3.5.1 Indirect access to data

The bank maintains a huge number of accounts for a large number of users. In the example above, the bank looks up all incoming account requests in this large pool of accounts. If the programmers forget about authorization tests somewhere, a user that is able dictate the passed number will have access to all those accounts, not only his own.

One possible solution is to forget about the ''global'' pool containing all possible accounts when doing account look-ups, and instead introduce *user-local* pools. The session object may help us implement user-local pools. As

soon as the user logs in, the bank queries the global account pool for all accounts available to this user, and stores the resulting list as an array in the user's session. Every incoming request for a user-controlled account is passed through the session-local list rather than going directly to the global pool. With this scheme, no matter how a malicious user modifies the input, he will not be able to look up any accounts other than his own.

Given this session-local array, any server-generated references to the user's account in web pages would be encoded as indexes into the array rather than as real account numbers. The `option` tags from the previous form would thus look like this:

```
<option value="1">1234.56.78901</option>
<option value="2">1234.65.43210</option>
```

Note how the `value` attributes now contain `1` and `2` rather than account numbers. The real account numbers will be looked up in the session-local array of accounts this user has access to.

The worst thing that could happen if an attacker modifies this array index and the programmer forgets to check its validity, is a run-time error or look-up of an undefined object. It would no longer be possible to have access to other accounts. (If you program C-like languages, or any other language in which index checking isn't performed, an index overflow may actually be quite serious. Most modern web languages include automatic index checking.)

When we accept indexes or other labels as incoming data and map them to "real" data on the server, we use a principle I like to call *data indirection*. With data indirection, we do not use incoming data as the target data, but rather use them to look up the target data somewhere on the server. In such cases we often do not need to check the validity of the incoming data: We look them up, and if no match is found among the real data, the incoming value was invalid. Note that we must be suspicious about the index or label during the look-up process: If the look-up process involves a subsystem, such as a database, we may need to pay attention to metacharacters. Data indirection won't solve all problems.

3.5.4 Authorization by obscurity

Normally, we need some identification of a user to decide whether he is authorized to perform an operation. The identity is verified using authentication, which often involves a user name and a password, a digital certificate, or a code calculator. Some developers seem to think that the "authenticated user with a certain role" scheme takes too much work to implement, so they stick to "whoever knows our little secret will be let through", as seen in the following example.

RIAA, the Recording Industry Association of America, is famous for doing everything possible to stop people from copying digital entertainment. In some circles, they're also famous for being *defaced* once a month. Defacing a web site refers to altering or replacing the content. Often the attackers just leave messages like "Your security sux", "Cracked by 31337 hax0rs" (elite hackers—those script kiddies can't even spell correctly), "Save the rain forest", "Israel rules", "Palestine rules" and variations thereof. In the RIAA case, however, the attackers seem to prefer making it look as if the piracy-fighting association offers pirated MP3 files from their web site.

Details of one of the defacements were given in an article [77] in *The Register* [78]. The attackers had got access to the administrative interface of RIAA's web. Through that interface, they could modify the pages. How did they get access? Apparently, they started by requesting a file called `robots.txt`, which many web servers keep in their web root. The file is intended to be read by search engines, and it contains rules that tell the search engines what they're allowed to index, and what they should skip [79]. The file is not a security mechanism, it just sort of says "please" to well-behaving search engines. RIAA's `robots.txt` looks like this:

```
User-agent: *
Disallow: /temp/
Disallow: /admin/
Disallow: /cgi-bin/
Disallow: /Archive/
```

Among other things, this file informs the attackers that there is a directory named `/admin/` on this web server. The attacker simply asked his browser to pay a visit to this `http://www.riaa.org/admin/` thing. Lo and behold, there was the administrator interface. It didn't even require a password.

The RIAA guys probably thought that as there was no link to the admin interface from the web pages, nobody would find it. This way of hiding things to implement "security" is called *security through obscurity*. As this case showed, security through obscurity generally doesn't work. Even if there hadn't been any `robots.txt` available, some people could have made a guess about `/admin/`. There are even programs that automatically search for obvious, hidden documents, of which `/admin/` is a clear candidate. Assuming that "only authorized people will know about this" is a dangerous upside down view of authorization.