Christian Hall

Encryption in the cyber security industry is seldom backed by strong system peripherals and strategy. This software idea has benefits including deception and creativity using private randomizing, fake packets, a network umbrella and one time use techniques. This paper concerns all companies interested in further securing their communication and data transfer systems.

SEEM

Software Elusive Encryption Method

Table of Contents

[Defense 2](#_Toc422172556)

[Data Transfer 3](#_Toc422172557)

[Data Flow 4](#_Toc422172558)

[Figure 1 4](#_Toc422172559)

[Mode of Operation 6](#_Toc422172560)

[Figure 2 7](#_Toc422172561)

[Method Description 7](#_Toc422172562)

[Figure 3 8](#_Toc422172563)

[Network Umbrella 9](#_Toc422172564)

[Private Randomizing 9](#_Toc422172565)

[Review of a secret randomizing process 10](#_Toc422172566)

[Optimization 11](#_Toc422172567)

[Time 12](#_Toc422172568)

[One Time Use 13](#_Toc422172569)

[Proof of Concept 14](#_Toc422172570)

[Figure 4 15](#_Toc422172571)

[Figure 5 15](#_Toc422172572)

[Fake Packet Option 16](#_Toc422172573)

[Conclusion 16](#_Toc422172574)

[Bibliography 18](#_Toc422172575)

SEEM – Software Elusive Encryption Method

Christian Hall

June 15, 2015

## Defense

The ability to think like a criminal is an attractive trait of Cyber Security Engineers. Using this trait effectively in this project will help us decide which secrets to keep from the attacker. We will not be keeping the algorithm of our method a secret, but by attempt to deceive the attacker, we can increase the security of our system. Deliberately making a system that intends on deceiving the attacker gives us the ability to better determine what they would do in a given circumstance, and we can assume they will have a smaller number of options to move forward with an attack on our system.

Kerckhoffs’s principle, developed by Auguste Kerckhoffs, says that we are to assume that the enemy will immediately gain full familiarity with our systems (Kowalczyk). Because of this, the security level of an encryption algorithm is said to be solely dependent on the secrecy of the key and, of course, not the algorithm. The part of the principle that people often overlook is referenced by James Massey. He says that the security of the cryptosystem should not only depend on the secrecy of the key, but the secrecy of the private randomizer (Massey).

Sun Tzu in *The Art of War* says to “mystify, mislead, and surprise the enemy” (Tzu). The Internet security world creates a classic case of “good guy” versus “bad guy”, police versus criminal, or good versus evil. Similarly, in our case, we have hacker versus defender. Sam Houston from the film “The Alamo” responds to his soldiers in this way when asked why he is continuing to retreat:

In 1815, Napoleon escaped from Elba. He moved swiftly to consolidate before the Grand Alliance could move against him. Wellington, with fewer men, retreated ahead of Napoleon, forcing Napoleon to chase him through Belgium. Wellington had a vision of a battlefield, he did not know where it was, but he knew he could know it when he saw it. He continued moving waiting for that ground and for Napoleon to make a mistake. Gentlemen, I do not consider myself to be Wellington; Santa Ana, however, considers himself to be Napoleon – the Napoleon of the West. I will continue to retreat gentlemen, until I find the ground in my vision and, when Santa Ana makes him mistake, I will attack (Quaid).

Sam Houston reminds us security engineers that defense is all about finding the right time to attack. We have to find a way to combat attack vectors we have yet to foresee. We will eventually be understood and we have to think ahead of our enemies. Therefore, we need to continue to create defenses that are in and of themselves attacks on the enemy.

In this project, we will work on creating additional entropy to encryption so that any kind of attacker will become confused. To do so, we are going to relate to an encryption mode of operation called Cipher Block Chaining (CBC) to create secure communications. Our goal is to confuse or deter an attacker all together with private randomizing, fake packets and one-time use techniques. If we are successful, we can send and receive messages or files securely using the same software.

