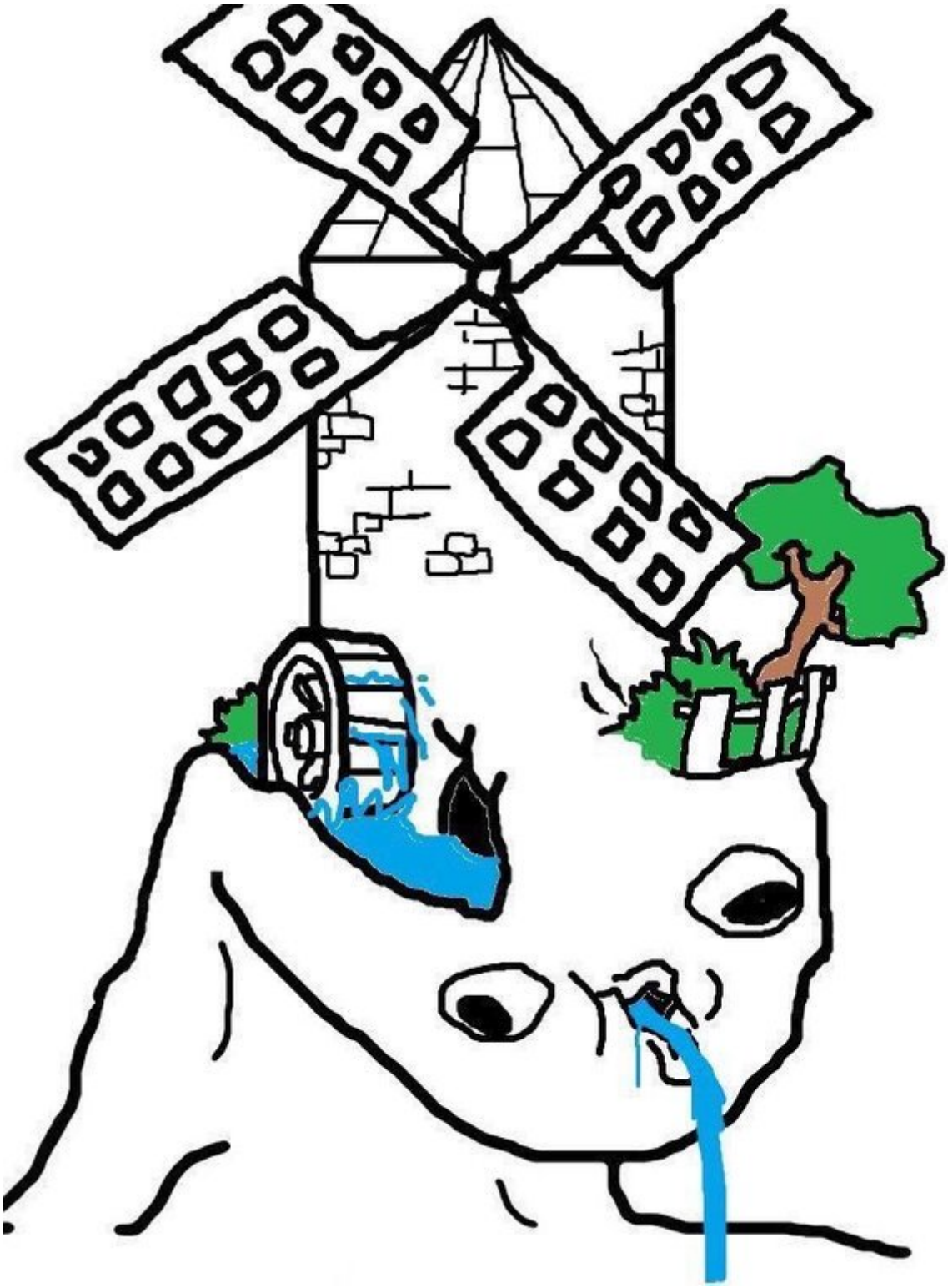
What is it?

