Kerberos Wireshark Captures: A SPNEGO Example

medium.com/@robert.broeckelmann/kerberos-wireshark-captures-a-spnego-example-e22e6b1d662a

Robert Broeckelmann 16 августа 2018 г.

Robert Broeckelmann

This is the second post that presents a real world example of the use of Kerberos. The first post captured the Kerberos protocol details of a Windows domain user login. Both of these posts are part of a series I created about Kerberos and Windows Security. In this post, we will look at the use of Kerberos authentication in SPNEGO that allows Kerberos to be used with the HTTP protocol to protect a web site.

These network traces were captured with Wireshark v2.6.0 on a set of Windows Server 2016 instances running on AWS. There were three servers: a domain controller, a test server where IE was run, and a web server running IIS. The network traces were captured on the test server where Internet Explorer was running.

What is SPNEGO?

The Simple and Protected Generic Security Service Application Program Interface (GSS-API) Negotiation Mechanism (or SPNEGO) is defined by RFC 4178. This functionality is extended by RFC 4559 entitled "SPNEGO-based Kerberos and NTLM HTTP Authenticationin Microsoft Windows." RFC 4559 defines the "SPNEGO" behavior most Windows users are familiar with in corporate settings. This RFC extends the Windows authentication negotiate concept to the HTTP(S) protocols — let's call it HTTP Negotiate or Microsoft's HTTP Negotiate. Microsoft created the concept. It was later formally defined in the RFC. It was originally implemented in Internet Explorer 5.x and Internet Information Server 5.0.

At one point, if you wanted Single Sign On to web applications in a corporate setting, you were configuring SPNEGO and the users had to use Internet Explorer. Luckily, the industry has moved on since then. Today, most of the major browsers support SPNEGO, but more importantly there are several other identity protocols that can be used in its place. It has been several years since I have seen SPNEGO in use at a client site; even then, it was a legacy system that no one had touched in several years.

I'm using it as an example here because it demonstrates Kerberos very nicely with something that everyone can relate to — accessing a web site.

Environment Setup

- The Windows Domain was setup as described in this .
- IIS was installed and configured as described .

- Kerberos was configured on IIS as described . Note, a few small things changed for Windows Server 2016.

Assumptions

- I skip most of the details of what each actor is doing and instead focus on the messages exchanged by the protocol here. See this for more information about the processing done by each Kerberos actor.
- Likewise, I skip most of the introductory material in this post and jump straight to what is needed in order to understand the network traces. If you are having trouble following, please see this .
- The network traces captured for this post were generated with Windows Server 2016 running on AWS.
- No effort was made to obfuscate any of the information in these screenshots. All traffic was generated in a test environment that will no longer exist by the time this post is published.

Import Data Structures

This information is being repeated from the first Kerberos example I published for completeness.

Structure of a Kerberos Ticket

A Kerberos Ticket includes the following information:

Unencrypted Part:

- Version number of ticket format.
- Service realm
- Service principal

Encrypted Part:

- Ticket flags*
- Session key
- Client realm
- Client principal (username)
- List of Kerberos realms that took part in authenticating the user to whom this ticket was issued.
- Timestamp and other meta data about last initial request.
- Time client was authenticated.
- Validity period start time (optional).
- Validity period end time.
- Ticket Granting Server (TGS) Name/ID
- Timestamp

- Client (workstation) Address
- Lifetime
- Authorization-data — used to pass authorization data from the principal on whose behalf a ticket was issued to the application service (see of for more information)

*The following flags can be used in a ticket:

- reserved(0)
- forwardable(1)
- forwarded(2)
- proxiable(3)
- proxy(4)
- may-postdate(5)
- postdated(6)
- invalid(7)
- renewable(8)
- initial(9)
- pre-authent(10)
- hw-authent(11)
- transited-policy-checked(12)
- ok-as-delegate(13)

We will see two tickets in this example: Ticket Granting Ticket (TGT) and Service Ticket.

Structure of a Kerberos Authenticator

A Kerberos Authenticator contains the following information (all encrypted):

- Timestamp
- client ID
- application-specific checksum
- initial sequence number KRB_SAFE or KRB_PRIV messages)
- session sub-key (used in negotiations for a session key unique to this particular session)

Authenticators must not be re-used. A server that encounters a replayed authenticator must reject the message.

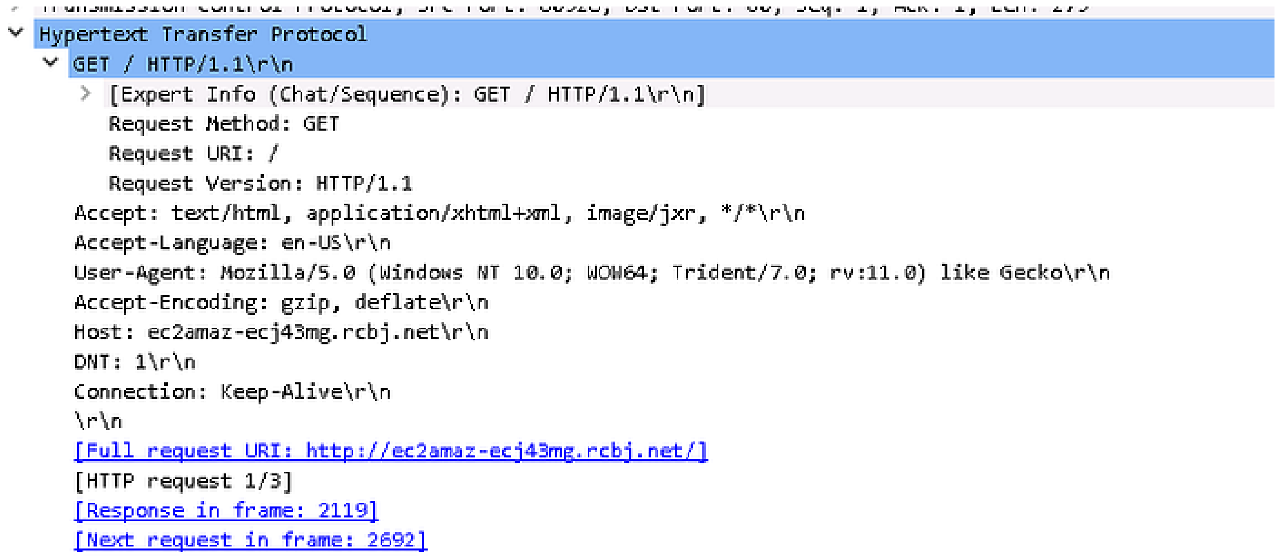
We will see one authenticator in this request: the authenticator sent with the TGT-REQ message.

SPNEGO Wire Captures

The part of this that is new is really at the bottom in the Client/Server Exchange. Up to that point, everything we see about Kerberos was present in the first example we looked at.

We will not go into a significant amount of detail about the Kerberos messages exchanged between the Kerberos client (server running the browser) and the domain controller (KDC).

