Cryptography

NOVEMBER 6, 2015 by ed harmoush leave a comment

Do you do online banking? Do you work from home? Do you use VPNs to access company resources? All these would not be possible without Cryptography.

Cryptography is the art of keeping secrets, specifically through any form of communication.

Cryptography has existed for thousands of years, but has become increasingly more important in recent history due to the explosion of the Internet and the need for data privacy and secure online communications.

But what does "secure communication" even mean? Typically, it refers to (at least) these four concepts:

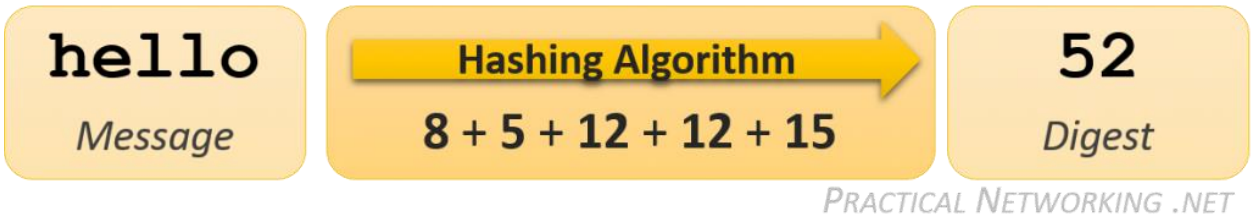
- **Confidentiality** – Assuring only the intended recipients in communication have access to the message.
- **Integrity** – Assuring that the message cannot be modified in transit without the other party being made aware.
- **Authentication** – Assuring the other party is indeed who they claim to be.
- **Anti-Replay** – Assuring the message cannot be maliciously re-transmitted.

Hashing Algorithm

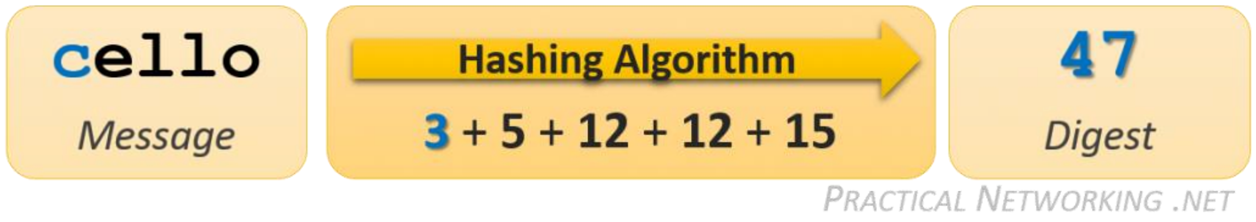
The first concept we need to discuss in our exploration of Cryptography is that of a **Hashing Algorithm**.

A Hashing Algorithm is a mathematical **formula that takes a Message of arbitrary length as input and produces as output a representational sample** of the original data.

For instance, a rudimentary example of a hashing algorithm is simply adding up all the letter values of a particular message. (A=1, B=2, C=3, etc…):



The result of a hashing algorithm is called a message **Digest** (or sometimes Checksum, or Fingerprint). The result of our example hashing on the original message of `hello` was `52`. If someone were to change our original message and process it through the same hashing algorithm, the result would be different:



By comparing the message digests of each calculation, it is easy to determine that our message has changed.

Obviously, the Hashing algorithm used in the example is full of flaws. There are many words that when processed through the example algorithm that might result in the same Digest. Had the original Message been changed to *`cellt`*, the resulting digest would still be 52, and we would be unaware that the original Message had been altered.

In reality, a legitimate hashing algorithm must maintain four qualities before it is approved for industry usage:

1. **It is mathematically impossible to extract the original message from the digest.**

It should be impossible to reverse the hashing algorithm and recover the original Message knowing just the resulting Digest. In fact, Hashing is sometimes referred to as *one-way encryption*: the message can be encrypted but is impossible to decrypt. This is accomplished using one-way functions within the hashing algorithm.

In a way, our example Hashing algorithm satisfied this condition. It is impossible to derive *hello* knowing only a resulting digest of *52*. Mostly because there could be thousands of messages that result in the identical digest.

2. **A slight change to the original message causes a drastic change in the resulting digest.**

Any minor modification – even as small as changing a single character – to the original Message should greatly alter the computed digest. This is sometimes referred to as the Avalanche effect.

It is possible because a Hashing algorithm is not simply one calculation. It is a series of calculations, done iteratively over and over. As a result, a small change in the beginning, creates an exponentially bigger and bigger change in the resulting digest. Just like a snowball tumbling down a mountain forming an avalanche.

3. **The result of the hashing algorithm is always the same length.**

It is vital for the resulting Digest to not provide any hints or clues about the original Message – including its length. A digest should not grow in size as the length of the Message increases.

In our example Hashing algorithm, the longer the word, the bigger the resulting digest would be as we are adding more and more letters together. However in an industry approved hashing algorithm, hashing the word *hello* would produce a digest the same size as hashing the entire library of congress.

4. **It is infeasible to construct a message which generates a given digest.**

With our example hashing algorithm, if given the digest of *52* , it would not be overly difficult to generate a list of words that might have been the original message. This is what this attribute is trying to prevent.

In a proper hashing algorithm, this should be infeasible — short of attempting every possible combination of messages until you found a match (aka, brute-forcing the algorithm). But even this becomes infeasible given a large enough digest size.

In the next article in this series, we will look at exactly *how* Hashing Algorithms are used to detect modified messages. But for now, we will continue to look at additional aspects of Hashing Algorithms.

Digest Lengths

Below is a table with commonly seen, industry recognized hashing algorithms:

Algorithm	Digest Length
MD5	128 Bits
SHA or SHA1	160 Bits
SHA256	256 Bits
SHA384	384 Bits

Each of these Hashing algorithms satisfy the four cryptography hashing algorithm properties, as described above. The primary difference between each of them is the size of the resulting digest.

As with passwords, it is typically considered that a hashing algorithm which results in a longer digest tends to be regarded as more secure.

Hashing Demonstration with Linux

To take it a step further, I would like to demonstrate how you can use a hashing algorithm from a standard Linux terminal. Note that if you are unfamiliar with Linux, feel free to skip this section — it is not crucial to learning the aforementioned concepts. If you have at least some Linux familiarity, however, it might help to see these algorithms in action.

