Идеальная проверка программиста перед приемом на работу.

Я начал программировать уже давно, году эдак в 2010. Если совсем честно, то я начал программировать из интереса, так как учился не писать программы, а ломать их. В этом мне сопутствовали книги Криса Касперски, желание быть похожим на кого-нибудь вроде Кевина Митника и конечно-же львиное упорство, достойное лучшего применения, чем просиживание штанов за компьютером долгие ночи напролет до 7:30 утра. Мне нужны были программы для тестов, а потом мне захотелось чего-то настоящего, сильного. Мне надоело ковыряться в чужих поделках, я захотел иметь свои. Ну кто я такой, если просто ломаю, но сам ничего не пишу? И я начал программировать, так как ломать стало скучно.

Я писал программы, изучал их в отладчике и дизассемблере, редактировал свои и чужие программы, и приобрел то, что нельзя пропить от слова совсем — выстраданный опыт. Понимаете-ли, выстраданный опыт, это такой опыт, который показывает человеку, что он может и чего он не может, проводит раскаленным железом жирную красную черту между человеком настоящим и тем павлином, которым он себя мнит, так как он человек, а человек существо ну уж очень много о себе думающее. Я тоже многое о себе думал, пока не напоролся брюхом на защиту от отладки в программе WinHex. Ну очень уж хотелось иметь профессиональный HEX-редактор имени себя любимого, да и еще который не просит файл лицензии. Из-за моей невнимательности, к тому-же я забыл включить API монитор, защита сработала и сделала программу постоянно просящей приобретения лицензии для специалистов в форензике (авторы защиты попались с юмором) а так-же отказалась удаляться с диска даже из под учетной записи администратора. Защита заехала мне «сапогом в пах» и я быстренько сполз с облаков самомнения по стене разочарования в реальный мир. Программу я так и не сломал, а чувство, что меня унизили, преследует меня до сих пор. Правильно говорят, раны заживут, душа — никогда.

Моей первой программой была даже не программа, а скрипт для Windows XP, который уводил компьютер в бесконечный цикл перезагрузки. Первые программы которые я писал на Pascal и Delphi были такими, про которые говорят «говнокод», «ошибка в ДНК» и «автор, убейся об стену». На часть такой критики я смотрел как на что-то не достойное не то что плевка, а моего драгоценного внимания (буду я еще удары сердца на всяких там [отборная матершина] тратить), к другой части прислушивался, так как это была полезная информация, обычно со ссылкой на стандарт, книжку или учебник, писал программы и мне было глубоко фиолетово кто там что выкудахтал про мои программы, которые я считал конечно же идеалом логики и всего такого научного.

С тех пор утекло много воды, возраст перевалил за 30 и жизнь чего-то уже совсем не та. Но, я знаю как тестировать таких как я, в смысле программистов, да тестировать так, чтобы ни один «программистишка-входильщик» не пролез в этот дивный мир добра и безусловно высоконаучных технологий, с каждой пикосекундой делающих мир все лучше и лучше. Вот например Microsoft Windows 11, ну чем не шедевр инженерии и высшей алгебры?

Разгрызая камни

В интернете написали, что программист это профессиональный конвертер галлюцинаций заказчика в жёсткую формальную систему. И знаете что, написана-то жесткая правда жизни. Заказчик может не знать чего хочет, когда это должно быть сделано и вообще на какой он планете, только сколько-бы заказчик не галлюцинировал, программист все равно должен знать, чем и как сделать так, чтобы этот галлюцинирующий был если не доволен как слон, то уж точно не накричал да и еще денежку заплатил.

Когда я только начинал программировать, я уже знал и про системные вызовы, и про стек, про переполнение буфера в стеке с перезаписью адреса возврата, и даже про передачу управления в середину команды (нуб, без вас знаю). Как делать большие вещи, вроде текстовых редакторов, интернет браузеров или вообще драйверов и операционных систем, я конечно не знал. Пришлось учится долго и упорно, чтобы соответствовать профессиональному уровню, который сам же себе и придумал, а придумал я ой-ёй. А я еще такой человек, который не то что легких путей не ищет, а сначала лезет в дымоход без фонаря и веревки, а застряв там, головой ломает его изнутри чтобы вылезти, потому что не знал куда лезет и думал «да я вам щас тут как наинженерю шо мама дорогая».

Первое что пришлось понять, это переменные. Что это вообще такое и зачем оно надо? Переменная (как до меня дошло) это такая именованная (!) область памяти (все значения хранятся в памяти, ну кроме регистровых) для которой задан диапазон хранимых значений а так-же список разрешенных операций, т.е переменная имеет тип. В переменной что-то хранят, число, строку, адрес, целую структуру и т.д, а еще с переменной что-то делают. Сами подумайте, вот у нас есть переменная типа int (я сейчас про язык Си) с именем X. Можем ли мы назначить ей начальное значение 0? Конечно можем. А увеличить его на 1? Тоже можем. А уменьшить на 1 и получить на 32-разрядной машине значение в HEX виде 0xFFFFFFFF? Ну конечно-же можем. Сделать вот так

```
X = *(&X + 1);
```

можем еще как, это же Си блин ё-моё! А вот сделать так

```
X = *(X-- + X--);
```

уже нет. Казалось бы, что тут такого, берешь число, отнимаешь 1, потом еще раз, складываешь с переполнением и трактуя результат вычисления выражения как адрес, разумеется плюя на отсутствие явного приведения типа результата вычисления выражения к (int*) и неопределенное поведение, получаешь число по этому адресу с помощью разыменования. Почему нельзя-то? Потому что нельзя использовать унарную операцию разыменования для выражений с операндами типа int. Для int никакого разыменования не предусмотрено. А вот сделать так

```
X = *(int *)(X-- + X++);
printf("%X\n", X);
```

совершенно законно. Такие уж правила языка, ничего не поделаешь. Компилятор вообще выдает нам строку cast to pointer from integer of different size не считая это ошибкой. Только программа что-то падает… Дело тут в том, что операционная система пристально следит за тем что процесс делает и не позволяет ему лазить по виртуальным адресам, которые в структуре данных именуемой процессом, относятся к какой-то другой области памяти, нежели к адресам в стеке, куче или секции данных. Можно долго рассуждать о виртуальной памяти, секциях, страничной адресации, каком-нибудь MMU, или вообще притянуть сюда VirtualProtect(), но что-то не хочется писать текст на 200 страниц, к тому-же до меня об этом итак уже много понаписали.

Короче получается, что пытаться переменной X присвоить значение, лежащее по адресу 0xFFFFFFFE (почему именно такому до сих пор гадаю) бесполезно, так-как по этому адресу прочитать ничего нельзя (ОС не позволит). В Си можно даже вырвав с корнем педаль тормоза и высунув голову через лобовое стекло, натворить такое

```
X = *(((X-- + X--) & 0) + &X);
printf("%X\n", X);
```

Попробуйте сделать такое в Pascal, компилятор тут-же и нещадно поцарапает вам вашу самоуверенность. Язык Си вот прямо провоцирует программиста на опасное лихачество, оно и понятно, так как создан хакерами в их уютном аквариуме бородато-волосатых физико-математиков, для себя любимых и себе подобных. Напоминаю, придуман чтобы заменить язык ассемблера, т.е облегчить себе работу!

Переменная открылась мне как бы коробочкой, которая может быть устроена черти-как, всегда расположенной по определенному смещению в гигантском массиве памяти, которая не может хранить данные иного типа и размера, чем определено изначально (если она статическая), для которой определен список действий, которые мы можем с ней вытворять. Это может быть инициализация (задать начальное значение), присваивание (открыть ларчик, положить туда значение, закрыть ларчик), инкремент, разыменование, взятие адреса и т. д. Причем не только переменные имеют какой-то там тип, но и сами значения, например 1 в языке Си, это число именно типа int.

Строки оказались устроенны сложнее. Они бывают статическими и динамическими, устроены они по разному, я расскажу только про Сишные и Pascal-строки, так как про строки, реализованные в виде структуры данных, хранящей указатель на односвязный список статических массивов а так-же целочисленную переменную, мне писать просто лень. Итак, в Си строк нет вообще, так как мы не можем их например присваивать, а есть указатели на массивы с 0 в конце, да и массивов тоже нет. Такие строки называются нуль-терминированные. Любая функция стандартной библиотеки языка Си для работы со строками, это функция которая сразу прекращает работу как только натыкается на байт 0x00, и если такого байта в строке нет, вы получаете в лучшем случае выход за границы массива при чтении. Сделать с «массивами» в Си что-то такое

```
char data1[10] = {0xC0};
char data2[10] = {0xDE};

data1 = data2;
```

мы не сможем никогда. Pascal другое дело, там такое провернуть легко, так как и строки и массивы там есть. Отличие в том, что Pascal-строка это массив, первый байт которого есть длина строки в байтах, а значит переменная типа Pascal-строка не может хранить данных больше чем 255 байт.