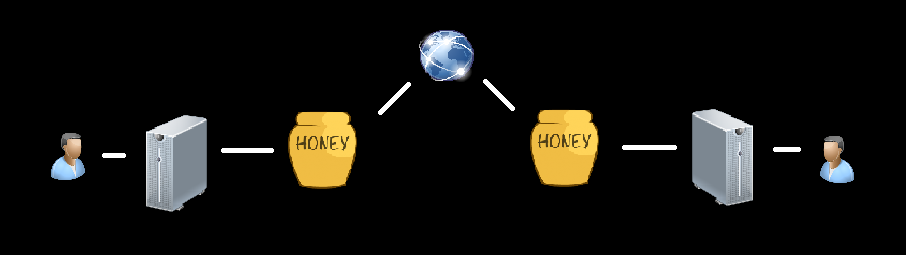
## Data Transfer

This method focuses on communication between two people. However, if you give each person the software needed to complete these communications, then it’s likely that the private randomizing will no longer be private. These randomizing techniques should be infused into software on a server. You will want to use a server that is managed locally, so that it does not have direct connected with the Internet. This should be congruent with what is happening to the encryption keys for the encryption that this mode is using. The software on the server will be able to send and receive messages for encrypting and decrypting.

### Data Flow

At this point, the process inside the company needs some explaining. What does the flow look like server side? These messages can be received over the Internet by a honeypot that is designed to communicate with the Internet and the local server. The honeypot is a final check for viruses before it is sent over the local area network (LAN) to the server. The information is then received by the server, sorted and decrypted by the software and then dumped into a final, simpler instant messaging type program on the company’s user’s computer. This way, not only can they have a simple way to receive these secure messages from the server, but they can also send some of their own messages back to the honeypot, and then to the server. The overall goal of the interface would be simplistic and include a way to decide who you are talking to and give users and option to not only send messages but to send files and to quickly download files that have been received. An illustration will help show the flow of the data coming from the sender.

### Figure 1



From left to right: User to server to honeypot to Internet to honeypot to server to user

Ideally, true random number generators (TRNGs), which are usually derived from a variety of measurements in nature, are accessible to this server so that pseudo random number generators (PRNGs) are not used because they can become replicable; therefore undermining and defeating the randomizing process. An example of this is when I run a program in the programming language C++, even if I change some of the code, the program will produce the same random numbers as it did previously. The only way it will produce different random numbers is if you change the code that has to do with random numbers. In this fashion, it’s possible to recreate these random numbers if you have the given code and the exact time the program was compiled.

To fix this, Mads Haahr introduces to us the way that RANDOM.ORG creates random numbers. They use a physical phenomenon of a radioactive source. There is no way to determine the point in time in which a radioactive source decays. Apparently, without much effort, these timing can fairly easily be inputted into a computer. This technique that is used to create random numbers is used by, for example, HotBits service at Fourmilab in Switzerland and RANDOM.ORG (Haahr). In summary, we would want to focus on using only TRNGs on our servers for our software using this method.

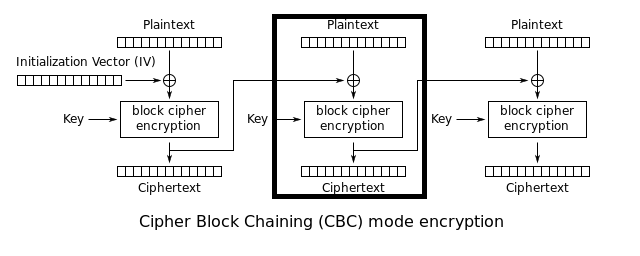
## Mode of Operation

Encryption is a powerful tool, but we can’t rely solely on encryption to keep our communications secure. To increase the effectiveness of the procedure surrounding encryption, such as a block cipher encryption, we would implement a block cipher mode of operation. This basically determines how each block of the encryption will interact with the other. The first and most basic encryption mode of operation is Electronic Codebook (ECB), which does not chain each encrypted block together. Cipher Block Chaining (CBC) mode of operation, for example, does chain each block together. More explanation included momentarily. In all effective encryption modes of operation (which doesn’t include ECB), an initialization vector (IV) is used to include randomness into the encryption. Most of the time, as long as you don’t use that same IV while using the same key, the IV does not need to be a secret.

