**内存检测方法（应用程序）**

|  |  |  |  |
| --- | --- | --- | --- |
| 版本号 | 作者 | 历史 | 日期 |
| V1.0 |  |  |  |
| 备注 |  | | |

**目录**

[**概要** 3](#_Toc334170594)

[**1.** **内存检测方法的目的** 3](#_Toc334170595)

[**1.1.** **内存泄露和内存写溢出的起因与影响** 3](#_Toc334170596)

[**2.** **内存检测方法的实现** 3](#_Toc334170597)

[**2.1.** **常见的内存检测方法** 3](#_Toc334170602)

[**2.2.** **内存检测方法的实现** 4](#_Toc334170603)

[**3.** **内存检测方法的应用** 6](#_Toc334170604)

[**3.1.** **应用介绍** 6](#_Toc334170605)

[**3.2.** **注意事项** 8](#_Toc334170606)

**概要**

在C/C++编程中内存错误一直是一个非常普遍和令人头痛的问题，主要是因为C/C++编程中内存的申请和释放完全是由程序员自主控制，稍不注意，就会在系统中导入内存错误，内存错误的导入往往非常严重，一般会带来诸如系统崩溃、内存耗尽这样严重的后果，而且与许多其他类型的常见错误不同，内存错误通常具有隐藏性，即它们很难再现，症状通常不能在相应的源代码中找到，无形中加大了排查难度。本文将主要介绍一种有效的内存检测方法，用以在短时间内定位和修复内存错误。

1. **内存检测方法的目的**

由于内存错误存在着潜伏期长，在较短的时间内无法进行复现，而且出现的症状通常不能在相应的源代码中找到，造成了问题排查的难度，因此有必要寻求一种快速、有效的方法进行内存错误的检测。在研究内存检测方法之前，先了解下一般内存错误的种类以及产生的机制。

内存错误一般包括有内存泄露、内存溢出、分配和释放函数不匹配、野指针、内存申请失败、内存未初始化、指针空挂和内存重复释放等等，本文将主要介绍两种常见的内存错误的检测，分别为内存泄露和内存写溢出。由于这两种内存错误一般需要较长时间才能复现，而且具有偶然性，排查这两种内存错误的难度要略高于其他内存错误，因此本文将主要讲解这两种内存错误的检测方法。在介绍其内存检测方法之前，先来了解下两种内存错误的起因和造成的影响。

* 1. **内存泄露和内存写溢出的起因与影响**

1. **内存泄露**

一般我们常说的内存泄露是指堆内存的泄露。堆内存是指程序从堆中分配的，大小任意的，使用完后需要程序员主动释放分配的内存。应用程序一般使用malloc，calloc和recalloc等函数（C++中使用new操作符）从堆中分配到一块内存，使用完之后，程序必须负责相应的调用free或delete释放该内存块；否则，这块内存就不能被再次使用，我们称这块内存泄露了。最初少量内存泄露可能不引人注目，但随着时间的推移，内存泄露越来越多，就会出现一些征兆，例如性能下降等，在应用程序内存不足时发生崩溃。更为严重的是，占用了所有可用内存的泄露应用程序可能会导致其他应用程序崩溃，从而无法确定问题出在哪个应用程序。

1. **内存写溢出**

写内存时超过所分配的内存空间，就发生了内存写溢出错误。内存写溢出一般会造成所写数据的丢失，若超出的部分太长，将其他内存块的重要数据覆盖，这可能造成重要数据的永久丢失。内存溢出是目前50%的系统漏洞的起因，存在内存溢出的程序很容易遭到黑客攻击，后果不堪设想。

1. **内存检测方法的实现**

为了能够更快、更准确对内存错误进行定位，该节将主要介绍几种常见的内存检测方法以及Yealink产品目前所采用的内存检测方法的实现机制。

* 1. **常见的内存检测方法**

目前，常见的内存检测方法有以下几种：

1. **采用代码审查的方式进行控制。**

这种方法是最容易想到的，但是效果也是相当的有限的。当程序复杂度增加的时候，这种方法就越加显得无能为力了。

1. **采用一定的工具来帮助发现内存错误。**

比如来自IBM的Purify、开源的Valgrind，等等。这些工具在使用时都不需要我们去改变程序源代码，其使用大至也可以分为两类。第一类是需要对代码进行重新编译，这种方法通常使用起来相对的麻烦，为了方便使用，通常需要将工具与项目的编译环境进行整合。上面所提及的Purify就是采用这类方法的，一般不建议采用。另一类则不需要对代码进行重新编译，因此，使用起来相当的方便，上面所说到的开源项目Valgrind就是属于这一类的。使用这些工具可用于检测内存泄漏和内存写溢出，不过使用时应当注意两点：第一，我们要保证我们的代码在测试时有尽可能高的代码覆盖率。这是因为，内存泄漏的检测需要代码被执行到了，检测工具才能发现。现实情况是，我们往往很难做到百分之百的代码覆盖率，因此检测的效果也是有限的。第二，由于这些工具对于被测程序（代码）的性能有及大的影响，因此，如果这些工具的使用造成程序无法正常运行，那么这类工具是无法使用的。由于以上两点在测试时很有可能造成条件无法满足，因此，不能成为内存检测的终级解决方法。

