Brian Malley

CSULB

CECS 579 Information Security

Prof. Mehrdad Aliasgari

Dec 10th 2015

Library to Secure Software from Unauthorized Use

Abstract:

Preventing an unauthorized user from using software has long been a problem due to the ease at which software is copied, and the deterministic nature of computers. Naive implementations of authentication systems are vulnerable to simple software copying and replay attacks. Use of a Server/Software paradigm can protect against replay attacks and an untrustworthy OS. In particular, I describe and implement two libraries, a server library and a user library, which together allow for simple verification of a software package using OpenSSL and Crypto++. A Unique ID for each software copy, as well as timestamp and direction bit are encrypted and sent to the server for verification. The server responds with a unique for each software copy ID, timestamp and direction bit. Together, these elements protect from typical attacks.

Introduction:

Naive implementations of software protections are easy to circumvent due to the deterministic nature of computers and the power of an adversary's operating system, where the code is expected to run. A simple validation key to unlock a copy of the software allows an adversary to simply copy that version and use the same key for as many copies as they desire, with only one legitimate copy. In addition, it is not trivial for software to determine whether it has been installed before, or even whether variables like system time are legitimate, since the software can be assumed to be on the adversary's computer and operating system, which are untrustworthy. These ineffective prevention measures are costly. In 2001, software piracy losses were estimated at US$11 Billion[3]. In 2008, in France alone, software developer losses due to piracy were estimated at 2.9 Billion Euros[2].

Several schemes have been used and proposed to prevent the unauthorized use of software. Microsoft has several schemes in place for its Windows software, including hardware locking and server authentication. For OEM systems, Microsoft attaches the software itself to the BIOS firmware of the computer. This helps prevent using the software on unintended systems[4]. For commercial Windows software, Microsoft provides a unique serial key for each copy of the software and requires activation via a Microsoft server before installation may proceed. With this scheme, since each software copy has a unique serial number, Microsoft can block the usage of specific keys which have been compromised [4].

Techniques involving communication with a server for software are common and involve a variety of approaches[2]. For this project I have chosen to implement a simple client authentication protocol with a server, which is activated once every time the software is executed. In this sense, the software with this scheme is not an offline solution, since communication with the server is required for execution. To protect against common attacks such as replay attacks, man in the middle attacks, mirror attacks, and usage of the server as a decryption oracle, this implementation uses a combination of well-understood cryptographic solutions. Specifically, this proposed implementation uses an SSL connection between the client and server to transmit an encrypted and authenticated unique ID, timestamp, and direction bit from the client to the server. Once the server has validated the information, it responds with its own encrypted and authenticated unique ID, timestamp, and direction bit, which the client will decrypt and validate before starting execution.

The encryption, authentication, and SSL implementations, as well as accessory pseudorandom number generator and secure byte implementations, are well understood and commonly used functions which are provided by two free libraries, OpenSSL and Crypto++. Both of these support the use of C++ and contain a large variety of cryptographic algorithms. In this implementation, the details of these are hidden from the user, who only needs to use the two libraries (client, server) I have developed for this project.

Literature Search:

In my preliminary research I did not find much in the way of authentication schemes. Unfortunately, as I elaborate on further in this paper, this initial research was not up to par.

However, I have done some further research and compiled some important literature sources. I investigated Microsoft and their version of software authentication, which uses serial keys and a connection to a remote server for authentication. Microsoft has a unique key for each software copy and can block specific ones[5]. Microsoft does not completely disallow use of violating software, instead opting to inform their clients of the piracy and the potential drawbacks. They also do not support non-verified software[5].

There are a variety of alternatives that exist, including such concepts as *software aging*, which is interesting incidentally in that it is a paradigm where software is continually updated, so that the pirate has to continually update the cracked version as well. This taxes the pirates resources and is a novel way to secure software[1]. Another alternative is that of *software splitting*, which is when critical components of the software are hosted on a remote server[1]. The pirate, if they deconstruct the executable, will still be unable to use the software if the functionality is accessed from a remote server.

