Three different algorithms for constructing licensing systems, their advantages and disadvantages using C#.NET environment.

Author: Artem Los

12345-67891

BAFYRS-JKPRFN-TMRWSE

KBXGH-EGVMY-PAHET-QVBOM

Abstract A key validation algorithm is one of the important parts in the protection of a computer application. Even if an already existing API is to be used, it is important to understand its weaknesses in order to compare it with alternative ones. Therefore, in this article, three different categories will be described with clear definitions that will make it possible to distinguish between them and allow an analysis of currently existing APIs. Every category is accompanied with examples and in some cases suggestions for further development. The categories described in this article are Checksum based key validation, Pattern based key validation, and Information based key validation. It is going to be found that the choice of a key validation system depends on the information that is to be stored in the key. It is also concluded that at this point it would be better to use online key validation instead.

Keywords Licensing systems, key validation, software protection, serial key.

Email: artem@artemlos.net

Page count: 13

Introduction

The problem of finding the appropriate licensing system might not seem that important when developing applications of different kinds. Usually, we focus on making the actual application as good as possible and leaving the licensing system to the end. However, if the protection against illegal use is important, licensing systems of different kinds have to be considered.

There are at this point at least three ways of protecting computer application. The first way is to use an already existing service like Windows Store. The second way is to use already existing APIs. The third way is to build your own system.

In this paper, I would like to describe three different ways of constructing your own key validation algorithm that works without internet connection, and at the same time convey the weaknesses of each of them. Even if you are not going to build your own algorithm, one of the aims of this article is to give the ability to analyse key validation algorithms once you know what group they are from. For this purpose, every section contains a strict definition of a given system, so that when you have found an already existing API, you can spot which group it belongs to and thus get to know its weaknesses.

History

- This article was written 2014.02.09.
- Added a bibliography, changed the cover page. 2014.04.20.

1 Checksum based key validation

This is the most common, very simple-to-implement licensing system that uses a function to calculate a checksum of specific data. A strict definition is:

```
A key validation algorithm where two types of data are present to the
user in such a way that the data2 is directly dependent on data1.
```

Usually, this means that the `data1` is the customer's name and `data2` is the serial key that is sent along with customer's name. Ideally, the function that generates `data2` based on `data1` is destructive, that is, it should not be possible to find `data1` given `data2`. When the serial key is to be validated, the end-user application has to check whether the relation between `data1` and `data2` is true.

Another way of looking at it is:

$$x = \text{name of the customer } (\texttt{data1})$$
$$f(x) = \text{the serial key } (\texttt{data2})$$

If it can be said that a key validation is check sum based, it is has following weaknesses.

Weaknesses

- If $f(x)$ is known or at least if there is a relation that can be seen between x_1 and $f(x_1)$, the key algorithm can be found.
- Requires the end-used program(client) to have $f(x)$ embedded into the code in some way. $\implies$ If the program is disassembled, $f(x)$ can be found.
- During runtime, a temporary hash is going to be calculated and stored in a variable. This value might be obtained by an external application.
- Does not give more information than if the key is valid or invalid.

2 Pattern based key validation

This is the validation algorithm that is what is usually referred to as a serial key. It is based on the idea that only specific combinations of characters are to be considered valid, hence the name 'pattern based'. The strict definition is:

```
A key validation algorithm where one type of data (key) is present to the
user, and which is validated by a set pattern.
```

Since there are so many applications that have used this technique, it will not be possible to list all kinds of pattern based key validations, so it can be noted that as long as a set pattern is defined and there is only one type of data, it can be referred to as pattern based key validation.

