Capture the Flag

50.042 Foundations of Cybersecurity

Reverse gnireenignE
Abinaya 1002941 | Glenn 1003118 | Kenneth 1002691 | Sumedha 1002876

Keyword: Reverse Engineering, Substitution, Transposition

Abstract: The goal of this challenge is to have fun and explore the concept of Reverse Engineering. In addition, we added several twists to the traditional shift ciphers and transposition ciphers that added on to what we learnt in class.

1 Challenge Description

For students:
In this challenge, we introduce Reverse Engineering while augmenting certain materials taught in class. There are 3 encrypted files given to you. One of them has contains the encryption scheme. This is all the information you need. Good luck.

For Prof:
Files to send to the groups: Students folder
Hints: Hints folder
Encryption: Encryption folder
Decryption: Decryption folder
Plaintext will be provided

2 Introduction

The overall high level encryption scheme is as follows:

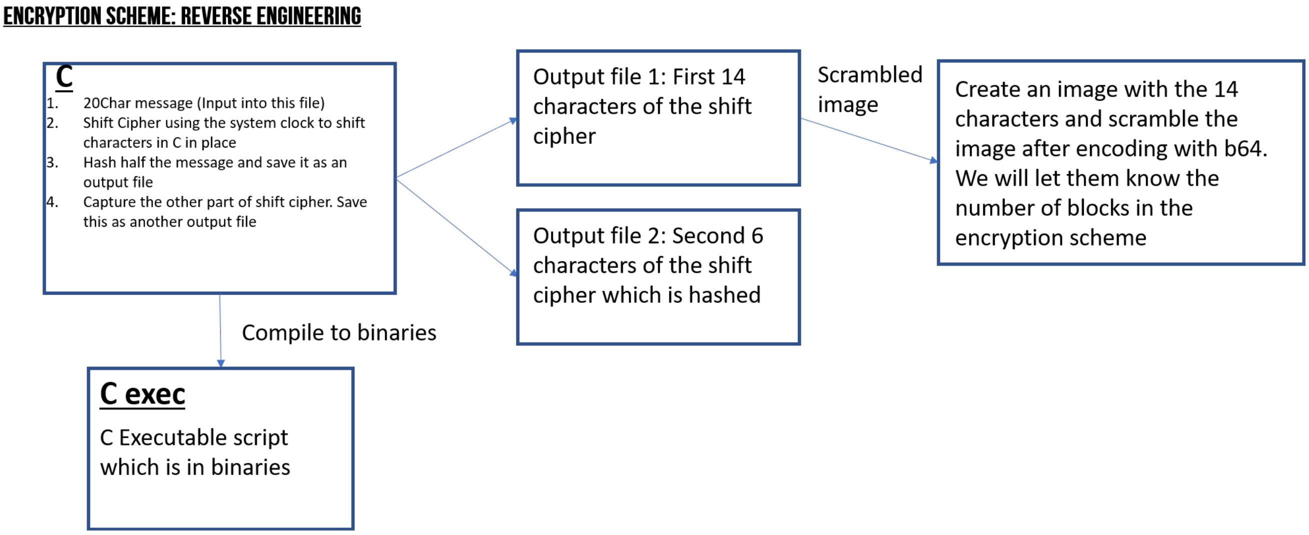

ENCRYPTION SCHEME: REVERSE ENGINEERING
C
1. 20Char message (Input into this file)
2. Shift Cipher using the system clock to shift characters in C in place
3. Hash half the message and save it as an output file
4. Capture the other part of shift cipher. Save this as another output file
Output file 1: First 14 characters of the shift cipher
Output file 2: Second 6 characters of the shift cipher which is hashed
Scrambled image
Create an image with the 14 characters and scramble the image after encoding with b64. We will let them know the number of blocks in the encryption scheme
Compile to binaries
C exec
C Executable script which is in binaries

Figure 1: Encryption high level

The overall high level decryption scheme is as follow

DECRYPTION SCHEME: REVERSE ENGINEERING

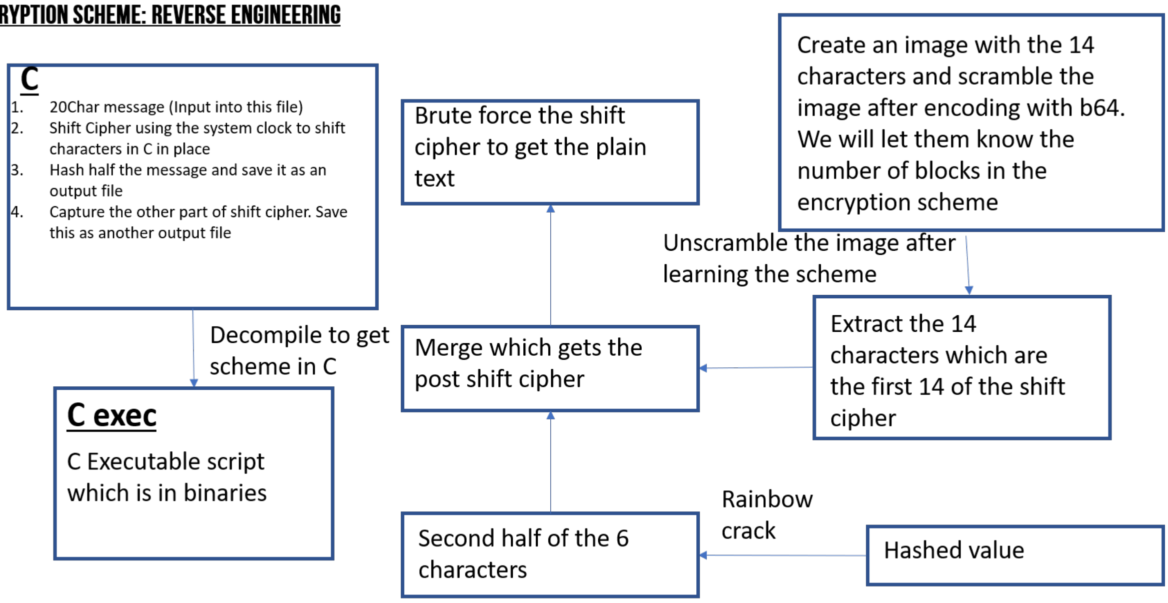

C
1. 20Char message (Input into this file)
2. Shift Cipher using the system clock to shift characters in C in place
3. Hash half the message and save it as an output file
4. Capture the other part of shift cipher. Save this as another output file
Decompile to get scheme in C
C exec
C Executable script which is in binaries
Brute force the shift cipher to get the plain text
Create an image with the 14 characters and scramble the image after encoding with b64. We will let them know the number of blocks in the encryption scheme
Unscramble the image after learning the scheme
Extract the 14 characters which are the first 14 of the shift cipher
Merge which gets the post shift cipher
Second half of the 6 characters
Rainbow crack
Hashed value

Figure 2: Decryption high level

3 Encryption

The description of the overall encryption scheme is as follows:

1. C code takes in an input which is the plain text file. The plain text file has the string **P1=** "**En 6!n3_f0rw0rdlSTDu**"
2. Generate the key using the current date including seconds. Result is **YYYYMMDDHHmmSSHHmmSS = 20191202190452190452**
3. **Optional** (to increase difficulty if needed) - We scramble the key by a shift (This shift is known to the attacker) so we have DDHHMMSSYYYYm- mYYYYMM

4. Based on these numbers we modified and augmented the Caesar’s cipher implementation to shift each letter accordingly. We end up with a string **P2 = “ Gn 6!p3_g0ra0teRSXlw “**

5. Split String p2 into 2 strings, p3 with 14 and P4 with 6 characters each,
   i. P3 will be screenshot to a PNG image and scrambled. (3.3)
   ii. We then hash the string P4. (3.2)

6. The attacker will get the executable, the scrambled text file of P3 and the hash of P4

3.1 Augmented shift cipher

The scheme works by first generating the current timestamp. An example would be 20191128093011 which have 14 characters. However we want 20 characters and repeating hour, minute and seconds again would gives us 20191202190452190452.

We then use this to shift the characters again. The interesting thing about C is that it does not handle shifts of white spaces, punctuation or integers.
A simplified example is as follows:

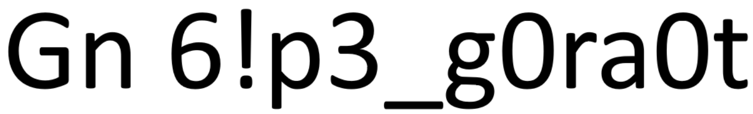
Position	0	1	2	3	4	5
Timestamp	2	0	1	9	1	1
Plain text	h	l	e	L	!	o
Cipher text	j	m	f	U	!	t

3.2 Partial Hashing

After the shift phase, we take the first 6 characters and hash it with C's MD5 hashing. This includes lowercase, uppercase and numbers. The result of the hashing function will be given to the attacker. (encrypt2.txt)

3.3 Image scrambling

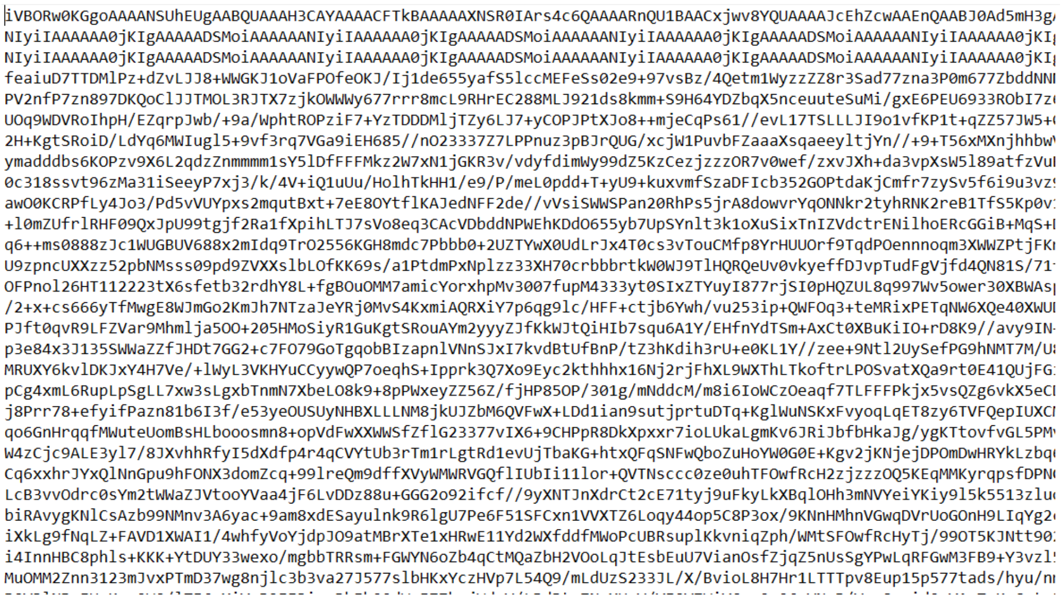
After the shift phase, we take the second half of 14 characters and create a black and white image from it.



Gn 6!p3_g0ra0t

Figure 3: Image of hash

We use Python to encrypt the image using b64 encoding. We also scramble the order randomly after being inspired by the present architecture. The result post scrambling is given to the attacker.