Problem Statement:

Since the software that we wish to authorize is going to be released to the public, we can assume a few things about our adversary:

1. The adversary has access to the executable code.
2. We cannot trust the Operating System of the machine the software will be installed on, because it is the adversary's.
3. The adversary will be able to prompt the server at will.
4. Messages that will be sent between the server and client can be eavesdropped, modified, recorded, or even invented by the adversary.
5. The adversary is limited to polynomial time, and typical resources of a high-end computer from the year 2015.
6. The adversary does not care about breaking all copies of the software, she only needs to break *one*.
7. The adversary will attempt to use Chosen Ciphertext Attack, if available.

With this adversary in mind, we can see how some implementations can fall short to common attacks. If the authentication message from the server is identical for every copy of software , the adversary can simply record that response and play it back at will to the software executable. If the authentication message from the client to server is the same as the message from the server to client, the adversary can simply mirror the client's transmission, breaking the authentication. Additionally, if the authentication messages are deterministic and depend only on the serial number, and as such are the same for every activation of one software executable, the adversary can simply manipulate to replay that authentication message, and while she has not broken the scheme for all copies of the software, she can simply copy that instance as much as she desires.

Turning to goals for this implementation, we can see that we want a solution that is resistant to replay, mirror, and spoofing attacks, and that allows the server operator to selectively deny certain software serial keys. Another goal is to design the protocol in such a way as to prevent the adversary from using the server as a decryption oracle. To obtain this, the server and client must not leak any information relating to the decryption or authentication used. Lastly, we want the scheme implemented to prevent a chosen ciphertext attack (CCA).

Related, but separately, we want the implementation to be easy to use for someone with little cryptographic knowledge. To that end it should use current best practices in a way completely opaque to the user. Ideally, the user would simply call a authentication function in their code before execution, and link a copy of the client Library.

Challenges:

In implementing this project there were a number of challenges related to the scope of the project, the details of implementation, and the adversarial model that had to be addressed. First off, the scope of the project was important in that it defined the amount of work necessary to achieve success. For my team, which included only myself, I had to keep that scope manageable.

Secondly, the details of implementation posed a challenge, in that many components have strict requirements. Cryptographic nonces cannot be reused under any circumstances, for instance, as doing so compromises security. Similarly, implementing an Authentication then Encrypt scheme (MtE) is not secure, while an Encrypt then Authenticate (EtM) scheme is secure. Other important technical pitfalls that had to be avoided were using old algorithms that are not considered secure anymore. MD5 and SHA-1 are two simple examples of this, another more interesting example is that of using SSL TSLv1.2 instead of using old legacy TSL that has vulnerabilities.

The most important challenge, in hindsight, that, unfortunately, wasn't fully addressed, was that of preliminary research. The research conducted before and during the project was crucial to the secure implementation and ultimate success of the project. In this sense, the research had to encompass the whole semester as new techniques were learned and applied to the project. In addition, understanding what has already been done in software authentication and further direction to take such work is crucial to preventing wasted effort.

Solutions:

The solutions to both the adversary proposed and the challenges previously elaborated are many and multi-faceted. As for the scope of the project, the solution I found was to limit the scope and try not to put so many things in my bag that it breaks. Another solution was to use resources that have done the nuts and bolts for you, the two important resources I used were OpenSSL and Crypto++. They had extensive algorithm libraries to call upon. Both have AES implementations, and Crypto++ has block ciphers, stream ciphers, AES, hash functions, HMAC, elliptic curve, and public key libraries.

As for the detailed implementation solutions, they will be elaborated in the implementation section, but briefly the key to solving that challenge was to be careful. Encryption, for instance, had to precede authentication in the client, otherwise the scheme is not secure. Reading the manuals for the code I used in my project was just a facet of being very careful. This was important on a practical level. When I make the call to create the SSL context, I used TLSv1.2 instead of the default. I only knew that the default supported old legacy connections like SSLv3, which is 18 years old and should not be used in future applications. In this way I was detail oriented and prevented a security vulnerability.

