UDC 004.9+004.056

INTEGRATED WINDOWS AUTHENTICATION IN WEB APPLICATIONS

DMITRY SAVCHENKO, OKSANA GOLUBEVA, TATYANA CHERNYAK
Polotsk State University, Belarus

The paper discusses a method of transparent user authentication within a web application running in an internal network organized into a domain by means of Microsoft Active Directory.

A web application typically needs to function differently for different users and to process specific data for every user. This requires that the web application is aware of which user interacts with it and this knowledge should be confirmed to be correct. This leads to the requirement of a user to authenticate before starting to interact with the application. A typical approach for this is so that at first the user should register in the application, thus, creating personal credentials, such as user name and password, and then as the first step of every interaction session the user is required to prove his or her identity by providing the credentials via a web form. An alternative approach would be to rely on a trusted third party to provide user authentication. This can be achieved by leveraging OAuth 2.0 protocol [1, 2] to obtain access token to the third-party service REST API [3] and then makinga subsequent API request to obtain some user identifier and in this way authenticate the user. This approach requires the user to be registered and authenticated within the third-party web service.

The two authentication approaches described above are mostly suitable and, thus, widely used by global web applications available worldwide in the internet. However, a lot of web services can be installed on private servers and used in internal networks of some organizations. These can be privately developed applications and/or applications that process business-sensitive information that should not be let outside the internal network. While methods of web form authentication and third-party authentication can still be used by internal web applications, they are typically redundant in such cases. A user has already been authenticated within organization network when logging in to his or her computer (most often Microsoft Active Directory [4] is used to organize network domain) and the best solution is for the internal web application to transparently authenticate such a user without asking him or her for credentials or performing communication with a third party.

The Generic Security Service Application Program Interface (GSS-API) is a security services standard, that defines a generic interface for callers to make requests to underlying security services [5]. The definitive concept of GSS-API usage is the exchange of opaque messages (they are called "tokens") which hide the implementation detail from the higher-level application. The client and server sides of the application are written to convey the tokens given to them by their respective GSS-API implementations. GSS-API tokens can usually travel over an insecure network as the mechanisms provide inherent message security. After the exchange of some number of tokens, the GSS-API implementations at both ends inform their local application that a security context has been established.

The Microsoft Security Support Provider Interface (SSPI) is the well-defined common API for obtaining integrated security services for authentication, message integrity, message privacy, and security quality of service for any distributed application protocol [6]. SSPI partially confirms with GSS-API interface while having some Windows-specific extensions. A single implementation of SSPI is called a security service package. The SSPI allows an application to use any of the available security packages on a system without changing the interface to use security services. Microsoft Windows provides several security packages, NTLM (NT LAN Manager) and more progressive Kerberos are among them. A higher level package Negotiate is also provided: it does not define different security service implementation, but serves as a decorator for NTLM and Kerberos instead. Negotiate package allows communication parties to select between Kerberos and NTLM depending on which of these two is supported by both parties, considering Kerberos as a preferred one.

The Hypertext Transfer Protocol (HTTP) provides a simple and generic challenge-response authentication mechanism which may be used by a server to challenge a client request and by a client to provide authentication information [7]. This mechanism allows various specific implementations that leverage different authentication techniques. Authentication information challenging is done by a server with the *WWW-Authenticate* header and response code *401* (unauthorized). Providing authentication information by a client is achieved with the *Authorization* header.

The GSS-API standard, SSPI interface and its NTLM, Kerberos and Negotiate implementations alongside with HTTP authentication mechanism together provide means for implementing integrated windows authentication for web applications that operate within a Windows network domain. This can be achieved by leveraging HTTP Negotiate Authentication scheme [8]. This scheme defines authentication protocol in terms of GSS-API and can be mapped to the terms of the SSPI Negotiate package implementation. According to this schema, the following protocol of data exchange is used by client and server application. When the server receives a request

for a resource that is access-protected and an acceptable Authorization header has not been sent, the server responds with a 401 (unauthorized) status code and sends a WWW-Authenticate header denoting "Negotiate" as a required authentication schema. No gss-api data is provided in the header of the initial response. Upon receipt of the response containing WWW-Authenticate header, the client is expected to retry the previous request, sending Authorization header along with it. The header should denote "Negotiate" as a used authentication schema along with base64-encoded [9] gss-api data. The gss-api data should contain security context token, obtained from the client system GSS-API implementation (SSPI Negotiate security package). The server then uses its system GSS-API implementation to validate security context token received and, thus, authenticate the user. The server then returns the requested resource with 200 (success) status code; WWW-Authenticate header may be also sent containing server security context token if mutual authentication is required. Depending on GSS-API implementation there may be multiple legs of security token exchange between the client and the server.

The following can be seen as an example of negotiate authentication.

