Midterm Project

CSC-6991

Kevin Clause

Go8212

For this project you will need:

1) Parrot installation

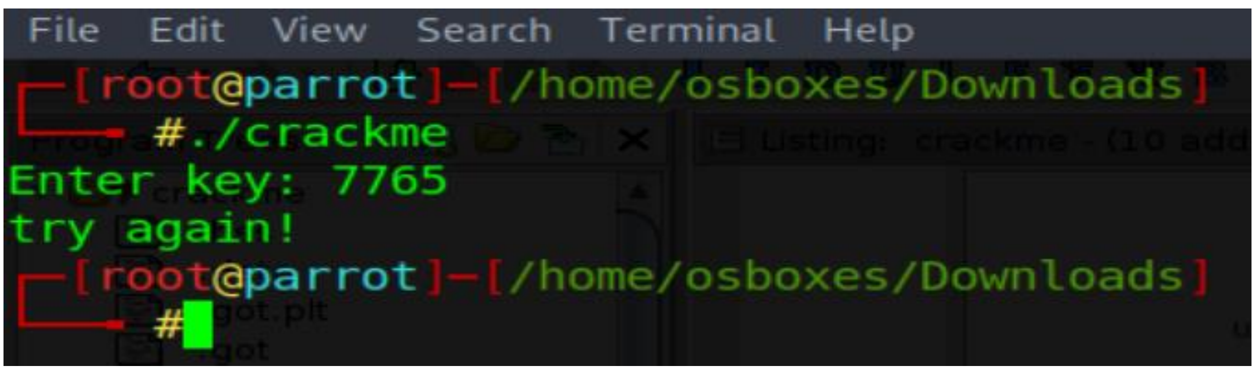
- ghidra application up and working

2) Download the following file on the Parrot node:

https://waynestateprod-my.sharepoint.com/:u:/g/personal/hg1702_wayne_edu/EV1RR9aRihxGqmP0veDW9HMBqaTzqUbhj8WrVPCC1RO6Ng?e=jHGoU6

1) Perform an analysis on the file.

What does the application do?



On our initial run of the file, we are able to see that it prompts the user for a password. Its then prints a statement regarding the correctness of our input. We are able to presume that this file executes a comparison between user input and some password. The program then executes separate branches regarding the user input.

```c
  local_10 = *(long *)(in_FS_OFFSET + 0x28);
  Mem_address = 0x3534323160376761;
  local_1a0 = 0x3a3b313b60613430;
  local_198 = 0x3161333a3360313b;
  local_190 = 0x67373b6132376667;
  local_188 = 0;
  local_178 = 0x3533356431383331;
  local_170 = 0x3638323930383737;
  local_168 = 0x3065633735393638;
  local_160 = 0x6161653163396262;
  local_158 = 0;
  local_148 = 0x61313b323a3b373b;
  local_140 = 0x616667606031 3b3a;
  local_138 = 0x6031316135326133;
  local_130 = 0x6734673730333734;
  local_128 = 0;
  printf("Enter key: ");
  __isoc99_scanf(&DAT_00102010,local_118);
  __s2 = (char *)Hash_function((long)&Mem_address,0x20,3);
  iVar1 = strcmp(local_118,__s2);
  if (iVar1 == 0) {
    puts("good job!");
  }
  else {
    puts("try again!");
  }
  if (local_10 != *(long *)(in_FS_OFFSET + 0x28)) {
                    /* WARNING: Subroutine does not return */
```

Looking to Ghidra we are able to decompile the file and take a closer look into the code. We are able to see that this application takes in user input, calculates a value __s2, and the compares the user input with the value stored in __s2. The result of this comparison is then stored into 'iVar1.' The program then prints out "good job!" or "try again!" depending on the Boolean value stored in iVar1.

