



Оглавление

Статья «ТИБЕРИУМНЫЙ РЕВЕРСИНГ»	1
Глава «What is SecuROM»	2
Глава «INTRO»	2
Глава «ПРАКТИКА»	3
Глава «Штирлиц на проводе. Центр, примите шифровку»	5
Глава «SecuROM 7 НАЧАЛО»	8
Глава «Виртуальная математика»	15
Глава «Перед заключением»	32
Глава «Заключение»	32
GLOSSARY	33
LINKS	36
Extended image part	37



Отлаживай & дизассемблируй

ТИБЕРИУМНЫЙ РЕВЕРСИНГ

© ELF, 2011 \ICQ: 7719116\

© CJ, 2011 \ICQ:3708307\

Благодарности: Nightshade, random, mak, DillerInc и всем остальным, чья помощь помогает в борьбе с DRM.

Внедрение X-кода и виртуальная машина: теория и практика

“Если ассоциировать SecuROM v7.33.17 с танком Абрамсом без динамической защиты, OllyDbg – РПГ-7, а X-code injection как кумулятивную гранату для нашего гранатомета, то, как и в реальности, такой выстрел навзничь прошьет броню этой тяжелой неповоротливой машины и достигнет поставленной цели – ОЕР... Выведенную из строя машину изучают Российские инженеры...”

What is SecuROM ?

SecuROM (полное название: **Sony DADC SecuROM**) – цифровая защита CD/DVD дисков от нелегального копирования.

Год рождения – 1999 г.

В роли **издателя** выступает Sony Digital Audio Disk Corporation AG (Sony DADC AG).

Местонахождение офиса – Австрия (**НЕ ПУТАТЬ С Австралией! там тоже есть их офис**), г. Аниф (недалеко от Зальцбурга), ул. Sonustraße 20 5081, тел. +43 6246 8800

В роли ведущего **автора** выступает – **Рейнгард Блаукович** (Reinhard Blaukovitsch)



Тип защиты – нанесение специальных физических меток на диск, которые якобы нельзя скопировать
Представительство в России – Ул. Стасовой д.4, 119 071 Москва; Телефон: +7 (495) 935 7218 312
Почта для посылки взломанного стафа – sales-russia@sonydadc.com

INTRO

Пока SecuROM 7 готовятся за кулисами, я расскажу, чем ты будешь заниматься ближайшие, по крайней мере, часа два. В нашем распоряжении имеется Command & Conquer 3: Tiberium Wars (с патчем 1.9). Замечу сразу, что нужный нам файл не spc3.exe (Но попозже он нам очень поможет!). Находим \RetailExe\1.9\ spc3game.dat. Грузим в отладчик упомянутый файл и по секции **.securom** понимаем, с чем имеем дело. F9... Не удалось запустить требуемый модуль безопасности... Тоже самое будет если в ольгу ничего не загружать и просто оставить открытой! SecuROM активно сопротивляется хакерскому инструментарию, поэтому требуется найти обходное решение. На моем 2k3 **IsDebuggerPresent** пропатченный был – **XOR EAX, EAX** сразу стояло после первой инструкции, поэтому эта ругательная табличка и без помощи отладчика вылезала. Сначала я решил, что защита определяет через хук отладчик, также стало понятным, что возиться с антиотладкой будет утомительно. Кстати, присоединиться к spc3game.dat процессу отладчиком нельзя – данная операция приведет к его завершению. Под конец выяснилось, что протектор успешно противостоит Prostop и API шпионам. Попробуем бить в борт защиты – вписываем в секцию **.securom** любые 10 байт и правим, к примеру, **TEST EAX, EAX** на **TEST EBX, EBX**... Замечательно! Вопиющая ошибка – отсутствующая проверка на целостность файла! Взломщик может спокойно пробить протектор до ОЕР невзирая на все старания разработчиков. Нам не понадобятся стелсеры, API мониторы –

антиотладку просто проигнорируем. Одним словом, предлагаю отвлечься от классики и взглянуть на защиту по новому.

Итак, начинаем. Во-первых, хотелось бы взглянуть на механизм проверки, однако дотянутся до `GetDriveTypeA` (безусловно, это WinAPI там есть, Ctrl+N в Ольке) нам мешает сообщение об обнаруженном отладчике. Во-вторых, нам нужно обязательно прибыть в ОЕР. Перейдя снова в Ольку и проанализировав код при активном сообщении, становится ясным, что основная часть протектора уже распакована. Замечательно! А теперь допустим, мы поставим вначале требуемой API переход на наш X-код, который сможет вытащить адрес возврата, показать нам его, положить обратно, проэмулировать перекрытые переходом байты и вернуть обратно управление! ... Чего? Чего? :)

ПРАКТИКА

Для этой главы используйте приложение к статье:

SOURCES.TXT (исходники асм кода)

По строго научному, X-код это код это осмысленная совокупность байт, внедренных посторонним лицом или программой в целевой код процесса, для выполнения определенной задачи. Все что связано с внедрением X-кода, чаще относят к области вирусологии, поэтому это сложная и объемная тема для разговора. Однако, в нашем случае все намного проще: X-код, который внедряем мы сами, выполняет одну несложную задачу - показ адреса возврата. Непосредственно внедрять свой X-код будем через отладчик **OllyDbg v2** aka Оля (Ольга, Ольга).

Для удобства разобьем нашу большую задачу на отдельные подзадачи.

1. **Задача:** Получить доступ на запись в секцию кода `kernel32`

Описание: По стандарту секция кода (`.text`) имеет атрибуты `Read/Execute` (чтение/выполнение). Поэтому код типа, где `77E41C00` - адрес в секции кода `kernel32`:

MOV BYTE PTR DS: [77E41C00], E9

Вполне на законном основании вызовет исключение `access_violation`, что не входит в наши планы. Так как у меня две винды, я решил патчить вистовский `kernel32` в 2k3 оффлайн (он же bithack). Вариант с использованием WinAPI `VirtualProtect` все-таки более предпочтителен (в исходниках X-кода он есть!), но я считаю, что на руку нестандартные решения. В 2k3 открытая на запись секция кода `kernel32` приводит к краху некоторые приложения (`VisualStudio 2008`, `OllyDbg 1.10`), зато данную шалость в висте они спокойно воспринимают! Жалко, что Windows не для хакеров пишут - закрытые на запись секции кода и непропатченный `IsDebuggerPresent` не есть хорошо :)

Решение: Заходим в 2k3. Запускаем `PeTools`. Не забывая выставить свои права доступа к библиотеке. Добавляем к секции кода `Write` атрибут и пересчитываем контрольную сумму. Аналогом `PeTools` может послужить утилита `editbin`, если установлен `VisualStudio`: например для пересчета checksum наберите в командной строке `"editbin /RELEASE E:\Windows\system32\kernel32.dll"`. Если виста запустилась и в раскладке карты памяти в ольге, в столбце `Access` секция кода `kernel32` любого процесса помечена как `RWE`, стало быть, все сделано правильно!

2. **Задача:** Найти свободное место для X-кода в `spc3game.dat`; найти адрес для установки перехода, который передаст управление на наш X-код; установить, какие байты будут перекрыты переходом; установить, по какому адресу потребуется вернуть управление после установки перехода

Описание: В `spc3game.dat` 11 секций (PE Header не при делах, хотя при большом желании внедриться можно и туда). Первые 6 однозначно не подходят (`text` - зашифрованный код самой игрушки; `data`, `rdata`, `rsrsc` - область данных программы; `tls` - TLS Callback; `rts.ver` - имеет атрибут только на чтение). `Ars` - при беглом просмотре код, подверженный обфускации, все атрибуты доступа, оставим в покое. `Est` - здесь находится точка входа, все атрибуты доступа, кандидатура подходит. Адресат внедрения - чаще всего, цепочка нулей в конце, оставленных для выравнивания секций и никем не используемых. `Artem` - название походит на шутку разработчиков, небольшой и не имеющий осмысленности код, секция имеет все атрибуты доступа, но лучше ее не трогать. `Celare` - однозначно все данные и ресурсы `SecuROM`'а, пропускаем. Одноименная секция `SecuROM` по утверждению Оли содержит таблицу импорта, имеет все атрибуты доступа, не трогаем. Теперь передача управления. Как показывает практика, переход лучше всего делать с распаковщика. В отличие от остальных структур, его код не модифицируется в процессе выполнения (хотя в нашем мире все непостоянно). Можно просто переставить точку входа на наш X-код, но для нас в этом нет необходимости.

Решение: Хвост секции `est` всегда свободен. В моем случае, начиная с адреса `11DB6D0h`, будет располагаться наш X-код. Ставим точку останова при доступе к секции `text` (распаковщик ее обязательно затронет и даст знать о себе). Запускаем приложение. Оказываемся в функции с адресом, в моем случае, равным `11DB1F9h`. Я выбрал адрес `11DB27Eh`. Переход будет длинным, значит один байт `E9` плюс 32х битный операнд, который займет 4 байта, значит всего 5 байт. По адресу `11DB27Eh` идут `ADD EDI,ESI` и `ADD ESI,EBX` которые занимают 2 байта и `MOVZX EBX,BYTE PTR DS:[ECX+7]` которая занимает целых 4 байта. Наш переход полностью перекрывает первые две и последним байтом задевает `MOVZX`, значит, эмулировать придется все эти три команды, запоминаем

их. Теперь вычислим, куда потребуется возвратить управление. В сумме все три, будучи эмулируемые нами, команды занимают $2+2+4 = 8$ байт. Значит, $11DB27Eh + 8h = 11DB286h$. К этому адресу мы и возвратимся. Пишем по адресу $11DB27E$: `JMP 11DB6D0` и добавляем три `NOP`. Стоит отметить, что никто нам не мешает возвратиться сразу на адрес после прыжка (точнее на первый `NOP`), впрочем, большого смысла в этом нет.

3. Задача: Первая часть X-кода (подготовительная). Проэмулировать перекрытые переходом `JMP 11DB6D0` инструкции. Передать управление собственно на X-код. Вычислить адрес `GetDriveTypeA`. Установить какие байты будут перекрыты переходом, ведущим ко второй части нашего X-кода и проэмулировать их. Организовать вычисление операнда инструкции `JMP` для 2го перехода. Корректно записать инструкцию прыжка на 2ю часть X-кода. Возвратить управление.

Описание/Решение: Три аспекта объясню подробно. В нашем случае идеально первая часть X-кода должна выполняться один раз, больше и не нужно. Но распаковщик работает в цикле, о чем говорит регистр `EAX`, который с каждым разом инкрементируется. Мы сначала проверим регистр `EAX` на одно фиксированное значение (в моем случае `6Ch`), если равно, то передаем управление X-коду. Получение адреса функции из библиотеки это две API:

`GetModuleHandle` и `GetProcAddress`

В регистре `EAX` – адрес требуемой API.

С перекрытыми байтами `GetDriveTypeA` и вообще в большей части Windows API есть полезная особенность компилятора VC++! Первые три инструкции в экспортируемых функциях системных библиотек винды это `MOV EDI, EDI; PUSH EBP; MOV EBP, ESP` которые и дадут в сумме нужные нам 5 байт, и предпринимать каких либо дополнительных мер не потребуется! Разбираем операнд инструкции `JMP`. Вся проблема в том, что `kernel32`, равно как и другие системные библиотеки может “плясать” по адресному пространству процесса даже в пределах одной винды. Отсюда выходит, что пропатчить верхушку `GetDriveTypeA` фиксированным значением нельзя, ведь, в конце концов, на месте адреса, который сейчас ты видишь в отладчике, вполне реально окажется, не то, что хотелось, к тому же и операнд прыжка укажет в совсем другое место! Почему? Напоминаю, что в качестве операнда для любой из инструкций прыжка указывается количество байт, которые надо перепрыгнуть, а не адрес назначения! Дополнительно: x86 операнд записывается в обратном порядке и независимо от условного или безусловного перехода операнды определяются одинаково! Чтобы правильно вычислить операнд, я сначала посчитал разницу между точкой входа в упомянутую WinAPI и адресом назначения прыжка, затем так как прыжок будет идти в сторону младших адресов, плюс, учитывая размер инструкции `JMP LONG` (5 байт), получил окончательное значение:

```
SUB EAX,cnc3game.011DB6E3
MOV EDI,-5 // EDI = FFFFFFFBh
SUB EDI,EAX
```

4. Задача: Вторая часть X-кода (основная). Извлечь адрес возврата. Преобразовать байты адреса возврата в ASCII, для корректного вывода. Вывести на экран адрес возврата. Корректно осуществить возврат в системную библиотеку и эмуляцию перекрытых байт.

Описание/Решение: Процесс запустился, первая часть X-кода успешно выполнялась, управление передается на `GetDriveTypeA`, затем через поставленный нами переход мы оказываемся в начале второй части нашего представления. Адрес `011DB6E3h`. Мы разберем самые хардкорные аспекты его действия. Если ты уже понял, то перед нами стоит проблема корректного отображения адреса возврата. Действительно, если в качестве аргумента для текста `API MessageBoxA`, скормить адрес возврата, то в сообщении нашего сателлита вместо адреса возврата будет откровенная ерунда (под словом “ерунда” понимается ASCII строка, которая начинается с адреса возврата), ведь для винды `00404001` это адрес/операнд, откуда надо считать строку, но не сама строка! Так как я не знаю API, которая могла бы выполнить такое преобразование(`_itoa` опустим), было принято решение написать самостоятельно код, который мог бы выполнить требуемую операцию. На самом деле это несложно! Я рассуждал так: каждому символу в ASCII соответствует свой код. Для того чтобы узнать как будет закодирован каждый байт адреса возврата, потребуется организовать цикл. Коды чисел от 0 до 9 в ASCII имеют значение `30-39h`, кодам букв (в нашем случае это числа) от `A` до `F` присвоены соответственно значения `41-46h`. Руководящая идея – организовать сравнение байт адреса возврата и эталонного значения, посредством инкрементов последнего и его кодов в ASCII таблице. Если эталонный байт = сравниваемому, то мы пишем два значения их кода по специальному отведенному адресу (в байте две цифры, поэтому мы используем старшую и младшую часть регистра `ECX` для ASCII кода каждой из двух циферок). Последнее замечание касается перескока с 9 на `A` – в ASCII таблице их коды `39h` и `41h` соответственно, а при инкременте `39h` следующее число `3Ah`, за которым закреплен другой символ. Чтобы не упустить момент, скажу сразу, что кроме `MessageBoxA` существуют еще множество вариантов с выводом информации:

1) С помощью `CreateWindowEx`. Создать окно с контролами (`Edit`, `ListBox`) и отсылать в ставку с помощью `SendMessage` секретную информацию.

2) В файл с помощью *CreateFile/WriteFile/CloseHandle* и их низкоуровневых аналогов. Сюда же можно приписать именованные каналы(PIPE).

3) Весьма оригинальный: с помощью DirectX(Draw или даже 3D). Благо игрушка сама подключает библиотеку. Однако здесь нужно уметь работать с интерфейсом от МелкоСофт, да и удобнее уже будет написать свою отдельную Dll'ку для работы и инжектировать ее с самой игрой.

5. **Задача:** Проверить работу внедренного X-кода.

Правильный ответ: Кроме зеленого логотипа игрушки на заднем плане, первым мы должны увидеть наш спутник – MessageBox сообщаящий нам 8 hex цифр. Это и есть первый вызов GetDriveTypeA и его точка возврата в обратном порядке (если X-код перед глазами, то надеюсь, понимаешь, почему он транслирует адрес в обратном направлении). Первый адрес не относится к spc3game.dat и лежит где-то в ntdll. Затем MessageBox выскочит еще n раз, где n – количество логических устройств на твоей машине (вместе с виртуальными приводами, естественно). После выходит окошко от SecuROM'a с просьбой вставить нужный диск. После нажатия "Повтор" спутник покажется n раз, затем снова окно с просьбой вставить требуемый диск.

Неправильные ответы и их возможные причины: Спутник не появился, но игрушка работает – Вторая часть X-кода не получила управления -> Первая часть X-кода ошибочно поставила переход в другом месте. Ошибка программы – Необрабатываемое исключение -> Кодовая секция kernel132 без атрибута записи ИЛИ X-код ошибочно передал управление в другую область памяти. Спутник появился, но вместо 8 циферок откровенная ерунда – вместо ACSII строки адрес возврата. Адрес в спутнике указывает на несуществующую область памяти – адрес возврата декодирован в ACSII строку неверно.



“Штирлиц на проводе. Центр, примите шифровку”

Наш Штирлиц уже передает нам hex адреса. Осталось скорректировать область его работы и записывать результаты на бумагу. Помимо уже поднадевшей API GetDriveTypeA можно немного переделать код и с нужными параметрами организовать прослушку *CreateThread*, *DriveIoControl*, *CreateFileA*, *RegQueryValueExA*, *KiUserCallbackDispatcher*. Одним словом работаем, как

стандартный API шпион. А в чем тогда разница между оным и методом в нашем случае? Типичный API-spy использует стандартные процедуры меж процессного взаимодействия (*ReadProcessMemory*, *WriteProcessMemory* или ниже) и может быть легко обнаружен протекторами. Причем хороших API-spy немного и они хорошо известны разработчикам защитных комплексов (причем этот факт легко проверить – прослушайте *CreateFileW* в протекторе). С появлением новых версий защитных систем, типичные методы работы шпионажа за API медленно и постепенно уходят в прошлое. Инновационным и самым современным подходом, который лишен всех недостатков обычных API-spy и достигает самой максимальной скрытности, за счет использования ресурсов самого протектора стал X-code injection. Настоящий “Штирлиц” в тылу врага.

Собственно, что представляет собой SecuROM 7 изначально? Код, подверженный обфускации, причем мусора не очень много, ставка сделана на ассемблерные трюки. Можно сказать это кладезь всевозможных ухищрений. Например:

```
MOV EAX,11D7B1C // Ахтунг! Начальный адрес зоны проверки, предъявите ваши программные точки
останова в развернутом виде :)
```

```
{
MOV ESI,DWORD PTR DS:[EAX] //грузим след DWORD
ADD DWORD PTR SS:[ESP+10],ESI //складываем с предыдущим DWORD'ом
ADD EAX,4 // the next offset DWORD
DEC WORD PTR SS:[ESP+0C] // 114 DWORD'ов над сложить
JNE SHORT 011D7B76 // - while (dword [ESP+0Ch] != 0)
}
OR BYTE PTR SS:[ESP+10],01 // добавляем в младший байт единицу
SUB DWORD PTR SS:[ESP+10],933 // вычитаем “контрольную сумму”, пасьянс сошелся если:
DWORD [ESP+10] <= 0. (Вообще-то там всегда нуль должен быть, отрицательное число –
разработчики просто решили подстраховаться)
PUSHFD //сохраняю флаги (EFL = 206h)
... //код, предназначенный для отвода глаз
POPFD //выстаскиваю флаги (EFL = 206h)
JBE SHORT 011D7BC1 //так сошелся ли все-таки наш пасьянс? (Zero Flag(Z) = 1 или(и) Carry
Flag(C) = 1?)
```

Также, чаще смотрите, что вы трассируете:

```
00C49160 PUSHFD
00C49161 MOV EAX,DWORD PTR SS:[ESP]
00C49164 NOP
00C49165 TEST AH,1
00C49168 JE SHORT cnc3game.00C4916F
00C4916A MOV ECX,7BE
00C4916F XOR EAX,EAX
```

Оказывается, в SONY DADC еще нашли весьма оригинальную замену инструкции MOV ESI, DWORD PTR DS:[ESI]:

```
MOV DWORD PTR SS:[ESP],ESI
XOR ESI,DWORD PTR DS:[ESI]
XOR ESI,DWORD PTR SS:[ESP]
```

Если поставить break on memory access на *IsDebuggerPresent*, то потрассировав процедуру, которая попадет в сети, можно увидеть:

```
SUB ESI, EAX
LEA ECX, DWORD PTR DS: [ESI+EBX-5]
MOV BYTE PTR DS: [EAX], 0E9
MOV DWORD PTR DS: [EAX+1], ECX
```

“Ба! Где-то я это уже видел :)”. Как и мы SecuROM 7(этот список еще дополняет AsProtect)не менее активно используют прием с постановкой переходников: вычисляют адрес API, вычисляют длину перехода, прибавляют к нему 5 байт (MOV EDI,EDI; PUSH EBP; MOV EBP, ESP), записывают переходник, ну и эмулируют открытие кадра стека. Теперь, если протектору требуется вызов API, то он делает это через переходник, эмулируя открытие кадра стека у себя и попадая на 5 байт “раньше” в область библиотеки с требуемой API. Вот мы и ознакомились с основным приемом против API-spy. Дело в том, что последние на инструкцию MOV EDI,EDI привыкли вешать хуки,

которая в нашем раскладе никогда не получит управление! Правда по каким-то неизвестным причинам, SecuROM 7 не задействует переходники в своей работе. Но на будущее в своей реализации X-code injection надо учитывать этот трюк. Итак, возвращаемся к показаниям “Штирлица”. К сожалению, время поджимает, равно, как и место, отведенное в журнале(самая оригинальная фраза в тексте). Пробежусь по основному. Отличительной особенностью этой версии протектора являются illegal instruction UD2(плюс аппаратные точки останова с ними), на которые повешены SEH-обработчики. Коих я насчитал всего два(00C996B0h и 00C99702h). Если проанализировать их, то все они ведут к JMP... и выполняют абсолютно одну и ту же работу. Но обо всем по порядку.

Цепочка `GetDriveTypeA`(набор приводов)->`CreateFileA`(открытие привода на секторном уровне)->`DeviceIoControl`(проверка на наличие диска в заданном приводе, количество секторов с дорожками)-> `DeviceIoControl`(первичные 5 заходов)-> `MOV DWORD PTR DS:[EBX+EDI*8+4],EAX` (Если проэмулировать наличие диска в приводе, находим головную часть проверки)->`QueryPerformanceCounter`(вторая часть проверки) и синхронные потоки...
`CMP BYTE PTR SS:[EBP-52],BL` // проверка на наличие диска в заданном приводе

Можно было конечно взяться и раскручивать проверочный механизм до конца со всеми этими подканалами и тдтп. Однако предлагаю прокрутить вперед, к процессу снятия дампа. Разберемся с ним, и тогда сразу поймете, на чем работает исследуемая версия SecuROM’a.

Еще пару интересных моментов с `GetDriveTypeA` и за hardware breakpoints . Я назвал эту шутку X-code flash bag. Зная расположение WinAPI и чтение ветки HKLM\System\MountedDevices в защитном механизме можно скрыть от протектора виртуальный привод, если это потребуется или сразу все приводы! Второй момент, связан непосредственно с вызовом самой API и ей подобных – в нескольких местах протектор вызывает их через серию JMP и стандартную тройку `LoadLibrary/GetModuleHandle/GetProcAddress`, первый раз такое мне довелось увидеть в `SafeDisk v4.5`. Что касается аппаратных точек останова, то как я уже упоминал выше, на них также повешены упомянутые SEH-обработчики, это я к тому, как без `ZwContinue` и подобных операций с контекстом SecuROM 7 выясняет наличие отладки. Протектор не проверяет, что породило исключение... Впрочем, это так, к слову!

