

“Я собираюсь обрушить на этих варваров железный молот цивилизации.”

Таня фон Дегуршафф

Во время проверок я заметил печальную тенденцию — незнание весьма важных понятий и инструментов для написания проектов. Причем, не только у новичков, но и “высокоуровневых”. Данная статья — попытка исправить это и помочь Вам закрыть возможные пробелы в Ваших знаниях.

Valgrind, Leaks, утечки, segmentation и тому подобное

Утечка— это явление, наблюдающееся, когда указатель на ранее выделенную память потерян. Т. е. данная память программе становится недоступна как для освобождения, так и для использования. И да, не важно, где ранее хранился указатель на выделенную память, важен сам факт его потери.

ОС — операционная система.

Нужно ли освобождать всю выделенную память перед выходом из программы? — Нет!

Если сказать кратко, при выделении ресурсов процессу **ОС** запоминает все, что выделила, и при использовании **exit** или **return** из **main** сама все ресурсы освободит. Но стоит помнить, что это работает только для приложений пользовательского уровня **Ring 3**. В случае с драйверами **Ring 0** такое не прокатит -нужно самому все освобождать.

Если ОС все сама освободит, тогда зачем мне этим заниматься? — ОС, конечно, все освободит, но сделает она это только после завершения работы программы. А до этого утечки с каждой минутой будут все больше и больше накапливаться, пожирая драгоценную оперативную память. Особенно если программа активно потребляет память во время своей работы.

Как мне находить утечки? — Для этого есть специальные утилиты: **Valgrind** и встроенный инструментариий в **MAC OS**. Можно их также вручную детектить.

Ручной способ определения leaks:

Во-первых, нужно придерживаться правила: все, что выделил, должно быть освобождено. Но никто не защищен от ошибок...

Во-вторых, при выходе во время отладки своей программы все-таки освобождай всю выделенную память несмотря на то, что по факту это необязательно.

В-третьих, обернуть вызовы **malloc** и **free** в свои функции, где вести в глобальном счетчике подсчёт выделенной и освобождённой памяти. Если счётчик не будет перед выходом из программы равен нулю, то все, ты попал. Но не стоит забывать, что некоторые функции, например, **getcwd**, сами выделяют память, это нужно учитывать.

Как ускорить работу программы? — Любой вызов функции ядра является очень ресурсоёмким, так как надо переходить с **Ring 3** в **Ring 0**, попутно сохраняя состояние процесса. Стоит для ускорения минимизировать использование функций, наподобие **write**. Например, в случае с **write** можно использовать буферизацию, чтобы не писать побайтно.

Также очень сильно влияет используемый алгоритм. До сих пор работаешь с пузырьковой сортировкой? Начни хотя бы использовать сортировку вставками...

При компиляции можно использовать флаг **-O2**(если в задании нет запрета на это). Имеются и другие флаги для оптимизации, но при их использовании есть шанс, что после оптимизации программа будет работать не так, как планировалось.

Что такое куча(heap)? — Зачастую программисту требуется делать многочисленные малообъёмные выделения памяти. Использовать системную функцию для выделения 50 байт? - думаю, ответ очевиден. И да, система по факту выделяет память страничками: то есть если у нее попросить 50 байт, выделить меньше, чем страницу, она не сможет - в случае 32-битной системы это 4096 байт. Поэтому и появилась потребность в куче - менеджере памяти пользовательского уровня **Ring 3**. Данный менеджер во время старта программы за

раз выделяет относительно большой объём памяти, допустим, 1 мегабайт. И с этого запаса по чуть-чуть выделяет ее тебе. В случае, если у него недостаточно запасов, обращается к системе за дополнительной памятью. Таким образом, мы получаем быстроедействие и экономим память. Обычно менеджер кучи также хранит в отданной тебе памяти свою служебную информацию, но только выше выданного тебе указателя. Так что попытки записать за границами выделенной тебе памяти приведут к повреждению менеджера кучи и к трудно уловимым багам.