Decompile: Hash_function - (crackme)

```c

long Hash_function(long Mem_address,int Int_32,byte Int_3)

{
  int i;

  i = 0;
  while (i < Int_32) {
    *(byte *)(Mem_address + i) = *(byte *)(Mem_address + i) ^ Int_3;
    i = i + 1;
  }
  return Mem_address;
}

```

In order to get a better idea of what this function is doing we also must look into the Hash_function. Decompiling the hash function and re-naming a few variables for clarity we are able to see that this function is being passed a memory address and two immediate values. These values are then XORed to calculate our hashed value. This as a loop that executed 32 times and hashes each character with the value 3. We are then storing the XORed value back into memory. Once our loop has finished executing, we return the XORed memory address.

```
__s2 = (char *)Hash_function((long)&Mem_address,0x20,3);
iVar1 = strcmp(local_118,__s2);
if (iVar1 == 0) {
  puts("good job!");
}
```

Finally, the now XORed memory address is compared to the user input. The program then executes one of two branches depending on the result of our strcmp.

What language is it written in?

```
Decompile: Hash_function - (crackme)

long Hash_function(long Mem_address,int Int_32,byte Int_3)

{
  int i;

  i = 0;
  while (i < Int_32) {
    *(byte *)(Mem_address + i) = *(byte *)(Mem_address + i) ^ Int_3;
    i = i + 1;
  }
  return Mem_address;
}
```

Decompiling our code we are able to see syntax that resembles some sort of C language.

```
00h
"libstdc++.so.6"
```

Searching throughout the assembly we are able to find a reference to a C++ library which allows us to conclude that this program was written in C++.

2) The end goal is to discover the password.

In order to solve this crackme we need to determine what the password. We can do this one of two ways. The first and most intuitive was to find that value stored in __s2. That way we can simply mimic the compared value and produce a true result.

Since the value stored in __s2 is a calculated value and not a stored value we are unable to find the password stored in memory. In order to find a calculated value we would either need to calculate the value returned by hash function or set a breakpoint post execution and verify the value recently stored in the previously unutilized __s2. Since we cannot De-bug programs through Ghidra we employ the use of ltrace. Ltrace is chosen because of its ability to intercept and print system calls executed by the program. It also shows the parameters of invoked functions and systems calls.

```
┌─[root@parrot]─[/home/osboxes/Downloads]
└──╼ #ltrace ./crackme
printf("Enter key: ")                                    = 11
__isoc99_scanf(0x55fb23793010, 0x7fffd6bb5930, 0, 0Enter key: hi
) = 1
strcmp("hi", "bd4c217637bc828982c090b2de41b84d"...) = 6
puts("try again!"try again!
)                                  = 11
+++ exited (status 0) +++
```

Running an ltrace on crackme we are able to see that our user input is compared to the highlighted string. We the copy the highlighted string and plug it back into the program. This is able to net us a true result and we have correctly solved this crackme.

```
┌─[root@parrot]─[/home/osboxes/Downloads]
└──╼ #ltrace ./crackme
printf("Enter key: ")                                    = 11
__isoc99_scanf(0x557046e49010, 0x7fff44d884f0, 0, 0Enter key: bd4c217637bc828982
c090b2de41b84d
) = 1
strcmp("bd4c217637bc828982c090b2de41b84d"..., "bd4c217637bc828982c090b2de41b84d"
...) = 0
puts("good job!"good job!
)                                  = 10
+++ exited (status 0) +++
┌─[root@parrot]─[/home/osboxes/Downloads]
└──╼ #
```

Here we can see that the crackme has been defeated.

But what would we do if we did not have access to ltrace, or even better yet, ltrace had not work?

Well, we would have to first identify the value that is being hashed by our function and then we would have to apply the hashing function to ascertain our password. Let’s do that.

```
printf("Enter key: ");
__isoc99_scanf(&DAT_00102010,local_118);
__s2 = (char *)Hash_function((long)&Mem_address,0x20,3);
iVar1 = strcmp(local_118,__s2);
if (iVar1 == 0) {
```

Looking at our Hashing function we are able to see that it is passed the memory address of variable Mem_address.

Decompile: Hash_function - (crackme)

```
long Hash_function(long Mem_address,int Int_32,byte Int_3)

{
  int i;

  i = 0;
  while (i < Int_32) {
    *(byte *)(Mem_address + i) = *(byte *)(Mem_address + i) ^ Int_3;
    i = i + 1;
  }
  return Mem_address;
}

```

This memory address is then has each bit XORed with the value 3, However it doesn't stop there. See this loop executes 32 times and the initial memory address only contains 8 bytes worth of information which leaves us with a very fun question. Were is the rest of the data? We have 8 bytes of the unhashed password and are missing the other 24 bits. Well if we go back to the main function we and look to the variable declarations we begin to truly understand what this program is doing.

