, ToLower, ToTitle
```

# Input Validation

## Handling Strings

`regex` package support for regular expressions to accommodate custom formats

`utf8` package implements functions and constants to support text encoded in UTF-8

includes functions to translate between runes and UTF-8 byte sequences

# Input Validation

## Handling Strings

Validating UTF-8 encoded runes:

```
Valid, ValidRune, ValidString
```

Encoding UTF-8 runes:

```
EncodeRune
```

Decoding UTF-8:

```
DecodeLastRune, DecodeLastRuneInString, DecodeRune,
DecodeRuneInString
```

# Input Validation

**Other techniques to ensure the validity of the data include**

Whitelisting, Boundary Checking, Character Escaping, Numeric Validation, Check for Null Bytes - (`%00`), Checks for New Line Characters - `%0d`, `%0a`, `\r`, `\n`, Checks for Path Alteration Characters - `../` or `\\..`, Checks for Extended UTF-8 - check for alternative representations of special characters

**What is the difference between whitelisting and blacklisting? What are the advantages of choosing one over the other?**

# Input Validation

## File Manipulation

Any time file usage is required ( **`read`** or **`write`** a file ), file check procedures such as **file existence check**, to verify that a filename exists should be performed

# Input Validation

## Data sources

Anytime data is passed from a trusted source to a less-trusted source, **integrity checks** should be made

This guarantees that the **data has not been tampered with** and we are receiving the intended data

Other data source checks include:

**Cross-System Consistency Checks**, **Hash Totals**, **Referential Integrity**

# Post-validation Actions

## Enforcement Actions

**Inform the user that submitted data has failed** to comply with the requirements and therefore the data should be modified in order to comply with the required conditions

**Modify user submitted data on the server side** without notifying the user of the changes made. This is most suitable in systems with interactive usage

# Post-validation Actions

## Advisory Action

Advisory Actions usually **allow for unchanged data to be entered, but the source actor is informed that there were issues with said data**. This is most suitable for non-interactive systems.

# Post-validation Actions

## Verification Action

Verification Action refer to special cases in Advisory Actions. In these cases, **the user submits the data and the source actor asks the user to verify the data and suggests changes**. The user then accepts these changes or keeps his original input.

# Sanitization

Sanitization refers to the process of removing or replacing submitted data

When dealing with data, after the proper validation checks have been made, sanitization is an **additional step** that is usually taken to **strengthen data safety**

# Sanitization

In the native package **`html`** there are two functions used for sanitization:

**`EscapeString()`** for escaping HTML text and

Note that this function only escapes the five characters: **`<`**, **`>`**, **`&`**, **`'`**, and **`"`**

**`UnescapeString()`** for unescaping HTML

# Sanitization

Remove line breaks, tabs and extra white space

The **`text/template`** and the **`html/template`** include a way to remove whitespaces from the template, by using a minus sign action's delimiter

Executing the template with source `{{- 23}} < {{45 -}}` will lead to the following output `23<45`

**NOTE**: If the minus `-` sign is not placed immediately after the opening action delimiter `{{` , or before the closing action delimiter `}}` the minus sign `-` will be applied to the value

# Sanitization

## URL request path

The **`ServeMux`** multiplexer in the **`net/http`** package takes care of sanitizing the URL request path, redirecting any request containing `.` or `..` elements or repeated slashes to an equivalent, cleaner URL

# Sanitization

## Output Encoding

As Web Applications become more complex, the more data sources they usually have, for example: users, databases, thirty party services, etc

At some point in time **collected data is outputted to some media** (e.g. a web browser) which has a specific context

This is exactly when injections happen if you do not have a strong Output Encoding policy

# Sanitization

## XSS - Cross Site Scripting

XSS has been on OWASP Top 10 security risks since 2003 and it's still a common vulnerability

You become vulnerable to XSS **if you do not ensure that all user supplied input is properly escaped**, or you do not verify it to be safe via server-side input validation, **before including that input in the output page**

# Sanitization

## XSS - Cross Site Scripting

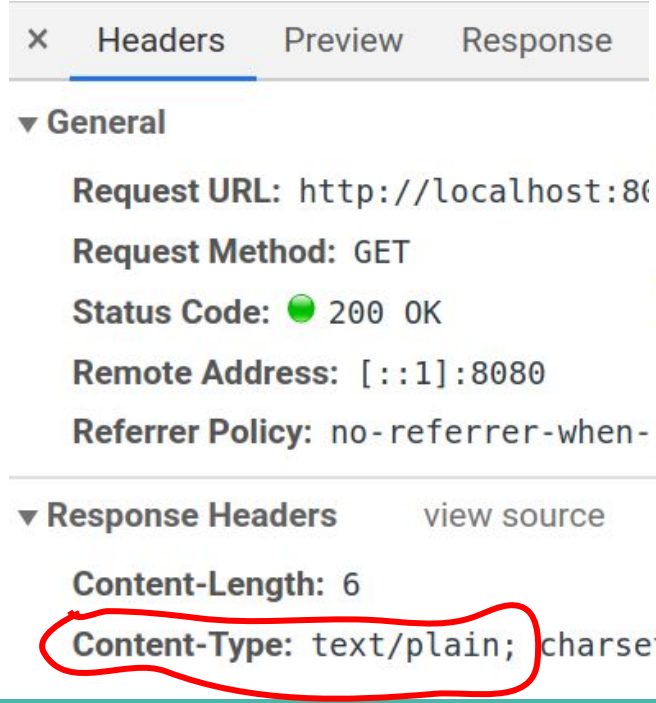
As **`Content-Type`** HTTP response header is not explicitly defined, Go **`http.DetectContentType`** default value will be used

