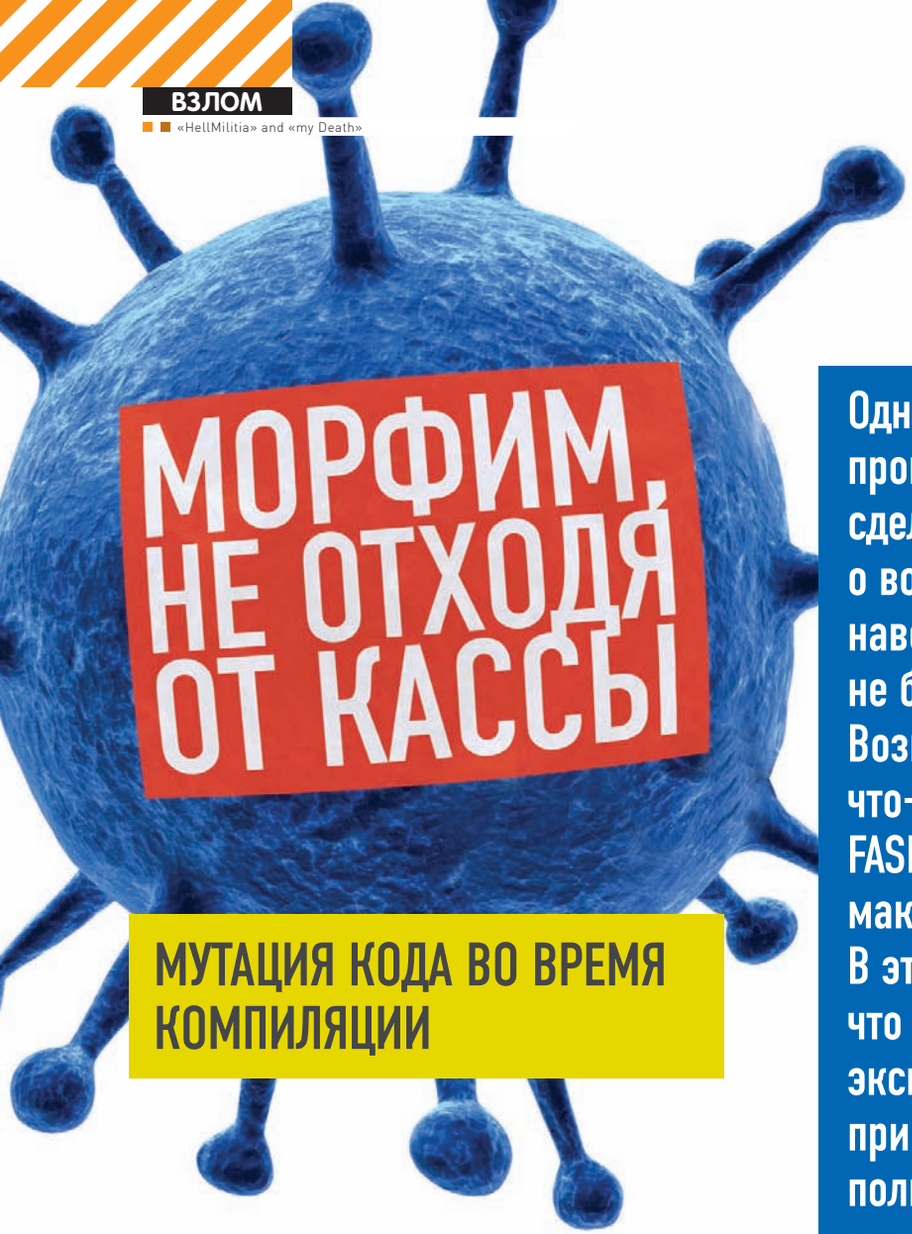


МУТАЦИЯ КОДА ВО ВРЕМЯ КОМПИЛЯЦИИ

Однажды, после написания программы, которую я хотел сделать платной, я задумался о вопросе ее защиты. Писать навесной протектор желания не было, да и времени тоже. Возможно ли сделать что-то средствами компилятора FASM, ведь у него мощнейший макроязык? В этой статье я решил описать, что вышло из моих экспериментов. Здесь на простых примерах будут описаны методы полиморфизма, пермутации, метаморфинга и оббфускации бинарного кода.

