

Тестовое задание на стажировку AppSecCloudCamp

Отчёт о выполненной работе

1. Вопросы для разогрева

- Расскажите, с какими задачами в направлении безопасной разработки вы сталкивались?
- Если вам приходилось проводить security code review или моделирование угроз, расскажите, как это было?
- Если у вас был опыт поиска уязвимостей, расскажите, как это было?
- Почему вы хотите участвовать в стажировке?

2. Security code review

Часть 1. Security code review: GO

Рассмотрев представленный в примере код, я нашёл только одну уязвимость. В функции «searchHandler» выполняется запрос к базе данных, на выходе мы получаем список товаров. Сам SQL запрос можем увидеть на 38 строчке, где из заголовка GET-запроса приложение получает наименование товара и совмещает это с заготовленным запросом. Рассмотрев всё это, можно сказать что данный фрагмент кода уязвим к SQLi. Данная уязвимость позволяет злоумышленнику изменять содержимое SQL запроса и получать данные, к которым он не имеет доступ.

```

26 func searchHandler(w http.ResponseWriter, r *http.Request) {
27     if r.Method != "GET" {
28         http.Error(w, "Method is not supported.", http.StatusNotFound)
29         return
30     }
31
32     searchQuery := r.URL.Query().Get("query")
33     if searchQuery == "" {
34         http.Error(w, "Query parameter is missing", http.StatusBadRequest)
35         return
36     }
37
38     query := fmt.Sprintf("SELECT * FROM products WHERE name LIKE '%s'", searchQuery)
39     rows, err := db.Query(query)
40     if err != nil {
41         http.Error(w, "Query failed", http.StatusInternalServerError)
42         log.Println(err)
43         return
44     }
45     defer rows.Close()
46
47     var products []string
48     for rows.Next() {
49         var name string
50         err := rows.Scan(&name)
51         if err != nil {
52             log.Fatal(err)
53         }
54         products = append(products, name)
55     }
56
57     fmt.Fprintf(w, "Found products: %v\n", products)
58 }

```

Для устранения данной уязвимости можем предоставлять значения параметров SQL в качестве аргументов функции, так как пакет `sql` в `golang` осуществляет проверку на данную уязвимость. Решая данную задачу, я даже наткнулся на страницу документации `golang`, где представлен пример как не следует выполнять SQL запросы, ознакомиться можно по ссылке <https://go.dev/doc/database/sql-injection>. Исправленный фрагмент кода представлен на рисунке №.

```

38 rows, err := db.Query("SELECT * FROM products WHERE name LIKE ?", searchQuery)
39 if err != nil {
40     http.Error(w, "Query failed", http.StatusInternalServerError)
41     log.Println(err)
42     return
43 }
44 defer rows.Close()

```

Часть 2. Security code review: Python

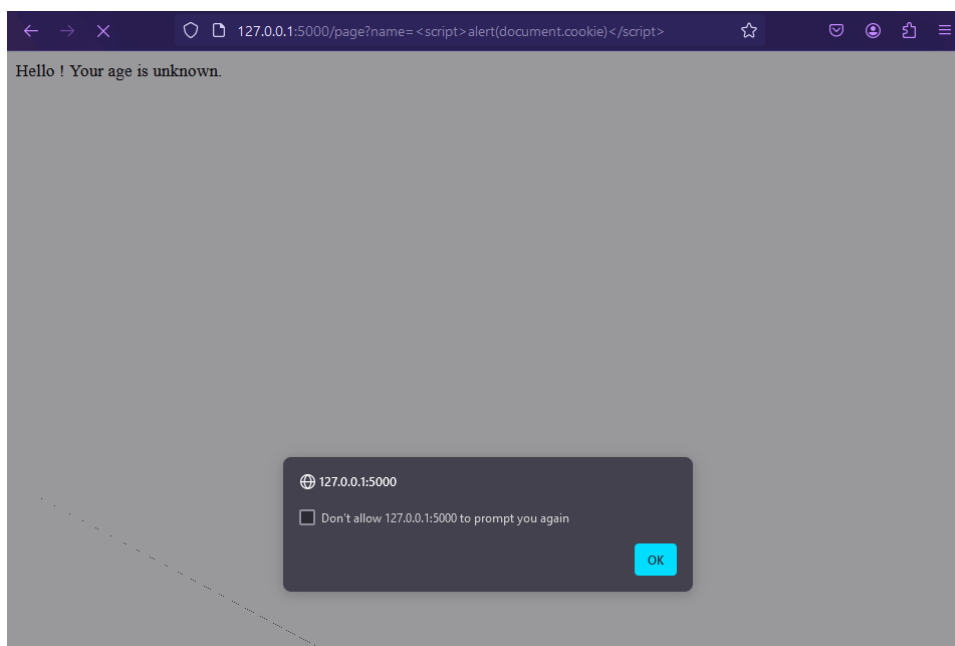
Рассмотрим пример 2.1, код которого представлен на рисунке №. После анализа, я заметил возможность наличия уязвимости на строчках 9-11, где по значениям в заголовке GET-запроса с аргументами «name», «age» и «unknown» устанавливаются значения переменным «name» и «age». После чего, данное веб-приложение отображает выводимое этих переменных без каких-либо проверок, что позволяет нам воспользоваться XSS атакой.

```

1  from flask import Flask, request
2  from jinja2 import Template
3
4  name = "main"
5  app = Flask(name)
6
7  @app.route("/page")
8  def page():
9      name = request.values.get('name')
10     age = request.values.get('age', 'unknown')
11     output = Template('Hello ' + str(name) + '! Your age is ' + age + '.').render()
12     return output
13
14 if name == "main":
15     app.run(debug=True)

