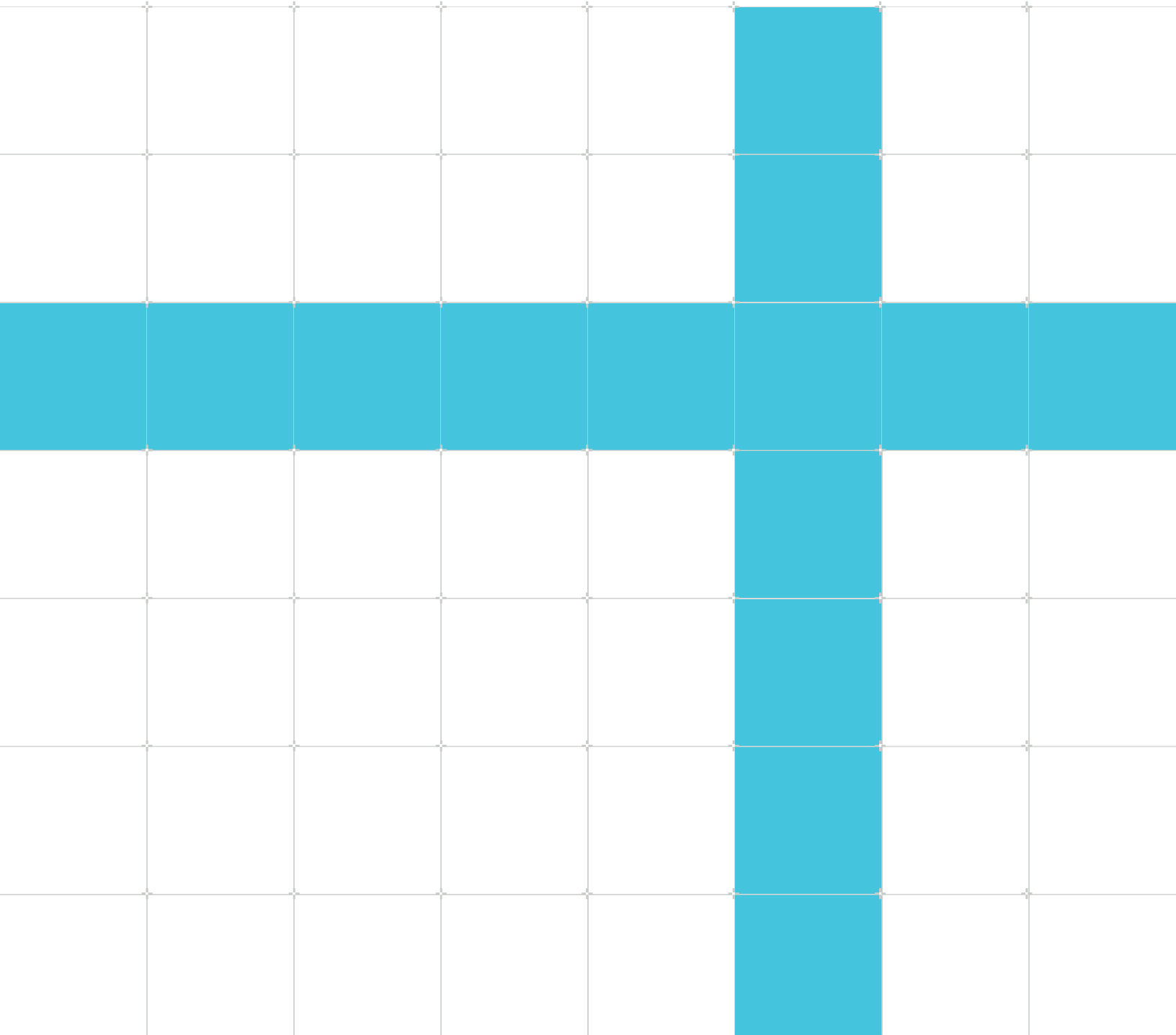
适用于 Android 操作系统的 MTE 用户指南

1.0 版

发行号 01
109279_0100_01_zh



适用于 Android 操作系统的 MTE 用户指南

版本信息

文档历史记录

发行号	日期	保密性	变更
0100-01	2023 年 5 月 25 日	非机密	最初版本 Chinese translation of 108035_0100_01_en

所有权声明

档，确保本文档或其任何部分不会在违反相关出口法律法规的情况下直接或间接出口。在提及 Arm 的客户时使用“合作伙伴”一词并不意味着 Arm 与任何其他公司建立合作伙伴关系，也不指示与任何其他公司存在合作伙伴关系。Arm 可以随时更改本文档，恕不另行通知。

为了方便使用，本文档可能被翻译成其他语言。您同意，本文档的英文版与翻译版之间出现冲突时，应以英文版条款为准。

Arm Limited。公司注册地为英国，注册号为 02557590。

110 Fulbourn Road, Cambridge, England CB1

9NJ.（LES-PRE-20349| 21.0 版）

保密状态

本文档的内容非机密。根据 ARM 与从 ARM 接收本文档的接收方签订的协议条款，使用、复制和披露本文档内容的权利可能会受到许可限制。

“非受限访问”是 ARM 内部的一种分类。

产品状态

本文档中的信息是开发完成的产品的最终信息。

反馈

Arm 欢迎对本产品及其文档提出反馈。如要提供产品反馈，请前往 https:/support.developer.arm.com 创建工单。

如要提供文档反馈，请填写以下调查问卷：https:/developer.arm.com/ documentation-feedback-survey。

包容性语言承诺

Arm 重视包容性社区。Arm 认识到我们和我们的行业使用的语言可能具有冒犯性。Arm 努力引领行业，开启变革。

我们相信本文档不包含任何冒犯性语言。如需报告本文档中的冒犯性语言，请发送电子邮件至 terms@arm.com。

目录

1. 内存标记扩展简介

ARM 在 Armv8.5 体系结构推出了内存标记扩展（Memory Tagging Extension，MTE）技术。MTE 是 Arm 体系结构的一个重要增强特性，通过检测和处理与内存相关的漏洞，提高连接设备的安全性。

本使用指南将介绍 MTE，向开发者展示如何使用 MTE 来增强软件的可靠性和安全性。

本指南包含以下内容：

- 内存安全错误：解释为什么需要 MTE 来检测内存错误。
- MTE 的工作原理：介绍 MTE 解决内存错误的原理。
- 在 Android 项目中启用 MTE：介绍在 Android 项目中启用 MTE 的几种不同的方法。
- MTE 错误报告：如何获取和解读 MTE 检测到内存错误后发出的错误报告。
- 利用 Android Studio 和 MTE 进行调试：介绍如何使用 Android Studio 执行代码调试，定位内存错误。
- 将 MTE 集成在内存管理系统中：指导用户实现自己的内存分配器。