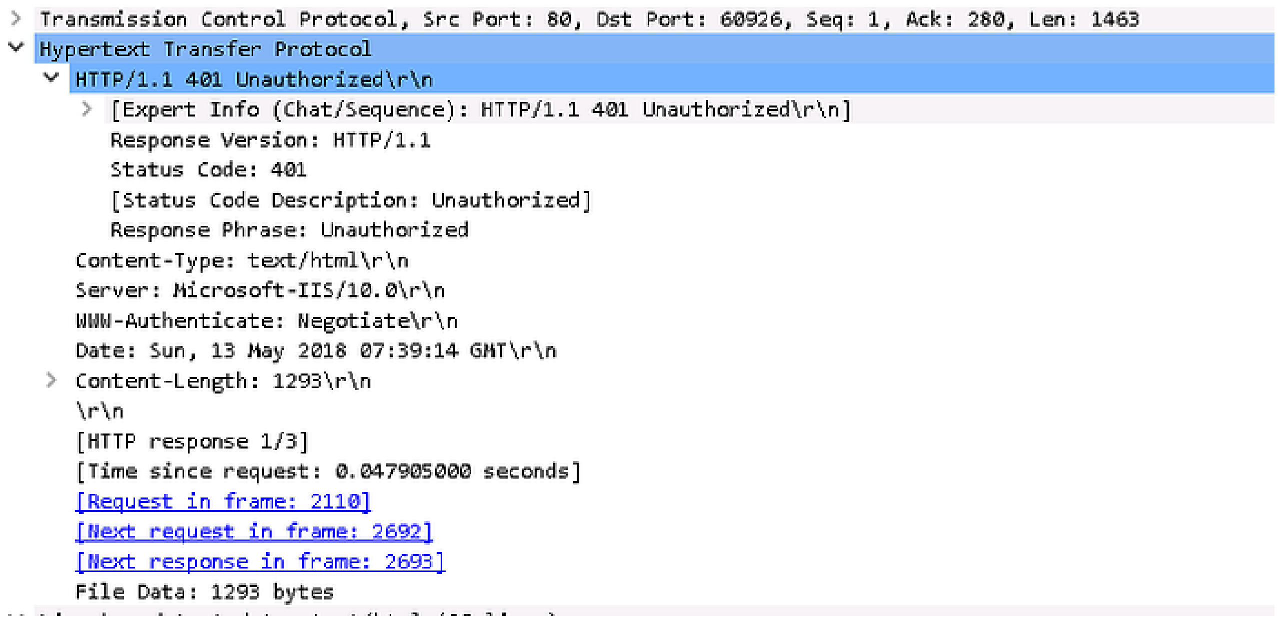
Initially the user points the web browser at the web site were Kerberos is enabled. That address would be “http://ec2amaz-ecj43mg.rcbj.net” in this particular example. The browser will send the following request to the web server.



```
v Hypertext Transfer Protocol
  v GET / HTTP/1.1\r\n
    > [Expert Info (Chat/Sequence): GET / HTTP/1.1\r\n]
      Request Method: GET
      Request URI: /
      Request Version: HTTP/1.1
    Accept: text/html, application/xhtml+xml, image/jxr, */*\r\n
    Accept-Language: en-US\r\n
    User-Agent: Mozilla/5.0 (Windows NT 10.0; WOW64; Trident/7.0; rv:11.0) like Gecko\r\n
    Accept-Encoding: gzip, deflate\r\n
    Host: ec2amaz-ecj43mg.rcbj.net\r\n
    DNT: 1\r\n
    Connection: Keep-Alive\r\n
    \r\n
    [Full request URI: http://ec2amaz-ecj43mg.rcbj.net/]
    [HTTP request 1/3]
    [Response in frame: 2119]
    [Next request in frame: 2692]
```

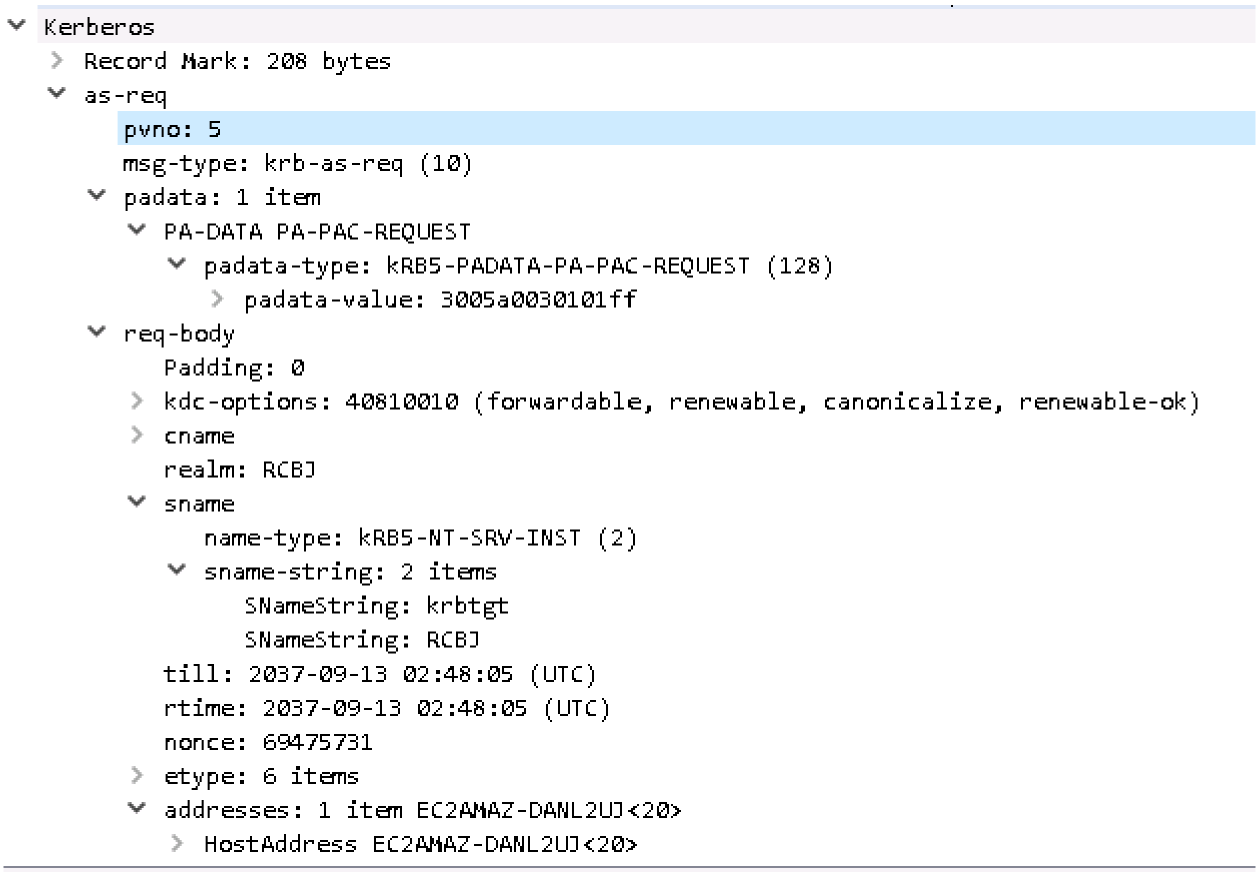
There is nothing particularly special about that HTTP GET request.

