An Alternative Solution to HTTPS for Secure Access to Web Services

Cristina Livia IANCU
IT&C Security Master
Bucharest University of Economic Studies
ROMANIA
livia.iancu@gmail.com

Abstract: This paper presents a solution for accessing web services in a light-secure way. Because the payload of the messages is not so sensitive, it is taken care only about protecting the user name and the password used for authentication and authorization into the web services system. The advantage of this solution compared to the common used SSL is avoiding the overhead related to the handshake and encryption, providing a faster response to the clients. The solution is intended for Windows machines and is developed using the latest stable Microsoft technologies.

Key-Words: Security Solution Avoiding HTTPS, Intermediate Security Level, Safe Communication, Web Services, Web Authentication System

1. Introduction

The idea of this communication solution appeared in the context of a large, integrated Transport Administration (TA) Software solution, used for managing shipments, sending packages, printing labels, and other logistic business. It is designated for companies of any size, no matter the carriers they deal with, or the ERP or e-Commerce solutions used. This TA software is presently used by over 7000 companies and has great growing opportunities. Because of such a big numbers of clients, performance – in terms of response time to clients, is a concerning issue, and also security come into discussion.

This main desktop application component is using Windows services, web applications, mobile applications and a set of around 20 web services. These web services provide information used for packaging and delivery of packages. Some of them offer general information, but some of them are specific to a certain client, for example, depending on the agreement between the client and a carrier.

These web services are publicly available because they have to be accessed from the machines where the main component is installed, so they need protection against unauthorized access by, let’s say, clients that no longer fulfills their payment obligation.

The application that first issued the security problem is also one related to the main TA component. It is designed as a plug-in and will be integrated with virtual web shops, helping the end-user/buyer to choose the delivery solution that fits him best. The user interface part is not interesting for this discussion, since all flow is based on web methods called directly from our clients’ web shops.

It analyses buyer’s address, packages dimensions and weight, and other characteristics of the shipment and suggests available freight services together with their prices and delivery times. The buyer selects one of them and confirms the payment with the virtual web shop. After that, the web shop is sending the confirmed order to the main TA component to be taken into account and processed.

The security issue that occurred here is that one web shop must be able to access only its own orders and its own configuration. It must not see other web shops deals with the carriers it uses; it must not be able to place orders in the name of another web shop and so on.

Protecting access information against unauthorized using is also an issue that must be solved.
Another problem related to this context, but also applicable in general environments, is that most companies restrict from security reasons, their computer infrastructure to openly access the internet by closing ports. So, clients running in such networks must also be capable of using our web services. All these described above, along with the need of quickly response to multiple concurrent users leads me to a solution that I named “The Light-Safe Communication Solution”.

2. Authorizing web services’ clients avoiding https

The main goal of the presented solution is to enable a light-safe communication between web clients and web services. Taking this into consideration, in this chapter I will define the solution requirements. Requirements also depend on existing hardware and software environment, so I will briefly state them:

- Server machines (where the web services are hosted):
  - Microsoft Windows operating system machines running Internet Information Services (IIS) for hosting the web servers;
  - Processors performance is medium to high (Intel Xenon CPU X5680 @ 3.33GHz, 4 processors, RAM: 4GB, 64-bit OS);
  - Internet connection bandwidth is 1 Gbps.
  - Relatively large number of different web services running;

- Clients machines:
  - Unknown capabilities, operating system and software, internet bandwidth, etc. We can expect a high range of situations.

- Connection over internet
- Message payload does not contain sensitive data. The company decided to provide information like: price calculation for freight products, suggesting best carriers for deliveries, mapping addresses to post codes, and so on.

Defining solution requirements is a very important step, saving resources and lowering the costs of a project. Correcting defects in early stages of development has proved to be far more efficient than at a later time.

As you are trying to search for a complete and correct definition for software requirement, you’ll find such diverse description of this term, but you still have to decide which to follow. A simple definition of requirements is that they represent what the solution must do and what constraints must be followed. Also for keeping it simple, I will divide them in two categories: functional and non-functional requirements for the solution I want to present you.

Functional requirements specify the software functionality that the developers must build into the product to enable users to accomplish their tasks [1]. They contain business requirements, user requirements and system requirements.

Main functional requirements of the solution are:

1. Using HTTP and SOAP. Because the web services we'll communicate to are already developed, the solution will have to comply with the present environment. The web servers are .net applications, using SOAP (Simple Object Access Protocol) over HTTP.

