Содержание

Глава 3

Уловки с указателями

В соавторстве с Робом Муравски (Rob Murawski)[1]

Да он трещит едва ль не вдвое тише,
Чем на огне у фермера каштаны.
Пугайте им детей![2]

— Уильям Шекспир (William Shakespeare)

Укрощение строптивой, акт 1, сцена 2

Уловки с указателями (pointer subterfuge) — обобщенное название эксплойтов, которые модифицируют значения указателей [162]. Языки программирования C и C++ различают указатели на *объекты* и указатели на *функции*. Тип указателя на `void` или на объектный тип называется *типом объектных указателей*, или *указателей на объекты*. Тип указателя, который указывает на функцию, называется *типом функциональных указателей*, или *указателей на функции*. Указатель на объект типа `T` для краткости будем называть просто "указателем на `T`". C++ определяет также *тип указателей на члены*, который представляет собой тип указателей, которые указывают на нестатические члены классов.

Указатели на функции могут быть перезаписаны злоумышленником таким образом, чтобы передавать управление предоставленному им же шеллкоду. Когда программа выполняет вызов через указатель на функцию, вместо предполагаемого кода выполняется код злоумышленника.

Указатели на объекты также могут быть изменены для выполнения произвольного кода. Если объектный указатель используется как целевой для последующего присваивания, взломщик может управлять адресом для изменения других ячеек памяти.

В этой главе подробно рассматриваются изменения функциональных и объектных указателей. Она отличается от других глав этой книги тем, что в ней рассматриваются механизмы, которые злоумышленник может применять для выполнения произвольного кода

[1] Роберт Муравски (Robert Murawski) — технический сотрудник CERT Program в Carnegie Mellon's Software Engineering Institute (SEI).

[2] Перевод П. Мелковой.

после того, как воспользуется изначально имеющейся уязвимостью (например, переполнением буфера). Предотвращение уловок с указателями достаточно сложное, так что лучше предпринять контрмеры по устранению первоначальной уязвимости. Прежде чем подробно рассматривать уловки с указателями, изучим связь между тем, *как* объявлены данные и *где* они хранятся в памяти.

3.1. Местоположение данных

Существует ряд уязвимостей, которые могут использоваться для замены значений указателей на функцию или объект, в том числе уязвимости, связанные с переполнением буфера.

Переполнение буфера чаще всего вызывается некорректно ограниченными циклами. Чаще всего эти циклы принадлежат одному из перечисленных далее типов.

- *Цикл, ограниченный сверху*: цикл, выполняющий N повторений, где N меньше или равно границе p, а указатель означает последовательность объектов, например от p до $p + N - 1$.
- *Цикл, ограниченный снизу*: цикл, выполняющий N повторений, где N меньше или равно границе p, а указатель означает последовательность объектов, например от p до $p - N + 1$.
- *Цикл, ограниченный адресом последнего элемента массива (Hi)*: цикл увеличивает указатель на элемент массива, пока он не станет равен Hi.
- *Цикл, ограниченный адресом первого элемента массива (Lo)*: цикл уменьшает указатель на элемент массива, пока он не станет равен Lo.
- *Цикл, ограниченный нулевым ограничителем*: цикл увеличивает указатель на элемент массива, пока элемент, на который он указывает, не будет нулевым.

Чтобы переполнение буфера в этих типах циклов можно было использовать для перезаписи указателя на функцию или объект, должны выполняться все следующие условия.

1. Буфер должен быть выделен в том же сегменте, что и целевая функция или указатель на объект.
2. В случае цикла, ограниченного сверху последним элементом массива или нулевым элементом, буфер должен находиться по адресам, меньшим, чем адрес целевой функции или указатель на объект. В случае цикла, ограниченного снизу или первым элементом массива, буфер должен находиться по адресам, бо́льшим, чем адрес целевой функции или указатель на объект.
3. Буфер должен не быть корректно ограничен, а потому должен быть подвержен уязвимости, связанной с переполнением буфера.

Чтобы определить, находится ли буфер в том же сегменте памяти, что и целевая функция или указатель на объект, необходимо понимать, как выделяется память для переменных разного типа в различных сегментах памяти.

Выполняемые файлы UNIX содержат сегмент данных и сегмент BSS[3]. Сегмент данных содержит все инициализированные глобальные переменные и константы. Сегмент BSS содержит все неинициализированные глобальные переменные. Инициализированные глобальные переменные отделены от неинициализированных переменных, так что ассемблеру не нужно записывать содержимое неинициализированных переменных (сегмент BSS) в объектный файл.

В примере 3.1 показаны взаимоотношения между объявлениями переменных и местом их хранения. Комментарии в исходном тексте указывают, где выделено место для каждой из переменных.

Пример 3.1. Объявления данных и организация памяти процесса

```
01 static int GLOBAL_INIT = 1;/* Сегмент данных, глобальная */
02 static int global_uninit;     /* Сегмент BSS, глобальная */
03
04 int main(int argc, char **argv) {  /* Стек, локальная */
05     int local_init = 1;            /* Стек, локальная */
06     int local_uninit;              /* Стек, локальная */
07     static int local_static_init = 1;
08                           /* Сегмент данных, локальная */
09     static int local_static_uninit;
10                              /* Сегмент BSS, локальная */
11           /* Память для buff_ptr - в стеке, локальная */
12                  /* Память выделена в куче, локальная */
13     int *buff_ptr = (int *)malloc(32);
14 }
```

Хотя в организации памяти в UNIX и Windows и есть различия, переменные, показанные в фрагменте программы в примере 3.1, выделяются в Windows так же, как и в UNIX.