Ну а дальше, самое интересное, что я подготовил! Переходим к нахождению OEP и снятию дампа. Нам понадобится **Daemon Tools**(или любой другой эмулятор привода) и мини-дамп оригинального диска игрушки. Хотя, если окончательно доразобрать процедуру проверки и поправить нужные переходы, то можно обойтись без эмуляции. Потребуется любой ценой попасть в оригинальную точку входа и снять корректный дамп, и эта задача действительно решается. Поможет нетронутый никем `спс3.exe`, который сообщает нам, что разработчики писали игрушку на Microsoft Visual C++ 7.0. Точка входа выглядит как:

```
004628DA CALL 004784B8
004628DF JMP 004626FA
```

Реально, точка входа находится по операнду прыжка - 004626FAh. Функция по адресу 004784B8h ничем полезным не занимается, и я ее всегда бросаю ф топку. Однако в ней мы находим, что она вызывает несколько API(`GetSystemTimeAsFileTime`, `GetCurrentProcessId`...). Для нас это означает только одно – если поставить нашего “Штирлица” на `GetSystemTimeAsFileTime` и после распаковки стартовый код действительно на своем законном месте... Принимаемся за реализацию. Монтируем мини-дамп. Запускаем игру. Нас интересуют все `MessageBox` после проверки: 00DDCE77,76B414D4,7C34207B, 0040A5AE... **СТОООП!** Последний, да ведь вызов идет из секции .text !!! *We need attach now!* Присоединяемся к процессу Ольгой 1.10 с дампером и переходим по последнему адресу (аттачиться уже можно, `DbgUiRemoteBreakin` правится только на время проверки диска). Невероятно! :) Перед нами действительно OEP! Получилось! Переходим по прыжку к адресу 0040A006 и дампит наш процесс. Открою сразу секрет – можно ювелирно попасть в точку оригинального входа, но об этом гораздо ниже. Теперь самое главное: SecuROM 7 имеет полномочия защищаться от дампинга. Дамп естественно получается изначально нерабочим. Беглый анализ приносит известия, что на некоторые вызовы (к примеру, чуть ниже, по адресу 00A400D) ведут не к дочерним процедурам, через серию `JMP DWORD PTR DS:[address]` и “базу запросов” попадают в аллочную память к всемогущей виртуальной машине(VM). После ее работы двойное слово по адресу прыжка магическим образом меняется и уже указывает на нужную процедуру. Причем еще до обращения к “базе запросов” она уже была по своему адресу. Глядя на бесконечно длинный ужасный код этого виртуального чуда, складывается впечатление – реально ли вообще отвязать SecuROM от программы? А может лучше постараться приклеить выделенную память протектора к игре в качестве новой секции (подавляющее большинство так и поступает, т.к. это наиболее простой способ, но в плане оптимизации – самый жуткий)?

SecuROM 7 НАЧАЛО

Для этой главы используйте приложение к статье:

Материалы по работе механизмов навесной защиты SecuRom версии 7.33.0017

Прежде чем перейти к разбору VM, не лишним будет взглянуть собственно на саму навесную броню SecuROM 7. Если тебя это мало интересует, то эту главу можно пропустить. Кстати я вскользь уже затронул выше некоторые аспекты деятельности протектора, но чтобы все осмыслить – лучше разложить по полочкам. Итак:

- **Информационный раздел**

SecuROM Data Block. Специальный зашифрованный блок данных, который содержит характеристики вшитой защиты. Самое полезное здесь – точный номер версии. Содержится в PE-хидере между MZ заголовком и структурой PE_DOS.

CreateThread. Кроме создания двух потоков для второй части проверки, используется для создания еще одного специального потока – для вывода сообщений на экран(типа: *не удалось загрузить требуемый модуль безопасности ака твой мать! Выгрузи отладчик!*).

- **Активный раздел**

IsDebuggerPresent. Недвусмысленная функция, которая известна даже начинающим программистам. Способ простой и понятный: в PEв при отладке устанавливается флаг BeginDebugged, соответственно и сама WinAPI обращается по указателю FS:[18] в TEB и через ссылку в PEв извлекает содержимое этого флага. Самое первое, что приходит на ум – вставить инструкцию XOR EAX, EAX... но в SONY DADC решили перехитрить реверсеров и в BeginDebugged заносить контрольные значения, больше единицы и проверяет, возвращает ли их функция. Тут понятно, что не факт что в следующих версиях Windows подход может поменяться, поэтому просто проверять содержание упомянутой функции по одному шаблону

```
MOV EAX,DWORD PTR FS:[18]
MOV EAX,DWORD PTR DS:[EAX+30]
MOVZX EAX,BYTE PTR DS:[EAX+2]
RETN
```

не есть гуд, а вот общая спецификация формата PE – врятли (по крайней мере, так думают в SONY DADC). Особенно хочется отметить тот факт, что *IsDebuggerPresent* более всех остальных лидирует по числу вызовов, которые как правило, тянутся из VM на SEN-обработчиках UD2/hardware breakpoints. Противодействие обнаружению сводится к тому, что мой X-код первично сбрасывает флаг в PEв и больше его не трогает.

CheckRemoteDebuggerPresent/NtQueryInformationProcess. Если *IsDebuggerPresent* отвечает за обнаружение отладки текущего процесса, то новая рассматриваемая функция палит запущенный отладчик вне зависимости что отлаживается или не отлаживается. *CheckRemoteDebuggerPresent* собственно является оберткой (wrapper) низкоуровневой *NtQueryInformationProcess*, которая вызывается с аргументом *ProcessInfoClass* равным 7, числом которое скрывает за собой *ProcessDebugPort*. В более поздних версиях, введен *ProcessInfoClass* со значением 0x1F(31) – тот самый код ошибки 8019. Противодействие. Во-первых, взглянем на прототип *NtQueryInformationProcess* в ntdll:

```
MOV EAX,0A1
MOV EDX,7FFE0300
CALL DWORD PTR DS:[EDX]
RETN 14
```

Как и полагается, в NativeAPI имеет место __fastcall декларация вызова. Далее извлекается адрес KiFastSystemCall, который в зависимости от современности процессора переносит нас в ядро

через SYSENTER или INT2E. Собственно для нашего X-кода существует только один кул-хакерский вариант перехватить управление – поменять операнд 7FFE0300 на операнд с адресом нашего обработчика. Таким образом, первично управление получит наш обработчик, который сверит, не равен ли третий аргумент 7 или 0x1F? Если да, то в буфере и в EAX возвращаем ноль.

Имя файла процесса-родителя. И хотя мне до исследования SecuROM 7 эта замечательная идея также пришла в голову, не менее интересно было посмотреть ее реализацию в протекторе. Способ рассчитан, прежде всего, на неопытных исследователей, которые не имеют привычки переименовывать имя главного исполняемого файла отладчика. Конструктивно все просто: через уже знакомый NtQueryInformationProcess получить PID процесса-родителя. Понятное дело, что если наша игрушка была запущена в Ольге, то Ольга и является матушкой этого процесса. Как и положено, последний наследует все права своего родителя, но не в этом суть... Суть в том, что, как *ollydbg.exe* был пять лет назад, так и в 2011 *ollydbg.exe* и остался. Та же песня и с *idag.exe*, *winddbg.exe*...

Hardware Breakpoints(аппаратные точки останова). Замечательная вещь и при этом еще и дефицитная – всего четыре на каждый процесс. Точки проверяются 2 раза(все четыре). Процесс установки своих аппаратных точек выглядит следующим образом:

1. Управление получает инструкция UD2, на которой всегда происходит исключение (Illegal instruction)
2. SEH-обработчик для UD2 выглядит следующим образом:

```
PUSH EBP
MOV EBP,ESP
PUSHAD
MOV EAX,DWORD PTR SS:[EBP+8]
MOV EAX,DWORD PTR DS:[EAX]
MOV EAX,DWORD PTR DS:[12129F0]
MOV DWORD PTR DS:[12022A0],EAX
MOV DWORD PTR DS:[12022A4],26182AEF <- MOV DWORD PTR DS:[12022A4],26182AED (2й)
MOV DWORD PTR DS:[12022A8],EBP
PUSH OFFSET 012022A0
CALL 00D44F40
MOV DWORD PTR DS:[12022A0],0
MOV DWORD PTR DS:[12022A4],0
MOV DWORD PTR DS:[12022A8],0
POPAD
XOR EAX,EAX
MOV ESP,EBP
POP EBP
RETN
```

Их вообще два: **00C996B0h** и **00C99702h** – для первой 4ки HB и соответственно для второй.

3. До того как перейти в секуромовский SEH-обработчик, управление всегда берет на себя ntdll.KiUserExceptionDispatcher(pExeptionRecord, pContext). Второй аргумент – ссылка на структуру контекста. **CALL 00D44F40** ведет в VM, которая меняет в стеке структуру Context (DR0, DR1, DR2, DR3).
4. Фокус! Фокус! После выхода из обработчика выполняется **ZwContinue** на следующей от UD2 инструкции, с уже секуромовским контекстом, т.е. с его аппаратными точками останова.
5. Срабатывает (DR0-DR3) аппаратная точка установка.
6. SEH-обработчик(VM) проверяет контекст, адрес(в котором произошло исключения) и еще уложит в DR7 свое контрольное значение. Если что не так – Вам дадут знать! :)

Программные точки останова и перекрывающий код. Если аппаратным точкам останова было уделено пристальное внимание, но что говорить за программные. В первую очередь все начинается с упомянутого выше трюка:

```
MOV EAX,11D7B1C
MOV ESI,DWORD PTR DS:[EAX]
ADD DWORD PTR SS:[ESP+10],ESI ...
```

Чтобы многократно усложнить трассировку и обнаружение важных асм инструкций и процедур рассматриваемый код, можно сказать, перекрывает расстояние между ними. Из-за чего сидеть и жать F7, F8 или включать анимированную трассировку столь же бесполезно, как “горохом ап стенку”. Разработчики рассчитывали, что у реверсеров сдадут нервы, и они будут плодить программные точки останова, как кроликов - чтобы быстрее выйти из этой ловушки. В итоге контрольная сумма участка нарушается и вместо желаемого места назначения, протектор посылает незадачливых хакеров туда же, куда и Макар телят не гонял. Если такое большое желание трассировать, то бороться с этим можно. Во-первых, ловушка характерная своим циклом,PUSHFD/POPFD и следующим условным переходом. Поэтому вычислить, куда будет передано управление, минуя при этом цикл, несложно. Во-вторых, аппаратные точки останова проверяются в сумме всего 8 раз(2 по 4) в самом начале, так что аппаратка за нами.

Кстати в «Безопасном диске» aka SafeDisk в роли перекрывающего кода служат «лягушки» - разбросанные условные и безусловно условные переходы, работающие по одному и тому же пути. Безусловно условные переходы, это когда (самый распространенный вариант):

```
XOR EAX, EAX
```

```
JZ SOME_LABEL
```

Несмотря на то что опкод перехода не JMP, прыгать он на SOME_LABEL будет всегда! Происходит это потому, что результат операции «исключающего ИЛИ» над одним и тем же регистром всегда – НОЛЬ!(NULL, ZERO, 0, бублик). Соответственно флаг Z(Zero Flag) процессора будет взведен (равен единице) и твой процессор, повинувшись взведенному Zero Flag, выполнит условие перехода.

Address	Hex dump	Command	Comme
004031C3	31C0	XOR EAX,EAX	
004031C5	74 F9	JE SHORT 004031C0	
004031C7	90	NOP	

Jump is taken
Dest=SRP11.004031C0

Registers (FPU)

EAX 00000000
ECX 0012FFB0
EDX 7C8285EC ntdll.KiFastSystemCallR
EBX 7FFDE000
ESP 0012FFC4
EBP 0012FFF0
ESI 00000000
EDI 00000000
EIP 004031C5 SRP11.004031C5
C 0 ES 0023 32bit 0(FFFFFFFF)
P 1 CS 001B 32bit 0(FFFFFFFF)
A 0 SS 0023 32bit 0(FFFFFFFF)
Z 1 DS 0023 32bit 0(FFFFFFFF)
S 0 FS 003B 32bit 7FFDD000 (14000)
T 0 GS 0000 NULL
D 0
O 0 LastErr 0000051D ERROR_NO_IMPER
EFL 00000246 (NO,NB,E,BE,NS,PE,GE,LE

Проверка программных точек останова/переходов в начале WinAPI. Ставить программные точки останова в начале/конце процедуры вообще является стандарт де-факто, что тут говорить. Благо проверяются не все WinAPI, но в список попали CreateFileA , FindWindow. Противодействие. Самое очевидное – не ставить программные точки останова на первой инструкции. Необязательно даже и на вторую. Например, как стандарт CreateFileA формирует UNICODE строку и вызывает CreateFileW, значит можно поставить и на вызов CreateFileW. Другое дело с переходниками. Как ни крути, а ставить их удобно вначале, ведь в большинстве хукаемых WinAPI первые три инструкции(пролог):

MOV EDI, EDI

PUSH EBP

MOV EBP, ESP

Для противодействия X-коду потребуется таблица экспорта системной библиотеки и таблица импорта образа и тот факт, что компилятор для выравнивания функций по кратным адресам оставляет место между ними. В первом случае, сырое смещение нужно сдвинуть вверх на один байт, и при этом попадаем(во всяком случае должны попасть) на NOP, а во втором случае мы уже поправим RVA функции. Таким образом, минимальными переделками заставим протектор проверять NOP, а переход ниже останется в безопасности. Коль речь пошла о таблице экспорта, не лишним будет упомянуть, что на *GetProcAddress* иногда также полезно ставить свой обработчик!

CreateFileA/ FindWindow/OpenSCManager/RegOpenKey/ FindWindowExA/EnumWindows/GetClassName...

Думаю, что тут все понятно. Единственный нюанс с *FindWindowExA*. В отличие от *FindWindow*, у нее несколько расширенный список проверяемых существующих окон. В частности с помощью *TrayNotifyWnd*, извлекается хендл трей, а оттуда идет перечисление имен иконок. Ну а с помощью хендла окна *ToolBarWindow32*, устанавливается PID обозревателя Windows(explorer.exe).

FindFirstFileA. Довольно параноидальный способ защиты от X-кода: поиск ntdll*.dll в папке с cnc3game.dat. По-видимому, разработчики задумывали, что реверсер может подменить оригинальную ntdll.dll на свою, которая набита X-кодом.

GetModuleHandleA. С первого взгляда безобидная для реверсера функция, напомнила о том, что таит в себе возможность обнаружения библиотек, прикрепленных к процессу методом DLL-hijacking. Собственно SecuROM пытается искать в памяти процесса библиотеку *asr.dll*(AntiSecuROM?)

DbgUiRemoteBreakin. Поставляет замечательную возможность отловить отладчик по установке первичной точки останова. В SONY DADC решили не церемониться и специально для любителей приаттачится, установили безусловный переходник в ExitProcess библиотеки kernel32. Как противодействие, X-коду протектора(можно сказать так), наш X-код имеет право нанести ответный удар и снять переходник или не дать его установить вообще.

- Процедура проверки диска в приводе ☺

Настоятельно рекомендуется к прочтению: <http://www.insidepro.com/kk/020/020r.shtml>

Разбита на **две** части и **ОДИНАКОВА** в версиях 7.3x-8.0x

LEVEL 1 ***** ПРОВЕРКА СИГНАТУРЫ *****

GetDriveTypeA. Живая классика! Как бы намекает, что сейшн скоро начнется.

CMP EAX, 5 //DRIVE_CDROM

CreateFileA. Открывает заданный диск (Имя файла в виде: \\.\D), получает хендл.

DeviceIoControl (**4 ПАЗА**). Главную роль здесь отвели DeviceIoControl, с чьей помощью можно все что угодно сделать или получить с устройства. Прежде всего важен ControlCode, который может принимать три значения:

- 1) (IOCTL SCSI PASS THROUGH DIRECT, 4D014h или 4D004h. Первоочередной возможностью является информация о физическом наличии диска в устройстве. Затем идет различная информация о дорожках, секторах и тд. Размер буфера всегда равен 50h(80байт).

CMP BYTE PTR SS:[EBP-52],BL //Вставлен ли диск в лоток?

Первая проверка только для подтверждения наличия диска в приводе. Если все ОК, то начинаем процедуру чтения и составления сигнатуры.

На данном этапе, при выполнении `WinAPI DeviceIoControl`, обращайте внимание на `InBuffer(ESP+8)`: от начала `InBuffer` по смещению `$+14h` расположен указатель на структуру спец. буфера, куда, после выполнения вызова, ложатся служебные данные (например, номер сектора).

А теперь... **три** разводные проверки, для прохождения первого левела:

Address	Hex dump	Command	Comments
01006FC7	8BC8	MOV ECX,EAX	
01006FC9	3F60F7FF	CALL SOME_CPP_COMPARE_FUNC	CHECK SIGNATURES ?
01006FCE	83E0 1F	AND EAX,0000001F	AND ?
01006FD1	3C 1F	CMP AL,1F	CMP FOR LENGHT?
01006FD3	9C	PUSHFD	
01006FD4	9C	PUSHFD	
01006FD5	83EC 24	SUB ESP,24	
01006FD8	C74424 20 BB	MOV DWORD PTR SS:[ESP+20],725C85BB	emulate(!) check software breakpoints. but ELF known...
01006FE0	C74424 1C 44	MOV DWORD PTR SS:[ESP+1C],44	
01006FE8	895424 18	MOV DWORD PTR SS:[ESP+18],EDX	
01006FEC	BA 546F0B01	MOV EDX,01006F54	
01006FF1	C14C24 20 00	ROR DWORD PTR SS:[ESP+20],0	Shift out of range
01006FF6	90	NOP	
01006FF7	897424 14	MOV DWORD PTR SS:[ESP+14],ESI	
01006FFB	0FACD2 00	SHRD EDX,EDX,0	Shift out of range
01006FFF	8B32	MOV ESI,DWORD PTR DS:[EDX]	
01007001	87C9	XCHG ECX,ECX	
01007003	017424 20	ADD DWORD PTR SS:[ESP+20],ESI	
01007007	83C2 04	ADD EDX,4	
0100700A	66:FF4C24 1C	DEC WORD PTR SS:[ESP+1C]	
0100700F	75 EA	JNE SHORT 01006FFB	
01007011	0B4C24 20 01	OR BYTE PTR SS:[ESP+20],01	
01007016	8B5424 24	MOV EDX,DWORD PTR SS:[ESP+24]	
0100701A	8B7424 20	MOV ESI,DWORD PTR SS:[ESP+20]	
0100701E	C1E1 00	SHL ECX,0	Shift out of range
01007021	895424 20	MOV DWORD PTR SS:[ESP+20],EDX	
01007025	90	NOP	
01007026	8B5424 18	MOV EDX,DWORD PTR SS:[ESP+18]	
0100702A	90	NOP	
0100702B	897424 24	MOV DWORD PTR SS:[ESP+24],ESI	
0100702F	8B7424 14	MOV ESI,DWORD PTR SS:[ESP+14]	
01007033	83C4 20	ADD ESP,20	
01007036	90	POPFD	
01007037	90	NOP	
01007038	75 17	JNE SHORT 01007051	here ? ;)
0100869E	EB F3	JMP SHORT 01008693	
010086A0	9D	POPF	
010086A1	80BF F2070000	CMP BYTE PTR DS:[EDI+7F2],0	
010086A8	9C	PUSHFD	
010086A9	68 7F2D0000	PUSH 2D7F	
010086AE	75 17	JNE SHORT 010086C7	// 4
010086B0	810424 105900	ADD DWORD PTR SS:[ESP],750B5910	
010086B7	C1E0 00	SHL EAX,0	Shift
010086BA	810424 C20000	ADD DWORD PTR SS:[ESP],8C0000C2	
010086C1	C1E6 00	SHL ESI,0	Shift
010086C4	EB F7	JMP SHORT 010086BD	
010086C6	25 83EC1CC7	AND EAX,C71CEC83	
0100A615	F9	STC	
0100A618	83D3 00	ADC EBX,0	
0100A61E	98 C0070000	TEST BYTE PTR DS:[EAX+7C0],BL	
0100A614	995D FC	MOV DWORD PTR SS:[EBP-4],EBX	
0100A617	B9 37F6FEFF	MOV ECX,FFFFFF637	
0100A61C	8D8C29 850900	LEA ECX,[EBP+ECX+10985]	
0100A613	9C	PUSHFD	
0100A614	68 053B0000	PUSH 3B05	
0100A619	74 15	JE SHORT 0100A619	// 2
0100A61B	810424 E72500	ADD DWORD PTR SS:[ESP],420A25E7	
0100A618	90	NOP	
0100A613	810424 C20000	ADD DWORD PTR SS:[ESP],8F0000C2	
0100A610	8BD2	MOV EDX,EDX	
0100A61C	EB F8	JMP SHORT 0100A618	

Правильное прохождение первой из проверок(и последующих), заставляет SecuROM 7-8 генерировать сообщение с повтором: *Cannot authenticate the original disc. Your disc may require a different software version.* Правда, есть еще 4 проверка, но её патчинг не влияет конечный на результат-*Level 1 completed!*

010D2AB7	A3 8B83E00F	MOV DWORD PTR DS:[0FE0838B],EAX
010D2ABC	3C 0F	CMP AL,0F
010D2ABE	8B0C24	MOV ECX,DWORD PTR SS:[ESP]
010D2AC1	8D6424 04	LEA ESP,[ESP+4]

ВАЖНО! Патчить нужно непосредственно операции, влияющие на флаги, а не сами переходы! `PUSHFD` защита сохраняет состояние `EFL` для следующих переходов (в 7.33.017 их не более 2х), т.е. условие проверяется несколько раз.

```

01006FCE 83E0 1F AND EAX,0000001F
01006FD1 3C 1F CMP AL,1F //AL должно быть равным 0x1F(31) или кратным этому числу
01006FD3 9C PUSHFD

01007036 90 POPFD
01007037 90 NOP
01007038 75 17 JNE SHORT 01007051 // ПЕРЕХОД НЕ ВЫПОЛНЯЕТСЯ
*** 2 ***

010A615E 8498 C0070000 TEST BYTE PTR DS:[EAX+7C0],BL //BL = 1. видно, что структура. [EAX+7C0] = 9 (прокатывало с любым != 1)

01007038 75 17 JNE SHORT 01007051 //ПЕРЕХОД НЕ ВЫПОЛНЯЕТСЯ
*** 3 ***

010086A1 80BF F2070000 CMP BYTE PTR DS:[EDI+7F2],0 //[EDI+7F2] = 1 (также прокатывало с другими числами)
010086A8 9C PUSHFD

010086AE 75 17 JNE SHORT 010086C7 //ПЕРЕХОД ВЫПОЛНЯЕТСЯ

```

ДВЕ ПОСЛЕДНИЕ ПРОВЕРКИ (ПО КРАЙНЕЙ МЕРЕ, 2) НАПРЯМУЮ СВЯЗАНЫ С **LEVEL 2** И СКОРЕЕ ВСЕГО ЗАДАЮТ ГРАНИЦЫ ИНТЕРВАЛА ПОПАДАНИЯ ВИРТУАЛЬНЫЙ ВАСКУР-НАСТОЯЩИЙ ФИЗИЧЕСКИЙ (или ЦИКЛА, КОЛ-ВО ИТЕРАЦИЙ).

