crackmes.one : noverify's GraxCode's Java CrackMe 1

01 May, 2018 10:11

Before anything else, I must say I'm bad at JAVA.
I actually hate JAVA with passion and I suck at JAVA reverse engineering.

There. Now let's do this shit.
To solve this crackme I'll use **Eclipse** (**Kepler**, because I couldn't make the debug plugin work under **Oxygen**) with the **Bytecode visualizer** plugin for tracing around and **Bytecode Viewer** for static analysis and decompilation of the classes.

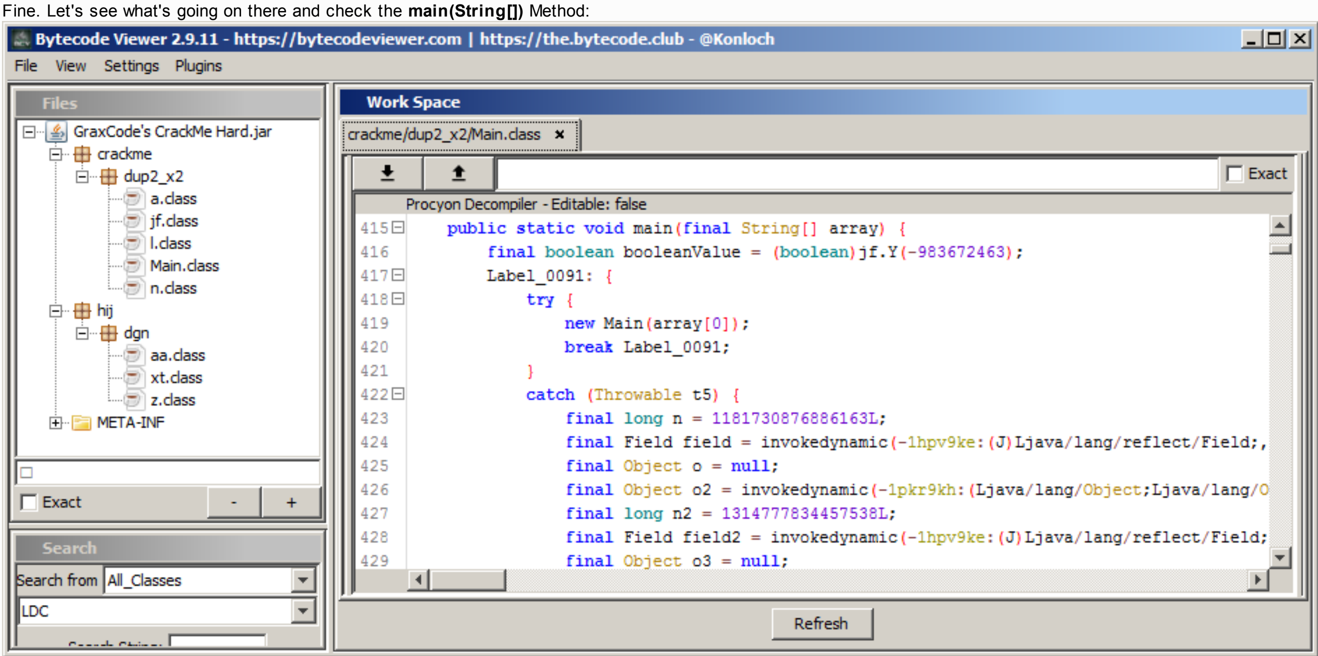
Get familiar with the target

The challenge is a console application written in JAVA.
It comes in a **.JAR package**, with the following hint by its author:

description

```
The goal is to find a valid key to input.
Keys are 32-bit integers. (eg. 123456).
java -jar "GraxCode's CrackMe Hard.jar" key
```

Fine. Let's see what's going on there and check the **main(String[])** Method:



No surprises here - the crackme is obfuscated, as the crackme name says by **GraxCode's obfuscator**, and it would be great if I knew that thing exists in the first place, but as usual I learned about it, right after I solved the crackme itself.
Oh well, let's defeat the obfuscation first.

Defeating GraxCode's obfuscation

Now, I know there's some JAVA wizards out there, that will dynamically load the crackme's .JAR, install some clever hooks and solve the whole thing in 10 minutes.
Well, I'm not one of you. I'm an idiot, and here's how I proceed.

There aren't many classes in the JAR, which is good so I started looking around to see if the decompilers will be of any help here.
Truth is, depending on the decompilation module used, a large portions of the classes were being decompiled to at least partially working code, which is kind of ok?
However, it turned out that **z.class** is not obfuscated at all, and judging by its code, it is a hash map manager, used for caching stuff.

The rest of the classes were obfuscated and didn't looked clear to me, so I fired up Eclipse and started setting up a **dummy crackme project**.
I copied all of the method declarations with their return types and input parameters and put those in my new project, to later mimic the code flow.

While doing that I noticed something interesting. Four of the classes **l.class**, **n.class** from the **crackme.dup2_x2** package plus, **aa.class** and **xt.class** from the **hij.dgn** package had the same structure:

l, n, aa and xt CLASS structure

```
public class class_name extends Thread {

   class_name(int arg0) { }

   public void run() { }

   private static final void method_1(int arg0, Object arg1) { }

   private static final int method_2(int arg0, int arg1) { }
```

```
    private static final int method_3(byte[] arg0, int arg1) { }

    private static final void method_4() { }

    private static final void method_5() { }

    static final String method_6(Object arg0) { }

}
```

The only difference in their code were their method names and constant values. That's good news for me, because reverse engineering any of them means I've reverse engineered them all.

Another thing that's also kind of important was the **<clinit>**, or static initialization code that was present in **Main**, **a** and **jf** classes. This code will be executed the first time any of these classes is called, so I'll for sure have to deal with them at some point.
Starting with the static initialization of **Main.class**:

Main.class, static init

```
public static  { // <clinit> //()V
    ldc "\u1270\uB5A4\u6482\u457F\uCA09\u4378\u13D5\uA7EE - snip -" (java.lang.String)
    invokestatic crackme/dup2_x2/l.U(Ljava/lang/Object;)Ljava/lang/String;
    iconst_m1
    goto L49
    L50 {
        astore0
        goto L51
    }
    L49 {
        swap
        invokedynamic crackme/dup2_x2/Main.pa(Ljava/lang/Object;Ljava/lang/Object;Ljava/lang/Object;)Ljava/lang/Object;
            : -19k39lo(Ljava/lang/Object;)Ljava/lang/Object;
        checkcast char[]
        // trimmed
```

