Немного об отладке. Часть 1

<https://habr.com/ru/articles/867478/>

**Средний**

**127 мин**

**7.6K**

[Отладка\*](https://habr.com/ru/hubs/debug/)[Системное программирование\*](https://habr.com/ru/hubs/system_programming/)[Linux\*](https://habr.com/ru/hubs/linux_dev/)[Программирование\*](https://habr.com/ru/hubs/programming/)

[✏️ Технотекст 7](https://habr.com/ru/technotext/2024/)

Отладка кода вдвое сложнее, чем его написание. Так что если вы пишете код настолько умно, насколько можете, то вы по определению недостаточно сообразительны, чтобы его отлаживать.

Брайан Керниган

Приветствую.

Вы пользовались отладчиком? Если нет, то дальше читать не стоит. Если да, то вопрос: "Вы знаете, что такое отладка?". Нет, я серьезно. Всякие точки останова, шаги, перехват исключений и т. д. Вы использовали эту функциональность как пользователь (программист), но интересовались, что там под капотом?

Я такой человек, что мне сложно что-то использовать, не понимая как это работает. Не люблю магию и черные ящики. Если вы такой же, как и я, то добро пожаловать. Приключение на 20 минут.

ВНИМАНИЕ 1: Автор не клеймит себя великим знатоком, он может (и скорее всего) ошибаться. Если нашли ошибку — сообщите, исправлю.

ВНИМАНИЕ 2: статья получилась объемной и некоторые части осилить сможет не каждый. Если что-то не поняли сейчас, то сможете вернуться позже. Но прочитав и поняв каждую секцию сильно прокачаетесь как разработчик, причем не важно какого стека - материал носит общий характер. Это база.

Содержание:

* [Ядро](https://habr.com/ru/articles/867478/#core)
  + [Ждем точку останова](https://habr.com/ru/articles/867478/#core-wait-breakpoint)
  + [Основные манипуляции](https://habr.com/ru/articles/867478/#core-manipulations)
  + [Продолжение работы](https://habr.com/ru/articles/867478/#core-continue)
  + [Изначальный запуск](https://habr.com/ru/articles/867478/#core-start)
  + [Собирая детали вместе](https://habr.com/ru/articles/867478/#core-put-together)
  + [int 0x3](https://habr.com/ru/articles/867478/#core-int3)
* [Форматы отладочных символов](https://habr.com/ru/articles/867478/#debug-formats)
  + [DWARF](https://habr.com/ru/articles/867478/#debug-formats-dwarf)
  + [STABS](https://habr.com/ru/articles/867478/#debug-formats-stabs)
  + [OMF](https://habr.com/ru/articles/867478/#debug-formats-omf)
  + [CTF](https://habr.com/ru/articles/867478/#debug-formats-ctf)
  + [BTF](https://habr.com/ru/articles/867478/#debug-formats-btf)
  + [VMS](https://habr.com/ru/articles/867478/#debug-formats-vms)
  + [CodeView](https://habr.com/ru/articles/867478/#debug-formats-codeview)
  + [ADATA](https://habr.com/ru/articles/867478/#debug-formats-adata)
  + [IEEE-695](https://habr.com/ru/articles/867478/#debug-formats-ieee695)
* [Пишем свой отладчик](https://habr.com/ru/articles/867478/#dmbg)
  + [Запуск](https://habr.com/ru/articles/867478/#dmbg-launch)
  + [Чтение отладочных символов](https://habr.com/ru/articles/867478/#dmbg-debug-syms)
  + [Точки останова](https://habr.com/ru/articles/867478/#dmbg-breakpoints)
  + [Шаги в исходном коде](https://habr.com/ru/articles/867478/#dmbg-steps)
  + [Чтение переменных](https://habr.com/ru/articles/867478/#dmbg-vars)
  + [Бэктрейс](https://habr.com/ru/articles/867478/#dmbg-backtrace)
  + [Полезности](https://habr.com/ru/articles/867478/#dmbg-utilities)
    - [Отображение исходного кода](https://habr.com/ru/articles/867478/#dmbg-utilities-src)
    - [Дизассемблирование](https://habr.com/ru/articles/867478/#dmbg-utilities-disasm)
* [Исследуем gdb](https://habr.com/ru/articles/867478/#gdb)
  + [Структура проектов](https://habr.com/ru/articles/867478/#gdb-projects)
  + [Архитектура](https://habr.com/ru/articles/867478/#gdb-arch)
  + [Запуск](https://habr.com/ru/articles/867478/#gdb-start)
  + [Точки останова](https://habr.com/ru/articles/867478/#gdb-breakpoints)
  + [Шаги](https://habr.com/ru/articles/867478/#gdb-steps)
  + [Чтение переменных](https://habr.com/ru/articles/867478/#gdb-vars)
  + [Бэктрейс](https://habr.com/ru/articles/867478/#gdb-backtrace)
* [Отладка управляемых языков](https://habr.com/ru/articles/867478/#managed)
  + [C#](https://habr.com/ru/articles/867478/#managed-csharp)
  + [Java](https://habr.com/ru/articles/867478/#managed-java)
  + [Python](https://habr.com/ru/articles/867478/#managed-python)
  + [JavaScript](https://habr.com/ru/articles/867478/#managed-javascript)

Ядро

Итак, мы хотим понять как происходит отладка. Для начала определим наш сетап. У меня Linux + amd64 (x86) и у читателя, думаю, тоже, поэтому ядро отладки рассмотрим для этой конфигурации.

Для этой конфигурации весь процесс отладки крутится вокруг 2 вещей:

* ptrace
* INT 3

ptrace — это системный вызов, который используется для отслеживания процесса (**p**rocess **trace**):

**long** ptrace(**enum** \_\_ptrace\_request request, **pid\_t** pid, **void** **\***addr, **void** **\***data);

Можно заметить сходство с другими вызовами по типу ioctl или fcntl — код запроса (перечисление \_\_ptrace\_request), идентификатор объекта (pid) и вспомогательные данные (addr, data).

Понимаю, что на этом вызове много обязанностей и трудно придумать более тонкий вариант, но это как-то контрастирует с философией Unix — делают что-то одно.  
Может я прикапываюсь.

Благодаря этому системному вызову можно:

* PTRACE\_CONT/PTRACE\_SINGLESTEP — получить контроль над выполнением.
* Прочитать и изменять:
  + PTRACE\_GETREGS/PTRACE\_SETREGS — регистры.
  + PTRACE\_PEEKDATA/PTRACE\_POKEDATA — память и код.
  + PTRACE\_PEEKUSER/PTRACE\_POKEUSER — из USER area.
  + PTRACE\_SET\_THREAD\_AREA — локальные данные потока (TLS).
* PTRACE\_SET\_SYSCALL — подменять вызываемые системные вызовы.
* PTRACE\_SECCOMP\_GET\_FILTER — получать информацию о [seccomp]([https://man.Можно](https://man.xn--o0a6cvacc/) заметить сходство с другими вызовами по типу ioctl или fcntl — код запроса (перечисление \_\_ptrace\_request ), идентификатор объекта ( pid ) и вспомогательные данные ( addr , data ).archlinux.org/man/seccomp.2.en).

Для PTRACE\_PEEKUSER сказал, что чтение происходит из USER area. Я не уверен в том, что это можно перевести как область пользователя или пользовательская область. Но ее суть в следующем — в области памяти процесса есть специальная область памяти, называемая USER area. В ней хранится специальная структура struct user, которая определена в заголовочном файле <sys/user.h>. Когда мы вызываем ptrace с PTRACE\_PEEKUSER (или POKE), то можем передать смещение нужного нам поля (в С это делается с помощью ключевого слова offsetof) - это и будет адрес чтения/записи.

Откуда пришли PEEK & POKE

Для начала рассмотрим общий взгляд на процесс отладки. Грубо говоря, процесс отладки итеративный и каждую итерацию можно представить в виде 3 шагов:

1. Ждем точку останова.
2. Выполняем необходимые действия.
3. Продолжаем работы.

Рассмотрим каждый шаг.

Далее буду использовать терминологию из man: tracer — отладчик, tracee — процесс, который отлаживаем.

Ждем точку останова

Во-первых, нам необходимо дождаться остановки tracee. Пока представим, что она есть и рано или поздно tracee до нее дойдет.

Главный принцип работы ptrace можно описать так: tracee получает сигнал (SIGTRAP) и останавливается, а tracer в этот момент получает контроль, что-то делает и продолжает работу tracee. Для ожидания этого сигнала tracer использует waitpid.

И сам смысл точки останова в следующем:

tracee останавливается каждый раз, когда *получает сигнал* (любой), а tracer в это время пробуждается и *WIFSTOPPED*, вызванный на полученном статусе, возвращает true.

То есть, отслеживаемый процесс получает любой сигнал, в этот момент останавливается и waitpid возвращается с соответствующим статусом.

Ждать остановки tracee требуется обязательно — практически все ptrace вызовы, которые изменяют состояние tracee, вернут ESRCH, если будут вызваны над работающим процессом. С одной стороны, ограничение — нельзя вставить палку в колеса едущему велосипеду, но с другой — интересно было бы посмотреть, что случится. Только 5 команд не требуют остановленного состояния, но и они используются для начала и окончания процесса отладки. Об этих командах можно посмотреть в секции Informational and restarting ptrace commands в man 2 ptrace.

Разные статусы остановки

Когда waitpid вернулся, то необходимо проверить его статус. Грубо говоря, статус можно разделить 2 на случая — останов и конец работы процесса. Определить это можно по упомянутому выше WIFSTOPPED. С учетом того, что останавливаться будем множество раз, то код отладчика в общем случае можно представить следующим образом:

**pid\_t** pid;  
**int** wstatus;  
**while** (true)  
{  
 /\*   
 \* Ждем остановки tracee   
 \*/  
 **pid\_t** ret\_pid = waitpid(pid, &wstatus, 0);  
 **if** (!WIFSTOPPED(wstatus))  
 {  
 /\*   
 \* Процесс завершил работу  
 \*/  
 **break**;  
 }  
  
 /\*   
 \* Выполняем необходимые действия  
 \* ...  
 \*/  
  
 /\*   
 \* И продолжаем работу  
 \* ...  
 \*/  
}

Основные манипуляции

На этом моменте, tracee остановлен, а нам (дебаггеру) вернули управление из waitpid. Представим, что точка останова сработала и процесс не завершил свою работу. Теперь начинается самое интересное — процесс отладки: просмотр и изменение состояния tracee.

Команды, которые призваны получать и изменять состояние tracee, в документации называются информационными (informational). Есть множество аспектов процесса, которыми мы можем управлять, и для них есть по 2 команды (PTRACE\_\*) — для чтения и записи.

Сейчас нас будут интересовать только PTRACE\_PEEKTEXT/PTRACE\_POKETEXT и PTRACE\_GETREGS/PTRACE\_SETREGS. Первая команда используется для получения и изменения данных и кода процесса, а вторая — для регистров.

Дополнительно есть PTRACE\_PEEKDATA/PTRACE\_POKEDATA.  
Эта команда должна работать с данными процесса (областью памяти), а PTRACE\_PEEKTEXT/PTRACE\_POKETEXT — с секцией кода (.text). Но, как описано в man, у линукса адресное пространство данных и кода единое, поэтому обе команды выполняют одно и то же.

В сигнатуре ptrace используется 4 аргумента: первые 2 для идентификации процесса и операции, а вторые 2 — для передачи специфичных для конкретной операции параметров. В частности, передаются 2 указателя. Их семантика отличается от команды к команде, а некоторые команды используют только часть, либо не используют эти параметры вообще, поэтому для каждой команды надо смотреть отдельно.

Для примера, если rax равен 0, то записать число 1 (4 байтное число), по адресу из регистра rbx. Реализовать это можно следующим образом:

**struct** user\_regs\_struct regs;  
ptrace(PTRACE\_GETREGS, child\_pid, NULL, &regs);  
**if** (regs.rax == 0)  
{  
 **void** **\***addr = (**void** **\***)regs.rbx;  
 **long** data = ptrace(PTRACE\_PEEKTEXT, child\_pid, addr, NULL);  
 \*((**int** **\***)&data) = 1;  
 ptrace(PTRACE\_POKETEXT, child\_pid, addr, (**void\***)data);  
}

Обратите внимание на то, как записывается число 1. Загвоздка в том, что команды PEEKTEXT и POKETEXT работают с *машинным словом*, грубо говоря, размером указателя. У меня 64-битная машина, поэтому его размер 8 байтов, а записываю я 4-х байтное. Если бы я хотел записать 8 байтное (long), то мог бы сделать это проще, без read-modify-write.

**struct** user\_regs\_struct regs;  
ptrace(PTRACE\_GETREGS, child\_pid, NULL, &regs);  
**if** (regs.rax == 0)  
{  
 **void** **\***addr = (**void** **\***)regs.rbx;  
 **long** data = 1L;  
 ptrace(PTRACE\_POKETEXT, child\_pid, addr, (**void\***)data);  
}

Ну и вообще, вся работа с памятью (чтение и запись) ведется подобным образом — читаем необходимый диапазон памяти по машинному слову и после записываем также по машинному слову.

Продолжение работы

Когда все необходимые действия выполнили, то надо продолжить работу процесса.  
Сделать это можно 2 способами:

* PTRACE\_SINGLESTEP — выполнить только 1 инструкцию.
* PTRACE\_CONT — продолжить выполнение.

Логично, что следующая остановка у первой команды будет у начала следующей инструкции, а у второй — при следующем сигнале.

Изначальный запуск

Последнее, что мы не покрыли — как начинается процесс отладки. У нас есть 2 варианта:

* Сами запускаем процесс для отладки
* Присоединяемся к уже работающему процессу

Самостоятельный запуск

Для самостоятельного запуска есть следующий паттерн:

1. Форкаемся.
2. Потомок вызывает ptrace(PTRACE\_TRACEME, 0, NULL, NULL) и запускает exec\*\* с нужной программой.
3. Родитель постоянно вызывает waitpid с пидом потомка и ждет, пока не остановится.

**void** child\_main()  
{  
 /\*   
 \* Позволяем родителю отслеживать себя  
 \*/  
 ptrace(PTRACE\_TRACEME, 0, NULL, NULL);  
 /\*   
 \* Запускаем отлаживаемую программу  
 \*/  
 execlp("executable", "arg1");  
}  
  
**void** parent\_main(**pid\_t** child)  
{  
 **int** wstatus;  
 **while** (1)  
 {  
 /\*   
 \* Ждем остановки потомка  
 \*/  
 waitpid(child\_pid, &wstatus, 0);  
 **if** (!WIFSTOPPED(wstatus))  
 {  
 /\*  
 \* Потомок завершил работу  
 \*/  
 **break**;  
 }  
 /\*  
 \* Выполняем необходимые действия над потомком  
 \*/  
 perform\_actions(child\_pid);  
 /\*   
 \* Продолжаем его работу  
 \*/  
 ptrace(PTRACE\_CONT, child\_pid, NULL, NULL);  
 }  
}  
  
**int** main()  
{  
 **pid\_t** child\_pid;  
 **if** ((child\_pid = fork()) == 0)  
 {  
 child\_main();   
 }  
 **else**  
 {  
 parent\_main(child\_pid);  
 }  
}

Присоединяемся к запущенному процессу

Если какой-то процесс уже запущен и мы хотим его отладить, то для этого используется уже другой вызов - PTRACE\_ATTACH.

При присоединении к другому процессу через PTRACE\_ATTACH отлаживаемому процессу посылается SIGSTOP - это нужно, чтобы мы могли остановить процесс и поработать с ним. Системные вызовы могут вернуть EINTR, когда они прерываются сигналом. Поэтому, можно сказать, что PTRACE\_ATTACH создает этот errno.

**int** main()  
{  
 **int** wstatus;  
 **pid\_t** child\_pid = obtain\_pid();  
 ptrace(PTRACE\_ATTACH, child\_pid, NULL, NULL);  
 **while** (1)  
 {  
 waitpid(child\_pid, &wstatus, 0);  
 **if** (!WIFSTOPPED(wstatus))  
 {  
 /\*  
 \* Потомок завершил работу  
 \*/  
 **break**;  
 }  
 /\*  
 \* Выполняем необходимые действия над потомком  
 \*/  
 perform\_actions(child\_pid);  
  
 /\*   
 \* Продолжаем его работу  
 \*/  
 ptrace(PTRACE\_CONT, child\_pid, NULL, NULL);  
 }  
}

Если нам НЕ нужно останавливать процесс после присоединения, то для этого используется PTRACE\_SEIZE. В этом случае, нам становится доступна еще одна команда - PTRACE\_INTERRUPT. Как понятно из имени, он нужен для прерывания потока.

В man'е описано 4 варианта развития событий при выполнении этой команды:

* При выполняющемся системном вызове этот вызов вернет EINTR
* Уже остановленный, но при применении PTRACE\_LISTEN
* При конкурентном событии остановки
* Все остальные случаи

Собирая детали вместе

На этом моменте, в голове уже имеется общая картина, того как работают отладчики: процесс останавливается сигналом, мы его препарируем ptrace'ом и продолжаем его работу.

Для примера, мы будем использовать программу для сложения 2 чисел. В ней будет ошибка и нам нужно будет ее отловить - вместо + используется -. Но эта программа будет не на C/Java/Go и т.д. Мы оперируем инструкциями, единственным что понимает процессор (и точки останова это машинные инструкции). Соответственно мы должны работать не с исходным кодом, а инструкциями (для простоты далее, буду говорить ассемблер).

Для начала, код нашей багованной программы:

Код tracee

Теперь нам нужно найти проблемное место. Представим, что знаем, что оно в функции sum и мы должны поставить точку останова прямо перед выходом, чтобы посмотреть, что эта функция возвращает.

Готового отладчика нет, поэтому придется написать самим. Мы не будем сейчас писать общий отладчик для всех и каждого - напишем конкретно для этой программы. Но для начала, что такое точка останова.

int 0x3

Процессор понимает только инструкции и точка останова одна из таких.

int 0x3 - это инструкция точки останова:

* int - это ассемблерная инструкция, генерирующая прерывание. Она принимает на вход аргумент
* 0x3 - номер прерывания точки останова

Когда выполнение доходит до этой инструкции, то вызывается обработчик из [вектора прерываний](https://wiki.osdev.org/Interrupt_Vector_Table#:~:text=On%20the%20x86%20architecture%2C%20the,older%20software%20(e.g.%20MS-DOS%20programs)). В результате Linux посылает сигнал SIGTRAP, который перехватывает отладчик и вот в момент его получения мы можем получить контроль над процессом.

SIGTRAP особенный

Но мало знать про молоток, надо знать как его использовать.

Во-первых, как эту точку останова добавить? Первым в голову приходит вставить эту инструкцию перед той, где нужно остановиться. Но это практически не реализовать: нам нужно будет сдвинуть все остальные инструкции далее (все существующие адреса останутся прежними), но при этом выделенное для секции кода место не расширить. Единственное решение - это переписать существующие инструкции. Кто-то может спрость "как так? секция .text недоступна для записи!". Да, но только для пользовательского пространства, а в случае с ptrace - 1) часть ядра, 2) может выполняться только если выполняется с полученными привилегиями. Короче говоря, с помощью ptrace можно изменять и секцию .text (в общем, он может манипулировать адресным пространством как захочет). Вы могли заметить команду PTRACE\_POKETEXT - как можно догадаться это она изменяет код программы (то что по переданному адресу находится).

int ARG - это полноценная инструкция с аргументом. x86 - это CISC, то есть все инструкции имеют разную длину. Если посмотреть в эту инструкцию, то окажется, что она занимает 2 байта: 0xCD ARG. А теперь представим, что мы хотим поставить точку останова в таком месте:

/\*   
 \* Пользователь обновил свой интервал бронирования номера - нам нужно уведомить об этом других  
 \*/  
**void** notify\_user\_rest\_interval(**long** start, **long** end)  
{  
 **if** (start == end)  
 {  
 /\*   
 \* Однодневный интервал пропускаем для оптимизации  
 \*/  
 **return**;  
 }  
  
 /\*   
 \* Уведомляем по телеге  
 \*/  
 notify\_telegram();  
}

После компиляции, мы можем получить такой ассемблерный код (справа машинный код):

notify\_user\_rest\_interval:

cmpq %rdi, %rsi # 48 39 FE

jnz pass # 75 01

ret # C3

pass:

call notify\_telegram # e8 00 00 00 00

Что случится, если поставим точку останова на ret? Подсказка, ret занимает 1 байт, а точка останова - 2. Правильно, мы затрем следующую инструкцию - call. А что тогда произойдет, если мы выполним jnz на этот (бывший) call? Правильно, ничего хорошего. Но для наглядности посмотрим, что будет при 2 байтной инструкции.

notify\_user\_rest\_interval:

cmpq %rdi, %rsi # 48 39 FE

jnz pass # 75 01

int $3 # CD

pass:

addl (%rax), %eax # 03 00

addb %al, (%rax) # 00 00

n/a # 00

Здесь важно заметить, что для call я ранее передавал нули, так как реальный адрес появится только во время выполнения. Поэтому для примера я также использую нули.

И уже тут можно заметить, что:

1. Вызов функции notify\_telegram исчез
2. Добавились новые инструкции, которые изменяют содержимое регистров (наше состояние), причем скорее всего мусором, так как используется rax - регистр для результата функции (не инициализирован)
3. В конце вообще появилась некорректная инструкция 00 - обозначил n/a

В данном случае, мы хотя-бы словим SIGILL за счет последней некорректной инструкции и ничего плохого не сделаем. Но с учетом того, что адрес будет случайным (спасибо ASLR), то на месте мной указанных появятся случайные инструкции и тогда будь что будет.

Хорошо что об этом уже позаботились. Для int 3 существует отдельная 1-байтовая инструкция - 0xCC. Вот ее мы и будем использовать.

Таким образом, в реализации точек останова можно выделить 2 основных алгоритма: ее выставление и момент захода в нее.

Выставление точки останова:

1. Получаем адрес нужной нам инструкции (ptrace(PEEKTEXT))
2. Пишем поверх нее инструкцию int 0x3/0xCC (запоминаем что затерли!) (ptrace(POKETEXT))

Заход в точку останова:

1. Точка останова срабатывает (генерирование прерывания)
2. Tracer подготавливает окружение:
   1. Получает управление (SIGTRAP)
   2. Переходит к предыдущей инструкции (RIP = RIP - 1)
   3. Восстанавливает затертую инструкцию (ptrace(POKETEXT))
3. Tracer выполняет необходимые манипуляции
4. Tracer продолжает выполнение tracee

Для tracee это все происходит прозрачно - кроме сигнала практически нет внешних эффектов.

Вот теперь, можно сказать, что тему ядра отладчиков покрыли. Суммируя:

1. Вся магия заключена в комбинации системного вызова ptrace и инструкции точки останова int 0x3
2. Отладчик просто ждет пока потомок не остановится при возникновении очередного сигнала
3. Для создания этого сигнала используется int 0x3, которая создает SIGTRAP сигнал
4. Точка останова - это int 0x3 записанная поверх исходных инструкций и, когда до нее доходим, восстанавливаем исходные инструкции и ступаем на 1 инструкцию назад

Возвращаюсь к исходной программе на ассемблере, которую мы хотели отладить (где вместо суммы было вычитание). Используя новые знанния мы можем написать такой ad-hoc отладчик:

ad-hoc отладчик

Но ставит точку останова на функции (точнее сказать метке) sum, по ее достижении выводит значения RDI и RSI, восстанавливает инструкцию, отходит назад (1 байт инструкции), выполняет 1 инструкцию (sub), снова выводит содержимое этих регистров.

Я получил такой вывод (передал аргументы 1 и 2):

Before sum: RDI = 1, RSI = 2

After sum: RDI = -1, RSI = 2

-1

Вроде бы просто, но нет - я схитрил. Linux использует [ASLR](https://en.wikipedia.org/wiki/Address_space_layout_randomization), то есть адрес этой метки должен быть случайным, но если посмотрите в код, то увидите, что этот адрес я уже знаю (константа). Почему? Потому что я специально скомпилировал код без поддержки ASLR (флаг -no-pie). По умолчанию, все бинари наоборот включают его, так как это повышает безопасность. А знают эту информацию символы отладки.

Файлы этого ad-hoc отладчика можно найти [здесь](https://github.com/ashenBlade/habr-posts/tree/post/how-debuggers-work/how-debuggers-work/src). Там же файлы для сборки.

Детали работы отладчиков

Когда tracee останавливается мы знаем где находимся сейчас относительно *машинных инструкций*, но больше особо ничего - ни о типах (структурах), ни об исходном коде, ни о чем другом не знаем. Это дополнительная информация, которая в основном полезна во время разработки (еще полезен стэк-трейс при кордампе, но это другая история). Чтобы эту информацию получить мы должны использовать отладочную информацию. Она хранится в отладочных символах, мосте, соединяющим 2 мира - машинный код и исходный код.

Форматы отладочных символов

Скорее всего, вы и так уже знаете, что такое отладочные символы. Это специальные символы (symbols), которые добавляют некоторую полезную информацию, относительно исходного кода.

Существует множество форматов для разных ЯП, но в ядре своем они нужны, чтобы получить 3 вида информации:

* Типы - встроенные/примитивные, пользовательские, модификаторы, размер
* Символы - переменные (глобальные или локальные), функции
* Исходный код - отображение машинных инструкций на строки исходного кода

Существует множество форматов этих символов. Я чаще всего слышал про DWARF (в контексте \*nix) и CodeView (в контексте Windows). Он не единственный, но начну с него.

DWARF

[DWARF, Debugging With Arbitrary Record Formats](https://dwarfstd.org/doc/DWARF5.pdf) - формат отладочной информации, широко распространенный в \*nix'ах (Linux, FreeBSD и т.д.).  
Существует несколько версий и на момент написания статьи последняя версия - 5.Дальнейшее описание будет для этой версии.

Символы хранятся в том же файле, что и код. Для хранения отладочной информации используются секции. Название всех секций начинается с префикса .debug\_. Работать будем с этими таблицами:

* .debug\_info - общая отладочная информация. Здесь хранится информация о типах и символах
* .debug\_line - информация для таблицы строк. Из нее создаем таблицу строк для отображения строк исходного кода на машинные инструкции

Кроме них есть еще и:

* .debug\_abbrev - "схема", хранит информацию об типах атрибутов и DIE. Например, то что DW\_AT\_name в DW\_TAG\_subprogram - это строка (что за DW\_xxx будет далее)
* .debug\_names - таблица публичных имен (строк), на которую ссылаются некоторые записи (вместо того, чтобы хранить строку непосредственно)
* .debug\_loc - список мест, в которых может находиться переменная т.е. может менять свое нахождение в процессе работы. Например, между вызовами функций хранится в разных регистрах (если код оптимизирован).
* .debug\_str - общая таблица строк. В ней хранятся строки, которые не попали в другие секции
* .debug\_macro - информация о макросах
* .debug\_addr - таблица адресов различных объектов, которые нуждаются в релокации

Внутри вся информация хранится иерархично, в виде дерева. Каждый узел этого дерева называется DIE - Debugging Information Entry. DIE хранят в себе самую различную информацию, например, могут представлять из себя выражение, которое нужно вычислить для получения адреса переменной или информацию об объектном модуле.

Для того, чтобы уметь различать DIE используются тэги. Название каждого тэга начинается с префикса DW\_TAG\_. Например, DW\_TAG\_constant - это константа. Дальнейшее содержимое определятся уже этим тэгом.

У каждого DIE есть свой список атрибутов (пар ключ-значение). Ключом также является перечисление, название которого начинается с DW\_AT\_. Например, DW\_AT\_const\_value - константное значение (может быть для константы, а может и для значений enum). Значением может быть выражение, константа и д.р. За каждым атрибутом закреплен свой список допустимых типов значений.

Так, для DW\_TAG\_constant допустимы следующие атрибуты:

* DW\_AT\_name - название константной переменной
* DW\_AT\_type - тип константы
* DW\_AT\_endianity - big-endian или little-endian значение константы
* DW\_AT\_const\_value - само значение константы
* DW\_AT\_external - видна ли другим CU
* DW\_AT\_declaration - является ли только объявлением
* DW\_AT\_accessibility - видимость переменной
* DW\_AT\_start\_scope - область, в которой переменная становится доступна
* DW\_AT\_visibility - видимость символа (локальная, глобальная ...)

