User Guide Of The Uncrackable Encryption Toolchain

Table of Contents

Features

- Cross-platform. Runs on Windows and Unix.
- Uses one-time pad for encryption. Unless the key is leaked, it is impossible to crack the encrypted data.
- Have basic support of AES encryption algorithm.
- Can be used to enhance the functionality of other encryption softwares.
- Support of regex parameter for file procession.
- Powerful data piping system via command line interface.
- The source code is released under BSD license. As long as you reproduce the license in the redistribution of this software, you are allowed to do anything with the source code, including selling this software or using it for commercial purpose.
- Will probably become the most sophisticated available in the Internet by 6th July, which is the public release date of the software.

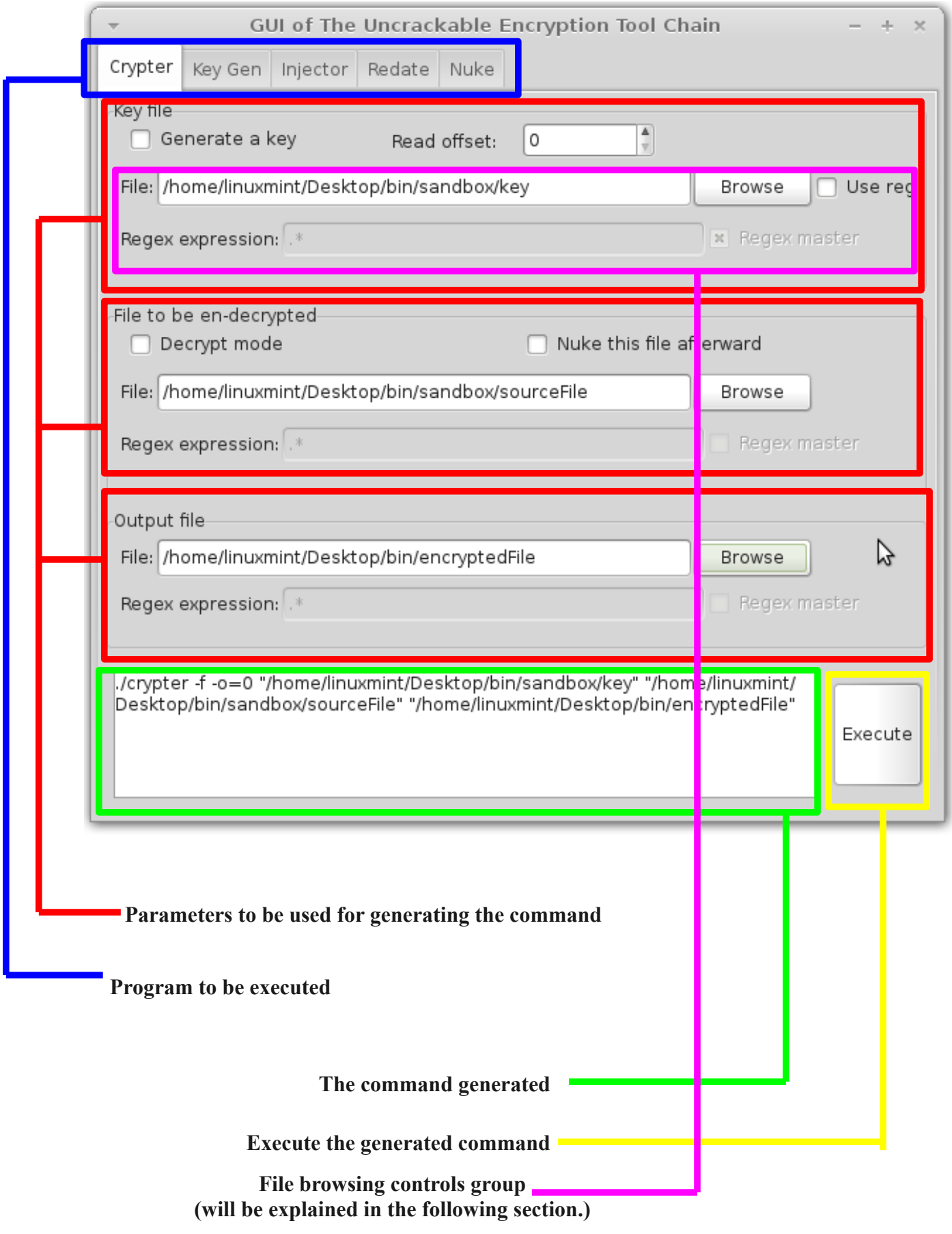
Overview

The Uncrackable Encryption Toolchain is a set of programs that is designed for encrypting and obfuscating sensitive information.
The toolchain includes:

- dummy – copies, redirects and encodes data to be interpreted by the toolchain
- keyGen – generates files with random content inside, which can be used as a key
- crypter – en-/decrypts files, using either one-time pad or AES algorithm
- injector – hides the content of a file into another file.
- redate – change the timestamps of files.
- nuke – overwrite the content and the filename of the files to prevent it from being revealed by data recovery softwares.
- GUI – a graphic user interface that executes the programs above.

Except GUI, all programs inside this toolchain backed by Piper, a command line interface framework, which was written as a proof of concept.

