Крис Касперски

НЕЯВНЫЙ САМОКОНТРОЛЬ КАК СРЕДСТВО СОЗДАНИЯ НЕ ЛОМАЕМЫХ ЗАЩИТ

kk@sendmail.ru

Крис Касперски
kk@sendmail.ru

Неявный самоконтроль как средство создания не ломаемых защит

Введение

Основная ошибка подавляющего большинства разработчиков защитных механизмов состоит в том, что они дают явно понять хакеру, что защита еще не взломана. Если защита сообщает "неверный ключевой файл (пароль)", то хакер ищет тот код, который ее выводит и анализирует условия, которые приводят к передаче управления на данную ветку программы. Если защита в случае неудачной аутентификации блокирует некоторые элементы управления и/или пункты меню, – хакер либо снимает такую блокировку в "лоб", либо устанавливает точки останова (в просторечии называемые бряками) на API-функции, посредством которых такое блокирование может быть осуществлено (как правило это EnableWindows), после чего он опять-таки оказывается в непосредственной близости от защитного механизма, который ничего не стоит проанализировать и взломать. Даже если защита не выводит никаких ругательств на экран, а просто тихо "кончает", молчаливо выходя из программы, то хакер либо ставит точку останова на функцию exit, либо тупо трассирует программу и, дождавшись момента передачи управления на exit, анализирует один или несколько последующих условных переходов в цепи управления, – какой-то из них непосредственно связан с защитой!

В некоторых защитных механизмах используется контроль целостности программного кода на предмет выявления его изменений. Теперь, если хакер подправит несколько байтиков в программе, защита немедленно обнаружит это и взбунтуется. Святая простота! – воскликнет хакер, – и отключит самоконтроль защиты, действуя тем же самым способом, что описан выше. По наблюдениям автора, типичный самоконтроль выявляется и нейтрализуется за несколько минут. Наиболее сильный алгоритм защиты: использовать контрольную сумму критических участков защитного механизма для динамической расшифровки некоторых веток программы ломаются уже не за минуты, а за часы (в редчайших случаях – дни). Алгоритм взлома выглядит приблизительно так: а) подсмотрев контрольную сумму в оригинальной программе, хакер переписывает код функции CalculateCRC, заставляя ее всегда возвращать это значение, не выполняя реальной проверки; б) если защита осуществляет подсчет контрольной суммы различных участков

программы и/или разработчик использовал запутанный самомодифицирующийся код, трудно предсказуемым способом меняющий свою контрольную сумму, то хакер может изменить защиту так, чтобы она автоматически само, восстанавливалась после того, как все критические участки будут пройдены; в) отследив все вызовы CalculateCRC, хакер может просто снять динамическую шифровку, расшифровав ее вручную, после чего надобность в CalculateCRC пропадает.

Стоит отметить, что независимо от способа своей реализации любой самоконтроль элементарно обнаруживается установкой точек останова на те участки защитного механизма, которые были изменены. Остальное – дело техники. Можно сколь угодно усложнять алгоритм подсчета контрольной суммы, – напичкивать его антиатладочными приемами, реализовать его на базе собственных виртуальных машин (то есть интерпретаторов), некоторые из которых, такие, например, как Стрелка Пирса, достаточно трудно проанализировать. Но... если такие меры и остановят взломщика, то ненадолго.

Техника неявного контроля

Ошибка традиционного подхода заключается в его предсказуемости. Любая явная проверка чего бы то ни было, независимо от ее алгоритма – это зацепка! Если хакер локализует защитный код, то все – пиши пропало. Единственный надежный способ отвадить его от взлома – "размазать" защитный код по всей программе с таким расчетом, чтобы нейтрализовать защиту без полного анализа всей программы целиком – было заведомо невозможным. К сожалению, существующие методики "размазывания" либо многократно усложняют реализацию программы, либо крайне неэффективны. Некоторые программисты вставляют в программу большое количество вызовов ***одной и той же защитной функции***, идущих из различных мест, наивно полагая тем самым, что хакер будет искать и анализировать их все. Да как бы не так! Хакер ищет ту самую защитную функцию и правит ее. К тому же, зная смещение вызываемой функции, найти, отследить ее вызовы можно без труда! Даже если встраивать защитную функцию непосредственно в место ее вызова, – хакер сможет найти все такие места тупым поиском по сигнатуре. Пускай, оптимизирующие компиляторы, несколько меняют тело inline-функций с учетом контекста конкретного вызова – эти изменения не принципиальны. Реализовать же несколько десятков ***различных защитных функций*** – слишком накладно, да и фантазии у разработчика не хватит и хакер, обнаружив и проанализировав пару – тройку защитных функций, настолько проникнется "духом" и ходом мысли разработчика, что все остальные найдет без труда.

Между тем существует и другая возможность – неявная проверка целостности своего кода. Рассмотрим следующий алгоритм защиты: пусть у нас имеется зашифрованная (а еще лучше упакованная) программа. Мы, предварительно скопировав ее в стековый буфер, расшифровываем (распаковываем) ее и... используем освободившийся буфер под локальные переменные защищенной программы. С точки зрения хакера, анализирующего дизассемблерный код, равно как

и гуляющего по защите отладчиком, все выглядит типично и "законно". Обнаружив защитный механизм (пусть для определенности это будет тривиальная парольная проверка), хакер правит соответствующий условный переход и с удовлетворением убеждается, что защита больше не ругается и программа работает. Как будто бы работает! – через некоторое время выясняется, что после взлома работа программы стала неустойчивой, – то она неожиданно виснет, то делает из чисел "винегрет", то... Почесав репу, хакер озадаченно думает: а как это вообще ломать? На что ставить точки останова? Ведь не анализировать же весь код целиком!

