Paxson
Spring 2013

CS 161 Computer Security

Homework 2

Due: Wednesday, March 6, at 10PM
Version 1.1 (02Mar13)

Instructions. This assignment must be done on your own, and in accordance with the course policies as framed on the class web page. Submission is electronic. To submit, put a single solution file `hw2.pdf` in a directory and run `submit hw2` from that directory. Put your name, your class account name (e.g., `cs161-xy`), your TA's name, and "HW2" prominently on the first page.

Version 1.1: we adjusted problem 4(e) to remove the previous requirement "*For full credit, your solution should not require the server to maintain extra server-side state.*" This will not be required for full credit.

Problem 1 *Work Factor* (25 points)

We use the term *work factor* to refer to how much effort an attacker must expend to achieve a particular attack. Some defensive techniques, such as introducing randomization, aim to significantly increase the attacker's work factor, hopefully enough to render a given attack infeasible given the resources available to the attacker. One such example presented in lecture is the randomization of TCP Initial Sequence Numbers in order to make blind spoofing of valid (non-SYN) TCP packets hard.

This problem examines a couple of approaches to increasing an attacker's work factor—how effective they are, and whether their use might come with some drawbacks.

Recently, many sites have seen large increases in the frequency of password-guessing attempts conducted against their SSH servers. Attackers sometimes try a large number of different passwords for well-known or likely-to-exist account names such as `admin`. An attacker who can *exhaustively* try all possible passwords will indeed ultimately discover the correct one, granting them illicit access to the account.

(a) Suppose a site's passwords were originally 6 characters long, and each character is either an uppercase letter, a lowercase letter, or a digit. In addition, suppose that an attacker can try 1,000 possible passwords every second. Approximately how many days will it take an attacker to discover a given account's password by brute force?

(b) One technique attackers use for guessing passwords is to compile large dictionaries of common password choices. Suppose the site knows that attackers are currently using a dictionary with 2,000,000 entries, and that the site's users indeed pick passwords present in that dictionary. When trying to brute-force access to a given account, attackers will methodically try every password in the dictionary.

To combat this, the site requires its users to append two additional randomly selected characters to their passwords (each again either an uppercase or lowercase letter, or a digit).

Assume the attacker knows that the site has made this change and modifies their brute-force search to try all possible combinations of two appended characters to each password in their dictionary. Approximately how many days will it take an attacker to discover a given account's password by brute force?

(c) Another technique the site considers is to run its publicly accessible SSH server not on the standard SSH port (which is TCP port 22) but on a different TCP port. Assume the attacker has no way of knowing the new port other than repeatedly attempting connections to the server, but the attacker can tell when they have found the correct port (since now they get an answer from it).

Given that the site also uses the technique in the previous question (two additional randomly selected password characters), by what factor does this change increase the amount of time the attacker must spend to guarantee finding a given account's password? For example, if this technique means that instead of taking 4 days, it will take 10 days, then that would be a factor of $10/4 = 2.5$.

(d) Suppose instead of using either of the approaches in (b) or (c) above, the site instead uses a different solution called *port knocking* to ward off the attackers.

This works as follows: even if a service is listening on a particular port, it looks closed to the outside world. Anyone who wants to communicate to the service has to 'knock' in order to open it. A "knock" is done by sending a set of UDP packets to a magic sequence of three ports. The server tracks UDP packets seen from each remote IP address R. Every time it sees a series of 3 such packets, the server checks to see whether they were sent to the magic sequence of ports (in the correct order).

For example, suppose the magic sequence was UDP ports 442, 16101, and 7777. Then if R sends UDP packets to ports 101, 442, 16101, 7777, 8989, 32123, the server will ignore R's attempt, since neither set of three ports it saw ($\langle 101, 442, 16101\rangle$ or $\langle 7777, 8989, 32123\rangle$) matches the magic sequence, even though the magic sequence was in fact embedded within the stream.

However, if a group of three UDP packets sent by R goes to exactly the magic sequence in order, then the server will now respond to the next incoming connection request that R makes to the SSH server on TCP port 22.

