Software security - take-home exam

Q1

Given the following variation of the buffer overflow assignment, does it still allow a shellcode to be invoked when protections are disabled?

The difference with the lab assignment is that the badfile is only 40 bytes instead of 517. With ASLR enabled, is there more or less chance of starting a shellcode compared to the original code with a buffer of 517 bytes? Explain your answers.

Do not use the alternative ASLR-breaking solution that was found in the lab.

```
int bof(char* str) {
  char buffer[24];
  strcpy(buffer, str);
  return 1;
}
int main(int argc, char**argv) {
  char str[40];
  FILE *badfile;
  badfile = fopen("badfile", "r");
  fread(str, sizeof(char), 40, badfile);
  printf("Calling bof\n");
  bof(str);
  printf("Returned Properly\n");
  return 1;
}
```

- It still possible to invoke shellcode when the buffer size is lowered this much. The exploit can be put at the beginning of badfile and the return address can be set to the start of the shellcode as before, now at the beginning of buffer. The return address is still within reach of badfile's buffer.
- With ASLR enabled, chances are now lower. When the process is manipulated to return to a location somewhere in the buffer part of memory, it will fall down a NOP sledge and eventually hit the exploit. To maximize hitting chance of the exploit, the NOP sledge is made as long as possible, as every location within the sledge will result in the exploit. As `fread` limits the input after 40 characters, the NOP sledge will be much shorter than it was before, therefore decreasing the chance to hit the crafted buffer by accidence.

Q2

Consider the following program fragments in the C and Java programming languages:

```
// C
for (int i = 0; i < n; i++) {
  char *x; … x = a[i]; … x …
}

// Java
for (int i = 0; i < n; i++) {
  String x; … x = a[i]; … x …
}

// Java
for (String x : a) { … x … }
```

Discuss the differences in memory and type safety that these languages / constructs provide. What errors can programmers make? What is the effect of such errors? What countermeasures can be taken by programmers? How does the language help? Motivate your answers. Use and explain the relevant terminology.

- Errors that programmers probably make are out of bounds errors, for example by setting the wrong upper bound for the iterating variable. Most often poisonous off-by-one errors occur.
- C does not limit your access to the array you want to get a value from, even if you run past its defined size. In that case, behavior is undefined – it really is unknown what might happen.
    - You may read from a memory location out of your scope and the operating system can segfault you.
    - You may get a random value and your program crashes – in which case you are pretty lucky.
    - You may ‘just’ get a value and it may work as well.
    - You may get a random value and your program does not crash – but the program starts to behave odd. That is dangerous.
- C also does not prevent you from most typing errors. For example, when writing to standard out you might accidentally pass an integer that is interpreted as a memory location and result in the undefined behavior explained before.
- For C there is nothing that can be done about this, except for programming carefully and avoiding to write C code and make the same mistakes as others have done. In some compilers flags can be set to give hints that you are about to do stupid stuff or warn you to code it in a more secure way.

- In contrast to C, Java does protect you from accessing array values that are out of bounds on runtime. In that case, you get an `ArrayIndexOutOfBoundsException`s and it becomes apparent you made some mistake.
- Moreover, when Java compiles code, it type checks a lot. Wrong type casts result in `ClassCastException`s as a message to improve the code.
- The second Java example shows foreach syntax sugar that prevents you from writing code that has to get an element from the array that has a certain type.
- To conclude, Java helps you a lot to get only valid values on forehand and gives appropriate errors when mistakes are made.

Q3

A web application with a search feature shows the entered search query on the page that presents the results. When entering the following query:

```
<script>alert(document.cookie);</script>
```

An alert popup is triggered on the results page and shows a value. What type of exploit is this page vulnerable to (be as specific as you can be to describe the exploit)? How could an attacker abuse this and what could he achieve? How could this exploit be prevented?

This page is vulnerable to an XSS attack. An attacker might craft a search query URL that (using JavaScript) steals a session cookie and sends it to to the attacker, for example by reflecting an image that contains the session cookie as part of the request. If he convinces the admin user to follow that crafted url, he could impersonate the admin user and do whatever the admin could do on that page and even for the whole website, as far that is allowed.

Q4

What vulnerability can be identified in the following HTML fragment, which is displayed after a user logs in. How can the vulnerability be expoited? Describe a countermeasure to prevent the vulnerability.

```
<form action="account.php" method="post">
new amount:
<input name="newvalue">
<input type="submit" value="save">
</form>
tips:
```

The form does something that requires you to log in, so it’s something that is not available to just any user. For example, it will update admin settings.

The login creates a session cookie in the browser. The account.php code on the server is not given so assume that the problem doesn’t lie there.

None of the form’s HTML is coming from user data stored or sent to the server.

Think of the common web vulnerabilities analyzed in the I2 lab.

This form could be vulnerable to mass assignments and be abused to elevate authorizations. For example, when a field in the account table tells whether you are admin or not, one could simply try to post a field like {‘is_admin’: true} and if it saved anyway (even when the current user is unauthorized) the user can obtain admin rights, when using the correctly guessed field names.
The application can be protected from this by using so-called strong parameters that explicitly check what is submitted and limit those parameters to the permissions of the user.

Q5

You use a web application that refers to a database with the following tables.

TABLE: users, with columns: (username, password, ssn)

TABLE: products, with columns: (productcode, productname, producer, color, price)
Your web application checks the username and password using the following query, where you control both $username and $password:

SELECT userid FROM users WHERE username = ‘$username’ AND password = ‘$password’
Suppose both username and password are vulnerable (non-sanitized); craft an sql injection that allows you to enter in the system as user “sandro” without knowing the password of “sandro”,

Followup question:

Besides logging in without a password, what else could an attacker achieve? Explain your answer.

- Exploit the SQL injection vulnerability by using the following username: `sandro’; --`
- As any SQL is allowed to execute by inserting any statements after the first one, it is possible to do almost anything, for example:

    - Reflect plain passwords by displaying them in the attacker's SSN field, by updating the attacker's SSN field with the result of a query to the password column in the users table.
    - Change the price of a product you want to buy, by inserting a update query that sets the product price to the desired value.

Q6

`What is the cause of SQL injection vulnerabilities in programs? Illustrate with an example. How should programmers defend against such vulnerabilities? Can these counter measures be enforced? That is, is it possible to guarantee the absence of SQL injection attacks? Sketch the design and implementation of a language that provides such guarantees. Can this design be generalized to other types of injection attacks?`

- SQL injections exist when user data is used without any checks to compose a raw SQL string, which is then just run at the server. Because the user data is used as-is, it can also be exploited to alter the behavior of the SQL statement.
- A vulnerable example is given in the previous exercise:
    - `SELECT username FROM users WHERE username = '$username' AND password = '$password'`
    - By using `sandro'; --` as username, the composed query looks like `SELECT username FROM users WHERE username = 'sandro'; --' AND password = '$password'`, which is valid SQL to get all users with username 'sandro'. All subsequent matching statements are ignored. Instead of only ignoring the statement, alternative statements can be fed to the application for execution.
- Programmers should rarely write any raw SQL statements. By building a domain-specific language to manipulate records any given input is completely in control. This results in absence of SQL injection attacks when applied appropriately. For example, ActiveRecord and similar frameworks enhance models to have database manipulation, while they also support more complicated statements. When strictly needed, it is still possible to write raw SQL. For those cases prepared statements (as strongly or even solely suggested by the documentation of those frameworks) will prevent SQL injections flaws. Code linters can be build for complete enforcement.
- This idea can be generalized and is in fact already commonly used. For example form helpers and printing methods guard applications from respectively both XSS attacks and CSRF attacks and reflection attacks.