■ 3.2. Указатели на функции

В то время как разрушение стека (как и многие атаки в куче) в сегменте данных невозможно, перезапись указателя на функцию одинаково эффективна в любом сегменте памяти.

В примере 3.2 приведена программа с уязвимостью, которая может быть использована для перезаписи указателя на функцию в сегменте BSS. Статический символьный массив `buff`, объявленный в строке 3, и статический указатель на функцию `funcPtr`, объявленный в строке 4, неинициализированы и хранятся в сегменте BSS. Вызов `strncpy()` в строке 6 представляет собой пример небезопасного использования ограниченной функции копирования строк. Переполнение буфера происходит, когда длина `argv[1]` превышает `BUFFSIZE`. Это переполнение буфера можно использовать для передачи управления произвольному коду путем перезаписи значения указателя на функцию адресом шеллкода. Когда программа вызывает функцию с помощью указателя `funcPtr` в строке 7, вместо `good_function()` вызывается шеллкод.

[3] Дословно “BSS” означает “block started by symbol” (блок, начинающийся с символа), но полное его название практически никогда не упоминается.

Пример 3.2. Программа, уязвимая к переполнению буфера в сегменте BSS

```
1 void good_function(const char *str) {...}
2 int main(int argc, char *argv[]) {
3     static char buff[BUFFSIZE];
4     static void (*funcPtr)(const char *str);
5     funcPtr = &good_function;
6     strncpy(buff, argv[1], strlen(argv[1]));
7     (void)(*funcPtr)(argv[2]);
8 }
```

Наивным подходом к устранению переполнения буфера является переобъявление буферов стека как глобальных или локальных статических переменных для уменьшения возможности атак, связанных с разрушением стека. Переобъявление буферов как глобальных переменных является неадекватным решением, потому что, как мы уже видели, переполнение буфера может возникнуть не только в стеке, но и в сегменте данных.

3.3. Указатели на объекты

Указатели на объекты вездесущи в C и C++. Керниган (Kernighan) и Ритчи (Ritchie) [124] отмечают:

> Указатели активно используются в C, отчасти потому, что они обычно ведут к более компактному и эффективному коду, чем код, который можно получить другими способами.

Указатели на объекты в C и C++ используются для ссылок на динамически выделенные структуры, в качестве аргументов функций при передаче по ссылке, массивов и других объектов. Эти указатели на объекты могут быть модифицированы взломщиками, например, при использовании уязвимости, связанной с переполнением буфера. Если впоследствии указатель используется в присваивании как целевой, взломщик может управлять этим адресом для изменения содержимого памяти в других местах (эта методика известна как *запись произвольной памяти* (arbitrary memory write)).

Пример 3.3 содержит уязвимую программу, которая может быть использована для записи произвольной памяти. Эта программа содержит неограниченное копирование памяти в строке 5. После переполнения буфера злоумышленник может перезаписать `ptr` и `val`. Когда впоследствии в строке 6 вычисляется `*ptr = val`, выполняется запись произвольной памяти. Указатели на объекты могут быть изменены злоумышленниками также и благодаря распространенным ошибкам в управлении динамической памятью.

Пример 3.3. Изменение указателя на объект

```
1 void foo(void * arg, size_t len) {
2     char buff[100];
3     long val = ...;
4     long *ptr = ...;
5     memcpy(buff, arg, len);
6     *ptr = val;
7     ...
8     return;
9 }
```

Запись произвольной памяти вызывает особую озабоченность на 32-битных платформах Intel (x86-32), поскольку `sizeof(void*)` равно `sizeof(int)`, `sizeof(long)` и составляет 4 байта. Другими словами, на платформе x86-32 имеется множество возможностей записать 4 байта в 4 байта и переписать адрес в произвольном месте.

3.4. Модификация указателя инструкции

Чтобы злоумышленник мог добиться выполнения произвольного кода на платформе x86-32, необходимо изменить значение указателя инструкций так, чтобы он указывал на шеллкод. Регистр указателя инструкции (`eip`) содержит смещение очередной выполняемой инструкции в текущем сегменте кода.

К регистру `eip` нельзя обратиться непосредственно с помощью программного обеспечения. Он продвигается от одной инструкции к другой при последовательном выполнении кода, или изменяется косвенно с помощью *инструкций передачи управления* (таких, как `jmp`, `jcc`, `call` и `ret`), прерываний и исключений [103].

Инструкция `call`, например, сохраняет информацию о возврате в стеке и передает управление вызываемой функции, определенной целевым операндом. Целевой операнд указывает адрес первой инструкции вызываемой функции. Этот операнд может быть непосредственным значением, регистром общего назначения или ячейкой памяти.

В примере 3.4 приведена программа, использующая указатель на функцию `funcPtr` для вызова функции. Указатель функции объявляется в строке 6 как указатель на статическую функцию, которая принимает в качестве аргумента константную строку. Указателю функции в строке 7 присваивается адрес `good_function`, так что при вызове `funcPtr` в строке 8 на самом деле вызывается `good_function`. Для сравнения функция `good_function()` вызывается статически в строке 9.

Пример 3.4. Пример программы, использующей указатель на функцию

```
01 void good_function(const char *str) {
02     printf("%s", str);
03 }
04
05 int main(void) {
06     static void (*funcPtr)(const char *str);
07     funcPtr = &good_function;
08     (void)(*funcPtr)("hi ");
09     good_function("there!\n");
10     return 0;
11 }
```

В примере 3.5 приведен дизассемблированный код двух вызовов `good_function()` из примера 3.4. Первый вызов (с использованием указателя на функцию) имеет место по адресу `0x0042417F`. Машинный код по этому адресу имеет вид `ff 15 00 84 47 00`. В архитектуре x86-32 имеется несколько вариантов команды вызова. В данном случае код операции `ff` (показан на рис. 3.1) используется вместе с параметром ModR/M `15`, указывающим абсолютный, косвенный вызов.

