Cracking an old ZIP file to help open source the ANC's "Operation Vula" secret crypto code

It's not often that you find yourself staring at code that few people have ever seen, code that was an important part in bringing down the a...

8 min. read ·   View original

It's not often that you find yourself staring at code that few people have ever seen, code that was an important part in bringing down the apartheid system in South Africa, and code that was used for secure communication using one-time pads smuggled into South Africa by a flight attendant on floppy disks. But I found myself doing that one morning recently after having helped decrypt a 30 year old PKZIP file whose password had long been forgotten.

Some time ago I became interested in the secure communications used by the ANC as part of Operation Vula in the late 1980s. Operation Vula was the infiltration of ANC leaders (and materiel) into South Africa to set

up an underground network bringing together
the various elements of ANC activism operating
inside the country.

The operation needed secure communications
to be successful and these were established
using 8-bit computers, DTMF tones, acoustic
couplers and a variety of other equipment for
exchanging one-time pad encrypted messages
using programs written in PowerBASIC.

I won't go into the detail of how this worked as
Tim Jenkin, the person primarily responsible for
the encryption system, has now open sourced
the original code, and which can be found here.
Tim's write up on the encryption system can be
found here. I thoroughly recommend reading it
for the details.

The code hadn't been open sourced before for one simple reason: on leaving the UK for South Africa in 1991 he had zipped up all the source code and set a password on it. In the intervening years he'd simply forgotten the password! So, when I emailed him to ask if it had been open sourced he replied:

> I still have the Vula source code but unfortunately most of it I can't access because when I left the UK in 1991 to return to South Africa, I zipped up all the files with a password. I was able to decode and extract one of the files but it was a very early version of the software. The rest I couldn't extract because I forgot the password. When I got back to SA there was no need to access the code. I thought I would never forget the password but when I tried to decode it a few years later, I couldn't remember it.
>
> If you could find out how to decode zipped files that would release the software, which I would be more than happy to share. I have made a few attempts to crack the code but so far have been unsuccessful.

I readily agreed and he sent me two files: ALLBAS.ZIP and CODMAY93.ZIP. These were both created with an early version of PKZIP and had passwords set on them. Luckily, there is a [known plain text attack](#) against the ZipCrypto

scheme used in that era's ZIP format. And an open source implementation of the attack called [bkcrack](#).

So, it was a "simple matter" of predicting 12 bytes of plain text at a known location inside the ZIP file. Here's a sample of what's inside the ZIP file:

```
$ bkcrack -L ALLBAS.ZIP | head -n 20

bkcrack 1.7.0 - 2024-05-26
Archive: ALLBAS.ZIP
Index Encryption Compression CRC32
Uncompressed  Packed size Name
----- ---------- ----------- -------- ------------ -------
----- ----------------
    0 ZipCrypto  Shrink     b0f86b1d        163
      117 A1PSW.BAS
    1 ZipCrypto  Shrink     8fa662d4        163
      118 A2PSW.BAS
    2 ZipCrypto  Shrink     0c5a7295        163
      119 A3PSW.BAS
    3 ZipCrypto  Shrink     49907f86        179
      125 A4PSW.BAS
    4 ZipCrypto  Shrink     3d20eb7a        163
      120 A5PSW.BAS
    5 ZipCrypto  Shrink     f8b558f0        136
      128 BIOS.INC
    6 ZipCrypto  Implode    799074ed
377        278 CHKERR.INC
    7 ZipCrypto  Implode    c44ea0a5
17906        5401 CODSUBS.INC
```

```
    8 ZipCrypto  Implode     7bd7e23d
27287       8297 COMAID.BAS
    9 ZipCrypto  Implode     03dc63da
 2109       1001 COMKEY.BAS
   10 ZipCrypto  Store       3500d320
 2372       2384 CONFIG.TIM
   11 ZipCrypto  Shrink      35a85089        147
      111 CONPSW.BAS
   12 ZipCrypto  Implode     55be75ce
 2094        825 DOS.INC
   13 ZipCrypto  Shrink      3387d043        134
      127 DOSVER.INC
   14 ZipCrypto  Implode     28a32efa
 1304        535 DOSX.INC
   15 ZipCrypto  Implode     6578a66c
 3196        966 EDDY.BAS
```

Tim had some unencrypted .BAS files lying around but they were different versions than those in the file and trying the bkcrack attack using them (after running them through original PKZIP in [DOSBox](#)) was unsuccessful and I decided to apply some brain power before attempting further attacks.

