сишные трюки
(9й выпуск)

крис касперски ака мыщъх, ака souriz, aka nezumi, aka elraton, no-email

unions vs нецензурный кастинг

Типизация, призванная оградить программиста от совершения ошибок, хорошо работает лишь на бумаге, а в реальной жизни порождает множество проблем (особенно при низкоуровневом разборе байтов), решаемых с помощью явного преобразования типов или, другим словами "кастинга" (от английского "casting"), например, так:

```
int *p; char x;
…
x = *(((char*)p)+3);   // получить байт, лежащий по смещению 3 от ячейки *p
```

Листинг 1 пример явного преобразования типов

Типизация была серьезно ужесточена в приплюснутом си, вследствие чего количество операций явного преобразования резко возросло, захламляя листинг и культивируя порочный стиль программирования.

Рассмотрим следующую ситуацию (см. листинг 2), функция f00 принимает указатель на char, а функция bar возвращает обобщенный указатель void*, который мы должны передать функции f00, но… не можем этого сделать!!!

```
f00(char *x);  // Функция, ожидающая указателя на char

void* bar();   // функция, возвращающая обобщенный указатель void

f00(bar());    // ошибка!!! указатель на char не равнозначен указателю void*
```

Листинг 2 жесткая типизация приплюснутого си трактует попытку передачи void* вместе char* как ошибку

Компилятор, сообщив об ошибке приведения типов, остановит трансляцию, вынуждая нас на явное преобразование void* в char*. Что здесь плохого? А то, что у программиста вырабатывается устойчивый рефлекс преобразовывать типы всякий раз, когда их не может "проглотить" компилятор, совершенно не обращая внимания на их "совместимость", в результате чего константы сплошь и рядом преобразуются в указатели, а указатели — в константы со всеми вытекающими отсюда последствиями. Но по другому программировать просто не получается!!! Различные функции различных библиотек по разному объявляют _физически_ _идентичные_ типы переменных так что от преобразования никуда не уйти, а ограничиться одной конкретной библиотекой все равно не получится. Платформа .NET выглядит обнадеживающей, но… похожая идея (объять необъятное) уже предпринималась не раз и не два, но всякий раз заканчивалась если не провалом, то… разводом и девичьей фамилией. Взять хотя бы MFC. И попытаться прикрутить ее к чему-нибудь еще, например, к API-функциям операционной системы. Преобразований будет там…

Но частые преобразования _очень_ аноят, особенно если их приходится выполнять над одним и тем же набором переменных. В этом случае можно (и нужно!) использовать объединения, объявляемые ключевым словом union, и позволяющие "легализовать" операции между разнотипными переменными.

Код, приведенный в листинге 1, с использованием объединений выглядит так:

```
union pint2char        /* декларация объединения */
{
        int  *pi;              // указатель на int
        char *pb;              // указатель на char
} ppp;

int *p; char x;                // объявление остальных переменных
…
ppp.pi = p; x = *(ppp.pb+3);  // элегантный уход от кастинга
```

Листинг 3 использование объединений в си для избавления от явного преобразования типов

На первый взгляд, вариант с объединениями даже более громоздкий, чем без них, но объединение достаточно объявить один раз, а потом можно использовать сколько угодно раз, и с каждым разом приносимый им выигрыш будет нарастать, не говоря уже о том, что избавление от явных преобразований улучшают читабельность листинга.

Приплюснутый Си идет еще дальше и поддерживает анонимные объединения, которые можно вызвать без объявления переменной-костыля, которой в данной случае является ppp. Переписанный Листинг 2, выглядит так:

```
union                   /* декларация анонимного объединения */
{
        void *VOID;             // обобщенный указатель *void
        char *CHAR;             // указатель на char
};
VOID = bar(); f00(CHAR);        // уход от кастинга
```

Листинг 4 использование анонимных объединений в приплюснутом си избавляет нас от кастинга, но делает логику работы кода менее очевидной

Анонимные объединения элегантно избавляют нас от кастинга, но… в то же самое время затрудняют чтение листинга, поскольку, из конструкции "VOID = bar(); f00(CHAR);" совершенно неочевидно, что функции f00 передается значение, возращенное bar и не видя объединения можно подумать, что VOID и CHAR это две _разные_ переменные, когда на самом деле это _одна_ физическая ячейка памяти.

В общем, получается замкнутый круг, выхода из которого нет…

сравнение структур

В языке Си отсутствуют механизмы сравнения структур, и все учебники, которые мыщъху только доводилось курить, пишут, что структуры вообще нельзя сравнивать. Во всяком случае — побайтово. Поэлементно — можно, но это не универсально (для каждой структуры приходится писать свою функцию сравнения), непроизводительно и вообще не по-хакерски.

Чем мотивирован запрет на побайтовое сравнение структур? А тем, что компиляторы по умолчанию выравнивают элементы структуры по кратным адресам, обеспечивая минимальное время доступа к данным. Величина выравнивания зависит от конкретной платформы и если она отлична от единицы (как это обычно и бывает), между соседними элементами могут образовываться "дыры", содержимое которых не определено. Вот эти самые дыры и делают побайтовое сравнение ненадежным.

