

генетический распаковщик своими руками

крис касперски

первое, что делает исследователь, когда берет в руки программу: смотрит упакована она или нет. И тут же бежит искать подходящий распаковщик, поскольку, большинство программ все-таки упакованы, а готовые распаковщики есть не для всех. **Предлагаем вам** алгоритм универсального распаковщика, "снимающего" даже сложные полиморфные протекторы.

введение

Как хитро я вас обманул! Ни о генах, ни о хромосомах речь здесь не идет. Искусственный интеллект отдыхает на задворках истории. genetic – в переводе с греческого "общий". Генетический распаковщик — универсальный распаковщик общий для всех упаковщиков/протекторов. Кстати говоря, "General Motors" это не "двигатели для генералов", а двигатели вообще. Почувствуйте разницу!

Так о чем же мы будем говорить? Все больше и больше программ распространяется в упакованном виде, и совсем не потому что так они занимают меньше места или **загружаются быстрее**, как это обещают разработчики упаковщиков (на самом деле, упаковка приводит к значительному перерасходу памяти и тормозит всю систему. Подробности — в статье "паковать или не паковать", электронную копию которой можно бесплатно скачать с <ftp://nezumi.org.ru>). Усложнить взлом, затруднить анализ — вот для чего на самом деле используются упаковщики, в конечном счете превратившиеся в протекторы.

Одно и то же оружие используется как против хакеров, так и **против** легальных исследователей. Вирусы, черви, троянские кони практически всегда обрабатываются самыми последними версиями полиморфных протекторов, для которых распаковщики еще не написаны, и прежде чем антивирус научится распознавать заразу, программистам приходится проделать колоссальный объем работы. Существуют и другие мотивы, побуждающие распаковать программу: многие протекторы отказываются работать в присутствии пассивных отладчиков, не дружат с виртуальными машинами и эмуляторами...

Но не будем философствовать на моральные темы, лучше поговорим о деле. [А говорить мы будем преимущественно о PE-файлах NT-подобных системах \(включая 64-битные редакции оных\), в UNIX-мире упаковщиков намного меньше, но все-таки они есть \(взять хотя бы Shiva ELF-protector\). Если не углубляться в "политические" тонкости разногласий между UNIX и Windows, алгоритм распаковки — тот же самый, поэтому не будет на нем останавливаться, а поскорее ринемся в бой!](#)

предварительный анализ

Прежде чем приступать к распаковке, не помешает убедиться: а упакована ли программа вообще? Графическая версия IDA Pro поддерживает специальную панель "overview navigator", позволяющую визуализировать структуру дизассемблерного кода. В нормальных программах машинный код занимает значительную часть от общего объема, остальная часть приходится на данные (**см. рис. 1**).

Рисунок 1 структура нормальной, неупакованной программы

В упакованных программах кода практически нет, и все пространство занято данными (**см. рис. 2**), объединенными в один (или несколько) огромных массивов, которые IDA Pro не смогла дизассемблировать.

Рисунок 2 структура упакованной программы

Впрочем, визуализация — довольно расплывчатый критерий. Существует сотни причин, по которым IDA Pro отвергает неупакованный код или, напротив, рьяно бросается дизассемблировать упакованную секцию, получая кучу бессмысленных инструкций, пример которых приведен в **листинге 1**:

```
01010072h    imul     esi, [edx+74h], 416C6175h
```

```

01010079h    ins    byte ptr es:[edi], dx
0101007Ah    ins    byte ptr es:[edi], dx
0101007Bh    outsd
0101007Ch    arpl   [eax], ax
0101007Eh    push  esi
0101007Fh    imul  esi, [edx+74h], 466C6175h
01010086h    jb     short loc_10100ED
01010088h    add   gs:[ebx+5319Dh], cl
0101008Fh    add   [ebx], cl

```

Листинг 1 пример бессмысленного (зашифрованного/упакованного) кода

Осмысленность — вот главный критерий! Если дизассемблированный код выглядит "диким" и неясно, если в нем присутствует большое количество привилегированных инструкций — скорее всего он упакован/зашифрован. Или... это код, предназначенный совсем для другого процессора (например, байт-код виртуальной машины), а может... это хитрый обфускатор такой. В общем, вариантов много.

Попробуем взглянуть на таблицу секций (в hiew'e это делается так: <F8> – header, <F6> – ObjTbl). В нормальных программах присутствуют секции с именами .text, .CODE, .data, rdata, .rsrc и, что самое главное, виртуальный размер (VirtualSize) кодовой секции практически всегда совпадает с физическим (PhysSize). К примеру, в "Блокноте", входящем в штатную поставку NT, разница составляет всего 6600h – 65CAh == 36h байт, объясняемых тем, что виртуальная секция, в отличие от физической не требует выравнивания (см. рис.3).

Рисунок 3 раскладка секций неупакованного файла

А теперь упакуем наш файл с помощью ASPack (или аналогичного упаковщика) и посмотрим, что от этого изменится (см. рис 4):

Рисунок 4 раскладка секций упакованного файла

Ого! Сразу появились секции .aspack и .adata с именами говорящими самими за себя. Впрочем, имена секций можно и изменить — это не главное. Виртуальный размер секции .text (7000h) в два раза отличается от своего физического размера (3800h). А вот это уже говорит о многом! На самом деле, создателю ASPack'a было лень использовать "правильную" стратегию программирования вот он и выделил виртуальный размер заблаговременно. Существуют упаковщики, "трамбуящие" упакованные секции так, что виртуальный размер совпадает с физическим, а необходимая память динамически выделяется в процессе распаковки вызовом VirtualAlloc.

Еще можно попробовать измерять энтропию (меру беспорядка или степень избыточности) подопытного файла. Популярный hex-редактор HTE это умеет (см. рис.5). Чем больше значение энтропии, тем выше вероятность, что файл упакован и, соответственно, наоборот. Однако, этот прием срабатывает далеко не всегда...

Рисунок 5 подсчет энтропии с помощью редактора HTE

Так же хотелось бы обратить внимание на бесплатную утилиту PEiD (<http://peid.has.it/>), автоматически определяющую большинство популярных упаковщиков/протекторов по их сигнатурам (см. рис. 6), и даже пытающуюся их распаковать. Впрочем, с распаковкой дела обстоят неважно и лучше воспользоваться специализированными распаковщиками или написать свой собственный (чуть позже, мы покажем как).

