ВЗЛОМ

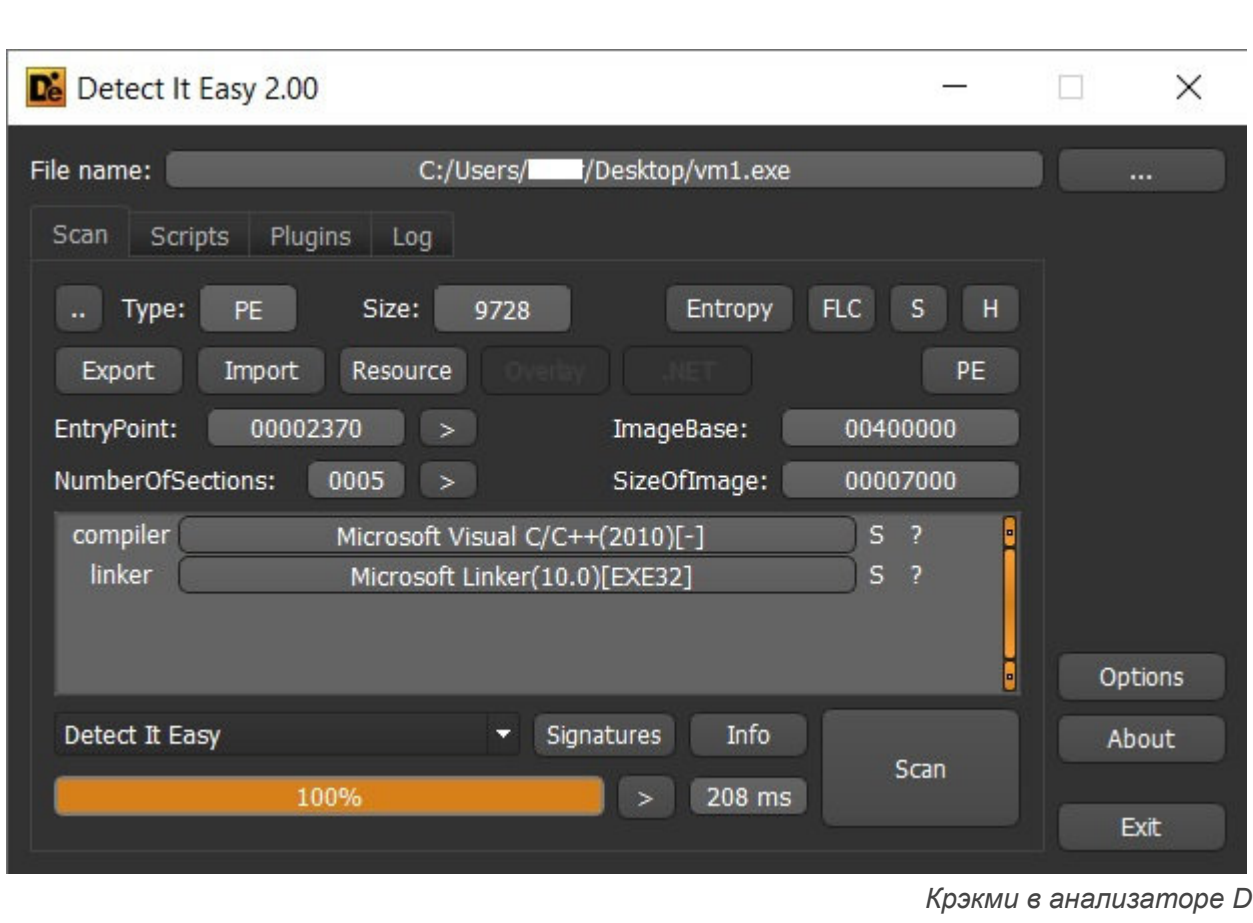
Ghidra vs crackme. Обкатываем конкурента IDA Pro на примере решения хитрой крэкми с VM

Nik Zerof, 21.03.2019 4 комментария 34,780 Добавить в закладки

Чтобы испытать новое средство для реверса, созданное в стенах АНБ США, я решил поломать замечательную и несложную крэкми MalwareTech. Выбрал ее неслучайно. В одной из своих статей я рассказывал о том, как устроена виртуализация кода, и мы даже написали простенькую виртуалку. А теперь давай посмотрим, как ломать такую защиту.

