Semi-automatic binary protection tampering

Yoann Guillot & Alexandre Gazet

Sogeti - ESEC

Abstract. Both on malicious binaries and commercial softwares like video games, the complexity of software protections, which aim at slowing reverse-engineering, is constantly growing. Analysing those protections and eventually circumventing them, require more and more elaborated tools. Through two examples, we illustrate some particularly interesting protection families and try to show their limits and how to remove them to recover a binary which is close to the original code. Each of our approaches is based on the use of the binary manipulation framework *Metasm*.

1 Virtual machines

In recent years, following constant processors performances growth, software protections became more and more ressource consuming. One class of protection perfectly illustrate this fact: virtual machine used as software protection. In the field of software protection, the term *virtual machine*, refers to a software component simulating a processor. We could also use the term of *virtual processor*. This "processor" is equipped with its own instruction set thus enabling the execution of any program specifically written in this machine code. In this paper, the term virtual machine always refers to a software protection component and never to more advanced methods or softwares, which would be dedicated to whole architecture virtualization like *VMWare* or *VirtualPC*.

This software protection method is now used in a large number of widely available commercial protection like *VMProtect*, *StarForce*, *Themida* or also *SecuROM*. Beyond those commercial products, we should also notice that many malwares use virtual machines to protect their own code.

When actually dealing with software protection, implementing a virtual machine amounts to add an abstraction level between the machine code — as it is perceived using a debugger or a disassembler — and its semantic, i.e. the function it operates. Analysing this abstraction level is often quite challenging and especially time consuming. The simulated processor possesses its own instruction set and machine code, which have to be analysed. Most of the time, the analyst has to develop tools to overcome this abstraction level and to be able to understand the code.

Few elements should be taken to assess the resistance to analysis of such a protection. Actually a virtual machine can be seen according to two following

models:

- A **concrete model**: it is the native code in which is implemented the virtual machine. Typically, we will find at this level all the primitives dedicated to memory manipulation, virtual registers, implementation of a *fetch-decode-execute* cycle and instruction handlers (a function which emulates an instruction or an opcode). Analysis complexity may be very important if the native code has been obfuscated. Obfuscation is another software protection technique that we will deal with later.
- An **abstract model**: a virtual machine simulates the behaviour of a given architecture or processor. The more the virtual architecture is complex and distant from the concrete architecture, the more the analysis is slowed down. First the translation process is slower, and second, the lack of references brought by the new architecture is a source of confusion. At the highest level of abstraction, the difficulty is also induced by the complexity of the program that is executed on the virtual processor.

1.1 Detection

The detection of a virtual machine is linked to its implementation. Most of the time, we can find some very characteristic schemes. The implementation of a *fetch-decode-execute* cycle is performed with a loop. An instruction handler is nothing more than a function taking some arguments, typically one or two, and returning the result. Virtual machines also often makes reference to those handlers using a function pointer table. From a structural point of view, printing the call graph is sometimes extremely revelating.

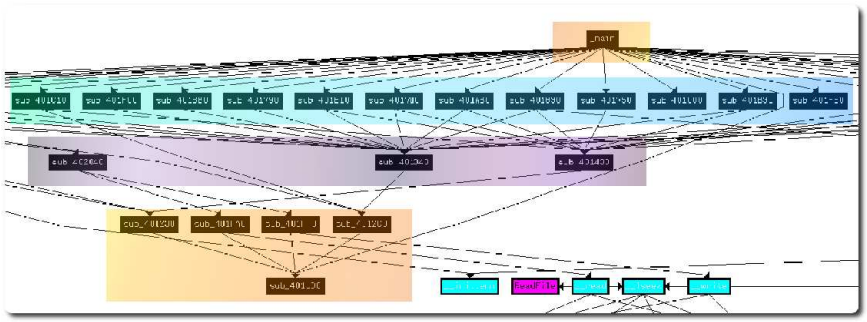

Fig. 1: Classic structure of a virtual machine.

We find in this example (Fig. 1) all the elements we have mentioned:

– At the top: the *main* function. It is the implementation of a *fetch-decode-execute* cycle. This loop dispatches the execution flow to the right handler, responsible for the current instruction treatment.
– Then we recognise our instruction handlers which are all located at the same logic level (here in blue).
– Handlers use *tool functions* (here in purple) to access the operands for example.
– Finally at the lowest level, all the primitives dedicated to the manipulation of physical virtual components: memory, registers, `IO` ports, etc. They are the link between the abstract and concrete models.

This scenario is the ideal case study. In practice, it is often harder to extract the whole structure. As a consequence, only the experience and the intuition will pay for the analyst.

1.2 Analysis

In a first time, the analysis of a virtual machine goes through the understanding of the abstract architecture or processor. Once it has been conceptualised, it is possible to identify each of the instruction handlers, and so the instruction set of the virtual machine. This analysis is most often based on a dynamic approach, like the observation of a data transfer between a register and a memory area. The main strength of a virtual machine lies in its abstract model while on the contrary its main weakness lies in its concrete model. The latter model contains all the clues that make the analysis possible: a context, a handler function pointer table, instructions and operands decoding primitives ...

The second step is the translation from the virtual machine code to a programming language which is easier to understand and which we have a good knowledge enough of. Typically it will be an *x86-assembler* like. This stage cannot be avoided and precedes advanced phases of reverse-engineering like decompilation.

1.3 Virtual machine hardening

There exist two main approaches for someone trying to harden a virtual machine based software protection. First, and most obvious, it consists of a complexification of the virtual machine itself; using a particularly exotic virtual architecture, an important instruction set, or by applying a destructuring process on the virtual machine. By destructuring process, we mean all processes which are able to conceal, split, and in a more general way to delay the concrete model analysis.

The second approach turns toward virtual machine multiplication and so making the analysis work increase. Once again there are two possible approaches which can be combined:

– A *flat* multiplication: in a binary, n parts are protected, each of them by a different virtual machine. If we consider d the performance deterioration

factor for one virtual machine, then whatever is the number of virtual machines, the performance deterioration factor for the whole binary is equal or lower to d.
- A *vertical* multiplication: here, the idea is to conceive virtual machines executing them-selves others virtual machines. If we consider the maximal number n of *stacked* virtual machines, then the performance deterioration factor may locally be equal to d^n. Even on a powerfull processor, performances are dramatically decreasing.

The goal of these techniques is to create an as great as possible asymmetry between the protection's cost and the analysis' cost. Nevertheless, both *flat* and *vertical* multiplication rely on the hypothesis that the author is able to produce unique and original virtual machines. We mean that analysing one instance of virtual machine should give as few as possible information concerning the analysis of another sample. Ideally, the author strives to force the analysis of each virtual machine. In practice, the author will simply try to complexify the automation of the analysis. Basic techniques of poly/metamorphism may appear to bring a sufficient and satisfying level of complexity.

This trend to more complex and elaborated virtual machine-based software protection clearly implies the need for tools able to carry out strong abstraction on the code.

2 Obfuscation

As previously mentioned, the strength of a software protection technique lies in the asymmetry between the protection's cost and the analysis cost. Obfuscation is a technique that consists of increasing analysis' complexity by deeply distorting code's readability. Obfuscation should be applied thoughtfully on important parts that are really valuable for an attacker, but it should not act as markers for these parts. An important problematic is to define the resistance and the effectiveness of an obfuscation function.[**?**].

An obfuscation process or function may be defined as a transformation applied to the code that preserves its **semantics**.

2.1 The semantics

Semantics is the meaning the we give to the code, its function or its role. If we consider a part of code denoded p, p' its obfuscated form and $\mathbb{I}$ the set of initial possible states, then preserving its semantics may be formulated as follows:

$$\forall s \in \mathbb{I}, p(s) = p'(s)$$

Locally, i.e. at instructions level, it is clear that contexts may differ partially. Just take a part of code which is responsible for the calculation of an addition:

once obfuscated, it should return the correct result, whatever the intermediate states it goes through. In a more concrete way, it means that many registers, or memory blocks, are essential because they contain the results or have an influence on it; others are not. This is what we call the **significant context**.

Preserving the semantics of the significant context is essential, since it guarantees the correct execution of the binary once the obfuscation function has been applied.

2.2 The transformations

Although processes of obfuscation differ depending on their objectives and the level of abstraction at which they are applied, all can be modeled in the form of a transformation. We will first illustrate this concept with few examples that we have found in different binaries we have analysed.

Neutral element. Obfuscation using neutral elements (Fig. 2) is relatively weak and we will try to see why.

```
1   ror eax, 0dh    ; @948c6d  c1c80d
2   xchg eax, edx   ; @948c70  92
3   ror edx, 13h    ; @948c71  c1ca13
4   xchg eax, edx   ; @948c74  92
```

```
1   rol eax, 0ah    ; @948bd7  c1c00a
2   xchg eax, ebx   ; @948bda  93
3   rol ebx, 16h    ; @948bdb  c1c316
4   wait            ; @948bde  9b
5   xchg eax, ebx   ; @948bdf  93
```

```
1   rol esi, 9      ; @948a94  c1c609
2   pushfd          ; @948a97  9c
3   add eax, esi    ; @948a98  01f0
4   wait            ; @948a9a  9b
5   sub eax, esi    ; @948a9b  29f0
6   lea esi, dword ptr ds:[esi] ; @948a9d  8d36
7   push eax        ; @948a9f  50
8   pop eax         ; @948aa0  58
9   popfd           ; @948aa1  9d
10  rol esi, 17     ; @948aa2  c1c617
```

Fig. 2: Neutral element based obfuscation

Actually this is the simplest scenario we can find when dealing with obfuscation: significant contexts before and after execution are equals. Independent one from the others, the design of each pattern is trivial. Each effect is subsequently cancelled, thus the semantics is preserved. In the last of the three examples

above, *sub* instructions cancel *add* instructions, *pop* instructions cancel *push* instructions, *rol 9* and *rol 17h* complement each other arithmetically... This property will be the basis of a method for automatic detection. Moreover, a second property is also very intersting and useful during a manual analysis: a sort of visual symmetry can be observed in the instruction blocks.

Once we have defined inserted patterns as neutral elements, we know that it is possible to reduce the code by simply masking those pattern. In practice, those sequences are often replaced by *nop* instructions, which happened to be a remarkable form of neutral element; a neutral element is substituted to another, then codes are equivalent. This attack is quite trivial. Once the patterns are identified, a basic matching at hexadecimal level is enough. There is no need to interpret the code. The more an attack is led at a low level of abstraction, the more it is simple and effective.

```
1  mov eax, var1
2  xor eax, 2c1a83c1h
3
4  mov ebp, var2
5  xor ebp, 0f91628c5h
6  xor ebp, 0d50cab04h
```

Fig. 3: Double XOR

Constants expansion. Constants expansion stands for a kind of transformation which aims at complexifying the expression of a constant. Let's consider the following pattern as an example (Fig. 3).

Parameters *var1* and *var2* are *XORed* using the same 32 bits key. The key appears distinctly for the first parameter, while this is more ambiguous until we realized that: `0f91628c5h` $\oplus$ `0d50cab04h` $==$ `2c1a83c1h`. Here the idea is to express the key as the result of the *XOR* of two constants.

The constants used in some algorithms are really significant, and can be used to immediately identify a function as a hash function for example. Thus, it is really interesting to try to hide them.

In our example, double *XOR* is really basic, but others systems may be much more arduous, both in their form (manipulations on registers, on the stack, in memory) and in their content (constant expressed like the result of a complex polynomial or trigonometric formula).

Structural obfuscation. Another type of frequently observed transformation may take the form of the parts of code represented in Fig. 4.

```
1   push loc_403f84h          ; @403f07  68843f4000
2   ret                       ; @403f0c  c3
```

```
1   push 89h                  ; @21730  6889000000
2   add dword ptr [esp], 179h ; @21735  81042479010000
3   popf                      ; @2173c  669d
4   jnz loc_2173e             ; @2173d  75ff
```

Fig. 4: Structural obfuscation

The first example consists in pushing an address on the stack and then to use the *ret* instruction as a jump. This type of transformation differs from simple neutral element insertion in that the flow graph is modified: this is the reason why we speak about structural obfuscation.

The second example is a well known type of structural obfuscation: the false conditional jumps. For these jumps, the calculation of the condition always returns true or always false. One of the two branches is never used. We will explain in detail the nature of this protection later in this document. In this example, using constants 89h and 179h, following by the *popf* instruction — let us recall that it reloads the processor's flag using the *dword* located at the top of the stack — the code writer controls the condition of the jump instruction located at *@2173f*.

In a general way, the apparent condition of the jump is used to artificially complicate the control flow graph. In addition, this technique has the quite interesting property to disrupt some disassembly engines. We said that one of the two branches is a dead one; it is possible to insert garbage instructions aiming at polluting the listing, inserting false references...

2.3 Complexity

The complexity resulting from the application of an obfuscation function relies on the difficulty for an analyst to understand the transformation. The more it is identifiable, the more it is easy to revert, and at a low level of abstraction. All constant, thus predictable, element represents free information for the analyst. This is particularly true when dealing with software protection. Inserting static patterns is conceptually weak: even if a great variety of patterns increase the difficulty, it is generally not enough to ensure a satisfying level of protection.

From developper's point of view, the more effective solution consist of designing a set of simple transformations (f, g...), that s/he masters easily and to apply them successively, the output of one being the input of another. These basic functions should be sufficiently varied: pattern insertion, trap insertion, control flow graph modification, variable expansion, disassembly engine trap... The development of each function is thus easier, and preservation of semantics is more easily provable.

Following composition rules, it is possible to obtain as a result a final obfuscation function f_{res}, defined as $f_{res} = f \circ g \circ \ldots$ which is a priori much more resistant to analysis that each individually function is.

