International Institute of Information Technology, Hyderabad

Information Security

Cryptography, Pwning and Exploit Research

Hacking Handbook

Author: Kalp Shah

April 6, 2020

Abstract

A thorough foray into exploitation and explanation of why it works the way it does. It is a compilation of sorts, and hence no correlation between chapters (No flow b/w chapters exist) and the book should exist as a case study and used to solve problems which look similar and also has some basics which I go learning on the way forward. "*If it moves, compile it*".

Sources

The websites and competitions referred for writing this piece:

- CTF 101
- Shellter Labs
- Linux Man Pages
- picoCTF
- trailofbits
- Florida University Computer Security
- Megabeets Radare Tutorial
- Android Developers Website

Chapter 1

Basics

1.1 Data

There are many places to store data on a typical computer like a hard drive (or any other secondary storage device), RAM (SRAM and DRAM), CPU Caches and Registers. A hierarchy is decided on their access speed. If a piece of data is required more frequently, it is stored on a faster storage device. (Faster implies lower latency i.e. time taken from being asked for data and providing it). The figure 1 shows this heirarchy.

Location

Register

A register is located in the CPU itself and as it is physically the closest, it also the fastest. It is the one that is accessed while running a program.
A sample data flow can be seen in Figure 2, which shows data tranfer from RAM to Register and then to the CPU.

Cache

A cache is small piece of memory located near the CPU and works as a faster RAM (sort of). It caches the data which the OS thinks is required the most.
L1 & L2 cache is built into the proccessor (recent ones anyway) and is individual for every physical core whereas L3 cache is located outside of the actual silicon but is in the chip and so is common for all cores.

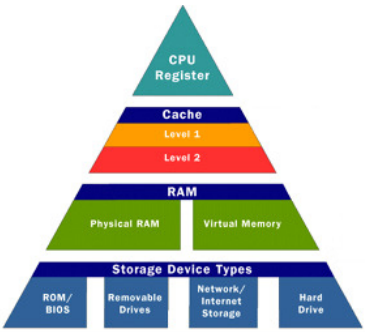

Figure 1.1: Latency Hierarchy

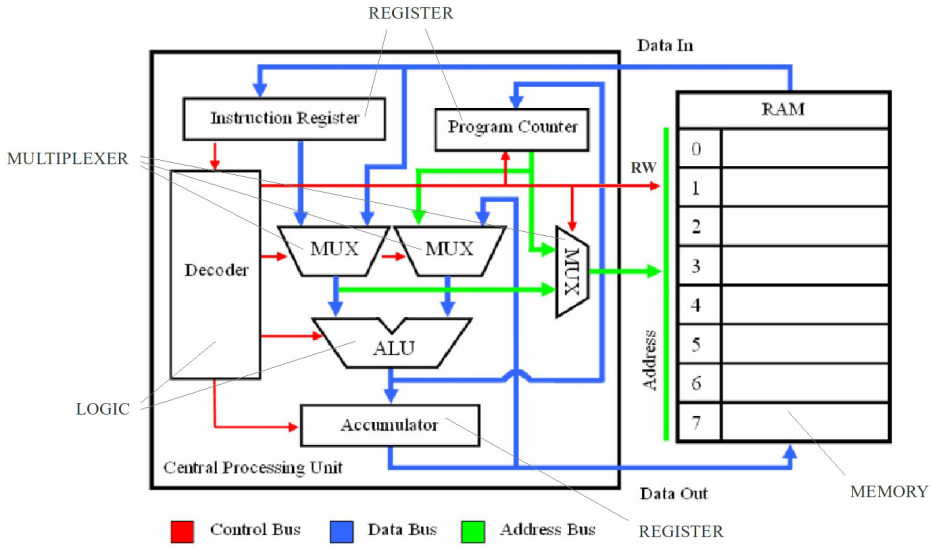

Figure 1.2: Circuit of CPU

RAM

RAM is a volatile memory where all the memory that is deemed by the OS as required is stored for instant access with the CPU. The data tranfer happens from RAM → Cache → Register.

Bridges

Bridges are small microcontrollers with built in logic which help in communication with external devices. There are two of there, and are descibed by their position on the board and also the threshold speed they can manage.

North Bridge

North bridge is a controller chip which connect high speed external devices to the chipset. It was originaly outside ot the main chip and an external silicon but now is mostly included in the SoC (System on a chip). The devices which can be connected to it are given in Figure 3.

South Bridge

South bridge is the chipset which connects slower and legacy devices to the main chip. It is still seperate on Intel boards but is now starting to get integrated in AMD motherboards.

