SOAPHound — tool to collect Active Directory data via ADWS

Nikos Karouzos · Follow
Published in FalconForce · 18 min read · Jan 26, 2024

TL;DR

SOAPHound is a custom-developed .NET data collector tool which can be used to enumerate Active Directory environments via the Active Directory Web Services (ADWS) protocol.

SOAPHound can be used as an alternative to a number of open source security tools which are commonly used to extract Active Directory data via the LDAP protocol. SOAPHound can extract the same information without directly communicating to the LDAP server. Instead, LDAP queries are wrapped within a series of SOAP messages, which are sent to the ADWS server using a NetTCPBinding communication channel. Next, the ADWS server unwraps the LDAP queries and forwards them to the LDAP server running on the same domain controller. As a result, LDAP traffic is not sent via the wire and therefore would not be easily detected by common monitoring tools.

Of course, this blog also contains some custom detections to alert you in case of SOAPHound(-like) behavior in your environment. ;-)

Link to SOAPHound tool:

- https://github.com/FalconForceTeam/SOAPHound

Link to detections:

- https://github.com/FalconForceTeam/FalconFriday/blob/master/Discovery/ADWS_Connection_from_Unexpected_Binary-Win.md
- https://github.com/FalconForceTeam/FalconFriday/blob/master/Discovery/ADWS_Connection_from_Process_Injection_Target-Win.md

Fun fact *— Originally, we started researching the ADWS endpoint as a potential target for NTLM relaying attacks. Although such an attack is not possible due to the default configuration of ADWS servers (which enforce encryption and signing), we obtained some knowledge about the underlying protocol which led to the creation of this tool.*

. . .

Introduction

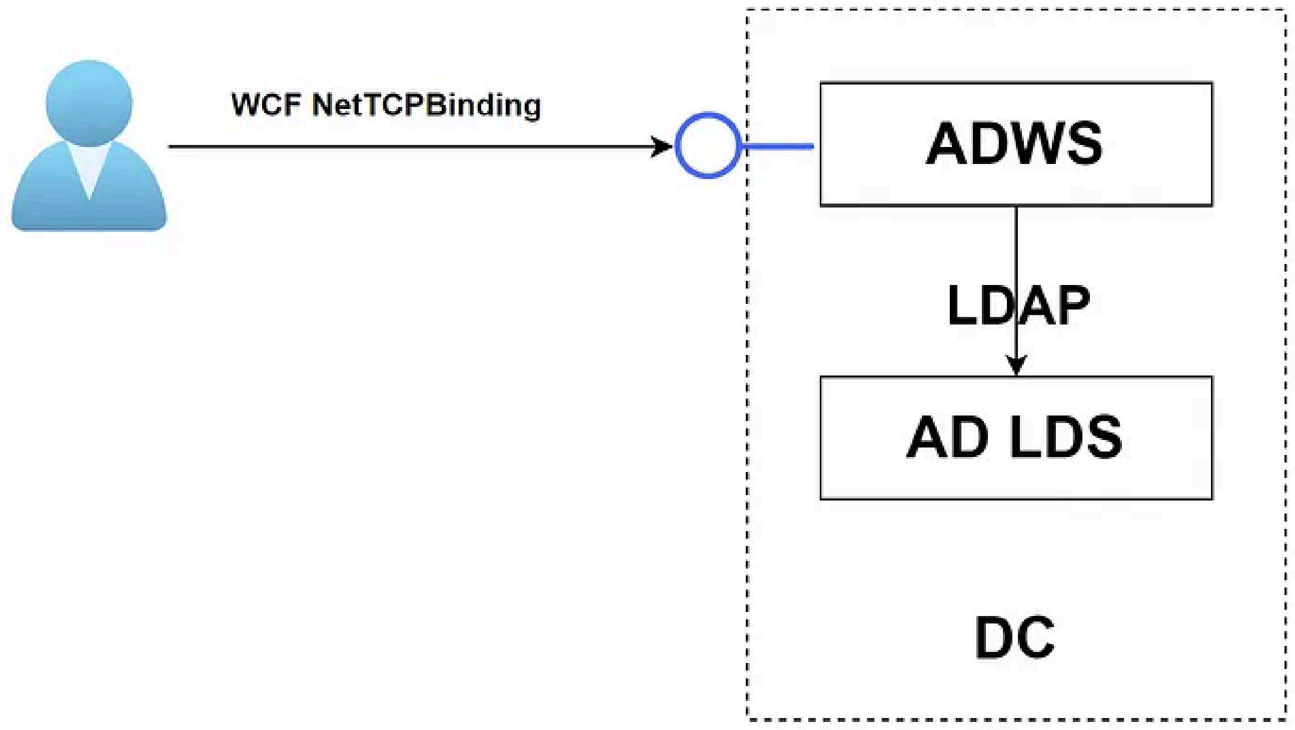
Adversaries commonly use the LDAP protocol to perform Active Directory (AD) enumeration via Active Directory Lightweight Directory Services (AD LDS). They extract information on the AD schema, such as domain users, devices, groups and their underlying access rights; providing a quick overview of the organization and significantly helping in the identification of potential attack paths. The AD schema also contains information about specific services of interest, such as Active Directory Certificate Services (ADCS), internal DNS records, etc. Please refer to the References section at the end of this blog for a list of open source tools capable of performing this type of reconnaissance.