To further strengthen the resistance, it may be efficient to vary the final function of obfuscation by randomizing the order of composition are the number of composed. We can also consider parameterised functions. The only limitations are the imagination of the author and the performance degradation that is tolerable. The technical constraint is becoming secondary as the capacity of processors is increasing year after year.

3 Metasm

Metasm[1] is an opensource framework in which it is possible to interract with machine code in many formats (hexadecimal, assembler, C). It is entirely written in Ruby[2]. That makes it the perfect tool for our needs : it will be easy to change the way things are done, as an example how binary instructions are disassembled. *Metasm* is a multiplatform and multiOS framework. Consequently, we should be able to create an object to interact with any virtual processor we may encounter.

The framework was first introduced during the SSTIC 2007 conference[**?**], and later the same year during the Hack.lu conference[**?**].

3.1 Code deobfuscation

A virtual machine used as a software protection is often implemented using obfuscated native code. In order to ease the preliminary code analysis, we will need to pass through this layer of protection.

This is acomplished by reading manually a few code sequences, finding the obfuscation patterns used, and reverting them, either by the removal of the useless instructions (*junk code*), or by restoring the standard instructions in the case of behavior-level obfuscation.

This can be done at different times :

- either in the binary file before disassembling,
- or dynamically while disassembling,
- or on the assembly source once the disassembling is completed.

The first option is only possible if we manage to find a binary signature for every pattern; however it may cause data corruption if the pattern has a false positive (e.g. if it appears in the middle of a data section).

The last option is safer, but needs the disassembling process to work on the obfuscated code. However it is quite possible that the obfuscated code implements some kind of function call, or a jump sequence, so that the disassembler misses it. In this case, we will work on a fraction of the interesting code only.

This is why we chose the second approach.

Standard disassembly. Out of the box, the disassembly engine in *Metasm* works this way :

1. Disassemble the binary instruction at the instruction pointer.
2. Analyse the effects of the instruction.
3. Update the instruction pointer.

[1] `http://metasm.cr0.org/`
[2] `http://ruby-lang.org/`

The analysis of the effects of a given instruction enables to tell whether the instruction does some memory access and/or changes the execution flow. If this analysis reveals such an effect and this effect depends on the value of a machine register, *Metasm* uses a backtracking technique to try to determine the value of those registers.

Backtracking. Backtracking in *Metasm* consists in the symbolic emulation of each instruction while walking all the execution flows that arrive to the currend address, until the traced expression's value is found. The flows used are tagged so that if we may later find a new code flow that will run into the flow we are examining, we are able to walk this new flow which may find a new value for the expression.

The backtracking method needs an arbitrary arithmetic expression, zero or more address to stop this backtracking process (addresses of start of symbolic execution) and the address to begin the backtrace (end of symbolic execution). The expression may include the symbolic value of any processor register (value at the end address). The method will then return the same expression expressed using the symbolic value of the registers at the start address.

For example, if we search the value of the *eax* register after execution of "*add eax, 4*", we will get *eax+4* : the value of *eax* at the end of the instruction equals the value of *eax* before the instruction plus four.

Patched disassembly. The way we will proceed is not very intrusive but will be unable to handle obfuscations that change the execution flow with jumps further than a few bytes at a time.

We will modify the first step of the disassembler loop : how a binary instruction is disassembled. This step is implemented in the method called *CPU#decode_instr_op*. When an instruction is decoded, it will be checked against a list of predefined patterns, determined by a manual observation, to see whether it matches the beginning of one of the pattern. If it matches, the next instruction is decoded to continue the pattern matching. If the whole pattern matches, the corresponding unobfuscated instruction is returned in place of the whole sequence. This will totally remove the obfuscation layer.

Doing it this way makes it recursive, which means that it will automatically solve interweaved patterns. This has the added benefit of reducing the length of the pattern list.

Additionally, *nop* instructions are always merged into the following instruction, so that the junk code is absent in the final assembly listing.

The following figures (Fig. 5) show the result on a sequence of instructions. We can see the result by looking at the binary encoding of the *pop eax* instruction.

In practice, we start by entering the most visible patterns, look at the result, and refine the pattern list until the output is readable enough.

A more sophisticated approach would be to automatically analyse all code sequence we encounter to determine it's effects, and try to express those effects

```
1   push 42h          ; @21d38h  6a42
2   ror ebp, 0dh      ; @21d3ah  c1cd0d
3   xchg edx, ebp     ; @21d3dh  87d5
4   ror edx, 13h      ; @21d3fh  c1ca13
5   xchg edx, ebp     ; @21d42h  87d5
6   pop eax           ; @21d44h  58
7   inc eax           ; @21d45h  40
```

Fig. 5: Original

```
1   ror %1, X
2   xchg %1, %2
3   ror %2, 0x20-X
4   xchg %1, %2
```

Fig. 6: Junk code pattern

```
1   push 42h          ; @21d38h  6a42
2   pop eax           ; @21d3ah  c1cd0d87d5c1ca1387d558
3   inc eax           ; @21d45h  40
```

Fig. 7: Final

with less instructions. This would work particularly well with junk code (like 'add 2, sub 2') ; but if the junk only preserves significant registers (for the program) and allows modifications to unused registers we would need to define manually what is significant and what is not. This is quite similar to *code decompilation*, as we will see later.

3.2 Automatic analysis of virtual machine handlers

This can only be done after a preliminary manual analysis, which is necessary to determine the virtual machine architecture :

- encoding of the virtual instructions,
- implementation of virtual registers (memory ? dedicated real register ?),
- virtual code flow.

This analysis will aswer those questions:

- How is the transition between virtual instructions done ?
- How are subfunctions called ?
- How do subfunctions return ?

It is then possible to automatically analyse the handlers, at least those implementing simple functions (like arithmetic operations or data movements), by comparing the virtual processor state before and after the handler symbolic execution.

We will use the backtracking engine of *Metasm* to modelize the transformations done on all virtual registers by the handler ; also to track all memory accesses.

These two informations suffice to summarize the handler's effects; we will call them the handler's *binding*. We can then compare those transformations to a set of known shapes to name the handlers (e.g. "addition between two registers") This is mostly useful to assign a mnemonic to each handler, in order to get a readable virtual instructions listing.

Handlers whose binding are not recognized will need to be manually analysed.

Note that this analysis may be done on moderately obfuscated handlers, as long as the binding can be accurately computed.

3.3 Pseudocode disassembly

Once all the handlers are identified, we can build a new *CPU* class for the virtual processor, and integrate it in *Metasm*. We get this way a full-blown disassembler able to work directly on the binary pseudo-code. This class may be automatically created from the results of the automatic handler analysis.

The handlers of a virtual machine are quite simple, so this modelisation is easy.

Furthermore, if we write a few other methods to handle the virtual assembly language parsing, we could have a working assembler for the pseudocode, able to generate a binary that could run on the virtual machine.

3.4 Decompilation

Most of the time a virtual machine instruction set is minimalist ; and it is cumbersome to write a program directly in this language. The author often uses another layer to ease his task, it may be a macro-assembler, or even a rudimentary C compiler.

```
1  mov reg1, addr_op1
2  load reg1, [reg1]
3  mov reg2, addr_op2
4  load reg2, [reg2]
5  add reg1, reg2
6  mov reg3, addr_result
7  stor [reg3], reg1
```

Fig. 8: Macro for an in-memory addition

The macros are easy to spot, and it is very feasible to regenerate the macro-code directly. We are then able to transform the low-level assembly to a higher-level language, very close to what the original author manipulated.

At this step, the protection mechanism applied to the binary is totally removed, and the real reverse-engineering work can begin to find the algorithms in use.

4 Solving the T2 2007 challenge

We propose to use the features of *Metasm* to solve the challenge of the T2 conference for year 2007. This challenge includes many of the features we were working on, it will be the perfect illustration of what we've done.

4.1 The challenge

The challenge is a simple Windows binary.

Once launched, it asks for a password and displays wether it is good or not. The goal is to find a password that the software will accept.

A quick disassembly of the binary shows the actions of the program:

1. extract a file named "driver" to the disk, from the program's resources,
2. load it as a kernel driver,
3. ask the password,
4. send the password to the driver through an *IOCTL* on a special file,
5. read the response from the driver,
6. display it to the user.

So we will need to look into this driver file, whose analysis is much more interesting. The driver handles the *IOCTL* with a massively obfuscated function ; the disassembler hangs on an indirect jump that it cannot resolve.

Note: the assembly listing produced by *Metasm* is the instruction followed by a comment where we find the address of the instruction (prefixed by a @), its binary encoding, and the effects of the instruction : memory access and code flow modifications. *010203..<+37>* means that the instruction is encoded starting by the bytes *010203* and goes on for 37 bytes.

4.2 Desobfuscation

A quick look at the code shows that the code is mostly junk code.

Wheels of confusion. We find many different obfuscation techniques.

Here is one of those patterns (Fig. 10).

We have here a little obfuscation recital in a few lines:

- First of all, structural obfuscation through a fake *call* (l. 3), followed by a modification of the return address (l. 5).
- Then a fake conditional jump : using the constants 89h et 179h and the *popf* instruction — which loads the processor flags from the stack — the author ensures that the *jnz* (l. 10) is always followed.

```
 1  // Xrefs: 1101ch
 2  loc_215f8:
 3   push esi                      ; @215f8h  56
 4   push ebx                      ; @215f9h  53
 5   lea esi, dword ptr [esi]      ; @215fah  8d36
 6   ror edi, 0dh                  ; @215fch  c1cf0d
 7   xchg ebx, edi                 ; @215ffh  87df
 8   ror ebx, 13h                  ; @21601h  c1cb13
 9   xchg ebx, edi                 ; @21604h  87df
10   push ebx                      ; @21606h  53
11   push ecx                      ; @21607h  51
12   lea ecx, dword ptr [ebx+4]    ; @21608h  8d4b04
13   xor ecx, edx                  ; @2160bh  31d1
14   xchg ecx, dword ptr [esp]     ; @2160dh  870c24
15   push edx                      ; @21610h  52
16   mov edx, dword ptr [esp+4]    ; @21611h  8b542404
17   rol edx, 0fh                  ; @21615h  c1c20f
18   mov dword ptr [esp+4], edx    ; @21618h  89542404
19   pop edx                       ; @2161ch  5a
20   pop dword ptr [esp+(-8)]      ; @2161dh  8f4424f8
21   pop ebx                       ; @21621h  5b
22   rol eax, 2                    ; @21622h  c1c002
23   rol eax, 1eh                  ; @21625h  c1c01e
24   pushfd                        ; @21628h  9c
```

Fig. 9: The driver's code

```
 1   pushfd                       ; @2164fh  9c
 2   push edi                     ; @21650h  57
 3   call loc_21656               ; @21651h  e8.. noreturn x:loc_21656
 4  loc_21656:
 5   add dword ptr [esp+0], 24h   ; @21656h  8184240000000024000000
 6   pop edi                      ; @21661h  5f
 7   push 89h                     ; @21662h  6889000000
 8   add dword ptr [esp], 179h    ; @21667h  81042479010000
 9   popfd                        ; @2166eh  9d
10   jnz loc_21670                ; @2166fh  75ff  x:loc_21670
11
12  // ------ overlap (1) ------
13  // Xrefs: 2166fh
14  loc_21670:
15   jmp edi                      ; @21670h  ffe7  x:loc_2167a
16
17  // ------ overlap (1) ------
18   out eax, 90h                 ; @21671h  e790
19   nop                          ; @21673h  90
20   mov eax, dword ptr [ebp+8]   ; @21674h  8b4508
21   adc eax, edi                 ; @21677h  11f8
22   int 3                        ; @21679h  cc
23
24  // Xrefs: 21670h
25  loc_2167a:
26   pop edi                      ; @2167ah  5f
27   popfd                        ; @2167bh  9d
```

Fig. 10: *Overlapping* and fake conditional jump

– A bit of *overlapping*. It consists in the encoding of two instructions, so that the last bytes of the first instruction are also the first bytes of the last instruction. This is possible because in the IA32 architecture, the instructions do not need to be aligned, and do not have a constant size So *jmp edi* (binary *ffe7*) is coded using the bytes from the instructions *jnz loc_21670* (*75ff*) and *out eax, 90h* (*e790*). This kind of obfuscation would be impossible on an ARM[3] processor for example, where the instructions have a fixed width and must be aligned.

This is quite charming, but has some major drawbacks:

– obfuscation patterns are almost never interweaved,
– they have very weak polymorphism (only on the registers used).

Those would allow us to get rid of almost all junk code using only a binary-match pattern. Because we are trying to be as correct as possible, we will however use the approach we talked about before: integration in the disassembler.

We find about 20 different patterns we can split in two groups: quite simple sequences (ex: rotating a register of 32bits), and more complex ones. These often involve pushing values on the stack and manipulating them ; however they are always put inside a *pushfd ... popfd* containment, which make them quite easy to spot.

Once those patterns are integrated to the disassembler, the code is much more human-friendly.

Set the record straight. We then encounter a quite intriguing construction.

It (Fig. 11) reads the value of the processor counter [4], and uses this value to chose which branch of a conditional jump to take. But this value is almost random, and hard to guess ! You have to be quite optimistic to play *heads or tails* in the code of a driver ; and as Albert said, "*God does not play dices*".

Lets take a step back.

A quick manual look shows that both branches that follows the conditional jump are exactly the same: they are both B and B' implementations of the same semantic A (Fig. 12).

