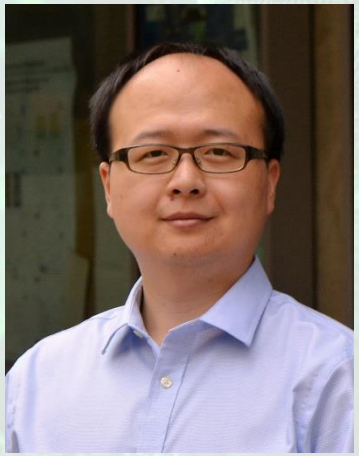
2017 中国互联网安全大会
China Internet Security Conference

基于源代码的自动程序分析在内核安全中的应用

宋程昱 加州大学河滨分校助理教授

个人简介

- UCR 助理教授，佐治亚理工博士，北京大学硕士、学士
- 从事安全研究超过10年
- 主要研究方向：漏洞的发现、利用及防护，安全攻防，可信计算
- 在系统安全顶级学术会议发表论文14篇
- 3个 BLACKHAT USA 报告
- DARPA CGC 决赛队伍成员
- 常年招收博士生



目录

基于LLVM的静态分析

应用案例

展望及讨论

ISC 2017 中国互联网安全大会

360互联网安全中心

基于LLVM的静态分析

LLVM编译器框架

- 模块化，高复用性的编译器框架
- 前端：CLANG (C/C++/OBJECTIVE-C), CUDA, HASKELL, ETC.
- 后端：X86, ARM, ARM64, NVPTX, RISCV, WEBASSEMBLY, ETC.
- 其他：LINKER，COMPILER-RT，KLEE，LLDB，LIBC++，ETC.

LLVM IR

- LLVM的核心（HTTPS://LLVM.ORG/DOCS/LANGREF.HTML）
- 强类型，RISC-LIKE，SSA（STATIC SINGLE ASSIGNMENT）
- 基于C++的开发接口
  （HTTPS://LLVM.ORG/DOCS/PROGRAMMERSMANUAL.HTML）

LLVM Pass

- 基于LLVM IR的分析和转化程序
- 层次：MODULE, SCC, FUNCTION, LOOP, REGION, BB, ETC.
- 模块化：PASSMANAGER, REQUIRED, PRESERVED
- 基础教程：
  HTTPS://LLVM.ORG/DOCS/WRITINGANLLVMPASS.HTML

漏洞挖掘常用分析

- 数据流分析：DEF-USE, TAINT ANALYSIS, ETC.
- 控制流分析：CFG, CALL GRAPH, DOMINATOR, ETC.
- 指针（POINT-TO）分析：BASICAA, ANDERSON, DSA, ETC.
- 符号执行：CLANG STATIC ANALYZER
- 类型（TYPE）分析
- 值域（RANGE）分析

相比于其他分析平台（如CIL），LLVM入门门槛低，且有大量资源

LINUX内核分析 (1)

难点一：如何将内核源代码编译为LLVM IR

- LINUX内核大量使用GCC独有的非标准C特性
- LINUX社区只考虑支持GCC
- LLVM社区不考虑支持非标准C特性
- LLVMLINUX项目（HTTP://LLVM.LINUXFOUNDATION.ORG/ ）

LINUX内核分析 (2)

难点二：如何进行全内核分析

- LLVM常见分析（PASS）的最大分析单元是MODULE，即单个源文件
- LLVM-LINK → 重命名问题（不推荐）
- LTO → 速度较慢，适合分析后需要插装的应用
- 自主链接MODULE（KINT，KERNEL-ANALYZER）

应用案例 (1)

自动化识别并保护内核中的重要数据 [NDSS 16]

动机

- 内存错误型漏洞在内核中较为常见
- 现有的ROOT，越狱工具都利用了内核中的内存错误漏洞
- 内存错误型漏洞的利用十分灵活
- **问题**：如何找到所有可以被攻击的对象

