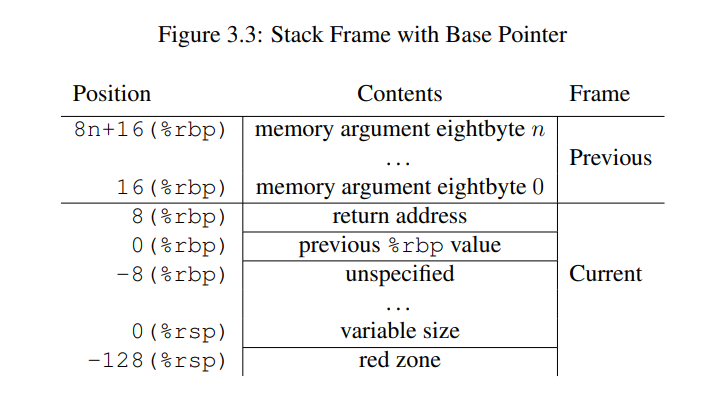
**Лекция 17**



Стек делится на фреймы. **Фрейм** — кусочек стека, который соответствует локальной области видимости одной функции. У фрейма есть base pointer и верхушка стека (адрес, с которого он начинается). Стек растет вниз.

rbp указывает на начало базы текущего фрейма, rsp указывает на верхушку стека.

Когда заходим в функцию, делаем push rbp — кладем на стек rbp (rsp автоматически сдвигается). При заходе в функцию процессор сохраняет старое значение rbp на стеке (которое указывает на начало предыдущего фрейма).

Затем mov rbp rsp — новое значение rbp равно текущему значению rsp (верхушке стека на текущий момент).

Также на стек мы кладем адрес возврата — адрес функции, на которую нужно прыгнуть, после того как вернемся из текущей функции.

**call** — инструкция, которая совершает вызов функции. Передаем в него адрес функции. Отличие от jmp в том, что call автоматически сохраняет адрес следующей инструкции (rip) на стек, и только после этого прыгает на новую функцию.

**ret** — выход из функции. Берет адрес возврата со стека (он лежит на верхушке стека, последние 8 байт), сдвигает rsp назад, и эти 8 байт и есть тот адрес, куда затем идем исполняться.

**Зачем нам сразу два поинтера?** Таким образом мы сохраняем все начала фреймов, и это позволяет нам смотреть stack trace. %rbp на стеке на самом деле образовывают некоторое подобие списка, по которому можно идти и узнавать о глубине вызовов и даже о функциях, ведь по пути у нас будут лежать адреса возвратов рядом с лежащими base pointers.

**-fomit-frame-pointer** (включается автоматически при -O1) – базовая оптимизация, которая отключает механизм base pointers, что приводит к отсутствию возможностей, предоставленных выше.

rsp в целях оптимизации может не всегда сдвигаться вниз (например, когда завели в функции мало переменных, ассемблер может не сдвигать rsp). Но если завести большой массив, например, можно увидеть сдвиг rsp посредством *sub rsp, 0xf30*

На самом деле компилятору разрешается шагать за верхушку стека без ee сдвига на некоторое небольшое количество (около 128) байт (red zone на рисунке выше).

**Передача аргументов в функцию:**

Куда складываются аргументы?

|  |  |
| --- | --- |
| void f(int p, int q) {  int a = 1, b = 1, c = 1;  }  int main() {  f(5, 8);  } | f(int, int):  push rbp  mov rbp, rsp  mov DWORD PTR [rbp-20], edi  mov DWORD PTR [rbp-24], esi  mov DWORD PTR [rbp-4], 1  mov DWORD PTR [rbp-8], 1  mov DWORD PTR [rbp-12], 1  nop  pop rbp  ret  main:  push rbp  mov rbp, rsp  mov esi, 8  mov edi, 5  call f(int, int)  mov eax, 0  pop rbp  ret |

rdi, rsi (здесь edi и esi) — регистры, которые используются для передачи аргументов в функцию.

Если аргументов слишком много, регистров перестанет хватать и тогда будет использоваться стек для хранения аргументов. Аргументы положатся на стек перед адресом возврата. Аналогично с возвращаемыми из функций значениями.

Обращаться к регистрам процессора быстрее, чем считывать данные из памяти.

**register** — ключевое слово в C, которое указывает на то, что данную переменную необходимо положить в регистр, а не на стек (register int x = 1). Удобно для счетчиков цикла. Удалено из C++17.

**Как выглядит динамическая память?**

|  |  |
| --- | --- |
| void g() {  int y = 5;  int\* p = new int;  \*p = 3;  } | push rbp  mov rbp, rsp  sub rsp, 16  mov DWORD PTR [rbp-4], 5  mov edi, 4  call operator new(unsigned long)  mov QWORD PTR [rbp-16], rax  mov rax, QWORD PTR [rbp-16]  mov DWORD PTR [rax], 3  nop  leave  ret |

<https://www.agner.org/optimize/instruction_tables.pdf> — этот документ содержит информацию об условной стоимости (в единицах μops - micro operation,

некоторая примитивная операция) каждой ассемблерной инструкции. Например в этом файле можно узнать, что операция деления – довольно дорогая операция по отношению к другим.