This obfuscation technique differs from the simple pattern insertion because it changes the execution flow inside the binary: it is a structural obfuscation. The use of *rdtsc* is clever: two executions of the file will not produce the same trace, because the value of the counter is not predictable. So a breakpoint set without caution on a first review of the code may never be triggered during later

[3] Advanced RISC Machine
[4] *RDTSC* means *ReaD TimeStamp Counter*.

```
 1  // Xrefs: 1101ch
 2  loc_215f8:
 3   push esi                ; @215f8h   56
 4   push ebx                ; @215f9h   53
 5   pushfd                  ; @215fah   8d36c1cf0d87dfc1cb13..<+37>
 6   rdtsc                   ; @21629h   9c5031c0668cc83d0900..<+15>
 7   imul ecx, ebx           ; @21642h   0fafcb
 8   cmp cl, 7fh             ; @21645h   80f97f
 9   jnb loc_21aba           ; @21648h   0f836c040000  x:loc_21aba
10
11   popfd                   ; @2164eh   9d
12   pop ebx                 ; @2164fh   9c57e800000000818424..<+48>
13   pop esi                 ; @21689h   c1c7099c01fa9b29fa8d..<+8>
14   [...]
15
16  loc_21abah:
17   popfd                   ; @21abah   9d
18   pop ebx                 ; @21abbh   57870c245f87cf5b
19   pop esi                 ; @21ac3h   c1ce0d87cec1c91387ce..<+9>
```

Fig. 11: Unobfuscated code

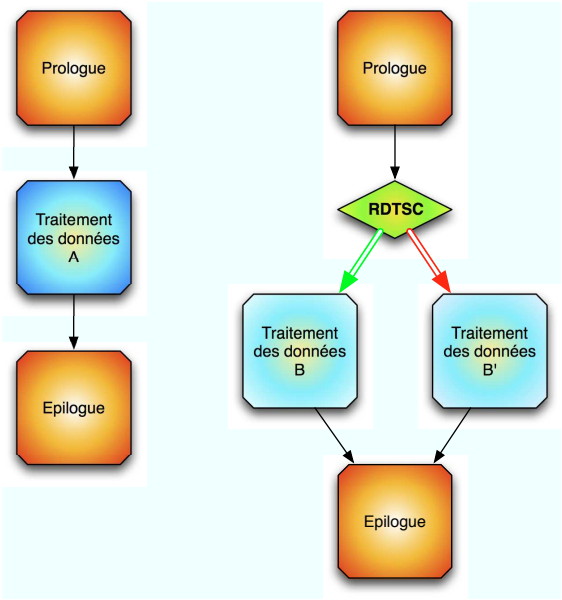

Fig. 12: Structural obfuscation

reexecutions. On the other hand, *rdtsc* is one of the instructions that will never appear in a normal code sequence, and will flag the code as suspicious for an attacker.

Note that the two branches are not exactly the same : they are independently obfuscated by random patterns that we've seen before ; therefore we cannot make a bit-to-bit comparison to detect the duplication.

We've chosen to consider this sequence as a junk code pattern, and the disassembler will always follow the first branch of the conditional jump from now on. The other branch is still disassembled however, just in case we would want to manually check the similarity of both ways, but it will not be shown in the samples we'll use in this paper.

Master of the rings. There are still some unexpected code sequences.

```
 1   pushfd              ; @131c0h  9c
 2   push eax            ; @131c1h  50
 3   xor eax, eax        ; @131c2h  31c0
 4   mov ax, cs          ; @131c4h  668cc8
 5   cmp eax, 9          ; @131c7h  3d09000000
 6   jle loc_131d5h      ; @131cch  7e07  x:loc_131d5h
 7   rdtsc               ; @131ceh  0f31
 8   imul eax, ecx       ; @131d0h  0fafc1
 9   jmp eax             ; @131d3h  ffe0  x:unknown
10
11  // Xrefs: 131cch
12  loc_131d5h:
13   pop eax             ; @131d5h  58
14   popfd               ; @131d6h  9d
```

Fig. 13: Test of ring 0 execution

This code (Fig. 13) checks if it is being run in the Windows kernel context (ring 0), or in the context of a userland standard process. In the kernel, the segment selector *cs*'s value is *8* ; in userspace it is *0x1b*. So the only goal of the whole sequence, that we'll encounter many times in the driver code, is to forbid the execution of the protection in a standard userland process context.

One of the characteristics of the challenge, for anybody who would want to analyse it using dynamic tools, is that all the intersting code is run in kernel context ; and this environment is not ideal for debugging. Indeed, most of the debugging tools are focused on userland code analysis, and few can handle the specificity of ring 0 debugging. So it would be tempting to run the interesting code in a standard process, to be able to use standard tools to watch its behaviour. When running this code sequence, the value of the *cs* selector will make the conditional jump (l. 6) not to be taken. In this case, the three instructions that follow will build a pseudo-random address and route the code flow to it

(using the jump at l. 9), which ensures an immediate crash for the program: in the best case the address is invalid for code execution, otherwise it will contain code that is not made to be run this way and will crash sooner or later.

```
 1  loc_173a7h:
 2   call loc_173ach            ; @173a7h  e8..  noreturn x:loc_173ach
 3  loc_173ach:
 4   pop edx                    ; @173ach  89ed9c873424569b9d5e5a
 5   cmp edx, 7fffffffh         ; @173b7h  c1c10a5156e800000000..<+18>
 6   jnb loc_17436h             ; @173d3h  0f835d000000  x:loc_17436h
 7
 8   rdtsc                      ; @173d9h  558d6d0811c5872c2450..<+17>
 9   add edx, eax               ; @173f4h  01c2
10   jmp edx                    ; @173f6h  c1..  x:unknown
11  // [45 data bytes]
12
13  // Xrefs: 173d3h
14  loc_17436h:
15   mov edx, dword ptr [ebp+0] ; @17436h  8b5500
```

Fig. 14: Other ring 0 test

Another sequence has the same behaviour (Fig. 14).

This one checks the address of the code ; if it is below *80000000h* (ie. userland[5]), a random jump is taken.

Those two sequences will join the ranks of the junk code patterns hidden by the disassembler.

When you're pushed... At this point of the analysis, the code shown is very reduced, but still includes long and unfriendly sequences. There are no more pure junk code sequences, but obfuscated sequences having a side-effect : this is behavioural obfuscation.

Most of these sequences will push a value and manipulate it on the stack (Fig. 15).

We also find references to the memory, always through an *[ebx+<offset>]* expression, that are very intriguing ; the offsets match nothing and seem randomly chosen (Fig. 16 l. 5, 13, 15). Further examination shows that a large memory area is used to hold temporary values only to obfuscate the code : in fact the two branches following the *rdtsc* must have the same semantic but the offsets are not shared between them. We will ignore such memory writes.

This enables us to remove large sequences of code, and we finally obtain a very concise listing.

[5] Yes, Windows may have booted with the /3G switch...

```
 1   push esi                     ; @150eah  56
 2   lea esi, dword ptr [ebp+8]   ; @150ebh  8d7508
 3   adc esi, ecx                 ; @150eeh  11ce
 4   xchg esi, dword ptr [esp]    ; @150f0h  873424
 5   push ecx                     ; @150f3h  51
 6   mov ecx, dword ptr [esp+4]   ; @150f4h  8b4c2404
 7   shl ecx, cl                  ; @150f8h  d3e1
 8   mov dword ptr [esp+4], ecx   ; @150fah  894c2404
 9   pop ecx                      ; @150feh  59
10   pop dword ptr [esp+(-0ch)]   ; @150ffh  8f4424f4
11
12   push dword ptr [ebx+0e92h]   ; @176beh  ffb3920e0000
13   push ecx                     ; @176c4h  51
14   mov cl, 11h                  ; @176c5h  b111
15   rol dword ptr [esp+4], cl    ; @176c7h  d3442404
16   pop ecx                      ; @176cbh  59
17   pop dword ptr [ebx+0e92h]    ; @176cch  8f83920e0000
```

Fig. 15: Behavioural obfuscation

```
 1  loc_175e6h
 2   mov eax, dword ptr [ebp+0ch]     ; @175e6h  9c600f3101c8c1c20a5
 3   xor eax, 1749c891h               ; @1767eh  c1ce0d87cec1c91387c
 4   push dword ptr [ebx+eax]         ; @1769bh  ff3403
 5   pop dword ptr [ebx+0e92h]        ; @1769eh  50e8000000009c81842
 6   [...]
 7
 8  rdtsc_17882h:
 9   mov ecx, dword ptr [ebp+0ch]     ; @1788eh  c1c00a5053e80000000
10   xor ecx, 1749c891h               ; @178a7h  6089f0d3c2619c81f19
11   push ecx                         ; @178c6h  9c55e80000000081842
12   mov ecx, dword ptr [ebx+ecx]     ; @178f4h  9c5031c0668cc83d090
13   mov dword ptr [ebx+25b8h], ecx   ; @1790eh  568d730431de8734245
14   pop ecx                          ; @1792eh  c1c102c1c11e59
15   push dword ptr [ebx+25b8h]       ; @17935h  9c5031c0668cc83d090
16   pop ecx                          ; @17952h  59
```

Fig. 16: Memory data junk

4.3 The virtual machine

Now we can begin the code analysis.

The first step is a big memory allocation (106000 bytes), whose address is saved in *ebx*.

The code run then is a sequence of blocks with a very similar structure. Each begins with the *rdtsc* splitting obfuscation sequence ; they manipulate *dword*-sized memory area, whose address is taken from the *ebp* register and then xored with a block-specific key. Those values are sometimes used as indexes in the table allocated at the beginning in *ebx*.

Finally an epilog will always update *ebp* with the value stored at *[ebp+4]*, and run the block whose address is stored at *[ebp]*.

We will interpret this execution scheme as the sequence of instructions of a virtual processor: the blocks will be the handlers, the data at *ebx* will be the virtual processor context, and the data at *ebp* will be the virtual instruction operands.

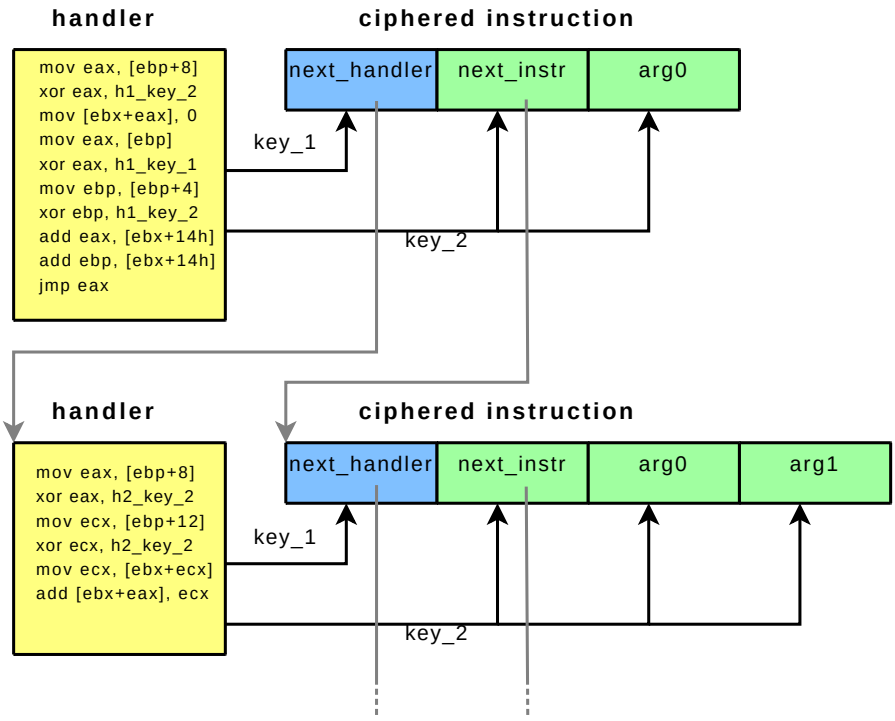

Fig. 17: T2 virtual machine architecture

In the belly of the beast. The *ebx* register always points to the beginning of the memory area allocated during initialisation, which holds the processor execution context. The *ebp* register points to the virtual instruction being run.

Those instructions are a sequence of 2 to 6 memory words. Each one is ciphered using a 32bits *xor* key, which is unique per handler. The first word is ciphered using another handler-defined key (Fig. 17).

This first word holds the offset of the next handler to run; it has to be added to the *.data* section base address to get the real memory address. The second word holds the offset of the next instruction, also relative to the section base. The following words have a handler-specific meaning. They are often an integer (*immediate* value), or an index in the *ebx* table (ie. a virtual register).

The virtual registers are a few memory words, stored at a fixed offset from *ebx*. The way the handlers are implemented, the register access method allows arbitrary memory access, but only a handful of offsets are used in practice.

Every register has a specific role[6] :

Offset	Name	Role
0x4	*esp*	stack pointer
0x8	*ebp*	frame pointer
0x64	*r64*	generic
0x68	*r68*	generic
0x78	*r78*	memory indirection
0x0	*esp_init*	stack pointer initial value
0xc	*host_esp*	host stack pointer
0x18	*retval*	subfunction return value

Initialisation. Let's get back to the initialisation sequence.

The first instructions compute the start address for the *.data* section, and store it in *esi* (l. 1-5).

A call is made to allocate 0x19e10 (106000) bytes, and the address of this buffer (the virtual machine cpu context) is saved in *ebx* (l. 7-9).

Then some fields of the context are initialized

- The *.data* section base address is stored in *[ebx+14h]* (l. 11).
- The real cpu stack is stored in *host_esp* (*[ebx+0ch]*) (l. 12).
- The virtual stack is initialized with the value *ebx+101d0h* (*ebx+66000*), this address is stored in *esp* and *esp_init* (l. 13-15).

A handler. To illustrate the working of virtual instructions, let's take a look at an addition handler.

1. The two first instruction retrieve and decipher the index of the source register from the field at *+0ch* in the virtual instruction (2^{nd} argument).