Пример 3.5. Дизассемблирование указателя на функцию

```
(void)(*funcPtr)("hi ");
00424178 mov esi, esp
0042417A push offset string "hi" (46802Ch)
0042417F call dword ptr [funcPtr (478400h)]
00424185 add esp, 4
00424188 cmp esi, esp

good_function("there!\n");
0042418F push offset string "there!\n" (468020h)
00424194 call good_function (422479h)
00424199 add esp, 4
```

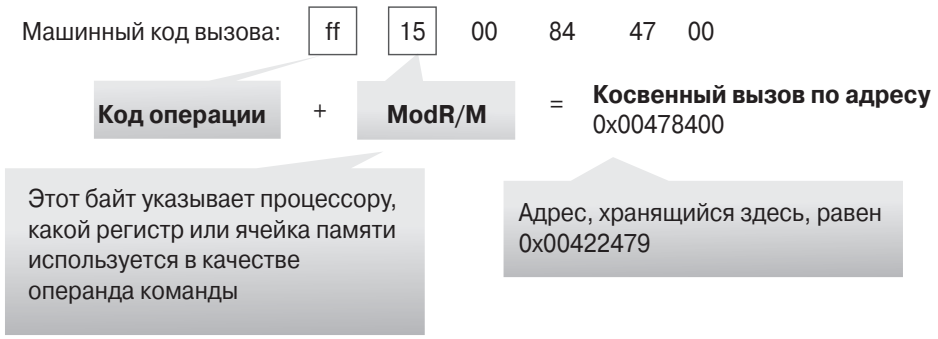

Рис. 3.1. Команда вызова в архитектуре x86-32

Последние 4 байта содержат адрес вызываемой функции (с одним уровнем косвенности). Этот адрес можно также найти в вызове `dword ptr [funcPtr (478400h)]` в примере 3.5. Фактический адрес `good_function()`, хранящийся в памяти по указанному выше адресу, равен `0x00422479`.

Второй, статический, вызов `good_function()` находится по адресу `0x00424194`. Машинный код по этому адресу имеет вид `e8 e0 e2 ff ff`. Здесь код операции `e8` указывает инструкцию вызова. Этот вид вызова подразумевает ближний вызов с указанием относительного смещения по отношению к следующей инструкции. Это смещение отрицательно, что означает, что `good_function()` располагается в более низких адресах.

Эти вызовы `good_function()` являются примерами инструкций вызова, которые могут быть атакованы и которые атакованы быть не могут. Статический вызов использует *непосредственное* значение в качестве относительного смещения, и переписать это значение нельзя, так как оно находится в сегменте кода. Вызов с помощью указателя на функцию использует *косвенную* ссылку, а адрес в указанном месте (обычно в сегменте данных или стека) может быть переписан. Такие косвенные ссылки на функции, как и вызовы функций, которые не могут быть разрешены во время компиляции, могут использоваться злоумышленниками для передачи управления произвольному коду. Конкретные цели для записи произвольной памяти, которая может передавать управление предоставленному взломщиком коду, описываются в оставшейся части данной главы.

3.5. Глобальная таблица смещений

Windows и Linux используют сходные механизмы для связывания и передачи управления библиотечным функциям. Основным различием с точки зрения безопасности является то, что версию Linux можно использовать для взлома, в то время как версия Windows таковой не является.

Бинарный формат по умолчанию в Linux, Solaris 2.x и в SVR4 называется *выполняемый и связываемый формат* (executable and linking format — ELF). Он был разработан и опубликован UNIX System Laboratories (USL) как часть бинарного интерфейса приложений (application binary interface — ABI). Позже стандарт ELF был принят комитетом Tool Interface Standards (TIS)[4] как переносимый формат объектного файла для различных операционных систем x86-32.

Пространство процесса любого бинарного файла ELF включает в себя раздел под названием "глобальная таблица смещений" (global offset table — GOT). GOT содержит абсолютные адреса, делая их доступными независимо от положения в тексте программы и обеспечивает возможность их совместного использования. Эта таблица имеет важное значение для работы процесса динамического связывания. Фактическое содержимое и вид этой таблицы зависят от процессора [212].

Каждая библиотечная функция, используемая программой, имеет запись в GOT, содержащую адрес фактической функции. Это позволяет легко переносить библиотеки в памяти процесса. Перед тем как программа использует функцию впервые, запись содержит адрес компоновщика времени выполнения (runtime linker — RTL). Если функция вызывается программой, управление передается RTL, который находит реальный адрес функции и вставляет его в GOT. Последующие обращения приводят к непосредственному вызову функции с помощью GOT без привлечения RTL.

Адрес записи GOT фиксирован в выполняемых ELF. В результате запись GOT находится в одном и том же адресе образа любого выполняемого процесса. Можно определить расположение записи GOT для функции с помощью команды `objdump`, как показано в примере 3.6. Смещения, указанные для каждой записи перемещения `R_386_JUMP_SLOT`, содержат адрес указанной функции (или функции связывания RTL).

Пример 3.6. Глобальная таблица смещений

```
% objdump --dynamic-reloc test-prog
format:     file format elf32-i386
DYNAMIC RELOCATION RECORDS
OFFSET   TYPE              VALUE
08049bc0 R_386_GLOB_DAT    __gmon_start__
08049ba8 R_386_JUMP_SLOT   __libc_start_main
08049bac R_386_JUMP_SLOT   strcat
08049bb0 R_386_JUMP_SLOT   printf
08049bb4 R_386_JUMP_SLOT   exit
08049bb8 R_386_JUMP_SLOT   sprintf
08049bbc R_386_JUMP_SLOT   strcpy
```

[4] Этот комитет является ассоциацией членов микрокомпьютерной промышленности, созданной для стандартизации программных интерфейсов средств разработки IA-32.

