Using UPX as a Security Packer

Dr Petri - drrpetri@gmail.com

March 21, 2012

Keywords: Packing, Obfuscation, Win32, Malware, Detection, Polymorphism, UPX, Stealth

Abstract

UPX is one of the most famous packer, and maybe the most commonly used around the world. The aim of this packer is to compress executables, in order to save space on your hard drive.

I will show a basic approach of the packing mechanism in UPX, then how to patch this awesome tool to bring obfuscation in it.

There is many advantages to modify an existing packer:

- Many security tools will detect a harmless UPX packer (Rather than Themida, Yoda...).
- You can develop your own packer without coding a painful PE parser: focus on what is important.
- Automated unpacker may encounter some problem to extract the data
- Spreading patched packers on the net will make automatic extraction more difficult

Would you like to know more?

Contents

1 How does UPX works

The UPX compressed file is juste a PE file slightly smaller than the original file on the disk, and bigger in memory.

The compressed file contains:

- ➢ An executable stub to decompress the data
- ➢ A compressed copy of the original file
- ➢ A empty memory space to put the original uncompressed data in

1.1 Hello World

The executable I will use in part 1 & 2 is the following:

hello.asm:

```
[bits 32]

EXTERN _MessageBoxA@16
GLOBAL _WinMain@16

section .text
        _WinMain@16:
                push 0h
                push title
                push msg
                push 0h
                call _MessageBoxA@16   ;MessageBox(0, "hello", "title", 0);
                ret
section .data
        title   db 'title', 0h
        msg     db 'hello', 0h
section .fill
        times 1000 db 0x90
```

Figure 1: Hello World Source Code

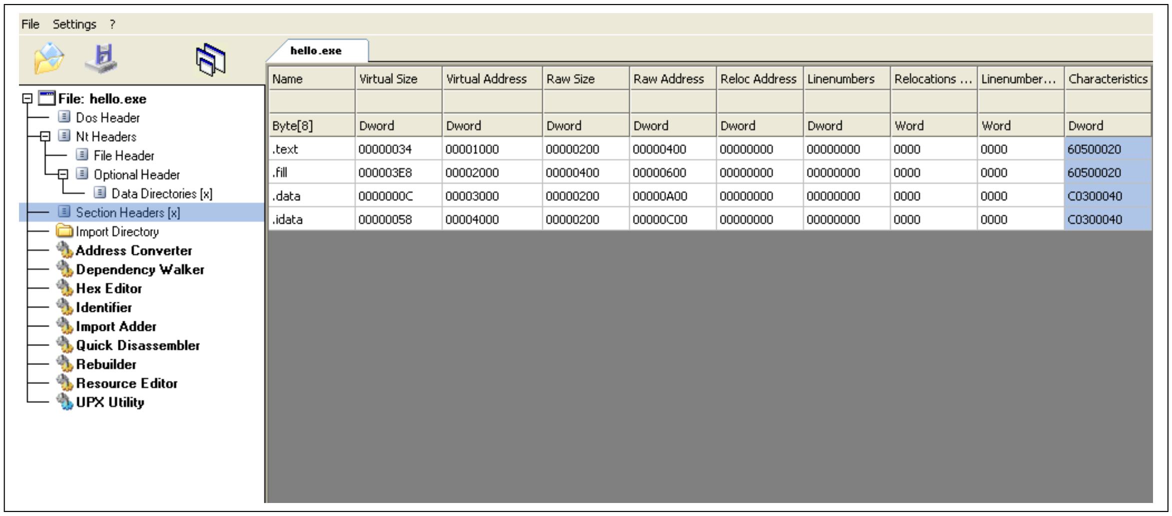
Let's compile to a win32 executable:

```
nasm -o hello.o hello.asm -fwin32
i686-pc-mingw32-gcc -o hello.exe hello.o -nostartfiles
strip -s hello.exe
```