Итак, мне кажется, я добился отличных результатов, ведь у меня с блеском получилось реализовать: полиморфизм (генерация мусорного кода), метаморфинг (замена инструкций аналогами), пермутацию (случайное перемешивание блоков кода с сохранением функционала и логики работы), обфускацию (метод запутывания логики кода, противодействие анализу), контроль целостности кода (для защиты от изменения, patch'ей), шифрование кода и данных. Рандомизация кода служит для защиты от автоматических распаковщиков, анализаторов, патчей, обфускация — для запутывания исследователя; при достаточной обфускации анализ программы может затянуться на долгие месяцы... Хватит болтовни, приступим!

0XBA11EE PRNG

Первоначально следует написать генератор псевдослучайных чисел — сердце любого движка рандомизации кода. Генератор я взял простой, наподобие ANSI C, для моих целей его вполне хватало.

```
rndseed = 100500
macro randomize {
    randseed = randseed * 1103515245 + 12345
    randseed = (randseed / 65536) mod 0x100000000
    rndnum = randseed and 0xFFFFFFFF
}
```

Работает он исправно, но, так как инициализирующее значение постоянно, каждый раз, при каждой компиляции будет выдана одна и та же последовательность чисел. После недолгих раздумий и чтения официального форума, я нашел значение, которым можно завести генератор — это timestamp, UNIX-время. Получить его можно вот таким образом: `randseed = %t`. Генерировать случайное число, к примеру, в диапазоне 0 - 0xDEAD, теперь можно так:

```
randomize
random_number = rndnum mod 0xDEAD - 1
```

0XBADCODE ИЛИ ГЕНЕРАЦИЯ МУСОРА

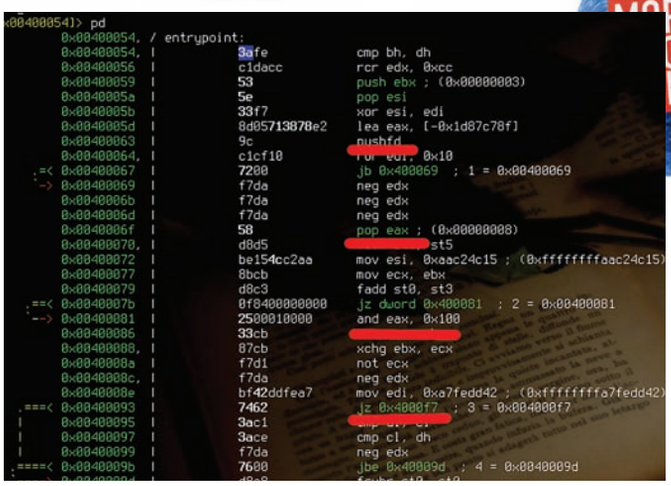
Для начала, попробуем написать макрос для генерации простой инструкции — int. Состоит int из двух байт — опкода 0xCD и номера прерывания, который и будет случаен. Получаем номер прерывания:

```
randomize
int_val = rndnum mod 0xFF
```

Далее пишем следующую незаурядную конструкцию:

```
db 0xCD
db num
```

Пока все просто. Оформив эти 4 строки в отдельный макрос gen_int и вызвав несколько раз, убеждаемся с помощью отладчика или дизассемблера, что код действительно случайный: `rept 7 { gen_int }`. И вот что получилось:



Затаившийся в бинарном мусоре антиотладочный трюк

```
cd78 | int 0x78
cda6 | int 0xa6
cdb4 | int 0xb4
cd36 | int 0x36
cdec | int 0xec
cd6a | int 0x6a
cd68 | int 0x68
```

Метод rept fasm'а выполняет код указанное количество раз. По-моему, начало более чем хорошее, нас ждет много интересного. Давай теперь рассмотрим генерацию инструкции lea; здесь я хочу осветить несколько аспектов. Сперва нужно завести константы, соответствующие регистрам:

```
REAX            = 0 ; AL
RECX            = 1 ; CL
REDX            = 2 ; DL
REBX            = 3 ; BL
RESP            = 4 ; AH
REBP            = 5 ; CH
RESI            = 6 ; DH
REDI            = 7 ; BH
```

Чтобы не нарушить работу кода, следует учитывать занятые регистры. Заведем переменные, хранящие их:

```
NOREG           = -1
USEDREG1        = NOREG
USEDREG2        = NOREG
RREG            = NOREG
```

