Diary of a reverse-engineer

Because we like to play with weird things.

- RSS

Search

Navigate...

- Blog
- Archives
- About
- Presentations

Taming a Wild Nanomite-protected MIPS Binary With Symbolic Execution: No Such Crackme

Oct 11th, 2014

As last year, the French conference No Such Con returns for its second edition in Paris from the 19th of November until the 21th of November. And again, the brilliant Eloi Vanderbeken & his mates at Synacktiv put together a series of three security challenges especially for this occasion. Apparently, the three tasks have already been solved by awesome @0xfab which won the competition, hats off :).

To be honest I couldn't resist to try at least the first step, as I know that Eloi always builds really twisted and nice binaries ; so I figured I should just give it a go!

But this time we are trying something different though: this post has been co-authored by both *Emilien Girault* (@emiliengirault) and I. As we have slightly different solutions, we figured it would be a good idea to write those up inside a single post. This article starts with an introduction to the challenge and will then fork, presenting my solution and his.

As the article is quite long, here is the complete table of contents:

REcon: Here be dragons

This part is just here to get things started: how to have a debugging environment, to know a bit more about MIPS and to know a bit more what the binary is actually doing.

MIPS 101

The first interesting detail about this challenge is that it is a MIPS binary ; it's really kind of exotic for me. I'm mainly looking at Intel assembly, so having the opportunity to look at an unknown architecture is always appealing. You know it's like discovering a new little toy, so I just couldn't help myself & started to read the MIPS basics.

This part is going to describe only the essential information you need to both understand and crack wide open the binary ; and as I said I am not a MIPS expert, at all. From what I have seen though, this is fairly similar to what you can see on an Intel x86 CPU:

- It is little endian (note that it also exists a big-endian version but it won't be covered in this post),
- It has way more general purpose registers,
- The calling convention is similar to __fastcall: you pass arguments via registers, and get the return of the function in *$v0*,
- Unlike x86, MIPS is RISC, so much simpler to take in hand (trust me on that one),
- Of course, there is an IDA processor,

- Linux and the regular tools also exists for MIPS so we will be able to use the “normal” tools we are used to use,
- It also uses a stack, much less than x86 though as most of the things happening are in registers (in the challenge at least).

Setting up a proper debugging environment

The answer to that question is Qemu, as expected. You can even download already fully prepared & working Debian images on aurel32’s website.

> get a working qemu environment

```
overclok@wildout:~/chall/nsc2014$ wget https://people.debian.org/~aurel32/qe
overclok@wildout:~/chall/nsc2014$ wget https://people.debian.org/~aurel32/qe
overclok@wildout:~/chall/nsc2014$ cat start_vm.sh
qemu-system-mipsel -M malta -kernel vmlinux-3.2.0-4-4kc-malta -hda debian_wh
overclok@wildout:~/chall/nsc2014$ ./start_vm.sh
[    0.000000] Initializing cgroup subsys cpuset
[    0.000000] Initializing cgroup subsys cpu
[    0.000000] Linux version 3.2.0-4-4kc-malta (debian-kernel@lists.debian.o
[...]
debian-mipsel login: root
Password:
Last login: Sat Oct 11 00:04:51 UTC 2014 on ttyS0
Linux debian-mipsel 3.2.0-4-4kc-malta #1 Debian 3.2.51-1 mips

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
root@debian-mipsel:~# uname -a
Linux debian-mipsel 3.2.0-4-4kc-malta #1 Debian 3.2.51-1 mips GNU/Linux
```

Feel free to install your essentials in the virtual environment, some tools might come handy (it should take a bit of time to install them though):

> aptitude install all-of-the-things

```
root@debian-mipsel:~# aptitude install strace gdb gcc python
root@debian-mipsel:~# wget https://raw.githubusercontent.com/zcutlip/gdbinit
root@debian-mipsel:~# mv gdbinit-mips ~/.gdbinit
root@debian-mipsel:~# gdb -q /home/user/crackmips
Reading symbols from /home/user/crackmips...(no debugging symbols found)...d
(gdb) b *main
Breakpoint 1 at 0x402024
(gdb) r 'doar-e ftw'
Starting program: /home/user/crackmips 'doar-e ftw'
-------------------------------------------------------------------
[registers]
  V0: 7FFF6D30  V1: 77FEE000  A0: 00000002  A1: 7FFF6DF4
  A2: 7FFF6E00  A3: 0000006C  T0: 77F611E4  T1: 0FFFFFFE
  T2: 0000000A  T3: 77FF6ED0  T4: 77FE5590  T5: FFFFFFFF
  T6: F0000000  T7: 7FFF6BE8  S0: 00000000  S1: 00000000
  S2: 00000000  S3: 00000000  S4: 004FD268  S5: 004FD148
  S6: 004D0000  S7: 00000063  T8: 77FD7A5C  T9: 00402024
  GP: 77F67970  S8: 0000006C  HI: 000001A5  LO: 00005E17
  SP: 7FFF6D18  PC: 00402024  RA: 77DF2208
-------------------------------------------------------------------
[code]
```

```
22 => 0x402024 <main>:     addiu   sp,sp,-72
23    0x402028 <main+4>:   sw      ra,68(sp)
24    0x40202c <main+8>:   sw      s8,64(sp)
25    0x402030 <main+12>:  move    s8,sp
26    0x402034 <main+16>:  sw      a0,72(s8)
27    0x402038 <main+20>:  sw      a1,76(s8)
28    0x40203c <main+24>:  lw      v1,72(s8)
29    0x402040 <main+28>:  li      v0,2
```

And finally you should be able to run the wild beast:

release the beast

```
1 root@debian-mipsel:~# /home/user/crackmips
2 usage: /home/user/crackmips password
3 root@debian-mipsel:~# /home/user/crackmips 'doar-e ftw'
4 WRONG PASSWORD
```

Brilliant :–).

The big picture

Now that we have a way of both launching and debugging the challenge, we can open the binary in IDA and start to understand what type of protection scheme is used. As always at that point, we are really not interested in details: we just want to understand how it works and what parts we will have to target to get the *good boy* message.

After a bit of time in IDA, here is how works the binary:

1. It checks that the user supplied one argument: the serial
2. It checks that the supplied serial is 48 characters long
3. It converts the string into 6 *DWORD*s (/!\ pitfall warning: the conversion is a bit strange, be sure to verify your algorithm)
4. The beast forks in two:
    1. [Father] It seems, somehow, this one is *driving* the son, more on that later
    2. [Son] After executing a big chunk of code that modifies (in place) the 6 original *DWORD*s, they get compared against the following string *[Synacktiv + NSC = <3]*
    3. [Son] If the comparison succeeds you win, else you loose

Basically, we need to find the 6 input *DWORD*s that are going to generate the following ones in *output*: *0x7953205b*, *0x6b63616e*, *0x20766974*, *0x534e202b*, *0x203d2043*, *0x5d20333c*. We also know that the father is going to interact with its son, so we need to study both codes to be sure to understand the challenge properly. If you prefer code, here is the big picture in C:

big picture

```
1  int main(int argc, char *argv[])
2  {
3      DWORD serial_dwords[6] = {0};
4      if(argc != 2)
5          Usage();
6
7      // Conversion
8      a2i(argv[1], serial_dwords);
9
10     pid_t pid = fork();
```

```
11     if(pid != 0)
12     {
13         // Father
14         // a lot of stuff going on here, we will see that later on
15     }
16     else
17     {
18         // Son
19         // a lot of stuff going on here, we will see that later on
20
21         char *clear = (char*)serial_dwords;
22         bool win = memcmp(clear, "[ Synacktiv + NSC = <3 ]", 48);
23         if(win)
24             GoodBoy();
25         else
26             BadBoy();
27     }
28 }
```

Let’s get our hands dirty

Father’s in charge

The first thing I did after having the big picture was to look at the code of the father. Why? The code seemed a bit simpler than the son’s one, so I figured studying the father would make more sense to understand what kind of protection we need to subvert. You can even crank up strace to have a clearer overview of the syscalls used:

> strace father

```
1  root@debian-mipsel:~# strace -i /home/user/crackmips $(python -c 'print "1"*
2  [7734e224] execve("/home/user/crackmips", ["/home/user/crackmips", "11111111
3  [...]
4  [77335e70] clone(child_stack=0, flags=CLONE_CHILD_CLEARTID|CLONE_CHILD_SETTI
5  [77335e70] --- SIGCHLD (Child exited) @ 0 (0) ---
6  [7733557c] waitpid(2539, [{WIFSTOPPED(s) && WSTOPSIG(s) == SIGTRAP}], __WALL
7  [7737052c] ptrace(PTRACE_GETREGS, 2539, 0, 0x7f8f87c4) = 0
8  [7737052c] ptrace(PTRACE_SETREGS, 2539, 0, 0x7f8f87c4) = 0
9  [7737052c] ptrace(PTRACE_CONT, 2539, 0, SIG_0) = 0
10 [7737052c] --- SIGCHLD (Child exited) @ 0 (0) ---
11 [7733557c] waitpid(2539, [{WIFSTOPPED(s) && WSTOPSIG(s) == SIGTRAP}], __WALL
12 [7737052c] ptrace(PTRACE_GETREGS, 2539, 0, 0x7f8f87c4) = 0
13 [7737052c] ptrace(PTRACE_SETREGS, 2539, 0, 0x7f8f87c4) = 0
14 [7737052c] ptrace(PTRACE_CONT, 2539, 0, SIG_0) = 0
15 [7737052c] --- SIGCHLD (Child exited) @ 0 (0) ---
16 [7733557c] waitpid(2539, [{WIFSTOPPED(s) && WSTOPSIG(s) == SIGTRAP}], __WALL
17 [7737052c] ptrace(PTRACE_GETREGS, 2539, 0, 0x7f8f87c4) = 0
18 [7737052c] ptrace(PTRACE_SETREGS, 2539, 0, 0x7f8f87c4) = 0
19 [7737052c] ptrace(PTRACE_CONT, 2539, 0, SIG_0) = 0
20 [7733557c] waitpid(2539, [{WIFSTOPPED(s) && WSTOPSIG(s) == SIGTRAP}], __WALL
21 [7733557c] --- SIGCHLD (Child exited) @ 0 (0) ---
22 [7737052c] ptrace(PTRACE_GETREGS, 2539, 0, 0x7f8f87c4) = 0
23 [7737052c] ptrace(PTRACE_SETREGS, 2539, 0, 0x7f8f87c4) = 0
24 [7737052c] ptrace(PTRACE_CONT, 2539, 0, SIG_0) = 0
25 [7737052c] --- SIGCHLD (Child exited) @ 0 (0) ---
26 [7733557c] waitpid(2539, [{WIFSTOPPED(s) && WSTOPSIG(s) == SIGTRAP}], __WALL
27 [7737052c] ptrace(PTRACE_GETREGS, 2539, 0, 0x7f8f87c4) = 0
28 [7737052c] ptrace(PTRACE_SETREGS, 2539, 0, 0x7f8f87c4) = 0
29 [7737052c] ptrace(PTRACE_CONT, 2539, 0, SIG_0) = 0
30 [7737052c] --- SIGCHLD (Child exited) @ 0 (0) ---
31 [7733557c] waitpid(2539, [{WIFSTOPPED(s) && WSTOPSIG(s) == SIGTRAP}], __WALL
```

```
[7737052c] ptrace(PTRACE_GETREGS, 2539, 0, 0x7f8f87c4) = 0
[7737052c] ptrace(PTRACE_SETREGS, 2539, 0, 0x7f8f87c4) = 0
[7737052c] ptrace(PTRACE_CONT, 2539, 0, SIG_0) = 0
[7737052c] --- SIGCHLD (Child exited) @ 0 (0) ---
[7733557c] waitpid(2539, [{WIFSTOPPED(s) && WSTOPSIG(s) == SIGTRAP}], __WALL
[7737052c] ptrace(PTRACE_GETREGS, 2539, 0, 0x7f8f87c4) = 0
[7737052c] ptrace(PTRACE_SETREGS, 2539, 0, 0x7f8f87c4) = 0
[7737052c] ptrace(PTRACE_CONT, 2539, 0, SIG_0) = 0
[...]
```

That's an interesting output that I didn't expect at all actually. What we are seeing here is the father driving its son by modifying, potentially (we will find out that later), its context every time the son is *SIGTRAP*ing (note *waitpid* second argument).

From here, if you are quite familiar with the different existing type of software protections (I'm not saying I am an expert in this field but I just happened to know that one :-P) you can pretty much guess what that is: nanomites this is!

Nanomites 101

Namomites are quite a nice protection. Though, it is quite a generic name ; you can really use that protection scheme in whatever way you like: your imagination is the only limit here. To be honest, this was the first time I saw this kind of protection implemented on a Unix system ; really good surprise! It usually works this way:

1. You have two processes: a driver and a driven ; a father and a son
2. The driver is attaching itself to the driven one with the debug APIs available on the targeted platform (*ptrace* here, and *CreateProcess*/*DebugActiveProcess* on Windows)
    1. Note that, by design you won't be able to attach yourself to the son as both Windows and Linux prevent that (by design): some people call that part the *DebugBlocker*
    2. You will able to debug the driver though
3. Usually the interesting code is in the son, but again you can do whatever you want. Basically, you have two rules if you want an efficient protection:
    1. Make sure the driven process can't run without its driver and that they are really tied to each other
    2. The strength of the protection is that strong/intimate bound between the two processes
    3. Design your algorithm such that *removing* the driver is really difficult/painful/driving mad the attacker
4. The driven process can *call*/*notify* the driver by just *SIGTRAP*ing with an *int3*/*break* instruction for example

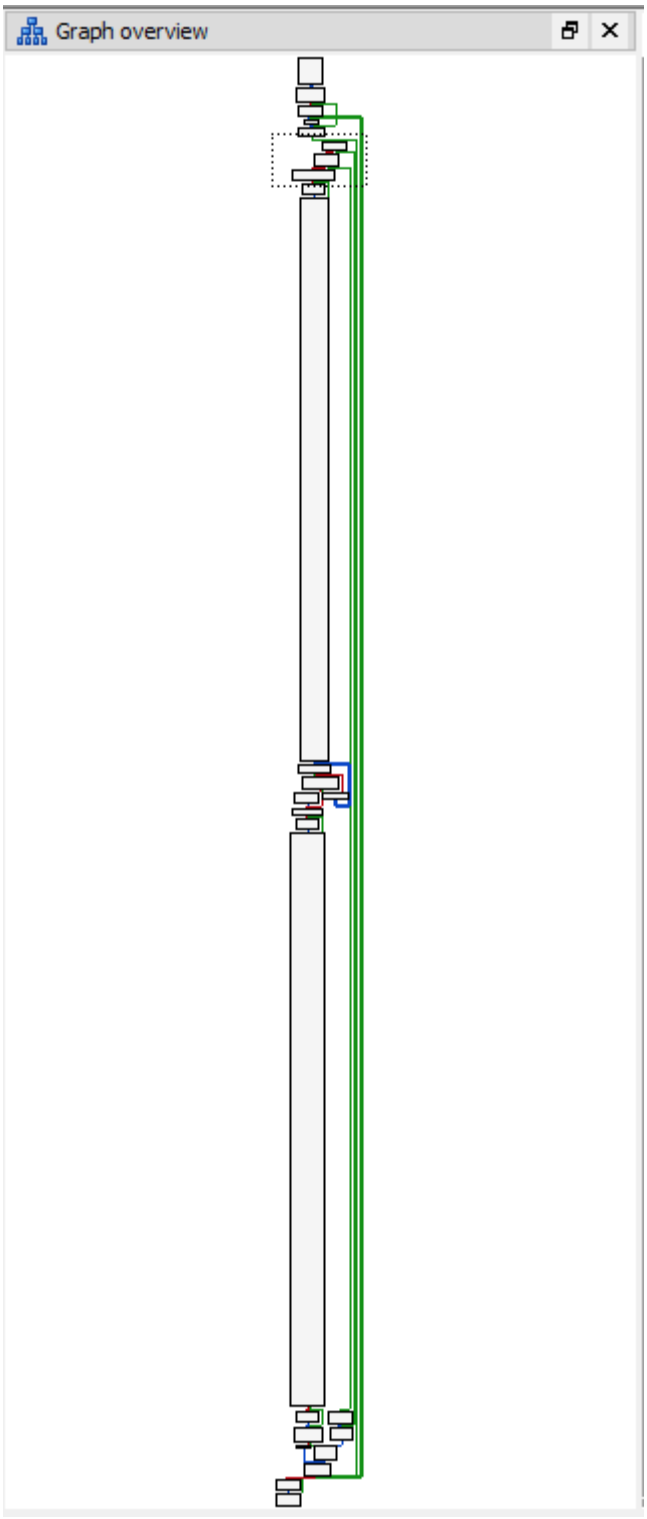
As I said, I see this protection scheme more like a *recipe*: you are free to customize it at your convenience really. If you want to read more on the subject, here is a list of links you should check out:

- Nanomite and Debug Blocker for Linux Applications: It gives a good overview of how you can get such a protection scheme to work on Linux,
- Unpackme I am Famous: This shows you what nanomites look like on Windows in a real protected product ; done by my mate @w4kfu,
- Debug me: Another sweet challenge that uses nanomites on Windows

How the father works

Now it is time to took into details the father ; here is how it works:

- The first thing it does is to *waitpid* until its son triggers a *SIGTRAP*
- The driver retrieves the CPU context of the son process and more precisely its *program counter*: *$pc*
- Then we have a huge block of arithmetic computations. But after spending a bit of time to study it, we can see that huge block as a black-box function that takes two parameters: the program counter of the son and some kind of counter value (as this code is going to be executed in a loop, for each *SIGTRAP* this variable is going to be incremented). It generates a single output which is a 32 bits value that I call the *first magic value*. Let's not focus on what the block is actually doing though, we will develop some tool in the next part to deal with that :–) so let's keep moving!



- This *magic value* is then used to find a specific entry in an array of *QWORD*s (606 *QWORD*s which is 6 times the number of *break* instructions in the son — you will understand that a bit later don't worry). Basically, the code is going to loop over every single *QWORD* of this array until finding one that has the high *DWORD* equals to the *magic value*. From there you get another *magic value* which is the lowest *DWORD* of the matching *QWORD*.
- Another huge block of arithmetic computations is used. Similarly to the first one, we can see it as a black-box function with two inputs: the second *magic value* and a round index (the son is executing its code 6 times, so this round index will start from 0 until 5 — again this will be a bit clearer when we look at the son, so just keep this detail in your mind). The output of this function is a 32 bits value. Again, do not study this block, we don't need it.
- The generated value is in fact a valid code address inside the son ; so straight after the

> computation, the father is going to modify the program counter in the previously retrieved CPU context. Once this is done, it calls *ptrace* with *SETREGS* to set the new CPU context of the son.