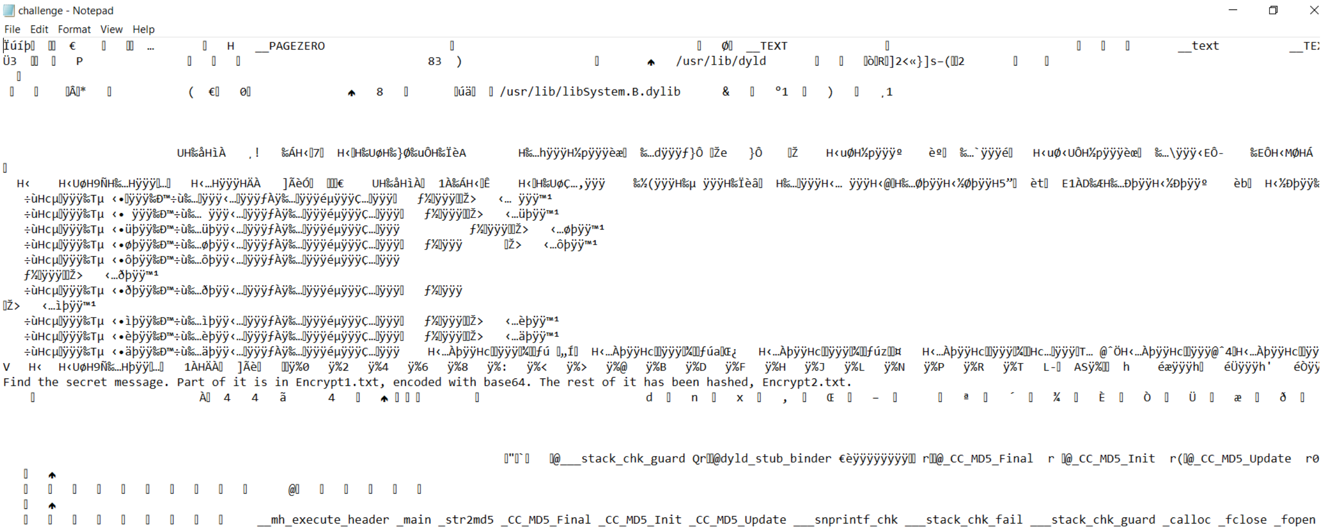


```
iVBORw0KGgoAAAANSUhEUgAABQUAAAH3CAYAAAACFTkBAAAAAXNSR0IArs4c6QAAAARnQU1BAACxjwv8YQUAAAAJcEhZcwAAEnQAABJ0Ad5mH3g
NIyiIAAAAAA0jKIgAAAAADSMoiAAAAAANIyiIAAAAAA0jKIgAAAAADSMoiAAAAAANIyiIAAAAAA0jKIgAAAAADSMoiAAAAAANIyiIAAAAAA0jKI
NIyiIAAAAAA0jKIgAAAAADSMoiAAAAAANIyiIAAAAAA0jKIgAAAAADSMoiAAAAAANIyiIAAAAAA0jKIgAAAAADSMoiAAAAAANIyiIAAAAAA0jKI
feaiuD7TTDMlPz+dZvLJJ8+WWGKJ1oVaFPOfeOKJ/Ij1de655yafS5lccMEFeSs02e9+97vsBz/4Qetm1WyzzZZ8r3Sad77zna3P0m677ZbddNN
PV2nfP7zn897DKQoClJJTMOL3RJTX7zjkOWWWy677rrr8mcL9RHrEC288MLJ921ds8kmm+S9H64YDZbqX5nceuuteSuMi/gxE6PEU6933RObI7z
UOq9WDVRoIhpH/EZqrpJwb/+9a/WphtROPziF7+YzTDDDMljTZy6LJ7+yCOPJPtXJo8++mjeCqPs61//evL17TSLLLJI9o1vfKP1t+qZZ57JW5+
2H+KgtSRoiD/LdYq6MWIugl5+9vf3rq7VGa9iEH685//nO23337Z7LPPnuz3pBJrQUG/xcjW1PuvbFZaaaXsqaeeyltjYn//+T56xMXnjhhbw
ymadddbs6KOPzv9X6L2qdzZnmmmm1sY5lDfFFFMkz2W7xN1jGKR3v/vdyfdimWy99dZ5KzCezjzzzOR7v0wef/zxvJXh+da3vpXsW5l89atfzVu
0c318ssvt96zMa31iSeeyP7xj3/k/4V+iQ1uUu/HolhTkHH1/e9/P/meL0pdd+T+yU9+kuxvmfSzaDFIcb352GOPtdaKjCmfr7zySv5f6i9u3vz
awO0KCRPfLy4Jo3/Pd5vVUYpxs2mqutBxt+7eE8OYtflKAJedNFF2de//vVsiSWWSPan20RhPs5jrA8dowvrYqONNkr2tyhRNK2reB1TfS5Kp0v
+l0mZUfrlRHF09QxJpU99tgjf2Ra1fXpihLTJ7sVo8eq3CAcVDbddNPWEhKDdO655yb7UpSYnlt3k1oXuSixTnIZVdctrENilhoERcGGiB+MqS+
q6++ms0888zJc1WUGBUV688x2mIdq9TrO2556KGH8mdc7Pbbb0+2UZTYwX0UdLrJx4T0cs3vTouCMfp8YrHUUOrf9TqdPOennnoqm3XWWZPtjFK
U9zpncUXXzz52pbNMsss09pd9ZVXXslbLOfKK69s/a1PtdmPxNplzz33XH70crbbbrtkW0WJ9TlHQRQeUv0vkyeffDJvpTudFgVjfd4QN81S/71
OFPnol26HT112223tX6sfetb32rdhY8L+fgBOuOMM7amicYorxhpMv3007fupM4333yt0SIxZTYuyI877rjSI0pHQZUL8q997Wv5ower30XBWAs
/2+x+cs666yTfMwgE8WJmGo2KmJh7NTzaJeYRj0MvS4KxmiAQRXiY7p6qg9lc/HFF+ctjb6Ywh/vu253ip+QWFOq3+teMRixPETqNW6XQe40XWU
PJft0qvR9LFZVar9Mhmlja5OO+205HMoSiyR1GuKgtSRouAYm2yyyZJfKkWJtQiHIb7squ6A1Y/EHfnYdTSm+AxCt0XBuKiIO+rD8K9//avy9IN
p3e84x3J135SWWaZZfJHDt7GG2+c7FO79GoTgqobBIzapnlVNnSJxI7kvdBtUfBnP/tZ3hKdih3rU+e0KL1Y//zee+9Ntl2UySefPG9hNMT7M/U
MRUXY6kvlDKJxY4H7Ve/+lWyL3VKHYuCCyywQP7oeqhS+Ipprk3Q7Xo9Eyc2kthhhx16Nj2rjFhXL9WXThLTkoftrLPOSvatXQa9rt0E41QUjFG
pCg4xmL6RupLpSgLL7xw3sLgxbTnmN7XbeLO8k9+8pPWxeyZZ56Z/fjHP85OP/301g/mNddcM/m8i6IoWCzOeaqf7TLFFFPkjx5vsQZg6vkX5eC
j8Prr78+efyifPazn81b6I3f/e53yeOUSUyNHBXLLLNM8jkUJZbM6QVFwX+LDd1ian9sutjprtuDTq+KglWuNSKxFvyoqLqET8zy6TVFQepIUXC
qo6GnHrqqfMWuteUomBsHLbooosmn8+opVdFwXXWWSfZflG23377vIX6+9CHPpR8DkXpxxr7ioLUkaLgmKv6JRiJbfbHkaJg/ygKTtovfvGL5PM
W4zCjc9ALE3yl7/8JXvhhRfyI5dXdfp4r4qCVYtUb3rTm1rLgtRd1evUjTbaKG+htxQFqSNFwQboZuHoYW0G0E+Kgv2jKNjejDPOmDwHRYkLzbq
Cq6xxhrJYxQlNnGpu9hFONX3domZcq+99lreQm9dffXVyWMWRVGQflIUbIi11lor+QVTNscc0ze0uhTFOwfRcH2zjzzzOQ5KEqMMKyrqpsfDPN
LcB3vvOdrc0sYm2tWWaZJVtooYVaa4jF6LvDDz88u+GGG2o92ifcf//9yXNTJnXdrCt2cE71tyj9uFkyLkXBqlOHh3mNVYeiYKiy9l5k5513zlu
biRAvygKNlCsAzb99NMnv3A6yac+9am8xdESayulnk9R6lgU7Pe6F51SFCxn1VVXTZ6Loqy44op5C8P3ox/9KNnHMhnVGwqDVrUoGOnH9LIqYg2
iXkLg9fNqLZ+FAVD1XWAI1/4whfyVoYjdpJO9atMBrXTe1xHRwE11Yd2WXfddfMWoPcUBRsuplKkvniqZph/WMtSFOwfRcHyTj/99OT5KJNtt90
i4InnHBC8phls+KKK+YtDUY33wexo/mgbbTRRsm+FGWYN6oZb4qCtMQaZbH2VOoLqJtEsbEuU7VianOsfZjqZ5nUsSgYPwLqRFGwM3FB9+Y3vzl
MuOMM2Znn3123mJvxPTmD37wg8njlc3b3va27J577slbHKxYczHVp7L54Q9/mLdUzS233JL/X/BvioL8H7Hr1LTTTpv8Eup15p577tads/hyu/n
```

Figure 4: Image Encryption with Python

3.4 Compiling the C code into an executable

We also compile the C code into an executable and this will be given to the attacker.



challenge - Notepad

```
__PAGEZERO    __TEXT    __text    __TE
83 )    /usr/lib/dyld
/usr/lib/libSystem.B.dylib
Find the secret message. Part of it is in Encrypt1.txt, encoded with base64. The rest of it has been hashed, Encrypt2.txt.
@___stack_chk_guard Qr@dyld_stub_binder €èÿÿÿÿÿÿÿ r@_CC_MD5_Final  r @_CC_MD5_Init  r(@_CC_MD5_Update  r0
__mh_execute_header _main _str2md5 _CC_MD5_Final _CC_MD5_Init _CC_MD5_Update ___snprintf_chk ___stack_chk_fail ___stack_chk_guard _calloc _fclose _fopen
```

Figure 5: Image Encryption with Python

4 Decryption

Essentially, attackers will first have to decompile the C code, understand the scheme by trying to make out the pseudocode provided by some of the tools. Subsequently they can proceed to solve the rest of the challenges. Our team decided to strike a balance between the difficulty of the RE and the rest of the schemes as suggested by Prof.

4.1 Learning the scheme – Decompiler

Groups will receive the hash value and the encryption file in .exe format. They are required to reverse engineer and decrypt the hash value to get the plain text.
Teams have to search online for valid RE software. One of them is Ghidra.
Install Ghidra with https://ghidra-sre.org/.
Once Ghidra has been installed, place .exe file into Ghidra. Next, run the CodeBrowser, which can be found in the Tool Chest or simply double clicking on the .exe file.

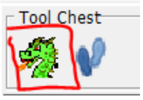
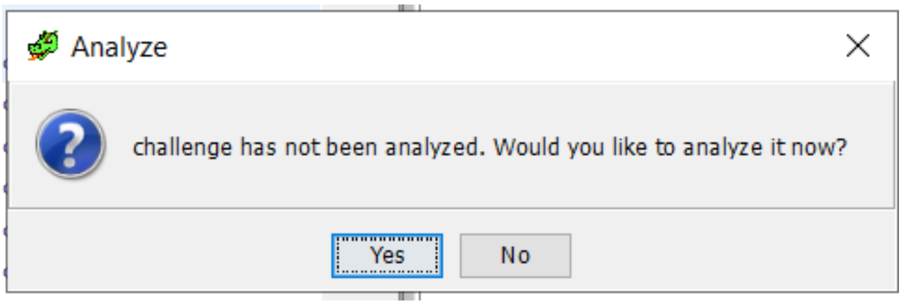


Figure 6: Tool chest

Once the .exe file has been opened, an Analyze prompt will appear, click ‘Yes´.



challenge has not been analyzed. Would you like to analyze it now?

