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.

> **Solution:** Your friend is not correct. Because of the Same-Origin Policy, a site only has access to cookies from the same origin.

(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]`?

> **Solution:** Because of the SOP, a malicious user of Google Sites could construct her own page which steals cookies. Since `google.com/sites` has the same domain as other `google.com` services, she could steal your Google/Gmail cookie and log in as you.

(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.)

Solution: One limitation of Google's existing design is that it allows one user's site to attack any other user's site. In other words, `googlesites.com/Cory` can attack `googlesites.com/Camron` (e.g., reading session cookies, tampering with the DOM/content of the other). They can't attack Gmail or other core Google services, but they can attack other sites created with Google Sites.

To address this, Google could use `cory.googlesites.com`. Then content on `cory.googlesites.com` is isolated from `google.com` by the same-origin policy, and can't have any interaction with cookie for `google.com` core services since the two domains are unrelated. For instance, Facebook uses a design like this for some user content.

(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.

Solution: Caleb tricked you into clicking the link, which is exploiting a CSRF vulnerability in Bramspam.

`https://www.bramspam.com/share?`
`text=Caleb%20is%20the%20best%20brother%20ever&theme=howboudah`

The linebreak was added for readability.

Note that in the query parameter part of a URL it's acceptable to encode spaces using `+` as an alternative to `%20`. However, in other parts of the URL (e.g. the path), `+` is treated as a literal value and spaces must be encoded as `%20`. It's

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

always safe to encode spaces as `%20` so it's good practice to do so. Check out this Stack Overflow discussion for more information.

(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.

Solution: Use a CSRF token, which is a unique unpredictable token associated with the form and user. When a request is made, the browser will also send the token and the server will only accept requests with the correct token. In this example, when you click on the tinyurl, the request sent to the server wouldn't contain a valid token (since Caleb can't guess what your token is) and the request would be considered invalid.

Checking the `Referer` header and disallowing requests referred from other domains is also a valid solution. However, this protection may be bypassed with an "open redirect vulnerability" (where a web application redirects to a URL that is specified via the request or some form data, and therefore that URL can potentially be tampered with to redirect users to an external, malicious URL), and it won't work for users who filter out the `Referer` header; CSRF tokens are the more standard approach.

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.)

Solution:

(a) In Level 1, any input to the field will be shown on the results page. Because no escaping or sanitization is done, the input will be parsed as HTML on the results page. This allows us to conduct a reflected XSS attack.

```
<script>alert(42);</script>
```

(b) In Level 2, we can make posts which are stored by the server. `<script>` tags are disallowed, but it seems that other HTML tags are left unescaped so users can format their text. We can look for other methods of running Javascript without a `<script>` tag, to produce a stored XSS attack.

```
<img src="" onerror="alert(42);"/>
```

Here, we are using the fact that the `onerror` attribute contains Javascript code that will be executed if the image fails to load.

(c) In Level 3, the URL after `#` is used to determine the image number `num`. Examining the source code reveals that `num` is parsed as a string and copied into part of the response:

```
html += "<img src='/static/level3/cloud" + num + ".jpg' />";
```

Instead of writing a simple integer to `num`, we can close the quote and `img` tag prematurely. Afterwards, we can insert arbitrary HTML code:

```
https://xss-game.appspot.com/level3/frame#1'><script>alert();</script>
```

After visiting that URL, the `num` and `html` variables will get the following values:

```
num: 1'><script>alert();</script>
html: ...<img src='/static/level3/cloud1'><script>alert();</script>.jpg' />
```

Thus, we've injected an `alert()` into the web page.

Another solution:

```
https://xss-game.appspot.com/level3/frame#0'onerror='alert();'/>
```

The following gets inserted into the web page:

```
<img src='/static/level3/cloud0'onerror='alert();'/>.jpg' />
```

This gets treated as an `img` tag with an `onerror` attribute containing Javascript code. When the image fails to load, the Javascript code gets executed.

(d) In Level 4, the Python server gets the value of the variable `timer` from the input form. Then, that variable is inserted into the HTML through a server-side *template*:

```
<img src="/static/loading.gif" onload="startTimer('{{ timer }}');" />
```

In the above line, the server replaces `{{ timer }}` with the value we input into the form. So, we choose our string to close the quote prematurely. This allows us to run more Javascript in the `onload` function. For example:

```
3');alert('42
```

will produce:

```
<img src="/static/loading.gif" onload="startTimer('3');alert('42');" />
```

which starts a timer for 3 seconds and then continues to spawn an alert.

