АНТИВИРУСНАЯ ЗАЩИТА БИЗНЕС-КЛАССА

Как обнаружить FinFisher. Руководство ESET

03 мая 2018 года

Благодаря серьезным мерам противодействия анализу, шпионское ПО FinFisher оставалось малоизученным. Это известный инструмент слежки, тем не менее, по предыдущим образцам был опубликован только частичный анализ.

Ситуация стала меняться летом 2017 года после выполненного ESET анализа кампаний кибершпионажа FinFisher. В ходе исследования мы определили атаки с участием в компрометации жертв интернет-провайдера.

Когда мы начали анализ малвари, основные усилия были затрачены на преодоление мер по противодействию анализу FinFisher в ее версиях под Windows. Комбинация продвинутого обфусцирования и проприетарной виртуализации делает процесс снятия маскировки с FinFisher крайне трудным.

В данном руководстве мы делимся тем, чему научились в процессе анализа FinFisher. Кроме советов по анализу виртуальной машины FinFisher, руководство поможет понять защиту при помощи виртуальной машины в целом, то есть проприетарные виртуальные машины, обнаруживаемые в бинарном коде и используемые для защиты ПО.

Мы также проанализировали версии FinFisher под Android, механизм защиты которого основан на обфускаторе LLVM из открытого доступа. Он не так сложен и интересен, как механизм версий для Windows, поэтому в настоящем руководстве мы его обсуждать не будем.

Надеемся, что руководство будет полезно исследователям информационной безопасности и

вирусным аналитикам для понимания инструментов и тактик FinFisher, а также для защиты клиентов от данной угрозы.

Меры против дизассемблирования

Открывая образец FinFisher в IDA Pro, мы замечаем в основной функции простой, но эффективный метод противодействия дизассемблированию, первую защиту.

FinFisher применяет распространенную технику против дизассемблирования – сокрытие хода исполнения путем подмены одной команды безусловного перехода двумя комплементарными условными переходами. Они указывают на одинаковую точку перехода, поэтому, вне зависимости от выполняемого перехода, порядок исполнения кода не меняется. После условных переходов идут бессмысленные байты кода. Они предназначены для того, чтобы сбить дизассемблер с толку, так как в нормальных условиях он не сможет распознать нерабочий участок и будет продолжать трудиться над дизассемблированием этого мусора.

Особенной эту малварь делает способ использования данной техники. В большинстве вредоносных программ, которые мы изучали, метод используется определенное количество раз. Однако FinFisher применяет этот трюк после каждой команды.

Эта защита очень эффективна против дизассемлера и запутывает его так, что множество участков кода не проходят процесс должным образом. И, конечно же, графический режим в IDA Pro использовать становится невозможно. Наша первая задача — избавиться от этой защиты.
Код явно был обфусцирован не вручную, а с помощью автоматического инструмента, и мы наблюдаем некий шаблон во всех парах команд перехода.

Есть два различных типа пар перехода – внутренний переход с отступом в 32 бита и короткий переход с отступом в 8 бит.

Код операции обоих условных внутренних переходов (где DWORD — отступ перехода) начинается с байта 0x0F, а вторые байты равняются 0x8?, где? в обоих командах перехода отличаются лишь на 1 бит. Это связано с тем, что коды операций x86 ОС для комплементарных переходов последовательны в числовом выражении. Например, эта схема обфусцирования всегда связывает в пару JE с JNE (опкоды 0x0F 0x84 и 0x0F 0x85), JP с JNP (опкоды 0x0F 0x8A и 0x0F 0x8B) и так далее.

После этих опкодов идет 32-битный аргумент, определяющий отступ, куда будет совершен переход. Так как размер обеих команд 6 байт, отступы двух последовательных переходов различаются ровно на 6 (см. рисунок 1).

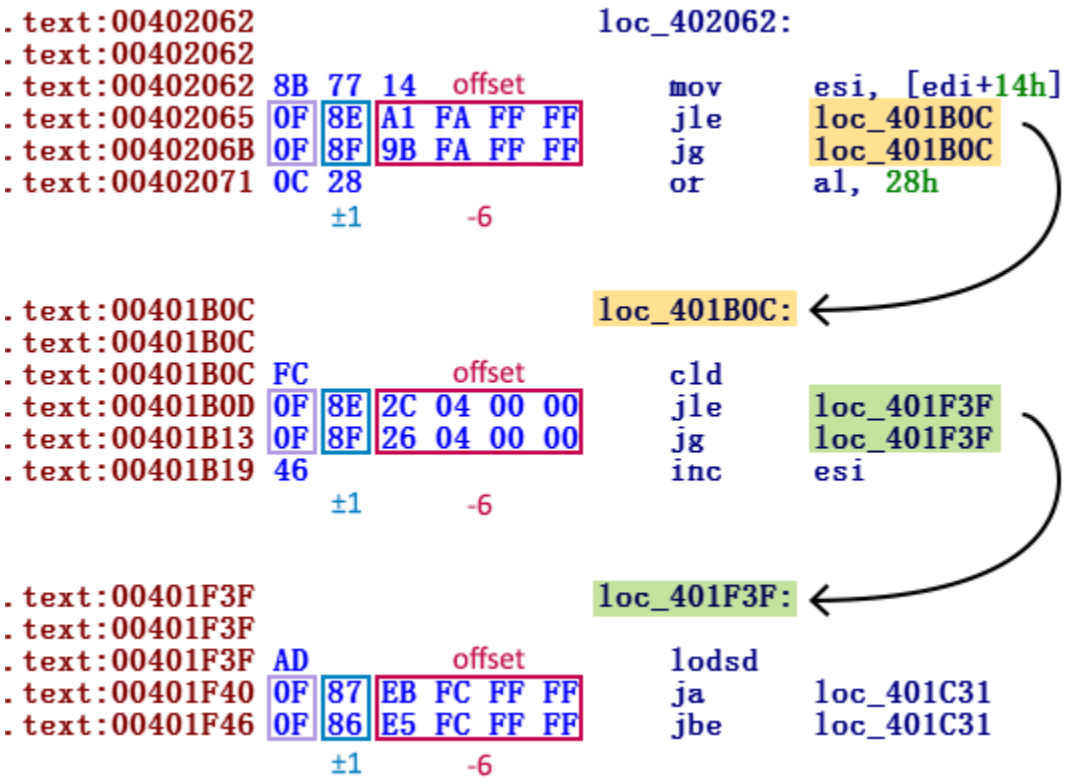

Рисунок 1. Скриншот с командами, после которых каждый раз идут два условных внутренних перехода

К примеру, код, приведенный ниже, можно использовать для обнаружения этих двух последовательных переходов:

```
def is_jump_near_pair(addr):
jcc1 = Byte(addr+1)
jcc2 = Byte(addr+7)
# они начинаются как внутренние условные переходы?
if Byte(addr) != 0x0F || Byte(addr+6) != 0x0F:
return False
# тут точно 2 последовательных внутренних условных перехода?
if (jcc1 & 0xF0 != 0x80) || (jcc2 & 0xF0 != 0x80):
return False
# комплементарны ли эти условные переходы?
if abs(jcc1-jcc2) != 1:
return False
# указывают ли они на ту же точку перехода?
dst1 = Dword(addr+2)
dst2 = Dword(addr+8)
if dst1-dst2 != 6
return False
return True
```

Деобфусцирование коротких переходов основано на похожей идее, лишь константы отличаются.

Опкод короткого условного перехода равен 0x7?, за ним следует один байт – отступ перехода. Поэтому опять мы ищем два последовательных условных внутренних перехода, и нам нужны коды операций: 0x7?; отступ; 0x7? ± 1; отступ -2. После первого опкода идет один байт, отличающийся на 2 в двух последовательных переходах (которые, опять же, имеют размер обеих команд) (рисунок 2).