Весь фокус в том, что некоторые из ячеек буфера, ранее занятого зашифрованной (упакованной) программой при передаче их локальным переменным не были проинициализированы! Точнее, они были проинициализированы теми значениями, что находились в соответствующих ячейках оригинальной программы. Как нетрудно догадаться, именно эти ячейки и хранили критичный к изменениям защитный код, а потому и неявно контролируемый нашей программой. Теперь я готов объяснить зачем вся эта котовасия с шифровкой (упаковкой) нам вообще понадобилась: если бы мы просто скопировали часть кода программы в буфер, а затем "наложили" на него наши локальные переменные, то хакер сразу бы заинтересовался происходящим и, бормоча под нос "что-то здесь не так", вышел бы непосредственно на след защиты. Расшифровка нам понадобилась лишь для усыпления бдительности хакера. Вот он видит, что код программы копируется в буфер. Спрашивает себя "а зачем?" и сам же себе отвечает: "для расшифровки!". Затем, дождавшись освобождения буфера с последующим затиранием его содержимого локальными переменными, хакер (даже проницательный!) теряет к этому буферу всякий интерес. Далее – если хакер поставит контрольную точку останова на модифицированный им защитный код, то он вообще не обнаружит к ней обращения, т. к. защита контролирует именно зашифрованный (упакованный) код, содержащийся в нашем буфере. Даже если хакер поставит точку останова на буфер, он быстро выяснит, что: а) ни до, ни в процессе, ни после расшифровки (распаковки) программы содержимое модифицированных им ячеек не контролируется (что подтверждает анализ кода расшифровщика/распаковщика – проверок целостности там действительно нет); б) обращение к точке останова происходит лишь тогда, когда буфер затерт локальными переменными и (по идее!) содержит другие данные.

Правда, ушлый хакер может обратить внимание, что после "затирания" значение этих ячеек осталось неизменным. Совпадение? Проанализировав код, он сможет убедиться, что они вообще не были инициализированы и тогда защита падет! Однако, мы можем усилить свои позиции: достаточно лишь добиться, чтобы контролируемые байты попали в "дырки", образующиеся при выравнивании структуры (этим мы отвечает хакеру на вопрос: а чего это они не инициализированы?), а затем скопировать эту структуру целиком (вместе с контролируемыми "дырками"!) в десяток-другой буферов, живописно разбросанных по всей программе. Следить за всеми окажется не так-то просто: во-первых, не хватит контрольных точек, а, во-вторых, это просто не придет в голову.

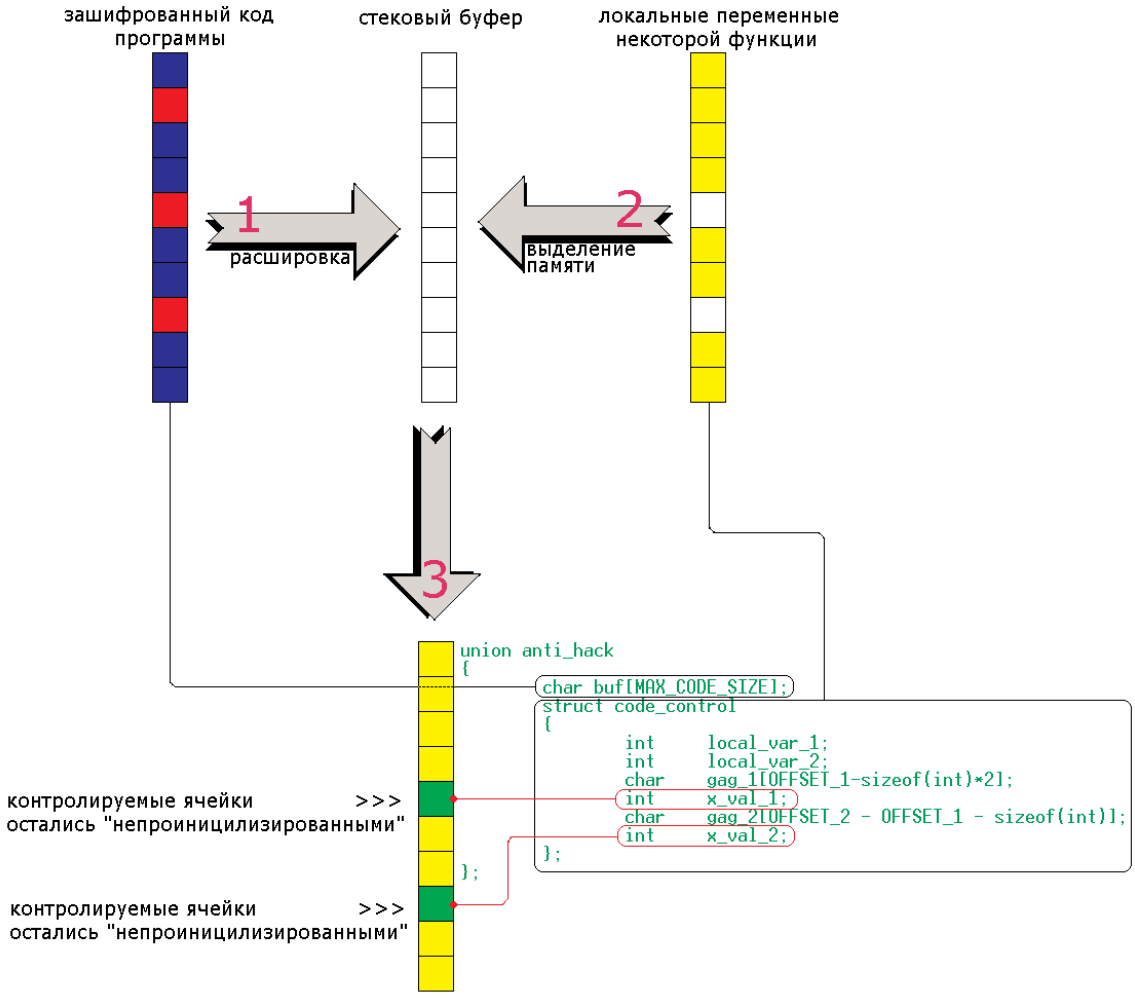

Рисунок 1.

Практическая реализация

Правила хорошего тона обязывают нас проектировать защитные механизмы так, чтобы они никогда, ни при каких обстоятельствах не могли нанести какой бы то ни было вред легальному пользователю. Даже если вам очень-очень хочется наказать хакера, ломающего вашу программу, форматировать диск в случае обнаружения модификации защитного кода, категорически недопустимо! Во-первых, это просто незаконно и попадает под статью о умышленном создании деструктивных программ, а во-вторых... задумайтесь, что произойдет, если искажение файла произойдет в результате действий вируса или некоторого сбоя? Если вы не хотите, чтобы пострадали невинные, вам придется отказаться от всех форм вреда, в том числе и преднамеренном нарушении стабильности работы самой защищенной программы.