Кстати, характерная деталь в этой версии для первой сигнатурной проверки – «липовый» подсчет контрольной суммы участка в перекрывающем коде.

QueryPerfomanceCounter & QueryPerfomanceFrequency. Начинают считать такты в нормальном (подготовительном режиме).

LEVEL 2 ***** ПРОВЕРКА ГЕОМЕТРИИ(ТАКТОВ) ИЛИ QPC/QPF *****

SetSystemCursor.



Тут все сложнее и на момент написания этих строк, очень зависело от везения (Вся проблема в базе с подсчетом интервала). Самое главное – прокатывало (И НЕ РАЗ!!!): запуск с абсолютно левым диском в виртуальном приводе (установщик Windows 7). ;)

После того, как на 7м с липовым виртуальным диском все прошло нормально, решил попробовать на 8м без Daemon Tools с обычным физическим приводом и диском. После успешного прохода сигнатурного уровня процесс всегда стал завершаться с кодом выхода 1. Сначала думал, что это защита. Но потом вспомнил за HKEY_CURRENT_USER\Software\SecuROM. Удалил ветку Key – процесс стал завершаться теперь после прохода геометрической проверки (в Bin таблицы, структуры данных проверок по времени и геометрии, метка диска под XOR и что-то еще). Короче «фиским»(010DF7A9) как показано на картинке ниже (signed int > 0). Если очень интересно покопаться проверку – ставим брекпоинт по доступу.

010DF7A6	897D 64	MOV DWORD PTR SS:[EBP+64],EDI	
010DF7A9	8B3D BF1D2D01 00	CMP BYTE PTR DS:[12D1DBF],0	-- 2fix
010DF7B0	9C	PUSHFD	
010DF7B1	68 AF060000	PUSH 6AF	
010DF7B6	76 15	JBE SHORT 010DF7CD	not taken
010DF7B8	810424 F70F3300	ADD DWORD PTR SS:[ESP],330FF7	
010DF7BF	C1E1 00	SHL ECX,0	Shift out
010DF7C2	810424 C3E1DA00	ADD DWORD PTR SS:[ESP],00DAE1C3	
010DF7C9	EB FA	JMP SHORT 010DF7C5	

Все что связано со вторым уровнем - связано с FPU операциями.

Если не будет пройдена одна из проверок – процесс завершится или выпадет исключение.

Между проверками и после 3й выполняется вход в VM на специально-отведенные островки.

Первая разводная проверка второго геометрического уровня

```
010EAFB2 FLD QWORD PTR DS:[11EDC78]
010EAFB8 MOV ESI,OFFSET 012D1E30
010EAFBD FSUB QWORD PTR DS:[11EDC80]
010EAFCD FMUL QWORD PTR DS:[14AC050]
010EAFD1 FCOMP QWORD PTR DS:[1483098]
010EAFD4 FSTSW AX //AH == 0
010EAFD1 TEST AH,05
010EAFD4 PUSHFD
010EAFD5 PUSH 17F7
010EAFDA JPE SHORT 010EAFD0 ; NOT TAKEN
```

Вторая разводная проверка второго геометрического уровня

```
010ED310 FLD QWORD PTR DS:[11EDC78]
010ED316 FSUB QWORD PTR DS:[11EDC80]
010ED31C FMUL QWORD PTR DS:[14A4E64]
010ED322 FCOMP QWORD PTR DS:[1483098]
010ED328 FSTSW AX
010ED32A TEST AH,05 //AH == 1
010ED32D PUSHFD
010ED32E PUSH 24F2
010ED333 JPO SHORT 010ED34A ; TAKEN !
```

Третья разводная дублируемая проверка второго геометрического уровня

```
010CD164 FLD QWORD PTR DS:[11EDC78]
010CD16A FSUB QWORD PTR DS:[11EDC80]
010CD170 FMUL QWORD PTR DS:[14A4E64]
010CD176 FCOMP QWORD PTR DS:[1483098]
010CD17C FSTSW AX
010CD17E TEST AH,05
010CD181 PUSHFD
010CD182 PUSH 14D
010CD187 MOV ESI,ESI
010CD189 JPE SHORT 010CD1A0
```


Исправление ошибки «Conflict with Emulation Software detected»:

при запросе *DeviceIoControl* с макросом *IOCTL_DISK_PERFORMANCE*(0x00070020) из второстепенных потоков(параллельная проверка геометрии), необходимо вернуть 0(ноль) в *EAX*. Иногда помогает установка(*SetAffinityMask*) минимального кол-ва разрешенных процессоров (всего одного) для работы target-процесса.

Существует также специальная функция, которая запрашивает коды выхода потоков:

```
010F146C PUSH EBP
010F146D MOV EBP,ESP
010F146F PUSH EBX
010F1470 PUSH ESI
010F1471 MOV ESI,DWORD PTR SS:[ARG.1]
010F1474 XOR EBX,EBX
010F1476 CMP BYTE PTR DS:[ESI],BL
010F1478 PUSH EDI
010F1479 JE SHORT 010F14C7
010F147B CMP BYTE PTR DS:[ESI+1],BL
010F147E JE SHORT 010F14C7
010F1480 PUSH EBX ; /Timeout => 0
010F1481 PUSH 1 ; |WaitAll = TRUE
010F1483 LEA EDI,[ESI+228] ; |
010F1489 PUSH EDI ; |HandleList
010F148A PUSH DWORD PTR DS:[ESI+4] ; |Count
010F148D MOV DWORD PTR SS:[ARG.1],EBX ; |
010F1490 CALL DWORD PTR DS:[<&KERNEL32.WaitForMul ; \KERNEL32.WaitForMultipleObjects
010F1496 TEST EAX,EAX
010F1498 JE SHORT 010F149E
010F149A PUSH 5
010F149C JMP SHORT 010F14C9
010F149E CMP DWORD PTR DS:[ESI+4],EBX
010F14A1 JLE SHORT 010F14BE
010F14A3 LEA EAX,[ARG.1]
010F14A6 PUSH EAX ; /pExitCode => OFFSET ARG.1
010F14A7 PUSH DWORD PTR DS:[EDI] ; |hThread
010F14A9 CALL DWORD PTR DS:[<&KERNEL32.GetExitCo ; \KERNEL32.GetExitCodeThread
010F14AF CMP DWORD PTR SS:[ARG.1],1
010F14B3 JNE SHORT 010F14C3
010F14B5 INC EBX
010F14B6 ADD EDI,4
010F14B9 CMP EBX,DWORD PTR DS:[ESI+4]
010F14BC JL SHORT 010F14A3
010F14BE XOR EAX,EAX
010F14C0 INC EAX
010F14C1 JMP SHORT 010F14CA
010F14C3 PUSH 6
010F14C5 JMP SHORT 010F14C9
010F14C7 PUSH 7
010F14C9 POP EAX
010F14CA POP EDI
010F14CB POP ESI
010F14CC POP EBX
010F14CD POP EBP
010F14CE RETN
```

Относительно файлов, вложенных в протектор и создаваемых во временной папке. Хотелось бы напомнить, что доступ во временную папку может быть закрыт, поэтому функции, расположенные в *drm_dyndata_7330017.dll*, могут быть дублированы в образе.

Ну а сейчас - виртуальная машина! Встречаем!

- **Paul.dll (паша.dll)**

Paul можно не патчить. Достаточно подделать результаты. Раньше в секуре были константы ответа пауля. В стиле 21- не прошли проверку 1- прошли. Или наоборот. Еще было что-то типа списка адресов фунок секура в памяти(типа IAT). Так вот в этой IAT секура были адреса типа "проверка не пройдена" и "пройдена проверка онлайн активации". Тупо меняя 3 dword местами можно было всегда идти по нужной ветке. Как сейчас - хз.

Nightshade

ВИРТУАЛЬНАЯ МАТЕМАТИКА

Может следующее утверждение кажется нелепым, но, в самом деле, вскрыть так называемую «виртуальную машину SecuROM 7» достаточно просто! Сложнее довести до читателя как это сделать и уточнить все детали. Но, дорогу осилит идущий!

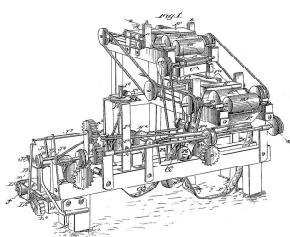


ОСНОВНОЕ "ГЛАВНОЕ" ХРАНИЛИЩЕ VM (СПЕЦИФИКАЦИЯ)

+0 ВХОД В "ХРАНИЛИЩЕ №1.5" - ОПЕРАНДЫ JMP EAX/EDI/...
+4 АДРЕС В 1М АРГУМЕНТЕ ИЗ "БАЗЫ ЗАПРОСОВ"
+8 ФЛАГИ СОСТОЯНИЯ (EFL)
+с ВХОД В ВИРТУАЛЬНУЮ МАШИНУ
+10 ROI-БАЙТ
+14 ВХОД В "ГЛАВНОЕ ХРАНИЛИЩЕ"
+18 ОБВЕРТКА ВИРТУАЛЬНОГО СТЕКА
+1с ВЕРХУШКА СТЕКА (STACK POINTER)
+20 ВХОД В "ХРАНИЛИЩЕ .artem"
+24 ВЗВЕДЕННЫЙ LOCK BYTE
+28 ВХОД В "ХРАНИЛИЩЕ №2"
>2с ИНОГДА ДЛЯ ХРАНЕНИЯ ПРОМЕЖУТОЧНЫХ ДАННЫХ
ИСПОЛЬЗУЮТ "ОСТРОВКИ" ВО ВРЕМЯ РАБОТЫ

Ключом, для понимания работы “виртуальной машины” (далее по тексту просто VM) является обычная структура, именуемая мною как “главное хранилище” (смещение которого от входа в VM чаще всего равно +2000h. В рассматриваемом случае оно равно 20A7000h, вход в VM - 20A5000h соответственно. Это спираль, возле которой танцует основная логика работы всех “островков” или более объективно – 255 кусков кода, каждый из которых работает по единому шаблону с элементами главного хранилища (+0, +4, +10, 14) и выполняет только одну свою задачу (общее число задач равно четырем, плюс несколько специфических). Очевидно, что островки, хотя по внешнему виду разные, но по факту – дублируются. Применительно к способам № 1 и 1A построенная цепь островков красиво укладывается в одну главную и простую цель – поставить по заданному адресу заданное значение/адрес! Все! Естественно эти два главных DWORD зашифрованы и находятся в специальной области памяти(виртуальном стеке) и в нужный момент их расшифрует нужный островок, который несет алгоритм декодирования, затем другой островок выполнит подмену. Способ №2 идет немного дальше, но реально его можно заменить одной асм инструкцией AND ECX, [NOT_ключ]. Он подогнан под специальный алгоритм прямого/обратного хода. Функция, которая его вызывает, принимает два аргумента, из первого в котором указан адрес, копируются ключи в стек для алгоритма, второй аргумент служит, как входное число, из которого по ключам будет получено требуемое. Тут все завязано на Odd-Even Based Cryptography(четный-нечетный). Переходим к более углубленному разбору.

Как найти VM?



JMP SHORT \$+

Одна из ASCII-Z строк (приведены самые распространенные):

«<space for rent>»

«You are now a restricted area»

«Nobody move, nobody gets hurt»

Для этой главы используйте приложение к статье:

Материалы по работе виртуальной машины SecuRom версии 7.33.0017

Следующие операции на любом островке всегда делаются по умолчанию:

1) Чтение ресурса (+4), операция сложения результата с ресурсом (+C), в итоге получение текущего виртуального указателя ESP на виртуальный стек.

2) Чтение (вытаскивание по аналогии с POP EDI, см. пункт 5) по виртуальному указателю:

А) одного двойного (контрольного) слова, если задача островка не связана с расшифровкой

Б) двух двойных слов. Первое требуется расшифровать, второе – контрольное.
3) Чтение ресурса (+10) – контрольного байта. Если островок действовал по 2-Б, то контрольным байтом будет выполнен первый этап расшифровки, второй этап – маска XOR с одним и тем же ключом. Далее из контрольного слова будет взят, специальный байт (или 3 байта), который будет прибавлен (в дополнение возможно: XOR с определенным ключом, вычитание, сложение) к текущему ROL-байту, новое значение будет сохранено в ресурсе (+10) для следующего островка.

4) Получение из контрольного слова байта, операция умножения в 4 раза, полученное смещение складывается с адресом хранилища №1.5, оттуда читается двойное слово, над которым будет произведен побитовый сдвиг направо ROR, (операция эквивалентна делению) затем сложение с адресом входа в VM. Так получается адрес следующего островка или выхода из VM. Этот алгоритм стабилен для всех.

5) Как следствие пункта 2, естественно требуется поправить виртуальный указатель ESP, к ресурсу (+4) прибавляется:

А) 4 байта, если было использовано (вытащено) одно двойное слово

Б) 8 байт, если было использовано (вытащено) два двойных слова

В) *Исключение:* 10(1) байт. Связано с некоторым различием работы VM в SEN-обработчиках самой защиты и VM непосредственно в “тибериумной драке”. Чтобы запутать взломщиков, в первом случае изначально виртуальный стек специально делается “кривым”(не кратен 4) и примерно с десятков “островков” не несут в себе полезной нагрузки. После их выполнения все станет на свои места.

По инструкции `ADD DWORD PTR DS:[(+4)], 8` и `ADD DWORD PTR DS:[(+4)], 4` легко идентифицировать тип островка!

6) Собственно прыжок на следующий островок `JMP EAX(JMP EDI/RET)` или выход из VM.

Пояснения к вышесказанному:

Как извлекаются смещения из контрольного слова(перед декодированием и последующим переходом на запрашиваемый островок). В примере, контрольное слово = 889AFC25; ROL-байт = D7h:

`SHL EDX, 30 ; EDX = FC250000`

`SHR EDX, 18 ; EDX = 000000FC` //получили нужный контрольный байт (2й справа)

`ADD DL, CL ; DL = FC ; CL = D7 ; => DL = D3` //ROL-байт переделает оригинальное значение!

`SHL EDX, 2 ; EDX = 000000D3 * 4 = 0000034C` /* увеличение результата в 4 раза!

(`SHL REG_32, 2`) Единственное арифметическое действие, без участия ROL-байта, производимое над результатом. Так мы получаем смещение для Хранилища 1.5 и ячеек в главном! */

`MOV EAX, DWORD PTR DS:[(+0)] ; EAX = FFF2C000` //извлекаем ресурс (+0)

`ADD EAX, DWORD PTR DS:[(+C)] ; EAX = FFF2C000 + 020A5000 = 01FD1000` //формируем виртуальный адрес Хранилища №1.5

`ADD EAX, EDX ; EAX = 01FD1000 + 0000034C = 01FD134C` //Полученное смещение складываем с ним

`MOV EDX, DWORD PTR DS:[EAX] ; EDX = C003FFF0` //читаем закодированное смещение следующего островка

Последние пять инструкций имеют одну эквивалентную инструкцию, которая также используется на некоторых островках в VM

(`EDX-ресурс(+0)` или адрес Хранилища №1.5):

`PUSH DWORD PTR DS:[EAX*4+EDX]`

Декодирование полученного смещения из Хранилища №1.5. Типичная операция в конце островка – переход на следующий. Допустим из Хранилища №1.5 было извлечено закодированное смещение C003FFF7(регистр EDI) и CPUID в AL возвратил 92h(для ROR операнд эквивалентен делению на F0h):

`MOV EAX, 1` //EAX = 1 для инструкции CPUID говорит о том что инструкция возвратит так называемую *сигнатуру CPU* – информацию о процессоре(модель, степпинг)

`CPUID` //кодирование/декодирование смещений в таблице Хранилища №1.5

осуществляется при помощи этой асм инструкции!

`AND EAX, FFFFFFFF` //получаем байт для сдвига через CPUID

`ROR EDI, CL ; EDI = C003FFF7/ F0 = FFFDF000` //ДЕКОДИРУЕМ

```
ADD EDI, DWORD PTR DS:[ (+C) ] ; EDI = FFFDF000 + 020A5000 = 02084000
/*ПОЛУЧЕННОЕ СМЕЩЕНИЕ СКЛАДЫВАЕМ С АДРЕСОМ ВХОДА В VM*/
JMP EDI ; EDI = 02084000 //ПРЫГАЕМ НА СЛЕДУЮЩИЙ ОСТРОВ
```

Для того чтобы положить свой адрес в №1.5 требуется выполнить обратные действия:

1. Вычислить дельту (RVA входа в VM минус RVA начала твоего кода)
2. Выполнить:

```
MOV EAX, 1
CPUID
AND EAX, FFFFFFFDF
ROL EDI, CL
```

Примечание: Короткий список допустимых значений регистра EAX перед вызовом CPUID.

Сигнатура ЦП (EAX = 1)

Возвращаемые в регистре EAX биты после выполнения CPUID.

3:0 – Stepping (степпинг процессора)

7:4 – Model (Модель)

11:8 – Family (Семья)

13:12 – Processor Type (тип процессора)

19:16 – Extended Model (расширенная модель)

27:20 – Extended Family (расширенная семья)

У AMD и Intel существуют различные нюансы в значениях, которые возвращают биты.

Например, для процессора AMD Phenom II X4 940 возвращаемое в EAX значения будет равно 00100F42. Соответственно 42h идет на кодирование/декодирование таблицы смещений Хранилища №1.5.

Другие значения EAX для выполнения CPUID.

EAX = 0 (ID поставщика ЦП)

В регистрах EBX, EDX, ECX – первые 12 байт от ASCII строки, которая идентифицирует фирму изготовителя. Например, AuthenticAMD говорит о том, что поставщиком процессора является компания Advanced Micro Devices (AMD).

EAX = 2 (Кэш и информация о TLB дескрипторе)

TLB – Translation Lookaside Buffer. В двух словах, этот буфер является КЭШем ЦП для увеличения скорости трансляции виртуальных адресов.

EAX = 3 (Серийный номер ЦП)

В зависимости от модели и производителя процессора в EDX:ECX (или EBX:EAX) возвращается серийный номер.

Виртуальный стек SecuROM 7 VM хранит в себе два типа данных:

1) Контрольное слово (Control DWORD). Манипулирует логикой работы VM.

Имеет в себе 4 байта из которых:

- Следующее значение ROL-байта для следующего островка. Всегда присутствует. Как стандарт 1 байт. Исключение: 2 байта (из них “клеится” один байт, который аналогично будет прибавлен к ресурсу (+10)).

- Адрес следующего островка по Хранилищу №1.5 [1 байт] Всегда присутствует. 1 байт. По существу является закодированным смещением.

- Адрес ячейки в главном хранилище. В зависимости от функции текущего островка. По существу является закодированным смещением.

2) Зашифрованные данные в размере одного DWORD. Такие как: Адрес запрашиваемой функции, адрес ASCIIZ строки, число. Этот тип используют непосредственно островки, занимающиеся их расшифровкой (с помощью ROL-байта).

СПОСОБ №1 – ВЫЗОВ ВНУТРЕННЕЙ ФУНКЦИИ(015110F0), АДРЕС ЛЕЖИТ В ВИРТУАЛЬНОМ СТЕКЕ	
0040A006 PUSH 58 0040A008 PUSH OFFSET 00B68A40 0040A00D CALL 0040A370 //вызов внутренней функции	Основной код программы
0040A370 JMP DWORD PTR DS:[15000C4]	“Стрелка”. Первый раз - Переход к базе запроса. В процессе работы VM будет заменен на 015110F0 , т.е. после первого вызова мы сразу попадаем в требуемую функцию
0157C830 PUSH OFFSET 0157C84A [0155741C] 0157C835 PUSH 0040365A 0157C83A PUSH OFFSET 00D611CC 0157C83F PUSHFD //мусор 0157C840 SUB DWORD PTR SS:[ESP+4],1C28C 0157C848 POPFD //мусор 0157C849 RETN //00D611CC-1C28C=00D44F40 0157C84A 1C 74 // 0155741C 0157C84C 55 0157C84D 0100	2й аргумент: LPDWORD = 0157C84A По адресу чуть ниже начинается первое DWORD из виртуального стека (0155741C) 1й аргумент: (VOID) 0040365A Не имеет смысла для способа №1, т.к. будет замен на адрес требуемой процедуры ближе к выходу из VM
00D44F40 JMP DWORD PTR DS:[12E043C]	Блокпост. Перебрасывает в VM .
JMP SHORT \$+14 <space for rent> PUSHAD PUSHFD CALL \$+5 CALL \$+7 Lock byte +1 byte POP EDX //lock byte offset LOCK DEC BYTE PTR DS:[EDX] JNS SHORT START_VM CMP BYTE PTR DS:[EDX],0 PAUSE JLE SHORT CMP JMP SHORT LOCK START_VM: MOV EBX,EDI MOV ECX,100 MOV EAX,0 REP STOS DWORD PTR ES:[EDI] //ОЧИСТКА ГЛАВНОГО ХРАНИЛИЩА ОТ ПРЕДЫДУЩЕГО ВЫЗОВА... // ...И ЕГО НОВАЯ ИНИЦИАЛИЗАЦИЯ MOV DWORD PTR DS:[EBX],EAX POP EAX MOV DWORD PTR DS:[EBX+4],EAX MOV EAX,DWORD PTR SS:[ESP] MOV DWORD PTR DS:[EBX+8],EAX MOV DWORD PTR DS:[EBX+0C],EDX MOV BYTE PTR DS:[EBX+10],95 //CRYPT-BYTE - 95h MOV DWORD PTR DS:[EBX+14],EBX	Spin-блокировка Space for rent – ASCII строка Первый островок

<pre> MOV DWORD PTR DS:[EBX+1C],ESP MOV EAX,OFFSET 01212A10 MOV DWORD PTR DS:[EBX+20],EAX MOV EAX,OFFSET 020A6000 MOV DWORD PTR DS:[EBX+28],EAX ... JMP EAX MOV EAX,1 //ВТОРОЙ ОСТРОВOK- por SHL EAX,2 ADD EAX,EBX ... </pre>	
<pre> MOV EBX, [V-stack pointer] ADD EBX, 4*количество DWORD до зашифрованного адреса MOV EAX, [EBX] //EAX = 243BBBD6 XOR EAX, 43E2AB9D //EAX = 15000C4 MOV [ЯЧЕЙКА В ОБЛАСТИ ХРАНЕНИЯ ДАННЫХ_1], EAX ADD EBX, 4*количество DWORD до след. зашифрованного адреса MOV EAX, [EBX] //EAX = 42B3BB6D XOR EAX, 43E2AB9D //EAX = 015110F0 MOV [ЯЧЕЙКА В ОБЛАСТИ ХРАНЕНИЯ ДАННЫХ_2], EAX MOV EAX, [ЯЧЕЙКА В ОБЛАСТИ ХРАНЕНИЯ ДАННЫХ_1] MOV EDX, [ЯЧЕЙКА В ОБЛАСТИ ХРАНЕНИЯ ДАННЫХ_2] MOV [EAX], EDX // ПЕРЕВОДИМ СТРЕЛКИ !!! MOV [ARG. 1], EDX //тот что VOID JMP ARG. 1 //переходим в запрашиваемую внутреннюю процедуру </pre>	<p>Что, по сути, происходит в общем виде</p> <p>V-stack pointer – (+1C)в главном хранилище</p> <p>43E2AB9D – ключ для расшифровки</p>

Сама VM имеет три режима работы - по количеству выходов из нее. А мы, тем временем, начинаем знакомиться с вариантами работы VM: Способ №1 – когда требуется вызвать внутреннюю функцию(к слову, это может быть и одна асм команда) в самой защищаемой программе; способ №1А – практически тоже самое, что и первый, однако его обращение к VM характерно прямым вызовом через регистр EAX(CALL EAX), который у себя формирует базу запроса и сразу переходит в VM (т.е. CALL 00D44F40), после конечного RET управление передается в запрашиваемую WinAPI и способ №2 – радикально отличается от первых двух, результаты его работы в регистрах EAX и ECX. База запросов. Самый главный первый аргумент, который является простой “обверткой” и указывает на начало области в секции .secuROM, с которой будет работать VM. Для понимания представьте ее в качестве “виртуального стека”, а ресурс (+4) в “главном хранилище” виртуальным аналогом регистра ESP, который в рассматриваемом примере первоначально устанавливается равным адресу 0155741C. Второй аргумент является адресом возврата, на который передаст управление последний RET в VM, для первых двух способов его заданное значение не используется (адрес 0040365A попадает в середину команды) и будет исправлено VM в стеке на адрес конечной запрашиваемой функций, обратная ситуация для способа №2, адрес возврата будет чуть ниже базы запросов и конечный RET в виртуальной машине передаст управление на него. Из базы запросов работает переход на адрес 00D44F40, своеобразный блокпост VM, который всегда делает прыжок на начало VM. Даже не смотря на то, что существуют его адреса-аналоги, все базы запроса указывают на него и все через него проходят в VM. Начало VM, разбираем суть работы. Первый островок (его я специально оставил как есть, с остальных обфускация снята.). Первым делом обфускация, суть ее проста: математическими операциями получать смещения в главном хранилище (как стандарт). Например:

```

MOV EDX,60
SHR EDX,3 ; EDX = 0Ch //ресурс (+C)

```

```

MOV ECX,72
XOR ECX,00000062; ECX = 10h //ресурс (+10)

```

Впрочем, такая простота присуща не всем островкам, на некоторых есть длинный запутанный алгоритм по получению смещения +4. После получения смещений и выполнения операций в главном хранилище, SecuROM возвращает указатель на его начало, вычитая использованное смещение. Стоит отметить, что чаще всего в обфускацию вовлечены два регистра EDX и ECX, предварительно их значения сохраняются в стек. Мусорных команд мало, поэтому, зная неизменную логику работы всех островков, можно спокойно пить чай и быстро снять всю обфускацию кода.