After processing the request, IIS will return the following 401 response.



```
> Transmission Control Protocol, Src Port: 80, Dst Port: 60926, Seq: 1, Ack: 280, Len: 1463
v Hypertext Transfer Protocol
  v HTTP/1.1 401 Unauthorized\r\n
    > [Expert Info (Chat/Sequence): HTTP/1.1 401 Unauthorized\r\n]
      Response Version: HTTP/1.1
      Status Code: 401
      [Status Code Description: Unauthorized]
      Response Phrase: Unauthorized
    Content-Type: text/html\r\n
    Server: Microsoft-IIS/10.0\r\n
    WWW-Authenticate: Negotiate\r\n
    Date: Sun, 13 May 2018 07:39:14 GMT\r\n
  > Content-Length: 1293\r\n
    \r\n
    [HTTP response 1/3]
    [Time since request: 0.047905000 seconds]
    [Request in frame: 2110]
    [Next request in frame: 2692]
    [Next response in frame: 2693]
    File Data: 1293 bytes
```

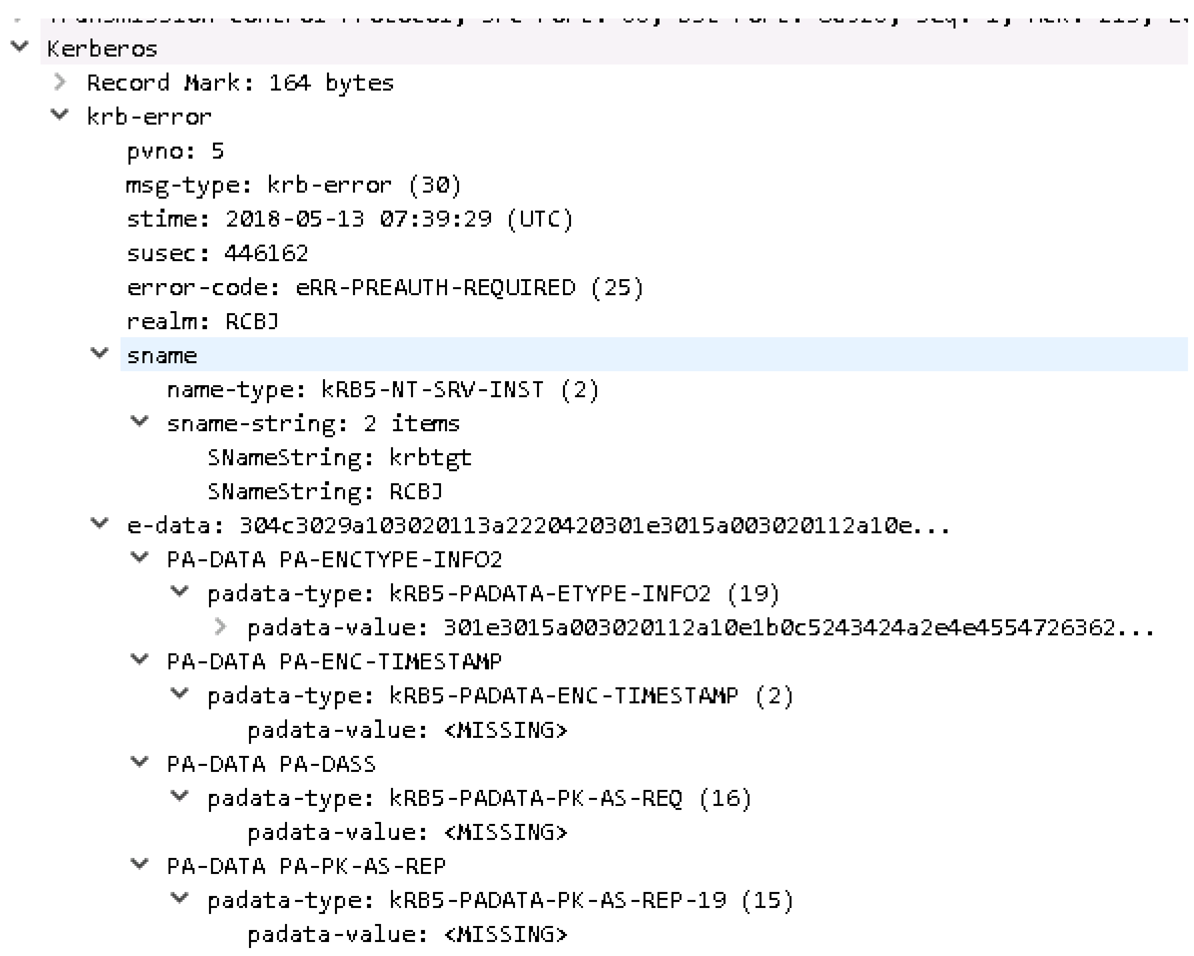
Notice the “WWW-Authenticate: Negotiate” HTTP Response Header. This tells the web browser (Internet Explorer in this case) that it needs to check with the local OS regarding what options it has available from the Negotiate Security Support Provider (SSP) to authenticate the user. From this description, there is a preference for Kerberos if available. So, Windows begins a Kerberos authentication sequence. The browser prompts for the user's credentials (domain user and password) — we want it to do that in this scenario (see end of the post for an explanation). The credential is then used to derive the user's secret key as described in my original write up on Kerberos. The server (running IE) then sends the following Kerberos AS-REQ message to the domain controller.



```
Kerberos
  Record Mark: 208 bytes
  as-req
      pvno: 5
      msg-type: krb-as-req (10)
    padata: 1 item
      PA-DATA PA-PAC-REQUEST
        padata-type: kRB5-PADATA-PA-PAC-REQUEST (128)
          padata-value: 3005a0030101ff
    req-body
        Padding: 0
      kdc-options: 40810010 (forwardable, renewable, canonicalize, renewable-ok)
      cname
        realm: RCBJ
      sname
          name-type: kRB5-NT-SRV-INST (2)
        sname-string: 2 items
            SNameString: krbtgt
            SNameString: RCBJ
        till: 2037-09-13 02:48:05 (UTC)
        rtime: 2037-09-13 02:48:05 (UTC)
        nonce: 69475731
      etype: 6 items
      addresses: 1 item EC2AMAZ-DANL2UJ<20>
        HostAddress EC2AMAZ-DANL2UJ<20>
```

This message is requesting a Ticket Granting Ticket (TGT) for the RCBJ realm that is forwardable, renewable, and canonicalized. The ticket is being requested to valid and renewable until a date far into the future.