This means that, at this point, the attacker knows your algorithm (i.e., AES encryption with CBC mode of operation), your IV, and all they need to do is find your key to be able to decrypt your message. However, if you can add to the number of randomizing techniques you can keep a secret, then you can increase the overall security of your systems or communications by exponential magnitudes. While we don’t use an IV, we use similar techniques for randomizing. Point being, randomizing at the initial step of your encryption is already a common theme to increase effectiveness.

To help understand what a CBC mode of encryption looks like a figure is included. A small chunk of our message, the plaintext, goes into a “block cipher encryption” where the entire encryption algorithm is ran. Afterwards, we have that chunk of encrypted text, the ciphertext, and we have more chunks to encrypt. The next chunk of plaintext uses a function called a XOR function that essentially makes the outcome of the next plaintext reliant on the value of the last ciphertext. That last concept is the important part; each part of our original message is reliant upon the last part of the message. As referenced to in the figure below, the outlined section is what each packet in our method will contain. Each of those packets will be intermingled among fake packets but, as explained, the order of the original packets is essential to being able to recreate the message. Therefore, a method to do so like a positional ID will be required.

### Figure 2



## Method Description

To begin, we start with some data to transfer. This data can be in the form of a message or in the form of chopped up bits of a picture or a hard drive. No matter what, the data will be sent, it will be encrypted, and it will contain an identifying variable so that the end program can recreate the original data in the correct order. The reason this is necessary is because in between the bits of data being sent will be random dummy packets. A key point of warning is that the more data to be encrypted, the exponentially longer the process will take because of how many times the program is encrypting. These dummy packets are fake packets that will contain a random sequence of bits. As an additional disclaimer, the program I am going to use to help show how this works will work only in ASCII characters (a, b, c) so these dummy packets will contain a random sequence of letters with a random length. Also, the amount of dummy packets will be randomized. For example, we’ll have two real words followed by some random packets. For an overview of what the packets will look like, refer to this table. It has the timestamp (which defines the order in which it is sent), the position it needs to be reordered in for decryption, and its ID (to determine if it’s a real packet or not).

### Figure 3

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Timestamp | 0.35 sec | 1.07 sec | 1.52 sec | 2.91 sec | 3.22 sec | 3.24 sec |
| Position | 2 | 1 | 7 | 4 | 1 | 3 |
| Data | are | What | the (fake) | doing | by (fake) | you |
| ID | 3 (odd) | 1 (odd) | 2 (even) | 9 (odd) | 6 (even) | 1 (odd) |
| Fake | (no) | (no) | (yes) | (no) | (yes) | (no) |

\*information inside parenthesis is not inside of the packet

The benefit of using a CBC block cipher is that you can’t conduct frequency analysis on it. So, if I have a message to send and I encrypt “the” at the beginning of the sentence, and I run into a “the” in the middle of the sentence, the encrypted text (each in individual packets) will be completely different. As reference to the earlier explanation of why ECB is insecure, we, additionally, don’t want to use ECB because it is prone to frequency analysis (the encrypted text of “the” will be the same as another “the” if we don’t chain the blocks together).

What we want to take a look at now is what would happen if an attacker did penetrate the system. This is likely to happen and is assumed to happen at some point. Say that you want to send a message. If your message is 96 letters in length, then you can fit it into 6 different blocks because each letter is one byte (or eight bits). This is important to us because if the average word length in your message is 4 letters (it probably isn’t but let’s say it is for simplicity’s sake), then you have about 24 words. The attacker is forced to consider 24 possible words in each position in the sentence (this is assuming you have no duplicate words), and 24 total positions. Therefore, we have 24 to the exponent of 24 possibilities for them just to determine the correct reordering of the sentence. This is a total of 1.3 × 1033 possibilities.