例子

```
1  static int acl_permission_check
2      (struct inode *inode, int mask)
3  {
4    unsigned int mode = inode->i_mode;
5
6    if (likely(uid_eq(current_fsuid(), inode->i_uid)))
7      mode >>= 6;
8    else if (in_group_p(inode->i_gid))
9      mode >>= 3;
10
11   if ((mask & ~mode &
12        (MAY_READ | MAY_WRITE | MAY_EXEC)) == 0)
13     return 0;
14   return -EACCES;
15 }
```

代码注入 / 修改型攻击

取消检查

控制流劫持型攻击

绕过检查

数据型攻击

误导检查

现有工作的不足

- SECURE BOOT 不能解决后两种攻击
- KCFI 不能解决数据型攻击
- KASLR 容易通过信息泄漏型攻击（包括测信道攻击）绕过
- 针对常用攻击对象（CRED）的保护可以被绕过

观察

- 提权攻击的本质是攻击内核的访问控制（ACCESS CONTROL）系统
- 访问控制系统有三条原则
- 全面性 → 不能被绕过 → 必须保护所有的控制数据
- 抗攻击 → 不能被误导 → 必须保护所有的数据依赖
- 正确性 → 假设正确

问题

如何系统性的发现所有需要保护的数据？

- 控制数据 → CODE POINTER INTEGRITY → 基于类型
- 数据依赖 → 安全检查 → 如何发现安全检查？

解决方案

利用系统调用的返回值

- 系统调用的返回值具有明确的语义
- LINUX：EACCESS, EPERM, EROFS, ETC.
- WINDOWS: ERROR_ACCESS_DENIED, ETC.
- 优点：系统，全面，可自动化

例子 (1)

第一步: 收集函数返回值

```
1  static int acl_permission_check
2      (struct inode *inode, int mask)
3  {
4    unsigned int mode = inode->i_mode;
5
6    if (likely(uid_eq(current_fsuid(), inode->i_uid)))
7      mode >>= 6;
8    else if (in_group_p(inode->i_gid))
9      mode >>= 3;
10
11   if ((mask & ~mode &
12        (MAY_READ | MAY_WRITE | MAY_EXEC)) == 0)
13     return 0;
14   return -EACCES;
15 }
```

例子 (2)

第二步: 找到控制节点

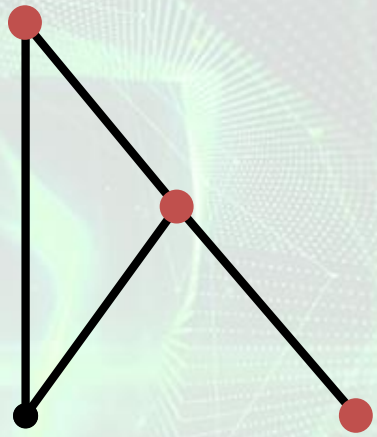
```
1  static int acl_permission_check
2      (struct inode *inode, int mask)
3  {
4    unsigned int mode = inode->i_mode;
5
6    if (likely(uid_eq(current_fsuid(), inode->i_uid)))
7      mode >>= 6;
8    else if (in_group_p(inode->i_gid))
9      mode >>= 3;
10
11   if ((mask & ~mode &
12        (MAY_READ | MAY_WRITE | MAY_EXEC)) == 0)
13     return 0;
14   return -EACCES;
15 }
```

例子 (3)

第二步: 找到控制节点

需考虑前序必经节点（DOMINATORS）

```
if (condition1 || condition2)
    return 0;
else
    return -EACCESS;
```