A program that will make you hate Java if you didn't already. Effectively it's a C/C++ program that uses JNI to run Java functions. I want to thank Jinmo for being based as fuck and giving us JNI structures that I could slap into IDA: https://gist.githubusercontent.com/Jinmo/048776db75067dcd6c57f1154e65b868/raw/89af26807eaa8bb31e35da63e102b0abfa311580/jni_all.h - this saved me from reaching the following vegetative state:



How to solve it?

The program takes a parameter (which is the password) and then performs two checks that we have to pass:

```
__int64 __fastcall HandleStuff(JNIEnv_ *env, __int64 password)
{
  __int64 result; // rax
  int v3; // eax
  int i; // [rsp+1Ch] [rbp-4h]

  if ( (unsigned int)check1(env, password) || (unsigned int)check2(env, password) )
  {
    print((__int64)env, (__int64)"No coffee for you!");
    result = 1LL;
  }
  else
  {
    print((__int64)env, (__int64)"Access granted.");
    for ( i = 0; coffemug[i]; ++i )
    {
      v3 = i;
      print((__int64)env, (__int64)coffemug[v3]);
    }
    print((__int64)env, (__int64)"Enjoy!");
    print((__int64)env, (__int64)"================================================");
    printstream((__int64)env, (__int64)"Also here is your flag: HTB{");
    *(_BYTE *)(enc_flag_len + flaglenmaybe + password) = 0;
    printstream((__int64)env, password);
    print((__int64)env, (__int64)"}");
    result = print((__int64)env, (__int64)"================================================");
  }
  return result;
}
```

Let's start with check1!

First two important things:

```

  // gets the encrypted flag
  encrypted_flag = get_bytearrays(1);
  sets_flag_bytes_to_var_in_jvm(env, (__int64)encrypted_flag);

  // gets some other bytes
  ptr_to_bytearray2 = get_bytearrays(2);
  sets_shorts_bytes_to_something_in_jvm(env, (__int64)ptr_to_bytearray2);

```

The first thing uses JNI to swap what values characters correspond to if you cast them to a byte, and the second one does a the same meme but with short values (2 byte). I.e. this function just swaps shit around to fuck with you:

```
__int64 __fastcall sub_11B2(JNIEnv_ *env, __int64 a2)
{
  __int64 result; // rax
  JNIEnv_ *v3; // [rsp+8h] [rbp-38h]
  int i; // [rsp+1Ch] [rbp-24h]
  __int64 v5; // [rsp+20h] [rbp-20h]
  __int64 v6; // [rsp+28h] [rbp-18h]
  __int64 v7; // [rsp+30h] [rbp-10h]
  __int64 v8; // [rsp+38h] [rbp-8h]

  v3 = env;
  v5 = ((__int64 (__fastcall *)(JNIEnv_ *, const char *))env->functions->FindClass)(env, "java/lang/Short");
  v6 = ((__int64 (__fastcall *)(JNIEnv_ *, __int64, const char *, const char *))v3->functions->GetStaticMethodID)(
         v3,
         v5,
         "valueOf",
         "(S)Ljava/lang/Short;");
  result = ((__int64 (__fastcall *)(JNIEnv_ *, __int64, const char *, const char *))v3->functions->GetFieldID)(
             v3,
             v5,
             "value",
             "S");
  v7 = result;
  for ( i = 0; i <= 255; ++i )
  {
    v8 = ((__int64 (__fastcall *)(JNIEnv_ *, __int64, __int64, _QWORD))env->functions->CallStaticObjectMethod)(
           env,
           v5,
           v6,
           (unsigned int)(char)i);
    result = ((__int64 (__fastcall *)(JNIEnv_ *, __int64, __int64, _QWORD))env->functions->SetShortField)(
               env,
               v8,
               v7,
               (unsigned int)*(char *)(i + a2));
  }
  return result;
}
```

With the structures, this should be relatively easy to see. Next we return to check1 and the following happens:

- We extract a .class file from the file
- We register the class to use its functions
- We take 0x1e bytes of the password
- We take some other bytes from the file
- We pass both the password bytes and the other bytes into the class and call its main function
- The function performs the check

