CT437
COMPUTER SECURITY AND FORENSIC COMPUTING

HASH CRACKING AND RAINBOW TABLES

Dr. Michael Schukat

Ollscoil na Gaillimhe
University of Galway

Lecture Overview

- Methods to reverse-engineer hashed passwords
- Rainbow tables
- A recap on SQL injection attacks (based on CT417 content), i.e.
  - SQL
  - HTTP get / post Methods and PHP
  - SQL injection attacks
  - SQL injection attack mitigation strategies

can be found at the end of this slide deck

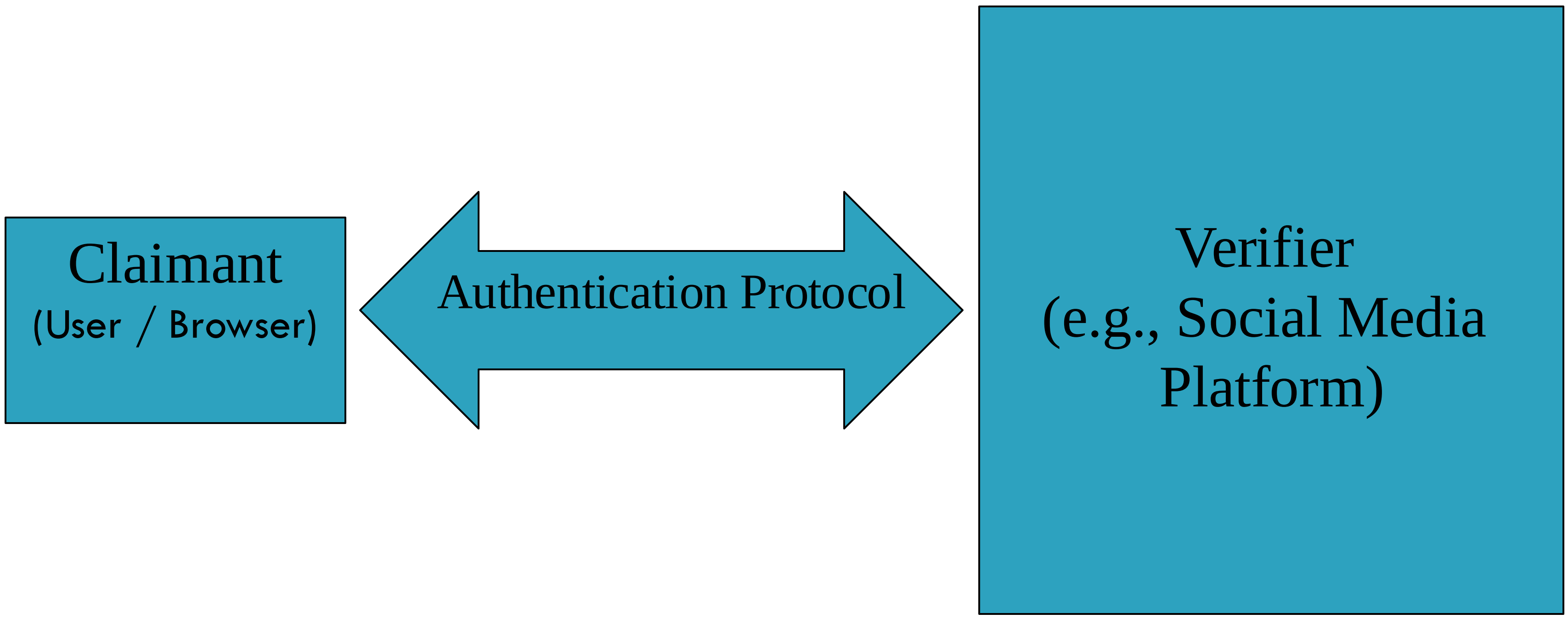
Lecture Motivation

- One-way property, weak and strong collision resistance are fundamental properties of a hash function
- These come also into play when we consider common password storage methods …
- … and approaches to undermine such methods
- Such approaches are summarised in this slide deck

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 memorised 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**

Claimant and Verifier



Storing User Passwords

- User passwords at rest (e.g., in database tables) are hashed instead of being stored in plaintext
- Idea:
  - "KenSentMe!" → "7b24afc8bc80e548d66c4e7ff72171c5"
    - <u>Note:</u> This token is in hex format, it is128 bit long (32 x 4 bits)
  - An attacker cannot algorithmically reverse-engineer a hash function to recover the original password
    - Recall hash function properties
  - The verifier does not have a plaintext copy of the password either

Why is this Form of Password Hash Management problematic?