Их может быть сколько угодно — зависит от логики работы программы, логики работы генератора и строения блока кода. Ниже представлен макрос, генерирующий случайный регистр, не учитывая занятые. Использовать будем только как источник.

```
macro rndreg {
  RREG = NOREG
  while (RREG = NOREG) | (RREG = RESP) | (RREG = REBP)
    randomize
    RREG = rndnum mod 8
  end while
}
```

В принципе, можно включить в варианты и Esp Ebp регистры, но мне захотелось так. Теперь макрос, генерирующий случайный незанятый регистр:

```
macro freereg {
  RREG = NOREG
  while (RREG = RESP) | (RREG = REBP) | (RREG = -1)
      | (RREG = USEDREG1) | (RREG = USEDREG2)
    randomize
    RREG = rndnum mod 8
  end while
}
```

Регистры Esp и Ebp не трогаем, дабы не сорвать стековый фрейм. Это первое, что я хотел осветить. Чтобы код был похож на произведенный нормальным компилятором (дабы не показывать сразу исследователю, что его водят за нос), следует немного ограничивать фантазию. Приведу пример на инструкции lea, которая, как известно, используется для получения\вычисления адреса. Принимающий регистр будет случайным, а как быть со вторым операндом? Возьмем значение в диапазоне Entry Point - (Entry Point + размер секции кода), ну или, для простоты, возьмем значение 0x1000. Для большего соответствия с нормальным кодом следует брать адреса из секции данных. Макрос, генерирующий инструкцию lea по правилам, описанным ранее:

```
macro gen_lea {
  freereg
  reg = (RREG * 8) + 5
  randomize
  address = (rndnum mod ((ENTRY_POINT + 0x1000 + 1)
      - ENTRY_POINT)) + ENTRY_POINT
  db 0x8D
  db reg
  dd address
}
```

Константу ENTRY_POINT объявляем заранее:

```
entry start
...
start:
ENTRY_POINT = $
```

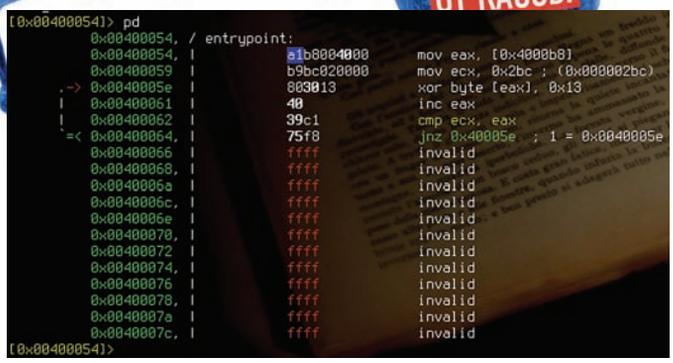
Или, что предпочтительнее: `ENTRY_POINT = $$`. Итог работы макроса, вызванного несколько раз:

```
8d3db10a4000 | lea edi, [0x400ab1]
8d154c044000 | lea edx, [0x40044c]
8d1d68054000 | lea ebx, [0x400568]
8d05e7024000 | lea eax, [0x4002e7]
8d15db0e4000 | lea edx, [0x400edb]
8d15670f4000 | lea edx, [0x400f67]
```

Как видишь, код случаен, и не бросается в глаза необычностью. Теперь не мешало бы объединить написанные макросы в один и построить код так, чтобы его было легко изменять или добавлять в него новые методы генерации инструкций, но для начала напишем еще один макрос для генерации FPU-инструкций:

```
macro gen_fpu {
    randomize
    type = rndnum mod 0x2F
    db 0xD8
    db 0xC0 + type
}
```

Проверим:



Код перед морфингом

```
d8d1 | fcom st0, st1
d8c9 | fmul st0, st1
d8d4 | fcom st0, st4
d8ed | fsubr st0, st5
d8d6 | fcom st0, st6
d8c2 | fadd st0, st2
```

Отлично! Теперь группируем, создаем макрос gen_trash, принимающий параметром количество генерируемых инструкций. Улучшить этот макрос можно, сделав параметром не количество инструкций, а максимальный размер в байтах. Еще лучшим ходом будет параметр, являющийся пределом случайному количеству инструкций/размеру в байтах. Реализуем первый, упрощенный, но немного уступающий другим вариант:

```
macro gen_trash length {
    repeat length
        randomize
        variant = randseed mod VARIANTS
        if variant = 0
           gen_lea
        else if variant = 1
            gen_fpu
        end if
    end repeat
}
```

