Computer Security

Practical 2

Secure Web Programming

School of Informatics
University of Edinburgh
`http://www.inf.ed.ac.uk/teaching/courses/cs`

This is an **unassessed** practical. It is designed to promote understanding of some of the material in the Computer Security module. We suggest that you work through the questions and make a note of your answers. Bring your solutions to second Tutorial where we will review concepts from this practical, as well as your answers. Since work on these questions is not assessed, you may like to discuss and collaborate with your fellow students. you can check your answers.

Introduction

This practical exercise looks at some common security problems in writing web applications. We will start with a particular application scenario and consider the possible risks and attacks. Then we will take a particular naive implementation to examine and discover vulnerabilities, and then implement countermeasures which reduce the vulnerabilities. After completing this practical you should know how to:

- conduct an informal security analysis of a small system;
- identify, explain and fix several kinds of vulnerabilities in web applications.

Throughout the coursework you might encounter concepts that are unfamiliar to you. It is part of the exercise that you pragmatically seek out the information that you will need to solve the questions. Here are some pointers:

- *Open Web Application Security Project* at `http://www.owasp.org/`. The OWASP project produces an extensive guide to securing applications.
- The application considered here uses the scripting language Ruby. Documentation is available at `http://www.ruby-lang.org/` and `http://www.ruby-doc.org/docs/ProgrammingRuby/`.
- A little knowledge on web internals may come in useful. See e.g., *HTTP: The Definitive Guide* by David Gourley and Brian Totty, O'Reilly 2002 (available in the library).
- You will need to read relevant man pages and the application source code. Google and Wikipedia may occasionally be helpful.

The web application we will consider is a simple online photo sharing site. To "sandbox" the experiments, we will use a *User-Mode Linux* (UML) machine. This provides a safer environment in which to explore the damage that flawed web applications can do, without directly risking your existing web pages or DICE account. The User-Mode Linux and sample application are described in the section beginning on page 6. Before looking at that, we will introduce the application scenario and consider possible security risks.

Photo-sharing for a camera club

Edinburgh Shutterbugs is a small photography club with a few dozen members. They have the idea to create a web site so that they can share their pictures with one another and discuss their work. Most of the members are not computer savvy. One member who is, Alice, says that she would like to do this as an exercise in writing a web application in a new scripting language. The application will be mounted on a web server with good bandwidth and lots of disk space which is accessible from the Internet; the ISP bills for hosting will be paid from the club subscriptions.

The basic photo-sharing application should have the following features:

- Any Internet user should be able to create an account, to become a user of the website.
- A website user should be able to upload photographs to the album using a simple mechanism, such as HTTP POST.
- Each photograph can have a description attached, editable only by the creator of the album.
- The album should allow resizing of photos online.

In the future, the application should be extended with some further features:

- Users should be restricted to members of the photography club.
- Users should be allowed to remove photos from their own albums.
- Each user should be able to group photographs into one or more albums.
- There should be distinction between public and private albums. Public albums could be discovered trivially by any user of the Internet, e.g., from a list on the website. Private albums (for which other people have to be given access, either by a "secret" URL or some account based access control) should not be easily discoverable by arbitrary Internet users.
- Someone who has access to a private album should not easily be able to discover the names of other albums or photos, so that users can selectively share photographs with other people.

So far, Alice has developed a sample application which meets the initial set of requirements; this will be the model web application we will study later.

Part A: Threats and risks

Before looking at the technical details of Alice's implementation, we will undertake a simple security analysis of the situation. There are five stages:

1. Identify the **assets**, along with their **owners**;
2. Survey possible **vulnerabilities** which might be exploited to damage the assets;
3. Consider specific **threats**: the actions and perpetrators that may exploit vulnerabilities;
4. Assign **costs** and **probabilities** to the threats, to assess the **risks**;
5. Consider **countermeasures** for reducing the risks.

For the photo sharing scenario, some example assets and their owners include:

- the *image data*, owned by the *website users*;
- the *website service*, owned by *Alice*;
- the *ISP web host machine* and *Internet connection*, owned by the *ISP*.

Owners are parties who are accountable for the assets concerned and wish to protect them; however, they may be in trust relationships with other stakeholders who are expected to implement appropriate countermeasures.