Также стоит упомянуть и о выражениях, тем которые надо вычислить для получения целевого значения, например, для получения адреса поля структуры. Для их вычисления создается стек (на него кладутся операнды и операторы), а затем вычисляются команды своей виртуальной машины.

Узлы, которые представляют собой операции/команды, имеют свое перечисление, начинающееся с DW\_OP\_, например, DW\_OP\_div - деление, а DW\_OP\_deref - разыменовывание указателя на вершине стека (значение на вершине стека интерпретируется как указатель).

Вся программа - это множество Compilation Unit (CU). Каждый CU хранит в себе специфичные для своего ЯП детали: модуль/неймспейс/импортированные объекты и т.д.

Если мы говорим о C, то часто в нем будут храниться функции. Для представления функций используется DIE - DW\_TAG\_subprogram. В нем нам интересны 3 атрибута:

* DW\_TAG\_name - название функции
* DW\_TAG\_low\_pc и DW\_TAG\_high\_pc - диапазон инструкций, занимаемых функцией

С помощью них можно понять, где мы сейчас находимся (на основании текущего RIP).

Для получения информации о переменных используется DIE DW\_TAG\_variable. В нем нам интересны следующие атрибуты:

* DW\_TAG\_name - название переменной
* DW\_TAG\_location - расположение переменной (выражение)
* DW\_TAG\_type - тип переменной

Для получений результирующей информации - обходим все функции внутри каждого CU и все переменные внутри каждой функции. Все это - обход дерева.

Таким образом, мы получаем информацию о символах. Теперь перейдем к представлению типов.

DWARF определяет несколько групп типов. Часть из них специфична для конкретных ЯП, например, coarray (массив, который хранится в другом процессе). Все их описывать не буду, остановлюсь на 2 подтипах: базовые и составные.

Базовые, DW\_TAG\_base\_type - это примитивные типы, которые встроены в язык. Например, int. Все такие типы по умолчанию считаются численными - целочисленные или с плавающей точкой. Для того, чтобы их различать используется атрибут DW\_AT\_encoding. Например, значение DW\_DS\_singed - знаковое число, а DW\_DS\_float - с плавающей точкой.

Составные, DW\_TAG\_structure\_type, DW\_TAG\_class\_type - это составные типы, структуры/классы. В нем интересны следующие поля:

* DW\_AT\_name - название структуры (если не анонимная)
* DW\_AT\_byte\_size - размер в байтах

Для членов структуры используется DIE DW\_TAG\_member (дочерний узел). В нем нам интересны следующие атрибуты:

* DW\_AT\_name - название поля
* DW\_AT\_type - тип этого поля
* DW\_AT\_data\_member\_location - выражение для получения адреса этого поля

Но это касается типов, которые представляют уже конкретные *данные* (назову так). Но вот когда мы указываем тип, то может указать дополнительные квалификаторы: указатель, volatile, const и т.д. Они обрабатываются интересным способом. Для каждого такого квалификатора определяется свой тэг. Например, для указателя это DW\_TAG\_pointer\_type, а для const - DW\_TAG\_const\_type. А далее, такие декораторы указывают на целевой тип. В результате, для const SampleStruct\*\* мы получим следующее дерево:

/\*

\* struct SampleStruct {

\* int value;

\* const double value2;

\* };

\*/

DW\_TAG\_pointer\_type

DW\_TAG\_pointer\_type

DW\_TAG\_const\_type

DW\_TAG\_structure\_type

DW\_AT\_name: SampleStruct

DW\_TAG\_member

DW\_AT\_name: value

DW\_TAG\_base\_type

DW\_AT\_name: int

DW\_TAG\_member

DW\_AT\_name: value2

DW\_TAG\_const\_type

DW\_TAG\_base\_type

DW\_AT\_name: double

Теперь должно стать понятно, как получать информацию о типах: обходим все CU и обрабатываем каждый DIE для типа в зависимости от тэга: базовые - просто запоминаем, составные - спускаемся и исследуем внутренности (поля).

Теперь, перейдем к получению информации об исходном коде. Эта информация хранится в секции .debug\_lines. При отладке нам нужна информация по каждой строке - файл, номер строки, номер столбца, сколько занимает, эпилог или пролог и т.д. Если хранить всю эту информацию в чистом виде, то размер исполняемого файла будет слишком огромным. Поэтому часто для хранения этой информации используют свои "виртуальные машины" - код, который нужно выполнить для получения таблицы строк. В DWARF также пошли этим путем.

Это программа, которая применяет команды для своей машины состояний. В DWARF это называется line number program. Вся работа заключается в том, чтобы изменять состояние регистров этой виртуальной машины. Примеры этих регистров: address - PC инструкции, file - индекс названия файла, line - номер текущей строки.

Сама программа состоит из заголовка с метаданными и последовательности команд. Команды могут иметь операнды, но не обязательно. Примеры команд:

* DW\_LNS\_advance\_pc - сместить текущий PC на указанное значение
* DW\_LNS\_advance\_line - сместить текущий номер строки на указанное значение
* DW\_LNS\_set\_file - выставить указанное название исходного файла

После каждой выполненной команды в таблицу строк добавляется новая запись об исходном коде. Например, после DW\_LNS\_advance\_pc будет новая запись, в которой разница с предыдущей только в 1-ом поле.

DWARF поддерживает широкий диапазон различных ЯП вместе с их особенностями. Согласно перечислению DW\_LANG (значения языков) это:

* C
* C++
* Ada
* COBOL
* Fortran
* Pascal
* Modula
* Java
* PLI
* Objective C
* Objective C++
* UPC
* D
* Python
* OpenCL
* Go
* Haskell
* OCaml
* Rust
* Swift
* Julia
* Dylan
* RenderScript
* BLISS

Для некоторых ЯП имеются отдельные значения для разных версий. Так для C это DW\_LANG\_C11 и DW\_LANG\_C89. Сама эта "особенная" информация относится к особенностям языка. Модули в Modula, level-88 condition из COBOL (конструкция похожая на switch/case), выражения with из Pascal или Modula. Для таких фич существуют отдельные атрибуты. Они не имеют смысла для других ЯП.

STABS

Теперь, рассмотрим другие форматы. И начнем со STABS.

[STABS](https://./stabs.pdf) (Symbol TABle Strings) - один из первых форматов отладочных символов (1980-е). Название исходит из того, что все данные для отладки хранились в виде простых строк в таблице символов объектного файла (тогда еще a.out - не ELF).

Изначально создавался для pdx - дебаггера Pascal. Сейчас поддерживается и для других языков.

Вся отладочная информация хранится в специальных таблицах символов - stab. Для добавления разной информации имеются разные директивы ассемблера:

* .stabs (stab string) - строковая информация: функции, структуры, строки исходного кода, глобальные переменные (используется чаще всего)
* .stabn (stab number) - числовая информация (скорее та, для которой не нужны строки): общее количество символов, адрес catch для GNU C++ (N\_CATCH)
* .stabd (stub dot) - то же, что и stabn, но значение - текущая строка (явно не указывается значение): (примеров не нашел)

Внимание акцентируем на .stabs. Первым аргументом эта директива принимает строку. И вот тут кроется сила этого формата, так как с помощью простой строки можно описать довольно много вещей. Но важен и 2 аргумент - type. Если коротко, то это тип этой записи. Она определяет семантику строки. Например, для функций используется 36 (N\_FUN), а для типов - 128 (N\_LSYM). Рассмотрим представление типов.

Для описания типов используется .stabs. Первым аргументом он принимает строку. В формате: name:symbol-descriptor type-information:

* name - название символа (вообще, кому принадлежит это название определяется следующим полем - символу, типу и т.д.)
* symbol-descriptor - тип этого символа. Например, c - константа, s - локальная переменная, а t - это тип
* type-information - определение свойств этого символа.

Основная информация о типе содержится в type-information и у него есть свой формат.

Вначале, идет символ - type descriptor. Он описывает какого рода перед нами тип: e - перечисление, r - поддиапазон (subrange), s - структура и т.д. Список можно найти [тут](https://sourceware.org/gdb/current/onlinedocs/stabs.html/Type-Descriptors.html#Type-Descriptors).

Далее, указывается идентификатор этого типа. Этим идентификатором может быть как число, так и пара чисел (номер файла, номер типа в этом файле). Последнее, GNU/Sun специфично.

После - список значений атрибутов, разделенных ;.

Посмотрим как представляются встроенные типы, такие как int или char. С ними ситуация интересная так как при определении типа мы можем общего подхода для указания встроенных типов не сложилось: acc (Sun) используется b и R в type-descriptor, а IBM - отрицательные индексы типа, но никто не запрещает указать их как самостоятельные типы (t). Для int и double я получил следующие директивы:

.stabs "int:t(0,1)=r(0,1);-2147483648;2147483647;",128,0,0,0

.stabs "double:t(0,15)=r(0,1);8;0;",128,0,0,0

Как видим, идентификатор - это пара чисел, для встроенных типов используется тип r, а далее идет 2 числа - начало и конец диапазона. И вот это как раз та "convoluted" часть: r - это поддиапазон, но не число. Таким образом, на отладчик возлагается большая ответственность в обнаружении встроенных типов. Но это становится не так сложно, учитывая, что мы знаем ЯП, а в начале type-information идет название самого типа.

Теперь сложные типы - структуры. Для них подтип - s. А вот список атрибутов уже не простые числа - каждый атрибут это отдельное поле и представляет из себя строку name:type id, bit offset from the start of the struct, number of bits in the element. Название поля: ID этого типа, смещение в битах от начала структуры, размер поля в битах.

Для SampleStruct будет следующая запись:

.stabs "SampleStruct:T(0,21)=s16value:(0,1),0,32;value2:(0,15),64,64;;",128,0,0,0

* (0, 21) - идентификатор (не с 1, так как до нее уже были определены другие типы)
* s - структура
* 16= - для структуры, всегда идет число 16, за которым =
* value:(0,1),0,32 - поле value типа int (см. выше), начинающееся с начала (0) и размером в 4 байта (32 бита)
* value2:(0,15),64,64 - поле value2 типа double (см. выше), начинающееся с 8 байта (64 бита) и размером в 8 байт (64 бита)

Также, если есть квалификаторы типа, то для них будут созданы свои типы, на которые далее будем ссылаться (k - const, B - volatile и т.д.). Практически также как и в DWARF.

Но типы это не единственное, что можно хранить с помощью .stabs. Как уже ранее сказал, вторым аргументом передается type - константа типа записи. Для типов это 128 (уже видели на примере директивы), но конечно же есть и другие.

Для функции этот тип будет N\_FUN (36). В самой строке будет name:return-type-id - название функции и тип возвращаемого значения. Передаваемые же аргументы представляются другим типом - N\_PSYM(160). Его строка имеет вид name:type-id - название аргумента и его тип. Таким образом, для входной точки int main(int argc, char \*\*argv) мы увидим нечто подобное:

.stabs "main:F(0,1)",36,0,0,main

.stabs "argc:p(0,1)",160,0,0,-4

.stabs "argv:p(0,23)=\*(0,24)=\*(0,2)",160,0,0,-16

Перейдем к хранению информации о строках исходного кода. Для этой цели используется директива .stabn. Этой директиве также передается ее type. Здесь он первый и для строки исходного кода равен N\_SLINE(68). Второе поле пропустим и сконцентрируемся на 2 последних: Line и Offset. Эта директива добавляется *перед* началом строки исходного кода, поэтому поле Line определяет строку, которой принадлежат дальнейшие инструкции, а Offset - адрес начала этого кода.

Давайте посмотрим, как же эта информация записывается:

**int** main(**int** argc, **char** **\*\***argv)  
{  
 **int** a;  
 **int** b;  
 a = 1;  
 b = 2;  
 a = a \* 10 + b;  
 **return** a;  
}

main:

.stabn 68,0,8,.LM0-.LFBB1

.LM0:

.LFBB1:

.LFB0:

pushq %rbp

movq %rsp, %rbp

movl %edi, -20(%rbp)

movq %rsi, -32(%rbp)

.stabn 68,0,11,.LM1-.LFBB1

.LM1:

movl $1, -8(%rbp)

.stabn 68,0,12,.LM2-.LFBB1

.LM2:

movl $2, -4(%rbp)

.stabn 68,0,13,.LM3-.LFBB1

.LM3:

movl -8(%rbp), %edx

movl %edx, %eax

sall $2, %eax

addl %edx, %eax

addl %eax, %eax

movl %eax, %edx

movl -4(%rbp), %eax

addl %edx, %eax

movl %eax, -8(%rbp)

.stabn 68,0,14,.LM4-.LFBB1

.LM4:

movl -8(%rbp), %eax

.stabn 68,0,15,.LM5-.LFBB1

.LM5:

popq %rbp

ret

Можете заметить, что компилятор добавил метки .LMx, которые обозначают новые строки, и прямо перед ними .stabs директивы с информацией о строках исходного кода.

OMF

[OMF](https://refspecs.linuxfoundation.org/elf/TIS1.1.pdf)(Relocatable Object Module Format) - это на самом деле формат объектных файлов, а не отладочной информации. Но поддержка отладочной информации встроена в него - отдельные, "well-known" записи.

Объектный файл представляется в виде последовательности записей, каждая из которых состоит из 4 "полей":

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Тип** | **Длина** | **Данные** | **Чек-сумма** |
| Размер (байт) | 1 | 2 | N | 1 |
| Описание | Определяет какие данные содержит запись | Длина оставшейся части (Данные и Чек-сумма) | Сама полезная нагрузка. Содержимое зависит от Типа | Чек-сумма всех остальных полей |

Как можно заметить, поле тип определяет содержимое записи. В TIS определили 28 типов записи. Например, запись PUBDEF (PUBlic DEFinitions) определяет список экспортируемых (глобальных) символов. Но пока сфокусируемся на отладке.

Для отладки можно использовать следующие записи:

* LINNUM - Отображение строк исходного кода, на адреса машинных инструкций
* LINSYM - Отображение функций на строки исходного кода (вместе с LINNUM позволяет понять адрес начала функции)
* PUBDEF, LPUBDEF - Функции и переменные - глобальные и локальные (в зависимости от префикса L). Можно сказать, что разница в том указывается ли static или нет
* COMDEF, LCOMDEF - Общие (communal) переменные - неинициализированные статические или те, что могут соответствовать инициализированным, но в другой единице компиляции

У PUBDEF и COMDEF есть свои братья LPUBDEF и LCOMDEF соответственно. Первые экспортируют свои символы, а вторые имеют видимость только в пределах своего модуля.

Перечисленные записи указываются в TIS как актуальные. Но в расширениях были определены и другие (в документе помечены как устаревшие):

* DEBSYM - Дополнительная отладочная информация
* TYPDEF - Информация о типе
* LOCSYM - Информация о локальных символах
* BLKDEF - Информация о блоке программы, которая содержит свою видимость, локальные переменные

Это довольно старый формат: последняя версия V5.0 была издана Intel в 1985. За все время существования многие компании создали свои ответвления/надстройки: в документе (ссылка сверху) дана спецификация для OMF, созданная из компиляции 6 различных источников. После 5 версии выкладывались и другие версии, но они были специфичны для различных платформ, например, IBM в 1993 выложил свою [спецификацию IBM-OMF](http://bitsavers.informatik.uni-stuttgart.de/pdf/ibm/pc/os2/OS2_OMF_and_LX_Object_Formats_Revision_8_199406.pdf). Но не думаю, что этот формат мертв. Как минимум, автор [этой статьи](https://habr.com/ru/articles/576770/) приложил усилия и перевел свою систему программирования на PL/1-KT с 16 на 64 бит.

Рассмотрим хранение информации о типах. Для этого используется запись TYPDEF. В сравнении с предыдущими форматами отладки, здесь поддерживаемая функциональность примитивная. Запись имеет следующую разметку:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Тип** | **Длина** | **Название** | **Флаг EN** | **Leaf-Descriptor** | **Checksum** |
| 0x8E | 2 байта | <var> | 1 байт | <var> | 1 байт |

Из всего этого нам интересно поле Leaf-Descriptor. В нем-то и содержится информация о типе. Это поле также подразбивается на 2 типа: NEAR и FAR. Первый доступен из того же data сегмента, а второй нет.

NEAR имеет следующую разметку:

|  |  |  |
| --- | --- | --- |
| **Код** | **Тип переменной** | **Размер в битах** |
| 0x62 | 1 байт | <var> |

Поле Тип переменной определяет что за тип перед нами. Поддерживается только 3 значения:

* 0x77 - массив
* 0x79 - структура
* 0x7B - скалярное значение

А Размер в битах говорит само за себя - это размер типа в битах. Заметьте еще что размер этого поля - переменный. Кодирование длины использует следующий принцип: первый байт меньше 128 - это и есть значение, 129 - размер занимает следующие 2 байта, 132 - 3 байта, 136 - 4 байта.

FAR имеет уже другую разметку, но поддерживает только тип массива. Останавливаться не буду.

Доступа к компилятору, который создаст мне этот формат, у меня нет, поэтому возьму пример из документации - представление int (16-битного на тот момент) в байтовом виде.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Код** | **Длина** | **Название** | **Флаг EN** | **Код Leaf-Descriptor** | **Тип переменной** | **Размер в битах** | **Чек-сумма** |
| 8E | 06 | 00 (пусто) | 00 | 62 (NEAR) | 7B (скаляр) | 10 (2 байта, 16 бит) | 7B |

Теперь, посмотрим на представление исходного кода. За это отвечает запись LINNUM - Line Numbers Record. Она имеет следующую разметку:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Код** | **Длина** | **Base group** | **Base segment** | **Info** | **Checksum** |
| 0x94/0x95 | 2 байта | 1 или 2 байта | 1 или 2 байта | <var> | 1 байт |

Поля Base group и Base segment позволяют определить код, которому принадлежит запись. А само отображение на исходный код хранится в поле Info. Вообще, это поле называется Debugger Style-specific Information и не просто так: чтобы понять, какого формата записи хранятся в этом поле необходимо прочитать другую запись - COMMENT. В документации описан формат, который использует Microsoft и это просто пара - номер строки (2 байта) и смещение, относительно сегмента (2 или 4 байта).

Таким образом, нам просто нужно пройтись по всему массиву, без необходимости что-то вычислять.

CTF

[CTF](https://www.sourceware.org/binutils/docs/ctf-spec.html) (Compact C Type Format) - это формат отладочной информации, который содержит информацию преимущественно об используемых типах (как можно догадаться из названия) и переменных (внутри функций).

Изначально он разработан для Solaris, но позже был добавлен и в другие ОС:

* Linux - в 2012 портировали libdtrace ([первый коммит](https://github.com/oracle/libdtrace-ctf/commit/248dc18174b23fec1a6a932263b3a2b5ce5adbcc))
* FreeBSD - в нем CTF это [модуль ядра](https://docs.freebsd.org/en/books/handbook/dtrace/#dtrace-implementation), без которого не будет его поддержки

Изначально, CTF проектировался как компактный и простой формат. Компактность достигается за счет:

* Тесного "содружества" с ELF форматом - CTF использует уже готовые таблицы символов, предоставляемые ELF файлом. В частности, переиспользует таблицу строк
* Использования сжатия с помощью zlib

Проста же достигается за счет того, что в символах отладки есть только информация о типах, без какой-либо лишней информации.

Отладочная информация хранится в том же объектном файле, что и программа, в секции .ctf. Внутри этой секции хранится словарь (dictionary). Словарь состоит из 2 частей: заголовка и множества секций.

Заголовок состоит из преамбулы, с помощью которой можно определить версию CTF и как его обрабатывать, и смещений до других секций.

Содержательная часть находится в самих секциях:

* type - описание всех top-level типов программы (можно назвать главной секцией)
* label - соотношение строки из strtab к ID типа
* symtypetab - это 2 секции, которые нужны для соотношения функции/символов из ELF на соответствующие им ID типов
* function info index
* variable - соотношение названия переменной к ее типу
* string - таблица строк в ELF формате

Типы хранятся в секции type, поэтому рассмотрим ее. Из себя она представляет массив записей, которые представляют какой-либо тип. Для идентификации типов используется ID. Это простое число от 0 до 2^32 (т.е. uint32), причем 0 - это ID типа, который представить нельзя.

Каждая запись типа состоит из обязательной части за которой может идти опциональная. Таким образом, размер записи переменный.

Обязательная часть имеет следующее расположение:

**typedef** **struct** ctf\_stype  
{  
 **uint32\_t** ctt\_name; // Name  
 **uint32\_t** ctt\_info; // Info  
 \_\_extension\_\_  
 **union**  
 {  
 **uint32\_t** ctt\_size;  
 **uint32\_t** ctt\_type;  
 }; // Size/Type  
} **ctf\_stype\_t**;

Name - смещение строки названия типа в таблице строк. Size/Type - это union, либо размер (единица измерения зависит от типа), либо сам тип. Но главная часть - Info. Это 4-байта поделенные на 3 поля:

* kind - Type kind (ниже)
* isroot - тип виден остальным при просмотре
* vlen - дополнительное место используемое типом

Из всего этого главным можно назвать Type kind. Он определяет что перед нами за тип. Всего их 14 штук:

/\* https://sourceware.org/git/?p=binutils-gdb.git;a=blob;f=include/ctf.h;h=4414cb0ebed95f9049ec84f9089ef8f149f21eda;hb=32778522c7d8777803c88684b8e428ee729f0b22#l408 \*/  
#define CTF\_K\_UNKNOWN 0 /\* Unknown type (used for padding and  
 unrepresentable types). \*/  
#define CTF\_K\_INTEGER 1 /\* Variant data is CTF\_INT\_DATA (see below). \*/  
#define CTF\_K\_FLOAT 2 /\* Variant data is CTF\_FP\_DATA (see below). \*/  
#define CTF\_K\_POINTER 3 /\* ctt\_type is referenced type. \*/  
#define CTF\_K\_ARRAY 4 /\* Variant data is single ctf\_array\_t. \*/  
#define CTF\_K\_FUNCTION 5 /\* ctt\_type is return type, variant data is  
 list of argument types (unsigned short's for v1,  
 uint32\_t's for v2). \*/  
#define CTF\_K\_STRUCT 6 /\* Variant data is list of ctf\_member\_t's. \*/  
#define CTF\_K\_UNION 7 /\* Variant data is list of ctf\_member\_t's. \*/  
#define CTF\_K\_ENUM 8 /\* Variant data is list of ctf\_enum\_t's. \*/  
#define CTF\_K\_FORWARD 9 /\* No additional data; ctt\_name is tag. \*/  
#define CTF\_K\_TYPEDEF 10 /\* ctt\_type is referenced type. \*/  
#define CTF\_K\_VOLATILE 11 /\* ctt\_type is base type. \*/  
#define CTF\_K\_CONST 12 /\* ctt\_type is base type. \*/  
#define CTF\_K\_RESTRICT 13 /\* ctt\_type is base type. \*/  
#define CTF\_K\_SLICE 14 /\* Variant data is a ctf\_slice\_t. \*/

Да, определено 15 значений, но смысл имеют только 14 - есть специальный тип CTF\_K\_UNKNOWN, 0. Например, в gcc имеется поддержка CTF, которая реализована хуком в [dwarf2ctf.c](https://gcc.gnu.org/git/?p=gcc.git;a=blob;f=gcc/dwarf2ctf.cc;h=00b876e2fe2cb901ca6e8e5c8c01e78f0d812723;hb=36774cec1f8d224e202dd3ca2012dae79d4e8ba9#l921), - если DIE нельзя отобразить в тип CTF, то используется CTF\_K\_UNKNOWN.

Я не нашел подходящего синонима, чтобы перевести type kind лучше, чем "тип типа". Поэтому буду писать на английском как в оригинале.

Дальнейшая обработка записи типа зависит от его type kind. Некоторые используют дополнительное место, а некоторым не надо - это определяется из самого типа.

Пример также на 2 типах: int и struct.

CTF\_K\_INTEGER (1) представляет собой целочисленный тип. Он заполняет поле Size количеством занимаемых байт. Но этого недостаточно и он использует дополнительные поля:

* Encoding - представление этого типа
* Offset - смещение, относительно начала структуры
* Bit-width - размер в битах

Поле Encoding необходимо для понимания того, как правильно представить пользователю этот тип данных (числовой знаковый/беззнаковый, char, bool ...). А вот Offset и Bit-width это уже функциональность для обработки битовых полей - они хоть и занимают Size байт, то полезные данные хранятся в Offset:Offset + Bit-width.

С помощью objdump можно получить информацию о типах CTF (если использовались они). Вот часть вывода, в котором отображаются

Types:

0x1: (kind 1) long unsigned int (format 0x0) (size 0x8) (aligned at 0x8)

0x2: (kind 1) unsigned int (format 0x0) (size 0x4) (aligned at 0x4)

0x3: (kind 1) unsigned char (format 0x2) (size 0x1) (aligned at 0x1)

0x4: (kind 1) short unsigned int (format 0x0) (size 0x2) (aligned at 0x2)

0x5: (kind 1) signed char (format 0x3) (size 0x1) (aligned at 0x1)

0x6: (kind 1) short int (format 0x1) (size 0x2) (aligned at 0x2)

0x7: (kind 1) int (format 0x1) (size 0x4) (aligned at 0x4)

Теперь перейдем к структурам. Их представляет CTF\_K\_STRUCT. Здесь уже используется поле vlen - оно хранит в себе количество полей этой структуры. Каждое поле представляется структурой:

**typedef** **struct** ctf\_member\_v2  
{  
 **uint32\_t** ctm\_name; // Name  
 **uint32\_t** ctm\_offset; // Offset  
 **uint32\_t** ctm\_type; // Type  
} **ctf\_member\_t**;

Name - имя поля (также смещение строки). Offset - смещение относительно начала структуры. Type - ID типа поля.

Также эта структура используется и для перечисления (CTF\_K\_ENUM). Разница в том, что поля перечисления могут пересекаться.

Осталось только сказать о том, как же хранятся квалификаторы типов: const, volatile, указатели и т.д. В общем-то также как это делалось в других форматах - мы создаем новый тип, оборачивающий старый и ссылаемся на него. Теперь, посмотрим что выдаст objdump:

0xa: (kind 6) struct SampleStruct (size 0x10) (aligned at 0x4)

[0x0] value: ID 0x7: (kind 1) int (format 0x1) (size 0x4) (aligned at 0x4)

[0x40] value2: ID 0xc: (kind 12) const double (size 0x8) (aligned at 0x8)

0xb: (kind 2) double (format 0x2) (size 0x8) (aligned at 0x8)

0xc: (kind 12) const double (size 0x8) (aligned at 0x8) -> 0xb: (kind 2) double (format 0x2) (size 0x8) (aligned at 0x8)

0xd: (kind 12) const struct SampleStruct (size 0x10) (aligned at 0x4) -> 0xa: (kind 6) struct SampleStruct (size 0x10) (aligned at 0x4)

0xe: (kind 5) int (\*) () (aligned at 0x8)

0xf: (kind 3) const struct SampleStruct \* (size 0x8) (aligned at 0x8) -> 0xd: (kind 12) const struct SampleStruct (size 0x10) (aligned at 0x4) -> 0xa: (kind 6) struct SampleStruct (size 0x10) (aligned at 0x4)

0x10: (kind 3) const struct SampleStruct \*\* (size 0x8) (aligned at 0x8) -> 0xf: (kind 3) const struct SampleStruct \* (size 0x8) (aligned at 0x8) -> 0xd: (kind 12) const struct SampleStruct (size 0x10) (aligned at 0x4) -> 0xa: (kind 6) struct SampleStruct (size 0x10) (aligned at 0x4)

Букв много, но разобравшись видим:

* структура SampleStruct с 2 полями:
  + value - ID типа 0x7 (int, см. выше)
  + value2 - ID типа 0xC (это const double, ссылающийся на double)
* const struct SampleStruct \*\* (0x10) -> const struct SampleStruct \* (0xF) -> const struct SampleStruct (0xD) -> struct SampleStruct (0xA)

BTF

[BTF](https://docs.kernel.org/bpf/btf.html) (BPF Type Format) - это формат метаданных, разработанный специально для [BPF программ](https://habr.com/ru/articles/493880/). Сегодня, на замену BPF пришел eBPF, но я буду использовать BPF - так короче. Он создан на основании CTF и изначально он поддерживал только информацию о типах (что отражается в самом названии), но после была добавлена поддержка информации о функциях, исходном кода и др.