The standard Linux terminal typically comes with at least two of the Hashing Algorithms mentioned above: MD5 and SHA1. You can use the **`echo`** command along with **`md5sum`** or **`sha1sum`** to run either algorithm on a string of text:

```
$ echo "Practical Networking .net" | md5sum
018aa3ff55842e546c661b7027aed5d7 -
```

If you typed the exact same command into any Linux terminal, the resulting digest would be the exact same. In fact, if you fed the string **`Practical Networking .net`** into any MD5 algorithm, you would see the exact same digest *(remember, the **`echo`** command also appends a new line character to the string)*. If we were to change something small, the resulting digest should be completely different. For example, we can capitalize the ***n*** in *`.net`* and take a look at how the digest changes:

```
$ echo "Practical Networking .Net" | md5sum
6b9298494fb90a1a57efddaee60fbfc1 -
```

We could also run a much larger sample of text through the MD5 algorithm. We can echo the same string 10,000 times and calculate the **`md5sum`**, and notice the resulting digest is still the same length (but the digest is different, of course):

```
$ for i in {1..10000}; do echo "Practical Networking.net"; done | md5sum
938a98abd28c5da2dee6aa07bbc25134 -
```

Try these same examples using **`sha1sum`**. If you don't have easy access to a Linux shell, you can use a free online Linux terminal.

Message Integrity

In the world of secured communications, Message **Integrity** describes the concept of ensuring that **data has not been modified in transit**. This is typically accomplished with the use of a Hashing algorithm. We learned earlier what a Hashing Algorithm does. Now we can take a look at how they are actually used to provide Message Integrity.

The basic premise is a sender wishes to send a message to a receiver, and wishes for the integrity of their message to be guaranteed. The sender will calculate a hash on the message, and include the digest with the message.

On the other side, the receiver will independently calculate the hash on *just* the message, and compare the resulting digest with the digest which was sent with the message. If they are the same, then the message must have been the same as when it was originally sent.

PRACTICAL NETWORKING .NET
Sender
Receiver

Pretty straight forward. Except for one major problem. Can you guess what it is?

If someone intercepted the message, changed it, and recalculated the digest before sending it along its way, the receiver's hash calculation would also match the modified message. Preventing the receiver from knowing the message was modified in transit!

So how is this issue averted? By adding a Secret Key known only by the Sender and Receiver to the message before calculating the digest. In this context, the Secret Key can be any series of characters or numbers which are only known by the two parties in the conversation.

Before sending the message, the Sender combines the Message with a Secret key, and calculates the hash. The resulting digest and the message are then sent across the wire (*without the Secret!*).

The Receiver, also having the same Secret Key, receives the message, adds the Secret Key, and then re-calculates the hash. If the resulting digest matches the one sent with the message, then the Receiver knows two things:

1. The message was definitely not altered in transit.
2. The message was definitely sent by someone who had the Secret Key — ideally only the intended sender.

This animation reflects this process:

When using a Secret Key in conjunction with a message to attain Message Integrity, the resulting digest is known as the **Message Authentication Code**, or **MAC**. There are many different methods for creating a MAC, each combining the secret key with the message in different ways. The most prevalent MAC in use today, and the one worth calling out specifically, is known as an **HMAC**, or **Hash-based Message Authentication Code**.

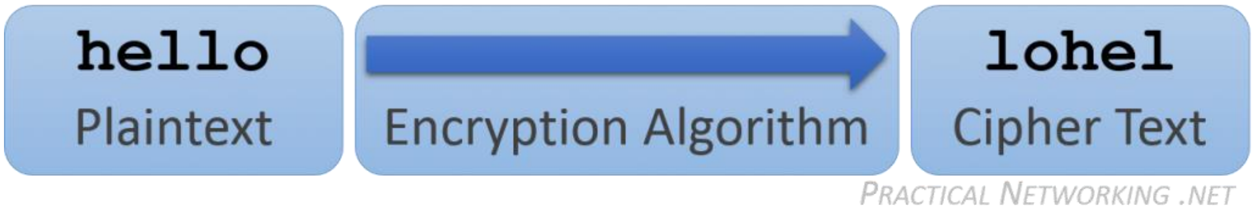
Of course, this doesn't answer the question of "How did the Sender and Receiver establish mutual secret keys?" This is known as the Key Exchange problem, which comes up a few times in cryptography. However, the answer lies outside the scope of the concept of Integrity, and will be discussed in another article in this series.

Confidentiality

Confidentiality is the concept of hiding or scrambling your data so that only the intended recipient has access. This is typically accomplished by some means of **Encryption**.

Data before it has been encrypted is referred to as **Plain text**, or **Clear text**. After the data has been encrypted, it is referred to as **Cipher text**. The Cipher text should be completely unrecognizable, revealing no patterns or hints as to what the original Plain text was. Only the intended receiver(s) should have the ability to **Decrypt** the Cipher text and extract the original Plain text.

The process by which the the Plain text is converted to Cipher text is known as the **Encryption Algorithm**.



In this basic encryption example, all that is needed to reverse the encryption and decrypt the Cipher text is insight into what happened in the Encryption Algorithm. You've probably picked up by now, that to take `hello` and scramble it to `lohel`, all I did was shift the letters forward twice. To undo this, you just need to shift the letters back twice.

There are, however, a few issues with this type of basic encryption:

- **It does not scale**. For each new person you wish to securely exchange data with, you would need to devise a new encryption algorithm. You wouldn't want the communication you had between you and your bank to be secured the same way as it was between you and your employer. How many different algorithms could you come up with before you were forced to reuse them?
- **Once the algorithm is discovered, the security is comprised for *all* time.** Everything that was secured with the compromised algorithm in the past is now fully decryptable. And everything that you might ever continue to secure with that algorithm in the future is now fully decryptable.

- **In the end, all you've done is obfuscate the data.** It may be enough to prevent a passerby from accidentally reading your Clear text, but it won't be enough to thwart a truly determined hacker.

As a result of these weaknesses, modern confidentiality makes use of what is sometimes referred to as **Cryptographic Encryption**. Which is **combining a publicly known encryption algorithm along with a secret key**.

The math behind the algorithm is publicly disclosed, which gives it the benefit of having been vetted by many mathematicians and cryptographers before any particular algorithm is accepted for common use.

The secret key can be a randomly generated set of characters — which makes it easy to produce. It is not difficult to use a different key for each entity you wish to speak securely with, even if the algorithm for each of these parties is the exact same. It is also not difficult to periodically regenerate the secret key, so even if a particular key becomes compromised, only a subset of your communication can be decoded.

