2025 BERLIN

24 - 26 September, 2025 / Berlin, Germany

# INSIDE AKIRA RANSOMWARE'S RUST EXPERIMENT

Ben Herzog

*Check Point Software Technologies, Israel*

benhe@checkpoint.com

## ABSTRACT

Rust binaries are notoriously resistant to 'full RE' that maps the binary functionality end to end. In this research we do just that – we analyse 'Akira v2', a Rust-based encryptor circulated by the prolific Akira RaaS group in early 2024. We work through the binary's opaque, sometimes barely documented, built-in types; its aggressively recursive inlining of library code; and other 'surprises' prepared for us by the Rust compiler. We explain in detail how we approached all these obstacles, and how one can methodically cross-reference Rust docs and idioms to understand code that would otherwise be impenetrable. We lay out the binary's structure, control flow, and even some of the original author's design process – and we see how all these factors translate into assembly, sometimes in surprising ways.


## INTRODUCTION

Despite the best efforts of the infosec and threat research community, reverse-engineering Rust binaries remains a not-quite solved problem. Just a few years ago, even the basics of how the Rust compiler behaves, and what output it produces for basic programming language idioms under 'toy laboratory' conditions, were shrouded in a thick fog. This initial fog has now largely dispersed thanks to many efforts, including our own contribution [1]. Efforts are now ongoing to streamline the Rust RE experience, including projects such as IDARustler [2] and 0xA11c [3], as well as *hex-rays*' own in-house Rust Analysis Plugin [4].

Earlier this year, *Talos* published [5] an update on the ongoing evolution of Akira ransomware-as-a-service (RaaS), which has become one of the more prominent players in the current ransomware landscape. According to this update, for a while in early 2024, Akira affiliates experimented with promoting a new cross-platform variant of the ransomware called 'Akira v2'. This new version was written in Rust and was capable of targeting ESXi bare metal hypervisor servers. This binary drew our attention and curiosity as a pure form RE challenge. While automation and better tooling are no doubt the future, we're still left to ask: can we in the present 'solve' a malicious Rust binary, end-to-end, like we'd all grown used to doing with C binaries?

In this research, we answered that question in the positive – after a fashion. We did fully take apart Akira v2, which is a bona fide malicious Rust binary found in the wild, and we did demonstrate several techniques and approaches that helped us get there; but we did not *at all* come to the conclusion that full, end-to-end RE of Rust binaries is *generally* 'solved' or even typically a cost-effective project to undertake, given the current state of the art. This project was carried not by the inevitable power of our approach, but by a grab-bag of key pieces of knowledge, methods, instincts, and outright strokes of luck. In this paper, we lay out some of the contents of this grab-bag so that the audience may know of them and tactically apply them where the need arises.

This is a more accessible, distilled version of the rather more dense and lengthy blog post of the same name [6], which you are welcome to peruse if you are feeling patient today.

## INITIAL FEEL FOR THE PROBLEM

The SHA256 digest of the analysed binary is:
`3298d203c2acb68c474e5fdad8379181890b4403d6491c523c13730129be3f75`

A skim of the binary, and some basic probing, yield the following insights:

1. This is a binary compiled in `Release` mode. There is no explicit flag declaring this, but `Debug` builds typically have built-in source code annotations, which one can verify are missing here with a simple `objdump --source <binary> | grep "let "`. Figure 1 shows the objdump of a test program that was compiled in Debug mode.
2. The General control flow of the binary proceeds in the following fashion: `Main` → `default_action` → `lock` → `lock_closure`, though at this time it is really not clear what is going on with the transfer of control from `lock` to `lock_closure`; we intuit from the context that it must occur, but there is no explicit call instruction that handles it.

Usually at this point we would take a hard look at our list of research questions, and ask 'how do we best answer these questions without the process taking the whole of next week?'. Unfortunately, for this particular adventure our research objective was to understand everything that happens in the binary, from beginning to end. And so we were reduced to the position of a novice malware analyst, naively pivoting to the `main` function and reading one assembly instruction after the other.