Рисунок 6 внешний вид утилиты PEiD автоматически определяющий тип упаковщика/протектора

распаковка и ее альтернативы

Позвольте дурацкий вопрос. В стремлении как можно быстрее распаковать программу, мы зачастую даже не успеваем задуматься: а зачем?! Сразу же слышу возражения: мол, упакованную программу невозможно дизассемблировать. Да, невозможно, ну и что? Зато ее можно отлаживать, используя классический набор техник, подробно описанный в "фундаментальных основах хакерства" (электронная копия лежит на <ftp://nezumi.org.ru>). Вот тут

кто-то говорит, что некоторые упаковщики активно сопротивляются отладке! Что ж, попробуйте запустить soft-ice уже *после* того, как упаковщик уже отработает свое и на экране появиться главное окно программы (NT, в отличии от 9x, позволяет пускать soft-ice в любое время) или установите неофициальный патч IceExt (<http://stenri.pisem.net/>), скрывающий отладчик от большинства защит.

Ряд утилит типа LordPE (<http://mitglied.lycos.de/yoda2k/news.htm>) позволяют снимать с программы дампы после завершения распаковки и хотя полученный образ ехе-файла зачастую оказывается вопиюще некорректным и работающим нестабильно, для дизассемблирования он вполне пригоден, особенно если его использовать в связке с отладчиком, помогающим "подсмотреть" значения некоторых переменных на разных стадиях инициализации.

Ах да! Упакованную программу нельзя модифицировать, то есть накладывать на нее patch'и. То есть, даже после того, как мы найдем заветный Jx, отключающий защиту, мы не сможем модифицировать упакованный файл, поскольку никакого Jx там, естественно нет! Стоп! Кто говорит о взломе?! Этот прием используют не только хакеры! Некоторые вирусы так глубоко вгрызаются в программу (и каждый раз немного по разному), что "выломать" их оттуда, не нарушив функциональности программы — нереально, вернее, реально, но очень-очень трудно, гораздо легче найти jx ведущий к процедурам "размножения" и "кастрировать" их, то же самое относится и к функциями деструкции...

Вот на этот случай и были придуманы он-лайновые патчеры (on-line patchers), правящие программу "налету" непосредственно в оперативной памяти, минуя диск. ~~Кстати говоря, модификация исполняемого файла — дурной тон. Лучше (и честнее) ничего не править, а писать свой собственный генератор серийных номеров/ключевых файлов, основанный на информации, почерпнутой из корявого дампа или "живой" отладки. Ну это у ж совсем уж никуда уж. Мы же не журнал «ХаКер»~~

Возникает неоднозначная ситуация: с одной стороны, избежать распаковки в большинстве случаев все-таки возможно, но с другой — работать с распакованным файлом намного удобнее и комфортнее (хотя бы по чисто психологическим соображениям).

Поэтому, универсальный распаковщик никогда не помешает!

алгоритм распаковки

Как пишутся распаковщики? Здесь есть разные пути. Хакер может долго и мучительно изучать алгоритм работы распаковщика в отладчике/дизассемблере, а потом столь же долго и мучительно разрабатывать автономный распаковщик, корректно обрабатывающий исполняемый файл с учетом специфики конкретной ситуации.

А вот другой путь: запускаем программу на "живом" процессоре (или под эмулятором типа BOCHS), определяем момент завершения распаковки и тут же сохраняем образ памяти на диск, формируя из него PE-файл. В результате получится универсальный распаковщик, который мы и собираемся написать, но прежде — немного теории, для тех, кто еще не умеет распаковывать программы "руками".

в поисках ОЕР

Создание распаковщика начинается с поиска ОЕР (Original Entry Point — Исходная Точка Входа). Наш депротектор должен как-то определить момент завершения распаковки/расшифровки "подопытной" программы — когда распаковщик выполнил все, что хотел и уже приготовился передавать управление распакованной программе-носителю. Это самая сложная часть генетических распаковщиков, поскольку определить исходную точку входа в общем случае невозможно, вот и приходится прибегать к различным ухищрениям. Чаще всего для этого используется пошаговая трассировка, которой очень легко противостоять (чем большинство упаковщиков/протекторов и занимается). Немногим лучше с задачей справляются трассеры нулевого кольца. Справиться с грамотно спроектированным трассером средствами прикладного уровня (а большинство упаковщиков/протекторов работают именно там), практически невозможно, однако, разработка ядерных трассеров — слишком сложная задача для начинающих, поэтому лучше использовать готовый, разработанный группой легендарного Володи с не менее легендарного WASM'a. Правда, коммерческая составляющая до сих пор неясна и доступ к трассеру есть не у всех.

На самом деле, прибегать к трассировке никакой необходимости нет — одних лишь аппаратных точек останова для нашей задачи будет вполне достаточно. Иногда приходится слышать мнение, что работа с аппаратными точками останова возможна только из режима

ядра, то есть из драйвера. Это неверно. В "записках мышья" (электронную копию которой можно свободно утянуть с [ftp://nezumi.org.ru](http://nezumi.org.ru)) показано как это сделать и с прикладного уровня, даже без прав администратора!

На первом этапе в качестве основного экспериментального средства мы будем использовать "Блокнот" из комплекта поставки NT, сжатый различными упаковщиками (которые мы только сможем найти) и знаменитый отладчик soft-ice. Кодирование последует потом. Чтобы писать красиво и по сто раз не переписывать уже написанное и отлаженное, необходимо иметь соответствующий нехитрый разговорное слово боевой опыт, для которого нам и понадобится soft-ice.

дамп живой программы

Самый простой (и самый популярный) способ борьбы с упаковщиками — *снятие дампа задолго после завершения распаковки*. Дождавшись появления главного окна программы, хакер сбрасывает ее дамп, превращая его в PE-файл. Иногда он работает, но чаще всего нет. Попробуем разобраться почему. Возьмем классический "Блокнот" (которое в защищенности не обвинишь!) и, *не упаковывая его никакими упаковщиками*, попробуем снять дамп с помощью одного из самых лучших дамперов: Lord PE Deluxe (см. рис. 7).

Рисунок 7 снятие дампа с работающего "Блокнота"

Процесс дампирования проходит успешно, и образовавшийся файл даже запускается (см. рис 8), но... оказывается не совсем работоспособен! Исчез заголовок окна и все текстовые надписи в диалогах! Если мы не сумели снять дамп даже с такого простого приложения, как Блокнот, то с настоящими защитами нам и подавно не справиться! В чем же дело?!

