как подделывают CRC16/32

крис касперски аргентинский болотный бобер nezumi el raton ака нутряк ибн мыщъх, no-email

алгоритм хеширования CRC16/32 считается несокрушимым и никто (исключая, быть может, богов) не может восстановить оригинальное содержание по контрольной сумме, но модифицировать файл так, чтобы его CRC16/32 не изменилось — проще просто. вот об этом мы и будем говорить!

введение

Алгоритм CRC16/32 изначально предназначался для контроля целости данных от ***непреднамеренных искажений*** и широко используется в проводных и беспроводных сетях, магнитных и оптических носителях, а так же микросхемах постоянной или перешиваемой памяти. "Надежность" CRC32 оценивается как 2^32 == 4.294.967.296, то есть вероятность, что контрольная сумма искаженного файла "волшебным" образом совпадает с оригиналом оценивается (в среднем) как один против четырех миллиардов раз. Отсюда следует, что передав восемь миллиардов пакетов через сильно зашумленный канал, мы рискуем получить пару "битых" пакетов, чьи искажения останутся необнаруженными. Но ведь в действительности все совсем не так!!! Природа большинства физических помех и дефектов такова носит групповой характер, с которым CRC32 легко справляется и в реальной жизни никакие искажения от CRC32 не ускользают!!! Отнюдь не глупые ученые и инженеры над этим работали!

Но вот алгоритм CRC32 просочился в массы! Программисты (все мы учились понемногу чему ни будь и как ни будь) стали применять хеширование в зашитых механизмах, призванных обеспечить информационную безопасность и противостоять против предумышленных искажений. Другими словами — защитить от хакерских атак. Машинный код, контролирующий свою целостность через CRC, не очень-то полезен, поскольку контрольная сумма храниться непосредственно в теле программы. Модифицировав программу, хакер рассчитывает новое значение контрольной суммы, корректирует поле CRC (а найти его в отладчике/дизассемблере совсем не проблема) и… защитный механизм продолжает считать, что в "Багдаде все спокойно".

Скажу откровенно: наличие механизмов самоконтроля серьезно раздражает хакеров и препятствует пошаговой трассировке step-by-step, при которой отладчик внедряет CCh после каждой команды. Однако, хакеров лучше не злить. Разъяренный хакер взрывоопасен! Размахивая soft-ice словно топором и прикрываясь дизассемблером как щитом, он находит и поле контрольной суммы, и тот код который ее контролирует, после чего обоим настает конец.

Кстати говоря, "правильная" контрольная сумма должна быть равна нулю. Это закон! Именно так работает механизм самоконтроля BIOS'ов. В этом случае контрольная сумма нигде не хранится, но к содержимому прошивки добавляются четыре (в CRC16 – два) байта, рассчитанных так, чтобы контрольная сумма всего контролируемого блока обращалась в ноль. В этом случае ломать защиту становится намного труднее (ведь поля контрольной суммы нет!), но… все-таки возможно. Достаточно установить аппаратную точку останова на модифицированную команду (в soft-ice это делается так: "bpm adders R") и отладчик приведет нас с коду, который вычисляет контрольную сумму и на каком-то этапе выносит заключение CRC OK или CRC не OK.

Обычно хакеры "отламывают" непосредственно сам проверяющий механизм, чтобы программа работала независимо от того, какой у нее CRC, однако, этот способ срабатывает не везде и не всегда. Вспомним спор, возникший в конференции SU.VIRUS, по поводу инспектора AdInfo: может ли вирус заразить файл так, чтобы его контрольная сумма осталась неизменной? Может! Даже если весь файл проверяется целиком от ушей до хвоста! Чуть позже мы покажем как это сделать, а сейчас рассмотрим другой случай: клиент серверную систему, в которой ключевой файл (условно называемый "сертификатом") находится у клиента, а его контрольная сумма храниться и проверяется на сервере. В состав сертификата может входить все, что угодно: название организации, IP-адрес клиента, его "рейтинг", уровень "полномочий" в системе и т. д. и т. п. Весьма распространенный подход, не правда ли? Чтобы повысить уровень своих полномочий клиент должен модифицировать файл сертификата, что с неизбежностью влечет за собой изменение контрольной суммы, скрупулезно проверяемой сервером, взломать который значительно сложнее, чем подправить пару байт в hiew'e!

Защита подобного типа растут как грибы (вот хорошая статья на эту тему "*Тайна золотого ключика или Особенности системы лицензионных ключей, используемой компанией Software AG*": http://www.wasm.ru/article.php?article=saglickey), разработчики которых свято верят в CRC32, забыв о том, что он страхует только от ненамеренных искажений, то есть искажений, которые происходят случайно и затрагивают один или несколько хаотичным образом разбросанных байт.

