Tricks for efficient non-AES zip encryption password-based brute force attacks

# Abstract

The original Zip encryption algorithm is well-known to have a number of flaws. This document outlines a few of these flaws and discusses methods for taking advantage of them to do a more efficient brute force password-based attack.

# Audience

It is expected that readers of this document have at least:

* A basic understanding of encryption-related concepts
* Working knowledge of programming and programming concepts
* Familiarity with zip files

# Motivation

Many tools available on the Internet perform a naïve brute force attack, attempting full decryption of the file with every possible password. This results in sub-standard performance, particularly compared to what is possible with more specialized software. For instance, Elcomsoft’s [Advanced Archive Password Recovery](https://www.elcomsoft.com/archpr.html) is one of the best such programs available, out-performing the naïve programs by 10 to 20 times.

This document outlines tactics used to make a tool that rivals the best programs available, easily surpassing most commonly available options, both free and paid. On a modern desktop (3.6 Ghz Intel® Core™ i7-4790), the techniques below were used to try over 170 million passwords per second on a small test zip file.

# Zip Encryption

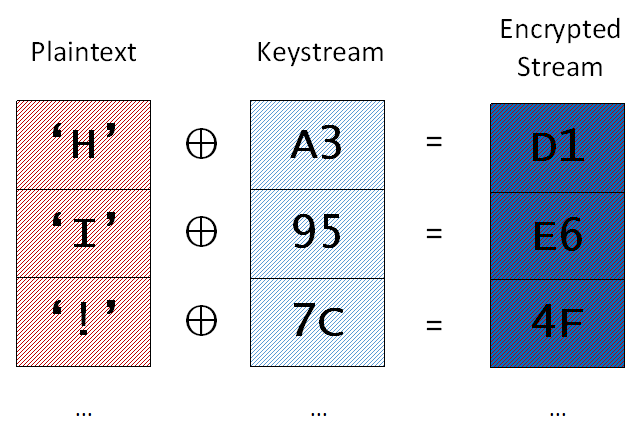
In order to discuss the weaknesses in the zip encryption algorithm and how they may be exploited, we must first understand how zip encryption works. Section 6 of the [AppNote](https://pkware.cachefly.net/webdocs/casestudies/APPNOTE.TXT) describes the steps involved in decrypting data in a zip file. It outlines them as:

1. Initialize the three 32-bit keys with the password.
2. Read and decrypt the 12-byte encryption header, further initializing the encryption keys.
3. Read and decrypt the compressed data stream using the encryption keys.

There are two key functions in zip decryption to which this document will be referring, named in the AppNote as “update\_keys” and “decrypt\_byte”. More specific details of these functions can be found in the section titled “Analysis of Encryption Algorithm and Optimizations”.

## Overview of algorithm

The zip encryption algorithm is a stream cipher. The idea is to produce a stream of pseudo-random bytes (the “keystream”) and xor this stream with the file data (the “plaintext” stream), encrypting it to produce the “encrypted” stream.



Plaintext is xor-ed with keystream to produce encrypted stream

Most stream ciphers generate the keystream based on internal state but zip files base the keystream on the combination of the keys and the plaintext bytes. In this way, the encrypted stream at byte X is based on all the data in the plaintext stream from byte 1 to X.

## 

Keystream is produced from previous key values and plaintext

## Initializing the decryption keys with the password

Zip files use three 32-bit keys to encrypt and decrypt the file data. The keys are derived from the password using a proprietary key derivation algorithm outlined in the appnote, paraphrased here:

1. Initialize the keys:

Key0 = 305419896 (0x

Key1 = 591751049 (0x23456789)

Key2 = 878082192 (0x34567890)

1. For each character in the password, call the update\_keys function with that character as the argument. More information on the update\_keys function can be found in the section “Update Keys”.