Рисунок 8 нормально работающий "Блокнот" (сверху) и тот же самый "Блокнот" после снятия дампа — все текстовые строки исчезли

Расследование показывает, что исчезнувшие текстовые строки хранятся в секции ресурсов и, стало быть, обрабатываются функцией LoadString. Загружаем оригинальный potepad.exe в IDA Pro (или любой другой дизассемблер по вкусу) и находим цикл, считывающий строки посредством функции LoadStringW (суффикс 'W' означает, что мы имеем дело со строками в формате Unicode). Ага, вот этот цикл! Рассмотрим его повнимательнее (будьте уверены, тут есть чему поучиться):

```
01004825h     mov     ebp, ds:LoadStringW    ; ebp - указатель на LoadStringW
0100482Bh     mov     edi, offst off_10080C0 ;указатель на таблицу ресурсов
01004830h
01004830h loc_1004830:                  ; CODE XREF: sub_10047EE+65;j
01004830h     mov     eax, [edi]            ; грузим очередной указатель на uID в eax
01004832h     push   ebx                   ; nBufferMax (максимальная длина буфера)
01004833h     push   esi                   ; lpBuffer (указатель на буфер)
01004834h     push   dword ptr [eax]       ; передаем извлеченный uID функции
01004836h     push   [esp+0Ch+hInstance]   ; hInstance
0100483Ah     call   ebp ; LoadStringW     ; считываем очередную строку из ресурса
0100483Ch     mov     ecx, [edi]           ; грузим тот же самый uID в ecx
0100483Eh     inc     eax                   ; увеличиваем длину считанной строки на 1
0100483Fh     cmp     eax, ebx             ; ?строка влезает в буфер?
01004841h     mov     [ecx], esi          ; сохраняем указатель на буфер поверх
01004841h                                     ; старого uID (он больше не понадобится)
01004841h
01004843h     lea    esi, [esi+eax*2]      ; позиция для следующей строки в буфере
01004846h     jg     short loc_100488B    ; если буфер кончился, то это облом
01004848h     add     edi, 4              ; переходим к следующему uID
0100484Bh     sub     ebx, eax            ; уменьшаем свободное место в буфере
0100484Dh     cmp     edi, offst off_1008150 ; ?конец таблицы ресурсов?
01004853h     jl     short loc_1004830    ; мотаем цикл пока не конец ресурсов
```

Листинг 2 хитро оптимизированный цикл чтения строковых ресурсов

В переводе на русский это звучит так: "Блокнот" берет очередной идентификатор строки из таблицы ресурсов, загружает строку, размещая ее в локальном буфере и сохраняя полученный указатель поверх... самого идентификатора, который уже не нужен! Это классический трюк с повторным использованием освободившихся переменных, известный еще

со времен первых PDP, если не раньше. А вы все Microsoft ругаете! "Блокнот" писал не глупый хакер, бережно относящийся в системным ресурсам и к памяти, в частности.

Для нас же это означает, что снятый с "живой" программы дампы будет неполноценным! Вместо реальных идентификаторов строк в секции ресурсов окажутся указатели на память, направленные в "космос", но ведь загрузчик ресурсов, приведенный [в листинге 2](#), ожидает *реальных* идентификатор и к встрече с указателями морально не готов! Это и есть та причина, по которой PE-файл, изготовленный из дампа, ведет себя неправильно.

Впрочем, бывает и хуже. Во многих программах встречается конструктор вида:

```
void *p = 0; // глобальная инициализированная переменная
// (на самом деле присваивать нуль необязательно
// поскольку все глобальные переменные "сами"
// обнуляются при запуске программы)

if (!p) p = malloc(BUF_SIZE); // выделить память, если она еще не выделена
```

Листинг 3 "защита" от дампа живых программ

Очевидно, если снять дампы с программы *после* выделения памяти, то при запуске PE-файла, полученного из такого дампа, память навсегда перестанет выделяться, а в глобальной переменной p окажется указатель, доставшийся ей в "наследство" от предыдущего запуска, однако, соответствующий регион памяти выделен не будет и программа либо рухнет, либо залезет в чужие данные, устроив там настоящий переполох!

Сформулируем главное правило: *снимать дампы с программы можно только в точке входа!* Остается разобраться: как эту точку входа отловить.

поиск стартового кода по сигнатурам в памяти

Начинающие программисты считают, что выполнение программы начинается с функции main (на Си/Си++) или ~~с ключевого слова процедуры~~ begin (на Паскале). Первым всегда вызывается стартовый код (start-up code), устанавливающий первичный обработчик структурных исключений, инициализирующий RTL, вызывающий GetModuleHandleA для получения дескриптора текущего модуля и т. д. То же самое относится к DLL, *главной функцией которой является DllMain.*

Существует огромное множество разновидностей start-up кодов, выбираемых компилятором в зависимости от типа программы, ключей компиляции и т. д. Исходные тексты стартовых кодов, как правило, открыты. В частности, Microsoft Visual C++ хранят их в каталоге \Microsoft Visual Studio\VC98\CRT\SRC под именами crt*.*. Всего их около десятка.

Ниже ([см. листинг 4](#)), в качестве примера, приведен один из стартовых кодов компилятора DELPHI:

```
CODE:00401EE8 start proc near
CODE:00401EE8 push ebp
CODE:00401EE9 mov ebp, esp
CODE:00401EEB add esp, 0FFFFFF0h
CODE:00401EEE mov eax, offset dword_401EB8
CODE:00401EF3 call @Sysinit@@InitExe$qqrpv ; Sysinit::InitExe()
CODE:00401EF8 push 0
...
CODE:00401D9C @Sysinit@@InitExe$qqrpv proc near ; CODE XREF: start+B1p
CODE:00401D9C push ebx
CODE:00401D9D mov ebx, eax
CODE:00401D9F xor eax, eax
CODE:00401DA1 mov ds:TlsIndex, eax
CODE:00401DA6 push 0 ; lpModuleName
CODE:00401DA8 call GetModuleHandleA
CODE:00401DAD mov ds:dword_4036D8, eax
CODE:00401DB2 mov eax, ds:dword_4036D8
CODE:00401DB7 mov ds:dword_40207C, eax
CODE:00401DBC xor eax, eax
CODE:00401DBE mov ds:dword_402080, eax
CODE:00401DC3 xor eax, eax
CODE:00401DC5 mov ds:dword_402084, eax
CODE:00401DCA call @SysInit@_16395 ; SysInit::_16395
CODE:00401DCF mov edx, offset unk_402078
CODE:00401DD4 mov eax, ebx
CODE:00401DD6 call sub_4018A4
```

Листинг 4 пример стартового кода на DELPHI

Собрав внушительную коллекцию стартовых кодов (или выдернув ее из IDA Pro) мы легко найдем ОЕР простым сканированием дампа, снятого с "живой" программы. Загружаем полученный дамп в hiew и, перебирая сигнатуры всех стартовых кодов, находим который из них "наш". Первый байт стартового кода и будет точкой входа в программу — ОЕР.