Стоп! Ведь выше мы говорили как раз об обратном. Единственный путь сделать защиту трудно ломаемой, – не выдавая никаких ругательных сообщений, по которым нас можно засечь, молчаливо делать "винегрет" из обрабатываемых данных. А теперь, выясняется, что делать этого по этическим (и юридическим!) соображением нельзя. На самом деле, если хорошо подумать, то все эти ограничения легко обойти. Что нам мешает оснастить защиту явной проверкой целостности своего кода? Хакер найдет и нейтрализует ее без труда, но это и не страшно, поскольку истинная защита находится совершенно в другом месте, а вся эта бутафория нужна лишь затем, чтобы предотвратить последствия непредумышленного искажения кода программы и поставить пользователя в известность, что все данные нами гарантии (как явные, так и предполагаемые) ввиду нарушения целостности оригинального кода, аннулируются. Правда, при обсуждении защиты данного типа, некоторые коллеги мне резонно возразили, а что, если в результате случайного сбоя окажутся изменены и контролируемые ячейки, и сама контрольная сумма? Защита сработает у легального пользователя!!! Ну что мне на это ответить? Случайно таких "волшебных" искажений просто не бывает, их вероятность настолько близка к нулю, что... К тому же, в случае срабатывания защиты мы ведь не форматируем легальному пользователю диск, а просто нарушаем нормальную работу программы. Путь и предумышленно, все равно, если в результате того или иного сбоя был искажен исполняемый файл, то о корректности его работы более говорить не приходится. Ну хорошо, если вы так боитесь сбоев, можно встроить в защиту хоть десяток явных проверок, – трудно нам что ли?!

Ладно, оставим этические проблемы на откуп тем самым пользователям, которые приобретают титул "лицензионных" исключительно через крак, и перейдем к чисто конкретным вещам. Простейший пример реализации данной защиты приведен в листинге 1. Для упрощения понимания и абстрагирования от всех технических деталей, здесь используется простейшая схема аутентификации, "ломать" которую совершенно необязательно: достаточно лишь подсмотреть оригинальный пароль, хранящийся в защищенном файле прямым текстом. Для демонстрационного примера такой прием с некоторой натяжкой допустим, но в реальной жизни, вам следует быть более изощренными. По крайней мере следует добиться того, чтобы ваша защита не ломалась изменением одного единственного байта, поскольку в этом случае даже неявный контроль будет легко выявить. Следует так же отметить, что контролировать все критические байты защиты – не очень-то хорошая идея, т. к. хакер сможет это легко обнаружить. Если защита требует для своего снятия хотя бы десяти модификаций в различных местах, три из которых контролируются, то с вероятностью ~70% факт контроля не будет обнаружен. Действительно, среднестатистический хакер следить за всеми модифицированными им байтами просто не будет. Вместо этого он, в надежде что тупая защита контролирует целостность своего кода целиком, будет следить за обращениями к одной, ну максимум двум-трем, измененным им ячейкам и... с удивлением обнаружит, что защита их вообще не контролирует.

После того, как контрольные точки выбраны, вы должны определить их смещение в откомпилированном файле. К сожалению, языки высокого уровня не позволяют определять адреса отдельных машинных инструкций и, если только вы не пишите защиту на ассемблерных вставках, то у вас остается один-единственный путь – воспользоваться каким ни будь дизассемблером (например, IDA).

Допустим, критическая часть защиты выглядит так и нам необходимо проконтролировать целостность условного оператора if, выделенного жирным синим шрифтом:

```
int my_func()
{
  if (check_user())
  {
         fprintf(stderr, "passwd ok\n");
  }
  else
  {
         fprintf(stderr, "wrong passwd\n");
         exit(-1);
  }
  return 0;
}
```

Загрузив откомпилированный файл в дизассемблер, мы получим следующий код (чтобы быстро узнать которая из всех процедур и есть my_func, опирайтесь на тот факт, что большинство компиляторов располагает функции в памяти в порядке их объявления, т. е. my_func будет вторая по счету функция):

```
.text:00401060 sub_401060    proc near                ; CODE XREF: sub_4010A0+AFvp
.text:00401060               call    sub_401000
.text:00401065               test    eax, eax
.text:00401067               jz      short loc_40107E
.text:00401069               push    offset aPasswd0k     ; "passwd ok\n"
.text:0040106E               push    offset unk_407110
.text:00401073               call    _fprintf
.text:00401078               add     esp, 8
.text:0040107B               xor     eax, eax
.text:0040107D               retn
.text:0040107E ; ---------------------------------------------------------------------
.text:0040107E
.text:0040107E loc_40107E:                            ; CODE XREF: sub_401060+7^j
.text:0040107E               push    offset aWrongPasswd ; "wrong passwd\n"
.text:00401083               push    offset unk_407110
.text:00401088               call    _fprintf
.text:0040108D               push    0FFFFFFFFh          ; int
.text:0040108F               call    _exit
.text:0040108F sub_401060    endp
```

Как нетрудно сообразить, условный переход, расположенный по адресу 0x401067 и есть тот самый "if", который нам нужен. Однако, это не весь if, а только малая чего часть. Хакер может и не трогать условного перехода, а заменить инструкцию test eax, eax на любую другую инструкцию, сбрасывающую флаг нуля. Так же он может модифицировать защитную функцию sub_401000, которая и осуществляет проверку пароля. Словом, тут много разных вариантов и на этом несчастном условном переходе свет клином не сошелся, а потому для надежного распознавания взлома нам потребуются дополнительные проверки. Впрочем, это уже детали. Главное, что мы определили смещение контролируемого байта. Кстати, а почему именно байта?! Ведь мы можем контролировать хоть целое двойное слово, расположенное по данному смещению! Особого смысла в этом нет, просто так проще.

