борьба с утечками ресурсов и переполняющимися буферами на языковом и внеязыковом уровне

крис касперски ака мыщъх, no-email

с утечками ресурсов и переполняющимися буферами программистское сообщество борется уже лет тридцать, но все безуспешно. в этой статье мыщъх делится своими собственными трюками и находками. на "серебряную пулю" они не претендуют, но ликвидируют значительное количество ошибок с минимальными накладными расходами

введение

Проблемы утечек ресурсов и переполнения буферов главным образом относятся к Си (и в меньшей степени к его преемнику — Си++). Это не покажется удивительным, если вспомнить, что Си — самый низкоуровневый из всех высокоуровневых языков, всего лишь на одну ступень отстоящий от ассемблера. Си делает только то, о чем его просят, вынуждая программиста самостоятельно выделять и освобождать память, контролировать границы буферов и заботиться о массе других вещей, которые на Java, C# и многих других языках автоматически выполняются компилятором (интерпретатором), однако, ни один из них не завоевал такой популярности как Си и навряд ли завоюет ее в дальнейшем. Почему? На Си написано огромное количество программ, которые необходимо развивать и поддерживать, Си обеспечивает максимальную производительность и переносимость, в то время как его конкуренты предлагают _либо_ производительность, _либо_ переносимость.

Что же касается Си++, то это гибрид, вобравший в себя множество концепций и парадигм программирования, позаимствованных из таких языков как, например, как ADA или Smalltalk, но все равно оставшийся "ручным". В нем нет ни сборщика мусора, ни динамических стековых массивов, ни автоматического контроля границ, ни многих других вещей, которые по-прежнему приходится делать руками. А нет их там потому, что вся эта "автоматизация" заметно снижает производительность и превращает Си++ в пародию на Visual Basic, ярчайшим примером которой является C#. Ходят устойчивые слухи, что значительная часть Longhorn'a была написана на C#, но несмотря на все усилия разработчиков, достичь приемлемой производительности и стабильности им так и не удалось (а что еще можно ожидать от Бейсика, пускай и продвигаемого под видом Си?). В конечном счете, компания была вынуждена похоронить миллионы строк, и начать разработку заново. Если не ошибаюсь, текущая версия Windows Vista базируется на коде Server 2003, написанного на смеси Си с Си++, а это, значит, что отказ от Си/Си++ (по крайней мере в крупных проектов) невозможен и вместо того, чтобы жаловаться на судьбу, лучше придумать пару-тройку новых методов борьбы с утечками и переполняющимися буферам, о чем мы сейчас и поговорим.

переполняющиеся буфера

Ошибок переполнения не избежала практически ни одна программа чуть более сложная, чем "hello, world!", а все потому, что ошибки переполнения в Си носят фундаментальный характер и язык не предоставляет никаких механизмов для их преодоления.

Контроль границ буфера приходится осуществлять вручную. К тому же, если говорить по существу, в Си вообще нет никаких буферов. Получив указатель на "буфер", функция не может определить его длину. Оператор `sizeof` возвращает размер указателя (как правило, равный DWORD), но отнюдь не размер блока, на который он указывает. Небольшую лазейку оставляет нестандартная функция `_msize`, сообщающая размер динамического блока, но она требует указателя на его начало, отказываясь работать с серединой.

Простейшая тестовая программа все это наглядно подтверждает:

```
// Функция, принимающая указатель
// и пытающаяся определить размер соответствующего ему блока
foo(char *p){printf("sizeof says = %Xh\n_msize says = %Xh\n",sizeof(p),_msize(p));}

main()
{
        char buf[0x666]; char *p = malloc(0x999);
```

```
        // передаем указатель на начало блока
        foo(buf);foo(p);

        // передаем указатель на середину блока
        foo(&buf[6]);foo(p+9);
}
```

Листинг 1 программа, демонстрируя невозможность определения размера блока по указателю

После запуска мы получим следующие весьма неутешительные данные:

```
// указатель на начало стекового буфера
sizeof says = 4h, _msize says = 12E8CEh      // sizeof и _msize провалились

// указатель на начало динамического буфера
sizeof says = 4h, _msize says = 9A0h         // sizeof провалилась, _msize - почти OK

// указатель на середину стекового буфера
sizeof says = 4h, _msize says = FFFFFFFFh    // sizeof и _msize провалились

// указатель на середину динамического буфера
sizeof says = 4h, _msize says = D200h        // sizeof и _msize провалились
```

