UPX Anti-Unpacking Techniques in IoT Malware

cujo.com/upx-anti-unpacking-techniques-in-iot-malware/

August 17, 2020

All posts
August 17, 2020

Attackers are always at the forefront of inventing new techniques to stay covert. It comes by no surprise that their tradecraft is also subject to continuous improvement and development. One interesting facet of their tactics is how they are utilizing binary packing. Packing plays an important part in evasion and covert deploying of malicious binaries:

- It helps attackers to avoid endpoint anti-virus detection software when deploying the malicious binary to the target device

- Packing reduces the size of the binary on disk and in transit significantly: this comes handy, when low visibility is required from the attacker's side or for instance, an exploit kit requires small binaries to be delivered, otherwise it would break, and crooks would not be able to disseminate malicious binaries properly
- Packing also enables to hide plain-text strings seen normally in the binary, throttling the analysis that defenders may do on the binary

A Primer on Packing

There are commonly four packer types that we distinguish, but oftentimes the boundary between these might be thin. These are:

- Compressors: greatly reduces the size of the binary
- Cryptors: using cryptographic algorithm to obfuscate the contents of the binary
- Protectors: used widely as copyright protection, for ex.: Virtual machine (VM)-based digital rights and copy protection
- Installer: binary wrapped around an installer for easy installment

There are many ways of identifying packed binaries:

- Examining visual representation of the binary: to explore similarities by visualizing certain byte patterns of the binary; other application of it is to spot important structures in the binary or to analyze given file formats in order to better understand it
- Non-standard section names
- Section with both Writable and Execute permissions may be a possible sign of a packer
- Address of entry point somewhere else, then in the first section
- Presence of certain function calls
- Increased entropy: taking the frequency of each byte value that are present in each block or section, then applying a certain entropy formula to calculate entropy scores for given sections: higher entropy scores may indicate the presence of encryption or packing
- Very few imports and very few recognizable strings
- Using file identification tools (file, trid, etc..)

Unpacking mechanism

A simple routine stub code is embedded into the now packed binary, that also acts as the entry point. As it starts running, it will allocate a new memory region in which it unpacks the original code. Then the program code jumps to the Original Entry Point (OEP) and continues with the execution of the original, unpacked program.

UPX

One of most known packers is UPX. It is an open-source implementation of an advanced file compressor, supporting lots of executable types, Linux and Windows too. Over the years, UPX has been judged both as a legitimate and a gray zone tool, as both innocent and malicious programs like to use and abuse it commonly.

UPX has been abused in a few different ways for many years:

- Use of Vanilla UPX: malware developers just take the original UPX compressor and apply it to their malware. Easy to unpack, either automatically or manually.
- Use of Vanilla UPX, then the packed binary is hex modified: from an attacker perspective, the goal is to break automatic unpacking by modifying some hex bytes. This will break the automatic *upx –d* unpacking method. Some of the most common modifications include:
- Rewriting the UPX! magic headers
- ELF magic bytes are modified
- Copyright string is modified
- Section header names are modified
- Extra junk bytes added throughout the binary
- Custom UPX: since UPX is open source, anyone can go and look at its source code on Github and modify certain methods or re-write complete functions. Once the custom UPX program code is compiled, and then applied to a malicious binary, there is no way to get a full picture and understand its custom functions or modified routines right off the bat. Our only resort to understand the mechanism of the custom packing is to manually reverse engineer it.

UPX Header Structures Abused

Since it was not an easy task to find abused, malicious packed binaries to every plot, we created skeleton, packed programs, that will raise different error messages on different abuse scenarios, to try to emulate all exceptions:

```
neo@zion:~$ upx -d ps.upx
                       Ultimate Packer for eXecutables
                          Copyright (C) 1996 - 2018
UPX 3.95        Markus Oberhumer, Laszlo Molnar & John Reiser   Aug 26th 2018

        File size         Ratio      Format      Name
   --------------------   ------   -----------   -----------
upx: ps.upx: CantUnpackException: p_info corrupted

Unpacked 1 file: 0 ok, 1 error.
```

p_info corrupted

```
neo@zion:~$ upx -d hello.go.upx
                       Ultimate Packer for eXecutables
                          Copyright (C) 1996 - 2018
UPX 3.95        Markus Oberhumer, Laszlo Molnar & John Reiser   Aug 26th 2018

        File size         Ratio      Format      Name
   --------------------   ------   -----------   -----------
upx: hello.go.upx: CantUnpackException: l_info corrupted

Unpacked 1 file: 0 ok, 1 error.
```