2. Passing through proxies. Web clients are not inside-company application, but they are open world-wide spread. In real life a lot of organizations have restricted environments with closed ports, proxies and other security policies implemented. This is why an important feature of the solution is to serve clients running in controlled/restricted environments.

3. Authenticating the clients. The solution must be capable to offer the web servers a way to identify and authenticate the clients in order to avoid clients to access or modify data without having the rights.

4. Authorizing the clients. Further than only authenticating, the solution should offer a way to allow clients to access the resources they are entitled to use and also to restrict their access to resources which they should not have.
More detailed, we must handle rights to call one web method, or another, or a specific list of methods. A client can have more roles; this means that it can access more web services.

5. Managing clients’ accounts. The system should have a method of easily manage accounts data. Sub-requirements are:
   - managing clients’ accounts and clients account information
   - resetting passwords
   - managing roles and assigning them to clients

6. Protecting client credentials. As clients and servers messages must travel on internet which, as we all know, is a threat-full environment, the solution must offer the functionality for encrypting username and password when they are sent over the wire to protect themselves. A related functionality is: Encrypting credentials is optional. As said above, we deal with all kinds of clients, so they should not be obliged to complicate their lives with password management, installing additional components and so on. If they want their identity and thus, their data on servers to be protected, they’ll go for encryption, otherwise they should be free to still use our services.

7. Storing/managing keys.

8. Transparency at client side. For easiness in operating, and also in developing clients - because a large number of clients will be software applications, not human users - the solution should not interfere when calling the web methods; no user interactions, no additional actions should be required.

Non-functional requirements include performance goals, descriptions of quality attributes [...] and imposed constrains [1]. I have considered the following to be the most important for my solution:

1. Software development process time for the solution should be as shortest as possible (Wouldn’t I heard this before? :)). Because human resource is the most expensive in this kind of business, this is a very important requirement of management team.

2. Easy integration. The solution should be very easy to attach and use by the web services, and the developing time spent with modifying the existing web services to support authorization should be minimum.

3. Reasonable response time. Web services responses time to the clients should not be affected, no more than 1-2 seconds, comparable to http communication and avoiding the overload of using secure http.

4. Configurability. Clients and services should have the option to use or not to use certain features, as well they should have the possibility to change attributes values after deployment, so developers won’t need to rebuild and redistribute the components.

5. Reliability and Availability. The components have to perform and maintain their functions in routine circumstances, as well as in unexpected circumstances.

6. Maintainability. This includes concepts of modularity, understandability, changeability, testability, reusability, and transferability from one development team to another. This will always be your software department requirement.

7. Capability and Scalability. The solution must perform its tasks for a large number of clients and web services. Also adding new entities into the system must not decrease the performance.

8. Error handling and logging. Wrong behaviors, error and warnings should be correctly treated and logged, so an administrator can easily pinpoint the problem and solve them within the shortest time.
3. Light-safe Communication Solution Architecture

This chapter describes solution components and other involved actors like users, applications, services, modules, etc. The main communication solution components are:

1. **Client management component** is responsible with creating, storing and updating clients’ accounts and their authorization information and permissions. Client management is a supporting component for the rest of the system. It interferes with the authentication component placed in the web service/s, every time a client calls a web method. It also communicates with the clients via a Product Manager when user name and password is generated and sent to them to be used. It is recommended that this component is operated by an administrator that manages the accounts and distributes the credentials to product managers. There are 2 sub-components needed:
   - Clients account database with information for:
     - accounts,
     - roles,
     - web-services, etc.
   - Client management web service containing methods to:
     - creating clients’ accounts
     - disabling clients’ accounts
     - setting validity start and/or end date and time for an account
     - resetting passwords
     - creating roles
     - deleting/disabling roles
     - assigning roles to clients

Because this web service is generating user name and password and transmitting them to the product manager’s machine through internet, to protect them from being intercepted and used in malicious actions, it will run over secure https.

2. **Authentication and authorization component**

The web services that provide information for the TA system need an add-on that authorizes the client. As I already explained, this information is not sensitive, but it is very important for the response to be quick. Users are filling in shipment forms, delivery requests, etc., and these calls should be able to auto-complete some of the data at the typing speed of the human operator, otherwise, they’ll have an unproductive and unpleasant experience.

