# Go Track

PowerUp! SG Tech Traineeship – Software Engineering

# Learning Objectives – Mod 17

At the end of the course, participants should be able to:

- Define security in coding.
- Define the difference with HTTP and TLS.
- Define error handling and logs handling.
- Define sessions security.
- Examine the use of authentication and password management.
- Demonstrate the use of security in coding.

# Error Handling & Logging

Error handling and logging are essential parts of application and infrastructure protection.

**Error Handling**

- Error handling involves capturing and handling of errors in application logic.

**Logging**

- Logging allows identification of all operations that occurred (including info like place and time). These are written on every log line depending on the configuration.
- They help to determine what actions to be taken to protect the system.
- Attackers often try to remove traces of their action by deleting logs, so it is important the logs are centralized.

# Error Handling

- Recall there are 3 possible ways when dealing with error handling:
  - Using the error type in Go.

```
if result, err := Concat(args...); err != nil {
          fmt.Printf("Error: %s\n", err)
} else {
          fmt.Printf("Concatenated string: '%s'\n", result)
}
```

  - Package-scope error variables using error.New function

```
var ErrTimeout = errors.New("The request timed out")
var ErrRejected = errors.New("The request was rejected")

func main() {
          response, err := SendRequest("Hello")
          for err == ErrTimeout {
                    fmt.Println("Timeout. Retrying.")
                    response, err = SendRequest("Hello")
          }
}
```

# Error Handling

- Custom error types:

```
type ParseError struct {
          Message string //error message
          Line, Char int //location
}

func (p *parseError) Error() string{
          format := "%s oln Line %d, Char %d"

          return fmt.Sprintf(format, p.Message, p.Line, p.Char)
}
```

- Recall as well fmt.Errorf() gives the option of using  formatting string on the error message.

```
If f < 0 {
          return o, fmt.Errorf("math:square root of negative number %g", f);
}
```

# Error Handling

- Developers should ensure no sensitive information is disclosed in the error responses.
- Need to guarantee no error handlers leak information (e.g. debugging, or stack trace information)
- Need to assure that in case of an error linked to the security controls, its access is denied by default.

- Recall also Go provides another way of indicating that things go wrong: *panic*, *recover* and *defer*
  - When an application panics, its normal execution is interrupted, any defer statements are executed,
  - Function returns to its caller. recover, usually inside defer statements allows the application to handle and recover from the panic, then return to normal execution.

# Error Handling

- An example of panic:

```
package main

import "fmt"

func main() {
        start()
        fmt.Println("Returned normally from start().")
}
func start() {
        defer func() {
                if r := recover(); r != nil {
                        fmt.Println("Recovered in start()")
                }
        }()
        fmt.Println("Called start()")
        part2(0)
        fmt.Println("Returned normally from part2().")
}
```

# Error Handling

- An example of panic:

```
func part2(i int) {
        if i > 0 {
                fmt.Println("Panicking in part2()!")
                panic(fmt.Sprintf("%v", i))
        }
        defer fmt.Println("Defer in part2()")
        fmt.Println("Executing part2()")
        part2(i + 1)
}
```

Output:
Called start()
Executing part2()
Panicking in part2()!
Defer in part2()
Recovered in start()
Returned normally from start().

# Error Handling

- In the log package, there is also log.Fatal. Fatal is effectively logging the message, then call os.Exit(1).
  - Defer statements not executed
  - Buffers not flushed.
  - Temporary files not removed.
  - Therefore log.Fatal should be used carefully.

- Possible uses of log.Fatal:
  - Set up logging and check whether we have a healthy environment and parameters. If we don't, then there's no need to execute our main().
  - An error that should never occur and that we know that it's unrecoverable.
  - If non-interactive process encounters an error and cannot complete, It's best to stop the execution before additional problems can emerge from this failure.

# Error Handling

- An example of initialization failure:

```
package main

func initFunc(i int) {
...
      //This is just to deliberately crash the function.
      if i < 2 {
            fmt.Printf("Var %d - initialized\n", i)
      } else {
            //This was never supposed to happen, so we'll terminate our program.
            log.Fatal("Init failure - Terminating.")
      }
}
func main() {
       i := 1
       for i < 3 {
            initFunc(i)
            i++
       }
      fmt.Println("Initialized all variables successfully")
}
```

Output:

```
Var 1 - initialized
2020/10/28 16:52:08 Init failure - Terminating.
exit status 1
```

# Logging

- A sample log trace produced by log package

```
TRACE: 2009/11/10 23:00:00:000000 /tmpfs/gosandbox-/prog.go:14: message
```

- The log entry contains
  - prefix
  - datetime stamp,
  - fullpath to source code writing to the log
  - the line of code performing the write
  - the message.