Data Flow

So the data flow happens as follows :

$$Solid\ State \rightarrow South\ Bridge \rightarrow RAM \rightarrow Register$$

So when a computer is started, it first POSTS and then goes to the BIOS after which the BIOS piont the program to the Magic Number (For legacy MBR systems) which has the bootloader in it (Like GRUB), after which the bootloader takes control of the System, then loades the kernel and with that the OS, which loads itself in the RAM, for fast access (The most recently executed programs still in the registers and cache) after which the OS loads

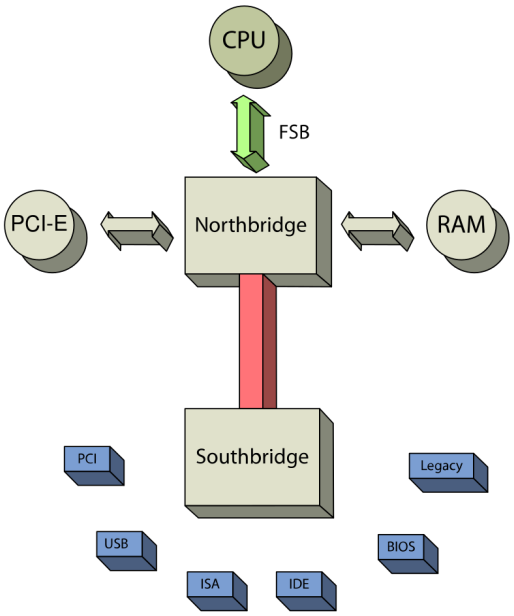

Figure 1.3: Use of bridges

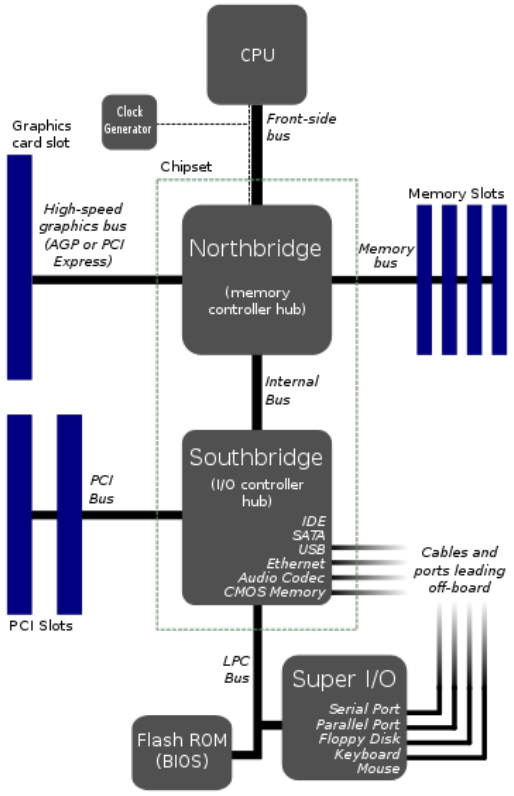

Figure 1.4: Schematic Design of Bridges

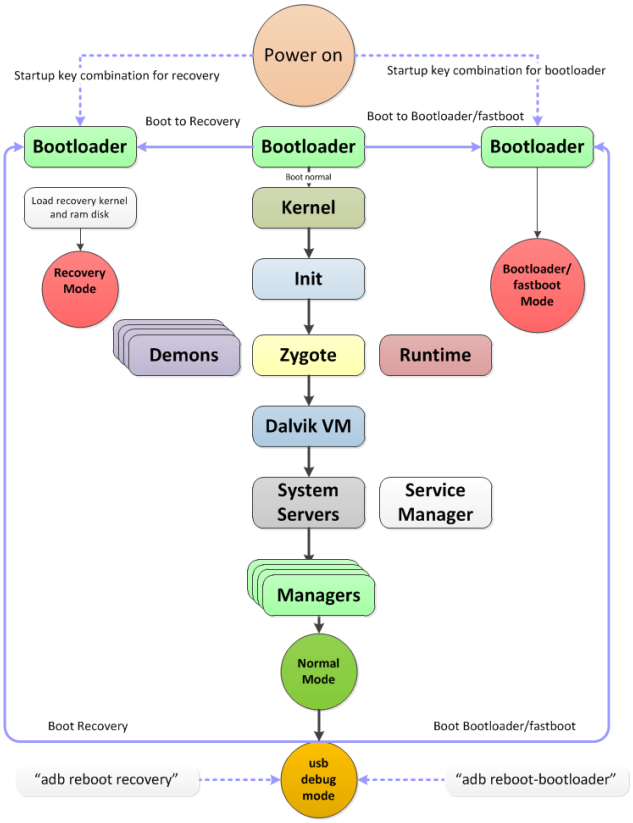

Figure 1.5: Android booting process

a GUI (If it has one) or just greets to a TTY terminal. For the non GUI version, it gets you to a log in screen, whicle for GUI an application is loaded which takes care of logging in (Known as a Display Manager (DM)).

Chapter 2

Tools

There are many tools which make analysing and understaing flaws of programs very easy. Most of them are not required and work can be done without them, but they are a good creature comfort and sholud be used as such. These tools are used to analyse a piece of code and understand its vulnerabilites and also to generate payloads to make exploiting them easier.

2.1 GDB

GDB is one of the most importatnt tools. It is one of, if not the most importatnt tool for exploiting binaries. It is essentially a debugger which allows for dissasmbly of binaries, is also usefull for checking the flow of the the binary, and before Ghidra was one of the most popular tools to understand the working of a program. It is still used for basic analysis, to check if the exploit works, and for initial routing checking. If someone is starting out with binary exploitation, then that someone should exclusively use GDB till the fundamentals of exploitation done are understood.

Basic Setup

GDB is usually pre installed on any linux machine. But if it isn't, it can be installed using the default package manager (Like apt or pacman). There are some extensions for GDB that make it a much easier tool to operate with. You can use any number of them to make it to your liking, but in the following section, only some extensions are used and explained.

