yz1-writeup.org

Preface

I recently ran into a super old-school buffer overflow while fuzzing the `Yz1` archive (de)compression library [0].

The intention with this write-up is to go from a crash to code execution in one of the file archival software that bundles the `Yz1` library – namely, `IZArc` [2]. The target platform is Windows 10 64bit (although both `IZArc` and `Yz1` are 32bit-only).

The analysis is made with Ghidra [6] coupled with the PHAROS OOAnalyzer plugin [7]. My (sometimes horribly misleading) renaming of functions, variables and structure/class members tend to be prefixed with `x_`.

The image base for `Yz1.dll` in our analysis is `0x10000000` and the version is 0.30 (as is shipped with `IZArc`, but newer versions of `Yz1` are also vulnerable).

Introduction

`Yz1` is an archaic compression format developed by YAMAZAKI at Binary Technology. It was part of their DeepFreezer archiver software. [1]

Both of these components are closed-source and proprietary. However, `Yz1` is distributed as a shareware binary-only DLL and it's bundled with a few modern file-archivers – see e.g. `IZArc` [2], `ZipGenius` [3] and `Explzh` [9].

The interface for `Yz1` is somewhat interesting. There are a few standalone functions that tries to verify that an archive is valid. There are also functions for retrieving filenames and their metadata and in an archive [4]. However, in order to compress or decompress files, there is a single (public) function named `Yz1` [4]:

```
int WINAPI Yz1(const HWND wnd, LPCSTR cmd, LPSTR buf, const DWORD siz);
```

Every argument other than `cmd` can be `NULL` or `0` for window-less use where no feedback is to be received from the module itself.

The `cmd` argument specifies what operation should be performed and with what options [5]. Some of these arguments include:

```
c       - Create archive
x       - Expand archive
-cN     - Check timestamp according to N
-iN     - Silnce status output according to N
```

This makes working with the `Yz1` API kind of like working with the command-line interfaces of traditional (un)archivers like `tar` or `zip`.

As mentioned, the focus in this write-up is on `IZArc` [2] and its use of the `Yz1` library. The functionality in `Yz1` that we'll pay attention to is the functionality that's used by `IZArc`.

The Yz1 header

The (for this write up) first relevant entrypoint, before the `Yz1()` function is reached, is `Yz1CheckArchive()`. `IZArc` uses this function to validate `Yz1` archives before processing them.

The prototype looks like this [4]:

```
BOOL WINAPI Yz1CheckArchive(LPCSTR filename, const int mode);
```

The first argument is the filename of an archive to check. The second argument is what mode to check. There are a number of checking modes defined in `Yz1.h` [10]:

```
CHECKARCHIVE_BASIC 1
[...]
CHECKARCHIVE_ALL 16
```

For any checking mode that isn't `CHECKARCHIVE_ALL`, the function will return `true` or `false` depending on whether the archive is valid. A mode of `CHECKARCHIVE_ALL` introduces more return values, despite the return type.

In any case, since our target program `IZArc` seem to always invoke `Yz1CheckArchive` with a mode of `CHECKARCHIVE_BASIC`, we can ignore the other modes.

The `Yz1CheckArchive()` function, as well as the generic `Yz()` function (in extraction mode) takes us to a class method with the following signature:

```
int __thiscall YzFile_DecodeHeader(yzFileDecode *this, char *x_path);
```

This method is far too complex to distil in its entirety, but it performs a number of noteworthy operations. First off, it starts by reading a 0x14 byte header from the input file:

```
/*
 * 1000e7b7
 */
x_size = _fread(&x_header,1,0x14,x_yz1File->fp);
```

For example, with an archive containing the following three files:

```
>>> from pathlib import Path
>>> for p in Path().glob("*.txt"):
...     print(f"{p}: {p.stat().st_size:#x} bytes")
...
aaaa.txt: 0x25 bytes
bbbbbbbbbbbbbbbb.txt: 0x1d bytes
cccccccccccccccccccc.txt: 0x21 bytes
```

The header will look something like this:

```
$ hexdump -e '4/1 "%02X" "\n"' demo.yz1
# 0: Archive magic (yz01)
797A3031
# 1: Flags; used to, e.g., indicate whether the archive is password-protected.
30363030
# 2. ???
000000B2
# 3. The chunk of memory required to decode the filenames.
#
#    > file_count * sizeof(DWORD) * 2 * len(all_filenames_incl_NUL)
#
#    In our example, that is:
#
#    > 3*4*2 + len("aaaa.txt\0bbbbbbbbbbbbbbbb.txt\0cccccccccccccccccccc.txt\0")
0000004F
# 4. File count
00000003
```