На самом деле, сравнивать структуры все-таки можно. Имеется как минимум два пути решения проблемы. Во-первых, выравнивание можно отключить соответствующей прагмой компилятора или ключом командной строки, тогда дыры исчезнут, но… вместе с ними исчезнет и скорость (во всяком случае, потенциально). Падение производительности в некоторых случаях может быть очень значительным (а некоторые процессоры при обращении к не выровненным данным и вовсе генерируют исключение) и хотя правильной группировкой членов структуры его можно избежать, это не лучшее решение.

Исследование "дыр" (и логики компиляции) показывает, что их содержимое легко сделать определенным. Достаточно перед объявлением структуры (или сразу же после объявления) проинициализировать принадлежащую ей область памяти, забив ее нулями и… это все! Компилятор _никогда_ не изменяет значение "дыр" между элементами структуры и даже если структура передается по значению, она копируется вся целиком вместе со всеми дырами которые только у нее есть (а дыра — это нора!), следовательно, побайтовое сравнение структур _абсолютно_ надежно. Главное, не забывать об инициализации дыр, которая в общем случае делается так:

```
struct my_struct                /* декларация произвольной структуры */
{
        int a;
        char b;
        int c;
};

struct my_struct XX;            // объявление структуры XX (дыры заполнены мусором)
struct my_struct XY;            // объявление структуры XY (дыры заполнены мусором)

memset(&XX, 0, sizeof(XX));     // инициализируем область памяти структуры XX
memset(&XY, 0, sizeof(XY));     // инициализируем область памяти структуры XY
```

```
…
// что-то делаем со структурами
…

if (!memcmp(&XX, &XY, sizeof(XX))
        /* структуры идентичны */
else
        /* структуры _не_ идентичны */
```

Листинг 5 "обнуление" области памяти, занятой структурой, дает зеленый свет операции побайтового сравнения

strncpy vs strcpy

В борьбе с переполняющимися буферами программисты перелопачивают тонны исходного кода на погонный метр, заменяя все потенциально опасные функции их безопасными аналогами с суффиксом "n", позволяющим задать предельный размер обрабатываемой строки или блока памяти.

Часто подобная замена делается чисто механически без учета специфики n-функций и не только не устраняет ошибки, но даже увеличивает их число. Вероятно, самым популярным ляпом является замена strcpy на strncpy.

Рассмотрим код вида:

```
f00(char *s)
{
        char buf[BUF_SIZE];
        …
        strcpy(buf,s);
}
```

Листинг 6 потенциально опасный код, подверженный переполнению

Если длина строки s превысит размер буфера buf, произойдет переполнение, результатом которого зачастую становится полная капитуляция компьютера перед злоумышленником, чего допускать ни в коем случае нельзя и в благородном порыве гражданского долга многие переписывают потенциально опасный код так:

```
f00(char *s)
{
        char buf[BUF_SIZE];
        …
        strncpy(buf,s, BUF_SIZE);
}
```

Листинг 7 исправленный, но по-прежнему потенциально опасный вариант того же самого кода

или так…

```
f00(char *s)
{
        char buf[BUF_SIZE];
        …
        strncpy(buf,s, BUF_SIZE-1);
}
```

Листинг 8 …еще один потенциально опасный вариант

Хе-хе. Если размер строки s превысит значение BUF_SIZE (или BUF_SIZE-1), функция strncpy прервет копирование, "забыв" поставить завершающий ноль. Причем, об этом будет очень трудно узнать, поскольку сообщение об ошибке в этом случае не возвращается, а попытка определить фактическую длину скопированной строки через strlen(buf) ни к чему хорошему не приводит, поскольку в отсутствии завершающего нуля в лучшем случае мы получаем неверный размер, в худшем — исключение.

Находятся программисты, добавляют завершающий ноль вручную, делая это приблизительно так:

```
f00(char *s)
```

```
{
        char buf[BUF_SIZE];
        …
        buf[BUF_SIZE-1] = 0;
        strncpy(buf,s, BUF_SIZE-1);
}
```

Листинг 9 не подверженный переполнению, но по-прежнему неправильно работающий код

Такой код вполне безопасен в плане переполнения, однако, порочен, греховен и ненадежен, поскольку, маскирует факт "обрезания" строки, что приводит к непредсказуемой работе программы. Вот только один пример. Допустим, в переменной s передается путь к каталогу для удаления его содержимого. Допустим так же, что в силу каких-то обстоятельств, длина пути превысит BUF_SIZE и он окажется усечен. Если усечение произойдет на границе "\", то удаленным окажется совсем другой каталог, причем каталог более высокого уровня!!!

Самый простой и единственно правильный вариант выглядит как показано в листинге 10, а функция strncpy, кстати говоря, изначально задумывалась для копирования не ASCIIZ строк, т.е. строк _не_ содержащих символа завершающего нуля и это совсем не аналог strcpy! Эти две функции _не_ взаимозаменяемы!

```
f00(char *s)
{
        char buf[BUF_SIZE];
        …
        if (strlen(s)>=BUF_SIZE) return ERROR; else strcpy(buf,s);
}
```

Листинг 10 безопасный и правильно работающий вариант