x86.Virtualizer – source code

author: ReWolf
date: IV/V.2007
rel.date: VIII.2007

e-mail: rewolf@rewolf.pl
www: http://rewolf.pl

Table of contents:

1. Usage:

- Put \bin\loader\meta.exe and \bin\protector\x86.virt.exe in one folder

- '...' <- select executable to protect (requires **VirtualAlloc** in Imports Table)

- **'VM over VM'** <- enable double VM layer (not recommended, this option is not fully tested and contain at least one serious bug, I don't know where ;])

- **'Add range'** <- add region to protect, for example:

```
00403E8E  /$ 6A 00          PUSH    0
00403E90  |. E8 96FFFFFF    CALL    x86_virt.00403E2B
00403E95  |. 59             POP     ECX
00403E96  \. C3             RETN
```

  Set:
     'From' : 403e8e
     'To'   : 403e97   <- notice: it's one byte after RETN offset

  You can add whole function as well as block of code (inside function), minimal range size is 5 bytes (long jump)

- **'Protect'** <- protect executable, it will add suffix _vmed to original executable filename

- **'Exit'** <- guess...

2. Compilation

Add all files from src directory to project.
Set these options in Visual Studio:
(tested on **VS2k5**)

General:
   Character Set: Not Set

3. Source code documentation

● Loader:

macros:

```
* storeESP            -\
* storeRealESP        - \ auxiliary macros to switch between virtual
* reStoreRealESP      - / machine stack and normal stack
```

```
  * reStoreESP         -/


  * setVar          -\
  * getVar          - >  macros provided to access some vm variables (I'm
  * getVarAddr      -/   not using 'delta' addressing to access it)

  * SHIFTS          -\   instructions definition macros
  * IMM32           -/
```

functions:

```
  * _vm_jump  - conditional jumps dispatcher, takes in edx condition
                number and return in edx 1 if jump is taken or 0 if not.
                Condition numbers are generated during virtualization
                process, look at permutateJcc() function in common.cpp

  * _vm_call  - takes as argument (esp+4) address of function to call.
                Called function can be normal native code, or another
                virtualized function.

  * poly      - polymorphic decrypter, generated by genPolyEncDec()
                (common.cpp)

  * _vm_init  - virtual machine initialization takes two arguments:
                pointer to vm stack buffer and protected module handle

  * _vm_start - main virtual machine function, it's called every time
                virtualized code is executed. It takes one argument -
                pointer to virtualized code (in .VM section)

  * _memmov   - simple __stdcall memmove(dest, src, length)
```

- **Protector:**

common.(cpp/h):

```
DWORD WINAPI _lde(BYTE* off);
```

wrapper for Hacker Disassembler Engine

```
int genCodeMap(BYTE* codeBase, int codeSize, DWORD* codeMap);
```

generates instructions map from code pointed by codeBase

```
void genPolyEncDec();
```

simple polymorphic encrypter/decrypter generator

```
void genPermutation(BYTE* buf, int size);
```

generates permutation
(http://en.wikipedia.org/wiki/Permutation)

```
void invPerm256(BYTE* buf);
```

inverse 256-byte permutation
(look at link)

```
void invPerm16(BYTE* buf);
```

inverse 16-byte permutation
(look at link)

```
void permutateJcc(WORD* buf, int elemCount, BYTE* permutation);
```

updates conditional jumps in _vm_jump (loader.asm)

```
int genRelocMap(BYTE* relocSeg, DWORD funcRVA, int funcSize,
                DWORD* relocMap);
```

transforms reloactions to simple table of RVAs

protect.(cpp/h)

```
int vm_init(BYTE** retMem, DWORD* _vmInit, DWORD* _vmStart);
```

```
virtualization engine initialization:
 - loads compiled loader to memory
 - generates polymorphic function
 - sets permutation for conditional jumps routine
 - generates random values for vm opcodes
```

```
BYTE* vm_getVMImg();
```

returns pointer to loaded vm engine (loader)

```
DWORD vm_getVMSize();
```

returns size of vm engine (loader)

```
void vm_free();
```

free memory allocated for loader

```
int vm_protect(BYTE* codeBase, int codeSize, BYTE* outCodeBuf,
               DWORD inExeFuncRVA, BYTE* relocBuf, DWORD imgBase);
```

```
core of x86 virtulizer
 - generates relocation table (genRelocMap)
 - generates map of instructions (genCodeMap)
 - main loop:
   * calls LDE on each instruction
   * checks for supported instruction
   * if supported then generating vm instruction in new buffer
   * if not, just copy original instruction with its length to new buffer
 - second loop:
   * corrects realtive jumps
   * encrypts each instruction
```

main.cpp

```
int vm_protect_vm(BYTE* vm_in_exe, BYTE* outBuf, DWORD imgBase,
                  DWORD vmRVA, DWORD newRVA)
```