There are two types of Cryptographic Encryption: **Symmetric Encryption** and **Asymmetric Encryption**. The main difference between the two types of encryption can be summarized as follows:

- Symmetric encryption – Encrypt and Decrypt using the *same* key.
- Asymmetric encryption – Encrypt and Decrypt using two *different* keys.

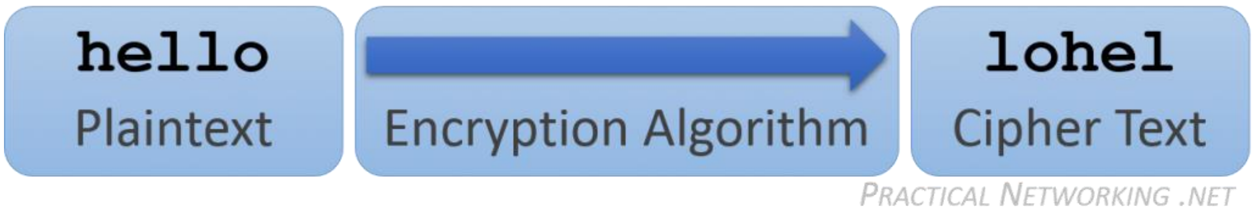
We will look at both of these and how they are used to provide Confidentiality in more detail in next articles in this series.

Confidentiality

Confidentiality is the concept of hiding or scrambling your data so that only the intended recipient has access. This is typically accomplished by some means of **Encryption**.

Data before it has been encrypted is referred to as **Plain text**, or **Clear text**. After the data has been encrypted, it is referred to as **Cipher text**. The Cipher text should be completely unrecognizable, revealing no patterns or hints as to what the original Plain text was. Only the intended receiver(s) should have the ability to **Decrypt** the Cipher text and extract the original Plain text.

The process by which the the Plain text is converted to Cipher text is known as the **Encryption Algorithm**.



In this basic encryption example, all that is needed to reverse the encryption and decrypt the Cipher text is insight into what happened in the Encryption Algorithm. You've probably picked up by now, that to take `hello` and scramble it to `lohel`, all I did was shift the letters forward twice. To undo this, you just need to shift the letters back twice.

There are, however, a few issues with this type of basic encryption:

- **It does not scale**. For each new person you wish to securely exchange data with, you would need to devise a new encryption algorithm. You wouldn't want the communication you had between you and your bank to be secured the same way as it was between you and your employer. How many different algorithms could you come up with before you were forced to reuse them?
- **Once the algorithm is discovered, the security is comprised for *all* time.** Everything that was secured with the compromised algorithm in the past is now fully decryptable. And everything that you might ever continue to secure with that algorithm in the future is now fully decryptable.

- **In the end, all you’ve done is obfuscate the data.** It may be enough to prevent a passerby from accidentally reading your Clear text, but it won’t be enough to thwart a truly determined hacker.

As a result of these weaknesses, modern confidentiality makes use of what is sometimes referred to as **Cryptographic Encryption**. Which is **combining a publicly known encryption algorithm along with a secret key**.

The math behind the algorithm is publicly disclosed, which gives it the benefit of having been vetted by many mathematicians and cryptographers before any particular algorithm is accepted for common use.

The secret key can be a randomly generated set of characters — which makes it easy to produce. It is not difficult to use a different key for each entity you wish to speak securely with, even if the algorithm for each of these parties is the exact same. It is also not difficult to periodically regenerate the secret key, so even if a particular key becomes compromised, only a subset of your communication can be decoded.

There are two types of Cryptographic Encryption: **Symmetric Encryption** and **Asymmetric Encryption**. The main difference between the two types of encryption can be summarized as follows:

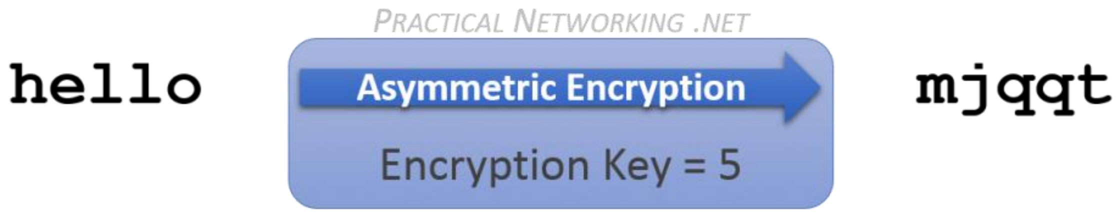
- Symmetric encryption – Encrypt and Decrypt using the *same* key.
- Asymmetric encryption – Encrypt and Decrypt using two *different* keys.

We will look at both of these and how they are used to provide Confidentiality in more detail in next articles in this series.

Asymmetric Encryption

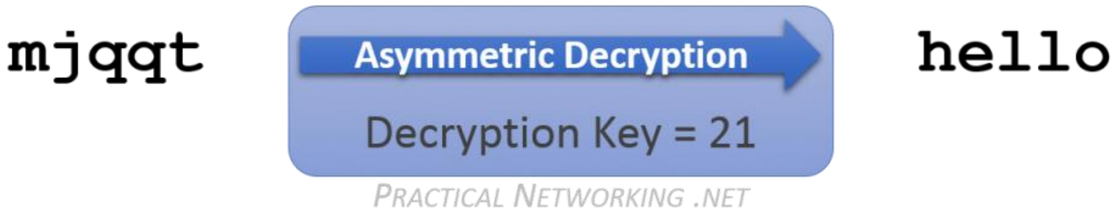
Earlier, we learned that Symmetric encryption is an encryption scheme that uses the *same* key to encrypt and decrypt. Conversely, **Asymmetric encryption**, uses *different* keys to encrypt and decrypt. Lets take a look at a simple example.

For the sake of simplicity, let us pretend for this example that there are only the lower case letters `a - z` available. No capitals, no numbers, no symbols. This would give us a total of 26 possible characters.



In the example above, we are taking the plain text of **`hello`**, and encrypting it with an Asymmetric encryption key of `5`. This results in our cipher text, **`mjqqt`**.

Instead of simply reversing the encryption, as you would for a Symmetric encryption, let us instead continue rotating the letters forward `21` more times:



Notice if we continue to move forward, with a Decryption key of `21`, we end up back where we started at the original plain text of **`hello`**.

Now obviously, in this simplistic example, we could have simply rotated backwards with the Encryption key of 5. But in a real Asymmetric encryption algorithm, attempting to re-use the Encryption key (either forwards or backwards) would simply scramble the text further.

That said, there is something significant worth pointing out that we can learn from the Asymmetric encryption example above.

