Simple Production Of Keys

Borja Gomez- kub0x@elhacker.net

April 2017

1 Introduction

There are many GPL applications for generating words nowadays. In this paper an enhanced application is presented for generating words in an intelligent way minimizing the overall cost of production. In addition techniques of generation will be discussed included the one used in this application.

1.1 Motivation

Since a long time I have been studying the crypto world and its applications in the real world and felt the need to built a personal tool for fast generation of sequences to create dictionary files for offline cracking. Our Spanish ethical hacking community elhacker.net eventually hosts a contest called Abril Negro (Black April) where discussion of methods related to security and its branches arise. That is the reason why I am developing this project.

1.2 What is S.P.O.K?

S.P.O.K stands for Simple Production Of Keys. It is an engine for generating words of fixed length for latter use in cryptographic cracking schemes like WPA/WPA2, NTLM, VPN records, Databases ....

1.3 What does make it different?

Traditional word generating applications uses conventional algorithms that are time consuming. A sequence of fixed length is generated, then increment length by 1 i.e: words of length 5, then length 6 .... S.P.O.K starts generating the sequence of maximum length and whilst, it is able to save subsequences of less length to a specified file. The final result is to generate all the words between a minimal length and a maximal length all at the same time.

2 Computational comparison between traditional applications

As been commented, S.P.O.K has an enhanced algorithm for generating words that saves time for producing these. But how can we measure this speedup?

2.1 Mathematical Overview

From a straightforward mathematical view we define the traditional application word generating scheme as following:

Let $S_n = \{a_1, \cdots, a_n\}$ be the charset of length n. We wish to generate all words among length i and j.

2.2 Iteration Speedup

Thus the total number of operations to generate all words will be

$$Ops = \Sigma_{x=i}^{j} n^x$$

S.P.O.K in the same context will need a total number of operations of:

$$Ops = n^j$$

Thus it will save a significant number of operations in the order of:

$$\Sigma_{x=i}^{j-1} n^x$$

2.3 Operation Saving

The S.P.O.K performed operations over traditional number of operations can be viewed as:

$$min = i, max = j \Rightarrow \varphi = \frac{n^j}{n^j + n^{j-1} + \cdots + n^i} = \frac{n^j}{n^i(n^{j-i} + n^{j-i-1} + \cdots + 1)} = \frac{n^{j-i}}{n^{j-i} + \cdots + 1}$$

2.4 Direct Impact

If $|S_n| = 16$ then $\varphi \approx 0.94$. This means that if the charset has length 16 (hexadecimal) S.P.O.K will save a 6% on operations compared to the standard method.

Note that this impact is positive since generating algorithm must log sequences into a file and alternatively hash these.

3 Application Development

Now is time to describe the programming techniques used for the development of this project. Is very recommended to think on the design before programming so you can choose all the tools for making it.

3.1 OS, IDE and Language

As a Linux user I decided to create this project using C++11 and Monodevelop IDE running on Fedora 25 x86_64.

3.2 Design

First I developed the recursive and iterative algorithm to check possible fails, and decided that iterative method is the one that fits here since recursive will throw a stack overflow when dealing with notable lengths. OOP has been deployed as it's a must, it allows me to ease the abstraction process.

Also decided to make this tool simple as a console project accepting multiple commands with arguments.

In addition the buffering logic has been ported to C in order to control all operations made in this context.

The use of threads also is a considerable option for obtaining more performance. Every time a buffer is full it's sent to a thread depending on buffering option selected by the end-user.

3.3 Generating Word Algorithm

Here is where magic happens:

For generating all words between i, j using charset $S_n = \{a_1, \cdots, a_n\}$ of length n, we will fill a vector with j elements all being a_1. Then permute v_j until its value is a_n, so reset v_j to a_1 and permute once previous element v_{j-1}. When v_{j-1} hits a_n reset to a_1 and permute once its left element until you hit $v_1 = a_n$. Program will stop when that condition is reached.

```
void Engine::Permute(){
        int seqcount = 0;
        Node *node = nodes.back();
        start = std::chrono::steady_clock::now();
        while(node != nullptr){
                if (node->getValuePos() < CHARSET.size()-1){
                        node->Permute(
                        CHARSET.at(node->getValuePos() + 1));
                        node = nodes.back();
                        if (verbose)
                        words++;

                        GenerateWords();
                }else{
                        //SAVE FUTURE ITERATIONS ON SUBSTR
                        if (!node->IsSignaled())
                                if (++seqcount == i)
                                        i++;
                        node->Reset();
                        node = node->Prev();
                }
        }
}
```

A remarkable point here is that seqcount variable is incremented every time a sub sequence of length k is completed. When its value reaches the minimum sub sequence to log, in this case i, i increments by 1. This is known as node signaling and avoids to loop extra rounds in sub sequence algorithm.