2.2 radare2

Radare is a tool which helps in understanding the binary, decompiling it and allowing to modify commands on the fly. It is considered one of the most difficult tools to master, so much so, that they themselves put Figure 6 on their website. I think this can be the Maya of exploiation tools.

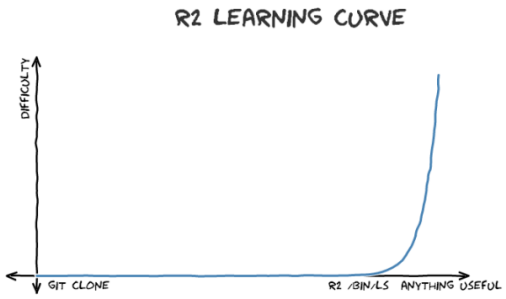

Figure 2.1: Radare Learning Curve

Chapter 3

Encryption and Decryption

3.1 Overview

Cryptography is traditionally associated with opening a message with a key, which is only known by the parties in the communication and no once else. This was true at that time and in some sense still is, but nowadays any form of data (or information) is protected with a cipher which helps in its secure transmission. Hence the definition has changed over the years.

Definition 1.1: Cryptography

It is the practice and study of techniques for secure communication in the presence of third parties called adversaries

From the above definition, it can clearly be seen how much it has changed over the years. This should be kept in mind when studying cryptography.

3.1.1 Uses

There are many examples where cryptography is used. It is almost as if every technology that exists wants to protect their information.

Example 1.1

FFor all types of communication via web, cryptography is used.

- Web Traffic : HTTPS Protocol
- Wireless Traffic
  - WiFi Traffic : 802.11 a/b/g/n
  - Telephone Signal : GSM

Example 1.2

EEFS file encryption and Truecrypt are used for protection of files.

Example 1.3

For content protection (DVDs and CDs with copyright media) these technologies are used:

- CSS: Content Scrambling System (Easily broken)
- AACS: Advanced Access Content System (Not that easy)

These are some of the examples of how cryptography is used everywhere. Hence understanding them is very important, and so is knowing how vulnerable they are. This can be the difference between building a robust system with good security and one which is easily exploited.

3.2 Historical Ciphers

In this chapter, historical ciphers will be explored. This is the time period where technology and digital communication were not prevalent, and hence these will be a product of the time when they were developed.

3.2.1 Substitution Cipher

This is one of the simplest ciphers that exists. It uses a 26 byte key which is a map from set of alphabets to itself.

Definition 2.1

Mathemtical definition of the substitution cipher is as follows:

- G : Set of ASCII Characters
- $f : G \to G$, f is bijective, such that
- $f(\alpha) = \beta$

$f \to$ Substitution Cipher

As it can be seen, the substitution cipher is a hash table, which stores the mapping of every character.

Code

This is a sample code on how the substitution cipher encodes. In this only small alphabets are taken into consideration.

```
import random

# A predefined hash table (Key)
key_table = random.sample(range(26),26)

m = 'known_message_here'

c = ''
for i in m:
    c += chr(key_table[ord(i)
             - ord('a')] + ord('a'))

print(c) # Cipher
```

3.2.2 Rot - X

There is a keyless version of the substitution cipher which is an even simpler version of the original cipher. In this, instead of a random key table, here it is a rotation by X (an integer), around which the alphabets are wrapped.

Definition 2.2

Mathemtical definition of the Rot-X cipher is as follows :

- G : Set of ASCII Characters
- $f : G \to G$,
- $f(\alpha) = (\alpha + X)\ mod\ 26$

$f \to$ Rot-X Cipher

Code

A sample code is written for it, even though it is not required. But well here it is:

```
X = 13 # Rot 13 is the most popular
m = 'my_super_secret_message'

c = ''
for i in m:
    c += chr((ord(i) - ord('a')
              + X) % 26 + ord('a'))

print(c) # Cipher
```

3.2.3 Caesar Cipher

A Rot-3 cipher is known as a Caesar cipher. This cipher does not even have a key, and hence is technically not a cipher, but came before the generalization Rot-X and hence was widely used at some point in time and still is used as an introduction to ciphers in CTFs.

Breaking the Cipher

This is one of the easiest ciphers to break. It is done using a method known as frequency analysis.

Definition 2.3

requency analysis is the study of letters or groups of letters contained in a ciphertext in an attempt to partially reveal the message.

So using frequency analysis, the frequency of letters is done on the ciphertext and is mapped to what is standard for the English language and hence the plaintext known by replacing the letters with their maps.

Here is an example to clear the doubts :

Example 2.1

Let us take ciphertext (c) be :

```
LIVITCSWPIYVEWHEVSRIQMXLEYVEOIEWHRXEXIPFEMVEWHKVSTYLXZIXLIKI
```

```
IXPIJVSZEYPERRGERIMWQLMGLMXQERIWGPSRIHMXQEREKIETXMJTPRGEVEKE
ITREWHEXXLEXXMZITWAWSQWXSWEXTVEPMRXRSJGSTVRIEYVIEXCVMUIMWERG
MIWXMJMGCSMWXSJOMIQXLIVIQIVIXQSVSTWHKPEGARCSXRWIEVSWIIBXVIZM
XFSJXLIKEGAEWHEPSWYSWIWIEVXLISXLIVXLIRGEPIRQIVIIBGIIHMWYPFLE
VHEWHYPSRRFQMXLEPPXLIECCIEVEWGISJKTVWMRLIHYSPHXLIQIMYLXSJXLI
MWRIGXQEROIVFVIZEVAEKPIEWHXEAMWYEPPXLMWYRMWXSGSWRMHIVEXMSWMG
STPHLEVHPFKPEZINTCMXIVJSVLMRSCMWMSWVIRCIGXMWYMX
```

I is the most common letter that appears here. It is known that e is the most common English alphabet, and hence I can be replaced with e. Similarly looking at the most common two letter sequence it is XL which can be replaced with in.