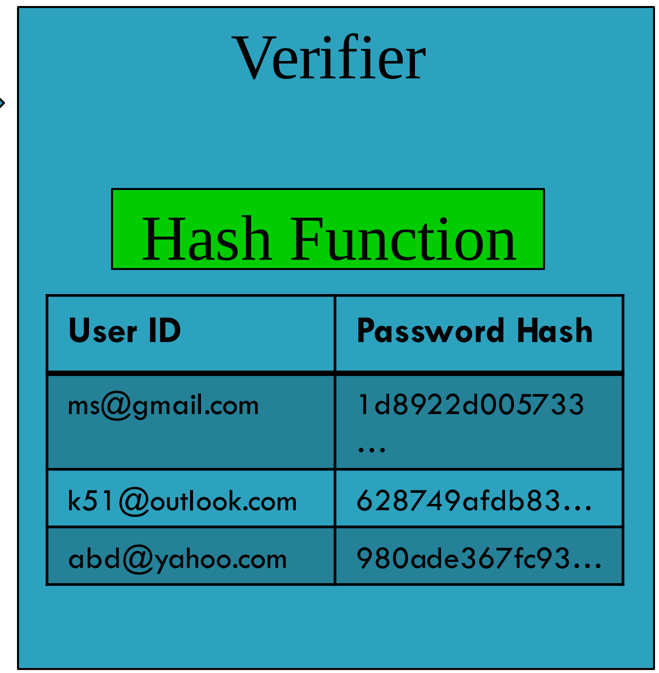
User ID	Password Hash
ms@gmail.com	1d8922d005733 …
k51@outlook.com	628749afdb83…
abd@yahoo.com	980ade367fc93…

Consider a webserver that stores user credentials.

A user registration entails the following steps:

1. The claimant visits the verifier's landing / login page using their web browser
2. The claimant enters and submits their user id and password
3. Both are sent to the verifier over the secure connection
4. The verifier calculates the hash, and and stores it together with the user name in the DB table

Server-Side Password Storage



1. The claimant enters user id and password (i.e., login details)
2. A client-side (e.g. JS) script calculates the password hash
3. User id and hashed password are sent to the verifier using the secure connection
4. The verifier checks if the transmitted user id and hashed password against the stored values in the table
5. The verifier notifies the claimant via the authentication protocol if the authentication was successful

Dictionary-Based Brute-Force Search

- Assume an attacker retrieves an entire DB table containing user IDs and hashed passwords
- Hash functions are one-way functions, so hash values cannot be transformed back to the original input
- However, assuming that a user picks a common word or phrase, or a known password as their own password, a simple dictionary search can be used to systematically identify a match for a given hash value
  - Here the underlying hash function must be known
- Such dictionaries are based on large word, phrase or password collections
- ☺ :
  - Straight forward process
  - Large dictionaries are readily available (next slide)
- ☹ :
  - Significant computational effort to find match
  - No guaranteed result

CrackStation's Password Cracking Dictionary

- https://crackstation.net/crackstation-wordlist-password-cracking-dictionary.htm

CrackStation's Password Cracking Dictionary

I am releasing CrackStation's main password cracking dictionary (1,493,677,782 words, 15GB) for download.

What's in the list?

The list contains every wordlist, dictionary, and password database leak that I could find on the internet (and I spent a LOT of time looking). It also contains every word in the Wikipedia databases (pages-articles, retrieved 2010, all languages) as well as lots of books from Project Gutenberg. It also includes the passwords from some low-profile database breaches that were being sold in the underground years ago.

The format of the list is a standard text file sorted in non-case-sensitive alphabetical order. Lines are separated with a newline "\n" character.

You can test the list without downloading it by giving SHA256 hashes to the free hash cracker. Here's a tool for computing hashes easily. Here are the results of cracking LinkedIn's and eHarmony's password hash leaks with the list.

The list is responsible for cracking about 30% of all hashes given to CrackStation's free hash cracker, but that figure should be taken with a grain of salt because some people try hashes of really weak passwords just to test the service, and others try to crack their hashes with other online hash crackers before finding CrackStation. Using the list, we were able to crack 49.98% of one customer's set of 373,000 human password hashes to motivate their move to a better salting scheme.

Download

Note: To download the torrents, you will need a torrent client like Transmission (for Linux and Mac), or uTorrent for Windows.

Torrent (Fast)

GZIP-compressed (level 9). 4.2 GiB compressed. 15 GiB uncompressed.

HTTP Mirror (Slow)

Checksums (crackstation.txt.gz)

```
MD5:    4748a72706ff934a17662446862ca4f8
SHA1:   efa3f5ecbfba03df523418a70871ec59757b6d3f
SHA256: a6dc17d27d0a34f57c989741acdd485b8aee45a6e9796daf8c9435370dc61612
```

Example

- Assume a hash code and the underlying hash function are known
- The dictionary contains 10^{10} entries
- A single laptop / PC can compute 10^5 hash values per second
- It takes 10^5 seconds (~29 hours) to search the entire dictionary for a match
- This process can be vastly improved by using pre-processed lookup tables

Lookup Table-Based Attacks

- For a given hash function and dictionary
  - Calculate the hash values for all dictionary entries
  - Insert 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**
  - Store the table
- Example table (assuming 44-bit hash values):

Hash value	Password
0x00000000354	gangster
0x00000001003	Bluemoon
0x00000001032	Z0om!
…	…

Lookup Table-Based Attacks

- A matching password for a given hash value can be recovered by systematically searching the look-up table via a binary search
- ☺ :
  - Such a table can be generated offline
  - The search process itself is fast (~$\log_2$(# of entries)) using binary search
    - A table containing 1.8×10^{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

