сишные трюки от мыщъх'а
(5й выпуск)

строки в hex-числах

Допустим, нам потребовалось прочитать значение ячейки памяти некоторого процесса и вывести ее на экран. Или распечатать дескриптор заданного окна. Да все что угодно! Суть в том, что в программах, написанных "под себя" это обычно делается так:

```
printf("hWnd: %Xh\n", FindWindow(0, "Калькулятор"));
```

Листинг 1 неправильный вывод на экран

Если же искомое окно отсутствует, функция `FindWindows()` возвратит ошибку и на экране появится "`hWnd: 0h`". Нормальные хакеры знают, что такого дескриптора в природе не существует и это символ ошибки, но… все равно получается как-то неаккуратно и "некультурно". Лучше, чтобы программа сообщала об этом явно. Проще всего использовать условный переход типа:

```
HWND hwnd;
hwnd = FindWindow(0, "Калькулятор");
printf("hWnd: "); if (hwnd) printf("%Xh\n"); else printf("error!\n");
```

Листинг 2 правильный, но неэлегантный способ вывода значения на экран

Однако, все это слишком по медвежачьи. Слишком прямолинейно, а прямолинейность для хакеров непростительна! К тому же нам потребовалось целых три вызова функции `printf()` вместо одного. Что, если на тачке установлен целый гектар, то отельные байты можно уже и не считать?! Некоторые, попытавшись неумело схитрить, преобразовывают `hWnd` в строку, посредством нестандартной функцией `_itoa()`, поддерживаемой Microsoft Visual C++, но отсутствующей во многих других компиляторах. В этот случае для вывода значения дескриптора требуется всего лишь один вызов `printf()`, да и сама программа становится намного прозрачнее:

```
char buf[12];  // 12 байт хватит для любого числа, хватило бы и 9 (8 символов + \x0)
HWND hwnd;     // но компилятор все равно выровняет размер buf до кратной 4х
hwnd = FindWindow(0,"Калькулятор");
printf("hWnd: %s\n", (hwnd)?_itoa((int)hwnd, buf, 0x10):"error!");
```

Листинг 3 более элегантный способ вывода дескриптора окна на экран

Программа стала намного более наглядной, но… все равно это не по-хакерски и слишком прямолинейно. А что если… подобрать такую шестнадцатеричную константу, которая бы читалась как осмысленное текстовое слово? Вот например, BADh?

```
HWND hwnd;
hwnd = FindWindow(0, "Калькулятор");
printf("hWnd: %Xh\n", (hwnd)?(int)hwnd:0xBAD);
```

Листинг 4 хакерский способ вывода дескриптора на экран

Исходный текст упростился до предела, оставшись наглядным и понятным даже обычным, "ванильным" программистам, которые кроме прикладных программ ничего другого писать не умеют.

