**Слайд 1**

Презентація орієнтована на Андроїд девелоперів в яких вже є досвід роботи з платформою андроїд і які тільки починають інтегрувати С++.

Або С++ девів які починають працювати з Андроїд платформою.

В цілому в презентації буде йти мова про типові помилки з якими можуть стикатись інженери і короткі поради як уникати таких проблем.

Я не буду розповідати про те як настроювати JNI підходи і як з ним працювати а буду фокусуюсь на тих проблемах з якими ми дуже часто зтикаємося на нашому проекті. Як приклад

**Слайд 2**

Я хочу сконцентрувати увагу на таких проблемах як типові помилки з роботою з ЖНІ посиланнями, проблеми які можуть виникнути при роботі з нативними потоками їхній взаємодії, також пару порад стосовно виключень.

Поговорим як зменшити розмір кінцевих апк файлів і як це потім впливає на роботу креш стеками

**Слайд 3**

Що таке НДК -

необхідний набір інструментарію для розробки компонентів програмного забезпечення для платформи [Android](https://uk.wikipedia.org/wiki/Android), який базується на [C](https://uk.wikipedia.org/wiki/C)/[C++](https://uk.wikipedia.org/wiki/C%2B%2B) та інших мовах програмування. Містить в собі лімітований набір загальновживаних низькорівневих (нативних) бібліотек та [API](https://uk.wikipedia.org/wiki/API), написаних на С/С++ та інших мовах програмування, документацію і мінімальний набір прикладів для демонстрації базового функціоналу. За допомогою NDK розробник застосунку для операційної системи Android може імплементувати окремі його частини, використовуючи такі мови, як C/C++

ЖНІ -

стандартний механізм для запуску коду, під управлінням [віртуальної машини Java](https://uk.wikipedia.org/wiki/%D0%92%D1%96%D1%80%D1%82%D1%83%D0%B0%D0%BB%D1%8C%D0%BD%D0%B0_%D0%BC%D0%B0%D1%88%D0%B8%D0%BD%D0%B0_Java) (JVM), який написаний на мовах [С](https://uk.wikipedia.org/wiki/C_(%D0%BC%D0%BE%D0%B2%D0%B0_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F))/[С++](https://uk.wikipedia.org/wiki/%D0%A1%2B%2B) чи Ассемблер, та скомпонований у вигляді динамічних бібліотек, дозволяє не використовувати статичне зв'язування. Це дає можливість викликати функції С/С++ з программи на [Java](https://uk.wikipedia.org/wiki/Java), і навпаки

**Слайд 4**

Для чого взагалі використовувати NDK якщо це приводить до збільшення часу написання коду і також на відладку?

Перш за все дуже багато старих проектів які були написані на С++ а потім потрібно інтегрувати під мобільну платформу. В нашому випадку андроїд. Скільки різних ігор написаних на ОпенГЛ було імпортовано на андроїд.

Інше це різні приклади роботи з секюріті і перфоманс. Більшість андроїд бібліотек які пропнують подібна жава інтервейси насправді делегують виклики до системних бібліотек.

Кожен може розпакувати апк файл і побачити хоч і обфускований але код. ДЕХ то ДЖАР і тому поідбні утиліти допомагають в цьому.

Для чого ми використовуємо НДК на проекті?

Але все таки на мою думку найбільша перевага це все таки можливість написання коду який одночасно можна викорустувати на різних платформах.

Наприклад на нашомe проекті ми пишемо СДК яке розповсюджується на різні мобільні платформи та також для макос і віндовс десктопів.

Структура проекту. Ми підтримуємо 5 різних платформ – перелічити. І для того щоб не імплементувати для кожної платформи наше СДК АПІ, в нас є С/С++ ядро. Воно покриває

Різні секьюрті протоколиб шифрування і збереження даних , нетворкінгб ауненифікація,

Різні внутрішні протоколи і тому подібне.

І коли для обжектівСІ та Віндовс Win32 нема проблем напряму комунікувати з С++ кодом (не беручи до уваги ), то для джави нам потрібно додавати додатковий рівень \_ ЖНІ.

Що може додами кучу проблем. З прешої точки зору код може виглядати цілком нормально, але під час роботи програми ми можемо зтикатися з різними проблема.

