Мини-доклад по предмету «Операционные системы»

«Прикладное применение API Linux для отладки в userspace»

~~xxxxxx xxxxxx~~, M33361

Как известно, в разработке ПО существует два популярных подхода для нахождения причины некорректного поведения программы (на сленге, причина таких ошибок и их следствия называются словом «баг»), подразумевающих активные действия разработчика – это отладка с помощью дебаггера, утилиты, способной контролировать исполнение программы и предоставлять множество полезной информации в рантайме, а также логгирование. Тонкая настройка второго метода полезна при дальнейшем мониторинге и поддержании проекта в production-среде, но в процессе разработки, если программист сталкивается с проблемой здесь и сейчас, зачастую метод вырождается в примитивный вывод множества слов-маркеров, обозначающих, что программа действительно дошла до этих строчек кода. Использование же дебаггера требует сосредоточенности, поэтому этот метод не применяется всеми.

Однако, такой инструмент может обладать широкими возможностями: от приостановки работы программы по требованию в заданной строке до общего анализа рантайма: визуализации стека вызовов, значений переменных, полей классов, а также информации о причине аварийного завершения программы в интерактивном режиме.

Для того, чтобы разобраться, какие средства ОС позволяют дебаггерам, наподобие популярной утилиты GDB, находясь в userspace пользоваться привилегиями управления исполнением процесса, чтения и записи в его память, а также перехвата исключений, проведем небольшое исследование на примере ОС Linux. Затем спроектируем приложение, которому эти средства будут полезны, но основная цель которого – не дебаг.

Википедия утверждает [1], что GDB опирается на системный вызов ptrace для реализации главного функционала, а именно:

- attach / detach (начало/прекращение отладки)
- continue / stop (управление исполнением)
- info registers (значения регистров CPU)
- x (чтение памяти процесса)

Точка останова – способ приостановить выполнение программы на определенной строчке кода. Их установка реализуется [2] при помощи либо замены программной инструкции по нужному адресу (software breakpoint), к примеру на int 3 – это инвазивное решение, которое не всегда может сработать, например, если память строго read-only; либо при помощи записи нужного адреса в Debug registers (hardware breakpoint), недостаток которых состоит лишь в том, что таких регистров ограниченное количество, зависимое от архитектуры и вендора. Оба способа реализуемы через ptrace.

Примечательно, что в коде ядра Linux встречаются [3] структуры, по задумке нужные именно для поддержки GDB. Это значит, что он не развивался отдельно от разработчиков ядра.

Когда точка останова срабатывает, GDB перехватывает сигнал SIGTRAP, детектирует остановку целевой программы (потока, далее - жертвы) и возвращает исходную инструкцию на место, если необходимо. Обработка сигналов на этом не заканчивается. Специфика такова, что любая программа, которая начала сессию над жертвой при помощи ptrace, имеет первоочередное право получать сигналы, посланные жертве, независимо от источника (будь то ОС или пользователь), а также фильтровать их, не давая жертве возможности узнать об этом сигнале. Исключение – сигнал SIGKILL, его поймать нельзя.

Перехват сигналов с помощью ptrace вносит вклад в анализ аварийного завершения приложения – позволяет [4] узнать номер сигнала, адрес последней инструкции, адрес в памяти, вызвавший прерывание, и прочее.

Теория с курса операционных систем подсказывает, что procfs также может быть полезной, например, чтобы получить маппинг памяти процесса, производить чтение/запись в память, узнавать значения некоторых регистров (instruction pointer и stack pointer). На этом инструментарий, предоставляемый операционной системой для написания отладчиков, заканчивается.

Реализация прочих возможностей дебаггеров опирается на этот инструментарий совместно с анализом исходного или машинного кода программ. К примеру, для получения стека вызовов или значений переменных, лежащих в кадре стека в GDB предусмотрены [5] два алгоритма:

1. Compiler-assisted – компилятор вкладывает дополнительную информацию в бинарный файл для возможности точного анализа кадров стека.
2. Prologue Analysis – дизассемблирование машинного кода с целью получения знаний о том, как строится кадр стека. Является резервным алгоритмом, так как его хрупкость очевидна.

В итоге, API для отладки приложений в userspace ОС Linux включает в себя два инструмента, достаточных для любых целей:

1. Системный вызов ptrace
2. Файловая система procfs

Для самоотладки и самодиагностики приложение может также использовать обработчики сигналов (signal handlers), чтобы переопределять стандартный ответ приложения на нештатную ситуацию – аварийное завершение. Далее мы рассмотрим в общих чертах, как разработать приложение стресс-тестировщик, полезное для олимпиад по программированию.