Сам по себе BPF это последовательность 64-битных инструкций, которые JIT-компилируются ядром. Но для разработки этот ассемблер не очень удобен, поэтому разработали библиотеку libbpf и формат ELF специально для BPF. libbpf читает этот ELF файл, выполняет необходимые манипуляции и сам уже вызывает необходимые системные вызовы. А для создания этого объектника можно использовать понятные нам языки по типу C или Rust.

Отладка возможна благодаря отладочным секциям, которые содержат нужную информацию.

* .BTF - Различная полезная информация - типы, переменные, квалификаторы, функции и т.д.
* .BTF.ext - Данные, необходимые загрузчику для манипуляций перед загрузкой в ядро.
* .BTF\_ids - Содержит BTF ID - идентификаторы, которые использует ядро.

Как можно понять, именно в секции .BTF содержится вся необходимая для нас информация. Она состоит из 2 частей: таблица строк (string table) и массив отладочной информации (в документации говорится type data, но мне кажется "массив отладочной информации" более говорящее название). Таблица строк - это просто массив C-style строк, к которой обращается каждый элемент type data.

Каждый элемент type data имеет общую структуру, но далее каждый конкретный тип расширяет эту структуру, добавляет свои специфичные данные. Под типом имеется ввиду не только type, но разные синтаксические конструкции и другая информация. В исходном коде, это называют KIND, но другого перевода я не нашел.

На данный момент, поддерживаются следующие типы.

/\* https://github.com/torvalds/linux/blob/1722389b0d863056d78287a120a1d6cadb8d4f7b/include/uapi/linux/btf.h#L59 \*/  
**enum** {  
 BTF\_KIND\_UNKN = 0, /\* Unknown \*/  
 BTF\_KIND\_INT = 1, /\* Integer \*/  
 BTF\_KIND\_PTR = 2, /\* Pointer \*/  
 BTF\_KIND\_ARRAY = 3, /\* Array \*/  
 BTF\_KIND\_STRUCT = 4, /\* Struct \*/  
 BTF\_KIND\_UNION = 5, /\* Union \*/  
 BTF\_KIND\_ENUM = 6, /\* Enumeration up to 32-bit values \*/  
 BTF\_KIND\_FWD = 7, /\* Forward \*/  
 BTF\_KIND\_TYPEDEF = 8, /\* Typedef \*/  
 BTF\_KIND\_VOLATILE = 9, /\* Volatile \*/  
 BTF\_KIND\_CONST = 10, /\* Const \*/  
 BTF\_KIND\_RESTRICT = 11, /\* Restrict \*/  
 BTF\_KIND\_FUNC = 12, /\* Function \*/  
 BTF\_KIND\_FUNC\_PROTO = 13, /\* Function Proto \*/  
 BTF\_KIND\_VAR = 14, /\* Variable \*/  
 BTF\_KIND\_DATASEC = 15, /\* Section \*/  
 BTF\_KIND\_FLOAT = 16, /\* Floating point \*/  
 BTF\_KIND\_DECL\_TAG = 17, /\* Decl Tag \*/  
 BTF\_KIND\_TYPE\_TAG = 18, /\* Type Tag \*/  
 BTF\_KIND\_ENUM64 = 19, /\* Enumeration up to 64-bit values \*/  
  
 NR\_BTF\_KINDS,  
 BTF\_KIND\_MAX = NR\_BTF\_KINDS - 1,  
};

Можно заметить, что практически все элементы перечисления похожи на типы CTF, но поддерживается гораздо больший диапазон типов. BTF был добавлен в [этом патче](https://git.kernel.org/pub/scm/linux/kernel/git/netdev/net.git/commit/?id=69b693f0aefa0ed521e8bd02260523b5ae446ad7) и вот отрывок из него:

Hence, it basically focus on the C programming language which the modern BPF is primary using.  
...  
BTF has its root from CTF (Compact C-Type format).

Информация о типах хранится в том же самом представлении, что и в CTF. Различаются только названия констант. Поэтому рассмотрим то, как хранится информация о строках исходного кода.

Для хранения этой информации используется секция .BTF.ext. По аналогии со словарем, эта секция также начинается с заголовка, организованного аналогично: преамбула с версией и смещения до нужных секций. Есть 2 раздела, но нас интересует line\_info - в нем и хранятся строки. Этот раздел внутри делится на несколько частей для разных секций и внутри каждой такой части хранится массив структур bpf\_line\_info:

**struct** bpf\_line\_info {  
 \_\_u32 insn\_off; /\* [0, insn\_cnt - 1] \*/  
 \_\_u32 file\_name\_off; /\* offset to string table for the filename \*/  
 \_\_u32 line\_off; /\* offset to string table for the source line \*/  
 \_\_u32 line\_col; /\* line number and column number \*/  
};

Подобное уже видели - храним смещение инструкции, название файла (смещение в таблице строк) и сам номер строки. Но вот чего еще не было - хранение строки исходного кода напрямую (поле line\_off).

VMS

[VMS](https://www.digiater.nl/openvms/freeware/v50/debug/alpha_dstrecrds.sdl) - это формат отладочной информации, созданный для ОС OpenVMS.

Информации у меня было очень мало. Единственный документ, который указал в ссылке, - это описание структуры DST (Debug Symbol Table), генерируемый компилятором Alpha для отладчика VMS, в формате SDL

В VMS имеется собственный формат объектного файла и информация об отладочных символах в них встроена. В заголовке исполняемого файла имеется поле IHD$W\_SMDBGOFF - адрес (смещение) до Debug Symbol Table.

Сам DST состоит из последовательности DST записей. Каждая запись в начале имеет 2-байтный заголовок: размер и тип. Причем, это не хаотичный набор записей, а иерархичная структура - имеются открывающие и закрывающие записи, которые внутри себя хранят потомков. Например, имеется 2 записи Routine Bebin и Routine End, они определяют границы функции, а внутри них хранятся записи Data - переменные. Пример вложенности из документации:

Program Structure DST Record Sequence

----------------- -------------------

MODULE M = Module Begin M

BEGIN

VAR SYM\_M1: INTEGER; Data SYM\_M1 (DTYPE\_L)

VAR SYM\_M2: REAL; Data SYM\_M2 (DTYPE\_F)

ROUTINE R1 = Routine Begin R1

BEGIN

VAR SYM\_R11: BOOLEAN; Data SYM\_R11 (BOOLEAN)

VAR SYM\_R12: INTEGER; Data SYM\_R12 (DTYPE\_L)

END; Routine End (for R1)

ROUTINE R2 = Routine Begin R2

BEGIN

VAR SYM\_R21: DOUBLE; Data SYM\_R21 (DTYPE\_D)

VAR SYM\_R22: INTEGER; Data SYM\_R22 (DTYPE\_L)

ROUTINE R2A = Routine Begin R2A

BEGIN

VAR SYM\_R2A: BYTE; Data SYM\_R2A (DTYPE\_B)

BEGIN Block Begin (no name)

VAR BLK\_V1: WORD; Data BLK\_V1 (DTYPE\_W)

ROUTINE R2BLKR = Routine Begin R2BLKR

BEGIN

FOO:BEGIN Block Begin FOO

VAR FOO\_V:REAL; Data FOO\_V (DTYPE\_F)

END; Block End (for FOO)

VAR R2BLK\_V2:REAL; Data R2BLK\_V2 (DTYPE\_F)

END; Routine End (for R2BLKR)

VAR BLK\_V2: DOUBLE; Data BLK\_V2 (DTYPE\_D)

END; Block End (for no name)

END; Routine End (for R2A)

VAR SYM\_R23: REAL; Data SYM\_R23 (DTYPE\_F)

END; Routine End (for R2)

END; Module End

За объявление базовых типов (не структур) отвечают записи Type Specification DST Record.

У каждой Type записи есть общая часть - это общий для всех записей заголовок, название типа (строка), общая длина записи, а также kind. В зависимости от этого kind происходит дальнейшая обработка.

Всего определено 32 kind:

DST$K\_TS\_ATOM, { 1, Atomic Type Spec

DST$K\_TS\_DSC, { 2, VAX Standard Desciptor Type Spec

DST$K\_TS\_IND, { 3, Indirect Type Spec

DST$K\_TS\_TPTR, { 4, Typed Pointer Type Spec

DST$K\_TS\_PTR, { 5, Pointer Type Spec

DST$K\_TS\_PIC, { 6, Pictured Type Spec

DST$K\_TS\_ARRAY, { 7, Array Type Spec

DST$K\_TS\_SET, { 8, Set Type Spec

DST$K\_TS\_SUBRANGE, { 9, Subrange Type Spec

DST$K\_TS\_ADA\_DSC, { 10, ADA Descriptor Type Spec

DST$K\_TS\_FILE, { 11, File Type Spec

DST$K\_TS\_AREA, { 12, Area Type Spec (PL/I)

DST$K\_TS\_OFFSET, { 13, Offset Type Spec (PL/I)

DST$K\_TS\_NOV\_LENG, { 14, Novel Length Type Spec

DST$K\_TS\_IND\_TSPEC, { 15, DEBUG internally generated pointer to

{ Type Spec (cannot appear in DST)

DST$K\_TS\_SELF\_REL\_LABEL,

{ 16, Self-Relative Label Type Spec (PL/I)

DST$K\_TS\_RFA, { 17, Record File Address Type Spec (BASIC)

DST$K\_TS\_TASK, { 18, Task Type Spec (ADA)

DST$K\_TS\_ADA\_ARRAY, { 19, ADA Array Type Spec

DST$K\_TS\_XMOD\_IND, { 20, Cross-Module Indirect Type Spec

DST$K\_TS\_CONSTRAINED, { 21, Constrained Type Spec (ADA)

DST$K\_TS\_MAYBE\_CONSTR, { 22, Might-be-constrained Type Spec (ADA)

DST$K\_TS\_DYN\_NOV\_LENG, { 23, Dynamic Novel Length Type Spec

DST$K\_TS\_TPTR\_D, { 24, Typed Pointer to descriptor

DST$K\_TS\_SCAN\_TREE, { 25, SCAN Tree Type Spec

DST$K\_TS\_SCAN\_TREEPTR, { 26, SCAN Treeptr Type Spec

DST$K\_TS\_INCOMPLETE, { 27, Incomplete Type Spec

DST$K\_TS\_BLISS\_BLOCK, { 28, Bliss Block DST

DST$K\_TS\_TPTR\_64, { 29, Typed 64-bit Pointer Type Spec

DST$K\_TS\_PTR\_64, { 30, 64-bit Pointer Type Spec

DST$K\_TS\_REF, { 31, C++ Reference Type

DST$K\_TS\_REF\_64 { 32, C++ Reference Type, 64-bit pointer

Можете заметить, что поддерживаются и специфичные для некоторых ЯП типы. Поддерживаемых языков много:

* Macro
* Macro64
* Fortran
* Bliss
* Cobol
* Basic
* PL/I
* Pascal
* C
* RPG
* Ada
* Scan
* Dibol
* Modula
* Pillar
* C++
* Amacro

Но я отвлекся. Вернемся к информации о типах.

В отличие от предыдущих форматов, здесь информация о базовых типах предопределена. Базовый тип представляет запись DST$K\_TS\_ATOM - Atomic type. Все что эта запись хранит - байт типа. Они уже определены:

DSC$K\_DTYPE\_Z = 0, { Unspecified (May not appear in DST).

DSC$K\_DTYPE\_V = 1, { Bit.

DSC$K\_DTYPE\_BU = 2, { Byte logical.

DSC$K\_DTYPE\_WU = 3, { Word logical.

DSC$K\_DTYPE\_LU = 4, { Longword logical.

DSC$K\_DTYPE\_QU = 5, { Quadword logical.

DSC$K\_DTYPE\_B = 6, { Byte integer.

DSC$K\_DTYPE\_W = 7, { Word integer.

DSC$K\_DTYPE\_L = 8, { Longword integer.

DSC$K\_DTYPE\_Q = 9, { Quadword integer.

DSC$K\_DTYPE\_F = 10, { Single-precision floating.

DSC$K\_DTYPE\_D = 11, { Double-precision floating.

DSC$K\_DTYPE\_FC = 12, { Complex.

DSC$K\_DTYPE\_DC = 13, { Double-precision Complex.

DSC$K\_DTYPE\_T = 14, { ASCII text string.

DSC$K\_DTYPE\_NU = 15, { Numeric string, unsigned.

DSC$K\_DTYPE\_NL = 16, { Numeric string, left separate sign.

DSC$K\_DTYPE\_NLO = 17, { Numeric string, left overpunched sign.

DSC$K\_DTYPE\_NR = 18, { Numeric string, right separate sign.

DSC$K\_DTYPE\_NRO = 19, { Numeric string, right overpunched sign

DSC$K\_DTYPE\_NZ = 20, { Numeric string, zoned sign.

DSC$K\_DTYPE\_P = 21, { Packed decimal string.

DSC$K\_DTYPE\_ZI = 22, { Sequence of instructions.

DSC$K\_DTYPE\_ZEM = 23, { Procedure entry mask.

DSC$K\_DTYPE\_DSC = 24, { Descriptor, used for arrays of

{ dynamic strings

DSC$K\_DTYPE\_OU = 25, { Octaword logical

DSC$K\_DTYPE\_O = 26, { Octaword integer

DSC$K\_DTYPE\_G = 27, { Double precision G floating, 64 bit

DSC$K\_DTYPE\_H = 28, { Quadruple precision floating, 128 bit

DSC$K\_DTYPE\_GC = 29, { Double precision complex, G floating

DSC$K\_DTYPE\_HC = 30, { Quadruple precision complex, H floating

DSC$K\_DTYPE\_CIT = 31, { COBOL intermediate temporary

DSC$K\_DTYPE\_BPV = 32, { Bound Procedure Value

DSC$K\_DTYPE\_BLV = 33, { Bound Label Value

DSC$K\_DTYPE\_VU = 34, { Bit Unaligned

DSC$K\_DTYPE\_ADT = 35, { Absolute Date-Time

= 36, { Unused (not supported by DEBUG)

DSC$K\_DTYPE\_VT = 37, { Varying Text

DSC$K\_DTYPE\_T2 = 38, { 16-bit char

DSC$K\_DTYPE\_VT2 = 39, { 16-bit varying char

Таким образом, int32 мы представим такой записью:

|  |  |  |  |
| --- | --- | --- | --- |
| **Type** | **Length** | **Type Spec Kind** | **Atomic Type** |
| DST$K\_TYPSPEC | 2 | DST$K\_TS\_ATOM | DSC$K\_DTYPE\_LU |

Теперь, перейдем к представлении структур. Если посмотрите на значения Type specification, то не найдете упоминания о структурах. Для определения структур используется другая запись - Record. Причем это не 1 запись, а 2 - Begin и End, а ее поля хранятся между ними. Поля структур представляют записи Data DST Record (их несколько разновидностей).

В заголовке хранится общая информация: название структуры, флаги. Но количество полей не хранится. Во-первых, мы и так сможем найти конец структуры с помощью Record End, а, во-вторых, кроме записей полей могут содержаться записи объявления типа.

В документации дано такое представление записей структуры:

Program Structure DST Record Sequence

----------------- -------------------

Data REC1 (SepTypSpec)

TYPE RECTYP = Record Begin (RECTYP)

RECORD OF

COMP1: INTEGER; Data COMP1 (DTYPE\_L)

COMP2: REAL; Data COMP2 (DTYPE\_F)

COMP3: DOUBLE; Data COMP3 (DTYPE\_D)

END; Record End (for RECTYP)

VAR REC1: RECTYP;

VAR REC2: RECTYP; Data REC2 (SepTypSpec)

Type Spec DST record

(Indirect Type Spec

pointing to RECTYP)

Из этой схемы становится понятно, что:

1. Тип структуры указывается при необходимости прямо после определения переменной
2. Он определяется единожды, а затем мы ссылаемся на него (Indirect Type Spec)

Record Begin рассмотрели, Record End - это просто маркер (там ничего нет), поэтому рассмотрим поближе Data запись. Одно из таких значений Data записей - Standard Data DST. Также можете заметить, что она используется и для определения переменных. Хранит в себе Atomic типы (примитивные) и состоит из названия (название поля или переменной), kind и полей расположения (регистр и смещение). Для определения базового типа это достаточно.

Здесь есть 3 Standard Data DST: COMP1, COMP2 и COMP3. Их типы указываются в скобках - Long, Float, Double.

Но вот если тип этой переменной сложный (читай структура), то используется уже другая Data запись - Separate Type Specification DST Record. В ней также имеется поле для названия, но вот внутри себя она хранит не код встроенного типа, а ничего. Если мы встретили структуру, то дальше должно идти определение этой структуры. Это мы видим в переменной REC1 - переменная определяется с помощью Seperate Type Specification, а после идет само определение структуры.

Чтобы избавиться от постоянного дублирования используется следующий подход - мы определяем структуру единожды, а далее просто ссылаемся на нее. Это используется во второй переменной REC2. В таком случае, после Separate Type Specification будет идти запись типа Indirect Type Specification (для нее имеется константа DST$K\_TS\_IND). Внутри себя она имеет единственное поле DST$L\_TS\_IND\_PTR - байтовое смещение с начала всего DST на нужную запись (такой, персистентый указатель).

С типам закончили, давайте рассмотрим механизм хранения строк исходного кода.

За это отвечает Line Number PC-Corrlation DST Record. Внутри себя она хранит массив не пар Line Number - PC, а команды. Эти команды манипулируют состоянием машины состояний. Сами команды имеют разный размер: первым байтом всегда идет opcode (код операции), а далее 1, 2 или 4 байта параметр. Также отдельно стоит специальный opcode - отрицательный. Тогда, это 1 байтная команда "увеличить PC на указанное значение байтов" (абсолютное значение этого opcode) и параметров не передается. Всего команд, исключая отрицательную ранее, 21:

DST$K\_DELTA\_PC\_W, { 1, Delta-PC Word command

DST$K\_INCR\_LINUM, { 2, Increment Line Number Byte command

DST$K\_INCR\_LINUM\_W, { 3, Increment Line Number Word command

DST$K\_SET\_LINUM\_INCR, { 4, Set Line Number Increment Byte command

DST$K\_SET\_LINUM\_INCR\_W, { 5, Set Line Number Increment Word command

DST$K\_RESET\_LINUM\_INCR, { 6, Reset Line Number Increment command

DST$K\_BEG\_STMT\_MODE, { 7, Begin Statement Mode command

DST$K\_END\_STMT\_MODE, { 8, End Statement Mode command

DST$K\_SET\_LINUM, { 9, Set Line Number Word command

DST$K\_SET\_PC, { 10, Set Relative PC Byte command

DST$K\_SET\_PC\_W, { 11, Set Relative PC Word command

DST$K\_SET\_PC\_L, { 12, Set Relative PC Longword command

DST$K\_SET\_STMTNUM, { 13, Set Statement Number Byte command

DST$K\_TERM, { 14, Terminate Line Byte command

DST$K\_TERM\_W, { 15, Terminate Line Word command

DST$K\_SET\_ABS\_PC, { 16, Set Absolute PC Longword command

DST$K\_DELTA\_PC\_L, { 17, Delta-PC Longword command

DST$K\_INCR\_LINUM\_L, { 18, Increment Line Number Longword command

DST$K\_SET\_LINUM\_B, { 19, Set Line Number Byte command

DST$K\_SET\_LINUM\_L, { 20, Set Line Number Longword command

DST$K\_TERM\_L { 21, Terminate Line Longword command

Состояние, которым они манипулируют состоят из 7 переменных. Основными можно назвать CURRENT\_LINE (текущая строка) и CURRENT\_PC (текущий PC). Также есть 2 режима работы, которые зависят от флага CURRENT\_STMT\_MODE: если он выставлен, то мы как бы говорим, что выполняем очередной стейтмент, но при этом остаемся на текущей строке. Это полезно, для ЯП, поддерживающих возможность указания нескольких стейтментов в 1 строке (например, в C их можно указать через ;).

CodeView

Источников информации совсем немного. Изначально, эту секцию я писал ориентируясь на [Microsoft Symbol and Type Information](https://openwatcom.org/ftp/devel/docs/CodeView.pdf) (далее MSTI). Но только когда дописал, то понял, что этот документ вышел в 199x году (скорее всего 1993).  
После было несколько попыток поиска актуальной документации. В результате я остановился на исходном коде для LLVM ([код](https://github.com/llvm/llvm-project/tree/0b0874755d4f339ae3ef6a43421405ebe9d645f3/llvm/lib/DebugInfo/CodeView), [заголовочные файлы](https://github.com/llvm/llvm-project/tree/0b0874755d4f339ae3ef6a43421405ebe9d645f3/llvm/include/llvm/DebugInfo/CodeView)).  
Далее будет то, что нашел в исходных кодах LLVM и microsoft-pdb. Иногда будут замечания по поводу MSTI.

Здесь мы приходим в мир Windows. Если вы ранее разрабатывали для него (или слышали), то знаете про файлы с расширением .pdb - это файлы, которые хранят символы отладки. Но они не определяют формат хранимых данных. Опять же мы говорим о Windows, поэтому с большей долью вероятности там будет находиться CodeView.

[CodeView](https://en.wikipedia.org/wiki/CodeView) - это сам формат отладочных символов, а PDB - это контейнер для отладочных символов (не обязательно CodeView). Для начала познакомимся с PDB.

PDB, Program DataBase - это файл с отладочными символами. Он используется в Windows и в нем отладочные символы хранятся отдельно от самого исполняемого файла.

PDB имеет разметку, которая называется MSF (Multi-Stream Format), то есть представляет из себя множество потоков, где каждый поток может быть поделен на несколько страниц (например, если поток большой, то удобно его разделить на несколько отдельных файлов). Такой формат называют "file system within a file" (поток - это "файл", чанки - это "страница"). Этот формат описан на [LLVM](https://llvm.org/docs/PDB/MsfFile.html).

О CodeView в MSTI говорится о том, что он нужен только для символов и типов, но в LLVM он используется и для исходного кода. Этот формат используется в экосистеме Windows - компиляторы Visual C++/C#/VB.NET, Visual Studio debugger, WinDbg.

CodeView и PDB файлы тесно связаны. Поэтому отделить одно от другого трудно.

Его спецификация никогда не публиковалась, но [в 2015 году решили подсобить LLVM](https://lists.llvm.org/pipermail/cfe-dev/2015-October/045780.html) и дали первую часть - информация о типах. Далее, создали [репозиторий](https://github.com/microsoft/microsoft-pdb), но он заархивирован год назад, а последние изменения в нем были 8 лет назад. На основании кода в этом репозитории пишут свой код уже в LLVM. Часто перед какой-либо структурой можно заметить комментарий по типу Corresponds to the CV\_... struct.

Хранить отладочную информацию мы можем в виде встраиваемых символов и PDB потоков. В обоих случаях, различается только способ доступа к информации, а сама эта информация имеет одинаковый формат.

Если CodeView встраивается, то поддерживаются 2 формата объектных файлов: OMF и COFF. В каждом разное наименование целевых секций (из MSTI):

* COFF (секции):
  + Типы - .debug$T
  + Символы - .debug$S
* OMF (записи LEDATA)
  + Типы - $$TYPES
  + Символы - $$SYMBOLS

Для них есть потоки в PDB:

* типы:
  + TPI Stream
  + IPI Stream
* символы:
  + Public Symbol Stream
  + Global Symbol Stream
  + Module Info Stream

Таблица символов - описывает символы. Таблица типов - типы для этих символов. Причем, символы могут ссылаться на типы, а типы на себя же.

В секции типов хранятся как примитивные (int, char, double), так и пользовательские (typedef, struct) типы. Информация о них хранится в виде ацикличного ориентированного графа (DAG):

* Узлы - сами типы
* Ребра - отношение включения типа (использует/ссылается)

У каждого типа есть свой индекс (считай идентификатор). Если нужно сослаться на какой-то тип, то используется он. Причем, для наиболее используемых, встроенных типов зарезервированы индексы до 4096 (0x1000), а 0 - особое значение "no type" (используется как NULL). Назначение индексов должно быть последовательным, а пропуски запрещены. То есть, индексы идут 0x1000, 0x1001, 0x1002 ... а 0x1000, 0x1003, 0x1004 запрещено.

Главная проблема здесь - это циклы. Например, можно сразу определить структуру, у которой поле - ссылка на саму себя. Эта проблема решается тем, что используется предварительное объявление. В [этом видео](https://youtu.be/5twzd06NqGU?si=rr6cZW7Tw1pnsUGY&t=590) более подробно описывается алгоритм работы с типами и описывается этот DAG.

Как это обрабатывается описано в [этих слайдах](https://www.llvm.org/devmtg/2016-11/Slides/Kleckner-CodeViewInLLVM.pdf). Вот отрывок:

0x1000: struct Foo;

0x1001: Foo\* # <0x1000>

0x1003: { Foo \*p; }; # <0x1001>

0x1004: struct Foo <0x1003>

Для примитивных типов индексы определены. Вот отрывок кода из LLVM с определением этих индексов:

Индексы простых типов

Для оптимизации производительности используется выравнивание записей. Если оно включено, то в начале таблицы добавляется заголовок 0x00000001.

Хранение типов реализовано с помощью DAG (направленного ацикличного графа) - это отражается и в том как представляются типы. У нас также есть представление дерева. Каждый узел этого дерева (структура) хранит в себе поле leaf - перечисление LF\_.... Оно говорит о том, что перед нами за структура. Возможные значения можно посмотреть [здесь](https://github.com/llvm/llvm-project/blob/18225c783a00bde62f19a177a57de388e20c2bba/llvm/include/llvm/DebugInfo/CodeView/CodeViewTypes.def).

Дальнейшее содержимое зависит от типа узла. Все leaf можно разделить на 4 группы:

1. Записи, на которые могут ссылаться из таблицы символов
   * LF\_MODIFIER - модификатор типа (const, volatile, unaligned ...)
   * LF\_POINTER - типизированный указатель
   * LF\_CLASS - класс
   * LF\_STRUCTURE - структура
   * LF\_PROCEDURE - функции
2. Записи, на которые НЕ могут ссылаться из таблицы символов, но могут из других записей
   * LF\_ARGLIST - аргументы функции
   * LF\_DEFARG - значение функции по умолчанию
   * LF\_FIELDLIST - список из полей структуры/класса/объединения/перечисления
   * LF\_BITFIELD - битовое поле
3. Используется для создания сложных структур
   * LF\_BCLASS - базовый класс
   * LF\_MEMBER - поле класса
   * LF\_METHOD - функция класса (как перегруженная, так и нет)
   * LF\_NESTTYPE - вложенный тип
   * LF\_FRIENDCLS - дружественный (friend) класс
4. Используется для хранения числовых значений. Здесь применили интересный подход: если при обходе записей мы знаем, что дальше должно быть значение, то мы смотрим на следующие 2 байта:
   * Если эти 2 байта меньше 0x8000 - то это нужное нам значение (константа)
   * В противном случае, мы интерпретируем это число как leaf:
     + LF\_CHAR - 1 байт
     + LF\_SHORT - 2 байта
     + LF\_LONG - 4 байта
     + LF\_REAL32 - 4 байтное с плавающей точкой

Предопределенные примитивные типы имеют индексы меньше 4096, т.е. 0x0FFF и меньше. Их индексы имеют такую маску:

|  |  |
| --- | --- |
| **Режим указателя** | **Тип** |
| 3 бита | 16 бит |

Режим указателя - какой тип указателя представляет:

* Не указатель
* Near pointer
* Far pointer
* Huge pointer

Тип - один из примитивных типов:

* Целочисленное знаковое
* Целочисленные беззнаковое
* Bool
* С плавающей точкой
* Комплексное
* Особый (special)

В MSTI другая разметка примитивного типа. Например, в ней есть поле размер. Оно интерпретируется в зависимости от типа:

* Целочисленные и bool - размер в байтах
* С плавающей точкой - битность (32, 64, 80, 128, 48)
* Особые значения:
  + Абсолютный символ
  + Сегмент
  + Void
  + Валюта

Для представления класса/структуры/интерфейса используется следующая запись:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Индекс** | **Кол-во полей** | **Свойства** | **Индекс списка** | **Наследует** | **V-Table** | **Размер** | **Название** |
| LF\_CLASS | int16 | int16 | int32 | int32 | int32 | int64 | string |

Поля:

* Индекс - тип записи (маркер). В примере указан LF\_CLASS, но может быть LF\_STRUCTURE и LF\_INTERFACE
* Кол-во полей - сколько полей хранится в классе
* Свойства - битовая маска свойств/характеристик класса. Например, вложенность, имеет конструктор/деструктор или перегруженный оператор
* Индекс списка - индекс списка (LF\_FIELDLIST), который описывает каждое поле структуры (то есть эта информация хранится отдельно)
* Наследует - индекс списка, который описывает другие классы, от которых этот класс наследуется
* V-Table - индекс виртуальной таблицы
* Размер - размер класса в байтах
* Название - название этого класса

Под символами имеются ввиду не только символы в понимании ассемблера (метки функций и т.д.), но грубо говоря, все что не типы. Например, имеется символ S\_UNAMESAPCE - using namespace или S\_COMPILE - флаги компиляции. Хранение информации происходит также иерархично, а для оптимизации поиска, записи, которые начинают область имеют ссылки на своих родителей (чтобы не читать все записи).

