ГРУППА КОМПАНИЙ
ПРОСОФТ СИСТЕМЫ

ИНЪЕКЦИЯ ОШИБОК, ИЛИ ЧЕМ ПОЛЕЗНЫ ПРИВИВКИ ДЛЯ КОДА

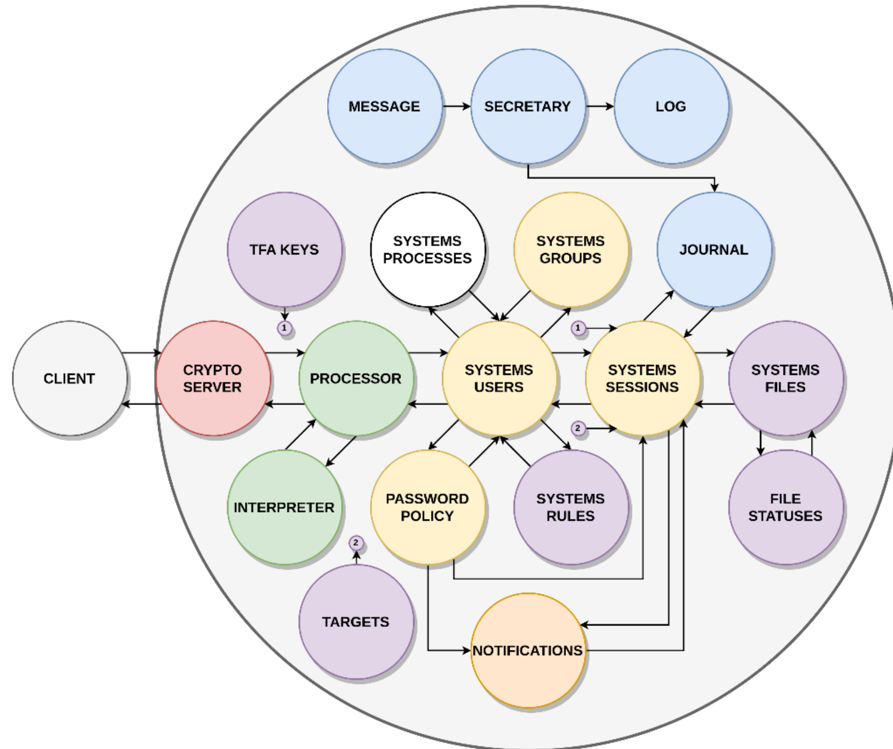
Михаил Беляев

ООО Прософт-Системы

ПРОМЫШЛЕННЫЕ КОНТРОЛЛЕРЫ ARIS

МОДУЛЬ ЗАЩИТЫ ИНФОРМАЦИИ

ГРУППА КОМПАНИЙ ПРОСОФТ СИСТЕМЫ



Функции модуля защиты:

- идентификация и аутентификация пользователей;
- авторизация и контроль доступа;
- проверка целостности информации;
- хранение секретов;
- ведение журнала информационной безопасности.

МОДУЛЬ ЗАЩИТЫ ИНФОРМАЦИИ

Функции модуля защиты:

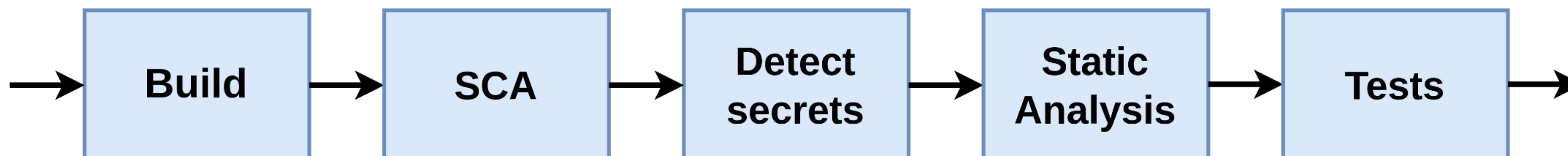
- идентификация и аутентификация пользователей;
- авторизация и контроль доступа;
- проверка целостности информации;
- хранение секретов;
- ведение журнала информационной безопасности.

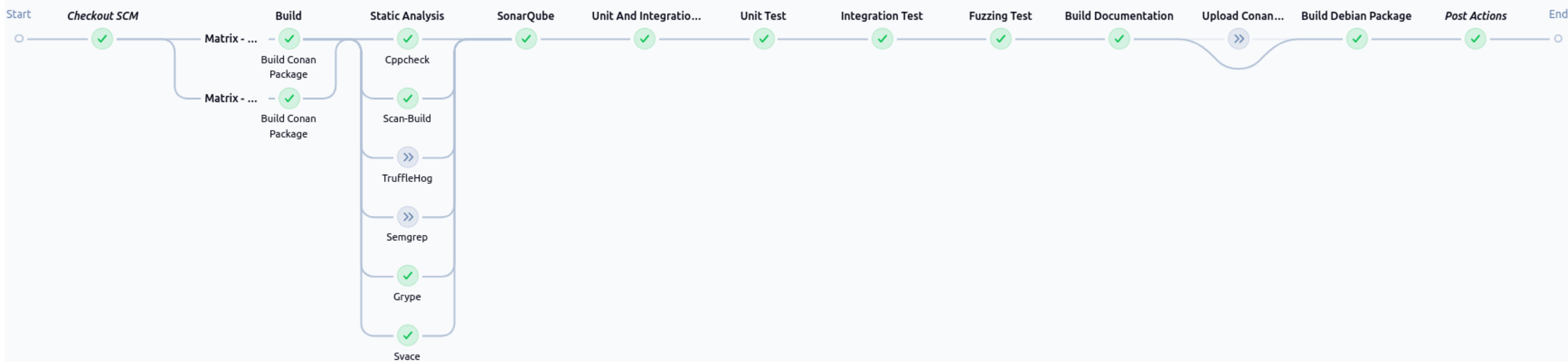
РАЗРАБОТКА БЕЗОПАСНОГО ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

ГРУППА КОМПАНИЙ ПРОСОФТ СИСТЕМЫ

КОНВЕЙЕР СБОРКИ

	Declarative: Checkout SCM	Build	Static analysis	Test	Build Documentation	Upload package	Build Debian package	Declarative: Post Actions
Average stage times: (full run time: ~1h 44min)	41s	2min 10s	15min 2s	1h 22min	2min 9s	0ms	1min 54s	2s
#225 янв. 12 10:40 No Changes	1s	2min 8s	14min 36s (paused for 2s)	1h 24min	2min 12s		2min 10s	3s



