Flame… Yeah, this one wasn't too bad all things considered. The greatest difficulty one might encounter with the challenge at first glance would be the fact that it was a PowerPC binary. If you understand any other assembly, then you know that most other languages follow a general form. Given that the PowerPC binary is compiled from C, then you can make a determination that the architecture works very similarly to x86.

So, before actually taking the binary apart, let's see how it runs. Being that it is a PowerPC binary, I ran it in a QEMU emulator.

```
$ qemu-ppc flame_4a1f5a4d2eedfe718fee7eff429bb7e63a0c1b69
*************************************
*                                   *
*   HITCON CTF 2016 Flag Verifier   *
*                                   *
*************************************
Check your flag before submission: plz_give_me_flag_:)
Your flag is incorrect :(
$
```

Okay… so, now we know that the flag is not *plz_give_me_flag_:)*

Time to open up IDA! (Since BinaryNinja doesn't support PowerPC binaries yet.)

```
.text:1000078C                 stwu      r1, -0x1B0(r1)
.text:10000790                 mflr      r0
.text:10000794                 stw       r0, 0x1B0+arg_4(r1)
.text:10000798                 stw       r31, 0x1B0+var_4(r1)
.text:1000079C                 mr        r31, r1
.text:100007A0                 lwz       r8, -0x7008(r2)
.text:100007A4                 stw       r8, 0x19C(r31)
.text:100007A8                 li        r8, 0
.text:100007AC                 lis       r9, unk_1007899C@ha
.text:100007B0                 addi      r10, r9, unk_1007899C@l
.text:100007B4                 addi      r9, r31, 0xAC
.text:100007B8                 mr        r8, r10
.text:100007BC                 li        r10, 0x8C
.text:100007C0                 mr        r5, r10
.text:100007C4                 mr        r4, r8
.text:100007C8                 mr        r3, r9
.text:100007CC                 bl        memcpy
.text:100007D0                 lis       r9, asc_100788B4@ha # "*************************************"
.text:100007D4                 addi      r3, r9, asc_100788B4@l # "*************************************"
.text:100007D8                 bl        puts
.text:100007DC                 lis       r9, asc_100788DC@ha # "*                                   *"
.text:100007E0                 addi      r3, r9, asc_100788DC@l # "*                                   *"
.text:100007E4                 bl        puts
.text:100007E8                 lis       r9, aHitconCtf2016F@ha # "*   HITCON CTF 2016 Flag Verifier   *"
.text:100007EC                 addi      r3, r9, aHitconCtf2016F@l # "*   HITCON CTF 2016 Flag Verifier   *"
.text:100007F0                 bl        puts
.text:100007F4                 lis       r9, asc_100788DC@ha # "*                                   *"
.text:100007F8                 addi      r3, r9, asc_100788DC@l # "*                                   *"
.text:100007FC                 bl        puts
.text:10000800                 lis       r9, asc_100788B4@ha # "*************************************"
.text:10000804                 addi      r3, r9, asc_100788B4@l # "*************************************"
.text:10000808                 bl        puts
.text:1000080C                 lis       r9, aCheckYourFlagB@ha # "Check your flag before submission: "
.text:10000810                 addi      r3, r9, aCheckYourFlagB@l # "Check your flag before submission: "
.text:10000814                 crclr     4*cr1+eq
.text:10000818                 bl        printf
.text:1000081C                 addi      r9, r31, 0x138
.text:10000820                 mr        r4, r9
.text:10000824                 lis       r9, aS@ha     # "%s"
.text:10000828                 addi      r3, r9, aS@l  # "%s"
.text:1000082C                 crclr     4*cr1+eq
.text:10000830                 bl        __isoc99_scanf
.text:10000834                 addi      r9, r31, 0x138
.text:10000838                 mr        r3, r9
.text:1000083C                 bl        strlen
.text:10000840                 mr        r9, r3
.text:10000844                 cmplwi    cr7, r9, 0x23
.text:10000848                 bne       cr7, loc_10000958
.text:1000084C                 li        r3, 0x1E61
```

IDA loads up the binary and jumps to the main. Upon first inspection, we can see a lot of the strings we are seeing when we run the program. Continuing through the program, we see that it will take input and call *strlen*. Most of the rest of the program will be unintelligible until we understand the the purpose of the various registers and the opcodes being called.