This is what roughly is going to be executed every time the son is going to hit a *break* instruction ; the father is definitely driving the son. And we can feel it now, the son is going to jump (via its father) through block of codes that aren't (necessary) contiguous in memory, so studying the son code as it is in IDA is quite pointless as those basic blocks aren't going to be executed in this order.

Long story short, the nanomites are used as some kind of runtime code flow scrambling primitive, isn't it exciting? Told you that @elvanderb is crazy :–).

Gearing up: Writing a symbolic executing engine

At that point, I can assure you that we need some tooling: we have studied the binary, we know how the main parts work and we just need to extract the different equations/formulas used by both the computation of the son's program counter and the serial verification algorithm. Basically the engine is going to be useful to study both the father and the son.

If you are not really familiar with symbolic execution, I recommend you take a little bit of time to read Breaking Kryptonite's Obfuscation: A Static Analysis Approach Relying on Symbolic Execution and check out z3-playground if you are not really familiar with Z3 and its Python bindings.

This time I decided to not build that engine as an IDA Python script, but just to do everything myself. Do not be afraid though, even if it sounds scary it is really not: the challenge is a perfect environment for those kind of things. It doesn't use a lot of instructions, we don't need to support branches and nearly only arithmetic instructions are used.

I also chose to implement this engine in a way that we can also use it as a simple emulator. You can even use it as a decompiler if you want! The two other interesting points for us are:

1. Once we run a piece of code in the symbolic engine, we will *extract* certain computations / formulas. Thanks to Microsoft's Z3 we will be able to retrieve input values that will generate specific output values: this is basically what you gain by using a solver and symbolic variables.
2. But the other interesting point is that you still can use the extracted Z3 expressions as some kind of black-box *functions*. You know what the function is doing, kind of, but you don't know how ; and you are not interested in the how. You know the inputs, and the outputs. To obtain a concrete output value, you can just replace the symbolic variables by concrete values. This is really handy, especially when you are not only interested in finding input values to generate specific output values ; sometimes you just want to go both ways :–).

Anyway, after this long theoretical speech let's have a look at some code. The first important job of the engine is to be able to parse MIPS assembly: fortunately for us this is really easy. We are directly feeding plain-text MIPS disassembly directly copied from IDA to our engine:

> mini_mips_symexec_engine.py@MiniMipsSymExecEngine._parse_line

```
1  def _parse_line(self, line):
2    addr_seg, instr, rest = line.split(None, 2)
3    args = rest.split(',')
4    for i in range(len(args)):
5      if '#' in args[i]:
6          args[i], _ = args[i].split(None, 1)
```

```
7
8    a0, a1, a2 = map(
9      lambda x: x.strip().replace('$', '') if x is not None else x,
10     args + [None]*(3 - len(args))
11   )
12   _, addr = addr_seg.split(':')
13   return int(addr, 16), instr, a0, a1, a2
```

From here you have all the information you need: the instruction and its operands (*None* if an operand doesn't exist as you can have up to 3 operands). The other important job that follows is to handle the different type of operands ; here are the ones I encountered in the challenge:

- General purpose register,
- Stack-variable,
- Immediate value.

To handle / convert those I created a bunch of dull / helper functions:

mini_mips_symexec_engine.py@MiniMipsSymExecEngine._is_*

```
1  def _is_gpr(self, x):
2    '''Is it a valid GPR name?'''
3    return x in self.gpr
4
5  def _is_imm(self, x):
6    '''Is it a valid immediate?'''
7    x = x.replace('loc_', '0x')
8    try:
9      int(x, 0)
10     return True
11   except:
12     return False
13
14 def _to_imm(self, x):
15   '''Get an integer from a string immediate'''
16   if self._is_imm(x):
17     x = x.replace('loc_', '0x')
18     return int(x, 0)
19   return None
20
21 def _is_memderef(self, x):
22   '''Is it a memory dereference?'''
23   return '(' in x and ')' in x
24
25 def is_stackvar(self, x):
26   '''Is is a stack variable?'''
27   return ('(fp)' in x and '+' in x) or ('var_' in x and '+' in x)
28
29 def to_stackvar(self, x):
30   '''Get the stack variable name'''
31   _, var_name = x.split('+')
32   return var_name.replace('(fp)', '')
```

Finally, we have to handle every different instructions and their encodings. Of course, you need to implement only the instructions you want: most likely the ones that are used in the code you are interested int. In a nutshell, this is the core of the engine. You can also use it to output valid Python/C lines if you fancy having a decompiler in your sleeve ; might be handy right?

This is what the core function looks like, it is really simple, dumb and so unoptimized ; but at least it's clear to me:

mini_mips_symexec_engine.py@step

```
def step(self):
  '''This is the core of the engine -- you are supposed to implement the sema
  of all the instructions you want to emulate here.'''
  line = self.code[self.pc]
  addr, instr, a0, a1, a2 = self._parse_line(line)
  if instr == 'sw':
    if self._is_gpr(a0) and self.is_stackvar(a1) and a2 is None:
      var_name = self.to_stackvar(a1)
      self.logger.info('%s = $%s', var_name, a0)
      self.stack[var_name] = self.gpr[a0]
    elif self._is_gpr(a0) and self._is_memderef(a1) and a2 is None:
      idx, base = a1.split('(')
      base = base.replace('$', '').replace(')', '')
      computed_address = self.gpr[base] + self._to_imm(idx)
      self.logger.info('[%s + %s] = $%s', base, idx, a0)
      self.mem[computed_address] = self.gpr[a0]
    else:
      raise Exception('sw not implemented')
  elif instr == 'lw':
    if self._is_gpr(a0) and self.is_stackvar(a1) and a2 is None:
      var_name = self.to_stackvar(a1)
      if var_name not in self.stack:
        self.logger.info(' WARNING: Assuming %s was 0', (var_name, ))
        self.stack[var_name] = 0
      self.logger.info('$%s = %s', a0, var_name)
      self.gpr[a0] = self.stack[var_name]
    elif self._is_gpr(a0) and self._is_memderef(a1) and a2 is None:
      idx, base = a1.split('(')
      base = base.replace('$', '').replace(')', '')
      computed_address = self.gpr[base] + self._to_imm(idx)
      if computed_address not in self.mem:
        value = raw_input(' WARNING %.8x is not in your memory store -- what
      else:
        value = self.mem[computed_address]
      self.logger.info('$%s = [%s+%s]', a0, idx, base)
      self.gpr[a0] = value
    else:
      raise Exception('lw not implemented')
[...]
```

The first level of *if* handles the different instructions, the second level of *if* handles the different encodings an instruction can have. The *self.logger* thingy is just my way to save the execution traces in files to let the console clean:

mini_mips_symexec_engine.py@__init__

```
def __init__(self, trace_name):
  self.gpr = {
    'zero' : 0,
    'at' : 0,
    'v0' : 0,
    'v1' : 0,
# [...]
    'lo' : 0,
    'hi' : 0
  }

  self.stack = {}
  self.pc = 0
  self.code = []
  self.mem = {}
  self.stack_offsets = {}
```

```
17   self.debug = False
18   self.enable_z3 = False
19
20   if os.path.exists('traces') == False:
21       os.mkdir('traces')
22
23   self.logger = logging.getLogger(trace_name)
24   h = logging.FileHandler(
25       os.path.join('traces', trace_name),
26       mode = 'w'
27   )
28
29   h.setFormatter(
30       logging.Formatter(
31           '%(levelname)s: %(asctime)s %(funcName)s @ l%(lineno)d -- %(messag
32           datefmt = '%Y-%m-%d %H:%M:%S'
33       )
34   )
35
36   self.logger.setLevel(logging.INFO)
37   self.logger.addHandler(h)
```

At that point, if I wanted only an emulator I would be done. But because I want to use Z3 and symbolic variables I want to get your attention on two common pitfalls that can cost you hours of debugging (trust me on that one :–():

- The first one is that the operator *__rshift__* isn't the logical right shift but the arithmetical one; which is quite different and can generate results you don't expect:

  LShR VS >>

```
1  In [1]: from z3 import *
2
3  In [2]: simplify(BitVecVal(4, 3) >> 1)
4  Out[2]: 6
5
6  In [3]: simplify(LShR(BitVecVal(4, 3), 1))
7  Out[3]: 2
8
9  In [4]: 4 >> 1
10 Out[4]: 2
```

To workaround that I usually define my own *_LShR* function that does whatever is correct according to the operand types (yes we could also replace *z3.BitVecNumRef.__rshift__* by *LShR* directly):

mini_mips_symexec_engine@_LShR

```
1  def _LShR(self, a, b):
2    '''Useful hook function if you want to run the emulation
3    with/without Z3 as LShR is different from >> in Z3'''
4    if self.enable_z3:
5      if isinstance(a, long) or isinstance(a, int):
6        a = BitVecVal(a, 32)
7      if isinstance(b, long) or isinstance(b, int):
8        b = BitVecVal(b, 32)
9      return LShR(a, b)
10   return a >> b
```

- The other interesting detail to keep in mind is that you can't have any overflow on *BitVec*s of the same size ; the result is automatically truncated. So if you happen to have mathematical

operations that need to overflow, like a multiplication (this is used in the challenge), you should store the temporary result in a bigger temporary variable. In my case, I was supposed to store the overflow inside another register, *$hi* which is used to store the high *DWORD* part of the result. But because I wasn't storing the result in a bigger *BitVec*, *$hi* ended up **always** equal to zero which is quite a nice problem when you have to pinpoint this issue in thousands lines of assembly :–).