```

Запустив локально программу я проверил наличие XSS-уязвимости, указав в качестве имени «<script>alert(document.cookie)</script>», что привело к результату, представленному на рисунке №. Данная уязвимость может позволить злоумышленнику провести кражу авторизационных cookie пользователей веб-приложения.



Для устранения данной уязвимости, я решил производить проверку вводимых параметров на специальные символы. В нашем случае приложение получает имя и возраст, так что возможность удаления лишних символов из имени исключена. На рисунке № представлен код с проверкой вводимых значений. После чего я повторно запустил приложения и провёл попытку XSS-атаки, результат представлен на рисунке №.

```

1  from flask import Flask, request
2  from jinja2 import Template
3
4  def parse_input_value(value : str) -> str:
5      forbidden_chrs = set(['<>/&%'])
6      return ''.join([c for c in value if c not in forbidden_chrs])
7
8
9  name = "main"
10 app = Flask(name)
11
12 @app.route("/page")
13 def page():
14     name = parse_input_value(str(request.values.get('name')))
15     print(name)
16     age = parse_input_value(request.values.get('age', 'unknown'))
17     output = Template('Hello ' + name + '! Your age is ' + age + '.').render()
18     return output
19
20 if name == "main":
21     app.run(debug=True)

```

127.0.0.1:5000/page?name= <script> alert(document.cookie)</script>

Hello scriptalert(document.cookie)script! Your age is unknown.

Также как вариант устранения уязвимости, можем воспользоваться библиотекой escape. Она не позволяет выполнять код в данных, переданных пользователем. Реализация представлена на рисунке №, результат попытки атаки на рисунке №.

```

1  from flask import Flask, request
2  from jinja2 import Template
3  from markupsafe import escape
4
5  name = "main"
6  app = Flask(name)
7
8  @app.route("/page")
9  def page():
10     name = escape(request.values.get('name'))
11     print(name)
12     age = escape(request.values.get('age', 'unknown'))
13     output = Template('Hello ' + name + '! Your age is ' + age + '.').render()
14     return output
15
16 if name == "main":
17     app.run(debug=True)

```

127.0.0.1:5000/page?name= <script> alert(document.cookie)</script>

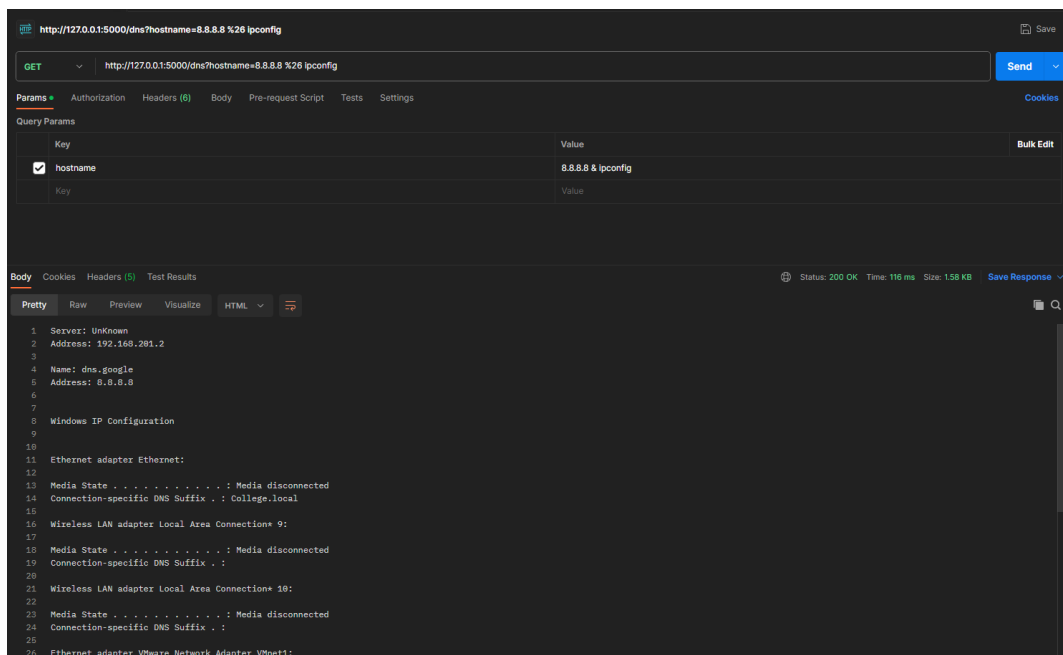
Hello <script>alert(document.cookie)</script>! Your age is unknown.