PowerPC guides I used for this challenge are:
http://www.ds.ewi.tudelft.nl/vakken/in1006/instruction-set/
http://cache.freescale.com/files/product/doc/MPC82XINSET.pdf?fasp=1
http://www.csd.uwo.ca/~mburrel/stuff/ppc-asm.html

After skimming through these, I was able to immediately recognize a few things happening at the very beginning of main. For one, we call *mr r31, r1* to set r31 to be r1, which allows us to see that r31 is being used as our stack pointer inside main. We can verify that this is the case when we the program calls *__isoc99_scanf* and *strlen*.

```
addi      r9, r31, 0x138
mr        r4, r9
lis       r9, aS@ha     # "%s"
addi      r3, r9, aS@l  # "%s"
crclr     4*cr1+eq
bl        __isoc99_scanf
addi      r9, r31, 0x138
mr        r3, r9
bl        strlen
```

It gets an offset from the base of the stack and puts that value in r9 before moving r9 into r4, which is an argument to *scanf*. Following the call to *scanf*, we see it gets that same offset and moves it into r3 as an argument to *strlen*.

```
bl        strlen
mr        r9, r3
cmplwi    cr7, r9, 0x23
bne       cr7, loc_10000958
```

After *strlen* is called, the return value of the function is put in r3. The value in r3 is moved into r9, then a compare is called between r9 and 0x23 (int 35). The status flags of the comparison are put in cr7. So when the program calls *bne*, it takes the status flags in cr7 and will jump to the specified location if the results of the previous comparison finds that the first argument is not equal to the second argument. If our input is not 35 characters in length, then we will go to fail.

```
li          r3, 0x1E61
bl          srandom
li          r9, 0
stw         r9, 0x18(r31)
b           loc_100008DC
```

```
loc_100008DC:
lwz         r9, 0x18(r31)
cmpwi       cr7, r9, 0x22
ble         cr7, loc_10000860
```

If we pass the character count check, we begin by calling srandom with 0x1e61 as an argument in r3. This will seed the random number generation which will occur later in the program. After srandom, 0 is moved into r9 and the value in r9 is stored at [r31+0x18]. The value at r31+0x18 is going to be our loop counter. As we see in this second part after the branch, we load the value at r31+0x18 into r9 and then compare it to 0x22 (int 34). While we are less than or equal to 34, we will execute the code at loc_10000860.

This next part is where it gets tricky. For the sake of clarity, I will split it up into 4 sections.

```
bl          rand
mr          r9, r3
clrlwi      r10, r9, 20
lwz         r9, 0x18(r31)
slwi        r9, r9, 2
addi        r8, r31, 0x1A0
add         r9, r8, r9
addi        r9, r9, -0x180
stw         r10, 0(r9)
```

In this first section, rand is called and the result is moved into r9. This next opcode was a bit confusing, but after reading through a couple of times, I discerned that it took the second argument (currently r9) and cleared the first x bits, where x is the third argument (currently 20), of r9 and stores the result in r10. As an example, if I had 00010010101011011011001010000001 (int 313373313) in r9 and called *clrlwi r10, r9, 20* the result that would be put in r10 would be 00000000000000000000001010000001 (int 641). Following this, the current loop counter is loaded into r9, then *slwi* is called on r9, which performs a left shift on r9 by the amount given as the third argument. So this effectively multiplies the value in r9 by 4. We then put another stack location in r8, r31+0x1a0 this time. We add the result of r9 * 4 to the location to in r8 and then subtract from this value 0x180. The result is a location on the stack which is 4 bytes in length between the location that will be referenced on the next iteration. This is how we are getting locations in an array on the stack. At the end of this section, we move the value in r10, which is the result of our call on clrlwi on the value returned by rand, to the location at r9.

```
lwz      r9, 0x18(r31)
slwi     r9, r9, 2
addi     r10, r31, 0x1A0
add      r9, r10, r9
addi     r9, r9, -0x180
lwz      r9, 0(r9)
mr       r8, r9
```

This second section is a bit redundant, so I will be brief. It just loads the value that was just stored into r8 by retrieving the value from the array by performing the same set of instructions used to store it in the first place.

```
addi     r10, r31, 0x138
lwz      r9, 0x18(r31)
add      r9, r10, r9
lbz      r9, 0(r9)
xor      r9, r8, r9
mr       r10, r9
```

This next section puts the location of our input into r10 (r31+0x138). We then put the value of the loop counter into r9. We add these two together to get the character location of the character corresponding to our current iteration. We then load that character value into r9 with the *lbz* opcode (load byte zero extend). We xor this character value with the value in r8, which is the value from rand after the first 20 bits have been cleared, and store the result in r9. We then put the result in r10 because why not.

```
lwz      r9, 0x18(r31)
slwi     r9, r9, 2
addi     r8, r31, 0x1A0
add      r9, r8, r9
addi     r9, r9, -0x180
stw      r10, 0(r9)
lwz      r9, 0x18(r31)
addi     r9, r9, 1
stw      r9, 0x18(r31)
```

In this next section, we go back to the array on the stack again, except this time, we will store at the location derived from the loop counter the result of the xor. After this, we increment our loop counter by 1 and then continue for another 34 times.

