# Unicorn's Aisle

**Escape unicorn emulator with features and 0-day**

jwang@Balsn

# Prelude

Understand the Guest Program

# Minimalistic FTG

- Server
  - main.c / main.h / guestcontext.h
  - mem.c / mem.h
  - handler.c / handler.h
  - utils.c / utils.h
  - ucutils.c / ucutils.h
  - encounter.S

- Client
  - didn’t finish in time, not released

# Game Logic (encounter.S)

```
def main():
        confrontState = getMem()
        resetWeaponMemPool()
        while True:
          actionType = adventurerAction(confrontState)
          if actionType==1:
            break
          if actionType!=2:
            unicornAction(confrontState)
          sendState(confronState)
          if actionType!=2:
            doAction(confrontState)
            if checkConfrontOver(confrontState)!=0:
              break
      write(1,"game over\n",10)
      exit(0)
```

# Game Logic (encounter.S)

- Standard FTG actions are supported
  - move
  - attack
  - defend
- An additional weapon system is introduced
  - craft weapon
  - switch weapon
  - enhance weapon
  - dispose weapon

# Target

- When player wins game (unicorn HP falls to 0), encounter.S does syscall 0xfedcba9876543210

```python
def checkConfrontOver(confrontState):
  if confrontState.isConfront==0:
    return 0
  if confrontState.unicornHP==0:
    syscall(0xfedcba9876543210)
    exit(0)
  if confrontState.adventurerHP==0:
    return 1
  return 0
```

- **preludeHandler** defined in handler.c handles syscall and prints flag

```c
void preludeHandler(uc_engine *uc,uint32_t intno
                    void *guestContext){
  SYSCALLCONTEXT regval;
  char buf[0x100];
  int fd,size;
  fetchSyscallRegs(uc,&regval);
  if(regval.rax!=0xfedcba9876543210)
    return;
  fd = open("./unicornPrelude",O_RDONLY);
  if(fd<0) printError("preludeHandler::open failed");
  size = read(fd,buf,0xff);
  if(size<0) printError("preludeHandler::read failed");
  buf[size] = '\0';
  puts(buf);
  _exit(0);
}
```

# Game Logic (encounter.S)

- The enemy (unicorn) has extremely high attack/defense
- Attack pattern spawns attacks from behind player character, making it almost impossible to effectively defend against attack
- It is nearly impossible to win the game by playing normally

# Game Logic (encounter.S)

- Game actions are exchanged with pure IO
- Server keeps track of a global confrontState
- Client actuates by sending actionStruct, union field contains action specific content
- Server responses game state / action result by sending entire confrontState excluding adventurerWeapons array to client

```
struc confrontState
  .unicornRand resq 1
  .isConfront resq 1
  .unicornStage resq 1
  .unicornHP resq 1
  .unicornAttack resq 1
  .unicornDefense resq 1
  .unicornState resq 1
  .unicornAttribute resq 1
  .unicornCD resq 1
  .unicornDefenseGauge resq 1
  .unicornLoc resq 1
  .unicornAttackSourceLoc resq 1
  .unicornAttackBoxLoc resq 1
  .adventurerHP resq 1
  .adventurerAttack resq 1
  .adventurerDefense resq 1
  .adventurerState resq 1
  .adventurerCD resq 1
  .adventurerDefenseGauge resq 1
  .adventurerLoc resq 1
  .adventurerAttackSourceLoc resq 1
  .adventurerAttackBoxLoc resq 1
  .adventurerWeaponIdx resq 1
  .adventurerName resb MAX_NAME_LEN
  .adventurerDesc resb MAX_DESC_LEN
  .adventurerWeapons reso MAX_WEAPON_IDX
endstruc

struc weaponEntry
  .ptr resq 1
  .size resq 1
endstruc

struc weapon
  .level resq 1
  .attack resq 1
  .defense resq 1
  .attribute resq 1
  .CD resq 1
  .range resq 1
  .delta resq 1
  .movespeed resq 1
  .name resb MAX_NAME_LEN
  .desc resb MAX_DESC_LEN
endstruc

struc actionStruct
  .action resq 1
  .union resb 0x278
endstruc
```

# Game Logic (encounter.S)

- To craft weapon, game client should send a structure including weapon stats to server

- Players can craft an arbitrarily strong weapon and equip it

- Attack with strong weapon to win game and retrieve flag

```
struc craftWeaponStruct
  .namelen resq 1
  .desclen resq 1
  .attack resq 1
  .defense resq 1
  .attribute resq 1
  .CD resq 1
  .range resq 1
  .delta resq 1
  .movespeed resq 1
  .name resb MAX_NAME_LEN
  .desc resb MAX_DESC_LEN
endstruc
```

# Interlude

## Guest Semi-Arbitrary Code Execution

# Target

- Now we know how to interact with game server, and successfully retrieved first flag

- Time to carry on to second flag

- Apparently we need to achieve guest code execution to trigger **interludeHandler**

```c
void interludeHandler(uc_engine *uc,uint32_t intno,
                      void *guestContext){
  SYSCALLCONTEXT regval;
  char key[0x10] = "unicorn";
  char buf[0x100];
  int fd,size;
  fetchSyscallRegs(uc,&regval);
  for(int i=0;i<7;i++){
   if((((size_t*)&regval)[i]&0xffffffffffffff00ULL)
      !=0xffffffffffffff00ULL) return;
   if((((size_t*)&regval)[i]&0xff)!=key[i]) return;
  }
  fd = open("./unicornInterlude",O_RDONLY);
  if(fd<0) printError("interludeHandler::open failed");
  size = read(fd,buf,0xff);
  if(size<0) printError("interludeHandler::read failed");
  buf[size] = '\0';
  puts(buf);
  _exit(0);
}
```

# Server Code Review-encounter.S

The first bug is a memory leak while starting a new battle

Since name/desc length is passed by user, and not directly calculated from input size

It is possible to specify name/desc length larger than actual input and read uninitialized stack content into confrontState structure

The values can then be retrieved when server sends confrontState to client

```
struc startBattleStruct
  .namelen resq 1
  .desclen resq 1
  .name resb MAX_NAME_LEN
  .desc resb MAX_DESC_LEN
endstruc

def parseAction(confrontState,actionStruct):
  if confrontState.isConfront==0:
    if actionStruct.action==STARTBATTLE:
      confrontState.isConfront = 1
      namelen = actionStruct.startBattleStruct.namelen
      if namelen>MAX_NAME_LEN:
        namelen = MAX_NAME_LEN
      memcpy(confrontState.adventurerName,
             actionStruct.startBattleStruct.name,
             namelen)
      desclen = actionStruct.startBattleStruct.desclen
      if desclen>MAX_DESC_LEN:
        desclen = MAX_DESC_LEN
      memcpy(confrontState.adventurerDesc,
             actionStruct.startBattleStruct.desc,
             desclen)
      ...
      return 0
    ...
  ...

def adventurerAction(confrontState):
  char buf[0x280]
  recvAction(buf,0x280)
  parseAction(confrontState,buf)
  return

def recvAction(buf,size):
  l = 4
  cursor = buf
  while l>0:
    res = read(0,cursor,l)
    if res==SYSCALLFAIL:
      exit(0)
    l-=res
    cursor+=res
  l = *(dword*)buf
  if l>size:
    exit(0)
  cursor = buf
  while l>0:
    res = read(0,cursor,l)
    if res==SYSCALLFAIL:
      exit(0)
    l-=res
    cursor+=res
  return
```

# Server Code Review-encounter.S

The second bug is that there are no checks on the idx when disposing weapons, allowing us to perform **munmap** on OOB array entry

Since we can leak contents beforehand, this is basically a **munmap** with arbitrary argument

At first glance, this bug might seem useless, but let's dive deeper before making conclusions

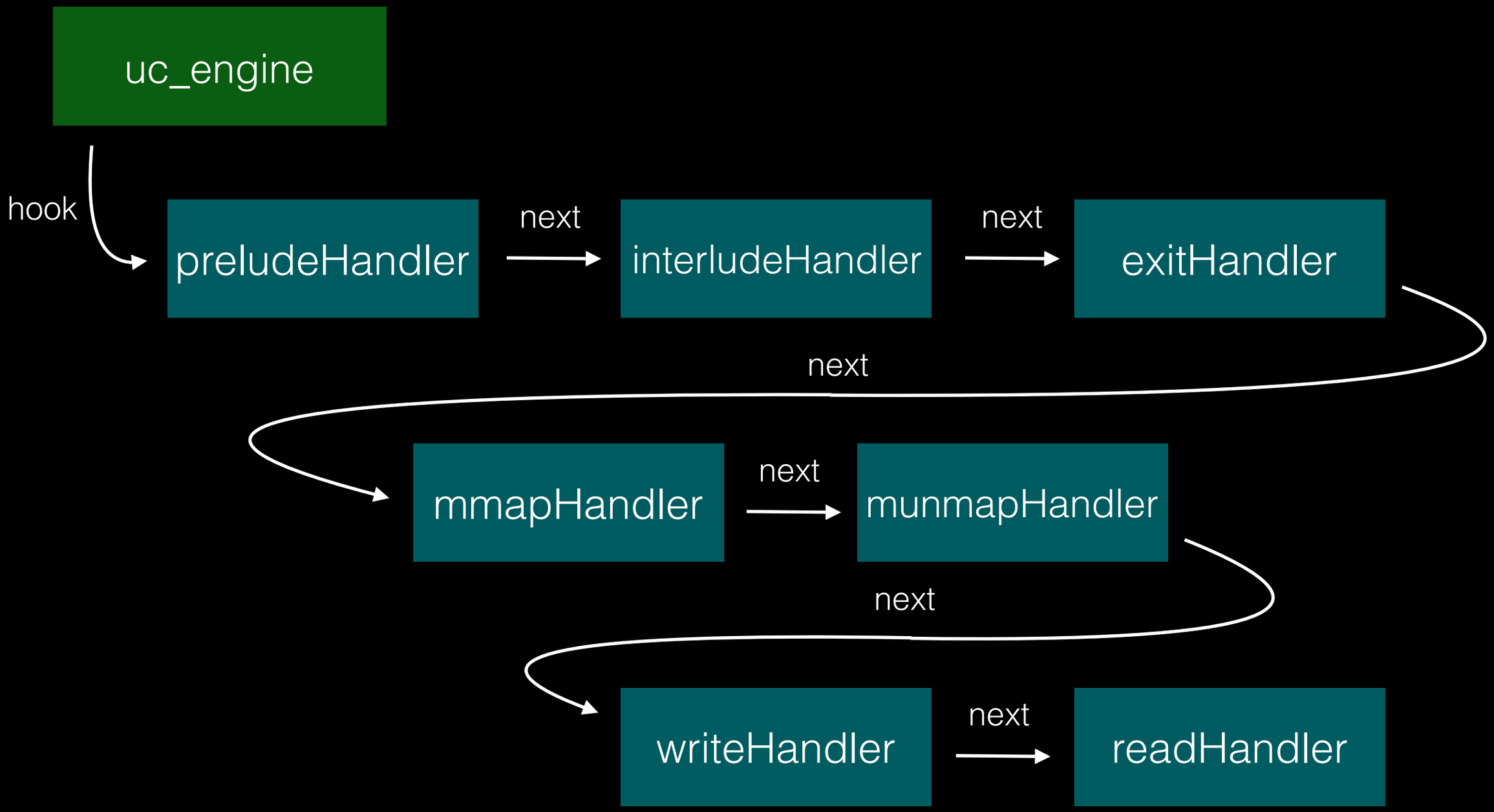
```
struc disposeWeaponStruct
  .targetIdx resq 1
endstruc

struc weaponEntry
  .ptr resq 1
  .size resq 1
endstruc

def parseAction(confrontState,actionStruct):
  ...
    elif actionStruct.action==DISPOSEWEAPON:
      if actionStruct.disposeWeaponStruct.targetIdx==
           confrontState.adventurerWeaponIdx:
        confrontState.adventurerWeaponIdx = SLOT_NONE
      curWeapon = confrontState.adventurerWeapons[
                    actionStruct.disposeWeaponStruct.targetIdx]
      munmap(curWeapon.ptr,curWeapon.size)
      curWeapon.ptr = NULL
      curWeapon.size = 0
      return 2
  ...
```

# Server Code Review-Syscall

- unicorn keeps track of syscall hooks with a linked list
- newly registered syscalls will be inserted at start of linked list by default
- Entire linked list is traversed and processed in order when sys call is invoked

```
1  void helper_syscall(CPUX86State *env, int next_eip_addend)
2  {
3      // Unicorn: call registered syscall hooks
4      struct hook *hook;
5      HOOK_FOREACH_VAR_DECLARE;
6      HOOK_FOREACH(env->uc, hook, UC_HOOK_INSN) {
7          if (hook->to_delete)
8              continue;
9          if (!HOOK_BOUND_CHECK(hook, env->eip))
10             continue;
11         if (hook->insn == UC_X86_INS_SYSCALL)
12             ((uc_cb_insn_syscall_t)hook->callback)(env->uc, hook->user_data);
13     }
14
15     env->eip += next_eip_addend;
16     return;
17 }
```

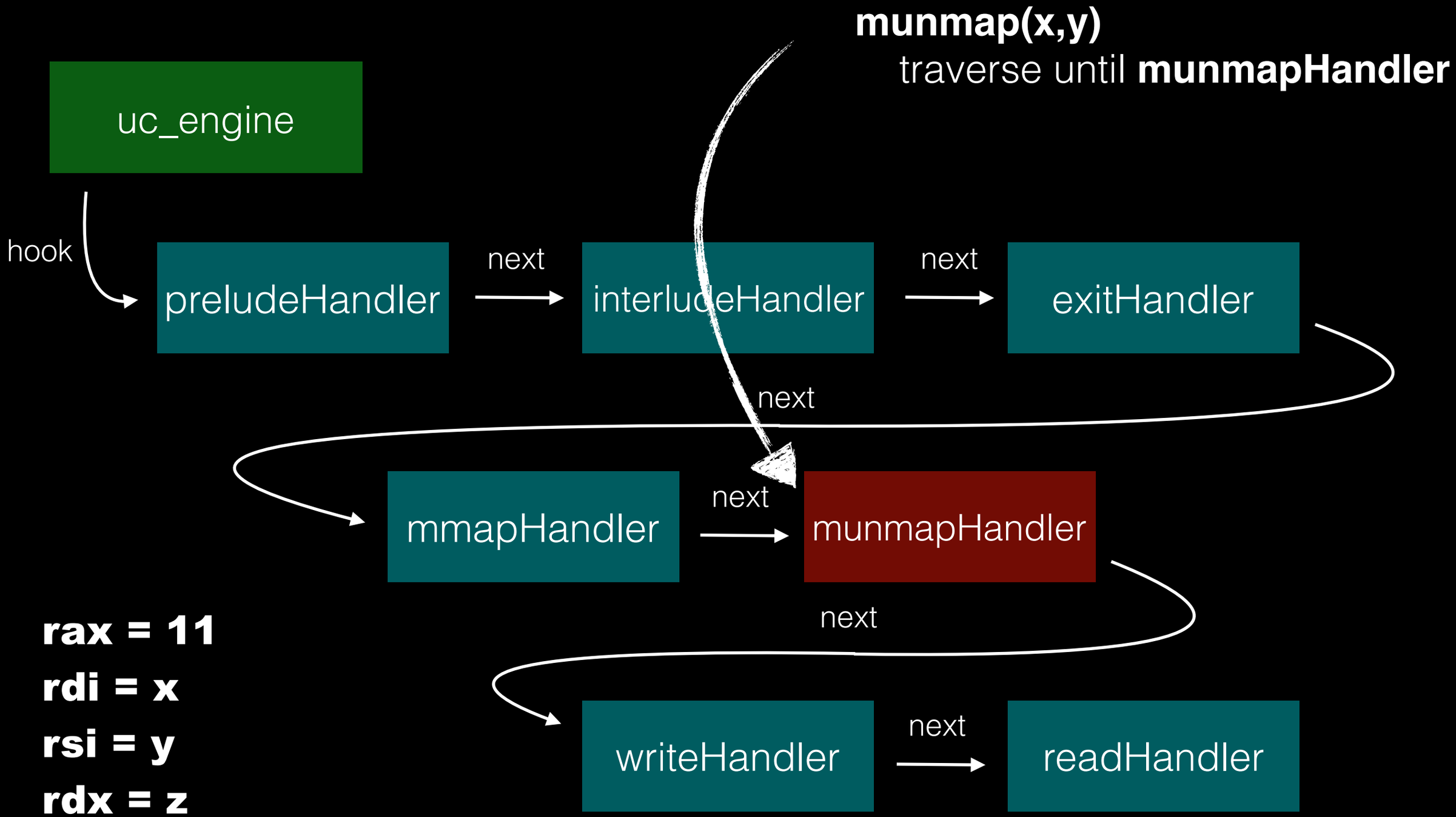
# Server Code Review-Syscall



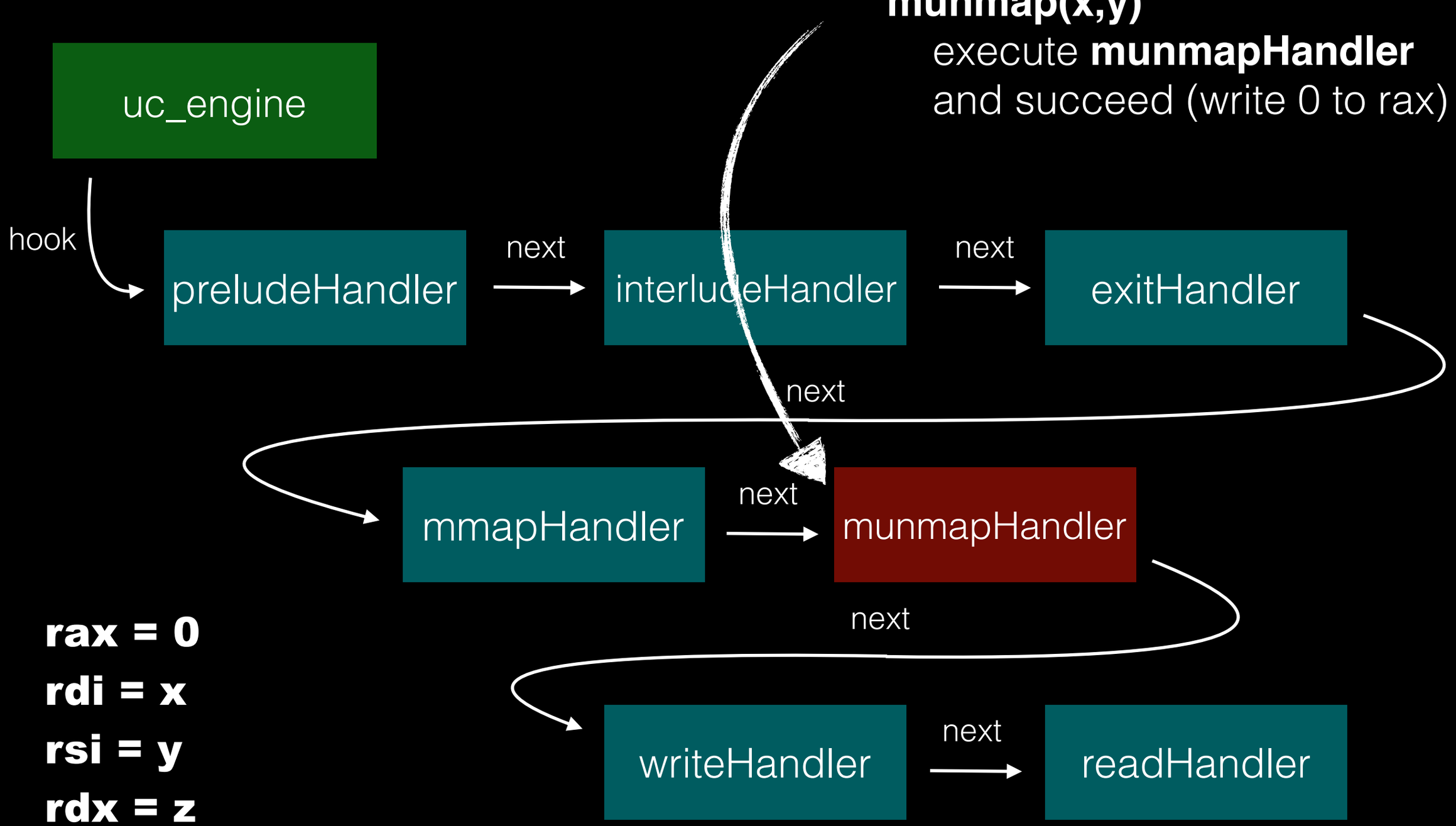
# Server Code Review-Syscall

- what's wrong with the setup?
- syscall hooks are processed in order
  - If hooks do not alter emulator state, everything works normally
  - But syscall hooks will write rax to return value, creating side-effects
    - This enables an interesting attack primitive which I named as Syscall Oriented Programming (SOP) here

# Syscall Oriented Programming

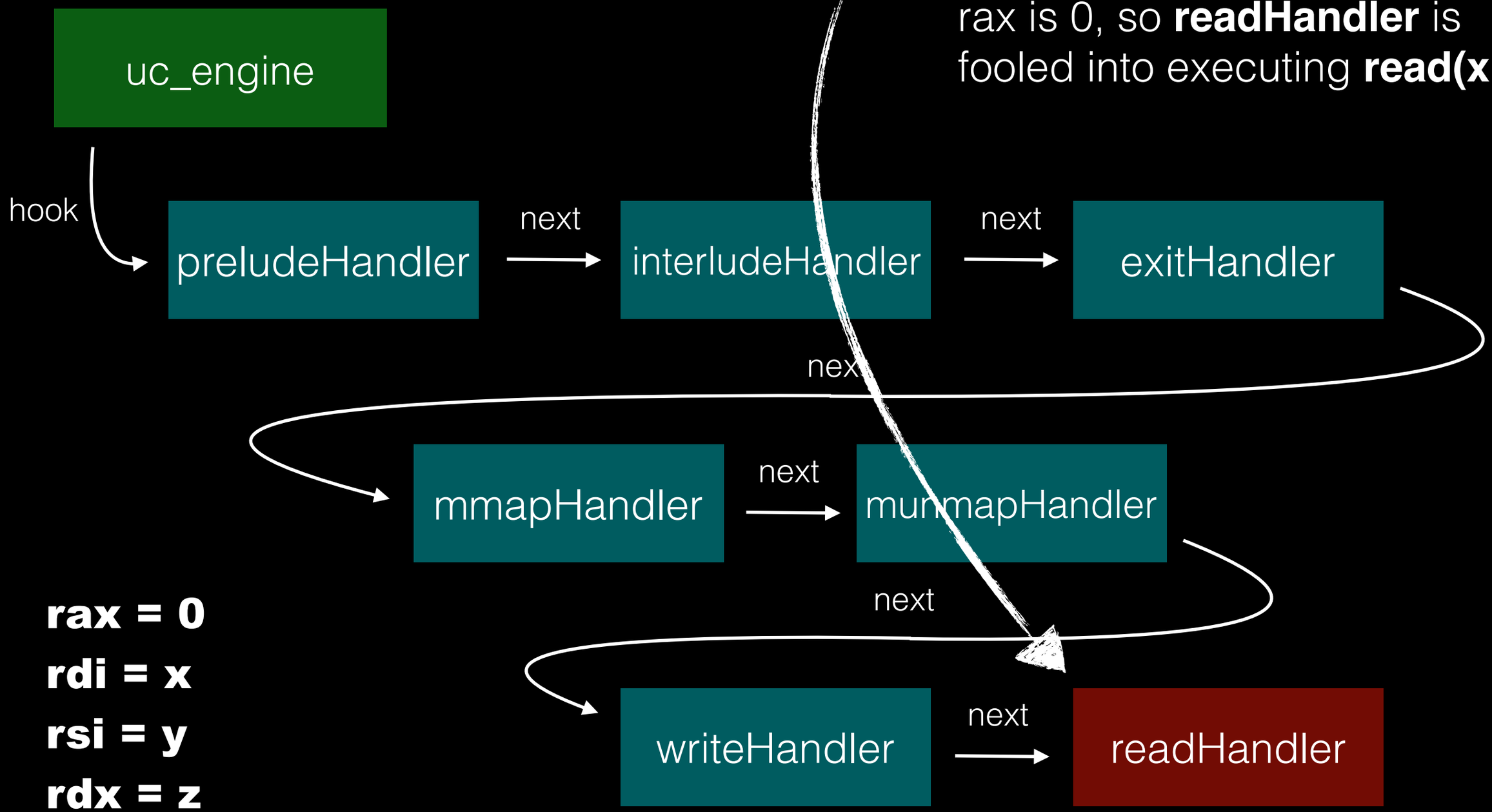


# Syscall Oriented Programming



# Syscall Oriented Programming

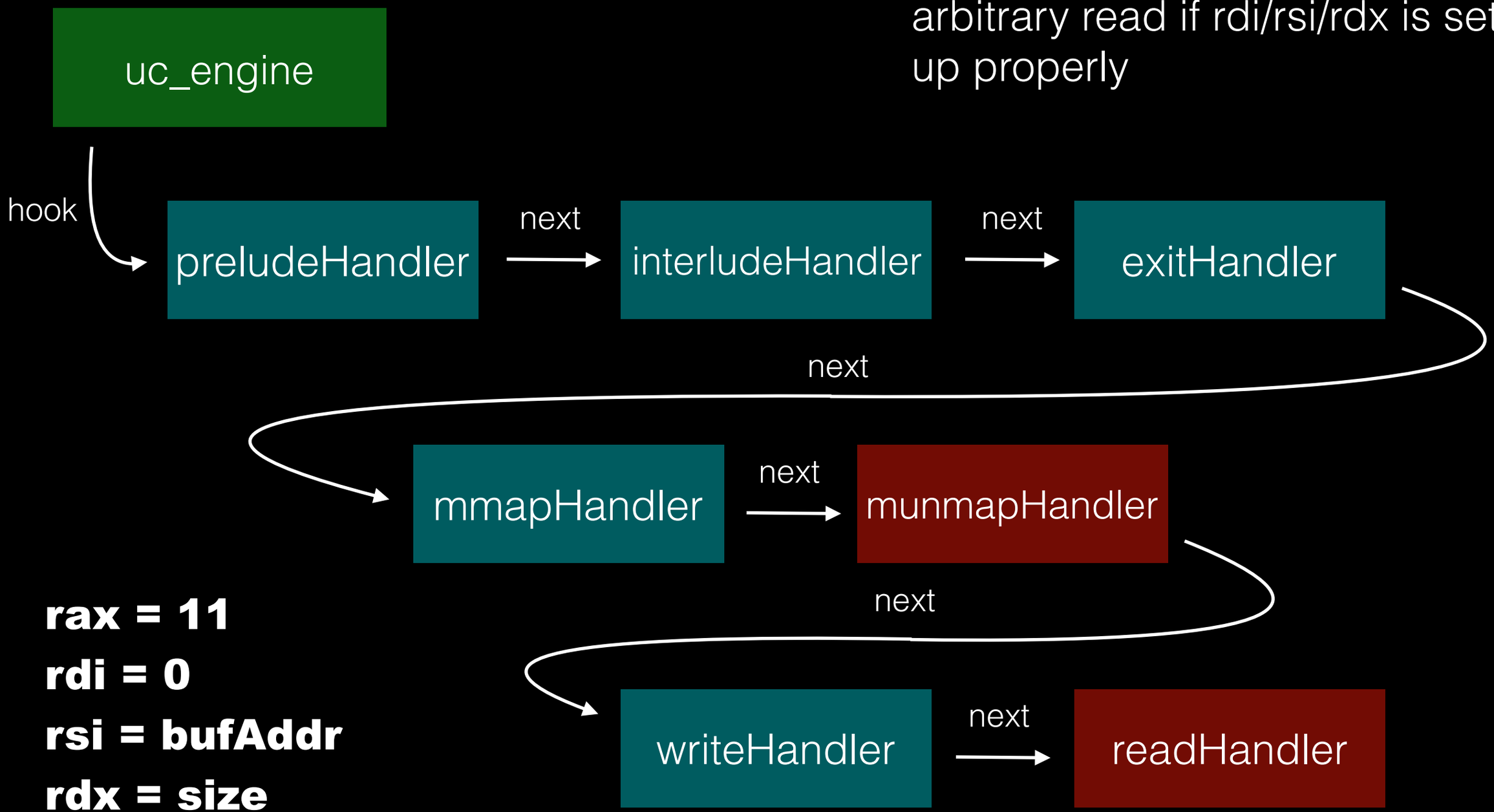
**munmap(x,y)**
rax is 0, so **readHandler** is fooled into executing **read(x,y,z)**




**rax = 0**
**rdi = x**
**rsi = y**
**rdx = z**

# Syscall Oriented Programming

This results in an unexpected arbitrary read if rdi/rsi/rdx is set up properly

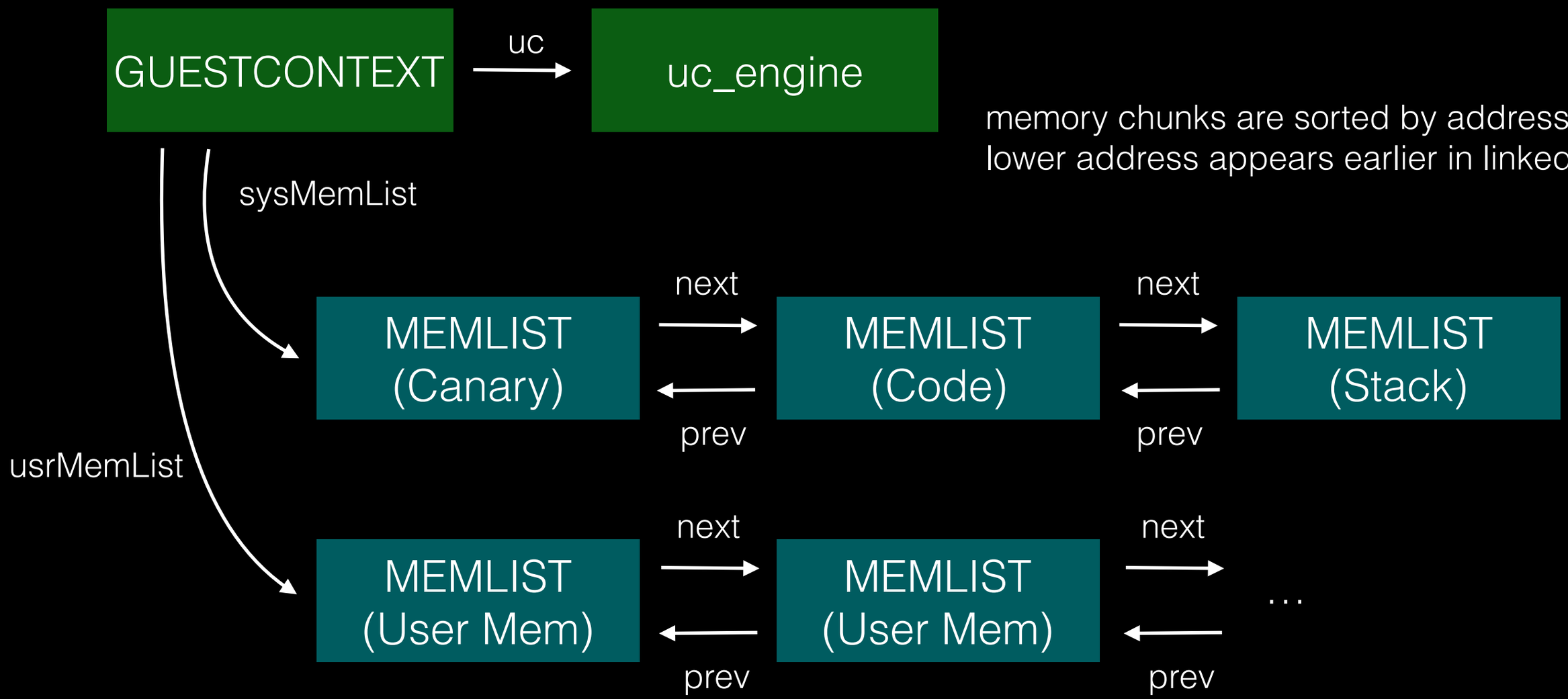



**rax = 11**
**rdi = 0**
**rsi = bufAddr**
**rdx = size**

# Server Code Review-Syscall

- So now we have SOP due to side effects
- And we have identified a potentially utilizable attack chain
- How can we utilize this bug?

# Server Code Review-Memory



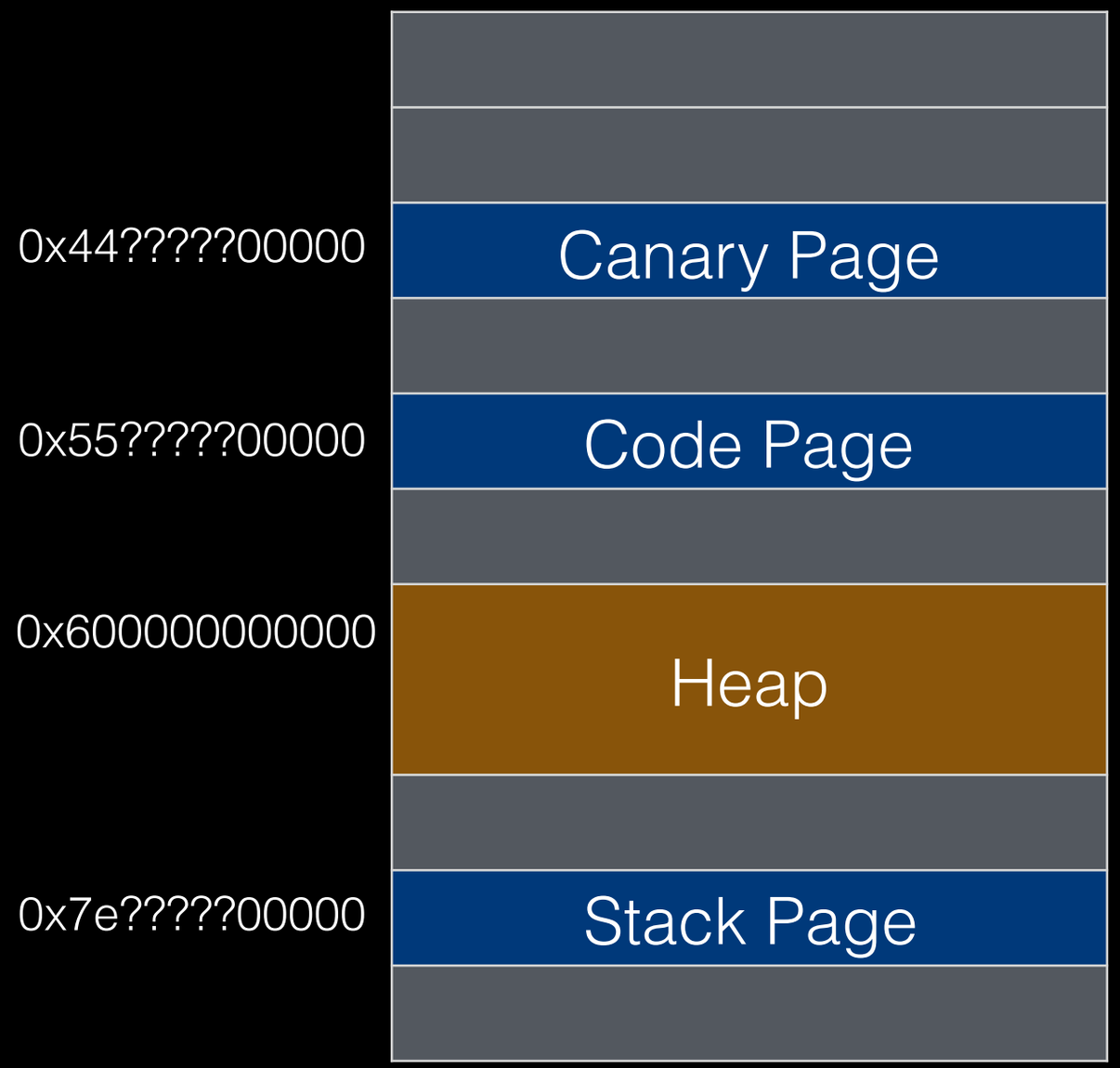

memory chunks are sorted by address,
lower address appears earlier in linked list

memory chunks allocated by guest program will be inserted in usrMemList,
and only userMemList are allowed to be released
additionally, guest is not allowed to allocate executable memory

# Server Code Review-Memory

The memory layout for game is as right graph, and only heap can be claimed/released by encounter.S

Canary/Code/Stack are system memory and NOT releasable



# Server Code Review-Memory

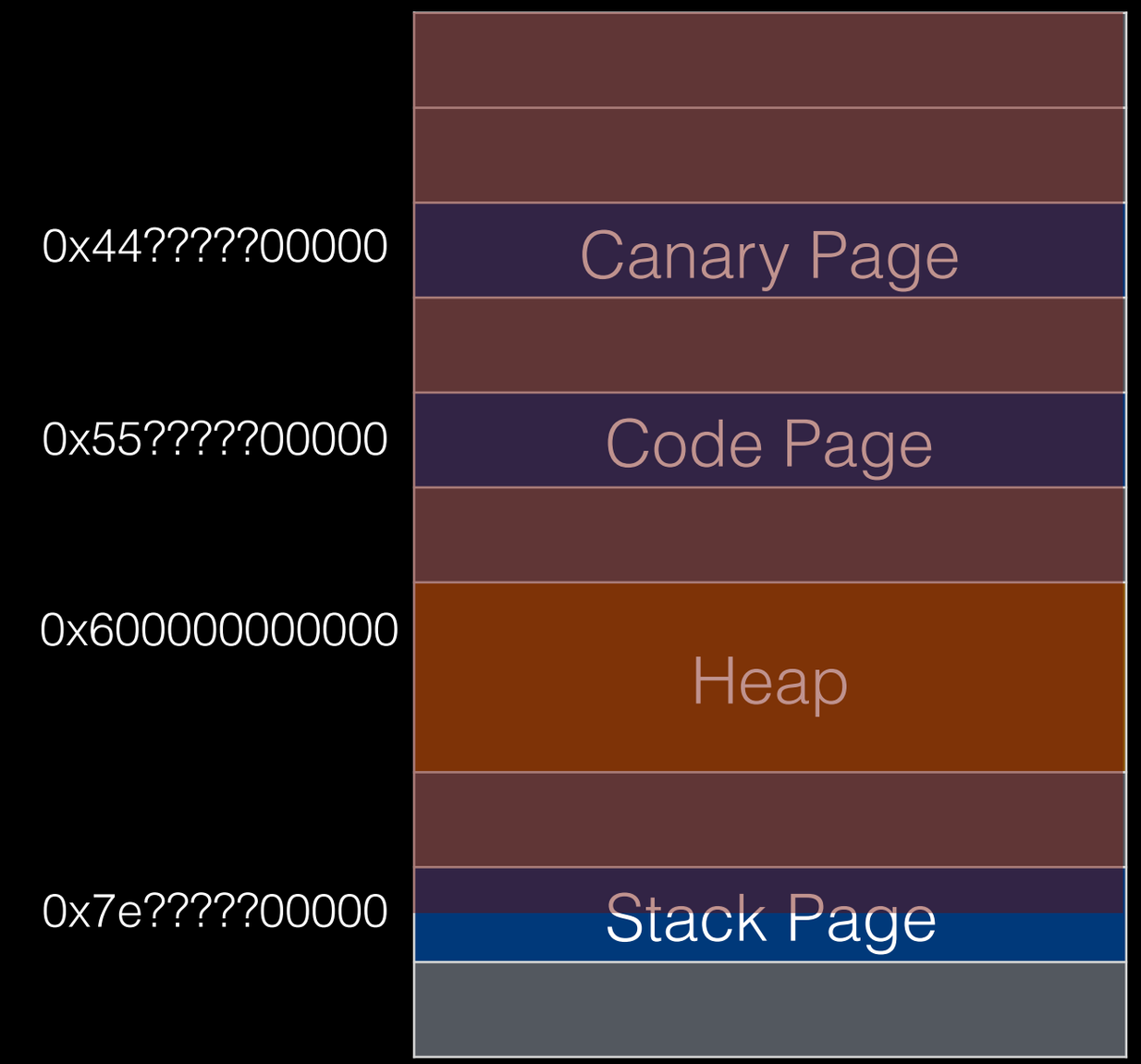
**munmap**

rdi = 0

rsi = some stack address

rdx = non 0 values

Only heap will be freed, canary/code/stack is kept intact due to design of separate user/system memory



# Server Code Review-Memory

**munmap** -> **read**

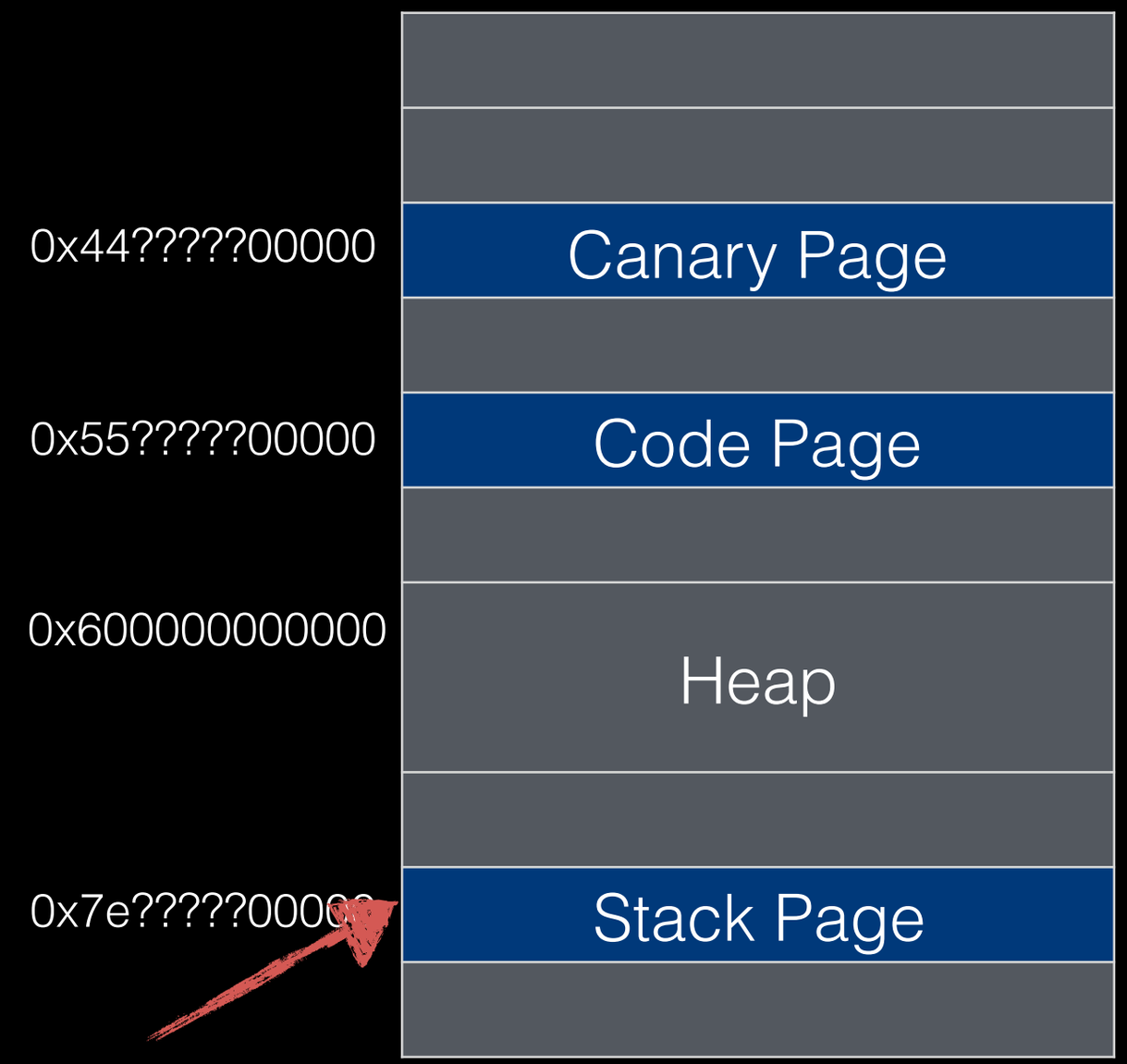
rdi = 0

rsi = some stack address

rdx = non 0 values

The unexpected read then allows overwrite of stack content

Can modify return address!



# Server Code Review- Calling Convention

Hijack return address!=ROP

encounter.S uses a custom calling convention similar to that of a VM

This minimizes appearance of ROPgadgets

It is not possible to construct any meaningful ROPchain with the gadgets present.