Информационная стойкость CRC32 равна 4 байтам и от этого никуда не уйти. Именно столько байт требуется внедрить в модифицированный файл (или дописать их к нему), чтобы его контрольная сумма полностью сохранилась, независимо от того, сколько байт было изменено! Естественно, эти четыре корректирующих байта должны быть рассчитаны по специальному алгоритму, но это потребует совсем немного времени (в смысле — вычислительных ресурсов). Самое интересное, что методики "подделки" CRC широко известны и описаны во множестве руководств, а у популярной утилите PEiD даже имеется плагин, специально предназначенный для этой цели!

>>> врезка как это было

Подделкой CRC мыщъх заинтересовался совершенно случайно. Все началось с экспериментов над Интеловскими биосами, формат прошивок которых был неизвестен и всякая попытка модификации приводила к краху с воплем о неправильном CRC. Вот только найти место хранения CRC никак не удавалось. Границы контролируемого блока были так же неизвестны. Казалось, что дело труба, но…

Используя тот факт, что в прошивке хранится не само CRC, а "корректирующие" байты, обращающие контрольную сумму в ноль, а все структуры выровнены на границу 4х килобайт, мыщъх решил действовать так: ***выбираем блок, считаем его CRC (алгоритм подсчета был выдран из прожигающей программы, который был тут же опознан как стандартный), если не нуль, движемся дальше до тех пор, пока контрольная сумма выбранного блока не обратиться в нуль***.

Достижение нуля сигнализирует о том, что границы контролируемого блока с той или иной степенью вероятности уже найдены. Модифицируем BIOS по своему желанию, после чего добавляем "корректирующие" байты так, чтобы контрольная сумма не изменялась, то есть осталось равной нулю!

Таким образом, чтобы "взломать" CRC, помимо алгоритма расчета еще необходимо знать откуда и до куда его считать! В противном случае попытка подделки контрольной суммы будет немедленно разоблачена!

теоретические основы CRC32

Прежде, чем ломать CRC32, необходимо научиться его считать. Из всех учебников и статей мне больше всего нравится "a painless guide to crc error detection algorithms", которое я настоятельно рекомендую для изучения. Имеется так же и русскоязычный перевод. Адреса обоих можно найти во врезке по ссылкам. Кстати говоря, в последнее время набирает силу CRC64, который рассчитывается аналогичным образом, но, естественно, другим полиномом. Кое-где используется даже CRC128, но пока он не стандартизован, а поэтому здесь не затрагивается.

Аббревиатура CRC расшифровывается как "***C****yclic* ***R****edundancy* ***C****heck*" (проверка избыточности циклической суммы) и стягивает под свою крышу целую кучу самых разных алгоритмов, разработанных в "геральдические" времена с ориентацией на воплощение в железо. В практическом плане это означает, что большинство CRC алгоритмов представляют собой тривиальный подсчет контрольной суммы в стиле а-ля: "for(a=0,CRC=INIT_VALUE;a < len;a++) CRC+=DATA[a];", лишь с той оговоркой, что сложение выполняется не алгебраическим путем, а происходит в полях Галуа в тесном сотрудничестве с полиномиальной арифметикой. В "железе" все это хозяйство реализуется на сдвиговых регистрах и логических вентилях, что очень дешево стоит и еще быстрее работает.

Программная "эмуляция" CRC на IBM PC мягко говоря... неоправданна. Приходится прибегать к табличным алгоритмам или сложным математическим преобразованиям, но первое требует памяти, второе — времени (процессорного) и в конечном счете мы раздуваем муху до размеров слона. Почему же тогда CRC пользуется такой популярностью? Его можно найти в zip, arj, различных защитных механизмах…. Такое впечатление, что кому-то некуда девать свои курсовые работы, а кто-то просто использует готовые библиотеки, пропуская теорию кодирования мимо ушей. Зачем она ему? Ведь все и так "работает"!

Но это все лирика. Вернемся к нашим баранам. Никакого единого стандарта на CRC не было и нет! Даже когда говорят о CRC16 или CRC32, то подразумевают всего лишь разрядность контрольной суммы, для вычисления которой необходимо знать два фундаментальных параметра: ***полином*** (poly) и ***начальное значение контрольной суммы*** (init).

А два других параметры: reflection — указывает на то, используется ли зеркальное битовое отражение или нет. Отражаться могут как входные данные, так и выходные. Если байт отражен, то нумерация в нем ведется слева направо, то есть наименее значимый бит располагается по меньшему адресу (как, например, на x86), и, соответственно, наоборот. Наиболее быстрыми будут те алгоритмы, которые соответствуют выбранной "железной" архитектуре, в противном случае, приходится прибегать к довольно дорогостоящей в плане процессорных ресурсов эмуляции.

Параметр XorOut — это то значение, которым XOR'ится рассчитанная контрольная сумма. Обычно она равна 0 или -1 (то есть FFFFh для CRC16 и FFFFFFFFh для CRC32).

Различные схемы (программы, защитные механизмы, аппаратные устройства) используют различные параметры подсчета контрольной суммы, отсутствие информации о которых существенно затрудняет взлом, поэтому ниже приводятся данные об основных алгоритмах, которые мне только удалось открыть.