Google 目前正在实施 MTE beta 版设备注册计划。如果您感兴趣，可点击以下链接注册：

- MTE beta 版设备注册

2. 内存安全错误

内存安全错误是指软件在处理内存时出现的错误。内存安全错误可能会在以下情况中出现：

- 软件以超出其获配的大小和内存地址的方式访问内存。这种情况称为空间内存安全错误。
- 软件访问超出数据预期生命周期的内存位置，例如在内存被释放并重新分配之后访问。这种情况称为时间内存安全错误。

Google 的内存安全报告指出，本地代码中的内存安全错误仍然是终端崩溃的主要原因，对质量和稳定性产生负面影响，并且是安全漏洞的最大来源。

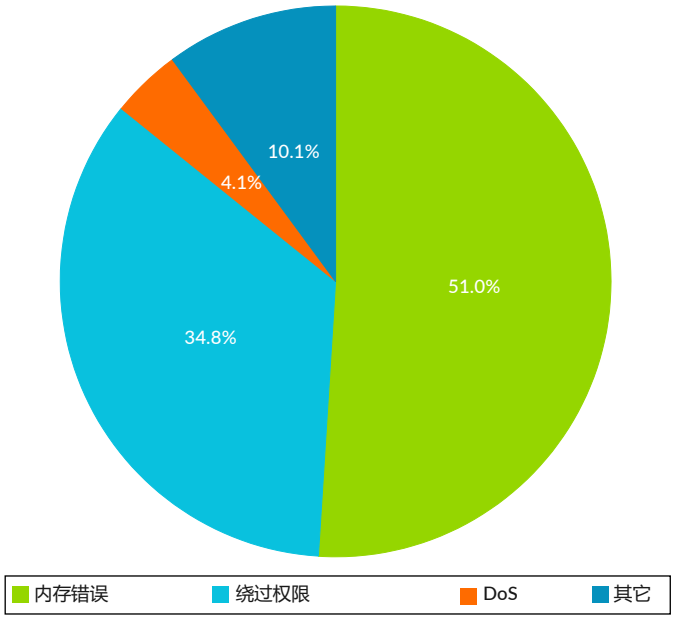
内存安全错误是 Android 代码库中最常见的问题，占高严重性安全漏洞的 70% 以上，导致数百万次用户可见的崩溃。

使用内存不安全语言（如 C、C++ 和汇编）编写的本地代码占 Android 平台代码的 70% 以上，Play Store 应用程序中大约有 50% 使用了此类本地代码。

内存错误会给质量和稳定性带来负面影响。终端设备上出现的崩溃，有很大一部分是内存错误造成的。高密度的内存安全错误导致用户体检不佳。

内存安全错误一直以来都是造成 Android 安全漏洞的主要原因。Google Project Zero 团队分享了他们的零日攻击追踪电子表。这份电子表格显示，在用作安全漏洞以实施攻击的错误中，大部分属于内存损坏问题。下表显示了，内存安全错误是导致 Android 安全漏洞的最大原因。数据来源：《Android 文档：内存安全》。

图 2-1：内存安全错误在 Android 系统漏洞中的占比



Android 生态系统每年在更新最新的安全修复程序上花费数百万美元。低级厂商代码中的内存安全错误密度很高，不仅增加了修复成本，也增加了检测成本。不过，在开发早期对这些错误进行检测可以降低成本。研究表明，如能尽早检测错误，成本可降至 1/6。

但是，检测和修复内存安全错误一般很难。要检测出内存错误，代码必须先触发错误。因为内存错误通常是间歇性的，且难以复现，所以，检测和修复错误通常既耗财又耗时。

Google 将在 Android 12 及更高版本系统中推出系统变更，来减少 Android 代码库中的内存安全错误。为此，Google 致力于：

- 扩展 Android 内存安全工具
- 发布新要求，鼓励 Android 生态系统解决内存安全错误

在这些新要求中，Android 兼容性定义文档 (CDD) 强烈建议在开发、持续集成 (CI) 和检测阶段使用内存安全工具。

现有的内存安全工具包括：

- AddressSanitizer (ASan)
- HWAddress Sanitizer (HWASan)
- Kernel Address Sanitizer (KASan)

这些工具在每个内存操作中进行编译器插桩，帮助检测广泛的内存安全错误。但是，这给性能、代码大小和内存占用带来了巨大的开销。

MTE 有效且易于使用，对性能的影响较小，占用的内存空间也较小，开发者可以利用这种工具检测内存错误。

2.1 常见的内存安全错误

内存安全问题分为两类：空间内存安全问题和时间内存安全问题。

空间内存安全问题包括缓冲区溢出漏洞。缓冲区溢出是指固定长度的缓冲区内存入了过多的数据，导致数据溢出到相邻的存储区域中。使用允许指针操作的语言（例如 C 和 C++）编写的应用程序中会出现这种问题。程序员必须确保指针保持在分配对象的边界内。

以下示例演示了缓冲区溢出错误：

```
extern "C"
JNIEXPORT void JNICALL
Java_com_example_mte_1test_MainActivity_heapOutOfBoundsC(JNIEnv *env, jobject thiz)
 {
    char * volatile p = new char[16];
    p[16] = 42; // Trying to access a non-allocated array element.
    delete[] p;
    p[0] = 42; // Use-after-free error
}
```

时间内存安全问题与内存位置在程序执行的不同时间包含不同数据有关。应用程序释放内存并重新分配时，无法假设原始数据仍然存在于内存中。这种问题通常在程序执行以下操作时出现：

1. 创建指向某个内存的指针。
2. 释放内存但保留原始指针。这种指针称为悬空指针。
3. 试图使用悬空指针来访问数据。

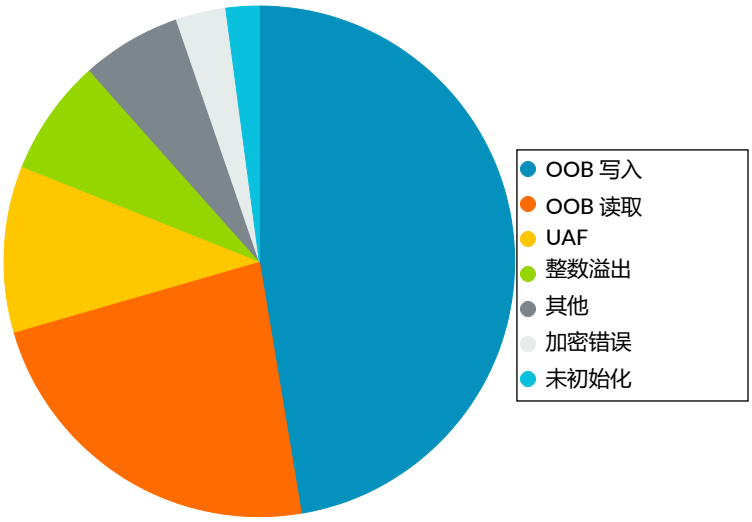
这种情况称为释放后重用 (UAF) 错误，会导致出现未定义的行为，此外，还会带来信息泄漏或攻击者接管应用程序的风险。