3.4 Dumping Words

The main problem that arise in the previous algorithm is to dump efficiently all words between a pair i, j (min,max). Since only words of length j are generated an efficient way to dump sub sequences of words from j to i is needed.

Also take into account that all sub sequences will be dumped so when condition $v_k > a_n$ is reached, sub sequence of length k must be avoided, therefore that restriction must be present via node signaling as seen in point 3.3, if not sub sequence would appear repeated in dictionary and that is not desirable.

Another concern while writing to file stream is the performance slow down depending on the hard drive of the target system. The end-user can use single buffer or multi-buffer mode options to cover its needed. Note that multi-buffering will consume more RAM and single buffer will have to wait when main buffer is full in case it has not been logged entirely to the dump file.

```
void Engine::SubStrCrypto(){
        int start = 0;
        for (int k = i; k <= j; k++){
        int p = 0;
        char word[j-start];
        for (int k = start; k < j; k++){
                word[p] = nodes.at(k)->getValue();//_str[k];
                p++;
        }
        Crypto::GetInstance()->HashWord(buffer, word, nhash, &bfpos);
        start++;
        }
}


void Engine::SubStrWord(){
        int start =0;
        int p = start;
        int offset = j - start;
        for (int h = i; h <= j; h++){
                for (int k = bfpos; k < bfpos + offset; k++){
                        buffer[k] = nodes.at(p)->getValue();
                        p++;
                }
                bfpos += offset;
                buffer[bfpos++] = '\n';
                start++;
                p=start;
                offset--;
        }
}
```

```
void Engine::GenerateWords(){
        //AVOID CHECKING HASH FLAG EVERY ITERATION. SPLIT INTO 2 FOR
        if (!nhash)
                SubStrWord();
        else
                SubStrCrypto();

        if (bfpos > BUFFSIZE-999){ //BUFF > 400 MB
                if (verbose)
                        ShowVerbose();
                //WAIT UNTIL BUFFER IS FREED IN THREAD.
                //AVOID DATA CORRUPTION
                std::thread t;
                if (mbuffer){
                        //WAIT SOME TIME TO NOT OVERLAP BUFFERS
                        std::this_thread::sleep_for(
                        std::chrono::milliseconds(mbuffer));
                        t = std::thread(&FileHandler::LogFileMB,
                        FileHandler::GetInstance(),
                        std::string(buffer), std::ref(savefile),
                        std::ref(saveparams), j);
                }else{
                //SINGLE BUFFER MODE. WAIT UNTIL REF BUFFER IS FREED
                        while (buff.size())
                        std::this_thread::sleep_for(
                        std::chrono::milliseconds(50));
                        buff = buffer;
                        t = std::thread(&FileHandler::LogFile,
                        FileHandler::GetInstance(), std::ref(buff),
                        std::ref(savefile), std::ref(saveparams),j);
                }
                memset(buffer,0,bfpos);
                t.detach();
                bfpos = 0;
        }
}
```

As seen in the code all sub sequences are logged into the buffer until it reaches a certain amount of data measured in MB, in this case BUFFSIZE=400MB. Then buffer is written into the dump file asynchronously. Note that async operation allows the execution to keep generating sequences while the other thread is storing words into the file. Checking hash parameter at the beginning avoids checks in a possible loop, thus splitting into two loops is needed.

3.5 Storage

Storage plays a relevant role as the main feature of the tool is to dump words into a dictionary for offline cracking purposes. Then analysis on this topic is needed.

Each word is measured on bytes as it is dumped on ASCII. Take into account that each line has an extra byte for newline byte 0x10.

