Fortify Security Report

11/7/23

Demo User

Executive Summary

Issues Overview

On 7 Nov 2023, a source code review was performed over the govwa code base. 103 files, 583 LOC (Executable) were scanned and reviewed for defects that could lead to potential security vulnerabilities. A total of 41 reviewed findings were uncovered during the analysis.

Issues by Fortify Priority Order	
Low	21
Medium	10
High	7
Critical	3

Recommendations and Conclusions

The Issues Category section provides Fortify recommendations for addressing issues at a generic level. The recommendations for specific fixes can be extrapolated from those generic recommendations by the development group.

Project Summary

Code Base Summary

Code location: C:/Users/klee/source/repos/govwa
Number of Files: 103
Lines of Code: 583
Build Label: SNAPSHOT

Scan Information

Scan time: 00:46
SCA Engine version: 23.2.0.0125
Machine Name: GBKLEE02
Username running scan: klee2

Results Certification

Results Certification Valid

Details:

Results Signature:

SCA Analysis Results has Valid signature

Rules Signature:

There were no custom rules used in this scan

Attack Surface

Attack Surface:
File System:
io.ioutil.null.ReadFile

Filter Set Summary

Current Enabled Filter Set:
Security Auditor View

Filter Set Details:

Folder Filters:
If [fortify priority order] contains critical Then set folder to Critical
If [fortify priority order] contains high Then set folder to High
If [fortify priority order] contains medium Then set folder to Medium
If [fortify priority order] contains low Then set folder to Low

Audit Guide Summary

Audit guide not enabled

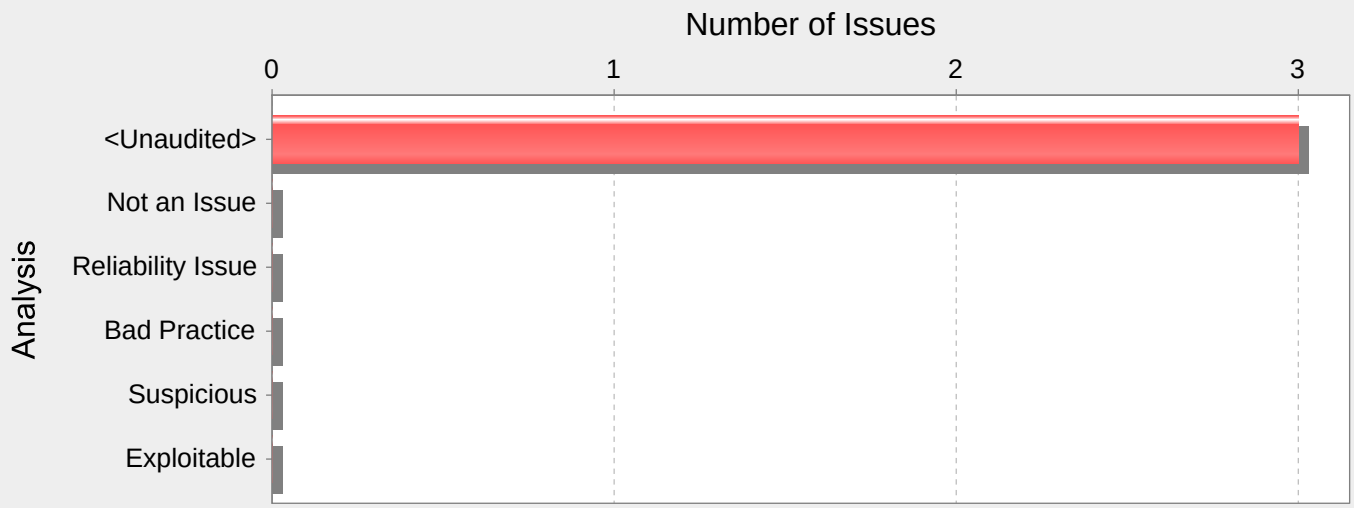
Results Outline

Overall number of results

The scan found 41 issues.