The code starts with some meaningless unicode string in the beginning is passed to **l.U()** and a **invokedynamic** call to **Main.pa()**
I'll deal with the dynamic invokes later, but for now I'll look at the **l.U()**:

l.class, decompiled with Fernflower

```
static final String U(Object var0) {
    if(l.X.get(var0) != null) {
        return (String)l.X.get(var0);
    } else {
        boolean var21 = false;
        boolean var22 = false;
        if(l.a == null) {
            a18983();
        }
        // stripped code
        label73:
        while(true) {
            if(var34 != 0) {
                break;
            }

            ++var34;
            int var11 = var28.length;
            int var31 = 0;

            while(true) {
                if(var31 >= var11) {
                    break label73;
                }

                if(var31 % 8 == 0) {
                    // stripped code
                    for(var33 = 4; var33 < 36; var33 += 4) {
                        // stripped code
                    }
                    // stripped code
                }

                var21 = false;
                var10000 = null;

                try {
                    label67:
                    while(true) {
                        try {
                            if(!var21) {
                                // stripped code
                            }
                            break;
                        } catch (Exception var26) { }
                    }
                    ++var31;
                } catch (Exception var27) {
                    break;
                }
            }
        }
```

```
        String var24 = new String(var28);
        X.put(var0, var24);
        return var24;
    }
}
```

As expected, the decompiled code looks like a giant mess. That **X.get(var0)** in the beginning is a call to **z.class** (**X** is set to **new z(96)** in the **l.class** static init) so this is part of the caching to the hash map class that I mentioned before.
After that, depending on the content of the private variable **l.a**, a method **l.a18983()** might or might not be executed. Because this is the first run of **l.class**, **l.a** is **null**, **l.a18983()** will be executed.
This method was completely decompiled, so there wasn't any messy obfuscation (only a tight one):

l.a18983() decompiled

```
private static final void a18983() {
    u(0, null);
    b();
    final l l = new l(1);
    l.start();
    l.join();
    final l i = new l(3);
    i.start();
    final l j = new l(4);
    j.start();
    i.join();
    j.join();
    final l k = new l(7);
    final l m = new l(8);
    k.start();
    m.start();
    k.join();
    m.join();
}
```

This looks pretty simple, and after few minutes looking here and there, it turned out that whole code will execute method **l.u(int, Object)**, by **iterating the int argument from 0 to 8**.

Some of the execution of **l.u()** are done as threads, as seen for 1, 3, 4, 7 and 8.
The **l.u()** method, that I wont paste here because it's quite big, is basically a giant switch, that can be explained in the following table (in order of execution):

int	Object	l.u(int, Object) description
0	null	Init l.a static variable to array of 8 Objects : byte[256], int[256], int[256], int[256], int[256], null, null, null
1	null	Generates the GF(256) Galois Fields (or finite fields) and passes it as second parameter of l.u() with int = 2
2	finite_fields	Generates AES S-Box, T0, T1, T2 and T3 lookup tables and stores them to l.a[0] to l.a[4]
3	null	Extract a AES key from a 2D matrix and passes it as second argument of l.u() with int = 5
5	aes_key	Init the keys schedule buffer and stores it to l.a[5], then call u() with int = 6 and Object = null
6	null	Generates AES RCon values and using the S-Box, expands the key to complete the key schedule table
4	null	Sets the IV to l.a[6], from a hard coded DWORD values
7	null	Modifies the first DWORD of the IV
8	null	Modifies the second DWORD of the IV

There's one final step of modifying the IV, where the hashCode() of the caller class and method is XORed with each DWORD of the IV.
In the end, putting the whole algorithm together, I have reconstructed this code:

Rijndael based decryptor

```
StackTraceElement[] st = Thread.currentThread().getStackTrace();
int XOR_magic = new String(st[(int)a[7]].getClassName()+st[(int)a[7]].getMethodName()).hashCode();

char[] message = ((String)arg0).toCharArray();   // Encrypted data
byte[] sbox = (byte[])a[0];                      // S-Box
int[] T0 = (int[])a[1];                          // T0
int[] T1 = (int[])a[2];                          // T1
int[] T2 = (int[])a[3];                          // T2
int[] T3 = (int[])a[4];                          // T3
int[] key = (int[])a[5];                         // AES Key
int[] IV = (int[])a[6];                          // AES IV

int Y0, Y1, Y2, Y3, X0, X1, X2, X3;

// Initial IV reconstruction, using the XOR_magic
X0 = XOR_magic ^ IV[0];
X1 = XOR_magic ^ IV[1];
X2 = XOR_magic ^ IV[2];
X3 = XOR_magic ^ IV[3];

for (int i = 0; i < message.length; i+=8) {
   // AddRoundKey()
   Y0 = X0 ^ key[0];
   Y1 = X1 ^ key[1];
   Y2 = X2 ^ key[2];
   Y3 = X3 ^ key[3];

   for(int j = 1; j < 10; j++) {
      // SubBytes(), ShiftRows(), MixColumns() and AddRoundKey()
      X0 = T0[Y0 & 255] ^ T1[Y1 >> 8 & 255] ^ T2[Y2 >> 16 & 255] ^ T3[Y3 >>> 24] ^ key[(j*4)+0];
      X1 = T0[Y1 & 255] ^ T1[Y2 >> 8 & 255] ^ T2[Y3 >> 16 & 255] ^ T3[Y0 >>> 24] ^ key[(j*4)+1];
      X2 = T0[Y2 & 255] ^ T1[Y3 >> 8 & 255] ^ T2[Y0 >> 16 & 255] ^ T3[Y1 >>> 24] ^ key[(j*4)+2];
```

```
      X3 = T0[Y3 & 255] ^ T1[Y0 >> 8 & 255] ^ T2[Y1 >> 16 & 255] ^ T3[Y2 >>> 24] ^ key[(j*4)+3];
      Y0 = X0; Y1 = X1; Y2 = X2; Y3 = X3;
   }

   // SubBytes(), ShiftRows() and AddRoundKey()
   X0 = sbox[Y0 & 255] & 255 ^ (sbox[Y1 >> 8 & 255] & 255) << 8 ^ (sbox[Y2 >> 16 & 255] & 255) << 16 ^ sbox[Y3 >>> 24] << 24 ^ key[40];
   X1 = sbox[Y1 & 255] & 255 ^ (sbox[Y2 >> 8 & 255] & 255) << 8 ^ (sbox[Y3 >> 16 & 255] & 255) << 16 ^ sbox[Y0 >>> 24] << 24 ^ key[41];
   X2 = sbox[Y2 & 255] & 255 ^ (sbox[Y3 >> 8 & 255] & 255) << 8 ^ (sbox[Y0 >> 16 & 255] & 255) << 16 ^ sbox[Y1 >>> 24] << 24 ^ key[42];
   X3 = sbox[Y3 & 255] & 255 ^ (sbox[Y0 >> 8 & 255] & 255) << 8 ^ (sbox[Y1 >> 16 & 255] & 255) << 16 ^ sbox[Y2 >>> 24] << 24 ^ key[43];

   // Decrypt block
   try {
      message[i+0] ^= X0 >> 16; message[i+1] ^= X0;
      message[i+2] ^= X1 >> 16; message[i+3] ^= X1;
      message[i+4] ^= X2 >> 16; message[i+5] ^= X2;
      message[i+6] ^= X3 >> 16; message[i+7] ^= X3;
   } catch(Exception e) { }
}
```

