nicole (noblenote)

a journey (starting on) reverse engineering

> game hacks on MacOS ARM and a few Rosetta facts

about me

nicole / noblenote

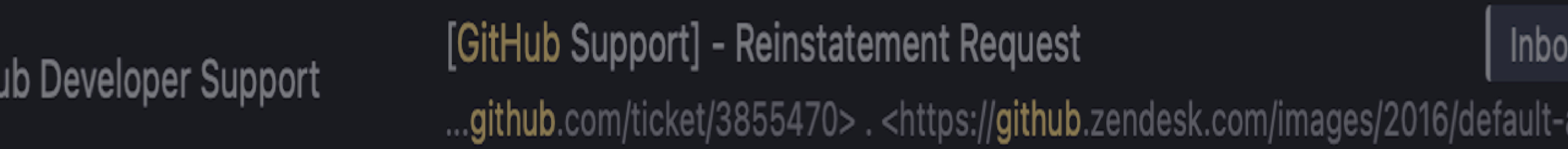
- mature age student
- previously a proj. manager/consultant
- interested in firmware analysis and reverse engineering

what you'll get out of this talk

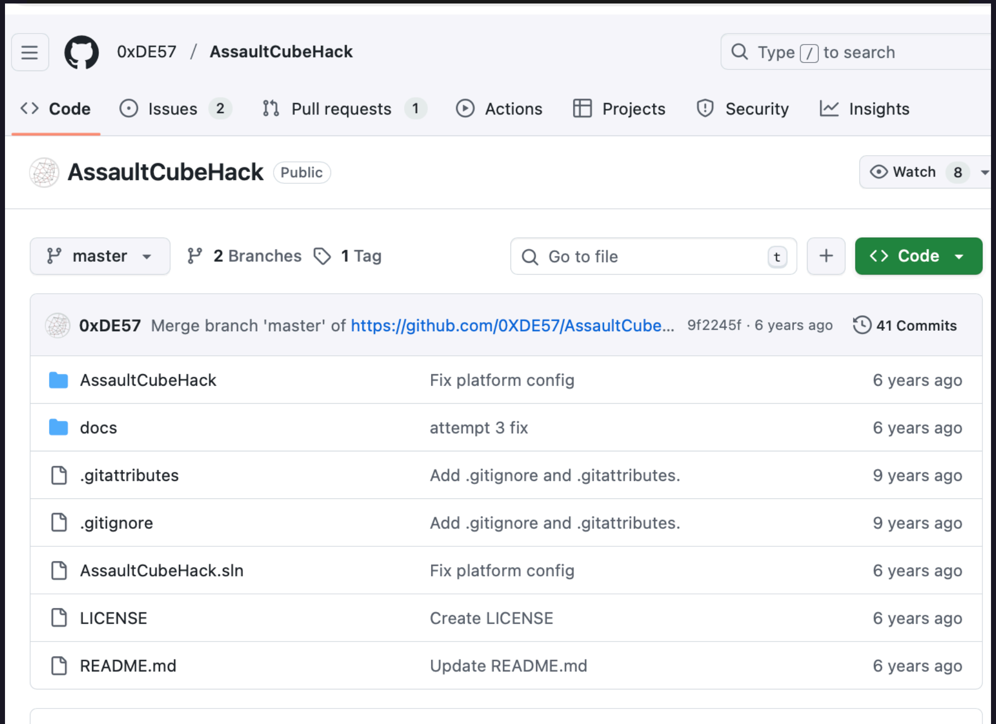
how I approached learning game hacking in TYOOL 2025 ❤️

- **understand basics of how Rosetta works for x86_64 binaries on MacOS**
- **my first punt of game hacking an x86 game with no reading of source, on Mac**
- **a small view of internal cheats for AssaultCube, the target game**

what you'll REALLY get out of this talk



nicole (noblenote)



many cheat guides are for Windows in x86_64, and getting to a decade old...

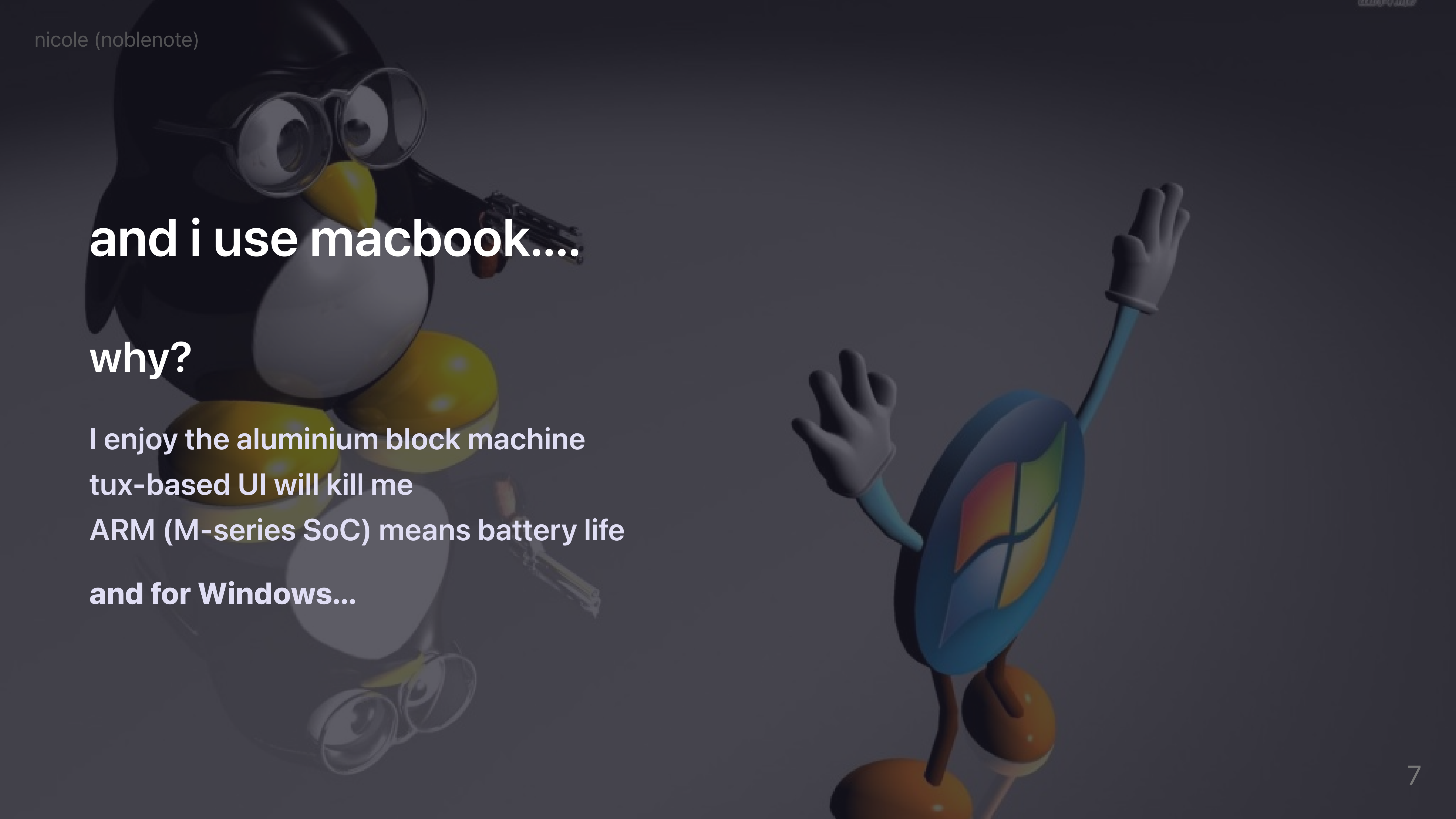
Academy :https://gamehacking.academy
Windows ESP: https://github.com/0xDE57/AssaultCubeHack

and i use macbook....

why?