```
package main

import (
    "io"
    "net/http"
)

func handler(w http.ResponseWriter, r *http.Request) {
    io.WriteString(w, r.URL.Query().Get("param1"))
}

func main() {
    http.HandleFunc("/", handler)
    http.ListenAndServe(":8080", nil)
}
```

So, making **`param1`** equal to "test", will result in **`Content-Type`** HTTP response header to be sent as **`text/plain`**

# Sanitization

## XSS - Cross Site Scripting



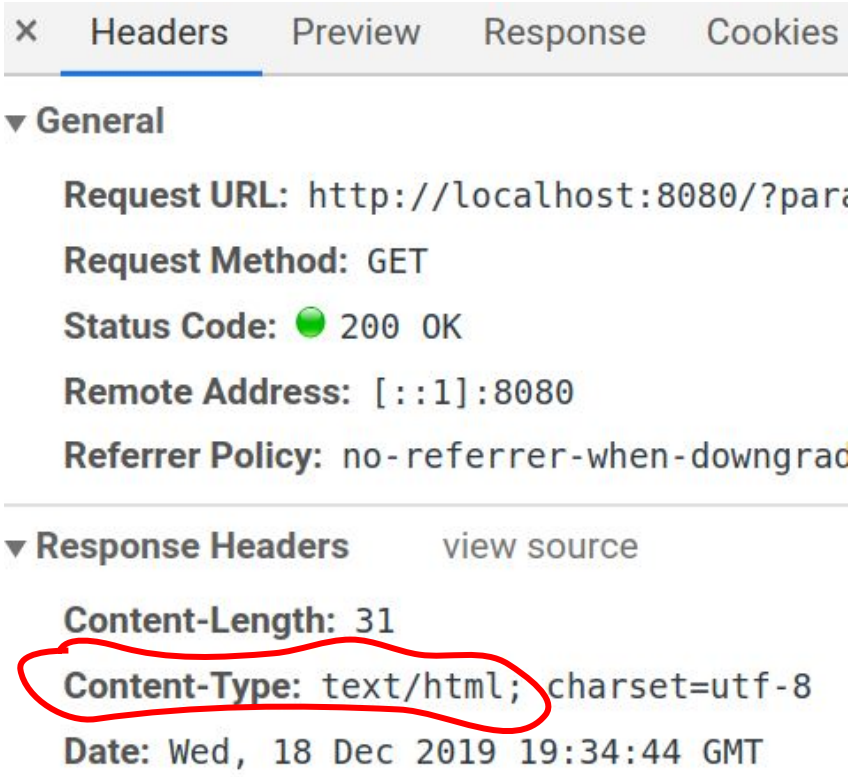

```go
package main

import (
    "io"
    "net/http"
)

func handler(w http.ResponseWriter, r *http.Request) {
    io.WriteString(w, r.URL.Query().Get("param1"))
}

func main() {
    http.HandleFunc("/", handler)
    http.ListenAndServe(":8080", nil)
}
```

# Sanitization

## XSS - Cross Site Scripting

If **`param1`** first characters contains some HTML tag such as **`<h1>`**, **`Content-Type`** will be **`text/html`**

If you put **`<script>alert("hello")</script>`** to **`param1`**, it will be executed

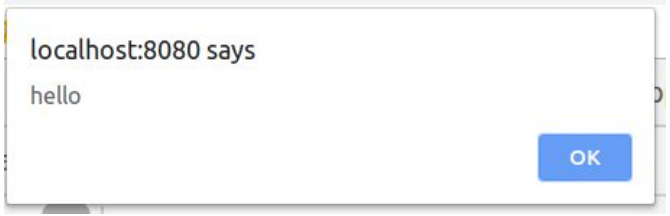



# Sanitization

## XSS - Cross Site Scripting

**`text/template`** package is also susceptible to XSS attack

If you put **`<script>alert("hello)</script>`** to **`param1`**, it will be executed




```go
package main

import (
    "net/http"
    "text/template"
)

func handler(w http.ResponseWriter, r *http.Request) {
    param1 := r.URL.Query().Get("param1")
    tmpl := template.New("hello")
    tmpl, _ = tmpl.Parse(`{{define "T"}}{{.}}{{end}}`)
    tmpl.ExecuteTemplate(w, "T", param1)
}
func main() {
    http.HandleFunc("/", handler)
    http.ListenAndServe(":8080", nil)
}
```

# Sanitization

## XSS - Cross Site Scripting

Escape special characters and use **`html/template`** to be safe from XSS attack

The escaped output after using **`html/template`**

```
package main

import (
    "html/template"
    "net/http"
)

func handler(w http.ResponseWriter, r *http.Request) {
    param1 := r.URL.Query().Get("param1")
    tmpl := template.New("hello")
    tmpl, _ = tmpl.Parse(`{{define "T"}}{{.}}{{end}}`)
    tmpl.ExecuteTemplate(w, "T", param1)
}
func main() {
    http.HandleFunc("/", handler)
    http.ListenAndServe(":8080", nil)
}
```

```
&lt;script&gt;alert(&#34;hello&#34;)
&lt;/script&gt;
```

# Sanitization

## SQL Injection

Another **common injection that's due to the lack of proper output encoding** is SQL Injection

This is mostly due to an old bad practice: string concatenation

Whenever a variable holding a value which may include arbitrary characters such as ones with special meaning to the database management system is simply added to a (partial) SQL query, you're vulnerable to SQL Injection

# Sanitization

## SQL Injection

Suppose you have the following Go code in your program

```
ctx := context.Background()
id := r.URL.Query().Get("id")
query := "SELECT * FROM grades WHERE studentID = " + id
row, _ := db.QueryContext(ctx, query)
```

When provided a valid student id value to the query parameter (id) that student’s grade(s) will be shown

# Sanitization

## SQL Injection

Suppose you have the following Go code in your program

```
ctx := context.Background()
id := r.URL.Query().Get("id")
query := "SELECT * FROM grades WHERE studentID = " + id
row, _ := db.QueryContext(ctx, query)
```

However, if the following value is given to the query parameter (id), the meaning of the query changes `1 OR 1=1;--`

```
SELECT * FROM grades WHERE studentID = 1 OR 1 = 1;
```

# Sanitization

## SQL Injection

Prepared statement allows to avoid this attack by separating the the code and the data part of the query

