Paxson
Spring 2011

CS 161 Computer Security

Midterm 1

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

SIGN your name: ____

PRINT your class account login: `cs161-`____________

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 7 questions, of varying credit (100 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	Points	Score
Short Answer	18	
Memory Safety	15	
Denial-of-service	20	
Spoofing	15	
Surviving Hot Spots	12	
Javascript	10	
XSS Defense	10	
Total:	100	

Problem 1 *Short Answer* (18 points)

(a) (3 points) The Soda Hall elevator requires card key access to go to the 4th floor. The door connecting the west stairwell and the 4th floor opens without cardkey. What security principle is missing in this design?

CIRCLE the best answer, and briefly explain.

(i) Fail-safe defaults. (ii) Complete mediation.
(iii) Separation of responsibilities. (iv) Least privilege.
(v) Human factors matter.

Justification:

Solution: The best answer is **(ii)**. Complete mediation refers to ensuring that all accesses to a protected resource go through the intended mechanism for access control.

(b) (3 points) Agree or disagree, and briefly explain: *It's better to use library calls with rich functionality, like `system()`, than those with more narrow functionality, like `execve()`, because then your code that uses the library is simpler, and so more likely to be correct.*

Solution: Disagree. A "feature-rich API" such as that provided by `system()` presents many possibilities for an attacker to exploit functionality provided by the API, even though that functionality is not needed by your code.

(c) (3 points) Agree or disagree, and briefly explain: *When sanitizing inputs to avoid injection attacks, it's better to use a white-list approach than a black-list approach.*

Solution: Agree. With black-listing, you have to identify *all* of the ways that an attacker might construct a problematic input. This is harder, and fails in an unsafe manner, compared to identifying only the sorts of inputs that you want to allow.

(Con't)

(d) (3 points) Agree or disagree, and briefly explain: *You can tell whether a web page your browser displays represents a phishing attack on* `mybank.com` *by checking whether the URL in the address bar starts with* `http://mybank.com`*.*

> **Solution: Disagree**. Phishing attacks can be based on URLs that *start* with a legitimate-looking prefix, but the actual domain name in the URL continues further, leading to a different host. A simple example is `http://mybank.com.badguy.com`. A more complex example is the use of non-ASCII characters that look like `'/'`, but in fact are not separators.

(e) (3 points) Suppose you have both the source code and a binary corresponding to a 5,000,000-line C program. Explain a technique you could use to try to discover whether it has a memory-safety vulnerability.

> **Solution:** One technique you could use is *fuzz testing*, where you supply random inputs to the program to see if you can induce a crash. Another technique would be to search the source code for the use of unsafe functions such as `gets()`.
>
> Approaches such as inspecting each pointer/array access, or proving invariants, are less viable, given the large code size.

(f) (3 points) List three problems with using CAPTCHAs.

> **Solution:** Problems include:
>
> - The "arms race" driven by increasingly powerful automated techniques for solving CAPTCHAs.
> - Related to this, usability issues as CAPTCHAs become hard for humans to solve.
> - Denial-of-service vulnerabilities for CAPTCHAs that require significant computation to generate.
> - "Out-sourcing" attacks whereby attackers get humans to solve CAPTCHAs for them.
> - Accessibility issues, such as for blind users who are asked to solve visual CAPTCHAs.

- The inability for CAPTCHAs to discriminate between malicious automated activity versus benign automated activity (such as legitimate web crawlers).

Problem 2 ***Memory Safety*** **(15 points)**

For the following code, assume an attacker can control the values of `item` and `n` passed into `print_report()`. CIRCLE any memory safety problems in the code and briefly explain them; or, if you believe there are no such problems, explain why the code is safe.

```c
/* Returns in "plural" the plural of the given string "str". */
void plural(char* str, char* plural)
{
    char* buf;
    size_t n;

    plural[0] = '\0';

    if ( str == 0 ) return;

    n = strlen(str);
    if ( n == 0 ) return;

    buf = malloc(n+1);
    if ( buf == 0 ) return;

    char last = str[n-1];
    if ( last == 's' )
        /* Assume it's already plural. */
        strcpy(buf, str);
    else if ( last == 'h' )
        /* fancy plural, like church -> churches */
        sprintf(buf, "%ses", str);
    else /* regular plural */
        sprintf(buf, "%ss", str);

    strcpy(plural, buf);
    free(buf);
}

void print_report(char *item, int n)
{
    char item_plural[256];
    plural(item, item_plural);
    printf("We sold %d %s\n", n, item_plural);
}
```

