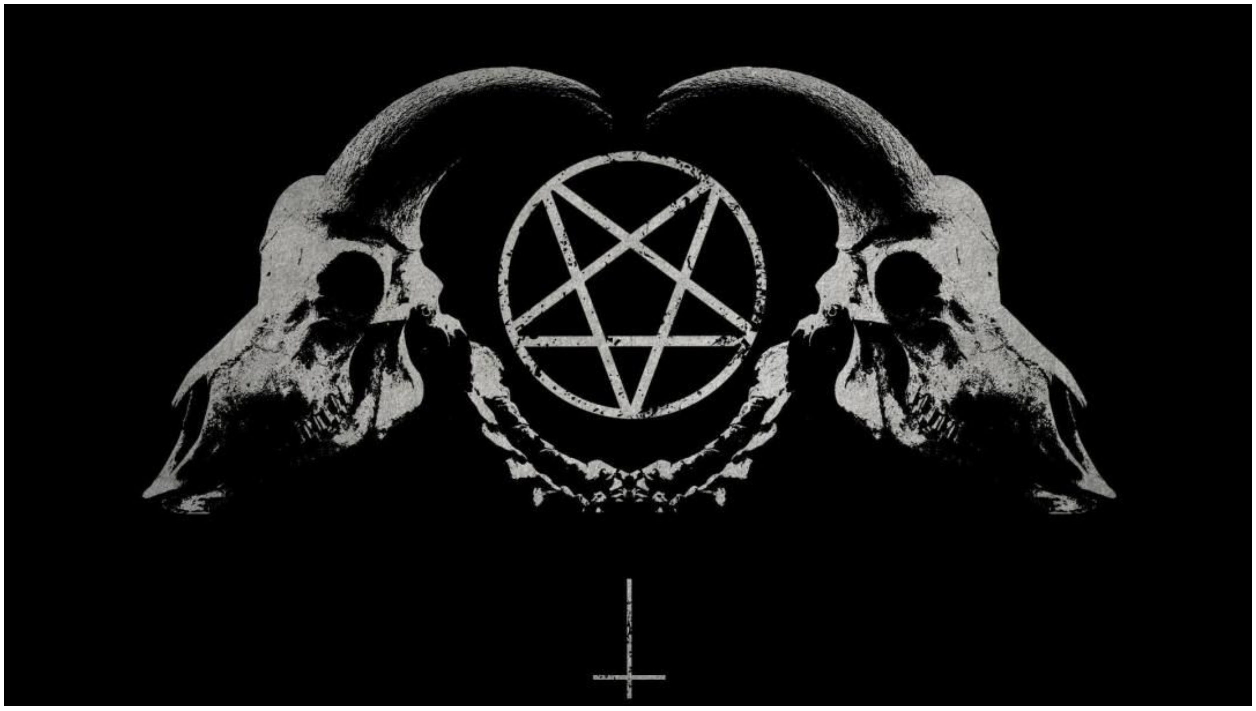
AUTOMATED REVERSE ENGINEERING: MISTFALL ENGINE



(x) 2000 Z0MBiE
xlated from russian in 2001

Our efforts are directed to develop such a method of executable program modification, that finding changes will require a maximal amount of time. Modification means addition of the viral code to some specified program, given in the PE format. It is obvious that the main viral body should be encrypted, and metamorphic (generated) virus decryptor should be integrated with the program's code.

Hence, our task is divided into 3 parts, or 3 global questions: WHAT?, WHERE? and HOW?

WHAT? - is a question about instructions, those will be inserted into a modifying program. It is explained in the article about metamorphism and shown in the CODEGEN metamorphic engine.

WHERE? - is a question of finding places in the modifying program, our instructions should be inserted into. This is a simple task; the engine's caller will resolve it in different ways. This article shows HOW to insert our own instruction between two arbitrary instructions of the modifying program. In other words, how to decompile, modify and compile the whole program.

THEORY

So, the task is to insert my own instructions between PE file's ones. But, because instructions are linked between each other, changing instructions involves change of links. But change of links involves change of other instructions, and so on, until a significant part of code will be changed.

