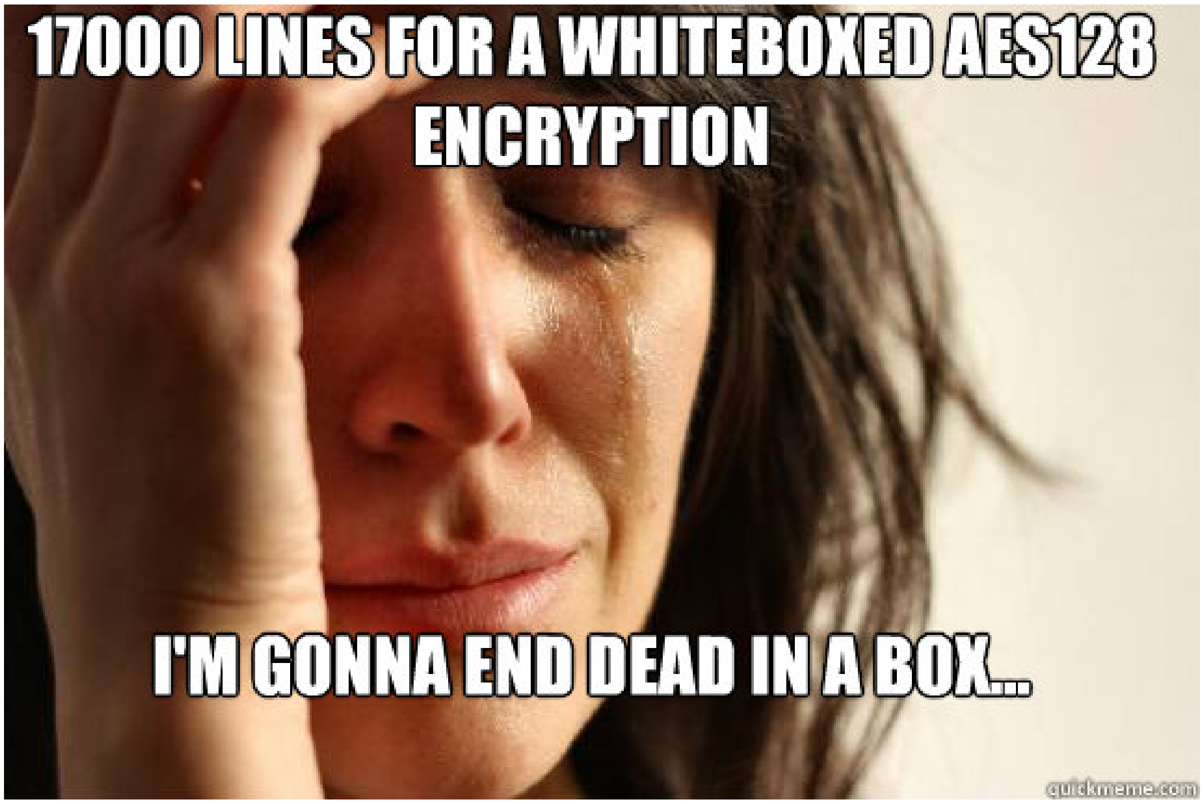
AES Whitebox Unboxing: No Such Problem

Written by:
Axel "0vercl0k" Souchet.

Twitter:
@0vercl0k


17000 LINES FOR A WHITEBOXED AES128 ENCRYPTION
I'M GONNA END DEAD IN A BOX...
quickmeme.com

May 26, 2013

1 Introduction

As every year, the organizers of the "No Such Con" event give the opportunity to the motivated people to solve some challenges just before the conference. The purpose of the challenge is really simple, you have a light GUI asking you for a login and a password and when they are submitted you either get a bad or a good boy message box. The winner of the challenge wins a trip to Hack In The Box Amsterdam 2014, quite cool: congratulation to **Florent Marceau** who solved it the first one! The challenge has been solved by approximately eight guys at the moment.

I haven't seen yet a public writeup, so here is mine: I will try to exactly explain the process I have been through from the beginning of the challenge until the very end, even when my ideas sucks. Last thing, I am really new in breaking those kind of binaries, so if you have figured a part in a different and more elegant/clever way I would really like to have a little chat with you, shoot me an email!

2 Recon

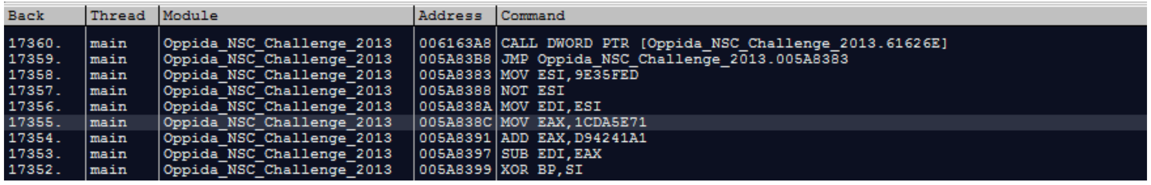
The binary is compressed with UPX, and is quite big: 2.4 mo. When you open it with IDA, you got a really clean disassembly (surprising, I thought it will be obfuscated) and you quickly understand the validation scheme works like that:

1. You enter your login and your password
2. A 16 bytes blob is returned by *MysteriousFunction* based on your password
3. Computes the MD5 of your login
4. Checks if the password is an hexadecimal string of 32 characters
5. Converts the hexadecimal string into the 16 bytes equivalent in memory
6. Finally, if the hash is different from the 16 bytes blob you lose, otherwise you win!