Злоумышленник может переписать запись GOT для функции с адресом шеллкода с помощью записи произвольной памяти. Когда программа впоследствии вызывает функцию, соответствующую этой записи, управление передается шеллкоду. Например, хорошо написанная программа на Си в конечном итоге вызывает функции `exit()`. Перезапись соответствующего элемента GOT для функции `exit()` при вызове этой функции передаст управление по указанному злоумышленником адресу. Таблица связей процедур (procedure linkage table — PLT) формата ELF обладает теми же недостатками [36].

Переносимый формат выполняемых файлов (portable executable — PE) в Windows выполняет функции, аналогичные формату ELF. PE-файл содержит массив структур данных для каждой импортируемой динамически компонуемой библиотеки DLL. Каждая из этих структур дает имя импортированной DLL и указывает на массив указателей на функции (таблицы импортируемых адресов — import address table, IAT). Каждый импортируемый API имеет собственное зарезервированное место в IAT, где загрузчиком Windows записан адрес импортированной функции. После загрузки модуля IAT содержит адрес, по которому осуществляется вызов при обращении к импортируемой функции. Записи IAT могут быть защищены от записи (и являются таковыми), потому что им не требуется меняться во время выполнения.

3.6. Раздел `.dtors`

Другой целью для записи произвольной памяти являются указатели на функции в разделе `.dtors` выполняемых файлов, генерируемых GCC [175]. GNU C позволяет программисту объявить атрибуты функции с помощью ключевого слова `__attribute__`, за которым следуют спецификации атрибута в двойных скобках [75]. Спецификации атрибута включают `constructor` и `destructor`. Конструктор указывает, что данная функция вызывается перед `main()`, а деструктор указывает, что функция вызывается после завершения `main()` или вызова `exit()`.

Программа из примера 3.7 демонстрирует использование атрибутов конструктора и деструктора. Эта программа состоит из трех функций: `main()`, `create()` и `destroy()`. Функция `create()` объявлена в строке 4 как конструктор, а функция `destroy()` в строке 5 — как деструктор. Ни одна из этих функций не вызывается из `main()`, которая просто выводит адрес каждой функции и завершает работу. В примере 3.8 показан результат выполнения программы из примера 3.7. Сначала выполняется конструктор `create()`, затем — `main()` и наконец — деструктор `destroy()`.

Пример 3.7. Программа с атрибутами конструктора и деструктора

```
01 #include <stdio.h>
02 #include <stdlib.h>
03
04 static void create(void) __attribute__ ((constructor));
05 static void destroy(void) __attribute__ ((destructor));
06
07 int main(void) {
08     printf("create: %p.\n", create);
09     printf("destroy: %p.\n", destroy);
10     exit(EXIT_SUCCESS);
11 }
```

```
12
13 void create(void) {
14     puts("create called.\n");
15 }
16
17 void destroy(void) {
18     puts("destroy called.");
19 }
```

Пример 3.8. Вывод программы из примера 3.7

```
% ./dtors
create called.
create: 0x80483a0.
destroy: 0x80483b8.
destroy called.
```

Конструкторы и деструкторы хранятся в разделах `.ctors` и `.dtors` в сгенерированном выполнимом образе ELF. Оба раздела имеют следующую схему.

```
0xffffffff {function-address} 0x00000000
```

Разделы `.ctors` и `.dtors` отображаются на адресное пространство процесса и по умолчанию являются записываемыми. Конструкторы в эксплойтах не используются, поскольку они выполняются до функции `main()`. В результате внимание уделяется деструкторам и разделу `.dtors`.

Содержимое раздела `.dtors` в выполнимом образе можно изучить с помощью команды `objdump`, как показано в примере 3.9. Как видно из приведенного текста, в нем между дескрипторами начала и конца раздела можно найти адрес функции `destroy()` (в формате с прямым порядком байтов).

Пример 3.9. Содержимое раздела `.dtors`

```
1 % objdump -s -j .dtors dtors
2
3 dtors:    file format elf32-i386
4
5 Contents of section .dtors:
6 804959c ffffffff b8830408 00000000
```

Злоумышленник может передать управление произвольному коду путем перезаписи адреса указателя на функцию в разделе `.dtors`. Если целевой бинарный файл может быть прочитан злоумышленником, ему относительно легко определить точное местоположение перезаписи путем анализа ELF-образа.

Интересно, что раздел `.dtors` имеется в наличии даже тогда, когда не указан ни один деструктор. В этом случае раздел состоит из дескриптора начала и конца раздела без адресов функций между ними. При этом злоумышленник все равно может передать управление, переписав дескриптор конца раздела `0x00000000` адресом шеллкода. Если осуществится выход из шеллкода, процесс будет продолжать вызывать последующие адреса, пока не встретит дескриптор конца раздела или пока не произойдет аварийное завершение.

Перезапись раздела .dtors имеет для злоумышленника то преимущество, что этот раздел всегда имеется в наличии и отображается в память.[5] Конечно, раздел .dtors существует только в программах, которые были скомпилированы и скомпонованы с помощью GCC. В некоторых случаях также может быть трудно найти место, в которое можно внедрить шеллкод так, чтобы он остался в памяти после того, как завершится функция main().

3.7. Виртуальные указатели

C++ допускает определения *виртуальных функций*. Виртуальная функция — это член-функция класса, объявленная с использованием ключевого слова virtual. Функции могут быть перекрыты функциями с тем же именем в производном классе. Указатель на производный класс может быть присвоен указателю на базовый класс, и вызов функции осуществляется через указатель. Без виртуальных функций вызывалась бы функция базового класса, так как она связана со статическим типом указателя. Однако при использовании виртуальных функций вызывается функция производного класса, так как она связана с динамическим типом объекта.