Каждая запись для символа имеет общий заголовок: длина + тип символа (индекс). Дальнейшее содержимое зависит от флага. Индексы можно разделить на 5 групп (примеры приведены):

1. Символы, формат которых не меняется в зависимости от окружения: символы регистров, символы пользовательских типов и т.д.
   * Флаги компиляции
   * Константы и перечисления
   * Название объектных файлов
   * Пользовательские типы
2. Символы для архитектур с сегментированным режимом адресации 16:16
   * Переменные, хранящиеся на стеке
   * Не экспортируемые символы
   * Глобальные и static функции
   * Области видимости (начало и конец блоков)
3. Символы для архитектур с сегментированным режимом адресации 16:32 (примеры те же)
4. Символы, специфичные для MIPS архитектуры или компилятора
   * Глобальные и static функции
5. Для CodeView оптимизации
   * Глобальные хэш таблицы для функций и переменных

Для хранения переменных используется несколько индексов - в зависимости от хранилища (регистр или стек) и битности (16, 32 и т.д.). Для представления переменной, хранящейся на стеке используется следующая запись:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Длина** | **Индекс** | **Смещение** | **Тип** | **Название** |
| int32 | S\_BPREL | int32 | int32 | string |

Поля:

* Длина - общая длина записи
* Индекс - индекс/маркер записи, равно S\_BPREL
* Смещение - смещение относительно регистра BP (EBP/RBP ...), где хранится переменная
* Тип - индекс типа из таблицы типов
* Название - название переменной (в 1 байте - длина строки)

Для получения таблицы строк исходного кода используется отдельная таблица. Если это PDB, то используется поток DBI - Debug Information. В нем содержится множество подпотоков. Информация об исходных файлах хранится иерархично: модуль -> файл -> сегменты. И так для каждого объектного модуля. Можно представить так:

* Модуль 1
  + Файл 1
    - Сегмент 1
    - Сегмент 2
  + Файл 2
    - Сегмент 1
    - Сегмент 2

Сама информация об отображении исходного кода на инструкции хранится в Debug Lines Subsection. Она состоит из 2 частей:

* Заголовок
* Список LineBlockFragment

В заголовке нам интересно только 1 поле - флаги. Один из флагов - наличие информации о столбцах в исходном коде. Таким образом мы можем хранить информацию о нескольких стейтментах в одной строке.

Сама информация по отображению исходного кода на инструкции хранится в LineBlockFragment - элементах списка. Каждый такой элемент хранит в себе 3 поля:

1. NameIndex - индекс файла, который этот блок описывает
2. NumLines - размеры следующих массивов (кол-во строк, которые этот блок хранит в себе)
3. Массив LineNumberEntry - отображение строки исходного кода на машинные инструкции
4. Массив ColumnNumberEntry - дополнительная информация о столбцах

3 и 4 - это параллельные массивы, причем 4 может отсутствовать - необходимо проверять флаг из заголовка (сказал ранее).

Каждый LineNumberEntry - это:

* Offset - адрес инструкции
* Start:24 - номер строки
* End:7 - размер строки (сколько строк занимает), может отсутствовать
* IsStatement:1 - это стейтмент или выражение

Последние 3 поля хранятся в виде битовых полей, в общем они занимают 4 бита, а вся запись в общем 8 байт.

ADATA

[ADATA](https://github.com/ashenBlade/habr-posts/blob/post/how-debuggers-work/how-debuggers-work/HLASM_ADATA.pdf) - это формат отладочной информации, изобретенный специально для [HLASM](https://en.wikipedia.org/wiki/High-level_assembler) (High Level Assembler), ассемблема IBM. Этот ассемблер имеет некоторые конструкции, специфичные для высокоуровневых языков, например, поддержка переменных, использование выражений в качестве операндов (не нужны отдельные инструкции для их вычислений), различные строковые операции (конкатенация, подстрока, длина) и т.д.

Вся отладочная информация хранится в отдельном файле - SYSADATA. Он создается во время компиляции.

Состоит этот файл из последовательности записей. Каждая запись имеет 12 байтный заголовок:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **Lanuage Code** | **Record Type** | **Architecture Level** | **Flag** | **Edition** | **Reserved** |
| Размер, байты | 4 | 2 | 4 | 1 | 4 | 1 |

Поля:

* Language Code - код исходного языка, пока константа = 16
* Record Type - тип записи
* Architecture Level - уровень архитектуры, в этом случае - версия заголовка
* Flag - различные флаги свойств: имеется ли продолжение, endianess ...
* Edition - версия самой записи (содержимого)
* Reserved - зарезервированная область

Есть множество типов записей, но нам будут интересны следующие:

* Source Analysis Record
* Machine Instruction Record
* Symbol Record
* DC/DS Record

Для создания таблицы исходного кода используются записи Source Analysis Record и Machine Instruction Record. Здесь стоит вспомнить, что формат создан для "высокоуровневого" ассемблера. Один из аспектов высокоуровневости - поддержка макросов, они могут создавать более 1 инструкции, поэтому необходимо знать соответствие исходного кода и ассемблера.

Source Analysis Record позволяет нам сопоставить исходный код с инструкциями. Эта запись позволяет сопоставлять 1 строку исходного кода с инструкциями.

Поэтому, эта запись, во-первых, имеет смысл, во-вторых, имеет множество подтипов:

* Комментарий - описывает строку комментария
* Инструкция - описывает строку, которая 1 на 1 отображается в инструкцию
* Стейтмент - описывает строку, которая может отобразиться на несколько ассемблерных инструкций. Это можно назвать стейтментом, а пример - условные инструкции, например, условное копирование
* Вызов макроса - вызов макроса
* Определение макроса - описывает реализацию макроса

Эта запись генерируется для каждой строки в каждом файле исходников. Формат записи одинаковый независимо от типа строки, поэтому некоторые поля могут не использоваться. Сам адрес инструкции хранится в поле Location Counter - IBM-специфичный аналог PC.

Также есть Machine Instruction Record - запись, которая содержит сгенерированный машинный код. Она идет после Source Analysis Record, если инструкции были сгенерированы. Таким образом, нам необходимо просто прочитать все Source Analysis записи и получить из них пары строка файла - Location Counter.

Также стоит отметить, что обе записи в себе содержат копию исходного кода или ассемблерных инструкций, поэтому они самодостаточны, т.е. можно воссоздать весь исходник с помощью файла отладочной информации

Другая запись - Symbol Record. Она хранит в себе информацию о символе.

Еще один аспект "высокоуровневости" - переменные. Для их создания используется отдельный стейтмент - DS, Define Storage. Переменная тоже является символом, поэтому будет включена в эту запись. HLASM - это ассемблер, а машина о типах не знает, но с другой стороны, это высокоуровневый ассемблер, поэтому поддержка типов есть, хотя и ограниченная.

За каждым символом закреплен свой Type attribute - его тип. Это перечисление, указывающее на тип символа. Обычно, оно выражается с помощью символа. [Здесь](https://faculty.cs.niu.edu/~hutchins/csci360/pdf/lang-ref.pdf) я нашел их список и разбил таким образом:

* Числа с плавающей точкой:
  + D - long floating-point
  + E - short floating-point
* Числа, с фиксированной точкой:
  + F, fullword fixed point
  + G - fixed point
  + H - half fixed point
* Адрес (тип адреса):
  + A - A-type, J-type
  + S - S-type
  + R - A, S, Q, J, R, V, Y-type
* Числовые значения:
  + B - binary (может использоваться как bool)
  + C - character
  + F - floating point
  + X - hexadecimal
  + Z - zoned decimal
  + @ - graphic
  + P - packed decimal
  + N - numeric
* Специальные:
  + U - undefined/unknown/unassigned
  + O - omitted

Эта информация хранится в записи DC/DS, то есть рядом с символом. Там же находится и другая вспомогательная информация, необходимая для уточнения типа, например, длина.

Все эти записи хранятся в файле SYSADATA, который генерируется во время компиляции. В документации он называется Associated Data File. Записи в файле организованы следующим образом.

* Compilation Unit Start
* Вспомогательные записи
* Основные записи
* Compilation Unit End

Compilation Unit Start/End - это записи обозначающие границы единицы компиляции. Вся информация по каждому CU хранится внутри них. В начале имеется несколько вспомогательных записей. Они хранят некоторые метаданные, например, Options Record - показывает с какими флагами была запущена компиляция. Дальше идут уже записи для отладки. Source Analysis, Machine Instruction и Symbol записи хранятся здесь.

В примерах, все записи были в порядке следования стейтментов исходного кода, но в документации я не нашел должен ли быть такой порядок. Как минимум, есть очерки, что после Source Analysis должен идти Machine Instruction

IEEE-695

[IEEE-695](https://github.com/ashenBlade/habr-posts/blob/post/how-debuggers-work/how-debuggers-work/IEEE695.pdf) - это формат объектного модуля, разработанный для встраиваемых систем (MUFOM - Microprocessor Universal for Object Modules). Но также как и OMF здесь есть поддержка отладочных символов.

Это разработка IEEE и она платная, но, так как это спецификация, то есть множество реализаций - документация для них есть в свободном доступе. Например, реализация от Microtec и HP (ссылка ранее). Я буду считать ее референсной, так как именно они (компании) положили основу формату и создали пропозал в IEEE.  
Дальше будет описание именно их реализации.

Также я [нашел](https://github.com/ashenBlade/habr-posts/blob/post/how-debuggers-work/how-debuggers-work/IEEE695-proposed.pdf) один из первых пропозалов. Уже не помню где, но главное что можно отметить - это способ хранения данных. Он текстовый, не бинарный. Это было сделано для того, чтобы сделать объектные модули как можно более переносимыми. Но вот после добавили поддержку бинарного формата и с этих пор допускаются 2 вида. Я буду говорить о бинарном.

Отладочная информация хранится в отдельной части модуля - Debug Information Part, которая состоит из записей.

Информация о типах хранится в записи TY - Define Types. Формат этой записи

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Маркер** | **Индекс типа** | **Маркер** | **Индекс названия** | **Код** |
| 0xF2 | число | 0xCE | число | число |
| Константа | Не меньше 256 | Константа | Не меньше 32 | Перечисление |

* Есть 2 поля маркер - это константы. Если первый указывает на тип самой записи, то зачем 2-ой маркер - не знаю.
* Индекс типа - это индекс этого типа в глобальной таблице типов. В этой таблице хранятся все типы, которые используются в программе.
* Индекс названия - это индекс NN записи. NN запись - это грубо говоря пара из числа (глобального индекса) и строки (названия). Ее используют не только для названий типов, например, для переменных или номеров строк.
* Код - это маркер самого типа, дискриминатор.

Есть несколько кодов типов (определяются ASCII символами):

* ! - неизвестный тип
* A - массив
* a - массив из FORTRAN (он в column-major порядке)
* N - перечисление
* E - простое перечисление
* O - small указатели
* P - large указатели
* R - диапазон (range)
* S - структура
* G - структура с битовыми полями
* U - объединение (union)
* X - внешняя процедура (объявление)
* V, v - void
* c, d - комплексное и двойное комплексное
* f - название файла (Pascal)
* g - битовое поле
* n - квалификатор (const, volatile)
* s - множество (set)
* x - процедура с зависимостями компилятора

Дальнейшие поля записи зависят от кода типа. Например, для структуры (S) и для объединения (U) дальше идет массив из тройки полей:

* Название поля - хранится простой строкой
* Индекс типа - индекс в глобальной таблице типов
* Размер типа - размер в байтах, причем для пользовательских типов можно указывать размер в битах (т.е. не кратный байтам).

Таким образом, структура/объединение - это просто массив таких троек.

Информация о структуре программы хранится в отдельной части объектного файла - Debug Information Part. Она содержит в себе блоки. Каждый блок - это информация по какой-то логической части исходной программы, например, определение глобальной функции.

Блоки определяют область (начало и конец), которая может содержать другие записи/блоки. Границы размечаются BB$ и BE$ - Block Begin и Block End, а $ - номер типа:

* 1 - типы в локальной видимости
* 2 - типы в глобальной видимости
* 3 - модуль
* 4 - глобальная функция (подпрограмма)
* 5 - строка исходного кода
* 6 - локальная функция (подпрограмма)
* 10 - дебаг информация для ассемблера
* 20 - информация для библиотеки

Дальше буду говорить *BB$*, вместо *блок типа $* - так короче, да и в документах и коде используется такой подход.

Для включения блоков друг в друга есть правила. Например, BB3 может содержать в себе BB4 (модуль определяет глобальную функцию), но наоборот - нет, что вполне логично. Для получения информации о типах используются BB1 и BB2 блоки. Они содержат TY и NN записи, описанные ранее. Для получения таблицы строк кода используется блок BB5 - File name for source line numbers.

Вот так выглядит заголовок блока BB5:

| Маркер начала | Размер блока в байтах | Название файла | Штамп | Маркер NN | Номер файла | ID файла |  
| ------------- | --------------------- | -------------- | --------- | --------- | ----------- |  
| BB5 | int | string | Timestamp | NN | int | string |

Заметки:

* Размер блока в байтах может быть 0 - значит неизвестен размер
* Название файла - абсолютный путь
* Штамп содержит в себе дату и время компиляции: год, месяц, число, час, минута, секунда
* ID файла в коде везде пустая строка

Последние 3 поля образуют 1 запись NN. В стандарте MRI/HP отмечается, что такая запись должна быть единственной на каждый файл исходного кода, но не могу сказать, есть ли в самом стандарте IEEE такое ограничение.

Дальше идет множество записей о строке исходного кода. Каждая логическая запись состоит из 2 отдельных записей: ATN и ASN.

ATN = id строки -> номер строки в файле

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Маркер ATN** | **Индекс символа** | **Тип символа** | **Дескриптор атрибутов** | **Номер строки** | **Номер столбца** |
| 0xF1CE | int | 0 | 7 | int | 0 |

Замечания:

* Указанные конкретные значения взял из кода. Они специфичны для ATN номеров строк.
* Дескриптор атрибутов можно интерпретировать как подтип ATN - он определяет сколько будет дальше полей и каков их смысл. Для номеров строк используется значение 7, но есть и другие, например, 3 - статическая переменная.
* Номер столбца 0, т.к. в коде нет возможности получить номер столбца, только номер строки.

ASN - id строки -> адрес инструкции

|  |  |  |
| --- | --- | --- |
| **Маркер ASN** | **Индекс строки** | **Адрес инструкции** |
| 0xE2CE | int | int |

Для воссоздания таблицы символов нужно прочитать все записи из BB5 для каждого файла.

Пишем свой отладчик

Чтобы понять, как это все объединить, предлагаю написать свой отладчик. Я назвал его [Dumbugger](https://github.com/ashenBlade/dumbugger), потому что он тупой максимально простой:

* Поддерживается только C
* Функции все простые (inline функции не поддерживаю)
* Из модификаторов только const или typedef (volatile, restrict и остальные не поддерживаю)
* Без макросов

Мы реализуем следующие фичи:

* Точки останова - на функции, на строке файла, на инструкции.
* Шаги по исходному коду - step over, step out, step in.
* Чтение значений переменных (примитивные типы и структуры).
* Получение цепочки вызовов (backtrace).
* Дизассемблирование.
* Чтение и изменение регистров.

Для тестирования функциональности буду использовать простенькую программу на C (смысла в ее работе нет).

#include <stdio.h>  
  
**typedef** **struct** Data {  
 **int** value1;  
 **long** value2;  
} Data;  
  
**int** do\_magic(Data \*data) {  
 **int** temp = data->value1 \* 2;  
 **if** (0 < temp) {  
 data->value2 \*= 3;  
 } **else** {  
 data->value1 -= 1;  
 }  
   
 **return** (temp + 2) / 6;  
}  
  
**int** main(**int** argc, **const** **char** **\*\***argv) {  
 Data data = {  
 .value1 = 3,  
 .value2 = 1  
 };  
 **if** (do\_magic(&data)) {  
 printf("value1 is: %ld", data.value2);  
 }  
 **return** 0;  
}

Запуск

В начале конечно же запуск. Ранее в примерах я уже показывал как запускается программа под отладкой, но то был пример, а сейчас уже реализация. Для запуска процесса для отладки используется функция dmbg\_run. Она форкается и вызывает ptrace (в обеих частях).

/\* https://github.com/ashenBlade/dumbugger/blob/030714733c9544cb77a13aa5b0f1b53175034361/src/dumbugger.c#L262 \*/  
DumbuggerState \*dmbg\_run(**const** **char** **\***prog\_name, **const** **char** **\*\***args) {  
 DumbuggerState \*state;  
  
 /\* Форкаемся \*/  
 **pid\_t** child\_pid = fork();  
 **if** (child\_pid == -1) {  
 free(state);  
 **return** NULL;  
 }  
  
 **if** (child\_pid == 0) {  
 /\* Запускаем main потомка \*/  
 run\_child(args);  
 exit(EXIT\_FAILURE);  
 }  
  
 state->pid = child\_pid;  
 state->wstatus = 0;  
  
 /\* Ждем остановки \*/  
 **if** (waitpid(child\_pid, &state->wstatus, 0) != child\_pid) {  
 kill(child\_pid, SIGKILL);  
 free(state);  
 **return** NULL;  
 }  
  
 **if** (WIFEXITED(state->wstatus) || WIFSIGNALED(state->wstatus)) {  
 /\*  
 \* Первая остановка происходит при запуске execvp.  
 \* Но если это не так и процесс завершился заранее, то код возврата -  
 \* код ошибки  
 \*/  
 errno = WEXITSTATUS(state->wstatus);  
 free(state);  
 **return** NULL;  
 }  
  
 /\* ??? \*/  
  
 **return** state;  
}  
  
**static** **void** run\_child(**const** **char** **\*\***args) {  
 **if** (ptrace(PTRACE\_TRACEME, 0, NULL, NULL) == -1) {  
 exit(errno);  
 }  
  
 execvp(args[0], (**char** **\*const** \*) args);  
 exit(errno);  
}

Как можете видеть это действительно практически тот же код, что и в примерах.

Чтение отладочных символов

В примере выше вы заметили ???? По названию этой секции уже можете догадаться, что речь пойдет про символы отладки. Я их инициализирую сразу, перед передачей управления пользователю.

Отладочная информация хранится в нашем исполняемом файле. Я использую libdwarf и работает он с файлами, поэтому дать ему адресное пространство нашего процесса не получится. Нам придется найти этот файл самим - с помощью procfs. Нам нужно прочитать следующие файлы:

1. /proc/<PID>/exe - путь до исполняемого файла (soft-link)
2. /proc/<PID>/maps - адреса загрузки различных файлов (исполняемых и библиотек)

Это сделать довольно просто: 1 - это прочитать ссылку (readlink), а 2 — спарсить файл. Также у 2 шага стоит заметить, что записи о нашей программе будут скорее всего первыми, но не уверен, что так будет всегда, то есть нельзя просто так возвращать первую прочитанную строку.

В maps мы находим первую строку, файл которой равен прочитанному из 1 шага, — в нем читаем начало диапазона. Например, вот такой кусок этого файла:

62d87e0a9000-62d87e0aa000 r--p 00000000 103:02 59118213 /home/user/a.out

62d87e0aa000-62d87e0ab000 r-xp 00001000 103:02 59118213 /home/user/a.out

62d87e0ab000-62d87e0ac000 r--p 00002000 103:02 59118213 /home/user/a.out

62d87e0ac000-62d87e0ad000 r--p 00002000 103:02 59118213 /home/user/a.out

62d87e0ad000-62d87e0ae000 rw-p 00003000 103:02 59118213 /home/user/a.out

62d87f207000-62d87f228000 rw-p 00000000 00:00 0 [heap]

Загрузочный адрес a.out (имя исполняемого файла) — 0x62d87e0a9000. Зачем он нам? Вспомним, что современные ОС используют ASLR для безопасности. Но если мы посмотрим на символы отладки (с помощью того же dwarfdump, например), то заметим, что там хранятся адреса без этого случайного смещения. Что делать? Просто вычесть этот загрузочный адрес из RIP.

Чтобы прочитать файл, нам потребуется путь до исполняемого файла, полученный на 1 шаге. Передаем этот путь и библиотека инициализируется. Теперь начнем инициализировать свое состояние.

Для начала посмотрим, какая отладочная информация есть в программе. Это можно сделать с помощью утилиты dwarfdump (из того же проекта, что и libdwarf).

Программу я представляю как множество функций, где каждая состоит из:

* Названия
* Расположения (файл с исходным кодом)
* Переменных
  + Название
  + Расположение
  + Тип (далее)
* Диапазона занимаемых адресов

Для простоты поддерживаю 3 типа:

* Примитив (int, long, char)
* Структура
* Указатель (на примитив, структуру или другой указатель)

Другие типы (union, enum), не поддерживаю.

Осталось только собрать эту информацию. Чтобы было проще и понятнее, я часто пользовался dwarfdump - запускал его на собранной программе и изучал вывод вместо того, чтобы искать все по спецификации. Из-за этого, в каких-то моментах могу ошибаться.

Некоторая отладочная информация

Точки останова

В отладчике мы можем указать 3 места, куда точки останова можно ставить:

1. Машинная инструкция
2. Начало функции
3. Строка исходного кода

Как ставить точку останова на машинную инструкцию уже известно - алгоритм описан ранее. Остальные 2 имеют ту же механику (кладем инструкцию 0xCC на место точки останова), но различается способ определения места расположения. Если мы ставим точку останова на инструкцию, то ее адрес нам уже должен быть известен, в противном же случае этот адрес необходимо определить самостоятельно и в обоих случаях необходимы символы отладки.

Чтобы поставить точку останова на функцию, необходимо знать где эта функция начинается. В DWARF для функции указывается ее диапазон: lowpc (начало) и highpc (конец).Нам необходимо поставить точку основа на ее начало, то есть на ее lowpc.

Но нельзя так просто взять и поставить точку останова на lowpc:

1. Не обязательно функция будет занимать *непрерывный* участок памяти. В случае оптимизаций, тело функции может быть разбито на несколько различных участков. DWARF в подобных случаях хранит список занимаемых диапазонов, уже не просто low/highpc.
2. Верхний адрес (lowpc) - не обязательно будет точкой входа, там может находится какой-нибудь другой код, например, для проверок целостности и т.д. . В DWARF поэтому имеется атрибут DW\_AT\_entry\_point - он указывает адрес точки входа и по хорошему точку останова надо ставить туда.
3. У функции могут быть пролог и/или эпилог - вспомогательные участки кода в начале и конце функции, соответственно. Например, они могут использоваться для настройки фрейма (обновление RBP регистра). По хорошему точку останова надо ставить в конце пролога и начале эпилога (вход и выход из функции), иначе можем наткнуться на не до конца инициализированное состояние (например, переменная хранится на стеке с доступом по смещению относительно RBP, но пролог ее еще мог не обновить этот регистр и он будет указывать на старый фрейм). Но я столкнулся с проблемой, что gcc (компилирую им) не добавляет информацию о прологе и эпилоге в DWARF. Поэтому я добавил костыль - когда я достигаю точки останова, то смотрю на 2 дальнейшие инструкции - если это стандартный эпилог (push %rbp; mov %rsp, %rbp;) то дополнительно выполняю 2 инструкции (есть еще endbr64, но проверять его не стал).

Вот так выглядит информация о диапазоне функции в DWARF:

< 1><0x000000c1> DW\_TAG\_subprogram

DW\_AT\_name main

DW\_AT\_type <0x00000058>

DW\_AT\_low\_pc 0x00001199

DW\_AT\_high\_pc <offset-from-lowpc> 80 <highpc: 0x000011e9>

Для целей отладки я компилирую без оптимизаций (-O0), поэтому занимается непрерывный диапазон памяти - highpc - это смещение относительно lowpc.

Чтобы поставить точку останова на строку исходного кода, нам нужен уже другой подход - мы будем использовать таблицу строк исходного кода. Она хранится в секции .debug\_line. Для тестовой программы она будет выглядеть следующим образом (справа добавил исходный код, чтобы было понятнее, dwarfdump этого не показывает):

.debug\_line: line number info for a single cu

Source lines (from CU-DIE at .debug\_info offset 0x0000000c):

NS new statement, BB new basic block, ET end of text sequence

PE prologue end, EB epilogue begin

IS=val ISA number, DI=val discriminator value

<pc> [lno,col] NS BB ET PE EB IS= DI= uri: "filepath"

0x00001139 [ 8,26] NS uri: "/home/user/dumbugger/src/sample.c" | int do\_magic(Data \*data) {

0x00001141 [ 9,20] NS | int temp = data->value1 \* 2;

0x00001147 [ 9, 9] NS | int temp = data->value1 \* 2;

0x0000114c [ 10, 8] NS | if (0 < temp) {

0x00001152 [ 11,13] NS | data->value2 \*= 3;

0x0000115b [ 11,22] NS | data->value2 \*= 3;

0x00001172 [ 13,13] NS | data->value1 -= 1;

0x00001178 [ 13,22] NS | data->value1 -= 1;

0x00001181 [ 16,18] NS | return (temp + 2) / 6;

0x00001187 [ 16,23] NS | return (temp + 2) / 6;

0x0000119a [ 16,23] DI=0x1 | return (temp + 2) / 6;

0x0000119c [ 17, 1] NS | }

0x0000119e [ 19,39] NS | int main(int argc, const char \*\*argv) {

0x000011ad [ 20,10] NS | Data data = {

0x000011c1 [ 24, 9] NS | if (do\_magic(&data)) {

0x000011cd [ 24, 8] NS DI=0x1 | if (do\_magic(&data)) {

0x000011d1 [ 25, 9] NS | printf("value1 is: %d", data.value2);

0x000011ee [ 27,12] NS | return 0;

0x000011f3 [ 28, 1] NS | }

0x000011f5 [ 28, 1] NS ET | }

Так как мы работаем с уже готовой отладочной информацией, а не с исходным кодом, то можно понять, что точки останова мы можем ставить указывая название файла и номер строки (возможно с номером столбца), но не номер строки внутри функции, так как работаем с простым текстом.

