目　录

介绍

代码安全指南

面向开发人员梳理的代码安全指南，旨在梳理API层面的风险点并提供详实可行的安全编码方案。

理念

基于DevSecOps理念，我们希望用开发者更易懂的方式阐述安全编码方案，引导从源头规避漏洞。

索引

规范	最后修订日期
C/C++安全指南	2021-05-18
JavaScript安全指南	2021-05-18
Node安全指南	2021-05-18
Go安全指南	2021-05-18
Java安全指南	2021-05-18
Python安全指南	2021-05-18

实践

代码安全指引可用于以下场景：

- 开发人员日常参考
- 编写安全系统扫描策略
- 安全组件开发
- 漏洞修复指引

贡献

欢迎通过Issue或PR的方式提交修订建议，示例如下：

标题：#JavaScript# 规范1.3.1条修订建议

内容：
1、问题描述
JavaScript代码安全规范的【1.3.1条】赋值或更新HTML属性部分，需补充

2、解决建议
应补充下列风险点：
area.href、input.formaction、button.formaction

通用类

代码实现

文件操作

系统接口

通信安全

敏感数据保护

加密解密

正则表达式

内存管理

代码实现

1.【必须】切片长度校验

- 在对slice进行操作时，必须判断长度是否合法，防止程序panic

```
// bad: 未判断data的长度，可导致 index out of range
func decode(data [] byte) bool {
    if data[0] == 'F' && data[1] == 'U' && data[2] == 'Z' && data[3] == 'Z' && d
ata[4] == 'E' && data[5] == 'R' {
        fmt.Println("Bad")
        return true
    }

    return false
}

// bad: slice bounds out of range
func foo() {
    var slice = []int{0, 1, 2, 3, 4, 5, 6}
    fmt.Println(slice[:10])
}

// good: 使用data前应判断长度是否合法
func decode(data [] byte) bool {
    if len(data) == 6 {
        if data[0] == 'F' && data[1] == 'U' && data[2] == 'Z' && data[3] == 'Z'
 && data[4] == 'E' && data[5] == 'R' {
            fmt.Println("Good")
            return true
        }
    }

    return false
}
```

2.【必须】nil指针判断

- 进行指针操作时，必须判断该指针是否为nil，防止程序panic，尤其在进行结构体Unmarshal时

```
type Packet struct {
    PackeyType        uint8
    PackeyVersion     uint8
    Data              *Data
}

type Data struct {
    Stat    uint8
    Len     uint8
    Buf     [8]byte
}

func (p *Packet) UnmarshalBinary(b []byte) error {
    if len(b) < 2 {
       return io.EOF
    }

    p.PackeyType = b[0]
    p.PackeyVersion = b[1]

    // 若长度等于2，那么不会new Data
    if len(b) > 2 {
        p.Data = new(Data)
        // Unmarshal(b[i:], p.Data)
    }

    return nil
}

// bad: 未判断指针是否为nil
func main() {
    packet := new(Packet)
    data := make([]byte, 2)
    if err := packet.UnmarshalBinary(data); err != nil {
        fmt.Println("Failed to unmarshal packet")
        return
    }

    fmt.Printf("Stat: %v\n", packet.Data.Stat)
}

// good: 判断Data指针是否未nil
```

```
func main() {

    packet := new(Packet)
    data := make([]byte, 2)

    if err := packet.UnmarshalBinary(data); err != nil {
        fmt.Println("Failed to unmarshal packet")
        return
    }

    if packet.Data == nil {
        return
    }

    fmt.Printf("Stat: %v\n", packet.Data.Stat)
}
```

