сишные трюки
(14h выпуск)

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

о переполняющихся буферах написано много, о переполнении целочисленных/вещественных переменных — чуть меньше, а ведь это одна из фундаментальных проблем языка си, доставляющая программистам массу неприятностей и порождающая целых ворох уязвимостей разной степени тяжести, особенно если программа пишется сразу для нескольких платформ. как быть, что делать? мыщъх делится своим личным боевым опыт (с учетом всех травм и ранений, понесенных в ходе сражений), надеясь, что читатели найдут его полезным, а я тем временем в госпитале с сестричкой…

трюк 1 — закон суров, но он закон!

Фундаментальность проблемы переполнения целочисленных переменных имеет двойственную природу. Стандарт декларирует, что результат выражения (a + b) в общем случае неопределен (undefined) и зависит как от архитектурных особенностей процессора, так и от "характера" компилятора. Положение усугубляется тем, что Си (в отличии от Паскаля, например) вообще ничего не говорит о разрядности типов данных больших чем байт. long int вполне может равняться int. И хотя начиная с ANSI C99 появились типы int32_t, int64_t, а некоторые компиляторы (в частности, MS VC) еще черт знает с какой версии поддерживают нестандартные типы _int32 и _int64, проблема определения разрядности переменных остается проблемой. Одним процессорам выгоднее обрабатывать 64-битные данные, другим — 32-битные и потому выбирать тип "на вырост", то есть с расчетом, что в него гарантированно влезут обозначенные значения — расточительно и не гуманно.

К тому же, _гарантии_ что переполнение не произойдет, у нас нет. Обычно при переполнении либо наблюдается изменение знака числа (небольшое знаковое отрицательное превращается в больше беззнаковое), либо "заворот" по модулю, физическим аналогом которого могут служить обычные механические часы. Хинт: 3 + 11 = 2, а вовсе не 14! Вот так неожиданность! И ищи потом на каком этапе вычислений данные превращаются в винегрет! А искать можно долго и ошибки возникают даже в полностью отлаженных программах, стоит только скормить им непредвиденную последовательность входных данных!

LIA-1 (см. приложение "H" к Стандарту ANSI C99) говорит, что в случае отсутствия "заворота" при переполнении знаковых целочисленных переменных, компилятор должен генерировать сигнал (ну, или, в терминах Microsoft, выбрасывать исключение). Поскольку, знаковый бит на x86 процессорах расположен по старшему адресу, заворота не происходит и некоторые компиляторы учитывают это обстоятельство при генерации кода. В частности, GCC поддерживает специальный флаг "-ftrapv". Посмотрим, как он работает?

```
foo(int a, int b)
{
        return a+b;
}
```

Листинг 1 исходная функция, складывающая два знаковых числа типа int

```
foo     proc near
        push    ebp                     ; открываем кадр
        mov     ebp, esp                ;              стека
        mov     eax, [ebp+arg_4]        ; грузим аргумент b в EAX
        add     eax, [ebp+arg_0]        ; EAX := (a + b)
        pop     ebp                     ; закрываем кадр стека
        retn                            ; возвращаем сумму (a+b) в EAX
foo     endp
```

Листинг 2 компиляция компилятором GCC с ключами по умолчанию

Очевидно, что результат работы данной функции непредсказуем, и, если сумма двух int'ов не влезет в отведенную разрядность, нам вернется черт знает что. А вот теперь используем флаг -ftrapv:

```
foo     proc near

        push    ebp                     ; открываем
        mov     ebp, esp                ;              кадр
        sub     esp, 18h                ;                   стека
        mov     eax, [ebp+arg_4]        ; грузим аргумент b в EAX
        mov     [esp+18h+var_14], eax   ; передаем аргумент b функции __addvsi3
        mov     eax, [ebp+arg_0]        ; грузим аргумент a в EAX
        mov      [esp+18h+var_18], eax  ; передаем аргумент a Функции __addvsi3
        call    __addvsi3               ; __addvsi3(a, b); // безопасное сложение
        leave                           ; закрываем кадр стека
        retn                            ; возвращаем сумму (a+b) в EAX
foo     endp
…
__addvsi3       proc near
        push    ebp                     ; открываем
        mov     ebp, esp                ;              кадр
        sub     esp, 8                  ;                   стека
        mov     [ebp+var_4], ebx        ; сохраняем EBX в лок. переменной
        mov     eax, [ebp+arg_4]        ; грузим аргумент b в EAX
        call    __i686_get_pc_thunk_bx ;грузим thunk в EBX
        add     ebx, 122Fh              ; -> GLOBAL_OFFSET_TABLE
        mov     ecx, [ebp+arg_0]        ; грузим аргумент a в ECX
        test    eax, eax                ; определяем знак аргумента b
        lea     edx, [eax+ecx]          ; EDX := a + b
        js      short loc_8048410       ; прыгаем если знак
; ———————————————————————————————————————————————————————————————————————
                                        ; работаем с беззнаковыми переменными
        cmp     edx, ecx                ; if ((a + b) >= a)
        jge     short loc_8048400       ;                   goto OK

loc_80483F5:                            ; если ((a + b) < a)…
        call    _abort                  ;       то имело место переполнение
        lea     esi, [esi+0]            ;              и мы абортаемся

loc_8048400:                            ; нормальное продолжение программы
        mov     ebx, [ebp+var_4]        ; восстанавливаем EBX
        mov     eax, edx                ; перегоняем в EAX (a+b)
        mov     esp, ebp                ; закрываем
        pop     ebp                     ;       кадр стека
        retn                            ; возвращаем (a+b) в EAX
; ———————————————————————————————————————————————————————————————————————
loc_8048410:                            ; работаем со знаковыми
        cmp     edx, ecx                ; if ((a+b) < a)
        jg      short loc_80483F5       ;              GOTO _abort
        jmp     short loc_8048400       ; -> нормальное продолжение
__addvsi3       endp
```