Decompile: Main - (crackme)

```
undefined8 Main(void)

{
  int iVar1;
  char *__s2;
  long in_FS_OFFSET;
  undefined8 Mem_address;
  undefined8 local_1a0;
  undefined8 local_198;
  undefined8 local_190;
  undefined local_188;
  undefined8 local_178;
```

Here we are able to see the additional variable defined directly after Mem_address. Why is this important? Well variables defined consecutively will be stored in memory in that same order, which mean that is there happened to be some sort of overflow and or overwrite those values would be affected. These additional values also add up to be 24 bytes in size. Aha we begin to see the full picture.

Decompile: Hash_function - (crackme)

```
long Hash_function(long Mem_address,int Int_32,byte Int_3)

{
  int i;

  i = 0;
  while (i < Int_32) {
    *(byte *)(Mem_address + i) = *(byte *)(Mem_address + i) ^ Int_3;
    i = i + 1;
  }
  return Mem_address;
}
```

Now looking back to out hashing function we are able to see that we start at the passed memory address and then XOR the following 32 bytes of data in order to calculate our password. This also means that we now have more data to work with.

```
local_10 = *(long *)(in_FS_OFFSET + 0x28);
Mem_address = 0x3534323160376761;
local_1a0 = 0x3a3b313b60613430;
local_198 = 0x3161333a3360313b;
local_190 = 0x67373b6132376667;
local_188 = 0;
```

Now looking at our initialized variables we begin to see that our password is actually the culmination of these first four variables. Sneaky sneaky.

Despite our recent break though we still have not deciphered the password to this crackme. In order to do so we must XOR each of the 32 Bytes with 3. Back in Ghidra we look the Bytes windows to grab the hex values.