So, after this loop, we will have a series of values stored on the stack. These values will be the result of xoring, one character at a time, our input with the 35 values that would be generated by calling rand with the seed 0x1e61 and clearing the first 20 bits of each one.

During the next part, we will start our answer verification loop.

```
li       r9, 0
stw      r9, 0x1C(r31)
b        loc_10000938

lwz      r9, 0x1C(r31)
cmpwi    cr7, r9, 0x22
ble      cr7, loc_100008F4
```

These parts set up the loop and will branch to a segment which tells you your flag is correct if you manage to iterate through the loop at least 35 times. The verification section is as follows:

```
lwz       r9, 0x1C(r31)
slwi      r9, r9, 2
addi      r10, r31, 0x1A0
add       r9, r10, r9
addi      r9, r9, -0xF4
lwz       r10, 0(r9)
lwz       r9, 0x1C(r31)
slwi      r9, r9, 2
addi      r8, r31, 0x1A0
add       r9, r8, r9
addi      r9, r9, -0x180
lwz       r9, 0(r9)
cmpw      cr7, r10, r9
bne       cr7, loc_10000960
```

Firstly, we retrieve a value from another array located on the stack using the loop counter (r31+0x1c) to index into it. This array happens to be our answer key. It gets moved onto the stack around the beginning of main. If you are looking at the screenshot of IDA at the very beginning of the writeup, it is labeled unk_1007899. If we go to this location, you can find an array of 4 bytes values. A value from this array is loaded into r10. During this next part of the code, we get the location of the result array of our earlier calculation and retrieve a value from it using the loop counter to index into it. This value is placed in r9. At the end, we compare r9 with r10. If they are not equal, then the loop quits and it prints you lose. If it doesn't do this 35 times, then we win.

So to solve this next part, I wrote a simple c program to give me the values that would be generated by 35 successive calls to rand with the seed 0x1e61 (int 7777). The program is as follows:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
int main()  /* Random number generator*/
{
 int i, n; /*initialize variables*/
 n = 40;   /*amount of numbers to be printed*/
 srand(7777);  /*initialized the random number generator*/
  for (i = 0; i < n; i++)
 {
   printf("%ld\n", random());
 }
 return(0);
}
```

Next, I extracted the values from memory that we check as our answer key. Using these values, I wrote a simple python script to brute find the characters that when xored with the one of the rand values with its first 20 bits cleared would give me the corresponding value in the answer key. The script is as follows:

```
def hashPPC(seedVal, inputVal):
    cleared = clear(seedVal)
    res = cleared ^ ord(inputVal)
    return res

def clear(num):
    binum = str(bin(num))[2:]
    res = '00000000000000000000'+pad(binum)[20:]
    return int(res,2)

def pad(strNum):
    res = strNum
    while(len(res) < 32):
        res = '0'+res
    return res

def getAnswer(index):
    alphanum =
'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789_{}'
    rando =
[205757590,998377520,1430092073,2047191058,1959523426,1763442792,1345239717,7933
58328,658433361,1016199565,1294268491,1322964720,2137247737,1405325116,642114049
,1392987601,988381069,740379636,1285459211,1904974096,943865137,605738913,155990
9246,926847391,1442292648,425531748,1307965978,1673880289,860617914,1317232,8318
69049,1066375504,999694753,114477475,966082914]
    check =
[3326,2137,2397,2161,1037,6,2782,4008,1377,2522,2168,1666,4009,3935,606,3504,4031,301
4,3384,2397,3337,2029,775,448,921,2390,2629,658,3210,2351,74,2404,404,2522,287]
    for i in alphanum:
        test = hashPPC(rando[index],i)
        print("Test character: "+i+"          ")
        print("HashPPC Result: "+str(test)+"\n")
        if(test == check[index]):
            print("Hit: "+i)
            return i
    print("You lose!... Good day, sir!")
    return 'nope'

def getAnswers():
    res = ''
    for i in range(0,35):
        retChar = getAnswer(i)
        if(retChar == 'nope'):
```

```
            print("Game over...")
            exit(0)
        else:
            res += retChar
    print(res)
    return

getAnswers()
```

Running this gives us the flag: hitcon{P0W3rPc_a223M8Ly_12_s0_345y}
We can verify that this is answer by plugging it back into the program.

```
$ qemu-ppc flame_4a1f5a4d2eedfe718fee7eff429bb7e63a0c1b69
************************************
*                                  *
*   HITCON CTF 2016 Flag Verifier  *
*                                  *
************************************
Check your flag before submission: hitcon{P0W3rPc_a223M8Ly_12_s0_345y}
Good job!! now you can submit your flag :)
```

Winner!