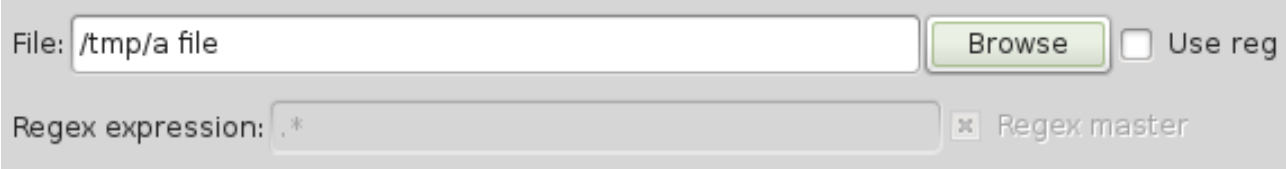
GUI


GUI of The Uncrackable Encryption Tool Chain
Crypter
Key Gen
Injector
Redate
Nuke
Key file
Generate a key
Read offset:
0
File: /home/linuxmint/Desktop/bin/sandbox/key
Browse
Use reg
Regex expression: .*
Regex master
File to be en-decrypted
Decrypt mode
Nuke this file afterward
File: /home/linuxmint/Desktop/bin/sandbox/sourceFile
Browse
Regex expression: .*
Regex master
Output file
File: /home/linuxmint/Desktop/bin/encryptedFile
Browse
Regex expression: .*
Regex master
./crypter -f -o=0 "/home/linuxmint/Desktop/bin/sandbox/key" "/home/linuxmint/Desktop/bin/sandbox/sourceFile" "/home/linuxmint/Desktop/bin/encryptedFile"
Execute
Parameters to be used for generating the command
Program to be executed
The command generated
Execute the generated command
File browsing controls group
(will be explained in the following section.)

File Browsing Controls Group

There are two modes of processing files. Single file mode and regex mode.
If the check box **Use regex**(or **Use reg** in some version of GTK) is checked, files will be processed under regex mode. Otherwise, they will be processed under single file mode.

Single File Mode



This file processing mode is pretty straight forward. What you have to do is clicking the button **Browse** and select a file that you desire. However, only one file can be processed at a time.

Regex Mode

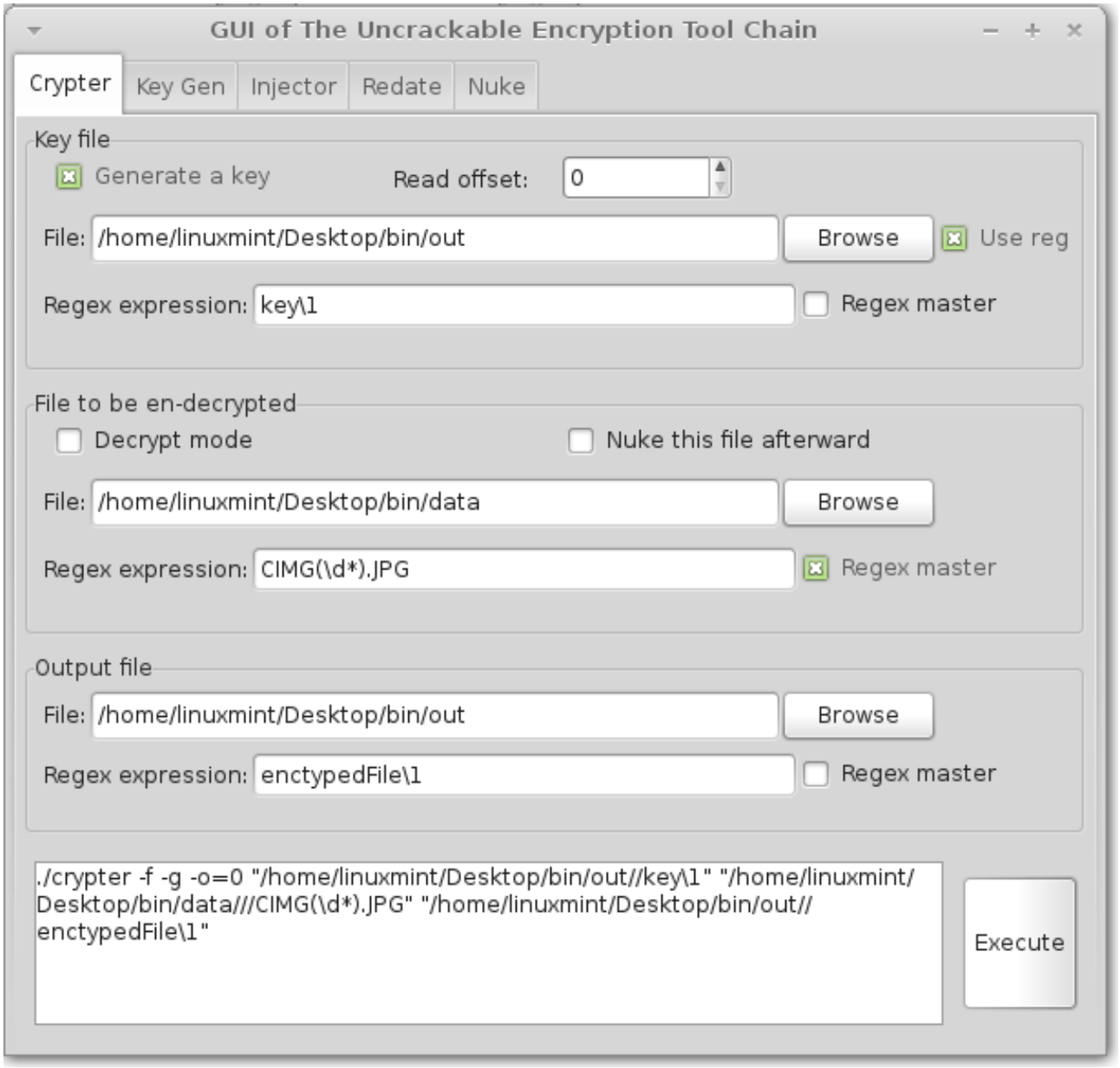
This file processing mode is powerful, yet complicated. Under this mode, multiple files can be processed with single command.