I enjoy the aluminium block machine
tux-based UI will kill me
ARM (M-series SoC) means battery life

and for Windows...



and i use macbook....

why?

I enjoy the aluminium block machine
tux-based UI will kill me
ARM (M-series SoC) means battery life

and for Windows...

Intel Mac to Mx Mac

x86-64 to ARM64

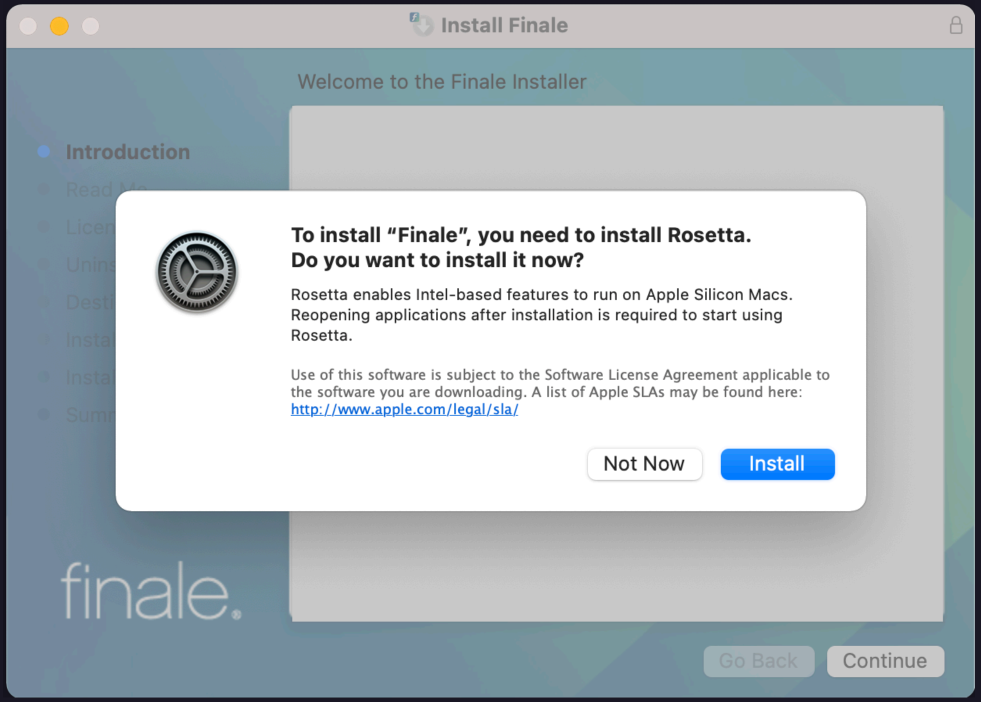
Games are still mostly x86-64

Wait.. how to run games on new Mac??

MacOS Games and Rosetta

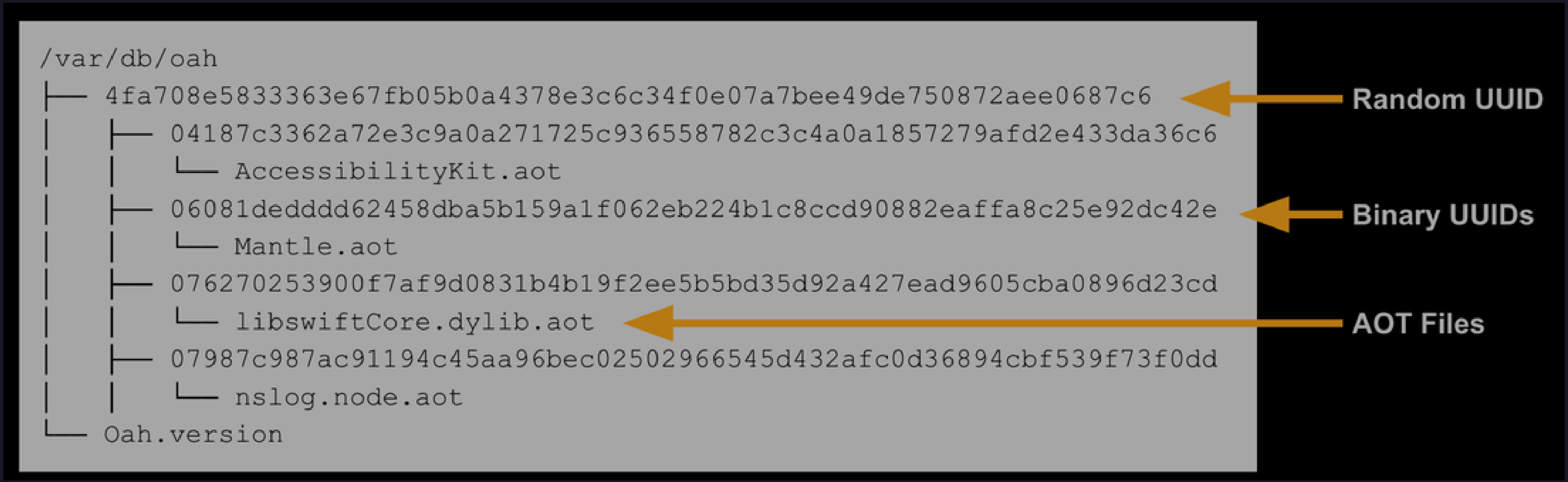
Rosetta 2 is the way x86-64 binaries like most games run on Apple Silicon. Phasing out in 2 major releases from now...

1. Translated ahead-of-time in a cache at `/var/db/oah` on first run.
2. Will parse *all* code needed from x86, including libraries (signed or not)
3. Pretty efficent with translation, keeps self-modifying code/indirect branches for JIT compilation.



if you wanted to look at OAH...