Листинг 2 результат работы программы, определяющий размер блока по указателю

Мы видим, что `_msize` ведет себя очень странно и когда не может определить размер блока, возвращает какой-то мусор, никак не сигнализируя об ошибке. Поэтому, выполнять контроль должна вызывающая функция, передавая вызываемой размер буфера как аргумент. Отсюда и появились: `char *fgets(char *string, int n, FILE *stream);` `char *strncpy(char *strDest, const char *strSource, size_t count)` и другие подобные функции. Теоретически, они на 100% застрахованы от переполнения, но вот практически… значение `n` приходится рассчитывать вручную, а, значит, существует риск ошибиться! К тому же, если длина строки превышает `n`, в буфер копируется лишь "огрызок", что само по себе является нехилым источников проблем и вторичных ошибок. Приходится навешивать специальный обработчик, выделяющий дополнительную память и считывающий оставшийся "хвост", что значительно усложняет реализацию, делает код более громоздким и менее наглядным. Обычно стараются выбрать `n` так, чтобы его значение превышало размер _наибольшей_ строки, выделяя память с запасом и не обращая внимания на то, что большинство строк использует лишь малую часть буфера…

Нет! Память лучше всего выделять динамически, по мере возникновения в ней потребности! В идеале, строка должна представляет собой _список_ (желательно двухсвязный), что не только ускорит операции удаления и вставки подстрок, но и ликвидирует проблему переполнения. О контроле границ заботиться уже необязательно, поскольку память под новые символы выделяется автоматически.

В простейшем случае, каждый элемент списка выглядит приблизительно так:

```
struct slist
{
        unsigned char c;
        struct slist *prev;
        struct slist *next;
        struct slist *first;
        struct slist *last;
};
```

Листинг 3 строка, реализованная в виде списка (простейшая реализация)

Это просто реализуется, но имеет дикий оверхид, требующий для хранения каждого символа 17 байт, поэтому на практике приходится использовать комбинированный способ, сочетающий в себе строковые буфера со списками:

```
#define STR_SIZE      256
struct slist
{
        unsigned int len;
        unsigned char buf[STR_SIZE];
        struct slist *prev;
        struct slist *next;
        struct slist *first;
```

```
        struct slist *last;
};
```

Листинг 4 строка, реализованная в виде списка (продвинутая реализация)

Размер буфера может быть как фиксированным, так и динамическим. Хорошей стратег выделяет под первый элемент списка ~64 байт, под второй ~128 байт и так далее вплоть до 1000h, что позволяет обрабатывать как длинные, так и кроткие сроки с минимальным оверхидом.

Списки на 100% защищены от ошибок переполнения (исключение составляют попытки обработать строку свыше 2 Гбайт, вызывающую исчерпание свободной памяти, но, во-первых, исчерпание это все-таки не переполнение и заслать shell-код злоумышленник не сможет, а во-вторых, это _явная_ ошибка, которую легко обработать, установив предельный лимит на максимально разумную длину строки).

Хуже другое. Реализовав свою библиотеку для работы со "списочными строками", мы будем вынуждены переписать _все_ остальные библиотеки, создавая обертки для каждой строковой функции, включая fopen, CreateProcess и т. д., поскольку все они ожидают увидеть непрерывный массив байт, а вовсе не список! Это чрезвычайно утомительная работа, но зато когда она будет закончена, о переполнения можно забыть раз и навсегда. Правда, производительность (за счет постоянного преобразования типов) падает и весьма значительно…

А вот более быстрое, но менее надежное решение. Отказываемся от стековых буферов, переходя на динамическую память. Выделяем каждому блоку на одну страницу больше, чем нужно и присваиваем последней странице атрибут PAGE_NOACCESS, чтобы каждое обращение к ней вызывало исключение, отлавливаемое нашим обработчиком, который в зависимости от ситуации либо увеличивал размер буфера, либо завершал работу программы с сообщением об ошибке. На коротких строках оверхид весьма значителен, но на длинных он минимален, однако, такая защита страхует лишь от последовательного переполнения, но бессильна предотвратить индексное (подробнее о видах переполнения можно прочитать в моей книжке "shellcoders's programming uncovered", которую можно найти в Осле), к тому же переход на динамические массивы порождает проблему утечек памяти и получается так, что одно лечим, а другое калечим.