# Logging

- A log example:

```
package main

import (
            "log"
)

func init() {
            log.SetPrefix("TRACE: ")
            log.SetFlags(log.Ldate | log.Lmicroseconds | log.Llongfile)
}

func main() {
            //Println writes to the standard logger.
            log.Println("message")

            //Fatalln is Println() followed by a call to os.Exit(1)
            log.Fatalln("fatal message")

            //Panicln is Println() followed by a call to panic()
            log.Panicln("panic message")
}
```

```
2020/10/28 19:12:19 message
2020/10/28 19:12:19 fatal message
exit status 1
```

# Logging

- init() function is executed before main() as part of program initialization.
- Common to set the log configuration of init() so that log package will be used immediately when program starts.
- "TRACE: " is set to be prefix for each line. Allows for identification of log line over normal program output. By convention, usually in capital letters.

# Logging

- There are several flags associated with log package to control other information on each log line:

```
const (
          Ldate         = 1 << iota     // the date in the local time zone: 2009/01/23
          Ltime                         // the time in the local time zone: 01:23:23
          Lmicroseconds                 // microsecond resolution: 01:23:23.123123.  assumes Ltime.
          Llongfile                     // full file name and line number: /a/b/c/d.go:23
          Lshortfile                    // final file name element and line number: d.go:23. overrides Llongfile
          LUTC                          // if Ldate or Ltime is set, use UTC rather than the local time zone
          Lmsgprefix                    // move the "prefix" from the beginning of the line to before the message
          LstdFlags     = Ldate | Ltime // initial values for the standard logger
)
```

- These flags are declared as constants. Value of iota for each constant gets incremented by 1 with initial value = 0

# Logging

- Behind the scenes with constant declarations:

```
const (
          Ldate         = 1 << iota     // 1 << 0 = 0000000001 = 1
          Ltime                         // 1 << 1 = 0000000010 = 2
          Lmicroseconds                 // 1 << 2 = 0000000100 = 4
          Llongfile                     // 1 << 3 = 0000001000 = 8
          . . .

)
```

- The << operator does a left bitwise shift of the bit pattern =>gives each constant a unique bit position.
- So the log options can be set via

```
log.SetFlags(log.Ldate | log.Lmicroseconds | log.Llongfile) = bits 1+ 4 +8 = 13 (00001101)
```

# Logging

- Good thing about log package is that loggers are *multigoroutine-safe*, meaning multiple goroutines can call these functions from same logger value at same time without writes colliding with each other.
- The standard logger and customized logger created will have this attribute.

# Cusomized loggers

- Useful when need to have different logging levels that can write logs to different destinations.
- Creating customized loggers require you create your own Logger type values.
- Each logger can be configured for a unique destination and set with its own prefix and flags.

# Logging

- Now let's create different logger type pointer values to support different logging levels:

```
package main

import (
            "io"
            "io/ioutil"
            "log"
            "os"
)

var (
            Trace   *log.Logger // Just about anything
            Info    *log.Logger // Important information
            Warning *log.Logger // Be concerned
            Error   *log.Logger // Critical problem
)
```

# Logging

- Now let's create different logger type pointer values to support different logging levels:

```
func init() {
        file, err := os.OpenFile("errors.txt",
                os.O_CREATE|os.O_WRONLY|os.O_APPEND, 0666)
        if err != nil {
                log.Fatalln("Failed to open error log file:", err)
        }

        Trace = log.New(ioutil.Discard,
                "TRACE: ",
                log.Ldate|log.Ltime|log.Lshortfile)

        Info = log.New(os.Stdout,
                "INFO: ",
                log.Ldate|log.Ltime|log.Lshortfile)

        Warning = log.New(os.Stdout,
                "WARNING: ",
                log.Ldate|log.Ltime|log.Lshortfile)
```

# Logging

- Now let's create different logger type pointer values to support different logging levels:

```
            Error = log.New(io.MultiWriter(file, os.Stderr),
                        "ERROR: ",
                        log.Ldate|log.Ltime|log.Lshortfile)
}

func main() {
            Trace.Println("I have something standard to say")
            Info.Println("Special Information")
            Warning.Println("There is something you need to know about")
            Error.Println("Something has failed")
}
```

# Customized loggers

- Each logger type pointer variable is configured differently to represent the importance that each represents.
- To create each logger, use log.New() from log package.
- First parameter is value that implements io.Writer interface, representing the destination the logger is to write to.
- Followed by prefix and the log flags.
- The variables can then be used to call the methods that are implemented by the log package.