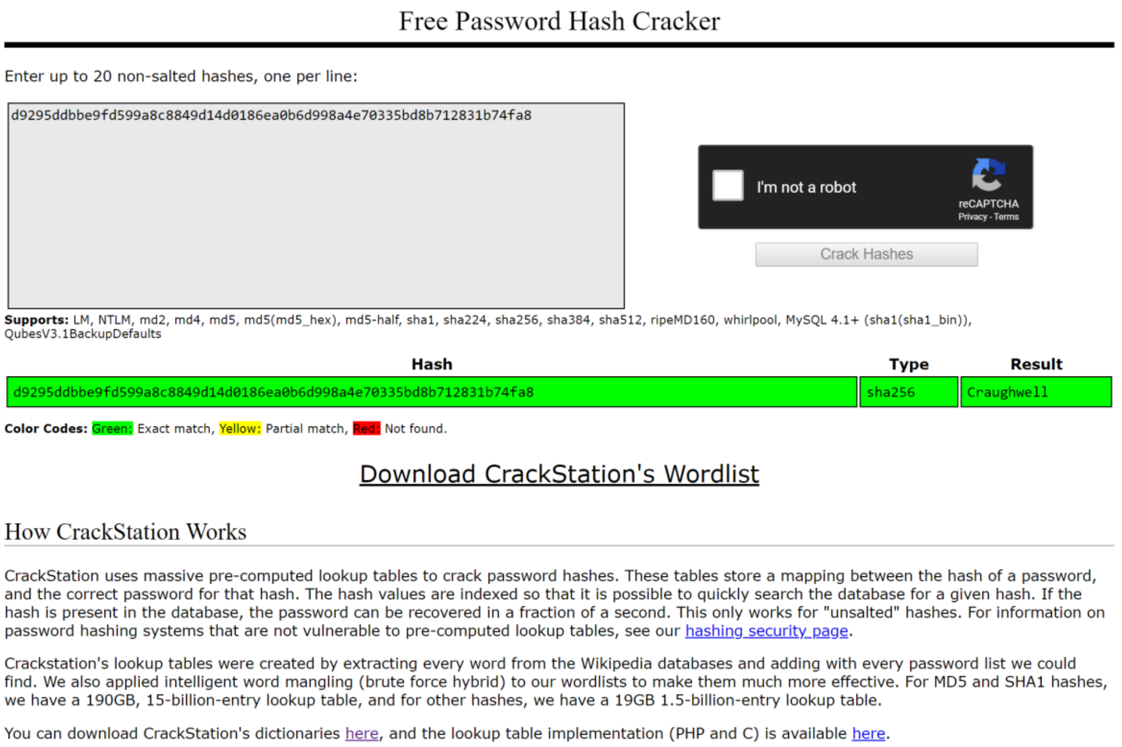
Lookup Table-Based Attacks: Example

- Assume a hash function that generates 16-byte (128 bit) hash values
- 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**

- However, there are online services available that host pre-computed look-up tables for password attacks (see next slide)

Crackstation's free Password Hash Cracker

- https://crackstation.net/



In-Class Activity: Password Recovery

- 5 minutes only, work alone or in a group
- What to do:
  - Pick a password and calculate its MD5 or SHA1 hash using https://defuse.ca/checksums.htm
  - Copy and paste the hash value into https://crackstation.net/ to see if it is can be recovered
  - Repeat the above and keep a list of all passwords
    - that **can** be cracked
    - that **cannot** be cracked

Rainbow Tables

- Look-up tables are huge and take up a lot of hard disk space
- Rainbow tables in contrast provide an efficient way to represent large numbers of hash values
- They require more processing time and less storage to find a match compared to a simple lookup table
- Rainbow tables are a practical example of a **space–time trade-off**
- They are based on pre-computed hash chains

Pre-Computed Hash Chains

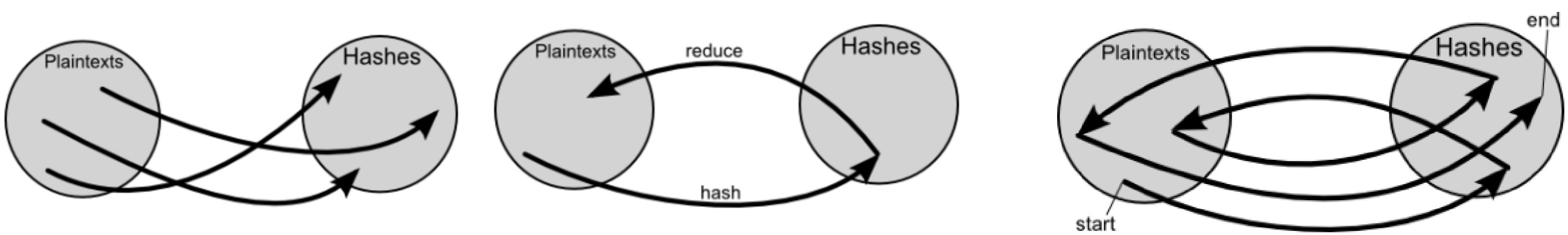
- Such chains contain long sequences of password candidates (green strings below) and hash values (black strings below)
- The are based on using a hash function "→" and a reduction function "→", e.g.,
  aaaaaa→173bdfede2ee3ab3 → jdjkuo →9fdde3a0027fbb36 →… → k3rtol
  - In this example we only consider passwords (green) that are 6 characters long, which are converted into 64-bit hash values
  - Each chain starts with a different password
  - Each chain has a fixed length, e.g. 100,000 passwords and their hashes
  - Here "→" converts the 64-bit hash value into an arbitrary 6-byte long string again, i.e. it's not an inverted hash function!
- We only store the first and the last value (starting point and end point), i.e. "aaaaaa" and "k3rtol"