Our main task is to reverse the *MysteriousFunction* and to somehow invert that routine, but we will see that function is, let's say, quite tough.

3 Baby steps

Now we have understood the global validation scheme, we need to dig hard into that routine. If you open it in IDA you can see it seems obfuscated, you have to manually undefine/redefine code everywhere because IDA analysis thinks it is data and not executable code. That is a perfect timing to fire up your favorite debugger in order to step into the routine. The first thing I did was to generate a complete execution trace of that function, just to have an idea of its size and complexity: with OllyDbg you can do that really easily, you put a software breakpoint before, after the call and then you use the *Trace into* entry of the *Trace menu*. Basically it means the debugger will go through every instruction and will step into every calls. Once the software breakpoint after the call is reached, you can open the trace window to see all the instructions executed by your CPU. That awkward moment when I scroll down the whole trace and I see **17 000** assembly instructions executed, without calling any win32 API. By the way, if you are interested in generating that kind of execution trace with your own tool, consider using the dynamic binary instrumentation framework called Pin developed by Intel.



Back	Thread	Module	Address	Command
17360.	main	Oppida_NSC_Challenge_2013	006163A8	CALL DWORD PTR [Oppida_NSC_Challenge_2013.61626E]
17359.	main	Oppida_NSC_Challenge_2013	005A83B8	JMP Oppida_NSC_Challenge_2013.005A8383
17358.	main	Oppida_NSC_Challenge_2013	005A8383	MOV ESI,9E35FED
17357.	main	Oppida_NSC_Challenge_2013	005A8388	NOT ESI
17356.	main	Oppida_NSC_Challenge_2013	005A838A	MOV EDI,ESI
17355.	main	Oppida_NSC_Challenge_2013	005A838C	MOV EAX,1CDA5E71
17354.	main	Oppida_NSC_Challenge_2013	005A8391	ADD EAX,D94241A1
17353.	main	Oppida_NSC_Challenge_2013	005A8397	SUB EDI,EAX
17352.	main	Oppida_NSC_Challenge_2013	005A8399	XOR BP,SI

Figure 1: It is a lot of assembly instructions.

Then, I saved that execution trace because it may be useful for the next parts. After passing some time going through a lot those instructions, you begin to see really clear assembly obfuscation patterns to prevent the reverse-engineer from extracting the logic of that routine.

4 Memory reference tracking

The first idea I had to avoid reading the 17000 instructions was to track the memory operations: memory reads and writes. I developed my own tiny memory tracer based on Pin in order to generate another trace, but this time a shorter one. The tracer was really simple, here is how it works:

- I instrument every instruction via *INS_AddInstrumentFunction*
- In my instrumentation callback, if the address of the instruction is the beginning of the *mysterious function* I set a boolean called *trace_generation_started*, if it is the address of the end of that same routine I unset the boolean variable
- Then, I use *INS_IsMemoryRead* and *INS_IsMemoryWrite* to know if a the instruction will make a memory operation: in that case I use *INS_InsertCall* to add a call to my analysis callbacks
- When one of my analysis callbacks is called, it means a memory operation is going to occur, so I generate a trace entry with the address of the memory read/written, the size and the disassembly of the instruction
- Finally I close the memory trace file

In less than 200 lines of C++ code, you can implement a similar tool and if you are lazy you can even find an official example that does pretty much the same thing, see Memory Reference Trace (Instruction Instrumentation). The memory tracking serves only one purpose: *removing* the assembly obfuscation, let's say like a *filter*. Now, if you open your memory trace you can see something like this:

```
0x0040fc56 mov cl, byte ptr [edi]             : R 0x00566369 (1 bytes - cd)
0x0040fc60 mov ebx, dword ptr [esi]           : R 0x005a9c89 (4 bytes - 0xae3ec6cc)
0x0040fc67 mov byte ptr [ebp], bl             : W 0x00450306 (1 bytes - 65)
```

The first column is the address of the instruction, the second its disassembly, and the third the type, address, size of the memory operation. After going through both the memory trace and the complete execution trace, we can see memory patterns:

1. Read a byte somewhere (sometimes from the buffer where our password is kept in memory)
2. Read four bytes from an array where the previous byte is used as an index
3. Write the resulting byte (the LSB of the double-word previously read) somewhere

4.1 Table type 1

The vigilant reader could say in the previous dump we don't see how the fetched byte is used as an index in the next memory read: but keep in mind this trace is a **memory** trace so we don't see the instructions that don't play with memory. That's why the complete execution trace is handy, for example you can check what happens to the data stored in the 8 bits register *CL* at 0x0040fc56:

```
0040FC51  MOV EDI,Oppida_NSC_Challenge_2013.00566369
0040FC56  MOV CL,BYTE PTR [EDI]  // EDI=00566369
0040FC58  MOV ESI,ECX  // ECX=0000009A
0040FC5A  ADD ESI,005A9BBC
0040FC60  MOV EBX,DWORD PTR [ESI]  // ESI=005A9C56
0040FC62  MOV EBP,00450306
0040FC67  MOV BYTE PTR [EBP],BL  // EBX=4FCD45EB, EBP=00450306
```

