Attack Surface Reduction

Date	@Apr 18, 2021
Source	usenix
Tags	kernel security

> 通过移除不必要的应用特性和代码可以有效的减少攻击面从而提高程序的安全性。在程序的初始化阶段一般会需要较多的系统调用（execve和fork），当程序初始化完成后不再需要此类系统调用时，可以禁用这些系统调用来提升程序运行时的安全性

Introduction

对于服务器应用程序来说，在程序启动的时候需要一些高特权的系统调用，但是在程序初始化后，这些系统调用大部分不再被需要。但是这些高特权的系统调用仍然可以被应用程序访问，也就增加了攻击面；下面是一些冗余情况：

- 不需要的模块和插件
- 没有被引用的库函数
- 不会使用的系统调用

目前的一些处理此类问题减少攻击面的方法：

- 通过静态分析来辨别不会再使用的共享库
- 用动态分析来确定不会再使用的程序部分
- 容器中也有使用该类方法来提高容器的安全性

本篇论文主要关注点在服务器应用程序并且将程序看成一个动态的变化过程将其分为初始化阶段和服务阶段（之前的处理方法是把应用程序当作单一不变的实体），程序的初始化阶段和服务阶段所依赖的共享库和系统调用必然会不同。该paper的motivation主要是

例如execve这样的危险的系统调用常常被用作exp，对于服务器应用程序来说socket套接字常常实在程序初始化阶段创建，当程序进入到稳定的运行阶段时候，对此类系统调用做一个限制。极大的减少了攻击面

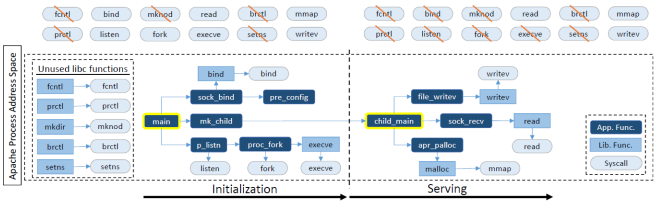
定位识别系统调用的主要是建立完备的CFG，现在的指针分析依然存在分析结果不精确的问题（比如会），本文提出了两个减枝算法可以减少CFG多余的边增加精确度。在确认需要的系统调用后，使用seccomp BPF来block系统初始化之后不再需要的系统调用来减少攻击面

对比现在的specialization approaches方法，本文可以禁用51%有安全风险的系统调用

1. 提出了基于服务器应用程序基于时序的系统调用具体化方法（把服务器软件的运行生命周期划分许多小phase，每个phase禁用不同的系统调用）
2. 提出了基于类型和基于地址的剪枝机制来提高静态分析构造的CFG的精确度
3. 使用6个流行的服务器软件（nginx ,apache httpd, lighttpd, bind, memcached,redis）和567种shellcode和17ROP样本来测试系统在减少攻击面和增加系统稳定性的有效性

Static vs. Temporal API Specialization

之前的减少软件攻击面都是把软件运行生命周期当作一个整体，但是对于服务器软件来说初始化阶段和稳定服务阶段所需要的系统调用是不同的。初始化阶段主要的操作包括解析配置文件，创建socket套接字，绑定网络端口，创建worker进程。但是稳定服务阶段主要操作就是从socket或者其他文件中读写数据，内存管理。



Seccomp BPF

由Linux内核提供的限制用户态可以访问的系统调用，seccomp BPF使用BPF语言来开发来限制系统调用。seccomp是内核提供的一种SYSCALL过滤机制，它基于BPF过滤方法，通过写入BPF过滤器代码来达到过滤的目的。BPF规则语言原生是为了过滤网络包，

情景比较复杂。针对SYSCALL场景，语法比较固定，可以自行撰写，也可以基于Libseccomp库提供的API来编写

Design

本文的主要目标是当服务器应用程序完成初始化后通过对与系统调用做一个限制来减少攻击者可以利用的系统调用。在服务器应用程序进入到server阶段时，如何确定在初始化阶段需要但是在服务阶段不需要的系统调用，需要如下steps：