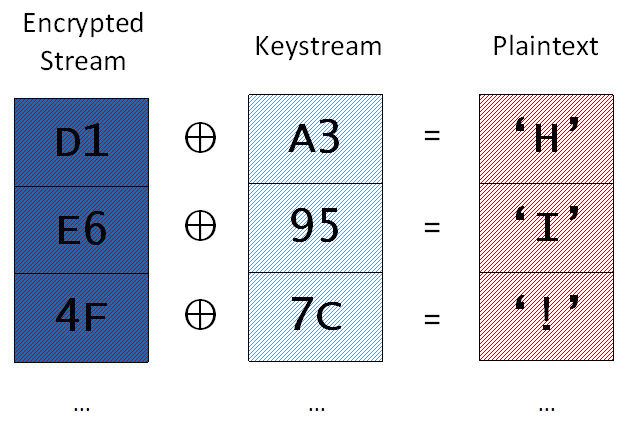
## Decrypting the encryption header

Each encrypted file in the zip has a 12 byte “encryption header” prepended to the file data. The first eleven bytes of this header are random values used to “further initialize the encryption keys”. The last byte (or two, depending on the zip software used) is a “check byte”, which can be used to help validate the password without needing to decrypt and decompress the whole file. This check byte matches the LSB of the file’s CRC value or the LSB of the file’s “last modified file time” value.

The encryption header is decrypted in the same way as the file data (see “Decrypting the file” below), but the values are not written as part of the decrypted file.

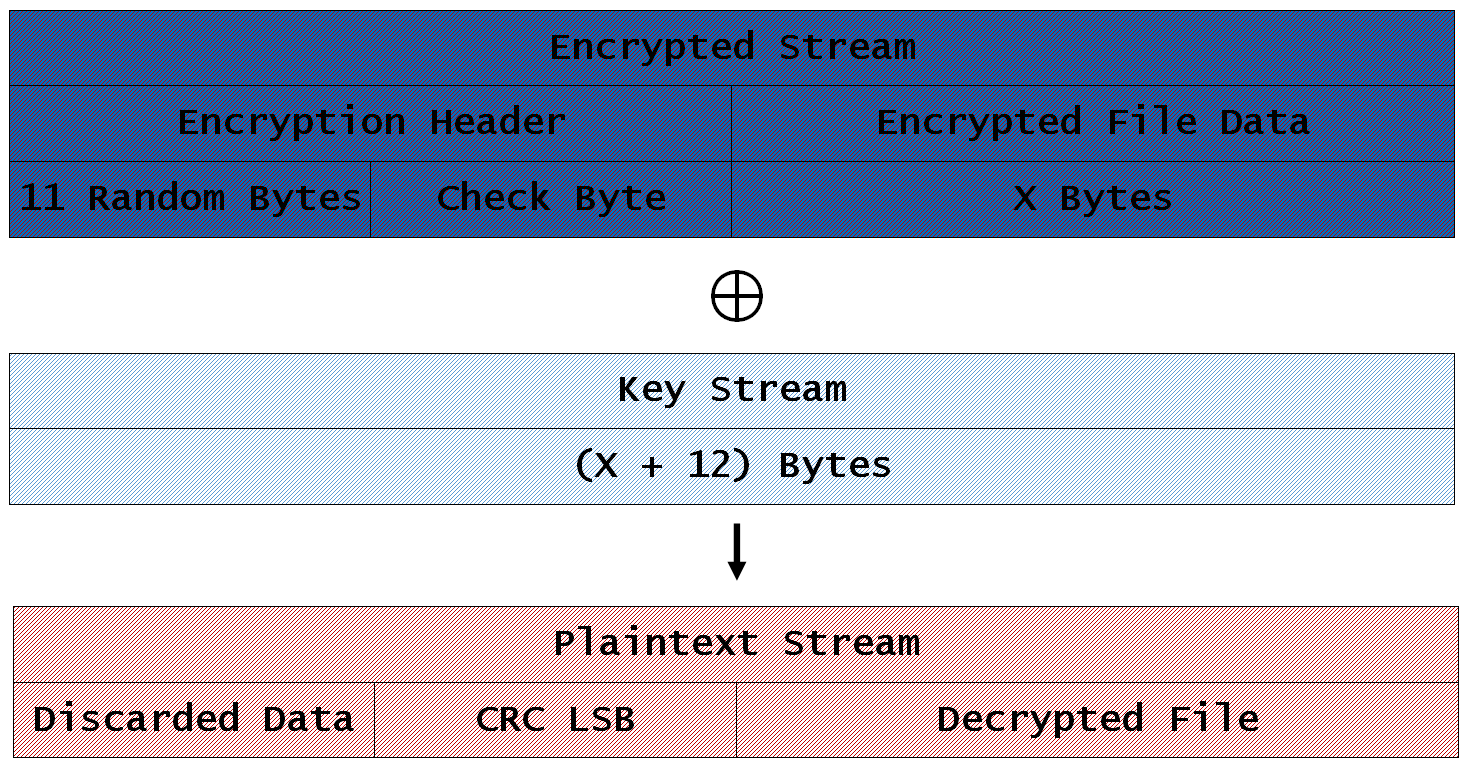
## Decrypting the file

Decrypting the file is a relatively simple process. The general idea is to xor the encrypted stream with the keystream to undo the encryption. If the same password was used as when encrypting the file, the keystream will match and the original plaintext stream will be produced.