We clearly see the register will be moved into *ESI* and the upper parts of that same register has been previously zeroed. The other important thing we can infer from those instructions is the size of that table: the index is coded only on one byte (*CL*), so the array can contain exactly 256 entries (one byte for each entry). To clarify we can translate that memory pattern easily in C:

```
unsigned char table[256] = {.., .., .., [...] };
unsigned char idx;
unsigned char *addr1, *addr2;
```

```
    idx = *addr1;
    *addr2 = table[idx];
```

Listing 1: Table type 1 in C

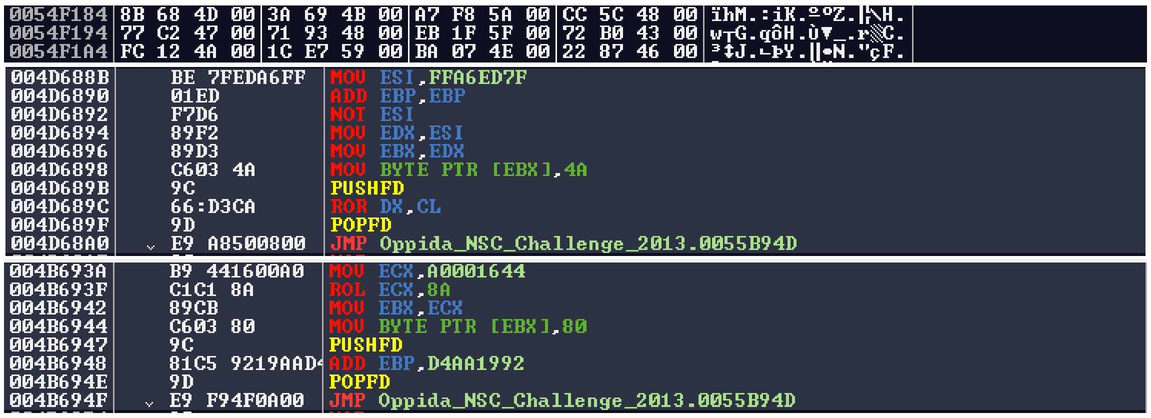
4.2 Table type 2

After spending some more hours in both the memory and execution traces, I actually spotted another different type of table. But you can also see in the memory trace things like:

```
0x005a83a3 mov ch, byte ptr [ebx]            : R 0x0061700e (1 bytes - 67)
0x005a8380 jmp dword ptr [ebx]               : R 0x0054f320 (4 bytes - 0x005a484e)
0x005a485e mov byte ptr [ebx], dh            : W 0x00591280 (1 bytes - 93)

correlation with the full execution trace:
005A839E  MOV EBX,OFFSET 0061700E
005A83A3  MOV CH,BYTE PTR [EBX]  // EDX=00000000, EBX=0061700E
005A83A5  MOV DL,CH  // ECX=0000EE00, EDX=00000000
005A83A7  DEC EBP
005A83A8  LEA ESI,[EDX*4+54F184] // EDX=000000EE, ESI=F61CA012
005A83AF  MOV EBX,ESI  // EBX=0061700E, ESI=0054F53C
005A83B1  JMP 005A8380
005A8380  JMP DWORD PTR [EBX]  // EBX=0054F53C
```

What we can see in the previous dump seems to be a typical handlers/jumps table: the table stored at 0x54F184 seems to contain function pointers. It is quite easy to verify, you can just open your debugger, dump that memory area and check if the double-words are addresses pointing to executable parts of the binary: Yes they are.



```
0054F184|8B 68 4D 00|3A 69 4B 00|A7 F8 5A 00|CC 5C 48 00|ïhM.:iK.º°Z.╠\H.
0054F194|77 C2 47 00|71 93 48 00|EB 1F 5F 00|72 B0 43 00|w┬G.qôH.Ù▼_.r▒C.
0054F1A4|FC 12 4A 00|1C E7 59 00|BA 07 4E 00|22 87 46 00|³↕J.∟þY.║•N."çF.

004D688B      BE 7FEDA6FF   MOV ESI,FFA6ED7F
004D6890      01ED          ADD EBP,EBP
004D6892      F7D6          NOT ESI
004D6894      89F2          MOV EDX,ESI
004D6896      89D3          MOV EBX,EDX
004D6898      C603 4A       MOV BYTE PTR [EBX],4A
004D689B      9C            PUSHFD
004D689C      66:D3CA       ROR DX,CL
004D689F      9D            POPFD
004D68A0    ⌄ E9 A8500800   JMP Oppida_NSC_Challenge_2013.0055B94D

004B693A      B9 441600A0   MOV ECX,A0001644
004B693F      C1C1 8A       ROL ECX,8A
004B6942      89CB          MOV EBX,ECX
004B6944      C603 80       MOV BYTE PTR [EBX],80
004B6947      9C            PUSHFD
004B6948      81C5 9219AAD4 ADD EBP,D4AA1992
004B694E      9D            POPFD
004B694F    ⌄ E9 F94F0A00   JMP Oppida_NSC_Challenge_2013.0055B94D
```

Figure 2: Beginning of the handlers table, and the code of two example handlers.

As previously, this type of table has 256 entries, but this time each entry is a double-word. But this is not finished, another interesting thing to see is that each handler referenced in that table is going to do two main operations:

1. Write a constant value somewhere (sometime the constant cannot be seen easily because of the assembly obfuscation though). But that somewhere is the same address for each handler.