If you want to experience the full scenic route starting there, which engages even the nit-pickiest concerns such as 'what struct stores command-line arguments in memory and what operations does it support?', by all means consult the aforementioned blog post [6]. Otherwise, we provide three highlighted roadblocks that we ran across during the full RE of the binary, and the main techniques we used to dismantle those roadblocks.

```
[ben@thinkpad-x1 debug]$ objdump --source hello_rust --disassemble=_ZN10hello_rust4main17hb3488c0d977d6289E

hello_rust:     file format elf64-x86-64


Disassembly of section .init:

Disassembly of section .plt:

Disassembly of section .text:

0000000000008a10 <_ZN10hello_rust4main17hb3488c0d977d6289E>:
fn main() {
    8a10:       48 81 ec 88 00 00 00    sub    $0x88,%rsp
    let greeting = "Hello, world!";
    8a17:       48 8d 05 3d e6 03 00    lea    0x3e63d(%rip),%rax        # 4705b <_fini+0x7e7>
    8a1e:       48 89 44 24 08          mov    %rax,0x8(%rsp)
    8a23:       48 c7 44 24 10 0d 00    movq   $0xd,0x10(%rsp)
    8a2a:       00 00
    8a2c:       48 89 44 24 18          mov    %rax,0x18(%rsp)
    8a31:       48 c7 44 24 20 0d 00    movq   $0xd,0x20(%rsp)
    8a38:       00 00
    8a3a:       48 8d 4c 24 18          lea    0x18(%rsp),%rcx
    8a3f:       48 89 4c 24 78          mov    %rcx,0x78(%rsp)
    8a44:       48 8d 05 95 01 00 00    lea    0x195(%rip),%rax        # 8be0 <_ZN44_$LT$$RF$T$u20$as$u20$core..fmt..Display$GT
$3fmt17h257e6ac1e0af7716E>
    8a4b:       48 89 84 24 80 00 00    mov    %rax,0x80(%rsp)
    8a52:       00
    8a53:       48 89 4c 24 68          mov    %rcx,0x68(%rsp)
    8a58:       48 89 44 24 70          mov    %rax,0x70(%rsp)
    8a5d:       48 8b 4c 24 68          mov    0x68(%rsp),%rcx
    8a62:       48 8b 44 24 70          mov    0x70(%rsp),%rax
    println!("{}", greeting);
    8a67:       48 89 4c 24 58          mov    %rcx,0x58(%rsp)
    8a6c:       48 89 44 24 60          mov    %rax,0x60(%rsp)
    8a71:       48 8d 7c 24 28          lea    0x28(%rsp),%rdi
    8a76:       48 8d 35 eb c8 04 00    lea    0x4c8eb(%rip),%rsi        # 55368 <_ZN3std3sys4unix4args3imp15ARGV_INIT_ARRAY17h
6f84f6047c429700E+0x40>
    8a7d:       ba 02 00 00 00          mov    $0x2,%edx
    8a82:       48 8d 4c 24 58          lea    0x58(%rsp),%rcx
    8a87:       41 b8 01 00 00 00       mov    $0x1,%r8d
    8a8d:       e8 4e fe ff ff          call   88e0 <_ZN4core3fmt9Arguments6new_v117hc3d276a52e601fb9E>
    8a92:       48 8d 7c 24 28          lea    0x28(%rsp),%rdi
    8a97:       ff 15 db f4 04 00       call   *0x4f4db(%rip)        # 57f78 <_GLOBAL_OFFSET_TABLE_+0x5f0>
}
    8a9d:       48 81 c4 88 00 00 00    add    $0x88,%rsp
    8aa4:       c3                      ret
```

*Figure 1: Objdump of a test program that was compiled in Debug mode. The original source is visible.*

## DECIPHERING THE CLI AND VISUALS (OR: HOW TO THINK LIKE A CYBERCRIME DEV)

Very early into the binary's `main` function, we encountered a mysterious large struct that is initialized, and then given as an argument to several method calls. The textbook approach in such a case is to create a new struct in your disassembler, with most fields having placeholder names, and start going through one method after the other to decipher what each of those struct fields do exactly.

Anyone who has done this knows it can be rather time-consuming. We can personally tell some stories where several entire days of reverse engineering were basically about untangling the mysteries of the one Giant Struct the whole binary revolves around, and which is passed to every function as an argument (this is sometimes unaffectionately referred to as a 'God Object').