The other way in which details were chosen and technical pitfalls were avoided was to carefully study the material presented in lecture. That material was very helpful to flesh out my ideas for the scheme of this project. I can directly point to using Encrypt then MAC and using HMAC for authentication as being inspired from the discussions in lecture. From the beginning, where I had a very limited view of cryptography (public key cryptography was surprising to learn about, for instance), to the present, where I have a little better understanding of what to do and what not to do, the class was a primary source for research.

The last challenge that I faced, and definitely the most important of all, was the preliminary research. In this I can honestly say that the results weren't up to my standard. A more careful perusing of the CSULB digital library and other scholarly sources would undoubtedly have paid major dividends in reducing the difficulty of this project, as well as defining a better proposal and goals. I feel like the project I have done unfortunately reflects more the basic, naive ideas I had coming into the course than the modern understanding of cryptography that more research could have formed. Indeed, the simple thought such as "What does Microsoft do to protect their software from illegal copying?" could have resulted in much better preliminary research. Also, the idea of blocking keys from the server that have been compromised would have turned up with better research as well. With all these things said, however, I still strongly feel that the implementation scheme I have devised here, while perhaps not entirely novel, still incorporates the cryptographic elements available in Crypto++ and OpenSSL into a secure authentication system. The solution for this last point is one of those which has belated benefits. Preliminary research will take a more frontal role in my work as I move forward, and as such a valuable lesson has been learned.

Your Individual Work:

As the only member of my team, I had the full responsibility of all areas of this project. I was responsible for the research, the implementation, the scheme I created, and the decisions about what adversary to target and what libraries to use. I am happy with the scheme and tools I chose to use to implement the project. I devised this scheme which I believe protects against replay attacks, spoofing attacks, man in the middle attacks, prevents CCA attacks, and provides little or no information leaking from the server. I should have investigated and included denial of service attacks, and how to prevent the server from being swamped with requests. I also should have done more preliminary research and identified prior state of the art. Unfortunately, I am also responsible for underestimating the difficulty of implementing the scheme and not providing a working demo. Aside from that, I implemented two .cpp files, BM_Server.cpp and BM_User.cpp to test my authentication scheme. I created a self-signed certificate to test the scheme, and got these two files (once compiled), to communicate with each other. I laid out the foundations for the encryption and HMAC of the authentication key, and devised the unique ID for both the server and client executables. I also implemented a direction bit and 32 timestamp bits to be in the message as well.

Comparison of work with teammates:

Since I do not have any fellow team members, I will devote this space to the three biggest weaknesses of my performance with this project. The first, and most important weakness was the with the preliminary research for this project. If I had performed more research, and asked the right research questions, I would have had a stronger base to stand upon for the rest of the project. Unfortunately, when this project began I don't think I asked the right questions or answered the questions I did ask in time. As a result this scheme is somewhat simple. This weakness is countered, however, by the strength of the implementation I designed. I believe that I have created a scheme which has done all the goals I set out to accomplish, in terms of the adversary. I do not believe she can use the resources available to her (as I elaborated with the tenets above) to break this scheme.

The second major weakness of this project and my work is that I was unable to get the demo to work. Unfortunately I hit a few snags in the development process which took far longer to resolve than I anticipated. In particular I had a real time trying to get the self-signed SSL certificates to be read as legitimate. This doesn't detract from the fundamental ideas I present in this work, and I don't believe there are any insurmountable barriers to the work.

The last major weakness/strength I will detail here is the attention to detail that I kept up in my implementation. I was very careful with things so that I didn't inadvertently introduce any security loopholes. I wanted to be absolutely sure that the scheme was secure, and if that created redundant parts that hurts performance. There really isn't any need to verify something twice if one verification is secure is there? Also, this hampered my productivity, since I wanted everything to be just so before moving on. This is kind of a strength, however, in that it can produce quality work.