название	разрядность, бит	poly	init	отражение		xorout
				init	out	
CRC-16	16	8005h	0000h	да	да	0000h
CRC-16/CITT	16	1021h	FFFFh	нет	нет	0000h
XMODEM	16	8408h	0000h	да	да	0000h
ARC	16	8005h	0000h	да	да	0000h
CRC-32	32	04C11DB7h	FFFFFFFFh	да	да	FFFFFFFFh
CRC-64	64	60034000F050Bh	FAC432B10CD5E44Ah	да	да	FFFFFFFFFFFFFFFFh

Таблица 1 параметры подсчета CRC, используемые в различных алгоритмах

Постойте! Но ведь "стандартный" полиномом для CRC32 равен EDB88320h, а совсем не 4C11DB7h. Это каждый студент знает! Все правильно, никакого противоречия здесь нет, закройте забрало и не высаживайтесь! Число EDB88320h представляет собой зеркально отраженный полином 4C11DB7h, в соответствии со словом "да" в колонке "отражение" (см. таблицу 1). Давайте переведем его к битовую виду и посмотрим, что получится. А получится у нас вот что:

```
04C11DB7h: 00000100110000010001110110110111
EDB88320h: 11101101101110001000001100100000
```

Листинг 1 стандартный CRC-32 полином и его общепринятая отображенная версия

Готовую процедуру расчета CRC приводить не будем, поскольку ее можно найти в любой книжке и в ссылках, приведенных ниже. Как уже говорилась, ее программная эмуляция на IBM PC крайне ненаглядна.

подделка CRC32, не преследуемая по закону

В общих чертах, идея "взлома" CRC заключается в добавлении к контролируемому блоку нескольких байт, подгоняющих конечную контрольную сумму под прежнее значение. Как легко показать, количество корректирующий равно разрядности контрольной суммы, то есть CRC16 требует двух, CRC32 – четырех, а CR-64 восьми байт, рассчитанных специальным образом (или полученных методом перебора, см. врезку "неизвестные алгоритмы и борьба с ними").

Как же их рассчитать? Помимо используемого алгоритма (включая такие интимные подробности как значение полинома, начальное значение и xor-out), необходимо знать откуда и докуда защита считает контрольную сумму. Выяснив это, необходимо решить другой, гораздо более щекотливый вопрос: собираемся ли мы ***внедрять*** корректирующие байты внутрь контролируемого блока или ограничимся их ***дописыванием***. Если защитный механизм проверяет целостность всего блока, то лучше и проще всего будет дописать, как это подробно описано в руководстве "CRC and how to Reverse it" с приложенными к нему исходными текстами. Однако, длина контролируемого блока (в роли которого чаще всего выступает исполняемый файл) в этом случае заметно изменится! К тому же многие защитные механизмы контролируют целостность фиксированного участка и хоть дописывай байты, хоть не

дописывай — защита не обратит на них никакого внимания, и CRC контролируемого блока после его модификации необратимо изменится!

Ниже приведен фрагмент исходного кода плагина "CRC" к утилите PEiD, которая как раз и дописывает 4'е корректирующих байта к модифицированному файлу. Пользуйтесь им на здоровье, только не спрашивайте, где взял.

```
#include<stdio.h>
unsigned long c,c2,p2,pol=0xEDB88320;
long n,k;
main()
{
        printf("CRC32 Adjuster (c) 2001 by RElf @ HHT/2\n");
        printf("Length of data: "); scanf("%ld",&n);
        printf("Offset to patch: "); scanf("%ld",&k);
        n = (n-k)<<3;
        printf("Current CRC32: 0x"); scanf("%x",&c);
        printf("Desired CRC32: 0x"); scanf("%x",&c2);
        c ^= c2;
        p2 = (pol << 1) | 1;
        while(n--) if(c&0x80000000) c = (c<<1)^p2; else c<<=1;
        printf("XOR masks:%02X%02X%02X%02X\n",c&0xff,(c>>8)&0xff,(c>>16)&0xff, c>>24);
}
```

Листинг 2 вот код от RElf'а для нахождения корректирующих байт ;)

Методика внедрения байт внутрь контролируемого блока гораздо более перспективна. Если это машинный код (где нужно исправить один условный переход на другой), нам наверняка удастся переписать несколько машинных команд так, чтобы высвободить место для 4х байт, не говоря уже о том, что компиляторы всюду вставляют NOP'ы для выравнивания. Если же это ключевой файл типа сертификата, что ж… втиснуться туда еще проще (хотя бы оттяпать от своего имени четыре символа, при условии, что там содержится имя).

Рассмотрим практический пример подделки CRC32 с little-endian порядком бит в потоке (наименее значимый бит располагается по меньшему адресу, то есть поток движется справа налево). Остальные алгоритмы взламываются аналогичным образом, с поправкой на специфику разрядности и направления потока.