The reason for the additional two DWORDs per file for the third field in the header is for metadata, as can be seen when `yzFileDecode::YzFile_DecodeHeader` allocates and decodes the filenames into a chunk of that size:

```
# _malloc(this->x_totalFilenameSize)
0:000> bu YZ1!yzFileDecode::YzFile_DecodeHeader + 0x56b
0:000> g
...
Breakpoint 0 hit
...

0:000> dd esp L1
0070eae8  0000004f

0:000> p
eax=03955330

0:000> bu YZ1!yzFileDecode::YzFile_DecodeHeader + 0x643
0:000> g
...
Breakpoint 1 hit
...

0:000> dd 03955330 L20
03955330  baadf00d baadf00d baadf00d baadf00d
03955340  baadf00d baadf00d baadf00d baadf00d
03955350  baadf00d baadf00d baadf00d baadf00d
03955360  baadf00d baadf00d baadf00d baadf00d
03955370  baadf00d baadf00d baadf00d abeefeee
03955380  abababab feababab 00000000 00000000
03955390  1dfe6d47 2000c430 000b0001 000b0004
039553a0  000b0003 000b000b 000b000b 000b000b

0:000> p
0:000> dc 03955330
03955330  25000000 1d000000 21000000 a26bf15e  ...%.......!^.k.
03955340  478ff05e 61616161 61616161 7478742e  ^..Gaaaaaaaa.txt
03955350  62626200 62626262 62626262 62626262  .bbbbbbbbbbbbbbb
03955360  78742e62 63630074 63636363 63636363  b.txt.cccccccccc
03955370  63636363 63636363 742e6363 ab007478  cccccccccc.txt..
03955380  abababab feababab 00000000 00000000  ................
03955390  1dfe6d47 2000c430 000b0001 000b0004  Gm..0.. ........
039553a0  000b0003 000b000b 000b000b 000b000b  ................
```

In the last memory display above, we see that the first three DWORDs correspond with the file sizes in big-endian (0x25, 0x1d, 0x21). After that are three DWORDs that I'm too lazy to figure out what they mean (yes, there really are three – notice that the file named `aaaa.txt` has 4 0x61). And finally are the NUL-separated filenames.

This chunk of memory is then processed in a method with the following signature:

```
yzDecHead *__thiscall yzDecHead(yzDecHead *this,
                                uchar *x_filenames,   /* chunk dumped above */
                                long *x_fileCount,    /* 0x3 */
                                yzFileEv *x_yzFileEv,
                                long *x_filenameSize, /* 0x4f */
                                bool *x_success);
```

The bounds for each filename is retrieved with the following C-ish code:

```
/*
 * 0x1000d2fd
 */
DVar8 = *x_fileCount;
[...]
if ((uint)*x_filenameSize < DVar8 * 0xc) {
    [...]
} else {
    x_fileCount = (long *)(x_filenames + DVar8 * 8); // adjust for metadata
    [...]
    uVar9 = 0;
    plVar7 = x_fileCount;
    while ((plVar7 < x_filenames + *x_filenameSize &&
           (*(uchar *)plVar7 != '\0'))) {
      plVar7 = (long *)((int)plVar7 + 1);
      uVar9 = uVar9 + 1;
    }
[...]
```

With our example archive, `*x_fileCount` is 3 and `*x_filenameSize` is 0x4f. The reuse of `x_fileCount` in the decompilation looks weird, but `x_filenames + DVar8 * 8` adjusts for the initial `x_fileCount * sizeof(DWORD) * 2` of metadata in the `x_filenames` buffer.

As can be seen, it doesn't matter how long any of the filenames are, so long as a NUL-byte is encountered somewhere in the `x_filenames` chunk (otherwise we'd run into an out-of-bounds read).

Even so, `Yz1` operates under the assumption that filenames are limited to `FNAME_MAX32` bytes. From the publically available `Yz1.h` [10]:

```
#if !defined(FNAME_MAX32)
#define FNAME_MAX32 512
#define FNAME_MAX   FNAME_MAX32
#else
#if !defined(FNAME_MAX)
#define FNAME_MAX   128
#endif
#endif
```

After `yzFileDecode::YzFile_DecodeHeader` and `yzDecHead::yzDecHead` has decoded and processed the header and filenames, the filenames are stored with their actual lengths for later use. This information is used when extracting the archive and/or listing its files with this exported structure and these functions:

```
typedef struct {
    DWORD   dwOriginalSize;
    DWORD   dwCompressedSize;
    DWORD   dwCRC;
    UINT    uFlag;
    UINT    uOSType;
    WORD    wRatio;
    WORD    wDate;
    WORD    wTime;
    char    szFileName[FNAME_MAX32 + 1];
    char    dummy1[3];
    char    szAttribute[8];
    char    szMode[8];
} INDIVIDUALINFO, FAR *LPINDIVIDUALINFO;

int Yz1FindFirst(HARC x_harc, LPCSTR x_pattern, LPINDIVIDUALINFO x_dst);
int Yz1FindNext(HARC x_harc, LPINDIVIDUALINFO x_dst);
```

