


# The malware analyst's guide to aPLib decompression

---

 [0xc0decafe.com/malware-analysts-guide-to-aplib-decompression/](https://0xc0decafe.com/malware-analysts-guide-to-aplib-decompression/)

January 8, 2021



[aPLib](#) is a compression library that is very easy to use and integrate with C/C++ projects. It is a pure LZ-based compression library. There is also an executable packer based on it called [aPACK](#). Due to its ease of use and tiny footprint, it's a very popular library utilized by many malware families like [ISFB/Ursnif](#), [Rovnix](#), and many more. Knowledge about aPLib detection and aPLib decompression is crucial for every malware analyst.

This blog post dives into many internals of aPLib, explains how to detect aPLib compression with your bare eyes as well as YARA, and finally shows you how to decompress aPLib compressed blobs with several tools.

## **aPLib v1.1.1 internals**

---

[aPLib](#) is a library implementing the compression algorithm found in the executable compressor [aPACK](#). It is a pure LZ compression implementation. The library aPLib is known for its fast decompression speed and tiny footprint of the decompression code. As of writing, the current version is `v1.1.1`.

The objective of this section is not to explain how the actual compression algorithm at aPLib's core works, but rather to give a quick overview of the library itself (project structure, relevant functions, relevant structs).

Since you'll likely encounter aPLib decompression during malware analysis, I'll focus on the decompression portion of aPLib in the following. Nevertheless, this blog post should be also valuable for those who look into aPLib compression.

## The aPLib library

---

The library can be found at the [author's website](#). It comprises a lot of valuable information. First and foremost, there is the source code for decompression (but not compression!):

```
└─ src
  ├── 32bit
  │   ├── crc32.asm
  │   ├── depack.asm
  │   ├── depackf.asm
  │   ├── depacks.asm
  │   ├── scheck.asm
  │   ├── sdepack.asm
  │   ├── sgetsize.asm
  │   └── spack.asm
  ├── 64bit
  │   ├── crc32.asm
  │   ├── depack.asm
  │   ├── depackf.asm
  │   ├── depacks.asm
  │   ├── scheck.asm
  │   ├── sdepack.asm
  │   ├── sgetsize.asm
  │   └── spack.asm
  ├── depack.c
  ├── depack.h
  ├── depacks.c
  └── depacks.h
```

Second, there is some additional user documentation ( [html](#) and [chm](#) ). Third, there are libraries to statically or dynamically link against. Several platforms are supported including Windows:

```
└─ lib
  ├── dll
  │   ├── aplib.dll
  │   ├── aplib.h
  │   └── aplib.lib
  ├── dll64
  │   ├── aplib.dll
  │   ├── aplib.h
  │   └── aplib.lib
```

The library offers several decompression functions that fall in three classes:

- `aP_depack` , `aP_depack_asm` , and `aP_depack_asm_fast` assume valid input data and may crash if the input is invalid
- `aP_depack_safe` and `aP_depack_asm_safe` catch decompression errors and do not crash on invalid input data
- `aPsafe_depack` is a function wrapper adding a header to the compressed data

At their core, they all utilize the LZ-based compression algorithm. In the following sections, I'll have a look at each of the three classes.

## **aP\_depack\***

---

The decompression function `aP_depack` decompresses a compressed binary blob. The counterpart of `aP_depack_safe` is the function `aPsafe_pack` .

The following function signature is defined in `depack.h` :

```
unsigned int aP_depack(const void *source, void *destination);
```

The functions of this class assume that the compressed data is valid. This saves some sanity checks, which in turn results in faster decompression and a smaller footprint of the decompression code. However, it is likely that they crash if an error is encountered.

## **aP\_depack\_safe\***

---

The functions of this class solely add additional sanity checks. If they encounter an error condition, they return `APLIB_ERROR` (defined as `0xFFFFFFFF` ). Furthermore, they require the length of the input (compressed data) and the size of the output (decompressed data) as seen in the signature of the `aP_depack_safe` function:

```
unsigned int aP_depack_safe(const void *source,
                          unsigned int srclen,
                          void *destination,
                          unsigned int dstlen);
```

Otherwise, they are equivalent to the functions from the `aP_depack*` class.

## **aPsafe\_depack**

---

The decompression function `aPsafe_depack` decompresses a compressed binary blob safely. It is a wrapper around the functions of the class. The counterpart of `aPsafe_depack` is the function `aPsafe_pack` .