Mathematical definition for storage can be viewed as follows:

$$\frac{(\Sigma_{x=i}^{j} n^x \cdot x) + \Sigma_{x=i}^{j} n^x}{1024^k} \forall k \in \mathbb{N}_0$$

Which is simplified into:

$$\frac{\Sigma_{x=i}^{j} n^x \cdot (x+1)}{1024^k} \forall k \in \mathbb{N}_0$$

where $k = 0$ turns denominator into bytes, 1 in KB, 2 in MB $\cdots$

First summation: for a charset of length n count all bytes of words of length i to j. All words of length i have i bytes, words of length $i+1$ have $i+1$ bytes and so on until reach words of length j.

Second summation: Counts the newline byte for each word from i to j

3.6 Hashing

Hashing words within S.P.O.K is yet another powerful point, since no external applications are needed for this purpose, thus saving time when dealing with offline cracking.

At this time MD5, SHA-1 and SHA-256 are supported, where you can choose which one to perform. API used for hashing is OpenSSL: md5.h and sha.h libraries.

The main drawback of hashing with these is the need to convert from byte to hex representation(unsigned char $->$ hex). For speed discussion head to Benchmark section 5.3.

3.7 Storage With Hashing

When hashing is enabled a method for calculating the final size of the dictionary is needed. Note that the end-user can choose which hash to perform, and can decide if run MD5+SHA1+SHA256, MD5+SHA1 or MD5 (via parameter -h num).

Mathematical definition is as follows:

$$(\Sigma_{x=i}^{j} n^x) \cdot (32 \cdot k_1 + 40 \cdot k_2 + 64 \cdot k_3 + num) \forall k_1, k_2, k_3 \in [0, 1]$$

where $k_1 = 1$ if $num > 0$, $k_2 = 1$ if $num > 1$ and $k_3 = 1$ if $num = 3$. Note that hash size has to be doubled since hex representation of a hash takes 2 hex characters per byte.

3.8 Parameter Parsing

As this application doesn't use a third-party library to parse CLI parameters an efficient algorithm is needed to avoid invalid input or commands that don't exist. The approach here is to create a command list with all commands that the application uses, then provide the CLI parameters in a vector and check if these parameters are inside of the list. If a CLI parameter is on command list then check if it needs an argument, if needed then expect it in next round. If a parameter is not in command list and a previous command wasn't expecting an argument then is considered illegal input. Also it takes into account command repetition and alias (i.e: -g or –generate), this is, when a parameter is inside command list it is checked if it's marked.

```
bool ParamHandler::ParseArguments
(const std::vector<std::string>& args, int* paramcount,
bool *verbose, bool *version, bool *mbuffer, std::string& dumpfile,
std::string& loadfile, std::string& savefile, std::string& charset,
 std::string& interval, std::string& hash, std::string& lastword){

        bool ret = true;
        std::vector<Param*> commands;

        [..]PARAM INSTANTATION. PUSH THESE INTO VECTOR[..]

        Param *curParam = nullptr, *c = nullptr;
        auto it = args.begin();

        while (it != args.end() && ret){
                c = IsInList(commands, *(it));
                if (c!= nullptr){
                        if (curParam == nullptr){
                                if (c->HasArg())
                        //Expect command argument in next iteration
                                        curParam = c;
                                if (!c->IsMarked()){
                                        c->SetMarked();
                                        (*paramcount)++;
                                }else{
                                        //Command repetition
                                        ret = false;
                                }
                        }else{
                                //Expected command argument
                                ret = false;
                        }
                }else{
                        if (curParam != nullptr){
                                curParam->SetArgument(*it);
                                curParam = nullptr;
                        }else{
                                ret = false; //Command not in list
                        }
                }
                it++;
        }
        if (curParam) ret = false;
        [...]UPDATE VALUE OF PASSED REFERENCES[...]
        [...]DELETE INSTANTIATED OBJECTS[...]
        return ret;
}
```

3.9 Saving Session State

Keeping in mind that applications have to be closed suddenly a save file for session states has been implemented. This allows the end-user to resume a previous process of word generation thus avoiding to start from the beginning.