Возьмем последовательность байт "`A:::::::::::::::::::::::::::::::::D`", контрольная сумма которой известна и равна CRC_OK, модифицируем ее некоторым образом, изменив один или несколько произвольно расположенных байт (они обозначены символом "x"): "`A:::::x:::::::::::::::::::::::::xxx::::x:x:D`", после чего контрольная сумма модифицированной последовательности будет CRC_NOT_OK, что очень нехорошо со всех точек зрения с какой только ни посмотри.

Подделка CRC начинается в выбора позиции "врезки" 4х корректирующих байт. С точки зрения алгоритма CRC их положение непринципиально и главным образом оно определяется внутренней структурой модифицируемого блока. Для наглядности, разместим корректирующие байты в середине "`A:::::x::::::B_12_3_4_C:::::::xxx:::x:x:D`".

Остается только рассчитать корректирующие байты _1_2_3_4_ и восстановить (в смысле "подделать") CRC. Будем действовать по шагам.

1) в точке "A" CRC равно начальному значению полинома (для CRC32 с отраженным полиномом EDB88320h это FFFFFFFFh). В точке "D" оно известно из условия задачи и равно CRC_OK;

2) если участок A-B не пуст — считаем CRC от точки "A" до точки "B", используя обычную процедуру update_CRC32(), исходный код которой приведен в листинге 3;

3) если участок C-D не пуст — считаем CRC от точки "C" до точки "D", используя инверсную процедуру reverse_crc32(), исходный код которой приведен в листинге 4;

4) двигаясь в обратном направлении от точки "C" к точке "B", вычисляем элементы CRC-таблицы, преобразующие CRC(C) в CRC(B), и запоминаем индексы этих элементов в массиве i (см. листинг 7).

5) двигаясь в прямом направлении от точки "B" к точке "C", по индексам, рассчитанным на шаге 4, вычисляем корректирующие байта _1_2_3_4_, записывания их в массив x (см. листинг 8).

Остается только переместить массив x на отрезок B-C, удостовериться, что контрольная сумма блока не изменилась, и бежать в ближайший ларек за свежим пивом (травой или грибами — это уже по вкусу).

Все необходимые для взлома листинги приводятся ниже. Обратите внимание на копирайт. За исключением первого и третьего из них, мыщъх не имеет к ним никакого отношения! Так что за хвост не дергать и ногами не пинать.

```
update_CRC32(unsigned long crc, unsigned char byte)
{
        return crc32nornal[(unsigned char)crc ^ byte] ^ (crc >> 8);
}
```

Листинг 3 функция вычисления "прямого" CRC (см. шаг 2)

```
// (c)Dmitry Tomashpolski
reverse_crc32(unsigned long crc, unsigned char byte)
{
        return ((crc<<8) ^ crc32inv[(unsigned char)(crc>>24)] ^ (byte));
}
```

Листинг 4 функция вычисления обратного CRC (см. шаг 3)

```
gen_crc32()
{
        unsigned int i, j, r;
        for (i = 0; i < 256; i++)
        {
                for(j = 8,r=i; j > 0; j--) if(r&1)r = (r >> 1) ^ CRCPOLY; else r >>= 1;
                crc32normal[i] = r;
        }
}
```

Листинг 5 функция построение прямой CRC-таблицы, на которую опирается процедура update_CRC32 (см. листинг 3)

```
// (c)Dmitry Tomashpolski
gen_crc32inv(void)
{
        int i; for(i = 0; i < 256; i++)
        unsigned long tmp = crc32(0, i),
        crc32dir[i] = tmp,
        crc32inv[(unsigned char)(tmp>>24)] = (tmp<<8) ^ i;
}
```

Листинг 6 функция для расчета инверсной CRC-таблицы, на которую опирается процедура reverse_CRC32 (см. листинг 4)

```
        // (c)Dmitry Tomashpolski
        k = 4;
        y = CRC_D;
        if((i[--k] = sr(crc32dir,  y, 0xFF000000ul)) < 0)  goto err;
        t = crc32dir[i[k]]; y = (y ^ t)<<8 | i[k];
        if((i[--k] = sr(crc32dir,  y, 0xFF000000ul)) < 0)  goto err;
        t = crc32dir[i[k]]; y = (y ^ t)<<8 | i[k];
        if((i[--k] = sr(crc32dir,  y, 0xFF000000ul)) < 0)  goto err;
        t = crc32dir[i[k]]; y = (y ^ t)<<8 | i[k];
        if((i[--k] = sr(crc32dir,  y, 0xFF000000ul)) < 0)  goto err;
        t = crc32dir[i[k]]; y = (y ^ t)<<8 | i[k];
```