For example: when inserting some instruction into a block of code, offsets of all the instructions coming after insertion are increased. This means that mostly all offsets - both relative and absolute, should be fixed. While fixing relative offsets, some jxx can grow, that means that all this shit should be repeated from the start.

As you can see, there are two kinds of offsets: absolute (offsets tself) and relative (jmp/call arguments). And we should find all these offsets. Absolute offsets may be found by means of analyzing PE structure, including fixup table. Finding relative offsets requires partial disassembly. Hence, we must disassemble file into some easy-modifiable substance, change this substance, and assemble the file back.

Algorithms will be formalized as an universal engine, working with PE files.

So, task is divided into 5 general steps:

1. Load PE file into virtual addresses; allocate flag table (DWORD for each byte), and initialize it with special PE structure - related bits; in other words, perform initial PE structure analysis.
2. Disassemble file (instruction by instruction), at the same time filling the flag table with new information about instruction offsets. At this step we can not make a mistake, i.e. to recognize code as data or vise versa, because such fault is fatal; so, dual situations should be excluded, and some files may not be processed at all.
3. Convert all known information into a list of: instructions, datablocks, labels and pointers. I.e. to make our information a bit more high-level, a bit nearer to source. Such a list is an easy-modifiable substance we were speaking about; it is generated because of easy manipulation with its elements.
4. Call (external relative to engine), which will modify the instruction list; for example insert additional instructions or remove existing ones.
5. Assemble file back: recalculate all offsets; fix PE structure; rebuild fixup table and so on.

DISASSEMBLING PROBLEMS

Main problem, of course, is disassembling. We have no such resources, as IDA, for example, - neither memory, neither time, nor signatures; moreover, our virus is limited to some kilobytes. And the main disassembling problem is duality of some labels: they can precede both code and data. I.e. If we have some data fixup that points to some label, and there is no code instruction pointing to the same label, then we can not correctly decide if this label precedes code or data. Such mis recognizing causes the following: code, interpreted as data, will not be fixed, and will fail the program when executed, commonly on such instruction as jmp, jxx or call. Vise versa, incorrectly fixed data (i.e. data interpreted as code) will with high probability cause fault too. This means that we must always correctly distinguish code and data. Some library functions may be found by signatures, but we can not insert signature base into the virus; also, jmp table-analysis can help, but only partially.

We can work with only special easy-decompilable files, containing only code in the CODE section, but such files are a real rarity. So, we will reduce the number of processing files (exclude packed ones and etc.), improve our disassembler as much as possible, and hope for luck.

IMPLEMENTATION

Really, there are much more little steps we should perform; and each step has an influence on the result. Here is a complete list of steps required for PE file reversing:

1. Check if the given PE file is valid; if the fixup table is present & etc.
2. Allocate memory for virtual program image and for flag table
3. Load into virtual addresses:
    a. Dos stub & PE header
    b. Sections
4. Analyze PE header; process pointers; mark physical / virtual start/end of each section & etc.
5. Process imports
6. Process exports
7. Process fixups
8. Process resources
9. Search for signatures (such as push ebp/mov ebp, esp) and mark 'em as for-next-analysis
10. Mark entry point
11. Disassemble file, instruction by instruction.
12. Algorithm is described in the article about permutation, but there are some differences: when there are no more bytes marked as for-next-analysis, we must search for labels-pointed-by-data-fixups, and check if they precedes code instructions. (see below)
    a. Build list of opcodes, labels, pointers & etc.
    b. Note, that such list will contain zero-length pointers (labels), i.e.transfer is performed to a higher level of abstraction, nearer to source.After converting linear arrays into list, linear arrays (memory & flag table) may be deallocated.
13. Modify list; while debugging, I was inserting NOPs between instructions.
14. Recalculate new virtual/physical addresses for each list entry
15. Recalculate fixup table
16. Recalculate pointers (i.e. rva's, fixups & relative arguments of the conditional jmps); if some jmps are grown,repeat from recalculating virtual addresses.
    a. Note, that here possible some iterations, it is normal.
17. Assemble list (collect all stuff into single file)
18. Write file to disk
19. Copy overlay, if present
20. Recalculate checksum, if non-zero