В примере 3.10 проиллюстрирована семантика виртуальных функций. Класс a определен как базовый и содержит обычную функцию f() и виртуальную функцию g().

Пример 3.10. Семантика виртуальных функций

```
01 class a {
02   public:
03     void f(void) {
04         cout << "base f" << '\n';
05     };
06
07     virtual void g(void) {
08         cout << "base g" << '\n';
09     };
10 };
11
12 class b: public a {
13   public:
14     void f(void) {
15         cout << "derived f" << '\n';
16     };
17
18     void g(void) {
19         cout << "derived g" << '\n';
20     };
21 };
22
23 int main(void) {
24     a *my_b = new b();
25     my_b->f();
26     my_b->g();
27     return 0;
28 }
```

5 Раздел .dtors не удаляется из бинарного файла командой strip(1).

Класс b является производным от класса a и перекрывает обе функции. В функции main() объявлен указатель my_b на базовый класс, но ему присвоен указатель на объект производного класса b. Когда в строке 25 вызывается невиртуальная функция my_b->f(), вызывается функция f(), связанная с базовым классом a. Когда в строке 26 вызывается виртуальная функция my_b->g(), вызывается функция g(), связанная с производным классом b.

Большинство компиляторов C++ реализуют виртуальные функции с помощью *таблицы виртуальных функций* (virtual function table — VTBL). VTBL представляет собой массив указателей на функции, который используется во время выполнения для диспетчеризации вызовов виртуальных функций. Каждый отдельный объект указывает на VTBL посредством *виртуального указателя* (virtual pointer — VPTR) в заголовке объекта. VTBL содержит указатели на каждую реализацию виртуальной функции. На рис. 3.2 показана структура данных из рассмотренного примера.

Рис. 3.2. Представление VTBL времени выполнения

Можно перезаписать указатели на функции в VTBL или изменить VPTR так, чтобы он указывал на другую произвольную VTBL. Это может осуществляться с помощью записи произвольной памяти или переполнения буфера непосредственно в объект. Буфер перезаписывает VPTR и VTBL объекта и позволяет злоумышленнику вызвать указатель на функцию для выполнения требуемого ему кода. Разрушение VPTR не столь широко распространено, но вполне может применяться, если не работают другие методы [162].

3.8. Функции atexit() и on_exit()

Функция atexit() определена в стандарте языка программирования C. Эта функция регистрирует другие функции, которые вызываются (без аргументов) при нормальном завершении программы. Язык C требует, чтобы реализация поддерживала регистрацию как минимум 32 функций. Функция on_exit() из SunOS обладает аналогичной функциональностью и присутствует также в libc4, libc5 и glibc [27].

Программа, приведенная в примере 3.11, использует atexit() для регистрации функции test() в функции main() в строке 8. Программа присваивает строку "Exiting.\n" глобальной переменной glob в строке 9, до завершения программы. После выхода из функции main() вызывается зарегистрированная функция test(), которая выводит эту строку.

Пример 3.11. Программа, использующая функцию atexit()

```
01 char *glob;
02
03 void test(void) {
04     printf("%s", glob);
05 }
06
07 int main(void) {
```

```
08     atexit(test);
09     glob = "Exiting.\n";
10 }
```

Функция `atexit()` работает путем добавления указанной функции в массив существующих функций, которые вызываются на выходе из программы. При вызове `exit()` эта функция вызывает все функции из массива в порядке "последний вошел, первый вышел" (LIFO). Поскольку и `atexit()`, и `exit()` требуется доступ к массиву, он является глобальным (`__atexit` в операционной системе BSD и `__exit_funcs` в Linux).

В примере 3.12 приведена сессия отладчика `gdb` для программы `atexit`, в которой показаны местоположение и структура массива `atexit`. В сессии отладки установлена точка останова перед вызовом функции `atexit()` в функции `main()`, после чего программа запущена на выполнение. Затем выполняется вызов `atexit()`, который регистрирует функцию `test()`. После регистрации функции `test()` на экран выводится память по адресу `__exit_funcs`. Каждая запись функции, содержащаяся в структуре, состоит из четырех двойных слов. Последнее двойное слово в каждой структуре представляет собой фактический адрес функции. Как показано в листинге, имеется три зарегистрированные функции: `_dl_fini()`, `__libc_csu_fini()` и наша функция `test()`. Можно передать управление произвольному коду, если воспользоваться записью произвольной памяти или переполнением буфера в структуру `__exit_funcs`. Заметим, что функции `_dl_fini()` и `__libc_csu_fini()` содержатся в структуре, даже если уязвимая программа не вызывает функцию `atexit()` явно.

Пример 3.12. Сессия отладки программы `atexit` с помощью `gdb`

```
(gdb) b main
Breakpoint 1 at 0x80483f6: file atexit.c, line 6.
(gdb) r
Starting program: /home/rcs/book/dtors/atexit

Breakpoint 1, main (argc=1, argv=0xbfffe744) at atexit.c:6
6 atexit(test);
(gdb) next
7 glob = "Exiting.\n";
(gdb) x/12x __exit_funcs
0x42130ee0 <init>:     0x00000000 0x00000003 0x00000004 0x4000c660
0x42130ef0 <init+16>:  0x00000000 0x00000000 0x00000004 0x0804844c
0x42130f00 <init+32>:  0x00000000 0x00000000 0x00000004 0x080483c8
(gdb) x/4x 0x4000c660
0x4000c660 <_dl_fini>: 0x57e58955 0x5ce85356 0x81000054 0x0091c1c3
(gdb) x/3x 0x0804844c
0x804844c <__libc_csu_fini>: 0x53e58955 0x9510b850 x102d0804
(gdb) x/8x 0x080483c8
0x80483c8 <test>: 0x83e58955 0xec8308ec 0x2035ff08 0x68080496
```

■ 3.9. Функция `longjmp()`

Стандарт C определяет макрос `setjmp()`, функцию `longjmp()` и тип `jmp_buf`, которые могут использоваться для обхода нормального вызова функций и возврата из них.