So, the decision was to use the plain version of http protocol, not the secure http, which involves some overhead related mainly to the 3-way handshake, but also related to encryption of the payload. Additionally, certain web services avoid using SOAP and the verbose XML format, replacing them with RESTfull services and Json format, or even a semi-binary encoding, like Google Protocol Buffers which yields a very compact payload in a very efficient way.
Nevertheless, clients have to be authenticated, and for this purpose, an authentication and authorization component has to be developed. This component has to follow these steps:
1. receiving the encrypted user name and password inside the SOAP message header;
2. decrypt them;
3. hash the password;
4. compare password hash to the database value corresponding to the user name received;
5. read client rights and determine if it is allowed to execute the method it called;
6. supplementary checks in some cases;

This component has to satisfy an important requirement, the one saying it has to be a small piece of software, very easy to be use/integrated by the developers in existing or in new web services.

3 Good tempering component

As mentioned before, most of the system's clients are software like desktop applications, web applications, web services, plug-ins, scripts or any other piece of software that is capable of consuming web services. For example, for an ERP like Microsoft Navision an add-on can be developed for this purpose.

The clients are shown in the left side of the picture. Also there, we can notice that clients are inside protected environments/intranets. Every network is using gateways, firewalls and proxies. A gateway is a device or a computer that passes traffic from the local subnet to devices on other subnets. A proxy could be a computer or even software which serves as a hub through which internet requests are processed. Among other functionalities, proxies and servers are used to filter web content. Many companies have internal policies which allow traffic to pass only to certain ports, like 80 (HTTP), 21 (FTP), 25 (SMTP).

The light-secure communication solution provides a transparent way for clients to be able to call the web services in these conditions, but providing a minimum level of protection. At client machine, a resident component will run, which will intercept http calls, identifies the ones that call our specific web service and replace the credentials with encrypted ones in the SOAP message; then retransmit the call to its original destination.

![Figure 2. Flowchart of the preliminary registration step](image)

This is the preliminary step flowchart of registering a client if it wants to participate in this system.

Client registrations involve a request of participation. Almost everyone can register; this is a matter of negotiation with our sales representatives. No further verifications are needed.

Operator of the authorization system calls the Client Management Web
Service which generates the user name and password for the client and creates a database record with its information. Then the credentials are sent to the client by e-mail, SMS, phone or any other media agreed. The client is using the credentials for calling our web services depending on the technologies he’s using. Using of the system only requires the client to generate the request in the form of a SOAP message having a header with the following fields:

\[
\text{<UserName>\text{</UserName>}}
\]

\[
\text{<Password>\text{</Password>}}
\]

1. Client is sending the request using SOAP formatted message with credentials in the SOAP header.
2. The Good Tempering Service modifies the credentials.
3. If on the server side everything is ok, the client will receive the wanted response.

![Figure 3. Main communication flowchart in with light-safe solution](image)

After the registration preliminary step, any client can benefit of the light-safe communication solution in a transparent way. This main workflow is a classical one with following steps:

![Figure 4. Good Tempering Service Flowchart](image)

The Good Tempering Service is the component that allows the light-safe communication. It provides a way to
send requests from a restricted environment, being able to communicate over a classic http and thus, more efficient one, but in the same time, providing authentication and gaining permissions on the remote site.

It is a resident component that does the following:

1. It sniffs the communication on the local computer and identifies those calls targeting our web services. The identification is made by the presence of some special characteristics in the SOAP message.

2. If the request is targeting our web services, Good Tempering (GT) Service will read the credentials. If not, the request is transmitted to its destinations as nothing has happened.

3. Then, GT will try to retrieve the encryption key from where it is stored, and if everything is ok, it goes to next step. If not, an error will be logged in a file and the request will be transmitted as it is, despite the fact that it could not be correct – we cannot afford to affect the communication further more.

4. GT encrypts the credentials.

5. GT reconstruct the message using the encrypted credentials instead of plain-text ones.

6. Add a flag so authentication component will recognize the credentials as encrypted

The request with modified, or unmodified credentials, will be send to the original server over http.

4. Implementation of Light-safe Communication Solution

4.1. Environment

This solution is designed for Windows machines, so the majority of the technologies and tools used are from Microsoft family.

The mostly used operating system in the development and testing phases was the latest version of Windows 7, but also Windows XP sometimes. Web services involved are hosted by Internet Information Services (IIS) 7.5, which is a web server provided as an integral part of Windows 7.

For database support, SQL Server 2008 R2 Standard Edition was used.

The modules of the solutions were written in C# language, using Visual Studio 2010 and .NET Framework 4.

Other tool and components such as Wireshark, SoapUI, WinpkFilter has been involved, but they will be described later in this paper.