2. Jump back somewhere, and that somewhere is the same address for all the handlers referenced the table. In the previous picture for example, the two handlers are jumping to 0x55B94D.

Finally we can write that behavior in C as we did previously:

```c
void handler1()
{
  *addr2 = 0x4A;
}

void handler2()
{
  *addr2 = 0x80;
}

unsigned int table[256] = {handler1, handler2, [...]};
unsigned char *addr1;

table[*addr1]();
// ...
```

Listing 2: Table type 2

This is a clever way to bother the reverse-engineer, because you can simplify this process by a simple table of type 1. In the previous example, the first entry of this table should be 0x4A, the second 0x80 and so on.

4.3 Table type 3

The last type of table I have encountered in this challenge is big tables that takes a 16 bits index. In fact, it's just a 256 table where each entry is also a 256 table of one bytes. It makes total sense because with a 16 bits index you can store 65536 bytes, but you can also store 256*256 bytes.

4.4 All your tables are belong to us

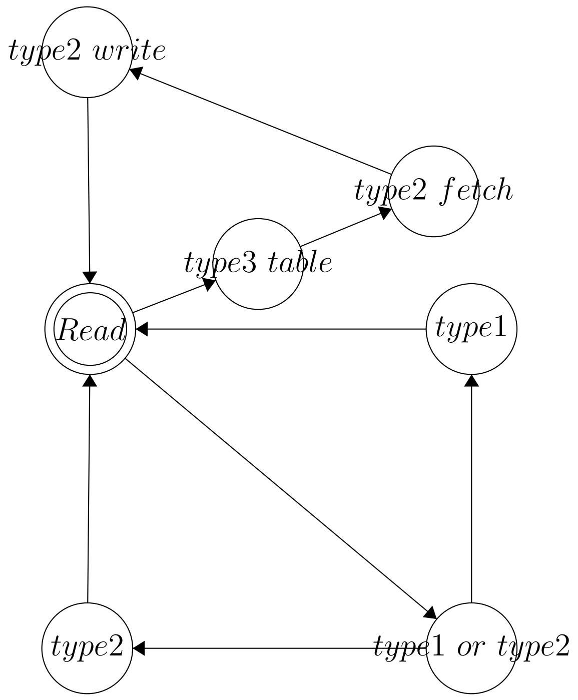
Once we have spotted all those different tables, we don't find any different memory access pattern from the trace. Basically it means the algorithm is made only of accessing table, storing bytes that will serve as index and so on. Nevertheless, the assembly obfuscation is really annoying so I came up with another idea: extract the logic of the algorithm without dealing with the obfuscation.

5 Logic extraction

My idea was really simple: I have been in this code so much time hitting F7 that I could extract only three different operations based on the previous table. Here there are:

1. When you have a table type 1: You read a byte at an address, you use that byte as an index, you read another byte and you write that byte somewhere.
2. When you have a table type 2: You read a byte at an address, you use that byte in a double-word table, then you redirect the execution flow somewhere, you write a byte somewhere and you jump somewhere.
3. When you have a table 3: You read a byte at an address, you read another byte at an address, you make a 16 bits number with those two bytes, you read a byte and you write it somewhere.

It was clearer, and clearer in my head: maybe I could use a *Finite State Machine* to match those different patterns in my complete execution trace: I will just have to read the register values to not deal with the assembly obfuscation (remember that the CPU context is saved before each instruction in the execution trace we made with OllyDbg at the beginning). At first glance, it was a bit crazy to only parse a text file, match some patterns and to convert them directly in C but if it worked I would be able to focus only on breaking the algorithm without having to deal with boring obfuscations. Here is the really simple *FSM* I used to bypass assembly obfuscations:



Note that it is easier to understand it with the implementation script and the execution trace, so check the codes on my github account! After launching my magic script, I get a C file with one 1024 lines (yay, it's aligned) where it does only the three things we described previously. The easiest way to verify our algorithm is good is just to create a DLL project and to inject it in the challenge (this way we don't need to dump the majority of the tables. Even if in fact, if you want to do that you will need to dump the tables of type 2 ; but it's really a detail: email me if you want to talk about how I managed to dump them all). Then you just have to verify your algorithm gives the same output than the crackme, and it was the case, amazing :).

So far, we have done almost the first half of the challenge: the cool thing is we can now entirely focus on the cryptographic algorithm.

6 Breaking the algorithm

Even if we have broken the assembly obfuscations, @elvanderb was evil enough to obfuscate the logic itself of the algorithm: long story short, it is a *real mess*. By the way, I found that type of obfuscation really harder than the previous one. My new idea was now, to simplify the algorithm in order to have more chances to understand how it works.

6.1 Static single assignment form

When you are looking at the generated code, you can see things like that:

```
*memory2 = T16_004EF23E[(*memory1 << 8) + *memory2];
```

Listing 3: Not really cool.

In this expression the byte stored at *memory2* is used as both the LSB of the index and where the read byte will be stored. Really confusing when you are going through the algorithm, moreover the addresses used in the algorithm are reused a bit everywhere so it is not really convenient to track the different value of a variable. The solution to this problem is simple, we will convert our algorithm into the SSA form. The idea of that form when you have a variable, you will see **only** a single assignment to that variable. It is really easily done with yet another python script, and it will transform the previous example like this:

```
*memory3 = T16_004EF23E[(*memory2 << 8) + *memory1];
```

Listing 4: SSA version.