MTE 能检测出导致内存错误的最常见原因：

- 释放后重用
- 缓冲区溢出
- 双重释放

根据 Google 安全博客文章，大多数 Android 漏洞是由 UAF 和越界 (OOB) 读写引起的，如下图所示：

图 2-2：内存漏洞产生的原因



3. MTE 的工作原理

Arm 与 Google 合作开发了 MTE，用于检测现有代码库和新编写的代码中的内存安全错误。MTE 提高了以 C 和 C++ 等内存不安全语言编写的大型现有生态系统的内存安全性。

MTE 可帮助您：

- 提高检测和模糊测试的有效性，在部署前发现潜在的漏洞
- 在部署后大规模检测漏洞

在 Android 12 中，内核和用户空间堆内存分配器可以通过元数据来增强每次分配。MTE 能够利用元数据检测出 UAF 和堆错误。这些是 Android 代码库中内存安全错误的最常见来源。这种使用模型称为堆标记，只有需要分配或释放内存的代码路径才需要了解 MTE。
因此，堆标记可以以向后兼容的方式部署，而且大多数应用程序不需要为此修改源代码。

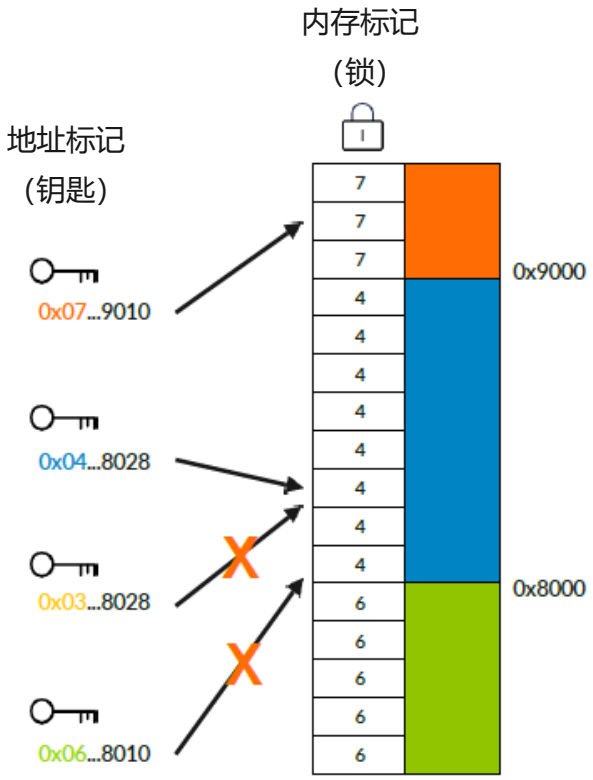
MTE 还有一种使用模型称为栈标记，即也标记在运行时堆栈上分配的内存。不过，这种技术的入侵度更高，要求重新编译，并且不可与旧版设备向后兼容。

MTE 使用基础的锁和钥匙模型来访问内存。运行时，当分配或释放内存时，该内存区域的一部分内存地址将被分配一个 4 位内存标记，也就是锁。使用指向该地址的指针进行内存访问时，请求地址中必须包含相同的标记或钥匙：

- 如果锁和钥匙匹配，则允许内存访问。
- 如果锁和钥匙不匹配，则发生内存访问违规。内存中锁标记和地址中钥匙标记不匹配会导致标记检查故障，引发错误。

下图显示了 MTE 对几个内存访问示例的处理方式：

图 3-1：MTE 锁和钥匙模型



本图中显示的内存访问过程如下：

- `0x07...9010`

  钥匙 `0x07` 与锁 `0x07` 匹配。内存访问成功。

- `0x04...8028`

  钥匙 `0x04` 与锁 `0x04` 匹配。内存访问成功。

- `0x03...8028`

  钥匙 `0x03` 与锁 `0x04` 不匹配。标记检查故障。

- `0x06...8010`

  钥匙 `0x06` 与锁 `0x04` 不匹配。标记检查故障。

标记检查机制有助于检测出不经常出现的、短暂的或难以检测的内存安全错误。可以通过配置标记检查故障来引发同步异常或异步异常报告。

由于标记必须从内存系统中获取，并存储到内存系统中，MTE 会导致一些性能开销。这种开销与内存分配的大小和生命周期，以及标记和数据是同时处理还是分别处理有关。

有几种方法可以尽量减少这种开销：

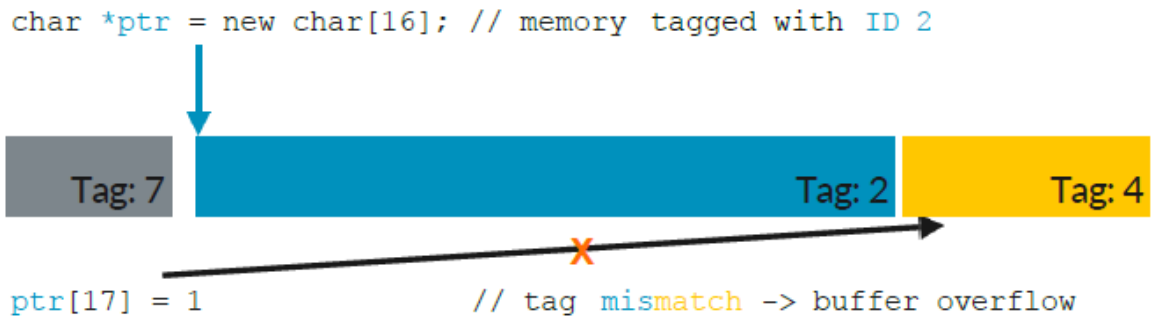
- 在实现分配器的同时，写入标记，并初始化内存。
- 避免过度分配永远没有数据写入的地址空间。
- 避免过度释放和重新分配。
- 避免在堆栈上进行大型大小固定的分配。