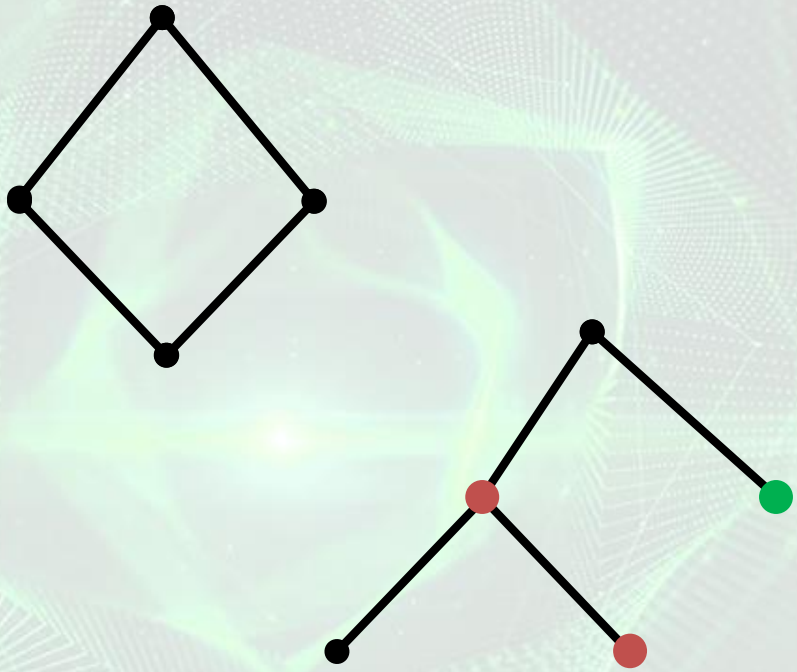
例子 (4)

第二步: 找到控制节点

避免爆炸

```
if (uid_eq)
    mode >> 6;
else
    mode >> 3;

if (condition)
    return -EINVAL;
```



例子 (5)

第三步: 后向切片

```
1  static int acl_permission_check
2      (struct inode *inode, int mask)
3  {
4    unsigned int mode = inode->i_mode;
5
6    if (likely(uid_eq(current_fsuid(), inode->i_uid)))
7      mode >>= 6;
8    else if (in_group_p(inode->i_gid))
9      mode >>= 3;
10
11   if ((mask & ~mode &
12        (MAY_READ | MAY_WRITE | MAY_EXEC)) == 0)
13     return 0;
14   return -EACCES;
15 }
```