For this exercise we will treat the analysis informally, considering some threats on a case-by-case basis. For each threat, we shall say what vulnerabilities it might target (without considering a specific implementation yet), and qualitatively, what the potential loss is, and the probability of attack. Suitable countermeasures should be suggested when feasible. Here is an example threat:

Photos are deleted by a non-owner. A malicious individual gains access to the web site and deletes a photo owned by one of the legitimate users. This may be achieved by spoofing, breaking the authentication system in some way. Considering the number of ways that authentication schemes

(especially simple ones) may be broken, this attack has *high probability*. However, although image data is highly valued by owners, the *risk is low* because users should keep backups of photos that they upload, and warned that data loss is not guaranteed against. Feasible countermeasures include *improving authentication*, especially preventing programming flaws which might lead to spoofing, along with keeping *image data backups* and *access logs* to record the upload and edit history.

Exercise. Following this example, consider a further **four threats** which are possible in the scenario, summarising in each case the vulnerabilities, the cost caused by damage to or recovery of the asset, and a suggestion for the probability and hence risk of the threat. Quantify the probability and risk each as either *high* or *low*. Give a suggestion of possible countermeasure strategies or say why they are not feasible. Restrict yourself to threats due to malicious, online behaviour of others (rather than, e.g., "natural" causes such as power outages or social engineering attacks).

Fill in your answers in the provided template `PartA.txt` and bring your answers to the corresponding Tutorial.

To complete this part, you may find it useful to study the Code of Practice for Information Security Management, Part 1 of BS ISO/IEC 17799:2000. A copy is available from the course web page, at:

`http://www.inf.ed.ac.uk/teaching/courses/cs/docs/BS7799/BS7799-1:2002.pdf`

It is also linked from the Resources page. More specific information useful throughout the practical appears at the *Open Web Application Security Project* at `http://www.owasp.org/`.

Part B: Exploring vulnerabilities

In this part of the practical we will explore some specific vulnerabilities present in the provided implementation of the photo web site. You are asked to explain and discover some of the problems; in Part C we will look at fixing some of them.

Fill in your answers to the questions in this part in the provided template `PartB.txt` and bring to the corresponding Tutorial.

At this stage it would be a good idea to try out the photo album web application. To do this, start up and investigate the UML machine as described on page 6.

B.1 Information discovery

An attacker wanting to discover details about a web site's implementation may start by examining the HTML that the site generates. Try loading up the example photo album at:

`http://localhost:8000/csphotos/album?user=john`

and choose "View Page Source" in your web browser. Observe that all of the images in the photo album (including both the thumbnails and the full size images) are returned by the script `image` — the URLs all begin with `http://localhost:8000/csphotos/image?`. Now try the following URL, copying it carefully:

`http://localhost:8000/csphotos/image?name=../../../../../../../../etc/passwd`

Exercise. You will find that this allows you to save a copy of the UML machine's password file. Examining the code, explain how is this allowed to happen. What kinds of information might an attacker usefully discover with this attack and similar variants?

B.2 Denial of service

Again by passing an unexpected URL, the `image` script file allows a simple denial-of-service attack, which can exhaust disk space and saturate the CPU of the web host.

Exercise. Give an example of a URL that will cause a denial of service attack of this kind. If your attack does not work straight away, you will have to experiment a little. Why is that?
Explain two independent ways in which the attack might be prevented.

Note: to see what is happening on the UML machine, you should log in to it with `slogin` before trying out the attack; you can use `top` to view the running processes. Once the DoS is underway, the computer you are using will most likely respond very sluggishly; you should still be able to locate the runaway process on the UML machine and kill it. You may also need to delete leftover temporary files in `/tmp` to recover the disk space.

B.3 SQL injection attacks

The web site uses an SQL database to store metadata for all of the photos. Unfortunately the script is not adequately protected against *injection attacks* on SQL queries.

Exercise. Create and try out an SQL injection attack on the application. It helps to look at the script files although an attacker might not be able to do this. If you do not already understand SQL injection you will need to find out about it first. Copy the attack into your answer file, and explain what you expect it to do.

B.4 Arbitrary command execution

While the attacks considered above are serious, they do not go so far as to allow the execution of arbitrary code on the web server.

Exercise. Describe the possible scope of the damage that could be caused by a vulnerability that allowed arbitrary commands to be run on the server. Can you see any way in which this application is vulnerable? Give an example if so.

Part C: Fixing some flaws

For this part, you are asked to investigate some repaired versions of the flawed scripts.