Тем не менее, лишний раз подстраховаться никогда не помешает! Чтобы защититься от переполнения кучи (которое в последнее время приобретает все большую популярность) после вызова любой функции, работающей с динамической памятью, необходимо защищать служебные данные кучи атрибутом PAGE_NOACCESS, а перед вызовом функции — снимать их. Для этого нам, опять-таки потребуется написать обертки вокруг всех функций менеджера памяти, что требует времени. К тому же, в реализации кучи от Microsoft Visual C++, служебные данные лежат по смещению -10h от начала выделенного блока, а защищать мы можем только всю страницу целиком, поэтому, во-первых, необходимо увеличить размер каждого блока до 512 Кбайт, чтобы начальный адрес совпадал с началом страницы, а во-вторых, использовать блок только со второй страницы. В результате, при работе с мелкими блоками мы получаем чудовищный оверхид, но зато компенсируемый надежной защитой от переполнения. Так что данный метод, при всех его недостатках, все-таки имеет право на жизнь.

утечки ресурсов

Утечка ресурсов возникает всякий раз, когда функция выделяет блок памяти, открывает файл, но при выходе забывает его освободить/закрыть. Чаще всего это происходит при преждевременном выходе из функции.

Рассмотрим следующий пример:

```
foo()
{
        FILE *ff
        char *p1, *p2;
        p1 = malloc(XXL);
        ff = fopen(FN,"r");
        …
        if (bar()==ERROR) return -1;
        …
        p2 = malloc(XXL);
        …
        free(p1);
        free(p2);
        fclose(ff);
```

```
        return 0;
}
```

Листинг 5 фрагмент типичной программы, страдающей утечками ресурсов

Функция `foo` намеревается выделить два блока памяти `p1` и `p2`, но реально успевает выделить лишь один из них, после чего `bar` завершается с ошибкой, делающей дальнейшее выполнение `foo` невозможным, вот программист и совершает возврат по `return`, забывая о выделенном блоке памяти.

Проблема в том, что в произвольной точке программы очень непросто сказать: какие ресурсы уже выделены, а какие еще нет и что именно нужно освобождать! Ну ведь не поддерживать же ради этого транзакции?! Разумеется, нет. Проблема имеет весьма простое и элегантное решение, основанное на том, что Стандарт допускает освобождение нулевого указателя. Правда, к файлам и другим объектам это уже не относится, но проверку на нуль легко выполнить и вручную.

Правильно спроектированный код должен выглядеть приблизительно так:

```
foo()
{
        int error = 0;
        FILE *ff = 0;
        char *p1 = 0; char *p2 = 0;
        {
                p1 = malloc(XXL);
                ff = fopen(FN,"r");
                …
                if ((bar()==ERROR) && (error=-1)) break;
                …
                p2 = malloc(XXL);
                …
        } while(0);
        free(p1); free(p2);
        if (ff) fclose(ff);
        return error;
}
```

Листинг 6 реконструированный вариант программы, свободный от утечек

Что изменилось? Абсолютно все! Теперь для внепланового выхода из программы (который осуществляется по `break`), нам уже не нужно помнить, что мы успели выделить или открыть! По завершении цикла while (который на самом деле никакой не цикл, а просто имитация критикуемого оператора `goto`), мы освобождаем (или точнее, пытаемся освободить) _все_ ресурсы, которые потенциально могли быть выделены. Структура программы значительно упрощается и главное тут — не забыть освободить все, что мы выделили, но программисты об этом все равно забывают.

Решение заключается в создании своей собственной обертки вокруг функции `malloc` (условно назовем ее `my_malloc`), которая выделят память, запоминает указатель в своем списке/массиве и перед возвращением в вызываемую функцию подменяет адрес возврата из материнский функции на свой собственный обработчик (конечно, без ассемблерных вставок тут не обойтись, но они того стоят). Как следствие — при выходе из `foo`, управление получает обработчик my_malloc, читающий список/массив и автоматически освобождающий все, что там есть, снимая тем самым эту ношу с плеч программиста.

