论文笔记

题目：SoK: Sanitizing for Security

出处：IEEE Symposium on Security and Privacy 2018

作者：Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen, Michael Franz

单位：University of California, Irvine

原文：https://arxiv.org/pdf/1806.04355

一、背景

目前，C/C++任然是主流的程序开发语言，特别是对于一些低级的系统软件，像操作系统内核、运行时库和浏览器等。它们被开发人员所青睐的主要原因是：这些语言所写的代码运行效率高，程序员可以充分控制系统底层软硬件资源。但是，开发人员在享受它们所带来的便利的同时，也必须足够小心的去实现自己的代码，如果代码没有考虑周全，那就可能会造成程序漏洞、运行时崩溃等问题。

二、提出的方法以及解决的问题

由于市面上已经出现了各种各样的漏洞检测工具，因此，针对前文提及C/C++程序容易引入漏洞的情况，作者搜集了一系列的检测工具，包括静态分析工具（有基于源码分析的、也有基于中间语言（IR）分析的）、和动态分析工具，并对Github上排名最靠前的100个C项目和100个C++项目进行评估、测试，并衡量这些测试工具的性能和测试结果。文章主要评估程序的以下几个方面：

- Memory Safety Violations（内存安全违规）

```
// 例如如下所示的例子会造成写越界
struct A { char name[7]; bool isAdmin; };
struct A a; char buf[8];
memcpy(/*dst*/ a.name, /*src*/ buf, sizeof(buf));
```

- Use of Uninitialized Variables（使用未初始化的变量）

```
// 如下例子只初始化部分成员变量
struct A { int data[2]; };
struct A*p = (struct A*)malloc(sizeof(struct A));
p->data[0] = 0; // Partial initialization
send_to_untrusted_client(p, sizeof(struct A));
```

- Pointer Type Errors（指针（转换）类型错误）

```
// 如下类指针的不兼容转换
class Base { virtual void func(); };
```

```
class Derived : public Base { public: int extra; };
Base b[2];
Derived *d = static_cast<Derived*>(&b[0]); // Bad-casting
d->extra = ...; // Type-unsafe, out-of-bounds access, which
// overwrites the vtable pointer of b[1]
```

- Variadic Function Misuse（变参函数误用）

```
// 如下所示的用户可控的字符串格式化参数
char *fmt2; // User-controlled format string
sprintf(fmt2, user_input, ...);
// prints attacker-chosen stack contents if fmt2 contains
// too many format specifiers
// or overwrites memory if fmt2 contains %n
printf(fmt2, ...);
```

- Other Vulnerabilities（还有其它类型的脆弱性，例如（整数）溢出错误）

```
// 如下所示可能会导致溢出
// newsize can overflow depending on len
int newsize = oldsize + len + 100;
newsize *= 2;
// The new buffer may be smaller than len
buf = xmlRealloc(buf, newsize);
memcpy(buf + oldsize, string, len); // Out-of-bounds access
```

三、技术方法

针对以上几种漏洞类型，作者给出了对应的检测措施和检测工具等。

首先，对于内存安全违规的问题，有两种对应的检测策略，分别是：Location-based Access Checkers（基于访问位置的检测策略）和 Identity-based Access Checkers（基于实体访问的检测策略）。前者检测内存访问的内存区域是否是合法的内存区域，并且对每一个可访问的字节都维护一个元数据结构，当程序访问内存的时候去检测对应的元数据，判断该内存访问是否合法，它使用的方法有： red-zone insertion、 guard pages 和 memory-reuse delay 等。而后者则是对于每一个被分配出来的内存对象维护一个元数据结构，并且有一个本地策略去判断程序的每一个指针访问是否合法，它使用的方法可以是：per-object bounds tracking、 per-pointer bounds tracking、 lock-and-key checking 和 dangling pointer tagging 等。

然后，对于使用未初始化变量的问题，使用扩展的 Location-based access checkers 策略来检测对未初始化内存的读取问题，它通过标记新分配出来的内存对象，并把标记信息存入元数据结构中，然后再对内存读写取指令进行插桩，如果是读指令，则检查元数据结构，判断该读操作所读取的内存区域是否是未初始化的，如果是写操作，则清除目标内存区域的元数据结构。