Скачать крэкми можно с сайта MalwareTech, пароль к архиву — тоже `MalwareTech`.

Итак, для начала посмотрим, что в архиве. Видим исполняемый файл `vm1.exe` и файл дампа `ram.bin`. В пояснении на сайте написано, что мы имеем дело с восьмибитной виртуальной машиной. Файл дампа — не что иное, как кусок памяти, в котором вперемешку расположены рандомные данные и флаг, который мы должны найти. Оставим пока в покое файл дампа и посмотрим на `vm1.exe` через программу DiE.



Крэкми в анализаторе Detect It Easy

DiE не показывает ничего интересного, с энтропией все в порядке. Значит, никакой навесной защиты нет, но проверить все равно стоило. Давай загрузим этот файл в Ghidra и посмотрим, что она выдаст. Я приведу полный листинг приложения без функций (он совсем небольшой) — чтобы ты понял, с чем мы имеем дело.

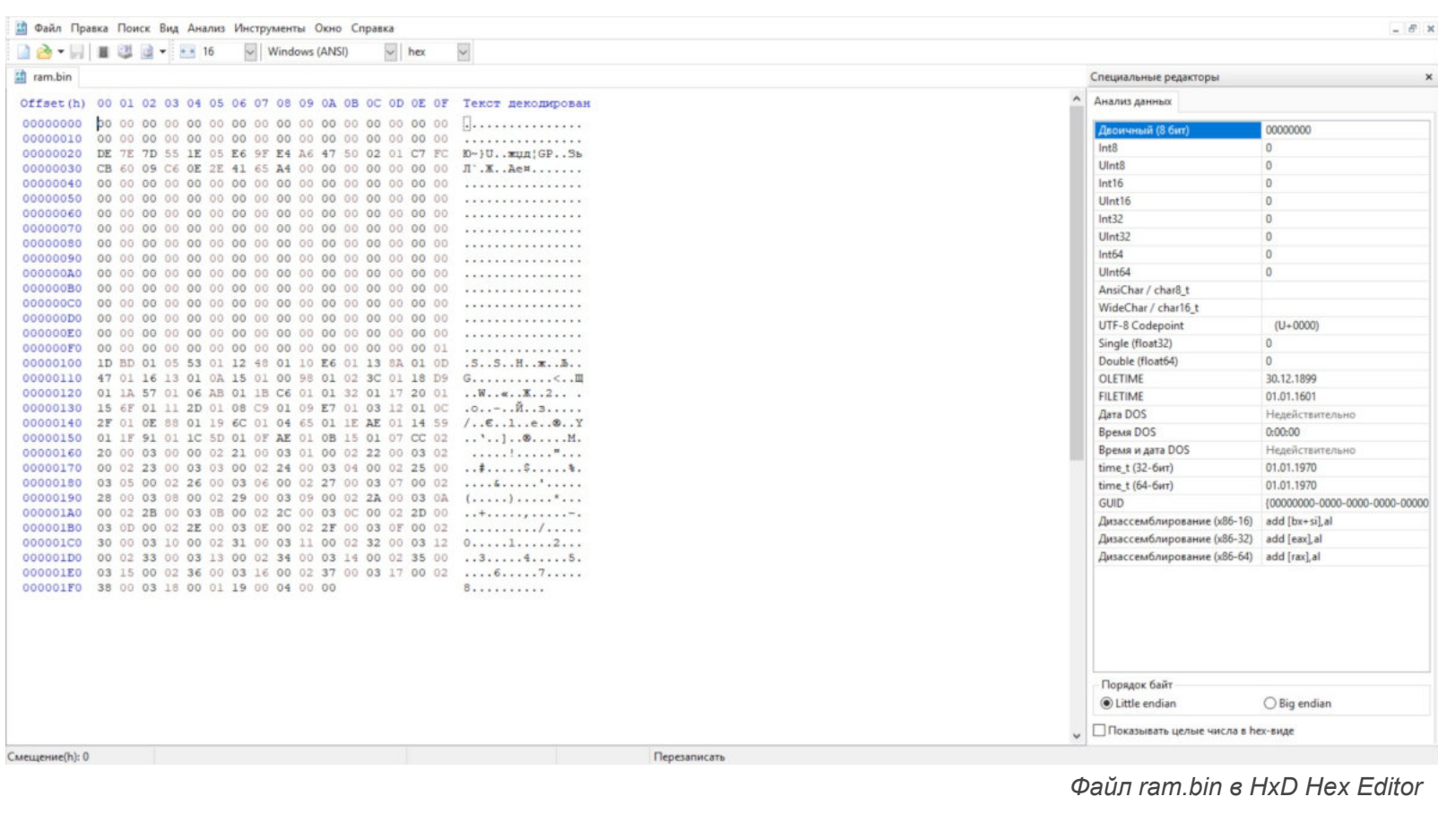
