John Murray | William Tiffany

G00152680 G00\*\*\*\*\*\*

GO Online Web Application Game

isa/SWE 681\_ software assurance ASSESSMENT report



# Contents

1. INTRODUCTION
2. GO SOFTWARE GAME DESIGN
   1. WEB APPLICATION ARCHITECTURE
      1. Software Components
         1. SQLite Database Server
         2. GO MVC Servlets
         3. HTML DOM Selection Objects
         4. AngularJS Package Manager
         5. Utility
   2. GAME OPTIONS MENU
      1. Login / Start Game / Check Leaderboard Scores / Logout
      2. Begin New Game or Complete Previously Unfinished Game
      3. Play Competitive (1vs1) GO
      4. View Leaderboard Scores
      5. View Game Manuel (How-to-Play) Pages
   3. GO README
      1. Installation Instructions
      2. User/Player Game Rules
      3. Game Server Logic
3. GO SOFTWARE ASSURANCE ASSESSMENT
   1. Software Assurance Case Report
   2. Secure Sockets Library Imports (OpenSSL/TLS)
   3. WebApp Security Protocol Configurations
   4. Countermeasures to Common WebApp Vulnerabilities
      1. Protection Against Cross Site Scripting (XSS)
      2. Protection Against Cross Site Request Forgery (CSRF)
         1. User Input Validation
         2. Regular Expressions (Regex)
         3. Data Cleansing & Sanitation
         4. Proper Escaping of Meta-characters
      3. Protection Against SQL-Injection
   5. Protection Against Low-level Systems Attacks
      1. Safeguards to Buffer Overflow / OpenSSL Heartbleed
      2. Safeguards to Format String Vulnerability
      3. Safeguards to Heap Overflow & Spraying
      4. Safeguards to Dangler Pointers (Double Frees & Use After Frees)
      5. Safeguards to Data Type Confusion
      6. Safeguards to File Path Manipulation
   6. Data Protection
      1. Logging of Sensitive Data
      2. Data Encryption & Confidentiality
      3. User Privacy / Anonymity & Password Protection
      4. Ensuring Availability & Protection Against Login Failure
   7. Access Control Protection Policy
      1. User Authentication & Session Management
         1. Safeguards to Man-in-the-Middle (MITM) Attacks
         2. Safeguards to Reply Attacks
      2. Principal of Least Privilege
      3. Protection Against Malicious Administrator Backdoors
   8. Game Event and Score Logging
      1. Audit Trails
      2. Logging
   9. Version Control System
      1. Cache Control
      2. Gameplay Rule Validations
      3. OWASP ZAP Testing
         1. Standard Mode
         2. Attacker Attribution
4. CONCLUSION
5. REFERENCES

# **INTRODUCTION**

GO is one of the World’s oldest and simplest games. Our implementation of the GO web application (WebApp) involves two players connecting to a single online game board session instance supported by a server over a secure HTTP/TLS connection with each player playing one side of the game remotely. The objective of GO is the capture the most tokens on the board thus gaining the highest percentage of points to win the game. The game ends when there are no more entries left on the board to place a token. When the game ends the player with the highest amount of points will be entered as the winner onto the virtual Leaderboard. Should either player quit the game prematurely even despite having a higher overall score the server will still recognize this as a forfeit and the player that did not quit will automatically be deemed the winner. If the game should end due to an outside unforeseen circumstance (ie. loss of Internet connectivity) whichever player maintained the highest score at the time of connection disruption will be deemed the winner.

# **GO SOFTWARE GAME DESIGN**

The GO online web application game was written in the GOLang programming language developed by Google which is highly optimized toward concurrency. This chapter will outline the Software Development Lifecycle (SDLC) we employed to develop the game functionality, logic and security.

## WEB APPLICATION ARCHITECTURE

Our online version of GO employs a client server-based architecture which was modeled using the \*\*\* framework. The server was implemented using the SQLite relational database management system (RDBMS) with Data Access Object (DAO). Our web application servlets were implemented using the Model View Control (MVC), Front Controller and Business Delegate design patterns. For the client-side web application interface we utilized HTML Document Object Model (DOM) Selection Objects that include event bubbling, targeting and capturing. For web application deployment we utilized AngularJS package manager to host the client-side command line interface onto port 8080.

