Анализ инструментаризации бинарников - Lab5

Matthew Rusakov m.rusakov@innopolis.university SD-03

May 2025

Предисловие

Я собрал 4 бинарника:

```
clang main.c json_fuzz.c -o json_parser -lm
afl-clang-fast main.c json_fuzz.c -o json_parser_afl -lm
afl-clang-fast -fsanitize=address main.c json_fuzz.c -o json_parser_afl_asan -lm
afl-clang-lto -fsanitize=memory main.c json_fuzz.c -o json_parser_msan -lm
```

1 Анализ бинарных файлов

1.1 json_parser (стандартная компиляция)

–

Размер: 25K
Инструментация: Отсутствует
Системные вызовы:

- Стандартная последовательность инициализации
- Минимальное использование `mmap/munmap`
- Нет доступа к `/proc/self/`

Дизассемблирование:

- Прямые вызовы стандартных библиотечных функций
- Отсутствие проверок памяти
- Чистый стековый фрейм без дополнительной instrumentation

Вывод: Базовый бинарник без средств диагностики

1.2 json_parser_afl (только AFL)

–

Размер: 137K (в 5.5x больше стандартного)
Инструментация:

- Добавление coverage-инструментации AFL
- Вставка кода для отслеживания путей выполнения

Системные вызовы:

- Дополнительные `mmap` для shared memory
- Чтение/запись в `.cur_input` для фаззинга
- Нет санитайзер-специфичных вызовов

Дизассемблирование:

- Вставки кода `__afl_` для сбора coverage
- Инструментация ветвлений (`__afl_prev_loc`)
- Сохранение контекста выполнения между итерациями

Вывод: Оптимизирован для сбора coverage-данных

1.3 json_parser_asan (AFL + ASan)

–

Размер: 1.6M (в 64x больше стандартного)
Инструментация:

- Полная memory instrumentation
- Red zones вокруг всех объектов
- Shadow memory mapping

Системные вызовы:

- Частые `mmap/munmap` для shadow memory
- Постоянный мониторинг `/proc/self/maps`
- Пользовательские обработчики сигналов (SIGSEGV)
- `madvise` для оптимизации доступа к памяти

Дизассемблирование:

- Вызовы `__asan_*` перед каждой операцией с памятью
- Замена стандартных функций на `__interceptor_*` версии
- Вставки проверок стека (`__asan_stack_malloc`)

Вывод: Максимальная детекция ошибок памяти ценой производительности

1.4 json_parser_msan (AFL + MSan)

–

Размер: 1.3M (в 52x больше стандартного)
Инструментация:

- Трассировка неинициализированной памяти
- Shadow memory для отслеживания битов инициализации

Системные вызовы:

- Схожи с ASan, но менее интенсивные
- Специфичные `msan_` обработчики
- Меньше обращений к `/proc`

Дизассемблирование:

- Проверки `__msan_` при загрузке значений
- Инструментация перемещения памяти (`memcpy/memset`)
- Отсутствие interceptors для файловых операций

Вывод: Специализирован на обнаружении use-of-uninitialized-memory

2 Выводы

- **Рост размера:** ASan > MSan > AFL > Baseline
- **Накладные расходы:**
  - ASan добавляет наибольшие overhead
  - MSan требует меньше памяти чем ASan
  - AFL минимально влияет на runtime-производительность
- **При каких случаях что лучше использовать:**
  - Для фаззинга: AFL + ASan (максимальный охват ошибок)
  - Для production: Стандартная компиляция или AFL-only
  - Для тестирования: MSan для сложных memory-багов

Список литературы

[1] GitHub Link: https://github.com/MattWay224/reverse-engineering-course В этом репозитории можно найти все лабы и информацию про каждое задание в каждой лабе