Going on like this, we can successfully find the substitution, which is as follows:

```
Hereupon Legrand arose, with a grave and stately air, and
brought me the beetle from a glass case in which it was
enclosed. It was a beautiful scarabaeus, and, at that time,
unknown to naturalists-of course a great prize in a scientific
point of view. There were two round black spots near one
extremity of the back, and a long one near the other. The
scales were exceedingly hard and glossy, with all the
appearance of burnished gold. The weight of the insect was
very remarkable, and, taking all things into consideration,
I could hardly blame Jupiter for his opinion respecting it.
```

For more in depth solution, go to : Frequency Analysis Wiki

Code

Code for the frequency analysis tool is taken from CodeDrome:

```
from operator import itemgetter
import json

def create_decryption_dictionary(plaintext_filepath,
    encrypted_filepath, dictionary_filepath):
    """
    Create an estimated mapping between encrypted letters
       and
    plaintext letters by comparing the frequencies in the
    plaintext and encrypted text.
    The dictionary is then saved as a JSON file.
    """

    sample_plaintext = _readfile(plaintext_filepath)
    encrypted_text = _readfile(encrypted_filepath)

    sample_plaintext_frequencies =
       _count_letter_frequencies(sample_plaintext)
    encrypted_text_frequencies = _count_letter_frequencies(
       encrypted_text)

    decryption_dict = {}
    for i in range(0, 26):
        decryption_dict[encrypted_text_frequencies[i][0]] =
            sample_plaintext_frequencies[i][0].lower()

    f = open(dictionary_filepath, "w")
    json.dump(decryption_dict, f)
    f.close()
```

```
def decrypt_file(encrypted_filepath, decrypted_filepath,
    dictionary_filepath):
    """
    Use the dictionary to decrypt the encrypted file
    and save the result.
    """
    encrypted_text = _readfile(encrypted_filepath)

    f = open(dictionary_filepath, "r")
    decryption_dict = json.load(f)
    f.close()

    decrypted_list = []

    for letter in encrypted_text:
        asciicode = ord(letter.upper())
        if asciicode >= 65 and asciicode <= 90:
            decrypted_list.append(decryption_dict[letter])

    decrypted_text = "".join(decrypted_list)

    f = open(decrypted_filepath, "w")
    f.write(decrypted_text)
    f.close()

def _count_letter_frequencies(text):
    """
    Create a dictionary of letters A-Z and count the
        frequency
    of each in the supplied text.
    Lower case letters are converted to upper case.
    All other characters are ignored.
    The returned data structure is a list as we need to
        sort it by frequency.
    """

    frequencies = {}

    for asciicode in range(65, 91):
        frequencies[chr(asciicode)] = 0

    for letter in text:
        asciicode = ord(letter.upper())
        if asciicode >= 65 and asciicode <= 90:
            frequencies[chr(asciicode)] += 1

    sorted_by_frequency = sorted(frequencies.items(), key =
        itemgetter(1), reverse=True)

    return sorted_by_frequency

def _readfile(path):

    f = open(path, "r")
    text = f.read()
    f.close()
    return text
```

Working

The code above initially makes a frequency analysis dictionary with the help of a language text (preferably large), after which it stores it in a JSON. Then that dictionary is used to decode an encoded ciphertext.

3.2.4 Vigenère Cipher

Vigenère Cipher is one of the more competent of the ciphers created by Blaise de Vigenère, who was a French cryptographer. In this cipher, the key is repeated till its size is equal to that of the message. After which it is added to message alphabet by alphabet.

Definition 2.4: Vigenere Cipher

Definition of Vigenère Cipher:

- Pad key till length(key) = length(message)
- ciphertext = key $+_{26}$ message

In this the key is padded to make it equal to the length of the message. Hence, the key length cannot be greater than that of the cipher (It can be, but won't affect the working of the cipher).

Example 2.2

Lets take an example of the cipher, to know how it works.

```
key = 'mykey'
message = 'plztransferthis'
len(key) != len(message)

padded_key = 'mykeymykeymykey'
message    = 'plztransferthis'
--------------------------------  + mod 26
ciphertext = 'bjjxpmlcjcdrrmq'
```

Code

Now that the working of the cipher is known, here is a code for the same:

```
key = 'mykey'
message = 'plztransferthis'

i=0
while len(key) <= len(message):
    key += key[i]
    i+=1

ciphertext = ''
for i,_ in enumerate(message):
    ciphertext += chr((ord(message[i]) + ord(key[i])
                       - 2 * ord('a')) % 26
                       + ord('a'))

print(ciphertext)
```

Breaking the Cipher

Start by assuming that the length of the key is known to be l. Hence, in the ciphertext every l^{th} character is coded by the same letter. Therefore, the ciphertext formed from this is a Rot-X.

This can be solved using frequency analysis. So the X is now known, the character representation of that X is a letter of the key. Do this for every l letters till the complete key is found, after which the cipher can be decoded.

