НЕЯВНЫЙ САМОКОНТРОЛЬ КАК СРЕДСТВО СОЗДАНИЯ НЕЛОМАЕМЫХ ЗАЩИТ

КРИС КАСПЕРСКИ

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 молчаливо выходит 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, которые ломаются уже не за минуты, а за часы (в редчайших случаях – дни). Алгоритм взлома выглядит приблизительно так:

- подсмотрев контрольную сумму в оригинальной программе, хакер переписывает код функции CalculateCRC, заставляя ее всегда возвращать это значение, не выполняя реальной проверки;
- если защита осуществляет подсчет контрольной суммы различных участков программы и/или разработчик использовал запутанный самомодифицирующийся код, труднопредсказуемым способом меняющий свою контрольную сумму, то хакер может изменить защиту так, чтобы она автоматически самовосстанавливалась после того, как все критические участки будут пройдены;
- 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 антиотладочными приемами, реализовать его на базе собственных виртуальных машин (то есть интерпретаторов), некоторые из них, например, Стрелку 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 и 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. Пусть оптимизирующие компиляторы несколько меняют тело inline-функций с учетом контекста конкретного вызова, эти изменения непринципиальны. 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) програм-

мой 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:

- 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);
- 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 выравнива-

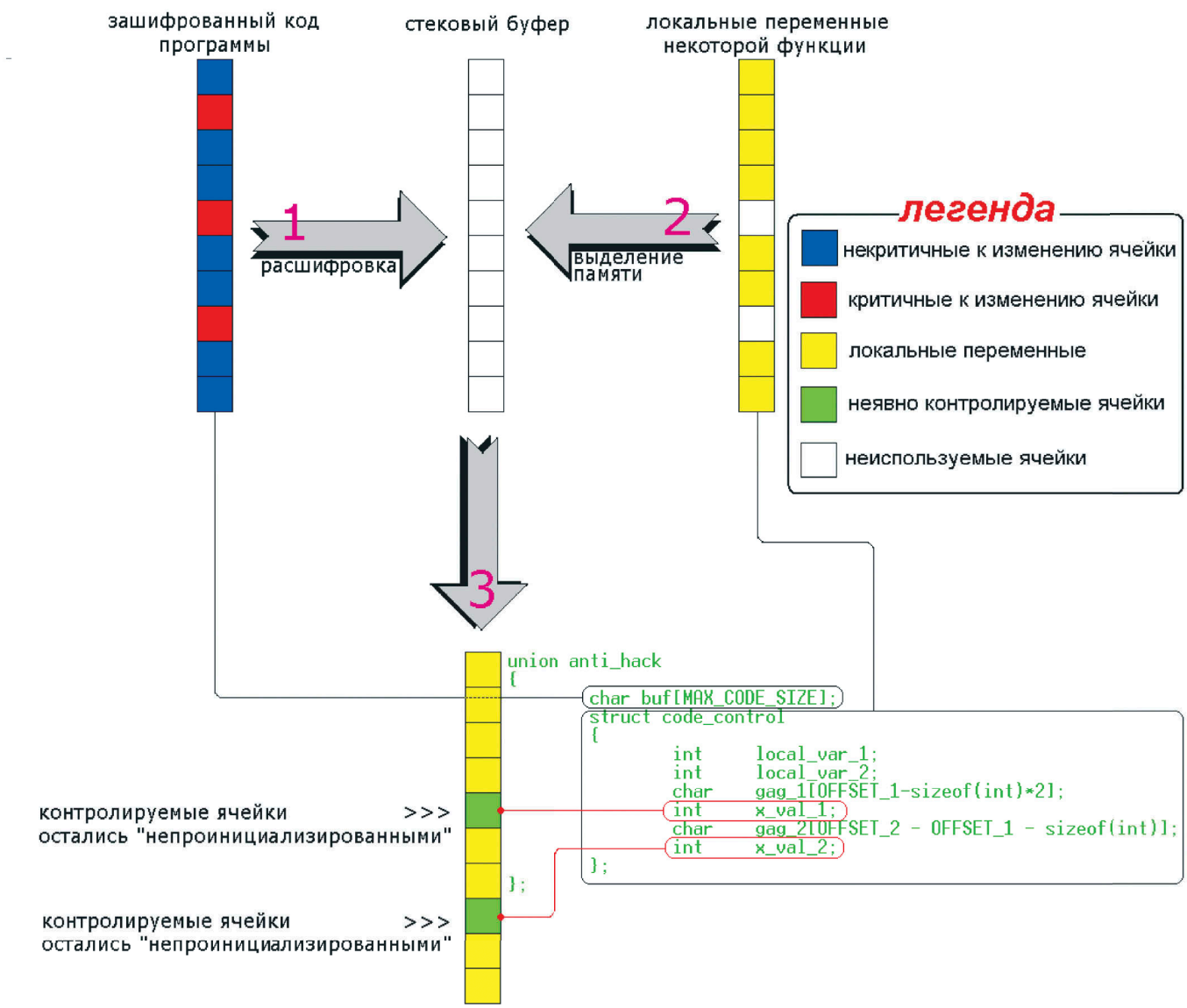

Рисунок 1.

нии структуры (этим мы отвечаем 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 на всеми окажется не так-то просто: во-первых, не хватит контрольных точек, а во-вторых, это просто не придет в голову.

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 трудноломаемой, 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!) соображениям 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. Пусть 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 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 также 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 ими 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-нибудь дизассемблером (например, IDA).

Допустим, критическая часть защиты выглядит так, и нам необходимо проконтролировать целостность условного оператора if, выделенного красным цветом:

```
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+AF?p
.text:00401060                  call   sub_401000