1. **采用一定的封装技术对内存的分配与释放进行接管。**

通常提供一个模块或库对malloc ()、calloc()、realloc()、strup()、strndup()、free ()、new、delete和delete []进行很薄的一层封装，然后向应用程序提供相应的API用于分配和释放内存。此外，这个封装层还提供一定的方式让我们能实时的得到运行时内存的使用情况。比如，我们可以看一看此时有哪些文件的哪一行分配了内存且还没有释放，甚至可以统计到目前为止共内存分配与释放情况，还可知所剩内存多少，这种方式对于我们现实产品很有意义。这种方法由于内存管理模块是作为最终软件产品的一部分，因此，我们可以随时得知内存的使用情况，这与前面提到的第二种方法是完全不一样的。

对于上述三种方法，方法三应当是首选。前面两种方法，在使用上过度的依赖于人或工具，所以很不方便，且对程序运行时内存检测存在一定的局限性。当然，第三种检测方法也存在一定的局限性，其一，对于每一次内存分配需要记录在哪个文件以及文件的哪一行以用于在需要时显示这些信息，造成一定的内存开销；其二，由于增加了一层的封装，尽管很薄，但内存的分配速度还是会有一点点下降。从上面两点来看，第三种方法是通过时间和空间来换取实用性的。Yealink产品所采用的内存检测方法目前主要为第三种检测方法——采用一定的封装技术对内存的分配与释放进行接管。

* 1. **内存检测方法的实现**

该小节将主要介绍第三种内存检测方法——采用一定的封装技术对内存的分配与释放进行接管的实现机制。

采用一定的封装技术对内存的分配与释放进行接管的内存检测方法主要利用C语言的宏调用来替代原有的函数调用，比如我们在代码中调用了malloc()，实际是调用了\_dbg\_mc\_malloc()。该内存检测方法主要通过维护MEM\_CHECK\_NODE\_SIZE串内存检测链表来实现对内存泄露和内存写溢出的检测，链表结构如图2-1所示。这些链表主要由两个结构体构成，分别为mc\_node结构体和mc\_info结构体（如图2-2和图2-3），每个链表的头节点都为mc\_node，该结构体记录着内存函数调用的次数以及指向存储内存调用信息的mc\_info结构体。而mc\_info结构体则主要记录所申请内存块的首地址、字节数、内存块的申请位置以及内存块数据的校验数据信息。每次调用内存函数时，就会更新这些链表。有了这些链表，我们就可以在适当的位置打印一些重要的信息，包括malloc、calloc、recalloc、free等内存函数调用的次数，申请或分配的内存数以及调用它们的文件和位置等等，信息非常详细，有了这些信息，就能比较容易定位内存使用的错误源。



图2-1 内存检测链表

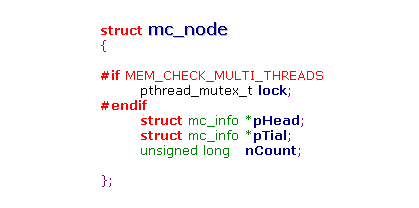


图2-2 mc\_node结构体

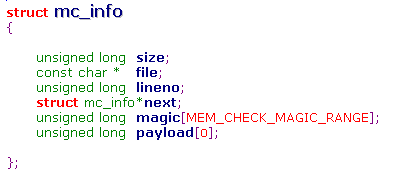


图2-3 mc\_info结构体

下面将主要分析该内存检测方法是如何对内存泄露和内存溢出这两种内存错误进行检测的。

1. **内存泄露的检测**

内存泄露只需通过查询这些链表即可，遍历这些记录内存申请信息的链表，如果链表的节点不为空，则链表中所指向的每个mc\_info节点都代表着一个泄露的内存块，实际上这些未必皆全是泄露的内存块，可能只是在初始化时分配的内存，在程序结束前才进行释放，因此在统计泄露内存块时，应先排除这部分内存块。

1. **内存溢出的检测**

当用户需要分配s字节的内存时，实际分配的内存是(s+2XMEM\_CHECK\_MAGIC\_SIZE)个字节，即在实际分配的s字节内存的前后分别分配MEM\_CHECK\_MAGIC\_SIZE个字节的额外内存，并在这两个额外内存内分别填入特殊数据MEM\_CHECK\_MAGIC\_HEAD和MEM\_CHECK\_MAGIC\_TIAL；然后检测是否出现越界访问时，可通过判断这块内存区域中额外内存的内容是否与MEM\_CHECK\_MAGIC\_HEAD或MEM\_CHECK\_MAGIC\_TIAL的值一致，如果不一致，则说明出现了越界访问。