# Customized loggers

- Declarations of the different logging methods:

```
func (l *Logger) Fatal(v ...interface{})
func (l *Logger) Fatalf(format string, v ...interface{})
func (l *Logger) Fatalln(v ...interface{})
func (l *Logger) Flags() int
func (l *Logger) Output(calldepth int, s string) error
func (l *Logger) Panic(v ...interface{})
func (l *Logger) Panicf(format string, v ...interface{})
func (l *Logger) Panicln(v ...interface{})
func (l *Logger) Prefix() string
func (l *Logger) Print(v ...interface{})
func (l *Logger) Printf(format string, v ...interface{})
func (l *Logger) Println(v ...interface{})
func (l *Logger) SetFlags(flag int)
func (l *Logger) SetPrefix(prefix string)
```

# Logging

- Logging should always be handled by the application and should not rely on a server configuration.
- Should be implemented by a master routine on a trusted system.
- Need to ensure no sensitive data is included in the logs (e.g. passwords, system details, session information etc) or any debugging or stack trace information.

Do we log everything?

- Logging should cover both successful and unsuccessful security events.
- Emphasis is on important log event data.

# Logging

- Common important event data:
  - Input validation
  - Authentication attempts, especially failures
  - Access control
  - Apparent tampering events, including unexpected changes to state data
  - Attempts to connect with invalid or expired session tokens.
  - System exceptions
  - Administrative functions, including changes to security configuration settings.
  - Backend TLS failure and cryptographic module failures.

# Logging

- Go's log package only implements simple logging.
- It does not provide levelled logging (e.g. debug, info, error, fatal, panic) and formatting support (e.g. logstash).
- These 2 features actually make logs sometimes unusable (e.g. for integration with Security Information or Event Management system).
- Other than creating customized loggers, there are third party libraries that offer these features and others. Popular ones are:
    - Logrus - https://github.com/Sirupsen/logrus
    - glog - https://github.com/golang/glog
    - loggo - https://github.com/juju/loggo

# Logging

- Fatal and panic functions have different behaviors after logging: Panic functions call panic but Fatal functions call os.Exit(1)
- Fatal terminates the program - preventing deferred statements to run, buffers to be flushed, and/or temporary data to be removed.
- Log access: only authorized individuals should have access to the logs.
- Developers should also make sure that a mechanism that allows for log analysis is set in place

# Logging

- Need to guarantee that no untrusted data will be executed as code in the intended log viewing software or interface.
- To guarantee log validity and integrity, a *cryptographic hash function* should be used as an additional step to ensure no log tampering has taken place.
- The log-file hashes must be stored in a safe place, and compared with the current log hash to verify integrity before any updates to the log.

# Logging

- Example to do a hash of the log. ComputeSHA256() calculates a file's SHA256.

```
{...}
// Get our known Log checksum from checksum file.
logChecksum, err := ioutil.ReadFile("log/checksum")
str := string(logChecksum) // convert content to a 'string'

// Compute our current log's SHA256 hash
b, err := ComputeSHA256("log/log")

if err != nil {
          fmt.Printf("Err: %v", err)
} else {
          hash := hex.EncodeToString(b)
          // Compare our calculated hash with our stored hash
          if str == hash {
                    // Ok the checksums match.
                    fmt.Println("Log integrity OK.")
          } else {
                    // The file integrity has been compromised...
          fmt.Println("File Tampering detected.")
          }
}
```

# Cryptographic Practices

Hashing ≠ Encryption

- A *hash* is a string or number generated by a function (hash) from source data. `hash = f(data)`
- The hash has fixed length. Its values may vary widely with small variations in input (collisions may still happen).
- A good hashing algorithm will not allow a hash to turn into its original source. MD5 was one of the popular algorithms, but BLAKE2 is the strongest and most flexible.
- Go supplementary cryptography libraries offers both BLAKE2b (or just BLAKE2) and BLAKE2s implementations: If BLAKE2 is unavailable, SHA-256 is the right option.