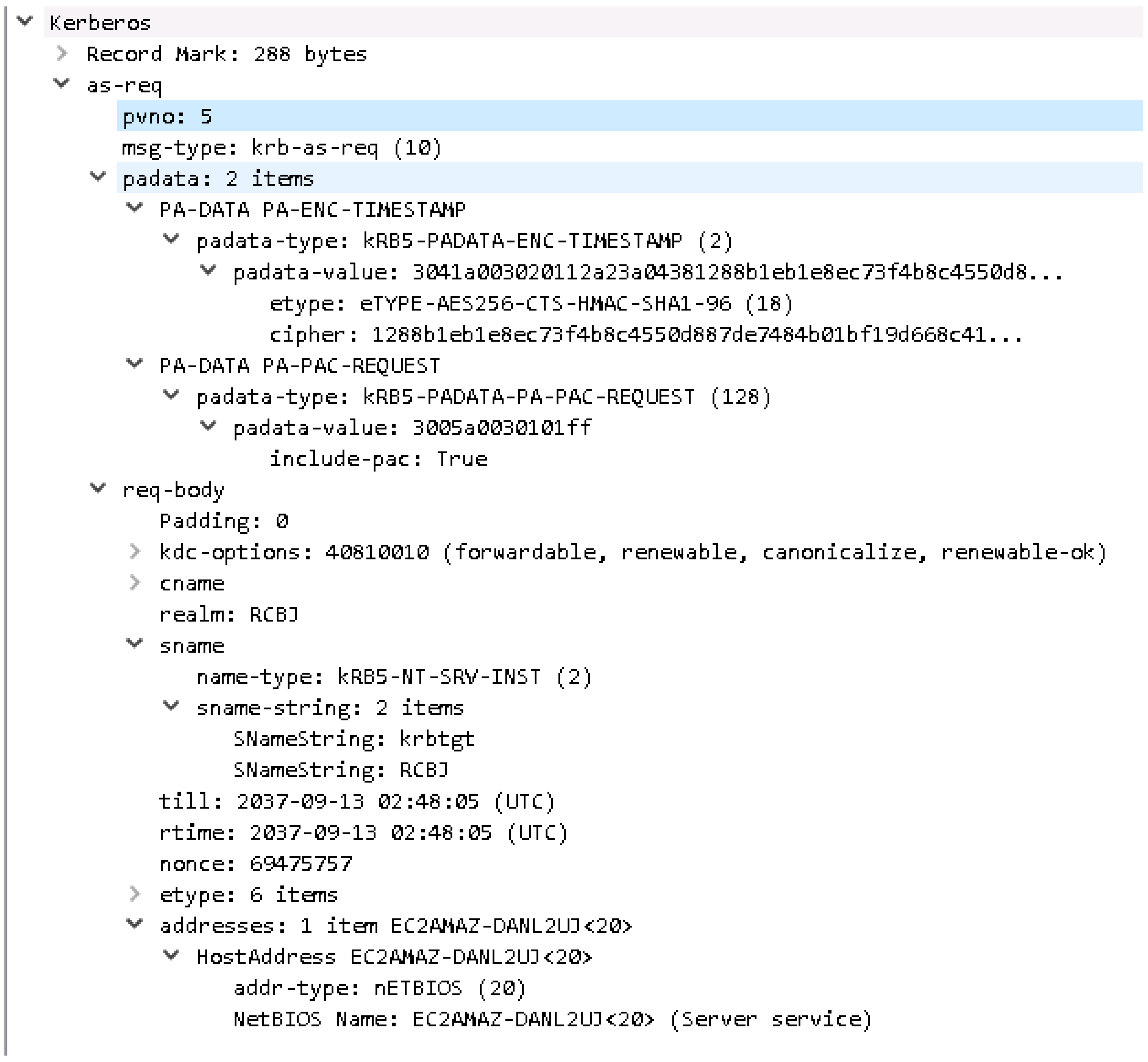
Since this request does not contain the desired pre-authentication data (an encrypted timestamp), the following error is returned.



```
Kerberos
  > Record Mark: 164 bytes
  v krb-error
      pvno: 5
      msg-type: krb-error (30)
      stime: 2018-05-13 07:39:29 (UTC)
      susec: 446162
      error-code: eRR-PREAUTH-REQUIRED (25)
      realm: RCBJ
    v sname
        name-type: kRB5-NT-SRV-INST (2)
      v sname-string: 2 items
          SNameString: krbtgt
          SNameString: RCBJ
    v e-data: 304c3029a103020113a2220420301e3015a003020112a10e...
      v PA-DATA PA-ENCTYPE-INFO2
        v padata-type: kRB5-PADATA-ETYPE-INFO2 (19)
          > padata-value: 301e3015a003020112a10e1b0c5243424a2e4e4554726362...
      v PA-DATA PA-ENC-TIMESTAMP
        v padata-type: kRB5-PADATA-ENC-TIMESTAMP (2)
            padata-value: <MISSING>
      v PA-DATA PA-DASS
        v padata-type: kRB5-PADATA-PK-AS-REQ (16)
            padata-value: <MISSING>
      v PA-DATA PA-PK-AS-REP
        v padata-type: kRB5-PADATA-PK-AS-REP-19 (15)
            padata-value: <MISSING>
```

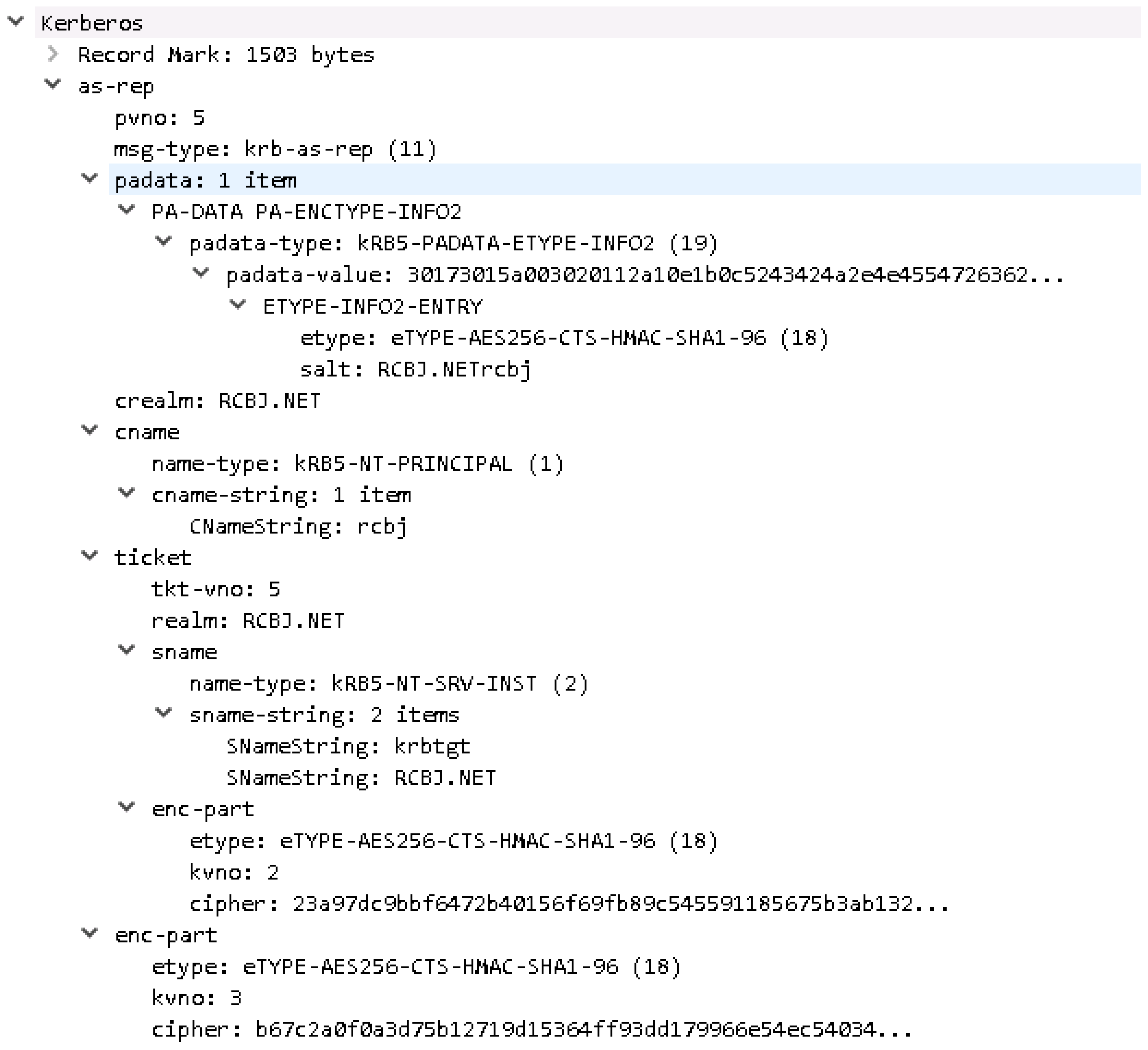
So, the server (where the browser is running) encrypts the current timestamp with the user's secret key and places it in the padata field in a new AS-REQ message. All other fields are essentially the same as the first request that is sent.