Example 1 A pattern can be as simple as a restriction of which characters can be entered and how long the key can be. It might also be constructed in such a way that the last digit is dependent on the sum of all other digits in the key, for instance: say the key length is 10, and the last digit is the sum of all digits in front of it modulo 7. Below, an example of such key:

1234567891

The last digit is 1 since,

$$1+2+3+4+5+6+7+8+9 \mod 7 = 1$$

This can, of course, be made even more complex by adding more rules to the pattern. As a result, instead of allowing 10^9 possible combinations, because there are now 8 places where digits can be placed in any way, and for each of these ways there is going to be one 'last digit', there are now 10^8 different combinations that would satisfy this rule. There more rules, the less combinations that will satisfy that pattern.

Design the system In order to construct such a system, there are at least two different approaches. One would be to work with Regular Expression (Reg Ex)[1] or to use a specialized API for this particular task. In most cases, Reg Ex allows any kind of pattern validation that can later be extended by additional code logic. However, in case there is a need to set up a pattern based validation very quickly, specialized API can be used.

Reg Ex is a powerful tool that opens doors for various pattern recognitions. This makes it possible to use the pre-built rules to construct any kind of pattern based validation. In order to check keys using the pattern that was defined in the introduction, following code can be used:

```
1  static void Main(string[] args)
2  {
3      System.Text.RegularExpressions.Regex Check = new System.Text.
           RegularExpressions.Regex("\\d{10}");
4
5      string key = "1234567891";
6
7      if(Check.IsMatch(key))
8      {
9          //The key length is 5 and it consists of only digits,
10         //so the probability is high that we have the right key
11
12         int sum = 0;
13
14         for (int i = 0; i < key.Length -1; i++)
15         {
16             sum += key[i];
17         }
18
19         if( sum % 7 == (int)Char.GetNumericValue(key[key.Length-1]))
20         {
21             Console.WriteLine("Valid");
22         }
23         else
24         {
25             Console.WriteLine("Invalid");
26         }
27     }
28     else
29     {
30         Console.WriteLine("Invalid");
31     }
32
33     Console.ReadLine();
34 }
```

[1]Read more: http://msdn.microsoft.com/en-us/library/system.text.regularexpressions.regex(v=vs.110).aspx, Last accessed 2014.02.03

The Reg Ex is used to see if the input string contains only 10 digits. Later on, using some logic, it can be checked that the last digit is following the rule.

In 2009, an API[2] was developed that would work as a specialized API for pattern based key validation, however, it should be noted that this is a universal idea, and it can be adjusted and optimized for specific needs.

The main idea is to construct two methods, one that will generate a key that follows this pattern and one that will validate a key to see if it follows the pattern. In the API, it was chosen to allow the developer to store the pattern as a string, where different symbols represent fragments of a small pattern(see Table 1). In order to make it even more complex, an ability to add functions that would calculate the modulo of a range of numbers in the key was added (see Table 2).

Now, in order to set up a pattern with the same rules as the one that was described in the beginning of this example, following pattern would be entered.

```
#########[+1,8/7]
```

Symbol	Definition
#	A random number
*	A random uppercase letter
@	A random lowercase letter
%	A random lowercase or uppercase letter
?	A random number or uppercase letter
!	A random number or lowercase letter

Table 1: The basic definitions of small pattern fragments.

Symbol	Description	Example
[XY]	Generates a random char from X to Y.	[AC] = either A,B or C [ac] = either a, b or c [35] = either 3,4 or 5
[X/Y]	Takes the char's ASCII value at X mod the number Y.	#[1/7] generates eg. 65 since 6 (54) $\implies$ and 54 mod 7= 5
[+X,Y/Z]	Adds chars' ASCII values (or the digit values if it is a digit) at X and Y and takes mod Z.	##[+1,2/7] generates eg.182, where 1+8 mod 7 = 6

Table 2: More complex fragments for construction of a pattern. Note that for the second and the third rule, the character that the function is using cannot not be after the the function itself, i.e. the key is read from left to right.

Example 2 Not only restriction of characters can be used as an example of pattern based key validation, but also explicit use of mathematical functions. The idea of this key validation was suggested by *PaulG* on StackOverflow[3].

