Salt Protocol©® an Identity-Based Authentication Protocol using Synchronized Systems

Pierre Innocent, Member IEEE
Tsert.Com
contact@tsert.com
http://www.tsert.com/

Abstract – The Salt protocol [patent pending] is our approach to the protection of Internet-based communication. Communication entities can reliably recognize each other in an non-private network, like the Internet, or more often referred to as the Web, without requiring a Secure Socket Layer (SSL) handshake and a certificate.

The Salt protocol, is an identity-based authentication protocol. It essentially requires a communication entity to identify itself with a specific access key, a sequence of bytes, generated by a cryptographic engine.

The protocol also requires, that two entities involved in a communication session, must be able to synchronize on a particular salt value, encryption algorithm, a cypher mode, an obfuscation mode, and a set of encryption characters. The set of required information is called the salt-setting.

The protocol also requires, that servers belonging to a given private network, remain synchronized with regards to salt settings, signatures, and user's public encryption keys.

Our approach is, usually, referred to as an N-factor authentication protocol, using the salt-setting, as the shared secret.

The Salt protocol, as with other modern Internet authentication protocols, rely on the Diffie-Hellman-Merkle key exchange, heretofore referred to as the Diffie-Hellman protocol, to initiate a shared secret exchange with unknown peers.

The password text itself is heretofore referred to as the nonce, nonce password, password text, or nonce text; and, the encrypted and salted nonce as the digest.

Index Terms – Protocol, Encryption, Algorithm, Digest, Cryptographic Engine, Multi-Cast, Salt Value, Protocol, SSL, Secure Socket Layer, Diffie-Hellman, Certificate, 2-factor. N-factor, Key Server, IP, FOB, HTTP, MITM, RNG, VPN, SASL.

I. Introduction

How do we secure communication on the current Internet ? Plain text communication is no longer recommended for private communication. Exchanges of user names and passwords can easily be compromised, by hackers with the proper tools.

Encrypted communication can be compromised when the public and private keys are weak, no matter how big the key size. Hackers can also find other ways of getting on your computer, e.g. browser hacks; they can replace your bundled certificates, with fake or weaker ones.

Encryption problems mainly resides in lack of entropy, or a weak random number generator (**RNG**). Keys which are generated with a weak RNG will tend to be weak, and the encrypted data easily decrypted.

Our approach, which is summarized in the solution section, tries to primarily solve the man-in-the-middle (**MITM**) problem.

A. Man-in-the-middle attacks (MITM)

These type of attacks consist of interposing oneself between the two communicating entities/parties. It is usually achieved by intercepting a request, extracting a party's identifying pieces of data, and forwarding the request to the opposite party. The hacker then uses the captured pieces of data to misrepresent the party whose identity was stolen.

Two of the most used methods of establishing secure communication, **2-factor** authentication and public-key communication, are susceptible to **MITM** attacks.

With FOB-based systems, users need to input an authentication code provided with the FOB key; as well as their password to setup a session. FOB-based systems are based on the **2-factor** authentication protocol and as such, can be hacked via **man-in-the-middle** attacks.

MITM attacks can be stopped by requiring the use of **shared secrets**. The shared secrets types of solutions only work if both parties know each other. The weakness is in the **initial** handshake, which we try to solve with an **Invitation to the Dance** approach.

II. Solution

Our approach uses several methods against **MITM** attacks, The first one is obviously the requirement of a shared secret. The shared secret is called the **salt-setting** which encapsulates how a key is to be generated and matched by the other party.

The **salt-setting** only addresses the protection of access keys, user name, and password exchanges. The setting is made difficult to guess with the use of randomization and encryption. Contrary to the FOB-based approach; our approach does not require the party, which initiates the communication session, to remember anything; asides from their user names and passwords (see **Conclusion**.)

The second method is the requirement that each server be, by default, a **public-key server**. Public key. digital signature, and salt-setting management is automatically taken care of by each server, on a given network. The additional requirement is that said servers keep their database synchronized. Communication, between servers, using public key cryptography is done primarily with a user's public key, and defaults to the system key, when the user's key is not available.

The initial exchange, between two unknown peers, is done using the **invitation to the Dance** approach (akin to the old modem callback method, see **Appendix D**). The **SALT** protocol approach is to use an initial Diffie-Hellman exchange from the peer asking to join the dance (the dance is a given Virtual Private Network). For example, a company's supplier may be invited to join the company's **VPN**, but **must** first ask to join. The company server, which receives the supplier's request to join, then makes **UDP** requests to 2-5 of its own peers to verify that the requester/peer is probably who it says it is; by making Diffie-Hellman exchanges of their own, with the supplier's machine. The result of these exchanges are collated, by the server, to perform a verification exchange. The **initial** exchanges must **always** be encrypted; they must be implemented using **SSL** connections, or other strong encryption protocols.

The **SALT** protocol can also be embedded into the initial **SSL** Diffie-Hellman handshake. The server, in this case, would, not only, include its response to the

requester's Diffie-Hellman challenge; but also append its own **verification** challenge, to which the requester must respond, before obtaining a **salt-setting**.
Routers[1], which by default, must synchronize their **IP** address database, could use the **SALT** protocol.

Hackers can defeat the **SALT** protocol's approach to **initial** exchanges, by accessing routers and **spoofing** the peer's **IP** address, so that every route to the peer's communication device is covered.

The **SALT** protocol is not susceptible to **DNS** attacks, since it relies on the host's **IP address**. and not its Fully Qualified Domain Name (**FQDN).**

An implementation may counter verify an **IP** address, by contacting a **DNS** server, with the **FQDN**; but should never rely, solely, on the returned information.

III. Salt setting

The **Salt protocol** relies on the randomization of several pieces of information. The **first** piece is the **algorithm** used for encryption, its **character set**, and its **cypher mode**, if any is applicable to the selected algorithm. The **second** piece is the **Salt key**. The **third** and **fourth** pieces are the **obfuscation** and **digest info;** how to obfuscate the incoming text to be encrypted, and the resulting password to be matched; and the **last** piece is the **salt value** represented as indices into a character set.

