Linux and Command Prompt

Basics and How-To

What's a Command Prompt?

1. Also called Terminal or Bash and is used to execute commands.
2. It is typically a black screen with white font and has a very simple User Interface. You type in commands, and it interprets these commands and runs operations in the computer

SSH and How to do it?

1. SSH or "Secure Shell" is essentially a magic way to safely access unsafe networks. This is commonly used to remotely login to a computer and execute commands.
2. To use SSH to remote login:
    a. On Windows: Use an SSH client like PuTTY.
    b. On Mac or Linux: In terminal, type "*ssh {user}@{host}*". User refers to the account you want to access and host is the domain or IP address of the computer you are trying to access.
3. For a more in-depth understanding of SSH, we recommend this article.

The Swiss Army Knife: Netcat!

1. Netcat is a tool that can help you read or write data over the internet and is called "The Swiss Army Knife of Information Security" by its fans.
2. It earned its nickname because one can use netcat to perform a lot of different tasks including file transfer, chatting, port scanning and can even serve as both a client and a server.
3. The basic syntax for netcat commands is "*nc [options] [destination] [port]*". Here,
    a. *Options* is an optional argument or "flag" that you can use to change the behavior of netcat. For example "nc -h" prints helpful information about nc.
    b. *Destination*, is the IP address of the computer you are trying to contact.
    c. *Port* is the endpoint and helps identify the type of communication happening
4. There are many different uses of netcat and in general, you can get all the information by typing in "*nc -h*" into your terminal window! We also

recommend this article to understand how to use netcat. This other article contains a great list of all the *option flags* (look at the "Netcat Command Flags" box).

Fun With Terminal

1. Terminal has a bunch of useful commands that span a wide range of functionalities. You can navigate between folders (aka directories) using *cd* ("change directory"). You can test your net connection(s). You can create, delete or even edit files from within Terminal! Here is a great list of everyday (beginner) commands and even some more nuanced ones to help you do some really cool things.
2. **Warning!** Be careful with the *rm* ("remove") command. If you execute "*rm -rf* " into your terminal, you can delete all your files! So be careful in the terminal. You have a lot of power (and a lot of responsibility) so make sure you use the correct command and don't be afraid to use your favorite search engine (Google, Bing, DuckDuckGo, etc.) to search for something if you are unsure!
3. **Command Line Text Editor?** One particular section that might confuse some of you from the article mentioned above is "Intermediate Command 3: nano, vi...". If you know what these are, awesome; you can begin your journey doing everything in terminal. However, if you think nano is an old iPod, then you should reference the below section on text editors. For now, it is a text editor like Notepad that you can access from your terminal.
4. **Root Privileges!** Another important command that usually confuses people is "*sudo"* or *"SuperUserDO*". Essentially, this command allows you to have root privileges in your terminal. This is very risky as the root user has absolute power over the system and you can essentially do anything and possibly cause a lot of harm. But you can also do a lot of cool things so if you're using sudo, make sure you know what you are doing and that you're careful.
5. We explain some more important commands at the end of this document.

Who needs a Text Editor when you got Terminal?

1. There are many terminal text editors out there: *Nano, Vim, Emacs* and many more. They all have similar functionality but each programmer has their own

reasons why one is clearly superior to the other. So, what is your choice of editor?

2. Executing in *nano <filename.txt>* or *vim <filename.txt>* in your terminal. If a file with the same name exists, you will open it and if it doesn't, it will create a new file.
3. Here is a basic guide to using *GNU Nano* in the terminal!

Numbers, Numbers, Numbers!

1. You are most probably familiar with the Base-10 Decimal system! However, computer understand 1's and 0's or Binary (aka Base-2)! There are also other systems for example, Base-8 or Octal and Base-16 or Hexadecimal. In general, a Base-N system has digits from 0 to N-1. For example, Base-2 consists of 0 and 1 as digits.

Let's do an in depth analysis of some of the most important number systems.

Binary (Base -2)

1. *Digits*: 0, 1
2. *To convert Binary to Decimal*:
    a. Multiply each digit with 2^(position of that number)
    b. Add the above for all digits
    c. For example: $(10011)_2 = (1*2^4 + 0*2^3 + 0*2^2 + 1*2^1 + 1*2^0)_{10} = (19)_{10}$
3. *To convert Decimal to Binary*:
    a. Divide the number by 2
    b. Keep track of the remainder
    c. Divide the quotient by 2 and keep repeating till you get a quotient of 0 while keeping track of remainders
    d. Read off the remainders in reverse order and that is your number in Binary
    e. For example: $(6)_{10} = 2(3) + 0$, $(3)_{10} = 2(1) + 1$, $(1)_{10} = 2(0) + 1$. Therefore, $(6)_{10} = (110)_2$
4. *Addition:*
    a. In binary, there are 4 possibilities when adding 1 bit to another.
        i. 0 + 0 = 0 with a carry of 0
        ii. 0 + 1 = 1 with a carry of 0
        iii. 1 + 0 = 1 with a carry of 0
        iv. 1 + 1 = 0 with a carry of 1
    b. Addition is commutative and works exactly the same way as in decimal.

5. *Two's Complement Representation*: This is a form of representing Binary numbers which makes it very easy to store both positive and negative numbers and to find the binary representation of the negation of a given number.
    a. The leftmost bit (most significant bit) is the "sign bit" which means that it tells you whether the number is positive or negative. '1' means that the number is negative and '0' means it is positive.
    b. The rest of the digits represent the absolute value of the number in the normal Base 2 way.
    c. This means that given n-bits to represent a number, we can only use n-1 bits to represent the number. This leads to a largest positive number of $2^{(n-1)} - 1$ and smallest negative number as $-2^{(n-1)}$. Can you prove this to yourself? Try a few examples.
    d. Now how do you negate a number?
        i. Take the original two's complement representation of the number and flip all the bits (1 becomes 0 and vice versa)
        ii. Add 1 to this new number and this leads to the negative of the original number.
    e. To convert a two's complement number to decimal, you first check the leftmost bit to see if it is positive or negative. If it is positive, you convert to decimal as normal. If it is negative, negate the number, convert to decimal as normal, then add the negative sign back in front of it.
    f. $(4)_{10} = (0100)_2$ in Two's Complement form. Now let's apply our little algorithm! Flipping bits yields $(1011)_2$ and when you add 1, you get $(1100)_2$. Convert it to decimal and you will see that 1100 is the two's complement form of -4.
    g. Arithmetic works pretty much the same as normal binary numbers.

Little Endian and Big Endian!

These are ways to store numbers or data in memory addresses. Let's use a 16-bit word as example[1], $(0xFEED)_{16}$ in this case. Let's also assume we are storing this word starting at address 0x4000. We store these words in terms of bytes and not bits, so we need some conversion. Recall that 16 bits is 2 bytes (since 1 byte is 8 bits). The word is stored in pairs to make up the required 1 byte per memory location so our two parts will be 'FE' and 'ED'.

[1] In this case, 'word' means any number in Base-16 or Hexadecimal. So, a 16-bit word is a hexadecimal number with 16 bits or 4 values.