3.【必须】整数安全

- 在进行数字运算操作时，需要做好长度限制，防止外部输入运算导致异常：
    - 确保无符号整数运算时不会反转
    - 确保有符号整数运算时不会出现溢出
    - 确保整型转换时不会出现截断错误
    - 确保整型转换时不会出现符号错误
- 以下场景必须严格进行长度限制：
    - 作为数组索引
    - 作为对象的长度或者大小
    - 作为数组的边界（如作为循环计数器）

```
// bad: 未限制长度，导致整数溢出
func overflow(numControlByUser int32) {
    var numInt int32 = 0
    numInt = numControlByUser + 1
    //对长度限制不当，导致整数溢出
    fmt.Printf("%d\n", numInt)
    //使用numInt，可能导致其他错误
}

func main() {
```

```
    overflow(2147483647)
}

// good:
func overflow(numControlByUser int32) {
    var numInt int32 = 0
    numInt = numControlByUser + 1
    if numInt < 0 {
        fmt.Println("integer overflow")
        return;
    }
    fmt.Println("integer ok")
}

func main() {
    overflow(2147483647)
}
```

4.【必须】make分配长度验证

- 在进行make分配内存时，需要对外部可控的长度进行校验，防止程序panic。

```
// bad
func parse(lenControlByUser int, data[] byte) {
    size := lenControlByUser
    //对外部传入的size，进行长度判断以免导致panic
    buffer := make([]byte, size)
    copy(buffer, data)
}

// good
func parse(lenControlByUser int, data[] byte) ([]byte, error){
    size := lenControlByUser
    //限制外部可控的长度大小范围
    if size > 64*1024*1024 {
        return nil, errors.New("value too large")
    }
    buffer := make([]byte, size)
    copy(buffer, data)
    return buffer, nil
}
```

5.【必须】禁止SetFinalizer和指针循环引用同时使用

- 当一个对象从被GC选中到移除内存之前，runtime.SetFinalizer()都不会执行，即使程序正常结束或者发生错误。由指针构成的“循环引用”虽然能被GC正确处理，但由于无法确定Finalizer依赖顺序，从而无法调用runtime.SetFinalizer()，导致目标对象无法变成可达状态，从而造成内存无法被回收。

```
// bad
func foo() {
    var a, b Data
    a.o = &b
    b.o = &a

    //指针循环引用，SetFinalizer()无法正常调用
    runtime.SetFinalizer(&a, func(d *Data) {
        fmt.Printf("a %p final.\n", d)
    })
    runtime.SetFinalizer(&b, func(d *Data) {
        fmt.Printf("b %p final.\n", d)
    })
}

func main() {
    for {
        foo()
        time.Sleep(time.Millisecond)
    }
}
```

6.【必须】禁止重复释放channel

- 重复释放一般存在于异常流程判断中，如果恶意攻击者构造出异常条件使程序重复释放channel，则会触发运行时恐慌，从而造成DoS攻击。

```
// bad
func foo(c chan int) {
    defer close(c)
    err := processBusiness()
    if err != nil {
        c <- 0
        close(c) // 重复释放channel
        return
    }
    c <- 1
```

```
}

// good
func foo(c chan int) {
    defer close(c) // 使用defer延迟关闭channel
    err := processBusiness()
    if err != nil {
        c <- 0
        return
    }
    c <- 1
}
```

7.【必须】确保每个协程都能退出

- 启动一个协程就会做一个入栈操作，在系统不退出的情况下，协程也没有设置退出条件，则相当于协程失去了控制，它占用的资源无法回收，可能会导致内存泄露。

```
// bad: 协程没有设置退出条件
func doWaiter(name string, second int) {
    for {
        time.Sleep(time.Duration(second) * time.Second)
        fmt.Println(name, " is ready!")
    }
}
```

8.【推荐】不使用unsafe包

- 由于unsafe包绕过了 Golang 的内存安全原则，一般来说使用该库是不安全的，可导致内存破坏，尽量避免使用该包。若必须要使用unsafe操作指针，必须做好安全校验。

```
// bad: 通过unsafe操作原始指针
func unsafePointer() {
    b := make([]byte, 1)
    foo := (*int)(unsafe.Pointer(uintptr(unsafe.Pointer(&b[0])) + uintptr(0xffff
fffe)))
    fmt.Print(*foo + 1)
}

// [signal SIGSEGV: segmentation violation code=0x1 addr=0xc100068f55 pc=0x49142
b]
```