l_info corrupted

```
neo@zion:~$ upx -d ps.upx
                       Ultimate Packer for eXecutables
                          Copyright (C) 1996 - 2018
UPX 3.95        Markus Oberhumer, Laszlo Molnar & John Reiser   Aug 26th 2018

        File size         Ratio      Format      Name
   --------------------   ------   -----------   -----------
upx: ps.upx: CantUnpackException: p_info corrupted

Unpacked 1 file: 0 ok, 1 error.
```

p_info corrupted

In order to understand what each of these corruption means, we need to dig further down and try to understand how the UPX header builds up after packing. We will find valuable information in the open source project’s source code, inside linux.h:

> struct *b_info // 12-byte header before each compressed block*
> {
>
> uint32_t sz_unc; *// uncompressed_size*
> uint32_t sz_cpr; *// compressed_size*
> unsigned char b_method; *// compression algorithm*
> unsigned char b_ftid; *// filter id*
> unsigned char b_cto8; *// filter parameter*
> unsigned char b_unused; *// unused*
>
> };
>
> struct *l_info // 12-byte trailer in header for loader (offset 116)*
> {
>
> uint32_t l_checksum; *// checksum*
> uint32_t l_magic; *// UPX! magic [55 50 58 21]*
> uint16_t l_lsize; *// loader size*
> uint8_t l_version; *// version info*
> uint8_t l_format; *// UPX format*
>
> };
>
> struct *p_info // 12-byte packed program header follows stub loader*
> {
>
> uint32_t p_progid; *// program header id [00 00 00 00]*
> uint32_t p_filesize; *// filesize [same as blocksize]*
> uint32_t p_blocksize; *// blocksize [same as filesize]*
>
> };

So, each of these structs store important information for the packer to work properly, so when the unpacking method is initiated, the target program is uncompressed as intended. If the corresponding hex values to these structs are altered, we will get the previously seen error messages.

Rundown

Now that we have an understanding how the fields and structures build up, let's look at an example where UPX has been abused in some way or shape. Looking at the following hash:

bc88a57e1203f5eec08d34b59d9de43fa121f9d92cc773c17ebfbe848a2f88cd

```
00000080  00 00 00 00  00 00 00 00  00 00 00 00  06 00 00 00  ................
00000090  04 00 00 00  10 CA C5 92  59 54 53 99  20 08 0D 0C  ........YTS. ...
000000A0  00 00 00 00  00 00 00 00  00 00 00 00  94 00 00 00  ................
000000B0  60 00 00 00  02 00 00 00  7F 3F 64 F9  7F 45 4C 46  `........?d..ELF
000000C0  01 00 02 00  03 00 0D 64  81 04 DF 6D  B3 DD 08 34  .......d...m...4
```

Packed malware UPX header

```
00005C50  EF 00 00 00  08 00 24 00  00 FF 00 00  00 00 59 54  ......$.......YT
00005C60  53 99 00 00  00 00 00 00  59 54 53 99  0D 0C 02 0A  S.......YTS.....
00005C70  C8 17 5E DE  0F 4A 9B F2  D0 01 00 00  BC 00 00 00  ..^..J..........
00005C80  58 B2 00 00  49 00 00 29  A0 00 00 00               X...I..)....
00005C90
```

Packed malware UPX trailer

We need to focus on the underlined hex values. The trained eye will immediately spot that at **0x98**, the **UPX!** Magic header has been altered with the hex bytes of **YTS**. That is part of the *l_info* structure. If we go further and try to match the bytes with the previously shown code structure, it is clear that "**20 08"** is the loader size. **0D** is the version info and **0C** should be the UPX format. Right after *l_info* structure is the *p_info* structure at **0xA0**. From the source code we know that **p_progid** should be "**00 00 00 00"**. After that comes **p_filesize** and **p_blocksize**, both storing the same size value, but in our case, it has been altered and erased. Fortunately, the value for the filesize and blocksize is also stored at **0x5C80**, which is "**58 B2 00 00"**. We just need to put this value into **p_filesize** and **p_blocksize**. The values at these 3 offsets should always be the same. We also see in the trailer section that the string **YTS** appears twice. We also need to alter these back to **UPX! (55 50 58 21).** The trailer section also contains "**0D 0C**" again, which is the version and format info from the *l_info*.