Vulnerability Examples by Category

Category: SQL Injection (3 Issues)



Abstract:

Line 24 of database.go invokes a SQL query built with input that comes from an untrusted source. This call could enable an attacker to modify the statement's meaning or to execute arbitrary SQL commands.

Explanation:

SQL injection errors occur when:

1. Data enters a program from an untrusted source.

2. The data is used to dynamically construct a SQL query.

Example 1: The following code dynamically constructs and executes a SQL query that searches for items matching a specified name. The query restricts the items displayed to those where the owner matches the user name of the currently-authenticated user.

```
...
rawQuery := request.URL.Query()
username := rawQuery.Get("userName")
itemName := rawQuery.Get("itemName")
query := "SELECT * FROM items WHERE owner = " + username + " AND itemname = " + itemName + ";"
db.Exec(query)
...
```

The query intends to execute the following code:

```
SELECT * FROM items
WHERE owner = <userName>
AND itemname = <itemName>;
```

However, because the code dynamically constructs the query by concatenating a constant base query string and a user input string, the query only behaves correctly if itemName does not contain a single-quote character. If an attacker with the user name wiley enters the string "name' OR 'a'='a" for itemName, then the query becomes the following:

```
SELECT * FROM items
WHERE owner = 'wiley'
AND itemname = 'name' OR 'a'='a';
```

The addition of the OR 'a'='a' condition causes the where clause to always evaluate to true, so the query becomes logically equivalent to the much simpler query:

SELECT * FROM items;

This simplification of the query enables the attacker to bypass the requirement that the query must only return items owned by the authenticated user. The query now returns all entries stored in the items table, regardless of their specified owner.

Example 2: This example examines the effects of a different malicious value passed to the query constructed and executed in Example 1. If an attacker with the user name wiley enters the string "name'; DELETE FROM items; --" for itemName, then the query becomes the following two queries:

SELECT * FROM items
WHERE owner = 'wiley'
AND itemname = 'name';

DELETE FROM items;

--'

Many database servers, including Microsoft(R) SQL Server 2000, allow the simultaneous execution of multiple SQL statements separated by semicolons. While this attack string results in an error on Oracle and other database servers that do not allow the batch-execution of statements separated by semicolons, on databases that do allow batch execution, this type of attack enables the attacker to execute arbitrary commands against the database.

Notice the trailing pair of hyphens (--), which specifies to most database servers to treat the remainder of the statement as a comment and to not execute it. [4]. In this case, the comment character serves to remove the trailing single-quote left over from the modified query. On a database where comments are not allowed to be used in this way, the general attack could still be made effective using a trick similar to the one shown in Example 1. If an attacker enters the string "name'; DELETE FROM items; SELECT * FROM items WHERE 'a'='a'", the following three valid statements are created:

SELECT * FROM items
WHERE owner = 'wiley'
AND itemname = 'name';

DELETE FROM items;

SELECT * FROM items WHERE 'a'='a';

One traditional approach to prevent SQL injection attacks is to handle them as an input validation problem and either accept only characters from an allow list of safe values or identify and escape a list of potentially malicious values (deny list). Checking an allow list can be a very effective means of enforcing strict input validation rules, but parameterized SQL statements require less maintenance and can offer more guarantees with respect to security. As is almost always the case, implementing a deny list is riddled with loopholes that make it ineffective to prevent SQL injection attacks. For example, attackers can:

- Target fields that are not quoted
- Find ways to bypass the need for certain escaped metacharacters
- Use stored procedures to hide the injected metacharacters

Manually escaping characters in input to SQL queries can help, but it does not make your application secure from SQL injection attacks.

Another solution commonly proposed to deal with SQL injection attacks is to use stored procedures. Although stored procedures prevent some types of SQL injection attacks, they fail to protect against many others. Stored procedures typically help prevent SQL injection attacks by limiting the types of statements that can be passed to their parameters. However, there are many ways around the limitations and many interesting statements that can still be passed to stored procedures. Again, stored procedures can prevent some exploits, but they do not make your application secure against SQL injection attacks.