Thankfully, a lead in a different direction caught our eye: the fact that the function directly called by `main` is named `default_action`. This is rather peculiar, as the malware has only one path of execution, which is encrypting all the victim's files. This is at least worth investigating (actually, at this point we already suspected what this was about). A *Google* search for 'Rust', 'CLI', 'default_action' proved our suspicion and pointed us at the repository for Seahorse [7], a Rust CLI framework which, among its many 'hello world' examples, has the following code, under the headline 'Multiple command application':

```
1  use seahorse::{App, Context, Command};
2  use std::env;
3  fn main() {
4      let args: Vec<String> = env::args().collect();
5      let app = App::new(env!("CARGO_PKG_NAME"))
6          .description(env!("CARGO_PKG_DESCRIPTION"))
7          .author(env!("CARGO_PKG_AUTHORS"))
8          .version(env!("CARGO_PKG_VERSION"))
9          .usage("cli [name]")
10         .action(default_action)
```

```
11          .command(add_command())
12          .command(sub_command());
13      app.run(args);
14  }
```

The generic string `cli [name]` also appears as-is in the malware. As we suspected, the authors of this malware have partaken in the time-honoured tradition of modifying the 'hello, world' example exactly until it does what they want it to, and not one modification further. This one insight is the key to most (though not all) of the logic of the `main` function. In particular, it allows us to decipher the large mystery struct we encountered before; it is an instance of the `App` object supplied by the Seahorse library, and its construction proceeds exactly according to the source code we see above.

```
mov     [r15+App.name.buf], rbp ; name = akira_v2
mov     rax, [rsp+2F8h+akira_v2_str]
mov     [r15+App.name.len], rax
mov     [r15+App.name.cap], 8
and     [r15+App.author.buf], 0 ; author = None
movaps  xmm0, [rsp+2F8h+?]
movups  xmmword ptr [r15+App.author.len], xmm0
and     [r15+App.description.buf], 0 ; Description = None
movaps  xmm0, [rsp+2F8h+??]
movups  xmmword ptr [r15+App.description.len], xmm0
mov     [r15+App.usage.buf], r14 ; usage = cli[args]
mov     [r15+App.usage.len], r13
mov     [r15+App.usage.cap], 0Ah
mov     [r15+App.version.buf], r12 ; version = 2024.1.30
mov     rax, [rsp+2F8h+version_str_len]
mov     [r15+App.version.len], rax
mov     [r15+App.version.cap], 9
and     [r15+App.commands.arr], 0 ; None
movaps  xmm0, [rsp+2F8h+???]
movups  xmmword ptr [r15+App.commands.len], xmm0
lea     rax, akira_v2::default_action
mov     [r15+App.action.func], rax ; action = akirav2::default_action
and     [r15+App.action_with_result.func], 0
lea     rdi, [rsp+2F8h+flags]
mov     rax, [rdi+10h]
mov     [r15+App.flags.cap?], rax
movaps  xmm0, xmmword ptr [rdi]
movups  xmmword ptr [r15+App.flags.buf?], xmm0
```

*Figure 2: Initialization of the App object.*

The internal methods supplied by the Seahorse crate are then used to add extra arguments to the `App` object (these will later be available for the end-user as part of the CLI). At runtime, the CLI is executed via a call to `App::run`.

```
push    4
pop     rdx
push    1
pop     rbp
mov     r14, rdi
mov     ecx, ebp
call    seahorse::flag::Flag::new
lea     rdx, aSpinnerGreenEl+0DF2h ; "Start path. Default value: /vmfs/volume"...
push    28h ; '('
pop     rcx
lea     r15, [rsp+2F8h+????]
mov     rdi, r15
mov     rsi, r14
call    seahorse::flag::Flag::description
lea     r14, [rsp+2F8h+flags]
lea     rsi, [rsp+2F8h+args] ; Args
mov     rdi, r14
mov     rdx, r15
call    seahorse::app::App::flag
lea     rsi, aSpinnerGreenEl+0E1Ah ; "idstopvmvmonlyCrypt only .vmdk, .vmem, "...
lea     r15, [rsp+2F8h+args] ; Args
push    2
pop     r13
mov     rdi, r15
mov     rdx, r13
mov     ecx, ebp
call    seahorse::flag::Flag::new
lea     rdx, aBuildIdstopVms ; "Build IDStop VMsstride: "
lea     r12, [rsp+2F8h+????]
mov     rdi, r12
mov     rsi, r15
mov     rcx, rbx
call    seahorse::flag::Flag::description
lea     r15, [rsp+2F8h+args] ; Args
mov     rdi, r15
mov     rsi, r14
mov     rdx, r12
call    seahorse::app::App::flag
```