Теперь для генерации 10 случайных инструкций указываем в коде: `gen_trash 10`. Следует расширить этот макрос, что не составит труда. Добавляй как можно больше инструкций\вариантов: ветвления; статистику повторения инструкций; порядок следования (куча FPU-инструкций вперемешку с обычным кодом — это подозрительно, ты не находишь? Или десяток инструкций lea, идущих подряд? А бесконтрольный генератор вполне может творить такое). Идей в процессе должно возникать великое множество — пробуй все, что придет в голову, не ограничивай себя. Теперь пара слов об использовании макроса gen_trash. Сделаем простой расшифровщик, разбавленный мусором:

```
    gen_trash 15
    mov eax, .CodeStart
    USEDREG1 = REAX
    gen_trash 27
    mov ecx, CodeSize
    USEDREG2 = RECX
    gen_trash 20
.again:
    xor byte[eax], XOR_KEY
    gen_trash 37
    inc eax
    gen_trash 10
    loop .again
    gen_trash 43
```

XOR_KEY, между прочим, тоже следует сделать случайным.

```
randomize
XOR_KEY = rndnum mod 0xFF
```

При большом количестве мусора и при достойном его качестве не так просто будет разобраться, что же в коде происходит, и как отделить его от мусора. Улучшить генератор можно, добавив работу с локальными\глобальными переменными, различные переходы, ветвления, процедуры, различные варианты инструкций, сложные инструкции вида lea eax,[ecx*4+100]... Но — главное!.. Самое главное — не забывай, что код должен быть схожим с генерируемым нормальным компилятором и одновременно хитрым, запутанным. Изучи частоту повторений инструкций в распространенных или входящих в состав операционной системы программ, а затем примени эту статистику в своем генераторе.

0XACED1A АНТИОТЛАДКА

Ни одна защита кода просто не представляется без антиотладочных трюков. Добавим и мы, но будем хитрее. Сделаем вставку случайного антиотладочного трюка в случайном месте, то есть просто добавим к макросу gen_trash, и трюк будет генерироваться наравне с инструкциями. Простой пример — если отладчик обнаружен, выполняется переход на случайный адрес в пределах секции кода.

```
macro adbg {
    randomize
    variant = rndnum mod N
    randomize
    destination = (rndnum mod ((ENTRY_POINT + 0x1000)
       - ENTRY_POINT)) + ENTRY_POINT
    if vatiant = 0
        invoke IsDebuggerPresent
        test eax,eax
        jnz $+destination
    else if variant = N
       .....
}
```

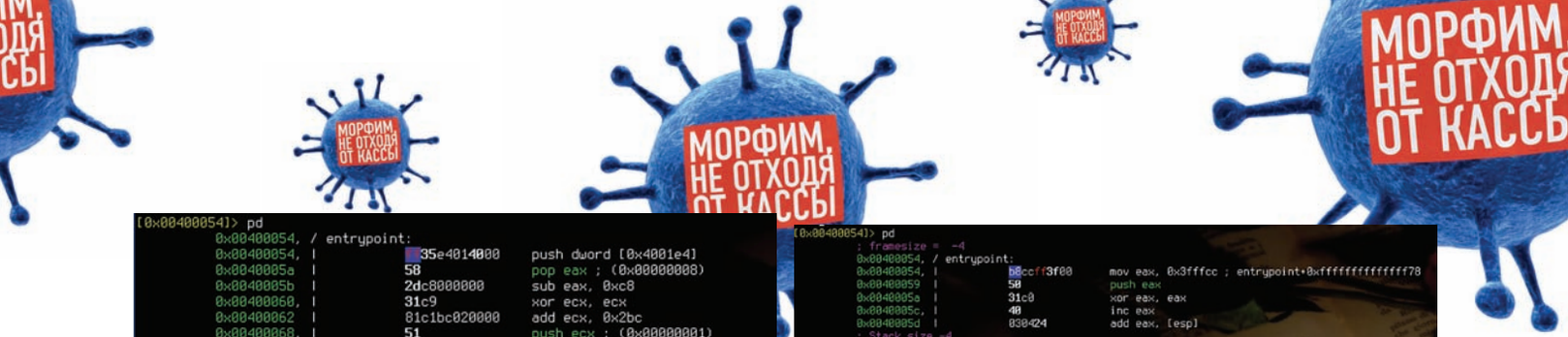
Также трюки следует разбавлять мусором. Добавляй больше антиотладки — больше сюрпризов исследователю.