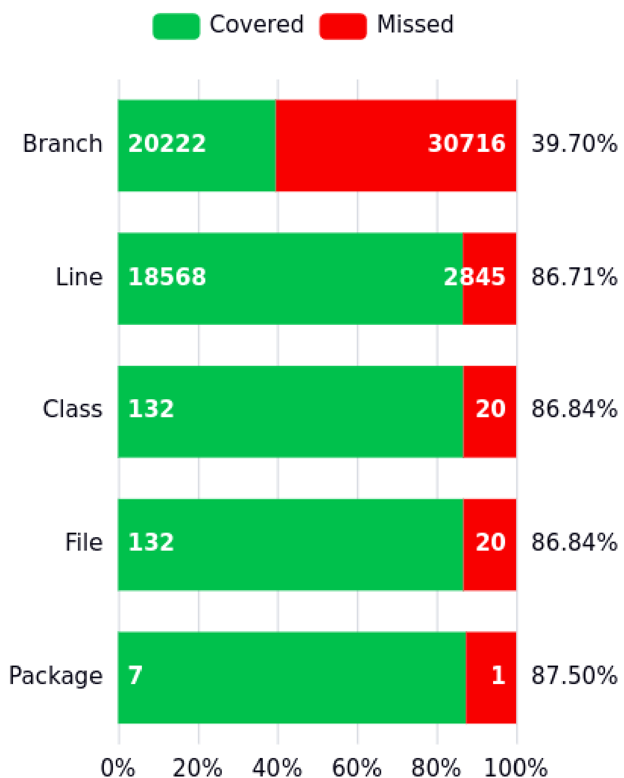


ТЕСТОВОЕ ПОКРЫТИЕ

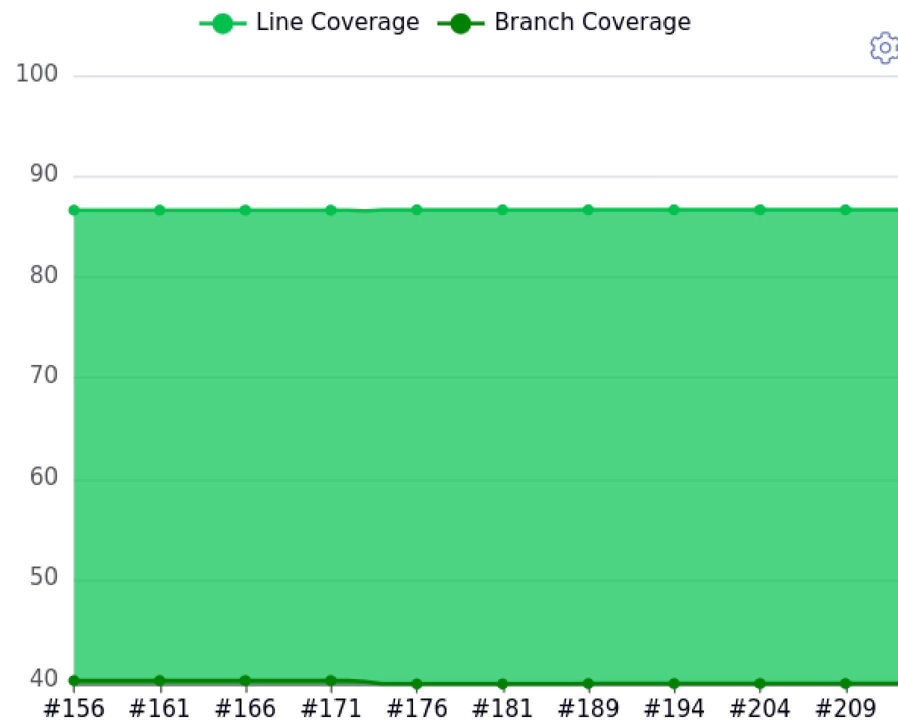
Unit Tests Coverage

Overview | Line Coverage | Branch Coverage | Cyclomatic Complexity | Files

Total coverage overview

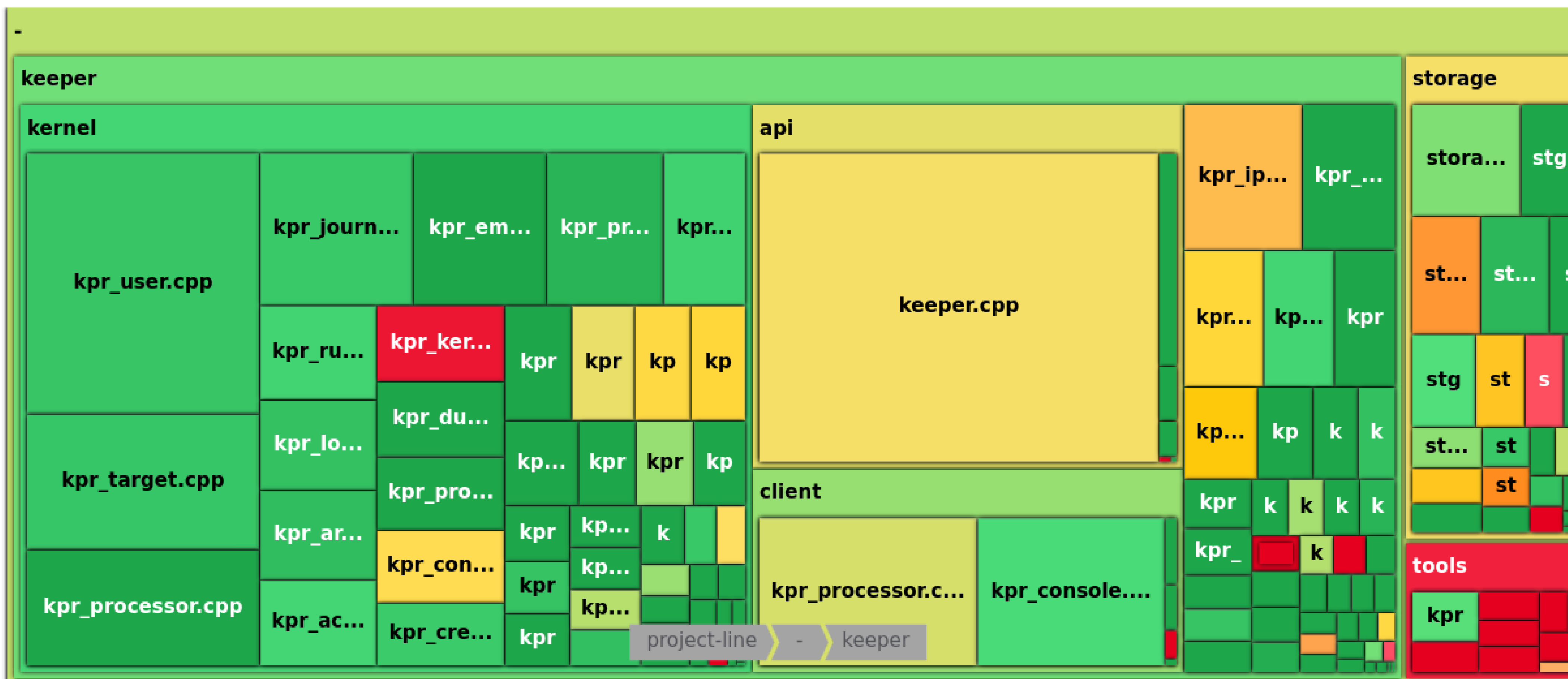


Coverage trend



ТЕСТОВОЕ ПОКРЫТИЕ

Line Coverage



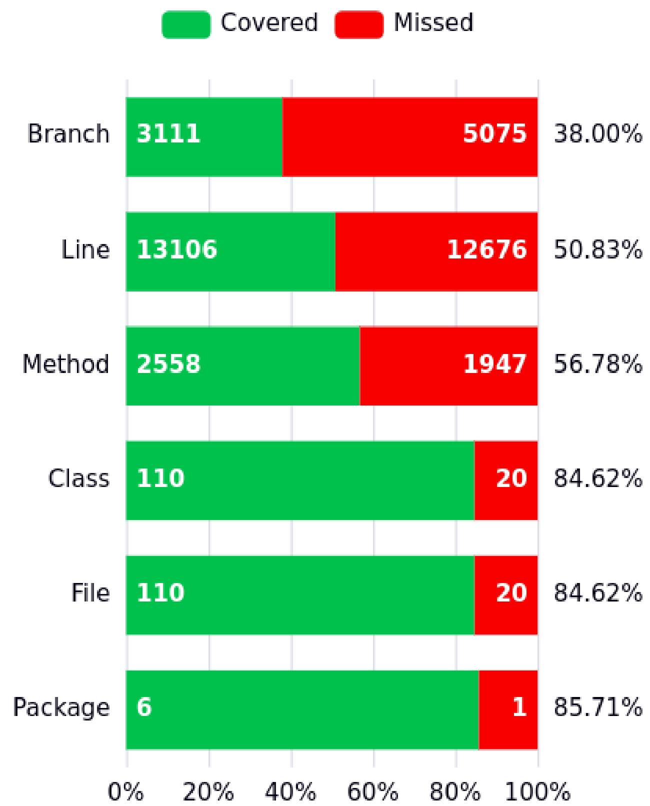
ТЕСТОВОЕ ПОКРЫТИЕ

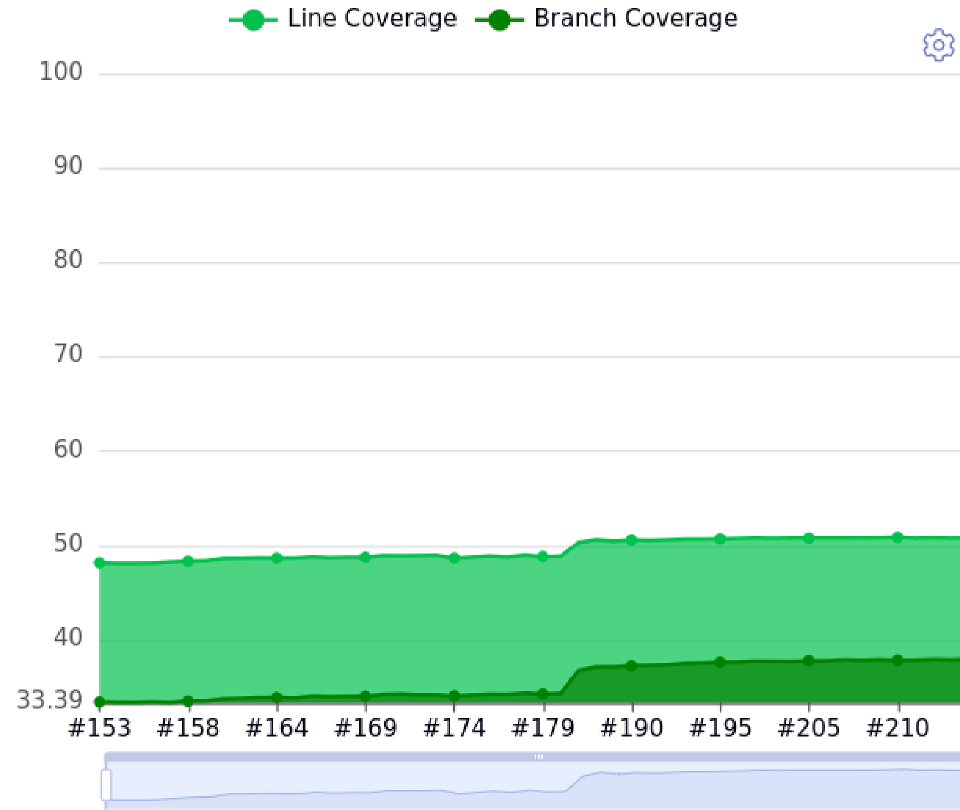

```
408     1 bool KeyReader::Read()
409       {
410   2/4     const std::string errMsg = "ошибка загрузки '" + m_path + "': ";
411   3/4     if (keeper::filesystem::observe::regular_file(m_path))
412           {
413   1/2         ReadOnlyFile file(m_path);
414   1/2         file.Open();
415   2/4         if (file.Size() > 0)
416               {
417   1/2             m_pkey = boost::make_shared<BioBuffer>(file);
418  1/18             KPR_ASSERT(m_pkey);
419     1             return true;
420               }
421  0/28         KPR_ERROR(errMsg << "некорректный размер файла");
422     0         return false;
423     1     }
424 11/28     KPR_ERROR(errMsg << "не удалось найти файл");
425     1     return false;
426     1 }
```

ПОКРЫТИЕ ФАЗЗИНГ-ТЕСТАМИ

Fuzzing Coverage

Overview | Line Coverage | Branch Coverage | Cyclomatic Complexity | Files

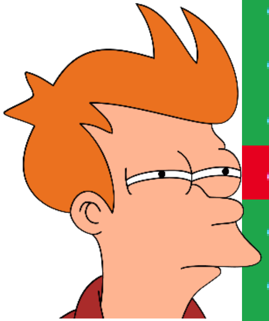




НАПРАВЛЕНИЯ ПО УЛУЧШЕНИЮ КАЧЕСТВА КОДА

1. Анализируй покрытие и пиши дополнительные **юнит-тесты**.
2. Анализируй покрытие и пиши новые **фаззинг-тесты**.
3. ? **?** ?

