Steve Syfuhs **@SteveSyfuhs** *Tue Sep 15 18:21:48 +0000 2020*

Okay folks, let's talk Kerberos. Strap in. It'll be a long one. **https://t.co/VE4VmLQfdD**

At it's core Kerberos is an authenticated key agreement protocol based on the Needham-Schroeder protocol.

We'll keep the crypto to a minimum because I'm kind of dumb, but here it is in it's most basic form. Wikipedia has everything you need on the crypto aspects of it. **https://t.co/gHYSJ9Spa2 https://t.co/DkhTJf2hMj**

$$A \rightarrow S : A, B, N_A$$

Alice sends a message to the server identifyir

$$S \rightarrow A : \{N_A, K_{AB}, B, \{K_{AB}, A\}_{K_{BS}}\}_{K_{AS}}$$

The server generates K_{AB} and sends back t
this key with.

$$A \rightarrow B : \{K_{AB}, A\}_{K_{BS}}$$

Alice forwards the key to Bob who can decryp

$$B \rightarrow A : \{N_B\}_{K_{AB}}$$

Bob sends Alice a nonce encrypted under K_{A}

$$A \rightarrow B : \{N_B - 1\}_{K_{AB}}$$

Anyway, the point of this key agreement protocol is so two different parties, A and B, can prove to one another that they have knowledge of the same key without leaking the key to the other party, all the while proving each party is who they say they are.

Why is this the basis for authentication? Because we don't actually care about how a user authenticated (I mean, we do, but...). What we really care about is the application communicating between client A and server B is doing so securely. This is done by using that key.

Remember, we're talking about a time long before SSL and TLS were invented, so this key is *the thing* securing all communications.

It turns out this agreement thing is incredibly difficult to do securely with just two parties (at scale, at least). So we introduce a third party C. Both A and B trust C, so C can act as a go-between for both parties.

If client A can authenticate to C, C can prove to B that A is actually A. Then, C can prove to A that B is actually B. This way A and B really don't need to know anything about each other, other than C says its all good. Let's build this into a protocol: Kerberos.

Okay, so we have our three parties: The client (user/human), the application (say SMB share), and the trusted third party (KDC).

The client knows it needs to talk to the application, so it begins by speaking to the KDC. The client and KDC don't implicitly trust each other, so they need to prove to one another that they are who they say they are. This is done by doing an authentication request (AS-REQ).

An AS-REQ is a message sent to the authentication service (AS). All the AS does is exchange credentials for tickets. These credentials can be anything, but are often passwords. **https://t.co/HAJ7T6ZHtl**

```
Kerberos
  Record Mark: 205 bytes
  as-req
      pvno: 5
      msg-type: krb-as-req (10)
    padata: 1 item
    req-body
        Padding: 0
      kdc-options: 40810010
      cname
          name-type: kRB5-NT-ENTERPRISE-PRINCIPAL (10)
        cname-string: 1 item
            CNameString: jack@threadabort.net
        realm: THREADABORT.NET
      sname
        till: 2020-09-16 18:29:12 (UTC)
        rtime: 2037-09-13 02:48:05 (UTC)
        nonce: 3863631250
      etype: 3 items
```

You'll notice there's nothing in there that includes any password information. That's because of the key agreement thing -- don't leak the key. The KDC happens to know the password, so generates a response (AS-REP).

The AS-REP includes two things: an encrypted ticket, and an encrypted client blob. The encrypted client blob is encrypted using the user password. The KDC has now proven to the client it is the KDC because only the KDC knows the password.