Must find other methods to control program flow

```
prologue:
  ;-------------------------------------------------------
  %ifdef COMMENT
    we change the calling convention slightly here
    and use r10 as pointer to function to call
    function call will be replaced with call prologue
    ret will be replaced with jmp epilogue
    dr0 is abused to store canary page address
    +------------+ <- rsp/rbp upon leave
    |   canary   | <= canary
    |   rbp      | <= rbp
    |   r15      | <= r15
    |   r14      | <= r14
    |   r13      | <= r13
    |   r12      | <= r12
    |   r11      | <= r11
    +------------+ <- rsp upon entry
    |   rip      | <= pushed by caller
    +------------+
  %endif
  ;-------------------------------------------------------
  %ifdef CODE
    push r11
    push r12
    push r13
    push r14
    push r15
    push rbp
    mov rbp, dr0
    mov rbp, [rbp]
    push rbp
    mov rbp, rsp
    jmp r10
  %endif

epilogue:
  ;-------------------------------------------------------
  %ifdef COMMENT
    abuse that rdi, rsi, rdx are never preserved
    +------------+ <- rbp upon entry
    |   canary   | => rdi
    |   rbp      | => rbp
    |   r15      | => r15
    |   r14      | => r14
    |   r13      | => r13
    |   r12      | => r12
    |   r11      | => r11
    |   rip      | => rsi
    +------------+ <- rsp upon leave
  %endif
  ;-------------------------------------------------------
  %ifdef CODE
    mov rsp, rbp
    pop rdi
    pop rbp
    pop r15
    pop r14
    pop r13
    pop r12
    pop r11
    pop rsi
    mov rdx, dr0
    mov rdx, [rdx]
    xor rdi, rdx
    test rdi, rdi
    jnz __stack_chk_fail
    jmp rsi
  %endif
```

# Unicorn Code Execution

Guest Arch Machine Code

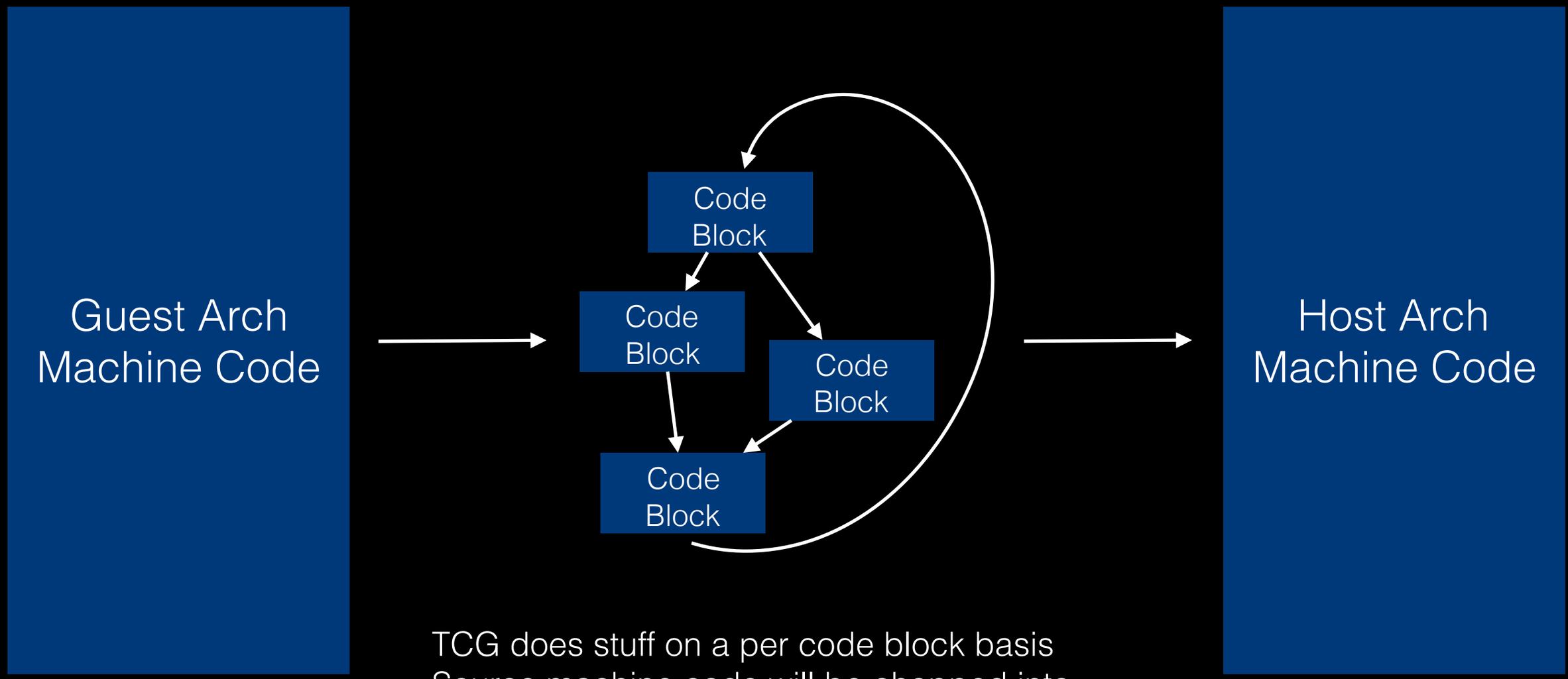
TCG

Host Arch Machine Code

Unicorn does arch translation through JITTing guest arch machine code to host arch machine code directly

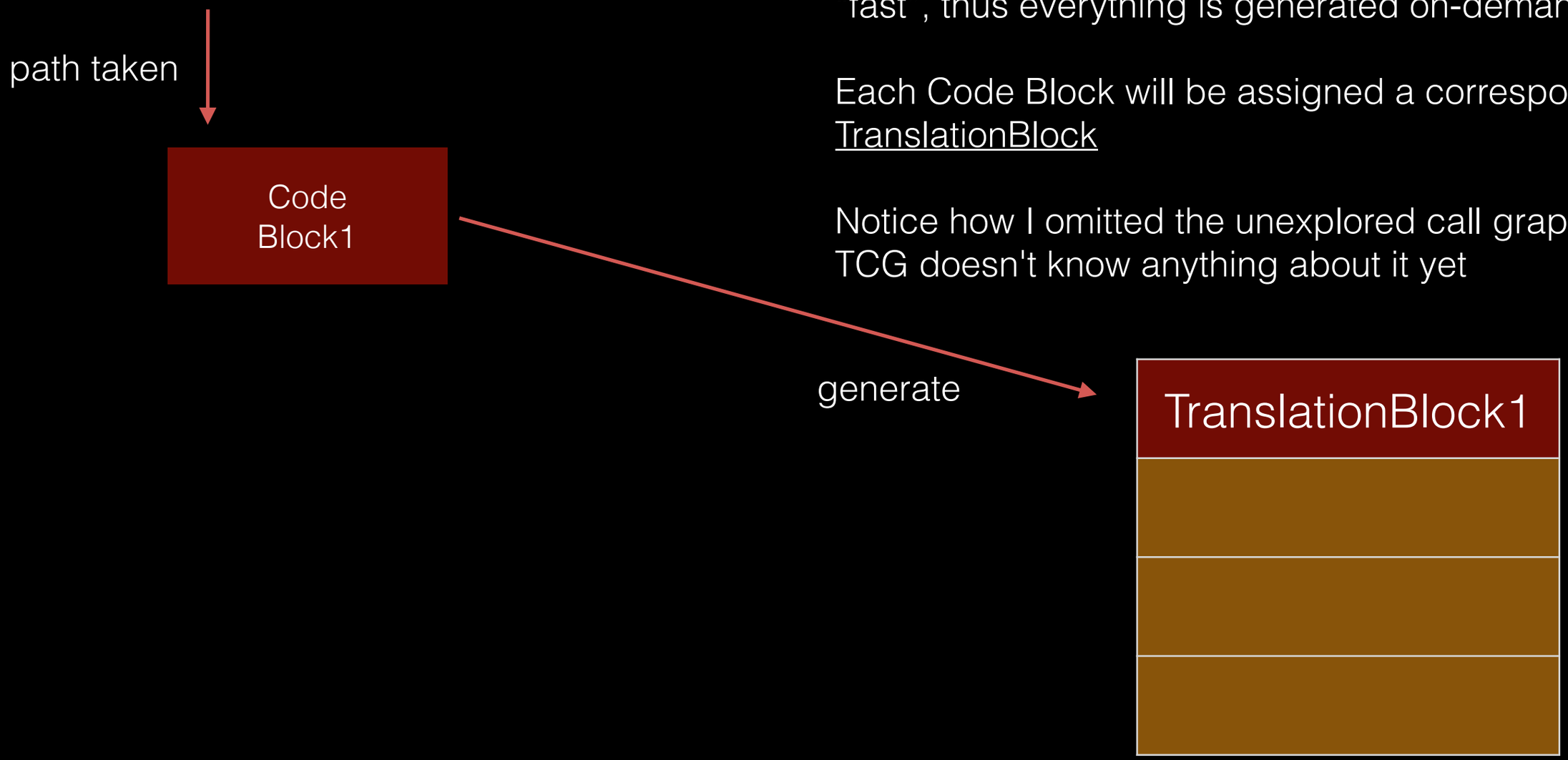
The JIT is done by Tiny Code Generator (TCG), which is a mechanism borrowed from qemu

# Unicorn Code Execution




TCG does stuff on a per code block basis
Source machine code will be chopped into
basic blocks with purely linear code
TCG then tackles each basic block separately

# Unicorn Code Execution

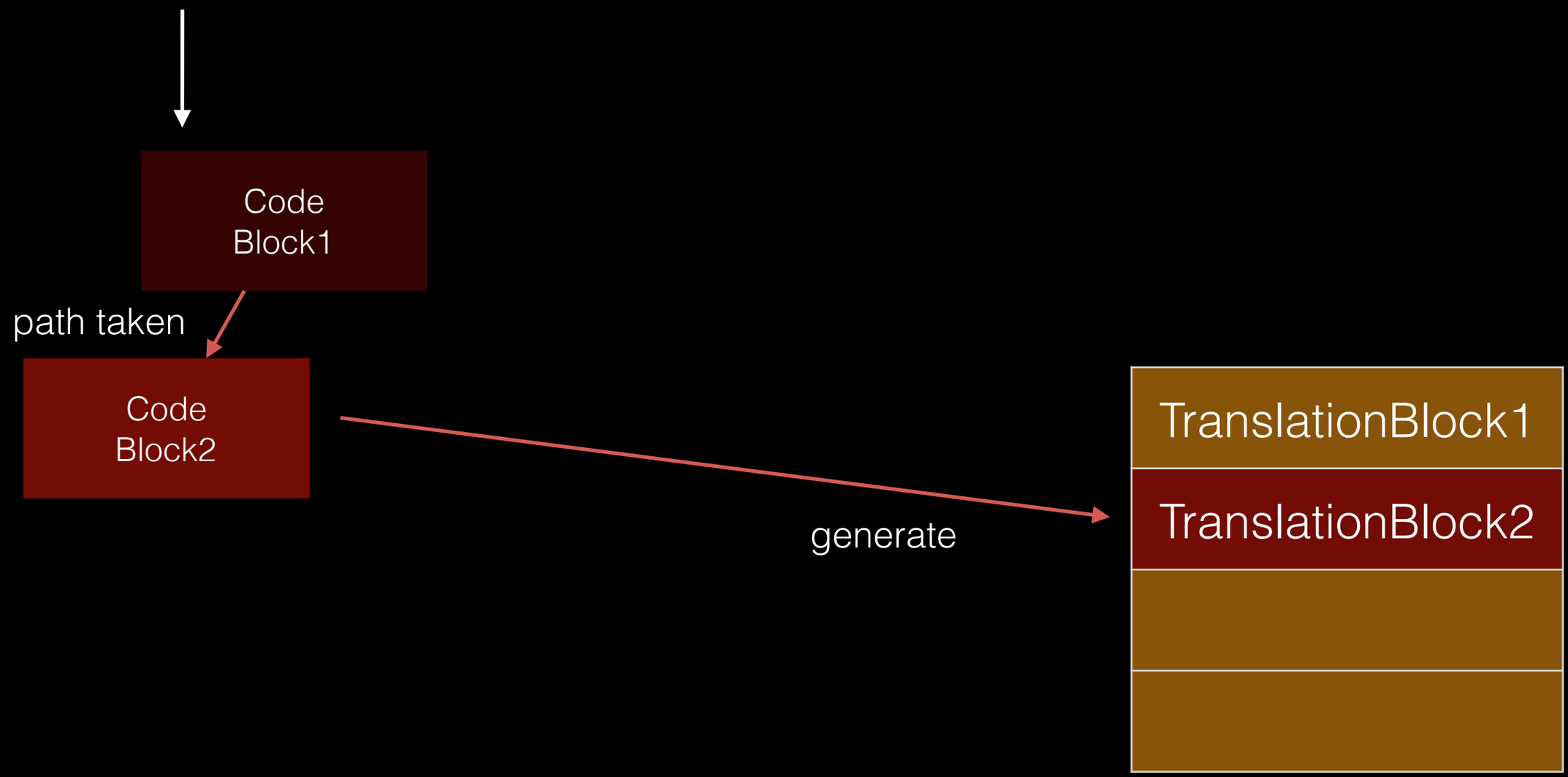



One of the design motivations for TCG is to make it “fast”, thus everything is generated on-demand

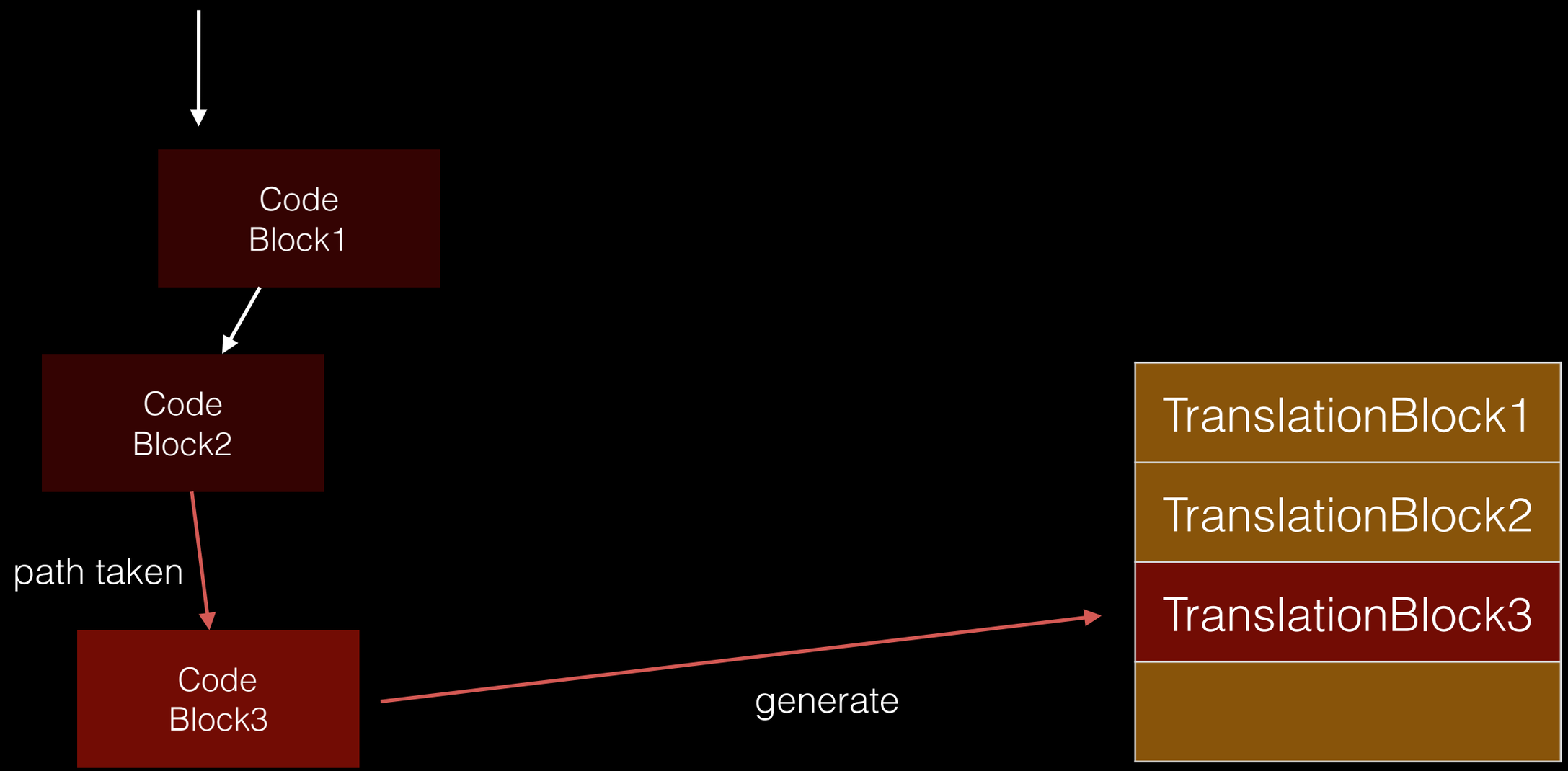
Each Code Block will be assigned a corresponding TranslationBlock

Notice how I omitted the unexplored call graph, since TCG doesn't know anything about it yet

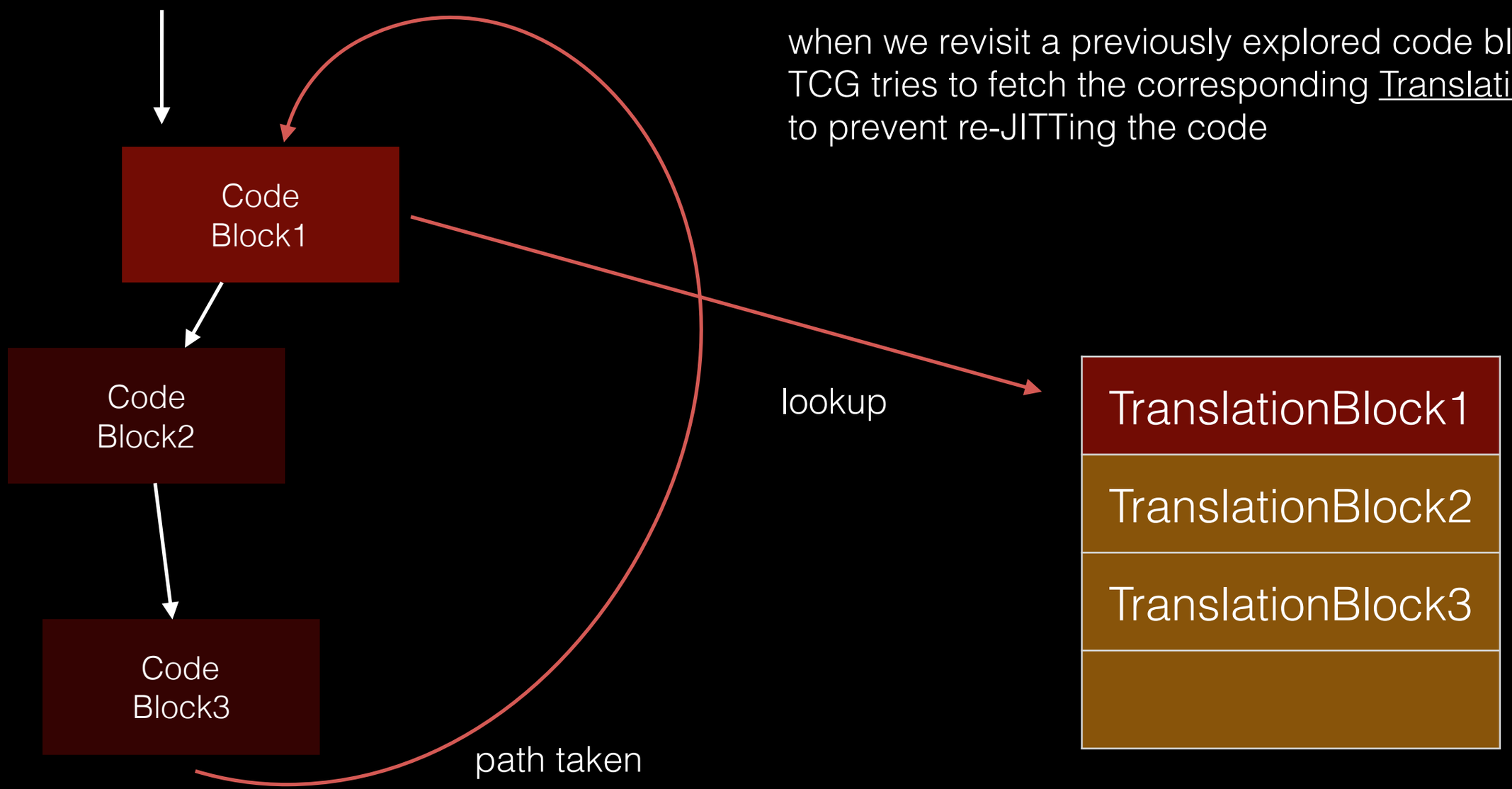
# Unicorn Code Execution



# Unicorn Code Execution

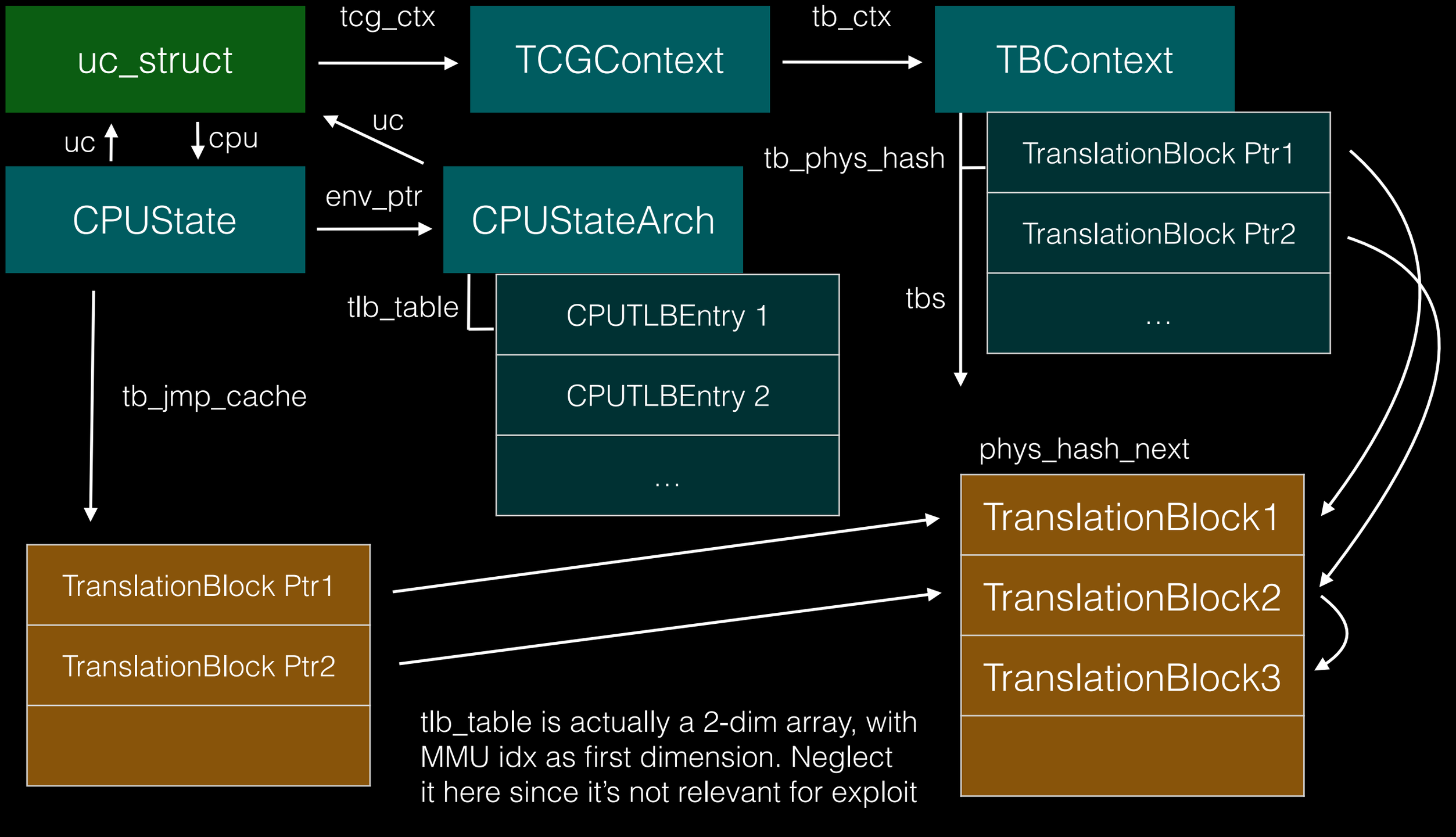


# Unicorn Code Execution




when we revisit a previously explored code block
TCG tries to fetch the corresponding TranslationBlock
to prevent re-JITTing the code

# Unicorn Code Execution



# Code Execution

whenever a new tb is required for code execution, unicorn calls **tb_find_fast** and attempts to lookup the block from cpu->tb_jmp_cache

if no suitable block is found, take the slow path of **tb_find_slow** instead

```
1  static TranslationBlock *tb_find_fast(CPUArchState *env)
2  {
3      CPUState *cpu = ENV_GET_CPU(env);
4      TranslationBlock *tb;
5      target_ulong cs_base, pc;
6      int flags;
7
8      /* we record a subset of the CPU state. It will
9         always be the same before a given translated block
10        is executed. */
11     cpu_get_tb_cpu_state(env, &pc, &cs_base, &flags);
12     tb = cpu->tb_jmp_cache[tb_jmp_cache_hash_func(pc)];
13     if (unlikely(!tb || tb->pc != pc || tb->cs_base != cs_base ||
14                  tb->flags != flags)) {
15         tb = tb_find_slow(env, pc, cs_base, flags);
16     }
17     return tb;
18 }
```

# Code Execution

**tb_find_slow** then attempts to lookup the corresponding tb from uc->tcg_ctx->tb_ctx.tb_phys_hash

note that before looking up from uc->tcg_ctx->tb_ctx.tb_phys_hash, an address translation by **get_page_addr_code** must be performed

if lookup still fails, the target tb is not present, and must be generated by **tb_gen_code**

```
1  static TranslationBlock *tb_find_slow(
2          CPUArchState *env, target_ulong pc,
3          target_ulong cs_base, uint64_t flags)
4  {
5      CPUState *cpu = ENV_GET_CPU(env);
6      TCGContext *tcg_ctx = env->uc->tcg_ctx;
7      TranslationBlock *tb, **ptb1;
8      unsigned int h;
9      tb_page_addr_t phys_pc, phys_page1;
10     target_ulong virt_page2;
11
12     tcg_ctx->tb_ctx.tb_invalidated_flag = 0;
13
14     /* find translated block using physical mappings */
15     phys_pc = get_page_addr_code(env, pc);
16     if (phys_pc == -1) { // invalid code?
17         return NULL;
18     }
19     phys_page1 = phys_pc & TARGET_PAGE_MASK;
20     h = tb_phys_hash_func(phys_pc);
21     ptb1 = &tcg_ctx->tb_ctx.tb_phys_hash[h];
22     for(;;) {
23         tb = *ptb1;
24         if (!tb)
25             goto not_found;
26         if (tb->pc == pc &&
27                 tb->page_addr[0] == phys_page1 &&
28                 tb->cs_base == cs_base &&
29                 tb->flags == flags) {
30             /* check next page if needed */
31             if (tb->page_addr[1] != -1) {
32                 tb_page_addr_t phys_page2;
33
34                 virt_page2 = (pc & TARGET_PAGE_MASK) +
35                     TARGET_PAGE_SIZE;
36                 phys_page2 = get_page_addr_code(env, virt_page2);
37                 if (tb->page_addr[1] == phys_page2)
38                     goto found;
39             } else {
40                 goto found;
41             }
42         }
43         ptb1 = &tb->phys_hash_next;
44     }
45 not_found:
46     /* if no translated code available, then translate it now */
47     tb = tb_gen_code(cpu, pc, cs_base, (int)flags, 0);
48     if (tb == NULL) {
49         return NULL;
50     }
51
52 found:
53     /* Move the last found TB to the head of the list */
54     if (likely(*ptb1)) {
55         *ptb1 = tb->phys_hash_next;
56         tb->phys_hash_next = tcg_ctx->tb_ctx.tb_phys_hash[h];
57         tcg_ctx->tb_ctx.tb_phys_hash[h] = tb;
58     }
59     /* we add the TB in the virtual pc hash table */
60     cpu->tb_jmp_cache[tb_jmp_cache_hash_func(pc)] = tb;
61     return tb;
62 }
63
```

# Code Execution

**tb_gen_code** then calls **gen_intermediate_code**, which calls **disas_insn** to generate TCG IR from guest assembly

details of this procedure is beyond this slide's topic, we will only look at the "feature" within **disas_insn** that powers our exploit

take nop (0x90) as example, we can see the insn is first fetched by **cpu_ldub_code**, then goes through several switch/case until it is handled

```
1  static target_ulong disas_insn(
2          CPUX86State *env, DisasContext *s,
3          target_ulong pc_start)
4  {
5      ...
6   next_byte:
7      b = cpu_ldub_code(env, s->pc);
8      s->pc++;
9      /* Collect prefixes.  */
10     switch (b) {
11     case 0xf3:
12         prefixes |= PREFIX_REPZ;
13         goto next_byte;
14     case 0xf2:
15         prefixes |= PREFIX_REPNZ;
16         goto next_byte;
17     case 0xf0:
18         prefixes |= PREFIX_LOCK;
19         goto next_byte;
20     ...
21     }
22     ...
23  reswitch:
24     switch(b) {
25         ...
26         case 0x90: /* nop */
27         /* XXX: correct lock test for all insn */
28         if (prefixes & PREFIX_LOCK) {
29             goto illegal_op;
30         }
31         /* If REX_B is set, then this is xchg eax, r8d, not a nop.  */
32         if (REX_B(s)) {
33             goto do_xchg_reg_eax;
34         }
35         if (prefixes & PREFIX_REPZ) {
36             gen_update_cc_op(s);
37             gen_jmp_im(s, pc_start - s->cs_base);
38             gen_helper_pause(tcg_ctx, cpu_env,
39                              tcg_const_i32(tcg_ctx, s->pc - pc_start));
40             s->is_jmp = DISAS_TB_JUMP;
41         }
42         break;
43         ...
44     }
45     ...
46 }
47
```

# Code Execution

one can immediately see that after **disas_insn** fetches the assembly and translates to IR, information about where this piece of assembly resides is abstracted away

meaning that **disas_insn** is the point of no return in terms of checking NX

as shown in right code, for instruction nop, if NX is to be checked in **disas_insn**, it can only be enforced in **cpu_ldub_code**

```
1  static target_ulong disas_insn(
2          CPUX86State *env, DisasContext *s,
3          target_ulong pc_start)
4  {
5      ...
6   next_byte:
7      b = cpu_ldub_code(env, s->pc);
8      s->pc++;
9      /* Collect prefixes.  */
10     switch (b) {
11     case 0xf3:
12         prefixes |= PREFIX_REPZ;
13         goto next_byte;
14     case 0xf2:
15         prefixes |= PREFIX_REPNZ;
16         goto next_byte;
17     case 0xf0:
18         prefixes |= PREFIX_LOCK;
19         goto next_byte;
20     ...
21     }
22     ...
23  reswitch:
24     switch(b) {
25         ...
26         case 0x90: /* nop */
27         /* XXX: correct lock test for all insn */
28         if (prefixes & PREFIX_LOCK) {
29             goto illegal_op;
30         }
31         /* If REX_B is set, then this is xchg eax, r8d, not a nop.  */
32         if (REX_B(s)) {
33             goto do_xchg_reg_eax;
34         }
35         if (prefixes & PREFIX_REPZ) {
36             gen_update_cc_op(s);
37             gen_jmp_im(s, pc_start - s->cs_base);
38             gen_helper_pause(tcg_ctx, cpu_env,
39                              tcg_const_i32(tcg_ctx, s->pc - pc_start));
40             s->is_jmp = DISAS_TB_JUMP;
41         }
42         break;
43         ...
44     }
45     ...
46 }
47
```

# Code Execution

**cpu_ldub_code** expands to the function shown on right

There are two major path, branch taken based on whether the target memory is cached with ADDR_READ attribute in env->tlb_table, and whether the read memory is aligned

```
1  glue(glue(cpu_ld, USUFFIX), MEMSUFFIX)(
2          CPUArchState *env, target_ulong ptr)
3  {
4      int page_index;
5      RES_TYPE res;
6      target_ulong addr;
7      int mmu_idx;
8
9      addr = ptr;
10     page_index = (addr >> TARGET_PAGE_BITS) & (CPU_TLB_SIZE - 1);
11     mmu_idx = CPU_MMU_INDEX;
12     if (unlikely(env->tlb_table[mmu_idx][page_index].ADDR_READ !=
13                  (addr & (TARGET_PAGE_MASK | (DATA_SIZE - 1))))) {
14         res = glue(glue(helper_ld, SUFFIX), MMUSUFFIX)(env, addr, mmu_idx);
15     } else {
16         uintptr_t hostaddr =
17             (uintptr_t)(addr + env->tlb_table[mmu_idx][page_index].addend);
18         res = glue(glue(ld, USUFFIX), _raw)(hostaddr);
19     }
20     return res;
21 }
```

# Code Execution

if target mem is not cached or not aligned, the first branch is taken, which ends up in **helper_le_ld_name**

This function checks against execution privs

```
1  WORD_TYPE helper_le_ld_name(CPUArchState *env, target_ulong addr, int mmu_idx,
2                              uintptr_t retaddr)
3  {
4      int index = (addr >> TARGET_PAGE_BITS) & (CPU_TLB_SIZE - 1);
5      target_ulong tlb_addr = env->tlb_table[mmu_idx][index].ADDR_READ;
6      uintptr_t haddr;
7      DATA_TYPE res;
8      int error_code;
9      struct hook *hook;
10     bool handled;
11     HOOK_FOREACH_VAR_DECLARE;
12
13     struct uc_struct *uc = env->uc;
14     MemoryRegion *mr = memory_mapping(uc, addr);
15
16     ...
17
18 #if defined(SOFTMMU_CODE_ACCESS)
19     // Unicorn: callback on fetch from NX
20     if (mr != NULL && !(mr->perms & UC_PROT_EXEC)) {  // non-executable
21         handled = false;
22         HOOK_FOREACH(uc, hook, UC_HOOK_MEM_FETCH_PROT) {
23             if (hook->to_delete)
24                 continue;
25             if (!HOOK_BOUND_CHECK(hook, addr))
26                 continue;
27             if ((handled = ((uc_cb_eventmem_t)hook->callback)
28                     (uc, UC_MEM_FETCH_PROT, addr,
29                      DATA_SIZE - uc->size_recur_mem, 0, hook->user_data)))
30                 break;
31         }
32
33         if (handled) {
34             env->invalid_error = UC_ERR_OK;
35         } else {
36             env->invalid_addr = addr;
37             env->invalid_error = UC_ERR_FETCH_PROT;
38             cpu_exit(uc->current_cpu);
39             return 0;
40         }
41     }
42 #endif
43     ...
44 }
```

# Code Execution

if target mem is not cached and correctly aligned, the second branch is taken, which ends up in the **ldub_p** series of function

```
1 static inline int ldub_p(const void *ptr)
2 {
3     return *(uint8_t *)ptr;
4 }
5
6 static inline int ldsb_p(const void *ptr)
7 {
8     return *(int8_t *)ptr;
9 }
10
11 ...
```

These functions perform no check against execution privilege

# Code Execution

- Combining the mentioned “features”, if we manage to get stack into tlb_table, and carefully align each instruction, it is possible to run shellcode on stack

- Getting stack into tlb_table is quite simple, any read/write operation on stack will do

- Let’s see how to leverage the return address overwrite primitive and do this

# Shellcode on Stack

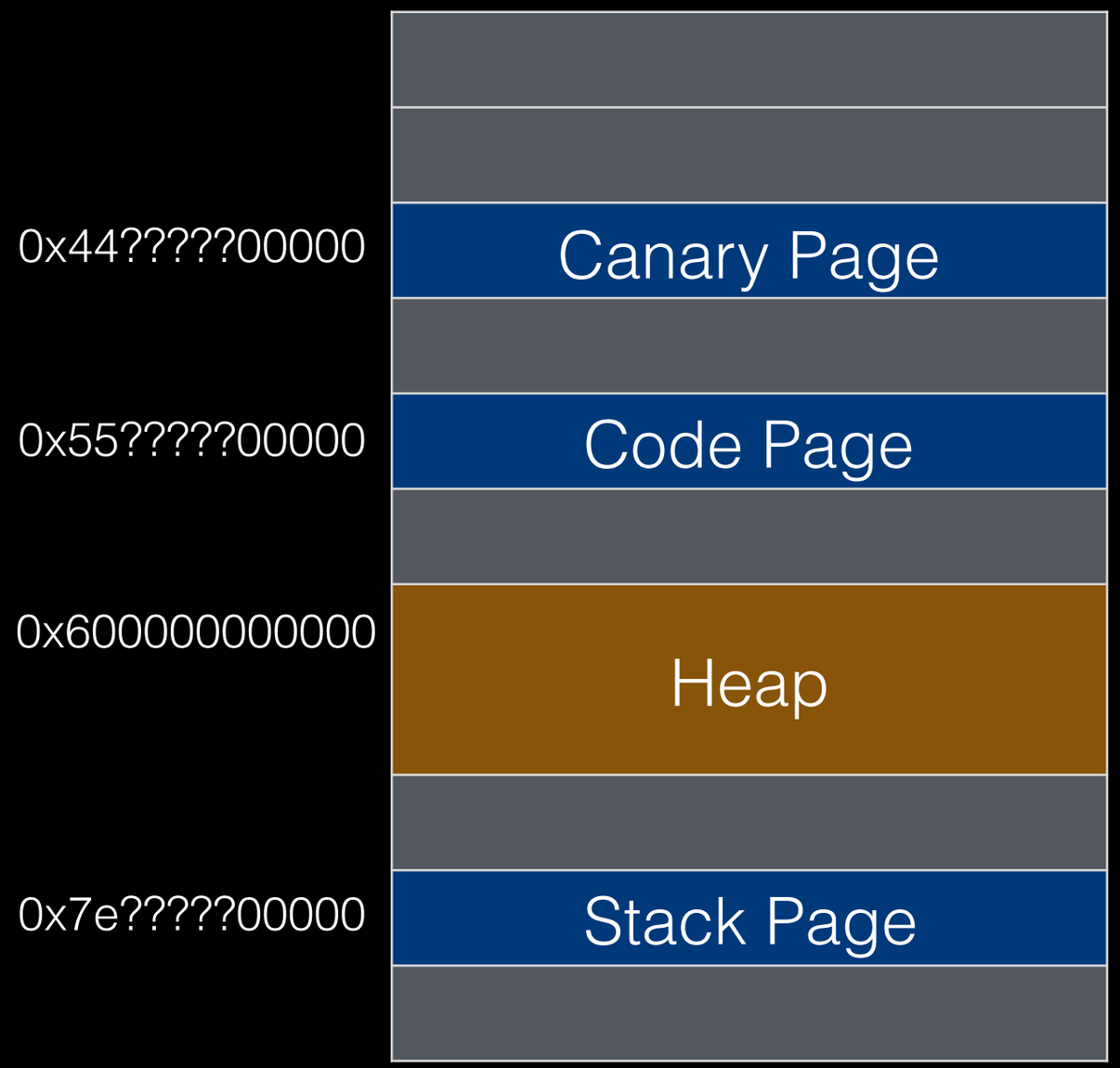
Notice how the server aligns all memory region by 20 bits

This alignment is chosen in correspondence to the tlb_table size

tlb_table indices are calculated by

```
1 (addr >> TARGET_PAGE_BITS) & (CPU_TLB_SIZE - 1);
```

meaning the 12~20th bits are used as index



# Shellcode on Stack

This indicates all pages will be stored in the same tlb_table entry

And access of any of the other pages will flush stack out of the cache table

By default, the canary page is last accessed before returning

Thus we have to perform an additional stack reference to fix tlb_table before running shellcode on stack

```
epilogue:
  ;-------------------------------------------------
  %ifdef COMMENT
    abuse that rdi, rsi, rdx are never preserved
    +------------+ <- rbp upon entry
    |   canary   | => rdi
    |   rbp      | => rbp
    |   r15      | => r15
    |   r14      | => r14
    |   r13      | => r13
    |   r12      | => r12
    |   r11      | => r11
    |   rip      | => rsi
    +------------+ <- rsp upon leave
  %endif
  ;-------------------------------------------------
  %ifdef CODE
    mov rsp, rbp
    pop rdi
    pop rbp
    pop r15
    pop r14
    pop r13
    pop r12
    pop r11
    pop rsi
    mov rdx, dr0
    mov rdx, [rdx]   //access canary page
    xor rdi, rdx
    test rdi, rdi
    jnz __stack_chk_fail
    jmp rsi
  %endif
```

# Shellcode on Stack

Since we don’t have any ret in encounter.S, it is natural to look for controllable jmp

prologue has one that seems promising

by returning to push rbp, we can insert stack into tlb_table, them perform a jmp

The last step is to try to control r10

```
prologue:
  ;-----------------------------------------------------
  %ifdef COMMENT
    we change the calling convention slightly here
    and use r10 as pointer to function to call
    function call will be replaced with call prologue
    ret will be replaced with jmp epilogue
    +------------+ <- rsp/rbp upon leave
    |   canary   | <= canary
    |   rbp      | <= rbp
    |   r15      | <= r15
    |   r14      | <= r14
    |   r13      | <= r13
    |   r12      | <= r12
    |   r11      | <= r11
    +------------+ <- rsp upon entry
    |   rip      | <= pushed by caller
    +------------+
  %endif
  ;-----------------------------------------------------
  %ifdef CODE
    push r11
    push r12
    push r13
    push r14
    push r15
    push rbp
    mov rbp, dr0
    mov rbp, [rbp]
    push rbp       // return here to reference stack + jmp
    mov rbp, rsp
    jmp r10
  %endif
```

# Shellcode on Stack

grep r10 in encounter.S and we can immediately find this snippet, where r10 is controllable if we manage to control r11

```
_isUnicornUnderAttack:
    ...
    mov rax, qword [r11+confrontState.unicornLoc]
    mov r10, rax
    add r10, UNICORN_HALF_SIZE
    sub rax, UNICORN_HALF_SIZE
    mov rdx, qword [r11+confrontState.adventurerState]
    mov r9, rdx
    and rdx, 1
    ...
```

r11 is trivially controllable since it is one of the registers popped in function epilogue

combining all of those, and we are finally able to run restricted shellcode on stack

# Postlude

Escape Unicorn

# Target

- Now we have restricted shell code execution on stack, time to look at the final stage
- There is no handler function that prints the postlude flag, indicating we have to pwn the emulator itself
- Furthermore, there are no other bugs* in the provided wrapper around unicorn engine, thus we have to find a 0-day in unicorn engine

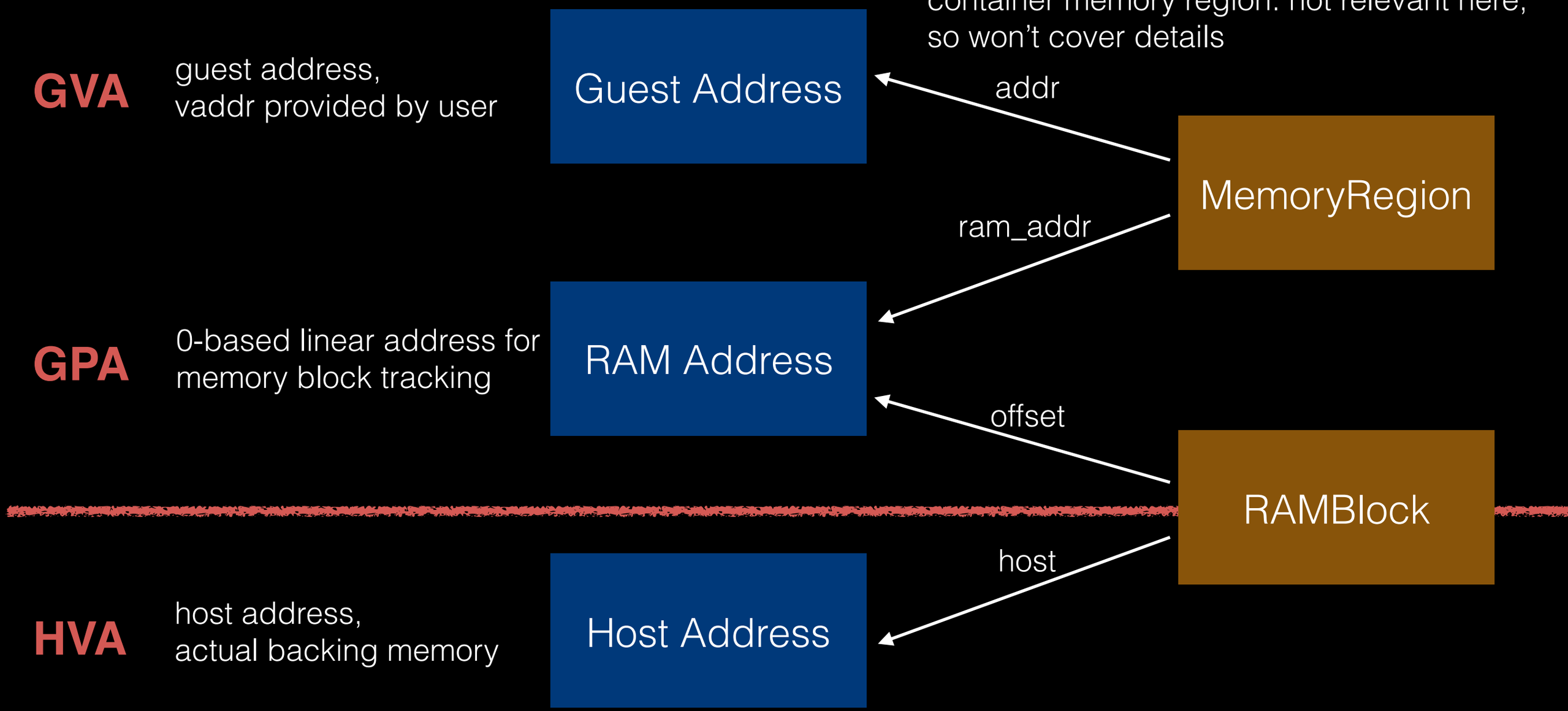
* there is actually an unintended bug found by hexrabbit@10sec that makes interlude far easier than intended, but the bug does not help in pwning postlude

# Unicorn Mem Management

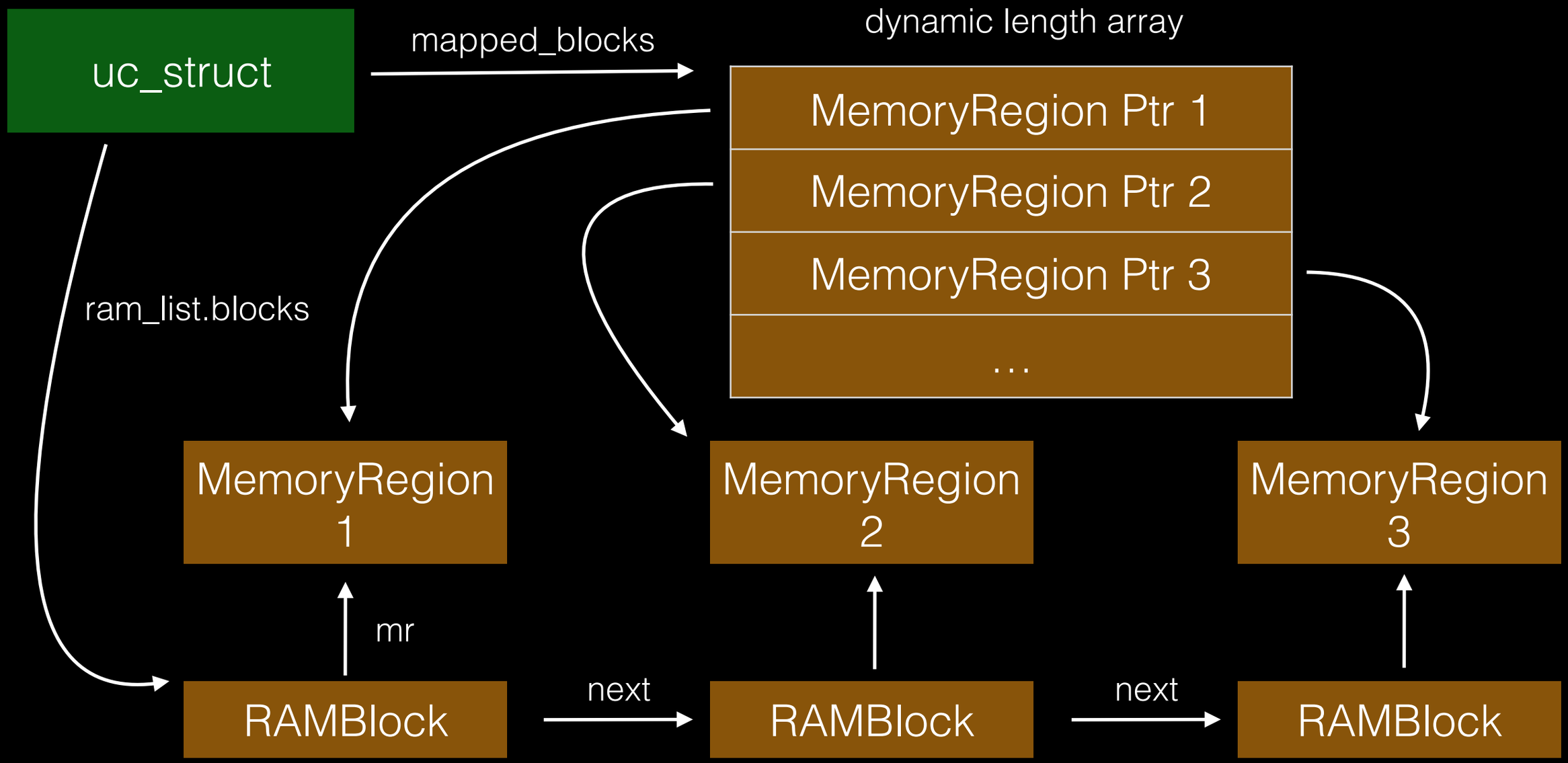
**GVA** guest address,
vaddr provided by user

**GPA** 0-based linear address for
memory block tracking

**HVA** host address,
actual backing memory

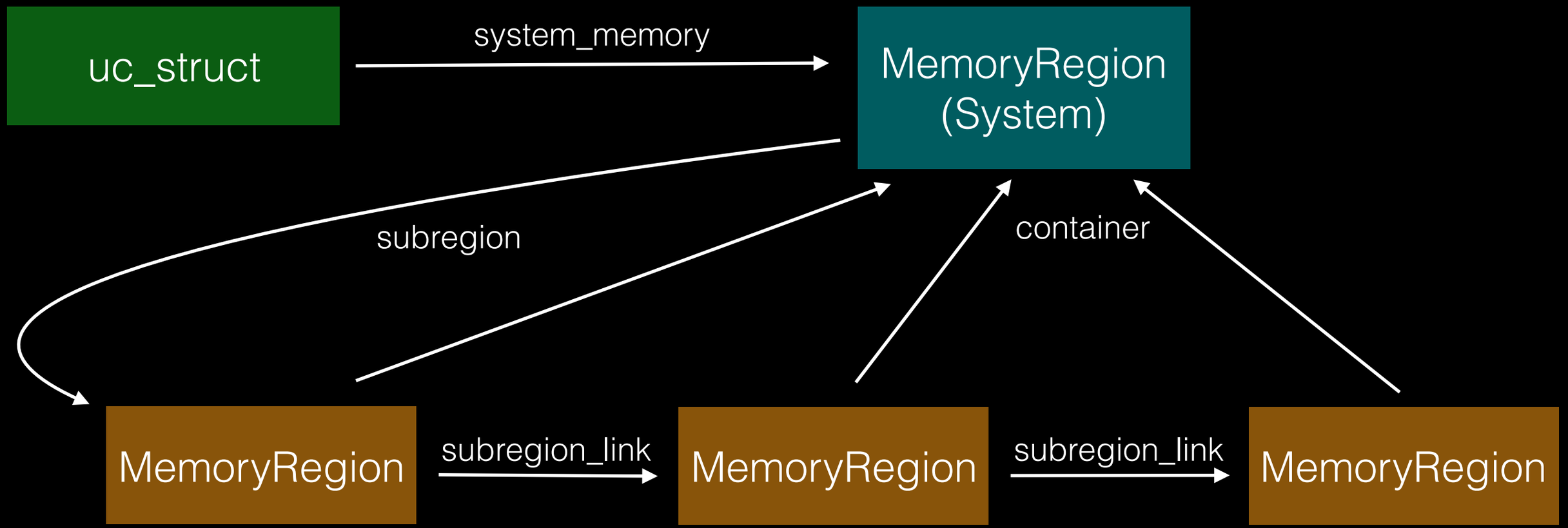


# Unicorn Mem Management




RAMBlocks are sorted by size (large -> small)