Листинг 7 код, занимающийся построением массива индексов (см. шаг 4)

```
        // (c)Dmitry Tomashpolski
        y = CRC_B;
        x[k] = (unsigned char) y ^ i[k];
        z = crc32f(z, x[k]); k++;
        y = z; x[k] = (unsigned char) y ^ i[k];
        z = crc32f(z, x[k]); k++;
        y = z; x[k] = (unsigned char) y ^ i[k];
        z = crc32f(z, x[k]); k++;
        y = z; x[k] = (unsigned char) y ^ i[k];
        z = crc32f(z, x[k]); k++;
```

Листинг 8 код, вычисляющий корректирующие байты (см. шаг 5), и помещающий их в массив x.

```
// (c)Dmitry Tomashpolski
int sr(unsigned long *a, unsigned long v, unsigned long m)
{
    int d;
    for(d = 256; --d >= 0;)
       if(((a[d]^v)&m) == 0)
       break;
    return d;
}
```

Листинг 9 служебная функция, занимающаяся поиском элементов в таблице t, которые по маске m, совпадают со значением v (см. листинг 7);

>>> врезка неизвестные алгоритмы и борьба с ними

Хорошо, если алгоритм подсчета контрольной суммы известен, но что делать если он недоступен? (воплощен в железе, находится на удаленном сервере и т. д.). А вот что — воспользоваться тупым перебором! Информационная стойкость CRC32 равна 32-битам, дающих всего лишь 4.294.967.296 комбинаций. Учитывая высокую скорость вычисления CRC32, подбор корректирующих байт займет совсем немного времени — буквально минуты, а то и десятки секунд!

Внедряем двойное слово, равное нулю, в произвольное место контролируемого блока и рассчитываем контрольную сумму по обычному алгоритму. Если CRC не OK, увеличиваем двойное слово на единицу и продолжаем до тех пор, пока желаемое CRC не будет найдено.

CRC16 подбирается вообще мгновенно! С CRC64, конечно, уже приходится основательно попыхтеть, но быстрые табличные алгоритмы на мощных процессорах найдут искомую комбинацию за несколько часов, ну в худшем случае за ночь (тут все, конечно, от размера контролируемого блока зависит).

CRC128 методом перебора уже не ломается (разве что задействовать сеть из нескольких машин), но элементарно восстанавливается по методике, описанной выше.

переходим от теории к практике

Прежде, чем разрабатывать собственный взломщик CRC32, попробуем повозиться с уже существующим. Скачаем популярный распознаватель упаковщиков PEiD (http://www.wasm.ru/baixado.php?mode=tool&id=67), установим набор дополнительных плагинов (http://www.wasm.ru/baixado.php?mode=tool&id=318), в число которых входит и модуль с FixCRC с названием, которое говорит само за себя (в аннотации на WASM'e сказано, что он предназначен для исправления поля CheckSum в PE-заголовке, но это не так!).

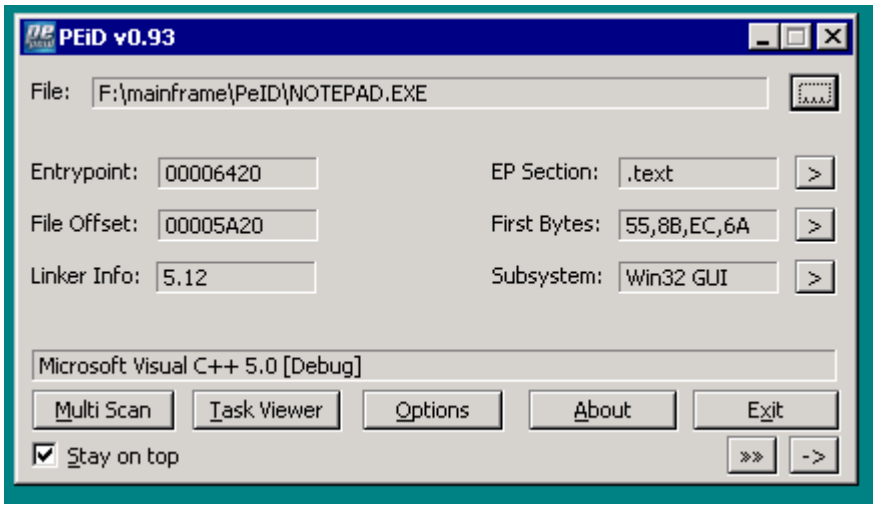

Рисунок 1 внешний вид утилиты PEiD

Запускаем PEiD и загружаем в него какое-нибудь приложение, например, общепринятый notepad.exe ака "блокнот" (см. рис. 1). Давим на кнопку со стрелочкой "–>" в правом нижнем углу, на экране появляется меню "Plugins" из которого мы выбираем пункт

"CRC32" (см. рис. 2). Возникает симпатичное диалоговое окошко, сообщающее нам контрольную сумму всего файла (у мыщъх'а она равна AFBF6001h). Записываем ее на бумажке (или запоминанием в голове, кстати говоря, после курса принятия ноотропов восьмизначные числа запоминаются только так!).