In this, an assumption was made that the length of the key was known, which is not the case. So, start with $l = 1$, till a comprehendible message is found.

Code

This is the code that solves the Vigenère cipher, which is taken from the alpha-k911 github.

```
import argparse
from peakutils.peak import indexes

def encrypt():
    a = ""

def guessKey(ciphertext=None,verbose=False):
    a = ciphertext
    b = ciphertext
    freq = []
    for i in range(0, len(a) - 1):
        b = " " + b[:-1]
        c = 0
        for j in range(i, len(a)):
            if b[j] == a[j]:
                c += 1
        freq.append(c)
    peaks = indexes(freq, thres=0.678)
    possible_keys = []
    possible_keys.append(peaks[1] - peaks[0])
    key = possible_keys[0]

    # cipher
    j = 0
    ciphers = []
    p = ""
    while (j < key):
        i = j
        p = ""
        while (i < len(a)):
            p = p + a[i]
            i = i + key
        j += 1
        ciphers.append(p)

    # frequency
    ciphers_freq = []
    for i in range(0, len(ciphers)):
        arr = [0] * 26
        p = ciphers[i]
        j = 0
        while (j < len(p)):
            pos = (ord(p[j]) - 13) % 26
            arr[pos] += 1
            j += 1
        ciphers_freq.append(arr)

    # map of fequency
    ind = []
    for i in range(0, 26):
        ind.append(i)

    def shift_left(f, t1):
        i = 0
        while i < f:
```

```
            t2 = t1
            t1 = t2[1:len(t2)] + [t2[0]]
            i += 1
        return t1

    j = 0
    char_freq = [0.08, 0.02, 0.03, 0.04, 0.13, 0.02, 0.02,
        0.06, 0.07, 0.0, 0.01, 0.04, 0.02, 0.07, 0.08, 0.02,
         0.0,
                 0.06, 0.06, 0.09, 0.03, 0.01, 0.02, 0.0,
                    0.02, 0.0]

    freq_sums = []

    def get_key(ciphers_freq, key_len):
        guessed_key = ""
        for l in range(0, key_len):
            freq_sums = []
            for i in range(0, 26):
                shift = shift_left(i, ciphers_freq[l])
                sum = 0
                for k in range(0, 26):
                    sum += shift[k] * char_freq[k]
                freq_sums.append(sum)
            freq_max = zip(freq_sums, ind)
            freq_max = list(freq_max)
            freq_max = sorted(freq_max, reverse=True)
            guessed_key += chr(65 + freq_max[0][1])
        return guessed_key

    print("[+] Guessed Key: " + get_key(ciphers_freq, key))
    return get_key(ciphers_freq, key)


if __name__ == "__main__":
    parser.add_argument("action",help="Action to perform on
        the input [encrypt/guesskey/decrypt] [e/g/d]")
    parser.add_argument("input",help="Input to be processed
        ")
    parser.add_argument("-v", "--verbose",help="print debug
        info",action="store_true")
    args = parser.parse_args()

    if args.action.lower() == "encrypt" or args.action.
       lower() == "e":
        pass
    elif args.action.lower() == "decrypt" or args.action.
       lower() == "d":
        pass
    elif args.action.lower() == "guesskey" or args.action.
       lower() == "g" or args.action.lower() == "guess":
        if len(args.input) < 500:
        guessKey(args.input)
```

3.2.5 Rotor Machines

During the renaissance, motors were name of the game. So naturally, people wanted to use it in a cipher scheme. Due to that a rotor machine cipher was formed.

One of the earlier and simpler rotor machine was the Hebern machine. It has a single rotor and hence encodes a simple 'rotating' substitution table.

Example 2.3

o the working of a Hebern machine is explained here. The machine encodes a substitution cipher, whose table moves every time a letter is typed.

```
N                 V                 .
K                 N                 V
P                 K                 N
T  Letter Typed   P  Letter Typed   K
O  ------------>  T  ------------>  P
.                 O                 T
.                 .                 O
.                 .                 .
V                 .                 .

Hence, if m = 'CCC',
then c = 'PKN'
```

This is more complex than the standard substitution cipher as same character is encoded to different character according to position of the rotor. Therefore the definition of a Hebern machine is as follows:

Definition 2.5: Hebern Machine

The definition of the Hebern machine, therefore is:

- G : Set of ASCII Characters
- $f : G \to G$, f is bijective, such that
- $f(\alpha + pos(\alpha)) = \beta$;
- $pos(\alpha) \to$ Index of α in the plaintext

Code

Now that the definition of the Hebern machine is known, this is the code:

```
import random

# A predefined hash table (Key)
key_table = random.sample(range(26),26)

m = 'known_message_here'

c = ''
for i,_ in enumerate(m):
    c += chr(key_table[(ord(m[i])
             - ord('a') + i)%26] + ord('a'))

print(c) # Cipher
```

It seems, as if there is no difference between the code of a substitution cipher and this one, but there is. In this the position of the letter is taken into consideration, thus making it a Hebern machine.

3.2.6 Enigma Machine

One of the most popular rotor machine. This was used by the Germans in the World War 2 for intra-communication. This machine had 3-5 rotors in it which would rotate at different speeds and would thus generate a very complicated (for that time) cipher text which was hard to break.