The function signature of `aPsafe_depack` resembles the signature of `aP_depack_safe` :

```
unsigned int aPsafe_depack(const void *source,
                          unsigned int srclen,
                          void *destination,
                          unsigned int dstlen);
```

`aPsafe_depack` requires that the compressed blob starts with a header. This header comprises additional information regarding the blob. This is for example very useful if we want to send an aPLib compressed blob over the network. The header structure looks like the following struct:

```
struct aPLib_header {
  DWORD tag;
  DWORD header_size;
  DWORD packed_size;
  DWORD packed_crc;
  DWORD orig_size;
  DWORD orig_crc;
}
```

The struct `aPLib_header` has a size of 24 bytes. This holds on `x86` and `x64` systems. But there are several checks in the library that ensure that the `header_size` is at least 24 bytes. The following screenshot shows a PE executable packed with `appack` :

```

0000h: 41 50 33 32 18 00 00 00 B7 93 00 00 66 9C 0E FC AP32....."..fæ.ü
0010h: 6B C9 01 00 C9 36 22 29 4D 38 5A 90 38 03 66 02 kÉ..É6")M8Z.8.f.
0020h: 04 09 71 FF 81 B8 C2 91 01 40 C2 15 C6 80 09 1C ..qÿ.Â'.@Â.Æ€.. aPLib
0030h: 0E 1F BA F8 00 B4 09 CD 21 B8 01 4C C0 0A 54 68 ..°ø.'.Í!..LÀ.Th
0040h: 69 73 20 0E 70 72 6F 67 67 61 6D 87 63 47 6E 1F is .proggam#cGn.
0050h: 4F 74 E7 62 65 AF CF 75 5F 98 69 06 44 4F 7E 53 Otcbe~iu ~i.DO~S

```

compressed PE executable: 24 bytes header (magic highlighted), followed by compressed payload

We can see the magic `AP32` ( `0x32335041` ), directly followed by the header size of `0x18` / 24 bytes. The next four DWORDs are the `packed_size` , `packed_crc` , `orig_size` , and `orig_crc` . After the header comes the payload, which starts in this case with `MZ` since we compressed a PE executable. We will use this fact later on for detection.

## Detect if a binary statically links against aPLib

Detecting if a binary comprises the capability of aPLib compression/decompression is straightforward in the case it dynamically links against aPLib and it comprises a valid IAT (Import Address Table). However, in the case of custom/malicious binaries, it typically statically links against aPLib.

Still, there are many ways how we can detect this. First, we've already seen constants like `AP32` ( `0x32335041` ), which we could leverage to detect aPLib. But there are also several strings present that refer to aPLib itself or its author (Jørgen Ibsen) as seen in the following screenshot:

21E0h:	2E 7A 66 B3	B8 4A 61 C4	02 1B 68 5D	94 2B 6F 2A	.zf³.JaÄ..h]”+o*
21F0h:	37 BE 0B B4	A1 8E 0C C3	1B DF 05 5A	8D EF 02 2D	7%.’;ž.Ä.ß.Z.ï.-
2200h:	0D 0A 0D 0A	61 50 4C 69	62 20 76 31	2E 31 2E 31	....aPLib v1.1.1
2210h:	20 20 2D 20	20 74 68 65	20 73 6D 61	6C 6C 65 72	- the smaller
2220h:	20 74 68 65	20 62 65 74	74 65 72 20	3A 29 0D 0A	the better :)..
2230h:	43 6F 70 79	72 69 67 68	74 20 28 63	29 20 31 39	Copyright (c) 19
2240h:	39 38 2D 32	30 31 34 20	4A 6F 65 72	67 65 6E 20	98-2014 Joergen
2250h:	49 62 73 65	6E 2C 20 41	6C 6C 20 52	69 67 68 74	Ibsen, All Right
2260h:	73 20 52 65	73 65 72 76	65 64 2E 0D	0A 0D 0A 4D	s Reserved....M
2270h:	6F 72 65 20	69 6E 66 6F	72 6D 61 74	69 6F 6E 3A	ore information:
2280h:	20 68 74 74	70 3A 2F 2F	77 77 77 2E	69 62 73 65	http://www.ibse
2290h:	6E 73 6F 66	74 77 61 72	65 2E 63 6F	6D 2F 0D 0A	nsoftware.com/..
22A0h:	0D 0A 00 00	00 00 00 00	00 00 00 00	00 00 00 00	.....
22B0h:	00 00 00 00	A8 E0 C8 53	00 00 00 00	B4 35 00 00	....”àÈS.....’5..