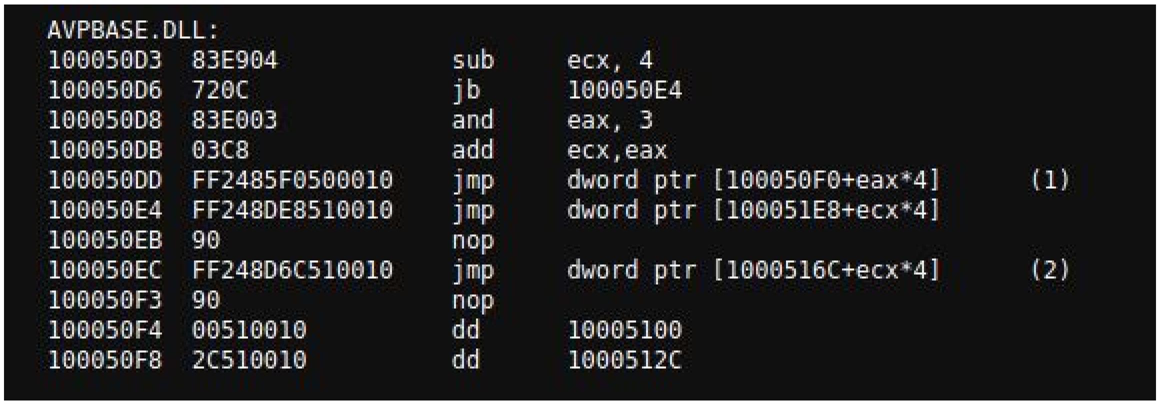
Phrases, such as "analyzing PE header" or "process imports" means that we analyze these structures and marking labels, pointers and other special objects in the flag table with special bits. For example byte at the pe_header+28h (28h=EntryPointRva) will be marked as FLAG_DWORD | FLAG_RVA, and byte it points to will be marked as FLAG_LABEL | FLAG_CREF.

Which special objects will be present in the instruction list?

1. Label, i.e. zero-length substance pointed by RVAs and FIXUPs.
2. RVA, or DWORD pointing to some label.
3. FIXUP, or DWORD, same as RVA, but increased by IMAGEBASE, requires to be inserted into the fixup table on the final steps.
4. So-called DELTA, or difference between virtual addresses of two labels.
5. instruction
6. Data Block

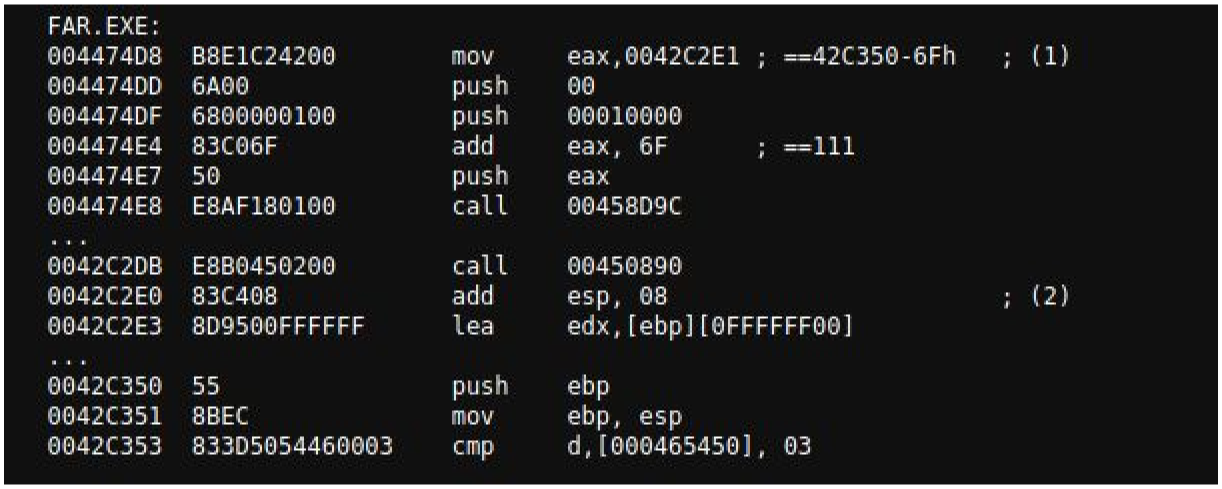
Now, about distinguishing code and data when only the existing reference is data-reference. Algorithm: take instruction by instruction; check if it is not 00 00, FF FF, F4 (hlt), CD (int), and so on -- i.e. fail if such instruction is never present in the standard PE file. Also fail, if one of the middle bytes of the instruction is marked as label, data, or other special object. Also, if the instruction is jmp,call,jxx,jecxz and so on, then it should not point into the middle of other instructions or to data. Analysis is repeated, until marked-as-code instruction, RET or JMP is found.