All parameters specified via CLI are logged included default ones if end-user didn't setup them. The most important part of saving current state is the time where the saving happens. State is saved when word buffer is full at capacity and after writing buffer to file. So if the end-user suddenly closes the tool, the last word written to file is taken as a reference of last program state.

```
[...]
if (!savefile.empty()){
        if (verbose)
                saveparams = "-v ";
        saveparams += "-g " + dumpfile + " -c " + CHARSET + " -i "
         + interval + " -h " + std::to_string(nhash) + " -w ";
}
[...]
if (bfpos > BUFFSIZE-900){ //BUFF > 400 MB
[...]
```

These two code fragments are from separated member functions. The first one is located at parsing algorithm hence its job is to determine if save argument is present, if true then composes the save parameters needed for saving session state in future execution.

Second one is located at word generating algorithm (do not confuse with permute) therefore its job is to compose sequences of variable length depending on the input interval. When the buffer is full, then the last composed word of maximum length is saved.

3.10 Loading Session State

Node signaling has to be taken into account when loading the save file since if the end-user was logging all words between i, j probably some sub sequences had already been entirely logged thus if node signaling is not used then these will appear repeated.

This is achieved by comparing each character of the last word with the initial word of the charset. If current character of last word is greater than initial word then enable signaling in next node: $lw_k > a_1, sign = true$

```
void Engine::FillNodeList(){
        bool signaled = false;
        char value = CHARSET[0];
        int valuepos = 0;
        Node *node = nullptr, *prev = nullptr;

        for (int k=0; k < j; k++){
                if (lastword.size()){
                        value = lastword[k];
                        valuepos = GetLetterPos(value);
                }
                node = new Node(value, valuepos, signaled, prev);
                nodes.push_back(node);
                prev = node;
                if (value != CHARSET[0])
                signaled = true;
        }

        if (!lastword.size()) //AVOIDS REPEATING LAST WORD
                GenerateWords(); //WRITE FIRST WORDS
        }
}
```

lastword.size() is used to detect if a loading process happened. If not, vector is filled by default like in section 3.3. But if true then the character value of each letter of lastword is used along with its position value in charset. Node signaling is enabled here to detect if sub sequence had been already logged in past execution.

Loading save parameters is quite straightforward, just load the save state file line in order to retrieve CLI parameters that were saved previously and pass these to the parsing algorithm in point 3.8.

4 Usage

Here the application usage will be discussed. S.P.O.K counts with a list of debug options and commands that accept multiple arguments.

4.1 Options

Debug and tool options:

-ver, –version: Show current version of the tool. At this moment v1.0 April 2017.

-v, –verbose: Show all the information related to elapsed time, words per second, current storage of file over total needed, writing speed and hashing per second (if -h specified as argument).

4.2 Commands

Tool's main commands. One of these must be present:

-g, –generate [FILE] [ARGUMENTS]: Generate a dictionary in FILE using ARGUMENTS. If no argument is specified defaults are used. See below for default arguments.

-l, –load [SAVEFILE]: Load previous saved session state contained in SAVEFILE.

4.3 Arguments

Arguments used on main commands. If omitted defaults are used.

-c, –charset [CHARSET]: Use specified CHARSET for word generation, default is: abcdefghijklmnopqrstuwvxyz.

-i, –interval $< i, j >$: Generate all words of length min & max, default is: $< 1, 5 >$.

-h, –hash $< number >$: hash output words into FILE using $< number >$ algorithms. $< number >$ ranges from (1,3). Available algorithms are: MD5, SHA-1 and SHA-256. Enabled only via argument.

-s, –save [SAVEFILE]: save current session state in SAVEFILE. Enabled only via argument.

-m, –multibuffer $< time >$: waits $< time >$ milliseconds before passing a copy of main buffer to thread. Experimental and should be adjusted.

4.4 Command Line Example

spok -v -g words.txt -s save.sav -i 4,8 -c '01234567890ABCDEF' -h 3 -m 250

Generate all words of length 4 and 8 using hexadecimal charset, displaying debug, hashing output with MD5, SHA-1, SHA-256, saving current session state and using multi-buffer option waiting 250ms before passing buffer's copy to thread.

5 Benchmark

Now is time to discuss the performance of S.P.O.K. For this verbose mode is needed to display words per second and minute as well as words stored per minute.

5.1 Target System

Tested on my desktop computer equipped with Motherboard: Asrock 970 Pro R2.0, CPU: AMD FX-9370 Eight-Core @4.20GHz, RAM: 8 GB DDR3 and SSD: 256GB TS256GSSD370.