Strings that

refer to aPLib and its author

These ASCII strings are:

- “aPLib v1.1.1 – the smaller the better :)”
- “Copyright (c) 1998-2014 Joergen Ibsen, All Rights Reserved.”
- More information: <http://www.ibsensoftware.com/>

Another way would be detection via matching the assembly code. Tools like mkYARA can help you to generate (strict/relaxed) YARA rules for assembly functions/algorithms.

However, at this point, I do not want to reinvent the wheel and I just refer to one of the freely available YARA rules like the one from “\_pusher\_”:

```

rule aPLib : Jorgen Ibsen
{
  meta:
  author="_pusher_"
  date="2016-09"
  description="www.ibsensoftware.com/products_aPLib.html"
  strings:
  $a0 = { 60 8B 74 24 24 8B 7C 24 28 8B 44 24 2C FC 33 DB B2 80 39 18 74 42 A4 B3 02
E8 6D 00 00 00 73 F6 33 C9 E8 64 00 00 00 73 }
  $a1 = { 60 8B 74 24 24 8B 7C 24 28 FC 33 DB 33 D2 A4 B3 02 E8 6D 00 00 00 73 F6 33
C9 E8 64 00 00 00 73 1C 33 C0 E8 5B 00 00 00 }
  $a3 = { B2 80 33 DB A4 B3 02 E8 6D 00 00 00 73 F6 33 C9 E8 64 00 00 00 73 1C 33 C0
E8 5B 00 00 00 73 23 B3 02 41 B0 10 E8 4F 00 00 00 12 C0 73 F7 75 3F AA EB D4 E8 4D
00 00 00 2B CB 75 10 E8 42 00 00 00 EB 28 AC D1 E8 74 4D 13 C9 EB 1C 91 48 C1 E0 08
AC E8 2C 00 00 00 3D 00 7D 00 00 }
  $a4 = { 61 94 55 B6 80 A4 FF 13 73 F9 33 C9 FF 13 73 16 33 C0 FF 13 73 1F B6 80 41
B0 10 FF 13 12 C0 73 FA 75 3A AA EB E0 FF 53 08 02 F6 83 D9 01 75 0E FF 53 04 EB 24
AC D1 E8 74 2D 13 C9 EB 18 91 48 C1 E0 08 AC FF 53 04 3B 43 F8 73 0A 80 FC 05 73 06
83 F8 7F 77 02 41 41 95 8B C5 }
  $a5 = { B2 80 A4 B6 80 FF 13 73 F9 33 C9 FF 13 73 16 33 C0 FF 13 73 1F B6 80 41 B0
10 FF 13 12 C0 73 FA 75 3C AA EB E0 FF 53 08 02 F6 83 D9 01 75 0E FF 53 04 EB 26 AC
D1 E8 74 2F 13 C9 EB 1A 91 48 C1 E0 08 AC FF 53 04 3D 00 7D 00 00 73 0A 80 FC 05 73
06 83 F8 7F 77 02 }
  $a6 = { B2 80 31 DB A4 B3 02 E8 6D 00 00 00 73 F6 31 C9 E8 64 00 00 00 73 1C 31 C0
E8 5B 00 00 00 73 23 B3 02 41 B0 10 E8 4F 00 00 00 10 C0 73 F7 75 3F AA EB D4 E8 4D
00 00 00 29 D9 75 10 E8 42 00 00 00 EB 28 AC D1 E8 74 ?? 11 C9 EB 1C 91 48 C1 E0 08
AC E8 2C 00 00 00 3D 00 7D 00 00 73 0A 80 FC 05 73 06 83 F8 7F 77 02 }
  $a7 = { 33 C9 FF D3 73 16 33 C0 FF D3 73 23 B6 80 41 B0 10 FF D3 12 C0 73 FA 75 42
AA EB E0 E8 46 00 00 00 02 F6 83 D9 01 75 10 E8 38 00 00 00 EB 28 AC D1 E8 74 48 13
C9 EB 1C 91 48 C1 E0 08 AC E8 22 00 00 00 3D 00 7D 00 00 73 0A 80 FC 05 73 06 83 F8
7F 77 02 41 41 95 }
  $a8 = { 33 C9 FF 14 24 73 18 33 C0 FF 14 24 73 21 B3 02 41 B0 10 FF 14 24 12 C0 73
F9 75 3F AA EB DC E8 43 00 00 00 2B CB 75 10 E8 38 00 00 00 EB 28 AC D1 E8 74 41 13
C9 EB 1C 91 48 C1 E0 08 AC E8 22 00 00 00 3D 00 7D 00 00 73 0A 80 FC 05 73 06 83 F8
7F 77 02 41 41 95 }
  $a9 = { 33 C0 FF 13 73 1F B6 80 41 B0 10 FF 13 12 C0 73 FA 75 3A AA EB E0 FF 53 08
02 F6 83 D9 01 75 0E FF 53 04 EB 24 AC D1 E8 74 2D 13 C9 EB 18 91 48 C1 E0 08 AC FF
53 04 3B 43 F8 73 0A 80 FC 05 73 06 83 F8 7F 77 02 41 41 95 }
  $a10 = { 60 8B 74 24 24 8B 7C 24 28 FC B2 80 33 DB A4 B3 02 E8 6D 00 00 00 73 F6 33
C9 E8 64 00 00 00 73 1C 33 C0 E8 5B 00 00 00 73 23 B3 02 41 B0 10 E8 4F 00 00 00 12
C0 73 F7 75 3F AA EB D4 E8 4D 00 00 00 2B CB 75 10 E8 42 00 00 00 EB 28 AC D1 E8 74
4D 13 C9 EB 1C 91 48 C1 E0 08 AC E8 2C 00 00 00 3D 00 7D 00 00 73 0A 80 FC 05 73 06
83 F8 7F 77 02 41 41 95 }
  //taken from r!sc aspr unpacker,
  $a11 = { B2 80 8A 06 46 88 07 47 02 D2 75 05 8A 16 46 12 D2 73 EF 02 D2 75 05 8A 16
46 12 D2 73 4A 33 C0 02 D2 75 05 8A 16 46 12 D2 0F 83 D6 00 00 00 02 D2 75 05 8A 16
46 12 D2 13 C0 02 D2 75 05 8A 16 46 12 D2 13 C0 02 D2 75 05 8A 16 46 12 D2 13 C0 02
D2 75 05 8A 16 46 12 D2 13 C0 74 06 57 2B F8 8A 07 5F 88 07 47 EB A0 B8 01 00 00 00
02 D2 75 05 8A 16 46 12 D2 13 C0 02 D2 75 05 8A 16 46 12 D2 72 EA 83 E8 02 75 28 B9
01 00 00 00 02 D2 75 05 8A 16 46 12 D2 13 C9 02 D2 75 05 8A 16 46 12 D2 72 EA 56 8B
F7 2B F5 F3 A4 5E E9 58 FF FF FF 48 C1 E0 08 8A 06 46 8B E8 B9 01 00 00 00 02 D2 75
05 8A 16 46 12 D2 13 C9 02 D2 75 05 8A 16 46 12 D2 72 EA 3D 00 7D 00 00 73 1A 3D 00
05 00 00 72 0E 41 56 8B F7 2B F0 F3 A4 5E E9 18 FF FF FF 83 F8 7F 77 03 83 C1 02 56
8B F7 2B F0 F3 A4 5E E9 03 FF FF FF 8A 06 46 33 C9 D0 E8 74 12 83 D1 02 8B E8 56 8B
F7 2B F0 F3 A4 5E E9 E8 FE FF FF 5D 2B 7D 0C 89 7D FC 61 }