Overall, the algorithm used is decrypting a non-padded wide char string and produces, again non-added multi byte one, using various layers of obfuscation of its key and IV. So, the input wide char string:

```
Input
Offset      0000 0001 0002 0003 0004 0005 0006 0007 0008 0009 000A 000B 000C 000D 000E 000F

00000000    1270 B5A4 6482 457F CA09 4378 13D5 A7EE 5740 2685 7D68 C983 F466 1E95 30E1 28B4
00000010    4B31 7B6F FD28 46CF 7D65 2264 4F44 4969 B737 E7F2 3DCD E685 6968 1595 E885 CEA9
00000020    5528 0884 6EDF 514F EB6A 5077 58F2 D215 0B32 A58F F6D5 82F6 0200 4B8C 88E9 C790
00000030    9F7E 2208 C421 D534 B86B 9F11 7E02 53C2 EB4E 9407 0AAF C262 C43B 30D8 8C8A 018F
00000040    E42C 42A8 CDF5 5769 F439 B6FB 441C EECE 4A00 31C1 696B 1E57 1FBE FBB0 4DEF A059
00000050    012B D026 AA29 C028 DD13 99B2 0A9B E94A 0883 3BFC E5D8 219D 77D4 12EA CCAC E99C
00000060    7E0C 4871 862B 06AD C0AB D8F1 2A60 CAB8 886C 5F77 F093 A0C3 AC64 6EC4 ABAC 8006
00000070    9BC3 B1CB D4B9 A2A9 CDB6 AA0A F0C6 A585 E039 0661 B069 BD08 E0A5 FCF7 AA70 64E8
00000080    CC9F 1287 2088 0B9E 5C91 99B6 F39C 53C9 B1BA 73AC E907 53FF 5700 6DD7 7A14 F9C6
00000090    C61F AC7C 83E6 4F7D F116
```

Gets decrypted to:

```
Output
Offset      00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

00000000    DA 60 EA F5 5D CE B6 B6 60 FA 53 C1 F8 B2 9A EA
00000010    DC 51 8C A3 C5 D9 EB 9C 97 C1 CC A4 B8 E0 D2 FB
00000020    C7 D2 82 8E FC E6 F7 47 50 98 82 C8 7A DB C3 F8
00000030    A0 B8 E8 F2 51 F7 C0 A0 1C C6 E8 DF CF F8 2A 9A
00000040    FC E0 C3 C3 D2 82 8E FC E6 88 5E 51 2B 1D 63 7B
00000050    50 5E 51 2B 1D 63 7B 50 AD B0 B2 BE C0 CE 4F 59
00000060    D2 86 B2 64 CE C9 F3 F2 2C 9C FA E6 D3 CB CA AE
00000070    B2 F6 E2 CB 59 F8 AE BC 64 D2 F3 C7 CA A2 DE 85
00000080    E5 DC DA F1 A7 1B F7 E1 F6 F2 FF 2D 99 65 E1 FC
00000090    C2 D3 A7 BF B5
```

That's meaningless for now, but since I reverse engineered the whole **l.claas**, by adjusting the method names and the constant values, I now have also **n.class**, **aa.class** and **xt.class** in my crackme project.

Moving on to the next part - the dynamically invoked functions by **Main.pa()**
The invoker itself is quite simple:

```
Main.pa()

private static CallSite pa(Lookup caller, String name, MethodType type) {
   try {
      return new ConstantCallSite(
         (caller).unreflect(
            jf.a((int)Integer.valueOf((String)name, 32))
         ).asType(type)
      );
   } catch (IllegalAccessException ex) {
      throw new BootstrapMethodError((Throwable)ex);
   }
}
```

The method **jf.a()** takes the integer representation of the strings passed and returns a **Method**, so **jf.class** is where I should continue my research.
jf.class has a static initialization that sets it private variables, but the **Procyon** decompiler was able to reconstruct it completely:

```
jf.class <clinit>

static {
    // n.u() is basically the same as l.U(), except the AES key and IV are different
    l = Integer.parseInt(crackme.dup2_x2.n.u((Object)"\uec31\uc4e9\u7d9e\u79a2\u4494\u60b5\ud5f7\u3993\u1004\ua2e8")); // 0x78EFE205
    A = Integer.parseInt(crackme.dup2_x2.n.u((Object)"\uec2e\uc4ed\u7d94\u79a2\u449a\u60b9\ud5f7\u399b\u100b\ua2ef")); // 0xE2DE53A8
    Z = new Method[50];  // cached Methods
    m = new Class[14];   // cached Classes
    F = new int[50];     // Method CRC values
    n = new short[50];   // Class magic values

    char[] data_F = crackme.dup2_x2.n.u((Object)"\u8ada\uaad9\ubde5\ue4e9...trimmed...").toCharArray();
    char[] data_n = crackme.dup2_x2.n.u((Object)"\uec0b\uc4dc\u7da4\u7996...trimmed...").toCharArray();

    for (int i = 0; i < 50; i++) {
        jf.F[i] = (data[i * 2] | data[i * 2 + 1] << 16);

        jf.n[i] = (short)data[i];
```

```
    }
}
```

In matter of fact, the whole **jf.class** was completely decompiled without a problem, so I take its code almost as-is.
The way **jf.a()** uses its parameter to resolve a **Method** goes like this:

jf.a()

```java
static Method a(int var0) {
    // stripped code: variable declarations

    // taking Integer.valueOf("uqimad", 32) that is 1034508621 as input parameter
    var0 = (((((var0 + 520679521) ^ jf.l) - 853128736) ^ -1913955857) + jf.A); // var0 = 0x5F47000C
    var1 = (var0 >>> 16);  // take HIWORD 0x5F47 to var1
    var0 = (var0 & 65535); // take LOWORD 0x000C to var0

    // Return the var0 (0x000C'th) element from the jf.Z array if it's not NULL
    // that's part of the caching, where resolved methods are saved to a cache array
    requested_method = (Method)jf.Z[var0];
    if (requested_method != null) {
        return requested_method;
    }

    // otherwise, proceed to resolve the method

    // jf.G() is additional table lookup, that returns the base class of the requested method
    // it further process var1 by the following equation: ((jf.n[var0] & 65535) + var1) % 14
    // the resulting value is a index for the jf.m array, that is serving as cache for the resolved Classes
    base_class = jf.G(var0, var1);

    // requested method is picked based on var0 index in a checksum table jf.F
    requested_crc = jf.F[var0];

    // CASE A - Requested method is inside Class or Interface
    while (base_class != null) {
        if (base_class.isInterface()) {
            list_methods = base_class.getMethods();
        } else {
            list_methods = base_class.getDeclaredMethods();
        }

        for(int i = 0; i < list_methods.length; i++) {
            cur_method = list_methods[i];

            // BOF: Checksum algorithm
            current_crc = (((var1 * 31) + cur_method.getName().hashCode()) * 31) + 40;

            list_classes = cur_method.getParameterTypes();
            for(int j = 0; j < list_classes.length; j++) {
                if (j != 0) {
                    current_crc = (current_crc * 31) + 44;
                }
                current_crc = (current_crc * 31) + list_classes[j].getName().hashCode();
            }
            current_crc = (((((current_crc * 31) + 41) * 31) + cur_method.getReturnType().getName().hashCode()) * 31) + var1;
            // EOF: Checksum algorithm

            if (current_crc == requested_crc) {
                cur_method.setAccessible(true);
                // Cache the resolved method
                jf.Z[var0] = cur_method;

                return cur_method;
            }
        }
        // move to the next super class
        base_class = base_class.getSuperclass();
    }

    // CASE B - Requested method is inside Class Interfaces
    // stripped code: enumeration of interfaces
    // The code is pretty much the same as in CASE A
}
```