Figure 7: analyze

We can filter out the main function as such

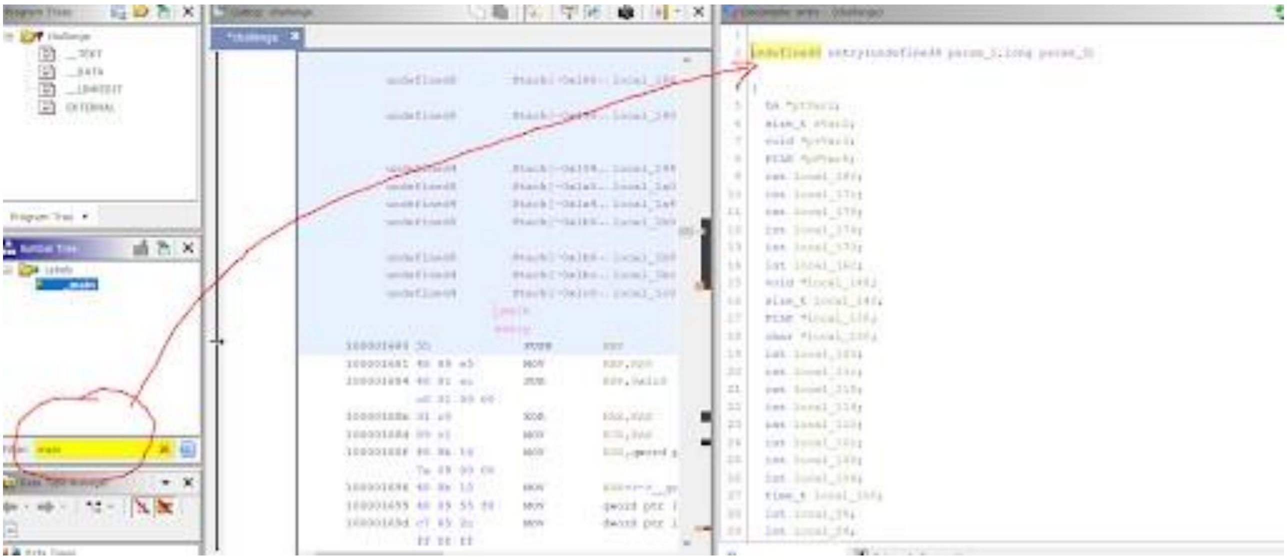
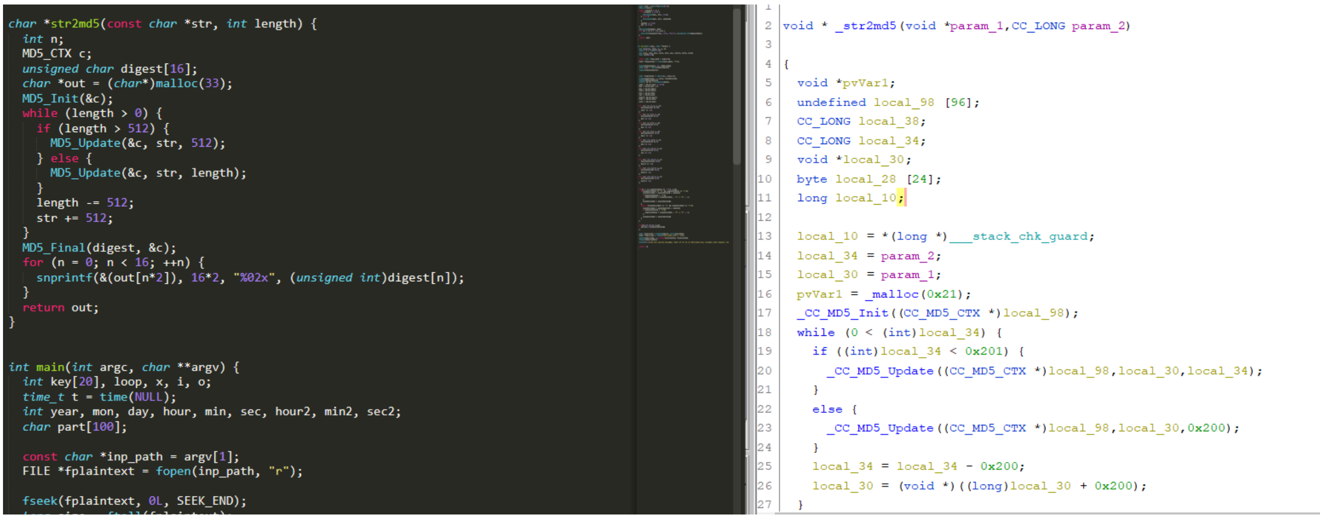


Figure 8: Filter for main function

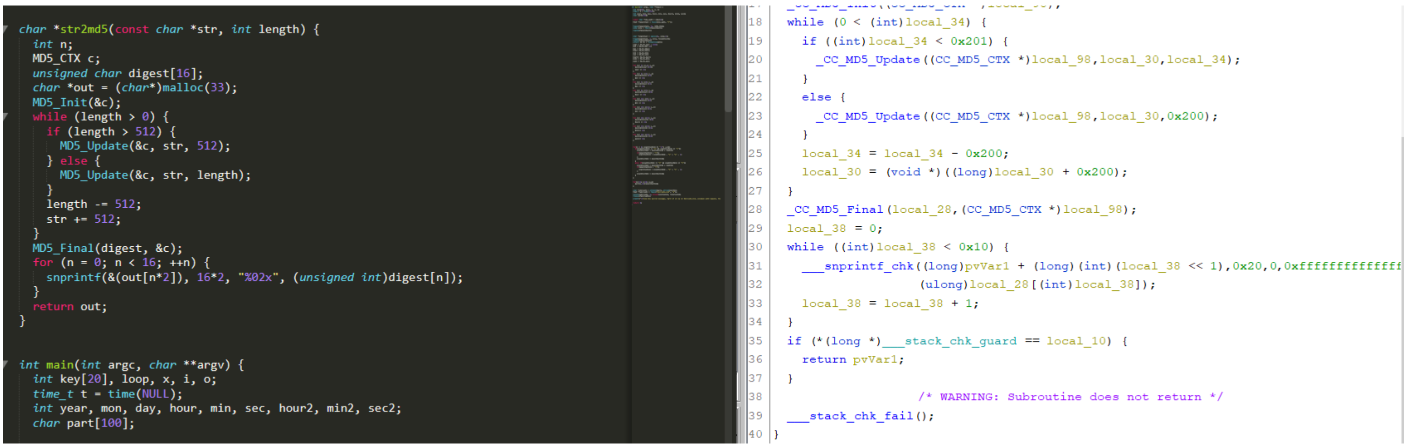
After this step, the rest of the C code can slowly be figured out



```c
char *str2md5(const char *str, int length) {
  int n;
  MD5_CTX c;
  unsigned char digest[16];
  char *out = (char*)malloc(33);
  MD5_Init(&c);
  while (length > 0) {
    if (length > 512) {
      MD5_Update(&c, str, 512);
    } else {
      MD5_Update(&c, str, length);
    }
    length -= 512;
    str += 512;
  }
  MD5_Final(digest, &c);
  for (n = 0; n < 16; ++n) {
    snprintf(&(out[n*2]), 16*2, "%02x", (unsigned int)digest[n]);
  }
  return out;
}


int main(int argc, char **argv) {
  int key[20], loop, x, i, o;
  time_t t = time(NULL);
  int year, mon, day, hour, min, sec, hour2, min2, sec2;
  char part[100];

  const char *inp_path = argv[1];
  FILE *fplaintext = fopen(inp_path, "r");

  fseek(fplaintext, 0L, SEEK_END);
```

```c

void * _str2md5(void *param_1,CC_LONG param_2)

{
  void *pvVar1;
  undefined local_98 [96];
  CC_LONG local_38;
  CC_LONG local_34;
  void *local_30;
  byte local_28 [24];
  long local_10;

  local_10 = *(long *)___stack_chk_guard;
  local_34 = param_2;
  local_30 = param_1;
  pvVar1 = _malloc(0x21);
  _CC_MD5_Init((CC_MD5_CTX *)local_98);
  while (0 < (int)local_34) {
    if ((int)local_34 < 0x201) {
      _CC_MD5_Update((CC_MD5_CTX *)local_98,local_30,local_34);
    }
    else {
      _CC_MD5_Update((CC_MD5_CTX *)local_98,local_30,0x200);
    }
    local_34 = local_34 - 0x200;
    local_30 = (void *)((long)local_30 + 0x200);
  }
```

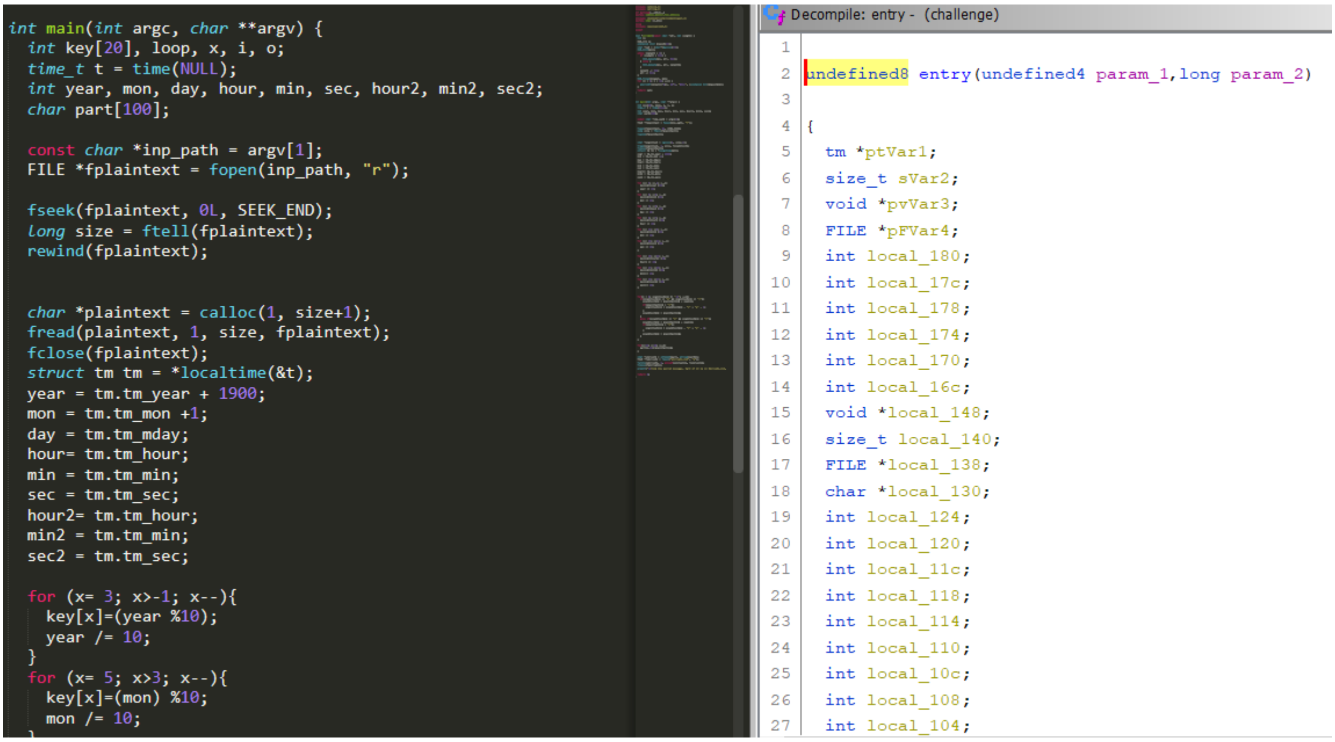
Figure 9: Decompile 1



```c
char *str2md5(const char *str, int length) {
  int n;
  MD5_CTX c;
  unsigned char digest[16];
  char *out = (char*)malloc(33);
  MD5_Init(&c);
  while (length > 0) {
    if (length > 512) {
      MD5_Update(&c, str, 512);
    } else {
      MD5_Update(&c, str, length);
    }
    length -= 512;
    str += 512;
  }
  MD5_Final(digest, &c);
  for (n = 0; n < 16; ++n) {
    snprintf(&(out[n*2]), 16*2, "%02x", (unsigned int)digest[n]);
  }
  return out;
}


int main(int argc, char **argv) {
  int key[20], loop, x, i, o;
  time_t t = time(NULL);
  int year, mon, day, hour, min, sec, hour2, min2, sec2;
  char part[100];
```

```c
  while (0 < (int)local_34) {
    if ((int)local_34 < 0x201) {
      _CC_MD5_Update((CC_MD5_CTX *)local_98,local_30,local_34);
    }
    else {
      _CC_MD5_Update((CC_MD5_CTX *)local_98,local_30,0x200);
    }
    local_34 = local_34 - 0x200;
    local_30 = (void *)((long)local_30 + 0x200);
  }
  _CC_MD5_Final(local_28,(CC_MD5_CTX *)local_98);
  local_38 = 0;
  while ((int)local_38 < 0x10) {
    ___snprintf_chk((long)pvVar1 + (long)(int)(local_38 << 1),0x20,0,0xffffffffffffff
                    (ulong)local_28[(int)local_38]);
    local_38 = local_38 + 1;
  }
  if (*(long *)___stack_chk_guard == local_10) {
    return pvVar1;
  }
                    /* WARNING: Subroutine does not return */
  ___stack_chk_fail();
}
```