Within the blob is metadata about that other (double) encrypted thing-- the ticket. One other part the metadata is this thing called the session key. The session key encrypts the doubly-encrypted ticket. **https://t.co/xYjbJUmcAs**

```
EncKDCRepPart   ::= SEQUENCE {
        key             [0] EncryptionKey,
        last-req        [1] LastReq,
        nonce           [2] UInt32,
        key-expiration  [3] KerberosTime OPTIONAL,
        flags           [4] TicketFlags,
        authtime        [5] KerberosTime,
        starttime       [6] KerberosTime OPTIONAL,
        endtime         [7] KerberosTime,
        renew-till      [8] KerberosTime OPTIONAL,
        srealm          [9] Realm,
        sname           [10] PrincipalName,
        caddr           [11] HostAddresses OPTIONAL
}
```

The client decrypts this ticket and gets, well, not a lot -- another encrypted blob. That's okay though because the client has everything it needs to set up a session with application B now. This might be confusing because I'm leaving a flow out, but we'll get there. Trust me. **https://t.co/8Iu8sg1rpA**

```
Ticket          ::= [APPLICATION 1] SEQUENCE {
        tkt-vno         [0] INTEGER (5),
        realm           [1] Realm,
        sname           [2] PrincipalName,
        enc-part        [3] EncryptedData
}
```

This opaque blob of a ticket was encrypted by the KDC to the target applications long term key -- it's password. The application knows the password so it can decrypt it, and since the KDC knew the password too, the application knows the KDC is the KDC.

Within the blob is a whole bunch of metadata, in fact the same metadata in the client blob. Including that same key. **https://t.co/1ZLUCVFkR6**

```
EncTicketPart   ::= [APPLICATION 3] SEQUENCE {
        flags                   [0] TicketFlags,
        key                     [1] EncryptionKey,
        crealm                  [2] Realm,
        cname                   [3] PrincipalName,
        transited               [4] TransitedEncoding,
        authtime                [5] KerberosTime,
        starttime               [6] KerberosTime OPTIONAL,
        endtime                 [7] KerberosTime,
        renew-till              [8] KerberosTime OPTIONAL,
        caddr                   [9] HostAddresses OPTIONAL,
        authorization-data      [10] AuthorizationData OPTIONAL
}
```

Aha. So now client and the application know a key only they know (well, the KDC too, hold that thought though). This means the client and application can talk to each other securely! Woohoo!

Hold up you say! I'm missing a huge important piece -- the TGS-REQ. Yep, lets talk about that.

Remember how I said the only job of the AS-REQ is to exchange credentials for tickets? Well it's true. You can say AS-REQ => (creds, 'host/some-app') and the KDC will happily oblige. It turns out this is incredibly inefficient and possibly insecure though.

Credentials are super secret. We don't want them around all the time, and we do some heavy crypto with those creds before we can encrypt stuff to them, so that could just be resource intensive. So what do?

What we do is an AS-REQ and we ask for a special service ticket to a special service called the Ticket-Granting-Service (TSG), i.e. krbtgt. This solves both credential problems. How does a TGS-REQ work?

The TGS-REQ is almost identical to the AS-REQ. In fact the message structure is identical. The difference is that we include a special thing called pre-auth data, which is that encrypted ticket (plus goo, we'll get into it). **https://t.co/5Vu9XMEHla**

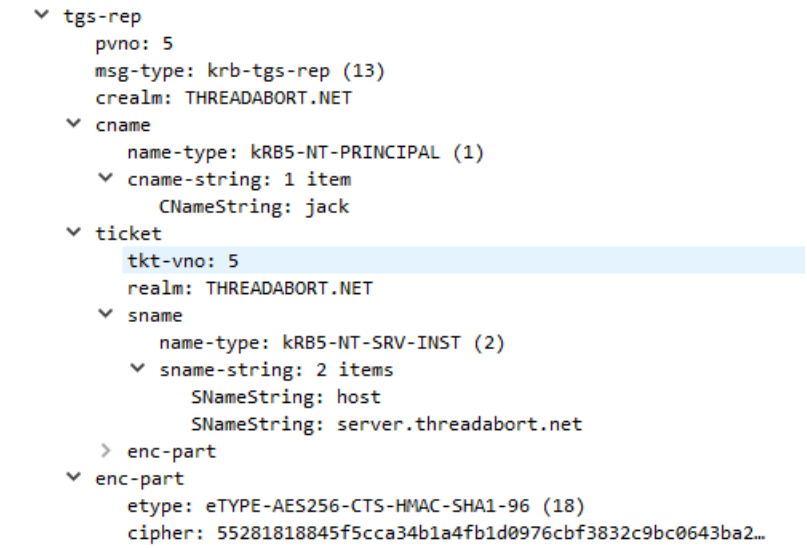
```
▼ tgs-req
    pvno: 5
    msg-type: krb-tgs-req (12)
  ▼ padata: 2 items
    ▼ PA-DATA PA-TGS-REQ
      ▼ padata-type: kRB5-PADATA-TGS-REQ (1)
        ▼ padata-value: 6e82021430820210a003020105a10302010ea20703050000…
          ▼ ap-req
              pvno: 5
              msg-type: krb-ap-req (14)
              Padding: 0
            ▶ ap-options: 00000000
            ▼ ticket
                tkt-vno: 5
                realm: THREADABORT.NET
              ▼ sname
                  name-type: kRB5-NT-SRV-INST (2)
                ▼ sname-string: 2 items
                    SNameString: krbtgt
                    SNameString: THREADABORT.NET
              ▶ enc-part
            ▶ authenticator
    ▶ PA-DATA PA-PAC-OPTIONS
  ▼ req-body
      Padding: 0
    ▶ kdc-options: 40810010
      realm: THREADABORT.NET
    ▼ sname
        name-type: kRB5-NT-SRV-INST (2)
      ▼ sname-string: 2 items
          SNameString: host
          SNameString: server.threadabort.net
      till: 2037-09-13 02:48:05 (UTC)
      nonce: 3843847656
    ▶ etype: 3 items
```