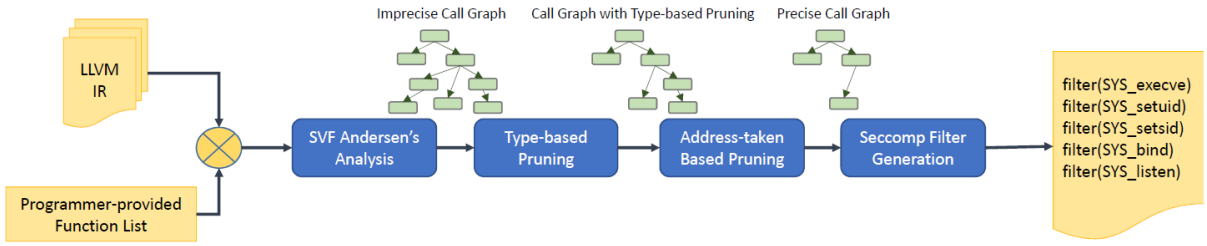


Figure 2: Overview of the process for generating a sound call graph to identify the system calls required by each execution phase.

1. 构建完备的调用图，并且追溯外部库导入的函数列表
2. 将程序的调用图和外部库中的函数映射到系统调用
3. 标识初始化阶段和服务阶段的开始地方
4. 基于CG，确定各个阶段需要的系统调用
5. 对未使用的系统调用创建seccomp的过滤函数限制这些系统调用的访问

Identifying the Transition Point

初始化阶段和服务阶段所依赖的系统调用函数是不同的，找到转换点是很重要的。Apache httpd和nginx的转变点在创建子进程的时候，memcached这类事件驱动服务器，转变点在event loop开始的时候

可以通过启发式算法或者动态分析也可以找到转换点，本文没有做出这个方面的工作，对于每个程序来说只需要找一次，工作量相对较少，所以都是手动找到转换点

Call Graph Construction

C/C++编写应用程序经常使用函数指针，在编译后会产生间接跳转指令。对于建立完备的调用图来说，正确处理间接跳转指令很重要。本文通过Andersen's指针分析来恢复间

接跳转，但是此类算法容易产生误报

Points-to Analysis Overapproximation

指针分析包括字段敏感性、上下文敏感性和路径敏感性（敏感度：path sensitivity > context sensitivity > field sensitivity）。一般而言，抽象过程中考虑的信息越多，程序分析的精度就越高，但分析的速度就越慢，Andersen's算法是field sensitivity，所得到的结果是不精确的。

Pruning Based on Argument Types

Andersen指针分析算法在求解约束图时没有考虑到任何关于指针类型的语义。在处理间接跳转时候，只考虑参数的个数，而不考虑参数的类型，所以在做指针分析后需要用参数类型做一个剪枝

Puring Based on Taken Address

一个函数只有在程序中某个地方获取了它的地址并存储到一个变量中，才能成为间接调用点的目标。因此，当一个函数地址是在程序中的某个点获得的，而这个点是从main函数不能到达的，那么它就永远不会成为间接调用的目标。通过这种方法可以对CG进行进一步的剪枝

Algorithm 1: Generation of Precise Call Graph

Input: LLVM IR bitcode for the target application
Output: *precise_cg*: precise application call graph

```
1  Run SVF's Andersen points-to analysis to get the
     (overapproximated) call graph cg;
2  /* Perform argument-type pruning */
3  foreach Indirect-callsite ic in cg do
4      foreach Target t of ic in cg do
5          if Argument types of t does not match that of ic
             then
6              Prune target t for ic;
7          end
8      end
9  end
10 addr_taken_fn_set ← ∅;
11 reachable_functions ← ∅;
12 /* Collect address-taken functions */
13 Traverse cg depth-first, starting from main;
14 foreach Reachable function func from main do
15     reachable_functions ∪ {func};
16 end
17 foreach Function f in reachable_functions do
18     foreach Address-taken function f_addr_tk in f do
19         addr_taken_fn_set ∪ {f_addr_tk};
20     end
21 end
22 /* Perform address-taken pruning */
23 foreach Indirect-callsite ic in cg do
24     foreach Target t of ic in cg do
25         if t ∉ addr_taken_fn_set then
26             Prune target t for ic;
27         end
28     end
29 end
30 precise_cg ← cg;
```