Если же выделенная функцией память по замыслу разработчика не должна освобождаться после ее завершения, на этот случай можно предусмотреть специальный флаг, передаваемый `my_malloc`, и сообщающий, что этот блок освобождать не надо и программист освободит его сам.

Одним их самых мерзких недостатков языка Си является отсутствие поддержки динамических стековых массивов. Стековая память хороша тем, что автоматически освобождается при выходе из функции, однако, выделение стековых массивов "на лету" невозможно и мы должны заранее объявить их размеры при объявлении переменных. В C99 сделаны небольшие подвижки в этом направлении и теперь мы можем объявлять массивы, размер которых задается аргументом, передаваемым функции, однако, это не решает всех проблем и к тому же C99 поддерживают далеко не все компиляторы.

В частности, компилятор GCC 2.95 нормально "переваривает" следующий код, а Microsoft Visual C++ увы, нет:

```
f(int n)
{
        char buf[n];
        return sizeof(buf);
}
```

Листинг 7 стековые массивы с переменным размером, появившиеся в Стандарте C99

На самом деле, выделять динамические массивы все-таки возможно, однако, только в том случае если компилятор, во-первых, адресует локальные переменные через EBP, а во-вторых, в эпилоге использует конструкцию `MOV ESP, EBP` вместо ADD ESP, n. К таким компиляторам, в частности, относится Microsoft Visual C++, автоматически переходящий на адресацию локальных переменных через регистр EBP, если в теле функции присутствует хотя бы одна ассемблерная вставка.

Фрагмент одной из таких функций приведен ниже:

```
text:00000010          push    ebp
text:00000011          mov     ebp, esp
text:00000013          push    esi
…
text:0000002B          sub     esp, 400h
…
text:00000034          mov     eax, [ebp+var_4]
…
text:00000048          pop     esi
text:00000049          mov     esp, ebp
text:0000004B          pop     ebp
text:0000004C          retn
```

Листинг 8 дизассемблерный фрагмент программы, откомпилированной компилятором Microsoft Visual C++ с максимальной оптимизацией (ключ /Ox)

Выделение памяти на стеке осуществляется путем "приподнимания" регистра-указателя стека на некоторую величину, что можно сделать командой "`SUB ESP, n`", где n – количество выделяемых байт. Поскольку, компилятор адресует локальные переменные через регистр EBP, то изменение ESP не нарушит работы функции и все будет ОК, но… так будет продолжаться недолго. При выходе из функции она попытается восстановить регистры, сохраненные на входе (в данном случае — это регистр ESI), но на вершине перемещенного стека их не окажется! В регистр ESI попадет мусор и материнская функция рухнет.

Существует по меньшей мере два решения проблемы: либо вручную опускаем регистр ESP при выходе из функции (если, конечно, не забудем это сделать), либо копируем на вершину выделенного блока порядка 20h байт памяти с макушки старого стека (обычного этого более чем достаточно, даже если функция сохраняет все регистры общего назначения ей требуется всего лишь 1Ch байт). В этом случае о ручном освобождении выделенной памяти можно не заботиться. Это сделает машинная команда `MOV ESP, EBP`, помещенная компилятором в эпилог функции.

Ниже приведена пара макросов для динамического выделения стековой памяти по первому сценарию:

```
#define stack_alloc(p,n,total) { __asm{sub esp,n};\
                                 __asm{mov dword ptr ds:[p],esp};\
                                 total += n;}

#define stack_free(total) {__asm{add esp,total};}
```

Листинг 9 макросы для динамического выделения/освобождения стековой памяти (только для Microsoft Visual C++)

А вот пример использования макросов `stack_alloc` и `stack_free`:

```
foo()
{
        char* p; int n;
        int total = 0;

        n = 0x100;
        stack_alloc(p, n, total);

        strcpy(p,"hello, world!\n");printf(p);
```

```
        stack_free(total);
}
```

Листинг 10 исходный текст программы, использующий динамические стековые массивы

Естественно, о вызове `stack_free` программист может забыть (и ведь наверняка забудет!), поэтому лучше выделять память так, чтобы при выходе из функции она освобождалась автоматически.

