

DarkCastle Grimoire



Karl Zander – pvial00@gmail.com

Abstract.

DarkCastle is an authenticated file encryption program, obfuscating the file's message and detecting if the message has been tampered with. The file is secured by Q'loQ public keys which encapsulate a passkey.

1. Introduction

DarkCastle is a program written purely in C with cryptographic primitives designed by myself and with the hope that others will donate primitives in the future. DarkCastle hopes to bring professional levels of security to files that are encrypted using the program. The program is cross platform, generally working on Linux, MacOS, FreeBSD, OpenBSD, NetBSD and Solaris.

2. Design goals

The application must employ best security practices when encrypting, decrypting or authenticating data.

The application should execute, encrypt and authenticate quickly.

Each message must be individually key wrapped with a random key.

The application must offer protection at 256-1024 bits.

The application must offer at least one public key algorithm.

At the very least the application must consist of a cipher, message authentication code and key derivation function.

3. Installation and usage

- Installation prerequisites

OpenSSL is required

GCC/Clang, make and git must be already installed

On Debian systems you may run ``apt install build-essential git libssl-dev`` to install the prerequisites

- Installation

To install DarkCastle run the following commands:

1. `git clone https://github.com/pvial00/DarkCastle`
2. `cd DarkCastle/src`
3. `make`
4. `mv castle /usr/local/bin` (This works on Mac systems, on Linux you may have to ``sudo mv castle /usr/local/bin``)

- Usage

Simply running ``castle`` will print the algorithms available and parameter format.

To encrypt a file with DarkCastle, do the following:

```
`castle <algorithm name> -e <input file> <output file> <public keyfile> <secret keyfile>`
```

To decrypt a file with DarkCastle, do the following:

```
`castle <algorithm name> -d <inputfile> <outputfile> <secret keyfile> <public keyfile>`
```

4. Key derivation

The Manja algorithm was selected to be the key derivation function for DarkCastle. Manja takes a maximum of 256 characters/bytes as it's input and thus DarkCastle has a 256 character limit for passwords.

DarkCastle uses 10,000 Manja iterations and a salt of "DarkCastleSui".

5. Key wrapping algorithm and key generation

- Key generation

Key generation is achieved using the `amagus_random()` PRNG which sources it's initial entropy for a random 1024 bit key and 128 bit nonce from `/dev/urandom`. The cipher used in the PRNG is Amagus.

For each security level a cipher of appropriate strength is selected to be the key wrapper.

256 bit key wrapper algorithm – Uvajda

512 bit key wrapper algorithm – Amagus

1024 bit key wrapper algorithm – Amagus

The key wrapper operates by first generating itself a random nonce using `amagus_random()` and a random key of appropriate length using the same generator. The cipher is then fed the key supplied by the KDF and the random nonce and encrypts key. The key is then XOR'ed with the KDF key and is written to file.

To decrypt the process is run in reverse.

6. Authentication

Authentication is achieved through use of the Ganja hash function. `ganja_hmac()` is called to generate 256 bit hashes which are used as message authentication codes for messages. The nonce/iv, key wrapped key, key wrap nonce and message are all run through `ganja_hmac()` to produce a MAC for the message that is attached to the front of the file.

Sender authentication is achieved through use of the Q'loQ (model X) public key encryption algorithm. 3072 bit keys (1536 P size) are utilized to secure the shared secret and authenticate the communication.

7. Ciphers

In the order listed in the program:

dark	256 bit
uvajda	256 bit
spock	256 bit
amagus	256 bit
amagus512	512 bit
amagus1024	1024 bit
qapla	256 bit
zanderfish2-cbc	256 bit
zanderfish2-ofb	256 bit
zanderfish2-ctr	256 bit
zanderfish3	256 bit
zanderfish3-512	512 bit
zanderfish3-1024	1024 bit
zanderfish3-ofb	256 bit

8. Cryptanalysis

TBD

9. Statistical Properties

DarkCastle was subjected to the Diehard battery of tests (dieharder) and NIST statistical testing for some ciphers. Some ciphers are still undergoing cryptanalysis and statistical testing.

ZanderFish2 and derivatives, ZanderFish3 and Dark have shown to have the best entropy qualities in testing thus far.

10. libdarkcastle API

TBD

DarkCastle has an API contained within castle_core.c that allows easy encryption or decryption of any file. This API is best accessed through use of libdarkcastle. The top level functions are as follows:

Dark - 256 bit Stream Cipher

```
void dark_encrypt(char *infile_name, long long fsize, char *outfile_name, int key_length, int  
nonce_length, int mac_length, int kdf_iterations, unsigned char * kdf_salt, unsigned char *password);
```

```
void dark_decrypt(char *infile_name, long long fsize, char *outfile_name, int key_length, int  
nonce_length, int mac_length, int kdf_iterations, unsigned char * kdf_salt, unsigned char *password);
```

Uvajda – 256 bit Stream Cipher:

```
void uvajda_encrypt(char *infile_name, long long fsize, char *outfile_name, int key_length, int  
nonce_length, int mac_length, int kdf_iterations, unsigned char * kdf_salt, unsigned char *password);
```

```
void uvajda_decrypt(char *infile_name, long long fsize, char *outfile_name, int key_length, int  
nonce_length, int mac_length, int kdf_iterations, unsigned char * kdf_salt, unsigned char *password);
```

WildThing – 256 bit Stream Cipher:

```
void wildthing_encrypt(char *infile_name, long long fsize, char *outfile_name, int key_length, int  
nonce_length, int mac_length, int kdf_iterations, unsigned char * kdf_salt, unsigned char *password);
```

```
void wildthing_decrypt(char *infile_name, long long fsize, char *outfile_name, int key_length, int  
nonce_length, int mac_length, int kdf_iterations, unsigned char * kdf_salt, unsigned char *password);
```

Spock in CBC mode:

ZanderFish2 OFB mode – functions:

```
void zander2ofb_encrypt(char *infile_name, long long fsize, char *outfile_name, int key_length, int  
nonce_length, int mac_length, int kdf_iterations, unsigned char * kdf_salt, unsigned char *password);
```

```
void zander2ofb_decrypt(char *infile_name, long long fsize, char *outfile_name, int key_length, int  
nonce_length, int mac_length, int kdf_iterations, unsigned char * kdf_salt, unsigned char *password);
```