6.2 Algorithm visualization

Now we need to really understand how the cryptography works. Usually the idea when you are not seeing something is to try to see it but in another way. Let's draw graph the flow of the different variables from the input password until the final sixteen bytes blob generated. The idea behind this tracking is simple, see the example:

```
If I have those C lines:
*a = T[*b];
*c = T2[*a];
output[0x1] = *c;

I will have a graph like that:
a -> T -> T2 -> output[0x1]
```

This way we won't show the variables used as a temporary location. Here is the final graph of the whole algorithm:

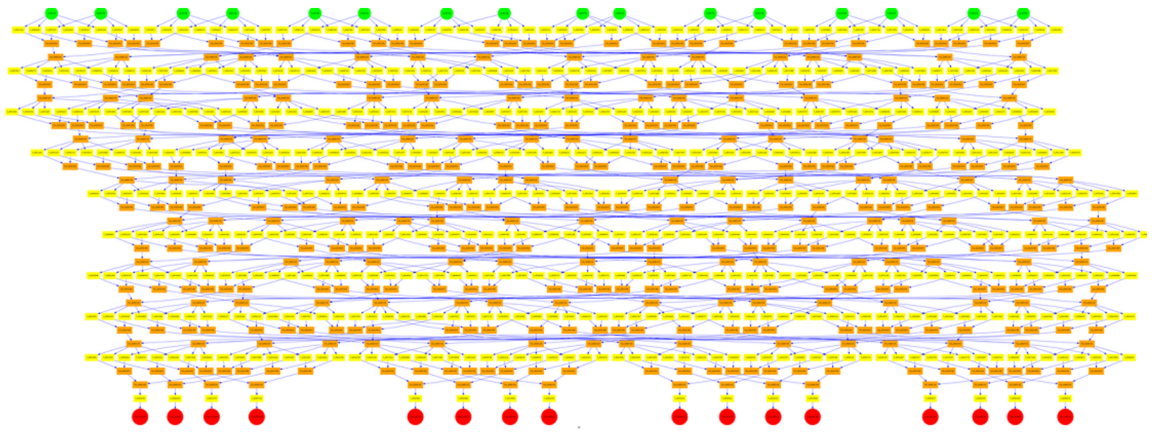


Figure 3: w0000t, seems cool.

The green nodes are the input bytes of the function (our password) and the red nodes are the output bytes (the blob). The yellow boxes are tables of type 1 and type 2, the orange boxes are the tables type 3. This picture was really a big help for me, because I could finally see important things:

- A kind of unrolled loop that is executed nine times: it is a round
- At the end of each round we can see always sixteen bytes: it is like a state, and the state of the last round is the output blob
- Each round seems to have three steps:
  1. It takes the current sixteen bytes state, use the yellow boxes to extend that state to sixty-four bytes
  2. Then it takes those sixty-four bytes and use the first "orange line" to reduce them to thirty-two bytes
  3. Finally, it takes those thirty-two bytes and reduce them to the size of what we called the state
- Only the last round has a special behavior

From those observations we can infer that the original algorithm looks like that:

```
void algo(unsigned char key[16], unsigned char out
   [16])
{
    unsigned char state[16], sixtyfour[64], thirtytwo
       [32];
    memcpy(state, key, 16);
    for(unsigned int i = 0; i < 9; ++i)
    {
        first_step(state, sixtyfour); // ShiftRows ?
```

```
            second_step(sixtyfour, thirtytwo); //
               TBoxesTyiTables ?
            third_step(thirtytwo, state); // XORTables ?
        }
        special(state);
        //the final blob is in state
    }
```

Listing 5: Weird whiteboxed encryption algorithm

While I was reverse-engineering this challenge, I was pretty sure it was a whiteboxed implementation of a known encryption algorithm because of all that tables lookup. At this time, I have read a lot of documentations and more precisely this one: A Tutorial on White-box AES. It seemed really familiar, but it wasn't exactly the same implementation anyway. So I was pretty convinced it was an implementation of AES (128?) but I didn't really know how to prove it. I'm really a newbie in cryptographic stuff, so I just kept in a corner of my mind that it could be an AES 128 encryption whiteboxed. The funny thing is the fact that you **really** don't need to know the algorithm in order to break it. Also we can see that the algorithm is not completely reversible without brute-force attack, because of the tables of type 3.

But now we saw a main loop, let's focus only on a round and try to break a round: if we manage to do that we will just have to break each round starting from the last, until the first.

6.3 Round analysis

Focusing on a round is a good idea because you don't have to think to a lot of things, you know that the other rounds will somehow have the same behavior. Here is the last round of the challenge:

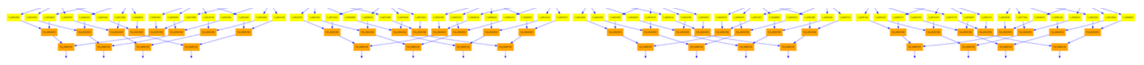


Figure 4: Last round.

Like earlier, from that graph we can extract important detail:

- The round seems to be split into four parts, the input of the round is the previous state, and the ouput is the new state
- Each quarter of the round generate four bytes of the state
- And more importantly, each quarter of the round is **independent**. For example on the first one, we clearly see its four output bytes come only from its four input bytes.