Example:
Assume that you have these files in the directory /home/linuxmint/Desktop/bin/ :

```
CIMG5321.JPG
CIMG4896.JPG
CIMG789.JPG
KIMG5014.JPG
KIMG8970.JPG
KIMG12345678.JPG
WIMG3.JPG
```

Says, you want to encrypt the files CIMG5321.JPG, CIMG4896.JPG and CIMG789.JPG , with automatically generated keys that named key<NUMBERS> and the encrypted files named encryptedFile<NUMBERS> to be written in /home/linuxmint/Desktop/bin/out/

The following setup will achieve what you want:



Explanation:
The checkbox **Use regex** is checked. It means that regex file processing mode is enabled. This will enable you to type the **Regex expression** in text controls.

Under regex mode, the **Browse** button is used for setting the base directory of the file.

One of the file expression have to be **Regex master**. Regex master is a parameter which search for files recursively with "Regex expression" under the base directory(which is labeled as **File:** in the GUI), and iterated the matched file names for procession.
By checking "Regex master", the parameter will be set as a regex master. This will uncheck **Regex master** of other parameters and implicitly set them as a regex slave.

Regex slave is a parameter which have a **Regex expression** for replacement. The expression is to be replaced by the file name that is iterated in **Regex expression** of **Regex master**.

In this example, there are only three files to be processed. The work can be done easily by choosing the file one by one. However, if there are tons of files with similar filename to be processed, using

regex mode is much more efficient than processing the files one by one.

Piper – The Command Line Interface Program Framework

Before introducing the programs, you need to know that all programs in this toolchain are based on Piper, which was a framework written as a proof of concept. There is some difference in the way of handling commands in between Piper and conventional command line programs.

Difference in Parameter HandlIng

Assume that there is a conventional program accept parameter in this way:

```
./program SRC DEST
```

This program would accept exactly two parameters. If you provide more than two, then there would be an syntax error.
However, it's different for Piper programs. Says, there is a piper program like this:

```
./piperProgram SRC DEST
```

This program would accepts number of parameter that's divisible by 2. For example, if four parameter is provided in this way:

```
./piperProgram src1 dest1 src2 dest2
```

Then this command would be equivalent to these commands:

```
./piperProgram src1 dest1
./piperProgram src2 dest2
```

“src1 dest1” are said to be a **block**, while “src2 dest2” are said to be another **block**.
To conclude, Piper programs accept parameter in this way:

```
./piperProgram [SRC DEST]...
```

Difference in Option Handling

In conventional programs these commands are equivalent:

```
./program -flag SRC DEST
./program SRC DEST -flag
```

It is because -flag applies to both SRC and DEST in both cases.
However, these commands are different for Piper programs:

```
./ piperProgram -flag SRC DEST
./ piperProgram SRC DEST -flag
```

In the former command, -flag applies to both SRC and DEST. However, the -flag in the second command have no effect.
This is because flags in Piper programs are *forward applying*. In this way, you can apply different flags for each parameter. You can do something cool like these:

```
./ piperProgram src -flag dest
./ piperProgram src1 dest1 -flag src2 dest2
```

Advance

Piper programs also have some powerful(but complicated) function*s*. They are documented in Advance Functions of Piper in this user guide.

Programs in This Toolchain

All the examples below are written for Unix. If you are using Windows, substitute ./program with program.exe

dummy

```
Usage: ./dummy  src   dest
Type:           in    out
Available special arguments: redir opt
Argument type description:
    in    input. read from either file(s) or stdin and load the it into buffer. can be used to redirect the *unprocessed* data to stdout.
    out   output. process the buffer in input, then write it to stdout or file(s), or both.
    redir     redirect. redirect the data from stdin to stdout. Triggered by '[r]'
    opt   option. Triggered by '-*', where * is an arbitrary number of characters.
Available options:
    -h    show this help message
    -f    force overwrite/nuke
    -F    prevent overwrite/nuke
```

This program is the simplest program in the toolchain. It make a copy of data from **src** to **dest**. This program can be useful for redirecting data from files to external programs, or vice versa.
Example 1:

```
./dummy first second
Effect: copy the file first, then paste it to second.
```