4.2. Implementation of Client Management Component

This component is implemented as a web service, named AuthenticationWS, which uses a database for storing clients’ information.

4.2.1 Database

The database name is Authenticator and contains:

- **Applications** are resources that clients want to access. In this scenario, most of them are web services.

- **Roles** are permissions that clients have inside an application, and they denote the method/s that a client has the right to execute. So, a role is associated with a single application. Because, at least right now, the number of applications and of roles is not that big (there are around 20 web services used, each one having 1, 2 or 3 roles), Application and Role tables will be fill in manually by the AuthenticationWS administrator.

- **ClientXRole** is a link table that allows clients to have multiple roles, and also allow roles/applications to have multiple clients (Many-to-Many relationship).

- **ClientAccount** stores information needed by a client to access the applications/web services.
The Authenticator Database diagram is presented below:

4.2.1 Client Management Web Service

AuthenticationWS has a simple 3-layer architecture:
- Database entities layer – implemented using LinqToSql entities and data context
- Business layer – contains the implementation of interface methods. All business related to authentication are contained in this module, so this layer, together with database entities layer are also used by the Authentication and Authorization Component (see next sub-section) that is added to the applications or web services to be protected.
- Interface layer – contains the methods signatures exposed to clients:
- bool ValidateCredentials (AuthHeader Credentials)
- AuthInfo CreateClientAccount (int consignorId, string Name, string Description, int roleId)
- AuthInfoResetPassword (string UserName)
- void AttachRole (string UserName, intRoleId)
- void SetActiveClientRole (string UserName, boolIsActive, intRoleId)
- void SetClientAccountValidityPeriod (string userName, DateTime? start, DateTime? end, DateTimeSetOnly setOnly)
- void SetActiveClientAccount (string userName, bool isActive)

AuthHeader entity contains:
- string Username;
- string Password;
- bool? IsEncrypted: this is used to determine if Username and Password are encrypted, because the solution must permit also plain text, for flexibility reasons.
AuthInfo contains the authentication elements that a client needs:
- string Username;
- string Password;
- string EncryptionKey.

DateTimeSetOnly is used for specifying if ValidFrom, ValidTo or both will be changed. This is needed because maybe it is not necessary both dates to be changed, and sending null means updating a specified field with null. So this enumeration it will be interpreted as “change only this value”.

DateTimeSetOnly enumeration values: Start, End, Both.

ValidateCredentials method decrypts credentials, searches for client, verifies its password, checks for isActive and also verifies if ValidFrom and ValidTo interval is correct. It also deals with Application/Role membership of the client.

CreateClientAccount method generates Username and Password and sends them back to the caller. Username is generated from the real Name of the client plus some random number. roleId will be generated for each application and send to the product administrator that will add clients for that particular application.

ResetPassword is used when a client password had been lost or compromised. This method is also called by the product manager. For that client, a new password is generated, hashed and stored in the database.

AttachRole allows an existing client to have access to the resources associated to an existing role.

SetActiveClientRole activate or deactivate a role for a client, supposing the role was previously attached to that client.

SetClientAccountValidityPeriod and SetActiveClientAccount are updating the ClientAccount. I will not insist since they are self-explanatory.

Authenticator WS must run on secure http, otherwise user and password generated by CreateClientAccount method will travel on wire in clear text.

4.3. Implementation of Authentication and Authorization Component

This component architecture is also 3-layer:
- Database entities layer (compiled as Authenticator.Database.dll)
- Business layer (compiled as Authenticator.Core.dll)
- SoapAuthenticator layer (compiled as SoapAuthenticator.dll)

As the first 2 layers are common with the previous component, I will only present here the ValidateCredentials method, which is mostly used by this component. Using the data layer and LINQ to SQL syntax, the client account is search for in the database, paying attention to validity criteria. If it exists, the role is checked, and then password hash is verified. If the IsEncrypted tag is true, then the credentials are supposed to be encrypted, so additional steps of decrypting them are necessary:

using (AuthenticatorDataContextDataContext = AuthenticatorDataContext.GetContext())
{
    if (Credentials.IsEncrypted.GetValueOrDefault())
    {
        Credentials.Username = Crypto.DecryptAES(Credentials.Username,
Convert.FromBase64String("aygWZjU6BwZGpAk/aNxNpJ5sOPY2pLlvzopsR65H9Ew="));
        ClientAccountca = DataContext.ClientAccounts.FirstOrDefault(p
            =>p/cli_UserName ==
        Credentials.Username&&p/cli_IsActiv&& (!p/cli_ValidFrom.HasValue ||
        p/cli_ValidFrom<= DateTime.Now)&&(!p/cli_ValidTo.HasValue ||
        p/cli_ValidTo>= DateTime.Now));
        if (ca == null)
            return false;
    }
    if (ca.ClientXRoles.Count(p =>p.Role.rol_Id==roleId&&p.cxr_IsActiv==true &&
if (!p.cxr_StartDate.HasValue || p.cxr_StartDate <= DateTime.Now) & p.Role.Application.app_IsActiv == true) !=1) return false;
if (Credentials.IsEncrypted.GetValueOrDefault())
Credentials.Password = Crypto.DecryptAES(Credentials.Password, ca.cli_EncryptionKey.ToArray());
return Crypto.VerifyHash(Credentials.Password, ca.cli_CodeHash.ToArray());
}

Crypto is a class in Authenticator.Core which contains mostly the cryptographic part. Crypto methods are:

- string GeneratePassword(string ClientName)
- string GenerateUserName(string ClientName)
- string GenerateEncryptionKey()
- byte[] ComputeHash(string plaintext, byte[] saltBytes)
- bool VerifyHash(string plaintext, byte[] hashValue)
- string DecryptAES (string encrypted, byte[] Key)
- string DecryptStringFromBytes_Aes (byte[] cipherText, byte[] Key, byte[] IV)

DecryptAES is used by ValidateCredentials to decrypt the user name and the password. For the user name, the encryption key is unique for all the users, but for password, the encryption key is retrieved from the database for that specific user. This method is, at its turn, use DecryptStringFromBytes_Aes. For encryption and decryption AES algorithm has been used, as it is implemented in .NET Framework AesCryptoServiceProvider, with the following parameters:

- Block size: 128 bits
- Key size: 128 bits
- Initial vector (IV) size: 128

### 4.3.1 Soap Authenticator

SoapAuthenticator layer contains AuthenticationExtension and AuthenticationExtensionAttribute which are used within the web services to implement validation of clients.

AuthenticationExtension is derived from SoapExtension .NET framework class, where, by overriding ProcessMessage method, username and password are read and ValidateCredentials from Authenticator.Core is called. If something is wrong a SoapException is thrown, so the caller of a web service will no longer be able to access what he intended to.

Username and password, together with “IsEncrypted” flag are defined in SoapAuthenticationHeader class which extends SoapHeader .NET framework class. In this manner, the soap header of a request will contain our custom fields.

AuthenticationExtensionAttribute extends a SoapExtensionAttribute, so a web method decorated with this attribute will implement authentication and authorization in a simple and elegant way.

### 4.3.2 Using Authenticator

From a programmer point of view, implementing authentication component within its own projects is a very easy task.

For console/desktop application steps are:

1. Add references to:
   Authenticator.Core.dll (Authenticator.Database will also be copied in the bin directory because it is used by Authenticator.Core)
2. call ValidateCredentials method

For web services steps are:

1. Add SoapAuthenticator.dll (Authenticator.Core.dll and Authenticator.Database will be copied)

Referring the database (get connection string):
The Authenticator.Database.dll has its own config file, so implementing Authenticator will not imply modification in the existing config file of a project. This file looks exactly like any app/web.config, so programmer should not know a specific format for this file. It can be used both for web and desktop or console applications. Now the information stored there is a connection key, named “Authenticator.Database.Properties.Settings.AuthenticatorConnectionString”. Authenticator also supports reading this key from host web/app.config file (classical way) if it is desired so. There is a limitation that “Authenticator.Database.dll.config” must be placed in the same folder as Authenticator.Database.dll, and for web services, this folder will be “/bin” (default binary folder for web services).

4.4. Implementation of Good tempering component

This component is a windows service that runs on clients’ machines and encrypts the username and password from the soap header in a transparent way for the user.

In the developing and testing process, a tool named SoapUI has been used. SoapUI is a free and open source cross-platform Functional Testing solution [2]. For analyzing the network traffic and understanding what is going wrong or right in the developing stage, a great help came from Wireshark tool. Wireshark is the world’s foremost network protocol analyzer and the de facto (and often de jure) standard across many industries [3].

Because .NET framework does not have a native library for intercepting, manipulating and retransmitting network communication, a third party WinpkFilter component was used. WinpkFilter is a high performance packet filtering framework for Windows for filtering raw network packets with minimal impact on network activity [4]. Because WinpkFilter API uses pointers, most of the c# code is unsafe.

