CS 161 Spring 2010 — Computer Security Paxson/Wagner — HW 1 Soln

1. (20 pts.) A simple web service

You need to write a web service that accepts trouble reports via a web form and then forwards them to a system administrator. More specifically, the web service should take the text of a message written by the user explaining the problem and a name for the problem supplied by the user, and your program should email this information to `admin@mysite.com`. You do so by invoking the `mail` program and providing it the problem explanation on stdin, and the problem name in the email subject via a command-line argument.

Your find the following code on the web to do just this:

```
void send_mail(char *problem_report, char *problem_name)
{
  FILE *mail_stdin;
  char buf[512];
  sprintf(buf, "mail -s \"Problem: %s\" admin@mysite.com",
          problem_name);

  mail_stdin = popen(buf, "w");
  fprintf(mail_stdin, problem_report);
  pclose(mail_stdin);
}
```

Identify at least three security problems with this code. For each problem, provide or describe an example of an input that would demonstrate or illustrate the existence of the problem.

HINT: Familiarize yourself with the workings of `popen()` and `pclose()` if they are new to you. You can read the manual pages for `popen()` by typing `man popen` at a shell prompt on a Unix system.

Solution:

1. If the user input `problem_name` is longer than 475 bytes, there is a buffer overflow in the call to `sprintf`, as it does not check the bounds of the buffer it is writing to.
2. The invocation of `fprintf` is a format string vulnerability because if the user input `problem_report` contains any format specifiers, they can be used to access or modify the stack.
3. There is a command injection vulnerability associated with `problem_name`. It could be exploited by introducing a double-quote character into `problem_name` to escape out of the quoted string. Even more malicious, if the input also contains a semicolon, the shell executed by `popen` will allow for the execution of arbitrary commands after the semicolon. For instance, if `problem_name` is the string "`none" foo@bar.com; rm -rf /; echo "`", this could potentially erase the entire file system, because the shell will be invoked with the following string:
   `mail -s "Problem: none" foo@bar.com; rm -rf /; echo "" admin@mysite.com`

2. (20 pts.) Security principles

Identify the security principle(s) relevant to each of the following scenarios, giving a one or two sentence explanation for each:

(a) At a closed event where the President will be speaking, there are security guards and metal detectors at the door, despite the presence of Secret Service agents throughout the hall.

Solution: *Defense in depth.* One line of security is not enough. If something unforeseen goes wrong with the first security guards, we want to ensure that there is a second layer to protect the President.

(b) On trains, there is often a "dead man's switch" which must be pressed down at all times for the train to be in motion. If something were to happen to the driver, the switch would be released and the train's brakes would be applied.

Solution: *Use fail-safe defaults.* If something goes wrong with the standard system, assume the worst. Usually, this means stopping all forward progress to make sure that nothing bad happens.

(c) On overnight school trips, a teacher places masking tape on the outside of all the students' hotel room doors, connecting the door to the frame, so that in the morning the teachers will have a way to know if one of the students snuck out in the middle of the night without permission.

Solution: *Detect if you can't prevent.* Teachers need to sleep, too. Since no one wants to stay up all night guarding the doors, teachers will detect those students who sneaked out and punish them later. This has the added benefit of acting as a deterrent to future rule violators.

(d) San Francisco Muni buses require those who need to buy tickets to enter at the front. However, the rear doors are often left open for passengers to disembark, and people sometimes jump on at the back of the bus without paying the fare.

Solution: *Ensure complete mediation.* In this case, Muni does not ensure complete mediation. For the sake of expediency, the buses provide an alternate entrance that is unregulated. This allows one to enter the bus without paying.

(e) Electronic music distributors sold their music with DRM (Digital Restriction/Rights Management) so that users were limited in how they could use the music. In response, some users either used software to break the DRM or turned to legally questionable sources for their music.

Solution: *Psychological acceptability* and/or *security is economics.* The people who most commonly used the system did not accept the security system put in place, so they actively worked around it. Because of this, people also stopped buying music with DRM. Thus, for economic reasons, the major music vendors dropped DRM because it was an economic hindrance. Although it provided greater security for the music labels, it decreased their sales.

