How I use a novel approach to exploit a limited OOB on Ubuntu at Pwn2Own Vancouver 2024

Pumpkin Chang (@u1f383)

November 7, 2024

DEVCORE

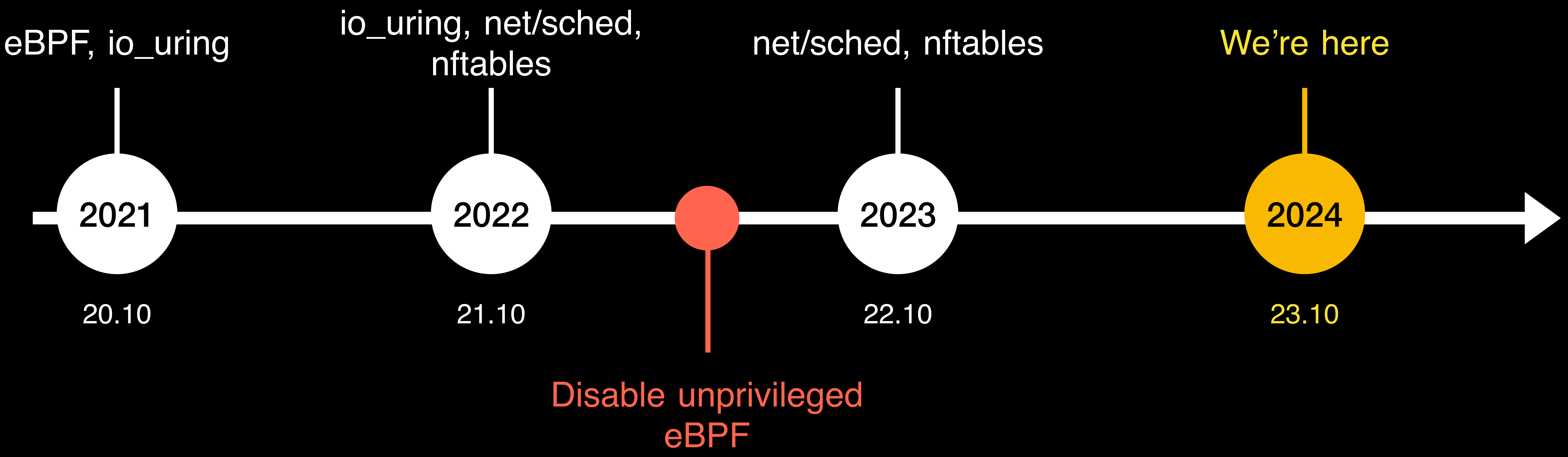
$ whoami

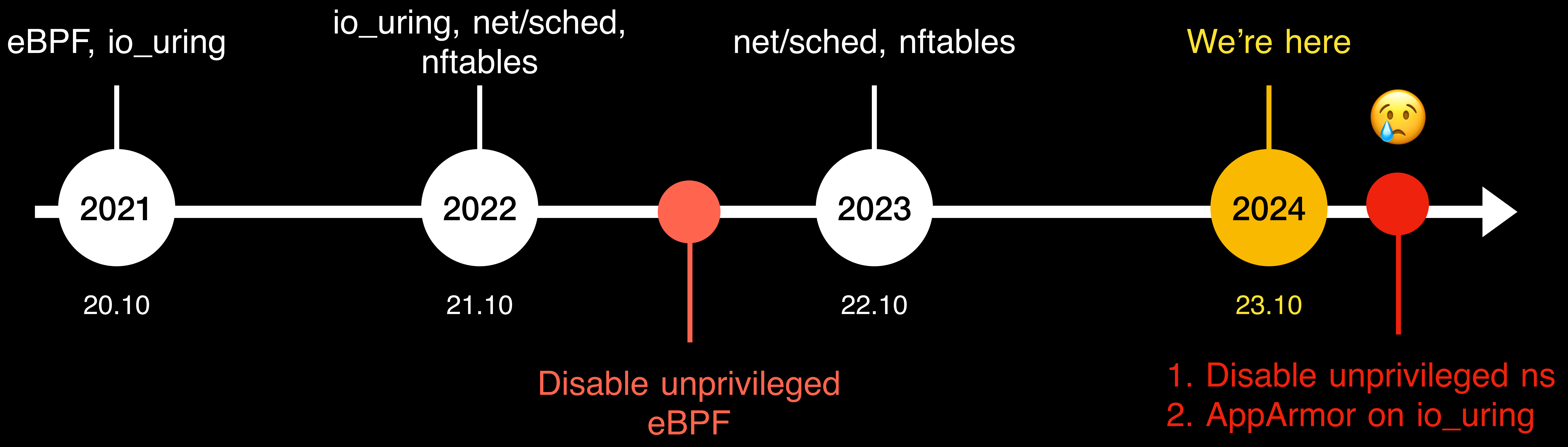
- Pumpkin 🎃 (@u1f383)
- Security researcher at DEVCORE
- Focus on Linux Kernel & Virtual Machine
- CTF Player in Balsn

$ ls -al ./outline

- Nov 28 2023 Target Selection
- Jan 19 2024 Bug Discovery
- Feb 21 2024 Crafting the Exploit
- Mar 20 2024 Achieving LPE
- Nov 7 2024 Takeaways

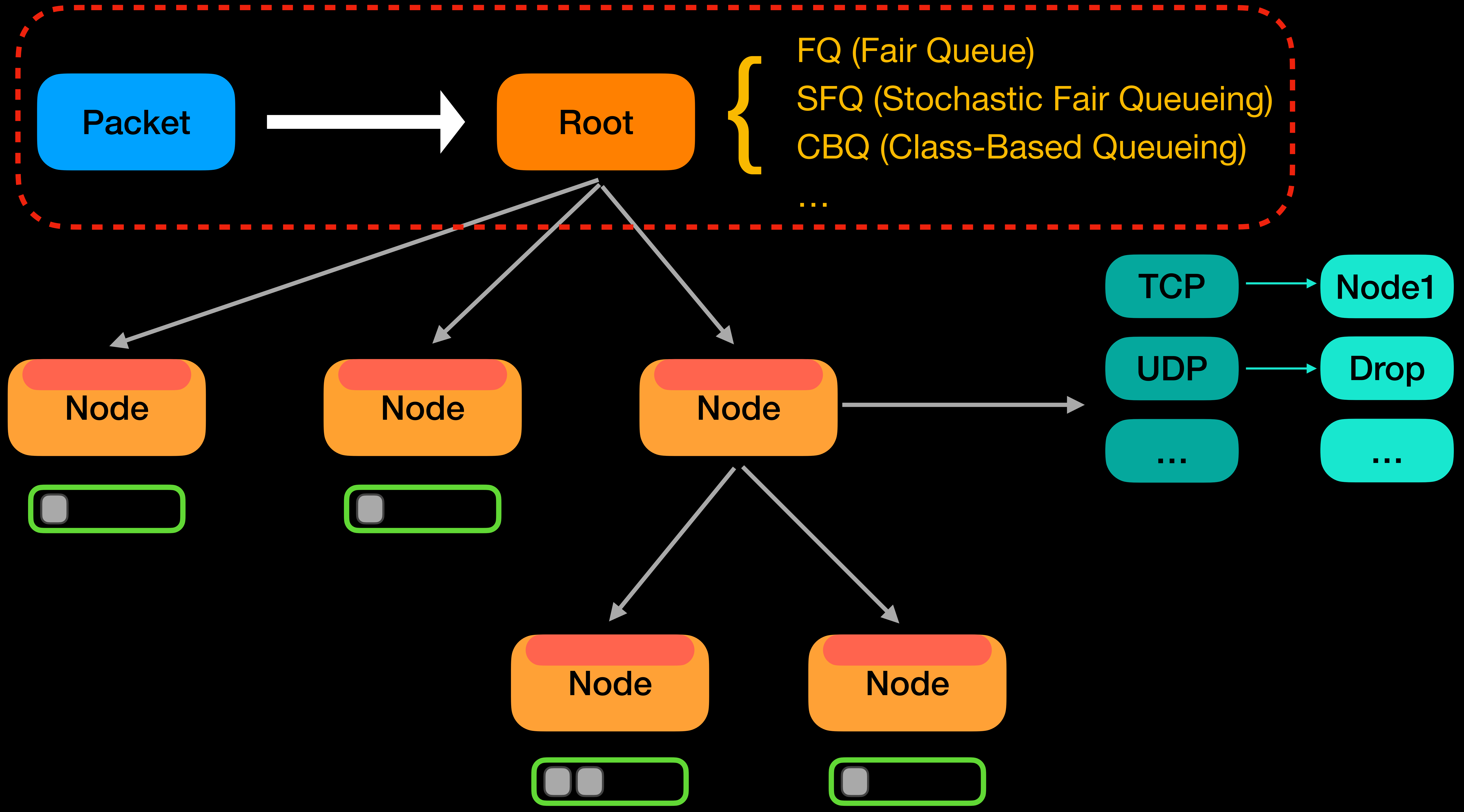
- Nov 28 2023 Target Selection
- Jan 19 2024 Bug Discovery
- Feb 21 2024 Crafting the Exploit
- Mar 20 2024 Achieving LPE
- Nov 7 2024 Takeaways


eBPF, io_uring
2021
20.10
io_uring, net/sched, nftables
2022
21.10
Disable unprivileged eBPF
net/sched, nftables
2023
22.10
We’re here
2024
23.10


eBPF, io_uring
2021
20.10
io_uring, net/sched, nftables
2022
21.10
Disable unprivileged eBPF
net/sched, nftables
2023
22.10
We're here
2024
23.10
1. Disable unprivileged ns
2. AppArmor on io_uring

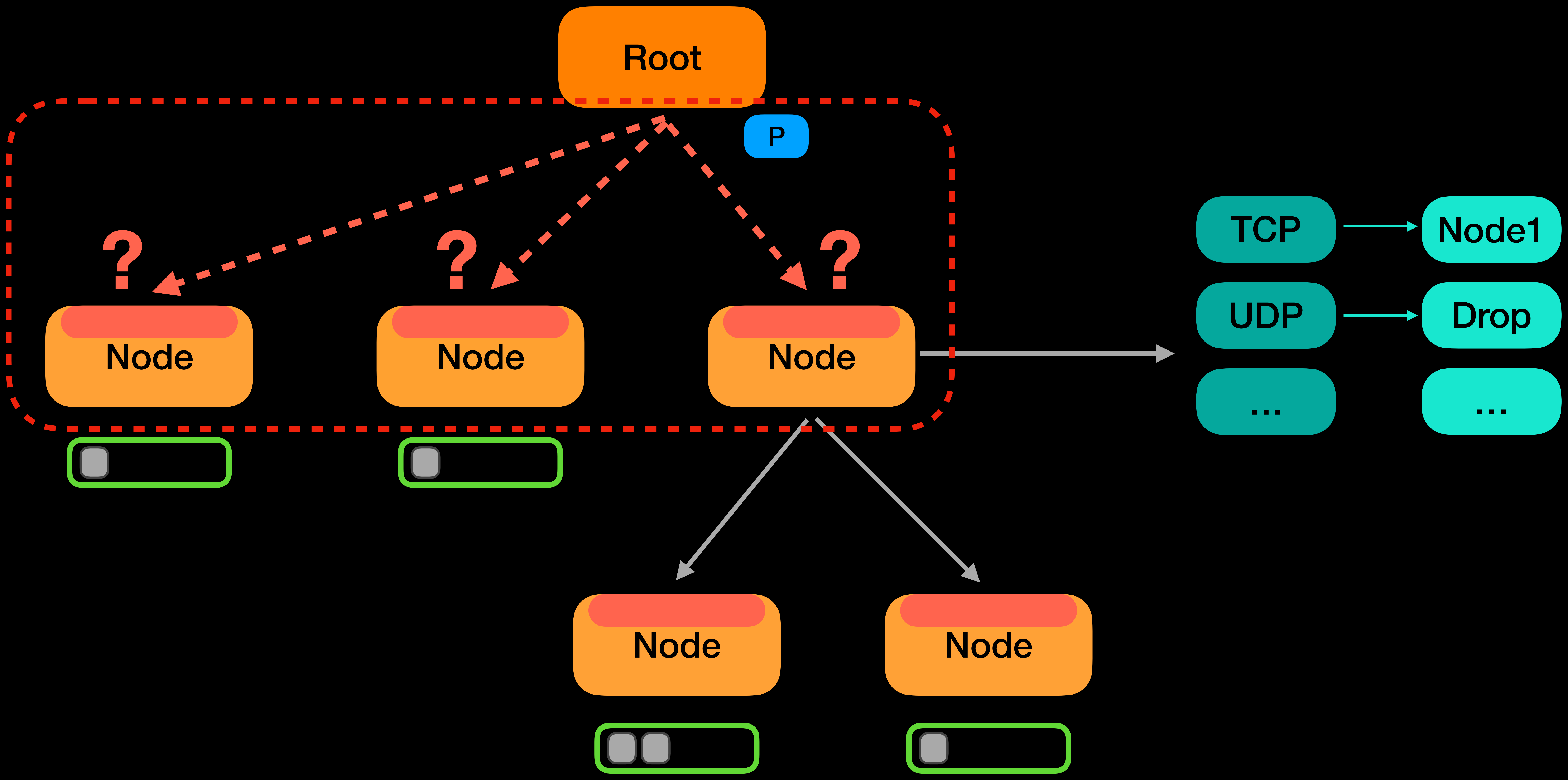
$ net/sched

- The Traffic Control (TC) subsystem in Linux consists of four core components:
    - Queueing Discipline (qdisc)
    - Class
    - Filter
    - Action



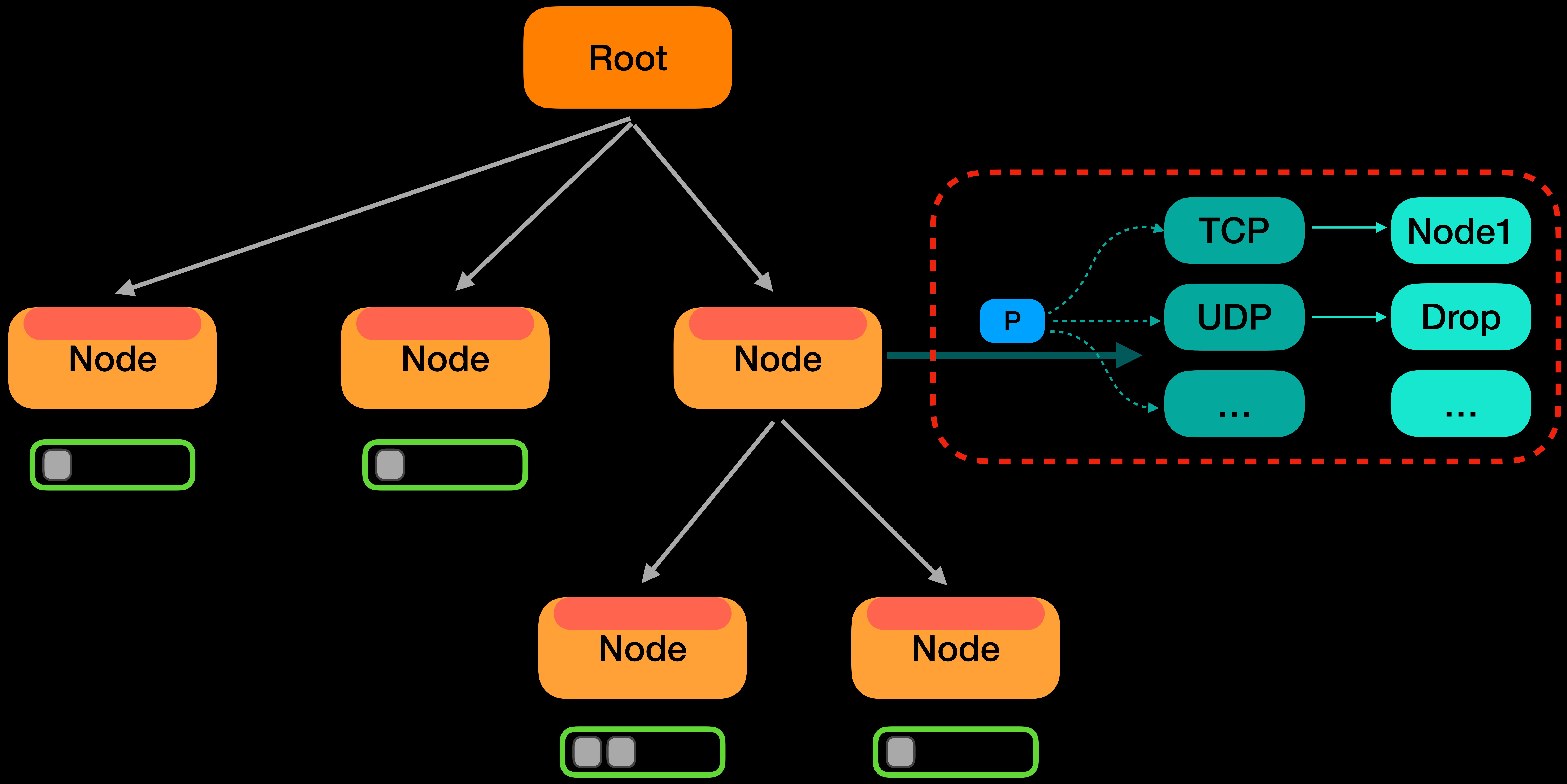
Qdisc

implement a scheduler in the dequeue algorithm



Class

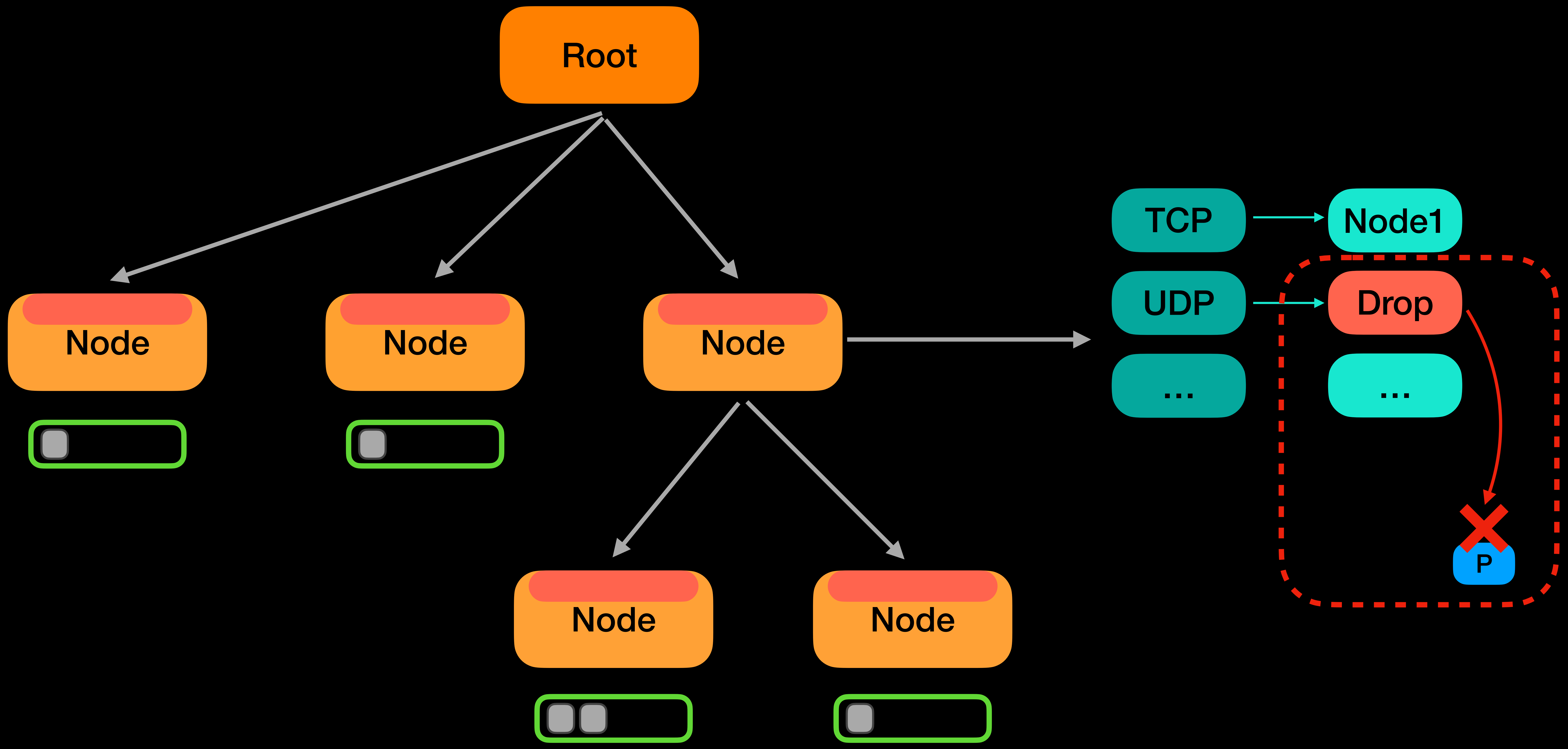
classify packets to qdiscs with different configurations



Filter

more fine-grained classification

by IP or protocol



Action

perform operation on packets,
such as drop and mirred

$ net/sched

- Interact with net/sched via NETLINK
- NETLINK APIs for data processing
  - Parsing - nla_parse_nested
  - Iteration - nla_for_each_nested_type
  - Retrieving attributes - nla_get_u32, …
- A nla_policy is required to ensure data safety

```c
static int taprio_parse_tc_entry(struct Qdisc *sch,
                                 struct nlattr *opt,
                                 u32 max_sdu[TC_QOPT_MAX_QUEUE],
                                 u32 fp[TC_QOPT_MAX_QUEUE],
                                 unsigned long *seen_tcs,
                                 struct netlink_ext_ack *extack)
{
	struct nlattr *tb[TCA_TAPRIO_TC_ENTRY_MAX + 1] = { };
	struct net_device *dev = qdisc_dev(sch);
	int err, tc;
	u32 val;

	err = nla_parse_nested(tb, TCA_TAPRIO_TC_ENTRY_MAX, opt,
			       taprio_tc_policy, extack);
	if (err < 0)
		return err;
```

```c
static const struct nla_policy taprio_tc_policy[TCA_TAPRIO_TC_ENTRY_MAX + 1] = {
	[TCA_TAPRIO_TC_ENTRY_INDEX]	= NLA_POLICY_MAX(NLA_U32,
						 TC_QOPT_MAX_QUEUE),
	[TCA_TAPRIO_TC_ENTRY_MAX_SDU]	= { .type = NLA_U32 },
	[TCA_TAPRIO_TC_ENTRY_FP]	= NLA_POLICY_RANGE(NLA_U32,
						   TC_FP_EXPRESS,
						   TC_FP_PREEMPTIBLE),
};
```

- Nov 28 2023 Target Selection
- Jan 19 2024 Bug Discovery
- Feb 21 2024 Crafting the Exploit
- Mar 20 2024 Achieving LPE
- Nov 7 2024 Takeaways

$ The Bug

- Time Aware Priority Scheduler (TAPRIO)
  - A Time-based scheduling algorithm
- Traffic class
  - Service device unit (SDU)
  - Frame preemption (FP)
  - Entry index (Index)

```c
static void add_tc_entries(struct nlmsghdr *n, __u32 max_sdu[TC_QOPT_MAX_QUEUE],
                           int num_max_sdu_entries, __u32 fp[TC_QOPT_MAX_QUEUE],
                           int num_fp_entries)
{
    struct rtattr *l;
    int num_tc;
    __u32 tc;

    num_tc = max(num_max_sdu_entries, num_fp_entries);

    for (tc = 0; tc < num_tc; tc++) {
        l = addattr_nest(n, 1024, TCA_TAPRIO_ATTR_TC_ENTRY | NLA_F_NESTED);

        addattr_l(n, 1024, TCA_TAPRIO_TC_ENTRY_INDEX, &tc, sizeof(tc));

        if (tc < num_max_sdu_entries) {
            addattr_l(n, 1024, TCA_TAPRIO_TC_ENTRY_MAX_SDU,
                      &max_sdu[tc], sizeof(max_sdu[tc]));
        }

        if (tc < num_fp_entries) {
            addattr_l(n, 1024, TCA_TAPRIO_TC_ENTRY_FP, &fp[tc],
                      sizeof(fp[tc]));
        }

        addattr_nest_end(n, l);
    }
}
```

Linux networking tool tc

$ The Bug

- When creating a TAPRIO qdisc, taprio_change is called
  - Internally, traffic classes will be parsed by taprio_parse_tc_entry

```
static int taprio_change(struct Qdisc *sch, struct nlattr *opt,
                         struct netlink_ext_ack *extack)
{

    err = nla_parse_nested_deprecated(tb, TCA_TAPRIO_ATTR_MAX, opt,
                                      taprio_policy, extack);
    if (err < 0)
        return err;

    // [...]

    err = taprio_parse_tc_entries(sch, opt, extack);
    if (err)
        return err;
}
```

```
static int taprio_parse_tc_entries(struct Qdisc *sch,
                                   struct nlattr *opt,
                                   struct netlink_ext_ack *extack)
{
    // [...]

    for (tc = 0; tc < TC_QOPT_MAX_QUEUE; tc++) {
        max_sdu[tc] = q->max_sdu[tc];
        fp[tc] = q->fp[tc];
    }

    nla_for_each_nested_type(n, TCA_TAPRIO_ATTR_TC_ENTRY, opt, rem) {
        err = taprio_parse_tc_entry(sch, n, max_sdu, fp, &seen_tcs,
                                    extack);
        if (err)
            return err;
    }
}
```

$ The Bug

- tapprio_parse_tc_entry tries to get entry index
  - The value of the entry index is uint32
  - But it assigned to an int32 variable
  - There is only a positive constant as the upper bound

```c
static const struct nla_policy taprio_tc_policy[TCA_TAPRIO_TC_ENTRY_MAX + 1] = {
	[TCA_TAPRIO_TC_ENTRY_INDEX]	   = { .type = NLA_U32 },
	[TCA_TAPRIO_TC_ENTRY_MAX_SDU]	   = { .type = NLA_U32 },
	[TCA_TAPRIO_TC_ENTRY_FP]	   = NLA_POLICY_RANGE(NLA_U32,
							  TC_FP_EXPRESS,
							  TC_FP_PREEMPTIBLE),
};
```

```c
static int taprio_parse_tc_entry(struct Qdisc *sch,
				 struct nlattr *opt,
				 u32 max_sdu[TC_QOPT_MAX_QUEUE],
				 u32 fp[TC_QOPT_MAX_QUEUE],
				 unsigned long *seen_tcs,
				 struct netlink_ext_ack *extack)
{
	struct nlattr *tb[TCA_TAPRIO_TC_ENTRY_MAX + 1] = { };
	int err, tc;
	// [...]

	err = nla_parse_nested(tb, TCA_TAPRIO_TC_ENTRY_MAX, opt,
			       taprio_tc_policy, extack);
	if (err < 0)
		return err;

	tc = nla_get_u32(tb[TCA_TAPRIO_TC_ENTRY_INDEX]);
	if (tc >= TC_QOPT_MAX_QUEUE /* 16 */) {
		NL_SET_ERR_MSG_MOD(extack, "TC entry index out of range");
		return -ERANGE;
	}
```

$ The Bug

- tapriо_parse_tc_entry tries to get entry index
  - The value of the entry index is uint32
  - But it assigned to an int32 variable
  - There is only a positive constant as the upper bound

```c
static const struct nla_policy taprio_tc_policy[TCA_TAPRIO_TC_ENTRY_MAX + 1] = {
	[TCA_TAPRIO_TC_ENTRY_INDEX]	   = { .type = NLA_U32 },
	[TCA_TAPRIO_TC_ENTRY_MAX_SDU]	   = { .type = NLA_U32 },
	[TCA_TAPRIO_TC_ENTRY_FP]	   = NLA_POLICY_RANGE(NLA_U32,
							      TC_FP_EXPRESS,
							      TC_FP_PREEMPTIBLE),
};
```

```c
static int taprio_parse_tc_entry(struct Qdisc *sch,
				 struct nlattr *opt,
				 u32 max_sdu[TC_QOPT_MAX_QUEUE],
				 u32 fp[TC_QOPT_MAX_QUEUE],
				 unsigned long *seen_tcs,
				 struct netlink_ext_ack *extack)
{
	struct nlattr *tb[TCA_TAPRIO_TC_ENTRY_MAX + 1] = { };
	int err, tc;
	// [...]

	err = nla_parse_nested(tb, TCA_TAPRIO_TC_ENTRY_MAX, opt,
			       taprio_tc_policy, extack);
	if (err < 0)
		return err;

	tc = nla_get_u32(tb[TCA_TAPRIO_TC_ENTRY_INDEX]);
	if (tc >= TC_QOPT_MAX_QUEUE /* 16 */) {
		NL_SET_ERR_MSG_MOD(extack, "TC entry index out of range");
		return -ERANGE;
	}
```

$ The Bug

- taprio_parse_tc_entry tries to get entry index
  - The value of the entry index is uint32
  - But it assigned to an int32 variable
  - There is only a positive constant as the upper bound
- What happens if we assign a negative integer to it?

```c
static const struct nla_policy taprio_tc_policy[TCA_TAPRIO_TC_ENTRY_MAX + 1] = {
	[TCA_TAPRIO_TC_ENTRY_INDEX]	= { .type = NLA_U32 },
	[TCA_TAPRIO_TC_ENTRY_MAX_SDU]	= { .type = NLA_U32 },
	[TCA_TAPRIO_TC_ENTRY_FP]	= NLA_POLICY_RANGE(NLA_U32,
							   TC_FP_EXPRESS,
							   TC_FP_PREEMPTIBLE),
};
```

```c
static int taprio_parse_tc_entry(struct Qdisc *sch,
				 struct nlattr *opt,
				 u32 max_sdu[TC_QOPT_MAX_QUEUE],
				 u32 fp[TC_QOPT_MAX_QUEUE],
				 unsigned long *seen_tcs,
				 struct netlink_ext_ack *extack)
{
	struct nlattr *tb[TCA_TAPRIO_TC_ENTRY_MAX + 1] = { };
	int err, tc;
	// [...]

	err = nla_parse_nested(tb, TCA_TAPRIO_TC_ENTRY_MAX, opt,
			       taprio_tc_policy, extack);
	if (err < 0)
		return err;


	tc = nla_get_u32(tb[TCA_TAPRIO_TC_ENTRY_INDEX]);
	if (tc >= TC_QOPT_MAX_QUEUE /* 16 */) {
		NL_SET_ERR_MSG_MOD(extack, "TC entry index out of range");
		return -ERANGE;
	}
```

```
[  807.835821] BUG: unable to handle page fault for address: ffffc9000009dcf0
[  807.835821] #PF: supervisor write access in kernel mode
[  807.835821] #PF: error_code(0x0002) - not-present page
[  807.835821] PGD 3400067 P4D 3400067 PUD 35d5067 PMD 35d6067 PTE 0
[  807.835821] Oops: 0002 [#1] PREEMPT SMP PTI
[  807.835821] CPU: 0 PID: 127 Comm: tc_dyn Not tainted 6.1.73 #4
[  807.835821] Hardware name: QEMU Standard PC (i440FX + PIIX, 1996), BIOS 1.16.0-debian-1.16.0-5 04/01/2014
[  807.835821] RIP: 0010:taprio_parse_tc_entries+0x1df/0x2a0
[  807.835821] Code: 72 3a b8 01 00 00 00 48 d3 e0 49 09 c7 48 8b 44 24 18 48 85 c0 74 21 48 8b 34 24 8b 40 04
[  807.835821] RSP: 0018:ffffc900000ff750 EFLAGS: 00000246
[  807.835821] RAX: 0000000000000000 RBX: ffff88800506a800 RCX: ffffffffffffe7960
[  807.835821] RDX: ffffc900000ffae0 RSI: ffff888005165000 RDI: ffffffff820bd180
[  807.835821] RBP: ffff888005165000 R08: 0000000000000003 R09: 0000000000000004
[  807.835821] R10: 0000000000000002 R11: ffffffffffffffff R12: ffff88800514ba8c
[  807.835821] R13: 0000000000000018 R14: ffffc900000ffae0 R15: 0000000100000000
[  807.835821] FS:  00007f41650c1440(0000) GS:ffff88800f200000(0000) knlGS:0000000000000000
[  807.835821] CS:  0010 DS: 0000 ES: 0000 CR0: 0000000080050033
[  807.835821] CR2: ffffc9000009dcf0 CR3: 000000000536c000 CR4: 00000000003006f0
[  807.835821] Call Trace:
[  807.835821]  <TASK>
[  807.835821]  ? __die_body.cold+0x1a/0x1f
[  807.835821]  ? page_fault_oops+0xd2/0x290
[  807.835821]  qdisc_create+0x1d7/0x510
[  807.835821]  tc_modify_qdisc+0x3fc/0x830
[  807.835821]  rtnetlink_rcv_msg+0x14e/0x3b0
[  807.835821]  ? __kmem_cache_alloc_node+0x156/0x290
[  807.835821]  ? __alloc_skb+0x88/0x1a0
[  807.835821]  ? rtnl_calcit.isra.0+0x140/0x140
[  807.835821]  netlink_rcv_skb+0x51/0x100
[  807.835821]  netlink_unicast+0x24a/0x390
[  807.835821]  netlink_sendmsg+0x250/0x4c0
[  807.835821]  __sock_sendmsg+0x5f/0x70
[  807.835821]  ____sys_sendmsg+0x231/0x260
[  807.835821]  ? copy_msghdr_from_user+0x7d/0xc0
[  807.835821]  ___sys_sendmsg+0x96/0xd0
[  807.835821]  __sys_sendmsg+0x6e/0xb0
[  807.835821]  do_syscall_64+0x5b/0x80
[  807.835821]  ? fpregs_assert_state_consistent+0x22/0x50
[  807.835821]  ? exit_to_user_mode_prepare+0x37/0x110
[  807.835821]  ? syscall_exit_to_user_mode+0x2b/0x50
[  807.835821]  ? do_syscall_64+0x67/0x80
[  807.835821]  ? fpregs_assert_state_consistent+0x22/0x50
[  807.835821]  ? exit_to_user_mode_prepare+0x37/0x110
[  807.835821]  entry_SYSCALL_64_after_hwframe+0x64/0xce
```

Boom! An out-of-bounds access occurs!