:
Example 2 [Read Advance Functions of Piper first.]:

```
./dummy "./src///(\w+_\w*[.](?=png|bmp|jpg|gif).*)" "./dest//\1"
Effect: copy the file in ./src/ that have the filename matches the regex \w+_\w*[.](?=png|bmp|jpg|gif).* to ./dest
```

Example 3 [Unix only, Read Advance Functions of Piper first.]:

```
nc -l 1234 | ./dummy "<[i]>" "./received//\1"
Effect: act as a server. Listen to the port 1234. Then write the received files to the directory ./received
./dummy "<[./directory///(.*)]>" /dev/null | nc 123.45.67.89 1234
Effect: send files inside the directory ./directory to 123.45.67.89:1234 .
```

keyGen

```
Usage: ./keyGen  size  outKey
Type:            s     out
Available special arguments: redir opt
Argument type description:
    s     size: size in bytes. K=1024, M=1024*1024, G=1024*1024*1024, KB=1000, MB=1000*1000, GB=1000*1000*1000
    out   output. process the buffer in input, then write it to stdout or file(s), or both.
```

```
    redir      redirect. redirect the data from stdin to stdout. Triggered by '[r]'
    opt  option. Triggered by '-*', where * is an arbitrary number of characters.
Available options:
    -h   show this help message
    -f   force overwrite/nuke
    -F   prevent overwrite/nuke
```

This program creates file(s) of specified size with random content, which can be used as a key. In Unix, this program uses /dev/urandom as a source of random data. In Windows, it uses CryptGenRandom() .

Key files can also be generated with the program **crypter** automatically, without using this program.
Example 1:

```
./keyGen 5M key
Effect: generates a 5MiB file named key with random content.
```

Example 2[Unix only, Read Advance Functions of Piper first]:

```
./keyGen 100M "(o)" | ent
Effect: pipe 100MiB random data to the external program ent , which tells the entropy of the key.
```

crypter

```
Usage: ./crypter  key file  en-/decryptedFile
Type:          in   in   out
Available special arguments: redir opt
Argument type description:
    in   input. read from either file(s) or stdin and load the it into buffer. can be used to redirect
the *unprocessed* data to stdout.
    in   input. read from either file(s) or stdin and load the it into buffer. can be used to redirect
the *unprocessed* data to stdout.
    out  output. process the buffer in input, then write it to stdout or file(s), or both.
    redir      redirect. redirect the data from stdin to stdout. Triggered by '[r]'
    opt  option. Triggered by '-*', where * is an arbitrary number of characters.
Available options:
    -h   show this help message
    -f   force overwrite/nuke
    -F   prevent overwrite/nuke
    -n   nuke the source file after the operation is finished.
    -a   encrypt the file with AES instead of one time pad. At most 128 bytes in the key is passed
into PBKDF2(SHA-256) to generate a hashkey. The hashkey is then used to encrypt the file by
using AES-256/CBC algorithm.
    -d   decrypt the file with the key provided.
    -g   enable key generation mode.
    -n   nuke mode. nuke the source file after the operation is finished.
    -^o=\d+ Set the read offset of the key, in bytes.[default=0]

Pipe note: only the data read in src is redirected to pipe
```

As the name of this program suggests, this program encrypts files.
Example 1:

```
./crypter -ng key data cipherText
./crypter -nd key cipherText data
```

Effects: The first command generates a **key** to encrypt **data** , then write the encrypted data to **ciperText**. Finally, it nukes **data**.
The second command decrypts **ciphterText** with **key** to get back the original **data**. Then nuke the **cipherText**.

Example 2:

```
./crypter -ng -o=10240 key.jpg data cipherText
./crypter -nd -o=10240 key.jpg cipherText data
```

Effects: Based on example 1. The difference is that it uses a .jpg file, skipping 10240 bytes, as a key. The read offset is used to get rid of the jpeg header because that the header of JPEG is likely to be predictable.

Example 3 [Unix only, Read Advance Functions of Piper first]:

```
echo "Calling from 16548: Our reinforcement with an atomic bomb is coming to Antarctica in an hour. " | ./crypter -o=10240 key.jpg "(i)" "(o)" | base64 -w0
echo -n "p8bS09HX0ZPa5+XkmKqwt7e8v6bW/fuqBPj9A/wGCvz/DwgSGcYeER0S0hQi1RcrJyYjJeMmNDMp6DI98jZDQj9FP/lOUQMlU1pIWkxeW1ZVFV9lGFpoH0JvgXUvIAw=" | base64 -d | ./crypter -o=10240 -d key.jpg "(i)" "(o)"
```

Effects: Based on example 2. The difference is that it encrypts a message with the cipherText, and write it to stdout in base64 format.
Assume that the cipher-text is “p8bS09HX0ZPa5+XkmKqwt7e8v6bW/fuqBPj9A/wGCvz/DwgSGcYeER0S0hQi1RcrJyYjJeMmNDMp6DI98jZDQj9FP/lOUQMlU1pIWkxeW1ZVFV9lGFpoH0JvgXUvIAw=” , then the second command can be used to get back the original text.

Example 4 [Read Advance Functions of Piper first]:

```
echo "password" | ./crypter -fan "(i)" data cipherText
echo "password" | ./crypter -fand "(i)" cipherText data
```

Effects: Based on example 1. The difference is that it uses a password and it uses AES encryption mode.