The TGS is almost always the same server as the AS, it just has a logically different purpose. The REQ message contains the name of the requested service (host/service.threadabort.net) in this case, plus the ticket granting ticket in the

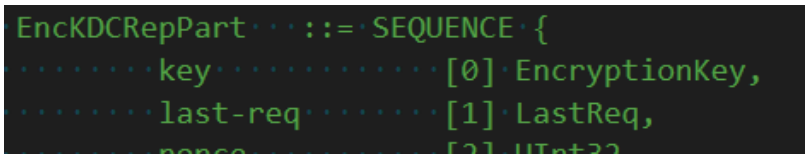
preauth-data.

The TGS receives this message and first checks the preauth-data, extracting that TGT. The TGT is encrypted to the krbtgt long term key -- it's password. Only the KDC knows the krbtgt password, so it knows its genuine and came from itself previously.

Once decrypted we now have that session key that only the client (and now TGS) knows. The KDC generates a response (REP) and instead of encrypting the client metadata to the client password, it encrypts it to the session key. **https://t.co/Y1WlJASScm**



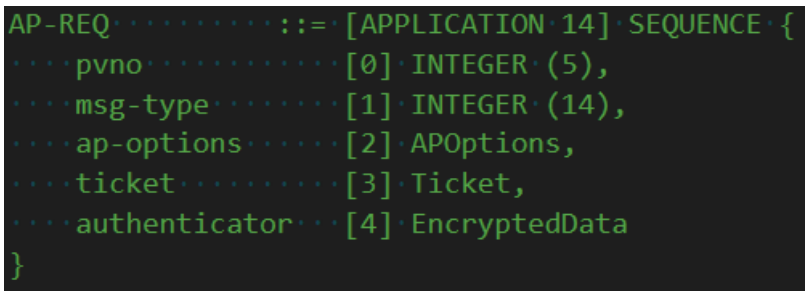
And so we're back to the client. The client receives the REP, decrypts the client metadata blob, and extracts the key. The key is used to decrypt the doubly-encrypted ticket blob, and voila the client now has a ticket and a key it can send to the application. **https://t.co/QywE636QoM**



But we glossed over the sending-to-the-application part. It turns out this is partially undefined, that way the application protocol (say SMB) can include the ticket as they need it. Mostly, we have some housekeeping first though.

That housekeeping is the act of converting the encrypted ticket blob into an application request (AP-REQ). Why do this, and not just send the ticket? Well, the AP-REQ serves a few purposes. First, it provides a way to prevent message replay, and includes a way to switch up keys.