Example for a simple Reduction Function


```
private static String reductionFunction(long val) {            // Hash value is just a long integer
    String car, out;                                            // The method returns an alphanumeric string
    int i;
    char dat;

    car = new String("0123456789ABCDEFGHIJKLMNOPQRSTUNVXYZabcdefghijklmnopqrstuvwxyz!#");
    out = new String("");

    for (i = 0; i < 8; i++) {
        dat = (char) (val % 63);
        val = val / 83;
        out = out + car.charAt(dat);
    }

    return out;
}
```

Coverage of Hash Chains

- The reduction function determines the range (i.e., length and composition) of plaintext (i.e., password) candidates that are covered
- Example:
  - Consider the password "Domino5"
  - In order to have this word stored in a chain, the reduction function must create outputs that are
    - At least 7 characters long
    - Contain small and capital letters, as well as numbers
  - Also, hash chains may not be able to cover all possible character combinations



Pseudo-Code to create a single Chain

- This example creates a chain with the start value "abcdefg" that covers 10,001 plaintext words
- Note that the last value of this chain is a hash value (i.e. ciphertext)
- We don't know for certain what type of words the reduction function returns, possible only words of length 7 that consist of small letters only

```
String plaintext, first, ciphertext;

plaintext = first = "abcdefg";

for ( int i=0; i<10000; i++ ) {
  ciphertext = hash_it (plaintext);
  plaintext = reduce_it (ciphertext);
}

System.out.printf ("%s:%s\n", first, ciphertext);
```

Chain Lookup

Assume we have a table with just 2 chains (with start and end values), i.e.

aaaaaa → 173bdfede2ee3ab3 → … → 8995tg → 9fdde3a0027fbb36 → … → k3rtol

hfk39f → 856385934954950 → … → delphi → 759858fde66e8aa8 → … → prp56e

… and a hash value "759858fde66e8aa8" we'd like to crack

Starting with this hash value we apply consecutively "→" and "→", until we

- hit a known end value (e.g., k3rtol), or
- have repeated "→" and "→" x times (with x being the length of the chain)

If we hit a known end value, e.g. "prp56e", we repeat the transformation, beginning with the start value of the chain, i.e., "hfk39f", until we hit "759858fde66e8aa8" again

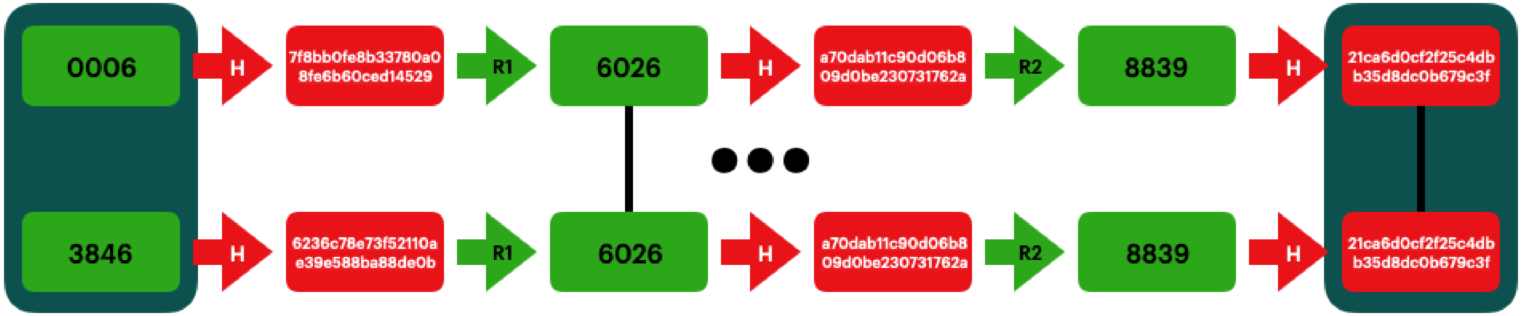
The input that led to the hash value (i.e., "delphi") is the solution

Chain Lookup Pseudocode

1. Input: Hash value H
2. Reduce H into another plaintext P
3. Look for the plaintext P in the list of final plaintexts (i.e. end values), if it is there, break out of the loop and goto step 6.
4. If it isn't there, calculate the hash H of the plaintext P
5. Goto step 2., unless you've done the maximum amount of iterations
6. If P matches one of the final plaintexts, you've got a matching chain; in this case walk through the chain in question again starting with the corresponding start value, until you find the text that translates into H