1. *Big Endian*: This refers to big end first, which means that we store the most significant byte at the smallest memory location and the rest follow normally. Therefore, to store our word in this case, memory location 0x4000 will have the byte *FE* and memory location 0x4001 will store the byte *ED*.
2. *Little Endian*: This refers to little end first and opposed to Big Endian, we store the least significant bit at the smallest memory address and the rest follow normally with the most significant bit at the last memory address. So, 0x4000 will have the byte *ED* and 0x4001 will have *FE*.

Big Endian is commonly used in Networking application, while Little Endian is most commonly used in processors.

Cryptography

How to Protect Your Data

Encryption is the act of changing information in such a way that only people who should be allowed to see the data are able to understand what the information is. The encrypted data can ideally only be read by trusted parties and look like a mess to everyone else. The simplest and oldest form of encryption is a Cipher!

Example Ciphers

1. **Caesar Cipher**: Probably one of the oldest, and easiest to break, ciphers.
    a. A key is chosen by the parties, which will be a number between 0 and 25. Then, each letter in the message is shifted forward or backward by that key to receive the encrypted message.
    b. For example, “This is an example” with a forward shift of 1 gives the encrypted message: “Uijt jt bo fybnqmf”.
    c. It is very easy to break as you can just apply all possible 26 forward and backward shifts and only one key will likely give an intelligible answer.

2. **Affine Cipher**: A substitution cipher where there is a formula for the substitution
    a. Each letter is assigned to a numerical value (usually starting the first letter at 0 and so on). Then a linear function is applied to the numerical value modulus the number of alphabets and then converted back to alphabet.
    b. So, let the numerical value of a certain alphabet be x. Then, the new value will be $(a*x + b) \bmod m$, where m is the number of alphabets, a and b are the keys picked by the two parties. This new value is then converted back to an alphabet.
    c. Also, notice that a Caesar cipher is an Affine cipher with a = 1.
3. **Vigenere Cipher**:
    a. This is a more complicated cipher than the ones we have seen already since we apply different shifts to each letter based on a pre-chosen key.

Here is an article with really good pictures that explains the method really well!

4. **Hill Cipher**: This technique requires a knowledge of Matrices. We cover the basics of Matrices here.
    a. The message, with n letters in it, is written as an n-element column vector. Each element of the vector is a number representation of the corresponding letter.
    b. The key is an n x n matrix, with random numbers in it.
    c. You then multiply the matrix with the vector and then modulo each element by 26. This basically means divide each element by 26 and replace the element with the remainder of that division.
    d. Then you convert the column vector back to letters and that is your encrypted message.
    e. Decryption is done in a similar way except you multiply the inverse of the key with the vector instead.

These are only a few important ciphers. However, we chose them because they cover most of the important topics in ciphers like shifting, modulus, matrices etc. but we recommend that you check out other ciphers too!

Hashing

Hashing is a very important concept in Computer Science and will function as a gateway from encrypting data to more foundational concepts of Cryptography in Information Security.

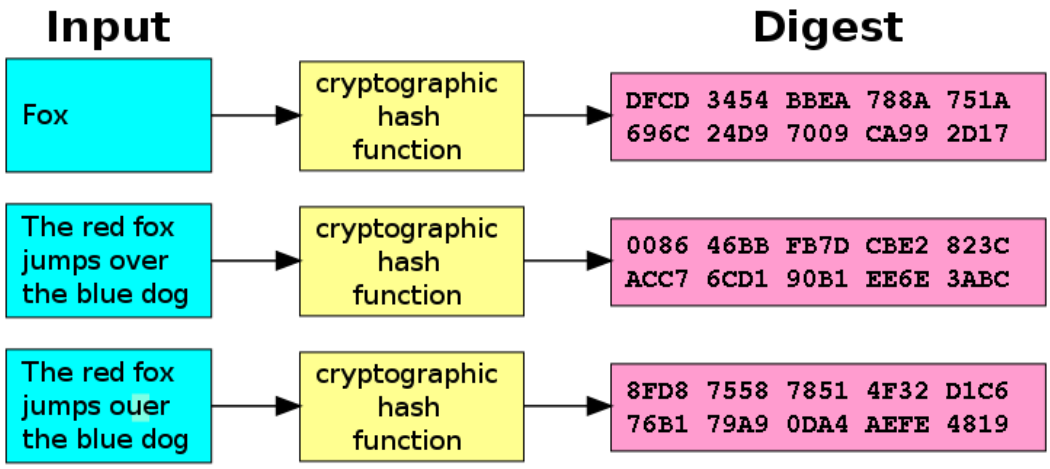
Hashing requires two things:

1. A Hash function: A mathematical function that takes an input and produces an output
2. A message: A string of symbols that is given as input to the hash function.

There are three main requirements of a good hash function:

1. It should be easy to calculate the hash value (output of hash function) given a message.
2. It should be very hard (ideally impossible) to calculate the original message given a hash value.

3. It should avoid collisions i.e. no two messages should have the same hash value.



Above, the input is the message and Digest is the hash value. In general, a hash function can be something as simple as f(x) = 1, where x is the input message. It is very easy to calculate and it's impossible to know what the message was given the hash value. However, every message has the same hash value and thus, this function is useless as it serves no purpose essentially.

Similarly, f(x) = x, is a horrible hash function! While it is easy to calculate, and we definitely avoid collisions, it is trivial to figure out the original message given the hash value, they are the same!

This is why these three conditions are chosen to make a good hash function. This is an in-depth exploration of hashing and how it is actually implemented in computers. However, it may be too complicated as it uses data structures, big-oh and other complicated Computer Science topics to explain a lot of important information on hashing. You have been warned!

Public Key Cryptography

Did you notice that there is a glaring issue with the ciphers that we discussed earlier? Everytime, we need to have a predetermined key and if someone gets access to the key, then they can easily decrypt the information and view all our secrets.

This wouldn't be as big an issue if both the parties live close-by and can meet to decide the key. But what if they live in different countries? What if one of them writes

the key down and then loses the paper? You don't want to have to meet everytime you want to change the key. With the advent of the internet, we need a new way encrypt information. You shouldn't have to go meet someone in person everytime you want to send them a message online. This gave birth to the idea of Public Key Cryptography.

Let's identify three individuals, Alice, Bob and Carl. Alice and Bob want to communicate over the internet and Carl wants to eavesdrop on their conversation! Alice and Bob need to create a key that -

1. Only they know
2. Can be decided over an unsecure network

At first, this seems impossible but there are many sophisticated techniques using some very simple Math, that can help Bob and Alice out. They will essentially need to decide their own personal keys that they keep private and a public key that everyone will have access to but do something clever so that only they can read the messages. One such technique is called the "*Diffie-Hellman Key Exchange*".