```
PUSH   EBP
MOV    EBP ,ESP
SUB    ESP ,0x94
LEA    ECX =>local_94 ,[0xffffff70  + EBP ]
CALL   MD5::MD5
PUSH   0x1fb
PUSH   0x0
CALL   dword ptr [->KERNEL32.DLL::GetProcessHeap ]
PUSH   EAX
CALL   dword ptr [->KERNEL32.DLL::HeapAlloc ]
MOV    [DAT_0040423c ],EAX
PUSH   0x1fb
PUSH   DAT_00404040
MOV    EAX ,[DAT_0040423c ]
PUSH   EAX
CALL   memcpy
ADD    ESP ,0xc
CALL   FUN_004022e0
MOV    ECX ,dword ptr [DAT_0040423c ]
PUSH   ECX
LEA    ECX =>local_94 ,[0xffffff70  + EBP ]
CALL   MD5::digestString
MOV    dword ptr [local_98  + EBP ],EAX
PUSH   0x30
PUSH   s_We've_been_compromised!_0040302c
MOV    EDX ,dword ptr [local_98  + EBP ]
PUSH   EDX
PUSH   0x0
CALL   dword ptr [->USER32.DLL::MessageBoxA ]
PUSH   0x0
CALL   dword ptr [->KERNEL32.DLL::ExitProcess ]
XOR    EAX ,EAX
MOV    ESP ,EBP
POP    EBP
RET
```