其次，对于指针类型的问题，检测的是不兼容指针转换与解引用。使用的策略是监控指针转换（例如C++的static_cast操作符等）和监控指针使用（例如指针转换为不兼容类型之后解引用）。

再其次，对于变参函数误用的问题，例如格式化字符串产生的漏洞和参数不匹配导致的漏洞等，文章中介绍的策略是重定向 printf 函数，拦截对 printf 函数的调用，并在调用 pirntf 函数之前对它的参数进行检查，检查通过之后再调用目标 printf 函数。

最后，对于其他类型的问题，例如整数溢出等，使用的检测策略是：Stateless Monitoring。例如文中提到的经典检测工具：UndefinedBehaviorSanitizer(UBSan)。

另外，文中作者所涉及到的大部分工具都是基于程序插桩而实现的，只有少部分工具不是使用程序插桩技术的。虽然大部分工具都使用程序插桩技术，但是他们插桩的时机并不相同，文中列举出34个检测工具，其中，基于源代码级别插桩的工具有16个，基于中间语言（IR）插桩的工具有16个，基于二进制插桩的工具有4个，基于类库之间的调用（Library interposition instrument）插桩的工具有3个，部分工具同时使用了不同级别的插桩技术。与此同时，文中介绍的这些工具在元数据管理方面也有不同，分别有：基于对象的元数据（每一个内存对象对应一个元数据结构）、基于指针的元数据（每一个指针对应一个元数据结构）和静态元数据（保存被编译器丢弃的数据信息，在运行时需要用到这些信息，用于检测漏洞）。

四、实验评估

作者在文章中只评估了排名靠前的十个检测工具。作者所使用的测试平台是：Intel Xeon E5-2660 CPU with 20MB cache and 64GB RAM running 64-bit Ubuntu 14.04.5 LTS。在测试过程中，对于每一个工具，作者运行三次，取中间值作为测试结果，然后再单纯运行目标程序三次，不运行检测工具，取三次的平均值作为基准，去衡量性能开销。这是个测试工具的测试性能如下所示：

- Memcheck: 平均是基准程序的19.6倍。
- AddressSanitizer: 平均是基准程序的1.99倍。
- Low-fat Pointer: 平均是基准程序的1.93倍。
- DangSan: 平均是基准程序的1.4倍。
- MemorySanitizer: 平均是基准程序的2.53倍。
- TypeSan: 平均是基准程序的1.26倍。
- HexType: 平均是基准程序的1.06倍。
- Clang CFI: 平均是基准程序的1.09倍。
- HexVASAN: 平均是基准程序的1.01倍。
- UndefinedBehaviorSanitizer: 平均是基准程序的2.97倍。

五、优缺点

优点：

- 作者做了一个比较大范围的工具搜集与测试工作，并分析了每一个工具使用的技术以及检测的漏洞类型。
- 作者对这些工具使用的技术原理讲解的比较清晰。
- 作者对这些漏洞进行了分类，并针对这些不同的类型，需要用到的检测技术也进行了分类了说明。

缺点：

- 作者没有给出这些工具对这些数据集的的误报率和漏报率。
- 作者在文章中只评估了前十个工具的测试性能与测试结果，没有对文中提到的所有工具做评估。
- 作者在检测每一个工具的时候选择的基准是单纯运行目标程序所需要的性能开销，但是如果是选择其中的某一个漏洞检测工具作为基准程序的话，其它的检测工具的性能开销都相对于该基准程序，这样可能会有另一种直观的、各个工具之间的比较结果。

六、个人观点

在这篇文章中，作者做了一个调查类的测试工作，搜集了各种漏洞检测工具，并一一对他们的性能和功能进行测试和评估，但是在文章中，作者也并没有对自己搜集的所有工具进行一个完全的测试，只是对靠前的十个工具进行了评估与测试，并且只有对这些工具的一个运行性能结果，没有对这些工具的误报率和漏报率进行汇报。文章中只说明了为什么会出现误报和漏报，但是并没有给出这些工具的具体误报率和漏报率，不知道这个问题是不是作者故意回避的问题。但是，作者做这篇文章的工作量还是很大的，需要花费大量的时间去收集这些工具，并让这些工具检测目标基准程序集，有时候编译目标项目可能还会遇到很多问题，作者在文章中虽然没有提及，不过这确实也会花费不少时间去解决程序编译的问题，以便于测试工具能够正确的去运行检测工作。