Crackmes.de – GordonBM's Reverse Keygenme

Posted By Johannes On July 9, 2014 @ 19:22 In information security | No Comments

This blog posts presents my solution to the *Reverse Keygen* by GordonBM's on the "reversers's playground" at crackmes.de [1]. I first used IDA Pro to disassembler to get the CIL [2] representation of the key generator. You can see the full listing of the key generator function here here [3].

The code responsible for generating the key is in the method `pump`. This function receives the message as the first argument. The function generates the key in either the simple of hard mode based on the check box status passed as the second argument. The function returns the key back to the caller, who simply displays the value by copying it to the message text box. These are the first few lines of `pump`:

```
.method public hidebysig instance string pump(string str, bool check)
                                        // CODE XREF: button1_Click+25p
{
.maxstack 5
.locals init (char[] V0,
                string[] V1,
                string V2,
                int32 V3,
                string V4,
                bool V5)
nop
ldarg.1
callvirt instance char[] [mscorlib]System.String::ToCharArray()
stloc.0
ldloc.0
ldlen
conv.i4
newarr   [mscorlib]System.String
stloc.1
ldstr    ""
stloc.2
ldarg.2
ldc.i4.0
ceq
stloc.s  5
ldloc.s  5
brtrue.s loc_4A1                        // branch if check is FALSE
nop
ldc.i4.0
stloc.3
br.s     loc_48F
```

The lines roughly translate to the following pseudo C# code:

```
string pump(string szMessage, bool bHardcoreMode) {
    /* v0:   rgchMessage (char array representation of szMessage)
       v1:   rgszResult  (empty string array of same size as v0)
       v2:   szResult    (empty null-terminated string)
       v3:   iIndex      (index of char array)
    */
    char[] rgchMessage = szMessage.ToCharArray();
    string[] rgszResult = new string[szMessage.Length];
    string szResult = "";

    if( bHardcoreMode ) {
        // do hardcore mode
    }
    else {
        // do simple mode at loc_4A1
    }
}
```

The snippet converts the message string to an array of characters and stores the result in `rgchMessage`. It also initializes two results variables `szResult` (an empty string) and `szResult` (an empty array with same size as the message). After that, the code either continues with the hardcore keygenerator (when the checkbox is set) or jumps to simple mode key generating at `loc_4A1`. Let's start with the easy version.

Simple Mode

Let's first investigate the simple version, which starts at line `loc_4A1`

```
loc_4A1:                              // CODE XREF: pump+1Fj
    nop
    ldc.i4.0
    stloc.3
    br.s     loc_4C1

(...)

loc_4C1:                              // CODE XREF: pump+64j
    ldloc.3
    ldarg.1
    callvirt instance int32 [mscorlib]System.String::get_Length()
    clt
    stloc.s  5
    ldloc.s  5
    brtrue.s loc_4A6
    nop

loc_4D1:                              // CODE XREF: pump+5Fj
    ldloc.2
    stloc.s  4
    br.s     loc_4D6

loc_4D6:
    ldloc.s  4
    ret
}
```

This snippet doesn't to much. We are probably looking at a `for` loop that iterates over all characters in `szMessage`. After the `for`-loop, the method `pump` returns the string in `szResult`. Here is the code in pseudo C#:

```
// loc_4A1:
int iIndex = 0          // in v3

// loc_4C1:
if( iIndex < szMessage.Length )
    // jump to loc_4A6

return szResult;
```

The body of the loop is at `loc_4A6`:

```
loc_4A6:                              // CODE XREF: pump+8Ej
    nop
    ldloc.1
    ldloc.3
    ldloc.0
    ldloc.3
    ldelem.u2
    call     string [mscorlib]System.Convert::ToString(int32)
    stelem.ref
    ldloc.2
    ldloc.1
    ldloc.3
    ldelem.ref
    call     string [mscorlib]System.String::Concat(string, string)
    stloc.2
    nop
    ldloc.3
    ldc.i4.1
    add
    stloc.3
```

We can decompile it to the following pseudo C# code:

```
/* if rgchMessage = {'t', ...} then nCharacter = 116 and
   szCharacterCode = "116" */
unsigned short nCharacter = rgchMessage[iIndex]; // ldelem.u2
string szCharacterCode = nCharacter.ToString();
rgszResult[iIndex] = szCharacterCode;
```

```
szResult = string.Concat(szResult, szCharacterCode);
iIndex++;
// continue at loc_4C1
```