Исполняемые файлы: это что такое? — Файлы, внутри которых по сути содержится такой же набор единичек и нулей, как и в любом другом файле. Единственное отличие в том, как мы будем обрабатывать информацию, находящуюся в нем. А обрабатывать будем как обычную структуру. Вот пример одной структуры формата исполняемого файла в **Windows**:

```
typedef struct _IMAGE_OPTIONAL_HEADER {  
    WORD        Magic;  
    BYTE        MajorLinkerVersion;  
    BYTE        MinorLinkerVersion;  
    DWORD       SizeOfCode;  
    DWORD       SizeOfInitializedData;  
    DWORD       SizeOfUninitializedData;  
    DWORD       AddressOfEntryPoint;  
    DWORD       BaseOfCode;  
    DWORD       BaseOfData;  
    DWORD       ImageBase;  
    DWORD       SectionAlignment;  
    DWORD       FileAlignment;  
    WORD        MajorOperatingSystemVersion;  
    WORD        MinorOperatingSystemVersion;  
    WORD        MajorImageVersion;  
    WORD        MinorImageVersion;  
    WORD        MajorSubsystemVersion;  
    WORD        MinorSubsystemVersion;  
    DWORD       Win32VersionValue;  
    DWORD       SizeOfImage;  
    DWORD       SizeOfHeaders;  
    DWORD       CheckSum;  
    WORD        Subsystem;  
    WORD        DllCharacteristics;  
    DWORD       SizeOfStackReserve;  
    DWORD       SizeOfStackCommit;  
    DWORD       SizeOfHeapReserve;  
    DWORD       SizeOfHeapCommit;  
    DWORD       LoaderFlags;  
    DWORD       NumberOfRvaAndSizes;  
    IMAGE_DATA_DIRECTORY DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES];  
} IMAGE_OPTIONAL_HEADER32, *PIMAGE_OPTIONAL_HEADER32;
```

Нетяжело догадаться, что, зная структуры в исполняемом файле, спокойно можно внедрить в него свой код, что и делают вирусы - привет ветка **Virus**. Правда, обычно файлы защищают от внедрения, причем не только контрольной суммой **CheckSum**, но и более сложными методами.

В этих структурах хранятся вся необходимая информация, нужная **ОС** для запуска данного файла: какие библиотеки нужно подгрузить (**секция импорт**), размер кучи, стека и т. п.

Что такое выравнивание данных и зачем оно нужно? — Выравнивание — это процесс расположения данных кратно своему размеру, а не в хаотичном порядке. Причем размер типа данных можно представить в виде 2 в

энной степени. В частности, **long double** - это плавающее число четырёхкратной точности размером в 10 байт, и это не есть 2 в энной степени - ближайший размер, подходящий под это условие, 16 байт, поэтому фактически **long double** занимает в памяти 10 байт реально используемых и плюс 6 байт для кратности.

Данные располагаются кратно своему размеру, когда полученный адрес, делится на размер типа переменной без остатка. Например, **long double** может располагаться на таких адресах: 0x10, 0x4000, 0x4080 ... Но отнюдь не на таких: 0x13, 0x4004, 0x4081. Таким образом, все переменные в памяти располагаются кратно своему размеру, структура же располагается в памяти кратно наибольшей переменной.

Делается это для быстродействия — процессор способен читать только выравненные данные, причем по размеру своего слова (в **x64 – 8 байт, x32 – 4 байта**). Да-да, даже когда тебе нужен лишь один байт, он на **x64** системе по факту считается как 8 байт, но в переменную запишется лишь один байт. Если же ты захочешь получить 8 байт на **x64** системе, и они не выравнены, системе придется два раза считывать по 8 байт, совместить эти данные, и после этого ты только получишь нужные тебе байты, что, как нетрудно догадаться, не есть эффективно.

Плюс некоторые процессоры выдают исключение при доступе к невыравненным данным или же их можно перевести в этот режим. Большинство **SSE** инструкций при работе с невыравненной памятью тоже дают исключение.