Diffie-Hellman Key Exchange

Here is how it works -

1. Alice and Bob publicly agree on two prime numbers, M and B.
2. Now, they both choose their own numbers called their private key, a and b respectively, and raise the base number B to the power of their respective private key modulo P, K_a and K_b respectively. They now publicly exchange these numbers. Note: So far, only *a* and *b* are unknown to Carl.

$$K_a = B^a \ (mod\ P)$$
$$K_b = B^b (mod\ P)$$

3. Now, they both raise the key they received to the power of their own private exponents.

Bob does : $K = (B^a mod\ P)^b = B^{ab}\ (mod\ P)$

Alice does: $K = (B^b mod\ P)^a = B^{ab}\ (mod\ P)$

4. Bob and Alice now have the same number *K* and Carl has no way of creating the key because he does not have access to *a* or *b*! Thus, *K* is a completely secret key that only Bob and Alice can know and Carl has no way of reproducing this key.
5. Alice and Bob can now begin encrypting their messages with this key and Carl can no longer eavesdrop!

Warning! The Diffie-Hellman approach is not perfect and has weaknesses that a hacker could take advantage of.

An important aspect of Public Key Cryptography is that it is an Asymmetric Key Encryption. This means that the key used to encrypt the message is different from the key used to decrypt the message.

In general, everyone has a pair of keys, called a key-pair, and these keys are mathematically linked in such a way that a message encrypted with one of them can only be decrypted with the other. You set one of them as your public key, which you publish to the world, and the other as your private key, which you tell no one. Now, this brings up two very important concepts in Cryptography - Authentication and Encryption!

Authentication: This refers to the authenticity of a message or basically is a way to confirm whether or not the message is from a particular user or someone else trying to copy that user. This is very important, because you cannot guarantee a secure connection if you don't know who the messages are coming from.

Encryption: We have covered this enough in our discussion but it essentially the act of encoding the message so only a particular user can read it. This is important because you cannot guarantee a secure connection if anyone can read your message.

How does Public Key Cryptography guarantee a secure connection?
As we discussed, to guarantee a secure connection, we must be able to successfully encrypt a message and ensure its authenticity. Our technique will be as follows -

1. When sharing a message, encrypt it with your partner's public key. This way, we are ensured that only the intended recipient can read the message since only they can decrypt the message using their private key (which no one else can figure out). Thus, we have ensured Encryption!
2. Also, encrypt your message with your own private key. Why? Well, since only you can encrypt a message with your private key, if a message sent out to the world can be decrypted by your public key, we are ensured that the message inside must be sent by you. This ensures the messages Authenticity!
3. Thus, when you use both techniques, you are guaranteed a secure connection without having to meet and physically exchange keys!

This video provides a really good explanation of this concept and it may be easier for you to follow along pictures than text. Both our discussion and the video, are abridged versions of these very complex concepts but will give you a basic idea of how this works. If you wish to know more, there are tons of great resources on the internet. Just go find them!

RSA Cryptosystem

Developed by Ron **Ri**vest, Adi **S**hamir and Leonard **A**dleman, the RSA encryption is one of the most commonly used type of public key cryptography. A *Cryptosystem* is a collection of similar algorithms needed to implement a particular service.

Here's a brief version of the algorithm:

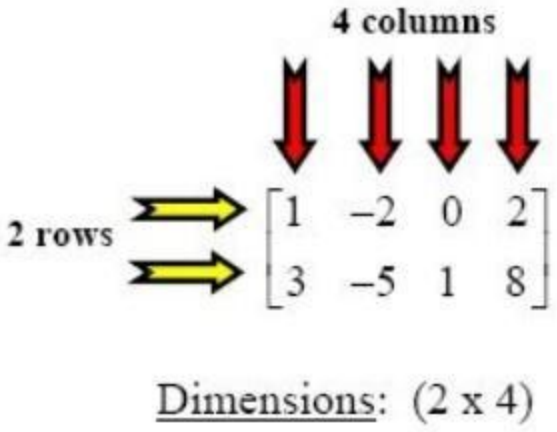
1. Privately select two large prime numbers, *P* and Q. If someone gains access to these, then you are vulnerable to attack.
2. Multiply the two numbers to create n = P x Q. This is your public key.
3. Calculate Φ(n) such that Φ(n) = (P – 1) x (Q – 1).
4. Choose a number, e, such that 1 < e < Φ(n).
5. Your total public key is (*n*, *e*).
6. Calculate d = (k*Φ(n) + 1)/e for some integer k. d is your private key!
7. Your total private key is (n,d).

To send a message m, the other person needs to calculate $x = m^e \pmod n$ and send x to you. This is the decrypted message. Now you encrypt it by calculating $x^d \pmod n$. This will give you back the original message m. The best way to use this algorithm is for the other person to sign the message with your public key and his own public key to ensure Authenticity and Encryption.

Mathematical Basis of Cyber Security

Matrices

A matrix is a mathematical tool to help store and manipulate numbers in an easy way!



Above is the example of a 2-by-4 matrix, which means it has 2 rows and 4 columns. The first number in dimensions is number of rows and second is number of columns. A Matrix can also have a name. Let's call this one 'A'. Each number is called an element of the matrix and can be identified by a unique pair of numbers or its position. For example, -5 is in the second row and second column, so we can call it A_{22}. Similarly, $A_{11} = A_{23} = 1$, where the first subscript is the row position and second number is the column position.

Matrix Algebra!

Matrix Addition and Subtraction are very similar to normal Addition and Subtraction, with the added requirement that the dimensions of the matrices must be the same. Then, you just have to add/subtract the corresponding elements to create a new third matrix.

Multiplication on the other hand, is a lot more involved. There are two kinds: Scalar and Matrix Multiplication. Scalar Multiplication is multiplying a Matrix by a number and for that, you just multiply each element by that number. Matrix Multiplication is multiplying a Matrix by another Matrix and here is a great tutorial on how to do Matrix Multiplication. NOTE: Order matters for Matrix Multiplication so $A \times B \neq B \times A$.

Matrix Division isn't really a thing but there is a way to sort of implement it, which we will talk about in a bit.

Important Matrices!

There are many important matrices that can be useful but the three most important for you to know are - Identity Matrix, Zero Matrix and Inverse Matrix.

Represented by the letter 'I', the *Identity Matrix* is the Matrix equivalent of '1' in that A x I = A. It is a square matrix (# of rows = # of columns) and all elements are 0 except for the elements on the diagonal, which are all 1.

The *Zero Matrix*, represented by the symbol 'ø', is the number equivalent of 0. It is a matrix of all 0's and functions in a similar way.

The *Inverse Matrix* doesn't refer to a specific Matrix but is a more general term for a matrix that when multiplied by another specific matrix, gives the identity matrix. Hence, it is a Multiplicative inverse of a particular matrix. For example, numbers have multiplicative inverses too. 2 has an inverse of ½ since, 2 x ½ = 1. Similarly, some matrices can have an inverse matrix represented by a superscript of -1. Here is a great article that talks about inverse matrices in depth and explains how one calculates it. This is also how we kind of do division in matrices, since multiplying by the inverse of a matrix is sort of like dividing by the original matrix!

There is a lot more to learn about Matrices. They are very important in solving systems of linear equations, in Computer graphics and in many cyber security problems! We cover a cool use of Matrices in Hill Ciphers here.