Those few lines implement the whole magic of the simple encryption:

- They take the letter at `iIndex` into the message. For example, if the message is "test" and `iIndex` is 2, then the character would be `e`
- The character is implicitly converted to an unsigned short. By doing this we get the ASCII code of the letter. In our example for letter `e` we get `101`.
- The integer code is then converted to a string, so `101` becomes `"101"`

Putting all parts together leads to the following key generation algorithm:

```
string pump(string szMessage, bool bHardcoreMode) {
    /* v0:   rgchMessage (char array representation of szMessage)
       v1:   rgszResult  (empty string array of same size as v0)
       v2:   szResult    (empty null-terminated string)
       v3:   iIndex      (index of char array)
    */
    char[] rgchMessage = szMessage.ToCharArray();
    string[] rgszResult = new string[szMessage.Length];
    string szResult = "";

    if( bHardcoreMode ) {
        // do hardcore mode
    }
    else {
        // do simple mode
        for( int iIndex = 0; iIndex < szMessage.Length; iIndex++ ) {
            unsigned short nCharacter = rgchMessage[iIndex];
            string szCharacterCode = nCharacter.ToString();
            rgszResult[iIndex] = szCharacterCode;
            szResult = string.Concat(szResult, szCharacterCode);
        }
        return szResult;
    }
}
```

Reversing encrypted text is straightforward. The only tricky part is differentiating between the two digit ASCII codes and the three digits ones. The latter always start with 1, while the former never do (the ASCII codes 10 to 19 are not printable). The following Python script first tokenizes the key into ASCII codes, and then transforms those code back to characters:

```
import argparse
import re


def decrypt(message):
    res = ""
    for c in re.findall('([2-9]\d|1\d{2})', message):
        res += chr(int(c))
    return res

if __name__=="__main__":
    parser = argparse.ArgumentParser(description="Simple Keygen")
    parser.add_argument("encrypted")
    args = parser.parse_args()
    print(decrypt(args.encrypted))
```

Example:

```
$ python simple_keygen.py 84104105115327311532653284101115116
This Is A Test
```

Hardcore Mode

Decompiling the Algorithm

The hardcore mode starts right after the branch to the simple version:

```
    nop
    ldc.i4.0
    stloc.3
    br.s      loc_48F
```

```
(...)

loc_48F:                                // CODE XREF: pump+24j
    ldloc.3
    ldarg.1
    callvirt instance int32 [mscorlib]System.String::get_Length()
    clt
    stloc.s  5
    ldloc.s  5
    brtrue.s loc_466
    nop
    br.s     loc_4D1

(...)

loc_4D1:                                // CODE XREF: pump+5Fj
    ldloc.2
    stloc.s  4
    br.s     loc_4D6

loc_4D6:
    ldloc.s  4
    ret
}
```

This snippet translates to:

```
int iIndex = 0          // in v3

// loc_48F
if( iIndex < szMessage.Length )
    // jump to loc_466

return szResult;
```

which, as for the simple mode, is simply a loop over all characters. The body of the loop at `loc_466` reads as:

```
loc_466:                                // CODE XREF: pump+5Cj
    nop
    ldloc.1
    ldloc.3
    ldloc.0
    ldloc.3
    ldelem.u2
    conv.r8
    ldarg.0
    ldarg.1
    callvirt instance int32 [mscorlib]System.String::get_Length()
    call     instance float64 Encrypter.Goodies.Engine::ran(int32 l)
    add
    call     string [mscorlib]System.Convert::ToString(float64)
    stelem.ref
    ldloc.2
    ldloc.1
    ldloc.3
    ldelem.ref
    call     string [mscorlib]System.String::Concat(string, string)
    stloc.2
    nop
    ldloc.3
    ldc.i4.1
    add
    stloc.3

loc_48F:                                // CODE XREF: pump+24j
    ldloc.3
    ldarg.1
    callvirt instance int32 [mscorlib]System.String::get_Length()
    clt
    stloc.s  5
    ldloc.s  5
    brtrue.s loc_466
    nop
    br.s     loc_4D1
```

or in C#:

```
double dbCharacter = (double)rgchMessage[iIndex];
double dbRanResult = ran(szMessage.Length);
double dbSum = dbCharacter + dbRanResult;
string szSum = dbSum.ToString();
rgszResult[iIndex] = szSum;
szResult = string.Concat(szResult, szSum);
iIndex++;
// continue loop at loc_48F
```