But, there are other kinds of dual situations. You can think that such an object as a label can not exist in the middle of some instruction, generated by the HLL compiler. But it can. Let's examine typical situation:


```
AVPBASE.DLL:
100050D3  83E904           sub     ecx, 4
100050D6  720C             jb      100050E4
100050D8  83E003           and     eax, 3
100050DB  03C8             add     ecx,eax
100050DD  FF2485F0500010   jmp     dword ptr [100050F0+eax*4]      (1)
100050E4  FF248DE8510010   jmp     dword ptr [100051E8+ecx*4]
100050EB  90               nop
100050EC  FF248D6C510010   jmp     dword ptr [1000516C+ecx*4]      (2)
100050F3  90               nop
100050F4  00510010         dd      10005100
100050F8  2C510010         dd      1000512C
```

As you can see, 100050F0-address, that is XREF-ed by (1), really is part of the instruction (2). It's not so hard to understand why it is so. As a result, we can not decide, if (2) is code or data. I.e. it is impossible to determine if (10050F0 == 100050F4 - 4) or (10050F0 == 100050EC + 4). So, pointer at (1) can not be fixed and file can not be processed.

Or, another one example:



```
FAR.EXE:
004474D8  B8E1C24200        mov     eax,0042C2E1 ; ==42C350-6Fh   ; (1)
004474DD  6A00              push    00
004474DF  6800000100        push    00010000
004474E4  83C06F            add     eax, 6F      ; ==111
004474E7  50                push    eax
004474E8  E8AF180100        call    00458D9C
...
0042C2DB  E8B0450200        call    00450890
0042C2E0  83C408            add     esp, 08                       ; (2)
0042C2E3  8D9500FFFFFF      lea     edx,[ebp][0FFFFFF00]
...
0042C350  55                push    ebp
0042C351  8BEC              mov     ebp, esp
0042C353  833D5054460003    cmp     d,[000465450], 03
```

And, in addition, another bad thing: parts of 16-bit code, inserted into 32-bit applications, such as antiviruses, formatters and so on.

The ENGINE

Though, it works. Engine is called MISTFALL, there is G.Martin's story named "With morning comes Mistfall".

Engine is written in Borland C++, but without classes or other shit. Main engine() subroutine with fucking lots of parameters comes first, followed by some internal additional subroutines. All this stuff is called kernel, and it is located in the engine.cpp & .hpp; code and constants correspondingly. Kernel calls a so-called user's mutator (mutate.cpp), which must be written by the engine's user. Mutator works only with an instruction list. List entries describe labels, opcodes, data blocks and so on. Moreover, all the engine interface technology is taken from the RPME engine.
The only difference is that RPME works with blocks of viral code, and MISTFALL works with PE files.

As a result, simplest file infection is the following:

1. Find any two non-jmp instructions; insert JMP after first instruction to the second one; insert encrypted viral body after JMP.
2. The same for decryptor.
3. The same for commands calling the decryptor.