Modular Arithmetic

This is another foundational topic in Mathematics and Computer Science. This is arithmetic bound to a particular Modular environment. Now, what does this mean?

The modulus operator, usually written as '%' sign is an operation that means the remainder of. For example, 5 % 2 = 1, since the remainder when you divide 5 by 2 is 1. A modular environment is basically, when the highest number that can be represented in that environment is a particular number — 1. For example, in an environment modulus 6, we can only use numbers 0,1,2,3,4,5, because these are the only numbers that be remainders in a division by 6. Therefore, when any operation is done in a

modular environment, you must end that operation by taking the modulus of that answer by the number. This is represented by putting a '*mod n*' at the end of every equation where n is the number we are considering. For example, 7 x 5 (mod 6) = 35 (mod 6) = 5 (mod 6). Therefore,

$$7 \, x \, 5 \; = \; 5 \; (mod \; 6)$$

This is obviously very important to computers since computers can only store a certain number of digits and will eventually not have enough space to store a large enough number and thus need to implement modular arithmetic to avoid the situation when the result of a computation is higher than the largest number it can store. Note: A clock is an environment that is modulo 12 and military time is essentially a modulo 24 environment. It takes some while to get used to modular arithmetic since you can get stuff like

$$0 \; = \; 8 \; = 64 \; = \; 800... \; (mod \; 8)$$

However, it will begin to make sense with practice and time. It helps to think of it as normal numbers but instead of them being on a line, they are on a circle, like with time!

Fermat's Little Theorem

This 'little' theorem, is very important to computer security and surprisingly serves as a basis for most modern cryptography! It can be expressed as

$$\forall a \in \mathbb{Z}, \; \forall p \in \{primes\} \; a^p \equiv a \; (mod \; p)$$

In non-weird mathy terms, this means that if p is a prime number, then for any integer a, a^p - p will be an integer multiple of p. It might be useful to take a minute to convince yourself that this is what the equation means. We further restrain that a and p must be coprime (a is not divisible by p). Now, the theorem shows that

$$a^{p-1} \equiv 1 \; (mod \; p)$$

Or that $a^{p-1} - 1$ is divisible by p. This concept is critical to RSA encryption!

Cyber Attacks

Crack Me If You Can

“A Cyber Attacks if any malicious code to alter a computer’s code, logic or data”. In general, we will be covering any attempts to circumvent security of a system to get access to protected data by deciphering encrypted messages, getting access to passwords or stealing someone else’s identity.

Password Cracking

So let’s begin our conversation with something you have likely been trying to do ever since you have been on the internet, cracking your friend’s facebook password (don’t try this at home please). Password cracking may be done by the user themselves, to recover a forgotten password, but is more widely used by hackers to gain access to someone’s account. There are plenty of online tools but let’s discuss some techniques you can try yourself.

Password cracking is mostly done by trying to guess encrypted passwords that have been leaked online. Most big companies have a database of passwords that are associated with their users to authenticate the user when they try to log in. However, these passwords are not stored in plaintext, they are encrypted by using various techniques. If these encrypted passwords are leaked online, there are ways to figure out the original passwords from the hashes.

Brute Force and Dictionary Attacks

These are some of the most common password cracking techniques given a list of hashed passwords. Let’s start with brute force. You are probably familiar with this technique. Essentially, we try all possible passwords of a given length, hash them and test them against all leaked passwords. If there is a match, then the cracker knows that the hash belongs to that password. So, for example, if we want to find all 6-length passwords in the database, we hash all passwords from *aaaaaa* to *zzzzzz*. To make it more inclusive, the hacker can try adding upper case letters, symbols and numbers. This technique works for small passwords but as soon as you get to passwords of length 9 and above, it becomes pretty hard to do those in a realistic amount of time even with a lot of computing power.

This is where dictionary attacks come in. Our issue with brute force attacks is that the search space too large for us to be able to check all of them. Therefore, we do an approach that may also be familiar with you. Have you ever tried to hack your friend? It's likely that you didn't go through all possible passwords of a particular length as that would take forever. However, you know your friend and therefore, you have an idea of what kind of password they would use and you keep trying things important to them like friends and family names.

The dictionary attack is a similar approach. Since the attacker doesn't usually know the victim, they have to go on more general knowledge about people's password choices. For example, the password *"password"* and variations on it are very common. Hackers also use passwords that were leaked online earlier and other such things. This is where the database comes in. The attackers have a bunch of real and commonly used passwords that they try the hashes against.

Now, this would be very useful, except some people try to change things up by making some small changes. For example, switching an E with a 3 or a ! with a 1 to name a few. Therefore, hackers put these "rules" in and apply every rule to the passwords in the database to increase their chances of getting the write hash by a lot. There are several different databases and rule sets available online put there by different hackers that work very efficiently and can retrieve up to 70% of passwords at times.

There are many other algorithms that can help increase the speed of finding the hashes of particular passwords and some that even allow one to remove the need to figure out the plaintext password altogether but these usually require some issues in the implementation. I have found the website searchsecurity to have well written and easy to understand articles on all of these algorithms. There are also many online tools like *hashcat* and *Jack the Ripper* with excellent online documentations and videos on how to use them. This is a great video that shows an example of using hashcat and even some stuff we talked about if you're more of an audio-visual learner.

Breaking Ciphers

This section will be a discussion about how to break some of the most popular ciphers throughout history. We will go through some basic techniques to do so. The talk on encryption has a description of the ciphers we are discussing here.

Breaking the Caesar Cipher

Breaking a caesar cipher is usually a very easy task and can be brute forced. It is easy to apply all shifts (size 1-25 in English) and see which sentence after the shift makes sense.

However, there is a similar approach to the caesar cipher where you apply a different shift to each letter and the brute force is not as easy an approach in this case, especially if the shifting key gets very long. However, *frequency analysis* works particularly well in this case.

Frequency Analysis

In cases where a particular letter become another particular letters in ciphers, frequency analysis is a very useful technique since it relies on the fundamental semantic relation of a character within a language. For example, the letter '*e*' and "*th*" are very common in the English language. Linguists have created a average frequency of characters in English by using the appearance of the characters in known english texts.

These frequency relations must remain through the encryption. For example, if *f* appears the most in our enciphered text, it is highly likely that *e* was enciphered as *f*. We make such deductions and then try to test it by switching the particular characters in and seeing if we get meaningful words.

You may realize that this is surprisingly useful as it is independent of place and time given that the cracker has access to enough texts from that time or place. We can create the frequency distributions on any text especially for Affine Ciphers. Here is a good example on how to break an Affine Cipher.

Similar techniques can also be used to break ciphers where a character may or may not become the same character, like the Vigenere Cipher.

Vigenère Cipher

This is generally not an easy task to accomplish but there are several clever techniques that can break this cipher. I won't go into an in depth discussion but we can discuss the basics. You might notice that if we know the length of the key used to encrypt the data, it would be a lot easier to guess the key and understand the message. Two

popular ways are *The Kasiski Test* and *The Friedman Test*. *The Kasiski Test* depends on the fact that a long enough test is likely to have two repeating phrases in the encrypted message. Assuming that this means that those phrases refer to the same unencrypted phrase, the gap between tells us something about the size of the key. For example, if the gap is of length 16, we know that the key must be of size 1,2,4,8,16 as these would lead the key to be repeated exactly at the start of that phrase. *The Friedman Test* is a little more complicated but we end up with a few guesses for the key size.