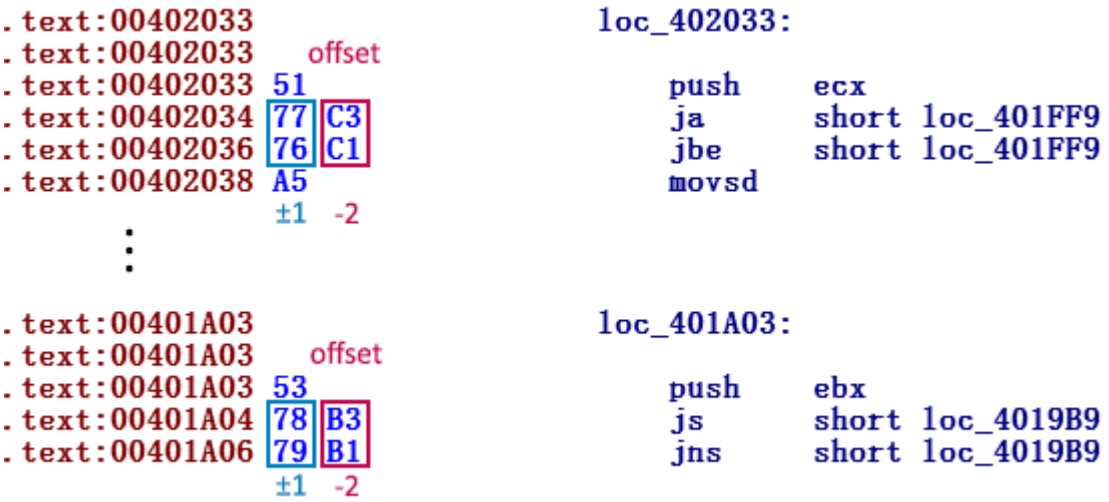

Рисунок 2. Примеры команд, после которых каждый раз идут два коротких условных перехода

Например, этот код может быть использован для обнаружения двух условных коротких переходов:

```
def is_jcc8(b):
return b&0xF0 == 0x70
def is_jump_short_pair(addr):
jcc1 = Byte(addr)
jcc2 = Byte(addr+2)
if not is_jcc8(jcc1) || not
is_jcc8(jcc2):
return False
if abs(jcc2-jcc1) != 1:
return False
dst1 = Byte(addr+1)
dst2 = Byte(addr+3)
if dst1 - dst2 != 2:
return False
return True
```

После обнаружения одной из этих пар условных переходов мы деобфусцируем код с помощью патча, превращая первый условный переход в безусловный (используя опкод 0xE9 для пар внутреннего перехода и 0xEB для пар короткого перехода) и заполняя оставшиеся байты пустыми командами (0x90)

```
def patch_jcc32(addr):
PatchByte(addr, 0x90)
PatchByte(addr+1, 0xE9)
PatchWord(addr+6, 0x9090)
PatchDword(addr+8,
0x90909090)
def patch_jcc8(addr):
PatchByte(addr, 0xEB)
PatchWord(addr+2, 0x9090)
```

Кроме этих двух ситуаций могут быть места, где пара переходов состоит из короткого и внутреннего, то есть разных типов. Но такое встречается только в нескольких местах, так что в образцах FinFisher это можно поправить вручную.

Используя эти вставки, IDA Pro начинает «понимать» новый код и готова (ну, или почти готова) к созданию диаграммы. Может так случиться, что нам потребуется выполнить еще одно улучшение: добавить окончания, то есть назначить ноду с указанием позиции перехода, чтобы на графике она

совпадала с командой перехода. С этой целью мы можем использовать функцию IDA Python `append_func_tail`.

Последний шаг обхода трюков, мешающих нормальной работе дизассемблера, состоит во внесении исправлений в определения функций. Может случиться так, что командой после перехода будет `push ebp`, в случае чего IDA Pro (ошибочно) считает это началом функции и, соответственно, начинает новое ее определение. В этом случае нам необходимо удалить определение функции, сделать корректную запись и добавить еще окончания.
Таким образом мы снимаем первый уровень защиты FinFisher, предназначенный против дизассемблирования.

Виртуальная машина FinFisher

После успешного деобфусцирования первого уровня защиты нам открывается главная функция, единственная цель которой – запуск специально созданной виртуальной машины и ее дальнейшее использование для интерпретации байткода с, собственно, полезной нагрузкой.

В отличие от обычного исполняемого файла, исполняемый файл с виртуальной машиной внутри использует набор виртуализированных команд вместо непосредственного исполнения команд процессора. Виртуализированные команды исполняются виртуальным процессором, который имеет собственную структуру и не конвертирует байткод в неуправляемый машинный код. Этот виртуальный процессор, как и байткод (и виртуальные команды), определяются тем, кто программирует виртуальную машину (рисунок 3).

Во введении мы говорили, что одним из известных примеров виртуальной машины является Java VM. Но в этом случае виртуальная машина находится внутри бинарного кода, поэтому здесь мы сталкиваемся с виртуальной машиной для защиты от реверс-инжиниринга. Существуют известные коммерческие средства защиты с помощью виртуальной машины, такие как VMProtect и Code Virtualizer.

Шпионское ПО FinFisher компилируется из исходников, а затем полученный бинарный файл защищается виртуальной машиной на уровне ассемблера. Процесс защиты включает в себя перевод инструкций исходного бинарного файла в виртуальные инструкции, а затем создание нового бинарного файла, содержащего байт-код и виртуальный процессор. Нативные инструкции из исходного бинарного файла теряются. Защищенный, виртуализированный образец должен иметь то же поведение, что и незащищенный образец.

Для анализа бинарного файла, защищенного виртуальной машиной, необходимо следующее:
1. Проанализировать виртуальный ЦП
2. Написать собственный дизассемблер для этого нестандартного виртуального ЦП и пропарсить байткод
3. Опционально: скомпилировать дизассемблированный код в бинарный файл, избавившись от виртуальной машины.

Первые две задачи требуют много времени, а первая вообще может стать довольно сложной. Она включает анализ каждого обработчика *vm_handler* и понимание того, как передаются регистры, доступ к памяти, вызовы и так далее.

Рисунок 3. Байткод, интерпретируемый виртуальным ЦП

Термины и определения

Для определения отдельных частей виртуальной машины не существует стандарта. Поэтому мы определим некоторые термины, к которым будем обращаться в данной работе:
• Virtual machine (vm) –виртуальный ЦП; содержит *vm_dispatcher, vm_start, vm_handlers*
• *vm_start* – инициализация; здесь происходит выделение памяти и процесс дешифрования
• Bytecode (также известный, как pcode) – здесь хранятся виртуальные опкоды *vm_instructions* с аргументами
• *vm_dispatcher* – вызывает и декодирует виртуальные опкоды; по сути готовит к исполнению какого-либо из *vm_handlers*
• *vm_handler* – реализация *vm_instruction*; исполнение одного *vm_handler* означает исполнение одной *vm_instruction*
• Interpreter (также известный, как *vm_loop*) – *vm_dispatcher* + *vm_handlers* – виртуальный ЦП
• Virtual opcode – аналог нативных опкодов
• *vm_context* (*vm_structure*) – внутренняя структура, используемая интерпретатором
• *vi_params* – структура внутри структуры *vm_context*; параметры виртуальных команд, используемых *vm_handler*; включает *vm_opcode* и аргументы

В процессе интерпретации байткода виртуальная машина использует виртуальный стек и единый виртуальный регистр:

• *vm_stack* – аналог нативного стека, используемого виртуальной машиной
• *vm_register* – аналог нативного регистра, используемого данной виртуальной машиной; далее называется tmp_REG
• *vm_instruction* – команда, определяемая разработчиками виртуальной машины; тело (реализация) команды вызывается ее *vm_handler*-ом

В следующих разделах мы опишем элементы виртуальной машины в технических подробностях и расскажем, как их анализировать.

Деобфусцированное графическое представление главной функции малвари состоит из трех частей: инициализации и двух других, которые мы назвали *vm_start* и интерпретатор (*vm_dispatcher* + *vm_handlers*).

