ECE455
Lab #2: Browser Security

This lab is an adaptation of an assignment developed by Stanford's CS155 course and MIT Course 6.858.

Overview

Goals

- Become familiar with the browser's security model, XSS and CSRF attacks
- Try "Challenge" exercises for extra credit (Challenge +X) indicates X/100 points of extra lab credit

Grading & Submission

- This lab is graded holistically, best efforts will be considered.
- This is a learning experience, not a test!
- Run make prepare-submit to generate lab4-handin.tar.gz.
- Submit the generated .tar.gz files via Teams.

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, if you are doing the challenges. Feel free to include any comments about your solutions in the answers.txt file.

Introduction

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

1. Cross-site scripting attack (XSS)
2. Cross-site request forgery attack (CSRF)
3. Phishing/Impostor attacks

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.

VM Setup

You should download the MIT's VM image, and unpack it on your computer. This virtual machine contains an installation of Ubuntu 18.04 Linux.

To start the VM using VMware, import 6.858-x86_64-v20.vmdk. Go to File > New, select "create a custom virtual machine", choose Linux > Debian 9.x 64-bit, choose Legacy BIOS, and use an existing virtual disk (and select the 6.858-x86_64-v20.vmdk file, choosing the "Take this disk away" option). Finally, click Finish to complete the setup.

To start the VM with kvm, run ./6.858-x86_64-v20.sh from a terminal (Ctrl+A x to force quit). If you get a permission denied error from this script, try adding yourself to the kvm group with sudo gpasswd -a `whoami` kvm, then log out and log back in.

You will use the **student** account in the VM for your work. The password for the student account is **6858**. You can also get access to the root account in the VM using sudo; for example, you can install new software packages using sudo apt-get install pkgname.

You can either log into the virtual machine using its console, or use ssh to log into the virtual machine over the (virtual) network. The latter also lets you easily copy files into

and out of the virtual machine with scp or rsync. How you access the virtual machine over the network depends on how you're running it. If you're using VMWare, you'll first have to find the virtual machine's IP address. To do so, log in on the console, run ip addr show dev eth0, and note the IP address listed beside inet. With kvm, you can use localhost as the IP address for ssh and HTTP. You can now log in with ssh by running the following command from your host machine: ssh -p 2222 student@IPADDRESS. To avoid having to type the password each time, you may want to set up an SSH Key.

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 provided in Lab 1 already ensure that port 8080 on localhost is forwarded to port 8080 in the virtual machine:

```
$ kvm -m 512 -net nic -net
user,hostfwd=tcp:127.0.0.1:2222-:22,hostfwd=tcp:127.0.0.1:8080-:8080 vm-
6858.vmdk
```

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. To do so, log in as root on the console, run ip addr show dev eth0, and note the IP address listed beside inet. (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 student@VM-IP-ADDRESS student@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, fetch the latest version of the course repository, and then create a local branch called lab4 based on our lab4 branch, origin/lab4. Here are the shell commands:

```
$ git clone https://web.mit.edu/6858/2020/lab.git
Cloning into 'lab'…

$ cd lab
$ git pull
Already up-to-date.

$ git checkout -b lab4 origin/lab4
Branch lab4 set up to track remote branch lab4 from origin.
Switched to a new branch 'lab4'
$ make
...
```

Note that lab 4's source code is based on the initial web server from Lab 1.

Now you can start the zookws web server:

```
$ ./zookd 8080
```

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 the instructor.

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, 8 and 13 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 Debian 10 (78.5.0esr 64-bit) 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 Safari or Chrome (for example) may not work in Firefox. We recommend that you develop and test your code on Firefox.

Note:
The provided VM comes with a script at /etc/rc.local that creates 10 virtual lxc bridges on common IP subnets. This makes it hard to get network communications working for people that happen to use these subnets.

The included grading system uses phantomjs which hasn't been updated since 2016, so it's missing many modern features.

PhantomJS defaults to sslv3 and doesn't seem to pick up the system certs, then even when provided with the system certs it still can't connect to the MIT logging script unless you explicitly tell it to ignore SSL errors using --ignore-ssl-errors=true

Part 1: Cross-site Scripting (XSS) Attacks
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 a hold 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.

- Read about how cookies are accessed from Javascript.
- 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. Chrome and Firefox have fantastic debugging tools: the JavaScript console, the DOM inspector, and the Network monitor. 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. The Network monitor allows you 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.

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: Log the cookie.

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

Please review the instructions at https://css.csail.mit.edu/6.858/2020/labs/log.php and use that URL in your scripts to record the stolen cookie. You may log as many times as you like while working on the project, but please do not attack or abuse the logging 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 (that you saved in Exercise 1).

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>.

(Challenge +5) Exercise 4. Steal cookies.

Modify the URL so that it doesn't print the cookies but logs them for the attacker. Put your attack URL in a file named answer-4.txt.

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

(Challenge +5) 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 logging 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 expressions like

```
<style>.warning{display:none}</style>
```

and 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. You can use the DOM inspector to modify the page in real-time to check what your CSS modifications do to the page rendering.

Part 2: Cross-site Request Forgery (CSRF) Attacks

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:

- Create an account on the zoobar site and click on transfer. View the source of this page (in Chrome/Firefox, from the right-click menu and click on "View 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.
- 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.
- 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. This let's you setup a relative base URL for all relative URLs in the document.

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 submit the form, requiring no user interaction. Your goal for this exercise is to add some JavaScript to answer-6.html that automatically 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.

(Challenge +5) 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 https://css.csail.mit.edu/6.858/2020/ 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.

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 (Phishing/Impostor Attack)

More sophisticated online attacks often exploit multiple attack vectors. For example, a phishing attack will link to an impostor site to steal your password. 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 attribute 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 log the username and password (separated by a slash) to you using the logging script when the user submits the login form. Submit your code in a file named answer-11.html.

Please note that after implementing this exercise, the attacker controlled web page will no longer redirect the user to be logged in correctly. You will be fixing this issue in Exercise 12.

Hint: When a form is submitted, outstanding requests are canceled as the browser navigates to the new page. This might lead to your request to log.php not getting through. To work around this, consider canceling 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 username and 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.

(Challenge +5) 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?

Acknowledgments
Thanks to Stanford's CS155 course staff for the original version of this assignment and to MIT 6.858 for the updated version.