9.【推荐】不使用slice作为函数入参

- slice是引用类型，在作为函数入参时采用的是地址传递，对slice的修改也会影响原始数据

```
// bad
// slice作为函数入参时是地址传递
func modify(array []int) {
    array[0] = 10 // 对入参slice的元素修改会影响原始数据
}

func main() {
    array := []int{1, 2, 3, 4, 5}

    modify(array)
    fmt.Println(array) // output: [10 2 3 4 5]
}

// good
// 数组作为函数入参时，而不是slice
func modify(array [5]int) {
   array[0] = 10
}

func main() {
    // 传入数组，注意数组与slice的区别
    array := [5]int{1, 2, 3, 4, 5}

    modify(array)
    fmt.Println(array)
}
```

文件操作

文件操作

1.【必须】 路径穿越检查

- 在进行文件操作时，如果对外部传入的文件名未做限制，可能导致任意文件读取或者任意文件写入，严重可能导致代码执行。

```
// bad: 任意文件读取
func handler(w http.ResponseWriter, r *http.Request) {
    path := r.URL.Query()["path"][0]

    // 未过滤文件路径，可能导致任意文件读取
    data, _ := ioutil.ReadFile(path)
    w.Write(data)

    // 对外部传入的文件名变量，还需要验证是否存在../等路径穿越的文件名
    data, _ = ioutil.ReadFile(filepath.Join("/home/user/", path))
    w.Write(data)
}

// bad: 任意文件写入
func unzip(f string) {
    r, _ := zip.OpenReader(f)
    for _, f := range r.File {
        p, _ := filepath.Abs(f.Name)
        // 未验证压缩文件名，可能导致../等路径穿越，任意文件路径写入
        ioutil.WriteFile(p, []byte("present"), 0640)
    }
}

// good: 检查压缩的文件名是否包含..路径穿越特征字符，防止任意写入
func unzipGood(f string) bool {
    r, err := zip.OpenReader(f)
    if err != nil {
        fmt.Println("read zip file fail")
        return false
    }
    for _, f := range r.File {
        p, _ := filepath.Abs(f.Name)
        if !strings.Contains(p, "..") {
```

```
            ioutil.WriteFile(p, []byte("present"), 0640)
        }
    }
    return true
}
```

2.【必须】 文件访问权限

- 根据创建文件的敏感性设置不同级别的访问权限，以防止敏感数据被任意权限用户读取。例如，设置文件权限为：`-rw-r-----`

```
ioutil.WriteFile(p, []byte("present"), 0640)
```

系统接口

系统接口

1.【必须】命令执行检查

- 使用`exec.Command`、`exec.CommandContext`、`syscall.StartProcess`、`os.StartProcess`等函数时，第一个参数（path）直接取外部输入值时，应使用白名单限定可执行的命令范围，不允许传入`bash`、`cmd`、`sh`等命令；
- 使用`exec.Command`、`exec.CommandContext`等函数时，通过`bash`、`cmd`、`sh`等创建shell，-c后的参数（arg）拼接外部输入，应过滤\n $ & ; | ‘ “ ( ) `等潜在恶意字符；

```
// bad
func foo() {
    userInputedVal := "&& echo 'hello'" // 假设外部传入该变量值
    cmdName := "ping " + userInputedVal

    //未判断外部输入是否存在命令注入字符，结合sh可造成命令注入
    cmd := exec.Command("sh", "-c", cmdName)
    output, _ := cmd.CombinedOutput()
    fmt.Println(string(output))

    cmdName := "ls"
    //未判断外部输入是否是预期命令
    cmd := exec.Command(cmdName)
    output, _ := cmd.CombinedOutput()
    fmt.Println(string(output))
}

// good
func checkIllegal(cmdName string) bool {
    if strings.Contains(cmdName, "&") || strings.Contains(cmdName, "|") || strings.Contains(cmdName, ";") ||
        strings.Contains(cmdName, "$") || strings.Contains(cmdName, "'") || strings.Contains(cmdName, "`") ||
        strings.Contains(cmdName, "(") || strings.Contains(cmdName, ")") || strings.Contains(cmdName, "\"") {
        return true
```

```
    }
    return false
}

func main() {
    userInputedVal := "&& echo 'hello'"
    cmdName := "ping " + userInputedVal

    if checkIllegal(cmdName) { // 检查传给sh的命令是否有特殊字符
        return // 存在特殊字符直接return
    }

    cmd := exec.Command("sh", "-c", cmdName)
    output, _ := cmd.CombinedOutput()
    fmt.Println(string(output))
}
```