Now, we have no option but to test each key size. Given a key size, we break up the text into columns with each column representing the characters that were encrypted by the same letter in the key. Now, we just have multiple caesar shifts and we use frequency analysis and brute force as mentioned earlier to figure out the exact key size and the value of the key.

Obviously this is making some assumptions but given a long enough text, these techniques do work. There are a lot of details that have been left out but this is a basic understanding of the different techniques used to break a Vigenere Cipher.

Problems with Key Exchange

In this section, we will discuss ways to attack the algorithms widely used over the internet for exchanging keys and establishing a secure connection. We have explained all these algorithms in an earlier guide on Cryptography.

Diffie-Hellman Key Exchange

I will assume that you understand the basics of this type of key exchange and are comfortable with how the algorithm works. If not, I would recommend reading the earlier section on how this is done.

The Diffie-Hellman Key Exchange is a great way to set up a connection with keys that are guaranteed to be secret. This checks the Encryption aspect of our key exchange algorithm. However, the biggest weakness of this algorithm is that it is impossible to verify the authenticity of the other person using it on its own. This kind of an attack is called a *man in the middle attack.*

Man In The Middle Attack

This is one of the oldest weaknesses of the Diffie-Hellman Key Exchange. Let's have our three users, Alice, Bob and Carl again. Alice and Bob are trying to establish a secure connection and are using the Diffie-Hellman Key Exchange to create their keys. However, Carl is trying to read their conversations without either of them knowing anything weird is happening.

Carl is stuck because he knows that the Diffie-Hellman algorithm, allows two users to create keys without anyone else finding out what the keys are. However, he realizes the flaw that he can put himself in the middle of this conversation. Once Alice wants to talk to Bob, she will start the algorithm, however, Carl intercepts her first message and tells Alice that he is Bob. There is no way of her telling the difference over the internet as all that is exchanged is numbers. So he establishes a "secure" connection with Alice and she thinks she has a secure connection with Bob. Now, Carl tells Bob that he is Alice and creates a secure connection with him. Now, there are two secure connections: Alice and Carl, Carl and Bob that work as one connection. Alice sends a message to Carl, he decrypts it with the Alice-Carl key, he does whatever he wants with that message and then encrypts it with the Bob-Carl key and sends it to Bob and so on. Therefore, the Alice-Bob conversation is now happening through Carl and neither of them are the wiser. This is a classic Man in the middle attack and happened very recently in 2015 (search superfish man in the middle attack).

There are more issues with the Diffie-Hellman approach but they are based on mistakes that the programmer who writes the algorithm can make. For example, if the value P, which is the large prime number that is the modular environment, is too small, then the possible key values in public might be easy to brute-force. The attacker can easily try all values of K^a and K^b. Since the have the K^{ab} they can realistically figure out the private keys and then the connection is insecure. Therefore, P needs to be large enough so that brute-force attacks are unfeasible.

RSA Cryptosystems

In general, it is hard to break the RSA algorithms. It is more important to discuss what mistakes can be made by a programmer in their implementation of an RSA algorithm that makes it easier to hack into. *Note: I will be assuming that you have an understanding of the RSA cryptosystem and will be using the same nomenclature as used in the previous section on encryption.*

Using Small Prime Numbers P and Q:

The most basic mistake that can be made is to use small prime numbers for P and Q. These numbers cannot be too large or too small. First, we must understand that *P*Q* is the modular environment *n*. This means that *n* defines what our permitted range of numbers will be (*0 - n-1*). Therefore, if *n* is too large, our algorithm would be too inefficient, which means starting up our process would be too slow. If the value is too small, then our algorithm would be open to a brute force attack. Since, the strength of and public key cryptography algorithm lies in the fact that the private keys cannot be reverse-engineered from the public components, if our modular environment is too small, the private keys can be extracted from the public components and we lose the *Encryption* aspect of our algorithm. There is a good range of values for *n* and this lies around 2000-4000 bits long.

Wiener's Attack:

Michael J. Wiener proved that if the value chosen for *d* is too small, an attacker can efficiently find the value of *d* which is the essential part if the private key. With this information, and the available public key *e* and attacker can easily encrypt the message. However, if *e* is large enough, it doesn't matter how small *d* is and Wiener's attack cannot be applied. *Note I glossed over a bunch of details but here is an in-depth look into the math behind this attack.*

This is a fairly detailed explanation of some more attacks. *Warning*: this article involves a lot of math and I wouldn't recommend reading it for anything more than recreational purposes unless you understand the RSA Cryptosystems well.

Multi-Prime RSA:

This refers to the implementation of the RSA cryptosystem where *n* is the product of more than 2 primes. This algorithm is useful because it is faster and more space-efficient in the decryption phase (thanks to the Chinese Remainder Theorem). We can break up the huge modulus step into smaller steps using the Chinese Remainder Theorem and this provides a huge boost in speed. However, this method hasn't been studied enough to prove it is just as secure. It is recognized that if 3-4 primes are used, a significant boost can be noticed in speed.

However, if more factors are used, the size of individual factors would become too small and it would become a lot easier to use techniques to factor out the *n* and thus break the algorithm.

Chinese Remainder Theorem

Chinese Remainder Theorem

Given pairwise coprime positive integers $n_1, n_2, \ldots, n_k$ and arbitrary integers $a_1, a_2, \ldots, a_k$, the system of simultaneous congruences

$$\begin{aligned} x &\equiv a_1 \pmod{n_1} \\ x &\equiv a_2 \pmod{n_2} \\ &\vdots \\ x &\equiv a_k \pmod{n_k} \end{aligned}$$

has a solution, and the solution is unique modulo $N = n_1 n_2 \cdots n_k$.

Source: https://brilliant.org/wiki/chinese-remainder-theorem/

Above is a very important theorem relating to RSA cryptosystem and its weaknesses. So we will briefly discuss it to aid the section on breaking the RSA Cryptosystem. Each mathematical expression means that if you divide x by n_k, the remainder is a_k. The theorem means that in any linear system as above, there is at least a single value for x that fulfills the requirements of the system modulo N.

1. Compute $N = n_1 \times n_2 \times \cdots \times n_k$.
2. For each $i = 1, 2, \ldots, k$, compute
$$y_i = \frac{N}{n_i} = n_1 n_2 \cdots n_{i-1} n_{i+1} \cdots n_k.$$
3. For each $i = 1, 2, \ldots, k$, compute $z_i \equiv y_i^{-1} \bmod n_i$ using Euclid's extended algorithm (z_i exists since $n_1, n_2, \ldots, n_k$ are pairwise coprime).
4. The integer $x = \sum_{i=1}^{k} a_i y_i z_i$ is a solution to the system of congruences, and $x \bmod N$ is the unique solution modulo N.