You will need to disable SIP as `oah` and it's daemon to check oah `oahd` can't be seen by root. SIP limits root somewhat on MacOS, and you should keep it on to not get computer lice



Mandiant - Rosetta 2 Artifacts in macOS

cd /Applications/AssaultCube.app/Contents/gamedata\nlldb -- assaultcube.app/Contents/MacOS/assaultc...

```
0x10009b55d: cvtsi2ss xmm3, ecx
0x10009b561: movss   xmm2, dword ptr [rip + 0xbd593]
0x10009b569: call    0x10009ab50     ; ___lldb_unnamed_symbol1840
0x10009b56e: mov     edi, 0x1700
0x10009b573: call    0x100156402     ; symbol stub for: glMatrixMode
0x10009b578: call    0x1001563f6     ; symbol stub for: glLoadIdentity
0x10009b57d: mov     rax, qword ptr [rip + 0x13a6c4] ; __progname + 143552
0x10009b584: movss   xmm0, dword ptr [rax + 0x40]
0x10009b589: movss   xmm3, dword ptr [rip + 0xbc263]
0x10009b591: xorps   xmm1, xmm1
0x10009b594: xorps   xmm2, xmm2
0x10009b597: call    0x100156444     ; symbol stub for: glRotatef
0x10009b59c: mov     rax, qword ptr [rip + 0x13a6a5] ; __progname + 143552
0x10009b5a3: movss   xmm0, dword ptr [rax + 0x3c]
0x10009b5a8: movss   xmm1, dword ptr [rip + 0xbc350]
0x10009b5b0: xorps   xmm2, xmm2
0x10009b5b3: xorps   xmm3, xmm3
0x10009b5b6: call    0x100156444     ; symbol stub for: glRotatef
0x10009b5bb: mov     rax, qword ptr [rip + 0x13a686] ; __progname + 143552
0x10009b5c2: movss   xmm0, dword ptr [rax + 0x38]
0x10009b5c7: xorps   xmm1, xmm1
0x10009b5ca: xorps   xmm3, xmm3
0x10009b5cd: movss   xmm2, dword ptr [rip + 0xbc21f]
0x10009b5d5: call    0x100156444     ; symbol stub for: glRotatef
0x10009b5da: movss   xmm0, dword ptr [rip + 0xbc28e]
0x10009b5e2: xorps   xmm2, xmm2
0x10009b5e5: xorps   xmm3, xmm3
0x10009b5e8: movss   xmm1, dword ptr [rip + 0xbc204]
0x10009b5f0: call    0x100156444     ; symbol stub for: glRotatef
0x10009b5f5: movss   xmm0, dword ptr [rip + 0xbc1f7]
0x10009b5fd: movss   xmm1, dword ptr [rip + 0xbc2fb]
0x10009b605: movaps  xmm2, xmm0
0x10009b608: call    0x10015644a     ; symbol stub for: glScalef
0x10009b60d: mov     rax, qword ptr [rip + 0x13a634] ; __progname + 143552
0x10009b614: movss   xmm0, dword ptr [rax + 0x8]
0x10009b619: movss   xmm1, dword ptr [rax + 0xc]
0x10009b61e: movaps  xmm3, xmmword ptr [rip + 0xbc25b]
0x10009b625: xorps   xmm0, xmm3
0x10009b628: xorps   xmm1, xmm3
0x10009b62b: movss   xmm2, dword ptr [rax + 0x10]
```

Get your x86 games on Apple Silicon!

Debug-friendly AOT binary preserves register names and symbols to refer to the original binary.

- lldb (see register/symbols)
- dtrace (syscalls & rosetta state)
- otool (disass w/ -tv)

Rosetta creates a return stack that handles RIP-relative addressing (using `ADRP` and `ADD` for PC-relative branches)

```
→ vmmap 4196 | grep 'Rosetta'
mapped file                 10d061000-10d0c9000  /Library/Apple/usr/libexec/oah/
mapped file                 10d0cd000-10d0d3000  /Library/Apple/usr/libexec/oah/
Rosetta Thread Context      10d0d6000-10d0d7000    [    4K      0K      0K      0K]
Rosetta Return Stack        10d0db000-10d0dc000    [    4K      0K      0K      0K]
Rosetta Generic             10d0e0000-10d0e1000    [    4K      0K      0K      0K]
__TEXT                   7ffd1fc56000-7ffd1fc5e000  /usr/lib/libRosetta.dylib  K]
__DATA_CONST             7ffd49259f70-7ffd4925a308  /usr/lib/libRosetta.dylib  K]
```

Process Layout in Rosetta 2

Original x86-64 Binary Kept for reference to JIT, and debugging

DATA Read-only data (consts)

AOT Translation *Primary executable* from `oah`

JIT Appended ARM64 for JIT usecases

Rosetta Support Code Syscall translation

📝 **Why is Rosetta 2 fast? - dougallj**
https://dougallj.wordpress.com/2022/11/09/why-is-rosetta-2-fast/

Why did I look at Rosetta 2?

our target games will be in x86, like many games - we'll need to keep this in mind!

- **Frida/many frameworks for x86 hacks did not work with Rosetta 2 translation.**
- **We can leverage that x86 gets translated to inject x86 libraries for the AOT cache.**
- **Without this game in mind, we could force conditions for JIT to always consider via self-modifying code.**

also... fun

📝 **This is explored in Koh Nakagawa's blog using execv to explore Rosetta 2**
https://ffri.github.io/ProjectChampollion/part1/

okay lets start the game hack finally

The Obvious Disclaimer

These Cool Three Things Help You Enjoy Modding Without Legal Aid

1. most EULAs prevent you from modifying the code
2. there ARE games that allow you and even encourage you to reverse engineer!
    - **Squally** https://github.com/Squalr/Squally
    - **PwnAdventure** https://www.pwnadventure.com

tldr: don't have this ruins peoples fun and/or be illegal

assaultcube - the game of choice

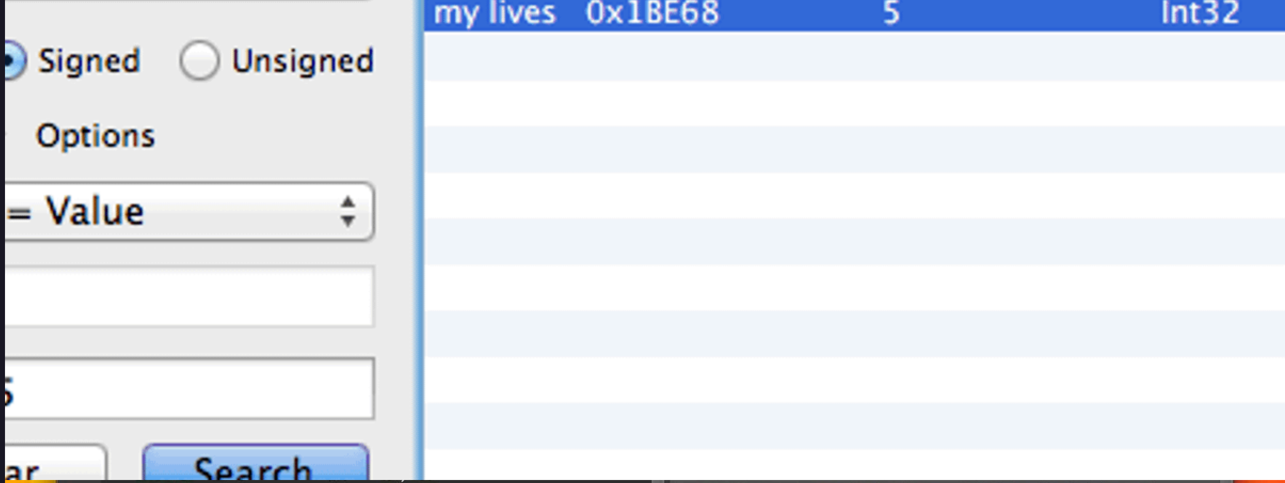
covered extensively on Windows and even has a run through on Intel Mac using Frida for game cheats