Defensive tools are focused on detecting AD enumeration techniques by monitoring unexpected LDAP traffic and analyzing the exchanged information to identify uncommon and / or suspicious LDAP queries.

SOAPHound is an AD enumeration tool which does not send any direct LDAP traffic to stay under the radar of monitoring tools. Instead, it uses the Active Directory Web Services (ADWS) protocol.

ADWS protocol(s)

With Windows Server 2008 R2, Microsoft introduced an alternate protocol to retrieve AD objects: Active Directory Web Services (ADWS). ADWS provides a Web service interface to Active Directory Lightweight Directory Services (AD LDS) instances that are running on the same server. ADWS is used by legitimate tools, most notably the Active Directory module for Windows PowerShell, and provides a much faster interface to retrieve LDAP data.



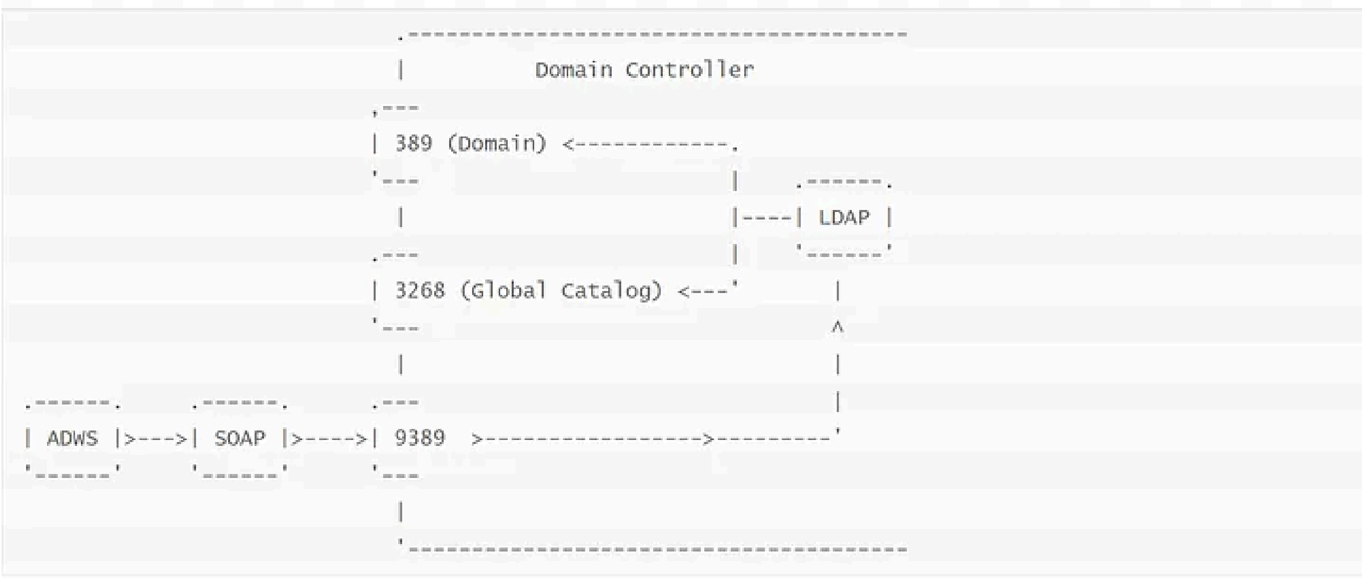


Image taken from https://tajdini.net/blog/forensics-and-security/pentest-windows-active-directory/

The ADWS server is listening on TCP port 9389 of domain controllers and facilitates SOAP-based search operations against directory services. It is compatible with LDAP filters, so it is possible to perform specific search queries and retrieve only the required properties. In fact, when ADWS is used, the DC performs LDAP requests internally to retrieve the results, in the context of the user sending the SOAP messages. For more information, please refer to the relevant Microsoft documentation.