$ The Bug

- The tc tool can't trigger this bug because the entry index is auto-assigned
- Prevent the bug from being easily discovered

```
static void add_tc_entries(struct nlmsghdr *n, __u32 max_sdu[TC_QOPT_MAX_QUEUE],
                           int num_max_sdu_entries, __u32 fp[TC_QOPT_MAX_QUEUE],
                           int num_fp_entries)
{
    struct rtattr *l;
    int num_tc;
    __u32 tc;

    num_tc = max(num_max_sdu_entries, num_fp_entries);

    for (tc = 0; tc < num_tc; tc++) {
        l = addattr_nest(n, 1024, TCA_TAPRIO_ATTR_TC_ENTRY | NLA_F_NESTED);

        addattr_l(n, 1024, TCA_TAPRIO_TC_ENTRY_INDEX, &tc, sizeof(tc));

        if (tc < num_max_sdu_entries) {
            addattr_l(n, 1024, TCA_TAPRIO_TC_ENTRY_MAX_SDU,
                      &max_sdu[tc], sizeof(max_sdu[tc]));
        }

        if (tc < num_fp_entries) {
            addattr_l(n, 1024, TCA_TAPRIO_TC_ENTRY_FP, &fp[tc],
                      sizeof(fp[tc]));
        }

        addattr_nest_end(n, l);
    }
}
```

Linux networking tool tc

- Nov 28 2023 Target Selection
- Jan 19 2024 Bug Discovery
- Feb 21 2024 Crafting the Exploit
- Mar 20 2024 Achieving LPE
- Nov 7 2024 Takeaways

$ Analysis

- The entry index is used to access two arrays: max_sdu and fp

```c
static int taprio_parse_tc_entry(/*...*/
                                 u32 max_sdu[TC_QOPT_MAX_QUEUE],
                                 u32 fp[TC_QOPT_MAX_QUEUE],
                                 /*...*/)
{
    struct nlattr *tb[TCA_TAPRIO_TC_ENTRY_MAX + 1] = { };
    struct net_device *dev = qdisc_dev(sch);
    int err, tc;
    u32 val;

    // [...]

    if (tb[TCA_TAPRIO_TC_ENTRY_MAX_SDU]) {
        val = nla_get_u32(tb[TCA_TAPRIO_TC_ENTRY_MAX_SDU]);
        if (val > dev->max_mtu) {
            NL_SET_ERR_MSG_MOD(extack, "TC max SDU exceeds device max MTU");
            return -ERANGE;
        }

        max_sdu[tc] = val;
    }

    if (tb[TCA_TAPRIO_TC_ENTRY_FP])
        fp[tc] = nla_get_u32(tb[TCA_TAPRIO_TC_ENTRY_FP]);

    return 0;
}
```

$ Analysis

- The entry index is used to access two arrays: max_sdu and fp
- Both are passed as parameters and are declared on the stack

```
static int taprio_parse_tc_entries(struct Qdisc *sch,
                                   struct nlattr *opt,
                                   struct netlink_ext_ack *extack)
{
	// [...]
	u32 max_sdu[TC_QOPT_MAX_QUEUE];
	u32 fp[TC_QOPT_MAX_QUEUE];
	// [...]

	nla_for_each_nested(n, opt, rem) {
		if (nla_type(n) != TCA_TAPRIO_ATTR_TC_ENTRY)
			continue;

		err = taprio_parse_tc_entry(sch, n, max_sdu, fp, &seen_tcs,
					    extack);
		if (err)
			return err;
	}
```

$ Analysis

- The entry index is used to access two arrays: max_sdu and fp
- Both are passed as parameters and are declared on the stack
- The OOB access can be triggered multiple times

```
static int taprio_parse_tc_entries(struct Qdisc *sch,
                                   struct nlattr *opt,
                                   struct netlink_ext_ack *extack)
{
    // [...]
    u32 max_sdu[TC_QOPT_MAX_QUEUE];
    u32 fp[TC_QOPT_MAX_QUEUE];
    // [...]

    nla_for_each_nested(n, opt, rem) {
        if (nla_type(n) != TCA_TAPRIO_ATTR_TC_ENTRY)
            continue;

        err = taprio_parse_tc_entry(sch, n, max_sdu, fp, &seen_tcs,
                                    extack);
        if (err)
            return err;
    }
```

$ Analysis

- The entry index is used to access two arrays: max_sdu and fp
- Both are passed as parameters and are declared on the stack
- The OOB access can be triggered multiple times
- It looks promising, right?

```
static int taprio_parse_tc_entries(struct Qdisc *sch,
                                   struct nlattr *opt,
                                   struct netlink_ext_ack *extack)
{
    // [...]
    u32 max_sdu[TC_QOPT_MAX_QUEUE];
    u32 fp[TC_QOPT_MAX_QUEUE];
    // [...]

    nla_for_each_nested(n, opt, rem) {
        if (nla_type(n) != TCA_TAPRIO_ATTR_TC_ENTRY)
            continue;

        err = taprio_parse_tc_entry(sch, n, max_sdu, fp, &seen_tcs,
                                    extack);
        if (err)
            return err;
    }
```

$ Restriction

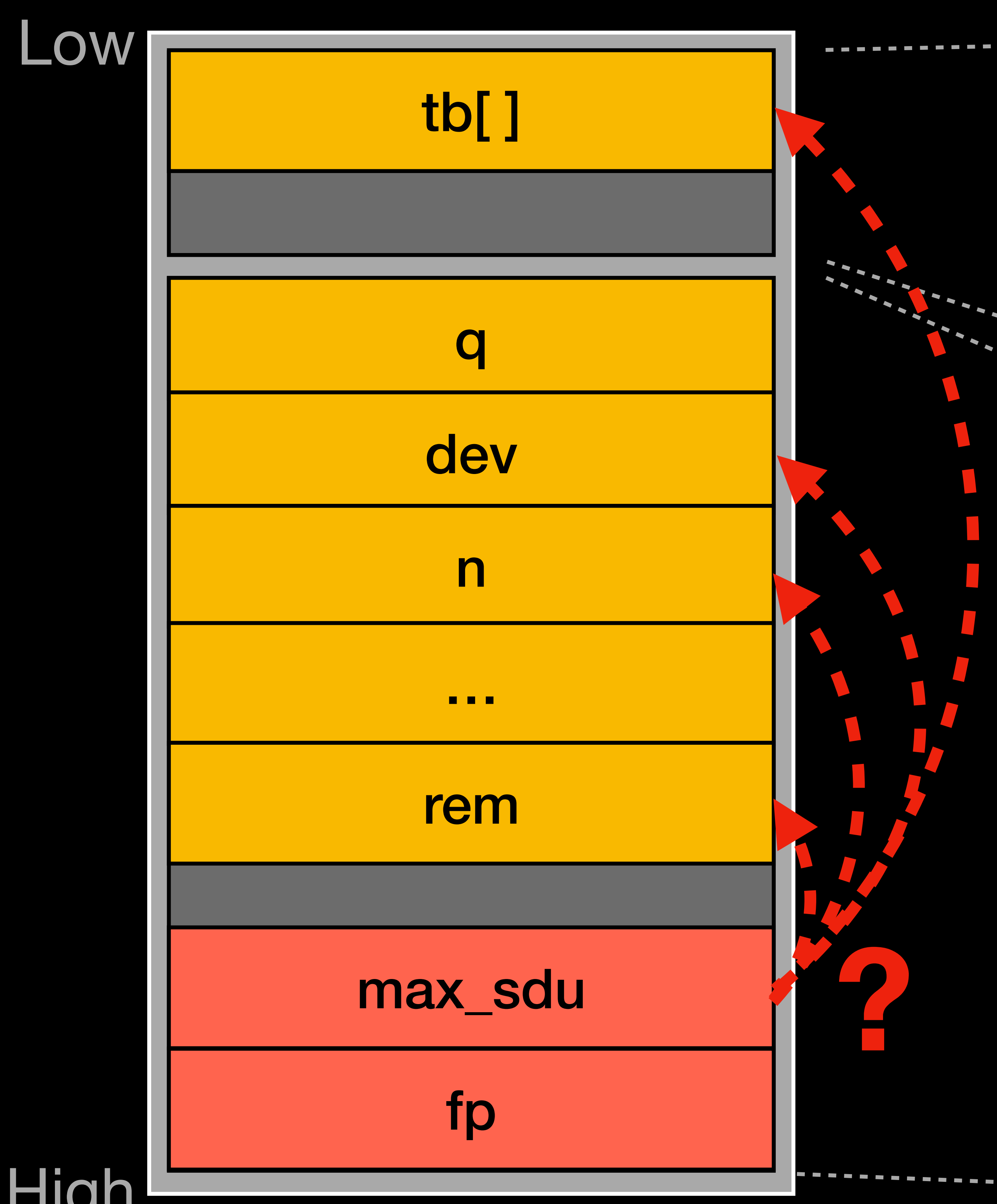
- Restrictions
  - max_sdu - cannot exceed device's MTU
  - fp - only 1 or 2 according to policy
- After reviewing the source code, we found the largest MTU is about 65535

```c
static int taprio_parse_tc_entry(
                /* [...] */
                u32 max_sdu[TC_QOPT_MAX_QUEUE],
                u32 fp[TC_QOPT_MAX_QUEUE],
                /* [...] */)
{
    int tc;
    u32 val;

    // [...]

    tc = nla_get_u32(tb[TCA_TAPRIO_TC_ENTRY_INDEX]);
    if (tb[TCA_TAPRIO_TC_ENTRY_MAX_SDU]) {
        val = nla_get_u32(tb[TCA_TAPRIO_TC_ENTRY_MAX_SDU]);
        if (val > dev->max_mtu) {
            NL_SET_ERR_MSG_MOD(extack, "TC max SDU exceeds device max MTU");
            return -ERANGE;
        }

        max_sdu[tc] = val;
    }

    if (tb[TCA_TAPRIO_TC_ENTRY_FP])
        fp[tc] = nla_get_u32(tb[TCA_TAPRIO_TC_ENTRY_FP]);

    return 0;
}
```

```c
static const struct nla_policy taprio_tc_policy[TCA_TAPRIO_TC_ENTRY_MAX + 1] = {
    [TCA_TAPRIO_TC_ENTRY_INDEX]    = { .type = NLA_U32 },
    [TCA_TAPRIO_TC_ENTRY_MAX_SDU]      = { .type = NLA_U32 },
    [TCA_TAPRIO_TC_ENTRY_FP]        = NLA_POLICY_RANGE(NLA_U32,
                                TC_FP_EXPRESS,
                                TC_FP_PREEMPTIBLE),
};
```

$ Restriction

Which variables are candidates for overwriting?



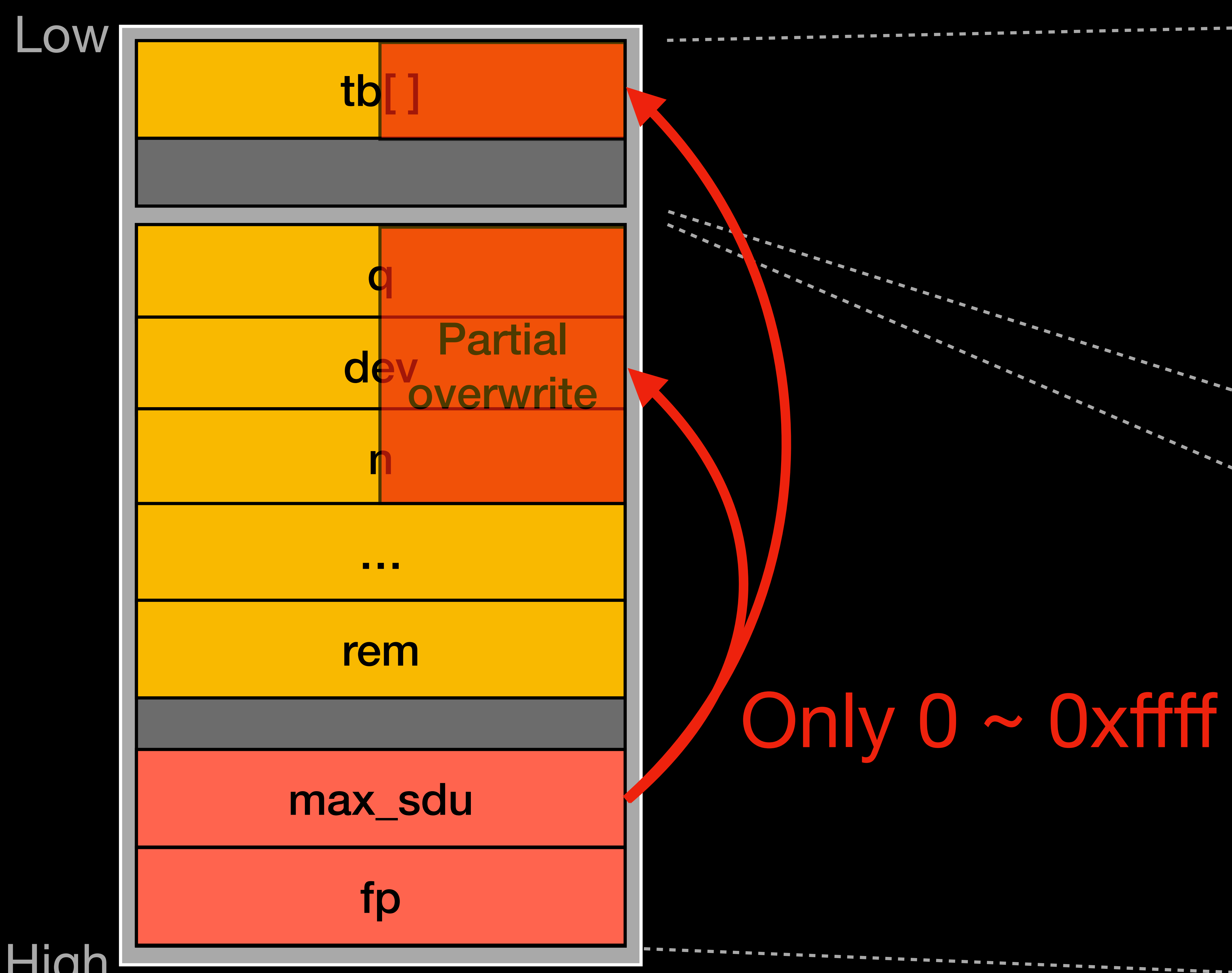
Stack layout

```c
static int taprio_parse_tc_entry(struct Qdisc *sch,
                                 struct nlattr *opt,
                                 u32 max_sdu[TC_QOPT_MAX_QUEUE],
                                 u32 fp[TC_QOPT_MAX_QUEUE],
                                 unsigned long *seen_tcs,
                                 struct netlink_ext_ack *extack)
{
    struct nlattr *tb[TCA_TAPRIO_TC_ENTRY_MAX + 1] = { };
    struct net_device *dev = qdisc_dev(sch);
    int err, tc;
    u32 val;
```

```c
static int taprio_parse_tc_entries(struct Qdisc *sch,
                                   struct nlattr *opt,
                                   struct netlink_ext_ack *extack)
{
    struct taprio_sched *q = qdisc_priv(sch);
    struct net_device *dev = qdisc_dev(sch);
    u32 max_sdu[TC_QOPT_MAX_QUEUE];
    bool have_preemption = false;
    unsigned long seen_tcs = 0;
    u32 fp[TC_QOPT_MAX_QUEUE];
    struct nlattr *n;
    int tc, rem;
    int err = 0;
```

$ Restriction

Which variables are candidates for overwriting?



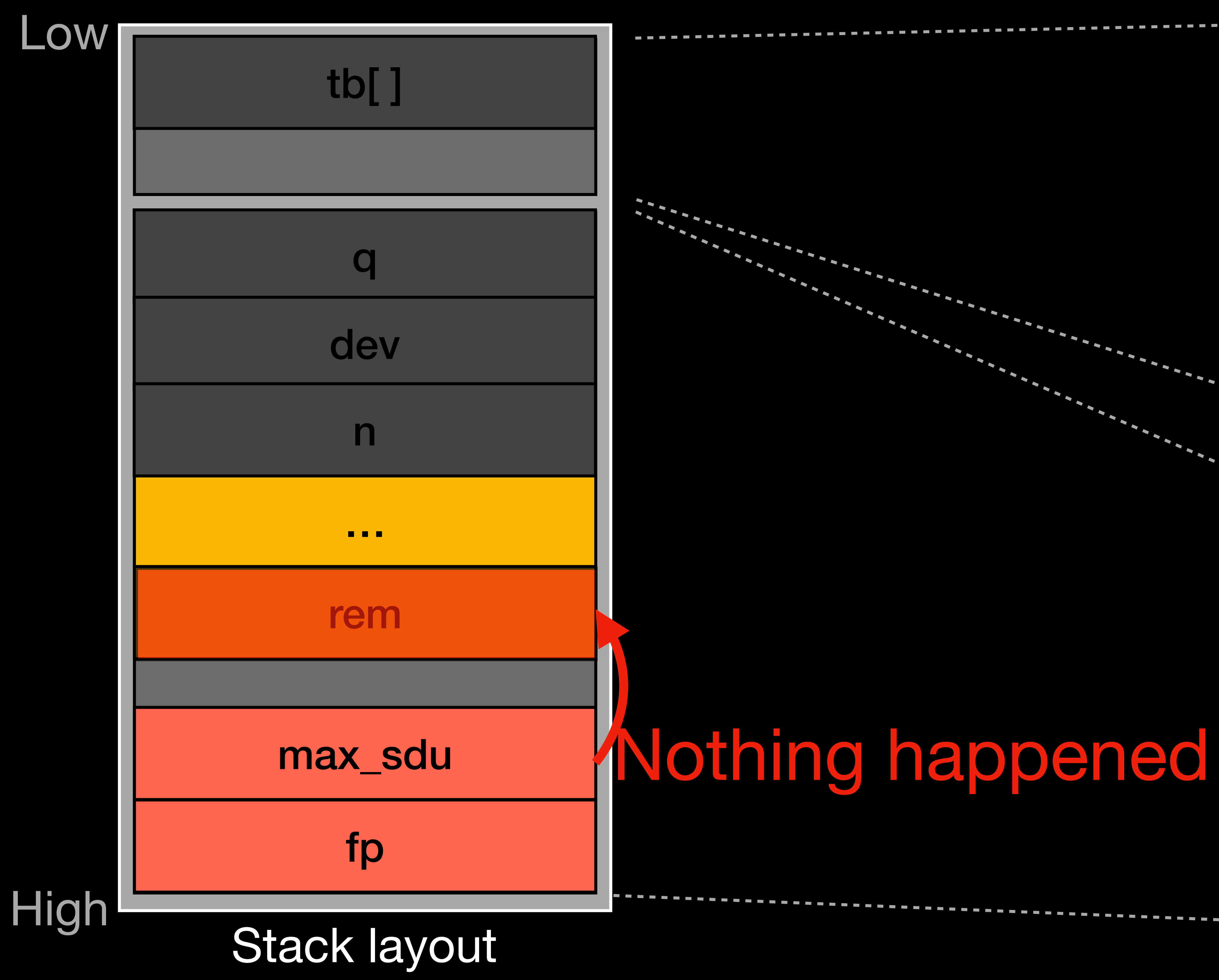
Stack layout

```c
static int taprio_parse_tc_entry(struct Qdisc *sch,
				 struct nlattr *opt,
				 u32 max_sdu[TC_QOPT_MAX_QUEUE],
				 u32 fp[TC_QOPT_MAX_QUEUE],
				 unsigned long *seen_tcs,
				 struct netlink_ext_ack *extack)
{
	struct nlattr *tb[TCA_TAPRIO_TC_ENTRY_MAX + 1] = { };
	struct net_device *dev = qdisc_dev(sch);
	int err, tc;
	u32 val;
```

```c
static int taprio_parse_tc_entries(struct Qdisc *sch,
				   struct nlattr *opt,
				   struct netlink_ext_ack *extack)
{
	struct taprio_sched *q = qdisc_priv(sch);
	struct net_device *dev = qdisc_dev(sch);
	u32 max_sdu[TC_QOPT_MAX_QUEUE];
	bool have_preemption = false;
	unsigned long seen_tcs = 0;
	u32 fp[TC_QOPT_MAX_QUEUE];
	struct nlattr *n;
	int tc, rem;
	int err = 0;
```

$ Restriction

Which variables are candidates for overwriting?

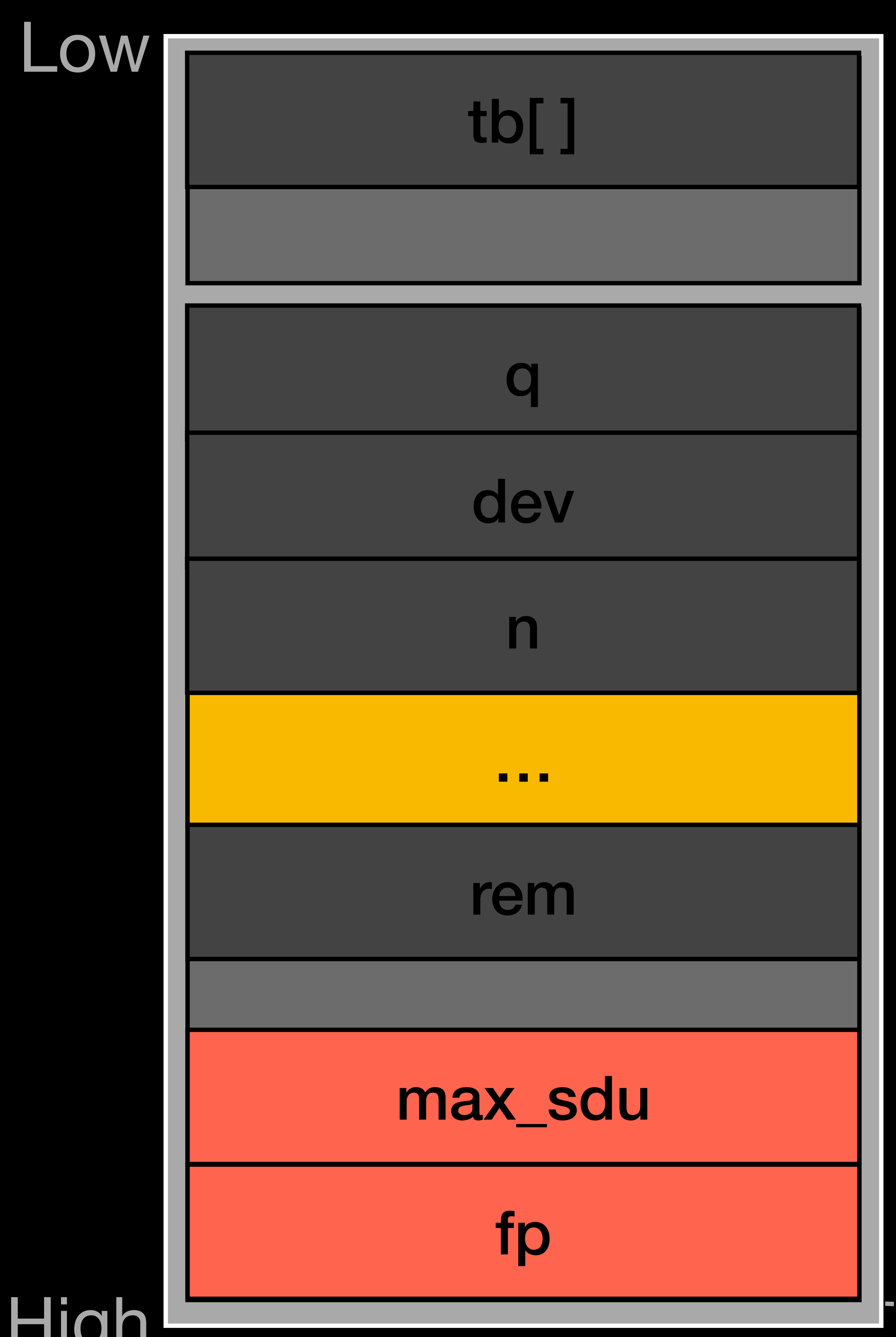



```c
static int taprio_parse_tc_entry(struct Qdisc *sch,
                                 struct nlattr *opt,
                                 u32 max_sdu[TC_QOPT_MAX_QUEUE],
                                 u32 fp[TC_QOPT_MAX_QUEUE],
                                 unsigned long *seen_tcs,
                                 struct netlink_ext_ack *extack)
{
    struct nlattr *tb[TCA_TAPRIO_TC_ENTRY_MAX + 1] = { };
    struct net_device *dev = qdisc_dev(sch);
    int err, tc;
    u32 val;
```

```c
static int taprio_parse_tc_entries(struct Qdisc *sch,
                                   struct nlattr *opt,
                                   struct netlink_ext_ack *extack)
{
    struct taprio_sched *q = qdisc_priv(sch);
    struct net_device *dev = qdisc_dev(sch);
    u32 max_sdu[TC_QOPT_MAX_QUEUE];
    bool have_preemption = false;
    unsigned long seen_tcs = 0;
    u32 fp[TC_QOPT_MAX_QUEUE];
    struct nlattr *n;
    int tc, rem;
    int err = 0;
```

$ Restriction

No… 🫠



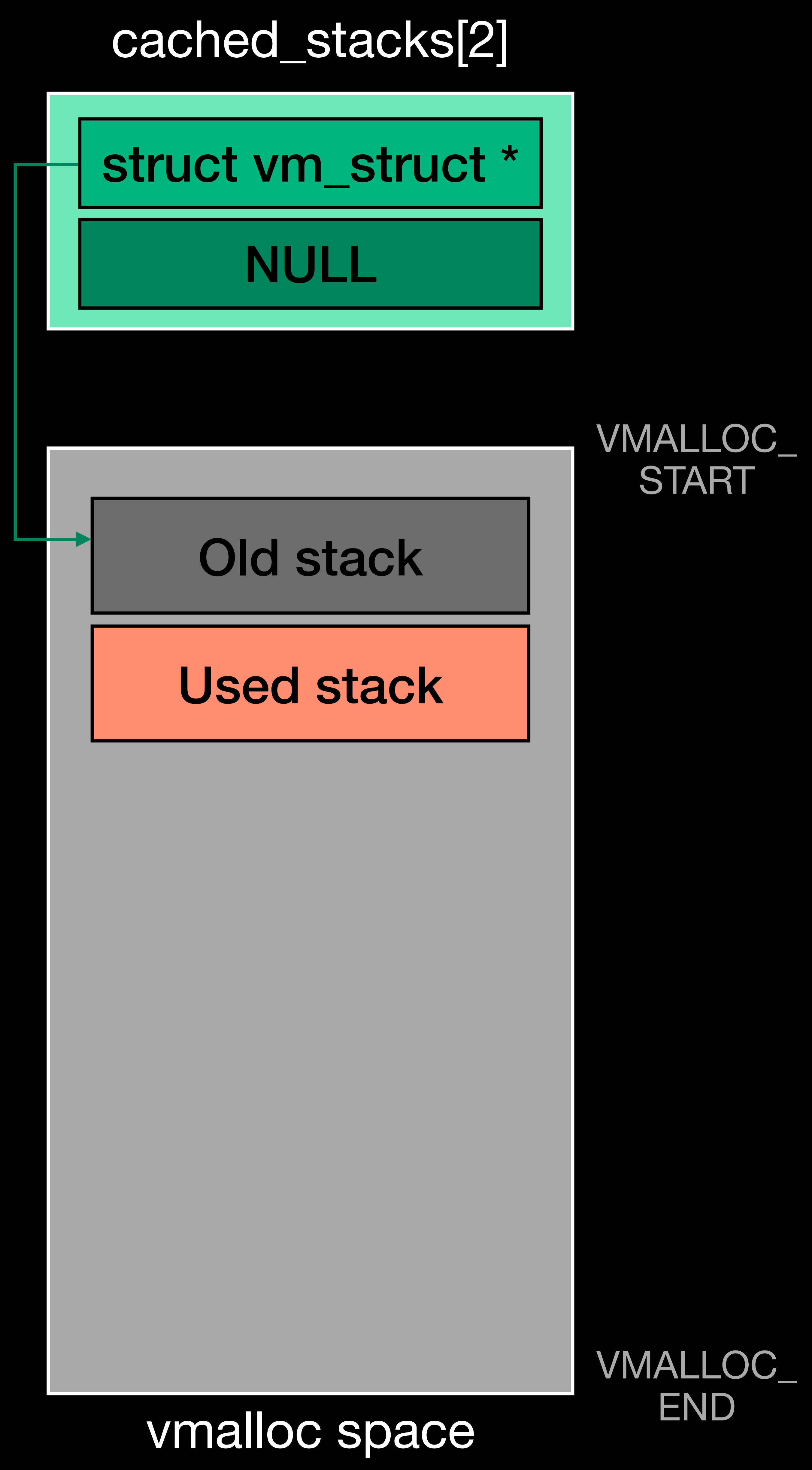
Stack layout

```c
static int taprio_parse_tc_entry(struct Qdisc *sch,
                                 struct nlattr *opt,
                                 u32 max_sdu[TC_QOPT_MAX_QUEUE],
                                 u32 fp[TC_QOPT_MAX_QUEUE],
                                 unsigned long *seen_tcs,
                                 struct netlink_ext_ack *extack)
{
    struct nlattr *tb[TCA_TAPRIO_TC_ENTRY_MAX + 1] = { };
    struct net_device *dev = qdisc_dev(sch);
    int err, tc;
    u32 val;
```

```c
static int taprio_parse_tc_entries(struct Qdisc *sch,
                                   struct nlattr *opt,
                                   struct netlink_ext_ack *extack)
{
    struct taprio_sched *q = qdisc_priv(sch);
    struct net_device *dev = qdisc_dev(sch);
    u32 max_sdu[TC_QOPT_MAX_QUEUE];
    bool have_preemption = false;
    unsigned long seen_tcs = 0;
    u32 fp[TC_QOPT_MAX_QUEUE];
    struct nlattr *n;
    int tc, rem;
    int err = 0;
```