Запоминаем ее адрес (в моей версии "Блокнота" она расположена по смещению 01006420h), загружаем упакованную программу в отладчик и устанавливаем аппаратную точку на исполнение по данному адресу — "ВРМ 1006420 X" (**внимание!** по умолчанию отладчик ставит точку останова на чтение/запись, что совсем не одно и то же!). Оп-ля! Обогнув распаковщик, отладчик (или наш дампер, который мы чуть позже отважимся написать) всплывает непосредственно в ОЕР! Самое время снимать дампы!

(Помниться мне, в далекие времена DOS существовал универсальный распаковщик (название забыл), который запускал программу в режиме трассировки и определял переход к исполнению программы по коду в окрестности точки входа — более-менее специфичному для каждого из компиляторов. Затем дампил и почти всегда получал рабочий ехе-шник. Правда, если запаковывали нечто, написанное на ассемблере- начинались проблемы — прим. редактора).

пара популярных, но неудачных способов: GetModuleHandleA и fs:0

Библиотека сигнатур это, конечно, хорошо, но слишком хлопотно. Новые версии компиляторов выходят чуть ли не каждую декаду, к тому же разработчики зачастую слегка модифицируют start-up код, ослепляя сигнатурный поиск. Что тогда? Достаточно часто в этих случаях рекомендуется установка точек останова на API-функцию GetModuleHandleA ("ВРМ GetModuleHandleA X") и на фильтр структурных исключений ("ВРМ FS:0"). Но это не очень хорошие способы и ниже будет показано почему.

Начнем с функции GetModuleHandleA, которая присутствует практически в каждом стартовом коде (исключая некоторые ассемблерные программы?). Нажимаем <Ctrl-D> для вызова soft-ice, даем команду "ВРМ GetModuleHandleA X" (можно так же дать "ВРХ GetModuleHandleA", но в этом случае soft-ice внедрит в начало GetModuleHandleA программную точку останова CCh, легко обнаруживаемую многими защитными механизмами) и запускаем наш подопытный "Блокнот", предварительно зажевав его любимым не слишком навороченным упаковщиком, например ASPack или UPX. Поскольку, установка точки останова носит глобальный характер, все программы, обращающие к GetModuleHandleA, будут вызывать всплывтия отладчика. Внимательно смотрите на имя программы, отображаемое отладчиком в правом нижнем углу (см. рис 9) — если это не наша программа, жмем "x" или <Ctrl-D> для выхода из отладчика.

После серии ложных срабатываний, в углу наконец-то появляется наш заветный NOTEPAD. Самое время сказать отладчику "P RET", чтобы выбраться из функции в непосредственно вызывающий ее код, однако... этим вызывающим кодом оказывается отнюдь не окрестности исходной точки входа, а код самого распаковщика! И до ОЕР нам еще трассировать и трассировать. Определить нашу дислокацию поможет карта памяти. Даем команду "МАР32" и смотрим, что скажет отладчик:

Рисунок 9 функция 1010295h:CALL [EBP+F4Dh] это на самом деле GetModuleHandleA, вызываемая таким заковыристым образом.

Файл notepad.exe состоит из нескольких секций: .text (сжатый код исходной программы), .data (сжатые данные исходной программы), .rsrc (сжатые ресурсы исходной программы), .aspack (код распаковщика) и .adata (данные распаковщика). Секция кода исходной программы заканчивается на адресе 100800h и все, что лежит ниже — ей уже не принадлежит. В нашем случае функция GetModuleHandleA, вызывается кодом, расположенным по адресу 1010295h, который, как нетрудно установить, принадлежит, секции .aspack — то есть непосредственно самому распаковщику и к ОЕР никакого отношения не имеет (помимо распаковщика, функцию GetModuleHandleA могут вызывать и динамические библиотеки, подключаемые статической компоновкой, то есть через секцию импорта).

Выходим из отладчика, нажимая <Ctrl-D> до тех пор, пока вызов GetModuleHandleA не окажется внутри секции .text (если лень **нажимать одну и ту же клавишу помногу разгонять жаргон — клавиатуру**, можно установить условную точку останова в стиле "ВРМ GetModuleHandleA X IF EIP < 0x1008000").

Ага! Вот, наконец, появилось что-то похожее на истину (см. рис. 10).

Рисунок 10 подлинный вызов GetModuleHandleA из окрестностей OEP

Данный вызов действительно подлинный и, прокручивая экран окна CODE вверх, нам остается всего лишь найти стандартный пролог PUSH EBP/MOV EBP,ESP или RET, которым заканчивается предшествующая процедура. С высокой степенью вероятности это и будет OEP, на которую можно поставить аппаратную точку останова, затем перезапустить программу еще раз (не забыв при этом удалить точку останова на GetModuleHandleA) и в момент всплытия отладчика сделать программе дамп. При этом следует помнить, что далеко не всегда GetModuleHandleA вызывается непосредственно из самого стартового кода! В частности DELPHI помещает GetModuleHandleA в функцию @Sysinit@@InitExe\$qqrpv, а вот она-то уже и вызывается стартовым кодом! (см. листинг 4).

Это значит, что для достижения OEP, с момента "ловли" GetModuleHandleA нам придется раскрутить стек (команда "STACK" в soft-ice), поднявшись на один или два уровня вверх. Но на сколько конкретно подниматься? Чтобы ответить на этот вопрос, нам необходимо исследовать состояние стека на момент вызова файла.