Сравнение строк это вообще песня. Вот нужно сравнить две строки, а строки у нас Сишные, ну мы и делаем так

```
if (strcmp(data_from_user, password) == 0) {
  return 0;
}
```

Т.е чтобы узнать равны-ли строки (самые обычные строки, которых в языке нет), нам приходится передавать адреса этих строк в функцию стандартной библиотеки, и функции конечно все равно, передаем мы нулевые адреса или нет (что кстати некоторых людей довело до психушки).

Функция, по переданным в качестве аргументов адресам, прочитает байты, вычислит их разность, и если разность не равна 0, то и байты не равны, а значит нужно вернуть эту разность, не важно со знаком она или без, так как эта разность означает насколько левая

строка «больше» правой. Если же разность байтов равна 0, это значит что байты равны, а значит осталось проверить, не нулевой ли байт нам попался в левой строке, и если попался, то байты из обоих строк нулевые. Разность нулей означает 0, и мы получаем равенство строк, что означает (что логично) равенство их длин.

Нормальная библиотечная функция должна делать вот такое

```
int strcmp(const char *l, const char *r) {
  register int t;

  while(1) {
    if (t = *l - *r)
      break;
    if (!*l)
      break;

    l++, r++;
  }

  return t;
}
```

Но мир у нас не нормальный, точнее не все соблюдают нормы, что понятно, так как не все люди нормальные и не все нормы установлены нормальными людьми. ЯФинн вот посчитал, что разность вычлененных из строк байтов возвращать не обязательно, как это делает нормальная библиотечная функция, и обошелся тремя возвращаемыми значениями, 0, 1 и -1. Вот код функции strcmp из ядра Linux

```
int strcmp(const char *cs, const char *ct)
{
    unsigned char c1, c2;

    while (1) {
        c1 = *cs++;
        c2 = *ct++;
        if (c1 != c2)
            return c1 < c2 ? -1 : 1;
        if (!c1)
            break;
    }
    return 0;
}
```

Функция вычленит из строк байты, сравнит их, и если они оказались не равны, то в зависимости от того, больше или меньше байт из левой строки, байта из правой строки, вернет 1 или -1, и плевать ему на разность кодов символов. Если же функция вернет 0, это значит что строки равны. С длинами строк функция strcmp не работает, делая вид что их как-будто нет. Их и правда нет, функция их даже не считает, но могла бы.

А зачем нам вообще может понадобиться знать, что первая строка «больше» второй на 2 или -27? Для сортировки виртуальных адресов в массиве указателей на строки, нам достаточно знать, больше-ли левая строка правой. На сколько больше, совершенно не важно. Можно обойтись всего двумя возвращаемыми значениями, 0 и 1. 0 будет означать равенство, 1 будет значить что левая строка больше правой. Если левая строка оказалась больше правой, т.е функция вернула 1, их как минимум нужно поменять местами так, чтобы адрес большей строки был в массиве справа, а меньшей слева. Отсортировать строки для будущей печати,

выполнив сортировку методом сравнения строк и какого-нибудь пузырька (представляете, я сам до этого додумался), мы могли бы например так

```
void str_sort(char ** arr_str, int len) {
  int    i, swp;
  char * tmp;

  while (1) {
    swp = 0;
    i   = 0;

    while (i < len-1) {
      if (strcmp(arr_str[i], arr_str[i+1]) > 0) {
        tmp          = arr_str[i];
        arr_str[i]   = arr_str[i+1];
        arr_str[i+1] = tmp;
        swp = 1;
      }
      i++;
    }

    len--;

    if (!swp) break;
  }
}
```

Кстати, не хотите-ли попробовать оптимизировать эту сортировку строк? Просто так. Выполнить оптимизацию ради оптимизации. Берете, и оптимизируете, чтобы работало побыстрее. Хорошая задачка чтобы «встряхнуть мозги».

Так зачем-же нам разность кодов не совпадающих символов-то? Зачем-нибудь, функция ведь так устроена. ЯФинн по своему прав, ему видимо в ядре разность не нужна, а вот разработчикам стандартной библиотеки языка Си нужна, как и множеству разработчиков программ, но вдумайтесь, в какое же болото мы забрели…

В Pascal и Delphi все совсем по другому. Вот у вас две строки, сравним же их

```
if data_from_user = password then
begin
  result := 0;
end;
```

Что такое знак равно в этом коде? Это и есть сравнение строк. Строки кстати хранят свои длины, которые вставленный компилятором код, посчитал в процессе присваивания строке ее значения. Это компилятор от нас скрывает. Так-же он скрывает описание сравнения (что происходит), вытворяя следующее. Для начала адреса обеих строк передаются функции, отвечающей за сравнение строк (компилятор сам ее вставит куда надо), сама же функция может быть устроена так, как посчитали правильным авторы компилятора, только вот функция должна соответствовать некоторым требованиям, но сейчас не о них. Первое что делает функция, это сравнивает длины строк, так как сами строки хранят в себе свои длины. Если длины строк равны и любая из длин равна 0, то функция вернет true, ничего больше не проверяя на равенство. Такие строки называются пустыми, т.е в такой строке нет данных, но строка все равно есть, и у нее есть длина, хоть и равна она 0. Тут нужно вспомнить, что в современных Pascal и Delphi используются строки типа AnsiString, строка этого типа может занимать целых 2 Гб, сама же длина строки занимает 4 байта, а память под строку

динамически выделяется кодом вставленным компилятором, например путем обращения к ядру системы напрямую, либо как-то там еще.

Понятно что если длины строк не равны, то и строки не равны, а вот если длины строк равны и любая из длин больше 0, то функция начинает блоками по 4 байта (если использовался 32-битный компилятор) сравнивать вычлененные из строк данные, ну или делает это по 1 байту, как в Си. Функция может делать такое сравнение через отнимание, принимая за равенство нулевую разность или через xor (исключающее или), принимая за равенство нулевой результат операции (xor как мне сказали, почти всегда быстрее). Какая именно у нас может получится разность не существенно, так как функция возвращает 0 если строки не равны либо 1 если равны.

Функция может, и будет сравнивать данные блоками, так как так быстрее, но чтобы сравнивать блоками и не вылезти за границу массива при чтении, нужно знать, сколько блоков мы можем прочесть и допустимо ли вообще чтение блоками.

Тут функция начинает свои расчеты. Вы понимаете? Расчеты! Если длины строк равны и любая из длин больше 0, функция вычисляет остаток от деления длины любой из строк (назовем ее опорной) на 4, вычленяя младшие 2 бита длины опорной строки операцией логическое И с числом 3 или 00000011b, и если получился не 0, значит у нас проблема, так как тогда длина строки от 1 до 3 байт, если же длина больше, то длина строки как минимум не кратна 4 (это мы обсудим позже).

Потом функция делит без остатка длину опорной строки на размер блока, через битовый сдвиг на 2 бита вправо, и смотрит, получился ли 0. Если получился, значит сравнение блоками по 4 байта невозможно, так как количество блоков для сравнения меньше минимального, т.е меньше 1, и начнется с выхода за границу массива при чтении, а этого допустить никак нельзя (у нас тут не Сишка). Тогда функция сравнивает вычлененные байты, допустим по одному. Если получился не 0, а допустим 3, это значит что как минимум 3 блока по 4 байта каждый, мы можем сравнить ну допустим через xor, вообще не думая почему именно xor. А вот дальше у нас ситуация ой-ёй.

Если длины обоих строк у нас 13 байт, а блоков по 4 байта у нас 3, то такая функция сравнит всего 12 байт за 3 операции xor, а 13-й байт она не заметит, что в некоторых случаях грозит катастрофой. И тут функция начинает хитрить. Если строки равны, то сравнение всех 3 блоков для функции заканчивается обнулением счетчика блоков, а значит сравнивать блоками больше нечего. Функция смотрит есть ли что еще сравнивать, проверяя переменную в которую были сохранены как раз-таки 2 ранее вычлененных младших бита длины опорной строки. Если там лежит 0, длины строк равны а сравнение строк мы уже закончили и установили их равенство, то мы попали в яблочко, и можно завершать сравнение, гордо вернув в вызывающий код абсолютно точное и безапелляционное true. Если там что-то кроме 0, то нужно установить, сравнивались ли блоки хотя-бы один раз, и если нет, то использовать побайтное сравнение (тогда длина опорной строки от 1 до 3). Если блоки сравнивались хотя-бы 1 раз, то адрес байта который требуется с чем-то там сравнить, а расположен байт за последним блоком, уж точно больше, чем адрес начала этой-же строки, и больше как минимум на 4. Если отнять от адреса этого байта 4, то при чтении данных по получившемуся адресу мы точно не выйдем за переднюю границу массива. Только вот отнимать от адресов сравниваемых строк нужно не 4, в разность ширины блока в байтах и остатка от деления длины строки на ширину блока в 4 байта. Если остаток у нас любой, например равен 1 (13 - (4 * 3) = 1) как в примере выше, то можно отнимать не разность, а наоборот, отнимать размер блока и прибавлять остаток, результат будет тот-же. А если совсем обнаглеть и написать такую функцию на языке Си, будучи пьяным в хламушку, то получится что-то такое

```
#define BLOCK_SIZE sizeof(size_t)

int pascal_strcmp(const unsigned char * l, const unsigned char * r) {
  const unsigned char *sl, *sr;
  unsigned char len;

  if (!l || !r) return 0;
  if (l == r)   return 1;

  if (*l ^ *r)  return 0; /* a xor !a == !0*/
  if (0 == *l)  return 1; /* empty strings */

  len = *l;
  sl  = l+1;
  sr  = r+1;

  if (len < BLOCK_SIZE) { /* len > 0 && len < (4 or 8) */
    do {
      if (*sl ^ *sr)
        return 0;

      sl++, sr++;
    } while(--len);
  }

  while (len >= BLOCK_SIZE) {
    if (*(size_t*)sl ^ *(size_t*)sr)
      return 0;

    sl  += BLOCK_SIZE;
    sr  += BLOCK_SIZE;
    len -= BLOCK_SIZE;
  }

  if (len && sl > l+1) { /* be copy min 1 block and len > 0 */
    if (*(size_t*)(sl - BLOCK_SIZE + len) ^ *(size_t*)(sr - BLOCK_SIZE + len))
      return 0;
  }

  return 1;
}
```