Figure 2: Hello World Compilation Command

1.2 Sections

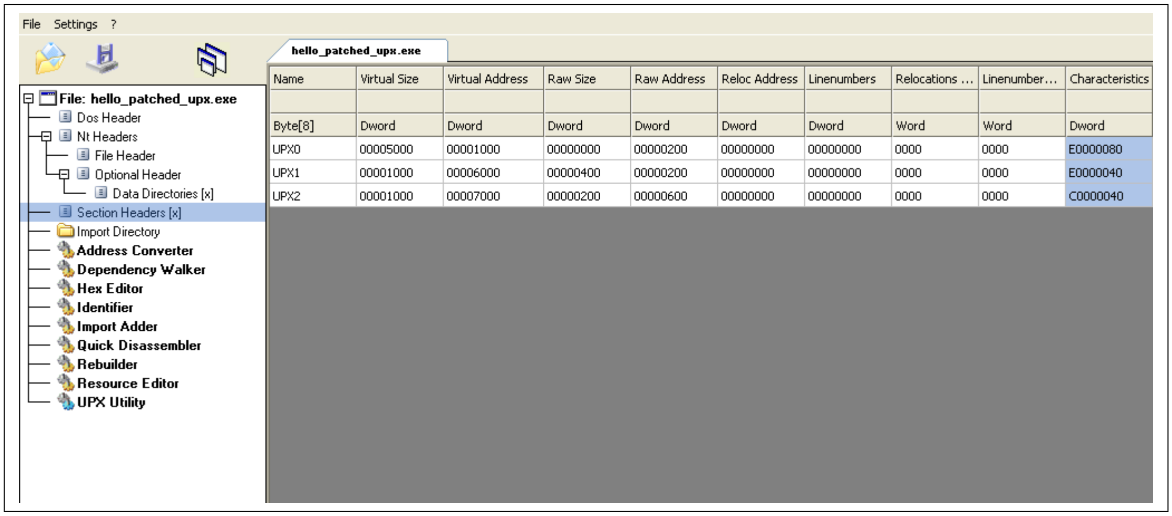
Here is the overview of the hello world sections in CFF Explorer.



Name	Virtual Size	Virtual Address	Raw Size	Raw Address	Reloc Address	Linenumbers	Relocations ...	Linenumber...	Characteristics
Byte[8]	Dword	Dword	Dword	Dword	Dword	Dword	Word	Word	Dword
.text	00000034	00001000	00000200	00000400	00000000	00000000	0000	0000	60500020
.fill	000003E8	00002000	00000400	00000600	00000000	00000000	0000	0000	60500020
.data	0000000C	00003000	00000200	00000A00	00000000	00000000	0000	0000	C0300040
.idata	00000058	00004000	00000200	00000C00	00000000	00000000	0000	0000	C0300040

Figure 3: Hello World - PIMAGE_SECTION_HEADER

The same program, packed by UPX:



Name	Virtual Size	Virtual Address	Raw Size	Raw Address	Reloc Address	Linenumbers	Relocations ...	Linenumber...	Characteristics
Byte[8]	Dword	Dword	Dword	Dword	Dword	Dword	Word	Word	Dword
UPX0	00005000	00001000	00000000	00000200	00000000	00000000	0000	0000	E0000080
UPX1	00001000	00006000	00000400	00000200	00000000	00000000	0000	0000	E0000040
UPX2	00001000	00007000	00000200	00000600	00000000	00000000	0000	0000	C0000040

Figure 4: Packed Hello World - PIMAGE_SECTION_HEADER

1.3 UPX0

This is an empty section. It actually contains the memory from 0x00401000 to 0x00406000, this memory area has the same size than the unpacked file's adresses area.

1.4 UPX1

This section is the entrypoint section, it contains the stub and all the compressed executable.

1.5 UPX2

This sections contains the imports only. Note that there is always:

- kernel32.dll:LoadLibraryA, kernel32.dll:GetProcAddress to load the original executable libraries
- kernel32.dll:VirtualAlloc, kernel32.dll:VirtualProtect Used to restore original sections rights

1.6 Stub

The stub of the compressed executable has to do exactly the same work than the kernel when an executable is launched. The packed executable's entrypoint points to this stub.

An overview of the sub's tasks:

- Memory allocation
- Extract the original file in section UPX0
- Resolve all the imports of the original executable using kernel32.dll:LoadLibraryA and kernel32.dll:GetProcAddress
- Restore relocations
- Call kernel32.dll:VirtualProtect to restore the original permissions
- Jump to the original entrypoint

2 POC: Patch an executable

Let's fail the UPX unpacker!
The example executable is the same hello world as the one used on part 1.1.