Writing this component, that manipulates the TCP transmissions, requires a detailed knowledge of the TPC protocol.

The Transmission Control Protocol is one of the core protocols used in the 4-layer Internet Protocol Suite model. It acts as an intermediary between the application layer and the internet/network layer and provides full-duplex uni-cast communication. Its main features are reliability and flow and error control. It establishes a virtual connection between 2 participants in a conversation, in 3 phases:

1. Connection establishment
2. Data transfer
3. Connection release

TCP operates by the principle of the finite state machine.

The events that determine the state transitions are user calls, sent as messages and have corresponding TCP header flags to indicate their functions:

- **SYN**: A synchronize message, used to initiate and establish a connection. It is so named since one of its functions is to synchronize sequence numbers between devices.
- **ACK**: An acknowledgment, indicating receipt of a message such as a SYN or a FIN. [7]
- **FIN**: A finish message, which is a TCP segment with the FIN bit set, indicating that a device wants to terminate the connection.
- **RST**: reset the connection message.

1. Communication establishment

TCP protocol relies for communication on Internet sockets. TCP links source and destination through sockets, using a 3-way handshake for synchronizing sequence and acknowledgement numbers. Sequence number is a (random) number identifying the first byte of a payload data. Acknowledgement number is the number of the next byte expected by the receiver.

2. Data transfer

After connection establishment, TCP starts transmitting the actual data. Each
TCP participant has a transmit buffer and a receive buffer, which are managed independently. Using these buffers, the TCP protocol implements a sliding window mechanism which allows a participant to receive the data segments in an arbitrary order and reassembling them in the right manner.

The TCP client places the segment data it transmits in the transmit buffer and waits for server to acknowledge the segment. If the acknowledgment is not received in a specified amount of time, the data segment is retransmitted from the buffer. This amount of time is increased with each retransmission and the number of retransmissions is limited. When the limit is reached, the connection is reset. So the data segment is flushed from the buffer when acknowledgment is received or in error cases.

The client also keeps track of the following sequence number after an initial transmission. So, the packages are sent with the sequence numbers computed a priori, and if the server responds with an acknowledgment number that is bigger or lesser that client expects, the retransmission will be automatically triggered.

If someone interferes and changes the packets length, he/she has to manipulate the entire remaining flow of packets until connection reset, otherwise TCP will be aware that something is wrong, and the transmission will be corrupted. This is a challenge in writing the Good tempering component.

3. Connection release
When data transfer is over, a connection termination is initiated, normally by the participant that requested it. There are several scenarios for connection release. The mostly used is 4-way connection release, but there are also 3-way and 2-way variants.

Tempering HTTP messages
The main thread is doing the following:
- opens the virtual driver;
- bounds the network interfaces information and selects the “Local Area Connection” which will be used subsequently;
- sets the adapter mode for tunneling;
- creates and sets notification event for the packets;
- creates a set of filters and bounds them to the event; This filters selects the packets that will be processed by the application. In this scenario, the packets needed are the ones that go to/come from a specific IP address (the one where our web services are running). For convenience, only one IP is considered here. The packets are also filtered by TCP protocol and source and destination port. The fact that all other packets are released untouched to the interface has to be explicitly specified.
- Allocates and initializes packet structures
- Waiting for a packet.

When a packet with payload is intercepted, the string “ServiceAuthenticationHeader” is looked for, because this is the way a request that must be processed is identified. Then the username and password are retrieved, encrypted and replaced in the payload. Also the flag “IsEncrypted” must be added.

Http “Content-Length” filed has to be replaced with the new one. Because the packet content and length had been changed, the IP header checksum and the TCP header checksum must be recomputed.

A complex situation is encountered here: because of the different packet length, the TCP flow is corrupted and a manipulation of the server response must be also done.

The next sequence number of a packet going from client to server will have the value of the first sequence number plus TCP payload length that the TCP server originally sent. But the acknowledgement number from the server is different because the length has been changed. So, the application must intercept the server response and
put back the acknowledgement number expected by the client. After this manipulation, the TCP checksum must have been recomputed.

