A Security Guide for GO Developers

INDEX

Overview..1

Invalid or Unsanitized Input....................................2

Injection Risks During Output...................................3

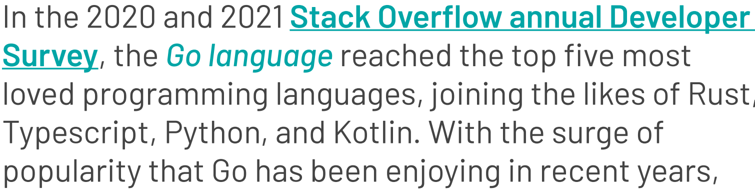
Unsound Authentication Policy and Password Management..5

Insufficient Error Handling Practices and Logging Policy..6

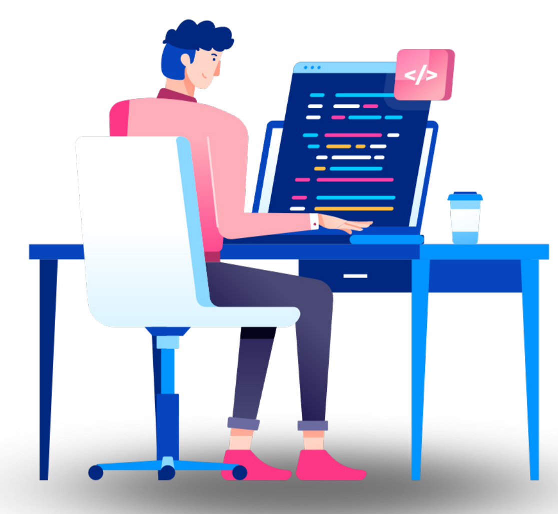
Lacking Data Protection Measures............................7

Security Considerations for Go vs. Other Languages..9

Protect Your Go Programs with Kiuwan..................10



Overview



In the 2020 and 2021 Stack Overflow annual Developer Survey, the *Go language* reached the top five most loved programming languages, joining the likes of Rust, Typescript, Python, and Kotlin. With the surge of popularity that Go has been enjoying in recent years, it's being increasingly used for projects that are becoming the building blocks of future digital infrastructure. The container management system Kubernetes is one such example. According to the Cloud Native Computing Federation, Kubernetes is used by nearly one-third of all backend developers — that's nearly 5.6 million people.

Go — the shortened nomenclature for Golang — has been described as a programming language that helps companies more easily execute their missions by focusing on the problems they want to solve as opposed to the tools they want to use. Much of the praise comes from the strength of its simplicity. Indeed, it's also one of the most in-demand languages that developers want to learn.

Go is on an upwards trajectory and will remain so for the foreseeable future. That means for companies looking to leverage the programming language to its fullest potential, they need to understand the nuts and bolts of its important facets, such as how it fares in terms of application security (appsec). Does Go have more or less security vulnerabilities compared to other popular languages? Is it easier to manhandle into appsec policies and standards, or is it harder?

While this Go security guide may not be exhaustive, it serves to highlight some of the most common security vulnerabilities developers should keep an eye out when programming with Go. It also touches on the differences between Go and other popular languages in terms of security, taking a look at the cost of slacking in appsec.

This guide will explore the following ***most common risks*** when developing in Go:

- Invalid or Unsanitized Input
- Injection Risks During Output
- Unsound Authentication Policy and Password Management
- Insufficient Error Handling Practices and Logging Policy
- Lacking Data Protection Measures

Invalid or *Unsanitized Input*

Validating inputs from users helps avoid opening up the system to attackers who send intrusive information, and, of course, it helps with functionality as well. Standardization of input validation also helps users become more confident by avoiding simple mistakes that may have disastrous consequences. Left unchecked, user input and any associated information can easily become an appsec concern. This potential threat is addressed via input validation and sanitization. Ideally, these are performed at every part of an application if the function of the server permits such.

As applications become richer and more complex, so too do their user interaction requirements and the amount of submitted data that's processed. This translates to more potential pitfalls when it comes to input validation and sanitization.

Generally, validation procedures must be performed on trusted systems, and consist of three general checks: validation, post-validation, and sanitization.

For input validation, developers should have these checks in place:

- **Validation of every user input and user interactivity** — whenever an application allows user input, it's a potential threat. Not only can malicious actors mislead users into compromising the application, but they can also take advantage of human error to gain an initial foothold through an inadvertent vulnerability.

