6.858 Spring 2017 Lab 4: Browser security

Handed out: Wednesday, April 12, 2017
Parts 1 and 2 Due: Friday, April 21, 2017 (5:00pm)
All Parts Due: Friday, April 28, 2017 (5:00pm)

Introduction

This lab will introduce you to browser-based attacks, as well as to how one might go about preventing them. The lab has several parts:

- Part 1: cross-site scripting attack
- Part 2: cross-site request forgery
- Part 3: side channel and phishing attack
- Part 4: a profile worm

Each part has several exercises that help you build up an attack. All attacks will involve exploiting weaknesses in the zoobar site, but these are representative of weaknesses found in real web sites.

Network setup

For this lab, you will be crafting attacks in your web browser that exploit vulnerabilities in the zoobar web application. To ensure that your exploits work on our machines when we grade your lab, we need to agree on the URL that refers to the zoobar web site. For the purposes of this lab, your zoobar web site must be running on `http://localhost:8080/`. If you have been using your VM's IP address, such as `http://192.168.177.128:8080/`, it will not work in this lab.

If you are using KVM or VirtualBox, the instructions we provided in lab 1 already ensure that port 8080 on `localhost` is forwarded to port 8080 in the virtual machine. If you are using VMware, we will use `ssh`'s port forwarding feature to expose your VM's port 8080 as `http://localhost:8080/`. First find your VM IP address. You can do this by going to your VM and typing `ifconfig`. (This is the same IP address you have been using for past labs.) Then configure SSH port forwarding as follows (which depends on your SSH client):

- For Mac and Linux users: open a terminal *on your machine* (not in your VM) and run

```
$ ssh -L localhost:8080:localhost:8080 httpd@VM-IP-ADDRESS
httpd@VM-IP-ADDRESS's password: 6858
```

- For Windows users, this should be an option in your SSH client. In PuTTY, follow these instructions. Use `8080` for the source port and `localhost:8080` for the remote port.

The forward will remain in effect as long as the SSH connection is open.

Setting up the web server

Before you begin working on these exercises, please use Git to commit your Lab 3 solutions, fetch the latest version of the course repository, and then create a local branch called `lab4` based on our lab4 branch, `origin/lab4`. Do *not* merge your lab 2 and 3 solutions into lab 4. Here are the shell commands:

```
httpd@vm-6858:~$ cd lab
httpd@vm-6858:~/lab$ git commit -am 'my solution to lab3'
[lab3 c54dd4d] my solution to lab3
 1 files changed, 1 insertions(+), 0 deletions(-)
httpd@vm-6858:~/lab$ git pull
Already up-to-date.
httpd@vm-6858:~/lab$ git checkout -b lab4 origin/lab4
Branch lab4 set up to track remote branch lab4 from origin.
Switched to a new branch 'lab4'
httpd@vm-6858:~/lab$ make
...
```

Note that lab 4's source code is based on the initial web server from lab 1. It does not include privilege separation or Python profiles.

Now you can start the `zookws` web server, as follows.

```
httpd@vm-6858:~/lab$ ./zookld
```

Open your browser and go to the URL `http://localhost:8080/`. You should see the `zoobar` web application. If you don't, go back and double-check your steps. If you cannot get the web server to work, get in touch with course staff before proceeding further.

Crafting attacks

You will craft a series of attacks against the `zoobar` web site you have been working on in previous labs. These attacks exploit vulnerabilities in the web application's design and implementation. Each attack presents a distinct scenario with unique goals and constraints, although in some cases you may be able to re-use parts of your code.

We will run your attacks after wiping clean the database of registered users (except the user named "attacker"), so do not assume the presence of any other users in your submitted attacks.

You can run our tests with `make check`; this will execute your attacks against the server, and tell you whether your exploits are working correctly. As in previous labs, keep in mind that the checks performed by `make check` are not exhaustive, especially with respect to race conditions. You may wish to run the tests multiple times to convince yourself that your exploits are robust.

Exercises 5, 13, and 14, as well as the challenge exercise, require that the displayed site look a certain way. The `make check` script is not smart enough to compare how the site looks with and without your attack, so you will need to do that comparison yourself (and so will we, during grading). When `make check` runs, it generates reference images for what the attack

page is *supposed* to look like (`answer-XX.ref.png`) and what your attack page actually shows (`answer-XX.png`), and places them in the `lab4-tests/` directory. Make sure that your `answer-XX.png` screenshots look like the reference images in `answer-XX.ref.png`. To view these images from `lab4-tests/`, either copy them to your local machine, or run `python -m SimpleHTTPServer 8080` and view the images by visiting `http://localhost:8080/lab4-tests/`. *Note that SimpleHTTPServer caches responses*, so you should kill and restart it after a `make check` run.