*Please notice that the flag -f is required because the program will ask you whether nuke the file or not. However, stdin is blocked because it is used as a source of password. By default, the program would not nuke the file and terminate the operation. Therefore, you have to pass -f flag to force it to nuke the file.

Technical Details: Encryption algorithms

This program supports two encryption algorithms. One-time pad and AES.

One-time Pad

This program uses one-time pad by default. This encryption algorithm proven to be impossible to

crack. It is used in many espionage activities. Here is how does it work:

Let's ask you a question:
(p + k) MOD 256 = 20
In the equation above, can you tell what is the value of p and k?
Definitely not! You must know either p or k to solve the equation.

Here is how do we use one-time pad to encrypt a byte of data
(p + k) MOD 256 = o, where p is the original text(plain text), k is a random value(key) and o is the cipher-byte(output)
Even you know the value of the cipher-byte, it's not possible to get the value of p unless you know the value of k.

What if you want to encrypt a file?
It's easy. Just encrypt each byte with the method above. So if the file have 3 bytes, then:
$(p_1 + k_1) \text{ MOD } 256 = o_1$
$(p_2 + k_2) \text{ MOD } 256 = o_2$
$(p_3 + k_3) \text{ MOD } 256 = o_3$
, where xn is the nth byte of x
Voilà, we have got an impossible-to-crack cipher-file.

In order to prevent the original data from being revealed, it is necessary to keep the key secret. It is also necessary to ensure that you use the key once only. The whole point of one-time pad is that the key is used one-time only. If you reuse it, it is no longer impossible to crack the data. **It is your responsibility to ensure that you don't reuse the key and keep it secret. If you reuse the key or released the key accidentally, and have the original data leaked, it is your fault.**

AES

This program also have basic support of AES encryption. AES encryption mode can be enabled with -a flag. Under this mode, the program takes as mode 128 bytes of data as a **pre-key**. Then, it hash the **pre-key** with PBKDF2(SHA-256), with 31415 iterations to generate a **key**. The **key** is then used to encrypt the file with AES-256/CBC algorithm.

This mode is inaccessible via GUI.

injector

```
Usage: ./injector  src  dest
Type:              in   out
Available special arguments: redir opt
Argument type description:
    in   input. read from either file(s) or stdin and load the it into buffer. can be used to redirect the *unprocessed* data to stdout.
    out  output. process the buffer in input, then write it to stdout or file(s), or both.
    redir      redirect. redirect the data from stdin to stdout. Triggered by '[r]'
    opt  option. Triggered by '-*', where * is an arbitrary number of characters.
Available options:
    -h   show this help message
    -f   force overwrite/nuke
    -F   prevent overwrite/nuke
```

-n nuke the source file after the operation is finished.
-d discard mode. can only be used with extraction mode. If enabled, it discards the content that is previously injected. CAUTION: a) This does NOT nuke the discarded content b) Using this flag with a non-injected file may corrupt the file.
-e extraction mode. extract the file src to dest.

Pipe note: Since this program uses append file opening mode, redirection of () and [] of dest are not allowed

This program inject the data of **src** to **dest** to hide the data of **src**.

Rationale

This program hide the data in this way:
Says, **src** contains “secret message” and **dest** contains “lame data”. Then, **dest** would become “secret messagelame data[9]”, where [9] is a 64bit field that contains the value 9, representing the length of the data injected, after the operation.
In the process of extraction, the last 8 bytes of the data from sec is read, which contains the length of the injected file. Then the injected file is extracted according to the length read. If -d flag is enabled, the data extracted inside **src**(which was **dest** during the injection) during the extraction would be discarded.

It is not easy to notice that data is injected in a file. It is because many file interpreters(like some image viewer, the executor of executable files, etc.) ignores extra trailing bytes.

Do not modify the file with data injected. If you do so, it is likely that the injected data will be lost as those file modifiers ignore the trailing bytes.

This program works best with an encrypter, either crypter in this toolchain or an external encryption software will work. It is a good tool to hide encrypted files.

Example 1:

./inject -fn alreadyEncryptedFile randomCatPhoto.jpg
./inject -ed randomCatPhoto.jpg alreadyEncryptedFile
Effects: The first command will inject alreadyEncryptedFile into randomCatPhoto.jpg . Then, nuke alreadyEncryptedFile. The second command is used to extract the previously injected data from randomCatPhoto.jpg to alreadyEncryptedFile. Then, discard the content extracted in randomCatPhoto.jpg

Example 2 [Read Advance Functions of Piper first]:

echo "not-so-secret message" | ./inject "(i)" whatever.dll
./inject -ed whatever.dll "(o)"
Effects: The first command injects the message “no-so-secret message” into whatever.dll .
The second command extracts the message from whatever.dll, and prints “not-so-secret message”.
*It is not recommended to inject unencrypted message into other files.

redate

This program changes the timestamps of files to current time. This program is similar to touch in unix. The difference is that this program change the timestamps by temporary changing the system

time. Please close all other applications before launching this program. Otherwise, files created/modified/accessed with other applications during the operation will have wrong timestamps.