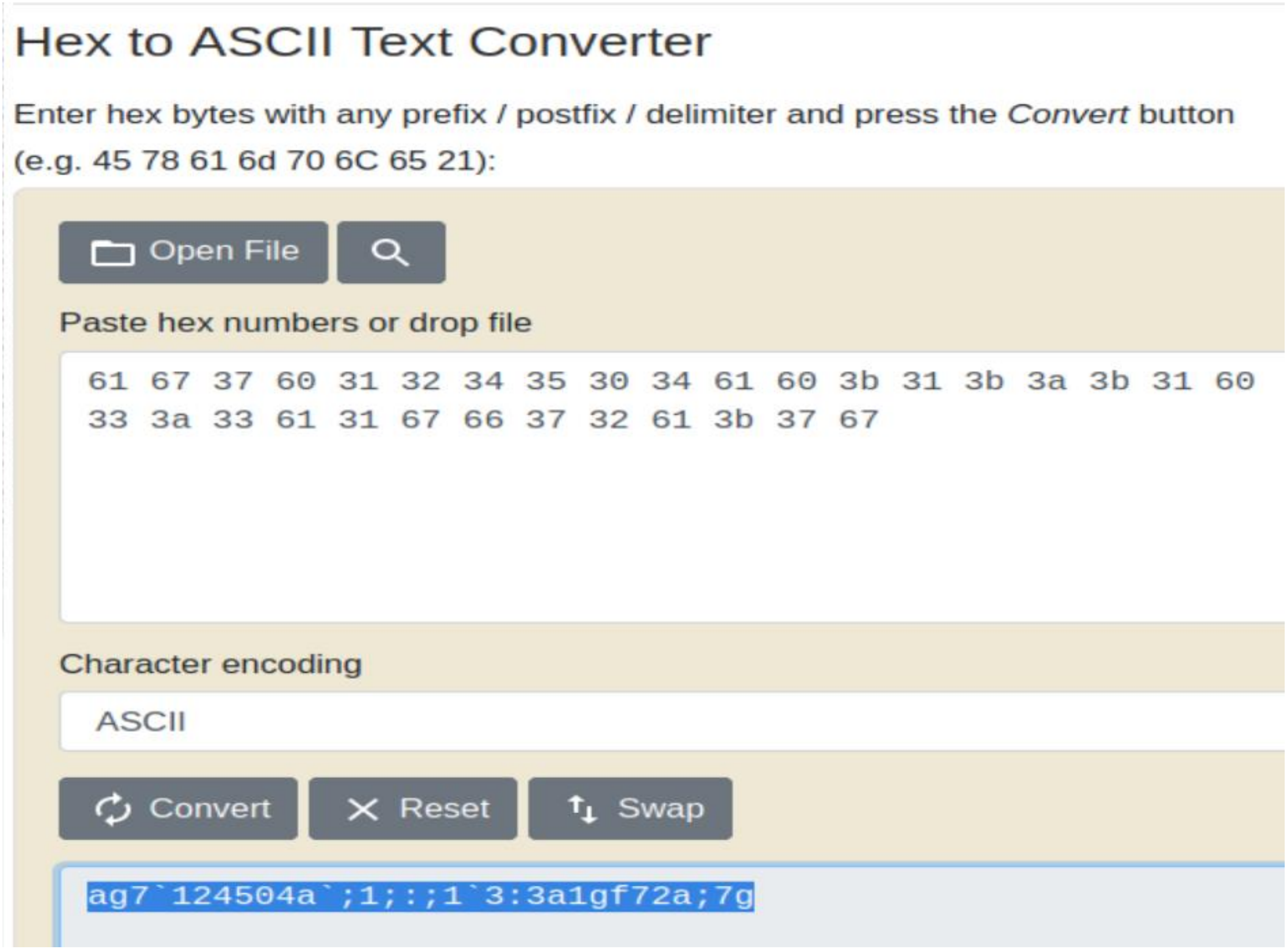
```
f 48 89 95 38 fe ff ff 48 b8 31
7 67 48 ba 37 3a 31 35 67 3b 3b
e ff ff 48 89 95 48 fe ff ff c6
0 48 b8 61 67 37 60 31 32 34 35
0 3b 31 3b 3a 48 89 85 60 fe ff
e ff ff 48 b8 3b 31 60 33 3a 33
5 37 32 61 3b 37 67 48 89 85 70
```

```
89 85 30 fe ff ff 48 89 95 38 fe ff ff 48 b8
61 30 61 67 31 37 67 48 ba 37 3a 31 35 67 3b
60 48 89 85 40 fe ff ff 48 89 95 48 fe ff ff
85 50 fe ff ff 00 48 b8 61 67 37 60 31 32 34
48 ba 30 34 61 60 3b 31 3b 3a 48 89 85 60 fe
ff 48 89 95 68 fe ff ff 48 b8 3b 31 60 33 3a
61 31 48 ba 67 66 37 32 61 3b 37 67 48 89 85
fe ff ff 48 89 95 78 fe ff ff c6 85 80 fe ff
00 48 b8 31 33 38 31 64 35 33 35 48 ba 37 37
```