Reminder: `strlen(s)` calculates the length of the string s, not including the terminating `'\0'` character. `strcpy(dst, src)` copies the string pointed to by `src` to `dst`, including the terminating `'\0'` character. `sprintf` works exactly like `printf`, but instead writes to the string pointed to by the first argument. It terminates the characters written with a `'\0'`.

Solution: There are memory safety problems at lines **23**, **25**, and **27**.

The first two of these will write past the end of the heap memory allocated for `buf`, since it only has room for the characters in `str` plus a terminating `'\0'` (due to using `n+1` in the call to `malloc()`). On the other hand, the `strcpy()` call at line 20 does not present a memory safety problem because there is (just barely) enough room in `buf` to hold a copy of `str`, including the terminating character.

The problem at line **27** arises because at line **33** the code declares `item_plural` to have enough storage to hold 256 characters. Therefore if the attacker supplies a value of `item` whose plural (including the terminator) exceeds 256 bytes in size, a stack overflow will result when line **27** executes the call to **strcpy()**.

Note that the code does *not* have a format-string vulnerability. Such vulnerabilities only arise when the *format* argument in a call to a function such as `printf()` or `sprintf()` comes from a value controlled by the attacker. In this code, the calls at lines **23**, **25**, and **35** all instead use fixed formats.

There is a potential integer overflow if the attacker passes in a string of length $2^{32} - 1$. This will cause the `malloc()` call at line **14** to have a size of zero, which the C standard states has undefined behavior. It was fine to point out this problem, but also fine to not mention it.

Problem 3 *Denial-of-service* (20 points)

An anti-spam company, GreenMail, uses a vigilante approach to fighting spam.[1] GreenMail's customers report their spam to GreenMail, and the company then automatically visits the websites advertised by the URLs in the spam messages and leaves complaints on those websites. For each spam that a user reports, GreenMail leaves a generic complaint. GreenMail operates on the assumption that as the community grows, the flow of complaints from hundreds of thousands of computers will apply enough pressure on spammers and their clients to convince them to stop spamming.

After a short while of operation, GreenMail's public web site comes under a massive DDoS attack that uses SYN flooding.

(a) (3 points) Briefly describe the type of traffic that an attacker sends to launch a SYN flooding attack.

> **Solution:** A SYN flooding attack sends a stream of TCP "initial SYN" packets to the targeted server. Each packet appears to represent a request to establish a new connection.
>
> Note that the attacker *may* spoof the source addresses of such SYNs to make them harder for the defender to filter them out, but this is not required. An attack that employs a large botnet, for example, might not use spoofing.

(b) (3 points) Briefly describe how the attack can cause a denial-of-service.

> **Solution:** For each incoming SYN packet, the server both responds and consumes memory because it records information (state) associated with the impending new connection. The attack primarily aims to exhaust the server's available memory for keeping this state.

(Con't)

[1] FYI, this is based on an actual company and its experiences.

(c) (6 points) Can GreenMail use a packet-filter firewall to defend itself against the DDoS that uses SYN flooding?

If so, describe what sort of rule or rules the firewall would need to apply, and what "collateral damage" the rules would incur.

If not, explain why not.

Solution: Here are two possible answers:

(1) If the flood uses a fixed number (not too large) of IP source addresses in its packets, then the target could install a number of firewall rules that deny traffic from those addresses. In this case, the collateral damage depends on how much legitimate traffic also comes from those addresses.

(2) If the flood uses a very large number of IP source addresses, either by employing a large number of different systems ("bots") to send the traffic, or by spoofing the IP source address in each SYN packet, then the target will not be able to specify enough firewall rules to defend against the attack. Note that the target cannot use a rule such as "drop any incoming TCP SYN sent to our web server" without enabling the attack to fully succeed, i.e., the collateral damage would be that no legitimate traffic can reach the server.

