

# ATTACKING GO

MODERN SYSTEM APPLICATION  
VULNERABILITIES COMPREHENSIVE  
ANALYSIS



# Attacking Golang

• Jun 24, 2024 •  26 min read

## Table of contents

1. SQL Injection

2. Command Injection

3. Cross-Site Scripting (XSS)

4. Insecure Deserialization

5. Directory Traversal

6. CSRF

- › 1. CSRF Tokens
- › 2. SameSite Cookies
- › 3. Referer Header Validation

7. SSRF

- › 1. Input Validation and Whitelisting
- › 2. Restrict IP Ranges

8. File Upload

- › 1. Validate File Type and Content
- › 2. Use Server-Side File Type Detection

9. Memory Management Vulnerabilities

- › 1. Bounds Checking
- › 2. Avoiding Memory Leaks
- › 3. Use of Pointers and References

## 10. Cryptography Failure

- › Issues in the Code:
- › 1. Use Strong Hashing Algorithms
- › 2. Secure Encryption with AES-GCM

## 11. LFI and RFI

- › Common Vulnerabilities and Mitigations

## 12. Basic Authentication (BasicAuth) alongside JSON Web Tokens (JWT)

- › 1. Lack of HTTPS/TLS Encryption
- › 2. Insufficient JWT Validation
- › 3. Lack of Rate Limiting

## 13. Golang pitfalls

- › 1. Unsafe Package Misuse
- › 2. Insecure Random Number Generation
- › 3. Improper Error Handling
- › 4. SQL Injection

## 14. RPC

- › 2. Denial-of-Service (DoS) Attack
- › Mitigation
  - › 1. Use Secure Authentication and Authorization
  - › 2. Validate and Sanitize Inputs

## 15. Timing Attack

- › 1. Constant-Time Comparison
- › References

Show less ^

Golang (or Go) is a statically typed, compiled programming language designed at Google. It is known for its simplicity, efficiency, and strong performance. However, like any programming language, improper coding practices in Go can lead to security vulnerabilities. This article explores common security issues and how to mitigate them in Go.

# 1. SQL Injection

SQL injection occurs when an attacker is able to manipulate a SQL query by injecting arbitrary SQL code into the input fields.

## Vulnerable Code:

```
package main

import (
    "database/sql"
    "fmt"
    _ "github.com/lib/pq"
    "net/http"
)

func main() {
    http.HandleFunc("/login", func(w http.ResponseWriter, r
    *http.Request) {
        username := r.FormValue("username")
        password := r.FormValue("password")

        db, err := sql.Open("postgres", "user=username dbname=mydb
        sslmode=disable")
        if err != nil {
            http.Error(w, err.Error(), http.StatusInternalServerError)
            return
        }
        defer db.Close()
```

COPY 

```

        query := fmt.Sprintf("SELECT * FROM users WHERE username='%s'
AND password='%s'", username, password)
        rows, err := db.Query(query)
        if err != nil {
            http.Error(w, err.Error(), http.StatusInternalServerError)
            return
        }
        defer rows.Close()

        if rows.Next() {
            w.Write([]byte("User Login Successful"))
        } else {
            w.Write([]byte("Wrong Username Password Combination"))
        }
    })

    http.ListenAndServe(":8080", nil)
}

```

An attacker can exploit this by entering `username' OR '1'='1` as the username and any password, which would allow access to the system.

### Improved Code:

```

package main

import (
    "database/sql"
    "net/http"
    _ "github.com/lib/pq"
)

func main() {
    http.HandleFunc("/login", func(w http.ResponseWriter, r

```

COPY 

```

*http.Request) {
    username := r.FormValue("username")
    password := r.FormValue("password")

    db, err := sql.Open("postgres", "user=username dbname=mydb
sslmode=disable")
    if err != nil {
        http.Error(w, err.Error(), http.StatusInternalServerError)
        return
    }
    defer db.Close()

    var user string
    err = db.QueryRow("SELECT username FROM users WHERE
username=$1 AND password=$2", username, password).Scan(&user)
    if err != nil {
        if err == sql.ErrNoRows {
            w.Write([]byte("Wrong Username Password Combination"))
        } else {
            http.Error(w, err.Error(),
http.StatusInternalServerError)
        }
        return
    }

    w.Write([]byte("User Login Successful"))
})

http.ListenAndServe(":8080", nil)
}

```

Using parameterized queries (prepared statements) prevents SQL injection by ensuring that user inputs are properly escaped.

## 2. Command Injection

Command injection occurs when an attacker can execute arbitrary commands on the host system via unsanitized user input.

### Vulnerable Code:

```
package main

import (
    "net/http"
    "os/exec"
)

func main() {
    http.HandleFunc("/run", func(w http.ResponseWriter, r
    *http.Request) {
        cmd := r.FormValue("cmd")
        out, err := exec.Command("sh", "-c", cmd).Output()
        if err != nil {
            http.Error(w, err.Error(), http.StatusInternalServerError)
            return
        }
        w.Write(out)
    })

    http.ListenAndServe(":8080", nil)
}
```

An attacker could enter `rm -rf /` as the command to delete all files on the server.

**Improved Code:** Avoid directly executing user input. If necessary, validate and sanitize inputs thoroughly, or use predefined commands.

```

package main

import (
    "net/http"
    "os/exec"
)

func main() {
    http.HandleFunc("/run", func(w http.ResponseWriter, r
*http.Request) {
        cmd := r.FormValue("cmd")
        allowedCommands := map[string]bool{
            "date": true,
            "uptime": true,
        }

        if !allowedCommands[cmd] {
            http.Error(w, "Invalid command", http.StatusBadRequest)
            return
        }

        out, err := exec.Command(cmd).Output()
        if err != nil {
            http.Error(w, err.Error(), http.StatusInternalServerError)
            return
        }
        w.Write(out)
    })

    http.ListenAndServe(":8080", nil)
}

```

This code restricts command execution to a predefined set of safe commands.

### 3. Cross-Site Scripting (XSS)



XSS occurs when an attacker injects malicious scripts into content from otherwise trusted websites.

### Vulnerable Code:

```
package main

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

func main() {
    http.HandleFunc("/greet", func(w http.ResponseWriter, r
    *http.Request) {
        name := r.FormValue("name")
        tpl := `<html><body>Hello {{.}}</body></html>`
        t := template.Must(template.New("greet").Parse(tpl))
        t.Execute(w, name)
    })

    http.ListenAndServe(":8080", nil)
}
```

COPY 

An attacker can input `<script>alert('XSS')</script>` as the name to execute a script on the client side.