```

```

condition:
any of them
}

```

Nevertheless, there is still the possibility that all strings are overwritten, constants like `AP32` are changed or are dynamically computed. Just remember that having not a match does not rule out aPLib usage completely but it makes it very unlikely.

## Detect aPLib compression with your bare eyes and YARA

The following three sections shows you how to detect aPLib compression with your bare eyes and suggest several YARA rules to automate detection.

### aPLib header

If the compressed blob is safely packed, then it is quite easy to find them within larger blobs. All we need to do is looking for the aPLib magic `AP32` and the default header size of 0x18:

```

0 1 2 3 4 5 6 7 8 9 A B C D E F 0123456789ABCDEF
0000h: 41 50 33 32 18 00 00 00 B7 93 00 00 66 9C 0E FC AP32....."..fæ.ü
0010h: 6B C9 01 00 C9 36 22 29 4D 38 5A 90 38 03 66 02 kÉ..É6")M8Z.8.f.
0020h: 04 09 71 FF 81 B8 C2 91 01 40 C2 15 C6 80 09 1C ..qÿ. Â'.@Â.Æ€.. aPLib
0030h: 0E 1F BA F8 00 B4 09 CD 21 B8 01 4C C0 0A 54 68 ..°ø.' .Í! .LÀ.Th
0040h: 69 73 20 0E 70 72 6F 67 67 61 6D 87 63 47 6E 1F is .proggam#cGn.
0050h: 4F 74 E7 62 65 AF CF 75 5F 98 69 06 44 4F 7E 53 Otcbe~ïu ~i.DO~S