Чтобы не работать с непосредственными смещениями (это неудобно и вообще некрасиво), давайте загоним их в специально на то предназначенную структуру следующего вида:

```
union anti_hack
{
  // буфер, содержащий оригинальный код программы
  char buf[MAX_CODE_SIZE];

  // локальные переменные программы
  struct local_var
  {
         int     local_var_1;
         int     local_var_2;
  };

  // неявно контролируемые переменные программы
  struct code_control
  {
         char    gag_1[OFFSET_1];
         int     x_val_1;
         char    gag_2[OFFSET_2 - OFFSET_1 - sizeof(int)];
         int     x_val_2;
  };
};
```

Массив buf – это тот самый буфер, в который загружается оригинальный код программы для его последующей расшифровки (распаковки). Поверх массива накладываются две структуры: local_val, содержащая в себе локальные переменные, которые в процессе своей инициализации затирают соответствующие им ячейки buf'а и тем самым создают впечатление, что прежнее содержимое буфера стало теперь ненужным и более уже не используется. Количество локальных переменных может быть любым, главное – следить за тем, чтобы они не перекрывали контрольные точки программы, изменять содержимое которых нельзя. В приведенном выше примере, по соображениям наглядности, контрольные точки вынесены в отдельную структуру code_control, два массива которой gag_1 и gag_2 используются лишь для того, чтобы переменные x_val_1 и x_val_2 были размещены компилятором по необходимым нам адресам. Как нетрудно долгаться: константа OFFSET_1 задает смещение первой контрольной точки, а OFFSET_2 – второй. Достоинство такой схемы заключается в том, что при добавлении или удалении локальных переменных в структуру local_var, структура code_contorl остается неизменной. Напротив, если объединить локальные переменные и контрольные точки одной общей крышей, то размеры массивов gag_1 и gag_2 станут зависеть от количества и размера используемых нами локальных переменных:

```
union anti_hack
{
  char buf[MAX_CODE_SIZE];
  struct code_control
  {
         int     local_var_1;
         int     local_var_2;
```

```
            char    gag_1[OFFSET_1-sizeof(int)*2];
            int     x_val_1;
            char    gag_2[OFFSET_2 - OFFSET_1 - sizeof(int)];
            int     x_val_2;
    };
};
```

Код, выделенный жирным шрифтом, как раз и отвечает за то, чтобы размер массива-пустышки gag_1, компенсировал пространство, занятое локальными переменными. Такая ручная "синхронизация" крайне ненадежна и служит источником потенциальных ошибок. С другой стороны, теперь мы можем не беспокоиться, что локальные переменные случайно затрут контрольные точки, т. к. если такое произойдет, длина массива gag_1 станет отрицательной и компилятор тут же выскажет нам все, что он о нас думает. Поэтому, окончательный выбор используемой конструкции остается за вами.

Теперь – пару слов о расшифровке (распаковке) нашей программы. Во-первых, нет нужды расшифровывать всю программу целиком, – достаточно расшифровать лишь сам защитный механизм, а то и его критическую часть. Причем, сама процедура расшифровки должна быть написано максимально просто и незамысловато. Поверьте, лишние уровни защиты здесь совершенно ни к чему. Хакер все равно их вскроет за очень короткое время, и, самое главное, чем круче окажется защита, тем внимательнее будет вести себя хакер. Мы же, напротив, должны убедить его, что шифровка это – так, защита от детишек, и "настоящая" защита спрятана где-то совсем в другом месте (пусть ищет то, чего нет!).

Правда, тут есть одна проблема. По умолчанию Windows запрещает модификацию кодовой секции PE-файла и потому непосредственная расшифровка кода невозможна! Первая же попытка записи ячейки, принадлежащей секции .text, вызовет аварийное завершение программы. Можно, конечно, обхитрить Windows, создав свою собственную секцию, разрешающую операции чтения, исполнения и записи одновременно, или, как еще более изощренный вариант, исполнять расшифрованный код непосредственно в стеке, однако, мы пойдем другим путем и просто отключим защиту кодового сегмента от его непредумышленной модификации. Достоинство этого приема в том, что он очень просто реализуется, а недостаток – ослабление контроля за поведением программы. Если в результате тех или иных ошибок, наша программа пойдет в разнос и начнет затирать свой собственный код, операционная система будет бессильна ее остановить, поскольку мы сами отключили защиту! С другой стороны, в тщательно протестированной программе вероятность возникновения подобной ситуации исчезающе мала и в ряде случаев ею можно смело пренебречь. Во всяком случае, в примере, приведенном ниже, мы поступим именно так (речь ведь все равно идет не о технике расшифровке, а о неявном контроле за модификацией кода).

Остается лишь обмолвится парой слов о способах определения диапазона адресов, принадлежащих защитному коду. Поскольку, большинство компиляторов размешают функции в памяти в порядке их объявления в программе, адрес начала защитного кода совпадает с адресом первой, относящейся к нему функции, а адрес конца равен адресу первой не принадлежащей к нему функции (т. е. первой функции, расположенной на его "хвостом").

Теперь, разобравшись с расшифровкой, переходим к самому интересному – неявному контролю за критическими точками нашего защитного механизма. Пусть у нас имеется контрольная точка x_val_1, содержащая значение x_original_1, тогда для его неявной проверки можно "обвязать" некоторые вычислительные выражения следующим кодом: some_var = some_var + (x_val_1 – x_original_1). Если контрольная ячейка x_val_1 действительно содержит свое эталонное значение x_original_1, то разность двух этих чисел равна нулю, а добавление нуля к чему бы то ни было, никак не изменяет его значения. Грубо говоря, x_val_1 уравновешивается противоположным ему по знаку x_origial_1 и за это данный алгоритм называют "алгоритмом коромысла" или "алгоритмом весов". Можно ли быстро обнаружить такие "весы" беглым просмотром листинга программы? Не спешите отвечать "нет", поскольку правильный ответ – "да". Давайте рассуждать не как разработчики защитного механизма, а как хакеры: вот в процессе взлома мы изменили такие-то и такие-то ячейки программы, после чего она отказала в работе. Существует два "тупых" способа контроля своей целостности: контроль по адресам и контроль по содержимому. Для выявления первого хакер просто ищет адрес "хакнутой" им ячейки в коде программы. Если его нет (а в данном случае его и нет!), он предпринимает попытку обнаружить ее содержимое! А вот содержимое контролируемой ячейки в точности равно x_original_1 и тривиальный контекстный поиск за доли секунды выявит все вхождения! Чтобы этого не произошло и наша защита так просто не сдалась, следует либо уменьшить протяженность контролируемых точек до байта (байт – слишком короткая сигнатура для контекстного поиска), либо не хранить x_original_1 в прямом виде, а получать его на основе некоторых математических вычислений. Только не забываете, что оптимизирующие компиляторы все константные вычисления выполняют еще на стадии компиляции и: (#define x_original_1 0xBBBBBA; some_var += (x_val_1 – 1 – x_original_1);) на самом деле не усилит защиту! Поэтому, лучше вообще отказаться от алгоритма "весов", тем более, что он элементарно "вырезается" в случае его обнаружения. Надежнее изначально спроектировать алгоритм программы так, чтобы она осмысленно использовала x_original, а не уравновешивала его "противовесом". Приведенный ниже пример умышленно ослаблен в целях демонстрации как можно использовать эту уязвимость для облегчения взлома.