**Improved Code:** Use Go's `html/template` package which automatically escapes HTML.

```
package main
```

```
import (
```

COPY 

```

    "html/template"
    "net/http"
)

func main() {
    http.HandleFunc("/greet", func(w http.ResponseWriter, r
    *http.Request) {
        name := r.FormValue("name")
        tpl := `<html><body>Hello {{.}}</body></html>`
        t := template.Must(template.New("greet").Parse(tpl))
        t.Execute(w, template.HTML(name))
    })

    http.ListenAndServe(":8080", nil)
}

```

The `html/template` package ensures that special characters are escaped, preventing XSS.

## 4. Insecure Deserialization

Insecure deserialization occurs when untrusted data is used to instantiate an object. This can lead to remote code execution.

**Vulnerable Code:**

```

package main

import (
    "encoding/gob"
    "net/http"
)

func main() {

```

COPY 

```


    http.HandleFunc("/deserialize", func(w http.ResponseWriter, r
    *http.Request) {
        var data map[string]interface{}
        gob.NewDecoder(r.Body).Decode(&data)
        // Use data...
    })

    http.ListenAndServe(":8080", nil)
}

```

An attacker can craft a payload that executes arbitrary code during deserialization.

**Improved Code:** Validate and sanitize input before deserialization.

COPY 

```

package main

import (
    "encoding/json"
    "net/http"
)

func main() {
    http.HandleFunc("/deserialize", func(w http.ResponseWriter, r
    *http.Request) {
        var data map[string]interface{}
        if err := json.NewDecoder(r.Body).Decode(&data); err != nil {
            http.Error(w, "Invalid input", http.StatusBadRequest)
            return
        }
        // Use data...
    })

    http.ListenAndServe(":8080", nil)
}

```

Using JSON deserialization is safer because it does not support arbitrary code execution.

## 5. Directory Traversal

Directory traversal occurs when an attacker is able to access files and directories that are outside of the intended directory.

### Vulnerable Code:

```
package main

import (
    "io/ioutil"
    "net/http"
)

func main() {
    http.HandleFunc("/file", func(w http.ResponseWriter, r
    *http.Request) {
        filename := r.URL.Query().Get("filename")
        data, err := ioutil.ReadFile("/var/www/" + filename)
        if err != nil {
            http.Error(w, "File not found", http.StatusNotFound)
            return
        }
        w.Write(data)
    })

    http.ListenAndServe(":8080", nil)
}
```

COPY 

An attacker can request `?filename=../../../../etc/passwd` to read sensitive files.

**Improved Code:** Validate the file path to ensure it does not traverse directories.

COPY 

```
package main

import (
    "net/http"
    "path/filepath"
    "strings"
    "io/ioutil"
)

func main() {
    http.HandleFunc("/file", func(w http.ResponseWriter, r
    *http.Request) {
        filename := r.URL.Query().Get("filename")
        cleanPath := filepath.Clean("/var/www/" + filename)

        if !strings.HasPrefix(cleanPath, "/var/www/") {
            http.Error(w, "Invalid file path", http.StatusBadRequest)
            return
        }

        data, err := ioutil.ReadFile(cleanPath)
        if err != nil {
            http.Error(w, "File not found", http.StatusNotFound)
            return
        }
        w.Write(data)
    })

    http.ListenAndServe(":8080", nil)
}
```

## 6. CSRF

In a CSRF attack, an attacker tricks the user into submitting a request to the web application in which the user is authenticated. This can lead to unintended actions being performed on behalf of the user without their consent.

### *Vulnerable Code Example*

Here's an example of a vulnerable Go application where a CSRF attack is possible:

```
package main

import (
    "fmt"
    "net/http"
)

func main() {
    http.HandleFunc("/transfer", func(w http.ResponseWriter, r
    *http.Request) {
        if r.Method == http.MethodPost {
            amount := r.FormValue("amount")
            to := r.FormValue("to")
            // Process the transfer (omitted)
            fmt.Fprintf(w, "Transferred %s to %s", amount, to)
        } else {
            http.Error(w, "Invalid request method",
            http.StatusMethodNotAllowed)
        }
    })

    http.ListenAndServe(":8080", nil)
}
```

COPY 

In this example, an authenticated user can make a POST request to `/transfer` to transfer money. An attacker can exploit this by crafting a malicious website that

sends a POST request to `/transfer` when the user visits the attacker's site.

## *Mitigation Strategies*

### 1. CSRF Tokens

One of the most effective ways to prevent CSRF attacks is to use CSRF tokens. These tokens are unique, unpredictable values that are associated with the user's session and included in each request. The server validates the token to ensure the request is legitimate.

#### Secure Code Example:

First, install the `gorilla/csrf` package:

```
go get github.com/gorilla/csrf
```

COPY 

Next, modify the application to use CSRF tokens:

```
package main

import (
    "fmt"
    "net/http"

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

func main() {
    r := mux.NewRouter()

    csrfMiddleware := csrf.Protect([]byte("32-byte-long-auth-key"))
```

COPY 

```

    r.HandleFunc("/transfer", func(w http.ResponseWriter, r
    *http.Request) {
        if r.Method == http.MethodPost {
            amount := r.FormValue("amount")
            to := r.FormValue("to")
            // Process the transfer (omitted)
            fmt.Fprintf(w, "Transferred %s to %s", amount, to)
        } else {
            http.Error(w, "Invalid request method",
            http.StatusMethodNotAllowed)
        }
    })

    http.ListenAndServe(":8080", csrfMiddleware(r))
}

```

In this code, the `csrf.Protect` middleware is used to generate and validate CSRF tokens. The token is automatically injected into forms as a hidden field and included in AJAX requests via custom headers.

## 2. SameSite Cookies

Another measure to mitigate CSRF attacks is to use `SameSite` attributes on cookies. This prevents the browser from sending the cookie along with cross-site requests.