We will grade your attacks with default settings using the current version of Mozilla Firefox on Ubuntu 12.04 (as installed on, e.g., the Athena workstations) browser at the time the project is due. We chose this browser for grading because it is widely available and can run on a variety of operating systems. There are subtle quirks in the way HTML and JavaScript are handled by different browsers, and some attacks that work or do not work in Internet Explorer or Chrome (for example) may not work in Firefox. We recommend that you develop and test your code on Firefox.

Part 1: a cross-site scripting (XSS) attack

The zoobar users page has a flaw that allows theft of a logged-in user's cookie from the user's browser, if an attacker can trick the user into clicking a specially-crafted URL constructed by the attacker. Your job is to construct such a URL. An attacker might e-mail the URL to the victim user, hoping the victim will click on it. A real attacker could use a stolen cookie to impersonate the victim.

You will develop the attack in several steps. To learn the necessary infrastructure for constructing the attacks, you first do a few exercises that familiarize yourself with Javascript, the DOM, etc.

Exercise 1: Print cookie.

Cookies are HTTP's main mechanism for tracking users across requests. If an attacker can get ahold of another user's cookie, they can completely impersonate that other user. For this exercise, your goal is simply to print the cookie of the currently logged-in user when they access the "Users" page.

1. Read about how cookies are accessed from Javascript.
2. Save a copy of `zoobar/templates/users.html` (you'll need to restore this original version later). Add a `<script>` tag to `users.html` that prints the logged-in user's cookie using `alert()`.

   Your script might not work immediately if you made a Javascript programming error. Fortunately, Firefox has fantastic debugging tools accessible from the Tools menu in the browser: the JavaScript console and the DOM inspector, both accessible from the Tools menu. The JavaScript console lets you see which exceptions are being thrown and why. The DOM Inspector lets you peek at the structure of the page and the properties and methods of each node it contains. You might also want to try Firebug.

3. Use the "Network" tab of the developer toolbar to inspect the requests going between your browser and the website. By clicking on one of the

requests, you can see what cookie your browser is sending, and compare it to what your script prints.
4. Put the contents of your script in a file named `answer-1.js`. Your file should only contain javascript (don't include `<script>` tags).

Exercise 2: Email cookie.

Modify your script so that it emails the user's cookie to the attacker using the email script. The attack should still be triggered when the user visist the "Users" page.

Please review the instructions at http://css.csail.mit.edu/6.858/2017/labs/sendmail.php and use that URL in your scripts to send emails. You may send as many emails as you like while working on the project, but please do not attack or abuse the email script. Note that the cookie has characters that likely need to be URL encoded. Take a look at `encodeURIComponent` and `decodeURIComponent`.

When you have a working script, put it in a file named `answer-2.js`. Again, your file should only contain javascript (don't include `<script>` tags).

Exercise 3: Remote execution.

For this exercise, your goal is to craft a URL that, when accessed, will cause the victim's browser to execute some JavaScript you as the attacker has supplied. In particular, for this exercise, we want you to create a URL that contains a piece of code in one of the query parameters, which, due to a bug in zoobar, the "Users" page sends back to the browser. The code will then be executed as JavaScript on the browser. This is known as "Reflected Cross-site Scripting", and it is a very common vulnerability on the Web today.

For this exercise, the JavaScript you inject should call `alert()` to display the victim's cookies. In subsequent exercises, you will make the attack do more nefarious things. Before you begin, you should restore the original version of `zoobar/templates/users.html`.

For this exercise, we place some restrictions on how you may develop your exploit. In particular:

- Your attack can not involve any changes to zoobar.
- Your attack can not rely on the presence of any zoobar account other than the victim's.
- Your solution must be a URL starting with http://localhost:8080/zoobar/index.cgi/users?.

When you are done, cut and paste your URL into the address bar of a logged in user, and it should print the victim's cookies (don't forget to start the zoobar

server: `./zookld`). Once it works, put your attack URL in a file named `answer-3.txt`. Your URL should be the only thing on the first line of the file.

Hint: You will need to find a cross-site scripting vulnerability on `/zoobar/index.cgi/users`, and then use it to inject Javascript code into the browser. What input parameters from the HTTP request does the resulting `/zoobar/index.cgi/users` page display? Which of them are not properly escaped?