*Figure 3: Repeated calls to Seahorse methods manipulating the App object in preparation of calling Run.*

A similar insight leads to the conclusion that the authors of this malware used a minimally modified version of sample code provided by the visual progress indicator crate Indicatif [8] – 'a Rust library for indicating progress in command line applications to users ... [that] provides progress bars and spinners as well as basic color support, but there are bigger plans for the future'. It's not difficult to see what attracted the malware authors to this library, which ships visualization like that shown in Figure 4 out of the box.

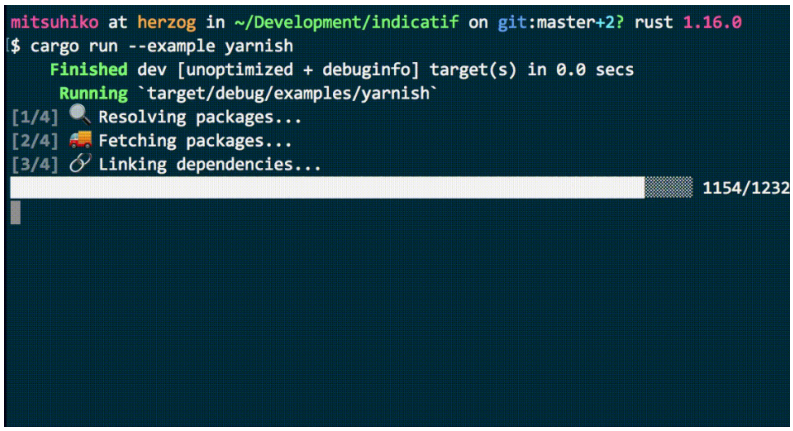


*Figure 4: Demonstration of Indicatif functionality, taken from the project repository.*

The moment we saw this gif (animated in the original readme) we knew the malware authors must have immediately reached for the corresponding source in order to tweak and modify it; the mental image of 'encrypting files' sitting there next to a pretty emoji and an ascii progress bar must have been too tempting to resist. The code (`yarnish.rs`) in the Indicatif repo is as follows:

```
let mut rng = rand::thread_rng();
let started = Instant::now();
let spinner_style = ProgressStyle::with_template("{prefix:.bold.dim} {spinner} {wide_msg}")
    .unwrap()
    .tick_chars("⠁⠂⠄⡀⢀⠠⠐⠈ ");
let m = MultiProgress::new();
let handles: Vec<_> = (0..4u32)
    .map(|i| {
        let count = rng.gen_range(30..80);
        let pb = m.add(ProgressBar::new(count));
        pb.set_style(spinner_style.clone());
        pb.set_prefix(format!("[{}/?]", i + 1));
        thread::spawn(move || {
            let mut rng = rand::thread_rng();
            let pkg = PACKAGES.choose(&mut rng).unwrap();
            for _ in 0..count {
                let cmd = COMMANDS.choose(&mut rng).unwrap();
                thread::sleep(Duration::from_millis(rng.gen_range(25..200)));
                pb.set_message(format!("{pkg}: {cmd}"));
                pb.inc(1);
            }
            pb.finish_with_message("waiting...");
        })
    })
    .collect();
for h in handles {
    let _ = h.join();
}
m.clear().unwrap();
```

Per our hunch, the strings `⠁⠂⠄⡀⢀⠠⠐⠈ ` and `{prefix:.bold.dim} {spinner} {wide_msg}` appear in the binary verbatim. On the face of it, this again could function as a key to decipher most of the malware's related logic (which this

time resides in the `default_action` invoked by `main`). Unfortunately, the disassembly did not quite match the pretty picture we were expecting. The `default_action` function has the following structure:

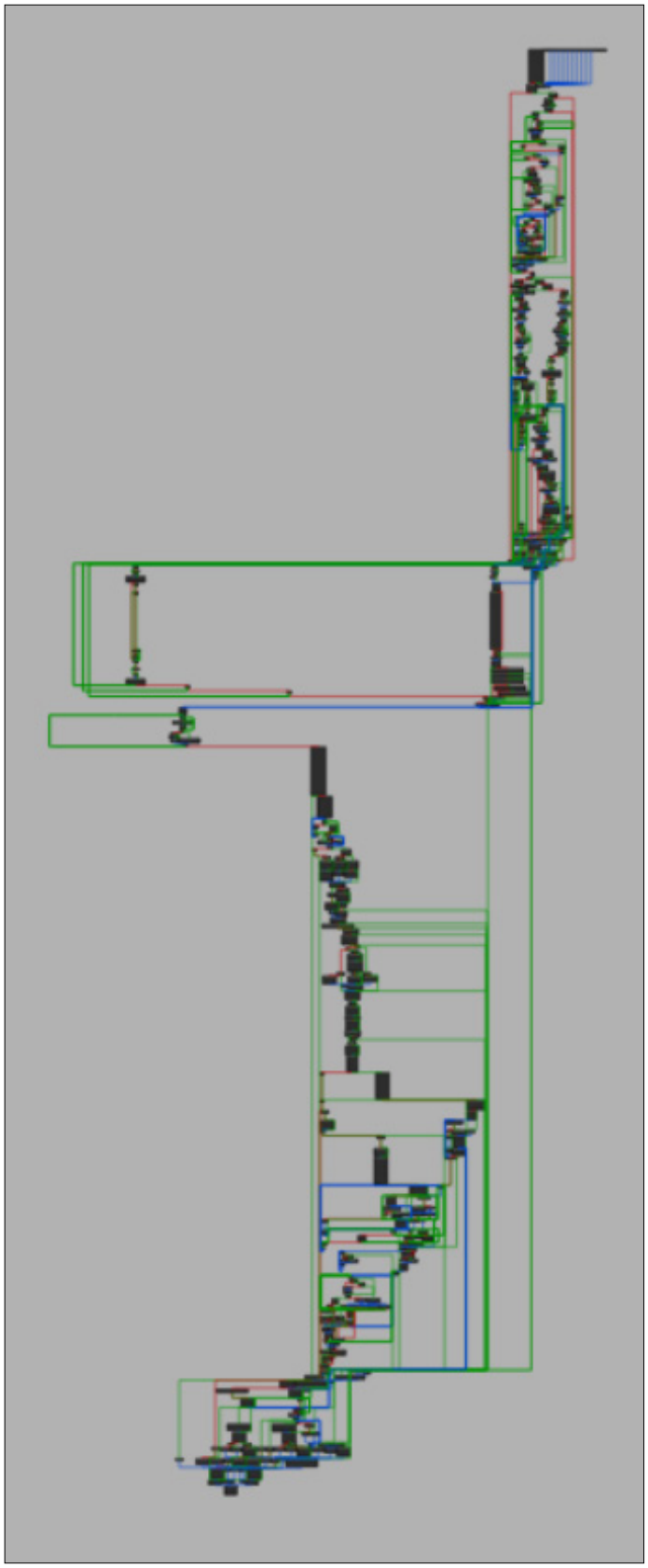

*Figure 5: Default_action basic block overview.*

Approaching this soup of basic blocks with undue optimism, we quickly ran into the following unexpected assembly:

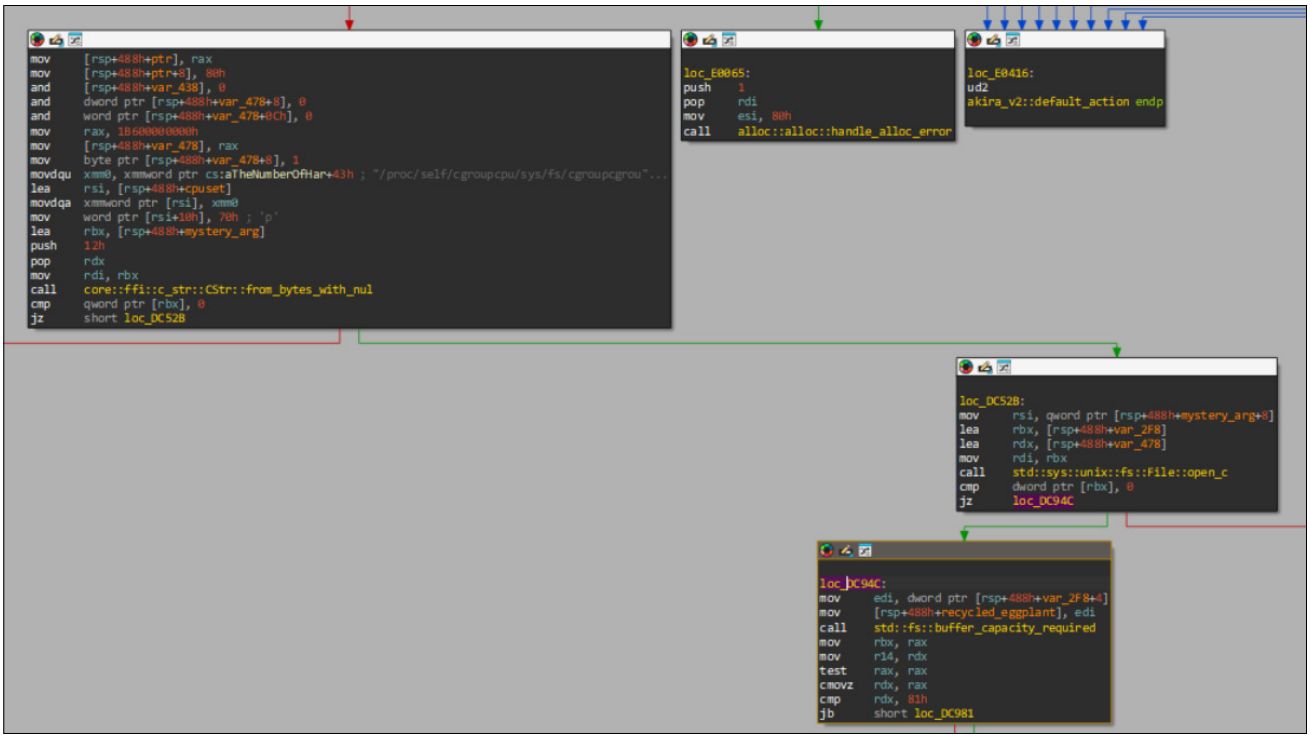


*Figure 6: First few basic blocks of* `default_action`.

The calls to `Cstr::from_bytes_with_nul` , `std::fs::buffer_capacity_required` don't feel quite at home here in this supposed user-level code penned by a malware author. If that's not enough of a clue, we next ran into an indecipherable monster basic block which goes on for over 200 instructions, and begins like this:

```
movdqa  xmm2, xmmword ptr [rsp+488h+cpuset.__bits]
movdqa  xmm6, xmmword ptr [rsp+488h+cpuset.__bits+10h]
movdqa  xmm4, xmmword ptr [rsp+488h+cpuset.__bits+20h]
movdqa  xmm3, xmmword ptr [rsp+488h+cpuset.__bits+30h]
movdqa  xmm0, xmm2
psrlw   xmm0, 1
movdqa  xmm1, cs:xmmword_385F0
pand    xmm0, xmm1
psubb   xmm2, xmm0
movdqa  xmm0, cs:xmmword_38600
movdqa  xmm5, xmm2
pand    xmm5, xmm0
psrlw   xmm2, 2
pand    xmm2, xmm0
paddb   xmm2, xmm5
movdqa  xmm7, xmm2
psrlw   xmm7, 4
paddb   xmm7, xmm2
movdqa  xmm2, cs:xmmword_38610
pand    xmm7, xmm2
pxor    xmm8, xmm8
psadbw  xmm7, xmm8
```

*Figure 7: Default_action basic block featuring optimized instructions.*

This was as far as 'get into the malware author's head and guess what source code they copied' got us on its own. There seemed to be no obvious way to connect the dots: what purpose does this jumble of `paddb`s and `movdqa`s serve? What part of the straightforward `yarnish.rs` code compiles into it? How, why? A piece of the puzzle was missing.

