Paxson
Spring 2017

CS 161
Computer Security

Homework 2

Due: Monday, February 13, at 11:59pm

Instructions. This homework is due **Monday, February 13, at 11:59pm**. No late homeworks will be accepted. This assignment must be done on your own.

Create an EECS instructional class account if you have not already. To do so, visit https://inst.eecs.berkeley.edu/webacct/, click "Login using your Berkeley CalNet ID," then find the cs161 row and click "Get a new account." Be sure to take note of the account login and password, and log in to your instructional account.

Make sure you have a Gradescope account and are joined in this course. The homework *must* be submitted electronically via Gradescope (not by any other method). Your answer for each question, when submitted on Gradescope, should either be a separate file per question, or a single file with each question's answer on a separate page.

Problem 1 ***Web Security Warm-Up*** **(20 points)**

(a) In class we learned about the SOP (Same-Origin Policy) for cookies and the DOM (Document Object Model) and how it protects different websites from each other. Your friend says that you should be careful of visiting any unfamiliar site, because the site's owners can read cookies from *any* other websites they want. Is your friend right? Explain in 1–2 sentences why or why not.

(b) Suppose Google had a website creation service at `googlesites.com/[NAME]`. This hypothetical service allows you to choose your own `NAME` and upload any script or HTML that you desire. Why is this a better design than putting user sites on `google.com/sites/[NAME]`?

(Not for credit: Even this design is not perfect. Once you solve this question, you might enjoy thinking about what the limitations of this design are and how it could be further improved. But we're not asking you to write about this in your answer; that's just a thought exercise for your own understanding.)

(c) You are the developer for a spiffy new social startup, Bramspam, and you have been tasked with building the web-based sharing form. You have set up a simple form with just two fields, the text to share and a theme that will influence how the post is displayed to other users. (Bramspam's killer feature is that it gives its users a plethora of festive and elegant themes to choose from for their posts.) When a user clicks submit, the following request is made:

```
https://www.bramspam.com/share?text=<the text to share>&theme=<the chosen theme>
```

You show this to your brother Caleb, and he thinks there is a problem. He later sends you this message:

```
Hey, check out this cute cat picture.  tinyurl.com/zchm2pn
```

You click on this link and later find out that you have created a post with the theme "`howboudah`" and the text "Caleb is the best brother ever". (TinyURL is a URL redirection service. Whoever creates the link can choose whatever URL it redirects to.)

How was this post created?[1] What did the tinyurl redirect to? Write the link in your solution.

(d) Continuing from part (c), how could you defend your form from the sort of attack listed in part (c)? Explain in 1–2 sentences.

Problem 2 ***XSS: The Game*** **(20 points)**

Visit https://xss-game.appspot.com/ and complete the first 4 levels. This game is similar to Project 1, except you'll be exploiting XSS vulnerabilities instead of buffer overflows. You may use the hints provided by the game.

For each level, describe the vulnerability and how you exploited it in 2–3 sentences. Show the code that you used or what you typed into the input fields.

We recommend using the Chrome browser for this. (We had problems getting past level 3 in Firefox.)

[1] A reminder: in URLs, spaces are encoded as `%20`.

Problem 3 ***SQL Injection*** **(30 points)**

You are discouraged to find the following Java code in the client login section of an online banking website:

```
/**
 * Check whether a username and password combination is valid.
 */
ResultSet checkPassword(Connection conn, String username, String password)
        throws SQLException {
    String query = "SELECT user_id FROM Customers WHERE username = '"
        + username + "' AND password = SHA256('" + password + "');";
    Statement s = conn.createStatement();
    return s.executeQuery(query);
}
```

Here `SHA256` is a special type of hash function (which we'll learn more about later in the course). For the purposes of this problem, you can treat it as doing a deterministic scrambling of `password` into the format in which passwords are stored inside the database. The particulars of this behavior are not important for the problem.

Clarification 2/8/2017: Assume that before issuing a request, the bank's server calls `checkPassword` and ensures that the returned `ResultSet` contains *exactly one* `user_id`. If this check fails, the bank fails the request. Otherwise the request is issued as the user represented by `user_id`.

Note: if there are 0 `user_id`s in the `ResultSet` then the username and/or password are wrong. If there are more than one then something went wrong somewhere on the bank's end since usernames should be unique (and consequently limit results to at most one). For the purposes of this question, what's important is that the request goes through iff the `ResultSet` contains exactly one `user_id`.

(a) What username could an attacker enter in order to delete the `Customers` table?

(b) What username could an attacker enter in order to issue a request as user "Admin", without having to know the password?

(c) When you point this out to the development team, a junior developer suggests simply escaping all the single quotes with a backslash. For example, the following line could be added to the top of the function:

```
username = username.replaceAll("'","\\\\'");
```

This code replaces each `'` in the username with `\'` before including it in the SQL query.

Modify your answer to part (b) above so it will work against this new code. Assume the database engine accepts either ' or " to enclose strings.