The following AS-REQ message is sent to the domain controller.



```
Kerberos
  Record Mark: 288 bytes
  as-req
      pvno: 5
      msg-type: krb-as-req (10)
    padata: 2 items
      PA-DATA PA-ENC-TIMESTAMP
        padata-type: kRB5-PADATA-ENC-TIMESTAMP (2)
          padata-value: 3041a003020112a23a04381288b1eb1e8ec73f4b8c4550d8...
              etype: eTYPE-AES256-CTS-HMAC-SHA1-96 (18)
              cipher: 1288b1eb1e8ec73f4b8c4550d887de7484b01bf19d668c41...
      PA-DATA PA-PAC-REQUEST
        padata-type: kRB5-PADATA-PA-PAC-REQUEST (128)
          padata-value: 3005a0030101ff
              include-pac: True
    req-body
        Padding: 0
      kdc-options: 40810010 (forwardable, renewable, canonicalize, renewable-ok)
      cname
        realm: RCBJ
      sname
          name-type: kRB5-NT-SRV-INST (2)
        sname-string: 2 items
            SNameString: krbtgt
            SNameString: RCBJ
        till: 2037-09-13 02:48:05 (UTC)
        rtime: 2037-09-13 02:48:05 (UTC)
        nonce: 69475757
      etype: 6 items
      addresses: 1 item EC2AMAZ-DANL2UJ<20>
        HostAddress EC2AMAZ-DANL2UJ<20>
            addr-type: nETBIOS (20)
            NetBIOS Name: EC2AMAZ-DANL2UJ<20> (Server service)
```

The authentication service on the domain controller processes the request and sends back an AS-REP message that is similar to the following.

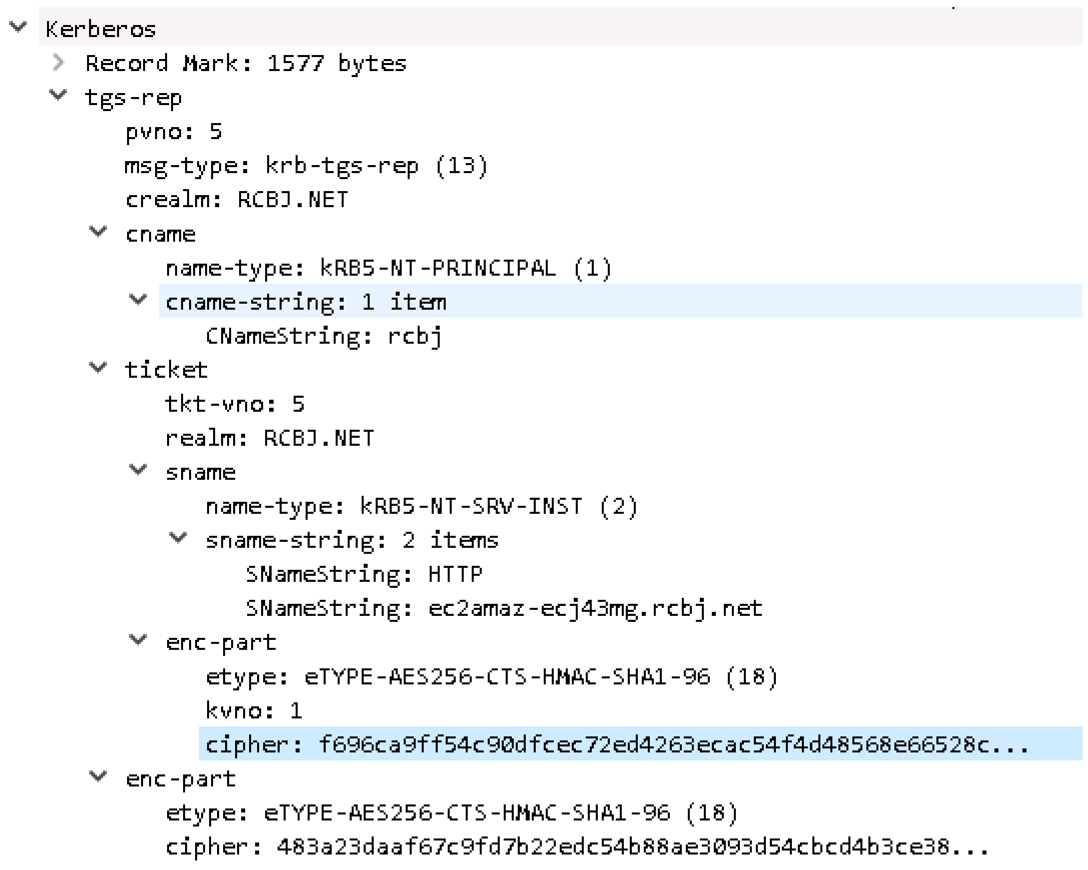


```
Kerberos
  Record Mark: 1503 bytes
  as-rep
    pvno: 5
    msg-type: krb-as-rep (11)
    padata: 1 item
      PA-DATA PA-ENCTYPE-INFO2
        padata-type: kRB5-PADATA-ETYPE-INFO2 (19)
          padata-value: 30173015a003020112a10e1b0c5243424a2e4e4554726362...
            ETYPE-INFO2-ENTRY
                etype: eTYPE-AES256-CTS-HMAC-SHA1-96 (18)
                salt: RCBJ.NETrcbj
    crealm: RCBJ.NET
    cname
      name-type: kRB5-NT-PRINCIPAL (1)
      cname-string: 1 item
        CNameString: rcbj
    ticket
      tkt-vno: 5
      realm: RCBJ.NET
      sname
        name-type: kRB5-NT-SRV-INST (2)
        sname-string: 2 items
          SNameString: krbtgt
          SNameString: RCBJ.NET
      enc-part
        etype: eTYPE-AES256-CTS-HMAC-SHA1-96 (18)
        kvno: 2
        cipher: 23a97dc9bbf6472b40156f69fb89c545591185675b3ab132...
    enc-part
      etype: eTYPE-AES256-CTS-HMAC-SHA1-96 (18)
      kvno: 3
      cipher: b67c2a0f0a3d75b12719d15364ff93dd179966e54ec54034...
```

The AS-REP message contains a session key (encrypted with the user’s secret key) and the TGT (encrypted with the TGS’s secret key). At this point, the session key is decrypted by the client and the TGT is available to be placed in the next message.