## DECIPHERING THE BASIC BLOCK SOUP (OR: HOW TO THINK LIKE AN OPTIMIZER)

In any other part of tech history, this is where we would have backed off and tried to come at the problem from literally any other angle than 'actually parse this assembly'. Fortunately (or unfortunately), we are living in the current part of tech history, and there is a relatively new thing that we can do. Brace yourselves, you know what's coming:

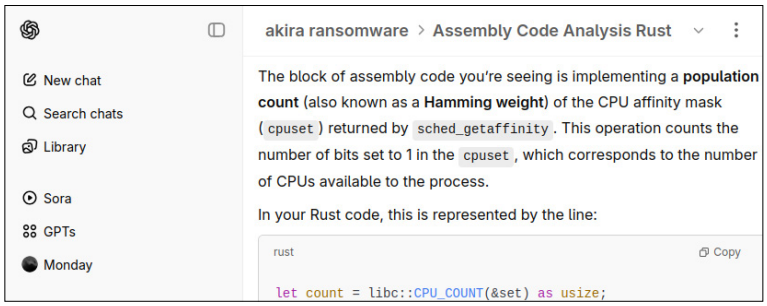


*Figure 8: Passing the problem on to our esteemed colleague, OpenAI o3.*

But where in the `yarnish.rs` code does any piece of logic need to compute the number of available CPUs? It doesn't. Thankfully, we were fresh out of a long marathon of seeing through the malware developer's eyes, and soon enough the solution to this conundrum occurred to us. The developer in charge of creating this malware must have noticed that this code hard codes the number of threads to `4`, which they correctly noted is nice for a 'hello world' example but not befitting of production code:

```
let handles: Vec<_> = (0..4u32).map(|i| {
                ..
        thread::spawn(move || { .. })
        ..
    })
    .collect();
..
```

So, first of all, they added Seahorse code allowing the malware operator to control the number of launched threads directly, using the following command-line argument:

```
--threads <int> Number of threads (1-1000)
```

Then they (again, correctly) asked themselves, 'but what if the operator doesn't supply a value for this flag? What is the right thing to do?' then, after a moment of thought, they amended it to the following:

```
--threads <int> Number of threads (1-1000). Default: number of logical CPU cores
```

And modified the source to something like:

```
let threads : u32 = match threads_flag_value {
    Some(n) => n,
    None => available_parallelism().unwrap()
}
...
let handles: Vec<_> = (0..threads)
    .map(|i| ..
```

Since `available_parallelism` calls the `POPCOUNT` macro eventually, somewhere down the call stack, this at least takes care of the ontological mystery of where the `POPCOUNT` macro could even have come from in principle. It does leave open the question of what the expanded macro was doing there inside a function ostensibly compiled from user-authored code, but now, with the question laid bare like that, the answer should be clear: inlining.

But let's take a moment to appreciate the *degree* of inlining we are talking about here. `Available_parallelism`, when compiled for *Linux* targets, invokes `cgroups::quota`. Under the hood, depending on the version of the cgroup api, this function can call either one of two subfunctions:

```
pub(super) fn quota() -> usize {
    let _: Option<()> = try {
        let mut buf = Vec::with_capacity(128); // find our place in the cgroup hierarc
        File::open("/proc/self/cgroup").ok()?.read_to_end(&mut buf).ok()?;

        quota = match version {
            Cgroup::V1 => quota_v1(cgroup_path),
            Cgroup::V2 => quota_v2(cgroup_path),
        };
    };
```

```
11      quota
12  }
```

Each of those subfunctions separately calls `libc::CPU_COUNT`, which is a thin wrapper for `CPU_COUNT_S`, which in turn has the following implementation:

```
1  pub fn CPU_COUNT_S(size: usize, cpuset: &cpu_set_t) -> c_int {
2      let mut s: u32 = 0;
3      let size_of_mask = core::mem::size_of_val(&cpuset.bits[0]);
4
5      for i in cpuset.bits[..(size / size_of_mask)].iter() {
6          s += i.count_ones();
7      }
8      s as c_int
```

Where `count_ones()` is a wrapper for an LLVM intrinsic `popcnt64`, which is what gets expanded into a modest assembly macro, with the entire loop `for i in cpuset.bits` unrolled – this is what results in the hefty size of that basic block.