Figure 10: Decompile 2

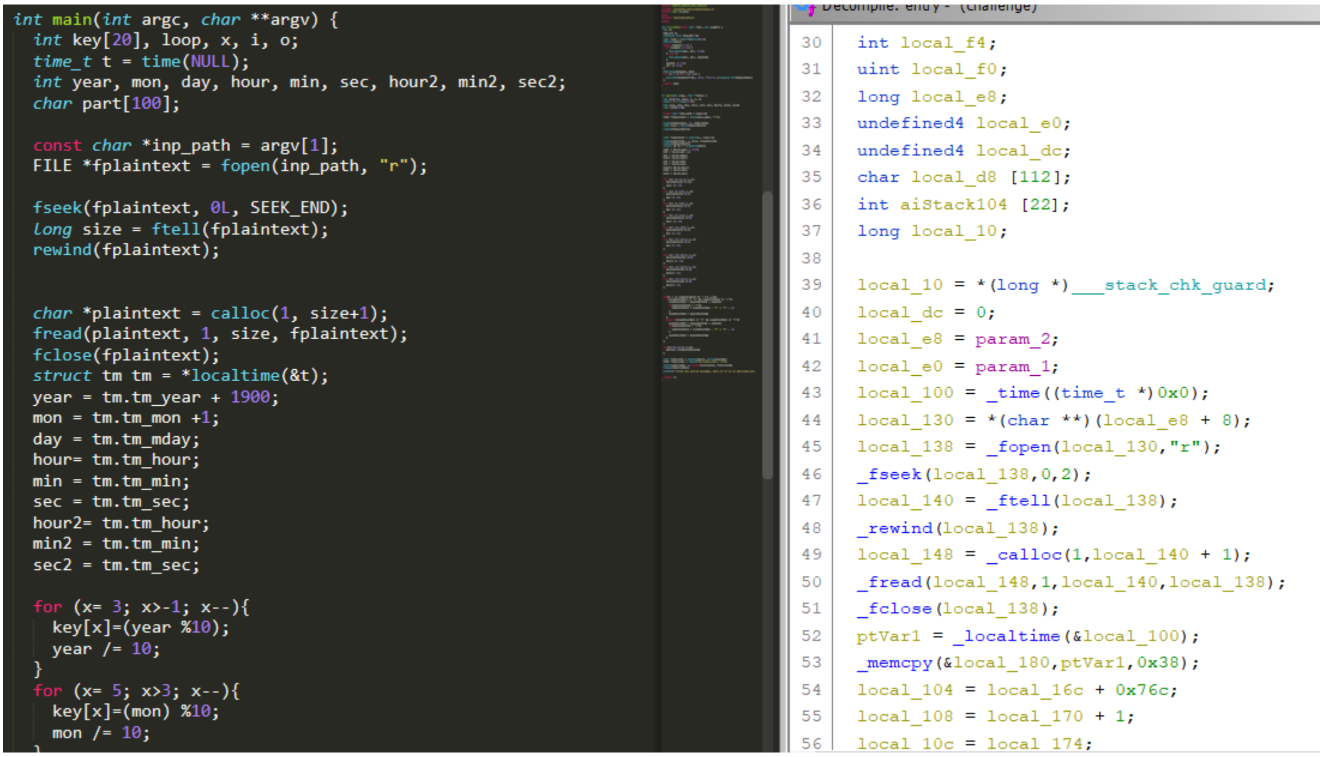


```c
int main(int argc, char **argv) {
  int key[20], loop, x, i, o;
  time_t t = time(NULL);
  int year, mon, day, hour, min, sec, hour2, min2, sec2;
  char part[100];

  const char *inp_path = argv[1];
  FILE *fplaintext = fopen(inp_path, "r");

  fseek(fplaintext, 0L, SEEK_END);
  long size = ftell(fplaintext);
  rewind(fplaintext);


  char *plaintext = calloc(1, size+1);
  fread(plaintext, 1, size, fplaintext);
  fclose(fplaintext);
  struct tm tm = *localtime(&t);
  year = tm.tm_year + 1900;
  mon = tm.tm_mon +1;
  day = tm.tm_mday;
  hour= tm.tm_hour;
  min = tm.tm_min;
  sec = tm.tm_sec;
  hour2= tm.tm_hour;
  min2 = tm.tm_min;
  sec2 = tm.tm_sec;

  for (x= 3; x>-1; x--){
    key[x]=(year %10);
    year /= 10;
  }
  for (x= 5; x>3; x--){
    key[x]=(mon) %10;
    mon /= 10;
  }
```

Decompile: entry - (challenge)

```c

undefined8 entry(undefined4 param_1,long param_2)

{
  tm *ptVar1;
  size_t sVar2;
  void *pvVar3;
  FILE *pFVar4;
  int local_180;
  int local_17c;
  int local_178;
  int local_174;
  int local_170;
  int local_16c;
  void *local_148;
  size_t local_140;
  FILE *local_138;
  char *local_130;
  int local_124;
  int local_120;
  int local_11c;
  int local_118;
  int local_114;
  int local_110;
  int local_10c;
  int local_108;
  int local_104;
```

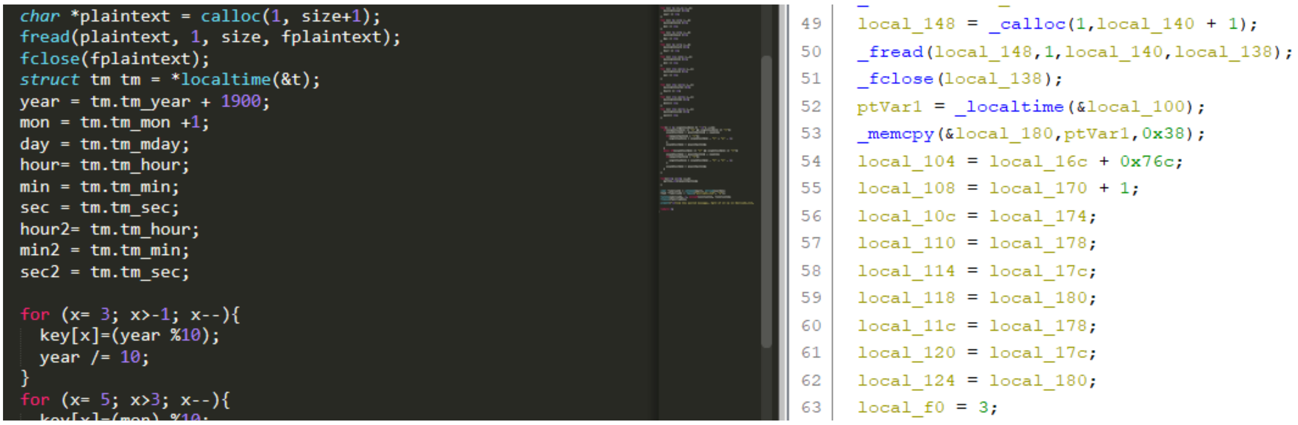
Figure 11: Decompile 3



```c
int main(int argc, char **argv) {
  int key[20], loop, x, i, o;
  time_t t = time(NULL);
  int year, mon, day, hour, min, sec, hour2, min2, sec2;
  char part[100];

  const char *inp_path = argv[1];
  FILE *fplaintext = fopen(inp_path, "r");

  fseek(fplaintext, 0L, SEEK_END);
  long size = ftell(fplaintext);
  rewind(fplaintext);


  char *plaintext = calloc(1, size+1);
  fread(plaintext, 1, size, fplaintext);
  fclose(fplaintext);
  struct tm tm = *localtime(&t);
  year = tm.tm_year + 1900;
  mon = tm.tm_mon +1;
  day = tm.tm_mday;
  hour= tm.tm_hour;
  min = tm.tm_min;
  sec = tm.tm_sec;
  hour2= tm.tm_hour;
  min2 = tm.tm_min;
  sec2 = tm.tm_sec;

  for (x= 3; x>-1; x--){
    key[x]=(year %10);
    year /= 10;
  }
  for (x= 5; x>3; x--){
    key[x]=(mon) %10;
    mon /= 10;
```

```c
  int local_f4;
  uint local_f0;
  long local_e8;
  undefined4 local_e0;
  undefined4 local_dc;
  char local_d8 [112];
  int aiStack104 [22];
  long local_10;

  local_10 = *(long *)___stack_chk_guard;
  local_dc = 0;
  local_e8 = param_2;
  local_e0 = param_1;
  local_100 = _time((time_t *)0x0);
  local_130 = *(char **)(local_e8 + 8);
  local_138 = _fopen(local_130,"r");
  _fseek(local_138,0,2);
  local_140 = _ftell(local_138);
  _rewind(local_138);
  local_148 = _calloc(1,local_140 + 1);
  _fread(local_148,1,local_140,local_138);
  _fclose(local_138);
  ptVar1 = _localtime(&local_100);
  _memcpy(&local_180,ptVar1,0x38);
  local_104 = local_16c + 0x76c;
  local_108 = local_170 + 1;
  local_10c = local_174;
```

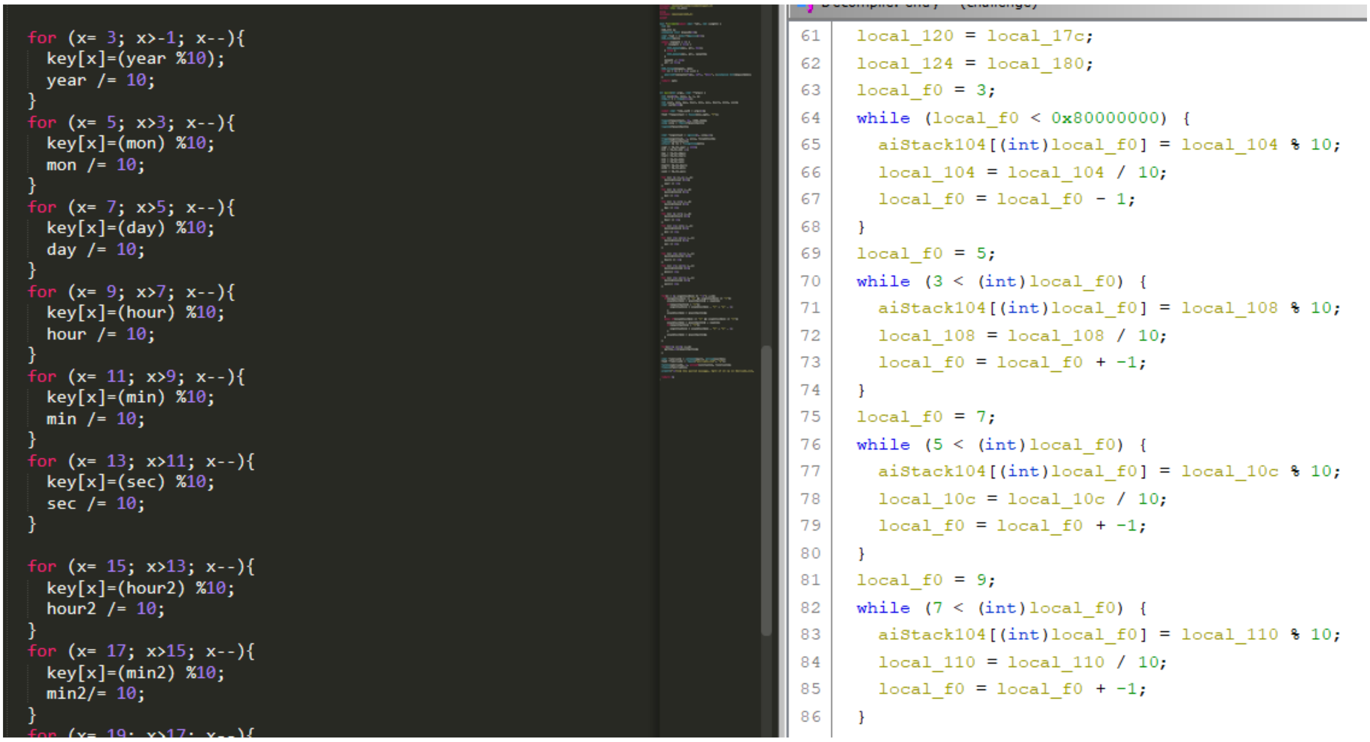
Figure 12: Decompile 4



```c
char *plaintext = calloc(1, size+1);
fread(plaintext, 1, size, fplaintext);
fclose(fplaintext);
struct tm tm = *localtime(&t);
year = tm.tm_year + 1900;
mon = tm.tm_mon +1;
day = tm.tm_mday;
hour= tm.tm_hour;
min = tm.tm_min;
sec = tm.tm_sec;
hour2= tm.tm_hour;
min2 = tm.tm_min;
sec2 = tm.tm_sec;

for (x= 3; x>-1; x--){
  key[x]=(year %10);
  year /= 10;
}
for (x= 5; x>3; x--){
  key[x]=(mon) %10;
```

```c
local_148 = _calloc(1,local_140 + 1);
_fread(local_148,1,local_140,local_138);
_fclose(local_138);
ptVar1 = _localtime(&local_100);
_memcpy(&local_180,ptVar1,0x38);
local_104 = local_16c + 0x76c;
local_108 = local_170 + 1;
local_10c = local_174;
local_110 = local_178;
local_114 = local_17c;
local_118 = local_180;
local_11c = local_178;
local_120 = local_17c;
local_124 = local_180;
local_f0 = 3;
```

