wuhui_gdnt的专栏

个人资料

wuhui_gdnt

访问： 295372次
积分： 8541
等级： BLOG 6
排名： 第769名

原创： 549篇 转载： 0篇
译文： 30篇 评论： 43条

C、C++中未定义行为的指引， 第1部分

分类： C++语言 2013-03-15 11:16 1165人阅读 评论(0) 收藏 举报

目录(?) [+]

作者：John Regehr

原作：http://blog.regehr.org/archives/213

编程语言通常区分正常的程序活动与错误的活动。对于图灵完备语言，我们无法可靠地离线确定一个程序是否潜在执行一个错误；我们必须运行它来看一下。

在一个安全的编程语言中，错误在发生时被捕捉。例如，通过其异常系统，Java大体上是安全的。在一个不安全的编程语言中，错误不被捕捉。相反，在执行一个错误操作后，程序继续运行，但以一个静默、错误的方式，稍后可能有可观察的后果。Luca Cardelli关于类型系统的论文对这些问题有一个非常清晰的介绍。C与C++在很大程度上是不安全的：相对于让错误操作有一个不可预测的结果，执行一个错误操作使得整个程序变得没有意义。在这些语言

中，错误操作被称为具有未定义行为（undefined behavior）。

C FAQ这样对于“未定义行为”：

任何事情都可能发生；标准没有强加任何要求。程序可能编译失败，或者它可能执行不正确（要么崩溃，要么悄悄地产生不正确的结果），或者它可能幸运地正好做了程序员所希望的。

这是一个好的总结。几乎所有C及C++程序员都理解访问一个空指针以及除0是错误的行为。另一方面，未定义行为的完整含义及其与进取编译器间的交互未能被很好领会。

未定义行为的一个模型

现在，我们可以忽略编译器的存在。仅存在“C实现”，如果该实现符合C标准——在执行一个合乎标准的程序时，与“C抽象机器”的行为相同。C抽象机器是C标准描述的一个简单的C解析器。我们可以用它来确定任何C程序的含义。

一个程序的执行包含简单的步骤，比如两个数相加或跳转到一个label。如果在一个程序执行中的每步具有已定义的行为，那么整个执行是定义良好的。注意即使定义良好的执行也可能因为未指定（unspecified）及实现定义的行为，而没有唯一的结果；这里我们将忽略这两者。

如果在一个程序的执行中任一步具有未定义行为，那么整个执行是没有意义的。这是重要的：对(1<<32)求值不是不可预测的结果，而是这个程序的整个执行没有意义。同样，不是直到该未定义行为发生点的执行有意义：不良影响实际上可以出现在未定义行为之前。

作为一个简单的例子，我们使用这个程序：

```
#include <limits.h>

#include <stdio.h>
```

```
int main (void)

{

  printf ("%d\n", (INT_MAX+1) < 0);

  return 0;

}
```

该程序要求C实现回答一个简单的问题：如果我们向最大可表达整数加1，结果是负的吗？对于一个C实现，这是一个非常合法的行为：

```
$ cc test.c -o test

$ ./test

1
```

这个也是：

```
$ cc test.c -o test

$ ./test

0
```

还有这个：

```
$ cc test.c -o test
```

```
$ ./test

42
```

还有这个：

```
$ cc test.c -o test

$ ./test

Formatting root partition, chomp chomp
```

有人可能会说：这些编译器中的一些行为不正确，因为C标准宣称一个关系操作符必须返回0或1。但因为程序完全没有意义。实现可以做任何它想做的事。未定义的行为超出C抽象机器的其它行为。

一个编译器实际产生的代码会破坏你的硬盘吗？当然不会，但记住实际上来说，未定义行为通常确实导致坏的事情，因为许多安全漏洞始于具有未定义行为的内存或整数操作。例如，访问一个界外数组元素是常规栈破坏攻击的关键部分。总而言之：编译器不需要产生格式化你硬盘的代码。相反，随着OOB数组访问，你的计算机将开始执行漏洞利用代码，正是该代码格式化你的硬盘。

行不通

人们很容易说——或至少认为这样：

x86 ADD指令用于实现C有符号数相加操作，在结果溢出时它具有2机制补码行为。我正在一个x86平台上开发，因此在32位有符号整数溢出时，我应该可以期望2机制补码语义。

这是不对的。你在表述这样的东西：

有人曾经告诉我，在篮球比赛中你不能抱着球跑。我拿了篮球尝试，这样挺好的。他显然不理解篮球。

（这个解释归功于Roger Miller通过SteveSummit做出）。

显然物理上捡起篮球，带着它跑是可能的。在一场比赛中你侥幸这样做也是可能的。不过，它违反了规则；好的运动员不会这样做，而差的运动员不会总是好运。在C或C++中对(INT_MAX+1)求值也一样：有时可能能工作，但不要指望总是这样。情形实际上有些微妙，让我们看深入些。