I now have working **jf.a(long)** function so the next thing I did was to get all the hash values passed to it and build this table:

original hash	int value of the hash	corresponding method
-14sr9le	-1238214318	public java.lang.Class java.lang.reflect.**Field.getType()**
-178r9ln	-1317906103	public static java.lang.Object[] crackme.dup2_x2.**a.a(java.lang.Object)**
-18bv9ld	-1354737325	public java.lang.String java.lang.**String.substring(int)**
-19a99mp	-1386522329	public synchronized java.lang.StringBuffer java.lang.**StringBuffer.append(char)**
-19k39lo	-1396811448	public char[] java.lang.**String.toCharArray()**
-1cgb9lk	-1493542580	public static java.lang.Object[] crackme.dup2_x2.**a.a(char)**
-1hpv9ke	-1671407246	public static java.lang.reflect.Field crackme.dup2_x2.**a.c(long)** throws java.lang.Throwable
-1kol9lh	-1770694321	public java.lang.Class[] java.lang.**Class.getInterfaces()**

-1pkr9kh	-1934468753	public java.lang.Object java.lang.reflect.**Field.get(java.lang.Object)** throws java.lang.IllegalArgumentException,java.lang.IllegalAccessException
-1ssv9lb	-2043651755	public static long java.lang.**Long.parseLong(java.lang.String,int)** throws java.lang.NumberFormatException
-1unb9la	-2104862378	public int java.lang.**String.indexOf(int,int)**
-2rf9mk	-95921876	public java.lang.String java.lang.reflect.**Method.getName()**
-4219lg	-136357552	public int java.lang.**String.indexOf(int)**
-5mj9li	-191473330	public static java.lang.Object[] crackme.dup2_x2.**a.a(int)**
-6259ll	-203597493	public char java.lang.**Character.charValue()**
-7n79lp	-259237561	public boolean java.lang.**Boolean.booleanValue()**
-a5d9kb	-341223051	public static java.lang.reflect.Method crackme.dup2_x2.**a.d(long)** throws java.lang.Throwable
-bb19lf	-380675759	public boolean java.lang.**String.equals(java.lang.Object)**
-dt19kg	-466658960	public static java.lang.Object[] crackme.dup2_x2.**a.b()**
-fq99l5	-530884261	public java.lang.reflect.Field[] java.lang.**Class.getDeclaredFields()** throws java.lang.SecurityException
-jat9m1	-648980161	public java.lang.String java.lang.**String.substring(int,int)**
-ter9mm	-988653270	public synchronized java.lang.StringBuffer java.lang.**StringBuffer.append(java.lang.String)**
-uvv9l3	-1040164515	public static java.lang.Class java.lang.**Class.forName(java.lang.String)** throws java.lang.ClassNotFoundException
12g6m92	1157847330	public java.lang.Class[] java.lang.reflect.**Method.getParameterTypes()**
17cimas	1321818460	public java.lang.String java.lang.reflect.**Field.getName()**
1bquma0	1471109440	public char java.lang.**String.charAt(int)**
1ca0mak	1486903636	public java.lang.String java.lang.**Class.getName()**
1cssmbj	1506695539	public int java.lang.**Integer.intValue()**
1gn0mbk	1634752884	public java.lang.Throwable java.lang.reflect.**InvocationTargetException.getTargetException()**
1n0m9b	57694507	public java.lang.reflect.Method[] java.lang.**Class.getDeclaredMethods()** throws java.lang.SecurityException
1rrsma2	2008963394	public int java.lang.**String.length()**
1v08mau	2114214238	public java.lang.String java.lang.**Throwable.toString()**
61um99	203381033	public native java.lang.Class java.lang.**Class.getSuperclass()**
892m91	277960993	public java.lang.Class java.lang.reflect.**Method.getReturnType()**
c6cm9e	409360686	public static java.lang.Integer java.lang.**Integer.valueOf(int)**
gj8m98	557078824	public synchronized java.lang.String java.lang.**StringBuffer.toString()**
hnkm9d	595220781	public static java.lang.Character java.lang.**Character.valueOf(char)**
lj0mbm	724588918	public java.lang.Object java.lang.reflect.**Method.invoke(java.lang.Object,java.lang.Object[])** throws java.lang.IllegalAccessException,java.lang.IllegalArgumentException,java.lang.reflect.InvocationTargetException
uqimad	1034508621	public native java.lang.String java.lang.**String.intern()**

Nice.

I can now replace all dynamic invokes to the corresponding function from this table and have a bit clearer code.
Because I have to figure out what these ~700 lines of **Main.class** static init do, I extracted all the dynamically invoked functions, put them in a table and found an interesting pattern:

Object[]	Method	invoke()	cast	error handler
a.a(int)	a.d(long)	Method.invoke(Object,Object[])	Character.charValue()	InvocationTargetException.getTargetException()
a.b()	a.d(long)	Method.invoke(Object,Object[])	Integer.intValue()	InvocationTargetException.getTargetException()
a.b()	a.d(long)	Method.invoke(Object,Object[])		InvocationTargetException.getTargetException()
a.a(Object)	a.d(long)	Method.invoke(Object,Object[])		InvocationTargetException.getTargetException()
a.a(char)	a.d(long)	Method.invoke(Object,Object[])		InvocationTargetException.getTargetException()
a.b()	a.d(long)	Method.invoke(Object,Object[])		InvocationTargetException.getTargetException()
a.b()	a.d(long)	Method.invoke(Object,Object[])	Boolean.booleanValue()	InvocationTargetException.getTargetException()
a.a(Object)	a.d(long)	Method.invoke(Object,Object[])		InvocationTargetException.getTargetException()
a.a(char)	a.d(long)	Method.invoke(Object,Object[])		InvocationTargetException.getTargetException()
a.b()	a.d(long)	Method.invoke(Object,Object[])		InvocationTargetException.getTargetException()
a.b()	a.d(long)	Method.invoke(Object,Object[])	Integer.intValue()	InvocationTargetException.getTargetException()
a.a(Object)	a.d(long)	Method.invoke(Object,Object[])		InvocationTargetException.getTargetException()
a.a(Object)	a.d(long)	Method.invoke(Object,Object[])		InvocationTargetException.getTargetException()

To sum it up, the function from the first column returns a **Object[]**, that is used as second parameter of **invoke()**.
I lookup these functions and it turned out they are simply putting the passed value to a object array:

value to Object[] functions

```
public static Object[] a(int n) {
   return new Object[] { Integer.valueOf(n) };
}
```

```
public static Object[] a(char c) {
   return new Object[] { Character.valueOf(c) };
}

public static Object[] a(final Object o) {
   return new Object[] { o };
}

public static Object[] b() {
   return new Object[0];
}
```

The **invoke()**, additional typecast method and the error handler **getTargetException()** are pretty clear, so I only have to figure out what the **a.d(long)** does.