# Unicorn Mem Management




subregions are sorted by priority (high -> low)
this is required for shared men between device, won't cover here

memory region is constructed in a hierarchical tree-like structure
this allows clear division between memory used by different objects/devices

# Unicorn Mem Management

uc_struct

address_spaces/as

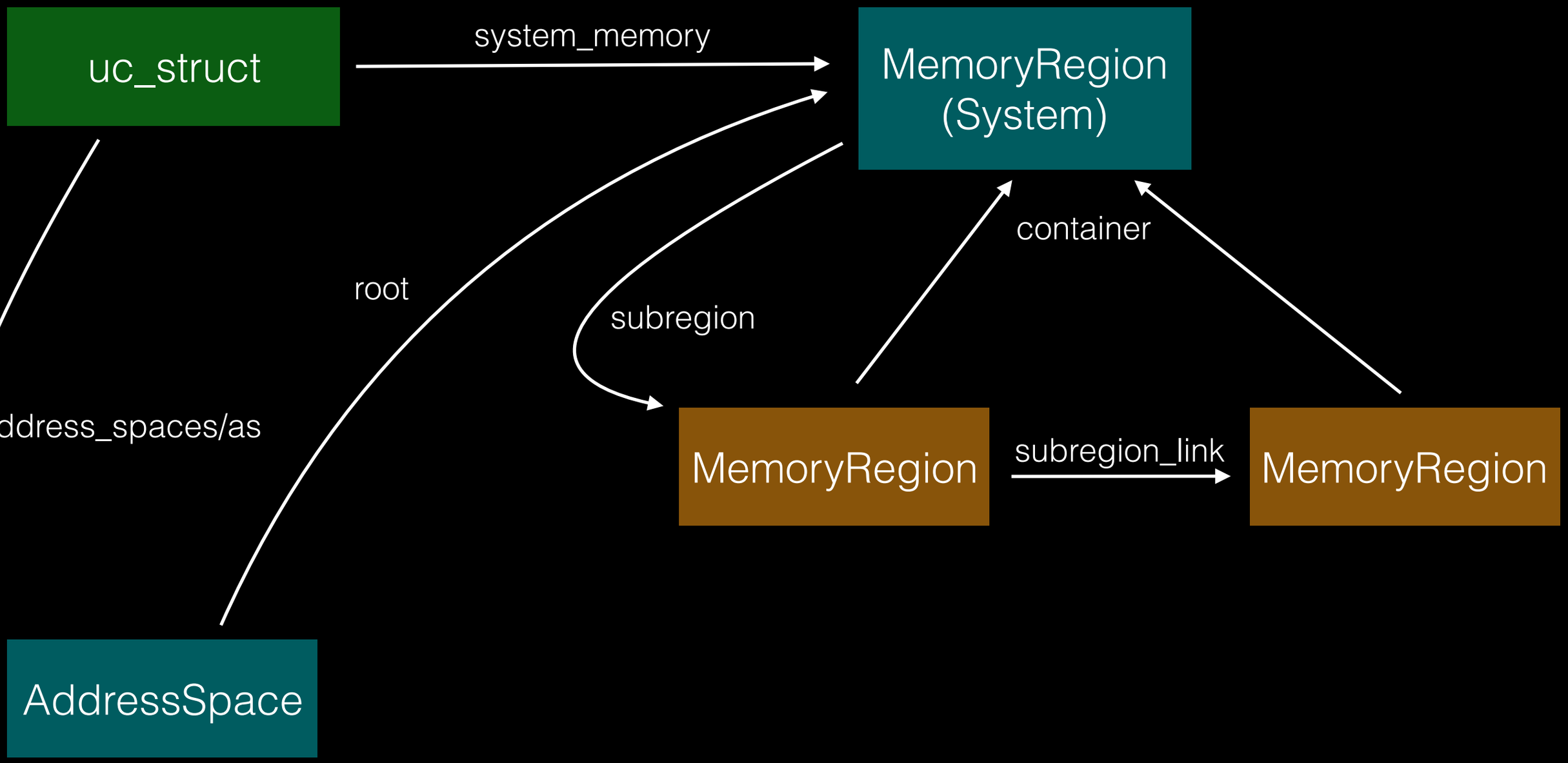
in practice, unicorn usually only have one address space

AddressSpace

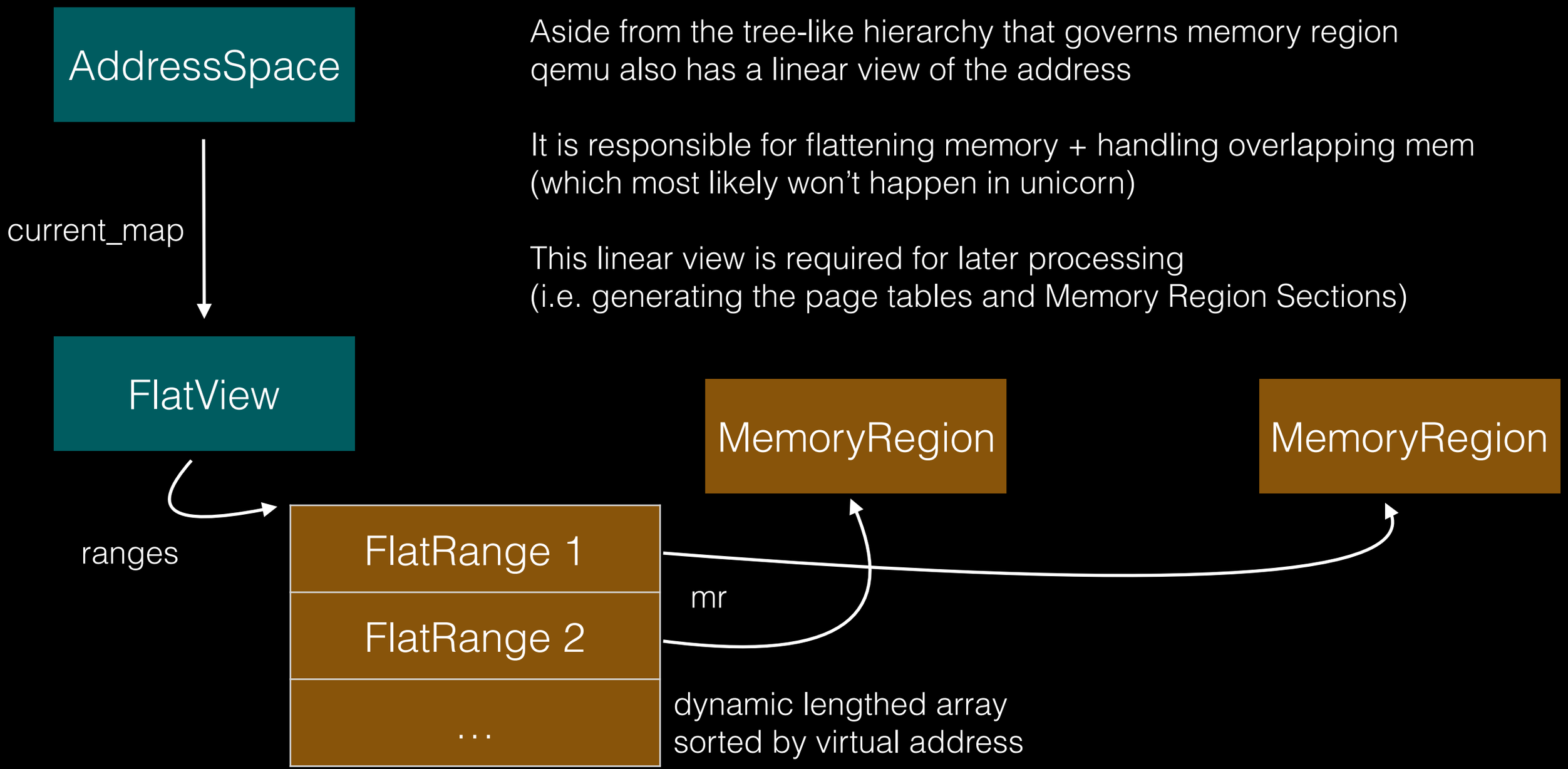
address_spaces_link

AddressSpace

# Unicorn Mem Management



# Unicorn Mem Management

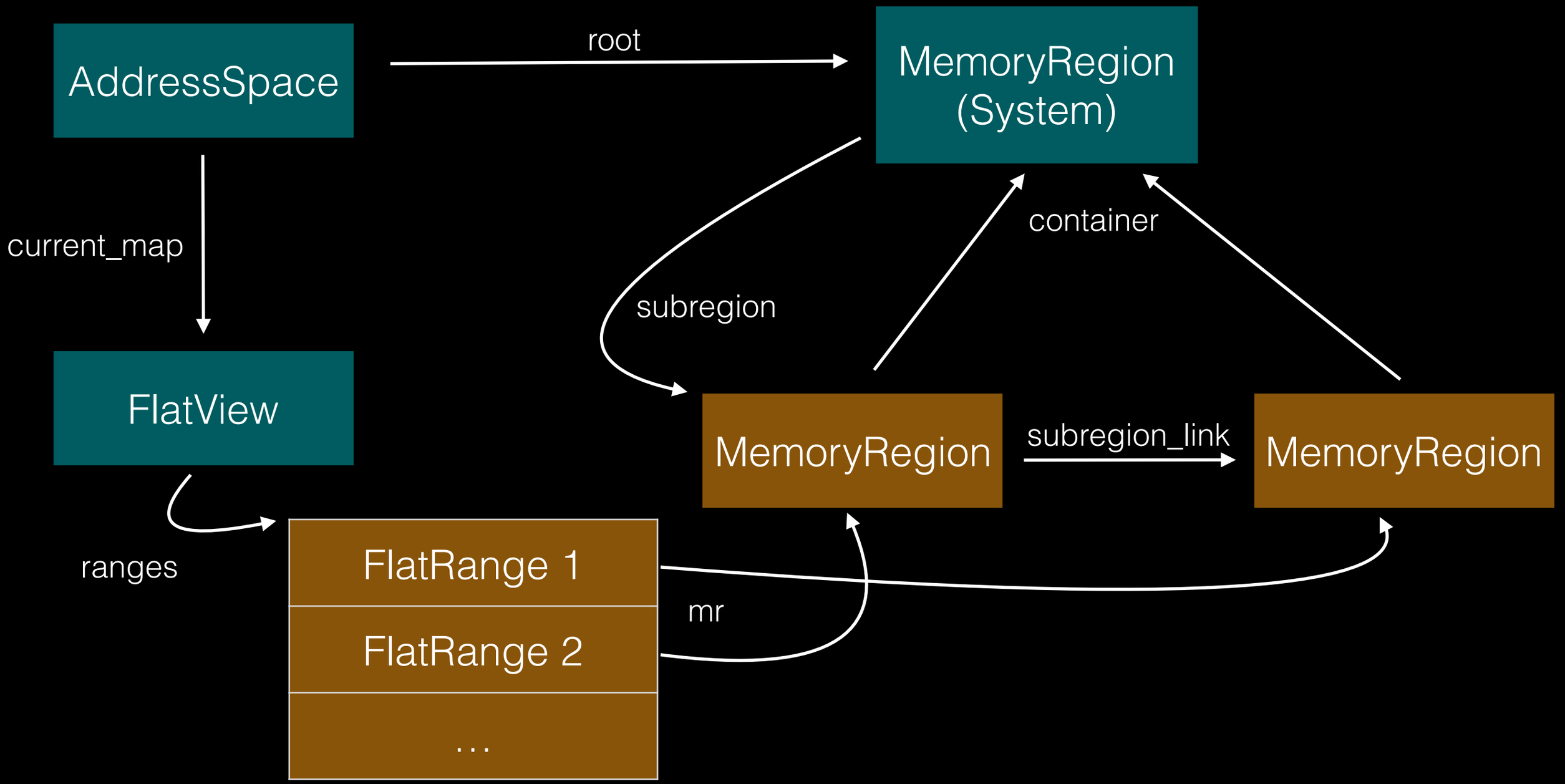



Aside from the tree-like hierarchy that governs memory region qemu also has a linear view of the address

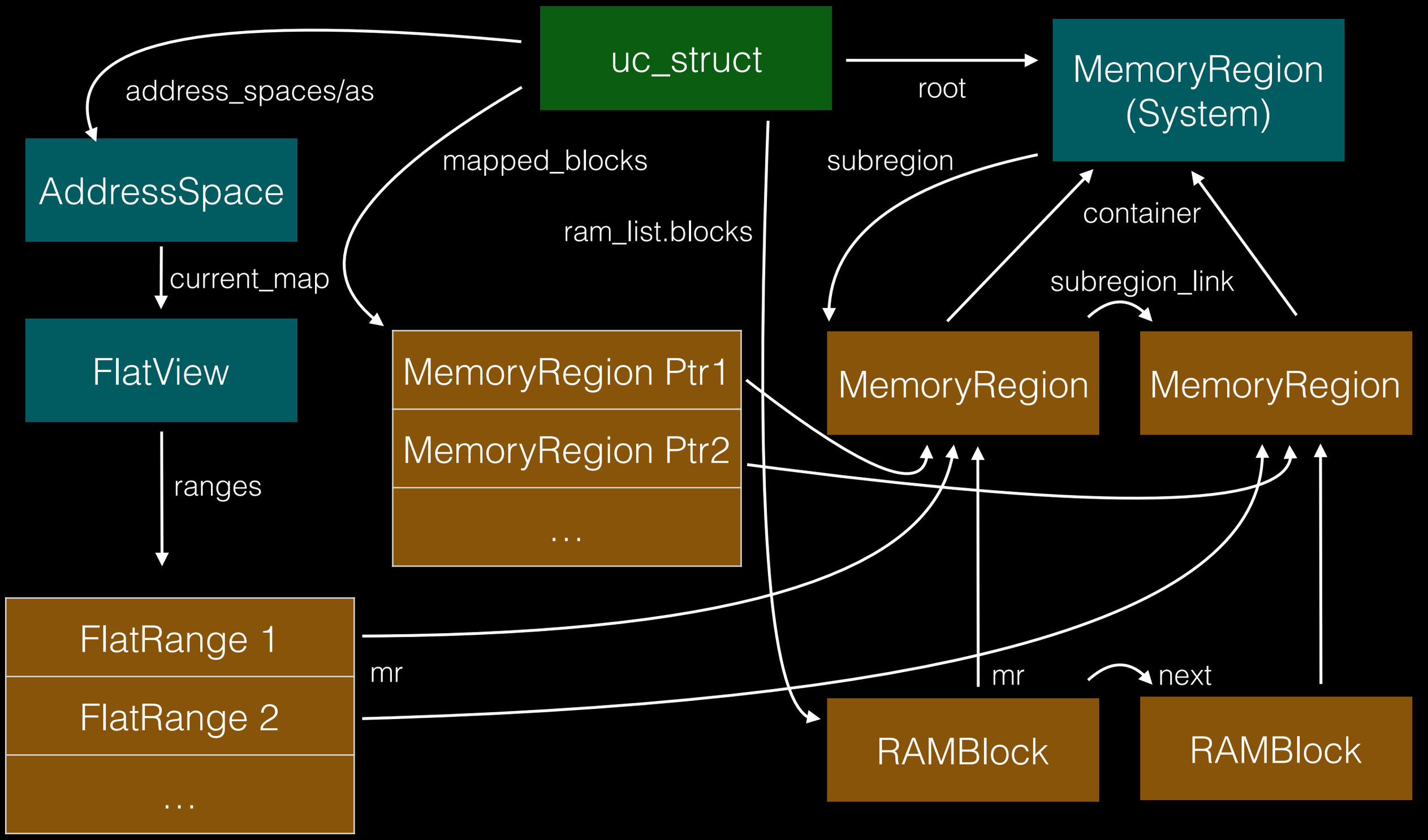
It is responsible for flattening memory + handling overlapping mem (which most likely won't happen in unicorn)

This linear view is required for later processing
(i.e. generating the page tables and Memory Region Sections)

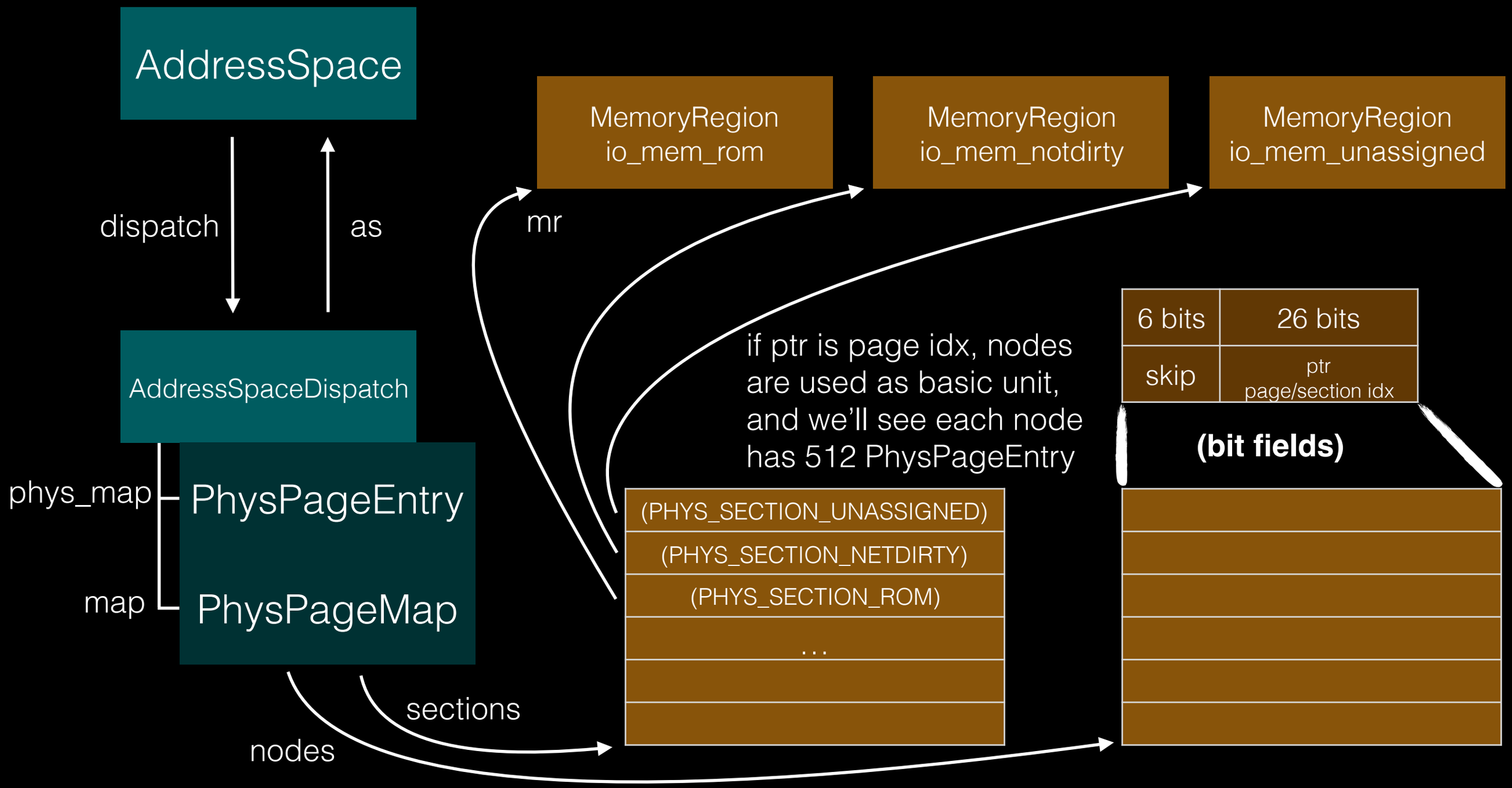
# Unicorn Mem Management



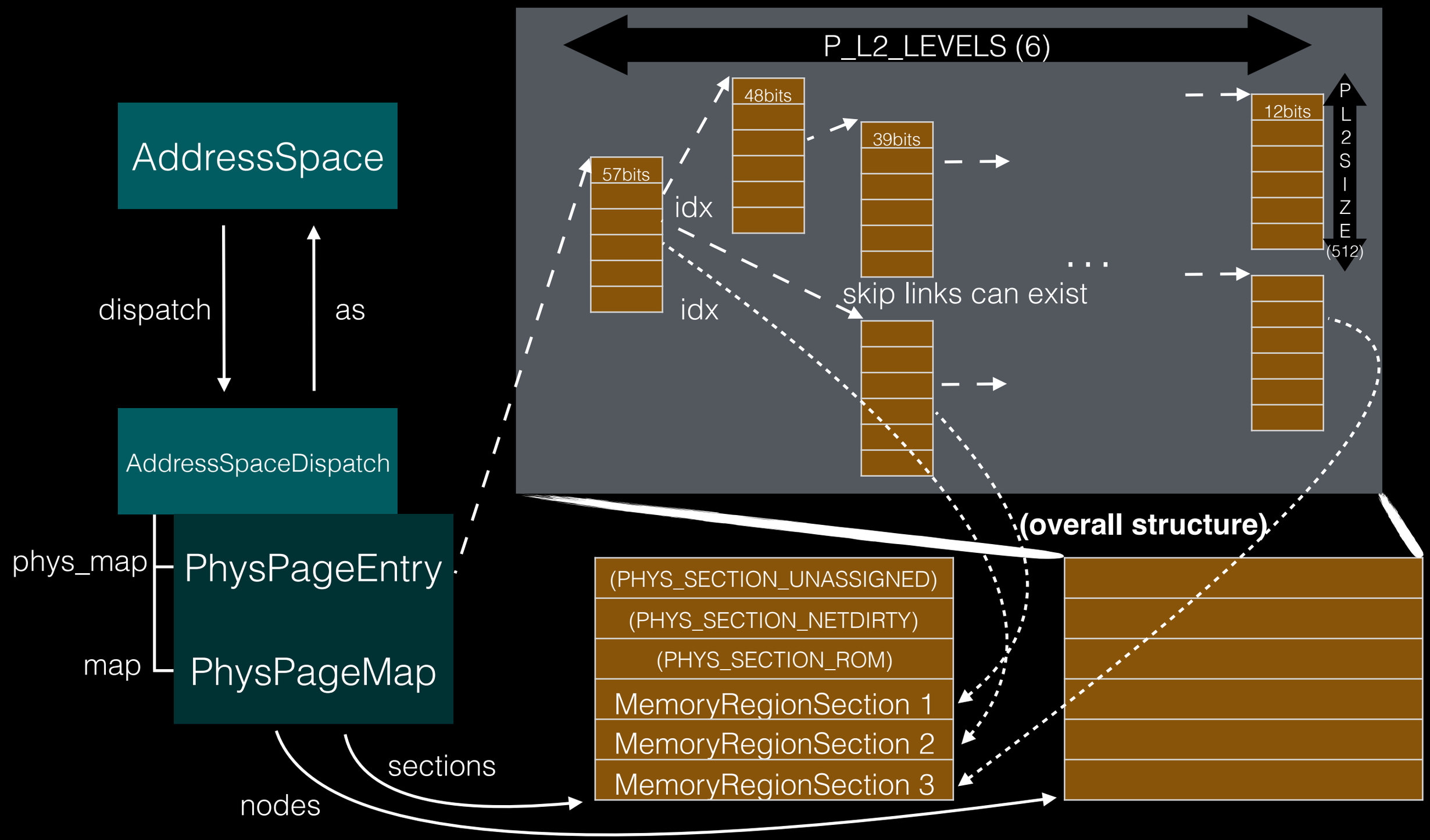
# Summary1


uc_struct
address_spaces/as
root
MemoryRegion (System)
AddressSpace
mapped_blocks
ram_list.blocks
subregion
container
subregion_link
current_map
FlatView
MemoryRegion Ptr1
MemoryRegion Ptr2
…
MemoryRegion
MemoryRegion
ranges
FlatRange 1
FlatRange 2
…
mr
mr
next
RAMBlock
RAMBlock

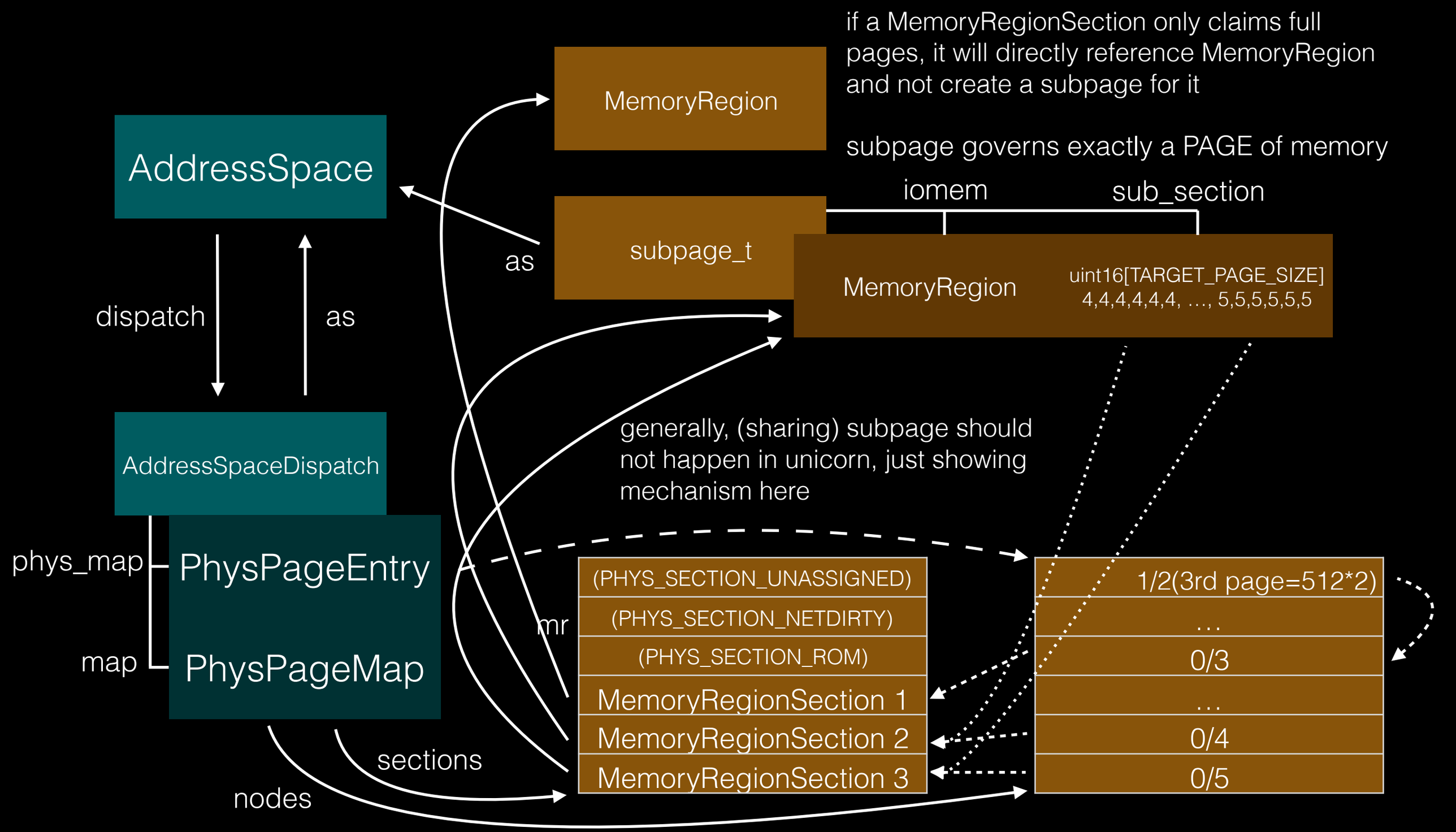
# Unicorn Mem Management



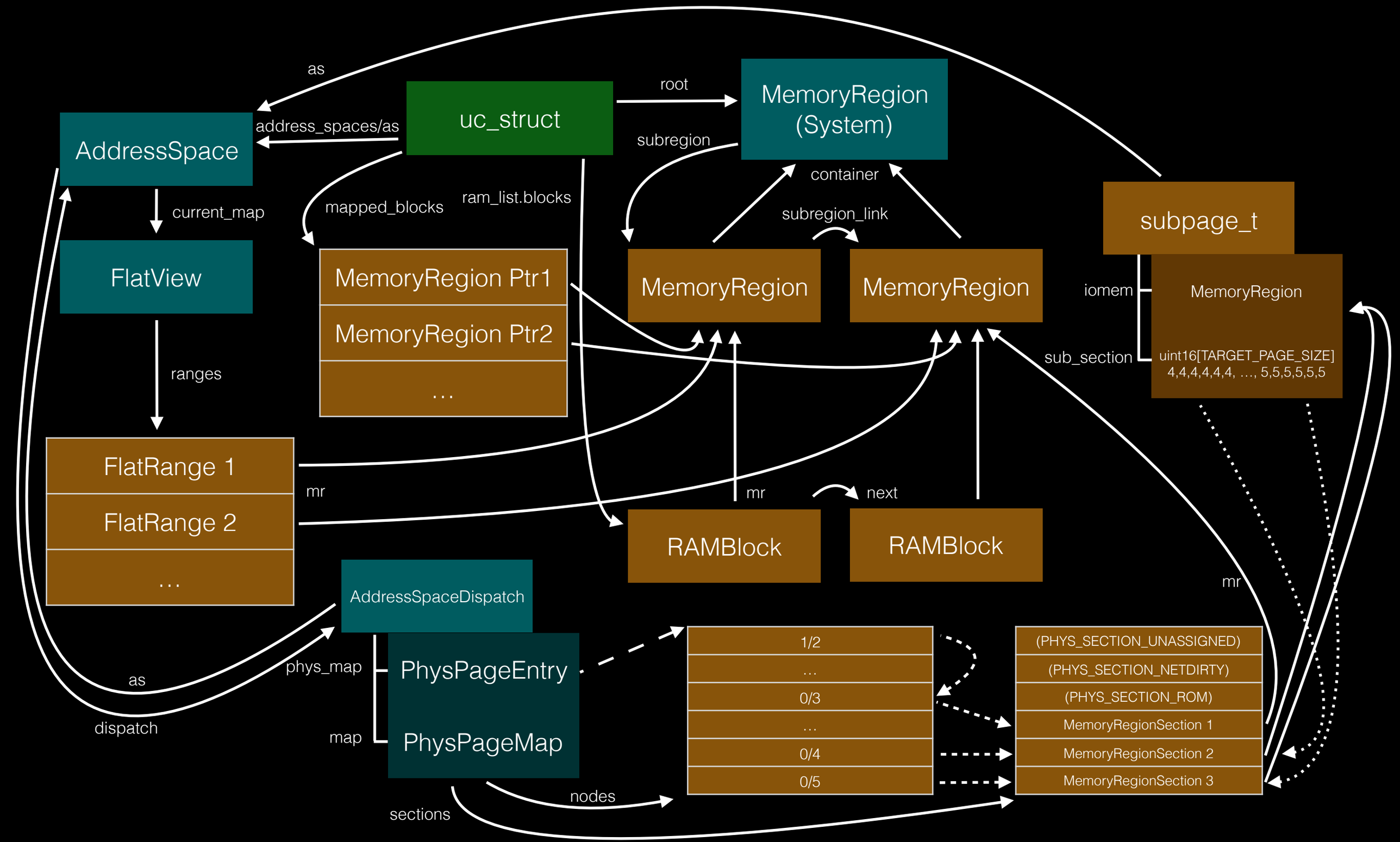
# Unicorn Mem Management



# Unicorn Mem Management



# Full Picture



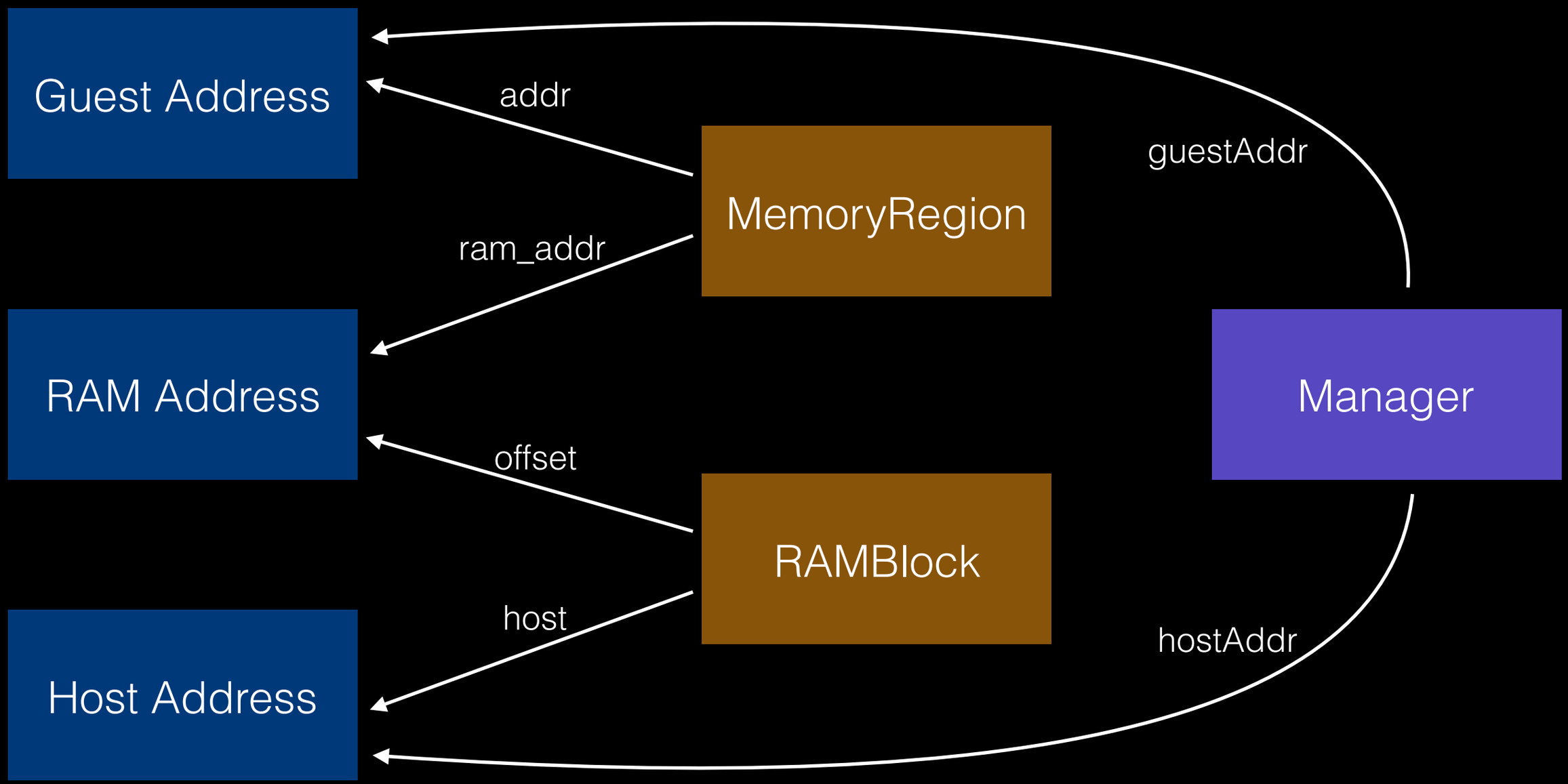
# Unicorn Memory Mapping API

```c
UNICORN_EXPORT
uc_err uc_mem_map(uc_engine *uc, uint64_t address, size_t size, uint32_t perms);

UNICORN_EXPORT
uc_err uc_mem_map_ptr(uc_engine *uc, uint64_t address, size_t size, uint32_t perms, void *ptr);
```

uc_mem_map -> unicorn implicitly handles memory management
    PRO : simple, easy to use
    CON : user has minimal direct control over mapped memory

uc_mem_map_ptr -> user provide the underlying memory map
    PRO : can emulate shared memory between different devices, more fine grained control on host side
    CON : must implement a memory manager to handle the mapped memories

# A Sane Memory Manager



# The Bug

when releasing memory, unicorn attempts to split the victim

```
1  UNICORN_EXPORT
2  uc_err uc_mem_unmap(struct uc_struct *uc, uint64_t address, size_t size)
3  {
4      MemoryRegion *mr;
5      uint64_t addr;
6      size_t count, len;
7
8      if (size == 0)
9          // nothing to unmap
10         return UC_ERR_OK;
11
12     // address must be aligned to uc->target_page_size
13     if ((address & uc->target_page_align) != 0)
14         return UC_ERR_ARG;
15
16     // size must be multiple of uc->target_page_size
17     if ((size & uc->target_page_align) != 0)
18         return UC_ERR_ARG;
19
20     if (uc->mem_redirect) {
21         address = uc->mem_redirect(address);
22     }
23
24     // check that user's entire requested block is mapped
25     if (!check_mem_area(uc, address, size))
26         return UC_ERR_NOMEM;
27
28     // Now we know entire region is mapped, so do the unmap
29     // We may need to split regions if this area spans adjacent regions
30     addr = address;
31     count = 0;
32     while(count < size) {
33         mr = memory_mapping(uc, addr);
34         len = (size_t)MIN(size - count, mr->end - addr);
35         if (!split_region(uc, mr, addr, len, true))
36             return UC_ERR_NOMEM;
37
38         // if we can retrieve the mapping, then no splitting took place
39         // so unmap here
40         mr = memory_mapping(uc, addr);
41         if (mr != NULL)
42             uc->memory_unmap(uc, mr);
43         count += len;
44         addr += len;
45     }
46
47     return UC_ERR_OK;
48 }
```

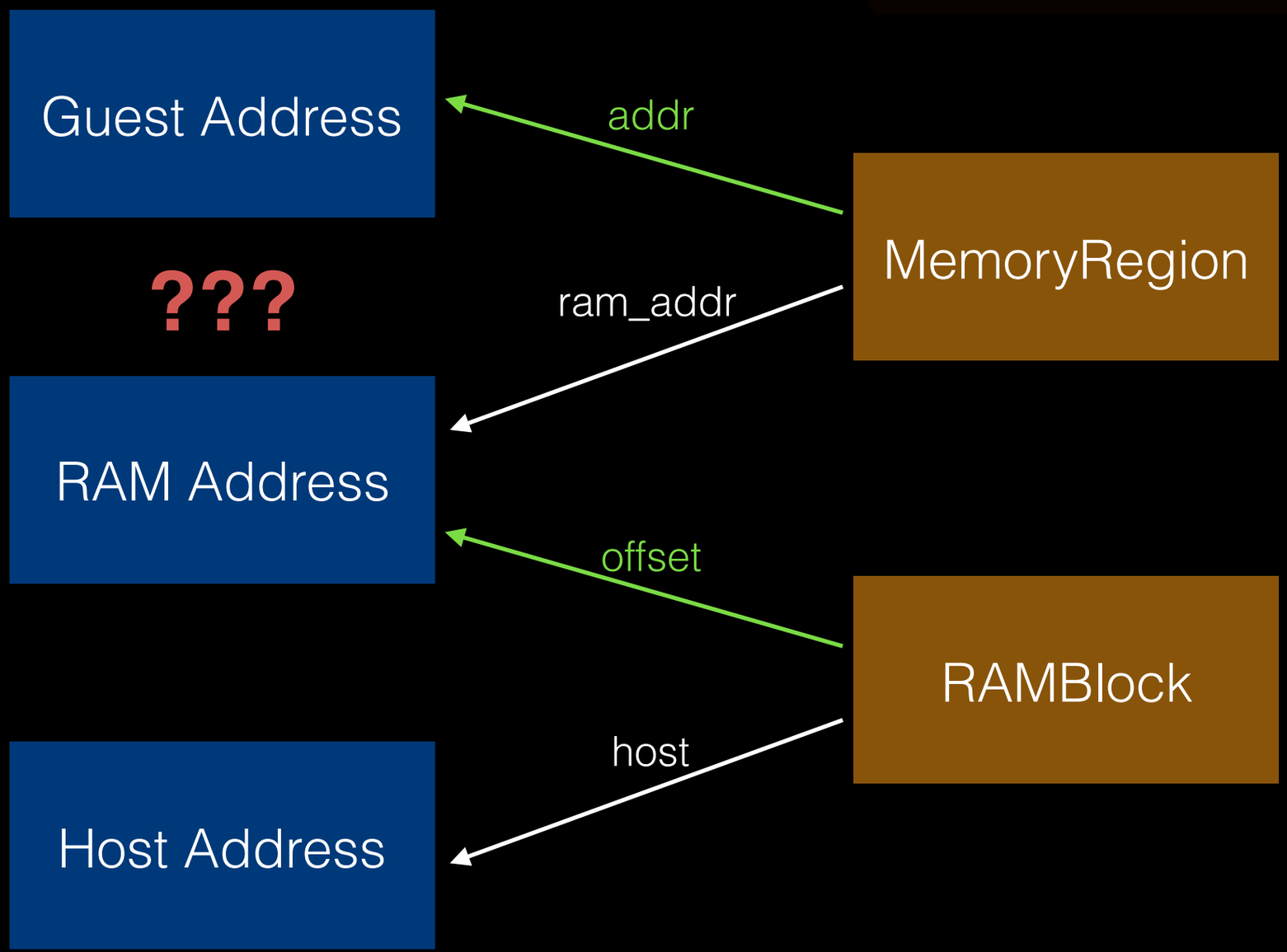
# The Bug

**split_region** checks if there is an actually need to split, and proceeds on fetching the victim RAMBlock if split is deemed necessary

```
1  static bool split_region(
2          struct uc_struct *uc, MemoryRegion *mr,
3          uint64_t address,
4          size_t size, bool do_delete)
5  {
6      uint8_t *backup;
7      uint32_t perms;
8      uint64_t begin, end, chunk_end;
9      size_t l_size, m_size, r_size;
10     RAMBlock *block = NULL;
11     bool prealloc = false;
12
13     chunk_end = address + size;
14
15     // if this region belongs to area [address, address+size],
16     // then there is no work to do.
17     if (address <= mr->addr && chunk_end >= mr->end)
18         return true;
19
20     if (size == 0)
21         // trivial case
22         return true;
23
24     if (address >= mr->end || chunk_end <= mr->addr)
25         // impossible case
26         return false;
27
28     QTAILQ_FOREACH(block, &uc->ram_list.blocks, next) {
29         if (block->offset <= mr->addr
30             && block->length >= (mr->end - mr->addr)) {
31             break;
32         }
33     }
34
35     ...
```

# The Bug

```
1     QTAILQ_FOREACH(block, &uc->ram_list.blocks, next) {
2         if (block->offset <= mr->addr
3             && block->length >= (mr->end - mr->addr)) {
4             break;
5         }
6     }
```



# The Bug

if the original backing memory is mapped through **uc_mem_map_ptr**, **split_region** fetches host ptr from incorrect RAMBlock

the original MemoryRegion & RAMBlock is then released through recursive call of **uc_mem_unmap**

the remaining memory is re-mapped again with **uc_mem_map_ptr**

```
1     if (block == NULL)
2         return false;
3
4     // RAM_PREALLOC is not defined outside exec.c and I didn't feel like
5     // moving it
6     prealloc = !!(block->flags & 1);
7
8     if (block->flags & 1) {
9         backup = block->host;
10    } else {
11        backup = copy_region(uc, mr);
12        if (backup == NULL)
13            return false;
14    }
15
16    // save the essential information required for the split before mr gets deleted
17    perms = mr->perms;
18    begin = mr->addr;
19    end = mr->end;
20
21    // unmap this region first, then do split it later
22    if (uc_mem_unmap(uc, mr->addr, (size_t)int128_get64(mr->size)) != UC_ERR_OK)
23        goto error;
24
25    // adjust some things
26    if (address < begin)
27        address = begin;
28    if (chunk_end > end)
29        chunk_end = end;
30
31    // compute sub region sizes
32    l_size = (size_t)(address - begin);
33    r_size = (size_t)(end - chunk_end);
34    m_size = (size_t)(chunk_end - address);
35
36    if (l_size > 0) {
37        if (!prealloc) {
38            if (uc_mem_map(uc, begin, l_size, perms) != UC_ERR_OK)
39                goto error;
40            if (uc_mem_write(uc, begin, backup, l_size) != UC_ERR_OK)
41                goto error;
42        } else {
43            if (uc_mem_map_ptr(uc, begin, l_size, perms, backup) != UC_ERR_OK)
44                goto error;
45        }
46    }
47
48    if (m_size > 0 && !do_delete) {
49        ...
50    }
51
52    if (r_size > 0) {
53        ...
54    }
55
56    ...
57 }
```

