SBD Laboratory - Solutions

Thomas Gingele

2023-10-09

Task 1

```
POST /WebGoat/HttpProxies/intercept-request HTTP/1.1
Host: localhost:8080
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:109.0) Gecko/20100101
    Firefox/115.0
Accept: */ *
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate, br
Content-Type: application/x-www-form-urlencoded; charset=UTF-8
X-Requested-With: XMLHttpRequest
Content-Length: 15
Origin: http://localhost:8080
Connection: close
Referer: http://localhost:8080/WebGoat/start.mvc
Cookie: JSESSIONID=e3dHiM5wF8CB2DJW6Sb_K1NAYbCAcl3W8PONY_oD;
    WEBWOLFSESSION=YR6BTRUQH_89HzCbp9q68HiWVQGdYCgWyDWVW7UL
Sec-Fetch-Dest: empty
Sec-Fetch-Mode: cors
Sec-Fetch-Site: same-origin

changeMe=haxx0r
```

Aufgabe 2

- `Host` -> This is the websites host. In this case `localhost` as the application is hosted locally.
- `User-Agent` -> Contains some basic information about the program that is being used to interact with the website. This may help with displaying the website correctly.
- `Accept` -> This is the media type that the browser will accept in the response.
- `Accept-Language` -> This is the language/s that the browser will accept in the response.
- `Accept-Encoding` -> These are the different types of encoding that the browser will accept in the response.
- `Content-Type` -> Describes what format is used to send data to the server. Only used in POST/PUT requests.
- `Origin` -> Enables Cross-Origin Resource Sharing, allowing a client to access otherwise restricted resources from a different domain then the resource is hosted on.
- `Connection` -> Decides whether the connection with the web server should be held open or be closed.
- `Referer` -> The URL at which the client was located when sending the request.
- `Cookie` -> Contains one or more cookie/s previously set by a `Set-Cookie` header by the server. Cookies have multiple purposes from serving as credentials to preventing basic brute force attacks and more.

- `Sec-Fetch-Dest` -> Can be used to let the server know what the response will be used for. This can help with formatting the reponse for the expected use case on the server.
- `Sec-Fetch-Mode` -> This header is used to distinguish between different uses for the response, for example whether it is for a user navigating a website or to load an image and so on.
- `Sec-Fetch-Site` -> Through this header, a client can tell the server whether a request is coming from the site itself, from a different site or from a completely user-generated request.

Task 3

WebGoat is using **Javascript** and **Java 17/Maven**.

Exposed Javascript frameworks:

- Backbone.js 1.4.0
- RequireJS 2.3.6

Exposed Javascript Libraries:

- jQuery 3.5.1
- jQuery UI 1.10.4
- Underscore.js version unknown

Task 4

The name of the input field is `changeMe`.

Task 5

When intercepting the request that was earlier used to send a mail to WebWolf and changing the target address to `idont@exist.welp`, the reponse looks like this:

```
HTTP/1.1 200 OK
Connection: close
X-XSS-Protection: 1; mode=block
X-Content-Type-Options: nosniff
X-Frame-Options: DENY
Content-Type: application/json
Date: Mon, 09 Oct 2023 18:50:57 GMT

{
  "lessonCompleted" : false,
```

```
11    "feedback" : "Of course you can send mail to user idont however you
          will not be able to read this e-mail in WebWolf, please use your
          own username.",
12    "output" : null,
13    "assignment" : "MailAssignment",
14    "attemptWasMade" : false
15  }
```