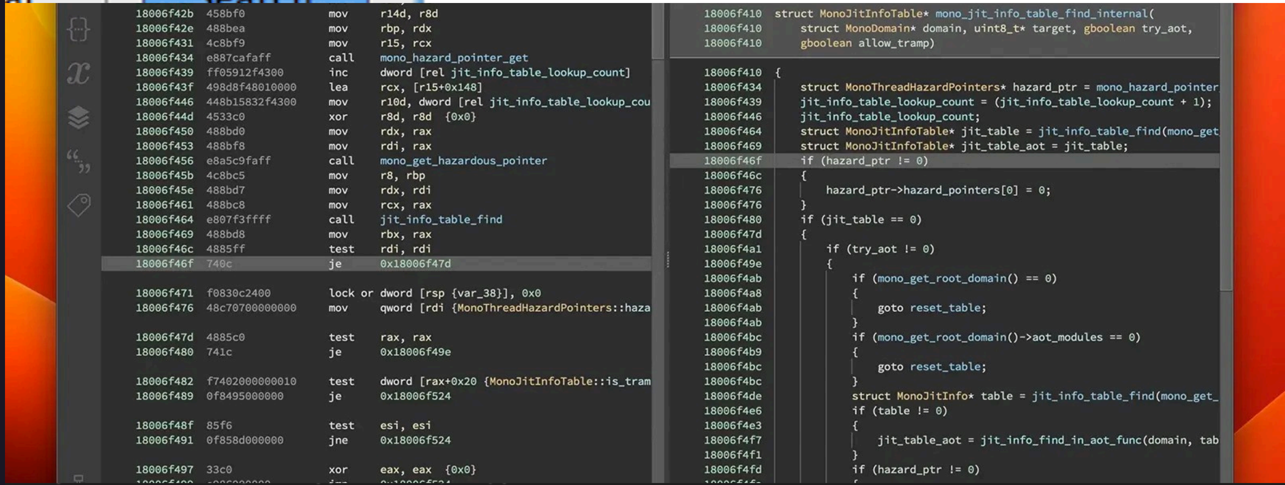
📝 Jai Verma presented an Intel Mac hack on AssaultCube BSides Delhi 2019 https://jaiverma.github.io/blog/ac-hack

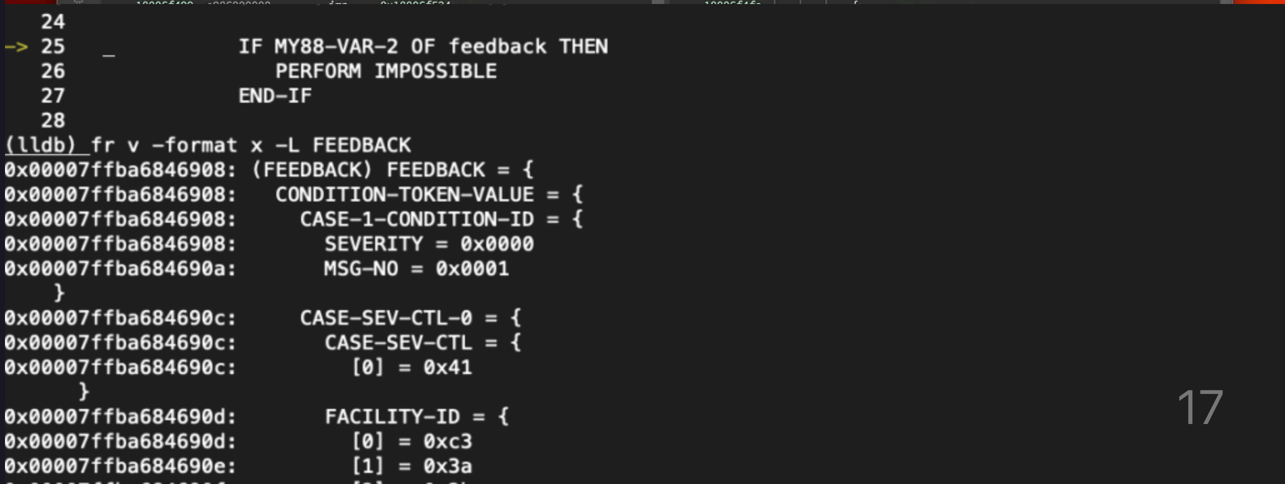
tools used

1. **BitSlicer (memory scanning/CheatEngine clone)**
2. **BinaryNinja (decompile that thing!)**
3. **LLDB (LLVMs gdb)**

- Frida could also be used - a pain due to it wanting to target x86 address space when running a Rosetta game...