So the hardcore mode, just like the simple version, operates on the ASCII codes of the letters and concatenates the results. This time, however, there is an additional method `ran` whose return value is added to the ASCII character codes. Also the code uses double types instead of integers. Let's try to decompile `ran`:

```
.method private hidebysig instance float64 ran(int32 l) // CODE XREF: pump+34p
{
    .maxstack 5
    .locals init (float64 V0,
                  int32 V1,
                  float64 V2,
                  bool V3)
    nop
    ldc.r8   0.0
    stloc.0
    ldc.i4.0
    stloc.1
    br.s     loc_53A

(...)

loc_53A:                                // CODE XREF: ran+Dj
    ldloc.1
    ldarg.1
    ldc.i4.2
    mul
    clt
    stloc.3
    ldloc.3
    brtrue.s loc_4EF
    ldloc.0
    stloc.2
    br.s     loc_548

loc_548:
    ldloc.2
    ret
}
```

The function decompiles to:

```
double ran(int iStringLength) {
    double dbV0 = 0.0;
    int iCounter = 0;                // in v1

    // loc_53A:
    double dbTemp = 2*iStringLength;
    if( dbTemp > dbV0 )
        // goto loc_4EF

    return dbV0;
}
```

At `loc_4EF` we find:

```
loc_4EF:                                // CODE XREF: ran+62j
    nop
    ldloc.1
    ldc.i4.2
    rem
    ldc.i4.0
    ceq
    ldc.i4.0
    ceq
    stloc.3
    ldloc.3
```

```
        brtrue.s loc_522
        nop
        ldloc.0
        ldloc.0
        ldc.r8    2.0
        mul
        ldarg.0
        ldfld     class [mscorlib]System.Random Encrypter.Goodies.Engine::random
        ldc.i4.0
        ldc.i4.2
        callvirt instance int32 [mscorlib]System.Random::Next(int32, int32)
        ldloc.1
        ldc.i4.6
        xor
        mul
        conv.r8
        add
        add
        stloc.0
        nop
        br.s      loc_535
```

which in C# is:

```
if( iCounter % 2 )
    // goto loc_522
else
    double mul = dbV0*2.0
    Random rnd = new Random();
    int nRandZeroOrOne = rnd.Next(0, 2);
    int iXOR = iCounter ^ 6;
    double mul2 = (double) iXOR * nRandZeroOrOne;
    mul2 = mul2 + mul;
    dbV0 = mul2 + dbV0;
    // go to loc_535
```

If the `iCounter` is odd, the snippet at `loc_522` gets executed:

```
    loc_522:                              // CODE XREF: ran+1Bj
        nop
        ldloc.0
        ldloc.0
        ldc.r8    2.0
        mul
        ldc.i4.0
        conv.r8
        add
        add
        stloc.0
        nop

    loc_535:
    (...)
```

which translates to the following C# pseudo code:

```
double mul = 2.0 * dbV0;
double res = mul + 0.0;
res = res + dbV0;
dbV0 = res
```

We can further simplify this to:

```
dbV0 = 3.0*dbV0;
```

After the two cases for `iCounter` even and odd follows line `loc_535`:

```
    loc_535:                              // CODE XREF: ran+40j

        nop
        ldloc.1
        ldc.i4.1
        add
        stloc.1
```

These lines just increment the `iCounter` variable:

```
iCounter++;
```

So to summarize all parts, this is the whole `ran` method:

```
double ran(int iStringLength) {
    double dbV0 = 0.0;

    for(int iCounter = 0; iCounter < 2.0*iStringLength; iCounter++ ) {
        if( iCounter % 2 )
            dbV0 = 3.0*dbV0;
        else
            double mul = dbV0*2.0
            Random rnd = new Random();
            int nRandZeroOrOne = rnd.Next(0, 2);
            int iXOR = iCounter ^ 6;
            double mul2 = (double) iXOR * nRandZeroOrOne;
            mul2 = mul2 + mul;
            dbV0 = mul2 + dbV0;
    }

    return dbV0;
}
```