# Exploit

MemoryRegion1

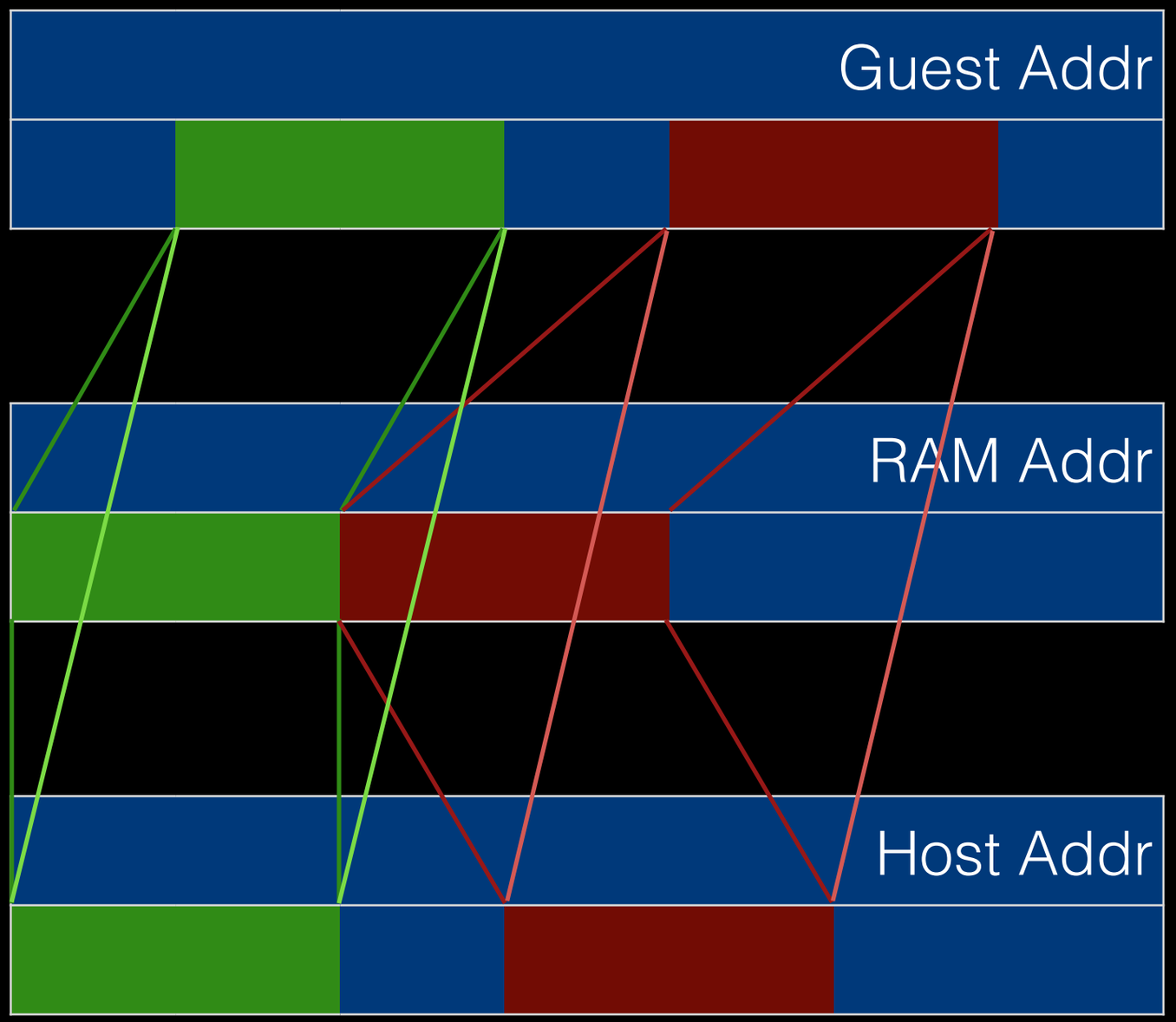
MemoryRegion2

Manager1

Manager2

RAMBlock1

RAMBlock2



action : free second half of Chunk2

# Exploit

MemoryRegion1

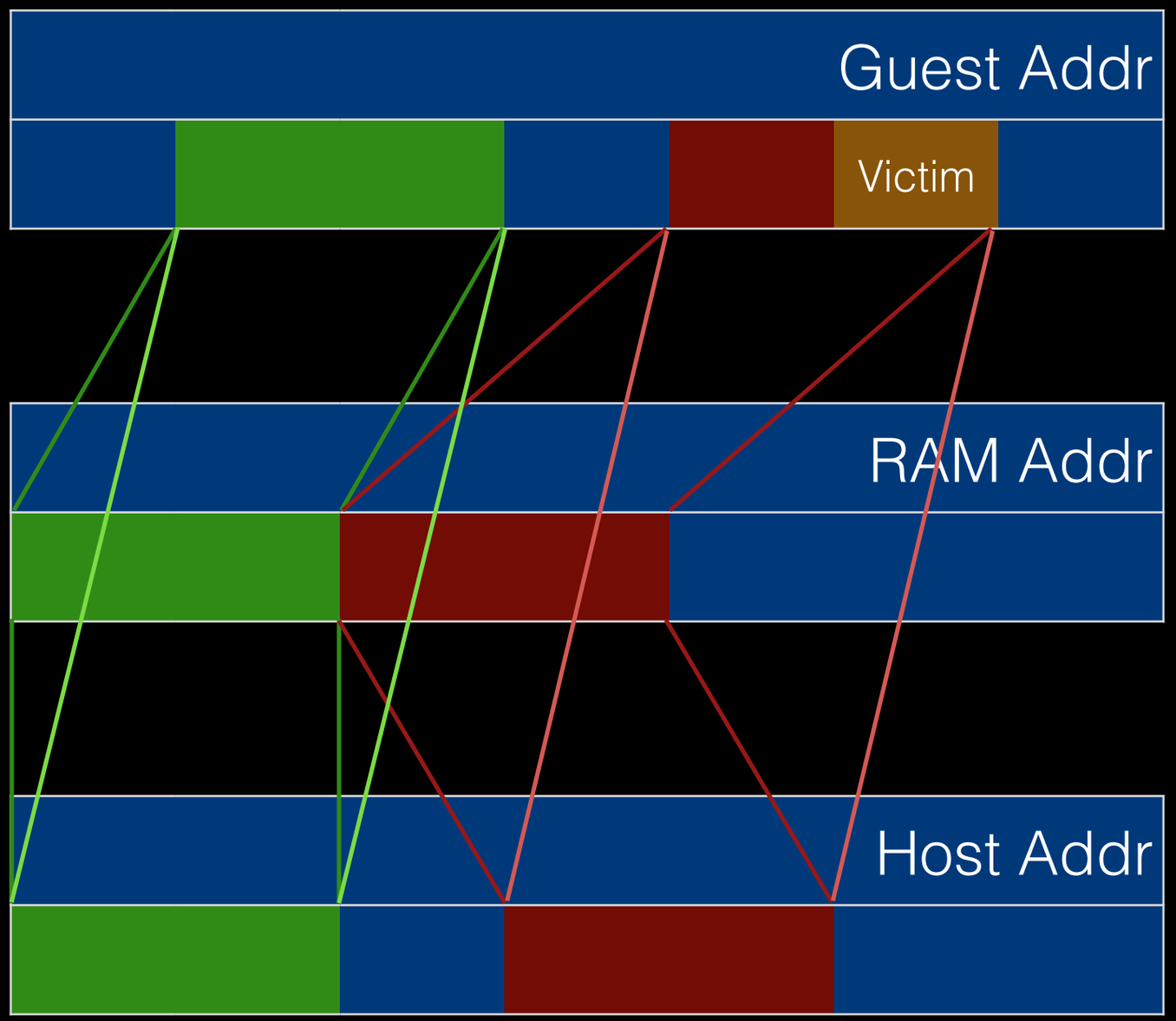
MemoryRegion2

Manager1

Manager2

RAMBlock1

RAMBlock2



# Exploit

action : free second half of Chunk2
find Manager

MemoryRegion1

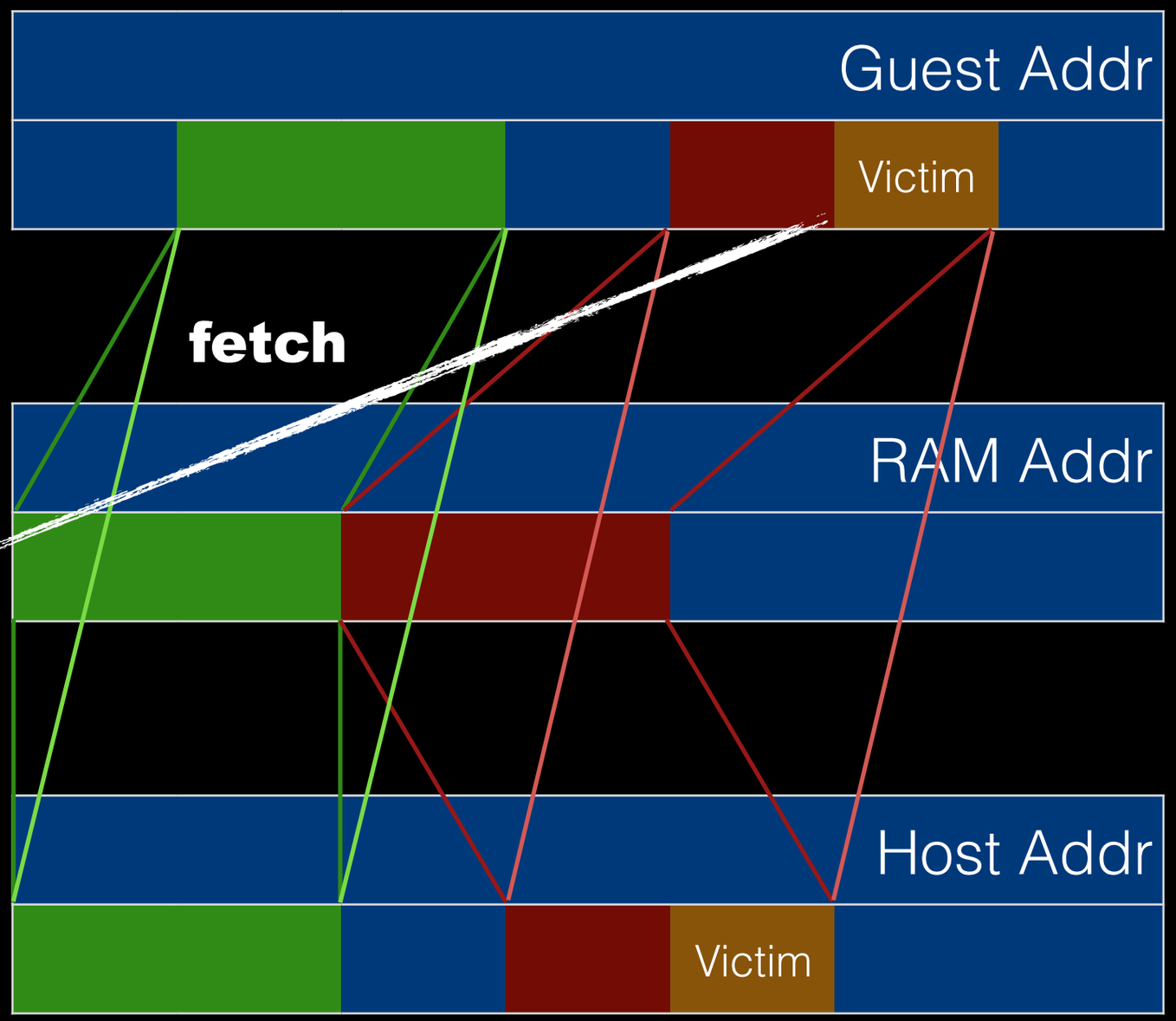
MemoryRegion2

Manager1

Manager2

RAMBlock1

RAMBlock2



# Exploit

action : free second half of Chunk2
request unicorn to release chunk via
**uc_mem_unmap**
unicorn fetches mr with guest addr

MemoryRegion1

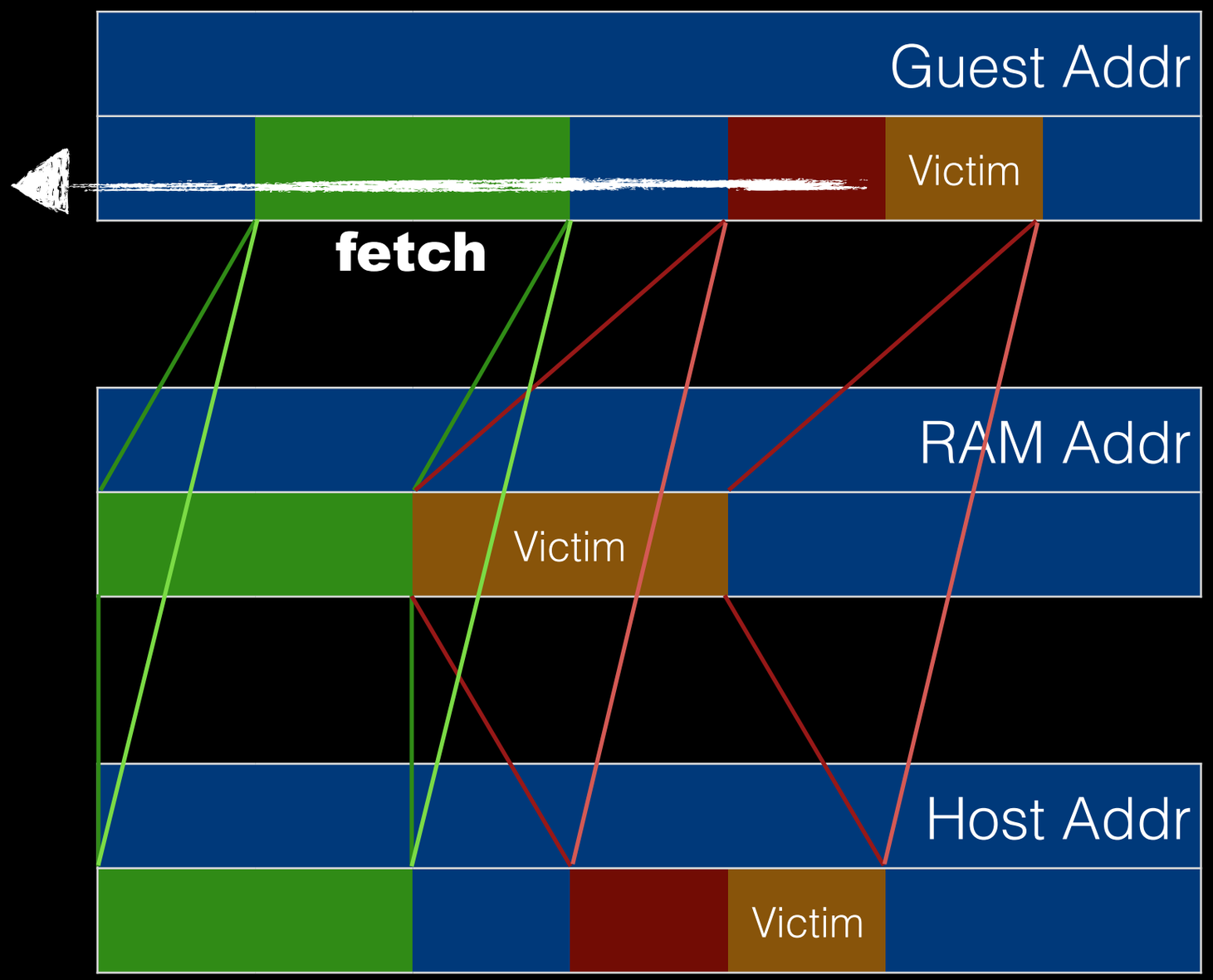
MemoryRegion2

Manager1

Manager2

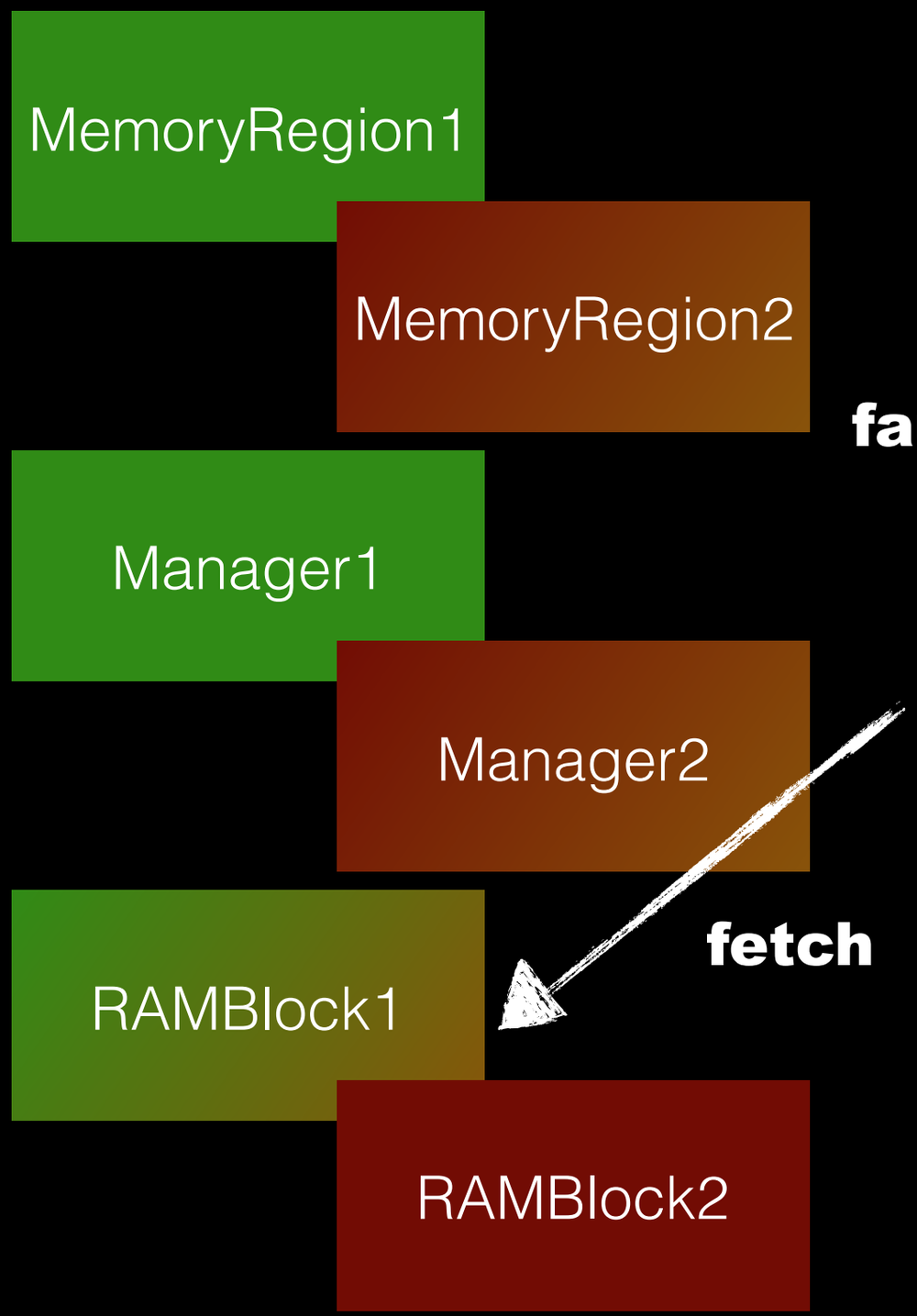
RAMBlock1

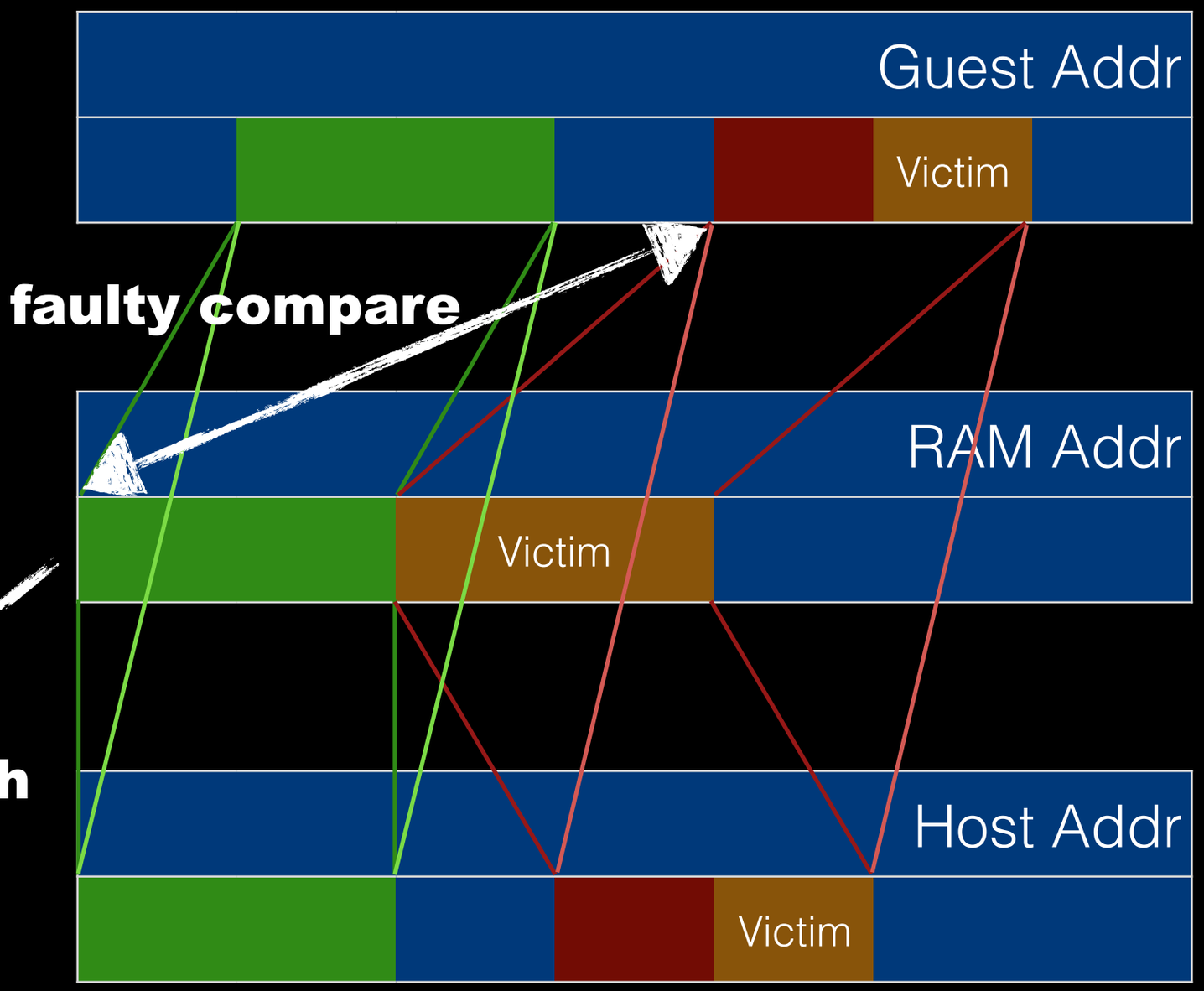
RAMBlock2



# Exploit

action : free second half of Chunk2
request unicorn to release chunk via
**uc_mem_unmap**
unicorn fetches incorrect RAMBlock



# Exploit

action : free second half of Chunk2
request unicorn to release chunk via
**uc_mem_unmap**
lookup backing host mem

MemoryRegion1

MemoryRegion2

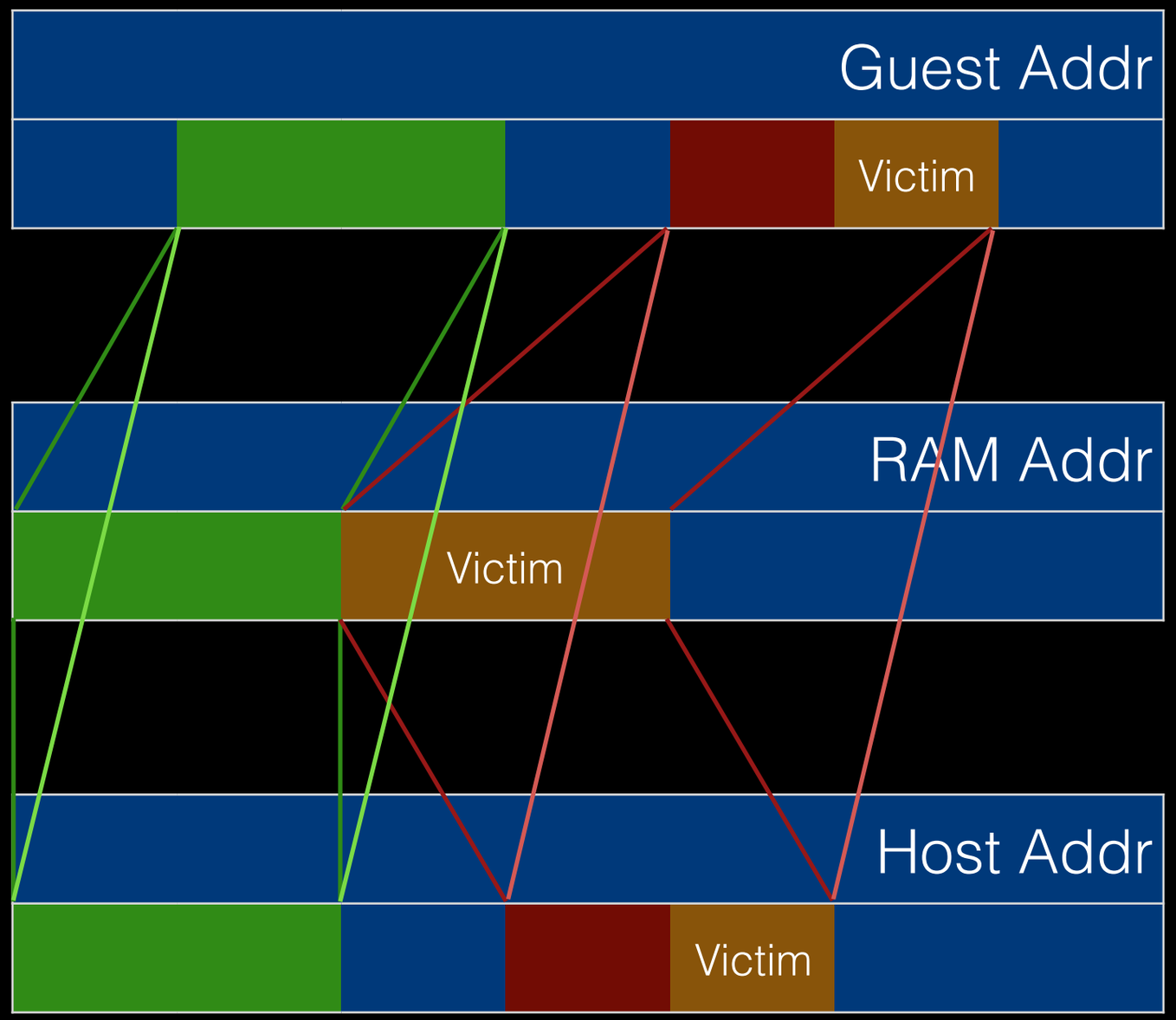
Manager1

Manager2

RAMBlock1

RAMBlock2

**lookup**



# Exploit

action : free second half of Chunk2
request unicorn to release chunk via
**uc_mem_unmap**
release MemoryRegion & RAMBlock

MemoryRegion1

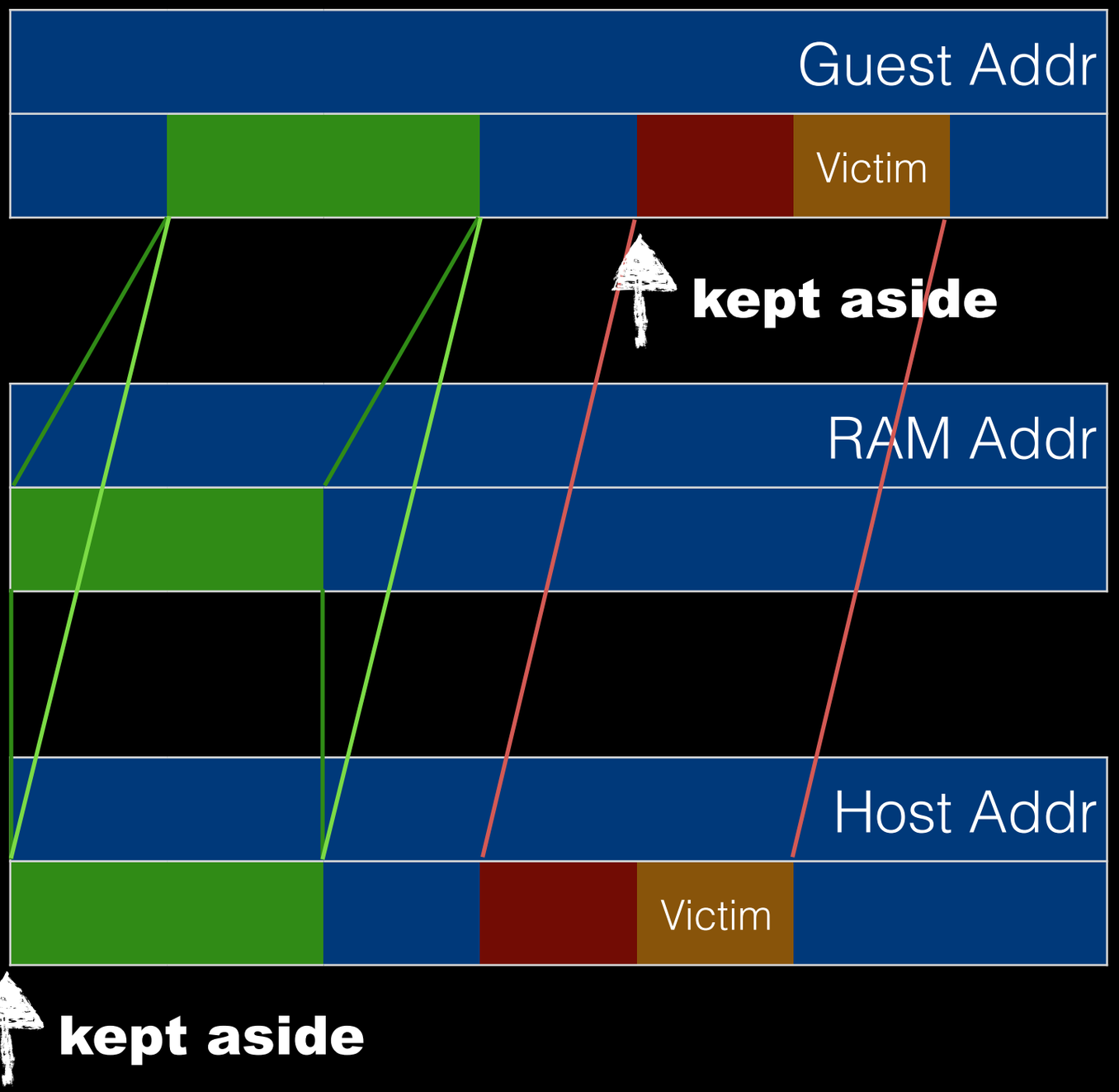
MemoryRegion2

Manager1

Manager2

RAMBlock1

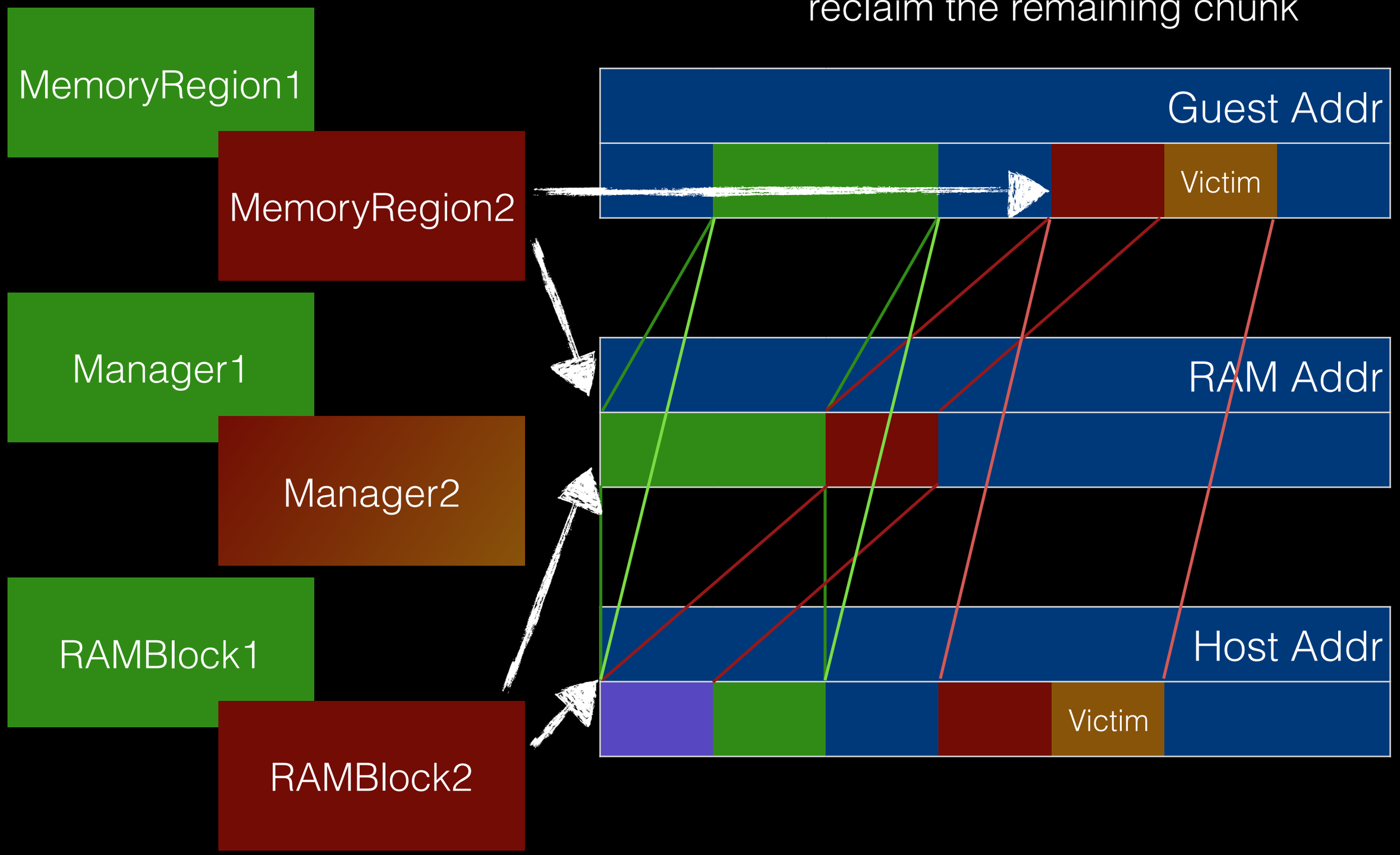
RAMBlock2




Note that the actual deletion of RAMBlock fetches correctly

# Exploit

action : free second half of Chunk2
request unicorn to release chunk via **uc_mem_unmap**
reclaim the remaining chunk



# Exploit

action : free second half of Chunk2
update Manager & release host mem

MemoryRegion1

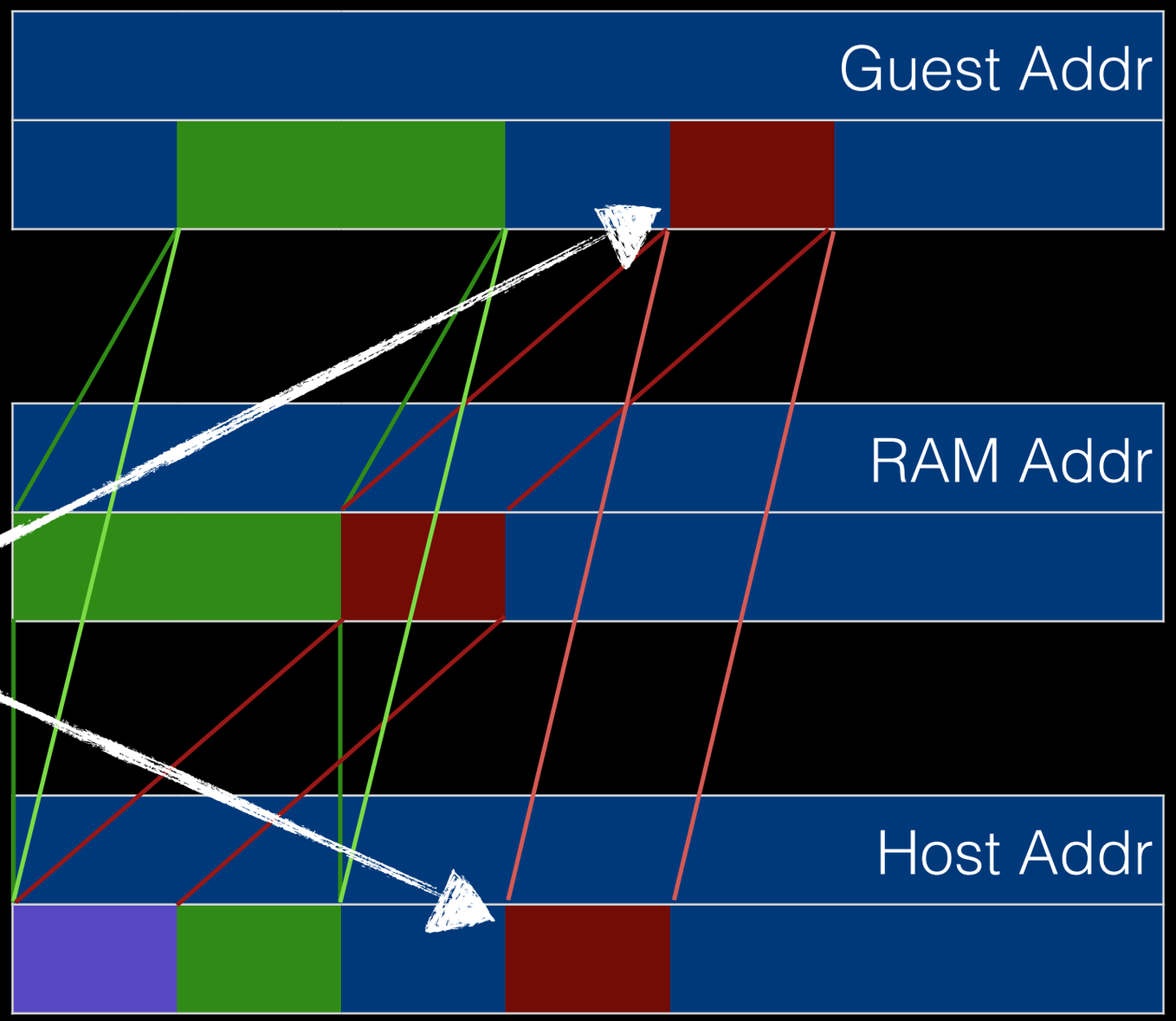
MemoryRegion2

Manager1

Manager2

RAMBlock1

RAMBlock2



# Exploit

At this point, we have 2 RAMBlocks sharing same host addr
How does this affect system integrity?

MemoryRegion1

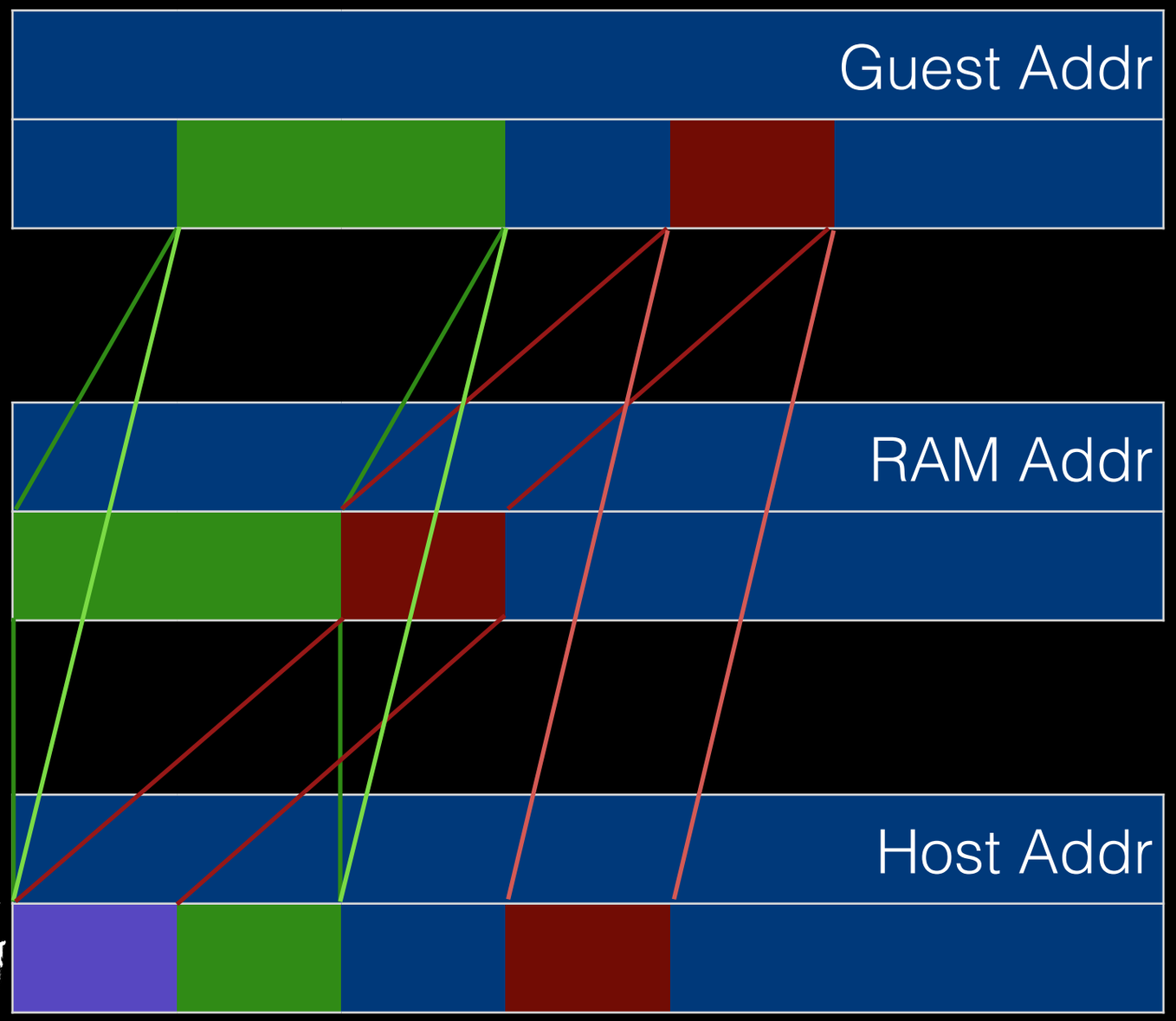
MemoryRegion2

Manager1

Manager2

RAMBlock1

RAMBlock2

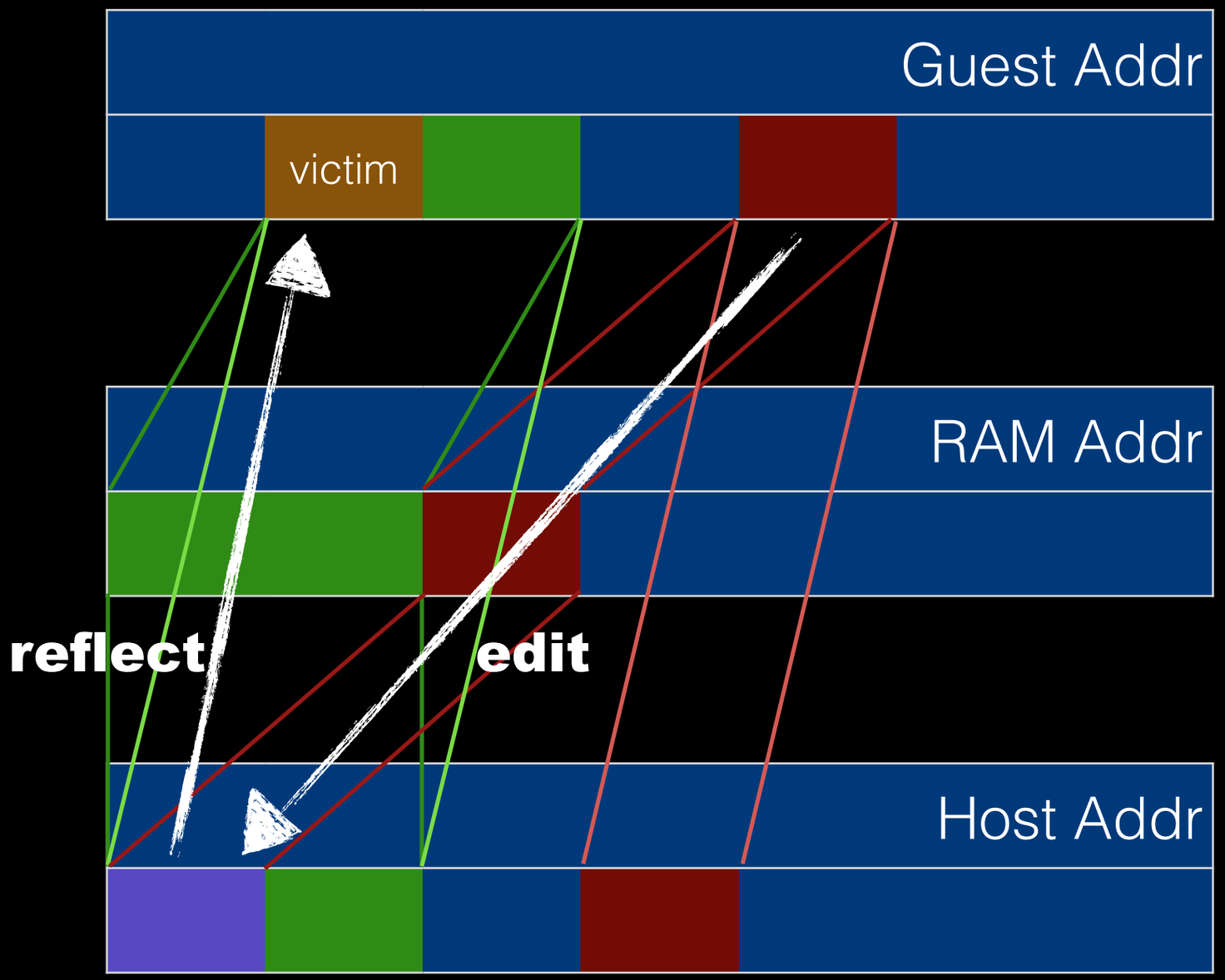


# Guest Exploit

by editing Mem2,
we can modify guest content for Mem1
effectively breaks all GVA assumptions
& protection

Mem1 Attr
host : rw
guest : rx

Mem2 Attr
host : rw
guest : rw



# Host Exploit

At this point we could elevate from semi-arbitrary code execution to true arbitrary code execution in guest program

But guest only exploit is lame, how should we further escape unicorn and gain code execution in host space?

