## SLAKE工作的总结

关注的漏洞模式：漏洞提供一个内核堆上的 overwrite primitive

对这种漏洞模式的利用方式：分配一个含有函数指针的结构体，通过漏洞的overwrite primitive修改函数指针，再调用该函数指针

为了寻找可用于这种利用模式的结构体，SLAKE的工作可以看作是对所有内核结构体进行了三次过滤：

1. 过滤出含有函数指针的结构体
2. 过滤出可以被分配的结构体（能够找到分配这个结构体的系统调用序列）
3. 过滤出包含的函数指针能够被调用的结构体（能够找到调用结构体内指针的系统调用序列）

显然，能够通过以上三次过滤的结构体就同时具有以下三种性质

1. 包含至少一个函数指针
2. 结构体可以被分配（已知分配它的系统调用序列）
3. 包含的函数指针可以被调用（已知调用其函数指针的系统调用序列）

即SLAKE找到的结构体是全部是同时具有这三种性质的结构体。

## SLAKE工作的缺陷

1. 只关注一种内核漏洞模式，对于这一种漏洞利用模式也只关注一种利用方式，从而SLAKE找到的这些结构体（和系统调用序列）适用范围非常窄：只有很少的内核漏洞可以提供SLAKE需要的那种 overwrite primitive

> （1）内核堆产生漏洞时，这个漏洞能够提供的能力，大多数时候和产生漏洞的的结构体（如，UAF悬空指针产生前指向的结构体，以下称漏洞结构体）的执行流高度相关，而大多数内核结构体不会直接从用户空间读取数据写入堆中；（2）在一些漏洞利用场景中，对于漏洞结构体的理解，即开发漏洞结构体的能力是漏洞利用的关键

2. SLAKE在对内核中所有结构体进行三次过滤的过程中，**忽略了大量已经发现的信息，这些信息可能对其他形式的漏洞利用非常有用，但由于不符合SLAKE关注的那种漏洞利用模式而被忽略**

例如，存在某个结构体StructA，包含至少一个函数指针，且能够发现调用其函数指针的执行流（满足SLAKE中过滤1和3的条件），但找不到分配该结构体的执行流（不满足过滤2）。按照SLAKE的思路，这样的结构体不符合SLAKE期望的漏洞利用模式，从而已经发现的关于这个结构体内含有函数指针及其调用方式的信息就会被忽略。若存在这样的情形：某个OOB漏洞提供了覆写StructA中函数指针的能力，则被SLAKE忽略的这样的信息在利用中就是有用的，可以直接实现控制流劫持。

在SLAKE构建数据库的过程中，类似这样的信息会大量被忽略（只有同时满足过滤1、2和3的结构体才会被记录），大量对漏洞利用可能有用的信息尽管被发现但不会被记录

# 我的思路

首先从SLAKE的工作出发，即只关注由堆漏洞导致控制流劫持的利用方式，我采用以下实现。

使用类似SLAKE实现方式，分别考量每个内核结构体具有的以下三种性质

1. 是否包含函数指针
2. 函数指针能否被调用（能否发现调用该函数指针的系统调用序列）
3. 结构体能否被分配（能否发现分配该结构体的系统调用序列）

但与SLAKE不同的是，我不期望找到的结构体同时具有这三种性质，只要求能具有部分性质即可，例如，发现某个结构体具有函数指针，也能够发现调用函数指针的系统调用序列，但找不到分配该结构体的系统调用，那么就只记录该结构体包含的函数指针信息，和触发函数指针调用方式的信息。这样的信息在诸如上一部分举例中的漏洞模式下就有用。

如果能够对尽可能多的结构体构建尽可能详尽的此类信息，就可以期望SLAKE发现的结构体和执行流，是这种思路下发现信息的子集，且在设计上可以解决SLAKE无法利用的第一类漏洞模式（漏洞仅允许写特定的结构体，如CVE-2017-1000112等多个漏洞）。

此外，将这三种性质分别考量还可能提供除控制流劫持这一种利用方式以外的有用信息，如，某个结构体不具有函数指针但发现了分配它的系统调用，这样的信息可以用来调整特定cache中的堆块布局；或某个结构体具有函数指针（但无法调用）且发现了分配它的系统调用，若漏洞提供read primitive则可用用来绕过KASLR等等。

## 进一步推广

在不明显增加实现难度和时间复杂度的前提下，对SLAKE的实现方式稍加修改，就可以发现其他有用的信息，如发现结构体中包含的函数指针的同时，发现其中的结构体指针（堆指针）；在关注kmalloc()函数调用的同时关注kfree()函数调用，记录结构体本身释放位点和其中堆指针释放位点。

采用这种方式对尽可能多的内核结构体进行分析，可以将发现的信息汇聚成形如下表的数据库

| 结构体名 | 函数指针 | 堆指针 | 分配 | 函数指针触发 | 堆指针释放 | ..... |
|---|---|---|---|---|---|---|
| StructA | 0,16 | NULL | fork() | exit(), NULL | NULL | |
| StructB | 24,48 | 0 | msgsnd() | NULL | msgrcv() | |
| StructC | 0 | 0,16 | NULL | open()->read() | NULL | |
| ... | | | | | | |

没有发现相关信息的表项留空即可，具有若干性质组合的结构体就能够用于不同的利用场景，如同时具有函数指针、分配、函数指针触发的结构体就可用于SLAKE关注的漏洞模式和利用模式；同时具有堆指针、分配、堆指针释放的结构体就能够可能用于任意地址释放。在面对一个漏洞时，若已知需要采取的利用思路，可用用这样的数据库查询可能的实现方式。

另一方面，如果在面对一个漏洞时缺乏利用思路，如仅有一个syzkaller报告的对于StructA若干偏移处做UAF写操作的bug，通过查询这样的数据库可以查看通过此UAF写操作可以直接实现的能力，可能能够为完成利用提供思路，或将当前的bug进行转换从而向完成利用的方向更进一步（如CVE-2021-26708的利用），或是在完成利用的基础上利用额外的信息实现防御机制绕过。

## 另一个推广

关于UAF漏洞利用的一个不成熟的想法：

UAF漏洞的总是伴随着悬空指针的产生，悬空指针大多数时候会和一个特定的内核结构体类型绑定（即悬空指针指向的结构体类型），利用UAF漏洞的一般思路不外乎两种：（1）UAF提供读写chunk的能力，此时通过分配一个结构体在UAF内存区域上，利用UAF漏洞提供的读写能力实现对该结构体内数据的非法修改（如修改一个函数指针，即SLAKE关注的模式）；（2）UAF指向的结构体本身具有利用价值（如自身就包含一个可被调用的函数指针），此时堆喷射数据进入该UAF结构体的位置修改若干数据，再利用该UAF结构体自身的执行流完成利用（如堆喷射数据修改UAF结构体中的函数指针再调用该指针）

对于方式（2），从用户空间堆喷射数据进入内核堆，已经有比较通用的技巧（`setxattr()`+`userfaultfd`，或利用`struct msg_msg`等），Chen, Y., Lin, Z., & Xing, X. (2020). A Systematic Study of Elastic Objects in Kernel Exploitation.这篇文章中对于利用可变长内核结构体完成堆喷射的方法也做了进一步挖掘。在一定程度上，内核堆喷射技术已经有了比较完备而现成的技巧来实现。

对于方式（1），如何将UAF漏洞理解很好地抽象成primitive目前似乎缺乏研究，这一点在SLAKE的文章未来展望部分也提到了。在有些情况下理解UAF漏洞本身提供的primitive是完成漏洞利用的核心（如CVE-2017-17052）。这个问题似乎也可以通过对每个结构体构建数据库解决，即关注系统调用对结构体中数值型成员变量的影响（如，发现fork()能使StructA中refcounter增加1）。如果能够积累足够多这样的信息，一旦UAF发生在某个特定结构体上，就可以立刻找出控制其中若干成员变量的方法，即实现UAF write的方法。

同时，积累这样的信息对于上述方式（2）也有帮助。上述提到到目前实现堆喷射堆方式具有一定堆限制，如无法实现对非通用cache中的堆块进行UAF写，或当内核禁用userfaultfd和system V通信，或需要对较小对堆块进行UAF写时（`setxattr()`+`userfaultfd`和`struct msg_msg`的方式都有写入堆块大小的下限）。找寻对内核结构体中若干成员变量的控制方法，结合可能存在的实现该内核结构体分配的系统调用序列，可以实现等同于堆喷射的效果，如以下情形：需写kmalloc-64的20字节偏移处的一个字节，已知StructA可被分配到kmalloc-64中，且已知控制StructA内20字节偏移处的成员变量堆方法，则可利用StructA完成“堆喷射”。

> 一些比较容易发现的模式可以被整合上一部分所构想的数据库中，如在解析结构体各成员变量时除了关注函数指针和堆指针外，关注counter型变量并追踪增加counter的系统调用。