Recommendations:

The root cause of a SQL injection vulnerability is the ability of an attacker to change context in the SQL query, causing a value that the programmer intended to be interpreted as data to be interpreted as a command instead. When a SQL query is constructed, the programmer knows what should be interpreted as part of the command and what should be interpreted as data. Parameterized SQL statements can enforce this behavior by disallowing data-directed context changes and preventing nearly all SQL injection attacks. Parameterized SQL statements are constructed using strings of regular SQL, but where user-supplied data needs to be included, they include bind parameters, which are placeholders for data that is subsequently inserted. In other words, bind parameters allow the programmer to explicitly specify to the database what should be treated as a command and what should be treated as data. When the program is ready to execute a statement, it specifies to the database the runtime values to use for each of the bind parameters without the risk that the data is interpreted as a modification to the command.

The following is a rewrite of the previous example to use parameterized SQL statements (instead of concatenating user-supplied strings) as follows:

...
rawQuery := request.URL.Query()
username := rawQuery.Get("userName")
itemName := rawQuery.Get("itemName")

...
db.Exec("SELECT * FROM items WHERE owner = ? AND itemname = ?", username, itemName)
...

More complicated scenarios, often found in report generation code, require that user input affect the structure of the SQL statement, for instance by adding a dynamic constraint in the WHERE clause. Do not use this requirement to justify concatenating user input to create a query string. Prevent SQL injection attacks where user input must affect command structure with a level of indirection: create a set of legitimate strings that correspond to different elements you might include in a SQL statement. When constructing a statement, use input from the user to select from this set of application-controlled values.

Tips:

1. A common mistake is to use parameterized SQL statements that are constructed by concatenating user-controlled strings. Of course, this defeats the purpose of using parameterized SQL statements. If you are not certain that the strings used to form parameterized statements are constants controlled by the application, do not assume that they are safe because they are not being executed directly as SQL strings. Thoroughly investigate all uses of user-controlled strings in SQL statements and verify that none can be used to modify the meaning of the query.

2. The SQL Injection issue might still be reported (with a reduced priority value) even after dataflow remediation efforts. When Fortify Static Code Analyzer finds clear dataflow evidence of user-controlled input being used to construct SQL statements, a high/critical priority dataflow issue is reported. When Fortify Static Code Analyzer cannot determine the source of the data but it can be dynamically changed, a low/medium priority semantic issue is reported. This strategy is adopted in a few select vulnerability categories such as SQL Injection where the potential impact of exploitation outweighs the inconvenience of auditing false positive issues.

database.go, line 24 (SQL Injection)

Fortify Priority:	High	**Folder**	High
Kingdom:	Input Validation and Representation		
Abstract:	Line 24 of database.go invokes a SQL query built with input that comes from an untrusted source. This call could enable an attacker to modify the statement's meaning or to execute arbitrary SQL commands.		

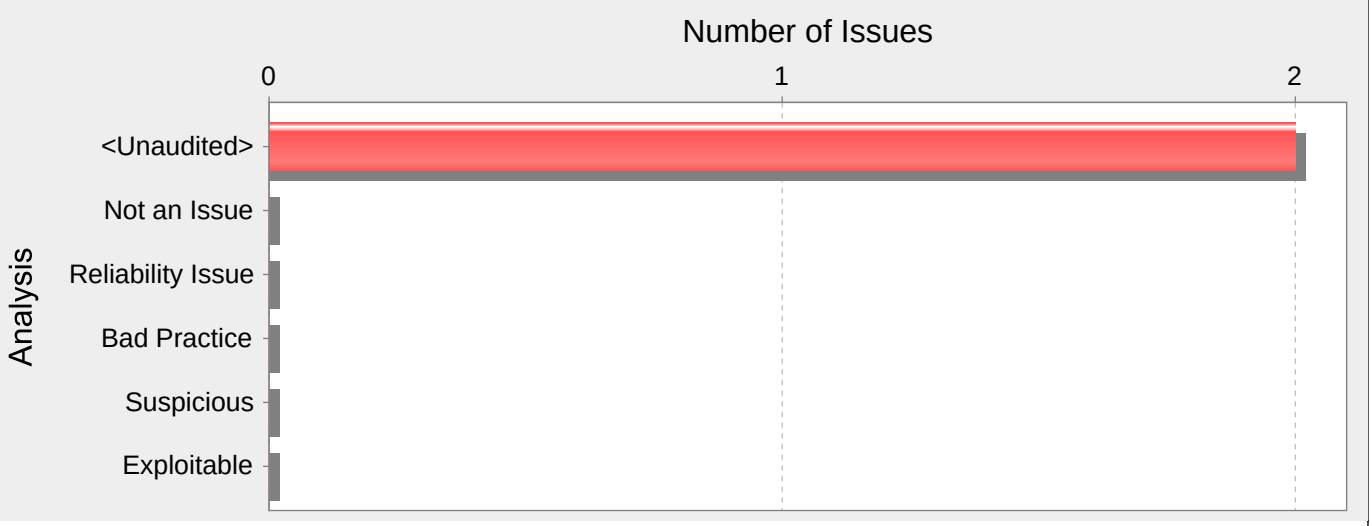
Source: config.go:24 io.ioutil.ReadFile()

```
22
23  func LoadConfig()*Config{
24   raw, err := ioutil.ReadFile("config/config.json")
25   if err != nil{
26    fmt.Println(err.Error())
```