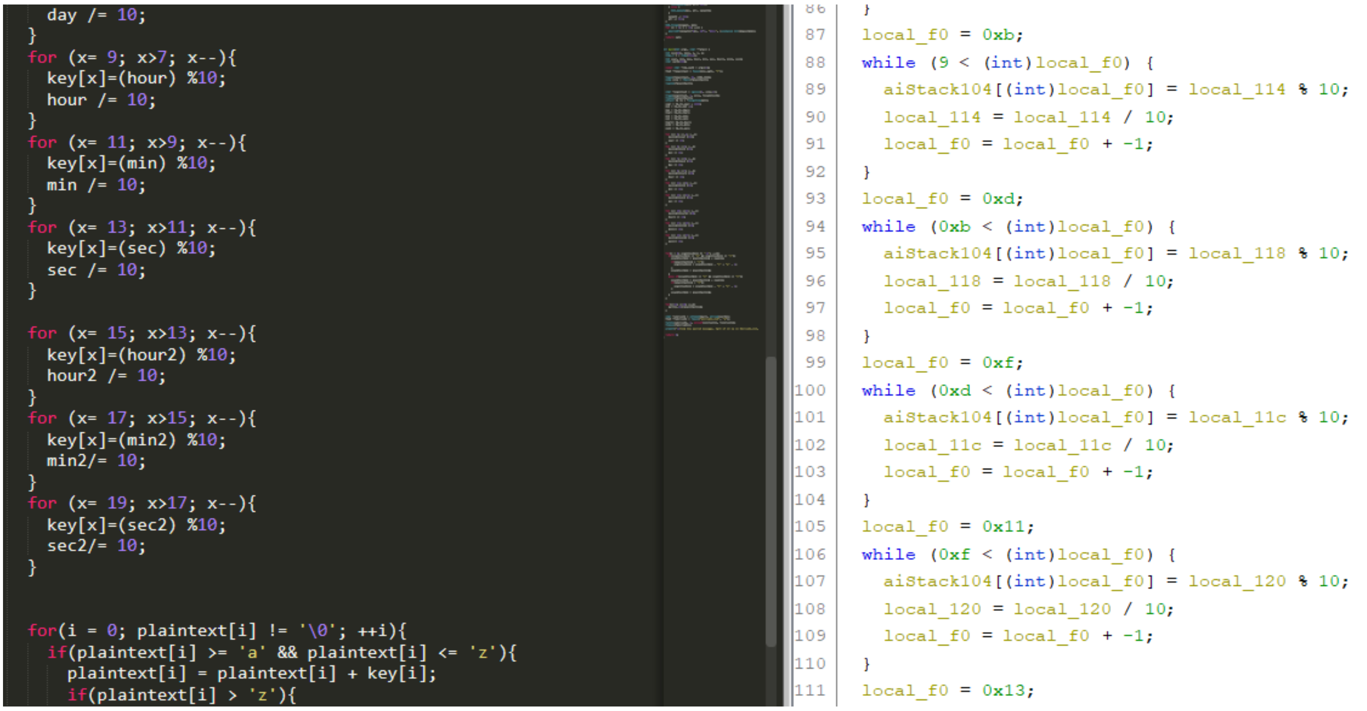
Figure 13: Decompile 5



```
for (x= 3; x>-1; x--){
  key[x]=(year %10);
  year /= 10;
}
for (x= 5; x>3; x--){
  key[x]=(mon) %10;
  mon /= 10;
}
for (x= 7; x>5; x--){
  key[x]=(day) %10;
  day /= 10;
}
for (x= 9; x>7; x--){
  key[x]=(hour) %10;
  hour /= 10;
}
for (x= 11; x>9; x--){
  key[x]=(min) %10;
  min /= 10;
}
for (x= 13; x>11; x--){
  key[x]=(sec) %10;
  sec /= 10;
}

for (x= 15; x>13; x--){
  key[x]=(hour2) %10;
  hour2 /= 10;
}
for (x= 17; x>15; x--){
  key[x]=(min2) %10;
  min2/= 10;
}
for (x= 19; x>17; x--){
```

```
local_120 = local_17c;
local_124 = local_180;
local_f0 = 3;
while (local_f0 < 0x80000000) {
  aiStack104[(int)local_f0] = local_104 % 10;
  local_104 = local_104 / 10;
  local_f0 = local_f0 - 1;
}
local_f0 = 5;
while (3 < (int)local_f0) {
  aiStack104[(int)local_f0] = local_108 % 10;
  local_108 = local_108 / 10;
  local_f0 = local_f0 + -1;
}
local_f0 = 7;
while (5 < (int)local_f0) {
  aiStack104[(int)local_f0] = local_10c % 10;
  local_10c = local_10c / 10;
  local_f0 = local_f0 + -1;
}
local_f0 = 9;
while (7 < (int)local_f0) {
  aiStack104[(int)local_f0] = local_110 % 10;
  local_110 = local_110 / 10;
  local_f0 = local_f0 + -1;
}
```

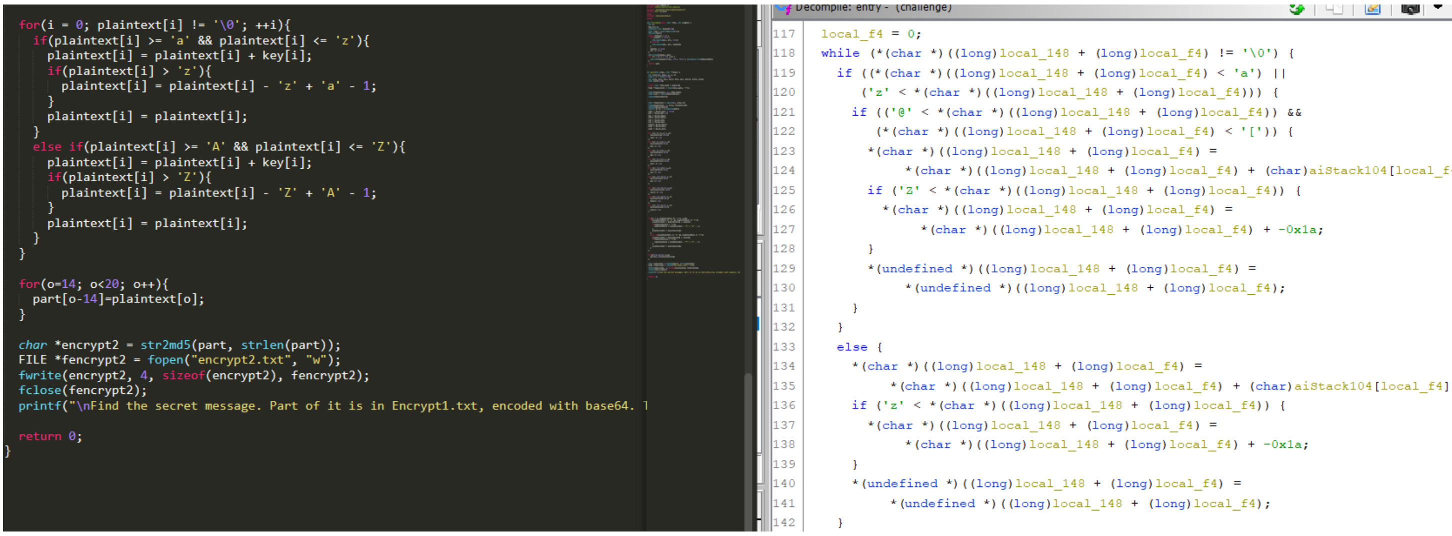
Figure 14: Decompile 6



```
  day /= 10;
}
for (x= 9; x>7; x--){
  key[x]=(hour) %10;
  hour /= 10;
}
for (x= 11; x>9; x--){
  key[x]=(min) %10;
  min /= 10;
}
for (x= 13; x>11; x--){
  key[x]=(sec) %10;
  sec /= 10;
}

for (x= 15; x>13; x--){
  key[x]=(hour2) %10;
  hour2 /= 10;
}
for (x= 17; x>15; x--){
  key[x]=(min2) %10;
  min2/= 10;
}
for (x= 19; x>17; x--){
  key[x]=(sec2) %10;
  sec2/= 10;
}


for(i = 0; plaintext[i] != '\0'; ++i){
  if(plaintext[i] >= 'a' && plaintext[i] <= 'z'){
    plaintext[i] = plaintext[i] + key[i];
    if(plaintext[i] > 'z'){
```

```
}
local_f0 = 0xb;
while (9 < (int)local_f0) {
  aiStack104[(int)local_f0] = local_114 % 10;
  local_114 = local_114 / 10;
  local_f0 = local_f0 + -1;
}
local_f0 = 0xd;
while (0xb < (int)local_f0) {
  aiStack104[(int)local_f0] = local_118 % 10;
  local_118 = local_118 / 10;
  local_f0 = local_f0 + -1;
}
local_f0 = 0xf;
while (0xd < (int)local_f0) {
  aiStack104[(int)local_f0] = local_11c % 10;
  local_11c = local_11c / 10;
  local_f0 = local_f0 + -1;
}
local_f0 = 0x11;
while (0xf < (int)local_f0) {
  aiStack104[(int)local_f0] = local_120 % 10;
  local_120 = local_120 / 10;
  local_f0 = local_f0 + -1;
}
local_f0 = 0x13;
```

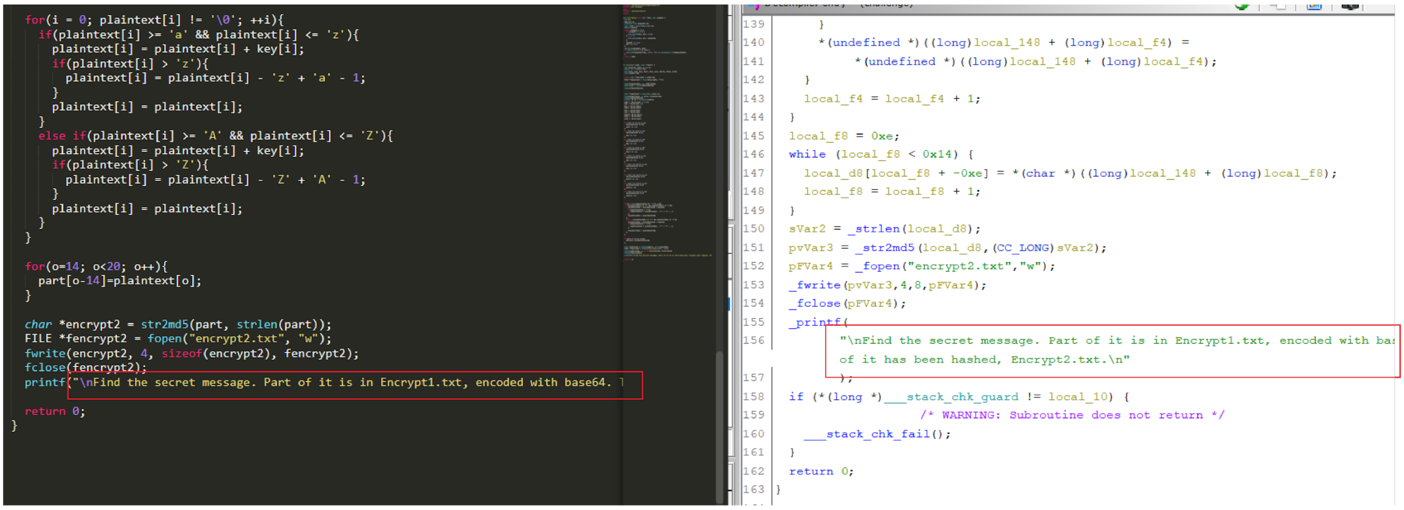
Figure 15: Decompile 7



```
for(i = 0; plaintext[i] != '\0'; ++i){
  if(plaintext[i] >= 'a' && plaintext[i] <= 'z'){
    plaintext[i] = plaintext[i] + key[i];
    if(plaintext[i] > 'z'){
      plaintext[i] = plaintext[i] - 'z' + 'a' - 1;
    }
    plaintext[i] = plaintext[i];
  }
  else if(plaintext[i] >= 'A' && plaintext[i] <= 'Z'){
    plaintext[i] = plaintext[i] + key[i];
    if(plaintext[i] > 'Z'){
      plaintext[i] = plaintext[i] - 'Z' + 'A' - 1;
    }
    plaintext[i] = plaintext[i];
  }
}

for(o=14; o<20; o++){
  part[o-14]=plaintext[o];
}

char *encrypt2 = str2md5(part, strlen(part));
FILE *fencrypt2 = fopen("encrypt2.txt", "w");
fwrite(encrypt2, 4, sizeof(encrypt2), fencrypt2);
fclose(fencrypt2);
printf("\nFind the secret message. Part of it is in Encrypt1.txt, encoded with base64. T

return 0;
}
```

```
  local_f4 = 0;
  while (*(char *)((long)local_148 + (long)local_f4) != '\0') {
    if ((*(char *)((long)local_148 + (long)local_f4) < 'a') ||
       ('z' < *(char *)((long)local_148 + (long)local_f4))) {
      if (('@' < *(char *)((long)local_148 + (long)local_f4)) &&
         (*(char *)((long)local_148 + (long)local_f4) < '[')) {
        *(char *)((long)local_148 + (long)local_f4) =
             *(char *)((long)local_148 + (long)local_f4) + (char)aiStack104[local_f
        if ('Z' < *(char *)((long)local_148 + (long)local_f4)) {
          *(char *)((long)local_148 + (long)local_f4) =
               *(char *)((long)local_148 + (long)local_f4) + -0x1a;
        }
        *(undefined *)((long)local_148 + (long)local_f4) =
             *(undefined *)((long)local_148 + (long)local_f4);
      }
    }
    else {
      *(char *)((long)local_148 + (long)local_f4) =
           *(char *)((long)local_148 + (long)local_f4) + (char)aiStack104[local_f4]
      if ('z' < *(char *)((long)local_148 + (long)local_f4)) {
        *(char *)((long)local_148 + (long)local_f4) =
             *(char *)((long)local_148 + (long)local_f4) + -0x1a;
      }
      *(undefined *)((long)local_148 + (long)local_f4) =
           *(undefined *)((long)local_148 + (long)local_f4);
    }
```