An AP-REQ is made up of two things: a ticket and an authenticator. We already know the ticket -- it's encrypted to the application long term key, and contains a special session key. The authenticator is special however. **https://t.co/uVQnDbvwfS**



The authenticator is encrypted with the ticket session key. The application should decrypt it because it includes additional metadata, like a sequence number. **https://t.co/BtaBN6yIJC**

```
Authenticator   ::= [APPLICATION 2] SEQUENCE  {
        authenticator-vno       [0] INTEGER (5),
        crealm                  [1] Realm,
        cname                   [2] PrincipalName,
        cksum                   [3] Checksum OPTIONAL,
        cusec                   [4] Microseconds,
        ctime                   [5] KerberosTime,
        subkey                  [6] EncryptionKey OPTIONAL,
        seq-number              [7] UInt32 OPTIONAL,
        authorization-data      [8] AuthorizationData OPTIONAL
}
```

This sequence number can be tracked by the application. If it sees the same number more than once it can treat it as a replay and kill the second attempt. See, each time a client kicks off a request it must generate a new AP-REQ, that way the application can prevent replay.

The other important bit of information in this authenticator is a sub-session key. This key is a special key that only the client (and now the application) know, so you can guarantee only the two parties know it without the KDC knowing it. Neat.

The application may or may not decide it wants to switch to the sub-session key. Regardless of that, it still needs to do one final thing: respond to the client with an AP-REP.

The AP-REP is mostly just an ACK. It contains a blob encrypted to the (sub-)session key, and this tells the client that the application is really who they say they are because it could decrypt the ticket issued by the KDC, and therefore the authenticator. **https://t.co/i9imq3aJIq**

```
AP-REP          ::= [APPLICATION 15] SEQUENCE {
        pvno            [0] INTEGER (5),
        msg-type        [1] INTEGER (15),
        enc-part        [2] EncryptedData
}

EncAPRepPart    ::= [APPLICATION 27] SEQUENCE {
        ctime           [0] KerberosTime,
        cusec           [1] Microseconds,
        subkey          [2] EncryptionKey OPTIONAL,
        seq-number      [3] UInt32 OPTIONAL
}
```

But then it has one last nugget: yet another sub-session key. It turns out the application may decide it wants to use a different key for whatever reason it wants. Now any future communications between client and application are encrypted using keys that can be authenticated.

BUT WAIT THERES MORE!

I go into great detail about how this all fits into the Windows logon process in this other thread, but maybe we can go a bit deeper?**https://t.co/1M9SxP4kWj**

When a client, like Windows, decides it wants to do Kerberos it first needs find a KDC. There are a couple ways this can work.

The first is pretty straightforward: hardcode a list of KDCs. This is how many clients work. MIT Kerberos for instance supports this. **https://t.co/JsTqIMC7fm**

```
[realms]
    ATHENA.MIT.EDU = {
        kdc = kerberos.mit.edu
        kdc = kerberos-1.mit.edu
        kdc = kerberos-2.mit.edu
        admin_server = kerberos.mit.edu
        master_kdc = kerberos.mit.edu
    }
    EXAMPLE.COM = {
        kdc = kerberos.example.com
        kdc = kerberos-1.example.com
        admin_server = kerberos.example.com
    }
```

My Kerberos .NET (**https://t.co/kGpj5Gh6sh**) library also supports this using the same schema. **https://t.co/4S3mLoT4AB**

```
udp_preference_limit = 0
verify_ap_req_nofail = false
request_pac = true

[realms]
THREADABORT.NET = {
   disable_encrypted_timestamp = false
   kdc = 10.211.212.220
   pkinit_eku_checking = kpkdc
   pkinit_dh_min_bits = 2048
   pkinit_require_crl_checking = false
}
CORP.IDENTITYINTERVENTION.COM = {
   disable_encrypted_timestamp = false
   kdc = 10.211.212.203
   pkinit_eku_checking = kpkdc
   pkinit_dh_min_bits = 2048
   pkinit_require_crl_checking = false
}

[domain_realm]

[capaths]

[appdefaults]

[logging]


bruce>
```

The other way is through DNS lookups. Many clients support this, but often as a secondary approach because DNS isn't entirely secure (at least it wasn't, at the time). This works by looking for an SRV record _kerberos._transport.realm.net. The results are KDCs. **https://t.co/Mz7GneYL71**

```
Queries
  _kerberos._tcp.THREADABORT.NET: type SRV, class IN
```

Windows prefers another approach entirely because hardcoding doesn't scale well when you have thousands of DCs, and DNS doesn't include enough detail. What it does is something called DC location using the DC locator service.