# Host Exploit

action : free Chunk1
fast forward through **uc_mem_unmap**

MemoryRegion1

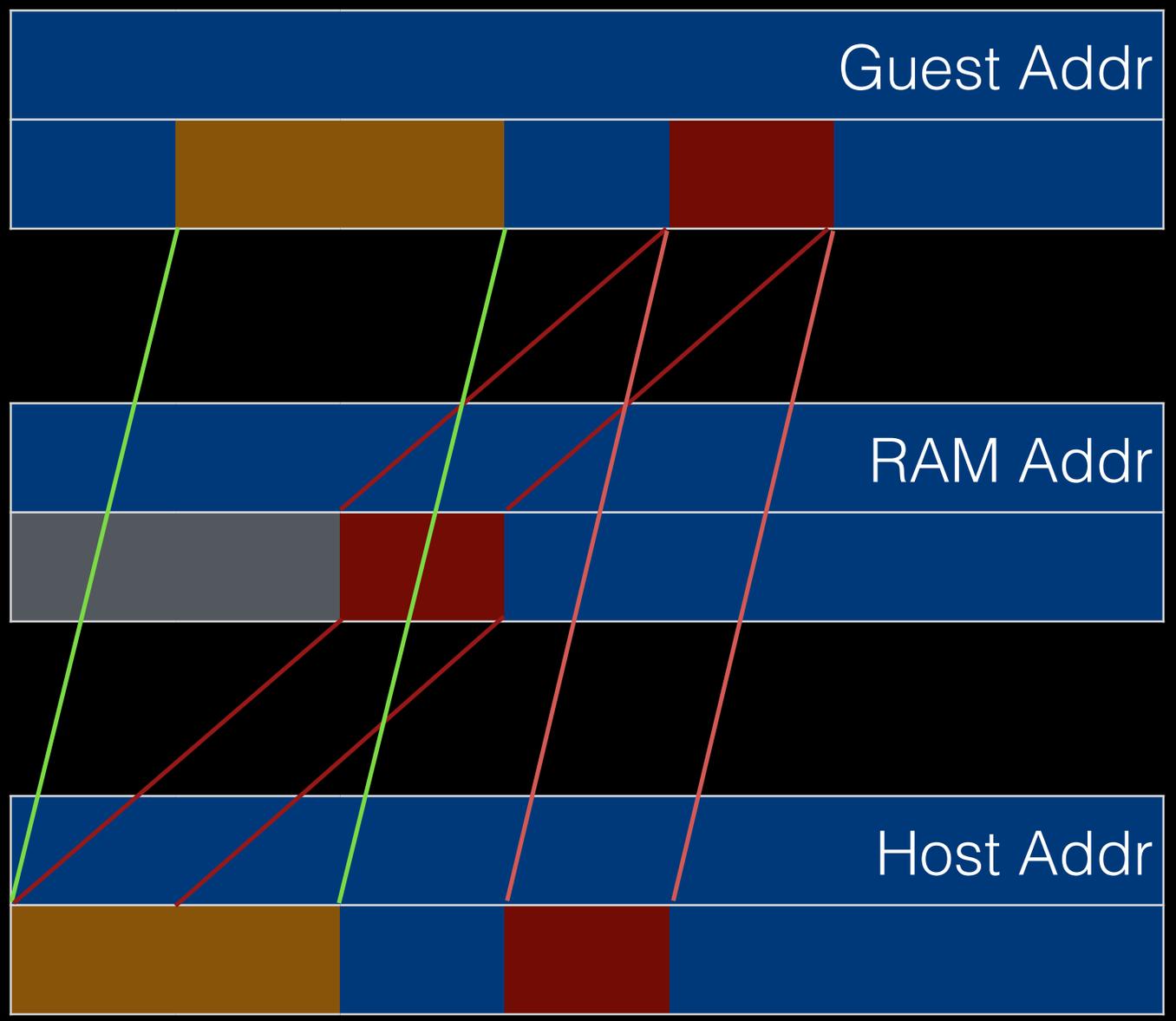
MemoryRegion2

Manager1

Manager2

RAMBlock1

RAMBlock2



# Host Exploit

action : free Chunk1
release Manager1,
and now we have a UAF for Chunk2

MemoryRegion1

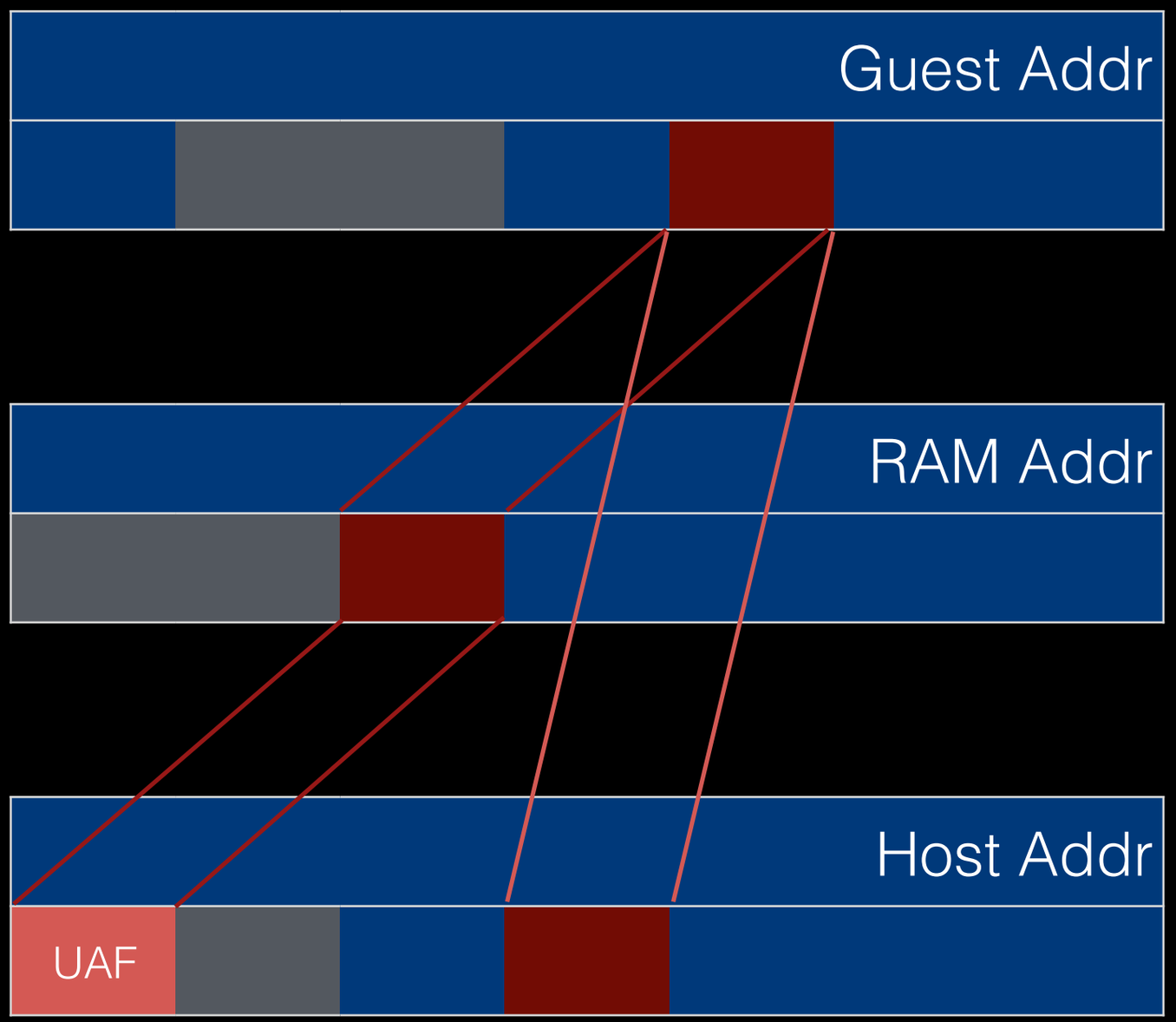
MemoryRegion2

Manager1

Manager2

RAMBlock1

RAMBlock2



# Host Exploit

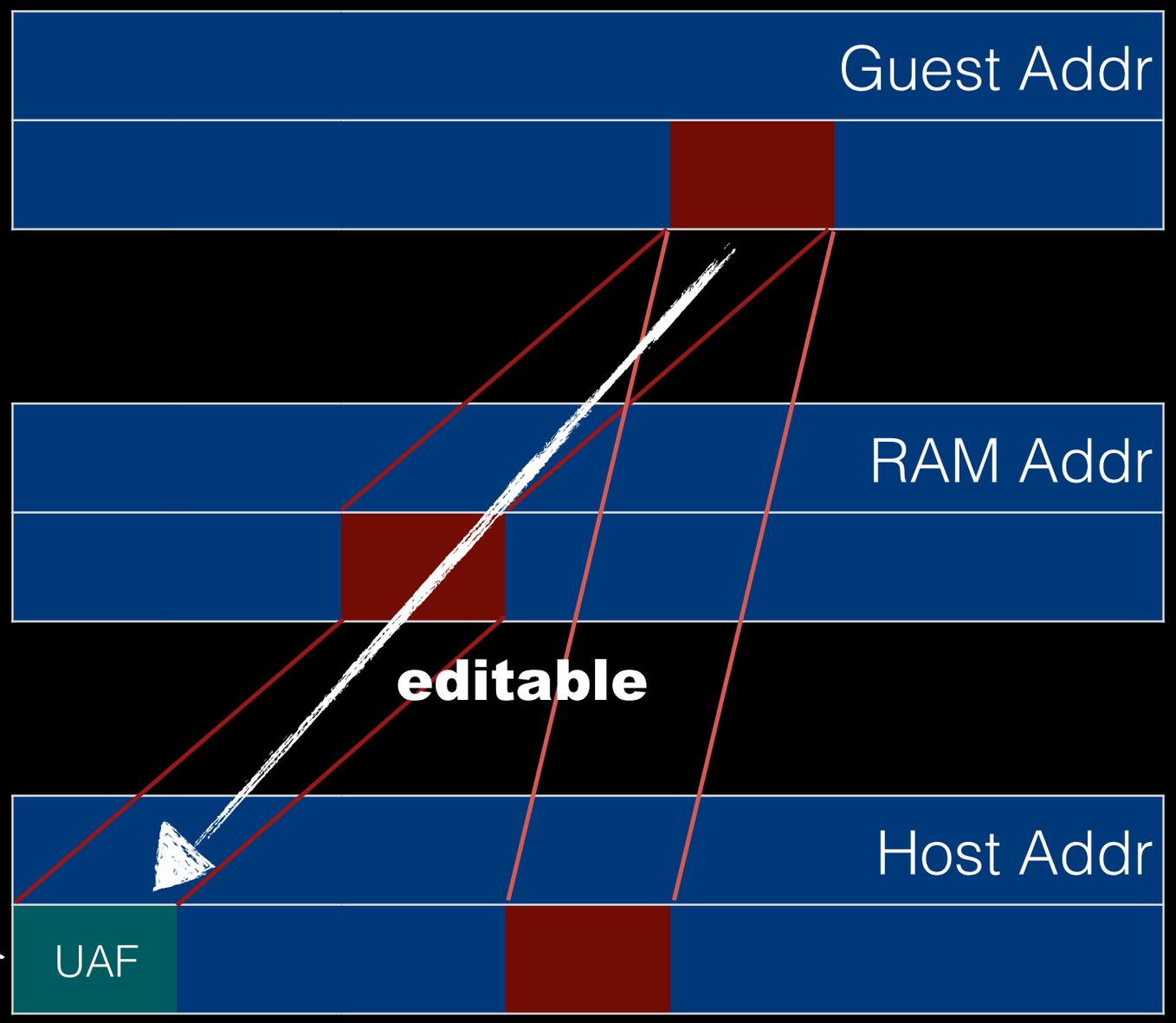
if host reclaims chunk at some point,
we can modify its memory content

MemoryRegion2

ManagerChunk2

RAMBlock2

hostUsage



# Host Exploit

let's go a step further,
if application does not reclaim host mem
is unicorn engine itself prone to attack?

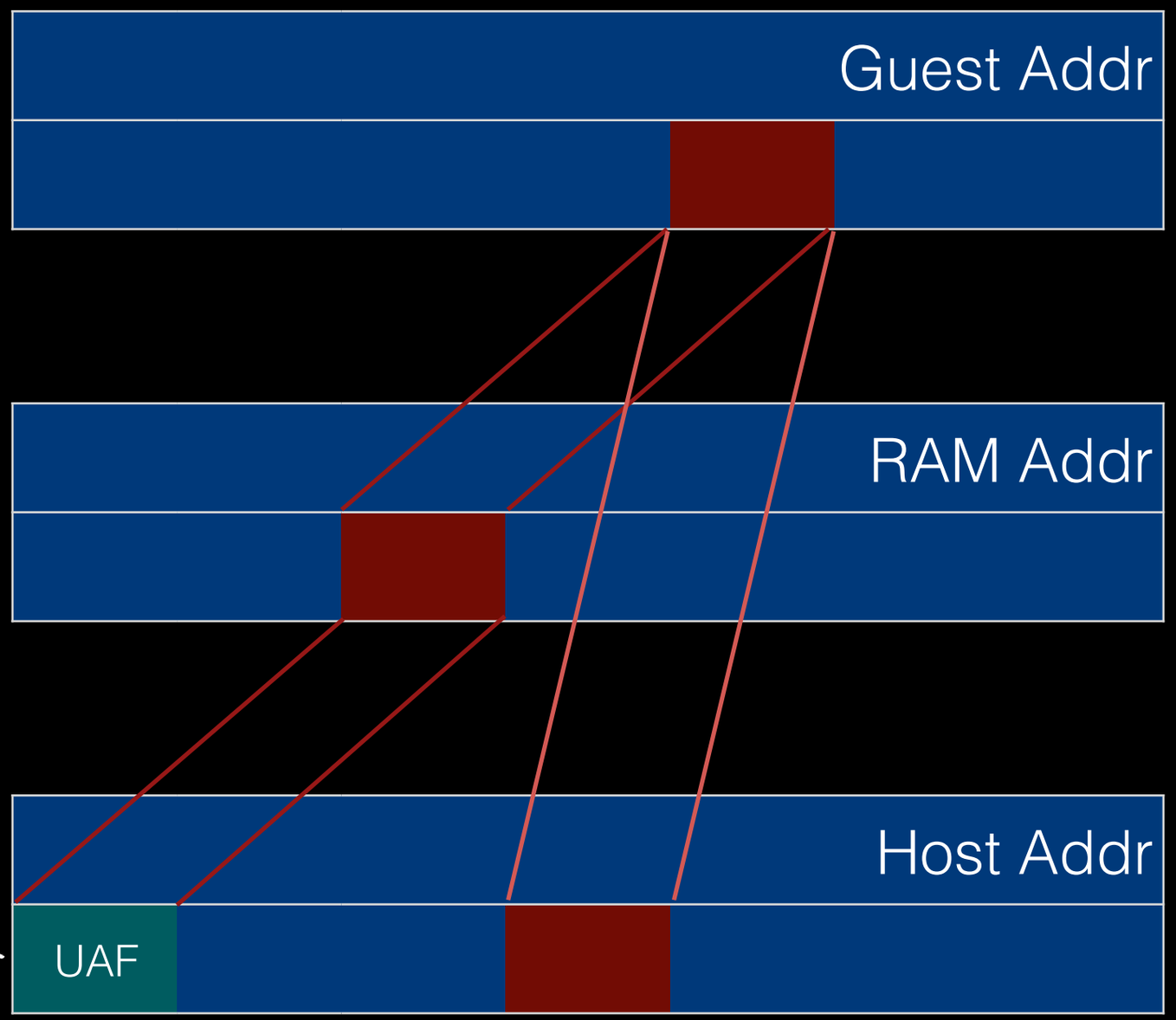
MemoryRegion2

ManagerChunk2

RAMBlock2

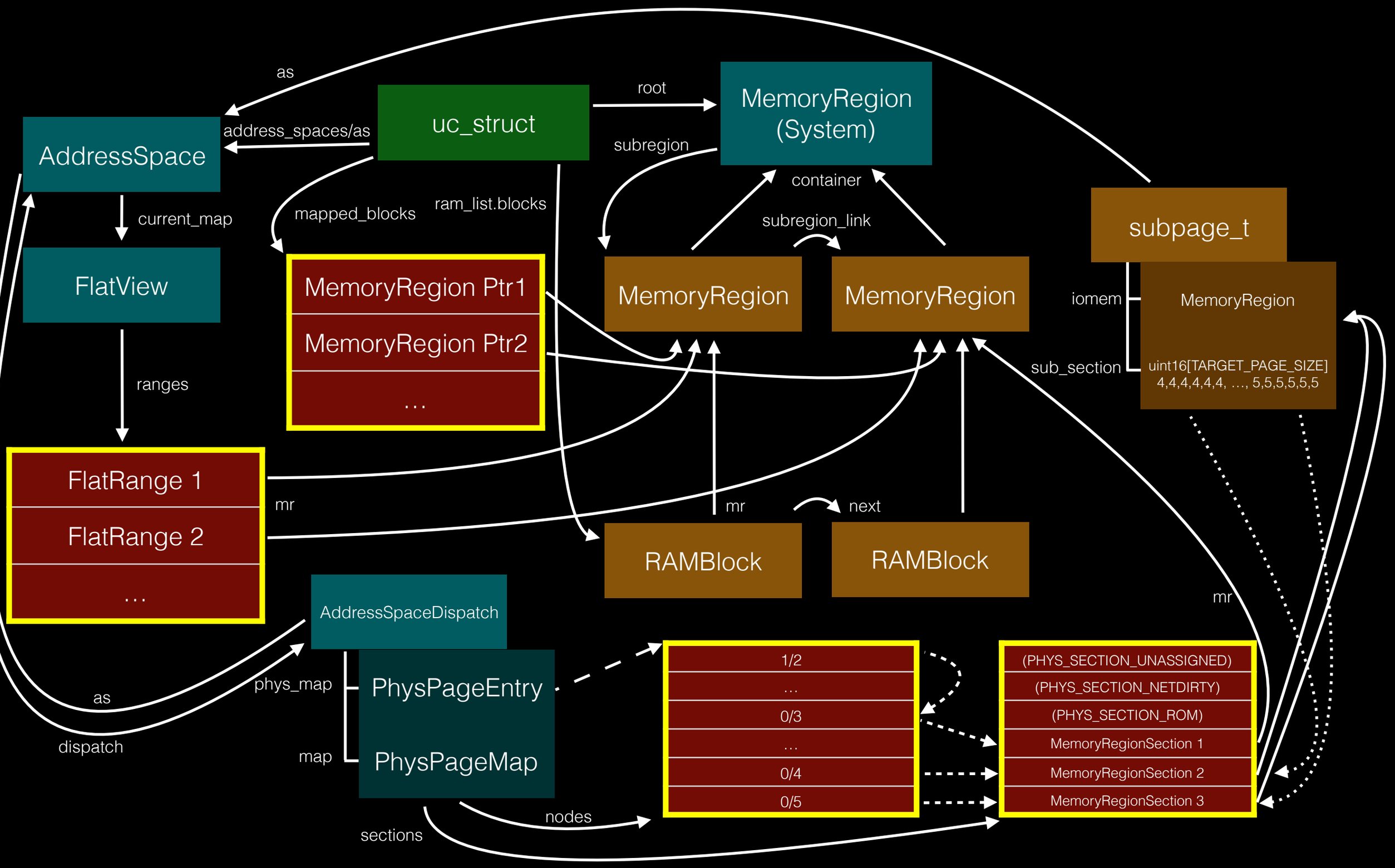
unicornUsage

?



# Select Target

assume all backing memories are mmapped +
limit attack interface to mem management
resizable arrays are the only possible candidates



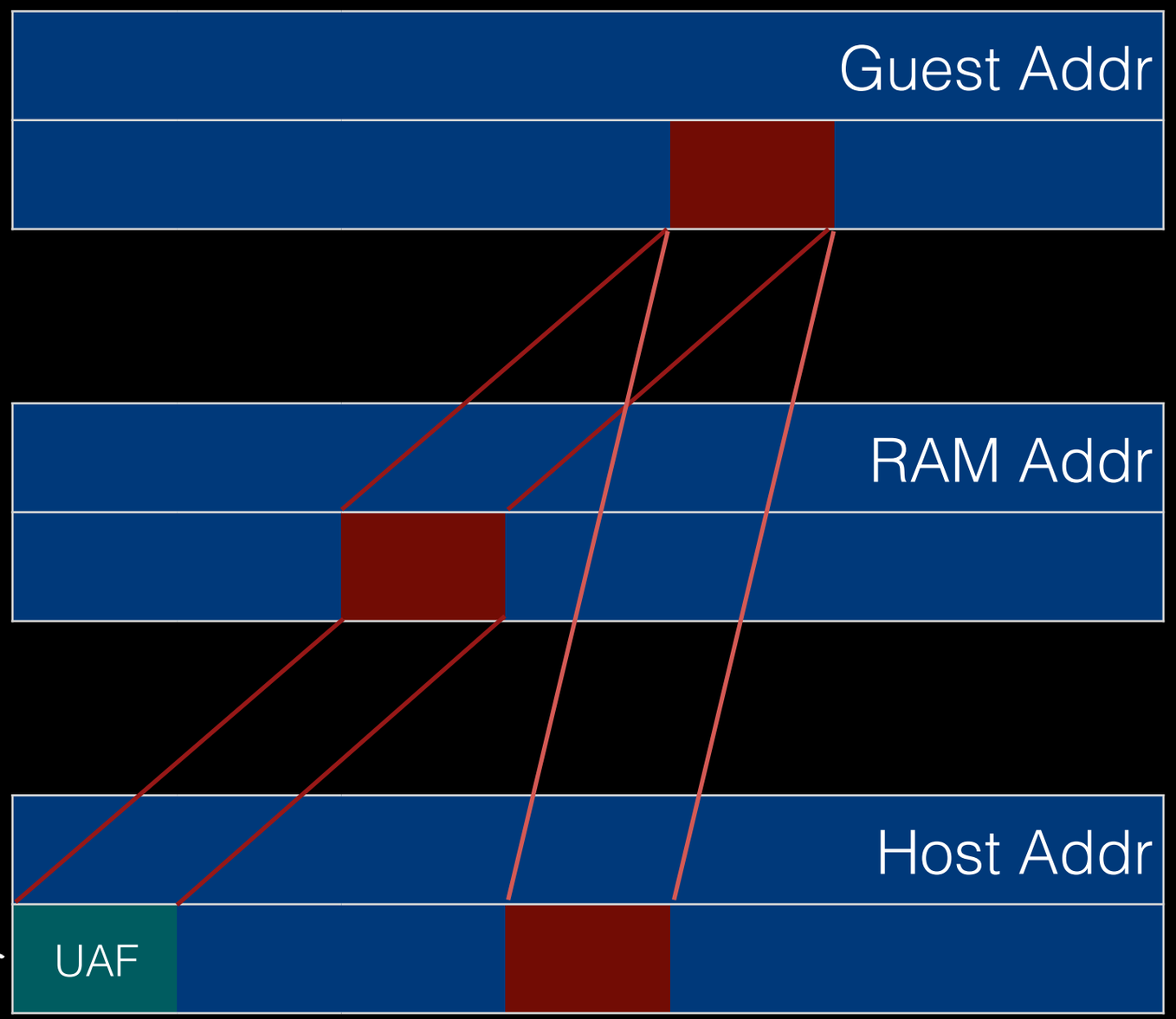
# Mapped_Blocks Exploit

starting with mapped_blocks first, we will get a setup like this

MemoryRegion

Manager

RAMBlock




MemoryRegion Ptr1

MemoryRegion Ptr2

...

# Mapped_Blocks Exploit

With this, we have arbitrary read/write control over the mapped_blocks array

This mean it is possible to

1. leak heap addr (since MemoryRegions are bound to live on heap)
2. modify MemoryRegion pointers

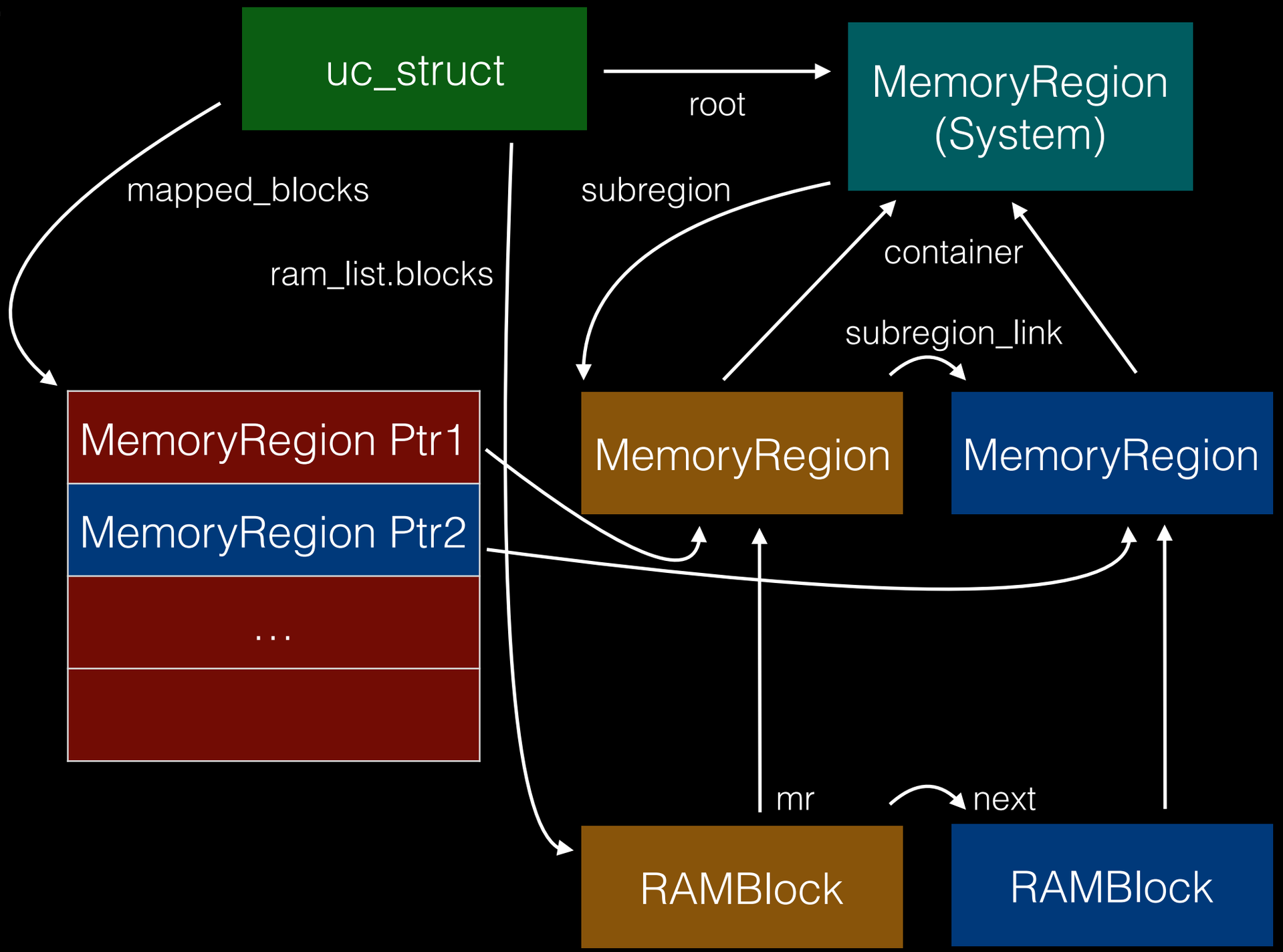
We do not have fine grained control over heap content, leaking heap addr is not that useful in itself

Will focus on how to corrupt the mapped_blocks array and bypass checks

Without fine grained control, we most likely will only be able to reuse freed MemoryRegion blocks

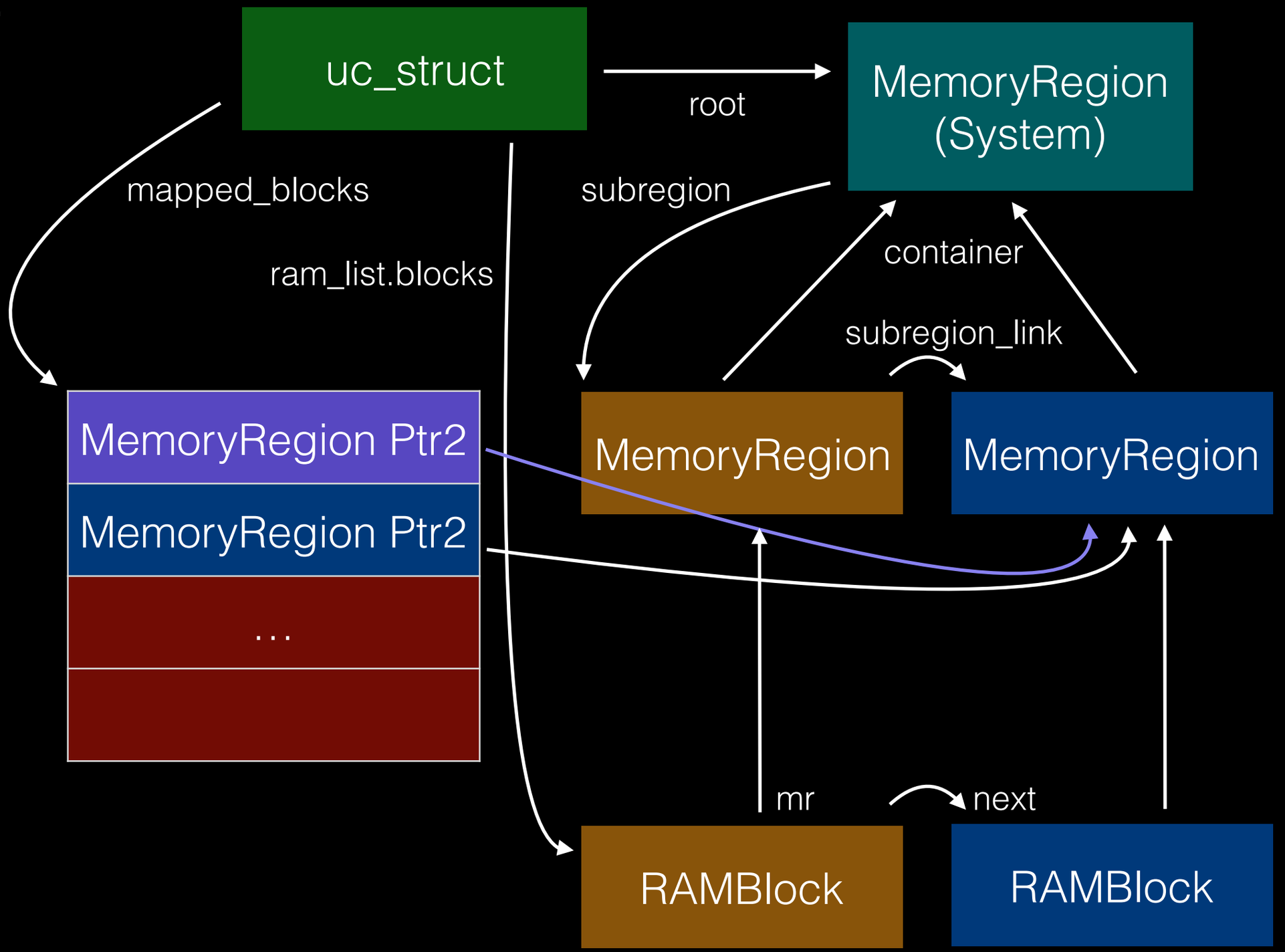
# Mapped_Blocks Exploit

release the second MemoryRegion



# Mapped_Blocks Exploit

modify first MemoryRegion to point to the freed region



# Mapped_Blocks Exploit

Previously, we only utilized desynchronize between RAMBlock/Manager to achieve shared Host Addr

Now given the MemoryRegion modifying primitive, we are also capable of desynchronizing MemoryRegion/RAMBlock or MemoryRegion/Manager

In other words, it's possible to gain shared Guest Addr / RAM Addr

Now we have gained this primitive, what can we possibly do?

Time to dive deeper into unicorn APIs

# API Review

unicorn read/write follows a similar set of logic

it first checks if target guest addr exist by calling **check_mem_area**

then reads to each memory chunk while respecting its size

note that **uc->read_mem** is passed an addr, not a MemoryRegion ptr

```c
UNICORN_EXPORT
uc_err uc_mem_read(
        uc_engine *uc, uint64_t address,
        void *_bytes, size_t size)
{
    size_t count = 0, len;
    uint8_t *bytes = _bytes;

    if (uc->mem_redirect) {
        address = uc->mem_redirect(address);
    }

    if (!check_mem_area(uc, address, size))
        return UC_ERR_READ_UNMAPPED;

    // memory area can overlap adjacent memory blocks
    while(count < size) {
        MemoryRegion *mr = memory_mapping(uc, address);
        if (mr) {
            len = (size_t)MIN(size - count, mr->end - address);
            if (uc->read_mem(&uc->as, address, bytes, len) == false)
                break;
            count += len;
            address += len;
            bytes += len;
        } else  // this address is not mapped in yet
            break;
    }

    if (count == size)
        return UC_ERR_OK;
    else
        return UC_ERR_READ_UNMAPPED;
}
```

# API Review

**check_mem_area** operates by by calling **memory_mapping**

**memory_mapping** then iterates mapped_blocks to find mr

```c
static bool check_mem_area(uc_engine *uc, uint64_t address, size_t size)
{
    size_t count = 0, len;

    while(count < size) {
        MemoryRegion *mr = memory_mapping(uc, address);
        if (mr) {
            len = (size_t)MIN(size - count, mr->end - address);
            count += len;
            address += len;
        } else  // this address is not mapped in yet
            break;
    }

    return (count == size);
}

MemoryRegion *memory_mapping(struct uc_struct* uc, uint64_t address)
{
    unsigned int i;

    if (uc->mapped_block_count == 0)
        return NULL;

    if (uc->mem_redirect) {
        address = uc->mem_redirect(address);
    }

    // try with the cache index first
    i = uc->mapped_block_cache_index;

    if (i < uc->mapped_block_count
        && address >= uc->mapped_blocks[i]->addr
        && address < uc->mapped_blocks[i]->end)
        return uc->mapped_blocks[i];

    for(i = 0; i < uc->mapped_block_count; i++) {
        if (address >= uc->mapped_blocks[i]->addr
            && address <= uc->mapped_blocks[i]->end - 1) {
            // cache this index for the next query
            uc->mapped_block_cache_index = i;
            return uc->mapped_blocks[i];
        }
    }

    // not found
    return NULL;
}
```

# API Review

it then goes through a couple of wrapper func before reaching **address_space_rw**

**address_space_rw** attempts to fetch mr with **address_space_translate**

```
1  static inline bool cpu_physical_mem_read(AddressSpace *as, hwaddr addr,
2          uint8_t *buf, int len)
3  {
4      return !cpu_physical_memory_rw(as, addr, (void *)buf, len, 0);
5  }
6
7  bool cpu_physical_memory_rw(AddressSpace *as, hwaddr addr, uint8_t *buf,
8          int len, int is_write)
9  {
10     return address_space_rw(as, addr, buf, len, is_write);
11 }
12
13 bool address_space_rw(AddressSpace *as, hwaddr addr, uint8_t *buf,
14         int len, bool is_write)
15 {
16     hwaddr l;
17     uint8_t *ptr;
18     uint64_t val;
19     hwaddr addr1;
20     MemoryRegion *mr;
21     bool error = false;
22
23     while (len > 0) {
24         l = len;
25
26         mr = address_space_translate(as, addr, &addr1, &l, is_write);
27         if (!mr)
28             return true;
29
30     ...
31 }
```

# API Review

let's see how mr block is fetched in **address_space_translate**

first, MemoryRegionSection is fetched by **address_space_translate_internal**

then mr is extracted from section->mr

mr->iommu_ops should not be set for RAM mr, so the fetched mr is returned directly

```
1  MemoryRegion *address_space_translate(AddressSpace *as, hwaddr addr,
2          hwaddr *xlat, hwaddr *plen,
3          bool is_write)
4  {
5      IOMMUTLBEntry iotlb;
6      MemoryRegionSection *section;
7      MemoryRegion *mr;
8      hwaddr len = *plen;
9
10     for (;;) {
11         section = address_space_translate_internal(
12                           as->dispatch, addr, &addr, plen, true);
13         mr = section->mr;
14         if (mr->ops == NULL)
15             return NULL;
16
17         if (!mr->iommu_ops) {
18             break;
19         }
20
21         iotlb = mr->iommu_ops->translate(mr, addr, is_write);
22         addr = ((iotlb.translated_addr & ~iotlb.addr_mask)
23                 | (addr & iotlb.addr_mask));
24         len = MIN(len, (addr | iotlb.addr_mask) - addr + 1);
25         if (!(iotlb.perm & (1 << is_write))) {
26             mr = &as->uc->io_mem_unassigned;
27             break;
28         }
29
30         as = iotlb.target_as;
31     }
32
33     *plen = len;
34     *xlat = addr;
35     return mr;
36 }
```

# API Review

the following lookup
is basically a wrapper
around
**phys_page_find**

```c
static MemoryRegionSection *
address_space_translate_internal(AddressSpaceDispatch *d,
        hwaddr addr, hwaddr *xlat,
        hwaddr *plen, bool resolve_subpage)
{
    MemoryRegionSection *section;
    Int128 diff;

    section = address_space_lookup_region(d, addr, resolve_subpage);
    /* Compute offset within MemoryRegionSection */
    addr -= section->offset_within_address_space;

    /* Compute offset within MemoryRegion */
    *xlat = addr + section->offset_within_region;

    diff = int128_sub(section->mr->size, int128_make64(addr));
    *plen = int128_get64(int128_min(diff, int128_make64(*plen)));
    return section;
}

static MemoryRegionSection *address_space_lookup_region(
        AddressSpaceDispatch *d, hwaddr addr, bool resolve_subpage)
{
    MemoryRegionSection *section;
    subpage_t *subpage;

    section = phys_page_find(d->phys_map, addr,
                             d->map.nodes, d->map.sections);
    if (resolve_subpage && section->mr->subpage) {
        subpage = container_of(section->mr, subpage_t, iomem);
        section = &d->map.sections[subpage->sub_section[SUBPAGE_IDX(addr)]];
    }
    return section;
}
```

# API Review

**phys_page_find** traverses the previously seen PhysPageMap page table to find corresponding pages

notice that it still checks whether the target addr is in fetched page

```
1  static MemoryRegionSection *phys_page_find(PhysPageEntry lp, hwaddr addr,
2          Node *nodes, MemoryRegionSection *sections)
3  {
4      PhysPageEntry *p;
5      hwaddr index = addr >> TARGET_PAGE_BITS;
6      int i;
7
8      for (i = P_L2_LEVELS; lp.skip && (i -= lp.skip) >= 0;) {
9          if (lp.ptr == PHYS_MAP_NODE_NIL) {
10             return &sections[PHYS_SECTION_UNASSIGNED];
11         }
12         p = nodes[lp.ptr];
13         lp = p[(index >> (i * P_L2_BITS)) & (P_L2_SIZE - 1)];
14     }
15
16     if (sections[lp.ptr].size.hi ||
17             range_covers_byte(sections[lp.ptr].offset_within_address_space,
18                 sections[lp.ptr].size.lo, addr)) {
19         return &sections[lp.ptr];
20     } else {
21         return &sections[PHYS_SECTION_UNASSIGNED];
22     }
23 }
24
25 static inline int range_covers_byte(uint64_t offset, uint64_t len,
26                                     uint64_t byte)
27 {
28     return offset <= byte && byte <= range_get_last(offset, len);
29 }
```

# API Review

after fetching mr, unicorn checks whether we are doing read or write

it then decides whether special care is required by querying **memory_access_is_direct**, which is always false for RAM

if the memory allows direct access, it first extracts ram_addr from mr, then uses it to fetch backing host mem ptr with **qemu_get_ram_ptr**, and finally proceeds to **memcpy** contents from/to it

```
1         if (is_write) {
2             ...
3         } else {
4             if (!memory_access_is_direct(mr, is_write)) {
5                 /* I/O case */
6                 l = memory_access_size(mr, l, addr1);
7
8                 switch (l) {
9                     case 8:
10                        /* 64 bit read access */
11                        error |= io_mem_read(mr, addr1, &val, 8);
12                        stq_p(buf, val);
13                        break;
14                    case 4:
15                        /* 32 bit read access */
16                        error |= io_mem_read(mr, addr1, &val, 4);
17                        stl_p(buf, val);
18                        break;
19                    case 2:
20                        /* 16 bit read access */
21                        error |= io_mem_read(mr, addr1, &val, 2);
22                        stw_p(buf, val);
23                        break;
24                    case 1:
25                        /* 8 bit read access */
26                        error |= io_mem_read(mr, addr1, &val, 1);
27                        stb_p(buf, val);
28                        break;
29                    default:
30                        abort();
31                }
32            } else {
33                /* RAM case */
34                ptr = qemu_get_ram_ptr(as->uc, mr->ram_addr + addr1);
35                memcpy(buf, ptr, l);
36            }
37        }
38        len -= l;
39        buf += l;
40        addr += l;
41    }
42
43    return error;
44 }
```

# API Review

**qemu_get_ram_ptr** calls **qemu_get_ram_block**, which traverses uc->ram_list.blocks for the block that contains RAM address

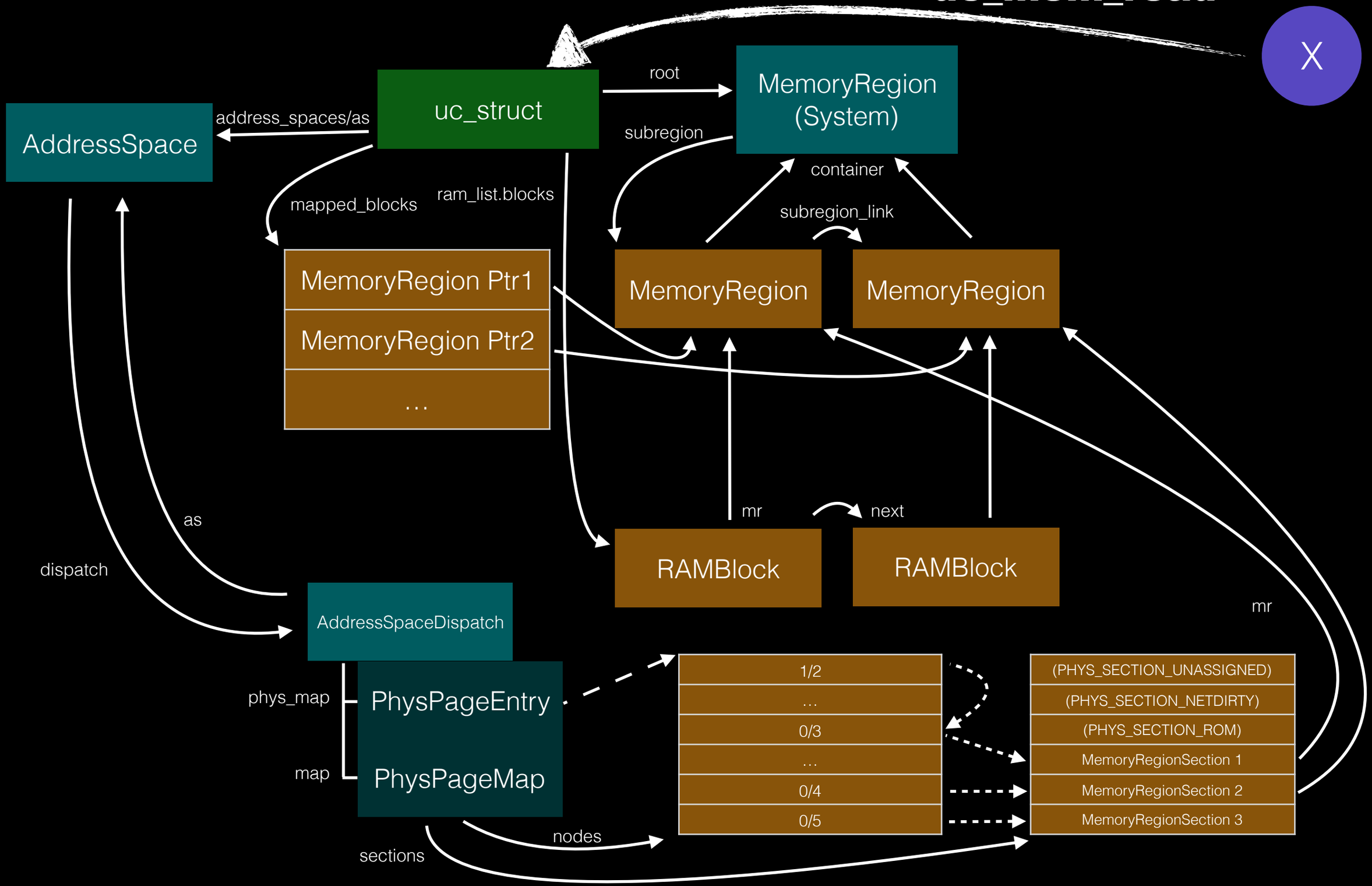
**qemu_get_ram_ptr** then gets block->host, add offset within RAMblock to it, and returns as backing mem ptr

```c
void *qemu_get_ram_ptr(struct uc_struct *uc, ram_addr_t addr)
{
    RAMBlock *block = qemu_get_ram_block(uc, addr);

    return block->host + (addr - block->offset);
}

static RAMBlock *qemu_get_ram_block(struct uc_struct *uc, ram_addr_t addr)
{
    RAMBlock *block;

    /* The list is protected by the iothread lock here.  */
    block = uc->ram_list.mru_block;
    if (block && addr - block->offset < block->length) {
        goto found;
    }
    QTAILQ_FOREACH(block, &uc->ram_list.blocks, next) {
        if (addr - block->offset < block->length) {
            goto found;
        }
    }

    fprintf(stderr, "Bad ram offset %" PRIx64 "\n", (uint64_t)addr);
    abort();

found:
    uc->ram_list.mru_block = block;
    return block;
}
```

# Walkthrough

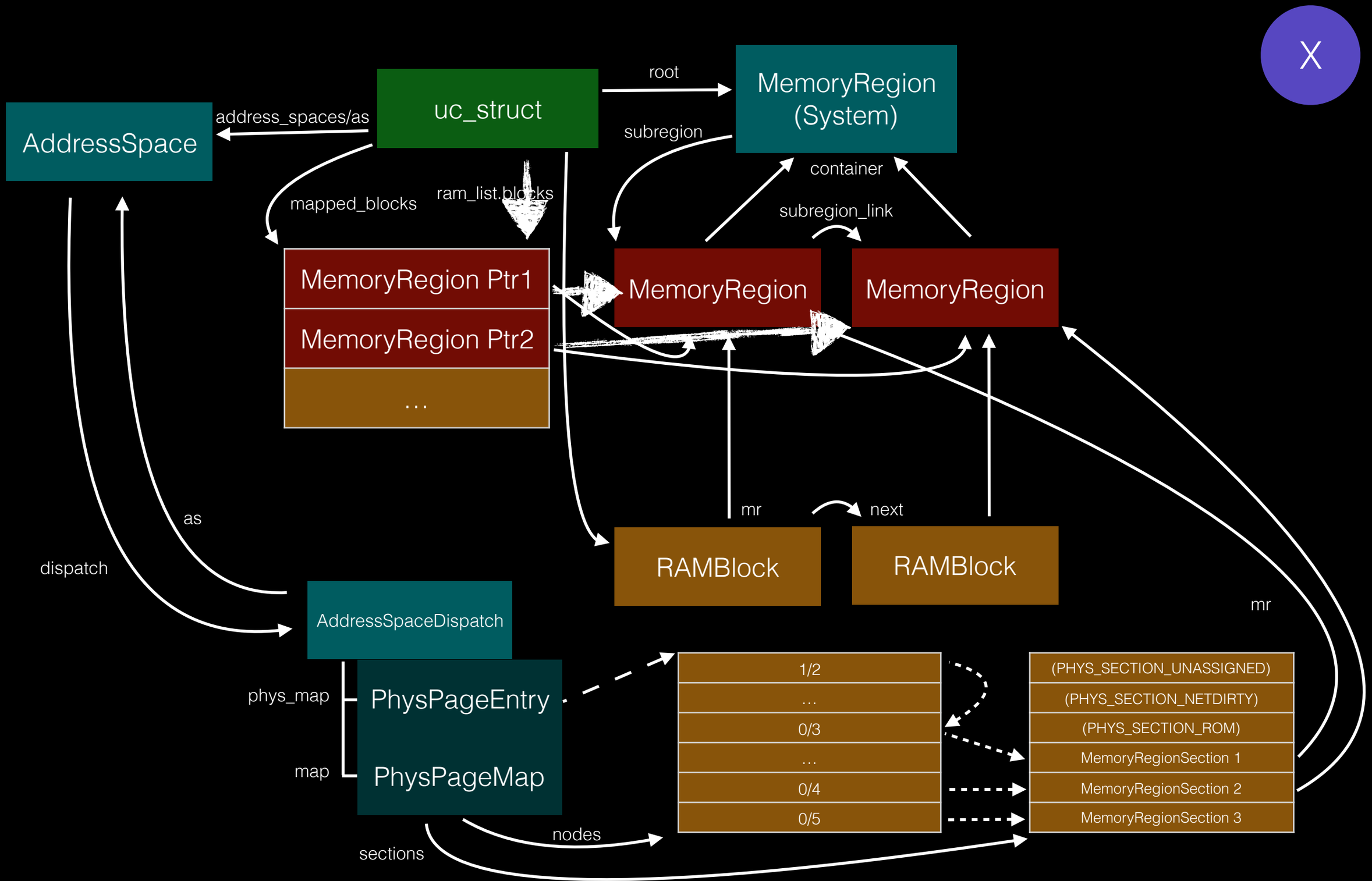
**uc_mem_read** from some addr range X
Note that FlatView/subpage is omitted from now on