Hint: Is this input parameter echo-ed (reflected) verbatim back to victim's browser? What could you put in the input parameter that will cause the victim's browser to execute the reflected input? Remember that the HTTP server performs URL decoding on your request before passing it on to zoobar; make sure that your attack code is URL-encoded (e.g. use `+` instead of space, and `%2b` instead of `+`). This URL encoding reference and this conversion tool may come in handy.

Hint: The browser may cache the results of loading your URL, so you want to make sure that the URL is always different while your developing the URL. You may want to put a random argument into your url: `&random=<some random number>`.

Exercise 4. Steal cookies.

Modify the URL so that it doesn't print the cookies but emails them to you. Put your attack URL in a file named `answer-4.txt`.

Hint: Incorporate your email script from exercise 2 into the URL.

Exercise 5: Hiding your tracks.

With the exploits you have developed thus far, the victim is likely to notice that you stole their cookies, or at least, that something weird is happening. For example, the Users page probably also printed an error message (e.g., "Cannot find that user").

For this exercise, you need to modify your URL to hide your tracks. Except for the browser address bar (which can be different), the grader should see a page that looks **exactly** the same as when the grader visits http://localhost:8080/zoobar/index.cgi/users. No changes to the site appearance or extraneous text should be visible. Avoiding the *red warning text* is an important part of this attack (it is ok if the page looks weird briefly before correcting itself). Your script should still send the user's cookie to the sendmail script.

When you are done, put your attack URL in a file named `answer-5.txt`.

Hint: You will probably want to use CSS to make your attacks invisible to the user. Familiarize yourself with basic expressions like `<style>.warning{display:none}</style>`, and feel free to use stealthy attributes like `display: none;` `visibility: hidden;` `height: 0;` `width: 0;`, and `position: absolute;` in the HTML of your attacks. Beware that

frames and images may behave strangely with `display: none`, so you might want to use `visibility: hidden` instead.

Part 2: a cross-site request forgery (CSRF) attack

In this part of the lab, you will construct an attack that transfers zoobars from a victim's account to the attacker's, when the victim's browser opens a malicious HTML document. Your HTML document will issue a CSRF attack by sending an invisible transfer request to the zoobar site; the browser will helpfully send along the victim's cookies, thereby making it seem to zoobar as if a legitimate transfer request was performed by the victim.

For this part of the lab, you should not exploit cross-site scripting vulnerabilities (where the server reflects back attack code), such as the one involved in part 1 above, or any of the logic bugs in `transfer.py` that you fixed in lab 3.

Exercise 6: Make a transfer zoobar form.

We will first write our own form to transfer zoobars to the "attacker" account. This form will be a replica of zoobar's transfer form, but tweaked so that submitting it will always transfer ten zoobars into the account of the user called "attacker". First, we need to do some setup:

1. Create an account on the zoobar site and click on transfer. View the source of this page (in Firefox, go to the Tools menu, then Developer, and click on "Page source"). Copy the form part of the page (the part enclosed in `<form>` tags) into `answer-6.html` on your machine. Alternatively, copy the form from `zoobar/templates/transfer.html`. Prefix the form's "action" attribute with `http://localhost:8080`.
2. Set up the destination account on the zoobar site: create an account called "attacker" on the zoobar site with any password, then log out of the attacker account, and log back into your own account.
3. Load `answer-6.html` into your browser using the "Open file" menu. After opening, the URL in the address bar will be something of the form `file://.../answer-6.html`. This form should now function identically to the legitimate Zoobar transfer form.

Now, tweak `answer-6.html` so that it transfers 10 zoobars to the "attacker" account when the user submits the form, without requiring them to fill anything out. You will have to modify the `<input>` fields with the necessary names and values.

Hint: You might find the `<base>` HTML element useful to avoid having to rewrite lots of URLs.

Exercise 7: Submit form on load.

For our attack to have a higher chance of succeeding, we want the CSRF attack to happen automatically; when the victim opens your HTML document, it should

programmatically submit the form, requiring no user interaction. Your goal for this exercise is to add some JavaScript to `answer-6.html` that autoamtically submits the form when the page is loaded.

Hint: `document.forms` gives you the forms in the current document, and the `submit()` method on a form allows you to submit that form from JavaScript. Submit your resulting HTML `answer-7.html`.

Exercise 8: Hiding your tracks.