通信安全

通信安全

1.【必须】网络通信采用TLS方式

- 明文传输的通信协议目前已被验证存在较大安全风险，被中间人劫持后可能导致许多安全风险，因此必须采用至少TLS的安全通信方式保证通信安全，例如gRPC/Websocket都使用TLS1.3。

```
// good
func main() {
  http.HandleFunc("/", func (w http.ResponseWriter, req *http.Request) {
    w.Header().Add("Strict-Transport-Security", "max-age=63072000; includeSubDomains")
    w.Write([]byte("This is an example server.\n"))
  })

  //服务器配置证书与私钥
  log.Fatal(http.ListenAndServeTLS(":443", "yourCert.pem", "yourKey.pem", nil))
}
```

2.【推荐】TLS启用证书验证

- TLS证书应当是有效的、未过期的，且配置正确的域名，生产环境的服务端应启用证书验证。

```
// bad
import (
    "crypto/tls"
    "net/http"
)

func doAuthReq(authReq *http.Request) *http.Response {
    tr := &http.Transport{
        TLSClientConfig: &tls.Config{InsecureSkipVerify: true},
    }
    client := &http.Client{Transport: tr}
    res, _ := client.Do(authReq)
    return res
```

```
}

// good
import (
    "crypto/tls"
    "net/http"
)

func doAuthReq(authReq *http.Request) *http.Response {
    tr := &http.Transport{
        TLSClientConfig: &tls.Config{InsecureSkipVerify: false},
    }
    client := &http.Client{Transport: tr}
    res, _ := client.Do(authReq)
    return res
}
```

敏感数据保护

敏感数据保护

1.【必须】敏感信息访问

- 禁止将敏感信息硬编码在程序中，既可能会将敏感信息暴露给攻击者，也会增加代码管理和维护的难度
- 使用配置中心系统统一托管密钥等敏感信息

2.【必须】敏感数据输出

- 只输出必要的最小数据集，避免多余字段暴露引起敏感信息泄露
- 不能在日志保存密码（包括明文密码和密文密码）、密钥和其它敏感信息
- 对于必须输出的敏感信息，必须进行合理脱敏展示

```
// bad
func serve() {
    http.HandleFunc("/register", func(w http.ResponseWriter, r *http.Request) {
        r.ParseForm()
        user := r.Form.Get("user")
        pw := r.Form.Get("password")

        log.Printf("Registering new user %s with password %s.\n", user, pw)
    })
    http.ListenAndServe(":80", nil)
}

// good
func serve1() {
    http.HandleFunc("/register", func(w http.ResponseWriter, r *http.Request) {
        r.ParseForm()
        user := r.Form.Get("user")
        pw := r.Form.Get("password")

        log.Printf("Registering new user %s.\n", user)

        // ...
        use(pw)
    })
```

```
    http.ListenAndServe(":80", nil)
}
```