C.1 Input validation

Exercise. Fix the scripts so that at least the vulnerabilities B.1–B.3, are addressed. Specifically:

1. To partially address the problem in B.1, you should repair the function `valid_photo_name?` in the common file `common.rb`.[1]

   The `image` script should only return images in photo albums, and no other files on the filesystem. This will require editing another file.

2. Create a simple fix for the denial of service vulnerability discussed in Question B.2. Your fix need not deal with other DoS attacks (such as disk space exhaustion by uploading images).

3. Fix the SQL injection vulnerability you demonstrated in your attack in Question B.3.

After completing the fixes, very briefly describe what you have done in `PartC.txt`.

[1] This file mentions two URLs from the "Programming Ruby" online book which have since moved. The current locations are: `http://www.rubycentral.org/pickaxe/tut_stdtypes.html#S4` (for regular expressions) and `http://www.rubycentral.org/pickaxe/ref_c_file.html#File.new` (for documentation about file modes). These are also available from the documentation at `www.ruby-doc.org` mentioned on page 1.

C.2 Other countermeasures

The minimal fixes asked for so far in this exercise are essentially preventative measures. However, the application contains further security holes that we haven't discussed and in general we can't hope to achieve perfect prevention. Therefore it is important when designing robust web applications to provide mechanisms to help detect and recover from security breaches.

A simple first step is to provide *application level logging*. You should consider the different types of events that it might be useful to record, based on possible threats. For instance, it turns out that the generated identifier used in the session cookie is easy to guess. Consider what type of logging might expose attempts to exploit that, supposing the developer hasn't spotted the flaw. For other examples, consider your threat analysis from Part A.

Exercise. Add a useful log to the application, considering the information and events it is useful to record. For each place where you add a logging statement, also insert a short comment into the code to explain what you are logging and why. There is a skeleton `log_message` function within `common.rb` that you might want to use. Indicate the files you have changed, making sure they still contain any fixes from question C.1, and briefly describe them in `PartC.txt`.

Part D: Going further

The sample script, even after the fixes suggested above, is far from an example of a robust web application. To explore the subject of web application security further, you may like to:

- Discover other flaws and construct exploits, e.g., for cross-site scripting or session-hijacking;
- Implement some of the secondary features in the specification in a secure way.

The User-Mode Linux virtual machine and photo album

User-Mode Linux (UML) is a way of running Linux as a virtual machine on an existing Linux installation; it runs as a normal process executable by non-root users. For this practical you are **highly recommended** to work on the X Console of a DICE desktop machine, i.e., *not* by remote login to a compute server or a desktop machine that may be in use by someone else. The UML setup assumes that you have unique access to the machine you are using. Moreover, we will execute scripts that consume large resources which may inconvenience other users on the same machine.

We refer to the DICE machine you have logged into to run UML as the *host machine* and the virtual UML machine as the *UML machine*. You should start the UML machine by running the provided `startuml` script like this:

```
/group/teaching/cs/startuml
```

The first time you successfully run this script, it will create a copy-on-write (COW) file called `/tmp/cow-file-sXXXXXXX` (where `sXXXXXXX` is your DICE username.) This file contains the modifications you have made to the initial root filesystem. Alternatively, you could provide your own name for the COW file as a single parameter to that script.

The copy-on-write file is created as a *sparse* file, so although it will appear to be very large in a normal file listing, in fact it only uses a much smaller number of blocks. You can see the space actually used with the `-s` parameter to `ls`. For example:

```
[cuttysark]sXXXXXXX: ls -lhs /tmp/cow-file-sXXXXXXX
22M -rw-------  1 sXXXXXXX people 1.8G Jan 25 08:17 /tmp/cow-file-sXXXXXXX
```

The first column shows the space taken by the allocated blocks and will be the amount that is deducted from your DICE quota, for instance.

Due to problems with locking the COW file over NFS, you should only use UML with COW files stored on local disk space (e.g. in `/tmp`). Once the UML process has cleanly exited you can move the file back to your DICE home directory, to reuse it later or work on a different machine. To ensure that the COW file is preserved as a sparse file when copying it, you must use the `--sparse=always` parameter to `cp`. The `mv` command can handle sparse files and does not require this parameter.