5.2 Execution benchmark

As an example using hexadecimal charset for generating and dumping all words of length 8:

```
./spok -v -g 1.txt -c 0123456789ABCDEF -i 8,8

Total storage needed: 36864 MB
Words per minute: 2050543747
Words per second: 34175729
Current storage: 17101.3/36864 MB
```

As commented in section 3.4 words are logged asynchronously thus current storage results can deceive end-user since last buffer can be written at the time the verbose is displayed.

Now lets log all words of length 1 to 8. Remember that the enhanced algorithm only generates all words of length 8 and construct sub sequences of these (with hex charset 6% of operations are saved):

```
./spok -v -g 1.txt -c 0123456789ABCDEF -i 1,8

Total storage needed: 39030.3 MB
Words per minute: 1910733947
Words per second: 31845565
Current storage: 17495.1/39030.3 MB
```

Notice how words per minute have decreased. This is due to the need of logging all sub sequences less than 8. But as soon as a decent amount of words with length less than 8 are completely logged, words per minute will increase until hit the result of the first test case execution.

Enabling multi-buffer option with 250ms, words per minute are 2097147014. Extra 46 million words are generated comparing to single buffer option.

5.3 Hashing Benchmark

Now is time to measure the performance of hashing algorithms in S.P.O.K. As said before, end-user can combine multiple hash primitives at a total of 3. These are MD5, SHA-1 and SHA-256. In this benchmark three executions are measured using one, two and three algorithms at the same time. Only words per minute will be analyzed, not hashes per minute. Note that buffer size is 400MB in this example.

Using MD5, hex charset and logging words from 6 to 9:

./spok -v -g 1.txt -i 6,9 -c 0123456789ABCDEF -h 1

Hashing with 1 algorithm(s)
Total storage needed: 2.30683e+06 MB
Words per minute: 11075138
Words per second: 184585
Current storage: 1200/2.30683e+06 MB

For each word it performs 1 hash, and since words from 6 to 9 are logged, 4 hashes are computed in each iteration. The target file has 2TB when finished. At a ratio of 11 million words and 1.2GB per minute.

Using MD5+SHA-1, hex charset and logging words of lenght 9:

./spok -v -g 1.txt -i 9,9 -c 0123456789ABCDEF -h 2

Hashing with 2 algorithm(s)
Total storage needed: 4.84966e+06 MB
Words per minute: 22671859
Words per second: 377864
Current storage: 1200/4.84966e+06 MB

In this case only words of length 9 are generated, thus 2 hashes per iteration (MD5 and SHA-1). Performance compared to the first example has improved, this is since due to the length of generation. The target file has more than 4TB when finished. At a ratio of 22 million words and 1.2GB per minute.

Using MD5+SHA-1+SHA-256, hex charset and logging words of lenght 9:

./spok -v -g 1.txt -i 9,9 -c 0123456789ABCDEF -h 3

Hashing with 3 algorithm(s)
Total storage needed: 9.1095e+06 MB
Words per minute: 13091820
Words per second: 218197
Current storage: 1600/9.1095e+06 MB

As above, only words of length 9 are generated, thus 3 hashes per iteration (MD5, SHA-1 and SHA-256). The target file has 9 TB when finished. At a ratio of 13 million words and 1.6 GB per minute.

5.4 Execution Time

The formula for calculating how much time is needed (in hours) is:

$$\frac{n^j}{60 \cdot W_m}$$

where n is charset length, j maximum length and W_m words per minute.

As in the first example (Single buffer mode):

$$\frac{16^8}{60 \cdot 2050543747} = 0.0349$$

thus in 2 minutes and 5 seconds all hex words of length 8 will be generated and logged into the dictionary.

In the second example taking average of $W_m = 2 \cdot 10^9$:

$$\frac{16^8}{60 \cdot 550 \cdot 10^6} = 0.13$$

therefore it will take 2 minutes and 8 seconds to log all words from length 1 to 8. It is a positive result since only takes 3 seconds more to generate all words of less length.

In case of my university they use a specific charset based on [a-z,A-Z,0-9] with total length 62. Student delivered passwords have length 8. Thus $n = 62, j = 8$

The naive workaround is to use this formula to calculate how much time is needed for enumerating the whole key space:

$$\frac{62^8}{60 \cdot 2050543747} = 1774.6521$$

Thus $\approx$ 74 days are needed. Does this seem huge, doesn't it? See Further Ideas below, because better approach exist for specific charset cases. Since general algorithm works with permutations with repetition (n^k) but specific ones are based on permutations of multisets.

