Trustwave® SpiderLabs®

Exploiting COF Vulnerabilities in the Linux kernel

Vitaly Nikolenko
@vnik5287

Ruxcon - 2016

Who am I?

Trustwave® SpiderLabs®

- Security researcher @ SpiderLabs
- Exploit dev / bug hunting / reverse engineering

Agenda

1. Counter overflows in the kernel
    - Exploitation techniques
    - Real case studies
    - Corner cases and challenges
2. COF static code analyser

Introduction

- All trivial bugs are fixed (mostly)
- Fuzzing (dumb, smart, guided by code coverage)
- kmemcheck
    - Detects use-after-free accesses and uninitialised-memory-reads
- SLUB_DEBUG / DEBUG_SLAB
    - Enables redzones and poisoning (writing magic values to check later)
    - Can detect some out-of-bounds and use-after-free accesses

Introduction

- DEBUG_PAGEALLOC
  - Unmaps freed pages from address space
  - Can detect some use-after-free accesses
- KASan
  - Fast and comprehensive solution for both UAF and OOB
  - Detects out-of-bounds for both writes and reads
- KTSan
  - Fast data-race and deadlock detector

Introduction

- Counter overflows are not easily detectable
- Would require triggering the vulnerable path 2^32 times before UAF
- Existing bug detection techniques are not very useful

Counter overflows

- The purpose of the OS is to allow *(concurrent)* consumers

- These consumers have a demand for *(shared)* resources that the OS needs to manage

- The kernel needs to keep *reference counters* for shared resources, e.g., file descriptors, sockets, process specific structs, etc.

Counter overflows

- Counter overflows - special case of integer overflows and UAF

- There's a vulnerable kernel path (reachable from user space) where

  - `counter increments > counter decrements` (counter overflow)

  - `counter increments < counter decrements` (counter underflow)

File refcounting

struct file

```
type = struct file {
    union {
        struct llist_node fu_llist;
        struct callback_head fu_rcuhead;
    } f_u;
    struct path f_path;
    struct inode *f_inode;
    const struct file_operations *f_op;
    spinlock_t f_lock;
    atomic_t f_count;
    unsigned int f_flags;
    fmode_t f_mode;
    struct mutex f_pos_lock;
    loff_t f_pos;
    struct fown_struct f_owner;
    const struct cred *f_cred;
    struct file_ra_state f_ra;
    ...
}
```

File refcounting

syscall(open, …)

```
struct file *get_empty_filp(void)
{
        const struct cred *cred = current_cred();
        static long old_max;
        struct file *f;
        int error;

        f = kmem_cache_zalloc(filp_cachep, GFP_KERNEL);
        if (unlikely(!f))
                return ERR_PTR(-ENOMEM);

        percpu_counter_inc(&nr_files);
        f->f_cred = get_cred(cred);
        error = security_file_alloc(f);
        if (unlikely(error)) {
                file_free(f);
                return ERR_PTR(error);
        }

        INIT_LIST_HEAD(&f->f_u.fu_list);
        atomic_set(&f->f_count, 1);
...
```

File refcounting

Sharing the fd

```
static inline struct file *
get_file(struct file *f)
{
        atomic_inc(&f->f_count);
        return f;
}
```

File refcounting

Closing fd/exiting the process

```
void fput(struct file *file)
{
    if (atomic_dec_and_test(&file->f_count)) {
        struct task_struct *task = current;
        file_sb_list_del(file);
        ...

        if (llist_add(&file->f_u.fu_llist,
&delayed_fput_list))
            schedule_delayed_work(&delayed_fput_work,
1);
    }
}
```

File refcounting

Atomic integers

- *Atomic* API implemented by the kernel:
  - `atomic_set` - set atomic variable
  - `atomic_inc` - increment atomic variable
  - `atomic_dec` - decrement atomic variable
  - `atomic_dec_and_test` — decrement and test
  - `atomic_inc_and_test` — increment and test
  - etc

File refcounting

Atomic integers

```
(gdb) ptype atomic_t
type = struct {
    int counter;
}
```

Counter overflows

- Data models:
  - x86 - ILP32
  - x86_64 - LP64
- Signed integer 0 to 0xffffffff
- Overflowing 4 bytes is quick right?

Counter overflows

i7-4870HQ CPU @ 2.50GHz - user space

```
#include <stdio.h>

int main() {
  unsigned int count;

  for (count = 0; count < 0xffffffff; count++)
    ;

  return 0;
}
```

```
test:~ vnik$ time ./t

real  0m8.293s
user  0m8.267s
sys   0m0.015s
```

Counter overflows

i7-4870HQ CPU @ 2.50GHz - kernel space

```
struct test {
        atomic_t count;
        struct rcu_head rcu;
};

static void increment() {
        atomic_inc(&testp->count);
}

static long device_ioctl(struct file *file, unsigned int cmd,
unsigned long args) {

        switch(cmd) {
        case IOCTL_SET: /* set counter value */
                atomic_set(&testp->count, args);
                break;
        case IOCTL_INCREMENT:
                increment();
                break;
...
}
```

Counter overflows

i7-4870HQ CPU @ 2.50GHz - kernel space

```
int main() {
  int fd;

  fd = open(DEVICE_PATH, O_RDONLY);

  if (fd == -1)
        return -1;

  ioctl(fd, IOCTL_SET, 0);

  unsigned count;

  for (count = 0; count < 0xffffffff; count++)
    ioctl(fd, IOCTL_INCREMENT, 0);
}
```

```
vnik@ubuntu:~/$ time ./trigger1

real 58m48.772s
user 1m17.369s
sys  32m49.483s
```

Counter overflows

Overflowing kernel integers

Trustwave® SpiderLabs®

- At least 30-60 min to overflow (approximately)

- Not very *practical* in certain exploitation scenarios (mobile root?)

Counter overflows

```
void * some_kernel_function() {
        ...
        struct file *f = fget(fd);
        ...
        if (some_error_condition)
                goto out;
        ...
        if (atomic_dec_and_test(&f->f_count)) {
                call_rcu(...); // fput(f)
out:
        return -EINVAL;
}
```

VMM

- Kernel implements a virtual memory abstraction layer
  - Using physical memory allocations is inefficient (fragmentation, increased swapping)
- Basic unit of memory is a page (>= 4KB)
- Kernel allocates memory *internally* for a large variety of objects

VMM

Terminology

- Pages are divided into smaller *fixed* chunks (power of 2) aka *slabs*

- Pages containing objects of the same size are grouped into *caches*

- *SLAB* allocator is the original slab allocator on implementation in OpenSolaris

    - *SLAB* (in caps) - specific slab allocator implementation

    - *slab* - generic term

VMM

Linux SLUB allocator

Trustwave® SpiderLabs®

- Starting from 2.6 branch, the slab allocator can be selected at compile time (SLAB, SLUB, SLOB, SLQB)

- SLUB is the default slab allocator on Linux

- All allocators perform the same function (and are mutually exclusive) but there’re significant differences in exploitation

VMM

Linux SLUB allocator

- General-purpose allocations (`kmalloc/kzalloc`) are for objects of size 8, 16, 32, 64, 128, …, 8192 bytes

- Objects that are not power of 2 are rounded up to the next closest slab size

- Special-purpose allocations (`kmem_cache_alloc`) are for frequently-used objects

- *Objects of the ~same size are grouped into the same cache*

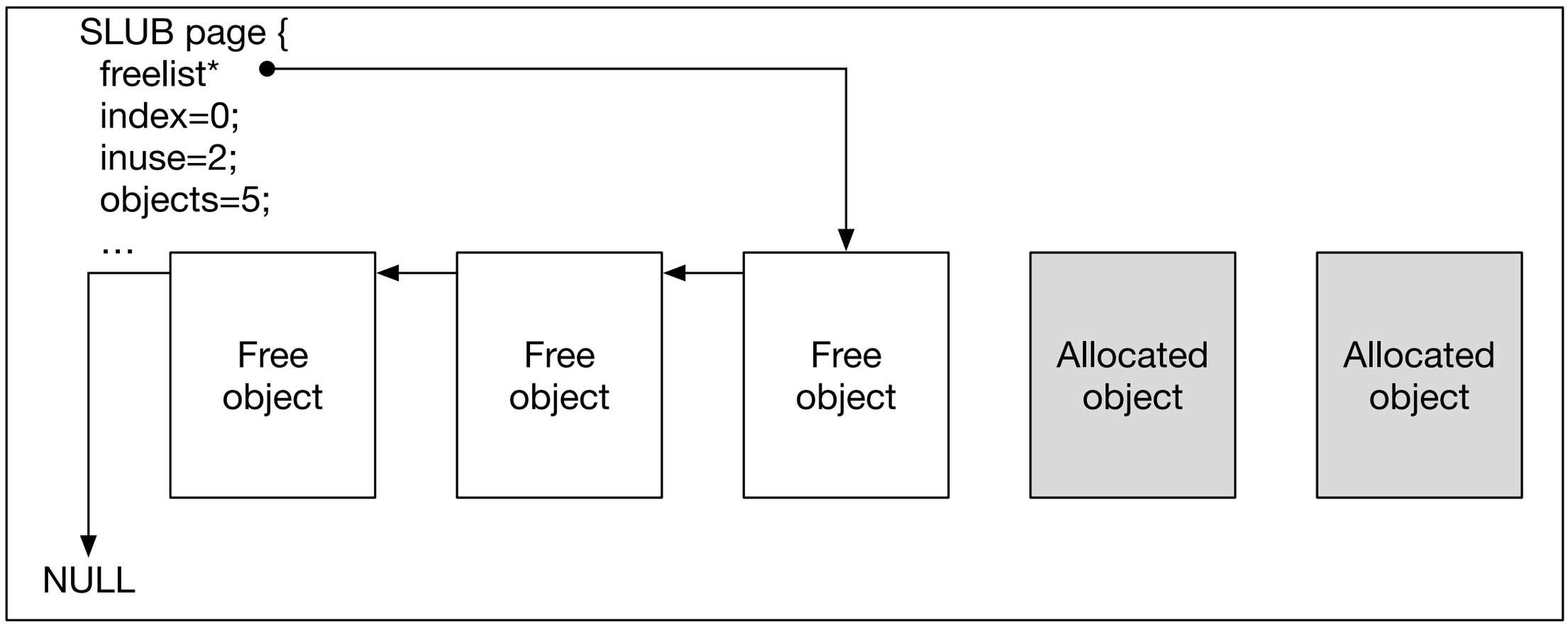
VMM

Linux SLUB allocator

- No metadata in slabs
- Instead, each slab page (`struct page`) has SLUB metadata (`freelist` ptr, etc)
- Free objects have a pointer (at `offset=0`) to the next free object in the slab (i.e., linked list)
- The last free object in the slab has its next pointer set to NUL

VMM

Linux SLUB allocator



Counter overflows

Exploitation procedure

1. Overflow the counter by calling the vulnerable path `A()` until the `counter -> 0`

2. Find path `B()` that triggers kfree() if `counter == 0`

3. Start allocating *target* objects to fill partial slabs of the same size `C()`

4. Use the old object reference to

    1. Modify the target object, or

    2. Execute the (*overwritten*) function ptr in the vulnerable object

5. Trigger the overwritten function ptr in the target object

Counter overflows

Step 4 - option #1

```
// assume sizeof(struct A)
   == sizeof(struct B)

struct A {
    atomic_t counter;

    int some_var;
    ...
};

struct B {
    void (*func)();
    ...
};
```

```
// Old kernel path
...
a->some_var = 0;
...
```

Counter overflows

Step 4 - option #2

```
// assume sizeof(struct A) ==
   sizeof(struct B)

struct A {
    atomic_t counter;

    void (*func)();

    ...
};

struct B {
    int dummy;
    long user_controlled_var;
    ...
};
```

```
// Old kernel path
...
a->func(...);
...
```

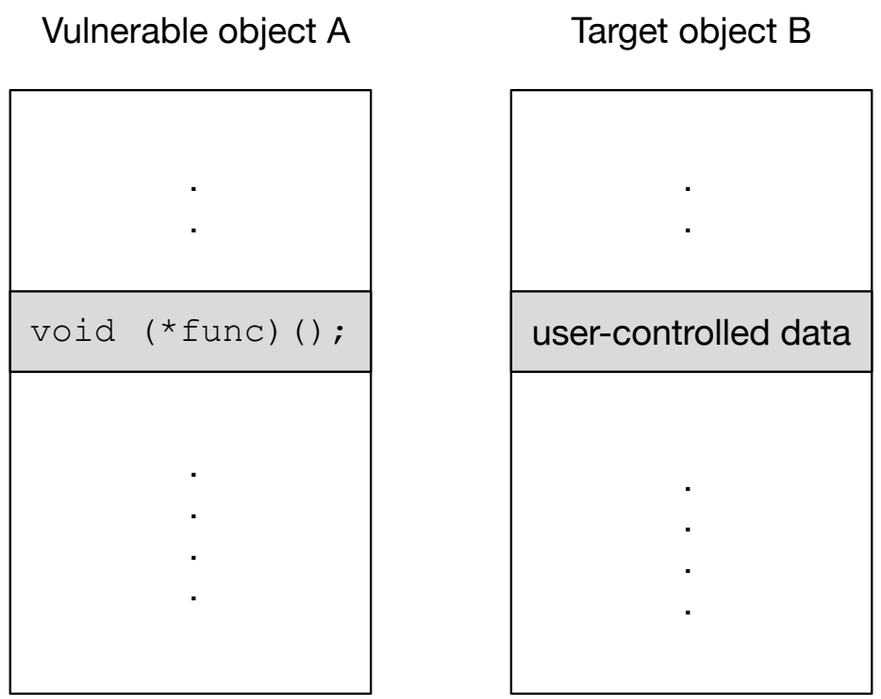
Counter overflows

Step 4 - option #2

- Find the target object *B*, s.t.,

    1. *B* has user-controlled variables

    2. The user-controlled variable is aligned with the function pointer in the vulnerable object *A*



Counter overflows

msgsnd() syscall

```
struct {
     long mtype;
     char mtext[ARBITRARY_LEN];
} msg;

memset(msg.mtext, 'A', sizeof(msg.mtext));

msqid = msgget(IPC_PRIVATE, 0644 | IPC_CREAT);

if (msgsnd(msqid, &msg, sizeof(msg.mtext), 0)
== -1) {
     ...
```

Counter overflows

msgsnd() syscall

```
long do_msgsnd(int msqid, long mtype, void __user *mtext,
                size_t msgsz, int msgflg)
{
    struct msg_queue *msq;
    struct msg_msg *msg;
    int err;
    struct ipc_namespace *ns;

    ns = current->nsproxy->ipc_ns;

    if (msgsz > ns->msg_ctlmax || (long) msgsz < 0 ||
msqid < 0)
            return -EINVAL;
    if (mtype < 1)
            return -EINVAL;

    msg = load_msg(mtext, msgsz);
    ...
```

Counter overflows

msgsnd() syscall

```
struct msg_msg *load_msg(const void __user *src, size_t
len)
{
     struct msg_msg *msg;
     struct msg_msgseg *seg;
int err = -EFAULT;
size_t alen;

msg = alloc_msg(len);
if (msg == NULL)
        return ERR_PTR(-ENOMEM);

alen = min(len, DATALEN_MSG);
if (copy_from_user(msg + 1, src, alen))
        goto out_err;
...
```

RCU

- Kernel counter decrements and object freeing are often implemented via RCU calls

- This introduces *indeterminism* in counter values

- If 0-check is done using an RCU callback, can skip the check and overflow past 0

RCU

- Read-Copy Update - synchronisation mechanism (replacement for read-writer locking)
  - Available since 2002
    - Low-overhead readers, immune to deadlocks, readers can run concurrently with updaters, etc.
- `rcu_read_lock()/rcu_read_unlock()`, `rcu_assign_pointer()`, `rcu_dereference()`, etc.
- Kernel API in `./Documentation/RCU/rcu.txt`

RCU

```
void call_rcu(struct rcu_head *head,
void (*callback) (void *head));
```

- This call invokes a callback function `callback()` after all the CPUs have gone through at least one *quiescent* state.

- "For performance reasons, good implementations of RCU do not wait for completion of the quiescent cycle. Instead, they queue the cycle in batches and return."

RCU

Quiescent state

- The CPU performs a process switch
- The CPU starts executing in user mode
- The CPU executes the idle loop

RCU

```
static void callback_fn() {
  atomic_dec(&testp->count);
}

static void testfn() {
  atomic_inc(&testp->count);
  call_rcu(&testp->rcu, callback_fn);
}
```

```
static long device_ioctl(struct file *file, unsigned
cmd, unsigned long args) {
  switch(cmd) {
  case IOCTL_SET: /* set counter value */
    atomic_set(&testp->count, args);
    break;
  case IOCTL_TEST: /* increment and decrement the count
*/
    testfn();
    break;
...
```

RCU

User-space trigger

```
int main() {
  int fd, i;

  fd = open(DEVICE_PATH, O_RDONLY);
if (fd == -1) {
  perror("open");
  return -1;
}

ioctl(fd, IOCTL_SET, 0);

for (i=0;  i < 100; i++) {
  ioctl(fd, IOCTL_TEST, NULL);
}
```

RCU

RCU calls

```
(gdb) b testfn
Breakpoint 17 at 0xffffffffa02250c0: file /home/vnik/rcu/rcu.c, line
64.
(gdb) commands
Type commands for breakpoint(s) 17, one per line.
>silent
>p testp->count
>c
>end
(gdb) c
Continuing.
$150 = {counter = 0}
$151 = {counter = 1}
$152 = {counter = 2}
$153 = {counter = 2}
$154 = {counter = 3}
$155 = {counter = 3}
$156 = {counter = 4}
$157 = {counter = 4}
$158 = {counter = 2}
$159 = {counter = 2}
$160 = {counter = 3}
...
```

RCU

User-space trigger

```
int main() {
  int fd, i;

  fd = open(DEVICE_PATH, O_RDONLY);

  if (fd == -1) {
    perror("open");
    return -1;
  }

  ioctl(fd, IOCTL_SET, 0);

  for (i=0;  i < 100; i++) {
     ioctl(fd, IOCTL_TEST, NULL);
     sleep(1); // let the CPU go through a quiescent
state
  }
```

RCU

Synchronised RCU calls

```
(gdb) b testfn
Breakpoint 18 at 0xffffffffa02250c0: file /home/vnik/rcu/rcu.c, line 64.
(gdb) commands
Type commands for breakpoint(s) 18, one per line.
>silent
>p testp->count
>c
>end
(gdb) c
Continuing.
$191 = {counter = 0}
$192 = {counter = 0}
$193 = {counter = 0}
$194 = {counter = 0}
$195 = {counter = 0}
$196 = {counter = 0}
$197 = {counter = 0}
$198 = {counter = 0}
$199 = {counter = 0}
$200 = {counter = 0}
$201 = {counter = 0}
$202 = {counter = 0}
$203 = {counter = 0}
...
```

Counter overflows

Test cases

- CVE-2014-2851 - `group_info` counter overflow (exploit published Jan 2016)
- CVE-2016-0728 - `keyring` counter overflow (write up published 3 weeks later)

CVE-2016-0728

- “This vulnerability has implications for approximately tens of millions of Linux PCs and servers, and 66 percent of all Android devices (phones/tablets).” - Perception Point

- Google said company researchers don't believe any Android devices are vulnerable to exploits by third-party applications. It also said researchers believe that the number of Android devices affected is "significantly smaller than initially reported."

CVE-2016-0728

- Reference leak in the keyrings facility
- System call interface – `keyctl` syscall is provided so that userspace programs can use the facility for their own purposes.
- Each process can create a keyring for the current session using `keyctl(KEYCTL_JOIN_SESSION_KEYRING, name)`
- The keyring object can be shared between processes by referencing the same keyring name.

CVE-2016-0728

User-space trigger

Trustwave® SpiderLabs®

```
for (i = 0; i < 0xfffffffd; i++) {
    serial = keyctl(KEYCTL_JOIN_SESSION_KEYRING,
"mykeyring");
    if (serial < 0) {
        perror("keyctl");
        return -1;
    }
}
```

CVE-2016-0728

Trustwave® SpiderLabs®

```
long join_session_keyring(const char *name)
{
        ...
        new = prepare_creds(); // increment the counter
        keyring = find_keyring_by_name(name, false);
        if (PTR_ERR(keyring) == -ENOKEY) {
        ...
        } else if (IS_ERR(keyring)) {
                ret = PTR_ERR(keyring);
                goto error2;
        } else if (keyring == new->session_keyring) {
                ret = 0;
                goto error2;
        }
        ...
        key_put(keyring);
okay:
        return ret;
error2:
        mutex_unlock(&key_session_mutex);
error:
        abort_creds(new); // decrement the counter via RCU

        return ret;
}
```

CVE-2016-0728

struct key

Trustwave® SpiderLabs®

```
struct key {
   atomic_t              usage;   /* number of references */
   key_serial_t          serial; /* key serial number */
      ...
   union {
        struct keyring_index_key index_key;
        struct {
            struct key_type *type; /* type of key */
            char                *description;
        };
   };
...
```

CVE-2016-0728

Vulnerable object

- `keyring` object is of type `struct key`
- The object size is 184 bytes —> kmalloc-192 cache
- `type` ptr points to `key_type` struct containing function pointers:

```
struct key_type {
    ...
    int (*vet_description)(const char *description);
    int (*preparse)(struct key_preparsed_payload *prep);
    void (*free_preparse)(struct key_preparsed_payload *prep);
    int (*instantiate)(struct key *key, struct key_preparsed_payload *prep);
    int (*update)(struct key *key, struct key_preparsed_payload *prep);
    int (*match_preparse)(struct key_match_data *match_data);
    void (*match_free)(struct key_match_data *match_data);
    void (*revoke)(struct key *key);
    ...
```

CVE-2016-0728

Exploitation

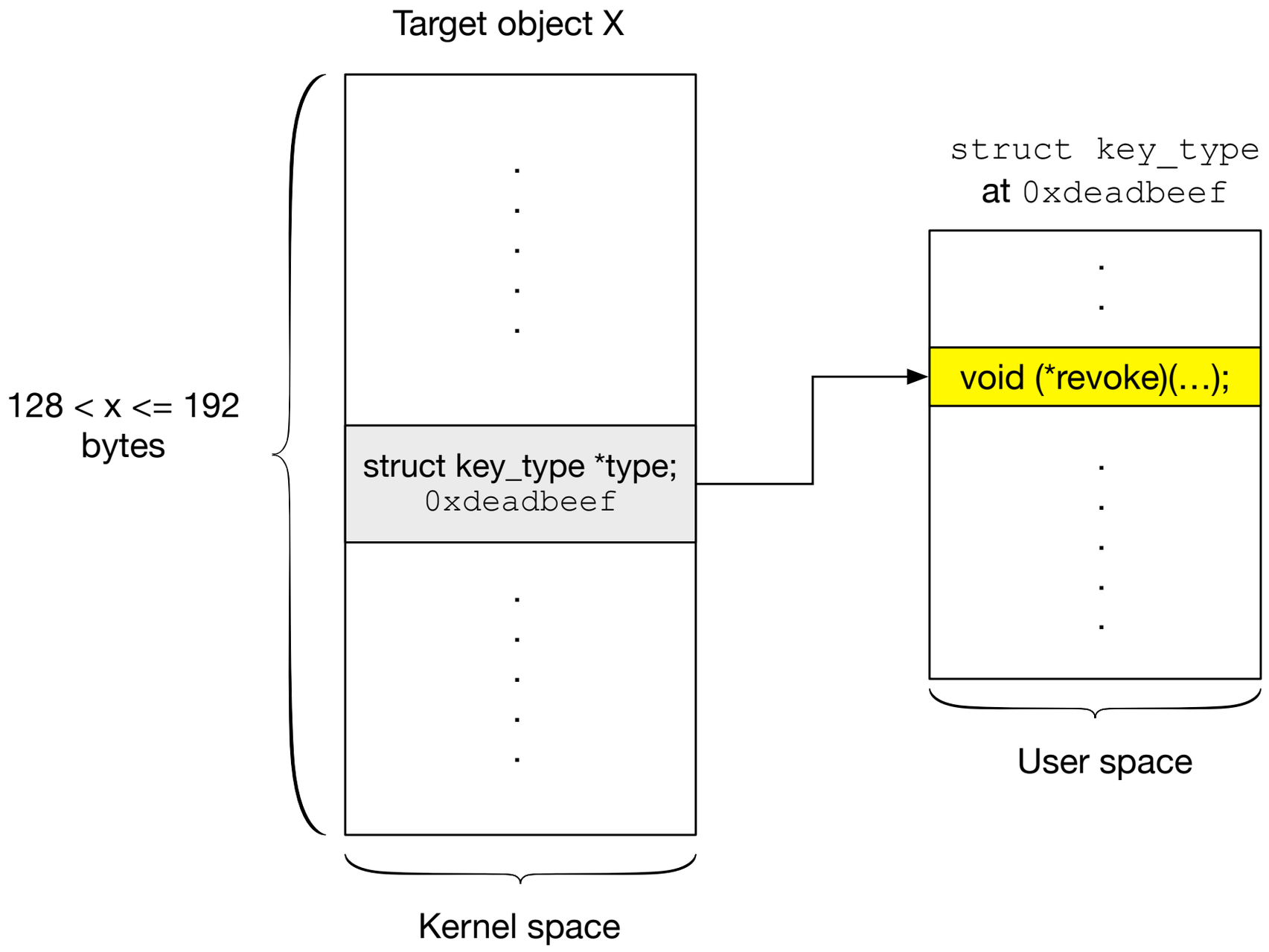
1. Overflow the `keyring` reference counter and trigger `kfree()`
   - Increment to `0xffffffff - eps`, then `eps` ordered steps with `sleep(1)`

2. Allocate the *target* object replacing the vulnerable object (option 2)

3. Trigger the overwritten function ptr in `keyring`, e.g., `type->revoke()`

CVE-2016-0728

Exploitation



CVE-2014-2851

- Counter overflow in `ping_init_sock()`

```
socket(AF_INET, SOCK_DGRAM, IPPROTO_ICMP);
```

- The vulnerable path can be reached by non-privileged users

CVE-2014-2851

Vulnerable path

Trustwave® SpiderLabs®

```
int ping_init_sock(struct sock *sk)
{
    struct net *net = sock_net(sk);
    kgid_t group = current_egid();
    struct group_info *group_info = get_current_groups();
    int i, j, count = group_info->ngroups;
    kgid_t low, high;

    inet_get_ping_group_range_net(net, &low, &high);
    if (gid_lte(low, group) && gid_lte(group, high))
        return 0;
...
```

CVE-2014-2851

struct group_info

Trustwave® SpiderLabs®

- sizeof(struct group_info) = 140 bytes (kmalloc-192)

```
type = struct group_info {
    atomic_t usage;
    int ngroups;
    int nblocks;
    kgid_t small_block[32];
    kgid_t *blocks[];
}
```

CVE-2014-2851

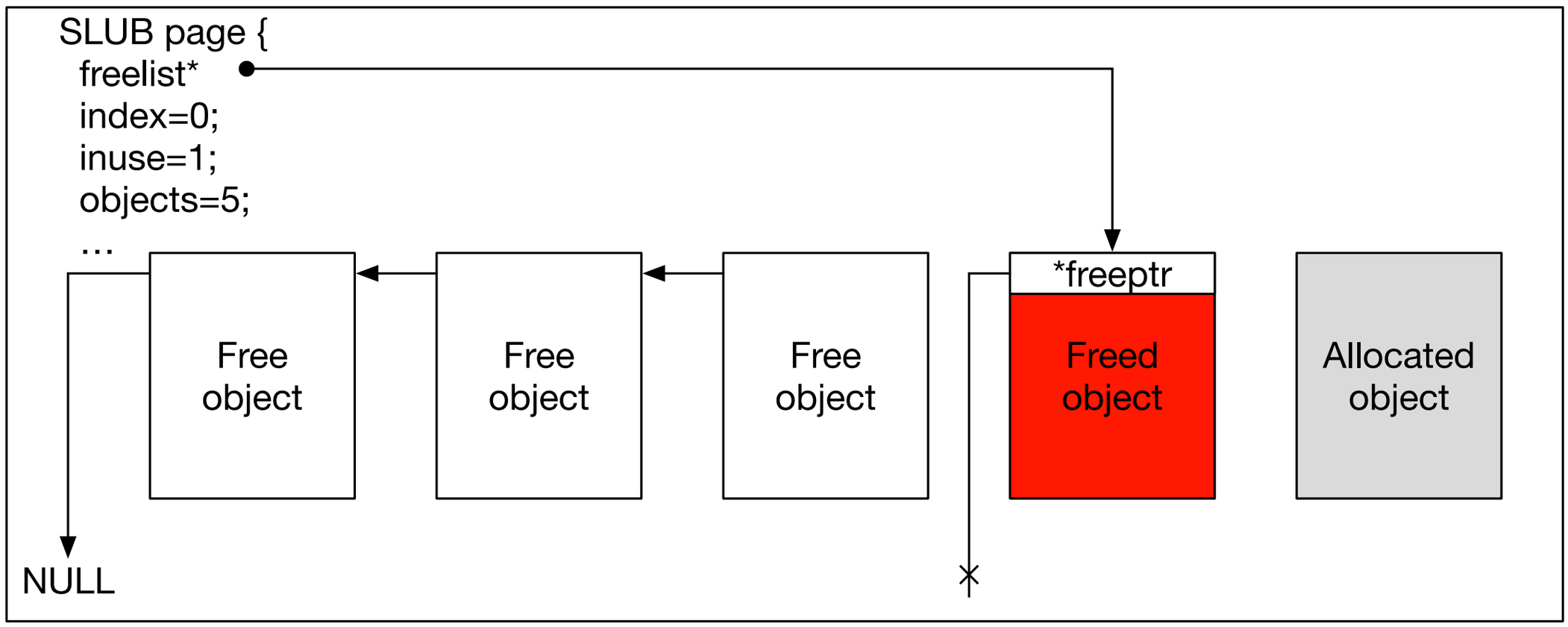
Challenges

- `group_info` is a frequently used object
- Does not contain any function pointers
- Cannot modify any object values (except for incrementing the refcounter)
- Race window between `group_info` free and target object allocation is very small
- Other processes use the old reference

CVE-2014-2851

Challenges

Trustwave® SpiderLabs®



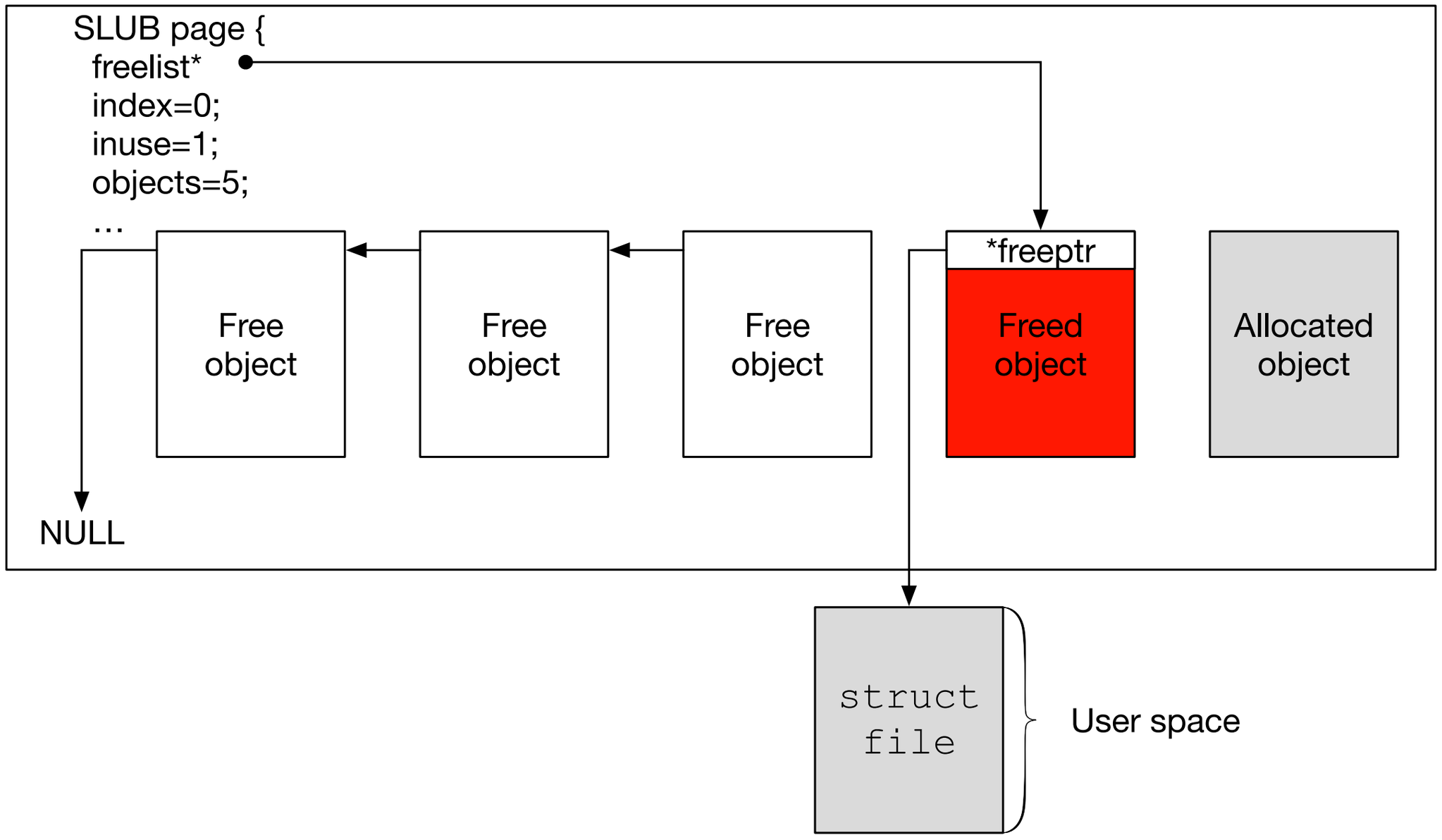
CVE-2014-2851

Exploitation - slab metadata overwrite

1. Increment the `group_info` usage counter close to `0xffffffff` by creating ICMP sockets
2. Increment the usage counter by 1 and try to free `group_info` via `faccessat()` && repeat
3. Keep incrementing the `group_info` usage counter until it points to some user-space memory address
4. Map this region in user space and memset it to 0
5. Start allocating structure X in kernel space that has the size 128-192 bytes
6. The SLUB allocator will allocate this structure X at our user-space address

CVE-2014-2851

Exploitation - slab metadata overwrite



Trustwave® SpiderLabs®

COF Static Code Analyser

COF Analyser

Algorithm

Trustwave® SpiderLabs®

Step 1: CFG with inlined function invocations

Step 2: CFG —> NFA

Step 3: NFA —> DFA

Step 4: DFA —> RE

COF Analyser

CFG

Trustwave® SpiderLabs®

- CFG for the entire kernel, then focus on user-space reachable code only (syscalls, ioctls, socket operations, etc.)

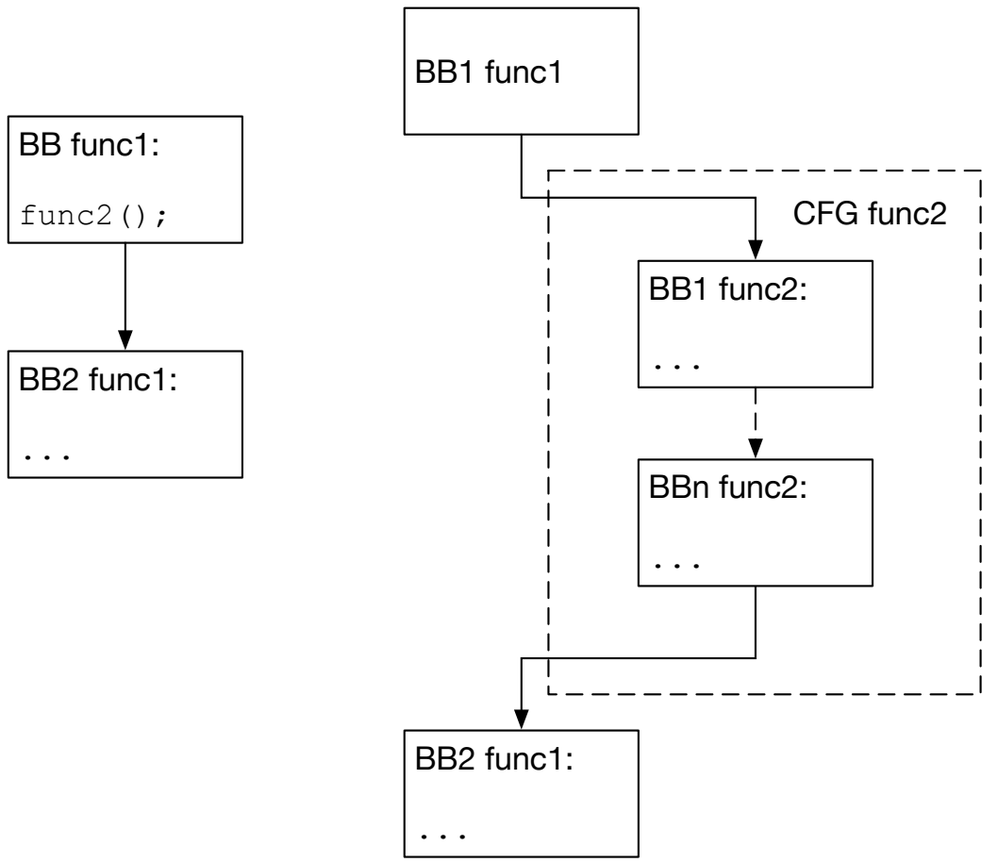
- LLVM for the kernel - http://llvm.linuxfoundation.org

- GCC Link Time Optimisation (LTO) - “gives GCC the capability of dumping its internal representation (GIMPLE) to disk, so that all the different compilation units that make up a single executable can be optimised as a single module.” - https://gcc.gnu.org/wiki/LinkTimeOptimization

COF Analyser

CFG

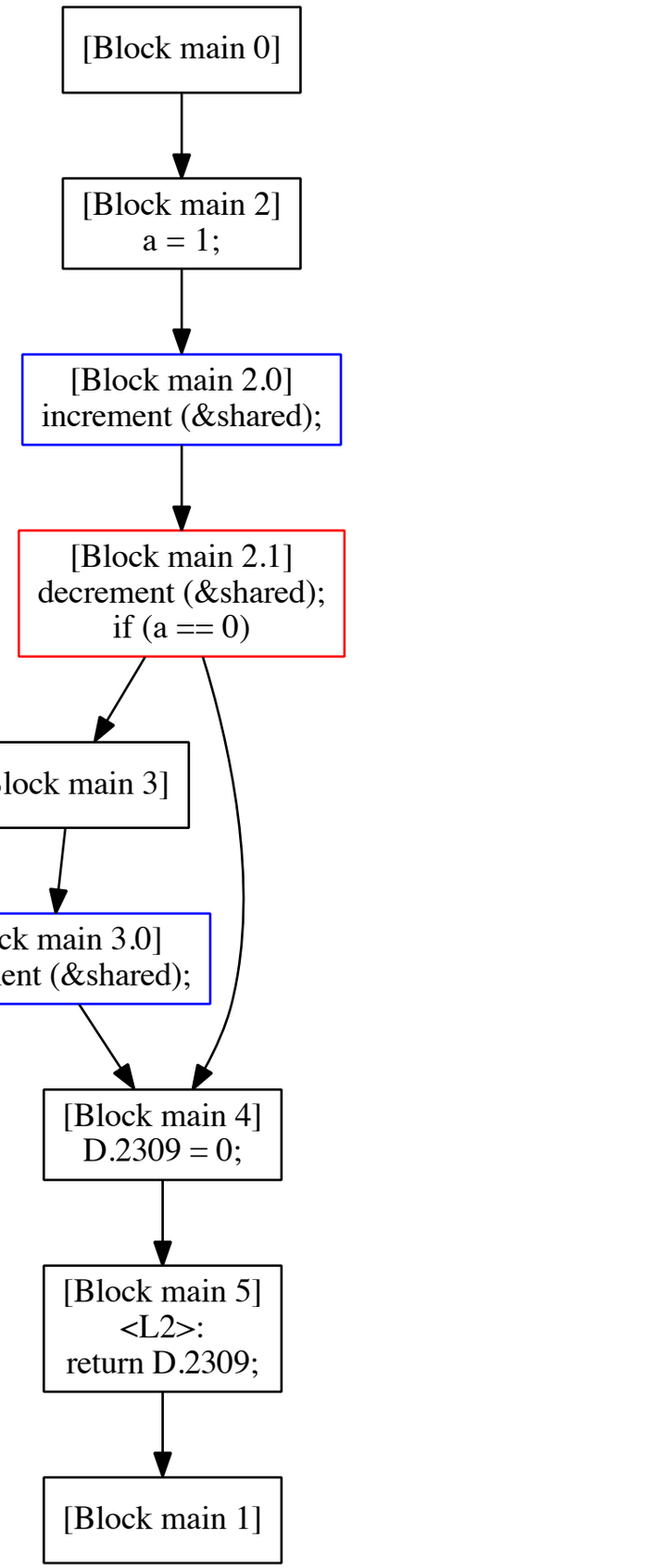
- GCC performs optimisation work in “passes” that form trees
- Custom pass called directly after the GCC “cfg” pass
- CFG for every function in the kernel, followed by manual function *inlining*



COF Analyser

CFG

```
struct shared_struct shared;

int main(int argc, char **argv) {
    int a = 0;

    increment(&shared);

    decrement(&shared);

    if (!a)
        increment(&shared);

    return 0;
}
```



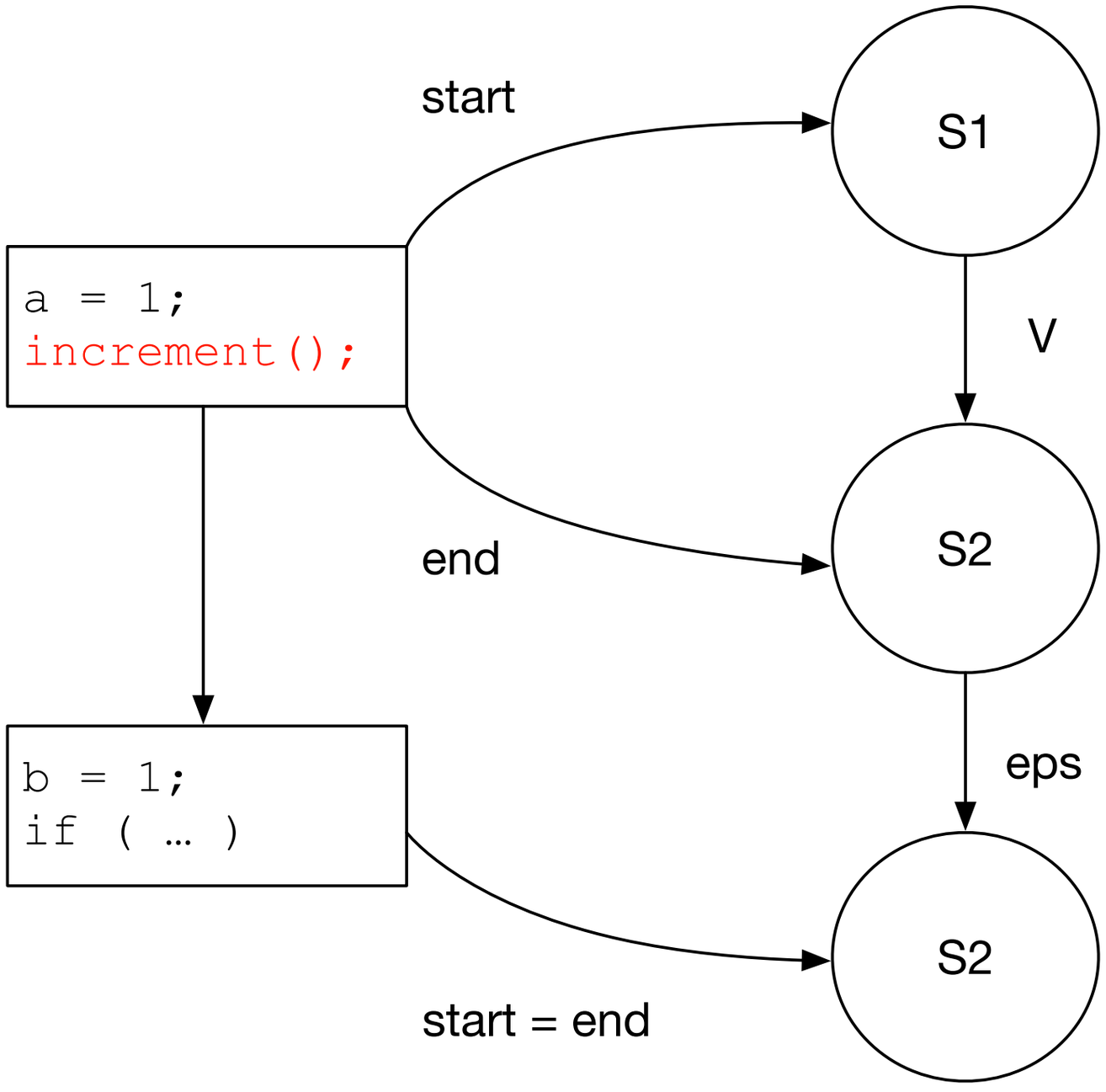
COF Analyser

CFG -> NFA

- Each basic block represents a state in the NFA
- Basic blocks containing the counter increment/decrement functions are split into *two* states
  - The edge (transition) between these two states is labeled with *V* (increment) or *P* (decrement)
- All other transitions are labeled with *epsilon*

COF Analyser

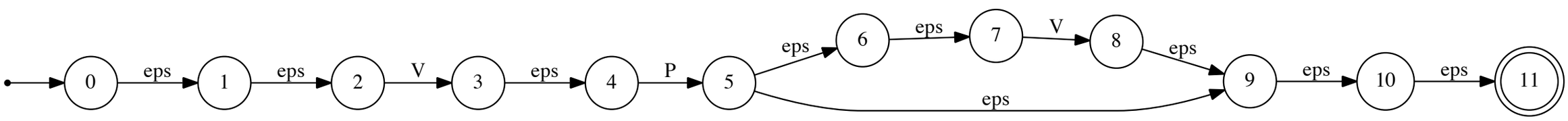
CFG -> NFA



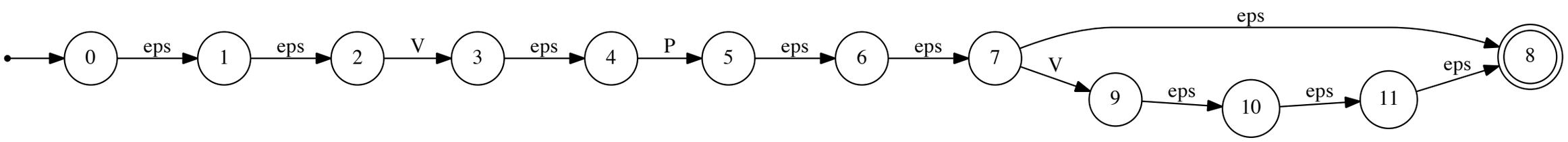
COF Analyser

NFA -> DFA

NFA



DFA



COF Analyser

DFA -> RE

- Convert the obtained DFA to a regular expression
- Change semantics of the language operations (language composition, union, closure, etc.)
- The outcome is the oracle algorithm: yes or no answer
- Can then reconstruct the original path from NFA

COF Analyser

Challenges and improvements

Trustwave® SpiderLabs®

- Eliminating false-positives
- “Points-to” analysis `some_struct->func_ptr()`
- Rewriting automata manipulation algorithms

COF Analyser

sendfile syscall

```
static ssize_t do_sendfile(int out_fd, int in_fd, loff_t
*ppos, size_t count, loff_t max)
{
        struct fd in, out;
        ...
        in = fdget(in_fd);
        if (!in.file)
                goto out;
        if (!(in.file->f_mode & FMODE_READ))
                goto fput_in;
fput_out:
                fdput(out);
fput_in:
                fdput(in);
out:
                return retval;
}
```

Trustwave® SpiderLabs®

Questions?