Note: regarding the use of four backslashes in the source code above: this arises due to there being two separate instances of escaping going on. First, the text:

```
"\\\\'"
```

is a string literal *in the Java source code*. The Java compiler interprets it as specifying a string whose literal *data* value is:

```
\\'
```

Second, the `replaceAll` method takes a regular expression for its second argument. It *parses* that regular expression, including processing any escape sequences in it. Given the above *data* value, it then interprets the regular expression as:

```
\'
```

and so it will change any instance of a single `'` to `\'`.

(You might wonder *why* `replaceAll` treats its second argument as a regular expression. It does so because it's looking for "backreferences" like "\1", which means "in this part of the replacement use the first sub-match in the original regular expression." That detail isn't important for this problem.)

(d) Rewrite the `checkPassword` function using prepared statements. Show your modified code. Explain why your code is safe from SQL injection attacks.

See the Java documentation for a discussion of Java `prepared` statements.

(e) The developers are now busy refactoring their code to use prepared statements. One of them approaches you because they're having difficulty adapting the function below.

```
/**
 * Filter transactions based on a dollar amount specified by the user and
 * sort based on user-supplied values.
 *
 * @param conn Database connection
 * @param amt  Look for transactions with at least this amount.
 * @param orderByCol Name of column by which to sort results
 *                   ("amount", "date" or "type").
 * @param orderByDir Sorting direction ("ASC" or "DESC").
 */
ResultSet searchTransactions(Connection conn, BigDecimal amt,
        String orderByCol, String orderByDir) throws Exception {
    String q = "SELECT * FROM Transactions WHERE ";
    q += " amount >= " + amt;
    q += " ORDER BY " + orderByCol + " " + orderByDir + ";";
    return conn.createStatement().executeQuery(q);
}
```

For example, this would allow queries like the following to be run:

```
SELECT * FROM Transactions WHERE amount >= 13.37 ORDER BY date DESC;
SELECT * FROM Transactions WHERE amount >= 42.00 ORDER BY amount ASC;
```

What makes this function more difficult to write a Java prepared statement for, and why? Rewrite the `searchTransactions` function so it uses prepared statements appropriately for this case and so it is secure against SQL injection. It's okay to make modifications to the code as long as queries like the examples above can still be run. Show your modified code.
Hint: since we'd only like to run the statements we could previously, whitelisting may be a good strategy here to make the prepared statement work.

Problem 4 ***Reasoning About Memory Safety*** **(30 points)**

Consider the following C code:

```
void dectohex(uint32_t decimal, char* hex) {
    char tmp[9];
    int digit, j = 0, k = 0;
    do {
        digit = decimal % 16;
        if (digit < 10) {
            digit += '0';
        } else {
            digit += 'A' - 10;
        }
        tmp[j++] = digit;
        decimal /= 16;
    } while (decimal > 0);
    while (j > 0) {
        hex[k++] = tmp[--j];
    }
    hex[k] = '\0';
}
```

Determine the precondition under which `dectohex` is memory-safe, and then prove it by filling in the invariants and precondition on the following page. Additionally, for each invariant, write a short explanation justifying why it holds. Your precondition cannot unduly constrain 'decimal', such as requiring it to only be zero. Make the precondition as conservative as possible. (That is, if a buffer needs to be at least `2n` bytes in length, don't require that it is more than `2n` bytes in the precondition.) Justify why the precondition you chose cannot be made any less restrictive while still ensuring memory safety.

Recall our proving strategy from lecture:

1. Identify each point of memory access
2. Write down any preconditions it requires
3. Propagate requirement up to the beginning of the function

Hint: Propagating the requirement up to the beginning of the function is more involved than in lecture. Here you need to reason about the properties that hold about the array indices after they are modified.

```
/* (a) Precondition:
 *                   ------------------------------------------- */
void dectohex(uint32_t decimal, char* hex) {
    char tmp[9];
    int digit, j = 0, k = 0;
    do {
        digit = decimal % 16;
        if (digit < 10) {
            digit += '0';
        } else {
            digit += 'A' - 10;
        }
        /* (b) Invariant:
         *                  ---------------------------------------- */
        tmp[j++] = digit;
        /* (c) Invariant:
         *                  ---------------------------------------- */
        decimal /= 16;
    } while (decimal > 0);
    while (j > 0) {
        /* (d) Invariant:
         *                  ---------------------------------------- */
        hex[k++] = tmp[--j];
        /* (e) Invariant:
         *                  ---------------------------------------- */
    }
    /* (f) Invariant:
     *                   ------------------------------------------- */
    hex[k] = '\0';
}
```

(a) **explanation**

(b) **explanation**

(c) **explanation**

(d) **explanation**

(e) **explanation**

(f) **explanation**

Problem 5 ***Feedback*** **(0 points)**

Optionally, feel free to include feedback. What's the single thing we could do to make the class better? Or, what did you find most difficult or confusing from lectures or the rest of class, and what would you like to see explained better? If you have feedback, submit your comments as your answer to Q5.