Исходный текст

```
#include <stdio.h>

#define PASSWD                "+++"
#define MAX_LEN               1023
#define MAX_CODE_SIZE  (0x10*1024)
#define OFFSET_1              0x42
#define OFFSET_2              0x67

#define x_original_1   0xc01b0574
#define x_original_2   0x44681574
```

```
#define x_original_all 0x13D4C04B

#define x_crypt                0x66

int check_user()
{
   char passwd[MAX_LEN];

   fprintf(stderr,"enter password:");
   fgets(passwd, MAX_LEN,  stdin);
   return ~strcmp(passwd, PASSWD);
}


int my_func()
{
   if (check_user())
   {
       fprintf(stderr, "passwd ok\n");
   }
   else
   {
       fprintf(stderr, "wrong passwd\n");
       exit(-1);
   }
   return 0;
}

main()
{
   int a, b = 0;
   #pragma pack(1)

   union anti_hack
   {
       char buf[MAX_CODE_SIZE];
       struct code_control
       {
              int     local_var_1;
              int     local_var_2;
              char    gag_1[OFFSET_1-sizeof(int)*2];
              int     x_val_1;
              char    gag_2[OFFSET_2 - OFFSET_1 - sizeof(int)];
              int     x_val_2;
       };
   };
   union anti_hack ZZZ;

   // TITLE
   fprintf(stderr, "crackeme.0xh by Kris Kaspersky\n");

   // расшифровка кода
   // ===========================================================================

   // копируем расшифровываемый код в буфер
   memcpy(&ZZZ, &check_user, (int) &main - (int) &check_user);
```

```
    // расшифровываем в буфере
    for (a = 0; a < (int) &main - (int) &check_user;  a++)
    {
        (*(char *) ((int) &ZZZ + a)) ^= x_crypt;
    }

    // копируем обратно
    memcpy(&check_user, &ZZZ, (int) &main - (int) &check_user);


    // явная проверка изменения кода
    // ============================================================================
    for (a = 0; a < (int) &main - (int) &check_user;  a++)
    {
        b += *(int *) ((int) &check_user + a);
    }
    if (b != x_original_all)
    {
        fprintf(stderr, "-ERR: invalid CRC (%x) hello, hacker\n", b);
        return 0;
    }

    // явная проверка "валидности" пользователя
    // ============================================================================
    my_func();

    // нормальное выполнение программы
    // ============================================================================

    // скрытый контроль
    ZZZ.local_var_1 = 2;
    ZZZ.local_var_2 = 2;x_original_2;
    sprintf(ZZZ.gag_1, "%d * %d = %d\n", ZZZ.local_var_1,
       ZZZ.local_var_2,
       ZZZ.local_var_1*ZZZ.local_var_2+((x_original_1^ZZZ.x_val_1)+(x_original_2^ZZZ.x_val_2)));

    printf("DEBUG: %x %x\n", ZZZ.x_val_1, ZZZ.x_val_2);
    fprintf(stderr, "%s",ZZZ.gag_1);

}
```

Как это ломают?

Если все сделано правильно, то полученный исполняемый файл не рушиться при его запуске, а победоносно выводит на экран: "crackeme.0xh by Kris Kaspersky\nenter password:" и ждет ввода пароля. Договоримся не обращать внимание на пароль, прямым текстом хранящийся в программе и попробуем взломать защиту другим, более универсальным путем, а именно: изучением алгоритма ее работы под дизассемблером. Запускаем нашу любимую ИДУ и, дождавшись

окончания процесса дизассемблирования, смотрим что у нас там. Ага, текстовые строки "passwd ok" и "wrong passwd" в сегменте данных действительно есть, но вот перекрестных ссылок, ведущих к коду, выводящему их, что-то не видно. Странно, ну да лиха беда начало! Запускам любой отладчик (например, WDB) и устанавливаем на адрес строки "wrong passwd" точку останова: "BA r4 407054". Даем команду "GO" для продолжения выполнения программы, вводим любой, пришедший нам на ум пароль, и... отладчик тут же всплывает, показывая адрес машинной команды, обращающейся к первому символу строки. Но что нам это дает? Ведь мы, судя по всему, находится в теле библиотечной функции out, осуществляющей вывод на консоль и в ее коде для нас нет ничего интересного. С другой стороны – эту функцию кто-то вызывает! Кто именно? Ну мало ли! Функция printf к примеру, код которой для нас ничуть не более интересен... Конечно, поднимаясь по цепочке вызовов вверх (окно call stack вам в помощь!), мы рано или поздно достигнем защитного кода, вызвавшего эту функцию, но вот как нам быстро определить где защитный код, а где библиотечные функции? Да очень просто! Та функция, один из аргументов которой представляет собой непосредственное смещение нашей строки, очевидно, и есть функция защитного кода! Последовательно щелкая мышкой по адресам возврата, перечисленных в окне "call stack", мы наконец находим:

```
0040106E 6854704000        push        407054h
00401073 6810714000        push        407110h
00401078 E88A010000        call        00401207
0040107D 6AFF              push        0FFh
```