The **salt setting** specification is **byte-based**. We chose, arbitrarily, a **20-bytes** quantity plus the salt indices. The setting itself, is always transmitted **off-line**. It is the encrypted password, obfuscated or not, that is transmitted within a session.

A. Algorithms

The algorithms, in use, are the ones that are in the public domain; for example the ones provided in the Unix **gcrypt** library. They include the symmetric and public-key (asymmetric) algorithms; as well as the hashing/signature algorithms.

There are approximately 30+ algorithms to be selected. The algorithm option, in the **salt-setting**, gives you 1**..256** possibilities. **Diffie-Hellman** is another reserved algorithm used, to achieve the same purpose as a text-based encrypted password. Some may use **only** the **Diffie-Hellman** algorithm to implement the **SALT** protocol at the **IP** level; mostly because it is mathematically, and not signature based.

The **SALT** protocol provides a way to allow both private and public communication. **Public** communication through **SALT** protected ports is possible, if and only if, the **reserved** set of algorithms is used; and if settings are **a priori** exchanged.

The **first** byte indicates the encryption algorithm. The algorithm is either from the public or private set, of algorithms. If the value is zero, then the **SHA1-256** signature/hash algorithm is used.

The public-key algorithms can also be used, to generate signatures; because their strength is as high or higher than the symmetric ones. But we truly **saw** no need, to use them in our protocol. If a user is that worried about their communication link; the advice is to simply encrypt the communication stream, using strong (>**2048**) public-key algorithms; while continuing to use symmetric algorithms for the SALT protocol.

When using asymmetric algorithms, the **salt** value that is used with symmetric ones is not used for the actual encryption. In our implementation, we use it as a password to the public/private key set associated with the algorithm setting.

The reserved set of algorithms is, for now, the publicly available ones,; the ones with no associated patent issue. **md5sum** and **sha1** are the ones most used. The public set ranges from 1**..128** possible algorithms. The default is the **sha1** algorithm when the setting equals one (see **Appendix E**.)

The private set of algorithms can be any algorithm you wish to use, in order to turn your network into a fully private one. You may choose to use the public set in a different order; or to slightly perturb the encrypted text of each public algorithm, in your own specific way; or to add your own algorithm to the set. The private set ranges from **129..256,** which is 128 possible algorithms.

1. Character Set

The character set is specified with the **first 4-bit** of the **second** byte.

The character set to use for encryption, is by default the A**scii** set. For the private set, one may choose the **EBCDIC** or an A**scii** representation of the **EBCDIC** set; the **ISO 8859-1** character set; or even the U**nicode UTF8** set of characters, in the **0..255** range.

[1]BGP Border Gateway Protocol

The **public/reserved** set of characters is indexed from 1**..8.** The private set is indexed from **9..16** character sets. The default set is the restricted **Ascii** set [a-zA-z/.], which is mainly used for generating the salt used in the **Unix crypt** algorithm..

The character sets are pre-randomized Ascii & Ascii-based Unicode character sets, reserved for obfuscation and **salt-key** generation (see **Appendix F**).

Character sets are registered **mainly** for the purpose of **obfuscation**; because of the nature of the text that is usually transferred on the Internet; it is preferable to use extra characters from the full A**scii** set [32..255]; to make password generation more random.

Some crypto implementations, **only for the private se**t, may opt to use the chosen character set as input to new algorithms, in order to generate different results; thereby, making the character set an additional cypher variable, such as the cypher mode below.

2. Cypher mode

The cypher mode is specified with the second 4-bit of the **second** byte. Most algorithms support only some of the cypher modes, if not a single one.

The set of cypher modes for the publicly available algorithms is also very small, even though, we again reserve a 4-bit quantity (**0..15**) for its setting. They are, in order: ECB, CBC, CFB, OFB, CTR, STREAM, CBC_CTS, CBC_MAC, MD_HMAC. The reserved set of cypher modes ranges from **1..9**, which is **nine** possible cypher modes. The default is the **ECB** mode, when the setting equals one; and zero means no mode specified.

3. Salt Value

The salt value must be generated according to the signing/encryption algorithm selected in the **beginning** byte.

This **third** byte is the index of a symmetric/assymetric algorithm to use; when two-way encryption is required to generate a password to be signed. The generated salt key must be truncated or concatenated to match the algorithm's required key length. This setting is only useful for header-based text protocols; such as **HTTP** and **FTP**, when the crypted password contains useful information.

If the key space is used to match the salt value, as specified in the capabilities section; then the **4 bytes** of the **stream extraction** settings are used to indicate which section of the key space to use for the generation of a matching salt value. Once the appropriate iteration start index into the key space is calculated; then there are 16 possible **salt** values to be tried, according to the **4-bits** of this setting (this setting is **guessable**). This option is used **only** when the salt indices are not specified during synchronization.

B. Obfuscation

In **clear text mode**, the password which is generated by using the above mentioned settings, **must** be obfuscated, to prevent easy guessing. The signing algorithms are prone to such guessing; because the length of their signature is fixed; some signature algorithms produce only 3 or 4 bytes. The obfuscation settings are only applicable to signing and symmetric algorithms, used in clear text mode. The use of **strong public-key** signatures negate the need for obfuscation.

1. Nonce Text

The information, for this setting, is used to obfuscate the incoming password text before encryption or signing. It is usually referred to as **DIGEST** information. It is the **result** of password text encryption which needs to be matched. The digest information, which is **always** required with the **IP** stack implementation of the **SALT** protocol, consist of the following **seven** bytes.

The **first** byte consists of two 4-bit quantities. The first 4-bit is the character set to use for the text to be inserted; and the second 4-bit is the character set to use for the text to be prepended or appended. The length of the password, to be signed or encrypted, should be, at least, **30** characters.

The **second** byte consists of two 4-bit quantities. The first 4-bit is the number of characters, a minimum of **four**, to be **prepended** to the password text; and the second 4-bit is the number of characters, also a minimum of **four**, to be **appended**.

