Prof. Paxson Spring 2013	CS 161 Computer Security	Midterm 1

PRINT your name: ______________________, ______________________
(last) (first)

I am aware of the Berkeley Campus Code of Student Conduct and acknowledge that any academic misconduct will lead to a score of zero (0) on the examination and the misconduct will be reported to the Center for Student Conduct.

SIGN your name: ____

PRINT your class account login: `cs161-`______ and SID: ______________________

Your TA's name: ____

Your section time: ____

Name of the person sitting to your left: ________________ Name of the person sitting to your right: ________________

You may consult one sheet of paper (double-sided) of notes. You may not consult other notes, textbooks, etc. Calculators and computers are not permitted. Please write your answers in the spaces provided in the test. We will not grade anything on the back of an exam page unless we are clearly told on the front of the page to look there.

You have 80 minutes. There are 5 questions, of varying credit (200 points total). The questions are of varying difficulty, so avoid spending too long on any one question.

Do not turn this page until your instructor tells you to do so.

Question:	1	2	3	4	5	Total
Points:	36	42	46	36	40	200
Score:						

Problem 1 *Principles* (36 points)

An astute attacker has just discovered a buffer overflow vulnerability present in a slightly outdated version of SSH. By sending well-crafted input at the password prompt, the exploit allows the attacker to gain a root shell.

Note that SSH servers run on TCP port 22 by default.

(a) The attacker searches for victims by connecting to TCP port 22 of the main web server for several sites whose servers he would like to compromise. Upon successfully connecting, the SSH server sends a string identifying what version it's running, which enables the attacker to determine whether the server has the vulnerability.

After some time (and with fruitless results), the attacker realizes that the main web server of one of the sites, `www.luser.com`, doesn't have an SSH server listening on port 22 at all. He "port scans" the target and finds that SSH is indeed running, but on a different, non-standard port. And that server is in fact running the vulnerable version of SSH!

Identify a relevant security principle and justify it in a sentence or two. (You only need to list one principle in cases where you think perhaps more than one applies.)

> **Solution:** *Don't rely on security through obscurity.* The site attempted to protect its vulnerable SSH server from exploitation by running it on a non-standard port, but once the attacker discovered the server, that protection no longer helped.
>
> Another possibility: *Least privilege.* The SSH server *should* have been set up such that even if it suffered from a vulnerability, that flaw would not allow the attacker to obtain root privileges.

(b) The attacker sends the evil input and, lo and behold, obtains a root shell. As the attacker pokes around the compromised system, he realizes that `www.luser.com` is also used as a staging machine to deploy source code to several other web application servers. The deployment script copies the files to target machines via SSH. To keep the copying process quick and easy, the web application developer team has enabled password-less access to the other servers, even though the site's security officers have set a policy that all access between machines must use two separate forms of authentication. The additional servers will accept any incoming SSH connection for user `root` that comes from `www.luser.com` without requiring authentication.

The attacker is delighted: his compromise of the staging server will now gain him root access to the several other servers as well!

Identify a relevant security principle and justify it in a sentence or two.

> **Solution:** *Least privilege*: copying source code to the application servers shouldn't require root access on the target machines.
>
> Alternatively, *Psychological acceptability*: The security policy places burdens on the developer team beyond what they're willing to work with, so they have subverted the policy instead.

(c) The attacker now wants to ensure that he will continue to have access to the compromised systems in the future, even if the SSH version is upgraded. Since each machine runs a web server, the attacker decides to replace the web server binary (`httpd`) with a modified version that includes a backdoor. The backdoor grants access to a root shell whenever the web server receives a special URL.

However, it turns out that each of the compromised systems is running a detection tool called `AIDE`. `AIDE` is a program that periodically takes a snapshot of a machine's file system by computing summaries of the files and recording their modification times. The site's security staff has configured `AIDE` so that whenever it detects a change to the files that it monitors, it sends out an alert email to the machine's administrator.