## Network Umbrella

For an attacker to even get to that situation, where he trying to determine the correct order of the sentence, is extremely unlikely. One of the main features of this method using your own software to communicate by sending packets over the Internet is that the traffic is covered by all sorts of other traffic. If I’m an average employee at a company and our router is channeling traffic of 15 other users then my traffic is harder to keep track of anyhow. This is also assuming that an attacker has found a way to sniff traffic on your network which is a grand gesture in itself if you have a decent security setup. The attacker that has a WireShark dump of your router’s traffic can, worst case scenario, easily organize and duplicate all of the packets sent from your honeypot to another computer. This is to show that even gathering all of the packets and distinguishing the real and the fake ones can be difficult, and even if they manage to get to that point their journey is far from over.

## Private Randomizing

In my proof on concept for the algorithm the private randomizer is simple, like randomizing between zero and six fake packets, numbers, or randomizing the number of letters in a word and randomizing those characters. However, randomizing can become as in depth as you let it. Creating a more in depth randomizing technique does not necessarily mean that you are complicating the system. The randomizing can potentially make it extremely difficult for an attacker to guess at what’s happening, but the random algorithm can remain simple.

For example, I could create a process where it’s impossible to reverse and to know what has happened. This is beneficial so the hacker can’t understand what happened, or how the algorithm worked, but it still can make sense for the program. A process like this would include a process as follows: generate a random character in the alphabet, turn that letter into its corresponding ASCII number and then find the remainder of that number after dividing by 10. You know that because the ASCII numbers are only in a range of 97 to 122. This means that the minimum remainder you can have if dividing by 10 is 0, and the maximum remain if diving by 10 is 9, or else another 10 would go into the original number. We can take the remainder and add 1, and it seems like we just created a random number between 1 and 10, when it fact we did nothing even close to that.

### Review of a secret randomizing process

1. Generated random letter: a-z
2. Relate to corresponding ASCII text: 97-122
3. Use a Mod 10 function (find the remainder if diving by 10): 0-9
4. Add one to the resulting number if desired for trickery: 1-10

The private randomizing is included with everything else as part of the security implementation for this procedure and it all facilitates to create a multi-layered process to strengthen security.

I’ve talked about how I used my ID as the data that’s included in the next chunk. In my proof of concept, I decided to make the distinguishing characteristic quite simple. The point to gather here, however, is this the ID is not reversible without the key. The key is that an even ID determines a real packet, and an odd ID determines a fake packet. This is something we’d want to keep secret. For example, if we wanted to keep how the ID determines a packet a secret we could have our SEEM server on an offline server that calls an encrypted header file. This way, the actual written algorithm is not stated inside of the program. Either way, the attacker would have to have access to the encrypted file or the program to determine what the *randomizing ID* function does.

At this point, we have a secret fake packet creation formula, a secret randomizing ID function and a secret encryption key. This is going to increase the effectiveness of our known algorithm and strike an attempt at increasing the overall security by magnitudes.

It is said that the time it takes to brute force your system is a good baseline for how secure it is. We are encrypting a large amount of data, and brute force would need to be used on the key. Also, brute force would need to be using on our private randomizing techniques because like I have mentioned.

## Optimization

This method has a bright future. When it comes to including random functions, there are a lot of different options. Ideally, a random time table would be included. If you are sending data and the program divides that data into chunks and includes random packets, every single packet could be sent at a random time. If we decide that we want all packets to be sent within four seconds, we can send those packets all at different times within four seconds. The receiving program would take longer to receive the data in this case. If it takes another couple of seconds for the server to reconstruct and then redirect the data to the user, then this can easily add another four seconds onto the total time it takes until the end user can receive the data.

In terms of fake packet creation and sending, the packets will all be created at the beginning so that the block cipher encryption can encrypt them all in a chain. This means that after they all need to be put back together in the same exact order so that decryption can be possible.

## Time

It is vital that the operators of this program understand the CIA triangle. The CIA triangle consists of confidentiality, integrity and availability (Perrin). A brief synopsis of such is that confidentiality generally regards access levels or keeping certain things a secret from certain people, integrity refers to replicating hash codes to ensure that data is not modified or deleted in any way, and availability defines not only whether it’s accessible, but how quickly it is accessible. Before this point in the paper, I have neglected to shed light on the speed of the operations before fully explaining how it will all work. The reality is, availability of this software is extremely low. In our case, the speed of our operations is slow. The time is takes to receive a simple message can easily exceed eight seconds. This, however, is expected. It is widely known that to be at the top of your game, two sides of the CIA triangle need to be squared away (well, “triangle”d away). The third side of the triangle is generally considered to take way too many resources to achieve then possible or necessary.