The **third** byte consists of two 4-bit quantities. The first 4-bit is the number of characters, a minimum of **four**, to be **inserted** in the password text; and the second 4-bit is its **point of insertion** into the extracted password text. Insertion is **done first**, before appending and prepending the other obfuscation text strings.

The **fourth** byte is the offset, at which the obfuscation characters to be inserted, must be extracted from the designated character set.

The **fifth** byte is the offset, at which the obfuscation characters to be prepended, must be extracted from the designated character set.

The **sixth** byte is the offset, at which the obfuscation characters to be appended, must be extracted from the designated character set.

The **seventh** byte designates the algorithm used to sign the password's text, instead of obfuscating it, before encryption. The value **zero** means that no such signing is to take place. This setting is **optional;** because, it is faster to simply pick off a set of characters from a given set to obfuscate the password text (the obfuscation is **guessable**), instead of invoking another algorithm.

For cases, where the text/password to be processed is not specified, then the following **stream** and **capabilities** settings are used. They are used to implement the **SALT** protocol at the **IP** level; or in **header-based** protocols, or to add a **salt** load[2] to SCRIPT fragments (see section **SCRIPT fragments**.)

2. Digest

This digest obfuscation setting is specified with **two** bytes. The **first** byte indicates the algorithm used to sign the digest; a value of zero means that no signing takes place. The digest **must** be signed with a **signing** algorithm generating at least **20** bytes; either, if a header was picked as nonce and a symmetric algorithm was picked, or if the length of the digest is less than **20** characters. Selecting a nonce, with an appropriate length, should negate the need for an additional signature.

The nonce password must first **must be obfuscated t**o the minimum required length of **32** bytes. The length of characters required to obfuscate the nonce password, can be deduced from its length. The **second** byte indicates the point of insertion of additional characters, into the nonce password.

The point of extraction of the additional characters is specified in the **fourth** byte of the password text obfuscation settings. If the point of extracton does not allow sufficient characters to be extracted; then the extraction point must be moved backwards the appropriate number of characters.

3. Stream

The stream extraction settings are specified with **4** bytes. The first byte indicates the point of insertion of the password into the character stream. The insertion point ranges from multiples of a default **MTU** size of **576** bytes, up to **32832** bytes. The **minimum** insertion point **is** at a minimum of **4608** bytes, i.e. seven times the default MTU size.

[2]See section SCRIPT fragments

The **second** byte indicates the point of extraction of the required **20** bytes/characters (using a multiple **16** bytes offset points), from the stream, giving the password to be obfuscated, encrypted, and matched. The password text may still be too predictable, and needs an additional section added to randomize further, such as a replay count or a nounce. The **drawback** to using a replay count is the need to keep it **non-easily** predictable and **trackable**; because only the first frame may contain it; the recipient of subsequent frames must deduce it from a **pre-agreed** method (T.B.D.)

The **third** byte specifies the sender's MTU size, according to the following bit values:

1. **576**
2. **1152,**
3. **1280,**
4. **1480,**
5. **1500,**
6. **1728,**
7. **2304,**
8. **3456**

The **fourth** byte is used to specify a specific header or header field to use as the password text instead of extracting it from the stream. Implementations of the **SALT** protocol can be made more performant, by using a specific header, in the underlying protocol, instead of waiting for enough data to be transmitted, for extraction of the password text. In the case of the **SALT** protocol, **URI** type entries defined in some protofols are seen as a possible header choices; and is indicated by index **zero**. In the HTTP protocol, the **Authorization**, **Cookie**, and **WWW-Authentication** headers are indicated by index 1, 2 , and 3 respecively.

For **IP-level** implementations; the first point of extraction more than often be at **offset zero,** on initiating a session (see section **IP Protocol**). The subsequent points of extraction must be in the previously specified range of **4608** to **32832**.

If stream extraction is not enabled; then the underlying protocol must provide a means of specifying the password text, such as the **Authorization** header in **HTTP,** or the **OPTIONS** header section at the IP level.

C. Capabilities

This setting is specified with **4** bytes, and it consists mostly of **one-bit** quantities indicating options and requirements.

Obfuscation of the password text is **mandatory** in clear text mode; unless a specififc header is chosen, which may include a nounce. Obfuscation is optional otherwise, indicated by the value of the **1st** bit. The **11th** bit indicates, whether or not, the stream is used for password text extraction, even though a password may have been transmitted.

Character padding setting is used when a required block size is needed for encryption. Space/blank is the default padding character.

Below are specified the used bit quantities:

1. Always enable obfuscation.
2. Do padding with **null** characters.
3. Use a dummy digest entry.
4. Use the private set of algorithms.
5. Use the private set of characters.
6. Use the key-space for salt selection.
7. Only use hash algorithms.
8. Allow Diffie-Hellman algorithm for salting.
9. Allow public-key algorithms for salting.
10. Allow public-key signatures.
11. Always extract the nonce from the stream.
12. Always sign the encrypted and hexified nonce.
13. Replace the header by the encrypted and hexified nonce (a **symmetric/assymetric** algorithm **must** be selected, and **must** be specified in the **3rd** byte.)
14. Use the encrypted header as the digest.
15. Fully encrypt the session using an **assymetric/symmetric** algorithm which **must** be specified in byte **three.**
16. Use a user specified key when a public-key (PK) algorithm is in use.
17. Append a replay count hexadecimal integer text value to the nonce; otherwise the replay count must be part of the header, which must have been specified in the fourth byte of the stream settings.
18. Use the received replay count, when calculating the response digest (the nonce that is sent must not include the replay count.)
19. Use the specified progression algorithms for generating successive frame counts.
20. Use protocol frame CRC as part of the nonce.
21. Use public-key protocol for initial handshake.
22. Use certitifcate for initial handshake.
23. Use base64 encoding instead of hexifying.

Bits 25 to 32 indicates 0 to 255 possible progression algorithms, see **Appendix C**.

D. Diffie-Hellman

For the signatures and cypher algorithms the extracted characters are encrypted, obfuscated or not. For the Diffie-Hellman algorithm, they are used to generate the secret variables **a, b, A**, **B**, **G**, and **prime**, needed in the following expressions :

$$A = G^{Secret\_a} \bmod prime,$$
$$\mathbf{B} = G^{Secret\_b} \bmod prime,$$
$$K = A^{Secret\_b} \bmod prime.$$

G is the result of adding the decimal values of the characters forming the password; and then finding the nearest lowest Sophie Germain's prime number.