Let's reduce the perimeter and focus only on a quarter of round now.

6.4 A quarter of round

One more time here is what looks like the first quarter of the last round:

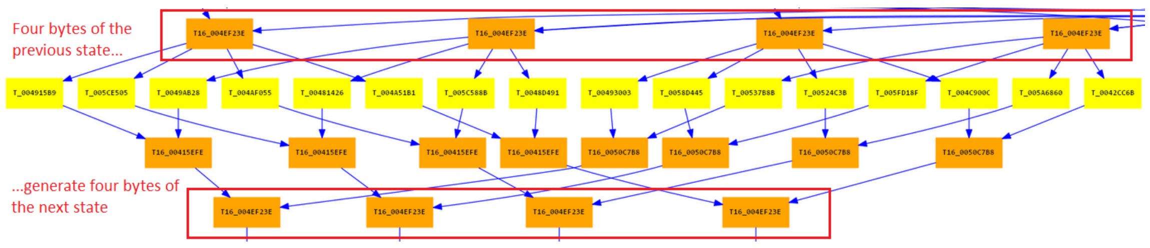


Figure 5: First quarter, last round.

Basically it seems we can brute-force the four inputs bytes in order to have whatever we want in the output bytes. There is one last problem, if you already looked at the C algorithm, the rounds and their quarters are not in ”order”. It is a part of @elvanderb’s evil logic obfuscation plan. Before going further, I really wanted at that moment to see if that what really feasible, I mean to break a quarter of round. To convince myself, I have extracted the last round, and its first quarter from the C algorithm and I tried to brute-force it:

```
void bruteforce_quarter_test()
{
    unsigned char memory[876] = {0}, out[148] = {0};
    unsigned char memory769_wanted, memory744_wanted, memory837_wanted, memory773_wanted;
    unsigned char out145_wanted, out143_wanted, out142_wanted, out141_wanted;

    out145_wanted = 0x00;
    out143_wanted = 0x11;
    out142_wanted = 0x22;
    out141_wanted = 0x33;

    for(unsigned int i = 0; i < 0x100; ++i)
    {
        memory[769] = i;
        for(unsigned int j = 0; j < 0x100; ++j)
        {
            memory[744] = j;
            for(unsigned int k = 0; k < 0x100; ++k)
            {
                memory[837] = k;
                for(unsigned int l = 0; l < 0x100; ++l)
                {
                    memory[773] = l;

                    memory[775] = T_0058D445[memory[769]];
                    memory[782] = T_004C900C[memory[769]];
                    memory[789] = T_00493003[memory[769]];
                    memory[802] = T_00524C3B[memory[769]];
                    //
                    memory[746] = T_0042CC6B[memory[744]];
                    memory[747] = T_005A6860[memory[744]];
                    memory[748] = T_005FD18F[memory[744]];
                    memory[755] = TH_00537B8B[memory[744]];
                    //
                    memory[839] = T_005C588B[memory[837]];
                    memory[842] = T_0049AB28[memory[837]];
                    memory[846] = T_00481426[memory[837]];
                    memory[862] = T_0048D491[memory[837]];
                    //
                    memory[776] = T_004915B9[memory[773]];
                    out[119] = T_005CE505[memory[773]];
                    memory[783] = TH_004AF055[memory[773]];
                    memory[803] = T_004A51B1[memory[773]];
                    //
                    memory[812] = T16_0050C7B8[memory[775]][memory[748]];
                    memory[788] = T16_0050C7B8[memory[782]][memory[746]];
                    memory[835] = T16_0050C7B8[memory[789]][memory[755]];
                    memory[807] = T16_0050C7B8[memory[802]][memory[747]];
                    //
                    memory[840] = T16_00415EFE[memory[839]][memory[783]];
                    memory[863] = T16_00415EFE[memory[842]][memory[776]];
```

```
                                memory[853] = T16_00415EFE[memory[846]][out[119]];
                                memory[868] = T16_00415EFE[memory[862]][memory[803]];
                                //
                                memory[874] = T16_004EF23E[memory[812]][memory[853]];
                                memory[872] = T16_004EF23E[memory[788]][memory[868]];
                                memory[869] = T16_004EF23E[memory[835]][memory[863]];
                                memory[845] = T16_004EF23E[memory[807]][memory[840]];
                                //
                                out[145] = TH_004866A5[memory[874]];
                                out[143] = T_004977CB[memory[872]];
                                out[142] = T_0049635D[memory[869]];
                                out[141] = T_005C2F74[memory[845]];

                                if(out[145] == out145_wanted && out[143] == out143_wanted && out[142] ==
                                    out142_wanted && out[141] == out141_wanted)
                                {
                                    memory769_wanted = memory[769];
                                    memory744_wanted = memory[744];
                                    memory837_wanted = memory[837];
                                    memory773_wanted = memory[773];
                                    printf("done\n");
                                    return 0
                                }
                            }
                        }
                    }
                }
            }
```

Listing 6: Quarter round brute-force attack

If you execute that code, it will compute the input values you need to get the output values you want in 40 seconds. Feasible then. We have to build a brute-force like the previous one four times by round, it means we will have nine times four brute-forces. If one is taking fourty seconds, it will take some dozen of minutes to get a key, fair enough, let's do this.