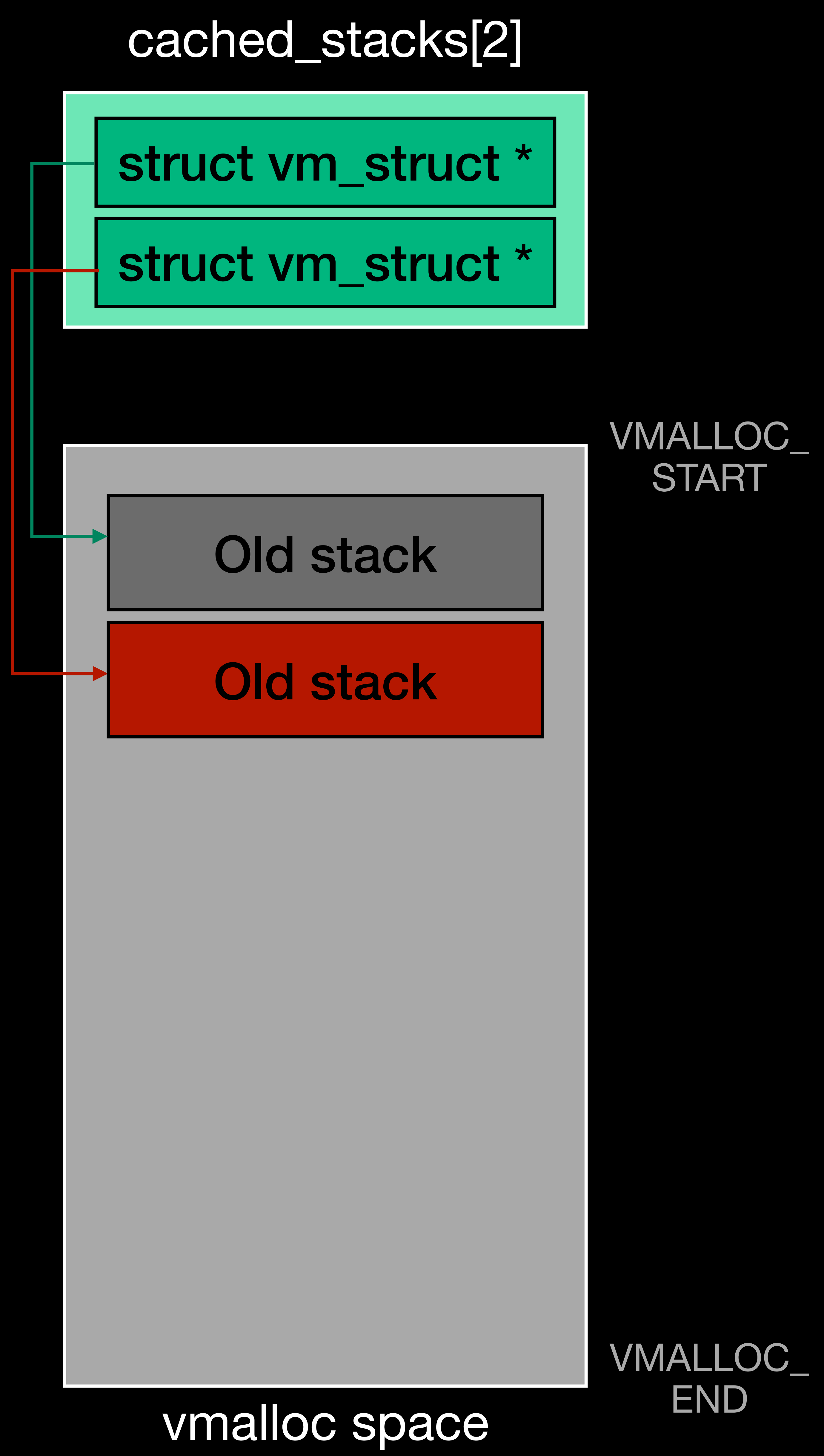
$ Allocate A Stack

- The kernel stack is allocated by alloc_thread_stack_node
- First, it attempts to reuse the old stack from the cache



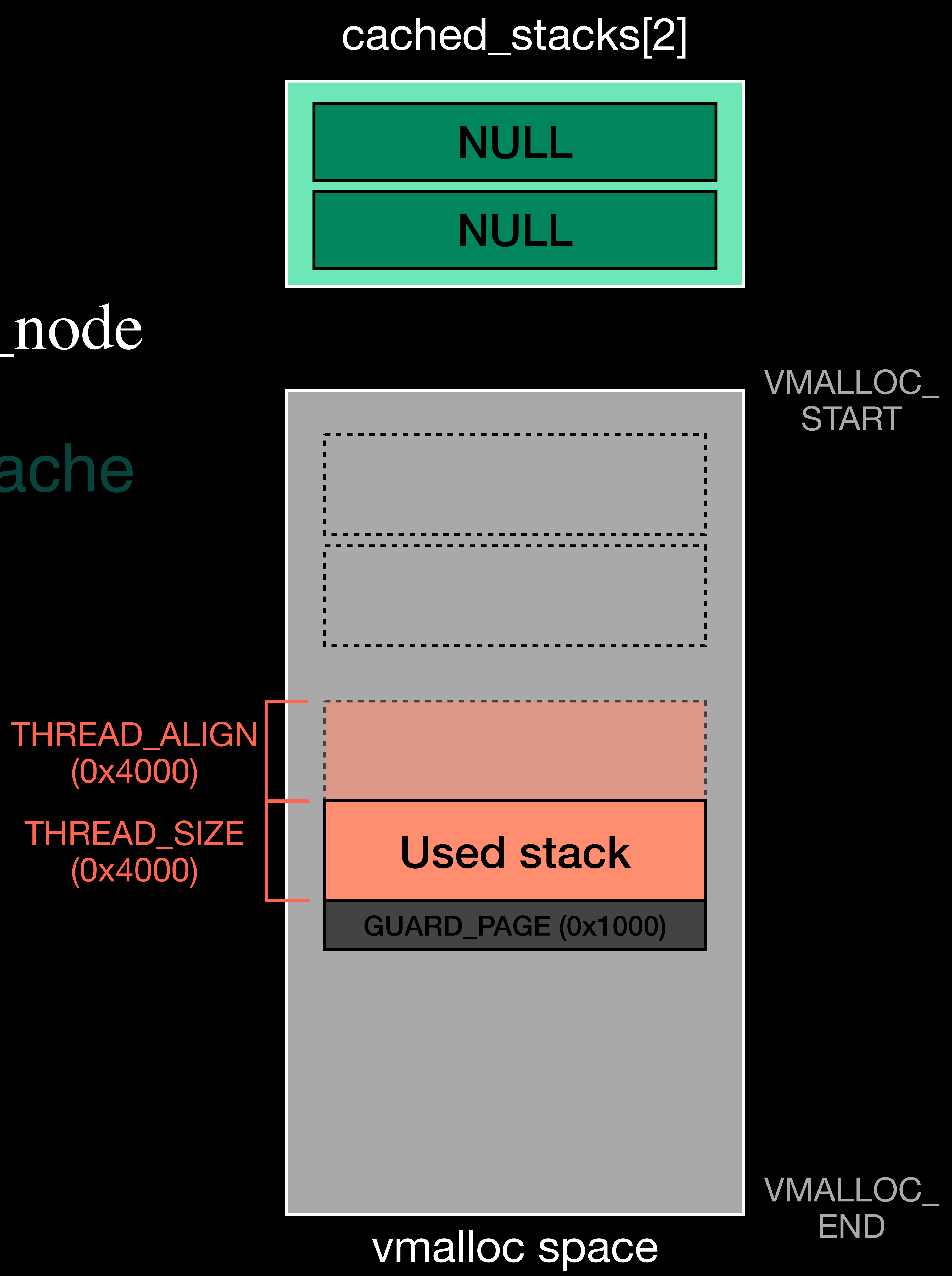
$ Allocate A Stack

- The kernel stack is allocated by alloc_thread_stack_node
- First, it attempts to reuse the old stack from the cache
    - Cache is refilled when old processes exit



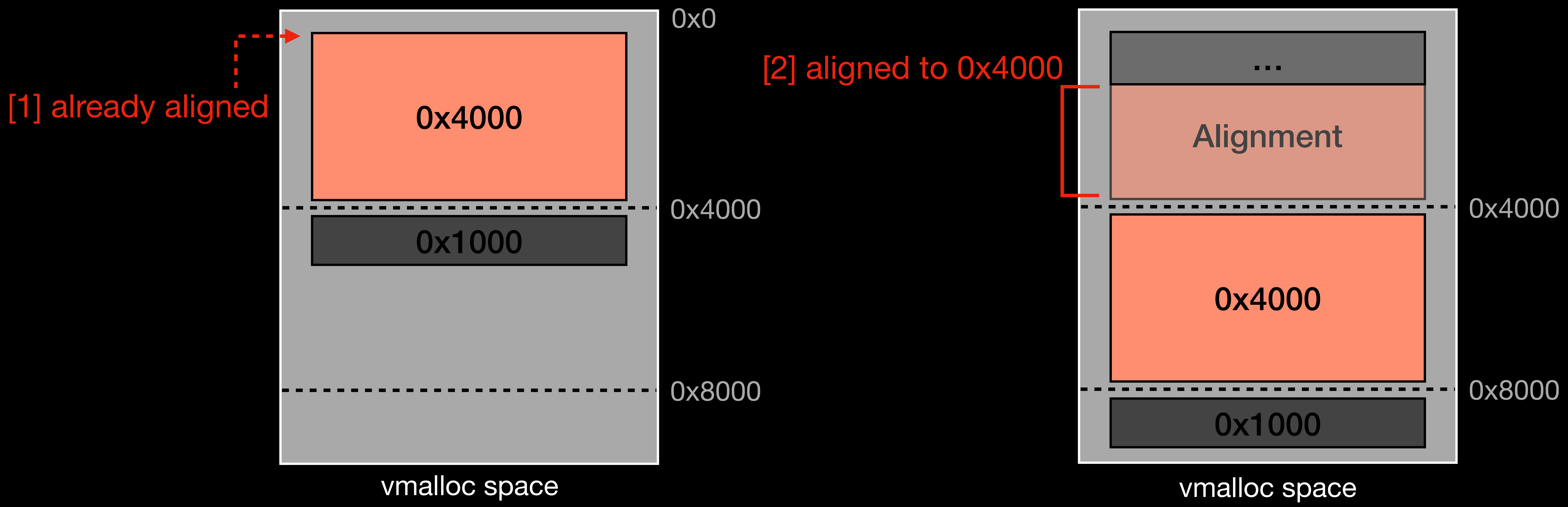
$ Allocate A Stack

- The kernel stack is allocated by alloc_thread_stack_node
- First, it attempts to reuse the old stack from the cache
  - Cache is refilled when old processes exit
- If it failed, it calls vmalloc to allocate a new one
  - Alignment: 0x4000
  - Size: 0x4000
  - Guard page: 0x1000



$ Allocate A Stack

- Three key points when vmalloc-ing a stack
  1. After 0x4000 alignment, the memory has two different layouts

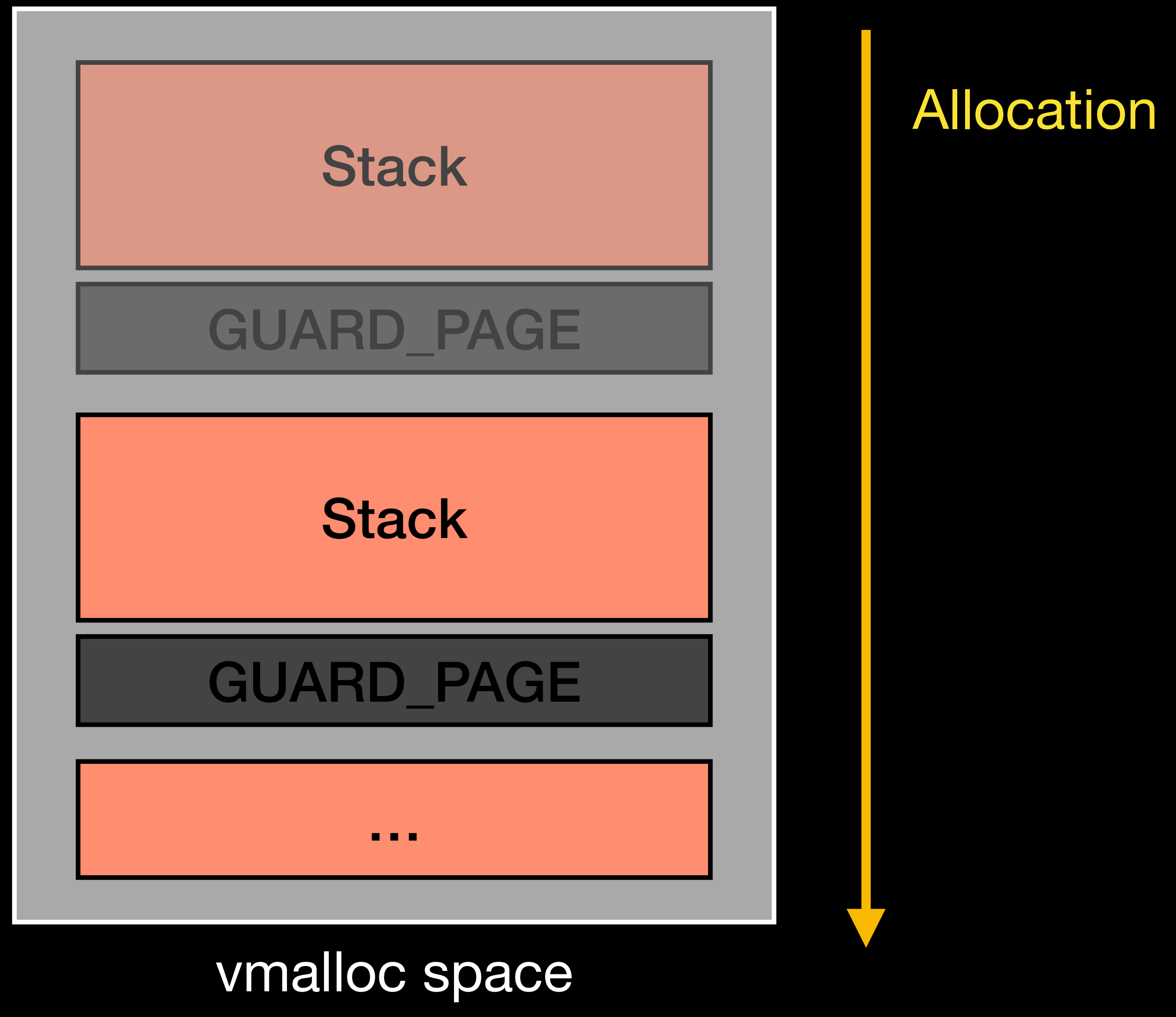


$ Allocate A Stack

- Three key points when vmalloc-ing a stack

1. After 0x4000 alignment, the memory has two different layouts

2. Memory regions allocated from the vmalloc space will be sequential



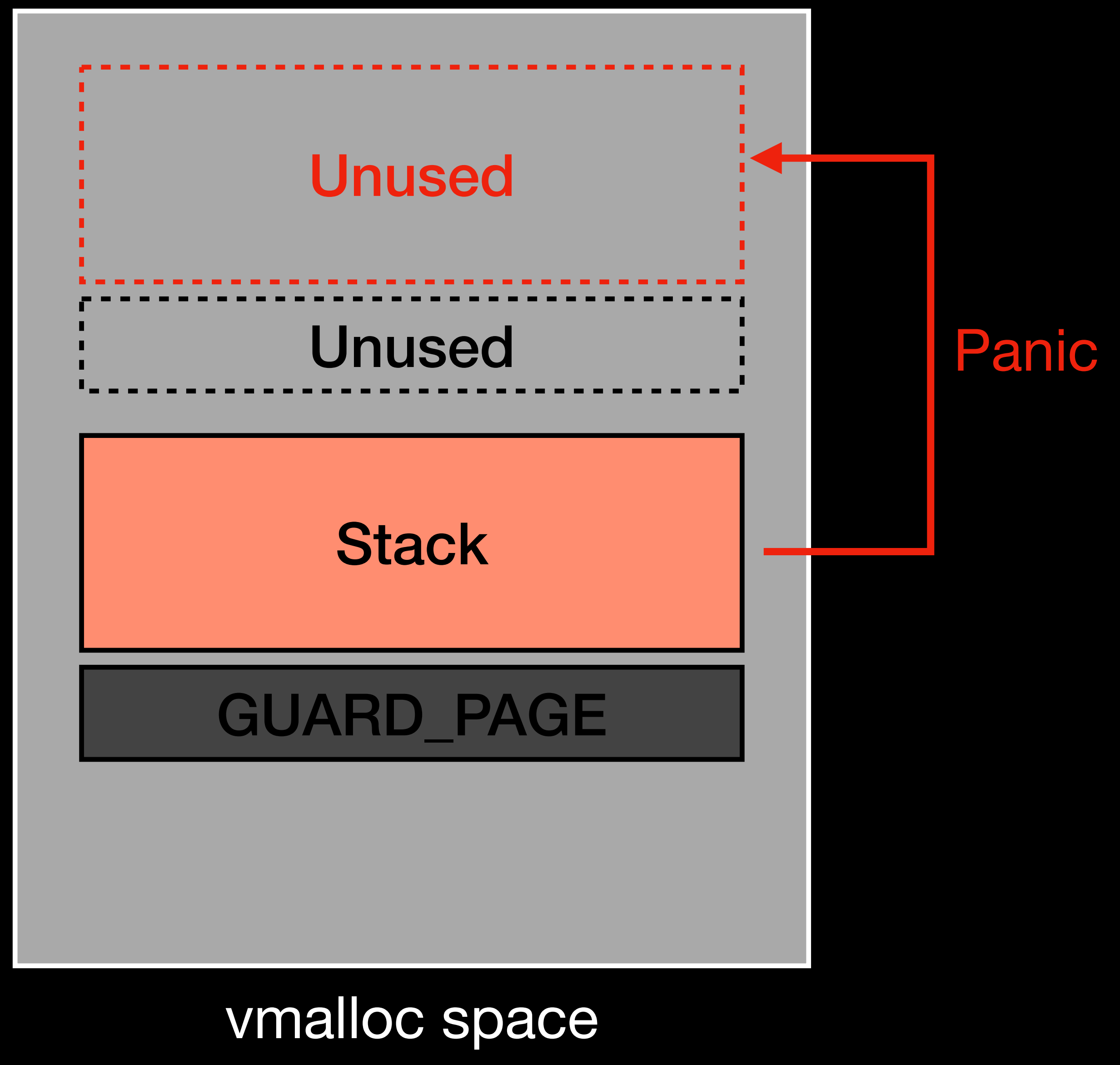
$ Allocate A Stack

- Three key points when vmalloc-ing a stack

   1. After 0x4000 alignment, the memory has two different layouts

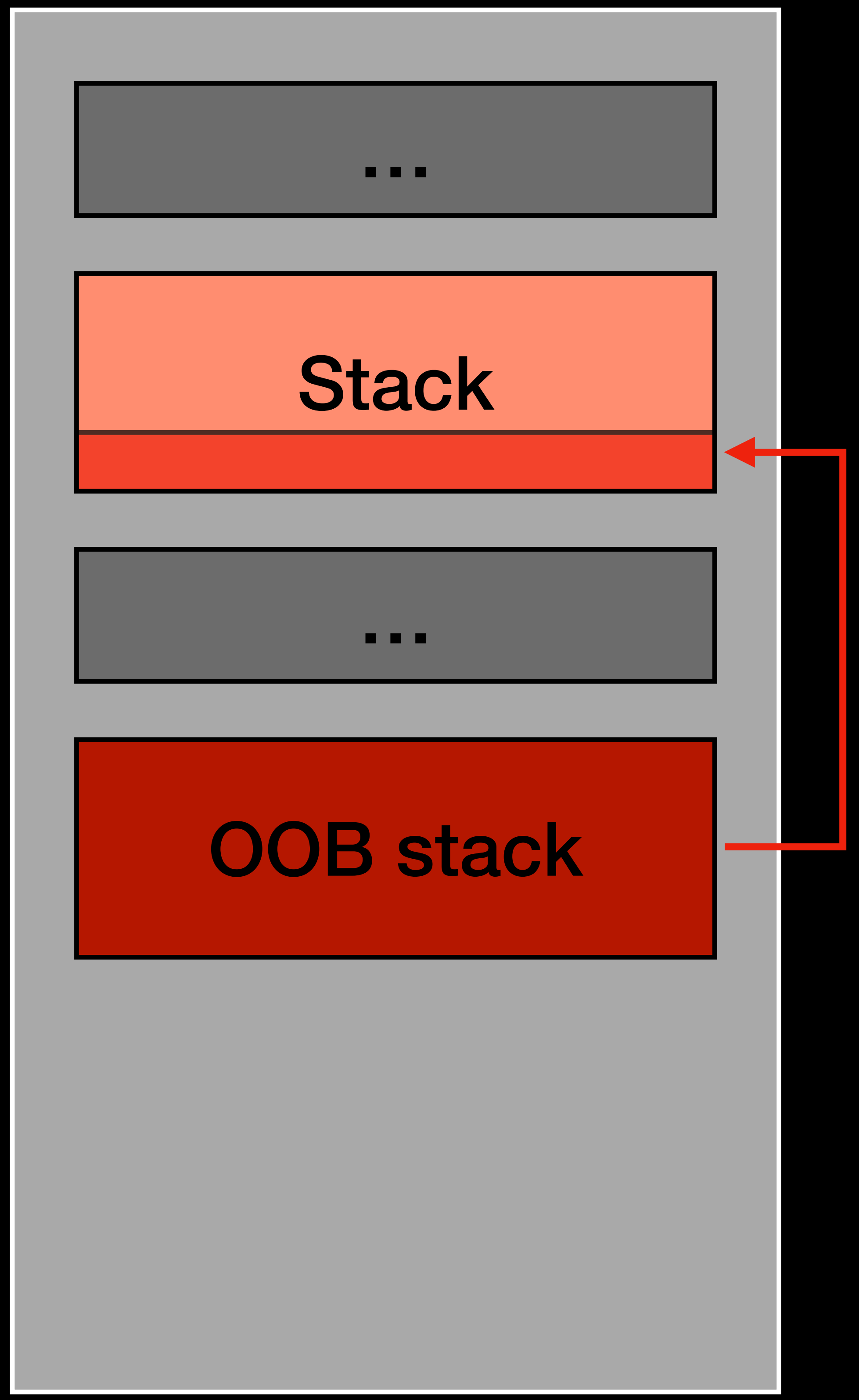
   2. Memory regions allocated from the vmalloc space will be sequential

   3. The chunk will become unmapped after being released



$ Ideas

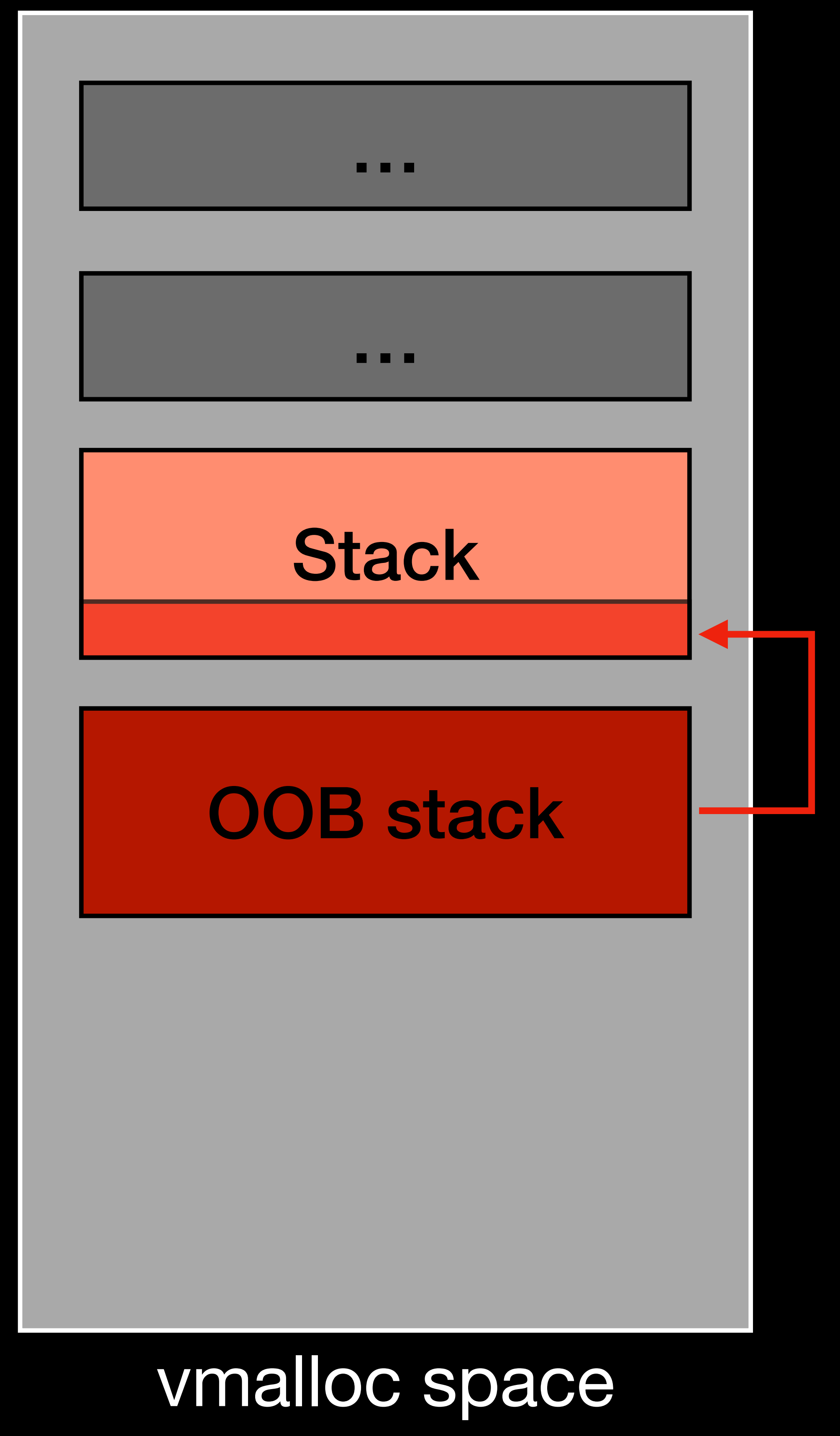
- Overwrite data in another stack
  1. Spawn the victim process before the OOB process
  2. The victim process performs a extended action
  3. The OOB process overwrites the victim process stack



vmalloc space

$ Ideas

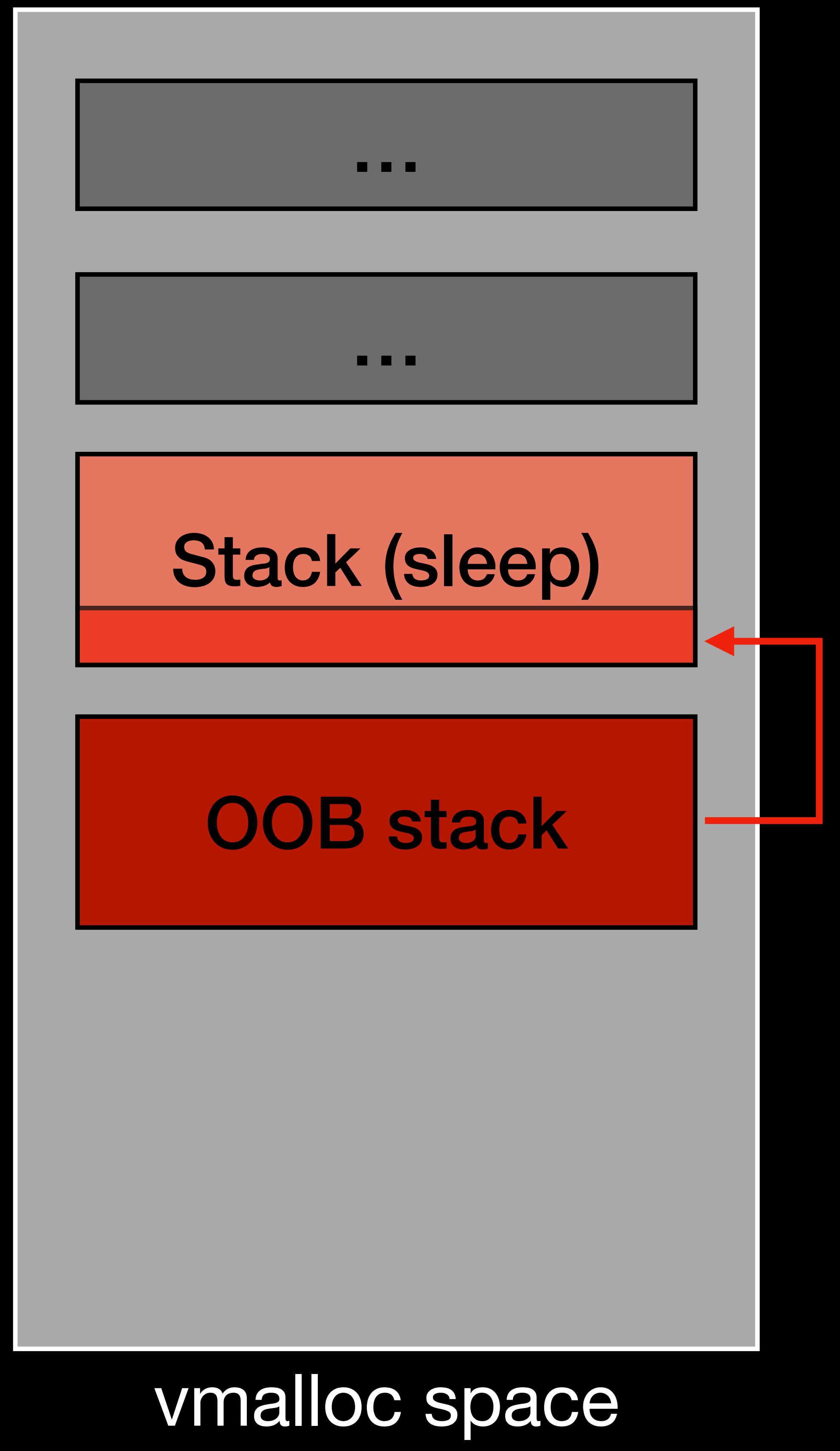
- Overwrite data in another stack
    1. Spawn the victim process before the OOB process
    2. The victim process performs a extended action
    3. The OOB process overwrites the victim process stack



$ Ideas

- Overwrite data in another stack

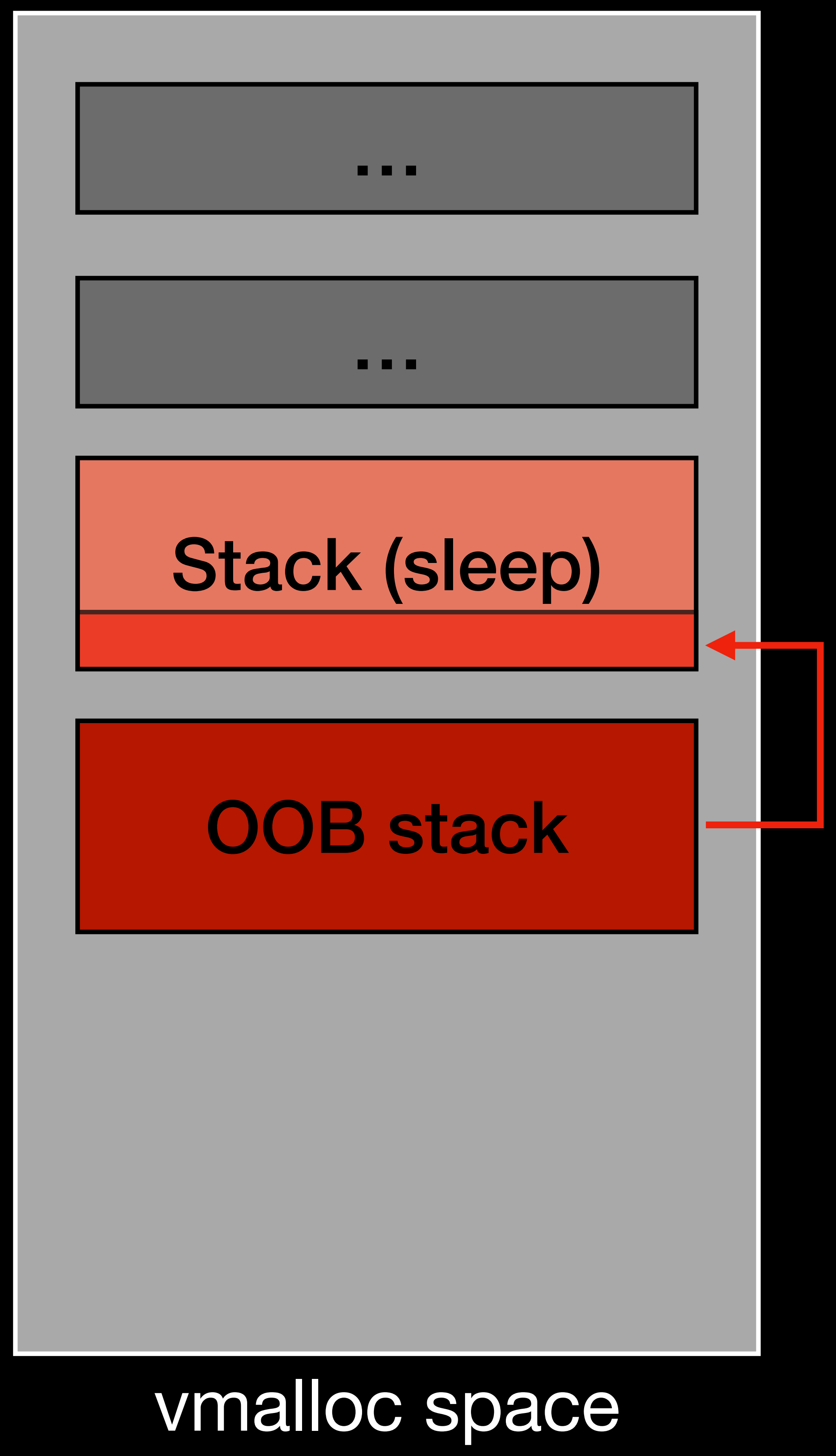
1. Spawn the victim process before the OOB process
2. The victim process performs a extended action
3. The OOB process overwrites the victim process stack