The client requests an access-protected resource from the server via HTTP GET method request:

Client: GET example/path/document.html

Since no Authorization header is sent by the client, the server should response with the request for authentication:

Server: HTTP/1.1 401 Unauthorized
Server: WWW-Authenticate: Negotiate

The client then is required to obtain user credentials using *GSS_Init_sec_context* call. In terms of SSPI the client should first call *AcquireCredentialsHandle* function to obtain Negotiate security package credentials handle (*client credentials handle*), and then call*InitializeSecurityContext* passing client credentials handle as an argument and receiving first part of the security context data. This context is then base64-encoded and sent to the server:

Client: GET example/path/document.html
Client: Authorization: Negotiate Y2xpZW50LW5lZ290aWF0==

Upon receiving the request, the server should validate the client security context data using *GSS_Accept_sec_context* call. In terms of SSPI it should as well first obtain Negotiate security package credentials handle (*server credentials handle*) by calling *AcquireCredentialsHandle* and then call *AcceptSecurityContext* passing server credentials handle and the client security context data as arguments. The *AcceptSecurityContext* function returns the status (whether the passed client security context is valid, invalid or not complete) and the server security context data that should be passed to the client in WWW-Authenticate header. If the client security context is incomplete, the server should return 401 (unauthorized) header along with base64-encoded server security context:

Server: HTTP/1.1 401 Unauthorized
Server: WWW-Authenticate: Neghotiate c2VydmVyLWNvbnRleHQ=

The client should decode received server security context data and pass it to *GSS_Init_sec_context* call. In terms of SSPI the data along with previously acquired client credentials handle is passed to *InitializeSecurityContext* call. The second part of client security context received from the function call is then base64-encoded and sent to the server:

Client: GET example/path/document.html
Client: Authorization: Negotiate c2Vjb25kLWNsaWVudC1jb250ZXh0

This cycle continues until the security context is complete. When the return value from the *GSS_accept_sec_context* function (*AcceptSecurityContext* in terms of SSPI) on the server indicates that the security context is complete, it may supply final authentication data to be returned to the client. If the server has more gss-api data to send to the client to complete the context, it is to be carried in a WWW-Authenticate header with the final response containing the HTTP body. So finally, the server returns 200 (success) status code along with the final part of the gss-api data and the requested document body:

Server: HTTP/1.1 200 Success
Server: WWW-Authenticate: Neghotiate ZmluYWwtc2VydmVyLWNvbnRleHQ=
Server:
Server: <!DOCTYPE html>
Server: <html>
Server: <head>
Server: …

Since Negotiate Authentication Scheme is supported by all major web browsers, described technique allows to perform fully transparent user authentication. The user neither needs to register within the web application, nor is required to supply his or her credentials before being able to use the application. The user simply navigates to the required URI via a web browser, authentication is then performed automatically in a secure manner, and the user gets recognized by the application and is able to interact with it.

REFERENCES

1. RFC 6749 — The OAuth 2.0 Authorization Framework [Electronic Resource] / Dick Hardt. – Mode of access: https://tools.ietf.org/pdf/rfc6749.pdf. – Date of access: 20.01.2017.
2. RFC 6750 — The OAuth 2.0 Authorization Framework: Bearer Token Usage [Electronic Resource] / Michael B. Jones, Dick Hardt. – Mode of access: https://tools.ietf.org/pdf/rfc6750.pdf. – Date of access: 20.01.2017.
3. Fielding, Roy. Architectural Styles and the Design of Network-based Software Architectures / Roy Thomas Fielding; University of California. — Irvine, 2000. — 162 p.
4. Active Directory [Electronic Resource] / Microsoft Corporation. – Mode of access: https://msdn.microsoft.com/en-us/library/bb742424.aspx. – Date of access: 20.01.2017.
5. RFC 2743 — Generic Security Service Application Program Interface Version 2, Update 1 [Electronic Resource] / John Linn. – Mode of access: https://tools.ietf.org/pdf/rfc2743.pdf. – Date of access: 20.01.2017.
6. The Security Support Provider Interface [Electronic Resource] / Microsoft Corporation. – Mode of access: https://msdn.microsoft.com/en-us/library/bb742535.aspx. – Date of access: 20.01.2017.
7. RFC 1945 — Hypertext Transfer Protocol — HTTP/1.0 [Electronic Resource] / Tim Berners-Lee, Roy T. Fielding, Henrik Frystyk Nielsen. – Mode of access: https://tools.ietf.org/pdf/rfc1945.pdf. – Date of access: 20.01.2017.
8. RFC 4559 — SPNEGO-based Kerberos and NTLM HTTP Authentication in Microsoft Windows [Electronic Resource] / Karthik Jaganathan, Larry Zhu, John Brezak. – Mode of access: https://tools.ietf.org/pdf/rfc4559.pdf. – Date of access: 20.01.2017.
9. RFC 4648 — The Base16, Base32, and Base64 Data Encodings [Electronic Resource] / Simon Josefsson. – Mode of access: https://tools.ietf.org/pdf/rfc4648.pdf. – Date of access: 20.01.2017.