### Setting SameSite Attribute in Go:

```

package main

import (
    "net/http"

func main() {

```

COPY 



```

    http.HandleFunc("/setcookie", func(w http.ResponseWriter, r
    *http.Request) {
        cookie := http.Cookie{
            Name:      "session_id",
            Value:      "some-session-id",
            HttpOnly: true,
            Secure:     true,
            SameSite: http.SameSiteStrictMode,
        }
        http.SetCookie(w, &cookie)
        w.Write([]byte("Cookie set"))
    })

    http.ListenAndServe(":8080", nil)
}

```

In this code, the `SameSite` attribute is set to `Strict`, ensuring the cookie is not sent with cross-site requests.

### 3. Referer Header Validation

As an additional measure, you can validate the `Referer` header to ensure the request originated from your own site.

#### Referer Header Validation Example:

```

package main

import (
    "net/http"
    "strings"
)

func main() {
    http.HandleFunc("/transfer", func(w http.ResponseWriter, r

```

COPY 

```

*http.Request) {
    referer := r.Header.Get("Referer")
    if !strings.HasPrefix(referer, "http://localhost:8080") {
        http.Error(w, "Invalid referer", http.StatusForbidden)
        return
    }

    if r.Method == http.MethodPost {
        amount := r.FormValue("amount")
        to := r.FormValue("to")
        // Process the transfer (omitted)
        w.Write([]byte("Transfer successful"))
    } else {
        http.Error(w, "Invalid request method",
http.StatusMethodNotAllowed)
    }
})

http.ListenAndServe(":8080", nil)
}

```

In this code, the `Referer` header is checked to ensure the request came from the same site.

## 7. SSRF

Server-Side Request Forgery (SSRF) is an attack where an attacker can trick a server into making requests to unintended locations. This could lead to unauthorized access to internal resources, such as accessing internal networks, metadata of cloud instances, or performing unauthorized actions.

### *Vulnerable Code Example*

Here is an example of vulnerable Go code where an SSRF attack is possible:

```
package main

import (
    "io/ioutil"
    "net/http"
)

func main() {
    http.HandleFunc("/fetch", func(w http.ResponseWriter, r
*http.Request) {
        url := r.URL.Query().Get("url")
        if url == "" {
            http.Error(w, "URL is required", http.StatusBadRequest)
            return
        }

        resp, err := http.Get(url)
        if err != nil {
            http.Error(w, "Failed to fetch URL",
http.StatusInternalServerError)
            return
        }
        defer resp.Body.Close()

        body, err := ioutil.ReadAll(resp.Body)
        if err != nil {
            http.Error(w, "Failed to read response",
http.StatusInternalServerError)
            return
        }

        w.Write(body)
    })
}
```

```
http.ListenAndServe(":8080", nil)
}
```

In this example, the server fetches the content of the URL provided by the user and returns it. An attacker can exploit this by providing URLs to internal services or sensitive endpoints.

## *Mitigation Strategies*

### **1. Input Validation and Whitelisting**

One effective way to prevent SSRF is to validate and sanitize user input, allowing only trusted domains.

#### **Secure Code Example:**

```
package main

import (
    "errors"
    "io/ioutil"
    "net/http"
    "net/url"
    "strings"
)

var allowedHosts = []string{"example.com", "api.example.com"}

func isAllowedHost(host string) bool {
    for _, allowedHost := range allowedHosts {
        if strings.Contains(host, allowedHost) {
            return true
        }
    }
    return false
}
```

COPY 

```
}
```

```
func fetchURL(userURL string) ([]byte, error) {  
    parsedURL, err := url.Parse(userURL)  
    if err != nil {  
        return nil, err  
    }  
  
    if !isAllowedHost(parsedURL.Host) {  
        return nil, errors.New("host not allowed")  
    }  
  
    resp, err := http.Get(userURL)  
    if err != nil {  
        return nil, err  
    }  
    defer resp.Body.Close()  
  
    return ioutil.ReadAll(resp.Body)  
}
```

```
func main() {  
    http.HandleFunc("/fetch", func(w http.ResponseWriter, r  
*http.Request) {  
        url := r.URL.Query().Get("url")  
        if url == "" {  
            http.Error(w, "URL is required", http.StatusBadRequest)  
            return  
        }  
  
        body, err := fetchURL(url)  
        if err != nil {  
            http.Error(w, err.Error(), http.StatusForbidden)  
            return  
        }  
  
        w.Write(body)  
    })  
}
```

```
    })

    http.ListenAndServe(":8080", nil)
}
```

In this code, the `isAllowedHost` function checks if the URL's host is in the list of allowed hosts, preventing requests to unauthorized locations.

## 2. Restrict IP Ranges

To add another layer of protection, you can restrict the IP ranges that the server is allowed to connect to. For example, you can prevent requests to internal IP addresses.

### IP Range Validation Example:

```
package main

import (
    "errors"
    "io/ioutil"
    "net"
    "net/http"
    "net/url"
    "strings"
)

var allowedHosts = []string{"example.com", "api.example.com"}

func isPrivateIP(ip net.IP) bool {
    privateIPBlocks := []*net.IPNet{
        {IP: net.IPv4(10, 0, 0, 0), Mask: net.CIDRMask(8, 32)},
        {IP: net.IPv4(172, 16, 0, 0), Mask: net.CIDRMask(12, 32)},
        {IP: net.IPv4(192, 168, 0, 0), Mask: net.CIDRMask(16, 32)},
        {IP: net.ParseIP("::1"), Mask: net.CIDRMask(128, 128)},
    }
```

COPY 

```

        {IP: net.ParseIP("fc00::"), Mask: net.CIDRMask(7, 128)},
        {IP: net.ParseIP("fe80::"), Mask: net.CIDRMask(10, 128)},
    }

    for _, block := range privateIPBlocks {
        if block.Contains(ip) {
            return true
        }
    }
    return false
}

func isAllowedHost(host string) bool {
    for _, allowedHost := range allowedHosts {
        if strings.Contains(host, allowedHost) {
            return true
        }
    }
    return false
}

func fetchURL(userURL string) ([]byte, error) {
    parsedURL, err := url.Parse(userURL)
    if err != nil {
        return nil, err
    }

    if !isAllowedHost(parsedURL.Host) {
        return nil, errors.New("host not allowed")
    }

    ips, err := net.LookupIP(parsedURL.Host)
    if err != nil {
        return nil, err
    }

    for _, ip := range ips {

```

```

        if isPrivateIP(ip) {
            return nil, errors.New("private IP not allowed")
        }
    }

    resp, err := http.Get(userURL)
    if err != nil {
        return nil, err
    }
    defer resp.Body.Close()

    return ioutil.ReadAll(resp.Body)
}

func main() {
    http.HandleFunc("/fetch", func(w http.ResponseWriter, r
    *http.Request) {
        url := r.URL.Query().Get("url")
        if url == "" {
            http.Error(w, "URL is required", http.StatusBadRequest)
            return
        }

        body, err := fetchURL(url)
        if err != nil {
            http.Error(w, err.Error(), http.StatusForbidden)
            return
        }

        w.Write(body)
    })

    http.ListenAndServe(":8080", nil)
}

```



In this example, the `isPrivateIP` function checks if the IP address is within a private range, preventing requests to internal network addresses.