Thus, soon after adding the backdoor, the administrator receives a warning of a potential intrusion, since the modification time of the `httpd` binary has changed.

Identify a relevant security principle and justify it in a sentence or two.

> **Solution:** *Detect if you can't prevent*: the site will significantly benefit from having discovered the backdoor even though their security measures failed to prevent it from occurring in the first place.

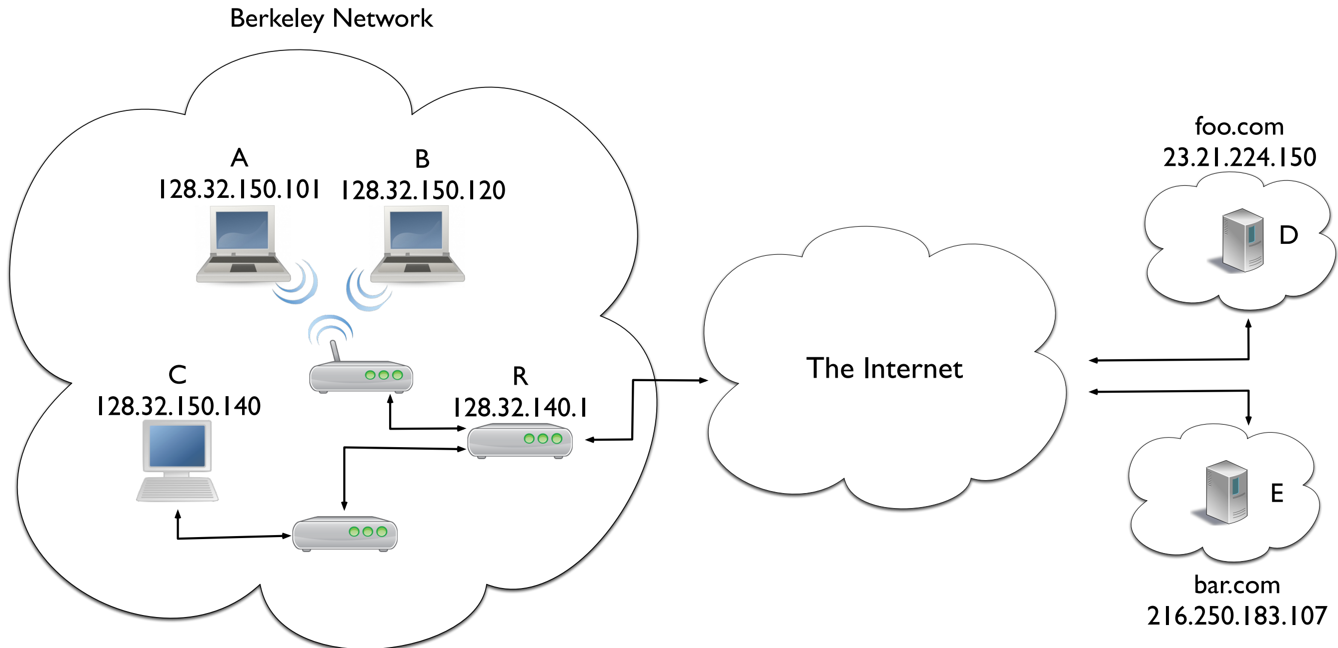
(d) Unfortunately, the security staff set the rules for the files that `AIDE` tracks too broadly. Several web server log files fall under these rules, and since these files change frequently, the administrator receives one or two dozen emails about potential integrity violations every day, and ends up ignoring them all.

Identify a relevant security principle and justify it in a sentence or two.

> **Solution:** *Consider human factors*: burdening the administrators with frequent notifications that often have no actual importance will understandably lead the administrators over time to ignore them, impairing the overall security benefits that the notifications are meant to achieve.
>
> It would also be fine to frame this as *Psychological acceptability*, with the argument that the administrators have found the alerting process not something they're willing to put up with, and thus they've decided to consciously subvert it by ignoring the notifications it produces.

Problem 2 ***Spoofing*** **(42 points)**

The following figure shows a diagram of three networks and the Internet.



