Informatie

Inhoudsopgave

SSL/TLS: 1

RADIUS 9

EAP 13

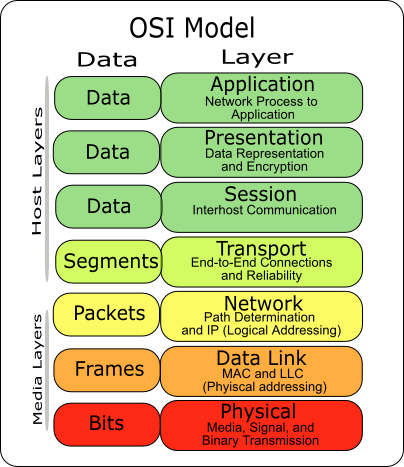
Eduroam: 15

SpaceNet: 15

Security Assertion Markup Language (SAML) 15

OpenSocial 17

VOMS 18



# SSL/TLS:

Secure Socket Layer (SSL) en diens opvolger Transport Layer Security (TLS) zijn encryptie-protocollen op de applicationlayer.

Gebruikt assymetrische crypthografie for key exchange.

Gebruikt symmetric encryption for betrouwbaarheid.

Gebruikt message authentication codes voor message integrity.

TLS:

Once the client and server have decided to use TLS they negotiate a [stateful](http://en.wikipedia.org/wiki/State_(computer_science)) connection by using a [handshaking](http://en.wikipedia.org/wiki/Transport_Layer_Security#TLS_handshake_in_detail) procedure.[[3]](http://en.wikipedia.org/wiki/Transport_Layer_Security#cite_note-3) During this handshake, the client and server agree on various parameters used to establish the connection's security.

1. The client sends the server the client's SSL version number, cipher settings, session-specific data, and other information that the server needs to communicate with the client using SSL.
2. The server sends the client the server's SSL version number, cipher settings, session-specific data, and other information that the client needs to communicate with the server over SSL. The server also sends its own certificate, and if the client is requesting a server resource that requires client authentication, the server requests the client's certificate.
3. The client uses the information sent by the server to authenticate the server. If the server cannot be authenticated, the user is warned of the problem and informed that an encrypted and authenticated connection cannot be established. If the server can be successfully authenticated, the client proceeds to the next step.
4. Using all data generated in the handshake thus far, the client (with the cooperation of the server, depending on the cipher being used) creates the pre-master secret for the session, encrypts it with the server's public key (obtained from the server's certificate, sent in step 2), and then sends the encrypted pre-master secret to the server.
5. If the server has requested client authentication (an optional step in the handshake), the client also signs another piece of data that is unique to this handshake and known by both the client and server. In this case, the client sends both the signed data and the client's own certificate to the server along with the encrypted pre-master secret.
6. If the server has requested client authentication, the server attempts to authenticate the client. If the client cannot be authenticated, the session ends. If the client can be successfully authenticated, the server uses its private key to decrypt the pre-master secret, and then performs a series of steps (which the client also performs, starting from the same pre-master secret) to generate the master secret.
7. Both the client and the server use the master secret to generate the session keys, which are symmetric keys used to encrypt and decrypt information exchanged during the SSL session and to verify its integrity (that is, to detect any changes in the data between the time it was sent and the time it is received over the SSL connection).
8. The client sends a message to the server informing it that future messages from the client will be encrypted with the session key. It then sends a separate (encrypted) message indicating that the client portion of the handshake is finished.
9. The server sends a message to the client informing it that future messages from the server will be encrypted with the session key. It then sends a separate (encrypted) message indicating that the server portion of the handshake is finished.

In [cryptography](http://en.wikipedia.org/wiki/Cryptography), a **message authentication code** (often **MAC**) is a short piece of information used to [authenticate](http://en.wikipedia.org/wiki/Authentication) a [message](http://en.wikipedia.org/wiki/Message) and to provide integrity and authenticity assurances on the message. Integrity assurances detects accidental and intentional message changes, while authenticity assurances affirms the message's origin.

TLS has a variety of security measures:

* Protection against a downgrade of the protocol to a previous (less secure) version or a weaker cipher suite.
* Numbering subsequent Application records with a sequence number and using this sequence number in the [message authentication codes](http://en.wikipedia.org/wiki/Message_authentication_code) (MACs).
* Using a message digest enhanced with a key (so only a key-holder can check the MAC). The [HMAC](http://en.wikipedia.org/wiki/HMAC) construction used by most TLS cipher suites is specified in [RFC 2104](http://tools.ietf.org/html/rfc2104) (SSL 3.0 used a different hash-based MAC).
* The message that ends the handshake ("Finished") sends a hash of all the exchanged handshake messages seen by both parties.
* The [pseudorandom](http://en.wikipedia.org/wiki/Pseudorandomness) function splits the input data in half and processes each one with a different hashing algorithm ([MD5](http://en.wikipedia.org/wiki/MD5) and [SHA-1](http://en.wikipedia.org/wiki/SHA-1)), then [XORs](http://en.wikipedia.org/wiki/Exclusive_or) them together to create the MAC. This provides protection even if one of these algorithms is found to be vulnerable.
* SSL 3.0 improved upon SSL 2.0 by adding SHA-1 based ciphers and support for certificate authentication.

**Simple TLS handshake**

A simple connection example follows, illustrating a handshake where the server (but not the client) is authenticated by its certificate:

1. Negotiation phase:
   * A client sends a **ClientHello** message specifying the highest TLS protocol version it supports, a random number, a list of suggested[CipherSuites](http://en.wikipedia.org/wiki/CipherSuite) and suggested compression methods. If the client is attempting to perform a resumed handshake, it may send a *session ID*.
   * The server responds with a **ServerHello** message, containing the chosen protocol version, a random number, CipherSuite and compression method from the choices offered by the client. To confirm or allow resumed handshakes the server may send a *session ID*. The chosen protocol version should be the highest that both the client and server support. For example, if the client supports TLS1.1 and the server supports TLS1.2, TLS1.1 should be selected; SSL 3.0 should not be selected.
   * The server sends its **Certificate** message (depending on the selected cipher suite, this may be omitted by the server).[[31]](http://en.wikipedia.org/wiki/Transport_Layer_Security#cite_note-openpgp-31)
   * The server sends a **ServerHelloDone** message, indicating it is done with handshake negotiation.
   * The client responds with a **ClientKeyExchange** message, which may contain a *PreMasterSecret*, public key, or nothing. (Again, this depends on the selected cipher.) This *PreMasterSecret* is encrypted using the public key of the server certificate.
   * The client and server then use the random numbers and *PreMasterSecret* to compute a common secret, called the "master secret". All other key data for this connection is derived from this master secret (and the client- and server-generated random values), which is passed through a carefully designed [pseudorandom](http://en.wikipedia.org/wiki/Pseudorandomness) function.
2. The client now sends a **ChangeCipherSpec** record, essentially telling the server, "Everything I tell you from now on will be authenticated (and encrypted if encryption parameters were present in the server certificate)." The ChangeCipherSpec is itself a record-level protocol with content type of 20.
   * Finally, the client sends an authenticated and encrypted **Finished** message, containing a hash and MAC over the previous handshake messages.
   * The server will attempt to decrypt the client's *Finished* message and verify the hash and MAC. If the decryption or verification fails, the handshake is considered to have failed and the connection should be torn down.
3. Finally, the server sends a **ChangeCipherSpec**, telling the client, "Everything I tell you from now on will be authenticated (and encrypted, if encryption was negotiated)."
   * The server sends its authenticated and encrypted **Finished** message.
   * The client performs the same decryption and verification.
4. Application phase: at this point, the "handshake" is complete and the application protocol is enabled, with content type of 23. Application messages exchanged between client and server will also be authenticated and optionally encrypted exactly like in their *Finished* message. Otherwise, the content type will return 25 and the client will not authenticate.

**Client-authenticated TLS handshake**

The following *full* example shows a client being authenticated (in addition to the server like above) via TLS using certificates exchanged between both peers.

1. Negotiation Phase:
   * A client sends a **ClientHello** message specifying the highest TLS protocol version it supports, a random number, a list of suggested cipher suites and compression methods.
   * The server responds with a **ServerHello** message, containing the chosen protocol version, a random number, cipher suite and compression method from the choices offered by the client. The server may also send a *session id* as part of the message to perform a resumed handshake.
   * The server sends its **Certificate** message (depending on the selected cipher suite, this may be omitted by the server).[[31]](http://en.wikipedia.org/wiki/Transport_Layer_Security#cite_note-openpgp-31)
   * The server requests a certificate from the client, so that the connection can be mutually authenticated, using a **CertificateRequest** message.
   * The server sends a **ServerHelloDone** message, indicating it is done with handshake negotiation.
   * The client responds with a **Certificate** message, which contains the client's certificate.
   * The client sends a **ClientKeyExchange** message, which may contain a *PreMasterSecret*, public key, or nothing. (Again, this depends on the selected cipher.) This *PreMasterSecret* is encrypted using the public key of the server certificate.
   * The client sends a **CertificateVerify** message, which is a signature over the previous handshake messages using the client's certificate's private key. This signature can be verified by using the client's certificate's public key. This lets the server know that the client has access to the private key of the certificate and thus owns the certificate.
   * The client and server then use the random numbers and *PreMasterSecret* to compute a common secret, called the "master secret". All other key data for this connection is derived from this master secret (and the client- and server-generated random values), which is passed through a carefully designed pseudorandom function.
2. The client now sends a **ChangeCipherSpec** record, essentially telling the server, "Everything I tell you from now on will be authenticated (and encrypted if encryption was negotiated). " The ChangeCipherSpec is itself a record-level protocol and has type 20 and not 22.
   * Finally, the client sends an encrypted **Finished** message, containing a hash and MAC over the previous handshake messages.
   * The server will attempt to decrypt the client's *Finished* message and verify the hash and MAC. If the decryption or verification fails, the handshake is considered to have failed and the connection should be torn down.
3. Finally, the server sends a **ChangeCipherSpec**, telling the client, "Everything I tell you from now on will be authenticated (and encrypted if encryption was negotiated). "
   * The server sends its own encrypted **Finished** message.
   * The client performs the same decryption and verification.
4. Application phase: at this point, the "handshake" is complete and the application protocol is enabled, with content type of 23. Application messages exchanged between client and server will also be encrypted exactly like in their *Finished* message.

**Resumed TLS handshake**

Public key operations (e. g., RSA) are relatively expensive in terms of computational power. TLS provides a secure shortcut in the handshake mechanism to avoid these operations. In an ordinary *full* handshake, the server sends a *session id* as part of the **ServerHello** message. The client associates this*session id* with the server's IP address and TCP port, so that when the client connects again to that server, it can use the *session id* to shortcut the handshake. In the server, the *session id* maps to the cryptographic parameters previously negotiated, specifically the "master secret". Both sides must have the same "master secret" or the resumed handshake will fail (this prevents an eavesdropper from using a *session id*). The random data in the**ClientHello** and **ServerHello** messages virtually guarantee that the generated connection keys will be different than in the previous connection. In the RFCs, this type of handshake is called an *abbreviated* handshake. It is also described in the literature as a *restart* handshake.

1. Negotiation phase:
   * A client sends a **ClientHello** message specifying the highest TLS protocol version it supports, a random number, a list of suggested cipher suites and compression methods. Included in the message is the *session id* from the previous TLS connection.
   * The server responds with a **ServerHello** message, containing the chosen protocol version, a random number, cipher suite and compression method from the choices offered by the client. If the server recognizes the *session id* sent by the client, it responds with the same *session id*. The client uses this to recognize that a resumed handshake is being performed. If the server does not recognize the *session id* sent by the client, it sends a different value for its *session id*. This tells the client that a resumed handshake will not be performed. At this point, both the client and server have the "master secret" and random data to generate the key data to be used for this connection.
2. The server now sends a **ChangeCipherSpec** record, essentially telling the client, "Everything I tell you from now on will be encrypted. " The ChangeCipherSpec is itself a record-level protocol and has type 20 and not 22.
   * Finally, the server sends an encrypted **Finished** message, containing a hash and MAC over the previous handshake messages.
   * The client will attempt to decrypt the server's *Finished* message and verify the hash and MAC. If the decryption or verification fails, the handshake is considered to have failed and the connection should be torn down.
3. Finally, the client sends a **ChangeCipherSpec**, telling the server, "Everything I tell you from now on will be encrypted. "
   * The client sends its own encrypted **Finished** message.
   * The server performs the same decryption and verification.
4. Application phase: at this point, the "handshake" is complete and the application protocol is enabled, with content type of 23. Application messages exchanged between client and server will also be encrypted exactly like in their *Finished* message.