In our case, we can say in confidence that we are heavily focused on the confidentiality and the integrity part of the triangle. We want to make sure that nobody has access to the data that we have no given permission to and that the integrity of our data is checked afterward because we will know if any of our information has been changed, tampered with or deleted. Overall, it’s necessary to have two sides of the triangle, but not worth the resources to attempt to get all three. The goal of our program is to maintain security even at the expense of time.

## One Time Use

One of the most popular ways in today’s security world in 2015 is to use the idea of “One-time use”. One time use combined with multi-layered security creates an environment that’s practically impossible to replicate by an attacker or hack or even find a window of time where it would be possible to find a way around the system. A great example of this is Google’s second level of login defense for user’s which is its text message authentication. This is called “2-Step Verification”. If a user logs in from a different machine than usual, or there is some suspicious behavior on their account (Google definitely has a more in depth procedure for determining “suspicious”), then after the user types in their password the user will be sent a text message with a six digit code to enter in. When Google describes it, they say “each code can only be used once” (Google). Blizzard Entertainment, however, has another great example of a one-time use application for account authentication called the Blizzard Authenticator. If you have this method set up, you try to log in and then you type in a code generated on your authenticator, which is generated every 30 seconds. Each code lasts approximately 2 minutes (it holds about 4 codes in memory for user quality’s sake). This eight-digit numeric code is used one time so that you have an extra layer of security on your account, however, if you type in the wrong password, it will still ask you for the authenticator code.

The number of 8 digits can be repeated eventually, but not very often. If you take the maximum number of possibilities that an 8-digit number can procure (because each digit has 10 possibilities), you will get 108 which equals 100 million different possible combinations. If you can generate one code every 30 seconds then the average time another code will be generated that has been used before will be about 95 years, which is longer than the average lifetime.

Google also has this type of authentication if you use the Google Authenticator. This mobile application from Google, however, allows other companies to have the same level of security. It allows multiple different authenticators to reside in the same app as long as you connect your serial code to the desired compatible program.

The entire value and purpose of this explanation is to relate an extremely forthright and common best security practice of today, show its use and advantage, and to explain that my program has the same capabilities. When you randomize in multiple layers (random time intervals, randomly inserted dummy packets, random size of dummy packets and random characters inside dummy packets), the likelihood of repetition is unlikely. Also, when we are using my process for encrypting and sending messages, the randomizing is done per message or text. It is shown that one-time use randomizing is not only an effective security method but probably the best possible method used by some of the most secure companies in the world.

## Proof of Concept

There are a couple of details about the SEEM proof of concept program that are noteworthy. Firstly, the ID that identifies whether a packet is a fake packet or not is sent to a text file along with the encrypted text. This is to simulate the action of sending a packet to another location. None of the data from one encrypted line is used in encrypting the next line. This is in contrast to how the official program would operate. The proof of concept program is to simulate the real world application, and is just to show the capabilities. Truthfully, its simplicity is a benefit to moving forward because with a baseline product, the possibilities for improvement are much more clear.

Another noteworthy function of the program is that it uses a Vigenère cipher. Essentially, this cipher is a circular Caesar cipher. A Caesar cipher rotates each letter in the message a certain times forward in the alphabet. So, the Vigenère cipher is a substitution cipher that includes many rotations for each letter. The Vigenère cipher uses a “tabula recta”, or, a square table, to substitute the correct characters (Janssen). The other important part about this cipher is that the key is continually used and rotated until the message is complete. Since we are splitting up each message into blocks for our program, this cipher simply uses the amount of letter necessary from the key.