In the wild, CSRF attacks are usually extremely stealthy. In particular, they take particular care to ensure that the victim cannot tell that something out-of-the-ordinary is happening. To add a similar feature to your attack, modify `answer-7.html` to redirect the browser to `http://css.csail.mit.edu/6.858/2017/` as soon as the transfer is complete (so fast the user might not notice). The location bar of the browser should not contain the zoobar server's name or address at any point. This requirement is important, and makes the attack more challenging. Submit your HTML in a file named `answer-8.html`, and explain why this attack works in comments inside your HTML file (using `<!--` and `-->`. In particular, make sure you explain why the Same-Origin Policy does not prevent this attack.

Note: Be sure that you do **not** load the `answer-8.html` file from `http://localhost:8080/...`, because that would place it in the same origin as the site being attacked, and therefore defeat the point of this exercise. When loading the form, you should be using a URL that starts with `file:///`.

Hint: You might find the combination of `<iframe>` tags and the `target` form attribute useful in making your attack contained in a single page. Remember to hide any iframes you might add using CSS.

Note: *Beware of Race Conditions*: Depending on how you write your code, this attack could potentially have race conditions. Attacks that fail on the grader's browser during grading will receive less than full credit. To ensure that you receive full credit, you should wait after making an outbound network request rather than assuming that the request will be sent immediately. You may find using `addEventListener()` to listen for the `load` event on an `iframe` element helpful.

Part 3: Fake Login Page

More sophisticated online attacks often exploit multiple attack vectors. In this part, you will construct an attack that will either (1) steal a victim's zoobars if the user is already logged in (using the attack from exercise 8), or (2) steal the victim's username and password if they are not logged in using a fake login form. As in the last part of the lab, the attack scenario is that we manage to get the user to visit some malicious web page that we control. In this part of the lab, we will first construct the login info stealing attack, and then combine the two into a single malicious page.

Exercise 9: Make a zoobar login form

Copy the zoobar login form (either by viewing the page source, or using `zoobar/templates/login.html`) into `answer-9.html`, and make it work with the existing zoobar site. Much of this will involve prefixing URLs with the address of the web server. This file will be used as a stepping stone to the rest of the exercises in this part, so make sure you can correctly log in to the website using your fake form. Note that you should make **no** changes to the zoobar code. Submit your HTML in a file named `answer-9.html`.

Exercise 10: Intercept form submission

In order to steal the victim's credentials, we have to look at the form values just as the user is submitting the form. This is most easily done by attaching an event listener (using `addEventListener()`) or by setting the `onsubmit` attribtue of a form. For this exercise, use one of these methods to alert the user's password when the form is submitted. Submit your code in a file named `answer-10.html`.

Exercise 11: Steal password

Modify `answer-10.html` to email the password to you using the email script when the user submits the login form. Submit your code in a file named `answer-11.html`.

Hint: When a form is submitted, outstanding requests are cancelled as the browser navigates to the new page. This might lead to your request to sendmail.php not getting through. To work around this, consider cancelling the submission of the form using the `preventDefault()` method on the event object passed to the submit handler, and then use `setTimeout()` to submit the form again slightly later. Remember that your submit handler might be invoked again!

Exercise 12: hide your tracks

Modify `answer-11.html` to hide your tracks: arrange that after stealing the victim's password that the user sees the official site. Submit your code in a file named `answer-12.html`.

Hint: The zoobar application checks *how* the form was submitted (that is, whether "Log in" or "Register" was clicked) by looking at whether the request parameters contain `submit_login` or `submit_registration`. Keep this in mind when you forward the login attempt to the real login page.

Exercise 13: Side Channels and Phishing.

Modify `answer-12.html` so that your JavaScript will steal a victim's zoobars if the user is already logged in (using the attack from Part 2), or otherwise follows exercise

12: ask the victim for their username and password, if they are not logged in, and steal the victim's password. As with the previous exercise, be sure that you do **not** load the `answer-13.html` file from `http://localhost:8080/`.

The grading script will run the code once while logged in to the zoobar site before loading your page. It will then run the code a second time while *not* logged in to the zoobar site before loading your page. Consequently, when the browser loads your document, your malicious document should sniff out whether the user is logged into the zoobar site. Submit your final HTML document in a file named `answer-13.html`.

Hint: The same-origin policy generally does not allow your attack page to access the contents of pages from another domain. What types of files can be loaded by your attack page from another domain? Does the zoobar web application have any files of that type? How can you infer whether the user is logged in or not, based on this?

Challenge: Password Theft.

Create an attack that will steal the victim's username and password, even if the victim is diligent about entering their password only when the URL address bar shows http://localhost:8080/zoobar/index.cgi/login. Your solution should be contained in a short HTML document named `answer-chal.html`.