2.1 Executable patching

I will now patch the entry point of the executable: the file offset is 0x400, the memory offset is: 0x401000.
Let's replace the value 0x68 by 0x00 (I used HT Disassembler).

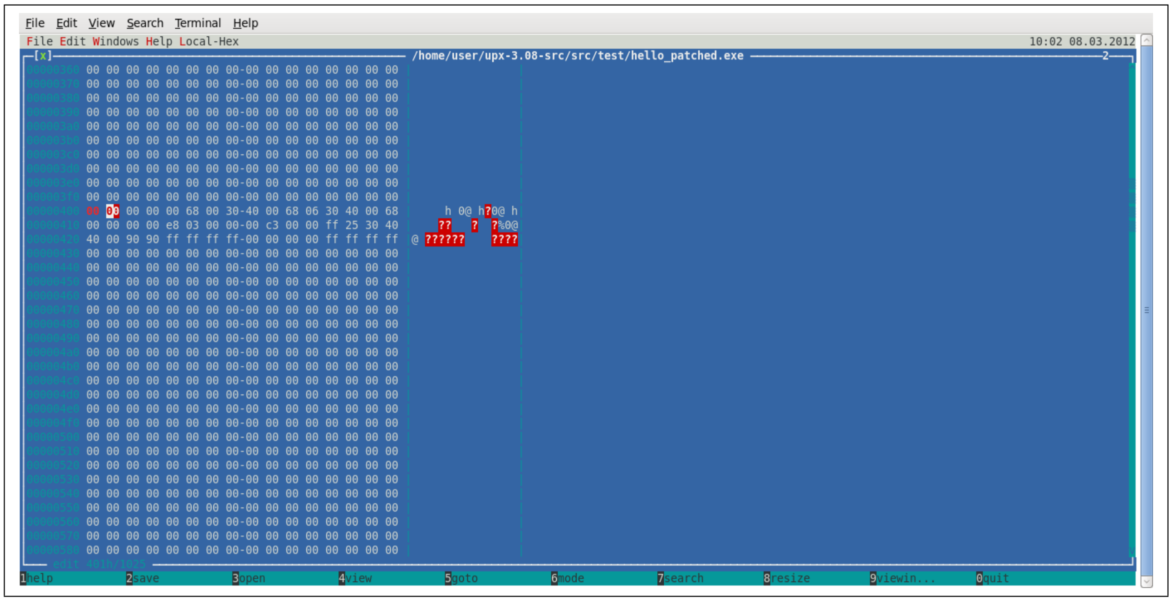


Figure 5: Hello World Hexdump

A brief overview of the instructions near the hello world entry point:
The first instruction will create a segmentation fault: perfect!

