疑难bug

malloc里的东西如果是临时变量，最好assert一下。
malloc和free要统计。
注意对象的析构。

改bug思路：
猜测（大量的经验，看bbs）->验证（快速理解代码，看大量的代码）

共享内存的原理：
http://www.ibm.com/developerworks/cn/linux/l-ipc/part5/index1.html

发现踩内存，然后都是0，怀疑：memset
应用层踩内核，只有一个途径：pkring

1.

http://www.ibm.com/developerworks/cn/linux/l-reent.html
fxq 可充入函数导致死锁，以及CString的浅复制导致的踩内存。
由于调用malloc的时候产生了段错误，然后调信号处理函数保存栈信息，里面调用了fopen打日志，fopen会调malloc，而malloc有内部全局结构，会加锁，不可重入，导致在一个线程里面两次加锁出不来了，因此死锁。解决方法： 改用write来写文件。
由于malloc里面会维护一个全局链表，当出现堆上分配的内存被踩的时候就会导致再去malloc或free时从链表里面读出一些非法的地址，导致段错误。
可以看到cstring的=运算符传的是指针引用，不是拷贝一个副本。线程一压入队列，调用copyseesion拷贝了一个指针进队列，线程二弹出session出队列，调用copyseesion又拷了一个csCatchPath的指针副本。如果这时候生产线程（线程一）正在释放Cmd_c里面的引用，消费线程正在使用csCatchPath的指针，那么就产生了冲突，把指针指向的内存写坏了，导致malloc内部结构损坏。
修改方法：使用另外一个operator函数，传入const char*字符串，该函数会拷贝副本(Gebuffer(0))。
fwlog就没有这个问题，因为是消费者用完释放，不是生产者，这是一个好的教训。

2.

多线程的fwlog，某一个线程（recorder）调用exit，导致存放线程共享的一个vector调用了析构函数，然后析构函数会调用delete释放所有元素，这时另一个线程（receiver）还没有退出，刚好在执行push_back，然后刚好内存不够需要扩展内存调用deallocate，就会造成double free了。
a) 不要在出core的时候，调用open_dir这些不可重入的函数，不然可能会死锁，CPU100%
b) 如果是malloc的内存错误，可以使用valgrind
c) 如果内存踩的很厉害，分析core就没什么价值了
d) 如果可以找到重现的方法，问题就好解决了
e) 可以试着从现场入手，修改代码缩小范围，就可以更容易出core
f) 让异常早点退出，core的信息就更准确，比如踩内存就在所有可疑点做强制检查（assert）
g) 针对工具的特性去修改代码，比如valgrind能识别非法读写，所以就要尽量多读写关心的内存区域。
exit会调用全局对象的析构函数，然后因为几个线程是new出来的，所以不会被析构掉，会继续跑。然后因为几个线程共享的一个vector是全局的，所以就会被析构掉，造成重复free。exit是这样，pthread_exit/ExitThread也是一样的。C++语言利用类似atexit一类的机制，注册了全局对象的析构函数作为钩子，保证调用exit时，所有全局对象都能得到析构。

3.

父进程初始化一块共享内存，再分配多个子进程，此时多进程使用这块共享内存，若父进程不幸死掉了，但其他子进程还活着，此时若再次启动父进程（可能是被软狗拉起的），父进程重新初始化共享内存，原有子进程仍然能访问这块新共享内存，新共享内存里有指向自己内部成员地址的指针（本bug是有链表头，初始化后指向了自己），子进程访问这种指针时就访问了非法内存。

4.

加载，退出，库更新，这些都是边界条件，必须得注意啊

5.

假如某个数据进程A正在使用锁，另外一个数据进程B半夜鸡叫重启，初始化资源就会把正在使用的锁释放了，这时候其他的数据进程C就有可能跑进去把共享内存破坏掉，等A再接着读链表数据的时候，就会发生段错误。共享内存锁的初始化应该由父进程完成，子进程继承，否则keep-alive每次重启数据进程会导致正在使用的锁被初始化，共享内存的内容被其他进程踩坏。
锁都是要进入临界区的代码才会有用，都是不管是否以及锁了，初始化还是能成功的，如果重新初始化锁，之前锁住的资源就没有保护了。

6.