Вот пример на выравнивание:

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

typedef struct
{
    char ups1;
    int ups2;
    double ups3;
} TMP;

int main(void)
{
    TMP test1; printf("%p %zu", &test1, sizeof(TMP));
    return (0);
}
```

Не умею писать make, что делать? — читать.

[Эффективное использование GNU Make](#)

[GNU Make](#)

[Введение в make](#)

Хочу почитать про инструмент Leaks — [Tracking Memory Usage](#)

Valgrind

Чтобы наиболее эффективно использовать **Valgrind**, программу нужно скомпилировать с отладочными символами. То есть при компиляции использовать флаг **-g**.

Сам по себе очень хороший инструмент, но, к сожалению, для **Mac OS** он не является родным, ну и плюс, как и любое человеческое творение, не лишён определенных недостатков. Если быть точнее: не на каждом маке он в принципе работает так, как от него ожидается.

Зачастую детектит **утечки**, которых в принципе нет, вот, например:

```
#include <stdlib.h>

int main(void)
{
    void *leaks;
    static void *leaks_not;

    if ((leaks_not = malloc(200)) != 0)
        return (0);
    if ((leaks = malloc(200)) != 0)
        return (0);
    return (0);
}
```

```
==5602== Memcheck, a memory error detector
==5602== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==5602== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==5602== Command: ./tests
==5602==
==5602== HEAP SUMMARY:
==5602==   in use at exit: 18,362 bytes in 160 blocks
==5602==   total heap usage: 176 allocs, 16 frees, 24,506 bytes allocated
==5602==
==5602== LEAK SUMMARY:
==5602==   definitely lost: 200 bytes in 1 blocks
==5602==   indirectly lost: 0 bytes in 0 blocks
==5602==   possibly lost: 72 bytes in 3 blocks
==5602==   still reachable: 400 bytes in 7 blocks
==5602==   suppressed: 17,690 bytes in 149 blocks
==5602== Rerun with --leak-check=full to see details of leaked memory
==5602==
==5602== For lists of detected and suppressed errors, rerun with: -s
==5602== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 4 from 4)
```

Здесь мы не потеряли ни один указатель на выделенную память, но все же он ухитрился найти утечку.

Также показывает, что **утечки** есть у **Mac**, и поэтому у многих укоренилось представление, что “маки текут”, но это полная ерунда. Если бы у них было столько **утечек**, тогда бы в течение часа вся оперативная память была бы израсходована. **OS** была написана людьми весьма неглупыми. Хоть это не отменяет факт возможности того, что где-то утечки есть, но их точно не найдешь с помощью **Valgrind**. Вот пример:

```
/Users/anttila/Subject/42sh
> ==8525== 72 bytes in 3 blocks are possibly lost in loss record 31 of 55
==8525==   at 0x100020C2: calloc (vg_replace_malloc.c:705)
==8525==   by 0x1005F0846: map_images_unlock (in /usr/lib/libobjc.A.dylib)
==8525==   by 0x10004F0F6: objc_objc::s::table::retainCount() (in /usr/lib/libobjc.A.dylib)
==8525==   by 0x100040386: dyld::notifyBatchPartial(dyld_image_states, bool, char const* (*)(dyld_image_states, unsigned int, dyld_image_info const*), bool, bool) (in /usr/lib/dyld)
==8525==   by 0x100042255: dyld::registerdyldNotifiers(dyld::ImageInfo const*, mach_header const* const*, void (*)(char const*, mach_header const*), void (*)(char const*, mach_header const*)) (in /usr/lib/dyld)
==8525==   by 0x100236006: dyld_objc_notify_register (in /usr/lib/system/libdyld.dylib)
==8525==   by 0x1005F0074: objc_init (in /usr/lib/libobjc.A.dylib)
==8525==   by 0x100165685: _os_object_init (in /usr/lib/system/libdispatch.dylib)
==8525==   by 0x100165634: libdispatch_init (in /usr/lib/system/libdispatch.dylib)
==8525==   by 0x100055905: libsystem_initializer (in /usr/lib/libsystem.B.dylib)
==8525==   by 0x100052A16: ImageLoaderMachO::doInitialization(ImageLoader::LinkContext const&) (in /usr/lib/dyld)
==8525==   by 0x100052C10: ImageLoaderMachO::doInitialization(ImageLoader::LinkContext const&) (in /usr/lib/dyld)
==8525==
==8525== 2,064 (16 direct, 2,048 indirect) bytes in 1 blocks are definitely lost in loss record 51 of 55
==8525==   at 0x100002091: malloc (vg_replace_malloc.c:352)
==8525==   by 0x100528101: _rc_table_new (in /usr/lib/system/libsystem_notify.dylib)
==8525==   by 0x100523F85: _notify_init_globals (in /usr/lib/system/libsystem_notify.dylib)
==8525==   by 0x100523C46: _os_once (in /usr/lib/system/libsystem_platform.dylib)
==8525==   by 0x100531A01: _os_alloc_once (in /usr/lib/system/libsystem_platform.dylib)
==8525==   by 0x1005240A0: _notify_fork_child (in /usr/lib/system/libsystem_notify.dylib)
==8525==   by 0x100058206: libsystem_atfork_child (in /usr/lib/libsystem.B.dylib)
==8525==   by 0x100246436: fork (in /usr/lib/system/libsystem_c.dylib)
==8525==   by 0x10000135C: fu_event (main.c:24)
==8525==   by 0x100001358: main (main.c:49)
==8525==
==8525== 2,064 (16 direct, 2,048 indirect) bytes in 1 blocks are definitely lost in loss record 52 of 55
==8525==   at 0x100002091: malloc (vg_replace_malloc.c:352)
==8525==   by 0x100528101: _rc_table_new (in /usr/lib/system/libsystem_notify.dylib)
==8525==   by 0x100523F85: _notify_init_globals (in /usr/lib/system/libsystem_notify.dylib)
==8525==   by 0x100523C46: _os_once (in /usr/lib/system/libsystem_platform.dylib)
==8525==   by 0x100531A01: _os_alloc_once (in /usr/lib/system/libsystem_platform.dylib)
==8525==   by 0x1005240A0: _notify_fork_child (in /usr/lib/system/libsystem_notify.dylib)
==8525==   by 0x100058206: libsystem_atfork_child (in /usr/lib/libsystem.B.dylib)
==8525==   by 0x100246436: fork (in /usr/lib/system/libsystem_c.dylib)
==8525==   by 0x10000135C: fu_event (main.c:24)
==8525==   by 0x100001358: main (main.c:49)
```

Если внимательно присмотреться, то видно, что все так называемые “утечки” происходят в системных вызовах. Что, в любом случае, к нашим программам отношения не имеет.

А вот пример настоящей **утечки** и реакция **Valgrind** на нее:

```

13 #include <stdlib.h>
14
15 int → → main(void)
16 {
17     char → → **spl;
18
19     if ((spl = malloc(sizeof(char *))) == 0)
20         return (0);
21     if ((spl[0] = malloc(sizeof(char *))) == 0)
22         return (0);
23     free(spl);
24 }

```

8 bytes in 1 blocks are definitely lost in loss record 1 of 42
 at 0x10009F691: malloc (vg_replace_malloc.c:312)
 by 0x1000011F0: main (main.c:21)

Как видно, мы создаем массив с указателями, но освободили только сам массив, а вот про указатель совсем забыли, что и показывает нам **Valgrind**. Чем это опасно, я писал выше (см. утечки)

Как по мне, нахождение **утечек** — это не самая полезная функция **Valgrind**, как ошибочно многие думают. Есть гораздо более интересные возможности:

определение факта использования непроинициализированной памяти или переменных и определение попытки чтения вне границ выделенной памяти.

Во всех этих случаях я не заметил, чтобы **Valgrind** неправильно детектил, в отличие от **утечек**. Так что советую к его предупреждению отнестись серьезно.

Valgrind способен определить, используется ли неинициализированная память. Вот пример:

```

13 #include <stdlib.h>
14 #include <string.h>
15
16 int → → main(void)
17 {
18     char → → → *seg;
19
20     if ((seg = malloc(200)) == 0)
21         return (0);
22     strlen(seg);
23     memset(seg, 'F', 200);
24     return (0);
25 }

```

Conditional jump or move depends on uninitialised value(s)
 at 0x1000A1C99: strlen (vg_replace_strmem.c:477)
 by 0x1000011DB: main (main.c:22)

Здесь мы с помощью **strlen** пытаемся работать с памятью, значение которой не установили, о чем нам и сообщает **Valgrind**. Это чревато тем, что результат работы нашей программы будет зависеть от воли случая, так как никто не знает, вернет ли **malloc** память, забитую нулями, или, может, буквами 'F'?

Также он ловит переменные, которые при определенных условиях могут быть непроинициализированы:

```

14 int → → main(void)
15 {
16     char → s1[] = "test";
17     char → *s;
18     int → i;
19
20     s = s1;
21     s[0] = 0;
22     while (s++[0] != 0)
23         i = 0;
24     if (i == 0)
25         return (0);
26     return (0);
27 }

```

Conditional jump or move depends on uninitialised value(s)
 at 0x1000011F7: main (main.c:24)

В данном случае мы никогда не войдем в цикл и не сможем проинициализировать **i**. Таким образом, в переменной может быть любое значение, и от неопределенного значения зависит, что вернет наша функция. Поэтому не стоит удивляться, если программа будет завершаться с **segmentation** по не совсем понятным причинам.

А также определяет, если неинициализированная память или переменные передаются в системные функции:

```

13 #include <unistd.h>
14
15 int main(void)
16 {
17     char s1[] = "test";
18     char *s;
19     int i;
20
21     s = s1;
22     s[0] = 0;
23     while (s++[0] != 0)
24         i = 0;
25     write(1, s1, i);
26 }

```

run: /usr/bin/dsymutil "./tests"

Syscall param write(count) contains uninitialised byte(s)
 at 0x1003BD7E6: write (in /usr/lib/system/libsystem_kernel.dylib)
 by 0x100001208: main (main.c:25)

Аналогично, что и выше, только в этой ситуации страдают системные вызовы.

Но иногда **Valgrind** “ошибается” на счет неинициализированной памяти, вот пример:

```

13 #include <stdlib.h>
14 #include <unistd.h>
15 #include <string.h>
16
17 typedef struct
18 {
19     uint8_t ups1;
20     uint32_t ups2;
21 } TMP;
22
23 int main(void)
24 {
25     TMP *test1;
26
27     test1 = malloc(sizeof(TMP));
28     test1->ups1 = 1;
29     test1->ups2 = 0;
30     strlen((void *)test1);
31     return (0);
32 }

```

--49297-- run: /usr/bin/dsymutil "./tests"

==49297== Conditional jump or move depends on uninitialised value(s)
 ==49297== at 0x10009AC99: strlen (vg_replace_strmem.c:477)
 ==49297== by 0x10000F71: main (main.c:30)

Несмотря на то, что мы в структуре проинициализировали все параметры, при обращении к ней с помощью **strlen** **valgrind** детектит ошибку. Связано это с выравниванием данных, в реальности получается, что между **uint8_t ups1** и **uint32_t ups2** находятся 3 байта с неизвестным содержанием. **uint8_t** занимает 1 байт, компилятор выделил для **uint8_t** 3 байта чтобы было кратно 4.

Когда **Valgrind** находит такое, я бы советовал исправлять это, так как потом сложно будет найти среди всех этих сообщений **Valgrind** реально влияющие на ваш код. Это можно сделать, используя переменные одного размера, чтобы компилятор не нашел “лишние” байты для выравнивания. В данном случае достаточно вместо **uint8_t** использовать **uint32_t**.

Valgrind выявляет случаи попыток записи или чтения из невыделенной памяти:

```

13 #include <stdlib.h>
14
15 int main(void)
16 {
17     unsigned char *seg;
18
19     if ((seg = malloc(200)) == 0)
20         return (0);
21     seg[201] = 0;
22     free(seg);
23     return (0);
24 }

```

--87524-- run: /usr/bin/dsymutil "./tests"

==87524== Invalid write of size 1
 ==87524== at 0x100001209: main (main.c:21)
 ==87524== Address 0x100b63bb9 is 1 bytes after a block of size 200 alloc'd
 ==87524== at 0x10009F691: malloc (vg_replace_malloc.c:312)
 ==87524== by 0x1000011EA: main (main.c:19)

Как видно, я пытаюсь записать в не выделенную мной память, о чем **Valgrind** успешно сообщает. Конечно, пока что ты выделяешь небольшие порции памяти из кучи, с большой долей вероятности даже тот факт, что ты пытаешься прочитать или записать за границей выделенной тобой памяти, не приведет к **segmentation**, так там будет находиться память кучи. Но рано или поздно есть шанс выйти за границы кучи, что благополучно приведет к **segmentation**. Да и тот факт, что программа может считать ненужные, лишние данные, — это плохо. К тому же есть способы сделать так, чтобы в таких случаях программа крашилась, некоторые уже в этом убедились на практике и на проверках. Возможно, я опубликую такой способ чуть позже))

У **Valgrind** есть такая графа- «**possibly lost**»(потенциальная утечка). К сожалению, он не способен даже наверняка определить настоящую утечку, что уж говорить про возможные. Просто не нужно обращать на нее внимание.

Также нужно понимать, что **Valgrind** использует сигналы. Так что часть программы, которая на этом работает, невозможно с помощью этой утилиты проверить. Например, в своем shell для синхронизации в случае **jobs** я использовал дополнительный процесс, с которым, само собой, **Valgrind** не будет работать.

Leaks

Данный инструмент представляет сама **Mac OS**, и это плюс. Но, в отличие от **Valgrind**, функционал у него поменьше. Зато он не “ошибается” и позволяет проверять проекты графической ветки.

Leaks способен проверить только выполняемую программу. Если программа слишком быстро завершает свою работу, нужно в том месте, где она это делает, расположить “бесконечный цикл” или функцию **sleep(10000)**, тем самым можно спокойно проверить программу с помощью **Leaks**.

В аргументах в **Leaks** нужно указать имя программы или ее **PID**. Например, **leaks tests**.

Вот пример работы **Leaks**:

```
13 #include<stdlib.h>
14 #include<unistd.h>
15
16 int→ → main(void)
17 {
18     char→ → **spl;
19
20     if·((spl:=·malloc(sizeof(char·*))·)·==·0)
21     → return·(0);
22     if·((spl[0]·:=·malloc(sizeof(char·*))·)·==·0)
23     → return·(0);
24     free(spl);
25     sleep(100000);
26 }
```

```
> leaks tests
Process:      tests [68032]
Path:         /Users/amatilda/Subject/tests/tests
Load Address: 0x1055f7000
Identifier:    tests
Version:      ???
Code Type:    X86-64
Parent Process: zsh [67819]

Date/Time:     2020-02-18 23:20:55.941 +0300
Launch Time:   2020-02-18 23:20:45.598 +0300
OS Version:    Mac OS X 10.12.6 (16G2128)
Report Version: 7
Analysis Tool:  /Applications/Xcode.app/Contents/Developer/usr/bin/leaks
Analysis Tool Version: Xcode 9.2 (9C40b)
-----
Leaks Report Version: 2.0
Process 68032: 154 nodes malloced for 13 KB
Process 68032: 1 leak for 16 total leaked bytes.
Leak: 0x7fa1bf500010 size=16 zone: DefaultMallocZone_0x105603000
0x00000000 0x00000000 0x00000000 0x00000000 .....P.....P
.. .. ..
```

Как видно, **Leaks** нашел, что в коде есть одна **утечка**, но, кроме самого факта ее, более подробной информации не предоставил. Тем самым, мягко говоря, шансов найти **утечку** у нас очень мало. Но если перед запусками проверяемой программы включить протоколирование всех вызовов **malloc**, то найти, где **утечки** не составит большого труда.

Включить можно так: **export MallocStackLoggingNoCompact=1**.