  For programming in Go, when dealing with strings, packages such as strconv and strings are applied to handle conversions to other data types. Furthermore, Go also supports regular expressions like utf8 and regexp for complex validations. Users may also rely on third-party packages that include easier validations for individual fields or structs — though take note that not only does Go prefer native libraries, strict security protocols may strictly dictate third-party package use.

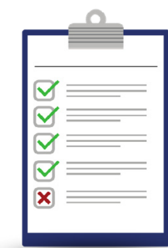
- **Validation during every file manipulation** — Every time file usage is required (i.e. a file is read or written), validation checks should be in place to ensure that file manipulation operations that deal with user information are handled correctly.

- **Validation of data sources** — Every time information is passed from one trusted source to somewhere less trusted, there should be integrity checks in place such as cross-system consistency checks, referential integrity checks, and hash totals.

Within post-validation, there are actions that vary depending on context and are generally split into three categories:

Enforcement actions – These actions are meant to better secure data and applictions. These actions include:

- Informing users that they should modify data input as it failed to comply with requirements
- Modifying input data to comply with requirements without user notification



Advisory actions – One level down from enforcement actions, advisory actions allow unchanged data to be submitted, but informs the source actor that the issues were detected with the input.

Verification actions – Special cases within advisory actions occur when users submit data and are then asked by the source actor to verify the information. The source actor may also suggest changes. The user then decides to keep the original input or implement the suggestions.

In terms of sanitization, it's simply the process of replacing or removing submitted data. Even after proper validation checks, input data should still go through sanitization as an additional step to further strengthen appsec.

Injection Risks During Output

In web application development, poor output encoding practices are, unfortunately, rampant. These undesirable practices multiply the risks associated with web apps becoming more complex and increasing amounts of data sources — from more users to additional databases and integrations of third-party services. At some point, collected data will be outputted to some media with specific context. And in that instance, injections can compromise applications without robust output encoding policy.

For Golang, two injections are particularly pesky: **cross-scripting (XSS)** and **SQL injections**.

An ***XSS attack*** exploits the capability to inject malicious code to modify output. These attacks occur when unfiltered strings are sent back to a web client, usually through the text/template package or io.WriteString(). A common example would be a threat actor sending JavaScript code as part of a query string in a URL, which is then executed once the app returns the user's value. This exploit exists because HTML tags in the returned strings will be rendered without encoding to the output stream, so they may be incorrectly defined as a plain/text response header content-type — unless it is explicitly set otherwise.

Fortunately, Go offers the html/template package to encode what apps return to users. It's just a matter of enforcing output encoding policy to encourage best practices in coding. If required, there are also third-party libraries that help with encoding authentication cookie values, for example.

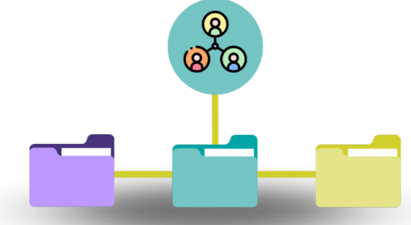
An **SQL injection**, on the other hand, typically stems from yet another bad practice: poor string concatenation. SQL injection usually happens when variables holding values that may include arbitrary characters assigned special meaning to the app database are added to partial SQL queries. SQL injections are a common vulnerability across many programming languages.

However, within Go, there are specific considerations, such as making sure users connecting to the database are given limited permissions. Of course, input sanitization is also highly recommended. Developers can also foolproof their apps by using the HTMLEscapeString function in the HTML template package to escape special characters.

By far, the most critical solution to this conundrum, of course, is the use of parameterized queries. In Go, you prepare statements on the database, not in a connection.

In case the database engine doesn't fully support the use of prepared statements or their involvement somehow affects query performance, users can employ the db.Query() function in conjunction with robust input sanitization measures.

Finally, third-party libraries can prevent SQL injections, e.g. sqlmap.



Unsound *Authentication Policy* and *Password Management*

Authentication and password policy and management are critical for all applications and systems, and they're often governed by detailed specs in terms of everything from user signup to storage of credentials to resetting passwords. Programming in Go is no exception.