The basic steps are:

1. Call decrypt\_byte to get the next byte of the keystream
2. Xor the next encrypted byte with the keystream byte to get the decrypted byte
3. Update the keys with the decrypted byte
4. Write out the decrypted byte



# Weaknesses

There are a couple of attributes of the zip encryption algorithm and format that can be used to more efficiently do a password-based brute force attack on an encrypted zip file.

## Inclusion of the “check byte” byte in the encryption header

Taking advantage of the check byte in the encryption header provides an enormous advantage when brute-forcing the password. The last byte of the encryption header for the file is the encrypted version of the LSB of the CRC of the file or the LSB of the last modified time of the file (depending on the software that was used to produce the zip file). These values are easily available from the file’s zip header.

The presence of this value vastly reduces the average amount of work that must be done to determine if a given password is correct. Instead of having to decrypt and decompress the whole file to check the CRC32 value, one only needs to decrypt 12 bytes to eliminate about 254 out of every 256 passwords (since the check byte could match one of up to 2 values), regardless of how large the file is. In these cases, the amount of work that is avoided is tremendous.

## Iterative key derivation function

The algorithm for deriving the keys’ values from the password is iterative, allowing for a significant savings in computational cost. As an example of this, the keys for the passwords “1234a”, “1234b”, and “1234c” all have the same values at the point where the “1234” prefix has been processed. By saving the keys after calculating them for “1234”, we can re-use them as a starting point when appending the last characters ‘a’, ‘b’, and ‘c’.

|  |  |  |  |
| --- | --- | --- | --- |
| Byte  Added | Key 0 | Key 1 | Key 2 |
|  | 0x | 0x23456789 | 0x34567890 |
| ‘1’ | 0x0f12cd62 | 0x11483398 | 0x9a8be5ce |
| ‘2’ | 0x6b644339 | 0xb7a8c716 | 0x2943427d |
| ‘3’ | 0xe0be8d5d | 0x70bb3140 | 0x7e983fff |
| ‘4’ | **0x348e6771** | **0xcecf3c76** | **0x51a09805** |
| ‘a’ | 0x1d839e03 | 0x9102925e | 0xf733f7c5 |

|  |  |  |  |
| --- | --- | --- | --- |
| Byte  Added | Key 0 | Key 1 | Key 2 |
|  | 0x | 0x23456789 | 0x34567890 |
| ‘1’ | 0x0f12cd62 | 0x11483398 | 0x9a8be5ce |
| ‘2’ | 0x6b644339 | 0xb7a8c716 | 0x2943427d |
| ‘3’ | 0xe0be8d5d | 0x70bb3140 | 0x7e983fff |
| ‘4’ | **0x348e6771** | **0xcecf3c76** | **0x51a09805** |
| ‘b’ | 0x848acfb9 | 0x47106dec | 0x98838024 |

Imagine that each character in the password is a fixed amount of work. In this tiny case, we’ve reduced computing:

“1234a” + “1234b” + “1234c” = 15 work units

Down to:

“1234” + “a” + “b” + “c” = 7 work units

# Analysis of Encryption Algorithm and Optimizations

The zip decryption algorithm provides a number of opportunities for optimization. This section analyzes the decryption functions and points out places where optimization could be helpful.

## Decrypt Byte

### Analysis

First we will look at the decrypt\_byte function. Here is the listing from the AppNote.txt file:

unsigned char decrypt\_byte()

local unsigned short temp

temp <- Key(2) | 2

decrypt\_byte <- (temp \* (temp ^ 1)) >> 8

end decrypt\_byte

While the function is named “decrypt\_byte”, it would probably more accurately called “generate\_keystream\_byte”. It takes the keys, or more accurately just key2, and uses it to generate a byte of the keystream.