Sink: database.go:24 database.sql.DB.Exec()

```
22    return nil, err
23   }
24   _, err = db.Exec("CREATE DATABASE IF NOT EXISTS " + config.Dbname)
25
26   if err != nil {
```

Category: Connection String Parameter Pollution (2 Issues)



Abstract:

The file database.go passes unvalidated data to a database connection string on line 19. An attacker might be able to override existing parameter values, inject a new parameter, or exploit variables that are out of direct reach.

Explanation:

Connection String Parameter Pollution (CSPP) attacks consist of injecting connection string parameters into other existing parameters. This vulnerability is similar to vulnerabilities, and perhaps more well known, within HTTP environments where parameter pollution can also occur. However, it also can apply in other places such as database connection strings. If an application does not properly sanitize the user input, a malicious user may compromise the logic of the application to perform attacks from stealing credentials, to retrieving the entire database. By submitting additional parameters that have the same name as an existing parameter to an application, the database might react in one of the following ways:

It might only take the data from the first parameter

It might take the data from the last parameter

It might take the data from all parameters and concatenate them together

This is dependent on the driver used, the database type, or even how APIs are used.

Example 1: The following code uses input from an HTTP request to connect to a database:

```
...
password := request.FormValue("db_pass")
db, err := sql.Open("mysql", "user:" + password + "@/dbname")
...
```

In this example, the programmer has not considered that an attacker could provide a db_pass parameter such as:
"xxx@/attackerdb?foo=" then connection string becomes:

"user:xxx@/attackerdb?foo=/dbname"

This will make the application connect to an attacker controller database enabling him to control which data is return to the application.

Recommendations:

Checking an allow list of acceptable characters is recommended. Alternatively, you can use APIs (eg: mysql.Config.FormatDSN) to specify connection string values that wrap the connection parameters and prevent this type of attack.

Example 2: The following code parametrizes input from an HTTP request to connect to a database:

```
...
password := request.FormValue("db_pass")
config := mysql.Config{User: "user", Passwd: password, DBName: "dbname"}
db, err := sql.Open("mysql", config.FormatDSN())
...
```

database.go, line 19 (Connection String Parameter Pollution)