**uc_mem_read**



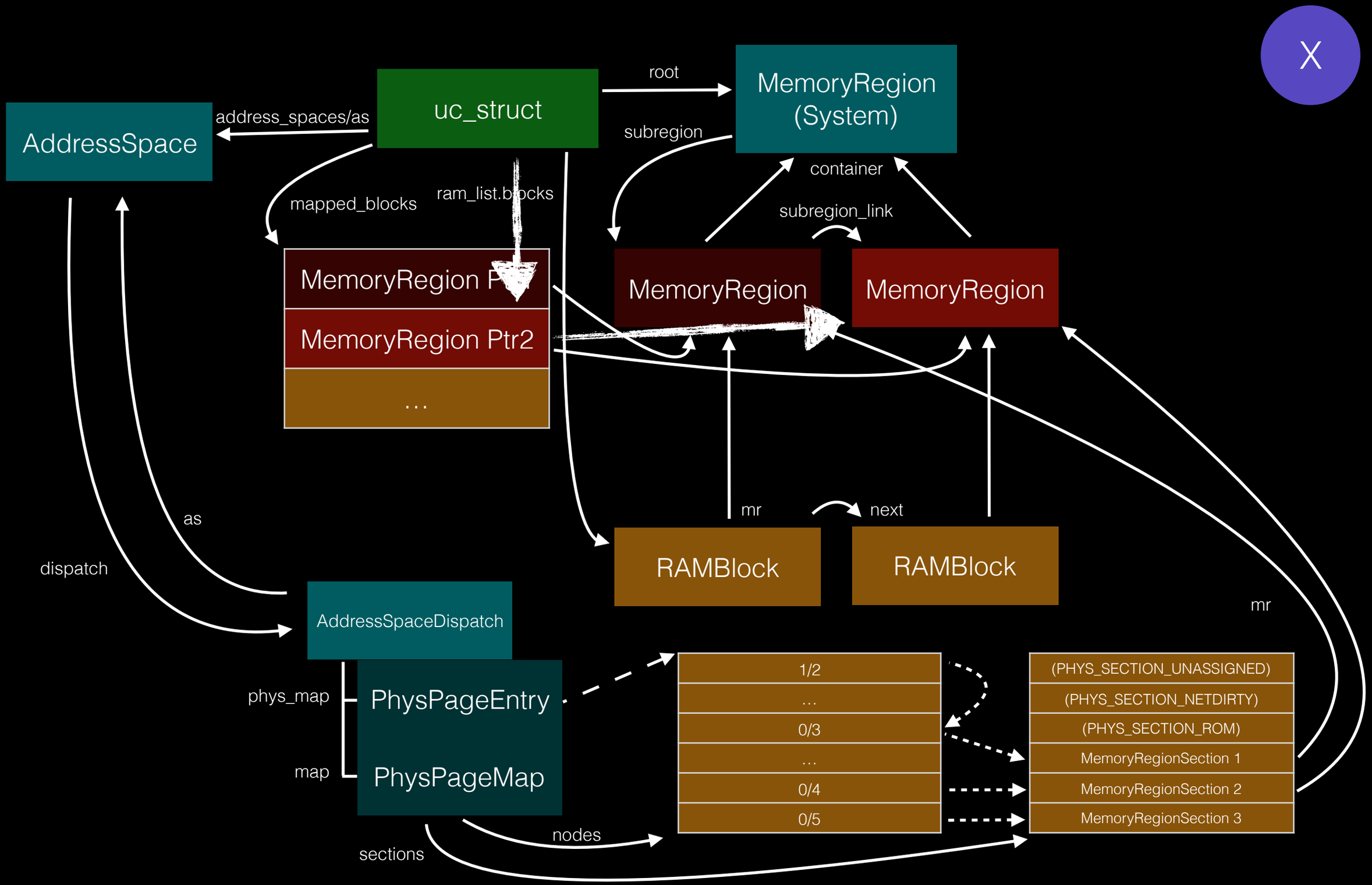
# Walkthrough

traverse mapped_blocks and confirm existence of entire X range



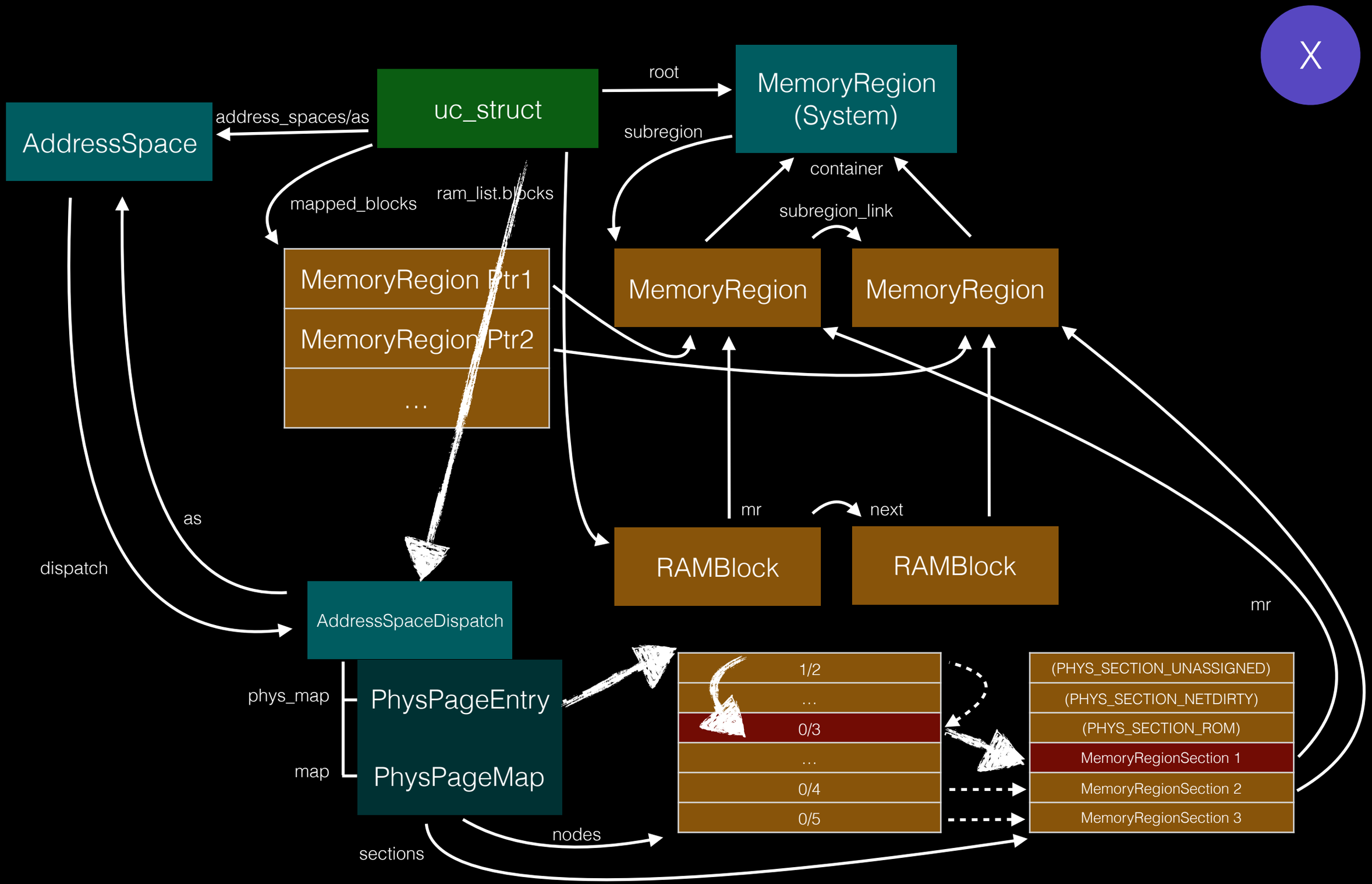
# Walkthrough

split the entire range X into smaller pieces that fit inside a single mr, and process in order



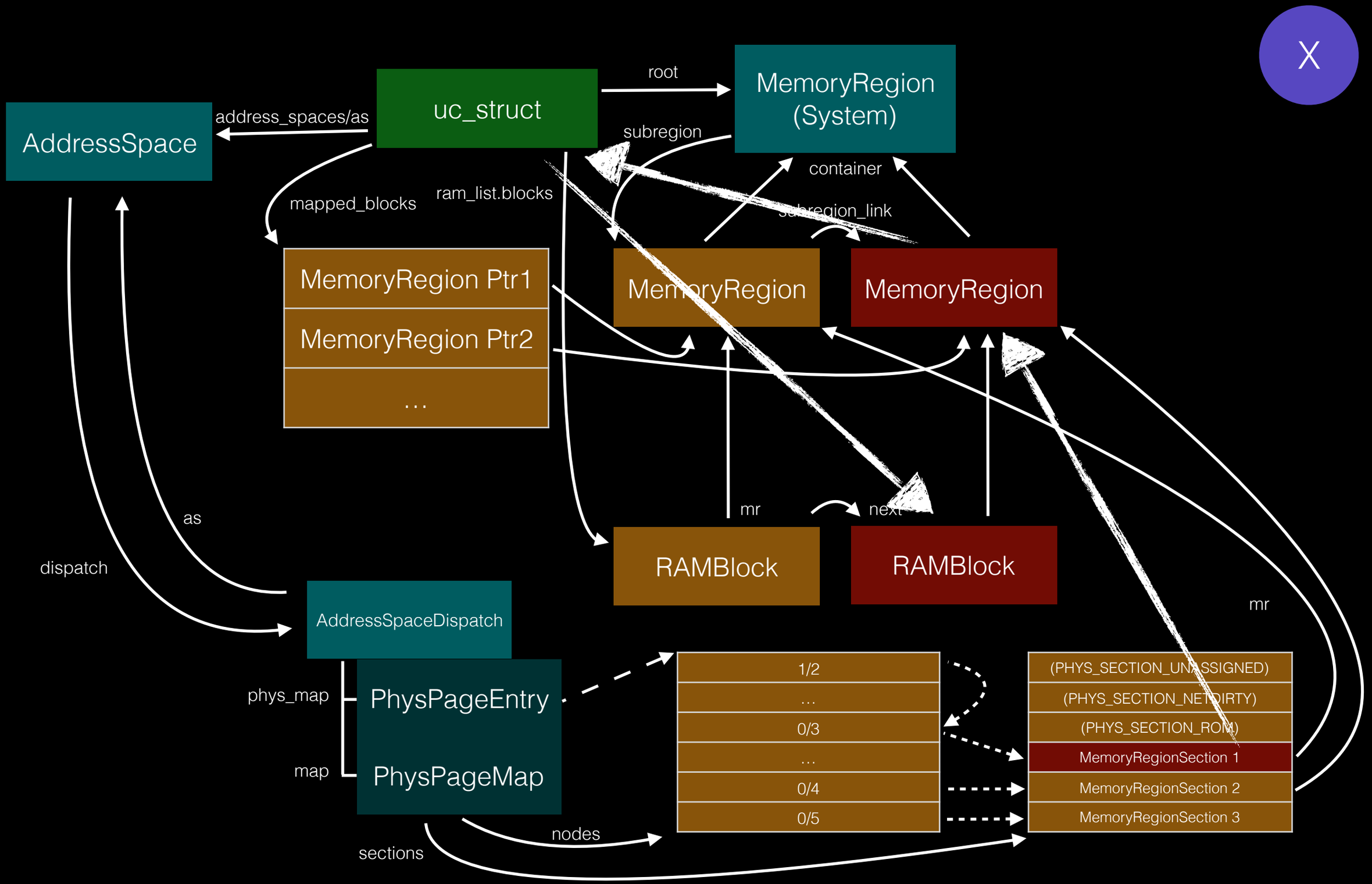
# Walkthrough

lookup corresponding section by walking PhysPageMap



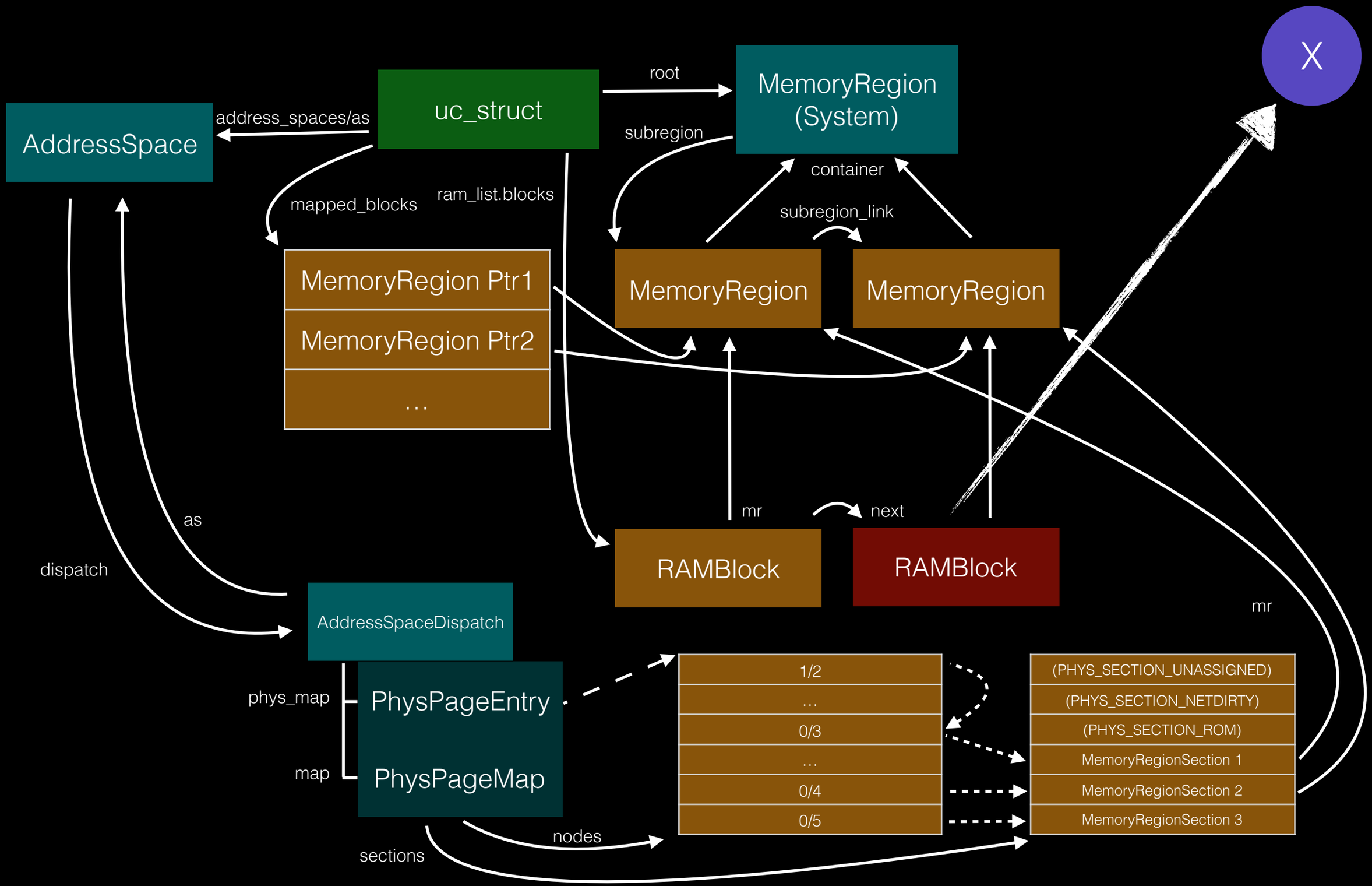
# Walkthrough

extract ram_addr from mr, and use it to lookup the corresponding RAMBlock



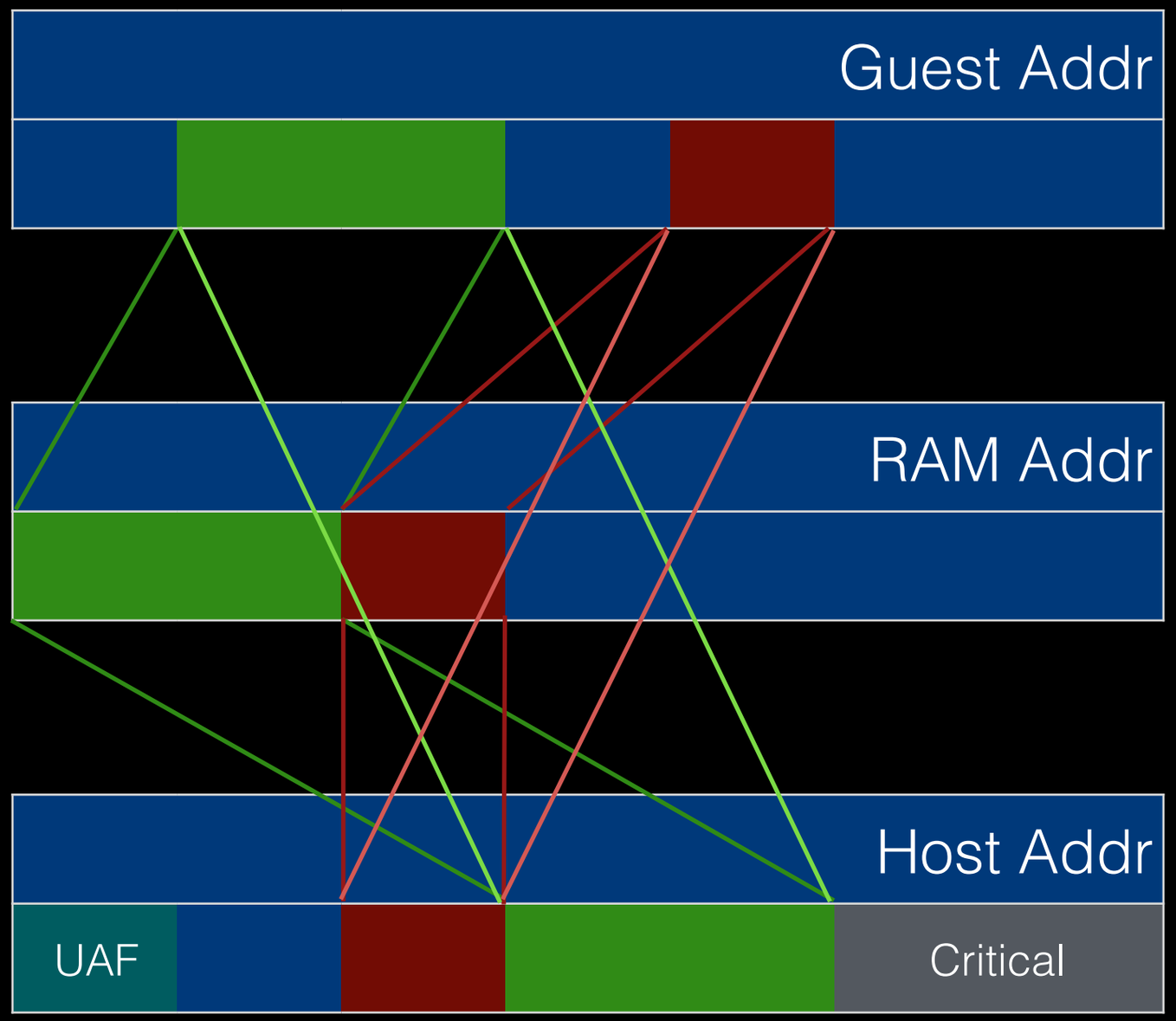
# Walkthrough

calculate backing mem address, and finally do **memcpy**



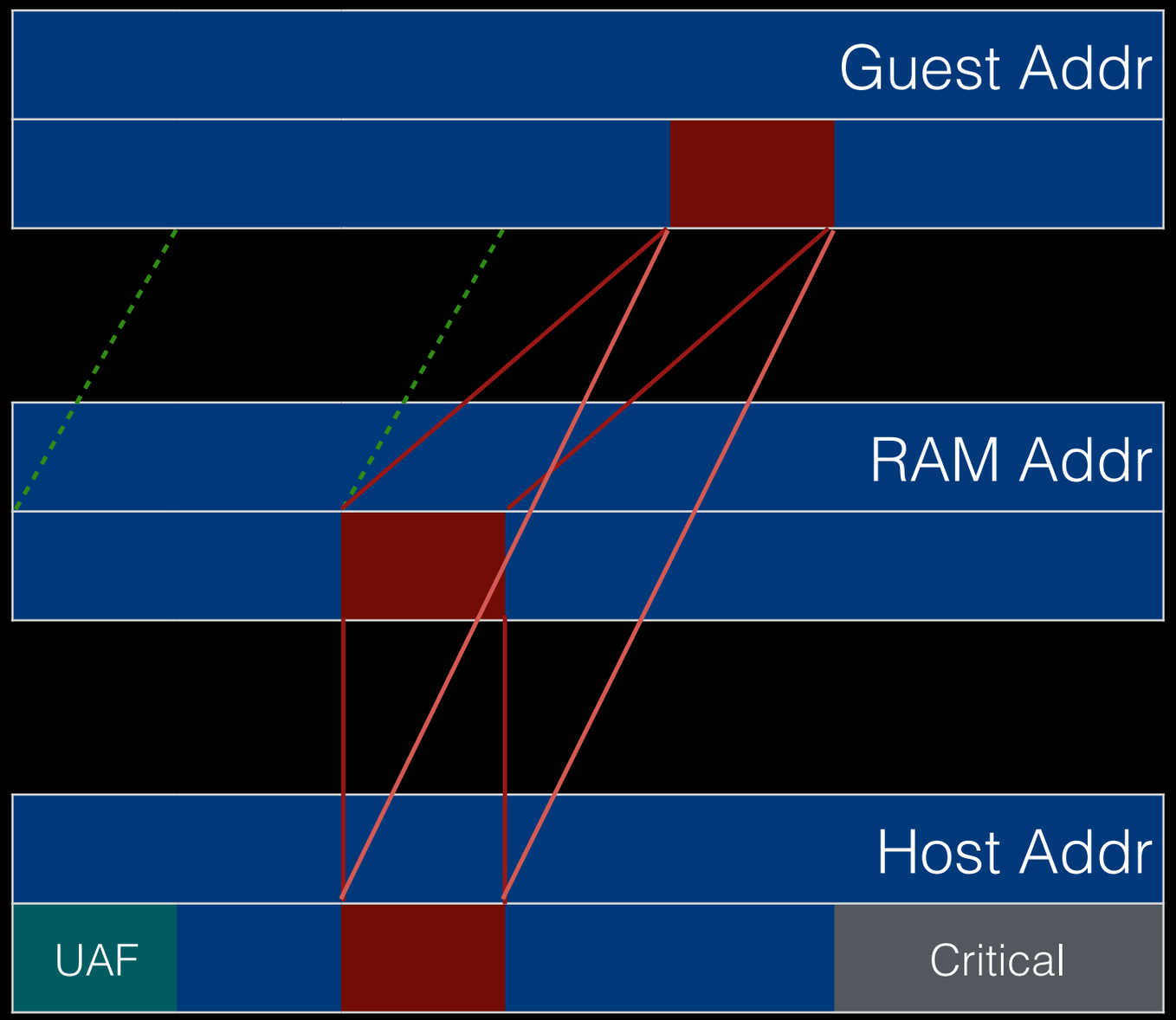
# Mapped_Blocks Exploit

we prepare a setup like this

# Mapped_Blocks Exploit

release Mem2




MemoryRegion UAF

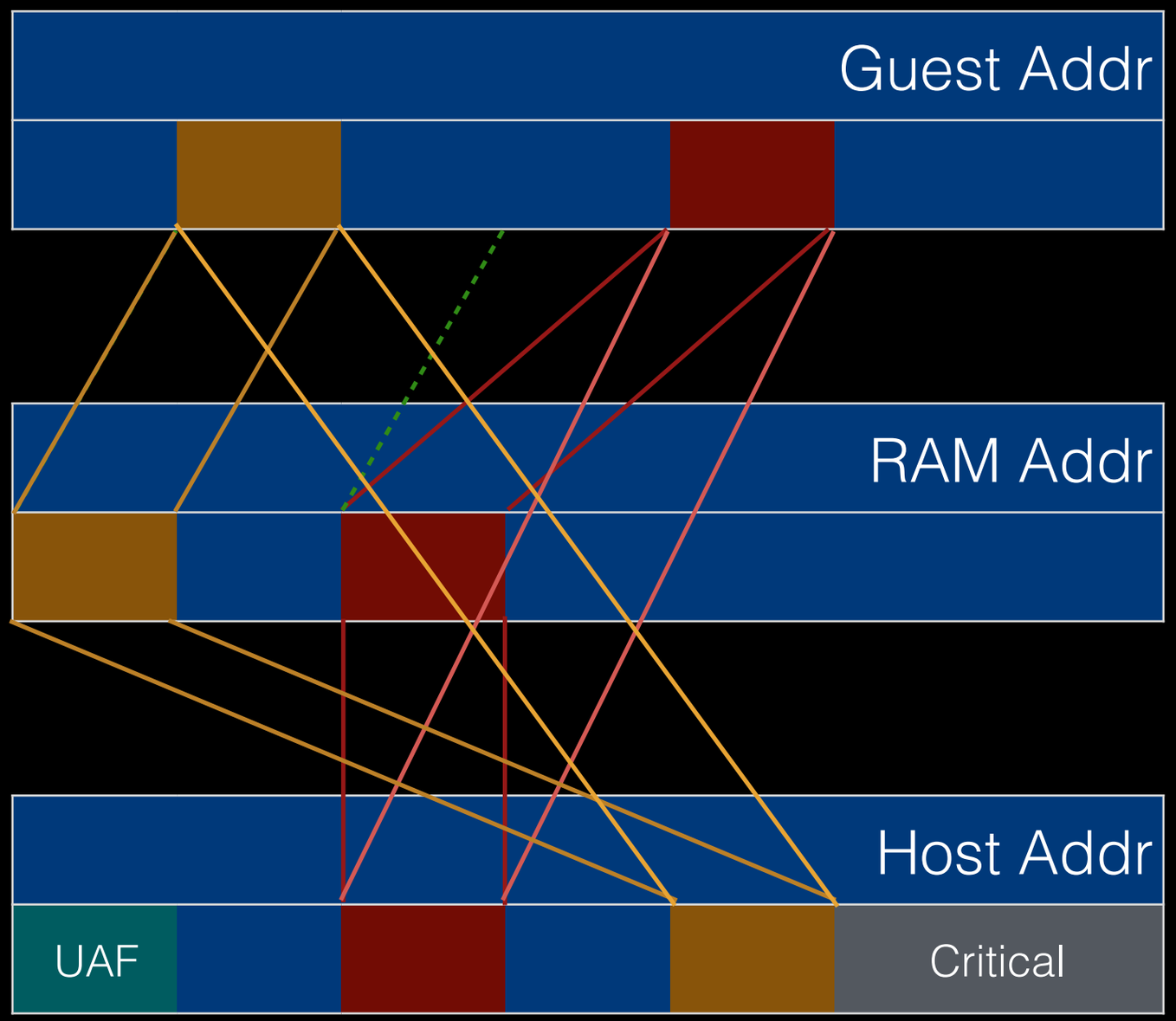
MemoryRegion Ptr1

MemoryRegion Ptr2

# Mapped_Blocks Exploit

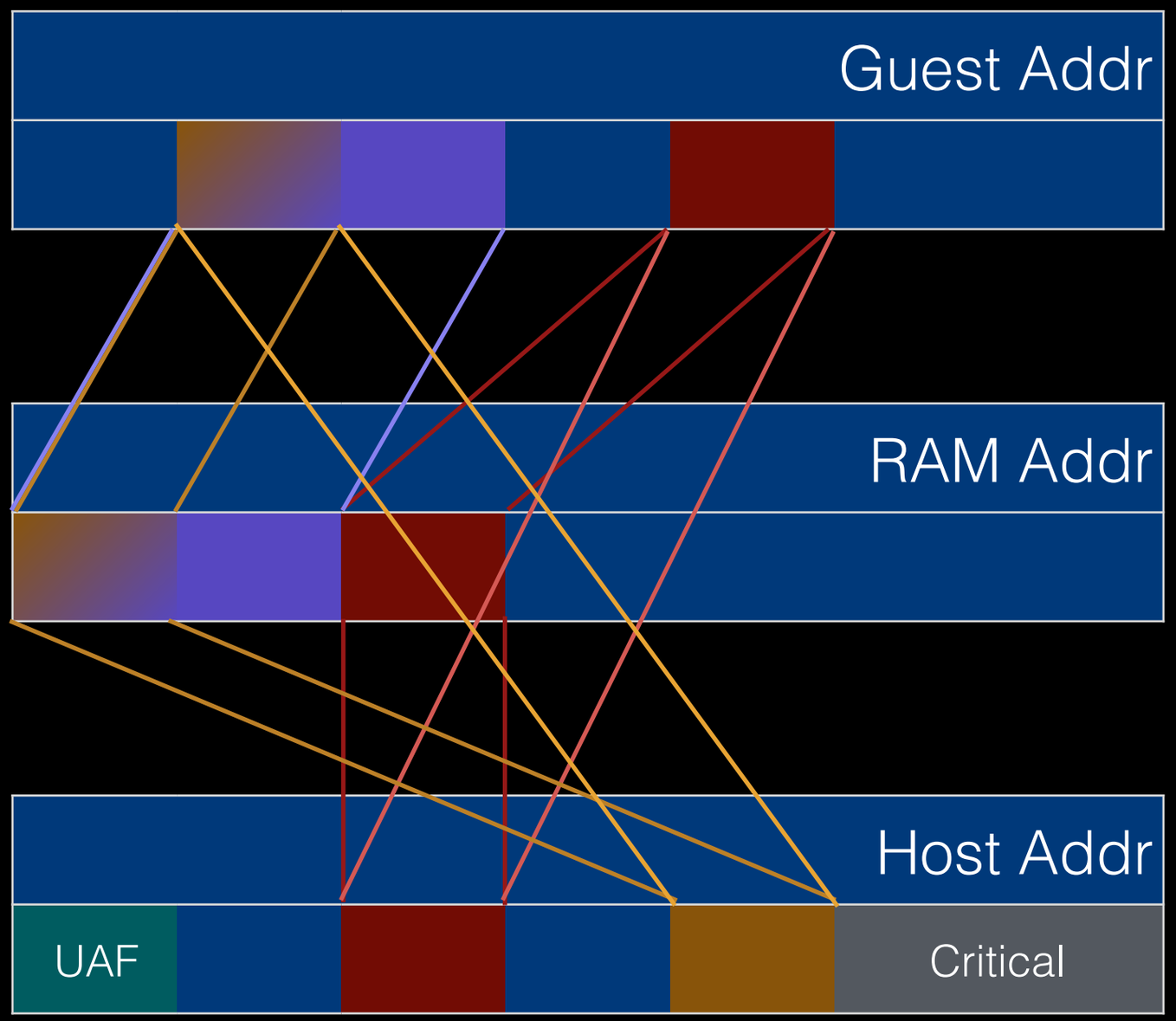
claim Mem3 at same GVA as Mem2

notice that we also want HVA of Mem3 to fall right before critical memory

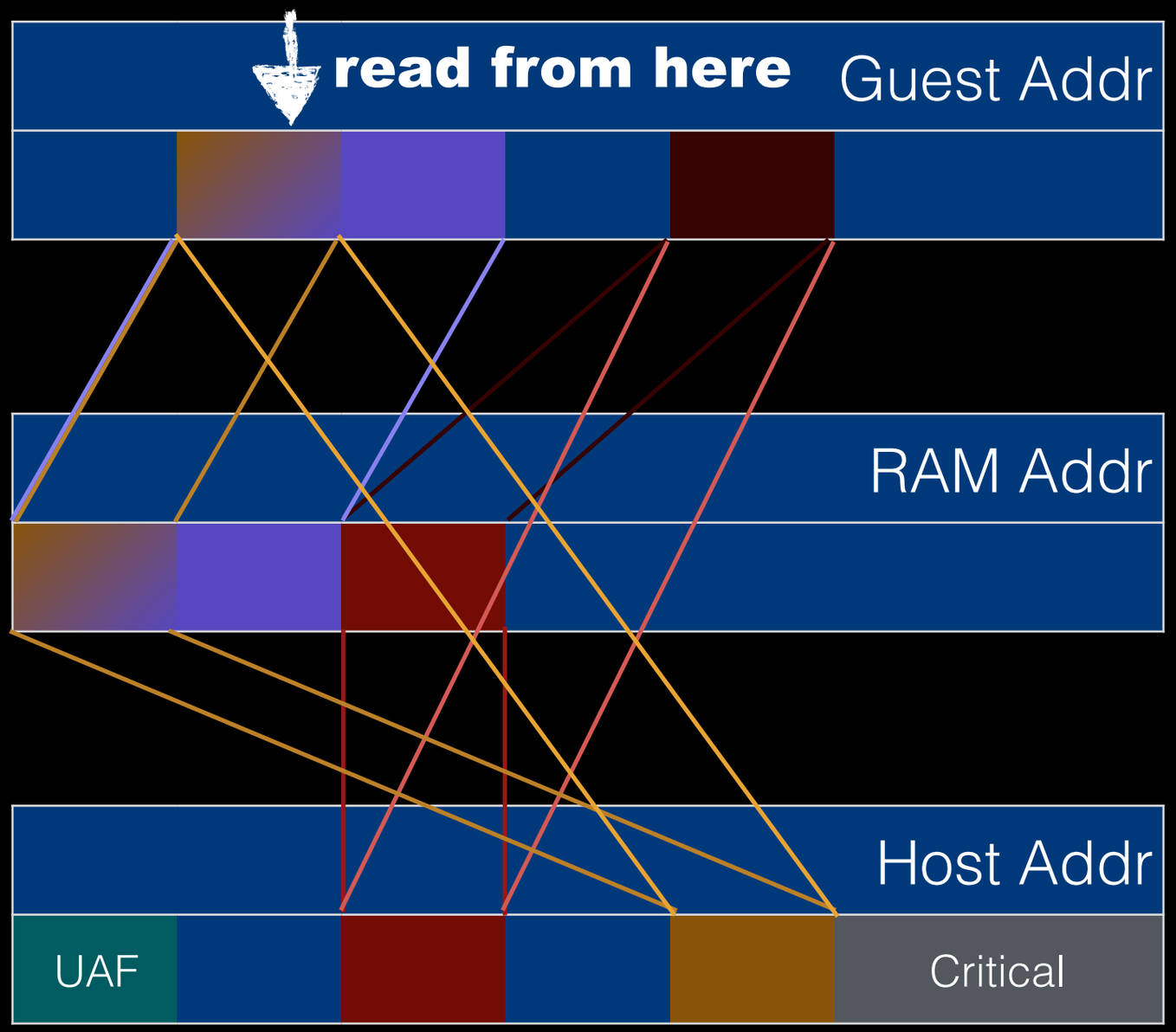
# Mapped_Blocks Exploit

modify MemoryRegion Ptr1 to MemoryRegion Ptr2

# Mapped_Blocks Exploit

Now we try to read 0x100 bytes starting from the last byte of Mem3

# Mapped_Blocks Exploit

action : **uc_mem_read**(x+0xfff,0x100)
check if the range in legal by fetching from mapped_blocks

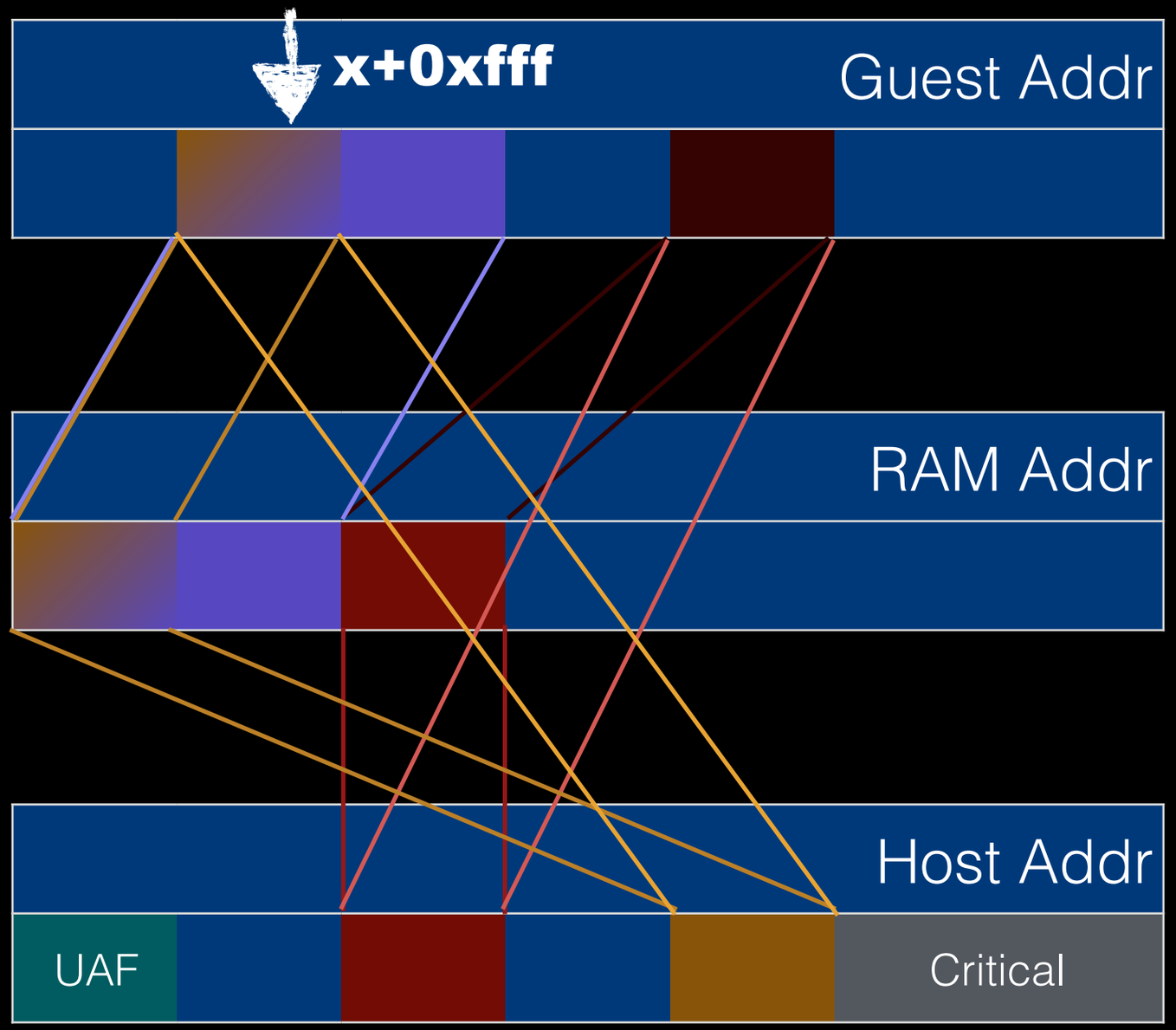
MemoryRegion
start : x
end : x+0x2000

**looks legit**

MemoryRegion UAF

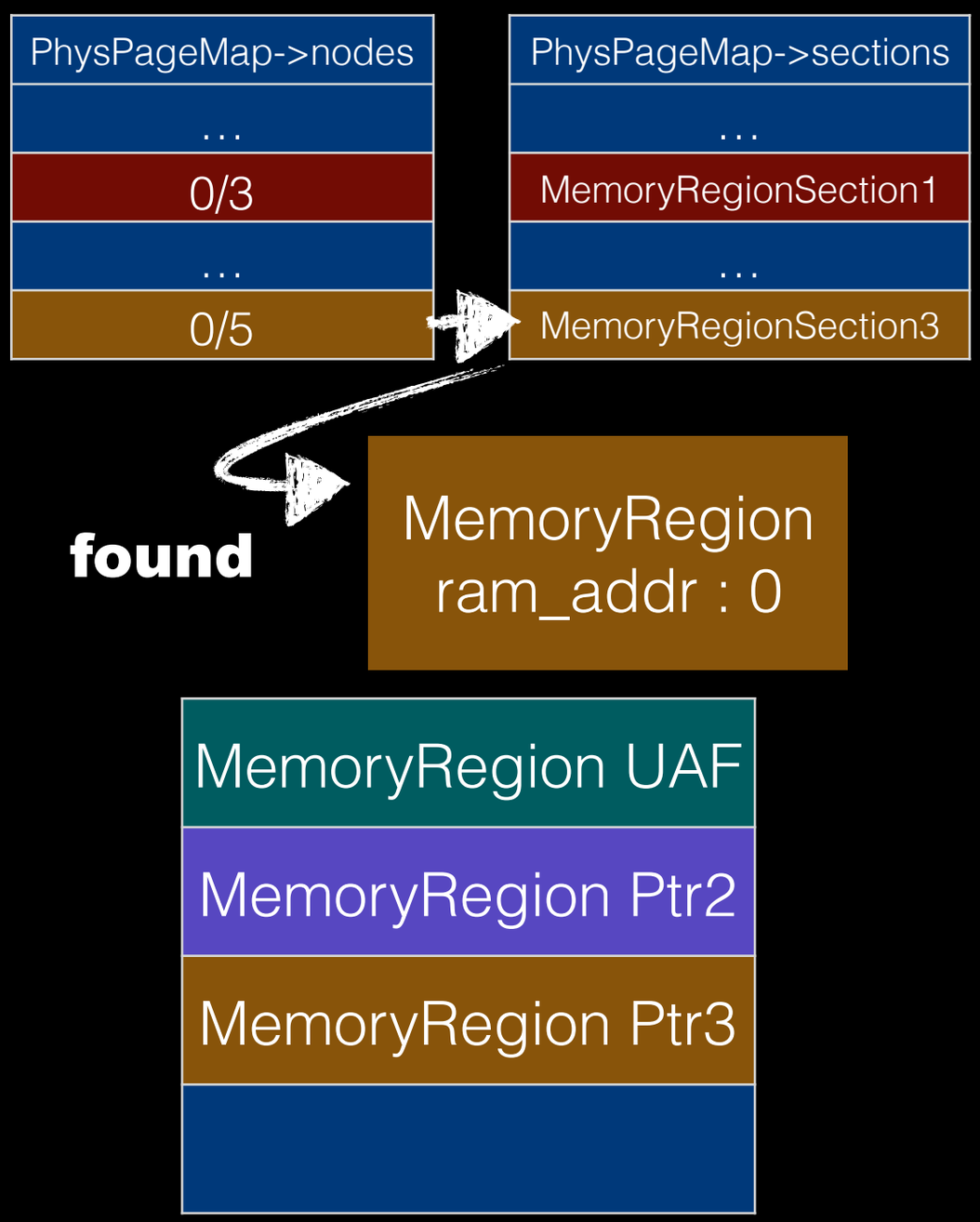
MemoryRegion Ptr2

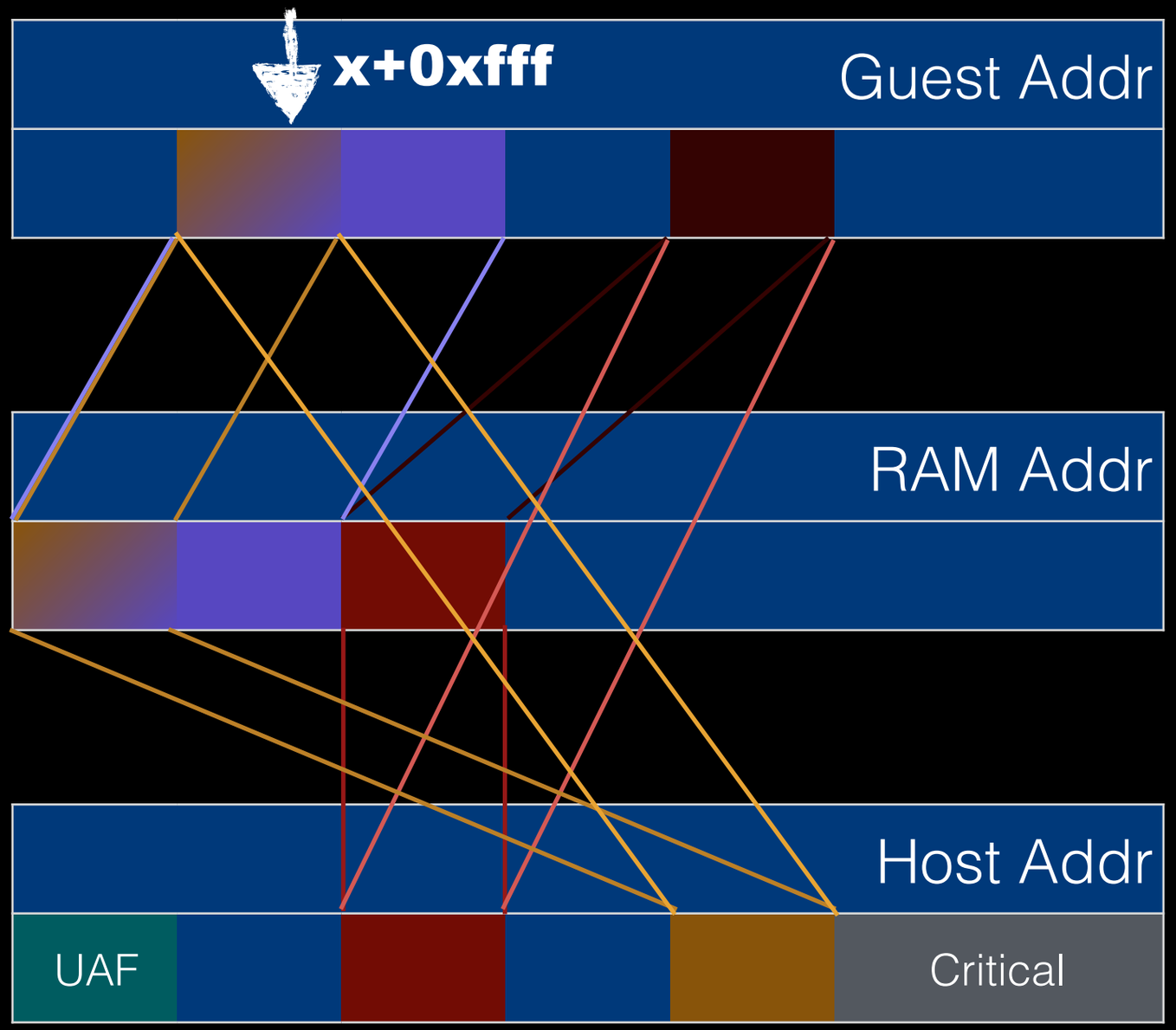
MemoryRegion Ptr3



# Mapped_Blocks Exploit

action : **uc_mem_read**(x+0xfff,0x100)
walk PhysPageMap for target mr, notice that MemoryRegion2 is forcefully added and does not exist in page table



# Mapped_Blocks Exploit

action : **uc_mem_read**(x+0xfff,0x100)
fetch RAMBlock for ram_addr+offset,
and calculate host+offset

found

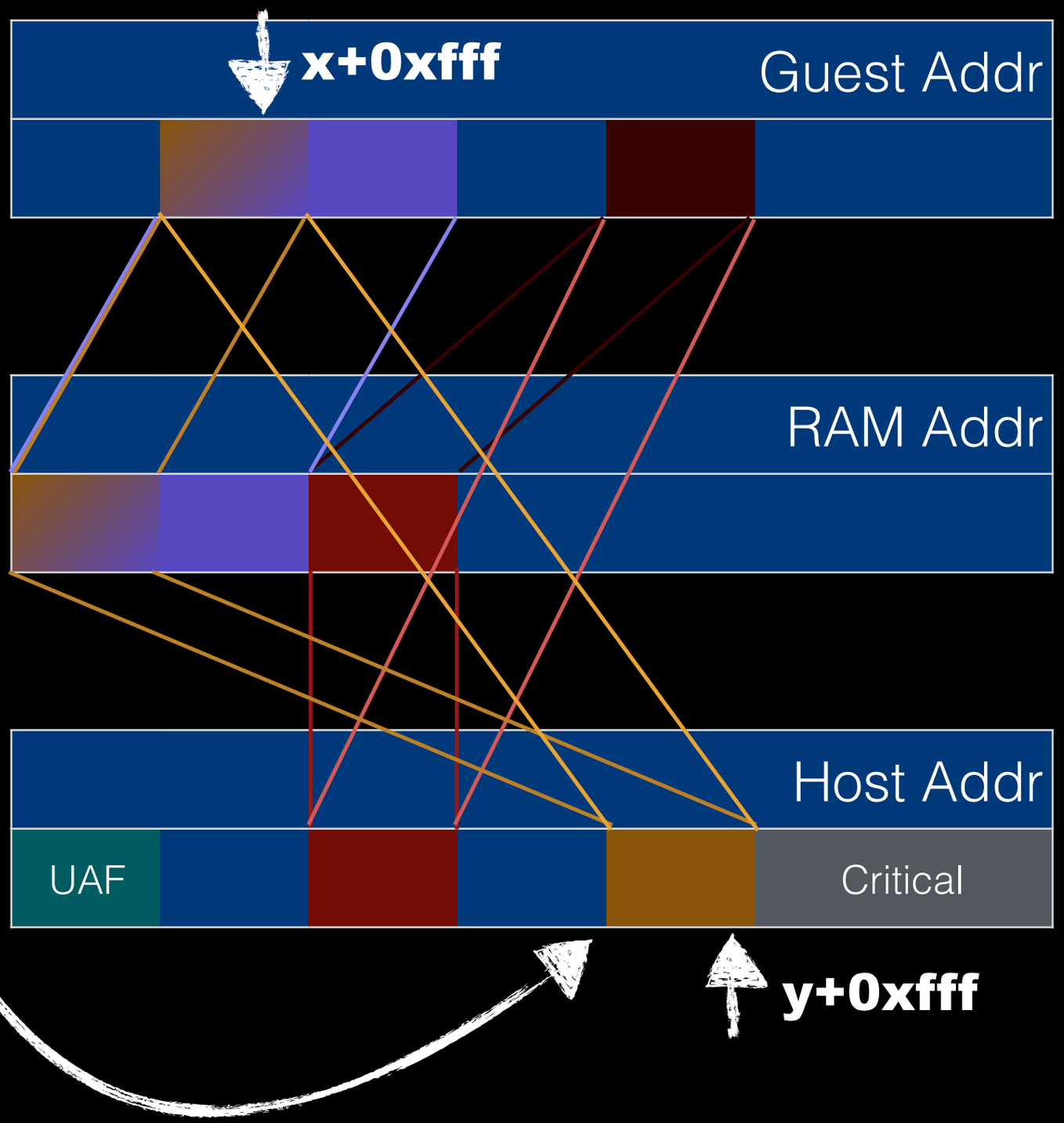
MemoryRegion
ram_addr : 0

RAMBlock
ram_addr : 0
size : 0x1000
host : y

MemoryRegion UAF

MemoryRegion Ptr2

MemoryRegion Ptr3



# Mapped_Blocks Exploit

action : **uc_mem_read**(x+0xfff,0x100)
**memcpy** and get OOB read

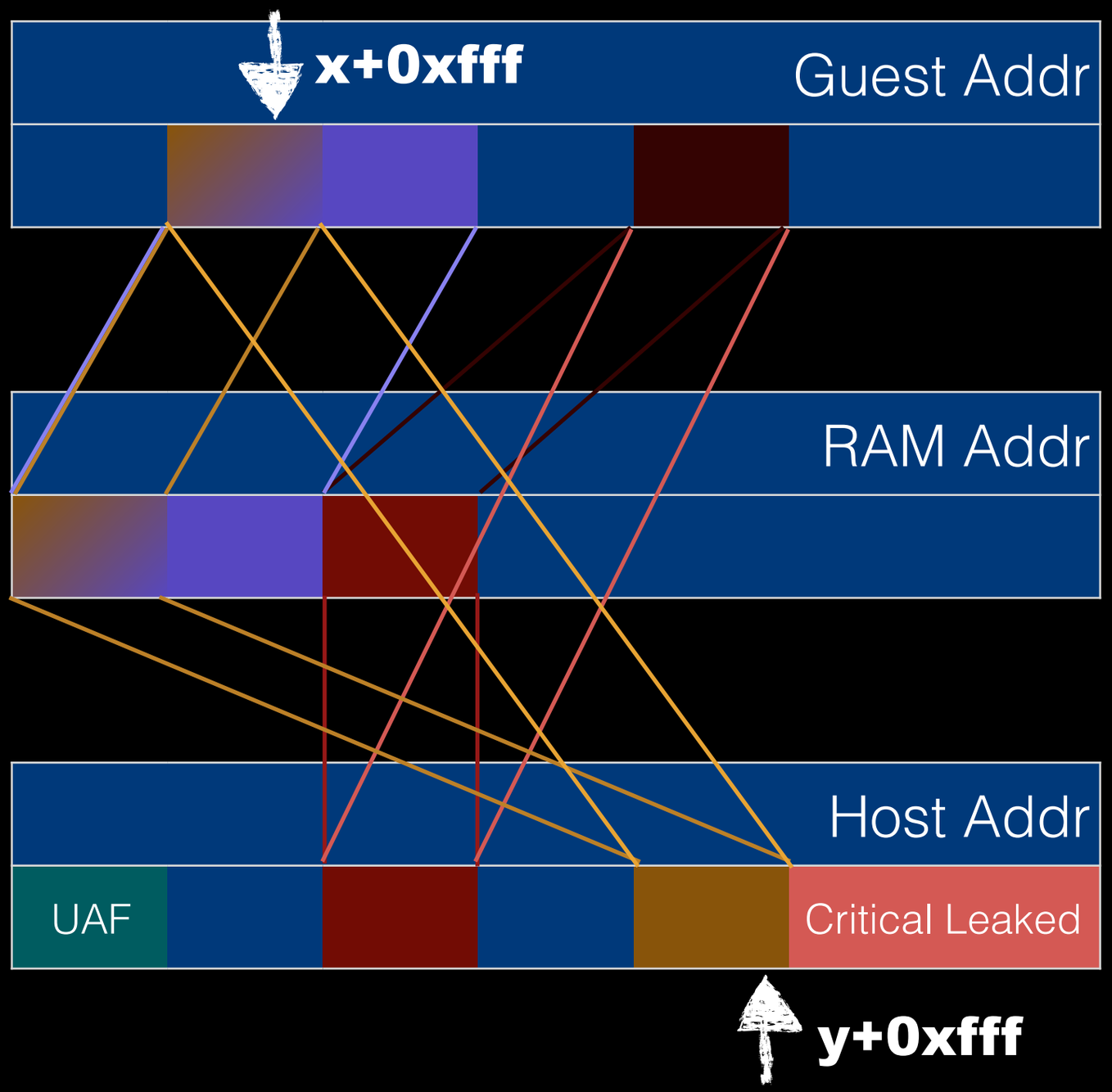
MemoryRegion
ram_addr : 0

RAMBlock
ram_addr : 0
size : 0x1000
host : y

MemoryRegion UAF

MemoryRegion Ptr2

MemoryRegion Ptr3

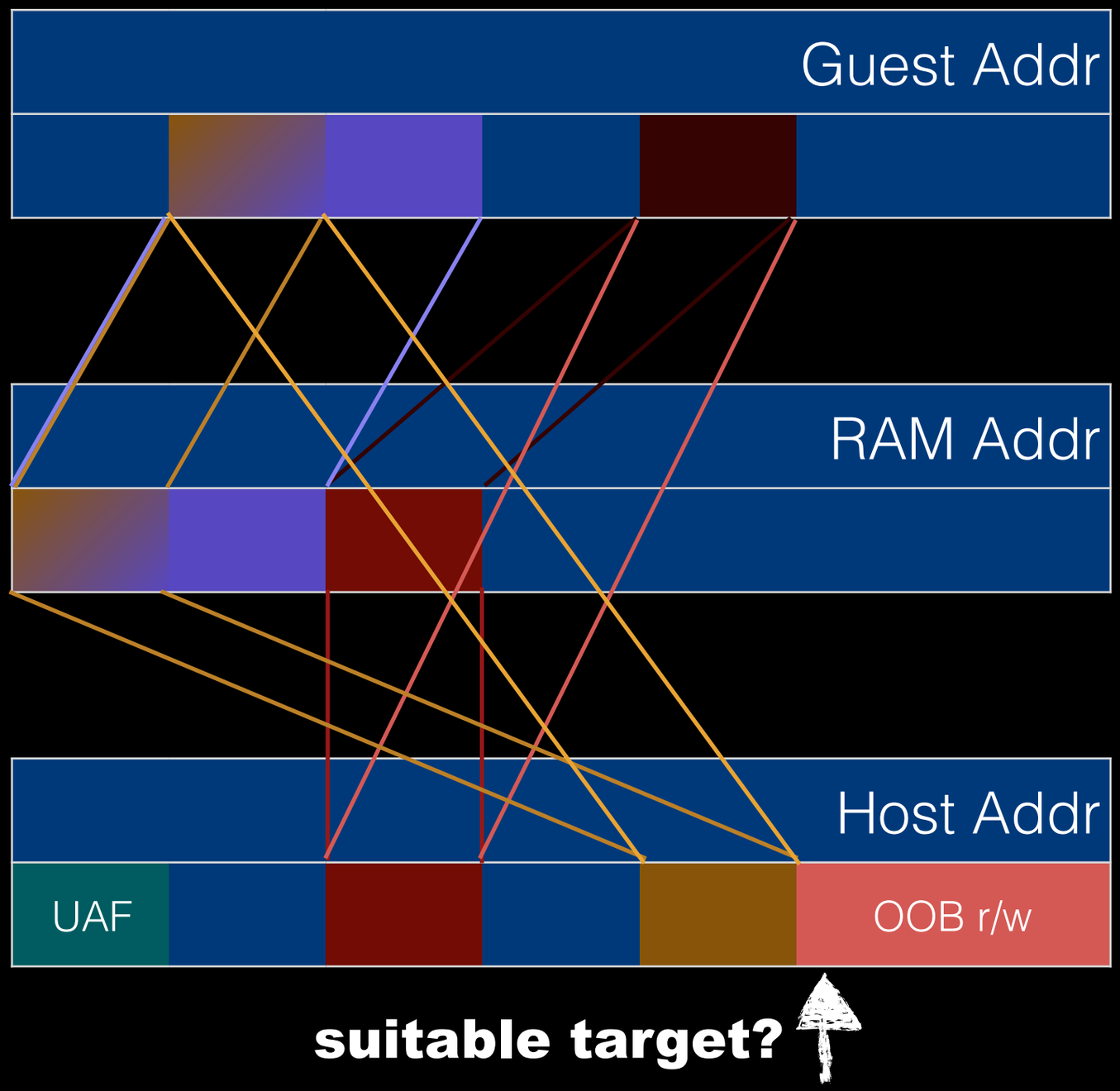


# Mapped_Blocks Exploit

OOB write is identical to OOB read
now we have a slightly stronger primitive,
next step is to find target critical mem
to attack

however, we need to first address something we've neglected up till now.