# Cryptograhics Practices

- Slowness is desired on a cryptographic hashing algorithm.
- Computers become faster over time, meaning that attacker can try more and more potential passwords as years pass. Thus the hashing function should be inherently slow.
- Rule of thumb: whenever you have something that you don't need to know what it is, but only if it's what it is supposed to be (like checking file integrity after download)=>use hashing

# Cryptographic Practices

```
package main

import (
            "crypto/md5"
            "crypto/sha256"
            "fmt"
            "io"

            "golang.org/x/crypto/blake2s"
)
func main() {
            hMd5 := md5.New()
            hSha := sha256.New()
            hBlake2s, _ := blake2s.New256(nil)
            io.WriteString(hMd5, "Welcome to Go Language Secure Coding Practices")
            io.WriteString(hSha, "Welcome to Go Language Secure Coding Practices")
            io.WriteString(hBlake2s, "Welcome to Go Language Secure Coding Practices")
            fmt.Printf("MD5 : %x\n", hMd5.Sum(nil))
            fmt.Printf("SHA256 : %x\n", hSha.Sum(nil))
            fmt.Printf("Blake2s-256: %x\n", hBlake2s.Sum(nil))
}
```

Need to run
go get golang.org/x/crypto/blake2s

```
MD5 : ea9321d8fb0ec6623319e49a634aad92
SHA256 : ba4939528707d791242d1af175e580c584dc0681af8be2a4604a526e864449f6
Blake2s-256: 1d65fa02df8a149c245e5854d980b38855fd2c78f2924ace9b64e8b21b3f2f82
```

# Cryptographics Practices

- Encryption on other hand turns data into variable length data using a key

```
encrypted_data = F(data, key)
```

- Unlike hash, data can be converted back from encryption_data by applying the right decryption function and key

```
data = F-1(encrypted_data, key)
```

# Cryptographic Practices


- In cryptography, *ciphertext* is the result of encryption performed on *plaintext* using an algorithm, called a *cipher*.

# Cryptographic Practices

- Encryption should be used whenever you need to communicate or store sensitive data, which you or someone else needs to access later on for further processing.

- A encryption use case is *HTTPS - Hyper Text Transfer Protocol Secure*.
- *AES* is the de facto standard when it comes to *symmetric key encryption*. This algorithm, similar to many other symmetric ciphers, can be implemented in different modes. (e.g. GCM (Galois Counter Mode) or the more popular CBC (Cipher Block Chaining)/ECB (Electronic Code Book).

# Cryptographic Practices

- Main difference between GCM and CBC/ECB is the fact that the former is an authenticated cipher mode; after the encryption stage, an authentication tag is added to the ciphertext, which will then be validated prior to message decryption, ensuring the message has not been tampered with.

# Cryptographic Practices

- *Public key cryptography* or *asymmetric cryptography* which makes use of pairs of keys: public and private.
- Public key cryptography offers less performance than symmetric key cryptography for most cases.
- Therefore, its most common use-case is sharing a symmetric key between two parties using symmetric cryptography, so they can then use the symmetric key to exchange messages encrypted with symmetric cryptography.
- Aside from AES, which is 1990's technology, Go authors have begun to implement and support more modern symmetric encryption algorithms, which also provide authentication. e.g.chacha20poly1305.

# Cryptographic Practices

- Another interesting package in Go is x/crypto/nacl. It is reference to Dr. Daniel J. Bernstein's NaCl library, a very popular modern cryptography library.
- nacl/box and nacl/secretbox in Go are implementations of NaCl's abstractions for sending encrypted messages for the two most common use-cases:
  - Sending authenticated, encrypted messages between two parties using public key cryptography (nacl/box)
  - Sending authenticated, encrypted messages between two parties using symmetric (or secret-key) cryptography
- It is very advisable to use one of these abstractions instead of direct use of AES, if they fit your use-case.