3. (20 pts.) XSS

Prof. Hinkley comes up with what he thinks is a great solution to the problem of cross-site scripting vulnerabilities. He suggests introducing a new HTML tag, `<NOJAVASCRIPT>`. In between `<NOJAVASCRIPT>` and `</NOJAVASCRIPT>`, JavaScript is disabled: browsers are supposed to ignore (and in particular, not execute) any JavaScript that may occur between these two tags. Prof. Hinkley suggests that web developers can use this to avoid cross-site scripting attacks: they should surround every place in their HTML page where they are including untrusted content with a `<NOJAVASCRIPT>` tag. For instance, consider the following vulnerable code:

```
w.write("Hello, " + name + "!  Welcome back.\n");
```

Because `name` comes from user input, the above code has a XSS vulnerability. Prof. Hinkley proposes that instead of writing code like the above, the web developer should use

```
w.write("Hello, <NOJAVASCRIPT>" + name
        + "</NOJAVASCRIPT>!  Welcome back.\n");
```

Similarly, instead of writing

```
w.write("Today's most popular link is: "
        + "<A HREF=\"" + url + "\">" + url + "</A>\n");
```

(which may be vulnerable, since `url` comes from user input), Prof. Hinkley proposes the web developer should write

```
w.write("Today's most popular link is: "
        + "<NOJAVASCRIPT><A HREF=\"" + url
        + "\">" + url + "</A></NOJAVASCRIPT>\n");
```

List at least two problems with Prof. Hinkley's proposal.

Solution:

1. If the attacker injects `</NOJAVASCRIPT> ... malicious stuff ... <NOJAVASCRIPT>`, he can escape the special tags.
2. JavaScript is not the only kind of malicious content. The attacker can inject other kinds of malicious active content, such as Flash, Java, etc. This is a case of incomplete mediation.
3. It might be possible to inject a link tag that executes a CSRF attack against the site itself, if the site does not protect itself from CSRF. Or it might be possible to inject malicious HTML content onto the page without introducing any Javascript: for instance, the attacker might be able to inject a login form that is designed to capture credentials from the user and send them elsewhere, essentially mounting a phishing attack against the user viewing the page.
4. Prof. Hinkley's proposal only protects against traditional XSS attacks. It does not protect against DOM-based XSS attacks (also known as client-side XSS), a type of XSS attack that attacks JavaScript that is already running on the page[1]. In particular, if the existing JavaScript uses user input safely (e.g., extracting text from the page and then using it to dynamically add HTML to the page, using `document.write` or `setInnerHtml`), then it might be vulnerable to XSS attacks. These attacks are known as DOM-based XSS, and are not prevented by Prof. Hinkley's proposal. Thus, if your page introduces legitimate JavaScript at some other point, your page may still be at risk, depending upon how carefully that JavaScript was written.
5. Depending on how `<NOJAVASCRIPT><IFRAME SRC="http://othersite.com/...">` is handled, it might be the case that you could disable JavaScript on the page included in the iframe. This would defeat frame busting and maybe other security mechanisms that the iframe'd website put in place. Thus, this defense against XSS could have the unintended consequence of introducing other security problems unrelated to XSS.
6. There are major deployment issues with this approach. You still have all of the old web pages out there that do not use the special tags, and it does not protect users with legacy browsers.

4. (20 pts.) Spot the bug

Here's a real security hole that occurred in `xterm`. At the time, the `xterm` application ran with permissions that let it read or write any file (for various reasons that aren't important here). `xterm` had a feature that allowed the user to enable logging, so they could log their terminal session to a file of their choice. Of

[1] See, e.g., `http://en.wikipedia.org/wiki/Cross-site_scripting` or `http://www.owasp.org/index.php/DOM_Based_XSS`

course, the user should only be allowed to enable logging to a file that the user has permission to write (for instance, we don't want to allow the user to overwrite the global password file, `/etc/passwd`). Therefore, `xterm` used the following code to ensure that the user has permission to write to the logfile, before writing to it:

```
if (access (logfile, W_OK) < 0)
    return ERROR;
fd = open (logfile, O_CREAT|O_WRONLY|O_TRUNC, 0666);
/* ... write to fd ... */
```

In this code, `access(logfile, W_OK)` checks file permissions on the specified file to see whether the user who launched `xterm` has permission to write to the specified file[2].