0XACE ИЛИ РАНДОМИЗАЦИЯ API-ВЫЗОВОВ

Помимо бинарного мусора, код следует сделать высокоуровневым. Вполне послужит для этого Windows API. Функции могут не нести смысла, а могут быть и неотъемлемой частью программы. Простой пример вставки случайного API-вызова:

```
macro gen_trash_api {
  randomize
  RandomParam1 = rndnum mod 0xFFFFFFFF
  randomize
  RandomParam2 = rndnum mod 0xFFFFFFFF
  randomize
  variant = rndnum mod 4
  if variant = 0
    invoke IsBadReadPtr,RandomParam1,RandomParam2
  else if variant = 1
    invoke IsBadWritePtr,RandomParam1,RandomParam2
  else if variant = 2
    invoke IsBadCodePtr,RandomParam1
  else if variant = 3
    invoke GetLastError
  end if
}
```

Не стоит забывать, что API-функции не сохраняют регистры Eax, Ecx




```
[0x00400054]> pd
        0x00400054, / entrypoint:
        0x00400054, |      68e4014000    push dword [0x4001e4]
        0x0040005a  |      58            pop eax ; (0x00000008)
        0x0040005b  |      2dc8000000    sub eax, 0xc8
        0x00400060, |      31c9          xor ecx, ecx
        0x00400062  |      81c1bc020000  add ecx, 0x2bc
        0x00400068, |      51            push ecx ; (0x00000001)
        0x00400069  |      50            push eax
        0x0040006a  |      d1e0          shl eax, 1
        0x0040006c, |      09c8          or eax, ecx
        0x0040006e  |      58            pop eax ; (0x00000008)
        0x0040006f  |      59            pop ecx ; (0x00000009)
    .-> 0x00400070, |      31db          xor ebx, ebx
    |   0x00400072  |      83f313        xor ebx, 0x13
    |   0x00400075  |      3018          xor [eax], bl
    |   0x00400077  |      50            push eax
    |   0x00400078, |      35dec0adde    xor eax, 0xdeadc0de
    |   0x0040007d  |      80042401      add byte [esp], 0x1
    |   0x00400081  |      58            pop eax ; (0x00000008)
    |   0x00400082  |      51            push ecx ; (0x00000001)
    |   0x00400083  |      50            push eax
    |   0x00400084, |      29c1          sub ecx, eax
    `=< 0x00400086  |      75e8          jnz 0x400070  ; 1 = 0x00400070
```

Код после морфинга

```
[0x00400054]> pd
        ; framesize = -4
        0x00400054, / entrypoint:
        0x00400054, |      b8ccff3f00    mov eax, 0x3fffcc ; entrypoint+0xffffffffffffff78
        0x00400059  |      50            push eax
        0x0040005a  |      31c0          xor eax, eax
        0x0040005c, |      40            inc eax
        0x0040005d  |      030424        add eax, [esp]
        ; Stack size -4
        0x00400060, |      83c404        add esp, 0x4
```

Обфусцированный mov

и Edx. Сохраняй значения этих регистров, если в них содержатся и используются важные значения. Вставим вызов этого макроса в gen_trash. Подключи фантазию; вызовы функций не обязательно должны быть одиночными, высокоуровневый мусор должен взаимодействовать с бинарным — не подкопаешься. Неплохо будет эмулировать некоторые функции, то есть реализовать их код у себя. Вызов или использование своего кода являются вариантами, пример:

```
macro GetLastError {
  rnd
  variant = rndnum mod 2
  if variant = 0
    mov eax,[fs:18h]
    mov eax,[eax+TEB.LastError]
  else if variant = 1
    invoke GetLastError
  end if
}
```

0xA11A5, или метаморфинг

Метаморфинг я реализовал как замену инструкций своими функциональными аналогами. FASM позволяет переопределять инструкции макросами, что очень удобно. Возьмем, к примеру, инструкцию mov reg32_1, reg32_2. Какие могут быть аналоги? Первое, что приходит в голову (вообще их можно придумать великое множество):

```
push reg32_2
pop reg32_1
push reg32_2
mov reg32_1,[esp]
add esp,4
push reg32_2
xchg reg32_1,reg32_2
pop reg32_1
```