有关尽量减少 MTE 开销的更多信息，请参阅《Armv8.5-A 内存标记扩展白皮书》。

3.1 示例：缓冲区溢出

以下示例演示了 MTE 如何检测缓冲区溢出错误：

图 3-2：MTE 检测缓冲区溢出错误



过程如下：

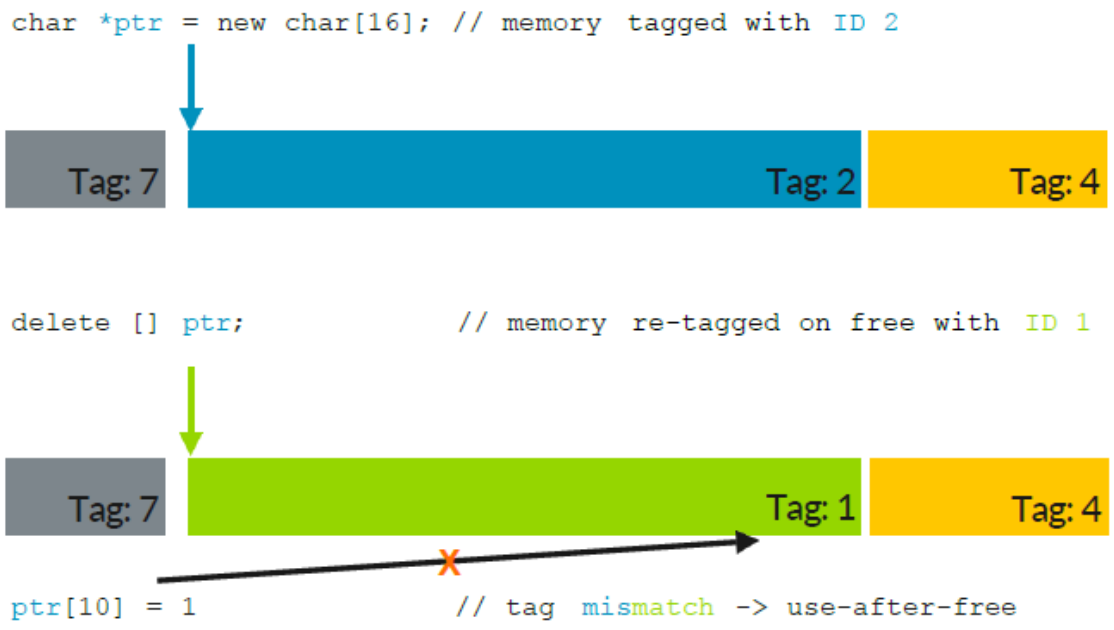
1. 创建新的数组指针 `ptr`，为数组分配一段内存区域。把该内存区域标记为 ID 2，而两侧的内存区域分别标记为 7 和 4。
2. 使用数组指针 `ptr` 访问内存时，指针包括与该内存区域相关联的标记：2。
3. MTE 检查内存访问中使用的标记是否与内存区域的标记相匹配。

   在这个例子中，内存访问溢出：试图在一个 16 元素数组中访问第 17 个元素。带有标记 2 的内存访问试图访问带有标记 4 的内存区域，因而不匹配。

3.2 示例：释放后重用

以下示例演示了 MTE 如何检测释放后重用错误：

图 3-3：MTE 检测释放后重用错误



过程如下：

1. 创建新的数组指针 `ptr`，为数组分配一段内存区域。把该内存区域标记为 ID 2，而两侧的内存区域分别标记为 7 和 4。
2. 删除数组指针 `ptr` 时，该内存区域被释放，并重新标记为 ID 1。
3. 使用数组指针 `ptr` 访问内存时，指针包括与已删除指针相关联的标记：2。
4. MTE 检查内存访问中使用的标记是否与内存区域的标记相匹配。

   在这个例子中，内存访问发生在内存被释放后。带有标记 2 的内存访问试图访问带有标记 1 的内存区域，因而不匹配。

3.3 MTE 模式

MTE 有以下几种模式：

- 同步模式 (SYNC)

  在同步模式下，地址中的标记与内存中的标记不匹配会导致同步异常。这种模式可以识别导致故障的精确指令和地址，不过会牺牲部分性能。

  相比性能，SYNC 模式更注重错误检测的准确性。在开发过程中或集成到连续集成系统中时，采用 SYNC 模式更有益。在这些情况下，精确的错误检测能力比性能开销更为重要。

- 异步模式 (ASYNC)

  在异步模式下，当标记不匹配时，处理器会继续执行指令，直到到达下一个内核入口，例如系统调用或计时器中断。此时，处理器使用代码 SEGV_MTEAERR 通过 SIGSEGV 终止进程。处理器不记录故障地址或内存访问。相比 SYNC 模式，ASYNC 模式对性能的影响更小。

  相比错误报告的准确性，ASYNC 模式更注重优化性能，报告中提供的错误发生的位置信息没有 SYNC 模式精确。ASYNC 模式是针对内存安全错误的低开销检测机制，对于更注重性能而非详细错误信息的生产系统十分有用。

- 非对称模式 (ASYMM)

  新版设备支持结合了 SYNC 模式和 ASYNC 模式优点的非对称 (ASYMM) 模式。在非对称模式下，读取内存访问在 SYNC 模式下处理，写入内存访问在 ASYNC 模式下处理。非对称模式对性能的影响一般非常接近 ASYNC 模式。如要确认您的设备是否支持 ASYMM 模式，请查看 `/proc/cpuinfo` 中是否存在字符串 `mte mte3`。

  Android 操作系统不允许程序员选择使用 ASYMM 模式。在兼容设备上，ASYNC 模式被重新定义为 ASYMM。此举是想告诉程序员，ASYNC 模式可能会生成同步故障，即带有代码 SEGV_MTESERR 的 SIGSEGV，而不是异步故障。

各个模式的主要区别在于对性能的影响和错误检测准确性。选择模式就是在性能和信息准确性之间权衡。

3.4 何时使用 SYNC 模式和 ASYNC 模式

如果仅在开发和测试期间使用 MTE，那么使用场景可能会受限。也就是说，有些错误可能无法发现。所以，应该考虑在生产中使用 ASYNC 模式来处理最关键的进程。

根据测试的工作负载和基准测试评估，ASYNC 模式的性能开销约为 1%~2%。也就是说，即使在生产系统中，ASYNC 模式通常也是可接受的。在经过充分测试的代码库中，如果已知内存安全错误密度较低，建议在生产中使用 ASYNC 模式。ASYNC 模式还可以防御以前未知的、很少发生的错误和零日攻击。

在开发和测试阶段建议使用 SYNC 模式，来帮助发现和修复内存错误。当目标进程表示易受攻击的攻击面时，SYNC 模式也可以在生产系统中发挥作用。例如，在安全性比运行时性能更重要的关键系统进程中。

此外，系统可以在 ASYNC 模式下运行，检测到错误时，使用运行时 API 将执行切换到 SYNC 模式，获取准确的错误报告。请参阅 MTE 错误报告，了解更多信息。

4. 在 Android 项目中启用 MTE

MTE 默认处于禁用状态。有几种不同的方式为系统进程和应用程序启用 MTE。启用 MTE 是进程属性：启用 MTE 适用于进程中的所有本机堆分配。

4.1 使用 Android 构建系统启用 MTE

作为进程属性，MTE 由主可执行文件的构建时设置控制。可通过以下选项对单个可执行文件或源树中整个子目录的此项设置进行更改。对于库和既不是可执行文件也不是测试的任何目标，此项设置将被忽略。

1. 要在特定项目的 Android 蓝图文件 `Android.bp` 中启用 MTE，请使用以下设置：

   启用 ASYNC 模式：

```
sanitize: {
    memtag_heap: true,
}
```

   启用 SYNC 模式：

```
sanitize: {
    memtag_heap: true,
   diag: {
       memtag_heap: true,
   },
}
```