'*vm over vm*' mode, At first it protects executable and then virtualize parts of vm engine (protected code is very slow)

```
int WINAPI AddDialogProc(HWND hDlg, UINT uMSg, WPARAM wParam,
                         LPARAM lParam)
```

GUI related function

```
DWORD rva2raw(WORD NumOfSections, IMAGE_SECTION_HEADER* FSH, DWORD rva)
```

converts RVA to RAW offset (in file)

```
DWORD searchFunction(BYTE* exeMem, char* functionName)
```

returns pointer to thunk of *functionName* in IAT (this function searches only imports from kernel32.dll)

```
void doProtect(HWND listBox, bool vmovervm, char* fileName)
```

PE struture handling, generates pre-loader

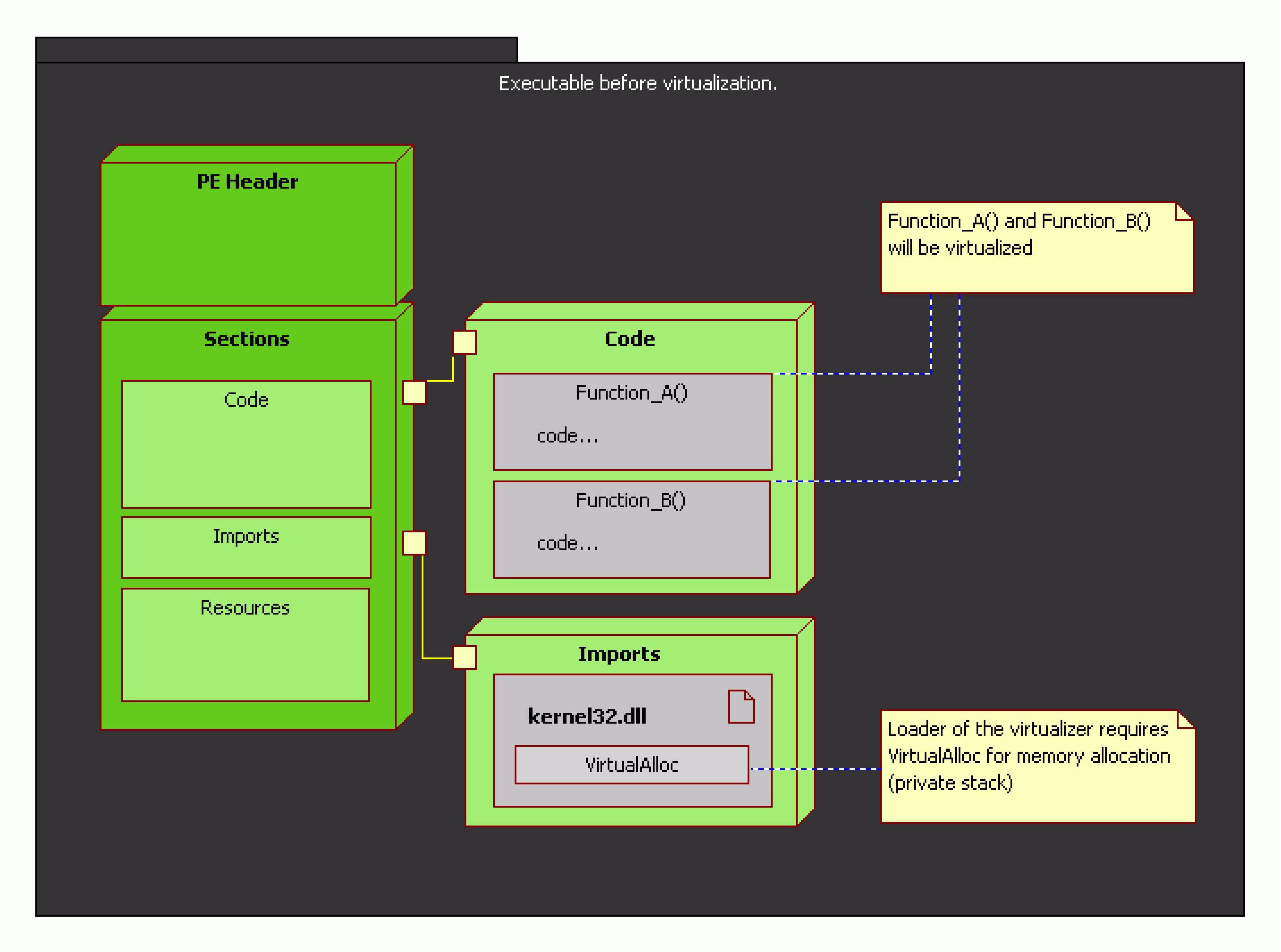
```
int WINAPI DialogProc(HWND hDlg, UINT uMSg, WPARAM wParam, LPARAM lParam)
```