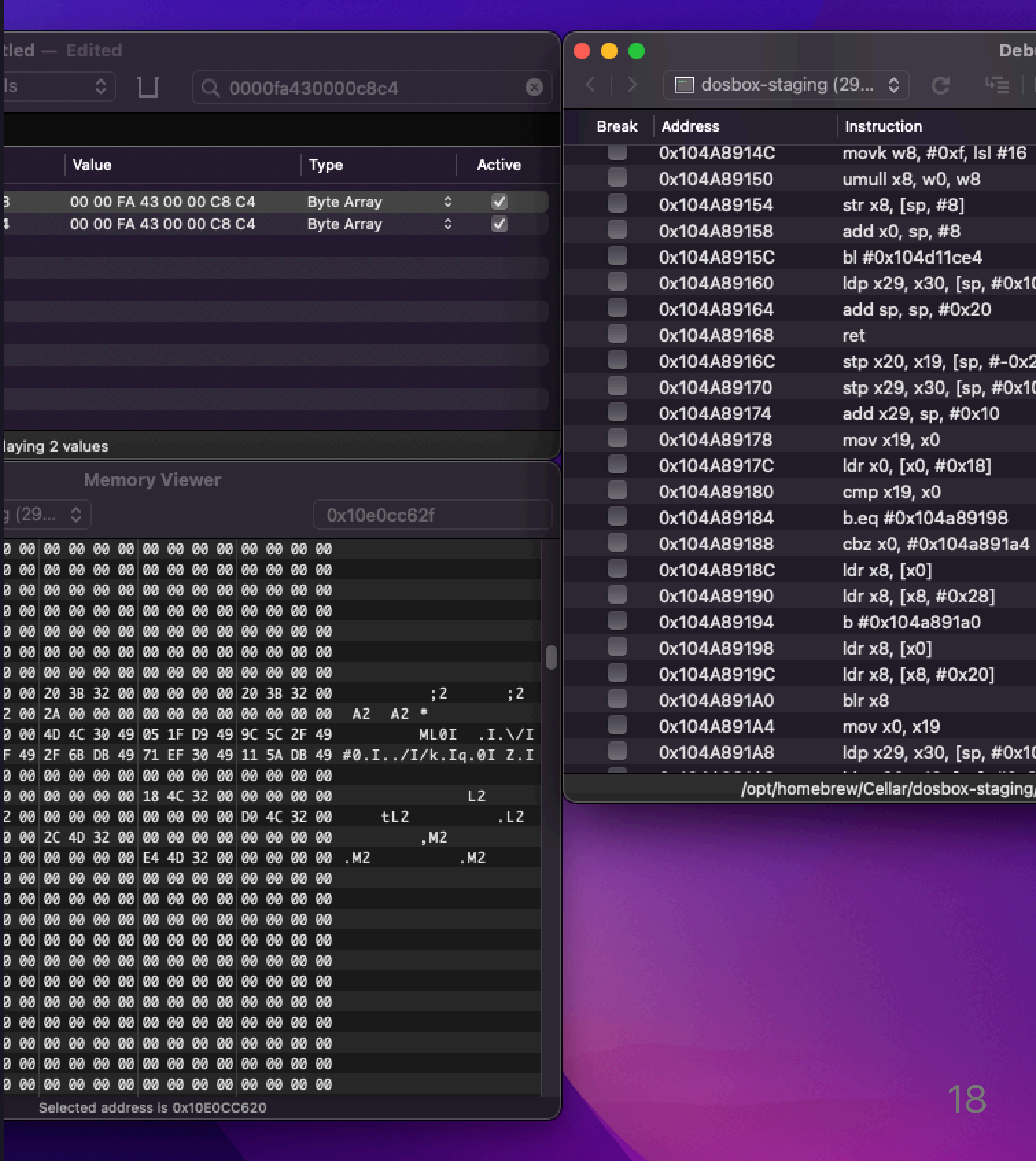
nicole (noblenote)

BitSlicer

https://github.com/zorgiepoo/Bit-Slicer

Similar to Cheat Engine but for MacOS

Narrow down values in memory, debug instructions and write scripts again vm regions and debugger methods...



cool! now what would we want to edit in a video game cheat?

Think about what gives you an advantage in a first-person shooter like AssaultCube.

What would it's type *likely* be? Where would it be in memory?

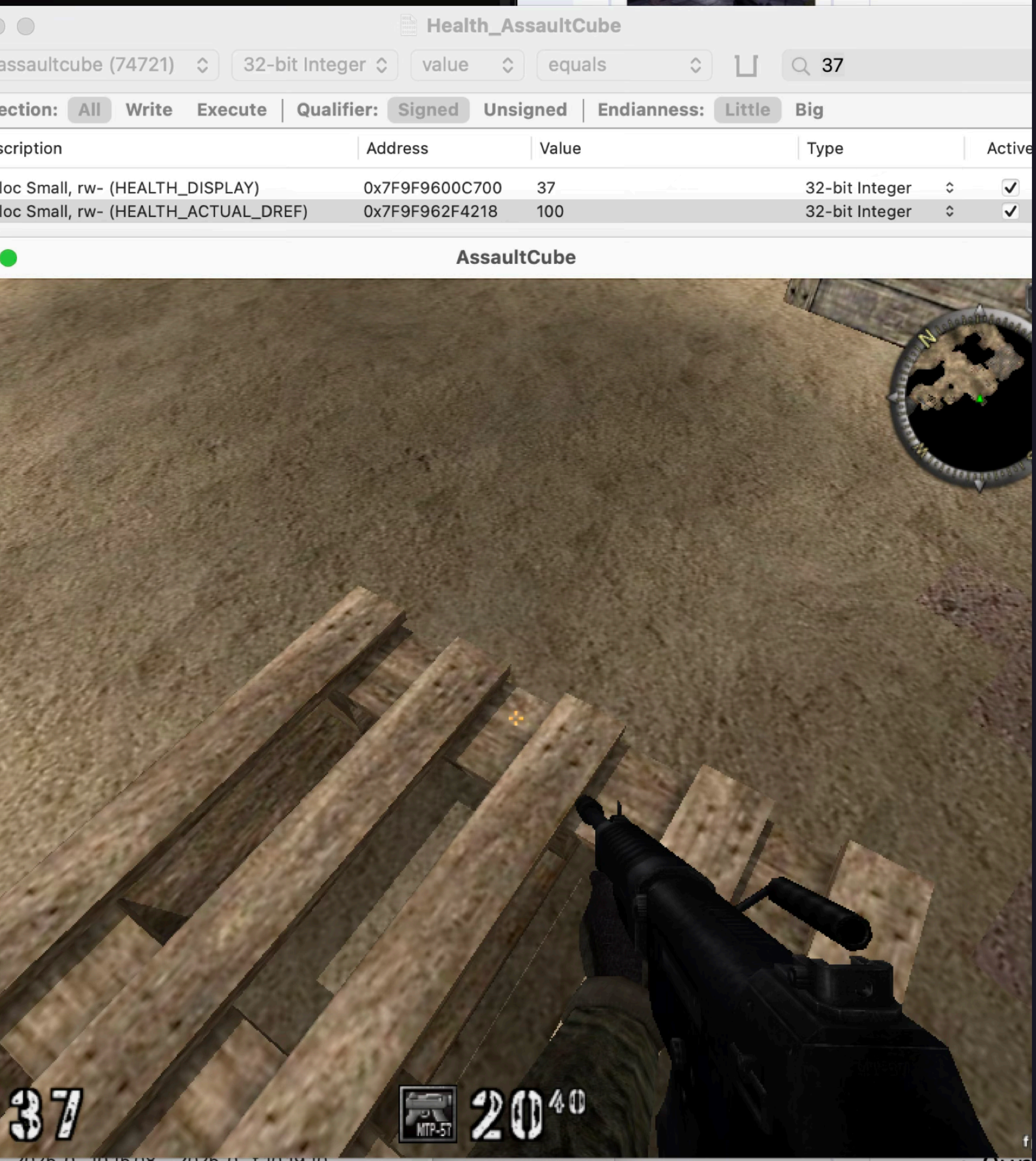
- **health** (likely an integer; could be defined and initialised per-player from a signature/definition)
- **ammo** (same....)
- **location** (could be a float or large int, with X/Y/Z locations, kept per-player *or* in a list of players?)

How would the game track all players, would it pass-by-reference or value?

finding these values in memory

We can try to use a debugger like LLDB at this stage, but that is very precise for what we are trying to do, getting a needle in a haystack. We can script it, or use BitSlicer to look for known values!