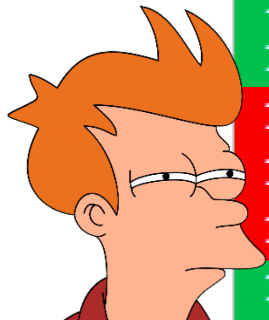
АНАЛИЗ ТЕСТОВОГО ПОКРЫТИЯ



Line	Hits	Source
448		`static int keeper_exec_cmd_result_ok(const keeper::api::string_t& cmd)`
449	1	`{`
450	1	`    try`
451	1	`    {`
452	1	`        const t_request_result result = keeper_client_send(cmd.str());`
453	1/2	`        if (result.code == KEEPER_OK)`
454	1	`        {`
455	2/2	`            return result.value == "ok" ? KEEPER_OK : keeper_bad_result(result.value);`
456	1	`        }`
457	0	`        return result.code;`
458	1	`    }`
459	1	`    catch (...)`
460	1	`    {`
461	0	`    }`
462	0	`    return KEEPER_ERR;`
463	1	`}`

АНАЛИЗ ТЕСТОВОГО ПОКРЫТИЯ

Line	Hits	Source
1224	1	`    int result = keeper_exec_cmd_result_ok_and_value(std::string("getdescription ") +`
1225	1	`        sid->data, description);`
1226	2/2	`    if (result == KEEPER_OK)`
1227	1	`    {`
1228	1	`        try`
1229	1	`        {`
1230	1	`            *desc = to_char(description);`
1231	1	`            KEEPER_UPDATE_CACHED_VALUE(sid, __func__, description);`
1232	1	`            return result;`
1233	1	`        }`
1234	1	`        catch (...)`
1235	1	`        {`
1236	0	`            keeper_free_output(desc);`
1237	0	`            result = KEEPER_ERR;`
1238	0	`        }`
1239	1	`    }`



ЧТО МОГУТ САНИТАЙЗЕРЫ

Санитайзеры могут найти ошибку только в том блоке кода, который был **вызван** в ходе выполнения юнит-теста или фаззинг-теста.

Неиспользуемый в тестах код не будет проанализирован.

СЛОЖНЫЙ КОД ДЛЯ ФАЗЗИНГ-ТЕСТА


```
1 uint8_t* hard_fuzz_me(const uint8_t *data, size_t size) {
2     if (size > 0) {
3         uint8_t *buffer = (uint8_t*)malloc(size);
4         std::string s(reinterpret_cast<const char *>(data), std::min(size, 4ul));
5         if (s == "FUZZ") {
6             memcpy(buffer, data, size);
7             return buffer;
8         }
9         free(buffer);
10     }
11     return nullptr;
12 }
```

СЛОЖНЫЙ КОД ДЛЯ ФАЗЗИНГ-ТЕСТА


```
1  uint8_t* hard_fuzz_me(const uint8_t *data, size_t size) {
2      if (size > 0) {
3          uint8_t *buffer = (uint8_t*)malloc(size);
4          std::string s(reinterpret_cast<const char *>(data), std::min(size, 4ul));
5          if (s == "FUZZ") {
6              memcpy(buffer, data, size);
7              return buffer;
8          }
9          free(buffer);
10     }
11     return nullptr;
12 }
```

1. Функция **malloc** может вернуть **NULL**
2. При конструировании строки можем получить исключение **bad_alloc**

ИНЪЕКЦИЯ ОШИБОК

ГРУППА КОМПАНИЙ ПРОСОФТ СИСТЕМЫ

Инъекция ошибок – это метод искусственного внесения разного рода неисправностей в программный код для тестирования отказоустойчивости.

БИБЛИОТЕКА ДЛЯ ИНЪЕКЦИИ ОШИБОК


```
 1 void enable(); ///< включить инъекцию ошибок, при выделении памяти
 2 void disable(); ///< выключить инъекцию ошибок, при выделении памяти
 3 bool is_enabled(); ///< проверить, что инъекция ошибок, при выделении памяти, включена
 4 void set_fault_mask(uint64_t value); ///< задать маску ошибок
 5 bool register_thread(); ///< зарегистрировать свой поток в списке потоков
 6 void unregister_thread(); ///< исключить свой поток из списка потоков
 7 void unregister_all_threads(); ///< исключить все потоки из списка потоков
 8
 9 /**
10  * Класс RAII обертка над включением инъекции ошибок
11  */
12 class FaultBreaker {
13 public:
14     FaultBreaker() {
15         enable();
16     }
17     ~FaultBreaker() {
18         disable();
19     }
20 };
```

ПРИМЕР ФАЗЗИНГ-ТЕСТА С ИНЪЕКЦИЕЙ ОШИБОК


```
1 extern "C" int LLVMFuzzerInitialize() {
2     fault_injection::allocation::register_thread();
3     return 0;
4 }
5
6 extern "C" int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size) {
7     FuzzedDataProvider provider(data, size);
8     fault_injection::allocation::set_fault_mask(provider.ConsumeIntegral<uint64_t>());
9     auto v = provider.ConsumeRemainingBytes<uint8_t>();
10     try {
11         fault_injection::allocation::FaultBreaker guard;
12         hard_fuzz_me(v.data(), v.size());
13     } catch (const std::bad_alloc& ) {
14         /* при инъекции ошибок нормально получать данный тип исключений  */
15     }
16     return 0;
17 }
```

ПРИМЕР ФАЗЗИНГ-ТЕСТА С ИНЪЕКЦИЕЙ ОШИБОК


```
1 extern "C" int LLVMFuzzerInitialize() {
2     fault_injection::allocation::register_thread();
3     return 0;
4 }
5
6 extern "C" int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size) {
7     FuzzedDataProvider provider(data, size);
8     fault_injection::allocation::set_fault_mask(provider.ConsumeIntegral<uint64_t>());
9     auto v = provider.ConsumeRemainingBytes<uint8_t>();
10    try {
11        fault_injection::allocation::FaultBreaker guard;
12        hard_fuzz_me(v.data(), v.size());
13    } catch (const std::bad_alloc& ) {
14        /* при инъекции ошибок нормально получать данный тип исключений  */
15    }
16    return 0;
17 }
```