It's also possible that the SYN flood traffic would have fields in its packet headers *other* than the source IP address that do not appear in legitimate traffic to the site. If so (and this in fact occurs in practice), then the target can set up their firewall to filter on those header fields, rather than the source IP addresses.

(d) (4 points) Explain how the GreenMail service could itself be used to mount a DoS attack.

Solution: An attacker could send a large number of bogus spam reports to GreenMail, falsely indicating some victim site V has been sending spam. GreenMail's servers will then visit V to lodge complaints, overwhelming V in the process if the volume of visits is high enough.

(e) (4 points) Briefly describe one approach that victims could use to defend themselves against the attack you sketched.

Solution: V could refuse to accept incoming connections from GreenMail in order to avoid the load from GreenMail's servers registering complaints.

Problem 4 ***Spoofing*** **(15 points)**

Suppose we could deploy a mechanism that would ensure IP source addresses always correspond to the actual sender of a packet—in other words, suppose it is impossible for an attacker to spoof source addresses. For each of the following threats, explain whether (and briefly why) the mechanism would either:

(i) Completely eliminate the threat.

(ii) Eliminate some instances of the threat, but not all of them.

(iii) Have no impact on the threat.

Here are the threats:

(a) (3 points) Reflected cross-site scripting (XSS) attacks.

Solution: (iii) Reflected XSS attacks are conducted over HTTP, which uses an established TCP connection. The attack does not require spoofing in any form.

(b) (3 points) Kaminsky's DNS cache poisoning attack.

Solution: (i) The Kaminsky attack is based on repeatedly spoofing DNS replies until ultimately one of them succeeds in having the correct transaction identifier.

That said, DNS resolvers used to in fact accept replies from source IP addresses other than those to which they sent a given request, providing that the transaction identifier matches. This behavior was only briefly mentioned in lecture, however. So answers that explicitly state why the attack can be conducted without spoofing are also acceptable.

(c) (3 points) DoS "amplification" where the attacker sends traffic to an Internet "broadcast" address.

Solution: (i) This attack works by spoofing traffic seemingly from the victim and sent to the broadcast address of an Internet subnet. All of the hosts on the subnet then generate replies to the victim, leading to amplification.

(d) (3 points) Clickjacking.

Solution: (iii) Attacks involving confusing the user regarding with which UI elements do not require spoofing of *IP addresses* in any form.

(e) (3 points) Format string vulnerabilities.

Solution: (iii) These vulnerabilities are problems in software; IP addresses do not have any relevance for them.

Problem 5 ***Surviving Hot Spots*** **(12 points)**

You go into Javalicious, a coffee shop with free WiFi. You settle in for an afternoon of web-surfing and tweeting. You learn that the network sends all packets unencrypted. You find this disconcerting. Worse, you see Prof. Evil seated at the table next to yours, using a laptop connected to the same WiFi network.

For your HTTP web connections, consider the basic security properties of *confidentiality*, *integrity*, and *availability*. For each of these, Circle YES if Prof. Evil can undermine the given property, or NO if not. Justify your answer.

Confidentiality (4 points): YES, can undermine NO, can't undermine

Solution: YES. Confidentiality refers to being able to keep one's data and communication private, if desired. However, on an unencrypted network Prof. Evil will be able to "sniff" your HTTP traffic and observe your web connections in their entirety.

Integrity (4 points): YES, can undermine NO, can't undermine

Solution: YES. Integrity refers to your data and communication remaining free from alteration by an attacker. However, the ability to readily observe your traffic, including TCP sequence numbers, then allows Prof. Evil to then *inject* traffic that receivers (either your browser, or the web servers with which you communicate) will accept as legitimate.

Availability (4 points): YES, can undermine NO, can't undermine

Solution: YES. Availability refers to being able to access data and services when you wish. Prof. Evil can undermine this in a number of ways. One simple example is by using **RST injection**, i.e., sending spoofed TCP RST packets that your browser (or the remote servers you communicate with) will interpret as meaning to terminate your HTTP connections. Such spoofing is easy for Prof. Evil to perform given the ability to eavesdrop on your traffic.