Ниже приведен исходный текст макроса `auto_alloc`, который именно так и работает:

```
#define auto_alloc(p,n) {__asm{add n,20h};\
                                    __asm{mov eax,esp};\
                                    __asm{sub esp,n};\
                                    __asm{mov p,esp};\
                                    __asm{push 20h};\
                                    __asm{push eax};\
                                    __asm{mov eax,p};\
                                    __asm{push eax};\
                                    __asm{call memcpy};\
                                    __asm{add esp,0Ch};\
                                    __asm{add p,20h};}
```

Листинг 11 исходный код макроса, выделяющего стековую память, автоматически освобождаемую при выходе из функции (только для Microsoft Visual C++)

Как его можно использовать на практике? А хотя бы вот так!

```
foo()
{
        char* p; int n; n = 0x100;
        auto_alloc(p, n);

        strcpy(p,"hello, world!\n");printf(p);
}
```

Листинг 12 демонстрационный пример программы, использующей макрос auto_alloc

При работе со стековой памятью следует помнить три обстоятельства: во-первых, по умолчанию каждый поток получает всего лишь 1 Мбайт стековой памяти, что по современным понятиям очень мало. Стековую память первичного потока легко увеличить, передав линкеру ключ "`/STACK:reserve[,commit]`", где `reserve` – зарезервированная, а `commit` – выделенная память. Размер стековой памяти остальных потоков определяется значением аргумента `dwStackSize` функции `CreateThread`.

Во-вторых, при старте потока Windows выделяет ему минимум страниц стековой памяти, размещая за ними специальную "сторожевую" страницу (PAGE GUARD) при обращении к которой возбуждается исключение, отлавливаемое системой, которая выделяет потоку еще несколько страниц, перемещая PAGE GUARD наверх. Если же мы попытаемся обратиться к памяти, лежащей _за_ PAGE GUARD — произойдет крах. Поэтому, при ручном выделении стековой памяти, необходимо последовательно обратиться хотя бы к одной ячейке каждой страницы: `#define stack_out(p,n) for(a=0;a<n;a+=0x100)t=p[a];`

В-третьих, размер выделяемых блоков памяти должен быть кратен четырем, иначе многие API и библиотечные функции откажут в работе.

Но что делать, если, несмотря на все усилия, память продолжает утекать? На этот случай у мыщъх'а припасено несколько грязных, но довольно эффективных трюков. Вот один из них: когда количество потребляемой приложением памяти достигает некоторой, заранее заданной отметки, мы "прогуливаемся по куче" API-функцией HeapWalk, сохраняя все выделенные страницы в специальный файл (устроенный по принципу файла подкачки) и возвращаем память системе, оставляя страницы зарезервированными и назначая им атрибут PAGE_NOACCESS. После чего нам остается только отлавливать исключения и подгружать содержимое реально используемых страниц, восстанавливая оригинальные атрибуты доступа (PAGE_READ или PARE_READWRITE). В результате, утекать будет только адресное пространство, которое, между прочим не бесконечно, и при интенсивной течи довольно быстро кончается. И что же тогда? Можно, конечно, просто завершить программу, но лучше рискнуть и попробовать освободить блоки к которым дольше всего не было обращений. Разумеется, мы не можем гарантировать, что именно они ответственны за утечку памяти. Быть может, программа в

самом начале выделила несколько блоков, планируя обратиться к ним при завершении процесса, но… риск благородное дело!

заключение

Помимо рассмотренных нами существуют и другие методы борьбы с утечками и переполняющимися буферами. Для мира BSD/LINUX характерны run-time верификаторы, встраиваемые непосредственно в сам компилятор (ведь его исходные тексты доступны). Под Windows более популярны статические анализаторы — потомки древнего LINT. Но всем им свойственны недостатки, поэтому, настоящие программисты никогда не останавливаются на достигнутом, а неуклонно движутся вперед, выдумывая все новые приемы и трюки. Одни — сметаются временем, другие — получают широкое распространение, попадая в учебники и справочные руководства.

Но лучшее руководство это свой свойственный опыт, который и описал мыщъх, не претендуя, впрочем, на новизну и новаторство.