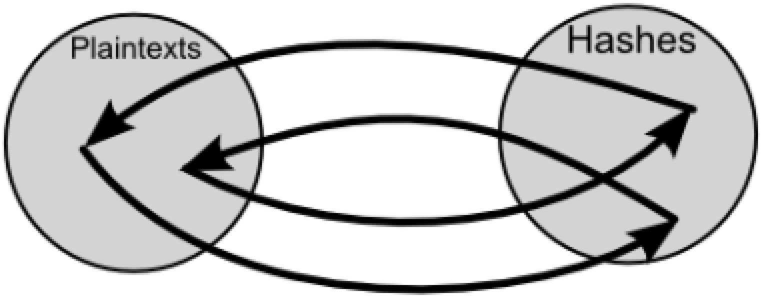
Chain Collisions

- Consider the following scenario:
  aaaaaa→ … → 173bdfede2ee3ab3 → delphi → 759858fde66e8aa8 →… → prp56e
  hfk39f→ … → 856385934954950 → delphi → 759858fde66e8aa8 →… → prp56e
- These 2 chains could merge, because
  - the reduction function translates two different hashes into the same password (as reduction functions are imperfect), or
  - the hash function translates two different passwords into the same hash (which should not happen → see hash function requirements)
- Because of these collisions or chain loops (next slide) hash chains will not cover as many passwords as theoretically possible despite having paid the same computational cost to generate
  - Previous chains are not stored in their entirety; therefore, it is impossible to detect this efficiently



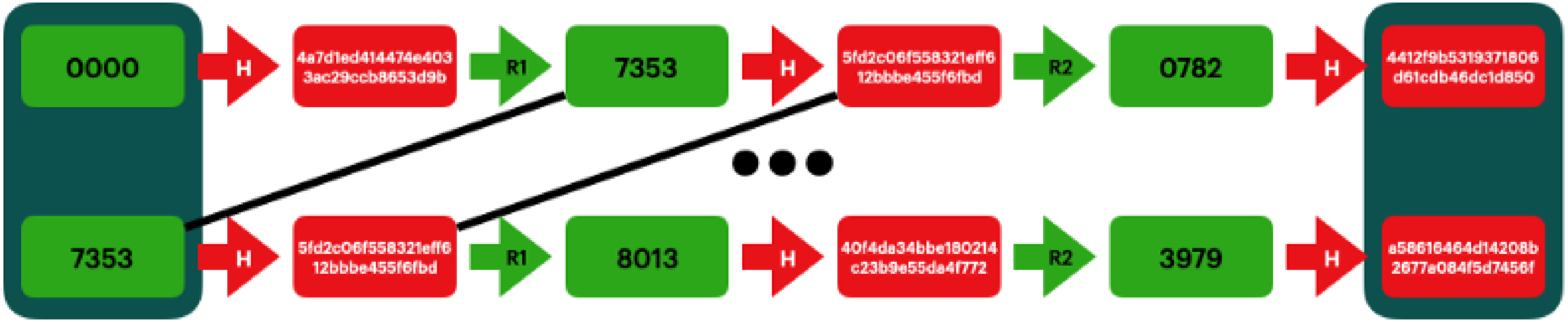
Chain Loops

- Here you find repetitions of hashes in a single chain
- The result of imperfect reduction functions that map two different hashes into the same plaintext



Rainbow Tables

- Rainbow tables effectively solve the problem of collisions with ordinary hash chains by replacing the single reduction function R with a sequence of related reduction functions R_1 through R_k (one reduction function per chain element)
- In this way, for two chains to collide and merge they must hit the same value on the same iteration, which is rather unlikely



Example for a Reduction Function for a Rainbow Table


```java
private static String reductionFunction(long val, int round) {  // Note that for the first function call "round" has to be 0,
    String car, out;                                              // and has to be incremented by one with every subsequent call.
    int i;                                                        // I.e. "round" created variations of the reduction function.
    char dat;

    car = new String("0123456789ABCDEFGHIJKLMNOPQRSTUNVXYZabcdefghijklmnopqrstuvwxyz!#");
    out = new String("");

    for (i = 0; i < 8; i++) {
        val -= round;
        dat = (char) (val % 63);
        val = val / 83;
        out = out + car.charAt(dat);
    }

    return out;
}
```

