Computer System Security (INGI2347)[1]

Lab Session 6: Web Security - solution

Xiao Chen, Marco Canini

April 21, 2015

XSS Challenges

1) File Upload XSS

Exploit and Fix

To exploit, upload a .html file containing a script like this:

```
<script>
alert(document.cookie);
</script>
```

To fix, host the content on a separate domain so the script won't have access to any content from your domain. That is, instead of hosting user content on example.com/username we would host it at username.usercontent.example.com or username.example-usercontent.com. (Including something like "usercontent" in the domain name avoids attackers registering usernames that look innocent like wwww and using them for phishing attacks.)

2) Reflected XSS

Exploit and Fix

To exploit, create a URL like the following and get a victim to click on it:

```
http://google-gruyere.appspot.com/768153640000/<script>alert(1)</script>
```

To fix, you need to escape user input that is displayed in error messages. Error messages are displayed using error.gtl, but are not escaped in the template. The part of the template that renders the message is {{message}} and it's missing the modifier that tells it to escape user input. Add the :text modifier to escape the user input:

```
<div class="message">{{_message:text}}</div>
```

[1] A part of these exercises comes from a codelab by Bruce Leban, Mugdha Bendre, and Parisa Tabriz in Google Code University. You can visit it here: https://google-gruyere.appspot.com/ .

This flaw would have been best mitigated by a design that escapes all output by default and only displays raw HTML when explicitly tagged to do so. There are also autoescaping features available in many template systems.

3) Stored XSS

Exploit and Fix

To exploit, enter any of these as your snippet (there are certainly more methods):

```
(1) <a onmouseover="alert(1)" href="#">read this!</a>

(2) <p <script>alert(1)</script>hello

(3) </td <script>alert(1)</script>hello
```

Notice that there are multiple failures in sanitizing the HTML. Snippet 1 worked because onmouseover was inadvertently omitted from the list of disallowed attributes in sanitize.py. Snippets 2 and 3 work because browsers tend to be forgiving with HTML syntax and the handling of both start and end tags is buggy.

To fix, we need to investigate and fix the sanitizing performed on the snippets. Snippets are sanitized in _SanitizeTag in the sanitize.py file. Let's block snippet 1 by adding "onmouseover" to the list of disallowed_attributes.

Oops! This doesn't completely solve the problem. Looking at the code that was just fixed, can you find a way to bypass the fix?

Hint

Take a close look at the code in _SanitizeTag that determines whether or not an HTML attribute is allowed or not.

Exploit and Fix

The fix was insufficient because the code that checks for disallowed attributes is case sensitive and HTML is not. So this still works:

```
(1') <a ONMOUSEOVER="alert(1)" href="#">read this!</a>
```

Correctly sanitizing HTML is a tricky problem. The _SanitizeTag function has a number of critical design flaws:

- It does not validate the well-formedness of the input HTML. As we see, badly formed HTML passes through the sanitizer unchanged. Since browsers typically apply very

lenient parsing, it is very hard to predict the browser's interpretation of the given HTML unless we exercise strict control on its format.
- It uses blacklisting of attributes, which is a bad technique. One of our exploits got past the blacklist simply by using an uppercase version of the attribute. There could be other attributes missing from this list that are dangerous. It is always better to whitelist known good values.
- The sanitizer does not do any further sanitization of attribute values. This is dangerous since URI attributes like href and src and the style attribute can all be used to inject javascript.

The right approach to HTML sanitization is to:

- Parse the input into an intermediate DOM structure, then rebuild the body as well-formed output.
- Use strict whitelists for allowed tags and attributes.
- Apply strict sanitization of URL and CSS attributes if they are permitted.

Whenever possible it is preferable to use an already available known and proven HTML sanitizer.

4) Stored XSS via HTML Attribute

Exploit and Fixes

To exploit, use the following for your color preference:

```
red' onload='alert(1)' onmouseover='alert(2)
```

You may need to move the mouse over the snippet to trigger the attack. This attack works because the first quote ends the style attribute and the second quote starts the onload attribute.