Fortify Priority:	High	Folder	High

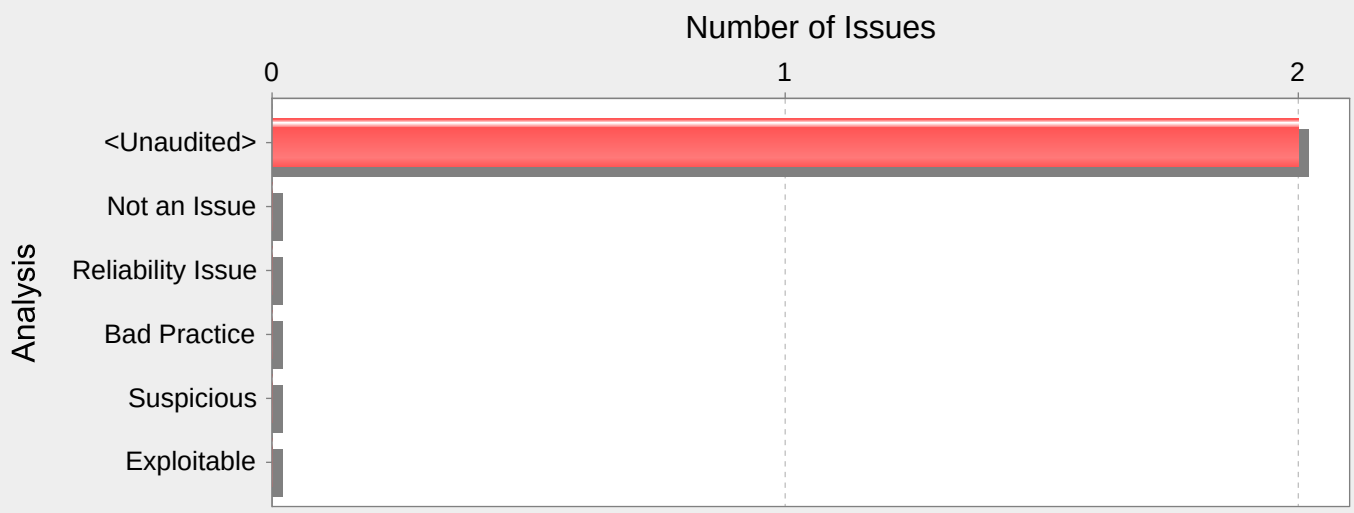
Kingdom:	Input Validation and Representation
Abstract:	The file database.go passes unvalidated data to a database connection string on line 19. An attacker might be able to override existing parameter values, inject a new parameter, or exploit variables that are out of direct reach.
Source:	config.go:24 io.ioutil.ReadFile()

```
22
23  func LoadConfig()*Config{
24   raw, err := ioutil.ReadFile("config/config.json")
25   if err != nil{
26    fmt.Println(err.Error())
```

Sink: database.go:19 database.sql.Open()

```
17
18   dsn = fmt.Sprintf("%s:%s@tcp(%s:%s)/", config.User, config.Password, config.Sqlhost,
     config.Sqlport)
19   db, err := sql.Open("mysql", dsn)
20
21   if err != nil {
```

Category: Cookie Security: HTTPOnly not Set (2 Issues)



Abstract:

The program creates a cookie in cookie.go on line 32, but fails to set the HttpOnly flag to true.

Explanation:

Browsers support the HttpOnly cookie property that prevents client-side scripts from accessing the cookie. Cross-site scripting attacks often access cookies in an attempt to steal session identifiers or authentication tokens. Without HttpOnly enabled, attackers have easier access to user cookies.

Example 1: The following code creates a cookie without setting the HttpOnly property.

```
cookie := http.Cookie{
Name:    "emailCookie",
Value:   email,
}
...
```

Recommendations:

Enable the HttpOnly property when you create cookies. Do this by setting the HttpOnly parameter in the SetCookie() call to true.

Example 2: The following code creates the same cookie as the code in Example 1, but this time sets the HttpOnly parameter to true.

```
cookie := http.Cookie{
Name:     "emailCookie",
Value:    email,
HttpOnly:  true,
}
...
```