Apart from the performance benefit, resumed sessions can also be used for single sign-on as it is guaranteed that both the original session as well as any resumed session originate from the same client. This is of particular importance for the [FTP over TLS/SSL](http://en.wikipedia.org/wiki/FTPS) protocol which would otherwise suffer from a man in the middle attack in which an attacker could intercept the contents of the secondary data connections.[[32]](http://en.wikipedia.org/wiki/Transport_Layer_Security#cite_note-32)

**Is one more secure than the other?**

While SSL and TLS differ in ways that make them inoperable with each other, they are generally considered *equal* in terms of security. The main difference is that, while SSL connections begin with security and proceed directly to secured communications, TLS connections first begin with an insecure “hello” to the server and only switch to secured communications after the handshake between the client and the server is successful. If the TLS handshake fails for any reason, the connection is never created.

Both Internet security protocols ensure that your data is encrypted as it is transmitted across the Internet.  They also both enable you to be sure that the server that you are communication with is the server you intend to contact and not some “middle man eavesdropper”.  This is possible because servers that support SSL and TLS must have certificates issued to them by a trusted third party, like Verisign or Thawte.  These certificates verify that the domain name they are issued for really belongs to the server.  Your computer will issue warnings to you if you try to connect to a server and the certificate that it gets back is not trusted or doesn’t match the site you are trying to connect to.

If you are mostly concerned about your level of security, you can’t really go wrong choosing either SSL or TLS.

**So then, should I choose to connect with TLS or SSL?**

The main benefit in opting for TLS over SSL is that TLS was incepted as an open-community standard, meaning TLS is more extensible and will likely be more widely supported in the future with other Internet standards. TLS is even backwards compatible, possessing the ability to “scale down” to SSL if necessary to support secure client-side connections that only understand SSL.

Another more immediate benefit, however, is that TLS allows both secure and insecure connections over the same port, whereas SSL requires a designated secure-only port. For users connecting to an email server via POP or IMAP, this means that using TLS will allow you to opt for secure connections but easily switch to insecure connections if necessary without needing to change ports. This is not possible with SSL.

However, as discussed in the previous section, it really doesn’t matter which one is used in terms of security.

Bron bovenstaande twee alinea’s: <http://luxsci.com/blog/ssl-versus-tls-whats-the-difference.html>

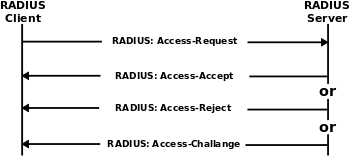
#### BEAST attack

On September 23, 2011 researchers Thai Duong and Juliano Rizzo demonstrated a "proof of concept" called **BEAST** ("Browser Exploit Against SSL/TLS") using a [Java applet](http://en.wikipedia.org/wiki/Java_applet) to violate [same origin policy](http://en.wikipedia.org/wiki/Same_origin_policy) constraints, for a long-known [Cipher block chaining](http://en.wikipedia.org/wiki/Cipher_block_chaining) (CBC) vulnerability in TLS 1.0.[[8]](http://en.wikipedia.org/wiki/Transport_Layer_Security#cite_note-8)[[9]](http://en.wikipedia.org/wiki/Transport_Layer_Security#cite_note-9)

**Public/private key:**

The distinguishing technique used in public-key cryptography is the use of asymmetric key algorithms, where the [key](http://en.wikipedia.org/wiki/Key_(cryptography)) used to [encrypt](http://en.wikipedia.org/wiki/Encryption) a message is not the same as the key used to [decrypt](http://en.wikipedia.org/wiki/Decryption) it. Each user has a pair of [cryptographic keys](http://en.wikipedia.org/wiki/Cryptographic_key) – a **public encryption key** and a **private decryption key**. The publicly available encrypting-key is widely distributed, while the private decrypting-key is known only to its proprietor, the recipient. Messages are encrypted with the recipient's public key, and can be decrypted*only* with the corresponding private key. The keys are related mathematically, but the parameters are chosen so that determining the private key from the public key is either impossible or prohibitively expensive. The discovery of algorithms that could produce public/private key pairs [revolutionized](http://en.wikipedia.org/wiki/History_of_cryptography#Public_key) the practice of cryptography, beginning in the mid-1970s.

# RADIUS



**Remote Authentication Dial In User Service** (**RADIUS**) is a networking [protocol](http://en.wikipedia.org/wiki/Communications_protocol) that provides centralized Authentication, Authorization, management for computers to connect and use a network service. RADIUS was developed by Livingston Enterprises, Inc., in 1991 as an access server authentication and accounting protocol and later brought into the [Internet Engineering Task Force](http://en.wikipedia.org/wiki/Internet_Engineering_Task_Force) (IETF) standards.[[1]](http://en.wikipedia.org/wiki/RADIUS#cite_note-Vollbrecht2006-1)

Because of the broad support and the ubiquitous nature of the RADIUS protocol, it is often used by [ISPs](http://en.wikipedia.org/wiki/ISP) and enterprises to manage access to the [Internet](http://en.wikipedia.org/wiki/Internet) or internal [networks](http://en.wikipedia.org/wiki/Computer_network), [wireless networks](http://en.wikipedia.org/wiki/Wireless_network), and integrated e-mail services. These networks may incorporate [modems](http://en.wikipedia.org/wiki/Modem), [DSL](http://en.wikipedia.org/wiki/DSL), [access points](http://en.wikipedia.org/wiki/Wireless_access_point), [VPNs](http://en.wikipedia.org/wiki/VPN), [network ports](http://en.wikipedia.org/wiki/Network_port), [web servers](http://en.wikipedia.org/wiki/Web_server), etc.[[2]](http://en.wikipedia.org/wiki/RADIUS#cite_note-2)

RADIUS is a client/server protocol that runs in the [application layer](http://en.wikipedia.org/wiki/Application_Layer), using [UDP](http://en.wikipedia.org/wiki/User_Datagram_Protocol) as transport. The [Remote Access Server](http://en.wikipedia.org/wiki/Remote_Access_Server), the [Virtual Private Network server](http://en.wikipedia.org/wiki/Virtual_Private_Network_server), the [Network switch](http://en.wikipedia.org/wiki/Network_switch) with port-based authentication, and the [Network Access Server (NAS)](http://en.wikipedia.org/wiki/Network_Access_Server), are all gateways that control access to the network, and all have a RADIUS client component that communicates with the RADIUS server. The RADIUS server is usually a background process running on a UNIX or Microsoft Windows server.[[3]](http://en.wikipedia.org/wiki/RADIUS#cite_note-3) RADIUS serves three functions:

1. to authenticate users or devices before granting them access to a network,
2. to authorize those users or devices for certain network services and
3. to account for usage of those services.