```
ctx := context.Background()
id := r.URL.Query().Get("id")
query := "SELECT * FROM grades WHERE studentID = $1"
row, _ := db.QueryContext(ctx, query, id)
```

**What makes this code safe from SQL Injection attack?**

# Authentication and Password Management

Includes security issues related to **user signup**, **credentials storage**, **password reset** and **private resources access**

All authentication controls must be enforced on a trusted system which usually is the server where the application's backend is running

Resources which require authentication should not perform it themselves. Instead, "redirection to and from the centralized authentication control" should be used

# Authentication and Password Management

Be careful handling redirection: you should redirect only to local and/or safe resources

Authentication should not be used only by the application's users, but also by your own application when it requires "connection to external systems that involve sensitive information or functions"

In these cases, authentication credentials for accessing services external to the application should be encrypted and stored in a protected location on a trusted system (e.g., the server)

The source code is NOT a secure location

# Authentication and Password Management

## Communicating authentication data

"communication" is used in a broader sense, encompassing **User Experience (UX) and client-server communication**

Not only is it true that password entry should be obscured on user's screen, but also the remember me functionality should be disabled

You can accomplish both by using an input field with **`type="password"`** and setting the **`autocomplete`** attribute to **`off`**

```
<input type="password" name="passwd" autocomplete="off" />
```

# Authentication and Password Management

## Communicating authentication data

**Authentication credentials should be sent only through encrypted connections (HTTPS)**. An exception to the encrypted connection may be the temporary passwords associated with email resets

Remember that requested URLs are usually logged by the HTTP server (access log). To prevent authentication credentials leakage to HTTP server logs, data should be sent to the server using the HTTP `POST` method

```
    [27/Feb/2017:01:55:09 +0000] "GET
/?username=user&password=70pS3c
```

# Authentication and Password Management

## Communicating authentication data

A well-designed HTML form for authentication would look like:

```
<form method="post" action="https://somedomain.com/user/signin" autocomplete="off">
    <input type="hidden" name="csrf" value="CSRF-TOKEN" />

    <label>Username <input type="text" name="username" /></label>
    <label>Password <input type="password" name="password" /></label>

    <input type="submit" value="Submit" />
</form>
```

# Authentication and Password Management

## Communicating authentication data

When handling authentication errors, **your application should not disclose which part of the authentication data was incorrect**

Instead of "Invalid username" or "Invalid password", just use **"Invalid username and/or password"**

**What are the advantages of doing this? Or the risks of doing the other**

# Authentication and Password Management

## Communicating authentication data

Using a generic message you do not disclose:

**Who is registered**: "Invalid password" means that the username exists

**How your system works**: "Invalid password" may reveal how your application works, first querying the database for the username and then comparing passwords in-memory

# Authentication and Password Management

## Communicating authentication data

After a successful login, the **user should be informed about the last successful or unsuccessful access date/time** so that he can detect and report suspicious activity

Additionally, it is also recommended to use a constant time comparison function while checking passwords in order to prevent a timing attack

# Authentication and Password Management

## Storing authentication data

More often than desirable, user account databases are leaked on the Internet

In the case of such an event, **collateral damages can be avoided if authentication data, especially passwords, are stored properly**

**All authentication controls should fail securely**

For sequential authentication implementations (like Google does nowadays), validation should happen only on the completion of all data input, on a trusted system

# Authentication and Password Management

## Storing password securely: the theory

You really don't need to store passwords since they are provided by the users (plaintext)

So, for security reasons, what you need is a **"one way" function** `H`, so that for every password `p1` and `p2` , `p1` is different from `p2`, `H(p1)` is also different from `H(p2)`

`H` should be such a function that there's no function $H^{-1}$ so that $H^{-1}(H(p1))$ is equal to `p1`

# Authentication and Password Management

## Storing password securely: the theory

if two different users provide the same password `p1`, we should store different hashed value. `salt` can be used to achieve this requirement

`salt`: a pseudo-random unique per user password value which is prepended to `p1` so that the resulting hash is computed as follows: `H(salt + p1)`

So each entry on a passwords store should keep the resulting hash, and the `salt` itself in plaintext: `salt` is not required to remain private

Avoid using deprecated hashing algorithms (e.g. `SHA-1`, `MD5`, etc)

# Authentication and Password Management

## Storing password securely: the practice

One of the most important sayings in cryptography is: **never roll your own crypto**

In the case of password storage, the hashing algorithms recommended by **OWASP** are **`bcrypt`**, **`PDKDF2`** , **`Argon2`** and **`scrypt`**

# Authentication and Password Management

## Storing password securely: `bcrypt` example

```
func main() {
    ctx := context.Background()
    email := []byte("john.doe@somedomain.com")
    pwd := []byte("47;u5:B(95m72;Xq")
    // Hash the password with bcrypt
    hashedPwd, err := bcrypt.GenerateFromPassword(pwd, bcrypt.DefaultCost)
    if err != nil {…
    }
    stmt, err := db.PrepareContext(ctx, "INSERT INTO accounts SET hash=?, email=?")
    if err != nil {…
    }
    result, err := stmt.ExecContext(ctx, hashedPwd, email)
    if err != nil {…
    }
}
```

# Authentication and Password Management

## Storing password securely: `bcrypt` example

Bcrypt also provides a simple and secure way to compare a plaintext password with an already hashed password

```
bcrypt.CompareHashAndPassword(hashedPassword,
[]byte(expectedPassword))
```

# Authentication and Password Management

## Password Policies

If your application sign-in requires a password, **enforce password complexity requirements**, requiring the use of alphabetic as well as numeric and/or special characters)"

**Password length should also be enforced**: eight characters is commonly used, but 16 is better or consider the use of multi-word pass phrases

To prevent users from reusing the same password, **enforce password changes**, and **prevent password reuse**

# Authentication and Password Management

## Password Reset

**Password reset mechanism is as critical as signup or sign-in**, and you're encouraged to follow the best practices to be sure your system does not disclose sensitive data and become compromised

Passwords should be at least one day old before they can be changed. This way you'll prevent attacks on password re-use

**Whenever using email based resets, only send email to a pre-registered address with a temporary link/password**, which should have a short expiration period

# Authentication and Password Management

## Password Reset

**Whenever a password reset is requested, the user should be notified**

**Temporary passwords should be changed on the next usage**

A common practice for password reset is the **"Security Question",** whose answer was previously configured by the account owner.

**Password reset questions should support sufficiently random answers**: asking for "Favorite Book?" may lead to "The Bible" which can easily be guessed

# Authentication and Password Management

## Other authentication guidelines

**Re-authenticate** users prior to performing critical operations

Use **Multi-Factor Authentication** for highly sensitive or high value transactional accounts

**Implement monitoring to identify attacks against multiple user accounts**, utilizing the same password. This attack pattern is used to bypass standard lockouts, when user IDs can be harvested or guessed

# Authentication and Password Management

## Other authentication guidelines

**Change all vendor-supplied default passwords** and user IDs or disable/remove the associated accounts

**Enforce account disabling after an established number of invalid login attempts** (e.g., five attempts is common). The account must be disabled for a period of time sufficient to discourage brute force guessing of credentials, but not so long as to allow for a denial-of-service attack to be performed

# Session Management

The flow of the session process can be seen in the following image


When dealing with session management, the application should only recognize the server's session management controls, and the session's creation should be done on a trusted system

# Session Management

Example using **JWT**