This machine was reverse engineered and decrypted by a British mathematician Alan Turing, who is very famously known for the Turing machine and the Church-Turing hypothesis.

Code

This is a sample code for Enigma machine. If a more detailed and better understanding for the Enigma code is wanted, it can be found on torognes github.

```
import random

# A predefined hash table (Key)
key_table = random.sample(range(26),26)
n = 3 #No of Rotors
m = 'known_message_here'

def rotor_speed(n):
    '''
    The function that decides the speed of each rotor
    '''
    return int(n)

c = ''
for i,_ in enumerate(m):
    c_i = m[i] # Temp location to store ith value
    for j in range(n):
        c_i = chr(key_table[(ord(c_i)
            - ord('a') + rotor_speed(j)*i)
            %26] + ord('a'))
    c += c_i

print(c) # Cipher
```

Chapter 4

CryptoGraphy

4.1 Ciphers and Secrecy

Definition 1.1: Cipher

A Cipher defined over ($\mathcal{K}$ (Key Space), $\mathcal{M}$ (Message Space), $\mathcal{C}$ (Cipher Space)) is a pair of efficient algorithms (E, D), Encryption algorithm $E : \mathcal{K} \times \mathcal{M} \to \mathcal{C}$ and Decryption algorithm $E : \mathcal{K} \times \mathcal{C} \to \mathcal{M}$ such that $D(E(\mathcal{C})) = \mathcal{C}$.

4.1.1 One Time Pad

Definition 1.2: Perfect Secrecy

A Cipher has perfect secrecy iff $\forall (m_1, m_2) \in \mathcal{M},\ len(m_1) = len(m_2),\ P[E(k, m_1) = c] = P[E(k, m_2) = c],\ k \in \mathcal{K}$picked uniformly, $\forall c \in \mathcal{C}$. (i.e. No Cipher Text only attack exists, ciphertext yeild no information about the plain text).

One-Time pad, i.e. a XOR with a Key is a way to get perfect secrecy. However, here the key length has to be the same as the message length, so it's not very useful (since if you can share the key, just use the same means to share the message). It can be proven that **any keyspace smaller than message space** cannnot obtain perfect secrecy.

4.1.2 Pseudo Random Generator

Now we try to get a more practical cipher which has a smaller key space. (We can still use the pads, but the key has to be smaller, so we will generate a Larger key from a smaller key, then XOR).

Definition 1.3: Pseudo Random Generator

A Pseudo Random Generator is a deterministic algorithm which maps a SEED, which is a binary string of length K, to a much longer binary string of length N, but the function is not predictable. i.e.

$$PRG(S) = R,\ R \in \{0,1\}^n, S \in \{0,1\}^k \tag{4.1}$$

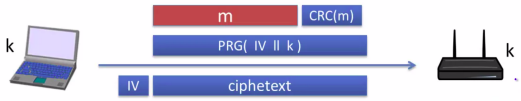

Figure 4.1: WEP Protocol

What it means to be predictable is that given the first i bits of the generated string, the i + 1 bit can be determined with probability $\geq 1/2 + \epsilon$. Practically ϵ is considered to be of the order $1/2^{80}$ (But $1/2^{30}$ is not, since it's likely to happen over 1GB of data). Theoretically, it's unpredictable when the value of ϵ falls off faster than $1/poly(\lambda)$.

4.1.3 Attacks on the Ciphers

Never use Two Time Pad

If the same key is used more than once, the messages can be XORed and then frequency analysis can yeild info on the messages. Therefore reusing a key in two different streams will be insecure.

MS-PPTP protocol in Windows NT used the same key from Server to Client and Client to Server. Though all the messages from Client were one stream, continuing on the Pseudo Random Generator, and so were all from the Server. But two streams used the same key, and that failed.

WEP protocol: has a lot of errors. IV was a counter (24 bits), and that was concatenated with 104 bit long-term key. After 2^24 frames, the IV will cycle, so it's like a 2-time pad. Since the keys only have a counter IV changing every frame, all keys are related, so Due to Shamir, After a 1000000 frames, we can recover the frames, and today even in about 30000 frames. The whole stream should have been viewed as a single stream, that would have worked better.

We should also not use this for Disk Encryption, since small changes to the file will change only a few bits. So the before and after edit files are encrypted with same key.

Integrity Violation

One Time Pad is malleable, i.e. We can change bits even without encrypting and decrypting. Eg. if we intercept a mail starting with "From: Bob", without knowing the key, we can xor and make it "From: Eve". So the message can be changed.

4.1.4 Real World Stream Cipher

RC4

Used in HTTPS and WEP. Weaknesses:

- Bias in initial output Pr[2nd Byte] = 2/256
- Pr[(0, 0)] = $1/256^2 + 1/256^3$.
- If keys are related, it's possible to recover secret.

CSS

Linear Feedback Shift Register (LFSR): In every clock cycle, the registers shift by 1, and some bytes are called tap registers, the XOR sum of which gives the results.

CSS has 2 Linear Feedback Shift Register, a 17-bit and a 25-bit LFSR. The Key is 5 bytes. First 2 bytes of key is put into 17-bit LFSR, next 3 bytes in 25-bit LFSR.