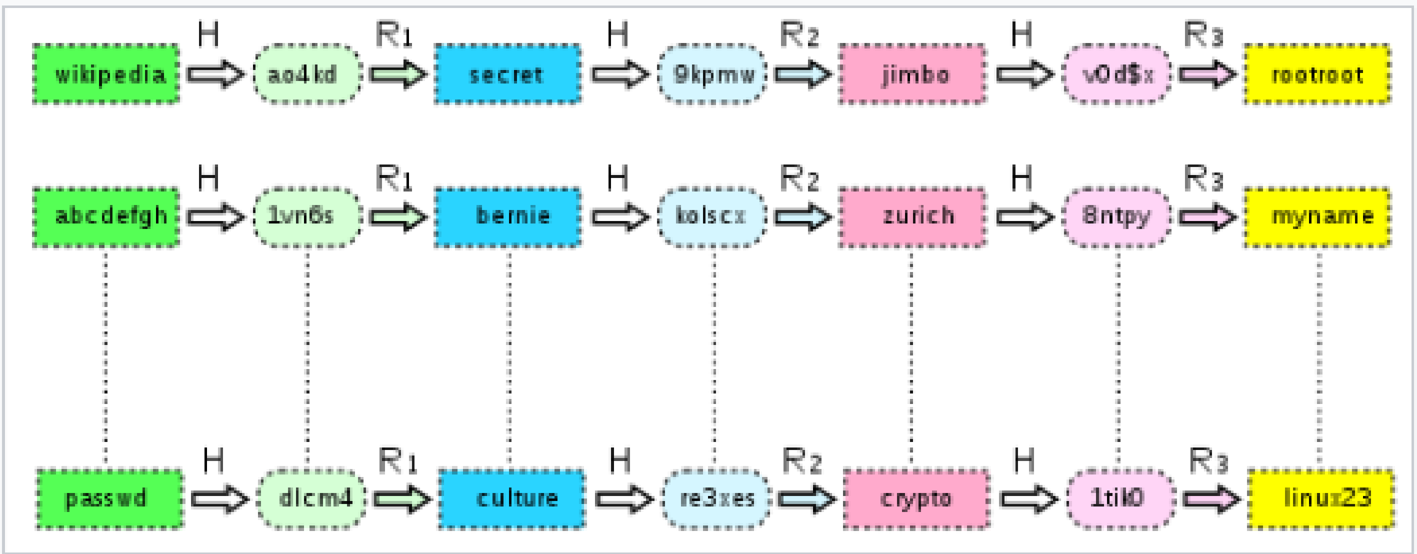
Coverage of Reduction Functions

- Rather than calculating a random string a reduction function may calculate an integer index value to identify an entry (word) in a large (password) dictionary
- Example:
  - H(lalo) = 368437FDA
  - R(368437FDA) = 6 → dict[6] = robot123
  - H(robot123) = DDA0087e73
  - …
- This is similar to a lookup table, but requires far less space, as hashes are not stored
- However, it may be difficult to design a hash function that covers all dictionary indices

#	dict entry
0	Dog5
1	Simple
2	fEED2
3	lalo
4	mEn
5	hat
6	robot123
7	rose
…	

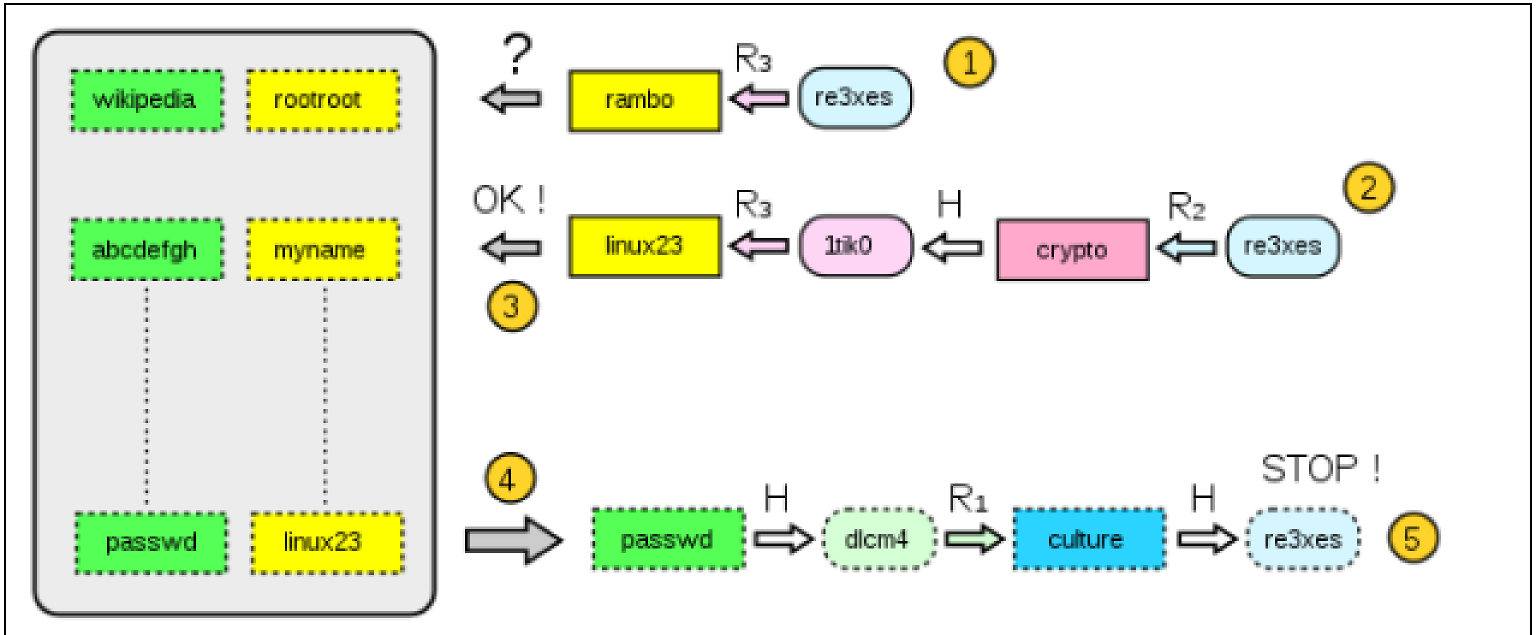
Searching a Rainbow Table (Wikipedia)