ALLBAS.ZIP contained a number of files that were uncompressed, because they were already binary and not worth compressing. These files are marked as Store:

```
$ bkcrack -L ALLBAS.ZIP | grep Store
   10 ZipCrypto  Store       3500d320
 2372       2384 CONFIG.TIM
```

```
23 ZipCrypto  Store       14a285ac           2
   14 KEYCOD.EXE
25 ZipCrypto  Store       d6343ce1        4767
   4779 KEYONE.ZIP
26 ZipCrypto  Store       650778b7        6523
   6535 KEYTHREE.ZIP
30 ZipCrypto  Store       12a711cd       58172
   58184 OLDCOD.ZIP
41 ZipCrypto  Store       00000000           0
   12 TAPCOD.EXE
44 ZipCrypto  Store       55000714
12716     12728 TECOD5.ZIP
45 ZipCrypto  Store       f4f4366c        9230
   9242 TECOD6.ZIP
```

Files that are Store'd are fruitful for plaintext prediction because they have not undergone compression and there's no need to have the original file to compress in order to obtain plaintext. Focussing on the ZIP files, since the ZIP files start with a PK header, seemed like a good place to find predictable plaintext at a known position. Here are the fields in the standard PK header at the very start of a ZIP file:

	0x0 0x1	0x2 0x3	0x4 0x5	0x6 0x7	0x8 0x9	0xA 0xB	0xC 0xD 0xE 0xF
0x00	P K	0x3 0x4	Version	Flags	Method	Time	Date	CRC32
0x10	CRC32	Compressed Size	Uncompressed Size	FN Size	Extra	
0x20	File name (variable length)						
0x30	Extra Field (variable length)						

A viable attack appeared to be to predict the name of the first file in the archive. If the file name was at least 8 characters (which seemed pretty easy since at least four characters were

used for .BAS, .INC etc.) then at least 12
characters of plaintext would be available when
the file name size (offset 0x1A, 0x1B) and the
length of the extra field (which appeared to be
0x00, 0x00 in all the ZIPs Tim sent) was
added.

In the worst case, it would be possible to
bruteforce the potential names of files given
that they all appear to be uppercase/number
combinations with a maximum length of eight
characters plus extension. But that turned out
not to be necessary.

Happily, Tim had a different version of
OLDCOD.ZIP (one of the ZIP files inside
ALLBAS.ZIP) and was able to tell me that the
first file in it was COMKEY.BAS. So, I whipped
up a quick Perl program to create the necessary
plaintext in that hope that the OLDCOD.ZIP
inside ALLBAS.ZIP did start with COMKEY.BAS:

$ cat maken.pl

use strict;

use warnings;

my $outfile = "hexname-$$.txt";

while (<>) {

```perl
    chomp;

    my $bas = $_;

    print("$bas / $outfile\n");

    my $n =
sprintf("%c\x00\x00\x00$bas",length($bas));

    open G, ">$outfile";

    print G $n;

    close G;

    system("bkcrack -C ALLBAS.ZIP -c
OLDCOD.ZIP -p $outfile -o 26 -j 8");

}
```

23 minutes later bkcrack spit out the key to the ALLBAS.ZIP file and was able to decrypt it. The same key worked for CODMAY93.ZIP also.

$ time echo "COMKEY.BAS" | perl maken.pl

COMKEY.BAS / hexname-41227.txt

bkcrack 1.7.0 - 2024-05-26

[07:49:38] Z reduction using 6 bytes of known plaintext

100.0 % (6 / 6)

[07:49:38] Attack on 925073 Z values at index 33

Keys: 98e0f009 48a0b11a c70f8499

80.6 % (745571 / 925073)

Found a solution. Stopping.

You may resume the attack with the option: --continue-attack 745571

[18:13:49] Keys

98e0f009 48a0b11a c70f8499

real23m4.371s

user162m3.520s

sys0m37.752s

bkcrack does the decryption once the key has been found:

$ bkcrack -C ALLBAS.ZIP -k 98e0f009 48a0b11a c70f8499 -D ALLBAS-DECRYPTED.ZIP

bkcrack 1.7.0 - 2024-05-26

[07:52:22] Writing decrypted archive ALLBAS-DECRYPTED.ZIP

100.0 % (81 / 81)

$ bkcrack -C CODMAY93.ZIP -k 98e0f009 48a0b11a c70f8499 -D CODMAY93-DECRYPTED.ZIP

bkcrack 1.7.0 - 2024-05-26

[07:58:31] Writing decrypted archive CODMAY93-DECRYPTED.ZIP

100.0 % (40 / 40)

And with that the long encrypted source code used to help set up secure communications for the ANC was available!

Had that failed I was going to attack one of the other ZIP files using the same attack (before resorting to bruteforceing file names). I'd guessed that TECOD5.ZIP was probably a ZIP of just the file TECOD.BAS (or maybe TECOD5.BAS) based on the compressed size of TECOD.BAS in ALLBAS.ZIP). Turns out I wouldn't have had to wait 23 minutes if I'd started there:

$ time echo "TECOD5.BAS" | perl maken.pl

TECOD5.BAS / hexname-41544.txt

bkcrack 1.7.0 - 2024-05-26

[18:14:51] Z reduction using 6 bytes of known plaintext

100.0 % (6 / 6)

[18:14:51] Attack on 880113 Z values at index 33

Keys: 98e0f009 48a0b11a c70f8499

2.4 % (20737 / 880113)

Found a solution. Stopping.