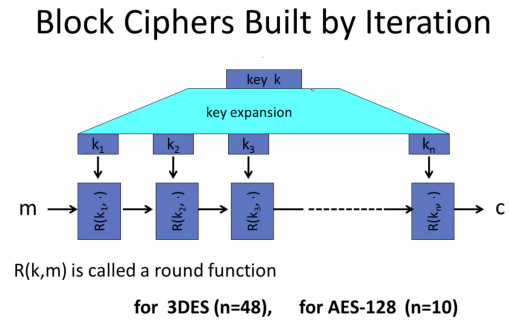

Figure 4.2: WEP Protocol

4.1.5 What is a Secure Cipher

Since Pseudo Random Generators are deterministic algorithms, and they output numbers uniformly in the space {0, 1} in the space .

$$Advantage_{PRG}[A, G] := |Pr[A(G(k)) = 1] - Pr[A(k) = 1]| \tag{4.2}$$

where k is a random seed, G is the

Salsa 20

XOR encryptions

4.2 Block Ciphers

4.2.1 Definitions, DES and AES

Block ciphers take exactly n-bits of input and map them to exactly n-bits of output. Some Examples are:

- AES, key = 168 bits, n = 64 bits
- 3DES, key = 128/192/256 bits, n = 128 bits

Block Ciphers are considerably slower than stream ciphers.

Definition 2.1: Pseudo Random Function

A Pseudo Random Function (PRF) defined over (K, X, Y) is $F : K \times X \to Y$, such that there exists an efficient algorithm to evaluate F(k, x).

Definition 2.2: Pseudo Random Permutation

A Pseudo Random Permutation (PRP) defined over (K, X) is $F : K \times X \to X$, such that

- There exists an efficient algorithm to evaluate F(k, x).
- There exists function E(k, .) that is one-to-one. (i.e. given key, message to cipher is bijective)
- There exists an efficient inversion algorithm D(k, y).

The consistancy constraint must obviously hold $D(k, F(k, x)) = x$.

A Pseudo Random Permutation is a Block Cipher, the terms might be used interchangably in different contexts.

4.2.2 Data Encryption Standard (DES)

56 bit key size, 64 bit block length. Widely used, but fell to complete search of the key.

For any functions $f_1, f_2, \ldots, f_d : \{0,1\}^n \to \{0,1\}^n$.

4.3 Message Authentication Codes

The Goal is to maintain Integrity not Confidentiality.

Definition 3.1: Message Authentication Codes

Message Authentication Codes (MAC) $I = (S, V)$, defined over keyspace $\mathcal{K}$, message space $\mathcal{M}$, and tag space $\mathcal{T}$ is a pair of algorithms:

- $S(k, m) \to t \in \mathcal{T}$ outputs a tag.
- $V(k, m, t) \to \{true, false\}$ outputs a tag.

Such that $V(k, m, S(k, m)) = true \; \forall (k, m)$.

We shall define the goal of the attacker to be **Existential Forgery**, i.e. if we allow the attacker to sample several message tag pairs on messages of his choice (m_i, t_i) $i = 1, 2, \ldots, q$, and we ask the attacker to produce a new message, tag pair not in the set of queries $(m', t') \notin \{(m_i, t_i) \; i = 1, 2, \ldots, q \}$ such that $V(k, m, t) = true$, the advantage (i.e. the probability of successfully outputing a key-value pair) is negligible.

Theorem 3.1: Advantage against MAC

Given a MAC I_F based on a Pseudo Random Function which outputs a tag of length |Y| being attacked by an adversary A is always less than that of it's Pseudo Random Function F being attacked by adversary B summed with the inverse length of the tag.

$$Adv_{MAC}[A, I_F] \leq Adv_{PRF}[B, F] + 1/|Y|$$

Chapter 5

Binary Exploits

Introduction

Binary exploitation is the process of subverting a compiled application t it violates some trust boundary in a way that is advantageous to you, the attacker.The exploits are explained breifly and then given case studies for it to be understood better.

5.1 Buffer Overflow

A buffer overflow occurs when a piece of data overflows the storage space given to it. In these types of exploits, usually stack smashing occurs which changes value of non intended variables and helps in changing the flow of the program in some way.

5.1.1 Stack

One of the most important component to understand for binary exploitation and by extention buffer overflow is the stack. It is the location where all the variables and

Chapter 6

Systems and Exploitation

Introduction

The following chapter is an introduction to case studies of well knowns exploits, intoducing mobile systems and gaming consoles (The easiest to hack, for their exploitation is mainstream). This is an introduction to the basics of actual life exploiting and will go into details on how the exploits were used, how to install them, their working and also how to replicate it. Specific systems will be covered in a one to one case based scenario.

Terminology

There are some very common terms encountered while browsing through this piece and also while refering to external resources specific to console exploitation (As I told you, it *was* (is) very mainstream). Exploitation of systems with only a software is known as a softmod, whereas one done by physically modifying your hardware is known as hardmod. There are obvious advantages to both, a softmod is usually protected from in the next firmware update whereas hardmods are usually based on hardware vulnerabilities (But not always) and thus are harder to protect against after product is in the hands of the public.

6.1 PSP

One of the most exploited (*Hacked*) system out there. It is the one that even I have reaped the benifits of, by doing *legal* things, of course. It is also the one which I remember following to check out if the next version was exploitable, was it safe, did the risks outweigh the positives, etc. So here, I will explain how the PSP was exploited, why was it *easy* and also replication.