把里面的动态分配改为静态，修改代码，缩小范围
注释代码，缩小范围
通过查看模块使用的物理内存，发现每几秒，增大4KB，猜测有4KB的内存块泄露
确定是在snmp库中泄露后，在库代码中添加日志，判断其最主要的业务流程。并审查流程中的代码，是否有泄露问题。
为每个动态内存块添加计数变量，并用日志打出来，查看malloc和free的计数。通过此方法发现了少量内存泄露，但修改后，内存泄露仍存在。
找到3.9版本，使用gdb分别跟中两个版本的代码，看差别在哪里，一步一步跟了好久，最后发现了一处以前没有注意的到的调用逻辑，最后发现最主要的内存泄露在里面。

7.

代码中malloc了一块内存，但传入的大小实际为0，但malloc会成功，且返回的是堆中的一个没人使用的内存的指针。但当我下次malloc时，返回的起始地址也是第一次malloc返回的指针。就导致了两个指针指向了同一块内存。

8.

timeout_heap堆顶为NULL，其他字段全部正常，包括malloc内存块头部，无踩内存迹象。
signal 7是你环境在运行ad_appd时替换库文件引起的
只踩单个指针，不是memcpy之类的函数越界造成，需要重点关注数组成员赋值，下标没检查的情况。
heap_st的bases数组某个元素出问题，能访问bases的只有heap_push和heap_pop两个接口 ，审了该接口的代码，不存在产生非法指针的情况。

9.

局部对象，在退出块之后就释放了，但是还有个退出的逻辑。在程序退出时，会调用CRuleresult类的析构函数（如下)释放成员m_list中为char* rulesid申请的空间 ，两次释放函数的调用造成了空间的重复释放

10.

因为没有对第1步接受的数据包头做检查。试想，若现在来了一个非法数据包，也就是说没有遵循约定的协议过来的数据，那么，有可能出现这样的包头，它的nSize值小于XHDR_SIZE.收到这样的一个包头后，它在第2步准备的收包的内存就会小于等于一个包头大小XHDR

11.

技术原因：客户端创建多个网络套接字，并添加到libevent中进行统一管理，同时还把进行间通信的FD也添加到libevent中。当界面点击停止扫描，CGI会发送消息给后台守护进程，后台守护进程接收消息后，会停止当前的扫描，同时会把已经添加到libevent中的事件删掉：event_del() 。但是event_base_dispatch 函数里，会仍然访问那些已经触发了事件的event对象，这样就有可能访问之前释放掉的事件结构体，进而访问到非法内存。解决方法：释放的指针要置NULL。

12.

把一些函数参数的指针类型，从int*修改为long*，恰巧这个指针的用途是作为返回值使用的。也就是说修改了该指针指向的内存。本来是要修改一个4字节变量，结果往下再偏4个字节，这倒好，一下子修改了8个字节。而这多余的4个字节正是另一个指针的一部分。这个指针指向的内容是要释放的。现在这个指针的值已发生了变化，当然也就释放不了了。

13.

gdb调试可以利用内存不可读，不可写这些特性

14.

在循环体内申请内存，如果申请失败，不能直接返回错误，否则会导致之前申请的内存泄露。这里应该使用continue

15.

将RwIni的全局对象改为局部的对象，防止多个线程同时执行rwini.Load

16.

函数栈内不能够使用很大的局部变量.
在栈中分配大量内存，导致分配失败引起段错误
(gdb) disassemble waf_pwd_crack_update 可以看死在哪一条指令
ulimit -s　　可以看函数栈的最大值，如果是递归的函数，就可能会超过这个最大值了。

17.

Helgrind寻找内存中被多 个线程访问，而又没有一贯加锁的区域，这些区域往往是线程之间失去同步 的地方，而且会导致难以发掘的错误。

18.