# Cryptographic Practices

```
package main

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

func main() {
            key := []byte("Encryption Key should be 32 char")
            data := []byte("Welcome to Go in Action 2")
            block, err := aes.NewCipher(key)
            if err != nil {
                        panic(err.Error())
            }
            nonce := make([]byte, 12)
            if _, err := rand.Read(nonce); err != nil {
                        panic(err.Error())
            }
```

# Cryptographic Practices

```
        aesgcm, err := cipher.NewGCM(block)
        if err != nil {
                panic(err.Error())
        }
        encryptedData := aesgcm.Seal(nil, nonce, data, nil)
        fmt.Printf("Encrypted: %x\n", encryptedData)

        decryptedData, err := aesgcm.Open(nil, nonce, encryptedData, nil)
        if err != nil {
                panic(err.Error())
        }
        fmt.Printf("Decrypted: %s\n", decryptedData)
}
```

```
Encrypted: 3b751a2695c4376526def7b0f738de4cf49182ab57bc1a08bd9e354410a0cb7b175a891a802f905b4f
Decrypted: Welcome to Go in Action 2
```

# Cryptographic Practices

- A policy needs to be established and used for how cryptographic keys will be managed ", protected from unauthorized access.
- The cryptographic keys should also not be hardcoded in the source code (as it is in the example).
- Go's crypto package collects common cryptographic constants, but implementations have their own packages, like the crypto/md5 one.
- Most modern cryptographic algorithms have been implemented under https://godoc.org/golang.org/x/crypto so developers could focus on those instead of the implementations in the crypto/* package.

# Pseudo-Random Generators

- According to OWASP:

  *"All random numbers, random file names, random GUIDs, and random strings should be generated using the cryptographic module's approved random number generator when these random values are intended to be un-guessable"*

- Cryptography relies on some randomness, but for the sake of correctness, what most programming languages provide out-of-the box is a pseudo-random number generator: for example, Go's math/rand is not an exception.

# Pseudo-Random Generators

- You should carefully read the documentation when it states that

  "Top-level functions, such as Float64 and Int, use a default shared Source that produces a deterministic sequence of values each time a program is run."

  https://golang.org/pkg/math/rand/#pkg-overview

# Pseudo-Random Generators

```
package main
import "fmt"
import "math/rand"


func main() {
          fmt.Println("Random Number: ", rand.Intn(1984))
}
```

- Running this program several times will lead exactly to the same number/sequence.

```
$ for i in {1..5}; do go run rand.go; done
Random Number:  1825
Random Number:  1825
Random Number:  1825
Random Number:  1825
Random Number:  1825
```

# Pseudo-Random Generators

- Go's math/rand is a deterministic pseudo-random number generator. Similar to many others, it uses a *Seed*.
- This Seed is solely responsible for the randomness of the deterministic pseudo-random number generator. If it is known or predictable, the same will happen to generated number sequence.
- We could "fix" this example quite easily by using the math/rand Seed function, getting the expected five different values for each program execution. But here we we would see crypto/rand package.

# Pseudo-Random Generators

```
package main

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

func main() {
            rand, err := rand.Int(rand.Reader, big.NewInt(1984))
            if err != nil {
                        panic(err)
            }
            fmt.Printf("Random Number: %d\n", rand)
}
```

```
Random Number: 277
Random Number: 1572
Random Number: 1793
Random Number: 1328
Random Number: 1378
```

# Pseudo-Random Generators

- Running crypto/rand is slower than math/rand, but this is expected since the fastest algorithm isn't always the safest. Crypto's rand is also safer to implement.
- An example of this is the fact that you CANNOT seed crypto/rand; the library uses OS-randomness for this, preventing developer misuse.

How this can be exploited?

- Think what happens if your application creates a default password on user signup, by computing hash of a pseudo-random number generated with Go's math/rand, as shown in the first example. One would be able to predict the user's password!

# Session Management


- The application should only recognize the server's session management controls.
- Creation of session should be done on a trusted system.
- Need to ensure the algorithms used to generate the session identifier are sufficiently random. This is to prevent session brute forcing.

```
...
token := jwt.NewWithClaims(jwt.SigningMethodHS256, claims)
signedToken, _ := token.SignedString([]byte("secret")) //our secret
...
```

# Session Management

- Next is to set the Domain, Path, Expires, HTTP only, Secure for the cookies.

```
// Our cookie parameter
cookie := http.Cookie{
          Name: "Auth",
          Value: signedToken,
          Expires: expireCookie,
          HttpOnly: true,
          Path: "/",
          Domain: "127.0.0.1",
          Secure: true
}
http.SetCookie(res, &cookie) //Set the cookie
```

# Session Management

- Upon sign-in, a new session is always generated.
- Old session is never re-used, even if not expired.
- Also should use the Expire parameter to enforce periodic termination of session to prevent session hijacking.