Макрос `setjmp()` сохраняет среду вызова для дальнейшего использования функцией `longjmp()`. Функция `longjmp()` восстанавливает среду, сохраненную последним вызовом макроса `setjmp()`. В примере 3.13 показано, как функция `longjmp()` возвращает управление в точку вызова `setjmp()`.

Пример 3.13. Пример использования функции `longjmp()`

```
01 #include <setjmp.h>
02 jmp_buf buf;
03 void g(int n);
04 void h(int n);
05 int n = 6;
06
07 void f(void) {
08     setjmp(buf);
09     g(n);
10 }
11
12 void g(int n) {
13     h(n);
14 }
15
16 void h(int n){
17     longjmp(buf, 2);
18 }
```

В примере 3.14 показана реализация структуры данных `jmp_buf` и связанных с ней определений в Linux. Структура `jmp_buf` (строки 11–15) содержит три поля. Вызывающая среда сохраняется в `__jmpbuf` (объявленном в строке 1). Тип `__jmp_buf` представляет собой целочисленный массив из шести элементов. Инструкции `#define` указывают, какие значения хранятся в каждом элементе массива. Например, базовый указатель (BP) хранится в `__jmp_buf[3]`, а программный счетчик (PC) — в `__jmp_buf[5]`.

Пример 3.14. Реализация структуры `jmp_buf` в Linux

```
01 typedef int __jmp_buf[6];
02
03 #define JB_BX 0
04 #define JB_SI 1
05 #define JB_DI 2
06 #define JB_BP 3
07 #define JB_SP 4
08 #define JB_PC 5
09 define JB_SIZE 24
10
11 typedef struct __jmp_buf_tag {
12     __jmp_buf __jmpbuf;
13     int __mask_was_saved;
14     __sigset_t __saved_mask;
15 } jmp_buf[1];
```

В примере 3.15 показаны команды ассемблера, генерируемые для команды `longjmp()` в Linux. Инструкция `movl` в строке 2 восстанавливает BP, а команда `movl` в строке 3 восстанавливает указатель стека (SP). Строка 4 передает управление сохраненному счетчику PC.

Пример 3.15. Команды ассемблера, генерируемые для команды `longjmp()` в Linux

```
longjmp(env, i)
1  movl i, %eax /* return i */
2  movl env.__jmpbuf[JB_BP], %ebp
3  movl env.__jmpbuf[JB_SP], %esp
4  jmp (env.__jmpbuf[JB_PC])
```

Функция `longjmp()` может быть использована злоумышленником путем перезаписи значения PC в буфере `jmp_buf` значением начала шеллкода. Эту задачу можно решить с помощью записи произвольной памяти или переполнения буфера непосредственно в структуру `jmp_buf`.

■ 3.10. Обработка исключений

Исключение представляет собой любое событие, которое находится вне обычных операций процедуры. Например, деление на ноль вызывает исключение. Многие программисты реализуют блоки обработчика исключений, чтобы обработать эти особые случаи и избежать неожиданного завершения программы. Кроме того, обработчики исключений объединяются в цепочки и вызываются в определенном порядке до тех пор, пока один из них не сможет обработать происшедшее исключение.

Microsoft Windows поддерживает три следующие типа обработчиков исключений. Операционная система вызывает их в указанном порядке до тех пор, пока один из них не сможет успешно обработать исключение.

1. Векторная обработка исключений (vectored exception handling — VEH). Эти обработчики вызываются первыми, до структурной обработки исключений. Данные обработчики были добавлены в Windows XP.
2. Структурная обработка исключений (structured exception handling — SEH). Эти обработчики реализованы как обработчики исключений на базе функций и на базе потоков.
3. Системная обработка исключений по умолчанию. Это фильтр и обработчик глобальных исключений для всего процесса, который вызывается, если исключение не обработано предыдущими обработчиками.

Структурная обработка исключений и системная обработка исключений по умолчанию рассматриваются в следующих разделах. Векторная обработка исключений не рассматривается, так как она не используется в эксплойтах сколь-нибудь широко.

Структурная обработка исключений

SEH обычно реализуется на уровне компилятора с помощью блоков `try...catch`, как показано в примере 3.16.

Пример 3.16. Блок `try...catch`

```
1 try {
2     // Выполняем здесь всю работу
3 }
```

```
4 catch(...){
5     // Здесь обрабатывается исключение
6 }
7 __finally {
8     // Очистка, завершение работы
9 }
```

Любое исключение, сгенерированное в процессе работы блока `try`, обрабатывается соответствующим блоком `catch`. Если блок `catch` не в состоянии обработать исключение, оно передается блоку предыдущей области видимости. Ключевое слово `__finally` является расширением языка C/C++ от Microsoft и используется для указания блока кода, который вызывается для проведения необходимых действий по очистке после блока `try`, например для закрытия открытых файлов или освобождения выделенной памяти. Эти действия выполняются независимо от того, как был осуществлен выход из блока `try`.

Для структурной обработки исключений Windows реализует специальную поддержку для обработчиков исключений потока. Код, создаваемый компилятором, записывает адрес указателя на структуру `EXCEPTION_REGISTRATION` в адрес, на который указывает регистр сегмента `fs`. Эта структура определяется на языке ассемблера в файле `EXSUPP.INC` в исходных текстах среды выполнения Visual C++ и содержит два элемента данных, как показано в примере 3.17.