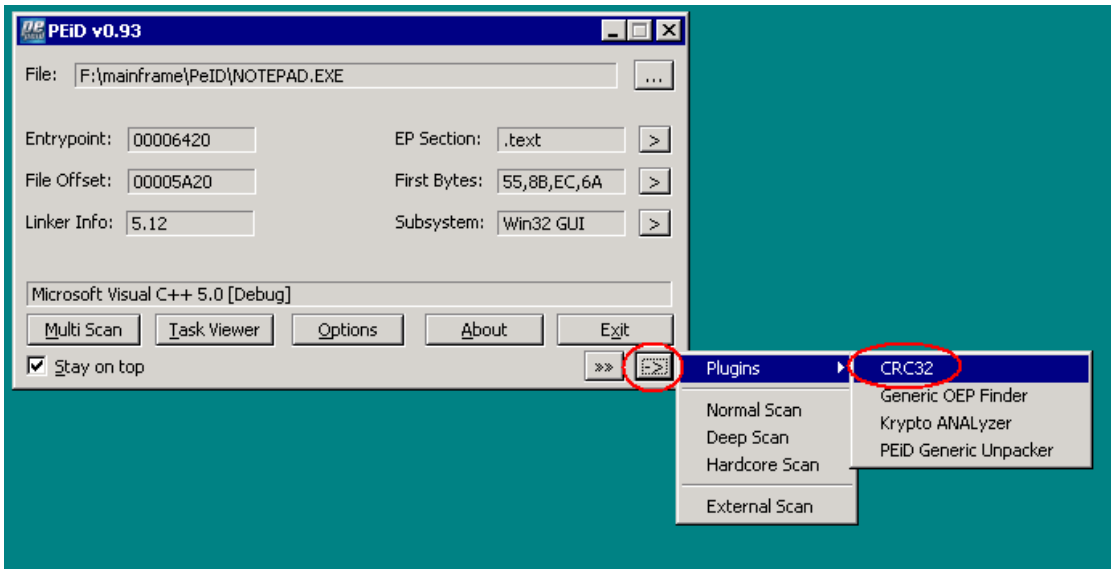

Рисунок 2 вызов плагина CRC32

Выходим из PEiD и правим файл по своему усмотрению в hiew'e или любом другом hex-редакторе, который вам больше по вкусу. Модифицированный файл вновь загружаем в PEiD и рассчитываем новую контрольную сумму, которая теперь равна A97343D5h. AFBF6001h != A97343D5h, что не есть хорошо и чтобы CRC было OK, его надо поправить. Заносим в поле NewCRC старую контрольную сумму оригинального файла (AFBF6001h) и ждем кнопку "Fix It". Плагин сообщает, что "4 bytes written" (см. рис. 3) и действительно дописывает к концу файла какую-то гадость (см. рис. 4)

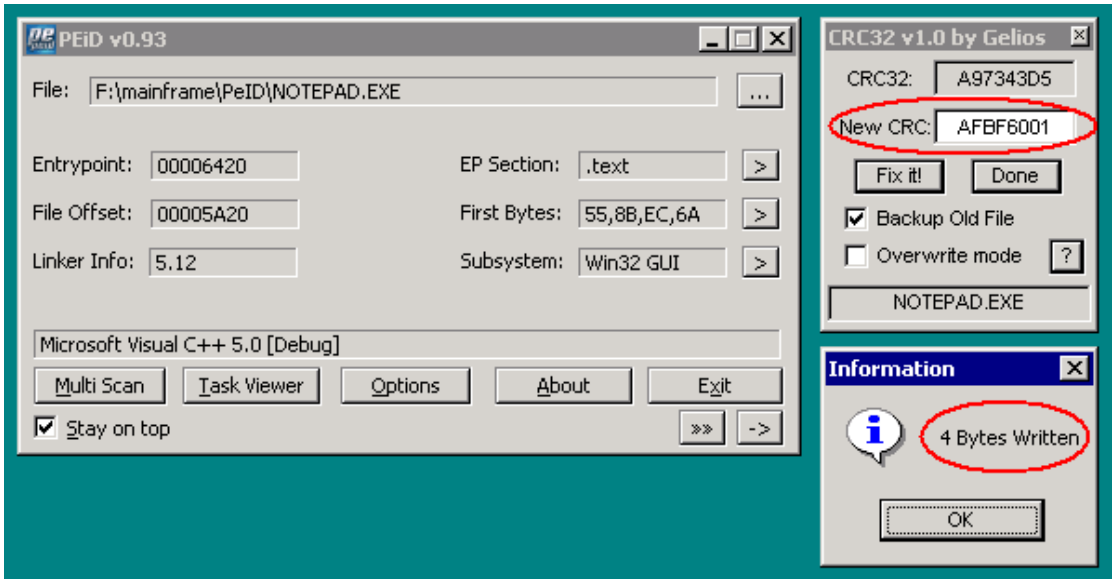

Рисунок 3 для "подделки" контрольной суммы, PEiD дописывает в конец файла 4 байта