**Уязвимость переполнения буфера:**

Когда в коде делаем ввод какого-нибудь набора символов, и этот набор символов является массивом на стеке, может произойти проблема. Scanf не проверяет, сколько символов было введено, тогда что будет, если пользователь введет символов больше, чем нужно, причем настолько, что перезатрет вышележащие по стеку сущности, в частности адрес возврата? Если ввести специальные бинарные символы так, что адрес возврата будет валидным адресом, указывающим куда-то в другое место, тогда процессор, выполнив ret, заставит программу исполнять на этом месте код, который может быть вредоносным.

**-fstack-protector** – флаг компиляции, который заставляет компилятор сгенерировать специальный код в функцию, который проверяет, не был ли перезатерт адрес возврата, прежде чем делать return из функции.

Компилятор берет некоторое рандомное число, и кладет его на стек сразу после rbp, а еще кладет его в rax. Перед тем как выйти из функции, соответствует ли адрес возврата тому числу, которое было сохранено ранее в rax. Если нет, попадаем в функцию \_\_stack\_chk\_fail.

|  |  |
| --- | --- |
| void f(int p, int q) {  char buf[10];  scanf("%s", buf);  } | .LC0:  .string "%s"  f(int, int):  push rbp  mov rbp, rsp  sub rsp, 48  mov DWORD PTR [rbp-36], edi  mov DWORD PTR [rbp-40], esi  mov rax, QWORD PTR fs:40  mov QWORD PTR [rbp-8], rax  xor eax, eax  lea rax, [rbp-18]  mov rsi, rax  mov edi, OFFSET FLAT:.LC0  mov eax, 0  call \_\_isoc99\_scanf  nop  mov rax, QWORD PTR [rbp-8]  sub rax, QWORD PTR fs:40  je .L2  call \_\_stack\_chk\_fail  .L2:  leave  ret |

Если запустить этот код в терминале, получим ошибку:

\*\*\* stack smashing detected \*\*\*: terminated

Aborted (core dumped)

**-fno-stack-protector** — флаг компиляции, который отключает защиту от перезатирания адреса возврата из функции. С этим флагом получим segfault.

**Intrinsincs**

**Intrinsincs** — cпециальные инструкции, который позволяют делать высокопроизводительные вычисления над большими массивами чисел.

Они позволяют за раз обрабатывать числа высокой разрядности вплоть до 512 бит.

Например, посчитать скалярное произведение векторов, побайтово просуммировать два длинных массива чисел.

По этой причине strcmp работает медленнее memcmp — первый сравнивает по 1 байту, а второй пачками по 8 байт. Также memcpy: за один раз копирует 8 байт, а strcpy по 1 байту. А для процессора скопировать 1 байт и 8 байт по сложности практически равноценные действия.

На современных процессорах есть инструкции, которые позволяют обрабатывать за один раз многобайтовые числа.

Эти инструкции не прописаны в стандарте, но могут быть предусмотрены конкретными компиляторами или процессорами.

**SSE** — streaming SIMD extensions — набор инструкций для повышения производительности в мультимедийных приложениях. Целью набора инструкций было быстро передать некоторые данные (например, часть DVD для декодирования), быстро обработать их (используя SIMD), а затем передать результат на вывод (например, в графическую оперативную память).

**Пример кода:** скалярное произведение векторов

#include <stdio.h>  
#include <emmintrin.h>  
  
#define ARRAY\_SIZE 8  
  
int main() {  
 float a[ARRAY\_SIZE] = {1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0};  
 float b[ARRAY\_SIZE] = {8.0, 7.0, 6.0, 5.0, 4.0, 3.0, 2.0, 1.0};  
  
 \_\_m128 sum = \_mm\_setzero\_ps();  
 for (int i = 0; i < ARRAY\_SIZE; i += 4) {  
 \_\_m128 a\_vec = \_mm\_loadu\_ps(&a[i]); // 4 float из массива a  
 \_\_m128 b\_vec = \_mm\_loadu\_ps(&b[i]); // 4 float из массива b  
 \_\_m128 prod = \_mm\_mul\_ps(a\_vec, b\_vec); // перемножение 4-х пар float  
 sum = \_mm\_add\_ps(sum, prod); // сложение получившихся этих 4 чисел  
 }  
  
 float result[4];  
 \_mm\_storeu\_ps(result, sum);  
  
 float scalar\_product = result[0] + result[1] + result[2] + result[3];  
 printf("The scalar product is %f\n", scalar\_product);  
}

Если дизассемблировать бинарный код, в main будут не обычные инструкции типа mov или add, а movss, movups, movaps и addss и т.п.

Иногда компиляторы сами умеют векторизовать, например, когда ставим флаг оптимизации -O3.

Если хотим использовать векторные инструкции, то нужно, чтобы объекты были выровнены (оператор alignof и спецификатор alignas для этого).

Начиная с C++17 в стандартной библиотеке появились версии стандартных алгоритмов, включающие в себя векторизацию. Например, std::copy. Для того, чтобы влиять на выполнение этой функции, мы должны передать туда определенный ExecutionPolicy. Он бывает разных типов: sequenced\_policy (обычное привычное выполнение), parallel\_policy (распараллеливание процесса на разные потоки), parallel\_unsequenced\_policy и unsequenced\_policy (векторизация).

С C++26 в стандарт хотят добавить std::sse :)