```
// create a JWT and put in the clients cookie
func setToken(res http.ResponseWriter, req *http.Request) {
    //30m Expiration for non-sensitive applications - OWASP
    expireToken := time.Now().Add(time.Minute * 30).Unix()
    expireCookie := time.Now().Add(time.Minute * 30)

    //token Claims
    claims := Claims{
        {...}
    }

    token := jwt.NewWithClaims(jwt.SigningMethodHS256, claims)
    signedToken, _ := token.SignedString([]byte("secret"))
```

# Session Management

Example using **`JWT`**

We must **ensure that the algorithms used to generate our session identifier are sufficiently random**, to prevent session brute forcing

```
...
token := jwt.NewWithClaims(jwt.SigningMethodHS256, claims)
signedToken, _ := token.SignedString([]byte("secret")) //our secret
...
```

# Session Management

Example using **`JWT`**

Now that we have a sufficiently strong token, we must also set the **`Domain`**, **`Path`**, **`Expires`**, **`HTTP only`**, **`Secure`** for our cookies

The **`Expires`** value is in this example set to 30 minutes since we are considering our application a low-risk application

```
// Our cookie parameter
cookie := http.Cookie{
    Name: "Auth",
    Value: signedToken,
    Expires: expireCookie,
    HttpOnly: true,
    Path: "/",
    Domain: "127.0.0.1",
    Secure: true
}

http.SetCookie(res, &cookie)
```

# Session Management

**Upon sign-in, a new session is always generated. The old session is never re-used, even if it is not expired**

We also use the `Expire` parameter to enforce periodic session termination as a way to **prevent session hijacking**

Another **important aspect of cookies is to disallow a concurrent login for the same username**. This can be done by keeping a list of logged in users, and comparing the new login username against this list. This list of active users is usually kept in a Database

# Session Management

Session identifiers should never be exposed in URL's

They should only be located in the HTTP cookie header

Session data must also be protected from unauthorized access by other users of the server

# Session Management

Regarding HTTP to HTTPS connection changes, special care should be taken to prevent Man-in-the-Middle (MITM) attacks that sniff and potentially hijack the user's session

The best practice regarding this issue, is to **use HTTPS in all requests**. In the following example the server is using HTTPS

```
err := http.ListenAndServeTLS(":443", "cert/cert.pem", "cert/key.pem", nil)
if err != nil {
  log.Fatal("ListenAndServe: ", err)
}
```

# Session Management

In case of highly sensitive or critical operations, the token should be generated per-request, instead of per-session

Always make sure the token is sufficiently random and has a length secure enough to protect against brute forcing

# Session Management

Applications should provide a way to logout from all pages that require authentication, as well as fully terminate the associated session and connection

When a user logs out, the cookie should be deleted from the client. The same action should be taken in the location where the user session information is stored

# Access Control

After generating a session token on the server-side, we can then store and **use the token to validate the user and to enforce Access Control model**

# Access Control

The component used for access authorization should be a single one, used site-wide

**In case of a failure, access control should fail securely**. In Go you can use defer to achieve this

If the application cannot access its configuration information, all access to the application should be denied

# Access Control

**Authorization controls should be enforced on every request**, including server-side scripts, as well as requests from client-side technologies like AJAX or Flash

It is also important to properly separate privileged logic from the rest of the application code

# Access Control

Other important operations where access controls must be enforced are the following:

> File and other resources, Protected URL's, Protected functions, Direct object references, Services Application data, User and data attributes and policy information

When implementing these access controls, it's important to verify that the server-side implementation and the presentation layer representations of access control rules are the same

# Access Control

**If state data needs to be stored on the client-side, it's necessary to use encryption and integrity checking in order to prevent tampering**

Application logic flow must comply with the business rules

When dealing with transactions, **the number of transactions a single user** or device can perform in a given period of time must be **above the business requirements but low enough to prevent a user from performing a Denial-of-Service (DoS) attack**

# Access Control

It is important to note that using only the `referer` HTTP header is insufficient to validate authorization, and should only be used as a supplemental check

**Regarding long authenticated sessions, the application should periodically re-evaluate the user's authorization to verify that the user's permissions have not changed**. If the permissions have changed, log the user "out" and force them to re-authenticate

# Access Control

User accounts should also have a way to audit them, in order to comply with safety procedures. (e.g. Disabling a user's account 30 days after the password expiration date)

**Applications must also support the disabling of accounts and the termination of sessions when a user's authorization is revoked**. (e.g. role change, employment status, etc.)

When supporting external service accounts and accounts that support connections from or to external systems, these accounts must use the lowest level privilege possible

# Error Handling

In Go, not only you can use the built-in errors, you can also specify your own error types. This can be achieved by using the **`errors.New`** function.

```
{...}
if f < 0 {
    return 0, errors.New("math: square root of negative number")
}
//If an error has occurred print it
if err != nil {
    fmt.Println(err)
}
{...}
```

# Error Handling

If you need to format the string containing the invalid argument to see what caused the error, the **`Errorf`** function in the **`fmt`** package allows us to do this.