Остановив отладчик в точке входа EP (не путать с OEP, которую еще только предстоит найти), даем команду "D ESP" и смотрим (чтобы данные отображались не байтами, а двойными словами, необходимо сказать отладчику "dd"):

```
:d esp
0023:0006FFC4 77E87903 FFFFFFFF 0011F458 7FFDF000 .y.w....X.....

:u *esp
0023:77E87903 E9470A0300 JMP 77EB834F

:u 77EB834F
0023:77EB834F 50 PUSH EAX
0023:77EB8350 E87A83FDFF CALL KERNEL32!ExitThread

:d fs:0
0038:00000000 0006FFE0 00070000 0006E000 00000000 .....
```

Листинг 5 состояние стека на момент вызова файла

Адрес 77E87903h (для наглядности выделенный подчеркиванием), указывает куда-то внутрь KERNEL32.DLL и команда ("U *ESP") позволяет узнать куда. Ага, здесь расположен безусловный переход на PUSH EAX/CALL KERNEL32!ExitThread. То есть, на процедуру завершения программы. Со времен MS-DOS не так уж и многое изменилось. Там тоже на вершине стека лежал адрес возврата на exit, поэтому завершать работу программы можно не только через API, но и простой инструкцией RETN. Если стек сбалансирован, она сработает правильно. Кстати, пара адресов на вершине стека 77E87903h/FFFFFFFh до боли напоминает терминирующий обработчик структурных исключений (терминирующий – то есть последний в цепочке), которым она по сути и является ибо содержимое двойного слова FS:[00000000h] указывает как раз на нее.

Сформулируем еще одно правило: *признаком OEP является ситуация FS:[00000000h] == ESP && *ESP == 77E87903h* (естественно, этот адрес варьируется от системы к системе и на каждой из них должен вычисляться индивидуально!).

Кстати, раз уж мы затронули структурные исключения, не мешает познакомиться с ними поближе. Практически каждый стартовый код начинает свою деятельность с установки своего собственного фильтра структурных исключений, записывая в ячейку FS:0 новое значение.

```
.text:01006420 55 push ebp ; ← исходная точка входа
.text:01006421 8B EC mov ebp, esp
.text:01006423 6A FF push 0FFFFFFFh
.text:01006425 68 88 18 00 01 push dword_1001888
.text:0100642A 68 D0 65 00 01 push loc_10065D0 ; ← "наш" обработчик
.text:0100642F 64 A1 00 00 00 00 mov eax, fs:0 ; ← берем старый фильтр
.text:01006435 50 push eax
.text:01006436 64 89 25 00 00 00+ mov fs:0, esp ; ← ставим новый фильтр
```

Листинг 6 установка нового фильтра структурных исключений в стартовом коде

Попробуем отловить это событие?! А почему бы и нет! Только надо учесть, что в отличии от API-функций, допускающих установку глобальных точек останова, точка останова на FS:0 должна устанавливаться из контекста подопытного приложения. А как в него попасть? Необходимо либо остановиться в точке входа в распаковщик (см. врезку "что делать если отладчик проскакивает точку входа в распаковщик"), либо установить точку останова на GetModuleHandleA, дождаться первого ложного всплытия отладчика, совершенного в контексте *нашего* приложения (в правом нижнем углу горит notepad), затем удалить точку останова на GetModuleHandleA и установить точку останова на фильтр структурных исключений: "BPM FS:0". Необходимо сразу подготовить себя к *огромному* количеству ложных срабатываний (структурные исключения активно использует не только распаковщик, но и сама операционная система), и чтобы закончить отладку до конца сезона, необходимо воспользоваться условными точками останова, ограничив диапазон срабатываний отладчика секцией .text. В нашем случае это будет выглядеть так: "BPM FS:0 IF EIP < 0X100800" (предыдущую точку останова перед этим следует удалить), после чего можно смело отправиться на кухню и варить пакетный рис, поскольку за компьютером нам делать уже нечего.

Рисунок 11 ловля OEP на обработчик структурных исключений

Даже незащищенный NOTEPAD.EXE вывалил окно отладчика спустя... 713.34 секунды (см. рис. 11), что немногим больше 12 минут. Это же целая вечность для процессора! (Для справки: эксперименты проводились на VM Ware, запущенным под древним Pentium-III 733 MHz, современные процессоры справляются с этой ситуацией намного быстрее — на то они и современные).

Но как бы там ни было, исходная точка входа найдена! Вот она, расположенная по адресу 10006420h! Как говорить, бери и властвуй!

побочные эффекты упаковщиков или почему не работает VirtualProtect

Очевидно, чтобы распаковывать кодовую секцию программы, упаковщик должен иметь разрешение на запись (отсутствующее по умолчанию), а по завершению распаковки восстанавливать исходные атрибуты, чтобы программа выглядела так, как будто ее не упаковали. Помимо кодовой секции, необходимо восстановить атрибуты секции .rdata, доступной, как и следует из ее названия, только на чтение. Для манипуляций с атрибутами страниц Windows предоставляет функцию VirtualProtect. И это практически единственный путь, которым можно что-то сделать (не считая VirtualAlloc с флагом MEM_COMMIT с которым связана куча проблем).

Логично предположить, что функция VirtualProtect должна вызываться после завершения распаковки в непосредственной близости от передачи управления на OEP. Означает ли это, что установив точку останова на VirtualProtect мы...

А вот и нет! Из трех наугад взятых упаковщиков: ASPack, PE-Compact и UPX, только PE-compact вызывал VirtualProtect для манипуляций с атрибуты секций (да и то совсем для других целей), а все остальные оставляли их открытыми на запись даже после завершения распаковки! Вопиющее нарушение спецификаций и неуважение к нормам хорошего поведения!

Смотрите сами:

Рисунок 12 исходный "блокноте"

В неупакованном "Блокноте" (см. рис. 12) секция .text имеет права на чтение и исполнение (executable/readable/code), секция .data - чтение и запись (writable/readable/инициализированные данные), секция .rsrc - только на чтение (readable/инициализированные данные). Короче, все как в приличных домах.

Рисунок 13 "блокнот", упакованный ASPack'ом

Теперь упакуем файл ASPack'ом. Запустим его и, дождавшись окончания распаковки (на экране появляется главное окно программы), снимем с него дамп (см. рис 13). Вот это номер! *Все* секции имеют атрибут C0000040h, то есть такой же как у секции данных — с правом на запись, но без прав исполнения. А вот это уже нехорошо! Мало того, что в кодовую секцию теперь может писать кто угодно, так еще на процессорах с NX/XD битами файл, обработанный

ASPack'ом исполняться не будет!!! Система выбросит грозное предупреждение с намеком на вирус и большинство пользователей просто сотрут такой файл от греха подальше, а это значит, что создатель упакованной программы потеряет клиента (и правильно — незачем было паковать!). К тому же, в памяти по-прежнему болтаются секции .aspack и .adata которые упаковщик не удосужился "подчистить". Это, хоть и не смертельно, но все-таки не аккуратно.