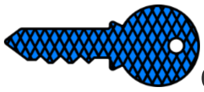
We used an Encryption key of 5, and were able to decrypt successfully with a Decryption key of 21. *BUT*, we could also have used an Encryption key of 21, and a successfully decrypted with a Decryption key of 5. The **Asymmetric keys are mathematically linked.** What one key encrypts, only the other can decrypt — and *vice versa*.

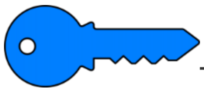
Can you determine another set of keys that would work as an Asymmetric key pair for the simple example above?

A Tale of Two Keys

So what can we do with an Asymmetric Key Pair?

We discussed earlier the existence of the two keys — that they are mathematically linked, and that whatever data is encrypted by one of the keys can only be decrypted by the other key.


One of these keys is stored securely, and *never* shared with anyone else. This key is from then on referred to as the **Private Key**. In the rest of this series, this key will always be checkered, and point to the left.


The other key is made available to the world. This key then becomes your **Public Key**. In the rest of the series, this key will always be solid colored, and pointing to the right. The “key pair” will be identified by the matching color.

Every participant in Asymmetric encryption has their own, unique key pair. Each of these keys can be used in different ways in order to attain different security features. These will be outlined in a dedicated article in this series.

Comparison with Symmetric Encryption

Understandably, the math involved in Asymmetric encryption is slightly more complex than what might be required with Symmetric encryption. As a result, the CPU cost of Asymmetric encryption tends to be higher than its symmetric counterpart.

Moreover, a side effect of the math is also that the resulting cipher text often ends up being larger than the original plain text. If you only intend to Asymmetrically encrypt a small amount of text, then this is negligible. But if you are looking to encrypt bulk data transfer, this makes Asymmetric encryption not ideal.

That said, the primary (and most significant) benefit to using Asymmetric encryption is **the Private Key *never* needs to be shared**. As opposed to Symmetric encryption, where the *same* Secret Key must exist on both sides of the conversation.

As a result, Asymmetric encryption is regarded as more secure than its symmetric counterpart. There is no risk of compromise while the key is being transferred (since it never needs to be transferred at all). There is no risk of compromise from the other party's potential lack of security (since the other party never has your Private key).

Using Asymmetric Keys

We’ve established how Asymmetric encryption makes use of two mathematically linked keys: One referred to as the Public Key, and the other referred to as the Private Key. We’ve also established that what one key encrypts, only the other can decrypt.

These two attributes allow us to perform two separate operations with a Key Pair.

Asymmetric Encryption

Below is an illustration of Bob (on the right in red) looking to send an encrypted message to Alice (on the left in purple).

Since Bob and Alice are two different entities, they each have their own set of Public and Private Keys. Their public keys are on the inside, available to each other. While their private keys are on the outside, hidden and out of reach.

PRACTICAL NETWORKING .NET

When Bob has a message he wishes to securely send to Alice, he will use **Alice’s Public Key to Encrypt** the message. Bob will then send the encrypted message to Alice. **Alice will then use her Private Key to Decrypt** the message and extract the original message.

Since Bob encrypted the message with Alice’s **Public** key, he knows that the only possible key that could extract the message is Alice’s **Private** key. And since Alice never shared her key with anyone, Bob knows that only Alice was able to read the message.

Thus, the concept of confidentiality can be provided with an Asymmetric key pair.

Asymmetric Message Signing

But confidentiality isn't the only thing you can do with a Public and Private Key. Remember, *either* key can be used for encryption. This fact can be used to give us one additional feature from an asymmetric key pair.

Let us imagine that now Alice wants to send a message to Bob. This time, however, Alice does not care about the confidentiality of her message. Which is to say, she doesn't care if anyone can read it. But she is very concerned that Bob knows beyond a shadow of a doubt that it was definitely Alice that sent the message.

Alice can use **her own Private Key to encrypt the message**. Which makes it so the only key in the world that can decrypt her message is her Public key — *which she knows Bob (and anyone else) has access to*.

The message is sent to Bob, who then uses **Alice's Public Key to decrypt the message**. If Bob was able to successfully extract a message, and not a scrambled series of bits, then he can be assured that the message must have been originally encrypted by Alice's Private Key. And since Alice never shared her Private Key with anyone, Bob can be assured that Alice indeed sent the message.

This process is known as **Message Signing**. It is a creative use of the fact that the keys are mathematically linked, and that what one key encrypts, only the other can decrypt.

Real World Usage

Now that we have illustrated the basic premise. We can take it a step further and really look at how these concepts are actually used in modern cryptography.

Real World Encryption

Earlier, we discussed that Asymmetric encryption is slower and has properties which make it not ideal for bulk encryption. We should instead find a way to use Symmetric encryption, since it is better suited for bulk data encryption. But with Symmetric encryption, we have to deal with the Key Exchange issue.

The solution is to use what is sometimes referred to as **Hybrid encryption**, which **combines the strengths of both Symmetric and Asymmetric encryption**, while avoiding all their weaknesses.

Let's describe how that works by continuing to use Alice and Bob from above as an example.

Bob starts by randomly generating a Symmetric Secret Key. Then, instead of Bob using Alice's public key to encrypt the *message* directly, Bob uses Alice's Public Key to encrypt the Symmetric Secret Key. This encrypted symmetric key is sent across the wire to Alice.

Alice can then use her Private Key to extract the Secret Key that Bob sent. At this point, both parties now have an identical Secret Key that they can use to *Symmetrically* encrypt as much data as they please, in both directions.

PRACTICAL NETWORKING .NET

In this way, Bob and Alice use Asymmetric Keys to securely exchange a Symmetric Key, which is then used for Symmetric encryption. They are getting the security of Asymmetric encryption, with the speed and efficiency of Symmetric encryption — the best of both worlds.

Real World Signatures

Similarly, the Message Signing process is more than simply using the Private Key to encrypt the message. Again, the limitations of Asymmetric encryption would end up imposing a limitation on what sort of data can be signed.

Can you guess what method is employed to reduce the message of variable length to a constant, more manageable representational value?

You guessed it… a Hashing algorithm. Lets talk through it using Bob and Alice.

Alice wants to sign a message to Bob. She runs her message through a Hashing Algorithm, and then encrypts the resulting digest with her own Private Key. The encrypted digest then gets sent to Bob, along with the original message.

Bob then uses Alice's public key to decrypt the digest he received, then he independently calculates the hash of the original message. Bob then compares the (now decrypted) digest which was sent, and the digest which he calculated.