The networks are configured as follows:

- Hosts A and B are laptops on the same open WiFi network, subnet 128.32.150.0/25.
- Host C is a wired desktop, subnet 128.32.150.128/25.
- R is the edge router for the Berkeley network and is on subnet 128.32.140.0/24.
- D and E are on separate networks run by different ISPs.

Assumptions for this problem are as follows:

- The networks do not employ any form of firewalling or filtering.
- All TCP implementations are modern.
- Attackers are lucky and will win timing races if they can act instantaneously upon observing an action of the victim.
- A "successful" attack requires that the attack will work without requiring more than a half dozen packets.

For each of the following attacks, **mark with an X** every host that can successfully carry out that attack against each specified victim. If no hosts can successfully carry out the attack, mark the entry for "*None*". Ignore the entries in the table with "–".

(a) (12 points) Attacker has the ability to successfully spoof DHCP offers directed at the victim, such that the victim will accept the offer as genuine:

		Attackers					
		A	B	C	D	R	*None*
Victims	A	–	X				
	C			–			X
	D				–		X

Solution: To successfully spoof a DHCP offer requires the ability to see the original DHCP request. This in turn requires an attacker connected to the same subnet as the victim. In the network considered in this problem, only A and B are connected to the same subnet.

(b) (12 points) Attacker can successfully spoof the IP source address of traffic sent to the victim to appear as though it comes from E:

		Attackers					
		A	B	C	D	R	*None*
Victims	A	–	X	X	X	X	
	C	X	X	–	X	X	
	D	X	X	X	–	X	

Solution: Since the problem stipulates that the networks do not employ any filtering or firewalling, an attacker at any of the systems considered can freely set the source address of any IP packets the attacker transmits to E's IP address, making those packets appear to A as though they came from E.

(c) (12 points) Attacker can successfully initiate and complete a TCP handshake with the victim, such that to the victim it appears as though E initiated the connection:

		Attackers					
		A	B	C	D	R	*None*
Victims	A	–	X			X	
	C			–		X	
	D				–		X

Solution: Given that the TCPs use modern implementations—and thus randomize their Initial Sequence Numbers (ISNs)—an attacker can only successfully ($\leq$ half dozen packets) spoof a TCP handshake appearing to come from E if the attacker can observe the ISN that the victim selects for its half of the connec-

tion. This requires an attacker that can either eavesdrop on traffic sent from the victim, such as B for traffic sent by A; or is itself in the forwarding path between the victim and E (to where the victim will send its responses), such as R for either A or C, since R is involved in forwarding those hosts' packets to E.

Note that none of the attackers considered in the problem can observe traffic sent from D to E.

(d) (6 points) Attacker can successfully spoof replies to the victim's HTTP GET requests made to E:

		Attackers					
		A	B	C	D	R	*None*
Victims	A	–	X			X	
	C			–		X	
	D				–		X

Solution: As in the previous part, to spoof replies to HTTP requests the attacker must observe the sequence numbers used in the traffic. This requires either the capability to eavesdrop on the traffic or that the attacker sits in the forwarding path between the victim and E. Thus, the answers to this part are the same as for the previous part.

Problem 3 *Web security* (46 points)

EasyWeb Inc.'s web server functions as follows: Whenever it receives a URL of the following form:

`http://www.easyweb.com/login.do?user=`*username*`&pass=`*password*

it logs in users assuming that the provided password indeed matches the one associated with the provided username.

Note: for this problem, do not concern yourself with network eavesdroppers, nor with whether users pick strong passwords. In addition, you can do the last three parts, **(c)** through **(e)**, separately from the first two, **(a)** and **(b)** (in case you get stuck on the first two).

(a) (8 points) **Name** and **briefly describe** a vulnerability in EasyWeb Inc.'s use of this approach. **Briefly sketch** a scenario in which an attacker would exploit the vulnerability.