Компонент инициализации задает уникальный идентификатор того, что может быть интерпретировано как точка входа байт-кода, и помещает его в стек. Затем идет переход к части *vm_start*, то есть процессу инициализации самой виртуальной машины. Происходит расшифровка байткода и контроль переходит к *vm_dispatcher*, который запускает циклы виртуальных команд байткода и интерпретирует их при помощи *vm_handlers*.

Запуск *vm_dispatcher* происходит с команды pusha и оканчивается на команде jmp dword ptr [eax+ecx*4] (или похожей на нее), то есть переходом на соответствующий *vm_handler*.

Vm_start

Графическая модель, создаваемая после деобфусцирования первого уровня, показана на рисунке 4. Часть, относящаяся к *vm_start*, не столь важна для анализа интерпретатора. Однако она помогает понять реализацию виртуальной машины в целом, то, как она использует и управляет виртуальными флагами, виртуальным стеком, и т.д. Вторая часть — *vm_dispatcher* с *vm_handlers* – это основа.

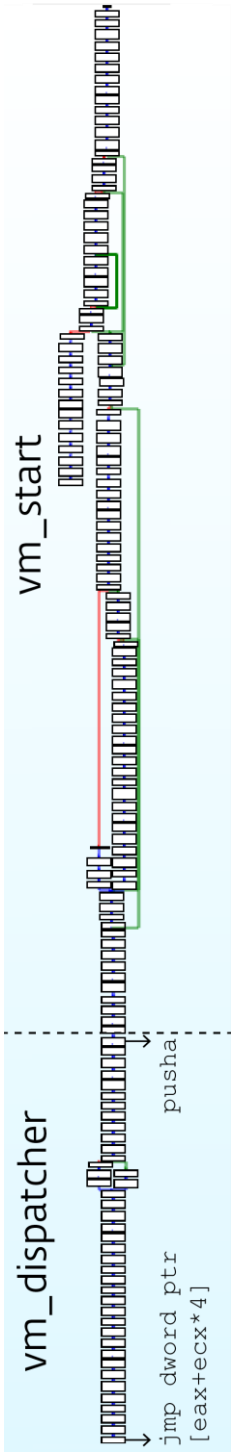

Рисунок 4. Графическое представление vm_start и vm_dispatcher

Вызов к *vm_start* производится из почти каждой функции, включая основную. Вызывающая функция всегда пушит виртуальный идентификатор и затем совершает переход к *vm_start*. Каждая виртуальная команда имеет собственный виртуальный идентификатор. В данном примере

идентификатор виртуальной точки входа, откуда начинается исполнение из основной функции, равняется 0x21CD0554 (рисунок 5).

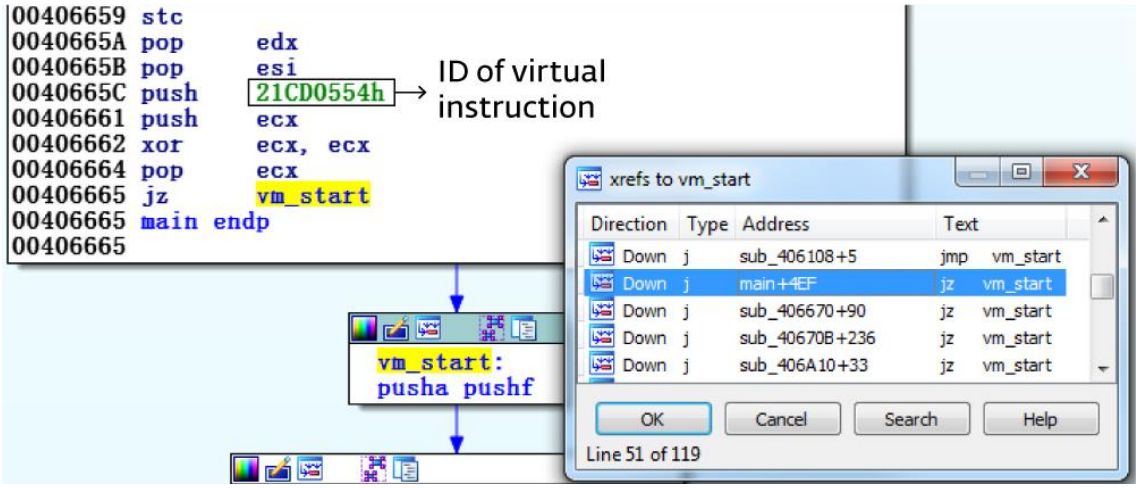

Рисунок 5. vm_start вызывается из каждой из 119 виртуализированных функций.

Идентификатор первой виртуальной команды соответствующей функции приведен в качестве аргумента.

В этой части большая часть кода отведена подготовке *vm_dispatcher* – в основном, байткода и выделению памяти для всего интерпретатора. Самые важные куски кода выполняют следующее:
1. Выделение для байткода и нескольких переменных 1 МБ памяти с разрешением на чтение, запись и исполнение.
2. Выделение 0x10000 байт с тем же разрешением для локальных переменных в виртуальной машине для текущей цепочки задач – *vm_stack*.
3. Дешифровка по XOR (исключающее ИЛИ). Дешифрованный код распаковывается как aPLib. Процесс дешифровки в образце использует слегка модифицированную версию ключа XOR dword. Он пропускает первые шесть dword и затем применяет XOR для оставшихся пяти dword с использованием ключа. Ниже приведен алгоритм процесса (в дальнейшем мы называем его XOR decryption_code):

```
int array[6];
int key;
for (i = 1; i < 6; i++) {
array[i] ^= key;
}
```

4. Вызов к процессу aPLib для распаковки байткода. После нее виртуальные опкоды все еще остаются зашифрованными (рисунок 6).

Подготовка виртуальных опкодов (шаги 1, 3 и 4) выполняется лишь раз, в начале, и пропускается в последующих исполнениях *vm_start*, в то время как выполняются только команды на использование флагов и регистров.