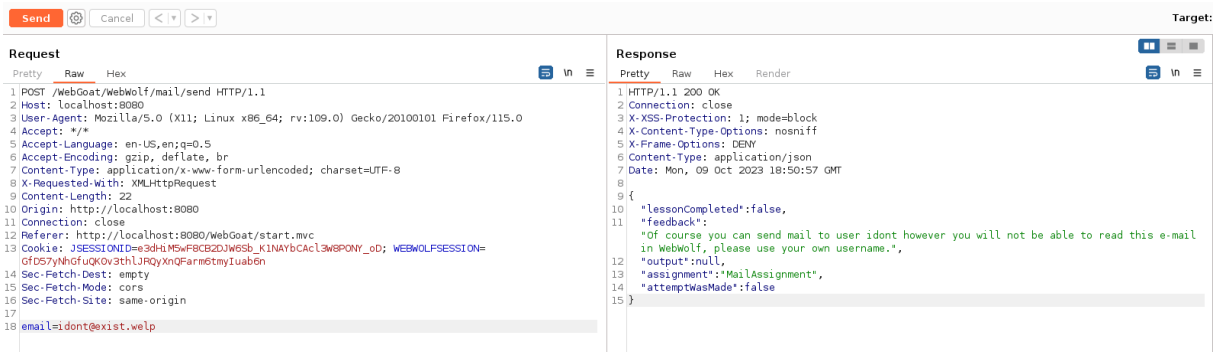

Figure 1: HTTP Proxies Task 7

Task 6

Since I feel quite comfortable with the developer tools already, I will simply present the solution to the WebGoat tasks here.

Developer Tools Section 4

Use the console in the dev tools and call the javascript function webgoat.customjs.phoneHome().

The answer for this question is randomly generated each time the function is called. Simply open your browsers developer console and run it to receive the result.

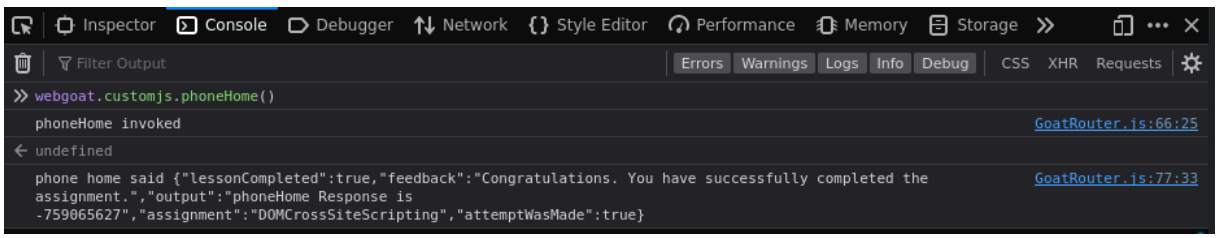

Figure 2: Developer Tools Task 4

Developer Tools Section 6

In this assignment you need to find a specific HTTP request and read a randomized number from it.

Open the “Network” tab in your browsers developer tools and take a look at the request body of the request. Note, that this answer is also randomly generated each time the request is send.

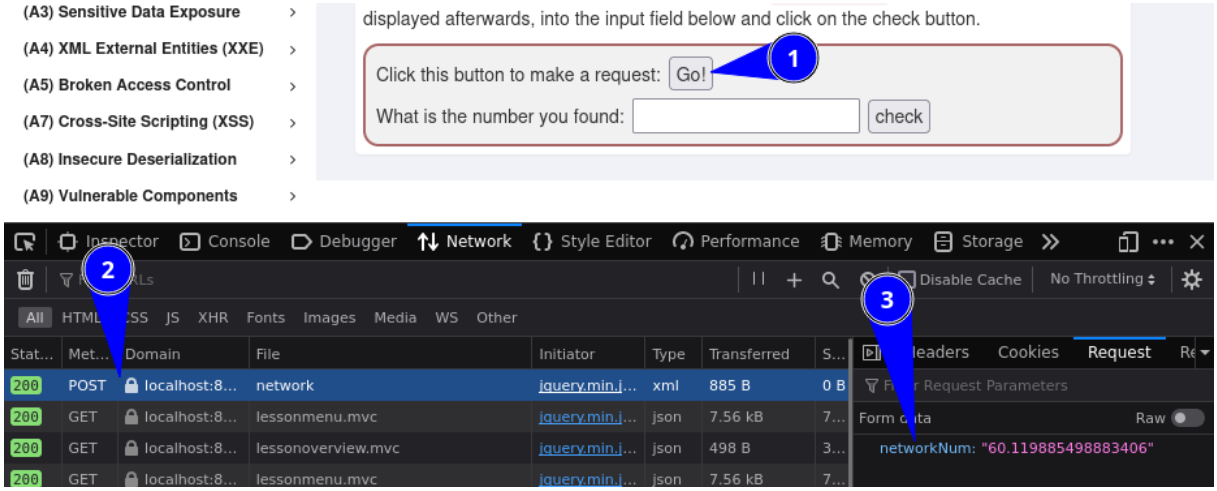

Figure 3: Developer Tools Task 6

Task 7

I did not understand this task. (TODO)

Task 8

1. *How could an intruder harm the security goal of confidentiality?*

Solution 3: By stealing a database where names and emails are stored and uploading it to a website.