$ Ideas

- Overwrite data in another stack

1. Spawn the victim process before the OOB process ✓
2. The victim process performs a extended action ✓
3. The OOB process overwrites the victim process stack

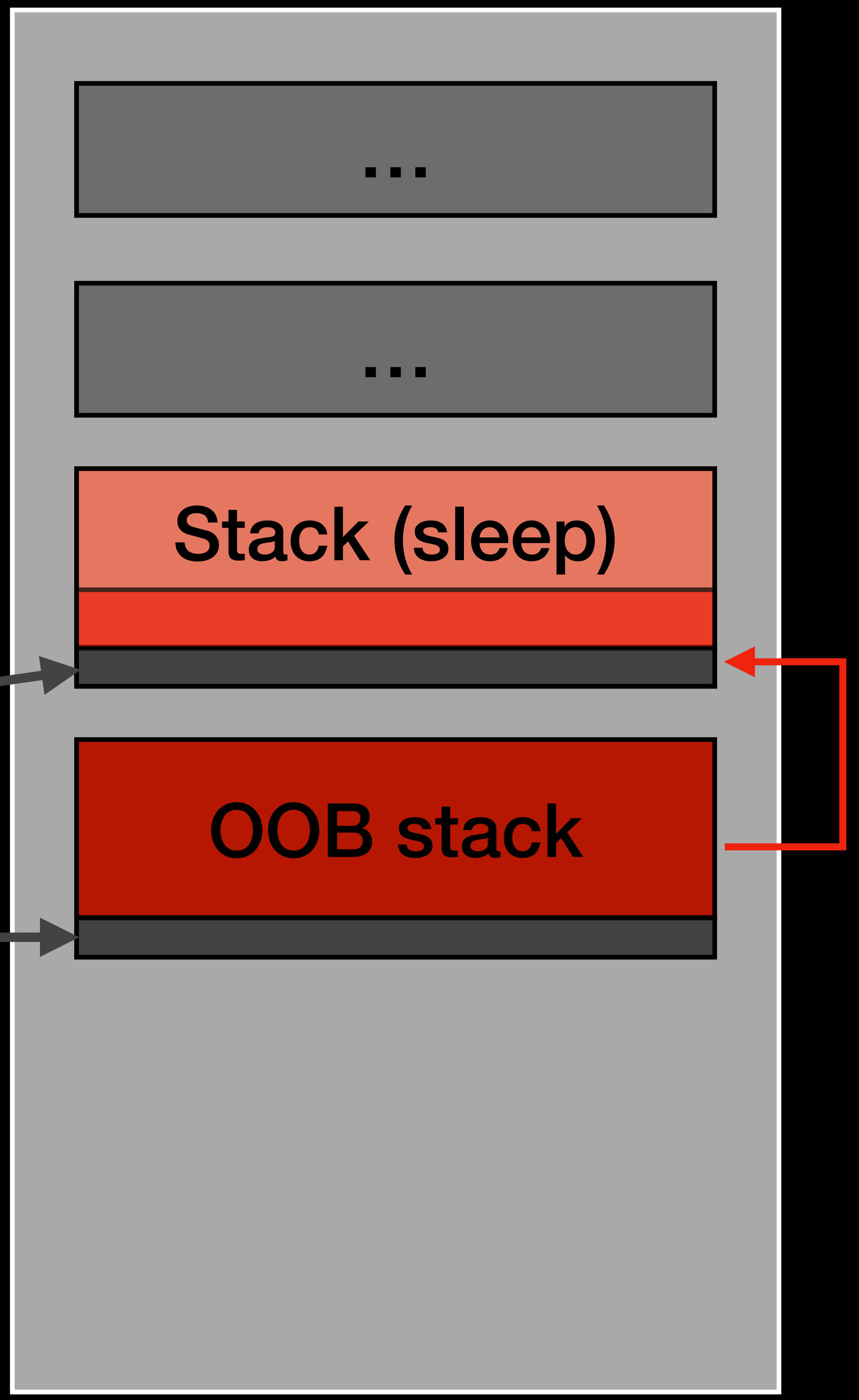


$ Ideas

- Overwrite data in another stack

1. Spawn the victim process before the OOB process
2. The victim process performs a extended action

```
__visible noinstr void do_syscall_64(struct pt_regs *regs, int nr)
{
    add_random_kstack_offset();
    nr = syscall_enter_from_user_mode(regs, nr);
```



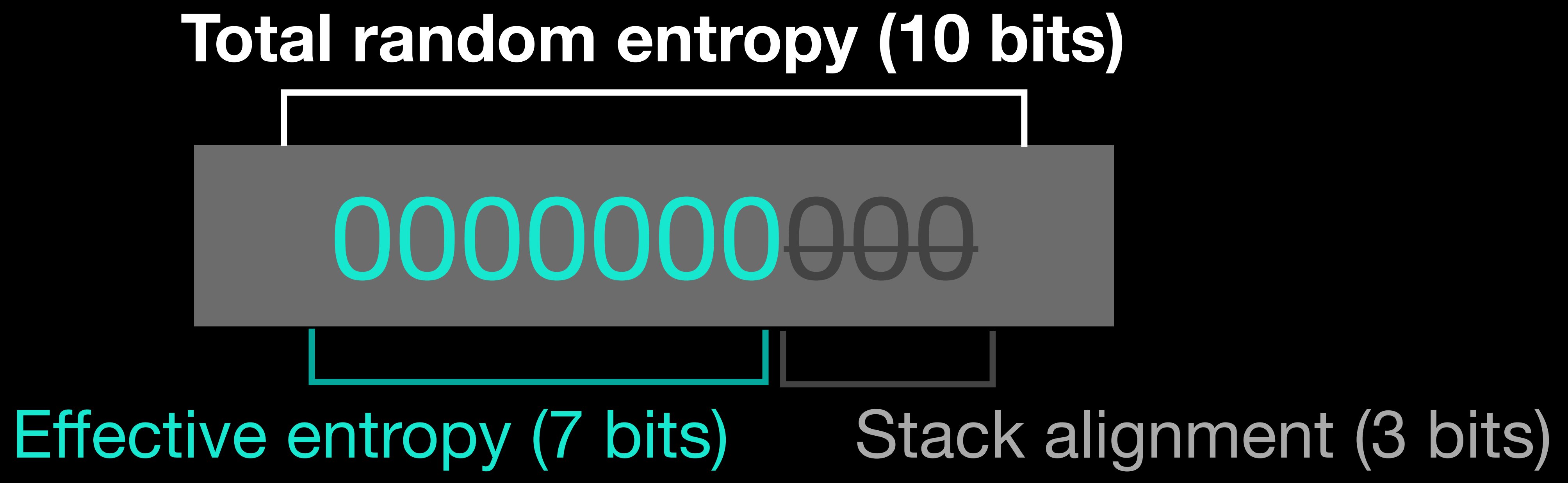
vmalloc space

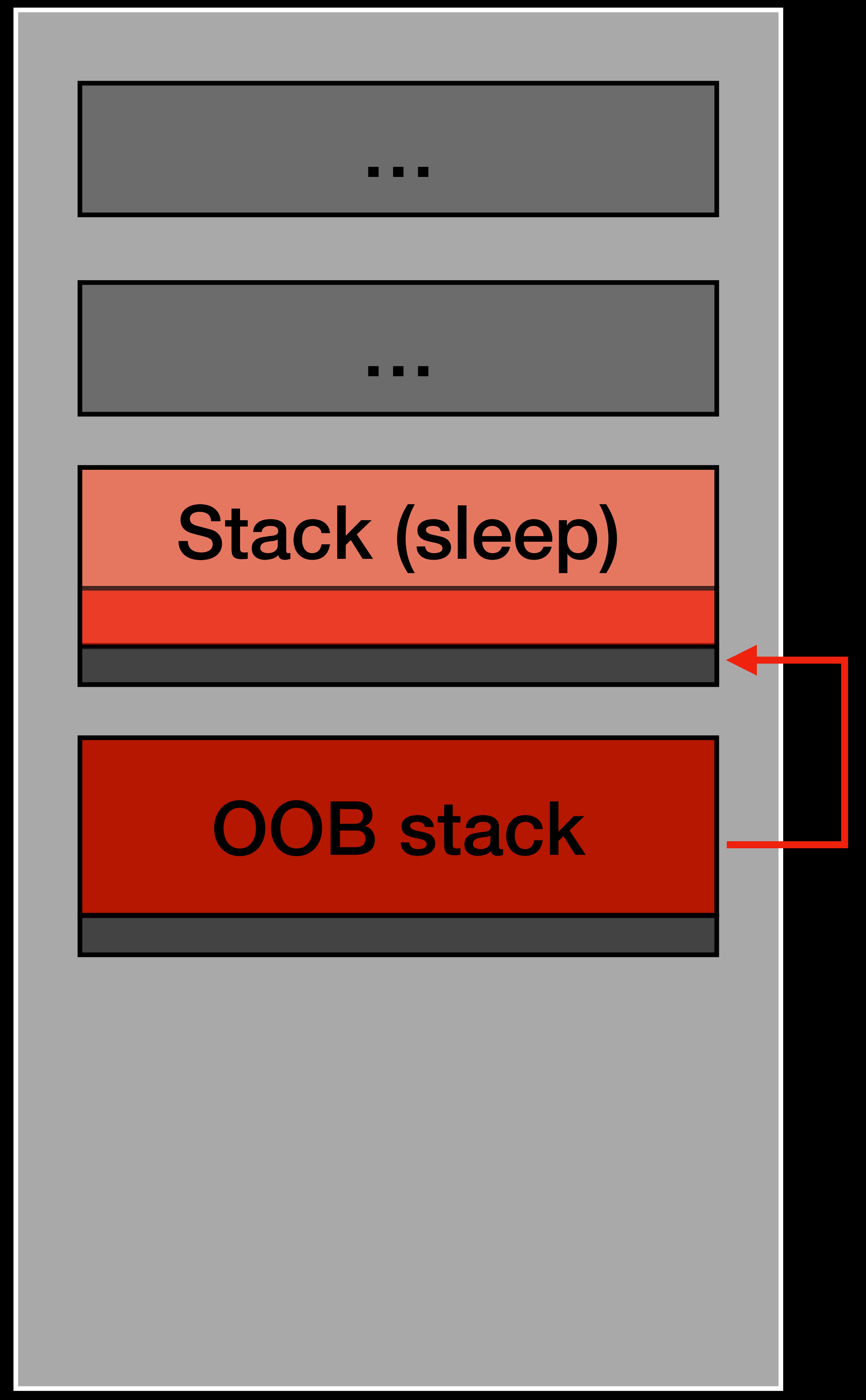
$ Ideas

- Overwrite data in another stack
  1. S… …ss
  2. T…
  3. The OOB process overwrites the victim process stack

```
#define add_random_kstack_offset() do {                    \
    if (static_branch_maybe(CONFIG_RANDOMIZE_KSTACK_OFFSET_DEFAULT, \
                &randomize_kstack_offset)) {           \
        u32 offset = raw_cpu_read(kstack_offset);          \
        u8 *ptr = __kstack_alloca(KSTACK_OFFSET_MAX(offset));    \
```

```
#define KSTACK_OFFSET_MAX(x)    ((x) & 0x3FF)
```





vmalloc space

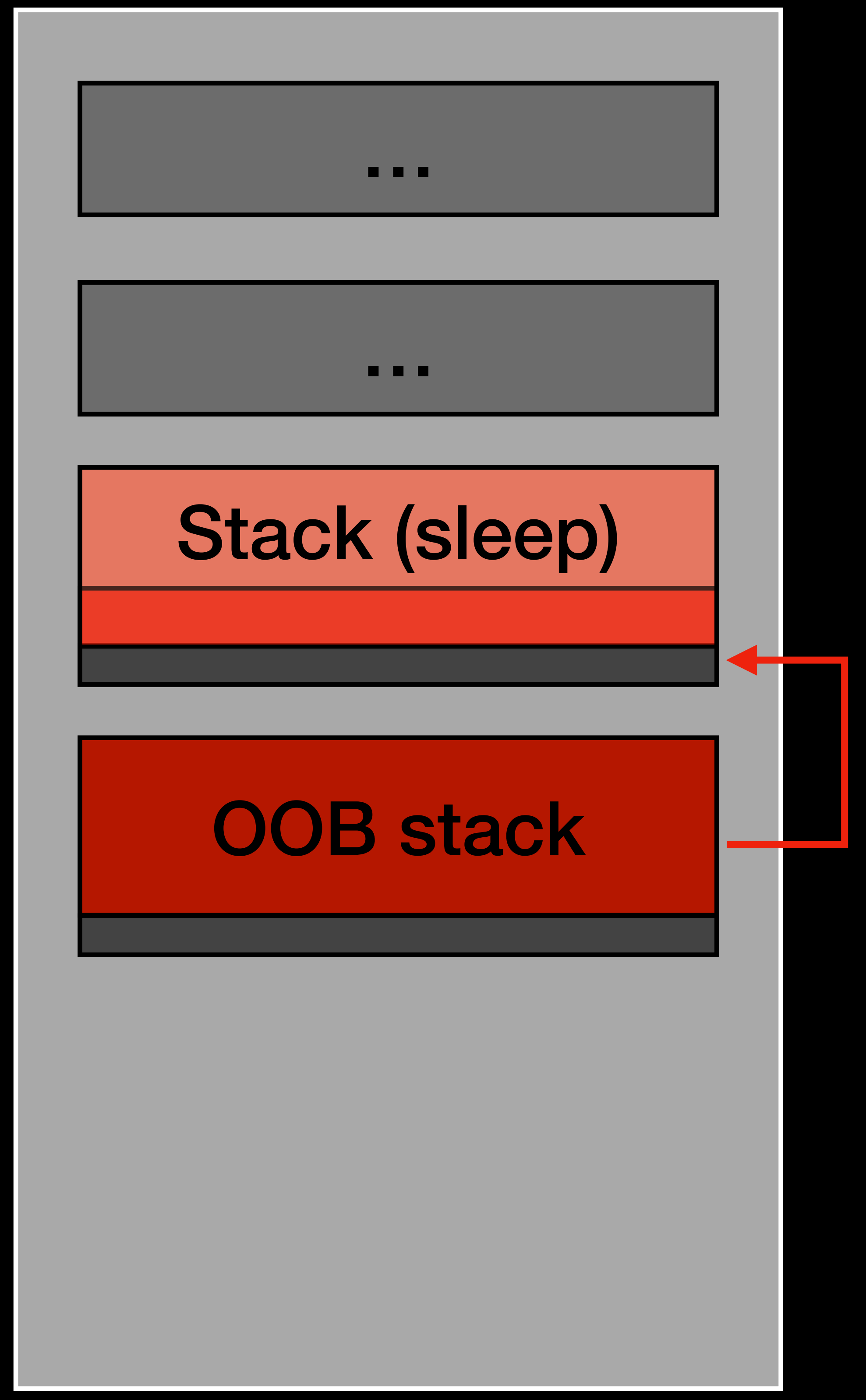
$ Ideas

- Overwrite data in another stack

✓ 1. Spawn the victim process before the OOB process

✓ 2. The victim process performs a extended action

✗ 3. The OOB process overwrites the victim process stack



vmalloc space

$ Ideas

- How the vmalloc space is used in Ubuntu?
  - /proc/vmallocinfo

```
0xffffb52cc0029000-0xffffb52cc002b000    8192 gen_pool_add_owner+0x4b/0xf0 pages=1 vmalloc N0=1
0xffffb52cc002c000-0xffffb52cc0031000   20480 dup_task_struct+0x5b/0x1b0 pages=4 vmalloc N0=4
0xffffb52cc0031000-0xffffb52cc0033000    8192 gen_pool_add_owner+0x4b/0xf0 pages=1 vmalloc N0=1
0xffffb52cc0034000-0xffffb52cc0039000   20480 dup_task_struct+0x5b/0x1b0 pages=4 vmalloc N0=4
0xffffb52cc0039000-0xffffb52cc003b000    8192 gen_pool_add_owner+0x4b/0xf0 pages=1 vmalloc N0=1
0xffffb52cc003c000-0xffffb52cc0041000   20480 dup_task_struct+0x5b/0x1b0 pages=4 vmalloc N0=4
0xffffb52cc0041000-0xffffb52cc0043000    8192 bpf_prog_alloc_no_stats+0x42/0x290 pages=1 vmalloc N0=1
0xffffb52cc0044000-0xffffb52cc0049000   20480 dup_task_struct+0x5b/0x1b0 pages=4 vmalloc N0=4
0xffffb52cc0049000-0xffffb52cc004b000    8192 acpi_os_map_iomem+0x20a/0x240 phys=0x00000000ffc00000 ioremap
0xffffb52cc004c000-0xffffb52cc0051000   20480 dup_task_struct+0x5b/0x1b0 pages=4 vmalloc N0=4
0xffffb52cc0053000-0xffffb52cc0058000   20480 pcpu_mem_zalloc+0x30/0x70 pages=4 vmalloc N0=4
0xffffb52cc0059000-0xffffb52cc005b000    8192 __pci_enable_msix_range+0x303/0x5b0 phys=0x00000000fea16000 ioremap
0xffffb52cc005c000-0xffffb52cc0061000   20480 dup_task_struct+0x5b/0x1b0 pages=4 vmalloc N0=4
0xffffb52cc0061000-0xffffb52cc0063000    8192 bpf_prog_alloc_no_stats+0x42/0x290 pages=1 vmalloc N0=1
0xffffb52cc0063000-0xffffb52cc0069000   24576 pcpu_mem_zalloc+0x30/0x70 pages=5 vmalloc N0=5
0xffffb52cc0069000-0xffffb52cc006b000    8192 vmxnet3_probe_device+0x253/0xd90 [vmxnet3] phys=0x00000000fe213000 ioremap
0xffffb52cc006c000-0xffffb52cc0071000   20480 dup_task_struct+0x5b/0x1b0 pages=4 vmalloc N0=4
```

$ Ideas

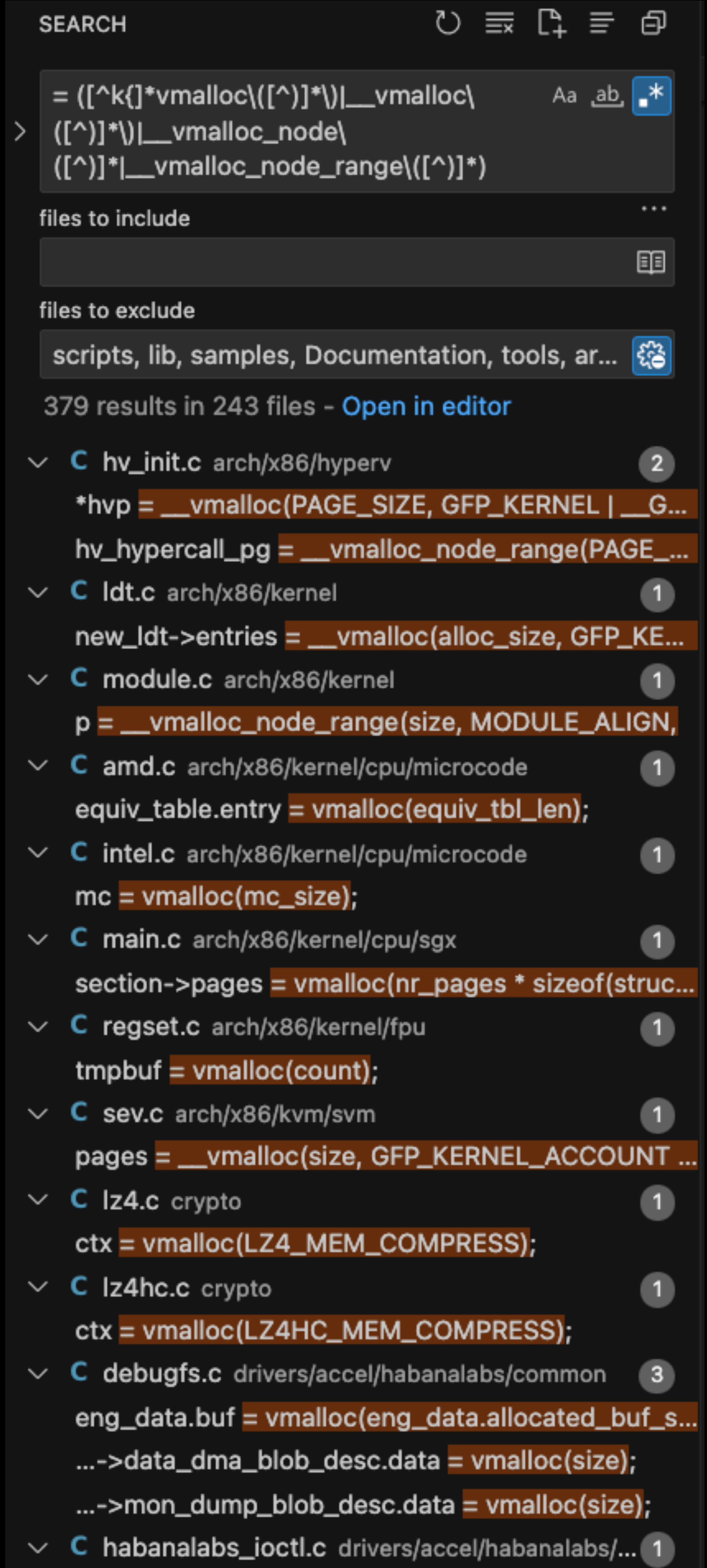
- How the vmalloc space is used in Ubuntu?
  - /proc/vmallocinfo

```
0xffffb52cc0029000-0xffffb52cc002b000    8192 gen_pool_add_owner+0x4b/0xf0 pages=1 vmalloc N0=1
0xffffb52cc002c000-0xffffb52cc0031000   20480 dup_task_struct+0x5b/0x1b0 pages=4 vmalloc N0=4
0xffffb52cc0031000-0xffffb52cc0033000    8192 gen_pool_add_owner+0x4b/0xf0 pages=1 vmalloc N0=1
0xffffb52cc0034000-0xffffb52cc0039000   20480 dup_task_struct+0x5b/0x1b0 pages=4 vmalloc N0=4
0xffffb52cc0039000-0xffffb52cc003b000    8192 gen_pool_add_owner+0x4b/0xf0 pages=1 vmalloc N0=1
0xffffb52cc003c000-0xffffb52cc0041000   20480 dup_task_struct+0x5b/0x1b0 pages=4 vmalloc N0=4
0xffffb52cc0041000-0xffffb52cc0043000    8192 bpf_prog_alloc_no_stats+0x42/0x290 pages=1 vmalloc N0=1
0xffffb52cc0049000-0xffffb52cc004b000    8192 acpi_os_map_iomem+0x20a/0x240 phys=0x00000000ffc00000 ioremap
0xffffb52cc004c000-0xffffb52cc0051000   20480 dup_task_struct+0x5b/0x1b0 pages=4 vmalloc N0=4
0xffffb52cc0053000-0xffffb52cc0058000   20480 pcpu_mem_zalloc+0x30/0x70 pages=4 vmalloc N0=4
0xffffb52cc0059000-0xffffb52cc005b000    8192 __pci_enable_msix_range+0x303/0x5b0 phys=0x00000000fea16000 ioremap
0xffffb52cc005c000-0xffffb52cc0061000   20480 dup_task_struct+0x5b/0x1b0 pages=4 vmalloc N0=4
0xffffb52cc0061000-0xffffb52cc0063000    8192 bpf_prog_alloc_no_stats+0x42/0x290 pages=1 vmalloc N0=1
0xffffb52cc0063000-0xffffb52cc0069000   24576 pcpu_mem_zalloc+0x30/0x70 pages=5 vmalloc N0=5
0xffffb52cc0069000-0xffffb52cc006b000    8192 vmxnet3_probe_device+0x253/0xd90 [vmxnet3] phys=0x00000000fe213000 ioremap
0xffffb52cc006c000-0xffffb52cc0071000   20480 dup_task_struct+0x5b/0x1b0 pages=4 vmalloc N0=4
```

🤔

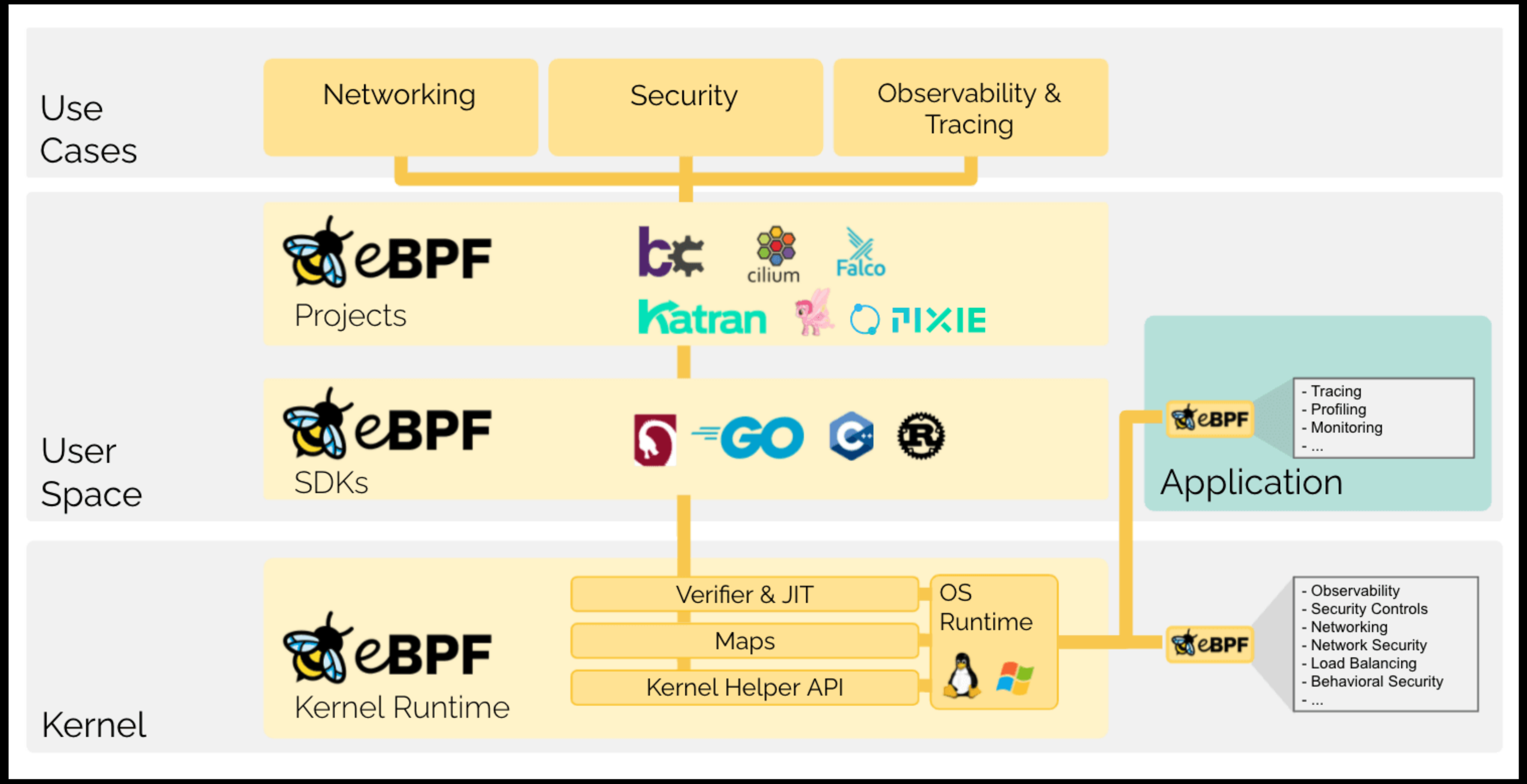
$ Ideas

- Search related functions
  - vmalloc, __vmalloc, __vmalloc_node, __vmalloc_node_range
- Primarily called by drivers, filesystems, and core features, which we are not interested in



$ eBPF 101

- Extended Berkeley Packet Filter
  - Initially developed as a subsystem for network packet filtering
  - Now capable of handling various tasks, including profiling and network monitoring



Ref: https://ebpf.io/static/e293240ecccb9d506587571007c36739/f2674/overview.png

$ eBPF 101

1. Write eBPF bytecode
2. Verify and compile it into a eBPF program
3. Attach program to sockets, cgroups and other interfaces
4. When receiving or sending data, the eBPF program will be executed

```c
struct bpf_insn prog[] = {
    // mov REG_0, 0
    ((struct bpf_insn){.code = BPF_ALU64 | BPF_MOV | BPF_K,
                       .dst_reg = BPF_REG_0,
                       .src_reg = 0,
                       .off = 0,
                       .imm = 0}),

    // return REG_0
    ((struct bpf_insn) {.code = BPF_JMP | BPF_EXIT,
                        .dst_reg = 0,
                        .src_reg = 0,
                        .off = 0,
                        .imm = 0})

};
union bpf_attr attr = {
    prog_type = BPF_PROG_TYPE_SOCKET_FILTER,
    insn_cnt  = prog_len / sizeof(struct bpf_insn),
    insns     = (__u64) prog,
    license   = (__u64) "GPL",
};
prog_fd = syscall_NR_bpf(BPF_PROG_LOAD, &attr, sizeof(attr));

socketpair(AF_UNIX, SOCK_STREAM, 0, sfds);
setsockopt(sfds[0], SOL_SOCKET, SO_ATTACH_BPF, &prog_fd, sizeof(prog_fd));

send(sfds[0], buffer, sizeof(buffer) - 1, 0);
// [...]
recv(sfds[0], buffer, sizeof(buffer) - 1, 0);
```

$ eBPF 101

1. Write eBPF bytecode
2. Verify and compile it into a eBPF program
3. Attach program to sockets, cgroups and other interfaces
4. When receiving or sending data, the eBPF program will be executed

```
struct bpf_insn prog[] = {
    // mov REG_0, 0
    ((struct bpf_insn){.code = BPF_ALU64 | BPF_MOV | BPF_K,
                       .dst_reg = BPF_REG_0,
                       .src_reg = 0,
                       .off = 0,
                       .imm = 0}),

    // return REG_0
    ((struct bpf_insn) {.code = BPF_JMP | BPF_EXIT,
                        .dst_reg = 0,
                        .src_reg = 0,
                        .off = 0,
                        .imm = 0})
};
union bpf_attr attr = {
    prog_type = BPF_PROG_TYPE_SOCKET_FILTER,
    insn_cnt  = prog_len / sizeof(struct bpf_insn),
    insns     = (__u64) prog,
    license   = (__u64) "GPL",
};
prog_fd = syscall_NR_bpf(BPF_PROG_LOAD, &attr, sizeof(attr));

socketpair(AF_UNIX, SOCK_STREAM, 0, sfds);
setsockopt(sfds[0], SOL_SOCKET, SO_ATTACH_BPF, &prog_fd, sizeof(prog_fd));

send(sfds[0], buffer, sizeof(buffer) - 1, 0);
// [...]
recv(sfds[0], buffer, sizeof(buffer) - 1, 0);
```