ПРИМЕР ФАЗЗИНГ-ТЕСТА С ИНЪЕКЦИЕЙ ОШИБОК


```
1 extern "C" int LLVMFuzzerInitialize() {
2     fault_injection::allocation::register_thread();
3     return 0;
4 }
5
6 extern "C" int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size) {
7     FuzzedDataProvider provider(data, size);
8 →   fault_injection::allocation::set_fault_mask(provider.ConsumeIntegral<uint64_t>());
9     auto v = provider.ConsumeRemainingBytes<uint8_t>();
10    try {
11        fault_injection::allocation::FaultBreaker guard;
12        hard_fuzz_me(v.data(), v.size());
13    } catch (const std::bad_alloc& ) {
14        /* при инъекции ошибок нормально получать данный тип исключений  */
15    }
16    return 0;
17 }
```

ПРИМЕР ФАЗЗИНГ-ТЕСТА С ИНЪЕКЦИЕЙ ОШИБОК


```
1 extern "C" int LLVMFuzzerInitialize() {
2     fault_injection::allocation::register_thread();
3     return 0;
4 }
5
6 extern "C" int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size) {
7     FuzzedDataProvider provider(data, size);
8     fault_injection::allocation::set_fault_mask(provider.ConsumeIntegral<uint64_t>());
9     auto v = provider.ConsumeRemainingBytes<uint8_t>();
10    try {
11        fault_injection::allocation::FaultBreaker guard;
12        hard_fuzz_me(v.data(), v.size());
13    } catch (const std::bad_alloc& ) {
14        /* при инъекции ошибок нормально получать данный тип исключений  */
15    }
16    return 0;
17 }
```

КАК РАБОТАЕТ ИНЪЕКЦИЯ ОШИБОК


```
1 bool AllocationManager::IsFault() noexcept {
2     std::lock_guard<std::mutex> guard{m_mutex};
3     if (m_enabled) {
4         const pthread_t id = pthread_self();
5         if (m_threads.count(id)) {
6             size_t index = m_faultIndex++ % (sizeof(uint64_t) * 8);
7             uint64_t mask = 1;
8             mask = mask << index;
9             if (mask & m_faultMask)
10                return true;
11        }
12    }
13    return false;
14 }
15
16 void* AllocationManager::Malloc(size_t size) noexcept {
17    if (IsFault())
18        return nullptr;
19    return real_malloc(size);
20 }
```

КАК РАБОТАЕТ ИНЪЕКЦИЯ ОШИБОК


```cpp
1  bool AllocationManager::IsFault() noexcept {
2      std::lock_guard<std::mutex> guard{m_mutex};
3      if (m_enabled) {
4          const pthread_t id = pthread_self();
5          if (m_threads.count(id)) {
6              size_t index = m_faultIndex++ % (sizeof(uint64_t) * 8);
7              uint64_t mask = 1;
8              mask = mask << index;
9              if (mask & m_faultMask)
10                 return true;
11         }
12     }
13     return false;
14 }
15
16 void* AllocationManager::Malloc(size_t size) noexcept {
17     if (IsFault())
18         return nullptr;
19     return real_malloc(size);
20 }
```

КАК РАБОТАЕТ ИНЪЕКЦИЯ ОШИБОК

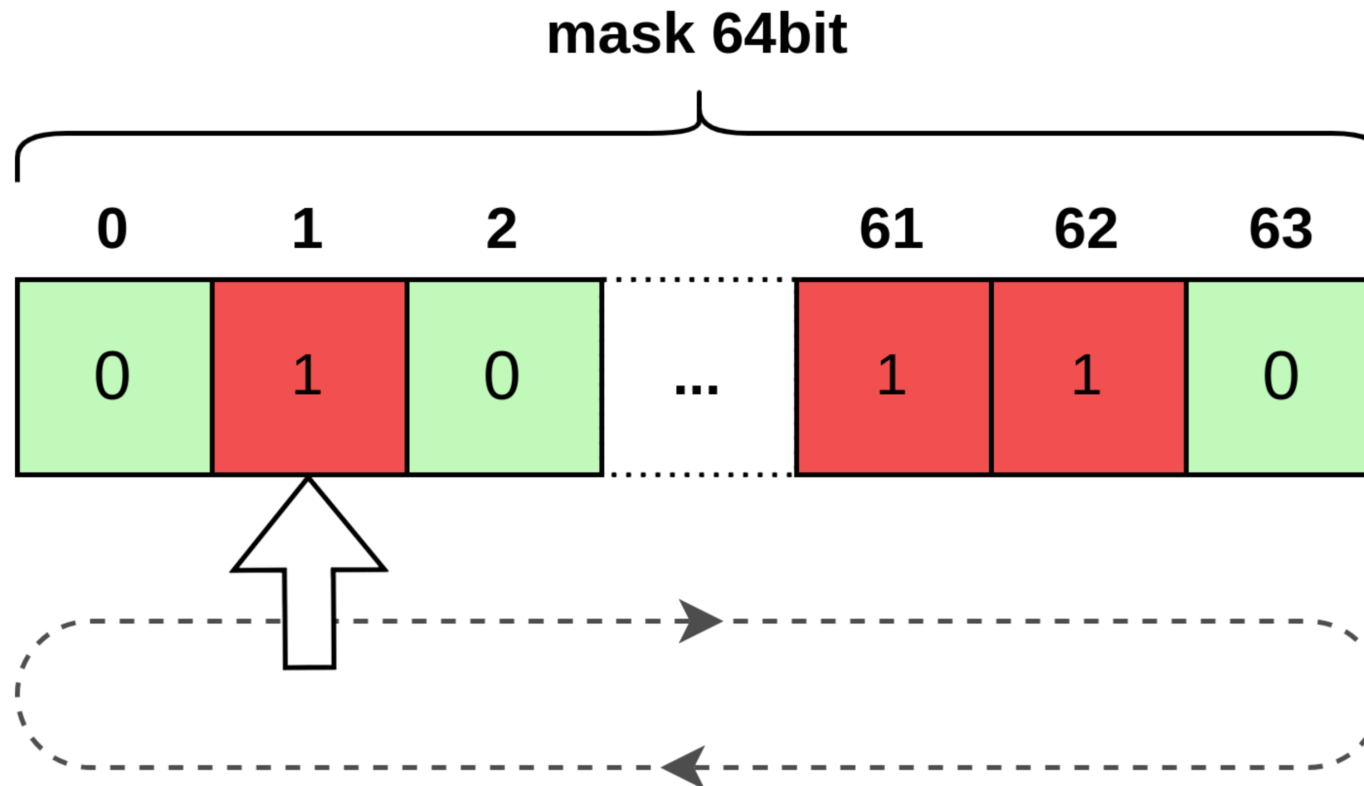

```
1  bool AllocationManager::IsFault() noexcept {
2      std::lock_guard<std::mutex> guard{m_mutex};
3      if (m_enabled) {
4          const pthread_t id = pthread_self();
5          if (m_threads.count(id)) {
6              size_t index = m_faultIndex++ % (sizeof(uint64_t) * 8);
7              uint64_t mask = 1;
8              mask = mask << index;
9              if (mask & m_faultMask)
10                 return true;
11         }
12     }
13     return false;
14 }
15
16 void* AllocationManager::Malloc(size_t size) noexcept {
17     if (IsFault())
18         return nullptr;
19     return real_malloc(size);
20 }
```