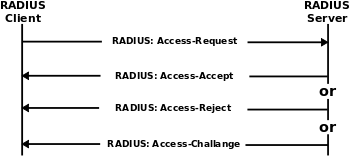
**Authentication and authorization**

The user or machine sends a request to a [Remote Access Server](http://en.wikipedia.org/wiki/Remote_Access_Server) (RAS) to gain access to a particular network resource using access credentials. The credentials are passed to the RAS device via the [link-layer](http://en.wikipedia.org/wiki/Link_Layer) protocol - for example, [Point-to-Point Protocol](http://en.wikipedia.org/wiki/Point-to-Point_Protocol) (PPP) in the case of many [dialup](http://en.wikipedia.org/wiki/Dialup) or [DSL](http://en.wikipedia.org/wiki/Digital_subscriber_line)providers or posted in an [HTTPS](http://en.wikipedia.org/wiki/HTTPS) secure web form.

In turn, the RAS sends a RADIUS *Access Request* message to the RADIUS server, requesting authorization to grant access via the RADIUS protocol.[[4]](http://en.wikipedia.org/wiki/RADIUS#cite_note-4)

This request includes access credentials, typically in the form of [username](http://en.wikipedia.org/wiki/Username) and [password](http://en.wikipedia.org/wiki/Password) or security certificate provided by the user. Additionally, the request may contain other information which the RAS knows about the user, such as its network address or phone number, and information regarding the user's physical point of attachment to the RAS.

The RADIUS server checks that the information is correct using authentication schemes such as [PAP](http://en.wikipedia.org/wiki/Password_authentication_protocol), [CHAP](http://en.wikipedia.org/wiki/Challenge-handshake_authentication_protocol) or [EAP](http://en.wikipedia.org/wiki/Extensible_Authentication_Protocol). The user's proof of identification is verified, along with, optionally, other information related to the request, such as the user's network address or phone number, account status, and specific network service access privileges. Historically, RADIUS servers checked the user's information against a locally stored flat file database. Modern RADIUS servers can do this, or can refer to external sources — commonly [SQL](http://en.wikipedia.org/wiki/SQL), [Kerberos](http://en.wikipedia.org/wiki/Kerberos_(protocol)), [LDAP](http://en.wikipedia.org/wiki/LDAP), or [Active Directory](http://en.wikipedia.org/wiki/Active_Directory) servers — to verify the user's credentials.



RADIUS Authentication and Authorization Flow

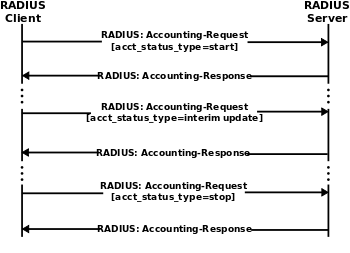
The RADIUS server then returns one of three responses to the RAS : 1) Access Reject, 2) Access Challenge, or 3) Access Accept.

* *Access Reject* - The user is unconditionally denied access to all requested network resources. Reasons may include failure to provide proof of identification or an unknown or inactive user account.
* *Access Challenge* - Requests additional information from the user such as a secondary password, PIN, token, or card. Access Challenge is also used in more complex authentication dialogs where a secure tunnel is established between the user machine and the Radius Server in a way that the access credentials are hidden from the RAS.
* *Access Accept* - The user is granted access. Once the user is authenticated, the RADIUS server will often check that the user is authorized to use the network service requested. A given user may be allowed to use a company's wireless network, but not its VPN service, for example. Again, this information may be stored locally on the RADIUS server, or may be looked up in an external source such as LDAP or Active Directory.

Each of these three RADIUS responses may include a Reply-Message attribute which may give a reason for the rejection, the prompt for the challenge, or a welcome message for the accept. The text in the attribute can be passed on to the user in a return web page.

Authorization [attributes](http://en.wikipedia.org/wiki/Radius_Values) are conveyed to the RAS stipulating terms of access to be granted. For example, the following authorization attributes may be included in an Access-Accept:

* The specific [IP address](http://en.wikipedia.org/wiki/IP_address) to be assigned to the user
* The address pool from which the user's IP should be chosen
* The maximum length that the user may remain connected
* An access list, priority queue or other restrictions on a user's access
* [L2TP](http://en.wikipedia.org/wiki/L2TP) parameters
* VLAN parameters
* Quality of Service (QoS) parameters



**Accounting**



RADIUS Accounting Flow

Accounting is described in [RFC 2866](http://tools.ietf.org/html/rfc2866).

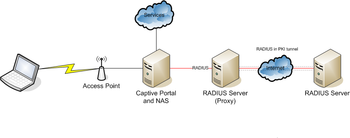
When network access is granted to the user by the [NAS](http://en.wikipedia.org/wiki/Network_access_server), an *Accounting Start* (a RADIUS Accounting Request packet containing an Acct-Status-Type attribute with the value "start") is sent by the NAS to the RADIUS server to signal the start of the user's network access. "Start" records typically contain the user's identification, network address, point of attachment and a unique session identifier.[[5]](http://en.wikipedia.org/wiki/RADIUS#cite_note-5)

Periodically, *Interim Update* records (a RADIUS Accounting Request packet containing an Acct-Status-Type attribute with the value "interim-update") may be sent by the NAS to the RADIUS server, to update it on the status of an active session. "Interim" records typically convey the current session duration and information on current data usage.