Usage: ./redate file
Type: Fn
Available special arguments: redir opt
Argument type description:
Fn File name argument. A argument that only accept file, {file}, {i} and {ir} as a parameter.
redir redirect. redirect the data from stdin to stdout. Triggered by '[r]'
opt option. Triggered by '-*', where * is an arbitrary number of characters.
Available options:
-h show this help message
-^[mc]*a[mc]*=(\d\d\d\d)?(\d\d){4}([.](\d\d)([.](\d\d\d){1,3})?)?$ Change the timestamp of access time of the file to specified time to instead of using current time. Format: [YYYY]MMDDhhmm[.ss[.mmm[uuu[nnn]]]] If the optionals are not provided, it will be substituted by the current system time.
-^[ac]*m[ac]*=(\d\d\d\d)?(\d\d){4}([.](\d\d)([.](\d\d\d){1,3})?)?$ Change the timestamp of modify time of the file to specified time to instead of using current time. Format: ditto
-^[am]*c[am]*=(\d\d\d\d)?(\d\d){4}([.](\d\d)([.](\d\d\d){1,3})?)?$ Change the timestamp of changed time of the file of the specified time instead of using current time(Unix only). Due to technical issue, this option is unimplemented in Windows. Format: ditto

In most cases: In linux, mtime<=ctime, atime<=ctime, and mtime<=atime, where <= means is earlier than. In Windows mtime<=atime . Ensure that the timestamp that you provide satisfies the requirements above. This is necessary to prevent others from knowing that you have modified the timestamps of the files. Please notice that this program does NOT perform these checks automatically.

This program change the timestamp of files. This is achieved by changing the system time temporary. In unix, this program allows the users to set the data and time up to the precision of nanoseconds, which cannot be done with most other programs like **touch**.

Example 1:

./redate -amc=202010301527.32.666666666 fileFromTheFuture
Effect: changes the atime, mtime and ctime of the file **fileFromTheFuture** to 30/10/2020 15:27:32.666666666

nuke

Usage: ./nuke file
Type: Dout
Available special arguments: redir opt
Argument type description:
Dout Descturtive output. Writes to file. only {i} and file are allowed.
redir redirect. redirect the data from stdin to stdout. Triggered by '[r]'
opt option. Triggered by '-*', where * is an arbitrary number of characters.
Available options:
-h show this help message

```
    -f    force overwrite/nuke
    -F    prevent overwrite/nuke
    -^n=\d+ Overwrite the file, including the file name \d+ times. (default=3)

Please notice that this utility would not work in case the file content is journaled by the filesystem.
```

This program nukes the content and the name of files by overwriting the file with random data, preventing the data and the filename from being recovered.

Please notice that this program does not work in filesystems that journals the content of files. Most filesystem does not do so by default through.

Example 1:

```
./nuke secretFile
Effect: nukes secretFile. It overwrites its content and filename of secretFile thrice.
```

Example 2:

```
./nuke -o=5000 superSecretFile
Effect: nukes superSecretFile by overwriting its content and its filename 5000 times (!)
```

Advance Functions of Piper

Most development time were spent on writing Piper because of the sophistication of this framework. However, I think that it worths it because the framework can be reused in other projects. Here are the advance functions of Piper:

Regex Mode

There are two parameter processing mode. They are **single file processing mode** and **regex mode**. Single file processing mode is rather straight forward. Here is an example:

```
./crypter -g key data cipherText
```

In this command, only single file is processed in each parameter.
However, this parameter processing mode comes to be a problem when you want to process files in bulk like processing 1000 files at once.

That is the motivation of developing **regex mode**. In **regex mode**, multiple files can be processed with the use of regex with the syntax of Perl Regular Expression.
Says, you have the following files in the directory *./directory*:

```
secret_apple.txt
secret_banana.txt
secret_durian.txt
secret_pear.txt
secret_orange.txt
non_secret_burger.txt
non_secret_coke.txt
non_secret_french_fired.txt
non_secret_hash_brown.txt
```

You want to encrypt files named *secret_<whatever>.txt* .
The **regex expression** *secret_(\w+)[.]txt* can be used to process the desired files.
*You don't need to write *^secret_(\w+)[.]txt$* as regex_match() instead of regex_search() is used for matching filenames in **regex mode**.
In this example, ./directory is said to be a **regex base**, and *secret_(\w+)[.]txt* is said to be a **regex expression**.
The syntax of **regex master parameter** is *regexBase///regexExpression*. So the parameter of the input of this example is *./directory///secret_(w+)[.]txt*

We've got the input. Now we need an output parameter.
Let's output the files in the directory *./output* . We are going to output the filename as *cipher_<whatever>.txt* . By using the match result from **regex master**, the replace expression is *cipher_\1* .
Since the expression is to be replaced by the one matched in **regex master**, this parameter is said to be a **regex slave**. The syntax of **regex slave parameter** is *regexBase//regexReplaceExpression.*
Therefore, the parameter of output in this example is ./output//cipher_\1

We are going to let **./crypter** generate a key itself. This can be done with -g flag.
The keys also need a name. Let's write the keys in *./key* with the replace expression *key_\1*. So the **regex slave parameter** is *./key//key_\1* .

Combining the parameter above, we get the following command:

```
./crypter -g "./key//key_\1" "./directory///secret_(\w+)[.]txt" "./output//cipher_\1"
```

Regex mode is also available in GUI. You can read this Example.