When grading, the grader will open the page using the web browser (while not logged in to zoobar). Upon loading your document, they should immediately be redirected to http://localhost:8080/zoobar/index.cgi/login. The grader will then enter a username and password, and press the "Log in" button.

Your mission, should you choose to accept it, is to make it so that when the "Log in" button is pressed, the username and password (separated by a comma) are sent by email using the email script. The login form should appear perfectly normal to the user; this means no extraneous text (e.g., warnings) should be visible, and as long as the username and password are correct, the login should proceed the same way it always does.

Hint: For this final attack, you may find that using `alert()` to test for script injection does not work; Firefox blocks it when it's causing an infinite loop of dialog boxes. Try other ways to probe whether your code is running, such as `document.loginform.login_username.value=42`.

Part 4: Profile Worm

Worms in the context of web security are scripts that are injected into pages by an attacker, and that automatically spread once they are executed in a victim's browser. The Samy worm is an excellent example, which spread to over a million users on the social network MySpace over the course of just 20 hours. In this part of the lab, you will create a

similar worm that, upon execution, will transfer 1 zoobar from the victim to the attacker, and then spread to the victim's profile. Thus, any subsequent visit to the victim's profile will cause additional zoobars to be transferred, and the worm to spread again. You will build up your solution in several stages, much like in the previous parts. This time, however, we won't spell out the steps through exercises.

Exercise 14: Profile Worm.

Your profile worm should be submitted in a file named `answer-14.txt`. To grade your attack, we will cut and paste the submitted profile code into the profile of the "attacker" user, and view that profile using the grader's account. We will then view the grader's profile with more accounts, checking for both the zoobar transfer and the replication of profile code.

In particular, we require your worm to meet the following criteria:

- When an infected user's profile is viewed, 1 zoobar is transferred from the viewing user's account into the account of the user called "attacker".
- When a user visits an infected profile, the worm (i.e., the infected profile's code) spreads to the viewing user's profile.
- An infected profile should display the message **Scanning for viruses...** when viewed, as if that was the entirety of the viewed profile.
- The transfer and replication should be reasonably fast (under 15 seconds). During that time, the grader will not click anywhere.
- During the transfer and replication process, the browser's location bar should remain at http://localhost:8080/zoobar/index.cgi/users?user=**<username>**, where **<username>** is the user whose profile is being viewed. The visitor should not see any extra graphical user interface elements (e.g., frames), and the user whose profile is being viewed should appear to have 10 zoobars, and no transfer log entries. These requirements make the attack harder to spot for a user, and thus more realistic, but they make the attack also harder to pull off.

To get you started, here is a rough outline of how to go about building your worm:

- Make a profile for the attacker to familiarize yourself with how profiles work in zoobar. You should inspect the source to get a feel for the layout of the HTML, both when editing your own profile, and when viewing someone else's.
- Modify your profile to transfer a zoobar from the user visiting the profile to the "attacker" account.
- Hide any tracks that the victim might observe.
- Arrange for the profile to be replicated

This detailed analysis of the MySpace worm may provide some inspiration.

Note: You will not be graded on the corner case where the user viewing the profile has no zoobars to send.

Hint: In this exercise, as opposed to the previous ones, your exploit runs on the same domain as the target site. This means that you are not subject to Same-Origin Policy restrictions, and that you can issue AJAX requests directly using `XMLHttpRequest` instead of `iframe`s.

Hint: For this exercise, you may need to create new elements on the page, and access data inside of them. You may find the DOM methods `document.createElement` and `document.body.appendChild` useful for this purpose.

Hint: If you choose to use `iframe`s in your solution, you may want to get access to form fields inside an `iframe`. Exactly how you do so differs by browser, but such access is always restructed by the same-origin policy. In Firefox, you can use `iframe.contentDocument.forms[0].some_field_name.value = 1;`.

Deliverables

Make sure you have the following files: `answer-1.js`, `answer-2.js`, `answer-3.txt`, `answer-5.txt`, `answer-6.html`, `answer-7.html`, `answer-8.html`, `answer-9.html`, `answer-10.html`, `answer-11.html`, `answer-12.html`, `answer-13.html`, `answer-14.txt`, and if you are doing the challenge, `answer-chal.html`, containing each of your attacks. Feel free to include any comments about your solutions in the `answers.txt` file (we would appreciate any feedback you may have on this assignment).

Run `make submit` to upload `lab4-handin.tar.gz` to the submission web site, and you're done!

Acknowledgments

Thanks to Stanford's CS155 course staff for the original version of this assignment.