$ eBPF 101

1. Write eBPF bytecode
2. Verify and compile it into a eBPF program
3. Attach program to sockets, cgroups and other interfaces
4. When receiving or sending data, the eBPF program will be executed

```
struct bpf_insn prog[] = {
    // mov REG_0, 0
    ((struct bpf_insn){.code = BPF_ALU64 | BPF_MOV | BPF_K,
                       .dst_reg = BPF_REG_0,
                       .src_reg = 0,
                       .off = 0,
                       .imm = 0}),

    // return REG_0
    ((struct bpf_insn) {.code = BPF_JMP | BPF_EXIT,
                        .dst_reg = 0,
                        .src_reg = 0,
                        .off = 0,
                        .imm = 0})
};
union bpf_attr attr = {
    prog_type = BPF_PROG_TYPE_SOCKET_FILTER,
    insn_cnt  = prog_len / sizeof(struct bpf_insn),
    insns     = (__u64) prog,
    license   = (__u64) "GPL",
};
prog_fd = syscall_NR_bpf(BPF_PROG_LOAD, &attr, sizeof(attr));

socketpair(AF_UNIX, SOCK_STREAM, 0, sfds);
setsockopt(sfds[0], SOL_SOCKET, SO_ATTACH_BPF, &prog_fd, sizeof(prog_fd));

send(sfds[0], buffer, sizeof(buffer) - 1, 0);
// [...]
recv(sfds[0], buffer, sizeof(buffer) - 1, 0);
```

$ eBPF 101

1. Write eBPF bytecode
2. Verify and compile it into a eBPF program
3. Attach program to sockets, cgroups and other interfaces
4. When receiving or sending data, the eBPF program will be executed

```
struct bpf_insn prog[] = {
    // mov REG_0, 0
    ((struct bpf_insn){.code = BPF_ALU64 | BPF_MOV | BPF_K,
                       .dst_reg = BPF_REG_0,
                       .src_reg = 0,
                       .off = 0,
                       .imm = 0}),

    // return REG_0
    ((struct bpf_insn) {.code = BPF_JMP | BPF_EXIT,
                        .dst_reg = 0,
                        .src_reg = 0,
                        .off = 0,
                        .imm = 0})
};
union bpf_attr attr = {
    prog_type = BPF_PROG_TYPE_SOCKET_FILTER,
    insn_cnt  = prog_len / sizeof(struct bpf_insn),
    insns     = (__u64) prog,
    license   = (__u64) "GPL",
};
prog_fd = syscall_NR_bpf(BPF_PROG_LOAD, &attr, sizeof(attr));

socketpair(AF_UNIX, SOCK_STREAM, 0, sfds);
setsockopt(sfds[0], SOL_SOCKET, SO_ATTACH_BPF, &prog_fd, sizeof(prog_fd));

send(sfds[0], buffer, sizeof(buffer) - 1, 0);
// [...]
recv(sfds[0], buffer, sizeof(buffer) - 1, 0);
```

$ eBPF 101

- Function bpf_prog_load is used to deal with eBPF bytecode
  - Check permissions
    - Capability CAP_BPF or CAP_SYS_ADMIN
    - Unprivileged eBPF is enabled

```c
static int bpf_prog_load(union bpf_attr *attr, bpfptr_t uattr,
{
    enum bpf_prog_type type = attr->prog_type;
    struct bpf_prog *prog, *dst_prog = NULL;
    struct btf *attach_btf = NULL;
    int err;
    char license[128];

    // [...]

    if (sysctl_unprivileged_bpf_disabled && !bpf_capable())
        return -EPERM;
```

```c
static inline bool bpf_capable(void)
{
    return capable(CAP_BPF) || capable(CAP_SYS_ADMIN);
}
```

$ eBPF 101

- Function bpf_prog_load is used to deal with eBPF bytecode
  - Check permissions
    - Capability CAP_BPF or CAP_SYS_ADMIN
    - Unprivileged eBPF is enabled
  - Allocate memory for bpf_prog using __vmalloc

```
struct bpf_prog *bpf_prog_alloc(unsigned int size, gfp_t gf
{
	gfp_t gfp_flags = bpf_memcg_flags(GFP_KERNEL | __GFP_ZE
	struct bpf_prog *prog;
	int cpu;

	prog = bpf_prog_alloc_no_stats(size, gfp_extra_flags);
	if (!prog)
		return NULL;
```

```
struct bpf_prog *bpf_prog_alloc_no_stats(unsigned int size,
{
	gfp_t gfp_flags = bpf_memcg_flags(GFP_KERNEL | __GFP_ZER
	struct bpf_prog_aux *aux;
	struct bpf_prog *fp;

	size = round_up(size, PAGE_SIZE);
	fp = __vmalloc(size, gfp_flags);
	if (fp == NULL)
		return NULL;
```

$ eBPF 101

- Function bpf_prog_load is used to deal with eBPF bytecode
  - Check permissions
    - Capability CAP_BPF or CAP_SYS_ADMIN
    - Unprivileged eBPF is enabled
  - Allocate memory for bpf_prog using __vmalloc
  - Verify bytecode

```c
/* run eBPF verifier */
err = bpf_check(&prog, attr, uattr, uattr_size);
if (err < 0)
    goto free_used_maps;
```

```c
int bpf_check(struct bpf_prog **prog, union
{
    // [...]

    ret = add_subprog_and_kfunc(env);
    if (ret < 0)
        goto skip_full_check;

    ret = check_subprogs(env);
    if (ret < 0)
        goto skip_full_check;

    // [...]
```

$ eBPF 101

- After verification, the kernel will choose between interpreter or JIT
  - Depend on kernel configuration
    - CONFIG_BPF_JIT=y
    - CONFIG_BPF_JIT_DEFAULT_ON=y
    - CONFIG_HAVE_EBPF_JIT=y
- By default, Ubuntu JITs eBPF programs

```
static inline bool ebpf_jit_enabled(void)
{
    return bpf_jit_enable && bpf_jit_is_ebpf();
}
```

```
#ifdef CONFIG_BPF_JIT
int bpf_jit_enable   __read_mostly = IS_BUILTIN(CONFIG_BPF_JIT_DEFAULT_ON);
```

```
static inline bool bpf_jit_is_ebpf(void)
{
# ifdef CONFIG_HAVE_EBPF_JIT
    return true;
# else
    return false;
# endif
}
```

$ eBPF 101

- Finally, the JIT compiler iterates over bytecode and emits it into machine codes

Original bytecode

```
struct bpf_insn prog[] = {
    // mov REG_0, 0
    ((struct bpf_insn){.code = BPF_ALU64 | BPF_MOV | BPF_K,
                .dst_reg = BPF_REG_0,
                .src_reg = 0,
                .off = 0,
                .imm = 0}),

    // return REG_0
    ((struct bpf_insn) {.code = BPF_JMP | BPF_EXIT,
                .dst_reg = 0,
                .src_reg = 0,
                .off = 0,
                .imm = 0})

};
```

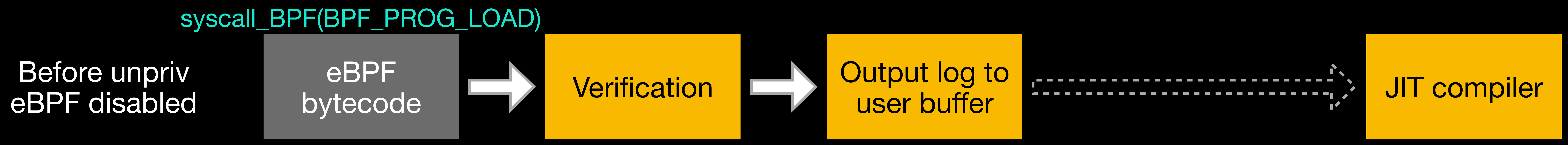
Function do_jit

```
case BPF_ALU64 | BPF_OR | BPF_K:
case BPF_ALU64 | BPF_XOR | BPF_K:
    maybe_emit_1mod(&prog, dst_reg,
            BPF_CLASS(insn->code) == BPF_ALU64);

    switch (BPF_OP(insn->code)) {
    case BPF_ADD:
        b3 = 0xC0;
        b2 = 0x05;
        break;
    // [...]
    }

    if (is_imm8(imm32))
        EMIT3(0x83, add_1reg(b3, dst_reg), imm32);
    else if (is_axreg(dst_reg))
        EMIT1_off32(b2, imm32);
    else
        EMIT2_off32(0x81, add_1reg(b3, dst_reg), imm32);
    break;
```

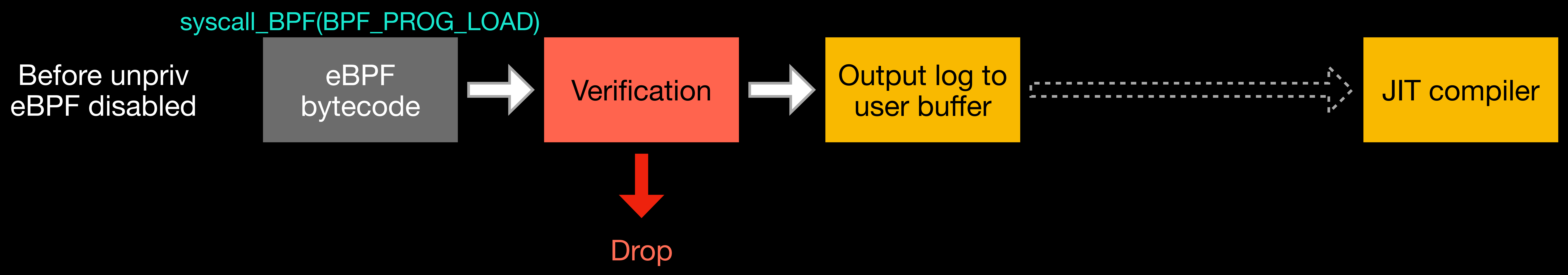
Emitted machine codes

```
0xffffffffc0000648:  nop    DWORD PTR [rax+rax*1+0x0]
0xffffffffc000064d:  xchg   ax,ax
0xffffffffc000064f:  push   rbp
0xffffffffc0000650:  mov    rbp,rsp
0xffffffffc0000653:  xor    eax,eax
0xffffffffc0000655:  leave
0xffffffffc0000656:  ret
```

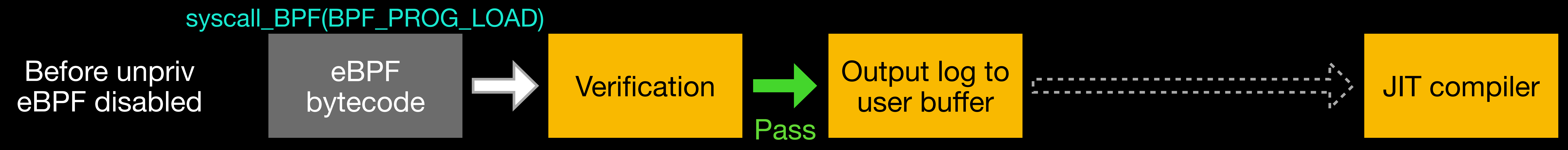
$ Bytecode Injection



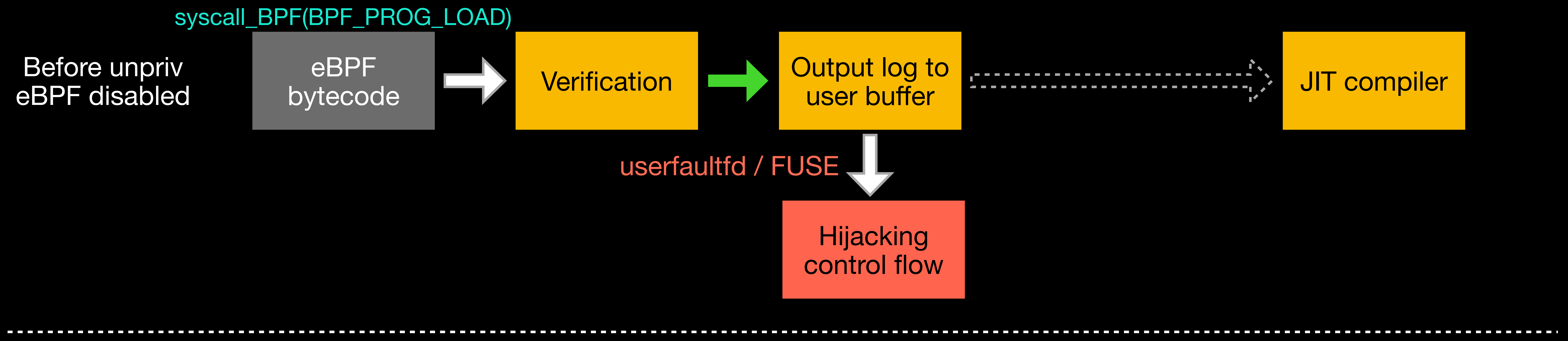
$ Bytecode Injection



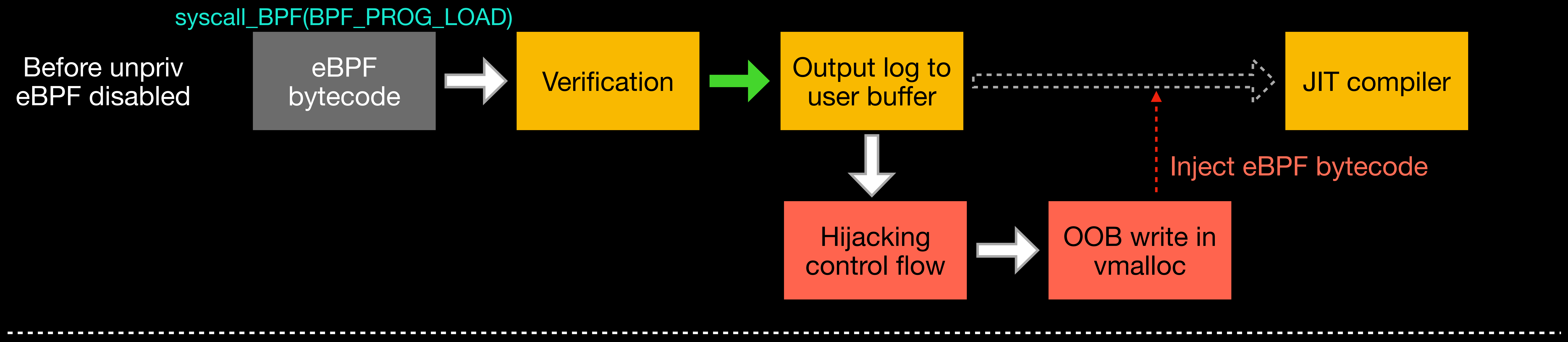
$ Bytecode Injection



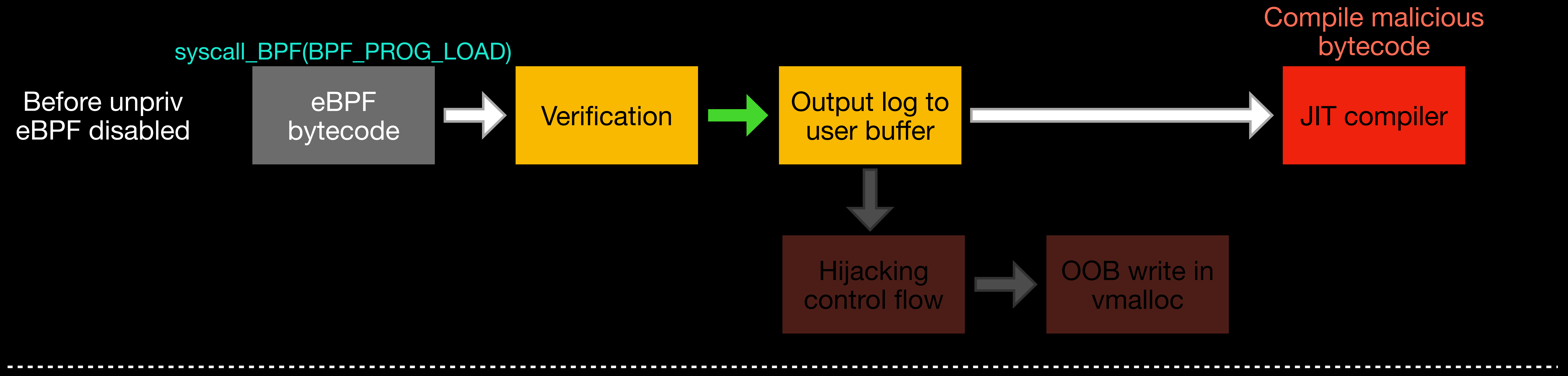
$ Bytecode Injection



$ Bytecode Injection



$ Bytecode Injection



$ Restricted eBPF

- Unfortunately, unprivileged eBPF has been disabled since March 2022
- We cannot create eBPF programs anymore 😢…

Unprivileged eBPF disabled by default for Ubuntu 20.04 LTS, 18.04 LTS, 16.04 ESM

Security kernel, security

alexmurray 2 ✎ Mar 2022

Mar 2022

1 / 1
Mar 2022

As part of the most recent round of kernel security updates for Ubuntu, another set of cross-domain transient execution attacks were addressed. Known as BTI and BHI 22 (branch target / history injection respectively) these attacks allow a local unprivileged user to leak privileged information from the kernel via execution of code gadgets. Currently the only known way to

$ Restricted eBPF

- Unfortunately, unprivileged eBPF has been disabled since March 2022
- ~~We cannot create eBPF programs anymore 😢…~~ is it true?

Unprivileged eBPF disabled by default for Ubuntu 20.04 LTS, 18.04 LTS, 16.04 ESM

Security kernel, security

alexmurray 2 ✎ Mar 2022

As part of the most recent round of kernel security updates for Ubuntu, another set of cross-domain transient execution attacks were addressed. Known as BTI and BHI 22 (branch target / history injection respectively) these attacks allow a local unprivileged user to leak privileged information from the kernel via execution of code gadgets. Currently the only known way to

Mar 2022

1 / 1
Mar 2022

$ Restricted eBPF

- Create a restricted eBPF program indirectly
  - Use seccomp with filter mode
  - Attach a filter to a socket
  - …

```c
struct sock_filter filter[] = {
    BPF_STMT(BPF_LD | BPF_W | BPF_ABS, offsetof(struct seccomp_data, nr)),

    BPF_JUMP(BPF_JMP | BPF_JEQ | BPF_K, SYS_read, 0, 1),
    BPF_STMT(BPF_RET | BPF_K, SECCOMP_RET_ALLOW),

    BPF_JUMP(BPF_JMP | BPF_JEQ | BPF_K, SYS_write, 0, 1),
    BPF_STMT(BPF_RET | BPF_K, SECCOMP_RET_ALLOW),

    BPF_JUMP(BPF_JMP | BPF_JEQ | BPF_K, SYS_exit, 0, 1),
    BPF_STMT(BPF_RET | BPF_K, SECCOMP_RET_ALLOW),

    BPF_STMT(BPF_RET | BPF_K, SECCOMP_RET_KILL_PROCESS),
};

struct sock_fprog prog = {
    .len = (unsigned short)(sizeof(filter) / sizeof(filter[0])),
    .filter = filter,
};

prctl(PR_SET_NO_NEW_PRIVS, 1, 0, 0, 0);
prctl(PR_SET_SECCOMP, SECCOMP_MODE_FILTER, &prog);
```

seccomp with filter mode

```c
struct sock_filter filter[] = {
    BPF_STMT(BPF_RET + BPF_K, SECCOMP_RET_ALLOW),
};

struct sock_fprog bpf_prog = {
    .len = sizeof(filter) / sizeof(filter[0]),
    .filter = filter,
};

int sock = sock = socket(AF_INET, SOCK_STREAM, 0);
setsockopt(sock, SOL_SOCKET, SO_ATTACH_FILTER, &bpf_prog, sizeof(bpf_prog));
```

Socket filter

$ Restricted eBPF

- Call bpf_prepare_filter internally
  - Verify the filter bytecode
  - Convert the filter bytecode to eBPF bytecode
  - Perform JIT compilation

```
static struct bpf_prog *bpf_prepare_filter(struct bpf_prog *fp,
                                           bpf_aux_classic_check_t trans)
{
    int err;

    fp->bpf_func = NULL;
    fp->jited = 0;

    err = bpf_check_classic(fp->insns, fp->len);
    // [...]
    if (!fp->jited)
        fp = bpf_migrate_filter(fp);

    return fp;
}
```

1. Opcode whitelist

```
static bool chk_code_allowed(u16 code_to_probe)
{
    static const bool codes[] = {
        /* 32 bit ALU operations */
        [BPF_ALU | BPF_ADD | BPF_K] = true,
        [BPF_ALU | BPF_ADD | BPF_X] = true,
        [BPF_ALU | BPF_SUB | BPF_K] = true,
        [BPF_ALU | BPF_SUB | BPF_X] = true,
        [BPF_ALU | BPF_MUL | BPF_K] = true,
        [BPF_ALU | BPF_MUL | BPF_X] = true,
```

$ Restricted eBPF

- Call bpf_prepare_filter internally
  - Verify the filter bytecode
  - Convert the filter bytecode to eBPF bytecode
  - Perform JIT compilation

```c
static struct bpf_prog *bpf_prepare_filter(struct bpf_prog *fp,
                                           bpf_aux_classic_check_t trans)
{
    int err;

    fp->bpf_func = NULL;
    fp->jited = 0;

    err = bpf_check_classic(fp->insns, fp->len);
    // [...]
    if (!fp->jited)
        fp = bpf_migrate_filter(fp);

    return fp;
}
```

2. Special checks

```c
switch (ftest->code) {
    case BPF_ALU | BPF_DIV | BPF_K:
    case BPF_ALU | BPF_MOD | BPF_K:
        /* Check for division by zero */
        if (ftest->k == 0)
            return -EINVAL;
        break;
```

$ Restricted eBPF

- Call bpf_prepare_filter internally
  - Verify the filter bytecode
  - Convert the filter bytecode to eBPF bytecode
  - Perform JIT compilation

1. Duplicate the filter bytecode

```
old_prog = kmemdup(fp->insns, old_len * sizeof(struct sock_filter),
                   GFP_KERNEL | __GFP_NOWARN);

/* 1st pass: calculate the new program length. */
err = bpf_convert_filter(old_prog, old_len, NULL, &new_len,
                         &seen_ld_abs);

/* Expand fp for appending the new filter representation. */
old_fp = fp;
fp = bpf_prog_realloc(old_fp, bpf_prog_size(new_len), 0);
fp->len = new_len;

/* 2nd pass: remap sock_filter insns into bpf_insn insns. */
err = bpf_convert_filter(old_prog, old_len, fp, &new_len,
                         &seen_ld_abs);
fp = bpf_prog_select_runtime(fp, &err);
// [...]
```

$ Restricted eBPF

- Call bpf_prepare_filter internally
  - Verify the filter bytecode
  - Convert the filter bytecode to eBPF bytecode
  - Perform JIT compilation

2. Calculate new program size

```
/* 1st pass: calculate the new program length. */
err = bpf_convert_filter(old_prog, old_len, NULL, &new_len,
                         &seen_ld_abs);

/* Expand fp for appending the new filter representation. */
old_fp = fp;
fp = bpf_prog_realloc(old_fp, bpf_prog_size(new_len), 0);
fp->len = new_len;

/* 2nd pass: remap sock_filter insns into bpf_insn insns. */
err = bpf_convert_filter(old_prog, old_len, fp, &new_len,
                         &seen_ld_abs);
fp = bpf_prog_select_runtime(fp, &err);
// [...]
```

$ Restricted eBPF

- Call bpf_prepare_filter internally
  - Verify the filter bytecode
  - Convert the filter bytecode to eBPF bytecode
  - Perform JIT compilation

```
old_prog = kmemdup(fp->insns, old_len * sizeof(struct sock_filter),
                   GFP_KERNEL | __GFP_NOWARN);

/* 1st pass: calculate the new program length. */
err = bpf_convert_filter(old_prog, old_len, NULL, &new_len
```

3. Reallocate program memory

```
/* Expand fp for appending the new filter representation. */
old_fp = fp;
fp = bpf_prog_realloc(old_fp, bpf_prog_size(new_len), 0);
fp->len = new_len;

/* 2nd pass: remap sock_filter insns into bpf_insn insns. */
err = bpf_convert_filter(old_prog, old_len, fp, &new_len,
                         &seen_ld_abs);
fp = bpf_prog_select_runtime(fp, &err);
// [...]
```

$ Restricted eBPF

- Call bpf_prepare_filter internally
  - Verify the filter bytecode
  - Convert the filter bytecode to eBPF bytecode
  - Perform JIT compilation

```
old_prog = kmemdup(fp->insns, old_len * sizeof(struct sock_filter),
                   GFP_KERNEL | __GFP_NOWARN);

/* 1st pass: calculate the new program length. */
err = bpf_convert_filter(old_prog, old_len, NULL, &new_len,
                         &seen_ld_abs);

/* Expand fp for appending the new filter representation. */
old_fp =
fp = bp
fp->len

/* 2nd pass: remap sock_filter insns into bpf_insn insns. */
err = bpf_convert_filter(old_prog, old_len, fp, &new_len,
                         &seen_ld_abs);
fp = bpf_prog_select_runtime(fp, &err);
// [...]
```

4. Convert the filter bytecode to eBPF bytecode

$ Restricted eBPF

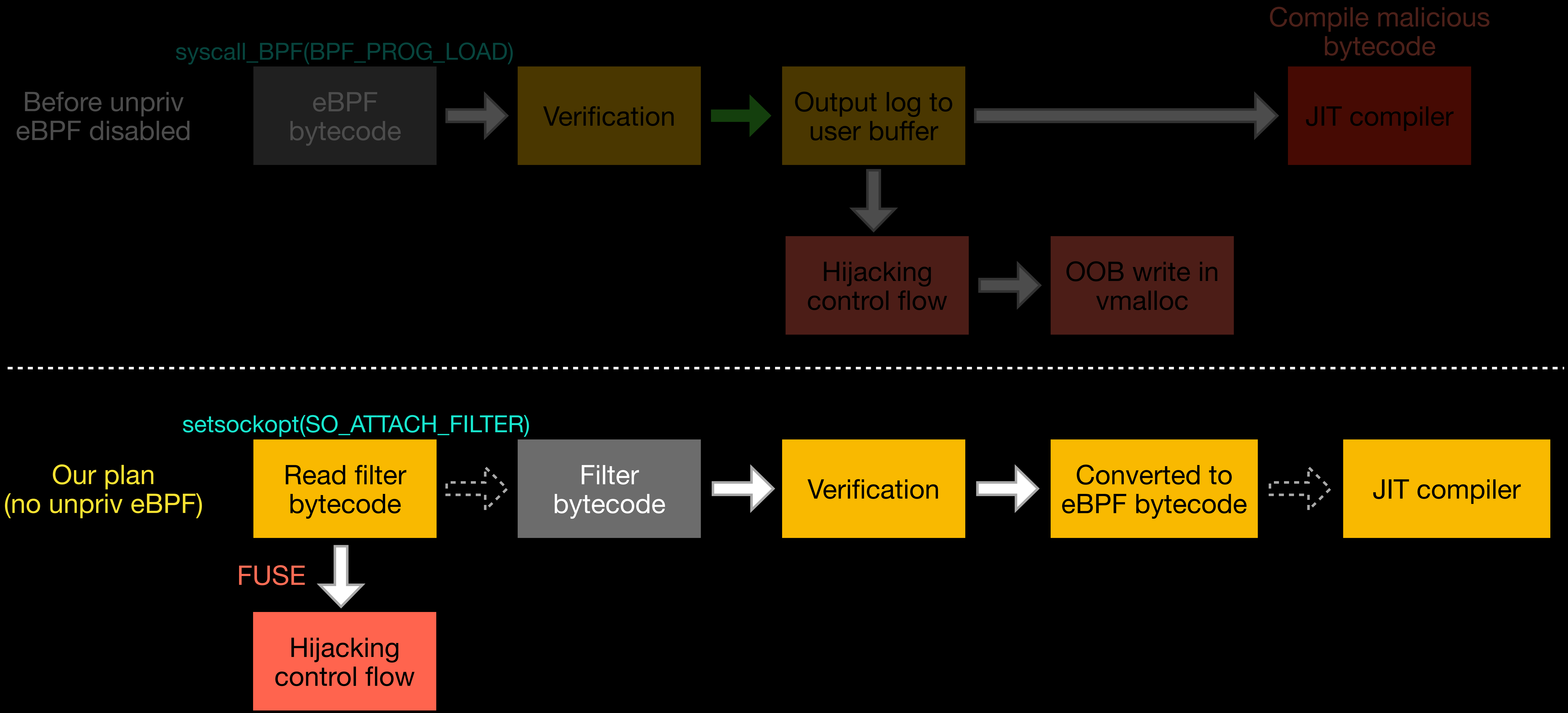
- Call bpf_prepare_filter internally
  - Verify the filter bytecode
  - Convert the filter bytecode to eBPF bytecode
  - Perform JIT compilation

```
old_prog = kmemdup(fp->insns, old_len * sizeof(struct sock_filter),
                   GFP_KERNEL | __GFP_NOWARN);

/* 1st pass: calculate the new program length. */
err = bpf_convert_filter(old_prog, old_len, NULL, &new_len,
                         &seen_ld_abs);

/* Expand fp for appending the new filter representation. */
old_fp = fp;
fp = bpf_prog_realloc(old_fp, bpf_prog_size(new_len), 0);
fp->len = new_len;

/* 2nd pass: remap sock_filter insns into bpf_insn insns. */
err = b

fp = bpf_prog_select_runtime(fp, &err);
// [...]
```