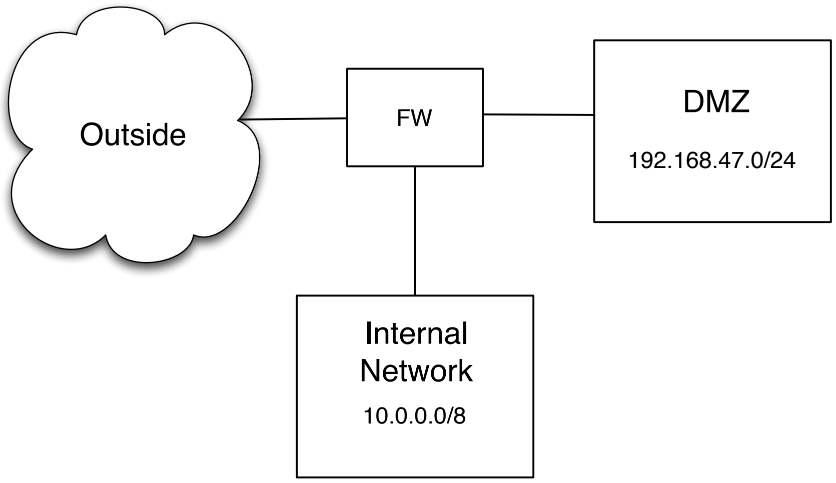
Assume that every second the attacker can try 10,000 port-knocking sequences (each of which consists of sending three UDP packets plus a TCP connection attempt to see whether the knock worked). For the situation as at the start of (b) above (6 character passwords that are present in the attacker's 2,000,000 entry dictionary), by approximately what factor does the port-knocking approach increase the amount of time the attacker must spend to guarantee finding a given account's password?

(e) Suppose that in addition to using the port-knocking approach, the site will *block* a remote IP address R if the server observes ten unsuccessful port-knocking sequences from R. Any subsequent traffic from R will be discarded, even if it later reflects a correct port-knocking sequence.

How feasible is it for an attacker to muster enough bots (each with its own IP address) to overcome this blocking?

What new vulnerability does using this blocking approach raise for the site?

Problem 2 *Firewalls* (25 points)



The diagram above shows the network used by a company. Like many enterprises, the main internal network is isolated from external networks using a *demilitarized zone*, or **DMZ**. (You can read about DMZs at http://en.wikipedia.org/wiki/DMZ_(computing).) You have been tasked with securing this network by ensuring that all of its network activity conforms to the following policy for TCP traffic:

- Unless otherwise specified, all traffic should be denied.
- All inbound mail from the Internet must be delivered to TCP port 25 of 192.168.47.12.
- 192.168.47.12 in turn delivers mail via TCP to the inside mail server, 10.0.0.17.
- SSH logins from the Internet are permitted to TCP port 22 of 10.0.37.47.
- The web server sits in the DMZ, on TCP ports 80 and 8080 of 192.168.47.17; both insiders and outsiders should be able to access it.
- All outbound connections are allowed, except to sites on 177.16.0.0/16 or if the connection is coming from 10.47.0.0/16.
- Hosts on 10.47.0.0/16 are only allowed to connect to hosts on 177.25.33.0/24.

For this problem, we consider three separate sets of firewall rules. As indicated in the diagram, there are three interfaces on the firewall: *inside* ("Internal Network"), *outside*, and *DMZ*. Rules always apply to packets **arriving** at an interface. The inside net is 10.0.0.0/8; the DMZ net is 192.168.47.0/24; everything else is on the outside. For example, rules associated with the *DMZ* interface apply to packets sent from the 192.168.47.0/24 network.

In your solution, list a separate ruleset for each interface. Use a format similar to that in lecture, but annotated with the interface. So, for example:

```
inside allow tcp 10.0.5.5:* -> 188.4.0.0/16:53 if ACK set
```

specifies that on the firewall's *inside* interface, TCP packets arriving from the internal network with a source address of `10.0.5.5` and any source port, destined for port 53 of any address in the /16 block `188.4.0.0`, should be allowed providing they have the ACK bit set.

You should aim to keep the rules as simple (minimal) as possible. Keep in mind the order in which rules are processed. A reminder: you only need to consider TCP traffic. You can leave "`tcp`" out of your rules if you wish.

Problem 3 *SQL Injection* (25 points)

Congratulations! You've just been hired as a security consultant for *BearBucks*, a bank catering to Cal students. They happen to be a Java shop (and we don't mean coffee!), and you've been tasked with reviewing the codebase for potential vulnerabilities.