[2] `http://skbl.clizware.net/help.html`. Last accessed 2014.01.31

[3] `http://stackoverflow.com/a/3007632/1275924`. Last accessed 2014.01.31

The idea behind this key validation technique is based on a simple concept in mathematics – a function.

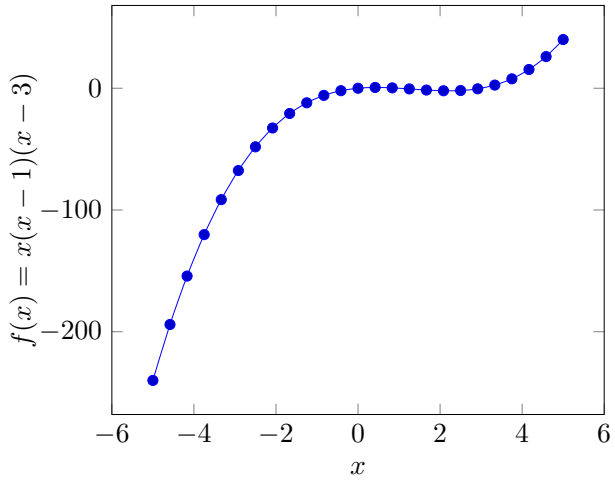

Figure 1: A graph for generation and validation of serial keys.

In order to generate a key, a specific number of points should be noted down. For simplicity, three points are going to be used. In order to set specific boundaries for the size of the input and the output of the function, modular arithmetic can be used. The final points can be converted into for instance, base 26, which will only include letters in the output. The code below illustrates this idea.

```
1  static void Main(string[] args)
2  {
3      Func<int, long> function = x => x * (x - 1) * (x - 3);
4
5      string key2 = CreateKey(function, 3456);
6      Console.WriteLine(ValidateKey(key2, function, 3456));
7      Console.ReadKey();
8  }
9
10 static string CreateKey(Func<int,long> f, int mod)
11 {
12     System.Security.Cryptography.RNGCryptoServiceProvider rng = new System.
           Security.Cryptography.RNGCryptoServiceProvider();
13     byte[] rndBytes = new byte[4];
14     rng.GetBytes(rndBytes);
15     int rand = modulo(BitConverter.ToInt32(rndBytes, 0), mod);
16     int key = modulo(f(rand), mod);
17
18     rng.GetBytes(rndBytes);
19     int rand2 = modulo(BitConverter.ToInt32(rndBytes, 0), mod);
20     int key2 = modulo(f(rand2), mod);
21
22     rng.GetBytes(rndBytes);
23     int rand3 = modulo(BitConverter.ToInt32(rndBytes, 0), mod);
24     int key3 = modulo(f(rand3), mod);
25
26
27     decimal outputData = 1; //this could've been 0 too, however, in that case,
           we would need
28                             //to take this into consideration when the key is
                                   deciphered (the length)
```

```
29
30      outputData *= (decimal)Math.Pow(10, mod.ToString().Length);
31      outputData += rand;
32      outputData *= (decimal)Math.Pow(10, mod.ToString().Length); //maybe need a
            one somewhere to fill up the space
33      outputData += key;
34      outputData *= (decimal)Math.Pow(10, mod.ToString().Length);
35
36      outputData += rand2;
37      outputData *= (decimal)Math.Pow(10, mod.ToString().Length);
38      outputData += key2;
39      outputData *= (decimal)Math.Pow(10, mod.ToString().Length);
40
41      outputData += rand3;
42      outputData *= (decimal)Math.Pow(10, mod.ToString().Length);
43      outputData += key3;
44
45      string output = base10ToBase26(outputData.ToString());
46
47      return output;
48  }
49
50  static bool ValidateKey(string key,Func<int,long> f, int mod)
51  {
52      string base10 = base26ToBase10(key);
53      int modLength = mod.ToString().Length;
54
55      for (int i = 0; i < 3; i++)
56      {
57          if (modulo(f(Convert.ToInt32(base10.Substring(1, modLength))), mod) ==
                Convert.ToInt32(base10.Substring(modLength + 1, modLength)))
58          {
59              base10 = base10.Substring(2 * modLength);
60          }
61          else
62          {
63              return false;
64          }
65      }
66
67      return true;
68  }
69
70  static decimal maxModValue()
71  {
72      //this is the maximum length of mod variable considering we
73      //have 3 points (1 point = 2 values).
74      return (decimal.MaxValue.ToString().Length - 1) / 6;
75  }
76
77  /* The functions below are simply to make the keys look better! the main logic
        is above this line. Please copy-paste those functions from: http://dev.
        artemlos.net/func/conf1.txt*/
```