6.2 Android

For android devices, it is not technically exploitaion as Android does not dissalow it, it is the OEMs which refuse access to root for customers. So rooting *technically*, if the OEM (like Xiaomi) allows it, is not exploitaiton, but it does allow access to the complete system, so I am going to proceed with calling it exploitaiton of Android Rooting on android differs from phone to phone (Or more like OEMs to OEMs), and can be trivially easy or a fairly complicated process. But the general flow of work goes like this: Normal Device → Unlocking Bootloader → Installing a custom recovery → Installing a root manager (Like magisk or SuperSU) → Reboot → Rooted Android

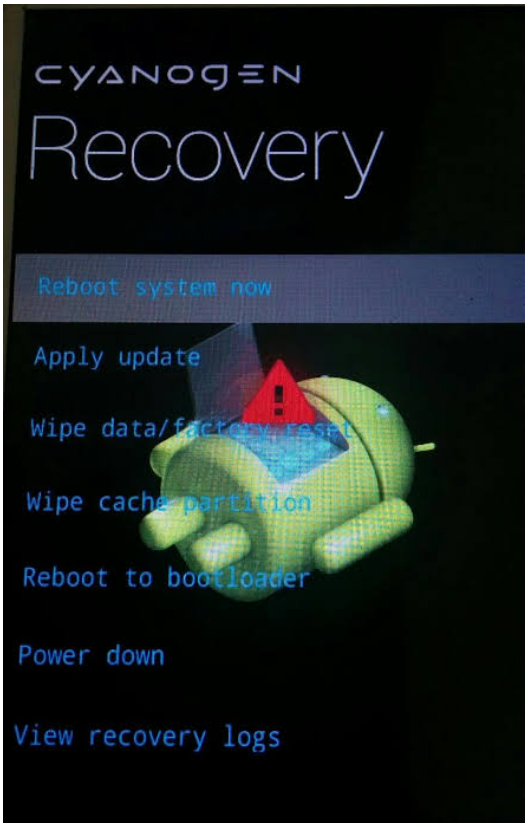

Figure 6.1: Recovery

6.2.1 Bootloader

The bootloader in an android device is hidden and is not accessible to the normal user. It is locked by the OEM, so in order to access it, it has to be unlocked first. The method differs for every phone, and the details can be found on XDA Developers website[1]. Bootloader is explained in detail in the Operating System section, but a brief understanding is that it is piece of code that points which OS (More specifically the kernel[2]) to load. So the bootloader, after being unlocked will allow us to load a custom OS (Known as recovery). This is a standalone OS which allows us to *flash* a zip to the main partition.

Unlocking the Bootloader

TO DO

6.2.2 Recovery

Recovery is an OS, which has a single purpose of allowing recovery. What it means is that it is a small OS which helps in fixing your main OS by providing tools to fix it (Kind of like the live linux systems). It is a minimal shell which allows for some fastboot and posix commands to run. It is most usefull in flashing zips which are either ROMs, custom kernels, or root binaries.The vanilla recovery that comes with the device does not allow for any such modifications, and this is where a custom recovery comes into picture. It allows for any modification and installation from within the recovery itself.

[1] xda-developers.com
[2] Learn More

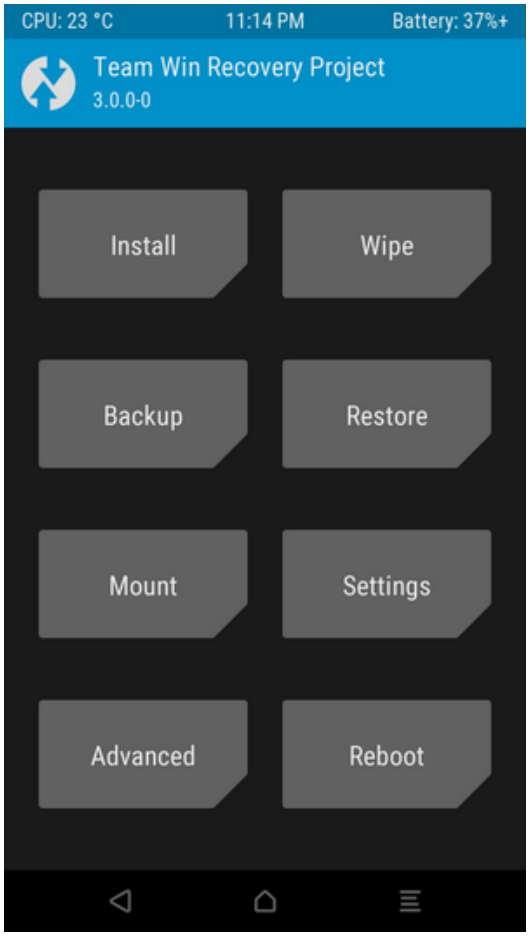

Figure 6.2: TWRP

Custom Recovery

Custom recovery is the software which performs function of a recovery but allows for remote application i.e no need for another device to give commands to the system for it to work. One of the earlier custom recovery software was ClockworkMod which was one of the primary custom recovery for Android versions till 4.0, after which TWRP started to take over as the preferred recovery software.

6.2.3 ADB and Fastboot

ADB and Fastboot are utilities

Andoid Debugging Bridge

Android Debugging Bridge (known as ADB) is a tool which allow for communication with an android device via USB. It is a shell with basic commands that allow a device to execute *debug* commands. It is a basic version of a UNIX shell.