1. **内存检测方法的应用**

第二节主要介绍内存检测的实现原理，该节将介绍在具体程序中应用该内存检测方法。

Yealink产品使用的内存检测方法为《采用一定的封装技术对内存的分配与释放进行接管》。该内存检测方法的实现文件有memcheck.c和memcheck.h，涉及C++语言时则还需添加memcheck\_cpp.cpp。这些文件的具体路径为【\\10.2.1.70\shared\文档仓库\知识仓库\技术规范\内存检测】。

在使用时只需将这些文件加入工程，在适当的地方调用相应的函数，则编译生成的可执行程序便可轻松实现对内存泄露和内存溢出等内存错误的检测。

* 1. **应用介绍**

Yealink产品使用的内存检测方法可实现检测功能有以下几种：

1. **统计各种内存函数调用次数**

调用memcheck\_info函数，便可打印出到目前为止，内存使用情况的统计信息，如malloc、calloc等内存函数调用的次数，具体信息如图3-1所示。从图3-1可看出，malloc在程序运行之后被调用了55次，calloc则被调用了61次，其他内存申请函数则未被调用。除了可查看内存申请次数，还可查询到内存前后共被释放了44次，因此可知目前系统还存在73块内存内释放。然而，该函数只是简单的打印一些统计信息，要查看具体泄露的源头还需通过调用其它函数查看。

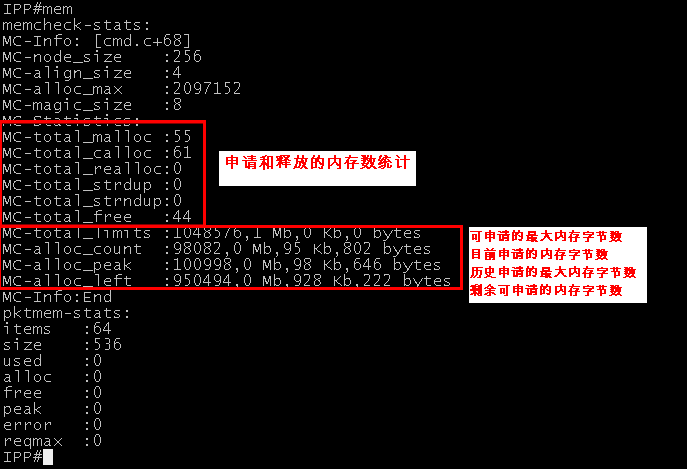


图3-1 内存使用统计信息

1. **查看未释放的内存块的调用信息**

调用memcheck\_dump函数，则可查看目前泄露的内存块的调用信息，例如在哪个文件中的哪一行被调用以及申请了多少字节的内存等信息，具体信息如图3-2所示。通过这些打印信息，我们就能很容易定位到泄露内存块的具体位置。

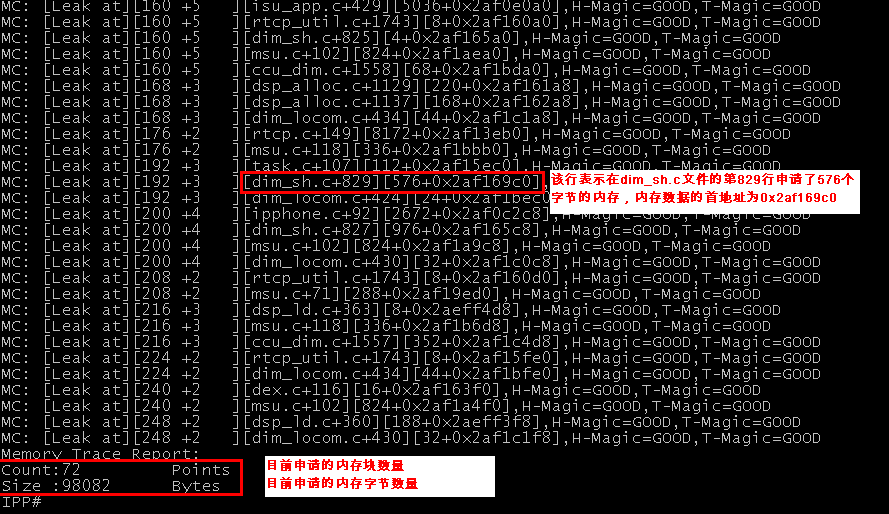


图3-2 未释放的内存块的调用信息

1. **统计目前申请的内存块和字节数**

由图3-1和3-2可知，memcheck\_info和memcheck\_dump函数除了统计各种内存函数调用次数和查看未释放的内存块的位置信息外，它们还统计目前申请未释放的内存块数量以及这些内存块所占字节数，方便我们了解内存的具体使用情况。