Якщо ми детектимо проблему під час дебагу то це цілком нормально. Але, якщо вже в релізному продукті – то це нетавні відгуки кастомерів. Тож полговоримо про типові помилки -

**Слайд 5**

**Слайд 6**

Перше на що я хочу звернути увагу – це локальні та глобальні посилання.

ПОказати типовий проект -

Для нормального збереження та передачі даних між С++ та Джавою ми використовуємо такі структри даних як локальні глобальні то слабкі посилання. В принципі це фраппер над виділенним ресурсом. Тут можна провести аналогію з жава обьетами хоча тут є свої нбанси. Локальне посилання буде автоматчно очистена ДЖВМом як тільки ми покинемо нативний метод. І тому якщо ми хочемо зберегти її на деякий час потрібно зробити з неї глобальне посилання. Глобальна в свою чергу не буде очищена поки ми не почистимо її вручну. Тож якщо ми хочемо зберегти обьект десь в С++ коді то потрібно зберігати її в вигляді глобального реверенс.

Веак референсе

Але:

Хоч і система чистить локальні посилання автоматично, все таки це не так. Всі локалні посилланя будуть очищені як тільки курування покине нативний метод і повернеться до віртуальної машини жави, і тому локалні посилання будуть накопичуватись (і навіть перезапис не допоможе). Що в результаті може приветси до такого крушу(показати креш)

Як бачимо по дефолту система дозволяє нам дозволяє створити лише 512 локальних посилань – це і є таблиця локильних посилань. Ми навіть можемо побачити декілька останніх в креш репорті. І тому тут можуть опинитися навіть паролі.

Ми можемо збільшити розмір даної таблиці але найкращий варінт одразу оищати посилання в ручну як тільки воно стає не поібним. (Приклад коду).

Це саме стосцється і глобальних посилань, але правда розмір таблиці більший(код і креш) але все одно не потрібно зловживати ними. Як порада – мінімізувати використання глобальних посилань.

Також якщо зберігаєте джава пір обекти як глобальні посилання в С++ коді – думайте що обьетк вже можу бути не валідним на стороні Джави. Наприклад актівіті –

Код.

В андроїді 8+, системо підтримує безлімітну кілкість локальних посилань, але все одно – не забувайти релізити- кількість 8 андроїдів хоч і зростає але все одно більшість девайсів працють на старіших версіях. Порада не тестуйте все на одномі девайсі – пробуйте перевіряти код на декількох

**Слайд 7**

При роботі з ЖНІ рекомендується зменшувати взаємодію з декільками потоками і по можливості робити все в одному потоці, але з розростанням проекту ми

можемо створювати велику кількість нативних потоків безпосередню в С++(POSIX, C++11) які можуть взаємодіяти між собою, але іноді виникає необхідність викликати якісь специфічні функції на стороні платформи. Як приклад – стан нетворку, телефонії, оріентація екрану, бекграунд тощо.

З кожною версією НДК набір АПІ збільшується, але все одно така необхідність буде залишатися.

Якщо ми просто викличемо якийсь Жава метод через JNI API в нативному потоці, то просто отримаємо креш – чому?

JNI визначає дві основні структури даних - "JavaVM" і "JNIEnv". Обидва ці по суті є вказівниками на функціональні таблиці.

JNIEnv забезпечує більшість функцій JNI. Але JNIEnv використовується для доступу виключно до threаd local storage. З цієї причини ви не можете поділитися JNIEnv між потоками.

І тому як нам правильно викликати JNI функціїї якщо ми не можемо не предети ЖНІЕНВ якщо ми не можемо не предевати її до іншого потоку ні зберігати десь в глобальній змінній.

Для цього JavaVM надає функції "інтерфейсу виклику. – Інвокатіон АПІ

По суті апі дозволяє створювати віртуальну машину та завантаження та використання класів Java. Для андроїд програм це робити не портібно, а використовуємо АПІ для використанян класів Java в декількох нативниї потоках.

