A Quick Intro to
Go Language Security Topics

Research by Ricardo Gonçalves

Are you new to the Go language (Golang), or are you an old schooler who just wants a single resource highlighting all the security advantages of Go? You've come to the right place! This summary is a compilation of the best practices that you need to follow if you want to create secure applications in Go. It's a summary of the extended work found in the Go Language Guide – Web Application Secure Coding Practices. This Guide is the first of its kind for the Go Language and Checkmarx is pleased to have supported and helped direct this effort for the software developer and AppSec community.

A frequent difficulty experienced when you're starting to use a new programming language is the lack of secure coding instruction and training about common pitfalls and coding errors during the language-learning process. The topic of security is often neglected by most articles pertaining to a language or security discussions are very dispersed around the thousands of blog posts. Also, most of the time you want or need to finish things in a timely fashion, and as we all know, security and time do not always go hand in hand!

Therefore, to help you quickly increase your knowledge of Go security in general, we've compiled this short summary of security topics you should be aware of when using Go. If you're ready to learn more... Let's Go!

1. Data Integrity – Validate and Sanitize

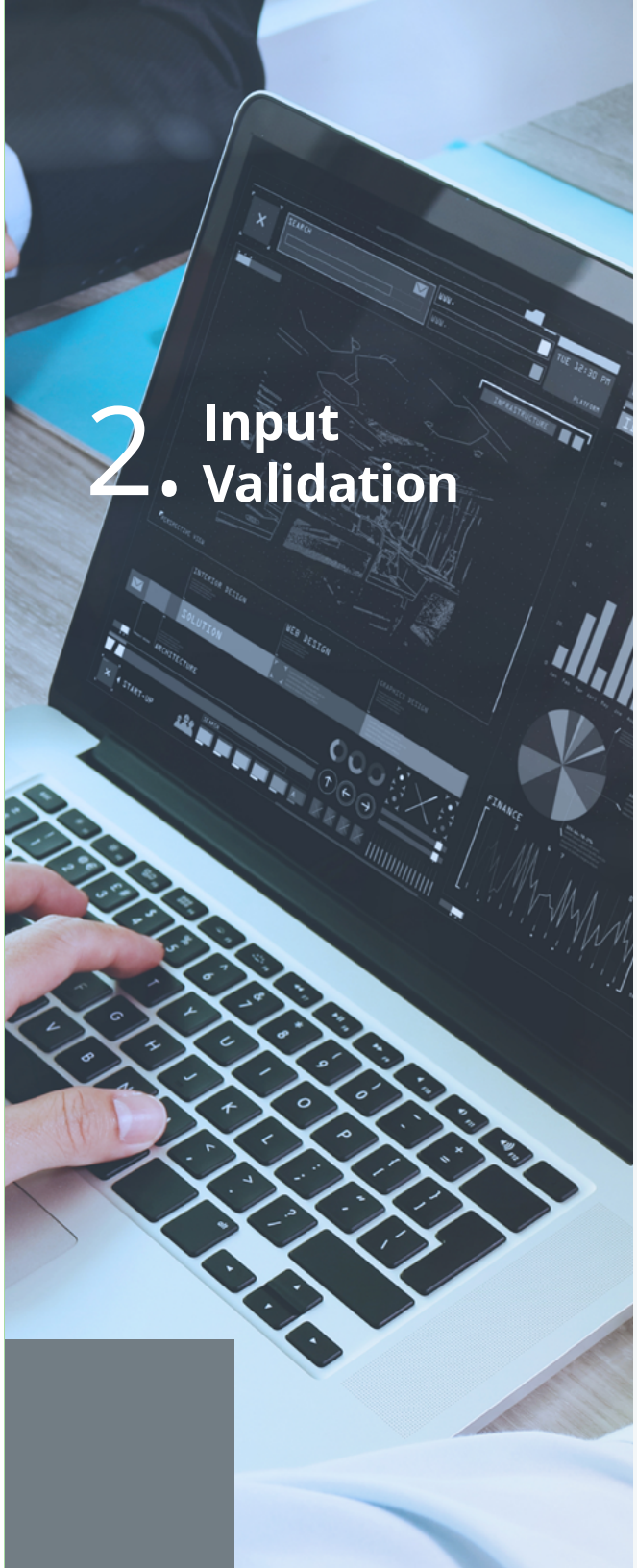
One thing you should consider during the development process is the amount of third parties that your application relies on. In such cases, user input should always be validated and sanitized prior to other actions that interact with the data. If you just rely on the third parties to perform this function, you might be leaving your application and clients' data at risk. The lack of validation and sanitization could expose your application to Cross-site Scripting (XSS), SQL Injection (SQLi), and other attacks.