The lesson here is that using string concatenation to build up HTML is very dangerous, and it's easy to introduce a XSS vulnerability. There are a number of sneaky ways of exploiting such vulnerabilities.

How can we prevent this? The safest way to do so directly is to HTML-escape all untrusted (user-controllable) data before introducing it into the HTML. Some web programming frameworks include template languages that offer automatic escaping of dynamic data; this can be a helpful way to help make sure you escape everything that needs to be escaped.

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?

Solution: The username "`alice'; DROP TABLE Customers; --`" will cause the query to be processed as:

```
SELECT user_id FROM Customers WHERE username = 'alice';
DROP TABLE Customers; -- ' AND password = SHA256('" + password + "');
```

(where we've introduced a linebreak after the first semicolon for readability)

Due to the "`--`" comment specifier, this is equivalent to:

```
SELECT user_id FROM Customers WHERE username = 'alice';
DROP TABLE Customers;
```

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

Solution: The username "`Admin'; --`" will cause the query to be processed as:

```
SELECT user_id FROM Customers WHERE username = 'Admin'; -- ...
```

and thus it will execute for the username being Admin.

(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.)

Solution: The `replaceAll` substitution will put a \ in front of any single quote in the username. Given that, for the first approach in the problem above, an attacker could instead represent the attack using a username of:

```
alice\'; DROP TABLE Customers; --
```

The `replaceAll` replacement will expand this to

```
alice\\'; DROP TABLE Customers; --
```

The SQL interpreter will process escape sequences one more time, leading it to conclude that the username comparison is with `alice\` (rather than with just plain `alice` as in the original problem), and the rest of the attack works the same as before.

Modifying the second attack takes more work. The problem is that the attacker doesn't want to mess with the appearance of "`Admin`". In particular, if they use the same trick as in the solution above, they will ultimately wind up with the SQL interpreter processing the query in terms of user `Admin'`, and that won't accomplish the attacker's goal.

The solution here is to use a more elaborate construction. The attacker sets the username to:

```
\' OR username = "Admin"; --
```

After the `replaceAll` replacement, the SQL interpreter will see its input (before it processes escape sequences) as:

```
SELECT user_id FROM Customers
WHERE username = '\\' OR username = "Admin"; -- ...
```

After processing escapes, the WHERE clause becomes a test for the username equal to either a single backslash, or to "Admin". (Here we have used the fact that some SQL implementations allow strings to appear in either single quotes or double quote—a point briefly mentioned in lecture.)

Note, if you're really paranoid (generally a good thing for a security person!), then you might worry about the database actually *having* a user with the weird name of single-backslash. That then won't execute the query strictly in accordance of what was asked for. You-the-attacker can fix this problem by nullifying the effect of the first instance of the username. For example, you could use:

```
\' AND 1=0 OR username = "Admin"; --
```

This then will be processed by the SQL interpreter (prior to its escape processing) as:

```
SELECT user_id FROM Customers
WHERE username = '\\' AND 1=0 OR username = "Admin"; -- ...
```

Precedence makes this the same as:

```
SELECT user_id FROM Customers
WHERE (username = '\\' AND 1=0) OR (username = "Admin"); -- ...
```

and therefore even if there *is* a weirdo single-slash username, its records won't be selected, only those for Admin will be.

(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.

Solution: Prepared Statements only allow "?" placeholders to contain data; all other SQL language elements must be predeclared, and thus the underlying structure of the SQL query can't change regardless of what data the user supplies.

```
ResultSet checkPassword(Connection conn, String username,
        String password) throws SQLException {
    String query = "SELECT user_id FROM Customers WHERE "
        + "username = ? AND password = SHA256(?);";
    PreparedStatement p = conn.prepareStatement(query);
    p.setString(1, username);
    p.setString(2, password);
    return p.executeQuery();
}
```

(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.

Solution: Column names and keywords (like `ASC` and `DESC`) cannot be arameters in a prepared statement. Why is this? The API imposes this limitation because these are SQL language elements, not data. If user input can appear as language elements, then the input can change the *meaning* of the SQL query, which is the basis for SQL injection attacks.