Как видишь, код простой и легко читается. Давай воспользуемся декомпилятором Ghidra и посмотрим, что он выдаст.

```
undefined4 entry(void)
{
HANDLE hHeap;
char *lpText;
DWORD dwFlags;
SIZE_T dwBytes;
MD5 local_94 [144];

MD5(local_94);

dwBytes = 0x1fb;
dwFlags = 0;

hHeap = GetProcessHeap();
DAT_0040423c = (char *)HeapAlloc(hHeap,dwFlags,dwBytes);
memcpy(DAT_0040423c,&DAT_00404040,0x1fb);

FUN_004022e0();

lpText = digestString(local_94,DAT_0040423c);
MessageBoxA((HWND)0x0,lpText,"We\'ve been compromised!",0x30);

ExitProcess(0);
return 0;
}
```

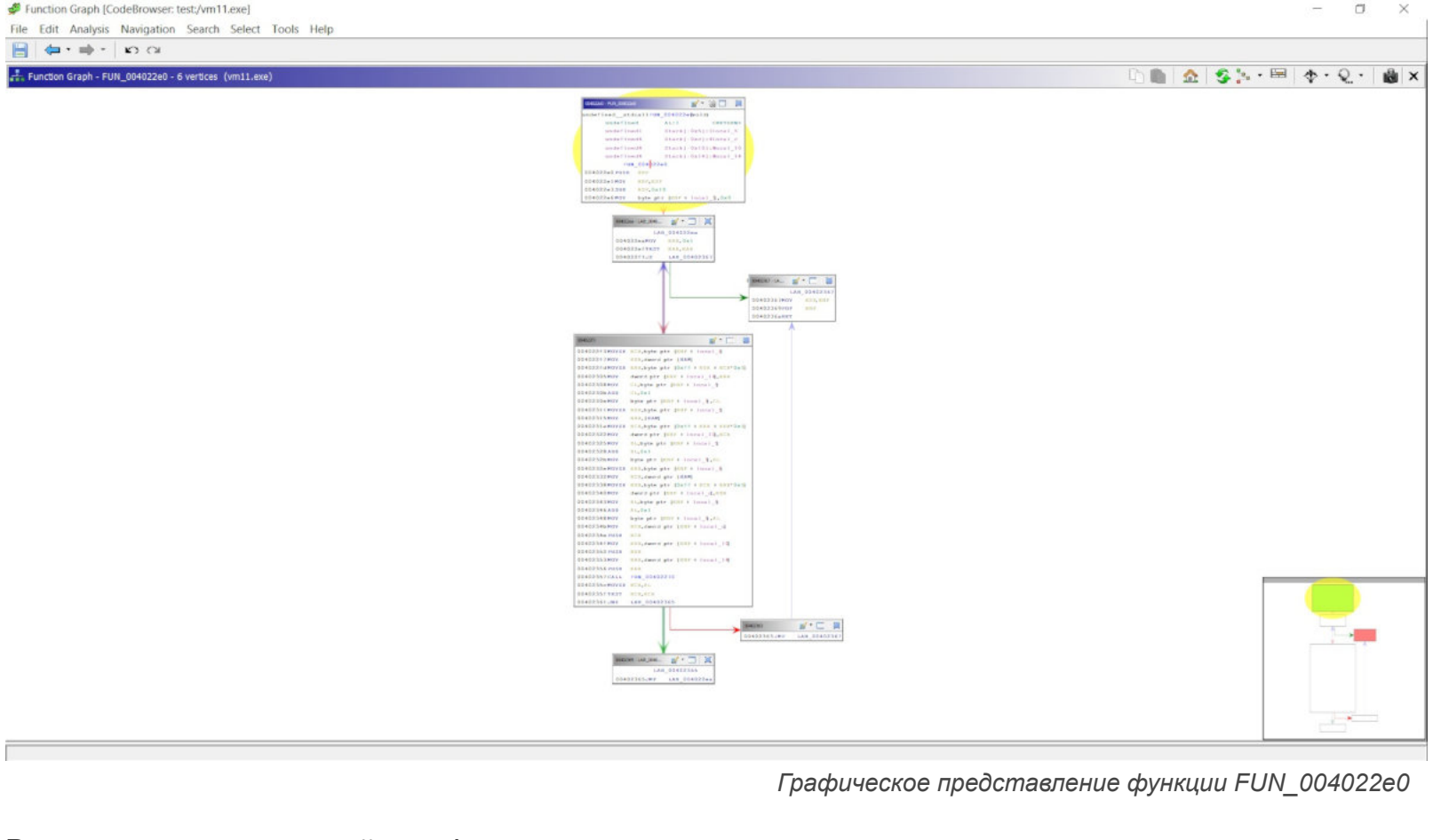
Я добавил отступы для удобочитаемости — отделил объявления переменных от остального кода. Код весьма простой: сначала выделяется память в куче (`GetProcessHeap` ... `HeapAlloc`), далее в нее копируется 0x1fb(507) байт из `DAT_00404040`. Но у нас нет ничего интересного в 00404040! Вспоминаем, что в инструкции к крэкми говорилось, что ram.bin — это кусок памяти. Разумеется, если посмотреть размер файла, он оказывается равным 507 байт.

Загружаем `ram.bin` в HxD или любой другой шестнадцатеричный редактор и смотрим.



Файл ram.bin в HxD Hex Editor

Увы, ничего внятного там не обнаруживаем. Но логика работы немного проясняется: `DAT_0040423c` — это ram.bin (наши выделенные 507 байт в куче). Давай переименуем `DAT_0040423c` в RAM, чтобы было удобнее ориентироваться в коде. Далее заходим в функцию `FUN_004022e0`.