The DC locator works by first querying DNS for some LDAP SRV records. These list all (K)DCs in the domain, and the locator goes through each record it finds and makes a UDP LDAP call to the server.

The server either responds immediately, or the request times out quickly and moves on to the next record. The locator checks each record for KDC info and tries to find one that's close enough based on IP subnet and the registered AD site.

As it narrows the list down it then looks to make sure the DC supports the things it needs. There's a couple dozen options to choose from: **https://t.co/Hv8S29PRL4** and **https://t.co/J3gTI5UOiZ https://t.co/gCW5WR1N1p**

```
/// <summary>
/// Forces cached domain controller data to be ignored.
/// </summary>
DS_FORCE_REDISCOVERY = 1 << 0,

/// <summary>
/// Requires that the returned domain controller support directory services.
/// </summary>
DS_DIRECTORY_SERVICE_REQUIRED = 1 << 1,

/// <summary>
/// Attempts to find a domain controller that supports directory service functions.
/// </summary>
DS_DIRECTORY_SERVICE_PREFERRED = 1 << 2,

/// <summary>
/// Requires that the returned domain controller be a global catalog server for
/// the forest of domains with this domain as the root.
/// </summary>
DS_GC_SERVER_REQUIRED = 1 << 3,

/// <summary>
/// Requires that the returned domain controller be the primary domain controller for the domain.
/// </summary>
DS_PDC_REQUIRED = 1 << 4,

/// <summary>
/// Requests that cached domain controller data should be used.
/// </summary>
DS_BACKGROUND_ONLY = 1 << 5,
```

Eventually it finds a DC that it likes and returns the address to the Kerberos stack.

Incidentally this functionality is why we can do amazing things with FAST, or Windows Hello, or FIDO and only require a handful of DCs running the latest bits instead of all of them. The client knows it needs a KDC that supports e.g. FIDO, so just it asks DC Locator for one.

However, suppose Windows isn't talking to an Active Directory domain. Maybe it's talking to an MIT or Heimdal KDC. They don't support DC Locator (well, probably don't?), so Windows will fall back to looking up the DNS SRV _kerberos records.

You'll notice I haven't said a thing about domain-joined vs workgroup vs AADJ vs whatever. This is by design because Windows doesn't give a crap about whether you're any of those (I mean, it does, but...). Windows is just another Kerberos client.

All Windows needs to make Kerberos work is have a user principal name, a credential, and a realm. It can guess about your realm based on your username. If I type jack@threadabort.net it's plausible my realm is **https://t.co/D5Z9dt5MPC**. At least, it's worth trying.

So Windows will DC locate threadabort .net: it'll jump down the LDAP rabbit hole, and then it'll move on the straight DNS, then it'll even try HTTP (we'll get to that in a bit).

Eventually it'll probably find KDC, and Windows will do the AS-REQ to get a krbtgt, then a TGS-REQ to the service in question. No domain-joined'ness involved.

So what does domain join do? First it configures your desktop authority. That means Windows will create a desktop and logon session for you if the configured domain gives you a ticket to itself. **https://t.co/SPlgx88YcL**

As it happens there's nothing inherently special about that last process, beyond knowing what knobs to turn. If you call all the correct APIs you too could create a logon session without being joined to that domain.

The second thing joining the domain does is it provides a bunch of hints to the Kerberos logon process. If my UPN is jack@threadbort.net, and my realm is **https://t.co/D5Z9dt5MPC**, Windows can reason through that. It can't reason through it if my realm is **https://t.co/1SNBhmSXOP**.

Domain join provides these hints to say that if it can't find the domain through its standard process, try the machine's configured domain. It's either the right one, or maybe it's a domain across a trust.

The third thing of course is kinda sorta related to the last one, but is more generically about policy: group policy. It's about management of the device from a centralized location.