2. *How could an intruder harm the security goal of integrity?*

Solution 1: By changing the names and emails of one or more users stored in a database.

3. *How could an intruder harm the security goal of availability?*

Solution 4: By launching a denial of service attack on the servers.

4. *What happens if at least one of the CIA security goals is harmed?*

Solution 2: The system’s security is compromised even if only one goal is harmed.

Task 9

Base64 works by taking any binary input and splitting it into a 6-bit character representation. Usually, this is used when it is necessary to transfer binary files like images via text, but it has also been adapted for encoding of large junks of text data, like cookies or some headers in HTTP/S requests.

Task 10

Base64 has one flaw: If the original size of the binary data is not a multiple of three, there may be some empty bytes. This is solved by appending enough empty bytes to the end of the input to pad the data to a 3-byte-multiple. Since empty bytes cannot be natively encoded with Base64, they are represented through = signs at the end of the encoded data.

Task 11

HTTP Basic Authentication uses Base64 encoding to transfer a clients credentials through a request header called `Authorization`. This header contains the credentials in the following format:

```
<username>:<password>
```

This is then encoded and prepended with the string “Basic” to let the server know what type of authorization is used. A full header would look like this:

```
Authorization: Basic eW91d2lzaHRoaXN3YXNteXBhc3N3b3JkOmJ1dG5vcGUK
```

Task 12

When storing passwords using XOR, the key is repeated until it fits the length of the input. This lengthened key is then XORed with this input to create the cipher.

XOR is a very simple encoding that simply compares both strings bit by bit and outputs `1` if the bits are *not* equal and `0` if they are.

Input Bit	Key Bit	Output Bit
0	0	0
0	1	1
1	0	1

Input Bit	Key Bit	Output Bit
1	1	0

Becaues of this, if a XOR encoded string is XORed with the same key again, it will be decoded.

Task 13

XOR can be attacked in multiple ways. Most importantly, any secret key can be reconstructed from the plaintext and cipher, simply by XORing them together. Additionally, since XOR ciphers repeat a key in order to stretch it to the size of the plaintext, multiple keys are valid for the same cipher.

Task 14

The script can be found on my Github as well: xor_plaintext_attack - Github

```python
# Copyright 2023 Thomas Gingele https://github.com/B1TC0R3
import argparse


def get_args() -> argparse.Namespace:
    parser = argparse.ArgumentParser(
        prog='XOR Plaintext Cracker',
        description='This script will attempt to find the secret key of a XOR encoded string by performing a plaintext attack',
        epilog='Copyright 2023 Thomas Gingele https://github.com/B1TC0R3'
    )

    parser.add_argument(
        '-c',
        '--cipher',
        help='The XOR encoded string',
        required=True
    )

    parser.add_argument(
        '-p',
        '--plaintext',
        help='The plaintext string',
        required=True
    )

    return parser.parse_args()
```

```python


def xor(cipher, key) -> bytearray:
    return bytearray(
        a ^ b for a, b in zip(*map(bytearray, [cipher, key]))
    )


def main():
    args      = get_args()
    cipher    = args.cipher.encode('utf-8')
    plaintext = args.plaintext.encode('utf-8')
    key       = str(xor(cipher, plaintext), 'utf-8')

    print(key)


if __name__ == '__main__':
    main()
```

Task 15

Crypto Basics Section 3

The following string needs to be decoded:

```
{xor}Oz4rPj0+LDovPiwsKDAtOw==
```

Apart from the `{xor}`, this seems to be encoded with `base64`. Using the command from *Section 2*, it decodes to:

```
;>+>=>,:/>,,(0-;
```

The `{xor}` may be a hint to how this string is encoded. It is possible to brute force XOR encoded strings using CyberChef - Github.io.