Рисунок 14 "блокнот", упакованный PE-compact

А вот PE-compact преподносит нам настоящий сюрприз (см. рис. 14)! Секции .text и .data объединены в одну большую секцию с атрибутами (executable/readable/writable/code/инициализированные данные), а секция .rsrc, которая не должна (по спецификации!) иметь права на запись, его все-таки имеет. Как говорится, с барского плеча ничего не жалко. Зато свои собственные секции PE-compact подчищает весьма аккуратно.

Рисунок 15 "блокнот", упакованный UPX'ом

Что же касается UPX'a (см. рис. 15), то в целом он движется по пути PE-compact'a: кодовую секцию с секций данных он не объединяет, но атрибуты им назначает те же самые (чтение/запись/исполнение), в добавок ко всему "забывая" вернуть им исходные имена (которые впрочем, никто не проверяет). Исключение совсем секция .rsrc — некоторые программы теряют способность находить ресурсы, если эта секция названа как-то иначе.

Вердикт — все упаковщики **несомненное злогадость опять же, давайте мягче. Дети же читают :-)**! Ни один из них не соответствует спецификациям на PE-файл, зато побочных эффектов от упаковки — хоть отбавляй! (удивительно как упакованные файлы еще ухитряются работать). Отсюда — устанавливать точку останова на VirtualProtect никакого смысла нет.

код	значение
00000004h	16-битные смещения кода
00000020h	code
00000040h	инициализированные данные
00000080h	неинициализированные данные
00000200h	комментарии или другая вспомогательная информация
00000400h	оверлей
00000800h	не подлежит загрузке в память
00001000h	Comdat.
00500000h	выравнивание по умолчанию
02000000h	может быть выброшено из памяти
04000000h	not cachable.
08000000h	not pageable.
10000000h	shareable
20000000h	executable
40000000h	readable
80000000h	writable

Таблица 1 расшифровка атрибутов секций PE-файла

универсальный прием поиска OEP, основанный на балансе стека

Вот мы и подобрались к самому интересному и универсальному способу определения OEP, который к тому же легко автоматизировать. Упаковщик (даже если это не совсем корректный упаковщик) просто обязан после распаковки восстановить стек, в смысле вернуть регистр ESP на его законное место, указывающее на первичный фильтр структурных исключений, устанавливаемый системой по умолчанию (см. листинг 5). Некоторые упаковщики еще восстанавливают и регистры, но это уже необязательно.

Возьмем, к примеру, тот же ASPack и посмотрим в его начало (см. листинг 7):

```
:u eip
001B:01010001 60          PUSHAD          ; сохранить все регистры в стеке
001B:01010002 E803000000    CALL  0101000A ; определить текущий EIP
```

```

001B:01010007 E9EB045D45 JMP 465E04F7 ; спрятанные JMPS/POP EBP/INC EBP
001B:0101000C 55 PUSH EBP ; \ "эмуляция" команды
001B:0101000D C3 RET ; / jmp 01010008h

```

Листинг 7 точка входа в распаковщик ASPack

Замечательно! Первая же команда сохраняет все регистры в стеке. Очевидно, что непосредственно перед передачей управления на OEP они будут восстановлены командой POPAD, выполнение которой очень легко отследить, установив точку останова на двойное слово, лежащее выше верхушки стека: "BPM ESP - 4".

Результат превосходит все ожидания (см. листинг 8):

```

001B:010103AF 61 POPAD ; ← на этой команде отладчик всплывает
001B:010103B0 7508 JNZ 010103BA (JUMP ↓)
001B:010103B2 B801000000 MOV EAX,00000001
001B:010103B7 C20C00 RET 000C
001B:010103BA 6820640001 PUSH 1006420 ; ← адрес OEP
001B:010103BF C3 RET ; ← передача управления на OEP

```

Листинг 8 передача управления на OEP

Распаковав программу, ASPack заботливо выталкивает сохраненные регистры из стека, вызывая всплытие отладчика и мы видим тривиальный код передающий управление на OEP "классическим" способом через PUSH offset OEP/RET. Поиск исходной точки входа не затратил и десятка секунд! Ну разве не красота?!

А теперь возьмем UPX и проверим, удастся ли нам повернуть этот трюк и над ним? Ведь мы же претендуем на универсальный прием!

```

001B:01011710 60 PUSHAD ; сохранить все регистры в стеке
001B:01011711 BE00D00001 MOV ESI,0100D000
001B:01011716 8DBE0040FFFF LEA EDI,[ESI+FFFF4000]
001B:0101171C 57 PUSH EDI

```

Листинг 9 так начинается UPX

Вот он, уже знакомый нам PUSHAD (см. листинг 9), сохраняющий все регистры в стеке и восстанавливающий их непосредственно перед передачей управления на OEP. Даем команду "BPM ESP-4" и выходим из отладчика, пока он не всплывет (см. листинг 10):

```

001B:0101185E 61 POPAD
001B:0101185F E9BC4BFFFF JMP 01006420 (JUMP ↑)

```

Листинг 10 так UPX передает управление на OEP

А вот и отличия! Передача управления осуществляется командой JMP 1006420h, где 1006420h — исходная точка входа. Похоже, что все упаковщики работают по одному и тому же алгоритму и ломаются с реактивной скоростью.

Но не будем спешить! Возьмем PE-сопрат и проверим свою догадку на нем.

```

001B:01001000 B874190101 MOV EAX,01011974
001B:01001005 50 PUSH EAX
001B:01001006 64FF3500000000 PUSH DWORD PTR FS:[00000000]
001B:0100100D 64892500000000 MOV FS:[00000000],ESP

```

Листинг 11 точка входа в файл, упакованный PE-сопрат

Плохо дело (см. листинг 11)! PE-сопрат никаких регистров вообще не сохраняет, а PUSH EAX используется только затем, чтобы установить свой обработчик структурных исключений. Тем не менее, на момент завершения распаковки указатель стека должен быть восстановлен, следовательно, точка останова на "BPM ESP-4" все-таки может работать....

```

001B:77F8AF78 FF7304 PUSH DWORD PTR [EBX+04]
001B:77F8AF7B 8D45F0 LEA EAX,[EBP-10]
001B:77F8AF7E 50 PUSH EAX

```

Листинг 12 первое срабатывание точки останова на esp-4

Так, ну это срабатывание (см. листинг 12) явно ~~левое~~ ложное (судя по EIP, равному 77F8AF78h, мы находимся где-то внутри KERNEL32.DLL, использующим стек для нужд производственной необходимости), нажимаем <Ctrl-D> не желая здесь больше задерживаться.

```

001B:010119A6 55          PUSH  EBP    ; ← эта команда вызывает всплытие
001B:010119A7 53          PUSH  EBX
001B:010119A8 51          PUSH  ECX
001B:010119A9 57          PUSH  EDI

```

Листинг 13 Кузьмич?! Где-то это я?