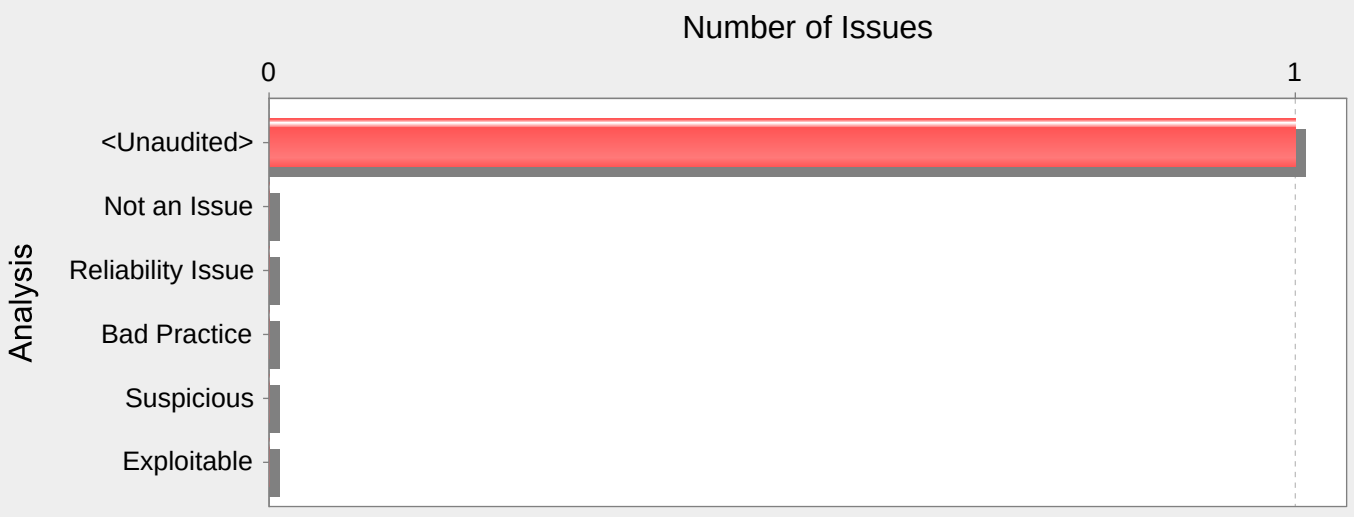
Do not be lulled into a false sense of security with HttpOnly. Several mechanisms for bypassing this security feature have been developed, so it is not always effective.

cookie.go, line 32 (Cookie Security: HTTPOnly not Set)

Fortify Priority:	High	**Folder**	High
Kingdom:	Security Features		
Abstract:	The program creates a cookie in cookie.go on line 32, but fails to set the HttpOnly flag to true.		
Sink:	cookie.go:32 AssignmentStatement()		

```
30
31  func SetCookie(w http.ResponseWriter, name, value string){
32   cookie := http.Cookie{
33    //Path : "/",
34    //Domain : "localhost",
```

Category: Dockerfile Misconfiguration: Default User Privilege (1 Issues)



Abstract:

The Dockerfile does not specify a USER, so it defaults to running with a root user.

Explanation:

When a Dockerfile does not specify a USER, Docker containers run with super user privileges by default. These super user privileges are propagated to the code running inside the container, which is usually more permission than necessary. Running the Docker container with super user privileges broadens the attack surface which might enable attackers to perform more serious forms of exploitation.

Recommendations:

It is good practice to run your containers as a non-root user when possible.

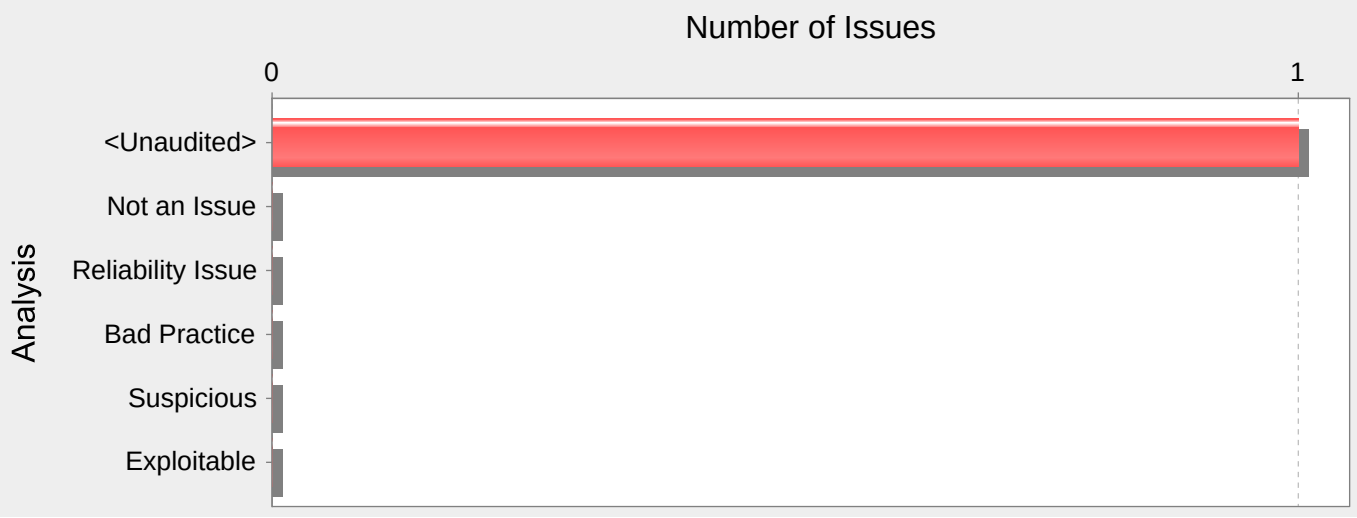
To modify a docker container to use a non-root user, the Dockerfile needs to specify a different user, such as:

RUN useradd myLowPrivilegeUser
USER myLowPrivilegeUser

Dockerfile, line 1 (Dockerfile Misconfiguration: Default User Privilege)

Fortify Priority:	High	**Folder**	High
Kingdom:	Environment		
Abstract:	The Dockerfile does not specify a USER, so it defaults to running with a root user.		
Sink:	Dockerfile:1 FROM()		
-1	`FROM golang:alpine AS builder`		
0			
1	`# Set necessary environmet variables needed for our image`		

Category: Insecure Transport (1 Issues)



Abstract:

The call to ListenAndServe() in app.go on line 79 uses an unencrypted protocol instead of an encrypted protocol to communicate with the server.

Explanation:

All communication over HTTP, FTP, or gopher is unauthenticated and unencrypted. It is therefore subject to compromise, especially in environments where devices frequently connect to unsecured public wireless networks.

Example 1: The following example sets up a Web server using the HTTP protocol (instead of using HTTPS).

```
helloHandler := func(w http.ResponseWriter, req *http.Request) {
io.WriteString(w, "Hello, world!\n")
}

http.HandleFunc("/hello", helloHandler)
log.Fatal(http.ListenAndServe(":8080", nil))
```

Recommendations:

Use secure protocols such as HTTPS to exchange data with the server whenever possible.

Example 2: The following example sets up a Web server using the HTTPS protocol.

```
http.HandleFunc("/", func(w http.ResponseWriter, req *http.Request) {
io.WriteString(w, "Hello, TLS!\n")
})

log.Fatal(http.ListenAndServeTLS(":8443", "cert.pem", "key.pem", nil))
```

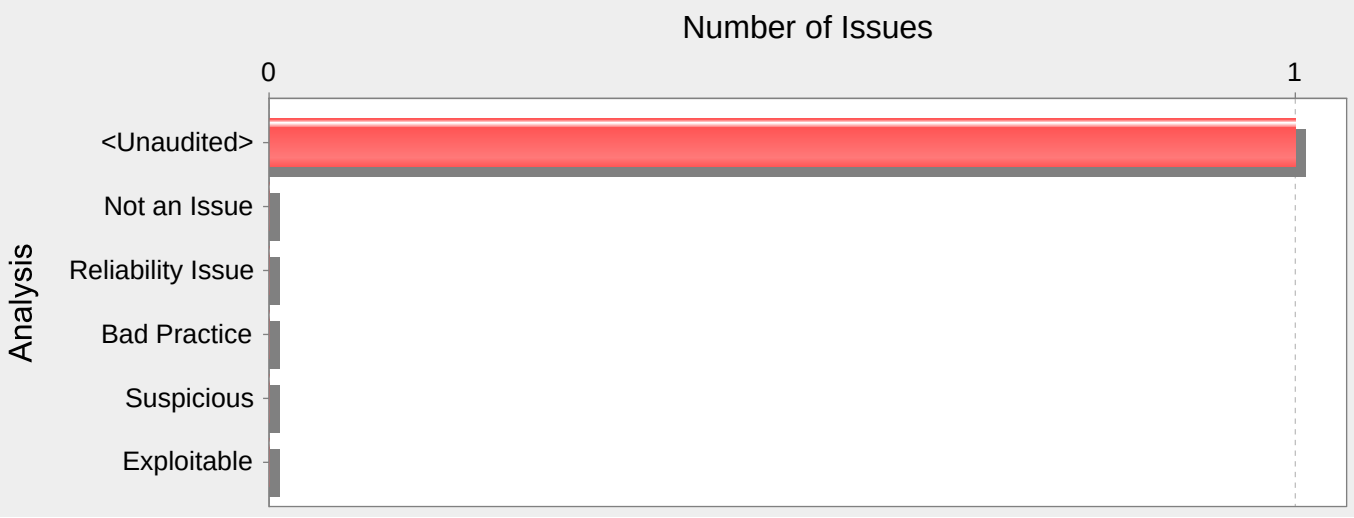
app.go, line 79 (Insecure Transport)