   在特定项目的 `Android.mk` 文件中启用 MTE：

MTE 模式	设置
ASYNC	`LOCAL_SANITIZE := memtag_heap`

SYNC	`LOCAL_SANITIZE := memtag_heap`
	
	`LOCAL_SANITIZE_DIAG := memtag_heap`

2. 要在源树的子目录上启用 MTE，请使用以下设置。

   使用 PRODUCT_MEMTAG_* 变量在源树的子目录上启用 MTE：

MTE 模式	设置
ASYNC	`PRODUCT_MEMTAG_HEAP_ASYNC_INCLUDE_PATHS`

MTE 模式	设置
SYNC	`PRODUCT_MEMTAG_HEAP_SYNC_INCLUDE_PATHS`

使用 MEMTAG_HEAP_* 变量在源树的子目录上启用 MTE：

MTE 模式	设置
ASYNC	`MEMTAG_HEAP_ASYNC_INCLUDE_PATHS`
SYNC	`MEMTAG_HEAP_SYNC_INCLUDE_PATHS`

通过指定可执行文件的排除路径，在源树的子目录上启用 MTE：

MTE 模式	设置

ASYNC	`PRODUCT_MEMTAG_HEAP_EXCLUDE_PATHS`
	或
	`MEMTAG_HEAP_EXCLUDE_PATHS`

SYNC	`PRODUCT_MEMTAG_HEAP_EXCLUDE_PATHS`
	或
	`MEMTAG_HEAP_EXCLUDE_PATHS`

4.2 使用系统属性启用 MTE

要覆盖“使用 Android 构建系统启用 MTE”章节所列构建设置，请设置以下系统属性：

```
arm64.memtag.process.<basename> = (off|sync|async)
```

其中 `basename` 表示可执行文件的基本名称。

例如，如果可执行文件是 `/system/bin/myapp` 或 `/data/local/tmp/myapp`，则名称为 `arm64.memtag.process.myapp`。

此属性仅在进程启动时读取一次。Arm 建议使用此机制进行快速原型设计，以及尝试不同的 MTE 模式。此属性不适用于 Java 应用程序。Java 应用程序的 MTE 设置基于 `AndroidManifest.xml` 配置。关于如何为 Java 应用程序启用 MTE，请参阅“为 Android 清单内应用程序启用 MTE”一节。该章节介绍如何使用兼容性框架 `compat` 功能，通过开发者选项或 ADB 命令，更改设置。

4.3 使用环境变量启用 MTE

通过定义下列环境变量，可以指定 MTE 模式：

```
MEMTAG_OPTIONS=(off|sync|async)
```

如果 `MEMTAG_OPTIONS` 环境变量和 `arm64.memtag.process.<basename>` 系统属性都被定义了，则环境变量优先。

命令行应用程序仅支持使用 `MEMTAG_OPTIONS` 环境变量来指定 MTE 模式。

4.4 为 Android 清单内应用程序启用 MTE

要对应用程序进行配置，从而启用 MTE，请在 `AndroidManifest.xml` 中的 `<application>` 或 `<process>` 标记下设置 `android:memtagMode` 属性。如果 `android:memtagMode` 属性未被指定，则 MTE 已禁用。

```
android:memtagMode=(off|default|sync|async)
```

例如：

```
    <application
        android:allowBackup="true"
        android:dataExtractionRules="@xml/data_extraction_rules"
        android:fullBackupContent="@xml/backup_rules"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/Theme.MTE_test"
        tools:targetApi="31"
        android:memtagMode="sync"
        tools:ignore="MissingPrefix">
    </application>
```

在 `<application>` 标记上设置时，该属性会影响该应用程序使用的所有进程。可以通过设置 `<process>` 标记来覆盖单个进程的属性。

实验时，可以使用兼容性更改来设置 `memtagMode` 属性的默认值，用于未在清单中指定值或指定 `default` 的应用程序。这些兼容性更改可在 **“系统** > **高级** > **开发者选项** > **应用兼容性变更”** 中的全局设置菜单下找到。设置 `NATIVE_MEMTAG_ASYNC` 或 `NATIVE_MEMTAG_SYNC`，为特定应用程

序启用 MTE。也可以使用下列 `am compat` 指令：

```
$ adb shell am compat enable NATIVE_MEMTAG_[A]SYNC my.app.name
```