If they are the same, then Bob knows that Alice indeed must have sent the original message.

Moreover, Bob also knows that the message has not changed since Alice calculated the original digest — the signature had the bonus effect of also ensuring the Integrity of the original message!

Math is Hard

Most people can wrap their mind around Symmetric encryption fairly easily. Take a starting value, perform some mathematical operation, and you end up with cipher text. To convert it back, you simply perform the operation in reverse.

But Asymmetric encryption is slightly more complicated. Without prior exposure to Asymmetric encryption, its difficult to imagine a mathematical operation that you can perform on a starting value that is *impossible* to reverse. Even if you know the Public Key and the Algorithm used.

To that end, we've added an article as an appendix to the Cryptography series which explores the math behind a widely used Asymmetric algorithm in use today.

If math causes your eyes to glaze over, feel free to skip it, so long as you understand the basic concepts described throughout this series. But if you are slightly curious about how an Asymmetric algorithm works, head on over to the post on the RSA algorithm.

Authentication

In Cryptography, the concept of **Authentication** serves to **provide proof that the other side of a communication is indeed who they claim to be**, and who you intend for them to be.

There are multiple ways to verify the opposing party's Authentication. We will look at three of the most common:

- Username and Password
- Pre-Shared-Key
- Digital Certificates

Username and Password

Using a username and a password to identify who you are to a server is extremely common. Each user on a website can create a unique username, as well as a password tied specifically to that user. If the user is able to reproduce the password, then we can be assured that they are indeed the user they claim to be.

This is how most bank websites and email clients identify who you are.

The password itself should not be sent across the wire. That would easily lead to potential compromise. Instead, the password is run through some sort of hashing algorithm, and the resulting digest is then sent across the wire.

On the receiving end, it would be poor security practice to store the user's passwords directly. Instead, all that is stored is a hash of the password. Then, the digest sent with the user can be compared to the digest in the server's password database to see if they match. If they match, then the user must have had the expected password.

This is why most online websites that use a username and password are unable to recover lost passwords — all they can do is reset them. The password itself is never stored, only the digest of the password — which, as you recall, is impossible to reverse engineer and 'decrypt'.

Within this Authentication scheme, there are three different types of passwords that can exist:

- Something you *know*
- Something you *have*
- Something you *are*

Something you *know*

This is is the common password. You memorize a series of letters, special characters, words, and/or numbers, and you prove you *know* them when asked for the password.

This scheme is susceptible to users using weak passwords, storing them insecurely, or reusing them for different websites.

Something you *have*

This requires you to reproduce a physical object that only you can have in order to validate you are who you say you are. An ATM card or a employee badge are examples.

Today, many websites will send a random code to your phone via SMS when you are trying to log in, forcing you to have possession of your phone to log in. In such a case, even though you are inputting the random code to prove you are who you say you are, the code's purpose is simply to validate that you *have* your phone.

This is also the same concept behind the various authentication tokens in use today. You carry it around, and when you need to identify yourself, you input the code on the token (which changes every 30-60 seconds). If you can put in the code the server is expecting, then you must have *had* the token.

Often, the code you input is further prefixed or suffixed with a password known only to you. This would then create a system that validates who you are with both something you *know* (the password), and something you *have* (the token).

Something you *are*

Lastly, the various types of bio-metric identification fall under the category of something you *are*. Systems like fingerprint scanners or retina scanners or hand-print scanners all identify you based upon an attribute that is physically tied to who you *are*. Only you can have your hand. Only you can have your eyes. And so on.

Facial recognition, or even voice recognition, also falls under this category.

A password can be eavesdropped or shoulder-surfed. A token or mobile phone can be stolen. But bio-metrics can not be compromised without seriously maiming or killing the user being impersonated.

Two-factor Authentication

Many websites or security services offer or require what is known as **Two-factor Authentication.** This means the user is being identified from a password scheme from *two different* categories above.

For instance, the example above of a random code sent to your phone via SMS and your regular username and password is a perfect example of Two-factor authentication. To successfully authenticate against such a system you would need to both *know* your password, as well as *have* your mobile phone.

If a website simply required you to enter two passwords, or a password and a pin, this would *not* qualify as two-factor authentication because both methods of authentication fall under *something you know*.

Pre-Shared-Keys

The concept of Pre-Shared-Key authentication is to share a secret key or passphrase between two communicating nodes, then see if the two nodes can show proof of having said key.

This Pre-Shared-Key (PSK) should be shared out-of-band. For example, if you mean to use PSKs to prove someone's identity on the other side of the Internet, you should not use the Internet to share the key. You might use the phone, or a fax machine, or carrier pigeons.

Later on, if either node can show they have the correct PSK, it proves that the party on the other side of the wire is indeed the same entity which you initially exchanged the PSK with.

Obviously, simply sending the PSK across the wire to prove you have it would be a huge security flaw. We also can't simply run the PSK through a hashing function and send the resulting digest, because an eavesdropper could then capture the digest and spoof our identity in the future by reusing the same digest.

Instead, you would want to combine the PSK with values that are tied to that particular authentication session, so that the hash of the PSK is only good for *that one* session.

For example, in the case of IPsec, both parties generate and publicly exchange a random number. The PSK is then hashed together with both random numbers, and the resulting digest is shared. If both parties can generate the same digest, then they must have had the correct PSK. Furthermore, if an eavesdropper captures the verification digest and tries to reuse it to spoof the identify of one of the parties in a future session, they will be unable to because the future session will have different random numbers.

There are two primary differences between Pre-Shared-Key authentication and Username and Password authentication. The first: the PSK must be initially shared out-of-band. It can not be initially established using the medium upon which you want a secure connection. The second: the PSK is shared among two individuals, whereas a username and password is always unique to each user (or ought to be, at least).

Digital Certificates

The final authentication scheme we are going to discuss is that of Digital Certificates. This is the primary method of identification in use on the Internet. The protocols securing your browsing session when visiting a webpage of HTTPS make heavy use of Digital Certificates (SSL/TLS).

A digital certificate works similar to a driver's license. It contains the identity of a particular individual or website, and it is issued by a governing entity (the state you live in).

Inside a digital certificate is a very important piece of information: a Public Key of an Asymmetric Key Pair. This key is used to verify that the entity who presents the certificate is the true owner of the certificate. Much like your picture or signature on your driver's license.