1. **检测内存溢出情况**

内存溢出的检测可通过调用memcheck\_check函数实现。通过图3-3和图3-4 内存块未溢出和溢出情况的对比可知，如果出现内存溢出情况，将提示Memory Corrupted错误，如果数据头溢出，则H-Magic=BAD；若数据尾溢出，则T-Magic=GOOD，可检测数据前后MEM\_CHECK\_MAGIC\_SIZE个字节溢出情况，超过此范围将检测失效。

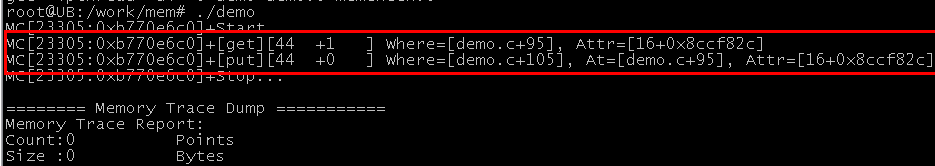


图3-3内存块未溢出情况

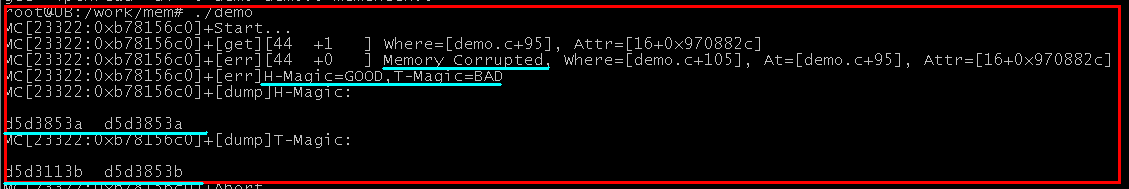


图3-4内存块溢出情况

1. **查看申请内存块内的数据**

调用memcheck\_dump函数还可查看申请内存块内的数据，具体信息如图3-5。查看数据的数量可由程序控制。

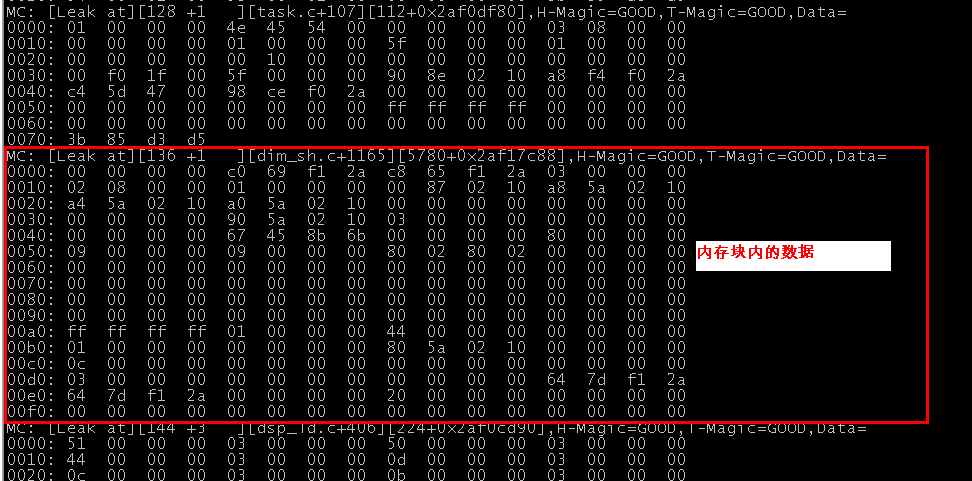


图3-5内存块数据

* 1. **注意事项**

1. **内存对齐问题**

由于内存对齐的缘故，因此每次实际申请的内存大小比预想的要大，一般为4的倍数，例如char \*p=malloc(17)，系统将会自动分配20个内存字节，这样的做法就会造成p[17],p[18],p[19]这三个字节内存空间检测不出是否存在内存写溢出情况，因此内存申请大小的取值一般为4的倍数；另一方面在检测超出数据前后MEM\_CHECK\_MAGIC\_SIZE个字节内存空间的溢出情况，也无法检测到。

1. **内存溢出检测时机**

在读写内存时，检测不出内存是否溢出。只有当主动调用函数查看或者释放内存时，才能检测出内存是否存在溢出情况。

1. **内存泄露误检测**

内存泄漏(memory leak)指由于疏忽或错误造成程序未能释放已经不再使用的内存的情况。而在本文的检测方法中，打印出的内存泄露信息是目前系统分配但仍未释放的内存，非直接打印出已不再使用的内存，因此在内存泄露检测时还需进行排除工作。