There are a few interesting things to note about this function:

1. The only input to the function is the last of the encryption keys
2. All of the calculations are based on the “temp” variable
3. The “temp” variable is of the type “unsigned short”, meaning it is only 16 bits.
4. The output of the function is a single byte

In effect, this function can be described at a high level as taking the least significant 16 bits from key2 and outputting a specific byte value based on it. Another way to think about this would be that it is a lookup table with 65,536 one-byte entries.

### Optimization

Given that the entire method is effectively a fixed array, we can pre-calculate it by running the function on all 65,536 values and storing the output.

\_\_declspec(align(128)) BYTE g\_rgbDecryptByteTable[65536];

void BuildDecryptByteTable()

{

for ( DWORD i = 0; i < 65536; i++ )

{

WORD wTemp = ((i & 0xffff) | 2);

g\_rgbDecryptByteTable[i] = ((wTemp \* (wTemp ^ 1)) >> 8);

}

}

Once this is done, instead of using the function we just take key2, truncate it to an unsigned short, and use that as an index into the table.

g\_rgbDecryptByteTable[(WORD)dwKey2]

This allows us to replace an or, multiply, xor, and shift operation with just a table lookup, which is a single instruction on x86 machines.

## Update Keys

### Analysis

The update\_keys method is given as:

update\_keys(char):

Key(0) <- crc32(key(0),char)

Key(1) <- Key(1) + (Key(0) & 000000ffH)

Key(1) <- Key(1) \* 134775813 + 1

Key(2) <- crc32(key(2),key(1) >> 24)

end update\_keys

The update\_keys function uses a plaintext byte to update the state of the decryption keys. Key0 is the CRC32 of the file so far. The LSB of this is then added to key1 and then it is mixed a little by multiplying it with a constant and adding 1. Lastly, key2 is the CRC32 of the MSBs of key1.

### Optimization

Given that the CRC32 algorithm is used twice, making it as efficient as possible is important. CRC32 is a very well-studied algorithm. The web page <http://create.stephan-brumme.com/crc32/> outlines the performance of a number of different ways to calculate it. Since the keys are updated one byte at a time, the “standard” algorithm is the best performing available algorithm listed.

Along with being fast, the standard algorithm is also very useful because, similarly to decrypt\_byte above, it is largely a table lookup. The entire function can be in-lined very easily, reducing the overhead of a function call.

As a result, the first crc32 call above becomes a more confusing, but faster:

dwKey0 = g\_rgdwCRC32Table[(BYTE)(dwKey0 ^ c)] ^ (dwKey0 >> 8);

# Other Optimizations

While many of the best improvements are related to flaws in the zip encryption algorithm, standard programming optimizations can also have a noted impact.

## Parallel processing

For many years now, CPUs have come with an increasing number of cores, a trend which will likely continue. In order to take best advantage of this, the software must be designed to scale as linearly as possible, with minimal interaction between the threads.

The key points where threads interact are:

1. Acquiring new passwords to try – The main coordination between threads is the doling out of the passwords. In order to prevent contention between the different threads, each thread is assigned a block of passwords sufficiently big to keep it busy for a while. The ratio of time the thread is processing on its own to the amount of time spent getting the next block is more than large enough to prevent significant contention or caravanning.
2. Tracking passwords processed – On the opposite end of the algorithm, it is important to track how fast the program is processing passwords. A similar tactic is taken here to 1 above, where each thread tracks its own total independently and periodically the main application thread will sum them all together without locking. It’s possible that this may cause a race condition occasionally, but it is extraordinarily rare (theoretical and not yet observed) and won’t have any lasting or detrimental effect.

The key point is to use a thread per processor with as little synchronization as possible.

## Efficient use of registers

The decryption algorithm is relatively simple and doesn’t require a lot of variables. Evaluating a single password doesn’t make use of many of the machine’s registers. By computing more than one password per loop, the compiler is encouraged to make greater use of the registers. Through experimentation, evaluating 3 passwords per loop seems to give the best performance, increasing passwords per second by over 100% versus processing a single password at a time.

## Miscellaneous

### Eliminating function overhead

Through different compiler options and directives, the amount of overhead per function call can be reduced. The following all contribute to making the program more efficient:

* Disabling exception handling
* Disable runtime checks
* Disable security checks
* Inline functions

### Align data

CPUs work best when data is aligned on certain data boundaries, either due to making memory fetching more efficient or making better use of the processor’s cache. Making sure all data used by the hot code path is aligned improves performance.