Obviously it should return a method that will be invoked, so the passed long value should be again some sort of a hash, just like in **jf.a()** before.

Because this code is in **a.class**, I had to reverse engineer the static init first.
There was a slight obfuscation here, so I did most of it by hand, and and in the end got this code that populates the class's global variables:

a.class static initialization

```
static {
    String decrypted_data = new String();

    // Encrypted data A, first chunk length
    decrypted_data += Character.toString((char)31);

    // Encrypted data A
    decrypted_data += crackme.dup2_x2.l.U("\u4C9F\uF467\u74B1\u257B\u085C\uA8B1\uF6C5\uD130\uA1D8\uAE79\uB263\u7016...");

    // Encrypted data B, first chunk length
    decrypted_data += Character.toString((char)40);

    // Encrypted data B
    decrypted_data += crackme.dup2_x2.l.U("\u4c95\uf450\u74aa\u257a\u082b\ua8f0\uf629\ud1d3\ua11f\uaee2\ub28f\u7011...");

    // XOR key
    char[] key = {68, 32, 36, 66, 65, 45, 51}, chunk_data;
    ArrayList<String> entries = new ArrayList<String>();

    for(int i = 0, chunk_len = 0; i < decrypted_data.length(); i+=chunk_len) {
        // get the length of the upcoming chunk
        chunk_len = decrypted_data.charAt(i);

        // skip the chunk length byte
        i++;

        // substring the chunk from the whole data
        chunk_data = decrypted_data.substring(i, i+chunk_len).toCharArray();

        // decrypt the chunk
        for(int j = 0; j < chunk_data.length; j++) {
            chunk_data[j] ^= key[j%key.length];
        }

        // add the decrypted chunk to the final array
        entries.add(new String(chunk_data));
    }

    crackme.dup2_x2.jf.u(-1682417339, (Object)entries);         // assign the entries array to a.c global variable

    crackme.dup2_x2.jf.u(-1900455610, (Object)new String[36]);  // assign the String[36] to a.d global variable

    crackme.dup2_x2.a();                                        // assigns the remaining a.a and a.b global variables
}
```

This produces an array of 36 strings (that are still encrypted) and passes it to **crackme.dup2_x2.jf.u()** with parameter **-1682417339**.
I already had half of **jf.class** reverse engineered in my project, and its **u()** method was in there along with its children methods:

jf.u()

```
static void u(final int n, final Object o) {
    h(null, n, o);
}

static void h(final Object o, final int n, final Object o2) throws Throwable {
    final Field o3 = o(n);
    if (o3 == null) {
        throw new NoSuchFieldError(Integer.toString(n));
    }
    o3.set(o, o2);
}

private static Field o(int n) throws Throwable {
    // stripped code: variable declarations

    // the code below is pretty much the same as the one in jf.a()
    var0 = ((((n + 520679521) ^ jf.l) - 853128736) ^ -1913955857) + jf.A;
    var1 = var0 >>> 16;
    var0 = var0 & 65535;

    requested_field = (Field)jf.Z[var0];
    if (requested_field != null) {
```

```
        return requested_field;
    }

    base_class = jf.G(var0, var1);

    // requested field is picked based on var0 index in a checksum table jf.F
    requested_crc = jf.F[var0];

    // CASE A - Requested Field is inside Class
    while (base_class != null) {
        list_fields = base_class.getDeclaredFields();
        for(int i = 0; i < list_fields.length; i++) {
            cur_field = list_fields[i];

            // Generate a checksum of the current field
            current_crc = (((var1 * 31) + cur_field.getName().hashCode()) * 31) + 58;
            current_crc = (((current_crc * 31) + cur_field.getType().getName().hashCode()) * 31) + var1;

            if (current_crc == requested_crc) {
                cur_field.setAccessible(true);
                // make private static visible outside the class
                if (Modifier.isStatic(cur_field.getModifiers())) {
                    if (Modifier.isFinal(cur_field.getModifiers())) {
                        // n.u() decrypts the string "modifiers"
                        cur_field_mods = Field.class.getDeclaredField(
                            crackme.dup2_x2.n.u("\u1563\ue613\ueba4\uf2c0\u8cf6\ud1ff\u36b6\u5970\u5917")
                        );
                        cur_field_mods.setAccessible(true);
                        cur_field_mods.setInt(cur_field, cur_field.getModifiers() & 239);
                    }
                }
                jf.Z[var0] = cur_field;
                return cur_field;
            }
        }
        base_class = base_class.getSuperclass();
    }

    // CASE B - Requested Field is inside Interface
    // stripped code: similar to CASE A
}
```

So, what **jf.u()** does is to assign the passed object to a Field resolved by the passed int argument:

jf.u() implementation

```
// this code
crackme.dup2_x2.jf.u(-1682417339, (Object)entries);

crackme.dup2_x2.jf.u(-1900455610, (Object)new String[36]);

// is doing that
crackme.dup2_x2.a.c = entries;

crackme.dup2_x2.a.d = new String[36];
```

That's a lot of effort to hide a simple variable assignment.

Alright, moving to the final call in **a.class** static init - **a.a()**:

a.a()

```
private static void a() {
    jf.u(1455380803, (Object)new String[32]);            // crackme.dup2_x2.a.b = new String[32]
    final Object[] array = new Object[32];
    jf.u(-1931257532, (Object)array);                     // crackme.dup2_x2.a.a = new Object[32]
    array[0] = a(24733, 15678);
    array[1] = a(24715, 29873);
    array[2] = a(24707, 12354);
    array[3] = jf.Y(-1622976167);                         // java.lang.Void.TYPE
    ((String[])jf.Y(1455380803))[3] = a(24709, -4364);   // crackme.dup2_x2.a.b[3] = ...
    array[4] = a(24705, -4282);
    array[5] = a(24726, 3321);
    array[6] = a(24712, -30210);
    array[7] = a(24721, -4951);
    array[8] = jf.Y(-1957865126);                         // java.lang.Boolean.TYPE
    ((String[])jf.Y(1455380803))[8] = a(24745, -18847);  // crackme.dup2_x2.a.b[8] = ...
    array[9] = jf.Y(-316188329);                          // java.lang.Character.TYPE
    ((String[])jf.Y(1455380803))[9] = a(24704, 7645);    // crackme.dup2_x2.a.b[9] =
    array[10] = a(24716, -12185);
    array[11] = jf.Y(-349480616);                         // java.lang.Integer.TYPE
    ((String[])jf.Y(1455380803))[11] = a(24724, 13165);  // crackme.dup2_x2.a.b[11] = ...
    array[12] = a(24718, 20829);
    // trimmed code: assignments to 32st element
}
```