[6] Registers in the last part of the table are scarcely ever used.

```
 1  loc_215f8h:
 2   call loc_216bch                 ; @215f8h  56538d36c1cf0d87dfc
 3  loc_216bch:
 4   pop esi                         ; @216bch  5e
 5   sub esi, 0e6bch                 ; @216bdh  81eebce60000
 6
 7   push 19e10h                     ; @216c3h  68109e0100
 8   call dword ptr [esi]            ; @216c8h  ff16  r4:xref_13000
 9   mov ebx, eax                    ; @216cah  89e489c3
10
11   mov dword ptr [ebx+14h], esi    ; @216ceh  c1c602c1c61e5089f08
12   [...]
13   mov dword ptr [ebx+0ch], esp    ; @218f7h  89ff9c871c24539b9d5
14   lea eax, dword ptr [ebx+101d0h] ; @21904h  8d83d0010100
15   mov dword ptr [ebx], eax        ; @2190ah  558704245d955089c09
16   mov dword ptr [ebx+4], eax      ; @21969h  56873c245e87fe89430
```

Fig. 18: VM initialisation

```
 1  loc_15336:
 2   mov ecx, dword ptr [ebp+0ch]  ; @15336h  9c600f3101f131cbc1c1..<+102>
 3   xor ecx, 842b1208h            ; @153a6h  528d550811ca87142451..<+49>
 4   mov ecx, dword ptr [ebx+ecx]  ; @153e1h  516089d0d3c2618b0c0b..<+30>
 5   mov eax, dword ptr [ebp+8]    ; @15409h  51e8000000009c818424..<+19>
 6   xor eax, 842b1208h            ; @15426h  9cc1c10a5155e8010000..<+37>
 7   add dword ptr [ebx+eax], ecx  ; @15455h  c1c60a5651e800000000..<+15>
```

Fig. 19: Addition of two registers

2. This index is used to read the virtual register.
3. The two following instructions in the handler retrieve and decipher the index of the destination register at *+8* in the virtual instruction (1^{st} argument).
4. Finally the addition is done and stored in the context.

Thus, control is given to the following couple handler/instruction, thanks to a code that is shared by all handlers.

```
1   mov ecx, dword ptr [ebp+0]    ; @1546eh  8b4d00
2   xor ecx, 149f0c63h            ; @15471h  c1ce0d87cec1c91387ce..<+12>
3   mov ebp, dword ptr [ebp+4]    ; @15487h  c1c20a92c1c0169b928b6d04
4   xor ebp, 842b1208h            ; @15493h  6089f0d3c26181f55204..<+12>
5   add ebp, dword ptr [ebx+14h]  ; @154a9h  036b14
6   add ecx, dword ptr [ebx+14h]  ; @154ach  c1c11287d1c1c20e9087..<+4>
7   jmp ecx                       ; @154bah  89ff9c871424529b9d5affe1
```

Fig. 20: Transition between two handlers

Next handler's (l. 1-2) and instruction's (l. 3-4) offsets, are decrypted in current instruction code (one should notice the use of the specific key for handler's offset), they are then converted into absolute adddresses by adding base address of *.data* section, stored in *[ebx+14h]* (l. 5-6). Finally control is given to the next handler, *ebp* pointing the virtual instruction to interpret.

4.4 Modelling

This architecture's main issue is that we have neither the handlers list, neither instructions list: we have to follow the execution flow to find decryption keys for each handler, and to decrypt each instruction to recover then next couple handler/instruction, again and again.

This operation is quite tedious to do by hands, that is the reason why we will automate it.

Follow the white rabbit. In order to do so, we use *Metasm* backtracking engine, which permits to recover *eip* and *ebp* values at handler's end depending on their initial values.

Thus we are able to find the two keys for each handler: the one for the next handler's offset is the result of $(backtrace(eip) - [ebx+14]) \oplus [ebp]$, the key for arguments is found using $(backtrace(ebp) - [ebx+14]) \oplus [ebp+4]$.

One these two keys acquired, we able to follow the instruction flow of the virtual machine.

```
 1  loc_19aa0h:
 2   mov ecx, dword ptr [ebp+10h] ; @19aa0h  8b4d10
 3   xor ecx, 2ce6fc22h           ; @19aa3h  9c81f122fce62cc1c60a..<+20>
 4   cmp dword ptr [ebx+ecx], 0   ; @19ac1h  c1ce0d87cec1c91387ce..<+7>
 5   jz loc_19b57h                ; @19ad2h  0f847f000000  x:loc_19b57h
 6
 7   mov ecx, dword ptr [ebp+0]   ; @19ad8h  538d5d0811c3871c2450..<+18>
 8   xor ecx, 3c606446h           ; @19af4h  8d12565e9c81f1466460..<+8>
 9   mov ebp, dword ptr [ebp+4]   ; @19b06h  9c5031c0668cc83d0900..<+16>
10   xor ebp, 2ce6fc22h           ; @19b20h  538d5b0431eb871c2455..<+30>
11   jmp loc_19b92h               ; @19b48h  89c09c873c24579b9d5f..<+5> x:
12
13  // Xrefs: 19ad2h
14  loc_19b57h:
15   mov ecx, dword ptr [ebp+8]   ; @19b57h  8b4d08
16   xor ecx, 2ce6fc22h           ; @19b5ah  9c6089d0d3c26181f122..<+10>
17   mov ebp, dword ptr [ebp+0ch] ; @19b6eh  c1c80d92c1ca13928b6d0c
18   xor ebp, 2ce6fc22h           ; @19b79h  c1c11287d1c1c20e9087..<+15>
19
20  // Xrefs: 19b48h
21  loc_19b92h:
22   add ebp, dword ptr [ebx+14h] ; @19b92h  036b14
23   add ecx, dword ptr [ebx+14h] ; @19b95h  034b14
24   jmp ecx                      ; @19b98h  518d4b109c19d99d870c..<+21>
```

Fig. 21: Conditional jump

This method works well until the 18^{th} handler, where an error occurs: each key has two possible values.

Manual analysis of this handler (Fig. 21) reveals that it is a conditional jump: if one of the virtual registers contains a non null value, control is normally given to next handler, however, if this value is null, virtual address to execute is encoded in fields 2 and 3 of the instruction.

These two fields are crypted with the key *key_args* of the handler.

We actually have two choices:

- code another pathfinding algorithm;
- re-use *Metasm* disassembly engine.

By looking at the first handlers we have just detected, their simplicity turns us towards the second solution.

I know there's an answer. The fundamental element of this approach is a generic handler analysis method.

The method is composed of the following steps:

1. Handler's disassembly, using the underlying native disassembler.
2. Examination of the handler's form:
   - How many basic blocs are there ?
   - How these blocks are laid out ?

   - How many exit points ?
3. Handler's effects analysis:
   - Which are the modifications on native processor's registers ?
   - Which are the modifications on memory ?

Tracing native register modifications is implicitly done by *Metasm* backtracking functionality: for each exit point, we list modifications applied on each register compared to the beginning of the handler.

The analysis of effects on memory is not so straightforward: one needs to go through the handler again, instruction by instruction, and to backtrace each access when one finds an instruction that writes the memory.

We then get an array listing all the elements modified by the handler and, for each of them, the value by which they would be replaced during execution. We call this the handler's *binding*.

Virtual machine architecture permits many shortcuts radically simplifying the binding expression.

If an handler preserves the *ebx* register value (which, for recall, contains the virtual machine context base address), and if *ebp* and *eip* bindings match the transition sequence between handler, as seen precedently, then we got the two decryption keys of the handler.

Arguments decryption key is then used to define the following symbolic entities:

- *arg0*, which is the first argument of the instruction, considered as an integer (*[ebp+8]* $\oplus$ *arg_key*),
- *reg0*, which stands for the first argument used as a virtual register index (*[ebx+arg0]*),
- *reg0b*, which also stands for the first argument used as a virtual register index (*byte ptr [ebx+arg0]*), but seen as a byte (an immediate parallel can be made with *al* with respect to *eax* when dealing with *x86* architecture),
- the same operations are repeated for each remaining arguments: *arg1*, *reg1* etc.

This information permits to identify an handler by comparison to a set of pattern that we define by hand. If none of the patterns matches, handler is tagged as unknown; one then has to define a new pattern covering this case, after a manual analysis.

Trivial patterns describe handlers made of only one bloc: we then sure to have the whole handler's semantic into the binding.

These handlers actually are the standart basic arithmetic operations, and then read/write memory operations (*indirections*).

```
 1  handler_13491h:
 2  // handler type: add reg, reg
 3  // "reg0" <- Expression["reg0", :+, "reg1"]
 4   mov eax, dword ptr [ebp+0ch]  ; @13491h  8b450c
 5   xor eax, 8d3f5d8bh            ; @13494h  9c358b5d3f8d9d
 6   mov eax, dword ptr [ebx+eax]  ; @1349bh  6089e8d3c26150c1c502..<+79>
 7   mov ecx, dword ptr [ebp+8]    ; @134f4h  89ed9c873424569b9d5e..<+3>
 8   xor ecx, 8d3f5d8bh            ; @13501h  c1c10a5156e800000000..<+29>
 9   add dword ptr [ebx+ecx], eax  ; @13528h  6089c0d3c26101040b
10   mov eax, dword ptr [ebp+0]    ; @13531h  558734245d87f58b4500
11   xor eax, 6f9078cch            ; @1353bh  c1c30a5350e801000000..<+69>
12   mov ebp, dword ptr [ebp+4]    ; @1358ah  c1c3099c01de9b29de8d..<+10>
13   xor ebp, 8d3f5d8bh            ; @1359eh  9c52e800000000818424..<+57>
14   add ebp, dword ptr [ebx+14h]  ; @135e1h  558d6b109c19c59d872c..<+22>
15   add eax, dword ptr [ebx+14h]  ; @13601h  c1e620034314
16   jmp eax                       ; @13607h  ffe0
```

Fig. 22: Result of an automatic handler analysis: an addition

Many other handlers call natives functions. These call always use a function pointer table, initialised when the driver is loaded.

The following functions are referenced into this table:

- *ExAllocatePool* : memory allocation,
- *ExFreePoolWithTag* : memoire free,
- a region of the driver, filled with zero (never called),
- a function displaying a debug string, using *vsprintf* and *DbgPrint* (never called),
- *MmGetSystemRoutineAddress*: retrieves the address of an exported system function from its name,
- a driver's function implementing a MD5 has.

We know the semantic of each of these function, and so the whole handler's semantic.

MmGetSystemRoutineAddress may has been problematic; but, in practice, it happens the all handlers that call it, use it to get the address of the native function *KdDebuggerEnabled*, with the purpose of crashing the process of a debugger is detected.

The sequence responsible for the crash (Fig. 23, l. 20), has been reduced by the deobfuscator.

Initially it was a random jump on the result of a *rdtsc*.

The pattern checks that the function whom we get the address actually match the address of *MmGetSystemRoutineAddress* function. If the test is positive, this handler tagged as *trap*, otherwise, it is treated as unknown.

Among remaining handlers, four are more complex to analyse, as they involve conditional jumps.

- A virtual conditional jump, which jumps on a virtual address or another function according to the nullity of a virtual register.

```
 1  handler_13fb6h:
 2  // handler type: trap
 3  // "call_arg0" <- Expression[81929]
 4  // "call" <- Expression[Indirection[[Indirection[[:ebx, :+, 20]..
 5   jmp loc_1401bh                ; @13fb6h  9c6..  x:loc_1401bh
 6  db "KdDebug"                  ; @14009h
 7  db "gerEnabled", 0            ; @14010h
 8  loc_1401bh:
 9   call loc_14020h              ; @1401bh  e80..  noreturn x:loc_14020h
10  loc_14020h:
11   pop eax                      ; @14020h  58
12   sub eax, 17h                 ; @14021h  2d17000000
13   push eax                     ; @14026h  50
14   mov eax, dword ptr [ebx+14h] ; @14027h  508b4314508b44240458..<+4>
15   call dword ptr [eax+18h]     ; @14035h  ff5018
16   cmp byte ptr [eax], 1        ; @14038h  803801  r1:unknown
17   jnz loc_14043h               ; @1403bh  7506  x:loc_14043h
18
19  loc_1403dh:
20   jmp loc_1403dh               ; @1403dh  0f3101c8ffe0  x:loc_1403dh
21
22  loc_14043h:
23   mov eax, dword ptr [ebp+0]   ; @14043h  8b4500
24   xor eax, 45f341a7h           ; @14046h  351465ef9335b3241cd6
25   mov ebp, dword ptr [ebp+4]   ; @14050h  8b6d04
26   xor ebp, 0b7048be8h          ; @14053h  81f5d4401e619b81f53c..<+3>
27   add ebp, dword ptr [ebx+14h] ; @14060h  036b14
28   add eax, dword ptr [ebx+14h] ; @14063h  034314
29   jmp eax                      ; @14066h  ffe0
```

Fig. 23: Kernel debugger detection check

– Three categories of handlers which define the value of a virtual register to 0 or 1, in accordance to the fact that its initial value is, respectively, greater, lower or equal than the value of another virtual register.

For these cases, we use a little heuristic, in order to keep the code concise: we look at the native instruction used for the conditional jump that we find in the implementation of the handler.