Логика выставления точки останова находится в dmbg\_set\_breakpoint\_function:

/\* https://github.com/ashenBlade/dumbugger/blob/030714733c9544cb77a13aa5b0f1b53175034361/src/dumbugger.c#L1293 \*/  
**int** dmbg\_set\_breakpoint\_function(DumbuggerState \*state, **const** **char** **\***function) {  
 STOPPED\_PROCESS\_GUARD(state);  
  
 **if** (function == NULL) {  
 errno = ENOENT;  
 **return** -1;  
 }  
  
 FunctionInfo \*func;  
 **int** res = debug\_syms\_get\_function\_by\_name(state->debug\_info, function, &func);  
 **if** (res == -1) {  
 **return** -1;  
 }  
  
 **if** (res == 0) {  
 errno = ENOENT;  
 **return** -1;  
 }  
  
 **return** set\_breakpoint\_at\_addr(state, func->low\_pc - state->load\_addr);  
}

Все что делаем - находим lowpc переданной функции, а затем выставляем точку останова по этому адресу (с учетом адреса загрузки). Но основная логика конечно же в функции set\_breakpoint\_at\_addr:

/\* https://github.com/ashenBlade/dumbugger/blob/030714733c9544cb77a13aa5b0f1b53175034361/src/dumbugger.c#L1264 \*/  
**static** **int** set\_breakpoint\_at\_addr(DumbuggerState \*state, **long** addr) {  
 **long** text;  
 /\*   
 \* Читаем инструкцию по переданному адресу  
 \*/  
 **if** (peek\_text(state, addr, &text) == -1) {  
 **return** -1;  
 }  
  
 /\*   
 \* Сохраняем прочитанную инструкцию, чтобы восстановить  
 \*/  
 breakpoint\_info bi = {  
 .address = addr,  
 .saved\_text = text,  
 };  
  
 /\*   
 \* Заменяем байт инструкции на 0xCC - точку останова x86  
 \*/  
 **long** new\_text = (text & ~0xFF) | ((**long**) 0xCC);  
 **if** (poke\_text(state, addr, new\_text) == -1) {  
 **return** -1;  
 }  
  
 /\*   
 \* Сохраняем эту точку останова  
 \*/  
 **if** (dumbugger\_state\_add\_breakpoint(state, &bi) == -1) {  
 **return** -1;  
 }  
  
 **return** 0;  
}

У меня получилась интересная ситуация, касательно адреса загрузки. Я решил скомпилировать gcc вручную, но не указал флаг --enable-default-pie. Из-за этого, при компиляции у меня создавались бинари без PIE и ASLR не применялся. В результате, адрес RIP совпадал с адресом из DWARF и, когда я пытался вызвать ptrace с измененным адресом (вычитание адреса загрузки), то валился с EIO, так как получал невалидный адрес.

Ранее сказал, что нужно пропускать пролог функции, а здесь его нет. Как же так? В своей реализации эту проверку я делаю после срабатывания точки останова. Для обнаружения читаю инструкцию по текущему адресу и если обнаружил, то просто выполняю 2 инструкции:

#define IS\_FRAMESETUP(text) (((text) & 0xFFFFFFFF) == 0xe5894855)  
  
/\* https://github.com/ashenBlade/dumbugger/blob/030714733c9544cb77a13aa5b0f1b53175034361/src/dumbugger.c#L468 \*/  
**static** **int** skip\_frame\_setup(DumbuggerState \*state, **long** rip) {  
 FunctionInfo \*cur\_func;  
 **if** (debug\_syms\_get\_function\_at\_addr(state->debug\_info, rip - state->load\_addr, &cur\_func) == -1) {  
 **if** (errno == ENOENT) {  
 **return** 0;  
 }  
 **return** -1;  
 }  
  
 **if** (cur\_func->low\_pc != rip - state->load\_addr) {  
 **return** 0;  
 }  
  
 **long** text;  
 **if** (peek\_text(state, rip, &text) == -1) {  
 **return** -1;  
 }  
  
 **if** (IS\_FRAMESETUP(text)) {  
 /\* Читаем 2 инструкции: push rbp и mov rsp rbp \*/  
 **if** (make\_single\_instruction\_step(state) == -1) {  
 **return** -1;  
 }  
  
 **if** (make\_single\_instruction\_step(state) == -1) {  
 **return** -1;  
 }  
 }  
  
 **return** 0;  
}

Шаги в исходном коде

Теперь, перейдем к возможности выполнения шагов. В начале, реализуем возможность выполнения 1 инструкции. Здесь я просто хочу показать использования ptrace(PTRACE\_SINGLESTEP). Это реализовано в функции make\_single\_instruction\_step (уже видели ранее).

/\* https://github.com/ashenBlade/dumbugger/blob/030714733c9544cb77a13aa5b0f1b53175034361/src/dumbugger.c#L509 \*/  
**static** **int** make\_single\_instruction\_step(DumbuggerState \*state) {  
 **if** (ptrace(PTRACE\_SINGLESTEP, state->pid, NULL, NULL) == -1) {  
 **return** -1;  
 }  
  
 **if** (waitpid(state->pid, &state->wstatus, 0) != state->pid) {  
 **return** -1;  
 }  
  
 **if** (WIFEXITED(state->wstatus) || WIFSIGNALED(state->wstatus)) {  
 state->state = PROCESS\_STATE\_FINISHED;  
 **return** 1;  
 }  
  
 **if** (handle\_child\_stopped(state) == -1) {  
 **return** -1;  
 }  
  
 state->state = PROCESS\_STATE\_STOPPED;  
 **return** 0;  
}

В начале делаю 2 проверки: WIFEXITED - процесс завершился нормально, WIFSIGNALED - процесс завершился из-за сигнала.

Я не стал делать шаг инструкции с пропуском ветвления (step over для инструкций) - не хочу зарываться в декодирование инструкций. Поэтому сразу перейдем к шагам в исходном коде, семейству step XXX команд:

* step out - выйти из текущей выполняемой функции
* step over - выполнить текущую строку без захода во внутрь других функций
* step in - выполнить текущую строку и зайти во внутрь выполняемой

Первой реализуем step in - выполняем 1 строку и если был вызов функции, то спускаемся в нее.

Это самая простая реализация, так как нам просто нужно выполнять инструкции пока не уйдем с текущей строки. Причем не важно куда - на строку в текущей или другой функции. Но также не стоит забывать о том, что при входе в другую функцию мы можем попасть на ее пролог. Если остановимся прямо перед ним, то состояние будет не совсем корректным - надо учесть.

/\* https://github.com/ashenBlade/dumbugger/blob/030714733c9544cb77a13aa5b0f1b53175034361/src/dumbugger.c#L608 \*/  
**int** dmbg\_step\_in(DumbuggerState \*state) {  
 STOPPED\_PROCESS\_GUARD(state);  
  
 **long** rip;  
 **if** (get\_rip(state, &rip) == -1) {  
 **return** -1;  
 }  
 rip -= state->load\_addr;  
 /\* Находим текущую строку кода и ее функцию \*/  
 **long** start;  
 **long** end;  
 **if** (debug\_syms\_get\_line\_bounds(state->debug\_info, rip, &start, &end) == -1) {  
 **return** -1;  
 }  
  
 /\*   
 \* Выполняем по 1 инструкции до тех пор, пока информация о контексте не  
 \* поменяется  
 \*/  
 **do** {  
 **switch** (make\_single\_instruction\_step(state)) {  
 **case** -1:  
 **return** -1;  
 **case** 1:  
 /\* Закончили выполнение \*/  
 **return** 0;  
 }  
  
 **if** (get\_rip(state, &rip) == -1) {  
 **return** -1;  
 }  
 rip -= state->load\_addr;  
  
 } **while** (start <= rip && rip <= end);  
  
 **if** (skip\_frame\_setup(state, rip) == -1) {  
 **return** -1;  
 }  
  
 **return** 0;  
}

И тут есть некоторое замечание - функция debug\_syms\_get\_line\_bounds. Я знаю только текущий RIP, но не номер строки кода. DWARF хранит только номер строки и адрес ее начала. Поэтому нужно найти такую запись, что адрес в этой записи был не меньше, а адрес следующей меньше. Говоря языком интервалов: [target\_line.addr .. RIP .. next\_line.addr).

Но опять загвоздка. Если смотрели вывод DWARF для строк кода, то заметили, что для одной и той же строки есть несколько записей (например, для 9, где выполняется умножение с присваиванием). Поэтому может оказаться, что этот next\_line относится к той же строке (номеру строки). Я не стал такую ситуацию обрабатывать - так проще.

Теперь step over. Здесь уже нельзя просто выполнять инструкции. Банальное "выполнять по 1 инструкции внутри вызванной инструкции (с возможными вызовами других функций) пока не вернемся" звучит интересно, но производительность будет страдать сильно. Поэтому применим другой подход - будем ставить точки останова на строке внутри функции. Наивный подход - ставить точку останова на следующую строку. Иногда он может сработать, но стоит вспомнить об управляющих конструкциях (for, if, goto) так станет понятно, что это нерабочий подход - управление может перейти куда угодно. Чтобы это исправить мы будем ставить точки останова на все строки (кроме текущей конечно), а когда остановимся, то удалим их.

/\* https://github.com/ashenBlade/dumbugger/blob/030714733c9544cb77a13aa5b0f1b53175034361/src/dumbugger.c#L802 \*/  
**int** dmbg\_step\_over(DumbuggerState \*state) {  
 **long** rip;  
 **if** (get\_rip(state, &rip) == -1) {  
 **return** -1;  
 }  
  
 FunctionInfo \*cur\_func;  
 **if** (debug\_syms\_get\_function\_at\_addr(state->debug\_info, rip - state->load\_addr, &cur\_func) == 0) {  
 errno = ENOENT;  
 **return** -1;  
 }  
  
 assert(cur\_func != NULL);  
  
 **long** rbp;  
 **if** (get\_rbp(state, &rbp) == -1) {  
 **return** -1;  
 }  
  
 **long** return\_addr;  
 **if** (peek\_text(state, rbp + 0x8, &return\_addr) == -1) {  
 **return** -1;  
 }  
  
 bp\_list \*set\_breakpoints;  
 **if** (bp\_list\_init(&set\_breakpoints) == -1) {  
 **return** -1;  
 }  
  
 breakpoint\_info return\_addr\_bp = {  
 .address = return\_addr,  
 .saved\_text = 0,  
 };  
 **if** (bp\_list\_add(set\_breakpoints, &return\_addr\_bp) == -1) {  
 bp\_list\_free(set\_breakpoints);  
 **return** -1;  
 }  
  
 rip -= state->load\_addr;  
  
 /\* Создаем массив наших точек останова \*/  
 SourceLineInfo \*line;  
 foreach (line, cur\_func->line\_table) {  
 /\* Пропускаем только текущую строку \*/  
 **if** (line->addr == rip) {  
 **continue**;  
 }  
  
 breakpoint\_info cur\_bp = {  
 .address = line->addr + state->load\_addr,  
 .saved\_text = 0,  
 };  
  
 **if** (bp\_list\_add(set\_breakpoints, &cur\_bp) == -1) {  
 **return** -1;  
 }  
 }  
  
 /\* Выставляем точки останова на все строки \*/  
 breakpoint\_info \*bp;  
 foreach (bp, set\_breakpoints) {  
 **if** (set\_breakpoint\_at\_addr(state, bp->address) == -1) {  
 **return** -1;  
 }  
 }  
  
 /\* Продолжаем выполнение \*/  
 **if** (dmbg\_continue(state) == -1) {  
 **return** -1;  
 }  
  
 **if** (dmbg\_wait(state) == -1) {  
 **return** -1;  
 }  
  
 /\* Удаляем точки останова \*/  
 foreach (bp, set\_breakpoints) {  
 **if** (remove\_breakpoint(state, bp->address) == -1) {  
 **return** -1;  
 }  
 }  
   
 **if** (bp\_list\_free(set\_breakpoints) == -1) {  
 **return** -1;  
 }  
  
 **if** (get\_rip(state, &rip) == -1) {  
 **return** -1;  
 }  
  
 **if** (rip != return\_addr) {  
 **return** 0;  
 }  
  
 **if** (exec\_after\_return(state) == -1) {  
 **return** -1;  
 }  
  
 **return** 0;  
}

Можно прочитать инструкции текущей строки и определить есть ли там инструкции перехода, но это сложно и затратно.

Как можете догадаться и тут не все так просто. Есть 2 момента и все они связаны с возвратом из функции.

Во-первых, инструкция возвращения из функции. Если сейчас мы находимся в на последней строке (возвращении), то наш код не сработает - мы выставили точки останова на строку внутри текущей функции. Но не учли, что после этого выполнение продолжится в следующей функции. Это исправляется тем, что мы выставляем точку останова на адресе возвращения. Код в начале обрабатывает эту ситуацию - читаем адрес возврата и ставим туда точку останова. Откуда мы ее читаем? Из стека - когда мы вызываем функцию (call), то на стеке сохраняется адрес возврата, а затем стандартный пролог дополнительно сохраняет RBP. Поэтому нам необходимо прочитать значение по адресу %RBP + 0x8

|-------------| <----- RBP + 16

| return addr |

|-------------| <----- RBP + 8

| prev rbp |

|-------------| <----- RBP

| ..... |

Но и это не всегда правда. Оптимизации могут создать листовую функцию - она никого не вызывает и поэтому RBP сохранять не нужно. В этом случае, этот код сработает неверно.

Во-вторых, что происходит после возвращения из функции. Когда мы выполняем ret, то переходим к следующей инструкции, после того call. Но мы можем быть еще в невалидном состоянии - после возвращения из функции часто восстанавливается контекст (регистры) или выполняется код сохранения результата (на стек). Поэтому, нам нужно обрабатывать такие ситуации - при возвращении из функции продолжаем выполнение, пока не изменим строку (на которую вернулись). Этой частью занимается функция exec\_after\_return:

/\* https://github.com/ashenBlade/dumbugger/blob/030714733c9544cb77a13aa5b0f1b53175034361/src/dumbugger.c#L648 \*/  
**static** **int** exec\_after\_return(DumbuggerState \*state) {  
 **long** rip;  
 **if** (get\_rip(state, &rip) == -1) {  
 **return** -1;  
 }  
 rip -= state->load\_addr;  
   
 FunctionInfo \*cur\_func;  
 SourceLineInfo \*cur\_line;  
 **if** (debug\_syms\_get\_context(state->debug\_info, rip, &cur\_func, &cur\_line) == -1) {  
 **if** (errno == ENOENT) {  
 **return** 0;  
 }  
 **return** -1;  
 }  
  
 FunctionInfo \*prev\_func;  
 SourceLineInfo \*prev\_line;  
 **if** (debug\_syms\_get\_context(state->debug\_info, rip - 1, &prev\_func, &prev\_line) == -1) {  
 **if** (errno == ENOENT) {  
 **return** 0;  
 }  
 **return** -1;  
 }  
  
 /\* Для обнаружения return проверяю текущую и предыдущую функции \*/  
 **if** (!(SOURCE\_LINE\_INFO\_EQUAL(cur\_line, prev\_line) && FUNCTION\_INFO\_EQUAL(cur\_func, prev\_func))) {  
 **return** 0;  
 }  
  
  
 FunctionInfo \*cur\_function;  
 **if** (debug\_syms\_get\_function\_at\_addr(state->debug\_info, rip, &cur\_function) == 0) {  
 **return** 0;  
 }  
   
 /\* Находим следующую строку - нам нужен адрес ее начала \*/  
 SourceLineInfo \*next\_line = NULL;  
 SourceLineInfo \*sl\_info;  
 foreach (sl\_info, cur\_function->line\_table) {  
 **if** (cur\_line == NULL) {  
 **if** (sl\_info->addr <= rip) {  
 cur\_line = sl\_info;  
 }  
 } **else** {  
 next\_line = sl\_info;  
 **break**;  
 }  
 }  
  
 /\*   
 \* Возможно, следующей строки нет, т.к. текущая была последней.  
 \* В этом случае, используем последний адрес функции (high\_pc) как конец  
 \* последней инструкции  
 \*/  
 **long** line\_end\_addr;  
 **if** (next\_line == NULL) {  
 line\_end\_addr = cur\_function->high\_pc;  
 } **else** {  
 line\_end\_addr = next\_line->addr;  
 }  
  
 /\*   
 \* Выполняем по 1 инструкции, пока не достигнем следующей функции  
 \*/  
 **do** {  
 **switch** (make\_single\_instruction\_step(state)) {  
 **case** -1:  
 **return** -1;  
 **case** 1:  
 **return** 0;  
 **default**:  
 **break**;  
 }  
  
 **if** (get\_rip(state, &rip) == -1) {  
 **return** -1;  
 }  
 rip -= state->load\_addr;  
 } **while** (cur\_line->addr <= rip && rip <= line\_end\_addr);  
 **return** 0;  
}

Последней рассмотрим - step out. Она просто выходит из текущей функции. То есть поставить точку останова на адресе возврата. Но не забываем (как в предыдущем шаге) выполнить инструкции до следующей строки, чтобы восстановить контекст. Все необходимые знания и функции (чтение адреса возврата и выполнение кода после return) для этого уже имеются:

/\* https://github.com/ashenBlade/dumbugger/blob/030714733c9544cb77a13aa5b0f1b53175034361/src/dumbugger.c#L733 \*/  
**int** dmbg\_step\_out(DumbuggerState \*state) {  
 /\* Читаем адрес возврата \*/  
 **long** rbp;  
 **if** (get\_rbp(state, &rbp) == -1) {  
 **return** -1;  
 }  
  
 **long** return\_addr;  
 **if** (peek\_text(state, rbp + 0x8, &return\_addr) == -1) {  
 **return** -1;  
 }  
  
 /\* Выставляем точку останова на адрес возврата \*/  
 **if** (set\_breakpoint\_at\_addr(state, return\_addr) == -1) {  
 **return** -1;  
 }  
  
 /\* И продолжаем исполнение \*/  
 **if** (dmbg\_continue(state) == -1) {  
 **return** -1;  
 }  
  
 **if** (dmbg\_wait(state) == -1) {  
 **return** -1;  
 }  
  
 **if** (state->state == PROCESS\_STATE\_FINISHED) {  
 **return** 0;  
 }  
  
 /\* Удалим точку останова \*/  
 **if** (remove\_breakpoint(state, return\_addr) == -1) {  
 **return** -1;  
 }  
  
 /\* И довыполним оставшиеся инструкции \*/  
 **if** (exec\_after\_return(state) == -1) {  
 **return** -1;  
 }  
  
 **return** 0;  
}

Чтение переменных

Переходим к другой основной функциональности - чтении значений переменных. Просто так выполнять код не имеет смысла, если мы не знаем над какими данными выполняются операции.

Отладчик поддерживает только DWARF 5 версии и только следующие типы:

* примитивы (int, long, char, ...)
* структуры
* указатели

Также для структур буду выводить только ее поля - без рекурсивного спуска в другие структуры. Если у поля структуры есть указатель или вложенная структура, то их выводить не буду.

Первым делом нам необходимо найти, где переменная хранится. Я предполагаю, что все переменные хранятся на стеке непрерывным участком, то есть без регистров и без покусочного хранения. В такой реализации, чтобы найти значение переменной, необходимо прочитать данные на стеке по определенному смещению.

Здесь все просто: Адрес переменной = RBP + смещение. (С 99.99999999% вероятностью это будет отрицательное смещение). Если посмотрим на символы DWARF, то увидим эти (отрицательные) смещения (DW\_AT\_location - DW\_OP\_fbreg -40):

< 1><0x00000113> DW\_TAG\_subprogram

DW\_AT\_name do\_magic

DW\_AT\_type <0x00000058>

DW\_AT\_low\_pc 0x00001139

DW\_AT\_high\_pc <offset-from-lowpc> 96 <highpc: 0x00001199>

DW\_AT\_frame\_base len 0x0001: 0x9c:

DW\_OP\_call\_frame\_cfa

< 2><0x00000135> DW\_TAG\_formal\_parameter

DW\_AT\_name data

DW\_AT\_type <0x00000152>

DW\_AT\_location len 0x0002: 0x9158:

DW\_OP\_fbreg -40

Переменная data хранится на стеке со смещением -40 относительно начала фрейма.

Можете заметить DIE DW\_AT\_frame\_base - это выражение, которое нужно вычислить для получения базы фрейма. Это может быть полезно для листовых функций, которые никого не вызывают и их RBP не меняется. Я предполагаю, что все функции настроили свой RBP.

Нахождение переменных в нетривиальных местах

Но стоит также сделать замечание касательно самого DWARF. Смещение в DW\_OP\_fbreg указывает на *САМОЕ* начало фрейма - с учетом адреса возврата и сохраненного регистра RBP. Поэтому выражение для получения адреса будет выглядеть так: %RBP + offset + 16 (16 = 8 (размер сохраненного RBP) + 8 (размер указателя)). Таким образом, вот так находим адрес значения переменной:

/\* https://github.com/ashenBlade/dumbugger/blob/030714733c9544cb77a13aa5b0f1b53175034361/src/dumbugger.c#L1703 \*/  
**int** dmbg\_get\_variable\_value(DumbuggerState \*state, **const** **char** **\***variable,  
 **char** **\*\*\***out\_values, **int** **\***out\_count) {  
 Variable \*var;  
 **if** (get\_current\_variable(state, variable, &var) == -1) {  
 **return** -1;  
 }  
  
 **long** rbp;  
 **if** (get\_rbp(state, &rbp) == -1) {  
 **return** -1;  
 }  
  
 **long** addr = rbp + var->frame\_offset + 16;  
  
 /\* ... \*/  
}

Теперь будем читать значения. Логика разделяется на 3 случая в зависимости от типа:

Примитив

Если это примитив, то мы читаем слово (PEEK\_TEXT) и отсекаем столько, сколько тип занимает. А когда будем отображать, то дополнительно учтем знак. В DWARF размер и знак указываются в атрибутах DW\_AT\_byte\_size и DW\_AT\_encoding, соответственно. Например, для int запись будет такой:

< 1><0x00000058> DW\_TAG\_base\_type

DW\_AT\_byte\_size 0x00000004

DW\_AT\_encoding DW\_ATE\_signed

DW\_AT\_name int

Для чтения базовых переменных имеется функция read\_simple\_value. Она читает значение и возвращает строку, представляющую значение. Я немного схитрил и не стал приводить значение прочитанного слова к общему - вместо этого я использую разные строки формата для одного и того же переданного значения.

/\* https://github.com/ashenBlade/dumbugger/blob/030714733c9544cb77a13aa5b0f1b53175034361/src/dumbugger.c#L1506 \*/  
**static** **int** read\_simple\_value(DumbuggerState \*state, BaseType \*type,   
 **long** addr, **char** **\*\***value) {  
 **if** (type->kind == TypeKindStruct) {  
 \*value = strdup("");  
 **if** (\*value == NULL) {  
 **return** -1;  
 }  
 **return** 0;  
 }  
  
 /\*   
 \* Предполагаю, что значения примитивных типов не превышает  
 \* размер указателя, что может быть вполне справедливо для 64-битной системы  
 \*/  
 **long** text;  
 **if** (peek\_text(state, addr, &text) == -1) {  
 **return** -1;  
 }  
  
 **char** buf[32];  
 memset(buf, 0, **sizeof**(buf));  
 **const** **char** **\***formatter;  
 **if** (type->kind == TypeKindPrimitive) {  
 PrimitiveType \*prim = (PrimitiveType \*)type;  
 **switch** (prim->byte\_size) {  
 **case** 1:  
 **if** (prim->is\_signed) {  
 formatter = "%d";  
 } **else** {  
   
 formatter = "%u";  
 }  
 **break**;  
 **case** 2:  
 **if** (prim->is\_signed) {  
 formatter = "%hd";  
 } **else** {  
 formatter = "%hu";  
 }  
 **break**;  
 **case** 4:  
 **if** (prim->is\_signed) {  
 formatter = "%d";  
 } **else** {  
 formatter = "%u";  
 }  
 **break**;  
 **case** 8:  
 **default**:  
 **if** (prim->is\_signed) {  
 formatter = "%ld";  
 } **else** {  
 formatter = "%lu";  
 }  
 **break**;  
 }  
 } **else** {  
 assert(type->kind == TypeKindPointer);  
 formatter = "%p";  
 }  
  
 **int** written = snprintf(buf, **sizeof**(buf), formatter, text);  
 **if** (written < 0) {  
 **return** -1;  
 }  
  
 \*value = strndup(buf, written);  
 **if** (\*value == NULL) {  
 **return** -1;  
 }  
  
 **return** 0;  
}

Структура

Значение структуры - это значение всех ее полей. Но как уже сказал, все поля обрабатываются как примитивные, без рекурсивного спуска в значения других структур. Сама структура - это множество полей. У каждого поля есть свое смещение относительно начала структуры.

Например, для структуры Data с 2 полями разного типа в DWARF будет следующая запись:

< 1><0x00000077> DW\_TAG\_structure\_type

DW\_AT\_name Data

DW\_AT\_byte\_size 0x00000010

DW\_AT\_sibling <0x0000009d>

< 2><0x00000084> DW\_TAG\_member

DW\_AT\_name value1

DW\_AT\_type <0x00000058>

DW\_AT\_data\_member\_location 0

< 2><0x00000090> DW\_TAG\_member

DW\_AT\_name value2

DW\_AT\_type <0x0000005f>

DW\_AT\_data\_member\_location 8

Каждое поле структуры указывает свой тип (DW\_AT\_type) и смещение (DW\_AT\_member\_location). Чтение структуры можно свести к циклу: читаем тип (DW\_AT\_type) по адресу base\_addr + DW\_AT\_member\_location, где base\_addr - это адрес начала структуры (то есть простые случаи).

Собственно, таким образом и реализуется логика чтения структуры:

/\* https://github.com/ashenBlade/dumbugger/blob/030714733c9544cb77a13aa5b0f1b53175034361/src/dumbugger.c#L1576 \*/  
**static** **int** get\_struct\_values(DumbuggerState \*state, **long** base\_addr,  
 StructType \*structure, **char** **\*\***values) {  
 StructMember \*member;  
 **bool** error = false;  
 **int** i = 1;  
 foreach (member, structure->members) {  
 **if** (read\_simple\_value(state, member->type, base\_addr + member->byte\_offset, &values[i + 1]) == -1) {  
 error = true;  
 **break**;  
 }  
 values[i] = strdup(member->name);  
 **if** (values[i] == NULL) {  
 error = true;  
 **break**;  
 }  
 i += 2;  
 }  
   
 **if** (error) {  
 **for** (i = 1; i < list\_size(structure->members); ++i) {  
 free(values[i]);  
 values[i] = NULL;  
 }  
 **return** -1;  
 }  
  
 **return** 0;  
}

Указатель

Последний тип - указатель. Когда встречается указатель, то вывожу то, на что он указывает, то есть разыменовываю указатель. А дальнейшая обработка зависит уже от типа на который указывал.

Единственное, что нам остается для него сделать - разыменовать его, то есть получить адрес, который он хранит.