Source: https://brilliant.org/wiki/chinese-remainder-theorem/

Above is the way to compute the value of x. A proof of the theorem can be found here.

Web Exploitation

How to become an online spider

Computer Networks

Modern life would be very different without computer networks. These generally comprise of multiple computers (‘*nodes*’), that are connected together to share data and resources. The most popular Computer Network is *The Internet*, which specifically connects computers that use the Internet Protocol or ‘IP’.

How does the Internet work?

Completely new and need to know the basics? Here is a great article that explains the very basic architecture of the internet and how data is transmitted.

Website Basics

Now information on the Internet is segregated by websites. They are a collection of web pages and are referred to by a domain name (like google.com, facebook.com). Each web page is referred to by its URL or Uniform Resource Locator.

1. What is a web page and website?
   A website is a collection of web pages. So website would be like a house and each webpage would be a room inside the house.

2. Breakdown of a URL:
   https://developer.mozilla.org/en-US/docs/Learn/Common_questions/What_is_a_URL

3. Querying:
   https://en.wikipedia.org/wiki/Query_string

4. Different parts of a website and how to mess with it (HTML, CSS, JS, Backend)
   a. HTML breakdown
      Here is a basic tutorial on HTML:
      https://www.w3schools.com/html/html_basic.asp

b. CSS breakdown
   Here is a basic tutorial on CSS:

   https://www.w3schools.com/css/css_intro.asp

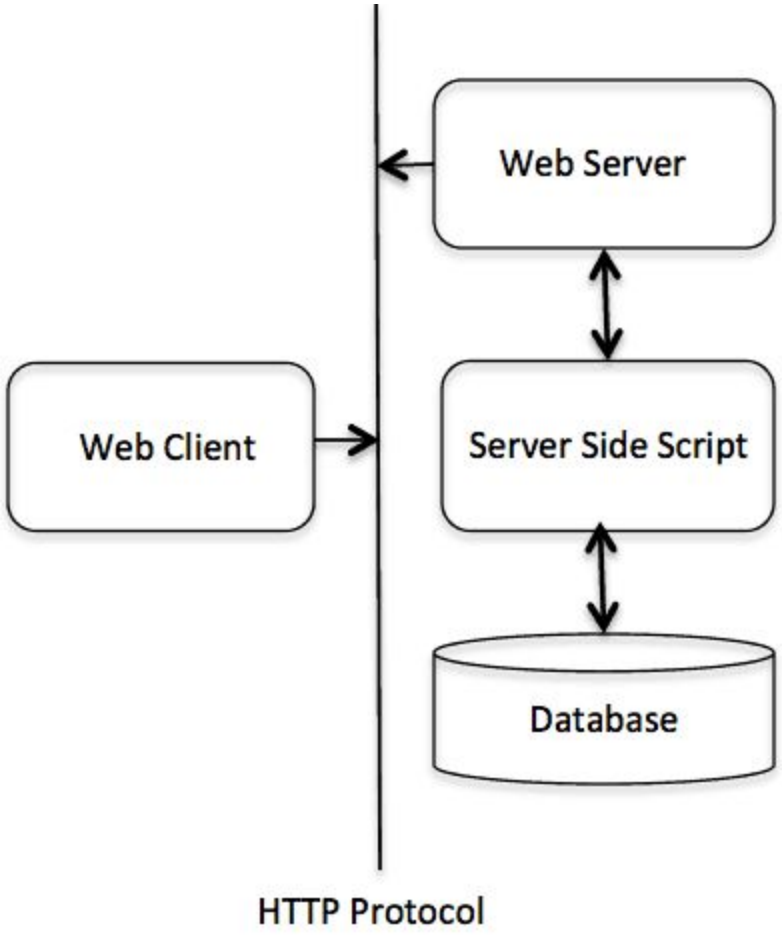
5. Viewing source
   By right clicking on Google Chrome or Firefox you can select the option “View Page Source” to see the code that the website is running on your computer. It allows you to see the HTML and CSS that is running on the website and it will also let you see the Javascript scripts running on your computer. The best part is, that you can edit the HTML directly and see it affect the website, so it lets you modify the website as you desire. You can also select “Inspect Element” to see the code that is running in a specific part of a website.

JS Breakdown

1. Why we need it?
   Javascript is used because it allows us to add interactivity between the user and the website. Javascript allows the user to interact with the website and have the website respond.
2. Basics - Editing elements HTML
   https://www.w3schools.com/js/js_htmldom_html.asp

HTTP breakdown

1. What is HTTP?
   It provides a standardized way for computers to communicate with each other over the internet. HTTP is a communication protocol, that is used to deliver data (HTML files, image files, query results, etc.) over the internet. HTTP dictates how data is sent between clients (you) and servers.
2. GET and POST request
   https://www.w3schools.com/tags/ref_httpmethods.asp
3. Basic Architecture



4. Basic idea of a proxy
   A proxy server is a computer on the web that redirects your web browsing activity. When you try to access any website, your Internet Service Provider (ISP) makes the request for you and gives the website your IP address. So when you use a proxy, your request goes from your ISP to the proxy server to the website you want to go to. This way allows you to mask your IP address as another address so that the websites you access don't know who you are.

Database breakdown

1. What they are and why they are useful
   A database is a collection of information that is organized so that it can be easily accessed, managed and updated. Databases can quickly query data and add/delete data instantly. They are used to hold every kind of data.
2. SQL and others

SQL is Structured Query Language (SQL), a programming language used for managing relational databases. Relational databases are tabular database in which data is defined so that it can be reorganized and accessed in a number of different ways. Relational databases are easy to extend, and a new data category can be added after the original database creation without requiring that you modify all the existing applications. Relational databases are made up of a set of tables with data that fits into a predefined category. Each table has at least one data category in a column, and each row has a certain data instance for the categories which are defined in the columns.

3. How they integrate into sites
   Databases are integrated into websites because they are the most optimal way to display/store data. User information like passwords are stored using databases. Databases also allow for quickly modifying the data displayed on the website. So if someone wants to update information on a website instead of modifying the HTML on the website, they can just change the data on the database that is displayed on the website.

4. Basic SQL syntax
   a. SELECT
      Extracts data
   b. ORDER BY
      Orders the results gotten from SELECT in a specific manner. For example, if one has a table of countries and their populations. One can select the countries starting with the letter R and then order them by their population.
   c. JOIN
      Joins data from two tables depending on a certain characteristic on the table. So if a theres two table one with customer IDs and their addresses and another table with customer IDs and their purchases. You can join both tables so that the customer addresses match their purchases.
   d. DELETE, INSERT
      Allows you to delete data or add new data to a table.
   e. AND,OR
      Allows you to modify queries so that they return information depending on multiple categories.
   f. MIN,MAX
      They return the smallest or largest value of a query.