Summarizing, naive approach enumerates all words but if we know the specific charset we can breakdown to permutation of multisets which can save a lot of time by reducing the key space.

6 Further Ideas

6.1 Computational Improvement

Parallel speedup can be implemented specifying the number of CPU cores that will perform the task.

6.2 Cracking Engine

The generated words can be used in an offline brute force attack (no dictionary) on the fly in either CPU or GPU. As far as I know you can pipeline a generating word tool and a cracking tool thus I don't think it will yield an advance in the speed, but I don't like to discard possibilities.

6.3 Word Generation Based In Sub intervals

From the logarithmic perspective the whole interval of iterations can be partitioned into sub intervals. So when generation of words start, interval is partitioned into 2^k sub intervals where each has $\frac{n^j}{2^k}$ words. Then fix a variable x and generate x words for each sub interval.

i.e: $n = 26, j = 8, k = 17$ therefore 131072 sub intervals will be needed each having a total of 1593224 possible words. Working on parallel as point 6.1 will considerably speedup the process and improve the ratio of succeed.This is one of the ideas I might develop in a future.

6.4 Permutation of multisets

As seen above the algorithm works with permutations with repetition, these are n^k operations. But this can be a naive method to use when the specific charset is known, i.e: Words of length 8 composed of 5 numbers and 3 words.

From $36^8 = 2.8211 \cdot 10^{12}$ operations to $\frac{8!}{5!3!} \cdot 10^5 \cdot 26^3 = 98425600000$ operations, thus permutation of multisets will do 91% less of operations.

In the example, permutations with repetition (naive algorithm) will last $\approx$ 22 hours and permutations of multisets just $\approx$ 47 minutes. As we can see a HUGE reduction.

As said in section 5.4 the time needed to enumerate all the passwords of my university is 250 days. If we reduce the key space to

$$\frac{8!}{4! \cdot 2! \cdot 2!} \cdot 10^2 \cdot 26^6 = 1.29744 \cdot 10^{13}$$

Then 4.3 days are needed, thus being a reduction of $\approx 94\%$ of total time.

The perfect mode of operation of this approach would be combining generation based in sub intervals and p2p.

6.5 Starting Word

By default the starting word for the main algorithm is composed of j equal characters that reference the first character of the charset this is $w_1 = \{a_1, \cdots, a_1\}$ and $|w_1| = j$. A powerful technique would be to allow the end-user to choose which word is the starting word based on the input charset.

This is a good approach to sieve the key space. Note that this technique is implemented when loading a previous session state (lastword implementation). As this tool is open source, I do not discard this change on a future.

6.6 Variable Length Generation

Another powerful technique would be variable length word based generation. At this moment only words between lengths i,j are allowed. Thing is to let the user choose which words to generate of variable length. For example, generate all words of length $1, 2, 4, 7, 9$. As seen above, algorithm works only with length 9 composing sub sequences of less length, thus with node signaling lengths not included in the list are avoided.

6.7 Avoiding Character Repetition

As discussed above, sometimes the specific charset is known. In those cases is desirable to not perform permutations with repetition and choose a better performance approach like permutations of multisets, however depending on the context, end-user doesn't need to include repetition.

It's resource waste if the context is based on a generation without repetitions, since main algorithm would be generating words with character repetition. Hence there is a significant need to include these techniques of single permutations without repetitions to cover these singular cases.

6.8 Rainbow Tables

One drawback of dictionary attacks is the involved amount storage needed. This drawback can be eliminated implementing Rainbow Tables with reduction functions at time-space trade-off. Note that salted hashes will yield these useless.

6.9 p2p Decentralized Network

p2p applications are well known for their scheme as they are used for end-users to share multiple content. The cost of maintaining a cluster of computers for storage would be huge for an application of this purpose, therefore end-users can take part generating dictionaries and storing them.

The approach here would be to construct a decentralized network consisting in a Rendezvous that connects each client in the bootstrap process. Every time a user wants to crack a hash it will get redirected to the p2p network for searching the word behind the hash. Techniques for p2p deployment can be discussed with me via mail or elhacker.net.