The general rules of thumb precede specific steps:

- All authentication controls need to be enforced on a trusted system
- Only standard and proven authentication services should be utilized to not only simplify the system but also to reduce possible points of failure
- Code sanitization is a must: closely inspect code for malicious parts
- Resources that need to undergo authentication should not perform it on themselves — ideally, the system can redirect to and from centralized authentication control
- Both users and the application itself should be able to use authentication to ensure that connection to external systems can also be authenticated without manual work

In terms of ***Go-specific steps***, the strict use of HTTPS comes first to mind. Most contemporary browsers require HTTPS on every website — Chrome, for instance, alerts users if websites aren't on HTTPS. If possible, the same should be done throughout an application, e.g. it should be standard appsec policy to enforce in-transit encryption when two services communicate. This means encryption connections from specific ports as well as utilizing proper certificates to enforce HTTP, lest threat actors downgrade the protocol and open up vulnerabilities.

In line with this, all authentication credentials should be sent only through HTTPS encrypted channels. Some exceptions can be reviewed, such as email reset requests for temporary passwords.

When applications handle authentication errors, they should not reveal which part of the authentication data failed the checks. For example, application error messages should say "invalid username or password" instead of specifying which was incorrect.

To further this practice, users should be informed about the last successful and unsuccessful dates of access after successfully logging in. This helps them more easily identify suspicious activity. To prevent timing attacks, it wouldn't hurt to use constant time comparison functions when checking passwords.

A final word on sensitive information management: encrypt, encrypt, encrypt. Note that a string in base-64 format, for example, is only hard to read — it doesn't necessarily mean its hidden value is kept secret. Successful encryption means hiding information in a way that cannot be easily decoded. For Go, some recommended encryption algorithms include scrypt, bcrypt, PDKDF2, or Argon2. Use these to heavily encrypt information such as passwords (for users and databases) and other confidential information.

For the strictest security measures, refrain from using third-party crypto packages outside of Go's standard ones. Go's crypto packages are well-audited and native, and thus will work well with any security process and management standards and policies you set up without needing additional scrutiny.

Insufficient *Error Handling Practices* and Logging Policy

Robust and comprehensive standards, processes, and policies should also govern the error handling and logging in golang applications. Logging, especially, may often be overlooked particularly from an appsec perspective, but like error handling, it's an essential part of protecting application and infrastructure.

Error handling refers to errors in application logic that may cause system crashes unless handled properly. Systems should not reveal too much information about the error to users, as these may then be used by attackers to make useful inferences, such as what technologies or services are being used. In Golang, there are no exceptions — errors are handled differently compared to other languages, and this naturally affects security considerations in error handling.

Furthermore, there are additional error handling functions in Go that allow for the recovery of an application instead of an unrecoverable failure. These functions should also be taken into consideration.

Logging is the reporting of operation highlights and requests that occurred in-system. By its very nature, logging lets users identify operations that have occurred. Likewise, it often indicates what actions are required to protect the application. This is why threat actors often try to delete logs to clear their tracks. This, in turn, is why logs are best centralized.

Go actually offers a native library to work with logs. Unfortunately, Go's log package "implements simple logging" according to its documentation, so expect it to fall short of some common and important features such as leveled logging and formatters support. Users will have to implement additional logging functions to make logs usable via integration, for example, with Security Information and Event Management systems.

Error logs, like error notifications, should not disclose too much sensitive information. Internal policy should also ensure that error handlers don't inadvertently leak critical data such as stack trace or debugging information.

As best practice, logging should be handled by the application. It should not rely on a server configuration. Ideally, logging is implemented by a master routine within a trusted system, with all sensitive information like session data, system details, and passwords dutifully scrubbed. Logging should also be enabled for both successful and unsuccessful security events.

Lacking Data Protection Measures

How effective a company's Golang security measures will be depends on the data protection measures they enforce. While there are generally adopted best practices that span every programming language, there are Golang-specific considerations that need to be factored in for companies developing applications in the language.

Among the foremost priorities when it comes to data protection measures for Golang is the creation and implementation of the proper privileges for each user which then restrict them to only the functions required by their role. To better illustrate this point, consider the following users in an online store and their function limitations:

A sales user should only be able to view a catalog

A marketing user can be given one level of access higher by checking stats and metrics

A developer is allowed all of that as well as the ability to modify webpages and applications

In addition, organizations should also define proper permissions in the web server.

After the priority step of managing permissions, company appsec policy should turn to the removal of sensitive information in temporary and cache files as soon as they are no longer required. In case of prolonged use, these temporary files should be encrypted or moved to protected areas — ideally both.

In line with error handling and logging measures discussed above, developer comments should also be governed to ensure programmers are not divulging too much information. Sometimes programmers are prone to leaving to-do lists in code comments, at worst leaving credentials for easier access at a future time.