Примени фантазию, не следуй шаблонам, и за небольшой промежуток времени можно будет написать достаточное количество аналогов для всех инструкций. Напишем макрос, переопределяющий инструкцию mov. Обязательно проверяем, что аргументы являются регистрами, так как у нас есть замена только этого варианта:

```
macro mov arg1,arg2 {
    if (arg1 eqtype eax) & (arg2 eqtype eax)
        rnd
        variant = rndnum mod 4
        if variant = 0
            push arg2
            pop arg1
        else if variant = 1
            push arg2
            mov arg1,[esp]
            add esp,4
        else if variant = 2
            push arg2
            xchg arg1,arg2
            pop arg2
        else if variant = 3
            mov arg1,arg2
        end if
    else
        mov arg1,arg2
    end if
}
```

Проверяем:

```
mov eax,ecx
mov ecx,ecx
mov edx,esp
```

Итог:

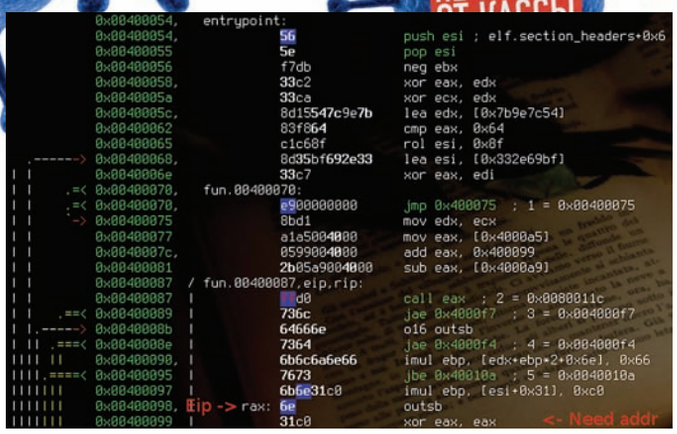
```
51   | push ecx
91   | xchg ecx, eax
59   | pop ecx
89e5 | mov ebp, esp
53   | push ebx
59   | pop ecx
```

Замечательно, не правда ли? Добавив как можно больше инструкций и вариантов замены, можно добиться замечательных результатов.

0xAB1E, или перестановка

Здесь все тоже предельно просто и дает мощный результат. Нам нужно изменить расположение некоторых блоков кода без изменения функциональности и без повреждения кода. Для начала за блоки возьмем процедуры, далее эти блоки следует максимально уменьшить. Над способом случайного изменения блоков кода я недолго думал, возможно, есть более изящное решение — подумай. Суть такова: каждую процедуру оборачиваем в макрос, создаем для нее переменную — флаг, сигнализирующий об использовании, дабы не вставлять процедуры несколько раз. Например (пермутируем три процедуры, скелет), код главной структуры теперича выглядит так:

```
fproc_1 = 0
fproc_2 = 0
...
entry $
;код главной процедуры
...
while (flag_1 = 0) | (flag_2 = 0)
    randomize
    sequence = rndnum mod 2
    if sequence = 0
        if flag_1 = 0
            proc_1
```



После патча контрольная сумма отличалась всего лишь на один байт, и теперь переход выполнен не туда, куда надо

```
            flag_1 = 1
        end if
    else if sequence = 1
        if flag_2 = 0
            proc_2
            flag_2 = 1
        end if
    end if
end while
macro proc_1 {
    proc AnyProcedure1
        ...
        ret
    endp
}
macro proc_2 {
    proc AnyProcedure2
        ....
        ret
    endp
}
```

Проверив этот код, убеждаемся, что процедуры выставляются как надо, случайно, код не портится.

0XDEFACED, ИЛИ ОБФУСКАЦИЯ: ДИНАМИЧЕСКОЕ ВЫЧИСЛЕНИЕ АДРЕСОВ

Один из способов противодействия дизассемблерам и обману анализаторов — динамическое вычисление адресов переходов или адресов переменных. Пример, как можно вычислять адрес:

```
push label - value
add [esp],value
jmp [esp]
....
label:
add esp,4; избавляемся от ненужного
```

Следует сделать случайными алгоритмы вычисления, варианты реализации алгоритма, и, естественно, значения для модификации. Примерами этого станут представленные ниже макросы o_jmp и olabel:

```
macro o_jmp destination {
    randomize
    variant = rndnum mod 2
    if variant = 0
        randomize
        value = rndnum mod IMAGE_BASE
        push destination - value
        add [esp],value
        jmp [esp]
    else if variant = 1
        randomize
        value = rndnum mod (0xFFFFFFFF - IMAGE_BASE
          - 0x1000)
        push destination + value
        sub [esp],value
        jmp [esp]
    end if
}
macro o_label name {
    label name
    add esp,4
}
```