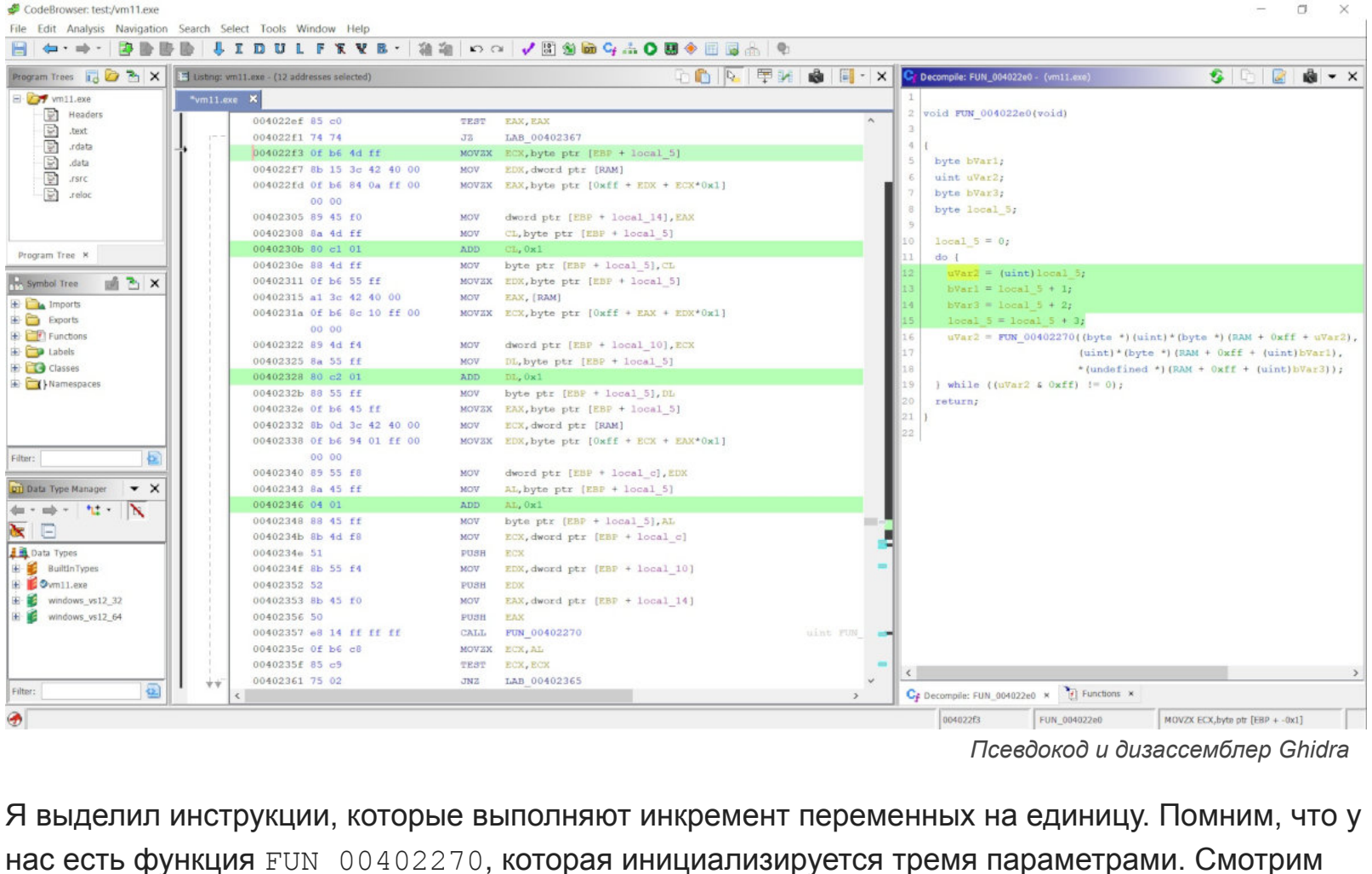


Графическое представление функции FUN_004022e0

Вот декомпилированный код функции:

```
void FUN_004022e0(void)
{
byte bVar1;
uint uVar2;
byte bVar3;
byte local_5;

local_5 = 0;
do {
uVar2 = (uint)local_5;
bVar1 = local_5 + 1;
bVar3 = local_5 + 2;
local_5 = local_5 + 3;
uVar2 = FUN_00402270((byte *)(uint)*(byte *)(RAM + 0xff + uVar2),
                     (uint)*(byte *)(RAM + 0xff + (uint)bVar1),
                     *(undefined *)(RAM + 0xff + (uint)bVar3));
} while ((uVar2 & 0xff) != 0);
return;
}
```

Поскольку мы все-таки знаем, что перед нами виртуальная машина, все становится более-менее понятно. Но чтобы действительно понять псевдокод, всегда нужно смотреть в дизассемблер, иначе псевдокод может запутать.