mini_mips_symexec_engine@step@multu

```
1  elif instr == 'multu':
2    if self._is_gpr(a0) and self._is_gpr(a1) and a2 is None:
3      self.logger.info('$lo = ($%s * $%s) & 0xffffffff', a0, a1)
4      self.logger.info('$hi = ($%s * $%s) >> 32', a0, a1)
5      if self.enable_z3:
6        a0bis, a1bis = self.gpr[a0], self.gpr[a1]
7        if isinstance(a0bis, int) or isinstance(a0bis, long):
8          a0bis = BitVecVal(a0bis, 32)
9        if isinstance(a1bis, int) or isinstance(a1bis, long):
10         a1bis = BitVecVal(a1bis, 32)
11
12       a064 = ZeroExt(32, a0bis)
13       a164 = ZeroExt(32, a1bis)
14       r = a064 * a164
15       self.gpr['lo'] = Extract(31, 0, r)
16       self.gpr['hi'] = Extract(63, 32, r)
17   else:
18     x = self.gpr[a0] * self.gpr[a1]
19     self.gpr['lo'] = x & 0xffffffff
20     self.gpr['hi'] = self._LShR(x, 32)
```

I think this is it really, you can now impress girls with your brand new shiny toy, check this out:

mini_mips_symexec_engine@main

```
1  def main(argc, argv):
2      print '=' * 50
3      sym = MiniMipsSymExecEngine('donotcare.log')
4      # DO NOT FORGET TO ENABLE Z3 :)
5      sym.enable_z3 = True
6      a = BitVec('a', 32)
7      sym.stack['var'] = a
8      sym.stack['var2'] = 0xdeadbeef
9      sym.stack['var3'] = 0x31337
10     sym.code = '''.doare:DEADBEEF                  lw      $v0, 0x318+var($fp
11 .doare:DEADBEEF                  lw      $v1, 0x318+var2($fp)  # Load Word
12 .doare:DEADBEEF                  subu    $v0, $v1, $v0    #
13 .doare:DEADBEEF                  li      $v1, 0x446F8657  # Load Immediate
14 .doare:DEADBEEF                  multu   $v0, $v1         # Multiply Unsigned
15 .doare:DEADBEEF                  mfhi    $v1              # Move From HI
16 .doare:DEADBEEF                  subu    $v0, $v1         # Subtract Unsigned
17     sym.run()
18
19     print 'Symbolic mode:'
20     print 'Resulting equation: %r' % sym.gpr['v0']
21     print 'Resulting value if `a` is 0xdeadb44: %#.8x' % substitute(
22         sym.gpr['v0'], (a, BitVecVal(0xdeadb44, 32))
23     ).as_long()
24
25     print '=' * 50
26     emu = MiniMipsSymExecEngine('donotcare.log')
27     emu.stack = sym.stack
28     emu.stack['var'] = 0xdeadb44
29     sym.stack['var2'] = 0xdeadbeef
```

```
30     sym.stack['var3'] = 0x31337
31     emu.code = sym.code
32     emu.run()
33
34     print 'Emulator mode:'
35     print 'Resulting value when `a` is 0xdeadb44: %#.8x' % emu.gpr['v0']
36     print '=' * 50
37     return 1
```

Which results in:

w00t, emu & symbolic execution works

```
1  PS D:\Codes\NoSuchCon2014> python .\mini_mips_symexec_engine.py
2  ==================================================
3  Symbolic mode:
4  Resulting equation: 3735928559 +
5  4294967295*a +
6  4294967295*
7  Extract(63,
8          32,
9          1148159575*Concat(0, 3735928559 + 4294967295*a))
10 Resulting value if `a` is 0xdeadb44: 0x98f42d24
11 ==================================================
12 Emulator mode:
13 Resulting value when `a` is 0xdeadb44: 0x98f42d24
14 ==================================================
```

Of course, I didn't mention a lot of details that still need to be addressed to have something working: simulating data areas, memory layouts, etc. If you are interested in those, you should read the codes in my NoSuchCon2014 folder.

Back into the battlefield

Here comes the important bits!

Extracting the function that generates the magic value from the son program counter

All right, the main objective in this part is to extract the formula that generates the first magic value. As we said earlier, this big block can be seen as a function that takes two arguments (or symbolic variables) and generates the *magic DWORD* in output. The first thing to do is to copy the code somewhere to feed it to our engine ; I decided to stick all the codes I needed into a separate Python file called *code.py*.

code.py

```
1  block_generate_magic_from_pc_son = '''.text:00400B8C                 lw
2  .text:00400B90                 sw      $v0, 0x318+tmp_pc($fp)  # Store Word
3  .text:00400B94                 la      $v0, loc_400A78  # Load Address
4  .text:00400B9C                 lw      $v1, 0x318+tmp_pc($fp)  # Load Word
5  .text:00400BA0                 subu    $v0, $v1, $v0    # (regs.pc_father -
6  .text:00400BA4                 sw      $v0, 0x318+tmp_pc($fp)  # Store Word
7  .text:00400BA8                 lw      $v0, 0x318+var_300($fp)  # Load Word
8  .text:00400BAC                 li      $v1, 0x446F8657  # Load Immediate
9  .text:00400BB4                 multu   $v0, $v1         # Multiply Unsigned
10 .text:00400BB8                 mfhi    $v1              # Move From HI
11 .text:00400BBC                 subu    $v0, $v1         # Subtract Unsigned
```

```
12 [...]
13 .text:00401424                lw      $v0, 0x318+var_2F0($fp)  # Load Word
14 .text:00401428                nor     $v0, $zero, $v0  # NOR
15 .text:0040142C                addiu   $v0, 0x20        # Add Immediate Unsi
16 .text:00401430                lw      $a0, 0x318+tmp_pc($fp)  # Load Word
17 .text:00401434                sllv    $v0, $a0, $v0    # Shift Left Logical
18 .text:00401438                or      $v0, $v1, $v0    # OR
19 .text:0040143C                sw      $v0, 0x318+tmp_pc($fp)  # Store Word'
```

Then we have to prepare the environment of our engine: the two symbolic variables are stack-variables, so we have to insert them in the context of our virtual environment. The resulting formula is going to be in *$v0* at the end of the execution ; this the holy grail, the formula we are after.

solve_nsc2014_step1_z3.py@extract_equation_of_function_that_generates_magic_value

```
1  def extract_equation_of_function_that_generates_magic_value():
2    '''Here we do some magic to transform our mini MIPS emulator
3    into a symbolic execution engine ; the purpose is to extract
4    the formula of the function generating the 32-bits magic value'''
5
6    x = mini_mips_symexec_engine.MiniMipsSymExecEngine('function_that_generates
7    x.debug = False
8    x.enable_z3 = True
9    pc_son = BitVec('pc_son', 32)
10   n_break = BitVec('n_break', 32)
11   x.stack['pc_son'] =  pc_son
12   x.stack['var_300'] = n_break
13   emu_generate_magic_from_son_pc(x, print_final_state = False)
14   compute_magic_equation = x.gpr['v0']
15   with open(os.path.join('formulas', 'generate_magic_value_from_pc_son.smt2'
16     f.write(to_SMT2(compute_magic_equation, name = 'generate_magic_from_pc_s
17
18   return pc_son, n_break, simplify(compute_magic_equation)
```

You can now keep in memory the formula & wrap this function in another one so that you can reuse it every time you need it:

solve_nsc2014_step1_z3.py@generate_magic_from_son_pc_using_z3

```
1  var_magic, var_n_break, expr_magic = [None]*3
2  def generate_magic_from_son_pc_using_z3(pc_son, n_break):
3    '''Generates the 32 bits magic value thanks to the output
4    of the symbolic execution engine: run the analysis once, extract
5    the complete equation & reuse it as much as you want'''
6    global var_magic, var_n_break, expr_magic
7    if var_magic is None and var_n_break is None and expr_magic is None:
8      var_magic, var_n_break, expr_magic = extract_equation_of_function_that_g
9
10   return substitute(
11     expr_magic,
12     (var_magic, BitVecVal(pc_son, 32)),
13     (var_n_break, BitVecVal(n_break, 32))
14   ).as_long()
```

The power of using symbolic variables here lies in the fact that we don't need to run the emulator every single time you need to call this function ; you get once the generic formula and you just have to substitute the symbolic variables by the concrete values you want. This comes for free with our code, so let's use it heh :–).