```
{...}
if f < 0 {
    return 0, fmt.Errorf("math: square root of negative number %g", f)
}
{...}
```

# Error Handling

**When dealing with error logs, developers should ensure that no sensitive information is disclosed in the error responses**, as well as guarantee that no error handlers leak information (e.g. debugging, or stack trace information)

# Error Handling

In Go, there are additional error handling functions, these functions are **`panic`**, **`recover`**, and **`defer`**

When an application state is **`panic`** its normal execution is interrupted, any **`defer`** statements are executed, and then the function returns to its caller

**`recover`** is usually used inside **`defer`** statements and allows the application to regain control over a panicking routine, and return to normal execution

# Error Handling

In the `log` package there is a `log.Fatal`. Fatal level is effectively logging the message, then calling `os.Exit(1):` Which means:

- Defer statements will not be executed
- Buffers will not be flushed
- Temporary files and directories are not removed

For this reason it should be used carefully

# Error Handling

Some examples of the possible usage of `log.Fatal` are

Set up logging and check whether you have a healthy environment and parameters. If not, then there's no need to proceed

An error that should never occur and that you know that it's unrecoverable

If a non-interactive process encounters an error and cannot complete, there is no way to notify the user about this error. It's best to stop the execution before additional problems can emerge from this failure

# Logging

Logging should always be handled by the application and should not rely on a server configuration

All logging should be implemented by a master routine on a trusted system, and the developers should also ensure no sensitive data is included in the logs (e.g. passwords, session information, system details, etc.), nor is there any debugging or stack trace information

**Logging should cover both successful and unsuccessful security events, with an emphasis on important log event data**

**List some user/application activities that your application should log**

# Logging

Important log event data most commonly used are:

- Input validation failures
- Authentication attempts, especially failures
- Access control failures
- Apparent tampering events, including unexpected changes to state data
- Attempts to connect with invalid or expired session tokens

# Logging

Important log event data most commonly used are:

- System exceptions
- Administrative functions, including changes to security configuration settings
- Backend TLS connection failures and cryptographic module failures

# Logging

From the perspective of log access, **only authorized individuals should have access to the logs**

Developers should also make sure that a mechanism that allows for log analysis is set in place, as well as guarantee that no untrusted data will be executed as code in the intended log viewing software or interface

To guarantee log validity and integrity, **a cryptographic hash function should be used as an additional step to ensure no log tampering has taken place**

# Logging

Checking log temparing

```
{...}
// Get our known Log checksum from checksum file.
logChecksum, err := ioutil.ReadFile("log/checksum")
str := string(logChecksum) // convert content to a 'string'

// Compute our current log's SHA256 hash
b, err := ComputeSHA256("log/log")
if err != nil {
  fmt.Printf("Err: %v", err)
} else {
  hash := hex.EncodeToString(b)
  // Compare our calculated hash with our stored hash
  if str == hash {
    // Ok the checksums match.
    fmt.Println("Log integrity OK.")
  } else {
    // The file integrity has been compromised...
    fmt.Println("File Tampering detected.")
  }
}
{   }
```

# Logging

It's important to note that the log-file hashes must be stored in a safe place, and compared with the current log hash to verify integrity before any updates to the log

# Data Protection

## Remove sensitive information

Temporary and cache files which contain sensitive information should be removed as soon as they're not needed. If you still need some of them, move them to protected areas or encrypt them.

## Comments

Sometimes developers leave comments like To-do lists in the source- code, and sometimes, in the worst case scenario, developers may leave credentials

```
// Secret API endpoint - /api/mytoken?callback=myToken
fmt.Println("Just a random code")
```

# Data Protection

## URL

Passing sensitive information using the HTTP GET method leaves the web application vulnerable

Note that parameters being passed through GET (aka query string) will be stored in clear, in the browser history and the server's access log regardless whether you're using HTTP or HTTPS

# Data Protection

## URL

HTTPS is the way to go to prevent external parties other than the client and the server, to capture exchanged data

Whenever possible **sensitive data such as an `api_key`, should go in the request body or some header**. The same way, whenever possible use one-time only session IDs or tokens

# Data Protection

## Information is power

You should always **remove application and system documentation on the production environment**

Some documents could disclose versions, or even functions that could be used to attack your web application (e.g. Readme, Changelog, etc.)

# Data Protection

## Information is power

As a developer, you should allow the user to remove sensitive information that is no longer used

For example, if the user has expired credit cards on his account and wants to remove them, your web application should allow it

All of the information that is no longer needed must be deleted from the application

# Data Protection

## Encryption is the key

Every piece of **highly-sensitive information should be encrypted** in your web application

Use the military-grade encryption available in Go

**Do not store passwords, connection strings or other sensitive information in clear text** or in any non-cryptographically secure manner, both on the client and server sides

**What is the difference between hashing and encryption?**

# Data Protection

## Disable Cache

Cache control in pages that contain sensitive information should be disabled

This can be achieved by setting the corresponding header flags, as shown in the following snippet:

```
w.Header().Set("Cache-Control", "no-cache, no-store")
w.Header().Set("Pragma", "no-cache")
```

# Communication Security

Types of communication include **server-client**, **server-database**, as well as all backend communications

These must be encrypted to guarantee data integrity, and to protect against common attacks related to communication security

**Failure to secure these channels allows** known attacks like **MITM**, which **allows attacker to intercept and read the traffic in these channels**

# Communication Security

## HTTP/TLS

TLS/SSL is a cryptographic protocol that allows encryption over otherwise insecure communication channels. The most common usage of **TLS/SSL** is to **provide secure HTTP communication**, also known as HTTPS

The protocol ensures **Privacy**, **Authentication**, **Data integrity**

Its implementation in **Go** is in the `crypto/tls` package

# Communication Security

## HTTP/TLS

Example in Go

```
...
type Certificates struct {
    CertFile    string
    KeyFile     string
}

func main() {
    httpsServer := &http.Server{
        Addr: ":8080",
    }

    var certs []Certificates
    certs = append(certs, Certificates{
        CertFile: "../etc/yourSite.pem", //Your site certificate key
        KeyFile:  "../etc/yourSite.key", //Your site private key
    })
```

# Communication Security

## HTTP/TLS

Example in Go

```
    config := &tls.Config{}
    var err error
    config.Certificates = make([]tls.Certificate, len(certs))
    for i, v := range certs {
        config.Certificates[i], err = tls.LoadX509KeyPair(v.CertFile, v.KeyFile)
    }

    conn, err := net.Listen("tcp", ":8080")

    tlsListener := tls.NewListener(conn, config)
    httpsServer.Serve(tlsListener)
    fmt.Println("Listening on port 8080...")
}
```

# Communication Security

## HTTP/TLS

It should be noted that **when using TLS, the certificates should be valid, have the correct domain name, should not be expired**, and should be installed with intermediate certificates when required

Important: Invalid TLS certificates should always be rejected. Make sure that the **`InsecureSkipVerify`** configuration is not set to true in a production environment