Псевдокод и дизассемблер Ghidra

Я выделил инструкции, которые выполняют инкремент переменных на единицу. Помним, что у нас есть функция `FUN_00402270`, которая инициализируется тремя параметрами. Смотрим инициализацию первого параметра.

```
MOVZX  ECX ,byte ptr [EBP  + local_5 ]
MOV    EDX ,dword ptr [RAM ]
MOVZX  EAX ,byte ptr [0xff  + EDX  + ECX *0x1 ]
MOV    dword ptr [EBP  + local_14 ],EAX
MOV    CL,byte ptr [EBP  + local_5 ]

ADD    CL,0x1    ; Инкремент переменной
```

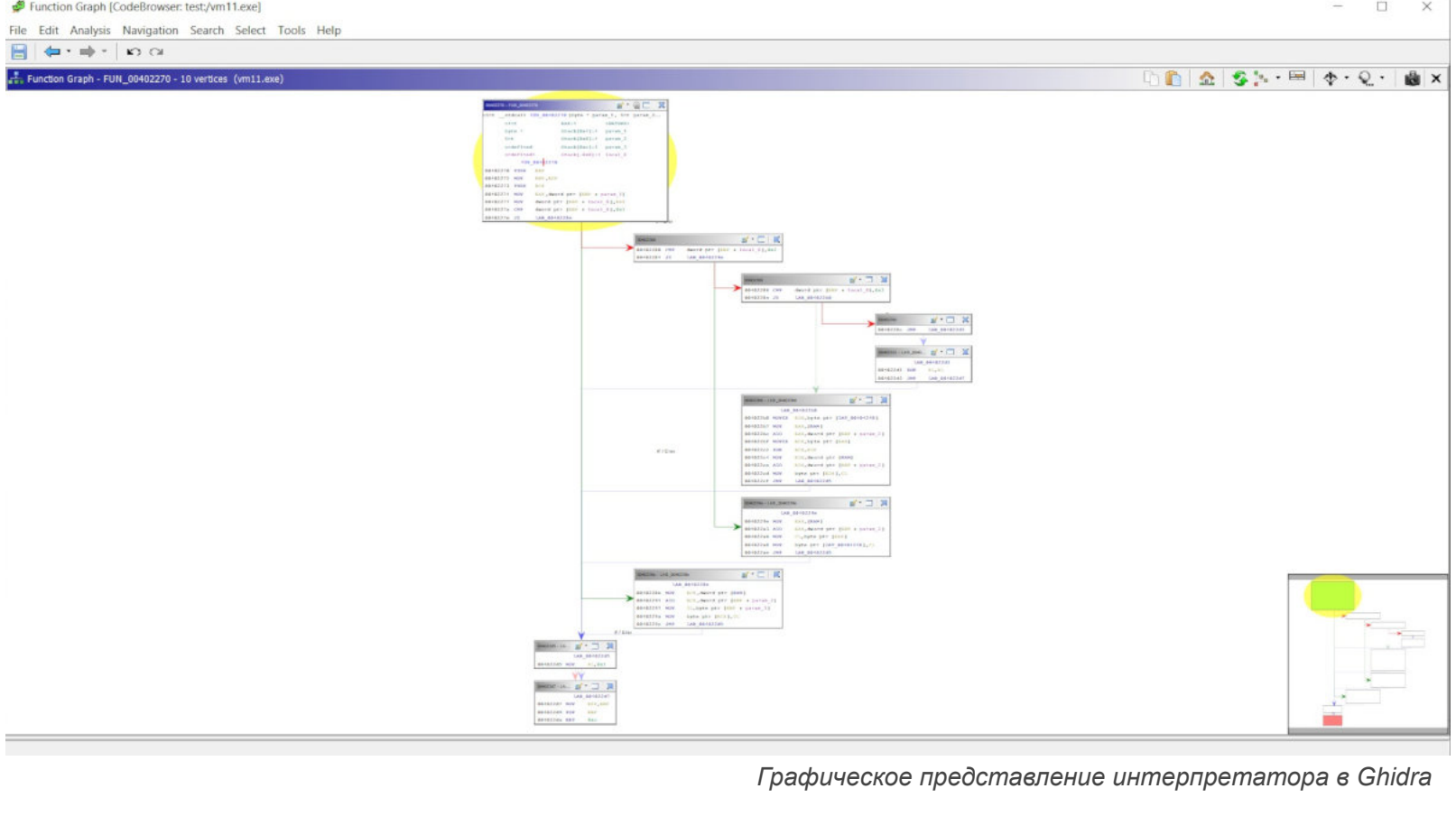
Очевидно, что берется байт из `[RAM]` и им инициализируется переменная. И такой же код при инициализации каждого аргумента функции, единственное отличие — меняются регистры, в которых будут аргументы функции `FUN_00402270`. В итоге вызов функции выглядит таким образом:

```
MOV    ECX ,dword ptr [EBP  + local_c ]
PUSH   ECX
MOV    EDX ,dword ptr [EBP  + local_10 ]
PUSH   EDX
MOV    EAX ,dword ptr [EBP  + local_14 ]
PUSH   EAX
CALL   FUN_00402270
```