The string decodes to:

```
databasepassword
```

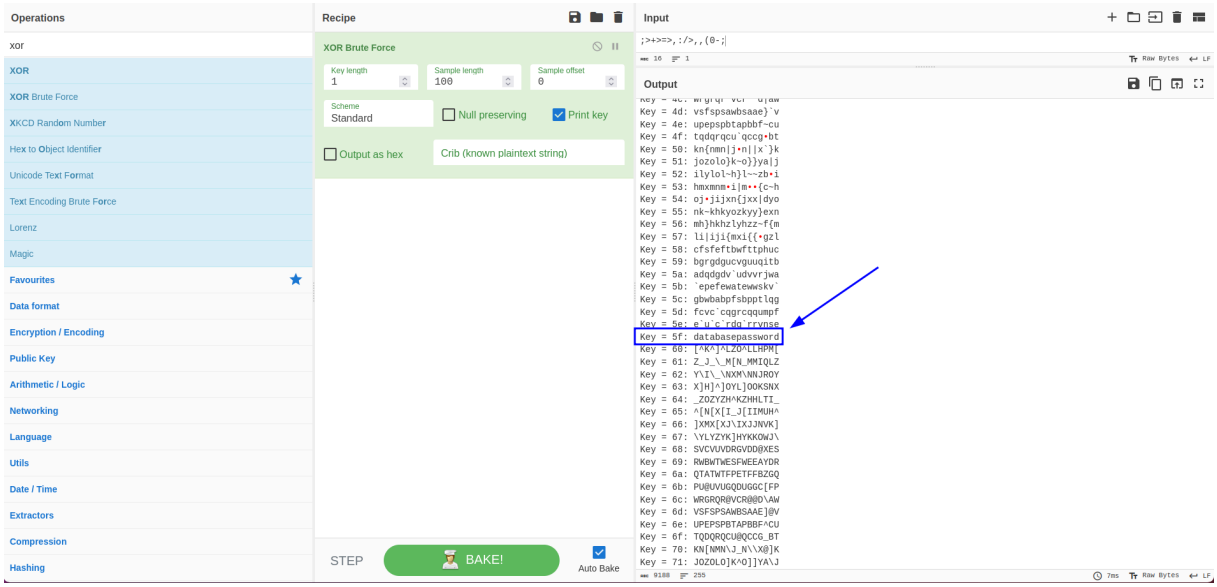

Figure 4: Crypto Basics Task 3

Alternatively, the script from **Task 14** can be used here.

Task 16

There are two approches to cracking hashes:

1. Looking up the hash in an online database. This method only works on non-salted hashes and is also ineffective against more modern hashing methods like `bcrypt`.
2. Cracking the hash with a wordlist or rainbowtable. With this approach, it may be easier to get a result but the success rate is higher. It is possible to attack salted hashes using rainbow tables and the hash method does not matter either. The biggest downside of cracking a hash like this is the time required to finish the computation, as brute forcing tends to be rather slow.

Task 17

The first hash is a `MD5` hash, the second one a `SHA256` hash.

When locally cracking the hash using `john`, both take less then a second to crack. Using crackstation.net, cracking both hashes at once takes *446 milliseconds*. A large portion of this time likely is lost transmitting the request and response.

Task 18

The final version of this script can be found here: sbd_lab_1_t1820.py - Github.com

```python
# Copyright 2023 Thomas Gingele https://github.com/B1TC0R3
import argparse
from Crypto.PublicKey import RSA


def get_args() -> argparse.Namespace:
    parser = argparse.ArgumentParser(
        prog='Solver for SBD Laboratory Task 18/20',
        epilog='Copyright 2023 Thomas Gingele https://github.com/B1TC0R3'
    )

    parser.add_argument(
        '-priv',
        '--private-key',
        help='a RSA private key file',
        required=True
    )

    return parser.parse_args()


def openssl_format(value: int) -> str:
    return hex(value)[2:].upper()


def main():
    args        = get_args()
    private_key = None

    with open(args.private_key) as key_file:
        private_key = RSA.import_key(key_file.read())

    # Remove prepending '0x' from hex string and uppercase it to match OpenSSL output
    modulus          = openssl_format(private_key.n)
    public_exponent  = openssl_format(private_key.d)
    private_exponent = openssl_format(private_key.e)

    print(f"MODULUS: {modulus}\n")
    print(f"PUBLIC EXPONENT: {public_exponent}\n")
    print(f"PRIVATE EXPONENT: {private_exponent}\n")


if __name__ == '__main__':
    main()
```