The TGS-REQ message is built by placing the TGT and an authenticator into an authentication header in the pre-authentication (padata) field. You can also see in the request body that a Service Ticket is being requested for HTTP@ec2amaz-ecj43mg.rcbj.net. So, a Service Ticket for the HTTP Service at the very server our original request was sent to. The TGS-REQ looks similar to the following:

The TGS processes the request and returns a TGS-REP message that looks similar to the following.



```
Kerberos
  Record Mark: 1577 bytes
  tgs-rep
    pvno: 5
    msg-type: krb-tgs-rep (13)
    crealm: RCBJ.NET
    cname
      name-type: kRB5-NT-PRINCIPAL (1)
      cname-string: 1 item
        CNameString: rcbj
    ticket
      tkt-vno: 5
      realm: RCBJ.NET
      sname
        name-type: kRB5-NT-SRV-INST (2)
        sname-string: 2 items
          SNameString: HTTP
          SNameString: ec2amaz-ecj43mg.rcbj.net
      enc-part
        etype: eTYPE-AES256-CTS-HMAC-SHA1-96 (18)
        kvno: 1
        cipher: f696ca9ff54c90dfcec72ed4263ecac54f4d48568e66528c...
    enc-part
      etype: eTYPE-AES256-CTS-HMAC-SHA1-96 (18)
      cipher: 483a23daaf67c9fd7b22edc54b88ae3093d54cbcd4b3ce38...
```

This response contains the Service Ticket (encrypted with the service's secret key) and a client-server (or service) session key (encrypted with the client-TGS session key).

At this point, the client can decrypt the service session key.

The service session key is used to encrypt a new authenticator that is included with the next message sent to the web server.

The client builds an AP-REQ message that contains the encrypted authenticator and the Service Ticket it just received.

The AP-REQ message contains the following fields:

- pvno: The Kerberos protocol version number (5).
- msg-type: Application class tag number(14).
- padding: Defined by RFC 1964, . Adds necessary padding to size data structure appropriately for encryption and digital signature algorithms.
- ap-options: Flags and options that can be set on this type of request (in this case mutual authentication is required, the server must authenticate itself). See below.
- ticket: The Service Ticket encrypted with the services secret key.
- ticket->tkt-vno: Ticket format version number (5).
- ticket->realm: The realm this ticket was issued for (rcbj.net).
- ticket->sname: The service name this ticket is meant to be used for (HTTP@ec2amaz-ecj43mg.rcbj.net)
- ticket->enc-part: The encrypted part of the ticket. It is encrypted with the service's (HTTP@ec2amaz-ecj43mg.rcbj.net) secret key.

- authenticator: The authenticator for this request (encrypted with the service session key).

The available ap-options defined by RFC4120 are:

- reserved(0)
- use-session-key(1)
- mutual-required(2)

This ap-req message is stored in the Kerberos field of the krb5_blob field.

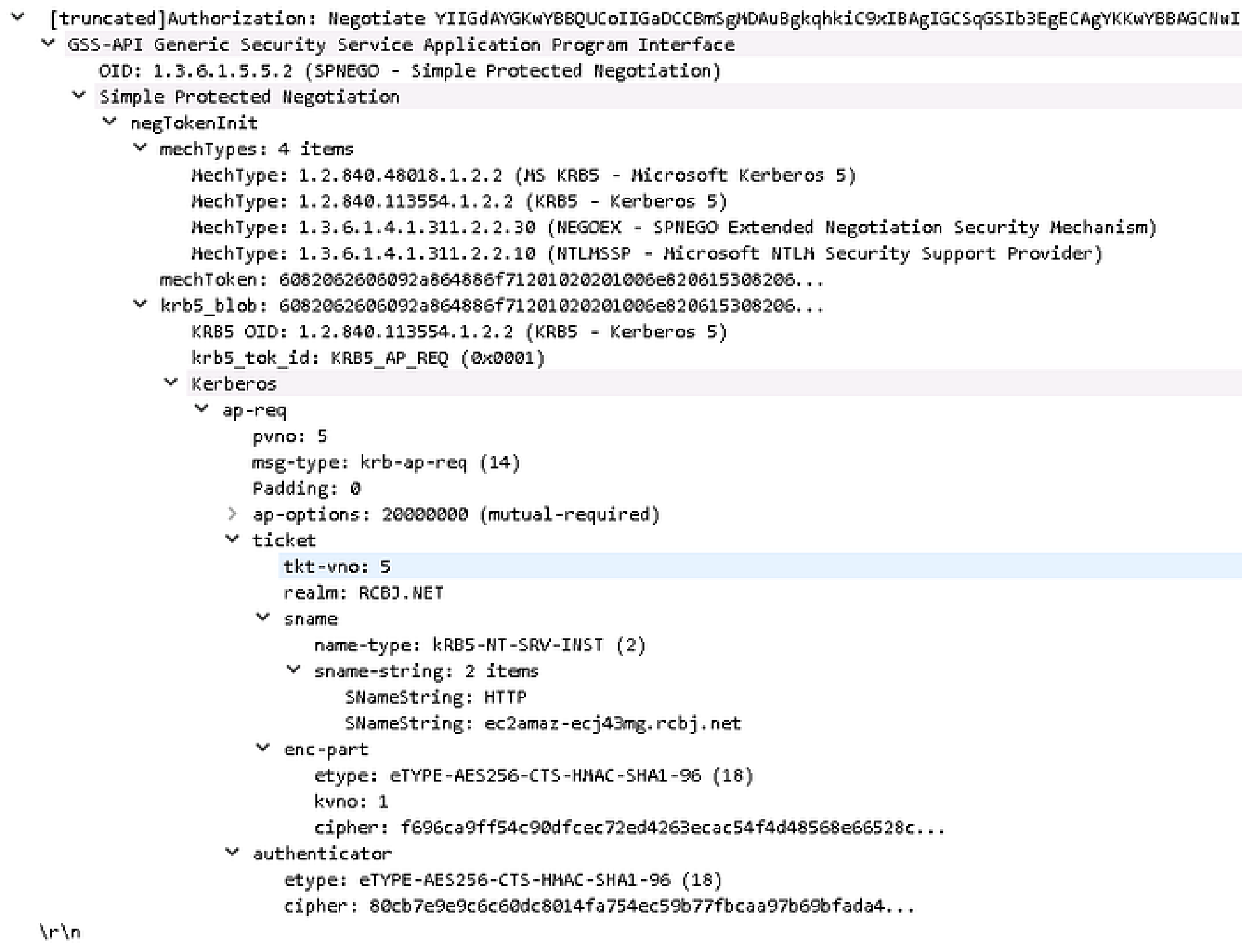
The mechTypes field (defines what authentication mechanisms are in play for a particular request), mechToken(the optimistic mechanism token), and krb5_blob (see above) are placed inside of a negTokenInit data structure. These are all defined by RFC 4178.

The negTokenInit message is placed in the Simple Protected Negotation message, which is in turn embedded in the GSS-API Generic Security Service Application Program Interface message.

