Ensuring Security within the Offsider framework

First up, a disclaimer. I am not an expert on security. Few of us really are. For this reason, you should take **everything** in this rant with a grain of salt. If security is really important in the system you are developing, consult a qualified security expert. I suggest you also get a second opinion. Then a third.

There is no formula

Every system is a special case, there is no general-purpose solution. This is for certain.

When you are specifying the functional requirements of your offsider, you need to think about its purpose, its usage and the security ramifications involved. You need to think about the types of security you require (is it data integrity, data theft, system up-time, or something else?) What are the consequences of a security breakdown?

When you are designing the implementation, you need to think about the security ramifications of that design.

When you have built your offsider, and are testing it, you need to do a complete audit of its components to ensure that the security you planned has actually been delivered.

The care with which you carry out these steps should be commensurate with the potential damage that would come from a security breach.

This needs to be done afresh for each and every offsider you build. *Zen mind, beginner's mind.*

The Offsider framework does not provide any security tools.

No Security API

There are no default methods or tools to help you with security. This is intentional. I am not a security expert, and it is not appropriate for me to deliver some half-baked, badly thought-out "solution". I don't want anyone to be lulled into a false sense of "security" just because I have provided a no-op method and called it `lockItUpTight`.

`UNIX` provides security

In `UNIX` (and by analogy, in `GNU/Linux`), you have mechanisms that are there to provide security of one form or another. These are:

users, groups and file permissions. You can put thought into setting up appropriate users and groups to fit the different roles that your offsider requires. You then set up file permissions within your offsider infrastructure to ensure that the wrong person cannot do damage.

firewalls. Firewalls are useful for limiting access across a network.

login profiles. For example, many systems don't allow users `www` or `nobody` to login.

strong passwords. Use passwords that are resistant to cracking, and don't leave those passwords lying around.

encryption. Encrypt data where neccessary. Use `ssh` or `https` across a network.

and don't forget:

physical security. Locks and keys. Access to rooms and buildings. Physical access to network infrastructure. Access to removable media. Even internal drives can be stolen if a thief has access, and time to do the deed.

the ones I don't know about and **the ones that I've forgotten about**. Again, consult an expert.

You are on your own

To summarise, the offsider framework doesn't provide security mechanisms, but `unix` does. It is up to you to figure out how to do it.

The rest of this document will provide some information and some hints to help you figure it out.

`Unix` file permissions

Most of my discussion from here on in will center around setting up `unix` file permissions, so a brief summary is in order.

Every file (and every directory) has an *owner*, and also a *group* associated with it. It also has three sets of file permissions. One each for the owner, the group and for everyone else. If your user name matches the file's owner, then the system will use the owner file permissions, if you belong to the file's group, then the system will use the group file permissions, otherwise it will use the *other* file permissions.

File permissions available are *read*, *write* and *execute*.

These three types of permission, together with the concepts of owner, group and other, give fine enough granularity to provide fairly flexible and robust security - **especially** when using offsiders!. It is difficult to get better security without going to a capability-based system (not supported by `unix`) implemented within a continuous and persistent memory-image.

For a directory:

read permission means you can access a file in the directory, provided you can provide its name.

write permission means you can create or delete files from the directory.

execute permission means you can get a complete list of all the names in the directory. (ie, the names of all the files and directories stored therein).

For a normal file:

read permission means you can read the contents of the file.

write permission means you can modify the contents of the file.

execute permission means the file is executable, and you have permission to run it as a process.

The offsider infrastructure

I have written many rants about offsiders, and this is the first one where I have gone into the details about the infrastructure. For normal application programming, the object-oriented principle of information hiding implies that you should not need to know anything about the internal details of the object. In offsiders, you don't.

However, to properly set up your offsider from security, you will need to think carefully about the ownership, and file permissions of individual directories and files within the infrustructure. For this you will need to understand how the infrustructure is set up and how it is all used.

An instructive example

Before we describe the **default** case, let's look at an unusual example. I do this to jolt you out of a sense of complacency.

First, create an empty directory.

```
mkdir ./emptyDirectory
```

Now, create a named executable pointing to this directory.

```
offsider -b ./emptyDirectory > ./null
chmod +x ./null
```

Believe it or not, you have just created an offsider. The directory is still completely empty, as you can easily verify:

```
ls -a ./emptyDirectory
```

On the other hand, it acts just like a real offsider:

```
./null allMethods
```

.. that's a lot of methods for a completely empty structure! Play around with this offsider, trying out different methods to see what they do. When you are done, look inside the directory again. Is it still empty?