Shortcomings and their reasons:

The shortcomings I encountered have been covered in some detail above, but I will briefly reiterate them here. The first shortcoming was the preliminary research. The project could definitely have benefited from a comparison of current implementations and an analysis of their shortcomings. In addition, further research could have offered new ideas for implementation such as blocking specific serial keys from the server. The other major shortcoming, in my opinion, is my failure to have a working demo, which I can only attribute to unexpected problems with SSL. I really underestimated implementing security certificates.

Lessons Learned:

As a result of this project and the work involved, I have learned a few important lessons. The first, and most important, in my opinion, is that the preliminary research before a project is critical to the success of a project. The reasoning is thus: if you don't know what has been done before, you are bound to make the same mistakes as the ones before you. Also, the building upon older ideas is one of the most important aspects of modern science. In any future project I work on, I will specifically make sure that the opening research is both thorough and broad.

Next in importance of the lessons I learned from this project, OpenSSL and Crypto++ are really excellent open source libraries. Especially Crypto++, because it has many algorithms to implement security concepts such as pRNGs, HMAC, encryption, and key derivation. OpenSSL is also excellent, in that it is open source and thus is open and waiting for an adversary to break it. Crypto++ is the same way. It is enlightening to realize that even though these libraries are open source, they are actually stronger as a result.

Two smaller lessons I learned that are more personal and are common knowledge in modern cryptography, are that you can have an authentication scheme that resists replay attacks, and that an adversary attempting to execute software in his own virtual machine sandbox is not all powerful. The resistance to replay attacks was surprising to me due the nature of computers being deterministic. It seems that, if a computer is deterministic, you could simply copy the state of the executing software and then it follows that you could just replay that any time you wanted.

Implementation and Results:

The implementation followed a plan which I will lay out here. Two C++ files were created, one to be used by the server, and one to be used by the client. The general scheme of the communications between these two files would be one which is resistant to the adversary described above, namely one who has the executable of the software she wishes to use without authorization, who has the capability of eavesdropping, modifying, and deleting messages from the client and server, who can prompt the server, who has complete control over the operating system and physical system of a computer where the software may be installed, and who only needs to break one copy due to the deterministic nature of computers. If she breaks one copy, she can, at worst, simply copy the state of the machine and then load that state any time she wishes to use the software.

Program Authentication Scheme

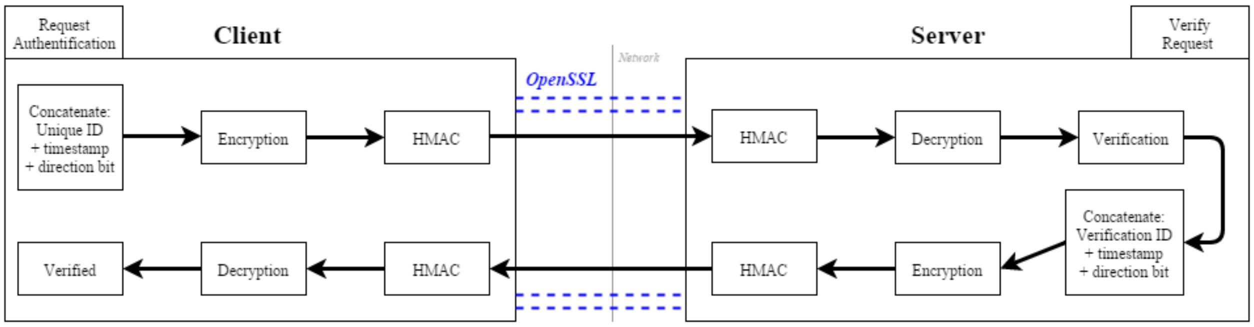
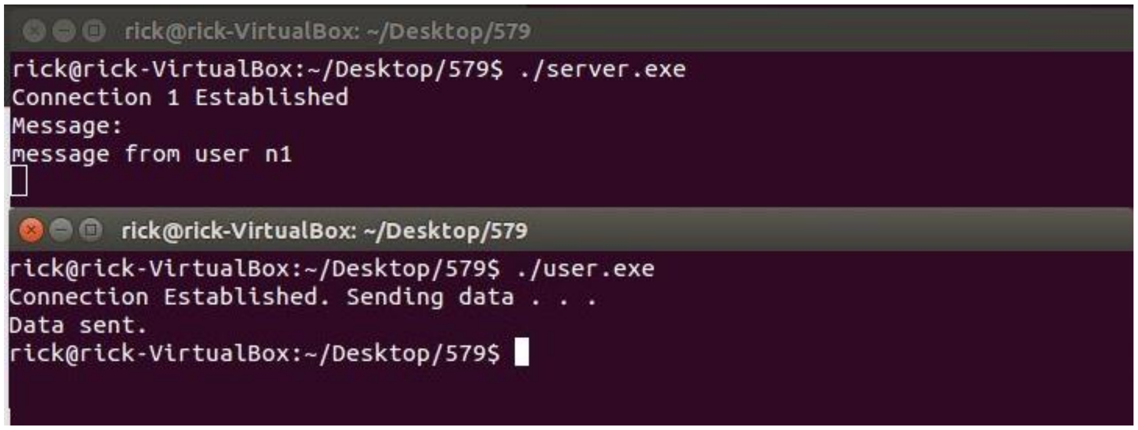