Если выбирать наилучший вариант реализации исправления данной уязвимости, то я бы выбрал второй вариант с использованием дополнительного функционала flask. По сути обе реализации осуществляют похожие действия, они обе проверяют на наличие дополнительных символов, но реализации flask не удаляет эти символы, а заменяет их на безопасные для HTML последовательности.

Рассмотрим пример 2.2, код которого представлен на рисунке №. На строках 8-11 я обнаружил возможную уязвимость. На 8 строчке приложение берёт значение «hostname» из заголовка GET-запроса пользователя, после чего совмещает с командой «nslookup» и запускает в командной строке. Результат выполнения команды отображается на сайте.

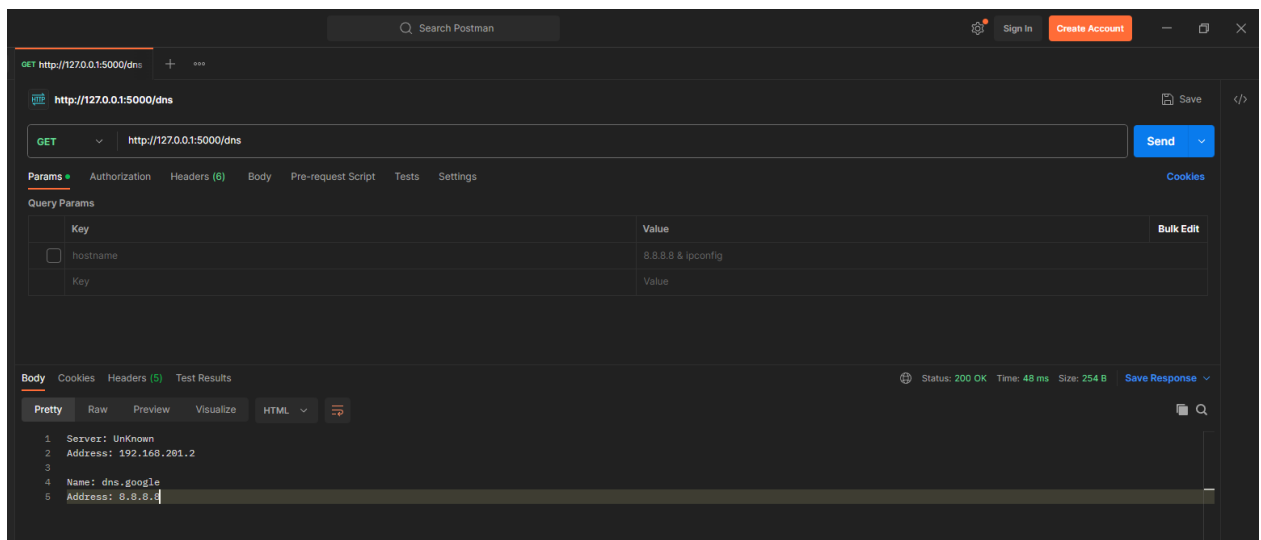
```
1  from flask import Flask, request
2  import subprocess
3
4  app = Flask(name)
5
6  @app.route("/dns")
7  def dns_lookup():
8      hostname = request.values.get('hostname')
9
10     cmd = 'nslookup ' + hostname
11     output = subprocess.check_output(cmd, shell=True, text=True)
12     return output
13
14 if name == "main":
15     app.run(debug=True)
```

Запустив данное приложение локально, я отправил запрос, указав в качестве «hostname» значение «8.8.8.8 & ipconfig», тем самым получаю информацию о сетевых устройствах системы, на котором работает данный сервис. Злоумышленник может воспользоваться данной уязвимостью для запуска любого кода на системе сервера, имея соответствующие права. Вкратце, получаем полный доступ к терминалу хоста, после чего можем даже настроить reverse shell, тем самым уже получаем полный доступ к системе от прав сервера.



Для устранения данной уязвимости, я решил провести проверку на вводимых значениях «hostname», а именно удалять всю часть строки после специального символа «&» и «;». Так, при любом запросе от пользователя, будет выполняться только команда «nslookup». Код представлен на рисунке №, результат попытки эксплуатировать уязвимость после изменений представлен на рисунке №.

```
1  from flask import Flask, request
2  import subprocess
3
4  name = "main"
5  app = Flask(name)
6
7  @app.route("/dns")
8  def dns_lookup():
9      hostname = request.values.get('hostname')
10     cmd = 'nslookup ' + str(hostname).split("&")[0].split(";")[0]
11     output = subprocess.check_output(cmd, shell=True, text=True)
12     return output
13
14  if name == "main":
15     app.run(debug=True)
```



Также, в качестве решения уязвимости можно аналогично первому примеру на Python: удалять специальные символы из вводимого значения пользователя.