You may resume the attack with the option: --
continue-attack 20737

[18:15:29] Keys

98e0f009 48a0b11a c70f8499

real0m38.152s

user4m35.318s

sys0m0.897s

The known plaintext attack against ZipCrypto
works quickly with the right plaintext. If you ever
have to do something similar it's worth
spending time thinking about the plaintext. In
particular, files that are Store'd in the ZIP file
are useful to examine since they are
uncompressed and it may be easier to predict
their contents (rather than having to find an
original file and compress it to match what's
stored in the ZIP).

Running the code

I compiled two of the programs and ran them using DOSBox. The first, RANDOM.BAS, was used to create disks of random numbers to be used as a one time pad, the other, TECOD.BAS, was used to encrypt and decrypt messages sent via email. The code I compiled and the generated executables can be found here.

Compilation is simply running the PowerBASIC compiler as follows:

C:\>EXE\PBC TECOD.BAS PowerBASIC Compiler Version 3.00b Copyright (c) 1989-1993 by Robert S. Zale Spectra Publishing, Sunnyvale, CA, USA C:\TECOD.BAS 2575 statements, 2329 lines Compile time: 00:12.0 Compilation speed: 12600 stmts/minute 45984 bytes code, 4880 bytes data, 2048 bytes stack Segments(1): 46k C:\>EXE\PBC RANDOM.BAS PowerBASIC Compiler Version 3.00b Copyright (c) 1989-1993 by Robert S. Zale Spectra Publishing, Sunnyvale, CA, USA C:\RANDOM.BAS 2194 statements, 1940 lines Compile time: 00:10.1 Compilation speed: 12600 stmts/minute 33328 bytes code, 4704 bytes data, 3072 bytes stack Segments(1): 34k C:\>

The first step is to create random data on disk to be used as a one time pad. RANDOM.EXE uses three different randomness generating algorithms (one based on a random key you type in yourself).

Encryption and decryption is done through TECOD.EXE which is password protected.
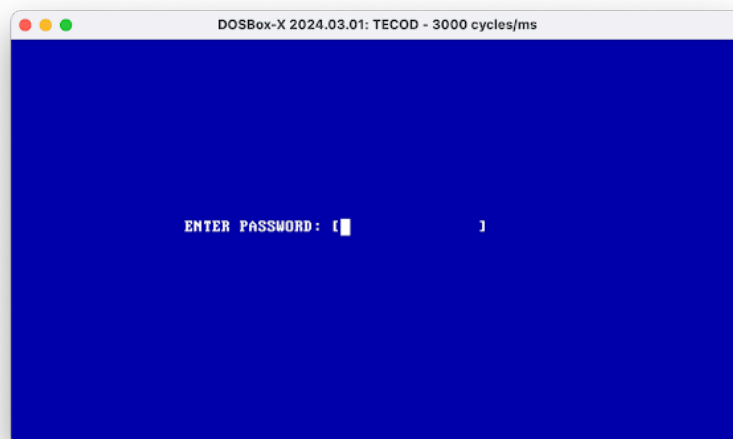
Although the password is embedded in the program and quite simple Tim Jenkin did obfuscate it as follows:

DIM PW$(PL)

PW$(9)=CHR$(66):PW$(4)=CHR$(66):PW$(1)=CHR$(84):PW$(5)=CHR$(79):PW$(2) = CHR$(73)

PW$(3)=CHR$(77):PW$(6)=CHR$(66):PW$(8)=CHR$(77):PW$(10)=CHR$(79):PW$(7)=CHR$(73)

In this particular version of the program that makes the password TIMBOBIMBO which when entered takes you to the main menu. Note that each version of these programs being sent to different members of the ANC had different passwords.

If you're interested in running these programs yourself, the manual is here.

Here are three short videos showing the creation of random data in RANDATA.1 for the key using RANDOM.EXE, followed by encrypting a message stored in PLAIN.TXT on a RAM disk (all crypto operations were meant to happen on

a RAM disk) and turning it into [PLAIN.BIN](#) (and the reverse).

Creating random data to be used as an encryption key

Encrypting a file

Here the programs (TECOD.EXE/TECOD.CNF) are on a floppy disk in A:, the data disk (containing the key file created above) are in B: and there's a RAM disk on R:. To get this to work the RANDATA.1 file created in the step above needs to be renamed to SNUM.

Decrypting a file

Here the programs (TECOD.EXE/TECOD.CNF) are on a floppy disk in A:, the data disk (containing the key file created above) are in B: and there's a RAM disk on R:. The RANDATA.1 needs to be called RNUM on B:.

There are lots of interesting details of how these programs work that deserve another longer blog post when I have time. Or a detailed study by someone else. For example, the key material is destroyed after use, the RANDOM.EXE program has multiple ways of making randomness and code to check the distribution of the random bytes created. There's an emphasis on using the RAM disk for all cryptographic operations.