And the inverse? An Active Directory realm with a Linux or Mac client? It pretty much works the same.

Have user, have cred, have realm.
Go find KDC for realm.
Do AS-REQ, TGS-REQ.
Connect to App.
Profit.

Where Windows differentiates itself here is the out-of-the-box capability to join the domain and be managed, and use the creds the user typed in to get to the desktop to make SSO magical behind the scenes.

Mac and Linux definitely have this capability, which can be configured through, say MDM. **https://t.co/zCXYB2jjKw** They kinda sorta work the same way. They start setting device defaults and optimistically using those values and creds.

But back to Windows. That covers domain join, which is starting to show its age. These days we have hybrid join, and AAD join. All of these do Kerberos to on-prem DCs.

In the hybrid case, you're still domain joined. Nothing special about that. AADJ is something else entirely though.

AADJ changes the desktop authority from your on-prem domain to Azure AD. It uses a different authentication provider (CloudAP instead of Kerberos). But it still does Kerberos with SSO. How?

Well, it's deceptively simple. Remember how I said Windows doesn't much care what domain state its in? If you give it a cred it'll just do Kerberos. AADJ works the same way. The difference is that when CloudAP gets a successful response back from AAD, it includes metadata.

That metadata includes useful information like your real realm name, and your fully qualified UPN. So even if I typed jack@threadabort.net, it'll return jack@corp.threadabort.net+**https://t.co/x7owvixTPJ**. CloudAP hands this off the Kerberos, says have it, and does an AS-REQ.

The difference here is that Windows now doesn't care if the AS-REQ succeeds. It either did, and now it has a TGT to do SSO to on-prem stuff, or it doesn't. Either way it's at the desktop and you still have your cloud creds for SSO to AAD.

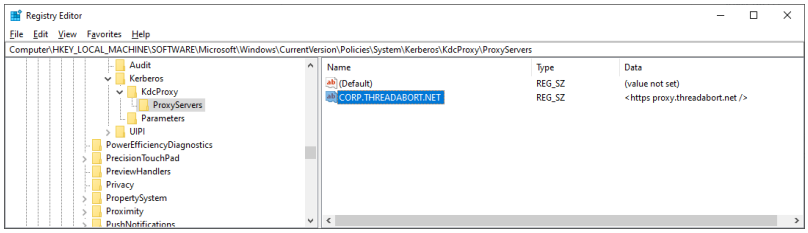
The move to the cloud puts you in an interesting predicament though. Half your stuff is in the cloud, and half your stuff is still on-prem. However, now you're out and about, or given the current state of things working from home because of a pandemic.

All the on-prem stuff isn't reeeeeally necessary for day-to-day work, so you don't set up a VPN and everything just kinda hums along while users access their email from the cloud using AAD on their hybrid joined device.