Here is a solution based on *whitelisting*:

```
ResultSet searchTransactions(Connection conn, BigDecimal amt,
        String orderByCol, String orderByDir)
        throws Exception {
    String q = "SELECT * FROM Transactions WHERE ";
    if (! (orderByCol.matches("^(amount|date|type)$")) )
        throw new Exception("Invalid column in ORDER BY");
    if (! (orderByDir.matches("^(ASC|DESC)$")) )
        throw new Exception("Invalid ORDER BY direction keyword");
    q += " amount >= ? ";
    q += " ORDER BY " + orderByCol + " " orderByDir + ";";
    PreparedStatement p = conn.prepareStatement(q);
    p.setBigDecimal(1, amt);
    return p.executeQuery();
}
```

You could instead use a solution that employs multiple Prepared Statements, one for each combination of comparison, field and direction by which to order. (Painful!)

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**

Solution:

Let `LEN = 1 + floor(log16(decimal))` for the duration of this proof, where `decimal` is the original value of "decimal" passed in to `dectohex`. (abstractly, `LEN` is the number of digits in the hexadecimal representation of decimal)

Let `log16(0) = 0`.

Notice that since `hex` and `tmp` are never reassigned and are non-`NULL` at the start of the function (this follows from `(a)` for `hex` and from the locality of `tmp`), they are non-`NULL` for `(b)` through `(e)`. It is okay if your solution explicitly includes statements `hex` being non-`NULL` in your invariants. There's no need for such a statement about `tmp` since its declaration guarantees it is non-`NULL`.

```
/* (a) Precondition: hex != NULL && LEN < size(hex)                  */
void dectohex(uint32_t decimal, char* hex) {
    char tmp[9]; // <- NOTE: this really only needed to be 8. tmp
                 // is never NUL-terminated.
    int digit, j = 0, k = 0;
    do {
        digit = decimal % 16;
        if (digit < 10) {
            digit += '0';
        } else {
            digit += 'A' - 10;
        }
        /* (b) Invariant: 0 <= j && j < LEN                          */
        tmp[j++] = digit;
        /* (c) Invariant: 0 < j && j <= LEN                          */
        decimal /= 16;
    } while (decimal > 0);
    while (j > 0) {
        /* (d) Invariant: 0 < j && j <= LEN && 0 <= k && k < LEN     */
        hex[k++] = tmp[--j];
        /* (e) Invariant: 0 <= j && j < LEN && 0 < k && k <= LEN     */
    }
    /* (f) Invariant: 0 < k && k <= LEN                              */
    hex[k] = '\0';
}
```

Notes:

1. `decimal` is a `uint32_t`, so `decimal <= 2^32 - 1`, which means that we can get a bound on `LEN`:
   `LEN = 1 + floor(log16(decimal)) <= 1 + floor(log16(2^32 - 1)) = 8`.

   It's also okay to assume `LEN` is always at its upper bound (that is, `LEN = 8`) for your precondition and consequently throughout your proof. While technically both are correct and will earn full credit, assuming `LEN = 8` is actually safer since it makes it impossible to violate the precondition without changing `decimal`'s type. Otherwise, the precondition will be violated if an unexpectedly large value of `decimal` is passed to `dectohex`, leading to unsafe memory accesses.

2. the `do { ... } while (decimal > 0);` loop will be referred to as loop 1 and the `while (j > 0) { ... }` loop will be referred to as loop 2.

(a) **explanation**

`Precondition: hex != NULL && LEN < size(hex)`
`dectohex` accesses the value stored at `hex` without any sort of validity checking, assuming that `hex` is a valid pointer. Thus `hex != NULL` is a necessary precondition.

Since the `hex` buffer must be able to store a string equal in length to the number of digits in the hexadecimal representation of `decimal`, plus a null-terminator, it must have a size of at least `LEN + 1`.

(b) **explanation**

`Invariant: 0 <= j && j < LEN`
`j` is initialized to 0 and never decremented. Thus `j` only gets larger on each iteration of loop 1, *and* loop 1 can only iterate at most LEN times (see next paragraph), which keeps `j` from overflowing. Therefore, `0 <= j`.