resulting formula in SMT2 format

```
; generate_magic_from_pc_son
(declare-fun n_break () (_ BitVec 32))
(declare-fun pc_son () (_ BitVec 32))
(let ((?x14 (bvadd n_break (bvmul (_ bv4294967295 32) ((_ extract 63 32) (bv
(let ((?x21 ((_ extract 63 32) (bvmul (_ bv1148159575 64) (concat (_ bv0 32)
(let ((?x8 (bvadd ?x21 (concat (_ bv0 1) ((_ extract 31 1) ?x14)))))
(let ((?x26 ((_ extract 31 6) ?x8)))
(let ((?x24 (bvadd (_ bv32 32) (concat (_ bv63 6) (bvnot ?x26)))))
(let ((?x27 (concat (_ bv0 6) ?x26)))
(let ((?x42 (bvmul (_ bv4294967295 32) ?x27)))
(let ((?x67 ((_ extract 6 6) ?x8)))
(let ((?x120 ((_ extract 7 6) ?x8)))
(let ((?x38 (concat (bvadd (_ bv30088 15) ((_ extract 14 0) pc_son)) ((_ ext
(let ((?x41 (bvxor (bvadd (bvor (bvlshr ?x38 (bvadd (_ bv1 32) ?x27)) (bvshl
(let ((?x63 (bvor ((_ extract 0 0) (bvlshr ?x38 (bvadd (_ bv1 32) ?x27))) ((_
(let ((?x56 (concat (bvadd (_ bv1 1) (bvxor (bvadd ?x63 ?x67) ?x67)) ((_ ext
(let ((?x66 (concat (bvadd ((_ extract 9 1) (bvadd (_ bv2142377237 32) ?x41)
(let ((?x118 (bvor ((_ extract 1 0) (bvshl ?x66 (bvadd (_ bv1 32) ?x27))) ((_
(let ((?x122 (bvnot (bvadd ?x118 ?x120))))
(let ((?x45 (bvadd (bvor (bvshl ?x66 (bvadd (_ bv1 32) ?x27)) (bvlshr ?x66 ?x
(let ((?x76 ((_ extract 4 2) ?x45)))
(let ((?x110 (bvnot ((_ extract 5 5) ?x45))))
(let ((?x55 ((_ extract 8 6) ?x45)))
(let ((?x108 (bvnot ((_ extract 10 9) ?x45))))
(let ((?x78 ((_ extract 13 11) ?x45)))
(let ((?x106 (bvnot ((_ extract 14 14) ?x45))))
(let ((?x80 ((_ extract 15 15) ?x45)))
(let ((?x104 (bvnot ((_ extract 16 16) ?x45))))
(let ((?x123 (concat (bvnot ((_ extract 31 29) ?x45)) ((_ extract 28 28) ?x45
(let ((?x50 (concat (bvnot ((_ extract 30 29) ?x45)) ((_ extract 28 28) ?x45
(let ((?x91 (bvadd (_ bv1720220585 32) (concat (bvnot (bvadd (_ bv612234822
(let ((?x137 (bvnot (bvadd (_ bv128582 17) (concat ?x104 ?x80 ?x106 ?x78 ?x1
(let ((?x146 (bvadd (_ bv31657 18) (concat ?x137 (bvnot ((_ extract 31 31) (
(let ((?x131 (bvadd (_ bv2800103692 32) (concat ?x146 ((_ extract 31 18) ?x9
(let ((?x140 (concat ((_ extract 18 18) ?x91) ((_ extract 31 31) ?x131) (bvn
(let ((?x176 (bvnot (bvadd (concat ((_ extract 4 4) ?x131) (bvnot ((_ extract
(let ((?x177 (bvadd (concat ?x176 (bvnot ((_ extract 31 4) (bvadd ?x140 ?x27
(let ((?x187 (bvadd (bvnot ((_ extract 13 4) (bvadd ?x140 ?x27))) (bvmul (_
(let ((?x180 (concat (bvadd ((_ extract 23 10) ?x177) (bvmul (_ bv16383 14)
(let ((?x79 (bvadd (bvxor (bvadd ?x180 ?x27) ?x27) ?x42)))
(let ((?x211 (concat (bvadd ((_ extract 17 10) ?x177) (bvmul (_ bv255 8) ((_
(let ((?x190 (concat (bvnot (bvadd (bvxor (bvadd ?x211 ?x26) ?x26) (bvmul (_
(let ((?x173 (bvadd (bvnot (bvadd (_ bv3113082326 32) ?x190 ?x27)) ?x27)))
(let ((?x174 ((_ extract 9 6) ?x8)))
(let ((?x255 ((_ extract 2 2) (bvadd (bvnot (bvadd (_ bv6 4) (bvnot ((_ extra
(let ((?x253 ((_ extract 3 3) (bvadd (bvnot (bvadd (_ bv6 4) (bvnot ((_ extra
(let ((?x144 ((_ extract 23 6) ?x8)))
(let ((?x233 ((_ extract 17 6) ?x8)))
(let ((?x235 (bvxor (bvadd ((_ extract 25 14) (bvadd (concat ?x187 ((_ extrac
(let ((?x244 (bvadd (_ bv122326 18) (concat (bvnot (bvadd ?x235 (bvmul (_ bv4
(let ((?x246 (bvadd (bvnot ?x244) ?x144)))
(let ((?x293 (concat (bvnot ((_ extract 24 23) ?x173)) ((_ extract 22 18) ?x1
(let ((?x324 (bvor ((_ extract 0 0) (bvshl ?x293 (bvadd (_ bv1 32) ?x27))) (
(let ((?x202 (bvadd (bvor (bvshl ?x293 (bvadd (_ bv1 32) ?x27)) (bvlshr ?x29
(let ((?x261 (concat ((_ extract 31 31) ?x202) (bvnot ((_ extract 30 29) ?x20
(let ((?x250 (concat ((_ extract 11 7) ?x202) (bvnot ((_ extract 6 5) ?x202)
(let ((?x331 (bvadd (_ bv1397077939 32) (concat (bvadd (_ bv4018 12) ?x250)
(let ((?x264 (bvor (bvshl (bvadd (bvnot ?x331) ?x27) (bvadd (_ bv1 32) ?x27)
(let ((?x298 (bvor (bvshl (bvadd (_ bv1031407080 32) ?x264 ?x42) (bvadd (_ bv
(let ((?x231 (bvor ((_ extract 31 17) (bvshl ?x298 (bvadd (_ bv1 32) ?x27)))
(let ((?x220 (bvor ((_ extract 16 0) (bvshl ?x298 (bvadd (_ bv1 32) ?x27)))
(let ((?x283 (bvor (bvshl (concat ?x220 ?x231) (bvadd (_ bv1 32) ?x27)) (bvls
(let ((?x119 (bvadd (_ bv4200859627 32) (bvnot (bvor (bvshl ?x283 (bvadd (_
(let ((?x201 (bvshl ?x119 ?x24)))
(let ((?x405 (bvadd (bvor ((_ extract 10 8) (bvlshr ?x119 (bvadd (_ bv1 32)
```

```
66 (let ((?x343 (concat (bvor ((_ extract 7 0) (bvlshr ?x119 (bvadd (_ bv1 32) 
67 (let ((?x199 (bvadd (_ bv752876532 32) (bvnot (bvadd ?x343 ?x27)) ?x27)))
68 (let ((?x409 (concat ((_ extract 31 29) ?x199) (bvnot ((_ extract 28 28) ?x1
69 (let ((?x342 (bvlshr (bvadd (_ bv330202175 32) ?x409) ?x24)))
70 (let ((?x20 (bvadd (_ bv1 32) ?x27)))
71 (let ((?x337 (bvshl (bvadd (_ bv330202175 32) ?x409) ?x20)))
72 (let ((?x354 (bvadd (_ bv651919116 32) (bvor ?x337 ?x342))))
73 (let ((?x414 (concat (bvnot ((_ extract 26 26) ?x354)) ((_ extract 25 25) ?x
74 (let ((?x464 (concat ((_ extract 22 22) ?x354) (bvnot ((_ extract 21 21) ?x3
75 (let ((?x474 (concat (bvadd (_ bv141595581 28) (bvnot (bvxor (bvadd (_ bv178
76 (let ((?x495 (bvadd (_ bv1994801052 32) (bvxor (_ bv1407993787 32) (bvor (bv
77 (let ((?x392 (concat (bvor ((_ extract 13 0) (bvlshr ?x495 ?x20)) ((_ extract
78 (let ((?x388 (bvlshr ?x392 ?x24)))
79 (let ((?x494 (concat (bvnot (bvor ((_ extract 31 31) (bvshl ?x392 ?x20)) ((_
80 (let ((?x450 (bvor (bvlshr ?x494 ?x20) (bvshl ?x494 ?x24))))
81 (bvor (bvlshr ?x450 ?x20) (bvshl ?x450 ?x24))))))))))))))))))))))))))))))))
```

Quite happy we don't have to study that right?

Extracting the function that generates the new program counter from the second magic value

For the second big block of code, we can do exactly the same thing: copy the code, configure the virtual environment with our symbolic variables and wrap the function:

solve_nsc2014_step1_z3.py@generate_new_pc_from_magic_high/extract_equation_of_functio

```
1  def extract_equation_of_function_that_generates_new_son_pc():
2    '''Extract the formula of the function generating the new son's $pc'''
3    x = mini_mips_symexec_engine.MiniMipsSymExecEngine('function_that_generates
4    x.debug = False
5    x.enable_z3 = True
6    tmp_pc = BitVec('magic', 32)
7    n_loop = BitVec('n_loop', 32)
8    x.stack['tmp_pc'] = tmp_pc
9    x.stack['var_2F0'] = n_loop
10   emu_generate_new_pc_for_son(x, print_final_state = False)
11   compute_pc_equation = simplify(x.gpr['v0'])
12   with open(os.path.join('formulas', 'generate_new_pc_son.smt2'), 'w') as f:
13     f.write(to_SMT2(compute_pc_equation, name = 'generate_new_pc_son'))
14
15   return tmp_pc, n_loop, compute_pc_equation
16
17 var_new_pc, var_n_loop, expr_new_pc = [None]*3
18 def generate_new_pc_from_magic_high(magic_high, n_loop):
19   global var_new_pc, var_n_loop, expr_new_pc
20   if var_new_pc is None and var_n_loop is None and expr_new_pc is None:
21     var_new_pc, var_n_loop, expr_new_pc = extract_equation_of_function_that_
22
23   return substitute(
24       expr_new_pc,
25       (var_new_pc, BitVecVal(magic_high, 32)),
26       (var_n_loop, BitVecVal(n_loop, 32))
27   ).as_long()
```

If you are interested in what the formula looks like, it is also available in the NoSuchCon2014 folder on my github.