For an in-depth understanding of how and why checksums must be recomputed, see the following tables:

**Table 1. IP checksum calculation**

<table>
<thead>
<tr>
<th>IP Header format (:field length in bits)</th>
<th>IP Header checksum</th>
<th>Why need re-compute checksum</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Version: 4</td>
<td>Algorithm: checksum is computed over the entire IP header</td>
<td>For packet going from client to server: Overall length had been changed</td>
</tr>
<tr>
<td>- Header length: 4</td>
<td>1. set checksum bits to 0 2. divide the header in 16 bits words and add them all together 3. if sum exceeds 4 hexadecimal digits, add the 4 rightmost digits to the 4 leftmost digits. 4. complement the result (subtract the sum from 0xFFFF)</td>
<td>For packet going from server to client: no need to re-compute.</td>
</tr>
<tr>
<td>- Service type: 8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- Overall length: 16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- ID: 16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- Flags: 3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- Fragmentation offset: 13</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- Time to live: 8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- Protocol: 8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- Header checksum: 8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- Source IP address: 32</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- Destination IP address: 32</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- Options + padding</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Table 2. TCP checksum calculation**

<table>
<thead>
<tr>
<th>TCP Header format (:field length in bits)</th>
<th>TCP Header checksum</th>
<th>Why need re-compute checksum</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Source port number: 16</td>
<td>Algorithm: checksum is computed exactly as the IP checksum, but it covers TCP header + TCP payload + pseudo-header. Pseudo-header is made up from: - IP Source Address - IP Destination Address - Protocol (TCP=6) - Total length of entire TCP segment</td>
<td>For packet going from client to server: Overall length and TCP payload had been changed</td>
</tr>
<tr>
<td>- Destination port number: 16</td>
<td></td>
<td>For packet going from server to client: Acknowledgement number had been changed</td>
</tr>
<tr>
<td>- Sequence number: 32</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- Acknowledgement no: 32</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- Header length: 4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- Reserved: 6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- Flags: 6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- Window size: 16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- Checksum: 16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- Urgent pointer: 16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- Options + padding</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Further, next packets sent from client to server must be modified to correct their sequence number, because the server is expecting a different sequence number than the client sends. Then, the server response acknowledgement number will again be changed, and so on.

### 5.4. Testing the Light-Safe Communication Solution

For testing the solution, I have made a small application.

The application is developed also in C# and is calling 501 times the same web method. First time is using https and then is using Light-Safe Communication solution. As you can notice in the upper right corner of the capture, the time difference is significant: 76751 milliseconds for https vs. 44341 milliseconds for Light-Safe Communication solution.

Also, the picture shows that the watch is started and stopped in exactly the same
moments for both phases and the calls are identical.

In the next print screen, there is a Wireshark capture while the above application was running, showing the encrypted Soap Header sent instead of the plain text one.

Figure 6. Test calls and results for 500 calls

Plain text user name is “Livia4C51432” with password “FqxX5esOxiU=v44” and they are reaching the web servers as user name
“fqXrWjnBPiHWpqyIKxfxHw=“ with password “rXtuzQ81yGDQn4FSctUK5A==“. These are Base64 encryption of the real AES encrypted bytes.

From this picture we can also see that Light-Safe Communication solution is sending smaller packets on the wire that TLS V1. In the upper part, there is the last call of https stream. For computing the sizes we must add line 302 and line 305: 507+731 = 1238 bytes sent from client to server. For the response we have: 107+715 = 822 bytes (lines 304 and 306).

For a Light-Safe Communication solution call, we can look at lines 315 and 316: 1159 bytes for request and 676 bytes for response.