Recall that with an Asymmetric Key pair, one key is kept private and never shared with anyone, and the other key is made public. Anyone can get a hold of anyone else's digital certificate (and therefore, public key), but theoretically, only one person can exist with the correlating private key. Before accepting a digital certificate as proof of someone’s identity, that someone must provide evidence that they are in possession of the matching private key.

There are two ways this can be verified. We’ll take a look at an example of Alice presenting a Digital Certificate to Bob, and how she can provide evidence that she is in possession of the private key.

1. If Alice presents Bob with her Certificate, Bob can generate a random value and encrypt it with Alice’s Public Key. Alice should be the only person with the correlating Private Key, and therefore, Alice should be the only person that can extract the random value. If she can then prove to Bob that she extracted the correct value, then Bob can be assured that Alice is indeed the true owner of the certificate.
2. Alice can encrypt a value known to both parties with her Private Key, and send the resulting Cipher Text to Bob. If Bob can decrypt it with Alice’s Public Key, it proves Alice must have had the correlating Private Key.

These two methods are the basis for how authentication works with digital signatures.

Anti-Replay

The Problem

Before we can describe the solution, we must first adequately describe the problem Anti-Replay is trying to solve.

Imagine your local bank branch office. Imagine someone going to that branch location, and depositing $100 in cash into their account. At some point following the transaction, that branch location will send some packets to the bank headquarters which essentially state to increase their account balance by $100.

Those packets will cross the branch office's Internet Service Provider (ISP) network on their route to the Bank Headquarters. If a malicious user (or even the depositor themselves) happens to work at the ISP, it would not be overly difficult to capture the packets going from the Branch to the Headquarters that day, and by time reference, narrow down the particular packet(s) that make up the deposit's transaction.

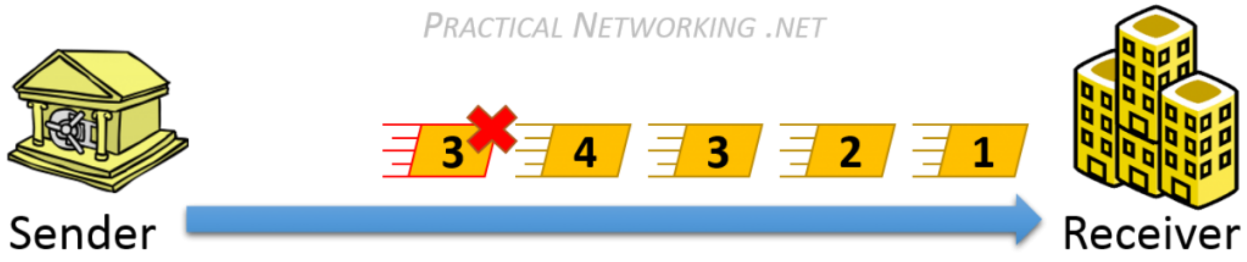
If the packets were protected by Confidentiality, the malicious user would be prevented from reading into the data packet itself. If the packets were protected by an Integrity scheme, the malicious user would be unable to increase the deposit amount to $1000 or $10,000. But, despite those two protections, there would be nothing that would prevent the malicious user from simply duplicating those packets and putting them back on the wire in order to continually re-deposit $100 over and over again.

…nothing if not for Anti-Replay protection, that is.

The Solution

Anti-Replay protection exists specifically to thwart the scenario described above. Anti-Replay injects what is known as a **Sequence Number** into the data packet. This number typically starts at 1, and increases with every packet sent, uniquely identifying one packet from the one prior.

On the receiving end, the last Sequence Number received is tracked. If a packet with a repeated sequence number is ever received, it can be discarded and presumed to be a Replayed packet. Take this example:



Notice, the packet with Sequence Number #3 is a replayed packet. The receiver can easily detect this because they have already received a packet with Sequence Number #3, and was expecting #5 next.

The sequence number, along with the data, will be protected by some sort of Message Integrity scheme to prevent a malicious user from tampering with the numbers in order to send a replayed packet. The sequence number should be included in the Hashing algorithm, along with the Message and the Secret Key. In this way, any illicit modification of any of these fields will be detected.

Sequence Number and Finite Fields

It is important to keep in mind that **the Sequence Number is a finite field**. Which is to say, it does not go on forever.

It has a pre-defined ranged based upon the number of bits allocated in the Data Packet. For example, if the data packet only allows a 16 bit field for the Sequence Number, you would have a total range of 1 – 65536.

It is important to consider this maximum so that it does not impose a limit on the number of packets that can be sent. Or worse, create a looped sequence number vulnerability when the sequence number rolls back to zero.

For example, using the banking transaction above, let us assume the packets only used a 16 bit Sequence Number field, and the captured packets included Sequence Numbers 10,000-10,099. The malicious user could simply wait for the Sequence number to loop past 65,536 and restart at 0, then count out 9,999 packets, and inject the replayed 10,000-10,099. Since the replayed packets would have arrived at the right time, they would have been accepted by the receiver.

This is addressed by forcing a rotation of the Secret Keys used in the Encryption and Integrity algorithms when the Sequence Number resets back to zero. This aids to ensure the continued, indefinite avoidance of replayed packets.

If an attacker attempts to replay a packet before the Sequence Number resets, then the repeated Sequence Number itself would identify the replayed packet. If an attacker attempts to replay a packet after the Sequence Number resets, then the *old packet* secured with the *old keys* will identify the packet was replayed.

Rivest Shaman Adleman

The RSA algorithm is the most widely used Asymmetric Encryption algorithm deployed to date.

The acronym is derived from the last names of the three mathematicians who created it in 1977: Ron **R**ivest, Adi **S**hamir, Leonard **A**dleman.

In order to understand the algorithm, there are a few terms we have to define:

- **Prime** – A number is said to be Prime if it is only divisible by 1 and itself. Such as: 2, 3, 5, 7, 11, 13, etc.
- **Factor** – A factor is a number you can multiple to get another number. For example, the factors of 12 are 1, 2, 3, 4, 6, and 12.
- **Semi-Prime** – A number is Semi Prime if its only factors are prime (excluding 1 and itself). For example:
  12 is *not* semi-prime — one of its factors is 6, which is not prime.
  21 *is* semi-prime — the factors of 21 are 1, **3**, **7**, 21. If we exclude 1 and 21, we are left with 3 and 7, both of which are Prime.
  (Hint: Anytime you multiply two Prime numbers, the result is always Semi Prime)
- **Modulos** – This is a fancy way of simply asking for a remainder. If presented with the problem `12 MOD 5`, we simply are asking for the remainder when dividing 12 by 5, which results in 2.

