Valgrind Memcheck 源码分析

Linsoft1994 于 2018-03-15 01:11:30 发布

分类专栏： Linux 文章标签： Valgrind Memcheck VEX

本文深入分析了Valgrind Memcheck的源码，介绍了Valgrind作为动态分析工具的框架，以及Memcheck如何检测内存错误。内容包括Memcheck的主要功能，常见内存错误类型，工作原理如Valgrind的启动、中间表示（VEX）和JIT执行，以及实验部分展示的内存错误检测效果。

Valgrind Memcheck

日期：2017-12-27 18:10:53 星期三

一、源代码的基本情况

Valgrind版本

3.13.0

主要涉及文件

/memcheck/mc_main.c
/memcheck/mc_translate.c
/memcheck/mc_errors.c
/VEX/pub/libvex_ir.h

功能概述

Valgrind是用于构建程序动态分析工具的重量级插桩框架。目前Valgrind内置的工具，可用于检测内存管理和线程竞争等等过程中产生的问题，更快速、更准确和更详细地分析所需的程序。

其中，Memcheck是valgrind应用最广泛的工具。它是一个重量级的内存检查器，能够发现C或者C++在开发过程中绝大多数导致程序崩溃或者不可预知的行为的内存相关的错误，比如：使用未初始化的内存、使用已释放内存、内存访问越界等。

二、常见内存错误

Memcheck检查的是C和C++程序中通用的的一些内存错误。其中，文件/memcheck/mc_errors.c定义了这些内存错误的枚举类型，每个枚举项的解释见右边注释。

```
/* What kind of error it is. */
typedef
   enum {
      Err_Value, // 使用未初始化变量
      Err_Cond, // 使用未初始化值进行条件跳转
      Err_CoreMem, // 地址中包含未初始化字节
      Err_Addr, // 地址不可读或不可写
      Err_Jump, // 程序运行跳转至非法内存
      Err_RegParam, // Syscall寄存器参数包含未初始化字节
      Err_MemParam, // Syscall堆栈参数包含未初始化或不可寻址字节
      Err_User,
      Err_Free, // 释放了非法内存
      Err_FreeMismatch, // new/new[]/malloc的内存没用对应delete/delete[]/free释放
      Err_Overlap, // strcpy/memcpy等函数的src和dest地址有重叠
      Err_Leak, // 内存泄露
      Err_IllegalMempool, // 内存池地址非法
      Err_FishyValue, // 函数参数中带有非法值，如size_t类型传入了负数
   }
   MC_ErrorTag;
```

总结起来，有以下几种常见的内存错误：

使用未初始化内存

位于程序不同段的变量，其初始值是不同的。位于BSS段的全局变量和静态变量初始为0，而位于Stack的局部变量和位于Heap的动态分配的变量，其初始值为随机值。如果程序使用了为随机值的未初始化变量或者由未初始化变量派生出的变量，那程序行为无法预估。

内存读写越界

读写了不应该或者无读写权限访问的地址空间。比如数组读写越界、堆内存读写越界、经典的off-by-one问题。

内存覆盖

在使用标准库中提供的内存操作函数，如strcpy、 memcpy、 strcat等，src 和 dst 指向的地址存在重叠，导致最终操作不可预期。

动态内存管理错误

申请和释放不匹配

用malloc/alloc/realloc申请的内存需要用 free 释放；用new/new[]申请的内存需要用delete/delete[]释放。不匹配的申请和释放操作会带不可预期的问题。

重复释放(Double Free)

同一块动态分配的内存先后被释放多次，导致内存错误，也是一些安全问题的源头。

释放后使用(Use After Free)

动态分配的内存释放后，继续进行操作。这类操作可能不会导致程序崩溃，但会导致数据错乱，黑客也可以此篡改malloc等分配器的结构体用于泄露或者修改内存。

内存泄漏

在程序中动态申请的内存，在使用完后即时没有释放，丢失相关引用，也无法被程序的其他部分访问，直到程序结束前仍然占用着。不断地内存泄露，会导致最终程序使用内存过大，产生OOM。

三、Memcheck工作原理

Valgrind概述

Valgrind是用于构建程序动态分析工具的重量级插桩框架，其中包含了很多出色的Valgrind工具，Memcheck是其中之一。

Valgrind的工具是在Valgrind内核基础上使用C语言编写的插件。最直观的表达就是：Valgrind内核 + 工具插件 = Valgrind工具。在Valgrind框架下编写一个二进制程序动态分析工具比从零开始简很多，因为Valgrind内核为新工具的编写提供了许多通用的工具集，比如错误记录、动态插桩等。

当Valgrind工具程序启动时，会将需要分析的程序加载与工具程序同一个进程空间中，然后使用JIT（just-in-time）的动态二进制重编译技术，将代码分成一个个小的代码块实施重编译。在重编译过程中，Valgrind内核会将相应代码块的机器码转化成中间表示，插件会在中间代码中进行相应分析代码的插桩，最后通过内核把中间表示转换成原本的机器码，在目标机器上执行。Valgrind内核大部分时间花在上述机器码和中间表示的相互翻译执行中，而原程序的所有机器码并没有执行，执行的都是插桩后的代码。