```
config := &tls.Config{InsecureSkipVerify: false}
```

# System Configuration

**Keeping things updated is imperative in security**

With that in mind, developers should keep Go updated to the latest version, as well as external packages and frameworks used by the web application

# System Configuration

## Directory Listing

If a developer forgets to disable directory listings, an attacker could check for sensitive files navigating through directories

In Go, you should also be careful with the following code

```
http.ListenAndServe(":8080", http.FileServer(http.Dir("/tmp/static")))
```

# System Configuration

## Directory Listing

To fix directory listing, you have three possible solutions:

- Disable directory listings in your web application
- Restrict access to unnecessary directories and files
- Create an index file for each directory

# System Configuration

Example to prevent directory listing

```
type justFilesFilesystem struct {
    fs http.FileSystem
}

func (fs justFilesFilesystem) Open(name string) (http.File, error) {
    f, err := fs.fs.Open(name)
    if err != nil {
        return nil, err
    }
    return neuteredReaddirFile{f}, nil
}
```

```
fs := justFilesFilesystem{http.Dir("tmp/static/")}
http.ListenAndServe(":8080", http.StripPrefix("/tmp/static", http.FileServer(fs)))
```

# System Configuration

## Remove/Disable what you don't need

On production environments, remove all functionalities and files that you don't need. Any test code and functions not needed on the final version (ready to go to production), should stay on the developer layer, and not in a location everyone can see - aka public

HTTP Response Headers should also be checked. **Remove the headers which disclose sensitive information** like: OS version, Web Server version, Framework or Programming Language version

# Database Security

## Secure database server installation

- Change/set a password for root account(s)
- Remove the `root` accounts that are accessible from outside the localhost
- Remove any anonymous-user accounts
- Remove any existing test database

Remove any unnecessary stored procedures, utility packages, unnecessary services, vendor content (e.g. sample schemas)

# Database Security

Install the minimum set of features and options required for your database to work with Go

Disable any default accounts that are not required on your web application to connect to the database

# Database Security

## Database Connections

In Go the context variant of `database/sql` interface (e.g. `QueryContext()` ) should always be used and provided with the appropriate Context

Package context defines the Context type, which carries deadlines, cancelation signals, and other request-scoped values across API boundaries and between processes

# Database Security

## Database Connections

At a database level, when the context is canceled, a transaction will be rolled back if not committed, a Rows (from QueryContext) will be closed and any resources will be returned

# Database Security

## Connection string protection

To keep your connection strings secure, **it's always a good practice to put the authentication details on a separated configuration file**, outside of public access

Instead of placing your configuration file at `/home/public_html/` consider `/home/private/configDB.xml` a protected area

# Database Security

## Connection string protection

Instead of placing your configuration file at **`/home/public_html/`** consider **`/home/private/configDB.xml`** a protected area

Then you can call the **`configDB.xml`** file for example on your Go file:

```
configFile, _ := os.Open("../private/configDB.xml")
```

```
<connectionDB>
  <serverDB>localhost</serverDB>
  <userDB>f00</userDB>
  <passDB>f00?bar#ItsP0ssible</passDB>
</connectionDB>
```

After reading the file, make the database connection:

```
db, _ := sql.Open(serverDB, userDB, passDB)
```

# Database Security

## Connection string protection

Of course, if the attacker has root access, he will be able to see the file. Which brings us to the most cautious thing you can do - **encrypt the file**

# Database Security

## Database Credentials

You should **use different credentials for every trust distinction and level,** for example:

User, Read-only user, Guest, Admin

That way if a connection is being made for a read-only user, they could never mess up with your database information because the user actually can only read the data.

# Database Security

## Database Authentication

If your Go web application only needs to read data and doesn't need to write information, create a database user whose permissions are read-only

**Always adjust the database user according to your web applications needs**

When creating your database access, choose a strong password. You can use password managers to generate a strong password

Most DBS have default accounts and most of them have no passwords on their highest privilege user. So, set passwords for default accounts

# Database Security

## Parameterized Queries

Prepared Statements (with Parameterized Queries) are the best and most secure way to protect against SQL Injections

```
customerName := r.URL.Query().Get("name")
db.Exec("UPDATE creditcards SET name=? WHERE customerId=?", customerName, 233, 90)
```

# Database Security

## Stored Procedures

Developers can use Stored Procedures to create specific views on queries to prevent sensitive information from being archived, rather than using normal queries

```
SELECT * FROM tblUsers WHERE userId = $user_input
```

```
CREATE PROCEDURE db.getName @userId int = NULL
AS
    SELECT name, lastname FROM tblUsers WHERE userId = @userId
GO
```

# File Management

**File uploads should only be permitted from authenticated users**

Another important aspect of security is to **make sure that only acceptable file types can be uploaded to the server** (whitelisting )

This check can be made using the following Go function that detects MIME types `func DetectContentType(data []byte) string`

# File Management

Detecting file type example

```
{...}
// Write our file to a buffer
// Why 512 bytes? See http://golang.org/pkg
buff := make([]byte, 512)

_, err = file.Read(buff)
{...}
//Result - Our detected filetype
filetype := http.DetectContentType(buff)
```

```
{...}
switch filetype {
case "image/jpeg", "image/jpg":
  fmt.Println(filetype)
case "image/gif":
  fmt.Println(filetype)
case "image/png":
  fmt.Println(filetype)
default:
  fmt.Println("unknown file type uploaded")
}
{...}
```

# File Management

Files uploaded by users should not be stored in the web context of the application. Instead, they should be stored in a content server or in a database

An important note is for the selected file upload destination **no**t to have **execution privileges**

Never send the absolute file path to the user, always use relative paths

# Memory Management

Buffer boundary checking is important aspect of memory management

```
func main() {
    strings := []string{"aaa", "bbb", "ccc", "ddd"}
    // This loop is not checking the slice length -> BAD
    for i := 0; i < 5; i++ {
        if len(strings[i]) > 0 {
            fmt.Println(strings[i])
        }
    }
}
```