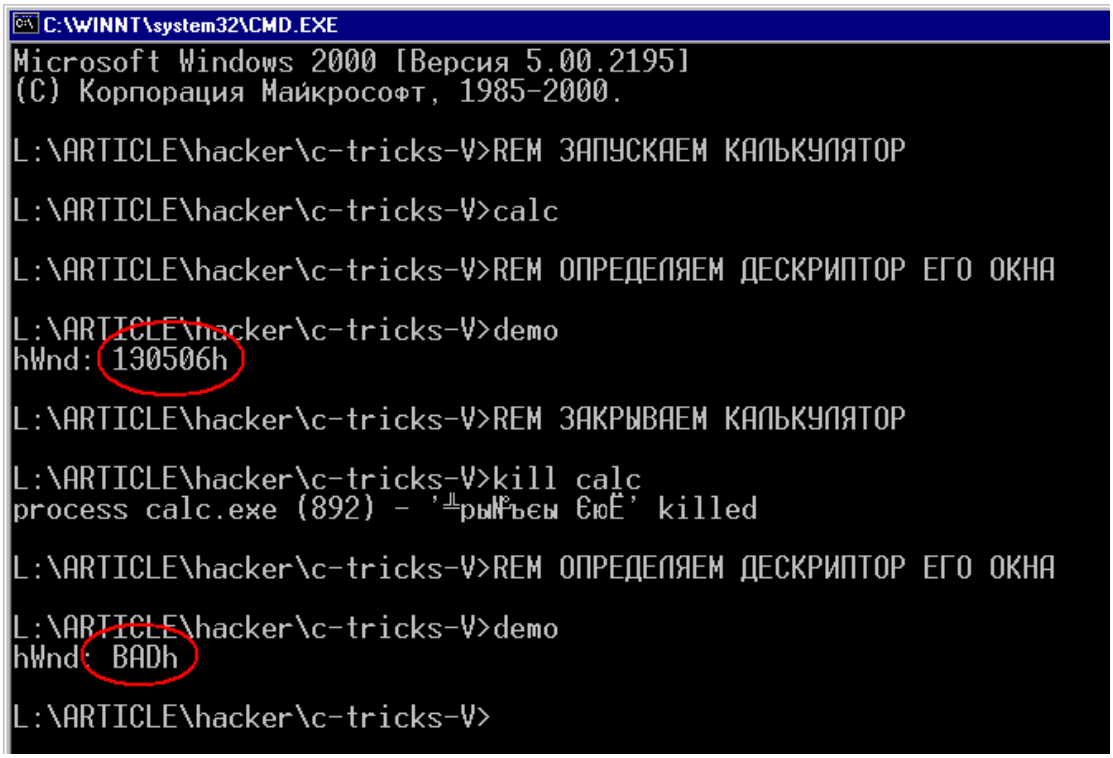

Рисунок 1 демонстрация хакерского способа вывода дескриптора на экран

Разумеется, данную методику можно применять не только с дескрипторами окон, но и вообще с _любыми_ возвращаемыми значениями и не только с API-функциями, но и своими собственными. Причем, своя собственная функция запросто может сделать `return 0xBAD` в случае ошибки. И тогда вместо проверки в стиле `if (foo()!=ERROR)` мы будем писать `if (foo()!=0xBAD)`, что намного более элегантнее. Элегантнее не потому, что 0xBAD короче ERROR (оба они одинаковы по длине), а потому, что при записи результата в лог (вы же ведь ведете отладочные логи, верно?) отпадает необходимость преобразования численного кода ошибки в его строковое представление.

Помимо 0xBAD существуют и другие комбинации, например, 0xDEADBEEF (дословно: бык умер), 0xDEADA11, 0xFA11ED, да много всего можно придумать! Главное — фантазию иметь. Кстати, составление осмысленных слов из hex-символов само по себе является нехилой головоломкой и отличной гимнастикой для мозгов! Так что дерзайте!

локальные переменные коллективного использования

Некоторые, может быть, еще помнят те времена когда в СССР существовало такое понятие как "персональный компьютер коллективного использования". Сейчас персоналки подешевели настолько, что эта проблема отпала сама собой, но вот локальные переменные… создают гораздо больше проблем, чем все члены политбюро все вместе взятые.

Этикет программирования ограничивает предельно разумную длину функций несколькими сотнями строк, рекомендуя дробить функции на элементарные функциональные единицы, которые проще отлаживать, да и компилируются они быстрее. Но это — теоретически. Практические же, при "расщеплении" одной большой функции на несколько маленьких возникает проблема с разделом локальных переменных. Да, мы можем обособить фрагмент большой функции в отдельный функциональный фрагмент, но при этом он потянет за собой множество неявных аргументов — например, флагов, управляющих отладочным выводом, дескрипторы файлов, окон, элементов управления, да мало ли еще что!

Конечно, можно передать все необходимые переменные через аргументы, но это будет медленно, неэлегантно и к тому же потребует чертову уйму ручной работы. В приплюсном си эта проблема стоит и не так остро, поскольку там все функции-члены класса могут разделять один и те же переменные, но… с ростом размеров класса количество разделяемых переменных все возрастает и возрастает, порождая путаницу, хаос, беспорядок и вытекающие отсюда ошибки.

А что если… все локальные переменные загнать в структуру, передаваемую всем родственным функциям (по ссылке, конечно, чтобы они могли менять знания как заблагорассудиться).

Вот например:

```
foo()
{
        int a, flag, x = 0, y = 0;

        flag = get_config(is_debug_output_enabled);

        for (a = 0; a < 0x669; a++)
        {
                x ^= a ^ (0-a);if (flag) printf("%d\n", x);
        }

        for (a = 0; a < 0x999; a++)
        {
                y ^= x + a >> ( a & 0xF); if (flag) printf("%d\n", y);
        }
}
```

Листинг 5 одна "большая" функция foo(), которая должна быть разбита на несколько маленьких

Допустим, мы хотим разбить функцию `foo()` на две или даже на три, чтобы улучшить читаемость листинга. В классическом варианте это будет выглядеть так:

```
zoo(int flag, int x)
{
        int a, y = 0;

        for (a = 0; a < 0x999; a++)
        {
                y ^= x + a >> ( a & 0xF); if (flag) printf("%d\n", y);
        } return y;
}

bar(int flag)
{
        int a, x = 0;

        for ( a= 0; a< 0x669; a++)
        {
                x ^=a ^ (0-a); if (flag) printf("%d\n", x);
        } return x;
}

foo()
{
        int a, flag;

        flag = get_config(is_debug_output_enabled);

        zoo(flag, bar(flag));
}
```