/\* https://github.com/ashenBlade/dumbugger/blob/030714733c9544cb77a13aa5b0f1b53175034361/src/dumbugger.c#L1605 \*/  
**static** **int** read\_pointer\_values(DumbuggerState \*state, **long** addr,   
 PointerType \*ptr, **char** **\*\*\***out\_values, **int** **\***out\_count) {  
 /\*   
 \* int \*value - разыменовываем  
 \* int \*\*value - значение самого указателя  
 \*/  
 **int** count = 2;  
 **char** **\*\***values = calloc(2, **sizeof**(**char** **\***));  
 **if** (values == NULL) {  
 **return** -1;  
 }  
  
 /\* Прочитаем, что находится на месте указателя \*/  
 **long** ptr\_value;  
 **if** (peek\_text(state, state->load\_addr + addr, &ptr\_value) == -1) {  
 free(values);  
 **return** -1;  
 }  
  
 **char** buf[16];  
 memset(buf, 0, **sizeof**(buf));  
 **if** (snprintf(buf, **sizeof**(buf), "0x%p", (**void\***)ptr\_value) < 0) {  
 free(values);  
 **return** -1;  
 }  
 buf[**sizeof**(buf) - 1] = '\0';  
  
 values[0] = strdup(buf);  
 **if** (values[0] == NULL) {  
 free(values);  
 **return** -1;  
 }  
  
 /\*   
 \* После, попытаемся прочитать, что находится уже по указателю  
 \*/  
 **char** **\***ptr\_deref\_value = NULL;  
 **if** (ptr\_value != 0 &&   
 read\_simple\_value(state, ptr->type, ptr\_value, &ptr\_deref\_value) == -1 &&  
 errno != EIO) {  
 /\*   
 \* Если указатель невалидный, то при попытке   
 \* прочтения мы получим errno == EIO  
 \*/  
 free(values[0]);  
 free(values);  
 **return** -1;  
 }  
  
 **if** (ptr\_deref\_value == NULL) {  
 ptr\_deref\_value = strdup("<invalid>");  
 **if** (ptr\_deref\_value == NULL) {  
 free(values[0]);  
 free(values);  
 **return** -1;  
 }  
 }  
  
 values[1] = ptr\_deref\_value;  
  
 \*out\_values = values;  
 \*out\_count = count;  
 **return** 0;  
}  
  
/\* https://github.com/ashenBlade/dumbugger/blob/030714733c9544cb77a13aa5b0f1b53175034361/src/dumbugger.c#L1703 \*/  
**int** dmbg\_get\_variable\_value(DumbuggerState \*state, **const** **char** **\***variable,  
 **char** **\*\*\***out\_values, **int** **\***out\_count) {  
 Variable \*var;  
 /\* ... \*/  
 **switch** (var->type->kind) {  
 **case** TypeKindPointer:  
 ptr = (PointerType \*)var->type;  
 **long** ptr\_value;  
 **switch** (ptr->type->kind) {  
 **case** TypeKindPrimitive:  
 **case** TypeKindPointer:  
 /\*  
 \* int \*value  
 \* int \*\*value  
 \*/  
 **if** (read\_pointer\_values(state, addr, ptr, &values, &count) == -1) {  
 **return** -1;  
 }  
 **break**;  
 **case** TypeKindStruct:  
 /\* Struct \*value \*/  
  
 /\* Читаем, что находится по этому указателю \*/  
 **if** (peek\_text(state, addr, &ptr\_value) == -1) {  
 **return** -1;  
 }  
  
 **char** buf[16];  
 memset(buf, 0, **sizeof**(buf));  
 **if** (snprintf(buf, **sizeof**(buf), "%p", (**void\***)ptr\_value) < 0) {  
 **return** -1;  
 }  
 **char** **\***ptr\_value\_str = strdup(buf);  
 **if** (ptr\_value\_str == NULL) {  
 **return** -1;  
 }  
  
 structure = (StructType \*)ptr->type;  
 /\*   
 \* 0 - значение указателя  
 \* [название поля, значение поля] - для каждого поля  
 \*/  
 count = 1 + 2 \* list\_size(structure->members);  
 values = calloc(count, **sizeof**(**char** **\***));  
 **if** (values == NULL) {  
 free(ptr\_value\_str);  
 **return** -1;  
 }  
   
 **if** (list\_size(structure->members) == 0) {  
 values[0] = ptr\_value\_str;  
 **break**;  
 }  
  
 /\*   
 \* Разыменовываем указатель и читаем поля структуры.  
 \* Если указатель невалидный, то просто отдадим  
 \* единственное значение - значение указателя.  
 \*/  
 **if** (get\_struct\_values(state, ptr\_value, structure, values) == -1) {  
 **if** (errno == EIO) {  
 free(values);  
 values = calloc(1, **sizeof**(**char** **\***));  
 **if** (values == NULL) {  
 **return** -1;  
 }  
  
 values[0] = ptr\_value\_str;  
 count = 1;  
 **break**;  
 }  
  
 free(ptr\_value\_str);  
 free(values);  
 **return** -1;  
 }  
 **break**;  
 **default**:  
 assert(false);  
 **return** -1;  
 }  
 **break**;  
 /\* ... \*/  
 }  
}

Бэктрейс

В процессе работы может быть вызвано множество функций. Было бы полезно просмотреть весь стек вызовов, то есть получить стэктрейс. Сейчас мы эту функциональность реализуем.

В процессе работы вызывается множество функций и каждая имеет свой фрейм. В этом фрейме может храниться что угодно, но нам важно самое начало, 16 байт. В этих 16 байтах хранятся:

* Сохраненный RBP - чтобы найти базу предыдущего фрейма
* Адрес возврата - чтобы найти функцию, которой принадлежит этот фрейм

Как они располагаются во фрейме уже рассматривали. В итоге, можно написать такой псевдокод обработки следующего фрейма:

1. Получаем адрес возврата текущей функции
2. Находим название этой функции
3. Получаем сохраненный RBP
4. Переходим к фрейму из этого RBP

Начинаем работу с текущего фрейма. Единственное отличие от алгоритма в том, что мы знаем адрес текущего фрейма - это RIP, а где RBP уже известно (хранится в нашем фрейме).

Но как понять, что пора остановиться? Когда прочитаем 0 в очередном RBP. В итоге, мы получаем следующую функцию для получения бэктрейса:

/\* https://github.com/ashenBlade/dumbugger/blob/030714733c9544cb77a13aa5b0f1b53175034361/src/dumbugger.c#L1916 \*/  
**int** dmbg\_get\_backtrace(DumbuggerState \*state, **int** max, **char** **\*\*\***out\_bt,  
 **int** **\***out\_count) {  
 /\* Сохраняем текущий контекст (функция + строка файла) \*/  
 **long** rip;  
 **if** (get\_rip(state, &rip) == -1) {  
 **return** -1;  
 }  
  
 **char** **\*\***backtrace;  
 **int** count;  
 backtrace = calloc(max, **sizeof**(**char** **\***));  
 **if** (get\_bt\_function\_at\_addr(state, rip - state->load\_addr, backtrace) == -1) {  
 **return** -1;  
 }  
  
 /\* Начинаем итерироваться по предыдущим фреймам \*/  
 **long** rbp;  
 **if** (get\_rbp(state, &rbp) == -1) {  
 **return** -1;  
 }  
  
 count = 1;  
 **while** (count < max) {  
 /\* Получаем контекст предыдущей функции по адресу возврата \*/  
 **long** ret\_addr;  
 **if** (peek\_text(state, rbp + 0x8, &ret\_addr) == -1) {  
 **return** -1;  
 }  
  
 **if** (get\_bt\_function\_at\_addr(state, ret\_addr - state->load\_addr, &backtrace[count]) == -1) {  
 **return** -1;  
 }  
  
 /\* Переходим к следующей - читаем предыдущий RBP на основании текущего \*/  
 ++count;  
  
 **if** (count == max) {  
 **break**;  
 }  
  
 **if** (peek\_text(state, rbp, &rbp) == -1) {  
 **return** -1;  
 }  
  
 /\*   
 \* Если полученный RBP - 0, то значит мы дошли   
 \* до первородной функции и дальше ничего нет  
 \*/  
 **if** (rbp == 0) {  
 **break**;  
 }  
 }  
  
 \*out\_bt = backtrace;  
 \*out\_count = count;  
 **return** 0;  
}

Полезности

Основная функциональность отладчика реализована, но осталось много вспомогательных фич: отображение исходного кода, чтение регистров и дизассемблирование.

Отображение исходного кода

Чтобы отобразить текущее положение в исходном коде нам необходимо знать только RIP и таблицу строк исходного кода: с их помощью мы получаем файл и номер строки в нем. Но это только путь к файлу, а не его содержимое. Хорошо, что путь к файлу - полный, с корнем. Поэтому алгоритм такой: читаем RIP, находим по нему файл и номер строки, открываем этот файл и читаем строки возле указанного номера.

Эту часть логики я разделил между бэкэндом (логикой отладчика) и фронтэндом (чтение файла).

/\* https://github.com/ashenBlade/dumbugger/blob/030714733c9544cb77a13aa5b0f1b53175034361/src/main.c#L504 \*/  
**static** **int** show\_src\_lines\_cmd(program\_state \*state, **int** argc,  
 **const** **char** **\*\***argv) {  
 **char** **\***line\_buf;  
 **int** target\_line\_no;  
 **if** (dmbg\_get\_src\_position(state->dmbg\_state, &line\_buf, &target\_line\_no) == -1) {  
 **if** (errno == ENOENT) {  
 printf("no source file found for current instruction\n");  
 **return** 0;  
 }  
 **return** -1;  
 }  
  
 FILE \*src\_file = fopen(line\_buf, "r");  
 **if** (src\_file == NULL) {  
 printf("could not open source file: %s\n", line\_buf);  
 free(line\_buf);  
 **return** 0;  
 }  
  
 free(line\_buf);  
 **ssize\_t** cur\_line\_len = 0;  
 **size\_t** buf\_len = 0;  
 **int** cur\_line\_no = 0;  
 **int** prefix\_start\_no = target\_line\_no - 4;  
 **int** suffix\_end\_no = target\_line\_no + 4;  
  
 **while** (0 < (cur\_line\_len = getline(&line\_buf, &buf\_len, src\_file))) {  
 **if** (prefix\_start\_no <= cur\_line\_no && cur\_line\_no <= suffix\_end\_no) {  
 **if** (cur\_line\_no == target\_line\_no - 1) {  
 printf("%d\t---> ", cur\_line\_no + 1);  
 } **else** {  
 printf("%d\t ", cur\_line\_no + 1);  
 }  
  
 printf("%s", line\_buf);  
 } **else** **if** (suffix\_end\_no < cur\_line\_no) {  
 **break**;  
 }  
  
 ++cur\_line\_no;  
 }  
 printf("\n");  
 fflush(stdout);  
 fclose(src\_file);  
 **if** (line\_buf != NULL) {  
 free(line\_buf);  
 }  
  
 **if** (cur\_line\_len < 0 && errno != 0) {  
 **return** -1;  
 }  
  
 **return** 0;  
}  
  
/\* https://github.com/ashenBlade/dumbugger/blob/030714733c9544cb77a13aa5b0f1b53175034361/src/dumbugger.c#L1434 \*/  
**int** dmbg\_get\_src\_position(DumbuggerState \*state, **char** **\*\***filename,  
 **int** **\***line\_no) {  
 **long** rip;  
 **if** (get\_rip(state, &rip) == -1) {  
 **return** -1;  
 }  
  
 FunctionInfo \*fi;  
 SourceLineInfo \*sli;  
 **if** (debug\_syms\_get\_context(state->debug\_info, rip - state->load\_addr, &fi, &sli) == -1) {  
 **return** -1;  
 }  
  
 \*filename = strdup(fi->decl\_filename);  
 \*line\_no = sli->logical\_line\_no;  
 **return** 0;  
}

Вся нужная информация уже хранится в символах отладки, поэтому сложная часть логики больше на стороне фронтэнда.

Дизассемблирование

Дизассемблирование не каждый день необходимо, но это весьма полезная фича. В частности, она необходима если мы хотим поставить точку останова на ассемблерной команде - нужно знать ее адрес.

Самостоятельно это реализовывать долго и сложно, поэтому я воспользовался библиотекой libopcodes. Она позволяет дизассемблировать код различных архитектур в различном формате (AT&T, Intel). Скорее всего, вы уже встречали ее - она используется в objdump.

Для дизассемблирования ей нужно передать сами инструкции для дизассемблирования. Причем, эта часть сделана обобщенной - мы можем передавать как уже полностью прочитанный файл, либо реализовать колбэк и при его вызове читать необходимый диапазон данных. Так как нужно дизассемблировать только текущий участок, то я использовал 2 вариант - реализовал свой колбэк.

На вход колбэку передается буфер (куда записывать прочитанные данные), адрес (по которому читать) и количество байт (сколько читать). Он реализован в функции read\_child\_memory\_opcodes\_callback, а сама логика в dmbg\_disassemble.

/\* https://github.com/ashenBlade/dumbugger/blob/030714733c9544cb77a13aa5b0f1b53175034361/src/dumbugger.c#L1132 \*/  
**static** **int** read\_child\_memory\_opcodes\_callback(bfd\_vma memaddr, bfd\_byte \*myaddr,  
 **unsigned** **int** length,  
 **struct** disassemble\_info \*dinfo) {  
 **if** (length == 0) {  
 **return** 0;  
 }  
  
 **unsigned** **int** left = length;  
  
 /\*   
 \* В каждом цикле читаем по 8 байт (размер слова)  
 \*/  
 **while** (0 < left) {  
 **long** read = ptrace(  
 PTRACE\_PEEKDATA,  
 ((dumbugger\_assembly\_dump\_buffer \*) dinfo->stream)->child\_pid,  
 (**void** **\***) memaddr, NULL);  
 **if** (read == -1 && errno != 0) {  
 **return** errno;  
 }  
  
 **if** (left < **sizeof**(**long**)) {  
 memcpy(myaddr, &read, left);  
 **break**;  
 }  
  
 memcpy(myaddr, &read, **sizeof**(**long**));  
 myaddr += **sizeof**(**long**);  
 left -= **sizeof**(**long**);  
 memaddr += **sizeof**(**long**);  
 }  
  
 **return** 0;  
}  
  
/\* https://github.com/ashenBlade/dumbugger/blob/030714733c9544cb77a13aa5b0f1b53175034361/src/dumbugger.c#L1167 \*/  
**int** dmbg\_disassemble(DumbuggerState \*state, **int** length,  
 DumbuggerAssemblyDump \*result) {  
 assert(state != NULL);  
 STOPPED\_PROCESS\_GUARD(state);  
  
 **if** (length < 0) {  
 errno = EINVAL;  
 **return** -1;  
 }  
  
 **if** (length == 0) {  
 memset(result, 0, **sizeof**(DumbuggerAssemblyDump));  
 **return** 0;  
 }  
  
 **long** rip;  
 **if** (get\_rip(state, &rip) == -1) {  
 **return** -1;  
 }  
  
 dumbugger\_assembly\_dump\_buffer buf = {  
 .insn\_index = 0,  
 .asm\_str\_index = 0,  
 .dump = result,  
 .child\_pid = state->pid,  
 };  
  
 memset(result, 0, **sizeof**(DumbuggerAssemblyDump));  
 result->insns = malloc((**size\_t**) length \* **sizeof**(result->insns[0]));  
 **if** (result->insns == NULL) {  
 **return** -1;  
 }  
 result->length = length;  
  
 **struct** disassemble\_info di;  
 init\_disassemble\_info(  
 &di, &buf, (fprintf\_ftype) fprintf\_dumbugger\_assembly\_dump,  
 (fprintf\_styled\_ftype) fprintf\_styled\_dumbugger\_assembly\_dump);  
 di.arch = bfd\_arch\_i386;  
 di.mach = bfd\_mach\_x86\_64;  
 di.endian = BFD\_ENDIAN\_LITTLE;  
  
 di.disassembler\_options = "att-mnemonic,att";  
 disassemble\_init\_for\_target(&di);  
  
 di.read\_memory\_func = read\_child\_memory\_opcodes\_callback;  
 di.buffer = NULL;  
 di.buffer\_length = 0;  
 di.buffer\_vma = 0;  
  
 disassembler\_ftype disasmler =  
 disassembler(di.arch, di.endian == BFD\_ENDIAN\_BIG, di.mach, NULL);  
  
 **while** (buf.insn\_index < length) {  
 **int** processed = disasmler((bfd\_vma) rip, &di);  
 **if** (processed == -1) {  
 **return** -1;  
 }  
 buf.dump->insns[buf.insn\_index].addr = rip;  
 buf.dump->insns[buf.insn\_index].str[buf.asm\_str\_index] = '\0';  
 rip += processed;  
 ++buf.insn\_index;  
 buf.asm\_str\_index = 0;  
 }  
  
 **return** 0;  
}

Разница между -O0 и -Og

Разница между -g и -ggdb

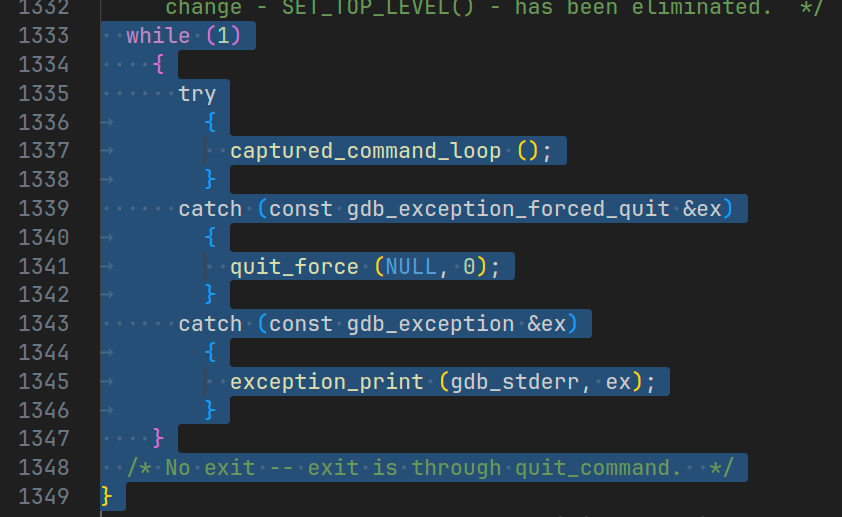
Исследуем gdb

Теперь давайте посмотрим на то, как работают настоящие отладчики. Сделаем это на примере gdb.

Для начала, историческая справка. gdb - это интерактивный (в первую очередь) отладчик, поддерживающий широкий диапазон (компилируемых) ЯП. Он является частью GNU и тесно связан с другими проектами (активно их использует) - о них далее.

На данный момент, последняя мажорная версия 15 - ее я и буду рассматривать, но этот код будет корректен и для других версий, так как сконцентрируемся на базовой логике, которая вряд ли поменяется.

Отступы



Структура проектов

Как уже сказал, gdb использует множество библиотек для своей работы. Все они содержатся внутри репозитория и являются частью GNU. Пройдемся по основным:

* libbfd (Binary File Descriptor) - обработка объектных файлов различного формата (ELF, a.out, IEEE 695, mach-o ...)
* libopcodes - библиотека для дизассемблирования инструкций
* cpu - множество файлов описывающих особенности некоторых архитектур ЦП. Файлы имеют расширения .cpu и .opc, хотя исходный код Scheme и C соответственно (в комментарии на 1 строке указан исходный код).
* gas - ассемблер GNU
* gprof - профилировщик GNU
* libintl - библиотека для интернационализации (основана на gettext)
* ld - линковщик GNU
* libdecnumber - библиотека для работы с десятичными числами (32/64/128 битных)
* libiberty - библиотека, содержащая множество вспомогательных функций, которые используются во многих проектах GNU (например, реализация random или sha)
* libreadline - библиотека, содержащая множество функций для реализации интерактивной командной строки (история, автодополнение, буферизация...)
* sim - симулятор некоторых ЦП (при запуске gdb в режиме симуляции)
* zlib - библиотека для сжатия
* binutils - набор различных утилит для работы с объектными и исполняемыми файлами (ld, gas, grpof, nm, objdump ...)
* gold - линковщик для ELF файлов с упором на скорость
* elfcpp - библиотека для чтения и записи ELF файлов

Архитектура

Архитектура gdb - событийно-ориентированная. Для обработки событий используется цикл событий. Его реализация располагается в файле [event-loop.cс](https://sourceware.org/git/?p=binutils-gdb.git;a=blob;f=gdbsupport/event-loop.cc;h=52e7fd2223fa90a7224e3ddd72466652b6ab3d57;hb=02c10eaecb63e5dbb99cbfdd1c5385e53ed031ff) - poll и неблокирующие дескрипторы. Всего есть 3 типа событий (в скобках указана функция обработчик типа события):

* Таймауты событий (poll\_timers)
* Файловые дескрипторы (gdb\_wait\_for\_event)
* Асинхронные события (check\_async\_event\_handlers)

Командная строка (уведомление о вводе) и события отлаживаемого процесса - это все файловые дескрипторы: stdin и пайп ([self-pipe trick](https://lwn.net/Articles/177897/)).

Регистрация обработчиков событий

Но вот внезапные сигналы файловыми дескрипторами обработать не сможем. Для них используются уже async\_event. Обработкой этих сигналов занимается функция infrun\_async\_inferior\_event\_handler. Но если опять посмотреть вглубь, то ничего сложного эта асинхронность из себя не представляет - мы просто вызываем waitpid для всех inferior'ов, а затем вызываем необходимые обработчики в зависимости от типа события:

Ожидание завершения

Таким образом, обработка событий от процесса поделилась на 2 типа событий: события файлового дескриптора и синхронное ожидание с помощью waitpid.

В проекте существует огромное количество файлов и каждый файл отвечает за свою функциональность. Каждый такой файл можно считать отдельным модулем со своими интерфейсом и состоянием. Их достаточно много и для автоматизации существует инфраструктура - каждый модуль определяет глобальную функцию \_initialize\_XXX (где XXX - название какой-либо функциональности) и в момент сборки (через make) создается функция initialize\_all\_files, которая поочередно вызывает все эти функции. Например, в файле breakpoint.c находится функционал для работы с точками останова и для его инициализации существует функция \_initialize\_breakpoint.

Кодогенерация для инициализации

Для выполнения команд также существует своя инфраструктура. Во-первых, все команды регистрируются в едином реестре (во время вызова \_initialize\_XXX). Во-вторых, не существует команды самой по себе - она принадлежит какому-то классу (даже может быть вложенность). Например, команды точек останова принадлежат классу class\_breakpoins, а далее для нее регистрируются уже сами команды. Регистрация команды выполняется с помощью функции add\_cmd (есть add\_com - обертка над этой), причем у обработчика должна быть следующая сигнатура:

/\*   
 \* args - строка аргументов (без разбиения)  
 \* from\_tty - исполняется в интерактивном режиме  
 \*/  
**typedef** **void** cmd\_simple\_func\_ftype (**const** **char** **\***args, **int** from\_tty);

Можно заметить, что все команды - это функции, не методы класса, и передаются только аргументы команды. Это потому что состояние хранится в глобальных переменных, либо получается через публичный интерфейс (но также хранится в static переменных).

В терминах gdb отлаживаемый процесс называется inferrior. Причина этого в том, что отлаживаться может не только процесс, но и core-dump или эмулируемый процесс (не настоящий).

gdb должен быть максимально кроссплатформенным и поддерживать самые разнообразные форматы отладки и объектных файлов. Это достигается с помощью абстракций.

Так, gdbarch - структура, которая представляет целевую платформу. Она хранит в себе не только состояние (например, размер переменной long double), но и функции взаимодействия (например, для получения PC регистра).

struct gdbarch

Если посмотрели на код, то могли заметить большое количество полей, название типа которых в конце имеет \_ftype. Можно сказать, что gdbarch - это интерфейс, у которого все методы по умолчанию не реализованы (значит не поддерживаются). Каждая поддерживаемая архитектура записывает в эти поля свои обработчики (реализует их). Например, для функции skip\_main\_prologue нашел поддерживаемые архитектуры: x86\_64, x86, FR-V, RS/6000, ARM.

Доступ к этим методам выполняется с первоначальной проверкой, что этот метод реализован (не NULL). Причем выполняется эта проверка не вручную, а с помощью функций gdbarch\_XXX\_p (XXX - это название функций). Для того же skip\_main\_prologue будет следующий интерфейс доступа:

Доступ к skip\_main\_prologue

Но это не единственная точка расширяемости - многая часть функциональности реализована в виде интерфейса (структура с указателями на функции). Многие из таких имеют суффикс ops. Например, есть интерфейсы для:

* Точек останова - breakpoint\_ops (поставить точку останова)
* Отлаживаемого процесса - target\_ops (продолжить выполнение, присоединиться)
* Скриптового языка - extension\_language\_script\_ops (GDB, python)
* Динамических библиотек - solib\_ops (открытие, адрес загрузки)
* Чисел с плавающей точкой - target\_float\_ops (мат. операции, приведение к строке, приведение к другому типу)
* Записи трейса - trace\_file\_write\_ops (открытие/закрытие файла, запись)
* Вычисления адреса символа - symbol\_computed\_ops (описание расположения, получение значения)

Есть много других, но о них не буду говорить, но остановлюсь на моем сетапе - Linux, x86\_64. Для нее будут использованы следующие ops/функции:

* svr4\_so\_ops - взаимодействие с динамическими библиотеками SVR4
* code\_breakpoint\_ops - дефолтный интерфейс создания точек останова
* amd64\_linux\_XXX - семейство функций для работы с платформой - регистры, стек ... (находятся в amd64-linux-tdep.c)
* i386\_gdbarch\_tdep - структура с платформно-зависимыми функциями (включает в себя amd64\_linux\_XXX функции выше)

Теперь, рассмотрим то, как реализуется функциональность отладчика.

Запуск

Для начала, рассмотрим что происходит на старте: запуск процесса и чтение символов.

Первым идет чтение символов запускаемого файла - как основные, так и отладочные. Для обработки ELF файлов используется функция elf\_symfile\_read. Символы отладки использую DWARF - для них используется функция cooked\_index\_worker. Точнее это интерфейс, который реализуют 2 основных класса и каждый ответственен за свою секцию:

* cooked\_index\_debug\_names - .debug\_names
* cooked\_index\_debug\_info - .debug\_info, .debug\_types ...

Первый - читает все строки из .debug\_names секции. Более интересен второй, так как в .debug\_info и .debug\_types содержится основная отладочная информация - DIE.

Логика обработки .debug\_info (со всеми его DIE) содержится в методе index\_dies класса cooked\_indexer. Там находится большой while цикл, проходящий по всем DIE и рекурсивно спускающийся к дочерним. Например, вот кусок обработки DW\_TAG\_subprogram:

Обработка DW\_TAG\_subprogram

Также стоит обратить внимание и на то как читаются отладочные символы. В больших программах отладочной информации много и прочитать их все в 1 поток может занять большое количество времени. Поэтому здесь вступает в работу фоновая многопоточная обработка.

В фоновом потоке запускается основная функция чтения отладочных символов - cooked\_index\_worker::start. Но так как к моменту запуска отлаживаемого процесса символы еще могут быть не считаны, то дополнительно используется лок, который ожидает gdb.

Но это еще не все. Для большей оптимизации эта работа распараллеливается. У нас есть множество CU, которые между собой не связаны. Это и есть точка распараллеливания. Но будет плохо если 1 поток возьмет 2 CU по 100Мб, а другой тоже 2, но по 1Мб. Поэтому для распределения CU используется их размер - каждому выделяется примерно равное количество CU по размеру (сумма размеров CU / количество потоков).

Иллюстрация

Точки останова

Перейдем к точкам останова.

В gdb имеется общая структура breakpoint. Она может представлять не только точки останова, но tracepoint'ы ([tracepoint](https://www.sourceware.org/gdb/current/onlinedocs/gdb.html/Tracepoints.html)), и watchpoint'ы ([watchpoint](https://www.sourceware.org/gdb/current/onlinedocs/gdb.html/Set-Watchpoints.html)). Дополнительно идет разделение на Software (программные) и Hardware (аппаратные) точки останова - последние реализуются с помощью поддержки железа (как можно догадаться). Например, они используются для реализации отслеживания изменения какого-либо участка памяти (data breakpoint).

Перечисление bp\_loc\_type хранит различаемые типы точек останова:

**enum** bp\_loc\_type  
{  
 bp\_loc\_software\_breakpoint,  
 bp\_loc\_hardware\_breakpoint,  
 bp\_loc\_software\_watchpoint,  
 bp\_loc\_hardware\_watchpoint,  
 bp\_loc\_tracepoint,  
 bp\_loc\_other  
};

Чтобы поставить точку останова, необходимо вызвать большое количество функций. По большей части это все абстракции. Но логика выставления точки останова практически такая же как написали мы: находим адрес инструкции, читаем старое значение, заменяем первый байт на 0xCC, записываем и сохраняем старое значение. Эта логика хранится в функции default\_memory\_insert\_breakpoint.

default\_memory\_insert\_breakpoint

Но перед тем как переходить к следующему шагу, надо рассмотреть то, как происходит работа с памятью отлаживаемого процесса. Здесь также используется своя инфраструктура. Функции target\_read\_raw\_memory и target\_write\_raw\_memory можно назвать интерфейсом этой инфраструктуры - чтение и запись, соответственно. Платформно-зависимые функции хранятся в интерфейсе target\_ops (см. выше). Здесь нам интересна функция/поле to\_xfer\_partial. Функции которые занимаются чтением/записью памяти имеют в себе xfer (X transFER) - потому что в ядре своем имеется единственная функция, логика которой зависит от передаваемых аргументов. gdb хранит внутри себя кэш памяти процесса. Когда происходит запрос на чтение или запись, то вначале информация ищется в нем.