КАК РАБОТАЕТ ИНЪЕКЦИЯ ОШИБОК

```
1 bool AllocationManager::IsFault() noexcept {
2     std::lock_guard<std::mutex> guard{m_mutex};
3     if (m_enabled) {
4         const pthread_t id = pthread_self();
5         if (m_threads.count(id)) {
6             size_t index = m_faultIndex++ % (sizeof(uint64_t) * 8);
7             uint64_t mask = 1;
8             mask = mask << index;
9             if (mask & m_faultMask)
10                return true;
11        }
12    }
13    return false;
14 }
15
16 void* AllocationManager::Malloc(size_t size) noexcept {
17    if (IsFault())
18        return nullptr;
19    return real_malloc(size);
20 }
```

КАК РАБОТАЕТ ИНЪЕКЦИЯ ОШИБОК

ГРУППА КОМПАНИЙ ПРОСОФТ СИСТЕМЫ



КАК РАБОТАЕТ ИНЪЕКЦИЯ ОШИБОК

Что осталось за скобками:

1. Проблемы с многопоточностью.
2. Неявная рекурсия при подмене **malloc**.
3. Совместная работа с **Address Sanitizer**.
4. И многое другое.

ТЕСТОВОЕ ПОКРЫТИЕ С ИНЪЕКЦИЕЙ ОШИБОК


```
1 int make_history_item(const uint8_t* data, size_t size, HistoryItem **item) {
2     *item = (HistoryItem*)malloc(sizeof(HistoryItem));
3     if (NULL == *item)
4         return -1;
5     (*item)->data = (uint8_t*)malloc(size);
6     if (NULL == (*item)->data)
7         return -1;
8     memcpy((*item)->data, data, size);
9     return 0;
10 }
```

ТЕСТОВОЕ ПОКРЫТИЕ С ИНЪЕКЦИЕЙ ОШИБОК

Покрытие без инъекции ошибок

```
1 int make_history_item(const uint8_t* data, size_t size, HistoryItem **item) {
2     *item = (HistoryItem*)malloc(sizeof(HistoryItem));
3     if (NULL == *item)
4         return -1;
5     (*item)->data = (uint8_t*)malloc(size);
6     if (NULL == (*item)->data)
7         return -1; //<-- утечка памяти!
8     memcpy((*item)->data, data, size);
9     return 0;
10 }
```

Покрытие с инъекцией ошибок

```
1 int make_history_item(const uint8_t* data, size_t size, HistoryItem **item) {
2     *item = (HistoryItem*)malloc(sizeof(HistoryItem));
3     if (NULL == *item)
4         return -1;
5     (*item)->data = (uint8_t*)malloc(size);
6     if (NULL == (*item)->data)
7         return -1; //<-- утечка памяти!
8     memcpy((*item)->data, data, size);
9     return 0;
10 }
```

ВИКТОРИНА - НАЙДИ ОШИБКУ В ФУНКЦИИ


```
1 std::string handle_command(const uint8_t* data, size_t size) {
2     static std::list<std::unique_ptr<HistoryItem, void(*)(HistoryItem*)>> history;
3     HistoryItem *item = NULL;
4     if (make_history_item(data, size, &item) < 0)
5         throw std::bad_alloc();
6     history.emplace_back(item, free_history_item);
7     if (history.size() > 10)
8         history.pop_front();
9     const uint8_t cmd[] = {'h','i','s','t','o','r','y'};
10    if (size == sizeof(cmd) && memcmp(cmd, data, size) == 0) {
11        for (auto& it : history) {
12            (void)(it);
13        }
14        return "history";
15    }
16    return "";
17 }
```

ЧЕМ ПОЛЕЗНЫ ПРИВИВКИ ДЛЯ КОДА

1. Генерация исключений в функции **noexcept**.
2. Использование **free** для мусора.
3. Неправильная обработка исключений.
4. Утечка памяти.
5. Работа с **NULL** указателями.
6. Не анализируется результат выполнения функции.
7. Алгоритмические ошибки.

ЧЕМ ПОЛЕЗНЫ ПРИВИВКИ ДЛЯ КОДА


```
1 boost::shared_ptr<EVP_MD_CTX> ctx(EVP_MD_CTX_new(), EVP_MD_CTX_free);
2 EVP_DigestSignInit(ctx.get(), NULL, EVP_sha256(),
3     NULL, m_evp_private_key.get());
```

