CVE-2018-1000006

Rao Zvorovski

University of Tartu
Institute of Computer Sciences
LTAT.06.002

2018-04-30
Abstract

The vulnerability CVE-2018-1000006 affects the open-source framework Electron, which allows the development of graphical desktop applications using web application components by integrating the Chromium web browser. The affected part of the framework is the processing of command line arguments in custom default protocol handlers (à la ‘myapp://’). As such, multiple applications built on certain versions of Electron were affected and contained a code injection vulnerability, where tricking users to click specially crafted links would cause arbitrary code execution. Amongst affected applications were Skype, Slack, Exodus cryptocurrency wallet and multiple others. Platforms other than Windows were not affected.
## Contents

1 Overview of the vulnerability .................................................. 2  
2 Exploitation ........................................................................... 2  
3 Fix ......................................................................................... 3  

## Listings

1 Enabling the protocol handler .................................................. 2  
2 A specially crafted link containing an example payload ........... 2  
3 Function to check command line arguments ................................ 4  
4 Function to check if the argument is in a blacklist ................. 5  

## List of Figures

1 Navigating to the link via JavaScript ........................................ 3  
2 Internet Explorer asking for user’s consent ............................... 3  
3 The payload in six instances after allowing the link ............... 4
1 Overview of the vulnerability

In applications built on versions of Electron prior to 1.8.2-beta.5, 1.7.12 and 1.6.17, that registered themselves as the default handler for a protocol such as ‘myapp://’ would allow passing arbitrary command line arguments to the Chromium engine built into Electron as part of a link. This vulnerability only affected the Windows platform.[1]

Certain Chromium flags, such as --gpu-launcher would allow for arbitrary code execution. Thus it is sufficient to trick an user who has a vulnerable application installed to click on a link, crafted by the attacker, to attain arbitrary command execution.[2]

Multiple significant applications were affected, among those are Skype, Slack, Visual Studio Code, and the Exodus cryptocurrency wallet.[3][4][2]

2 Exploitation

Due to the nature of the vulnerability, many different applications are vulnerable to the same exploit. The difference in exploitation mostly lies in the protocol identifier used. The author uses an example application to showcase the vulnerability.

The important part of the code is enabling of the protocol handler (listing 1), which sets the registry keys to launch the application to handle the protocol for the application; “exploit://” in this example.[5]

```javascript
const PROTOCOL = 'exploit://'
const PROTOCOL_PREFIX = PROTOCOL.split(':')[0]
app.setAsDefaultProtocolClient(PROTOCOL_PREFIX)
```

Listing 1: Enabling the protocol handler

Then, when the user clicks on a ‘exploit://’ link (or navigates to that address in another way), the app is started, with the URI of the link as a command line argument, enclosed in double quotes[2]. However, double quotes are not escaped, so they can be broken out of just by using another double quote in the link. Then the --gpu-launcher flag, which is a Chromium debug flag for launching the GPU process, can be appended, which can then include the payload[2][6]. Thus we get a suitable link (listing 2). The author notes that the payload is ran multiple times by Electron’s Chromium, not just once — payload should be able to handle that.

```javascript
exploit://asd” --gpu-launcher=”cmd /c start calc” asd
```

Listing 2: A specially crafted link containing an example payload
The link can be triggered for example by using JavaScript’s `window.location` (fig. 1)[2]. The author notes that this cannot be done silently: browsers usually ask for user consent when running other applications via protocol handlers (fig. 2). It can, however, be expected that users will ignore these warnings[7]. The author also notes that the Chrome browser url-encodes the link, which makes exploitation at least harder.

Then, after the user gives their consent, the vulnerable application runs, and then the payload (in this case, the Windows command prompt, which launches the Windows calculator) is run multiple times — in author’s tests the number of runs is 6 (fig. 3).

Figure 1: Navigating to the link via JavaScript

Figure 2: Internet Explorer asking for user’s consent

3 Fix

For the vulnerable applications, there are two main fixes: upgrade to a version of Electron for which the vulnerability is patched, or append ‘—’ as the last argument to the command, when enregistering the protocol client, which stops the processing of arguments for Chromium, thus working around the issue.[1]

For Electron itself, the patch involved filtering the passed command line arguments when launched as protocol handler, based on a blacklist[2][8]. The commit added a function (listing 3) to check if the command line arguments are good after decoding, using the blacklist (listing 4) only if the application is launched from an URL — if a blacklisted argument is discovered, the launch is stopped[8]. The contents of the blacklist are too long to reproduce in this report.
This patch effectively fixes the issue on the Electron side due to attackers no longer being able to invoke the Chromium flags which caused the issue; applications could still be vulnerable in their own processing of command line switches, but that is out of scope for Electron to deal with. There could also still be ways around this through strange encodings; there are some tests included with the commit for that[8]. The author believes that the patch is sufficient.

The author notes a flaw with the fix: it requires maintaining a current (to the version of Chromium included in Electron) list of dangerous command line flags, which is prone to human error, since some potentially dangerous flags could be missed.

```cpp
bool CheckCommandLineArguments(int argc, base::CommandLine::CharType** argv) {
  const base::CommandLine::StringType dashdash(2, '-');
  bool block_blacklisted_args = false;
  for (int i = 0; i < argc; ++i) {
    if (argv[i] == dashdash)
      break;
    if (block_blacklisted_args) {
      if (IsBlacklistedArg(argv[i]))
        return false;
    } else if (IsUrlArg(argv[i])) {
      block_blacklisted_args = true;
    }
  }
  return true;
}
```

Listing 3: Function to check command line arguments
bool IsBlacklistedArg(const base::CommandLine::CharType* arg) {
#if defined(OS_WIN)
  const auto converted = base::WideToUTF8(arg);
  const char* a = converted.c_str();
#else
  const char* a = arg;
#endif

  static const char* prefixes[] = {"--", "-", "/"};

  int prefix_length = 0;
  for (auto& prefix : prefixes) {
    if (base::StartsWith(a, prefix, base::CompareCase::SENSITIVE)) {
      prefix_length = strlen(prefix);
      break;
    }
  }

  if (prefix_length > 0) {
    a += prefix_length;
    std::string switch_name(a, strcspn(a, "="));
    for (auto& item : blacklist) {
      if (switch_name == item)
        return true;
    }
  }

  return false;
}

Listing 4: Function to check if the argument is in a blacklist
References