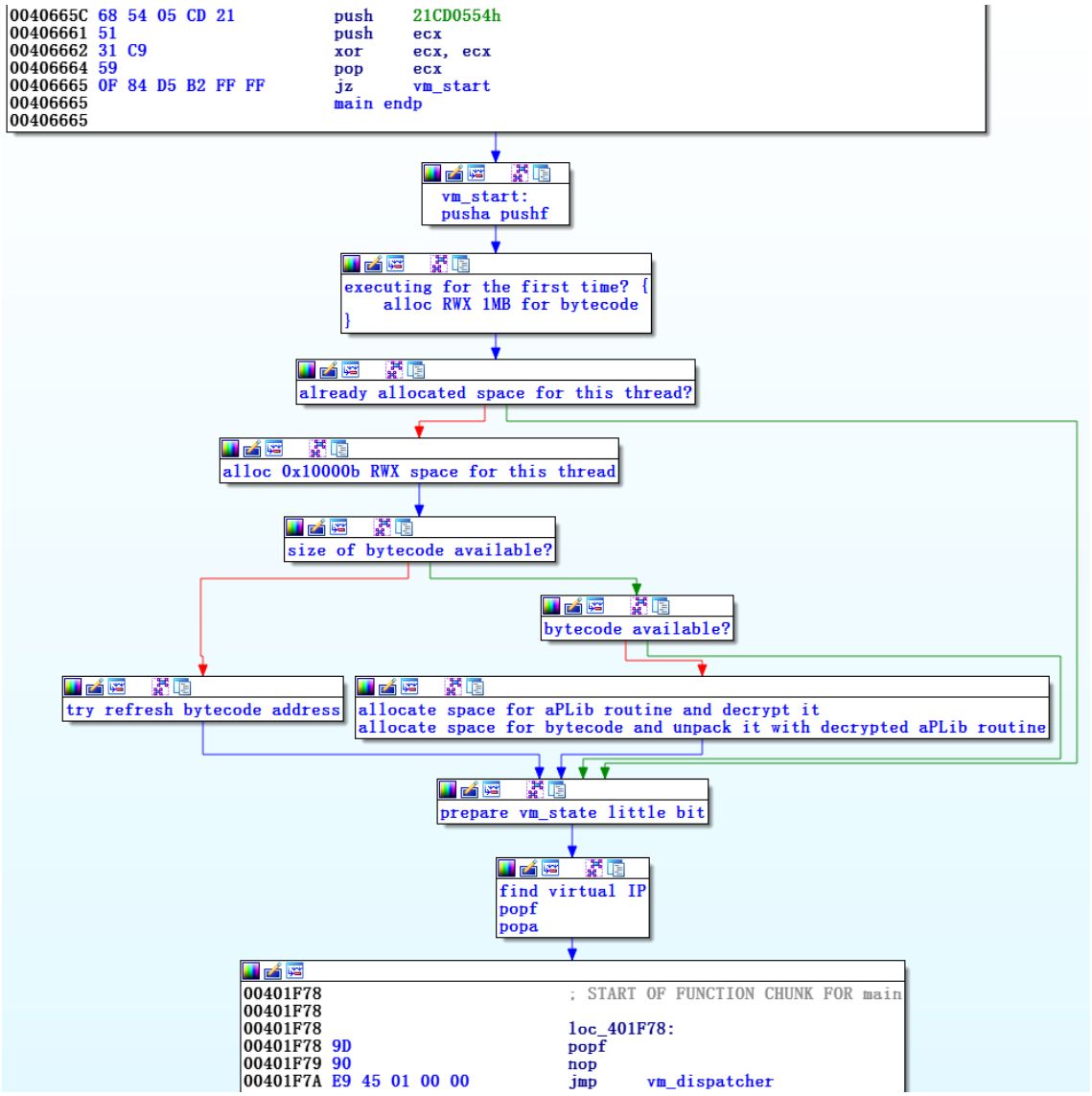

Рисунок 6. Весь код от vm_start до vm_dispatcher сгруппирован и назван в зависимости от назначения.

Интерпретатор FinFisher

Эта часть включает *vm_dispatcher* со всеми *vm_handlers* (34 в образцах FinFisher) и является важным элементом для анализа и/или девиртуализации виртуальной машины. Интерпретатор исполняет байткод.

Команда `jmp dword ptr [eax+ecx*4]` производит переход к одной из 34 *vm_handlers*. Каждый *vm_handler* реализует одну команду виртуальной машины. Чтобы понять, что делает каждый из *vm_handler*, нужно разобраться с *vm_context* и *vm_dispatcher*.

1. Создание графической структуры в IDA

Лучше понять интерпретатор поможет создание хорошо структурированной диаграммы. Мы рекомендуем его разделение на две части – *vm_start* и *vm_dispatcher*, то есть нужно определить начало функции в первой команде *vm_dispatcher*. В этом случае все еще отсутствуют сами *vm_handlers*, на которые ссылается *vm_dispatcher*. Для соединения этих обработчиков с диаграммой *vm_dispatcher*можно использовать следующую функцию:

```
AddCodeXref(addr_of_jmp_instr,
vm_handler,XREF_USER|fl_JN)
```

добавление в начало *vm_handlers* ссылок последней команды *vm_dispatcher*

`AppendFchunk`
производит добавление в конце

После добавления каждого обработчика *vm_handler* к функции диспетчера диаграмма выглядит так, как показано на рисунке 7 ниже.

2. Vm_dispatcher

Эта часть отвечает за обработку и декодирование байткода. Она выполняет следующие шаги:
• Исполняет команды pusha и pusf для подготовки виртуальных регистров и виртуальных флагов для последующего исполнения виртуальных команд
• Получает адрес программы в памяти и адрес *vm_stack*
• Читает 24 байта байткода, определяющего следующую команду *vm_instruction* и ее аргументы
• Дешифрует байткод с помощью ранее описанной процедуры XOR
• Добавляет адрес программы в памяти в качестве аргумента в байткоде в случае, если аргумент — это глобальная переменная
• Получает виртуальный опкод (число в диапазоне 0-33) из дешифрованного байткода
• Совершает переход к соответствующему обработчику *vm_handler* который интерпретирует виртуальный опкод

После того, как *vm_handler* для команды исполнен, повторяется та же последовательность для той, что следует за ней, начиная с первой команды *vm_dispatcher*. В случае обработчика *vm_call* управление передается *vm_start* (кроме тех случаев, когда за ней следует невиртуализированная функция).

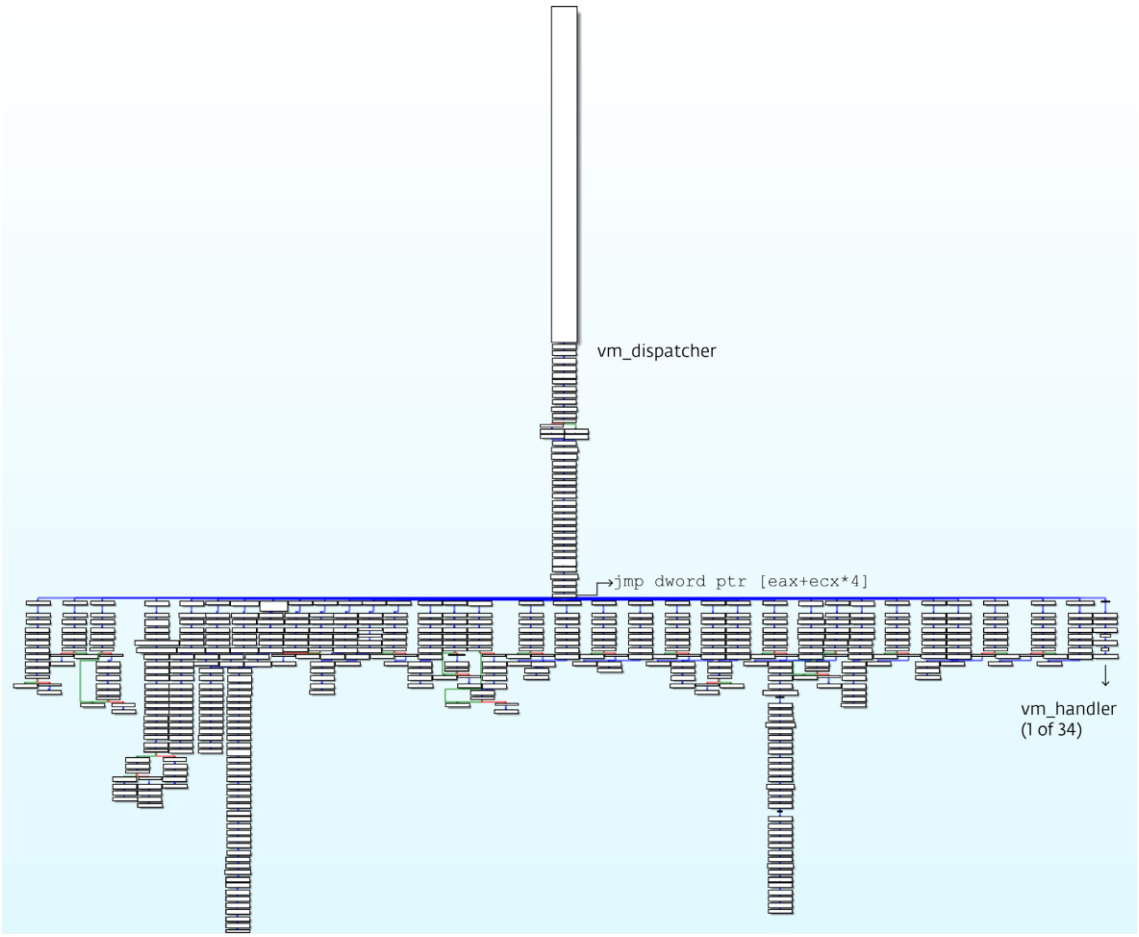

Рисунок 7. Диаграмма vm_dispatcher со всеми 34 vm_handlers.