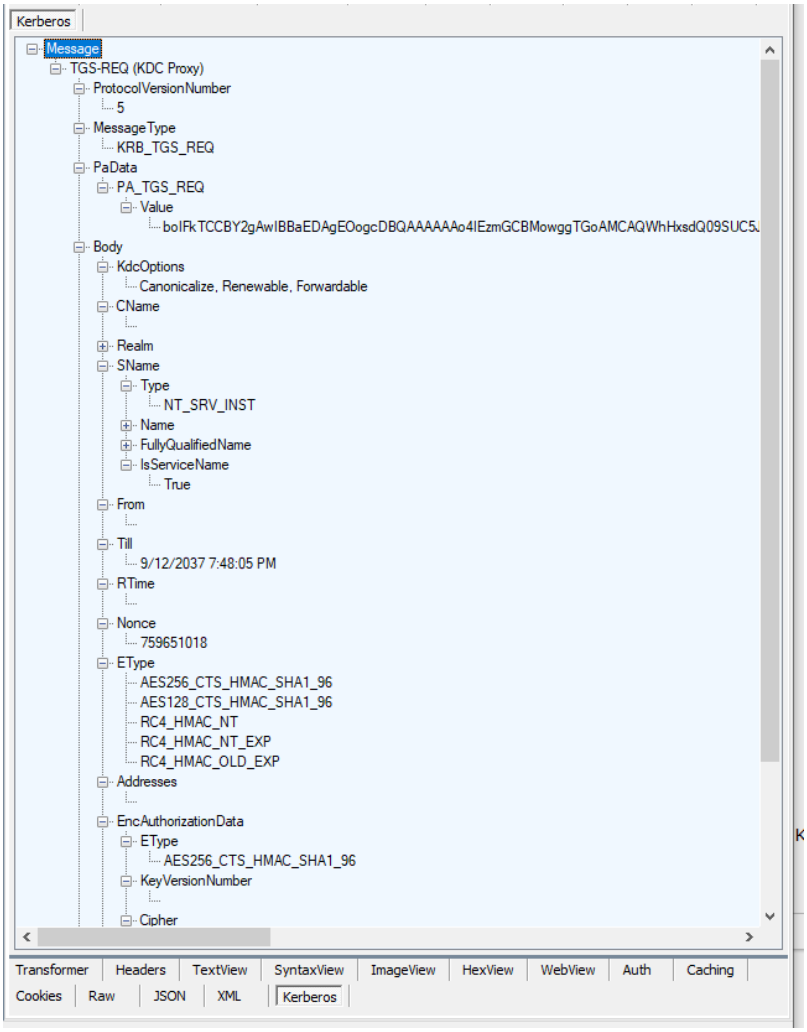
It's been 6 months and your password expired some time ago. You need to change it because stuff is starting to lock you out. The only way to change your password is through CTRL-ALT-DEL, but you need line of sight to a DC and you don't have a VPN ARRRRRGGG.

Sometime before 2012 Windows introduced this idea of KDC Proxy. The ability to tunnel KDC messages (AS/TGS/ChangePW) over an HTTP channel instead of requiring line of sight on port 88. **https://t.co/eTXUfafWCu**

If configured, Windows will natively use HTTP(S) to proxy KDC messages. It's pretty cool. The way it works is also pretty simple. Windows is configured with a couple registry keys that map a realm to a proxy server. **https://t.co/lf9Ykkhtbo**



Windows will attempt DC location and fail. It'll try DNS SRV _kerberos and fail. It'll check this registry location and go AHA! **https://t.co/znNe9558UZ**



The protocol is pretty simple. It's just the Kerberos binary format (ASN.1 DER encoded bytes) wrapped in a special KDC Proxy DER structure, and fired off in an HTTP POST.

This is somewhat less useful for AS and TGS because you're not connecting to internal resources, but there's another message flow: Change Password!

the Change password protocol is an extension of Kerberos. You start with an AS-REQ to the 'kadmin/changepw' service instead of krbtgt. This tells the KDC to ignore if the password is expired or required to change, because obviously we're gonna change it in a moment.

Then you send the service ticket to the changepw service on port 464. The message is an AP-REQ + an encrypted structure. The structure contains the new password, and is encrypted to the session key in the authenticator of the AP-REQ. See how it all ties back together? **https://t.co/5ZQ0IWjvwk**

```
ChangePasswdData ::=  SEQUENCE {
                      newpasswd[0]   OCTET STRING,
                      targname[1]    PrincipalName OPTIONAL,
                      targrealm[2]   Realm OPTIONAL
                      }
```

In any case, now with a configured Windows device you can do CTRL-ALT-DEL and change your password when you're not on the corporate network. Neat.

And that concludes the introductory section to Kerberos.

.

.

.

In our next session we'll talk crypto and passwords because someone asked and brain dumps like this are kinda fun.

Way back in the beginning of time I mentioned the KDC encrypts the AS-REP client bit to the client password. Doesn't this mean the KDC needs to know the password??

I mean, yeah. Kinda.

In actual fact, what the KDC really needs to know is what the derived key is. Kerberos doesn't just take your password and encrypt stuff to it. Passwords are too weak. They need to be run through key-derivation functions, which is just a fancy name for hashing a thousand times.