- Another important part about cookies is to disallow concurrent login for the same username.
- This is done by maintaining list of logged in users, and compare the new login username against this list. This list is usually kept in database or other forms of persistent storage.

# Session Management

- Session identifiers should not be exposed in URL's.
- They should only be located in the HTTP cookie header.
- Session identifiers or any sensitive information should not be passed as HTTP GET parameters. Should try to use POST or HTTPS.
- Session data should be protected from unauthorized access by other users of the server.
- For HTTP to HTTPS connection changes, need special care to prevent MITM attacks (that sniff and hijack the user's session). Best practice is to use HTTPS in all requests.

```
err := http.ListenAndServeTLS(":443", "cert/cert.pem", "cert/key.pem",
nil)
if err != nil {
          log.Fatal("ListenAndServe: ", err)
}
```

# Session Management

- For highly sensitive or critical operations, token should be generate per-request, instead of per-session. (but very rarely practiced because hard to keep track and pose difficult in troubleshooting when there are issues)
- Make sure the token is sufficiently random and has length secure enough to protect against brute forcing.

# Session Management

- Final consideration is the Logout functionality. Application should provide a way to logout from all pages that require authentication, and also fully terminate the associated session and connection. E.g. when user logs out, cookie is deleted from the client.
- Same action be taken in location where user session information is stored.

# Authentication & Password Management

Rule of thumb:

*"All authentication controls must be enforced on a trusted system" which usually is the server that the application's backend is running."*

- For the sake of system's simplicity, and to reduce the points of failure, standard and tested authentication services should be used.
- Usually *frameworks* already have such module and it is encouraged to use them as they are developed, maintained, and used by many people behaving as a centralized authentication mechanism.

# Authentication & Password Management

- Inspection of program code is needed to ensure it is not affected by any malicious code", and be sure that it follows the best practices.
- Resources which require authentication should not perform it themselves. Instead, "redirection to and from the centralized authentication control" should be used.
- Be careful handling redirection: you should redirect only to local and/or safe resources.
- Authentication should not be used only by the application's users, but also by your own application when it requires "connection to external systems that involve sensitive information or functions".

# Authentication & Password Management

- In these cases, "authentication credentials for accessing services external to the application should be encrypted and stored in a protected location on a trusted system (e.g., the server). The source code is NOT a secure location ".

# Communicating Authentication Data

- Not only is it true that "password entry should be obscured on user's screen ", but also the "remember me functionality should be disabled".
- Both can be done by using an input field with type="password" , and setting the autocomplete attribute to off .

```
<input type="password" name="passwd" autocomplete="off" />
```

- Authentication credentials should be sent only through encrypted connections (HTTPS). An exception to the encrypted connection may be the temporary passwords associated with email resets.

# Communicating Authentication Data

- Remember that requested URLs are usually logged by the HTTP server ( access_log ), which include the query string.
- To prevent authentication credentials leakage to HTTP server logs, data should be sent to the server using the HTTP POST method.

```
xxx.xxx.xxx.xxx - - [27/Feb/2017:01:55:09 +0000] "GET /?username=user&password=70pS3c
```

- A well-designed HTML form for authentication would look like:

```
<form method="post" action="https://somedomain.com/user/signin" autocomplete="off">
    <input type="hidden" name="csrf" value="CSRF-TOKEN" />

    <label>Username <input type="text" name="username" /></label>
    <label>Password <input type="password" name="password" /></label>

    <input type="submit" value="Submit" />
</form>
```

# Communicating Authentication Data

- When handling authentication errors, your application should not disclose which part of the authentication data was incorrect.
- Instead of "Invalid username" or "Invalid password", just use "Invalid username and/or password" interchangeably:

```
<form method="post" action="https://somedomain.com/user/signin" autocomplete="off">
    <input type="hidden" name="csrf" value="CSRF-TOKEN" />

    <div class="error">
        <p>Invalid username and/or password</p>
    </div>

    <label>Username <input type="text" name="username" /></label>
    <label>Password <input type="password" name="password" /></label>

    <input type="submit" value="Submit" />
</form>
```

# Communicating Authentication Data

- Using a generic message you do not disclose:
  - Who is registered: "Invalid password" means that the username exists.
  - How your system works: "Invalid password" may reveal how your application works, first querying the database for the username and then comparing passwords in-memory.
- After a successful login, the user should be informed about the last successful or unsuccessful access date/time so that he can detect and report suspicious activity.