We looked at *l_info* and *p_info*, but still have not touched *b_info*. Actually, there are two *b_info* structures in a UPX packed binary, one for the compressed target program and one for the compressed part of the loader itself.

If we look inside i386-linux.elf-entry.S (ELF x86), we will find that the offset of the first struct *b_info* for the compressed program is given in **.long O_BINFO**. The other *b_info* for the compressed part of the loader, is located soon after the instruction **call unfold** near the label **main**:, reached from **_start**.

```
main:
        pop ebp  // &decompress
        call unfold
        .long O_BINFO
            // compressed fold_elf86 follows
```

Offset of the first struct b_info for the compressed program

We can even debug the whole process and find the exact offsets by uncommenting the seen int3 instruction and recompiling the UPX binary. Once we debugged a sample file and found the offsets, we can make note of them and see the corresponding hex values:

```
sz_pack2 = -4+ _start
_start: .globl _start
////     nop; int3   // DEBUG
         push eax   // space for entry address
```

Uncomment the int3 instruction to manually debug

```
00000110  00 00 00 00  00 00 40 FF  0C AC 00 00  F8 51 00 00  ......@......Q..
00000120  02 49 00 00  EE 6B BF FC  55 89 E5 53  E8 00 18 81  .I...k..U..S....
00000130  C3 77 AF 06  0A 78 04 BF  0B D8 FF A3  88 5B 5D C3  .w...x.......[].
```

The first b_info struct at 0x118

```
00005590  5D E8 47 FF  FF FF AC 00  00 00 AC 06  00 00 D1 05  ].G.............
000055A0  00 00 02 49  00 00 FB FF  FF FF 57 51  50 89 E6 81  ...I......WQP...
000055B0  EC 00 10 00  00 89 E7 6A  08 59 F3 A5  55 89 E5 AD  .......j.Y..U...
```

The second b_info struct at 0x559A

The **b_method** and the **b_ftid** must be the same for all *b_info* in the same file. There is a quick way to gain that information out of a binary, by running *upx —fileinfo* on a sample, packed binary.

```
neo@zion:~$ ./upx/src/upx.out --fileinfo dateupx
                       Ultimate Packer for eXecutables
                          Copyright (C) 1996 - 2020
UPX git-d7ba31+ Markus Oberhumer, Laszlo Molnar & John Reiser   Jan 23rd 2020

dateupx [amd64-linux.elf, linux/amd64]
    48784 bytes, compressed by UPX 13, method 2, level 8, filter 0x49/0x08

WARNING: this is an unstable beta version - use for testing only! Really.
```

UPX fileinfo argument

We have yet to see values for *b_info* being altered in the wild, but this might be another abuse surface for UPX packed binaries at later stages. Currently, the most prevalent UPX abuses are alteration of *l_info* and *p_info*.

Let's summarize our findings of values for the structures in detail:

b_info for the compressed portion of the stub loader [12 bytes]					
sz_unc [4]	sz_cpr [4]	b_method [1]	b_ftid [1]	b_cto8 [1]	b_unused [1]
0C AC 00 00	F8 51 00 00	02	49	00	00

b_info for the target program [12 bytes]					
sz_unc [4]	sz_cpr [4]	b_method [1]	b_ftid [1]	b_cto8 [1]	b_unused [1]
AC 06 00 00	D1 05 00 00	02	49	00	00