Функция отмучилась, и теперь гордо вернет результат, 1 или 0. И все это безобразие делается как-бы само, т.е программист сравнивающий строки тут не причем, он написал код, а компилятор сделает за него грязную работу. Программист всего-то написал знак равенства между двумя переменными строкового типа, а тут получается вот такая вакханалия, еще адреса какие-то, битовые операции, коррекция указателя при сравнении строк, длины которых хоть и совпали, но сами длины не кратны длине блока (т.е размеру регистра процессора в 8-битных байтах, под который писался компилятор). И это только реализация попавшаяся мне, а так как Pascal-ей существует целая куча, то и реализаций такой вот функции тоже может быть например больше двух. Всего-то нужно было сравнить проклятые строки, а я вон уже сколько понаписал. Ну жесть и трепет же!

В других языках строки устроены по разному, в некоторых языках строки это вообще не строки а целые объекты (например как в Python), а значит если дать строке имя X, то совершенно нормально делать со строкой что-то такое

```
string X;
```

```
if (X.Read(stdin) == false)
  return ERR_READ_STR;

pos = X.Search('\n');

if (pos < 0)
  return ERR_SRC_NSTR;

X.SetLength(pos);

print(X);
```

Да я знаю что этот код очень уж похож на код нуба только-только начинающего штурмовать C++, но если строка это объект, то это не просто какие-то там данные, а целый Object, с которым еще нужно уметь обращаться, для чего нужно обязательно понимать ООП. И заметьте, почти все что написано выше, в полубреду разумеется, но написано только про переменные всего двух типов данных, про целые числа и строки, а так-же лишь про сравнение строк в двух языках программирования. Это что такое в мире делается-то?

Это и был первый камень, который пришлось разгрызть на пути становления программистом, а камушков в программировании столько, что на десятилетие хватит. И вот я принялся за переменные, но так как человек я не совсем обычный, то начал изучать переменные и типы данных как привык, в отладчике и дизассемблере. Ох и огреб же я тогда…

Про Python я нагло умолчу. Вы только полюбуйтесь на эту красоту

```
list_users = {
  'erik':   '#F318-541-909',
  'dakota': '#F318-541-910',
  'sam':    '#F318-541-911',
  'entony': '#F318-541-912',
  'petra':  '#F318-541-913',
  'harry':  '#F318-541-914',
  'jack':   '#F318-541-915'
}

active = True

while active:
  str = input("Name for searching:")

  if str == "":
    print("[!] Empty data for search!")
    continue

  str = str.strip()
  str = str.lower()

  if str == "~exit":
    active = False
    print("<Exit>")
    continue

  if str in list_users.keys():
    print(str.title() + "\tUID:" + list_users[str])
  else:
    print("[!] String " + "<" + str + ">" + " not found!")
```

```
exit()
```

Лишь упомяну, что переменная list_users в коде выше, на первый взгляд НЕ имеет типа а также НЕ может содержать значения с совпадающими ключами, а если такие и попадутся, то Python нагло проигнорирует все, кроме последнего. Я не шучу, я проверял.

Почему же я умолчу? Чего же я боюсь, почему не хочу порассуждать-то? Ну… Это просто переменная какого-то там типа, тип вот так устроен, типизация языка программирования Python динамическая, на этом все должны быть довольны. Если не довольны, ну и что теперь? Я не буду писать сотни строк на языке Си, чтобы ответить на вопрос, почему это е*учая переменная не может содержать значения с повторяющимися ключами, так как я не знаю, но подозреваю, что это просто сахарок поверх хэш-таблицы. Чем еще объяснить одинаковую скорость возвращения значения по любому ключу?

Наше всё

Есть такая страшная для многих людей из мира IT вещь, которую нехорошие люди все чаще и чаще используют для издевательства над людьми. Вешь эта не хорошая и не плохая, она часто помогает, иногда бесит, но ее боятся до ужаса, особенно те, кто использует библиотеки или вообще фреймворки (кирилицей). Её не боятся математики (мне так кажется), информатики из какого-нибудь ИТМО, а так-же люди которые писали например компиляторы, СУБД с нуля, разработчики серверов потокового видео ну и конечно-же какой-нибудь анестезиолог из Мельбурна. Да, я имею ввиду алгоритмы и структуры данных. Я не просто так решил об этом написать. Будет скучно, больно, и все попытки увидеть во мне что-то человеческое, сверните в трубочку и… Короче погнали.

Начнем со стека, так как массивы мы уже задели, они не интересные, ну их в баню. Что такое stack (англ. Стопка [например стопка книг])? Это такое место, куда можно затолкать что-то (число например) и потом извлечь. Все что попало в стек, хранится в том порядке, в котором помещали, или в обратом, и именно по этому стеки бывают разные. В них можно помещать значение с начала, с конца, и извлекать так-же. Мы разберем тот стек, который для меня стал первым, и его же я считаю самым легким. Этот стек называют LIFO, т.е Last Input, First Output (англ. Последним пришел — первым ушел).

Как-то мне понадобился стек в трансляторе моего игрушечного языка программирования Chipmunk. Стек хранил место в коде, перед которым происходил вызов чего-либо, что было нужно для прыжка назад. Приводило это к возврату в то же самое место после завершения вызванной операции. Например при реализации цикла требуется запоминать где он начался, чтобы в конце цикла вернутся к условию в его операторе, или при вызове функции нужно запомнить адрес следующей команды, чтобы начать выполнение с нее, когда вызванная функция отработает и вернет управление. Каждый раз при запоминании места вызывать malloc() мне показалось лишним, ну я и сделал так:

```
#define STACK_SIZE 1000

typedef struct stack {
  int    number[STACK_SIZE];
  size_t counter;
  struct stack * prev;
} stack_t;

void stack_free(stack_t ** stack) {
  stack_t * tmp;
```

```
  while(*stack) {
    tmp = *stack;
    *stack = (*stack)->prev;
    free(tmp);
  }
}

void stack_new(stack_t ** stack, int number) {
  stack_t * tmp = (stack_t *)malloc(sizeof(stack_t));

  if (NULL == tmp) {
    stack_free(stack);
    exit(1);
  }

  tmp->number[0] = number;
  tmp->counter   = 1;
  tmp->prev      = *stack;
  *stack         = tmp;
}

void stack_push(stack_t ** stack, int number) {
  if (*stack && (*stack)->counter < STACK_SIZE) {
    (*stack)->number[(*stack)->counter] = number;
    (*stack)->counter++;
    return;
  }

  stack_new(stack, number);
}

void stack_pop(stack_t ** stack, int * number) {
  stack_t * tmp;

  if (NULL == *stack) return;

  if (0 == (*stack)->counter) {
    tmp = (*stack)->prev;
    free(*stack);
    *stack = tmp;
  }

  if (NULL == *stack) return;

  *number = (*stack)->number[(*stack)->counter-1];
  (*stack)->counter--;
}
```

Не стоит пугаться, но если вы напуганы или даже смущены, ничего страшного, это скоро пройдет. Я тоже когда-то смотрел на весь этот код как на каракули дитеныша десептикона, и ничего, почти не сошел с ума.

Теперь подробно. Структура по имени stack_t содержит три элемента. Массив number это массив чисел типа int, хранящий слева направо уже помещенные в стек значения. Значение переменной counter при выделении памяти под структуру сначала равна 1, так как это минимальное число значений, которое хранит массив. Когда значение достигает 1000, выделяется память под новую структуру, и значения помещаются уже в нее. Указатель на саму структуру по имени prev сначала содержит значение 0, и только если массива на 1000 чисел не хватило для хранения помещаемых в стек чисел, указатель начинает хранить адрес новой такой-же структуры. Короче получился не стек, а односвязный список статических массивов с индексацией, работающий как стек. Во как! Функция stack_push помещает число

в стек, если стека нет, создает его. Функция stack_pop извлекает число из стека, а если числа в стеке иссякли, стек испаряется.