- 避免通过GET方法、代码注释、自动填充、缓存等方式泄露敏感信息

3.【必须】敏感数据存储

- 敏感数据应使用SHA2、RSA等算法进行加密存储
- 敏感数据应使用独立的存储层，并在访问层开启访问控制
- 包含敏感信息的临时文件或缓存一旦不再需要应立刻删除

4.【必须】异常处理和日志记录

- 应合理使用panic、recover、defer处理系统异常，避免出错信息输出到前端

```
defer func () {
        if r := recover(); r != nil {
            fmt.Println("Recovered in start()")
        }
    }()
```

- 对外环境禁止开启debug模式，或将程序运行日志输出到前端

错误例子：

```
dlv --listen=:2345 --headless=true --api-version=2 debug test.go
```

正确例子：

```
dlv debug test.go
```

加密解密

加密解密

1.【必须】不得硬编码密码/密钥

- 在进行用户登陆，加解密算法等操作时，不得在代码里硬编码密钥或密码，可通过变换算法或者配置等方式设置密码或者密钥。

```
// bad
const (
    user     = "dbuser"
    password = "s3cretp4ssword"
)

func connect() *sql.DB {
    connStr := fmt.Sprintf("postgres://%s:%s@localhost/pqgotest", user, passwor
d)
    db, err := sql.Open("postgres", connStr)
    if err != nil {
        return nil
    }
    return db
}

// bad
var (
    commonkey = []byte("0123456789abcdef")
)

func AesEncrypt(plaintext string) (string, error) {
    block, err := aes.NewCipher(commonkey)
    if err != nil {
        return "", err
    }
}
```

2.【必须】密钥存储安全

- 在使用对称密码算法时，需要保护好加密密钥。当算法涉及敏感、业务数据时，可通过非对称算法协商加密密钥。其他较为不敏感的数据加密，可以通过变换算法等方式保护密

钥。

3.【推荐】不使用弱密码算法

- 在使用加密算法时，不建议使用加密强度较弱的算法。

错误例子：

```
crypto/des，crypto/md5，crypto/sha1，crypto/rc4等。
```

正则表达式

正则表达式

1.【推荐】使用regexp进行正则表达式匹配

- 正则表达式编写不恰当可被用于DoS攻击，造成服务不可用，推荐使用regexp包进行正则表达式匹配。regexp保证了线性时间性能和优雅的失败：对解析器、编译器和执行引擎都进行了内存限制。但regexp不支持以下正则表达式特性，如业务依赖这些特性，则regexp不适合使用。
    - 回溯引用Backreferences和查看Lookaround

```
// good
matched, err := regexp.MatchString(`a.b`, "aaxbb")
fmt.Println(matched) // true
fmt.Println(err)     // nil (regexp is valid)
```

后台类

输入校验

SQL操作

网络请求

服务器端渲染

Web跨域

响应输出

会话管理

访问控制

并发保护

输入校验

输入校验

1.【必须】按类型进行数据校验

- 所有外部输入的参数，应使用`validator`进行白名单校验，校验内容包括但不限于数据长度、数据范围、数据类型与格式，校验不通过的应当拒绝

```
// good
import (
    "fmt"
    "github.com/go-playground/validator/v10"
)

var validate *validator.Validate
validate = validator.New()
func validateVariable() {
    myEmail := "abc@tencent.com"
    errs := validate.Var(myEmail, "required,email")
    if errs != nil {
        fmt.Println(errs)
        return
        //停止执行
    }
    // 验证通过，继续执行
    ...
}
```

- 无法通过白名单校验的应使用`html.EscapeString`、`text/template`或`bluemonday`对`<, >, &, ',"`等字符进行过滤或编码

```
  import(
      "text/template"
  )

  // TestHTMLEscapeString HTML特殊字符转义
  func main(inputValue string) string{
      escapedResult := template.HTMLEscapeString(inputValue)
```

```
    return escapedResult
}
```