Figure 1 - Program Authentication Scheme Block View

A diagram of the secure communication scheme can be seen here. This broad overview shows the steps needed to authenticate the client software. First, the client, upon beginning execution, concatenates it's unique ID or serial number with a timestamp and direction bit, encrypts that message, uses HMAC to authenticate that message, then sends the result to the server. Before this ciphertext can be sent to the server, an SSL link has to be established to securely send that data. Once the server gets the ciphertext, it does the reverse of the client's operations. First, it authenticates the message using HMAC, then it decrypts the message and verifies the timestamp and direction bit. At this point, the server concatenates the unique response ID that correlates with the request ID with a timestamp and a direction bit. This concatenation is then encrypted and authenticated with HMAC and sent to the client over the same SSL network. The client then authenticates, decrypts, and verifies the contents of the message before commencing execution.

Now I will elaborate on the technical specifics of the scheme involved. The message from the client consists of 9 bytes - a 32 bit unique ID which is unique to the client, a 32 bit timestamp which signifies the result of a call to time(), and an 8 bit char which is either 0x01 if the message originates from the client, or 0x00 if the message originates from the server. These 9 bytes are then encrypted by AES 128 with a key which is derived from the unique ID and an Initial Value (IV) which is generated using a Crypto++ pRNG. Once this is done, the IV and encrypted message are concatenated and passed through the HMAC function, which provides an authentication tag with SHA-256. This HMAC result is concatenated to the encrypted message + IV, and then the ciphertext is ready to be sent to the server.

Before the server can accept any message, or even before the client can send a message, they have to form a connection to each other. This secure connection is done using OpenSSL with TLSv1.2. The client and server both have certificates and once the SSL handshake is completed the connection is established. An example of what the connection looks like from the terminal after both executables have been compiled can be seen in the figure.


rick@rick-VirtualBox: ~/Desktop/579
rick@rick-VirtualBox: ~/Desktop/579

```
rick@rick-VirtualBox:~/Desktop/579$ ./server.exe
Connection 1 Established
Message:
message from user n1
```

```
rick@rick-VirtualBox:~/Desktop/579$ ./user.exe
Connection Established. Sending data . . .
Data sent.
rick@rick-VirtualBox:~/Desktop/579$
```

Figure 2 - Terminal Result of Client-Server Communication

Since the SSL connection is established before any encryption is performed, once the ciphertext is ready it is immediately sent to the server. It can be seen from the contents of the message that this scheme prevents replay attacks. The timestamp and randomly generated IV ensure that the message is not the same every time. In addition, even if the time was set to be the same the encryption and HMAC are such that the message will not be the same. The unique ID allows the server to control whether a particular copy of the software is allowed to execute, and the direction bit prevents mirror-type attacks that pit the client against another client by redirecting the ciphertext away from the server and towards the same client or another client.