0040A006	> 6A 58	PUSH 58
0040A008	68 408AB600	PUSH OFFSET 00B68A40
0040A00D	E8 5E030000	CALL 0040A370

0040A370	\$ FF25 C400500	JMP DWORD PTR DS:[15000C4]	
0040A376	EB 98	JMP SHORT <JMP.&MSUCR80.sprintf>	Jump to MSUCR80.sprintf

0157C830	68 4AC85701	PUSH OFFSET 0157C84A	
0157C835	68 5A364000	PUSH 0040365A	
0157C83A	68 CC11D600	PUSH OFFSET 00D611CC	
0157C83F	9C	PUSHFD	
0157C840	816C24 04 8CC2	SUB DWORD PTR SS:[ESP+4],1C28C	
0157C848	9D	POPFD	
0157C849	C3	RETN	
0157C84A	1C 74	SBB AL,74	
0157C84C	55	PUSH EBP	

00D44F40	FF25 3C042E01	JMP DWORD PTR DS:[12E043C]	
00D44F46	EB	PUSH EBX	

020A5000	EB 12	JMP SHORT 020A5014	
020A5002	3C 20	CMP AL,20	
020A5004	73 70	JNB SHORT 020A5076	
020A5006	61	POPAD	
020A5007	6365 20	ARPL WORD PTR SS:[EBP+20],SP	
020A500A	66:6F	OUTS DX,WORD PTR DS:[ESI]	
020A500C	72 20	JB SHORT 020A502E	
020A500E	72 65	JB SHORT 020A5075	
020A5010	6E	OUTS DX,BYTE PTR DS:[ESI]	
020A5011	74 20	JE SHORT 020A5033	
020A5013	3E:60	PUSHAD	
020A5015	9C	PUSHFD	
020A5016	E8 00000000	CALL 020A501B	
020A5018	E8 02000000	CALL 020A5022	
020A5020	0101	ADD DWORD PTR DS:[ECX],EAX	
020A5022	5A	POP EDX	
020A5023	F0:FE0A	LOCK DEC BYTE PTR DS:[EDX]	

Edit data at address 020A5002

ASCII

< space for rent >

UNICODE

□ □ □ □ □ □ □ ?

HEX+00

3C 20 73 70 61 63 65 20 66 6F 72 20 72 65 6E 74
20 3E 60

OVR

☐ Keep size

OK

Cancel

015110F0	68 70EA4500	PUSH 0045EA70	
015110F5	64:FF35 000000	PUSH DWORD PTR FS:[0]	
015110FC	8B4424 10	MOV EAX,DWORD PTR SS:[ESP+10]	
01511100	896C24 10	MOV DWORD PTR SS:[ESP+10],EBP	
01511104	8D6C24 10	LEA EBP,[ESP+10]	
01511108	20FA	SUB ESP,EAX	

Собственно переходим к разбору функционала. Стандартные операции сохранения регистров и флагов, взвод spin-блокировки, говорит о том, что с VM имеет право работать только один поток, остальные будут ожидать сброса блокировки, которая произойдет на выходе из VM. Теперь требуется инициализировать главное хранилище, предварительно обнулив его содержимое. Обратите внимание, что наш виртуальный ESP или ресурс (+4) будет хранить указатель на виртуальный стек в неявном виде, об этом говорит команда по смещению ACh от входа VM, которая вычитет из адреса-указателя адрес входа в VM (точнее говоря, хранится его смещение). Соответственно, чтобы прочитать контрольное слово по виртуальному ESP, машинке требуется обратно прибавить адрес входа в VM, который будет также помещен в главное хранилище в ресурсе (+C). В дополнение к (+4), по смещению +18 лежит "обертка" виртуального стека, по которой всегда можно установить его начало. После ресурса (+4), еще больше внимания заслуживает ресурс (+10) и непосредственно DWORDs в виртуальном стеке.

По первому, можно увидеть, что первоначально его значение всегда равно 95h. Почему 95h? На самом деле неважно сколько, важно соблюдение правил всей системы. Представьте, что у вас есть два очень простых уравнения:

$$X+Y = 5 \quad (1)$$

$$4+Y = 7 \quad (2)$$

Если не учитывать, второе, то число 5 можно получить разными способами, однако, правильным будет только один вариант, когда Y = 3, а X = 2. Последний это и есть аналог 95h. Очевидно, что по названию ресурс(+10) связан с асм командой ROL, которая знакома программистам как побитовый сдвиг налево. Она и несет основную функцию декодирования, а с помощью контрольных байт, каждый раз ее второй операнд получает нужный сдвиг. Однако, чтобы усложнить жизнь реверсерам, островки выполняют с ROL-байтом дополнительные операции сложения, вычитания, XOR с определенным ключом, поэтому красиво уложится в один алгоритм и снять все за раз не получится. таким образом, все что связано с расшифровкой - опирается на ROL-байт. Если представление об этом ресурсе у вас не складывается - представьте, как бы выглядела виртуальная машина без него: в виртуальном стеке контрольные байты для операций с одинаковыми

ячейками в Хранилищах совпадали, ну а забегая наперед – закодированные адреса можно было бы вычислить сразу с помощью одной маски XOR – таким образом, глянув на виртуальный стек можно с очень высокой точностью воспроизвести алгоритм работы VM без копирования в островках, не говоря уже о том, что островки с закрепленными за ним задачами, просто бы повторяли друг друга. Поэтому “благодаря” ROL-байту реверс-инженеру требуется уточнять детали декодирования настоящих байт в контрольном слове на каждом островке. Не лишним отметить, что ROL-байт принимает участие в Хранилище №1.5 только для декодирования его реального контрольного байта, а в свою очередь, алгоритм для шифрования/расшифровки смещений, где используется результат работы CPUID в регистре AL с тандемом операции побитового сдвига, но уже направо – ROR, идет отдельно. По поводу остальных ресурсов: так как островки разбросаны по всей выделенной памяти, адрес которой – результат API *VirtualAlloc*, то существует проблема “что где лежит”. Ее решает ресурс(+0) со только что упомянутым хранилищем №1.5, в котором хранятся неизменные смещения от начала входа в VM всех островков(по сути: вычисленные Дельта-смещения). В нашем случае он равен 01FD1000, и как положено, в главном хранилище хранится его смещение. Не менее важным является и то, что для каждого процессора результат работы CPUID может быть разным, поэтому очевидно что расшифровать смещение правильно можно только с тем же значением от CPUID с коим оно и было зашифровано! Отдельно (чуть ниже) от общего инициализатора главного хранилища, в ресурсы(+20) и (+28) ложатся другие два адреса. По существу с ними не ведется никакой работы, разве что при инициализации хранилища №2 в первое двойное слово будет положен регистр EAX после выполнения CPUID, младшая часть которого будет использована в качестве ключа, в то время как свой собственный CPUID островок проигнорирует, тут ясно, что от этой перестановки ключ для ROR не поменяется и второе слово, где лежит результат 0h-EAX(CPUID), островок, который вытаскивает это значение, выполняет обратную операцию, в результате всегда – ноль, который прибавляется к ROL-байту в главном хранилище, тут тоже понятно что операция бессмысленна. По факту функция первого островка: инициализация главного хранилища, которая только что была рассмотрена. Далее идет работа по умолчанию: чтение контрольного слова через “обвертку” (“обвертка” нужна только для первого), затем через виртуальный ESP вытаскивается контрольное слово (BCB2C5D6). Байты B2 и C5, будут одинаковы для всех верхних контрольных слов, которые вытаскивает первый островок. Из этого следует, что по ним можно отождествлять начала всех имеющихся виртуальных стеков. Кроме того, они еще говорят, что второй островок будет одинаков для всех вызывающих VM. Как видим, VM извлекает сначала байт D6, затем прибавляет его к ресурсу (+10), тем самым подготовив следующий ROL-байт для второго островка. Байт C5 будет использован вместе с текущим контрольным байтом (95h), чтобы получить смещение в хранилище №1.5. Над ними первоначально будет произведена операция сложения, затем результат увеличивается в 4 раза, полученное смещение будет сложено с 01FD1000. Прерывает этот процесс прибавление к ресурсу (+4) 4 байт, понятное дело, было использовано только одно контрольное слово, и наш виртуальный указатель сместился вниз (вверх по старшим адресам), тем самым подготовив адрес 1557420h для второго островка. Одним словом, типичный POP EAX. Возвращаемся к прерванному. VM осталось запросить в хранилище №1.5 зашифрованное смещение второго островка. Для декодирования используется связка CPUID(EAX = 1); MOV CL,AL;ROR EDX, CL. Сырое смещение будет сложено с началом VM, полученный RVA будет извлечен из стека в регистр EAX. Следующий прыжок перенесет нас на второй островок. Первым делом, становится очевидным, что 6 принципов, выделенных выше и детально рассмотренных в первом островке, без изменений реализованы во втором (ну и в третьем, четвертом и так до конца). Не лишним будет заметить, что существует иерархия в использовании регистров. Так ECX закреплен за текущим ROL-байтом, EDX за контрольным словом, EAX за виртуальным указателем или зашифрованным двойным словом (2-Б), EBX за указателем на главное хранилище, EDI как помощник предыдущего, работает с ресурсами (уместна аналогия с временными переменными, которые внедряет компилятор). Этим и объясняется наличие в главном хранилище специальной области, служащей для обмена данными непосредственно между островками. Такая потребность, например, используется для островков с алгоритмом перезаписи в “стрелке”. Им ведь только надо знать, что и где переписать. Поэтому два островка, которые будут декодировать адрес перезаписи(15000C4) и адрес запрашиваемой функции(015110F0) поместят свои результаты в указанную область. Ну и естественно, в контрольном слове для этого предусмотрены контрольные байты, которые укажут на ресурсы, где хранятся вышесказанные адреса. Собственно второй островок и связан с этой областью, но как видите, он ассистент: его функция – копировать данные из ресурса в ресурс. Так как работа только началась, копировать естественно нечего, поэтому на данном этапе его можно считать сродни инструкции пор. К слову говоря, можно сформулировать правило: настоящая работа VM начинается с островка, который первым занес данные в область хранения главного хранилища. Понятное дело, до этого остальные только “крутили” ROL-байт и “съели” несколько контрольных слов из виртуального стека. Следующий третий (он, четвертый и пятый логически взаимосвязаны) островок будет самым интригующим, все-таки алгоритм расшифровки как-никак. Как было упомянуто выше, первым делом, VM нужны два двойных слова в стеке: первое, это зашифрованный адрес и его

нужно декодировать(243BBBD6), второе – уже знакомое нам контрольное слово (4A191A68). Я думаю, что на такой подход (непосредственные адреса в виртуальном стеке) ребята из SONY DADC, вынуждены были пойти сознательно из-за ограничений, накладываемой 32 битной разрядностью процессора. В контрольном слове помещается всего 4 байта и все заняты. Поэтому нельзя создать хранилище и как следствие ROL-байт для расшифровываемых адресов, для него попросту нет места. Другое дело – 64 разрядный процессор с его 16 регистрами! Контрольное слово могло бы вместить 8 контрольных байт, существенно расширяя границы для новых хранилищ, а удвоенное количество регистров дало бы простор и новые трюки. Обратная сторона медали – разрастание в размерах островков и как следствие, куда большие потери в скорости работы защищаемого приложения. Но возвратимся обратно в 32 битный мир: следующим делом рассматриваемый островок заботится о следующем, готовя для него новый ROL-байт. Обыденно, но зато дальше идет то, ради чего они и были созданы: ROR DL, CL; ROR EDX, 8. Представьте револьвер, барабан которого рассчитан на 4 пули, первая ассемблерная команда идет на роль бойка, вторая – поворачивает барабан на следующий патрон: над каждым байтом декодируемого адреса, производится циклический сдвиг вправо на величину контрольного байта! Здесь стоит отметить одну особенность – в зависимости от “дальности” сдвига можно получить умножение и получение “зеркального отражения”. Да что там говорить!!! Вы все сами видите:

При CL = FC; DL = 24 => 42 , DL = 3B => B3

При CL = 64; DL = 7D => D7

При CL = D7; DL = 21 => 42

На последнем шаге над результатом будет произведен XOR’инг, причем для каждого островка с расшифровщиком ключ одинаков (43E2AB9D). По существу алгоритм довольно тривиален, но как уже было неоднократно сказано – при расшифровке все опирается на текущий ROL-байт в младшем регистре CL! Итак, расшифровщик превратил закодированное двойное слово в адрес 015000C4, он нам уже знаком по “стрелке”. Далее, от контрольного слова выдергивается первый байт, умножается на 4, и одним словом, в “главное хранилище” (020A7110) заносится результат вышеупомянутого хоровода. Как видите, тезис о том, что островки не связаны регистрами или стеком верен. Последние две операции, комментировать уже бессмысленно. Разве что наш виртуальный стек съедет вверх на законные два двойных слова. Перепрыгиваем к четвертому. Самый догадливые уже поняли, что это также расшифровщик. Результат его работы – 015110F0 перейдя к оному, несложно догадаться, что это и есть адрес назначения нашего CALL 0040A370! Отлично! Адрес будет сдан на хранение (ну еще бы) в ячейке 020A7290. Пятый островок. У нас есть 015110F0 и 015000C4, по операнду которого работает “стрелка”. Что остается сделать? Ну конечно, перевести! И ведь переводит – в контрольное слово вшиты соответствующие байты – извлекаются адреса 020A7290 и 020A7110. Последнее слово за POP DWORD PTR DS:[EDX], которая и осуществляет магическую подмену из базы запросов на требуемую функцию. Таким образом, следующие обращения к 0040A370(015110F0) будут напрямую. Шестой островок. Кажется, что логика его работы не поддается никакому объяснению. Откуда в 020A701C взялся адрес 0022FF6C? Он и был там до этого (ниже узнаете, кто его оставил). Это сохраненная верхушка стека на момент выполнения следующей асм команды этого островка. Проанализировав последние 4 островка, станет понятным ее назначение. Седьмой островок, являющейся очередным, необычным расшифровщиком, который извлекает число 24h. Необычным, если сопоставить декодируемое двойное слово 02482334h и единственную команду для расшифровки XOR EAX, 02482310h, это не что иное как:

AND EAX, 000000FF

XOR AL, 10h

То, что нужен только последний байт, догадаться нетрудно. Равно как и в стеке по адресу 0022FF90 (0022FF6C + 24h) лежит знакомый 0040365A. С самого начала адрес возврата не имел никакого смысла, поэтому его замена на адрес требуемой процедуры – событие закономерное, причем как последний этап. Адрес отправляется на хранение в ячейку 020A70A0, и мы уже в восьмом, который декодирует и перезаписывает 015110F0 в ячейке 020A7290 – этакая перестраховка. Итоги работы последних трех островков подводит девятый, который извлекает из последних упомянутых ячеек свежеспеченные адресаты и перезаписывает адрес возврата. После девятого островка начинается стандартная процедура выхода: сброс spin-блокировки, восстановление регистров и флагов процессора, коррекция стека и долгожданный переход в требуемую функцию по адресу 015110F0.

СПОСОБ №1А – ВЫЗОВ WinAPI(SetUnhandledExceptionFilter), смещение берется из таблицы импорта	
0044F64C PUSH 00406E34 //arg. 2 для VM 0044F657 PUSH 004011C3 //arg. 1 для VM 0044F65C MOV EAX,B3A7335C 0044F661 PUSHFD //мусор 0044F662 XOR EAX,B3737C1C //B3A7335C ^ B3737C1C = D44F40 0044F667 POPFD //мусор 0044F668 JMP SHORT 0044F669 0044F669 CALL EAX	<p>Основной код программы</p> <p>В отличии от Способа №1 база запроса отсутствует, а способ вызова характерный CALL EAX</p>
00D44F40 JMP DWORD PTR DS:[12E043C]	<p>Блокпост. Характерная “Стрелка” отсутствует. Аргументы заданы – переходим сразу к VM.</p>
JMP SHORT \$+14 <space for rent> PUSHAD PUSHFD CALL \$+5 CALL \$+7 Lock byte +1 byte POP EDX //lock byte offset LOCK DEC BYTE PTR DS:[EDX] JNS SHORT START_VM CMP BYTE PTR DS:[EDX],0 PAUSE JLE SHORT CMP JMP SHORT LOCK START_VM: MOV EBX,EDI MOV ECX,100 MOV EAX,0 REP STOS DWORD PTR ES:[EDI] //ОЧИСТКА ГЛАВНОГО ХРАНИЛИЩА ОТ ПРЕДЫДУЩЕГО ВЫЗОВА... // ...И ЕГО НОВАЯ ИНИЦИАЛИЗАЦИЯ MOV DWORD PTR DS:[EBX],EAX POP EAX MOV DWORD PTR DS:[EBX+4],EAX MOV EAX,DWORD PTR SS:[ESP] MOV DWORD PTR DS:[EBX+8],EAX MOV DWORD PTR DS:[EBX+0C],EDX MOV BYTE PTR DS:[EBX+10],95 //CRYPT-BYTE - 95h MOV DWORD PTR DS:[EBX+14],EBX MOV DWORD PTR DS:[EBX+1C],ESP MOV EAX,OFFSET 01212A10 MOV DWORD PTR DS:[EBX+20],EAX MOV EAX,OFFSET 020A6000 MOV DWORD PTR DS:[EBX+28],EAX ... JMP EAX MOV EAX,1 //ВТОРОЙ ОСТРОВOK- nop SHL EAX,2 ADD EAX,EBX ...	<p>Spin-блокировка</p> <p>Space for rent – ASCII строка</p> <p>Первый островок</p>

<pre> MOV EBX, [V-stack pointer] ADD EBX, 4*количество DWORD до зашифрованного адреса MOV EAX, [EBX] //EAX = 4340FA51 XOR EAX, 43E2AB9D //EAX = 00A251C MOV EAX, [EAX] // SetUnhandledExceptionFilter MOV [ячейка в области хранения данных_1], EAX MOV EAX, [ячейка в области хранения данных_1] MOV [ARG. 1], EAX //тот что VOID JMP ARG. 1 //переходим в WinAPI </pre>	<p>Что, по сути, происходит в общем виде</p> <p>V-stack pointer – (+1C)в главном хранилище</p> <p>00A251CC – RVA WinAPI SUEF в таблице импорта</p> <p>43E2AB9D – ключ для расшифровки</p>
---	---

Что касается способа №1А, то часть вызовов подчиняется процедуре PreStaticInitDebug(0044F4D1). Она разрезана переходниками в стиле PUSH/RET. Плюс прыжки в середину асм инструкций. Вот неполный список вызываемых WinAPI: SetUnhandledExceptionFilter, GetModuleFileNameA, DeleteFileA (\errors.txt), GlobalFree.

Address	Disassembly	Comment
004F669	CALL EAX	
004F66B	68 00020000 PUSH 200	
004F670	8D85 00FFFFFF LEA EAX, [EBP-200]	
004F676	50 PUSH EAX	
004F677	56 PUSH ESI	
004F678	68 0E284F6F PUSH 6F4F280E	
004F67D	68 E1114000 PUSH 004011E1	
004F682	9C PUSHFD	
004F683	817424 08 4E XOR DWORD PTR SS:[ESP+8], 6F9B674E	
004F68B	816C24 08 4E SUB DWORD PTR SS:[ESP+8], 6F9B674E	
004F693	817424 08 6A XOR DWORD PTR SS:[ESP+8], 553B276A	
004F69B	817424 08 6A XOR DWORD PTR SS:[ESP+8], 553B276A	
004F6A3	814424 08 4E ADD DWORD PTR SS:[ESP+8], 6F9B674E	
004F6A8	9D POPFD	
004F6AC	58 POP EAX	
004F6AD	870424 XCHG DWORD PTR SS:[ESP], EAX	
004F6B0	CALL EAX	
004F6B2	8D85 00FFFFFF LEA EAX, [EBP-200]	
004F6B8	6A 5C PUSH 5C	
004F6BA	50 PUSH EAX	
004F6BB	68 CFF64400 PUSH 004F66CF	
004F6C0	FF35 605A20 PUSH DWORD PTR DS:[&MSUCR00.strchr]	
004F6C6	C3 RETN	

SetUnhandleExceptionFilter

GetModuleFileNameA

REI is used as a jump

Кстати, за строку “\errors.txt” хотелось бы отметить особо, кроме “стрелок”, для некоторых ASCIIZ строк VM также восстанавливает адреса, а в отдельных случаях – числовые значения(небольшие величины). По аналогии с первым, в виртуальном стеке хранится адрес, который указывает на смещение WinAPI, в таблице импорта игрушки. Из смещения и берется точка входа в процедуру.