```
Output
aaa
bbb
ccc
ddd
panic: runtime error: index
out of range
```

**How can you improve the code to avoid the crash?**

# Memory Management

When our application uses **resources**, additional checks must also be made to ensure they have been **closed**, and not rely solely on the Garbage Collector

This is applicable when dealing with connection objects, file handles, etc. In Go we can use `defer` to perform these actions

# Memory Management

## Cross-Site Request Forgery

It is **an attack that forces an end user to execute unwanted actions on a web application** in which they're currently authenticated

CSRF attacks are not focused on data theft. Instead, they target state-changing requests

**With** a little **social engineering** (such as sharing a link via email or chat) the **attacker may trick users to execute unwanted web-application actions** such as changing account recovery email

# Memory Management

## Cross-Site Request Forgery (Attack scenario)

Let's say that `foo.com` uses HTTP GET requests to set the account's recovery email as shown

```
GET https://foo.com/account/recover?email=me@somehost.com
```

# Memory Management

## Cross-Site Request Forgery (Attack scenario)

A simple attack scenario may look like:

Victim is authenticated at `https://foo.com`

Attacker sends a chat message to the Victim with the following link:

```
https://foo.com/account/recover?email=me@attacker.com
```

Victim's account recovery email address is changed to `me@attacker.com`, giving the Attacker full control over it

# Memory Management

## Cross-Site Request Forgery (Solution)

CSRF targets state-changing requests. Concerning Web Applications, most of the time that means POST requests issued by form submission

When a user first requests the page which renders the form, the server computes a **nonce** (**an arbitrary number intended to be used once**)

**This token is then included into the form as a field** (most of the time this field is hidden but it is not mandatory)

# Memory Management

## Cross-Site Request Forgery (Solution)

```
<form method="post" action="https://somedomain.com/user/signin" autocomplete="off">
    <input type="hidden" name="csrf" value="CSRF-TOKEN" />

    <label>Username <input type="text" name="username" /></label>
    <label>Password <input type="password" name="password" /></label>