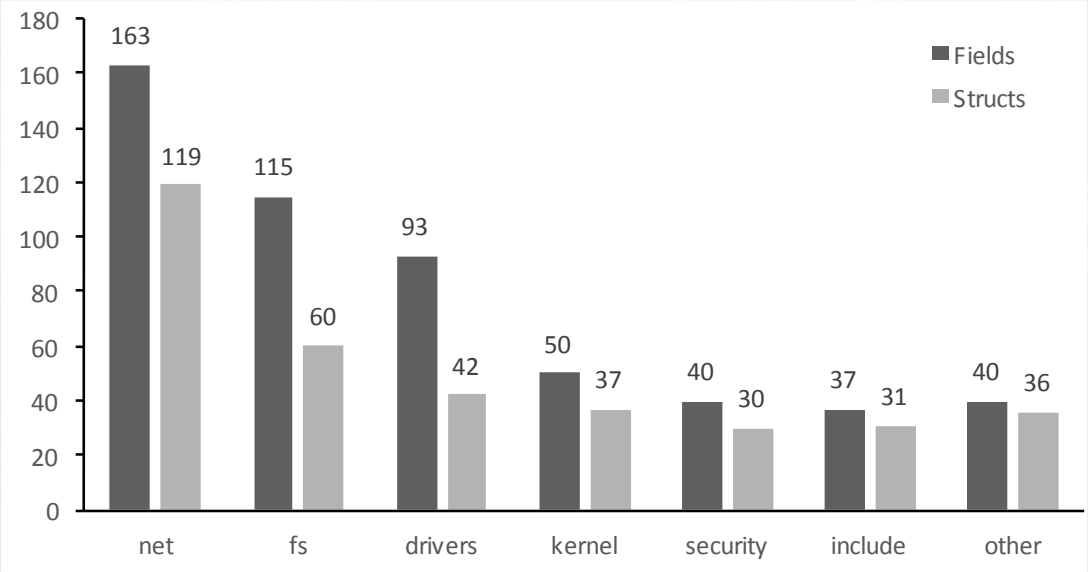
完整性

为保证完整性，还需

- 递归地进行数据和控制依赖的分析
- 考虑指针

分析结果

- NEXUS 9 ANDROID KERNEL
- 3个返回值：EACCESS，EPERM，EROFS
- 634个数据结构中的1240个域
- 279个全局变量



小节

分析过程利用到了

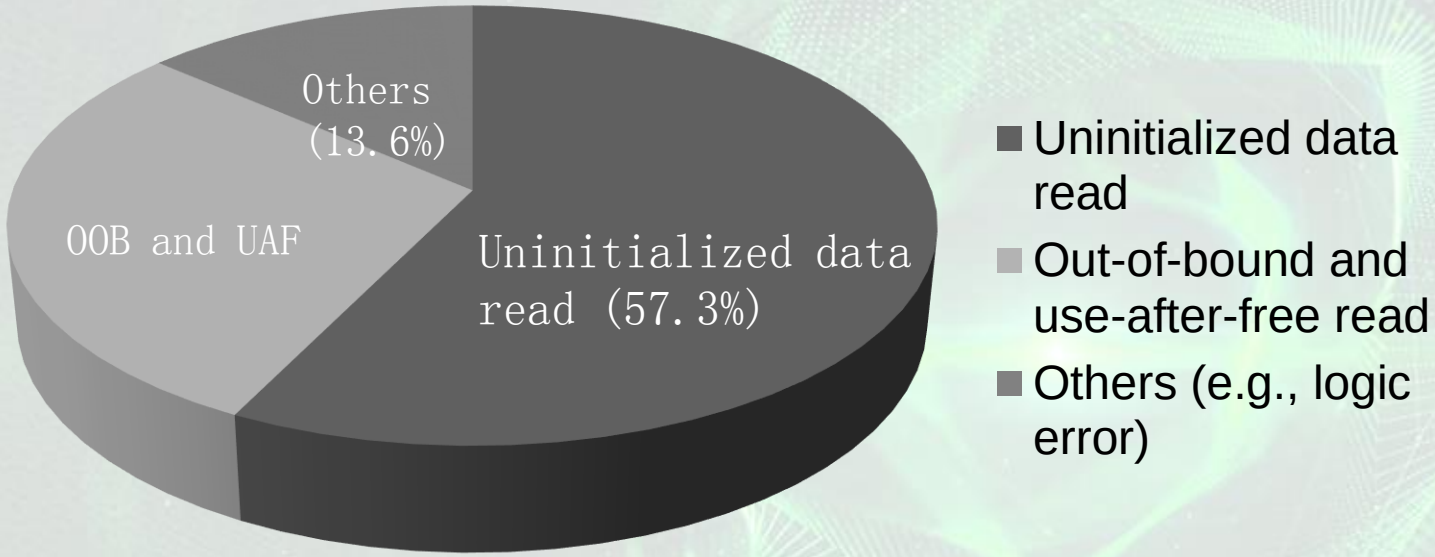
- 数据流分析：寻找可能的返回值
- 控制流分析：寻找控制节点
- 程序切片
- 别名分析：数据结构间的指向和包含关系

应用案例 (2)

自动化识别并修复内核中由于未初始化变量造成的信息泄漏 [CCS 16]