Если понадобилось затолкать в стек 10000 чисел от 0 до 9999 и напечатать их в обратном порядке, можно сделать это вот так просто (так как колличество помещаемых и извлекаемых чисел равно, вызов stack_free тут лишний):

```
stack_t * stack = NULL;

for (int i = 0; i < 10000; i++)
  stack_push(&stack, i);

for (int i = 0; i < 10000; i++) {
  int number;
  stack_pop(&stack, &number);
  printf("%d\n", number);
}
```

На какие мысли вас наколкнул этот код? Ничего не напоминает? Мне вот что-то напоминает, но не помню что. В интернет лезть лень, а на часах уже 04:57, голова квадратная, думать сложно. Я точно помню что что-то такое я уже где-то видел, в каком-то языке программирования, только с нуля писать почти ничего не нужно было, и хранить можно было данные любых типов, а так-же извлекать их с любого конца. Ну ладно, со стеком покончено.

Как-то вечером я наткнулся на видео Тимофея Хирьянова про сортировки. Конкретно разбиралась поразрядная сортировка, а я о такой первый раз слышал. Посмотрел видео, попробовал написать код и с подозрением смотря на свою писанину, застыл. Вот код:

```
void radix_sort(uint8_t * data, size_t size) {
  size_t i, ps0, ps1;
  uint8_t   k = 0;
  uint8_t   r;
  uint8_t * p[2];

  p[0] = (uint8_t *)malloc(size);
  p[1] = (uint8_t *)malloc(size);

  if (NULL == p[0] || NULL == p[1]) {
    if (p[0]) free(p[0]);
    if (p[1]) free(p[1]);
    return;
  }

  r = 1;

  for (i = 0; i < size; i++) {
    k = k | data[i];
    if (k & 0x80)
      break;
  }

  i = 8 - 1;

  while (i >= 0) {
    if ((k >> i) & 1) {
      break;
    }

    if (i == 0) {
```

```
      break;
    }

    i--;
  }

  k = 8 - (8 - (i + 1));

  while (1) {
    ps0 = 0;
    ps1 = 0;

    for (i = 0; i < size; i++) {
      if (data[i] & r) {
        p[0][ps0] = data[i];
        ps0++;
      }
      else {
        p[1][ps1] = data[i];
        ps1++;
      }
    }

    memcpy(data,       p[1], ps1);
    memcpy(data + ps1, p[0], ps0);

    if ((k == 0) || (r & 0x80)) {
      break;
    }

    r <<= 1;
    k--;
  }

  free(p[0]);
  free(p[1]);
}
```

Я никогда не писал шедевров и никогда не любил хвастаться, но вдумавшись в писанину понял, что сделал больше чем мне рассказал Тимофей в своей видеолекции. Больше! Внимательно прочитайте этот код и скажите, какую сложность он имеет. Я не про измучившее мир О-большое, я про то, насколько алгоритм по инженерному хорош. Вот мы берем массив чисел типа unsigned char, и в цикле вычленяем из него числа с нулем и единицей в позиции сначала самого младшего бита, потом постарше и заканчивая самым старшим битом. Это приводит нас к O(n*k), где n — размер массива, а k — колличество битов числа, причем имеем сложность O(2N) по памяти. Анализируем биты операцией AND, так как по другому я не знаю как сделать. Числа с 0 идут налево, числа с 1 направо. Создав два подмассива, объединяем их, копируя в начало сортируемого массива тот подмассив где нули, в конец копируем подмассив где единички, и делаем так пока не пройдем все 8 битов числа типа unsigned char. Хм, я ведь ничего не упустил?

Посмотрев на код, до меня дошло, что я понял мысль Тимофея, но не так как он объяснял, точнее пока он объяснял, что-то в голове булькнуло, и я понял его слова по своему. Я допонял! Я сначала начал объединять операцией ИЛИ все числа массива, проверяя после каждой операции не установился ли самый старший бит. Если установился, значит придется сортировать массив по всем разрядам числа, если нет, то сортировку этого разряда можно выбросить. Например, у нас есть 16 чисел от 0 до 15, эти числа в битах будут от 0000 до 1111. Старшие 4 бита любого числа массива всегда нулевые, а это значит что эти нули не нужно сортировать, от них ничего не зависит. Ну а зачем их сортировать? Чтобы было как в

учебнике? По научненькому? Щас! Получилось у меня O(n*(k-j)), где j — колличество ничего не значащих битов, т.е ни на что не влияющих. Сначала мне показалось что у меня O(N * k + N + r), но что-то я устал писать о сортировке.

Алгоритмы важны, нужны, полезны, но зачем мучить ими людей? Я вот знаю некоторые из них, и узнал я про них только потому что они были мне нужны, и если бы не велосипедирование на дому, я бы к ним на километр даже с огнеметом не подошел. Зачем запытывать питониста логарифмической кривой? Что он вам сделал? Человек написал код, он работает хорошо, быстро, он понятен (ну или нет), не падает и делает полезную работу, и может быть делает работу так хорошо, насколько позволяет язык, ОС и железо.

Я написал код поразрядной сортировки посмотрев видеолекцию, до этого никогда не слышав об этом алгоритме. Если бы меня спросили про эту сортировку на собеседовании, я бы поплыл почувствовав себя тупицей. Это значит что я не знаю и почему-то считаю свое незнание постыдным, и всё! Больше ничего! Это не значит что я баран, тушкан, снорк или кто-либо еще! Я не знал этой сортировки, но знал кучу всего другого, что делало меня если не программистом, то хорошим кандидатом на его место. Есть задачи в которых алгоритмы наше все, я согласен, например игровые движки, анализ данных, предсказание траектории полета ракеты исходя из ее характеристик, но во всем остальном они зачем нужны? Чтобы людей мучить? Об интеллекте их знание или незнание ничего не говорит, о способности решать проблемы тоже. Тогда о чем говорит? Да и вообще, если я, круглый троечник и серийный прогульщик, въехал в эту муть, то программист с боевого проекта чего не поймет? Все он поймет, когда-нибудь.

O(N * k + N + r) — потому что O-большое отражает алгоритмическую сложность именно для самого худшего варианта входных данных. В худшем случае, например если мы имеем 1000 чисел от 0 до 255, сортировка имеет сложность именно такую, так как ее оптимизация только все усложняет. Но если в массиве случайные числа, или еще лучше маленькие, то оптимизация через объединение чисел масссива уменьшит время выполнения, но делает реализацию чуть сложнее. Самое дурацкое, что это знали еще в начале XX века, а я понял только сейчас.

Рентгеновское зрение

Как-то раз мне понадобилось написать приложение для себя любимого. Писал я программу для шифрования файлов. Да, я знаю что сейчас понабегут крысы с бурундуками и как обкудахчат меня за самодельную криптографию по самое «я не такая», но мне как-то все равно. За свою практику я прочитал столько книг по защите информации, программированию и проектированию, что их число недавно перевалило за 76. Я знаю что делаю, по крайней мере, мне так думается по утрам…

Начинается все с файла. Ну точнее с уникальной строки в файле специального типа, именуемого директорией. Директория это тоже уникальная строка, просто помеченная как директория, в таком-же файле, который так-же именуется директорией, и лежит эта строка еще в одной директории. Утка в яйце, яйцо в гнезде, и так до самого корня древа каталогов.

С именем файла, где-то в недрах файловой системы, ассоциировано число, которое обозначает порядковый номер записи фиксированного размера на диске, в которой хранятся метаданные файла, такие как размер файла, дата создания, редактирования, последнего доступа, а главное порядковый номер начала цепочки записей, в которых хранятся сами данные. Если данных настолько мало, что их можно хранить в одной записи, их хранят в одной записи, не создавая никаких цепочек.

Короче берем файл и читаем из него блок размером 8 МБ. Почему именно 8, объяснять не буду, но на своем HDD я перепробовал разные размеры блока, и 8 МБ просто оказался больше и при этом не медленнее меньших. Если прочитали меньше 8 МБ, наверное это ошибка чтения (тогда смотрим значение возвращаемое ferror), если не ошибка, то мы прочитали весь файл целиком и его размер меньше 8 МБ. Хм, логично.

Чтобы было быстро и без запинок, делаем чтение так (это Си подобный псевдо-код)

```
int    result_code = 0; /* OK */
FILE * file = fopen(filename, PARAM_RONLY);

if (!file) {
  return READ_FILE_NOT_OPEN;
}

while (1) {
  realread = fread(buffer, 1, BUFFER_SIZE, file);

  if (realread < BUFFER_SIZE) {
    if (ferror(file)) {
      result_code = READ_FILE_ERROR;
      break;
    }

    if (0 == realread) break;
  }

  for (nblock = 0; nblock < realread; nblock += vector_size) {
    encrypt(ctx, cipher_ptr, vector, gamma);
    memxor(buffer + nblock, gamma, vector_size);
    memcpy(vector, buffer + nblock, vector_size);
  }

  hmac_sha_2_256(ctx, buffer, realread);

  if (fwrite(buffer, 1, realread, file_output) < realread) {
    result_code = WRITE_FILE_ERROR;
    break;
  }

  fflush(file_output);
  re_keying += realread;

  if (re_keying >= 0x80000000) {
    re_keying = 0;
    internal_re_keying(ctx);
  }
}

fclose(file);

meminit(buffer, 0x00, BUFFER_SIZE);
meminit(vector, 0x00, vector_size);
meminit(gamma,  0x00, vector_size); /* sizeof(vector) == sizeof(gamma) */
meminit(ctx,    0x00, sizeof(GLOBAL_MEMORY));
ctx = NULL;

return result_code;
```

Если вам было страшно читать код выше или вы испытываете чувство тревоги при виде странных закорючек, букв и слов, соединенных в какую-то логическую фигню — вы не одиноки.