**Table 3: Light-safe communication solution compared to https measurements:**

<table>
<thead>
<tr>
<th></th>
<th>1st run</th>
<th>2nd run</th>
<th>3rd run</th>
<th>4th run</th>
<th>5th run</th>
<th>6th run</th>
<th>7th run</th>
<th>8th run</th>
<th>9th run</th>
<th>10th run</th>
<th>AVG</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>https</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>17251</td>
</tr>
<tr>
<td></td>
<td>1817</td>
<td>1782</td>
<td>1742</td>
<td>1686</td>
<td>1692</td>
<td>1711</td>
<td>1705</td>
<td>1742</td>
<td>1660</td>
<td>1709</td>
<td></td>
</tr>
<tr>
<td><strong>LSComm</strong></td>
<td>9303</td>
<td>9566</td>
<td>9227</td>
<td>9294</td>
<td>9959</td>
<td>9299</td>
<td>9640</td>
<td>9073</td>
<td>9035</td>
<td>9056</td>
<td>9345.2</td>
</tr>
<tr>
<td><strong>Diff:</strong></td>
<td>8867</td>
<td>8259</td>
<td>8201</td>
<td>7573</td>
<td>6963</td>
<td>7819</td>
<td>7415</td>
<td>8354</td>
<td>7573</td>
<td>8034</td>
<td>7905.8</td>
</tr>
<tr>
<td><strong>Avg diff:</strong></td>
<td>7905.8</td>
<td>8 sec</td>
<td>54% of https time</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>1st run</th>
<th>2nd run</th>
<th>3rd run</th>
<th>4th run</th>
<th>5th run</th>
<th>6th run</th>
<th>7th run</th>
<th>8th run</th>
<th>9th run</th>
<th>10th run</th>
<th>AVG</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>https</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>73899 .4</td>
</tr>
<tr>
<td></td>
<td>7513</td>
<td>7301</td>
<td>7323</td>
<td>7675</td>
<td>7178</td>
<td>7197</td>
<td>7177</td>
<td>7262</td>
<td>7643</td>
<td>7627</td>
<td></td>
</tr>
<tr>
<td><strong>LSComm</strong></td>
<td>4406</td>
<td>4486</td>
<td>4613</td>
<td>4434</td>
<td>4285</td>
<td>4426</td>
<td>4423</td>
<td>4412</td>
<td>4343</td>
<td>4331</td>
<td>44163 .6</td>
</tr>
<tr>
<td><strong>Diff:</strong></td>
<td>3106</td>
<td>2814</td>
<td>2709</td>
<td>3241</td>
<td>2893</td>
<td>2771</td>
<td>2753</td>
<td>2850</td>
<td>3299</td>
<td>3296</td>
<td>29735 .8</td>
</tr>
<tr>
<td><strong>Avg diff:</strong></td>
<td>2973.5</td>
<td>30 sec</td>
<td>59% of https time</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>1st run</th>
<th>2nd run</th>
<th>3rd run</th>
<th>4th run</th>
<th>5th run</th>
<th>6th run</th>
<th>7th run</th>
<th>8th run</th>
<th>9th run</th>
<th>10th run</th>
<th>AVG</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>https</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>14497 1.3</td>
</tr>
<tr>
<td></td>
<td>1409</td>
<td>1486</td>
<td>1414</td>
<td>1408</td>
<td>1415</td>
<td>1482</td>
<td>1461</td>
<td>1501</td>
<td>1450</td>
<td>1450</td>
<td></td>
</tr>
<tr>
<td><strong>LSComm</strong></td>
<td>9034</td>
<td>8733</td>
<td>9251</td>
<td>9013</td>
<td>9027</td>
<td>9033</td>
<td>9500</td>
<td>9136</td>
<td>8693</td>
<td>8788</td>
<td>90213 .5</td>
</tr>
<tr>
<td><strong>Diff:</strong></td>
<td>5064</td>
<td>6129</td>
<td>4891</td>
<td>5074</td>
<td>5129</td>
<td>5788</td>
<td>5110</td>
<td>5875</td>
<td>5981</td>
<td>5712</td>
<td>54757 .8</td>
</tr>
<tr>
<td><strong>Avg diff:</strong></td>
<td>5475.7</td>
<td>55 sec</td>
<td>62% of https time</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Also the number of packages sent on the network is smaller for Light-Safe Communication solution.

### 6. Conclusions

The entire Light-safe communication solution is used to authenticate and authorize access in a specific system, and the Good Tempering component can be used to protect credentials over the wire. In this way we can avoid using of SSL and https, which are costly in terms of response time. Just the username and password are encrypted and also the overhead of https handshake is skipped.

I have run the tests described in the previous chapter many times for 100 calls, 500 calls and 1000 calls and I synthesize the measurements in Table 3.
As we can see from the figures above, Light-safe communication solution is much better from call elapsed time point of view. The tests showed that is more than 2 times faster https calls and the performance increases with the number of calls (54% faster for 100 calls, 59% faster for 500 calls and 62% faster for 1000 calls).

It sends less data on the wire and also the number of round-trips between client and server is smaller for Light-Safe Communication solution.

It is a flexible, extendible and easy to be parameterized solution. With some filters modifications, it can be used in restricted environments.

It uses a strong encryption algorithm, AES with 128-bit key being strong enough to protect US Government classified information, as they state.

An aspect that can be improved in Light-Safe Communication solution is Key Management, in the aspects of distribution and storage. Because in this version of Good Tempering Service, only basic IP, TCP and HTTP messages were considered, another improvement will be taking care of more complex IP, TCP and HTTP messages, for example, treating TCP Header Options like selective acknowledgements, window scaling, congestion control.

Also, we can look forward at extending this solution for other purposes as, for example, encrypting instant messaging traffic.

7. References