That outer message is then stored in base64-encoded form in the Authorization header in the form:

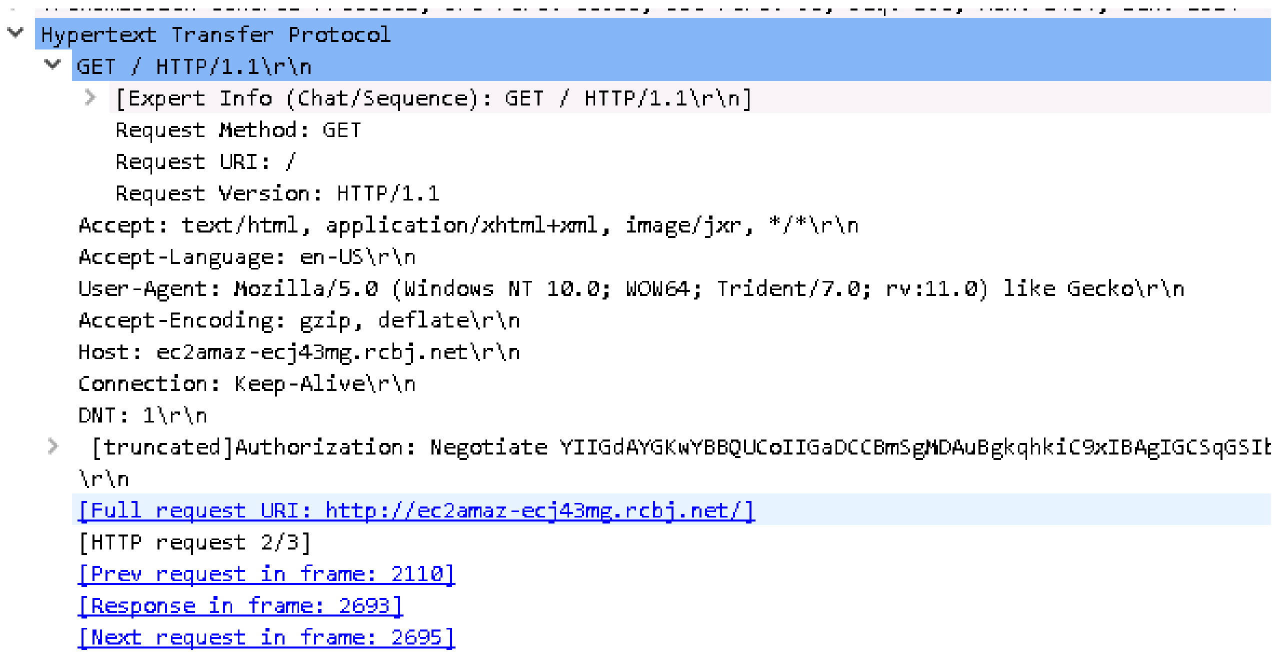
```
Authorization: Negotiate blah_blah_blah
```

The unencoded message looks like the following:



```
v  [truncated]Authorization: Negotiate YIIGdAYGKwYBBQUCoIIGaDCCBmSgMDAuBgkqhkiC9xIBAgIGCSqGSIb3EgECAgYKKwYBBAGCNwI
  v GSS-API Generic Security Service Application Program Interface
      OID: 1.3.6.1.5.5.2 (SPNEGO - Simple Protected Negotiation)
    v Simple Protected Negotiation
      v negTokenInit
        v mechTypes: 4 items
            MechType: 1.2.840.48018.1.2.2 (MS KRB5 - Microsoft Kerberos 5)
            MechType: 1.2.840.113554.1.2.2 (KRB5 - Kerberos 5)
            MechType: 1.3.6.1.4.1.311.2.2.30 (NEGOEX - SPNEGO Extended Negotiation Security Mechanism)
            MechType: 1.3.6.1.4.1.311.2.2.10 (NTLMSSP - Microsoft NTLM Security Support Provider)
          mechToken: 6082062606092a864886f71201020201006e820615308206...
        v krb5_blob: 6082062606092a864886f71201020201006e820615308206...
            KRB5 OID: 1.2.840.113554.1.2.2 (KRB5 - Kerberos 5)
            krb5_tok_id: KRB5_AP_REQ (0x0001)
          v Kerberos
            v ap-req
                pvno: 5
                msg-type: krb-ap-req (14)
                Padding: 0
              > ap-options: 20000000 (mutual-required)
              v ticket
                  tkt-vno: 5
                  realm: RCBJ.NET
                v sname
                    name-type: kRB5-NT-SRV-INST (2)
                  v sname-string: 2 items
                      SNameString: HTTP
                      SNameString: ec2amaz-ecj43mg.rcbj.net
                v enc-part
                    etype: eTYPE-AES256-CTS-HMAC-SHA1-96 (18)
                    kvno: 1
                    cipher: f696ca9ff54c90dfcec72ed4263ecac54f4d48568e66528c...
              v authenticator
                  etype: eTYPE-AES256-CTS-HMAC-SHA1-96 (18)
                  cipher: 80cb7e9e9c6c60dc8014fa754ec59b77fbcaa97b69bfada4...
  \r\n
```

The base64-encoded message in the Authorization request header looks something like this:



```
Hypertext Transfer Protocol
  GET / HTTP/1.1\r\n
    [Expert Info (Chat/Sequence): GET / HTTP/1.1\r\n]
    Request Method: GET
    Request URI: /
    Request Version: HTTP/1.1
  Accept: text/html, application/xhtml+xml, image/jxr, */*\r\n
  Accept-Language: en-US\r\n
  User-Agent: Mozilla/5.0 (Windows NT 10.0; WOW64; Trident/7.0; rv:11.0) like Gecko\r\n
  Accept-Encoding: gzip, deflate\r\n
  Host: ec2amaz-ecj43mg.rcbj.net\r\n
  Connection: Keep-Alive\r\n
  DNT: 1\r\n
  [truncated]Authorization: Negotiate YIIGdAYGKwYBBQUCoIIGaDCCBmSgMDAuBgkqhkiC9xIBAgIGCSqGSIt
  \r\n
  [Full request URI: http://ec2amaz-ecj43mg.rcbj.net/]
  [HTTP request 2/3]
  [Prev request in frame: 2110]
  [Response in frame: 2693]
  [Next request in frame: 2695]
```

The browser sends the above message to the IIS server it originally tried to communicate with. It's replaying the request, but with the requested authentication information this time.

Upon receiving this message, IIS extracts the authenticator, service ticket, and meta data that tells it that the provided information is a Kerberos ticket.

IIS uses its secret key to decrypt the service ticket. It uses the session key inside the service ticket to decrypt the authenticator. Then, it compares the username inside the authenticator to the username in the service ticket. If the users match and all other checks are successful, then the request is authenticated and processing proceeds as normal.

IIS builds a SPNEGO data structure that is placed in the WWW-Authenticate response header.