Finally, when the user's network access is closed, the NAS issues a final *Accounting Stop* record (a RADIUS Accounting Request packet containing an Acct-Status-Type attribute with the value "stop") to the RADIUS server, providing information on the final usage in terms of time, packets transferred, data transferred, reason for disconnect and other information related to the user's network access.

Typically, the client sends Accounting-Request packets until it receives an Accounting-Response acknowledgement, using some retry interval.

The primary purpose of this data is that the user can be [billed](http://en.wikipedia.org/wiki/Bill_(payment)) accordingly; the data is also commonly used for [statistical](http://en.wikipedia.org/wiki/Statistical) purposes and for general network monitoring...



**Roaming**

Roaming using a proxy RADIUS AAA server.

RADIUS is commonly used to facilitate [roaming](http://en.wikipedia.org/wiki/Roaming) between [ISPs](http://en.wikipedia.org/wiki/Internet_service_provider), for example:

* by companies which provide a single global set of credentials that are usable on many public networks;
* by independent, but collaborating, institutions issuing their own credentials to their own users, that allow a visitor from one to another to be authenticated by their home institution, such as in [eduroam](http://en.wikipedia.org/wiki/Eduroam).

RADIUS facilitates this by the use of *realms*, which identify where the RADIUS server should forward the AAA requests for processing.

**Realms**

A realm is commonly appended to a user's user name and delimited with an '@' sign, resembling an email address domain name. This is known as *postfix*notation for the realm. Another common usage is *prefix* notation, which involves prepending the realm to the username and using '\' as a delimiter. Modern RADIUS servers allow any character to be used as a realm delimiter, although in practice '@' and '\' are usually used.

Realms can also be compounded using both prefix and [postfix](http://en.wikipedia.org/wiki/Postfix) notation, to allow for complicated roaming scenarios; for example, somedomain.com\username@anotherdomain.com could be a valid username with two realms.

Although realms often resemble domains, it is important to note that realms are in fact arbitrary text and need not contain real domain names.

**Proxy operations**

When a RADIUS server receives an AAA request for a user name containing a realm, the server will reference a table of configured realms. If the realm is known, the server will then *proxy* the request to the configured home server for that domain. The behaviour of the proxying server regarding the removal of the realm from the request ("stripping") is configuration-dependent on most servers. In addition, the proxying server can be configured to add, remove or rewrite AAA requests when they are proxied.

**Security**

Roaming with RADIUS exposes the users to various security and privacy concerns. More generally, some roaming partners establish a secure tunnel between the RADIUS servers to ensure that users' credentials cannot be intercepted while being proxied across the internet. This is a concern as the MD5 hash built into RADIUS is considered insecure.[[6]](http://en.wikipedia.org/wiki/RADIUS#cite_note-6)

# EAP

**Extensible Authentication Protocol**, or **EAP**, is an [authentication](http://en.wikipedia.org/wiki/Authentication) framework frequently used in [wireless networks](http://en.wikipedia.org/wiki/Wireless_LAN) and [Point-to-Point connections](http://en.wikipedia.org/wiki/Point-to-Point_Protocol). It is defined in [RFC 3748](http://tools.ietf.org/html/rfc3748), which made [RFC 2284](http://tools.ietf.org/html/rfc2284) obsolete, and was updated by [RFC 5247](http://tools.ietf.org/html/rfc5247).

EAP is an authentication framework providing for the transport and usage of [keying material and parameters](http://en.wikipedia.org/wiki/Key_(cryptography)) generated by EAP methods.[[1]](http://en.wikipedia.org/wiki/Extensible_Authentication_Protocol#cite_note-rfc3748_sec1-1) There are many methods defined by RFCs and a number of vendor specific methods and new proposals exist. EAP is not a [wire protocol](http://en.wikipedia.org/wiki/Wire_protocol); instead it only defines message formats. Each protocol that uses EAP defines a way to [encapsulate](http://en.wikipedia.org/wiki/Encapsulation_(networking)) EAP messages within that protocol's messages.

EAP is in wide use. For example, in [IEEE 802.11](http://en.wikipedia.org/wiki/IEEE_802.11) (WiFi) the [WPA](http://en.wikipedia.org/wiki/Wi-Fi_Protected_Access) and [WPA2](http://en.wikipedia.org/wiki/WPA2) standards have adopted [IEEE 802.1X](http://en.wikipedia.org/wiki/IEEE_802.1X) with five EAP types as the official authentication mechanisms.

EAP is an authentication framework, not a specific authentication mechanism.[[1]](http://en.wikipedia.org/wiki/Extensible_Authentication_Protocol#cite_note-rfc3748_sec1-1) It provides some common functions and negotiation of authentication methods called EAP methods. There are currently about 40 different methods defined. Methods defined in [IETF](http://en.wikipedia.org/wiki/IETF) RFCs include [EAP-MD5](http://en.wikipedia.org/wiki/EAP-MD5), [EAP-POTP](http://en.wikipedia.org/w/index.php?title=EAP-POTP&action=edit&redlink=1),[EAP-GTC](http://en.wikipedia.org/wiki/EAP-GTC), [EAP-TLS](http://en.wikipedia.org/wiki/EAP-TLS), [EAP-IKEv2](http://en.wikipedia.org/wiki/EAP-IKEv2), [EAP-SIM](http://en.wikipedia.org/wiki/EAP-SIM), [EAP-AKA](http://en.wikipedia.org/wiki/EAP-AKA) and [EAP-AKA'](http://en.wikipedia.org/wiki/EAP-AKA%27), and in addition a number of vendor specific methods and new proposals exist. Commonly used modern methods capable of operating in wireless networks include EAP-TLS, [EAP-SIM](http://en.wikipedia.org/wiki/EAP-SIM), [EAP-AKA](http://en.wikipedia.org/wiki/EAP-AKA), [LEAP](http://en.wikipedia.org/wiki/Lightweight_Extensible_Authentication_Protocol) and [EAP-TTLS](http://en.wikipedia.org/wiki/EAP-TTLS). Requirements for EAP methods used in wireless LAN authentication are described in [RFC 4017](http://tools.ietf.org/html/rfc4017).