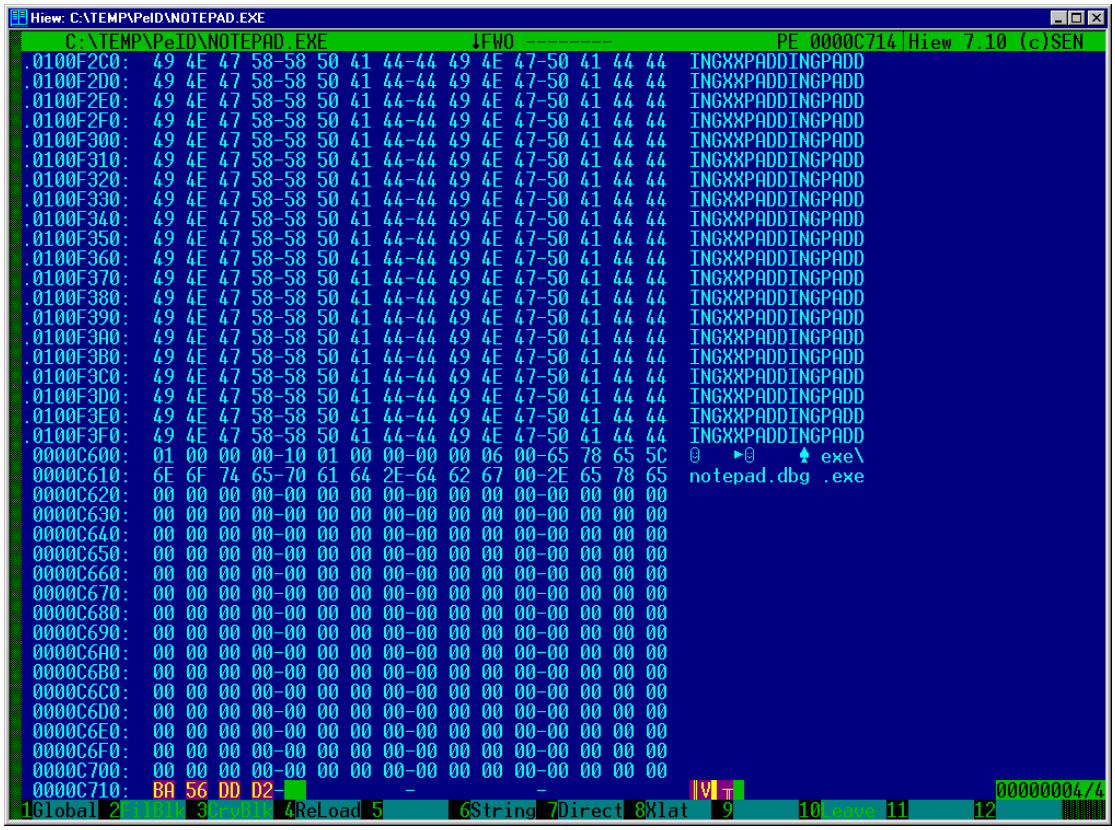

Рисунок 4 последние 4'е байта (выделенные красным маркером) были вставлены PEiD для коррекции CRC

Зато контрольная сумма файла вновь равна AFBF6001h, какой она и была до модификации. Правда, длина файла изменилась… к тому же с защитами, контролирующими контрольную сумму "от сих до сих" такой трюк уже не прокатывает и приходится хитрить.

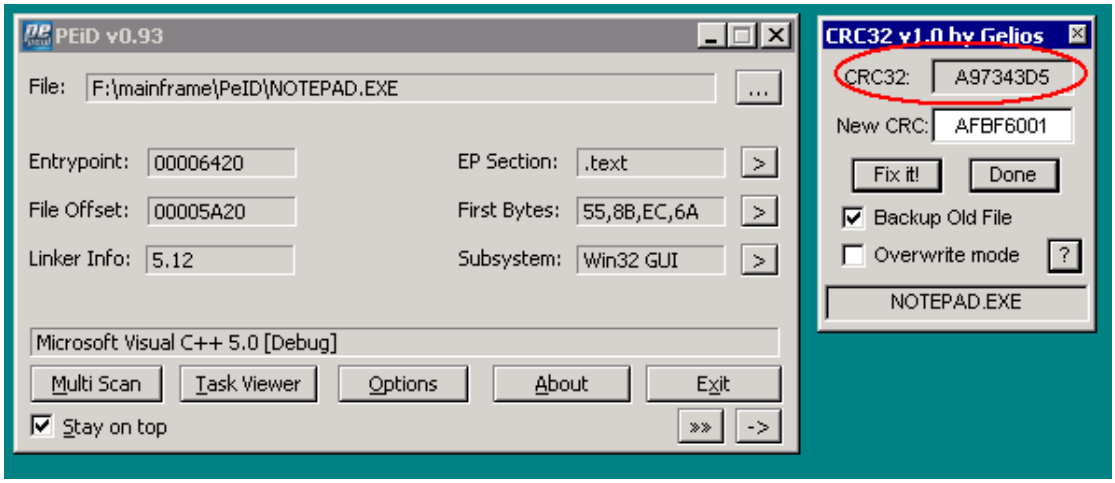

Рисунок 5 PEiD скорректировал CRC модифицированного файла так, чтобы никто не заметил подмены