首先，存在在有符号整数溢出时，确保2机制补码行为的C实现吗？显然有的。例如，许多编译器在关闭优化时具有这个行为，GCC有一个选项（-fwrapv）在所有优化级别强制这个行为。其它编译器默认在所有优化级别具有这个行为。

不言而喻，也存在编译器，对有符号溢出，没有2机制补码行为。另外，有些编译器（像GCC）多年来以特定的方式处理整数溢出，然后某个时候优化器变得更聪明些，而整数溢出突然悄悄地不能预期工作了。就标准而言，这完全没问题。尽管对开发者不友好，它应该被视为编译器团队的胜利，因为它将提升基准评分。

总结：带着一个球跑本质上没有不对的地方，同样以33位移位一个32位数本质上也没有不对。但一个违反了篮球规则，而另一个违反了C与C++的规则。在这两个情形下，设计游戏的人已经制订了随意的规则，我们要么遵循它们，要么寻找更喜欢的游戏。

为什么未定义行为是好的？

好的——关于C/C++中未定义行为仅有的好处！是：它简化了编译器的工作，使得在特定情形下产生高效的代码成为可能。这些情形通常涉及紧凑循环。例如，高性能数组代码不需要执行边界检查，避免使用复杂的优化遍从循环移出这些检查。类似地，当编译一个递增一个有符号整数的循环时，C编译器无需担心该变量溢出及变为负的情形：这使几种循环优化变得容易。我听说当允许编译器利用有符号整数溢出的未定义属性时，某些紧凑的循环加速了30%~50%。类似地，有C编译器可选地向无符号整数溢出给出未定义语义，以加速其它循环。

为什么未定义行为是坏的？

当程序员不能被信任能避开未定义行为时，我们最终具有悄悄不能正确工作的程序。对于像网络服务器及网络浏览

Linux

Modutils v2.4下载点

器这样处理敌对数据的代码，这已经被证明是一个非常糟糕的问题，因为这些程序最终妥协并运行从网络来的代码。在许多情形下，我们实际上不需要利用未定义行为来获取性能提升，但由于老旧的代码及工具链，我们被恶劣的后果缠住了。

一个不那么严重、更令人烦恼的问题是：行为被未定义，完全在于使得编译器作者的工作更容易些，并没有性能的提升。例如一个C实现具有未定义行为，当：

在符号化期间，在一逻辑行上遇到一个未匹配的‘或”字符。

无意冒犯C标准委员会，这是懒惰。要求C实现者在引号不匹配时发出一个编译器时错误消息，这会是个过分的负担吗？这方面一个甚至有30年历史（C99在那时标准化）的系统编程语言都做得更好。有人怀疑C标准只是习惯把行为丢入“未定义”箩中，并且有点过火了。实际上，C99标准列出191种未定义行为，说他们太过火是公允的。

理解编译器如何看待未定义行为

设计一个带有未定义行为的编程语言背后关键的洞察是：编译器仅有义务考虑行为是有定义的情形。我们现在将探索这个问题的含义。

如果想象一个C程序将要被C抽象机器执行，未定义行为非常容易理解：程序执行的每个操作要么是已定义的，要么是未定义的，通常这相当清楚。当我们开始关注一个程序所有可能的执行，未定义行为开始变得难以处理。应用程序开发者，需要代码在任何情形下都正确，对此在意，同样还有编译器开发者，他们需要产生在所有可能的执行中正确的机器代码。

讨论一个程序所有可能的执行有点棘手，让我们做一些简化的假设。首先，我们将讨论单个C/C++函数，而不是整个程序。其次，我们将假定函数对每个输入都终止。第三，我们将假定函数的执行是确定性的；例如，它没有通过共享内存与其它线程协作。最后，我们假设具有无限的计算资源，使穷举测试该函数成为可能。穷举测试意味着尝试所有可能的输入，不管它们来自实参，全局变量，文件I/O，还是其它。