Теперь вопрос - как ведется работа с памятью в процессе? В своем отладчике я использовал ptrace для чтения/записи в его адресное пространство. Это рабочий подход, но если потребуется прочитать большой участок памяти, то могут возникнуть проблемы с производительностью. В реальном мире эта задача решается тем, что мы используем procfs - читаем из файла памяти процесса: /proc/<PID>/mem. Для чтения/записи используются системные вызовы pread64 и pwrite64 соответственно (либо обычные read/write если отсутствуют).

Сама функция для чтения/записи памяти из файла:

linux\_proc\_xfer\_memory\_partial\_fd

У этого подхода есть преимущество - мы можем продолжить читать память даже если какой-то поток будет уничтожен. Более того - это файл, значит мы можем читать память даже тогда, когда все потоки запущены (ptrace такое не допускает).

Но это возможно только если /proc/<PID>/mem файл доступен для записи. Для устаревших ядер (примером в документации указывается ядро RHEL6) происходит откат к ptrace реализации. Ее логика работы заключена в функции inf\_ptrace\_peek\_poke. Она довольно проста - последовательно вызываем ptrace пока буфер не окажется нужной длины. Единственное, что хочется отметить - ptrace обернут своей функцией gdb\_ptrace.

inf\_ptrace\_peek\_poke

Кроме этих способов есть еще один - системные вызовы process\_vm\_readv и process\_vm\_writev. Они позволяют читать и писать в адресное пространство другого процесса, причем передать за раз можно несколько буферов (векторное чтение, v в названии). Но его использование отклонили из-за нескольких проблем:

1. process\_vm\_writev не позволяет записывать в RO страницы. Это критический недостаток, так как именно таким способом и ставятся точки останова.
2. process\_vm\_\*v может вызвать гонку в моменты, когда дочерний процесс вызывает какой-либо exec, так как адресное пространство будет изменено.

Различные точки останова

Шаги

Теперь самая интересная часть - шаги в коде. Пойдем также - step in, step over, step out. У gdb другое именование - step, next и fininsh соответственно. Я буду использовать step XXX именование для единообразия. В начале рассмотрим шаг инструкции.

Single instruction step

Семантика step over и step in распространяется и на инструкции. Только здесь эту логику реализовать немного проще - мы точно знаем какие инструкции могут вызвать ветвление. Их надо просто декодировать и определить.

Начнем с простого - step in. Тут все просто - нужно вызвать ptrace(PTRACE\_SINGLESTEP). Эта логика содержится в методе i386\_linux\_nat\_target::low\_resume.

Шаг инструкции step in

Более интересная ситуация со step over. Как уже сказал, нам нужно обойти инструкцию ветвления, но вот никакого декодирования инструкции не происходит. Все довольно просто - мы выполняем 1 шаг (PTRACE\_SINGLESTEP) и сохраняем информацию о том, что начали шаг step over. Когда получаем сигнал остановки, то проверяем не в процессе ли мы шага. Если так, то ставим точку останова на адресе возврата. Описание того, как это происходит в следующей секции.

Step over

Обработка step over разделена на 2 части: запуск и обработка останова. Вспомним, что архитектура событийно-ориентированная, поэтому эти 2 части можно описать так:

* Запуск - настройка окружения и запуск процесса
* Обработка останова - решаем что делать, когда процесс остановился

И вот тут начинается разница с реализацией dumbugger - я ставил точку останова на все строки внутри этой функции, а после удалял. В gdb используется другой подход: выполняем по одной инструкции до тех пор, пока текущая строка не поменяется, а если мы встретим вызов функции, то ставим точку останова на адрес возврата.

Работает это таким образом. На этапе запуска вызывается step\_1. Главное, что она выполняет - инициализирует thread\_control\_state. Это структура, которая хранит состояние для выполнения логики step. Например, диапазон PC текущей строки, фрейм, в котором начали выполнение, и т.д.

После того как подготовили свое состояние, начинаем выполнять по одной инструкции - ptrace(PTRACE\_SINGLESTEP). На этом завершается 1-ая и начинается 2-ая часть. Она заключается в постоянной обработке остановки дочернего процесса - каждый раз после его остановки соответствующий сигнал будет отправляться gdb (событийно-ориентированная архитектура).

Когда мы выполняем инструкции, у нас есть 2 исхода - переходим к следующей инструкции или вызываем функцию (ветвления по типу jmp или loop - это к step in). Каждый раз мы проверяем где остановились:

1. В том же диапазоне инструкций
2. Вышли за диапазон, но заметили, что находимся в начале какой-то функции
3. Вышли за диапазон

3-ий случай - завершение шага. На этом можно вернуть управление пользователю и продолжать выполнять его команды. Но нас интересуют 2 остальных.

Если мы находимся в том же диапазоне, то продолжаем выполнение. Но выполняем по одной инструкции за раз. Благодаря этому нам не надо постоянно перезаписывать память процесса (ставить точки останова). Обнаруживаем это с помощью такой проверки:

/\* https://sourceware.org/git/?p=binutils-gdb.git;a=blob;f=gdb/infrun.c;h=8a10119487c4d58660970f27a08acd69ce7acd03;hb=HEAD#l7679 \*/  
**if** (pc\_in\_thread\_step\_range (ecs->event\_thread->stop\_pc (),  
 ecs->event\_thread)  
 && (execution\_direction != EXEC\_REVERSE  
 || \*curr\_frame\_id == original\_frame\_id))  
 {  
 /\* ... \*/  
 keep\_going (ecs);  
 }

Но что если мы вызывали функцию? gdb в этом случае ставит точку на адресе возврата и продолжает выполнение. По возвращении выполняет ту же самую проверку (3 варианта выше). За эту ситуацию отвечает этот код:

/\* https://sourceware.org/git/?p=binutils-gdb.git;a=blob;f=gdb/infrun.c;h=8a10119487c4d58660970f27a08acd69ce7acd03;hb=HEAD#l7844 \*/  
**if** ((get\_stack\_frame\_id (frame)  
 != ecs->event\_thread->control.step\_stack\_frame\_id)  
 && get\_frame\_type (frame) != SIGTRAMP\_FRAME  
 && ((frame\_unwind\_caller\_id (frame)  
 == ecs->event\_thread->control.step\_stack\_frame\_id)  
 && ((ecs->event\_thread->control.step\_stack\_frame\_id  
 != outer\_frame\_id)  
 || (ecs->event\_thread->control.step\_start\_function  
 != find\_pc\_function (ecs->event\_thread->stop\_pc ())))))  
 {  
 /\* ... \*/  
 **if** (ecs->event\_thread->control.step\_over\_calls == STEP\_OVER\_ALL)  
 {  
 insert\_step\_resume\_breakpoint\_at\_caller (frame);  
 keep\_going (ecs);  
 **return**;  
 }  
 }

Можно заметить, что для проверки функций используется структура frame\_id. gdb для каждой функции создает свой уникальный слепок (просто структура с полями). Он вычисляется в функции compute\_frame\_id. Внутри же вызывается функция из поля this\_id - именно в ней и содержится основная логика. Например, для моей программы вызвалась dwarf2\_frame\_this\_id, так как я использовал символы отладки DWARF.

Вычисление слепка фрейма

Step in

На самом деле, step in и step over - это один и тот же код, просто изначальные условия (thread\_control\_state) немного отличаются: поле step\_over\_calls для step over равно STEP\_OVER\_ALL, а для step in - STEP\_OVER\_NONE. Эту разницу можно увидеть так (см. арг. skip\_subroutines):

/\* https://sourceware.org/git/?p=binutils-gdb.git;a=blob;f=gdb/infcmd.c;h=9fcbadbe0232d8e2a6dc7c545adb1dad39a5d446;hb=HEAD#l758 \*/  
/\* Step until outside of current statement. \*/  
**static** **void**  
step\_command (**const** **char** **\***count\_string, **int** from\_tty)  
{  
 step\_1 (0, 0, count\_string);  
}  
  
/\* https://sourceware.org/git/?p=binutils-gdb.git;a=blob;f=gdb/infcmd.c;h=9fcbadbe0232d8e2a6dc7c545adb1dad39a5d446;hb=HEAD#l766 \*/  
/\* Likewise, but skip over subroutine calls as if single instructions. \*/  
**static** **void**  
next\_command (**const** **char** **\***count\_string, **int** from\_tty)  
{  
 step\_1 (1, 0, count\_string);  
}  
  
/\* https://sourceware.org/git/?p=binutils-gdb.git;a=blob;f=gdb/infcmd.c;h=9fcbadbe0232d8e2a6dc7c545adb1dad39a5d446;hb=HEAD#l831 \*/  
**static** **void**  
step\_1 (**int** skip\_subroutines, **int** single\_inst, **const** **char** **\***count\_string)  
{ /\* ... \*/ }

В таком случае, проверка на STEP\_OVER\_ALL (см. код выше) не сработает и мы пойдем дальше. А дальше нас ждем другое условие, под которое мы попадаем:

/\* https://sourceware.org/git/?p=binutils-gdb.git;a=blob;f=gdb/infrun.c;h=8a10119487c4d58660970f27a08acd69ce7acd03;hb=HEAD#l7397 \*/  
**static** **void**  
process\_event\_stop\_test (**struct** execution\_control\_state \*ecs)  
{  
 /\* ... \*/  
 **struct** symtab\_and\_line tmp\_sal;  
  
 tmp\_sal = find\_pc\_line (ecs->stop\_func\_start, 0);  
 **if** (tmp\_sal.line != 0  
 && !function\_name\_is\_marked\_for\_skip (ecs->stop\_func\_name,  
 tmp\_sal)  
 && !inline\_frame\_is\_marked\_for\_skip (true, ecs->event\_thread))  
 {  
 **if** (execution\_direction == EXEC\_REVERSE)  
 /\* ... \*/  
 **else**  
 handle\_step\_into\_function (gdbarch, ecs);  
 **return**;  
 }  
 /\* ... \*/  
}

handle\_step\_into\_function - функция, которая продолжает выполнение до первой строки исходного кода.

Step out

Step out реализован немного по другому. Вначале мы также выставляем точку останова: ставим ее на адрес возврата и продолжаем выполнение. Состояние программы представляется как цепочка фреймов (вызовов функций). Для получения адреса возврата нам нужно получить предыдущий фрейм в этой цепочке и получить его PC. Предыдущий фрейм находится с помощью функции get\_prev\_frame.

В функции finish\_forward заключаются действия 1-ой части - ставим точку останова и продолжаем работу:

/\* https://sourceware.org/git/?p=binutils-gdb.git;a=blob;f=gdb/infcmd.c;h=9fcbadbe0232d8e2a6dc7c545adb1dad39a5d446;hb=HEAD#l1730 \*/  
**static** **void**  
finish\_forward (**struct** finish\_command\_fsm \*sm, **const** frame\_info\_ptr &frame)  
{  
 **struct** frame\_id frame\_id = get\_frame\_id (frame);  
 **struct** gdbarch \*gdbarch = get\_frame\_arch (frame);  
 **struct** symtab\_and\_line sal;  
 **struct** thread\_info \*tp = inferior\_thread ();  
  
 sal = find\_pc\_line (get\_frame\_pc (frame), 0);  
 sal.pc = get\_frame\_pc (frame);  
  
 sm->breakpoint = set\_momentary\_breakpoint (gdbarch, sal,  
 get\_stack\_frame\_id (frame),  
 bp\_finish);  
  
 /\* We want to print return value, please... \*/  
 tp->control.proceed\_to\_finish = 1;  
  
 proceed ((CORE\_ADDR) -1, GDB\_SIGNAL\_DEFAULT);  
}

Когда точка останова выставлена, то процесс продолжает работу. При остановке точка удаляется.

Если помните, то в dumbugger после возвращения из функции я дополнительно выполнял код до следующей строки. Так вот - gdb этого не делает, поэтому кода не будет.

Из интересного - обнаружение main - входной точки. В случае step out это важно, так как выходить за пределы main нет смысла. За это отвечает функция inside\_main\_func. Ее логика проста: получаем название входной точки (символ) и проверяем, что текущий фрейм не этой функции. 2-ая часть проста - надо проверить адрес. Но 1-ая часть не тривиальна, так как в разных ЯП разное название входной точки. Но это решается просто - функция main\_name возвращает символ входной точки. В начале проверяется, что есть символы специфичные для ЯП: \_\_gnat\_ada\_main\_program\_name (ADA), D main (D), main.main (go), \_p\_\_M0\_main\_program или pascal\_main\_program (Pascal). Если не нашлись, то используется стандартный main.

Обнаружение названия входной точки

Чтение переменных

Так как значения типов зависят от языка, то и за отображение ответственен он (ЯП). Другая важная функциональность - вывод состояния (значения переменных, адресов памяти и т.д.). В gdb есть 2 команды для печати:

* x - скалярное значение по переданному адресу
* print - вычисление выражение и вывод результата в соответствии с его реальным типом

Они разные, но в конечном счете обе используют одни и те же функции, только с разными аргументами. Можно выделить 2 функции:

* value\_print\_scalar\_formatted
* value\_print

Вначале рассмотрим первую функцию. Она выводит в лог скалярное значение. Максимальная типизация - указание размера числа и знаковость. Спускаясь ниже по стеку вызовов, за саму печать разных форматов чисел используются разные функции:

* print\_XXX\_chars - число указанной длинны в указанном XXX основании, например, print\_hex\_chars
* print\_floating - число с плавающей точкой
* print\_address - адрес

Многие из этих функций достойны внимания.

* print\_octal\_chars - выводит число в восьмеричной системе. Загвоздка заключается в том, что каждая цифра представляется 3 битами - не кратно 4. Поэтому было принято решение - разделить логику на циклы по 3 шага. В каждом происходит перенос оставшихся бит на следующий шаг. Таким образом, за 3 шага мы обрабатываем 24 бита = 3 байта. Идея аналогична base64.

print\_octal\_chars

* print\_decimal\_chars - эта функция выводит число в десятичном представлении. В ней используется достаточно интересный алгоритм - он переводит в десятичное представление шестнадцатеричное *любой* длины в байтах (то есть не только кратное 2 - short, int, long).

print\_decimal\_chars

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

* print\_floating - выводит число с плавающей точкой. Здесь интересно то, что для работы с плавающей точкой используется интерфейс - target\_float\_ops. На моей машине используется формат float IEEE 754 одинарной точности, поэтому для работы с ней используется структура floatformat. Для вывода значения используется 3 шага:
  1. создаем строку форматирования
  2. конвертируем в формат float исполняемой машины
  3. выводим с помощью printf
* print\_binary\_chars - выводит двоичное представление числа. Здесь все просто - проходим по всем битам и выводим. Но даже здесь есть интересная деталь - обнаружение ниббла без всех нулей. Если такой встретился, то мы должны вывести его весь. Причина проста - нибблы равные 0000 отображаются просто как 0 (чтобы уменьшить занимаемое место).

print\_binary\_chars

Когда дело доходит до команды print ситуация иная. Ему на вход поступает *выражение* из родного ЯП. Его нужно не только вычислить (для получения результата), но и понять тип для его корректного отображения.

Информация о ЯП представляется базовым классом language\_defn - от него наследуются все конкретные ЯП. Например, для C это будет класс c\_language. Также есть и для ADA, D, rust и др. В этом классе нам интересен метод value\_print - он печатает переданное значение. Опять же для C реализация есть в функции c\_value\_print\_inner.

Если посмотреть в нее, то можно заметить уже знакомый шаблон: проверяем typedef, а затем выводим в соответствии с типом (gdb использует внутреннее представление типов, общие для многих ЯП, как например, массив или целое число).

c\_value\_print\_inner

Как выводятся целые числа (TYPE\_CODE\_INT, TYPE\_CODE\_CHAR) знаем - value\_print\_scalar\_formatted (вызывается внутри). Рассмотрим то, как выводится структура (TYPE\_CODE\_STRUCT).

За ее отображение отвечает функция cp\_print\_value\_fields. Отображение полей структуры можно грубо описать циклом по каждому полю: выводим название поля и его значение. Для представления типов используется структура type. Она хранит в себе всю информацию о типе, который представляет. Поля структуры (у типа) хранятся в виде массива и доступ к ним происходит аналогично - по индексу. Само поле представляется структурой field. У нее есть интересующий нас метод - name, он возвращает название поля.

Но для получения поля структуры используется не type и не field - структура value. Она представляет какое-либо значение, но о своем типе не знает. Там есть метод primitive\_field - он позволяет получить значение (другой value) поля. Но так как value о своем типе не знает, то мы его передаем отдельно (сам тип родительской структуры). Если посмотреть на реализацию primitive\_field, то можно увидеть уже знакомые шаги:

* Получаем тип родителя и поля
* Проверяем их typedef
* Находим смещение поля относительно структуры
* Читаем значение поля

primitive\_field

Бэктрейс

Теперь перейдем к команде backtrace, отображении цепочки вызовов до текущей функции. Вся логика это просто цикл, в котором мы идем по каждому фрейму, начиная с текущего, и отображаем информацию о нем. Это реализуется в цикле:

backtrace\_command\_1

Но эта функция особо ничего не говорит нам, так как в ней используется уже готовый код получения предыдущего фрейма.

На самом деле, реализация не так проста. В dumbugger я просто проходил по фреймам, используя RBP и адрес возврата. Но если пользовались gdb, то знаете, что он дополнительно выводит значения аргументов. Для того, чтобы реализовать подобное, используются символы отладки и интерфейс ЯП совместно: символы отладки говорят как получить стек вызовов, а ЯП выводит значения переданных аргументов. Это очень платформно-зависимая часть (регистры процессора, символы отладки, C), поэтому объясню для моего сетапа.

Представим, что текущий фрейм нам известен и нам нужно найти предыдущий. За развертку стека отвечает интерфейс frame\_unwind и в нем имеется метод prev\_register - он и возвращает значение предыдущего регистра. Вспомним, что наивная (dumbugger) реализация просто читала адрес возврата из фрейма (базу фрейма находили через RBP), но в реальном мире опять не все так просто. Из-за различных оптимизаций некоторая информация могла стереться/видоизмениться и нам придется искать пути, чтобы ее найти. А кто это знает? Правильно - символы отладки. В моем случае, интерфейс frame\_unwind принадлежит DWARF, а сам метод prev\_register реализует dwarf2\_frame\_prev\_register.

dwarf2\_frame\_prev\_register

Можете заметить, что для получения адреса возврата мы просто читаем его со стека (DWARF2\_FRAME\_REG\_SAVED\_OFFSET) по известному смещению - addr = cache->cfa + cache->reg[regnum].loc.offset. Правое слагаемое нам понятно - это само смещение (оно равно -8). Но вот о левом пока не говорили: что такое cfa? CFA - Canonical Frame Address. Можно сказать, что это адрес, по которому начинается фрейм функции. В DWARF ей посвящена отдельная секция с описанием того, как с ней работать. Почему это не так просто - даны пояснения.

Почему вычисление CFA не просто

Так как же мы этот CFA получаем? Вычисляем выражения. Оно хранится в атрибуте DW\_AT\_frame\_base, а инструкции имеют префикс DW\_OP\_. Чаще всего я встречал единственную операцию DW\_OP\_call\_frame\_cfa - она просто читает RBP.

Отладка управляемых языков

До этого момента мы говорили о C (возможно C++), но это низкоуровневый язык (относительно). Значительная часть разработчиков использует более высокоуровневые языки - Python, Java, C#, JavaScript. В них процесс отладки устроен по другому и мы это рассмотрим. Есть и другие ЯП, но рассмотрю только их.

C#

Начнем с C#, а точнее было бы сказать с платформы .NET реализации самих Microsoft. Репозиторий находится [тут](https://github.com/dotnet/runtime). Я буду рассматривать 8 версию (.NET 8).

Для начала справка. .NET - это кроссплатформенный рантайм для IL (intermediate language) - промежуточного языка для виртуальной машины. Весь исполняемый код всегда JIT компилируется, без этапа интерпретации.

Архитектура

В директории src/coreclr/debug содержатся все экспортируемые сущности, используемые для отладки.

У отладчика разделяют 2 части - left-side и right-side:

* Left-side - отладчик на стороне самого процесса (RC, Runtime-Controller)
* Right-side - это сторона запускающего (DI, Debugger Interface)

Таким образом, процесс отладки - это взаимодействие отладчика на стороне отлаживаемого процесса и отладчика с нашей стороны. Рассматривать будем LHS - способ взаимодействия с управляемым кодом. Файлы для LHS/RC описаны в директории ee (execution engine), а для RHS/DI - в di (debug engine).

Для взаимодействия между LHS и RHS используются события. Структура DebuggerIPCEvent представляет такое событие. Можно заметить, что используется IPC. Все IPC события описаны в файле src/coreclr/debug/inc/dgbipceventtypes.h, причем в обе стороны - как RHS -> LHS, так и наоборот.

DebuggerIPCEvent

В поле type хранится сам тип события. Оно определяет содержимое этого перечисления. У каждого значения есть префикс DB\_IPCE\_.

Для отправки событий используются различные стратегии в зависимости от цели. Если отлаживаемый процесс находится локально, то используется механизм общей памяти - событие записывается в память отлаживаемого процесса (DCB, Debugger IPC Control Block), а затем выставляется Condition Variable и LHS узнает о событии.

Отправка события в общую память

Рассмотрим как реализуются основные операции.

Точки останова

У точки останова 2 события:

* DB\_IPCE\_BREAKPOINT\_ADD - выставление
* DB\_IPCE\_BREAKPOINT\_REMOVE - удаление

Сама точка останова представляется классом DebuggerBreakpoint. Так как код управляемый и JIT компилируется, то в процессе выполнения наш код может быть либо уже скомпилированным в нативный, либо все еще в IL. Этот класс отвечает за оба случая.

RAII

Для выставления точки останова в машинном коде используется метод AddBindAndActivateNativeManagedPatch. Выставление точки останова и применение патчей схожи. Это отражается и в коде:

* класс DebuggerControllerPatch представляет изменения в коде
* метод DebuggerController::ApplyPatch применяет переданный патч
* в глобальном переменной-массиве g\_patches хранятся все патчи (точки останова)

Инструкция точки останова платформно-зависимая. За ее выставление (непосредственно по адресу) ответственен метод CORDbgInsertBreakpoint. У нее есть множество реализаций - на каждую поддерживаемую архитектуру:

* x86\_64
* x86
* LoongArch
* RISC-V
* ARM

Если посмотрим на реализацию x86\_64, то увидим знакомые действия.

/\* https://github.com/dotnet/runtime/blob/81cabf2857a01351e5ab578947c7403a5b128ad1/src/coreclr/debug/inc/amd64/primitives.h#L186 \*/  
**inline** **void** CORDbgInsertBreakpoint(UNALIGNED CORDB\_ADDRESS\_TYPE \*address)  
{  
 \*((**unsigned** **char\***)address) = 0xCC; // int 3 (single byte patch)  
 FlushInstructionCache(GetCurrentProcess(), address, 1);  
}

Другие архитектуры

За патч в IL код отвечает функция AddILPatch. Но нельзя просто взять и вписать инструкцию останова, так как один и тот же код может быть за-JIT-ен множество раз - причина generic'и. Поэтому используется такой подход: мы также создаем патч, но не применяем его - он будет применен уже после JIT компиляции. Когда код будет скомпилирован, то для него будет создан новый патч - и вот уже он будет применен. Такой патч (для IL кода еще не скомпилированного) называется Primary Patch и за его выставление ответственна функция AddILPrimaryPatch.

ApplyPatch

За удаление точки останова отвечает метод UnapplyPatch (противоположный ApplyPatch).

UnapplyPatch

Шаги в исходном коде

Шаги также реализованы событиями. Команды для них:

* DB\_IPCE\_STEP - шаг в коде (поведение зависит от аргументов)
* DB\_IPCE\_STEP\_OUT - step out

В результате их выполнения мы получаем такие события:

* DB\_IPCE\_STEP\_RESULT - успешная обработка команды выше
* DB\_IPCE\_STEP\_COMPLETE - команда выше завершена

Разница между ними в том, что 1 отправляется сразу (синхронный ответ), а 2 только когда шаг был завершен (асинхронный ответ). Это необходимо, так как выполнение шага может занять большое количество времени, а за 1 раз можно обрабатывать только 1 событие. Таким образом, работа ведется следующим образом:

1. <- DB\_IPCE\_STEP/DB\_IPCE\_STEP\_OUT
2. -> DB\_IPCE\_STEP\_RESULT
3. *проходит время*
4. -> DB\_IPCE\_STEP\_COMPLETE

Разберем как работает DB\_IPCE\_STEP. Структура для этого события выглядит следующим образом:

Структура события для шага

Можно заметить поле stepIn - это флаг. Если он выставлен, то необходимо выполнить step in, иначе step out. Есть и другие, например, поле rangeIL - это тоже флаг и используется для указания того, что переданные данные для IL кода, а не нативного (прямо как в gdb можно указать какое количество шагов выполнить, только здесь не машинные инструкции). Логика шагания заключена в классе DebuggerStepper и его методе Step.

Также есть класс DebuggerJMCStepper. Он используется для JMC, [Just My Code](https://learn.microsoft.com/en-us/visualstudio/debugger/just-my-code?view=vs-2022). То есть отладчик будет работать только с *нашими* функциями и не будет проваливаться во внутренние функции платформы или чужие модули. Если флаг IsJMCStop в событии выставлен, то будет использоваться эта структура.

Для реализации мы ставим точку останова на следующей выполняемой инструкции. Также мы используем предсказывание переходов.

Чтобы получить следующую инструкцию, используется специальный класс NativeWalker. Его главная задача - дать понять, какого типа инструкция идет дальше. Он различает 8 типов инструкций. Они определены в перечислении WALK\_TYPE.

/\* https://github.com/dotnet/runtime/blob/81cabf2857a01351e5ab578947c7403a5b128ad1/src/coreclr/debug/ee/walker.h#L22 \*/  
**enum** WALK\_TYPE  
{  
 WALK\_NEXT,  
 WALK\_BRANCH,  
 WALK\_COND\_BRANCH,  
 WALK\_CALL,  
 WALK\_RETURN,  
 WALK\_BREAK,  
 WALK\_THROW,  
 WALK\_META,  
 WALK\_UNKNOWN  
};

При создании инстанса этого класса ему передается PC и состояние регистров. После этого, мы можем получить тип инструкции, на который PC указывает. С помощью этого мы и понимаем как ставить точку останова.

TrapStep

Обрабатываются 3 ситуации (в зависимости от текущей инструкции):

* Обычная инструкция - ставим точку останова на следующей инструкции
* RETURN - ставим точку останова по адресу возврата
* CALL - ставим точку останова по указанному в CALL адресу

Но инструкция CALL не такая простая, как я описал. Для него предусмотрено несколько граничных случаев:

* Адрес находится внутри текущей функции
* Используется Tail Call (поддерживается платформой)
* Вызов funclet'ов (мини-функций: внутренних функций, try/catch блоки и т.д. - функция, которая использует стек родителя)

Эта логика выполняется для управляемого кода, то есть скомпилированного из IL рантаймом. Кроме него есть и неуправляемый код. Если мы его встречаем, то ничего сделать не можем. В таких случаях, документация говорит выполнить step out и поставить точку останова уже в управляемом коде.

Теперь, перейдем к step out. Его логика также заключена в классе DebuggerStepper, но метод уже StepOut. Сама логика выставления точки останова заключена в TrapStepOut. Happy path простой - ставим точку останова на адресе возврата.