Therefore, perform some housekeeping first by verifying the integrity of the data by using input validation, and try to use open source libraries that have active development and high ratings, which might also be a double-edge sword... as new exploits targeting your open source choices can affect your application, so always be sure you're up-to-date!

This point is important because Go has some security features out of the box. For example, Cross-site Scripting can be addressed with automatic context detection using the standard library HTML templates, and SQL Injections can be address with the database/sql package, which provides a well-defined interface, but needs to be used in conjunction with a database driver.

Apart from these two security problems that can be addressed by filtering the input and escaping output, there are more challenges that you need to be aware of. In the following sections, we'll point out the security preventions and configurations that you can use in the Go language to address the following security aspects:

- **Regular Expressions Denial of Service**
- **CSRF Attacks**
- **Security Headers**
- **Sessions and Cookies**
- **Communication Security**
- **Secure Data Storage**
- **Memory Management and Concurrency**
- **System Configuration**



2. Input Validation

When you're building an application, it's never a good idea to trust a user's input data. Therefore, you should always perform *input validation and sanitization,* for example, checking that only properly formed data is entering your application, and modifying the input to ensure that it is valid.

In Go, you can use some native libraries to protect your application from malicious input data. These native libraries, together with the common best practices for input validation [1], can spare you from future headaches. Once you know the type of data that is expected for a specific input field, you can validate it with following native packages:

- **`strconv:`** package to convert strings input by users into specific types
- **`strings:`** package that contains all functions that handle strings and its properties
- **`regexp:`** package for regular expressions to accommodate custom formats
- **`utf8:`** package that implements functions and constants to support text encoded in UTF-8

Apart from the native packages, there are some third-party ones that can help you in troubled times:

- **`gorilla:`** is one of the most used packages for web application security. It has support for WebSockets, cookie sessions, RPC, among others
- **`govalidator:`** is a package of validators and sanitizers for strings, structs, and collections that's based on validator.js
- **`Validator:`** is a package validator that implements value validations for structs and individual fields based on tags

The previous references focused on input validation, but in addition to that, another technique that can be used is input sanitization. It collects the inputted values and removes or replaces some of the data. This is often used as an extra layer of security after input validation. In Go, the native package `html` is composed of two functions that can help in input sanitization:

- `EscapeString:` accepts a string and returns the same string with special characters escaped (it only escape few characters)
- `UnescapeString:` function to convert from entities to characters

Since these packages do not address all of the characters, there are some third-party packages that you can rely on as well:

- `bluemonday:` takes untrusted user-generated content as an input, and will return HTML that has been sanitized against a whitelist of approved HTML elements and attributes so that you can safely include the content in your web page
- `sanitize:` this package provides functions for sanitizing text in Golang strings