Смещение, выделенное жирным шрифтом, – есть ни что иное, как смещение искомой строки, соответственно, адрес 0x40106E (так же выделенный жирным шрифтом) лежит где-то в гуще защитного кода. А ну-ка, глянем сюда дизассемблером – чего это вдруг ИДА не создала перекрестную ссылку к строке?

```
.text:00401000 dword_401000   dd 62668AE7h, 31306666h, 2616560Eh, 17760E66h, 968E6626h
.text:00401000                                          ; DATA XREF: sub_401090+23vo
.text:00401000                                          ; sub_401090+28vo ...
.text:00401000                dd 0E666667h, 662616B6h, 724222EBh, 6665990Eh, 0E38E3666h
.text:00401000                dd 0E5666667h, 26D972A2h, 0EB662616h, 0DF6E4212h, 66666663h
.text:00401000                dd 0C095B455h, 6939A4EDh, 0E738A6F2h, 666266A2h, 0F6F6A566h
.text:00401050 dword_401050   dd 9999CD8Eh, 12A6E399h, 162E0E73h, 760E6626h, 8E662617h
.text:00401050                                          ; CODE XREF: sub_401090+AFvp
.text:00401050                dd 666667F9h, 556EA2E5h, 320EA5A6h, 0E662616h, 66261776h
.text:00401050                dd 6667EC8Eh, 8E990C66h, 666664FDh, 556AA2E5h, 0F6F6A5A6h
.text:00401050                dd 0F6F6F6F6h
```

Вот это номер! Она вообще не посчитала это за код, объявив его массивом! Хорошо, заставим ее дизассемблировать этот фрагмент вручную. Переместив курсор к самому началу массива, нажимаем <U> для его удаления, а затем <C> для превращения байтовой цепочки в код.

```
text:00401000                                           ; sub_401090+28vo ...
text:00401000                out     8Ah, eax           ; DMA page register 74LS612:
text:00401000                                           ; Channel 7 (address bits 17-23)
text:00401002                bound   sp, [esi+66h]
text:00401006                xor     [ecx], dh
text:00401008
text:00401008 loc_401008:                               ; CODE XREF: .text:0040102Dvj
```

```
text:00401008                   push    cs
text:00401009                   push    esi
text:0040100A                   push    ss
text:0040100B                   db      26h, 66h
text:0040100B                   push    cs
text:0040100E                   jbe     short loc_401027
text:00401010                   db      66h
text:00401010                   mov     ss, es:[esi+0E666667h]
text:00401018                   mov     dh, 16h
text:0040101A                   db      26h, 66h
text:0040101A                   jmp     short small near ptr unk_401040
```

Хм! Что за ерунда у нас получается?! Вновь переключившись на отладчик, мы убеждаемся, что тот же самый код в нем выглядит вполне нормально:

```
00401000 81EC00040000           sub     esp,400h
00401006 56                     push    esi
00401007 57                     push    edi
00401008 6830704000             push    407030h
0040100D 6810714000             push    407110h
00401012 E8F0010000             call    00401207
```

Такое впечатление, что защитный механизм зашифрован... А почему бы в самом деле и нет? Возвращаясь к дизассемблеру, щелкаем по перекрестной ссылке и видим:

```
.text:004010AE                  mov     eax, offset sub_401090
.text:004010AE ; загружаем в регистр EAX непосредственное смещение процедуры sub_401090,
.text:004010AE ; чем и выдаем наш бесхитростный расшифровщик с головой если бы целевой
.text:004010AE ; адрес вычислялся бы на основе некоторых математических операций, то вы-
.text:004010AE ; явить расшифровщик было бы сложнее (но по аппаратным контрольным точкам -
.text:004010AE ; все равно возможно)
.text:004010B3                  mov     esi, offset loc_401000
.text:004010B3 ; загружаем в регистр esi непосредственное смещение процедуры loc_401000
.text:004010B3 ;
.text:004010B8                  sub     eax, offset loc_401000
.text:004010B8 ; вычисляем длину зашифрованного фрагмента
.text:004010B8 ;
.text:004010BD                  lea     edi, [esp+14h]
.text:004010BD ; устанавливаем EDI на локальный буфер esp+14h
.text:004010BD ;
.text:004010C1                  mov     ecx, eax
.text:004010C3                  add     esp, 8
.text:004010C6                  mov     edx, ecx
.text:004010C8                  shr     ecx, 2
.text:004010CB                  repe movsd
.text:004010CD                  mov     ecx, edx
.text:004010CF                  and     ecx, 3
.text:004010D2                  repe movsb
.text:004010D2 ; копируем фрагмент [0x40100 - 0x401090) в локальный буфер
.text:004010D2 ;
.text:004010D4                  xor     ecx, ecx
.text:004010D6                  test    eax, eax
.text:004010D8                  jle     short loc_4010EA
.text:004010DA ; есть что расшифровывать? ;-)
.text:004010DA ;
.text:004010DA loc_4010DA:                              ; CODE XREF: sub_401090+58vj
```

```
.text:004010DA do{
.text:004010DA                mov     dl, [esp+ecx+0Ch]
.text:004010DE                xor     dl, 66h
.text:004010E1                mov     [esp+ecx+0Ch], dl
.text:004010E1 ; производим над каждым байтом зашифрованного кода операцию XOR 0x66
.text:004010E1 ;
.text:004010E5                inc     ecx
.text:004010E5 ; берем следующий байт
.text:004010E5 ;
.text:004010E6                cmp     ecx, eax
.text:004010E8                jl      short loc_4010DA
.text:004010E8 ; } while (ecx < eax)
.text:004010EA
.text:004010EA loc_4010EA:                            ; CODE XREF: sub_401090+48^j
.text:004010EA                mov     ecx, eax
.text:004010EC                lea     esi, [esp+0Ch]
.text:004010F0                mov     edx, ecx
.text:004010F2                mov     edi, offset loc_401000
.text:004010F7                shr     ecx, 2
.text:004010FA                repe movsd
.text:004010FC                mov     ecx, edx
.text:004010FE                and     ecx, 3
.text:00401101                repe movsb
.text:00401101 ; записываем расшифрованные данные обратно постой, как записываем обратно?!
.text:00401101 ; ведь модификация секции .text обычно запрещена?! но ведь "обычно" еще
.text:00401101 ; не "всегда", верно?  смотрим атрибуты секции: Flags E0000020: Text
.text:00401101 ; Executable Readable Writable. ага! защита от записи была вручную от-
.text:00401101 ; ключена разработчиком! поэтому перезапись расшифрованного фрагмента
.text:00401101 ; происходит без ошибок и препирательств со стороны Windows
```

Теперь, когда алгоритм расшифровки установлен (см. выделенную жирным шрифтом строку), мы можем самостоятельно расшифровать его. Для этого нажимаем <F2> в окне IDA и вводим следующий скрипт:

```
auto a;
for (a=0x401000; a < 0x401090; a++)
{
    PatchByte(a, Byte(a) ^ 0x66);
}
```

