Security professional

By

Ramy Naiem

a. XSS Challenges:

i. File Upload XSS: Can you upload a file that allows you to execute arbitrary script on the google-gruyere.appspot.com domain?

Yes, the user can upload a HTML file. The file can also contain a script that will run on the website

FIX: The script doesn't have access to your site's content, put the material on a different domain.

ii. Reflected XSS: Find a reflected XSS attack. What we want is a URL that when clicked on will execute a script

https://google-gruyere.appspot.com/5509607633096798270975960094097661536833/%c0%be%c0%bc

FIX: A design that escapes all output by default and only shows raw HTML when expressly labelled to do so would have been the best way to address this vulnerability.

iii. Stored XSS: Now find a stored XSS. What we want to do is put a script in a place where Gruyere will serve it back to another user.

read this!

FIX: Rebuild the body as well-formed output after parsing the input into an intermediate DOM structure.

Allow just certain tags and characteristics to be used.

If URL and CSS elements are allowed, use stringent sanitization.

iv. Stored XSS via HTML Attribute: you can also do XSS by injecting a value into an HTML attribute. Inject a script by setting the colour value in a profile

To activate the assault, you may have to move your cursor over the

excerpt. Because the first quotation ends the style property and the second quote begins the onload attribute, the assault is successful.

This assault, on the other hand, should not be successful. Examine the colour rendering in home.gtl. Style='color:text' directs it to escape text, as we saw before. So why isn't it possible to get out of this? It calls cgi.escape(str(value)) in gtl.py, which accepts an optional second argument indicating that the value is being used in an HTML attribute. cgi.escape(str(value),True) may be used instead. That, however, is not the solution! The issue is that cgi.escape expects your HTML attributes to be encased in double quotes, but this file only has single quotes. (This should teach you to read the documentation for the libraries you use carefully and to test them to ensure they perform what you want.)

Onload and on mouseover are used in this attack. This is because, despite the fact that the W3C only allows onload events on body and frameset elements, certain browsers allow them on other elements. As a result, the attack always succeeds if the target is using one of those browsers. If the user moves the mouse, it succeeds otherwise. Multiple attack vectors are often used by attackers.

FIX: We need to utilize a text escaper that can handle both single and double quotes. To use the text escaper, add the following method to gtl.py and use it instead of cgi.escape:

```
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
```

Stored XSS via AJAX: Find an XSS attack that uses a bug in Gruyere's AJAX code

```
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", ...})
```

FIX: First, when the text is presented in the JSON response, it is erroneously escaped on the server side.

v. Reflected XSS via AJAX: Find a URL that when clicked on will execute a script using one of Gruyere's AJAX features

https://google-

gruyere.appspot.com/5509607633096798270975960094097661536833/feed.gtl?uid=<script>alert(1)</script>

https://google-gruyere.appspot.com/5509607633096798270975960094097661536833/feed.gtl?uid=%3Cscript%3Ealert(1)%3C/script%3E

FIX: You must ensure that your JSON content cannot be misinterpreted as HTML. Despite the fact that literal and > are permitted in JavaScript strings, you must ensure that they do not occur literally in places where a browser may misunderstand them. As a result, you'll need to change...:js to use the JavaScript escapes x3c and x3e instead. It is usually safer to use 'x3cx3e' instead of '>' in Javscript strings. (As previously stated, employing HTML escapes and > is inappropriate.)

Always specify the content type of your replies, in this example application/javascript for providing JSON results. This does not address the issue since browsers do not always respect the content type: browsers will occasionally "sniff" results from servers that do not give the right content type in order to "fix" them.

There's more, however! Gruyere also doesn't specify the content encoding. Furthermore, certain browsers attempt to guess the encoding type of a document, or an attacker may be able to insert material that specifies the content type in a document. Because +ADw- and +AD4- are alternative encodings for and >, an attacker might, for example, include a script tag as +ADw-script+AD4- if the browser thinks the document is UTF-7. As a result, always specify the content type and encoding of your answers, for example, for HTML:

charset=utf-8; content-type: text/html

Client State Manipulation:

i. Elevation of Privilege: Convert your account to an administrator account

ttps://google-

gruyere.appspot.com/550960763309679827097596009409766153683/saveprofile?action=update&is_admin=True

FIX: Checking for authorization on the server when the request is received is the proper procedure.

ii. Cookie Manipulation: Get Gruyere to issue you a cookie for someone else's account

Create a new account with the username "foo|admin|author" to get Gruyere to give you a cookie for someone else's account. When you connect into this account, the cookie "hash|foo|admin|author||author" is set, which allows you to log into foo as an administrator.