The standard also describes the conditions under which the AAA key management requirements described in [RFC 4962](http://tools.ietf.org/html/rfc4962) can be satisfied.

**EAP-TLS**

EAP-Transport Layer Security (EAP-TLS), defined in [RFC 5216](http://tools.ietf.org/html/rfc5216), is an [IETF](http://en.wikipedia.org/wiki/IETF) [open standard](http://en.wikipedia.org/wiki/Open_standard) that uses the [Transport Layer Security](http://en.wikipedia.org/wiki/Transport_Layer_Security) (TLS) protocol, and is well-supported among wireless vendors. The security of TLS is strong, provided the user understands potential warnings about false credentials. It uses[PKI](http://en.wikipedia.org/wiki/Public_Key_Infrastructure) to secure communication to a [RADIUS](http://en.wikipedia.org/wiki/RADIUS) authentication server or another type of authentication server.

EAP-TLS is the original, standard wireless LAN EAP authentication protocol. Although it is rarely deployed, it is still considered one of the most secure EAP standards available and is universally supported by all manufacturers of wireless LAN hardware and software. The requirement for a client-side certificate, however unpopular it may be, is what gives EAP-TLS its authentication strength and illustrates the classic convenience vs. security trade-off. A compromised password is not enough to break into EAP-TLS enabled systems because the intruder still needs to have the client-side private key. The highest security available is when client-side keys are housed in [smart cards](http://en.wikipedia.org/wiki/Smart_card).[[4]](http://en.wikipedia.org/wiki/Extensible_Authentication_Protocol#cite_note-4) This is because there is no way to steal a certificate's corresponding private key from a smart card without stealing the card itself. It is significantly more likely that the physical theft of a smart card would be noticed (and the smart card immediately revoked) than a (typical) password theft would be noticed. Up until April 2005, EAP-TLS was the only

EAP type vendors needed to certify for a WPA or WPA2 logo.[[5]](http://en.wikipedia.org/wiki/Extensible_Authentication_Protocol#cite_note-5) There are client and server implementations of EAP-TLS in 3Com, Apple, [Avaya](http://en.wikipedia.org/wiki/Avaya), Brocade Communications, Cisco, Enterasys Networks, Foundry, Hirschmann, HP, Juniper, and Microsoft, and open source operating systems. EAP-TLS is natively supported in Mac OS X 10.3 and above, [wpa\_supplicant](http://en.wikipedia.org/wiki/Wpa_supplicant), Windows 2000 SP4, Windows XP and above, Windows Mobile 2003 and above, Windows CE 4.2, and Apple's iOS mobile operating system.

Unlike most TLS implementations of [HTTPS](http://en.wikipedia.org/wiki/HTTPS) like major [web browsers](http://en.wikipedia.org/wiki/Web_browser), most TLS implementations of EAP-TLS require the client to use [X.509](http://en.wikipedia.org/wiki/X.509) certificates, which some have identified as potentially dramatically reducing adoption of EAP-TLS.[[6]](http://en.wikipedia.org/wiki/Extensible_Authentication_Protocol#cite_note-6) As of 22 August 2012, [hostapd](http://en.wikipedia.org/wiki/Hostapd) (and wpa\_supplicant) added support in its [Git](http://en.wikipedia.org/wiki/Git_(software)) repository for an UNAUTH-TLS vendor specific EAP type which only does server authentication.[[7]](http://en.wikipedia.org/wiki/Extensible_Authentication_Protocol#cite_note-7) This will allow for situations much like HTTPS, where for instance a wireless hotspot allows free access and does not authenticate station clients, but station clients wish to use encryption ([IEEE 802.11i-2004](http://en.wikipedia.org/wiki/IEEE_802.11i-2004) i.e. [WPA2](http://en.wikipedia.org/wiki/WPA2)) and potentially authenticate the wireless hotspot.

**EAP-TTLS**

EAP-Tunneled Transport Layer Security (EAP-TTLS) is an EAP protocol that extends [TLS](http://en.wikipedia.org/wiki/Transport_Layer_Security). It was co-developed by [Funk Software](http://en.wikipedia.org/wiki/Funk_Software) and [Certicom](http://en.wikipedia.org/wiki/Certicom). It is widely supported across platforms. Microsoft did not incorporate native support for the EAP-TTLS protocol in [Windows XP](http://en.wikipedia.org/wiki/Windows_XP), Vista, or 7. Supporting TTLS on these platforms requires a third-party ECP certified software. [Microsoft Windows](http://en.wikipedia.org/wiki/Microsoft_Windows) starts supporting EAP-TTLS with [Windows 8](http://en.wikipedia.org/wiki/Windows_8).[[11]](http://en.wikipedia.org/wiki/Extensible_Authentication_Protocol#cite_note-11)

The client can but does not have to be authenticated via a [CA](http://en.wikipedia.org/wiki/Certificate_Authority)-signed [PKI](http://en.wikipedia.org/wiki/Public_key_infrastructure) certificate to the server. This greatly simplifies the setup procedure as a certificate does not need to be installed on every client.

After the server is securely authenticated to the client via its CA certificate and optionally the client to the server, the server can then use the established secure connection ("tunnel") to authenticate the client. It can use an existing and widely deployed authentication protocol and infrastructure, incorporating legacy password mechanisms and authentication databases, while the secure tunnel provides protection from [eavesdropping](http://en.wikipedia.org/wiki/Eavesdropping) and [man-in-the-middle attack](http://en.wikipedia.org/wiki/Man-in-the-middle_attack). Note that the user's name is never transmitted in unencrypted cleartext, thus improving privacy.

Two distinct versions of EAP-TTLS exist: original EAP-TTLS (a.k.a. EAP-TTLSv0) and EAP-TTLSv1. EAP-TTLSv0 is described in [RFC 5281](http://tools.ietf.org/html/rfc5281), EAP-TTLSv1 is available as an Internet draft.[[12]](http://en.wikipedia.org/wiki/Extensible_Authentication_Protocol#cite_note-12)

**RADIUS and Diameter**

*Main articles:*[*RADIUS*](http://en.wikipedia.org/wiki/RADIUS)*and*[*Diameter (protocol)*](http://en.wikipedia.org/wiki/Diameter_(protocol))

Both the [RADIUS](http://en.wikipedia.org/wiki/RADIUS) and [Diameter](http://en.wikipedia.org/wiki/Diameter_(protocol)) [AAA protocols](http://en.wikipedia.org/wiki/AAA_protocol) can encapsulate EAP messages. They are often used by [Network Access Server](http://en.wikipedia.org/wiki/Network_Access_Server) (NAS) devices to forward EAP packets between IEEE 802.1X endpoints and AAA servers to facilitate IEEE 802.1X.