> **Solution:** Cross-Site Request Forgery (CSRF). The attacker can trick a victim into logging in as the attacker by providing the victim with a URL containing the attacker's login information (a form of *Session Fixation*). After logging in as the attacker, the victim can then be tricked into divulging credit card information, paying the attacker's bill, or other such nastiness.

(b) (10 points) Briefly describe a defense against the vulnerability identified in part (a).

> **Solution:** Any of the following answers are correct.
>
> - CSRF Tokens. Require users to obtain a token from the server before logging in (such as by visiting a login page that will include tokens in it when returned from the server) and to include that token in their login request.
> - `Referer` header check. Require users to submit the login information from another page on EasyWeb Inc.'s website, and verify that the `Referer` header is a URL from EasyWeb Inc.
> - `Origin` header check. Very similar to the `Referer` header check. Require users to submit the login information from another page on EasyWeb Inc.'s website, and verify that the `Origin` header is `www.easyweb.com`.

An astute Berkeley student has submitted a bug report to EasyWeb Inc. and they have changed the way their website works. EasyWeb Inc. now requires that all login requests go through a specific login form page. However, to make the approach somewhat backwards-compatible, they designed the mechanism as follows. When a user of the service first surfs to the URL `http://www.easyweb.com/login.do?user=`*username*, the website returns a web page that conveniently pre-fills part of the login form for the user, like this:

The text *username* will be replaced with the **literal** (exact) value from the URL. The form also contains **hidden fields** that will be sent to the web server upon clicking Sign in. These fields contain, among other things, the *username* from the URL.

(c) (8 points) In using this approach, EasyWeb Inc. has introduced a *new* vulnerability while fixing the previous one. **Name** the vulnerability and **briefly describe** it.

> **Solution:** Reflected Cross-Site Scripting (XSS). Whatever is supplied as the *username* will get included on the response page. An attacker can prepare a URL containing a script as the username, which will then get executed by the victim's browsers in the context of EasyWeb Inc.'s website.

(d) (10 points) Explain how an attacker can use this new vulnerability to perform an attack. Briefly sketch how the attacker sets up the attack and what happens when the attack occurs.

> **Solution:** Either of the following answers is correct.
>
> - An attacker can now log in a victim using XSS. The `user` parameter in the URL would contain a script that changes the hidden user field to the attacker's username, fills in the password field with the attacker's password, and automatically submits the form. (Because the request is coming from the original login form, the defense from part (b) is circumvented.)
> - The attacker can include a script in the `user` parameter in the URL that monitors what's typed into the password field and sends off the field's value to the attacker.

(e) (10 points) Briefly sketch a potential defense that will prevent at least some in-

stances of the attacks enabled by the vulnerability in part (d), even if not all of them. Discuss a drawback or limitation of the defense.

Solution: Either of the following answers is correct.

- EasyWeb Inc. should sanitize the value of the `user` parameter before including it in the response. A limitation is that it is hard to get the input sanitization right, especially because the value is used twice, once as text and once as the value of an HTML attribute.
- EasyWeb Inc. could check whether the `user` parameter contains a valid user before including it in the response. Assuming they use sane usernames, it is impossible to include scripts this way. A drawback is that this allows an attacker to test for existing usernames.

Problem 4 *Short Answers* (36 points)

(a) (7 points) Suppose you implement a web **service** entirely in Java. For database access, you are careful to always use Prepared statements. Assume there are no flaws in the Java environment.

Because of the use of Java and Prepared statements, for the following **circle** all of the threats that you can confidently conclude the web service will **not** be vulnerable to:

[Buffer overflow]

CSRF

TOCTTOU vulnerabilities

[SQL injection]

Reflected XSS

Stored XSS

None of the above

> **Solution:** Writing the web server in Java ensures it will operate in a memory-safe fashion, and using Prepared statements for its database access will prevent SQL injection attacks. None of the other threats are necessarily addressed by this implementation approach.

(b) (7 points) Suppose a developer implements a full-featured web browser entirely in Java. The browser does not use a database, and again assume there are no flaws in the Java environment.