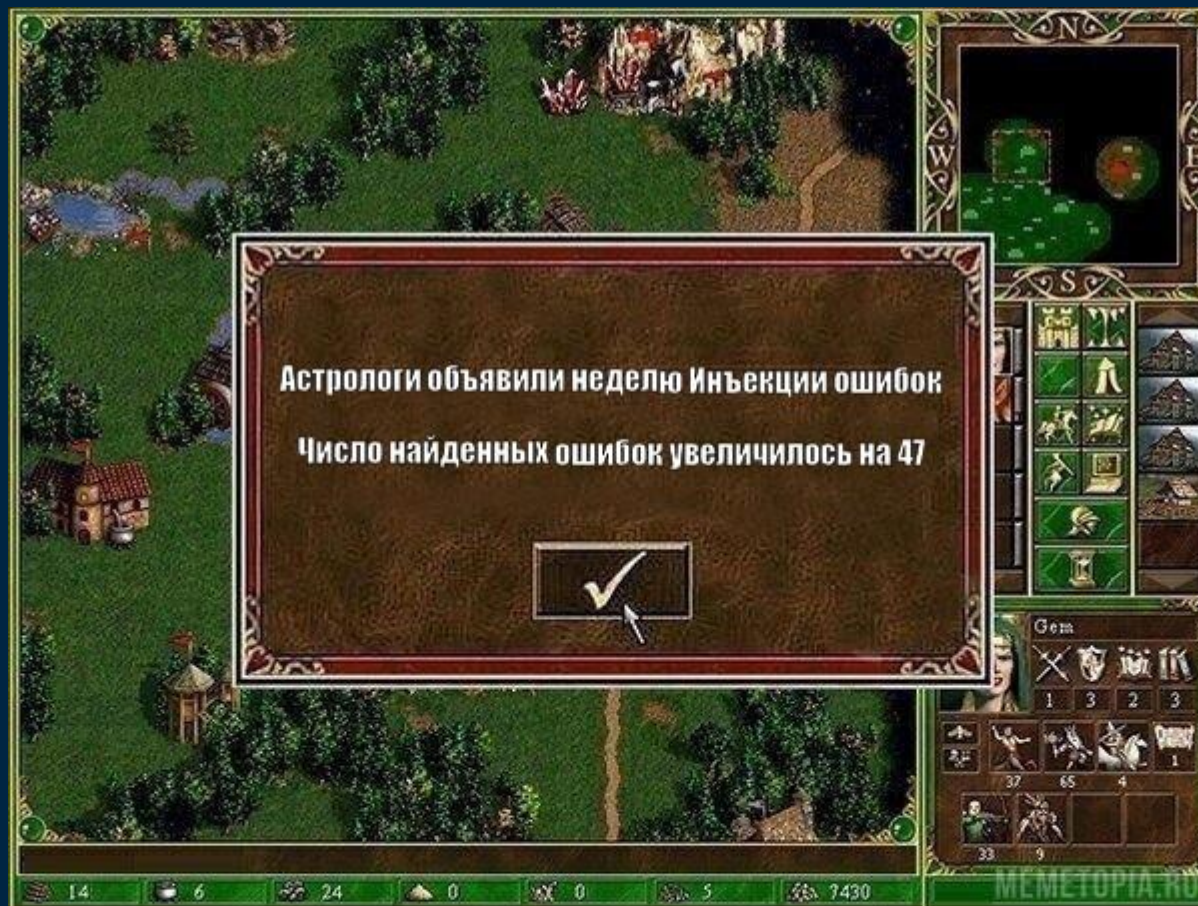
ЧЕМ ПОЛЕЗНЫ ПРИВИВКИ ДЛЯ КОДА


```
1 EVP_DigestUpdate(ctx.get(), &buffer[0], size);
2 unsigned int digestSize = static_cast<unsigned int>((*phash).size());
3 EVP_DigestFinal_ex(ctx.get(), &((*phash)[0]), &digestSize);
```

ЧЕМ ПОЛЕЗНЫ ПРИВИВКИ ДЛЯ КОДА


```
1 try {
2     // может быть выброшено исключение
3 } catch (const std::exception& e) {
4     // обработка негативного сценария
5 }
```

ЧЕМ ПОЛЕЗНЫ ПРИВИВКИ ДЛЯ КОДА



https://github.com/boostorg/date_time/issues/249

Memory leak in stream operators for `posix_time::time_duration` #249

Open

belyaev-ms opened last month

Memory leak in stream operators for `posix_time::time_duration`

The `<<` and `>>` operators for `boost::posix_time::time_duration` may cause memory leaks when an exception is thrown during memory allocation within these operators.

Steps to reproduce:

The following example simulates a memory allocation failure during the stream output operation for a `time_duration` variable. Custom `new` / `delete` operators are overloaded to force an exception on the 8th allocation call. The example should be built with AddressSanitizer (ASAN) to confirm the leak.

```
#include <vector>
#include <string>
#include <iostream>
#include <stdint.h>
#include <atomic>
```

Assignees

No one assigned

Labels

No labels

Type

No type

Projects

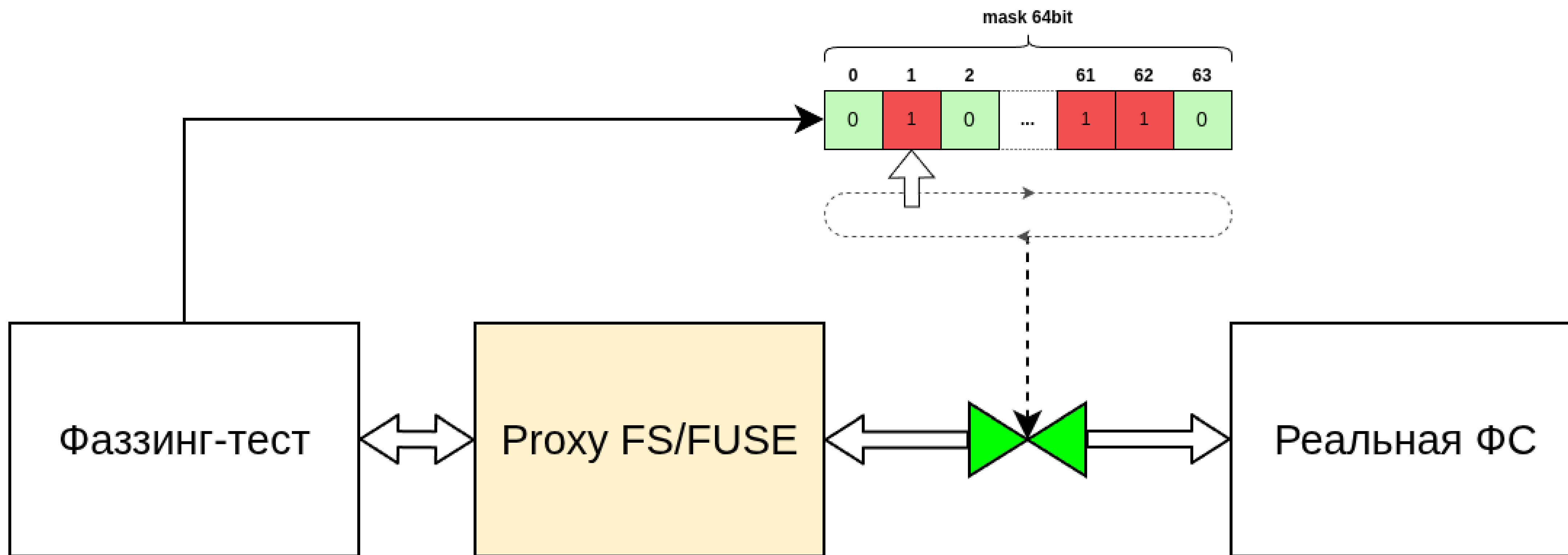
No projects

Milestone

No milestone

Relationships

ИНЪЕКЦИЯ ОШИБОК В ФАЙЛОВУЮ СИСТЕМУ



ОРГАНИЗАЦИЯ ПРОКСИРУЮЩЕЙ ФС

```
original_dir/
├── logs/
│   ├── test1.log
│   ├── test2.log
│   └── test3.log
├── config.json
├── history.bak
└── history.txt
```

```
proxy_dir/
├── .finj/
│   ├── enabled
│   ├── fault_mask
│   ├── filter
│   └── stat
├── logs/
│   ├── test1.log
│   ├── test2.log
│   └── test3.log
├── config.json
├── history.bak
└── history.txt
```