Injections

1. How to perform a basic SQL injection and how its possible
   SQL Injection (SQLi) refers to an injection attack where an attacker can execute whatever SQL commands they want that control a web application's database server. Websites use the information you give them to query SQL, for example when logging onto a website, the website will query your username and password to see if you are an authorized user. A SQL injection would allow the user to supply their own SQL code and run it on the website.
2. How to safeguard against SQL injection(safely accept user input)
   a. Prepared statements
      Prepared statements are a way to separate code and user supplied input. It's the most common way to avoid attackers from attempting to run SQL code by supplying it as input. Prepared statements will not execute the SQL code and will treat the input as a query and not as code.
3. How to perform a PHP object injection
   https://www.tarlogic.com/en/blog/how-php-object-injection-works-php-object-injection/

Glossary

1. *'IP' or 'Internet Protocol'*: A set of rules that govern how data is transmitted over The Internet.
2. *IP Address*: A unique name given to every computer connected to the internet. It looks like 'a.b.c.d' where each of a,b,c,d is a number between 0 and 255.
3. *Packet*: In terms of the Internet, if the amount of data being transmitted is too large, we break it down into smaller chunks, called packets.
4. *Port Number*: In Networking, a port is an endpoint of the communication and the port number is the specific number associate with a particular port.
5. *Domain Name Service (DNS)* : A database which stores the IP address of each website and its *domain name* (like google.com).
6. *Client:* These are usually computers of users looking to access web pages or search engines. These are usually the ones looking to get a particular service.
7. *Server:* These are computers that store web pages, services or applications. They are usually the ones providing the service.
8. *Webpage:* A single hypertext document that is connected to the World Wide Web.
9. *Website*: A collection of related web pages usually connected to one common domain name.

Forensics

Who Framed Roger Rab-bit?

Files and the File System

A File System is a like an index for all the files in your computer system! You can find a reference to all the files in your computer.

File Systems have a lot of different parts but the main ones include ‘Files’, ‘Directories/Folders’ and ‘Metadata’. Files are used to store data and have two parts - ‘Filename’ or the name used to refer to the file and ‘type’ or the kind of data that is stored in the file. Directories are collections of files grouped together in some way. Metadata is the data about the file itself like its length, time created and author etc.

When a file is created, the data is stored at some position in memory and the filename is the reference to that data. It is like the address of your friends house. The address of your friends house has nothing to do with your friends house but it is a simple way that we have come up with to remember where certain places are located. If you forget the address, your friends house doesn’t get removed. Similarly, sometimes when you delete a file, all that is happening is that your computer just doesn’t remember where that file is stored. It probably still exists at that same memory location until you write something there again.

A file consists of Blocks, which is the smallest part of the data that is stored in memory. The header block contains the starting point of the file whereas the footer block contains the end point of the file.

File Carving

“*File carving* refers to the technique to extract data from a disk drive without having the normal file system to easily recover the files.” It is a method of recovering files when there is not reference to them in the computer. It is mostly used to recover deleted files by criminal detectives.

From our previous conversation about deleting files, we know that when a file gets deleted, the data doesn't get deleted but just the reference to that data. Another important change that happens when you delete is file is that the disk location where the file is stored gets marked as unallocated and can thus be overwritten. However, it is possible to use techniques to recover most, if not all, of the data.

Binary Exploitation

This section talks about exploiting information at a register level. We will talk about debugging programs, how to hack into programs to make them do something different from their intended use, how to safeguard against such attacks and much more.

Debugging is essential for any serious programmer. It is unlikely that you will write code that works flawlessly the first time. However, there are many tips and tricks that can help the sometimes painful process of debugging go more smoothly.

There are many things that you can do but the easiest way to debug is print information that you think is useful and try to identify the location and the source of the bug. Start by identifying where the execution of the program halts, then try removing parts of the code until you know exactly what causes the crash or the error to occur. Now, is the “easy” part of understanding why the error is happening, what you intended to happen and what you should do to fix the error. This was just the first step to debugging and there are many more techniques and tools that can help you out. Let’s discuss one of the most important - GDB.

GDB - The GNU Project Debugger

GDB or The GNU Project Debugger is a very widely used software that can help programmers catch bugs. It can help start and stop the execution of the program at any stage, look at memory and register values, change things around in the program and many more useful tools to help in debugging. This is an essential tool in any Systems programmers arsenal so we will cover the essential part of GDB. *Note: You should have a working knowledge of C/C++, registers and memory before you continue. We recommend checking out our section on Reversing.*

Now, it isn’t important to know the inner workings of GDB to use it but we will cover just the basics to get you to appreciate the tool. GDB consists of two major parts -

1. The Symbol Side: This is concerned with the “symbolic side” of the program, which means the metadata about functions, variables, the source code and essentially the program in the way it was written.

2. The Target Side: This part deals with all the manipulations that can be done to the execution of the program like reading registers, accepting signals and starting or stopping the execution. In UNIX, it uses the system call called "ptrace" to accomplish this.

Both the parts are largely separate, however, the command interpreter and main control loop tie these two parts together. Here is an excellent article on all the inner workings of the GDB and talks about all the data structures used, how the different parts work and how it came about. It is a very interesting read but might be too complicated if you are a new programmer.

Here is a tutorial on almost everything you could want to know about GDB but we will cover some of the more fundamental concepts now.

Starting up the environment:

To compile the code (in C), type

```
$ gcc -g filename.c -o executableName
```

Here, gcc is the GNU C Compiler, the *-g* flag tells the compiler that you intend to use GDB, filename.c is the name of the file you wish to compile and *-o* is the optional flag that tells the compiler what you want the name of the executable file to be (default is a.out).

Now to start up GDB, type

```
$ gdb executableName
```

This just says to open up the executable file with gdb. You should see a bunch of information talking about the gdb (this mostly not important). There will be a new interface starting with *(gdb),* and this is GDB interface where you can use many different commands. Here are some important ones -

1. 'break' or 'b': This command is always associated with a line in the code and tells the compiler to stop program execution when it reaches this line. It is usually followed by a line in order to put a 'breakpoint' at that line. You can obtain information about all breakpoints by typing 'info b'.
2. 'run' or 'r': This command runs the file that was loaded into gdb. This program will run the file normally if no issues but will crash if there is an error or break and give useful information about why it stopped.

3. ‘step’ or ‘s’: Runs the next line of the program and essentially steps through the code. It will enter each function it encounters as opposed to ‘next’, which just runs sub routines executing them line by line.
4. ‘Print’ or ‘p’: Prints the value of an argument (variable or registers) given to it.

These commands can help you run, stop and step through your program and debug it by using printing and thus make up the most important commands. This is a great tutorial explaining the use of some more basic commands. Execute ‘*layout regs*’ before running your program as this will show you the assembly code and the register and memory address values while you run your code. This is really helpful while debugging.

Reversing

Time to get with the program

This guide is a brief introduction to C, Assembly Language, and Python that will be helpful for solving *Reversing* challenges.

Writing a C Program

C is one of the most important modern programming languages and finds use in almost all system programming applications. First thing first, here is a great tutorial on C and goes into great detail about programming in C. The C language is a must know for any modern programmer, so let's cover some important Computer Science concepts using C.

Let's start with the concept of a function. A function is essentially a chunk of code that takes an input and returns an output. It has a name, an arbitrary number of optional parameters or inputs, instructions, and an optional return value or output.

```
int doubleNum(int a) {
   // Take a number and multiply it by 2
   int x = a * 2;
   return x;
}
```

This is a very basic program but the concepts in it are the same concepts used to write huge, complicated applications.