Putting it all together: building a function that computes the new program counter of the son

Obviously, we don't really care about those two previous functions, we just want to combine them together to implement the computation of the new program counter from both the round number & where the son *SIGTRAP*'d. The only missing bits is the lookup in the *QWORD*s array to extract the *second magic value*. We just have to dump the array inside another file called *memory.py*. This is done with a simple IDA Python one-liner:

Dump the QWORD array with IDAPy

```
1 values = dict((0x00414130+i*8, Qword(0x00414130+i*8)) for i in range(0x25E))
```

Now, we can build the whole function easily by combining all those pieces:

solve_nsc2014_step1_z3.py@generate_new_pc_from_pc_son_using_z3

```
1  def generate_new_pc_from_pc_son_using_z3(pc_son, n_break):
2    '''Generate the new program counter from the address where the son SIGTRAP
3    the number of SIGTRAP the son encountered'''
4    loop_n = (n_break / 101)
5    magic = generate_magic_from_son_pc_using_z3(pc_son, n_break)
6    idx = None
7    for i in range(len(memory.pcs)):
8      if (memory.pcs[i] & 0xffffffff) == magic:
9        idx = i
10       break
11
12   assert(idx != None)
13   return generate_new_pc_from_magic_high(memory.pcs[idx] >> 32, loop_n)
```

Sweet. Really sweet.

This basically means we are now able to *unscramble* the code of the son and reordering it completely without even physically running the binary nor generating traces.

Unscramble the code like a sir

Before showing, the code I just want to explain the process one more time:

1. The son executes some code until it reaches a *break* instruction
2. The father gets the *$pc* of the son and the variable that counts the number of *break* instruction the son executed
3. The father generates a new *$pc* value from those two variables
4. The father sets the new *$pc*
5. The father continues its son
6. Goto 1!

So basically to unscramble the code, we just need to simulate what the father would do & log everything somewhere. Couple of important details though:

- There are exactly 101 *break* instructions in the son, so 101 *chunks* of code will be executed and need to be *reordered*,
- The son is executing 6 *rounds* ; that's exactly why the *QWORD* array has 6*101 entries.

Here is the function I used:

solve_nsc2014_step1_z3.py@generate_son_code_reordered

```
1  def generate_son_code_reordered(debug = False):
2      '''This functions puts in the right order the son's block of codes witho
3      relying on the father to set a new $pc value when a break is executed in
4      With this output we are good to go to create a nanomites-less binary:
5        - We don't need the father anymore (he was driving the son)
6        - We have the code in the right order, so we can also remove the break
7      It will also be quite useful when we want to execute symbolic-ly its cod
8      '''
9      def parse_line(l):
10         addr_seg, instr, _ = l.split(None, 2)
11         _, addr = addr_seg.split(':')
12         return int('0x%s' % addr, 0), instr
13
14     son_code = code.block_code_of_son
15     next_break = 0
16     n_break = 0
17     cleaned_code = []
18     for _ in range(6):
19         for z in range(101):
20             i = 0
21             while i < len(son_code):
22                 line = son_code[i]
23                 addr, instr = parse_line(line)
24                 if instr == 'break' and (next_break == addr or z == 0):
25                     break_addr = addr
26                     new_pc = generate_new_pc_from_pc_son_using_z3(break_addr
27                     n_break += 1
28                     if debug:
29                         print '; Found the %dth break (@%.8x) ; new pc will l
30                     state = 'Begin'
31                     block = []
32                     j = 0
33                     while j < len(son_code):
34                         line = son_code[j]
35                         addr, instr = parse_line(line)
36                         if state == 'Begin':
37                             if addr == new_pc:
38                                 block.append(line)
39                                 state = 'Log'
40                         elif state == 'Log':
41                             if instr == 'break':
42                                 next_break = addr
43                                 state = 'End'
44                             else:
45                                 block.append(line)
46                         elif state == 'End':
47                             break
48                         else:
49                             pass
50                         j += 1
51
52                     if debug:
53                         print ';', '='*25, 'BLOCK %d' % z, '='*25
54                         print '\n'.join(block)
55                     cleaned_code.extend(block)
56                     break
57                 i += 1
58
59     return cleaned_code
```

And there it is :–)

The function outputs the unrolled and ordered code of the son. If you want to push further, you could

theoretically perform an open-heart surgery to completely remove the nanomites from the original binary, isn't it cool? This is left as an exercise for the interested reader though :–)).

Attacking the son: the last man standing

Now that we have the code unscrambled, we can directly feed it to our engine but before doing so here are some details:

- As we said earlier, it looks like the son is executing 6 times the same code. This is not the case **at all**, every round will execute the same amount of instructions but not in the same order
- The computations executed can be seen as some kind of light encoding/encryption or decoding/decryption algorithm
- We have 6 *rounds* because the input serial is broken into 6 *DWORD*s (so 6 symbolic variables) ; so basically each round is going to generate an output *DWORD*

As previously, we need to copy the code we want to execute. Note that we can also use *generate_son_code_reorganized* to generate it dynamically. Next step is to configure the virtual environment and we are good to finally run the code:

solve_nsc2014_step1_z3@get_serial first part

```
1  def get_serial():
2    print '> Instantiating the symbolic execution engine..'
3    x = mini_mips_symexec_engine.MiniMipsSymExecEngine('decrypt_serial.log')
4    x.enable_z3 = True
5
6    print '> Generating dynamically the code of the son & reorganizing/cleanin
7    # If you don't want to generate it dynamically like a sir, I've copied a ve
8    # code.block_code_of_son_reorganized_loop_unrolled :-)
9    x.code = generate_son_code_reorganized()
10
11   print '> Configuring the virtual environement..'
12   x.gpr['fp'] = 0x7fff6cb0
13   x.stack_offsets['var_30'] = 24
14   start_addr = x.gpr['fp'] + x.stack_offsets['var_30'] + 8
15   # (gdb) x/6dwx $s8+24+8
16   # 0x7fff6cd0:     0x11111111      0x11111111      0x11111111
17   #                 0x11111111      0x11111111      0x11111111
18   a, b, c, d, e, f = BitVecs('a b c d e f', 32)
19   x.mem[start_addr +  0] = a
20   x.mem[start_addr +  4] = b
21   x.mem[start_addr +  8] = c
22   x.mem[start_addr + 12] = d
23   x.mem[start_addr + 16] = e
24   x.mem[start_addr + 20] = f
25
26   print '> Running the code..'
27   x.run()
```

The thing that matters this time is to find a, b, c, d, e, f so that they generate specific outputs ; so this is where Z3 is going to help us a **lot**. Thanks to that guy we don't need to manually invert the algorithm.

The final bit now is basically just about setting up the solver, setting the correct constraints and generating the serial you guys have been waiting for so long:

solve_nsc2014_step1_z3@get_serial second part

```
print '> Instantiating & configuring the solver..'
s = Solver()
s.add(
  x.mem[start_addr +   0] == 0x7953205b, x.mem[start_addr +   4] == 0x6b63
  x.mem[start_addr +   8] == 0x20766974, x.mem[start_addr +  12] == 0x534e
  x.mem[start_addr +  16] == 0x203d2043, x.mem[start_addr +  20] == 0x5d20
)

print '> Solving..'
if s.check() == sat:
  print '> Constraints solvable, here are the 6 DWORDs:'
  m = s.model()
  for i in (a, b, c, d, e, f):
    print ' %r = 0x%.8X' % (i, m[i].as_long())

  print '> Serial:', ''.join(('%.8x' % m[i].as_long())[::-1] for i in (a, 
else:
  print '! Constraints unsolvable'
```

There we are, the final moment; *drum roll*

python solve_nsc2014_step1_z3.py + YAY

```
PS D:\Codes\NoSuchCon2014> python .\solve_nsc2014_step1_z3.py
==================================================
Tests OK -- you are fine to go
==================================================
> Instantiating the symbolic execution engine..
> Generating dynamically the code of the son & reorganizing/cleaning it..
> Configuring the virtual environement..
> Running the code..
> Instantiating & configuring the solver..
> Solving..
> Constraints solvable, here are the 6 DWORDs:
 a = 0xFE446223
 b = 0xBA770149
 c = 0x75BA5111
 d = 0x78EA3635
 e = 0xA9D6E85F
 f = 0xCC26C5EF
> Serial: 322644EF941077AB1115AB575363AE87F58E6D9AFE5C62CC
==================================================

overclok@wildout:~/chall/nsc2014$ ./start_vm.sh
[    0.000000] Initializing cgroup subsys cpuset
[...]
Debian GNU/Linux 7 debian-mipsel ttyS0

debian-mipsel login: root
Password:
[...]
root@debian-mipsel:~# /home/user/crackmips 322644EF941077AB1115AB575363AE87F
good job!
Next level is there: http://nsc2014.synacktiv.com:65480/oob4giekee4zaeW9/
```

Boom :–).

Alternative solution

In this part, I present an alternate solution to solve the challenge. It's somehow a shortcut, since it requires much less coding than Axel's one, and uses the awesome Miasm framework.

Shortcut #1 : Tracing the parent with GDB

Quick recap of the parent's behaviour

As Axel has previously explained, the first step is to recover the child's execution flow. Because of *nanomites*, the child is driven by the parent; we have to analyze the parent (i.e. the `debug` function) first to determine the correct sequence of the child's `pc` values.