Finally, for the two last type of handlers, one is an indirect jump which loads the offsets of the next handler and next instructions from the values of two virtual registers. The last one is more complex than others: it involves a loop and seems to implement a kind of decryption routine (Fig. 24).

```
 1  handler_23c8fh:
 2  // handler type: decryptcopy reg, imm, imm, imm
 3   mov edi, dword ptr [ebp+8]    ; @23c8fh  50578d00578d1b50585f..<+138>
 4   xor edi, 2c1a83c1h            ; @23d23h  9c81f7c1831a2c9d
 5   push dword ptr [ebx+edi]      ; @23d2bh  ff343b
 6   pop edi                       ; @23d2eh  5f
 7   mov esi, dword ptr [ebp+0ch]  ; @23d2fh  8b750c
 8   xor esi, 2c1a83c1h            ; @23d32h  9c81f6c1831a2c9d
 9   add esi, dword ptr [ebx+14h]  ; @23d3ah  037314
10   mov eax, dword ptr [ebp+10h]  ; @23d3dh  8b4510
11   xor eax, 2c1a83c1h            ; @23d40h  9c35c1831a2c9d
12   mov ecx, dword ptr [ebp+14h]  ; @23d47h  8b4d14
13   xor ecx, 2c1a83c1h            ; @23d4ah  9c81f1c1831a2c9d
14
15  loc_23d52h:
16   mov edx, dword ptr [(esi+(4*ecx))+(-4)]      ; @23d52h
17   xor edx, eax                  ; @23d61h  31c2
18   mov dword ptr [(edi+(4*ecx))+(-4)], edx      ; @23d63h
19   rol eax, cl                   ; @23d67h  505188c9d34424045958
20   add eax, ecx                  ; @23d71h  01c8
21   loop loc_23d7ah               ; @23d73h  e205  x:loc_23d7ah
22   jmp loc_23d7fh                ; @23d75h  e905000000  x:loc_23d7fh
23
24  loc_23d7ah:
25   jmp loc_23d84h                ; @23d7ah  e905000000  x:loc_23d84h
26
27  loc_23d7fh:
28   jmp loc_23d89h                ; @23d7fh  e905000000  x:loc_23d89h
29
30  loc_23d84h:
31   jmp loc_23d52h                ; @23d84h  e9c9ffffff  x:loc_23d52h
32
33  loc_23d89h:
34   mov eax, dword ptr [ebp+0]    ; @23d89h  8b4500
35   xor eax, 0a9c47c96h           ; @23d8ch  351d38ce7c358b440ad5
36   mov ebp, dword ptr [ebp+4]    ; @23d96h  8b6d04
37   xor ebp, 2c1a83c1h            ; @23d99h  81f5c52816f99b81f504..<+3>
38   add ebp, dword ptr [ebx+14h]  ; @23da6h  036b14
39   add eax, dword ptr [ebx+14h]  ; @23da9h  034314
40   jmp eax                       ; @23dach  ffe0
```

Fig. 24: Decryption handler

This handler actually accepts four arguments: a register containing the address of a destination buffer, an integer which is an offset in the *.data* section, another integer standing for a size (in *dwords*), and a last one, used as decryption key. It then copies the data from the *.data* section to the destination buffer, after xoring them with the key. The key is modified at each round, using a shift and an addition implying the index of the next dword to decrypt (l.19 et 20).

The bugle sounds as the charge begins ! Armed with this information, we can now associated to each of our handlers:

- a virtual opcode name, to display the assembly listing,
- a list of symbolic arguments, to decode and interpret the arguments for each virtual instruction,
- a binding which express instruction's effects on the virtual processor's context,
- the two encryption keys (when existing), to decode the arguments and to follow the execution flow.

These data can be calculated once for all: we backup them into a cache file in order to speed up the script. Actually handlers native code disassembling and deobfuscation is the most time consuming step.

For information, cache initialisation for all handlers (112) lasts almost 15 minutes on a standart configuration, while the whole treatment as describes in this paper, with an already filled cache, lasts less than 30 seconds.

The idea to dynamically build a ruby class using this automatic analysis method to interpret the virtual instruction handlers on the fly.

This class will be used as *CPU* for the standart *Metasm* disassembly engine, in order to use it in a transparent way on the virtual code.

It works in accordance with the following description.

First, we define a virtual space of code, where an instruction address is the couple *(handler's address, instruction's address)*. Such an object, standing for the first virtual instruction, is passed to *Metasm* as entry point of a program, whom the *CPU* is an instance of the aforesaid *T2CPU* class.

This cpu contains a reference to an instance of standart *Disassembler*, the same the we have used to generate listings used as examples in this article.

As we have seen in the part introducing *Metasm*, the disassembler ask to the cpu to decode and analyse the instruction at the current address, update this address et so on. This is where intersting things begin.

When a decoding request is received, our virtual process analyse it automatically to determine instruction to send back, in addition to its effects; in particular next instruction's address.

Thus, in a transparent way, *Metasm* disassembles each of virtual instruction like a classical program, providing us backtracking features on virtual registers.

The obtained listing obtained thanks to this step is already remarkable (Fig. 25).

```
 1  entrypoint_219feh_21ea6h:
 2      nop                          ;  @219feh_21ea6h
 3      mov r68, 28h                 ;  @138fah_35748h
 4      add r68, host_esp            ;  @1501dh_2adc7h
 5      mov r64, dword ptr [r68]     ;  @175e6h_38670h r4:dword_host_esp+28h
 6      mov dword ptr [esp], r64     ;  @156d1h_368a2h w4:dword_ctx+101d0h
 7      mov r64, 4                   ;  @13184h_34e02h
 8      add esp, r64                 ;  @15e23h_34d28h
 9      mov r68, 2ch                 ;  @138fah_37a96h
10      add r68, host_esp            ;  @15336h_3a431h
11      mov r64, dword ptr [r68]     ;  @18231h_31642h r4:dword_host_esp+2ch
12      mov dword ptr [esp], r64     ;  @16f1bh_36aa2h w4:dword_ctx+101d4h
13      mov r64, 4                   ;  @13626h_2fca1h
14      add esp, r64                 ;  @13491h_3594eh
15      trap                         ;  @18c00h_35b6eh
16      mov ebp, esp                 ;  @22d86h_34262h
17      mov r64, 234h                ;  @13184h_1c968h
18      trap                         ;  @1bf14h_36f9ah
19      add esp, r64                 ;  @1501dh_2b162h
20      trap                         ;  @15a3ch_2a44fh
21      mov r78, 200h                ;  @138fah_2f305h
22      trap                         ;  @14121h_3a473h
23      add r78, ebp                 ;  @1501dh_3402ah
24      mov r64, dword ptr [r78]     ;  @175e6h_35d98h r4:dword_ctx+103d8h
25      xor r64, 1                   ;  @17f53h_37befh
26      jrz loc_2d630h_2d8ffh, r64 ;  @19aa0h_1c6ffh x:loc_2d630h_2d8ffh
27      mov r68, 0ch                 ;  @13184h_394ebh
28      syscall_alloc_ptr r64, r68 ;  @25d49h_35b2fh
29      mov r78, 200h                ;  @138fah_2926ch
30      add r78, ebp                 ;  @15e23h_32249h
31      mov dword ptr [r78], r64     ;  @15fc3h_3119ch w4:dword_ctx+103d8h
32  loc_2d630h_2d8ffh:
33      decryptcopy r64, 100h, 751734b1h, 3 ;  @2d630h_2d8ffh w4:unknown
34      mov r78, 0                   ;  @13184h_1aab2h
35      add r78, ebp                 ;  @15336h_2d1abh
36      mov dword ptr [r78], r64     ;  @156d1h_2a854h w4:dword_ctx+101d8h
37      mov r78, 0                   ;  @13626h_3a2ebh
38      add r78, ebp                 ;  @15336h_33a37h
39      mov r64, dword ptr [r78]     ;  @175e6h_2e159h r4:dword_ctx+101d8h
40      [...]
```

Fig. 25: Virtual machine code

We observe that it is a very low level assembly, using for example many instructions to do the equivalent of a *push*. Instructions also seem to only manipulate variable on the stack.

Chronologically, it is at this time that we have been able to assign a name and a role to each of the virtual registers.

By looking at handlers cache file, we notice that most of them are duplicated: there are for example four handlers able to perform an addition between two virtual registers, with no semantic differences.

4.5 Macro assembler

By filtering *nop*s and others *traps*, we quickly come to the conclusion that the virtual assembly language that we seen is the result of a macro-instruction oriented programmation.

Indeed we find again and again identical instructions sequences, with only few exceptions; these sequences are contiguous and perfectly cut out the text in elementary blocks.

The work needed to rebuild macro-instructions from the listing is quite similar to what have been done to handle the deobfuscation process of driver's code: concatenate several contiguous instructions in another one, express the same semantic in a concise way.

Here, pattern is really simple: it mainly consists of spotting an address in a register and then resolving an indirection; in practice these operations only involve many *mov* and *add* instructions.

As macro-instructions do not have a precise definition, we are free to use unusual constructions, involving many memory references in the same instruction, or involving an indirection degree greater than one; which is classically forbidden in real assembly language.

We also have to our disposal all information necessary to decipher parts of *.data* section used by *decryptcopy* instructions; the optional pass permits to make explicit many strings.

At this stage, the listing is quite concise and has a satisfying readability (Fig. 26).

Functions calls. We then recognise a quite intersting pattern: indirect jump instruction is systematically used to execute an instruction whom address has been pushed on the stack few instructions before. This remains ourself a classical function call convention.

This one is a bit strange: first the return address is manually pushed onto the stack, then frame pointer is backuped and finally arguments are pushed; after which the exectution flow follows the subfunction's code.

Typically, arguments are push before return address, and the the execution flow is modified to enter into the function.

The epilogue is also distinctive: code first removes reserved space on the stack for arguments and local variables, then it checks if stack pointer is equal to its initial value (when virtual machine started), by comparing it with the *esp_init*

```
 1  entrypoint_219feh_21ea6h:
 2   mov dword ptr [esp], dword ptr [host_esp+28h] ;  @219feh_21ea6h  r4:d
 3   add esp, 4                                    ;  @13184h_34e02h
 4   mov dword ptr [esp], dword ptr [host_esp+2ch] ;  @138fah_37a96h  r4:d
 5   add esp, 4                                    ;  @13626h_2fca1h
 6   mov ebp, esp                                  ;  @18c00h_35b6eh
 7   add esp, 234h                                 ;  @13184h_1c968h
 8   mov r64, dword ptr [ebp+200h]                 ;  @15a3ch_2a44fh
 9   xor r64, 1                                    ;  @17f53h_37befh
10   jrz loc_2d630h_2d8ffh, r64                    ;  @19aa0h_1c6ffh  x:loc_2d630h_2d8f
11   syscall_alloc_ptr r64, 0ch                    ;  @13184h_394ebh
12   mov dword ptr [ebp+200h], r64                 ;  @138fah_2926ch
13  loc_2d630h_2d8ffh:
14   decryptcopy r64, "\000\000\000\000\000\000\000\000\376\324\004"
15   mov dword ptr [ebp], r64                      ;  @13184h_1aab2h
```

Fig. 26: The same listing using macro-instruction abstraction-level.

value. If values match, the code exits from VM and gives back control to non-obfuscated driver's code, which itself sends back the answer to the user; when they differ, the frame pointer and the return address are popped and control is given back to the caller (Fig. 27).

Integrating these macros into *Metasm* permits to accurately handle subfunctions; therefore, we cover a much more extensive virtual code sequence.

We now have a classical code design: few basic functions, like a *strlen*-like function and also three others more substantial functions.

By displaying complete driver's code once disassembled, and after having tagged bytes encoding virtual instructions, it is possible to check that we actually have interpreted the whole binary; expect for few very short random bytes sequences inserted between handlers.

This phase is thus successful.

We are at last able to focus ourselves on the code to get an idea of the implemented algorithm.

The code really looks like to some C code, compiled without optimisation (or even unoptimised). All operations are done on the stack, using space reserved for local variables.

It is also quite redoundant: many values are copied unnecessarily in different temporary variables on the stack, before reaching their final destination.

We rename variables according to their offset in the *stack frame*: *dword ptr [ebp+128h]* is seen as *var128*, always for the sake of readability (Fig. 28).

4.6 Decompilation

Due to the multiplication of temporary variables, the code is still tedious to read.

```
 1   mov r68, 4                          ;  @138fah_353c6h
 2   mov dword ptr [esp], 2507dh         ;  @13626h_2fa7eh
 3   add esp, r68                        ;  @15e23h_1e097h
 4   mov dword ptr [esp], 14cch          ;  @13184h_3302ch
 5   add esp, r68                        ;  @1501dh_29749h
 6   mov dword ptr [esp], ebp            ;  @16a01h_22b17h
 7   add esp, r68                        ;  @15e23h_32514h
 8   mov dword ptr [esp], dword ptr [ebp] ;  @13184h_2be36h
 9   add esp, r68                        ;  @13491h_351c1h
10   mov dword ptr [esp], dword ptr [ebp+4] ;  @13bf0h_38c9eh
11   add esp, r68                        ;  @15336h_39e68h
12   mov ebp, esp                        ;  @1d735h_386abh
13   add esp, 204h                       ;  @13626h_323b7h
14
15   [function body]
16
17   add esp, -20ch                      ;  @13626h_32000h
18   mov r64, esp                        ;  @22174h_34eeah
19   seteql r64, esp_init                ;  @1fc6eh_3629ah
20   xor r64, 1                          ;  @19357h_21febh
21   jrz loc_272dah_35a76h, r64          ;  @1ab4fh_32340h
22   add esp, -4                         ;  @13184h_369a0h
23   mov ebp, dword ptr [esp]            ;  @187beh_251a3h
24   add esp, r64                        ;  @15e23h_378dch
25   mov r68, dword ptr [esp]            ;  @175e6h_39ee2h
26   add esp, r64                        ;  @1bf14h_1c1edh
27   mov r6c, dword ptr [esp]            ;  @175e6h_39419h
28   jmp r68+13000h_r6c+13000h           ;  @23fb0h_381e2h
29
30  loc_272dah_35a76h:
31   syscall_free_exitvm                 ;  @272dah_35a76h
```

Fig. 27: Subfunction call and return macros.

```
 1   syscall_alloc_ptr r64, 24h          ;  @138fah_2f2f0h
 2   mov var20c, r64                     ;  @13184h_3924eh
 3   decryptcopy r64, "p0wered by T2 - http://www.t2.fi\000?\365\256"
 4   mov var4, r64                       ;  @13bf0h_354a1h
 5   call sub_13626h_323b7h, var0, var4 ;@138fah_353c6h x:sub_13626h_323b7h
 6   mov var0, var21c                    ;  @144cch_3807dh
 7   call sub_138fah_33176h, var0        ;  @138fah_23c63h x:sub_138fah_33176h
 8   mov var0, retval                    ;  @168f4h_38615h
 9   mov var4, 10h                       ;  @13184h_399eeh
10   mov r64, var0                       ;  @13184h_376f2h
11   seteql r64, var4                    ;  @1486dh_22be1h
12   xor r64, 1                          ;  @17f53h_181f5h
13   mov var0, r64                       ;  @13bf0h_37a11h
14   jrz loc_13184h_3276fh, var0         ;  @14121h_2c78dh x:loc_13184h_3276fh
15   add esp, -23ch                      ;  @138fah_1fbe1h
16   syscall_free var200                 ;  @138fah_311e6h
```

Fig. 28: Macro-instruction virtual code and local variable.