<p>Входа в процедуру:</p> <p>СПОСОБ №2 – РАБОТА С ФУНКЦИЯМИ АЛГОРИТМА ПРЯМОГО И ОБРАТНОГО ХОДА</p>	
<p>АЛГОРИТМ ПРЯМОГО ХОДА (0152253D) И ПРИСТАВЛЕННЫЙ К НЕМУ ВЫЗОВ VM</p> <p><u>ДЛЯ ОБРАТНОГО ХОДА(1520C0A) ВСЕ-ТО ЖЕ САМОЕ ЗА ИСКЛЮЧЕНИЕМ ОТСУТСТВИЯ СКРЫТЫХ КОМАНД В VM!</u></p>	
<p>0152253D PUSH EBP (Func_ARG 1. - <u>DWORD</u> , Func_ARG.2 - <u>LPDWORD</u>)</p> <p>...</p> <p>01522551 MOV EAX,CFCECDCC //характерный мусор</p> <p>01522556 MOV EAX,00413F5A //характерный мусор</p> <p>0152255B MOV EAX,1 //характерный мусор</p> <p>01522560 MOV EAX,0 //характерный мусор</p> <p>01522565 MOV EAX,CFCECDCC //характерный мусор</p> <p>...</p> <p>0152256B PUSH 00401283</p> <p>01522570 PUSH OFFSET 0152257E</p>	<p>Основной код программы</p> <p>Характерной особенностью данного способа является:</p> <ol style="list-style-type: none"> 1) Полная связанность с двумя процедурами (0152253D) 2) Как следствие первого типичный код двух алгоритмов (П и Об) 3) Характерный мусорный код в начале/конце процедур П и Об (MOV EAX, мусор) 4) В отличие от первых двух способов для VM имеет смысл адрес возврата, задаваемый в первом аргументе (возвращаемся всегда на несколько байт ниже, т.к. управление не уходит за

АЛГОРИТМ ПРЯМОГО ХОДА (0152253D) И ПРИСТАВЛЕННЫЙ К НЕМУ ВЫЗОВ VM

ДЛЯ ОБРАТНОГО ХОДА(1520C0A) ВСЕ-ТО ЖЕ САМОЕ ЗА ИСКЛЮЧЕНИЕМ ОТСУТСТВИЯ СКРЫТЫХ КОМАНД В VM!

0152253D PUSH EBP (Func_ARG 1. - <u>DWORD</u> , Func_ARG.2 - <u>LPDWORD</u>) ... 01522551 MOV EAX,CFCECDCC //характерный мусор 01522556 MOV EAX,00413F5A //характерный мусор 0152255B MOV EAX,1 //характерный мусор 01522560 MOV EAX,0 //характерный мусор 01522565 MOV EAX,CFCECDCC //характерный мусор ... 0152256B PUSH 00401283 01522570 PUSH OFFSET 0152257E	Основной код программы Характерной особенностью данного способа является: 1) Полная связанность с двумя процедурами (0152253D) 2) Как следствие первого типичный код двух алгоритмов (П и Об) 3) Характерный мусорный код в начале/конце процедур П и Об (MOV EAX, мусор) 4) В отличие от первых двух способов для VM имеет смысл адрес возврата, задаваемый в первом аргументе (возвращаемся всегда на несколько байт ниже, т.к. управление не уходит за
--	--

<pre> 01522575 JMP 00D44F40 //go to VM 0152257A NOP 0152257B NOP 0152257C NOP 0152257D NOP 0152257E MOV EAX,DWORD PTR SS:[EBP-20] //продолжение основной процедуры ... (AND / XOR) 01522662 MOV DWORD PTR SS:[EBP-10],EAX 01522666 MOV EAX,CFCECDCC //характерный мусор 0152266B MOV EAX,00413F5A //характерный мусор 01522670 MOV EAX,0 //характерный мусор 01522675 MOV EAX,0 //характерный мусор 0152267A MOV EAX,CECDCCCB //характерный мусор 01522680 MOV EAX, DWORD PTR SS:[EBP-10] //нам важен EAX! 01522684 LEAVE 01522685 RETN </pre>	<p>основную процедуру)</p>
<pre> 00D44F40 JMP DWORD PTR DS:[12E043C] </pre>	<p>Блокпост. Характерная “Стрелка” отсутствует. Аргументы заданы – переходим сразу к VM.</p>
<pre> JMP SHORT \$+14 <space for rent> PUSHAD PUSHFD CALL \$+5 CALL \$+7 Lock byte +1 byte POP EDX //lock byte offset LOCK DEC BYTE PTR DS:[EDX] JNS SHORT START_VM CMP BYTE PTR DS:[EDX],0 PAUSE JLE SHORT CMP JMP SHORT LOCK START_VM: MOV EBX,EDI MOV ECX,100 MOV EAX,0 REP STOS DWORD PTR ES:[EDI] //ОЧИСТКА ГЛАВНОГО ХРАНИЛИЩА ОТ ПРЕДЫДУЩЕГО ВЫЗОВА... // ...И ЕГО НОВАЯ ИНИЦИАЛИЗАЦИЯ MOV DWORD PTR DS:[EBX],EAX POP EAX MOV DWORD PTR DS:[EBX+4],EAX MOV EAX,DWORD PTR SS:[ESP] MOV DWORD PTR DS:[EBX+8],EAX MOV DWORD PTR DS:[EBX+0C],EDX MOV BYTE PTR DS:[EBX+10],95 //CRYPT-BYTE - 95h </pre>	<p>Spin-блокировка</p> <p>Space for rent – ASCII строка</p> <p>Первый островок</p>

<pre> MOV DWORD PTR DS:[EBX+14],EBX MOV DWORD PTR DS:[EBX+1C],ESP MOV EAX,OFFSET 01212A10 MOV DWORD PTR DS:[EBX+20],EAX MOV EAX,OFFSET 020A6000 MOV DWORD PTR DS:[EBX+28],EAX ... JMP EAX MOV EAX,1 //ВТОРОЙ ОСТРОВК- nop SHL EAX,2 ADD EAX,EBX ... </pre>	
<pre> MOV EAX, [Func_ARG.2] MOV EBX, [EAX+2C] MOV [EBP-28], EBX MOV EBX, [EAX+30] MOV [EBP-24], EBX MOV EBX, [EAX+13] MOV [EBP-20], EBX MOV EBX, [EAX+18] MOV [EBP-1C], EBX MOV EBX, [EAX+1C] MOV [EBP-18], EBX MOV EBX, [EAX+20] MOV [EBP-14], EBX MOV EBX, [EAX+24] MOV [EBP-C], EBX MOV EBX, [EAX+34] MOV [EBP-8], EBX MOV EBX, [EAX+28] MOV [EBP-4], EBX MOV ECX, DWORD PTR SS:[Func_ARG.2] MOV ECX, DWORD PTR DS:[ECX] MOV EAX, DWORD PTR SS:[Func_ARG.1] AND EAX, ECX MOV ECX, EAX JMP [ARG.1] </pre>	<p>Что, по сути, происходит в общем виде</p> <p>Func_ARG.2 – ВТОРОЙ АРГУМЕНТ ПРОЦЕДУРЫ, указатель на таблицу ключей(XOR, AND)</p> <p>Конструкция: MOV EBX, [EAX+2C] MOV [EBP-28], EBX – стандартный перенос данных из таблицы ключей в стек</p> <p>+</p> <p>Скрытые асм команды для основного хода (в реальности в VM нет этих инструкций, всю работу по смыслу организуют островки из отдельных примитивов), <i>НО</i></p> <p>Единственный островок (выполняется перед выходом из VM), который полностью дублирует инструкцию AND и служит только для второго способа: AND EAX, [ячейка в области хранения данных_1]</p>

Настоятельно рекомендуется к прочтению: http://www.iaeng.org/IJAM/issues_v36/issue_1/IJAM_36_1_12.pdf

Способ №2. Имеет совсем иные задачи в отличие от вышерассмотренных и является частью одного механизма. Вызовы по адресам (0152256Bh и 01520C38) и процедуры, в которых они находятся, включительно те, которые вызывают. Если смотреть общим планом, то все две работают со вторым аргументом по смещению [EBP+0xC].

Здесь все завязано на Odd-Even Based Cryptography (четная-нечетная базовая криптография).

01520C0A	55	PUSH EBP	0152253D	55	PUSH EBP
01520C0B	FF0D 9A2B5801	DEC DWORD PTR DS:[1582B9A]	0152253E	FF0D 962B5801	DEC DWORD PTR DS:[1582B96]
01520C11	0F84 45C0F4FE	JE 0046CC5C	01522544	0F84 911D0600	JE 015842DB
01520C17	8BEC	MOV EBP,ESP	0152254A	8BEC	MOV EBP,ESP
01520C19	83EC 28	SUB ESP,28	0152254C	83EC 28	SUB ESP,28
01520C1C	56	PUSH ESI	0152254F	56	PUSH ESI
01520C1D	50	PUSH EAX	01522550	50	PUSH EAX
01520C1E	B8 CCCDCECF	MOV EAX,CFCECDCC	01522551	B8 CCCDCECF	MOV EAX,CFCECDCC
01520C23	B8 F7404100	MOV EAX,004140F7	01522556	B8 5A3F4100	MOV EAX,00413F5A
01520C28	B8 01000000	MOV EAX,1	01522558	B8 01000000	MOV EAX,1
01520C2D	B8 00000000	MOV EAX,0	01522560	B8 00000000	MOV EAX,0
01520C32	B8 CCCDCECF	MOV EAX,CFCECDCC	01522565	B8 CCCDCECF	MOV EAX,CFCECDCC
01520C37	58	POP EAX	0152256A	58	POP EAX
01520C38	68 61124000	PUSH 00401261	0152256B	68 83124000	PUSH 00401283
01520C3D	68 4F0C5201	PUSH OFFSET 01520C4F	01522570	68 7E255201	PUSH OFFSET 0152257E
01520C42	E9 F94282FF	JMP 00D44F40	01522575	E9 C62982FF	JMP 00D44F40
01520C47	90	NOP	0152257A	90	NOP
01520C48	90	NOP	0152257B	90	NOP
01520C49	90	NOP	0152257C	90	NOP
01520C4A	90	NOP	0152257D	90	NOP
01520C4B	90	NOP	0152257E	8B45 E0	MOV EAX,DWORD PTR SS:[EBP-20]
01520C4C	90	NOP	01522581	F7D0	NOT EAX
01520C4D	90	NOP	01522583	2345 0C	AND EAX,DWORD PTR SS:[EBP+0C]
01520C4E	90	NOP	01522586	8B4D E0	MOV ECX,DWORD PTR SS:[EBP-20]
01520C4F	034D FC	ADD ECX,DWORD PTR SS:[EBP-4]	01522589	F7D1	NOT ECX
01520C52	0FAF4D D8	IMUL ECX,DWORD PTR SS:[EBP-28]	0152258B	234D 0C	AND ECX,DWORD PTR SS:[EBP+0C]
01520C56	33C1	XOR EAX,ECX	0152258E	034D E8	ADD ECX,DWORD PTR SS:[EBP-18]
01520C58	FF0D 9E2B5801	DEC DWORD PTR DS:[1582B9E]	01522591	0FAF4D E4	IMUL ECX,DWORD PTR SS:[EBP-1C]
01520C5E	0F84 E02AF6FE	JE 00483744	01522595	33C1	XOR EAX,ECX
01520C64	8B4D 0C	MOV ECX,DWORD PTR SS:[EBP+0C]	01522597	8B4D E0	MOV ECX,DWORD PTR SS:[EBP-20]
01520C67	234D E0	AND ECX,DWORD PTR SS:[EBP-20]	0152259A	F7D1	NOT ECX
01520C6A	034D DC	ADD ECX,DWORD PTR SS:[EBP-24]	0152259C	234D 0C	AND ECX,DWORD PTR SS:[EBP+0C]
01520C6D	0FAF4D F8	IMUL ECX,DWORD PTR SS:[EBP-8]	0152259F	034D F4	ADD ECX,DWORD PTR SS:[EBP-0C]
01520C71	33C1	XOR EAX,ECX	015225A2	0FAF4D EC	IMUL ECX,DWORD PTR SS:[EBP-14]
01520C73	8B4D E0	MOV ECX,DWORD PTR SS:[EBP-20]	015225A6	33C1	XOR EAX,ECX

В первой заложен прямой ход алгоритма, во второй - обратный. Принцип его работы, с помощью логической операции AND которая выполняется над аргументом сначала с маской 5CAC5AC5, затем маска переворачивается(NOT) и логическое И выполняется теперь с маской A353A53A (если выполнить XOR над двумя половинками, то результат - первоначальный вид аргумента). Причем, первая половинка хранится в двух экземплярах. Результат в регистре EAX. К примеру, аргументом будет 0790A442, тогда у нас будет его две половинки 04800040 и 310A402. Насколько я понял, из обоих посредством магических констант(ключей) вытаскиваются два числа 0 и 1, и каждое прибавляется к двух экземплярам первой половинки (04800040 и 04800041). Результирующей операцией XOR получаем 1. Обратный ход по адресу 01520C38 делает все с точностью, наоборот - из 1 и смещений 0 и 1 получает 0790A442. Обращаю ваше внимание на закономерность, что есть прямая зависимость от четности/нечетности чисел и чередование значений в 3м столбце.

Прямой ход - Аргумент/ Обратный ход- Результат	Прямой ход- результат/Обратный ход - аргумент	Прямой ход: AND EAX, 5CAC5AC5	Прямой ход: AND EAX, A353A53A*
0	3DB1F4C7	0	0
A590217B	0	04800041	A110213A
0790A442	1	04800040	0310A402
630A0CFD	2	400808C5	23020438
615BA9FC	3	400808C4	2153A138
2783017F	4	04800045	2303013A
A5D1A57E	5	04800044	A151A53A
401AA8F9	6	400808C1	0012A038
E0092DC0	7	400808C0	A0012500

*NOT(5CAC5AC5)= A353A53A

(XOR) 04800041 ^ A110213A = A590217B

Магические константы лежат по адресу 00B93AFC, оба вызова копируют их в стек, причем первый дополнительно содержит в себе:

```
MOV ECX, DWORD PTR SS:[ESP+34] //00B93AFC
MOV ECX, DWORD PTR DS:[ECX] //5CAC5AC5
MOV EAX, DWORD PTR SS:[ESP+38] //сам аргумент
AND EAX, ECX //или AND EAX, 5CAC5AC5
```

Кстати, в протоколе трассировки его протяженность составляет чуть более 30 000 строк, это объясняется тем, что островки работают в цикле, копируя из адреса в адрес. У первых двух эта цифра в 10 раз ниже! Но пожалуй самое забавное, что разработчики решив хоть как-то увеличить скорость исполнения, высадили на измену, иначе как объяснить, что в VM, есть островок, с абсолютно открытым(без обфускации) алгоритмом работы. Сдали с потрохами :)

```

OllYDbg - cnc3game.dat - [CPU - main thread]
File View Debug Trace Options Windows Help

029D6EE8 8B43 04 MOV EAX,DWORD PTR DS:[EBX+4]
029D6EEB 8343 0C ADD EAX,DWORD PTR DS:[EBX+0C]
029D6EEE 8B70 04 MOV ESI,DWORD PTR DS:[EAX+4]
029D6EF1 8B00 MOV EAX,DWORD PTR DS:[EAX]
029D6EF3 89C2 MOV EDX,EAX
029D6EF5 8A4B 10 MOV CL,BYTE PTR DS:[EBX+10]
029D6EF8 81E2 FF000000 AND EDX,000000FF
029D6EFE 0053 10 ADD BYTE PTR DS:[EBX+10],DL
029D6F01 89F2 MOV EDX,ESI
029D6F03 D2CA ROR DL,CL
029D6F05 C1C2 08 ROL EDX,8
029D6F08 D2CA ROR DL,CL
029D6F0A C1C2 08 ROL EDX,8
029D6F0D D2CA ROR DL,CL
029D6F0F C1C2 08 ROL EDX,8
029D6F12 D2CA ROR DL,CL
029D6F14 C1C2 08 ROL EDX,8
029D6F17 89D6 MOV ESI,EDX
029D6F19 89C2 MOV EDX,EAX
029D6F1B C1EA 18 SHR EDX,18
029D6F1E 013493 ADD DWORD PTR DS:[EDX*4+EBX],ESI
029D6F21 8143 04 000000 ADD DWORD PTR DS:[EBX+4],8
029D6F28 8B13 MOV EDX,DWORD PTR DS:[EBX]
029D6F2A 0353 0C ADD EDX,DWORD PTR DS:[EBX+0C]
029D6F2D C1E0 10 SHL EAX,10
029D6F30 C1E8 18 SHR EAX,18
029D6F33 00C8 ADD AL,CL
029D6F35 FF3482 PUSH DWORD PTR DS:[EAX*4+EDX]
029D6F38 B8 01000000 MOV EAX,1
029D6F3D 53 PUSH EBX
029D6F3E 0FA2 CPUID
029D6F40 25 DFFFFFFF AND EAX,FFFFFFFD
029D6F45 5B POP EBX
029D6F46 88C1 MOV CL,AL
029D6F48 8B43 0C MOV EAX,DWORD PTR DS:[EBX+0C]
029D6F4B D30C24 ROR DWORD PTR SS:[ESP],CL
029D6F4E 010424 ADD DWORD PTR SS:[ESP],EAX
029D6F51 58 POP EAX
029D6F52 FFE0 JMP EAX

```

[illegible]

Если ты уже запомнил, что в VM существует иерархия в использовании регистров, то автоматически придет на ум что в EBX = началу главного хранилища. А что у нас по смещению +24 в главном? Как и в начале этой главы, так и здесь там до сих пор находится сторожевое число. Отсюда следует, что происходит сброс SPIN блокировки. Логический итог – теперь с VM может работать любой другой поток, что недвусмысленно указывает, что скоро мы покинем виртуальные владения виртуальной машины. Нашу гипотезу подтверждают две последующие инструкции восстановления флагов и регистров процессора. Как говорится, что и требовалось доказать! Далее идет типичное замечание следов работы VM, т.е. 12 раз в стек будет положен ноль, затерты 48 байт(30h). В конечном итоге, верхушка съедет обратно на место, где она была до первого PUSH 0 и долгожданный RET 4. Фактически возврат происходит по второму аргументу, который был подменен во всех случаях, кроме вызова от SEH-обработчика (первый – от базы запросов убирается). Как ни странно, но по выходу способ №2 опять выделяется среди первых двух, можно даже сказать – идет впереди, используя прыжки в середину команд:

```
JMP SHORT 2
JMP SHORT 0A
JMP SHORT 1
POP EAX
JMP SHORT 9
JMP SHORT 6
PUSH DWORD PTR DS:[EBX+8]
JMP SHORT 0
PUSH 5773
XOR DWORD PTR SS:[ESP],00005757
ADD EBX,DWORD PTR SS:[ESP]
ADD ESP,4
JMP SHORT 2
MOV DWORD PTR DS:[EBX],0
JMP SHORT 8
POPF
JMP SHORT E
POPAD
JMP SHORT 2
RET 4
```

Подметать в стеке после себя во втором случае виртуальной машине лень. Что касается XOR, то тут даже не вычисляя очевидно, что результат будет равен 24h. Не менее очевидно, что первый PUSH работает на POPFD, т.е. в главном хранилище по смещению +8 находятся сохраненные флаги процессора(EFL). Согласитесь, что все достаточно просто и предсказуемо!



Номер способа	Контрольный байт	Смещение от начала(по ячейкам)
1	B6	2D8
1A	83	20C
2	BF	2FC

Собственно, если ты не забыл, наша задача окончательно отвязать протектор от тибериума. Давай рассуждать, как мы это сделаем. Ну, со вторым типом вызова VM вроде все ясно – мы просто перепишем этот неизвестный алгоритм, зашьем в операнды ключи и допишем найденные скрытые асм команды. Со способом 1A тоже не должно возникнуть проблем – требуемые WinAPI есть в таблице импорта, тут уж все просто. А вот способ №1 пожалуй самый проблемный. Во-первых, кол-во обращений по нему переваливает за 5k раз, во-вторых, как можно максимально быстро сgrabить восстанавливаемые им значения. Конечно, самый очевидный вариант – из 255 островков найти те, задача которых – переводить стрелки и переписывать значения(POP DWORD PTR DS:[REG_32]), но это довольно долго, да длинна только что упомянутой инструкции меньше на 3 байта, чем длинна JMP, тут без SEH обработчика не обойтись. Кул хакерское решение все-таки нашлось – среди 255 различных адресов в 1.5 есть один, который единственный в своем роде. Ну конечно, это же выход для способа №1 из VM! В рассматриваемой версии протектора он находится в ячейке по смещению 2D8(контрольный байт B6). После него мы всегда попадаем в запрашиваемый игрушкой код, значит, у нас уже есть адрес назначения! Но постойте, а откуда мы возьмем адрес самой стрелки? Если ты был наблюдательным, то сейчас твой ответ будет таким: в главном хранилище в

области, которая служит для обмена данными между островками и в одной из ячеек и хранится наш искомый адрес, ведь она будет очищена только при следующем заходе в VM! И было бы все замечательно и являлось отличной брешью в защите, если бы после пробного запуска X-кода, оказалось, что сграбленный дамп неполный. Все же в SONY DADC догадывались, что в 2011 какой-то русский инженер сможет докопаться до сути, поэтому решили подстраховаться – за один заход VM может восстанавливать несколько значений/адресов и при этом для декодированных значений/адресов в главном хранилище использует одни и те же ячейки. Таким образом, X-код на выходе грабит только последнее восстановленное значение! Из этого следует, что самый очевидный вариант – самый правильный.

Заключительным этапом будет перестроение в нормальный вид секции кода программы. В том числе: замена

PUSH 01234567

RET

Непосредственно на JMP 01234567;

удаление посторонних внедрений типа:

0044F4D2 DEC DWORD PTR DS:[158297A]

0044F4D8 JE NODVD.00482DE5

00482DE5 MOV DWORD PTR DS:[158297A],18D

00482DEF JMP NODVD.0044F4DE

замена вызовов через “стрелку” на непосредственные, перенос процедур и адресов переменных из секции .secuROM в .text и .data. Кстати, используйте олькин Call Tree (Ctrl+K), на последнем этапе вещь незаменимая.

Ах! Да! Чуть не забыл! Переход на OEP действительно осуществляется из VM – считая от первого SEH-обработчика на UD2, который естественно передает управление в VM, на 52 вызове, конечно же, по способу №1 управление передается на оригинальную точку входа. В главном хранилище ее адрес(0040A2C7) будет находиться в ячейке по смещению A0. Теперь напоминаю о секрете как попасть ювелирно в OEP. Ошибка умов из SONY DADC очевидна – островок выхода из VM в хранилище №1.5. Тут есть два фактически идентичных варианта для действий:

1. Мы подменяем адрес островка выхода на свой X-код, который ожидает первый адрес возврата в секцию кода. Тут интересно будет упомянуть, что кроме адреса возврата есть и другие приметы, например, ROL-байт равен EAh. Явный референс в сторону **Electronic Arts**. Вообще скажу по этому поводу только одно: Разработчики SecuROM – явно люди с хорошим чувством юмора и это радует!

Единственный нюанс – правильно посчитать закодированное значение начального адреса нашего перехватчика для записи в таблицу Хранилища №1.5 и перезаписать.