ADWS clients (e.g., via the Active Directory PowerShell module) establish a NetTCPBinding session with the server, generating a run-time communication stack which uses transport security, TCP for message delivery, and a binary message encoding. This binding is an appropriate Windows Communication Foundation (WCF), system-provided choice for communicating with the ADWS server.

The ADWS protocol supports various protocols to send SOAP messages over TCP, such as:

- MS-WSTIM (WS Transfer Identity Management)
- WXFR (Web services transfer)
- WSENUM (Web services enumeration)
- MS-WSDS (WS Enumeration Directory Services)
- MS-ADDM (Data Model & Common Elements)
- MS-ADCAP (Custom Action Protocol)
- MS-NMFMB (.NET Message Framing MSMQ Binding)
- MS-WSMAN (Web services management protocol)

For the full list of Web Services protocols used by Active Directory systems, please refer to the MS-ADSO (Active Directory System Overview).

On network level, the SOAP messages are wrapped within the Microsoft.NET NegotiateStream Protocol Specification (MS-NNS) messages and sent over TCP streams. MS-NNS is used to establish the security context of the operations and MS-NNS in turn uses the Simple and Protected GSS-API Negotiation (SPNEGO) mechanism to determine which underlying security protocol to use. Eventually, and depending on the ADWS endpoint, SOAP messages are sent using different authentication mechanisms, as shown in the below table:

Development challenges

Understanding the protocols used by ADWS was a huge challenge on its own, since limited documentation was publicly available to help us develop a custom enumeration tool. However, the amazing work of Vincent Le Toux in the PingCastle project provided great insights on how to use ADWS to extract Active Directory data and helped us tremendously in both realizing the potential of the protocol, as well as developing the initial versions of SOAPHound.

Challenge #1 — Debugging

Our first challenge when playing around with the ADWS protocol was to debug the actual traffic sent to the ADWS server. Initially, we used the PowerShell AD module as our client, which is the most common tool using this protocol.

Running a simple Get-ADUser command within a Windows Domain context shows basic information of the requested user object:

Using Wireshark didn't provide much help, since it can only identify that the MS-NNS protocol is used, but all underlying traffic is encrypted.

This default applied encryption is also shown by reversing the ADWS server binary, showing that "EncryptAndSign" protection level is enforced by default:

However, debugging can be enabled on the server side by enabling WCF logging through the ADWS configuration file, located in "C:\Windows\ADWS\Microsoft.ActiveDirectory.WebServices.exe.config" on the domain controller, as detailed here.

The created log file produces WCF debug logs, which can be viewed with the SvcTraceViewer tool from Microsoft. To install the SvcTraceViewer tool, you need to download the Microsoft SDK. The tool can then be found in the path where you installed the SDK, for example C:\Program Files (x86)\Microsoft SDKs\Windows\v10.0A\bin\NETFX 4.8.1 Tools\SvcTraceViewer.exe

Challenge #2 — Creating service definition code from WSDL

The second challenge was to create the structure of the SOAP messages that are accepted by the ADWS server. The code definition files are available in WSDL format on the *mex* endpoint of the ADWS server, which is located in net.tcp://<dc>:9389/ActiveDirectoryWebServices/mex.

Visual Studio or Visual Studio Code can be used to create the service references via the WCF web service endpoint and create the basic code needed to send messages to the ADWS SOAP endpoints. However, the service references do not define the exact structure of these messages.

Obtain WSDL files in Visual Studio by creating a service reference from the 'mex' endpoint of the ADWS server.

Based on the ADWS debug log files, we observed the correct structure of the XML messages. These XML messages heavily use 'XML namespacing', which makes it a bit tricky to produce XML messages in the same structure. After some back and forth with ChatGPT, we generated C# code that could produce XML messages in the required format.

The key point is that multiple requests are needed to run an LDAP query via ADWS. First, an enumeration request is sent that contains the query to run and which properties to return. The response contains an 'EnumerationContext' which can be used to pull the actual results.