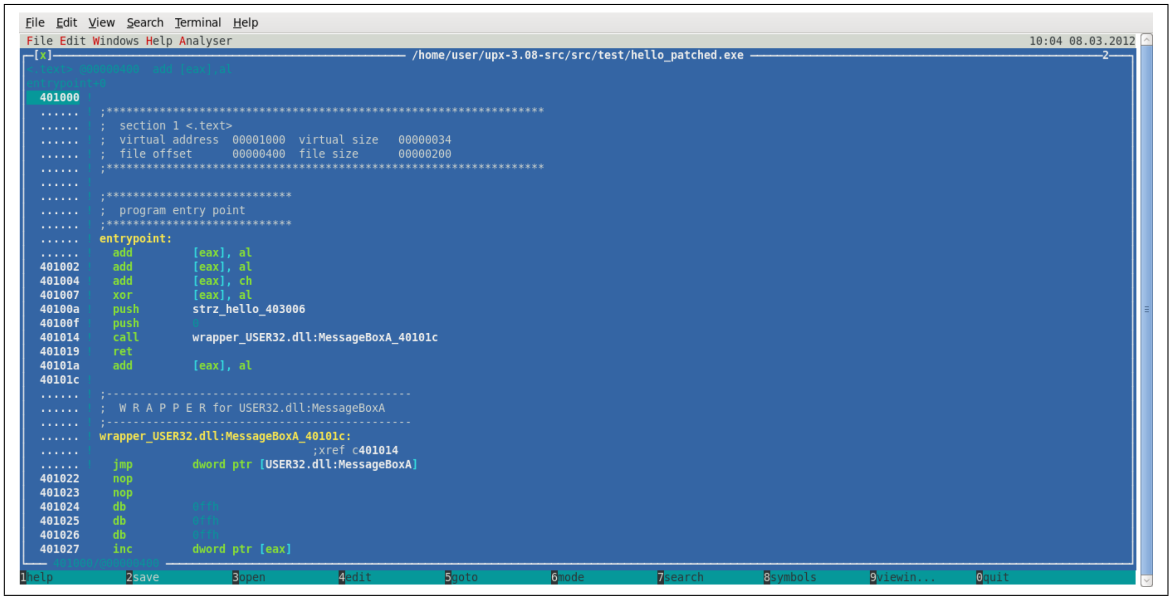


Figure 6: Hello World Image section ".text"

2.2 UPX Patching

In order to make this program run correctly, the UPX stub has to restore the value 0x68 at the address 0x401000.
Open the file src/stub/src/i386-win32.pe.S.
Apply the patch after the file has been restored in UPX0, and before VirtualProtect sets the section not writable.

1 line only is needed, adding code before the imports is OK:

```
// =============
// ============= IMPORTS
// =============
section PEIMPORT
    //ADD THIS LINE
                mov BYTE PTR [0x401000], 0x68
                lea     edi, [esi + compressed_imports]
next_dll:
                mov     eax, [edi]
                or      eax, eax
                jz      imports_done     //jmps to jmp
                mov     ebx, [edi+4]     // iat
```

Figure 7: Original byte restoration

Note that this code will be compiled using GCC from assembly with intel syntax.

2.3 Test

Let's compile the stub, then UPX itself (type "make" in src/stub, then make in src/).

The stub Makefile is quite painful to use, you can copy the makefile:
Pack the hello world we just modified, then compress and execute it:

Figure 8: IT WORKS!

- ✓ PEID detects the file as an UPX file (No difference with original UPX)
- ✓ UPX can decompress it without any warning
- ✓ Extracting the file with "UPX -d" will create a corrupted file

When executing the extracted file, an error occur at the address is 0x401000: the address of the byte we set to 0.

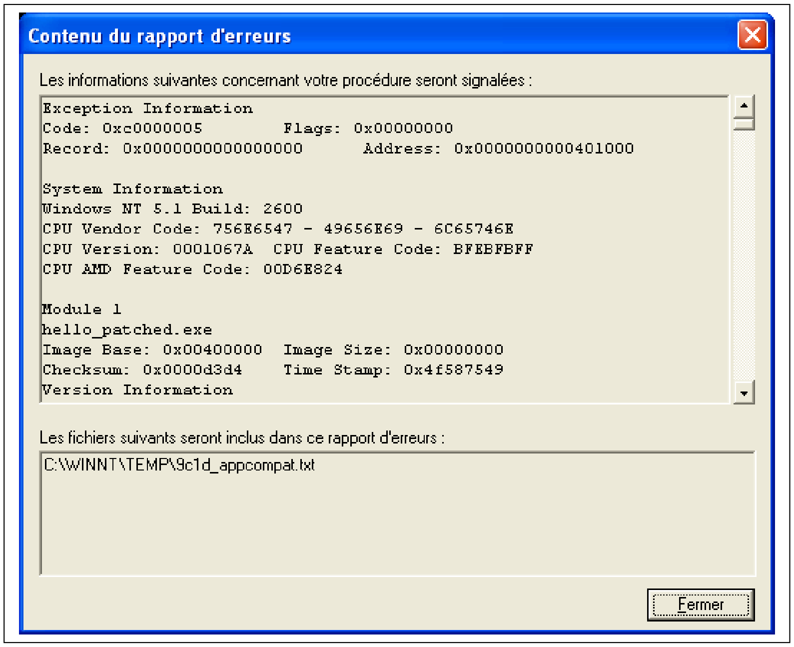


Figure 9: Segmentation fault output

Let's go to the next step: automatize the binary patch.

3 Build an UPX release

Let's patch UPX to automatically modify the original executable. I will use a simple xor encryption with a random 1byte key. This is not the most secure algorithm, but it is enough to slow down analysis, and even bust many automated unpacker, and add polymorphism. Feel free to improve it.
I will use calc.exe this time.