Important Note: During the practical it is very likely that the COW file will become corrupted or too big to easily store in your DICE account (e.g., after running denial of service attacks, or when the machine crashes or gets killed unexpectedly). For this reason, you should edit a master copy of the web application, along with any configuration or log files you want to keep, in your DICE account. Then you can remove the corrupted COW file and copy things back onto the virtual machine restarted with a fresh COW. See "Editing files on the UML machine" below.

On running the `startuml` script you should see a familiar Linux boot-up sequence. Once the UML machine has booted, you can log into it from a second terminal window with:

```
slogin -p 2200 -X alice@localhost
```

The UML machine has a `root` account with **password** `fr0st1es` (that's with a zero and a one) and a non-root user called `alice` with **password** `alicetoo`. Of course these are both extremely poor choices of password. You **must** change them with `passwd` once you've logged in to prevent other people easily logging in to your UML machine.

Once you've finished using the UML machine, shut it down by executing `shutdown -h now` as root. You should always shut down the UML machine in this way, or you may leave the filesystem in an inconsistent and possibly unrecoverable state. If something goes wrong you may need to kill the UML process. In this case, you can use the command:

```
/group/teaching/cs/killuml
```

which will try to kill the UML machine as gracefully as possible — but watch out for corruption of the COW file if you had to do this. When you have finished your session on a particular DICE machine, move your COW file to your DICE home directory since `/tmp` is cleaned periodically.

Editing files on the UML machine

When you are logged into the UML machine with `slogin` as above, the `ssh` program should forward the X protocol to the host machine's X server. To edit files you can use any of the X-enabled or console editors installed on the machine (e.g., Emacs, XEmacs or VIM).

For better safety, however, you should develop your solutions to this practical exercise in your DICE account. That way, if the UML machine crashes, you will not lose any work. To copy files *from* the UML machine into your DICE account, use a command like this on the *host* machine:

```
[cuttysark]sXXXXXX: scp -P 2200 -pr alice@localhost:/home/alice/csphotos/ .
```

And to copy the other way round (i.e. from the DICE machine *to* the UML machine) use a command like this on the host machine:

```
[cuttysark]sXXXXXX: scp -P 2200 -pr csphotos alice@localhost:/home/alice/
```

(you can also use `scp` on the UML machine, but that would be rather trusting given its name...).

The Photo Album web application

The demonstration web application is written in the scripting language Ruby. A full understanding of Ruby is *not* required to complete the practical: you will find some helpful hints in comments in the code which explain what it does. Further information is available at `http://www.ruby-lang.org/` (which has a link to a useful draft book "Programming Ruby").

To use the photo album application, launch a web browser on the host machine and check that these URLs work:

```
http://localhost:8000/csphotos/home
http://localhost:8000/csphotos/album?user=john
```

This first is a page where you can create a new account for uploading photos, the second is an example album; the user `john` has password `hello`. To understand the application's behaviour you should explore it: e.g., try navigating to one of the photos' pages, choosing an alternative size, logging in, editing a photo description, and logging out. **Important Note:** when testing fixes for the application (and as usual for developing web applications), please be sure to restart your web browser, or disable the browser cache.

The source code for the photo album application is contained in the directory `~alice/csphotos` on the UML machine. Examine the files in this directory, and note that each of them loads commonly used code from the file `common.rb` in that same directory. All of these scripts run as the `www-data` user. The actual image files are stored under the directory `/home/alice/csphotos-data/` with one subdirectory per user. Within each of those directories the images are stored in files named after their MD5sums, as a simple way of generating a unique filename.

The application uses a PostgreSQL database. To interactively query this database on the UML machine, you can switch to the `www-data` user from `root` with `su`, and run `psql csphotos`. For example:

```
insecure:/home/alice# su www-data
insecure:/home/alice$ psql csphotos
Welcome to psql 7.4.9, the PostgreSQL interactive terminal.
...
csphotos=> select * from photos;
 username |              md5sum              |                  description
----------+----------------------------------+-----------------------------------------------
 john     | e0b72f79de809be23cd659d3122086e7 | Minature figures in a ‘‘hospital’’
 john     | 5b30b55ca12160fedd8e17b24aff7789 | The bar in the Mole Antonelliana, Torino
 .....
(8 rows)
csphotos=> \q
insecure:/home/alice$
```

Note that there is no direct login for the `www-data` user.

David Aspinall and Mark Longair. Updated by David Aspinall, Gavin Keighren and Mike Just
2nd March 2010
2009/01/23 16:38:29.