Весь код приводить не буду, во избежание звона серпов, чириканья, криков, лая и приступов психоза (не только у читателя). Допустим мы прочитали весь файл. Тогда шифруем данные шифром Rijndael с ключом 256 битов в режиме CFB, и пишем все это безобразие в файл назначения. Используем Rijndael (AES) чтобы не засмеяли и не боялись необычного, ибо Serpent мало кому известен. Режим CFB был выбран ради упрощения архитектуры и его «магии самовосстановления» при порче данных, ну а ключ 256 битов это чтобы битов ключа было побольше.

Файл назначения создается заранее, но его размер не устанавливается в самом начале процесса, как это сделано в торрент-клиентах. Разумеется перед записью в файл нужно спросить у ОС, а хватит ли нам прав на запись и места для записи шифр-текста на диск. А ведь записать-то надо не только шифр-текст, но и вектор инициализации (уникальный для каждого ключа), ключ шифрования (случайный), контрольную сумму шифр-текста, перемешанную с ключом аутентификации (не связанным с ключом шифрования), вычисленную функцией HMAC-SHA-2-256. Так-же нельзя не учесть нагрузку на ключ, и сделать автоматическую смену ключа по достижении определенного количества зашифрованных байтов. Так как защита данных парольная, пароль придется превращать в ключ шифрования функцией формирования ключа, да такой, чтобы была простой, понятной и легкой в программной реализации, а так-же чтобы было невозможно ускорить перебор пароля, распределив вычисления по ангару с видеокартами. И конечно-же придется использовать принципы защитного программирования, например такие как перезапись данных в памяти сразу после точки следования, после которой данные больше точно не понадобятся (пароли, ключи, хэши), обнуление критически важных переменных (размер ключа, указатель на что-нибудь важное) и т.д. Короче проблем куча-могуча, и ее пришлось долго разгребать и конечно-же программировать. Но сначала о самом простом, и под Windows. Бдите, ибо будет больно...

Пытаемся открыть файл на чтение, если не открылся, запрашиваем у операционной системы код ошибки и по нему устанавливаем, что писать пользователю в окошке, озаглавленное как «Ошибка открытия файла». Тут все топорно и даже описывать не хочется. Но вот что делать, если файл открылся, а работать программа должна с файлами чуть-ли не любого размера, да еще начиная с Windows XP SP3 и заканчивая Windows 11? Так-же программа должна обладать скоростью обработки данных настолько высокой, насколько это возможно (ну или близко), быть крошечной и конечно-же никогда не падать. Сам зашифрованный файлик должен быть неприступен для любой мр*зи у которой нет пароля, и так как я параноик страдающий шпиономанией, которому нужно контролировать всё до последнего бита, я писал всё это на Си, используя C++ только там, где писать на Си ну уж очень больно (я имею в виду конкатенацию строк).

Так как Microsoft Windows вроде-как обратно-совместимая, пишем код так, как его писали 20 лет назад

```
fsize_t SizeOfFile(const char * filename) {
  HANDLE fh;
  DWORD low_size;
  DWORD high_size = 0;
  fsize_t result  = -1LL;

  if (filename) {
    fh = CreateFile(filename, GENERIC_READ, FILE_SHARE_READ | FILE_SHARE_WRITE,
```

```
                    0, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, 0);

    if (INVALID_HANDLE_VALUE != fh) {
      low_size = GetFileSize(fh, &high_size);

      if (!GetLastError()) {
        result  = (fsize_t)high_size << 32;
        result += (fsize_t)low_size;
      }

      CloseHandle(fh);
    }
  }

  return result;
}
```

Тип fsize это не тип языка программирования Си или C++, это введенный мной синоним типа off_t, а тип off_t это синоним типа signed int языка Си. Если же переопределить значение символа _FILE_OFFSET_BITS значением 64, то получится синоним типа signed long long. Только вот в разных компиляторах, некоторые вещи, если не отсутствуют, то сделаны немножечко иначе. В C++ Builder 6.0, который был выбран из-за простоты и быстроты формошлепства, высокой скорости компиляции и крошечный размер исполняемого файла собранного проекта (около 890 КБ после статической линковки), никакого символа _FILE_OFFSET_BITS нет. Не беда, делаем синоним нужного типа через #define.

Сразу присваиваем переменной, значение которой будет возвращено функцией, число -1LL, чтобы выполнение кода в случае любой ошибки, проваливалось к строке return result (я считаю что это изящно).

Поскольку это Windows, а Windows это чудо чудесное, пытаемся открыть файл, даже если это и не файл вовсе. И не просто так, а с флагами, запрещающими удаление, запись и чтение всем, кроме нашей программы. Предпоследний флаг WinApi вызова CreateFile я так и не познал. Зачем он нужен я найти не мог, но в документации сказано что он нужен для обычных файлов по умолчанию. Я серьезно, вот так там и написано. Если когда-нибудь захотите открыть документацию к WinApi и поискать описание аргументов функции CreateFileA из файла fileapi.h, запаситесь 75-миллиметровым орудием, огнетушителем и валидолом… Это правда кто-то знает? Вот этот раскатанный через футбольное поле рулон кто-то прочитал, понял за что отвечает каждый параметр и теперь хорошо программирует под Microsoft Windows? Он что, псих?

Дальше проверяем, открылся ли файл. Если нет, плюем на файл (если он этого достоин) и завершаем функцию, вернув -1LL. Нафига он нужен-то если его даже открыть нельзя?

Если открыли, пытаемся (именно пытаемся) узнать его размер в байтах функцией GetFileSize. Если размер почти получили (именно почти), отдаемся страсти настоящего программирования. Так как функция возвращающая размер файла GetFileSize (возвращает она, ага) может вернуть число 0xffffffff, нужно узнать как это число собственно понимать. Для этого нужно вызвать функцию GetLastError (еще одну, да) которая ответит нам, вернула функция GetFileSize ошибку или нет. Если вернула ошибку, то число 0xffffffff это и есть признак ошибки, а значит это точно не младшая часть 64-битного размера файла и можно возвращать -1LL. Но и это не всё! Функция GetFileSize может завершатся с ошибкой вернув число отличное от 0xffffffff (может нет, а может да), и что тогда?

Короче, забыли про число, и просто проверили, завершился ли вызов GetFileSize с ошибкой. Если завершился без ошибок, нагло делаем старшую часть размера файла старшей частью 64-битного возвращаемого значения (даже если там 0), сдвигая её на 32 бита влево, и прибавляем число, которое вернула функция GetFileSize. Адрес переменной, которая играет роль старшей части размера файла, нужно передавать вторым аргументов функции, и если размер файла оказался меньше 4 ГБ, то функция присвоит переменной по ее адресу число 0 (вроде-бы так). Теперь у нас есть 64-битный знаковый размер файла в байтах. Всё, отмучились...

Кто, как и зачем написал вот этот вот весь непонятный и запутанный WinApi? Он что, тоже псих? Почему так-то? Он системные вызовы в Linux видел?

Теперь о криптографической составляющей программы. Вот у нас есть файл, назовем его hot_kats.txt, его нужно зашифровать, да так, чтобы никакая мр*зь не смогра прочитать этот файл без пароля. Начнем с пароля.

Пользователь вводит в программу пароль, пароль отдается на растерзание KDF (функции формирования ключа), функция выплелвывает криптографически стойкий (что бы это не значило) ключ шифрования, сформированный на основе пароля и соли, которая необходима для предотвращения атаки на пароль с предвычисленными хэшами уже известных строк. Соль размером 256 битов мы попросим у ОС используя базовый криптопровайдер PROV_RSA_FULL, которую запишем в итоговый файл одним из блоков.

Пароль и соль отдаем на пытки KDF, получаем ключ, PROFIT! Далее приступаем к шифрованию. Читаем из файла 8 МБ, разбиваем на блоки, и каждый блок шифруем полученным ключом, в качестве первого блока зашифровав разумеется IV (вектор инициализации), который тоже любезно попросим у ОС. Ну а откуда еще? А вот откуда. Выделяем на стеке массив размером 4 КБ, хешируем, и если хеш массива на стеке не равен хешу массива забитого значением 0x00, считаем что хеш мусора из стека у нас хороший (плевать что там было раньше), и ксорим его с IV. Так сказать двойная защита.