Some of this can be seen here:

```
41
42  Verify1Class = (env_cpy->functions->DefineClass)(env_cpy, "Verify1", 0LL, class_data, class_data_len);
43
44  newObjArrayFunc = env->functions->NewObjectArray;
45  javaStrObject = (env_cpy->functions->NewStringUTF)(env_cpy, &nullstr);
46  JavaStringClass = (env_cpy->functions->FindClass)(env_cpy, "java/lang/String");
47  StringArrayObject = (newObjArrayFunc)(env, 2LL, JavaStringClass, javaStrObject);
48
49  NewPasswordSTringObject = (env_cpy->functions->NewStringUTF)(env_cpy, password);
50
51
52  passwordStrObj = NewPasswordSTringObject;
53  StringClassProbably = (env_cpy->functions->GetObjectClass)(env_cpy, NewPasswordSTringObject);
54
55  strClass = StringClassProbably;
56  SubstringMethod = (env_cpy->functions->GetMethodID)(
57                      env_cpy,
58                      StringClassProbably,
59                      "substring",
60                      "(II)Ljava/lang/String;");
61
62  substrMethod = SubstringMethod;
63  setObjArrayElem = env->functions->SetObjectArrayElement;
64
65  PasswordSubstringFrom0To0x1e = (env_cpy->functions->CallObjectMethod)(
66                                    env_cpy,
67                                    passwordStrObj,
68                                    SubstringMethod,
69                                    0LL,
70                                    enc_flag_len);
71
72
73  // takes substr of password from 0 to 0x1e, then sets the stringobject to it
74  (setObjArrayElem)(env, StringArrayObject, 0LL, PasswordSubstringFrom0To0x1e);
75
76  SetObjArrayElement = env->functions->SetObjectArrayElement;
77  NewString = (env_cpy->functions->NewStringUTF)(env_cpy, &check1_verification_probably);
78  (SetObjArrayElement)(env, StringArrayObject, 1LL, NewString);
```

Ok, so I just extracted Verify1 (class) bytes, and slapped them into an online decompiler, we get:

```java
//
// Decompiled by Procyon v0.5.36
//

public class Verify1
{
    private static boolean compareByte(final Byte b, final Short n) {
        return b == (short)n;
    }

    public static void main(final String[] array) {
        if (array == null || array.length != 2) {
            System.out.println("Verifying requires source and target");
            System.exit(1);
            return;
        }
        final String s = array[0];
        final String s2 = array[1];
        if (s == null || s2 == null) {
            System.out.println("Source and Target may not be null");
            System.exit(2);
            return;
        }
        if (s.length() != s2.length()) {
            System.out.println("Source and Target don't have the same length");
            System.exit(2);
            return;
        }
        System.out.println("Verifying user is of terrestrial origin...");
        for (int i = 0; i < s.length(); ++i) {
            if (!compareByte((byte)s.charAt(i), (short)(byte)s2.charAt(i))) {
                System.out.println("=> User might be an alien!!!");
                System.exit(3);
                return;
            }
        }
    }
}
```

Cool, it just compares if the password byte value corresponding to the short value of the random bytes it supplies as the second argument. But remember that it fucked with the values earlier? Yep, we have to write a script. This will be supplied as solvepart1.py. Effectively what we do is find what we are comparing against by indexing the swapped short values. Using that fact, we can reverse the swapping done for byte values and get: **th3_s3cr3t3_r3c1p3_1s_23_h0ur5_str41ght_0f_r34d1ng_J4v4_d0c5**. Time to get the second half of the flag.