You're soon dismayed to discover the following code in the client login section of the online banking site:

```
/**
 * Check whether a username and password combination is valid.
 */
ResultSet checkPassword(Connection conn, String username, String password)
        throws SQLException {
    String query = "SELECT user_id FROM Customers WHERE username = '"
        + username + "' AND password = SHA1('" + password + "');";
    Statement s = conn.createStatement();
    return s.executeQuery(query);
}
```

Here `SHA1` is a special type of hash function (which we'll learn more about later in the course). For the purposes of this problem, you can treat it as doing a deterministic scrambling of `password` into the format in which passwords are stored inside the database. The particulars of this behavior are not important for the problem.

(a) What usernames could an attacker enter in order to do the following?

- Delete the `Customers` table.
- Issue a request as user Admin, without having to know the password.

(b) When you point this out to the development team, a junior developer suggests simply escaping all the single quotes with a backslash. For example, the following line could be added to the top of the function:

```
username = username.replaceAll("'","\\\\'");
```

Modify either of the injection attack inputs you listed in your answer above to still work in this scenario.

(c) Rewrite the `checkPassword` function using `prepared` statements. Explain why your code is safe from SQL injection attacks.

See http://bit.ly/preparedstatement for a discussion of Java `prepared` statements.

(d) The developers are now busy refactoring their code to use `prepared` statements. One of them approaches you with a dilemma: she is having difficulty adapting the function below.

```
/**
 * Filter transactions based on a dollar amount specified by the user and
 * sort based on user-supplied values.
 *
 * @param conn Database connection
 * @param amt  Filter based on this amount and the comparison operator.
 * @param cmp  Comparison operator by which to filter (>,>=,=,<=,<).
 * @param orderByCol Name of column by which to sort results
 *                   ("amount", "date" or "type").
 * @param orderByDir Sorting direction ("ASC" or "DESC").
 */
ResultSet searchTransactions(Connection conn, BigDecimal amt, String cmp,
        String orderByCol, String orderByDir) throws Exception {
    String q = "SELECT * FROM Transactions WHERE ";
    q += " amount " + cmp + " " + amt;
    q += " ORDER BY " + orderByCol + " " + orderByDir + ";"
    return conn.createStatement().executeQuery(q);
}
```

For example, this would allow queries like the following to be run:

```
SELECT * FROM Transactions WHERE amount > 13.37 ORDER BY date DESC;
SELECT * FROM Transactions WHERE amount <= 42.00 ORDER BY amount ASC;
```

Why is the developer having trouble? What limitation in the `prepared` statement API is causing this issue? Why does the API impose this limitation?

(e) Rewrite the `searchTransactions` function using `prepared` statements. It's okay to make modifications to the code as long as queries like the examples above can still be run.

Problem 4 *Web Vulnerabilities* (25 points)

After having helped *BearBucks* straighten out their web server security, you get a consulting gig examining the code of *Farcebook*, the latest up-and-coming ironic social network. You learn that the developers there have heard troubling rumors that their code contains XSS vulnerabilities.

Farcebook users can use HTML on their profile pages and in private messages. The developers are already doing some sanitization: servers strip the literal strings `<script>` and `</script>` from any user input.

(a) You immediately realize that this is not enough to keep potentially malicious Javascript from being executed on the page. To demonstrate, you craft a message that, upon being read by the recipient, grabs the recipient's cookies (including the session cookie used to keep the user logged in to the site) and passes them as a query string parameter to a special logging script that you have set up at the following address:

`http://xss.cs161.com/log?data=COOKIE_DATA_STRING`

Show an example of such a message.

(b) Is this a case of a reflected or persistent XSS vulnerability? Briefly justify your answer and explain the difference between the two.

(c) Read about `HttpOnly` cookies here: http://bit.ly/nE1NPv

How would using this feature prevent the attack above?

Sketch an XSS attack to which users would still be vulnerable.

(d) You soon spot another problem. As in most social networks, when you are logged in to *Farcebook* you can send a friend request to another user. To do so, you visit a *Farcebook* page that has a form on it. When you enter the user's name into the form and click on *Submit*, your browser transmits a URL that looks like:

`http://www.farcebook.com/send_friend_request?recipient=USER`

where `USER` is the username of the user you want to friend.

Assuming that the XSS vulnerabilities have been fixed, how might Louisa (a lonely *Farcebook* user) still use this to artificially increase her friend count, such that any time someone visits Louisa's profile, the visitor automatically sends Louisa a friend request?

(e) What can *Farcebook* do to address this loophole, so that a server will only honor friend requests that users indeed submitted by deciding to visit the site's standard form page?