The parent's main loop is obfuscated, but by browsing cross-references of stack variables in IDA, we can see where each one is used. After a bit of analysis, we can try to decompile by hand the algorithm, and write a pseudo-Python code description of what the `debug` function does (it is really simplified):

debug_pseudo_code.py

```
1  counter = 0
2  waitpid()
3
4  while(True):
5      regs = ptrace(GETREGS)
6
7      # big block 1
8      addr = regs.pc
9      param = f(counter)
10     addr = obfu1(addr, param)
11
12     for i in range(605):
13         entry = pcs[i]  # entry is 8 bytes long (2 dwords)
14         if(addr == entry.first_dword):
15             addr = entry.second_dword
16             break
17
18     # big block 2
19     addr = obfu2(addr, param)
20
21     regs.pc = addr
22     ptrace(SETREGS, regs)
23     counter += 1
24
25     if(not waitpid()):
26         break
```

The "big blocks" are the two long assembly blocks preceding and following the inner loop. Without looking at the gory details, we understand that a `param` value is derived from the counter using a function that I call `f`, and then used to obfuscate the original child's `pc`. The result is then searched in a `pcs` array (stored at address `0414130`), the next dword is extracted and used in a 2nd obfuscation pass to finally produce the new `pc` value injected into the child.

The most important fact here is that that this process does not involve the input key at anytime. **The output `pc` sequence is deterministic and constant**; two executions with two different keys will produce the same sequence of `pc`'s. Since we know the first value of `pc` (the first `break` instruction at 040228C), we can theoretically compute the correct sequence and then reorder the child's instructions according to this sequence.

We have two approaches for doing so:

- statical analysis: somehow understand each instruction used in obfuscation passes and rewrite

the algorithm producing the correct sequence. This is the path followed by Axel.
- dynamic analysis: trace the program once and log all pc values.

Although the first one is probably the most interesting, the second is certainly the fastest. Again, it only works because the input key does not influence the output `pc` sequence. And we're lucky: the child is already debugged by the parent, but nothing prevents us to debug the parent itself.

First attempt at tracing

Tracing is pretty straightforward with GDB using `bp` and `commands`. In order to understand the parent's algorithm a bit better, I first wrote a pretty verbose GDB script that prints the loop counter, `param` variable as well as the original and new child's `pc` for each iteration. I chose to put two breakpoints:

- The first one at the end of the first obfuscation blocks (0x401440)
- The second one before the `ptrace` call at the end of the second block (0x0401D8C), in order to be able to read the child's `pc` manipulated by the parent.

Here is the script:

gdb_trace1_script.txt

```
1  ###################################
2  # A few handy functions
3  ###################################
4
5  def print_context_pc
6      printf "regs.pc = 0x%08x\n", *(int*)($fp-0x1cc)
7  end
8
9  def print_param
10     printf "param = 0x%08x\n", *(int*)($fp-0x2f0)
11 end
12
13 def print_addr
14     printf "addr = 0x%08x\n", *(int*)($fp-0x2fc)
15 end
16
17 def print_counter
18     printf "counter = %d\n", *(int*)($fp-0x300)
19 end
20
21 ###################################
22
23 set pagination off
24 set confirm off
25 file crackmips
26 target remote 127.0.0.1:4444 # gdbserver address
27
28 # break at the end of block 1
29 b *0x401440
30 commands
31 silent
32 printf "\nNew round\n"
33 print_counter
34 print_context_pc
35 print_param
36 print_addr
37 c
38 end
39
```

```
40 # break before the end of block 2
41 b *0x0401D8C
42 commands
43 silent
44 print_context_pc
45 c
46 end
47
48 c
```

To run that script within GDB, we first need to start `crackmips` with gdbserver in our `qemu` VM. After a few minutes, we get the following (cleaned) trace:

gdb_trace1.txt

```
1  New round
2  counter = 0
3  regs.pc = 0x0040228c
4  param = 0x00000000
5  addr = 0xcd0e9f0e
6  regs.pc = 0x00402290
7
8  New round
9  counter = 1
10 regs.pc = 0x004022bc
11 param = 0x00000000
12 addr = 0xcd0e99ae
13 regs.pc = 0x00402ce0
14
15 New round
16 counter = 2
17 regs.pc = 0x00402d0c
18 param = 0x00000000
19 addr = 0xcd0e420e
20 regs.pc = 0x00402da8
21
22 [...]
```

By reading the trace further, we realize that `param` is always equal to `counter/101`. This is actually the child's own loop counter, since its big loop is made of 101 pseudo basic blocks. We also notice that the `pc` sequence is different for each child's loop: round 0 is not equal to round 101, etc.

Getting a clean trace

Since we're only interested in the final `pc` value for each round, we can make a simpler script that just outputs those values. And organize them in a parsable format to be able to use them later in another script. Here is the version 2 of the script:

gdb_trace2_script.txt

```
1  def print_context_pc
2      printf "0x%08x\n", *(int*)($fp-0x1cc)
3  end
4
5  set pagination off
6  set confirm off
7  file crackmips
8  target remote 127.0.0.1:4444
9
10 # break before the end of block 2
```

```
11 b *0x0401D8C
12 commands
13 silent
14 print_context_pc
15 c
16 end
17
18 c
```

The cleaned trace only contains the 606 `pc` values, one on each line:

gdb_trace.txt

```
1 0x00402290
2 0x00402ce0
3 0x00402da8
4 0x00403550
5 [...]
6 0x004030e4
7 0x004039dc
```

Mission 1: accomplished!

Shortcut #2 : Symbolic execution using Miasm

We now have the list of each start address of each basic block executed by the child. The next step is to understand what each one of them does, and reorder them to reproduce the whole algorithm.

Even though writing a symbolic execution engine from scratch is certainly a fun and interesting exercise, I chose to play with Miasm. This excellent framework can disassemble binaries in various architectures (among which x86, x64, ARM, MIPS, etc.), and convert them into an intermediate language called IR (*intermediate representation*). It is then able to perform symbolic execution on this IR in order to find what are the side effects of a basic block on registers and memory locations. Although there is not so much documentation, Miasm contains various examples that should make the API easier to dig in. Don't tell me that it is hard to install, it is really not (well, I haven't tried on Windows ;). And there is even a docker image, so you have no excuse to not try it!

Miasm symbolic execution 101

Before scripting everything, let's first see how to use Miasm to perform symbolic execution of one basic block. For the sake of simplicity, let's work on the first basic block of the child's main loop.

miasm_example.py (1/5)

```
1 from miasm2.analysis.machine import Machine
2 from miasm2.analysis import binary
3
4 bi = binary.Container("crackmips")
5 machine = Machine('mips32l')
6 mn, dis_engine_cls, ira_cls = machine.mn, machine.dis_engine, machine.ira
```

First, we open the crackme using the generic `Container` class. It automatically detects the executable format and uses *Elfesteem* to parse it. Then we use the handy `Machine` class to get references to useful classes we'll use to disassemble and analyze the binary.

miasm_example.py (2/5)

```
1 BB_BEGIN = 0x00402290
2 BB_END = 0x004022BC
3
4 # Disassemble between BB_BEGIN and BB_END
5 dis_engine = dis_engine_cls(bs=bi.bs)
6 dis_engine.dont_dis = [BB_END]
7 bloc = dis_engine.dis_bloc(BB_BEGIN)
8 print '\n'.join(map(str, bloc.lines))
```

Here, we disassemble a single basic block, by explicitly telling Miasm its start and end address. The disassembler is created by instantiating the `dis_engine_cls` class. `bi.bs` represents the binary stream we are working on. I admit the `dont_dis` syntax is a bit weird; it is used to tell Miasm to stop disassembling when it reaches a given address. We do it here because the next instruction is a `break`, and Miasm does not normally think it is the end of a basic block. When you run those lines, you should get this output:

miasm_example.py output

```
1  LW          V1, 0x38(FP)
2  SLL         V0, V1, 0x2
3  ADDIU       A0, FP, 0x18
4  ADDU        V0, A0, V0
5  LW          A0, 0x8(V0)
6  LW          V0, 0x38(FP)
7  SUBU        A0, A0, V0
8  SLL         V0, V1, 0x2
9  ADDIU       V1, FP, 0x18
10 ADDU        V0, V1, V0
11 SW          A0, 0x8(V0)
```

Okay, so we know how to disassemble a block with Miasm. Let's now see how to convert it into the Intermediate Representation:

miasm_example.py (3/5)

```
1 # Transform to IR
2 ira = ira_cls()
3 irabloc = ira.add_bloc(bloc)[0]
4 print '\n'.join(map(lambda b: str(b[0]), irabloc.irs))
```

We instantiated the `ira_cls` class and called its `add_bloc` method. It takes a basic block as input and outputs a list of IR basic blocs; here we know that we'll get only one, so we use `[0]`. Let's see what is the output of those lines:

miasm_sample.py output

```
1  V1 = @32[(FP+0x38)]
2  V0 = (V1 << 0x2)
3  A0 = (FP+0x18)
4  V0 = (A0+V0)
5  A0 = @32[(V0+0x8)]
6  V0 = @32[(FP+0x38)]
7  A0 = (A0+(- V0))
8  V0 = (V1 << 0x2)
9  V1 = (FP+0x18)
10 V0 = (V1+V0)
```

```
11 @32[(V0+0x8)] = A0
12 IRDst = loc_00000000004022BC:0x004022bc
```

Each one of those lines are instructions in Miasm's IR language. It is pretty easy: each instruction is described as a list of side-effects it has on some variables, using expressions and affectations. `@32[...]` represents a 32-bit memory access; when it's on the left of an `=` sign, it's a *write* access, when it's on the right it's a *read*. The last line uses the pseudo-register `IRDst`, which is kind of the IR's `pc` register. It tells Miasm where is located the next basic block.