Figure 16: Decompile 8



```
for(i = 0; plaintext[i] != '\0'; ++i){
  if(plaintext[i] >= 'a' && plaintext[i] <= 'z'){
    plaintext[i] = plaintext[i] + key[i];
    if(plaintext[i] > 'z'){
      plaintext[i] = plaintext[i] - 'z' + 'a' - 1;
    }
    plaintext[i] = plaintext[i];
  }
  else if(plaintext[i] >= 'A' && plaintext[i] <= 'Z'){
    plaintext[i] = plaintext[i] + key[i];
    if(plaintext[i] > 'Z'){
      plaintext[i] = plaintext[i] - 'Z' + 'A' - 1;
    }
    plaintext[i] = plaintext[i];
  }
}

for(o=14; o<20; o++){
  part[o-14]=plaintext[o];
}

char *encrypt2 = str2md5(part, strlen(part));
FILE *fencrypt2 = fopen("encrypt2.txt", "w");
fwrite(encrypt2, 4, sizeof(encrypt2), fencrypt2);
fclose(fencrypt2);
printf("\nFind the secret message. Part of it is in Encrypt1.txt, encoded with base64. T

return 0;
}
```

```
      }
      *(undefined *)((long)local_148 + (long)local_f4) =
           *(undefined *)((long)local_148 + (long)local_f4);
    }
    local_f4 = local_f4 + 1;
  }
  local_f8 = 0xe;
  while (local_f8 < 0x14) {
    local_d8[local_f8 + -0xe] = *(char *)((long)local_148 + (long)local_f8);
    local_f8 = local_f8 + 1;
  }
  sVar2 = _strlen(local_d8);
  pvVar3 = _str2md5(local_d8,(CC_LONG)sVar2);
  pFVar4 = _fopen("encrypt2.txt","w");
  _fwrite(pvVar3,4,8,pFVar4);
  _fclose(pFVar4);
  _printf(
         "\nFind the secret message. Part of it is in Encrypt1.txt, encoded with bas
         of it has been hashed, Encrypt2.txt.\n"
         );
  if (*(long *)___stack_chk_guard != local_10) {
                    /* WARNING: Subroutine does not return */
    ___stack_chk_fail();
  }
  return 0;
}
```

Figure 17: Final Decompile

From Figure 15, we can identify the encryption scheme in the printf function. We can derive that half of the secret message is in encrypt1.txt, encoded with base64. While the other half in encrypt2.txt, has been hashed.

4.2 Breaking the hash

There is no easy way around this, groups will have to modify the code for generating rainbow tables that was learnt in class. In class we only did lowercase characters with numbers but now they have to include uppercase letters and numbers as well.

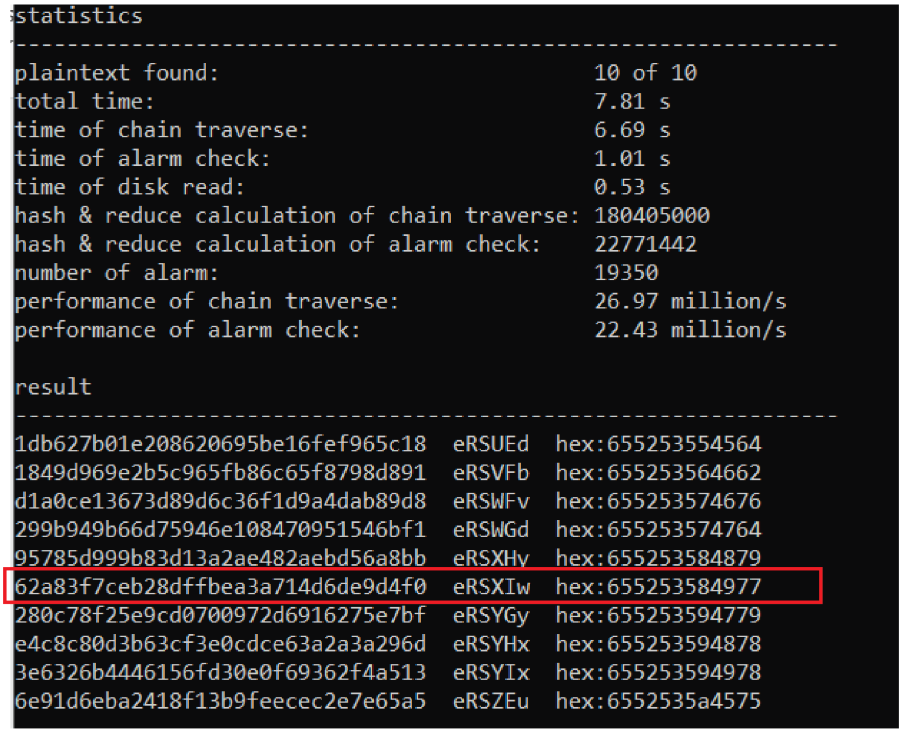
To change it, they have to read the documentation, eventually coming to this link to docs. We can see some screenshots of our attempts at the rainbow crack being successful.

Charset used: mixalpha-numeric = ”abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789”

1. We generated 6 rainbow tables of different reduction index:
    - rtgen md5 mixalpha-numeric 6 6 1 3800 10000000 0
    - rtgen md5 mixalpha-numeric 6 6 2 3800 10000000 0
    - rtgen md5 mixalpha-numeric 6 6 4 3800 8000000 0
    - rtgen md5 mixalpha-numeric 6 6 6 3800 5000000 0
    - rtgen md5 mixalpha-numeric 6 6 8 3800 5000000 0
    - rtgen md5 mixalpha-numeric 6 6 10 3800 5000000 0

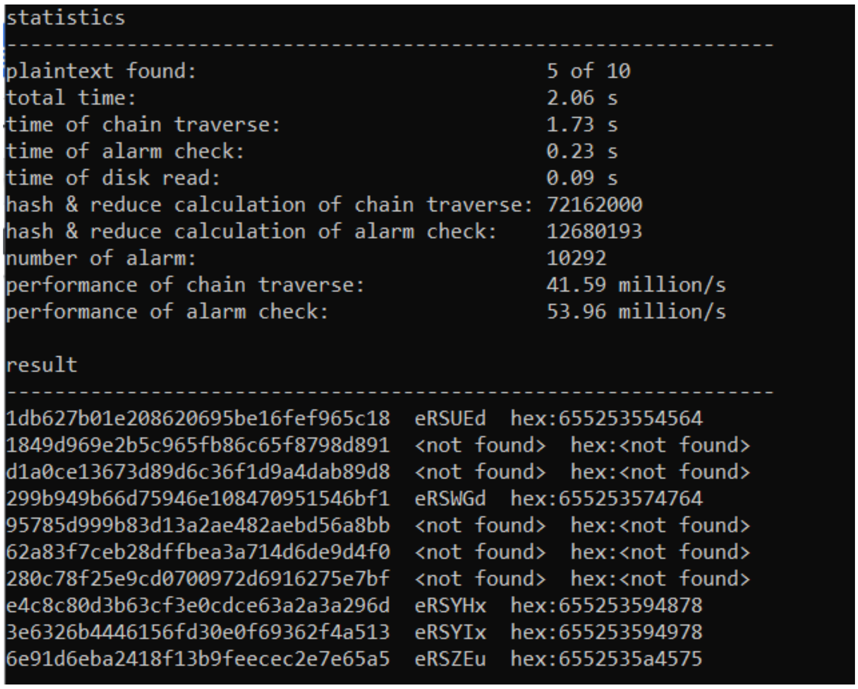
2. Next, we do a rtsort to sort all the rainbow tables.
3. Lastly, we use rtcrack to crack the hash and get p3.

Our initial intention was to use ascii-32-95=[!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~]and a plaintext length of 10/8, however due to the number of permutations and combinations possible and the amount of time needed to generate multiple rainbow table with 50 million inputs each (~3hrs), we were unable to find the plaintext and decided to drop the difficulty.



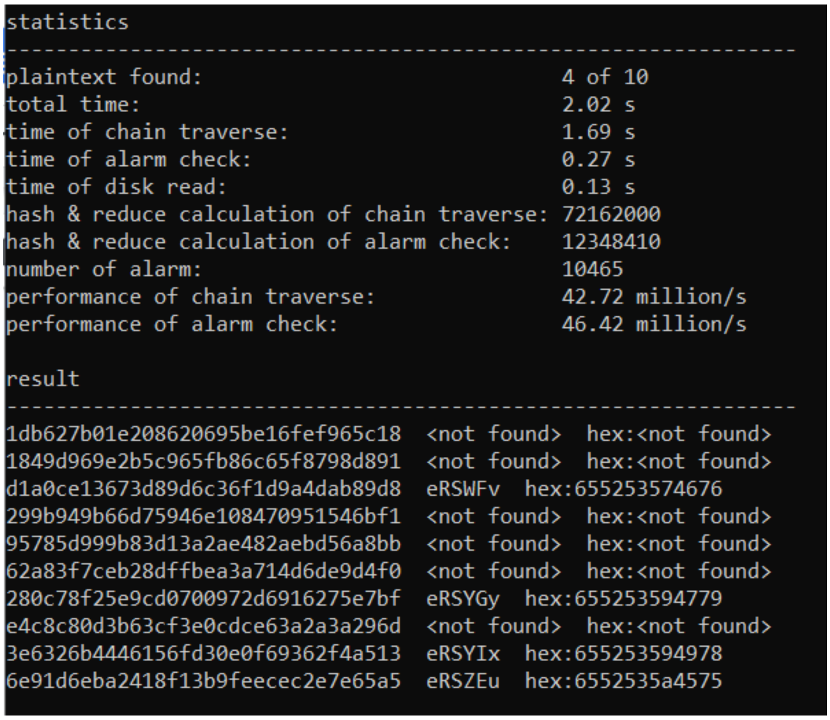
```
statistics
----------------------------------------------------------------
plaintext found:                             10 of 10
total time:                                  7.81 s
time of chain traverse:                      6.69 s
time of alarm check:                         1.01 s
time of disk read:                           0.53 s
hash & reduce calculation of chain traverse: 180405000
hash & reduce calculation of alarm check:    22771442
number of alarm:                             19350
performance of chain traverse:               26.97 million/s
performance of alarm check:                  22.43 million/s

result
----------------------------------------------------------------
1db627b01e208620695be16fef965c18  eRSUEd  hex:655253554564
1849d969e2b5c965fb86c65f8798d891  eRSVFb  hex:655253564662
d1a0ce13673d89d6c36f1d9a4dab89d8  eRSWFv  hex:655253574676
299b949b66d75946e108470951546bf1  eRSWGd  hex:655253574764
95785d999b83d13a2ae482aebd56a8bb  eRSXHy  hex:655253584879
62a83f7ceb28dffbea3a714d6de9d4f0  eRSXIw  hex:655253584977
280c78f25e9cd0700972d6916275e7bf  eRSYGy  hex:655253594779
e4c8c80d3b63cf3e0cdce63a2a3a296d  eRSYHx  hex:655253594878
3e6326b4446156fd30e0f69362f4a513  eRSYIx  hex:655253594978
6e91d6eba2418f13b9feecec2e7e65a5  eRSZEu  hex:6552535a4575
```