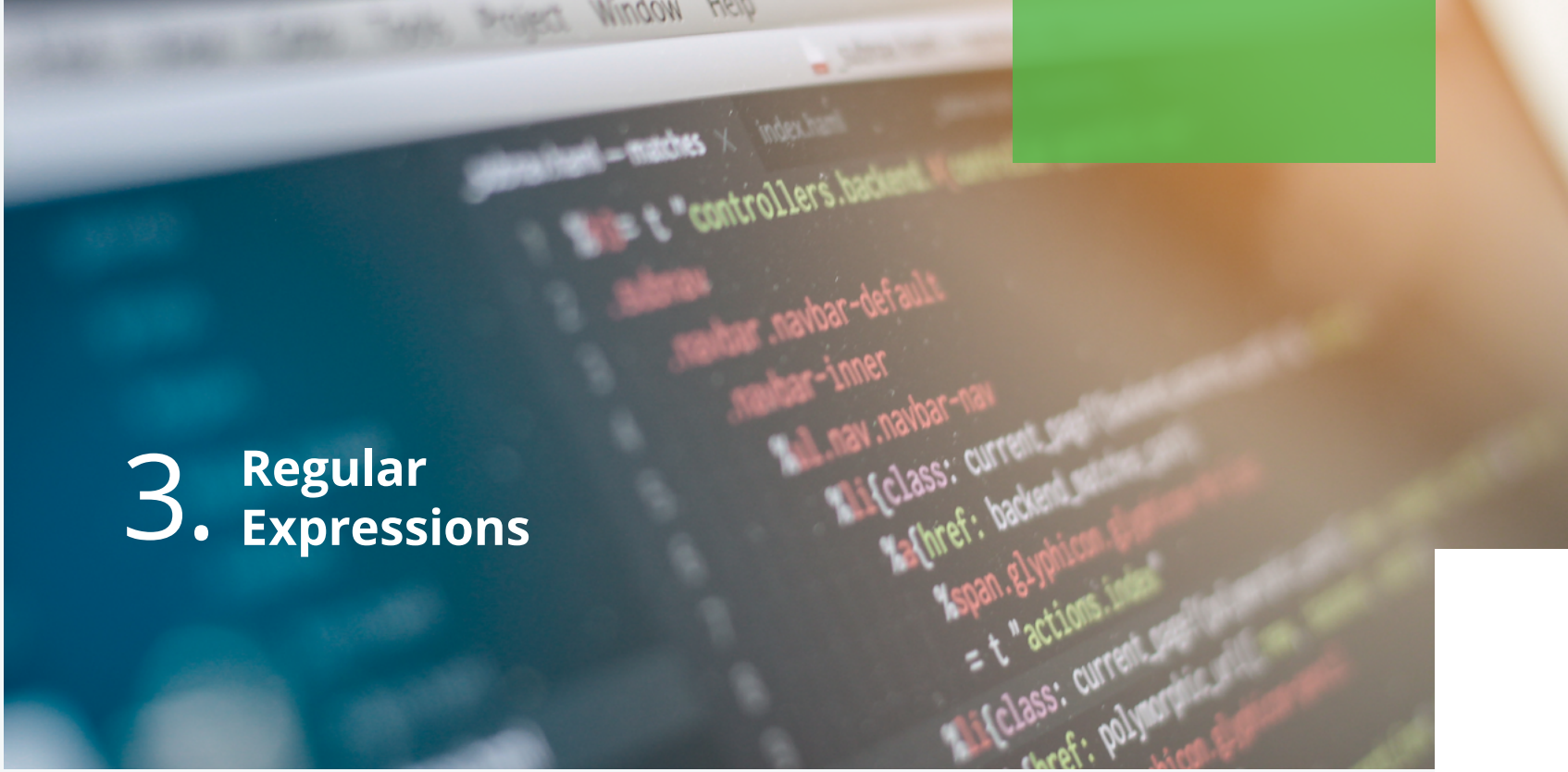
Even if you validate and sanitize all of the data that is entering your application, it is also important to validate it before outputting it to clients, since it is at the output phase that injections actually occur. If you neglect input or output validations, you might be exposing your application to some of the most notorious web application attacks: XSS and SQLi.

Without going in to great detail, XSS allows an attacker to inject malicious code into web pages [2]. In Go, you can prevent XSS with some HTML filtering functions from the `text/html` package:

- `HTMLEscapeString:` returns the escaped HTML equivalent of the plain text inputted data
- `JSEscapeString:` returns the escaped JavaScript equivalent of the plain text inputted data

An important point that you should also take into consideration is the content type in your HTTP headers. It should be validated so it does not render the content in an unspecified and potentially dangerous manner.

SQLi happens because of the lack of proper input or output validation. To keep it simple, if a variable that contains characters (with special meaning to the database management system) is added to a SQL query, your application becomes vulnerable to SQLi [3]. To prevent SQLi, you should follow the standard good practices to address it [4], like the usage of prepared statements or stored procedures. In Go, you can use the **Prepare** function from the `database/sql` package, which will allow you to create parameterized queries. However, you need to import the appropriate driver for your database, and the chosen `driver` needs to implement the required method. With these conditions, the database/sql package will be aware of what special characters it needs to handle—and will escape them.