Let's dissect each part of the program to gain a deeper understanding. The block of code in general is called a *function* and is a set of instructions that usually complete a logical function like multiplying a number by 2. It usually has a name, here 'doubleNum'. Now, another important aspect of most programs is *datatype*, which quite literally is the kind of data that we are dealing with. It is usually associated with a *variable* which is just a structure that stores data. Here, the "int a" is a parameter for the function "doubleNum" and means that this function takes one number as input. The first "int" means that this function returns an integer as output.

x and *a* are variables that both store integers and are connected since $x = 2*a$.

The return statement tells the computer that this function should give *x* as an output. The last important part is the comments, denoted by the '//' and is essentially telling the computer to ignore the rest of the words on that line. You can write whatever you want there and is essentially just meant to explain the logic in your code.

Now that you know the essentials of writing a C function (or a function in any language really), we can talk more about how a computer even understanding what you are trying to get it to do. Whenever, I say that a particular keyword tells the computer to do something, I really mean that the programmer is telling the compiler to understand the rest of the stuff in a particular context. The compiler essentially converts a High level language into a low level language. It eventually gets converted into assembly language and finally into 1's and 0's, which finally the computer can understand and execute.

Compiling and Executing a C Program

There are multiple steps to converting a program from C to something a computer can understand. These steps include Compiling, Assembling and Linking. The compiler converts the code from C to assembly language (which we'll discuss later), and the assembler converts the code from assembly language to executable object. The main compiler in C is called GCC (GNU C Compiler) and has the following syntax:

```
$ gcc -Wall filename.c -o filename
```

The *filename.c* is a placeholder for the C file you want to compile. The *-Wall* and *-o* are flags that tell the compiler options that can change the way the compiler works. *-o* tells the compiler that the next token is the name of the executable object, *filename.o* and *-Wall* enables warning messages. Now, to run the file, you need to type

```
$ ./filename
```

which runs the file and takes any inputs and returns any outputs.

Assembly

We want to learn how to program in Assembly to cause changes without the added layer of abstraction provided by higher level languages and thus write code that runs faster and has more capabilities. Here is a great tutorial on everything you could want to know about assembly programming but as always, I will go through some of the more important concepts and provide a basic overview to help you get started.

Assembly programming is very different from normal programming in that we have to worry about things that high level language programmers don't have to worry about, like where something is stored and how it is stored in memory. However, we still have a program made of statements that the programmer writes that follows the syntax as such -

Now, let's dissect this -

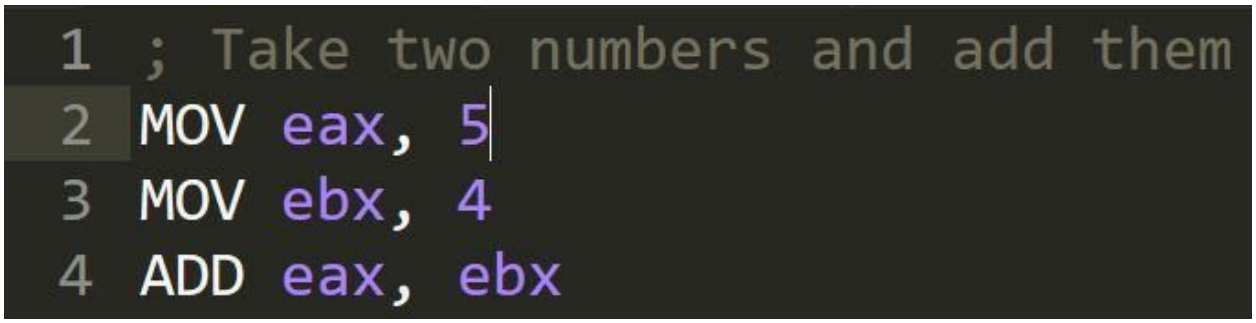
1. *label* - This is a optional personal name we give to a statement in our code
2. *name* - This is the name of the operation we want to perform (ADD, MOV etc.)
3. *operands* - This is the optional list of parameters to the function
4. *comment* - This is denoted by a ; and functions as a comment in normal languages

Registers

Before we can move to talking about different instructions in assembly language, we need to talk about a very important concept - *registers*. In general, we want to be able to store variables and we cannot store them directly into main memory as that would make access to the data very slow. Therefore, most computers have 32 special locations in their CPU where we can store 32 bits of information. There special locations are called registers and help us store and recall variables quickly and easily.

Some Common Instructions

Here is a list of common instructions in the X86-64 instruction set. However, let's discuss some important instructions to get an idea of what is going on. Here is a sample program that performs a simple exercise of adding two numbers.



We start this off with a comment explaining what this code snippet is doing. *MOV* is the instruction that takes a register and a register or data point and puts it into the other register. It has the syntax as follows

MOV Dest, Source

Here, the destination is eax which is a register and 5 which is a data value. Therefore, we put the integer values 5 and 4 into registers *rax* and *rbx* respectively. Then, we perform the *ADD* instruction that adds the two data points it is given.

ADD Dest, Source

This instruction takes the data in *Source* and adds it to the data in *Dest* and stores it back in dest. Therefore, it adds the value 4 into the *rax* register and store the result, 9 back into *rax* register.

This is only a very simple example of assembly programming. However, you can understand what some code is trying to do by following the similar concept. Pull up the descriptions of the instructions online and start converting the snippet, line by line, and piece together what the code is trying to do. Note, we did not use labels.

However, using just labels and a JMP statement, we can implement loops, functions and recursion!

Big Endian vs. Little Endian

These are ways to store numbers or data in memory addresses. Let's use a 16-bit word as example[1], $(0xFEED)_{16}$ in this case. Let's also assume we are storing this word starting at address 0x4000. We store these words in terms of bytes and not bits, so we need some conversion. Recall that 16 bits is 2 bytes (since 1 byte is 8 bits). The word is stored in pairs to make up the required 1 byte per memory location so our two parts will be 'FE' and 'ED'.

1. *Big Endian*: This refers to big end first, which means that we store the most significant byte at the smallest memory location and the rest follow normally. Therefore, to store our word in this case, memory location 0x4000 will have the byte *FE* and memory location 0x4001 will store the byte *ED*.
2. *Little Endian*: This refers to little end first and opposed to Big Endian, we store the least significant bit at the smallest memory address and the rest follow normally with the most significant bit at the last memory address. So, 0x4000 will have the byte *ED* and 0x4001 will have *FE*.

Big Endian is commonly used in Networking application, while Little Endian is most commonly used in processors

Python

Python is an amazing High Level language that is both easy to learn and allows the programmer to do a lot of different tasks. That is probably why it is used so frequently! Here is a link to Carnegie Mellon's introductory CS course using Python and you can pretty much find whatever you could want to know about the basics of Python and CS on this website.

[1] In this case, 'word' means any number in Base-16 or Hexadecimal. So, a 16-bit word is a hexadecimal number with 16 bits or 4 values.