Data protection measures should also be reinforced when passing sensitive information between sources, especially when using the HTTP GET method. This method opens up a web app to **vulnerabilities** such as:

- **Data being intercepted by Man-in-the-Middle attacks if communicated without HTTPS** (an application api_key can be stolen when transmitted to a third-party site that's not using HTTPS)
- **Browser history storing user information** (session IDs and tokens that have low entropy or don't expire can be extracted from URLs)
- **Search engines indexing URLs found in pages**
- **HTTP servers writing requested URLs including query strings onto unencrypted log files**

Aside from these concerns, note that regardless of whether an organization uses HTTP or otherwise, parameters passed through GET are stored in clear, the browser's history, and the access log of the server.

Always enforce HTTPS, especially when transmitting or communicating with external parties in order to capture exchanged data. Critical pieces of information such as the api_key are best included in some header or in the request body. In the same vein, make use of one-time-only tokens or session IDs to minimize the potential for misuse.

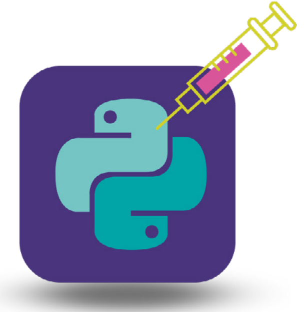
The production environment should be rid of all application and system files that are no longer needed. While they may seem fairly innocent, unattended documentation like ReadMe or Changelog files may disclose versions or functions that can either be directly used in an attack, leveraged to gain an initial foothold, or even serve as additional authenticating information used in phishing lures.

Lastly, in the same way unnecessary files are scrubbed, systems may disable services or applications that are not actively needed in operation, such as autocomplete features and caches.

Security Considerations for Go vs. Other Languages

Go is often compared with other programming languages due to similar features. It was primarily designed to facilitate faster compilation — that principle in and of itself already creates distinctions from other languages. Compared to C++, for instance, Go increases memory and decreases dependencies. Put up against Java, Go certainly compiles faster and doesn't require a virtual machine, not to mention the design principles of the latter were meant to counter the verbosity and complexity apparent in the former.

Of course, that also means security considerations for Go are inherently different from those of other languages. To better illustrate, consider the common vulnerabilities for two other languages often compared with Go: Ruby and Python.



Python is also prone to SQL injection — this is a near-constant commonality — but one of its well-known vulnerabilities is directory traversal. The mechanism exploited here is similar to Go: improper user input sanitization upon file access. Still, the resulting vulnerability is unique to Python when compared to Go. Another typical point of failure for Python is outdated dependencies or modules. Python's popularity and wide-scale adoption are partly to blame for this vulnerability, and so far, while Go has been gaining in popularity, it hasn't run into this issue as much. One more example is Python's built-in assertion functionality, where insufficient logic may lead to code inadvertently skipping validation statements due to being in debug mode.

In Ruby, too, there are vulnerabilities distinct from Go. Unsafe serializing and deserializing of user input is one such pitfall. Another is input-output hijacking, where fairly innocuous functions like Kernel::open can be exploited due to additional features such as spawning processes from which output can be piped. Naturally, there are "versions" of the same vulnerabilities in Go and Python — Ruby also shares the same SQL injection vulnerability. The difference is in the method and the results.

The cost for developing using these languages without due concern for appsec is high. Equifax found out in 2017 how much massive data breaches cost: over $575 million in settled lawsuits stemming from a hack that exposed information of over 145 million people. A study in 2015 found that costs incurred for each compromised record containing confidential information amounted to $154 with the average cost of a data breach reaching $3.8 million.

Kiuwan's Capabilities Secure Go Applications

So, what does this mean for a development company's Go applications? It means they can't adopt a wait-and-see approach. What starts out small can snowball into catastrophic proportions. Kiuwan can help secure Go applications with code security and analysis tools that automatically identify and remediate vulnerabilities.

Of course, aside from being able to secure Go applications, Kiuwan covers over 30 programming languages with its security capabilities. Kiuwan offers DevSecOps capabilities that can identify code vulnerabilities and third-party components in your code, and our solutions integrate with your existing DevOps tech stack to automate security processes.

Book a demo with Kiuwan today.

YOU KNOW CODE, WE KNOW SECURITY!

GET IN TOUCH:

Headquarters

2950 N Loop Freeway W, Ste 700

Houston, TX 77092, USA

United States **+1 732 895 9870**

Asia-Pacific, Europe, Middle East and Africa **+44 1628 684407**

contact@kiuwan.com

Partnerships: **partners@kiuwan.com**