But this attack shouldn't work at all. Take a look at home.gtl where it renders the color. It says style='{{color:text}}' and as we saw earlier, the :text part tells it to escape text. So why doesn't this get escaped? In gtl.py, it calls cgi.escape(str(value)) which takes an optional second parameter that indicates that the value is being used in an HTML attribute. So you can replace this with cgi.escape(str(value),True). Except that doesn't fix it! The problem is that cgi.escape assumes your HTML attributes are enclosed in double quotes and this file is using single quotes. (This should teach you to always carefully read the documentation for libraries you use and to always test that they do what you want.)

You'll note that this attack uses both onload and onmouseover. That's because even though W3C specifies that onload events is only supported on body and frameset elements, some browsers support them on other elements. So if the victim is using one of those browsers, the attack always succeeds. Otherwise, it succeeds when the user moves the mouse. It's not uncommon for attackers to use multiple attack vectors at the same time.

To fix, we need to use a correct text escaper, that escapes single and double quotes too. Add the following function to gtl.py and call it instead of cgi.escape for the text escaper.

```
def _EscapeTextToHtml(var):
  """Escape HTML metacharacters.

  This function escapes characters that are dangerous to insert into
  HTML. It prevents XSS via quotes or script injected in attribute values.

  It is safer than cgi.escape, which escapes only <, >, & by default.
  cgi.escape can be told to escape double quotes, but it will never
  escape single quotes.
  """
  meta_chars = {
      '"': '&quot;',
      '\'': '&#39;',  # Not &apos;
      '&': '&amp;',
      '<': '&lt;',
      '>': '&gt;',
      }
  escaped_var = ""
  for i in var:
    if i in meta_chars:
      escaped_var = escaped_var + meta_chars[i]
    else:
      escaped_var = escaped_var + i
  return escaped_var
```

Oops! This doesn't completely solve the problem. Even with the above fix in place, the color value is still vulnerable.

Hint 1

Some browsers allow you to include script in stylesheets.

Hint 2

The easiest browser to exploit in this way is Internet Explorer which supports dynamic CSS properties.

Another Exploit and Fix

Internet Explorer's dynamic CSS properites (aka CSS expressions) make this attack particularly easy.

To exploit, use the following for your color preference:

```
expression(alert(1))
```

While other browsers don't support CSS expressions, there are other dangerous CSS properties, such as Mozilla's -moz-binding.

To fix, we need to sanitize the color as a color. The best thing to do would be to add a new output sanitizing form to gtl, i.e., we would write {{foo:color}} which makes sure foo is safe to use as a color. This function can be used to sanitize:

```
SAFE_COLOR_RE = re.compile(r"^#?[a-zA-Z0-9]*$")

def _SanitizeColor(color):
  """Sanitizes a color, returning 'invalid' if it's invalid.

  A valid value is either the name of a color or # followed by the
  hex code for a color (like #FEFFFF). Returning an invalid value
  value allows a style sheet to specify a default value by writing
  'color:default; color:{{foo:color}}'.
  """

  if SAFE_COLOR_RE.match(color):
    return color
  return 'invalid'
```

Colors aren't the only values we might want to allow users to provide. You should do similar sanitizing for user-provided fonts, sizes, urls, etc. It's helpful to do input validation, so that when a user enters an invalid value, you'll reject it at that time. But only doing input validation would be a mistake: if you find an error in your validation code or a new browser exposes a new attack vector, you'd have to go back and scrub all previously entered values. Or, you could add the output validation which you should have been doing in the first place.

5) Stored XSS via AJAX

Exploit and Fixes

To exploit, Put this in your snippet:

```
all <span style=display:none>"
+ (alert(1),"")
+ "</span>your base
```

The JSON should look like

```
_feed(({..., "Mallory": "snippet", ...}))
```

but instead looks like this:

```
_feed({..., "Mallory": "all <span style=display:none>"
+ (alert(1),"")
+ "</span>your base", ...})
```

Each underlined part is a separate expression. Note that this exploit is written to be invisible both in the original page rendering (because of the <span style=display:none>) and after refresh (because it inserts only an empty string). All that will appear on the screen is all your base. There are bugs on both the server and client sides which enable this attack.

