SAGE: Secure, Air-Gapped Encryption for the Raspberry Pi

Aaron Smith and Carl Block
Math 194, Professor Bruff
github.com/midnightdev/SAGE
github.com/midnightdev/encryption_algorithm

Abstract. Air-gapped encryption would be more accessible to the public if encryption software with a simple user interface could be run on inexpensive software such as the Raspberry Pi. Our solution is composed of two pieces: a custom encryption algorithm written in Python and a front-end web application built on the Pyramid web framework. Key generation and retrieval will be simplified to a single button press, and encryption or decryption will only require selecting a text file from the file system. The encryption algorithm uses a Hill Cipher padding scheme along with public key encryption.

1. Introduction

The National Security Agency leaks by former security contractor Edward Snowden revealed the great lengths the NSA and GCHQ, the British intelligence service, have taken since 9/11 to create an Internet dragnet. Metadata and private communications between citizens both inside and outside the United States have been collected by the NSA with oversight only from a secret court (FISA), a Senate committee, and the executive branch. These revelations, brought to the public's attention by Guardian journalists Glenn Greenwald and Laura Poitras, were divisive. While some citizens were seemingly unconcerned by the mass collection of communications, others have called for reform.

As Greenwald and Poitras became international celebrity journalists overnight, their security practices inspired others to learn to use encryption [1]. One important practice involves an air-gapped computer used solely for encryption. An air-gapped computer makes it more difficult for documents to be stolen remotely. For the general public, however, an air-gapped computer is an expensive investment. An inexpensive alternative is needed.

2. Hardware

The $25 Raspberry Pi is arguably the cheapest possible option for a fully functional air-gapped computer. While a keyboard, mouse, and monitor all need to be acquired to use the device, many homes already have them available. A USB drive must also be purchased so that encrypted files can be transferred to and from the Raspberry Pi.

3. Key Generation

Key generation on the front-end of the application is simple. There are buttons for generating new public or private keys and downloading those keys to the Raspberry Pi to be shared.

Creating a private key involves generating two random prime numbers, p and q, that are not equal. These two numbers are then used to generate a corresponding public key pair. The first number in the public key pair, n, is found by multiplying the p and q together. The second number in the public key pair, e, is another random prime number with the one constraint being that (p - 1) * (q - 1) % e does not equal 0. In summary:

Private Key

p = random prime number
q = random prime number (where p != q)

Public Key

*n = p * q*
*e = random prime number (where (p - 1) * (q - 1) % e != 0)*

4. Padding

The first step of encryption, padding, requires that one or two matrices are created, depending on the length of the message. Since a matrix of any size will not divide evenly into every possible message length, the algorithm must respond differently to messages of different lengths.

The default matrix length is 15 characters, but can be easily changed by passing a custom size into the program when initializing the EncryptMessage() object. The algorithm then compares the matrix length with the message length. If the message length is a multiple of the matrix length, then only one matrix is created. When the matrix cannot be divided evenly into the message, however, two matrices are created. The first matrix is the specified size (again defaulting to 15), while the second matrix is equal to the remainder when the first matrix is divided by the message length. When the remainder is one and the message is longer than one character, however, we use one fewer default matrix, and have a matrix of size one bigger than the default at the end (i.e. 16 when the default is 15).

A requirement of the Hill Cipher is that the inverse of the cipher be an integer matrix. This property means that every cipher created must be a unimodular matrix. A unimodular matrix has a determinant of 1 or -1 [2]. Making a unimodular matrix requires that the only row operations used are the addition of an integer multiple of a row to another, or the switching of rows. Our method only uses the addition of one row to another. Therefore, the following Python method is used:

```
matrix = numpy.eye(size)
for i in range(HILL_STRENGTH):
    a = random.randint(0, size - 1)
    b = random.randint(0, size - 1)
    while b == a:
        b = random.randint(0, size - 1)
    matrix[a] = matrix[a] + matrix[b]
```

The length of a Hill Cipher's alphabet should be a prime number because this increases the keyspace of the encryption, making it more secure[4]. When the padding process begins, each letter in the message string is converted to it's corresponding integer place in the alphabet. The matrix sizes are then determined and the message is iterated through and padded.

Kerckhoffs' principles tell us that the security of a cipher is entirely depended on its key and that cryptographers should assume that the encryption method is already known. The strength of the Hill Cipher, therefore, is directly influenced by the cipher size and the randomness of cipher entries. Given the requirement that any matrix used as a cipher must have a determinant of 1 or -1, the number of possible ciphers are significantly limited. In addition, this encryption algorithm only performs 50 row operations because performing more takes a significant amount of time. These two factors severely limit the strength of the padding scheme.