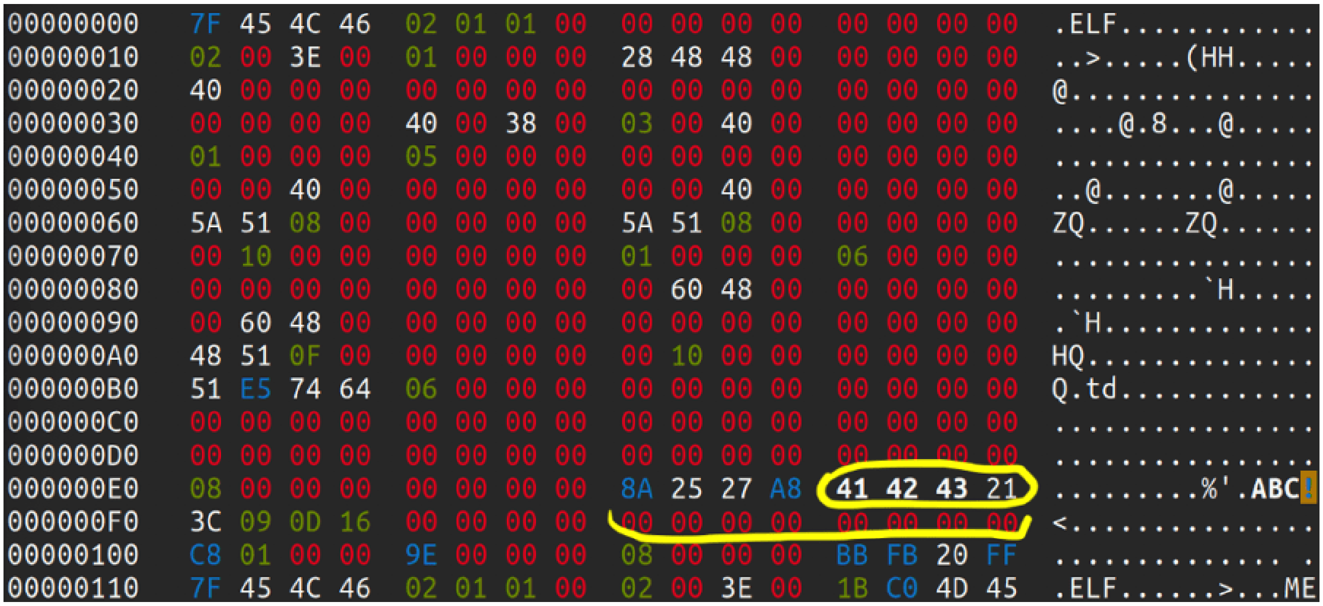
l_info [12 bytes]				
l_checksum [4]	l_magic [4]	l_size [2]	l_version [1]	l_format [1]
10 CA C5 92	59 54 53 99	20 08	0D	0C

p_info [12 bytes]		
p_progid [4]	p_filesize [4]	p_blocksize [4]
00 00 00 00	58 B2 00 00	58 B2 00 00

Once we put those in, our UPX packed binary now successfully unpacks.