Ok, code is pretty simple. I followed that **a.a(int, int)** method, and it seems it's a XOR based decryptor:

a.a(int, int)

```
public static String a(int arg0, int arg1) throws Throwable {
    // trimmed code: variable declarations

    index = (arg0 ^ 24713) & 0xFFFF;
    if (crackme.dup2_x2.a.d[index] != null) {
        return crackme.dup2_x2.a.d[index];
```

```
    }

    data = crackme.dup2_x2.a.c[index].toCharArray();

    int[] base_data = {0xF0, 0x2C, 0x90, 0x80, 0x7A, 0xDE, 0x3E, 0xC9, 0x47, 0xCF, 0x24, 0x06, 0x26, 0x1F, 0x9E, 0x17,
                       0xB3, 0x97, 0x8A, 0xAD, 0x29, 0x74, 0xF3, 0xDF, 0xAE, 0xEE, 0x0B, 0xBF, 0x81, 0xE1, 0x7F, 0x32,
                       0x4D, 0x9B, 0x34, 0xE7, 0xA6, 0x23, 0x9F, 0x3A, 0xE9, 0xC3, 0xB1, 0x75, 0x27, 0x6B, 0x3D, 0xFB,
                       0x13, 0x3C, 0x7D, 0xE5, 0x6A, 0x82, 0x48, 0x9A, 0x7E, 0x87, 0x49, 0x15, 0x22, 0x19, 0xE6, 0xAF,
                       0x46, 0xEA, 0xC6, 0xC7, 0xCA, 0x8D, 0xBA, 0x5F, 0x96, 0xC5, 0x4F, 0x07, 0x1E, 0x08, 0x4B, 0xA5,
                       0xCC, 0x11, 0xCB, 0xDB, 0xB0, 0x5A, 0xE4, 0xCD, 0xDD, 0x63, 0x88, 0x35, 0xC4, 0x10, 0x5B, 0x16,
                       0x76, 0x62, 0x50, 0xC8, 0x4E, 0x36, 0x02, 0xA7, 0x9D, 0x78, 0x58, 0xD7, 0x73, 0x59, 0x85, 0xCE,
                       0x98, 0xDC, 0xED, 0xAB, 0x43, 0xFC, 0x83, 0xC0, 0x6F, 0x52, 0xF4, 0x2A, 0xAA, 0x72, 0xDA, 0x1B,
                       0x51, 0x2D, 0x65, 0x70, 0x44, 0x42, 0x14, 0xF5, 0xD1, 0x03, 0xFD, 0x86, 0x28, 0x55, 0x94, 0xB4,
                       0x67, 0x2F, 0x21, 0x79, 0xD2, 0x3B, 0x9C, 0x2B, 0x54, 0xB9, 0xFE, 0xC2, 0xA8, 0x40, 0xD4, 0x95,
                       0xD5, 0x1A, 0xB6, 0x3F, 0x30, 0x56, 0xF8, 0x69, 0xD9, 0x6D, 0x53, 0x0C, 0xBE, 0x5E, 0x37, 0x71,
                       0xBB, 0x92, 0xD8, 0x89, 0x31, 0xAC, 0x09, 0xB2, 0xE2, 0xF2, 0xEF, 0xB5, 0xFF, 0x4A, 0x0F, 0x04,
                       0x7C, 0x93, 0xF6, 0x8B, 0xB8, 0x38, 0xA2, 0x66, 0x0E, 0xD0, 0x6E, 0xF7, 0x0D, 0x05, 0xF1, 0xFA,
                       0x91, 0x7B, 0x77, 0x64, 0x57, 0x5C, 0xA9, 0xA4, 0x99, 0x39, 0x33, 0xC1, 0xD6, 0xF9, 0x18, 0x8C,
                       0xE8, 0xBC, 0xA3, 0xEB, 0x45, 0x6C, 0xD3, 0x4C, 0x68, 0x8E, 0x12, 0x84, 0x00, 0x0A, 0x41, 0x20,
                       0x5D, 0x01, 0x60, 0x8F, 0xE3, 0x61, 0x2E, 0xBD, 0xE0, 0x25, 0x1D, 0xEC, 0xB7, 0xA1, 0xA0, 0x1C};

    base = base_data[data[0]%0xFF];

    key_A = (arg1 & 0xFF) - base;
    key_A += ((key_A < 0) ? 0x100 : 0);

    key_B = ((arg1 & 0xFFFF) >>> 8) - base;
    key_B += ((key_B < 0) ? 0x100 : 0);

    for(i = 0; i < data.length; i++) {
        if (i % 2 == 0) {
            data[i] ^= key_A;
            key_A = (((key_A >>> 3) | (key_A << 5)) ^ data[i]) & 0xFF;
        } else {
            data[i] ^= key_B;
            key_B = (((key_B >>> 3) | (key_B << 5)) ^ data[i]) & 0xFF;
        }
    }
    crackme.dup2_x2.a.d[index] = new String(data).intern();
    return crackme.dup2_x2.a.d[index];
}
```

This decryptor doesn't produce any readable data, so again this is only a layer of obfuscation.

Alright, the **a.class** static init is ready, and the global variables **a.a**, **a.b**, **a.c** and **a.d** are all populated with data as follows:
a.a = Array of encrypted strings, where only **"void"**, **"boolean"**, **"char"** and **"int"** are decrypted so far
a.b = Array of classes, currently holding **java/lang/Void**, **java/lang/Boolean**, **java/lang/Character** and **java/lang/Integer**
a.c = Array of encrypted data
a.d = Array of encrypted data

Finally, I can move to the **a.d(long)** method:

a.d(long)

```
public static Method d(long arg0) throws Throwable {
    data_index = a(arg0);   // decrypt data in a.b global, and return its index

        // check if the requested Method not decrypted yet
    cached_method = crackme.dup2_x2.a.a[data_index];
    if (cached_method instanceof String) {
        // take the method data, that is now decrypted
        method_data = crackme.dup2_x2.a.b[data_index];

                // Find first occurrence of 0x08 char, and take the first data chunk
        offset_A = method_data.indexOf(8);
        pt1_Class = b(Long.parseLong(method_data.substring(0, offset_A), 36));
                // a.b(long) does a lookup and locates the requested class

        offset_A++;

                // Find second occurrence of 0x08 char, and take the second chunk
        offset_B = method_data.indexOf(8, offset_A);
        pt2_String = method_data.substring(offset_A, offset_B);

                // stripped code: count how many chunks are present

                // stripped code: iterate through the remaining chunks and using a.b(long), extract the Method's arguments type classes

                // Locate the method, cache it for future usage, and return it
        requested_method = a(pt1_Class, pt2_String, pt3_Class, pt4_int, pt5_Class_array);
        if (requested_method != null) {
            crackme.dup2_x2.a.a[data_index] = requested_method;
            return requested_method;
        }
    }
    return (Method)cached_method;
}
```