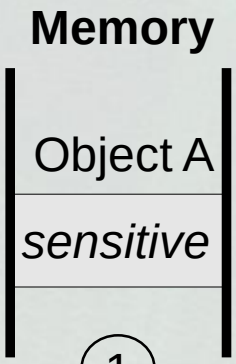
动机

- 内核中的信息泄漏漏洞，尤其是未初始化数据造成的泄漏越来越普遍

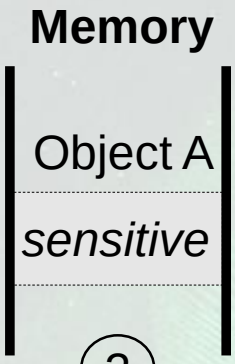


2013年以来LINUX内核中信息泄漏漏洞的分类

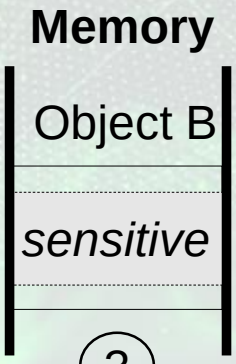
例子



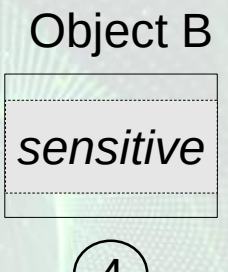
User A allocates object A and writes "*sensitive*" in to it



User A deallocates object A; "*sensitive*" is not cleared



User B allocates object B without Initialization; "*sensitive*" kept



User B reads and discloses Object B; "*sensitive*" leaked!

问题

如何尽可能精确地找到所有该类型漏洞？

- 全面性：不能有漏报
- 精确性：**尽可能降低误报**

解决方案

该类漏洞需要同时满足两个条件

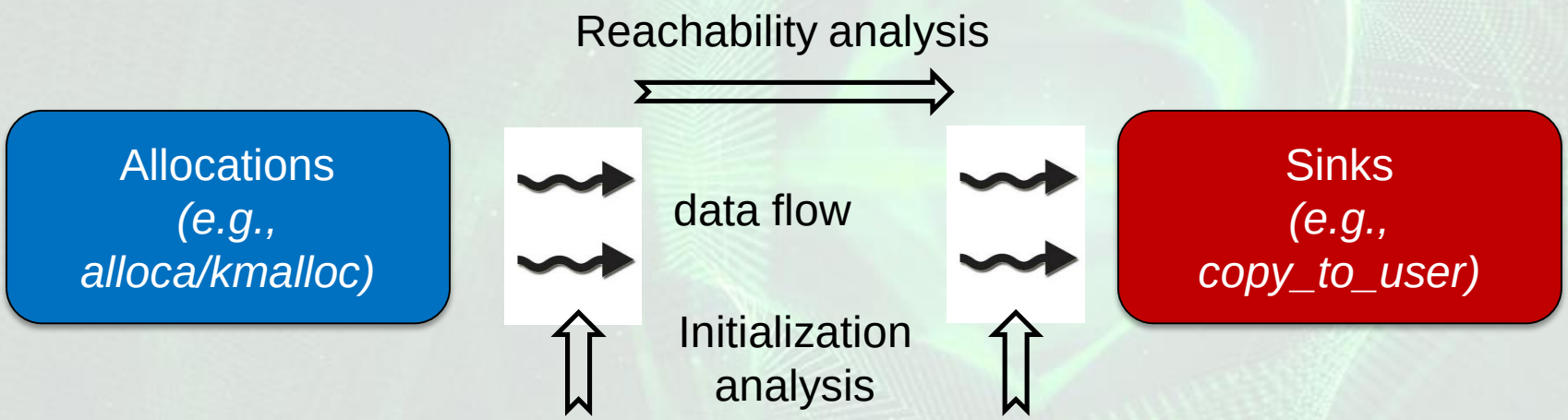
- 未初始化
- 可泄漏

适合使用静态污点分析

- SOURCE：内存分配
- SINK：泄漏点（COPY_TO_USER, NETWORK）
- SANITIZER：赋值语句（初始化）

UniSan

- 字节（BYTE）粒度的静态污点分析
- 考虑函数调用（INTER PROCEDURAL）
- 考虑调用的上下文（CONTEXT SENSITIVE）



分析结果 (1)