How did the initial UAF that allowed us to overwrite mapped_blocks work?



# Getting UAF

all 4 dynamic length arrays listed as attack candidate earlier follow similar pattern, they grow upon filled up to its capacity

glibc has a mmap_threshold, if we manage to push array size pass this value, the backing memory will be handled by mmap, which means potential overlap with UAF chunk

however pushing size pass threshold is not easy

```c
static uc_err mem_map(uc_engine *uc, uint64_t address,
        size_t size, uint32_t perms, MemoryRegion *block)
{
    MemoryRegion **regions;

    if (block == NULL)
        return UC_ERR_NOMEM;

    if ((uc->mapped_block_count & (MEM_BLOCK_INCR - 1)) == 0) {
        //time to grow
        regions = (MemoryRegion**)g_realloc(uc->mapped_blocks,
                sizeof(MemoryRegion*) *
                (uc->mapped_block_count + MEM_BLOCK_INCR));
        if (regions == NULL) {
            return UC_ERR_NOMEM;
        }
        uc->mapped_blocks = regions;
    }

    uc->mapped_blocks[uc->mapped_block_count] = block;
    uc->mapped_block_count++;

    return UC_ERR_OK;
}
```

# Getting UAF

first, the default mmap_threshold is 0x21000, which requires 0x4200 mmaped_blocks entries to reach

not doable in reasonable time since **uc_mem_map/uc_mem_unmap** both include O(n) operations, and have an extremely high constant

second, mmap_threshold grows as we start freeing large chunks, and can be up to 0x2000000, which makes it even harder to reach

```c
#ifndef DEFAULT_MMAP_THRESHOLD_MIN
#define DEFAULT_MMAP_THRESHOLD_MIN (128 * 1024)
#endif

#ifndef DEFAULT_MMAP_THRESHOLD_MAX
  /* For 32-bit platforms we cannot increase the maximum mmap
     threshold much because it is also the minimum value for the
     maximum heap size and its alignment.  Going above 512k (i.e., 1M
     for new heaps) wastes too much address space.  */
# if __WORDSIZE == 32
#  define DEFAULT_MMAP_THRESHOLD_MAX (512 * 1024)
# else
#  define DEFAULT_MMAP_THRESHOLD_MAX (4 * 1024 * 1024 * sizeof(long))
# endif
#endif

static struct malloc_par mp_ =
{
  .top_pad = DEFAULT_TOP_PAD,
  .n_mmaps_max = DEFAULT_MMAP_MAX,
  .mmap_threshold = DEFAULT_MMAP_THRESHOLD,
  .trim_threshold = DEFAULT_TRIM_THRESHOLD,
#define NARENAS_FROM_NCORES(n) ((n) * (sizeof (long) == 4 ? 2 : 8))
  .arena_test = NARENAS_FROM_NCORES (1)
#if USE_TCACHE
  ,
  .tcache_count = TCACHE_FILL_COUNT,
  .tcache_bins = TCACHE_MAX_BINS,
  .tcache_max_bytes = tidx2usize (TCACHE_MAX_BINS-1),
  .tcache_unsorted_limit = 0 /* No limit.  */
#endif
};

void
__libc_free (void *mem)
{
  ...

  if (chunk_is_mmapped (p))                       /* release mmapped memory. */
    {
      /* See if the dynamic brk/mmap threshold needs adjusting.
     Dumped fake mmapped chunks do not affect the threshold.  */
      if (!mp_.no_dyn_threshold
          && chunksize_nomask (p) > mp_.mmap_threshold
          && chunksize_nomask (p) <= DEFAULT_MMAP_THRESHOLD_MAX)
        {
          mp_.mmap_threshold = chunksize (p);
          mp_.trim_threshold = 2 * mp_.mmap_threshold;
          LIBC_PROBE (memory_mallopt_free_dyn_thresholds, 2,
                      mp_.mmap_threshold, mp_.trim_threshold);
        }
      munmap_chunk (p);
    }

  ...
}
libc_hidden_def (__libc_free)
```

# Getting UAF

mapped_blocks is not exploitable unless we tune mmap_threshold through **mallopt**

Let’s review the other 3 candidates

size :

- FlatRange : 0x30
- MemoryRegionSection : 0x38
- Node : 0x800

map->nodes seems like a reasonable target

```c
static void flatview_insert(FlatView *view, unsigned pos, FlatRange *range)
{
    if (view->nr == view->nr_allocated) {
        view->nr_allocated = MAX(2 * view->nr, 10);
        view->ranges = g_realloc(view->ranges,
                                 view->nr_allocated * sizeof(*view->ranges));
    }
    memmove(view->ranges + pos + 1, view->ranges + pos,
            (view->nr - pos) * sizeof(FlatRange));
    view->ranges[pos] = *range;
    memory_region_ref(range->mr);
    ++view->nr;
}

static uint16_t phys_section_add(PhysPageMap *map,
        MemoryRegionSection *section)
{
    /* The physical section number is ORed with a page-aligned
     * pointer to produce the iotlb entries.  Thus it should
     * never overflow into the page-aligned value.
     */
    assert(map->sections_nb < TARGET_PAGE_SIZE);

    if (map->sections_nb == map->sections_nb_alloc) {
        map->sections_nb_alloc = MAX(map->sections_nb_alloc * 2, 16);
        map->sections = g_renew(MemoryRegionSection, map->sections,
                map->sections_nb_alloc);
    }
    map->sections[map->sections_nb] = *section;
    memory_region_ref(section->mr);
    return map->sections_nb++;
}

static void phys_map_node_reserve(PhysPageMap *map, unsigned nodes)
{
    if (map->nodes_nb + nodes > map->nodes_nb_alloc) {
        map->nodes_nb_alloc = MAX(map->nodes_nb_alloc * 2, 16);
        map->nodes_nb_alloc = MAX(map->nodes_nb_alloc, map->nodes_nb + nodes);
        map->nodes = g_renew(Node, map->nodes, map->nodes_nb_alloc);
    }
}
```

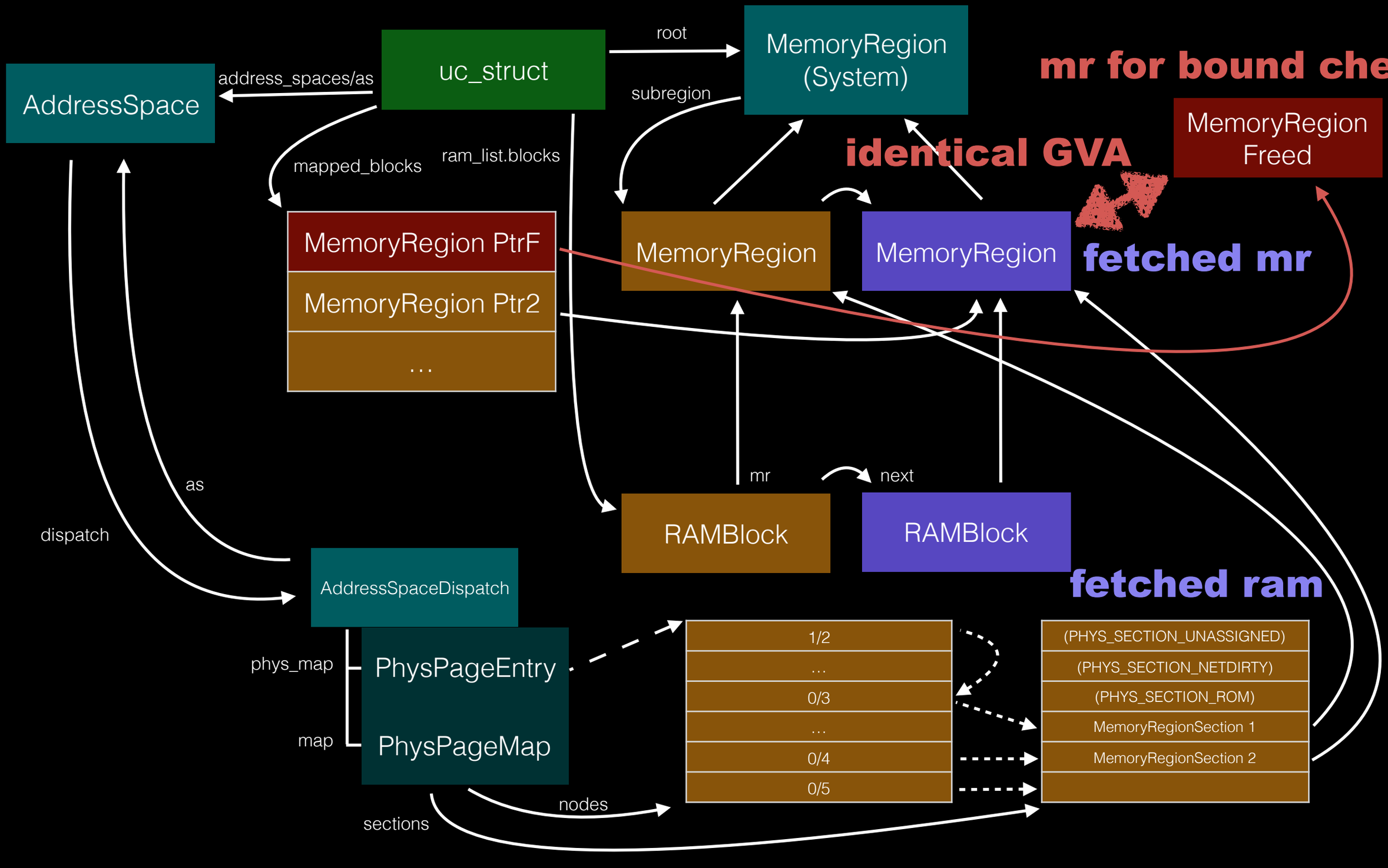
# map->nodes Exploit

before diving into the details, it is convenient to keep in mind that the overall target is similar, but entire setup is a lot more complex than previous case

we basically want to decouple the mr that gets its bound checked, and the mr used to fetch RAMblock

# Concept Recap

the previous attack on mapped_blocks
can be summarized as this
since bound check is done on a different
freed object, we can get OOB



# map->nodes
# Exploit

there are a few things to note here to help understanding the exploit

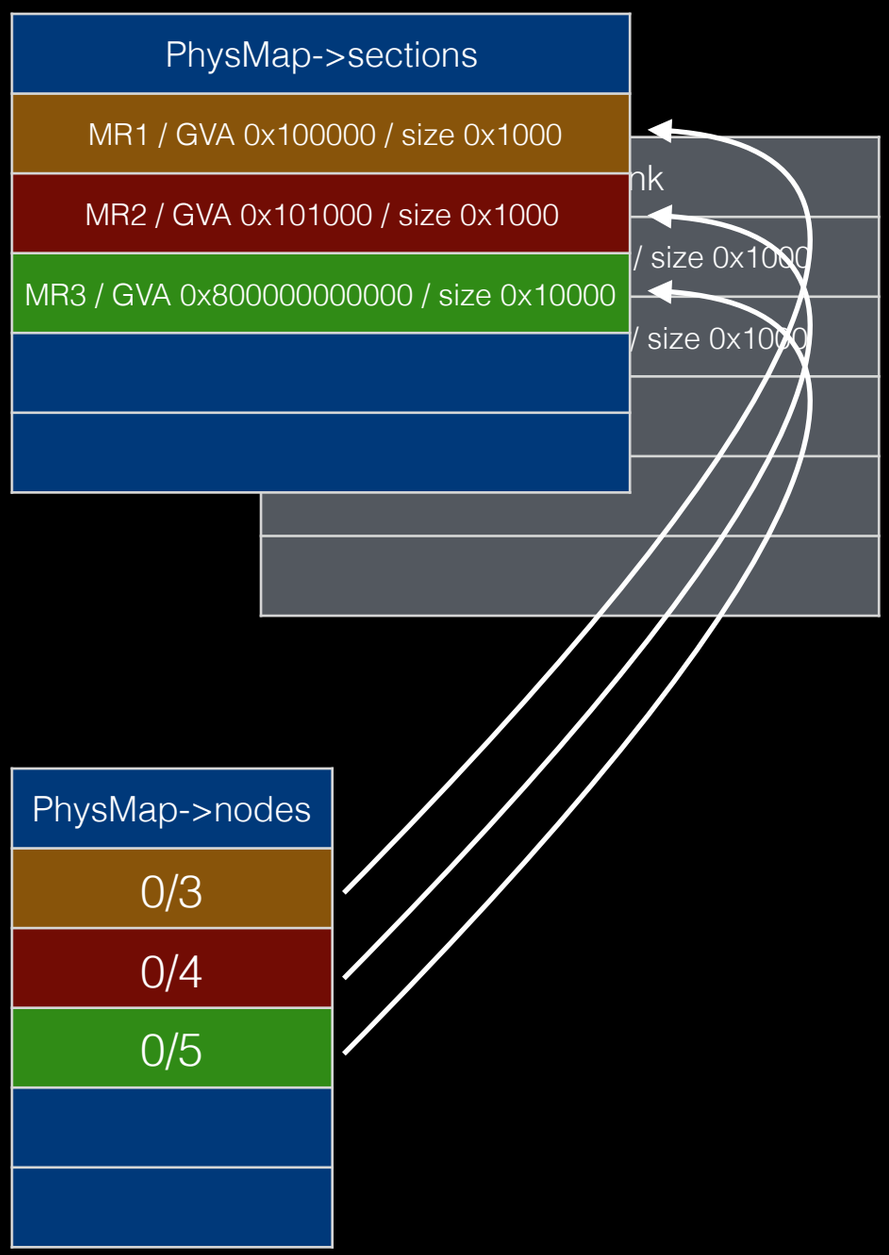
1. MemoryRegion struct size = 0xf0, and there are no other objects of same size in unicorn
   => we have full control over 0x100 tcache

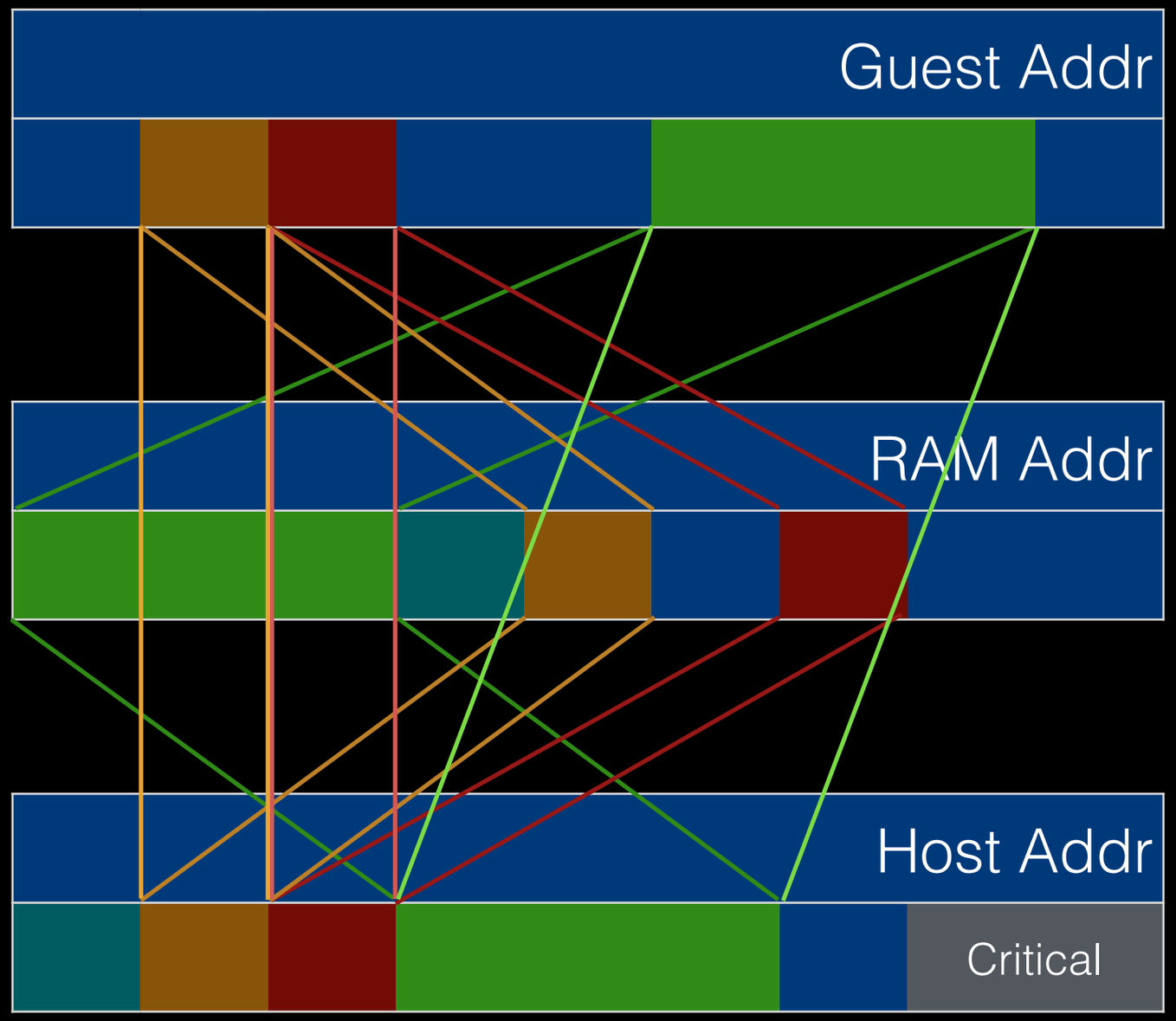
2. AddressSpaceDispatch.PhysPageMap->sections has a minimum of 0x10 entries, and only resizes buffer when we go above this number
   => static 0x38*0x10 array size if we manage things properly
   Once again, no other objects share the 0x390 tcache entry

3. AddressSpaceDispatch.PhysPageMap is re-constructed from scratch on each **uc_mem_mmap**, **uc_mem_unmap**
   => section array ptr to oscillate between two 0x390 chunks

# map->nodes Exploit

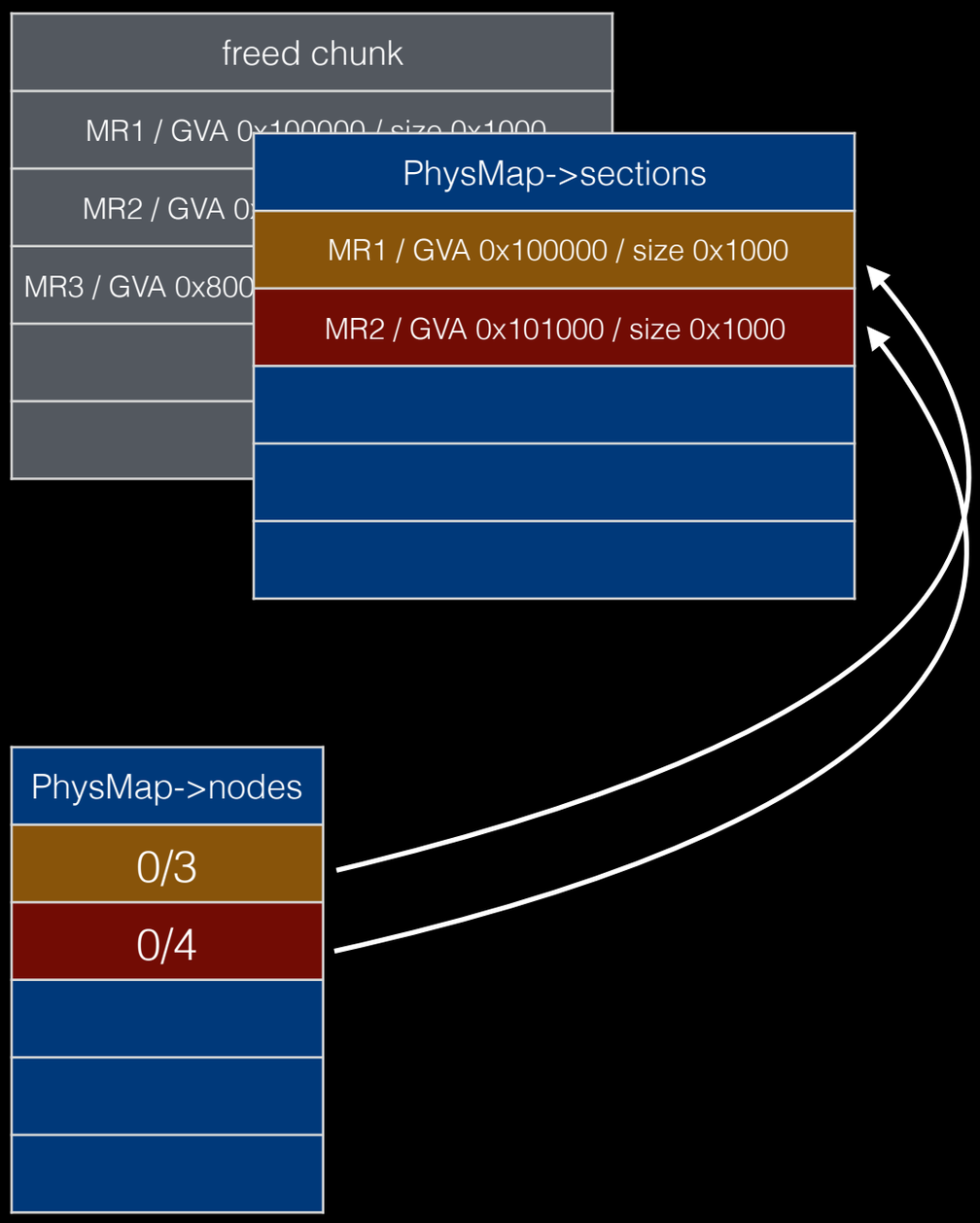
we prepare a setup like this, at this point UAF over Nodes array hasn't happened yet

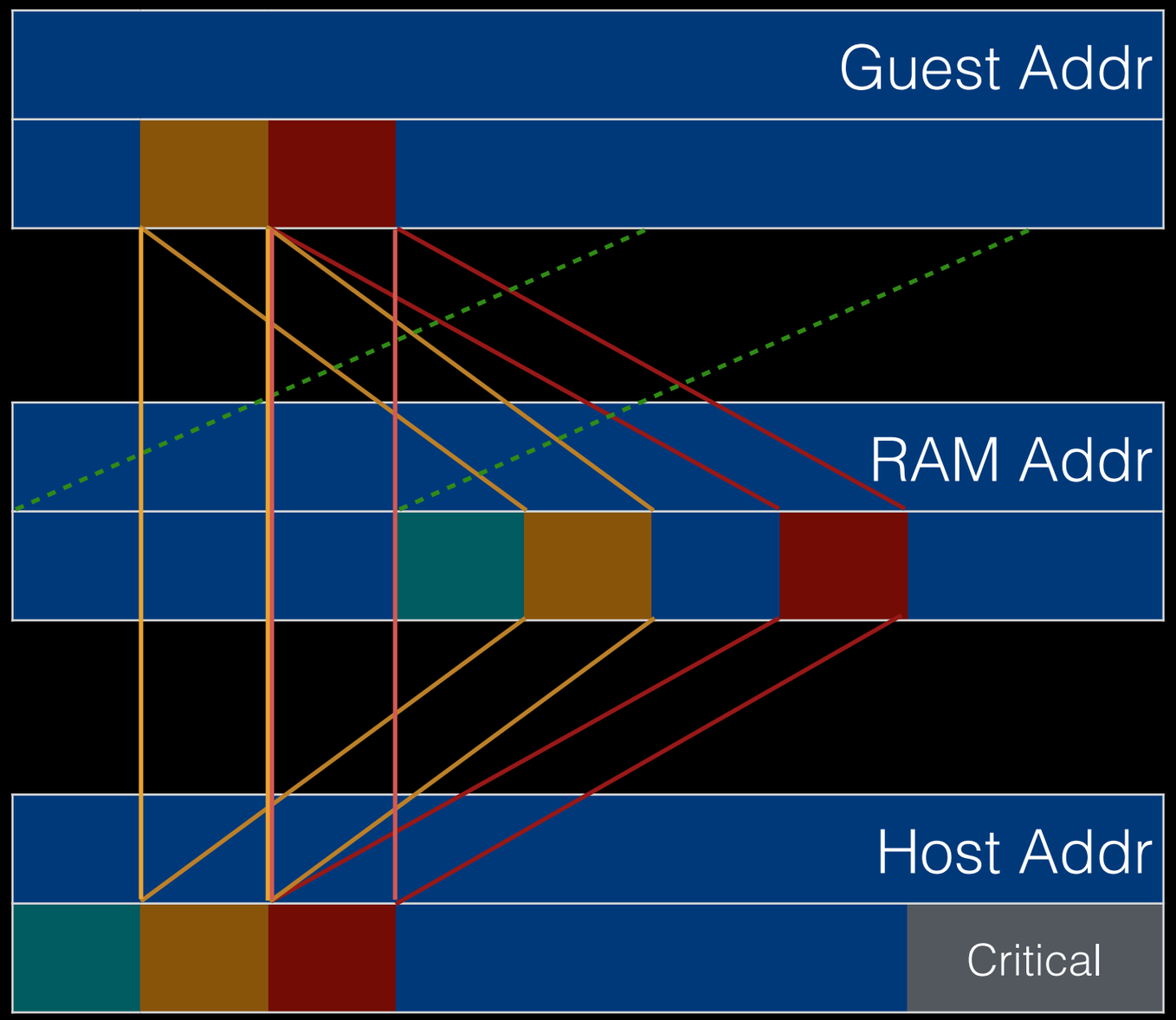


# map->nodes Exploit

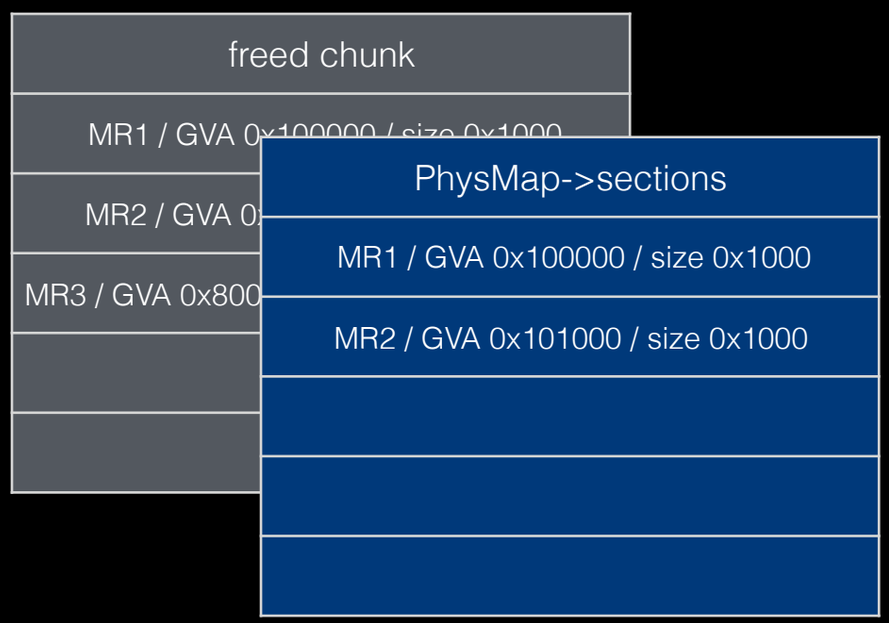
we first release MemoryRegion3
Note that how the sections array under control switches upon map/unmap

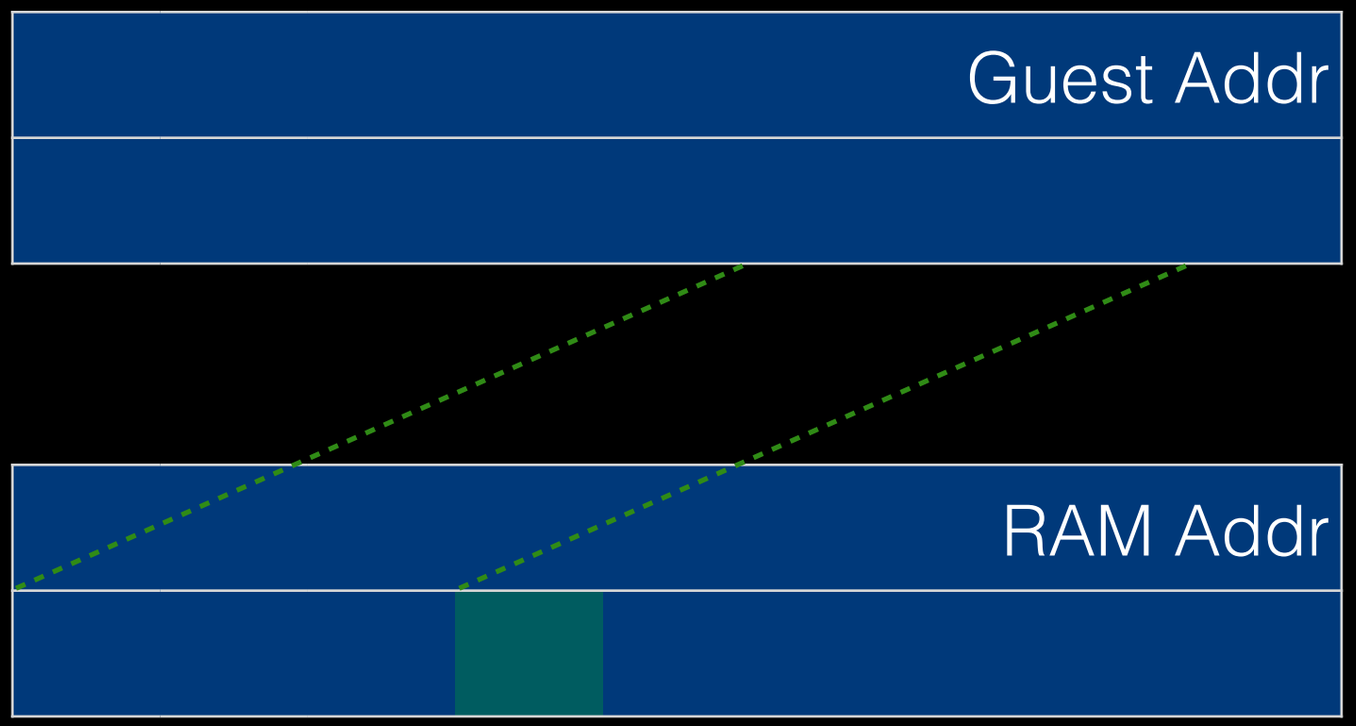


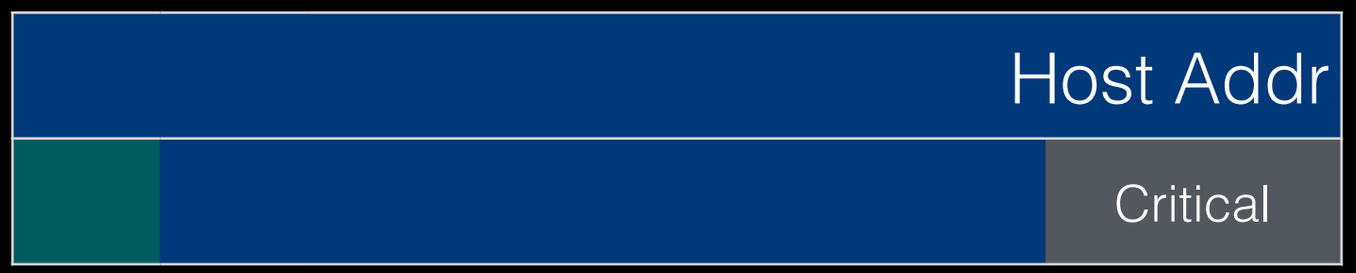
# map->nodes Exploit

the we release the MemoryRegion1/2
we can see that freed entry in sections
is not cleared if it is the last entry



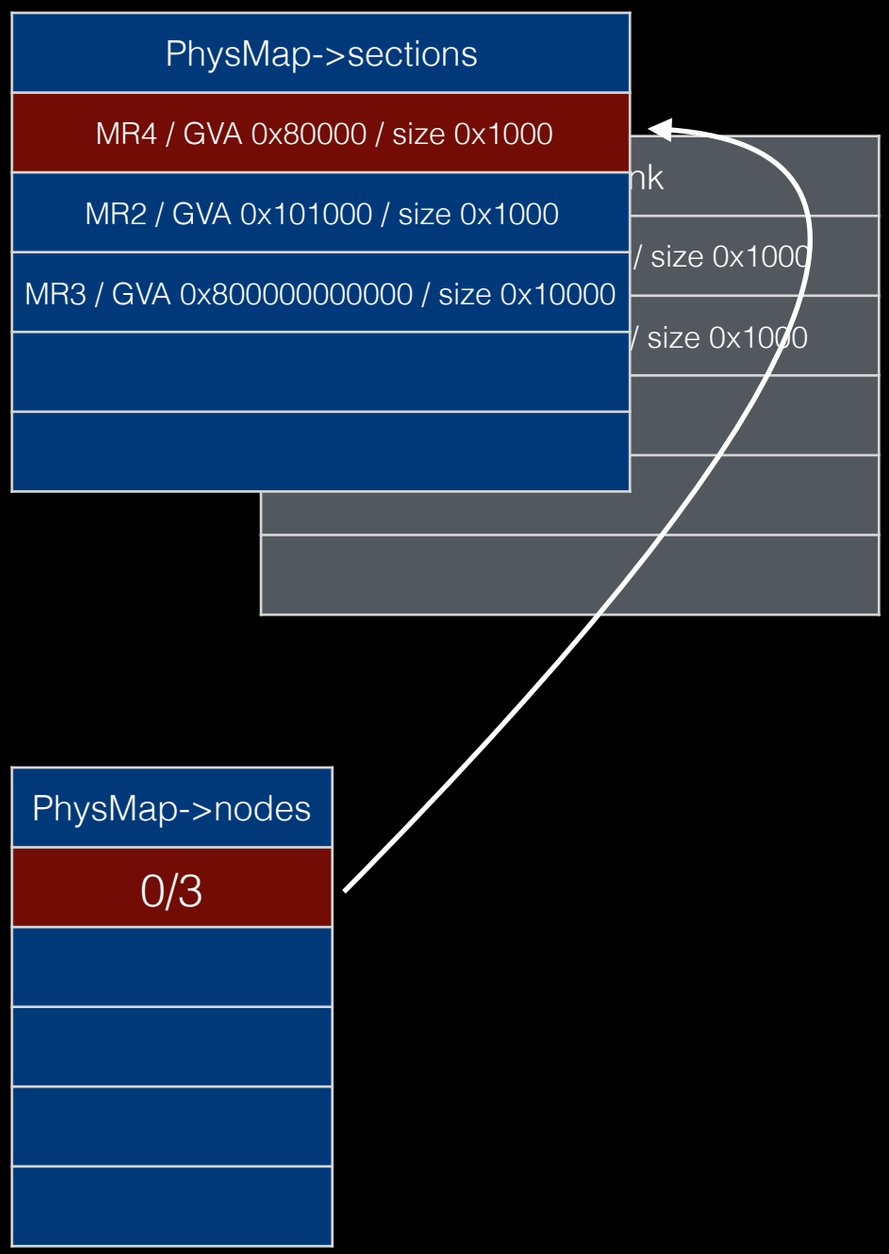



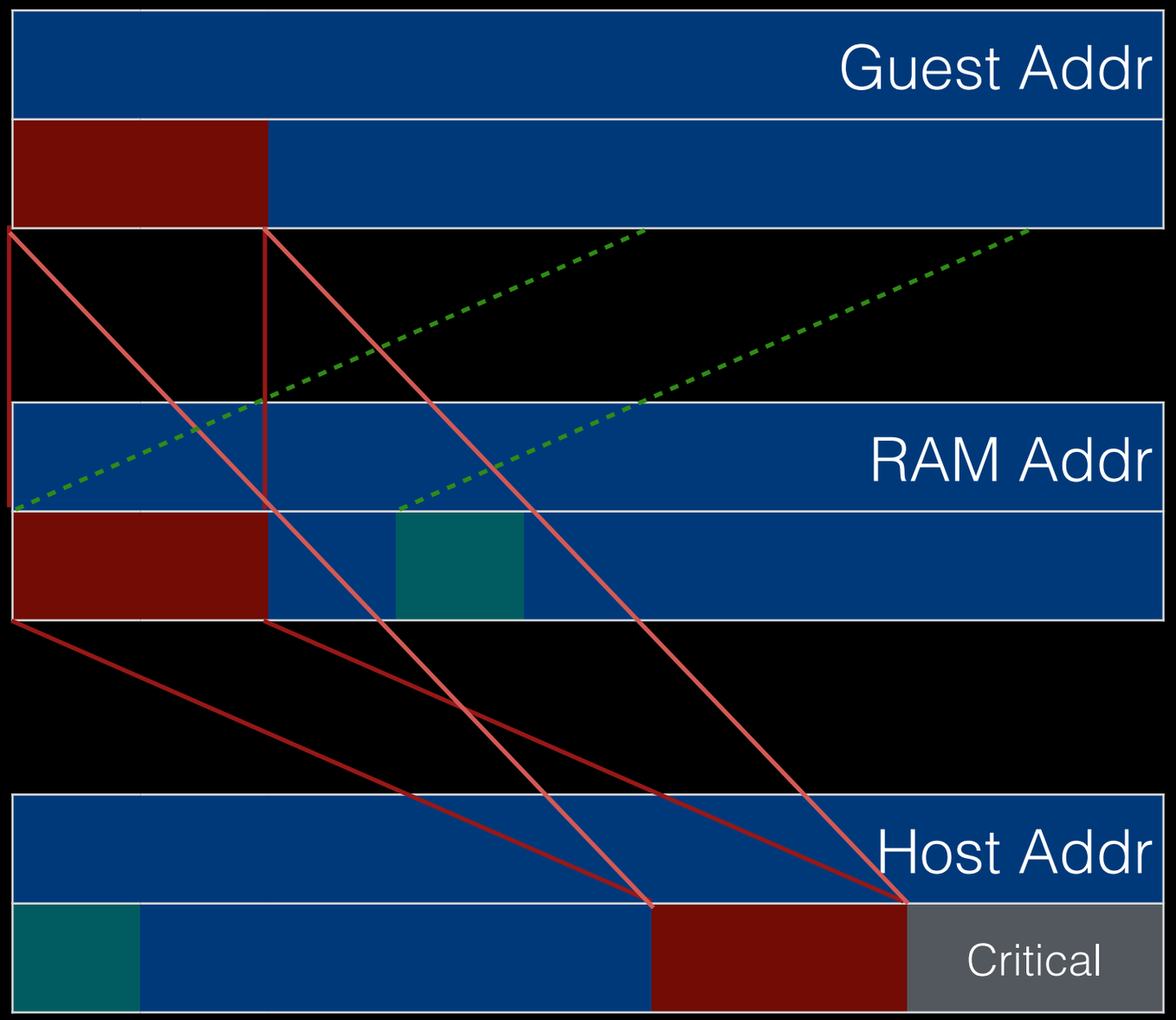




# map->nodes Exploit

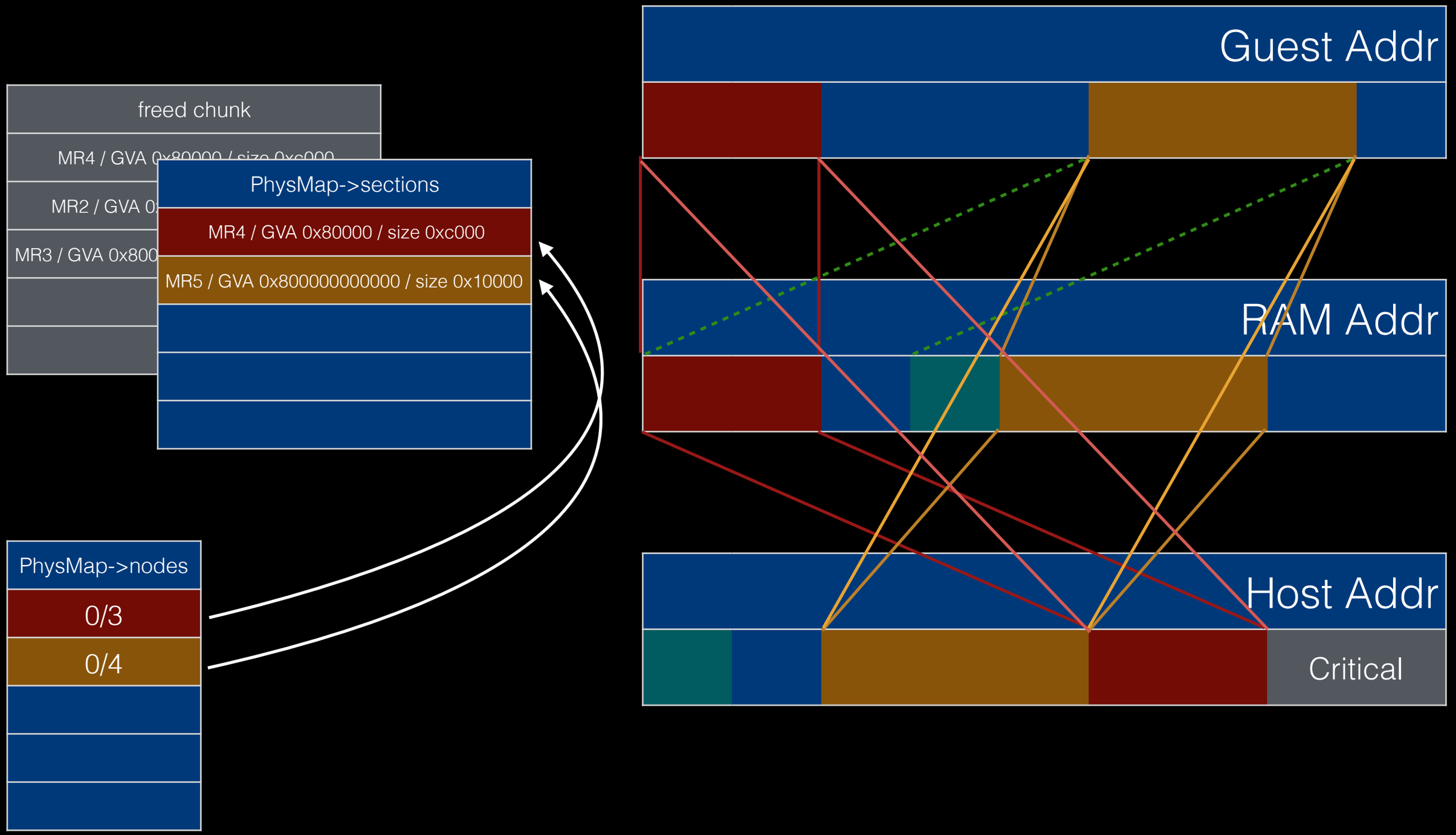
reclaim RAM for MemoryRegion3 with a smaller map size, this chunk should also fall right before Critical Region