> AssaultCube isn't in ARM, but translated over x86 for M-series Macs. This means we'll need to keep this in mind when we see instructions in our debuggers or tools.



the offset safari

Using BitSlicer - ask for all values at 100, then take damage and ask again till you have very few, then edit them!

After that, set a watch on your addresses for read/write.

```
str w3, [x15, #0x418]
ldr w23, [x14, #0x550]
; W is lower 32-bits of X-registers in AArch64
```

pointing to pointer chains

okay.... so we have the value and the offset from x15 (temp registers) for further work

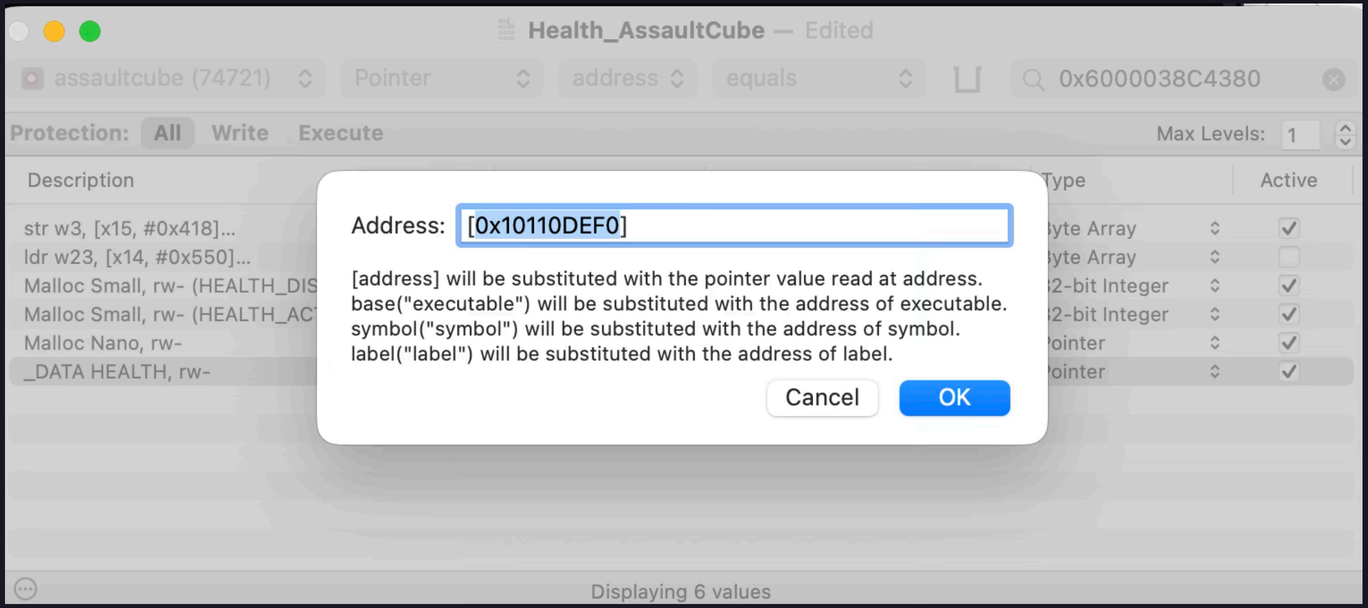
```
str w3, [x15, #0x418]
Accessed 1 time
Registers
... etc etc
x15 = 0x7F9F962F3E00 (140323396206080) <- this plus 0x418
fp = 0x201223F20 (8608956192)
Ir = 0x101280ED4
```

`x15` + `0x418` is `0x7F9F962F4218`!

Bit Slicer can help us here again - resolving variables from a pointer!

Using this value to resolve, we'll get the health variable in _DATA.

slay! this is our base_addr + offset value.



Why is it still indirect? In the x86-64 tutorials, this is just the address offset (eg: `base() + 0x1000`).

This is due to Rosetta's AOT and how it resolved to shared data, but that's okay - we just can get the base address and do napkin math!

calculator time

now we just get the base address of our currently running process.

we can use:

- LLDB's `image list`
- `vmmap`
- Practice with `task_for_pid()` and `mach_vm_region()` to learn more about how we'd make an external cheat with Mach tasks.

```
 ptnmutt  .../assaultcube_health_rust/target/debug
→ sudo ./get_task_for_pid
Enter pid: 74721
pid = 74721
Got Mach task = 0x1d03
Found executable region: 0x100f34000 - 0x100f35000
Task is running 16 threads
Thead  0
fp   = 0x0000000064000000
lr   = 0x00007ff89a5240d8
sp   = 0x000000010940da70
pc   = 0x00007ff89521eb80
cpsr = 0x88001000
x0  = 0x000000000000ae03
x1  = 0x0000000000000002
x2  = 0x0000000000000042
x3  = 0x0000000000000000
x4  = 0x0000000000000000
x5  = 0x0000000000000000
```

📝 **attilathedud/macos_task_for_pid**
https://github.com/attilathedud/macos_task_for_pid

Function

mach_vm_allocate

macOS 10.4+

```
kern_return_t mach_vr
```

Kernel / Kernel Data Types / kern_r

Type Alias

kern_return_t

macOS 10.0+

Mad World by Tears For Fears Original HQ 1983

we are back - `0x10110DEF0` (health) - `0x100F34000` (base) = `0x1D9EF0`

```rust
unsafe fn get_base_address(task: task_t) -> Option<u64> {
    let mut address: mach_vm_address_t = 1; // skip null page
    let mut size: mach_vm_size_t = 0;
    let mut info = vm_region_basic_info_64::default();
    let mut count = VM_REGION_BASIC_INFO_COUNT_64;
    let mut object_name: mach_port_name_t = 0;

    unsafe {
        while mach_vm_region(
            //regions (_info/64/_recurse)
            task,
            &mut address,
            &mut size,
            VM_REGION_BASIC_INFO_64,
            &mut info as *mut _ as vm_region_info_t,
            &mut count,
            &mut object_name,
        ) == KERN_SUCCESS
        // run till kernel yells at us
        {
            // look for read + exec
            let prot: i32 = info.protection.try_into().unwrap();
            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
}
```

making game cheats with these offsets

I'll be working on an internal cheat to slip into the Rosetta 2 AOT cache, which will be internal. But here's some definitions on external vs. internal cheats

- **External** (use `task_for_pid()` or alike methods to control process memory via another process. Another process accesses region to own targets flow)
- **Internal** (load your codecave/cheat as a library that the process brings along for the ride, or directly recompile)

✨there's way more styles✨

- hardware (DMA)
- hypervisor

well now we can leverage the fact it's x86!

Similar to `LD_PRELOAD` forces libraries to load ahead of a process, we can use `DYLD_INSERT_LIBRARIES` to throw a .dylib file

```
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
```

Rosetta 2 has less bounds compared to signed ARM/Univeral binaries, and since this is just x86, we can chuck in this DYLD!

This is also how the POOLRAT (2023) Python2 executed x86-64 versions of ping/chmod to ARM via Rosetta 2.. but also how we have the AOT evidence of it in MacOS victims!