Итак, в `FUN_00402270` передаются три параметра — три байта из `[RAM]`, следующие друг за другом. Заходим в функцию `FUN_00402270`, вот ее псевдокод:

```
uint FUN_00402270(byte *param_1,int param_2,undefined param_3)
{
if (param_1 == (byte *)0x1) {
*(undefined *)(RAM + param_2) = param_3;
}
else {
if (param_1 == (byte *)0x2) {
  param_1 = (byte *)(RAM + param_2);
  DAT_00404240 = *param_1;
}
else {
  if (param_1 != (byte *)0x3) {
    return (uint)param_1 & 0xffffff00;
  }
  param_1 = (byte *)(RAM + param_2);
  *(byte *)(RAM + param_2) = *param_1 ^ DAT_00404240;
}
}
return CONCAT31((int3)((uint)param_1 >> 8),1);
}
```

Здесь проверяется первый переданный в функцию байт, и, если он совпадает с `0x1`, `0x2` или `0x3`, обрабатываются следующие два аргумента. Парсинг первого параметра особенно явно читается в дизассемблерном листинге. По всей видимости, это интерпретатор команд виртуальной машины, который содержит всего три команды ВМ.



Графическое представление интерпретатора в Ghidra

```
PUSH   EBP
MOV    EBP ,ESP
PUSH   ECX
MOV    EAX ,dword ptr [EBP  + param_1 ]
MOV    dword ptr [EBP  + local_8 ],EAX
CMP    dword ptr [EBP  + local_8 ],0x1
JZ     LAB_0040228e
CMP    dword ptr [EBP  + local_8 ],0x2
JZ     LAB_0040229e
CMP    dword ptr [EBP  + local_8 ],0x3
JZ     LAB_004022b0
JMP    LAB_004022d1
```

На этом этапе я остановлюсь немного подробнее, чтобы подвести промежуточный итог. Итак, мы имеем приложение, работающее с 507 байт памяти, дамп которых у нас есть — это ram.bin. Внутри этого дампа данные, интересные нам, перемешаны с другими, ненужными нам данными. Приложение `vm1.exe` читает побайтово память в поисках инструкций `0x1`, `0x2` и `0x3`, и, как только одна из них находится, обрабатываются следующие два байта после них.

Другими словами, мы имеем мнемонические команды (p-code, пи-код), которые работают со своими двумя аргументами, а область памяти в 507 байт — не что иное, как лента пи-кода, перемешанная с мусором. На самом деле не стоит пугаться мусора — обработка команд начнется с нахождения нужного байта опкода, и будут взяты следующие два значения, а мусор попросту пропущен.

INFO

P-code, или «пи-код», — реализация мнемоник для собственного интерпретатора команд. Его еще называют кодом «гипотетического процессора» — ведь, по сути, процессор для исполнения пи-кода написан кем-то самостоятельно.

Теперь давай разберем запрограммированные опкоды команд, парсинг которых выполняет код, показанный выше. Я буду сразу приводить код на языке C, аналогичный дизассемблерному листингу.

```
LAB_0040228e:
MOV    ECX ,dword ptr [RAM ]
ADD    ECX ,dword ptr [EBP  + param_2 ]
MOV    DL,byte ptr [EBP  + param_3 ]
MOV    byte ptr [ECX ],DL
JMP    LAB_004022d5
```

Начнем восстанавливать логику работы виртуальной машины. Объявим `char ram[507]` — это будет память виртуальной машины. В этот массив при помощи функций `fopen` → `fread` → `fwrite` запишем содержимое файла ram.bin. Четыре строчки ассемблерного кода и переход — все просто: в массив `ram` по значению `[EBP + param_2]` перемещаем значение `param_3`. В коде это будет выглядеть таким образом:

```
ram[val_01] = val_02;
```