3. Regular Expressions

A regular expression, *regex*, is a sequence of characters that defines a search pattern, which is used to search for one or more characters within a string. You may want to take advantage of it to various ends, some of them even related to the input validation point. For example, in some programming languages, when you use the built-in regular expressions module/package, you might be indirectly exposing yourself to a security vulnerability, called *ReDoS*, which stand for regular expression denial of service [5]. In short, it happens because of some vulnerable regular expressions that can take exponential amounts of time while trying to match a maliciously-crafted input, exposing your application to a possible denial of service.

In Go, you can rest assure by using the **regexp** package, as it guarantees to run in linear time [6].

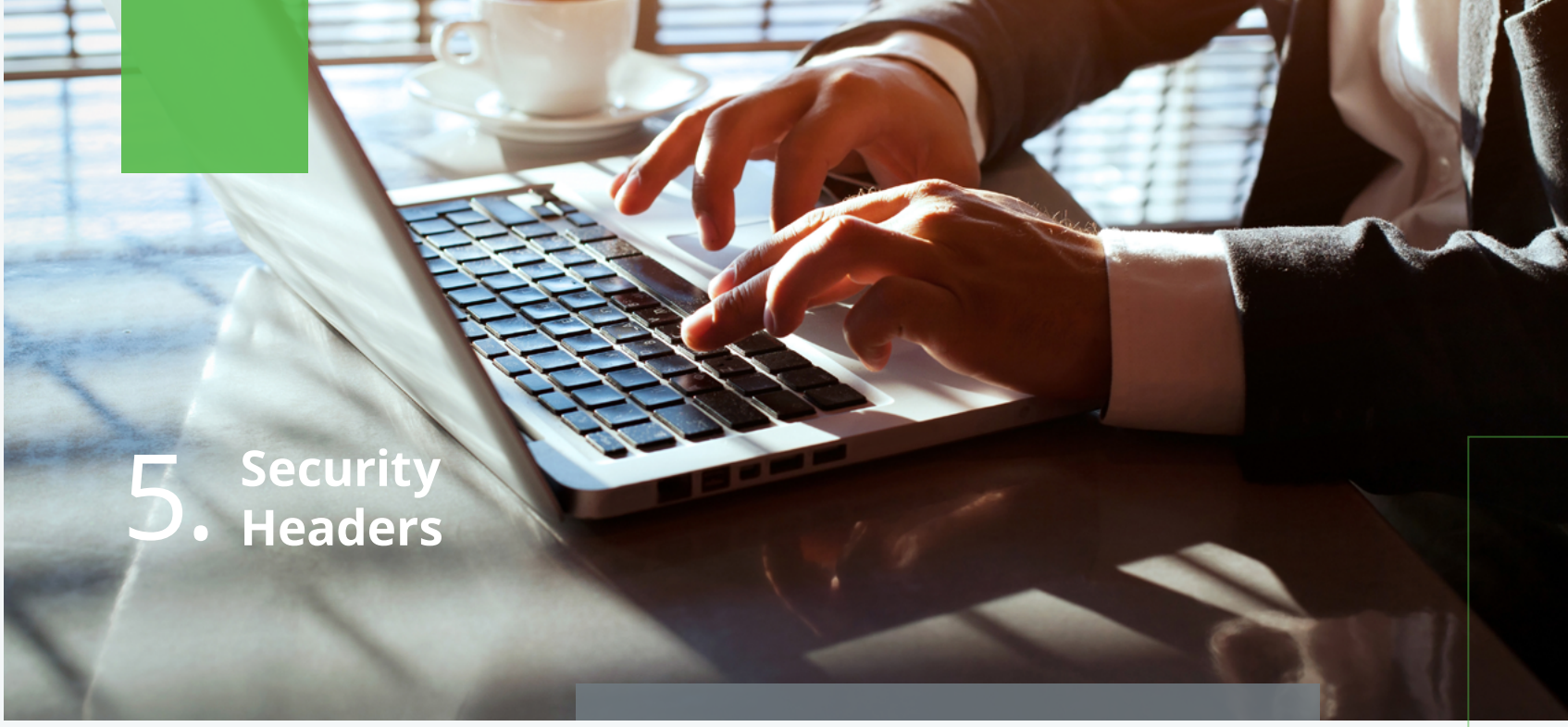
4. Cross-site Request Forgery (CSRF)