We parsed only the fields required for ingestion from the results using LINQ. Most properties are simple strings that can be easily parsed. Some are more complex and are encoded as a list of strings; for example, group members, byte arrays, etc. All data was parsed into a generic class called ADObject that has the appropriate definitions.

Challenge #3 — nTSecurityDescriptor attribute

The nTSecurityDescriptor is one of the most crucial LDAP attributes to enumerate, since it contains the Access Control Entries (ACEs), owner information, and inheritance assigned to the object. If parsed properly, it can lead to the identification of attack paths within an Active Directory environment (thanks to Bloodhound).

The nTSecurityDescriptor is a binary data structure of changeable length that contains security information associated with an object using the Security Descriptor Definition Language (SDDL) format.

An example nTSecurityDescriptor field of a Computer object.

Initially, our attempts to retrieve the nTSecurityDescriptor attributes of objects failed due to permission errors. The reason was explained in this blog post. The nTSecurityDescriptor attribute contains 4 separate pieces of information: DACL (Discretionary ACL), SACL (System ACL), Owner, and Primary Group. When you query Active Directory it will attempt by default to retrieve all parts of the security descriptor. However, your account may not have access to all parts (most notably, the SACL). When this happens, AD decides to simply not return anything. To get around the limitation above and still query the nTSecurityDescriptor, you need to use an LDAP control to specify you do not want the SACL. The control is LDAP_SERVER_SD_FLAGS_OID.

As a result, we added the above control in our EnumerationContext requests, which resulted in proper retrieval of nTSecurityDescriptor attributes via ADWS.

However, another challenge emerged: each ACE included in the nTSecurityDescriptor contains the below 6 flags:

- ACE type (allow/deny/audit)
- ACE flags (inheritance and audit settings)

- Permissions (list of incremental permissions)
- **ObjectType (GUID)**
- **Inherited Object Type (GUID)**
- **Trustee (SID)**

To properly parse the ACEs and define the trust relationships between objects, we needed to know the SID and ObjectType of the Tustee object. This information is not included in the EnumerationContext of our target object. This issue concerns the on-the-fly processing of ACEs to define trust relationships and is not really related to ADWS. As a result, we went through the SharpHound source code to understand how it handles this issue. We saw that SharpHound performs additional LDAP requests for every trustee object that is unknown at the time of processing and adds this information in a cache file, which contains mapping of SIDs and ObjectTypes. However, this approach results in sending a separate LDAP request for every unknown object and thus generating a lot of (unnecessary) traffic.

SOAPHound uses a different approach to generate the cache file containing the mapping of SIDs and ObjectTypes, which are required for processing ACEs. Instead of sending separate LDAP requests for every object that is found on the fly, it creates the full cache of all objects as an initial step and saves it to a file. This file is loaded and re-used during the data collection phase, so that the trust relationships are directly created during the retrieval of the EnumerationContext of objects. This cache is created with a single LDAP query, which is run before collecting the actual data and retrieves the "ObjectSID", "ObjectGUID", "DistinguishedName" and "ObjectClass" attributes of all objects within the Active Directory. Refer to the " — buildcache" mode of the SOAPHound tool below.

Challenge #4 — Optimize size and stealthiness

Another point of interest was to stay under the radar of monitoring tools. Not only regarding the network traffic (which was more or less sorted by design, by using ADWS protocol), but also in the actual LDAP queries being sent by the tool. We did not want to trigger detections by using known LDAP enumeration queries for different types of objects, such as Users, Computers, etc. Instead, being inspired by ADExplorer's snapshot capability, we implemented SOAPHound collection methods as follows. We first retrieve all objects of Active Directory and then process the different types (i.e., User, Computer, Group, Domain, GPO, Container). This allows us to send a minimum number of LDAP queries and retrieve all information that we need to create the BloodHound output.

When creating a snapshot with ADExplorer, you retrieve all attributes of all AD objects, most of which are not really needed for Bloodhound processing.

This is thus unnecessary traffic, which makes the whole process significantly slower (especially in large domains). With SOAPHound we only collect the LDAP attributes that we need (+/- 35 attributes) and therefore decrease the data transfer volume by more than 50%, compared to the ADExplorer snapshot.