Шифровать будем по схеме:

```
k    = KDF(password, salt);
IV   = sha256sum(stack_trash_4kb) xor OSRandomGen();
C[0] = IV;
C[i] = P[i] xor Ek(C[i-1]); от i=1 до i = n; где n = filesize/blocksize;
```

Защищать файл от подмены будем по схеме:

```
HMAC = h(k xor 0x5C || h(k xor 0x36 || P));
```

Но не все так просто. Тут мы натыкаемся на первый подводный камень размером с дом. Допустим зашифровали файл в 10 разных зашифрованных файлов, с паролем «PasswordingNamording» и отослали его 10 разным людям, т.е каждый человек получил свой уникальный файл, полученный из файла hot_kats.txt. Каждый кто имеет пароль, может расшифровать данные и получить исходный файл. Но какая-то мр*зь тоже хочет знать что это за файл и разумеется отслеживает ВСЕ каналы передачи данных, особенно те, которые редко используются (вдруг они для особых случаев?). Нашу мр*зь интересует всё! Совсем! Кто, кому, когда, просто так или в связи с чем-то, как долго, в каком объеме и с какой периодичностью посылает. Ну натуральныя же мр*зь! Она знает криптосистему, так как ее разбирали люди знающие C, C++, C#, asm x86_64, OllyDbg, IDA Pro и вообще у них есть мозги и они читали Шнайера с Фюргесоном. Мр*зь знает что при использовании одного

пароля и разных IV, генерация гаммы блочным шифром будет давать разные гаммы. Даже если использовать один и тот-же пароль для защиты миллионов копий одного и того-же файла, в зашифрованном виде эти файлы будут радикально отличатся, так как будут зашифрованы разными гаммами. Но, при использовании ключа шифрования в качестве ключа аутентификации, HMAC всех таких файлов будет одинаковым. Т.е вы зашифровали файл паролем во множество радикально отличающихся зашифрованных файлов, разослали по списку контактов, каждый получил свой экземпляр файла. Каждый контакт владея паролем может расшифровать файл и сделать с ним что пожелает, но мр*зь уже знает не только кто кому что послал, но так-же страшную вешь, а именно: все пользователи получили одинаковый файл, и следует это из идентичности HMAC, даже если шифротексты разные. Если файл зашифровать тем-же паролем и отослать повторно, мр*зь будет знать, что это повторная передача одного и того-же файла, даже если благодаря разному IV, зашифрованные данные каждый раз разные. Контрольные суммы файлов будут отличатся, как и их размеры, но мр*зь будет знать, что это один и тот же файл. Понимаете? Мр*зь даже никакую атаку не стала проводить, недостаток защиты метаданных даже не при передаче, а при защите самих данных уже дал мр*зи важную информацию, которая может стоить людим жизни. Например, если файл это электронная копия приказа начальника штаба дивизии, то получение подчиненными разных зашифрованных файлов с одинаковым HMAC, дает мр*зи понять что это один и тот-же приказ. Дальше, как всегда по обстановке.

Так что хешировать нужно не открытый текст как могло показаться раньше (мы же файл защищаем), а шифротекст, а если это проблематично, использовать IV и уникальный ключ аутентификации, причем разные для каждого акта шифрования. Лучше всего защищать контрольную сумму зашифрованных данных и структуры метаданных открытого текста. Только так можно гарантировать что при шифровании одинаковых файлов одним и тем-же ключом, шифротексты как и HMAC-и будут отличатся, что не позволит установить, что это за данные и например, передавались ли они раньше. Ну что-же, делаем вот так

```
HMAC = h(k xor 0x5C || h(k xor 0x36 || h(SP || C[0..n-1])));
```

и живем спокойно.

Дальше хуже, т.е дальше у нас еще один, и даже не камень, а риф! И пьяной лисе понятно, что использовать режим ECB можно только в качестве демонстрации того, как не надо делать. Режимы с аутентификацией я не рассматривал из-за их относительной новизны и сложности программирования, а старые но простые режимы требуют дополнения, что усложняет архитектуру приложения. Допустим я бы использовал режим CBC, он обычно описывается вот такой схемой:

```
C[0] = IV;
C[i] = Ek(P[i] xor C[i-1]);
```

если длина последнего блока открытого текста меньше размера блока шифра, блок данных придется дополнить до размера блока какими-то данными. Допустим я дополнил бы конец блока байтом

```
byte =  (unsigned char)(BLOCK_SIZE-datalen);
for (i = datalen; i < BLOCK_SIZE; i++)
  block[i] = byte;
```

получилось бы допустим так

```
00 01 02 03 04 05 06 07 08 09 0A 05 05 05 05 05
```

или еще как нибудь. Как это обработать? Как я узнаю, что байт 0x05 это именно дополнение а не часть открытого текста? Может быть мне дамп памяти прислали? Можно конечно хранить размер дополнения зашифрованным в начале файла, но это усложнение архитектуры программы, а я хочу сделать максимум минимальными средствами. Это должно быть просто и чтобы хрен взломали!

Не долго думая я выбрал CFB, он у нас вот такой:

```
C[i] = IV;
C[i] = P[i] xor Ek(C[i-1]);
```

Шифрование в этом режиме позволяет выкинуть из кода функцию расшифровки, не думать о дополнении, реализовать HMAC самому ни на кого не полагаясь (вспомните все те дыры в openssl) и конечно-же получать шифротексты размером с открытые тексты, экономя место (спорно, признаю).

Зашифровали файл в режиме CFB, приклеили в начало IV, в конец HMAC, и отправили файл. Мы защищены, и никто не сможет прочитать содержимое.

Ага, щас! - с усмешкой закричала математика.

Каждый блочный шифр используемый в качестве генератора гаммы обладает свойством, которое я называю повторимость. Это когда вы шифруете ключом какие-то данные, например в режиме OFB, и рано или поздно настает такой момент, когда очередная получившаяся гамма вдруг оказывается равна первой, второй, третьей т.д. И что делать? Первое что пришло в голову, использовать другой ключ. А получать его можно шифрованием какой нибудь константы, например счетчика по следующей схеме:

```
x = 0;
while {
  k = Ek(x);

  C[0] = IV;
  C[i] = P[i] xor Ek(C[i-1]); от i=1 до i=n;

  x = x + 1;
}
```

где n — максимальное число блоков которое можно зашифровать одним ключом. Для блочного шифра с шириной блока 128 битов, я взял консервативную нагрузку на ключ 2 ГБ, так как очень боюсь шпионов и всяких вуаеристов. Работает схема так. Шифруем ключом массив нулей, получая ключ шифрования данных, шифруем этим ключом первые 2 ГБ, увеличиваем счетчик с 0 до 1, шифруем увеличенный на 1 массив используя в качестве ключа шифрования предыдущий ключ и получаем ключ для шифрования следующих 2 ГБ данных. Итого: ключ шифрования, полученный из пароля вообще не используется в качестве ключа шифрования данных, а используется в качестве одного из двух зёрен генератора ключей, по одному ключу на каждые 2 ГБ данных. В RFC8645 написали конечно немного иначе, но меня пока все устраивает.

В процессе проектирования, программа обросла плюшками, колокольчиками и гирляндами, я даже прикрутил к программе возможность парить самописный конфигурационный файл, который полностью продумал сам, так как не хотел привязывать программу к библиотеке .ini

и тем более какому-нибудь XML. Как оказалось, прикрутить самописный конфиг к программе, это не просто а очень просто. Свой конфиг, это же круто? Нет? Ну и ладно…

Писал я код этой программы несколько месяцев, как всегда психовал из-за тупых ошибок, опечаток, проваленных тестовых обработок данных, криво переведенной документации к WinApi, непонятных Access violation и прочих прелестей моего ремесла. Ну чем не зимняя романтика?

Спустя множество бессонных ночей, пожертвовав десятками тысяч нервных клеток утерянных без возможности восстановления, после сотни раз перечитанного и причесанного кода, я попробовал программу в деле и она заработала как атомные часы (в смысле надо молотком заехать, чтобы треснуло). Ну я и расслабился. Спустя время я решил заменить оператор выбора if в одной из функций на оператор switch, так как второй мне больше нравится, ну я и переписал код по новому. Вы наверное уже догадались, что не всё и всегда идет по маслу и каждому человеку в этой жизни придется принять, что не все желания исполняются?