Следующее всплытие отладчика (см. листинг 13), ничуть не более осмысленное, чем предыдущее. Ясно только одно: в стек заталкивается регистр EBP вместе с кучей других регистров. Судя по всему, это распаковщик сохраняет их с одной лишь ему ведомой целью. **Жмем** <Ctrl-D> и ждем дальше — что нам еще покажут?

```

:u eip-1
001B:01011A35 5D          POP   EBP
001B:01011A36 FFE0       JMP   EAX (01006420h)

```

Листинг 14 переход на OEP

А вот на этот раз (см. листинг 14) нам повезло! Регистр EBP выталкивается из стека и вслед за этим осуществляется переход на OEP посредством команды JMP EAX. Мы все-таки достигли ее!!! Вот только ложные срабатывания напрягают. Это мы, опытные хакеры, можем "визуально" отличить где происходит передача на OEP, а где нет. С автоматизацией в этом плане значительно сложнее — у компьютера интуиция отсутствует напрочь. А ведь мы всего лишь развлекаемся с давно побежденными упаковщиками... Про борьбу с протекторами речь еще не идет.

Возьмем более серьезный упаковщик **FSG 2.0** by bart/xt (<http://xtreeme.prv.pl/>, <http://www.wasm.ru/baixado.php?mode=tool&id=345>) и начнем его пытаться.

```

001B:01000154 8725B4850101 XCHG  ESP, [010185B4]
001B:0100015A 61          POPAD
001B:0100015B 94          XCHG  EAX, ESP
001B:0100015C 55          PUSH  EBP
001B:0100015D A4          MOVSB

```

Листинг 15 многообещающая точка входа в упаковщик FSG

Разочарование начинается **вс** первых же команд (см. листинг 15). FSG переназначает регистр ESP и хотя через некоторое время восстанавливает его вновь — особой радости нам это не доставляет. Упаковщик очень интенсивно использует стек, поэтому точка останова на "BPM ESP-4" выдает миллион ложных срабатываний, причем большинство из них относятся к циклам (см. листинг 16):

```

001B:010001C1 5E          POP   ESI
001B:010001C2 AD          LODSD
001B:010001C3 97          XCHG  EAX, EDI
001B:010001C4 AD          LODSD
001B:010001C5 50          PUSH  EAX
001B:010001C6 FF5310     CALL  [EBX+10]

```

Листинг 16 фрагмент кода, генерирующий ложные срабатывания точки останова

Необходимо внести какое-то дополнительное условие (к счастью, soft-ice поддерживает условные точки останова!), автоматически отсеивающее ложные срабатывания или хотя бы их часть. Давайте подумаем! Если стартовый код упакованной программы начинается со стандартного пролога типа PUSH EBP/MOV EBP, ESP, то точка останова "BPM ESP-4 IF *(ESP) == EBP", отсеет кучу мусора, но... ~~при этом, она~~ будет срабатывать на любом стандартном прологе нулевого уровня вложенности, а ~~во-вторых, если~~ упакованная программа ~~может иметь~~ оптимизированный пролог, в котором регистр EBP не используется, ~~наш "хитрый" прием вообще не работает!~~

А вот другая идея: допустим, управление на OEP передается через PUSH offset OEP/RETN, тогда на вершине стека окажется адрес возврата, что опять-таки легко запрограммировать в условной точке останова. Еще управление может передаваться через MOV EAX, offset OEP/JMP EAX, что также легко проконтролировать и отследить, но вот против "прямых" команд JMP offset OEP или JMP [OEP] условные точки останова бессильны. К тому же слишком много вариантов получается. Пока их переберешь... Ложные

срабатывания неизбежны! Попробуйте повоюйте с FSG... В какой-то момент кажется, что решения нет, и наше дело труба, но это не так!

Все известные автору упаковщики (и значительная часть протекторов) не желая перемешивать себя с кодом упаковываемой программы, размещается в отдельной секции (или вне секции), расположенной либо перед упаковываемой программой, либо после нее! **Код упаковщика сосредоточен в одном определенном месте (нескольких местах) и никогда не пересекается с кодом распаковываемой программы!** Вроде бы очевидный факт. Сколько раз мы проходили мимо него, даже не задумываясь, что **он позволяет автоматизировать процесс поиска ОЕР!**

Взглянем на карту памяти еще раз (см. листинг 17):

```
MAP32
NOTEPAD-fsg      0001  001B:01001000  00010000  CODE  RW
NOTEPAD-fsg      0002  001B:01011000  00008000  CODE  RW
```

Листинг 17 две секции упакованной программы

Мы видим две секции, принадлежащие упакованной программе. Сам черт не поймет какая из них секция кода, а какая данных, тем более что должна быть еще одна секция — секция ресурсов, но коварный упаковщик каким-то образом скомбинировал их друг с другом, и мы не знаем каким, впрочем код самого распаковщика, как мы уже видели, сосредоточен в районе адреса 10001xxh, то есть вне этих двух секций (отдельной секции для себя он-распаковщик создавать не стал).

Чтобы отсеять лишние всплывтия отладчика, мы сосредоточимся на диапазоне адресов, принадлежащих упакованной программе, то есть от начала первой секции до конца последней, автоматически контролируя значение регистра EIP на каждом срабатывании точки останова.