CSRF (or XSRF) stands for Cross-Site Request forgery [7], and it is commonly known as *one click attack* or *session riding*. In short, it is an attack that forces the end user to perform unwanted actions on a web application in which they are currently authenticated. CSRF is possible because the server does not distinguish between requests that are made during a legitimate user session workflow, and malicious requests. Therefore, to prevent for CSRF attacks, the most commonly used approaches are [8]:

- **Check standard headers to verify that the request is from the same origin**
- **Include a pseudo-random number with non-POST requests (aka CSRF token)**

To address this security vulnerability in your Go application, you can use one of the following frameworks to easily solve this issue:

- `gorilla/csrf:` provides CRSF prevention middleware for Go web applications and services
- `nosurf:` is an HTTP package for Go that helps you prevent CRSF attacks
- `csrf:` middleware for CSRF attacks that generates and validates CSRF tokens for Macaron framework



5. Security Headers

To give an extra layer of security to your web application, you can always rely on security headers. These will tell the browser how to behave when handling your site's content [9]. A good example is the Content Security Policy (CSP[10] , which will address some types of attack by allowing you to control the resources the user agent is allowed to load for a given page. These attacks can be: XSS, clickjacking, and other data injection attacks. If you'd like to add an extra layer of protection in your application against XSS attacks, you could use the *script-src* definition, which block scripts unless they are from the sources specified in the policy [11].

In Go, you can simply enable and configure all the security headers that you want, referring to one of the following packages:

- **secure:** is an HTTP middleware for Go that facilitates some quick security wins. It's a standard net/http Handler and can be used with many frameworks or directly with Go's net/http package

6. Sessions and Cookies

The proper configuration of sessions and cookies are key points to secure your web application and its clients. Also, it is quite obvious that you don't want your applications' clients to get their cookies stolen, right?

For session management, your application should only recognize the server's session management controls, and the session creation should be done on a trusted source [12].

For cookies security, you can use the `net/http` package, specifically the `SetCookie` function to properly configure it. You will be able to easily set the *httpOnly* and *secure* flags with this function.

7. Communication Security

In order to guarantee data integrity for your web application, and be protected against attacks targeting communications security, (e.g., MITM [13],) you should configure secure channels for your application.

In Go, you can address communication security using the `crypto/tls` package. With this package you will be able to do things like: implement SSL and TLS, and enforce HSTS.

For the TLS implementation, there are some points that should be reviewed:

- Set the **InsecureSkipVerify** to false
- Use the correct hostname to set the server name
- Use the **GetClientCertificate** and its associated error code to guarantee no insecure connections are established

8. Secure Data Storage

Starting at password storage, when looking for a mechanism for your web application, you should consider a package that uses the strongest algorithms, since it will impede an attacker from easily breaking a password hash from a leaked database.

You have a couple of good alternatives in Go:

- `bcrypt` package, implements Provos and Mazieres' [14] bcrypt adaptive hashing algorithm
- `blake2b` package, implements the BLAKE2b hash algorithm defined by RFC 7693, and the extendable output function (XOF) BLAKE2Xb that will allow you to hash, store, and validate user passwords.

If you want to encrypt some other sensitive data, the crypto package can help you with the right functions. For example, the `crypto/aes` package is a great choice for a proper and reliable encryption.

9. Memory Management & Concurrency

When developing your application, one of the most important points is the memory management. It is very important to have a stable application with efficient memory consumption so it does not leak memory, which eventually could lead to an unpleasant crash of the application.

Go language uses garbage collection for memory management [15], releasing you from that overhead, and allowing you to focus on other points of the development. This assists in concurrent and multi-threaded programming. As objects get passed among threads, it becomes difficult to guarantee they become freed safely. With automatic garbage collection, concurrent coding gets easier to write.

The Go language checks for boundaries in strings, arrays, and slices [16], but when dealing with functions that accepts a number of bytes to copy, you should perform buffer boundary checking. In such cases, the size of the destination array must be checked to ensure it does not write beyond the allocated space. If you do not perform this boundary check, your Go application will **Panic**, and at some point, crash. Panic is a built-in function that stops ordinary flow of control. So, always perform boundary checks.