Secret_a is the result of listing the decimal values of the characters forming the password.

Secret_b is the result of listing the decimal values of the characters forming the password, in reverse order.

Prime is the result of value **G**, with the password's length as exponent to obtain the nearest lowest Sophie Germain prime number. If there are no lower prime number; then select the nearest highest Sophie Germain prime number (the generated key must be, at least, 512 bits long.)

The above selection is done to ensure, that the Diffie-Hellman algorithm generates a secure value **B.** Large values must be chosen for **prime** and secret exponent values **a** and **b**. The value **B** is the password that is inserted in the **IP** header, or the **HTTP** Authorization header.

An implementation of the SALT protocol, must keep a list of pre-calculated Sophie Germain prime numbers. The list must have, at least, **50** numbers; and start from prime numbers that are, at a minimum, **100** characters in length. The alternative is to use the OpenSSL library, which implements the Diffie-Hellman algorithm. The list of Sophie Germain prime numbers can be pre-calculated using the OpenSSL BIGNUM library.

IV. IP Protocol

For **IP**[3] level connections, the obfuscation values of the SALT setting, are **always** examined and processed. The generated password is inserted in the **IP Authentication**[4] segment of the **IP** header, at the specified point (MTU-based offset) of insertion into the stream. Frame CRCs may be used as part of the nonce.

The first instance of an encrypted password **must** always be inserted , in an header, at the beginning of a given **IP** stream, initiating a communication session. If the buffer length does not exceed the extraction offset, by the number of characters required; then after a given delay (no more than **200 ms**), the number of characters required is created from the collation of the pieces of obfuscation text that are to be inserted, appended and prepended. The extracted characters, making the password, are then encrypted and verified.

To avoid replay attacks, there must be a replay count.. The implementation **must** combine the replay count with the characters which are extracted from the stream, (a nonce may be chosen as an alternative T.B.D.)

To avoid the **200ms** latency, implementations may choose to select the **salt** replay count header, as the password; which is randomly generated on session initiation. After the initial password is generated; the extraction offset is then used to generate the following ones. The passwords are inserted in the **IP header**, when the insertion offset has been reached.

On reception, if the number of characters present is not enough; then an implementation must delay responding with an error (**ESALT** on Unix**)**, until enough characters are received or if the connection is closed. After the initial password is matched; then the extraction offset, starting before the insertion point, is used to extract the characters required to generate the subsequent passwords.

When included in the **IP** stack, public-key encryption should not be used to crypt the password (because of performance considerations). Synchronization still needs to be performed **off-line** in an highly encrypted exchange.

The **Salt** protocol is added as a **socket option**, to be turned on or off, depending on whether or not, one desires a **salted** port/connection. Every signed or encrypted text which is transmitted, at higher than the IP level protocol, **must** be **hexified** or **base64 encoded**.

[3]RFC 1108 Extended Security

[4]RFC 1826 IP Authentication Header (RFC 4302)

V. HTTP Protocol

For **HTTP** level connections, unless all values are zero, the obfuscation info, is examined and processed, for every exchange. The **Authorization**, **Cookie**, and **WWW-Authentication**[5] HTTP headers are used, to exchange the **SALT** load (encrypted password, and optional password text). An implementer should normally send the password text, only on encrypted connections, to prevent an **MITM** from intercepting the password text, facilitating their decryption tasks.

1. Authorization

A user agent that wishes to authenticate itself with a server—usually, but not necessarily, after receiving a 401 response--MAY do so by including an **Authorization** request-header field with the request. The Authorization field value consists of credentials containing the authentication information of the user agent for the realm of the resource being requested.

2. WWW-Authentication

The WWW-Authenticate response-header field MUST be included in 401 (Unauthorized) response messages. The field value consists of at least one challenge that indicates the authentication scheme(s) and parameters applicable to the Request-URI.

3. Cookie

The two state management headers, **Set-Cookie** and **Cookie**, have common syntactic properties involving attribute-value pairs. It is a way for an origin server to send state information to the user agent, and for the user agent to return the state information to the origin server.

Only the **Cookie** header is used, since salt credentials and settings are exchanged through synchrornization requests. When using the **Cookie** header as the password text; **always** include the URI as a key-value pair in the cookie information.

4. Content-Type

The Content-Type entity-header field indicates the media type of the entity-body sent to the recipient or, in the case of the HEAD method, the media type that would have been sent had the request been a GET.

[5]Hypertext Transfer Protocol RFC2616

A. Exchange

Exchanges at the **HTTP** level is done; either, by initiating a session **handshake**, or by sending an initial request with a salt load to the server. A **handshake** is a simple **GET** request with only **slash** '/' specified as the path. The server must respond with an **WWW-Authenticate** response, with embedded password to be matched. An initial **GET** request must use the **Authorization** or **Cookie** header, if a handshake is not required.

The **handshake** is the preferred approach; because it reduces susceptibility to **MITM** attacks. The contacted server must respond first, with a proper **SALT** load, which must be verified, to confirm that the right **HTTP** server responded.

Care must be taken, when implementing the **SALT** protocol at the **HTTP** level, to avoid repeating a given password, within a given session. A **repeated** password may be an indication of an **MITM** replay attack.

The preferred approach is to embed a replay count in the **URI, Authorization,** or C**ookie** text.

Private servers may use the headers, **AccessText** and **AccesKey|AccessPassword** for **salt** credentials verification.

VI. SASL protocol

A plug-in can be provided, as the handshake mechanism in the **SASL** protocol, replacing **DIGEST**. Measures must be taken to ensure the validity of the plug-in; especially if your are using the plug-in in a fully-private network.

VII. Synchronization

Synchronization is performed **off-line** in an highly encrypted exchange., using fragments (see **section A**, Fragments). We use an HTTP-like **fragment cache semantics** (**patent pending**) to perform the synchronization of shared secrets.