3.1 Upx compactor

In order to automatize the binary file patching, we have to edit the win32 packer source code file: src/p_w32pe.cpp

Add the following code in the function void PackW32Pe::pack(OutputFile *fo); Around line 1040. You will see the function compressWithFilters, just as the exemple below:

```
//*******************************************
//* PATCH - must be just before compressWithFilters();
//*******************************************
#define UNINITIALIZED      0x16  //flag in section Characteristics
#define MAX_SECTIONS    4   //the maximum number of sections allowed to encrypt
srand(time(NULL));
unsigned char key;

struct encrypted_t{ //this struct will store the informations for the stub
    unsigned int start_decrypt;
    unsigned int end_decrypt;
    char key_val;
}encrypted[MAX_SECTIONS];

unsigned int encrypted_sections=0;

for(unsigned int i=0; i<ih.objects; i++){        //run through all the sections
    //Do not encrypt uninitialized functions, and not more than MAX_SECTIONS
    if((isection[i].flags & UNINITIALIZED) == 0 && encrypted_sections < MAX_SECTIONS){
cout << "Encrypted section:" << isection[i].name << endl;    //print the section name
        key=rand()%254;               //generate a random key.
        //encrypt data
        for(unsigned int j=isection[i].vaddr; j<isection[i].vaddr+isection[i].vsize; j++)       //encrypt the whole section
            ibuf[j]^=key;    //xor encryption
        //store stub informations
        encrypted[encrypted_sections].start_decrypt = isection[i].vaddr + ih.imagebase;
        encrypted[encrypted_sections].end_decrypt = isection[i].vaddr + ih.imagebase + isection[i].vsize;
        encrypted[encrypted_sections].key_val = key;
        encrypted_sections++;
    }
}
//*******************************************
//* END OF PATCH
//*******************************************
    compressWithFilters(&ft, 2048, NULL_cconf, filter_strategy,
                        ih.codebase, rvamin, 0, NULL, 0);
// info: see buildLoader()
//*******************************************
//* PATCH - must be just after compressWithFilters();
//*******************************************
    char label[20];
    for(unsigned int i=0; i<MAX_SECTIONS; i++){    //Define the symbols for the stub
        snprintf(label, sizeof(label), "start_decrypt%i", i);
        linker->defineSymbol(label, encrypted[i].start_decrypt);
        snprintf(label, sizeof(label), "end_decrypt%i", i);
        linker->defineSymbol(label, encrypted[i].end_decrypt);
        snprintf(label, sizeof(label), "key_val%i", i);
        linker->defineSymbol(label,  encrypted[i].key_val);
     }
    linker->defineSymbol("section_nb", encrypted_sections);
//*******************************************
//* END OF PATCH
//*******************************************
```

Figure 10: src/p_w32pe.cpp

3.2 Stub

Now add the stub patch. Once compiled, the stub will be 147 bytes bigger. For each encrypted section, the values of the key, start end end of encrypted data are retrieved. The symbols in the stubs are those we defined in "p_w32pe.cpp" with the function linker->defineSymbol();.

```
// =============
// ============= IMPORTS
// =============
section PEIMPORT
/////////PATCH XOR
lea eax, section_nb
cmp eax, 0
je end_patch

lea ecx, end_decrypt0
lea edi, start_decrypt0
lea ebx, key_val0
call decrypt_xor

lea eax, section_nb
cmp eax, 1
je end_patch

lea ecx, end_decrypt1
lea edi, start_decrypt1
lea ebx, key_val1
call decrypt_xor

lea eax, section_nb
cmp eax, 2
je end_patch

lea ecx, end_decrypt2
lea edi, start_decrypt2
lea ebx, key_val2
call decrypt_xor
lea eax, section_nb
cmp eax, 3
je end_patch

lea ecx, end_decrypt3
lea edi, start_decrypt3
lea ebx, key_val3
call decrypt_xor

jmp end_patch

decrypt_xor:
    label1:
        xor al, al
        mov al, [edi]
        xor al, bl
        mov [edi], al
        inc edi
        cmp edi, ecx
        jne label1
    ret
end_patch:
/////////END PATCH

                lea     edi, [esi + compressed_imports]
next_dll:
                mov     eax, [edi]
```


Figure 11: src/stub/src/i386-win32.pe.S

4 Analysis

4.1 Tools

Despite the stub suffer from modification, the new packed executable sustain those properties:

- ✓ PEID does detect a UPX compacted executable.
- ✓ "UPX -d" extracts the binary without any warning.