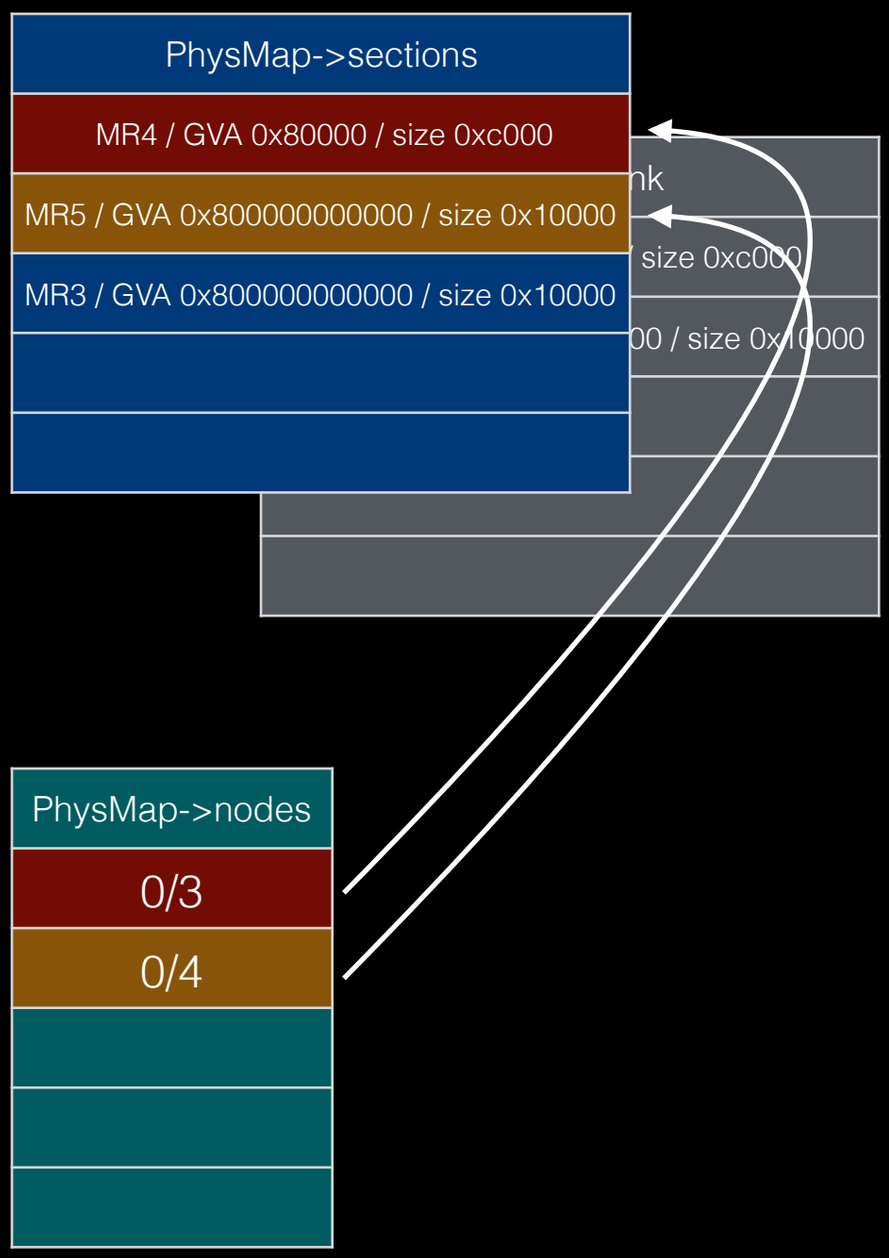
# map->nodes Exploit

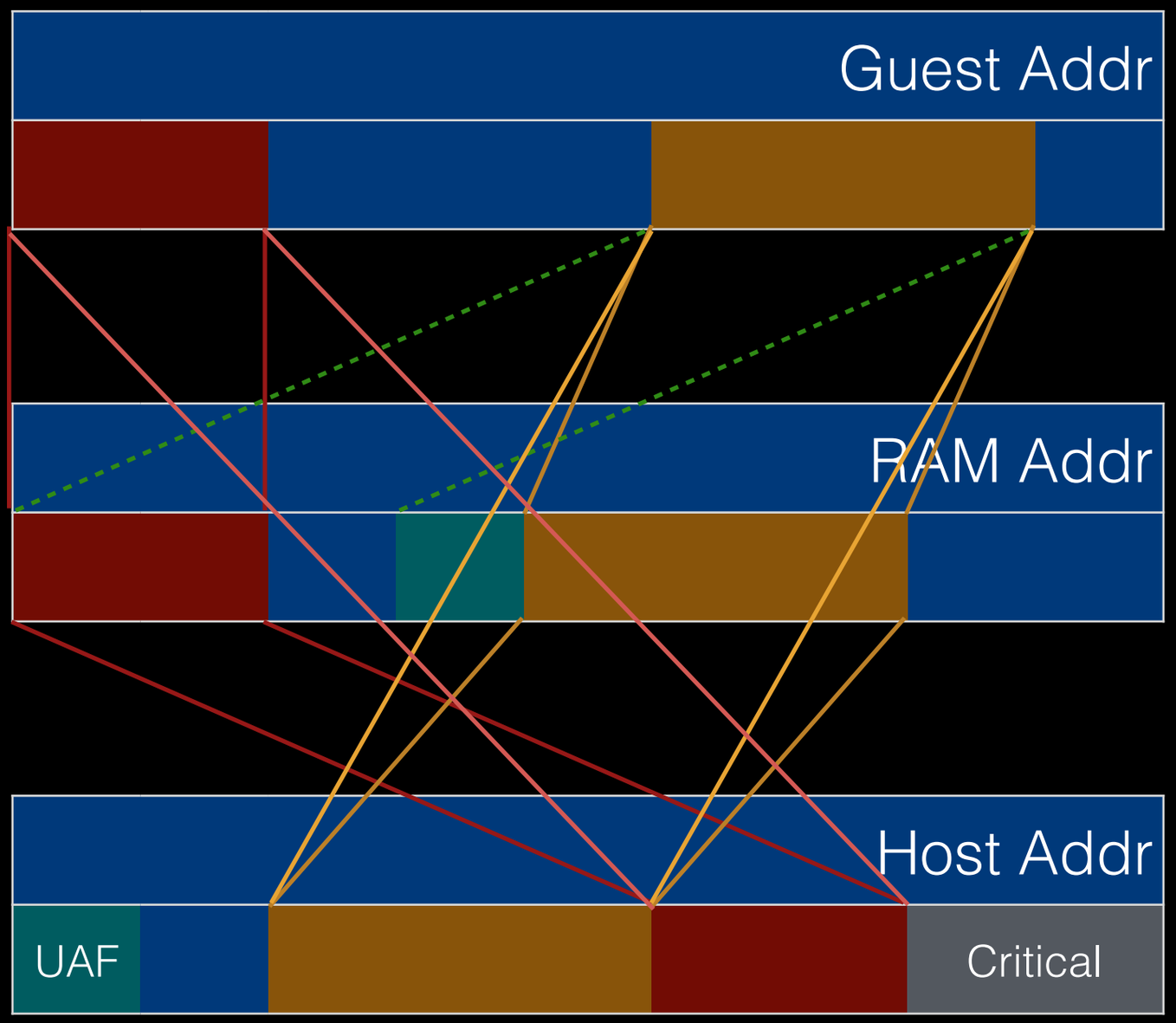
reclaim GVA for MemoryRegion3 with a larger map size than MemoryRegion4



# map->nodes Exploit

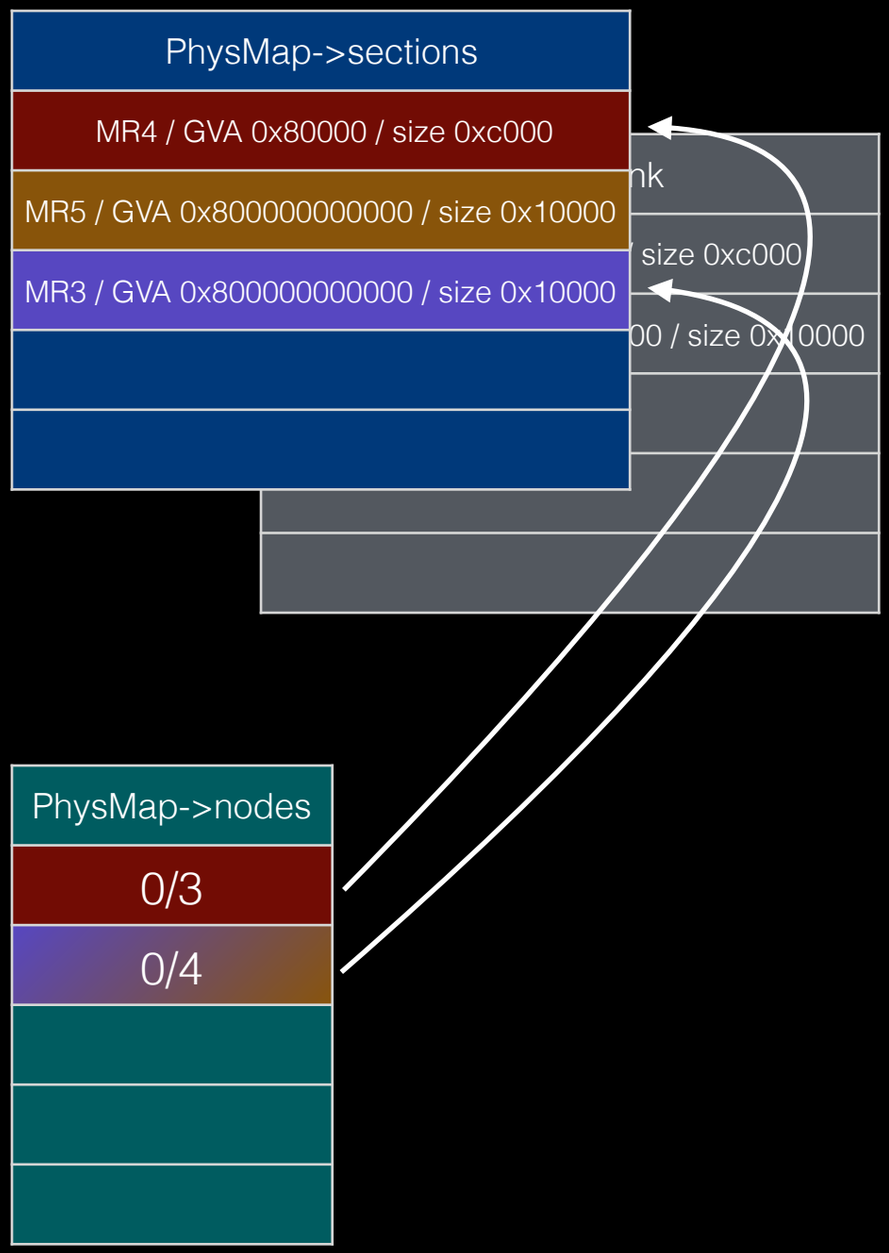
trigger UAF
note that the new Nodes array should be mapped onto UAF simultaneously

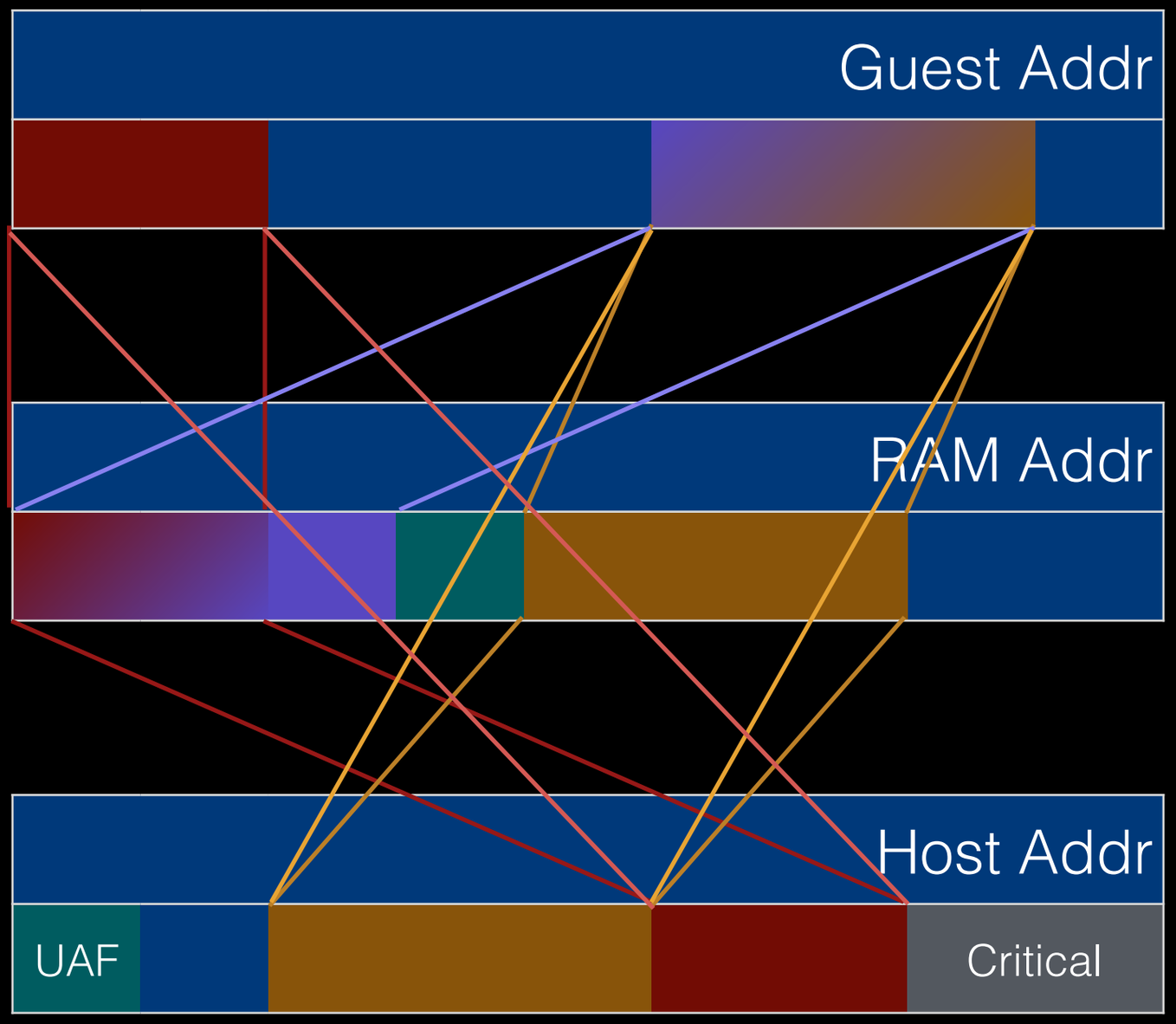


# map->nodes Exploit

modify idx in Nodes array for MemoryRegionSection5 to point to dangling MemoryRegionSection3



# map->nodes Exploit

now we're done with the entire setup, time to see OOB in action,
**uc_mem_read**(0x80000000bfff,0x100)

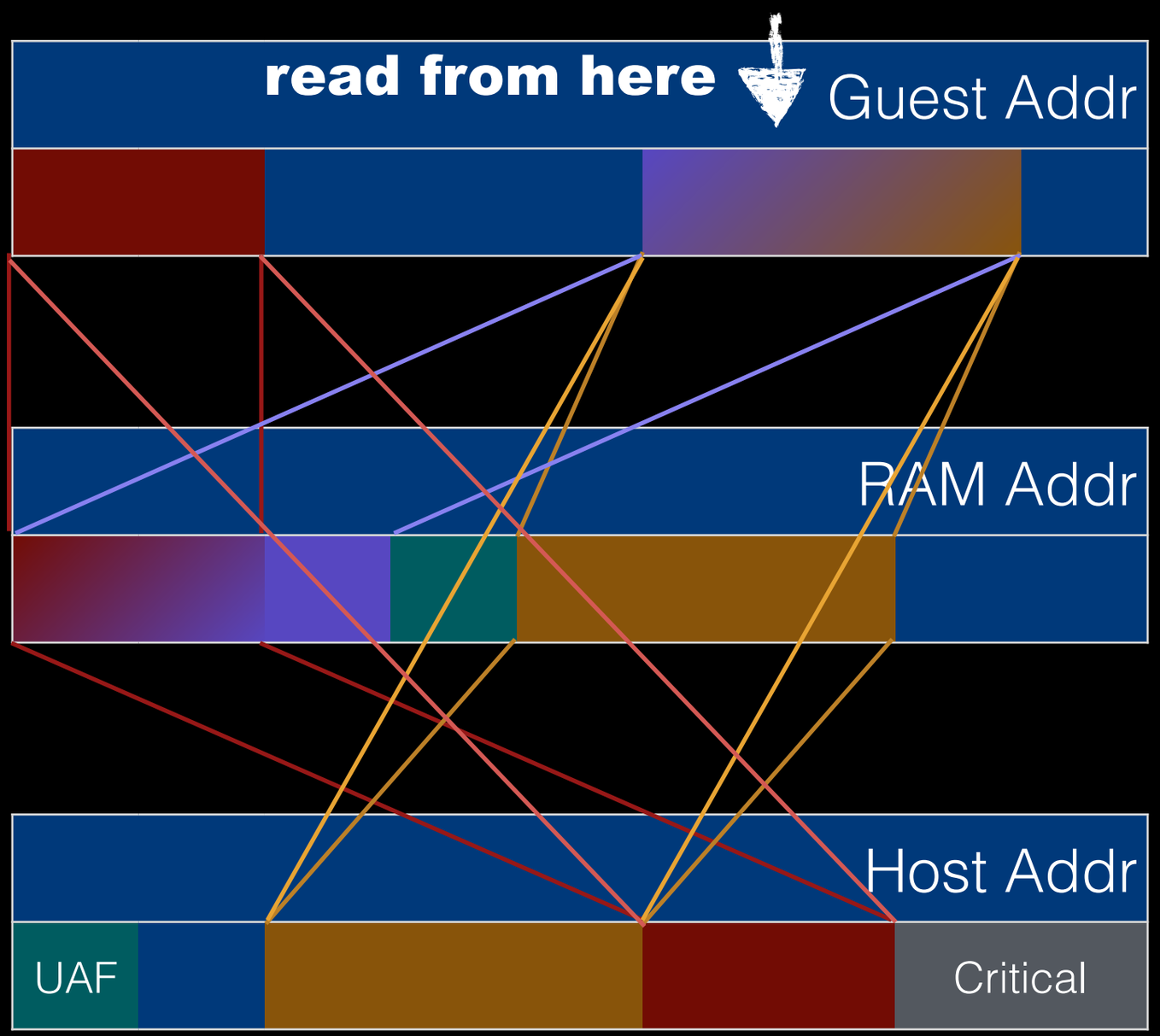
MemoryRegion
start : 0x80000
end : 0x8c000
ram_addr : 0

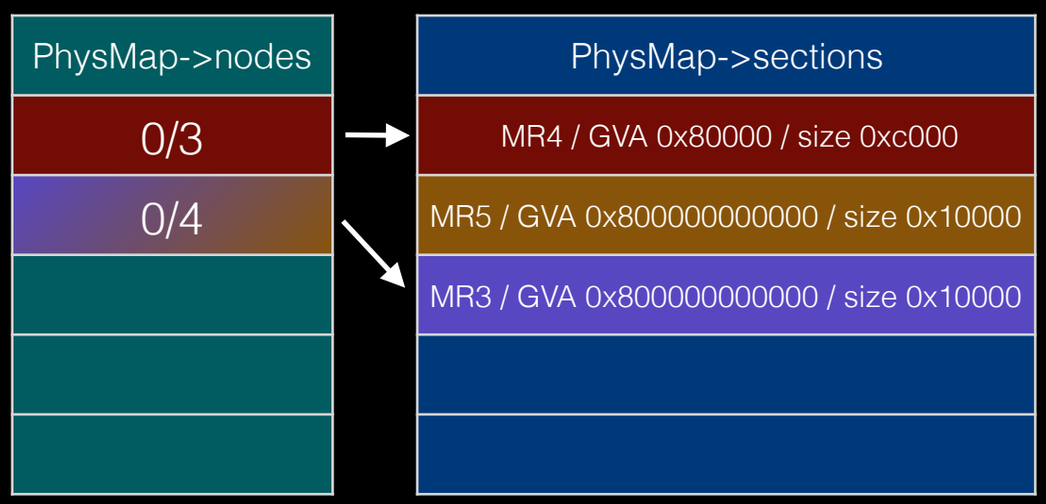
RAMBlock
ram_addr : 0
size : 0xc000
host : y

MemoryRegion
start : 0x800000000000
end : 0x800000010000
ram_addr : 0x11000

RAMBlock
ram_addr : 0
size : 0x10000
host : z

MemoryRegion
start : 0x800000000000
end : 0x800000010000
ram_addr : 0



# map->nodes Exploit

**uc_mem_read**(0x80000000bfff,0x100)
check if the range in legal by fetching from mapped_blocks, note the UAF (purple) mr is not in array

**looks legit**

MemoryRegion
start : 0x80000
end : 0x8c000
ram_addr : 0

RAMBlock
ram_addr : 0
size : 0xc000
host : y

MemoryRegion
start : 0x800000000000
end : 0x800000010000
ram_addr : 0x11000

RAMBlock
ram_addr : 0
size : 0x10000
host : z

MemoryRegion
start : 0x800000000000
end : 0x800000010000
ram_addr : 0

PhysMap->nodes

0/3

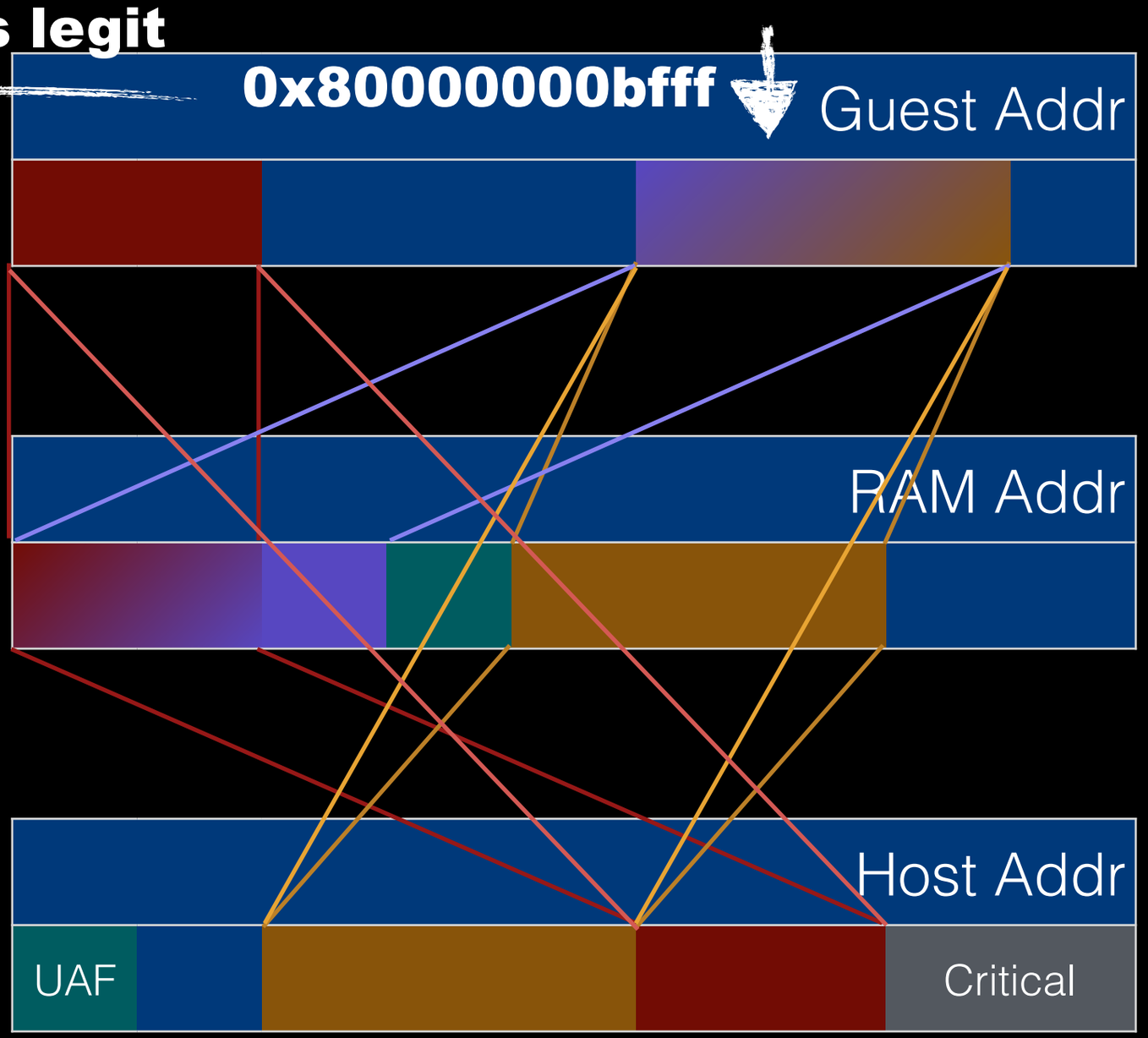
0/4

PhysMap->sections

MR4 / GVA 0x80000 / size 0xc000

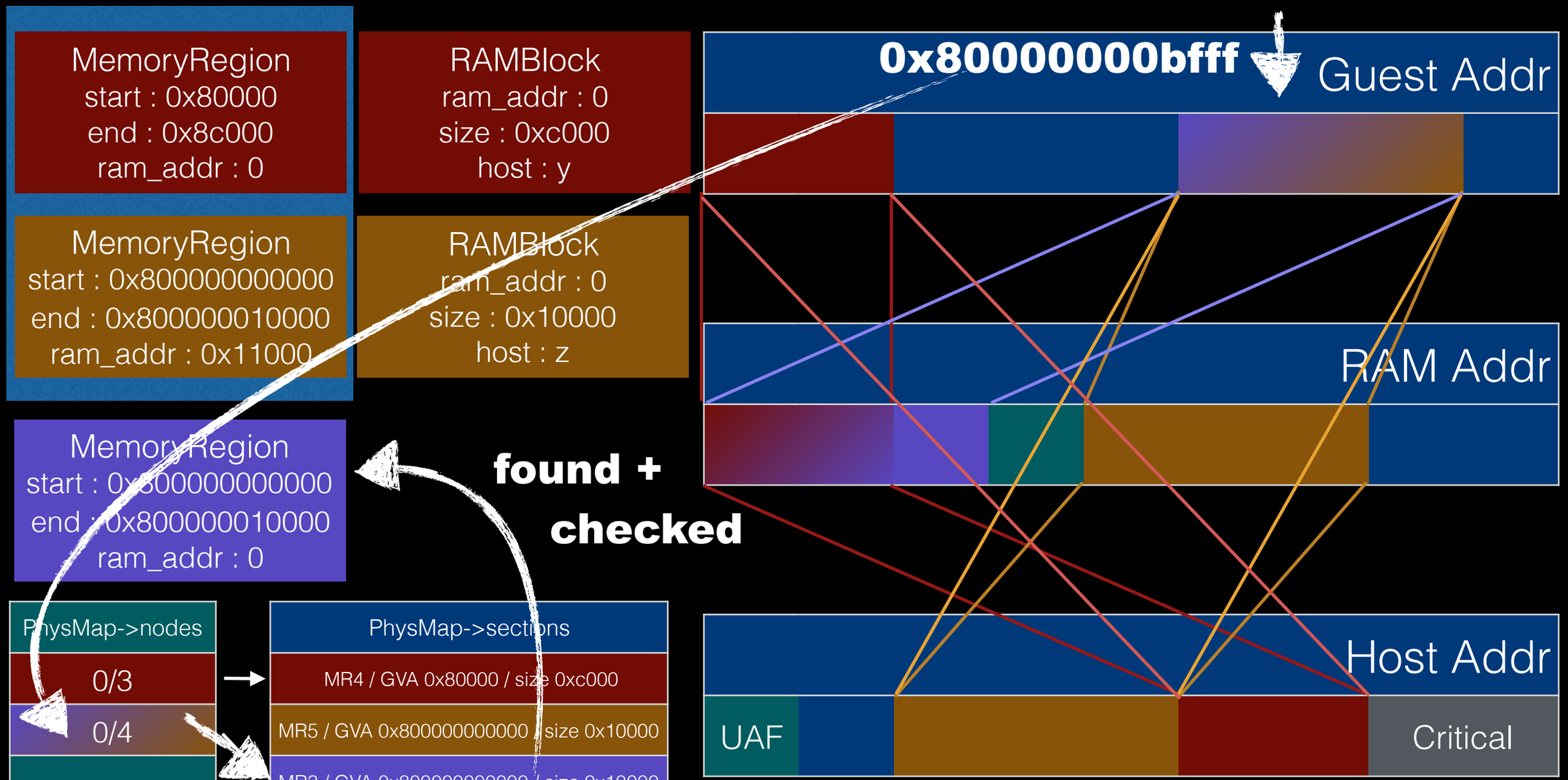
MR5 / GVA 0x800000000000 / size 0x10000

MR3 / GVA 0x800000000000 / size 0x10000



# map->nodes Exploit

**uc_mem_read**(0x80000000bfff,0x100)
walk PhysPageMap for target mr
note that the check on mr is what makes things so complicated



# map->nodes Exploit

**uc_mem_read**(0x80000000bfff,0x100)
fetch RAMBlock for ram_addr+offset,
and calculate host+offset

MemoryRegion
start : 0x80000
end : 0x8c000
ram_addr : 0

RAMBlock
ram_addr : 0
size : 0xc000
host : y

MemoryRegion
start : 0x800000000000
end : 0x800000010000
ram_addr : 0x11000

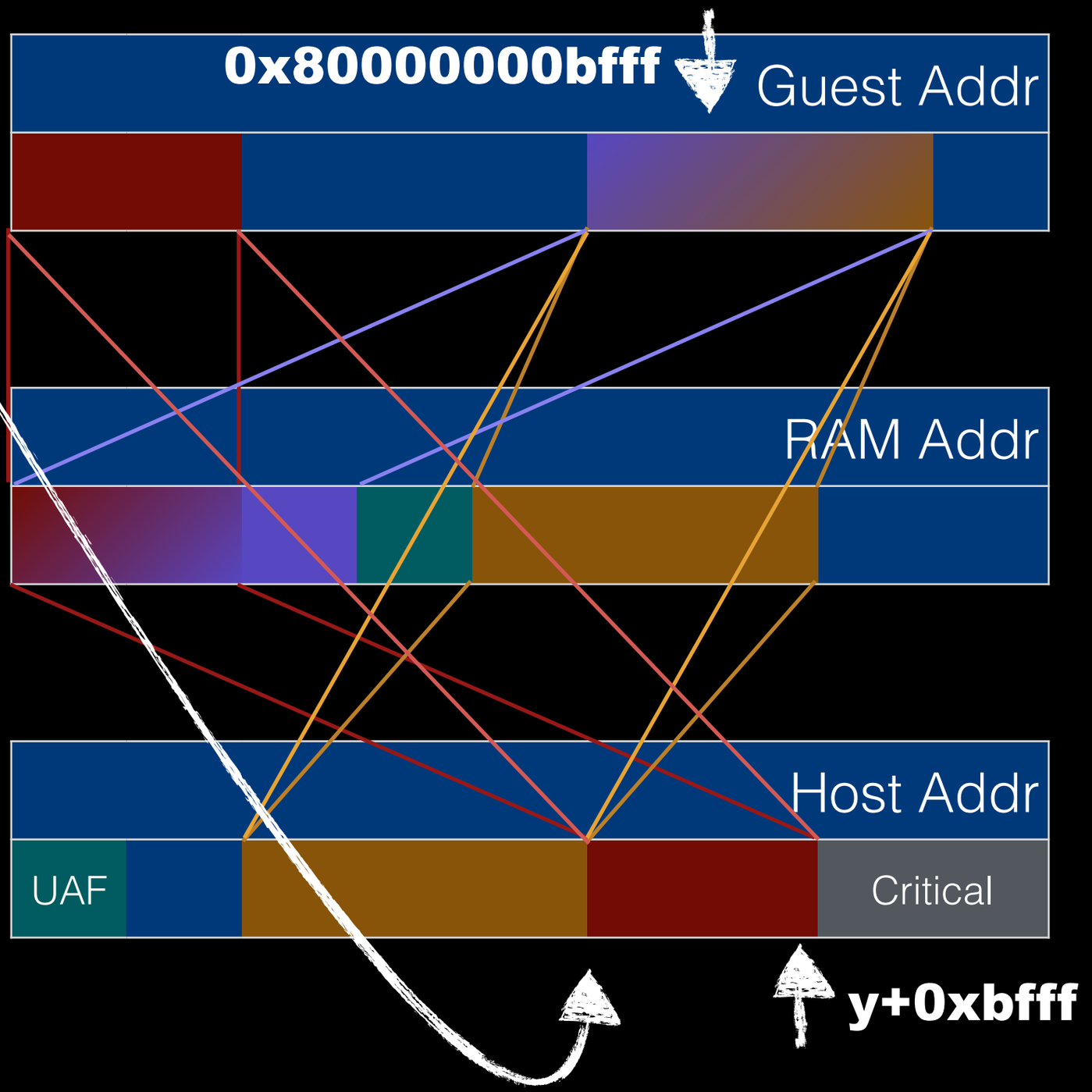
RAMBlock
ram_addr : 0
size : 0x10000
host : z

**found**

MemoryRegion
start : 0x800000000000
end : 0x800000010000
ram_addr : 0

| PhysMap->nodes |
|---|
| 0/3 |
| 0/4 |
| |
| |
| |

| PhysMap->sections |
|---|
| MR4 / GVA 0x80000 / size 0xc000 |
| MR5 / GVA 0x800000000000 / size 0x10000 |
| MR3 / GVA 0x800000000000 / size 0x10000 |
| |
| |



# map->nodes Exploit

action : **uc_mem_read**(x+0xfff,0x100)
**memcpy** and get OOB read

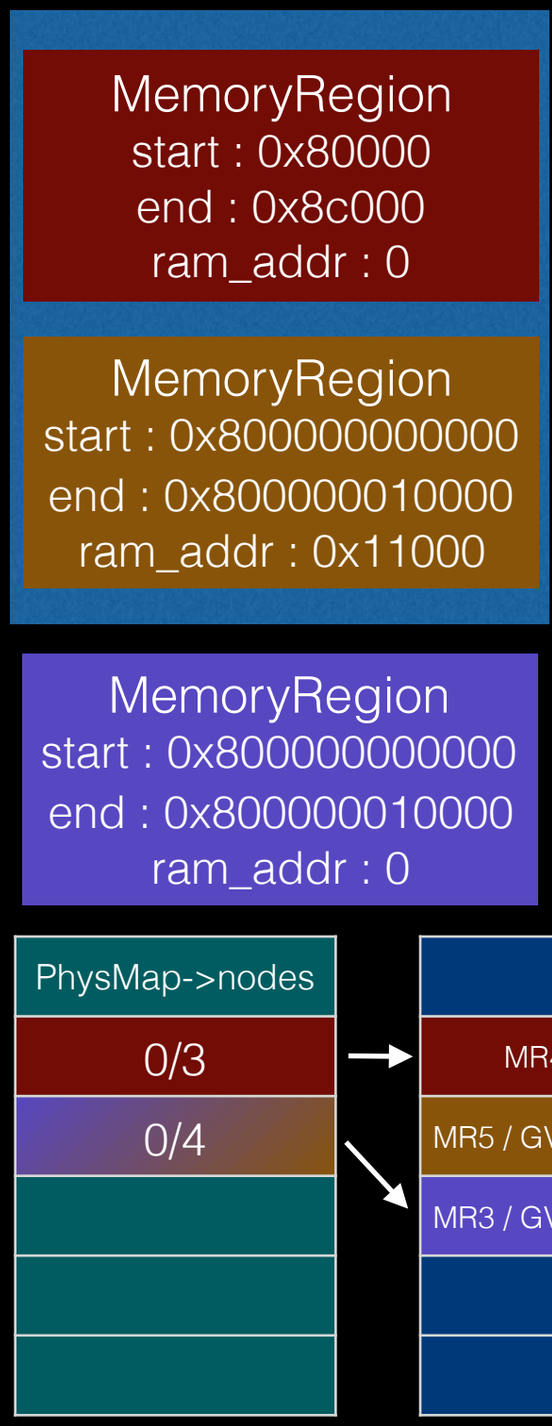
MemoryRegion
start : 0x80000
end : 0x8c000
ram_addr : 0

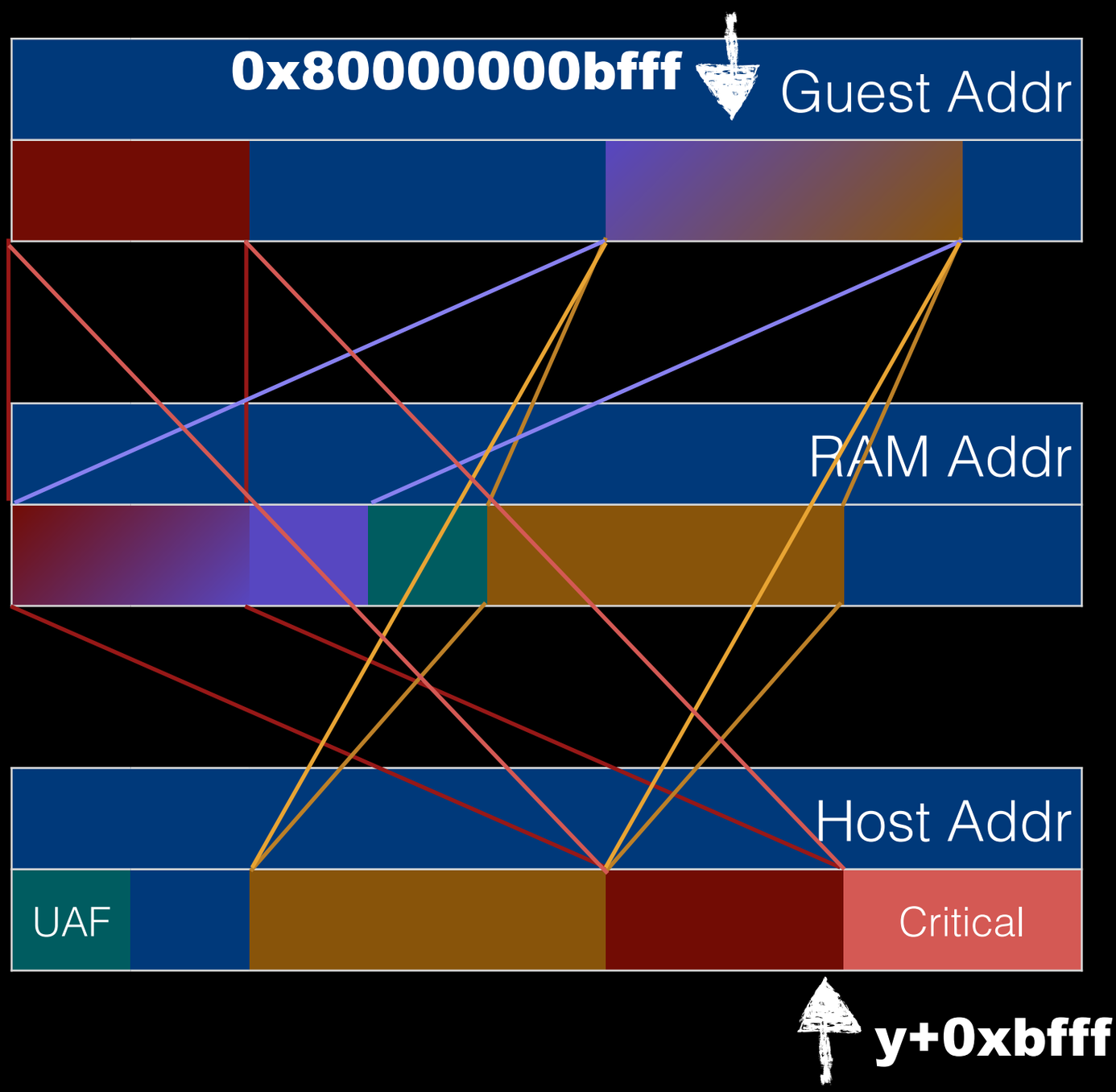
RAMBlock
ram_addr : 0
size : 0xc000
host : y

MemoryRegion
start : 0x800000000000
end : 0x800000010000
ram_addr : 0x11000

RAMBlock
ram_addr : 0
size : 0x10000
host : z

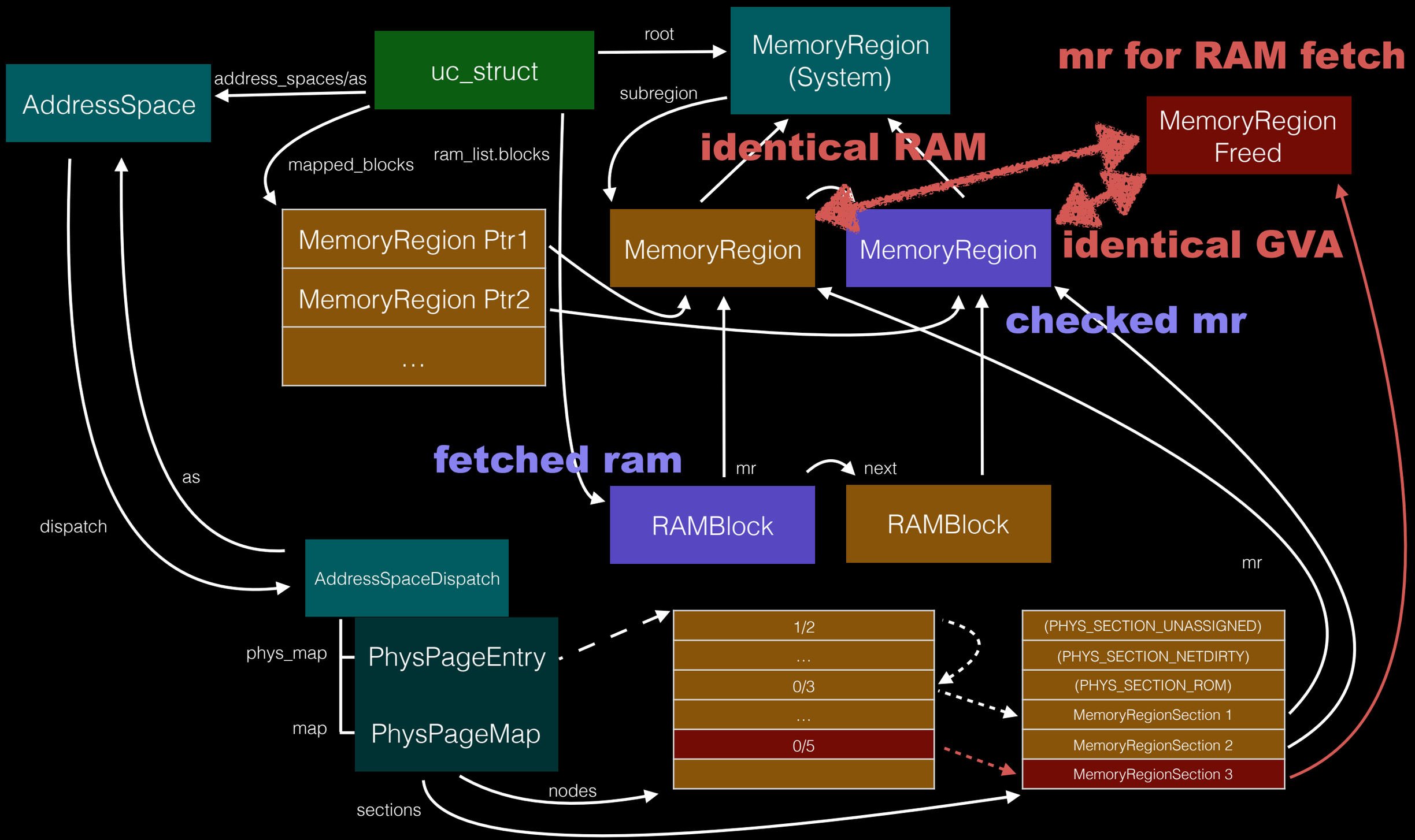
MemoryRegion
start : 0x800000000000
end : 0x800000010000
ram_addr : 0



# Concept Recap

the attack on map->nodes can be summarized as this



# Exploit

There are several unaddressed issues including

1. get a page right before critical region
2. prevent RAM address to line up correctly
3. ensure mr in tcache for UAF does not get claimed by other mr
4. force Nodes to fall on dangling mmap location at correct time

While some of these above challenges are non-trivial, I won't go on the details, since those are relatively distant to the core exploit concept. Feel free to read exploit code for details

At this point, we are finally ready to get our hands on finding a suitable "Critical Region" for OOB read/write

# Exploit

unicorn has a few large buffers/structures that will be allocated on launch (size here is taken w.r.t x86_64 host)

1. uc->tcg_ctx
   TCGContext struct
   size : 0xdd4a8
2. uc->tcg_ctx->code_gen_buffer
   rwx buffer for JITTed code
   size : 0x800000
3. uc->tcg_ctx->tb_ctx.tbs
   TranslationBlock array
   tcg_ctx->code_gen_max_blocks entries (0xfff8)
   size : 0x78*0xfff8 = 0x77fc40

those are allocated in listed order, uc->tcg_ctx->tb_ctx.tbs will be mapped on top of the other two, making it the only possible attack candidate

# Exploit

TranslationBlock has a pointer into rwx page which marks the code to execute

leak+modify this tc_ptr to point to some JOP code that we injected through JITTing constants

call the target tb and profit : )

```
1  struct TranslationBlock {
2      target_ulong pc;   /* simulated PC corresponding to this block (EIP + CS base) */
3      target_ulong cs_base; /* CS base for this block */
4      uint64_t flags; /* flags defining in which context the code was generated */
5      uint16_t size;      /* size of target code for this block (1 <=
6                             size <= TARGET_PAGE_SIZE) */
7      uint16_t cflags;    /* compile flags */
8  #define CF_COUNT_MASK  0x7fff
9  #define CF_LAST_IO     0x8000 /* Last insn may be an IO access.  */
10
11     void *tc_ptr;    /* pointer to the translated code */
12     /* next matching tb for physical address. */
13     struct TranslationBlock *phys_hash_next;
14     /* first and second physical page containing code. The lower bit
15        of the pointer tells the index in page_next[] */
16     struct TranslationBlock *page_next[2];
17     tb_page_addr_t page_addr[2];
18
19     /* the following data are used to directly call another TB from
20        the code of this one. */
21     uint16_t tb_next_offset[2]; /* offset of original jump target */
22 #ifdef USE_DIRECT_JUMP
23     uint16_t tb_jmp_offset[2]; /* offset of jump instruction */
24 #else
25     uintptr_t tb_next[2]; /* address of jump generated code */
26 #endif
27     /* list of TBs jumping to this one. This is a circular list using
28        the two least significant bits of the pointers to tell what is
29        the next pointer: 0 = jmp_next[0], 1 = jmp_next[1], 2 =
30        jmp_first */
31     struct TranslationBlock *jmp_next[2];
32     struct TranslationBlock *jmp_first;
33     uint32_t icount;
34 };
```

# Some Thoughts

The 0-day shown in postlude is assigned CVE-2021-44078

Given how easy the bug can be triggered and crash + ossfuzz has been fuzzing unicorn and found some bugs since last year

We once again demonstrate the incompetence/inefficiency of modern bug hunting methods

I personally believe that fuzzing is NOT the ultimate solution to bug hunting, and fundamental changes in methodology must be made to yield significant improvement (and no, deep nn is not the solution)

Feel free to contact me if you have interesting ideas that you'd like to share, or other project you feel like we could collaborate on

- twitter :  j_wang_11
- gmail  :  sec.james.wang@gmail.com
- github : jwang-a