2. Модифицировать непосредственно островок, внедрив туда переход или непосредственно вписав код перехватчика.

Здесь есть один подводный камень для записи кода перехватчика – для островка выхода виртуальной машины изначально не предусмотрена отдельная выделенная область памяти, поэтому он всегда расположен в толще другого кода или нередко в самом конце выделенной памяти. Т.е. места для всего кода перехватчика может не хватить. Стоит взять на заметку факт, что нередко островки для выхода способов №1 и №1A расположены близко.

Первый вариант, на мой взгляд, более элегантен. Впрочем, многие, кто прочел текущий абзац, скорее всего, поступят проще – сделают все в отладчике, прибегнув к точкам останова по условию (conditional breakpoint).

Собственной персоной секция .ars – именно код, расположенный в ней, отвечает за создание виртуальной машины SecuROM 7 (WinAPI *VirtualAlloc*, REPNE MOVSB из секции .securom и XOR BYTE PTR DS:[EAX],CL). Дополнительно стоит отметить еще одну аномальную вещь: по адресу 00C93AE3

вызывается VM.

00C93AB8	4A	DEC EDX
00C93ABB	4A	DEC EDX
00C93ABC	75 2A	JNE SHORT 00C93AE8
00C93ABE	C745 C4 985CAE	MOV DWORD PTR SS:[EBP-3C],41AESC98
00C93AC5	EB 10	JMP SHORT 00C93AD7
00C93AC7	C745 C4 FFA18E	MOV DWORD PTR SS:[EBP-3C],938EA1FF
00C93ACE	EB 07	JMP SHORT 00C93AD7
00C93AD0	C745 C4 39BD1A	MOV DWORD PTR SS:[EBP-3C],221ABD39
00C93AD7	A1 F0292101	MOV EAX,DWORD PTR DS:[12129FA]
00C93ADC	8945 C0	MOV DWORD PTR SS:[EBP-40],EAX
00C93ADF	8D45 C0	LEA EAX,[EBP-40]
00C93AE2	50	PUSH EAX
00C93AE3	E8 58140B00	CALL 00D44F40
00C93AE8	EB 07	JMP SHORT 00C93AF1
00C93AEA	2222	AND AH,BYTE PTR DS:[EDX]
00C93AEC	2200	AND AL,BYTE PTR DS:[EAX]
00C93AEE	06	PUSH ES
00C93AEF	0001	ADD BYTE PTR DS:[ECX],AL
00C93AF1	7E 02	JLE SHORT 00C93AF5
00C93AF3	EB 69	JMP SHORT 00C93B5E
00C93AF5	83EC 14	SUB ESP,14

Однако как-то необычно вызывается(по Способу №1) –управление передается на следующую инструкцию короткого перехода, причем передается как обычно - через адрес возврата положенный вызовом, т.е. в ячейках главного хранилища в конце он не фигурирует! К слову сказать, на этом и подорвался мой X-код, грабящий адреса. Если за’NOP’ить вызов, то игрушка будет функционировать без ошибок. Предположение о том, что существуют фейковые вызовы VM, подтверждает факт разбора этого вызова: сначала он заносит число 10h в ячейку и занимается его декрементом, причем в ячейке оно одно, отсюда следует, что из других операций VM, она больше не может ничего с ним сделать! Операция декремента повторяется несколько раз с разными числами до нуля в ячейке. Затем далее идут другие бессмысленные значения и операции, например как чтение ячейки в Хранилище №1.5 со смещением +400 (контрольный байт FFh), в которой естественно ноль! В ячейках главного хранилища фигурирует также адрес процедуры, в которой находится вызов – 00C9A350. К тому весь этот код расположен в секции .ars, а не в .secuROM что не характерно для способа №1. Кстати в .ars расположены SEH для UD2, да и вообще по виду процедура больше походит на проверочный код, что в навесной броне защиты (например, проверка TF). Обобщая только что сказанное, скажу одно – не надо ожидать, что разработчики всегда делают абсолютно все под один шаблон.

Достаточно очевидным является и другое правило, что базы запросов должны всегда совпадать с кодом игрушки, к которому они приставлены! Но ведь и недвусмысленно ясно, что базы запросов в способах №1 и 1-А можно менять местами! И хотя действительно VM выполнит операции руководствуясь подмененной базой(более точнее: виртуальным стеком), в конечном результате запрашиваемая процедура/асм команда будет из подмененной базы, что не сулит ничего хорошего для кода, оперирующего запрашиваемой процедурой. Ну а если опять же вспомнить, что в некоторых базах восстанавливаются адреса и начальные переменные, тут уж тем более жди вагон необрабатываемых исключений.

Ну что ж! Статья неуклонно подходит к концу. Делаем выводы и подводим итоги. Было представлено серьезное кумулятивное оружие для уничтожения основной навесной брони SecuROM v7, которое основывается на просчетах разработчиков защитного комплекса. Конечно это не ноу-хау в полном смысле, а скорее всего попытка привести в нормальный вид имеющуюся теорию и показать на реальном примере работу нашего кумулятивного снаряда. Как видишь – все работает! Каким бы страшным изначально не казался SecuROM v7 с его VM, в конце концов, останется один факт – все действительно просто! Особенно, если сейчас я снова сделаю новое умоушачивающее заявление: Хорошая новость для тех, кто хочет дисассемблировать протектор в OllyDbg v2.0, забыв про все фантомы и другие подобные стелс-плагины. Невероятно, но до 7.40 этот нехитрый и вполне очевидный трюк работает (вспомни, что только при условии: вместо ollydbg.exe будет elf.exe... короче все что угодно, но только не оригинальное имя исполняемого файла отладчика):

```
MOV ESI,DWORD PTR FS:[18] ; < НОВАЯ ТОЧКА ВХОДА
MOV ESI,DWORD PTR DS:[ESI+30]
MOV DWORD PTR DS:[ESI],0 ; АКТИВИРУЕМ КОМПЛЕКС "НАКИДКА"
PUSH 011DBA77 ; ASCII "NTDLL"
MOV EAX,DWORD PTR DS:[<&KERNEL32.GetModuleHandleA>]
CALL EAX
PUSH 011DBA7E ; ASCII "NtQueryInformationProcess"
PUSH EAX
MOV EAX,DWORD PTR DS:[<&KERNEL32.GetProcAddress>]
CALL EAX
MOV ESI,DWORD PTR DS:[<&KERNEL32.VirtualProtect>]
PUSH 011DBA9A
PUSH 40
MOV EDI,EAX
PUSH 100
PUSH EAX
CALL ESI
ADD EDI,6
MOV EAX,011DB8B7
XCHG DWORD PTR DS:[EDI],EAX ;подмена вызова KiFastSystemCall
SUB EDI,6
MOV DWORD PTR DS:[11DB8BB],EAX
MOV AX,40
MOV WORD PTR DS:[11DBA9A],AX
PUSH 011DBA9A
PUSH 20
PUSH 100
PUSH EDI
```

```

CALL ESI
XOR EDI,EDI
MOV ESI,EDI
JMP 011D7B30 ; ВОВРАЩАЕМСЯ НА ПРЕДЫДУЩУЮ ТОЧКУ ВХОДА
011DB8B7 C0B8 1D010003F
011DB8BE 7F 00
MOV ECX,DWORD PTR SS:[ESP+0C] ;КОД НАШЕГО ПЕРЕХВАТЧИКА
MOV CH,7 ; if(ProcessInfoClass == ProcessDebugPort)...
SUB CH,CL
JNE SHORT 011DB8D8
XOR EAX,EAX
MOV ECX,DWORD PTR SS:[ESP+10]
MOV DWORD PTR DS:[ECX],EAX
ADD ESP,4
RETN 14

```

А в чем фокус? Хах! Ловкость инженерных рук и никакого мошенничества. Да и вообще решение уже было дано в предыдущей главе, так что тебе остается только самому попробовать использовать этот трюк против протектора. И не надо удивляться, когда первое сообщение от протектора будет просьба вставить диск, а не ругань по поводу облома по запуску модуля безопасности. Единственное что, наш защитный код не защищает от твоих установленных аппаратных точек останова и программных точек вначале контролируемых WinAPI, однако при желании эта проблема также решается. Но прежде всего, хочу обратить внимание на WinAPI *VirtualProtect*. Внедрение X-кода строилось на необходимости получить доступ на запись в секцию кода, и делал я это самым простым, но крайне не рекомендуемым способом. Использование *VirtualProtect* – самый правильный способ изменить атрибуты страницы, в данном случае атрибут защиты меняется с PAGE_EXECUTE_READ(20h) на PAGE_EXECUTE_READWRITE(40h) и обратно.

ПЕРЕД ЗАКЛЮЧЕНИЕМ

Прежде чем перейти к оригинальному заключению, хотелось бы сказать пару слов отдельно. Чтобы осмыслить все вышесказанное, прежде всего, нужно понять, что статья техническая, а не художественная. Это значит что, читая ее шаг за шагом нужно проделывать оное и в отладчике! Только так можно познакомиться ближе с SecuROM 7 и осознать насколько медленно работает игрушка с VM! Возможно, я раскрыл не все секреты, а X-код, приведенный в сорцах имеет первоначальный неоптимизированный вид. Обрадую тебя также, что следующие версии 7й серии в Cnc3: Tiberium Wars – Kane Wrath и GTA IV (7.35) конструктивно представляют тоже самое(эх! хотя бы ROL-байт в VM поменяли бы ... 95h), но с доработками и улучшениями(особенно с кодом ошибки 8019...). Кстати в случае с GTA IV в дело вступает библиотека для интернет - активации продукта: *paul.dll*. Если, что-то непонятно с нашей рассматриваемой версией - внимательно изучи разбор вызовов со снятой обфускацией, там уж все настолько очевидно, что статью в принципе можно было и не писать. Короче говоря, я оставил тебе возможность провести время с пользой :)

ЗАКЛЮЧЕНИЕ

Ну что можно сказать? Когда сам президент сказал, что нам нужны инновации, технический прогресс. X-code injection сравнительно молода, но это безусловно перспективное направление в котом стоит копать! Разработчики протекторов также не сидят, сложа руки, на момент написания статьи живет и здравствует 8я серия SecuROM. Если перед тобой сейчас твой компьютер и на мониторе - приветливая картинка *OlllyDbg v2* с загруженным модулем *cnc3game.dat* и *SecuRom_7 Profiler*, то мне остается только дать наставление “никогда не сдаваться, даже если буде трудно, даже если сил нет и твоих знаний не хватает, чтобы достичь цели”! Я сам не сразу дошел до своей цели, путь всегда тернист. Да! Казалось очень сложно, руки опускались, мысли “нет! я не смогу!”. И, тем не менее, упорство и труд были вознаграждены! Через две недели я знал о виртуальной машине не хуже ее создателей. Я очень многому научился, много взял к себе на вооружение, многое перенял, переосмыслил некоторые свои подходы к дисассемблированию и отладке, мой запас знаний существенно пополнился. Я просто поблагодарю SONY Digital Audio Disk Corporation за все ее старания сделать интересный и действительно удачный продукт... для хакеров в частности :) Теперь мне осталось лишь только одно – пожелать удачи в твоих будущих реверс - инженерных свершениях и до новых встреч на страницах журнала]]

GLOSSARY

SecuROM 7 VM – (мое определение) совокупность кода, состоящая из “островков” и использующая “хранилища” для восстановления требуемых при обращении к нему данных, таких как адреса вызываемых внутренних функций защищаемой программы, данные самой вызываемой функции, запросы к WinAPI и сокрытые некоторые операции в специальных островках (способ №2). Прямое назначение – анти-дампинг защищаемого приложения.

“Хранилища” – структурная единица, специально отведенные SecuROM’ом участки памяти, в которых храниться вся информация для работы VM. Обращение к ресурсу хранилища идет как смещение от начала данного хранилища. “Главное хранилище” – основной юнит, на котором построена работа виртуальной машины.

На рисунке ниже – Хранилище №1.5, которое всегда является поставщиком RVA всех островков и выходов из VM.

Ячейка	Зашифр offset	Декод offset	Адрес	Контрольный б...
13D803F8	F8540003	FE150000	151B0000	000000FE
13D803F4	FFE80003	FFFA0000	17000000	000000FD
13D803F0	F4940003	FD250000	142B0000	000000FC
13D803EC	F4D00003	FD340000	143A0000	000000FB
13D803E8	AC904D43	EB241350	022A1350	000000FA
13D803E4	F5E00003	FD780000	147E0000	000000F9
13D803E0	FF780003	FFDE0000	16E40000	000000F8
13D803DC	AC907603	EB241D80	022A1D80	000000F7
13D803D8	F9280003	FE4A0000	15500000	000000F6
13D803D4	AC9034C3	EB240D30	022A0D30	000000F5
13D803D0	FBFC0003	FEFF0000	16050000	000000F4
13D803CC	F7AC0003	FDEB0000	14F10000	000000F3
13D803C8	F8040003	FE010000	15070000	000000F2
13D803C4	AC905563	EB241558	022A1558	000000F1

“Ресурс в хранилище” – специальный адрес, имеющий свое строго определенное смещение от начала хранилища. Это определение я отношу по существу к главному хранилищу с диапазоном смещений от 0 до 2Ch.

“Островки” – структурная единица, обособленный участок кода в VM, конец которого отождествляется JMP EAX(EDI), RET или непосредственным указателем, иногда после прыжков стоят обращения к реверсерам или мусорный код, впрочем, читать первые необязательно. Их единый стандарт: работа с ресурсами (+4) и (+10) в “главном хранилище” и “хранилищем №1.5” для получения адреса следующего “островка”. Выполняемые функции одного островка могут быть следующими:

- 1) Копирование адресов(чисел) из ячейки в ячейку главного хранилища или копирование в ячейки из запрашиваемых адресов вне хранилищ.
- 2) Декодирование заложенных зашифрованных адресов в виртуальном стеке
- 3) Переадресация адресов в стеке(для способов №1 и №1A – адрес возврата, для способа №2 – сохраненные в стеке регистры EAX и ECX)
- 4) Правка стрелки для способа №1

“База запроса” – формирует собственный запрос к VM из двух аргументов.

- 1) Указатель на Виртуальный стек(обертка)
- 2) Адрес, куда требуется передать управление после выхода из VM. По существу имеет смысл для Способа №2. В способе №1 заменен на адрес запрашиваемой внутренней процедуры игрушки/асм инструкции, а в способе №1A – адрес запрашиваемой WinAPI. Для UD2/hardware breakpoints в SEH он равен нулю и в этом случае управление передается на следующую инструкцию после CALL 00D44F40

“Виртуальный стек” – концепция аналогично обычному машинному стеку. Отведенная область памяти и размерность в одно двойное слово(DWORD). Но важным различием является перевернутая структура виртуального: его верхушка – всегда младший адрес, и от младшего к старшим адресам смещается виртуальный указатель, т.е. секуромовский виртуальный стек как раз и “НЕ растет сверху вниз”, а “уменьшается снизу вверх”.

В своем виртуальном стеке для VM существует только два типа данных:

- 1) **Контрольное слово.** Содержание байт может варьироваться (зависит от “загруженности” островка) от 1 до 3 для каждого действия(1 байт – как стандарт). Прежде всего, это следующее значение для ROL-байта на следующем островке. Далее по значимости идет смещение по которому из Хранилища №1.5 извлекается адрес следующего островка или выхода из VM. Следующим по значимости идут смещения для ячеек в главном хранилище – туда заносятся/извлекаются адреса

или какие либо промежуточные значения в работе VM. Все что касается контрольных байт-смещений:

После извлечения контрольного байта над ним будет произведено декодирование истинного его значения, естественно с помощью текущего значения ROL-байта(т.е. в виртуальном стеке они хранятся зашифрованными если можно так выразиться), затем будет произведено увеличение истинного значения в 4 раза(x4) - в подавляющем большинстве случаев эту операция выполняет асм инструкция **SHL REG_32, 2** (битовый сдвиг влево на 2 позиции равен целочисленному умножению на 4!), полученное сырое смещение будет сложено с начальным адресом нужного хранилища.

2) **Зашифрованный адрес/число**. Одновременное использование всех двух типов данных островком говорит о том, что он выполняется процедура расшифровки адреса(успею заметить, что в этом случае в контрольном слове подготовлен байт, который укажет на ячейку в главном хранилище куда оно будет положено, ну и повторюсь что для остальных островков коим требуется этот адрес в их контрольных словах также будет указатель на ту же самую ячейку).

С помощью револьверного алгоритма(в EDX – взятый из виртуального стека **Зашифрованный адрес/число**; **CL- ROL-байт**):

ROL DL, CL

ROL EDX, 8

Который в сумме выполнится 4 раза, последним XOR с определенным ключом выполнится расшифровка.

Справедливы следующие правила:

- На один островок – одно контрольное слово. На один островок с алгоритмом расшифровки – один зашифрованный адрес в виртуальном стеке.

- Для островка с алгоритмом расшифровки в контрольном слове всегда предусмотрен байт-указатель на свободную ячейку в главном хранилище, куда требуется поместить результат.

Естественно островки, которые будут пользоваться этим результатом, уведомлены через свои контрольные слова из какого смещения требуется прочесть адрес.(естественно из-за накручивания ROL-байта значения контрольных байт-указателей в этом случае не совпадают, к тому же место такого контрольного байта-указателя в контрольном слове всегда может быть различным)

- В случае использования островком только контрольного слова указатель виртуального стека(он же ресурс (+4) в главном хранилище) съезжает вниз к старшим адресам на 4 байта(одно двойное слово).

- В случае использования островком контрольного слова и зашифрованного адреса указатель виртуального стека съезжает вниз к старшим адресам на 8 байт(два двойных слова). По инструкции **ADD DWORD PTR DS:[(+4)], 8** и **ADD DWORD PTR DS:[(+4)], 4** в обфусцированном коде островка очень просто различить его назначение!

“Обвертка на виртуальный стек” – адрес, который всегда заносится в 1й аргумент базы запросов.

По нему первый островок определяет начало виртуального стека.

Для способа №1 адрес начала самого виртуального стека обычно расположен сразу после базы запроса.

“Виртуальный указатель” или **V-ESP** – он же ресурс (+4) в главном хранилище. Аналогично прямому назначению регистра ESP указывает в виртуальном стеке его верхушку.

Адрес	DWORD's
014EA729	A5B4BF99
014EA72D	CE003A2E
014EA731	47E1DC8C
014EA735	EEE7C5E8
014EA739	017A0D45
014EA73D	CFBC9D95
014EA741	A9BC0787
014EA745	71D54108
014EA749	893918A1
014EA74D	71D5C43C

НАЧАЛО

V-ESP

**уменьшается!
от младших
к старшим
адресам**

Именно по островок сначала и узнает, какое двойное слово ему предназначено.

Правила для хранилищ:

Максимально допустимое смещение для **Хранилища №1.5** и **Главного хранилища** не может превышать $3FCh$, т.к. $FFh * 4 = 3FCh$

($FFh(255)$ – максимально допустимое значение в одном байте – контрольном байте)

Минимально допустимое смещение для Главного хранилища не может быть менее $2Ch$ (минимальные контрольный байт для ячеек = Vh), т.к. ниже область занята под специальную структуру ресурсов самого Главного Хранилища.

“Прыжок в середину инструкции” – известный трюк, основанный на правиле дисассемблирования двоичного кода. По аналогии с примером, который Крис приводил в своей известной книге **“Искусство дисассемблирования”** (Изд-во: БВХ-Петербург, 2008): у нас есть некоторый набор букв – **ПОДОРОГЕЕХАЛАМАШИНА**. Процессор начинает перебирать варианты для того, чтобы построить из него нормальное предложение (правильно дисассемблировать все четыре инструкции):

П ОДОРОГЕЕХАЛАМАШИНА, ПО ДОРОГЕЕХАЛАМАШИНА, ... ПО ДОРОГЕ ЕХАЛАМАШИНА... пока не дойдет до единственно правильного варианта – **ПО ДОРОГЕ ЕХАЛА МАШИНА**. Естественно такой расклад увидит у себя в окне хакер. Заметьте, что слово **МАШИНА** содержит еще одно знакомое слово – **ШИНА**. Используя переход в “середину” слова (асм инструкции) **МАШИНА**, после слова **ЕХАЛА**, можно построить новое предложение: **ПО ДОРОГЕ ЕХАЛА ШИНА**.

“Дельта”, “вычисление дельты” – Виртуальная машина всегда располагается по новым адресам (результаты работы **VirtualAlloc**), естественно нельзя выполнить привязку к конкретным значениям, поэтому возникает необходимость знать текущее реальное местоположение кода в памяти. **Дельта** – разность между текущим реальным положением кода и его заявленным положением при компиляции (или от какой либо фиксированного адреса).

Типичный набор инструкций для вычисления дельты:

CALL delta

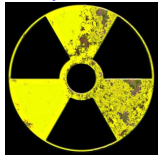
delta: POP EAX

SUB EAX, offset delta // в **EAX** будет значение дельты

В виртуальной машине Дельта-смещения можно встретить в Хранилище №1.5. Вычисление происходит вычитанием **RVA** островков от **RVA** островка входа в **VM**.

LINKS

<https://exelab.ru/f/index.php?action=vthread&forum=13&topic=19719>



<http://tuts4you.com/download.php?view.2090>

В самом конце, благодаря *Nightshade*, я совершенно случайно узнал, что существует англоязычная статья по виртуальной машине SecuROM 7.30. Было приятно узнать, что в целом наши выводы совпали. Однако в статье структура VM излагается в несколько ином виде, да и некоторых аспектов работы VM я не увидел.

<http://www.playground.ru/cheats/4932/>

NoDVD для CnC3: Tiberium Wars v1.9. Отличительной особенностью является пристроенная в секцию .методу виртуальная машина. В статье взят в качестве основного примера. Новичкам рекомендуется начать с него.

ftp://ftp.ea.com/pub/eapacific/cnc3/CNC3_patch109_russian.exe

Непосредственно полная версия протектора.

http://www.exelab.ru/rar/dl/CRACKLAB.rU_11.rar

SecuROM 4 в Empire Earth 2.

<http://www.exelab.ru/art/?action=view&id=316>

SecuROM 7 в F.E.A.R.

<http://web.textfiles.com/software/secuROM.txt>

Англоязычная статья по SecuROM

<http://En.wikipedia.org/wiki/cpuid>

Вся доступная информация об инструкции CPUID

<http://SecuROM.com>

Капитан очевидность. Однако ценной информацией там вряд ли поделятся.

<http://pid.gamecopyworld.com/>

Тузла ProtectionID позволяет точно определить версию протектора

<http://rutracker.org/forum/viewtopic.php?t=3637042>

Еще один наш подарок для Electronic Arts

<http://en.wikipedia.org/wiki/SecuROM>

На данный момент в Википедии только англоязычная статья

<http://www.ollydbg.de/version2.html>

Всегда доступна для загрузки свежая версия OllyDbg v2 (на текущий момент с поддержкой плагинов)

<http://www.pcweek.ru/themes/detail.php?ID=52199>

Статья: Защита от копирования SecuROM

<http://www.lki.ru/text.php?id=4868>

Статья: Взлом vs защита. (Особенно доставляет абзац “Он нас подвел”)*

<http://citforum.ru/security/articles/analiz/>

Статья: Анализ рынка средств защиты от копирования и взлома программных средств*

*Авторы были куплены с потрохами Protection Technology.