Weaknesses

- Almost all examples of pattern based key validation (except when there is a restriction for length, type of input, i.e. what can be restricted using Reg Ex), there is a `data2` that

is dependent on `data1`. This means that weaknesses of checksum based key validation should be considered when assessing a pattern based key validation.

- A pattern of a valid key makes it more difficult to guess the right key, however, if a pattern allows too many keys to be valid, for example, if it is only restricted for digit input only, the probability is high that the key can be guessed.
- The more keys that are available to the user, the more vulnerable is the pattern. For example, if the user knows that all keys have a common tendency, this can be used to find keys that would satisfy that pattern.

3 Information based key validation

This type of licensing system is one that can be used as an alternative to online based key validation (when the key is checked using an external database). Depending on how it is implemented, it can allow a very strong protection for an application.

```
A key validation algorithm where one type of data is present to the user.
Some information is stored in the data also.
```

There are two different ways of implementing such a system. One is to use symmetrical cryptography, which will reduce the key length, and the other is to use asymmetrical cryptography, which generally will produce a larger key length.

Usually, there is a trade off involved. The securer an algorithm is, the longer the output key will be and the less information can be stored and considered useful (any information except for the checksum).

An algorithm of this kind contains at least two different types of encryptions. One is responsible for the checksum or the signature of the information and the other one is responsible for the encryption of both the checksum and the information. If the information inside the key is not confidential, which it should not be, the second encryption step can be avoided to save key output. This is then a clear example of checksum based key validation, where information is `data1` which affects the checksum `data2`, and thus the limitations of checksum based key validation should be considered when assessing the vulnerability of a given system.

The structure of the information, i.e. what the information is built up from, for instance *date of creation*, *set interval of time*, et cetera, is similar to pattern based key validation. Depending on how the information is designed, and what type of information is stored, they both contribute to a pattern. Therefore, it should still be considered to use the second step of encryption, that is when both the checksum and the useful information are encrypted, even if the useful information in itself is not confidential, the way it is structured poses a threat to the licensing system. The less a user knows about the system, the securer the system is.

Symmetric cryptography SKGL API[4] contains an information storage structure similar to the one in Figure 2. It uses both a checksum to check for alteration of data and it also encrypts it together with the useful information.

A possible key (decrypted), using the key structure in Figure 2 could be similar to:

$$(693937080\ 20120430\ 030\ 000\ 80966)_{10}$$

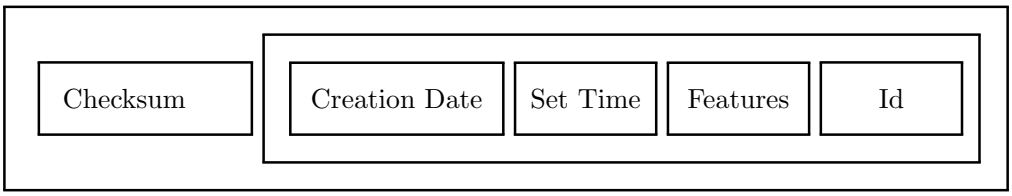

Figure 2: The architecture of a key generated with SKGL API.

Checksum	693937080
Creation date	20120430
Set time	030
Features	000
Id	80966

Table 3: An example of a key that follows the defined structure in Figure 2

In Table 3 an example of the different pieces of information can be seen.