Найдите ошибку за 100 попыток

```
switch(ctx->cipher) {
  AES:
    rijndael_ctx = (uint32_t *)cipher_ptr;
    rijndael_key_encrypt_init(rijndael_ctx, ctx→real_key,
                                            ctx->real_key_length*8);
    break;

  BLOWFISH:
    blowfish_ctx = (BLOWFISH_CTX *)cipher_ptr;
    blowfish_init(blowfish_ctx, ctx->real_key, ctx->real_key_length);
    break;

  SERPENT:
    serpent_ctx = (SERPENT_CTX *)cipher_ptr;
    serpent_init(serpent_ctx, ctx->real_key_length * 8, ctx->real_key);
    break;

  THREEFISH:
    threefish_ctx = (THREEFISH_CTX *)cipher_ptr;
    threefish_init(threefish_ctx, (threefishkeysize_t)(ctx->real_key_length*8),
                   (uint64_t *)ctx->real_key, (uint64_t *)ctx->real_key);
    break;

  TWOFISH:
    twofish_ctx = (TWOFISH_CTX *)cipher_ptr;
    twofish_init(twofish_ctx, ctx->real_key, ctx->real_key_length);
    break;
}
```

Если нашли, хорошо, вот вам +1 к опыту. Я же когда это писал, видимо был совсем не там, где мой компьютер, и посчитав что в коде все хорошо, собрал проект. C++ Builder 6.0 тоже ничего плохого не заметил, но в этом-то вся и соль! При первом же запуске, программа упала с грохотом, искрами, конфетти и красным окошком ошибки. Ну я долго матерился, когда как-будто по велению гигантского голубого камня, МОЯ ПРОГРАММА стала падать с ошибкой, довольно позорной для моего, как мне казалось, не маленького уровня

Инструкция по адресу "0x0040CD8A" обратилась к памяти по адресу "0x00000000". Память не может быть "read"

Попсиховав минуточки две-три, я прикинул и смекнул. Так, вот у меня код обращается по адресу 0, читает по этому адресу что-то там, а так как это операция чтения по адресу 0, ОС увидев такую наглость, решила прибить процесс и прикрикнуть на меня знаменитым и незабываемым красным окошком. Что-же может быть не так? Вспомнив архитектуру кода проекта, я стал думать мысли. Каждое динамическое выделение памяти происходит в самом начале работы программы (чтобы потом память просить не пришлось), и обязательно проверяет указатель на 0, и если там 0, завершается, обязательно пытаясь уведомить пользователя о том, что памяти его компьютера не хватило для работы моей драгоценной и безусловно нужной программы. Последние изменения в код программы, которые касаются указателей, я вносил в функцию инициализации структур данных, которые нужны для работы алгоритмов шифрования (код выше), а так как все указатели во всем проекте, имеют присвоенное им корректное значение, ранее возвращенное функцией malloc(), а возвращаемое ей значение всегда проверяется на 0, всё должно быть хорошо. Ведь так?

Думание мыслей завело меня в тупик, так как такого просто не может быть. Понимаете? Случилось то, чего случится не могло. Совсем! Вот у нас указатель, не важно какое значение он хранит перед присваиванием ему значения другого указателя, ведь следствием присваивания всегда является исчезновение старого значения и возникновение на его месте нового, того, которое присваивали. Присвоили мы допустим переменной rijndael_ctx значение переменной cipher_ptr, сама же переменная cipher_ptr хранит в себе реальный адрес, т.е такой, который во первых не равен 0, а во вторых который функция malloc() успешно запросила у ОС, и ОС любезно ответила, что можно этим адресом пользоваться, так как памяти хватает. Что возвращает функция malloc() или любая другая функция, которая может, пусть даже гипотетически, вернуть плохое значение, проверяется всегда (таков неписанный закон олдов), и если возвращается не 0 (хорошее значение), то этот самый «не ноль» присваивается переменной cipher_ptr. Переменная cipher_ptr хранит значение пока программа не будет завершена пользователем, и только тогда память освобождается функцией free() и переменной cipher_ptr нагло присваивается NULL. Все должно работать. Так ведь? А какого черта спрашивается программа падает? И главное, где именно? Что за колдовство?

Разочаровавшись в себе и отчаявшись, я вызвал тяжелую артиллерию — отладчик. Первое же всплытие отладчика указало мне на инструкции

```
mov eax, [ebp+124]
mov ebx, [eax]
```

приглядевшись, я заметил, что значение переменной на стеке, которое копируется в регистр eax всегда равно 0. По этому адресу программа что-то читает в регистр ebx, а так как по адресу 0 ничего читать нельзя, ОС совершенно законно убивает процесс. Хм, неужели так было всегда, а программа не падала только потому что мне все это время везло? Нет, такого просто не может быть!

Попробовав разные шифры в настройках программы, я с удивлением обнаружил, что в зависимости от выбранного шифра, программа падает в разных местах. Т.е разные подпрограммы обращаются к 0 адресу, и программа падает то с ошибкой чтения, то с ошибкой записи. Присмотревшись к стеку, я прикинул, что там такого у меня лежит, что оно всегда равно 0. Обнаружив на стеке 5 глобальных переменных адресного типа, я посмотрел на код в сто первый раз и не нашел ничего, что могло бы приводить к падению. Тут я начал думать, что второй раз сошел с ума.

Спустя два дня самобичевания и отдыха от компьютера, я листал свой старый Delphi код и заметил, что оператор case выбирает какой код выполнить по константному значению, и

перед значением нет никакого слова case. С подозрением открыв код проекта, который так потрепал мне нервы, я увидел оператор switch языка Си, и тоже не увидел слова case. Приписав слово case перед каждым константным значением, я тыкнул на кнопку Build, код собрался и больше никогда не падал. Занавес!

Компилятор мр*зь, даже предупреждения не выдал, и прочитал код так, что первая строчка после каждого константного значения оператора выбора switch не попала на анализ компилятору, т.е компилятор проигнорировал всё, начиная от конца всех константных значений до точек с запятой. Нет, ну вы понимаете? Проигнорировал!

Да признаю, я перепутал два языка, Delphi и C++, и написал код для C++ компилятора с оператором выбора switch, думая о нем как об операторе case из Delphi. Знаю что позорище, и знаю что использовать такой устаревший софт для компиляции проекта это вот прям непростительно, но компилятор должен был увидеть отсутствие слова case перед константным значением, так как это ключевое слово оператора языка блин!

Позже мне стало интересно, а насколько моя программа защищена от дурака. Я изначально не хотел связываться с многопоточкой, поэтому я ее имитировал. Вызываем простую функцию прямо из обработчика, функция запускается и обработчик не завершится, пока функция не вернет управление. Но ничто не мешает второй раз вызвать тот-же обработчик, и в его начале проверить значение глобальной переменной, и если переменная установлена в true, то спросить пользователя, не желает ли он прервать процесс установив значение в false. Функция обработки файла на каждой итерации проверяет этот флаг а так-же постоянно будит окошко вызовом Application->ProcessMessages() во избежание зависания, и завершается как только обнаружит во флаге false. Если процесс обработки уже начат, его можно прервать вот так:

```
  EnterCriticalSection(&Form1->CrSec);
  if (PROCESSING) {
    if (MessageForUser(MB_ICONQUESTION + MB_YESNO, STR_PROGRAMM_NAME,
STR_QUESTION_BREAK) == IDYES) {
      PROCESSING = false;
    }
    LeaveCriticalSection(&Form1->CrSec);
    return;
  }
  LeaveCriticalSection(&Form1->CrSec);
```

Я открыл программу, запустил шифрование файла размером 3.92 ГБ, в процессе шифрования навел курсор мыши на иконку приложения на панели задач Windows и жмакнув по правой кнопке мыши, выбрал "Закрыть окно". Программа грохнулась с ошибкой "Access violation" чем очень меня обидела. Ну думаю ладно, черт с тобой, пора перенести тебя на многопоточку. Вы уже в предвкушении, да, мои маленькие любители жестокости?

Я пошел самым топорным путем (другого и нет, ну кроме голово WinApi) и решил запускать шифрование/расшифровку в отдельном потоке, отдав поток на растерзание ОС. Если программу убьет пользователь, то ОС должна позаботится о закрытии потока и освобождении ресурсов. По крайней мере я хочу в это верить. Итак, программа запускается, пользователь вводит данные и жмет на "Самую Главную Кнопку". Обработчик события (нажатия кнопки) проверяет данные в полях формы, и если они корректны, создает объект класса TThread, но в замороженном состоянии, если нет, показывает окошко с желтым восклицательным знаком и текстом, куда смотреть и где исправлять свои каракули. Деактивированный объект класса TThread означает, что он начинает свое существование как хранилище данных, но ничего не делает, пока не будет вызван метот Execute(). Для метода

Execute() я выбрал задать только приоритет потока и уничтожение объекта по завершении метода. Приоритет потока при выполнении кода метода Execute() задается переменной Priority, завершение потока сразу после выхода из метода задается переменной FreeOnTerminate. Если FreeOnTerminate = true, то объект исчезнет сразу после завершения метода Execute(), т.е не надо убивать его через delete. Преждевременно попытаться (именно так, да) завершить метод Execute() можно методом Terminate(), который устанавливает значение переменной Terminated в true. Значение этой переменной придется проверять вручную внутри метода Execute(), и если при проверке значение окажется true, это будет означать что кто-то другой передал потоку значение true. Если стало true, то можно сделать return и это приведет не только к завершению кода метода Execute() но и уничтожению объекта. Я ничего не буду писать про то, что делать если метод Execute() повиснет в блокирующем вызове fread(), особенно если начались проблемы с диском и в логе ОС вы видите "Обнаружена ошибка на устройстве \Device\Harddisk1 во время выполнения операции страничного обмена". Это уже не ко мне а к друидам. Но чтобы был хоть какой-то шанс сообщить пользователю что все пошло не так как он ожидал, возвращаемые значения функций fread() и fwrite() проверяются всегда, как и ferror(), ибо таков неписаный закон олдов.