SecuRom_7 Profiler v1.0 – уже готов к работе :)

Extended image part

Доступ к секции кода kernel32 - Чтение, Запись и Выполнение

elf - cnc3game.dat - [Memory map]									
Address	Size	Owner	Section	Contains	Type	Access	Initial acce	Mapped	
77C01000	00043000	GDI32	.text	Code,imports,exports	Img	R E	Memory access rights	CopyOnWr	
77C44000	00002000	GDI32	.data	Data	Img	RW	RWE	CopyOnWr	
77C46000	00001000	GDI32	.rsrc	Resources	Img	R	RWE	CopyOnWr	
77C47000	00002000	GDI32	.reloc	Relocations	Img	R	RWE	CopyOnWr	
77C50000	00001000	RPCRT4		PE header	Img	R	RWE	CopyOnWr	
77C51000	00098000	RPCRT4	.text,.or	Code,imports,exports	Img	R E	RWE	CopyOnWr	
77CE9000	00001000	RPCRT4	.data	Data	Img	RW	RWE	CopyOnWr	
77CEA000	00001000	RPCRT4	.rsrc	Resources	Img	R	RWE	CopyOnWr	
77CEB000	00005000	RPCRT4	.reloc	Relocations	Img	R	RWE	CopyOnWr	
77E40000	00001000	kernel32		PE header	Img	R	RWE	CopyOnWr	
77E41000	0008A000	kernel32	.text	Code,imports,exports	Img	RWE	CopyOnWr	RWE	CopyOnWr
77ECB000	00005000	kernel32	.data	Data	Img	RW	CopyOnWr	RWE	CopyOnWr
77ED0000	0006B000	kernel32	.rsrc	Resources	Img	R	RWE	CopyOnWr	
77F3B000	00007000	kernel32	.reloc	Relocations	Img	R	RWE	CopyOnWr	
78130000	00001000	MSUCR80		PE header	Img	R	RWE	CopyOnWr	
78131000	00063000	MSUCR80	.text	Code	Img	R E	RWE	CopyOnWr	
78194000	0002B000	MSUCR80	.rdata	Imports,exports	Img	R	RWE	CopyOnWr	
781BF000	00007000	MSUCR80	.data	Data	Img	RW	CopyOnWr	RWE	CopyOnWr

Первая часть X-кода (подготовительная)

elf - cnc3game.dat - [CPU - main thread, module cnc3game]		
Address	Disassembly	Comment
011DB26A	0FB659 04	MOVZX EBX, BYTE PTR DS:[ECX+4]
011DB26E	03FE	ADD EDI,ESI
011DB270	03F3	ADD ESI,EBX
011DB272	0FB659 05	MOVZX EBX, BYTE PTR DS:[ECX+5]
011DB276	03FE	ADD EDI,ESI
011DB278	03F3	ADD ESI,EBX
011DB27A	0FB659 06	MOVZX EBX, BYTE PTR DS:[ECX+6]
011DB27E	E9 7D040000	JMP 011DB700
011DB283	90	NOP
011DB284	90	NOP
011DB285	90	NOP
011DB286	03FE	ADD EDI,ESI
011DB288	03F3	ADD ESI,EBX

elf - cnc3game.dat - [CPU - main thread, module cnc3game]		
File View Debug Trace Plugins Options Windows Help		
File View Debug Trace Plugins Options Windows Help		
011DB6CB	CC	INT3
011DB6CC	CC	INT3
011DB6CD	CC	INT3
011DB6CE	CC	INT3
011DB6CF	CC	INT3
011DB6D0	60	PUSHAD
011DB6D1	6A 40	PUSH 40
011DB6D3	50	PUSH EAX
011DB6D4	90	NOP
011DB6D5	6A 00	PUSH 0
011DB6D7	6A 00	PUSH 0
011DB6D9	90	NOP
011DB6DA	FF15 FEB94C01	CALL DWORD PTR DS:[<&USER32.MessageBoxA
011DB6E0	EB 34	JMP SHORT 011DB716
011DB6E2	0058 A3	ADD BYTE PTR DS:[EAX-5D],BL
011DB6E5	EB B6	JMP SHORT 011DB69D
011DB6E7	1D 01EB3100	SBB EAX,01EB01
011DB6EC	0000	ADD BYTE PTR DS:[EAX],AL
011DB6EE	0000	ADD BYTE PTR DS:[EAX],AL
011DB6F0	0000	ADD BYTE PTR DS:[EAX],AL
011DB6F2	0000	ADD BYTE PTR DS:[EAX],AL
011DB6F4	0000	ADD BYTE PTR DS:[EAX],AL
011DB6F6	0000	ADD BYTE PTR DS:[EAX],AL
011DB6F8	0000	ADD BYTE PTR DS:[EAX],AL
011DB6FA	0000	ADD BYTE PTR DS:[EAX],AL
011DB6FC	0000	ADD BYTE PTR DS:[EAX],AL
011DB6FE	0000	ADD BYTE PTR DS:[EAX],AL
011DB700	01F7	ADD EDI,ESI
011DB702	01DE	ADD ESI,EBX
011DB704	0FB659 07	MOVZX EBX, BYTE PTR DS:[ECX+7]
011DB708	83F8 6C	CMPL EAX,6C
011DB70B	0F85 75FBFFFF	JNE 011DB286
011DB711	E9 83000000	JMP 011DB799
011DB716	61	POPAD
011DB717	E9 CF000000	JMP 011DB7EB
011DB71C	50	PUSH EAX
011DB71D	EB 08	JMP SHORT 011DB727

elf - cnc3game.dat - [CPU - main thread, module cnc3game]		
File View Debug Trace Plugins Options Windows Help		
File View Debug Trace Plugins Options Windows Help		
011DB793	FEC2	INC DL
011DB795	FEC5	INC CH
011DB797	EB A8	JMP SHORT 011DB741
011DB799	60	PUSHAD
011DB79A	68 B6B71D01	PUSH 011DB7B6
011DB79F	FF15 1ABC4C01	CALL DWORD PTR DS:[<&KERNEL32.GetModuleHandleA>]
011DB7A5	68 C2B71D01	PUSH 011DB7C2
011DB7AA	50	PUSH EAX
011DB7AB	FF15 46BD4C01	CALL DWORD PTR DS:[<&KERNEL32.GetProcAddress>]
011DB7B1	EB 1E	JMP SHORT 011DB7D1

ASCII "KERNEL32"

ASCII "GetDriveTypeA"

elf - cnc3game.dat - [CPU - main thread, module cnc3game]		
File View Debug Trace Plugins Options Windows Help		
File View Debug Trace Plugins Options Windows Help		
011DB7CD	65:41	INC ECX
011DB7CF	0000	ADD BYTE PTR DS:[EAX],AL
011DB7D1	89C1	MOV ECX,EAX
011DB7D3	2D E3B61D01	SUB EAX,011DB6E3
011DB7D8	BF FBFFFFFF	MOV EDI,-5
011DB7DD	29C7	SUB EDI,EAX
011DB7DF	C601 E9	MOV BYTE PTR DS:[ECX],0E9
011DB7E2	8979 01	MOV DWORD PTR DS:[ECX+1],EDI
011DB7E5	61	POPAD
011DB7E6	E9 9BFAFFFF	JMP 011DB286

Результат работы первой части X-кода

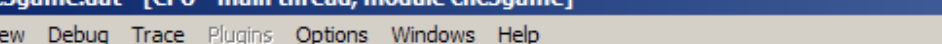
```

elf - cnc3game.dat - [CPU - main thread, module kernel32]
File View Debug Trace Plugins Options Windows Help
[Icons] [Navigation] [Disassembly] [Registers] [Memory] [Comments] [Log]

77E49A02 90 NOP
77E49A03 90 NOP
77E49A04 90 NOP
77E49A05 E9 D91C3989 JMP 011DB6E3 DRIVE_X kernel32.GetDriveTypeA(RootPath)
77E49A0A 837D 08 00 CMP DWORD PTR SS:[RootPath],0
77E49A0E 74 1D JE SHORT 77E49A2D
77E49A10 FF75 08 PUSH DWORD PTR SS:[RootPath] [Arg1 => [RootPath]]
77E49A13 E8 0FB20100 CALL 77E64C27 kerne132.77E64C27
77E49A18 85C0 TEST EAX,EAX
77E49A1A 0F84 A35B0300 JE 77E7F5C3
77E49A20 8B40 04 MOV EAX, DWORD PTR DS:[EAX+4]
77E49A23 > 50 PUSH EAX [RootPath]
77E49A24 E8 97B90100 CALL GetDriveTypeW [KERNEL32.GetDriveTypeW]
77E49A29 > 5D POP EBP
77E49A2A C2 0400 RETN 4
77E49A2D > 33C0 XOR EAX,EAX
77E49A2F EB F2 JMP SHORT 77E49A23

```

Вторая часть X-кода (непосредственно сам код перехватчика)



The screenshot shows the OllyDbg interface with the following details:

- File:** cnc3game.dat - [CPU - main thread, module cnc3game]
- Menu Bar:** File, View, Debug, Trace, Plugins, Options, Windows, Help
- Toolbar:** Includes icons for file operations, navigation, and execution.
- Assembly View:**

Address	Disassembly	Comment
011DB6E3	58	POP EAX
011DB6E4	A3 EBB61D01	MOV DWORD PTR DS:[11DB6EB],EAX
011DB6E9	EB 31	JMP SHORT 011DB71C
011DB6EB	3BF2	CMP ESI,EDX
011DB6ED	E6 77	OUT 77,AL

elf - cnc3game.dat - [CPU - main thread, module cnc3game]			
File View Debug Trace Plugins Options Windows Help			
011DB717	✓ E9 CF000000	JMP	011DB7EB
011DB71C	50	PUSH	EAX
011DB71D	✓ EB 08	JMP	SHORT 011DB727
011DB71F	90	NOP	
011DB720	90	NOP	
011DB721	90	NOP	
011DB722	68 C2B71D01	PUSH	011DB7C2
011DB727	60	PUSHAD	
011DB728	33C0	XOR	EAX,EAX
011DB72A	A0 EBB61D01	MOV	AL,BYTE PTR DS:[11DB6EB]
011DB72F	BE EBB61D01	MOV	ESI,011DB6EB
011DB734	BF 00B91D01	MOV	EDI,011DB900
011DB739	33C9	XOR	ECX,ECX
011DB73B	B5 30	MOV	CH,30
011DB73D	B1 30	MOV	CL,30
011DB73F	33D2	XOR	EDX,EDX
011DB741	3AC2	CMP	AL,DL
011DB743	✓ 74 21	JE	SHORT 011DB766
011DB745	✓ EB 00	JMP	SHORT 011DB747
011DB747	80F9 39	CMP	CL,39
011DB74A	✓ 75 03	JNE	SHORT 011DB74F
011DB74C	B1 40	MOV	CL,40
011DB74E	90	NOP	
011DB74F	80F9 46	CMP	CL,46
011DB752	✓ 74 09	JE	SHORT 011DB75D
011DB754	90	NOP	
011DB755	90	NOP	
011DB756	90	NOP	
011DB757	FEC1	INC	CL
011DB759	FEC2	INC	DL
011DB75B	^ EB E4	JMP	SHORT 011DB741
011DB75D	80E9 16	SUB	CL,16
011DB760	✓ EB 28	JMP	SHORT 011DB78A
011DB762	90	NOP	
011DB763	90	NOP	
011DB764	90	NOP	
011DB765	90	NOP	
011DB766	882F	MOV	BYTE PTR DS:[EDI],CH
011DB768	47	INC	EDI
011DB769	880F	MOV	BYTE PTR DS:[EDI],CL

elf - cnc3game.dat - [CPU - main thread, module cnc3game]			
File View Debug Trace Plugins Options Windows Help			
011DB7E6	E9 9BFAFFFF	JMP 011DB286	
011DB7EB	68 B6B71D01	PUSH 011DB7B6	ASCII "KERNEL32"
011DB7F0	FF15 1ABC4C01	CALL DWORD PTR DS:[&KERNEL32.GetModuleHandleA]	
011DB7F6	68 C2B71D01	PUSH 011DB7C2	ASCII "GetDriveTypeA"
011DB7FB	50	PUSH EAX	
011DB7FC	FF15 46BD4C01	CALL DWORD PTR DS:[&KERNEL32.GetProcAddress]	
011DB802	83E8 00	SUB EAX,0	
011DB805	2D 28B81D01	SUB EAX,011DB828	
011DB80A	C605 28B81D01	MOV BYTE PTR DS:[11DB828],0E9	
011DB811	A3 29B81D01	MOV DWORD PTR DS:[11DB829],EAX	
011DB816	C605 A043FB00	MOV BYTE PTR DS:[0FB43A0],74	
011DB81D	C605 A63FFB00	MOV BYTE PTR DS:[0FB3FA6],0F9	
011DB824	61	POPAD	
011DB825	55	PUSH EBP	
011DB826	89E5	MOV EBP,ESP	
011DB828	0000	ADD BYTE PTR DS:[EAX],AL	
011DB82A	0000	ADD BYTE PTR DS:[EAX],AL	
011DB82C	0000	ADD BYTE PTR DS:[EAX],AL	
011DB82E	0000	ADD BYTE PTR DS:[EAX],AL	

Работа SecuROM v7 опирается как и на WinAPI...

elf - cnc3game.dat - [CPU - main thread, module kernel32]			
File View Debug Trace Plugins Options Windows Help			
77E41A5E	90	NOP	Registers (MMX)
77E41A5F	90	NOP	EAX 7FFDFBF8
77E41A60	90	NOP	ECX 7FFDFC00
77E41A61	90	NOP	EDX 011E5440 ASCII "\\.\cnc3game.dat"
77E41A62	90	NOP	EDX 80000000
77E41A63	8BFF	MOV EDI,EDI	ESP 00219470
77E41A65	55	PUSH EBP	EBP 0021948C
77E41A66	8BEC	MOV EBP,ESP	ESI 77E41A63 kernel32.CreateFileA
77E41A68	FF75 08	PUSH DWORD PTR SS:[FileName]	EDI 00000000
77E41A6B	E8 87310200	CALL 77E64C27	EIP 77E41A69 kernel32.77E41A69
77E41A70	85C0	TEST EAX,EAX	C 0 ES 0023 32bit 0(FFFFFFFF)
77E41A72	74 1E	JS SHORT 77E41A92	P 0 GS 001B 32bit 0(FFFFFFFF)
77E41A74	FF75 20	PUSH DWORD PTR SS:[hTemplate]	A 0 SS 0023 32bit 0(FFFFFFFF)
77E41A77	FF75 1C	PUSH DWORD PTR SS:[Attributes]	Z 0 DS 0023 32bit 0(FFFFFFFF)
77E41A7A	FF75 18	PUSH DWORD PTR SS:[CreationDistribution]	S 0 FS 003B 32bit 7FFDF000(4000)
77E41A7D	FF75 14	PUSH DWORD PTR SS:[pSecurity]	T 0 GS 0000 NULL
77E41A80	FF75 10	PUSH DWORD PTR SS:[ShareMode]	D 0
77E41A83	FF75 0C	PUSH DWORD PTR SS:[DesiredAccess]	0 0 LastErr 00000002 ERROR_FILE_NOT_FOUND
77E41A86	FF70 00	PUSH DWORD PTR DS:[EAX+4]	EFL 00000202 (NO,NB,NE,a,NS,P,0,GE,C)
77E41A89	E8 0B200200	CALL CreateFileW	MM0 0000 0000 0000 0000
77E41A8E	5D	POP EBP	MM1 0000 0000 0000 0000
77E41A8F	C2 1C00	RETN 1C	MM2 0000 0000 0000 0000
77E41A92	83C8 FF	OR EAX,FFFFFFFF	MM3 0000 0000 0000 0000
77E41A95	EB F7	JMP SHORT 77E41A8E	MM4 0000 0000 0000 0000
77E41A97	90	NOP	MM5 0000 0000 0000 0000
77E41A98	90	NOP	MM6 C000 0000 0000 0000
77E41A99	90	NOP	MM7 F43E 2901 F480 0000
77E41A9A	90	NOP	
77E41A9C	8BFF	MOV EDI,EDI	MM8 00000000 00000000 00000000 00000000
77E41A9E	55	PUSH EBP	MM9 00000000 00000000 00000000 00000000
77E41A9F	8BEC	MOV EBP,ESP	MM10 00000000 00000000 00000000 00000000
77E41AA1	56	PUSHESI	MM11 00000000 00000000 00000000 00000000
77E41AA2	8B35 F412E47	MOV ESI,DWORD PTR DS:[&ntdll.NtProtect]	MM12 00000000 00000000 00000000 00000000
77E41AA8	57	PUSH EDI	MM13 00000000 00000000 00000000 00000000
77E41AA9	FF75 18	PUSH DWORD PTR SS:[pOldProtect]	MM14 00000000 00000000 00000000 00000000
77E41AAC	8D45 10	LEA EAX,[Size]	MM15 00000000 00000000 00000000 00000000
77E41AAF	FF75 14	PUSH DWORD PTR SS:[NewProtect]	MM16 00000000 00000000 00000000 00000000
77E41AB2	5B	PUSH EAX	MM17 00000000 00000000 00000000 00000000
Dest=77E64849 (kernel32.CreateFileW)			
<div> <div>Address Hex dump ASCII</div> <div> 00073000 00 00 00 10 FF FF FF FF 01 00 FF FF 00 00 00 00 00073010 80 02 00 00 C2 00 C0 90 00 00 00 00 00 0F 01 B B 0h? Я 00073020 F5 00 00 00 00 00 45 00 78 00 63 00 65 00 70 00 x E x c e p 00073030 74 00 69 00 6F 00 6E 00 00 00 00 00 00 00 01 t i o n 00073040 40 00 53 00 20 00 53 00 61 00 6E 00 73 00 20 00 M S S a n s 00073050 53 00 65 00 72 00 69 00 66 00 00 00 00 00 00 00 S e r i f 00073060 00 00 00 00 01 00 01 50 06 01 E0 00 32 00 0E 00 r -P!-a 2 00073070 01 00 00 00 FF FF 80 00 4F 00 40 00 00 00 00 00 r яяъ D K 00073080 00 00 00 00 00 00 00 00 00 00 01 50 71 01 E0 00 Pq;a 00073090 32 00 0E 00 04 00 00 00 FF FF 80 00 44 00 65 00 2 яъ J яяъ D e 000730A0 62 00 75 00 67 00 20 00 62 00 72 00 65 00 61 00 b u g b r e a 000730B0 68 00 00 00 00 00 00 00 00 00 00 00 00 00 00 k 000730C0 81 00 02 50 07 00 07 00 09 01 09 00 64 00 00 00 f :P* * r d 000730D0 FF FF 02 00 45 00 70 00 54 00 79 00 70 00 65 00 яя, E x t y p e 000730E0 00 00 00 00 00 00 00 00 00 00 00 00 80 10 02 50 t-P 000730F0 07 00 13 00 09 01 1C 00 65 00 00 00 FF FF 82 00 * r e яя, </div> </div>			
Breakpoint at kernel32.77E41A89			

...Так и на низкоуровневые функции из ntdll. Обратите внимание на адрес возврата и аргумент *ProcessInfoClass*. Однако в любом случае отладчик не будет обнаружен – мы “зашторились”

elf - cnc3game.dat - [CPU - main thread, module ntdll]

Address	Disassembly	Comment
7C82740C	90	NOP
7C827410	B8 A1000000	MOV EAX, 0A1
7C827412	BA B7B81D01	MOV EDX, 11DB8B7
7C827417	FF12	CALL DWORD PTR DS:[EDX]
7C827419	C2 1400	RET 14
7C82741C	90	NOP
7C82741D	B8 A2000000	MOV EAX, 0A2
7C827422	BA 0003FE7F	MOV EDX, 7FFE0300
7C827427	FF12	CALL DWORD PTR DS:[EDX]
7C827429	C2 1400	RET 14
7C82742C	90	NOP

Return to cnc3game.00CF2380

ProcessHandle = INVALID_HANDLE_VALUE
ProcessInfoClass = ProcessDebugPort
Buffer = 002194C4 -> 00
Bufsize = 4
Length = 002194C0 -> -2121148723.

С 7.33 наша защита работает!!! OllyDbg v2.01(a4) без Фантомов и Стронхов

elf - cnc3game.dat - [CPU - main thread, module cnc3game]

Address	Disassembly	Comment
011DB83E	0000	ADD BYTE PTR DS:[EAX], AL
011DB840	64:8B35 180000	MOV ESI, DWORD PTR FS:[18]
011DB847	8B76 30	MOV ESI, DWORD PTR DS:[ESI+30]
011DB84A	893E	MOV DWORD PTR DS:[ESI], EDI
011DB84C	31F6	XOR ESI, ESI
011DB84E	90	NOP
011DB84F	90	NOP
011DB850	90	NOP
011DB851	90	NOP
011DB852	90	NOP
011DB853	68 77BA1D01	PUSH 011DBA77
011DB858	A1 1ABC4C01	MOV EAX, DWORD PTR DS:[<KERNEL32.GetModuleHandle>]
011DB85D	FFD0	CALL EAX
011DB85F	68 7EBA1D01	PUSH 011DBA7E
011DB864	50	PUSH EAX
011DB865	A1 46BD4C01	MOV EAX, DWORD PTR DS:[<KERNEL32.GetProcAddress>]
011DB86A	FFD0	CALL EAX
011DB86C	8B35 76BE4C01	MOV ESI, DWORD PTR DS:[<KERNEL32.VirtualProtect>]
011DB872	68 9ABA1D01	PUSH 011DBA9A
011DB877	6A 40	PUSH 40
011DB879	89C7	MOV EDI, EAX
011DB87B	68 00010000	PUSH 100
011DB880	50	PUSH EAX
011DB881	FFD6	CALL ESI
011DB883	83C7 06	ADD EDI, 6
011DB886	B8 B7B81D01	MOV EAX, 011DB8B7
011DB88B	8707	XCHG DWORD PTR DS:[EDI], EAX
011DB88D	83EF 06	SUB EDI, 6
011DB890	A3 BB881D01	MOV DWORD PTR DS:[11DB88B], EAX
011DB895	66:B8 4000	MOV AX, 40
011DB899	66:A3 9ABA1D01	MOV WORD PTR DS:[11DBA9A], AX
011DB89F	68 9ABA1D01	PUSH 011DBA9A
011DB8A4	6A 20	PUSH 20
011DB8A6	68 00010000	PUSH 100
011DB8AB	57	PUSH EDI
011DB8AC	FFD6	CALL ESI
011DB8AE	31FF	XOR EDI, EDI
011DB8B0	89FE	MOV ESI, EDI
011DB8B2	E9 79C2FFFF	JMP 011D7B30
011DB8B7	C0B8 1D010003	SAR BYTE PTR DS:[EAX+30001D], 0FE
011DB8BE	7F 00	JG SHORT 011DB8C0
011DB8C0	8B4CE4 0C	MOV ECX, DWORD PTR SS:[ESP+0C]
011DB8C4	B5 07	MOV CH, 7
011DB8C6	28CD	SUB CH, CL
011DB8C8	75 0E	JNE SHORT 011DB8D8
011DB8CA	31C0	XOR EAX, EAX
011DB8CC	8B4CE4 10	MOV ECX, DWORD PTR SS:[ESP+10]
011DB8D0	8901	MOV DWORD PTR DS:[ECX], EAX
011DB8D2	83C4 04	ADD ESP, 4
011DB8D5	C2 1400	RET 14

< НОВАЯ ТОЧКА ВХОДА

АКТИВИРУЕМ КОМПЛЕКС "НАКИДКА"

ЗАЩИЩАЕМСЯ. "ШТОРА-1"

ASCII "NTDLL"

ASCII "NtQueryInformationProcess"

ВОВРАЩАЕМСЯ НА ПРЕДЫДУЩУЮ ТОЧКУ ВХОДА

Shift out of range

GetDriveTypeA для поиска приводов

```

elf - cnc3game.dat - [CPU - main module, module cnc3game]
File View Debug Trace Plugins Options Windows Help
[Icons] [L][E][M][W][T][C][R][...] [B][M][H] [≡]

00D85F62 L C3 RETN
00D85F63 $ 55 PUSH EBP
00D85F64 . 8BEC MOV EBP,ESP
00D85F66 . 83EC 0C SUB ESP,0C
00D85F69 . 807D 08 20 CMP BYTE PTR SS:[Arg1],20
00D85F6D ✓ 74 29 JE SHORT 00D85F98
00D85F6F . 0FB4 05 08 MOVSX EAX,BYTE PTR SS:[Arg1]
00D85F73 . 50 PUSH EAX
00D85F74 . 8D45 F4 LEA EAX,[LOCAL.3]
00D85F77 . 68 649B4601 PUSH OFFSET 01469B46
00D85F7C . 50 PUSH EAX
00D85F7D . E8 40560500 CALL 00D0B5C2
00D85F82 . 83C4 0C ADD ESP,0C
00D85F85 . 8D45 F4 LEA EAX,[LOCAL.3]
00D85F88 . 50 PUSH EAX
00D85F89 . FF15 7EBD4C01 CALL DWORD PTR DS:[<KERNEL32.GetDriveTypeA>]
00D85F8F . 83F8 05 CMP EAX,5
00D85F92 ✓ 75 04 JNE SHORT 00D85F98
00D85F94 . B0 01 MOV AL,1
00D85F96 ✓ EB 02 JMP SHORT 00D85F9A
00D85F98 > 32C0 XOR AL,AL
00D85F9A > C9 LEAVE
00D85F9B L C2 0400 RETN 4

cnc3game.00D85F63(guessed Arg1)

Format = "%c:\\\\"
Arg1 => OFFSET LOCAL.3
cnc3game.00D0B5C2

RootPath => OFFSET LOCAL.3
KERNEL32.GetDriveTypeA
CONST 5 => DRIVE_CDROM

```

CreateFileA и DeviceIoControl – две ключевые функции в первой части проверки.

