Handling False Data Injection Using Principals of MD5 Algorithm

Veena Jain, Prof. Vidya Bodhe
Student, Dept. of CSE/IT, Abha Gaikwad-Patil College of Engineering, Nagpur, India
Professor, Dept. of CSE/IT, Abha Gaikwad-Patil College of Engineering, Nagpur, India

ABSTRACT: In my previous paper, I have presented the brief idea of how to deal with the major problem of cyber attacks in wireless networks. False data injection attacks have recently been introduced as an important class of cyber attacks against smart grid’s wide area measurement and monitoring systems. These attacks aim to compromise the readings of multiple power grid sensors and phasor measurement units in order to mislead the operation and control centers. There are several methods suggested by many renowned experts in the field of communications. One such algorithm, which is considered as error prone is MD5 algorithm. This method is suggested by Mr. Ronald Rivest in 1992. All though it is found that the method has some shortcomings, I still think that this method offers one of the suitable solution for the common problem of false data injection.

KEYWORDS: BECAN, Data Injection, MD5 Algorithm

1. INTRODUCTION

MD5 digests have been widely used in the software world to provide some assurance that a transferred file has arrived intact. For example, file servers often provide a pre-computed MD5 (known as md5sum) checksum for the files, so that a user can compare the checksum of the downloaded file to it. Most unix-based operating systems include MD5 sum utilities in their distribution packages; Windows users may install a Microsoft utility, or use third-party applications. Android ROMs also use this type of checksum.
MD5 processes a variable-length message into a fixed-length output of 128 bits. The input message is broken up into chunks of 512-bit blocks (sixteen 32-bit words); the message is padded so that its length is divisible by 512. The padding works as follows: first a single bit, 1, is appended to the end of the message. This is followed by as many zeros as are required to bring the length of the message up to 64 bits less than a multiple of 512. The remaining bits are filled up with 64 bits representing the length of the original message, modulo 2^64.

The main MD5 algorithm operates on a 128-bit state, divided into four 32-bit words, denoted A, B, C, and D. These are initialized to certain fixed constants. The main algorithm then uses each 512-bit message block in turn to modify the state. The processing of a message block consists of four similar stages, termed rounds; each round is composed of 16 similar operations based on a non-linear function F, modular addition, and left rotation. Figure 1 illustrates one operation within a round. There are four possible functions F; a different one is used in each round:

- $F(B, C, D) = (B \land C) \lor (\neg B \land D)$
- $G(B, C, D) = (B \land D) \lor (C \land \neg D)$
- $H(B, C, D) = B \oplus C \oplus D$
- $I(B, C, D) = C \oplus (B \lor \neg D)$

The 128-bit (16-byte) MD5 hashes (also termed message digests) are typically represented as a sequence of 32 hexadecimal digits. The following demonstrates a 43-byte ASCII input and the corresponding MD5 hash:

MD5("The quick brown fox jumps over the lazy dog") =
9e107d9d372bb6826bd81d3542a419d6
Even a small change in the message will (with overwhelming probability) result in a mostly different hash, due to the avalanche effect. For example, adding a period to the end of the sentence:

MD5("The quick brown fox jumps over the lazy dog.") =
e4d909c290d0fb1ca068ffaddf22cbd0

The hash of the zero-length string is:

MD5("" ) =
d41d8cd98f00b204e9800998ecf8427e

The MD5 algorithm is specified for messages consisting of any number of bits; it is not limited to multiples of eight bit (octets, bytes) as shown in the examples above. Some MD5 implementations such as md5sum might be limited to octets, or they might not support streaming for messages of an initially undefined length.

md5 deep is a software package used in the computer security, system administration and computer forensics communities to run large numbers of files through any of several different cryptographic digests. It was originally authored by Jesse Kornblum, at the time a Special Agent of the Air Force Office of Special Investigations. As of 2015, he still maintains it. The name md5deep is misleading. As of version 2.0, the md5deep package contains several different programs able to perform MD5, SHA-1, SHA-256, Tiger192 and Whirlpool digests, each of them named by the digest type followed by the word deep. Thus, the name may confuse some people into thinking it only provides the MD5 algorithm when the package supports many more.

md5deep can be invoked in several different ways. Typically users operate it recursively, where md5deep walks through one directory at a time giving digests of each file found, and recursing into any subdirectories within. Its recursive behavior is approximately a depth-first search, which has the benefit of presenting files in lexicographical order. On Unix-like systems, similar functionality can be often obtained by combining find with hashing utilities such as md5sum, sha256sum, or tthsum.

md5deep exists for Windows and most Unix-based systems, including OS X. It is present in OS X's Fink, Homebrew and MacPorts projects. Binary packages exist for most free Unix systems. Many vendors initially resist including md5deep as they mistakenly[citation needed] believe its functions can be reproduced with one line of shell scripting. The matching function of the program, however, cannot be done easily in shell.

Because md5deep was written by an employee of the U.S. government, on government time, it is in the public domain. Other software surrounding it, such as graphical front-ends, may be copyrighted.

Considering all these features and advantages, I personally decided to implement the algorithm to solve to some extent, the problem of false data injection in grid control system. The most important point about md5 and its variants is that is license free.

**MD5 Message Digest Algorithm**

Sometimes we have to serialize objects, e.g. to send them over a network, store and restore them locally or for any other reason. Now it can be useful if we would know after the deserializing process, if the object has been restored correctly. Especially, if you have objects which have internal states or if you must manage multiple instances of a class. A possible solution to this problem is using the System. Guid struct to identify the objects. But in this way, you cannot be sure that the internal states, etc. were deserialized correctly.

