сишные трюки
(15h выпуск)

крис касперски ака мыщъх, a.k.a. souriz, a.k.a. nezumi, no-email

сегодня мы займемся укрощением gets и подобным ей функциям, возвращающим заранее непредсказуемый объем данных — быть может десяток байт, а быть целую сотню мегабайт. ограничивать предельный объем (как это часто делается) негуманно, а в некоторых случаях — невозможно или же требует серьезного редизайна всего кода, но если немножко схитрить, то ни ограничивать, ни редизайнить не потребуется, все будет работать и так!

метод 1 — malloc(maxinux maximore)

ОК, возьмем функцию gets (название, естественно, условное, и на ее месте может окажется любая функция, возвращающая заранее непредсказуемый объем данных) и скалькулируем сколько памяти она может затребовать в _худшем_ случае. Для определенности остановимся на отметке в 100 Мбайт. Выделяем нужное количество памяти через malloc, а после возвращения из gets определяем актуальный размер данных и тут же реаллоцируем блок памяти, усекая его вызовом realloc до нужного размера.

Просто как дважды два, но, увы, ресурсоемко и не совсем безопасно (точнее, _совсем_ небезопасно). Хотя Windows выделяет физическую память лишь при реальном обращении к страницам, Си-функция malloc (вызывающая API-функцию VirtualAlloc с атрибутом MEM_COMMIT), увеличивает "Working Set" процесса и, если виртуальной памяти не хватает, происходит неизбежный рост файла подкачки (даже при наличии свободной физической памяти!), что снижает производительность, не говоря уже о том, что на системах с квотированием такие программы просто не выживают.

К тому же, в случае с gets, выделение 100 Мбайт памяти проблемы не решает и риск переполнения буфера не исчезает, а всего лишь уменьшается. Чтобы программа не пошла вразнос и не стала жертвой атаки, последней странице буфера рекомендуется присвоить атрибут PAGE_NOACCESS вызовом API-функции VirtualProtect (а сам блок памяти выделять не через malloc, а через VirtualAlloc).

Тогда, при достижении конца буфера возникнет исключение, которые мы сможем перехватить, установив SEH-обработчик на EXCEPTION_ACCESS_VIOLATION и тем или иным образом обработать ситуацию.

Сделать это можно следующим образом (обработка ошибок для упрощения понимания сведена к минимуму).

```
#define XXL            (100*1024*1024)
#define PAGE_SIZE      0x1000

#define Is2power(x)            (!(x & (x-1)))
#define ALIGN_DOWN(x, align)   (x & ~(align-1))
#define ALIGN_UP(x, align)     ((x & (align-1))?ALIGN_DOWN(x,align)+align:x)
main()
{
        DWORD old; char *p; int real_size=0;
        p=VirtualAlloc(0,XXL,MEM_COMMIT,PAGE_READWRITE);
        VirtualProtect((p-PAGE_SIZE), PAGE_SIZE, PAGE_NOACCESS, &old);
        __try{
                gets(p);
        }

        __except( GetExceptionCode() == EXCEPTION_ACCESS_VIOLATION) {
                printf("too much!\n"); real_size=-1;
        }

        if (real_size==-1)
                VirtualFree(p,XXL,MEM_DECOMMIT);
        else
                VirtualFree(p+ALIGN_UP((strlen(p)+1),PAGE_SIZE),
                        XXL-ALIGN_UP((strlen(p)+1),PAGE_SIZE), MEM_DECOMMIT);
}
```

Листинг 1 простой способ укрощения gets

Не такой уж и сложный код, во всяком случае он намного проще, чем блочное чтение с помощью fgets и других функций, работающих с буферами памяти произвольного размера. Однако, следует помнить, что если запись в буфер происходит не последовательно (как в случае с gets), а "скачками", то защита последней страницы нам ничем не поможет, т. к. вызываемая функция запросто может "перепрыгнуть" ее. В принципе, существует возможность прижать конец буфера к вершине нижней половины адресного пространства — в верхней находится код операционной системы, который себя в обиду не даст , однако, гарантий, что эта память уже не занята у нас нет, увы!

метод 2 — VirtualAlloc(,,MEM_RESERVE,)

Главным недостатком предыдущего способа была и остается его ресурсоемкость. Совершенно нецелесообразно "отбирать" у системы XXL байт памяти не будучи при этом уверенным, что из них потребуется хотя бы половина.

Поступим умнее. Заменив флаг MEM_COMMIT на MEM_RESERVE, мы заставим функцию VirtualAlloc не выделять, а всего лишь _резервировать_ память без неизбежного роста Working Set'a и размера файла подкачки. Резервирование памяти осуществляется практически мгновенно. А вот при всяком доступе к зарезервированной странице возникает исключение типа EXCEPTION_ACCESS_VIOLATION и нам остается всего лишь написать свой собственный SEH-фильтр, вызывающий VirtualAlloc с атрибутом MEM_COMMIT для выделения запрошенной страницы.

То есть память в натуре выделяется динамически по мере ее потребления и потому, не жадничая особо, мы можем увеличить XXL хоть на порядок. Главное — чтобы адресного пространства хватило! А в распоряжении приложения, работающего под управлением 32-битных версий Windows, _как_ _правило_ имеется по меньшей мере ~1 Гбайт.