- LINUX KERNEL: X86_64, AARCH64

Arch	Module	Static Alloca	Dyn. Alloca	Static Malloc	Dyn. Malloc	Unsafe Alloca	Unsafe Malloc	Percent
X86_64	2,152	17,854	24	1,768	1,161	1,493	386	**9.0%**
AArch64	2,030	15,596	32	1,790	1,233	1,485	451	**10.3%**

Arch	Disable reachability analysis (# unsafe)	Disable initialization analysis (# unsafe)
X86_64	14,094 (**78.8%**)	3,380 (**18.9%**)
AArch64	11,209 (**71.7%**)	2,961 (**18.9%**)

分析结果 (2)

- 手工分析了350个报告的不安全内存分配
- 确认了19个漏洞

两类常见的成因

- 开发人员忘记进行初始化
- 编译器为了对齐而加入的PADDING

PADDING (1)

```
struct usbdevfs_connectinfo {
    unsigned int devnum;
    unsigned char slow;
    /* 3-bytes padding
       for alignment */
};
```

```
/* both fields (5 bytes) are initialized */
struct usbdevfs_connectinfo ci = {
    .devnum = ps->dev->devnum,
    .slow = ps->dev->speed == USB_SPEED_LOW
};
/* leaking 3-byte uninitialized padding bytes
     sizeof(ci) = 8 */
copy_to_user(arg, &ci, sizeof(ci));
```

PADDING (2)

```
struct usbdevfs_connectinfo {
    unsigned int devnum;
    unsigned char slow;
    /* 3-bytes padding
       for alignment */
};
```

```
/* copy initializer */
struct usbdevfs_connectinfo ci1, ci2;
memset(&ci2, 0, sizeof(ci2));
ci1 = ci2;

/* default initializer */
static struct usbdevfs_connectinfo ci1;

/* partial initializer */
static struct usbdevfs_connectinfo ci1 = {0};
```

小节

分析过程利用到了

- 静态污点分析
- 类型分析：数据结构各个域的大小

其他案例

其他案例

基于源代码的符号执行

- 自动挖掘文件系统的中的BUG：JUXTA [SOSP 15]
- 挖掘API的不正确使用：APISAN [SECURITY 16]

静态插装

- 修复漏洞：DANGNULL [NDSS 15]，CAVER [SECURITY 15]，UNISAN [CCS 16]
- 防御攻击：KENALI [NDSS 16]，VTRUST [NDSS 16]

展望及讨论

基于源代码的自动漏洞挖掘

随着开源软件的流行（例如ANDROID），基于源代码的自动漏洞挖掘将变得更加常见和强大

- RID [ASPLOS 16]
- DR. CHECKER [SECURITY 17]

常用的分析技术

虽然针对的目标不同，但基本的分析技术是通用的

- 控制流分析
- 数据流 / 污点分析
- 指针别名分析
- 符号执行

常用的套路

虽然针对的目标不同，但基本的套路是通用的

- 分析并提取已知漏洞的特征
- 设计特征检测方法
- 应用到目标对象上

资源

- NEXUS 9 KERNEL HTTPS://GITHUB.COM/SSLAB-GATECH/KENALI-KERNEL
- LINUX 4.9 KERNEL HTTPS://GITHUB.COM/CHENGYUSONG/LLL-49
- KERNEL ANALYZER HTTPS://GITHUB.COM/SSLAB-GATECH/KERNEL-ANALYZER
- UNISAN HTTPS://GITHUB.COM/SSLAB-GATECH/UNISAN
- JUXTA HTTPS://GITHUB.COM/SSLAB-GATECH/JUXTA
- APISAN HTTPS://GITHUB.COM/SSLAB-GATECH/APISAN

谢　谢