SQL操作

SQL操作

1.【必须】SQL语句默认使用预编译并绑定变量

- 使用 `database/sql` 的prepare、Query或使用GORM等ORM执行SQL操作

```
  import (
    "github.com/jinzhu/gorm"
    _ "github.com/jinzhu/gorm/dialects/sqlite"
  )

  type Product struct {
    gorm.Model
    Code string
    Price uint
  }
  ...
  var product Product
  db.First(&product, 1)
```

- 使用参数化查询，禁止拼接SQL语句，另外对于传入参数用于order by或表名的需要通过校验

```
// bad
  import (
      "database/sql"
      "fmt"
      "net/http"
  )

  func handler(db *sql.DB, req *http.Request) {
      q := fmt.Sprintf("SELECT ITEM,PRICE FROM PRODUCT WHERE ITEM_CATEGORY='%s'
 ORDER BY PRICE",
          req.URL.Query()["category"])
      db.Query(q)
  }

// good
func handlerGood(db *sql.DB, req *http.Request) {
```

```
    //使用?占位符
        q := "SELECT ITEM,PRICE FROM PRODUCT WHERE ITEM_CATEGORY='?' ORDER BY PRIC
E"
        db.Query(q, req.URL.Query()["category"])
}
```

网络请求

网络请求

1.【必须】资源请求过滤验证

- 使用 `"net/http"` 下的方法 `http.Get(url)` 、 `http.Post(url, contentType, body)` 、 `http.Head(url )` 、 `http.PostForm(url, data)` 、 `http.Do(req)` 时，如变量值外部可控（指从参数中动态获取），应对请求目标进行严格的安全校验。

- 如请求资源域名归属固定的范围，如只允许 `a.qq.com` 和 `b.qq.com` ，应做白名单限制。如不适用白名单，则推荐的校验逻辑步骤是：

    - 第 1 步、只允许HTTP或HTTPS协议

    - 第 2 步、解析目标URL，获取其HOST

    - 第 3 步、解析HOST，获取HOST指向的IP地址转换成Long型

    - 第 4 步、检查IP地址是否为内网IP，网段有：

        ```
        // 以RFC定义的专有网络为例，如有自定义私有网段亦应加入禁止访问列表。
        10.0.0.0/8
        172.16.0.0/12
        192.168.0.0/16
        127.0.0.0/8
        ```

    - 第 5 步、请求URL

    - 第 6 步、如有跳转，跳转后执行1，否则绑定经校验的ip和域名，对URL发起请求

- 官方库 `encoding/xml` 不支持外部实体引用，使用该库可避免xxe漏洞

```
import (
    "encoding/xml"
    "fmt"
    "os"
```

```
)

func main() {
    type Person struct {
        XMLName   xml.Name `xml:"person"`
        Id        int      `xml:"id,attr"`
        UserName string   `xml:"name>first"`
        Comment string `xml:",comment"`
    }

    v := &Person{Id: 13, UserName: "John"}
    v.Comment = " Need more details. "

    enc := xml.NewEncoder(os.Stdout)
    enc.Indent("  ", "    ")
    if err := enc.Encode(v); err != nil {
        fmt.Printf("error: %v\n", err)
    }

}
```

服务器端渲染

服务器端渲染

1.【必须】模板渲染过滤验证

- 使用 `text/template` 或者 `html/template` 渲染模板时禁止将外部输入参数引入模板，或仅允许引入白名单内字符。

```
   // bad
    func handler(w http.ResponseWriter, r *http.Request) {
      r.ParseForm()
      x := r.Form.Get("name")

      var tmpl = `<!DOCTYPE html><html><body>
    <form action="/" method="post">
        First name:<br>
    <input type="text" name="name" value="">
    <input type="submit" value="Submit">
    </form><p>` + x + ` </p></body></html>`

      t := template.New("main")
      t, _ = t.Parse(tmpl)
      t.Execute(w, "Hello")
    }

// good
    import (
        "fmt"
        "github.com/go-playground/validator/v10"
    )

    var validate *validator.Validate
    validate = validator.New()
    func validateVariable(val) {
        errs := validate.Var(val, "gte=1,lte=100")//限制必须是1-100的正整数
        if errs != nil {
            fmt.Println(errs)
            return False
        }
        return True
    }
```

```
    func handler(w http.ResponseWriter, r *http.Request) {
        r.ParseForm()
        x := r.Form.Get("name")

        if validateVariable(x):
            var tmpl = `<!DOCTYPE html><html><body>
            <form action="/" method="post">
            First name:<br>
            <input type="text" name="name" value="">
            <input type="submit" value="Submit">
            </form><p>` + x + ` </p></body></html>`
            t := template.New("main")
            t, _ = t.Parse(tmpl)
            t.Execute(w, "Hello")
        else:
            ...
    }
```