## 8. File Upload

File upload vulnerabilities occur when an application allows users to upload files without proper validation and security checks. Attackers can exploit these vulnerabilities to upload malicious files (e.g., scripts or executables) that can compromise the server or other users' systems.

### *Vulnerable Code Example*

Here's a basic example of a file upload handler in Go that is vulnerable to attacks:

```
package main

import (
    "fmt"
    "io"
    "net/http"
    "os"
)

func uploadFileHandler(w http.ResponseWriter, r *http.Request) {
    if r.Method != "POST" {
        http.Error(w, "Method not allowed",
            http.StatusMethodNotAllowed)
        return
    }

    file, header, err := r.FormFile("file")
    if err != nil {
        http.Error(w, "Failed to retrieve file",
            http.StatusBadRequest)
        return
    }
}
```

COPY 

```

    }
    defer file.Close()

    // Save the file to disk
    destinationFile, err := os.Create("./uploads/" + header.Filename)
    if err != nil {
        http.Error(w, "Failed to create file on server",
http.StatusInternalServerError)
        return
    }
    defer destinationFile.Close()

    _, err = io.Copy(destinationFile, file)
    if err != nil {
        http.Error(w, "Failed to write file to disk",
http.StatusInternalServerError)
        return
    }

    fmt.Fprintf(w, "File uploaded successfully")
}

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

```

In this code:

- The `/upload` endpoint allows users to upload files using a POST request.
- It uses `r.FormFile("file")` to retrieve the uploaded file.
- The file is saved directly to the server's disk without any validation or checks.

## 1. Validate File Type and Content

Validate the file type and content before saving it to the server. This prevents users from uploading malicious files with unexpected content types or executable scripts.

### Secure Code Example:

COPY 

```
package main

import (
    "fmt"
    "io"
    "net/http"
    "os"
    "path/filepath"
)

func uploadFileHandler(w http.ResponseWriter, r *http.Request) {
    if r.Method != "POST" {
        http.Error(w, "Method not allowed",
            http.StatusMethodNotAllowed)
        return
    }

    file, header, err := r.FormFile("file")
    if err != nil {
        http.Error(w, "Failed to retrieve file",
            http.StatusBadRequest)
        return
    }
    defer file.Close()

    // Validate file extension
    ext := filepath.Ext(header.Filename)
    if ext != ".jpg" && ext != ".jpeg" && ext != ".png" {
        http.Error(w, "Only JPG, JPEG, and PNG files are allowed",
```

```

http.StatusBadRequest)
    return
}

// Save the file to disk
destinationFile, err := os.Create("./uploads/" + header.Filename)
if err != nil {
    http.Error(w, "Failed to create file on server",
http.StatusInternalServerError)
    return
}
defer destinationFile.Close()

_, err = io.Copy(destinationFile, file)
if err != nil {
    http.Error(w, "Failed to write file to disk",
http.StatusInternalServerError)
    return
}

fmt.Fprintf(w, "File uploaded successfully")
}

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

```

In this improved version:

- We validate the file extension ( `jpg` , `jpeg` , `png` ) before saving it.
- This prevents users from uploading files with potentially dangerous executable content.

## 2. Use Server-Side File Type Detection

Rather than relying solely on the client-provided filename and extension, perform server-side content-type detection to verify that the uploaded file matches its expected type.

### Secure Code Example using MIME type detection:

COPY 

```
package main

import (
    "fmt"
    "io"
    "net/http"
    "os"
    "path/filepath"
)

func uploadFileHandler(w http.ResponseWriter, r *http.Request) {
    if r.Method != "POST" {
        http.Error(w, "Method not allowed",
            http.StatusMethodNotAllowed)
        return
    }

    file, header, err := r.FormFile("file")
    if err != nil {
        http.Error(w, "Failed to retrieve file",
            http.StatusBadRequest)
        return
    }
    defer file.Close()

    // Detect file MIME type
    buffer := make([]byte, 512)
    _, err = file.Read(buffer)
    if err != nil {
        http.Error(w, "Failed to read file content",
```

```
http.StatusInternalServerError)
    return
}
fileType := http.DetectContentType(buffer)

// Validate MIME type
if fileType != "image/jpeg" && fileType != "image/png" {
    http.Error(w, "Only JPG and PNG images are allowed",
http.StatusBadRequest)
    return
}

// Save the file to disk
destinationFile, err := os.Create("./uploads/" + header.Filename)
if err != nil {
    http.Error(w, "Failed to create file on server",
http.StatusInternalServerError)
    return
}
defer destinationFile.Close()

_, err = io.Copy(destinationFile, file)
if err != nil {
    http.Error(w, "Failed to write file to disk",
http.StatusInternalServerError)
    return
}

fmt.Fprintf(w, "File uploaded successfully")
}

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

In this example, `http.DetectContentType` is used to determine the MIME type of the uploaded file content. This approach helps prevent users from uploading files with misleading extensions.

## 9. Memory Management Vulnerabilities

Memory management vulnerabilities in Go often arise from incorrect use of memory-related functions, leading to issues like buffer overflows, memory leaks, and use-after-free errors. These vulnerabilities can be exploited by attackers to execute arbitrary code, crash the application, or leak sensitive information.

1. **Buffer Overflows:** Occur when writing data beyond the allocated buffer size, potentially overwriting adjacent memory, causing crashes or executing malicious code.
2. **Memory Leaks:** Happen when memory allocated dynamically is not released after use, leading to exhaustion of system resources over time.
3. **Use-After-Free:** Occurs when accessing memory that has already been freed, potentially leading to crashes or execution of unintended code.

### *Vulnerable Code Example*

Here's an example of vulnerable code that could lead to a memory management issue:

```
package main

import (
    "fmt"
    "os"
)

func main() {
    // Allocate a slice with insufficient capacity
```

COPY 

```

slice := make([]int, 0, 5)

// Attempt to access memory outside slice capacity
for i := 0; i <= 5; i++ {
    slice[i] = i // Potential buffer overflow
}

fmt.Println("Slice:", slice)
}

```

In this code snippet:

- We allocate a slice `slice` with a length of `0` and capacity of `5`.
- The loop tries to access `slice[5]`, which is outside the allocated capacity (`index out of range` error).
- This could lead to a buffer overflow if `slice[i] = i` were to write beyond the allocated memory.

## *Secure Coding Practices*

### 1. Bounds Checking

Always perform bounds checking when accessing slices or arrays to ensure that the access is within the allocated memory range.

#### Secure Code Example:

```

package main

import (
    "fmt"
)

func main() {

```

COPY 



```
// Allocate a slice with sufficient capacity
slice := make([]int, 0, 5)

// Bounds-checked loop
for i := 0; i < 5; i++ {
    slice = append(slice, i) // Append to slice safely
}

fmt.Println("Slice:", slice)
}
```

In this corrected example:

- We use `append()` to safely add elements to `slice`.
- Bounds checking ensures that we don't access memory outside the allocated capacity.

## 2. Avoiding Memory Leaks

Properly release memory that is no longer needed to prevent memory leaks. Use `defer` statements to ensure resources are released even if errors occur.

### Secure Code Example:

```
package main

import (
    "fmt"
    "os"
)

func main() {
    file, err := os.Open("example.txt")
    if err != nil {
```

COPY 

```

        fmt.Println("Error opening file:", err)
        return
    }
    defer file.Close()

    // Read and process file content
    // ...

    // File will be closed automatically at function exit
}

```

In this example:

- `defer file.Close()` ensures that `file` is closed when the function exits, preventing memory leaks from unclosed file handles.

### 3. Use of Pointers and References

Be cautious when using pointers and references to avoid unintended use-after-free errors. Ensure that memory referenced by pointers remains valid throughout its intended usage.

#### Secure Code Example:

```

package main

import "fmt"

func main() {
    var ptr *int

    // Allocate memory for ptr
    ptr = new(int)

    // Use ptr safely

```

COPY 

```
*ptr = 10

fmt.Println("Value of ptr:", *ptr)

// Free memory after use
// delete(ptr) // delete is not necessary in Go, handled by
garbage collector
}
```

In this example:

- `new(int)` allocates memory for an integer and returns a pointer to it.
- The value is assigned to `*ptr` and used safely.
- Go's garbage collector handles memory deallocation, so `delete(ptr)` is unnecessary.

## 10. Cryptography Failure

Cryptography failures in Go typically involve incorrect implementation or misuse of hashing and encryption algorithms. These failures can lead to vulnerabilities such as data leaks, weak encryption, or susceptibility to attacks like brute force or chosen plaintext attacks.

1. **Weak Hashing Algorithms:** Use of insecure hash functions like MD5 or SHA-1, which are vulnerable to collisions and brute force attacks.
2. **Insecure Encryption Algorithms:** Incorrect use or weak encryption algorithms (e.g., ECB mode without proper padding) can lead to ciphertext manipulation and decryption vulnerabilities.
3. **Improper Key Management:** Inadequate key generation, storage, or sharing practices can compromise the confidentiality and integrity of encrypted data.

*Vulnerable Code Example*

Here's an example of vulnerable code that demonstrates improper use of encryption in Go:

COPY 

```
package main

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

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

    ciphertext := make([]byte, aes.BlockSize+len(plaintext))
    iv := ciphertext[:aes.BlockSize]
    if _, err := io.ReadFull(rand.Reader, iv); err != nil {
        return nil, err
    }

    mode := cipher.NewCBCEncrypter(block, iv)
    mode.CryptBlocks(ciphertext[aes.BlockSize:], plaintext)

    return ciphertext, nil
}

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

    mode := cipher.NewCBCDecrypter(block, ciphertext[:aes.BlockSize])
    mode.CryptBlocks(ciphertext[aes.BlockSize:], ciphertext[aes.BlockSize:])

    return ciphertext[aes.BlockSize:], nil
}
```

```

    iv := ciphertext[:aes.BlockSize]
    ciphertext = ciphertext[aes.BlockSize:]

    mode := cipher.NewCBCDecrypter(block, iv)
    mode.CryptBlocks(ciphertext, ciphertext)

    return ciphertext, nil
}

func main() {
    key := []byte("examplekey123456") // Insecure key generation

    plaintext := []byte("hello world")

    ciphertext, err := encrypt(key, plaintext)
    if err != nil {
        fmt.Println("Encryption error:", err)
        return
    }

    fmt.Printf("Ciphertext: %x\n", ciphertext)

    decrypted, err := decrypt(key, ciphertext)
    if err != nil {
        fmt.Println("Decryption error:", err)
        return
    }

    fmt.Println("Decrypted:", string(decrypted))
}

```

### Issues in the Code:

- **Insecure Key Generation:** The key `examplekey123456` is directly hardcoded, which is insecure for cryptographic purposes.

- **Improper Encryption Mode:** The code uses CBC mode without proper padding, which can lead to padding oracle attacks and plaintext recovery.

## *Secure Coding Practices*

### 1. Use Strong Hashing Algorithms

Use secure hash functions like SHA-256 or SHA-3 for hashing passwords and sensitive data. Here's an example of using SHA-256 in Go:

```
package main

import (
    "crypto/sha256"
    "encoding/hex"
    "fmt"
)

func main() {
    password := "mypassword123"

    hash := sha256.Sum256([]byte(password))
    hashedPassword := hex.EncodeToString(hash[:])

    fmt.Println("Hashed Password:", hashedPassword)
}
```

COPY 

### 2. Secure Encryption with AES-GCM

Use authenticated encryption modes like AES-GCM for secure encryption and decryption in Go:

```
package main
```

COPY 

```

import (
    "crypto/aes"
    "crypto/cipher"
    "crypto/rand"
    "fmt"
)

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

    ciphertext := make([]byte, aes.BlockSize+len(plaintext))
    iv := ciphertext[:aes.BlockSize]
    if _, err := rand.Read(iv); err != nil {
        return nil, err
    }

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

    encrypted := aead.Seal(nil, iv, plaintext, nil)
    ciphertext = append(iv, encrypted...)