Пример 3.17. Определение структуры `EXCEPTION_REGISTRATION`

```
1 _EXCEPTION_REGISTRATION struc
2     prev        dd    ?
3     handler     dd    ?
4 _EXCEPTION_REGISTRATION ends
```

В этой структуре `prev` является указателем на предыдущую структуру `EXCEPTION_HANDLER` в цепочке, а `handler` — указателем на функцию обработки исключений.

Windows налагает несколько правил на обработчик исключений для обеспечения целостности цепочки обработчиков исключений и системы.

1. Структура `EXCEPTION_REGISTRATION` должна располагаться в стеке.
2. Структура `prev` типа `EXCEPTION_REGISTRATION` должна находиться в стеке по более высокому адресу.
3. Структура `EXCEPTION_REGISTRATION` должна быть выровнена на границу двойного слова.
4. Если в заголовке исполнимого образа перечисляются адреса обработчика `SAFE SEH`,[6] адрес обработчика должен быть указан как обработчик `SAFE SEH`. Если в заголовке не перечислены адреса обработчика `SAFE SEH`, может вызываться любой обработчик структурных исключений.

Компилятор инициализирует кадр стека в прологе функции. Типичный пролог функции в Visual C++ показан в примере 3.18. Этот код устанавливает кадр стека, показанный в табл. 3.1. Компилятор резервирует место в стеке для локальных переменных. Поскольку локальные переменные следуют сразу за адресом обработчика исключений, адрес

[6] Компиляторы Microsoft Visual Studio .NET обеспечивают построение кода с поддержкой SAFE SEH, но такая проверка обеспечивается только в Windows XP Service Pack 2.

обработчика исключения может быть заменен произвольным значением с помощью переполнения буфера в стековой переменной.

Пример 3.18. Инициализация кадра стека

```
1  push      ebp
2  mov       ebp, esp
3  and       esp, 0FFFFFFF8h
4  push      0FFFFFFFFh
5  push      ptr [Exception_Handler]
6  mov       eax, dword ptr fs:[00000000h]
7  push      eax
8  mov       dword ptr fs:[0], esp
```

Таблица 3.1.

Смещение в стеке	Описание	Значение
-0x10	Обработчик	[Обработчик_исключения]
-0x0C	Предыдущий обработчик	fs:[0] при запуске функции
-8	Защита	-1
-4	Сохраненные значения	ebp ebp
0	Адрес возврата	Адрес возврата

В дополнение к перезаписи указателей отдельных функций можно также заменить указатель в блоке среды потока (thread environment block — TEB), который указывает на список зарегистрированных обработчиков исключений. Злоумышленнику необходимо сделать копию элемента списка и изменить первое поле обработчика исключений с помощью записи произвольной памяти. Хотя в последние версии Windows и были добавлены проверки элементов списка, Личфилд (Litchfield) продемонстрировал множество успешных эксплойтов для таких случаев [135].

Системная обработка исключений по умолчанию

Windows предоставляет глобальный фильтр исключений и обработчик для всего процесса; он вызывается, если предыдущий обработчик не может обработать исключение. Многие программисты реализуют фильтр необработанного исключения для всего процесса в целом для корректной обработки неожиданных ошибок и отладки.

Функция фильтра необработанного исключения назначается с помощью функции `SetUnhandledExceptionFilter()`. Эта функция вызывается как последний уровень обработчика исключений для процесса. Однако, если злоумышленник перезапишет нужные адреса с помощью записи произвольной памяти, фильтр необработанного исключения может быть перенаправлен для выполнения произвольного кода. Однако Windows XP Service Pack 2 кодирует адреса указателей, что делает такой взлом нетривиальной операцией. В реальной ситуации злоумышленнику было бы сложно правильно закодировать значения указателей без знания подробной информации о процессе.

■ 3.11. Стратегии противодействия

Наилучший способ предотвратить уловки с указателями — это устранить уязвимости, которые позволяют злоумышленнику перезаписать память. Уловки с указателями могут быть результатом перезаписи указателей на объекты (как показано в этой главе), вызваны распространенными ошибками управления динамической памятью (глава 4, "Управление динамической памятью") и уязвимостями форматных строк (глава 6, "Форматированный вывод"). Устранение этих источников уязвимости является наилучшим способом предотвращения уловок с указателями. Есть и другие контрмеры, которые могут помочь в решении этого вопроса, но на которые нельзя полностью полагаться как на решение данной проблемы.

Стековые канарейки

В главе 2, "Строки", мы уже рассматривали стратегии противодействия уязвимостям, обусловленным ошибками при работе со строками и разрушением стека, включая стековые канарейки. К сожалению, канарейки полезны только при атаках, которые используют переполнение буфера в стеке и пытаются перезаписать указатель стека или другую защищенную область. Канарейки не защищают от эксплойтов, которые изменяют переменные, указатели на объекты или на функции. Канарейки не в состоянии предотвратить переполнение буфера в произвольном месте, включая сегмент стека.

W^X

Одна из контрмер против описанных действий — уменьшение привилегий уязвимых процессов. Стратегия W^X, описанная в главе 2, "Строки", позволяет делать сегмент памяти либо записываемым, либо выполняемым, но не то и другое одновременно. Однако такая стратегия не может предотвратить перезапись, например, памяти, обращение к которой требуется функцией `atexit()`, так как она должна быть во время выполнения программы и записываемой, и выполняемой. Кроме того, эта стратегия не так уж широко реализована.