```

compressed blob beginning with AP32 header

This boils down to searching for the byte sequence `0x4150333218000000`. We can write a quick and dirty YARA signature:

```

rule aplib_compressed_blob_with_header {
  meta:
    author = "Thomas Barabosch"
    version = "20200109"
    description = "Detects aPLib compressed blobs that comprise an aPLib
header."
  strings:
    $aplib_compressed_with_header = { 41 50 33 32 18 00 00 00 }
  condition:
    $aplib_compressed_with_header
}

```

### Compressed PE executables without aPLib header

However, as a malware analyst, you will stumble upon aPLib compressed blobs that do not comprise an aPLib header very frequently. At least, the good news is that the trained eye can easily spot aPLib compressed PE files. These blobs of compressed PE files do not start with the classic `MZ` magic but with `M8Z`:





equal: `0x7F07454C4602011E1501` . These bytes include the ELF magic `ELF` , which is not at the beginning but does not get disfigured like the PE magic `MZ` to `M8Z` . Right now, I am not sure if this is a consistent behavior across all ELF files or just for ELF x64 files.

Again, we can write a YARA rule for this:

```
rule aplib_compressed_elf_executable {
  meta:
    author = "Thomas Barabosch"
    version = "20200109"
    description = "Detects aPLib compressed ELF executables. Note that there may be a aPLib header starting 24 bytes BEFORE the match!"
  strings:
    $aplib_compressed_elf = { 7F 07 45 4C 46 02 01 1E }
  condition:
    $aplib_compressed_elf
}
```

The YARA rules that I've presented here leave room for improvement. For example, we can check for cases where we have an aPLib header followed by a compressed PE executable, and so on. Be creative and let me know what you've found out!

## aPLib decompression

---

Finally, we've learned so much about aPLib compression, how to spot it with our bare eyes and detect it with YARA. But there is one final piece missing: we need to talk about aPLib decompression. The following sections show you how to achieve aPLib decompression with three different tools.

### aPLib decompression with appack

---

Before looking at more complex scenarios, we can always resort to the tools that come with [aPLib](#). The library comes with an example tool called `appack` . The source of this tool is stored under `examples/appack.c` and there are several `make` files for various platforms. `appack` offers two commands `c` and `d` :

```
appack, aPLib compression library example
Copyright 1998-2014 Joergen Ibsen (www.ibsensoftware.com)
Syntax:
Compress      : appack c <file> <packed_file>
Decompress    : appack d <packed_file> <depacked_file>
```

For instance, we can decompress an aPLib compressed blob with the `d` command as in the following snippet illustrated:

```
> appack d bin_ls.bin bin_ls
appack, aPLib compression library example
Copyright 1998-2014 Joergen Ibsen (www.ibsensoftware.com)
Decompressed 66101 -> 151352 bytes in 0.01 seconds
```

That's it, pretty simple. But this is not suitable for more complex scenarios, e.g. automation. Here, we have two options. First, we can use the aPLib library itself and write C programs. Second, we can automate aPLib decompression using Python.

## aPLib decompression with malduck

---

Lately, I utilize [Malduck](#) a lot. So, let's see how we can decompress an aPLib compressed blob with it. We can decompress these files with the following script based on [Malduck](#):

```
import malduck
import sys
def main(argv):
    if len(argv) != 2:
        print('Usage: aplib.py PATH_TO_APLIB_COMPRESSED_BUFFER')

    with open(argv[1], 'rb') as f:
        data = f.read()
        try:
            res = malduck.aplib(data)
            if res:
                with open(argv[1] + '_aplib_decompressed', 'wb') as g:
                    g.write(res)
            else:
                print(f'Malduck did not decompress the buffer.')
        except Exception as e:
            print(f'Could not aplib decompress: {e}')
if name == 'main':
    main(sys.argv)
```

The function `malduck.aplib` may take a flag called `headerless`. This flag forces headerless compression and does not check for the `AP32` magic. It defaults to `True`.

Even though this script seems to be trivial, it is a perfect skeleton for more complex tasks that resolve around aPLib compression. For instance, if you've to write a script for extracting the malware configuration of a specific family, which happens to use aPLib as part of the way how it stores its configuration.

## aPLib decompression with aplib-ripper

---

Another great tool is [aplib-ripper](#) by [herrcore](#). It rips one or several aPLib compressed PE executables from a blob. It searches for the string `M8Z`, forces a headerless decompression, and finally verifies and trims the output with [pefile](#).