Листинг 3 компиляция компилятором GCC с ключом -ftrapv

Сложение с флагом -ftrapv безопасно, но… как же оно тормозит!!! Кстати, на уровне оптимизации -O2 и выше, флаг -ftrapv игнорируется. Но даже без всякой оптимизации он не ловит переполнения при умножении и что самое печальное, поддерживается не всеми компиляторами.

трюк 2 — пишем закон сами!

На самом деле, для "безопасного" сложения чисел у нас есть все необходимые ингредиенты. Причем, это будет работать с любым компилятором на любом уровне оптимизации и с достаточно приличной скоростью (уж во всяком случае побыстрее, чем __addvsi3 в реализации от GGC).

Функция безопасного сложения двух переменных типа int в простейшем случае выглядит так:

```
#include <limits.h>           // здесь содержатся лимиты всех типов
int safe_add(int a, int b)
{
        if(INT_MAX - b < a)    return _abort(ERROR_CODE);
        return a + b;
}
```

Листинг 4 функция безопасного сложения

Дизассемблерный листинг не приводится за ненадобностью. Если компилятор заинлайнит safe_add, то мы имеем следующий оверхид: одно лишнее ветвление, одно лишнее сравнение и одно лишнее вычитание. Конечно, в особо критичных фрагментах (да еще и в глубоко вложенных циклах) этот оверхид непременно даст о себе знать, и тогда лучше отказаться от safe_add и пойти другим путем. Например, обосновать, что переполнения (в данном месте) не может произойти в принципе даже при обычном сложении.

трюк 3 — отправляемся в плаванье

Вещественные переменные, в отличии от целочисленных, работают чуть медленнее, хотя… это еще как сказать! С учетом того, что ALU и FPU блоки современных ЦП работают параллельно, то для достижения наивысшей производительности, целочисленные и вещественные переменные должны использоваться совместно (конкретная пропорция определяется типом и архитектурой процессора).

Главное, что x86 (и некоторые другие ЦП) поддерживают генерацию исключений при переполнении вещественных переменных, хотя по умолчанию она выключена и включить ее, увы, средствами "чистого" языка Си нельзя, но вот если прибегнуть к функциями API или нестандартным расширениям….

Рассмотрим следующую программу:

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

main()
{
        // объявляем вещественную переменную
        // (это может быть так же и float)
        double f = 666;

        // считываем значение управляющего слова
        // сопроцессора через MS-specific функцию
        int cw = _controlfp(0, 0);

        // задействуем исключения для след. ситуаций
        cw &=~(EM_OVERFLOW|EM_UNDERFLOW|EM_INEXACT|EM_ZERODIVIDE|EM_DENORMAL);

        // обновляем содержимое управляющего слова сопроцессора
        _controlfp( cw, MCW_EM );

        __try{                          // в блоке try мы будем делать исключения

                while(1)                // в бесконечном цикле вычисляем f = f * f
                {                       // выводя его содержимое на экран
                        printf("%f\n", f = f * f);
                }
        }

        except(puts("in filter"), 1)    // а тут мы ловим возникающие исключения!
        {
                puts("in except");      // для упрощения обработка исключений опущена
        }
}
```

Листинг 5 активация исключений при работе с вещественными переменными

В зависимости от компилятора (и процессора) данный пример будет тормозить в большей или меньшей степени. В частности, на x86 вещественное деление _намного_ быстрее целочисленного. С другой стороны, компилятор MS VC выполняет вещественное сложение в разы медленнее, главным образом потому, что не умеет сохранять промежуточный результат вычислений в регистрах сопроцессора и постоянно загружает/выгружает их в переменные, находящиеся в памяти. GCC такой ерундой не страдает и при переходе с целочисленных переменных на вещественные быстродействие не только не падает, но местами даже и возрастает.

Плюс, вещественные переменные имеют замечательное значение "не число", которое очень удобно использовать в качестве индикатора ошибки. У целочисленных с этим — настоящая проблема. Одни функции возвращают ноль, другие минус один, в результате чего возникает путаница, а если и ноль, и минус один входят в диапазон допустимых значений,

возвращаемых функцией, приходится не по детски извращаться, возвращая код ошибки в аргументе, переданном по указателю или же через исключения.

А с вещественными переменными все просто! И удобно! И это удобство стоит небольшой платы за производительность!