Because of the developer's use of Java, and the browser's absence of any database usage, for the following **circle** all of the threats that the developer can confidently conclude the users of the browser will **not** be vulnerable to:

[Buffer overflow]

CSRF

TOCTTOU vulnerabilities

SQL injection

Reflected XSS

Stored XSS

None of the above

Solution: As in the first part, writing the browser in Java will ensure it operates in a memory-safe fashion.

SQL injection vulnerabilities are server-side flaws, not browser flaws. When designing the question, we had intended the answer to be that users of the browser will still be vulnerable to it (i.e., it shouldn't be circled). However, in finalizing the solutions we realized it's legitimate to argue that because SQL injection concerns only servers, it's correct to conclude that users of the browser are not vulnerable to them. Thus, we accept either answer (circled or not) for SQL injection.

None of the other threats are necessarily addressed by this implementation approach, so they should not be circled.

(c) (10 points) Suppose an attacker wishes to disrupt access to a web-based service.

If the attacker finds they lack the resources to launch a successful *network-layer* DoS attack, they might still be able to launch a successful *transport-layer* DoS attack: **[TRUE]** **FALSE**

If the attacker finds they lack the resources to launch a successful *network-layer* DoS attack, they might still be able to launch a successful *application-layer* DoS attack: **[TRUE]** **FALSE**

If the attacker finds they lack the resources to launch a successful *application-layer* DoS attack, they might still be able to launch a successful *network-layer* DoS attack: **[TRUE]** **FALSE**

DoS attacks fundamentally concern exhausting some sort of resource required by the target: **TRUE** **[FALSE]**

Fuzz testing can be used to find some forms of DoS vulnerabilities: **[TRUE]** **FALSE**

Solution: An important characteristic of DoS attacks is that what's required to stress a service at one protocol layer is generally completely separate from what might stress it at another layer. For example, a successful network-layer attack will generally involve overwhelming the data capacity of a network link or the forwarding capacity of a router along the network path to the target. A successful transport-layer attack might involve SYN flooding, while an application-layer attack might be done in terms of making expensive requests to the target service. The inability to have sufficient resources to successfully conduct one of these does not preclude being able to conduct the others, hence the first three are all **TRUE**.

Regarding the last two, (1) attackers can induce denial-of-service not only via resource exhaustion, but also instead by exploiting program flaws that lead directly to crashes (such as inputs that cause dereferencing of NULL pointers), and (2) any time fuzz testing manages to crash a server, it has found an input that can be used to deny service for the associated service.

(d) (12 points) Company A requires that its employees pick 6-character passwords made up of combinations of lowercase letters, uppercase letters, and digits. No other characters are allowed, and a given user's password must not use any character twice.

Company B requires that its employees pick 12-character passwords, where each of the 12 can be any of 100 possible characters. Unlike for Company A, Company B's employees *can* reuse characters in their passwords. However, Company B finds that users often make mistakes with these long passwords, so if an authentication attempt fails, the login server helps the user by telling them how many of the initial letters were correct. For example, if a password entered was `"abcdefghij"` and the server replies *Wrong, but the first 4 letters were correct*, then `"abcd"` are correct, `'e'` is wrong, and nothing is revealed about the correctness of the letters after `'e'`.

Suppose an attacker is using exhaustive brute force to guess the password of user U1 at Company A, and user U2 at Company B. Both usernames are valid at the respective companies and the users have chosen passwords that conform with the policy.

1. Write down an expression for the maximum number of attempts the attacker needs for guessing the password of user U1 at Company A.

Solution: There are 62 possibilities for the first character in the user's password, 61 possibilities for the second character, 60 possibilities for the third, and so on. Therefore an expression for the total number of possibilities that the attacker must try is:

$$\#\text{ possible passwords} = 62 \cdot 61 \cdot 60 \cdot 59 \cdot 58 \cdot 57$$

2. Write down an expression for the maximum number of attempts the attacker needs for guessing the password of user U2 at Company B.