```
leaks Report Version: 2.0
Process 68678: 151 nodes malloced for 12 KB
Process 68678: 1 leak for 16 total leaked bytes.
Provided dSYM: [/Users/amatilda/Subject/tests/./tests.dSYM/Contents/Resources/DWARF/tests] does not match symbol owner 0x7fdd13407690
Leak: 0x7fb752502560 size=16 zone: DefaultMallocZone_0x10d942000
0x00000000 0x70000000 0x00000000 0x70000000 .....P.....P
Call stack: [thread 0x7fff19573c0]: | 0x1 | start | main main.c:22 | malloc
```

Ну вот, теперь уже появилось больше информации, и четко видно, где у нас потерялась память.

Также встроенный инструментарий позволяет забивать мусором выделенную память, чтобы она однозначно не оказалась занулена.

Кратко о том, что нас в первую очередь интересует:

- **export MallocStackLoggingNoCompact=1** или **export MallocStackLogging=1** — логирует, в какой функции была выделена память и тем самым позволяет отследить, где произошла **утечка**.

- **Export MallocGuardEdges=1** — при очень больших выделениях памяти **malloc** окружает выделенную память защитными страницами памяти, выход за границы выделенной памяти приведёт к **segmentation**.
- **export MallocScribble=1** — выделенная память **malloc** инициализируется **0xAA**, т. е. повышает шанс на крах программы при выходе за границы выделенной памяти.
- **export MallocScribble=1** — вся освобожденная память инициализируется **0x55**, т. е. повышает шанс на крах программы при попытке повторно использовать ранее освобожденную память.

Objdump

Данная программа показывает информацию об исполняемом файле. У нее много опций, но, в первую очередь, нам интересна **-lazy-bind**. Данная опция показывает содержимое секции импорта — другими словами, покажет все внешние функции. Вот пример:

```
13 #include <stdlib.h>
14 #include <unistd.h>
15
16 int main(void)
17 {
18     char* spl;
19
20     if ((spl = malloc(sizeof(char*))) == 0)
21         return 0;
22     if ((spl[0] = malloc(sizeof(char*))) == 0)
23         return 0;
24     free(spl);
25     free(spl);
26 }
27
```

```
> objdump -lazy-bind ./tests

./tests:          file format Mach-O 64-bit x86-64

Lazy bind table:
segment section          address      dylib          symbol
__DATA    __la_symbol_ptr  0x100001010 libSystem      _free
__DATA    __la_symbol_ptr  0x100001018 libSystem      _malloc
```

Как видно, все функции, показанные утилитой, мы, действительно, использовали. Но нужно иметь в виду, что для оптимизации компилятор иногда сам использует некоторые функции, так что не нужно удивляться, если там окажется **memcpy**:

```
13 #include <stdlib.h>
14 #include <dirent.h>
15 #include <unistd.h>
16
17
18 int main(void)
19 {
20     DIR* dirp1;
21     DIR* dirp2;
22
23     dirp2->dd_buf = 0;
24     dirp1 = dirp2;
25     return 0;
26 }
```

```
> objdump -lazy-bind ./tests

./tests:          file format Mach-O 64-bit x86-64

Lazy bind table:
segment section          address      dylib          symbol
__DATA    __la_symbol_ptr  0x100001018 libSystem      ____stack_chk_fail
__DATA    __la_symbol_ptr  0x100001020 libSystem      _memcpy
/Users/amatilda/Subject/tests
```

Послесловие

Моя первая статья из цикла на этом закончена. Надеюсь, Вы смогли почерпнуть для себя что-то новое, а я — смог быть полезен и дать ответы на некоторые мучающие Вас вопросы.

В дальнейшем я планирую написать как минимум статью о проектах **minishell**, **21sh** и **42sh**, о том, как их делать и как проверять.

Очень хотелось бы, чтобы низкая грамотность в плане утечек и используемых инструментов для нахождения проблемных мест в программе искоренялась среди студентов, в том числе с помощью таких статей.

Моя благодарность @cjoaquin за вычитку

До следующих встреч Ваш @amatilda

Спасибо за внимание!

02.03.2020