## SOFTWARE COMPONENTS

### SQLite RDBDMS

### GO MVC Servlets

### HTML DOM Selection Objects

### AngularJS Package Manager

### Utility

## GO GAME OPTIONS MENU

## Login / Start Game / Logout

## Begin New Game or Complete Previously Unfinished Game

## Play Competitive (1vs1) GO

## View Leader-board Scores

## View Game Manuel (How-to-Play) Pages

## GO README

## Installation Instructions

## User/Player Game Rules

Go is an ancient territorial game for 2 players each placing different types of tokens onto a symmetric game board or matrix. The objective of GO is to secure territory, but also involves the capture of enemy stones, which are added to territory gains.

Once the GO game has started players will be presented with a simple command line interface and game board matrix where player moves are token entries placed on the gameboard and entered via array indexes of both X and Y coordinates planes. Assuming the player token entry (game move) is legal the game token will be marked as an entry onto the gameboard and control will then switch to the opponent to allow for a corresponding follow-up move.

Go begins with an empty board. Player 1 enters moves first and moves then alternate between Player 1 and Player 2. When both players pass consecutively the game is over and the score is based on the resulting final board position. The player deemed the winner is the one who holds the greatest amount of territory.

All intersections occupied by tokens of a certain type are territory scores for the corresponding game player. Similarly, any connected empty intersections surrounded by only one token type occupying all the adjacent intersections are also scores for that game player. Connected empty intersections with adjacent stones of both player 1 & 2 token types are neutral and as such are not adding to the score of either side. The number of tokens captured by each game player is added to the running total. Whichever side has the highest score is deemed the winner.

## Game Server Logic

# **GO SOFTWARE ASSURANCE ASSESSMENT**

## Software Assurance Case Report

## Secure Sockets Library Imports (OpenSSL/TLS)

## WebApp Security Protocol Configurations

## Countermeasures to Common WebApp Vulnerabilities

## Protection Against Cross-Site Scripting (XSS)

Cross-Site Scripting (XSS) is when the attacker injects malicious code into the client web application in efforts to hack sensitive data including: cookies and browser storage. Attackers can hack sensitive data upon finding any loophole where their query reflects HTML within the web application instead of HTML entities. Attackers can inject any language that will execute within browsers without injecting server-side language.

### User Input Validation

The User Input Validation is used to mark specific action methods from game players whose game play move inputs must be validated.

### Regular Expressions (Regex)

Regular Expressions have been employed to validate the username and password files to ensure that game players create secure credentials.

### Data Cleansing & Sanitization

### Proper Escaping of Metacharacters

## Protection Against Cross-Site Request Forgery (CSRF)

Cross-site Request Forger (CSRF) involves the concept of finding a loophole within a client-side API request where the attacker can send the desired data using the client web browser ID in efforts to target the API. This can occur when any updating any authenticated data or transmitting data from authenticated users like password changes.

## Protection Against SQL-Injection

## Protection Against Low-level Systems Attacks

## Safeguards to Buffer Overflow / OpenSSL Heartbleed

Stack-based Buffer Overflows are a form of out-of-bounds write to a memory area allocated to a buffer residing on the Stack when data written to a buffer exceeds such memory allocated for that Buffer on the Stack. Buffer Overflows can lead to program crashes, memory corruption, segmentation faults. unpredictable program behavior as well as arbitrary code execution. Buffer Overflows overwrite local variables, return addresses, function pointers and anything else that exists within higher memory.

After researching GOLang we have found that as a high-level software programming language employ the follow counter measures against Buffer Overflow including use of StackGuard, special stack canary values which are checked on return to guarantee no stack return address overwrites and address space layout randomization (ASLR). ASLR randomizes the locations of key structures within memory to make reliable exploitation of vulnerabilities difficult by allocation of stack memory at random offsets.

## Safeguards to Format String Vulnerability

Format Strings consist of format parameters and data of which correspond so such parameters. Format String vulnerabilities happen when the input string is interpreted as a command by the program thus allowing the attacker to specify the format string which can lead to program crashes, reading of stack or otherwise arbitrary memory, and writing to arbitrary memory.