Both of these functions invoke a method starting at `0x10002de0` that enforce the `FNAME_MAX32` (512/0x200) byte limit (sorry for the lack of cleanup!):

```
[...]
/*
 * LAB_10002f17
 */
if (*(uint *)(*(int *)(iVar3 + 0x10) + 0x14 + uVar2 * 0x1c) < 0x200) {
    iVar3 = x_getPathInstance((cls_10002bc0 *)
                              (*(int *)(this->mbr_34 + 4) + 0xc),this->mbr_48);
    /*
     * NOTE: This is not important right now, but it will matter during
     * exploitation.  Filenames shorter than 0x10 bytes are stored
     * inline at iVar3 + 4.  Filenames GTE 0x10 are allocated a separate
     * buffer whose address is stored at iVar3 + 4.
     */
    if (*(uint *)(iVar3 + 0x18) < 0x10) {
        x_filenameSrc = (char *)(iVar3 + 4);
    }
    else {
```

```
        x_filenameSrc = *(char **)(iVar3 + 4);
    }
    x_filenameDst = x_dst->szFileName;
    do {
        x_chr = *x_filenameSrc;
        *x_filenameDst = x_chr;
        x_filenameSrc = x_filenameSrc + 1;
        x_filenameDst = x_filenameDst + 1;
    } while (x_chr != '\0');
}
else {
    /*
     * x_dst->szFileName = "too_long_file_name\0"
     */
    *(undefined4 *)x_dst->szFileName = 0x5f6f6f74;          /* _oot */
    *(undefined4 *)(x_dst->szFileName + 4) = 0x676e6f6c;    /* gnol */
    *(undefined4 *)(x_dst->szFileName + 8) = 0x6c69665f;    /* lif_ */
    *(undefined4 *)(x_dst->szFileName + 0xc) = 0x616e5f65;  /* an_e */
    *(undefined2 *)(x_dst->szFileName + 0x10) = 0x656d;     /* em */
    x_dst->szFileName[0x12] = '\0';
}
[...]
```

A stack-based buffer overflow

Not all code paths pay attention to the recorded lengths of the filenames. The one my fuzzer ran into is a function that starts at `0x10005080`. It `sprintf(..., "expanding %s", ...)` with the file currently being extracted for a logging message.

It's kind of interesting too, because – similar to the snippet above – the call to `sprintf()` also checks whether the filename is inline (that is, if its length is below 0x10). But it doesn't check that the filename is below `FNAME_MAX32`.

```
/*
 * 100055b5
 */
if (this_00->mbr_18 < 0x10) {
    pDVar6 = &this_00->mbr_4;
}
else {
    pDVar6 = (DWORD *)this_00->mbr_4;
}
_sprintf(&local_264,"expanding %s",pDVar6)
```

Mo' bugs mo' problems

In working to exploit the fuzzed bug in the last section, I ran into a situation where we had written N bytes on the stack before the first [RJC]OP gadget. However, after the first gadget we could only write a handful of subsequent gadgets. Otherwise, we'd run into another bug earlier in the extraction process.

Similar to the previous flaw, this flaw is caused by a stack overflow. It happens in `yzFileDecode::DecodeFile`. Ghidra produces a somewhat wonky decompilation of this method, so the following C-ish code has been rewritten for clarity (at the expense of not being an accurate representation of its disassembly – although the important locations are commented):

```
/*
 * 1000eec0
 */
int yzFileDecode::DecodeFile(char *param_1, int *param_2)
{
    int rc;
    int duplicateCount = 0;
    unsigned int i = 0;
    char buf[XXX];

    [...];

    /*
     * LAB_1000efa0
     */
    do {
        if (this->x_yzDecHead->filenames == NULL) {
            [...];
        }

        /*
         * 1000f170
         */
        _sprintf(buf, "%s%s", this->x_dirname,
                 this->x_yzDecHead->x_filename[4 + i * 0x1c]);

        rc = x_hasFile(buf);
        if (rc) {
            duplicateCount += 1;
        }
    } while (i < this->x_yzDecHead->x_fileCount); /* 1000f02e */

    [...];

    /*
     * 1000f036
     */
    if (duplicateCount > 0) {
        x_overwriteWarning();
```

```
    }

    [...];
}
```

In the snippet above, each filename in the archive is checked for existence on disk. If it already exist, a warning message *may be* presented to the user (`Yz1` only shows GUI messages if it's been given a `HWND`).

As with the previous bug, the call to `sprintf()` is unchecked. If a path in `1000f170` is large enough, we'll overflow the stack.