    <input type="submit" value="Submit" />
</form>
```

# Memory Management

## Cross-Site Request Forgery (Solution)

Next, **when the form is submitted, the hidden field is sent along with other user input**. The server should then validated whether the token is part the request data, and determine if it is valid

The specific nonce/token should obey to the following requirements:

> Unique per user session, Large random value, and Generated by a cryptographically-secure random number generator

# Reading Assignment

**OAuth 0.2**

https://oauth.net/2/

https://www.oauth.com/

**OpenID Connect**

https://auth0.com/docs/protocols/oidc

https://www.oauth.com/oauth2-servers/openid-connect/

**JWT**

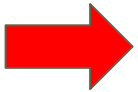
https://jwt.io/introduction/

https://www.oauth.com/oauth2-servers/access-tokens/self-encoded-access-tokens/

# Reference

https://github.com/OWASP/Go-SCP

https://astaxie.gitbooks.io/build-web-application-with-golang/content/en/09.0.html

**The are more slides** 

# Read the following security guidelines written in the form of dos and don’ts

(As most of the topics are already covered in the previous slides, they will serve you as a summary)

# Unvalidated Input

Information from web requests is not validated before being used by a web application

Attackers can use these flaws to attack users and backend components through a web application

This includes **Cross Site Scripting**, **Buffer Overflows** and **Injection Flaw** vulnerabilities

# Unvalidated Input

## Dos

Assume all input is malicious

Validate everything – inspect what is expected, and reject anything unexpected

Accept only “Known Good” characters

Ensure input is validated server-side

Validate parameters against a "positive" specification, limiting input permitted to characters appearing in a whitelist

# Unvalidated Input

## Dos

If possible, implement an exact match validator

Check input value range to make sure that the data lie within a specified range of values

Perform validation on every tier

Centralise the validation code

Use an input validation framework such as the OWASP ESAPI Validation API

# Unvalidated Input

## Dos

Repeatedly decode and normalise (i.e. canonicalization) until input == output

A generic error page for most errors is recommended when developing code

This approach makes it more difficult for attackers to identify signatures of potentially successful attacks

# Unvalidated Input

## Don’ts

Rely on client-side data validation

Trust anything a user or other process sends to the application

Rely on a blacklists

Validate data before it has been filtered

Accept null or empty input in non-optional input fields

Validate input before it is canonicalized

# Broken Access Control

Restrictions on what authenticated users are allowed to do are not properly enforced

Attackers can exploit these flaws to access other users' accounts, view sensitive files, or use unauthorised functions

# Broken Access Control

## Dos

Implement Principle of Least Privilege

Perform access control validation to check that a user is authorised to perform a task upon every user request

Think through the application's access control requirements and capture the web application security policy

Use an access control matrix to define the access control rules

Centralised access control routines

# Broken Access Control

## Dos

Use built-in platform or framework authorisation facilities

Clearly document the site's access control policy as part of the design documentation. The policy should document what types of users can access the system, and what functions and content each of these types of users should be allowed to access

Use multiple mechanisms, including HTTP headers and meta tags, to be sure that pages containing sensitive information are not cached by user's browsers

# Broken Access Control

## Don’ts

Rely on not displaying certain functions which the users do not have authorisation to use

Implement authorisation controls by including a file or web control or code snippet on every page in the application

Use custom or write your own authorisation code

Perform authorisation checks at or near the end of code implementing sensitive activities

# Broken Authentication and Session Management

Account credentials and session tokens are not properly protected

Attackers that can compromise passwords, keys, session cookies, or other tokens can defeat authentication restrictions and assume other users' identities

# Broken Authentication and Session Management

Accounts should be locked after a predetermined number of failed attempts

A value between 3 and 5 is typical and acceptable

Consider removing the account lockout after a set time window to limit the ability for a malicious user to create a Denial of Service scenario

This will also minimise administrator support to re-enable these accounts

Industry best practice is to lock accounts for increasing periods following subsequent failed login attacks (i.e. 3rd invalid attempt = 5 min lockout, 6th = 10 mins, 9th = 15 mins and so on).

# Broken Authentication & Session Management

## Dos

Enforce strong and complex passwords on users by using known and proven password generation methods/frameworks

Only store ‘salted’ passwords using a computationally expensive one-way hash algorithm (i.e. bcrypt or PBKDF2)

Limit session lifetime for both inactivity (e.g. 5 min or business/design requirement) and hard-limit

# Broken Authentication & Session Management

## Dos

Upon log-out, authorisation failure and session timeout destroy the session and overwrite the session cookie

Session identifier / cookie should not be predictable. Rely on a leading web frameworks for token generation

Check if the session is valid prior to servicing any user requests

Assign users a new session ID upon successful authentication

# Broken Authentication & Session Management

## Dos

Use secure channel (e.g. SSL) to transport cookies

If possible, conduct all traffic over HTTPS

Verify that the ‘secure’ directive/flag is set on the cookie so they are not served over non-SSL tunnels

# Broken Authentication & Session Management

## Don'ts

Store passwords in cleartext

Develop a custom session identifier

Pass session Identifiers using a HTTP GET with the session ID in the query string

Implement “remember me” functionality

Reuse session IDs for HTTPS transport that have once been used over HTTP

# Improper Error Handling

Error conditions that occur during normal operation are not handled properly

If an attacker can cause errors to occur that the web application does not handle, they can gain detailed system information, deny service, cause security mechanisms to fail, or crash the server

# Improper Error Handling

## Do

Catch every potential exception in the application code

Assure that the application fails safely under all possible error conditions

Use a generic error page for all exceptions containing no sensitive data

Where required by the user, display a custom error reference in the generic error page

# Improper Error Handling

## Dos

Document when exceptions occur

Consider expiring user's session and lock out the user where severe exceptions occur; and notify the administrator

Study and understand how the order of error handling events work in the chosen development language to understand the error strategy of the application

# Improper Error Handling

## Don’ts

Use default error pages

Display sensitive information/data such as stack trace, line number where the error occurred, class name, method name, paths on the local file system or any internal system information in the error messages

Include people’s names or any internal contact information in the error messages

# Information Leakage

Excessive or unnecessary information disclosure

# Information Leakage

## Dos

Transmit sensitive information via the body of a POST request

Proper Error Handling

Examine the data logged to determine if any sensitive information is being stored in the logs (e.g. userID, passwords)

Review and remove, where possible, redundant, readable and downloadable files on a web server, such as old, backup and renamed files

# Information Leakage

## Dos

Disable Autocomplete using AUTOCOMPLETE=OFF attribute for form fields containing sensitive information

Only store the session ID in the cookies

# Information Leakage

## Don'ts

Transfer sensitive data using the GET method

Disclose sensitive data in the source code comments that an attacker may gain access to

Include elements, such as technical or other sensitive information, within response data that could aid an attacker

# Information Leakage

## Don'ts

Return messages to a user that could aid a compromise. In particular, certain messages such as “User does not exist” allow an attacker to enumerate valid user names

Cache pages containing sensitive information on the local machine

Reveal information about directories within robots.txt files

# Application Denial of Service

Attackers can consume web application resources to a point where other legitimate users can no longer access or use the application

Attackers can also lock users out of their accounts or even cause the entire application to fail

# Application Denial of Service

## Dos

Limit the resources allocated to any user to a bare minimum

For authenticated users, establish quotas so that the amount of load a particular user can put on the system is limited

Split resource hungry tasks into smaller manageable tasks

Consider imposing a time limit between subsequent requests for resource hungry tasks

# Application Denial of Service

## Dos

Prevent attackers from being able to permanently lock-out accounts by making sure users can unlock locked accounts in a safe manner such as sending a user a password reset link to the users registered email address

Where file upload functionality is implemented, limit file upload sizes and number of allowed files

Load testing

# Application Denial of Service

## Dos

Store logs on a separate file system and transfer them to a separate machine if possible

Think about whether attacker controlled data ends up in a hash table, if so use data structures such as treemaps to avoid hash collision DoS unless the framework used protects against this

# Application Denial of Service

## Don’ts

Allow resource intensive functions where possible

Allow users to have unlimited resources and quotas allocated to them

Allow users to upload large files

Create an account lock-out mechanism that difficult for the user to unlock and requires administrative resources etc

Use non-randomised hash functions for attacker controlled data

# Default Install, Poor Configuration and Patch Management

Having a strong server configuration standard is critical to a secure web application

These servers have many configuration options that affect security and are not secure out of the box

# Default Install, Poor Configuration and Patch Management

## Dos

Define, document and implement the security configuration for the deployed application frameworks (e.g., Struts, Spring, ASP.NET), application server, web server, database server, and platform

Define a process to keep all software up to date including all code libraries used by the application

Restrict access to administrator interfaces to users that require access by for example IP address

# Default Install, Poor Configuration and Patch Management

**Dos**

Disable/change default passwords

Disable/remove anything unnecessary (e.g. ports, accounts, services, pages, privileges, frameworks, add-ons)

If SSL is used, there are potential security implications and best security practise should be followed to secure the service such as disabling SSL Service Support for "Weak" Cipher Suites

# Default Install, Poor Configuration and Patch Management

## Dos

Keep logs for substantial period of time to facilitate continuous monitoring and incident handling. Some incidents may be detected after weeks or months

Protect cookies with the HttpOnly attribute/flag from the risk of a client side script accessing it

# Default Install, Poor Configuration and Patch Management

**Don’ts**

Handover and leave the application without having defined and documented the security configuration and process for keeping all related software up to date

Leave the system in a default install state

Allow access to administration interfaces from untrusted networks

Rely on other people to configure SSL, should the application require it, in line with best practice

# Other

This is a generic catch all for vulnerabilities that do not fit elsewhere

- Unfinished code
- Sensitive information (passwords, passphrases) found to be stored within the source code
- Use of dangerous functions
- Use of weak encryption algorithms in the source code (e.g. MD5 and DES)

# Other

This is a generic catch all for vulnerabilities that do not fit elsewhere

- Web application includes files from a third party
- Application does not display last unsuccessful login
- Application can be embedded in third party websites using IFRAMEs
- No search engine protection e.g. use of robots.txt or Meta Tags

# Reference

OWASP Developer's Guide:
**https://www.owasp.org/index.php/OWASP_Guide_Project**
OWASP Code Review Guide:
**https://www.owasp.org/index.php/Category:OWASP_Code_Review_Project**
OWASP Secure coding practices quick reference:
**https://www.owasp.org/index.php/OWASP_Secure_Coding_Practices_-_Quick_Reference_Guide**
**https://www.owasp.org/images/b/ba/Web_Application_Development_Dos_and_Donts.ppt**