During the construction of this algorithm, one of the conclusions that was drawn is that the checksum should be placed in the beginning of the key because it will contribute to a much greater change in the value of this large number, and thus a single change of the useful information will cause a notable change in the key in base 26.

It was also noticed that it is a good idea to check for the maximum and minimum values that a key can be. For example, the checksum function will output all possible combinations of nine digit numbers except for those smaller than 10^8, that is $\{n|10^8 \leq n \leq 10^9 - 1, n \in \mathbb{Z}^+\}$. Almost the same is assumed for the creation date, but instead it is all combinations of eight digit numbers, $\{n|0 \leq n \leq 10^8 - 1, n \in \mathbb{Z}^+\}$. The set time can be any three digit number, $\{n|0 \leq n \leq 10^3 - 1, n \in \mathbb{Z}^+\}$ and the Id is any five digit number $\{n|0 \leq n \leq 10^5 - 1, n \in \mathbb{Z}^+\}$. Since each feature can be either true or false and there are eight features in total, the maximum value is $2^8 - 1$, so $\{n|0 \leq n \leq 2^8 - 1, n \in \mathbb{Z}^+\}$. In this way, by proving that the largest key, that is, when n is as great as possible and the smallest key where n is as small as possible have the same key length in base 26, it can be claimed that all keys with these specifications are going to be of the same length in base 26. This is a good result not only for aesthetic purposes but also because there is now another pattern that keys are to follow in order to be valid. In order to quickly check whether the key is valid or invalid, this is one of the small checks that can be performed to reduce the time for a validation.

The key with the largest value would have the following value:

$$(999999999\ 99999999\ 999\ 255\ 99999)_{10} = (\text{NBFRV FEVRO CGGQU KZQCD})_{26}$$

and the key with the smallest value would have following value:

$$(100000000\ 00000000\ 000\ 000\ 00000)_{10} = (\text{BHXZE SSRTY VAQGX MERIM})_{26}$$

Therefore, as long as the n value is within its boundaries, the key will have a constant length.□

Further development The development of an information based key validation system requires consideration of the way the system can be optimized. In the SKGL API, it can be

[4] `http://skgl.codeplex.com/`. Last accessed 2014.02.02

seen that the data is stored in radix 10 and later converted into radix 26. Given this, by letting the maximum value where features are stored be $2^9 - 1$ instead of $2^8 - 1$, it can be seen that the key with the largest value will still have a constant length, that is 25 characters. Thus nine features can be used instead, because the number of digits of $2^9 - 1$ will be the same as the number of digits of $2^8 - 1$, that is, 3 digits.

Since radix 10 is used, the data optimization is made in such a way that the maximum value for a specific part of the information, for example the creation date, is as close to the largest value that can be stored in radix 10 with the same number of digits.

To clarify this a bit more, because the maximum value of the creation date will be $10^8 - 1 = 99999999$, it can be seen that the number of digits that has to be allocated is eight, and when this is compared to the maximum value that can be constructed in base 10 with eight digits, it is in fact also $10^8 - 1 = 99999999$, so it can be said that this piece of information is optimized.

On the other hand, when analysing the storage of features, where the maximum value is $2^8 - 1$, it can be seen that it is not entirely optimized. That is because the maximum value in base 10 that has three digits is $10^3 - 1 = 999$. Percentagewise, $2^8 - 1$ is only 26% of $10^3 - 1$, thus 74% of the value is not used at all. Even if nine features would be allowed, where the maximum value would be $2^9 - 1$, in contrast to the largest possible three digit number, $10^3 - 1$, it would constitute 51% only, which is roughly a half. Therefore, this is not an optimized way of storing that data, and it should be considered to choose base 2 instead because the initial data is stored in binary.

Asymmetric cryptography This option is in most cases more secure than symmetric cryptography, because it works on the principle of digital signing. The public key is stored inside the client application, which can be used to verify the signature[5] of the useful information. The private key that can generate these signatures is stored on a server or in the publisher's application.