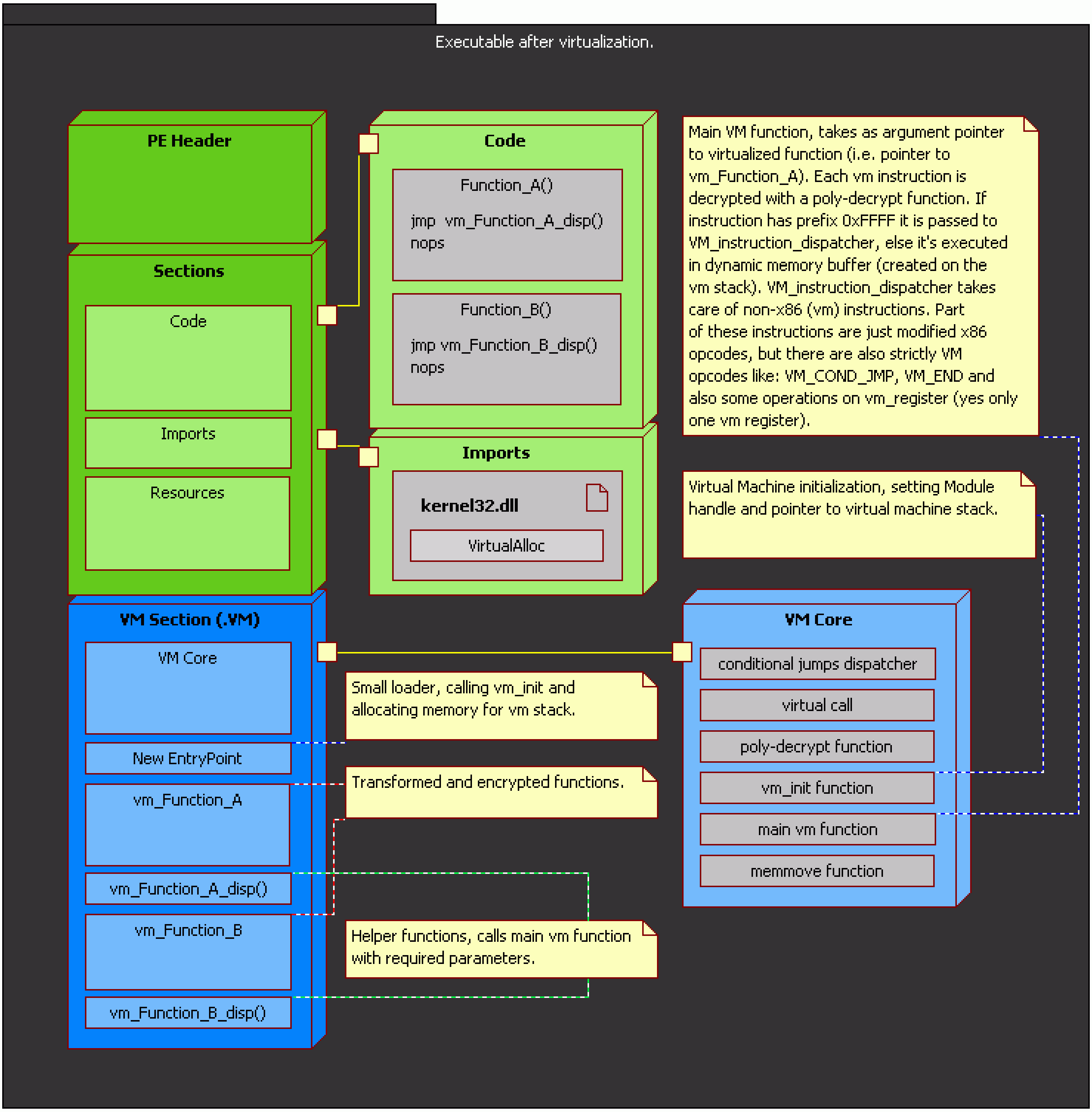
GUI related function

```
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                   LPSTR lpCmdLine, int nShowCmd)
```

main ;p

4. Diagrams


Executable before virtualization.
PE Header
Sections
Code
Imports
Resources
Code
Function_A()
code...
Function_B()
code...
Imports
kernel32.dll
VirtualAlloc
Function_A() and Function_B() will be virtualized
Loader of the virtualizer requires VirtualAlloc for memory allocation (private stack)



Main VM function, takes as argument pointer to virtualized function (i.e. pointer to vm_Function_A). Each vm instruction is decrypted with a poly-decrypt function. If instruction has prefix 0xFFFF it is passed to VM_instruction_dispatcher, else it's executed in dynamic memory buffer (created on the vm stack). VM_instruction_dispatcher takes care of non-x86 (vm) instructions. Part of these instructions are just modified x86 opcodes, but there are also strictly VM opcodes like: VM_COND_JMP, VM_END and also some operations on vm_register (yes only one vm register).

Virtual Machine initialization, setting Module handle and pointer to virtual machine stack.

Small loader, calling vm_init and allocating memory for vm stack.

Transformed and encrypted functions.

Helper functions, calls main vm function with required parameters.