There a rule in regex mode. Only one **regex master** is allowed in each block. It is because it is ambiguous to have multiple **regex master**. However, it is not necessary to have the rest of parameter be **regex slaves**. In this case, the file without regex slave would be read/written in each match of **regex master**. I believe that this can potentially be useful for reading/writing data from/to special devices in unix like */dev/tcp*.

Pipe Descriptor

Pipe descriptor is another powerful function of Piper. Pipe descriptors are used for redirecting data to stdout as well as receiving data from stdin. **Pipe descriptor** is a dark magic for average computer users as it is very complicated, and inaccessible via GUI.

The simplest pipe descriptors are (i) and (o). (i) can be used as an **input argument**, while (o) can be used as an **output argument**. By writing (i), the program will process data from stdin. By writing (o), the program will output the data to stdout.
Example 1:

```
echo "random string" | ./dummy "(i)" outputFile
Effect: creates outputFile with content “random string”
```

Example 2:

```
echo "hello" | ./dummy "(i)" "(o)"
Effect: prints “hello” on the terminal/console
```

There are a lot more pipe descriptors in The Piper Framework. They are described below:

Each parameters in Piper has three **base functions**. **source**, **destination** and **redirection**.

- **Source** means the source of data. This is where the file is read from.
- **Destination** means the source of data. This is where the data written to.
- **Redirection** is a little bit special. For input argument, writes the unprocessed data to stdout. For output argument, it redirects the data that is processed.

There are four **encoding system** for each function. They are:

- file. No encoding is used. Just read from/write to the file.
- () pipe. No encoding is used. Only one () can be used for stdin, and one for stdout. This is useful for interacting with external programs.
- [] pipe. Encode the data that data from multiple parameters can be written or read in the same command. Data written with [] pipe cannot be interpreted by external programs.
- {} pipe. Read/write the filename from/to pipe. For input argument, it open the file with the filename read as a source of data.. For output argument, it open the file with the filename read as the destination of data.

The syntax of pipe descriptor is BfB, where B is a bracket, f is a pipe function.
There are five **pipe functions** of pipe descriptor, which enables **base functions**.

- i. Enables **source**. Both bracket enclosing it have to be identical.
- o. Enables **destination**. Both bracket enclosing it have to be identical.
- r. Enables **redirection**. Brackets enclosing it can be different. Can be used for converting the type of data. The data received is NOT processed. It can only be used at the beginning of block. This parameter is neither an **input argument** nor an **output argument**. Instead, it is a **redirect argument**, which is triggered by the regex *[\{\[\(]r[\)\]\}]* . It is treated as a standalone block with single **redirect argument**.
- ir. Enables **source** and **redirection**. Brackets enclosing it can be different. The opening bracket is associated with **source**, while the closing one is associated with **destination**.
- file. Enables **source**(for Input arguments) or **destination**(for Output arguments). Brackets are not required. If there are brackets enclosing it, both brackets have to be the same. The brackets will enable and associated with **redirection**.

Here is the list of pipe descriptors along with the **encoding system** used in the base function:
To simplify the expression, (i), [i] and {i} are represent by {[(i)]}.

Pipe Descriptor	Can be used for Input Argument	Can be used for Output Argument	Source	Destination	Redirection
{i}	✔	(see below)	{} pipe	-	-
{o}	(see above)	✔	-	{} pipe	-
[(i)]	✔	X	[()] pipe	-	-
{[(o)]}	X	✔	-	-	{[()]} pipe
{[(r)]}	X*	X*	{[()]} pipe, depends on the opening bracket	-	{[()]} pipe, depends on the closing bracket
{ir)]}	✔	(see below)	{} pipe	-	{[()]} pipe, depends on the closing bracket
{ir)]}	(see above)	✔	-	{} pipe.	{[()]} pipe,

					depends on the closing bracket
[(ir)]}	✔	X	[()] pipe, depends on the opening bracket	-	{[()]} pipe, depends on the closing bracket
{[(file)]}	✔	(see below)	file	-	{[()]} pipe
{[(file)]}	(see above)	✔	-	file	{[()]} pipe

*{[(r)]} can neither be used as **an input argument** or an **output argument**. It can only be used as an **redirect argument**, which is triggered by the expression above. **Redirect argument** can only be used at the beginning of the block. It is treated as a standalone block with single **redirect argument**. The difference between {[(r)]} and {[(ir)]} is that {[(r)]} does not pass the data to **output argument** for procession, while {[(ir)]} does. {[(r)]} is only used for redirection.

Here is the table of meaning of base functions and encoding system of **input argument** and **redirect argument**.

Base function	Source meaning	Destination meaning	Redirect meaning
file	Reads data from file	-	-
() pipe	Reads data from pipe without any encoding	-	Writes the unprocessed data to stdout without any encoding
[] pipe	Reads data from pipe with encoding	-	Writes the unprocessed data to stdout with encoding
{} pipe	Reads filename from pipe, then read the file with the filename received.	-	Writes the filename to stdout. Can only be used when the information of filename is available.

Here is the table of meaning of base functions and encoding system of **output argument**.