Check2 does even more memes that are quite similar but a bit more complex to check1. First thing it does is swap boolean values, as we can see here:

```
1  __int64 __fastcall SetBooleanFields(JNIEnv_ *a1)
2  {
3    JNIEnv_ *env; // [rsp+8h] [rbp-38h]
4    __int64 BooleanClass; // [rsp+18h] [rbp-28h]
5    __int64 Boolean.FALSE; // [rsp+20h] [rbp-20h]
6    __int64 Boolean.TRUE; // [rsp+28h] [rbp-18h]
7    __int64 truefield; // [rsp+30h] [rbp-10h]
8    __int64 falsefield; // [rsp+38h] [rbp-8h]
9
10   env = a1;
11   BooleanClass = (a1->functions->FindClass)(a1, "java/lang/Boolean");
12   Boolean.FALSE = (env->functions->GetStaticFieldID)(env, BooleanClass, "FALSE", "Ljava/lang/Boolean;");
13   Boolean.TRUE = (env->functions->GetStaticFieldID)(env, BooleanClass, "TRUE", "Ljava/lang/Boolean;");
14   truefield = (env->functions->GetStaticObjectField)(env, BooleanClass, Boolean.TRUE);
15   falsefield = (env->functions->GetStaticObjectField)(env, BooleanClass, Boolean.FALSE);
16
17   (env->functions->SetStaticObjectField)(env, BooleanClass, Boolean.FALSE, truefield);
18
19   return (env->functions->SetStaticObjectField)(env, BooleanClass, Boolean.TRUE, falsefield);
20 }
```

It literally sets false to be true, and true to be false. It then sets up some other byte swapping that fucks with the numerical values of characters. It also does another meme where it changes the System.exit() function call here:

```
IDA View-A | Pseudocode-B | Pseudocode-A | Pseudocode-C | Local Types | Pseudoc
1 __int64 __fastcall registerShutdown(JNIEnv_ *env, __int64 sub_must_be_0_set)
2 {
3   __int64 class_1; // rax
4
5   someFuncPtrs = sub_must_be_0_set;
6   class_1 = (env->functions->FindClass)(env, "java/lang/Shutdown");
7   return (env->functions->RegisterNatives)(env, class_1, nativemethods, 2LL);
8 }
```

This sets the call to be this function:

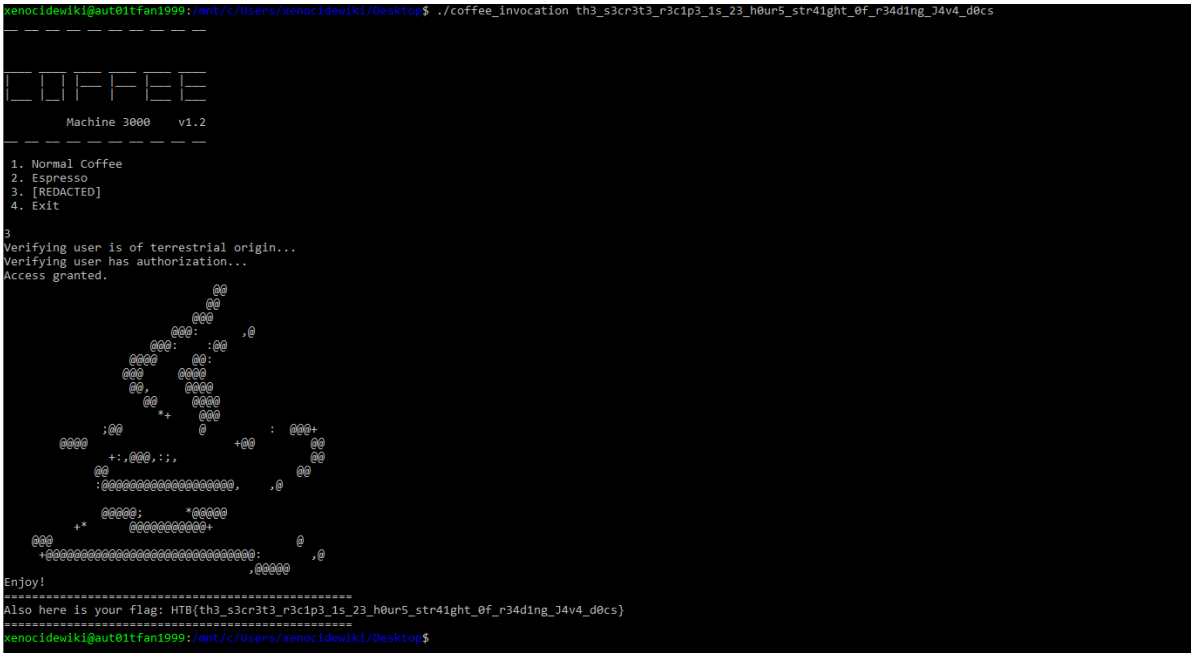
```
1 __int64 __fastcall fucktheasciitableagain(JNIEnv_ *a1, __int64 a2, int a3)
2 {
3   __int64 result; // rax
4   void *v4; // rax
5
6   if ( a3 <= 2 )
7   {
8     result = must_be_0;
9     if ( a3 > must_be_0 )
10    {
11      result = a3;
12      must_be_0 = a3;
13    }
14  }
15  else
16  {
17    if ( a3 > must_be_0 && a3 - must_be_0 == 1 )
18      must_be_0 = a3;
19    if ( must_be_0 == num_15 + 2 && a3 == num_15 + 2 )
20      must_be_0 = 0;
21    result = num_15;
22    if ( a3 - 2 < num_15 )
23    {
24      v4 = get_bytearrays(a3 + 1);
25      result = copy_ascii_table(a1, v4);
26    }
27  }
28  return result;
29 }
```