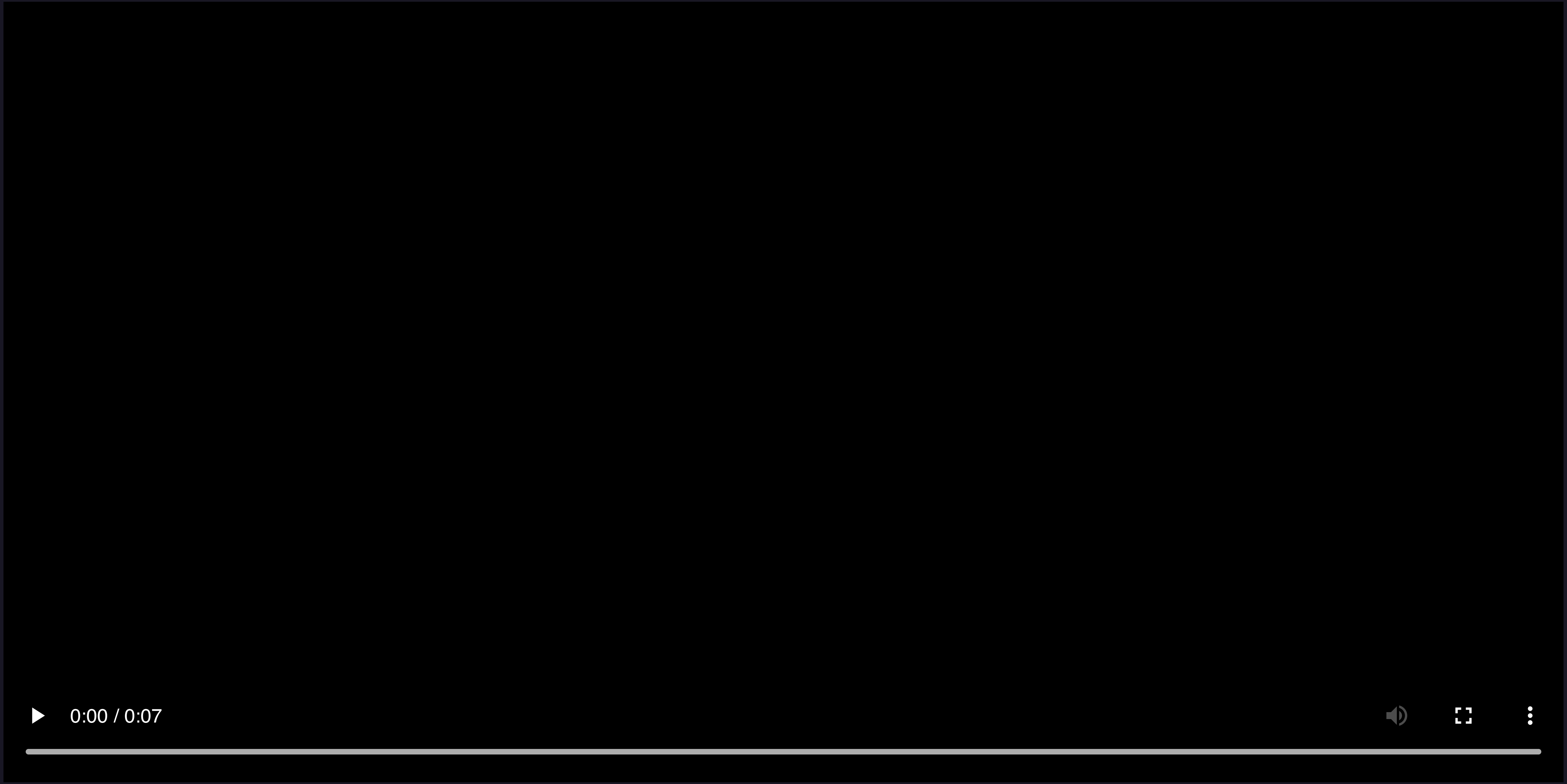
📝 **3CX Compromise via POOLRAT**
https://cloud.google.com/blog/topics/threat-intelligence/3cx-software-supply-chain-compromise

psuedocode of a library to access health

```
fn get_base() -> Option<usize> {
    return Some(_dyld_get_image_header(0) as usize) // check where it's loaded via image list in lldb
}
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
        // 0x1D9EF0 => 0x0 (ptr->player) => 0x418
        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!");

}
```

we did it folks... the hello world of game cheats



It's Just Memory for: pairs of 3x floats (XYZ/velocity), ammo, weapon choice, grenades...

what about finding *how* our player gets edited?

We know the location for our health value, they are not instructions, and we know that Rosetta 2 is running this on AArch64.

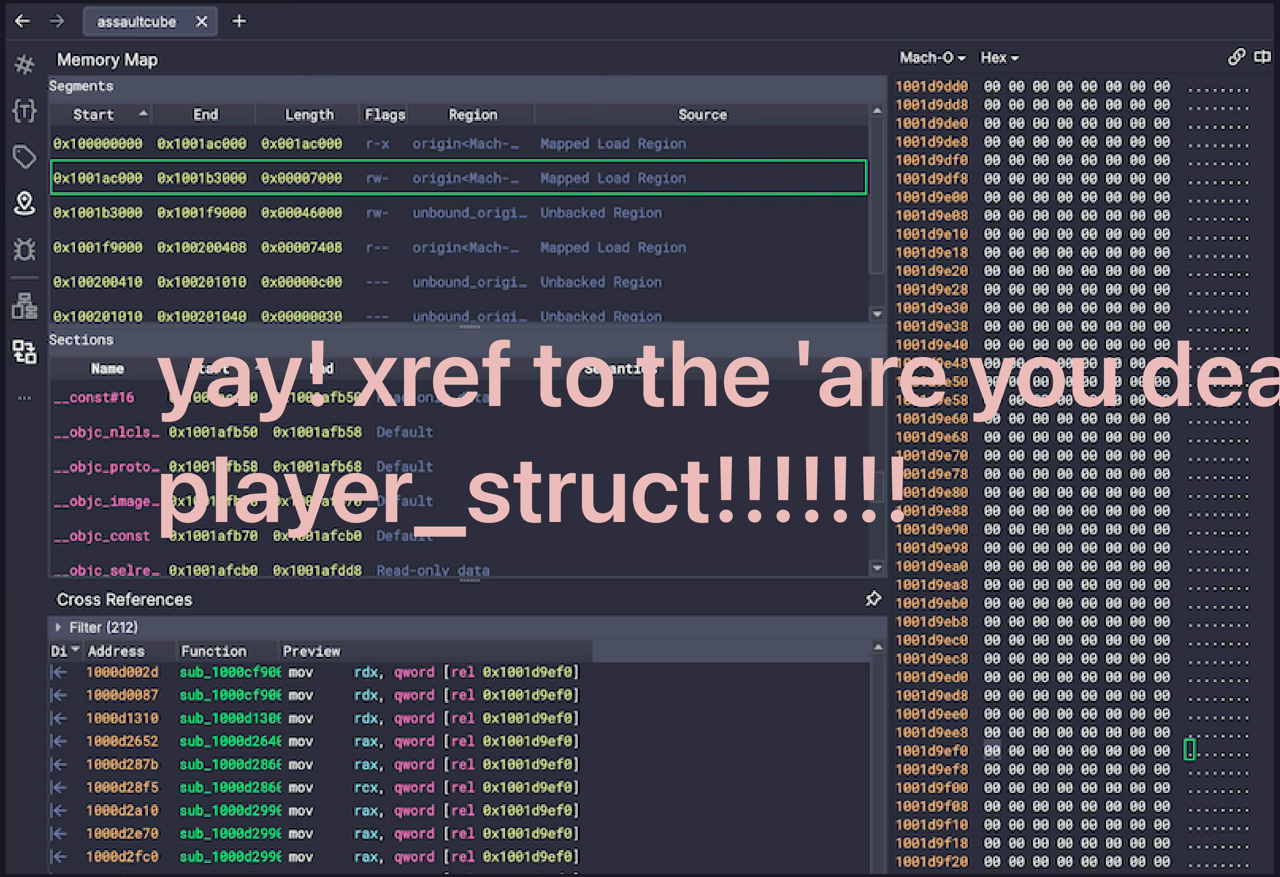
We can check in a disassembler the health location to understand how the program works in x86 via disassembly, then checking for symbol stubs in the AOT.