Now, observe that on each iteration of loop 1 the line `decimal /= 16` logically removes the last hexadecimal digit from `decimal` and shifts the remaining hexadecimal digits one position to the right. Loop 1 only continues while `decimal > 0`, so after all hexadecimal digits have been removed from `decimal`, loop 1 will exit. We know that `decimal`'s hexadecimal representation has `LEN` digits, which means that loop 1 executes no more than `LEN` times, and consequently increments `j` no more than `LEN` times (per our coments above about overflowing). Thus, `j <= LEN`. However, since `j` is incremented *after* `(b)`, the increment on the last iteration will never be observed at `(b)`. Thus, `j < LEN`. So `0 <= j && j < LEN` holds.

Why this ensures safety: The subsequent `tmp` access is done with the value of `j` at `(b)` (`j++` evaluates to the value of `j` *before* the increment). From `(b)`,

we know that `0 <= j < LEN <= 8 < 9 = size(tmp)`, so `tmp` accesses here are safe.

(c) **explanation**

`Invariant: 0 < j && j <= LEN`
Since `j` is incremented precisely once between `(b)` and `(c)`, and `0 <= j && j < LEN` holds at `(b)`, then `0 < j && j <= LEN` holds at `(c)` (we're just shifting the bounds by 1).

(d) **explanation**

`Invariant: 0 < j && j <= LEN && 0 <= k && k < LEN`
Loop 2's `while`-condition guarantees that `0 < j` at its start. Furthermore, by `(c)`, we know that when loop 1 exits, we have `j <= LEN`. Since `j` is not modified between loops 1 and 2, and since loop 2 only subtracts from `j`, we can be certain that `j <= LEN` at `(d)`. (Loop 2's `while`-condition protects us from integer underflow.) Thus, `0 < j && j <= LEN` holds.

Another way to argue this latter part (that `0 < j && j <= LEN` holds for subsequent iterations of the loop) is to observe that it follows from Invariant (e) plus the loop's guard condition (`j > 0`).

Now we analyze `k`. On the first iteration of loop 2, `k == 0`, since `k` is initialized to 0 and has not yet been modified. Throughout the loop, `0 <= k` will hold, since `k` is only incremented. Regarding possible overflow, observe that each time `k` is incremented, `j` is decremented. We've shown that `0 < j`, which means that `j >= 1`. We've also shown that `j` has a max value of `LEN` at the start of the loop. This means that `j` will be decremented, and consequently `k` incremented, no more than `LEN` times. Thus, we don't have to worry about integer overflow for `k`.

On the first time through, since `k == 0` we can directly conclude that `k < LEN`. For subsequent iterations, we then argue from Invariant (e). It gives us `k <= LEN`. Per the argument above that we will exit the loop after `LEN` iterations, the case of `k == LEN` will not arise at Invariant (d). Thus, we have `k < LEN` at (d).

Why this ensures safety: The subsequent `hex` access is done with the value of `k` at `(d)` (`k++` evaluates to the value of `k` *before* the increment). The function's precondition gives us that `LEN < size(hex)`. For a buffer of this size, all accesses with indexes in the range `0 <= k && k < LEN` are valid.

(e) **explanation**

`Invariant: 0 <= j && j < LEN && 0 < k && k <= LEN`
At `(d)`, we know that `0 < j && j <= LEN`. Between `(d)` and `(e)`, 1 is subtracted from `j`, shifting `j`'s range of possible values down by 1. Therefore, at `(e)`, `0 <= j && j < LEN` holds. The same is true for `k`, but in reverse. That is,

1 is added to, rather than subtracted from, `k`. Thus `0 < k && k <= LEN` also holds.

Why this ensures safety: The previous `tmp` access is done with the value of `j` at `(e)` (`--j` evaluates to the value of `j` *after* the increment). From `(e)`, we know that `0 <= j < LEN <= 8 < 9 = size(tmp)`, so `tmp` accesses here are safe.

(f) **explanation**

`Invariant: 0 < k && k <= LEN`
`(e)` is true whenever the previous while loop exits. In addition, we need to consider the possibility that the `while` loop *never* iterates, which could happen if its condition fails upon the first entry into it. However, the previous `do ... while` will always execute at least once, and it exits with `0 < j`, per Invariant (c). Thus, the `while` loop will always iterate at least once—and thus `(e)` will hold when we consider `(f)`.

Given that `(e)` holds, and since nothing alters `k` between the last iteration at `(e)` and our arrival at `(f)`, `(f)` immediately follows from `(e)`.

Why this ensures safety: from `(f)` and `(a)` we know `0 < k <= LEN < size(hex)`, so this `hex` access is safe.

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.

Solution: Feedback is always welcome!