Итог работы макросов:

```
68001127b6      | push dword 0xb6271100
812c249b10e7b5  | sub dword [esp], 0xb5e7109b
ff2424          | jmp dword near [esp]
31c0            | xor eax, eax
83c404          | add esp, 0x4
31c0            | xor eax, eax
```

Без трассировки и не узнаешь, куда ведет переход, следовательно, статический анализ обламывается. Здесь также стоит учитывать занятые\свободные регистры в генераторе мусора, так как постоянное использование Esp ставит клеймо на способе, да и само по себе накладно. Еще одной неплохой уловкой является вставка переходов на данные, но переходы эти никогда не выполняются (или выполняются только при наличии отладчика). Это сбивает с толку анализаторы, и они пытаются дизассемблировать данные. Пример макроса:

```
macro facke_code_ref data_addr,jmp_addr {
    xor eax,eax
    inc eax
    jnz jmp_addr
    call data_addr
    ;trash
}
```

В итоге адрес data_addr будет анализироваться как код.

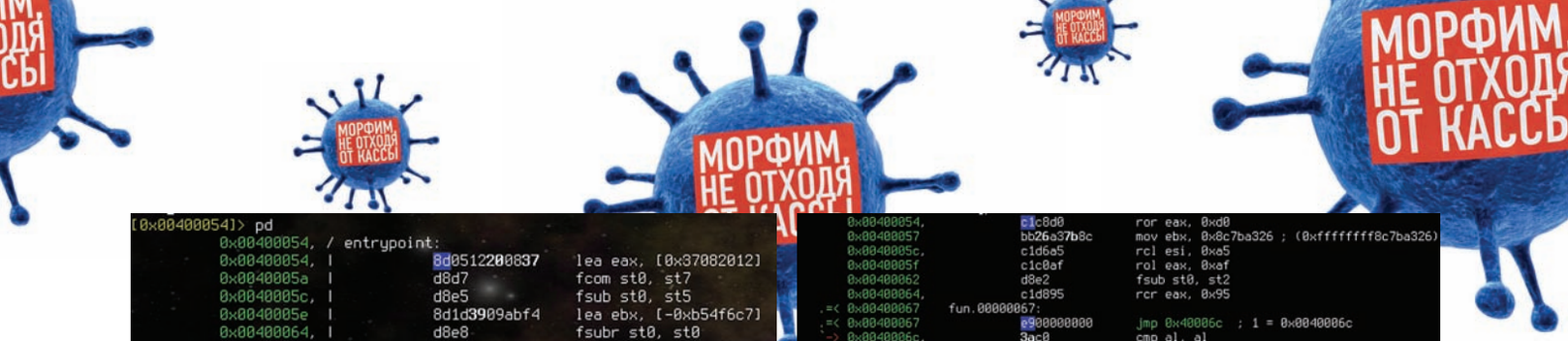
0XA55 — ЗАШИФРОВКА КОДА\ДАННЫХ

Замечательными функциями макроязыка FASM, отличающими его от других макроассемблеров, являются load и store. Использовать их можно для шифрования кода или данных. Простой пример, для шифрования используется xor:

```
macro xor_data start,length,key {
    repeat length
        load x from start+%-1
        x = x xor key
        store x at start+%-1
    end repeat
}
```

Очень полезный макрос, я его использовал для зашифровки строковых данных. Пример использования:

```
randomize
XOR_KEY = rndnum mod 0xFF
xor_data strings, strings_size, XOR_KEY
strings:
```



Взгляд на многообещающее будущее через окно radare

```
    any_string db 'Mate.Feed.Kill.Repeat'
strings_size = $ - strings
```

0xABA51A, или Контроль целостности кода

Вычислив на стадии компиляции контрольные суммы участков кода, можно защититься от модификации, пересчитывая и проверяя при выполнении эти суммы. Также подобным образом можно детектировать трассировку посредством вставки в код прерывания int3, как делают многие отладчики. Макрос, вычисляющий crc32 сумму блока кода:

```
CRC32_SUM = 0
macro calc_crc32 start, size {
  local b,c
  c = 0xffffffff
  repeat size
    load b byte from start+%-1
    c = c xor b
    repeat 8
    c = (c shr 1) xor (0xedb88320 * (c and 1))
  end repeat
  CRC32_SUM = c xor 0xffffffff
}
```