Web跨域

Web跨域

1.【必须】跨域资源共享CORS限制请求来源

- CORS请求保护不当可导致敏感信息泄漏，因此应当严格设置Access-Control-Allow-Origin使用同源策略进行保护。

```
// good
  c := cors.New(cors.Options{
      AllowedOrigins: []string{"http://qq.com", "https://qq.com"},
      AllowCredentials: true,
      Debug: false,
  })

  //引入中间件
  handler = c.Handler(handler)
```

响应输出

响应输出

1.【必须】设置正确的HTTP响应包类型

- 响应头Content-Type与实际响应内容，应保持一致。如：API响应数据类型是json，则响应头使用`application/json`；若为xml，则设置为`text/xml`。

2.【必须】添加安全响应头

- 所有接口、页面，添加响应头`X-Content-Type-Options: nosniff`。
- 所有接口、页面，添加响应头`X-Frame-Options`。按需合理设置其允许范围，包括：`DENY`、`SAMEORIGIN`、`ALLOW-FROM origin`。用法参考：MDN文档

3.【必须】外部输入拼接到HTTP响应头中需进行过滤

- 应尽量避免外部可控参数拼接到HTTP响应头中，如业务需要则需要过滤掉`\r`、`\n`等换行符，或者拒绝携带换行符号的外部输入。

4.【必须】外部输入拼接到response页面前进行编码处理

- 直出html页面或使用模板生成html页面的，推荐使用`text/template`自动编码，或者使用`html.EscapeString`或`text/template`对`<, >, &, ',"`等字符进行编码。

```
import(
    "html/template"
)

func outtemplate(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)
}
```

会话管理

会话管理

1.【必须】安全维护session信息

- 用户登录时应重新生成session，退出登录后应清理session。

```
import (
    "net/http"
    "github.com/gorilla/mux"
    "github.com/gorilla/handlers"
)

//创建cookie
func setToken(res http.ResponseWriter, req *http.Request) {
    expireToken := time.Now().Add(time.Minute * 30).Unix()
    expireCookie := time.Now().Add(time.Minute * 30)
    ...
    cookie := http.Cookie{
        Name: "Auth",
        Value: signedToken,
        Expires: expireCookie, // 过期失效
        HttpOnly: true,
        Path: "/",
        Domain: "127.0.0.1",
        Secure: true
    }

    http.SetCookie(res, &cookie)
    http.Redirect(res, req, "/profile", 307)
}
// 删除cookie
func logout(res http.ResponseWriter, req *http.Request) {
    deleteCookie := http.Cookie{
        Name: "Auth",
        Value: "none",
        Expires: time.Now()
    }
    http.SetCookie(res, &deleteCookie)
    return
}
```

2.【必须】CSRF防护

- 涉及系统敏感操作或可读取敏感信息的接口应校验 `Referer` 或添加 `csrf_token` 。

```
// good
import (
    "net/http"
    "github.com/gorilla/csrf"
    "github.com/gorilla/mux"
)

func main() {
    r := mux.NewRouter()
    r.HandleFunc("/signup", ShowSignupForm)
    r.HandleFunc("/signup/post", SubmitSignupForm)
    //使用csrf_token验证
    http.ListenAndServe(":8000",
        csrf.Protect([]byte("32-byte-long-auth-key"))(r))
}
```