Тому маючи вказівник на JavaVM ми можемо використовувати

AttachCurrentThread або AttachCurrentThreadAsDaemon функції які посуті притачать наш потік до віртульної машини

До тих пір, поки потік не буде додано потоку, він не має власного JNIEnv і не може робити виклики JNI. Приєднання потоку до ЖВМ створить об'єкт java.lang.Thread на стороні жави, який буде доданий до "main" ThreadGroup, роблячи його видимим для системи і дебагера.

Такі потоки, додані через JNI, повинні викликати DetachCurrentThread, перш ніж вони виходять.

Якщо у нас нема вказівника на ЖВМ то [JNI\_OnLoad](https://docs.oracle.com/javase/9/docs/specs/jni/invocation.html) функція використовується для отримання поінтера на ЖВМ.

Для цього використовуємо функцію [GetEnv](https://docs.oracle.com/javase/9/docs/specs/jni/invocation.html).

Так як і ЖНІЕНВ локальних посиланью. Їх не можн ашарити між потоками. Тобто

Навітіь якщло нативний метод одного потоку ще не закінчив роботу а ми створили локальне посилання і додали його в якийсь глобальний контейнер, інший потік не може його використовувати.

Давайте ці пробеми подивимося на приклід коду

Код. Шляхи вирішення

**Слайд 8**

Тепер пару слів про типові проблеми лінковики Джава і С++.

Перше часто під час написання коду і подальшому запуску ми не бачимо ніяких проблем під час компіляції а вже потім в рантаймі ми отримуємо креші такого типу (перелічити).

Типові помилки: змінили сігнатуру методу в джаві, перевірили Андроїд студією що всі викліки коректні. Але потім JNI вже в рантаймі не може знайти імплементацію

Чи змінили сігнатуру функцій ЖНІ хедеру.

Також не потрібно забувати що ЖНІ функції мають С а не С++ тип лінковки, і тому не потрібно забувати додавати екстерн С . Що це таке - В С++ ми можемо перегружати функції і методи і тому для правильної ліоквки С++ компілятор додає додаткову інформацію про функції в її ІД. Які потім будуть додані до символьної таблиці.

І тому під час лінковуи жави і С++ бібліотеки в рантаймі ми шукаємо саму функції що мають С тип ліноковки.

Дуже часто на нашому проекті ми зтикаємо с з такою – пишемо новий функціонвл на стороні Джави і С++ компілимо тестимо на локальній машині – все працює – сабмітимо зміни в репозиторій. Але через деякий хтось каже що ми поламали білд – в чому проблема –

обфускації жава коду в релізнму білді яку ми не перевірили.

Після обфускації жава код стає не хюман редабле що також є проблемою для ЖНІ де ми також покладаємося на точні імена класів і методів.

Тут як варінт нам потрібно додавати різні виключення в правила настройки прогварда і дексгварда. Тобто ті методи та класи ЖАва які потрібні ЖНІ не обфускувати.

Також якщо додаємо пітримку різних архтектур як (перелічити) – потрібно пересвідчитися що і інші бібліотеки які ми додамо в проект також пітримують дані архітектури. Приклад

Ми релізимо сдк з пітримкою 3 архітектур. 64 платформа якщо не знайде найбіль підходящу арх то вибере наприклад 32. Якщо вибере

Варто зазначити всі перераховані проблеми на етапі компіляції не детектаться а можуть лише в рантаймі і навіть тільки після якихосб специфічних дій користувача і тому тут потрібно приділяти дуже багато уваги до таких проблем.

**Слайд 9**

Інше на чому хочу звернути увагу – це підтримка експшинів в С++, JNI та джаві –

ексепшни в С++ ця функціональність є не опціональною і можуе бути вимкнена для підвищення швидкодії і зменшення розміру бібліотк і виконуваних програм.

І тому якщо ми під час компіляції бачимо щось подібне – то значить фічу потрібно вімкнути.

Для цього додаємо це як С++ білд флаг. .

Тут одне вмикає інше вимикає вічц відповідно..