### 64-bit compilation

Compiling for 64-bit CPUs improved the performance about 25% versus the 32bit code. 64 bit code allows access to more registers and produces better decompression code.

# Future Improvements

Quite some time was spent brainstorming ideas for ways to improve the speed of the algorithm, and not all of the ideas were implemented or tested.

## Specialized decompression

Once a candidate password has been found, the standard zlib library is used to fully test it. Not all of what zlib does is required for testing for the password. For example, the decompressed data is not needed other than to calculate the file’s CRC. By creating custom decompression code, the extraneous functionality can be skipped.

The gains for this will likely be small, since the zlib code path isn’t often hit, but the larger the file, the bigger a difference it will make.

## GPGPU

General Purpose Graphics Processing Unit (GPGPU) computation can make an extraordinary difference when used on the right task due to the potential availability of thousands of processors. Password cracking, particularly zip password cracking, would be an excellent use of this power, since it is trivially parallelizable and operates on fixed data with only small changes.

## Multimedia Instructions

The full instruction set available on modern x64 processors provides a number of sets of specialized instructions, like MMX, SSE, SSE2, AVX and AVX2. No attempt was made to use these instructions, but it’s possible that they might be able to be used to process passwords more quickly. These operations often allow the use of larger registers (AVX2 uses 256 bit registers), which may be useful for processing multiple bytes or passwords simultaneously.

## Distributed Processing

The algorithm used can easily be expanded to allow processing passwords across multiple machines in a cluster. A single master machine can manage and distribute chunks of passwords to workers.

# Conclusions

With a little effort, it was possible to find methods to vastly speed up password-based brute force attacks. A reasonably powerful server can process half a billion passwords per second, even without resorting to GPGPU-based solutions or custom hardware.

Despite the speed reached, it is still not nearly enough to guarantee success in even a remotely reasonable amount of time. With a sufficiently strong password, these techniques will not provide access to the zip contents within a person’s lifetime, even with an extreme amount of computing power.

To put this struggle in perspective: In the worst case scenario, a password would be long enough to make it more feasible to attack the keyspace than the password itself. This would probably only take about 16 characters if we include upper and lower-case, numbers, and common symbols. Zip files use something similar to a 96 bit key. In an unrealistic best case scenario, one could have a million custom devices each capable of attempting one trillion passwords a second. Even then, it would still take over a thousand years, on average, to find the keys that work for a single archive.

# Other Flaws

There are other known flaws related to zip encryption that are useful to note.

## Encryption Header Too Short to Affect All Key Bits

The key data is not sufficiently changed per byte of plaintext. Because of this, it is possible to work backwards from known plaintext to the original key values after the password has been processed. While the full set of keys can’t be deduced, it eliminates portions of the keyspace. Depending on the number of bytes known, the keyspace can be reduced to a size where attacking the keys themselves becomes feasible.

Because of this flaw, it is sometimes possible to decrypt a file within an hour, regardless of the password used. There are at least 2 known methods used for this purpose:

### Known Plaintext Attack

Very often a zip file will contain a number of files in it. Information about these files can be deduced from the filenames, which are not hidden in any way with standard zip encryption. For example, the extension of the file may provide a very good guess as to what the beginning of the file contains, since most file formats have “magic” bytes at the start of the file to indicate the file type. It’s also possible that the filename itself indicates a well-known file, giving away the entire file’s contents. Known plaintext attacks are reasonably effective against zip encryption, not only helping eliminate potential passwords for brute-forcing, but also in reducing the possible keyspace.

<http://old.honeynet.org/scans/scan24/sol/pedram/reference/mike_zipattacks.htm>

### Reused Key Attack

It is very common to use the same password for all the files in a zip file. This makes it so that, if there isn’t enough information from a single file for a known plaintext attack as above, data from a number of different files within the same archive can produce enough information to quickly determine the encryption keys.

## Poor RNG used

Some older zip programs were known to use very unsafe methods for generating the “random” values in the encryption header. These methods allowed the bytes to be narrowed down to a small set of values. This significantly weakens any protection the header might otherwise provide.

<http://www.securiteam.com/securitynews/5LP0A0096O.html>

# References

<https://pkware.cachefly.net/webdocs/casestudies/APPNOTE.TXT>

# See Also

<https://www.unix-ag.uni-kl.de/~conrad/krypto/pkcrack.html>

<http://web.archive.org/web/20060420195614/http://www.bokler.com/zipcrack.txt>