Начинаем анализировать следующую подпрограмму:

```
LAB_0040229e:
MOV    EAX ,[RAM ]
ADD    EAX ,dword ptr [EBP  + param_2 ]
MOV    CL,byte ptr [EAX ]
MOV    byte ptr [r1 ],CL    ; DAT_00404240
JMP    LAB_004022d5
```

Она очень похожа на предыдущую, это тоже аналог операции MOV, но здесь уже используется один из двух регистров виртуальной машины (`DAT_00404240` в листинге), в который кладется значение из памяти ВМ. А с нашей точки зрения — из массива `ram`, который адресован `param_2` в дизассемблерном коде, а в нашем — `val_01`. Другими словами, операция `MOV reg,[mem]`.

```
int r1 = 0, r2 = 0; // Объявим регистры ВМ
r1 = ram[val_01];
```

Последняя подпрограмма в два раза сложнее — вместо четырех строчек кода здесь восемь! Мы берем значение из памяти (помнишь про наш массив `ram`, куда мы записали содержимое ram.bin?) и сохраняем его в регистр виртуальной машины (EDX), далее берем первое значение после мнемоники в пи-коде (ECX) и выполняем между ними операцию XOR. Результат кладем обратно в память.

```
LAB_004022b0:
MOVZX  EDX ,byte ptr [r1 ]  ; DAT_00404240
MOV    EAX ,[RAM ]
ADD    EAX ,dword ptr [EBP  + param_2 ]
MOVZX  ECX ,byte ptr [EAX ]
XOR    ECX ,EDX
MOV    EDX ,dword ptr [RAM ]
ADD    EDX ,dword ptr [EBP  + param_2 ]
MOV    byte ptr [EDX ],CL
JMP    LAB_004022d5
```

На языке C это будет выглядеть таким образом:

```
r2 = ram[val_01];
ram[val_01] = r2 ^ r1;
```

Вот, собственно, и все. Виртуальная машина из трех команд восстановлена, осталось применить результаты нашего труда к файлу ram.bin, чтобы заполучить искомый флаг крэкми. Как я уже говорил, для этого читаем файл в `char ram[507]` и применяем декомпилятор кода ВМ. В качестве бонуса цикл выведет мнемоники виртуальной машины в удобочитаемом виде, а в конце напечатает искомый флаг. Я добавил в код уточняющие комментарии.

```
char ram[507];      // Память ВМ, ram.bin
int r1 = 0, r2 = 0; // Регистры ВМ

for (;;)
{
    int command = (int)ram[x];  // Берем опкод команды
    int val_01 = (int)ram[x + 1];   // Первый операнд команды
    int val_02 = (int)ram[x + 2];   // Второй операнд команды

    // Дешифровка кода
    if (command == 0x1)
    {
        ram[val_01] = val_02;
        cout << "mov " << "[" <<(int)ram[val_01] << "]" << "," << val_02 << endl;
    }
    if (command == 0x2)
    {
        r1 = ram[val_01];
        cout << "mov " << "r1" << "," << "[" << (int)ram[val_01] << "]" << endl;
    }
    if (command == 0x3)
    {
        r2 = ram[val_01];
        ram[val_01] = r2 ^ r1;
        cout << "xor " << "r2" << "," << "r1" << endl;
    }
    if (command > 3 || command < 1) break;
    x += 3;
}

printf("\n%s\n", &ram);     // Напечатаем результат
```

После выполнения этого кода мы получим дизассемблированную ВМ и флаг.

Результат работы восстановленной виртуальной машины

Заключение

Я надеюсь, что, прочитав статью, ты перестанешь пугаться слов «виртуальная машина» или «пи-код». Конечно, в настоящих коммерческих протекторах вроде VMProtect или Themida все будет намного сложнее: там может применяться множество команд виртуальной машины, их мнемоники-коды могут постоянно меняться, встречаются виртуальные машины, разные антиотладочные и антидамповые приемы, написанные на пи-коде, и многое другое. Но первое представление ты получил.

Заодно мы более близко познакомились с инструментарием под названием Ghidra и совершили с помощью него первый взлом, пусть даже крэкми.