Solution: The key for this part of the problem is that the attacker can quickly learn from their initial guesses; the feedback provided by the login process enables the attacker to *decouple* determining one letter in the user's password from the next.

To start, the attacker can try 100 passwords that each differ in their first character. One of these must succeed. In addition, when it succeeds, in the worst case the attacker is told that the second character in the attempted password is incorrect. Therefore, once the attacker learns that the first character is correct, they also can eliminate 1 of the possibilities for the second character.

At this point, they make another $100 - 1 = 99$ guesses, each of which uses the first character learned in the previous step, and tries a different second character (excluding the character that the attacker has already learned is not correct for the second position).

This process continues until they try candidates for all 12 positions, requiring at worst a total of:

$$\begin{aligned} \text{\# possible passwords} &= 100 + 99 + 99 + \dots \\ &= 100 + 99 \cdot 11 \\ &= 1189 \end{aligned}$$

Problem 5 ***Software Vulnerabilities*** **(40 points)**

For the following code, assume an attacker can control the value of `basket` passed into `eval_basket()`. The value of `n` is constrained to correctly reflect the number of elements in `basket`.

The code includes several security vulnerabilities. **Circle *three* such vulnerabilities** in the code and **briefly explain** each of the three.

```
1  struct food {
2     char name[1024];
3     int calories;
4  };
5
6  /* Evaluate a shopping basket with at most 32 food items.
7     Returns the number of low-calorie items, or -1 on a problem. */
8  int eval_basket(struct food basket[], size_t n)
9  {
10    struct food good[32];
11    char bad[1024], cmd[1024];
12    int i, total = 0, ngood = 0, size_bad = 0;
13
14    if (n > 32)
15       return -1;
16
17    for ( i = 0; i <= n; ++i ) {
18       if (basket[i].calories < 100)
19          good[ngood++] = basket[i];
20       else if (basket[i].calories > 500) {
21          size_t len = strlen(basket[i].name);
22          snprintf(bad + size_bad, len, "%s ", basket[i].name);
23          size_bad += len;
24       }
25
26       total += basket[i].calories;
27    }
28
29    if (total > 2500) {
30       const char *fmt = "health-factor --calories %d --bad-items %s";
31       fprintf(stderr, "lots of calories!");
32       snprintf(cmd, sizeof cmd, fmt, total, bad);
33       system(cmd);
34    }
35
36    return ngood;
37 }
```

Reminder: `strlen` calculates the length of a string, not including the terminating `'\0'` character. `snprintf(buf, len, fmt, ...)` works like `printf`, but instead writes to `buf`, and won't write more than `len - 1` characters. It terminates the characters written with a `'\0'`. `system` runs the shell command given by its first argument.

Solution: There are significant vulnerabilities at lines **17/19**, **22**, and **33**.

Line **17** has a fencepost error: the conditional test should be `i < n` rather than `i <= n`. The test at line **14** assures that `n` doesn't exceed 32, but if it's equal to 32, and if all of the items in `basket` are "good", then the assignment at line **19** will write past the end of `good`, representing a buffer overflow vulnerability.

At line **22**, there's an error in that the length passed to `snprintf` is *supposed* to be the available space in the buffer (which would be `sizeof bad - size_bad`), but instead it's the length of the string being copied (along with a blank) into the buffer. Therefore by supplying large names for items in `basket`, the attacker can write past the end of `bad` at this point, again representing a buffer overflow vulnerability.

At line **33**, a shell command is run based on the contents of `cmd`, which in turn includes values from `bad`, which in turn is derived from input provided by the attacker. That input could include shell command characters such as pipes ('`|`') or command separators ('`;`'), facilitating *command injection*.

Some more minor issues concern the `name` strings in `basket` possibly not being correctly terminated with `'\0'`s, which could lead to reading of memory outside of `basket` at line **21** or line **22**.

Note that there are no issues with format string vulnerabilities at any of lines **22**, **31**, or **32**. For each of those, the format itself does not include any elements under the control of the attacker.