A UDP-based **geometric** progression protocol is used; where a node, in a given network, sends a given fragment to **2-5** of its nearest peers. Each of these peers, then in turn sends the same fragment to 2-5 of its own peers. The protocol may rely on the **multi-casting** features of the **UDP** protocol.

Peers may verify their database of settings, signatures, public keys, and certificates with each other. Secure **HKP**-like requests can be made for public-keys. This type of verification ensures that no fake signature or key can easily be introduced into the network. The requests can be made either with our **UDP-based fragment©®** protocol, or the **HTTP** protocol itself.

A. Fragments

Fragments are sections of text that start and end with an **SGML**-like tag. **SGML** is the precursor language of **HTML**, **XML**, and other derivatives. Fragments are seen as just another **Content-Type** (**fgr/html, fgr/xml, fgr/cfg©®,** etc.); and obey the same cache semantics as other content types. A **GET** or **PUT** of a fragment must use the hierarchy path (referred as **XPATH** for SGML-based documents). The path is required, so that an application (e.g. browser) can update a given document, by placing the fragment, at the right level, in the document.

A synchronization fragment must contain, a SALT setting, one or more signatures, and one or more public keys. The signatures and public keys must specify their issuer (see **Appendix A**, for the synchronization XML schema). The issuer must be the **fqdn** of the machine where the **SALT** setting, signatures, or keys were generated. The issuer may additionally be identified by login or user name, and an optional shared secret.

Fragments are always encrypted with the peer"s public key. They can be prepended with a set of HTTP headers. When a peer receives a **UDP** fragment, it first examines whether it is just a basic fragment, or an HTTP fragment, specifying a **Content-Type**. For example, a server could send its entire database of salt-settings, signatures, and public-keys to the peer, in a compressed format, using a UDP connection.

Exchanging entire databases, is not recommended; unless, a peer's identity has been verified to a high degree of certainty, using the ***invitation to the dance protocol.*** The form of the database is left to the implementer's choice.

A given host may provide, a single salt-setting to every peer, or a distinct one to every single peer. Every single port of a given host may also have their own salt-setting.

B. SCRIPT fragments

SCRIPT fragments that are part of an **HTML** page, can carry their own **SALT** load. The load, in this case, consists, of the signature of the text found in between the **SCRIPT** tags; and the encrypted/signed signature to be matched. The script engine (e.g. Java script engine), **should not execute** the script code, if the **SALT** load is not verified.

The signature of the script is optional; if it is specified, then the salt-settings are used to match the specified password; if it is not included as part of the script tag, then the text of the script itself becomes the text to be salted, in order to generate the matching password. The script text, in this case, is seen as the stream of characters; and the **SALT** protocol **obfuscation** settings are used.

VIII.Conclusion

Our approach may not necessarily prevent all possible **MITM** type of attacks; but it does reduce the possibility that such attacks can take place on a ***Salt-protocol*** protected network. Denial of service attacks are still possible; if a hacker finds a way to strip **SALT** passwords from **IP** headers, preventing synchronization from taking place. Concerted attacks on **routers**, can also defeat the protocol; since only the **IP** address is relied upon, to verify a requester's identity.

User names and passwords management, by desktop users, can be made unnecessary. The approach is to use a **USB** key, usually referred to as dongle key, which contains your encrypted personal information, to boot your computer. It may additionally contain **SALT** settings information, to access **Web-based** accounts. Once booted, your desktop will manage password creation and update. Your only responsibility will be to keep your **USB** key safe.

Our **SALT** protocol, being a handshaking protocol, can be included at **any** level of the **IP** stack, or used at the **HTTP** protocol level, or both. For ports, other than the **Tsert.com** registered ones, the **SASL** protocol could be used.

The salt setting is not just meant for communication between peers; it is also meant as a means of accessing all kinds of resources, such as printers. Printers, on a given network, can be given their own **UUID** and salt-setting; and, unless the proper saltt negotiation is performed, printing tasks can be disabled.

As computer processing power increases, the need for clear text communication will disappear; and only encrypted communication will be in common use; therefore, the obfuscation parts of the **SALT** protocol may no longer be necessary.

IX. References

[1] The First Ten Years of Public-Key Cryptography, Whitfield Diffie, Proceedings of the IEEE, vol. 76, no. 5, May 1988, pp: 560-577.

[2] U. M. Maurer and S. Wolf, The Diffie-Hellman protocol, Designs, Codes, and Cryptography, 19, pp. 141-171, 2000.

[3] Atkinson, R., "The IP Authentication Header", RFC 1826, August 1995.

[4] Harkins, D. and D. Carrel, "The Internet Key Exchange (IKE)", RFC 2409, November 1998.

[5] Kent, S. and R. Atkinson, "Security Architecture for the Internet Protocol", RFC 2401, November 1998.

[6] Maximally Periodic Reciprocals, R.A.J. Matthews (1992). Bulletin of the Institute of Mathematics and its Applications; vol 28 pp 147-148.

[7] H. Riesel, Prime Numbers and Computer Methods for Factorization, 2nd ed., Birkhäuser 1994.

[8] R.T. Morris, "A Weakness in the 4.2BSD UNIX TCP/IP Software", CSTR 117, 1985, AT&T Bell Laboratories, Murray Hill, NJ.

[9] S. Bellovin, "Security Problems in the TCP/IP Protocol Suite", April 1989, Computer Communications Review, vol. 19, no. 2, pp. 32-48.

[10] Rivest, R., "The MD5 Message-Digest Algorithm", RFC 1321, April 1992.

[12] L. Joncheray, "A Simple Active Attack Against TCP, 1995, Proc. Fifth Usenix UNIX Security Symposium.