The idea is now to cut ourselves off from the implementation, and to try to express the whole semantic, and not individual instructions. This seems feasible as each of these instructions is really simple, without side-effect, and instruction set is tiny.

It also happens that *Metasm* includes a C compiler, and thus has to its disposal all the objects necessary to manipulate code in this language. We will try to generate transcription of the listing into C code.

Ideally, the goal is to transcribe the program functionally, ignoring most of implementation details; at first, we have to define significant actions from useless ones. We will completely ignore register modifications and stack variables, and only put attention for:

- functions calls, both internal and external, including their arguments,
- memory writings (outside the stack),
- and the predicated associated with conditional jumps.

This very simplified approach is ineffective for a concrete advanced language, as data on the stack are meaningful: there are buffers, structures (in the C language meaning of the term).

There also are problems linked with operating systems interactions, in particular threads or signals, without speaking about exceptions handling and others joys...

In practice, we will content ourselves with working at instruction blocks level, which is by far simpler than a global program approach, and provides quite satisfying results.

We will use a recursive approach, function oriented, as one goes along through the code from an entry point: when a function call is encountered, it triggers the analysis of this function. This approach supposes that main code segment is a function; this assertion is most of times verified.

The main objective is to reduce all intermediate assignments to stack variables. Thus, we need to ascertain which one are significant.

In a first pass on the code, we mark for each of them, which variable is read and which is written. Then using function control flow graph, we know which one we need to keep: the ones that are read in another block without being overwritten in the meantime.

The Clairvoyant. We transform each basic block into its equivalent in C (Fig. 29): arithmetic operations are merged in a way that conveys the whole bloc binding with respect to a significant variable expressed as a unique C expression; jumps are translated into *goto* and conditional jumps into *if (..) goto label;*.

We are not interested in variables' type for the moment, we will just considered them as integer (*int*); however, we keep indirections' type: for the ones referencing a byte, type is *char*, *int* for others.

```
 1  void sub_13626h_323b7h(int arg4, int arg0)
 2  {
 3    int var0;
 4    int var200;
 5  sub_13626h_323b7h:
 6    var200 = 0;
 7    var0 = var200;
 8  label_13184h_31e59h:
 9    if (*((char*)(var0 + arg0)) == 0)
10      goto loc_13bf0h_1aa3bh;
11    *((char*)(arg4 + var200)) = *((char*)(var200 + arg0));
12    var200 += 1;
13    var0 = var200;
14    goto label_13184h_31e59h;
15  loc_13bf0h_1aa3bh:
16    *((int*)(arg4 + var200)) = 0;
17    return;
18  }
```

Fig. 29: Preliminary decompilation phase

There's a time to live and a time to die. Next phase, and probably the most improtant, is the recognition of C standard control structures: *if*, *if/else* et *while*.

Let's start with the easiest: *if*.

while browsing the execution flow, if we encountered a conditional jump (ie. a *if (..) goto label;*) whom label is located further in the function, we transform it by inverting the *if* condition and replacing the goto by the whole code located between the *if* and the label.

One needs to repeat the same treatment on what is now the *then* block, in order to handle imbricated tests.

We quickly deduce how to handle *if/else* structures: if the *then* bloc, freshly discovered ends with a *goto label* whom destination is in the code which remains to analyse, we can remove the goto and move the sequence between *if* end and the label in the *else* block.

The obtained code is clearer, but a pattern appears like a *if/else* (Fig. 30): a *then* which contains a label and ends with a *goto*; moreover code following the *if* jumps on this label. A test is added to correctly handle this case (Fig. 31).

The underlying code is quite repetitive; which results, among others, by some *if/else* whom last expression are similar between the *then* and the *else*. We take advantage of this phase to factorise the code and extract it from the *if* structure.

```
1  if (cond) {
2    a;
3  label:
4    b;
5    goto anywhere;
6  }
7  c;
8  goto label;
```

Fig. 30: if/else pattern

```
1  if (cond) {
2    a;
3  } else {
4    c;
5  }
6  label:
7  b;
8  goto anywhere;
```

Fig. 31: Solved if/else pattern

Once all the code have been processed according to this method, *while* handling is simple: it's a label following by a *if* whom the last instruction is a *goto* jumping on this label. Few additional tests permit to recognise the associated *continue* and *break* as well.

Finally, a last cosmetic pass is done, in order to remove unused labels in the code. The result is now very satisfying (Fig. 32).

```
1  void sub_13626h_323b7h(int arg4, int arg0)
2  {
3    int var0;
4    int var200;
5    var200 = 0;
6    var0 = var200;
7    while (*((char*)(var0 + arg0)) != 0) {
8      *((char*)(arg4 + var200)) = *((char*)(var200 + arg0));
9      var200 += 1;
10     var0 = var200;
11   }
12   *((int*)(arg4 + var200)) = 0;
13   return;
14 }
```

Fig. 32: Intermediate decompilation phase

Project II. The last phase consists of determining variables' type.

A first pass permits to notice which are the variables assigned with immediate integers: these ones are typed as *int*.

Then we look for indirections with *type casting*, which, coupled with the integers list, permit to determine the ones typed as pointer. Moreover, size of referenced data as used to guess the type of pointed data.

A final pass remains to do, to correct *cast* sequences; one also needs to fix pointer addition, substraction.

Final result goes beyond our expectations (Fig. 33).

```
 1  void sub_13626h_323b7h(char *arg4, char *arg0)
 2  {
 3    int var0;
 4    int var200;
 5    var200 = 0;
 6    var0 = var200;
 7    while (arg0[var0] != 0) {
 8      arg4[var200] = arg0[var200];
 9      var200 += 1;
10      var0 = var200;
11    }
12    *((int*)(arg4 + var200)) = 0;
13  }
```

Fig. 33: Final decompilation phase

We thus obtain a complete listing of the obfuscated algorithm, in only 352 lines, which can manually be reduced to as few as 200 lines.

This is now the time to have a moved thought for art craftmen, goldsmiths of reverse, who have solved the challenge by hand...

For recall, native code consists of almost 40.000 obfuscated instructions which implement 112 handlers used by 3.000 virtual instruction, the whole designed to be painful to read.

4.7 It's a trap !

The final examination of C code reveals that even at this abstraction level, challenge's designers still have affection gesture for us.

We discover few strange sequences (Fig. 34).

Decryptcopy is used to decrypt two strings in the original binary (here in clear), pointed by *var20c* and *var210*. *var210* is clearly a ciphered string, which is deciphered into a buffer passed as argument. Deciphering key, is calculated at

```
 1  void sub_13626h_38b14h(char *arg4, int arg0)
 2  {
 3    var20c = malloc(20);
 4    decryptcopy(var20c, "Q\213D$\b\213L$\f\323\350Y"
 5        "\302\b\000\000\000\006\271\004");
 6    var210 = malloc(20);
 7    decryptcopy(var210, "X\244{\022\322\246\023\\|"
 8        "\350\350\201\210\000\000\000\000_\335\271");
 9    var214 = 0;
10    while (var214 < 13) {
11      r64 = ((int(*)(int, int))var20c)(arg0, var214);
12      arg4[var214] = var210[var214] ^ r64;
13      var214 += 1;
14    }
15    *((int*)(arg4 + 13)) = 0;
16    [...]
```

Fig. 34: Native shellcode

the line 9, in a strange way: the string pointed by *var20c* is called as if it was a function body.

```
 1  entrypoint_0:
 2  // function binding: eax -> (dword ptr [esp+4]>>dword ptr [esp+8]), esp
        -> esp+0ch
 3  // function ends at 0ch
 4   push ecx                        ; @0  51
 5   mov eax, dword ptr [esp+8]      ; @1  8b442408
 6   mov ecx, dword ptr [esp+0ch]    ; @5  8b4c240c
 7   shr eax, cl                     ; @9  d3e8
 8   pop ecx                         ; @0bh  59
 9   ret 8                           ; @0ch  c20800  endsub entrypoint_0
10  db 0, 0, 6, 0b9h, 4             ; @0fh
```

Fig. 35: Shellcode pointed by *var20c*

It happens that this native shellcode is quite basic (Fig. 35), it returns its first argument shifted to the right by a number of bits specified using its second argument.

In this case, the first argument is one of the function parameters we study, and the second is an index in the string being decoded.

The use of native code, in such a hijacked way, suggests that others ambushes are waiting for us.

It is confirmed precisely in the function's epilogue. (Fig. 36).

```
1    *((int*)(arg4 + 13)) = 0;
2    var208 = malloc(8);
3    decryptcopy(var208, "1\300f\214\310\303\0000");
4    r64 = ((int(*)(void))var208)();
5    *((int*)(arg4 + 14)) = r64 + -8;
6    free(var20c);
7    free(var210);
8    free(var208);
9  }
```

Fig. 36: Killing zone...

Once again, a shellcode is used in the algorithm. Here, the returned value is used to fill a part of the string deciphered by the function.

```
1  entrypoint_0:
2  // function binding: eax -> cs, esp -> esp+4
3  // function ends at 5
4   xor eax, eax  ; @0  31c0
5   mov ax, cs    ; @2  668cc8
6   ret           ; @5  c3  endsub entrypoint_0
7  db 0, 30h      ; @6
```

Fig. 37: ... firefight

The shellcode is still really basic, however its goals seems much less friendly: it is a test to check that code effectively run in ring 0 (Fig. 37), as the ones find during desobfuscation part.

This time, the test is quite pernicious should we say: if the test is negative (ie. we are in ring 3), *cs* is equal to 8. The countermeasure is not a straightforward crash but like we were used to, but it subtly modifies the result of a function, surely crucial for the algorithm.

This kind of trap is most of time tedious to detect and to tamper, as effects are visible much later during program execution.

Another deceit lies in another function (Fig. 38).

This function takes as arguments a string, applies a *MD5* hash on it, and returns a value derived from the hash. However a small modification, apparently insignificant, comes to put a spoke in our wheel: things happen on lines 12 and 13 of the listing.

They check the address returned by malloc, and looks for a positive value. Actually this is were the bias lies, as the value is signed. Once again, it the same kind of ring level execution test.

```
 1  int sub_13bf0h_2b788h(int arg0)
 2  {
 3    register int r64;
 4    register int retval;
 5    int *var200;
 6    int var204;
 7
 8    r64 = malloc(16);
 9    md5(r64, arg0, 8);
10    var200 = r64;
11    var204 = var200[0] ^ var200[1] ^ var200[2] ^ var200[3];
12    if (r64 > 0)
13      var204 += 1;
14
15    free(var200);
16    retval = var204;
17    return retval;
18  }
```

Fig. 38: Another ring 0 test.

4.8 So long, and thanks for all the fish

Once these last pitfall crossed, we can reconstitute the while algorithm.

1. Password's length should be equal to 16 chars.
2. 3 integers $(h1, h2, h3)$ are deducted from the password. Actually it uses a base64 encoding, with a personalised base.
3. This 3 integers have to fulfil few requirements: one should get "T2" by xoring the high and the low words of one of them; the third should be equal to the MD5 hash of the two first, xored by itself ($sub\_13bf0h\_2b788h(md5(h1 \oplus h3, h2 \oplus h3)) == h3$).
4. Password should not contains chars $+$ or $/$.
5. Once this requirements fulfilled, the first integer is used as a key to decipher, an hardcoded string.
6. This string MD5 hash value is check as a final test.

By studying how the final string is deciphered, we discover that only 20 bits of the key are significant. This value is easily bruteforceable: a tiny program coded in assembly tests all the key in a fraction of a second.

One ends up at finding that only one integer deciphered the string passes the MD5 test; this string is *"t207@owned.by"*.

Thus we know 20 bits of the first integer.

The "T2" test also provides us with 16 additional bits for the second integer; the relationship with the xor and the MD5 also permit us to deduce the third bit of the two first integers.

Finally, the test carried out by this algorithm is really slack, it accepts a great number of possible solutions (2^{28} when ignoring test on invalid characters).

Epitaph. To conclude, the challenge revealed quite a good level, and was particularly interesting.

Solving it, in a purely static way was an exciting challenge which has led to many improvements to *Metasm*

We are awaiting the 2008 edition forward!

5 Securitech 2006: a structural approach

The last part of this article is dedicated to an obfuscation technique that we have already discussed quickly: structural obfuscation or control flow graph obfuscation. To illustrate our talks , we have chosen the challenge No. 10 of the **2006 Securitech**. This challenge was proposed by Fabrice Desclaux.

The binary. It is a `Win32` executable; it takes a string as input and uses it to generate an output which seems to include a hash. The goal consists off finding the input that produces a given output. Binary's main function is massively obfuscated, preventing or dramatically slowing down all attempts to reverse it.

Initial goal was to force the challengers to solve it using a black box approach, without access to the implementation. Our goal is to eliminate this obfuscation layer and to recover the exact algorithm.