Fig 18: Cracking various hashes to get the plaintext



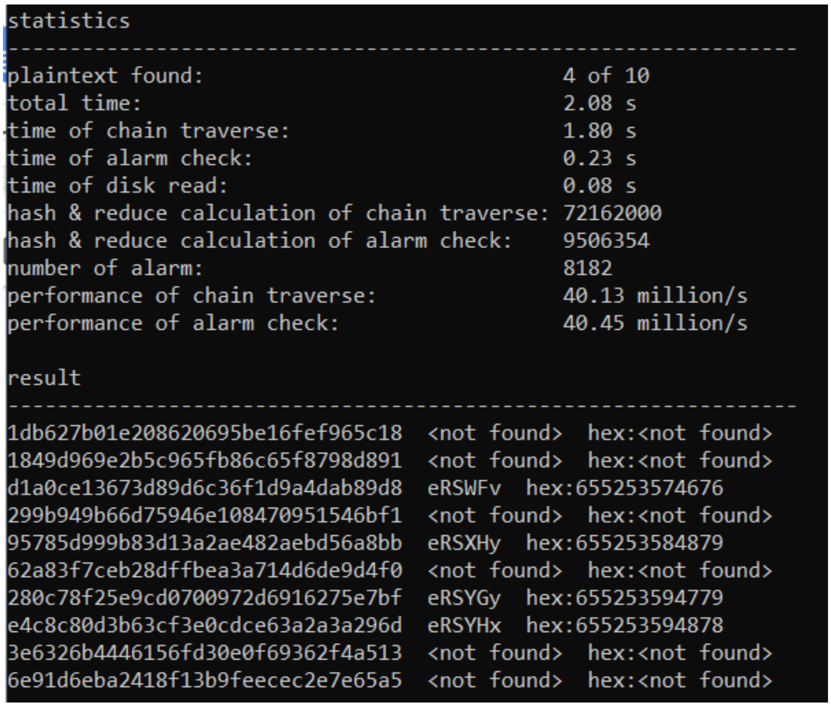
```
statistics
---------------------------------------------------------------
plaintext found:                             5 of 10
total time:                                  2.06 s
time of chain traverse:                      1.73 s
time of alarm check:                         0.23 s
time of disk read:                           0.09 s
hash & reduce calculation of chain traverse: 72162000
hash & reduce calculation of alarm check:    12680193
number of alarm:                             10292
performance of chain traverse:               41.59 million/s
performance of alarm check:                  53.96 million/s

result
---------------------------------------------------------------
1db627b01e208620695be16fef965c18  eRSUEd  hex:655253554564
1849d969e2b5c965fb86c65f8798d891  <not found>  hex:<not found>
d1a0ce13673d89d6c36f1d9a4dab89d8  <not found>  hex:<not found>
299b949b66d75946e108470951546bf1  eRSWGd  hex:655253574764
95785d999b83d13a2ae482aebd56a8bb  <not found>  hex:<not found>
62a83f7ceb28dffbea3a714d6de9d4f0  <not found>  hex:<not found>
280c78f25e9cd0700972d6916275e7bf  <not found>  hex:<not found>
e4c8c80d3b63cf3e0cdce63a2a3a296d  eRSYHx  hex:655253594878
3e6326b4446156fd30e0f69362f4a513  eRSYIx  hex:655253594978
6e91d6eba2418f13b9feecec2e7e65a5  eRSZEu  hex:6552535a4575
```

Fig 19: Cracking a list of hashes using reduction 1



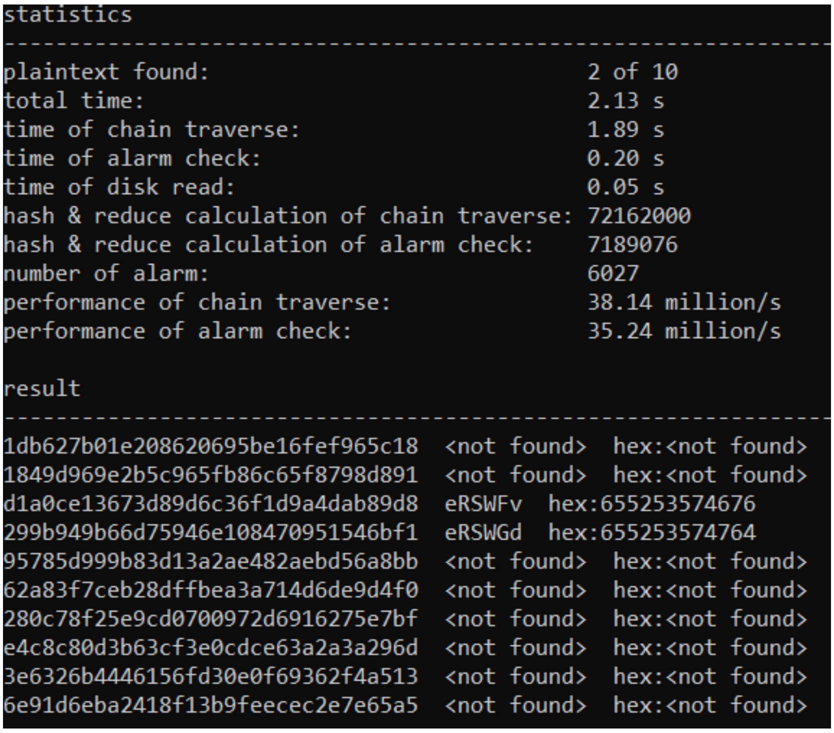
```
statistics
---------------------------------------------------------------
plaintext found:                             4 of 10
total time:                                  2.02 s
time of chain traverse:                      1.69 s
time of alarm check:                         0.27 s
time of disk read:                           0.13 s
hash & reduce calculation of chain traverse: 72162000
hash & reduce calculation of alarm check:    12348410
number of alarm:                             10465
performance of chain traverse:               42.72 million/s
performance of alarm check:                  46.42 million/s

result
---------------------------------------------------------------
1db627b01e208620695be16fef965c18  <not found>  hex:<not found>
1849d969e2b5c965fb86c65f8798d891  <not found>  hex:<not found>
d1a0ce13673d89d6c36f1d9a4dab89d8  eRSWFv  hex:655253574676
299b949b66d75946e108470951546bf1  <not found>  hex:<not found>
95785d999b83d13a2ae482aebd56a8bb  <not found>  hex:<not found>
62a83f7ceb28dffbea3a714d6de9d4f0  <not found>  hex:<not found>
280c78f25e9cd0700972d6916275e7bf  eRSYGy  hex:655253594779
e4c8c80d3b63cf3e0cdce63a2a3a296d  <not found>  hex:<not found>
3e6326b4446156fd30e0f69362f4a513  eRSYIx  hex:655253594978
6e91d6eba2418f13b9feecec2e7e65a5  eRSZEu  hex:6552535a4575
```

Fig 20: Cracking a list of hashes using reduction 2



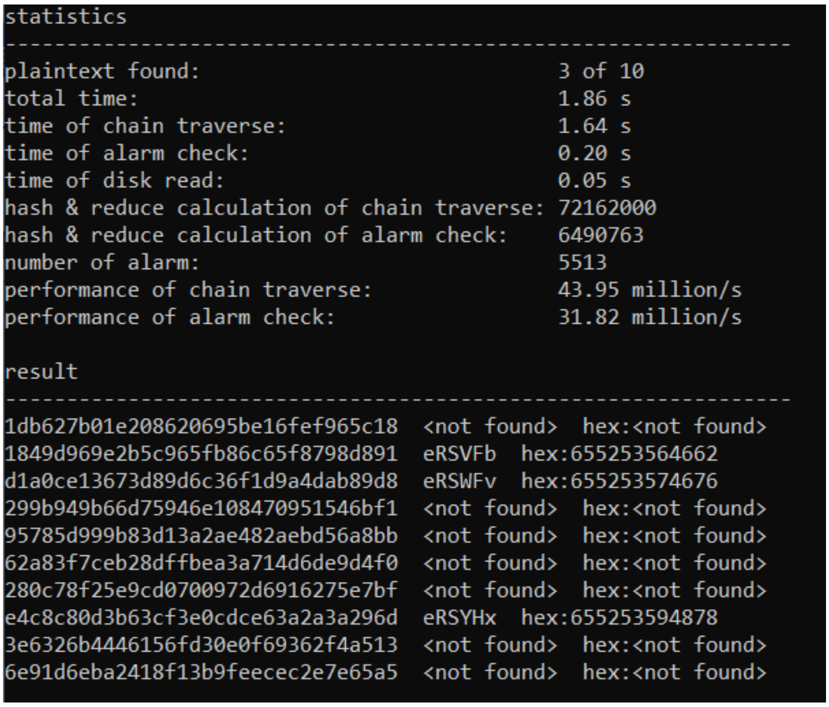
```
statistics
-------------------------------------------------------------
plaintext found:                            4 of 10
total time:                                 2.08 s
time of chain traverse:                     1.80 s
time of alarm check:                        0.23 s
time of disk read:                          0.08 s
hash & reduce calculation of chain traverse: 72162000
hash & reduce calculation of alarm check:    9506354
number of alarm:                            8182
performance of chain traverse:              40.13 million/s
performance of alarm check:                 40.45 million/s

result
-------------------------------------------------------------
1db627b01e208620695be16fef965c18  <not found>  hex:<not found>
1849d969e2b5c965fb86c65f8798d891  <not found>  hex:<not found>
d1a0ce13673d89d6c36f1d9a4dab89d8  eRSWFv  hex:655253574676
299b949b66d75946e108470951546bf1  <not found>  hex:<not found>
95785d999b83d13a2ae482aebd56a8bb  eRSXHy  hex:655253584879
62a83f7ceb28dffbea3a714d6de9d4f0  <not found>  hex:<not found>
280c78f25e9cd0700972d6916275e7bf  eRSYGy  hex:655253594779
e4c8c80d3b63cf3e0cdce63a2a3a296d  eRSYHx  hex:655253594878
3e6326b4446156fd30e0f69362f4a513  <not found>  hex:<not found>
6e91d6eba2418f13b9feecec2e7e65a5  <not found>  hex:<not found>
```