X. Appendix A

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE synchronization [
<!ENTITY copyright "Copyright 1996-2012, Tsert.com, All Rights Reserved.">

<!ELEMENT sync:synchronization ( sync:peer+ )>

<!ELEMENT sync:peer ( sync:issuer, sync:salt-settings?, sync:signatures?, sync:public-keys?, sync:certificates?  )>
<!ELEMENT sync:issuer ( sync:description? )>
<!ELEMENT sync:salt-settings ( salt-setting+ )>
<!ELEMENT sync:signatures ( sync:signature+ )>
<!ELEMENT sync:public-keys ( sync:public-key+ )>
<!ELEMENT sync:certificates ( sync:certificate+ )>
<!ELEMENT sync:verifiers ( sync:verifier+ )>
<!ELEMENT sync:devices ( sync:device+ )>

<!ELEMENT sync:salt-setting sync:description? sync:peers? (#PCDATA) #REQUIRED>
<!ELEMENT sync:signature sync:description? (#PCDATA)>
<!ELEMENT sync:public-key sync:description? (#PCDATA | #CDATA) #REQUIRED>
<!ELEMENT sync:certificate sync:description? sync:issuer? sync:ssl-metadata? sync:verifiers
                (#PCDATA | #CDATA) #REQUIRED>
<!ELEMENT sync:ssl-metadata> /* see Secure Socket Layer Spec for attributes */
<!ELEMENT sync:verifier ( sync:description? )>
<!ELEMENT sync:device ( sync:description? )>
<!ELEMENT sync:description (#PCDATA) #IMPLIED>

<!--
The issuer is always the owner of the salt-setting by default.
The issuer is represented by the Fully Qualified Domain Name (FQDN)
of the  computer used to generate the salt-setting, signatures, and public-keys.
The owner is the user to whom the salt-setting, public-key, signature, or certificate belongs.
It can be specified by an email address, a UUID, or some other form of identification
which is unique for the issuing peer.
-->
<!ATTLIST sync:issuer fqdn NMTOKEN #REQUIRED>
<!ATTLIST sync:issuer ip-addr NMTOKEN #REQUIRED>
<!ATTLIST sync:issuer mac-addr NMTOKEN #IMPLIED>
<!ATTLIST sync:issuer login-name NMTOKEN #IMPLIED>
<!ATTLIST sync:issuer shared-secret NMTOKEN #IMPLIED>

<!-- An alternative host machine which can use a non-host based salt-key -->
<!ATTLIST sync:device fqdn NMTOKEN #REQUIRED>
<!ATTLIST sync:device ip-addr NMTOKEN #IMPLIED>
<!ATTLIST sync:device mac-addr NMTOKEN #REQUIRED>
<!ATTLIST sync:device login-name NMTOKEN #IMPLIED>
<!ATTLIST sync:device shared-secret NMTOKEN #IMPLIED>

<!ATTLIST sync:verifier fqdn NMTOKEN #REQUIRED>
<!ATTLIST sync:verifier default ( yes | no ) #REQUIRED>
<!ATTLIST sync:verifier ip-addr NMTOKEN #REQUIRED>
<!ATTLIST sync:verifier mac-addr NMTOKEN #IMPLIED>
<!ATTLIST sync:verifier login-name NMTOKEN #IMPLIED>
<!ATTLIST sync:verifier shared-secret NMTOKEN #IMPLIED>
<!ATTLIST sync:salt-setting owner NMTOKEN #REQUIRED>
<!ATTLIST sync:salt-setting title NMTOKEN #REQUIRED>

<!-- You should really have a separate salt-setting for the IP protocol -->
<!ATTLIST sync:salt-setting protocol ( HTTP | IP | TCP | SASL | ANY )  “HTTP”>

<!-- The type of the setting i.e. device|network|club|member|account|etc. -->
<!ATTLIST sync:salt-setting type NMTOKEN #REQUIRED>

<!-- The update frequency of the setting i.e. handshake|hourly|daily|weekly|monthly|yearly|etc. -->
<!ATTLIST sync:salt-setting update NMTOKEN #REQUIRED>

<!-- The salt key is the main key used for encryption and password matching. →
<!-- Its value is hexadecimally encoded -->
<!ATTLIST sync:salt-setting salt NMTOKEN #REQUIRED>

<!-- The hmac key is the key used for signaturing and encryption. -->
<!ATTLIST sync:salt-setting hmac NMTOKEN #REQUIRED>

<!-- The hexadecimally encoded value of the SALT  indices. -->
<!ATTLIST sync:salt-setting indices NMTOKEN #IMPLIED>

<!ATTLIST sync:signature owner NMTOKEN #REQUIRED>
<!ATTLIST sync:signature title NMTOKEN #REQUIRED>
<!ATTLIST sync:signature date NMTOKEN #REQUIRED>
<!ATTLIST sync:signature expiry NMTOKEN #IMPLIED>
<!ATTLIST sync:signature photo NMTOKEN #IMPLIED>
<!ATTLIST sync:signature password NMTOKEN #IMPLIED>
<!ATTLIST sync:signature identifier NMTOKEN #REQUIRED>
<!ATTLIST sync:signature hmac NMTOKEN #IMPLIED>
<!ATTLIST sync:signature fingerprint CDATA #REQUIRED>

<!ATTLIST sync:public-key owner NMTOKEN #REQUIRED>
<!ATTLIST sync:public-key type (private|public) “public”>
<!ATTLIST sync:public-key title NMTOKEN #REQUIRED>
<!ATTLIST sync:public-key date NMTOKEN #REQUIRED>
<!ATTLIST sync:public-key expiry NMTOKEN #IMPLIED>
<!ATTLIST sync:public-key photo NMTOKEN #IMPLIED>
<!ATTLIST sync:public-key password NMTOKEN #IMPLIED>
<!ATTLIST sync:public-key identifier NMTOKEN #REQUIRED>
<!ATTLIST sync:public-key numeric-id NMTOKEN #REQUIRED>
<!ATTLIST sync:public-key fingerprint CDATA #REQUIRED>

<!ATTLIST sync:certificate owner NMTOKEN #REQUIRED>
<!ATTLIST sync:certificate title NMTOKEN #REQUIRED>
<!ATTLIST sync:certificate date NMTOKEN #REQUIRED>
<!ATTLIST sync:certificate expiry NMTOKEN #IMPLIED>
<!ATTLIST sync:certificate photo NMTOKEN #IMPLIED>
<!ATTLIST sync:certificate identifier NMTOKEN #REQUIRED>
<!ATTLIST sync:certificate numeric-id NMTOKEN #REQUIRED>
<!ATTLIST sync:certificate fingerprint CDATA #REQUIRED>