ОРГАНИЗАЦИЯ ПРОКСИРУЮЩЕЙ ФС

```
proxy_dir/
├── .finj/
│   ├── enabled
│   ├── fault_mask
│   ├── filter
│   └── stat
├── logs/
│   ├── test1.log
│   ├── test2.log
│   └── test3.log
├── config.json
├── history.bak
└── history.txt
```

файл для задания маски ошибок (64 бит)

ОРГАНИЗАЦИЯ ПРОКСИРУЮЩЕЙ ФС

```
proxy_dir/
├── .finj/
│   ├── enabled
│   ├── fault_mask
│   ├── filter
│   └── stat
├── logs/
│   ├── test1.log
│   ├── test2.log
│   └── test3.log
├── config.json
├── history.bak
└── history.txt
```

файл для включения инъекции ошибок (`0` или `1`)

ОРГАНИЗАЦИЯ ПРОКСИРУЮЩЕЙ ФС

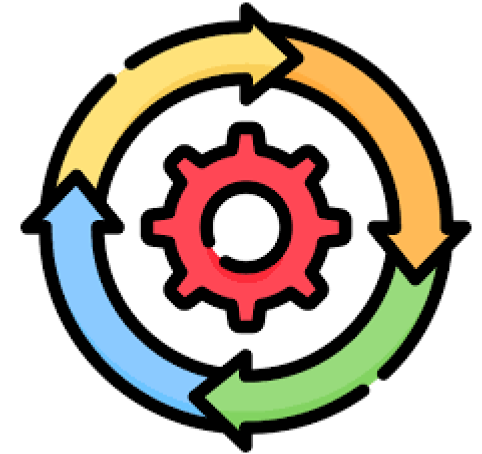

```
proxy_dir/
├── .finj/
│   ├── enabled
│   ├── fault_mask
│   ├── filter
│   └── stat
├── logs/
│   ├── test1.log
│   ├── test2.log
│   └── test3.log
├── config.json
├── history.bak
└── history.txt
```

файл для задания целей инъекции ошибок
пример "/history.*:/logs/test*.log"

ОРГАНИЗАЦИЯ ПРОКСИРУЮЩЕЙ ФС

```
proxy_dir/
├── .finj/
│   ├── enabled
│   ├── fault_mask
│   ├── filter
│   └── stat
├── logs/
│   ├── test1.log
│   ├── test2.log
│   └── test3.log
├── config.json
├── history.bak
└── history.txt
```

файл со статистикой работы ФС (stat)



АЛГОРИТМ РАБОТЫ ФАЗЗИНГ-ТЕСТА

1. Смонтировать проксирующую ФС в каталог, с которым фаззинг-тест будет взаимодействовать
   ```
   finj_fs --basedir=/original_path/ -f -o allow_other /proxy_path/
   ```
2. Откусить от входных данных фаззера кусочек для маски утечек и записать их в файл `/proxy_path/.finj/fault_mask`
3. Подать остальные данные на вход тестируемой функции.

ИНЪЕКЦИЯ ОШИБОК В ФАЙЛОВУЮ СИСТЕМУ


```
echo "/history.*:/logs/test*.log" > /path_for_test/.finj/filter
printf '\xDE\xAD\xBE\xEF' > /path_for_test/.finj/fault_mask
echo "1" > /path_for_test/.finj/enabled
```

ИНЪЕКЦИЯ ОШИБОК В ФАЙЛОВУЮ СИСТЕМУ


```
1 std::ofstream filter("/path_for_test/.finj/filter", std::ios::trunc);
2 filter << "/history.*:/logs/test*.log";
```

```
1 auto data = provider.ConsumeIntegral<uint64_t>();
2 std::ofstream faultMask("/path_for_test/.finj/fault_mask", std::ios::binary |
  std::ios::trunc);
3 faultMask.write(reinterpret_cast<char*>(&data), sizeof(data));
```

```
1 std::ofstream enabled("/path_for_test/.finj/enabled", std::ios::trunc);
2 enabled << '1';
```

ПРИМЕР ФАЗЗИНГ-ТЕСТА С ИНЪЕКЦИЕЙ ОШИБОК


```
1 extern "C" int LLVMFuzzerInitialize() {
2     fault_injection::filesystem::initialize();
3     fault_injection::filesystem::set_path_filters("/history.*:/logs/test*.log");
4     return 0;
5 }
6
7 extern "C" int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size) {
8     FuzzedDataProvider provider(data, size);
9     fault_injection::filesystem::set_fault_mask(provider.ConsumeIntegral<uint64_t>());
10     auto v = provider.ConsumeRemainingBytes<uint8_t>();
11     {
12         fault_injection::filesystem::FaultBreaker guard;
13         hard_fuzz_me(v.data(), v.size());
14     }
15     return 0;
16 }
```

СКРИПТ ДЛЯ ЗАПУСКА ФАЗЗИНГ-ТЕСТА


```
finj_run.sh /original_path /path_to_test [arguments]
```

При его выполнении:

1. Выполняется "подмена" оригинального каталога, через монтирование проксирующей ФС.
2. Запуск фаззинг-теста и ожидание его завершения.
3. Приведение оригинального каталога в первоначальное состояние.

КОМБИНИРОВАННЫЙ ФАЗЗИНГ-ТЕСТ


```
1 extern "C" int LLVMFuzzerInitialize() {
2     fault_injection::allocation::initialize();
3     fault_injection::allocation::register_thread();
4     fault_injection::filesystem::initialize();
5     fault_injection::filesystem::set_path_filters("/history.*:/logs/test*.log");
6     return 0;
7 }
8
9 extern "C" int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size) {
10     FuzzedDataProvider provider(data, size);
11     fault_injection::allocation::set_fault_mask(provider.ConsumeIntegral<uint64_t>());
12     fault_injection::filesystem::set_fault_mask(provider.ConsumeIntegral<uint64_t>());
13     auto v = provider.ConsumeRemainingBytes<uint8_t>();
14     {
15         fault_injection::FaultBreaker guard;
16         hard_fuzz_me(v.data(), v.size());
17     }
18     return 0;
19 }
```

ОШИБКА КАК СУСЛИК

Ошибка как суслик, ты ее не видишь, а она **есть**...