EIP中保存CPU下次要执行的指令地址即调用完函数后要执行的指令地址； EBP中保存的是当前函数栈基地址，一直不变，即栈底； ESP 中保存的是当前函数调用栈顶
只有core看不出堆栈，才需要分析指令地址来判断堆栈。

19.

free出错：
　　a) 双重free
　　b) free的指针出错
　　c) free的指针写溢出
1) delete/free后需要将指针置为0,不使用delete,free释放非堆内存;
2) close关闭的文件描述符必须置为-1；
3) fclose,pclose关闭的FILE指针必须置为0;
4) CloseHandle关闭的句柄必须置为

20.

mmap会导致一次性分配大量的内存，比如mmap映射了100个文件没有释放，当向映射的文件中写入40M的内容时，就会一下子占用4G的内存。调用munmap函数时注意传入正确的地址。写文件时采用了映射内存的方式，调用mmap，但是写完文件调用的munmap因为传入的地址有误导致释放失败，该函数并没有判断返回值，所以并不知道没有释放成功。通过查看/proc/id/maps，看到进程映射了几十个文件。

21.

alarmd在将uint32_t改成uint64_t后出现一些很奇怪的逻辑错误
技术原因（简单描述）：由于alarmd中多个结构体使用memcmp函数去重载operator <运算符，在uint32_t改成uint64_t后，由于结构体对齐问题，结构体内部出现一些内存空洞，导致memcmp比较不是我们期望的值。
结构体定义要遵循先小后大原则，尽量避免出现内存空洞。
C++，构造函数和拷贝构造函数在有空洞的时候，要用memset。

22.

传给host处理函数，会修改字符串，所以要注意传进去的参数的是否是一个合法的指针（比如不是一个长度为1的指针，因为host处理函数会设置两个字符）。
如果环境一直在出core，就应该利用这个特点，比如你知道有几个类型的数据库表会导致出错，就逐一删掉数据表，就能定位了。

23.

进程B发送get消息到进程A，获取msg1。进程A收到进程B的获取消息，发送msg1到进程B。 为了保证消息可靠。进程A收到进程B的get消息，进程A开启循环定时器发送消息msg1，进程B收到A发送的msg1，回复一个ACK.进程A收到ACK后，关闭循环定时器。异常情况，进程B异常重启没有回复ACK,由发送get消息到进程A，进程A会又重开一个定时器，导致上一个定时器泄露。

24.

模块间异步消息通讯需处理模块退出而消息到达的异常 . 处理其他模块消息，并通过查找对应的session对象处理这个消息。即从静态成员函数，调用非静态成员函数。当消息到达，而对应的session已被销毁时，此调用处core。解决方法：析构函数时将此会话从等待消息的map中删除。

25.

在一个信号中断处理函数中，会暂时屏蔽该信号，直到处理函数退出。如果我们没有为信号处理函数指定mask，此时如果收到另一

个不同的信号，信号中断处理函数同样会被中断，转而执行新的信号中断处理函数。

26.

ldd查看dlp的库连接情况，指示dlp连接了libre2_compile.so，dlp的正则匹配依赖于动态库libdlpre2.so，这个库是由re2开源代码修改得来，而libre2_compile.so这个库也是由修改re2代码得来，于是怀疑是动态链接.libre2_compile.so和libdlpre2.so大多数外部接口名称都相同，导致程序运行时，有些函数调用（这里的正则的预编译函数）就连接到了libre2_compile.so，而匹配函数（由于dlp使用的匹配接口做了修改）则连接了libdlpre2.so。解决方法：于是后来干脆就把整个类的名称给修改掉了。

27.

修改后让fork出来的子进程调用setpgrp()使自己成员进程组组长（以后该子进程还要fork出wget,ping等子进程），然后父进程就可以使用kill(-pid, SIGKILL);杀整个子进程所在的进程组，使用SIGKILL是因为它是强制的，不允许被忽略，比SIGINT更可靠地杀进程。

28.

fwlog写入中间文件的host长度限制为（6*2046），midtable加载host的长度为（512），导致midtable栈溢出。从core可以看到bt都是乱码。

29.

综上所述，我们在设计类似需要应用程序从驱动获取信息的程序时，需要主要应用程序和驱动交互的时间，需要通过各种机制来保证驱动陷入内核的时间不能太久，否则可能会导致在某个CPU上长时间不发生调度。解决这类问题的通用方法是通过应用程序从驱动中多次获取，而不是一次获取完成，这样通过把一个耗时的获取过程分解为多个小的获取过程可以解决此类问题。

30.

c,d两种情况，按照公式1，应该都会得到正数，结果却是一正一负。造成这一问题的原因在于int32_t的表示范围不够用。因此换成double类型来表示时间差，结果正确。

31.