3. Vm_context

В этой части мы опишем *vm_context* – структуру, используемую виртуальной машиной, содержащую всю необходимую для исполнения *vm_dispatcher* и каждого *vm_handler* информацию.

При более детальном изучении кода *vm_dispatcher* и *vm_handlers* мы можем заметить, что в нем присутствуют команды, связанные с обработкой данных, ссылающиеся на `ebx+offset`, где offset — это число от 0x00 до 0x50. На рисунке 8 можно видеть, как выглядит основная часть vm_handler 0x05 в одном из образцов FinFisher.

Регистр ebx указывает на структуру, которую мы назвали *vm_context*. Нам необходимо понимание того, как используется эта структура – каковы ее члены, что они означают, и как применяются. Для решения этой задачи в первый раз придется угадывать, как используются *vm_context* и его части.

Рисунок 8. Скриншот одного из vm_handlers

Для примера давайте посмотрим на последовательность команд в конце *vm_dispatcher*:

```
movzx ecx, byte ptr [ebx+0x3C]
// опкод для vm_handler
jmp dword ptr [eax+ecx*4]
// переход к одному из 34 vm_handlers
```

Так как мы знаем, что последняя команда — это переход к *vm_handler*, мы можем заключить, что ecx содержит виртуальный опкод, и поэтому 0x3C элемент *vm_struct* ссылается на виртуальный опкод.

Давайте сделаем еще одно небезосновательное предположение. В конце почти каждого vm_handler есть следующая команда:

```
add dword ptr [ebx], 0x18.
```

Тот же элемент *vm_context* был использован ранее в коде *vm_dispatcher* – прямо перед переходом к *vm_handler*. *vm_dispatcher* копирует 24 байта из элемента структуры в другое место (`[ebx+38h]`) и дешифрует его по XOR с целью получить часть текущего байткода.

То есть мы можем начать считать элемент *vm_context* (`[ebx+0h]`) указателем *vm_instruction_pointer*,

а дешифрованное положение (от `[ebx+38h]` до `[ebx+50h]`) как ID виртуальной команды, ее виртуальный опкод и аргументы. Всей этой структуре мы дали название *vi_params*.

После выполнения шагов выше и использования программы отладки мы видим, какие значения содержатся в соответствующих элементах структуры, то есть мы можем определить все элементы *vm_context*.

После анализа мы можем восстановить структуры и *vm_context*, и *vi_params FinFisher'a*:

```
struct vm_context {
DWORD vm_instruct_ptr; // указатель команды к байткоду
DWORD vm_stack; // адрес vm_stack
DWORD tmp_REG; // используется как "регистр" в виртуальной машине
DWORD vm_dispatcher_loop; // адрес vm_dispatcher
DWORD cleanAndVMDispatchFn; // адрес функции, выдающей значения и
совершающей переход к vm_dispatcher, пропуская первые несколько команд
DWORD cleanUpDynamicCodeFn; //адрес функции, очищающей vm_instr_ptr и
вызывающей cleanAndVMDispatchFn
DWORD jmpLoc1; // адрес позиции для перехода
DWORD jmpLoc2; // адрес следующего vm_opcode – простого исполнения следующей
vm_instruction
DWORD Bytecode_start; // адрес начала байткода в разделе данных
DWORD DispatchEBP;
DWORD ImageBase; // адрес программы в памяти
DWORDESP0_flags;//собственный стек (сохраненные флаги vm_code)
DWORDESP1_flags;//аналогично предыдущему
DWORD LoadVOpcodesSectionFn;
vi_params bytecode; // все необходимое для исполнения vm_handler, ниже
DWORD limitForTopOfStack; // вершина стека
};

struct vi_params {
DWORD Virtual_instr_id;
DWORD OpCode; // values 0 – 33 -> сообщает, какой обработчик исполнить
DWORD Arg0; // 4 аргумента dword для vm_handler
DWORD Arg4; // иногда не используется
DWORD Arg8; // иногда не используется
DWORD ArgC; // иногда не используется
};
```

4. Применение виртуальных команд – vm_handlers

Каждый из *vm_handler* оперирует одним виртуальным опкодом, то есть на 34 обработчика *vm_handlers*приходится максимум 34 виртуальных опкода. Исполнение одного *vm_handler* означает исполнение одной *vm_instruction*, поэтому для определения того, что выполняет *vm_instruction*, нам нужно проанализировать соответствующий *vm_handler*.

После реконструкции *vm_context* и наименования всех отступов в ebx, ранее показанный vm_handler становится более удобным для чтения, он показан на Рисунке 9.
В конце данной функции мы замечаем последовательность команд, начинающихся с *vm_instruction_pointer*, который увеличивается на 24, то есть на размер структуры *vi_params* каждой из *vm_instruction*. Так как эта последовательность повторяется в конце почти каждого *vm_handler* мы заключаем, что это стандартный завершающий код функции, а само тело *vm_handler* можно записать просто как:

```
mov [tmp_REG], Arg0
```

Итак, мы только что проанализировали первую команду данной виртуальной машины :-)

Рисунок 9. Предыдущий vm_handler после вставки в структуру vm_context

Для иллюстрации работы проанализированной команды давайте примем, что заполнение структуры vi_params выполняется следующим образом:

```
struct vi_params {
DWORD ID_of_virt_instr = любое значение, роли не играет;
DWORD OpCode = 0x0C;
DWORD Arg0 = 0x42;
DWORD Arg4 = 0;
DWORD Arg8 = 0;
DWORD ArgC = 0;
};
```

Из приведенного выше мы видим, что исполняется следующая команда:

```
mov [tmp_REG], 0x42
```

На этом этапе мы уже должны понимать, что выполняет одна из *vm_instructions*. Совершенные шаги должны служить демонстрацией работы всего интерпретатора.

Однако есть некоторые *vm_handlers*, которые проанализировать сложнее. Условные переходы этой виртуальной машины труднее понять из-за того, как происходит преобразование флагов.

Как отмечалось ранее, *vm_dispatcher* запускается с получения нативных EFLAGS (из *vm_code*) в вершину собственного стека. Таким образом, когда обработчик соответствующего перехода решает, совершить его или нет, он сверяется с EFLAGS в собственном стеке и применяет собственный метод перехода. Рисунок 10 иллюстрирует, как применяется виртуальный обработчик JNP (Jump if no parity) через проверку признака четности.

Рисунок 10. Скриншот JNP_handler

Для остальных виртуальных условных переходов может быть обязательной проверка нескольких признаков – например, результат перехода виртуализированного JBE (Jump if below or equal) зависит от обоих значений, флага переноса и флага нуля, но принцип тот же.

После анализа всех 34 *vm_handlers* в виртуальной машине FinFisher мы можем описать ее виртуальные команды следующим образом:

```
.text:00402ABA VM_table dd offset case_0_JL_loc1
.text:00402ABE dd offset case_1_JNP_loc1
.text:00402AC2 dd offset case_2_JLE_loc1
.text:00402AC6 dd offset case_3_vm_jcc
.text:00402ACA dd offset case_4_exec_native_code; same as case 6
.text:00402ACE dd offset case_5_mov_tmpREGref_Arg0; mov [tmpREG], Arg0
.text:00402AD2 dd offset case_6_exec_native_code
.text:00402AD6 dd offset case_7_JZ_loc1
.text:00402ADA dd offset case_8_JG_loc1
.text:00402ADE dd offset case_9_mov_tmpREG_Arg0; mov tmpREG, Arg0
.text:00402AE2 dd offset case_A_zero_tmpREG; mov tmpREG, 0
.text:00402AE6 dd offset case_B_JS_loc1
.text:00402AEA dd offset case_C_mov_tmpREGDeref_reg; mov [tmpREG], reg
.text:00402AEE dd offset case_D_mov_tmpREG_reg
.text:00402AF2 dd offset case_E_JB_loc1
.text:00402AF6 dd offset case_F_JBE_loc1
.text:00402AFA dd offset case_10_JNZ_loc1
.text:00402AFE dd offset case_11_JNO_loc1
.text:00402B02 dd offset case_12_vm_call
.text:00402B06 dd offset case_13_mov_tmpREG_reg; mov tmpREG, reg
.text:00402B0A dd offset case_14_JP_loc1
.text:00402B0E dd offset case_15_mov_reg_tmpREG; mov reg, tmpREG
.text:00402B12 dd offset case_16_JO_loc1
.text:00402B16 dd offset case_17_JGE_loc1
.text:00402B1A dd offset case_18_deref_tmpREG; mov tmpREG, [tmpREG]
.text:00402B1E dd offset case_19_shl_tmpREG_Arg0; shl tmpREG, (byte)Arg0
.text:00402B22 dd offset case_1A_JNS_loc1
.text:00402B26 dd offset case_1B_JNB_loc1
.text:00402B2A dd offset case_1C_push_tmpREG; push tmpREG
.text:00402B2E dd offset case_1D_JA_loc1
.text:00402B32 dd offset case_1E_add_tmpREG_reg; add tmpREG, reg
.text:00402B36 dd offset case_1F_vm_jmp
.text:00402B3A dd offset case_20_add_tmpREG_arg0
.text:00402B3E dd offset case_21_mov_tmpREG_to_Arg0Deref; mov [Arg0], REG
```