Під час таких змін варто робити клін білду і рефрещити прилінколваний С++ проект (В вндроїд студії) щоб наші зміни були точно застосовані

**Слайд 10**

Далі – наприклад ми хочемо викликати якись метод на стороні Джави який може генерувати експпш. Дуже багато Джава АПІ які кидають різні ксепшини які не є фатальними для роботи програми. І якщо експшн був кинутий на стороні Джави то на повернуний результат ми не можемо покладатися хоча ми і отрумумо результат В с++. Якщо в потоці висить пендінг ексепшин то ми більше не можемо викликати інші методі на стороні жави через JNI API. Є лімітований набір методів які ми можеми викликати поки у нас висить пендінг ексепшн здебільшого звязаний з очисткою ресурсів. Якщо ми спробуємо викликати інші методи – JNI викине помилку і за абортить програму.

Для цього нам після виклики потрібно перевіряти пендінг експшнини. Для цього викоритосвуємо дану функцію –

Для того зоб отримати тип експшина ми може використати функцію. Вона повертає jthrowable який ми може розпарсити і отримати тип ексепшина. Але Find class не може бути викликаний поки э пендінг – або чистимо експшн або кешуємо класс на ще до виклику методу. Ось типовий приклад - код

Ексепшн можна почисти функцією і працювати далі – в іншому випадку отримаю креш на рівні апплікейшина.

Також для Для підтримки стилю джави ми можемо кидати експшн з С++ коду на сторону Джави.

Для цього використовуємо пару функцій –

Приклад ось

Хочу зазначити що на стороні джави ескпшн буде викинутий тільки коли нативний метод поверне керування в ЖВМ. Тож навіть після того як ми викликали ThrowNEw ми можемо ще виконати якысь дії на стороні С/С++. Але це виглядає не логічно

**Слайд 11**

Додавання С++ рівня в проект додає суттево в розмірі остаточного АПК файлу і тому одна з задач яке перед нами може стати це зменшення бібліотек.

Перше що тут скащати що на выдміну від джава коду який компілюється в dex формат без різниці яка архітеркура процесора – це задача АРТ (чи далвіка для більш старих платформ) –які вже потім будуть самі адаптувати dex byteкод під платформу. – то з С біблотеками нам потрібно предаставляти по лібі для кожної платформи – ось типовий релізний апк файл – бачимо що ми маємо один декс файл і аж чотири динамічні со бібліотеки. Тут потрібно думати чи хочемо ми супортити ту чи іншу платформу і чи потрібно наприклад супортити 64 арм архітектури бо іноді може бути достаньо і 32 – вони компатібл але не навпаки . Інтел мало розповсюджені але емулятори саме інтел.

Також як було зазначено, використання експшинів приводить до генерації додаткового коду і що значить до збільшення фінальної бібліотеки –використанням флагів компілтора можна вмикати вимикати фічу і як реультат поступове зменшення бібілотеки для кожної архітектури

RTTI – також фіча мови С++ що дозволяє в рантаймі дізнаватися тип обьекта і тому піодбне – це також приводить до генерації додаткового коду компіляторм. ТАкож можна вимкнути якщо це вам не потрібно.

Більше користі нам дасть зрізання символів і таблиць символів які можуть йти разом з динамісною бібліотекою.

Ефективним способом зменшення розміру бінарного коду є використання функції видимості gcc. Ця функція дозволяє вам контролювати, які функції буде експортовано до таблиці символів.

JNI поставляється з макросом JNIEXPORT, який позначає функції JNI як публічними. Вам просто потрібно перевірити, що всі функції, що використовуються JNI, префіксуються JNIEXPORT, як це.

Інший цікавий підхід - видалити невикористовуваний код у двійковій системі. Це може суттєво зменшити його розмір, якщо, наприклад, частина вашого коду використовується лише для тестів.

Щоб увімкнути цю функцію, потрібно просто змінити прапори компіляції C та C ++ та компонувальника. Але не забувайте що це може привести до не можливості дебагінгу С++ коду

І тому потрібно коректно конфігурувати нащ білд

**Слайд 12**

Для додавання всіх параметрів компіляціх нам потрібно правильно передати ці флаги компілятору.

Зараз Android рекомендує використовувати смаке як тула для білда ндк коду, хоча ndk\_ build ще також підтримується. Він використовує лайт версію маке файлів – applicaiotn mk

Якщо використовуємо Смаке то можна передавати чи через білд градл – наприклад. Наспраді параметри компіляціїї можна передавати так само і для ndk build ынтсруменут

Чи напряму в Смаке ліст ткст файл – варто зазначити що нам окремо потрібно передавати флаги для с++ так і для С.