TrapStepOut - Happy Path

Но интересно не это. Если посмотрели на код, то могли заметить обработку граничных случаев:

* Динамически скомпилированный код - .NET позволяет своими силами в рантайме создавать сборки, компилировать их и запускать. Представьте ситуацию - динамически скомпилированный код вызвал нашу функцию и мы в ней остановились. Что будет, если мы пошлем команду step out? В .NET решили, что если на пути встретится фрейм динамического кода, то мы его **пропустим**.
* Единственный фрейм - такое может случиться, если это Main. Так как это единственный фрейм, то выход из него означает завершение работы программы. То есть, в этом случае мы ничего не будем делать.
* Вычисление динамического выражения - это случаи, когда отладчик отправляет в рантайм выражения, которые необходимо вычислить. Такие выражения ближе к динамически скомпилированному коду, но ведет себя как единственный фрейм, то есть ничего не делаем. В общем, поведение логичное - этот фрейм, грубо говоря, существует в вакууме, так как ни к кому не привязан.
* Политики безопасности - с помощью атрибутов можно задавать различные политики безопасности (декларативная безопасность). Если на методе есть такой атрибут, то перед его вызовом будет вызван перехватчик (interceptor) - код, который проверяет права. Если мы находимся в таком контексте, то необходимо поставить точку останова на функции выше в цепочке вызовов.
* Остальные случаи - этим я обозвал все случаи (в управляемом коде), которые не обработались выше. Если добрались сюда, то ставим точку останова на следующем фрейме, который будем выполнять. Такое поведение в комментарии назвали step next.
* Неуправляемый код - ставим точку останова на адресе возврата.

Для работы с фреймами используется отдельный класс Frame. Он базовый - абстрактный. Всего я нашел 18 подклассов Frame. Например, FuncEvalFrame - это фрейм, который создается отладчиком во время вычисления выражения для вызова функций.

Java

Для начала: Java - это язык, JVM - это виртуальная машина и спецификация. Правильнее, будет говорить, что я рассматриваю не Java, а JVM. В заголовке написал Java, чтобы было понятнее. И из всего множества реализаций JVM, я буду рассматривать HotSpot.

[Java Platform Debugger Architecture (JPDA)](https://docs.oracle.com/en/java/javase/23/docs/specs/jpda/architecture.html) - описание архитектуры JVM для целей отладки. Она состоит из нескольких слоев:

* JVM TI (Java VM Tool Interface) - определяет сервисы, которые JVM (отлаживаемая) предоставляет для отладки
* JDWP (Java Debug Wire Protocol) - коммуникационный протокол между отладчиком и JVM
* JDI (Java Debug Interface) - интерфейс Java для взаимодействия с отладчиком (отладка Java через Java)

На сайте JPDA дана такая визуализация этих компонентов.

Components Debugger Interfaces

/ |--------------|

/ | VM |

debuggee ----( |--------------| <------- JVM TI - Java VM Tool Interface

\ | back-end |

\ |--------------|

/ |

comm channel -( | <--------------- JDWP - Java Debug Wire Protocol

| \ |

| -------------- |

| front-end |

| -------------- | <------- JDI - Java Debug Interface |

| UI |

| -------------- |

Для отладки JVM определен свой протокол - [Java Debug Wire Protocol (JDWP)](https://docs.oracle.com/en/java/javase/21/docs/specs/jdwp/jdwp-spec.html). Он описывает коммуникационный протокол между JVM и отладчиком.

Протокол использует пакеты для общения. Каждый пакет имеет заголовок фиксированного размера. Пакеты могут быть 2 типов: команда (command) и ответ (reply). В заголовке каждого пакета можно выделить 2 главных поля:

* id - идентификатор пакета
* flags - различные флаги

Сейчас во флагах может быть только 1 значение - 0x80. Если он выставлен, то пакет - ответ, иначе - команда. Также есть 2 поля, специфичные для команды/ответа:

* command set (команда) - номер набора команд
* command (команда) - номер команды в указанном наборе команд. Вместе с command set уникально идентифицирует команда
* error code (ответ) - код ошибки

Перед тем как начать взаимодействие по этому протоколу необходимо установить соединение. Делается это с помощью handshake пакета из 14 байт - JDWP-Handshake.

Протокол асинхронный и можно отправить несколько пакетов за раз. Для их разграничения существует поле id. Кроме того, в каждом пакете в конце идет поле переменной длины - Data. В ней хранятся данные, специфичные для конкретной команды. Чтобы определить длину этого поля, в заголовке первым полем идет length - длина всего пакета.

Теперь, перейдем к [JVM Tool Interface](https://docs.oracle.com/en/java/javase/23/docs/specs/jvmti.html). Напомню, что TI - это интерфейс самой JVM. Он используется не только для отладки, но и для мониторинга, профилирования и различного рода анализа.

Также, стоит сказать, что этот интерфейс нативный, то есть реализован в C. JVM имеет концепцию агентов - линкующихся библиотек. С помощью JVM TI (и заголовочного файла jvmti.h) можно получить доступ к нему.

Но это нативный интерфейс - есть еще и JDWP. Он описывает протокол взаимодействия. Согласитесь, странно описывать бинарный протокол для нативных функций. Этот протокол для передачи пакетов использует отдельный транспорт. В [спецификации](https://docs.oracle.com/en/java/javase/17/docs/specs/jpda/conninv.html#transports) описаны TCP сокет и общая память, но есть возможность подключить и сторонний транспорт (Service Provider Interface).

Теперь перейдем к самой функциональности.

Точки останова

За взаимодействие через JVM TI в HotSpot отвечает класс JvmtiEnv. Для выставления точки останова используется метод SetBreakpoint.

Но как водится в Java перед тем как сделать что-то встречаем большое количество абстракций и косвенности (насчитал еще 11 вспомогательных классов). Если спуститься вглубь, то в конечном счете используется метод set класса BreakpointInfo. Он работает так:

1. Получаем текущий поток
2. Находим адрес инструкции
3. Выставляем туда инструкцию точки останова - Bytecodes::\_breakpoint

Можно заметить разницу с предыдущим рантаймом: в .NET точка останова ставится нативная (уже скомпилированный код), а в Java выставляется инструкция в байт-код. Эти решения исходят из дизайна: .NET всегда JIT компилирует, а JVM имеет перед собой выбор - интерпретация или JIT компиляция. Работать с байт-кодом намного удобнее.

BreakpointInfo::set

Чтобы удалить точку останова есть аналогичный метод clear:

BreakpointInfo::clear

Шаги

Чтобы понять, как устроены шаги в исходном коде, стоит вспомнить, что архитектура отладки многоуровневая:

1. JVM
2. Агенты
3. JDWP
4. Клиенты

Из всего этого не рассматривали только 2 - агентов. Агентами называют подключаемые библиотеки, которые работают вместе с JVM. Агенты могут свободно взаимодействовать с JVM. Их можно и написать самим, но нам сейчас важен агент jdk.jdwp.agent - агент, реализующий протокол JDWP.

Но зачем это нужно знать? Дело в том, что в JVM TI предоставляет возможно выполнить только SINGLE\_STEP - 1 инструкцию *байт-кода*. Поэтому реализовывать логику шагов (step XXX) нужно самим. Вот тут и помогает агент jdk.jdwp.agent.

Это поведение можно сравнить и с ptrace: JVM как ОС предоставляет возможность выполнить 1 инструкцию, а мы как пользователи (агент) придаем смысл этим шагам.

В JDWP для шагов также есть команда, но только одна. Ее поведение различается в зависимости от аргументов (каждый аргумент имеет определенные константные значения):

* depth - поведение шага
  + INTO = 0 - step into
  + OVER = 1 - step over
  + OUT = 2 - step out
* size - гранулярность единицы выполнения
  + MIN = 0 - шагнуть как можно меньше (часто это просто 1 инструкция байт-кода)
  + LINE = 1 - шагнуть до следующей строки исходного кода (если этой информации нет, то MIN)

Если size = LINE, то эта команда принимает уже знакомые очертания step IN(TO), step OVER и step OUT.

Сами шаги - это асинхронные события. Поэтому, когда в JVM отправляем SINGLE\_STEP, то ответ приходит в виде события (когда шаг окончен). В общем, step XXX реализованы через *постоянную отправку SINGLE\_STEP* и отслеживания текущего положения. Очень похоже на реализацию gdb, но тут есть большая разница в одном моменте - обработка вызова функции. В gdb, когда мы запросили step over и была вызвана функция - мы ставим точку останова на адресе возврата. В JVM (как минимум в jdk.jdwp.agent) сделано иначе - мы *отслеживаем удаление фрейма*. Для этого используется отдельное событие FramePop

Таким образом, реализации шагов следующие (представим, что знаем где границы текущей строки):

* step in
  + выполняем SINGLE\_STEP пока:
    - не будет вызвана другая функция (отслеживаем по количеству фреймов)
    - либо не изменится текущая строка
* step over
  + выполняем SINGLE\_STEP пока не изменится текущая строка
  + если была вызвана функция, то подписываемся на событие удаления фрейма (NotifyFramePop) - продолжим когда вернемся обратно
* step out
  + подписываемся на событие удаления фрейма (NotifyFramePop)

Это все реализовано не 1 функцией, а несколькими - для нескольких событий. Например, в функции stepControl\_handleStep - логика при получении события окончания SINGLE\_STEP, а в handleFramePopEvent - при получении события удаления фрейма. Также есть обработчики для JVMTI\_EVENT\_EXCEPTION\_CATCH и JVMTI\_EVENT\_METHOD\_ENTRY - бросок исключения и вызов метода.

stepControl\_handleStep

handleFramePopEvent

fullspeed режим

Python

Дальше мы будем препарировать Python, точнее его интерпретатор [CPython](https://github.com/python/cpython/tree/main). Он сильно отличается от предыдущих. И не только тем, что он только интерпретируемый.

Главное отличие - как реализована отладка, точнее отладчик. Отладчик Python реализован в Python: точки останова, шаги, вычисления выражений - это все в самом языке. Встроенный модуль bdb содержит 2 класса:

* Bdb - базовый класс отладчика: шаги, трейсинг, вычисление выражений, запуск/останов
* Breakpoint - класс точки останова: создание, удаление, (де)активация

Но я не говорил, что отладчик *полностью* реализован в Python. На самом деле весь отладчик реализован поверх единственной функции из модуля sys - settrace. Эта функция сохраняет колбек, который будет вызываться при любом интересном событии интерпретатора:

* Вызов/начало/продолжение/возвращение функции
* Исполнение новой строки (исходного) кода
* Исполнение новой инструкции (байт-кода)
* Обработка исключений
* Корутины

Как можно догадаться, работа отладчика заключается в том, чтобы корректно обработать эти события. Этим и занимается метод Bdb.trace\_dispatch (он и регистрируется в sys.settrrace):

Bdb.trace\_dispatch

За trace отвечают события:

* line - выполняется новая строка кода
* call - была вызвана функция
* return - функция возвращается (выход из нее)
* exception - возникло исключение

Если посмотреть в документацию, то можно найти событие opcode - оно вызывается, когда интерпретатор собирается выполнить очередную инструкцию. Но в Python это не особо нужно.

Но можете заметить также события с префиксом c\_. Они относятся не к settrace, а к setprofile и работают с C-функциями. Точно не знаю почему их добавили, но предполагаю что для совместимости и устранения багов, так как для обеих функций используется 1 подсистема. Это подтверждает еще и то, что для этих событий нет обработчика (возвращается та же функция).

Если посмотреть на код, то можно заметить, что некоторые события возвращают функции. Это не спроста: колбек функция должна возвращать функцию, которая будет использоваться для трассировки нового scope'а. То есть, есть глобальная функция трассировщик, которая в зависимости от окружения/аргументов, может переопределить поведение следующего трассировщика.

Последний вопрос - как это все запустить? Когда мы отлаживаем код, то не вставляем явных вызовов sys.set\_trace (это функция, запускающая отладчик). Ответ тоже простой - отдельный (встроенный) модуль pdb.

Главное, что в нем есть - класс Pdb, интерактивный отладчик. Он наследуется от Bdb и использует его функциональность (так обычно и создают отладчики). В принципе нам даже не нужно иметь IDE для отладки - достаточно вызвать отладчик из консоли: python -m pdb script.py, либо вообще из кода - import pdb; pdb.set\_trace() (в новых версиях это можно заменить на единственный вызов breakpoint()). После этого отладчик запустится и нам станет доступна интерактивная консоль. Какие команды дефолтный Pdb понимает есть в [документации](https://docs.python.org/3/library/pdb.html#debugger-commands).

Перейдем к функциональности. Если запускать через модуль, то идя ниже по стеку вызовов находим то, как реализуется отладка (стек вызовов):

* pdb.main - получаем цель для выполнения (модуль/скрипт)
* target.code - получаем код цели:
  + скрипт - читаем файл и компилируем его: exec(compile(fp.read()!r, path, 'exec')) (это простая строка!)
  + модуль - используем вспомогательный модуль runpy для получения кода модуля
* Pdb.\_run - выполняем цель
* Bdb.run - отлаживаем переданное выражение (run - функция для отладки выражений)
  + compile(cmd) - комплируем код, если передали простую строку (как в target.code для скрипта)
  + sys.settrace(self.trace\_dispatch) - регистрируем колбэк для отладки
  + exec(cmd) - выполняем переданный код

Стек вызовов

Вот так и реализуется отладка. На этом этапе уже должно стать понятно, как реализуются всякие step XXX, точки останова и т.д., но для полноты картины рассмотрим.

Точки останова

Точку останова представляет класс Breakpoint. Это простой дата-класс, без логики. Но вот их хранение разделено:

* В Pdb хранится словарь файл -> номер строки где поставлена точка останова
* В Breakpoint хранится статический словарь breaks: (файл + номер строки) -> список объектов Breakpoint.

Точка останова зависит от номера строки, поэтому в trace\_dispatch за проверку точек останова ответственна функция dispatch\_line. Проверка находится в функции break\_here и осуществляется так:

1. Проверяем, что в Pdb эта точка останова зарегистрирована
2. Получаем список всех точек останова находящихся на этой строке
   1. Пропускаем неактивные
   2. Пропускаем те, что не принадлежат функции (например, точка останова на объявлении функции (def))
   3. Пропускаем игнорируемые (счетчик ignore)
   4. С помощью eval проверяем условие
3. Если найденная точка останова временная - удаляем

Шаг 2 реализован в функции effective, что говорит само за себя - находим *действующую* точку останова.

break\_here

А кто все это дело вызывает? Функция трассировщик - на событие выполнения новой строки кода вызывается свое событие.

Вызов трассировщика

Шаги

За шаги отвечают методы того же Bdb:

* set\_next - step over
* set\_step - step in
* set\_return - step out

Но если посмотреть, то можно увидеть, что все они реализуются тем, что просто выставляют определенные значения полей:

* stopframe - фрейм, на котором надо остановиться
* returnframe - фрейм, на котором надо остановиться при *выходе*
* stoplineno - номер строки, на которой надо остановиться (остановиться на строке не меньшей, чем указано)

Реализация set\_XXX

Таким образом, нужное поведение достигается различной комбинацией этих параметров:

|  |  |  |  |
| --- | --- | --- | --- |
|  | **stopframe** | **returnframe** | **stoplineno** |
| step over | текущий фрейм | None | 0 |
| step in | None | None | 0 |
| step out | предыдущий фрейм | текущий фрейм | 0 |
| step out (корутина) | текущий фрейм | None | -1 |

После при вызове колбэка определяем нужно ли останавливаться. В этой таблице более-менее все ясно кроме последней строки. Причина в деталях реализации: итерирование реализовано с помощью while цикла, который бросает исключение StopIteration по окончании. Генераторы - это сопрограммы, также итеративно возвращают значения, но и им новые значения также можно передать (с помощью send). Реализованы аналогично, но бросают GeneratorExit при окончании. Логическое окончание их работы наступает не тогда, когда их фрейм изменился (при возвращении из функции), а тогда, когда было выброшено соответствующее исключение. Поэтому отследить их окончание с помощью return нельзя. Но мы можем отлеживать исключения - для них есть отдельные события и свой обработчик - dispatch\_exception. В нем и проверяется подобная ситуация.

dispatch\_exception

А как же мы проверяем что нужно остановиться? Проверка в методе stop\_here внутри обработчика события line:

dispatch\_line

Вопрос остался не покрытым - как же мы отправляем команды отладчику? В тех обработчиках, что я показывал мы сразу же возвращали обработчик и выполнение продолжалось. Все просто: Bdb - это базовый отладчик, он содержит основную логику, а вот наследники уже знают как эту функциональность использовать. Одного такого наследника мы знаем - Pdb. Как же он получает управление? С помощью хуков - методов, которые наследники должны переопределить своим кодом. Такие обработчики в названии имеют префикс user\_. В той же логике шагов есть вызов обработчика user\_line. В Pdb эта функция возобновляет интерактивную сессию:

Перехват управления

Python это конечно хорошо, но вы видели C? Нельзя не поговорить о питоне, не затронув C, поэтому представляю эту подсистему событий: реализация sys.settrace и генерация события при переходе к новой строке:

Базовая часть

JavaScript

Последним будем рассматривать JavaScript и в частности его рантайм NodeJS. Но NodeJS основан на движке V8, поэтому мы скорее будем рассматривать V8. Поэтому я мои слова будут справедливы для многих рантаймов, основанных на V8. Рассматриваю версию [23.1.0](https://github.com/nodejs/node/tree/v23.1.0), а версия самого [V8](https://github.com/v8/v8/tree/12.9.202.23) - 12.9.

Если движок встраивается, то разработчики для взаимодействия с отладчиком используют Inspector Protocol - протокол взаимодействия с отладчиком. Например, им пользуется Dev Tools в Chrome, но конечный пользователь им (протоколом) не пользуется.

Для начала опишу сам протокол инспектора. Все команды представляются в виде JSON объектов примерно такой структуры:

{  
 // Номер-идентификатор сообщения для отслеживания  
 "seq": "number",  
 // Тип сообщения: запрос, ответ, событие  
 "type": "request | response | event",  
 // Если type === 'request'  
 // Название команды  
 "command": "string",  
 // Объект из именованных пар: название - значение аргумента  
 "arguments": {  
 "argName": "value"  
 },  
 // Если type === 'response'  
 // Ответ на какой запрос  
 "request\_seq": "number",  
 // Выполнявшаяся команда  
 "command": "string",  
 // Успешно или нет выполнилось  
 "success": "boolean",  
 // Возвращаемое значение (в зависимости от команды)  
 "body": "object",  
 // Сообщение об ошибке, если !success  
 "message": "string",  
 // Флаг того, что VM приостановлена командой  
 "running": "boolean",  
 // Идентификаторы связных объектов (такое объекты имеют поле handle, на которое и идет ссылка)  
 "refs": [],  
 // Если type === 'event'  
 // Название события  
 "event": "string",  
 // Данные события  
 "body": "object"  
}

Транспорт не оговаривается, но к инспектору обычно подключаются через HTTP, а в NodeJS через веб-сокеты (если запустить node с флагом --inspect, то будет выведен URL с веб-сокетом).

Но это вопрос представления объекта сообщения. Другой момент - как он кодируется. Если отправляем по HTTP, то кодировка ему на откуп (текстовый/бинарный), но вот внутри все сериализованные сообщения представляются в кодировке [CBOR](https://cbor.io/) - Concise Binary Object Representation (RFC 8949). Если очень коротко - это бинарный JSON: есть аналогичные литералы (true, false, null, undefined) и типы (строки, числа, объекты, массивы), нет строгой схемы. Он даже имеет CWT - аналог JWT, но для CBOR.

Строгой спецификации протокола я не нашел (возможно ее и нет), но все сообщения, используемые протоколом описываются в файле js\_protocol.pdl ([ссылка](https://github.com/v8/v8/blob/12.9.202.23/include/js_protocol.pdl)), специальном формате, используемом для Chrome DevTools.

Также я файл в репозитории bugger-v8-client - [PROTOCOL.md](https://github.com/buggerjs/bugger-v8-client/blob/master/PROTOCOL.md) (последний коммит 9 лет назад). Он в более щадящем человеческом виде описывает протокол.

Теперь, перейдем к архитектуре отладчика.

Архитектура отладки 2-х уровневая - Клиент и Бэкэнд. Если посмотреть поближе на реализацию, то увидим большое количество абстракций (не удивительно, ведь вдохновлялись HotSpot):

* V8Inspector
* V8InspectorSession
* UberDispatcher
* DomainDispatcher
* FronendChannel
* Backend/Frontend

Замечу, что 1) есть еще абстракции и 2) для каждой может быть несколько реализаций. Поэтому, чтобы было проще, опишу в 2 словах кто за что отвечает и как это работает:

V8Inspector - это интерфейс к движку V8. Главное, что нам от него надо - доступ к V8InspectorSession, основному классу взаимодействия. Его можно получить через метод connect. Когда мы его получили, то все сообщения протокола отправляем ему. В самом V8InspectorSession нам нужен только 1 метод - dispatchProtcolMessage.  
Все что делает этот метод - перенаправляет сообщение уже объекту UberDispatcher.

Ненадолго вернемся к протоколу. Все команды разделены по доменам (Domain). Можно сказать, это просто области различной функциональности. Название каждой команды в начале имеет название ее домена. Например, Debugger.stepOver - команда step over для отладчика, а Profiler.startPreciseCoverage - начало сбора информации о покрытии кода.

Так вот, задача UberDispatcher - это найти домен, которому принадлежит команда и отправить ее. Каждый такой домен (его обработчику) - класс DomainDispatcher (его инстанс). Когда команда доходит до него, то она выполняется.

Последний этап - отправка ответа. Здесь стоит заметить, что UberDispatcher отправляет команды в 1 сторону, обратно ничего не возвращает. Это вполне сходится с асинхронной природой команд для отладчика (видели такое и раньше). Но как же отправляются ответы клиенту? Здесь мы приходим к последней абстракции - FrontendChannel. Из названия становится понятно, что это канал взаимодействия с фронтэндом (кто встраивает). Для взаимодействия с фронтэндом со стороны бэкэнда есть 2 основных метода: SendProtocolResponse и sendProtocolNotification (ответ и событие).

И вот мы подошли к финишной черте. Единственная абстракция, про которую мы не говорили - Backend. Это интерфейс для работы с рантаймом. Если посмотреть на его объявление, то можно заметить, что все его методы похожи на сгенерированный код какого-нибудь API: принимаемые аргументы указываются жестко (байтность, опциональность и т.д.), а возвращает объект класса Response.

В реально так и есть - этот класс (как и много других описанных ранее классов) - продукт кодогенерации. Поэтому ссылок на них нет, только здесь могу код показать.

Backend

Этот интерфейс реализует класс V8DebuggerAgentImpl. Если посмотреть на его реализацию, то увидим, что и он на самом деле просто обертка над другим объектом. Но на этот раз последним - V8Debugger. Внутри него и содержится логика отладки. Например, вот реализация метода stepOverStatement:

V8Debugger::stepOverStatement

Собирая все вместе, можно построить такую схему взаимодействия с отладчиком

V8Inspector

|

|

| connect()

|

|

dispatch(msg) V

Frontend ------------------> V8InspectorSession

(клиент) |

^ |

| | dispatch(msg)

| |

| V

| UberDispatcher

| |

| |

| |

| | dispatch(msg)

| |

| SendResponse() V

FrontendChannel <--------------- DomainDispatcher

^ |

| |

| | invoke function

| |

| V

Backend

^ |

| |

| |

| |

| V

V8Debugger

(магия)

Теперь переходим к рассмотрению реализации функциональность отладчика.

Точки останова

В установке точки останова принимает участие еще один важный объект (V8Debugger здесь не участвует, Backend отправляет запрос описанным далее объектам): Script. Это еще одна абстракция (интерфейс). Почему становится ясно, когда понимаешь, что V8 может выполнять не только JS, но и WASM, а способы выставления точек останова в них могут (так и есть) различаться. Поэтому за выставление точек останова ответственен сам скрипт. И эта логика содержится в его методе SetBreakPoint. Также в нем есть объект другого класса - Debug. Этот класс ответственен за работу с точками останова (так как скрипт лучше знает как в нем реализованы точки останова).

Перед выставлением точки останова необходимо подготовить окружение. Речь идет о байт-коде V8. Поведение не сильно отличается от, например, Java: выстраиваем функции (создаем обычные функции из inline), деоптимизация, получение байт-кода. Когда эта процедура закончена, то мы сохраняем точку останова в отдельном массиве, а после выставляем уже в коде. Само выставление точки останова находится в методе ApplyDebugBreak:

ApplyDebugBreak

Можно заметить, что здесь мы себя ведем точно также, как и с обычными машинными инструкциями - перетираем инструкцию точкой останова. Разве что, мы дополнительно проверяем, что там уже стоит точка останова. Когда мы хотим удалить точку останова, то делаем аналогичные вещи:

ClearBreakPoint

В коде используется класс BreakIterator. Это итератор по точкам останова. Выставление точек останова реализовано не кусками, а единым целым - удаляются и выставляются все одновременно.

А что с WASM

Теперь, когда выполнение доходит до точки останова рантайм об этом узнает и вызовет обработчик Runtime\_DebugBreakOnBytecode. Сама логика обработки точки останова находится в том же классе Debug, методе Break. Вот кусок этого метода, ответственный за обработку точек останова:

Debug::Break

Кодогенерация

Шаги

Мы уже могли видеть метод stepOverStatement в классе V8Debugger. И да, остальные методы для шагов находятся в том же классе. Все эти методы-обработчики устроены по одному и тому же принципу: вызываем v8::debug::PrepareStep с нужным аргументом, а затем continueProgram для продолжения работы. Функция v8::debug::PrepareStep принадлежит уже знакомому классу Debug.

Сам алгоритм их обработки схож с тем, что видели в предыдущих рантаймах: сохраняем желаемое состояние (фрейм, строка и т.д.) и продолжаем выполнение, а как только программа остановилась, то сверяем текущее и желаемое состояние. В этом состоянии нам нужно знать только о 2 переменных:

1. last\_step\_action - последний выполнявшийся шаг (ставим равным запрошенному). Нужен просто чтобы обнаружить, что на момент остановки мы запросили шаг.
2. traget\_frame\_count - количество фреймов, которое хотим достичь (глубина стека), или -1, если без разницы. Вспомните, что step into отличается от step over необходимостью остановки при вызове функции.

После, мы вызываем функцию FloodWithOneShot, которая позволит нам выполнить 1 шаг, и продолжаем выполнение.

За обработку шагов отвечает тот же метод, что и за точки останова - Debug::Break. Но уже 2 часть (оставил там в коде троеточие).

Debug::Break

И вот тут нам нужно состояние, которое мы выставляли ранее: В случае step over мы проверяем, что находимся во фрейме не ниже требуемого. А дальше все одинаково - просто останавливаемся (return без переустановки шагов и других функций).

* step out: если находимся во фрейме ниже требуемого, то продолжаем выполнение
* step over: если находимся во фрейме ниже требуемого, то продолжаем выполнение (идем в step into)
* step into: останавливаемся если (либо):
  + строка/кадр изменились
  + возможно мы в цикле из 1 стейтмента
  + на инструкции возврата

Еще можно заметить, что в step into дополнительно обрабатываются генераторы, а точнее yeild стейтмент. Если мы до них доходим, то переходим в режим step out, так как нам нужно вернуть результат, а это возврат из функции.

Конец 1 части

На этом 1 часть заканчивается, так как Хабр ломается от размера статьи. Заключительная часть [здесь](https://habr.com/ru/articles/867482/).

**Теги:**

* [отладка](https://habr.com/ru/search/?target_type=posts&order=relevance&q=%5b%D0%BE%D1%82%D0%BB%D0%B0%D0%B4%D0%BA%D0%B0%5d)
* [отладчики](https://habr.com/ru/search/?target_type=posts&order=relevance&q=%5b%D0%BE%D1%82%D0%BB%D0%B0%D0%B4%D1%87%D0%B8%D0%BA%D0%B8%5d)
* [отладка программ](https://habr.com/ru/search/?target_type=posts&order=relevance&q=%5b%D0%BE%D1%82%D0%BB%D0%B0%D0%B4%D0%BA%D0%B0+%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%5d)
* [debugger](https://habr.com/ru/search/?target_type=posts&order=relevance&q=%5bdebugger%5d)
* [debug](https://habr.com/ru/search/?target_type=posts&order=relevance&q=%5bdebug%5d)
* [ptrace](https://habr.com/ru/search/?target_type=posts&order=relevance&q=%5bptrace%5d)

**Хабы:**

* [Отладка](https://habr.com/ru/hubs/debug/)
* [Системное программирование](https://habr.com/ru/hubs/system_programming/)
* [Linux](https://habr.com/ru/hubs/linux_dev/)
* [Программирование](https://habr.com/ru/hubs/programming/)