Хочу заметить, что операции вида `if(original_hash != current_hash) Error()` абсолютно бесполезны! Хотя используются повсеместно, даже в крутых протекторах. А вот нечто подобное:

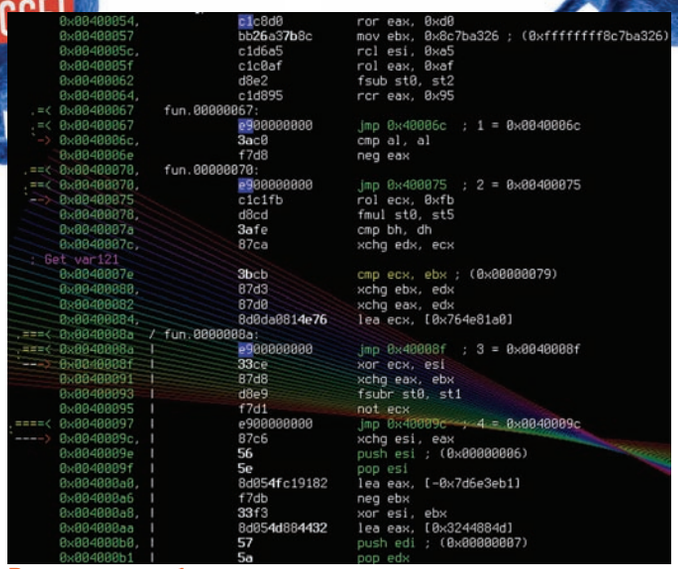
```
mov eax,address + original_hash
sub eax,current_hash
call eax
```

Совсем другое дело. Двух зайцев сразу: обфускация — динамическое вычисление адреса перехода, и контроль целостности кода, то есть, если код был каким-либо образом изменен, будет выполнен переход кот знает куда.

0xACCEDE, или Обман анализаторов

Анализаторы исполняемых файлов вроде PEiD используют сигнатурный поиск, в базе находятся цепочки байт, которые встречаются в популярных протекторах\упаковщиках. Для того, чтобы сбить с толку взломщиков своей программы, я создал макрос, добавляющий в Entry Point программы случайную сигнатуру. Воспользовавшись вышеупомянутым анализатором или его аналогом и получив ложный результат, взломщик попытается распаковать программу либо автоматическим распаковщиком, либо вручную, следуя описанию. И, конечно же, ничего не получится, кроме тяжелого ступора.

```
macro facke_sign {
    randomize
    variant = rndnum mod N
    if vatiant = 0
        ;PE Protect 0.9 -> Christoph Gabler
        push edx
        push ecx
        push ebp
        push edi
        db 0x64, 0x67, 0xA1, 0x30, 0x00
    ;FASM генерирует длинный формат инструкции
```



Результат работы немного расширенного макроса

```
    ;mov eax,[fs:0x30], поэтому записал таким образом
        test eax,eax
        js @f+1
        call .end.sign
        pop eax
        add eax,7
        db 0xC6
        nop
        ret
    @@:
        db 0xE9,0x00,0x00,0x00,0x00
    .end.sign:
    else if variant = 1
        ;CD-Cops II -> Link Data Security
        push ebx
        pushad
        mov ebp,0x90909090
        lea eax,[ebp-0x70]
        lea ebx,[ebp-0x70]
        call $+5
        lea eax,[ecx]
        db 0xE9,0x00,0x00,0x00,0x00
        ...
    else if variant = N
        ...
}
```

0xAD105. Заключение

Грамотное использование и комбинирование описанных мною техник позволяет сделать серьезную защиту. Это и очень удобно: написав, отладив программу, с минимальными правками исходного кода превращаем ее в неприступный бастион. После того, как я написал свой набор макросов, протестировал и применил их к своей программе, мне пришла в голову еще одна замечательная идея. Данный метод я еще и автоматизировал следующим способом: поместил на сервер исходный код программы и компилятор FASM, при запросе пользователем trial-версии программы она автоматически компилируется; таким образом получается, что каждому пользователю выдается уникальная версия программы. Универсальные взломщики (патчеры, crack'и и т.п.) просто бессильны — придется ломать каждую копию отдельно. А это ведь непросто, учитывая, что весь код изменен, а не как у навесных протекторов, только «сверху». Мне, как разработчику, остается только чаще обновлять исходники и совершенно не волноваться о том, что мою программу могут взломать. Так что, open your eyes, open your mind!