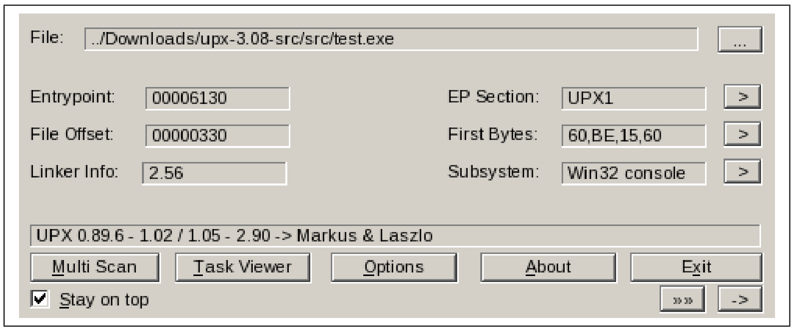


Figure 12: PEID output

Next step is to test an actual malware on anti viruses. I packed a malware with my patched UPX, And submited it on the website. The result is not perfect but not so bad for 1 hour of coding: we drop down from 95% detection to 23%.

Note that only one anti virus actually detected the same malware, I assume it uses a sandbox; feel free to add anti debugging tricks in the stub.

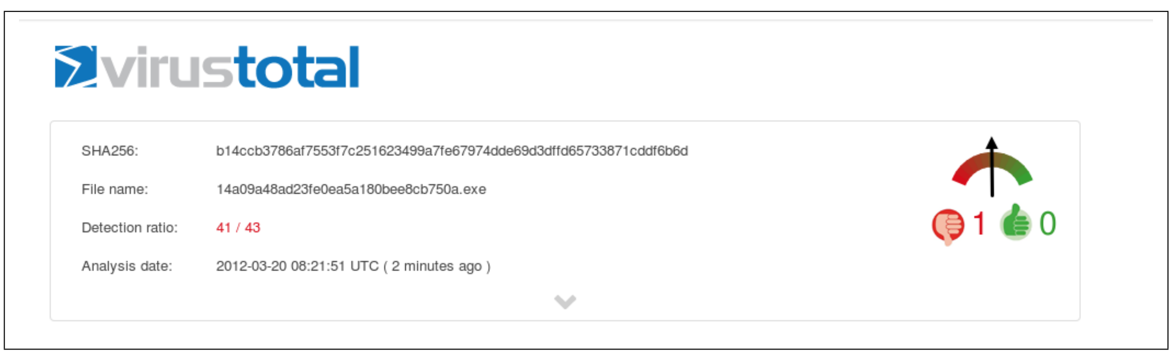


Figure 13: Virus total output of the malware

Figure 14: Virus Total output of the packed malware

Figure 15: Avast analysis of the malware

Figure 16: Avast analysis of the packed malware

4.2 Polymorphism

4.2.1 Packed executables

This is the binary diff of 2 packed executables of the same calc. The differences between the 2 files are due to the random key.

The following picture shows the differences between those files:

- On abscissa axe, there is the file offset in bytes.
- A red bar means the byte in the compressed files are the same at the current offset.