Рисунок 11. vm_table со всеми 34 vm_handlers

Обратите внимание, что ключевое слово «*tmp_REG*» относится к виртуальному регистру, используемому виртуальной машиной, временному регистру в структуре *vm_contex*t, в то время как “*reg*” относится к собственному регистру процессора, т.е. eax.

Давайте посмотрим на проанализированные команды виртуальной машины.
Например, *case_3_vm_jcc*— это общий обработчик команды перехода, который может исполнить любую команду перехода процессора, условную и безусловную.

Очевидно, что эта виртуальная машина не виртуализирует каждую аппаратную команду, вот где нам будут полезны команды из списка выше (case 4 и case 6).

Эти два *vm_handler* применяются для исполнения кода напрямую. Все, что они делают — это чтение опкода команды процессора в качестве аргумента и исполнение команды.

Еще нужно отметить, что *vm_registers* всегда находятся в вершине собственного стека, в то время как идентификатор исполняемого регистра хранится в последнем байте arg0 виртуальной команды. Для получения доступа к соответствующему виртуальному регистру можно использовать следующий код:

```
def resolve_reg(reg_pos):
stack_regs = ['eax', 'ecx', 'edx', 'ebx', 'esp', 'ebp', 'esi', 'edi']
stack_regs.reverse()
return stack_regs[reg_pos]
reg_pos = 7 - (state[arg0] & 0x000000FF)
reg = resolve_reg(reg_pos)
```

5. Написание собственного дизассемблера

После корректного анализа всех *vm_instructions* остается еще один шаг, который необходимо сделать до начала анализа образца – нам необходимо написать собственный дизассемблер для байткода (парсить его вручную будет проблематично по причине его размера).

Приложив усилия и написав более надежный дизассемблер, мы сэкономим себе силы впоследствии, когда виртуальная машина FinFisher изменится и обновится.
Начнем с *vm_handler 0x0C*, который исполняет следующую команду:

```
mov [tmp_REG], reg
```

Эта команда принимает точно один аргумент – идентификатор собственного регистра, используемого в качестве reg. Этот идентификатор должен отображаться с именем собственного регистра, например, используя команду `resolve_reg` в примере выше.

Чтобы дизассемблировать этот *vm_handler* можно использовать следующий код:

```
def vm_0C(state, vi_params):
global instr
reg_pos = 7 – (vi_arams[arg0]
& 0x000000FF)
tmpinstr = "mov [tmp_REG],
%s" % resolve_reg(reg_pos)
instr.append(tmpinstr)
return
```

Опять же, *vm_handlers* для переходов сложнее понять. В случае переходов компоненты *vm_context.vi_params.Arg0* и *vm_context. vi_params.Arg1* хранят отступ, на который происходит переход. Этот "отступ перехода" на самом деле является отступом в байткоде. Чтобы пропарсить обработчики перехода нам нужно переставить маркер на область перехода. Нам подойдет такой код:

```
def computeLoc1(pos, vi_params):
global instr
jmp_offset = (vi_params[arg0]
& 0x00FFFFFF) + (vi_params[arg1]
& 0xFF000000)
if jmp_offset < 0x7FFFFFFF:
jmp_offset /= 0x18 # their
increment by 0x18 is my
increment by 1
else:
jmp_offset = int((-
0x100000000 + jmp_offset)
/ 0x18)
return pos+jmp_offset
```

Наконец, есть *vm_handler*, отвечающий за исполнение нативных команд с аргументами, которым требуется особое обращение. Для этого нам нужен дизассемблер для собственных команд x86, например, инструмент Distorm из открытого доступа.

Длина команды хранится в vm_context.vi_params.OpCode & 0x0000FF00. Опкод собственной команды для исполнения хранится в аргументах. Чтобы пропарсить *vm_handler*, исполняющий родной код,

можно использовать код ниже:

```
def vm_04(vi_params, pos):
global instr
nBytes = vi_params[opCode] & 0x0000FF00
dyn_instr = pack("<LLLL", vi_params[arg0], vi_params[arg4],
vi_params[arg8], vi_params[argC])[0:nBytes]
dec_instr = distorm3.Decode(0x0, dyn_instr, distorm3.Decode32Bits)
tmpinstr = "%s" % (dec_instr[0][2])
instr.append(tmpinstr)
return
```

К этому моменту мы написали все функции на Python, чтобы разобрать каждый из *vm_handler*-ов. Все они, включая код, отвечающий за разметку областей для перехода, определение ID виртуальной команды после вызова, и некоторые другие, необходимы для написания собственного дизассемблера.

После всего этого его можно прогнать по байткоду.

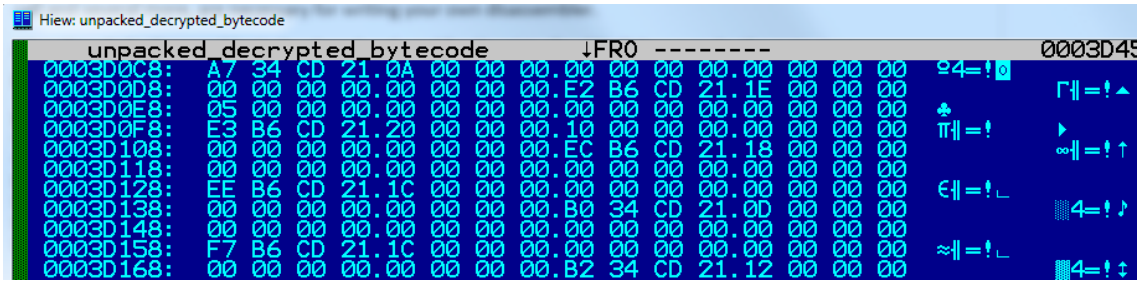

Рисунок 12. Часть нераспакованного и дешифрованного байткода FinFisher

К примеру, из байткода, приведенного в рисунке 12, мы можем получить следующий вывод:

```
mov tmp_REG, 0
add tmp_REG, EBP
add tmp_REG, 0x10
mov tmp_REG, [tmp_REG]
push tmp_REG
mov tmp_REG, EAX
push tmp_REG
```

6. Понимание применения данной виртуальной машины

После анализа всех виртуальных обработчиков и построения собственного кастомного дизассемблера мы можем снова взглянуть на виртуальные команды, чтобы понять основную идею их создания.

Сначала мы должны понять, что защита виртуализацией была применена на уровне ассемблера. Авторы преобразовали собственные команды в свои, неким образом усложненные, которые исполняются специальным виртуальным ЦП. Для этого используется временный «регистр» (*tmp_REG*).