5.1 Control flow graph definition

The control flow graph is a fundamental structure, used both while the compilation phase and while a possible analysis and disassembly phase. We will briefly remind the main concepts associated with it.

- **Control Transfert Instruction (CTI)**
  A *CTI*, is an instruction, whom the intrinsic nature is to possibly modify the execution flow. This type includes (non exhaustive list): jumps (conditional or not), calls and their counterpart returns, or interruptions. It is important to emphasise that the primary purpose of these instructions is precisely to act on the execution flow; contrary to others instructions, like a *mov*, that may eventually throw an exception (ex: null pointer), and so disrupt execution flow, it can be seen as a side effect, and it is not their intrinsic nature.

- **Basic Block**
  A basic block is a list of contiguous instructions, whom only the first may be the target of a *CTI* and only the last may be a *CTI*. It is the atomic element of the control flow graph. We can make the parallel with a critic section that is impossible to preempt while being executed.

The control flow graph groups together these two notions. Graph's nodes are the basic blocks. Arcs represent the different relationships between these different blocks; they represent a transfer of the execution flow: jump, function call, return, etc. The control flow graph is probably the favourite level of abstraction to quickly and effectively visualise the code's logic: loop, *while*, *do-while*, *if-then-else*... For example `IDA` graph mode, since its version 5.0, is the perfect illustration of the usefulness of this level.

While disassembling, *Metasm* implicitly rebuilds this control flow graph. Indeed, internally, the object *Disassembler* creates and manages *InstructionBlock* objects, which are the implementation of the notion we have just discussed. The arcs are managed with a very fine granularity preserving all information. For example we distinguish a *normal* arc, for example a jump, from an *indirect* arc like the one produced by a function return. To visually take advantage of this abstraction level, a script has been developped to create a bridge between *Metasm* and the graph editor *yEd*[7]. This editor takes as input a *.graphml* file, this file format is based on *XML* and dedicated to graph description.

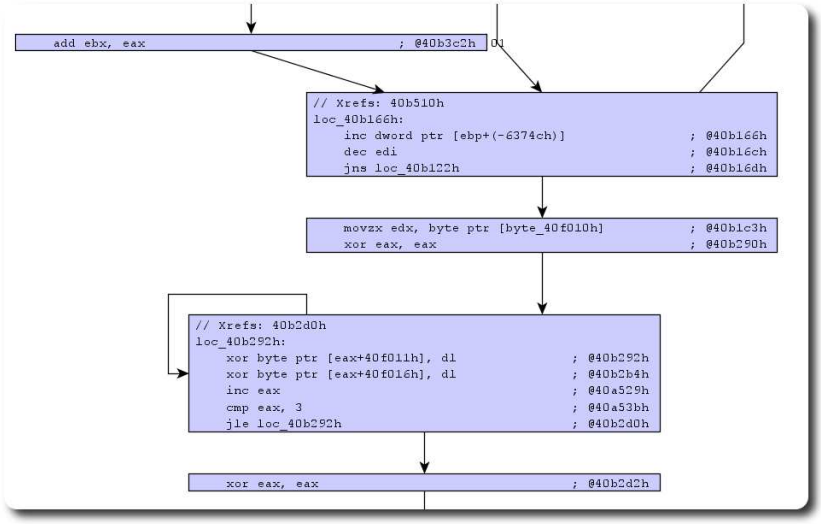

Fig. 39: *yEd* and *Metasm*

5.2 Predicate definition

From a functional point of view, a conditional jump is a predicate following by a connection. When dealing with a *legitimate* conditional jump, a first set of initial states (just call it $\mathbb{A}$) causes the execution of one of the two branches, a second set ($\mathbb{B}$), complementary with the first one, causes the execution of the other branche. $\mathbb{A}$ and $\mathbb{B}$ cover the set of all possible initial states, and are defined according to the predicate. A conditional jump inserted during an obfuscation process is, most of the time, corrupted, and the predicate contains a trap. When it happens, $\mathbb{A}$ and $\mathbb{B}$ sets have remarkable properties.

[7] `http://www.yworks.com/en/products_yed_about.html`

Obscure predicate. An obscure predicate is a boolean function returning always *true* or always *false*, but a priori, we are not able to predict the result. One of our two sets, $\mathbb{A}$ or $\mathbb{B}$, is empty. Actually the predicate has to be complex enough and/or obfuscated, in order to be unpredictable in a trivial way[**?**].

```
1  if( x^4*(X-5)^2  >= 0) {
2    goto real_code;
3  } else {
4    goto no_man's_land;
5  }
```

Fig. 40: Obscure predicate (*pseudocode*)

This few lines of *pseudocode* (Fig. 40) is a good example. Here, the predicate is a small polynomial whom value is always positive or null, so it always returns *true*. Written in `C`, the bias is very easy to guess; however, if we take a look at the same function compiled using *GCC 4.1.2* (Fig. 41).

```
1  loc_8048403h:
2   fstp qword ptr [esp+8]              ; @8048403h  dd5c2408
3   fstp qword ptr [esp]                ; @8048407h  dd1c24
4   call thunk_pow                      ; @804840ah  e8e5feffff
5   fstp qword ptr [ebp+(-20h)]         ; @804840fh  dd5de0
6   mov eax, dword ptr [ebp+(-0ch)]     ; @8048412h  8b45f4
7   sub eax, 5                          ; @8048415h  83e805
8   push eax                            ; @8048418h  50
9   fild dword ptr [esp]                ; @8048419h  db0424
10  lea esp, dword ptr [esp+4]          ; @804841ch  8d642404
11  fld qword ptr [xref_8048590h]       ; @8048420h  dd0590850408
12  fstp qword ptr [esp+8]              ; @8048426h  dd5c2408
13  fstp qword ptr [esp]                ; @804842ah  dd1c24
14  call thunk_pow                      ; @804842dh  e8c2feffff
15  fld qword ptr [ebp+(-20h)]          ; @8048432h  dd45e0
16  fmulp ST(1)                         ; @8048435h  dec9
17  fldz                                ; @8048437h  d9ee
18  fxch ST(1)                          ; @8048439h  d9c9
19  fucompp                             ; @804843bh  dae9
20  fnstsw                              ; @804843dh  dfe0
21  sahf                                ; @804843fh  9e
22  jnbe loc_8048444h                   ; @8048440h  7702
23  jmp loc_8048452h                    ; @8048442h  eb0e
```

Fig. 41: Assembly resulting from predicate function compilation

This implementation makes an heavy use of floatting point computations (using the floatting point unit *FPU*). Recover the semantic of such a part of code while reading it is not immediate. However, in a dynamic approach, an emulator, or a static analysis tool[**?**], should be able to automate the discovery

and characterisation of such a predicate. This example is voluntarily basic, but it is easy to reach much more complex construction.

At last, in our obscure predicate example, the *else* branche is a dead one: it is never executed. To add confusion to this structure, one should duplicate important parts of code into this branch and/or create false references to disrupt disassembly engine, for example using jump or call that target an address in the middle of a *real* instruction (overlapped code obfuscation).

Complete hazard. This new construction differs from the precedent as one branch or the other it is taken indifferently. The predicate is just an hazard source. In order to preserve whole binary semantic, the two branches are semantically equivalent. Code's duplication may be seen as a negative factor, as binary's size increase; that is the reason why duplicated parts are often scaled-down, which quickly reveals *diamond* type constructions.

```
1  if(rand()%2) {
2    real_code_A
3  } else {
4    real_code_B
5  }
```

Fig. 42: Complete hazard (*pseudocode*)

The only requirement of this structure (Fig. 42) is to that **real_code_A** and **real_code_B** are semantically equivalent. We have already tackle this technique, but without explaining it, during the part dedicated to *T2* challenge, (ch 4.2); it was a very tiny part of the protection.

The waterer watered. Conditional jumps inserted to the end of obfuscation reveal, most of times, biased predicates. As such, they present themselves properties liable to be detected and analysed. A tool able to model the mathematical predicate function, will guess the true nature of the conditional jump.

From a developper point of view, it is interesting to hide the true nature of the predicate. Thus, concerning the complete hazard form, he/she could take care to involve significant variables in the calculation of the predicate to deceive the analyst (or a tool). This precaution has in part been taken on the *T2* challenge, but too simply. Actually finding a *RDTSC* instruction involved in the calculation of the predicate, in privileged mode, is really suspicious and finally quickly alerts the analyst.

In the same mood, when using obscure predicates, function should be varied and complex enough, both on paper and in their implementation, to circumvent or slow down all possible forms of analysis.

5.3 Portrait of a man

Before going further, we should recall an essential point. We have been able to build the complete control flow graph thanks to the disassembling quality proposed by *Metasm*. In few words, *Metasm* implemented a virtuous circle, in which the *dataflow* is used to increase our knowledge of the *controlflow*, which itself increase knowledge of the *dataflow*, and so on... To study a protected binary, a simple disassembly engine is not sufficient, the mnemonic, a simple textual translation from the opcode, is not enough. The main asset of *Metasm* is its ability to express in an abstract way, the exact semantic of the instruction, and a step further to backtrack instruction effects over the execution flow, in order to improve the discernment of its disassembling. Once this has been said, we can focus ourselves on the challenge itself. It massively uses obfuscation to protect the algorithm. Actually five main techniques are used.

Uncontional jump insertion. Basic blocks are re-ordered and jumps are inserted between them to preserve the code's semantic. A parallel can be made with a permutation round in a cryptographic algorithm. This technique is effective against an analyst trying to trace step-by-step the code using a debugger for example; s/he will "wander" from one end to another of the executable, and it will be hard to stand back to rebuild a higher level logic. However, graphical control flow graph visualisation tools, like IDA or Metasm with yEd, make it totally ineffective.

Jump emulation. This technique may be seen like an extension of the precedent, a bit more elaborate in its implementation. Basically, it consists off pushing an address on the stack and using a `ret` instruction as a jump to this address.

```
1   push loc_4042fch ; @4027adh  68fc424000
2   ret              ; @4027b2h  c3
```

Fig. 43: *push-ret* used as a jump

The main interest of this construction is that only the knowledge of the semantic of the two instruction, permits to follow the correct execution flow.

False call insertion. Without recalling call convention, we can say that the main property of these inserted calls is that they modify their return address on the stack. This property is quite effective to push some disassembly engines to fail; actually those which, by hypothesis, suppose that the call return to the instruction immediately following the `call` instruction.

```
1  push esi                  ; @401873h  56
2  push ebx                  ; @401874h  53
3  call loc_403592h          ; @401875h  e8181d0000  x:loc_403592h
4  push ebx                  ; @403595h  53
5  add dword ptr [esp+4], 4  ; @403598h  8344240404
6  add esp, 4                ; @409d3eh  83c404
7  ret 8                     ; @40c17bh  c20800  x:loc_40187e
```

Fig. 44: False call skeleton

We have refined this example (Fig. 44) in order to keep only the code specific to this pattern. In practice, these instructions are intertwined with others patterns and *real* instructions. Moreover, this pattern is polymorphic, concerning both the *delta* applied to the return address and the number of registers pushed on the stack. Nevertheless, it is remarkable enough to easily match it. On point to notice, there is dead code between the supposed and real return address, 4 bytes in this example.

Flow duplication. Here is the implementation of a biased predicate, using a complete hazard form. Obfuscation engine selects a part of code, generally limited, duplicates it in the two branches of a conditional jump.

As the two branches are semantically equivalent, whatever the hazard source is, the main point is to make it seem plausible to disrupt the analyst.

Apparent hazard. This one is a direct consequence of the previous. Coupled with the insertion of conditional jumps, we find hazard insertion, here it uses the form of *test* and *cmp* instructions. For recall, these two instructions update the processor's flags, by comparing the two operands.

This example (Fig. 46) summarises the strengthes and weaknesses of this technique. At first glance, it is tempting to seek for the origin of registers *edi* et *ebp* used by the *test* instruction, this results in time waste and confusion. On *IA32* architecture, contrary to others architectures like *ARM* for example, many instructions implicitly update the flags, in particular all arithmetic instructions. In our example, we see that flags, which set the conditional jump *jnz* (l. 6), are overwritten two times by the *add* and *and* instructions. This is the main weakness of this technique, it is relatively easy to filter legitimate comparison instructions using a basic data flow analysis.