5. MTE 错误报告

MTE 检测到内存错误时，应用程序会退出。应用程序退出后，错误报告便可访问。错误报告包含设备日志、堆栈跟踪和其他诊断信息，帮助您找到并修复应用程序中的内存错误。

错误报告可以通过以下几种方式获取：

- 使用 Android 设备上的**开发者选项**。
- 使用开发机器上的 Android 调试桥 (`adb`)。

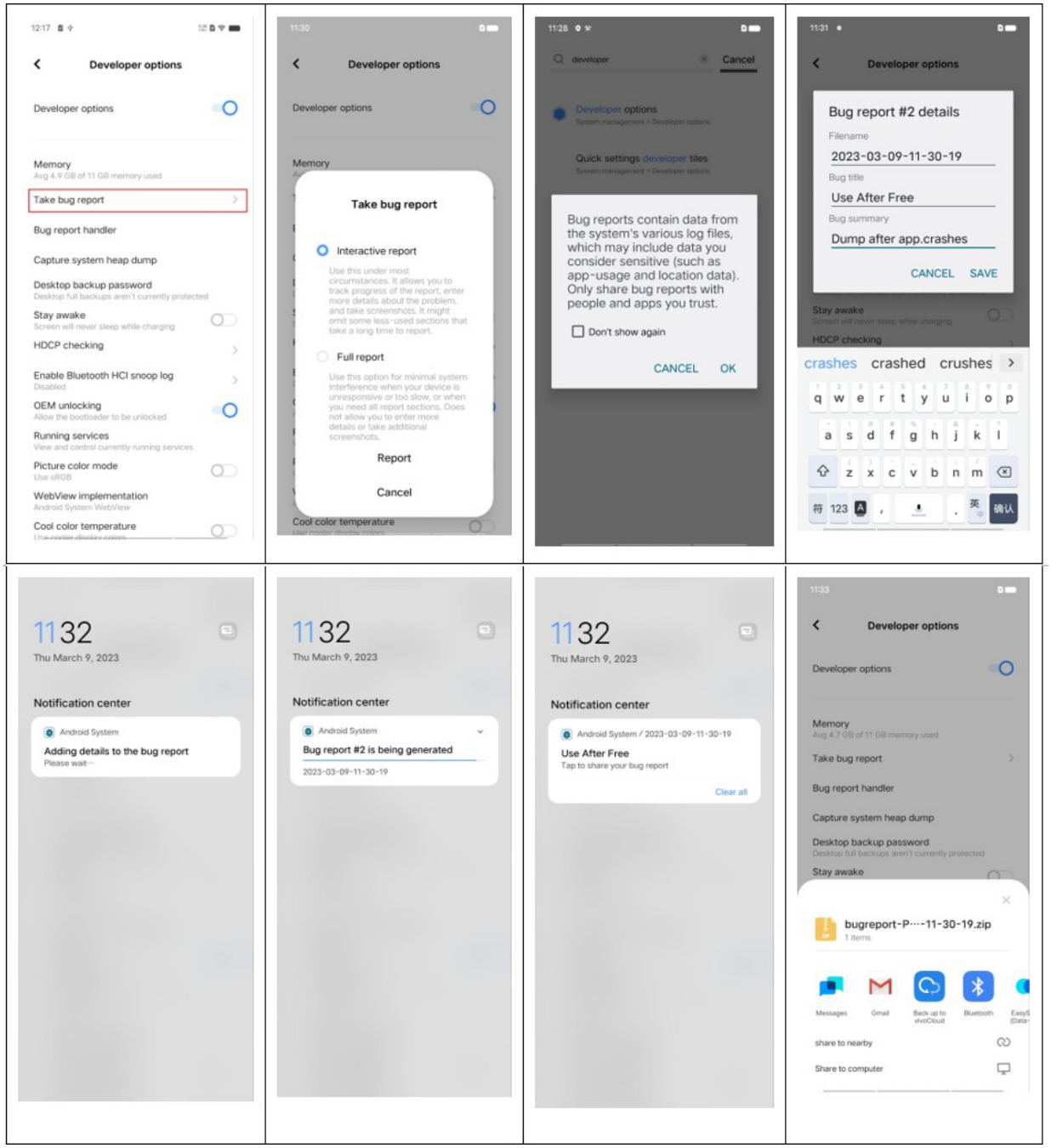
5.1 使用 Android 设备上的开发者选项获取错误报告

要直接从 Android 设备获取错误报告，请执行以下操作：

1. 在 Android 设备上启用**开发者选项**。有关如何执行此操作的更多信息，请参阅《Android 操作系统文档：配置设备上的开发者选项》。
2. 在**开发者选项**中，点击**获取错误报告**。
3. 选择想获取的错误报告类型，点击**报告**。
4. 一小段时间后，会出现一个通知，告诉您错误报告已生成。
5. 点击通知可分享错误报告。

以下图片显示了在运行启用了 MTE 的 Android 设备中获取错误报告过程的屏幕截图。设备不同，此过程的确切操作顺序也会不同。

图 5-1：在设备上访问错误报告



5.2 使用开发机器上的 adb 获取错误报告

要使用开发机器上的 Android 调试桥 (adb) 获取错误报告，请执行以下操作：

1. 从控制台运行以下 adb 指令：

   ```
   adb bugreport
   ```

   错误报告将以 zip 文件的形式保存在控制台提示路径下，文件名包含构建 ID 和日期，例如：

   ```
   bugreport-PD2183-TP1A.220624.014-2023-03-09-11-48-36.zip
   ```

2. 解压缩错误报告 zip 文件。

   zip 文件包含多个文件。错误报告是与文件夹同名的文本文件：

   ```
   bugreport-BUILD_ID-DATE.txt
   ```

5.3 解读错误报告

错误报告包含系统服务的诊断输出 (dumpsys)、错误日志 (dumpstate) 和系统消息日志 (logcat)。系统消息包括设备抛出错误时的堆栈跟踪，以及所有应用程序使用 Log 类写入的消息。

在 SYNC 模式下运行时，Android 分配器记录所有分配和解除分配的堆栈跟踪，并据此生成错误报告。

错误报告包括每个内存错误的解释，例如释放后重用或缓冲区溢出，还包括相关内存事件的堆栈跟踪。这些报告包含的上下文信息更多，有助于跟踪和修复错误。遇到标记不匹配时，处理器立即中止执行，并使用代码 SEGV_MTESERR 通过 SIGSEGV 终止进程，同时记录有关内存访问和故障地址的完整信息。此外，崩溃报告还显示进程 ID、线程 ID 和崩溃原因。

要在错误报告中找到此项信息，搜索 SEGV_MTESERR。此搜索可定位到分段故障的初始块，找到 CPU 寄存器 x0-x29 在收到 SIGSEGV 时记录的内容。

以下是可在错误报告中找到的信息示例：

```
Softversion: PD2183C_A_13.1.6.4.W10.V000L1
Time: 2023-03-09 13:56:30
```

```
*** *** *** *** *** *** *** *** *** *** *** *** *** *** *** ***
Build fingerprint: 'vivo/PD2183/PD2183:13/TP1A.220624.014/compiler03081056:user/
release-keys'
Revision: '0'
ABI: 'arm64'
Timestamp: 2023-03-09 13:56:30.251141624+0000
Process uptime: 6s
Cmdline: com.example.mte_test
pid: 9477, tid: 9477, name: xample.mte_test >>> com.example.mte_test <<<
uid: 10237
tagged_addr_ctrl: 000000000007fff3 (PR_TAGGED_ADDR_ENABLE, PR_MTE_TCF_SYNC,
 mask 0xfffe)
pac_enabled_keys: 000000000000000f (PR_PAC_APIAKEY, PR_PAC_APIBKEY, PR_PAC_APDAKEY,
 PR_PAC_APDBKEY)
signal 11 (SIGSEGV), code 9 (SEGV_MTESERR), fault addr 0x08000072b1c2a500
x0 08000072b1c2a4f0 x1 0000007ff60668b0 x2 fffffffffffffff0 x3
 0000007ff6066ac0
x4 0000007ff6066b30   x5  0000000000000004  x6 0000000000000000 x7
 000000729a5e9c02
x8 08000072b1c2a4f0   x9  000000000000002a  x10 0000000000010000 x11
 0000000000000001
x12 00000000b1c2a500  x13 00000075d317c518  x14  ffffffffffffffff  x15
 00000000ebad6a89
x16 00000075d2f13dc0  x17 00000075d2e9cc60  x18  00000075d79a4000  x19
 0200007411a05380
x20 0000000000000000  x21 0000000000000000  x22  0000007285f7c322  x23
 000000000000106e
x24 000000729f800880  x25 0000007ff6066dd0  x26 0000000010380011 x27
 0000000000000004
x28 0000007ff6066cd0  x29 0000007ff6066cb0
lr 000000728644f428   sp 0000007ff6066c80   pc 000000728644f434   pst
 0000000060001000
```

在此示例中，可以看到以下行：

```
signal 11 (SIGSEGV), code 9 (SEGV_MTESERR), fault addr 0x08000072b1c2a500
```