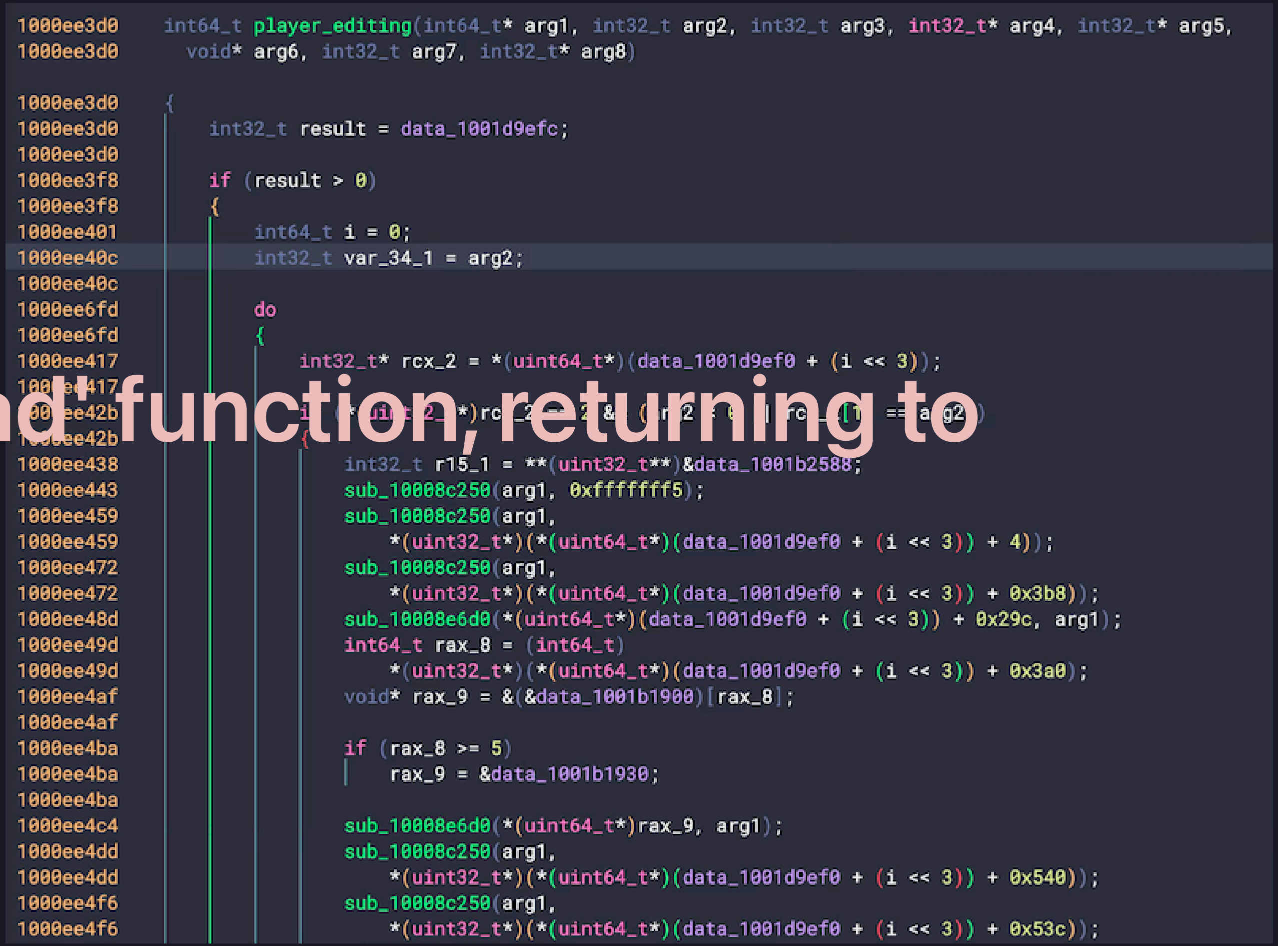
This is where it gets interesting. Sure, we can find it in our x86 code, but what about those ARM `LDR/STR` instructions in BitSlicer and LLDB?

disassembly idea...

We could use the XREF of `base_address + 0x1DEF90` (health) to understand what functions are using this value...

We can look at possible functions where values in an instantiations of player values get edited.





yay! xref to the 'are you dead' function, returning to player_struct!!!!!!!

final takeaways

amazon kindle title: How I Learned to Stop Worrying a\x9SIGILL

1. Learning RE against larger binaries is honestly way more obtuse than CTFs.
   - Often you won't get **anywhere**. We have toolchains, fuzzers et. al for this stuff if you want to break it quick.
2. This is amazing at getting you across how binaries work and what the hell you are looking at!
   - This is so informative, especially if working in an environment you work with all the time, like MacOS, on the trade-offs of fast, efficent translation of x86.
3. get curious even if you don't know what the end goal is
4. VR optimist: assume a value is something that *can be found*; the skill of finding it and if it's useful comes with time and skill

thank you!

thanks to:

- **dougallj** (https://dougallj.wordpress.com/2022/11/09/why-is-rosetta-2-fast/)
- **tsunekoh** (https://ffri.github.io/ProjectChampollion/)
- **josh goddard** (https://cloud.google.com/blog/topics/threat-intelligence/rosetta2-artifacts-macos-intrusions)
- **jai verma** (https://jaiverma.github.io)

code will be up when my github is not banned for putting memory scanning tools on a newish account...lol - will notify on mastodon