Great! Let's see now how to perform symbolic execution on this IR basic block.

miasm_example.py (4/5)

```
1  from miasm2.expression.expression import *
2  from miasm2.ir.symbexec import symbexec
3  from miasm2.expression.simplifications import expr_simp
4
5  # Prepare symbolic execution
6  symbols_init = {}
7  for i, r in enumerate(mn.regs.all_regs_ids):
8      symbols_init[r] = mn.regs.all_regs_ids_init[i]
9
10 # Perform symbolic exec
11 sb = symbexec(ira, symbols_init)
12 sb.emulbloc(irabloc)
13
14 mem, exprs = sb.symbols.symbols_mem.items()[0]
15 print "Memory changed at %s :" % mem
16 print "\tbefore:", exprs[0]
17 print "\tafter:", exprs[1]
```

The first lines are initializing the symbol pool used for symbolic execution. We then use the `symbexec` module to create an execution engine, and we give it our fresh IR basic block. The result of the execution is readable by browsing the attributes of `sb.symbols`. Here I am mainly interested on the memory side-effects, so I use `symbols_mem.items()` to list them. `symbols_mem` is actually a dict whose keys are the memory locations that changed during execution, and values are pairs containing both the previous value that was in that memory cell, and the new one. There's only one change, and here it is:

miasm_example.py output

```
1 Memory changed at (FP_init+(@32[(FP_init+0x38)] << 0x2)+0x20) :
2   before: @32[(FP_init+(@32[(FP_init+0x38)] << 0x2)+0x20)]
3   after: (@32[(FP_init+(@32[(FP_init+0x38)] << 0x2)+0x20)]+(- @32[(FP_init+0x
```

The expressions are getting a bit more complex, but still pretty readable. `FP_init` represents the value of the `fp` register at the beginning of execution. We can clearly see that a memory location as modified since a value was subtracted from it. But we can do better: we can give Miasm simplification rules in order to make this output much more readable. Let's do it!

miasm_example.py (5/5)

```
1  # Simplifications
2  fp_init = ExprId('FP_init', 32)
3  zero_init = ExprId('ZERO_init', 32)
4  e_i_pattern = expr_simp(ExprMem(fp_init + ExprInt32(0x38), 32))
```

```
5  e_i = ExprId('i', 32)
6  e_pass_i_pattern = expr_simp(ExprMem(fp_init + (e_i << ExprInt32(2)) + ExprI
7  e_pass_i = ExprId("pwd[i]", 32)
8
9  simplifications = {e_i_pattern      : e_i,
10                    e_pass_i_pattern : e_pass_i,
11                    zero_init        : ExprInt32(0) }
12
13 def my_simplify(expr):
14     expr2 = expr.replace_expr(simplifications)
15     return expr2
16
17 print "%s = %s" % (my_simplify(exprs[0]) ,my_simplify(exprs[1]))
```

Here we declare 3 replacement rules:

- Replace `@32[(FP_init+0x38)]` with `i`
- Replace `@32[(FP_init+(i << 0x2)+0x20)]` with `pwd[i]`
- Replace `ZERO_init` with 0 (although it is not really useful here)

There is actually a more generic way to do it using pattern matching rules with jokers, but we don't really need this machinery here. This the result we get after simplification:

miasm_example.py final output

```
1 pwd[i] = (pwd[i]+(- i))
```

That's all! So all this basic block does is a subtraction. What is nice is that the output is actually valid Python code :). This will be very useful in the last part.

Generating the child's algorithm

So in less than 60 lines, we were able to disassemble an arbitrary basic block, perform symbolic execution on it and get a pretty understandable result. We just need to apply this logic to the 100 remaining blocks, and we'll have a pythonic version of each one of them. Then, we simply reorder them using the GDB trace we got from the previous part, and we'll be able to generate 606 python lines describing the whole algorithm.

Here is an extract of the script automating all of this:

miasm_symbexec.py

```
1  def load_trace(filename):
2      return [int(x.strip(), 16) for x in open(filename).readlines()]
3
4  def boundaries_from_trace(trace):
5      bb_starts = sorted(set(trace))
6      boundaries = [(bb_starts[i], bb_starts[i+1]-4) for i in range(len(bb_sta
7      boundaries.append((0x4039DC, 0x04039E8)) # last basic bloc, added by han
8      return boundaries
9
10 def exprs2str(exprs):
11     return ' = '.join(str(e) for e in exprs)
12
13 trace = load_trace("gdb_trace.txt")
14 boundaries = boundaries_from_trace(trace)
15
16 print "# Building IR blocs & expressions for all basic blocks"
```

```
17 bb_exprs = []
18 for zone in boundaries:
19     bb_exprs.append(analyse_bb(*zone))
20
21 print "# Reconstructing the whole algorithm based on GDB trace"
22 bb_starts = [x[0] for x in boundaries]
23 for bb_ea in trace:
24     bb_index = bb_starts.index(bb_ea)
25     #print "%x : %s" % (bb_ea, exprs2str(bb_exprs[bb_index]))
26     print exprs2str(bb_exprs[bb_index])
```

The `analyse_bb()` function perform symbolic execution on a single basic block, given its start and end addresses. This is just wrapping what we've been doing so far into a function. The GDB trace is opened, parsed, and a list of basic block addresses is built from it (we cheat a little bit for the last one of the loop, by hardcoding it). Each basic block is analyzed and the resulting expressions are pushed into the `bb_exprs` list. Then the GDB trace is processed, by outputting the expressions corresponding to each basic block.

This is what we get:

output_algo.py

```
1  # Building IR blocs & expressions for all basic blocks
2  # Reconstructing the whole algorithm based on GDB trace
3  pwd[i] = (pwd[i]+(- i))
4  pwd[i] = ((0x0|pwd[i])^0xFFFFFFFF)
5  pwd[i] = (pwd[i]^i)
6  pwd[i] = (pwd[i]^i)
7  pwd[i] = (pwd[i]+0x3ECA6F23)
8  pwd[i] = (pwd[i]+0x6EDC032)
9  [...]
10 pwd[i] = ((pwd[i] << 0x14)|(pwd[i] >> 0xC))
11 pwd[i] = ((pwd[i] << ((i+0x1)&0x1F))|(pwd[i] >> ((((0x0|i)^0xFFFFFFFF)+0x20)&
12 i = (i+0x1)
```

Solving with Z3

Okay, so now we have a Python (and even C ;) file describing the operations performed on the 6 dwords containing the input key. We could try to bruteforce it, but using a constraint solver is much more elegant and faster. I also chose Z3 because it has nice Python bindings. And since its expression syntax is mostly compatible with Python, we just need to add a few things to our generated file!

sample_solver.py

```
1  from z3 import *
2  import struct
3
4  solution_str = "[ Synacktiv + NSC = <3 ]"
5  solutions = struct.unpack("<LLLLLL", solution_str)
6  N = len(solutions)
7
8  # Hook Z3's `>>` so it works with our algorithm
9  # (logical shift instead of arithmetic one)
10 BitVecRef.__rshift__  = LShR
11
12 pwd = [BitVec("pwd_%d" % i, 32) for i in range(N)]
13 pwd_orig = [pwd[i] for i in range(N)]
```

```
14 i = 0
15
16 # paste here all the generated algorithm from previous part
17 # BEGIN ALGO
18 pwd[i] = (pwd[i]+(- i))
19 pwd[i] = ((0x0|pwd[i])^0xFFFFFFFF)
20 # [...]
21 pwd[i] = ((pwd[i] << ((i+0x1)&0x1F))|(pwd[i] >> ((((0x0|i)^0xFFFFFFFF)+0x20)&
22 i = (i+0x1)
23 # END ALGO
24
25 s = Solver()
26
27 for i in range(N):
28     s.add(pwd[i] == solutions[i])
29
30 assert s.check() == sat
31
32 m = s.model()
33 sol_dw = [m[pwd_orig[i]].as_long() for i in range(N)]
34 key = ''.join(("%08x" % dw)[::-1].upper() for dw in sol_dw)
35
36 print "KEY = %s" % key
```

We've declared the valid solution, the list of 6 32-bit variables (`pwd`), pasted the algorithm, and ran the solver. We just need to be careful with the >> operation, since Z3 treats it as an arithmetic shift, and we want a logical one. So we replace it with a dirty hook.

The solution should come almost instantly:

```
1 $ python sample_solver.py
2 KEY = 322644EF941077AB1115AB575363AE87F58E6D9AFE5C62CC
```

Alternative solution – conclusion

I chose this solution not only to get acquainted with Miasm, but also because it required much less effort and pain :). It fits into approximately 20 lines of GDB script, and 120 of python using Miasm and Z3. You can find all of those in this folder. I hope it gave you an understandable example of symbolic execution and what you can do with it. However I strongly encourage you to dig into Miasm's code and examples if you want to really understand what's going on under the hood.

War's over, the final words

I guess this is where I thank both @elvanderb for this really cool challenge and @synacktiv for letting him write it :–). *Emilien* and I also hope you enjoyed the read, feel free to contact any of us if you have any remarks/questions/whatever.

Also, special thanks to @__x86 and @jonathansalwan for proofreading!

The codes/traces/tools developed in this post are all available on github here and here!

By the way, don't hesitate to contact a member of the staff if you have a cool post you would like to see here — you too can end up in doar-e's wall of fame :–).

Posted by Axel "0vercl0k" Souchet & Emilien "tr4nce" Girault Oct 11th, 2014 MIPS, NoSuchCon,

reverse-engineering, symbolic execution, z3, z3py

« Dissection of Quarkslab's 2014 security challenge Spotlight on an unprotected AES128 white-box implementation »

Recent Posts

- Debugger data model, Javascript & x64 exception handling
- Binary rewriting with syzygy, Pt. I
- happy unikernels
- Token capture via an llvm-based analysis pass
- Keygenning with KLEE