This example illustrates a few points:

- you can't assume that your offsider contains any particular files or directories of neccessity.

- you can specifically set it up to have any structure you please.

- the structure may change in ways you weren't expecting.

- the offsider can execute code that is external to its infrustructure.

You need to be aware of these points. If they are a problem for you, you need to specifically address those issues.

The default infrustructure.

An offsider's infrustructure means everything that is contained within its base directory.

By default, all offsiders are created with the following subdirectories:

keys

Contains the offsider's keys. Each key is stored as a single file, named as the name of the key. For example, the value of the key `foo` is the contents of the file `keys/foo`.

methods

Contains the offsider's own methods. Each method is stored as a single executable file, named as the name of the method. For example, the method `foo` is the executable `methods/foo`.

Note, that the offsider will also know about methods that are not its own.

var

Contains metadata for this offsider. Not actually used by the offsider for anything, but may well contain sensitive information about the user who created it, and the environment that was used to create it. The method `destroyCompletely` will look here to find out the named executable.

Other directories are created as required:

attachments

Contains the offsider's attachments. Each attachment is stored as a single file, named as the name of the attachment. For example, the value of the attachment `foo` is the contents of the file `attachments/foo`.

childDictionaries

Contains the offsider's child offsiders. Each child offsider is stored within a single directory, named as the name of the child offsider. For example, the base directory of the child offsider `foo` is `childDictionaries/foo`.

Security implications of the offsider framework

The first thing to keep in mind is that a potential intruder won't need to go through the offsider framework to access your offsider. They can just use raw `unix` tools to access the infrustructure directly. This means the contents of the offsider's base directory.

On the other hand, they need to know what the base directory is. Offsiders (especially personal ones) are typically named using `unique.timestamp`, which makes the base directory pretty much impossible to guess, even for a program with lots of spare time. On the other other hand, the base directory is usually fully visible within the named executable (which is normally a shell script). Also, if they can list your `~/.offsiders` directory, then they have all the names anyway.

The only way to lock everything up tight is to deny read access to everything: especially the base directory, but also the named executable, and `~/.offsiders` (or wherever you store your offsiders).

$PATH

The `$PATH` environment variable determines where executables can be found. If you get this wrong, you can end up running the wrong executable. (It will have the correct name, but it will be found in the wrong directory).

A similar thing applies with offsiders, but with added complexity. The offsider framework goes about its work by looking for executables of a particular name in a variety of places. This is normally tied very strongly to the `$PATH` environment variable. Because the logic of the `offsider` executable is strongly geared to late binding, and allows individual offsiders lots of chances to override every part of the process, there is lots of scope here for manipulation. I am not sure exactly what the attack vectors might look like.

Security implications of the offsider infrustructure.

Let's look in turn at each part of the infrustructure, and consider it's security implications.

The base directory

To do anything at all, you need to have access to the base directory. Therefor you need at least `read` permission to the base directory. Withdraw `read` access and you deny access completely.

`keys`

You need execute permission to the `keys` directory to know what keys are present. You need read permission to that directory to access individual keys by name, and write permission to remove or create keys. You need read permission to individual keys to get the values of those keys, and write permission to modify those values.

You can set both ownership and file permission individually on every separate key, which gives you a lot of very fine-grained control over access to keys.

`methods`

You need execute permission to the `methods` directory to know what methods are present. You need read permission to that directory to access individual keys by name, and write permission to remove or create methods. You need read permission to individual methods to look at the contents of those methods, and write permission to modify the methods. You need execute permission to run the method.

You can set both ownership and file permission individually on every separate method, which gives you a lot of very fine-grained control over access to methods.

`attachments`

Attachments are completely analogous to keys. Everything that was said for keys applies to attachments.

`childDictionaries`

You need execute permission to the `childDictionaries` directory to know what child offsiders are present. You need read permission to that directory to access individual child offsiders by name, and write permission to remove or create child offsiders.

Since each child offsider is itself a complete offsider, all of the information just given applies to that offsider, starting at its base directory.

You can set both ownership and file permission individually on every separate child offsider, which gives you a lot of very fine-grained control over access to each one.