With that out of the way, we can get into the algorithm itself.

RSA Key Generation

The heart of Asymmetric Encryption lies in finding two mathematically linked values which can serve as our Public and Private keys. As such, the bulk of the work lies in the generation of such keys.

To acquire such keys, there are five steps:

1. **Select two Prime Numbers: P and Q**

This really is as easy as it sounds. Select two prime numbers to begin the key generation. For the purpose of our example, we will use the numbers **`7`** and **`19`**, and we will refer to them as **`P`** and **`Q`**.

2. **Calculate the Product: (P*Q)**
We then simply multiply our two prime numbers together to calculate the product:

```
7 * 19 = 133
```

We will refer to this number as **N**. Bonus question: given the terminology we reviewed above, what kind of number is N?

3. **Calculate the Totient of N: (P-1)*(Q-1)**
There is a lot of clever math that goes into both defining and acquiring a Totient. Most of which will be beyond the intended scope of this article. So for now, we will simply accept that the formula to attain the Totient on a Semi Prime number is to calculate the product of one subtracted from each of its two prime factors. Or more simply stated, to calculate the Totient of a Semi-Prime number, calculate P-1 times Q-1.

Applied to our example, we would calculate:

```
(7-1)*(19-1) = 6 * 18 = 108
```

We will refer to this as **T** moving forward.

4. **Select a Public Key**
The Public Key is a value which must match three requirements:

- It must be Prime
- It must be less than the Totient
- It must NOT be a factor of the Totient

Let us see if we can get by with the number 3: 3 is indeed Prime, 3 is indeed less than 108, but regrettably 3 is a factor of 108, so we can not use it. Can you find another number that would work? Here is a hint, there are multiple values that would satisfy all three requirements.

For the sake of our example, we will select **29** as our **Public Key**, and we will refer to it as **E** going forward.

5. **Select a Private Key**
Finally, with what we have calculated so far, we can select our Private Key (which we will call **D**).The Private Key only has to match one requirement: The Product of the Public Key and the Private Key when divided by the Totient, must result in a remainder of 1. Or, to put it simply, the following formula must be true:

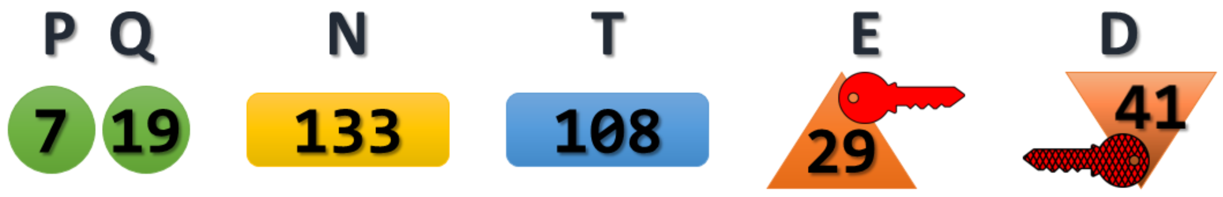
```
(D*E) MOD T = 1
```

There are a few values that would work for the Private Key as well. But again, for the sake of our example, we will select **41**. To test it against our formula, we could calculate:

```
(41*29) MOD 108
```

We can use a calculator to validate the result is indeed 1. Which means **41** will work as our Private Key.

And there you have it, we walked through each of these five steps and ended up with the following values:



Now we simply pick a value to be used as our Plaintext message, and we can see if Asymmetric encryption really works the way they say it does.

For our example, we will go ahead and use **99** as our Plaintext message.

(the math gets pretty large at this point, if you are attempting to follow along, I suggest to use the Linux Bash Calculator utility)

Message Encryption

Using the keys we generated in the example above, we run through the Encryption process. Recall, that with Asymmetric Encryption, we are encrypting with the Public Key, and decrypting with the Private Key.

The formula to Encrypt with RSA keys is: `Cipher Text =` **`M^E MOD N`**

If we plug that into a calculator, we get:

`99^29 MOD 133 =` **`92`**

The result of **92** is our Cipher Text. This is the value that would get sent across the wire, which only the owner of the correlating Private Key would be able to decrypt and extract the original message. Our key pair was 29 (public) and 41 (private). So lets see if we really can extract the original message, using our Private Key:

The formula to Decrypt with RSA keys is: `Original Message = M^D MOD N`

If we plug that into a calculator, we get:

```
92^41 MOD 133 = 99
```

As an experiment, go ahead and try plugging in the Public Key (29) into the Decryption formula and see if that gets you anything useful. You'll notice that, as was stated before, it is impossible to use the same key to both encrypt and decrypt.

Message Signing

But remember, that isn't all we can do with a key pair. We can also use same key pair in the opposite order in order to verify a message's signature.

To do this, we will use the same formulas as above, except this time we will revere the use of the Public and Private Key. We're going to encrypt with the Private Key and see if we can decrypt with the Public Key.

We'll use the same formula to encrypt, except this time we will use the Private Key:

```
Signature = M^D MOD N
```

If we plug that into a calculator, we get:

```
99^41 MOD 133 = 36
```

The result of **36** is the Signature of the message. If we can use the correlating public key to decrypt this and extract the original message, then we know that only whoever had the original Private Key could have generated a signature of 36.

Again, the same Decryption formula, except this time we will use the Public Key:

```
Original Message = M^E MOD N
```

If we plug that into a calculator, we get:

```
36^29 MOD 133 = 99
```

Diffie-Hellman

NOVEMBER 4, 2015 by ed harmoush 6 comments

This article is a part of a series on Cryptography. Use the navigation boxes to view the rest of the articles.

How can two people in a crowded room derive a secret that only the pair know, without revealing the secret to anyone else that might be listening?

That is exactly the scenario the Diffie-Hellman Key Exchange exists to solve.

The **Diffie-Hellman Key Exchange is a means for two parties to jointly establish a shared secret over an unsecure channel**, without having any prior knowledge of each other.

They never actually exchange the secret, just some values that both combine which let them attain the same resulting value.