So, this **a.d(long)** method fetcher is a parser working together with **a.a(long)** as decryptor, **a.b(long)** as Class lookup and **a.a(Class, String, Class, int, Class[])** does the final lookup.
The method **a.a(long)** is a simple decryption algorithm:

a.a(long) decryptor

```
private static int a(long arg0) throws Throwable {
    // trimmed code: variable declarations

    // caching
    item_index = (int)(arg0 >>> 0x2E);
    if (crackme.dup2_x2.a.b[item_index] != null) {
        return item_index;
    }

    encrypted_data = crackme.dup2_x2.a.a[item_index];

    int[] base_data = {0x1A, 0x39, 0x03, 0x18, 0x12, 0x1C, 0x37, 0x3C, 0x31, 0x22, 0x11, 0x06, 0x15, 0x08, 0x10, 0x16,
                       0x3F, 0x0B, 0x0C, 0x2E, 0x09, 0x07, 0x00, 0x3E, 0x3A, 0x36, 0x1B, 0x3D, 0x25, 0x38, 0x19, 0x2A,
                       0x24, 0x21, 0x28, 0x2B, 0x05, 0x2C, 0x34, 0x14, 0x33, 0x0D, 0x1E, 0x3B, 0x23, 0x01, 0x27, 0x1D,
                       0x1F, 0x17, 0x29, 0x30, 0x32, 0x0E, 0x26, 0x04, 0x0A, 0x2D, 0x2F, 0x13, 0x20, 0x0F, 0x02, 0x35};
    base = base_data[(int)(arg0 >>> 0x2A & 0x3FL)];

    // construct the decrypt key
    key = new int[6];
    for(int i = 0; i < 6; i++) {
       key_byte = (int)(((arg0 >>> (7 * (5 - i))) & 0x7F) - base);
       if ((int)key_byte < 0) {
           key_byte += 0x80;
       }
       var5[i] = key_byte;
    }

    // decrypt loop
    decrypted_data = ((String)encrypted_data).toCharArray();
    for(int i = 0; i < decrypted_data.length; i++) {
        var8 = var5[i%var5.length];
        if (var8 == 0) {
            break;
        }
        decrypted_data[i] ^= (char)var8;
    }

    // assign the decrypted data to a.b
    crackme.dup2_x2.a.b[item_index] = new String(decrypted_data);
    return item_index;
}
```

Nice. That explains a lot about how methods are resolved.

I can finally get back to the **Main.class** static init and since I know how **a.d(long)** works, I've took all the long integers passed to **a.d()** and build this table of resolved Methods:

a.d(long) value	resolved Method
1255656867296710l	public void java.io.PrintStream.println(java.lang.String)
1350654340882987l	private static java.lang.Throwable crackme.dup2_x2.Main.b(java.lang.Throwable)
1429177141414625l	public java.lang.String java.lang.StringBuilder.toString()
1538972900944217l	public boolean java.lang.String.isEmpty()
1551680559876879l	public java.lang.StringBuilder java.lang.StringBuilder.append(char)
1665227756912309l	public static java.lang.String java.lang.String.valueOf(java.lang.Object)
1721362425867234l	public int java.lang.String.length()
1813506537578514l	public char java.lang.String.charAt(int)
1858909163626447l	public char[] java.lang.String.toCharArray()
1907841105922202l	public static int java.lang.Integer.parseInt(java.lang.String) throws java.lang.NumberFormatException
2020246543040315l	public boolean java.lang.Object.equals(java.lang.Object)
2071814875667432l	public int java.lang.String.hashCode()
2135288574399915l	public final native java.lang.Class java.lang.Object.getClass()
2235160856853667l	public java.lang.StringBuilder java.lang.StringBuilder.append(java.lang.String)

Alright, that's everything I need to start deobfuscating the code in **Main.class**

I already have a bunch of data decrypted with **AES**, and I have deobfuscated the Method calls, so I was able to reconstruct the rest of the static init code:

Main.class static init code flow

```
decrypted_data = "\xDA\x60\xEA\xF5\x5D\xCE\xB6\xB6\x60\xFA\x53\xC1\xF8\xB2\x9A\xEA";

// additional XOR decryption is applied here
char[] key = {94, 32, 56, 19, 29, 18, 104};
for(int i = 0; i < decrypted_data.length; i++) {
    decrypted_data[i] ^= key[i%key.length];
}

char[] data = new String(decrypted_data).toCharArray();
String[] decrypted_obj = new String[6];
int decrypted_string_i = 0;

for(i = 0; i < data.length; i++) {
    // First byte, XORed with the whole data len determines the following chunk length
```

```
    int chunk_len = (int)data[i] ^ data.length;
    if (chunk_len > 0) {

        // The chunk decrypt algorithim is simple chunk[i] ^ chunk_i >> 1
        StringBuilder decrypted_string = new StringBuilder("");
        for(int j = 0; j < chunk_len; j++,i++) {
            decrypted_string.append((char)(((int)data[i+1] ^ decrypted_string_i) >> 1));
        }

                // The decrypted chunks are converted to strings and stored in the decrypted_obj array
        decrypted_obj[decrypted_string_i] = decrypted_string.toString();
        decrypted_string_i++;
    }
}
```

After executing this code, I've obtained a list of the crackme's strings in their deobfuscated form:

decrypted message	description
is not a number!	error message
XVCe	no idea what this is (for now), but it does look like a odd-based integer in a ASCII form
Congratulations, you entered the right comination	Success message (typo is not my fault)
	Just a bunch of spaces
Sorry, but your combination was wrong	Failed message
Please input a number	Welcome message

And that's how I defeated the GraxCode's obfuscation of the crackme!
What left now is to reverse engineer the CLI main() method.

Solving the crackme

Now that I know how the Methods are resolved and have everything decrypted, I can take a look at the **main()** Method.
The decompilers fail to produce any JAVA code but after replacing obfuscated stuff I get to here:

main(String[])

```
public static void main(java.lang.String[]);
    Code:
        // -- snip --
        11: istore_1                          // set to false, returned by jf.Y(-983672463).booleanValue()
        12: new           #2                  // class crackme/dup2_x2/Main
        15: aload_0                           // / args
        16: iconst_0                          // | 0
        17: aaload                            // \ pass args[0] to Main()
        18: invokespecial #131                // Method "<init>":(Ljava/lang/String;)V
        21: goto          91
        // bunch of exception handling and chunks of dead code
```

So there's nothing interesting in here. The code user entered is directly passed to that odd looking **Main()** Method.
Again, the decompilers didn't work, so I started debugging it in JVM code.
Right in the start there was a call, taking the boolean **false** then storing it at local variable number 2 - **loc2**:

Main(String) Method

```
 4: ldc           #22                 // int -983672463
 6: invokestatic  #28                 // Method crackme/dup2_x2/jf.Y:(I)Ljava/lang/Object; - returns "false"
 9: checkcast     #30                 // class java/lang/Boolean                           - casts it to boolean
12: invokevirtual #34                 // Method java/lang/Boolean.booleanValue:()Z         - take its booleanValue()
15: istore_2                                                                               - stores it at loc2
```

And that wouldn't be interesting at all, if I didn't saw these conditional jumps, down the code:

Main(String), code snippets of loc2

```
274: iload_2
275: ifeq          409  // always jumps
// -- snip --
316: iload_2
317: ifne          351  // never jump
320: ifne          337  // will jump or not, depending on previous stack value
// -- snip --
333: iload_2
334: ifeq          369  // always jumps
// -- snip --
347: iload_2
348: ifne          331  // never jump
351: ifne          362  // will jump or not, depending on previous stack value
354: iload_2
355: ifeq          729  // always jumps
// -- snip --
387: iload_2
388: ifne          416  // never jump
// -- snip --
412: iload_2
413: ifne          406  // never jump
// -- snip --
421: iload_2
422: ifne          642  // never jump
// -- snip --
```

```
485: iload_2
486: ifne          314  // never jump
// -- snip --
549: iload_2
550: ifne          782  // never jump
// -- snip --
638: iload_2
639: ifne          557  // never jump
642: iload_2
643: ifne          670  // never jump
646: ifne          661  // will jump or not, depending on previous stack value
// -- snip --
653: iload_2
654: ifeq          729  // always jumps
// -- snip --
725: iload_2
726: ifeq          34   // always jumps
```

There's not a single place where **loc2** will be switched to **true**, so it seems like this is part of the obfuscation.
Another thing that the obfuscator did here was to put **athrow** instructions on random places, that decompilers try to logically link to a **try-catch block**, fail and in the end - be unable to decompile the code.
Stripping these, and deobfuscating the method calls I got myself a quite readable code:

Main(String), validation algorithm

```
119: istore        4         // resolve Integer.parseInt(java.lang.String), invoke it over arg0 and store the result in loc4
// trimmed code: dead code and some exception handling
264: iconst_0                // / 0
265: istore        5         // \ loc5 = 0
267: iload         4         // / loc4
269: bipush        8         // | 8
271: ishl                    // | loc4 << 8
272: istore        6         // \ loc6 = loc4 << 8
275: goto       409

// begin of loop
311: iload         4         // / loc4
313: iconst_2                // | 2
314: irem                    // | loc4 % 2
315: ineg                    // | - (loc4 % 2)
320: ifne          337       // \ if (-(loc4 % 2) == 0) { GOTO 337
323: iload         4         // / loc4
325: iconst_1                // | 1
326: ishr                    // | loc4 >> 1
331: istore        4         // \ loc4 = (loc4 >> 1)
334: goto          369
337: iload         4         // / loc4
339: iconst_2                // | 2
340: ixor                    // | loc4 ^ 2
341: istore        4         // \ loc4 = loc4 ^ 2
343: iload         6         // / loc6
345: iconst_2                // | 2
346: ishl                    // | loc6 << 2
351: ifne          362       // \ if ((loc6 << 2) == 0) { GOTO 362
355: goto          729       // this is the else clause, going directly to the bad message
362: iinc          4, -1     // loc4++
369: iload         6         // / loc6
371: iload         4         // | loc4
373: ishl                    // | loc6 << loc4
374: iload         4         // | loc4
376: iconst_5                // | 5
377: irem                    // | loc4 % 5
378: ixor                    // | (loc6 << loc4) ^ (loc4 % 5)
379: istore        6         // \ loc6 = (loc6 << loc4) ^ (loc4 % 5)
381: iload         6         // / loc6
383: iconst_2                // | 2
384: ishl                    // | loc6 << 2
385: iload         4         // | loc4
391: if_icmpne     409       // \ if ((loc6 << 2) == loc4) { GOTO 409
398: iload         4         // / loc4
400: bipush        6         // | 6
406: ixor                    // | loc4 ^ 6
407: istore        4         // \ loc4 = loc4 ^ 6
409: iload         4         // / loc4
411: iconst_1                // | 1
416: if_icmpgt     311       // \ if (loc4 > 1) { GOTO 311
// end of loop

419: iload         6         // loc6
// from here and below, the code takes the hashCode() value of that "XVCe" string I decrypted earlier
// and compares it with the loc6 value
// if they match, the "Congratulations" message is printed

// otherwise, a jump to 729 is taken, where the "Sorry" message gets printed
```

Looks like I found the algorithm that validates the user code.
There might be a better way, but the only solution I thought of was to bruteforce it.
And I did so, by writing this bruteforcer:

Valid codes bruteforcer

```
int user_code_temp, user_code_crc, valid_code_crc;
```

```
// a bit of optimization
valid_code_crc = "XVCe".hashCode();

// outer, "permutation" loop
for(int user_code = 1; user_code != 0; user_code++) {
    user_code_temp = user_code;
    user_code_crc = user_code_temp << 8;

    // inner, basic validation loop
    for(int j = 0; user_code_temp > 1; j++) {
        if ((-(user_code_temp%2)) == 0) {
            user_code_temp >>= 1;
        } else {
            user_code_temp = user_code_temp ^ 2;
            if (user_code_crc << 2 == 0) {
                // a bit more optimizations
                break;
            }
            user_code_temp--;
        }
        user_code_crc = (user_code_crc << user_code_temp) ^ (user_code_temp % 5);
        if ((user_code_crc << 2) == user_code_temp) {
            user_code_temp = user_code_temp ^ 6;
        }
    }

    // second stage validation
    // at this point, if this validation passes, we print the valid input value
    if (user_code_crc == valid_code_crc) {
        System.out.println(String.format("%d", user_code));
    }
}
```

Running it takes about 3 minutes to check the whole **32 bit integer** range from **+2 147 483 647** to **-2 147 483 648**.
In the end, only these 128 numbers (yes, they are all negative) were flagged as valid:

Valid codes							
-2147473076	-2130695860	-2113918644	-2097141428	-2080364212	-2063586996	-2046809780	-2030032564
-2013255348	-1996478132	-1979700916	-1962923700	-1946146484	-1929369268	-1912592052	-1895814836
-1879037620	-1862260404	-1845483188	-1828705972	-1811928756	-1795151540	-1778374324	-1761597108
-1744819892	-1728042676	-1711265460	-1694488244	-1677711028	-1660933812	-1644156596	-1627379380
-1610602164	-1593824948	-1577047732	-1560270516	-1543493300	-1526716084	-1509938868	-1493161652
-1476384436	-1459607220	-1442830004	-1426052788	-1409275572	-1392498356	-1375721140	-1358943924
-1342166708	-1325389492	-1308612276	-1291835060	-1275057844	-1258280628	-1241503412	-1224726196
-1207948980	-1191171764	-1174394548	-1157617332	-1140840116	-1124062900	-1107285684	-1090508468
-1073731252	-1056954036	-1040176820	-1023399604	-1006622388	-989845172	-973067956	-956290740
-939513524	-922736308	-905959092	-889181876	-872404660	-855627444	-838850228	-822073012
-805295796	-788518580	-771741364	-754964148	-738186932	-721409716	-704632500	-687855284
-671078068	-654300852	-637523636	-620746420	-603969204	-587191988	-570414772	-553637556
-536860340	-520083124	-503305908	-486528692	-469751476	-452974260	-436197044	-419419828
-402642612	-385865396	-369088180	-352310964	-335533748	-318756532	-301979316	-285202100
-268424884	-251647668	-234870452	-218093236	-201316020	-184538804	-167761588	-150984372
-134207156	-117429940	-100652724	-83875508	-67098292	-50321076	-33543860	-16766644

Alright, I did it! Crackme solved.