- Let's assume a Rainbow table of length 3 with 3 different reduction functions R_1, R_2 and R_3
- Again, we just store start (green) and end (yellow) value of each chain



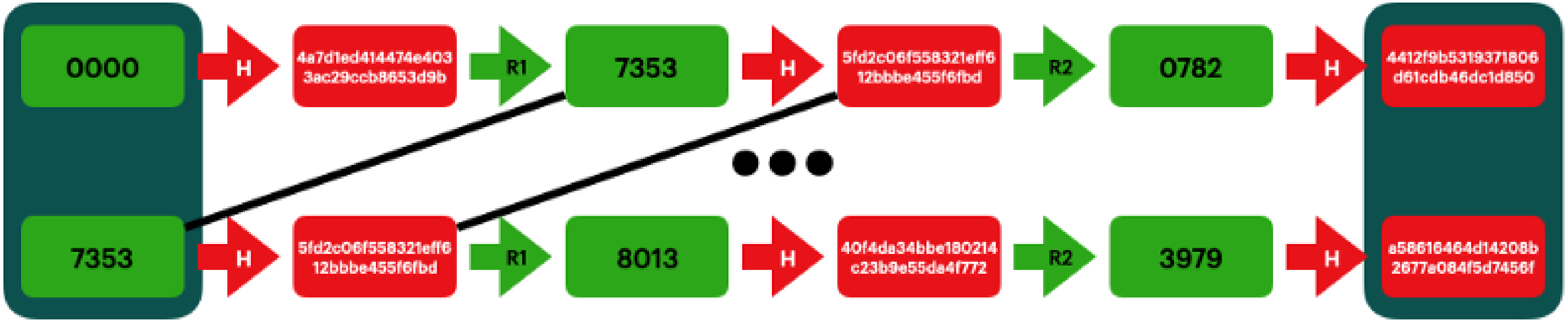
Searching a Rainbow Table (Wikipedia)

- Consider you have the rainbow table below and the password hash “re3xes”
  - Calculate R3(“re3xes”) and check if the result matches any of the chain ends (yellow boxes)
  - Calculate R2(H(R3(“re3xes”))) and check if the result matches any of the chain ends
  - ..
  - Repeat this process until the algorithm reaches R1, or a match is found
  - If a match is found, traverse through the chain in question as seen before, to find the solution



Perfect and non-perfect Rainbow Tables

- In a **perfect rainbow table** any word does not appear in more than one chain
- **Non-prefect rainbow tables (as shown below)** have redundant entries
  - They are easier to compute, but less memory-efficient because of these repetitions (which are not collisions!)



Defense against Rainbow Tables

- Idea:
  - Increase the (required minimum) length of a password
  - By doing so there are many more potential passwords to be considered by a rainbow table …
    - … up to a point where such tables are simply no more economical to generate
  - Increasing the password length can be either done by the
    - password owner (e.g., on the client side), or
    - algorithmically (e.g., on the client or server side)

Defence against Rainbow Tables

Client-side defence:

- A user requirement to choose long passwords that contain different types of characters,
  e.g. consider passwords that contain “A…Z”, “a…z”, “1-8”:
  - 6 characters long passwords result in 6^{60} = 46,656,000,000 combinations
  - 10 characters long passwords result in 10^{60} = 604,661,760,000,000,000 combinations

Server- (and potentially client-) side defence:

1. Password salting
   - A unique and random, but known string (“salt”) per user that is appended to each password before its hash is calculated
   - The salt is stored in the user database

User ID	**Salt**	**Password Hash**	**Password (not part of table)**
ms@gmail.com	12367	1d8922d005733…	12367KenSentme!
k51@outlook.com	56f87	628749afdb83…	56f87Fluffybear
abd@yahoo.com	465d0	980ade367fc93…	46d05Limerick

Defense against Rainbow Tables

2. Password peppering
   - Similar to Salting, but a unique ***secret*** string is concatenated to all passwords before they are hashed
4. Multiple iterations
   - A password is hashed multiple (e.g., 1000) times before stored in the database
5. Combination approach
   - Different techniques are combined to create a complex hash algorithm, e.g.,
   - NewHash(password) = hash(hash(password) || salt)

SQL Attacks

Some revision material covering

- SQL
- HTTP get / post Methods and PHP
- SQL injection attacks
- SQL injection attack mitigation strategies

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
- Such attacks are closely linked to various web technologies, i.e. HTTP and PHP

HTTP get / post Methods and PHP

- PHP is a general-purpose server-side scripting language especially suited to web development
- PHP originally stood for Personal Home Page, but it now stands for the recursive initialism PHP: Hypertext Pre-processor
- 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.

```html
<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>
```

foo.php:

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

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>
```

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

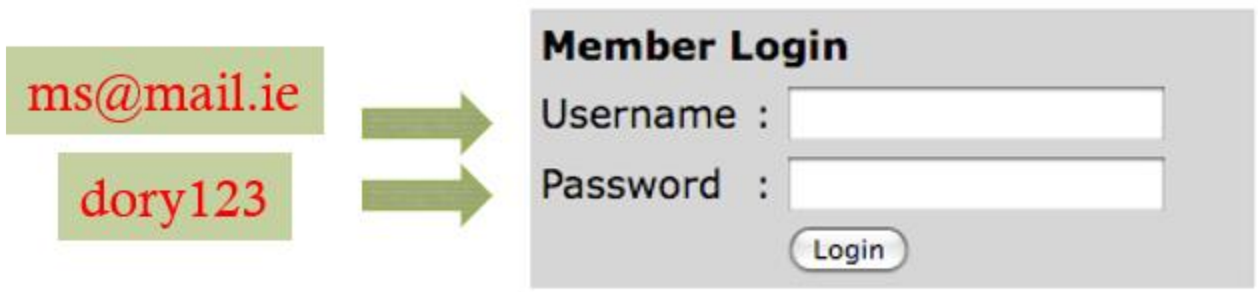
SQL Syntax Review

- 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

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



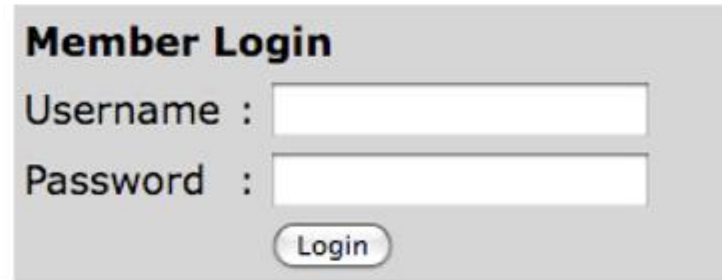

```
$email = $_POST['username'];
$pwd = $_POST['password'];

$password = MD5($pwd);

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

Table tblclinician:

Email	Hashed Password
ms@mail.ie	af47f8d1ac4
…	…

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**
- More generally, If DB details are not known to the attacker, **blind SQL injections** are used

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 = 1284ffa WHERE Email = ms@mail.ie ;--' AND `Password`=''

- The first injection creates a new user (hacker) including password hash
- The second injection replaces a user's password hash

Types of SQL Injection Attacks

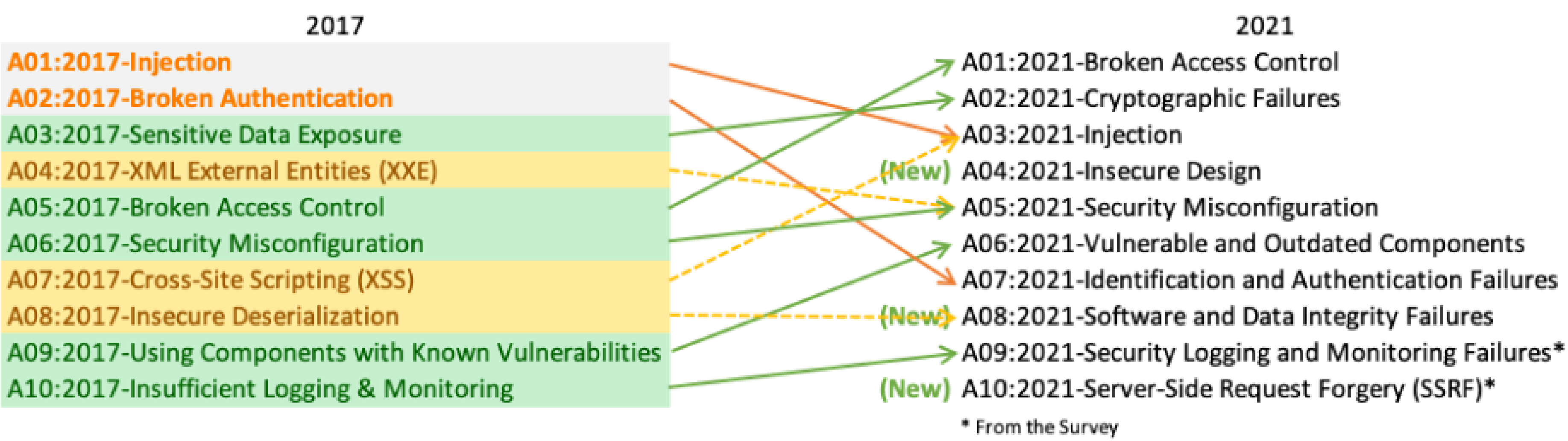
- **Blind SQL Injection**
  - Enter an attack on one vulnerable page but it may not display results
  - A second page would then be used to view the attack results
- **Conditional Response**
  - Test input conditions to see if an error is returned or not
  - Depending on the response, the attacker can 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 one of the frontrunners on the OWASP top ten list
  - A well understood attack, but still not fully grasped by the developer community

OWASP Top 10

- The Open Web Application Security Project (OWASP) is a non-profit foundation dedicated to improving the security of software



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