### Figure 4

|  |  |  |
| --- | --- | --- |
| Message: | love | turtles |
| Key: | (xfdp)rmbqf | (xfdprmb)qf |
| Output: | ityt | qzuicqt |

As seen in the figure, the key is 64 bits but it is consistent and simple. This is also not consistent to real world because if we were to use AES block cipher encryption, for example, then each key would be 128, 196 or 256 bits and this key would be extremely difficult to crack or guess (Rouse). Realistically, this if you are using AES as the bread and butter of the encryption on your systems, you can have a large amount of confidence in it. As a recap, key generation, number of times key is used, and the ability to keep the key a secret all substantially affects the security of your process.

As alluded to previously, the random ID is included on the same line as the encrypted text and I wanted to give an example from the program of what that might look like. Keep in mind that this file is accessed and sorted through to distinguish the real and fake packets and then decrypted.

### Figure 5



## Fake Packet Option

An option in creating the fake packets is to make them all the same. If we are using a chaining mode then we can put the same thing inside of a fake packet every time and we will still create entropy because they are being chained with the real packets. Using an encryption mode like ECB, an attacker would be able to tell that the encryption of one packet was “00000000” and the encryption of another packet was the same, so they would be able to use frequency analysis to determine that maybe all of the fake packet’s encrypted text will be the same. Therefore, when we use a chaining mode, we could simply put 16 bytes that include the exact same order of binary code, like all 1s, so that we know that when we decrypt the packet to find that, we know that it’s a fake packet. This could be extremely valuable because if we did this we would not need to include the function to determine whether if it’s a fake packet inside of the packet at all. Additionally, this would be beneficial because the less information we can potentially give to an attacker the better.

## Conclusion

Kerckhoffs’s principle was created in 1833 and revolved around cryptographic systems that are outdated today, but the implications of his articles lives on. We talked about how private randomizing can help boost security just like the secrecy of the private key can. We found that in terms of defense, which is most closely related to security, it can be a beneficial tactic to attack your enemy. If you can mislead your enemy into a predictable corner, you can attack your opponent even when you are the defender.

The tactic promoted incorporates fake packets into the normal encryption process. Encryption modes normally are straight forward but inserting fake packets can confuse the attacker. Of course, if the attacker is familiar with our method, then they will know that we use fake packets. In this fashion, it’s important to have secure and private techniques for randomizing IDs. In any case, the network cover, and one time use techniques help strengthen this algorithm more than normal encryption modes.

Furthermore, we stressed that this method prioritizes security over speed. For companies that want to communicate with each other, the confidentiality of the message is often much more important. However, we find that even if we can achieve security for a certain length of time, the message is no longer valid, even if it was of the highest level of value previously.

Overall, we simply tried to explore the possibilities of a creative way to defend systems because in the era in the second millennium, attackers are relentless in pursuit because of numbers and shear computing power. Because of this, we need to think of ways to increase security that defy the standard “do I have more computing power than you” situation. Nonetheless, we have successfully explored different and creative options for security that uses encryption even if we have not broken new ground.

The SEEM program name came from what “seem”s to be, whether it is an idea that security is impossible, or that all things in an algorithm are supposed to be known by the attacker, is an illusion. We now know that preventing beaches is impossible, but what matters is that in the end, we have beaten the attacker. Therefore, we focus on deceiving, and we strive to show one thing to an attacker, but mean another. In other words, what SEEMs to be, isn’t.

# Bibliography

Google. *google.com*. 21 October 2014. www.google.com/landing/2step/. 15 June 2015.

Haahr, Dr Mads. *RANDOM.ORG*. n.d. 15 June 2015.

Janssen, Cory. *techopedia*. n.d. 15 June 2015.

Kowalczyk, Chris. *CRYPTO-IT*. 2013. Web. 15 June 2015.

Massey, James L. *Cryptography: Fundamentals and Applications* (1993): 2.5. course notes.

Perrin, Chad. *TechRepublic*. 30 June 2008. 15 June 2015.

Rouse, Margaret. *SearchSecurity*. November 2014. 15 June 2015.

*The Alamo*. Dir. John Lee Hancock. Perf. Dennis (Sam Houston) Quaid. 2004.

Tzu, Sun. *The Art of War*. Mineola, New York: Dover Publications, Inc., 2002. Translation.