The derivation functions are complicated. For AES+Sha1 you take the password and

hash = PBKD2(password, 4096 iterations)
constant = dk(...)
ki = n-fold(constant)
key = AESCTS(ki, hash)

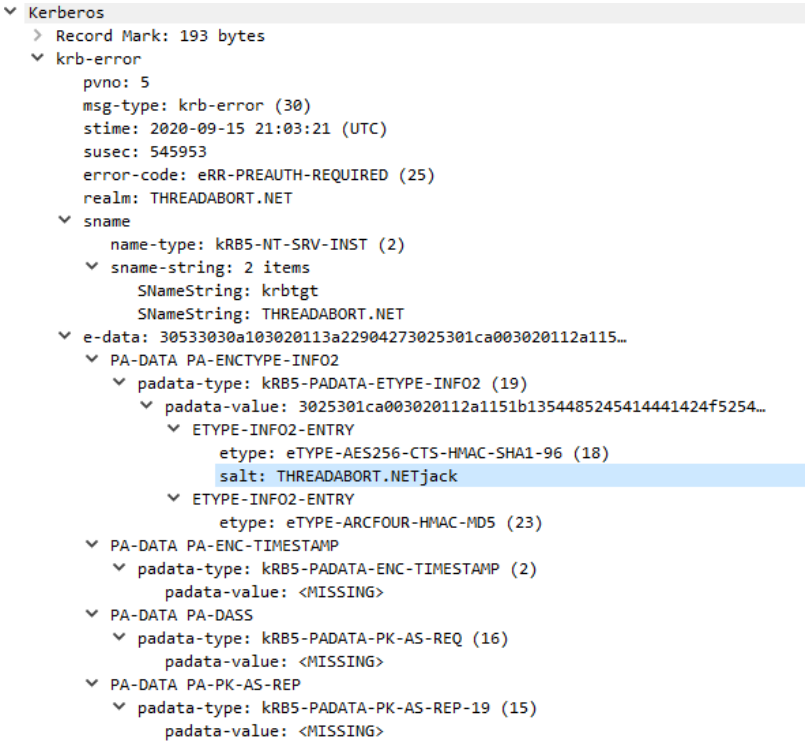
See here if you can stomach it: **https://t.co/WpylLUyNF2**

AES+Sha1 is the most common key mode out there today, but there are others. DES, 3DES, RC4, AES+Sha256, Camelia, etc. DES, 3DES, and RC4 are deprecated by standards bodies.

RC4 is still in use by Windows by default for backwards compatibility, but we're trying to make it go away. **https://t.co/uWpqj2honJ**

But back to the key thing. No, the KDC doesn't store the raw passwords. The KDC stores these derived keys. The client knows how to convert the password into these keys too, so when it needs to encrypt or decrypt something, it'll run through it with some info provided by the KDC.

That information, specifically the key salt, provided by the KDC is returned in an error to the original AS-REQ. **https://t.co/2MmbMVn0a9**



There's also another bit that I left out, which is this idea of pre-authentication. Despite the key derivation process described earlier, they're not perfect. With enough effort, or a weak enough password, you can crack them.

If you couple that with a KDC that'll return a blob encrypted to that derived key, you get a nice little crypto oracle. You can take that blob offline, crack it, and now you have a password. Oops.

We need to make that first step of requesting the encrypted blob a little harder.

This is where pre-auth data comes in. Client does AS-REQ, KDC responds with an error, including that salt data, and says "by the way, you need to prove yourself. I support XYZ ways." One of those ways is an encrypted timestamp.

It's kinda simple. Take the salt you were just given, join it to the password and derive. Then take the current timestamp and encrypt it. Stick it into that pre-auth data section of the AS-REQ and away you go.

Much like on the TGS side of things, the AS sees the PA data, decrypts it, checks the timestamp is within the last few minutes and then returns whatever ticket was requested.

And so we have a handful of these hardness makers, or preauth types. I talked about FAST in this other thread: **https://t.co/E2lysK1ala**

And I talked about PKINIT (certificates) and Windows Hello here: **https://t.co/hTdmdlOjWB**

And the Kerberos IETF working group is continuing to build out more options like SPAKE pre-auth **https://t.co/7dDJFXz24r**

Anyway, that's it. Here's Riley trying to follow along. **https://t.co/KVh2FvXayr**

And posted: **https://t.co/uNgeFphNoU**