FIX: Create a new account with the username "foo|admin|author" to get Gruyere to give you a cookie for someone else's account. When you connect into this account, the cookie "hash|foo|admin|author||author" is set, which allows you to log into foo as an administrator. (This is also a privilege elevation attack.)

Because there are no constraints on the characters that may be used in usernames, we must be cautious while working with them. The cookie processing algorithm in this situation is tolerant of faulty cookies, which it shouldn't be. When creating the cookie, it should escape the username and reject any cookie that does not fit the precise pattern it expects.

c. XSRF: Find a way to get someone to delete one of their Gruyere snippets

https://google-gruyere.appspot.com/550960763309679827097596009409766153683/deletesnippet?index=0

FIX: Because this is a state-changing activity, we should

make /deletesnippet operate through a POST request first. Change method='get' to method='post' in the HTML form. GET and POST requests seem the same on the server, but they normally invoke separate handlers. Gruyere, for example, employs Python's BaseHTTPServer, which uses do GET and do POST for GET and POST requests, respectively.

d. XSSI: Find a way to read someone else's private snippet using XSSI

```
<script>
function _feed(s) {
  alert("Your private snippet is: " +
s['private_snippet']);
}
</script>
<script src="https://google-
gruyere.appspot.com/5509607633096798270975960094097661536833/feed.gtl"></script>
```

FIX: The most common method is to apply a non-executable prefix to it. Scripts on the same domain can read the answer and take off the prefix, but scripts in other domains can't.

Path Traversal:

i. Information disclosure via path traversal: Find a way to read secret.txt from a running Gruyere server

You may steal secret.txt by visiting this URL:

https://google-gruyere.appspot.com/550960763309679827097596009409766153683/../secret.txt

Some browsers, such as Firefox and Chrome, remove../ from URLs. Because an attacker will use percent 2f to represent / in the URL, or a tool like curl, a web proxy, or a browser that doesn't conduct the optimization, this provides no

security protection. However, if you test your app using one of these browsers to determine whether it's susceptible, you may believe you're safe when you're not.

FIX: Access to files outside the resource's directory must be restricted. Validating file paths is a little challenging since there are many methods to mask path parts like "../" or "" that enable you to escape out of the resources folder. The greatest defense is to deliver just specified resource files. You may either hardcode a list or browse the resource directory and construct a list of files when your application begins. Only accept requests for those files after that. You may also apply some optimization here, such as caching tiny files in memory to speed up your programme. If you want to validate a file path, you should do it on the end path rather than the URL, since there are many ways to express the same characters in URLs. It is important to note that changing file permissions will not function. This file must be readable by Gruyere.

ii. Data tampering via path traversal: Find a way to replace secret.txt on a running Gruyere server

Before using it, you should replace any harmful characters in the username with safe ones. It was previously proposed that we limit the characters that may be used in a username, but you probably didn't realise that "." was a harmful character. It's worth mentioning that this approach has a vulnerability that only affects Windows servers. Filenames are not case sensitive on Windows, but Gruyere usernames are. So, by inventing an identical username that varies only in case, such as BRIE instead of brie, one person may target another user's stuff. So we need to transform the username to a canonical form that is distinct for each username, not only escape hazardous characters. Alternatively, we might avoid all of these problems by issuing each user a unique identification.

FIX: Admins should never keep user data in the same location as application files, however this will not prevent

you from these attacks since a user may insert destructive code into the website.

Denial of Service:

i. DoS Quit the Server: Find a way to make the server quit

https://google-gruyere.appspot.com/5509607633096798270975960094097766153683/quitserver

FIX: To fix, add /quitserver to the URLS only administrators may access.

ii. DoS Overloading the Server: find a way to overload the server when it processed a request

create a file named menubar.gtl containing:[[include:menubar.gtl]]DoS[[/include:menubar.gtl]]
FIX: implement the route traversal and template uploading safeguards mentioned previously.