To fix, first, on the server side, the text is incorrectly escaped when it is rendered in the JSON response. The template says {{snippet.0:html}} but that's not enough. This text is going to be inserted into the innerHTML of a DOM node so the HTML does have to be sanitized. However, that sanitized text is then going to be inserted into Javascript and single and double quotes have to be escaped. That is, adding support for {{...:js}} to GTL would not be sufficient; we would also need to support something like {{...:html:js}}.

To escape quotes, use \x27 and \x22 for single and double quote respectively. Replacing them with &#27; and &quot; is incorrect as those are not recognized in Javascript strings and will break quotes around HTML attribute.

Second, in the browser, Gruyere converts the JSON by using Javascript's eval. In general, eval is very dangerous and should rarely be used. If it used, it must be used very carefully, which is hardly the case here. We should be using the JSON parser which ensures that the string does not include any unsafe content. The JSON parser is available at json.org.

6) Client-State Manipulation (extra)

6a) Elevation of Privilege

Exploit and Fixes

You can convert your account to being an administrator by issuing either of the following requests:

- http://google-gruyere.appspot.com/768153640000/saveprofile?action=update&is_admin=True
- http://google-gruyere.appspot.com/768153640000/saveprofile?action=update&is_admin=True&uid=username (which will make any username into an an admin)

After visiting this URL, your account is now marked as an administrator but your cookie still says you're not. So sign out and back in to get a new cookie. After logging in, notice the 'Manage this server' link on the top right.

The bug here is that there is no validation on the server side that the request is authorized. The only part of the code that restricts the changes that a user is allowed to make are in the template, hiding parts of the UI that they shouldn't have access to. The correct thing to do is to check for authorization on the server, at the time that the request is received.

6b) Cookie Manipulation

Exploit and Fix

You can get Gruyere to issue you a cookie for someone else's account by creating a new account with username "foo|admin|author". When you log into this account, it will issue you the cookie "hash|foo|admin|author||author" which actually logs you into foo as an administrator. (So this is also an elevation of privilege attack.)

Having no restrictions on the characters allowed in usernames means that we have to be careful when we handle them. In this case, the cookie parsing code is tolerant of malformed cookies and it shouldn't be. It should escape the username when it constructs the cookie and it should reject a cookie if it doesn't match the exact pattern it is expecting.

Even if we fix this, Python's hash function is not cryptographically secure. If you look at Python's string_hash function in python/Objects/stringobject.cc you'll see that it hashes the string strictly from left to right. That means that we don't need to know the cookie secret to generate our own hashes; all we need is another string that hashes to the same value, which we can find in a relatively short time on a typical PC. In contrast, with a cryptographic hash function, changing any bit of the string will change many bits of the hash value in an unpredictable way. At a minimum, you should use a secure hash function to protect your cookies. You should also consider encrypting the entire cookie as plain text cookies can expose information you might not want exposed.

And these cookies are also vulnerable to a replay attack. Once a user is issued a cookie, it's good forever and there's no way to revoke it. So if a user is an administrator at one time, they can save the cookie and continue to act as an administrator even if their administrative rights are taken away. While it's convenient to not have to make a database query in order to check whether or not a user is an administrator, that might be too dangerous a detail to store in the cookie. If avoiding additional database access is important, the server could cache a list of recent admin users. Including a timestamp in a cookie and expiring it after some period of time also mitigates against a replay attack.

Another challenge: Since account names are limited to 16 characters, it seems that this trick would not work to log in to the actual administrator account since "administrator|admin" is 19 characters. Can you figure out how to bypass that restriction?

Additional Exploit and Fix

The 16 character limit is implemented on the client side. Just issue your own request:

```
http://google-gruyere.appspot.com/768153640000/saveprofile?action=new&uid=administrator|admin|author&pw=secret
```

Again, this restriction should be implemented on the server side, not just the client side.