Також можна вибирати релізний чи дебажний режим.. От якрас не рекомендуэться оптимызувати код для дебар режиму через різні особливотсі процесу дебагу

В аplication mk файлі це буде мати вигляд – . Окремо можна також передавати флаги лінковки – на цьому уваги акцентувати не буду.

**Слайд 13 Crash handling**

Так всі маніпуляції з кодом було виконано – все оптимізовано все працює – ми релізимось в плей маркет – і отрмуємо перші репорти в Dev consoli –

**Слайд 14**

Приклад типового крешу який можна знайти на гугл дев консолі в Android Vitals .

Як бачимо – типова статисткиа по версім продукту, андроїда і пристроям.

І беспосередньо сам креш –

Як бачимо не дуже багато корисної інформації для початку. Бектрейс не показує ніяким методів чи символів (на відміну від джава стеку)

Зменшення розміру Зрізання різних символів приводить до таких наслідків як повністю обфускований стек крешу. Це корисно для різного захисту від пен тестерів але як бачимо ми також не можемо отримати миттєво ніякої інформаціїї.

Тож давайте поглянемо на більш детальний tombstone креш і розберемося що для чого

І як та інформація що в нас є допоможе нам

**Слайд 15**

Коли запускається dynamically linked executable файл, реєструються декілька оброблювачів сигналів, які в разі виникнення збою призводять до написання основного дампа аварійного завершення роботи в logcat та більш детального файлу " tombstone ", який буде записаний до / data / tombstone /. tombstone - це файл із додатковими даними про крешнутий процес. Зокрема, він містить стек трейси для всіх потоків у крешнутому процесі (а не тільки потоку, що спіймав сигнал), повну карту пам'яті та список всіх відкритих дескрипторів файлів.

Приклад детального кресування згенерованого на тестовому проекті -

Лінія зірочок з пробілами корисна, якщо ви шукаєте журнал для лакальних збоїв. Рядок

"\*\*\* \*\*\*" рідко з'являється в логах, відмінних від крешів.

1 fingerprint дозволяє точно визначити, на якому місці відбулася креш

revision стосується апаратного забезпечення, а не програмного забезпечення. Зазвичай це не використовується, але може бути корисним для автоматичного ігнорування помилок, які, як відомо, виникають через погане hardware.

ABI Application Binary Interface - це одна з arm, arm64, mips, mips64, x86 або x86-64. Це в основному корисно для скриптів і тулзів обробки стека, які я розгляну потім

Цей рядок визначає конкретний потік у процесі, який зазнав крешнувся. У цьому випадку це був основний потік процесу, тож відповідає ідентифікатор процесу та ідентифікатор потоку. Ім'я - ім'я потоку, а ім'я оточене >>> і <<< - це ім'я процесу. Для додатка ім'я процесу, як правило, є повноцінним ім'ям пакета (наприклад, com.facebook.katana), що корисно при подачі помилок або спробі знайти додаток у Google Play. Під і тід може також бути корисним для пошуку відповідних логів, що передують крешу.

Ця лінія повідомляє вам, який сигнал (SIGABRT) отримано, та інше про те, як він був отриманий (SI\_TKILL). Сигнали, про які повідомляють debuggerd, є SIGABRT, SIGBUS, SIGFPE, SIGILL, SIGSEGV та SIGTRAP. Коди, специфічні для сигналу, залежать від конкретного сигналу.

2 - регістри

Дамп реєстру показує зміст регістрів ЦП під час отримання сигналу.

Наскільки корисними вони будуть, це буде залежати від крешу.

3 - bactrace

backtrace показує вам, де в коді ми були під час крешу. Перший стовпець - це frame . Значення PC відносно розташування спільної бібліотеки, а не абсолютні адреси. Наступний стовпчик - це назва mapped region, нарешті, якщо символи доступні, відображається символ, який відповідає значенню PC ,

разом із зміщенням на цей символ у байтах.

tombstone містить таку ж інформацію, що й аварійне звалище, а також кілька додаткових матеріалів. Наприклад, це включає в себе backtraces для всіх потоків

**Слайд 16**

Що нас тут цікавить так це сам бектрейс бо тільки він присутній в типових репортах на Гугл дев консолі.