Base function	Source meaning	Destination meaning	Redirect meaning
file	-	Writes the processed data to file	-
() pipe	-	-	Writes the processed data to stdout without any encoding
[] pipe	-	-	Writes the processed data to stdout with encoding
{} pipe	-	Reads filename from pipe, then read the file with the filename received.	Writes the filename to stdout.

Remember to use escape character, or use a couple of quotation marks to enclose (i) because parentheses have other uses in both Unix terminal and Windows console.

You are probably brain-screwed. Let's give you some examples, from the simpler one to the more complicated one.

Example 1:

```
./redate -amc=197010151238.17 {file} | ./nuke {i}
```
Effect: change the timestamp of **file** to 15/10/1970 12:38:17 , then nuke the file.

Example 2:

```
./crypter key data "(o)" | ./injector "(i)" dest
```
Effect: encrypts **data** with **key**, then inject the encrypted data to **dest**.

Example 3:

```
./crypter key1 data1 [o] key2 data2 [o] | ./injector [i] dest1 [i] dest2
```
Effect: Encrypts **data1** with **key1** and inject the encrypted data to **dest1**. Then, encrypts **data2** with **key2**, then inject the encrypted data to **dest2**. Please notice that [] pipe instead of () pipe because multiple files is piped. () pipe is not able to pipe multiple files.

Pipe Descriptor with regex mode

Pipe descriptors can also be used in regex mode. To do that, you can use <> wrapper to enclose the pipe parameter. Currently, only {[i]}, {[ir]} and {[file]} can be used with <> wrapper. Theoretically, {[o]} should also be allowed to be used with <> wrapper. However, it is unimplemented due to time constraint.
The usage of pipe with <> wrapper are generally the same, except that <> wrapper enable users to pipe regex parameter. Please notice that <{[i]}> is treated as a regex master which reads filenames from pipe.

Here is an example:
Example 1[Unix only]:

```
./dummy -f "<[./data///CIMG(\d*).JPG]>" /dev/null > archive
cat archive | ./dummy -f "<[i]>" "./out//\1.jpg"
```
Effect: The first command writes the regex expression and filenames of the file that matches the expression *./data///CIMG(\d*).JPG* along with its data to **archive**. The second command reads from **archive** and writes the files read in the first command to the directory **./out**

Advantages of Pipe Descriptor

The main advantage of using pipe descriptor is that file can be processed by another program without haven data written on the hard drive(as long as you have enough RAM, or have swap disabled). This can be useful for top secrets because most filesystems journals the metadata of files. Even though the content and the filename of nuked files cannot be viewed, it can be known that there is a file deleted by using appropriate softwares testdisk.

Another advantage is that pipe descriptor enables piper programs to interact with external programs. You can see there are pretty much commands that make use of external programs in the examples in this user guide. Some useful external programs includes echo, cat, nc, base64, dd, you name it!

The final advantage is that it is a more sophisticated data piping system than conventional programs. Conventional programs cannot pipe multiple files with single command, while Piper programs can. Piper programs can also be used with regex, while most conventional programs does not even have regex support. You can do much more with pipe descriptor than conventional data

piping system.

Fun fact: The origin of the name of this framework **Piper** is pipe descriptor.

Legal Information

This software is written by Wong Cho Ching. The source code are licensed under BSD 2-clause license, which is available in <root of this software>/LICENSE_BSD_2.txt
Meanwhile, the executables are licensed under multiple license. I, as well as the developers of libraries that is used in this software, hold the copyright ownership of the executables.

This software was written with the use of the following libraries:
wxWidgets
Botan
Boost
The executables of this software are linked to Boost and Botan statically, and are linked to wxWidgets dynamically.

Please notice that the following text have no legal effect. This message merely tell you the licenses applied to this software by my own interpretation. The accuracy and the correctness are not guaranteed in the following text.

The executables are statically linked with Botan and Boost. Therefore, the executables are derived works of Botan and Boost, which have to be licensed under Boost Software License and BSD 2-clause license. Redistribution of the executables are restricted by these licenses.
Meanwhile, the executables are dynamically linked with wxWidgets. Therefore, the executables are "work that uses the library". These files are not derived works of wxWidgets. As stated in LGPL, which is the license of wxWidgets, these executables are not restricted by LGPL.
However, the .dll files for Windows executables are licensed under LGPL as these files are built with the source code of wxWidgets, which are derived works of wxWidgets.

The licenses of wxWidgets is available in <root of this software>/LICENSE_LGPL.txt and <root of this software>/LICENSE_WIDGET.txt .
The license of Boost is available in <root of this software>/LICENSE_BOOST.txt
The license of Botan is available in <root of this software>/LICENSE_BOTAN.txt
These licenses can also be obtained from their websites. You can obtain a copy of the library from their websites as well:
wxWidgets: http://www.wxwidgets.org
Boost: http://www.boost.org/
Botan: http://botan.randombit.net/
To summarize:

Files	License(s) applied[copyright holder]
Source code	BSD 2-clause license[Wong Cho Ching]
Executables	BSD 2-clause license[Developers of Botan], BSD 2-clause license[Wong Cho Ching] and Boost Software License[Developers of Boost]
Dynamic linking files for Windows executables	LGPL(version 2)[Developers of wxWidgets] and wxWindows Library Licence(version 3.1)[Developers of wxWidgets]