However, one major issue with this flaw is that the call to `sprintf()` at `1000f170` prepends the extraction directory to the filename. This complicates exploitation. It also makes it difficult to exploit the first bug mentioned in this writeup, because whether this bug is triggered before depends partly on something we can't control (i.e. where the user chooses to extract the archive).

With that in mind, one positive aspect of this bug is that we can overwrite `this->x_yzDecHead` and cause an invalid memory access in the `do-while()` conditional. This leads to quick control of execution if we overwrite a SEH.

Sploitin' like its the 00s

There are two important aspects of the decoding process of the archive header and its filenames:

1. As mentioned above, filenames are separated by their terminating NUL-byte in the initial processing.
2. The chunk referenced as `x_filenames` above will contain as much decoded data as is specified by the third DWORD in the archive header (excl. leading metadata).

As will be seen in the PoC, I haven't bothered reverse engineering and reimplementing the (de|en)coding algorithm (presumably based on Huffman). However, it seems that the archive filenames and their content are adjacent each other such that:

```
- filename_0
- filename_1
- filename_2
- ...
- content_of_filename_0
- content_of_filename_1
- content_of_filename_2
- ...
```

If we'd modify the third DWORD in the header (0x4F in the demonstrative archive above) to a larger value, the buffer referenced as `x_filenames` would not only contain the decoded filenames, but also (part of, depending on the value) their decoded contents.

This means that we can use the `Yz1` library itself to write our exploit for the unchecked calls to `sprintf()` . The general approach looks like:

1. Create an archive with N files.
2. Set a breakpoint before the filenames are encoded, but after the metadata has been constructed.
3. Remove the terminating NUL-byte for one of the filenames (effectively concatenating them).
4. Let the process finish.
5. Increase the third DWORD in the header to a size that includes the length of all file content.

The result is that the decoding process will interpret file contents as filenames. This gives us ample opportunity to create a source buffer large enough to overflow the stack in the call to `sprintf()` .

This can be accomplished with `pykd` [8] – which is not only a plugin for `WinDbg` but also very usable as a standalone Python module for automated debugging.

As for exploit mitigations, the changelog for `IZArc` mentions that ASLR and DEP was introduced in `IZArc` version 4.3. However, that only applies to the main executable and *some* plugins (presumably the plugins for which the author has access to the source code).

With that said, only two of the shipped modules are non-rebased: `Tar32.dll` and `cabinet5.dll` .

Anyway, after having removed the `NUL` between two filenames in the archive, the file contents that will later be interpreted as a filename will contain the following:

1. Enough data to overflow the stack (incl. SEH).
2. A SEH gadget that adjusts `esp` and returns into our ROP sled.
3. Gadgets that prepares the stack with appropriate arguments for `VirtualAlloc()`
4. Gadget to invoke `VirtualAlloc()` by using its IAT slot in `Tar32.dll` .
5. Our shellcode.

Unfortunately, it's difficult to write a reliable exploit due to the extraction directory being prepended to our overflowing "filename". The approach taken in the PoC is to spray the SEH overwrite after adjusting the initial bogus data in an attempt for the overwrite to land on an appropriate DWORD boundary. The alignment is done in the interval `[0,4)` – i.e. `len(path) % 4` (where `path` includes the trailing `\`). So, there's a 1 in 4 shot for success if the extraction path is unpredictable.

Demonstration

Because the PoC uses `Yz1.dll` and `pykd` to create the payload, and because `Yz1.dll` is a 32-bit Windows-only module, the payload has to be created on a Windows system with a 32-bit Python >=3.6.

Example:

```
> "C:\Program Files (x86)\Python38\python.exe" exploit.py   \
    --dll "C:\Program Files (x86)\IZArc\Yz1.dll"            \
    --output C:\Users\user\Downloads\archive.yz1            \
    --align C:\Users\user\Downloads\archive
=> created: C:\Users\user\Downloads\archive.yz1
=> extraction path alignment: 0
```

Note that `--align` can also be an integer `[0, 4)` or left out completely (in which case it's derived from the `--output` path).

References

1. https://www.madobe.net/archiver/lib/yz1.html
2. https://ja.wikipedia.org/wiki/DeepFreezer
3. https://www.izarc.org/
4. http://zipgenius.com/
5. https://gist.github.com/illikainen/6f228c42b77c21c1e2954966b54179fc
6. https://gist.github.com/illikainen/b33fbc933246981ce49d8d62aabd43cf
7. https://ghidra-sre.org/
8. https://github.com/cmu-sei/pharos
9. https://githomelab.ru/pykd/pykd
10. https://www.ponsoftware.com/en/
11. https://gist.github.com/illikainen/16ce066720e58dffd8a80fffe877df14