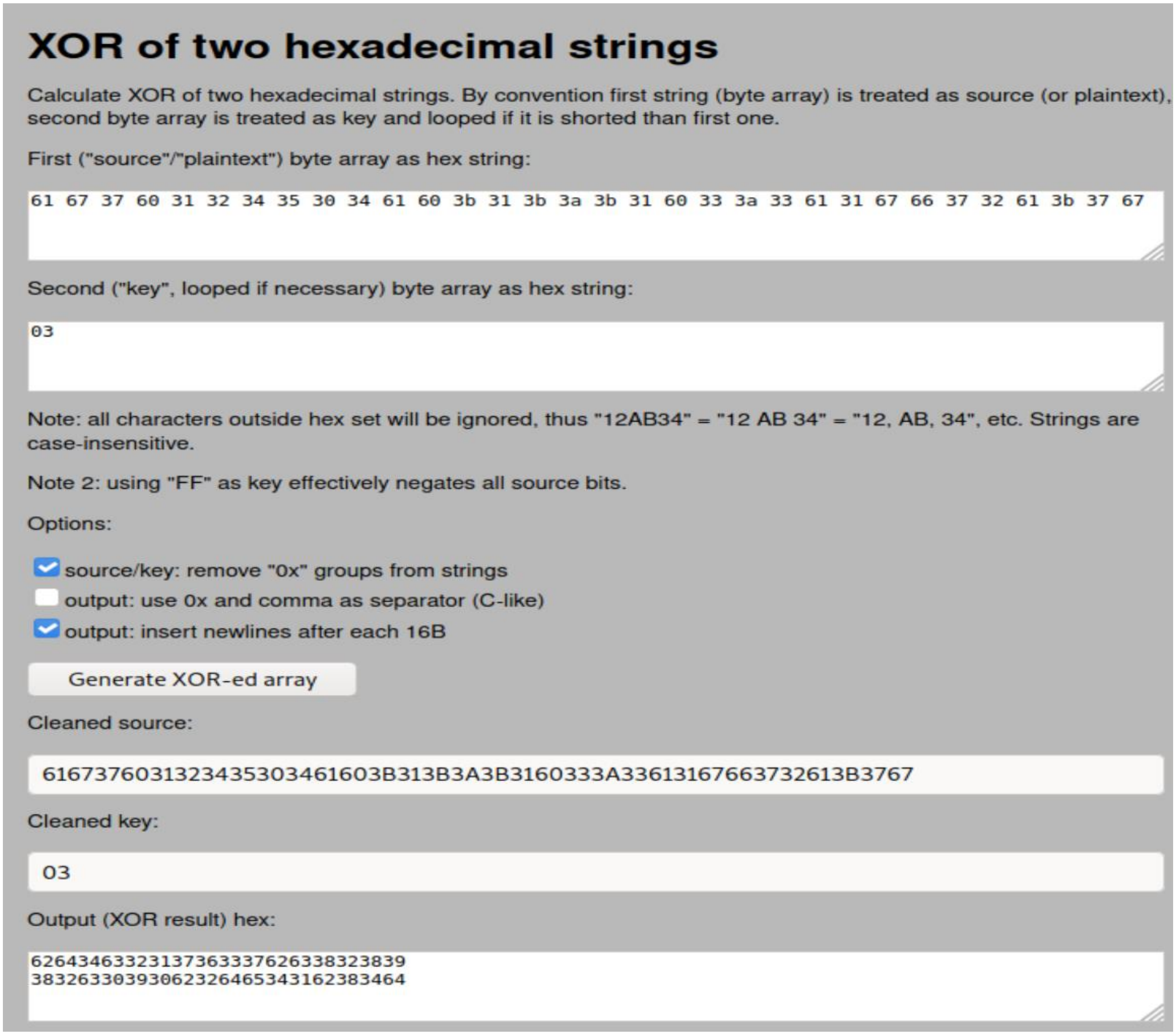
```
60 48 89 85 40 fe ff ff 48 89 95 48 fe ff ff c6
85 50 fe ff ff 00 48 b8 61 67 37 60 31 32 34 35
48 ba 30 34 61 60 3b 31 3b 3a 48 89 85 60 fe ff
ff 48 89 95 68 fe ff ff 48 b8 3b 31 60 33 3a 33
61 31 48 ba 67 66 37 32 61 3b 37 67 48 89 85 70
fe ff ff 48 89 95 78 fe ff ff c6 85 80 fe ff ff
00 48 b8 31 33 38 31 64 35 33 35 48 ba 37 37 38
```

```
60 48 89 85 40 fe ff ff 48 89 95 48 fe ff ff c6
85 50 fe ff ff 00 48 b8 61 67 37 60 31 32 34 35
48 ba 30 34 61 60 3b 31 3b 3a 48 89 85 60 fe ff
ff 48 89 95 68 fe ff ff 48 b8 3b 31 60 33 3a 33
61 31 48 ba 67 66 37 32 61 3b 37 67 48 89 85 70
fe ff ff 48 89 95 78 fe ff ff c6 85 80 fe ff ff
00 48 b8 31 33 38 31 64 35 33 35 48 ba 37 37 38
30 39 32 38 36 48 89 85 90 fe ff ff 48 89 95 98
```