When Panic occurs at *goroutines*, you should use the built-in function `Recover` to regain control of it. Recover is only useful inside *deferred* functions. In Go, a `Defer` statement is used to defer the execution of a function until the surrounding functions returns. Hence, if a goroutine is panicking, a call to Recover will capture the value given to panic and resume the normal execution.

Another point in Go memory management is the usage of the *compile* tool. This tool can help you performing Escape Analysis [17]. If you are concerned that a created variable can escape out of its function or to other *goroutines*, you can use the **go tool compile -m** to print out optimization decisions, that will include escape analysis.

10. System Configuration

When your application is built, and you've followed the security points above to properly secure it, you will be one step closer to a proper security-maturity level. However, you could still leave everything at risk if you do not properly configure the system security that will maintain your web application.

In Go and in other technologies, you should always keep things updated. So, a starting point is to always use the latest version for the whole project, external packages, and frameworks.

A common vulnerability that happens from an insecure system configuration is directory listing, whereas its names suggests, it will allow a non-authorized user to view sensitive files and navigate through directories. To address this issue, you should make sure that the directory does not contains sensitive information, in addition to restricting directory listings. In this case, the mitigation for this vulnerability is independent to the language used, therefore, the best way to address it is to follow some of the proposed countermeasures:

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

Associated with this, there is also the *Directory Traversal* or *Path Traversal*, which allows attackers to access restricted directories and execute commands outside of the web server's root directory. To address this in Go, you can benefit from the http package, specifically the `HandleFunc` and `Handler` functions, using them to block a directory, or limiting it to an allowed map.

It is also important to remove or disable functionalities that you don't need that could help an attacker compromise your application. Disabling debugging modes, removing headers that disclosure sensitive information (like webserver, framework, programming language, or operating system versions) etc., will keep your application safer.

Summary

In this short summary, we've reviewed some key points of web application security for applications written in the Go Language. The main objective of this summary was to walk you through some key points to help you address the most common web application vulnerabilities and keep your Go application safe from harm. It is intended that implementation details are to be further investigated by following the pointed references at the end of this summary. As stated in the beginning, this is a summary of the **Go Language Guide – Web Application Secure Coding Practices**, and therefore, this summary should be kept on-hand when following that Guide.

Disclaimer: As a final note, please remember that no system or programming language is 100% secure, and one needs to be constantly reviewing, testing, and carefully configuring the application to stay ahead of malicious actors. However, if you follow these principles, you should be Good to Go!

References

[1] **OWASP Input Validation Cheat Sheet,**
Available at: https://www.owasp.org/index.php/Input_Validation_Cheat_Sheet

[2] **OWASP Cross-Site Scripting,**
Available at: https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)

[3] **OWASP SQL Injection**
Available at: https://www.owasp.org/index.php/SQL_Injection

[4] **OWASP SQLi Prevention**
Available at: https://www.owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet

[5] **ReDoS**
Available at: https://www.owasp.org/index.php/Regular_expression_Denial_of_Service_-_ReDoS

[6] **ReDoS in Go**
Available at: https://www.checkmarx.com/2018/05/07/redos-in-go/

[7] **OWASP CSRF**
Available at: https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)

[8] **OWASP CSRF Prevention**
Available at: https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)_Prevention_Cheat_Sheet

[9] **OWASP Security Headers**
Available at: https://www.owasp.org/index.php/OWASP_Secure_Headers_Project#tab=Headers

[10] **OWASP Content Security Policy**
Available at: https://www.owasp.org/index.php/Content_Security_Policy

[11] **An Introduction to Content Security Policy**
Available at: https://www.html5rocks.com/en/tutorials/security/content-security-policy/

[12] **OWASP Session Management**
Available at: https://www.owasp.org/index.php/Session_Management_Cheat_Sheet

[13] **OWASP MITM attack**
Available at: https://www.owasp.org/index.php/Man-in-the-middle_attack

[14] **A Future-Adaptable Password Scheme, Niels Provos and David Mazieres**
Available at: https://www.usenix.org/legacy/event/usenix99/provos/provos.pdf

[15] **Getting to Go: The Journey of Go's Garbage Collector**
Available at: https://blog.golang.org/ismmkeynote

[16] **Index expressions**
Available at: https://golang.org/ref/spec#Index_expressions

[17] **Escape analysis**
Available at: https://en.wikipedia.org/wiki/Escape_analysis