Нажав <Ctrl-Enter> для его выполнения, мы становится свидетелями успешной расшифровки кода защитного механизма. Теперь с ним можно беспрепятственно работать без всяких преград. Кстати, посмотрим, создала ли IDA перекрестные ссылки к строкам "passwd ok" и "wrong passwd"...

```
.text:00401050 sub_401050     proc near               ; CODE XREF: sub_401090+AFvp
.text:00401050                call    sub_401000
.text:00401055                test    eax, eax
.text:00401057                jz      short loc_40106E
.text:00401059                push    offset aPasswdOk  ; "passwd ok\n"
.text:0040105E                push    offset unk_407110
.text:00401063                call    _fprintf
.text:00401068                add     esp, 8
.text:0040106B                xor     eax, eax
.text:0040106D                retn
.text:0040106E ; ---------------------------------------------------------------------
```

```
.text:0040106E loc_40106E:                             ; CODE XREF: sub_401050+7^j
.text:0040106E                 push    offset aWrongPasswd ; "wrong passwd\n"
.text:00401073                 push    offset unk_407110
.text:00401078                 call    _fprintf
.text:0040107D                 push    0FFFFFFFFh          ; int
.text:0040107F                 call    _exit
.text:0040107F sub_401050      endp
.text:0040107F
```

Держи нас за хвост! Перекрестные ссылки действительно созданы и ведут к приведенному выше коду, который слишком прост, чтобы его комментировать. Смотрите: подпрограмма loc_40106E, выводящая надпись "wrong passwd" на экран и прерывающая выполнение программы вызовом функции _exit, имеет перекрестную ссылку, sub_401050+7, ведущую к условному переходу jz short loc_401064 (в листинге он выделен жирным шрифтом), который, судя по всему, и есть тот самый условный переход, что нам нужен! Забив его машинными командами NOP, мы, очевидно, добьемся того, что защита перестанет "ругаться" на неверные пароли и любой введенный пароль воспримет как правильный.

Ну что, запустим HIEW и запишем по адресу .401057 последовательность "90h 90h"? Не спешите, не все так просто! Ведь исходная программа зашифрована и записанные нами команды NOP после расшифровки превратятся неизвестно во что. Какой из этого выход? Да очень простой: записав последовательность 90h 90h в HIEW'е мы тем же самым HIEW'ом ее и зашифруем! OK, приступаем. Итак, <Enter> для перевода HIEW'а в hex-режим, <F5> и ".401057" для перехода по требуемому адресу, <F3> для входа в режим редактирования, 90, 90 – забивает условный переход, <Left Arrow> (четыре раза) для перемещения курсора на начало редактируемого фрагмента, <F8>, <"66"> и еще раз <F8> для шифровки. Наконец, <F9> для сохранения внесенных изменений.

Победно запускаем взломанный файл и...

```
crackeme.0xh by Kris Kaspersky
-ERR: invalid CRC (d7988417) hello, hacker
```

...и тут выясняется, что защита отнюдь не так непроходима тупа, как нам это показалось вначале! Судя по надписи она как-то контролирует целостность своего кода и прекращает работу в случае его изменения. Что ж! Открываем очередное пиво и продолжаем взлом. Можно поступить двояко: или поискать перекрестную ссылку на строку "-ERR: invalid CRC" или же установить контрольную точку на модифицированный нами условный переход. Кинем монетку, если выпадет орел – ищем перекрестную ссылку, ну а если монета упадет решкой – используем контрольную точку. Так, а где у нас монетка? Нету монетки?! Ну тогда, как истинные хакеры, мы быстренько пишем собственный генератор случайных чисел и... решка! (Если у вас выпал орел, значит, нам с вами не по пути).

```
> BA r4 0x407054
> G
Hard coded breakpoint hit
```

Отладчик WDB сообщает, что сработала контрольная точка останова. Пропускаем ее, – это защита копирует код программы в локальный буфер для его последующей расшифровки (это следует из того, что мы всплыли на инструкции

movs). Следующее всплытие отладчика соответствует обратной операции – копированию уже расшифрованного кода на место постоянного проживания. А вот третье по счету всплытие уже интересно:

```
0401109  BA00104000          mov     edx,401000h
0040110E 8B3C0A              mov     edi,dword ptr [edx+ecx]
00401111 03DF                add     ebx,edi
00401113 41                  inc     ecx
00401114 3BC8                cmp     ecx,eax
00401116 7CF1                jl      00401109
00401118 81FB80EC0040        cmp     ebx,4000EC80h
0040111E 741F                je      0040113F
```

Тривиальный алгоритм подсчета контрольной суммы буквально сам бросается в глаза. "Или автор защиты полный идиот, или же он специально хотел быть обнаруженным" – ворчим мы себе под нос, попутно размышляя, что лучше: скорректировать контрольную сумму или же просто заменить условный переход в строке 0x40111E на безусловный, так чтобы он вообще не контролировал свою целостность? Ладно, будем приучать себя к аккуратности. Подгоняем курсор к строке 0x401118 и даем команду "Run to cursor", не забыв предварительно заблокировать установленную точку останова (иначе отладчик просто зациклиться) и смотрим: какое значение содержит в себе регистр EBX. Как следует из окна "Registers" оно равно 0xd7988417, в то время как оригинальная контрольная сумма защищенного файла была – 0x4000EC80 (см. строку 0x401118 приведенного выше листинга). Запускаем HIEW и переписываем ее по живому, меняя "cmp ebx, 4000EC80h" на "cmp ebx, 0xd7988417". Проверяем! wow! Это работает! Выломанный файл успешно запускается и, молчаливо проглотив любой введенный пароль, смиренно сообщает "passwd ok" и продолжает нормальное выполнение программы. Обмыв это дело на радостях двойным количеством пива, хакер раздает выломанную программу всем, нуждающимся в ней пользователям и...