Для декодування і отримання більш детальної інформації по бексрейсу нам потрібна така утіліта як ндк стек.

Інструмент ndk-stack дозволяє фільтрувати обфусковані стеки, як вони з'являються на виході adb logcat. Він також замінює будь-яку адресу в бібліотеці за допомогою відповідних значень <source-file>: <number-line> з вашого вихідного коду, що полегшує їх виявлення.

На вході вона нам потрібно передати песпосередньо сам бектрейс і шлях до бібліотеки з включенеми таблицею символів - вона може бути знайдена в білд оутпуті cімека чи ндк білда (приклад смаке) і тому варто завжди зберігати такі дані десь на сервері і мапати їх до тої версії апплікейшинів.

Також, потрібно вказувати версію білотеки для тої архітектури де стався креш.

Прогнати команду – показати результат

Як бачимо результат трохи запутаний але якщо правильно його відфільтрувати то можна отримати точне місце креша. Ось. Знайдемо його в коді

**Слайд 17**

**Показати приклад що було що стало**

**Слайд 18**

Ми також можемо написати свій креш хендлер. Для цього його потрібно правильно ініціалузіувати. Ми можемо перехоплювати всы системні сигнали і використовувати його для власних цілей.

system call is used to change the action taken by a

process on receipt of a specific signal. (See [signal(7)](http://man7.org/linux/man-pages/man7/signal.7.html) for an

overview of signals.)

можна установлювати декылька сигнла хенделерівб

але дефолтний андроїд сигнала хенделр взажмодіє з debuggerd – debugger deamon який насправді і виводить наш креш стек в логкат.

Таким чином для додавання додатковоъ логыки ми можемо викоритосувати новий сигнла хендлер, ы одночасно залишати старий який буде робити всю ыншу роботу.

**Слайд 19**

**Є певні особливості при роботі з креш хедлерами – обмеження**

Але під час обробки сигналів ми не можемо використовувати майже нічого через

ASync safety – всі функції в середині сігнал хендлеру повинні бути бути рентрант

Бо весь наш сигнал хндлер повинне бути ренетрант

ЩО це значить

Ми не можемо викликати функції які мають глобальні сайд ефекти – тобто ми не можемо

Захоплювати різні локи та мютекси і як результат доступатися до якихось глобальних змінних.

Щому саме таке обмеження – уявіть ситуацію що ви отримуєте сигнал, викликається сигнал хендлерб захоплюється мютекс і код заходить далі в критичну секцію. В цей самий час інший сигнал може бути доставлений до нашого процесу - і що ми маємо ми будемо чекати на мютексі що не є допустимим. Тобто ми не можемо виділяти пам'ять – і як наслідок не можемо викоритосвувати більшість функцій і методів з бібліотек лібс і лібс++.

НЕасправді є лімітований набір функцій які є async safe –

Інше обмеження ми не можемо мати доступ до даних які живуть в статік стораж – тобто різні статичні змінні та також глобальні –

Чому – уявіть ще одну ситауцію – стартує програма – починають ініціалізуватия всі глобальні обьекти в конструкторі якогось кидається сигнал.

Потім ми ще й хочемо отримату доступ до цього обьекту – що буде – undefined behaviour.

Що значить – ми не можемо передавати ныякий стейт до нашого хендлеру – як приклад імя файлу в який ми хочемо щось записати стосовно крешу.

Маючи таке обмеження взагалі дуже важко щось обробляти і тому більшість креш хендлерів – порушують – технічно це не правильно але прраткично – це працює

Далі – ще складніше – нам потрібно якось відновити стек викликів. Є декілька ндк бібліотек

Які дають змогу відновлювати стек але як розуміємо – обмеження

Більшість інструментів

Тут якщо нам не потрібно нічого особливого – достаньо того що є

Або використання інших утіліт Crashlytics, Breakpad

І тому якщо нам не потрібно нічого особливого а вистачить того що нам дає дебагрд – то цього буду достатньо – якщо ні – то це вже тема насткпної доповіді