5. JIT the eBPF bytecode

$ Bytecode Injection Revenge

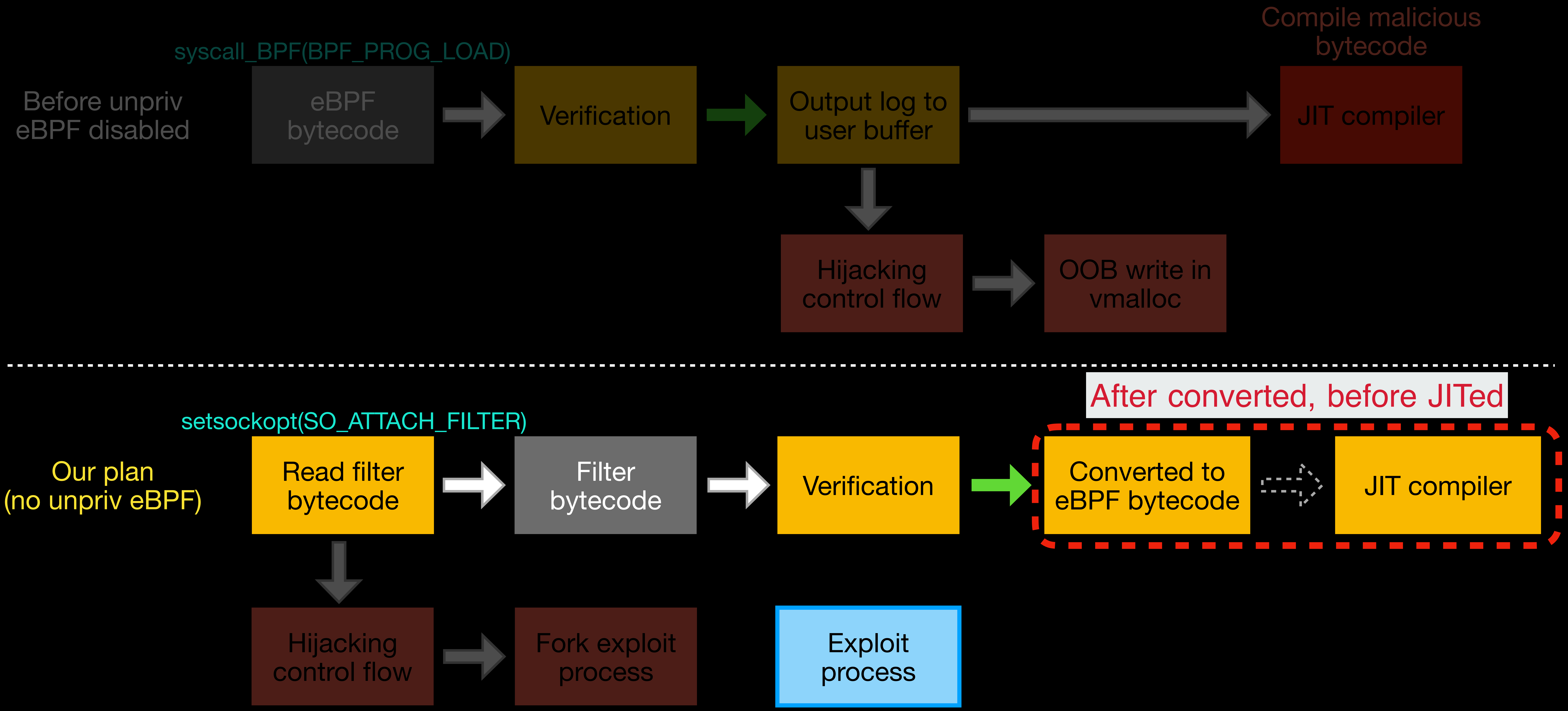
$ Bytecode Injection Revenge



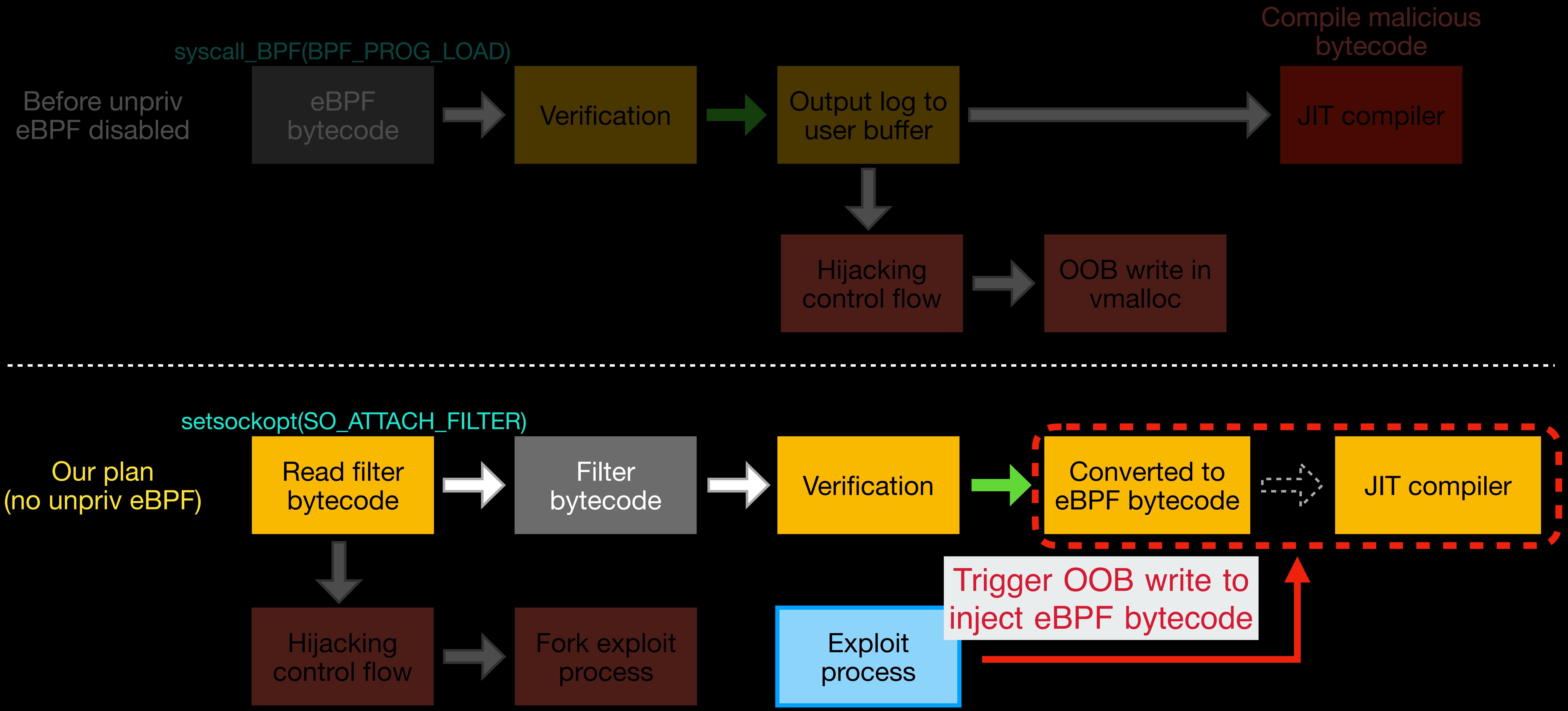
$ Bytecode Injection Revenge

$ Bytecode Injection Revenge

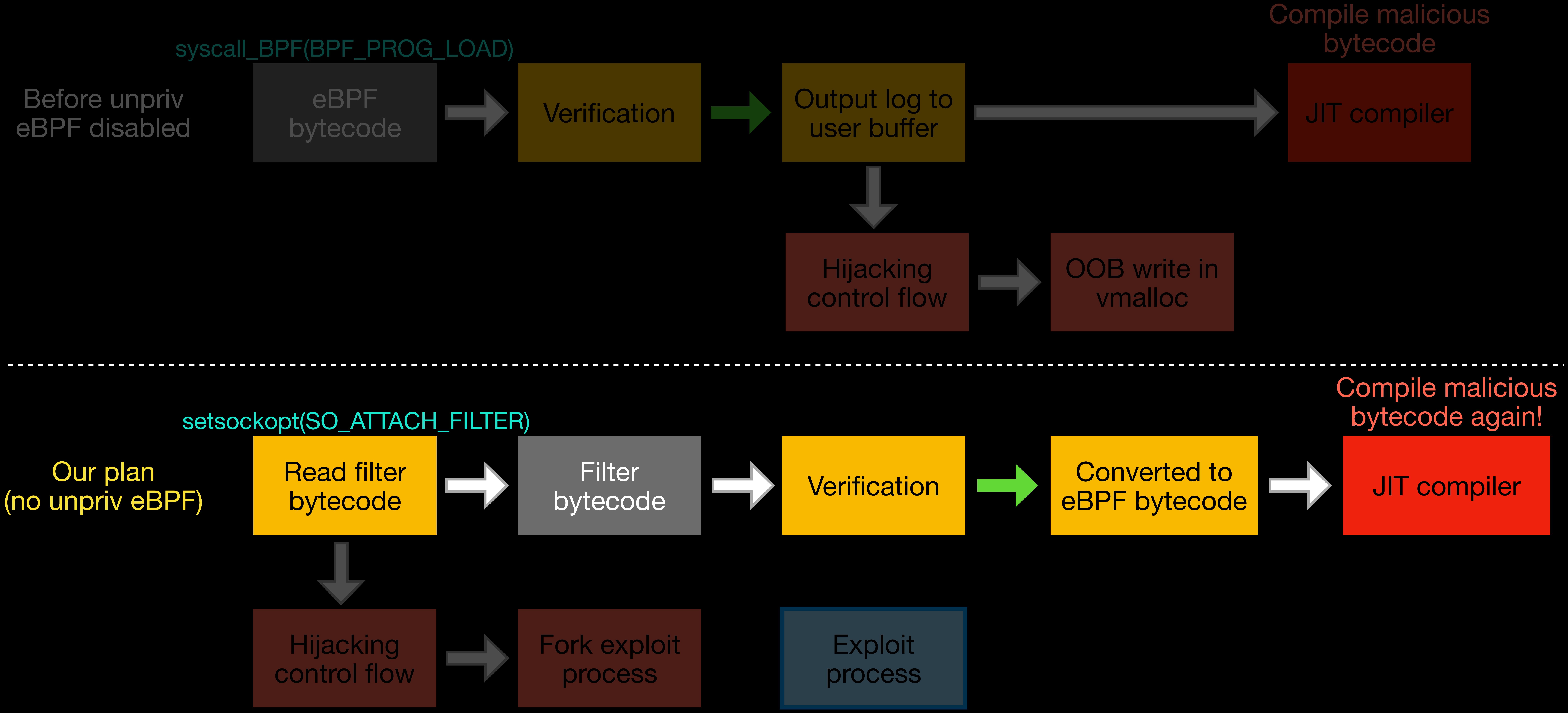
$ Bytecode Injection Revenge



$ Bytecode Injection Revenge

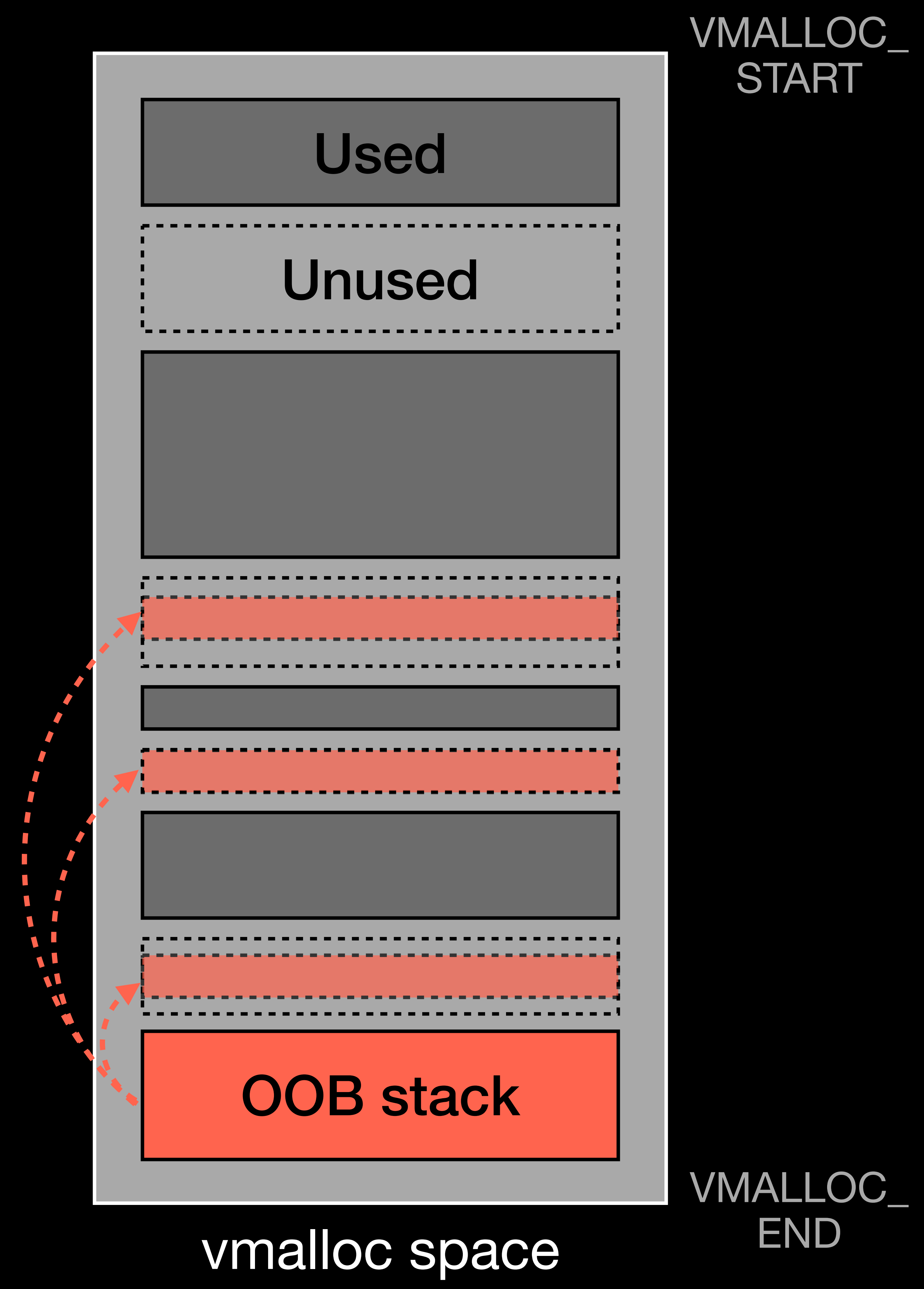


$ Bytecode Injection Revenge



$ Heap Shaping

- The initial vmalloc layout is unknown
  - Which memory slot is allocated for a new memory region is unpredictable



$ Heap Shaping

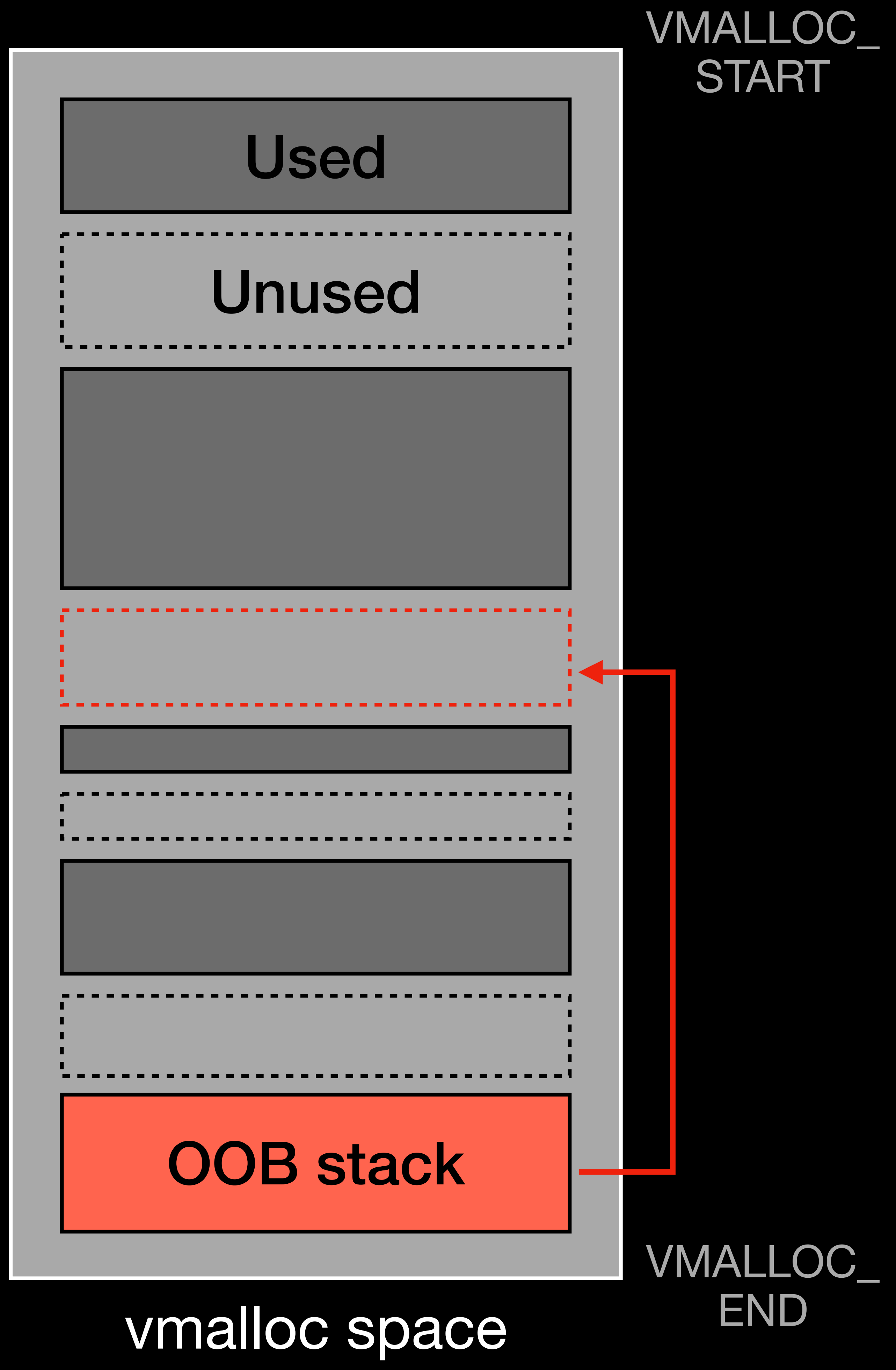
- Accessing unmapped memory causes only a single CPU to halt
  - Ideally, we have a total of CPU# chances 😄

```
aaa@aaa:~/Desktop$ lsb_release -d
No LSB modules are available.
Description:    Ubuntu 23.10
aaa@aaa:~/Desktop$ cat /proc/sys/kernel/panic_on_oops
0
```

panic_on_oops

Controls the kernel's behaviour when an oops or BUG is encountered.

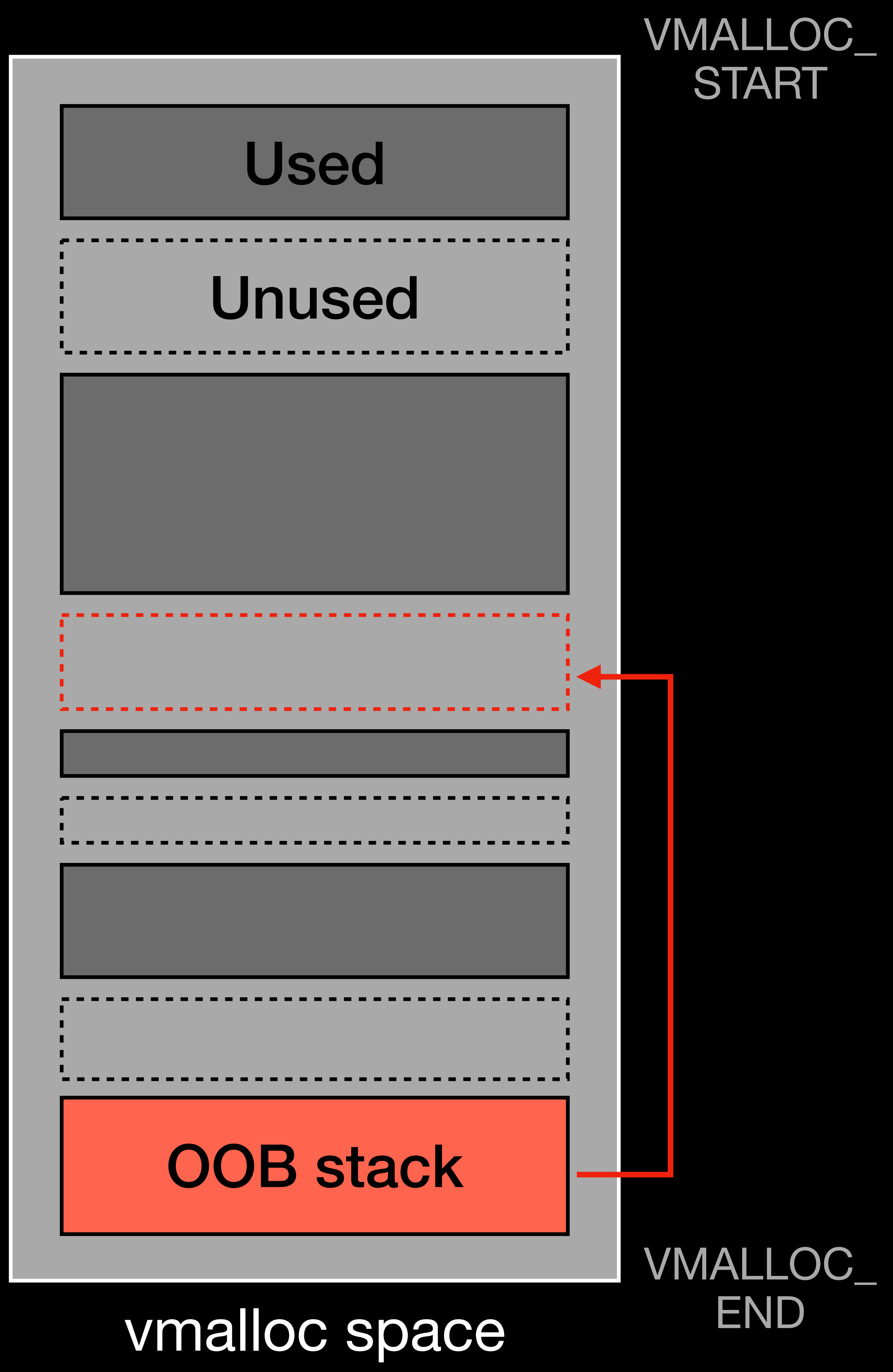
0	Try to continue operation.
1	Panic immediately. If the *panic* sysctl is also non-zero then the machine will be rebooted.



$ Heap Shaping

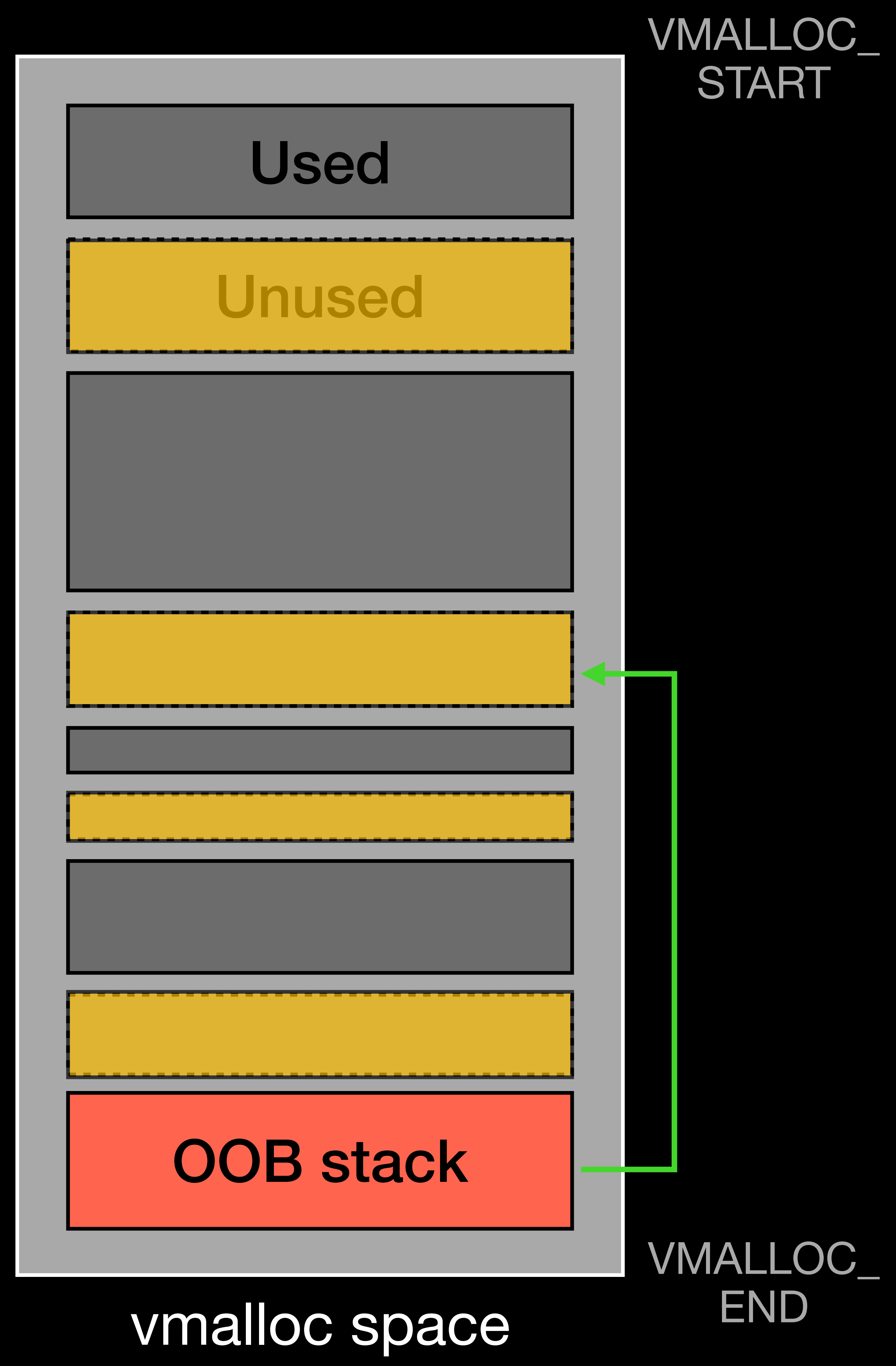
- Accessing unmapped memory causes only a single CPU to halt
  - ~~Ideally, we have a total of CPU# chances~~
  - Hold an RTNL big lock when triggering the bug 😭

```
static int rtnetlink_rcv_msg(struct sk_buff *skb, struct nlmsghdr *nlh,
                             struct netlink_ext_ack *extack)
{
    // [...]
    rtnl_lock();
    link = rtnl_get_link(family, type);
    if (link && link->doit)
        err = link->doit(skb, nlh, extack); // tc_modify_qdisc
    rtnl_unlock();
```

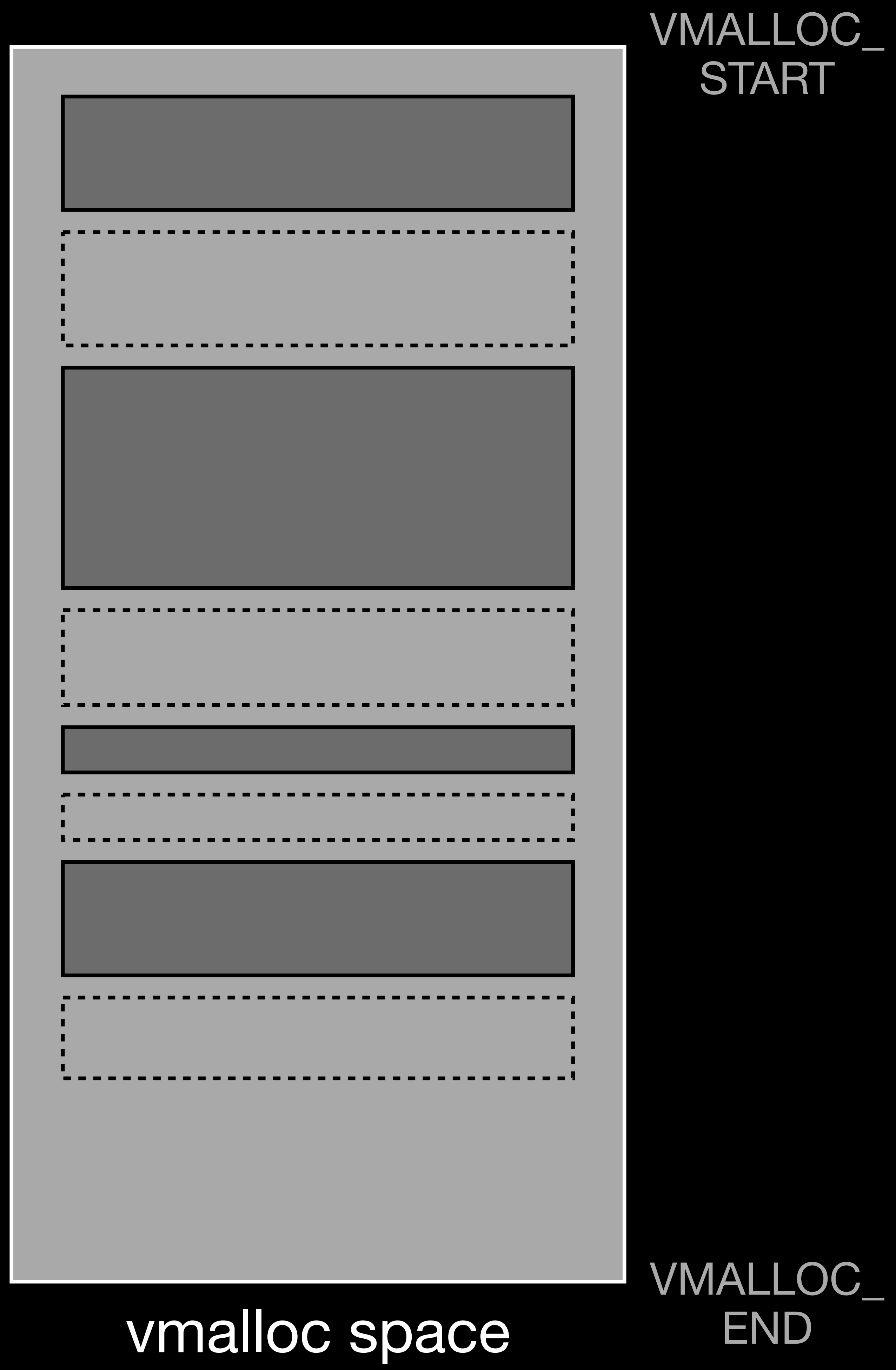


$ Heap Shaping

- We have only one shot at the attack
- Need to exclude conditions that cause invalid memory access

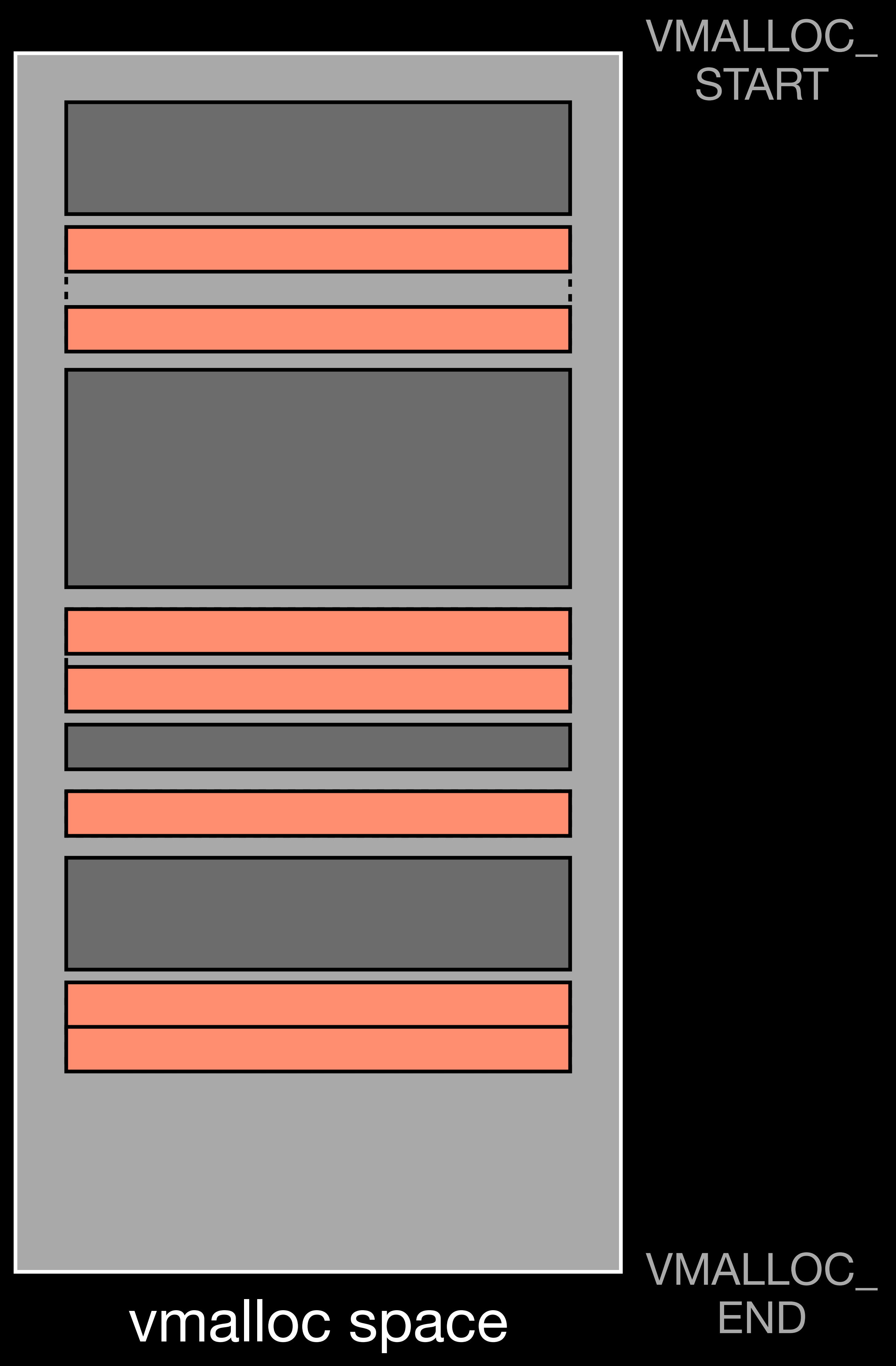


$ Heap Shaping



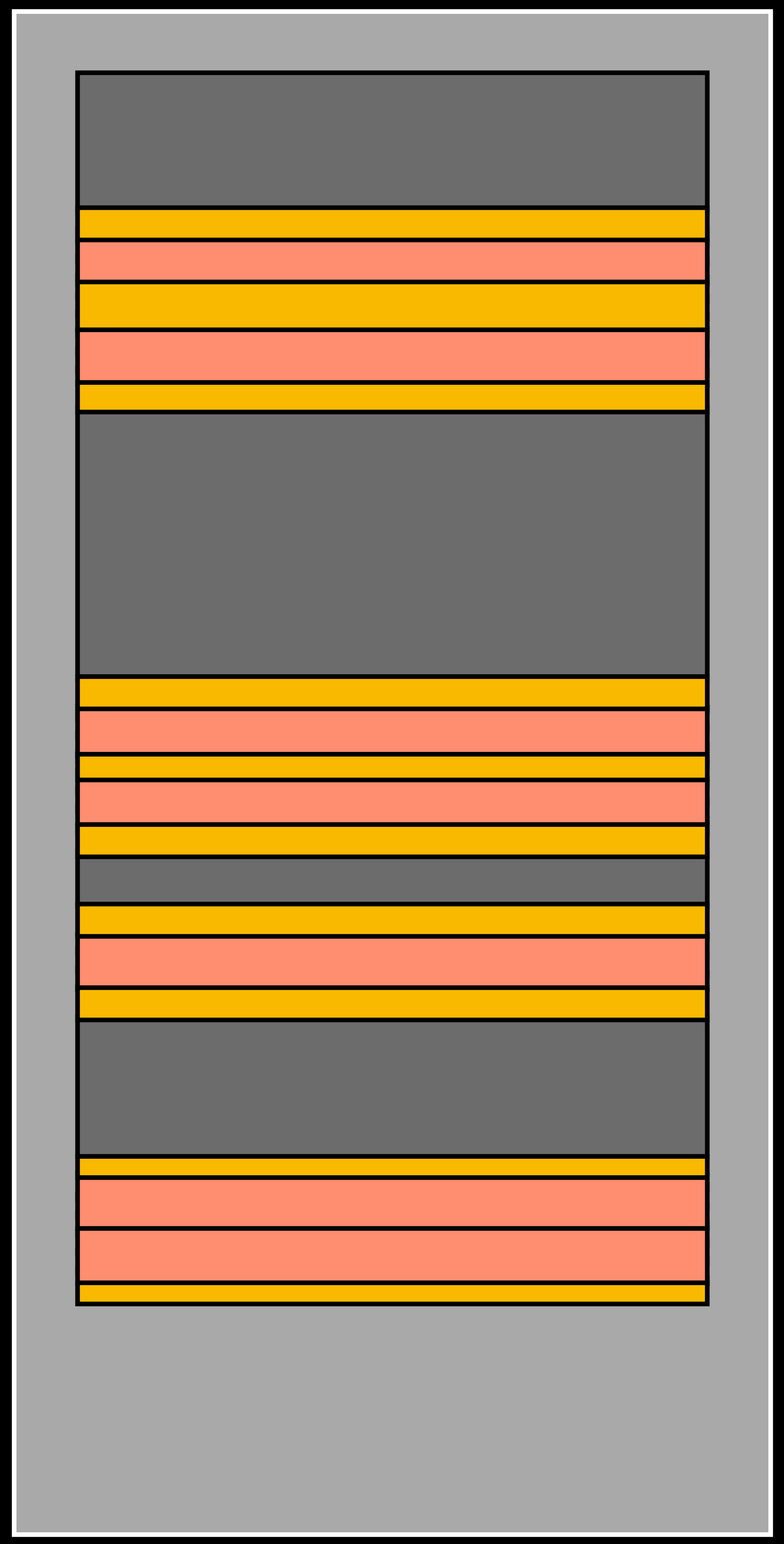
1. Initial vmalloc space is messy

$ Heap Shaping



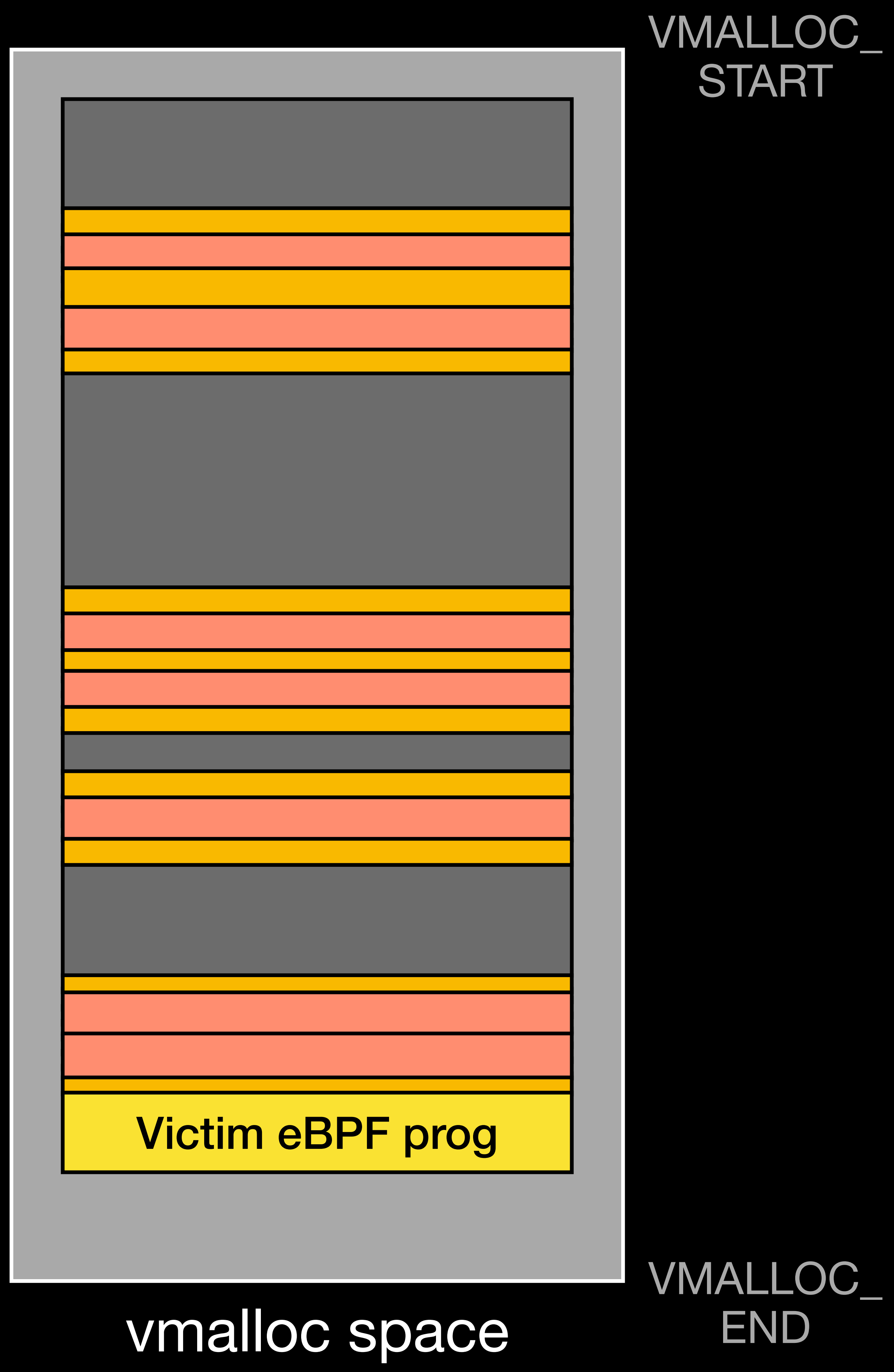
2. Fork multiple processes to fill large gaps

$ Heap Shaping



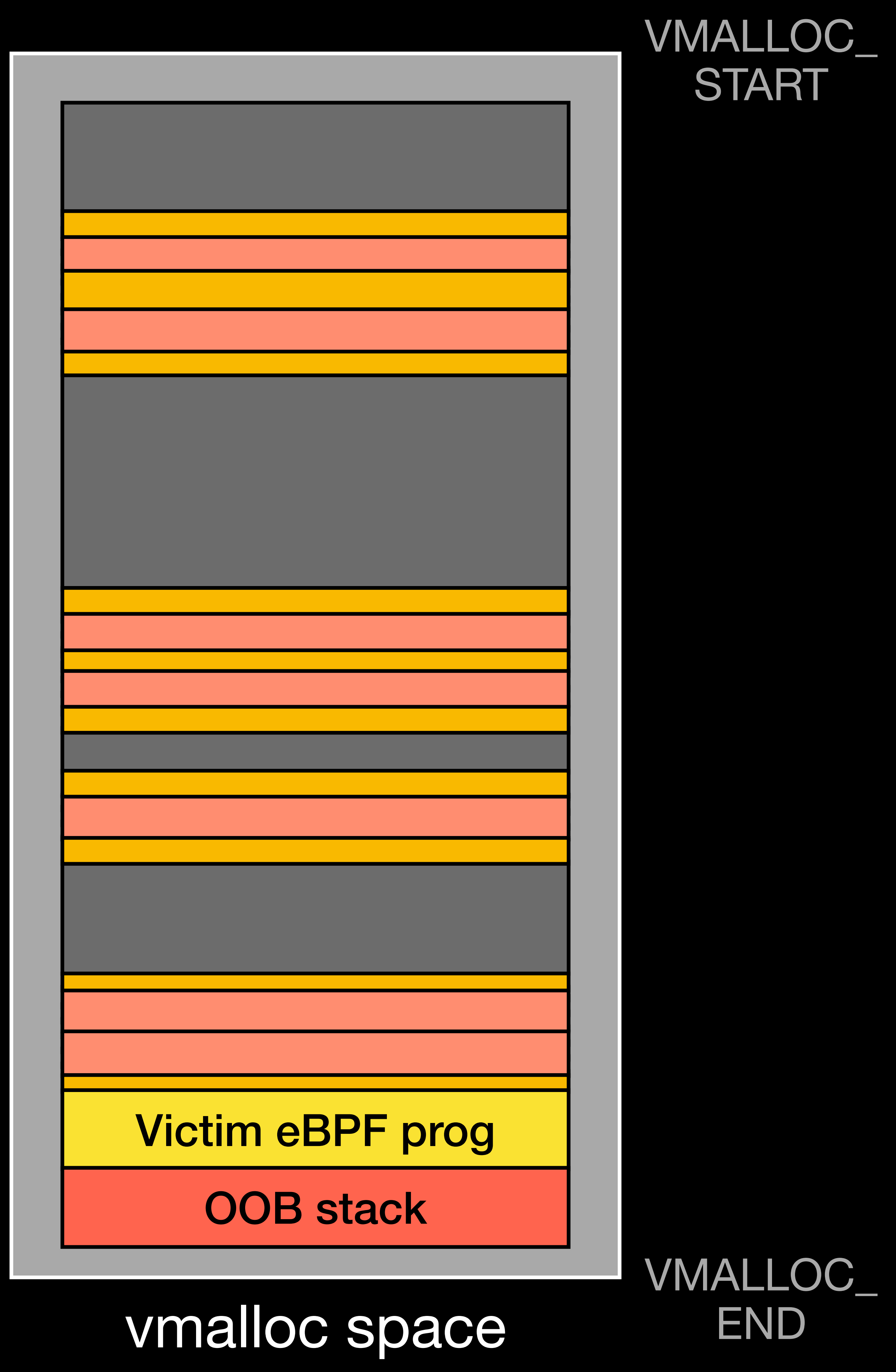
3. Spray eBPF programs to fill small gaps

$ Heap Shaping



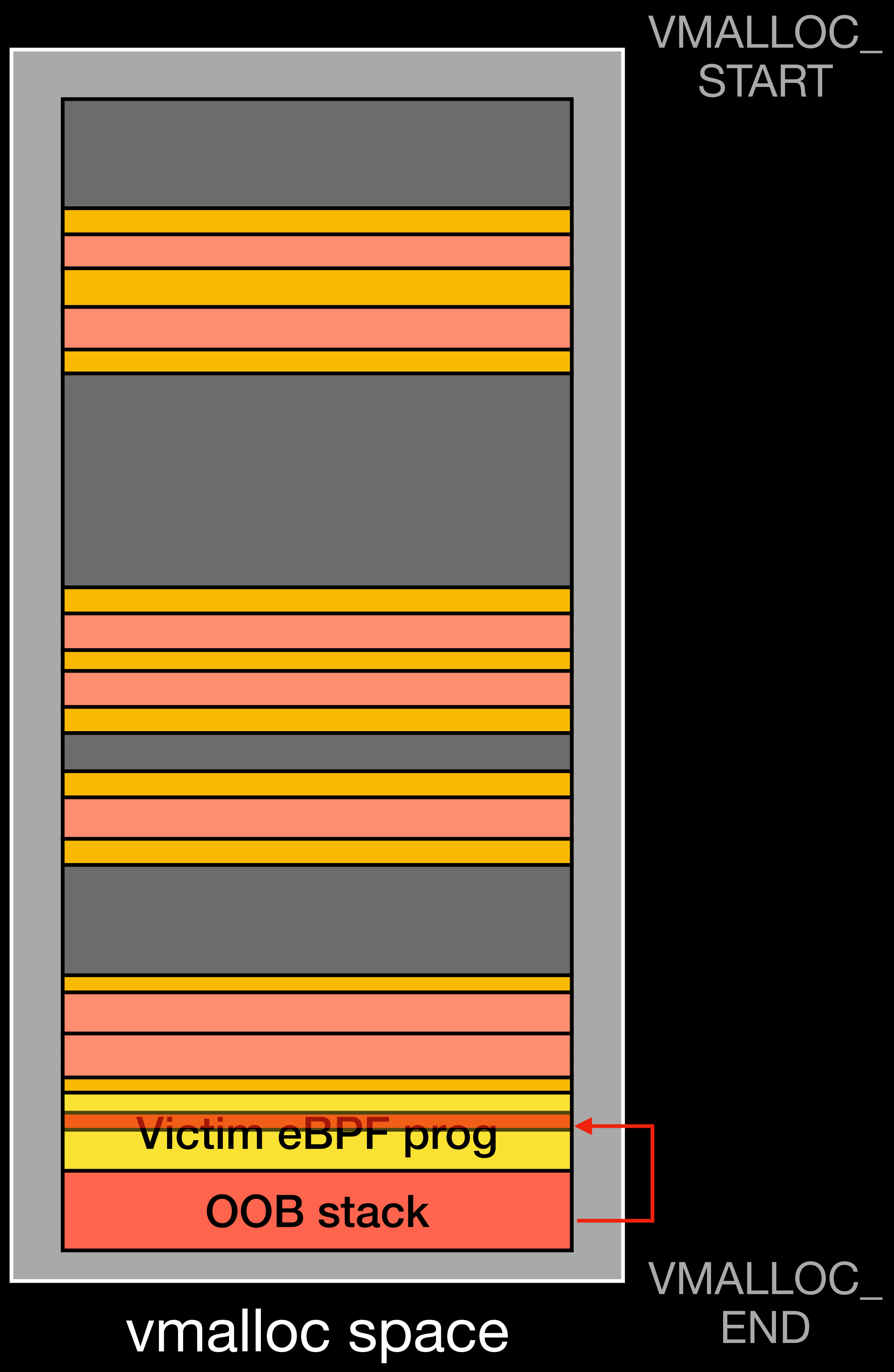
4. Allocate victim eBPF programs

$ Heap Shaping



5. Spawn the OOB write process

$ Heap Shaping

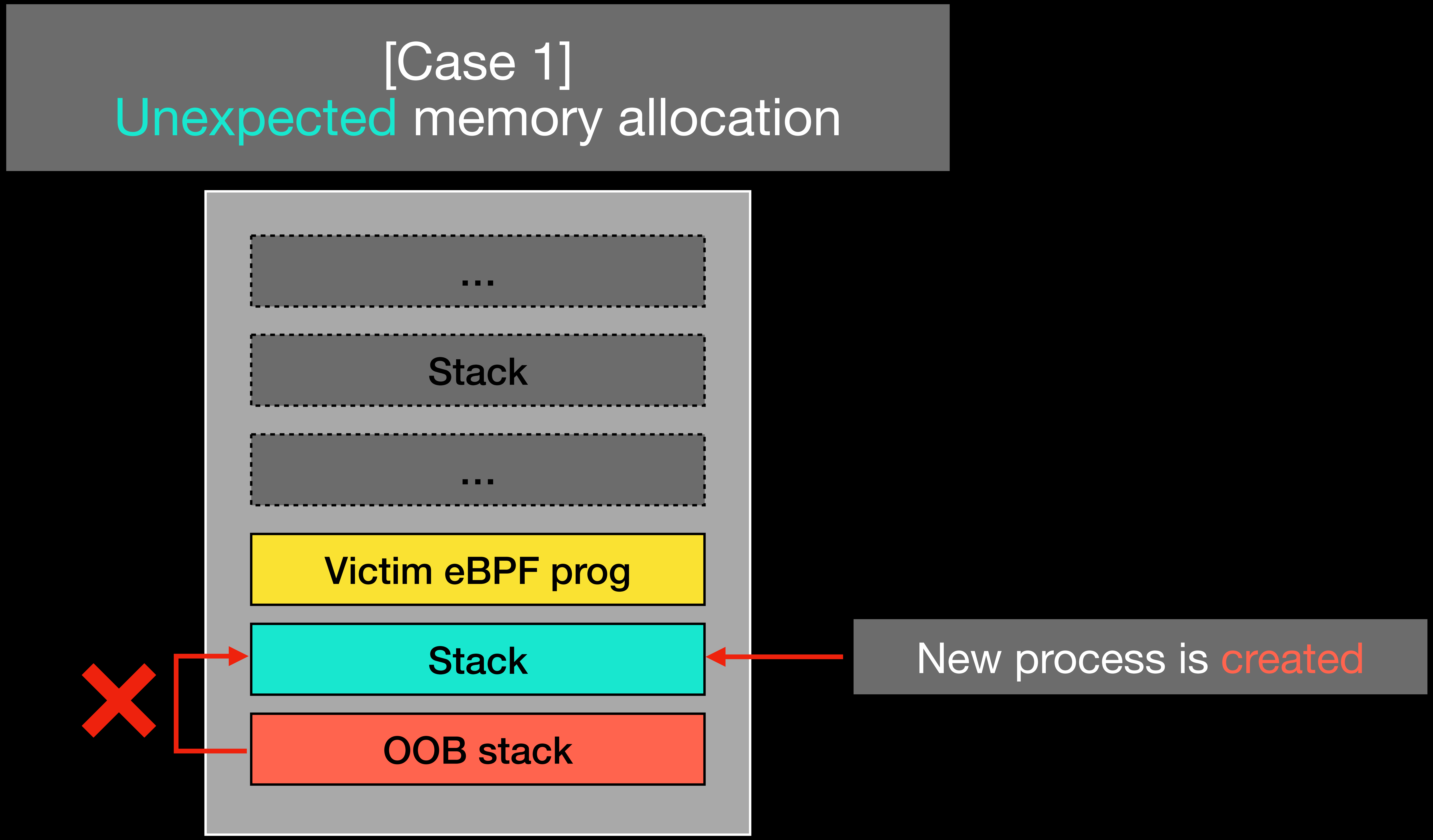


6. Inject eBPF bytecode by OOB write

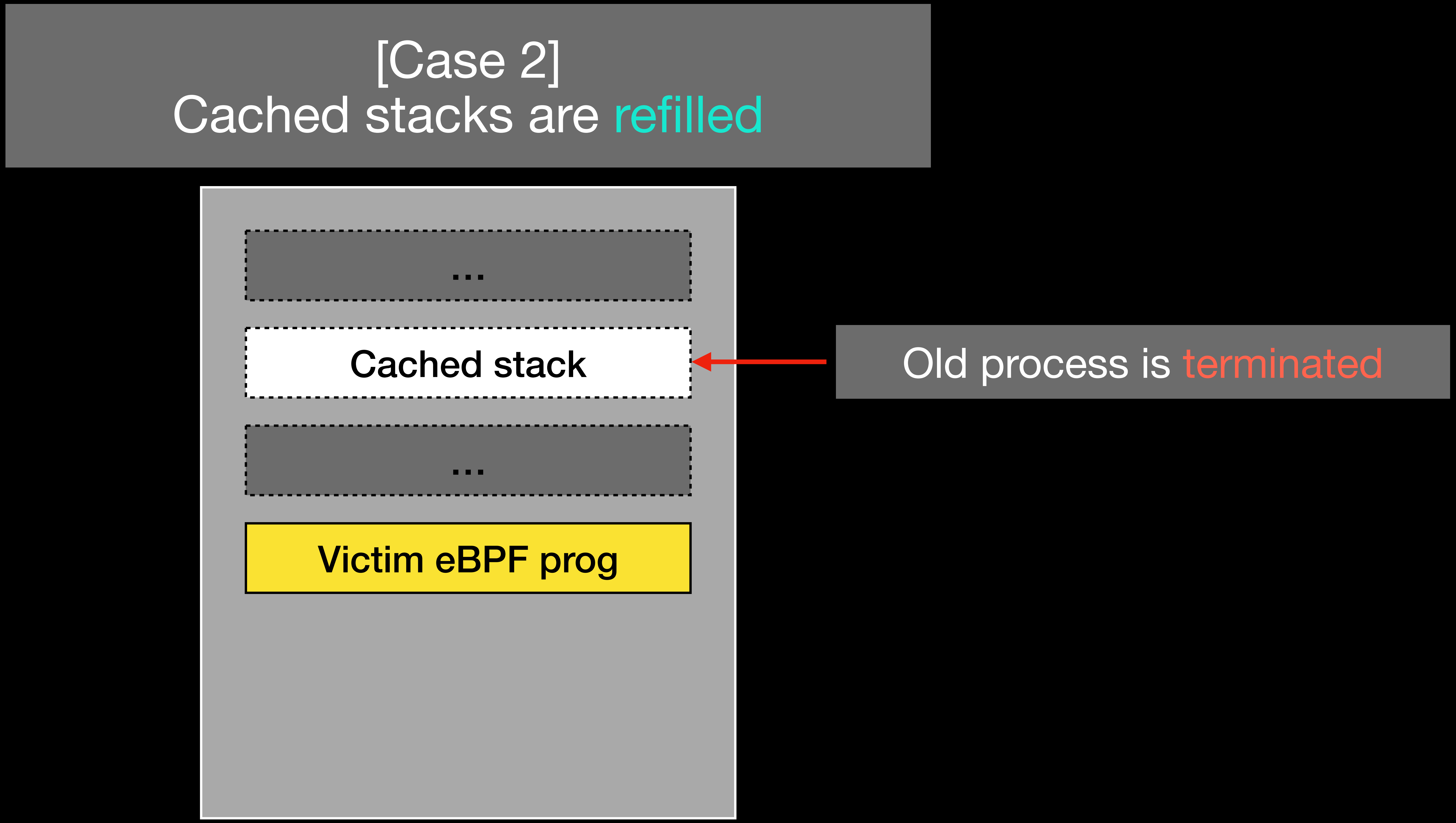
$ Heap Shaping

- In fact, processes creation and termination occur frequently in Ubuntu
  - Refill the cache stacks
  - Reorder memory layout
  - …
- Even after shaping, vmalloc space layout remains somewhat unpredictable

$ Heap Shaping

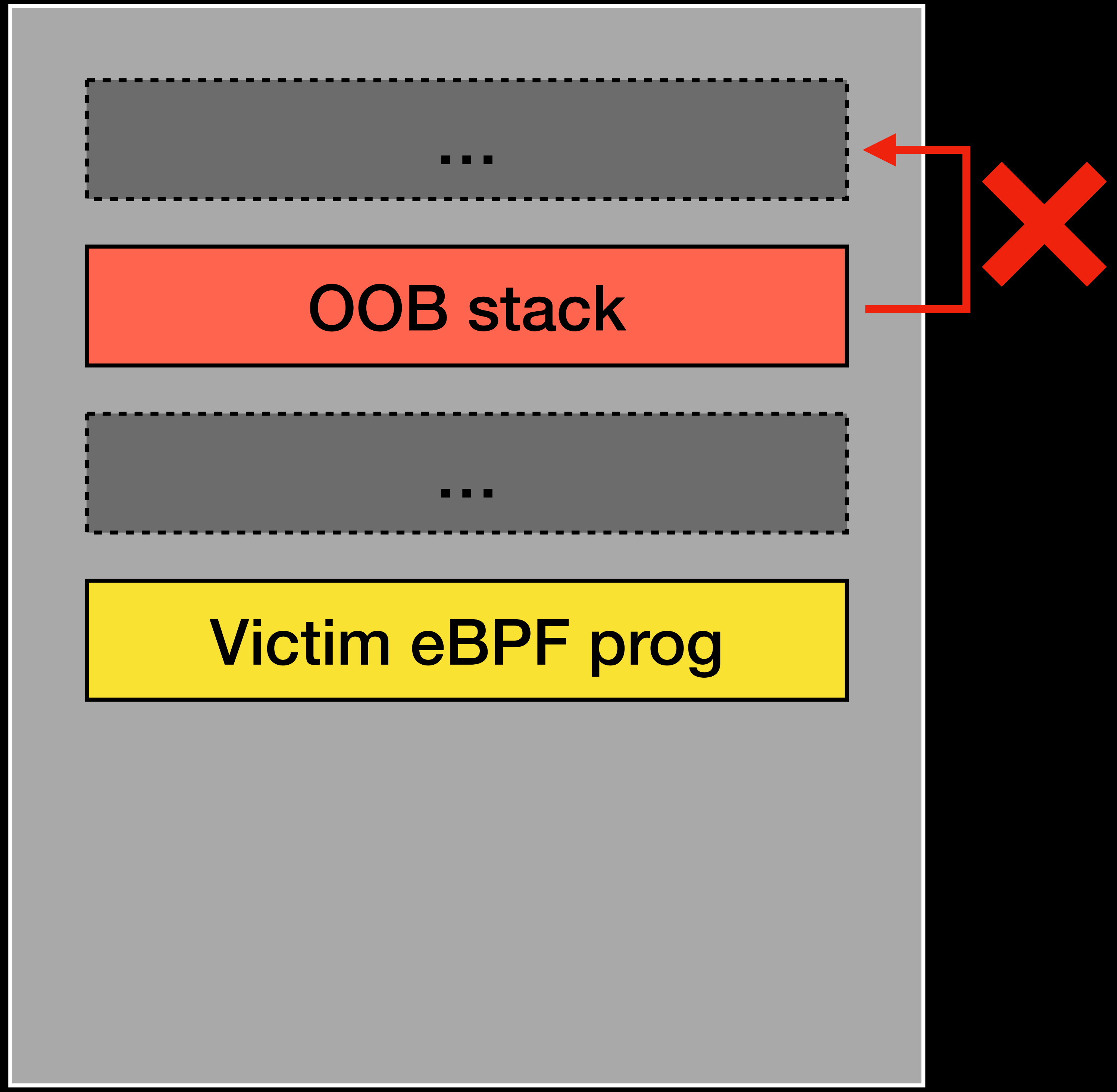


$ Heap Shaping



$ Heap Shaping

[Case 2]
Cached stacks are refilled



$ Heap Shaping

- To prevent these situations from occurring

❌SIGKILL-ing needless processes

1. The GNU session will be terminated if interdependent processes are killed
2. Some processes are still restarted by their parent processes, further worsening the situation

$ Heap Shaping

- To prevent these situations from occurring

❌ ~~SIGKILL-ing needless processes~~

✔ SIGSTOP-ing is more feasible

1. Daemons running as root will not generate any complaints, so there will be no side effects

2. Even if the processes freeze, we can send a SIGCONT to restore them

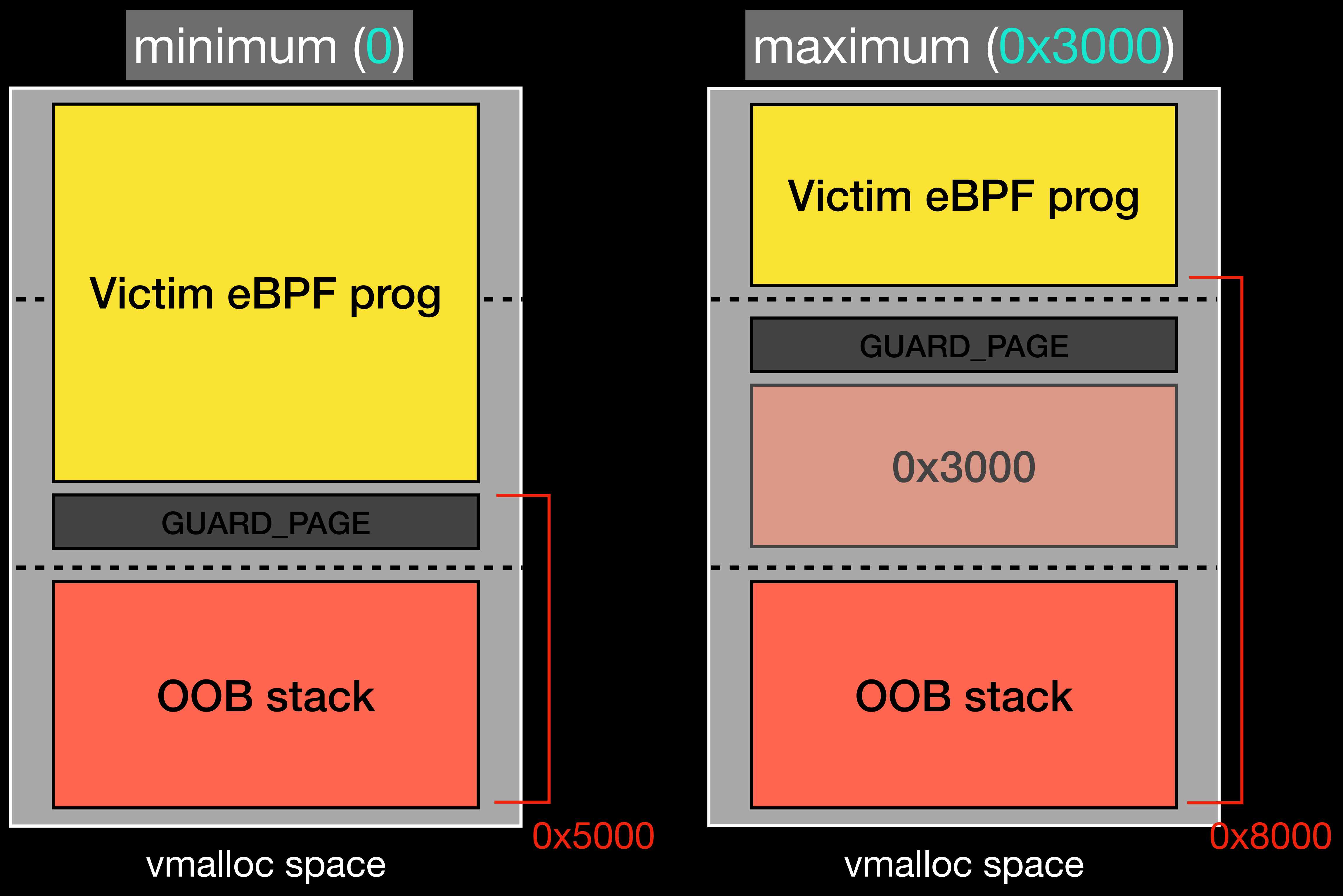
$ Heap Shaping

- Which out-of-bounds offsets should we use for exploitation?
  - The max eBPF program size is 0x5000

```
static bool __sk_filter_charge(struct sock *sk, struct sk_filter *fp)
{
	u32 filter_size = bpf_prog_size(fp->prog->len);
	int optmem_max = READ_ONCE(sysctl_optmem_max); // 0x5000

	/* same check as in sock_kmalloc() */
	if (filter_size <= optmem_max &&
	    atomic_read(&sk->sk_omem_alloc) + filter_size < optmem_max) {
		atomic_add(filter_size, &sk->sk_omem_alloc);
		return true;
	}
	return false;
}
```

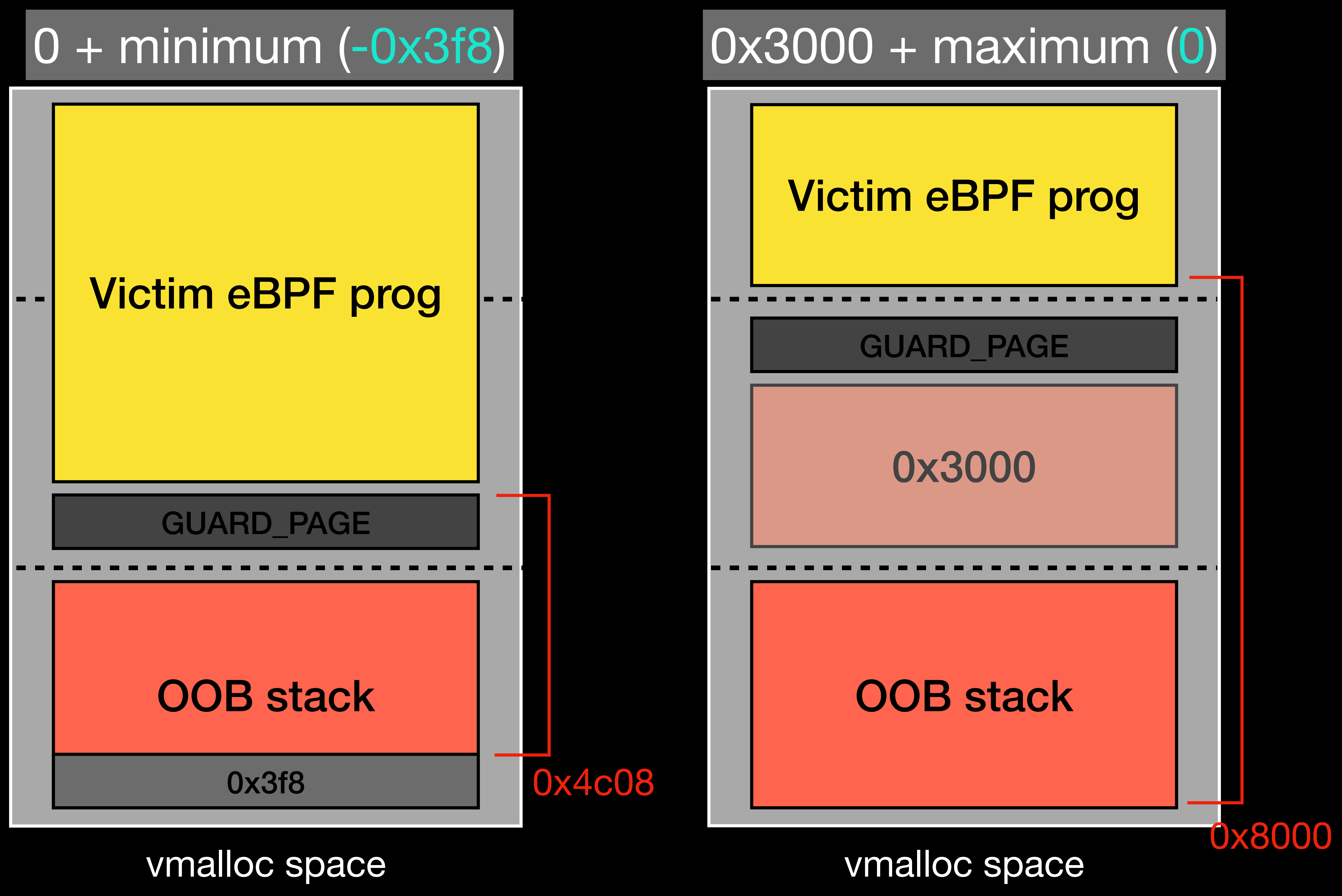
$ Heap Shaping

- Which out-of-bounds offsets should we use for exploitation?
  - The max eBPF program size is 0x5000
  - Alignment: 0 ~ 0x3000



$ Heap Shaping

- Which out-of-bounds offsets should we use for exploitation?
  - The max eBPF program size is 0x5000
  - Alignment: 0 ~ 0x3000
  - Randomization: -0x3f8 ~ 0

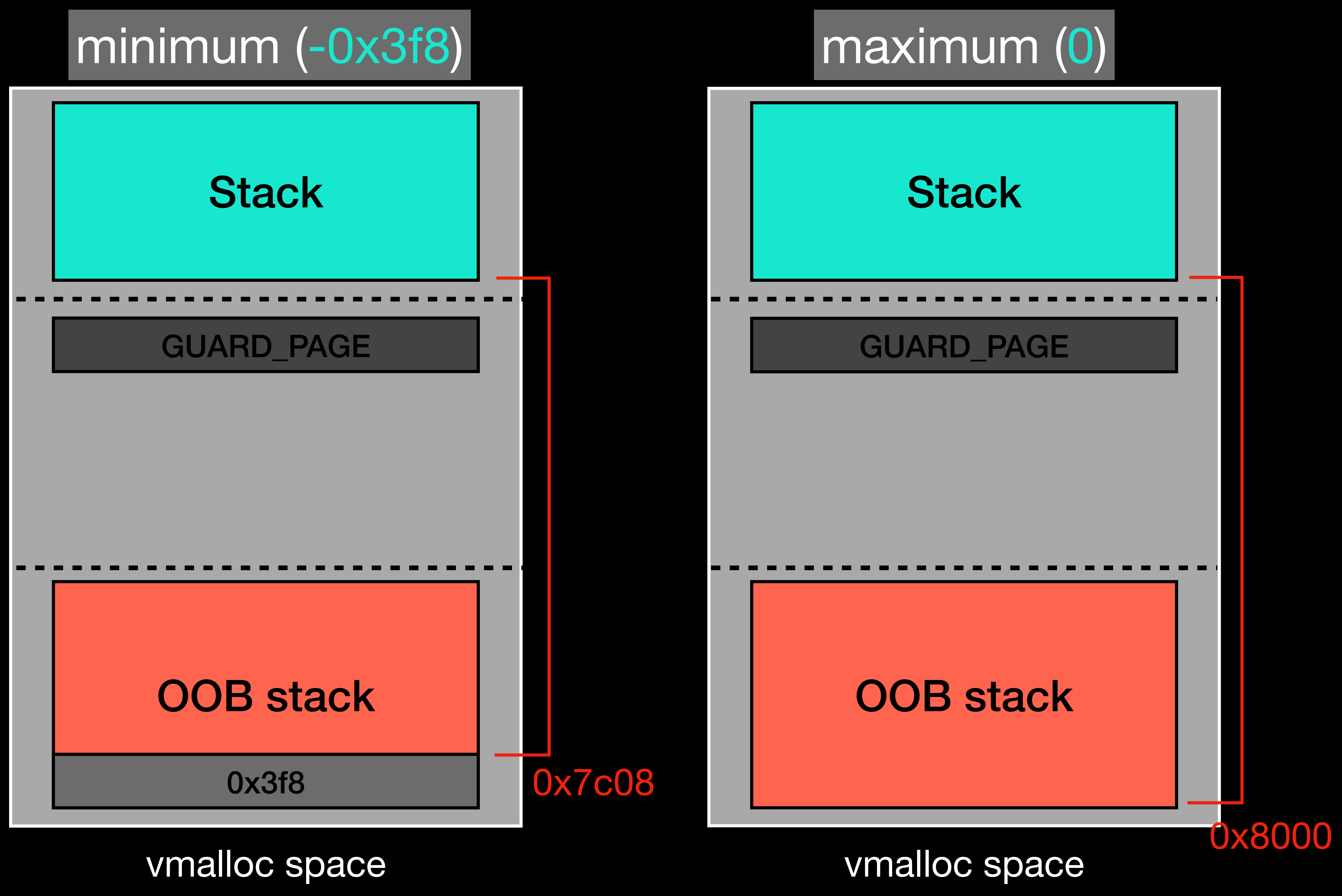


$ Heap Shaping

- Corresponding offset ranges for overwriting the eBPF program
    1. 0x4c08 to 0x9c08 (0x4c08 plus the max eBPF program size)
    2. 0x8000 to 0xd000 (0x8000 plus the max eBPF program size)
- The offset range 0x8000 to 0x9c08 is considered safe for overwriting the eBPF program

$ Heap Shaping

- SIGSTOP sent by a normal user does not work on root processes
- An unexpected stack is allocated above the OOB stack
    - The stack size is 0x4000

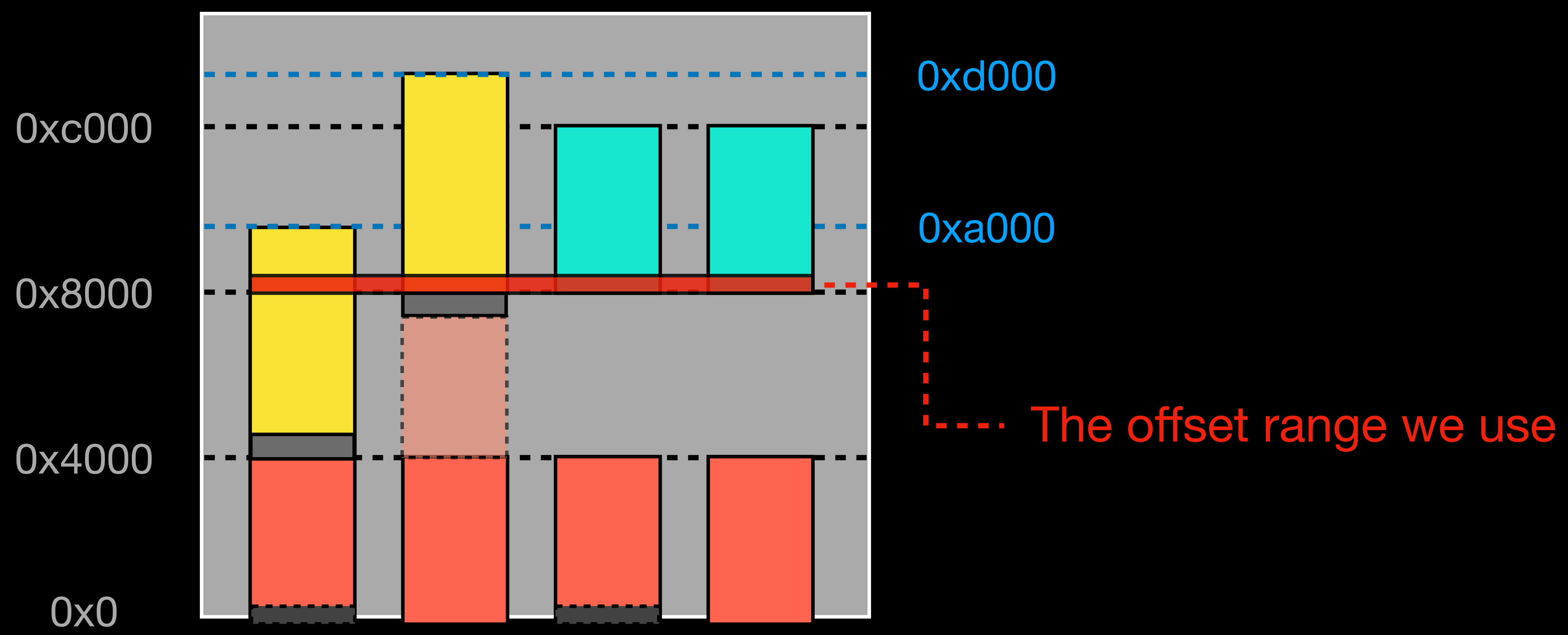


$ Heap Shaping

- Corresponding offset ranges for accessing the unexpected stack
    1. 0x7c08 to 0xbc08 (0x7c08 plus the stack size)
    2. 0x8000 to 0xc000 (0x8000 plus the stack size)
- The offset range 0x8000 to 0xbc08 is considered safe for overwriting the stack

$ Heap Shaping

- Finally, we obtained an offset range avoiding most panic situations, regardless of whether a new stack or a eBPF program is above
  - 0x8000 to 0x9c08
- In practice, the offset range needs to be adjusted due to the exploitation environment



$ Hijack modprobe_path

- The simplest way to escalate privilege is by overwriting modprobe_path
    1. Leak a kernel address to obtain the address of modprobe_path
    2. Construct an arbitrary write to overwrite the modprobe_path data

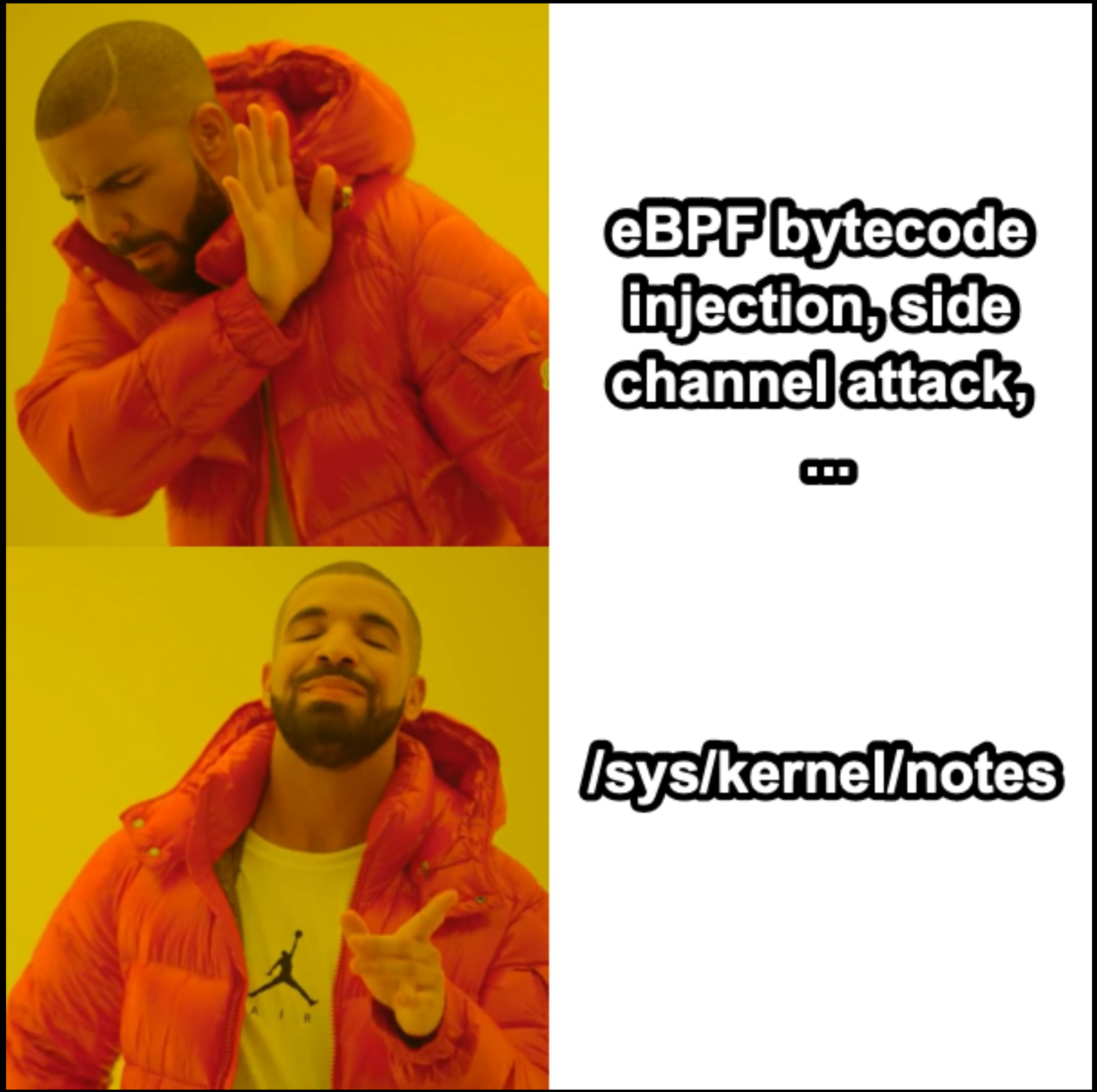
$ Hijack modprobe_path

- The simplest way to escalate privilege is by overwriting modprobe_path
    1. Leak a kernel address to obtain the address of modprobe_path
    2. Construct an arbitrary write to overwrite the modprobe_path data
- We cannot inject too many bytecode due to the limited race window
- The bytecode value also needs to be smaller than the MTU

$ Hijack modprobe_path

1. Leak a kernel address
    - Get startup_xen address from /sys/kernel/notes

```
aaa@aaa:~/Desktop$ sudo cat /proc/kallsyms | grep startup_xen
[sudo] password for aaa:
ffffffffa5094420 T startup_xen
aaa@aaa:~/Desktop$ xxd /sys/kernel/notes | grep "ffff ffff"
000000c0: 0000 0080 ffff ffff 0400 0000 0800 0000  ...........
000000f0: 2044 09a5 ffff ffff 0400 0000 1500 0000   D.........
00000190: 00d0 b3a3 ffff ffff 0400 0000 0400 0000  ...........
aaa@aaa:~/Desktop$ lsb_release -d
No LSB modules are available.
Description:    Ubuntu 23.10
```



$ Hijack modprobe_path

* **CVE-2024-26816: x86, relocs: Ignore relocations in .notes section**
@ 2024-04-10 13:54 Greg Kroah-Hartman
  0 siblings, 0 replies; only message in thread
From: Greg Kroah-Hartman @ 2024-04-10 13:54 UTC (permalink / raw)
  To: linux-cve-announce; +Cc: Greg Kroah-Hartman

Description
===========

In the Linux kernel, the following vulnerability has been resolved:

x86, relocs: Ignore relocations in .notes section

When building with CONFIG_XEN_PV=y, .text symbols are emitted into the .notes section so that Xen can find the "startup_xen" entry point. This information is used prior to booting the kernel, so relocations are not useful. In fact, performing relocations against the .notes section means that the KASLR base is exposed since /sys/kernel/notes is world-readable.

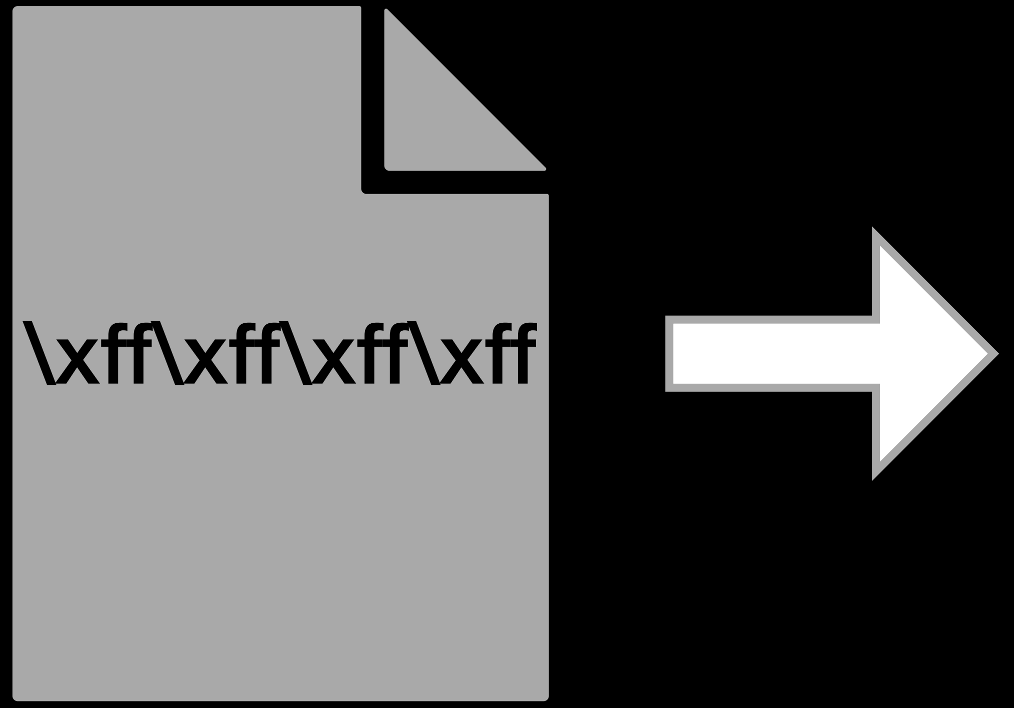
To avoid leaking the KASLR base without breaking unprivileged tools that are expecting to read /sys/kernel/notes, skip performing relocations in the .notes section. The values readable in .notes are then identical to those found in System.map.

$ Hijack modprobe_path

2. Construct an arbitrary write

- Goal: overwrite modprobe_path from “/sbin/modprobe” to “/tmp//modprobe”

Unknown executable format



Function call_modprobe

```
argv[0] = modprobe_path;
argv[1] = "-q";
argv[2] = "--";
argv[3] = module_name;
argv[4] = NULL;

info = call_usermodehelper_setup(modprobe_path, argv, envp, GFP_KERNEL,
                                 NULL, free_modprobe_argv, NULL);
ret = call_usermodehelper_exec(info, wait | UMH_KILLABLE);
```

Writable kernel data

```
char modprobe_path[KMOD_PATH_LEN] = CONFIG_MODPROBE_PATH;
```

/sbin/modprobe

$ Hijack modprobe_path

2. Construct an arbitrary write

- Setup eBPF program registers by normal filter bytecode

```
val = (modprobe_path + 1) & 0xffffffff;
val = (1UL << 32) - val;

filter[i++] = (struct sock_filter){.code = BPF_LD   | BPF_IMM, .k = 0x2f706d74};
filter[i++] = (struct sock_filter){.code = BPF_MISC | BPF_TAX, .k = 0};
filter[i++] = (struct sock_filter){.code = BPF_LD   | BPF_IMM, .k = val};
```

Filter bytecode

r0	~~0~~ ~(mobprobe_path + 1)
r1	0
r7	~~0~~ 0x2f706d74

eBPF registers

$ Hijack modprobe_path

2. Construct an arbitrary write

   - Inject 2 malicious eBPF bytecodes
     - 0x41F BPF_ALU64_REG(BPF_SUB, BPF_REG_1, BPF_REG_0)
     - 0x7463 BPF_STX_MEM(BPF_W, BPF_REG_1, BPF_REG_0)

$ Hijack modprobe_path

2. Construct an arbitrary write

- Inject 2 malicious eBPF bytecodes
  - 0x41F BPF_ALU64_REG(BPF_SUB, BPF_REG_1, BPF_REG_0)
  - 0x7463 BPF_STX_MEM(BPF_W, BPF_REG_1, BPF_REG_0)

$$
\begin{aligned}
r_1 &= r_1 - r_0 \\
&= 0 - \sim(\text{modprobe\_path} + 1) \\
&= \text{modprobe\_path} + 1
\end{aligned}
$$

Bytecode 0x41F

r0	~(mobprobe_path + 1)
r1	~~0~~ mobprobe_path + 1
r7	2F706D74

eBPF registers

$ Hijack modprobe_path

2. Construct an arbitrary write
   - Inject 2 malicious eBPF bytecodes
     - 0x41F BPF_ALU64_REG(BPF_SUB, BPF_REG_1, BPF_REG_0)
     - 0x7463 BPF_STX_MEM(BPF_W, BPF_REG_1, BPF_REG_0)

$$
\begin{aligned}
[r_1] &= r_7 \\
&= \text{"/tmp//modprobe"}
\end{aligned}
$$

Bytecode 0x7463

r0	~(mobprobe_path + 1)
r1	mobprobe_path + 1
r7	2F706D74 (“tmp/“)

eBPF registers

`char modprobe_path[KMOD_PATH_LEN]`

/tmp//modprobe

- Nov 28 2023 Target Selection
- Jan 19 2024 Bug Discovery
- Feb 21 2024 Crafting the Exploit
- Mar 20 2024 Achieving LPE
- Nov 7 2024 Takeaways

$ Chain All Together

$ Chain All Together

$ Chain All Together

$ Chain All Together

$ Chain All Together

$ Chain All Together

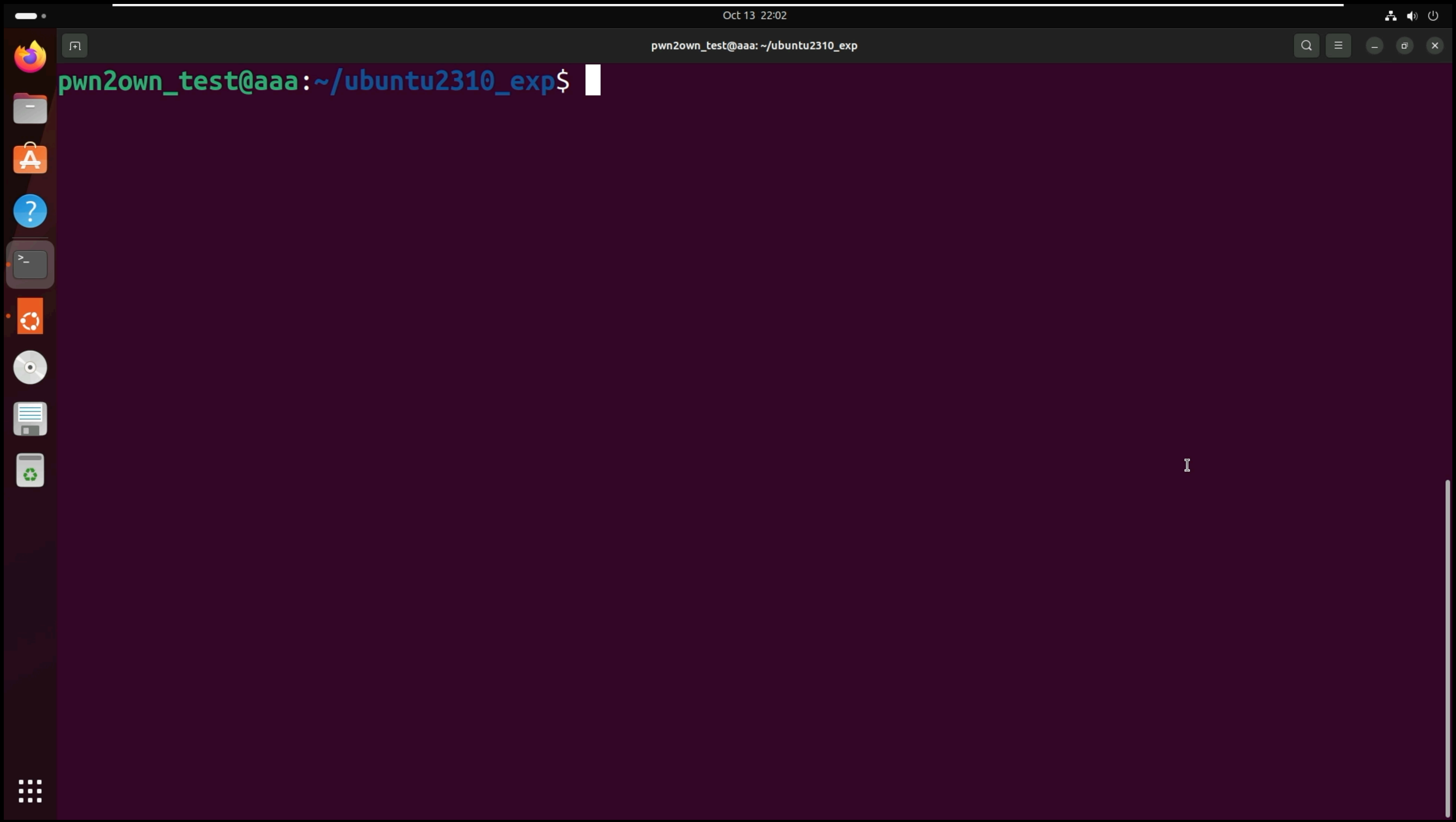
$ Chain All Together

$ Chain All Together

$ Chain All Together

- It is not possible to filter out all noise, such as vmalloc invoked by root processes or kernel threads
- Achieving a 100% success rate remains challenging
- But it is sufficient under Pwn2Own's three-attempt rule 🙂

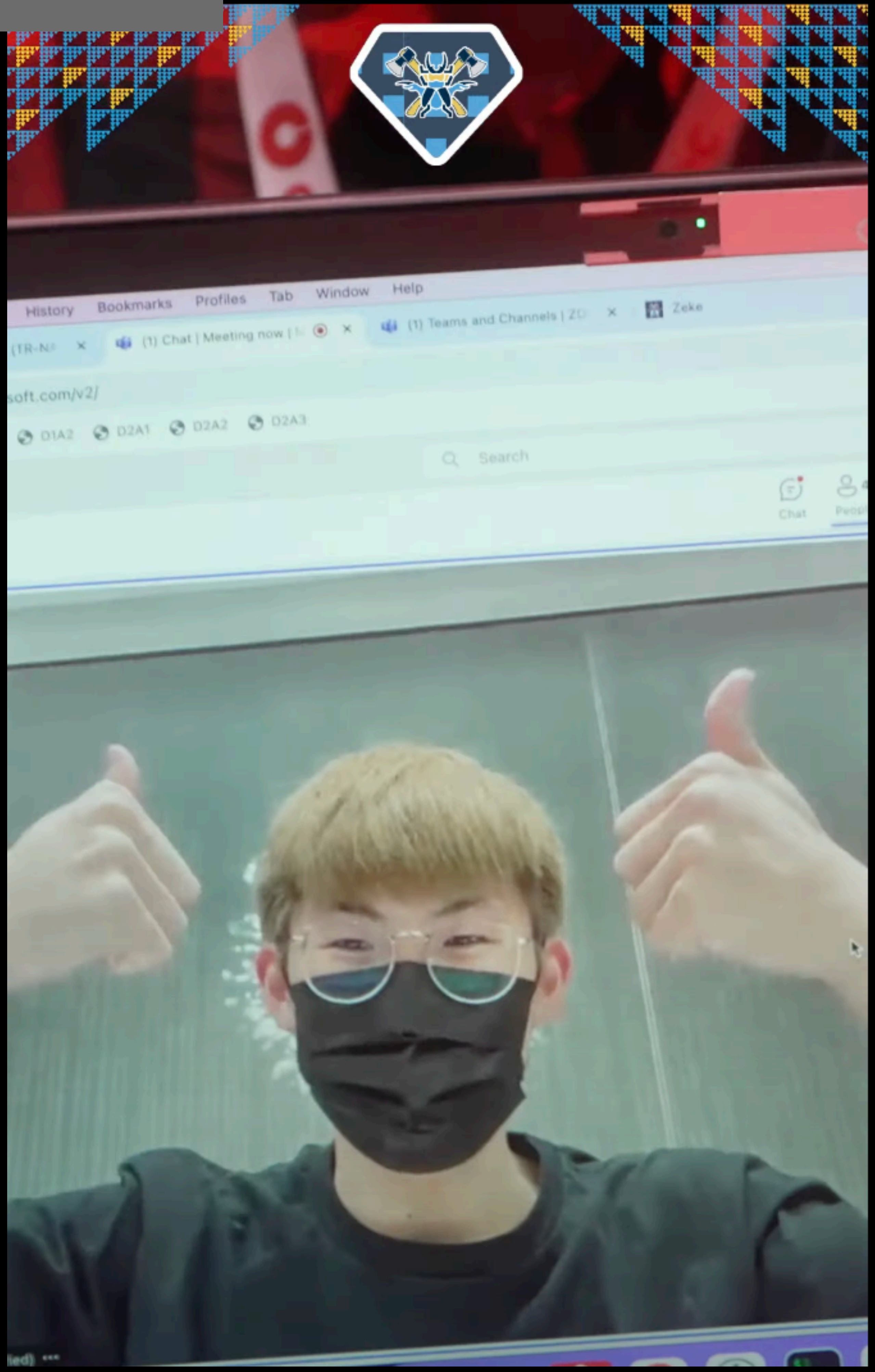
$ Demo



Oct 13 22:02
pwn2own_test@aaa: ~/ubuntu2310_exp
pwn2own_test@aaa:~/ubuntu2310_exp$

$ Demo

We won !!



Ref: https://youtu.be/fXUrMIM2KYc

- Nov 28 2023 Target Selection
- Jan 19 2024 Bug Discovery
- Feb 21 2024 Crafting the Exploit
- Mar 20 2024 Achieving LPE
- Nov 7 2024 Takeaways

$ Takeaways

- Memory allocation in the vmalloc space is exploit-friendly
- (Unprivileged) eBPF remains a valuable gadget for exploitation
- SIGSTOP is a simple and effective way to reduce memory noise
- Exploring new attack surfaces in Ubuntu is inevitable

DEVCORE

Thanks!

Pumpkin 🎃 (@u1f383)
https://u1f383.github.io/