

# Requesting Azure AD Request Tokens on Azure-AD-joined Machines for Browser SSO



Lee Chagolla-Christensen · Follow  
Published in Posts By SpecterOps Team Members · 7 min read · Jul 14, 2020

--

1

RequestAADRefreshToken is a tool that returns OAuth 2.0 refresh tokens for an Azure-AD-authenticated Windows user (i.e. the machine is joined to Azure AD and a user logs in with their Azure AD account) wanting to perform SSO authentication in the browser. An attacker can use this to authenticate to Azure AD in a browser as that user.

I discovered this feature while reading through the Azure AD documentation where it stated

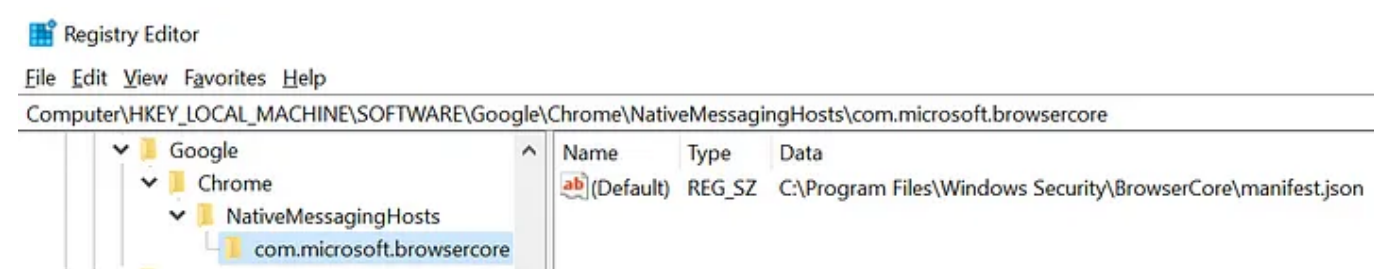
*In Windows 10, Azure AD supports browser SSO in Internet Explorer and Microsoft Edge natively or in Google Chrome via the Windows 10 accounts extension*

I installed the Windows 10 Accounts extension in Chrome on an Azure-AD-joined machine. Upon doing so, I could now login to Azure-AD-authenticated services like portal.azure.com without needing to enter my password! I thought that being able to generate Azure AD authentication material was a useful piece of tradecraft to have, so I began reversing how the Chrome extension works.

The Windows 10 Accounts Chrome extension communicates with Windows using the `chrome.runtime.sendMessage` function from the Chrome Native Messaging APIs:

```
background.js X
5 background.js > chrome.runtime.onMessage.addListener() callback
9 chrome.runtime.sendMessage(
10   "com.microsoft.browsercore",
11   request,
12   function (response) {
13     if (response != null) {
14       if (response.status && response.status == "Fail" && response.code) {
15         sendResponse(response);
16       }
17       else {
18         sendResponse({
19           status: "Success",
20           result: response
21         });
22       }
23     }
24     else {
25       sendResponse({
26         status: "Fail",
27         code: "NoSupport",
28         description: chrome.runtime.lastError.message,
29       });
30     }
  }
```

As can be seen, the first argument passed to `sendMessage` is the string "com.microsoft.browsercore", which refers to a native messaging host that is created in the registry when Chrome is installed:



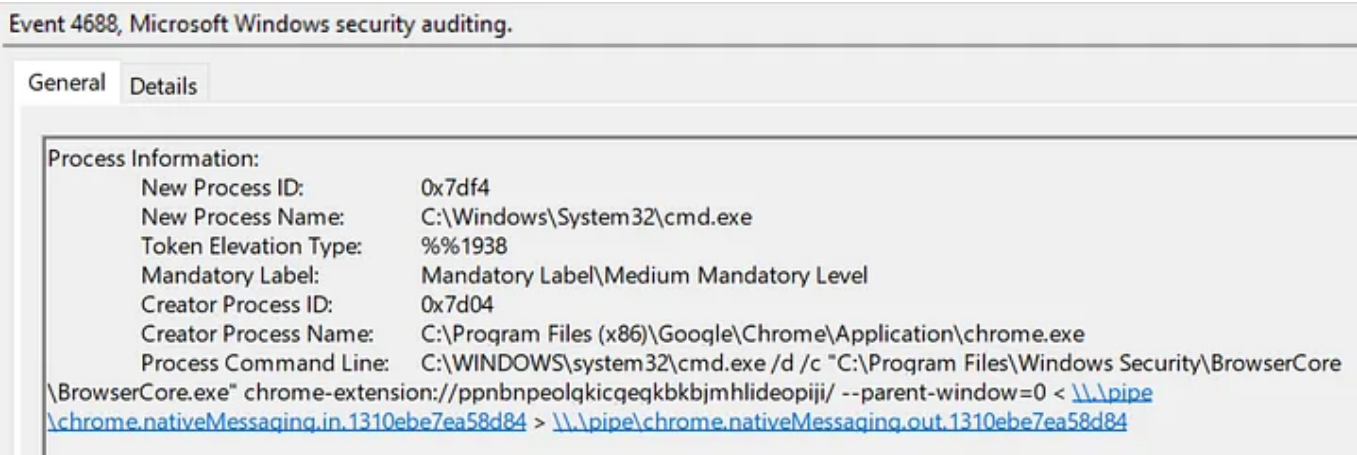
Looking at the JSON manifest specified there, we can see that the actual application that the extension communicates with is BrowserCore.exe (Full path: C:\Program Files\Windows Security\BrowserCore\BrowserCore.exe):

```
{ manifest.json X
c: > Program Files > Windows Security > BrowserCore > { manifest.json > ...
1 {
2   "name": "com.microsoft.browsercore",
3   "description": "BrowserCore",
4   "path": "BrowserCore.exe",
5   "type": "stdio",
6   "allowed_origins": [
7     "chrome-extension://ppnbnpeolgkicgegkbkbjmhlideopiji/",
8     "chrome-extension://ndjpnladcallmjemlbaebfadecfhkep/"
9   ]
10 }
```

Next, I wanted to see what this communication looked like. Per the docs, each time the extension calls `sendMessage`, BrowserCore.exe should start a receive input via stdin and send responses via stdout. I had no clue how Chrome or BrowserCore would actually implement that, so to maybe

give me more clues about what’s going on, I thought I’d just use the extension and see the messaging dance in action.

First attempt was to open Process Explorer, try and open Chrome’s incognito window repeatedly with the extension enabled, and try and authenticate a bunch of times - racing the process to see if I could click on it before it closed. Ultimately, my clicks (and Process Explorer) were too slow to catch BrowserCore.exe before it exited. However, since I have process and command line logging enabled for occasions exactly like this (like any normal human does), I cracked open event viewer, opened a new Chrome browser session, visited the Azure Portal login page, and this is what I saw:



Ahhh clever, so Chrome devs did a little shortcut implementing native messaging I/O and used named pipes in conjunction with the Windows command prompt’s stdin/stdout redirection operators. Cool, so now I knew Chrome spawns BrowserCore.exe and uses named pipes to pass messages around.

Next I wanted to know what BrowserCore.exe was actually doing to get the refresh token. So I cracked open dnSpy, crossing my fingers that it’d be a .NET executable since my binary reversing skills are crap and was hoping the fates would be nice to me today, but was sadly disappointed - it wasn’t .NET. Okay fine. I put a couple extra sticks of RAM into my machine, opened Ghidra, (re)figured out how to setup symbols in Ghidra, and started looking at a bunch of windows that I mostly don’t know how to use. One of the first things I do when looking at an unknown binary is look at the imports, and while doing that with BrowserCore.exe I noticed it imports `CoCreateInstance` :

I’ve done some COM programming and research in my time, so I knew applications use `CoCreateInstance` to instantiate COM objects. The function is only referenced once and given that this binary isn’t very large and I’m horrible at reversing, I figure I’d start there since it’s something I’m familiar with. `CoCreateInstance`’s first argument is a GUID — the CLSID of the COM object:

Looking up the CLSID `{a9927f85-a304-4390-8b23-a75f1c668600}` in the registry revealed our friendly neighborhood COM DLL  
`MicrosoftAccountTokenProvider.dll`:

I put on my gloves and goggles, grabbed myself some whiskey, and and braced for another similarly vicious round of Ghidra reversing so I could figure out COM interface...

But then I decided to Google the CLSID and see what would happen. And whad’ya know! It’s documented!

So with that I implemented a quick program (`RequestAADRefreshToken`) that leverages the COM object just to see what would happen when I invoked the `GetCookieInfoForUri` method. And lucky for me, leveraging this COM object did indeed return a token that I could use in a cookie when authenticating! To use it, one could take the following steps:

1. Obtain access to a Azure AD user context on an Azure-AD-joined device.  
An easy way to tell is to run the command `dsregcmd.exe /status`. If this is abusable, there will be a section titled “SSO State” and `AzureAdPrt` will be set to YES.
2. On an Azure-AD-joined machine where an Azure AD user has logged in, run `RequestAADRefreshToken.exe`. This will return the refresh token.