The server is selective in its responses. First, it takes the ciphertext from the client and authenticates it with the HMAC. if this succeeds, it then decrypts the ciphertext with the key generated from the unique ID and the IV which is concatenated to the encrypted message. Remember, these functions are used from the Crypto++ library and use the specific algorithms of AES 128 and SHA-256 for the encryption and HMAC, respectively. The next step is the validation of the message. The unique ID is checked and the corresponding response ID is taken to become the start of the return message. The timestamp is checked, and if the stamp is within 5 seconds it is accepted. The server takes its own timestamp and concatenates it to the response ID. The direction bit is checked last, and if it is valid the direct bit is appended to the message. Now, with the message validated, the server takes that data and encrypts it, just like the client, with AES 128 using a key generated from the serial number and a generated IV. Similarly, the server then concatenates the IV with the encrypted message and uses HMAC on the result to generate an authentication tag. This is appended to the message and the ciphertext is thus then sent to the client.

Note that the server and client both check timestamps, direction bits, and the unique ID. This makes it difficult for the adversary to use copies of old ciphertexts to fool the system, or copies of ciphertexts intended for one software copy to fool another copy. If the message is intercepted and tampered, the MAC check will fail. If the message is held and examined for longer than 5 seconds the authentication will fail because of time stamp. One serious area for further research is to examine how a denial of service attack might work, and how to avoid such an attack. I did not take this attack into account so I will not elaborate further on this topic.

Once the message from the server is received, the client performs the verification in a manner similar enough to the way the server does it that it needs no further explanation. If this verification is successful, the software can then proceed with execution.

The results I obtained were in accordance with this scheme. I was unable to fully implement the scheme as I elaborated here, but I believe the ideas remains sound. In particular, I had some trouble implementing the SSL connection and getting the self-signed certificates to be read as valid.

Conclusion and Future Work:

In conclusion, this scheme may be secure but definitely needs a comparison with current state of the art and similar schemes. The concept is simple enough that I am sure it has been implemented elsewhere, by people before me. Microsoft's serial key authentication immediately springs to mind as a similar type of scheme. The idea of blacklisting certain serial numbers from authentication is an area which also deserves further work, and promises an extra layer of control for the software developer.

Investigation into denial of services attacks is also another area where further analysis of this scheme is warranted. Since the conditions which allow code execution are so stringent, any adversary which tampers with the messages can deny the client access to the software. In many environments this type of weakness, if it leads to downtime of mission-critical software, is unacceptable.

Another area of possible investigation is to analyze the scheme and how secure it is. A paper detailing an attack on this implementation may provide an excellent resource for improving the set up. Even the attempts are valuable if they do not work, since they can thus demonstrate the strength of the scheme in some respects.

Bibliography:

1. **Virtual leashing: Internet-based software piracy protection.** *Ori Dvir, Maurice Herlihy, Nir Shavit.* Proceedings - International Conference on Distributed Computing Systems, p 283-292, 2005. Institute of Electrical and Electronics Engineers Inc. 2001.
2. **A Robust Approach to Prevent Software Piracy.** *Ajay Nehra, Rajkiran Meena, Deepak Sohu, and Om Prakash Rishi*. June 2015
3. **Preventing Piracy, Reverse Engineering, and Tamperin**g. *Gleb Naumovich, Nasir Memon.* Polytechnic University. November 2003
4. **Microsoft's Software Protection Platform: Innovations for Windows Vista and Windows Server "Longhorn"**. *Microsoft Corporation.* White Paper. October 2006

5. **Windows Activation Technologies in Windows 7.** *Microsoft Corporation.* June 2009

Appendix:

Selected code snippets:

Functions to create and verify messages (Client):

```
void GenAuthCode(char* bytes, long ID)
{
//authentication code is composed of unique ID, timestamp, and
direction bit concatenated into 9 chars.
//bytes must be an array of size 9

//ID
bytes[0] = (ID >> 24) & 0xFF;
bytes[1] = (ID >> 16) & 0xFF;
bytes[2] = (ID >> 8) & 0xFF;
bytes[3] = ID & 0xFF;

//TimeStamp
time_t T = time();

bytes[4] = (T >> 24) & 0xFF;
bytes[5] = (T >> 16) & 0xFF;
bytes[6] = (T >> 8) & 0xFF;
bytes[7] = T & 0xFF;

//Direction Bit
bytes[8] = 0x01;

}

int VerifyVerifCode(char* bytes, long goodID)
{
     //verify the ID code, timestamp, and direction bit
     long ID = (bytes[0] << 24) + (bytes[1] << 16) + (bytes[2]
<< 8) + bytes[3];
     time_t T = (bytes[4] << 24) + (bytes[5] << 16) + (bytes[6]
<< 8) + bytes[7]

     if (goodID != ID)
     {
          cout<<"wrong ID\n";
          return 0;
```

```
	}
	else if ( (T >= (time() + 5)) || (T <= (time() - 5)) )
	{
		cout<<"wrong time\n";
		return 0;
	}
	else if (bytes[8] != 0x00)
	{
		cout<<"wrong direction bit\n";
		return 0;
	}
	else
	{
		return 1;
	}

}
```

Functions to create and verify messages (Server):

```
int VerifyAuthCode(char* bytes, long goodID)
{
	//verify the ID code, timestamp, and direction bit
	long ID = (bytes[0] << 24) + (bytes[1] << 16) + (bytes[2]
<< 8) + bytes[3];
	time_t T = (bytes[4] << 24) + (bytes[5] << 16) + (bytes[6]
<< 8) + bytes[7]

	if (goodID != ID)
	{
		cout<<"wrong ID\n";
		return 0;
	}
	else if ( (T >= (time() + 5)) || (T <= (time() - 5)) )
	{
		cout<<"wrong time\n";
		return 0;
	}
	else if (bytes[8] != 0x01)
	{
		cout<<"wrong direction bit\n";
		return 0;
	}
	else
	{
		return 1;
	}
}

void GenVerifCode(char* bytes, long ID)
{
//authentication code is composed of unique ID, timestamp, and
direction bit concatenated into 9 chars.
//bytes must be an array of size 9

//ID
bytes[0] = (ID >> 24) & 0xFF;
bytes[1] = (ID >> 16) & 0xFF;
bytes[2] = (ID >> 8) & 0xFF;
bytes[3] = ID & 0xFF;

//TimeStamp
time_t T = time();

bytes[4] = (T >> 24) & 0xFF;
bytes[5] = (T >> 16) & 0xFF;
bytes[6] = (T >> 8) & 0xFF;
bytes[7] = T & 0xFF;

//Direction Bit
bytes[8] = 0x00;
}
```

Pseudocode Implementations of EtM protocol (Client):

```
//******** ENCRYPTION THEN HMAC *************
//generate AES IV
//Encrypt with AES 128 with key + IV
//Concatenate IV with ciphertext
//HMAC <SHA256> the concatenated ciphertext with key
// concatenate the ciphertext and MAC
//send the result
```

Pseudocode Implementations of EtM protocol (Server):

```
//******** HMAC then DECRYPT *************
//Check MAC integrity
//Decrypt with AES 128 with key + IV (given)
//check that the result is valid
//Create verification Code if valid
//******** ENCRYPTION THEN HMAC *************
//generate AES IV
//Encrypt with AES 128 with key + IV
```

```
//Concatenate IV with ciphertext
//HMAC <SHA256> the concatenated ciphertext with key
// concatenate the ciphertext and MAC
//send the result
```