Четыре дня я проектировал класс, тестировал работу потока, грохал приложение через диспетчер задач, и все было хорошо. На пятый день я последним штрихом прикрутил к программе чтение ключа шифрования из файла а так-же генерацию ключа из пароля. Вот код:

```
  if (FileExists(MemoKey->Text) == true) {
    if (thread->ReadKeyFile(MemoKey->Text.c_str()) == false) {
      delete thread;
      return;
    }
  }
  else {
    if (thread->GeneratingCryptKey("Сгенерировать ключ из пароля?") == false) {
      delete thread; // FUCKING BORLAND!!! WHY DOUBLE FREE OBJECT???
      return;
    }

    if (thread->KeyGenFromPass(MemoKey->Text.c_str()) == false) {
      delete thread;
      return;
    }
  }
```

Что вы думаете об этом участке обработчика нажатия кнопки? Нет, это вопрос риторический, тут и так все ясно. Тут нет ошибок, код вроде нормально читается, понятно что делается (мне точно понятно), НО! Если возвращаемое методом GeneratingCryptKey значение равно false, т.е если пользователь увидел окно с вопросом и нажал на кнопку "Нет", происходит уничтожение объекта. Ровно один раз. Один! Там так написано! Я сам это писал! C++ Builder 6 так не посчитал, и сгенерировал код который вызывал деструктор два раза. Вот так. Написано сделать один раз, а компилятор делает два.

Прошлые уроки научили меня, что психику надо беречь, а криками истерички делу точно не помочь, и я стал думать. Вот у меня есть объект, он пока заморожен, но может хранить данные. Запускается поток тогда, когда я сделаю thread->Resume(), а завершается либо сам, либо когда я грохну его вызовом Terminate(). Метод GeneratingCryptKey вызывает Application->MessageBox() спрашивая пользователя, не желает ли он сгенерировать ключ шифрования из пароля. Если желает, продолжаем дальше, если нет, убиваем объект. Убиваем

его один раз, а у меня убивается два раза. Деструктор первый раз отрабатывает нормально, второй (которого быть не должно) отрабатывает с глюком, намертво вешая программу. В других местах где объект уничтожается такого не проиходит. Почему? Поцелуй меня динозавр если знаю!

Три дня я искал ошибку... И не нашел, потому что ее там нет. Проблема решилась спонтанным перенесением присваивания переменным FreeOnTerminate и Priority их значений из конструктора в начало метода Execute(). Т.е не надо было присваивать переменным потока что-либо до запуска потока методом Execute(), так как это плохо влияет на что-то там и приводит к странному и даже опасному поведению программы, которое объяснить даже Интернет оказался не в состоянии. Программа заработала, но я чувствовал что что-то не так. Что это было-то? Поиск в интернете результатов не дал, мануалы молчали, а я ходил по комнате и не мог смирится с тем, что я так и не понял что это было. Было чувство какой-то заставшей меня врасплох магии, саморазгадавшейся загадки, невидимого ублюдка давшего мне подзатыльник, которого я так и не поймал. Сволочь конечно убежала и больше не возвращалась, но это странное чувство не дает мне покоя до сих пор. Это напомнило мне горький опыт откручивания от WinHex функции проверки подлинности лицензии. Я пытался, правда пытался, но не смог.

Раньше у меня не было «рентгеновского зрения», а теперь, как я обнаружил, его и сейчас нет. Если я вижу косяк, то почти всегда знаю, что пошло не так, а если и не знаю, то рано или поздно обязательно найду. Все благодаря выстраданному опыту, и страдания никогда не закончатся, а опыт будет расти сам, всю мою жизнь. Я почти всегда виноват в том, что в программе что-то работает не так как ожидалось, но я не несу ответственность за всё. Я именно что почти всегда виноват, ибо не я один на свете криворучка.

Пока вы читали эту писанину, написанную в полубреду, вы могли не заметить, как в тексте я коснулся довольно таки большого количества всякого разного и сложного. Мы задели следующие темы.

0.) Целые 32-битные числа, операции над такими числами, ввод и вывод, системы счисления, операторы ветвления и выбора, циклы конечные и бесконечные, с постусловием и предусловием, оператор прерывания цикла.
1.) Переменные и типы данных, подпрограммы, подпрограммы возвращающие данные.
2.) Массивы, строки, индексация и указатели.
3.) Структуры данных, сортировки, списки, связки структур данных.
4.) Языки программирования и работа программ «под капотом».
5.) Операционные системы и программирование с учетом капризов их разработчиков, статическое и динамическое выделение памяти.
6.) Архитектура программ, криптография, декомпозиция.
7.) Многопоточная архитектура, ее имитация с помощью хаков, критические секции по данным.
8.) Отладка программы - обнаружение ошибки, переживание острой реакции на ошибку, поиск причины возникновения ошибки, страдания, поиск виноватых, отчаяние и устранение ошибки.

Список получился хоть и маленький, но уже этого оказалось достаточно, чтобы бросится в огонь безудержного рассуждательства о программировании. Если-бы я описывал процесс разработки в подробностях, т.е какие проблемы решал, чему удивлялся и от чего бесился, какие книги читал, какие форумы листал, сколько непростреливаемого бреда перелопатил, пытаясь вычленить хоть что-то разумное и полезное, у меня бы получилась целая книга. Каждый, кто выработал так называемый «взгляд сверху», уже это знает по себе.

Взгляд сверху — это когда вы способны посмотреть на программистский «театр военных действий» как-бы со стороны пилота разведывательного самолета, оценить что творится на земле и сказать где отступать, где наступать а где сразу бомбить, потому что там сам чёрт в ужасе трясется. Такой взгляд есть только у того, кто прошел через Ад проблем и их решений, больше нигде такого опыта не приобрести. Он выработается сам, со временем, а если время было потрачено на чтение таких вот статей а не на решение проблем (компьютер зависает, YouTube не показывает) то никакого опыта и взгляда сверху у вас конечно-же не будет. Откуда он возьмется-то?

То что я понаписал, само собой разумеется для любого программиста средней руки, про сеньоров я умолчу.

Когда я учился (сам) играть на гитаре, то до меня дошло, что результат выйдет круче, если комбинировать разные аккорды так, чтобы брать то легкий, то сложный аккорд, но вразнобой. Как мне сказали, опытные гитаристы потратили около года на то, что-бы научится уверенно брать из любого положения кисти, как малое так и большое баррэ. Подобное «выпрямление левой руки вдоль грифа», требует времени, большого упорства, желания играть а так-же умения терпеть боль в кончиках пальцев. Гитарку я так и не освоил (стоит теперь пыльная в углу), но понял, что есть вещи, которые можно только «выправить вдоль грифа». То же самое вам подтвердит любой профессиональный ремесленник, будь это хоть кожевник из XVIII века.

Проблема «поргомистишек-входильщиков» как раз в том, что их мозги никогда не были «выправлены об компьютер». Их умы никогда не спотыкались о казалось бы простые вещи, которые вдруг начинают почему-то работать не так, как от них ожидали. На этом их можно валить пачками, так как они просто не умеют решать проблемы, потому что не выстрадали эти проблемы. Оно и понятно, они их и не решали никогда, за них это сделали разработчики программных средств разработки, ну те самые, которые ученые-информатики из MIT. Дернуть метод и дурак может, и это не плохо, но вот сконструировать класс так, чтобы метод дергался без психования и искания виноватых со стороны дергающих, может пожалуй только профессиональный программист.

Должен признаться. Я неотдыхаемо устал. Устал читать всякую муть про вайтишку, про HR-ов (вахтеры плять), про то как кто-то там что-то куда-то прикрутил и теперь вы увеличили что-то там затем что-бы что-нибудь. И все это в [название любой новомодной фигни, которую забудут через год-другой]. Я устал. Где жесткачи? Мясное чтиво? Где те самые бородачи-трюкачи что под Windows 2000 в SoftICE ломали софт просто от скуки? Неужели поумирали? Куда подевалась крутотень? Помните статью про ошибку 217, которая вам ничего не скажет? А статью про самую нужную программу на свете? Практику программирования Кернигана и Пайка читали? Ну был же запал! Где все те, на ком держится вообще-то вся индустрия? У кого я могу учиться? Я устал. Куда всё ушло…

Мне нужна была программа и я написал программу. Простую как кованый молоток, крошечную, быструю, безопасную, потому что это хорошо. Пользователь запускает ее, вводит данные, тыдыкает по кнопке, немного ждет и видит окошко в котором написано, что все прошло успешно. Если нет, программа обязательно скажет, что пошло не так. Всё! Его проблема решена, или он видит текст с намеком, куда смотреть и что делать, чтобы в следующий раз всё уж точно было так как он хочет. Компьютер выполнил свою работу, сделал то, ради чего создавался инженерами — прочитал данные, изнахратил их как-то там, записал куда надо. Сколько читать, как именно их изнахратить и куда писать, позволено решать мне — программисту. Я порешал, и так порешал, что теперь единственное что

заставляет пользователя ждать окончания операции, это чтение с диска и запись на него. На моем железе это около 148 МБ/сек, так как я немного побаиваюсь SSD.

Я люблю и ненавижу компьютеры. Люблю и ненавижу программирование. Люблю и ненавижу решать проблемы. Я ничего не выбирал, не проходил тестов на профориентацию, я тут вообще залётный, которому почему-то зашло. Мне хорошо и плохо, ведь я погромистишка, т.е программист. Но я играю эту роль, как две сестры, любовь и боль, живут во мне необъяснимо...