    return ciphertext, nil
}

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

    if len(ciphertext) < aes.BlockSize {

```

```

        return nil, fmt.Errorf("ciphertext too short")
    }

    iv := ciphertext[:aes.BlockSize]
    ciphertext = ciphertext[aes.BlockSize:]

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

    decrypted, err := aead.Open(nil, iv, ciphertext, nil)
    if err != nil {
        return nil, err
    }

    return decrypted, nil
}

func main() {
    key := make([]byte, 32) // Generate a secure random key
    if _, err := rand.Read(key); err != nil {
        fmt.Println("Key generation error:", err)
        return
    }

    plaintext := []byte("hello world")

    ciphertext, err := encrypt(key, plaintext)
    if err != nil {
        fmt.Println("Encryption error:", err)
        return
    }

    fmt.Printf("Ciphertext: %x\n", ciphertext)

    decrypted, err := decrypt(key, ciphertext)

```



```

    if err != nil {
        fmt.Println("Decryption error:", err)
        return
    }

    fmt.Println("Decrypted:", string(decrypted))
}

```

## 11. LFI and RFI

LFI is a vulnerability that allows an attacker to include files that are already locally present on the server. This can lead to the disclosure of sensitive information or the execution of arbitrary code if the included file contains executable code.

**RFI (Remote File Inclusion):** RFI is a vulnerability where an attacker can include files from a remote server. This can result in the execution of arbitrary code from the attacker's server, potentially compromising the entire system.

### Common Vulnerabilities and Mitigations

#### 1. LFI Vulnerability Example:

```

package main

import (
    "fmt"
    "io/ioutil"
    "net/http"
)

func main() {
    http.HandleFunc("/view", func(w http.ResponseWriter, r
    *http.Request) {
        filename := r.URL.Query().Get("file")
        content, err := ioutil.ReadFile(filename)
    }
}

```

COPY 

```

        if err != nil {
            http.Error(w, "File not found", http.StatusNotFound)
            return
        }
        fmt.Fprintf(w, "%s", content)
    })

    http.ListenAndServe(":8080", nil)
}

```

- **Issue:** In the above example, an attacker could manipulate the `file` parameter in the URL to include sensitive files from the server's filesystem.

**Mitigation:** Always sanitize user inputs and validate file paths before using them. Restrict file access to only necessary directories.

- **RFI Vulnerability Example:**

```

package main

import (
    "fmt"
    "io/ioutil"
    "net/http"
)

func main() {
    http.HandleFunc("/include", func(w http.ResponseWriter, r
    *http.Request) {
        url := r.URL.Query().Get("url")
        resp, err := http.Get(url)
        if err != nil {
            http.Error(w, "Failed to fetch remote file",
            http.StatusInternalServerError)
            return
        }
    })
}

```

COPY 

```
        defer resp.Body.Close()
        body, err := ioutil.ReadAll(resp.Body)
        if err != nil {
            http.Error(w, "Failed to read remote file content",
http.StatusInternalServerError)
            return
        }
        fmt.Fprintf(w, "%s", body)
    })

    http.ListenAndServe(":8080", nil)
}
```

**Issue:** The `url` parameter allows an attacker to specify any remote file URL, potentially executing malicious code from a remote server.

**Mitigation:** Avoid including files dynamically from remote sources unless absolutely necessary. If required, validate and sanitize the URL input to ensure it only includes trusted sources.

## 12. Basic Authentication (BasicAuth) alongside JSON Web Tokens (JWT)

Using Basic Authentication (BasicAuth) alongside JSON Web Tokens (JWT) to manage user permissions in Golang is a common practice, but it can introduce security vulnerabilities if not implemented correctly. Let's discuss potential attacks and secure coding practices for this scenario.

### 1. Lack of HTTPS/TLS Encryption

**Issue:** Basic Authentication transmits credentials (username and password) in Base64-encoded format, which is not secure unless used over HTTPS/TLS. Without encryption, credentials can be intercepted via network sniffing.

**Secure Coding Practice:** Always enforce HTTPS/TLS for transmitting sensitive data like credentials. Golang provides easy integration with TLS using `ListenAndServeTLS` for HTTPS.

Example:

```
err := http.ListenAndServeTLS(":443", "server.crt", "server.key", nil)
if err != nil {
    log.Fatal("ListenAndServeTLS: ", err)
}
```

COPY 

## 2. Insufficient JWT Validation

**Issue:** JWTs are used for authorization after successful Basic Authentication. If not properly validated, forged or tampered JWTs can grant unauthorized access to protected resources.

**Secure Coding Practice:** Verify the JWT's authenticity, integrity, and expiry using a strong JWT library like [github.com/dgrijalva/jwt-go](https://github.com/dgrijalva/jwt-go). Validate the token's signature to ensure it hasn't been tampered with.

Example:

```
token, err := jwt.Parse(tokenString, func(token *jwt.Token)
(interface{}, error) {
    // Validate signing method and secret key
    if _, ok := token.Method.(*jwt.SigningMethodHMAC); !ok {
        return nil, fmt.Errorf("Unexpected signing method: %v",
token.Header["alg"])
    }
    return []byte("secret"), nil // Replace "secret" with your actual
secret key
})
```

COPY 

```
if err != nil {
    // Handle token validation error
    fmt.Println("Invalid token:", err)
    return
}

if token.Valid {
    // Token is valid; proceed with user authorization
} else {
    // Token is invalid; reject access
}
```

### 3. Lack of Rate Limiting

**Issue:** Without rate limiting, attackers can perform brute-force attacks on Basic Authentication credentials, potentially compromising user accounts.

**Secure Coding Practice:** Implement rate limiting to restrict the number of authentication attempts from a single IP address or user within a specified timeframe.

Example:

```
// Using github.com/juju/ratelimit for rate limiting
bucket := ratelimit.NewBucket(time.Minute, 100) // Allow 100 requests
per minute

http.HandleFunc("/login", func(w http.ResponseWriter, r *http.Request)
{
    if bucket.TakeAvailable(1) < 1 {
        http.Error(w, "Rate limit exceeded",
http.StatusTooManyRequests)
        return
    }
}
```

COPY 