# Communicating Authentication Data

- Additionally, it is also recommended to use a constant time comparison function while checking passwords in order to prevent a timing attack. The latter consists of analyzing the difference of time between multiple requests with different inputs.
- In this case, a standard comparison of the form record == password would return false at the first character that does not match. The closer the submitted password is, the longer the response time.
- By exploiting that, an attacker could guess the password. Note that even if the record doesn't exist, we always force the execution of subtle.ConstantTimeCompare with an empty value to compare it to the user input.

# Validation and storing authencation data

- More often than desirable, user account databases are leaked on the Internet. In the case of such an event, collateral damages can be avoided if authentication data, especially passwords, are stored properly.

  *"all authentication controls should fail securely".*

- We've covered recommendations about reporting back wrong authentication data and how to handle logging.
- One other preliminary recommendation is as follow: for sequential authentication implementations (like Google does nowadays), validation should happen only on the completion of all data input, on a trusted system (e.g. the server).

# Storing password securely

- Never roll your own crypto. By doing so, one can put the entire application at risk. It is a sensitive and complex topic.
- It is important to use cryptography tools that provides features and standards reviewed and approved by experts, instead of trying to re-invent the wheel.
- In the case of password storage, the hashing algorithms recommended by OWASP are bcrypt , PDKDF2 , Argon2 and scrypt . They take care of hashing and salting passwords in a robust way.
- Go authors provide an extended package for cryptography, that is not part of the standard library. It provides robust implementations for most of these algorithms. It can be downloaded using go get : go get golang.org/x/crypto

# Storing password securely

- Here's an example how to use bcrypt:

```
package main

import (
            "context"

            "golang.org/x/crypto/bcrypt"
)

func main() {
            ctx := context.Background()
            email := []byte("john.doe@somedomain.com")
            password := []byte("47;u5:B(95m72;Xq")
            // Hash the password with bcrypt
            hashedPassword, err := bcrypt.GenerateFromPassword(password, bcrypt.DefaultCost)
            if err != nil {
                        panic(err)
            }
```

# Storing password securely

```
            // this is here just for demo purposes
            //
            // fmt.Printf("email : %s\n", string(email))
            // fmt.Printf("password : %s\n", string(password))
            // fmt.Printf("hashed password: %x\n", hashedPassword)
            // you're supposed to have a database connection
            stmt, err := db.PrepareContext(ctx, "INSERT INTO accounts SET hash=?, email=?")
            if err != nil {
                        panic(err)
            }
            result, err := stmt.ExecContext(ctx, hashedPassword, email)
            if err != nil {
                        panic(err)
            }
}
```

# Storing password securely

- Bcrypt also provides a simple and secure way to compare a plaintext password with an already hashed password.

```
ctx := context.Background()

// credentials to validate
email := []byte("john.doe@somedomain.com")
password := []byte("47;u5:B(95m72;Xq")

// fetch the hashed password corresponding to the provided email
record := db.QueryRowContext(ctx, "SELECT hash FROM accounts WHERE email = ? LIMIT 1"

var expectedPassword string
if err := record.Scan(&expectedPassword); err != nil {
// user does not exist
// this should be logged (see Error Handling and Logging) but execution
// should continue
}
```

# Storing password securely

```
if bcrypt.CompareHashAndPassword(password, []byte(expectedPassword)) != nil {
// passwords do not match
// passwords mismatch should be logged (see Error Handling and Logging)
// error should be returned so that a GENERIC message "Sign-in attempt has
// failed, please check your credentials" can be shown to the user.
}
```

- If you're not comfortable with password hashing and comparison options/parameters, better delegating the task to specialized third- party package with safe defaults.
- Always opt for an actively maintained package and remind to check for known issues.

# Storing password securely

- passwd - A Go package that provides a safe default abstraction for password hashing and comparison. It has support for original go bcrypt implementation, argon2, scrypt, parameters masking and key'ed (uncrackable) hashes.