穷举测试算法是简单的：

1. 计算下一个输入，如果我们已经尝试了所有，终止
2. 使用这个输入，在C抽象机器中运行这个函数，记录是否执行了任何带有未定义行为的操作
3. 回到步骤1

枚举所有的输入不是太困难。

以该函数接受的最小输入（以比特衡量）开始，尝试该大小所有可能的比特模式。然后进入下一个大小。这个过程可能终止、也可能不终止，但没关系，我们有无限的计算资源。

对于包含未指定及实现定义行为的程序，每个输入可能导致几个或许多可能的执行。这不会从根本上使情况变得更复杂。

好了，我们的思维实验达到了什么目的？对于我们的函数，现在我们知道它落入了哪个类别：

- 类型1：对所有输入行为是已定义的
- 类型2：对某些输入行为是已定义的，而对于其它则没有
- 类型3：对于所有输入，行为是未定义的

类型1函数

这些对它们的输入没有限制：对于所有可能的输入，它们都行为良好（当然，“行为良好”可能包括返回一个错误值）。通常，API级别的函数以及处理未净化数据的函数，应该是类型1的。例如，这是一个进行整数除法，而不会执行未定义行为的实用函数：

```
int32_t safe_div_int32_t (int32_t a, int32_t b) {

  if ((b == 0) || ((a == INT32_MIN) && (b == -1))) {

    report_integer_math_error();
```

```
    return 0;

  } else {

    return a / b;

  }

}
```

因为类型1函数永远不会执行带有未定义行为的操作，无论函数的输入是怎样的，编译器都有义务产生合理的代码。我们无需进一步考虑这些函数。

类型3函数

这些函数没有容许良好定义的执行。严格地说，它们完全是无意义的：编译器甚至没有义务产生一条return指令。类型3的函数真实存在吗？是的，而且事实上它们是普遍的。例如，不管输入，一个函数使用一个变量而不初始化它，是很容易无意地写出的。在识别及利用这种代码方面，编译器越来越聪明。这里是来自Google NativeClient项目的一个极好的例子：

当从信任或非信任代码返回时，在获取返回地址之前，我们必须净化它。这确保非信任代码不能使用syscall接口向量执行（vector execution）到一个任意地址。这个任务委托给在sel_ldr.h中的函数NaClSandboxAddr。不幸的是，自r572起，在x86上这个函数成了一个空操作。

——发生了什么？

在一个例程重构期间，代码以前读作

```
aligned_tramp_ret= tramp_ret & ~(nap->align_boundary - 1);
```

改变为读作

```
return addr& ~(uintptr_t)((1 << nap->align_boundary) - 1);
```

除了变量重命名（这是内部且正确的），引入了一个移动，把nap->align_boundary处理作包大小的log2。

我们没有注意到这，因为在x86上NaCl使用一个32字节的包大小。在x86上使用gcc，(1 << 32) == 1。（我相信标准把这个行为保留为未定义，但我对此生疏）。这样，整个沙盒序列成了一个空操作。

这个改动有4个登记的审核者，两个明确表示我看没问题。看起来没有人注意到这个改动。

——影响

在32位x86上的非信任代码通过构造一个返回地址，并进行一个syscall，潜在不对齐其指令流的可能。这可以破坏验证器。一个类似的漏洞可能影响x86-64。

出于历史原因ARM不受影响：ARM实现使用不同的方法来掩码非信任返回地址。

发生了什么？一个简单的重构使得包含这个代码的函数成为类型3。发送这个信息的人相信x86-gcc把(1<<32)求值为1，但没有理由期望这个行为是可靠的（事实上，在我尝试的几个x86-gcc版本上，不是这个行为）。这种构造绝对是未定义的，编译器当然可以做任何它想做的事。对应一个未定义操作，典型地，一个C编译器选择不生成任何指令（C编译器的第一要务是产生高效的代码）。一旦Google程序员给了编译器杀人执照，它会毫不犹豫地杀戮。有人会问：如果当检测到一个类型3的函数时，编译器提供一个警告或类似的东西不是很棒吗？是的！但这不是编译器优先要做的。

Native Client是一个好例子，因为它展示了合格的程序员会被一个优化编译器利用未定义行为的秘密行径所迷惑。在开发者看来，在识别并悄悄销毁类型3函数方面非常智能的编译器实际上变得邪恶。

类型 2函数

这些的行为对于某些输入具有定义，而对其它则没有定义。对于我们的目的，这是最有趣的的案例。有符号整数除法构成了一个好例子：

```
int32_t unsafe_div_int32_t (int32_t a, int32_t b) {

  return a / b;

}
```

这个函数有一个先决条件；它仅应该为满足这个断言的实参调用：

```
 (b != 0) && (!((a == INT32_MIN) && (b == -1)))
```

当然，这个断言看起来非常像这个函数类型1版本的测试不是巧合。如果你，调用者，违反了这个先决条件，你程序的意义将被破坏。编写像这样带有非平凡先决条件的函数可以吗？通常，对于内部使用的函数，这完全没问题，只要先决条件被清晰文档化了。

现在让我们看一下当把这个函数翻译到目标代码时，编译器的工作。编译器进行一个案例分析：

- 案例1: (b != 0) && (!((a == INT32_MIN) && (b == -1)))

/操作符的行为是已定义的à编译器有义务产生计算a / b的代码。

- 案例2: (b == 0) || ((a == INT32_MIN) && (b == -1))