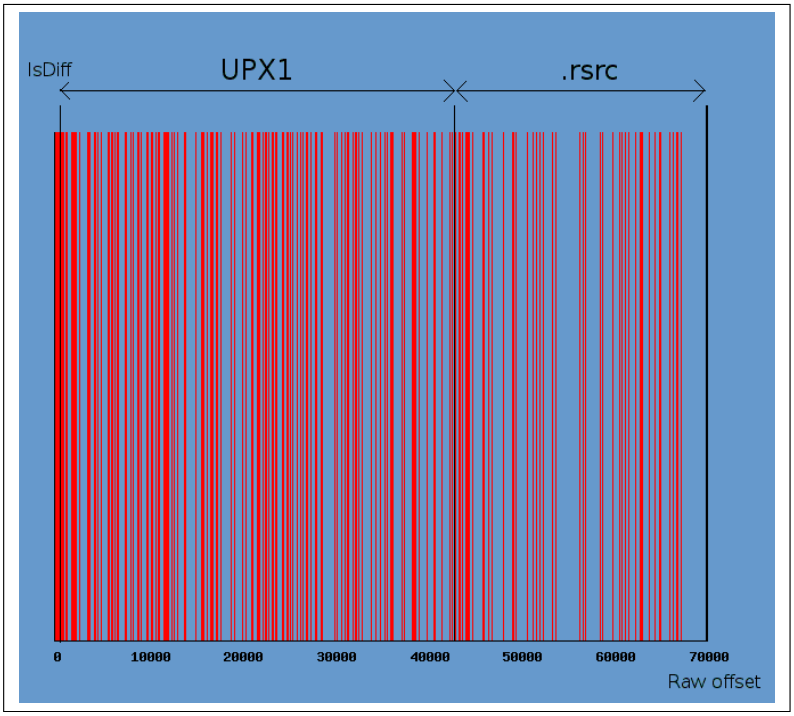


Figure 17: Binary diff of packed files

Observations:

✓ The PE Header is the same.

✓ The stub is the same too.

✓ Sections UPX1 and .rsrc are nicely randomly scrambled.

4.2.2 Exctracted executables

By extracting the last 2 files with UPX, I obtained 2 different files as well. Both of them are not valid.

The following picture shows the differences between those 2 files:

➢ On abscissa axe, there is the file offset in bytes.

➢ A red bar means the byte in the extracted files are the same at the current offset.

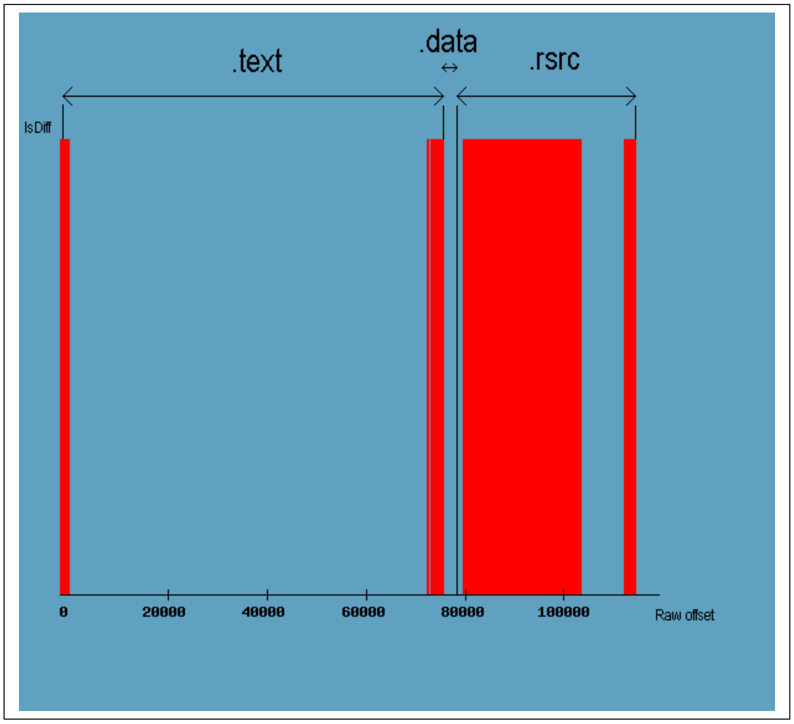


Figure 18: Binary diff of extracted files

Observations:

- ✓ The PE Header is the same (we did not encrypt them at all).
- ✓ The section imports have been restored in the ".text" section, but the code is entirely obfuscated.
- ✓ The section ".data" is completely obfuscated.
- ✓ The resources have been partially restored. It is still corrupted.

5 Bibliography

UPX Official website:
`http://UPX.sourceforge.net`
MSDN Microsoft PE file format documentation:
`http://msdn.microsoft.com/en-us/magazine/cc301805.aspx`
Nasm official website:
`http://www.nasm.us`
CFF explorer tool: Explore a PE executable:
`http://www.ntcore.com/exsuite.php`
HT disassembler one of my favorites:
`http://hte.sourceforge.net`
PEID a tool to identify packer:
`http://www.peid.info`
Themida a well known security packer:
`http://www.oreans.com/themida.php`
Another security packer:
`https://sourceforge.net/projects/yodap`

6 Annexes

The full patched UPX attachement: