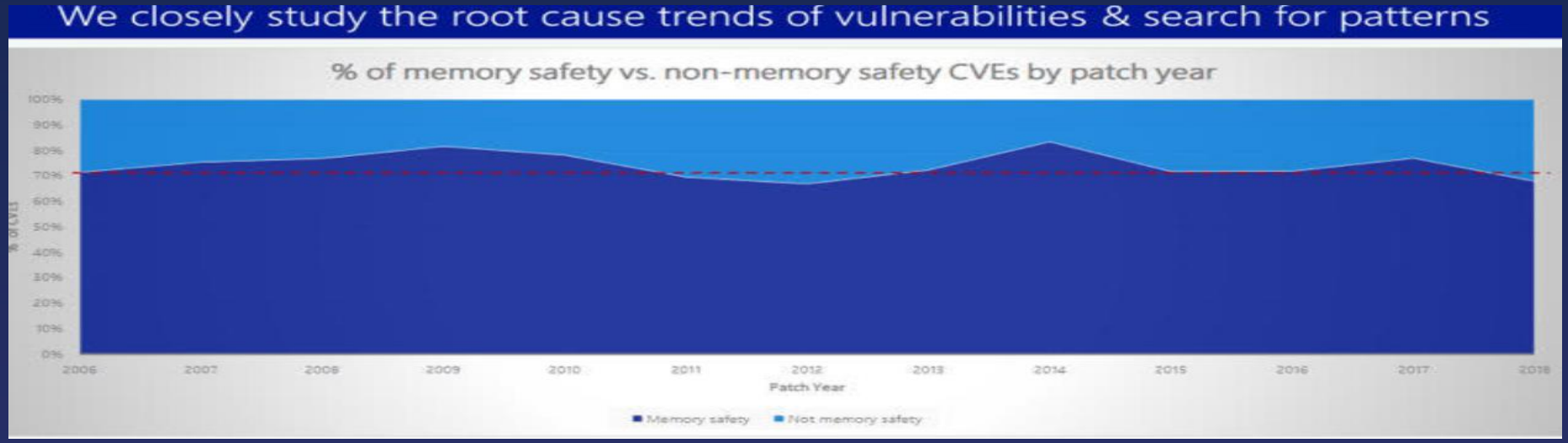
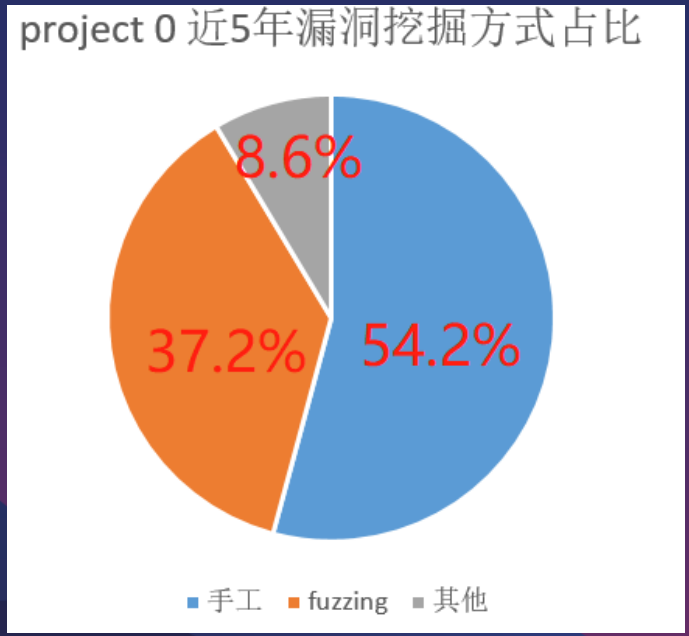

# Dumart fuzzer:
# Make the dumb fuzzer smart

陈振宇 个人安全研究员

# 背景




- 内存类漏洞占很大比例
- Fuzzing仍然是发现内存类漏洞的重要手段

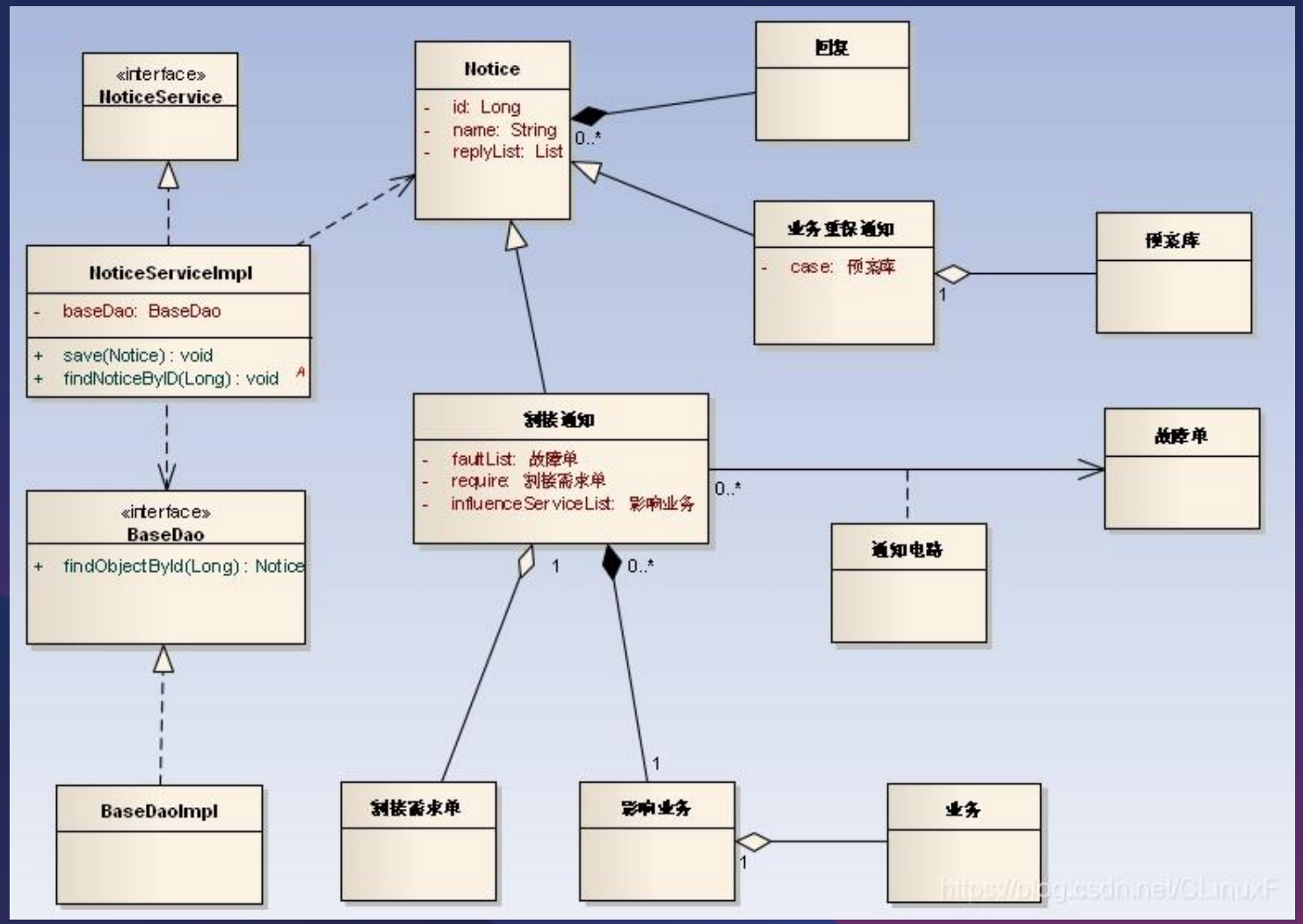
- 有源码fuzzing工具已经很完善
- 无源码fuzzing的方法缺点较多，一般无源码的Fuzzing也叫黑盒fuzz

- 有时不得不进行黑盒fuzz。如闭源项目。

# 背景：黑盒fuzz的困难

- 状态机
  - dlopen
  - qemu, unicorn
- 效率低
- 缺乏路径反馈
  - 没有路径反馈的一般称为dumb fuzz，反之会叫smart fuzz
- 没有地址消杀器
  - fuzzing without ASAN is a waste of CPU

# 状态机：是什么，为什么麻烦




- 所有CPU以及内存里的值
  - 指针，虚函数表等
  - 变量（内存）里保存的值
  - 某状态时，cpu寄存器的值

大型软件中，对象间有聚合，依赖，包含。所有对象都需要满足才能运行。要运行fuzz，有时即使有源码也是很复杂的事情 。

# 背景：如何能做一个最简单的黑盒fuzz

- 简单来说，需要2点：
- 1、fuzz就是不断地运行某一段代码每次运行，输入变异了的参数

- 2、要运行一段代码，需要准备好代码运行的上下文。这个上下文有时又叫状态机。

# 状态机：如何简单地获取状态机，提高效率

- 状态机天然在存在我们的测试环境里（业务可以正常运行）。
- 直接发送报文（效率低），只能从处理入口开始fuzz，崩溃检测慢

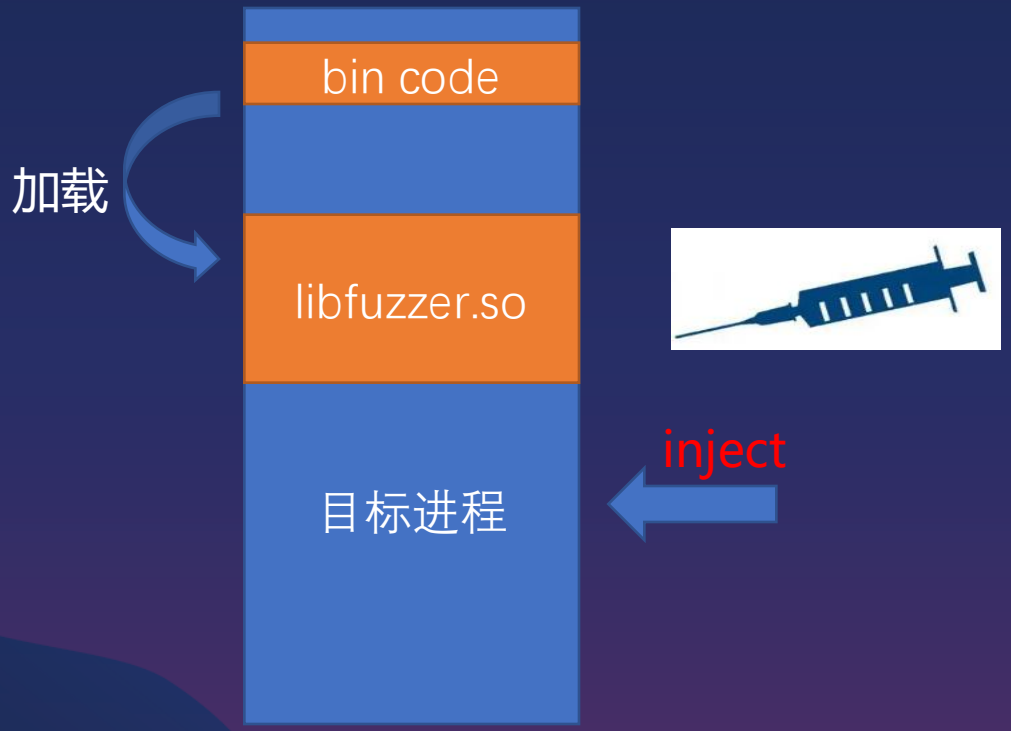
- 如果可以在正常运行的程序中注入我们的代码就可以获得这个状态机和不断运行被测试对象。

效率会大大提升。

——进程注入( https://github.com/gaffe23/linux-inject)

- 使用系统调用fork，保存系统状态
- 在新进程中可以选择任意函数，甚至任意片段进行fuzz

# 状态机：注入原理




- 是通过ptrace在目标进程写入一段二进制代码
- 二进制代码执行了__libc_dlopen_mode，把目标库加载进进程空间
- 恢复进程原本上下文

# 状态机: 为什么不在原进程fuzz

- 其实是可以的

- 但是:
- 一些嵌入式系统有看门狗，在原进程上fuzzing，可能由于响应慢等导致非漏洞的重启
- 需要启动一个新线程来做fuzz
  - 效率较低
  - 全局变量的访问

# Fuzzer: 快照，快照的作用

```
void start_fuzz(){

    while(1){
        if (count == 10000){
            show_status();
            count = 0;
            total_count ++ ;
            if (total_count % 1000000 == 0){
                printf("restore mem\n");
                restore_mem();
            }
        }

        input_size = get_one_corpu(input);
        input_size = mutate(input, input_size , PAYLOAD_LEN);
        //printf("input:%s\n",input);
        target_function(input, input_size);
        count ++;

        memset(input,0, input_size);
        reinit_chunks(); // restore chunks states

        //sleep(20);
    }
}
```

- Afl 有fork server，会不停地执行fork来运行一次测试对象
- Fork可以恢复运行的状态。但是时间开销较大。
  - Afl就有persistent mode减少fork开销。假定每次运行都是无状态的 。重复运行测试对象。

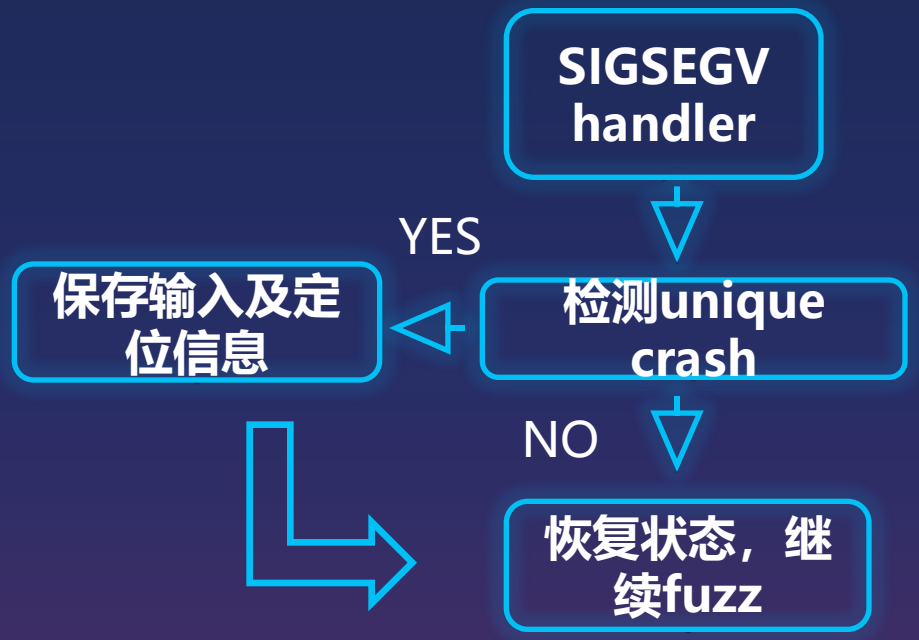
- Dumart fuzzer模仿persistent mode。保存快照。用于恢复。

# Fuzzer: 保存快照，怎么保存

- 内存:
  - 扫描进程的内存空间，保存目标的可写段
  - 所有堆内存
  - 不是一个变量一个变量地记录。整个段保存下来。

- 寄存器:
  - 在需要保存状态的地方设置断点，通过断点处理程序记录寄存器状态

# Fuzzer: 崩溃检测

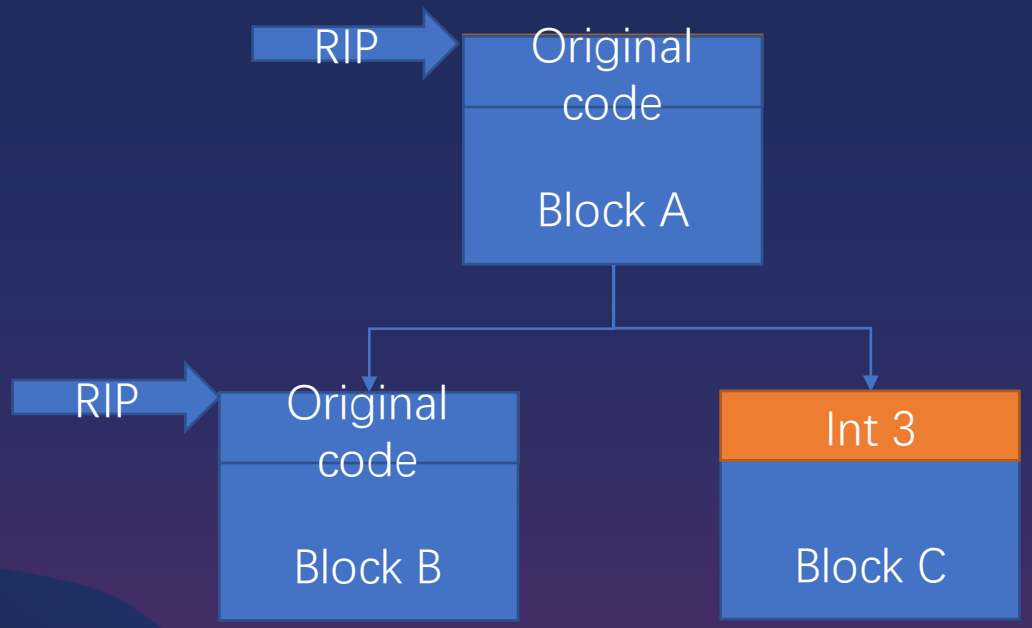



- 内存类漏洞崩溃时到底发生了什么?
  - 收到信号SIGSEGV
- 注册SIGSEGV信号处理程序，接管SIGSEGV信号
  - Unique crash detect：检查RIP是否唯一
  - Unique crash 则保存输入
  - 恢复上下文

# Fuzzer: 路径反馈是什么

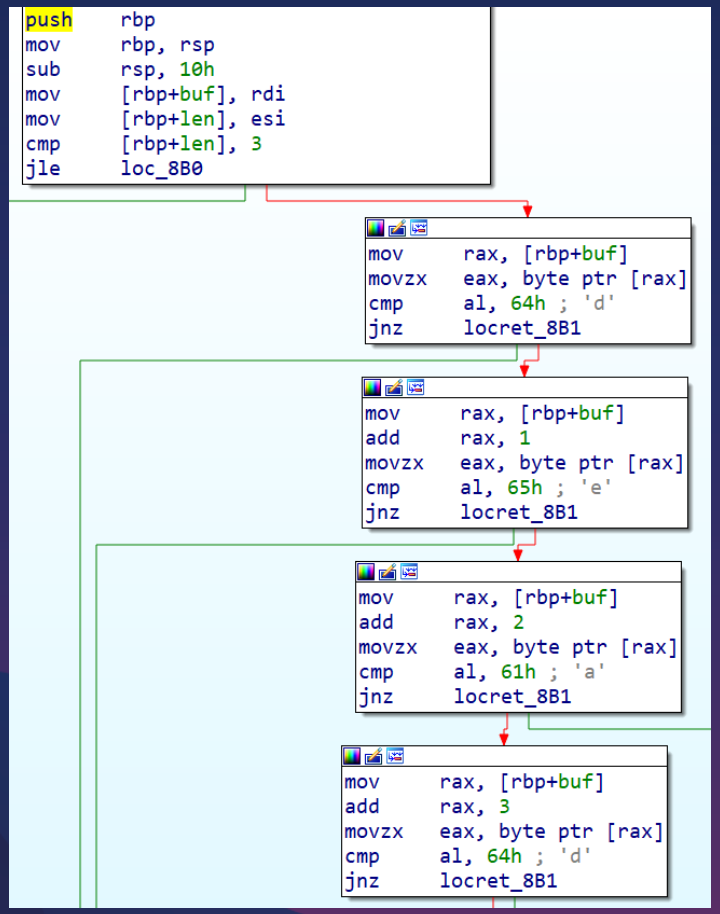
- 路径反馈可以让fuzz运行的更深
- 发现更多问题
- 这是smart fuzz和dumb fuzz最重要的区别

# Fuzzer: 路径反馈




- 通过ida脚本，把被测对象的所有block的地址记录下来
- 断点
  - 在每个代码块的起始地址改写为断点指令(int 3)
  - 注册信号处理程序
    - 触发新代码块的输入作为种子保存
    - 恢复原本的指令
    - 继续运行

# Fuzzer: 路径反馈




- 打断点
- 断点触发，
- 恢复代码

```
0x7f3cf4e0079a <test>:      push   rbp
0x7f3cf4e0079b <test+1>:    mov    rbp,rsp
0x7f3cf4e0079e <test+4>:    sub    rsp,0x10
0x7f3cf4e007a2 <test+8>:    mov    QWORD PTR [rbp-0x8],rdi
0x7f3cf4e007a6 <test+12>:   mov    DWORD PTR [rbp-0xc],esi
0x7f3cf4e007a9 <test+15>:   cmp    DWORD PTR [rbp-0xc],0x3
0x7f3cf4e007ad <test+19>:   jle    0x7f3cf4e008b0 <test+278>
0x7f3cf4e007b3 <test+25>:   mov    rax,QWORD PTR [rbp-0x8]
0x7f3cf4e007b7 <test+29>:   movzx  eax,BYTE PTR [rax]
0x7f3cf4e007ba <test+32>:   cmp    al,0x64
0x7f3cf4e007bc <test+34>:   jne    0x7f3cf4e008b1 <test+279>
0x7f3cf4e007c2 <test+40>:   int3
0x7f3cf4e007c3 <test+41>:   mov    eax,DWORD PTR [rbp-0x8]
0x7f3cf4e007c6 <test+44>:   add    rax,0x1
0x7f3cf4e007ca <test+48>:   movzx  eax,BYTE PTR [rax]
0x7f3cf4e007cd <test+51>:   cmp    al,0x65
0x7f3cf4e007cf <test+53>:   jne    0x7f3cf4e008b1 <test+279>
0x7f3cf4e007d5 <test+59>:   int3
0x7f3cf4e007d6 <test+60>:   mov    eax,DWORD PTR [rbp-0x8]
0x7f3cf4e007d9 <test+63>:   add    rax,0x2
0x7f3cf4e007dd <test+67>:   movzx  eax,BYTE PTR [rax]
0x7f3cf4e007e0 <test+70>:   cmp    al,0x61
0x7f3cf4e007e2 <test+72>:   jne    0x7f3cf4e008b1 <test+279>
0x7f3cf4e007e8 <test+78>:   int3
0x7f3cf4e007e9 <test+79>:   mov    eax,DWORD PTR [rbp-0x8]
0x7f3cf4e007ec <test+82>:   add    rax,0x3
0x7f3cf4e007f0 <test+86>:   movzx  eax,BYTE PTR [rax]
0x7f3cf4e007f3 <test+89>:   cmp    al,0x64
0x7f3cf4e007f5 <test+91>:   jne    0x7f3cf4e008b1 <test+279>
0x7f3cf4e007fb <test+97>:   int3
```

# ASAN：是什么

- 发生内存类漏洞时，程序是否一定有异常表现？
  - NO。如小量越界读写，只要不是读写发生在非法地址，程序是不会崩溃的。
  - UAF类型漏洞，程序崩溃现场，调用栈上很可能是没有漏洞的。因为这不是漏洞发生的第一现场。给定位问题带来困难。

ASAN，一种内存类型漏洞检测技术，在溢出，UAF等漏洞发生时，马上触发程序崩溃

# ASAN：编译器实现

```c
#include <stdlib.h>

int main(){

char* a = malloc(10);

a[0]='a';


return 0;

}
```

- 源码

```
main proc near

var_8= qword ptr -8

; __unwind {
push    rbp
mov     rbp, rsp
sub     rsp, 10h
mov     edi, 0Ah        ; size
call    _malloc
mov     [rbp+var_8], rax
mov     rax, [rbp+var_8]
mov     byte ptr [rax], 61h ; 'a'
mov     eax, 0
leave
retn
; } // starts at 64A
main endp
```

- 没有ASAN

```
; __unwind {
push    rbp
mov     rbp, rsp
sub     rsp, 10h
mov     edi, 0Ah        ; size
call    _malloc
mov     [rbp+var_8], rax
mov     rax, [rbp+var_8]
mov     rdx, rax
shr     rdx, 3
add     rdx, 7FFF8000h
movzx   edx, byte ptr [rdx]
test    dl, dl
setnz   cl
mov     rsi, rax
and     esi, 7
cmp     sil, dl
setnl   dl
and     edx, ecx
test    dl, dl
jz      short loc_884
```

```
mov     rdi, rax
call    __asan_report_store1
```

```
loc_884:
mov     rax, [rbp+var_8]
mov     byte ptr [rax], 61h ; 'a'
mov     eax, 0
leave
retn
```

- 开启ASAN

# 黑盒ASAN

- ASAN的编译器实现
  - 编译时插桩
  - Shadow memory，动态库支持

- 以上2个条件都是黑盒fuzz不具备的
- 所以需要实现一个带asan功能的堆，替换libc的堆

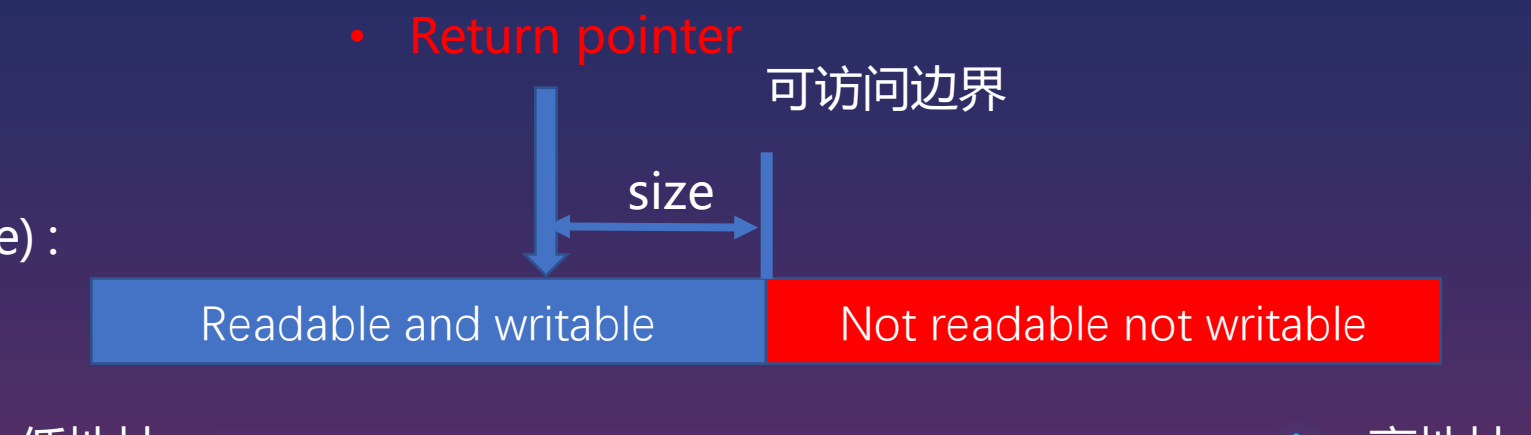
- 堆函数的替换方式，GOT表覆盖。

# 黑盒ASAN：上溢出检测

- 检测原理，页属性，读，写，执行。
- 准备2个相邻的页作为一个基本内存管理单元（chunk），其中一个是不可读写，用于作越界检测。

上溢出检测

- my_malloc(size) :



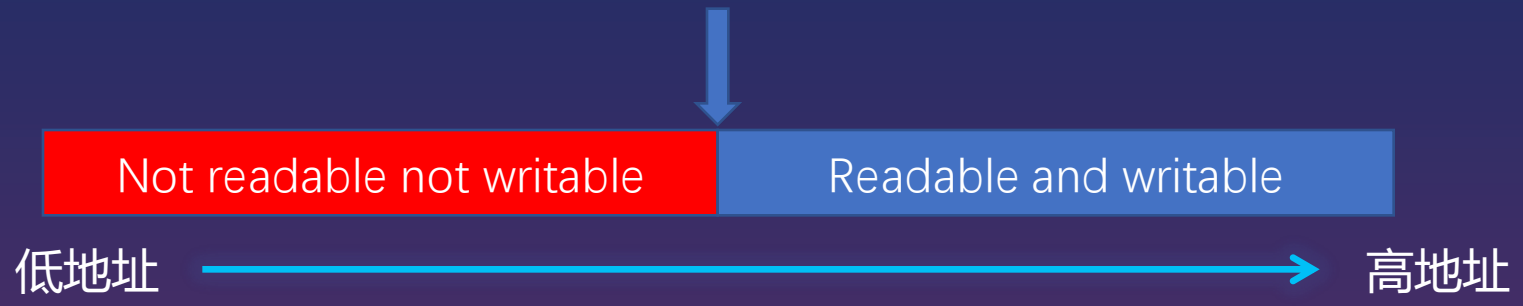

- 使用return pointer，如果访问超过size，会触发段错误

# 黑盒ASAN：下溢出检测

- 检测原理，页属性，读，写，执行。

- my_malloc(size) :
- 下溢出检测

- Return pointer

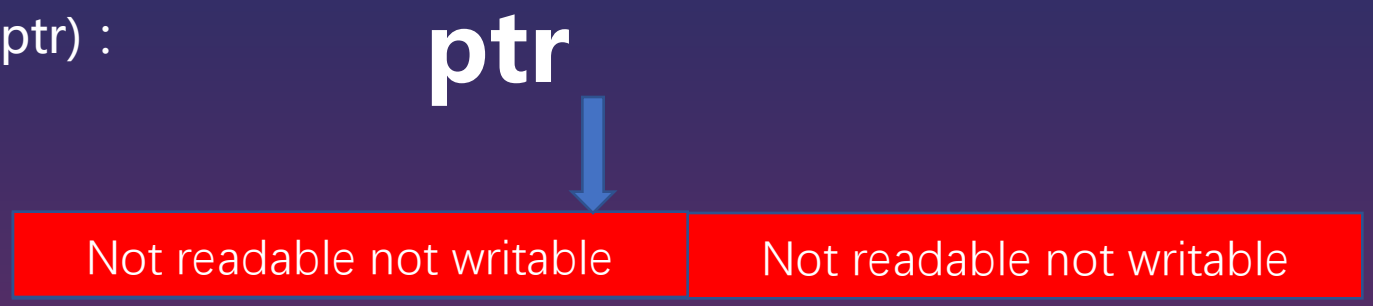



- 检测上溢出和下溢出的chunk随机提供

# 黑盒ASAN: UAF

- 运行一次被测片段时，chunk不需要重用
- 设置页不可读不可写
- 运行完成一次被测片段，重新初始化堆

- my_free(ptr)：




- 释放后再使用ptr会触发段错误

# 黑盒ASAN：chunk的管理

```
7fb7a97a9000-7fb7a97aa000 ---s 00000000 00:00 420498                     /dev/zero
7fb7a97aa000-7fb7a97ab000 rw-s 00001000 00:00 420498                     /dev/zero
7fb7a97ab000-7fb7a97ac000 ---s 00002000 00:00 420498                     /dev/zero
7fb7a97ac000-7fb7a97ad000 rw-s 00003000 00:00 420498                     /dev/zero
7fb7a97ad000-7fb7a97ae000 ---s 00004000 00:00 420498                     /dev/zero
7fb7a97ae000-7fb7a97af000 rw-s 00005000 00:00 420498                     /dev/zero
7fb7a97af000-7fb7a97b0000 ---s 00006000 00:00 420498                     /dev/zero
7fb7a97b0000-7fb7a97b1000 rw-s 00007000 00:00 420498                     /dev/zero
7fb7a97b1000-7fb7a97b2000 ---s 00008000 00:00 420498                     /dev/zero
7fb7a97b2000-7fb7a97b3000 rw-s 00009000 00:00 420498                     /dev/zero
7fb7a97b3000-7fb7a97b4000 ---s 0000a000 00:00 420498                     /dev/zero
7fb7a97b4000-7fb7a97b5000 rw-s 0000b000 00:00 420498                     /dev/zero
7fb7a97b5000-7fb7a97b6000 ---s 0000c000 00:00 420498                     /dev/zero
7fb7a97b6000-7fb7a97b7000 rw-s 0000d000 00:00 420498                     /dev/zero
7fb7a97b7000-7fb7a97b8000 ---s 0000e000 00:00 420498                     /dev/zero
7fb7a97b8000-7fb7a97b9000 rw-s 0000f000 00:00 420498                     /dev/zero
7fb7a97b9000-7fb7a97ba000 ---s 00010000 00:00 420498                     /dev/zero
7fb7a97ba000-7fb7a97bb000 rw-s 00011000 00:00 420498                     /dev/zero
7fb7a97bb000-7fb7a97bc000 ---s 00012000 00:00 420498                     /dev/zero
7fb7a97bc000-7fb7a97bd000 rw-s 00013000 00:00 420498                     /dev/zero
7fb7a97bd000-7fb7a97be000 ---s 00014000 00:00 420498                     /dev/zero
7fb7a97be000-7fb7a97bf000 rw-s 00015000 00:00 420498                     /dev/zero
7fb7a97bf000-7fb7a97c0000 ---s 00016000 00:00 420498                     /dev/zero
7fb7a97c0000-7fb7a97c1000 rw-s 00017000 00:00 420498                     /dev/zero
7fb7a97c1000-7fb7a97c2000 ---s 00018000 00:00 420498                     /dev/zero
7fb7a97c2000-7fb7a97c3000 rw-s 00019000 00:00 420498                     /dev/zero
7fb7a97c3000-7fb7a97c4000 ---s 0001a000 00:00 420498                     /dev/zero
7fb7a97c4000-7fb7a97c5000 rw-s 0001b000 00:00 420498                     /dev/zero
```

- libc为了复用，chunk使用链表

- 为了效率，这里chunk的管理使用数组。
- 数组每一项管理一个chunk。
- 一次分配一块连续很大的内存（知道基址）
- 设置页属性（可/不可读写）

- 这样保证了chunk是连续的
- 然后有一个管理结构（数组），对应每一个chunk
- 通过地址可以知道下标

- 不是我们定义chunk范围的指针，交给libc处理。

# 黑盒ASAN: double free

```
struct chunk
{
    enum State state;     // AVAILABLE or ALLOCED or free
    long address;
};
```

- 运行一次代码片段时，chunk是不会真正free的。
- 每个Chunk有一个管理结构。记录是分配状态还是释放状态。
- 在free(ptr)时，会检查这个状态是否已经是free。是则发生了double free。

# 黑盒ASAN:内存泄露

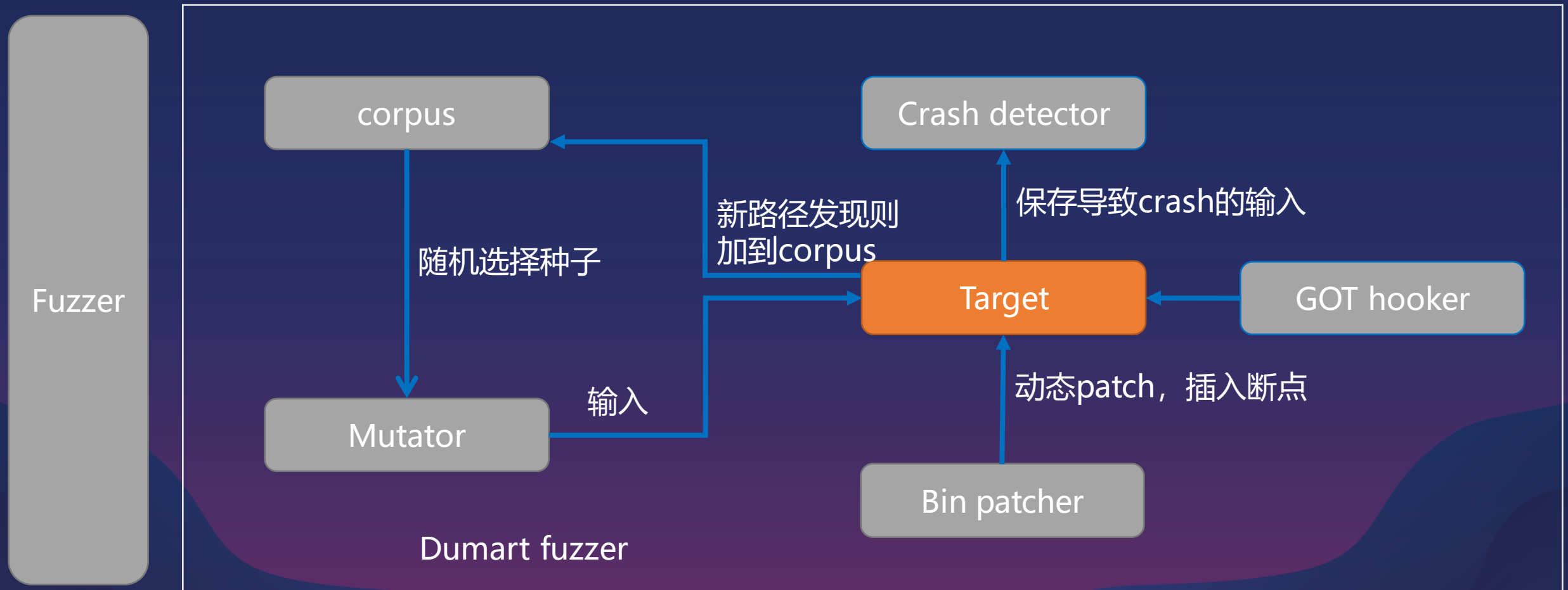
```
309  void start_fuzz(){
310
311      while(1){
312          if (count == 10000){
313              show_status();
314              count = 0;
315              total_count ++ ;
316              if (total_count % 1000000 == 0){
317                  printf("restore mem\n");
318                  restore_mem();
319              }
320          }
321
322          input_size = get_one_corpu(input);
323          input_size = mutate(input, input_size , PAYLOAD_LEN)
324          //printf("input:%s\n",input);
325          target_function(input, input_size);
326          count ++;
327
328          memset(input,0, input_size);
329          reinit_chunks(); // restore chunks states
330
331          //sleep(20);
332      }
333  }
```

- 检测原理，chunk在运行一次过程中，是没有真正释放的（只是设置页属性为不可读写）

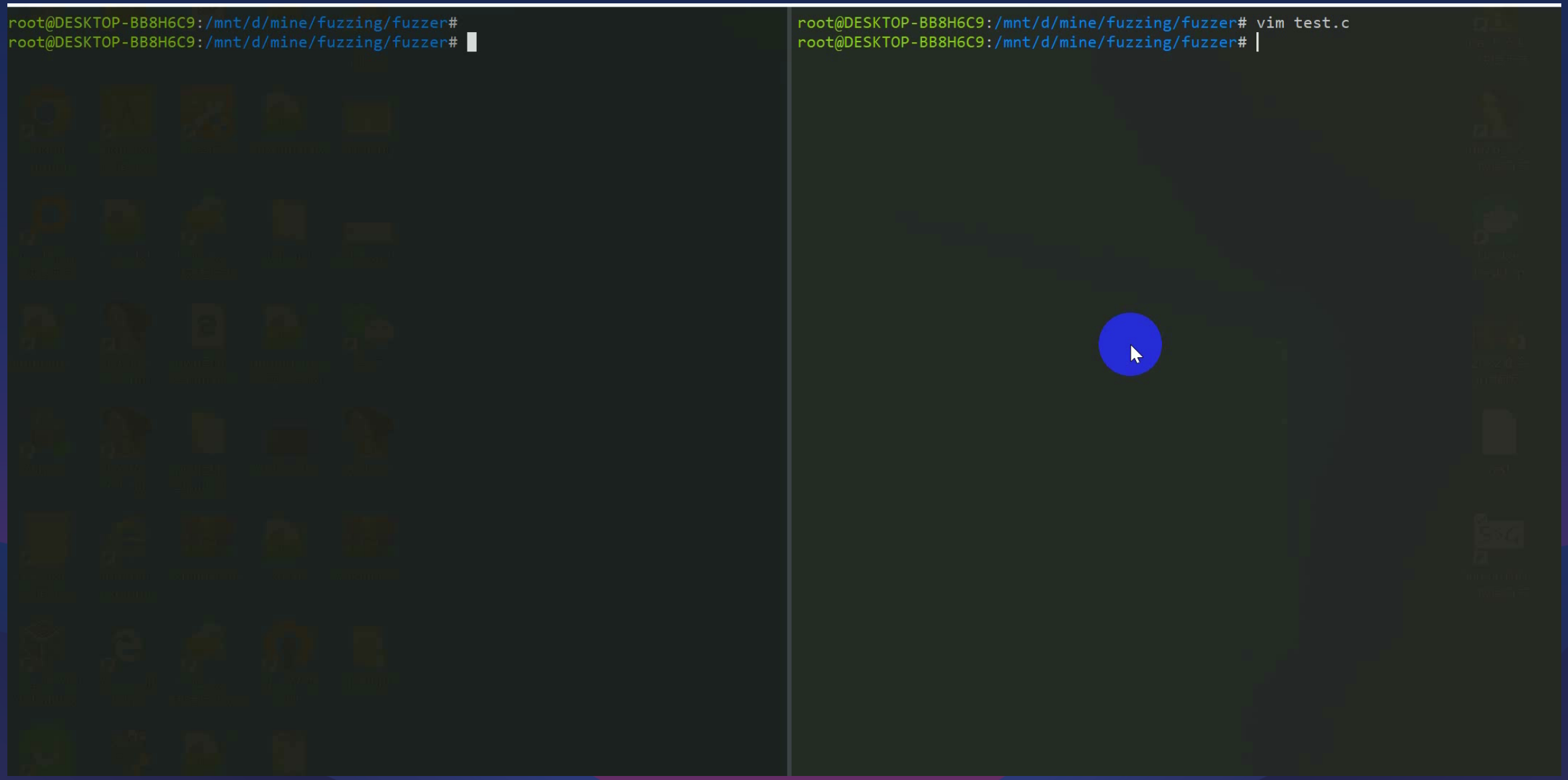
- 运行完一段代码片段一次时，检查malloc和free的数量是否一致。

- 但是这个检测准确的前提是，业务中的堆内存不会保存到全局变量中后续使用。

# Fuzzer框架


Fuzzer
corpus
Crash detector
随机选择种子
新路径发现则
加到corpus
保存导致crash的输入
Target
GOT hooker
输入
Mutator
动态patch，插入断点
Bin patcher
Dumart fuzzer

# DEMO


root@DESKTOP-BB8H6C9:/mnt/d/mine/fuzzing/fuzzer#
root@DESKTOP-BB8H6C9:/mnt/d/mine/fuzzing/fuzzer#
root@DESKTOP-BB8H6C9:/mnt/d/mine/fuzzing/fuzzer# vim test.c
root@DESKTOP-BB8H6C9:/mnt/d/mine/fuzzing/fuzzer#

# 背景：各种黑盒Fuzz工具比较

| | 路径反馈 | ASAN | 效率 | 状态机 | 其他 |
|---|---|---|---|---|---|
| Afl-qemu | Edge反馈 | QASAN | 中 | 困难 | |
| Afl unicorn | Edge反馈 | 不支持 | 中 | 困难 | |
| peach | 不支持 | 不支持 | 低 | 困难 | Dumb fuzz |
| Frida-fuzzer | Edge反馈 | 不支持 | 高 | 容易 | |
| Dumart fuzzer | Block反馈 | 支持 | 很高 | 容易 | 需要有root权限的真实设备 |

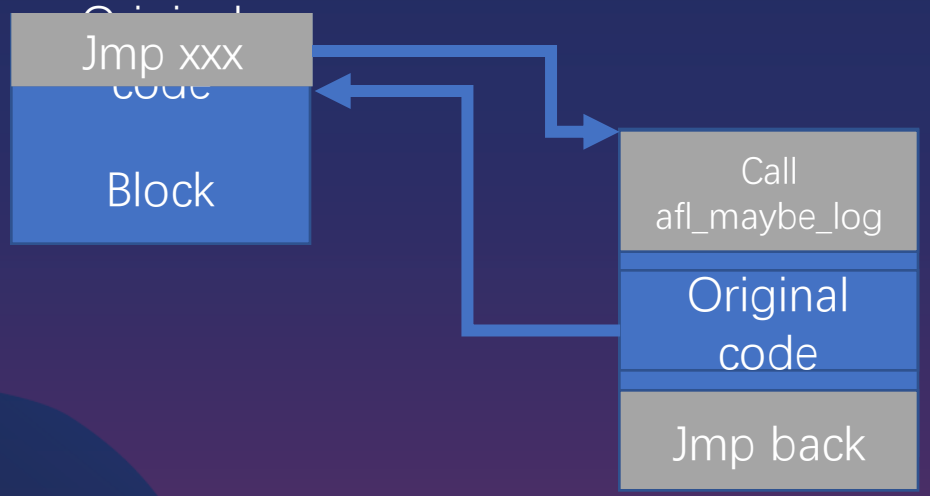
# 为什么叫Dumart fuzz

- Make dumb fuzz smart

dumart

without SB

# Todo: edge反馈




- 参考AFL
- 动态hook，跳转AFL_maybe_log
- 加入位图，记录edge状态

# Todo

- 更多的架构：如Arm
- 变异加入字典功能等