3. Clear your browser cookies and go to <https://login.microsoftonline.com/login.srf>
4. In Chrome, Open the Developer Tools (F12) -> Application -> Cookies
5. Delete all cookies. Then add one named `x-ms-RefreshTokenCredential` and set its value to the JSON web token (JWT) in the `Data` field that RequestAADRefreshToken.exe output.

6. Refresh the page (or visit <https://login.microsoftonline.com/login.srf> again) and you'll be logged in. How neat is that? That's pretty neat!

## Telemetry Sources

### DLL Load Events

When using the COM object, the process will load the DLL `C:\Windows\System32\MicrosoftAccountTokenProvider.dll`. A less precise detection could monitor for whenever this DLL is loaded and baseline the normal processes in an environment (this still could be useful for correlation of activities). Some processes that we've observed normally loading this DLL are:

- BackgroundTaskHost.exe - `ServerNameBackgroundTaskHost.WebAccountProvider`
- devenv.exe
- git-credential-manager

- iexplore.exe
- MicrosoftEdge.exe -  
ServerName:MicrosoftEdge.AppXdnjhccw3zf0j06tkg3jtqr00qdm0khc.mca

## TraceLogging/Event Tracing for Windows (ETW)

TraceLogging provides some telemetry when executing RequestAADRefreshToken. While reversing MicrosoftAccountTokenProvider.dll I noticed the code was littered with trace logging functions. For example, this call from `GetCookieInfoForUri`:

Ultimately these tracing functions end up calling EventWriteTransfer underneath, writing the log message to a ETW provider (if you're unfamiliar with ETW, I can suggest a few great resources). Since TraceLogging is built on top of ETW, this all makes sense.

So given that it's using TraceLogging/ETW, how could I tap into the events it's generating? The first step was identifying the ETW provider's GUID. One method I took to determine this is by locating calls to EventRegister, which takes as the ETW provider's GUID as its first argument and registers the provider in the application. Looking at MicrosoftAccountTokenProvider.dll's imports, I could easily do this:

Finding references to EventRegister via the file's PE import table

Jumping to the actual call to `EventRegister` . Note the first argument refers to a variable containing the ETW provider's GUID

Viewing the ETW provider's GUID as {05f02597-fe85-4e67-8542-69567ab8fd4f}

To further confirm this, I ran Matt Graeber's [TLGMetadataParser.psm1](#) script against `MicrosoftAccountTokenProvider.dll` to carve out `TraceLogging` providers and `TraceLogging` event metadata from the DLL:

[etwbreaker](#) from the great folks over at Airbus also looks like an interesting tool that I could have also used to further confirm this(Disclaimer: I haven't used it yet). With that knowledge, I could then start a trace session and explore the generated events. I wrote a simple PowerShell cmdlet — [Start-EtwTrace](#) — a while back to quickly do this for me, saving the output to a .evtx as well as returning the events in PS so I could interact with them:

```
$Events = Start-EtwTrace `
    -ProviderGuid '05f02597-fe85-4e67-8542-69567ab8fd4f' `
    -OutputFile .\out3.evtx `
    -ProcessPath C:\RequestAADRefreshToken.exe
```

```
$Events | select  
ProcessId,Id,LevelDisplayName,ProviderName,KeywordsDisplayNames,Message | ft -AutoSize
```



As can be seen, the second event informs us of the call to `GetCookieInfoForUri` as well as other function calls. It is worth noting that the the behavior changes depending on the login URL. For example, in the above screenshot, all events originated from the the `RequestAADRefreshToken.exe` executable because the default Azure AD login URL (`https://login.microsoftonline.com`) was used. However, in my testing when I switched the login URL to the Microsoft Account URL (`https://login.live.com`), I saw events both from `RequestAADRefreshToken.exe` and `lsass.exe` (assumedly from the authentication package `MicrosoftAccountCloudAP.dll`).

### Summary

Regardless of the technology, network defenders should be diligent in their understanding of their authentication systems and how users and administrators can access the system’s credentials. As more and more Windows hosts are using Azure AD for authentication, it’s important to understand authentication changes on these machines and what new credential sources might be on these machines. `RequestAADRefreshToken.exe` takes advantage of one of these “new” credentials and grants an attacker the ability to generate a user credential without knowledge of the user’s plaintext password.

- Azure Active Directory
- Credentials

 --  1





Written by Lee Chagolla-Christensen

86 Followers · Writer for Posts By SpecterOps Team Members

Follow



<https://twitter.com/tifkin> <https://github.com/leechristensen>