Кодирование и декодирование указателей на функции

Вместо указателя на функцию программа может хранить зашифрованную версию указателя. Чтобы перенаправить указатель на другой код, злоумышленнику необходимо взломать используемый шифр. Аналогичные подходы рекомендуются для работы с конфиденциальными или личными данными, такими как ключи шифрования или номера кредитных карт.

Томас Плам (Thomas Plum) и Арджун Биджанки (Arjun Bijanki) [165] на конференции WG14 в Санта-Кларе в сентябре 2008 года предложили добавить в стандарт C11 функции `encode_pointer()` и `decode_pointer()`. По предназначению эти функции аналогичны (хотя и отличаются в деталях) функциям из Microsoft Windows (`EncodePointer()` и `DecodePointer()`), которые используются в библиотеках времени выполнения Visual C++.

Функция `encode_pointer()` имеет следующую спецификацию.

Обзор

```
#include <stdlib.h>
void (*)() encode_pointer(void(*pf)());
```

Описание

Функция encode_pointer() выполняет преобразование аргумента pf, такое, что функция decode_pointer() выполняет обратное преобразование.

Возврат

Функция возвращает результат преобразования.

Функция decode_pointer() имеет следующую спецификацию.

Обзор

```
#include <stdlib.h>
void (*)() decode_pointer(void(*pf)());
```

Описание

Функция decode_pointer() выполняет преобразование аргумента pf, обратное преобразованию функцией encode_pointer().

Возврат

Функция возвращает результат преобразования.

Эти две функции определены таким образом, что для любого указателя на функцию pfn, использованного в выражении

```
decode_pointer(encode_pointer((void(*)())pfn));
```

результат приведения к типу указателя pfn дает значение pfn.

Однако описанная взаимосвязь между encode_pointer и decode_pointer() является неверной, если вызовы encode_pointer() и decode_pointer() имеют место при определенных условиях, определенных реализацией. Например, если вызовы происходят в различных процессах выполнения, функции не являются обратными одна к другой. В такой реализации метод преобразования указателя может, например, использовать номер процесса в алгоритме кодирования/декодирования.

Процесс кодирования указателя не предотвращает переполнение буфера или запись произвольной памяти, но усложняет использование такой уязвимости. Кроме того, предложение WG14 было отклонено, поскольку было сочтено, что кодирование и декодирование лучше выполнять компилятором, а не библиотекой. Так что кодирование и декодирование осталось вопросом “качества реализации”.

CERT’у в настоящее время не известен ни один компилятор, который бы выполнял кодировку и декодирование указателей на функции, даже как необязательную функциональность. Программисты, разрабатывающие код для Microsoft Windows, должны использовать для кодирования указателей на функции EncodePointer() и DecodePointer(). Корпорация Майкрософт использует эти функции в коде своей системы для предотвращения записи в произвольную память. Однако для того, чтобы это решение было эффективным, должны быть защищены все указатели приложения (в том числе указатели на функции). Для других платформ необходимо сначала разработать соответствующую функциональность.

3.12. Резюме

Переполнение буфера может использоваться для перезаписи указателя на функцию или объект таким же образом, как и атака разрушением стека используется для перезаписи адреса возврата. Возможность перезаписать указатель на функцию или объект зависит от близости переполняемого буфера к целевому указателю, но чаще всего они находятся в одном и том же сегменте памяти.

Перезапись указателя на функцию позволяет злоумышленнику непосредственно передать управление предоставленному им коду. Возможность модификации указателя на объект и присваивания значения обеспечивает запись произвольной памяти.

Независимо от среды имеется множество возможностей передачи управления произвольному коду, обеспечиваемых записью произвольной памяти. Одни из них являются результатом возможностей стандарта языка программирования C (например, `longjmp()`, `atexit()`), другие специфичны для конкретных компиляторов (например, раздел `.dtors`) или операционных систем (например, `on_exit()`). Кроме возможностей, описанных в этой главе, имеется множество других — известных и пока неизвестных.

Запись произвольной памяти может легко разрушить защиту с использованием канареек. Защита целевой памяти от записи затруднена из-за наличия программ, в которых требуется модификация значений в этой памяти (например, указателей на функции) во время выполнения. Одной из стратегий предупреждения является хранение указателей только в закодированном виде.

Переполнение буфера в любом сегменте памяти может быть использовано для выполнения произвольного кода, так что перемещение переменных из стека в сегмент данных или кучу решением не является. Наилучший подход к предотвращению уловок с указателями в результате переполнения буфера заключается в ликвидации условий переполнения буфера.

В следующей главе рассматриваются уязвимости и эксплойты, связанные с памятью в куче. Эти уязвимости позволяют злоумышленнику перезапись адреса в произвольном месте. Такие эксплойты являются результатом уязвимостей, связанных с переполнением буфера в куче, записью в освобожденную память и двойным освобождением памяти.

3.13. Дополнительная литература

Атаки на основе уловок с указателями были разработаны главным образом в ответ на создание стековых канареек, проверяемых в StackGuard и других продуктах. Рафал Войчук (Rafal Wojtczuk) рассматривает перезапись элемента GOT для преодоления неисполнимого стека Solar Designer [231]. Статья 1999 года Мэтта Коновера (Matt Conover), посвященная работе с кучей, включает несколько примеров атак, основанных на уловках с указателями [42]. Бульба (Bulba) и Жерардо Ришарт (Gerardo Richarte) также описывают эксплойты, преодолевающие схемы защиты StackShield и StackGuard [30, 173]. Дэвид Личфилд (David Litchfield) рассматривает нападения на обработчики исключений [135, 136]. В *Phrack* 56 [176] имеется материал о разрушении указателя на таблицу виртуальных функций C++. Хороший обзор атак на основе уловок с указателями можно найти в [162].