# Eduroam:

**eduroam** (**edu**cation **roam**ing) is a secure international [roaming](http://en.wikipedia.org/wiki/Roaming) service for users in [higher education](http://en.wikipedia.org/wiki/Higher_education). The European eduroam confederation (a confederation of autonomous roaming services) is based on a set of defined organisational and technical requirements that each member of the confederation must agree to (by signing the eduroam policy GN2-07-328) and follow.[[1]](http://en.wikipedia.org/wiki/Eduroam#cite_note-1)

**RADIUS**

eduroam requires participating [RADIUS](http://en.wikipedia.org/wiki/RADIUS) servers to support realms. Within a single institution a user would usually simply be represented by a [username](http://en.wikipedia.org/wiki/Username)and corresponding [password](http://en.wikipedia.org/wiki/Password) for their [authentication](http://en.wikipedia.org/wiki/Authentication). By contrast a visitor using eduroam requires a representation of their home institution as 'username@realm', where the 'realm' is usually closely related to the visitor's home [DNS](http://en.wikipedia.org/wiki/Domain_Name_System) name.

The role of the RADIUS hierarchy is to forward a user's credentials to their home institution for authentication. The RADIUS server at a participating institution recognises the '@realm' component and proxies anything non-local to its national top-level RADIUS (NTLR) service, which is normally operated by the National Research and Education Network ([NREN](http://en.wikipedia.org/wiki/NREN)) of that country and which has a complete list of the participating eduroam institutions in that country.

For international roaming, a regional top-level RADIUS server is needed in order to roam the users request to the right country.

Not all RADIUS servers are capable of offering the '@realm' eduroam support. For instance, commercial products from some USA-based suppliers, such as Infoblox, lack this capability.

# SpaceNet:

http://www.myservicestar.com/myservicestar/White%20Papers/tct\_whitepapers/spacenet/ssl-acceleration.pdf

http://www.spacenet.com/services/managed-services/

# Security Assertion Markup Language (SAML)

**Security Assertion Markup Language** (**SAML**, pronounced "sam-el"[[1]](http://en.wikipedia.org/wiki/Security_Assertion_Markup_Language#cite_note-1)) is an [XML](http://en.wikipedia.org/wiki/XML)-based [open standard](http://en.wikipedia.org/wiki/Open_standard) data format for exchanging [authentication](http://en.wikipedia.org/wiki/Authentication) and[authorization](http://en.wikipedia.org/wiki/Authorization) data between parties, in particular, between an [identity provider](http://en.wikipedia.org/wiki/Identity_provider) and a [service provider](http://en.wikipedia.org/wiki/Service_provider). SAML is a product of the [OASIS](http://en.wikipedia.org/wiki/OASIS_(organization)) Security Services Technical Committee. SAML dates from 2001; the most recent update of SAML is from 2005.

The single most important problem that SAML addresses is the [web browser](http://en.wikipedia.org/wiki/Web_browser) [single sign-on](http://en.wikipedia.org/wiki/Single_sign-on) (SSO) problem. [Single sign-on](http://en.wikipedia.org/wiki/Single_sign_on) solutions are abundant at the[intranet](http://en.wikipedia.org/wiki/Intranet) level (using [cookies](http://en.wikipedia.org/wiki/HTTP_cookie), for example) but extending these solutions beyond the intranet has been problematic and has led to the proliferation of non-interoperable proprietary technologies. (Another more recent approach to addressing the browser SSO problem is the [OpenID](http://en.wikipedia.org/wiki/OpenID) protocol.)

The SAML specification defines three roles: the principal (typically a user), the identity provider (aka IdP), and the service provider (aka SP). In the use case addressed by SAML, the principal requests a service from the service provider. The service provider requests and obtains an identity assertion from the identity provider. On the basis of this assertion, the service provider can make an [access control](http://en.wikipedia.org/wiki/Access_control) decision - in other words it can decide whether to perform some service for the connected principal.

Before delivering the identity assertion to the SP, the IdP may request some information from the principal - such as a user name and password - in order to authenticate the principal. SAML specifies the assertions between the three parties: in particular, the messages that assert identity that are passed from the IdP to the SP. In SAML, one identity provider may provide SAML assertions to many service providers. Conversely, one SP may rely on and trust assertions from many independent IdPs.

SAML does not specify the implementation of the identity provider service; it may use a username/password, it may use multifactor authentication, it may have an opaque implementation. A company's directory service, which allows users to login with a user name and password, is a typical example of an identity provider. Any of the popular common internet social services also provide identity services that in theory could be used to support SAML exchanges.

**SAML building blocks**

SAML is built upon a number of existing standards:

**Extensible Markup Language (XML)**

Most SAML exchanges are expressed in a standardized dialect of [XML](http://en.wikipedia.org/wiki/XML), which is the root for the name SAML (Security Assertion Markup Language).

**XML Schema**

SAML assertions and protocols are specified (in part) using [XML Schema](http://en.wikipedia.org/wiki/XML_Schema_(W3C)).

**XML Signature**

Both [SAML 1.1](http://en.wikipedia.org/wiki/SAML_1.1) and [SAML 2.0](http://en.wikipedia.org/wiki/SAML_2.0) use digital signatures (based on the [XML Signature](http://en.wikipedia.org/wiki/XML_Signature) standard) for authentication and message integrity.

**XML Encryption**

Using [XML Encryption](http://en.wikipedia.org/wiki/XML_Encryption), [SAML 2.0](http://en.wikipedia.org/wiki/SAML_2.0) provides elements for encrypted name identifiers, encrypted attributes, and encrypted assertions (SAML 1.1 does not have encryption capabilities).

**Hypertext Transfer Protocol (HTTP)**

SAML relies heavily on [HTTP](http://en.wikipedia.org/wiki/HTTP) as its communications protocol.