Определим желаемые качества стресс-тестировщика (далее - стресс):

1. Производительность (количество тестов за промежуток времени)
2. Интерпретируемость результата теста
3. Возможность накладывать ограничения на исполнение
4. Совместимость с форматом соревнования

Стресс должен уметь запускать решение участника с нужным тестом, валидируя его ответ с эталонным – это действие, которое повторяется в цикле. Улучшить производительность этого цикла можно несколькими способами:

1. Устранение необходимости в перезапуске решения
2. Уменьшение взаимодействия с файловой системой
3. Устранение необходимости в перезапуске валидатора (демонизация)

Первый пункт требует модификации исходного кода участника, а третий – неважная техническая деталь, поэтому мы их не рассмотрим. Второй пункт можно трактовать как предзагрузку тестов в оперативную память, а затем передача в исследуемую программу и получение результата посредством перенаправления IO без активного использования файловой системы.

Результат теста может быть разным – от строки «yes» до падения тестируемой программы. В случае упора на интерпретируемость, стресс должен попытаться объяснить причину падения. Для анализа будем использовать ptrace и procfs.

Пусть решение было запущено и присоединено к отладчику (стрессу). Тогда по описанным ранее возможностям, стресс будет перехватывать сигналы, на основе которых создается подсказка для пользователя, где искать баг.

С помощью маппинга памяти и информации о сигнале имеется возможность различать:

1. Разыменование нулевого указателя
2. Попытку записи в read-only память
3. Попытку исполнения данных (см. DEP, NX bit)
4. Вычисление сигнального NaN или целочисленное переполнение
5. Деление на ноль
6. Переполнение стека
7. Обращение к невыделенной памяти

Подтверждение гипотезы о переполнения стека является самым трудным делом из вышеперечисленных, поскольку требует знания о том, какие сегменты

памяти из листинга /proc/PID/maps являются стеками всех потоков. Поскольку операционная система не предоставляет таких данных, то во время выполнения решения участника необходима трассировка системных вызовов CLONE (при допущении, что участник не создавал потоки иным образом), используя ptrace с флагом остановки PTRACE_O_TRACECLONE, откуда извлекается адрес стека нового потока. Адресы стеков ассоциируются с соответствующими сегментами памяти, и из всех таких сегментов необходимо найти ближайший к адресу, вызвавшему сигнал SIGSEGV. В целом, можно просто запретить многопоточные программы.

Рассмотрим процесс дифференцирования ошибок:

- Характерным признаком разыменования нулевого указателя является сигнал SIGSEGV или SIGBUS с адресом обращения равным нулю.
- Попытку записи в read-only память и попытку исполнения данных объединяет то, что эти ошибки вызваны невозможностью сделать действие с памятью по причине отсутствия соответствующих прав. Для подтверждения этой ошибки достаточно найти сегмент памяти, к которому пыталось обратиться приложение, и просмотреть его права.
- Ошибки, связанные с арифметикой, приводят к возникновению сигнала SIGFPE, вместе с которым уже приходит вся необходимая информация для разделения на случаи (деление на ноль, переполнение, и прочее).
- Переполнение стека – несовпадение stack pointer с каким-либо стековым сегментом памяти и попытка обращения по адресу между ближайшим стековым сегментом и stack pointer при условии получения сигнала SIGSEGV.
- Обращение к невыделенной памяти – альтернатива для гипотезы об ошибке переполнения стека при условии получения сигнала SIGSEGV.

Следующее желаемое качество стресс-тестировщика – возможность накладывать ограничения на исполнение по времени и памяти. Подсчет времени может осуществляться по количеству затраченного процессорного

времени с чтением файла /proc/PID/stat, либо наивно, имея текущую временную метку и метку начала выполнения. Объем потребляемой памяти можно извлекать из файла /proc/PID/status. В случае превышения предела на время исполнения или потребления памяти, следует завершать целевую программу с соответствующим вердиктом, если того требует формат соревнования.

Если стресс-тестировщик получил сигнал от ОС на завершение своей работы (пользователь вызвал Ctrl+C или процесс наткнулся на ошибку), то, используя переопределенные обработчики сигналов, он может корректно остановить запущенные тестирования, записать лог на диск и только после этого завершиться с соответствующим сообщением.

Таким образом, функционал гипотетического стресс-тестировщика реализован с использованием API Linux для отладки приложений в userspace.

Попытка автора воплотить данный проект в реальность привела к созданию репозитория letstatt/stress (GitHub) [6].

Примечания:

1. GNU Debugger - Wikipedia
2. Internals/Breakpoint Handling - GDB Wiki (sourceware.org)
3. user.h source code [include/x86_64-linux-gnu/sys/user.h] - Codebrowser
4. sigaction(2) - Linux manual page (man7.org)
5. Internals/Prologue Analysis - GDB Wiki (sourceware.org)
6. letstatt/stress at dev (github.com)

Источники:

1. Internals - GDB Wiki (sourceware.org)
2. Ubuntu Manpage: ptrace - process trace
3. The /proc Filesystem — The Linux Kernel documentation