.text:00401065                  test   eax, eax
.text:00401067                  jz     short loc_40107E
.text:00401069                  push   offset aPasswdOk ↵
    ; "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 его часть. Хакер может и не трогать условного перехода, а заменить инструкцию test eax, eax на любую другую инструкцию, сбрасывающую флаг нуля. Также 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:

```
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;
    };
};
```

Исходный текст:

```
Листинг 1.
#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);

}
```

Заключение

Итак, для надежной защиты своих программ от вездесущих хакеров вам совершенно необязательно прибегать к помощи широко разрекламированных, но дорогостоящих электронных ключей (которые, как известно, склонны в самый неподходящий момент "сгорать", к тому же "отвязать" программу от ключа для опытного взломщика не проблема). Также совершенно необязательно спускаться на уровень "голого" ассемблера (ассемблерные защиты непереносимы и к тому же чрезвычайно трудоемки в отладке, про сопровождение я и вовсе молчу). Как было показано выше, практически неломаемую защиту можно создать и на языках высокого уровня, например, на Си/Си++, Delphi или Фортране.

Защиты, основанные на неявном контроле целостности своего кода, ни один здравомыслящий хакер ломать не будет (конечно, при условии, что они не содержат грубых ошибок реализации, значительно упрощающих взлом), ибо трудоемкость взлома сопоставима с разработкой аналогичной программы "с нуля", а ведь каждую новую версию защищенной программы придется ломать индивидуально и опыт предыдущих взломов ничуть не упрощает задачу, а сама логика защиты до безобразия проста. Как защита проверяет целостность своего кода, хакеру более или менее ясно, но вот где конкретно осуществляется такая проверка, он не сможет сказать до тех пор, пока не проанализирует всю программу целиком.

Допустим, объем кода защищенного приложения составляет порядка 256-512 Кб (не слишком много, правда?), тогда при средней длине одной машинной инструкции в 2,5 байта, хакеру придется проанализировать 100 – 200 тысяч ассемблерных команд! При "крейсерской" скорости анализа 10-20 инструкций в минуту (а это предел мечтаний для профессионалов экстракласса) ориентировочное время взлома составит по меньшей мере полтораста часов работы – почти неделя напряженного труда! А на практике (с учетом затрат на тестирование взломанного приложения) даже более того. Другими словами, взломать защиту за разумное время нереально.