如果读写锁当前没有读者，也没有写者，那么写者可以立刻获得读写锁，否则它必须自旋在那里，直到没有任何写者或读者。如果读写锁没有写者，那么读者可以立即获得该读写锁，否则读者必须自旋在那里，直到写者释放该读写锁。
检测冲突并添加应该在同一锁内操作，先读锁判断再写锁添加等于没加锁，在多核下会导致添加多个节点。

```
fun()
{
    read_lock_bh
        find
    read_unlock_bh          <=  多个内核CPU都会跑到这里
    write_lock_bh
        add
    write_unlock_bh
}
```

32.

fork之后，子进程调用execl，失败应该退出，而不是返回，否则会存在两个同样的进程

33.

为了防止int_16和int_32比较的时候（变量union），高位是乱码导致的问题，struct这些变量都要初始化memset。

34.

先注册一个信号SIGINT，然后不停的在死循环中获得一个锁，并释放，中间插入一个sleep以增加互斥重叠的概率。
当Ctrl C之后，会触发回调，然后也会获取锁，若main函数的死循环已经获得了锁，就会死锁。
http://200.200.0.17/cms/supesite/?action-viewnews-itemid-2095

进程P持有锁lock1，然后被中断打断，软中断也去获取锁lock1，发生死锁。
解决方法：进程获取锁时，需要使用bh，防止在持有锁的过程中被软中断打断。

http://200.200.0.17/cms/supesite/?action-model-name-defect-itemid-604

在中断执行：list_for_each_entry_rcu
在ioctrl直接删除节点，没有执行call_rcu
http://200.200.0.17/cms/supesite/?action-model-name-defect-itemid-468

假如我们已经确定了不是因为真死锁导致宕机，而是因为在代码内部产生了死循环，那么我们将如何确定死循环代码的位置呢？
解决问题的有效办法是：使用内核提供的魔术键
a) 去掉Sangfordog死锁检测机制，这样当假死锁发生时不会被检测到而重启系统。
b) 修改内核魔术键代码，或者注册我们自己的魔术键也可以
c) 采用二分的办法对多个函数进行基数，这样依次类推下去，很对就会定位出在哪个函数的哪个部分产生了死循环。
http://200.200.0.17/cms/supesite/?action-model-name-creative-itemid-1808

fopen内部会用到CRT锁，当调用fopen一类的函数遇到信号时，fopen就将重入，导致死锁。
将fopen改为open，fprintf改为write;open/write这些函数是信号安全函数
http://200.200.0.17/cms/supesite/?action-viewnews-itemid-2793

read_lock_bh 会导致系统中断被禁用，而频繁的调用copy_to_user会导致发生缺页中断，因为此时中断被禁止，没有足够的内存页可以被使用，故调用失败。checklist3.5，在read_lock/read_lock_bh锁定区域内不得调用copy_to_user。对于这种把列表拷贝到应用层的情形，比较好的方法是，用kmalloc分配足够大的临时空间（一个最大的缓冲区大小或和用户空间大小一样），加锁，把内容全拷贝到这个临时空间，然后解锁，通过copy_to_user拷贝到用户空间。
http://200.200.0.17/cms/supesite/?action-viewnews-itemid-1677

35.

http://200.200.0.17/cms/supesite/?action-model-name-defect-itemid-1106
使用strace+lsof排查CPU占用问题：
背景：测试部阴阳边有个稳定性环境的设备，使用VPN测试工具进行并发测试、随机接入断开VPN连接，十几天之后停掉测试工具，发现
有个lmdlan线程一直占着10%左右的CPU。
a) strace -p pid查看线程的系统调用情况，发现都是以下输出在一直重复: time+select，但仅依靠此两个系统调用，还不能判断出是哪个线程。
b) 使用lsof -p pid -n查看线程使用的句柄。根据select中的句柄号，查出select对应的具体是什么资源的句柄。
I) 根据以上的资源路径，很容易得出此线程是VpnKernel线程。
II) 从select的范围，也可以看出select的范围有异常
c)
I) strace提供了查看时间戳的开关（-tt），使用strace -tt -p pid。观察strace的时间戳和select的时间间隔，结合代码看，此处 是计算 定时器的操作ScanAllTimer。而此函数能够消耗CPU的唯一原因就是定时器过多了，即定时器存在泄漏
II) 从select的范围，也可以看出select的范围有异常