```
// Handle authentication logic here  
})
```

## 13. Golang pitfalls

### 1. Unsafe Package Misuse

**Issue:** Golang's `unsafe` package allows direct manipulation of memory and pointers, which can lead to buffer overflows, memory corruption, and other vulnerabilities if used incorrectly.

**Secure Coding Practice:** Avoid using `unsafe` package unless absolutely necessary and ensure strict validation and bounds checking when manipulating memory.

Example:

```
package main  
  
import (  
    "fmt"  
    "unsafe"  
)  
  
func main() {  
    var x int32 = 10  
    ptr := unsafe.Pointer(&x)  
  
    // Incorrect usage, potentially unsafe  
    var y float64 = *(*float64)(ptr)  
  
    fmt.Println(y)  
}
```

COPY 

### 2. Insecure Random Number Generation

**Issue:** Using `math/rand` instead of `crypto/rand` for generating random numbers in security-sensitive contexts can lead to predictable outputs and cryptographic weaknesses.

**Secure Coding Practice:** Always use `crypto/rand` for generating random numbers, especially for cryptographic operations.

Example:

```
package main

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

func main() {
    // Secure random number generation
    n, err := rand.Int(rand.Reader, big.NewInt(100))
    if err != nil {
        fmt.Println("Error generating random number:", err)
        return
    }

    fmt.Println("Random number:", n)
}
```

COPY 

### 3. Improper Error Handling

**Issue:** Ignoring or mishandling errors returned by Golang functions (like file operations, network calls) can lead to unexpected behavior and security vulnerabilities.

**Secure Coding Practice:** Always check and handle errors appropriately to prevent panics or security breaches.

Example:

```
package main

import (
    "fmt"
    "os"
)

func main() {
    // Incorrect: ignoring error
    file, _ := os.Open("filename.txt")

    // Correct: handle error
    file, err := os.Open("filename.txt")
    if err != nil {
        fmt.Println("Error opening file:", err)
        return
    }

    // Process file
    defer file.Close()
}
```

COPY 

## 4. SQL Injection

**Issue:** Constructing SQL queries by concatenating strings without proper sanitization can lead to SQL injection attacks.

**Secure Coding Practice:** Use parameterized queries or ORM libraries like `gorm` to safely handle user input in SQL queries.



Example:

COPY 

```
package main

import (
    "database/sql"
    "fmt"
    _ "github.com/go-sql-driver/mysql"
)

func main() {
    db, err := sql.Open("mysql",
        "user:password@tcp(127.0.0.1:3306)/database")
    if err != nil {
        fmt.Println("Error connecting to database:", err)
        return
    }
    defer db.Close()

    // Incorrect: vulnerable to SQL injection
    username := "admin"
    query := fmt.Sprintf("SELECT * FROM users WHERE username='%s'",
        username)
    rows, err := db.Query(query)
    if err != nil {
        fmt.Println("Error executing query:", err)
        return
    }

    // Correct: use parameterized query
    query = "SELECT * FROM users WHERE username=?"
    rows, err = db.Query(query, username)
    if err != nil {
        fmt.Println("Error executing query:", err)
        return
    }
}
```

```
defer rows.Close()

for rows.Next() {
    var id int
    var username string
    err := rows.Scan(&id, &username)
    if err != nil {
        fmt.Println("Error scanning row:", err)
        return
    }
    fmt.Println("User:", username)
}
}
```

## 14. RPC

An attacker intercepts and alters communication between RPC client and server to eavesdrop or manipulate data.

**Mitigation:** Use TLS (Transport Layer Security) for encryption and authentication to secure RPC communication.

Example using `net/rpc` package with TLS:

```
package main

import (
    "crypto/tls"
    "fmt"
    "log"
    "net"
    "net/rpc"
)
```

COPY 

```
type Args struct {
    A, B int
}

type Arith int

func (t *Arith) Multiply(args *Args, reply *int) error {
    *reply = args.A * args.B
    return nil
}

func main() {
    arith := new(Arith)
    rpc.Register(arith)

    tlsConfig := &tls.Config{
        // Generate or use proper certificate files
        // Certificates should be signed by a trusted CA
        // For testing purposes, you can use self-signed certs
        InsecureSkipVerify: true, // should be false in production
    }
    listener, err := tls.Listen("tcp", ":1234", tlsConfig)
    if err != nil {
        log.Fatal("Listen error:", err)
    }

    fmt.Println("RPC server is listening on port 1234 ...")
    for {
        conn, err := listener.Accept()
        if err != nil {
            log.Fatal(err)
        }
        go rpc.ServeConn(conn)
    }
}
```

## 2. Denial-of-Service (DoS) Attack

**Description:** An attacker floods the RPC server with malicious requests, exhausting resources and causing service disruption.

**Mitigation:** Implement rate limiting, request validation, and timeouts to mitigate DoS attacks.

Example using `net/rpc` package with request validation:

```
package main

import (
    "fmt"
    "log"
    "net"
    "net/rpc"
)

type Args struct {
    A, B int
}

type Arith int

func (t *Arith) Multiply(args *Args, reply *int) error {
    if args.A > 1000 || args.B > 1000 {
        return fmt.Errorf("Values A and B must be less than or equal to 1000")
    }
    *reply = args.A * args.B
    return nil
}

func main() {
    arith := new(Arith)
```

COPY 

```
rpc.Register(arith)

listener, err := net.Listen("tcp", ":1234")
if err != nil {
    log.Fatal("Listen error:", err)
}

fmt.Println("RPC server is listening on port 1234 ...")
for {
    conn, err := listener.Accept()
    if err != nil {
        log.Fatal(err)
    }
    go rpc.ServeConn(conn)
}
}
```

## Mitigation

### 1. Use Secure Authentication and Authorization

**Description:** Validate and authorize RPC requests to prevent unauthorized access to sensitive functions or data.

**Practice:** Implement authentication mechanisms like JWT (JSON Web Token) or session-based authentication.

Example with JWT authentication:

```
package main

import (
    "fmt"
    "log"
    "net"
```

COPY 

```

    "net/http"
    "net/rpc"
    "strings"
)

type Args struct {
    A, B int
}

type Arith int

func (t *Arith) Multiply(args *Args, reply *int) error {
    *reply = args.A * args.B
    return nil
}

func main() {
    arith := new(Arith)
    rpc.Register(arith)

    http.HandleFunc("/rpc", func(w http.ResponseWriter, r
*http.Request) {
        // Extract and validate JWT token from request header
        token := r.Header.Get("Authorization")
        if !strings.HasPrefix(token, "Bearer ") {
            http.Error(w, "Unauthorized", http.StatusUnauthorized)
            return
        }
        // Validate and decode JWT token
        // Example code for JWT validation
        // Verify token validity, claims, etc.
        // Example: jwt.Parse(token)

        conn, _, err := w.(http.Hijacker).Hijack()
        if err != nil {
            log.Fatal("Hijack error:", err)
        }
    })
}

```