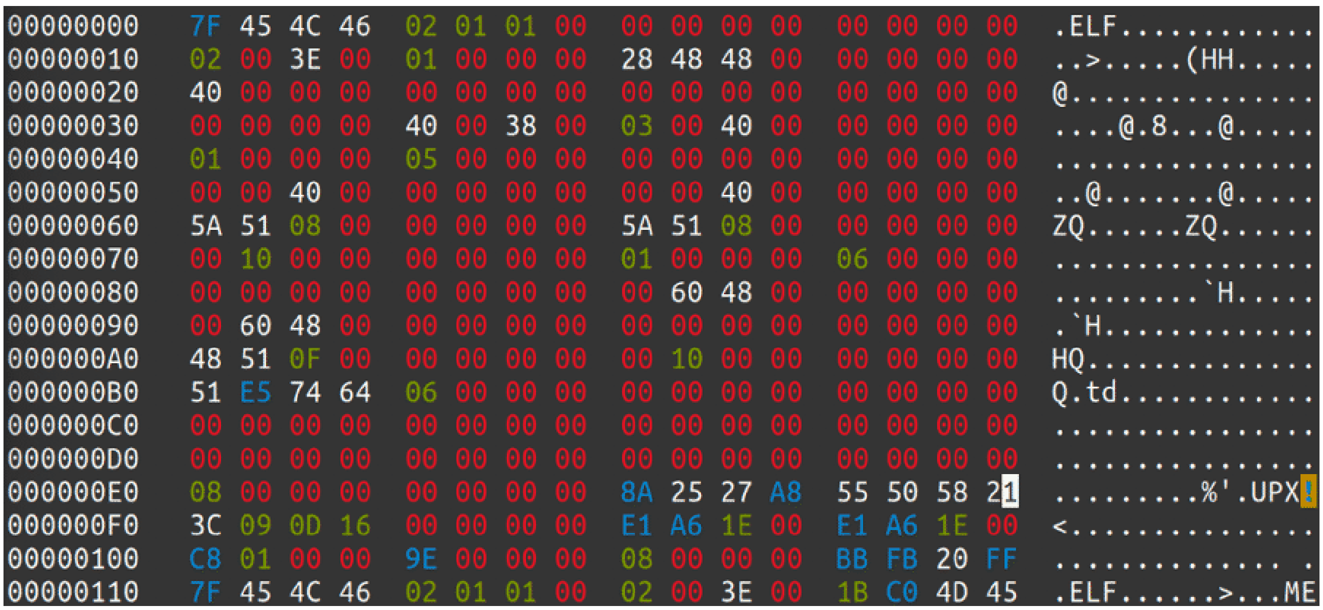
Mozi

Let's look at another example: Mozi is one of the prevalent IoT malware families in 2020. It is a perfect example for *p_info* alteration, as UPX packed Mozi binaries have been observed to come with 0 value of the **p_filesize** and **p_blocksize** fields. This will defeat automatic unpacking, and in order to get the unpacked binary, we would need to figure out the correct values of these fields. Employing what we learned previously we quickly find the corresponding filesize values in the trailer, and we can add that into *p_info*: **"E1 A6 1E 00"**



```
00000000  7F 45 4C 46  02 01 01 00  00 00 00 00  00 00 00 00  .ELF............
00000010  02 00 3E 00  01 00 00 00  28 48 48 00  00 00 00 00  ..>.....(HH.....
00000020  40 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  @...............
00000030  00 00 00 00  40 00 38 00  03 00 40 00  00 00 00 00  ....@.8...@.....
00000040  01 00 00 00  05 00 00 00  00 00 00 00  00 00 00 00  ................
00000050  00 00 40 00  00 00 00 00  00 00 40 00  00 00 00 00  ..@.......@.....
00000060  5A 51 08 00  00 00 00 00  5A 51 08 00  00 00 00 00  ZQ......ZQ......
00000070  00 10 00 00  00 00 00 00  01 00 00 00  06 00 00 00  ................
00000080  00 00 00 00  00 00 00 00  00 60 48 00  00 00 00 00  .........`H.....
00000090  00 60 48 00  00 00 00 00  00 00 00 00  00 00 00 00  .`H.............
000000A0  48 51 0F 00  00 00 00 00  00 10 00 00  00 00 00 00  HQ..............
000000B0  51 E5 74 64  06 00 00 00  00 00 00 00  00 00 00 00  Q.td............
000000C0  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
000000D0  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
000000E0  08 00 00 00  00 00 00 00  8A 25 27 A8  41 42 43 21  .........%'.ABC!
000000F0  3C 09 0D 16  00 00 00 00  00 00 00 00  00 00 00 00  <...............
00000100  C8 01 00 00  9E 00 00 00  08 00 00 00  BB FB 20 FF  .............. .
00000110  7F 45 4C 46  02 01 01 00  02 00 3E 00  1B C0 4D 45  .ELF......>...ME
```

An example of employing UPX header corruption and erased p_filesize and p_blocksize fields



```
00000000  7F 45 4C 46  02 01 01 00  00 00 00 00  00 00 00 00  .ELF............
00000010  02 00 3E 00  01 00 00 00  28 48 48 00  00 00 00 00  ..>.....(HH.....
00000020  40 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  @...............
00000030  00 00 00 00  40 00 38 00  03 00 40 00  00 00 00 00  ....@.8...@.....
00000040  01 00 00 00  05 00 00 00  00 00 00 00  00 00 00 00  ................
00000050  00 00 40 00  00 00 00 00  00 00 40 00  00 00 00 00  ..@.......@.....
00000060  5A 51 08 00  00 00 00 00  5A 51 08 00  00 00 00 00  ZQ......ZQ......
00000070  00 10 00 00  00 00 00 00  01 00 00 00  06 00 00 00  ................
00000080  00 00 00 00  00 00 00 00  00 60 48 00  00 00 00 00  .........`H.....
00000090  00 60 48 00  00 00 00 00  00 00 00 00  00 00 00 00  .`H.............
000000A0  48 51 0F 00  00 00 00 00  00 10 00 00  00 00 00 00  HQ..............
000000B0  51 E5 74 64  06 00 00 00  00 00 00 00  00 00 00 00  Q.td............
000000C0  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
000000D0  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
000000E0  08 00 00 00  00 00 00 00  8A 25 27 A8  55 50 58 21  .........%'.UPX!
000000F0  3C 09 0D 16  00 00 00 00  E1 A6 1E 00  E1 A6 1E 00  <...............
00000100  C8 01 00 00  9E 00 00 00  08 00 00 00  BB FB 20 FF  .............. .
00000110  7F 45 4C 46  02 01 01 00  02 00 3E 00  1B C0 4D 45  .ELF......>...ME
```