Листинг 6 классическая разбивка "большой" функции foo() на три маленьких

Данный пример не выглядит ужасающим только потому, что код "раскулачиваемой" функции `foo()` сравнительно невелик и переменных там раз два и обчелся. Но все-таки…. Попробуем их загнать в структуру?

```
struct L{int a; int x; int y; int flag;};

zoo(struct L *l)
{
        for (l->a = 0; l->a < 0x669; l->a++)
        {
                l->y ^= l->x + l->a >> ( l->a & 0xF);
                if (l->flag) printf("%d\n", l->y);
        }
}
```

```
bar(struct L *l)
{
        for (l->a = 0; l->a < 0x669; l->a++)
        {
                l->x ^= l->a ^ (0 - l->a);
                if (l->flag) printf("%d\n", l->x);
        }
}

foo()
{
        struct L l; memset(&l, 0, sizeof(l));

        l.flag = get_config(is_debug_output_enabled);

        zoo(&l); bar(&l);
}
```

Листинг 7 функция foo(), "раздербаненная" при помощи структуры L, обеспечивающей совместное использование переменных между дочерними функциями

В данном случае преимущество не столь очевидно, но в больших проектах оно дает о себе знать! А что на счет эффективности?! Не снижается ли она за счет постоянных операций типа "`l->a`"? Отнюдь! Современные компиляторы легко определяют эффективный адрес элементов структур без промежуточных вычислений. А вот засылка множества аргументов в стек изрядно тормозит.

Кроме того, предложенный метод позволяет безболезненно менять прототипы функций (в том числе и публичных). Скажем, захотелось нам добавить к функции, выводящей изображение спрайта на экран, новый аргумент — коэффициент прозрачности. В классическом случае мы _ничего_ не можем сделать, поскольку это потребует изменений во _всем_ проекта и коллеги из соседних отделов нас тут же поубивают. А в случае со структурами — можно добавлять сколько угодно аргументов. Старый код их просто "не замечает ", зато новый та да!

структуры в борьбе с переполняющимися буферами

Переполняющиеся буфера существовали, существуют и будут существовать. Существует множество защитных механизмов типа Stack-Guard или Stack-Shield, но все это — детские игрушки, не способные остановить атакующего. Протектор Pro-Police, зародившийся в недрах японского отделения IBM (http://www.research.ibm.com/trl/projects/security/ssp/), — это, без преувеличения, самый сложный и самый совершенный механизм, реализующий модель безопасного стека (Safe Stack Usage Model), самой главной инновацией которого является переупорядочивание локальных переменных.

Pro-police разбивает переменные на две группы: массивы и все остальные. На вершину карда стека попадают обычные (т. е. скалярные) переменные. Массивы идут за ними. Переполняющиеся буфера могут воздействовать друг на друга, но до указателей им уже не достать, во всяком случае не таким простым путем. К сожалению, Pro-police работает только с компилятором GCC, а всем остальным остается только сосать (лапу) и в остервенении грызть свой хвост, или… воспользоваться структурами.

Дело в том, что размещение локальных переменных в памяти может и не совпадать с порядком их объявления в программе (уж так компиляторы устроены и против них не попрешь), поэтому, у нас нет никаких гарантий, что переменные p и s окажутся расположенными выше локальных буферов:

```
foo()
{
        int a;
        int b;
        int *p;
        char *s
        char buf1[669];
        char buf2[996];
}
```

Листинг 8 фрагмент программы, потенциально подверженной переполнению с передачей управления на shell-код

А это значит, что при переполнении одного из буферов, атакующий может воздействовать на указатели p и s со всеми вытекающими отсюда последствиями. Напротив, в структурах размещение элементов в памяти _всегда_ совпадет с порядком их объявления!

```
struct L
{
        int a;
        int b;
        int *p;
        char *s
        char buf1[669];
        int canary_1;
        char buf2[996];
        int canary_2;
};
```

Листинг 9 устойчивый к переполнению вариант

Здесь canary_1 и canary_2 – магические переменные, инициализируемые случайным образом при входе в функцию и проверяемые перед выходом из нее. Если же они вдруг оказались искажены, значит, один из буферов был переполнен и адрес возврата, возможно, смотрит на вредоносный shell-код, поэтому вместо возврата мы завершаем программу в аварийном режиме (самое простое, что можно сделать) или передаем управление на специальную функцию, сохраняющую несохраненные данные.

В общем, структуры — это сила! Да пребудут они с вами!