Challenge #5 — Timeout issues

At this point, we had created a working and stable version of SOAPHound, which has been thoroughly tested in our lab. However, when we tried to actually run it in a client Active Directory environment, we ran into a timeout issue. More specifically, our LDAP request to retrieve all AD objects was failing after 30 minutes, which is the default expiration interval for EnumerationContext requests. After some additional research, we realized that the 30 minute timeout was set by default by the ADWS server and we could not renew the EnumerationContext at the client side without losing the previously obtained data. The only solution was to save the data that we have already obtained and reconnect with a new EnumerationContext request.

As a result, we came up with an idea to split the retrieval of AD objects based on the first character of each object. For example, we get all objects starting with "a" by sending the LDAP query: (cn=a*), then (cn=b*), and so on. The split functionality worked quite well in a couple of relatively small clients. However, it failed in a larger environment, where too many objects started with the same letter. As a result, SOAPHound's AutoSplit mode was implemented, where we check if a certain letter has more objects than a defined threshold. If yes, we split twice for that letter, by sending LDAP queries such as (cn=aa*), (cn=ab*) and so on. Refer to the --*autosplit* and --*threshold* options of SOAPHound tool below.

Functionalities of the SOAPHound tool

Below we summarize the main functionalities of SOAPHound.

> *SOAPHound output files are compatible with BloodHound version 4.*

Authentication options

SOAPHound supports the following authentication methods:

- Using the existing authentication token of the current user. This is the default option, if no username and password are supplied.
- Supplying a username and password on the command-line (--*user* and --*password* arguments, respectively).

Connection options

When SOAPHound runs in a domain-joined machine, it will automatically attempt to connect to the Domain Controller of the domain the machine is joined to. This can be overridden by supplying the --*dc* and --*domain* command-line arguments.

Collection methods

One of the following collection methods must be specified:

```
--buildcache: Only build cache and not perform further actions
--bhdump: Dump BloodHound data
--certdump: Dump AD Certificate Services (ADCS) data
--dnsdump: Dump AD Integrated DNS data
```

Method 1: building the cache (--buildcache)

SOAPHound can generate a cache file that contains basic information about all domain objects, such as Security Identifier (SID), Distinguished Name (DN), and ObjectClass. This cache file is required for BloodHound-related data collection (i.e., the --*bhdump* and --*certdump* collection methods), since it is used when crafting the trust relationships between objects via the relevant Access Control Entries (ACEs).

An example command to build the cache file is:

```
SOAPHound.exe --buildcache -c c:\temp\cache.txt
```

This will generate a cache file in the *c:\temp* folder. The cache file is a JSON-formatted mapping of basic information about all domain objects.

LDAP queries being sent:

```
LDAP base: defaultNamingContext of domain
LDAP filter: "(!soaphound=*)"
LDAP properties: "objectSid", "objectGUID", "distinguishedName"
```

To view some statistics about the cache file (i.e., the number of domain objects starting with each letter), you can use the --*showstats* command-line argument:

```
SOAPHound.exe --showstats -c c:\temp\cache.txt
```

Showstats runs locally and does not send any network traffic.

Method 2: collecting BloodHound Data (--bhdump)

After the cache file has been generated, you can use the --*bhdump* collection method to collect data from the domain that can be imported into BloodHound.

An example command to collect BloodHound data is below. Do note that this references the cache file generated in the previous step:

```
SOAPHound.exe -c c:\temp\cache.txt --bhdump -o c:\temp\bloodhound-output
```

LDAP queries being sent:

```
LDAP base: defaultNamingContext of domain
LDAP filter: "(!soaphound=*)"
LDAP properties: "name", "sAMAccountName", "cn", "dNSHostName", "objectSid", "object
```

If the targeted domain does not use LAPS, you can use the --*nolaps* command-line argument to skip the LAPS-related data collection.

This command will generate the *c:\temp\bloodhound-output* folder and produces a number of JSON files that can be imported into BloodHound.

The JSON files contain the collected Users, Groups, Computers, Domains, GPOs and Containers, including their relationships.

LDAP queries being sent:

```
LDAP base: defaultNamingContext of domain
LDAP filter: "(!soaphound=*)"
LDAP properties: "name", "sAMAccountName", "cn", "dNSHostName", "objectSid", "object
```

Dealing with large domains:

If you are dealing with a large domain, you may run into issues with the amount of data that can be retrieved in a single request. To deal with this, SOAPHound supports the --*autosplit* and --*threshold* command-line arguments.

The --*autosplit* command-line argument enables the AutoSplit mode, which will automatically split object retrieval on two depth levels, based on a defined threshold.