This was only one of lots of possible variants; the more such methods we will use, the harder it will be to analyze the virus

USAGE

Engine uses lots of memory. It can fail on an incorrect file; and on a normal one too. Memory allocation, same as in RPME, is provided by the engine's caller. Before calling the engine, you must allocate about 32 MB of memory, and write your own malloc() subroutine which will provide engine pointers to little parts of this big block. It is just a kind of heap. Engine's allocation strategy is specific: it will only allocate memory and never deallocate. But, it's ok because the same big memory block will be used for each file. Really, engine uses (17*SizeOfImage) bytes of memory for linear arrays and 40 bytes of memory for each list entry. Engine's code is permutable, i.e. it satisfies most of the demands described in the "Demands to engines" article. Engine can work in ring-0, but, because of lots of required memory & time, don't call it there. Time used by the engine is unpredictable; minimal time for processing one file is some minutes in realtime priority.

Obviously, that such engine cant be invoked on system events, such as openfile. You should work with file queue, passing file by file to the engine in the individual thread or process.

SPECIAL FEATURES

Engine works only with standard files. I.e. all the section names must be known: such as .text, .data and so on. Otherwise the file is packed or whatever, and will not be processed.

For all the standard files it is accepted that code exists only in the first section; this allows us to increase disassembly quality.

SIGNATURE LIBRARY

This is another external subroutine, performing 2 tasks:

1. On start: search for known codeblocks, and
2. Add existing code blocks into library

I.e. before disassembling the signature manager will mark known codeblocks as CODE, for-next-analysis, and, on exit, it will update the signature library with new information.

When applied to viruses, a signature library will be created per each local machine, performing a kind of self-education, with each new file. Signature here means some constant code bytes, not containing fixups and relative offsets.

ENGINE INTERFACE

```
// return: 0==success, non-zero error codes listed in ENGINE.HPP
int __cdecl engine(
            DWORD  user_arg,              // user's parameter (whatever)
            BYTE*  buf,                   // PE file
            DWORD  inbufsize,             // initial size of file
            DWORD* outbufsize, // ptr to resulting size of file
            DWORD  maxbufsize,            // maximal allocated buf size
            int    __cdecl user_disasm(DWORD,BYTE*), // length-disassembler
            void*   __cdecl user_malloc(DWORD,DWORD), // memory allocator
            DWORD   __cdecl user_random(DWORD,DWORD), // randomer
            int    __cdecl user_mutate(                 // mutator
            DWORD user_arg,                              // (described below)
            PE_HEADER*    pe,
            PE_OBJENTRY* oe,
            hooy*        root,
            int    __cdecl (*)(DWORD user_arg,BYTE*),
            void* __cdecl (*)(DWORD user_arg,DWORD),
            DWORD __cdecl (*)(DWORD user_arg,DWORD)
            ),
```

```
int    __cdecl user_sigman(    // signature manager or NULL
            DWORD        user_arg,
            PE_HEADER*    pe,
            PE_OBJENTRY* oe,
            BYTE*        memb,
            DWORD*       flag,
            DWORD        action)     // 0/1
            ); //engine
```

```
int __cdecl mutate( //mutator
            DWORD                user_arg,    // user's parameter
            PE_HEADER*           pe,          // pe header
            PE_OBJENTRY*         oe,          // pe objtable
            hooy*                root,        // list root
            int    __cdecl  user_disasm(DWORD user_arg,BYTE*),
            void*   __cdecl  user_malloc(DWORD user_arg,DWORD),
            DWORD   __cdecl  user_random(DWORD user_arg,DWORD)
            ); //mutate
```

```
int __cdecl sigman( //signature manager
             DWORD        user_arg,     // user parameter
             PE_HEADER*   pe,           // pe header
             PE_OBJENTRY* oe,           // pe objtable
             BYTE*        memb,         // virtual memory block to work with
             DWORD*       flag,         // flag table
             DWORD        action        // 0==before(mark), 1==after(update)
             ); //sigman
```

This shit shown above... of course it doesn't mean that you should program in C++ to use an engine. Engine may be called both from asm & cpp.

EXAMPLES

Mistfall engine was used in the Z-10 virus, all sources are published in the TZ #1 e-zine.