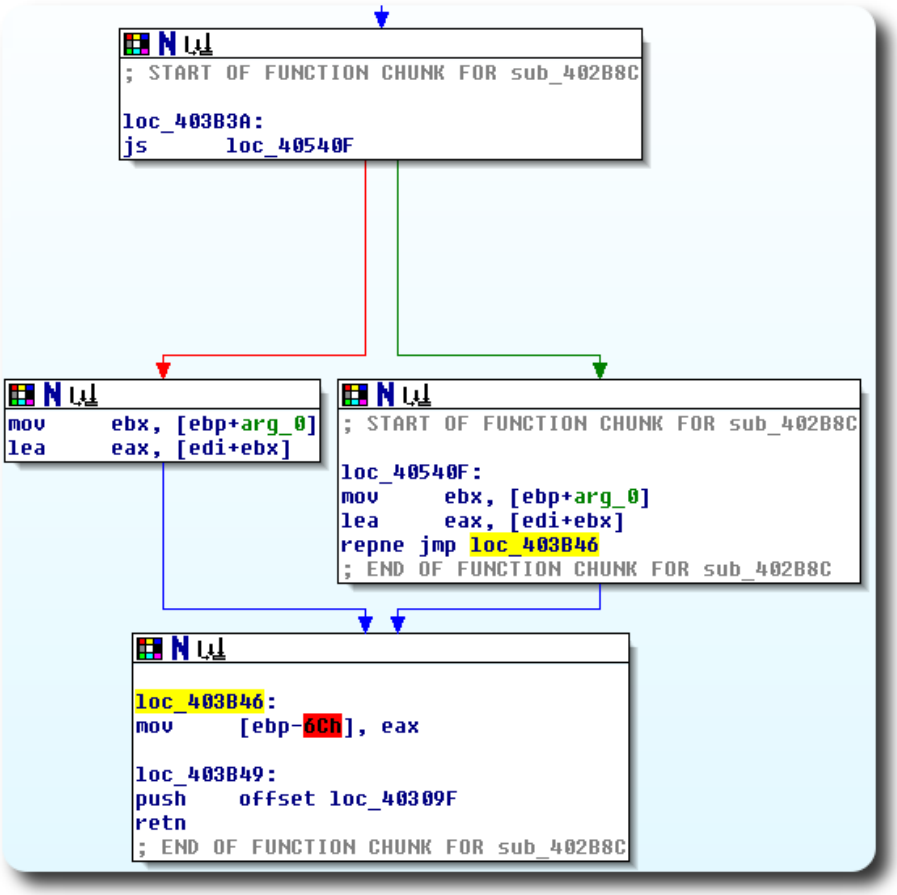

Fig. 45: Flow duplication and false predicate insertion, seen using *IDA*

```
1  402C44h test    edi, ebp
2  402C46h mov     ebx, [ebp+arg_C]
3  402C49h mov     esi, edi
4  402C4Bh add     [ebx], edi
5  402C4Dh and     esi, 3Fh
6  402C50h jnz     short loc_402C5A
```

Fig. 46: *test* instruction insertion

5.4 Control flow graph analysis and factoring

To represent the magnitude of the problem, here's the graph of control of the *main* function's epilogue, as found in the protected binary (Fig. 47).

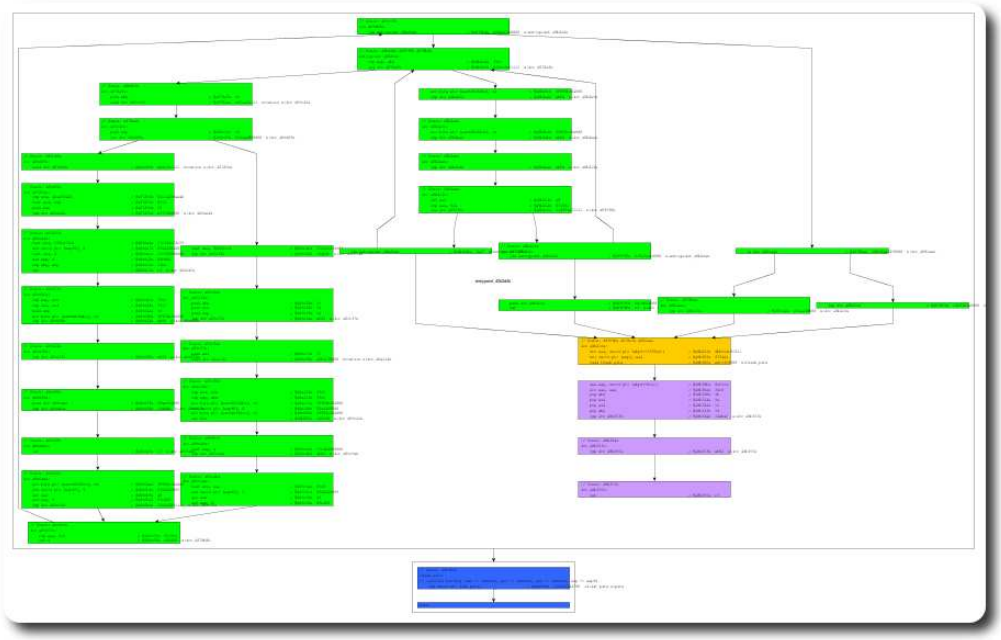

Fig. 47: Protected *main* function epilogue.

Our approach takes advantages of our knowledge of the complete control flow graph. From a given entry point, we linearly follow the execution flow until we met a conditional jump. Once it happens, we build the execution flows associated with each of the two branches. This process is recursive to handle code duplicated many times.

With these two flows in our possession, the treatment is done in few stages:

1. Removal of *test* and *cmp* instructions, according to the method developed precedently. *Metasm* associates each instruction with its semantic, using an abstract expression, allowing to check if an instruction read/write processor's flag.
2. Removal of unconditional jumps inserted to compensate for the basic blocks arrangement trick.

3. Removal of false call, their structure is remarkable enough to be matched. The execution flow is built, finding a `ret` instruction causes the reconstruction of the call stack. Then it is easy to match the pattern to avoid false positives (i.e. removal of *real* code).
4. Comparison of the two flows cleaned. For the sake of simplicity, we have implemented a simple textual comparison of the two flows, instruction by instruction. In the case of a more advanced protection, using poly/metamorphism, it would have been possible to proceed to a behavioural analysis, like what has been done on T2 challenge to recognise handlers' behaviour.

If the two flows are equivalents, we have a duplication structure. Then the false conditional jump is deleted and, more important, the control flow graph is modified: one branche is trashed out. All instructions tagged as "*illegitimate*" are removed from the final listing presented to the analyst. Here is an intermediate result on which we can see that the number of blocks has been divided by approximately 5 (Fig. 48).

The result is already pleasant, but not yet optimal. Parts of the binary have not been cleaned: actually only flows implied in a test of duplication have been cleaned from *junk code*. That is the reason why an additional pass is done on the whole control flow graph, contiguous blocks are also merge to make it more concise. Final result is quite satisfying, we recover the original code completely rid of the protection. Number of basic blocks has been divided by a factor of about 10 (Fig. 49).

On the whole control flow graph, the measured reduction factor on the number of basic blocks is approximately equal to 7.7. Besides number of instruction has also drasticly been reduced: almost 70% of the instructions have been removed from final listing.

5.5 Icing on the cake: interoperability

We have in our hands a disassembly listing quite close from the original, we will benefit from it and rebuild an executable devoid of protection.

Protected binary is a console application which requires very few system libraries: it is possible to port the binary to an *ELF* file format, without too much work.

We replace the *GCC stub* located at the entry point by our own code (Fig. 51).

The original binary starts by allocating a great amount of space on the stack, and the function implementing this allocation is not compatible with Linux and triggers a *SEGFAULT*. Thus, we add a small sequence to our loader that will transfer the stack into the heap before giving the control to the original code.

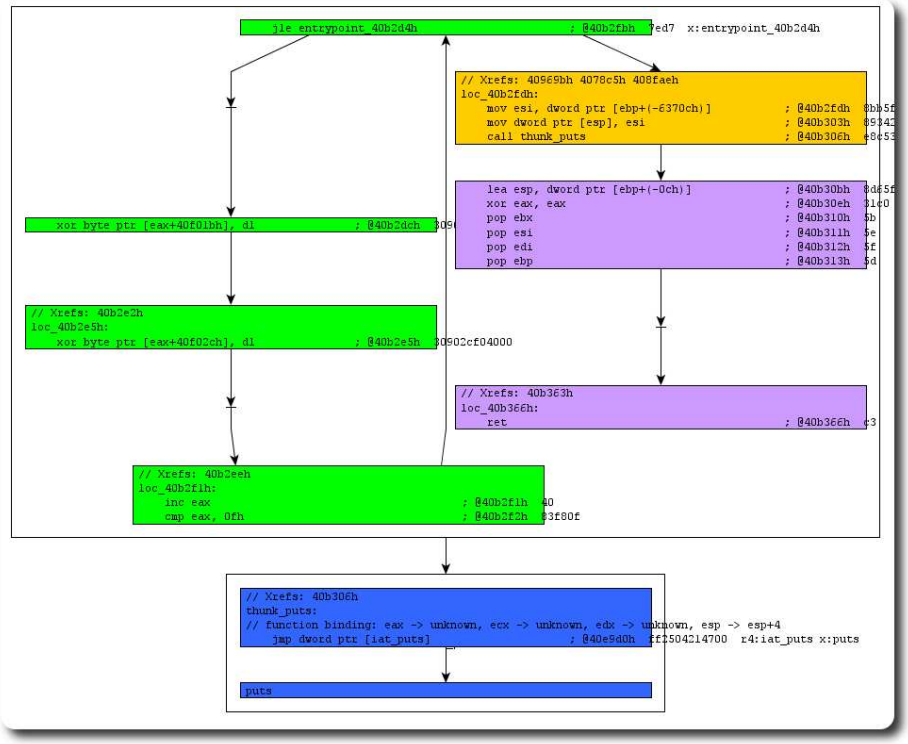

Fig. 48: Epilogue with factored flows.

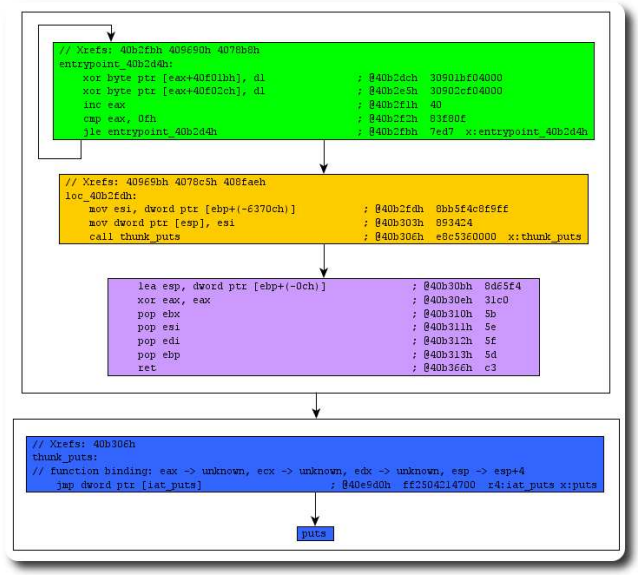

Fig. 49: Epilogue completely cleaned.

A compiler being included into *Metasm*, we do not even need to use an external program to get a binary running on a unix operating system.

Well, basically, it is useless, but it's fun.

5.6 solving

Having a clean code quite close from the original, the challenge is not so hard. The password is split into many blocks, which are manipulated to produce the output, few values are calculated: sum and product of a subgroup of chars, a *CRC* and a *MD5* hash. These different constraints permit, after a little bruteforce, to find the correct solution to the problem. Actually the code contains, various calculations that do not seem to be linked to the generation of the output string, it may be intersting to investigate them.

```
require 'metasm'
include Metasm

# read the binary
file = 'poeut.exe'
pe = PE.decode_file file
pe.cpu = pe.cpu_from_headers

# clean code section compilation
src = File.read('poeut.asm').sub('entrypoint:', '')
pe.parse '.section ".clrtext" rx'
pe.parse '.entrypoint'
pe.parse src
pe.assemble

# labels resolution, etc.
text = pe.sections.last.encoded
text.fixup! 'loc_0' => 0, 'loc_1' => 1
text.reloc.values.map { |r| r.target.reduce_rec }.grep(::String).uniq.
    sort.each { |t|
  if t =~ /(?:dword|byte)_(4\w+)h/ and not text.export[t]
    rva = $1.to_i(16) - pe.optheader.image_base
    s = pe.sections.find { |s| s.virtaddr <= rva and s.virtaddr + s.
        virtsize >= rva }
    s.encoded.add_export t, rva-s.virtaddr
  end
}

# clean binary generation
pe.encode_file 'unpoeut.exe'
```

Fig. 50: Script generating a clean executable

```
 1  .entrypoint hooked_entrypoint
 2  hooked_entrypoint:
 3    ; alloc heap space for 'stack'
 4    push 0x80100
 5    call malloc
 6    add esp, 4
 7    lea ebp, [eax+0x80000]
 8    ; disable libc init (weird things w/ FindAtomA)
 9    or dword ptr [40fc00h], 1
10
11    ; init argc/argv/envp as args for main
12    mov ecx, [esp]
13    mov [ebp], ecx
14    lea eax, [esp+4]
15    mov [ebp+4], eax
16    lea eax, [eax+4*ecx+4]
17    mov [ebp+8], eax
18
19    ; call main w/ new stack
20    mov esp, ebp
21    call loc_403db0h
22
23    ; exit
24    mov eax, 1
25    mov ebx, 0
26    int 80h
27
28  ; puts is not autoresolved to libc
29  puts:
30    push [esp+4]
31    push puts_format
32    call printf
33    add esp, 8
34    ret
```

Fig. 51: Stub to port the binary to a ELF format

6 Conclusion

Through many examples, we have tried to illustrate the need for tools offering always more abstraction in approaches of reverse-engineering techniques, and in particular the study of protected code.

Reverse-engineering consists of rebuilding a stack of levels of abstraction. At the base of this stack, we found the basic unit of information: the instruction. It is the modelling of its abstract behaviour the enables *Metasm* to implement an effective backtracking. The textual instruction is replaced by its semantics. This powerful concept is the foundation of our works. On the *Securitech* challenge, this has initially permit ourselves to gain the complete control flow graph, where a classical disassembler would break on the very first obfuscation pattern.

Get back at a higher level, this property also enables us to go into basic behavioural analysis: to cut off from the implementation: to concentrate on the semantics. Thus, we have been able to automatically identify the behaviour of each of the virtual machine handlers, and finally to model a virtual processor to solve *T2* challenge. The behavioural aspect that we deal with in this article is very promising, and could later be developed by taking advantage of achievements in the field of static analysis[**?**].

We have stressed on the semi-automatic nature of the proposed approachesr; indeed all rely on a part of manual analysis: awareness of a virtual machine, extracting obfuscation patterns... Even if patterns matching is easily automatable, their identification still remains a manual process. This is also a promising subject that we should consider in future developments.

From our approaches, on constant brings out: to understand a software protection, one must be placed at a level of abstraction higher than or equal to it. To conclude, we would like to say that *Metasm* is a powerfull binary manipulation framework, which is able to interact at every level of abstraction, from the lowest — the hardware — to the highest one: the source code.