```
        defer conn.Close()

        rpc.ServeConn(conn)
    })

    fmt.Println("RPC server with JWT authentication is listening on
port 8080 ...")
    log.Fatal(http.ListenAndServe(":8080", nil))
}
```

## 2. Validate and Sanitize Inputs

**Description:** Validate and sanitize all inputs to prevent injection attacks or unexpected behavior.

**Practice:** Use parameterized RPC methods or validate inputs rigorously.

Example with input validation:

```
package main

import (
    "fmt"
    "log"
    "net"
    "net/rpc"
    "strconv"
)

type Args struct {
    A, B int
}

type Arith int
```

COPY 

```

func (t *Arith) Multiply(args *Args, reply *int) error {
    if args.A > 1000 || args.B > 1000 {
        return fmt.Errorf("Values A and B must be less than or equal
to 1000")
    }
    *reply = args.A * args.B
    return nil
}

func main() {
    arith := new(Arith)
    rpc.Register(arith)

    listener, err := net.Listen("tcp", ":1234")
    if err != nil {
        log.Fatal("Listen error:", err)
    }

    fmt.Println("RPC server with input validation is listening on port
1234 ...")
    for {
        conn, err := listener.Accept()
        if err != nil {
            log.Fatal(err)
        }
        go rpc.ServeConn(conn)
    }
}

```

## 15. Timing Attack

In Golang, timing attacks can occur when performing cryptographic comparisons using functions like `hmac.Equal`. This function is commonly used to compare two HMACs (Hash-based Message Authentication Codes), which are used for verifying the integrity and authenticity of messages.



Here's an example of using `hmac.Equal` in Golang:

COPY 

```
package main

import (
    "crypto/hmac"
    "crypto/sha256"
    "fmt"
    "time"
)

func main() {
    key := []byte("secret-key")
    message1 := []byte("Hello, world!")
    message2 := []byte("Hello, world?")

    start := time.Now()
    mac1 := hmac.New(sha256.New, key)
    mac1.Write(message1)
    expectedMac := mac1.Sum(nil)
    fmt.Println("Time taken for mac1:", time.Since(start))

    start = time.Now()
    mac2 := hmac.New(sha256.New, key)
    mac2.Write(message2)
    actualMac := mac2.Sum(nil)
    fmt.Println("Time taken for mac2:", time.Since(start))

    // Compare HMACs using hmac.Equal (vulnerable to timing attack)
    start = time.Now()
    isValid := hmac.Equal(expectedMac, actualMac)
    fmt.Println("Time taken for comparison:", time.Since(start))

    fmt.Println("Are the HMACs equal?", isValid)
}
```

To mitigate timing attacks in Golang and similar languages, consider the following secure coding practices:

## 1. Constant-Time Comparison

Use constant-time comparison functions instead of `hmac.Equal` for sensitive comparisons. Golang's `crypto/subtle` package provides a `ConstantTimeCompare` function for this purpose:

```
import "crypto/subtle"

isValid := subtle.ConstantTimeCompare(mac1, mac2) == 1
```

COPY 

Here's how you can implement a secure comparison function in Golang using `crypto/subtle`:

```
package main

import (
    "crypto/subtle"
    "fmt"
)

func secureCompare(x, y []byte) bool {
    return subtle.ConstantTimeCompare(x, y) == 1
}

func main() {
    expectedMac := []byte("expected-mac")
    actualMac := []byte("actual-mac")

    isValid := secureCompare(expectedMac, actualMac)
```

COPY 

```
fmt.Println("Are the MACs equal?", isValid)
```

```
}
```

## References

- <https://github.com/OWASP/Go-SCP>
- Security with Go by John Daniel Leon
- [https://blog.csdn.net/weixin\\_30706691/article/details/97829172?ops\\_request\\_misc=%257B%2522request%255Fid%2522%253A%2522171920807716800184137706%2522%252C%2522scm%2522%253A%252220140713.130102334.pc%255Fall.%2522%257D&request\\_id=171920807716800184137706&biz\\_id=0&utm\\_medium=distribute.pc\\_search\\_result.none-task-blog-2~all~first\\_rank\\_ecpm\\_v1~rank\\_v31\\_ecpm-25-97829172-null-null.142^v100^pc\\_search\\_result\\_base9&utm\\_term=golang%20attacks&spm=1018.2226.3001.4187](https://blog.csdn.net/weixin_30706691/article/details/97829172?ops_request_misc=%257B%2522request%255Fid%2522%253A%2522171920807716800184137706%2522%252C%2522scm%2522%253A%252220140713.130102334.pc%255Fall.%2522%257D&request_id=171920807716800184137706&biz_id=0&utm_medium=distribute.pc_search_result.none-task-blog-2~all~first_rank_ecpm_v1~rank_v31_ecpm-25-97829172-null-null.142^v100^pc_search_result_base9&utm_term=golang%20attacks&spm=1018.2226.3001.4187)

## Subscribe to our newsletter

Read articles from **DevSecOpsGuides** directly inside your inbox. Subscribe to the newsletter, and don't miss out.

**SUBSCRIBE**

golang

Go Language

Devops

DevSecOps


appsec

secure coding

Written by



Reza Rashidi

 Add your bio

Published on



DevSecOpsGuides

 Add blog description

## MORE ARTICLES

 **Reza Rashidi**



### Ansible Playbooks

Ansible playbooks are essential tools in the DevSecOps toolkit, enabling the automation of complex I...

 **Reza Rashidi**



### eBPF cheatsheet

eBPF (Extended Berkeley Packet Filter) is a powerful technology for monitoring and analyzing system ...

 **Reza Rashidi**



### DevSecOps Security Architecture

In the rapidly evolving landscape of cybersecurity, DevSecOps Security Architecture emerges as a cri...

©2024 DevSecOpsGuides

[Archive](#) · [Privacy\\_policy](#) · [Terms](#)



Write on Hashnode

Powered by [Hashnode](#) - Home for tech writers and readers