/操作符的行为是未定义的à编译器没有特定的义务

现在编译器作者问他们自己这个问题：什么是这两个情形最有效的实现？因为情形2不招致任何义务，最简单的做法是不考虑它。编译器可以仅为情形1产生代码。

相反，一个Java编译器在情形2中负有义务，必须处理它（虽然在这个特定的案例中，很可能没有运行时开销，因为处理器通常为整数除0提供陷入行为（trapping behavior））。

让我们看另一个类型2函数：

```
int stupid (int a) {

  return (a+1) > a;

}
```

避免未定义行为的先决条件是:

```
 (a != INT_MAX)
```

这是由一个C或C++优化编译器进行的案例分析:

- 案例1: **a != INT_MAX**

+的行为已定义 à 编译器有义务返回1

- 案例2: **a == INT_MAX**
  +的行为未定义 à 编译器没有特定的义务

再次，案例2从编译器的推理中退化并消失。案例1就是一切。这样，一个好的x86-64编译器将产生:

```
stupid:

  movl $1, %eax

  ret
```

如果我们使用-fwrapv标记告诉GCC整数溢出具有2机制补码行为，我们得到一个不同的案例分析:

- 案例1: **a != INT_MAX**

+的行为已定义 à 编译器有义务返回1

- 案例2: **a == INT_MAX**
  +的行为已定义 à 编译器有义务返回0

这里的案例不会崩溃，编译器有义务实际执行加法并检查其结果：

```
stupid:
  leal 1(%rdi), %eax
  cmpl %edi, %eax
  setg %al
  movzbl %al, %eax
  ret
```

类似地，一个预编译（ahead-of-time）Java编译器也必须执行加法，因为在一个有符号整数溢出时，Java责令2机制补码行为（对x86-64我使用GCJ）：

```
_ZN13HelloWorldApp6stupidEJbii:
  leal 1(%rsi), %eax
  cmpl %eax, %esi
  setl %al
  ret
```

这个未定义行为崩溃案例的观察提供了一个有力的方式来解释编译器实际如何工作。记住，它们的主要目的是给你遵守法律条文的快速代码，因此它们将尝试尽可能快地忘记未定义行为，而且不会告诉你这个。

一个有趣的案例分析

大约1年前，Linux内核开始使用一个特殊的GCC标记来告诉编译器避免优化掉无用空指针检查。促使开发者添加这个标记的代码看起来像这样（我稍微清洁了一下该例子）：

```
static void __devexit agnx_pci_remove (struct pci_dev *pdev)

{

  struct ieee80211_hw *dev = pci_get_drvdata(pdev);

  struct agnx_priv *priv = dev->priv;



  if (!dev) return;

  ... do stuff using dev ...

}
```

这里的习语是得到指向一个设备结构体的指针，测试它是否空，如果使用它。但有一个问题！在这个函数里，在空指针检查前该指针被提领了。这导致一个优化编译器（例如，gcc在-O2或更高优化级别）执行以下案例分析：

- 案例1：**dev == NULL**

dev->priv有未定义行为 à 编译器没有特定的义务

- 案例2：**dev != NULL**

空指针检查不会失败 à 空指针检查是死代码，可能被删除

正如我们现在很容易看出，没有一个案例使得空指针检查成为必须。该检查被移除，潜在地创建了一个可利用的安全漏洞。

当然，这个问题是pci_get_drvdata()返回值的检查前使用，这必须通过把使用移到检查后来修正。但直到可以审查所有这样的代码（人工或通过工具）前，告诉编译器稍微保守些，被认为更安全。由于像这样的可预测分支导致的效率损失总体可以忽略不计。在内核的其它部分也找到类似的代码。

与未定义行为和平共处

从长远来看，非安全的编程语言将不会为主流开发者使用，而是专用于高性能及低资源足迹是关键的情形。与此同时，未定义行为的处理不完全是直截了当的，拼凑（patchwork）的做法看起来是最好的：

- 启用并注意编译器警告，最好使用多个编译器
- 使用静态分析器（像Clang，Coverity等）来得到更多警告
- 使用编译器支持的动态检查；例如，gcc的-ftrapv标记产生捕捉有符号整数溢出的代码
- 使用像Valgrind的工具来得到额外的动态检查
- 当函数是上面分类的“类型2”时，归档它们的先决条件与后承条件
- 使用断言来验证该函数的先决条件是实际成立的后承条件
- 特别在C++中，使用高质量的数据结构库

最基本地：非常小心，使用好的工具，抱乐观的希望。

上一篇 位置无关代码
下一篇 C、C++中未定义行为的指引， 第2部分

主题推荐 c++ linux内核 网络服务器 编程语言 安全漏洞

查看评论

暂无评论

您还没有登录,请[登录]或[注册]