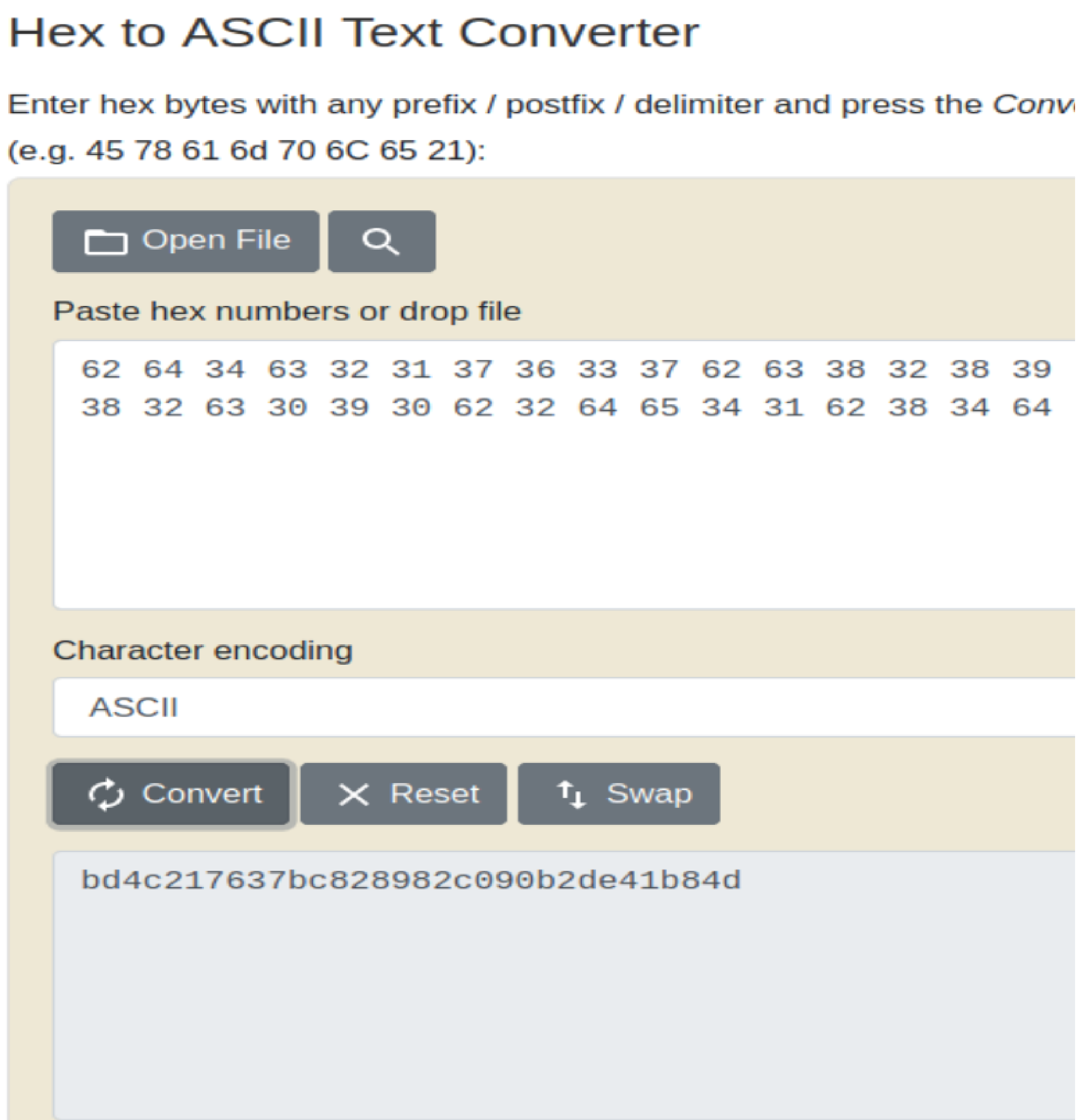
The data values are stored in little endian format which means that the least significant bytes are stored before the most significant bytes.



Plugging these values into a Hex to ASCII converter we are able to see the un-hashed password.



Here we take the 32 bytes stored in memory and XOR each one of them with the value 3 in order to derive our password.



We then convert this calculated value from Hex to ASCIII and compare iit with our earlier findings or in the event that ltrace failed we would test our resultant ASCII string on crackme.

In conclusion we were able to demonstrate two methods for solving this crackme. We first used ltrace in order to see our password in clear text. This was able to provide a correct result, howwever it left us with a shallow understand of how crackme worked. We then utilitized Ghidra to explore deeper into the crackme and discovered a deeper understand of how crackme functioned as a whole. We then we able to determine what the unhashed password was from memory and then calculate its hashed value which allowed us to correctly solve the crackme.