Мы можем взглянуть на несколько примеров для того, чтобы понять, как работает это преобразование. Возьмем виртуальную команду из предыдущего примера –

```
mov tmp_REG, EAX
push tmp_REG
```

– она была преобразована из собственной команды `push eax`. Если применена виртуализация, используется временный регистр во временном шаге для изменения команды в нечто более сложное.

Возьмем другой пример:

```
mov tmp_REG, 0
add tmp_REG, EBP
add tmp_REG, 0x10
mov tmp_REG, [tmp_REG]
push tmp_REG
```

Вот собственные команды, которые были преобразованы в эти виртуализированные команды (где reg — один из собственных регистров):

```
mov reg, [ebp+0x10]
push reg
```

Однако это не единственный способ виртуализировать набор команд. Есть другие применения виртуальной машины для защиты, с другими подходами. Например, существует коммерческая реализация защиты через виртуальную машину, которая использует математическую логику NOR (когда оба ввода отрицательны), с несколькими временными регистрами вместо одного.

В свою очередь, FinFisher не пошел так далеко и не преобразовал все собственные команды. В то время как многие из них виртуализованы, некоторые не могут быть таковыми – это математические команды, такие как `add`, `imul` и `div`. Если эти команды оказываются в оригинальном бинарном файле, *vm_handler*, отвечающий за исполнение собственных команд вызывается для их обработки в защищенном файле. Единственное происходящее изменение состоит в том, что EFLAGS и собственные регистры извлекаются прямо перед исполнением собственной команды, а затем снимаются после завершения исполнения. Так возможно избежать виртуализации каждой собственной команды.

Существенным недостатком защиты бинарников с помощью виртуальной машины заключается в отрицательном воздействии на производительность. В случае с виртуальной машиной FinFisher мы приблизительно оцениваем скорость ее работы как более чем в сто раз медленную, чем в случае с внутренним кодом, на основании подсчета количества команд, исполняемых для обработки каждой из *vm_instruction* (*vm_dispatcher* + *vm_handler*).

Поэтому есть смысл защищать только выборочные части бинарника – и именно так поступают в тех образцах FinFisher, которые мы анализировали.

Более того, как уже говорилось, некоторые из обработчиков виртуальной машины могут напрямую вызывать собственные функции. В результате пользователи защиты виртуальной машиной (то есть авторы FinFisher) могут на стадии ассемблера решить, какие функции защитить при ее помощи. Для отмеченных функций их команды будут виртуализированы; для остальных оригинальные функции будут вызваны соответствующим виртуальным обработчиком. Таким образом, исполнение кода может быть менее затратным по времени, в то время как наиболее интересные части бинарного файла остаются защищенными (рисунок 13).

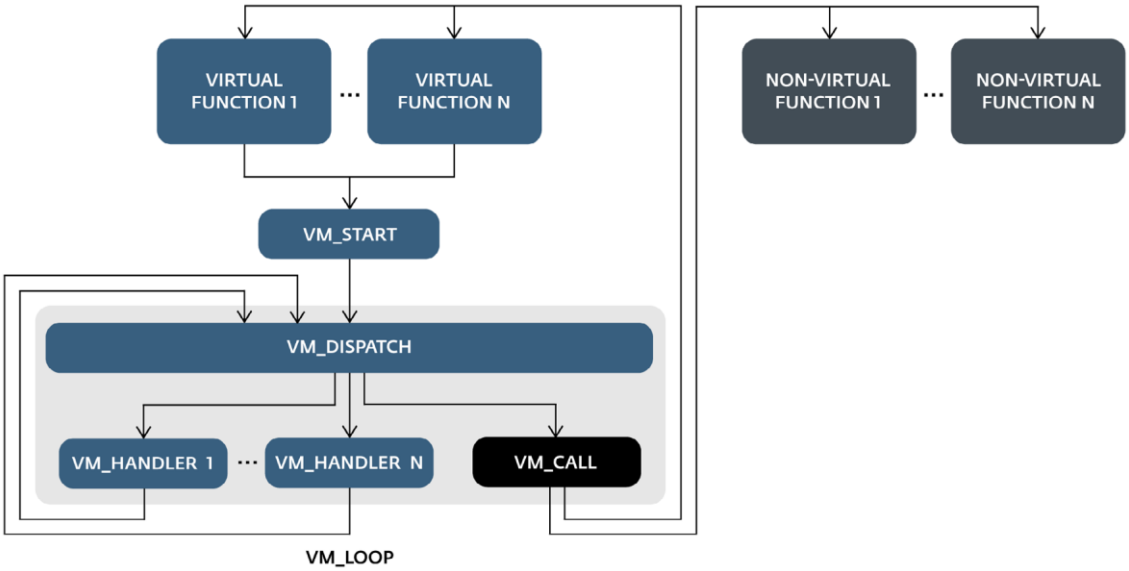

Рисунок 13. Схема, представляющая всю защиту FinFisher, и то, как процесс может совершать переход за пределы виртуальной машины

7. Автоматизация процесса дизассемблирования для большего количества образцов FinFisher

Помимо длины байткода, который нашему парсеру приходится обрабатывать, необходимо помнить, что в некоторых образцах FinFisher есть определенное перемешивание. И хотя для защиты используется одна и та же виртуальная машина, определение соответствия между виртуальными опкодами и *vm_handlers* не всегда совпадает. Они могут быть (и такое случается) рандомно спарены, и эти пары отличаются для разных образцов FinFisher, анализируемых нами. Это означает, что *vm_handler* для виртуального опкода 0x5 в этом образце обрабатывает команду `mov [tmp_REG], arg0`, а в другом защищенном образце он может быть назначен другому виртуальному опкоду.

Чтобы решить эту проблему, мы можем использовать сигнатуру для каждой из анализируемых *vm_handlers*. Скрипт для IDA Python в приложении A может быть применен после получения диаграммы из рисунка 7 (особенно важно убрать обфусцирование переходов jz/jnz, как это описано в первой секции данного руководства), чтобы дать имена обработчикам на основании их сигнатур. (С небольшими доработками этот скрипт можно также использовать и для восстановления сигнатур в случае, если *vm_handlers* будут изменены в обновленной версии FinFisher.)

Как указывалось выше, первый *vm_handler* в образце FinFisher, который вы встретите во время анализа, может отличаться от JL, который мы привели на примере образца, но скрипт определит все *vm_handlers* верным образом.

8. Компиляция дизассемблированного кода без виртуальной машины

После дизассемблирования и нескольких изменений код возможно скомпилировать. Мы будем использовать виртуальные команды как собственные. В результате мы получим чистый бинарный код без защиты.

Большинство команд *vm_instructions* можно скомпилировать простым копированием, так как на выходе дизассемблера, в основном, команды выглядят как нативные. Но некоторые участки нужно особенным образом доработать:

• tmp_REG – так как мы определили tmp_REG как глобальную переменную, нам нужно внести изменения в код для тех случаев, когда разыменовывается адрес, хранимый в ней. (Так как разыменовывание адреса, находящегося в глобальной переменной, невозможно для набора команд x86.). Например, виртуальная машина содержит виртуальную команду `mov tmp_REG, [tmp_REG]` которую необходимо переписать следующим образом:

```
push eax
mov eax, tmp_REG
mov eax, [eax]
mov tmp_REG, eax
pop eax
```

• Флаги – виртуальные команды не меняют флаги, а собственные математические команды делают это. Поэтому нам важно, чтобы виртуальные математические команды в девиртуализированном бинарнике не делали этого тоже, что означает необходимость сохранения флагов перед исполнением команд и их восстановления после завершения исполнения.

• Переходы и вызовы – нам необходимо переместить маркер на область виртуальной команды (перехода) или функции (вызова).

• Вызовы API функции – в большинстве случаев они загружаются динамически, в остальных случаях обращение к ним происходит из IAT (таблицы импорта адресов) бинарного файла, поэтому их нужно обрабатывать соответственно.

• Глобальные переменные, собственный код – некоторые глобальные переменные необходимо хранить в девиртуализированном бинарнике. Также, в дроппере FinFisher есть функция переключения между x64 и x86, которая выполняется в режиме процессора (на самом деле это выполняется только при помощи команды `retf`). Все это необходимо сохранить в процессе компиляции.