**SOAP**

SAML specifies the use of [SOAP](http://en.wikipedia.org/wiki/SOAP_(protocol)), specifically [SOAP 1.1](http://www.w3.org/TR/2000/NOTE-SOAP-20000508/).

**SAML Assertions**

A SAML *assertion* contains a packet of security information:

<**saml:Assertion** ...> ... <**/saml:Assertion**>.

Loosely speaking, a relying party interprets an assertion as follows:

Assertion *A* was issued at time *t* by issuer *R* regarding subject *S* provided conditions *C* are valid.

SAML assertions are usually transferred from identity providers to service providers. Assertions contain *statements* that service providers use to make access-control decisions. Three types of statements are provided by SAML:

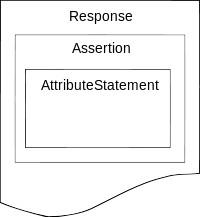
1. **Authentication statements**
2. **Attribute statements**
3. **Authorization decision statements**

*Authentication statements* assert to the service provider that the principal did indeed authenticate with the identity provider at a particular time using a particular method of authentication. Other information about the authenticated principal (called the *authentication context*) may be disclosed in an authentication statement.

An *attribute statement* asserts that a subject is associated with certain attributes. An *attribute* is simply a name-value pair. Relying parties use attributes to make access-control decisions.

An *authorization decision statement* asserts that a subject is permitted to perform action *A* on resource *R* given evidence *E*. The expressiveness of authorization decision statements in SAML is intentionally limited. More-advanced use cases are encouraged to use [XACML](http://en.wikipedia.org/wiki/XACML) instead.

[[edit](http://en.wikipedia.org/w/index.php?title=Security_Assertion_Markup_Language&action=edit&section=9)]**SAML protocols**



A SAML *protocol* describes how certain SAML elements (including assertions) are packaged within SAML request and response elements, and gives the processing rules that SAML entities must follow when producing or consuming these elements. For the most part, a SAML protocol is a simple request-response protocol.

The most important type of SAML protocol request is called a *query*. A service provider makes a query directly to an identity provider over a secure back channel. Thus query messages are typically bound to SOAP.

Corresponding to the three types of statements, there are three types of SAML queries:

1. **Authentication query**
2. **Attribute query**
3. **Authorization decision query**

Of these, the *attribute query* is perhaps most important (and still the object of much research). The result of an attribute query is a SAML response containing an assertion, which itself contains an attribute statement. See the SAML 2.0 topic for [an example of attribute query/response](http://en.wikipedia.org/wiki/SAML_2.0#SAML_Attribute_Query).

# OpenSocial

**OpenSocial** is a public specification that defines a component hosting environment (container) and a set of common [application programming interfaces](http://en.wikipedia.org/wiki/Application_programming_interface) (APIs) for [web](http://en.wikipedia.org/wiki/World_Wide_Web)-based applications. Initially it was designed for [social network applications](http://en.wikipedia.org/wiki/Social_network_service) and was developed by [Google](http://en.wikipedia.org/wiki/Google) along with [MySpace](http://en.wikipedia.org/wiki/MySpace) and a number of other social networks. In more recent times it has become adopted as a general use runtime environment for allowing untrusted and partially trusted components from third parties to run in an existing web application. The OpenSocial Foundation has also moved to integrate or support numerous other open web technologies. This includes [Oauth](http://en.wikipedia.org/wiki/Oauth) and OAuth 2.0, [Activity Streams](http://en.wikipedia.org/wiki/Activity_Streams_(format)), and portable contacts, among others.

It was released on November 1, 2007.[[1]](http://en.wikipedia.org/wiki/OpenSocial#cite_note-press_release-1) Applications implementing the OpenSocial APIs will be[interoperable](http://en.wikipedia.org/wiki/Interoperability) with any social network system that supports them.

**Structure**

Based on [HTML](http://en.wikipedia.org/wiki/HTML) and [JavaScript](http://en.wikipedia.org/wiki/JavaScript), as well as the [Google Gadgets](http://en.wikipedia.org/wiki/Google_Gadgets) framework, OpenSocial includes multiple APIs for [social software](http://en.wikipedia.org/wiki/Social_software) applications to access data and core functions on participating [social networks](http://en.wikipedia.org/wiki/Social_network).[[2]](http://en.wikipedia.org/wiki/OpenSocial#cite_note-Andreessen-2) Each API addresses a different aspect.[[3]](http://en.wikipedia.org/wiki/OpenSocial#cite_note-API_documentation-3) It also includes APIs for contacting arbitrary third party services on the web using a proxy system and OAuth for security.

In version 0.9 OpenSocial added support for a tag-based language.[[4]](http://en.wikipedia.org/wiki/OpenSocial#cite_note-OpenSocial_Specification_Release_Notes-4) This language is referred to as OSML and allows tag-based access to data from the OpenSocial APIs that previously required an asynchronous client side request. It also defined a rich tag template system and adopted an expression language loosely based on the Java [Expression Language](http://en.wikipedia.org/wiki/Expression_Language).

Starting in version 2.0 OpenSocial adopted support for Activity Streams format [[4]](http://en.wikipedia.org/wiki/OpenSocial#cite_note-OpenSocial_Specification_Release_Notes-4)

# VOMS

**VOMS** is an acronym used for *Virtual Organization Membership Service* in [grid computing](http://en.wikipedia.org/wiki/Grid_computing). It is structured as a simple account database with fixed formats for the information exchange and features single login, expiration time, backward compatibility, and multiple virtual organizations. The database is manipulated by authorization data that defines specific capabilities and roles for users. Administrative tools can be used by administrators to assign roles and capability information in the database. A command-line tool allows users to generate a local proxy credential based on the contents of the VOMS database. This credential includes the basic authentication information that standard Grid proxy credentials contain, but it also includes role and capability information from the VOMS server. VOMS-aware applications can use the VOMS data to make authentication decisions regarding user requests. VOMS was originally developed by the European DataGrid and [Enabling Grids for E-sciencE](http://en.wikipedia.org/wiki/Enabling_Grids_for_E-sciencE) projects and is now maintained by the [European Middleware Initiative](http://en.wikipedia.org/wiki/European_Middleware_Initiative).