6.5 Writing the brute-force attack

The idea of the brute-force is simple, let's see a round like a function with a sixteen bytes input parameter (the previous state) and a sixteen bytes output (the next state). Here is a little picture of how the brute-force works, I hope it will be clearer:

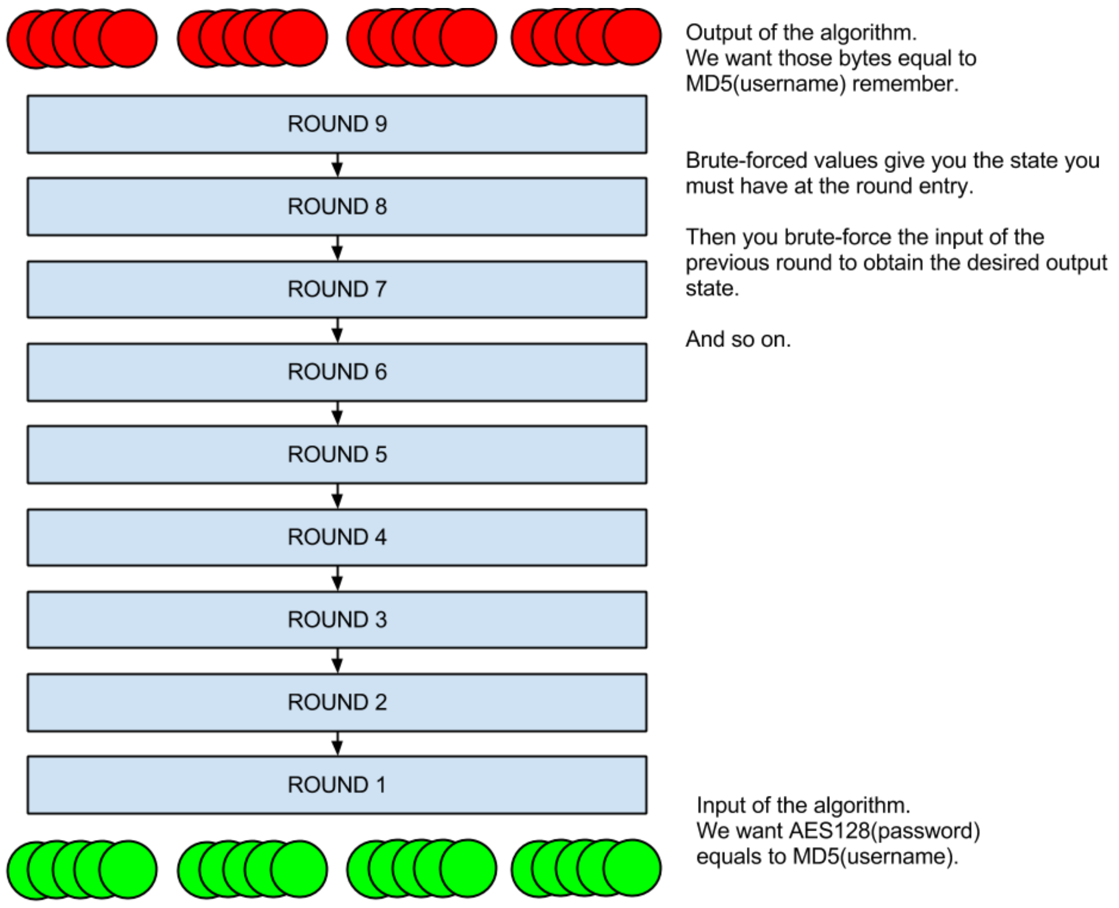
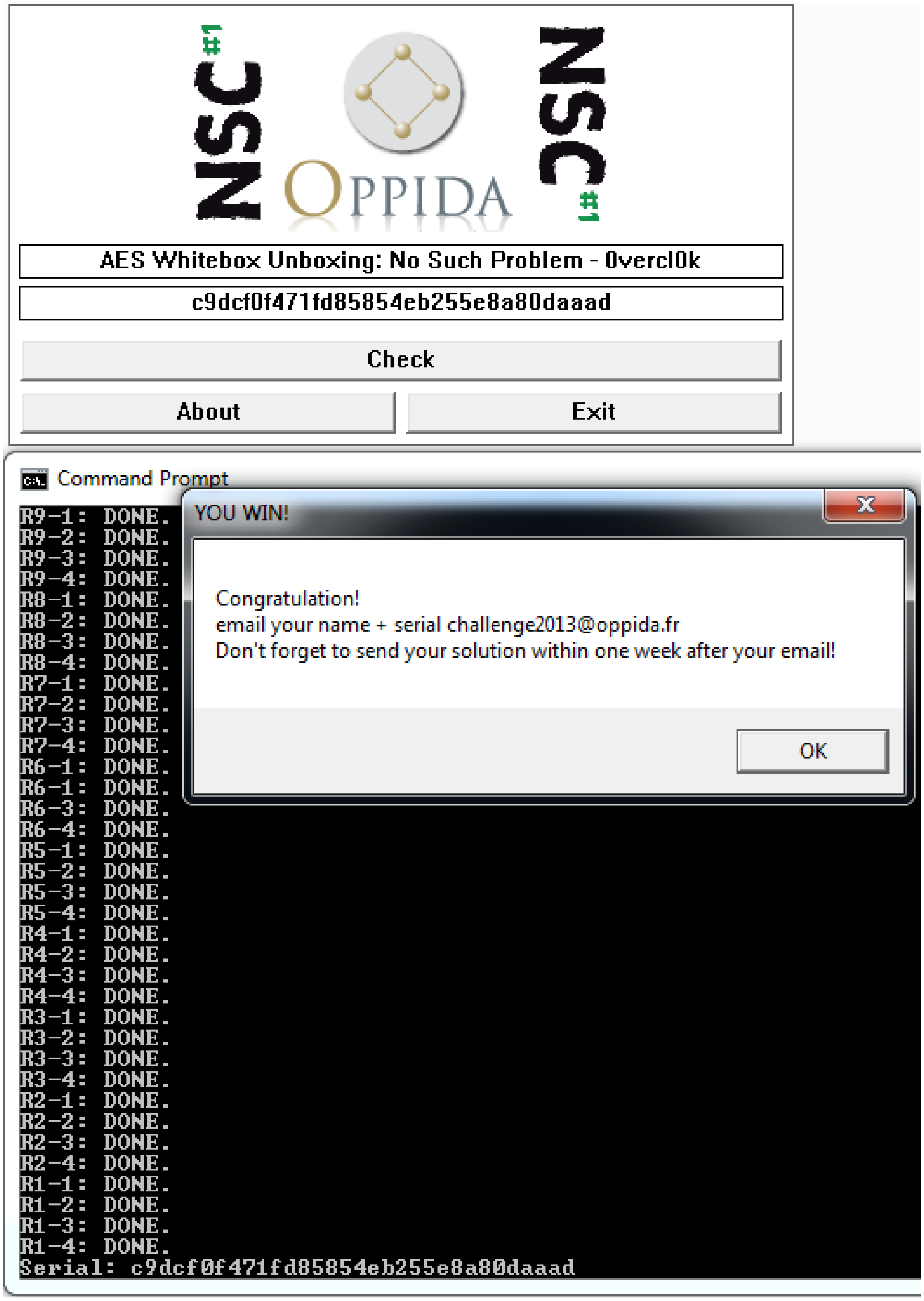