В зависимости от результата, на выходе вашего дизассемблера вам может еще понадобиться внести еще пару изменений для получения чисто собственных команд, которые можно будет скомпилировать. Затем вам нужно будет скомпилировать код при помощи предпочитаемого вами компилятора в бинарник без виртуальной машины.

Вывод

В данном руководстве мы описали, как FinFisher использует два способа защиты основного доставляемого компонента. Цель защиты – не противодействие обнаружению антивирусом, а сокрытие файлов конфигурации и новых техник, примененных в шпионском ПО, путем создания затруднений для реверс-инжиниринга. Поскольку на сегодняшний день другого детального анализа обфусцированного шпионского ПО FinFisher опубликовано не было, задачу разработчиков данного механизма защиты до текущего момента можно было считать успешно выполненной.

Мы показали, как уровень защиты, противостоящей дизассемблированию, можно преодолеть автоматическим методом, и как эффективно проанализировать виртуальную машину.

Надеемся, что руководство поможет специалистам по реверс-инжинирингу в анализе защищенных виртуальной машиной образцов FinFisher, а также обеспечит лучшее понимание специфики защиты с помощью виртуальной машины в целом.

Приложение А

Скрипт IDA Python для именования обработчиков vm_handlers в FinFisher

Этот скрипт также доступен в репозитории ESET на GitHub.

```
import sys
SIGS={'8d4b408b432c8b0a90800f95c2a980000f95c03ac275ff631c':'case_0_JL
_loc1','8d4b408b432c8b0a9400074ff631c':'case_1_JNP_loc1','8d4b408b432c
8b0a94000075a90800f95c2a980000f95c03ac275ff631c':'case_2_JLE_loc1','8d4
b408b7b508b432c83e02f8dbc38311812b5c787cfe7ed4ae92f8b066c787d3e7e
4af9b8e80000588d80' : 'case_3_vm_jcc', '8b7b508b432c83e02f3f85766c77ac6668
137316783c728d7340fb64b3df3a4c67e98037818b43c89471c64756c80775af83318588b6
32c' : 'case_4_exec_native_code', '8d4b408b98b438898833188b43c8b632c' : 'c
ase_5_mov_tmp_REGref_arg0', '8b7b508b432c83e02f3f85766c77ac6668137316783c7
28d7340fb64b3df3a4c67e98037818b43c89471c64756c80775af83318588b632c' : 'cas
e_6_exec_native_code','8d4b408b432c8b0a94000075ff631c':'case_7_JZ_loc1' ,
'8d4b408b432c8b0a94000075a90800f95c2a980000f95c03ac275ff6318' : 'case_8_
JG_loc1','8d43408b089438833188b43c8b632c':'case_9_mov_tmp_REG_arg0','3
3c9894b8833188b632c8b43c' : 'case_A_zero_tmp_REG', '8d4b408b432c8b0a980000
75ff631c':'case_B_JS_loc1','8d4b40fb69b870002bc18b4b2c8b548148b4b889118
33188b43c8b632c' : 'case_C_mov_tmp_REGDeref_tmp_REG', '8d4b40fb69b870002bc
18b4b2c8b4481489438833188b43c8b632c' : 'case_D_mov_tmp_REG_tmp_REG', '8d4b
408b432c8b0a9100075ff631c':'case_E_JB_loc1','8d4b408b432c8b0a9100075a94
000075ff631c':'case_F_JBE_loc1','8d4b408b432c8b0a94000074ff631c':'cas
e_10_JNZ_loc1','8d4b408b432c8b0a9080074ff631c':'case_11_JNO_loc1','8b7
b50834350308d4b408b414343285766c773f50668137a231c6472c280772aa8d57d83c7389
1783ef3c7477a300080777cb83c7889783ef8c647cf28077c3183c7dc67688b383c0188947
183c7566c7777fe668137176283c72c672d803745895f183c75c67848037df478b4314c674
08037288947183c75c67928037515f8b632c' : 'case_12_vm_call', '8d4b40b870002b
18b532c8b4482489438833188b43c8b632c' : 'case_13_mov_tmp_REG_tmp_REG_notRly
','8d4b408b432c8b0a9400075ff631c':'case_14_JP_loc1','8d4b40fb69b870002
bc18b4b2c8b5388954814833188b43c8b632c' : 'case_15_mov_tmp_REG_tmp_REG', '8
d4b408b432c8b0a9080075ff631c':'case_16_JO_loc1','8d4b408b432c8b0a90800f
95c2a980000f95c03ac274ff631c':'case_17_JGE_loc1','8b4388b089438833188b4
3c8b632c' : 'case_18_deref_tmp_REG', '8d4b408b4388b9d3e089438833188b43c8b6
32c' : 'case_19_shl_tmp_REG_arg0l', '8d4b408b432c8b0a98000074ff631c' : 'ca
se_1A_JNS_loc1','8d4b408b432c8b0a9100074ff631c':'case_1B_JNB_loc1','8b
7b2c8b732c83ef4b924000fcf3a4836b2c48b4b2c8b438894124833188b43c8b632c' : 'c
ase_1C_push_tmp_REG', '8d4b408b432c8b0a94000075a9100075ff6318' : 'case_1D_
JA_loc1','8d4b40b870002b18b532c8b448241438833188b43c8b632c':'case_1E_ad
d_stack_val_to_tmp_REG', '8b7b508343503066c77ac3766813731565783c728d4b40c6
72e803746fb6433d3c783c058947183c758d714fb64b3df3a45ac671280377a8b383c01889
47183c7566c777f306681371fac83c72c671f803777895f183c75c677080372b47c6798037
618b4b14894f183c75c67778037b48b632c8d12' : 'case_1F_vm_jmp', '8d4b408b914b
8833188b43c8b632c' : 'case_20_add_arg0_to_tmp_REG', '8d4b408b98b4388918331
88b632c8b43c' : 'case_21_mov_tmp_REG_to_arg0Dereferenced' }
SWITCH = 0 # addr of jmp dword ptr [eax+ecx*4] (jump to vm_handlers)
SWITCH_SIZE=34
sig = []
def append_bytes(instr, addr):
for j in range(instr.size):
sig.append(Byte(addr))
addr += 1
return addr
defmakeSigName(sig_name,vm_handler):
```

```
print “naming %x as %s” % (vm_handler, sig_name)
MakeName(vm_handler,sig_name) return
if SWITCH == 0:
print “First specify address of switch jump - jump to vm_handlers!”
sys.exit(1)
foriinrange(SWITCH_SIZE): addr = Dword(SWITCH+i*4) faddr = addr
sig = []
while 1:
instr = DecodeInstruction(addr)
if instr.get_canon_mnem() == “jmp” and (Byte(addr) == 0xeb or Byte
(addr) == 0xe9):
addr = instr.Op1.addr
continue
if instr.get_canon_mnem() == “jmp” and Byte(addr) == 0xff and Byte
(addr+1) == 0x63 and (Byte(addr+2) == 0x18 or Byte(addr+2) == 0x1C):
addr = append_bytes(instr, addr)
break
if instr.get_canon_mnem() == “jmp” and Byte(addr) == 0xff:
break
if instr.get_canon_mnem() == “jz”:
sig.append(Byte(addr))
addr += instr.size
continue
if instr.get_canon_mnem() == “jnz”:
sig.append(Byte(addr))
addr += instr.size
continue
if instr.get_canon_mnem() == “nop”:
addr += 1
continue
addr = append_bytes(instr, addr)
sig_str = “”.join([hex(l)[2:] for l in sig])
hsig = ‘’.join(map(chr, sig)).encode(“hex”)
for key, value in SIGS.iteritems():
if len(key) > len(sig_str): ifkey.find(sig_str)>=0:
makeSigName(value,faddr)
else: ifsig_str.find(key)>=0: makeSigName(value,faddr)
```