Берем оригинальный notepad.exe, привычным действием вычисляем контрольную сумму, затем отрезаем от его хвоста 4 байта (*в hiew'е это делается так: загружаем файл, нажимаем <ENTER> для перехода в hex-mode, перемещаемся в конец файла по <Ctrl-End>, отступаем курсором на четыре байта назад, давим <F3> для перехода в режим редактирования и говорим <F10> (truncate), подтверждая всю глубину серьезности своих намерений клавишей "Y"*). Модифицируем файл по своему усмотрению и fix'им его в PEiD. Как легко догадаться, корректирующие байты будут дописаны на место отрезанных. Ни длина, ни контрольная сумма файла теперь не изменится!

Аналогичным образом можно корректировать отдельные блоки, если предварительно вырезать их из программы и сохранить в отдельном файле, который после модификации и исправления CRC вновь вернуть там, где они лежали.

заключение

Учите мат. часть мужики! Незнание теории не освобождает от ответственности, существенно упрощая взлом "несокрушимых" (на бумаге) программ и навороченных криптосистем, буквально разваливающихся под собственной тяжестью. Алгоритм CRC16/32/64/128 страхует только от непреднамеренных искажений, но для защиты от хакеров он непригоден. Используйте MD5 и другие, более продвинутые криптографические алгоритмы (кстати говоря, по производительности MD5 вполне сопоставим с CRC32 и слухи о его "неповоротливости" слишком преувеличены).

Конечно, при желании можно подделать и MD5, однако, для этого потребуется глубокое значение криптографии и нехилые вычислительные мощности, которых, в распоряжении хакера скорее всего не окажется!

>>> врезка интересные ссылки по теме

- **A PAINLESS GUIDE TO CRC ERROR DETECTION ALGORITHMS**:
  - отличное руководство по устройству и реализации различных CRC-алгоритмов, ориентированное на Си-программистов, с готовыми примерами, лучше которого мыщъх ничего не видел (на английском языке): http://www.cs.waikato.ac.nz/~312/crc.txt;
  - перевод вышеупомянутого руководства на великий и могучий русский язык: http://www.onembedding.com/info/crc/crc_rus.zip;
  - домашняя страница автора руководства по CRC с кучей полезной (и не очень) информации и ссылками на родственные сайты (на английском языке): http://www.ross.net/crc/;
- **CRC-64 и как его считать**:
  - любопытная заметка на форуме, дающая базовое представление об алгоритме CRC-64 и методах его вычисления (на английском языке): http://www.pdl.cmu.edu/mailinglists/ips/mail/msg02982.html;
- **Cyclic redundancy check**:
  - CRC на wikipedia! довольно слабая, путанная, местами неверная и откровенно неполная статья, зато с кучей ссылок на другие сайта (на английском языке):
  - http://en.wikipedia.org/wiki/Cyclic_redundancy_check;
- **CRC**:
  - статья по CRC из русской редакции wiki. это _не_ перевод английской, а самостоятельный материал, куцый как мой хвост (на русском языке): http://ru.wikipedia.org/wiki/CRC;
- **CRC and how to Reverse it Anarchriz/DREAD**:
  - хакерская статья, ориентированная на астматиков и доходчиво рассказывающая как вычисляется и подделывается CRC32 путем дописывания 4х корректирующих байт в конец контролируемого блока (на английском языке): http://foff.astalavista.ms/tutorialz/Crc.htm;
  - добротный перевод той же самой статьи на русский язык, оформленный в pdf: http://www.pilorama.r2.ru/library/pdf/crcrevrs.pdf;
- **FAQ по вычислению CRC**:
  - бессистемный FAQ по вычислению и взлому CRC, подробно описывающий способ внедрения корректирующих байт внутрь контролируемого блока, с законченными примерами на языке Си, как нетрудно догадаться метод Anarchriz'a являются частным случаем данного способа (на русском языке): http://faqs.org.ru/progr/common/crc_faq.htm;
- **PEiD**:
  - бесплатно распространяемая утилита, использующаяся для идентификации упаковщиков, компиляторов, протекторов и многих других хакерских целей: http://www.wasm.ru/baixado.php?mode=tool&id=67;
- **plugs**:

- небольшая коллекция плагинов к PEiD, среди которых есть и плагин, ответственный за подделку CRC32, путем дописывания 4х байт в конец файла: http://www.wasm.ru/baixado.php?mode=tool&id=318;