Code Execution: Find a code execution exploit

make a copy of gtl.py (or sanitize.py) and add some exploit code. Now you can either upload a file named ../gtl.py or create a user named .. and upload gtl.py. Then, make the server quit by browsing to https://google-gruyere.appspot.com/5509607633096798270975960094097766153683/quitserver

Fix: The two previous exploits have to be fixed

Configuration Vulnerabilities:

i. Information disclosure 1: Read the contents of the database off of a running server by exploiting a configuration vulnerability

you can use the debug dump page dump.gtl to display the contents of the database via the following URL: https://googlegruyere.appspot.com/550960763309679827097596009409766153683/dump.gtl

FIX: Ensure that no debug features are installed. Delete dump.gtl in this scenario. This is an example of a debug feature that might be accidentally left in a programme. If a debug feature is required, it should be securely secured: only admin users should have access, and requests from debug IP addresses should be permitted.

This vulnerability makes the passwords of the users public. Cleartext passwords should never be used. Password hashing should be used instead. The concept is that you don't need to know a user's password to authenticate them; all you need to know is that they know it. You only save a cryptographic hash of the password and a salt value when the user sets their password. You recompute the hash when the user re-enters their password later, and if it matches, you know the password is correct. If an attacker has the hash value, reversing it to recover the original password is very tough. (Which is a good thing, since despite all the caution, users routinely reuse the same weak password across several sites.)

ii. Information disclosure 2: Even after implementing the fix described above, an attacker can undo it and execute the attack! How can that be?

Gruyere lets users upload any form of file, including.gtl files. As a result, the attacker may simply upload and read their own copy of dump.gtl or a similar file. Indeed, as previously said, hosting arbitrary material on the server, whether it's HTML, JavaScript, Flash, or anything else, is a big security issue. Allowing a file with an unknown file type may result in a future security flaw.

FIX: Only Gruyere-related files should be considered as templates.

Store user-submitted files separately from application files.

iii. Information disclosure 3: Even after implementing the fixes described above, a similar attack is still possible through a different attack vector. Can you find it?

To exploit, add this to your private snippet:{{_db:pprint}}

FIX: Change the template code so that added variable values are never reparsed. ExpandTemplate calls _ExpandBlocks then _ExpandVariables, but _ExpandBlocks calls ExpandTemplate on nested blocks, resulting in a bug in the code. If a variable is enlarged within a nested block and includes anything that looks like a variable template, the variable will be expanded again. That sounds difficult because it is difficult. Because parsing blocks and variables individually is a fundamental fault in the expander's architecture, the repair is not simple.

iv.

v. Because the template language allows for unrestricted database access, this attack is conceivable. It would be safer if the templates could only access data that was supplied to them expressly. A template, for example, may be linked to a database query, with just the data matching that query being provided to the template. This would confine the scope of a problem like this to data that the user already has access to.AJAX Vulnerabilities:

vi.

i. DoS via AJAX: Find an attack that prevents users from seeing their private snippets on the home page

Create a user called private snippet and at least one snippet to exploit. The JSON response will then be 'private snippet': 'user's private snippet>,...,'private snippet': 'attacker's snippet>, and the attacker's snippet will take the place of the user's.

FIX: The AJAX code must ensure that the data is only sent to the correct location. The problem is that the JSON

structure isn't very sturdy. [private snippet>, user>: snippet>,...] would be a preferable structure.

ii. Phishing via AJAX: Find a way to change the sign in link in the upper right corner to point to https://evil.example.com.

Create a menu-right user and publish a snippet that matches the right side of the menu bar.

Sign in| Sign up

The navigation bar will seem incorrect if the user is already signed in. But that's OK since the user will most likely believe they were accidently logged out of the website and will log back in.

FIX: The DOM modification procedure should be made more robust. When using user values as DOM element IDs, make sure there aren't any conflicts, as there is here, for example, by prefixing user values with id="user_." Better still, instead of using user values, create your own IDs.

When the user hits Sign in and is redirected to evil.example.com, this spoofing attack is readily identified. A savvy attacker might replace the Sign in link with a script that presents the sign in form on the current page, with the form submission travelling to their server, making it harder to detect.