此行的意思是，收到了一个 SIGABRT 信号，附带代码 SEGV_MTESERR，由访问内存地址 0x08000072b1c2a500 造成。

此数据块后面有一个回溯，显示了崩溃时代码的位置，例如：

```
backtrace:
#00 pc 000000000000d384 /data/app/~~8QwzBMNBT6U1-xi0qH9OdQ==/com.example.mte_test-
AE8qJx9ASOlRNa8HmD3iJA==/lib/arm64/libmte_test.so
#01 pc 0000000000021a354 /apex/com.android.art/lib64/libart.so
 (art_quick_generic_jni_trampoline+148) (BuildId: 2d8a73ff5c99d5a227b31111c86db3e6)
#02 pc 000000000020a2b0 /apex/com.android.art/lib64/libart.so (nterp_helper+4016)
 (BuildId: 2d8a73ff5c99d5a227b31111c86db3e6)
#03 pc 00000000006fa40a /data/app/~~8QwzBMNBT6U1-xi0qH9OdQ==/com.example.mte_test-
AE8qJx9ASOlRNa8HmD3iJA==/oat/arm64/base.vdex
And after a number of lines, we see a message pointing directly to the cause of the
 crash: use-after-free and the lines produced by the unwinder.
Note: multiple potential causes for this crash were detected, listing them in
 decreasing order of likelihood.
Cause: [MTE]: Use After Free, 42 bytes into a 128-byte allocation at 0x782d277bf0
deallocated by thread 10124:
      #00 pc 00000000000493a8 /apex/com.android.runtime/lib64/bionic/libc.so
 (scudo::Allocator<scudo::AndroidConfig,
      #01 pc 00000000000442b4 /apex/com.android.runtime/
lib64/bionic/libc.so (scudo::Allocator<scudo::AndroidConfig,
 &(scudo malloc postinit)>::deallocate(void*,
```

```
      #02 pc 000000000000d340 /data/app/~~8QwzBMNBT6U1-xi0qH9OdQ==/
com.example.mte_test-AE8qJx9ASOlRNa8HmD3iJA==/lib/arm64/libmte_test.so
      #03 pc 000000000021a354 /apex/com.android.art/lib64/libart.so
 (art_quick_generic_jni_trampoline+148) (BuildId: 2d8a73ff5c99d5a227b31111c86db3e6)
```

回溯以列的形式显示信息，如下：

1. 帧号。
2. PC 值。PC 值对应的是共享库的位置，不是绝对地址。
3. 映射区的名称。此名称通常是共享库或可执行文件，但也可能是 JIT 编译的代码。
4. 如果有符号可用，展开器会显示 PC 值对应的符号，以及该符号的偏移量（以字节数为单位）。

几行后有一条消息，直接指出崩溃的原因：释放后重用，同时还有展开器生成的几行信息：

```
Note: multiple potential causes for this crash were detected, listing them in
 decreasing order of likelihood.
Cause: [MTE]: Use After Free, 42 bytes into a 128-byte allocation at 0x782d277bf0
deallocated by thread 10124:
      #00 pc 00000000000493a8 /apex/com.android.runtime/lib64/bionic/libc.so
 (scudo::Allocator<scudo::AndroidConfig,
      #01 pc 00000000000442b4 /apex/com.android.runtime/
lib64/bionic/libc.so (scudo::Allocator<scudo::AndroidConfig,
 &(scudo_malloc_postinit)>::deallocate(void*,
      #02 pc 000000000000d340 /data/app/~~8QwzBMNBT6U1-xi0qH9OdQ==/
com.example.mte_test-AE8qJx9ASOlRNa8HmD3iJA==/lib/arm64/libmte_test.so
      #03 pc 000000000021a354 /apex/com.android.art/lib64/libart.so
 (art_quick_generic_jni_trampoline+148) (BuildId: 2d8a73ff5c99d5a227b31111c86db3e6)
```

有关如何解读错误报告的更多信息，请参阅以下 Android 文档：

- Android 操作系统文档：诊断本机崩溃
- Android 操作系统文档：调试本机 Android 平台代码

5.4 Tombstone 文件

应用程序崩溃时，会有一个基本的崩溃转储写入 Android Studio 的 **Logcat** 窗口。更详细的信息会被写入 `/data/tombstones/` 目录下的一个 tombstone 文件。这个 tombstone 文件包含崩溃进程的详细数据，包括：

- 崩溃进程中所有线程（包括捕获信号的线程）的堆栈跟踪
- 完整的内存映射
- 所有打开文件描述符的列表

崩溃事件发生时，Android Studio 中的 **Logcat** 窗口会显示 tombstone 文件的位置信息。例如：

```
Tombstone written to: /data/tombstones/tombstone_07
```

要检索 tombstone 文件，请使用 `adb bugreport` 获取错误报告，具体方法参照“使用 Android 设备上的开发者选项获取错误报告”。解压缩错误报告文件后，可在 `/FS/data/tombstones/` 文件夹下找到 tombstone 文件。

有关 tombstone 文件的更多信息，请参阅《Android 操作系统文档：崩溃转储和 tombstone 文件》。

6. 利用 Android Studio 和 MTE 进行调试

在 Android Studio 中启用 MTE，可以调试应用程序中的内存错误。MTE 检测到内存错误，令应用程序退出时，Android Studio 会指示导致错误的代码行。

要在启用了 MTE 的情况下使用 Android Studio，请执行以下操作：

1. 为 Android 项目启用 MTE。请参阅“在 Android 项目中启用 MTE”一节，了解几种不同的启用方法。
2. 在 Android 设备上启用**开发者选项**。有关如何执行此操作的更多信息，请参阅《Android 操作系统文档：配置设备上的开发者选项》。
3. 根据设备连接 Android Studio 的方式，启用 USB 调试或无线调试。请参阅《Android 操作系统文档：在设备上启用 USB 调试》或《Android 操作系统文档：通过 Wi-Fi 连接设备》，了解更多信息。
4. 连接设备，等待工具栏显示设备。
5. 单击**调试**按钮，启动应用程序。

   应用程序遇到内存错误时，MTE 会令应用程序退出。Android Studio 调试器指示导致内存违规的代码行，如下图所示：

图 6-1：Android Studio 调试器中 MTE 检测到的内存错误

如要测试此功能，可以编写一段能实现内存错误的简单代码，并将代码链接到一个按钮。例如：

```
extern "C"
JNIEXPORT void JNICALL
Java_com_example_mte_1test_MainActivity_useAfterFreeC(JNIEnv *env, jobject thiz) {
    // TODO: implement useAfterFreeC()
    char * volatile p = new char[10];
    delete[] p;
    p[5] = 42; //Trying to access an array element that no longer exists!
}
```