В данном случае, условная точка останова будет выглядеть так (см. листинг 18) :

```
BPM ESP-4 IF EIP > 0X1001000 && EIP < 0X1011000
```

Листинг 18 "магическая" последовательность, приводящая нас к ОЕР

Невероятно, но после продолжительного молчания (а он и будет молчать, ведь стек распаковщиком используется очень интенсивно), отладчик неожиданно всплывает непосредственно в ОЕР (см. листинг 19)!

```
001B:01006420 55          PUSH  EBP
001B:01006421 8BEC       MOV   EBP,ESP
001B:01006423 6AFF       PUSH  FF
001B:01006425 6888180001 PUSH  01001888
001B:0100642A 68D0650001 PUSH  010065D0
001B:0100642F 64A100000000 MOV   EAX,FS:[00000000]
001B:01006435 50         PUSH  EAX
001B:01006436 64892500000000 MOV   FS:[00000000],ESP
```

Листинг 19 отсюда начинается распакованный код исходной программы

Фантастика!!! А ведь FSG далеко не самый слабый упаковщик, фактически граничащий с протекторами. Однако, данная методика поиска ОЕР применима и к протекторам. Выделяем секции, принадлежащие упакованной программе, и устанавливаем точку останова на ESP - 4 в их границах. Даже если стартовый код использует оптимизированный пролог, первый же регистр (локальная переменная) заталкиваемая в стек, вызовет срабатывание отладчика. Если мы попадем не в саму ОЕР, то будет где-то очень-очень близко от нее... А найти начало оптимизированного пролога можно и автоматом!

Таким образом, мы получаем в свои руки мощное оружие многоцелевого действия, которого легко реализовать в виде **подключаемого модуля** к LordPE, IDA Pro или самостоятельной утилиты. **(Разработчикам упаковщиков на заметку: чтобы ослепить этот алгоритм, никогда не отделяйте код распаковщика от кода программы! перемешивайте их в хаотичном порядке на манер "мясного рулета", тогда определять ОЕР по данной методике уже не получится, но методики, описанные выше, по прежнему будут работать, если, конечно не предпринять против них определенных усилий).**

>>> врезка что делать если отладчик проскакивает точку входа в распаковщик

Берем исполняемый файл, загружаем его в NuMega SoftICE Symbol Loader, внимательно-предварительно убедившись, что пиктограмма, изображающая "электрическую лампочку", горит в полный накал, горячая лампочка ???, а опция "Start at WinMain, Main,DllMain" активирована (см. рис. 16), что по идее должно отладчик заставить останавливаться в точке входа, но... При попытке загрузки программы soft-ice отладчик коварно игнорирует наши распоряжения, нахально проскакивает точку входа совсем не собираясь в ней останавливаться, полностью утрачивая управление и контроль.

Рисунок 16 символьный загрузчик. хорошая штука, но не всегда работающая

Это известный глюк soft-ice, с которым борются по всем направлениям. Самый популярный (но не самый лучший) способ заключается во внедрении INT 03h (CCh) в точку входа программы.

Берем PE-файл, открываем его в hiew'e, нажимаем <enter> для перехода в hex-режим, теперь <F8> (header) и <F5> (переход к точке входа в файл или сокращенно EP, не путать с OEP, которую нам еще предстоит найти). Запоминаем содержимое байта под курсором (записываем на бумажке), переходим в режим редактирования по <F3> и пишем "CC". Сохраняем изменения по <F9> и выходим. В самом soft-ice должен быть предварительно установлен режим всплытия по INT 03 (команда "I3HERE ON").

Запускаем программу (просто запускаем без всяких там loader'ов) и... попадаем в soft-ice, который непременно должен всплыть, иначе здесь что-то сильно не так. Теперь необходимо вернуть исправленный байт на место. Это делается так: даем команду "WD" для отображения окна дампа (если только оно уже не отображается на экране), затем "DB", чтобы отображение шло по байтам (начинающим — так удобнее) и говорим "D EIP-1". Минус один появился оттуда, что soft-ice останавливается после CCh, увеличивая EIP на единицу. Даем команду "E" и редактируем дампы в интерактивном режиме меняя "CC" на "60" (число, записанное на бумажке). Остается только скорректировать регистр EIP, что осуществляется командой: "R EIP = EIP - 1". Все! Команда "." (точка) обновляет окно дизассемблера перемещая нас в текущую позицию. Теперь можно отлаживать!

Медленно? Неудобно? Куча лишних операций? Операции — это ерунда, их можно "повесить" на макросы (благо, что они однотипные), хуже, что некоторые упаковщики/проекторы контролируют целостность файла, отказываясь запускаться если он изменен. Как быть тогда?

А вот второй способ. Гораздо более элегантный. Загружаем программу в hiew, переходим в hex-режим, жмем <F8> и вычисляем адрес точки входа путем сложения Entry point RVA (в нашем случае — 10001h) с Image Base (в нашем случае 1000000h). Получается 1010001h. Если считать лень, можно просто нажать <F5>, чтобы hiew перенес нас в точку входа, сообщив ее адрес (однако, это не срабатывает на некоторых защищенных файлах с искаженной структурой заголовка). **Хорошо**, адрес EP получен. Вызываем soft-ice путем нажатия на <Ctrl-D> и устанавливаем точку останова на *любую* API-функцию, которую вызывает наша программа (откуда — не суть важно). Это может быть и GetModuleHandleA ("BPH GetModuleHandleA") и CreateFileA — да все что угодно! Выходим из soft-ice и запускаем нашу программу. Отладчик всплывает по точке останова на API. Убедившись, что правый нижний угол отражает имя нашего процесса (если нет, выходим из soft-ice и ждем следующего всплытия), устанавливаем аппаратную точку останова на EP, отдавая команду "BPH 0x1010001 X", где — 0x1010001 адрес точки входа в PE-файл. Выходим из soft-ice и перезапускаем программу. hint: soft-ice запоминает установленные точки вместе с контекстом отлаживаемой программы и не удаляет их даже после ее завершения. При повторном (и всех последующих) перезапусках, soft-ice будет послушно останавливаться на EP в точке останова. Ну разве это не здорово?!

Внимание! *Динамические библиотеки, статически скомпонованные с программой, получают управление до EP!!! Это позволяет защитам предпринять некоторые действия еще до начала отладки! Поэтому, если что-то идет не так, первым делом проверьте код, содержащийся в DllMain всех динамических библиотек.*

заключение

Вот мы и научились находить ОЕР! Остается самая малость — сбросить дампы программы на диск. Но здесь не все так просто, как может показаться вначале и многие упаковщики/проекторы этому всячески сопротивляются. В следующей статье мы покажем как реализовать универсальный дампер, обходящий в том числе и продвинутый механизм динамической шифровки, известный под именем `СоруМемII` — это когда вся программа зашифрована и отдельные страницы памяти расшифровываются непосредственно перед их употреблением, а потом зашифровываются вновь. Так же мы коснемся вопросов восстановления таблицы импорта и поговорим про распаковку DLL.

В конечном счете, получится мощный генетический распаковщик, обходящий всех своих конкурентов.