访问控制

访问控制

1.【必须】默认鉴权

- 除非资源完全可对外开放，否则系统默认进行身份认证，使用白名单的方式放开不需要认证的接口或页面。
- 根据资源的机密程度和用户角色，以最小权限原则，设置不同级别的权限，如完全公开、登录可读、登录可写、特定用户可读、特定用户可写等
- 涉及用户自身相关的数据的读写必须验证登录态用户身份及其权限，避免越权操作

```
-- 伪代码
select id from table where id=:id and userid=session.userid
```

- 没有独立账号体系的外网服务使用 `QQ` 或 `微信` 登录，内网服务使用 `统一登录服务` 登录，其他使用账号密码登录的服务需要增加验证码等二次验证

并发保护

并发保护

1.【必须】禁止在闭包中直接调用循环变量

- 在循环中启动协程，当协程中使用到了循环的索引值，由于多个协程同时使用同一个变量会产生数据竞争，造成执行结果异常。

```
// bad
func main() {
    runtime.GOMAXPROCS(runtime.NumCPU())
    var group sync.WaitGroup

    for i := 0; i < 5; i++ {
        group.Add(1)
        go func() {
            defer group.Done()
            fmt.Printf("%-2d", i) //这里打印的i不是所期望的
        }()
    }
    group.Wait()
}

// good
func main() {
    runtime.GOMAXPROCS(runtime.NumCPU())
    var group sync.WaitGroup

    for i := 0; i < 5; i++ {
        group.Add(1)
        go func(j int) {
            defer func() {
                if r := recover(); r != nil {
                    fmt.Println("Recovered in start()")
                }
                group.Done()
            }()
        fmt.Printf("%-2d", j) // 闭包内部使用局部变量
        }(i)  // 把循环变量显式地传给协程
    }
```

```
    group.Wait()
}
```

2.【必须】禁止并发写map

- 并发写map容易造成程序崩溃并异常退出，建议加锁保护

```
// bad
func main() {
  m := make(map[int]int)
  //并发读写
  go func() {
      for {
          _ = m[1]
      }
  }()
  go func() {
      for {
          m[2] = 1
      }
  }()
  select {}
}
```

3.【必须】确保并发安全

敏感操作如果未作并发安全限制，可导致数据读写异常，造成业务逻辑限制被绕过。可通过同步锁或者原子操作进行防护。

通过同步锁共享内存

```
// good
var count int
func Count(lock *sync.Mutex) {
    lock.Lock()// 加写锁
    count++
    fmt.Println(count)
    lock.Unlock()// 解写锁，任何一个Lock()或RLock()均需要保证对应有Unlock()或RUnlock()
}

func main() {
    lock := &sync.Mutex{}
    for i := 0; i < 10; i++ {
        go Count(lock) //传递指针是为了防止函数内的锁和调用锁不一致
```

```
    }
    for {
        lock.Lock()
        c := count
        lock.Unlock()
        runtime.Gosched()//交出时间片给协程
        if c > 10 {
            break
        }
    }
}
```

- 使用 `sync/atomic` 执行原子操作

```
// good
import (
    "sync"
    "sync/atomic"
)

func main() {
    type Map map[string]string
    var m atomic.Value
    m.Store(make(Map))
    var mu sync.Mutex // used only by writers
    read := func(key string) (val string) {
        m1 := m.Load().(Map)
        return m1[key]
    }
    insert := func(key, val string) {
        mu.Lock() // 与潜在写入同步
        defer mu.Unlock()
        m1 := m.Load().(Map) // 导入struct当前数据
        m2 := make(Map)      // 创建新值
        for k, v := range m1 {
            m2[k] = v
        }
        m2[key] = val
        m.Store(m2)    // 用新的替代当前对象
    }
    _, _ = read, insert
}
```