Fig 21: Cracking a list of hashes using reduction 4



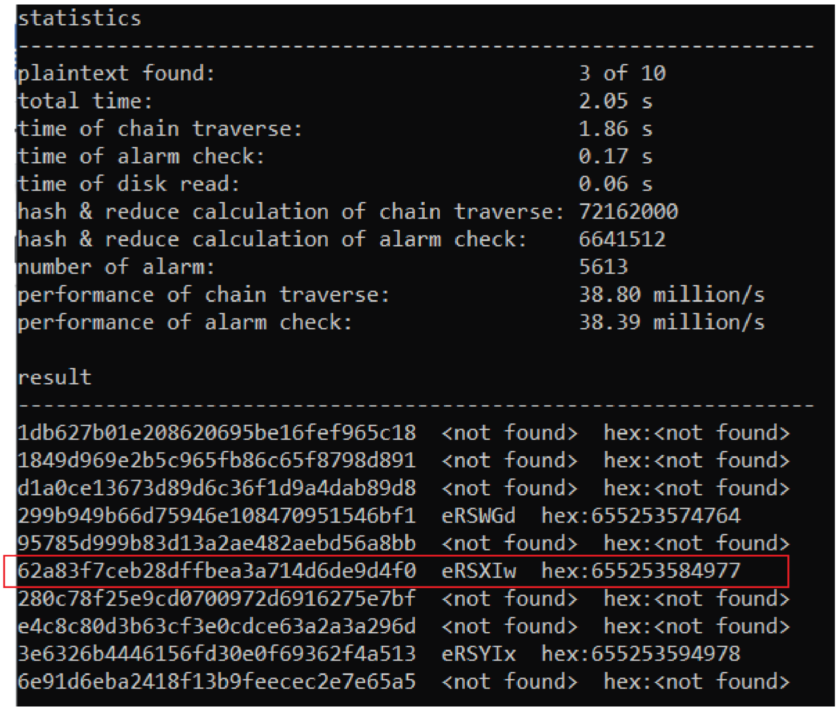
```
statistics
-------------------------------------------------------------
plaintext found:                            2 of 10
total time:                                 2.13 s
time of chain traverse:                     1.89 s
time of alarm check:                        0.20 s
time of disk read:                          0.05 s
hash & reduce calculation of chain traverse: 72162000
hash & reduce calculation of alarm check:    7189076
number of alarm:                            6027
performance of chain traverse:              38.14 million/s
performance of alarm check:                 35.24 million/s

result
-------------------------------------------------------------
1db627b01e208620695be16fef965c18  <not found>  hex:<not found>
1849d969e2b5c965fb86c65f8798d891  <not found>  hex:<not found>
d1a0ce13673d89d6c36f1d9a4dab89d8  eRSWFv  hex:655253574676
299b949b66d75946e108470951546bf1  eRSWGd  hex:655253574764
95785d999b83d13a2ae482aebd56a8bb  <not found>  hex:<not found>
62a83f7ceb28dffbea3a714d6de9d4f0  <not found>  hex:<not found>
280c78f25e9cd0700972d6916275e7bf  <not found>  hex:<not found>
e4c8c80d3b63cf3e0cdce63a2a3a296d  <not found>  hex:<not found>
3e6326b4446156fd30e0f69362f4a513  <not found>  hex:<not found>
6e91d6eba2418f13b9feecec2e7e65a5  <not found>  hex:<not found>
```

Fig 22:   Cracking a list of hashes using reduction 6



```
statistics
-------------------------------------------------------------
plaintext found:                            3 of 10
total time:                                 1.86 s
time of chain traverse:                     1.64 s
time of alarm check:                        0.20 s
time of disk read:                          0.05 s
hash & reduce calculation of chain traverse: 72162000
hash & reduce calculation of alarm check:    6490763
number of alarm:                            5513
performance of chain traverse:              43.95 million/s
performance of alarm check:                 31.82 million/s

result
-------------------------------------------------------------
1db627b01e208620695be16fef965c18  <not found>  hex:<not found>
1849d969e2b5c965fb86c65f8798d891  eRSVFb  hex:655253564662
d1a0ce13673d89d6c36f1d9a4dab89d8  eRSWFv  hex:655253574676
299b949b66d75946e108470951546bf1  <not found>  hex:<not found>
95785d999b83d13a2ae482aebd56a8bb  <not found>  hex:<not found>
62a83f7ceb28dffbea3a714d6de9d4f0  <not found>  hex:<not found>
280c78f25e9cd0700972d6916275e7bf  <not found>  hex:<not found>
e4c8c80d3b63cf3e0cdce63a2a3a296d  eRSYHx  hex:655253594878
3e6326b4446156fd30e0f69362f4a513  <not found>  hex:<not found>
6e91d6eba2418f13b9feecec2e7e65a5  <not found>  hex:<not found>
```

Fig 23: Cracking a list of hashes using reduction 8

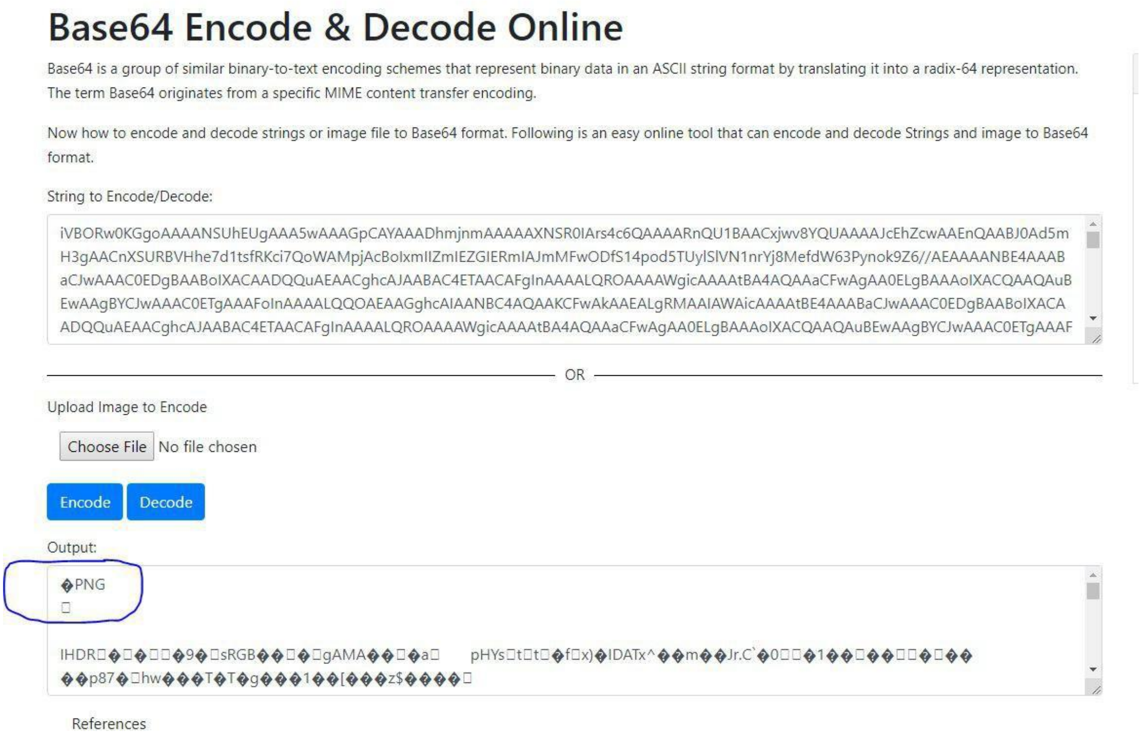


```
statistics
-------------------------------------------------------------
plaintext found:                            3 of 10
total time:                                 2.05 s
time of chain traverse:                     1.86 s
time of alarm check:                        0.17 s
time of disk read:                          0.06 s
hash & reduce calculation of chain traverse: 72162000
hash & reduce calculation of alarm check:    6641512
number of alarm:                            5613
performance of chain traverse:              38.80 million/s
performance of alarm check:                 38.39 million/s

result
-------------------------------------------------------------
1db627b01e208620695be16fef965c18  <not found>  hex:<not found>
1849d969e2b5c965fb86c65f8798d891  <not found>  hex:<not found>
d1a0ce13673d89d6c36f1d9a4dab89d8  <not found>  hex:<not found>
299b949b66d75946e108470951546bf1  eRSWGd  hex:655253574764
95785d999b83d13a2ae482aebd56a8bb  <not found>  hex:<not found>
62a83f7ceb28dffbea3a714d6de9d4f0  eRSXIw  hex:655253584977
280c78f25e9cd0700972d6916275e7bf  <not found>  hex:<not found>
e4c8c80d3b63cf3e0cdce63a2a3a296d  <not found>  hex:<not found>
3e6326b4446156fd30e0f69362f4a513  eRSYIx  hex:655253594978
6e91d6eba2418f13b9feecec2e7e65a5  <not found>  hex:<not found>
```

Fig 24: Finding the plaintext only at reduction 10

4.3 Generating a valid image

After learning the scheme from the decompiled C code, teams can find out that the text has been encrypted in b64. Following up on this, they can use an online tool to check the original file type of this as shown in the figure below. In the figure, it indicates that it is a PNG but we see the rest of the output is scrambled which means that they will know that we have scrambled it.



Base64 Encode & Decode Online

Base64 is a group of similar binary-to-text encoding schemes that represent binary data in an ASCII string format by translating it into a radix-64 representation. The term Base64 originates from a specific MIME content transfer encoding.

Now how to encode and decode strings or image file to Base64 format. Following is an easy online tool that can encode and decode Strings and image to Base64 format.

String to Encode/Decode:

```
iVBORw0KGgoAAAANSUhEUgAAA5wAAAGpCAYAAADhmjnmAAAAAXNSR0IArs4c6QAAAARnQU1BAACxjwv8YQUAAAAJcEhZcwAAEnQAABJ0Ad5m
H3gAACnXSURBVHhe7d1tsfRKci7QoWAMpjAcBoIxmIlZmIEZGIERmIAJmMFwODfS14pod5TUylSlVN1nrYj8MefdW63Pynok9Z6//AEAAAANBE4AAAB
aCJwAAAC0EDgBAABoIXACAADQQuAEAACghcAJAABAC4ETAACAFgInAAAALQROAAAAWgicAAAAtBA4AQAAaCFwAgAA0ELgBAAAoIXACQAAQAuB
EwAAgBYCJwAAAC0ETgAAAFoInAAAALQQOAEAAGghcAIAANBC4AQAAKCFwAkAAEALgRMAAIAWAicAAAAtBE4AAABaCJwAAAC0EDgBAABoIXACA
ADQQuAEAACghcAJAABAC4ETAACAFgInAAAALQROAAAAWgicAAAAtBA4AQAAaCFwAgAA0ELgBAAAoIXACQAAQAuBEwAAgBYCJwAAAC0ETgAAAF
```

OR

Upload Image to Encode

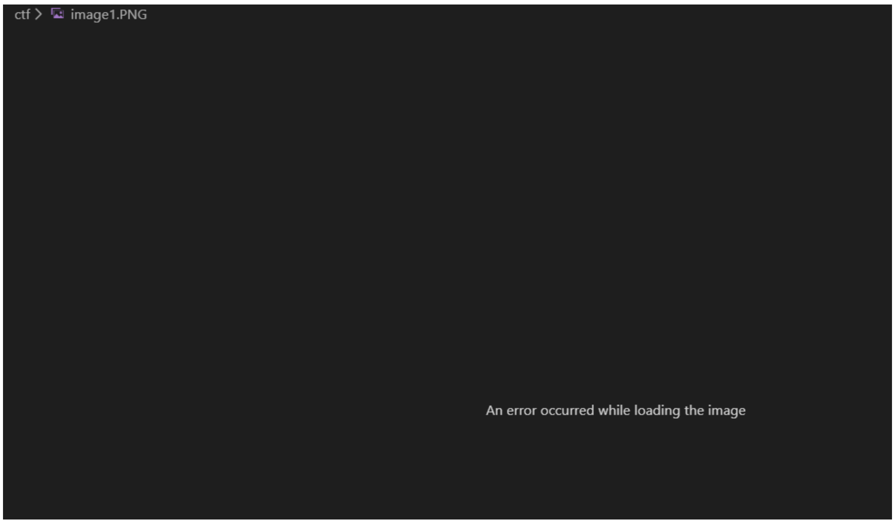
Output:

```
�PNG
□

IHDR□�□�□□�9�□sRGB��□�□gAMA��□�a□      pHYs□t□t□�f□x)�IDATx^��m��Jr.C`�0□□�1��□��□□�□��
��p87�□hw���T�T�g���1��[���z$����□
```

Figure 25: online tool b64

Teams will have to permute the different sized blocks that the image could have been scrambled by. For each sized block X, they have to permute the possibilities and check for a valid image. There are 2 methods of doing this.

The manual method would be to create multiple images and open them one at a time. This will show that the image has an error



An error occurred while loading the image

Figure 26: Error in opening image

The automated method is provided in the python code.

When they eventually get the valid image, this will be the first half of the shift cipher. The code can be found in image_scramble.py

4.4 Breaking the shift cipher

Based on the scheme, teams can easily guess the year and month that was used. This was intentional to reduce the number of combinations. It was 201911 or 201912. The day will have a range between 27 Nov- 05 Dec which is when we can still change the cipher texts. They will then have to brute force the hours, minutes and seconds which is not difficult as there are limited possibilities (~1000000 possibilities).

The steps to decrypt the code, we have provided a sample code with comments to explain the thought process involved in cracking it. The sample code is brute_force.py.