...в процессе эксплуатации взломанной программы выясняется, что ведет она себя мягко выражаясь не совсем адекватно. В частности, в нашем случае она выводит на экран: "2 * 2 = 34280". Вот это номер! Поскольку доверять такому взлому со всей очевидностью нельзя, лучше всего не испытывать судьбу, а приобрести легальную копию программы (особенно, если дело касается бухгалтерского ПО, ошибки которого зачастую несопоставимы с его стоимостью). Но все-таки, хотя бы в плане спортивного интереса, – можно ли взломать такую программу или нет? Условимся, что мы не будем анализировать код, вычисляющий дважды два, поскольку в реальном, полновесном приложении очень легко добиться, чтобы ошибка проявлялась не в месте ее возникновения, а в совсем другой ветке программы, делая тем самым обратную трассировку невозможной.

Первое, что попытается сделать любой здравомыслящий хакер – поискать смещение и/или содержимое модифицированных им ячеек, надеясь что они хранятся в программе прямым текстом. Причем, следует помнить о том, что некоторые защиты контролируют не сам модифицированный байт, а некоторую протяжную область к которой он принадлежит. В частности, если контролируется целостность первого байта условного перехода, то разработчик защиты может схитрить, обратившись к двойному слову, расположенному на три байта "выше". Что ж!

Сказано – сделано. Ищем... Быстро выясняется, что ничего похожего на смещение модифицированного нами перехода в защищенной программе нет, но вот его оригинальное содержимое на наше удивление все-таки обнаруживается:

```
.text:00401090 arg_3F          = dword  ptr  43h
.text:00401090 arg_53          = dword  ptr  57h
.text:00401144                 mov      ecx, [esp+0Ch+arg_53]
.text:00401148                 mov      edx, [esp+0Ch+arg_3F]
.text:0040114C                 xor      ecx, 48681574h
.text:00401152                 xor      edx, 5EC0940Fh
.text:00401158                 mov      eax, 2
```

Мало того! Рядом с ним валяется указатель 0x57, который "волшебным" образом совпадает с относительным смещением модифицированного нами байта, отсчитываемого от начала тела первой зашифрованной процедуры (развитие зрительной памяти невероятно ускоряет взлом программ). Так вот ты какой, северный олень! Буквально за одну-две секунды мы вышли на след защитного кода, который по замыслу автора мы ни за что не должны были обнаружить! А обнаружили мы его только "благодаря" тому обстоятельству, что и смещение, и содержимое контрольной точки хранилось в программе в открытом виде. Вот если бы оно вычислялось на лету на основе запутанных математических операций... впрочем, не будет повторяться, мы об этом уже говорили.

Хорошо, условимся считать, что поиск по содержимому не дал результатов и хакер остался с защитой один на один. Что он еще может предпринять? А вот что – аппаратная точка останова на модифицированный байт! Да, конечно, мы уже устанавливали ее, но ранее слишком быстро отсекали "лишние" срабатывания. Теперь же настало время заняться этим вопросом вплотную. Вновь запустив порядком затосковавший за это время WDB, мы даем ему уже знакомую команду "ba r4 0x401057" (не обязательно набивать ее на клавиатуре, достаточно лишь нажать стрелку вверх и отладчик сам извлечет ее из истории команд). Первое срабатывание приходится на следующий код:

```
004010C8  C1E902           shr          ecx,2
004010CB F3A5              rep movs     dword ptr [edi],dword ptr [esi]
004010CD 8BCA              mov          ecx,edx
```

Узнаете? Ну да, были мы здесь недавно и все тщательно проанализировали, так и не обнаружив ничего интересного. Идем дальше? Стоп! А точку останова на буфер-приемник кто будет ставить? OK, отдаем отладчику следующую команду: "ba r4 (edi-4)". Почему (edi-4)? Так ведь точки останова срабатывают после выполнения соответствующей им команды, т. е. на момент всплытия отладчика, регистр EDI указывает на ***следующее*** двойное слово, а совсем не на то, которые содержит только что скопированный в буфер код.

Очередное всплытие отладчика приводит нас к коду расшифровщика, уже знакомому нам и не содержащему абсолютно ничего интересного. Не тратя на него понапрасну свое драгоценное время мы отдаем команду "G" и... через серию последовательных всплытий отладчика, отождествляем расшифровку защитного кода, его обратное копирование, явную проверку контрольной суммы и, наконец, сталкивается с малопонятным на первый взгляд кодом, про который можно сказать лишь одно: он использует значение тех самых ячеек защитного кода, которые мы варварски "модернизировали":

```
0040113F E80CFFFFFF        call        00401050
0401144  8B4C2463          mov         ecx,dword ptr [esp+63h]
00401148 8B54244F          mov         edx,dword ptr [esp+4Fh]
0040114C 81F174156848      xor         ecx,48681574h
00401152 81F20F94C05E      xor         edx,5EC0940Fh
00401158 B802000000        mov         eax,2
0040115D 8D4C1104          lea         ecx,[ecx+edx+4]
00401161 8D54240C          lea         edx,[esp+0Ch]
```

Конечно, в данном демонстрационном примере алгоритм "балансировки" распознается без особого труда и серьезных умственных усилий, но как бы там ни было, аппаратные точки останова позволили выявить тот самый код, что осуществляет неявный контроль целостности защиты. Кстати, аппаратных контрольных точек всего четыре, а количество буферов, в которые можно запихать "клоны" копий оригинального кода программы – не ограничено много. Словом, если чуть-чуть постараться, можно очень сильно умерять хакерский пыл – за всеми буферами так просто не уследить, придется анализировать огромное количество кода, лишь часть из которого непосредственно относится к защитному механизму, а все остальное – мусор. Чтобы еще больше запутать хакера можно осуществлять неявный контроль целостности не при каждом запуске программы, а, скажем, на основе датчика случайных чисел, – один раз, эдак, из десяти. "Плавающая" защита – что может быть хуже?! Да, теоретически можно и ее сломать, но: во-первых, даже трудно себе представить сколько на нее придется угробить времени, а, во-вторых, никто не даст и кончика хвоста на отсечение, что выявлены и нейтрализованы все уровни защиты. Ведь аппаратные точки срабатывают лишь в момент обращения к ним, а дизассемблирование бессильно выявить адреса, получаемые на основе сложных математических манипуляций с указателями.

Но все-таки давайте доломаем нашу защиту. В данном конкретном случае мы можем нейтрализовать защитный механизм, просто заменив команду xor ecx, 48681574h на xor ecx, 48689090h, т. е. просто скорректировав "балансир". Однако, при взломе реальной программы хакер должен убедиться, что корректируемый им балансир и не балансирует что-то еще.