Problem 6 ***Javascript*** **(10 points)**

Suppose a user turns Javascript completely off in their browser. For each of the following threats, indicate whether (and briefly explain why) doing so would either:

(i) Completely eliminate the threat.

(ii) Eliminate some instances of the threat, but not all of them.

(iii) Have no impact on the threat.

Here are the threats:

(a) (2 points) Cross-site request forgery (CSRF).

Solution: (iii). In general, CSRF attacks do not have a Javascript component. An attacker conducts them by manipulating a victim into fetching a URL that has side effects beneficial to the attacker.

Or possibly **(ii)**: some forms of manipulation might require the browser to process Javascript. Answers that explicitly point out this possibility are also acceptable.

(b) (2 points) SQL injection.

Solution: (iii). SQL injection attacks target web *servers*, not browsers. (Again, one could argue that certain attacks may require some Javascript to set up. Answers specifying **(ii)** that explicitly point out this possibility are also acceptable.)

(c) (2 points) Reflected cross-site scripting (XSS).

Solution: (i). The whole goal of a reflected XSS attack is to get a victim's browser to execute malicious Javascript. (Here, an answer of **(ii)** is also acceptable if accompanied by the observation that one could also reflect *other* scripting languages, such as Flash.)

(d) (2 points) Stored cross-site scripting (XSS).

Solution: (i). Similarly, the whole goal of a stored XSS attack is to get a victim's browser to execute malicious Javascript. (Again an answer of **(ii)**

is acceptable if accompanied by the observation that one could also induce execution of other scripting languages.)

(e) (2 points) Phishing.

Solution: (iii). Phishing in general does not need Javascript. Such attacks simply need to present a user with a convincing impersonation of another party (e.g., email sender or web site) that the user trusts.

The impersonation *could* involve Javascript in order to render a suitably convincing web page for the user, so answers specifying **(ii)** that frame this consideration are also acceptable..

Problem 7 ***XSS Defense*** **(10 points)**

Suppose a browser attempts to protect the user from XSS attacks as follows. Any time the user clicks on a URL link on a page the browser is displaying, the browser scans the URL looking for any text that matches the pattern `/<[^>]*>/`. This regular expression will match an open-tag character (`<`), followed by an arbitrary number of characters other than a close-tag character, followed by a close-tag character (`>`).

If the browser finds a match, it remembers the associated text. It then scans the response it receives for the URL for any instance of that text. If it finds the same text in the response as it did in the match, then it will interpret the response as a pure HTML document, and refuse to execute any script that it contains.

(a) (5 points) Briefly explain to what degree this mechanism protects against reflected XSS.

Solution: In a reflected XSS attack, a URL fetched by the browser (typically due to a user clicking on it) includes within it characters that the server will echo in its response. The user's browser then interprets this response as including a script due to the presence within the response of tags such as `<script>`.

The regular expression given in the problem will match any occurrences of such tags, and thus will prevent reflected XSS attacks where a tag with the exact same name appears reflected in the server output.

However, if the server *processes* the URL in some manner before reflecting it, then the reflection might not directly correspond to the text inspected by the regular expression. For example, using the normal HTTP hex-escape rules, a server will translate a received URL that includes "`...%3cscript%3e...`" to "`...<script>...`". The regular expression will miss matching the original URL, since it does not include literal `<` and `>` characters, but the browser will still treat the response as including a `<script>` tag.

Thus, this defense will defeat some reflected XSS attacks, but not all of them.

(b) (5 points) Briefly explain to what degree this mechanism protects against stored XSS.

Solution: In a *stored* XSS attack, the URL used to retrieve the injected attack might not have any tags within it at all. These attacks have two separate connections, one in which the attacker uploads content to a web site that includes an embedded script, and the other where the victim visits a page that returns that content. The victim's visit can be via an ordinary URL that doesn't have any tags within it.

The browser mechanism could detect the *initial* phase of the attack, where the attacker accesses the web service and uploads their script for storage, if as part of the upload process the web server shows a page that includes the now-stored script. However, the browser detecting this event will not *protect* against the attack, because the detection occurs by the attacker's browser (so the attacker is free to ignore the detection), not the victim's.