We can simplify this code by unrolling the even and odd case:

```
double ran(int iStringLength) {
    double dbV0 = 0.0;

    for(int iCounter = 0; iCounter < iStringLength; iCounter++ ) {
        double mul = dbV0*2.0
        Random rnd = new Random();
        int nRandZeroOrOne = rnd.Next(0, 2);
        int iXOR = (2*iCounter) ^ 6;
        double mul2 = (double) iXOR * nRandZeroOrOne;
        mul2 = mul2 + mul;
        dbV0 = mul2 + dbV0;
        dbV0 = 3.0*dbV0;
    }

    return dbV0;
}
```

Putting all calculations in one line leads to:

```
double ran(int iStringLength) {
    Random rnd = new Random();
    double dbV0 = 0.0;

    for(int iCounter = 0; iCounter < iStringLength; iCounter++ ) {
        result += 3*( ((2*iCounter) ^ 6) * rnd.Next(0, 2) + dbV0*3)

    return dbV0;
```

So to summarize: The hardcore mode also iterates over all characters and takes the ASCII codes. But then it adds random noise with the function `ran`. The hardest part about reversing this key generation algorithm is definitely the uncertainty of this noise. The next section investigates this noise in more detail.

Learning about the Noise

Strings of Length 1

Let's start with the easiest case – strings of length 1. The for loop is executed exactly once. The expression (2*iCounter) ^ 6 evaluates to 6. The random function rnd.Next(0, 2) either evaluates to 0 or to 1. This means the result of `ran` is either 0 or 18.

Strings of Length 2

The `for`-loop gets executed twice for two letter words. After the first pass, `dbV0` holds either 0 or 18 as seen before. For the next iteration (2*iCounter) ^ 6 evaluates to 4. The random function rnd.Next(0, 2) again is 0 or 1. So if `dbV0` was 0 after the first iteration, we get either 0 or 4. If, on the other hand, `dbV0` was 18, we get the value 162 or 174.

Strings of Length n

The following recursive Python function generates all possible noise values for a given length:

```
def ran_values(length, values=None, pos=0, val=0):
    """get all potential random values for length as list """

    if values is None:
        values = set()
    if pos == length:
        values.add(val)
    else:
        xor = (2*pos) ^ 6
        pos += 1

        # case 1: rand is 0
        next_val = 9*val
        ran_values(length, values, pos, next_val)

        # case 2: rand is 1
        next_val = 3*(val*3 + xor)
        ran_values(length, values, pos, next_val)

    return sorted(values)

if __name__ == "__main__":
    for i in range(1,7):
        tmp = "<tr><td>{}</td><td>{}</td></tr>"
        print(tmp.format(i, ', '.join([str(x) for x in ran_values(i)])))
```

Running it yields these values:

iStringLength	noise values
1	0, 18
2	0, 12, 162, 174
3	0, 6, 108, 114, 1458, 1464, 1566, 1572
4	0, 54, 972, 1026, 13122, 13176, 14094, 14148
5	0, 42, 486, 528, 8748, 8790, 9234, 9276, 118098, 118140, 118584, 118626, 126846, 126888, 127332, 127374
6	0, 36, 378, 414, 4374, 4410, 4752, 4788, 78732, 78768, 79110, 79146, 83106, 83142, 83484, 83520, 1062882, 1062918, 1063260, 1063296, 1067256, 1067292, 1067634, 1067670, 1141614, 1141650, 1141992, 1142028, 1145988, 1146024, 1146366, 1146402

The number of noise values doubles for each extra letter. The only exception is the step from 3 to 4 characters which both lead to 8 noise values. This anomaly is due to the XOR expression that becomes zero for `iCounter = 3`.

Not only do the number of potential noise values increase, the number of digits also varies more and more. For 6 letter strings, for example, the noise could be 0 up to the 7 digit variable 1146402. So an additional problem is to know how many characters the message has. The next section examines the mean key length for different message sizes.