Output of the algorithm.
We want those bytes equal to MD5(username) remember.
ROUND 9
ROUND 8
ROUND 7
ROUND 6
ROUND 5
ROUND 4
ROUND 3
ROUND 2
ROUND 1
Brute-forced values give you the state you must have at the round entry.
Then you brute-force the input of the previous round to obtain the desired output state.
And so on.
Input of the algorithm.
We want AES128(password) equals to MD5(username).

Figure 6: We want a state, we brute-force the input values to obtain that state, and so on.

In order to build that brute-forcer I have coded a simple Python script that extracted from the algorithm a quarter of a round. I did it thirty-six times and I got a 2000 lines of C file to brute-force a key!



AES Whitebox Unboxing: No Such Problem - 0vercl0k

c9dcf0f471fd85854eb255e8a80daaad

Congratulation!
email your name + serial challenge2013@oppida.fr
Don't forget to send your solution within one week after your email!

```
R9-1: DONE.
R9-2: DONE.
R9-3: DONE.
R9-4: DONE.
R8-1: DONE.
R8-2: DONE.
R8-3: DONE.
R8-4: DONE.
R7-1: DONE.
R7-2: DONE.
R7-3: DONE.
R7-4: DONE.
R6-1: DONE.
R6-1: DONE.
R6-3: DONE.
R6-4: DONE.
R5-1: DONE.
R5-2: DONE.
R5-3: DONE.
R5-4: DONE.
R4-1: DONE.
R4-2: DONE.
R4-3: DONE.
R4-4: DONE.
R3-1: DONE.
R3-2: DONE.
R3-3: DONE.
R3-4: DONE.
R2-1: DONE.
R2-2: DONE.
R2-3: DONE.
R2-4: DONE.
R1-1: DONE.
R1-2: DONE.
R1-3: DONE.
R1-4: DONE.
Serial: c9dcf0f471fd85854eb255e8a80daaad
```

Figure 7: O.M.G.

7 End of story

This is the end folks, I hope you really enjoyed diving into that original challenge. Hats off for the crazy @elvanderb who developed the challenge. One important thing I won't forget for the next challenges I will do is: don't be lazy, automate your work and don't quit. It exists a lot of cool library/framework/plugins to write easily enough piece of code to really speed-up your progression on understanding something: play with Pin, ODBGScript, IDAPython, Detours, DLL Injection, pygraphviz, pydbg, etc. Also, when something is hard to see, don't hesitate to use another view of your problem: like the graph we did earlier. I sincerely think without this graph I would had quit the challenge, because the logic obfuscation was really causing pain to my brain.

If you are interested in the whitebox cryptography subject, I have read cool articles (cool articles for newbies, I am sure the crypto guys already know that stuff anyway):

- Practical cracking of white-box implementations
- A Tutorial on White-box AES
- WBC: protecting cryptographic keys in software applications

For those who want to continue the adventure, there are still some cool undone tasks to work on:

- Factorize the implementation: loop un-unrolling
- Key extraction

If you manage to do either one, feel free to contact me, I would love to see how you did it! By the way, almost all of my scripts have been uploaded here: NoSuchCon2013! I haven't cleaned them, I thought it would give some authenticity to my solution.

Special thanks for the reviewers and the guys who helped me: @__x86 and @kutioo.