The --*threshold* command-line argument defines the split threshold, based on the number of objects per starting letter.

An example command to collect BloodHound data in AutoSplit mode is:

```
SOAPHound.exe -c c:\temp\cache.txt --bhdump -o c:\temp\bloodhound-output
--autosplit --threshold 1000
```

This will generate the output in batches of a maximum of 1000 objects per starting letter. If there are more than 1000 objects for a single starting letter, SOAPHound will use two depth levels to retrieve the objects. This will result in larger number of queries, each one returning a maximum of 1000 objects.

For example, if there are 2000 objects starting with the letter `a`, SOAPHound will retrieve objects starting with `aa`, `ab`, `ac`, etc., each in a separate query to avoid timeouts.

LDAP queries being sent:

```
Retrieval of Domain objects (Domains do not have a CN)
LDAP filter: "(ms-DS-MachineAccountQuota=*)";

Retrieval of Trusted Domain objects
LDAP filter: "(trustType=*)"

Retrieval of letter with entries less than threshold (example)
LDAP filter: "(cn=a*)"

Retrieval of letter with entries more than threshold (example)
LDAP filter: "(cn=ba*)

Retrieval of non alphanumeric objects
LDAP filter: "(&(cn=*)(!(cn=a*))(!(cn=b*))(!(cn=c*))(!(cn=d*))(!(cn=e*))(!(cn=f*))(!
```

Method 3: collecting ADCS Data (--certdump)

After the cache file has been generated, you can use the --*certdump* collection method to collect ADCS data from the domain that can be imported into BloodHound. ADCS output data is classified as GPOs in BloodHound.

This collection method does not support the --*autosplit* and --*threshold* command-line arguments. An example command to collect ADCS data is below. Note that this references the cache file generated in a previous step.

```
SOAPHound.exe -c c:\temp\cache.txt --certdump -o c:\temp\bloodhound-output
```

This command will generate the *c:\temp\bloodhound-output* folder and produce two JSON files that can be imported into BloodHound, containing information about the Certificate Authorities (CA) and Certificate Templates.

LDAP queries being sent:

```
Create cache of Certificate templates
LDAP Base: "CN=Configuration,DC=domain,DC=com"
LDAP Filter: "(!soaphound=*)";
LDAP properties: "name", "certificateTemplates"

Dump ADCS data
LDAP Base: "CN=Configuration,DC=domain,DC=com"
LDAP Filter: "(!soaphound=*)";
LDAP properties: "name", "displayName", "nTSecurityDescriptor", "objectGUID", "dNSHo
```

Method 4: collecting AD-integrated DNS data (--dnsdump)

Besides BloodHound data, SOAPHound can also be used to collect AD-integrated DNS data. This does not require a cache file and does not support the --*autosplit* and --*threshold* command-line arguments.

An example command to collect AD Integrated DNS data is:

```
SOAPHound.exe --dnsdump -o c:\temp\dns-output
```

This command will generate a file DNS.txt in the *c:\temp\dns-output* folder that contains a dump of all the AD-integrated DNS data.

LDAP queries being sent:

```
LDAP base: "CN=MicrosoftDNS,DC=DomainDnsZones,DC=domain,DC=com"
LDAP filter: "(&(ObjectClass=dnsNode))";
LDAP properties: "Name", "dnsRecord"
```

. . .

Detecting SOAPHound

At FalconForce, we always strive to release detections along with any offensive security content that we create. There are a number of ways in which information gathering attempts using SOAPHound can be detected.

Revisiting previous FalconFriday detections in the context of ADWS

We previously released a FalconFriday blog post about detecting Active Directory data collection in 2021: https://falconforce.nl/falconfriday-detecting-active-directory-data-collection-0xff21/.

This 2021 blog post described three methods to detect the collection:

1. Client-side LDAP query logging using Microsoft Defender for Endpoint.
2. Domain controller LDAP query logging via Microsoft Defender for Identity.
3. Domain controller object access logging via SACLs and audit policies.

In this section we will evaluate how well these detections work in the context of ADWS.

1. Client-side LDAP query logging using Microsoft Defender for Endpoint

The first method (using Client-side LDAP query logging collected be Microsoft Defender for Endpoint (MDE)) turns out not to be very effective against ADWS based collection.