This code has a security vulnerability. What is it?

Solution: This is an example of a TOCTTOU (time of use to time of check) vulnerability. Because the access control check and the opening of the file are not atomic, it is possible that the state of the system will change between the two, defeating the access control check. This is an extra serious problem because you, the user, run xterm, but it runs as a root process in order to execute all of its funcitonality.

Consider the following scenario. You have a file that you own, `/tmp/foobar`, that you establish a symlink to. You ask xterm to begin logging and give it the location of the symlink as the logfile to write to. When xterm calls `access()` to perform the access check, it passes because you own the file that the symlink points to. However, after the access check, you quickly remove the symlink and reassign it to `/var/log/syslog`, which is owned by root. Since the access control check has already passed, xterm does not know that you, the user, do not really have access to `/var/log/syslog`. xterm happily opens, truncates, and writes over it as root.

On first glance, this attack might sound difficult to mount, because there is such a tiny time window between the `access()` and `open()` calls. However, researchers—some of them from Berkeley—have discovered a number of ways that these vulnerabilities can be exploited with nearly-perfect success rate, by forcing the `access()` or `open()` system calls to take a very long time[3]. As a result, you'd better assume that a knowledgeable attacker can exploit this vulnerability successfully.

5. (20 pts.) Cookies

When a site sets a cookie on your browser, the cookie is typically associated with the domain of the server, or a set of domains. For instance, when I visit `http://www.paypal.com/`, Paypal sets a cookie on my browser that my browser will send back to the server any time I visit a page on `www.paypal.com`. However, the browser will not send this cookie to other third-party sites, like (say) `blogger.com` or `ha.ckers.org`, due to the *Same Origin Policy*. Similarly, pages from `www.paypal.com` can set this cookie to a different value at any time, but other third-party sites (like `blogger.com` or `ha.ckers.org`) cannot overwrite the value of this cookie.

(a) Why is it important that third-party sites must not be able to see the cookies set by Paypal? What could go wrong if they could?

[2] If you're curious, `xterm` ran with its effective UID set to 0, i.e., root, and its real UID set to the userid of the user who launched `xterm`. The manual pages for `access()` and `open()` specify their behavior in a bit more detail. However you shouldn't need to know all this to answer the question

[3] See, e.g., `http://www.cs.berkeley.edu/~daw/papers/races-usenix05.pdf` or `http://www.cs.sunysb.edu/~rob/papers/races2.pdf`

Solution: There is sensitive information that is stored in cookies. For example, there are session ids that, if an attacker got a hold of, would allow him to start his own connection to Paypal logged in as the victim.

(b) Why is it important that third-party sites must not be able to overwrite the cookies set by Paypal? What could go wrong if they could?

Solution: The main problem is session fixation. An attacker who controls some third-party web site could set a session id in a victim's Paypal cookie so that the attacker knows the session id of the victim's session with at Paypal. This would allow the attacker to jump in midstream after the victim has entered her username and password into Paypal, edit his browser's cookie jar so that he appears to be associated with the victim's session, and thereby join the victim's session, issuing requests that Paypal will think came from the victim.

Another variant of this is, for example, if the attacker had earlier logged into Google, he could set the victim's session id to his own session id. Then when the victim goes to Google search, all of her search history would be stored in the attacker's history for the attacker to view at his leisure.

See Section 3.3 from the optional reading for 2/5/2010 for more information on these kinds of attacks, or `http://keepitlocked.net/archive/2007/12.aspx` or `http://projects.webappsec.org/Session-Fixation`.