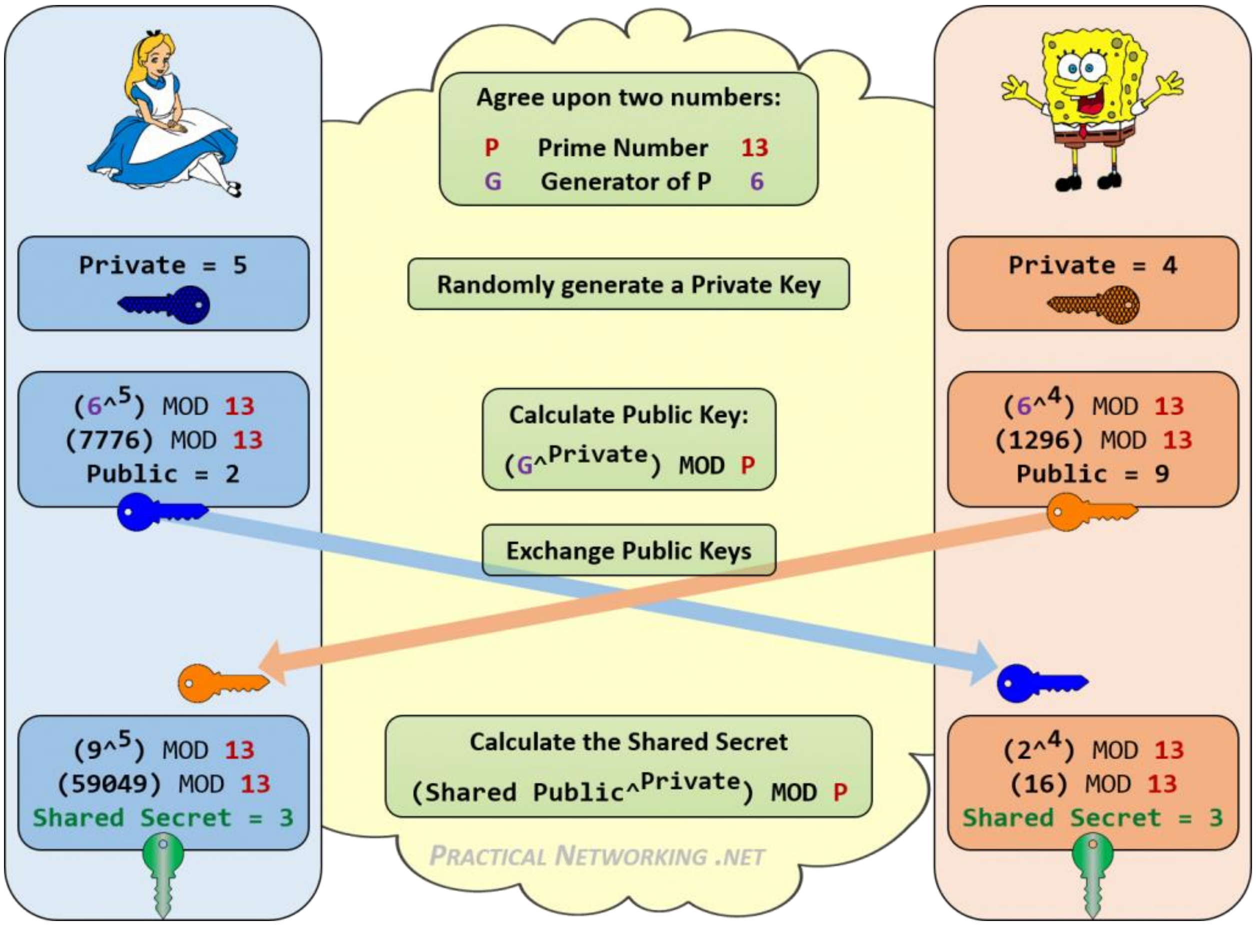
Conceptually, the best way to visualize the Diffie-Hellman Key Exchange is with the ubiquitous paint color mixing demonstration. It is worth quickly reviewing it if you are unfamiliar with it.

However, in this article we want to go a step further and actually show you the math in the Diffie-Hellman Key Exchange.

DH Math

Before you get into the math of Diffie-Hellman, you will want to have a basic understanding of what a **Prime** number is, and what the **Modulus** operation is (aka, remainder division). Both of these terms have been defined in another article.

Below is an infographic outlining all the steps of the Diffie-Hellman exchange between Alice and Bob.



Notice how both Alice and Bob were able to attain the same Shared Secret of 3. Anyone listening in on their DH Key exchange would only know the Public Values, and the starting P and G values. There is no consistent way to combine those numbers (13, 6, 2, 9) to attain 3.

DH Numbers

In our example, we used a Prime number of 13. Since this Prime number is also is used as the Modulus for each calculation, the entire key space for the resulting Shared Secret can only ever be 0-12. The bigger this number, the more difficult a time an attacker will have in brute forcing your shared secret.

Obviously, we were using very small numbers above to help keep the math relatively simple. True DH exchanges are doing math on numbers which are vastly larger. There are three typical sizes to the numbers in Diffie-Hellman:

DH Group 1	768 bits
DH Group 2	1024 bits
DH Group 5	1536 bits

The bit-size is a reference to the Prime number. This directly equates to the entire key space of the resulting Shared Secret. To give you an idea of just how large this key space is:

In order to fully write out a 768 bit number, you would need 232 decimal digits.
In order to fully write out a 1024 bit number, you would need 309 decimal digits.
In order to fully write out a 1536 bit number, you would need 463 decimal digits.

Using the Shared Secret

Once the Shared Secret has been attained, it typically becomes used in the calculation to establish a joint Symmetric Encryption key and/or a joint HMAC Key – also known as Session Keys.

But it is important to point out that the Shared Secret itself should not directly be used as the Secret Key. If it were, all you can be assured of is that throughout the secure conversation you are still speaking to the same party that was on the other side of the Diffie-Hellman exchange.

However, you still have no confirmation or assurance as to who the other party is. Just that no one else can all of a sudden pretend to be them in the middle of your secure conversation.

The generation of the actual Session Keys should include the DH Shared Secret, along with some other value that would only be known to the intended other party, like something from the Authentication scheme you chose.

Sources:

https://www.practicalnetworking.net/series/cryptography/cryptography/

https://www.practicalnetworking.net/series/cryptography/hashing-algorithm/

https://www.practicalnetworking.net/series/cryptography/message-integrity/

https://www.practicalnetworking.net/series/cryptography/confidentiality/

https://www.practicalnetworking.net/series/cryptography/symmetric-encryption/

https://www.practicalnetworking.net/series/cryptography/asymmetric-encryption/

https://www.practicalnetworking.net/series/cryptography/using-asymmetric-keys/

https://www.practicalnetworking.net/series/cryptography/authentication/

https://www.practicalnetworking.net/series/cryptography/anti-replay/

https://www.practicalnetworking.net/series/cryptography/rsa-example/

https://www.practicalnetworking.net/series/cryptography/diffie-hellman/