Average Key Length

To guess how many characters were in the message, we need to have a statistic for the average key length given a certain message length. The following Python script simply generates all potential noise values with the method `ran_values` shown above. It then adds the average ASCII code value. For the average ASCII code, I'm assuming the characters of the message are withing the range 32 to 126. This ranges includes all printable characters. The mean character would therefore have a code of 79. This is, of course, a very rough estimate. Better algorithms would determine the mean based on average messages. But to just guess the string length it should be fine. Here's a script that lists the average length of the key for different message lengths up to 14:

```
from noise_overview import ran_values

ASCII_RANGE = [32, 126]

def generate_len_db(limit):
    """generate a list of mean key lengths for all msg lengths

        Args:
            limit: up to which msg length should the mean be calculated
        Returns:
            a list of mean key lengths, index i corresponds to msg length i
    """
    len_db = []
    noise_values = set()
    mean_off = sum(ASCII_RANGE)/float(2)
```

```
    for i in range(0,limit):
        noise_values = ran_values(i)
        mean_val = 0
        for noise in noise_values:
            char = noise + mean_off
            mean_val += len(str(char))
        mean_val *= i
        mean_val /= len(noise_values)
        len_db.append(mean_val)
    return len_db

for i,l in enumerate(generate_len_db(15)):
    print("<tr><td>{}</td><td>{}</td></tr>".format(i, l))
```

Note that the code takes a while to complete. But the values need to be computed only once and can later be hard coded into the reverse key generator.

message length	avg. key length
0	0
1	2
2	5
3	9
4	16
5	23
6	33
7	44
8	56
9	72
10	90
11	110
12	132
13	156
14	182

Given the average key length we can estimate the length of the message.

Average Key Length

The example key given by the GordonBM is `9247109931023928283286380308924708453882326686447837` and has 52 characters. The closest value in the above table is 56 which corresponds to a message length of 8. The real message "GordonBM" indeed has 8 characters. The following Python snippet returns the best guess for the message length based on the hardcoded results from the previous section:

```
def guess_length(key):
    """get the best guess for the length of the message based on hardcoded
       mean key lengths"""

    len_db = [0,2,5,9,16,23,33,44,56,72,90,110,132,156,182]
    len_msg = len(key)
    diffs = [abs(c-len_msg) for c in len_db]
    val, idx = min((val, idx) for (idx, val) in enumerate(diffs))
    return (idx, val)
```

With the above code we can guess the expected length of the message. Given this information we can then calculate the noise values. The "only" remaining part is to actually crack the key. I'm doing this with brute force.

Brute-Forcing the Message

With the length information of the message and, more importantly, the resulting noise values, we can brute force the message. The following algorithm starts at the beginning of the key and iterates over all potential noise values. It then checks if any character from the `ASCII_RANGE = [32, 126]` could have produced the key at hand. If yes, the algorithm advances the resulting number of digits and repeats the procedure. If it manages to reach the end of the key with all valid message characters (meaning they are in the `ASCII_RANGE`), then the function tests if the length of the message checks out and simply prints the message to stdout:

```
ASCII_RANGE = [32, 126]

def brute_force(crypt, msg_len, noises, pos=0, res=""):
```

```
    """brute-force potential messages

        Prints all potential messages (characters in ASCII_RANGE)

        Args:
            crypt: the key that should be reverse
            msg_len: the length of the message
            noises: the list of potential noise values for the given length

        Returns:
            nothing, prints all strings to stdout
    """
    for noise in noises:
        low, upp = (tmp + noise for tmp in ASCII_RANGE)
        for span in range(len(str(low)), len(str(upp))+1):
            digits = crypt[pos:pos+span]
            code = int(digits)
            if low <= code <= upp:
                msg_char = chr(code - noise)
                concat = res + msg_char
                if pos+span >= len(crypt):
                    if len(concat) == msg_len:
                        print(concat)
                    else:
                        pass # string does not match expected nr of chars
                else:
                    brute_force(crypt, msg_len, noises, pos+span, concat)
```