36.

http://200.200.0.17/cms/supesite/?action-viewnews-itemid-1831
kmalloc刚好分配出1页内存，当出现踩内存时就相当于踩了下一个内存页，导致内核在不同地方出现宕机。
proc的钩子函数中，需确保输出不超过4K的长度，如果输出字符串长度不确定，需使用输出长度限制（比如：sprintf(buf, "%30.20s", str)可限制最多输出20个字符）。

37.

http://200.200.0.17/cms/supesite/?action-viewnews-itemid-1204
挖掘valgrind的高级功能。
valgrind –leak-check=full –undef-value-errors=no待测试程序，因为这会减少输出无用的信息。
高级的功能必须修改程序，参考它的使用手册中的4.6. Client Requests。
a) 以前一直认为valgrind对“假泄漏”是无能为力的，因为他的统计结果是到程序退出时才显示出来，而“假泄漏”的程序都是做得很完美，在程序退出时把所有对象释放了，这样valgrind的统计结果为零。发现了VALGRIND_DO_LEAK_CHECK这个函数，会在程序运行时把未释放的内存显示出来，这就可以解决问题了。可以写个信号处理函数，然后调用观察内存情况。
b) 我们常有这样的烦恼：某个变量肯定被踩了，但不知道是谁踩。valgrind对于栈变量或栈内存是无能为力的，它只对堆，即用new或malloc分配出来的内存作检测。
I)

```
struct config{
    int a;
    char pad1[10];
    int b;          <= 要检测的变量，前后填充内存，因为我们不知道内存是从上踩下来，还是从下踩上去
    char pad2[10];
    int c;
};
```

Ⅱ)
VALGRIND_MAKE_MEM_NOACCESS(cfg.pad1,10);　　<= 设置为不可写
VALGRIND_MAKE_MEM_NOACCESS(cfg.pad2,10);

38.

http://200.200.0.17/cms/supesite/?action-model-name-creative-itemid-11320
在 应用层设置内存也权限有个函数是mprotect。
http://blog.csdn.net/ustc_dylan/article/details/6941768
应注意的是， malloc 返回的内存区域通常并不与内存页面对齐，甚至在内存的大小是页大小整数倍的情况下也一样。如果您想保护从 malloc 获得的内存，您不得不分配一个更大的内存区域并在其中找到一个与页对齐的区间。
您可以选择使用 mmap 系统调用来绕过 malloc 并直接从 Linux 内核中分配页面对齐内存，先映射/dev/zero一个页面大小的内容，然后写第一个字节获取一个COW，然后就是页对齐了（原因请看在深信服工作->工作->mmap笔记）。

```
struct mem_t
{
	char i;
	char j;
	char reserved[1024 * 3 + 1022];		<= 对齐
	fuc func;				<= 保护的指针
	char reserved1[1024 * 3 + 1016];	<= 对齐
};
mprotect((void *)((unsigned long)(p) & (~(PAGESIZE - 1))), 1024, PROT_READ)
```

39.

http://200.200.0.17/cms/supesite/?action-model-name-creative-itemid-10037
变长结构体避免使用sizeof计算偏移

```
struct B
{
 int size;
 long data[0];
};
struct C
{
 long value;
 int size;
 char data[0];
};
 printf("B\t size:%lu offset:%lu\n", sizeof(struct B), offsetof(struct B, data[0]));  => B size:8 offset:8
 printf("C\t size:%lu offset:%lu\n", sizeof(struct C), offsetof(struct C, data[0]));  => C size:16 offset:12
```

正确用法：

```
a) struct B* b = (struct B*)((char*)a+offsetof(struct A, data[0]));
b) struct B* b = ((struct A*)a)->data;
```

40.

AF5.6bug:
a) 下发程序不能调用阻塞操作，如：acModule前台调用daemon，导致卡死
b) 字符串截断操作（把字符串数组某个位置置'\0'，最好随便把之后的所有字符也置'\0'）
如下程序就会导致栈溢出：

```
char s[] = "/0abcdsfd";
memset(s, 0, strlen(s));
s[0] = '-';
```