Task 19

This task can be finished with the command:

```
openssl rsa -in myprivate.pem -pubout -out mypublic.key
```

Task 20

The script can also be found here: sbd_lab_1_t1820.py - Github.com

```
# Copyright 2023 Thomas Gingele https://github.com/B1TC0R3
import argparse
import base64
from Crypto.PublicKey import RSA
from Crypto.Hash      import SHA256
from Crypto.Signature import pkcs1_15


def get_args() -> argparse.Namespace:
    parser = argparse.ArgumentParser(
        prog='Solver for SBD Laboratory Task 18/20',
        epilog='Copyright 2023 Thomas Gingele https://github.com/
           B1TC0R3'
    )

    parser.add_argument(
        '-priv',
        '--private-key',
        help='a RSA private key file',
        required=True
    )

    parser.add_argument(
        '-pub',
        '--public-key',
        help='a RSA public key file',
        required=True
    )

    return parser.parse_args()


def openssl_format(value: int) -> str:
    return hex(value)[2:].upper()


def main():
    args        = get_args()
    private_key = None
```

```
39      public_key  = None
40
41      with open(args.private_key) as key_file:
42          private_key = RSA.import_key(key_file.read())
43
44      with open(args.public_key) as key_file:
45          public_key = RSA.import_key(key_file.read())
46
47      signer   = pkcs1_15.new(private_key)
48      verifier = pkcs1_15.new(public_key)
49
50      # Remove prepending '0x' from hex string and uppercase it to match
            OpenSSL output
51      modulus          = openssl_format(private_key.n)
52      public_exponent  = openssl_format(private_key.d)
53      private_exponent = openssl_format(private_key.e)
54
55      data      = SHA256.new(modulus.encode())
56      signature = signer.sign(data)
57      b64_sign  = base64.b64encode(signature).decode()
58
59      # This will raise a 'ValueError' if the signature is invalid
60      verifier.verify(data, signature)
61
62      print(f"MODULUS: {modulus}\n")
63      print(f"PUBLIC EXPONENT: {public_exponent}\n")
64      print(f"PRIVATE EXPONENT: {private_exponent}\n")
65      print(f"SIGNATURE: {b64_sign}\n")
66
67
68  if __name__ == '__main__':
69      main()
```

Task 21

The modulus can be extracted from a private key file with `openssl`.

```
1  openssl rsa -in myprivate.pem -noout -modulus
```

This value can then be signed using the same parameters as the Python script.

```
1  openssl rsa -in myprivate.pem -noout -modulus | cut -d '=' -f 2 | tr -d
       '\n' | openssl dgst -sha256 -sign myprivate.pem | base64 >
     signature.txt
```

Task 22

The signature produced and validated by the Python script is the same as the signature contained in `signature.txt`.

```
asLKttVADwMG+L8FqPih7oKg9WbBdHEeACgW7Yx
/GE4cWPZUkZtTk+lJPOGLWqWoJmN46ZHfkI7WBN
K5YHdBnUQjgSsS1PuraSYqoIWRUqOksWllshvxw
F6jKk6TCzvJBT5jmPx9xV0nEmnNZuYFyOJm4r8w
1yKo+HerAdidXzqTjkNH6HcHJHl3mjWVDecadNH
ABEVKjS5KHM45YTIj9idZHxUbfaiDbacHQmtVqe
ZDMypQsL3kNLbDOxWqIQ9D7qOWCoMFzmcSDOJnB
9YRDWlo77nWazYvx7hRMcMs8X3s6wCy0C5AGjuC
brydXjzckz9yxksSIs77TMZSilX1sA==
```