Как ни парадоксально, но динамическое выделение памяти даже упрощает код:

```
souriz(struct _EXCEPTION_POINTERS *exception_pointers)
{
        DWORD old;
        if (exception_pointers->ExceptionRecord->ExceptionCode==
                                                EXCEPTION_ACCESS_VIOLATION)
        {
                VirtualAlloc((char*)(exception_pointers->
                        ExceptionRecord->ExceptionInformation[1]),
                                PAGE_SIZE, MEM_COMMIT, PAGE_READWRITE);

                return EXCEPTION_CONTINUE_EXECUTION;
        }
        return EXCEPTION_CONTINUE_SEARCH;
}
main()
{
        char *p=VirtualAlloc(0,XXL,MEM_RESERVE,PAGE_READWRITE);

        __try{
                gets(p);
        }

        __except ( souriz( GetExceptionInformation())) {}

        VirtualFree(p+ALIGN_UP((strlen(p)+1),PAGE_SIZE),
                XXL-ALIGN_UP((strlen(p)+1),PAGE_SIZE), MEM_DECOMMIT);
}
```

Листинг 2 динамический способ укрощения gets

А вот производительность по сравнению с листингом 1 не только не поднимется, но даже упадет. Конкретно так упадет, ведь обработка исключений — операция не из дешевых, а постраничная стратегия выделения памяти — кретинизм еще тот. ОК, меняем стратегию — изначально выделяем в буфере несколько страниц памяти, а затем (при первом вызове обработчика исключения) выделяем одну страницу, при втором — две, при следующем — четыре и… так вплоть до ~16 страниц, время обработки которых вызываемой функцией заметно превышает накладные расходы (оверхид) на отлов исключений, хотя точная цифра завит как от мощности ЦП так и от специфики поставленной задачи. На слабых машинах (типа P-III) мыщъх

рекомендует выделять по 64 страницы за раз, однако, в условиях дефицита памяти можно сойтись и на 32х.

метод 3 — доверяемся автоматике

Отслеживать исключения — довольно нудное и утомительное дело. А нет ли в Windows-системах готового механизма, поддерживающего динамическое выделение памяти, который бы все делал за нас?!

Такой механизм есть и имя ему — стек! При создании нового потока система не выделяет ему память, а лишь резервирует ее. Точнее стеку выделяется всего одна страница за которой (на самой вершине стека) находится злой пес Цербер — страница памяти с атрибутом PAGE_GUARD известная под именем "сторожевой". При обращении к ней процессор генерирует исключение, перехватываемое системой, которая выделяет запрошенную страницу в пользование потока, перемещая пса Цербера на еще одну страницу назад (в область младших адресов, куда растет стек).

Возникает следующая идея. Создаем пустой поток со стеком размера XXL. Указатель на стек передаем основному потоку с функцией типа gets, которая начинает планомерно "отъедать" память. После ее завершения остается только определить реальный размер возвращенных данных и вызывать функцию VirtualAlloc, чтобы выделить обозначенные страницы _еще_ _один_ _раз_ (первый раз их выделила система, второй — мы). Менеджер кучи увеличивает специальный счетчик и теперь при завершении потока освобождаются все страницы, за исключением страниц, выделенных нами, и их может использовать любой другой поток данного процесса!!!

Это становится возможным благодаря одному очевидному, но малоизвестному обстоятельству, а именно — на низком уровне стек и куча управляются _один_ и _тем_ _же_ менеджером памяти! То есть, мы используем стек потока как своеобразный динамический массив, а тело потока пустует. Теперь становится понятно, почему gets следует размещать именно в основном, а не во вспомогательном потоке — после того, как gets вернет свои данные, все остальное стековое пространство вспомогательного потока автоматически освобождается путем его завершения по return или TerminateThread. А вот если бы gets была расположена во вспомогательном потоке, то с завершением возникли бы проблемы.

Впрочем, проблемы возникнут и так. Поскольку, стек растет вверх, то с функцией gets он не станет работать однозначно (она заполняет буфер сверху вниз, т.е. в обратном порядке). Однако, если у нас есть возможность переписать код gets (или другой функции подобной ей), этот трюк может сработать. "Может" потому, что: а) система выделяет стековое пространство постранично, вследствие чего мы имеем большой оверхид и тормоза; б) система рассчитывает, что стек заполняется последовательно и не прощает прыжки через сторожевую страницу, генерируя при этом исключение, которое, конечно, нетрудно обработать и самостоятельно, но тогда исчезает все очарование простоты кода, за которое мы боролись.

Таким образом, второй метод самый оптимальный. Он не сложен в реализации, не отъедает лишнюю память, достаточно быстро работает и надежно страхует от переполняющихся буферов. Дело ведь не в самой gets, которая выбрана всего лишь в качестве наглядного примера. Хорошо, пускай нам необходимо читать данные со стандартного потока ввода и мы заранее не можем сказать сколько их будет — десяток байт, мегабайт или целый гигабайт. Конечно, можно читать блоками по несколько десятков килобайт, объединяя блоки в списки или занимаясь их реаллокацией, но… все это либо слишком сложно в реализации, либо непродуктивно. Так что исключения рулят (причем совершенно без руля).