Fortify Priority:	Critical	**Folder**	Critical
Kingdom:	Security Features		
Abstract:	The call to ListenAndServe() in app.go on line 79 uses an unencrypted protocol instead of an encrypted protocol to communicate with the server.		

Sink: app.go:79 ListenAndServe()

```
77    fmt.Printf("Open this url %s on your browser to access GoVWA", config.Fullurl)
78    fmt.Println("")
79    err := s.ListenAndServe()
80    if err != nil {
81     panic(err)
```

Category: Privacy Violation: Autocomplete (1 Issues)



Abstract:

The form in template.login.html uses autocompletion on line 35, which allows some browsers to retain sensitive information in their history.

Explanation:

With autocompletion enabled, some browsers retain user input across sessions, which could allow someone using the computer after the initial user to see information previously submitted.

Recommendations:

Explicitly disable autocompletion on forms or sensitive inputs. By disabling autocompletion, information previously entered will not be presented back to the user as they type. It will also disable the "remember my password" functionality of most major browsers.

Example 1: In an HTML form, disable autocompletion for all input fields by explicitly setting the value of the autocomplete attribute to off on the form tag.

```
<form method="post" autocomplete="off">
Address: <input name="address" />
Password: <input name="password" type="password" />
</form>
```

Example 2: Alternatively, disable autocompletion for specific input fields by explicitly setting the value of the autocomplete attribute to off on the corresponding tags.

```
<form method="post">
Address: <input name="address" />
Password: <input name="password" type="password" autocomplete="off"/>
</form>
```

Note that the default value of the autocomplete attributed is on. Therefore do not omit the attribute when dealing with sensitive inputs.

template.login.html, line 35 (Privacy Violation: Autocomplete)

Fortify Priority:	High	**Folder**	High
Kingdom:	Security Features		
Abstract:	The form in template.login.html uses autocompletion on line 35, which allows some browsers to retain sensitive information in their history.		

Sink: template.login.html:35

```
33                                      </div>
34                                      <div class="form-group">
35                                          <input type="password" name="password" value=""
class="form-control" placeholder="Password" />
36                                      </div>
37                                      <input type="submit" name="submit" value="Log in"
class="btn btn-success btn-lg btn-block" />
```

Issue Count by Category

Issues by Category	
Cross-Site Request Forgery	8
Insecure Transport: External Link	5
Hidden Field	4
Log Forging	3
SQL Injection	3
Weak Cryptographic Hash	3
Connection String Parameter Pollution	2
Cookie Security: Cookie not Sent Over SSL	2
Cookie Security: HTTPOnly not Set	2
Cookie Security: Missing SameSite Attribute	2
Hardcoded Domain in HTML	2
Header Manipulation: Cookies	2
Dockerfile Misconfiguration: Default User Privilege	1
Insecure Transport	1
Privacy Violation: Autocomplete	1

Issue Breakdown by Analysis

Issues by Analysis