To understand why this method is less effective we have to take the 'capping' mechanism of MDE into consideration. This capping mechanism ensures that the amount of telemetry collected per machine remains within reasonable limits. For example, if a machine generates millions of LDAP queries per day, this would lead to a very large amount of telemetry being collected and stored.

We discovered that for LDAP queries the following capping is applied by the MDE agent:

- Within a 24 hour period only one entry is logged if the SearchFilter, DistinguishedName and Initiating process are the same.
- A maximum of 1000 LDAP queries are logged per machine per day, regardless of the source process.

Based on our own testing, the 1000 LDAP queries per day per machine can be reached within 10 minutes on a typical domain controller in an enterprise environment. This means that there are massive blind-spots in the LDAP queries performed via the ADWS service, since this will typically be deployed on domain controllers and the ADWS process on the domain controller is responsible for making the LDAP queries.

Another downside of this detection method is that if the query is logged there is nothing in the telemetry linking it to the user or the device performing the ADWS collection. Making it hard to investigate any alert triggered.

2. Domain controller LDAP query logging via Microsoft Defender for Identity

The second method used in our original blog post relies on the telemetry collected from the AD server by the Microsoft Defender for Identity (MDI) agent. The downside of this detection method is that it only records LDAP queries that it finds suspicious. It is possible for an attacker to modify the exact LDAP queries used; to make them appear as benign. For example, by avoiding the usage of *ObjectClass=**. Since most queries performed by SOAPHound at this time are not considered to be suspicious by MDI, it is hard to use this method to detect collection. Unfortunately, the identification of which queries are considered to be suspicious in MDI cannot be configured by using a method similar to custom detections.

3. Domain controller object access logging via SACLs and audit policies

The third detection from the original blog is based on the number of AD objects accessed by a user compared to a baseline of how many objects are typically accessed. **In our testing, this method works regardless of the exact collection method being used.** For this detection the usage of ADWS does not make a difference when compared to LDAP. However, the existing caveats for this detection remain: it requires a custom SACL to be configured to enable logging of all directory object access, which will result in a large volume of logs being generated.

Building detections specific to ADWS-based data collection

This leaves us with detections that specifically target the collection using ADWS. We identified two methods of doing this.

The first is by looking at the source process for connections to ADWS, based on the target port that is used: 9389. To reduce false positives for other traffic over this port that is not aimed at ADWS, we can filter this traffic to only include IPs where the *microsoft.activedirectory.webservices.exe* binary is running. This only works when the target machine is also enrolled in MDE.

In a typical environment there are only a handful of programs that are making connections to the ADWS service:

- Active Directory Administrative Center (dsac.exe)

- Microsoft Monitoring Agent
- PowerShell

Based on this, we can create a detection that will trigger when a process outside of this list makes a connection to ADWS.

This first query is available on our Github account: https://github.com/FalconForceTeam/FalconFriday/blob/master/Discovery/ADWS_Connection_from_Unexpected_Binary-Win.md

The first detection has an issue where an attacker can inject malicious code into one of the allowed processes. To detect this behavior, we can create a second rule that looks at the combination of a process injection targeting a process and that process subsequently making an ADWS connection.

This second query is available on our Github account. https://github.com/FalconForceTeam/FalconFriday/blob/master/Discovery/ADWS_Connection_from_Process_Injection_Target-Win.md

. . .

Conclusion

Information gathering methods in Active Directory were already challenging to detect. The release of this new tool that targets ADWS instead of directly targeting LDAP further increases the need for custom detections in this area. The existing method based on the number of AD Objects being accessed by a user is still the most reliable way to detect data collection, but it does require a custom SACL to be configured and has a high log volume. Existing telemetry can be used to specifically detect the ADWS-based collection, by looking at the source processes making ADWS connections.

References

- SharpHound (https://github.com/BloodHoundAD/SharpHound/tree/dev)
- PingCastle (https://github.com/vletoux/pingcastle)
- StandIn (https://github.com/FuzzySecurity/StandIn)
- Certify (https://github.com/GhostPack/Certify)
- ADExplorerSnapshot.py (https://github.com/c3c/ADExplorerSnapshot.py)
- Adidnsdump (https://github.com/dirkjanm/adidnsdump)
- Certipy (https://github.com/ly4k/Certipy)
- BloodHound.py (https://github.com/dirkjanm/BloodHound.py)

Security Red Teaming Active Directory Bloodhound

--

Written by Nikos Karouzos

21 Followers · Writer for FalconForce

Follow