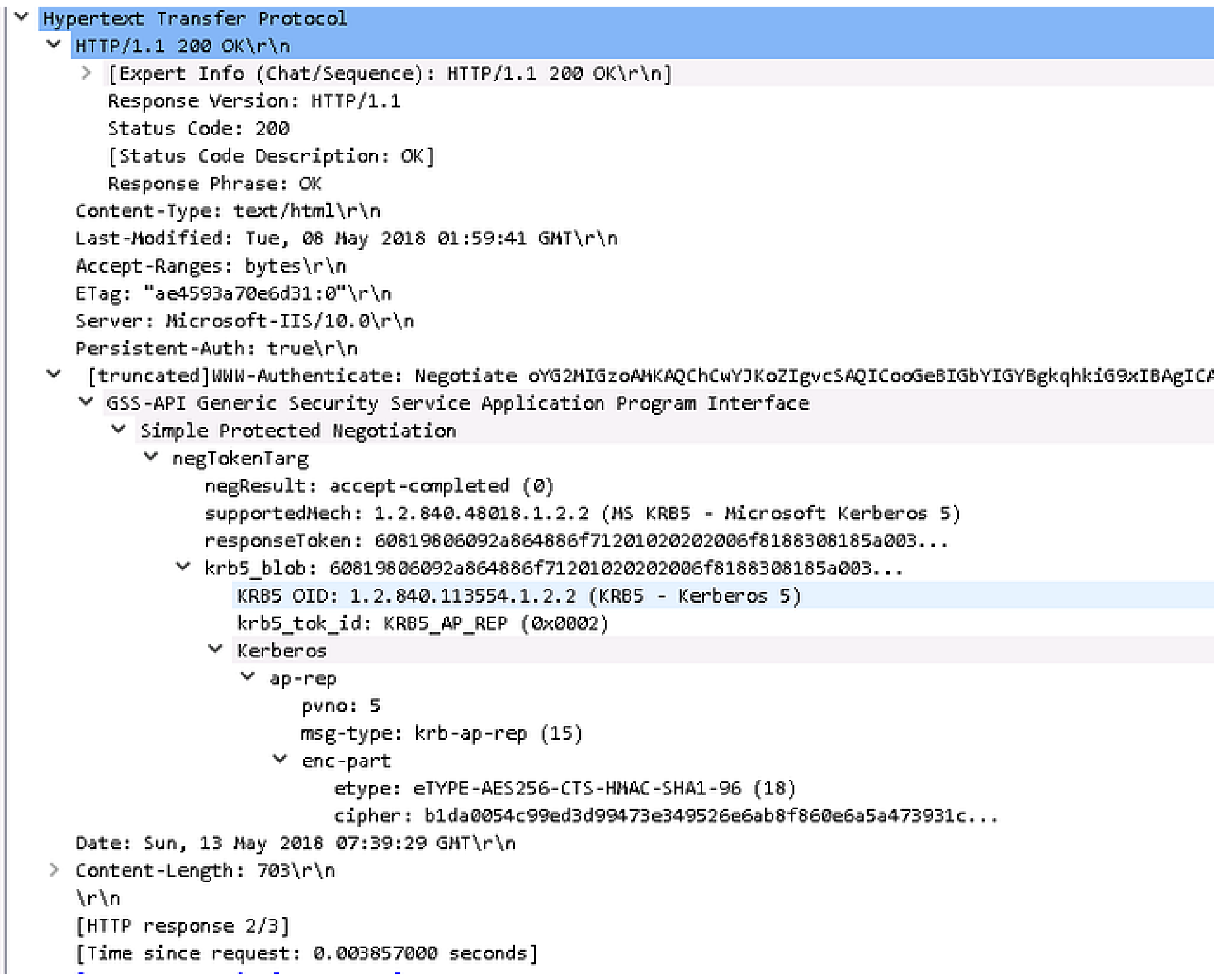
It contains a Kerberos AP-REP message in the krb5_blob field.

The AP-REP message contains the following fields:

- pvno: The Kerberos protocol version number (5).
- msg-type: Application class tag number (15).
- enc-part: contains the timestamp that was sent to the service in the authenticator (in the AP-REQ message). It is encrypted with the service session key.

You will notice that the supported mechanisms (supportedMech) field contains only one entry for "MS KRB5 — Microsoft Kerberos 5".

The response IIS sends will looks similar to the following:



```
v Hypertext Transfer Protocol
  v HTTP/1.1 200 OK\r\n
    > [Expert Info (Chat/Sequence): HTTP/1.1 200 OK\r\n]
      Response Version: HTTP/1.1
      Status Code: 200
      [Status Code Description: OK]
      Response Phrase: OK
    Content-Type: text/html\r\n
    Last-Modified: Tue, 08 May 2018 01:59:41 GMT\r\n
    Accept-Ranges: bytes\r\n
    ETag: "ae4593a70e6d31:0"\r\n
    Server: Microsoft-IIS/10.0\r\n
    Persistent-Auth: true\r\n
  v  [truncated]WWW-Authenticate: Negotiate oYG2MIGzoAMKAQChCwYJKoZIgvcSAQICooGeBIGbYIGYBgkqhkiG9xIBAgICA
    v GSS-API Generic Security Service Application Program Interface
      v Simple Protected Negotiation
        v negTokenTarg
            negResult: accept-completed (0)
            supportedMech: 1.2.840.48018.1.2.2 (MS KRB5 - Microsoft Kerberos 5)
            responseToken: 60819806092a864886f71201020202006f8188308185a003...
          v krb5_blob: 60819806092a864886f71201020202006f8188308185a003...
              KRB5 OID: 1.2.840.113554.1.2.2 (KRB5 - Kerberos 5)
              krb5_tok_id: KRB5_AP_REP (0x0002)
            v Kerberos
              v ap-rep
                  pvno: 5
                  msg-type: krb-ap-rep (15)
                v enc-part
                    etype: eTYPE-AES256-CTS-HMAC-SHA1-96 (18)
                    cipher: b1da0054c99ed3d99473e349526e6ab8f860e6a5a473931c...
    Date: Sun, 13 May 2018 07:39:29 GMT\r\n
  > Content-Length: 703\r\n
    \r\n
    [HTTP response 2/3]
    [Time since request: 0.003857000 seconds]
```

When the client receives this token, it extracts and decrypts the timestamp using the service session key. If the timestamp matches what it put in the authenticator, the service is authenticated. If the values do not match, an error will be displayed to the user.

Most likely, the web server would set a session cookie in this initial response that is used to track all subsequent requests that are part of this authenticated session. You could likely also keep sending the same Authorization header in subsequent requests. In a SPA (Single Page Application) where an OAuth2 access token is being passed in each API request back to the server, that approach would be undesirable.

In this particular example, Internet Explorer prompted for new user credentials rather than using the cached TGT that was already present for the authenticated user. The web server was not listed in the Local Intranet site list, which is a requirement for the cached Kerberos tokens to be utilized for the current logged in user — see here for more information. This let me capture the entire Kerberos protocol exchange, which is what I was after. If we add the web server URL to the site list, then the server will only make a call to the TGS to obtain a new service ticket for the web server. Or, if there is already a ticket cached for the web server, then that will be used and no Kerberos protocol messages will be exchanged with the domain controller.

This SPNEGO (HTTP Negotiate) example demonstrates an important point very well. While the AS-REQ, AS-REP, TGT-REQ, and TGT-REP messages are exchanged between actors whose function is largely defined by the Kerberos v5 RFC, AP-REQ and AP-RES are passed between actors using a protocol that probably didn’t have Kerberos in mind when it was being developed. So, we end up with Kerberos extensions that map AP-REQ and AP-RES to protocols like HTTP v1.1, which is what SPNEGO and HTTP Negotiate are all about.