Putting all together

The entire reverse key generators looks as follows:

```
"""Reverse key generator for GordonBM's Reverse Keygenme
   see http://www.crackmes.de/users/gordonbm/reverse_keygenme/
   (hardcore mode)"""
import argparse

ASCII_RANGE = [32, 126]

def ran_values(length, values=None, pos=0, val=0):
    """get all potential random values for length as list """

    if values is None:
        values = set()
    if pos == length:
        values.add(val)
    else:
        xor = (2*pos) ^ 6
        pos += 1

        # case 1: rand is 0
        next_val = 9*val
        ran_values(length, values, pos, next_val)

        # case 2: rand is 1
        next_val = 3*(val*3 + xor)
        ran_values(length, values, pos, next_val)

    return sorted(values)

def guess_length(key):
    """get the best guess for the length of the message based on hardcoded
       mean key lengths"""

    len_db = [0,2,5,9,16,23,33,44,56,72,90,110,132,156,182]
    len_msg = len(key)
    diffs = [abs(c-len_msg) for c in len_db]
    val, idx = min((val, idx) for (idx, val) in enumerate(diffs))
    return (idx, val)


def brute_force(crypt, msg_len, noises, pos=0, res=""):
    """brute-force potential messages

        Prints all potential messages (characters in ASCII_RANGE)

        Args:
            crypt: the key that should be reverse
```

```
            msg_len: the length of the message
            noises: the list of potential noise values for the given length

        Returns:
            nothing, prints all strings to stdout
    """
    for noise in noises:
        low, upp = (tmp + noise for tmp in ASCII_RANGE)
        for span in range(len(str(low)), len(str(upp))+1):
            digits = crypt[pos:pos+span]
            code = int(digits)
            if low <= code <= upp:
                msg_char = chr(code - noise)
                concat = res + msg_char
                if pos+span >= len(crypt):
                    if len(concat) == msg_len:
                        print(concat)
                    else:
                        pass # string does not match expected nr of chars
                else:
                    brute_force(crypt, msg_len, noises, pos+span, concat)

def crack(key):
    """crack the key"""
    msg_len, error = guess_length(key)
    print("message length is probably {}, (delta {})".format(msg_len, error))
    noise = ran_values(msg_len)
    print("there are {} different noise values".format(len(noise)))
    print("potential messages are:")
    brute_force(key, msg_len, noise)

if __name__ == "__main__":
    """ reverses keys like:
        142641551691142
        9247109931023928283286380308924708453882326686447837
    """
    parser = argparse.ArgumentParser(description="Hardcore Reverse Keygen")
    parser.add_argument('key', help="the key to be reversed")
    args = parser.parse_args()
    crack(args.key)
```

Let's test it with the key `142641551691142` which was produced entering the message "`test`":

```
$ python hardcore_keygen.py 142641551691142
message length is probably 4, (delta 1)
there are 8 different noise values
potential messages are:
test
```

Nice! It found the message. Now let's try the longer example key `9247109931023928283286380308924708453882326686447837`:

```
$ python hardcore_keygen.py 9247109931023928283286380308924708453882326686447837
message length is probably 8, (delta 4)
there are 128 different noise values
potential messages are:
_ordonBe
_ordonBM
_ordon*e
_ordon*M
_ordWnBe
_ordWnBM
_ordWn*e
_ordWn*M
_orLonBe
_orLonBM
_orLon*e
_orLon*M
_orLWnBe
_orLWnBM
_orLWn*e
_orLWn*M
_oZdonBe
_oZdonBM
_oZdon*e
_oZdon*M
```

```
_oZdWnBe
_oZdWnBM
_oZdWn*e
_oZdWn*M
_oZLonBe
_oZLonBM
_oZLon*e
_oZLon*M
_oZLWnBe
_oZLWnBM
_oZLWn*e
_oZLWn*M
GordonBe
GordonBM
Gordon*e
Gordon*M
GordWnBe
GordWnBM
GordWn*e
GordWn*M
GorLonBe
GorLonBM
GorLon*e
GorLon*M
GorLWnBe
GorLWnBM
GorLWn*e
GorLWn*M
GoZdonBe
GoZdonBM
GoZdon*e
GoZdon*M
GoZdWnBe
GoZdWnBM
GoZdWn*e
GoZdWn*M
GoZLonBe
GoZLonBM
GoZLon*e
GoZLon*M
GoZLWnBe
GoZLWnBM
GoZLWn*e
GoZLWn*M
```

Because the message is longer (and therefore also the potential noise values), we get not just one message back but 64 slightly different ones. Among those values is also the original message "`GordonBM`". But without additional knowledge about the message we can't do better than provide the extensive list of potential message.

Article printed from Blog of Johannes Bader: **http://www.johannesbader.ch**

URL to article: **http://www.johannesbader.ch/2014/07/crackmes-de-gordonbms-reverse-keygenme/**

URLs in this post:

[1] crackmes.de: **http://www.crackmes.de/users/gordonbm/reverse_keygenme/**
[2] CIL: **http://en.wikipedia.org/wiki/Common_Intermediate_Language**
[3] here: **https://github.com/baderj/crackmes/blob/master/Reverse_Keygenme_by_GordonBM/encrypter.goodies.ci**

Copyright © 2014 Blog of Johannes Bader. All rights reserved.