We have verified that GoLang mitigates the format string vulnerability by disallowing format strings constructed from user input, uses secure output functions and annotation headers that enable compile-time checks for valid format strings.

## Safeguards to Heap Overflow & Spraying

Heap-based Buffer Overflows while comparable to Stack Overflows are another form of out-of-bounds write to an area of memory allocated to a buffer residing on the Heap where the buffer is dynamically allocated using malloc/new as opposed to a static Stack allocation. Heap Overflows can lead to program crashes, memory corruption and CPU or RAM denial of service. Heap overflow exploits are not as straightforward as Stack as no EIP or return address exists to provide easy control flow hi-jack several potential vulnerabilities exist including: static allocation size with user provided data that exceeds it and user provided allocation size.

GoLang employs use of automatic memory management so our software code is free of any weaknesses which would otherwise exist with the manual freeing any allocated heap memory objects. Additionally, GoLang also uses ASLR to randomize allocations of heap memory at random offsets to ensure that overwriting data allocated to the heap buffer is significantly harder.

## Safeguards to Dangling Pointers (Double Frees & Use After Frees)

***Dangling pointers (CVE-2014-1776)*** are a special breed of memory safety vulnerabilities and programming errors involving direct memory manipulation. Several vulnerabilities exist: pointer double frees, use after frees and expired pointer dereference. The consequences of which lead to “unpredictable behavior” including program crashes from reading/writing invalid memory, memory corruption from writing to valid but unintended memory location and arbitrary code execution should a dangling pointer point to a virtual function table (VFT). Modifying pointer size to include the entire process memory space allows the declared memory object to read/write all process memory thus allowing for code injection and hi-jack.

To circumvent such software weaknesses in our program we employed the use of a high-level software language (GoLang) to ensure automatically explicitly set pointers and all copies of such pointers are set to NULL after freeing and call free() on any pointers pointing to out of scope objects.

## Safeguards to Data Type Confusion

***Object type confusion (CVE-2011-3521, CVE-2012-0507)*** is a particular instance when a program expects an object or resource to be a of given type that does not match the actual type. Essentially a ‘type’ is nothing but a name for a particular layout of memory hence why type confusion vulnerabilities affect how memory is read/written. Such consequences include logic errors concerning accessed objects will have different properties than expected, memory corruption of which memory allocated for a certain type with data of another type can lead to extremely volatile unpredictability.

We have verified that the Golang software programming language ensures such safeguards and support to guarantee type safety through object verification assurances on resources accessed.

## Safeguards to File Path Manipulation

## Data Protection

## Logging of Sensitive Data

## Data Encryption & Confidentiality

AES 256 Symmetric Encryption

## User Privacy / Anonymity & Password Protection

User passwords are stored within the database as iterated salted secure SHA 256 hashes.

## Ensuring Availability & Protection Against Login Failure

## Access Control Protection Policy

### User Authentication & Session Management

#### Safeguards to Man-in-the-Middle (MITM) Attacks

#### Safeguards to Reply Attacks

Employ use of Nonce (number used once) with web cookies.

#### Safeguards to Session Hijacking

Session hijacking is a web application vulnerability involving the exploitation of a web session where the attacker will attempt to hack the session ID’s or user data from the client. Once succeeded in hijacking client the attacker will set the session data in the browser within the same server hosted website and when the attacker refreshes the browser, they will automatically be logged into the same website session.

### Principal of Least Privilege

Principal of Least Privilege ensures that each game player operates using the fewest privileges possible thus limiting the number of potential interactions among privilege processes (including game logging, player score keeping and number of games won.)

To enforce the Principal of Least Privilege our program ensures complete mediation and non-bypassability such that every player access attempt is checked and cannot be subverted. Our program minimizes privileges granted by not granting root and avoids creating setuid root programs. Our game server does all the access checking as our client-side application is untrusted. Privileges have also been separated such that multiple checks are initiated upon game player moves which are dependent on more than one condition. Our web application game maintains a separate administrative interface for the server with different privileges with the client-side being granted the fewest amount of privileges possible.

### Protection Against Malicious Administrator Back-doors

# **Conclusion**

# **References**