A commonly used technique in the Internet is to provide a MD5 - Hash String so the receiver can compare if the file has been transmitted without any modifications.

**II. LITERATURE SURVEY**

1. BECAN Scheme

   A novel bandwidth-efficient cooperative authentication (BECAN) scheme for filtering

1) First, we study the random graph characteristics of wireless sensor node deployment, and estimate the probability of k-neighbors, which provides the necessary condition for BECAN authentication;

2) Second, we propose the BECAN scheme to filter the injected false data with cooperative bit-compressed authentication technique. With the proposed mechanism, injected false data can be early detected and filtered by the en-route sensor nodes. In addition, the accompanied authentication information is bandwidth-efficient; and
3) Third, we develop a custom simulator to demonstrate the effectiveness of the proposed BECAN scheme in terms of en-routing filtering probability and false negative rate on true reports.

2. Early detecting the injected false data by the en-route sensor nodes

The sink is a powerful data collection device. Nevertheless, if all authentication tasks are fulfilled at the sink, it is undoubted that the sink becomes a bottleneck. At the same time, if too many injected false data flood into the sink, the sink will surely suffer from the Denial of Service (DoS) attack. Therefore, it is critical to share the authentication tasks with the en-route sensor nodes such that the injected false data can be detected and discarded early. The earlier the injected false data are detected, the more energy can be saved in the whole network.

3. Gang Injecting False Data Attack

We introduce a new stronger injecting false data attack, called gang injecting false data attack, in wireless sensor networks. This kind of attack is usually launched by a gang of compromised sensor nodes controlled and moved by an adversary A. As shown in Fig. 2, when a compromised source node is ready to send a false data, several compromised nodes will first move and aggregate at the source node, and then collude to inject the false data. Because of the mobility, the gang injecting false data attack is more challenging and hard to resist.

III. PROPOSED METHOD

The .NET Framework gives us a struct to uniquely identify our objects, the System.Guid struct in the mscorlib.dll. This struct can be used to give each class its own identifier. And that's the crux of the matter. What we need is not an identifier for the class, we need an identifier for each instance of the class. Implicitly this identifier must also represent some internal values like state. Otherwise our recipient of the object cannot be sure, that he has received / deserialized the same object. Also our recipient cannot "create" a GUID on his own. Once it is created by the sender, it is not reproducible.

We must also provide a functionality, which can be executed by both, sender and recipient, to identify an object. This identifier must also implicitly regard on the fields which are relevant for this object. And these relevant fields can be different for each class!

The idea I had was to use MD5 hashes for that. Each object has a built-in function called .GetHashCode(). This method returns an Integer, although according to the name of the method, you would expect a string. That's because these HashValues are intended to be used as Keys in e.g. a HashTable. But fortunately, there exists a class named MD5CryptoServiceProvider in the System.Security.Cryptography namespace. Unfortunately, this class is not easy to use. The main problem for most programmers could be that the class only accepts a byte-array as input and not a reference to an object. So I decided to wrap all the needed functionality into a generator class. This class could then generate the Hash for me, and I have to write just one line of code.

The code file above contains a class called MD5HashGenerator. This class has a static method .generate Key(Object source Object), which does the "magic" for you. Include the class into your project, and use it as follows:

To use the class (as a publisher), you have to do the following things:

1. Mark the object as Serializable(). Mark all variables which should not be serialized as Non Serializable().
2. Call the static method MD5 Hash Generator. Generate Key(Object source Object). You get the MD5 - Hash for the object as a String.
3. Serialize the object, publish / store it and the hash.

If you are the receiver, then:

1. Deserialize the received object.
2. Call the static method MD5 Hash Generator. Generate Key(Object source Object) on the deserialized object.
3. Compare the hashes.
4. Because we use the system method Date Time. Now to initialize the field just A Time, each instance of the class should be different. It is important to "mark" the class as Serializable, because this is asked by the MD5HashGenerator-class.
5. The generator class uses the BinaryFormatter for serialization, so all fields (whether they are private or not are automatically included in the serialization process). But exclude handles and pointers, if you are using them MD5 Crypto Service Provider class to generate the Hashstring. I encapsulated the use of the method in the ComputeHash(byte[] objectAsBytes) method. The MD5CryptoServiceProvider class wants a byte array as input. It does not accept an object directly. What you get out of it is not a string as we would like to have, but a byte array. Therefore I added the conversion from byte array to Hex. The conversion is done by using the Byte.ToString() method. The method accepts a formatstring as input. And "X2" here means that each byte is converted into a two-char-string-sequence (e.g. 01011100 => 5C or 00000111 => 07).

Now there is still the question as to how to convert an object into a byte array. We know that our object is serializable. So we can serialize it into the memory (using a MemoryStream and a BinaryFormatter) and getting out of the memory the needed byte array. Because the whole thing should be thread-safe, we lock the Serialization of the object.

IV. A SURVEY ON

Generating MD5-hashes can be useful, if you must have a procedure both sides can execute to ensure the uniqueness and changeless serialization / deserialization of objects. The most difficult part for me was to convert an object into a byte array and the conversion of a byte array to an Hex - String. Using Guide is also a possibility. But the Guide is created when the object is initialized and the consumer cannot "recreate" the Guide to ensure that no changes on the object were done. He just knows that he has received the same object the producer has created.

REFERENCES

[3] MD5 Hash Algorithm - Version 1.0 (Part of W3C DSig 1.0 Recommendation) http://www.w3.org/TR/1998/REC-DSig-label/MD5-1_0