]>
```

XI. Appendix B

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE vpn-invitation [
<!ENTITY copyright "Copyright 1996-2012, Tsert.com, All Rights Reserved.">

<!ELEMENT invitation ( vpn:request | vpn:verification )>
<!ELEMENT vpn:request ( request:source, request:destination ( request:parameter | request:response )? )>
<!ELEMENT vpn:verification ( request:source, request:destination, verification:signature, verification:peers? )>
<!ELEMENT request:source>
<!ELEMENT request:destination>
<!ELEMENT request:parameter>
<!ELEMENT request:response>
<!ELEMENT verification:signature>
<!ELEMENT verification:peers ( request:source+ )>

<!-- The Fully Qualified Domain Name (fqdn) must be provided -->
<!ATTLIST request:source source:fqdn NMTOKEN #REQUIRED>
<!ATTLIST request:source source:ip-addr NMTOKEN #REQUIRED>
<!ATTLIST request:source source:mac-addr NMTOKEN #REQUIRED>

<!ATTLIST request:destination destination:fqdn NMTOKEN #REQUIRED>
<!ATTLIST request:destination destination:ip-addr NMTOKEN #REQUIRED>
<!ATTLIST request:destination destination:mac-addr NMTOKEN #REQUIRED>

<!ATTLIST verification:signature signature:type ( “SHA1” | “MD5” | “SHA256”  | “SHA512” ) “SHA1”>

<!ATTLIST request:parameter parameter:dh-cyclic NMTOKEN #REQUIRED>
<!ATTLIST request:parameter parameter:dh-prime NMTOKEN #REQUIRED>
<!ATTLIST request:parameter parameter:dh-value NMTOKEN #REQUIRED>

<!--
The returned value is either the result of the Diffie-Hellman equation, or the SHA1
signed collated result of the Diffie-Hellman secret K keys, that the requester
is supposed to have exchanged with peers requesting verification.
The secret K keys must be collated in the order of specification of the peers ( verification:peers ).
-->
<!ATTLIST request:response response:collated ( “true” | “false” ) “false”>
<!ATTLIST request:response response:value NMTOKEN #IMPLIED>

]>
```

An HTTP **POST** request can be used with www-url-encoded content with the same fields.

```
POST / HTTP/1.1
Content-Type: application/x-www-form-url-encoded
Authorization: <salted password ...>
Host: <server ...>
Content-Length: xxx
request:source:fqdn=xxxx&request:source:ipaddr=xxx.xxx.xxx.xxx
&request:parameter:dh-cyclic=XXXX& etc...
```

The response is an HTTP **OK** response with an www-url-encoded request:response.
The peers's IP addresses are separated by a **colon** character '**:**'.

XII. Appendix C

The obvious progression are specified first; and then, the possible progression algorithms.

[1] `The cardinal numbers.`

[2] `The even cardinal numbers.`

[3] `The odd cardinal numbers.`

[4] `The prime numbers.`

[...]

XIII.Appendix D

$$\begin{tikzcd}\text{Client} \arrow[r, "E1"] & \text{Server}\end{tikzcd}$$

The client initiates a connection to the server, requesting an initial exchange. The client is unknown to the server, except for its **IP address**. In the invitation to the dance protocol, only publicly available information, is used, in this case the IP address. **All exchanges are Diffie-Hellman exchanges.** The client's temporary public key is sent along with the request.

$$\begin{tikzcd}\text{Server} \arrow[r, "E2"] & \text{Peer 1-5}\end{tikzcd}$$

As in a real dance, the host, in this case the server, queries other parties that have already joined the dance, to verify the requester's identity.

$$\begin{tikzcd}\text{Peer 1-5} \arrow[r, leftrightarrow, "E3"] & \text{Client}\end{tikzcd}$$

Each party queried, makes a verification exchange with the requester (the system asking to join the dance.) The requester must keep track of all these exchanges; because it will then be asked, by the host, the server, to verify said exchanges. These exchanges must be made, on **non-salted** ports; ports that are not protected by the **SALT** protocol.

$$\begin{tikzcd}\text{Peer 1-5} \arrow[r, "E4"] & \text{Server}\end{tikzcd}$$

Each party passes on the requester's responses to the server.

$$\begin{tikzcd}\text{Server} \arrow[r, "E5"] & \text{Client}\end{tikzcd}$$

The server then collates the received responses, and transmits to the client a new request, resulting from the collation. The server sends its own temporary public key to the client.

$$\begin{tikzcd}\text{Client} \arrow[r, "E6"] & \text{Server}\end{tikzcd}$$

The client receives the server's responses, and collates the set of responses, it sent to the other parties, to respond to the server's Diffie-Hellman challenge.

$$\begin{tikzcd}\text{Server} \arrow[r, "E7"] & \text{Client}\end{tikzcd}$$

The server confirms the client's response, and then terminates the handshake, by sending a **salt setting**. The salt setting is then used, to establish further communication.

XIV.Appendix E

The first algorithm is Diffie-Hellman.

1. Diffie-Hellman

The **signature** algorithms start at index **two.**

2. SHA1 (now preferred over md5sum),
3. SHA224,
4. SHA256,
5. SHA384,
6. SHA512,
7. RMD160,
8. TIGER,
9. HAVAL,
10. WHIRLPOOL,
11. MD5SUM,
12. MD4SUM,
13. MD2SUM,
14. CRC32,
15. CRC32_RFC1510,

The **cypher** algorithms start at index 16 with two default Unix ones.

16. CBC_CRYPT,
17. ECB_CRYPT,
18. 3DES,
19. CAST5,
20. BLOWFISH,
21. AES128 - RIJNDAEL128,
22. AES192 - RIJNDAEL192,
23. AES256 - RIJNDAEL256,
24. TWOFISH,
25. TWOFISH128,
26. ARCFOUR,
27. DES,
28. SERPENT128,
29. SERPENT192,
30. SERPENT256,
31. SEED,
32. RFC2268_40,
33. RFC2268_128,
34. SAFER_SK128 (not implemented),
35. DES_SK (not implemented)
36. IDEA (currently patented, not in use)
37. CAMELLIA128 (currently patented, not in use)
38. CAMELLIA192 (currently patented, not in use)
39. CAMELLIA256 (currently patented, not in use)

The **asymmetric** algorithms start at index **48**.