单击执行此代码的按钮时，应用程序会退出，调试器会停止在以下代码行：

```
delete[] p;
```

有关在 MTE 设备上使用 Android Studio 调试应用程序的更多信息，请观看《内存标记扩展简介》视频。

7. 将 MTE 集成在内存管理系统中

Unity 创建了自己的内存管理系统，来优化内存利用率，尽可能地减少内存处理对性能的影响。本节介绍了 Unity 在其内存分配器中实现 MTE 的经验。

Unity 内存管理系统含有一个集中式内存管理器，用于所有用户空间分配。内存管理器使用多个基础类型的分配器，在分配内存时，根据类别选择使用。这些分配器在运行时初始化过程中被创建时，会定义一个低级别的备用分配器。低级别的备用分配器通常由分页虚拟内存提供支持，但是类型不限，比如可以是 C 标准库，也可以是 malloc 调用。

7.1 将内存标记添加到现有的内存管理代码中

Unity 使用的方法分为以下两个部分：

1. MTE 实用程序实现。此实现是实用程序函数的命名空间，是一种执行内存标记和相关功能的低级实用程序实现，可以在内存分配的范围之外使用，例如，开发者想要在特定场景下检查特定代码片段时就可以使用。
2. MTE 分配器包装器。这个包装器被内存管理器用作顶级分配器，继承现有的基础分配器，并在相应的基本方法调用前后，使用 MTE 实用程序实现执行与内存标记相关的代码。因此，这个包装器继承的每个分配器都不会感知到内存标记，也不需要针对现有的分配器编写特定代码。

7.1.1 MTE 实用程序实现

MTE 实用程序实现是命名空间内的一组静态方法，执行以下函数：

- 系统初始化
  - 初始设置，在启动时调用一次
- 标记指针
  - 带有和不带有标记位筛选
  - 从指针中提取标记 ID，有助于重新标记指针
- 内存地址
  - 检查输入地址是标记地址还是实际地址
  - 检查标记粒度

- 标记
  - 标记内存和取消标记内存
- 内存拷贝
  - 当对齐关闭时，将数据拷贝到标记内存，以及从标记内存中拷贝数据。

所有实用程序方法都是低级别的静态函数，没有内存分配器的感知功能。

每个平台都可以有条件地包含一个特定的实现。如果 MTE 被禁用，除了内存拷贝函数会回退到标准内存拷贝函数之外，所有函数都将保持无操作状态。

如果特定的内存区域可以被标记和测试，开发者可以使用实用程序函数临时执行测试。不过，MTE 实用程序实现主要是为分配器包装器服务，这样，包装器就无需感知特定的 MTE 实现。

7.1.2 MTE 分配器包装器类

在内存管理器进行运行时初始化时，分配器会检查此操作是否符合 MTE 的要求，具体是检查是否使用虚拟分页内存来进行备份。

如果启用了 MTE 目标构建并且符合要求，则使用分配器包装器类对分配器进行包装。这是干预性最小的实现，因为除了少数小规模的异常之外，这种实现不需要修改高（管理器）代码或低（分配器）代码。

包装器还检查特定分配是否满足标记内存的要求。也就是说，分配必须对齐，并且具有实用程序类所述的大小粒度。

大小未对齐时，则发生异常。但是，基础分配器经过设计，可为了符合要求过度分配，因此可以标记内存。

7.2 函数测试

分配器包装器的测试内容与其他分配器一样，但需要额外进行粒度测试。函数测试使用虚假的分配器装置进行，从而返回特定结果，并确保分配器的行为符合预期。

实用程序实现属于极低级函数，执行基本测试，确保其实现按预期工作。此外，还实现了使用信号处理程序捕获 `SEGV_MTE*ERR` 错误的测试，确保系统处于活动状态且正常工作。

7.3 性能测试

实用程序实现包括以下性能测试：

- 标记内存区域和取消标记内存区域，包括小的 64 字节块和较大的 4k 块。这项测试会检查除了实际标记循环之外是否存在特定开销，是对 `STG` 和 `ST2G` 指令的测试。
- 读取地址信息。测试 `LDG` 指令。
- 读写标记内存。之后将结果与同设备上非 MTE 构建进行比较。

7.4 实现 MTE 时的注意事项和技巧

本节面向实现 MTE 的人员提供了建议。

7.4.1 过度分配和未标记分配

如上述 “MTE 分配器包装器类” 小节所述，Unity 不会标记不符合 MTE 标记要求的分配。也就是说，如果基础分配器的分配粒度小于 MTE 粒度，则会存在未标记的分配。那么，除了基础分配器的分配之外，还会执行过度分配，MTE 标记的内存大小会匹配实际分配的内存，而不是请求的内存大小。

另一种处理方法是增加内存大小，满足粒度大小要求。但是，这种方法会改变没有标记的原始代码的内存占用空间和布局。这两种方案各有优缺点。Unity 决定，至少最初采用第一种方法，并尽可能维持原始内存占用空间和布局。

7.4.2 应将 MTE 视为现有工具的补充而非替代

如前一节所述，有些情况下 MTE 并不能发现故障。

由于大小限制，MTE 可能不会标记特定分配。如果选择第二种方法，那么内存占用空间和布局会掩盖这个问题。

但是，MTE 本身非常强大。其主要优点是能够在实际故障发生的指令处停止，而不是在发生释放内存等情况后进行事后验证。

7.4.3 尽可能使用 ST2G

使用 `ST2G` 后，测得性能提高 2.5 倍。


Note

`ST2G` 的粒度虽然是 32 字节，但只需对齐 16 字节即可，因此会比预期更方便使用。

7.4.4 关于 PROT_MTE

确保使用 `PROT_MTE` 保护虚拟页面，确保虚拟页面在系统内受到预期保护。请查看针对 mprotect/advice 实施保护的文档，了解禁用保护的条件。这类条件可能看起来显而易见，但是有些漏洞，比如与 madvice 有关的漏洞，很容易漏检。

7.5 其他 MTE 实现方法

在实现 MTE 时，还考虑了以下方法，但这些方法与所选方法相比存在不足。

7.5.1 内存管理器实现，高级别

这种方法对现有系统的代码干预性最小，乍看是最好的选择。当前的分配器不需要感知内存标记，实际的 MTE 代码会在调用当前分配器前后执行。

但是，由于内存管理器本身的功能相当复杂，而且没有区分系统标记内存和用户空间内存的好方法，这种方法很快就变得复杂。虽然这种方法可行，不过，我们也开始寻找其他替代方案。

7.5.2 按分配器类型实现，低级别

由于每个分配器处理内存的方式不同，所以这种方法对性能的影响最小，也因此成为可行方案。优化的示例有：

- 在重新分配时进行部分标记
- 在某些情况下忽略对解除分配取消标记的情况

但是，这种方法的干预性最强，而且每个分配器都必须针对大部分方法实现 MTE。在许多情况下，需要修改测试。由于新的分配器需要修改代码，这种方法也会带来大量维护成本。

8. 相关信息

以下是与本指南相关的资源：

- 通过内存标记扩展增强安全性
- 内存标记扩展白皮书
- Android 操作系统文档：Arm 内存标记扩展
- Android 操作系统文档：诊断本机崩溃
- Android 操作系统文档：调试本机 Android 平台代码