[illegible]

Вторая часть проверки – синхронные потоки, DeviceIoControl и QueryPerformanceCounter

elf: cnc3game.dat - [CPU - main thread, module cnc3game]		
File View Debug Trace Plugins Options Windows Help		
010F12E9	8945 FC	MOV DWORD PTR SS:[LOCAL.1],EAX
010F12EC	0F8E 9A000000	JLE 010F138C
010F12F2	56	PUSH ESI
010F12F3	57	PUSH EDI
010F12F4	8DBB A8000000	LEA EDI,[EBX+0A8]
010F12FA	68 A0000000	PUSH 0A0
010F12FF	E8 3E99CEFF	CALL 00DDAC42
010F1304	68 A0000000	PUSH 0A0
010F1309	8BF0	MOV ESI,EAX
010F130B	6A 00	PUSH 0
010F130D	56	PUSH ESI
010F130E	E8 6D9FCEFF	CALL 000DB280
010F1313	8B45 FC	MOV EAX,DWORD PTR SS:[LOCAL.1]
010F1316	8A4403 08	MOV AL,BYTE PTR DS:[EAX+EBX+8]
010F131A	8806	MOV BYTE PTR DS:[ESI],AL
010F131C	8D47 80	LEA EAX,[EDI+80]
010F131F	8946 04	MOV DWORD PTR DS:[ESI+4],EAX
010F1322	8D87 80000000	LEA EAX,[EDI+80]
010F1328	8946 0C	MOV DWORD PTR DS:[ESI+0C],EAX
010F132B	83C4 10	ADD ESP,10
010F132E	8D87 00010000	LEA EAX,[EDI+100]
010F1334	8946 10	MOV DWORD PTR DS:[ESI+10],EAX
010F1337	8D45 F8	LEA EAX,[LOCAL.2]
010F133A	50	PUSH EAX
010F133B	33C0	XOR EAX,EAX
010F133D	50	PUSH EAX
010F133E	56	PUSH ESI
010F133F	68 990F0F01	PUSH 010F0F99
010F1344	50	PUSH EAX
010F1345	50	PUSH EAX
010F1346	897E 08	MOV DWORD PTR DS:[ESI+8],EDI
010F1349	FF15 DEBB4C00	CALL DWORD PTR DS:[<&KERNEL32.CreateThread>]
010F134F	85C0	TEST EAX,EAX
010F1351	8987 80010000	MOV DWORD PTR DS:[EDI+100],EAX
010F1357	89B7 00020000	MOV DWORD PTR DS:[EDI+200],ESI
010F135D	75 05	JNE SHORT 010F1364
010F135F	8845 0B	MOV BYTE PTR SS:[ARG.1+3],AL
010F1362	EB 09	JMP SHORT 010F136D
010F1364	6A 02	PUSH 2
010F1366	50	PUSH EAX
010F1367	FF15 66BC4C00	CALL DWORD PTR DS:[<&KERNEL32.SetThreadPriority>]
010F136D	FF45 FC	INC DWORD PTR SS:[LOCAL.1]
010F1370	8B45 FC	MOV EAX,DWORD PTR SS:[LOCAL.1]
010F1373	83C7 04	ADD EDI,4
010F1376	3B43 04	CMP EAX,DWORD PTR DS:[EBX+4]
010F1379	0F8C 7BFFFFFF	JL 010F12FA
010F137F	807D 0B 00	CMP BYTE PTR SS:[ARG.1+3],0
010F1383	5F	POP EDI

Arg3 = 0A0

Arg2 = 0

Arg1

cnc3game.000DB280

pThreadId => OFFSET LOCAL.2

CreationFlags => 0

Parameter

StartAddress = cnc3game.10F0F99

StackSize => 0

pSecurity => NULL

KERNEL32.CreateThread

Priority = THREAD_PRIORITY_HIGHEST

hThread

KERNEL32.SetThreadPriority

			Callback
010F0F99	55	PUSH EBP	
010F0F9A	8D6C24 8C	LEA EBP,[LOCAL.29]	
010F0F9E	81EC DC000000	SUB ESP,0DC	
010F0FA4	53	PUSH EBX	
010F0FA5	56	PUSH ESI	
010F0FA6	8B75 7C	MOV ESI,DWORD PTR SS:[ARG.1]	
010F0FA9	8B46 04	MOV EAX,DWORD PTR DS:[ESI+4]	
010F0FAC	8945 6C	MOV DWORD PTR SS:[LOCAL.2],EAX	
010F0FAF	8B46 08	MOV EAX,DWORD PTR DS:[ESI+8]	
010F0FB2	8945 60	MOV DWORD PTR SS:[LOCAL.5],EAX	
010F0FB5	8B46 0C	MOV EAX,DWORD PTR DS:[ESI+0C]	
010F0FB8	8945 68	MOV DWORD PTR SS:[LOCAL.3],EAX	
010F0FBB	8B46 10	MOV EAX,DWORD PTR DS:[ESI+10]	
010F0FBE	33DB	XOR EBX,EBX	
010F0FC0	57	PUSH EDI	
010F0FC1	8945 64	MOV DWORD PTR SS:[LOCAL.4],EAX	
010F0FC4	895D 7C	MOV DWORD PTR SS:[ARG.1],EBX	
010F0FC7	895D 70	MOV DWORD PTR SS:[LOCAL.1],EBX	
010F0FCA	8D7E 2C	LEA EDI,[ESI+2C]	
010F0FCD	8B45 60	MOV EAX,DWORD PTR SS:[LOCAL.5]	
010F0FD0	68 F4010000	PUSH 1F4	Timeout = 500. ms
010F0FD5	FF30	PUSH DWORD PTR DS:[EAX]	hObject
010F0FD7	FF15 DAB84C0	CALL DWORD PTR DS:[<KERNEL32.WaitForSingleObject>]	KERNEL32.WaitForSingleObject
010F0FDD	3D 02010000	CMP EAX,102	CONST 102 => WAIT_TIMEOUT
010F0FE2	75 1B	JNE SHORT 010F0FF5	
010F0FE4	8B45 64	MOV EAX,DWORD PTR SS:[LOCAL.4]	
010F0FE7	53	PUSH EBX	Timeout => 0
010F0FE8	FF30	PUSH DWORD PTR DS:[EAX]	hObject
010F0FEA	FF15 DAB84C0	CALL DWORD PTR DS:[<KERNEL32.WaitForSingleObject>]	KERNEL32.WaitForSingleObject
010F0FF0	85C0	TEST EAX,EAX	
010F0FF2	0F85 FC000000	JNE 010F10F4	
010F0FF8	33C0	XOR EAX,EAX	
010F0FFA	E9 02010000	JMP 010F1101	
010F0FFF	3BC3	CMP EAX,EBX	
010F1001	75 F5	JNE SHORT 010F0FF8	
010F1003	6A 58	PUSH 58	Arg3 = 58
010F1005	8D45 F0	LEA EAX,[LOCAL.33]	
010F1008	53	PUSH EBX	Arg2 => 0
010F1009	50	PUSH EAX	Arg1 => OFFSET LOCAL.33
010F100A	E8 71A2CEFF	CALL 000DB280	cnc3game.000DB280
010F100F	6A 58	PUSH 58	Arg3 = 58
010F1011	8D45 98	LEA EAX,[LOCAL.55]	
010F1014	53	PUSH EBX	Arg2 => 0
010F1015	50	PUSH EAX	Arg1 => OFFSET LOCAL.55
010F1016	E8 65A2CEFF	CALL 000DB280	cnc3game.000DB280
010F1018	83C4 18	ADD ESP,18	
010F101E	53	PUSH EBX	pOverlapped => NULL
010F101F	8D45 70	LEA EAX,[LOCAL.1]	
010F1022	50	PUSH EAX	BytesReturned => OFFSET LOCAL.1
010F1023	6A 58	PUSH 58	OutSize = 88.
010F1025	8D45 F0	LEA EAX,[LOCAL.33]	
010F1028	50	PUSH EAX	
010F1029	8B45 6C	MOV EAX,DWORD PTR SS:[LOCAL.2]	OutBuffer => OFFSET LOCAL.33
010F102C	53	PUSH EBX	InSize => 0
010F102D	53	PUSH EBX	InBuffer => NULL
010F102E	68 20000700	PUSH 70020	IoControlCode = IOCTL_DISK_PERFORMANCE
010F1033	FF30	PUSH DWORD PTR DS:[EAX]	hDevice
010F1035	FF15 76BD4C0	CALL DWORD PTR DS:[<KERNEL32.DeviceIoControl>]	KERNEL32.DeviceIoControl

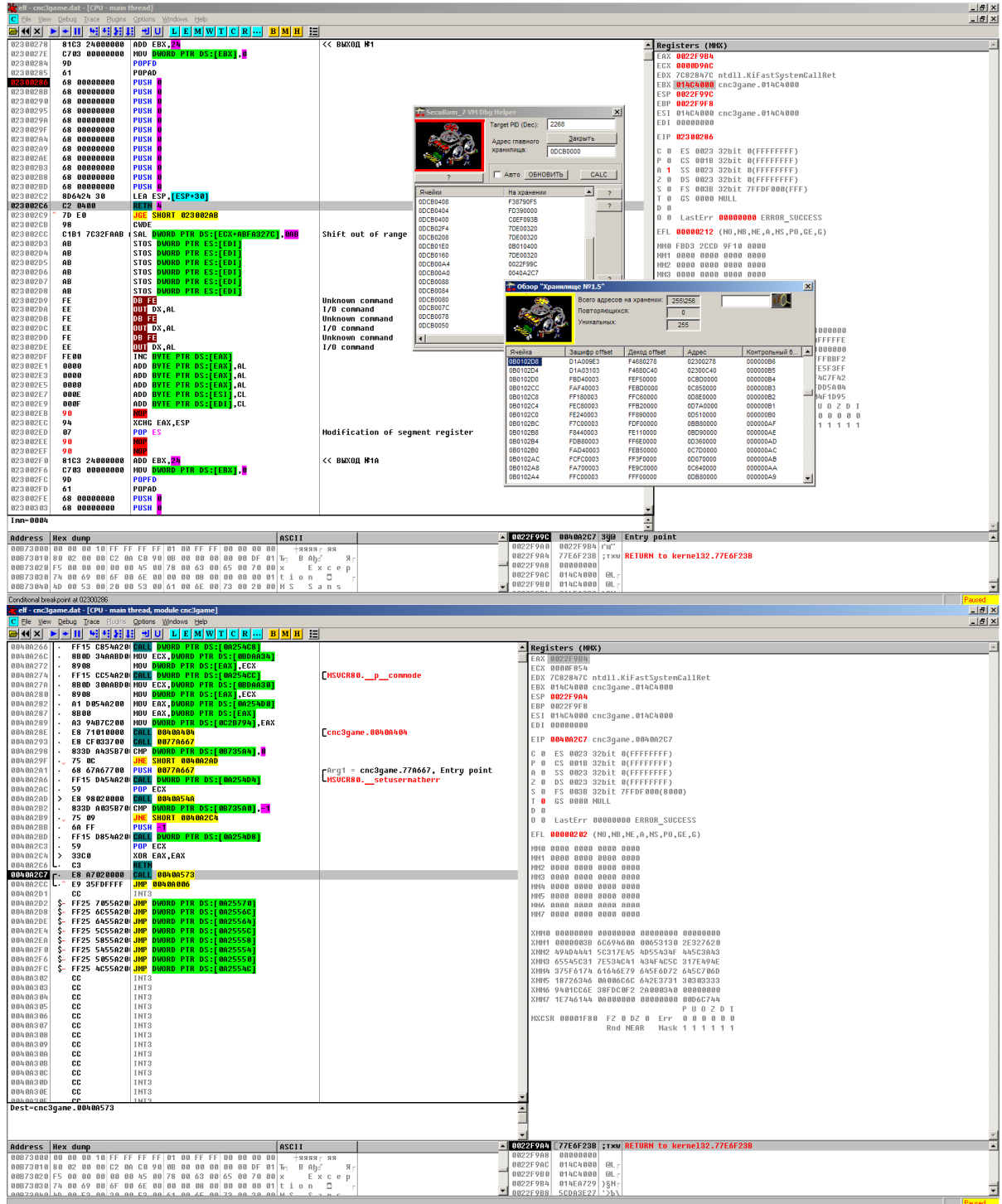
00E1FB7A	55	PUSH EBP	
00E1FB7B	8BEC	MOV EBP,ESP	
00E1FB7D	83EC 0C	SUB ESP,0C	
00E1FB80	57	PUSH EDI	
00E1FB81	33FF	XOR EDI,EDI	
00E1FB83	397D 08	CMP DWORD PTR SS:[ARG.1],EDI	
00E1FB86	74 64	JE SHORT 00E1FBEC	
00E1FB88	837D 0C 08	CMP DWORD PTR SS:[ARG.2],8	
00E1FB8C	72 5E	JB SHORT 00E1FBEC	
00E1FB8E	3BC7	CMP EAX,EDI	
00E1FB90	74 56	JE SHORT 00E1FBE8	
00E1FB92	FF30	PUSH DWORD PTR DS:[EAX]	
00E1FB94	E8 C1FFFFFF	CALL 00E1FB5A	[Arg1 => [ARG.EAX] cnc3game.00E1FB5A
00E1FB99	3BC7	CMP EAX,EDI	
00E1FB9B	74 48	JE SHORT 00E1FBE8	
00E1FB9D	53	PUSH EBX	
00E1FB9E	56	PUSH ESI	
00E1FB9F	8D70 08	LEA ESI,[EAX+8]	
00E1FBA2	32DB	XOR BL,BL	
00E1FBA4	8365 FC 00	AND DWORD PTR SS:[LOCAL.1],00000000	
00E1FBA8	8D45 F4	LEA EAX,[LOCAL.3]	
00E1FBAB	50	PUSH EAX	
00E1FBAC	FF15 DABC4C	CALL DWORD PTR DS:[<&KERNEL32.QueryPerformanceCounter>]	
00E1FBB2	8A45 F4	MOV AL,BYTE PTR SS:[LOCAL.3]	
00E1FBB5	8B4D FC	MOV ECX,DWORD PTR SS:[LOCAL.1]	
00E1FBB8	24 01	AND AL,01	
00E1FBBA	D2E0	SHL AL,CL	
00E1FBBC	0AD8	OR BL,AL	
00E1FBBE	FF45 FC	INC DWORD PTR SS:[LOCAL.1]	
00E1FBC1	837D FC 08	CMP DWORD PTR SS:[LOCAL.1],8	
00E1FBC5	7C E1	JL SHORT 00E1FBA8	
00E1FBC7	8B1C3E	MOV BYTE PTR DS:[EDI+ESI],BL	
00E1FBCA	47	INC EDI	
00E1FBCB	83FF 08	CMP EDI,8	
00E1FBCE	7C D2	JL SHORT 00E1FBA2	
00E1FBD0	8B45 08	MOV EAX,DWORD PTR SS:[ARG.1]	
00E1FBD3	6A 08	PUSH 8	Arg3 = 8
00E1FBD5	83C0 04	ADD EAX,4	Arg2
00E1FBD8	56	PUSH ESI	Arg1
00E1FBD9	50	PUSH EAX	cnc3game.00DDA5C0
00E1FBDA	E8 E1A9FBFF	CALL 00DDA5C0	
00E1FBDF	83C4 0C	ADD ESP,0C	
00E1FBE0	55	PUSH EBP	

Товарищ командир, задание выполнено! ОЕР была успешно достигнута!

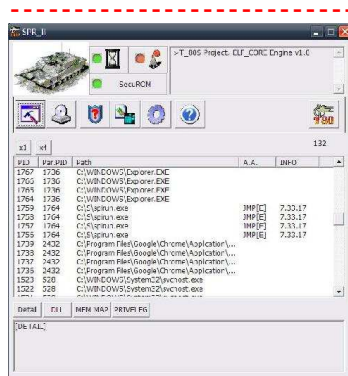
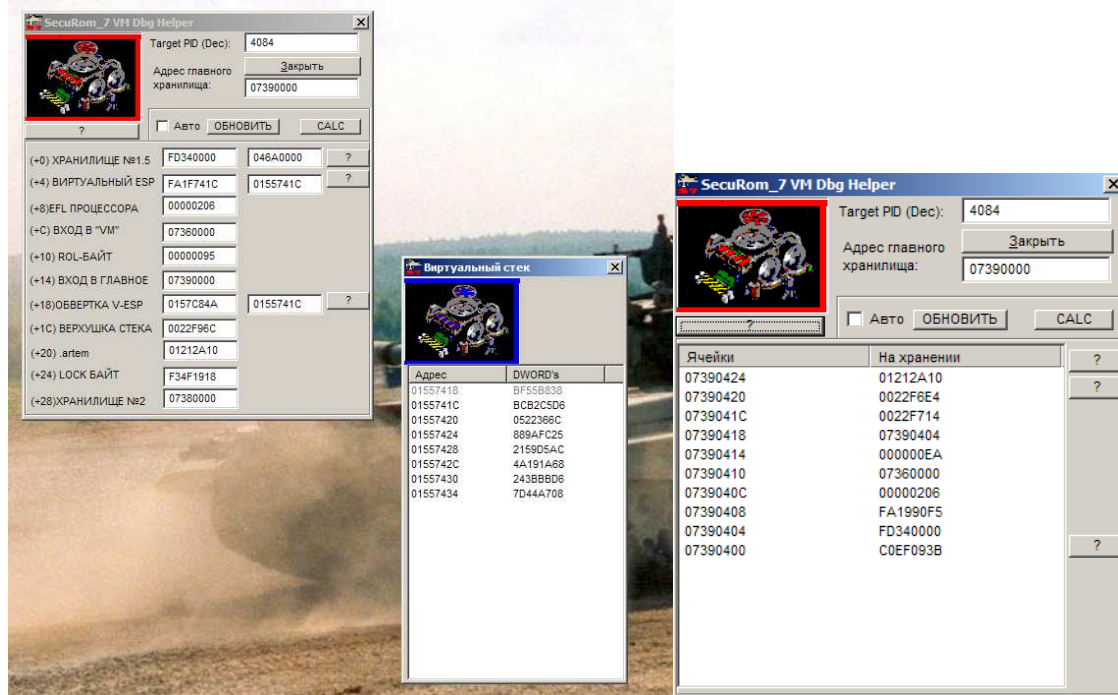
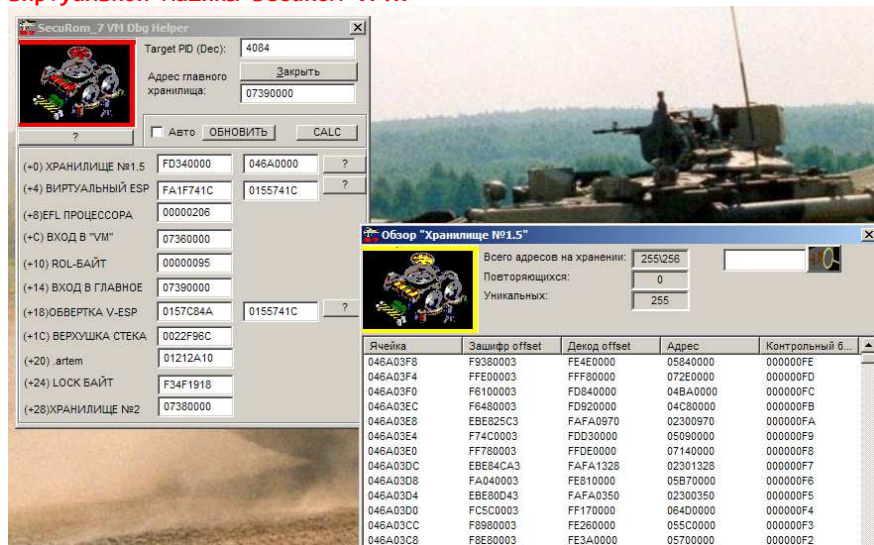
Адрес: 0040A2C7. 52 выход из VM.

На первом изображении можно увидеть:

1. Островки выхода способов №1 и 1А находятся рядом. Нередкая ситуация.
2. SecuRom_7 Profiler: Адрес островка в №1.5 и адрес ОЕР в ячейке +0А Главного хранилища



SecuRom_Profiler v1.0 – наш первый удачный шаг по визуализации параметров работы виртуальной машины SecuROM v7.x



Будет ли второй – зависит только Вас! ;)