There must be some applications that have implemented this idea. By searching through CodePlex, Activatar[6] can be found, which works on the idea of public key cryptography. It uses RSA for the signature mechanism.

If such a system is to be designed now, it would be better to use Elliptic key cryptography since it will reduce the output key size and still be quite secure.

Weaknesses

- The checksum is a function that generates a value `data2` based on `data1`, so the weaknesses of checksum based key validation should be considered.
- The way the information is arranged in the key and features like key length and type of information are rules the key should obey in order to be valid. In this way, if the key contains both a checksum function and uses the second step of encryption (when the checksum is encrypted together with the information), it is a good idea to review the weaknesses of pattern based key validation.

[5] `http://msdn.microsoft.com/en-us/library/hk8wx38z(v=vs.110).aspx`. Last accessed 2014.02.03
[6] `http://activatar.codeplex.com/`. Last accessed 2014.02.03

Conclusion

There are several conclusions we can draw from these different groups of key validation algorithms. First, the choice of a system depends on the information that is to be stored in the key. If there are only two license options, registered and unregistered, checksum based key validation or pattern based validation can be used. If, however, more information that has to be stored, information based key validation can be used instead. The second conclusion is that if a key validation system is to be set up at this point, online key validation should be considered also. That is because the validation method is not stored on the client's computer, and so cannot be found that easily. Online key validation also gives more control to the application developer since it makes it possible to block a key, or change any data associated with the key.

Bibliography

- http://msdn.microsoft.com/en-us/library/system.text.regularexpressions.regex(v=vs.110).aspx, Last accessed 2014.02.03
- http://skbl.clizware.net/help.html. Last accessed 2014.01.31
- http://stackoverflow.com/a/3007632/1275924. Last accessed 2014.01.31
- http://skgl.codeplex.com/. Last accessed 2014.02.02
- http://activatar.codeplex.com/. Last accessed 2014.02.03
- http://msdn.microsoft.com/en-us/library/hk8wx38z(v=vs.110).aspx. Last accessed 2014.02.03

Appendix A – Help functions

This is the code that has to be added to Example 2 in Pattern based key validation section. It can also be downloaded at `http://dev.artemlos.net/func/conf1.txt`.

```
static string base10ToBase26(string s)
{
    char[] allowedLetters = "ABCDEFGHIJKLMNOPQRSTUVWXYZ".ToCharArray();

    decimal num = Convert.ToDecimal(s);
    int reminder = 0;

    char[] result = new char[s.ToString().Length + 1];
    int j = 0;


    while ((num >= 26))
    {
        reminder = Convert.ToInt32(num % 26);
        result[j] = allowedLetters[reminder];
        num = (num - reminder) / 26;
        j += 1;
    }

    result[j] = allowedLetters[Convert.ToInt32(num)];

    string returnNum = "";

    for (int k = j; k >= 0; k -= 1)
    {
        returnNum += result[k];
    }
    return returnNum;

}
static string base26ToBase10(string s)
{
    string allowedLetters = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
    System.Numerics.BigInteger result = new System.Numerics.BigInteger();

    for (int i = 0; i <= s.Length - 1; i += 1)
    {
        BigInteger pow = powof(26, (s.Length - i - 1));
        result = result + allowedLetters.IndexOf(s.Substring(i, 1)) * pow;
    }
    return result.ToString();
}
static BigInteger powof(int x, int y)
{
    BigInteger newNum = 1;

    if (y == 0)
    {
        return 1;
    }
    else if (y == 1)
    {
        return x;
    }
    else
    {
```

```
57          for (int i = 0; i <= y - 1; i++)
58          {
59              newNum = newNum * x;
60          }
61          return newNum;
62      }
63  }
64  static int modulo(long _num, long _base)
65  {
66      return (int)(_num - _base * Convert.ToInt64(Math.Floor((decimal)_num / (
            decimal)_base)));
67  }
```