5. Public Key Encryption

After padding is complete, the sizes and entries of both matrices are appended to the front of the padded message array. An array with a 2x2 cipher and 4 character message would look like this:

```
matrix_size = 2

matrix = [ 3, 1
           2, 1 ]

[2, 0, 3, 1, 2, 1, 45, 84, 23, 54]
```

The public key pair is used to encrypt each entry of this array. Given *m*, an entry in the padded array above, and *(e, n)*, the public key pair, the following equation produces *c*, the ciphertext [5]:

$$c = m^e \pmod n$$

This process takes place for each number in the padded array except for the size value. Once every number has been encrypted by the public key, they are written to a text document. Each number in the document is separated by a period (.). The written text may look similar to this:

```
432.523.754.5643.43.256.362.764.8347.252
```

6. Decryption using the Private Key

When the message is received by the intended recipient, the public key encryption must be undone. To do this, the private key pair *(p, q)* is used in conjunction with *e* from the public key pair to find *d (Note: phi(n) = (p-1)(q-1))*:

Equation: (d * e) = 1 mod(phi(n))

```
while (a * phi_n + 1) % self.e != 0:
   a += 1
d = (a * phi_n + 1) / self.e
```

Once *d* is found, it and *n* can be used in conjunction with *c*, the encrypted part of the matrix or message, to undo the public key encryption. [5]

$m = c^d \pmod n$

7. Decrypting the Padding Scheme

Once the second layer of encryption has been removed, the padding scheme can be undone. The decryption method uses the sizes of the matrices and the length of the message to determine how many times to use each matrix during decryption. The matrices are inverted using the Python library numpy and rounding errors (an issue caused by the computer) are accounted for.

The algorithm then iterates through each section of the message, multiplying the correct matrix by that portion of the message until the whole message has been decrypted. The message is still a list of numbers, so the correct 89 character alphabet must be used to convert the message back to plain text.

8. Vulnerabilities

It is important to note that the encryption method described in this document should not be used in real-world applications. Setting aside the sheer simplicity of this algorithm compared to other methods, there are a few notable security gaps that are a result of conscious decisions made during development.

The first vulnerability is a result of including the Hill Ciphers used for padding the message with the message when it is sent. Ideally, these ciphers would be heavily encrypted and sent separately so they could be used for multiple messages. This would make the padding component of the algorithm a form of symmetric cryptography similar to those seen in commercial algorithms.

The second vulnerability is the relative simplicity of the cipher matrices. Because only 50 row operations are performed on the matrix due to limitations in computing power, many rows in the matrix are left as 1 or 0. This poses a problem when the matrix is encrypted by the public keys because those entries equal to 1 or 0 stay equal to 1 or 0. This makes the padding scheme much more susceptible to brute force attacks.

The third notable vulnerability is caused by the USB drive. The drive can be compromised when plugged into the computer that is not air-gapped. Viruses targeting air-gapped computers can execute when the USB is plugged into the air-gapped device and will store as much extra data onto the USB as possible. That data can then be transmitted remotely once the USB is plugged into an Internet-enabled computer [6].

To combat this form of theft, purchase a USB with a small amount of storage and/or one that lights up when files are being transferred. If the light is on when files are not being transferred by the user, the system may be compromised. Likewise, a USB with a small amount of storage space ensures that only data up to the amount of extra space on the USB can be stolen at one time [6].

9. References

[1] Lennard, Natasha, “Five tips to stay secure despite NSA encryption cracking,” http://www.salon.com/2013/09/05/five_tips_to_stay_secure_despite_nsa_encryption_cracking, September 2013.

[2] Weisstein, Eric W. "Unimodular Matrix." From *MathWorld*--A Wolfram Web Resource. http://mathworld.wolfram.com/UnimodularMatrix.html.

[3] Petitcolas, Fabien. “la cryptographie militaire.” http://petitcolas.net/fabien/kerckhoffs/.

[4] Overbey, Jeffrey. “On the Keyspace of the Hill Cipher.” http://jeff.over.bz/papers/undergrad/on-the-keyspace-of-the-hill-cipher.pdf.

[5] Ouwehand, Martin. “The (simple) mathematics of RSA.” http://certauth.epfl.ch/rsa/node3.html#SECTION00033000000000000000.

[6] Schneier, Bruce. “Want to Evade NSA Spying? Don’t Connect to the Internet.” http://www.wired.com/opinion/2013/10/149481/.