After fixing the corrupted UPX header and the values of p_filesize and p_blocksize

Manual Unpacking from radare2

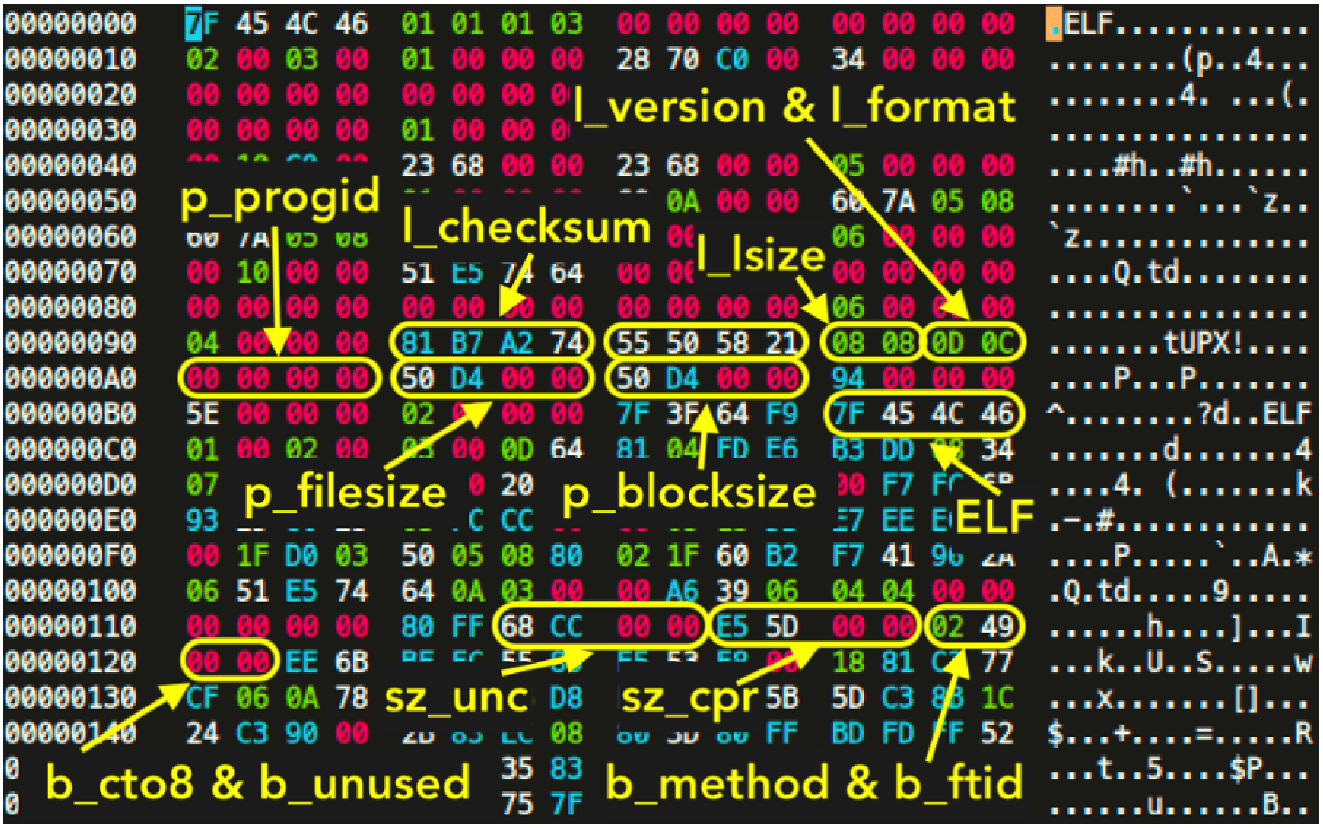
Where the automatic unpacking does not work with *upx –d* tool, even after fixing all the mentioned discrepancies and modified fields, we may attempt to manually extract the unpacked executable image from memory, like the following:

Resources:

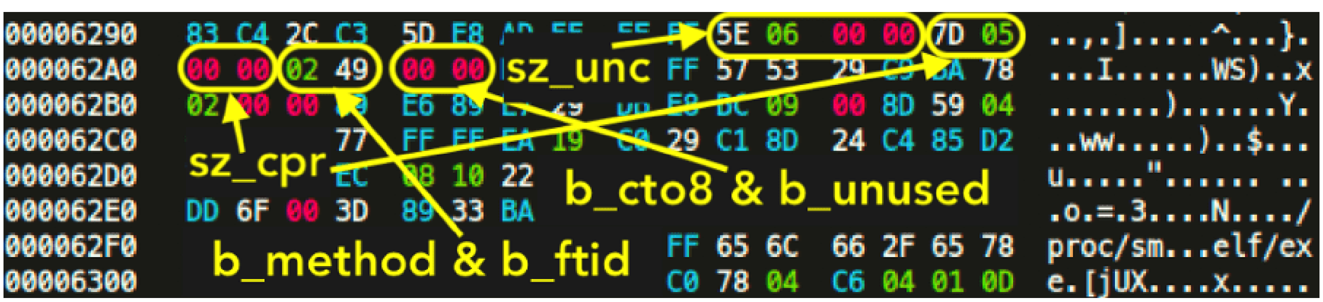
- https://github.com/upx/upx
- https://github.com/radareorg/radare2
- https://github.com/upx/upx/issues/389

- https://github.com/upx/upx/blob/master/src/stub/src/i386-linux.elf-entry.S
- https://github.com/upx/upx/blob/master/src/stub/src/amd64-linux.elf-entry.S

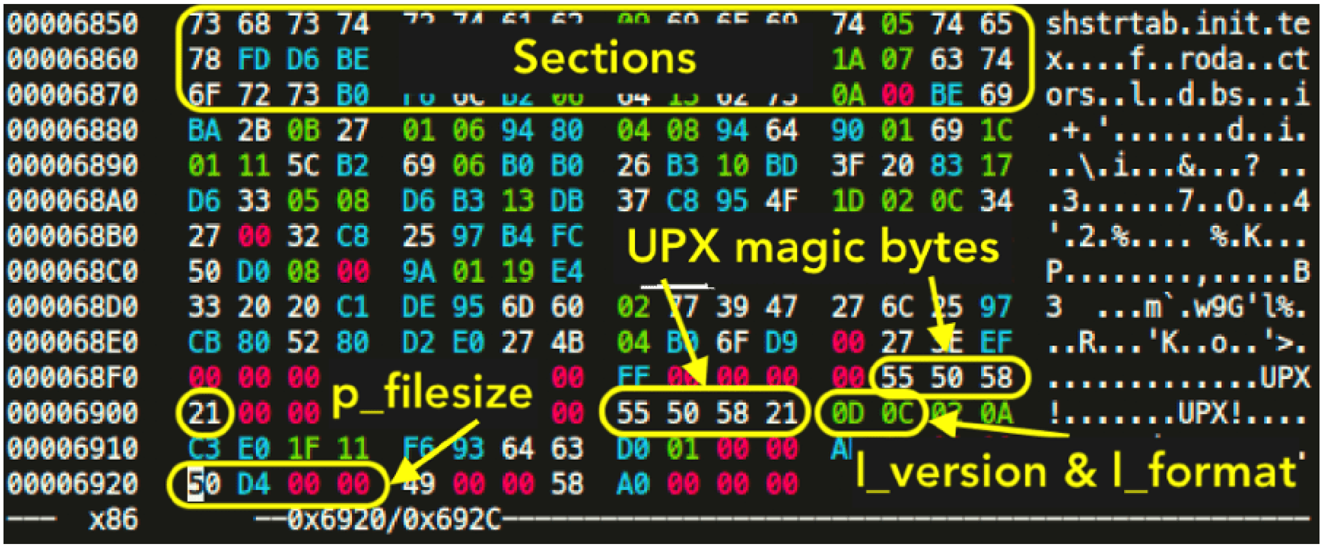
Appendix



Appendix A



Appendix B



Appendix C

Used malware hash for analysis:

bc88a57e1203f5eec08d34b59d9de43fa121f9d92cc773c17ebfbe848a2f88cd

Special thanks to @unixfreaxjp for his previous research on **ELF** packing.