所有的Valgrind工具都是使用静态链接的可执行文件，里面包含了Valgrind内核和工具插件。虽然这样会导致每个工具程序中都需要包含一份Valgrind内核，内核大概2.5MB左右，稍微浪费一些磁盘空间，但是静态链接可以使整个可执行文件加载到非标准的启动地址，方便把待分析程序加载进同一个进程空间中，然后使用Valgrind重编译技术将待分析程序机器码重编译到别的地址执行。

Valgrind工作原理

工具启动

Valgrind工具，是在命令行中通过valgrind程序通过指定tool命令行参数为具体的工具名称来启动的。valgrind命令行程序只是一个很简单的封装程序，根据tool命令行来调用execv去执行目标Valgrind工具程序。然后Valgrind内核会先初始化各种子系统，如地址空间管理器、内部内存分配器等，再将待分析程序的.text和.data段等映射到同一个程序的地址空间中，并为其配置好堆栈等。接着，开始内核和插件中的命令行参数初始化和加载，并完成内核中更多子系统的加载，如翻译表、信号处理机制、线程调度器和调试信息等等。此时，Valgrind工具准备完毕，开始从待分析程序的第一条指令执行重编译和运行。

Valgrind内核和插件本身是运行在物理CPU上的，而待分析程序是运行在虚拟出的CPU上。由于有JIT手段来加速，整体看起来不像在解析执行，执行效率也比较高。物理CPU是直接使用真实CPU上的寄存器的，而虚拟CPU使用的寄存器则是保存在内存中的虚拟寄存器，同时影子寄存器也是类似的方法方便模拟出来。Valgrind为每个线程提供一个ThreadState结构，用于保存所有的虚拟寄存器和影子寄存器，方便后续的分析工作。

中间表示（VEX）

在Valgrind的重编译的过程中，使用的中间表示是一种平台无关的语言——VEX，通过屏蔽硬件平台的差异性，节省了大量针对不同平台的插桩代码。

Statement（结构体IRStmt）表示有副作用的操作，如写寄存器、写内存、临时变量赋值等。其中，Statement由Expression组成。Expression(结构体IRExpr)表示没有副作用的操作，如读内存、做算术运算等，这些操作可以包含子表达式和表达式树。

在Valgrind中，代码被分解成多个小的代码块，每个代码块里包含VEX的Statement列表。每个代码块的结构体是IRSB，IRSB是单入口多出口的，代码如下所示：

```
typedef
   struct {
      IRTypeEnv* tyenv; // 表明IRSB中每个临时变量的类型
      IRStmt**   stmts; // VEX语句列表
      Int        stmts_size; // Statements总长度
      Int        stmts_used; // 实际上使用的Statements的数目
      IRExpr*    next; // 下一跳的位置
      IRJumpKind jumpkind; // 最后代码块结束jump的类型
      Int        offsIP; // IP寄存器的偏移
   }
   IRSB;
```

Valgrind根据一定规则将代码划分为很多小代码块后，会进行以下八个阶段，将插件的分析代码进行插桩并优化：
1. **反汇编**：机器码 → 树状中间表示
2. **优化1**：树状中间表示 → 扁平中间表示
3. **插桩**： 扁平中间表示 → 带桩的扁平中间表示
4. **优化2**：带桩的扁平中间表示 → 优化的扁平中间表示
5. **重建树**：优化的扁平中间表示 → 带桩的树状中间表示
6. **汇编选择**：带桩的树状中间表示 → 目标汇编代码
7. **寄存器分配**：目标汇编代码 → 寄存器优化的目标汇编代码
8. **汇编**：寄存器优化的目标汇编代码 → 机器码

JIT执行

通过上面的插桩优化后得到的机器码会保存到一个定长、使用线性探测的哈希表中。如果哈希表达到了80%的程度，使用FIFO的策略将八分之一旧的重编译得到

的机器码丢弃。

当一个代码块执行完毕，下一个代码块的路由是靠dispatcher和scheduler两个组件来实现。首先，翻译后的代码执行完毕后，代码执行会陷入由汇编代码写的dispatcher，dispatcher内部有一个小的保存最近使用代码块的快速缓存，用于快速查找下一个需要用到的代码块，这样切换的效率非常高。

如果快查找失败了，控制流会到用C写的scheduler上面。scheduler会在上述的哈希表中查找已经翻译好的代码块，找不到的话，会进行重编译并加入哈希表中，还有更新dispatcher内部的快速缓存，以便dispatcher快速的后续使用。

对于system call、signal处理、事件机制等就不展开讨论了。

Valgrind回调Memcheck

```
static void mc_pre_clo_init( void );
static void mc_post_clo_init ( void );
IRSB* MC_(instrument) ( VgCallbackClosure* closure,
                        IRSB* sb_in,
                        const VexGuestLayout* layout,
                        const VexGuestExtents* vge,
                        const VexArchInfo* archinfo_host,
                        IRType gWordTy, IRType hWordTy );
```