Which tl;dr does some checks and swaps bytes around even more (will be used in the java class). SOOOOOO, the program then gets another class called Verify2 which does this:

</> check2.class

```
import java.util.stream.Collector;
import java.util.stream.Collectors;
import java.util.function.Function;
import java.util.Arrays;

//
// Decompiled by Procyon v0.5.36
//

public class Verify2
{
    private static boolean compareByte(final Byte b, final Short n) {
        System.out.println(invokedynamic(makeConcatWithConstants:(II)Ljava/lang/String;, (byte)b, (short)n));
        return b == (short)n;
    }

    private static String complexSort(final String s, final Boolean b) {
        final Object[] array = s.chars().mapToObj(n -> Character.valueOf((char)n)).toArray();
        if (b) {
            Arrays.sort(array);
        }
        return Arrays.stream(array).map((Function<? super Object, ?>)Object::toString).collect((Collector<? super
Object, ?, String>)Collectors.joining());
    }

    private static Boolean verifyPassword(final String s, final String anObject) {
        return s.equals(anObject);
    }

    public static void main(final String[] array) {
        if (array == null || array.length != 1) {
            System.out.println("Verifying requires source");
            System.exit(1);
            return;
        }
        final String s = array[0];
        if (s == null) {
            System.out.println("Source may not be null");
            System.exit(1);
            return;
        }
        if (s.length() % 2 != 0) {
            System.out.println("Length must be even");
            System.exit(1);
            return;
        }
        System.out.println("Verifying user has authorization...");
        for (int i = 0; i < s.length() / 2; ++i) {
            if (complexSort(s.substring(i * 2, i * 2 + 2), true).equals(complexSort("Cr1KD5mk0_uUzQYifaGVqlN2B3wvp
gPtSx6Odo{8hjJLHy9IXb4RnWZ}TAFEsMce7", false).substring(i * 2, i * 2 + 2))) {
                System.exit(i + 3);
            }
        }
        if (!verifyPassword(s, "Tinfoil")) {
            System.out.println("Please enter the correct password");
            System.exit(2);
        }
    }
}
```

And as you can see, there's a couple boolean things here and there which will be swapped, the System.exit(i+3) will swap the value tables at every iteration of the loop etc. etc. What this means is that every 2 characters will need a different substitution table that is stored in the file. What we can do is to write a script which simulates and reverses this. Basically, that random string **Cr1KD5mk...** gets sorted, then we do a few comparisons and reversals which are scuffed as fuck, solvepart2.py does this. My ingenious script is broken, and gives me **_str41ght_0f_r34d1ng_J4v4_d0**, missing the last few characters, I just went and guessed those to be **cs**, giving us **_str41ght_0f_r34d1ng_J4v4_d0cs**. Beep boop, we can now yeet the password into the program and:



FLAG: HTB{th3_s3cr3t3_r3c1p3_1s_23_h0ur5_str41ght_0f_r34d1ng_J4v4_d0cs}