48. RSA_512,
49. RSA_1024
50. RSA_2048,
51. RSA_4096,
52. RSA_8192,
53. DSA_512,
54. DSA_1024
55. DSA_2048,
56. DSA_4096,
57. DSA_8192

XV. Appendix F

Character Set 1:

```
,}i`eRjWa6~G/=*qziuGAFPsaExcSegZBYJHQ8mnM2W9ktDhb6.qTOl3jop5LRvX17/VNrIwyf0UdC4KgMJCq4O
Zr9nFfexouyLIikAahKDWpY1S0PlvT5RGE8c.zXsmU67H/V2twNQd3bBjw10UDXsZoWbm9Pglhv6jMATHSz2
YOIfLKkucBp73GqxRCaEyQNitV58/FJner4.dB6pTw0aIrVq74A8edscJCxURLHF3lSKzb2XY9/o.gjEkuNPZDyn
OvMmGf1WQht5i
```

Character Set 2:

```
lpV|vE"M)23yNSI_Q7!KUmf-r.O@nsD]b#YZB[+Lk:F9tw{>,}i`eRjWa6~G/=*q<dCh8P4(xcA^T;&o5z?
%H0g'XuJ1$|dZg5I.qj@iSrP4<_%eYQx0tcGLO7>k2&D~:p+,
[My1BKHa*REn"?}v'JF`flAzomw)X$89V3T{^uUh;!W=6]/C#s(bN-mOKi_.s~MJ6#H{!+pN2T"Z7G-B>?5zx
%RIk/[c0(3F*`D,h;auj]U|&bAEwX@rYn<Cvo$)
```

Character Set 3:

```
(gRH_CKd9Z+:EuArp$3U>#4`P2=vh8,n!b7c&eGD@S[iI|V0"lmBq{}6.<-wW;aXxt1zQT5o%kNLj'Ys^?
*OF)yMJ~/f]uI!i]y<32-9$^_n`7XGSgxmp[41QK*bCRc,5YJEqVds6=awN8;&"0{rv)%LW/+~hl.zFTA?:}'MBfDk|
UoteZPHjO#>(@Ne&Z[K3ybBvgk/G(|D_q?L].1T!='#XUja4C-F>rm7VSH"cAJ;6psw{}Iz9Q~o)dP2*`Y@
```

Character Set 4:

```
KpCcx&>=UdzZ[ya0:1%tIeA]@5l9Yv*)iOm_R,!-q|gJsGW3Xb('FLhn`N$<E4H6j7w;D{M/r.fkT?
~2"#PSV+QBu^o8}~9<6.W]${ps,RZfrQ!xVBH)Je='-o/bnE}*STcUNPG`(&Dz^Cv5j?;4+a>3OKu@ltI:M%w|
dA_iF"h#Xg720kY1Lq[y8m"u(I7QT-M5O*|v/e{XD:kFp6R'xlh%!
jz+;4Yf@&}g1w`>^]n$sWHcotiaZPmVBb_L~=?8
```

Character Set 5:

```
%^cdZ)-yPO"+rmwp9UtXu@g'eb$sAoaDKG.1Q6*(h04Vz,J7]BlS`RYF=_>fM&/#T8;3IC:<WLvH[5k!nNxE}?{2~|
qijrMU$a^H|lT@*<`LN.R!F-58BtE7p1eu%0O)q(Xnx,4~KmSy}"wG36bV#'gk=Jd?[9DP;v:/
{Q2ZAfW>hIco]CYi+&js_zg"/+IE}[P|~YD36hBp#yZK2srLu1w?4aA>ifbeF0dCm,U&vzqtoN<(H8S;MR^X.%!
$l`j9
```

Character Set 6:

```
]xH`k0L&-%gyl!jJ8V(Fv$ba>Uo)~[KYTt<B*mG4q3/Z6z,IP2rE9S.R1{X+_"i;w'^}p#?e@DMOAQshcf7n:C=N5|
duW4|g:Re+Tvlf5-/KU}3,]Dup[21o"ZYrnSV(.A9b=$JM6^zhQ0t;Ok&7~@!{s_mc>HqFi')
%8PWC<N#jaxyB`*IXGLd?Ew~@|pejynq(EHBTzG;Y=8l$A3b_ZDwatv%ofdW^SMk'!
2U{V}N#5C*R[h97/6)-:.&,uciP
```

Character Set 7:

```
7_|>Xd.4"qvAj9[GWS'6ymP$F2N10!,Lc*i?ouQngH/U`lz;C~M<p#B8:kKJrVsR^YeEI-3+aDh5(fZ]t}w@
%&b={)TxOJ05!w]L-r^g9B"I~RxZQd(SmhibnAD@qUWpH38Y)z>=7Ck6vcuEF}y.`_tM&:s{lK[;a%
$j#<14e'PVXG+,N*O2o?f|T/lX3V#}`(rbM,!>1Z"v-0*Y8On{mRoL9^+H:2Iu;@[fG]$U6%PC5_xE&iDs?
4JTpQy~aAth
```

Character Set 8:

```
3']2Zw%CMPnGR:vghr7D@_Ase^/$)qm;yKLS?Txl!b5Q},8c"O+E.N6u-I{4=jW19|
fYU`HV>d#<pB~ko&*zXFi[aJt(0?4a;x7iR]8o:9($T)qA/}5dE"F-j21>VKwmD.Yrln_&'g,[s{v|u0ZB3!
Lt=S+by#PC@%*~f`6kOHUJGWM<cpQez^XINhg-b5As7Ep%'uyN$Id/lx1@kL}^,Se?
6PMv`mhz3W~9Ot_Q=G[*BnH.:JqD|Xc+&4(j"oa;
```