We thought understanding what was going on in `default_action` would involve a fearless drill-down into weird assembly. Instead, it became an exercise in archaeology, where we traced each basic block in the function's compiled final form to the original library source from which it had originated:

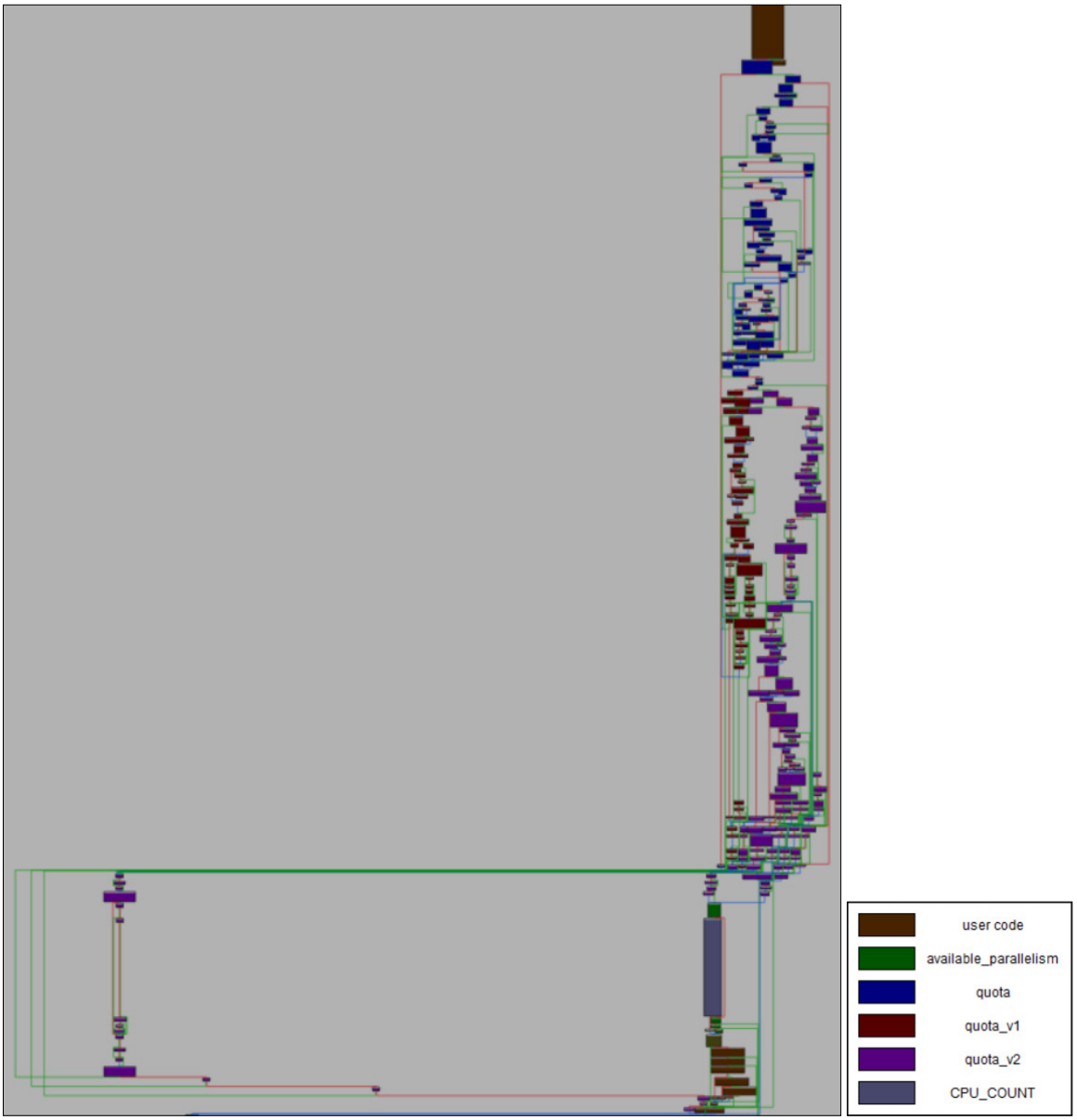


*Figure 9: Breakdown of default_action by source of functionality.*

In other words, when we first laid eyes on that complex basic block, we were unwittingly looking at an unrolled loop that lived *five levels deep* into library code hierarchy. We even have basic blocks from different levels into the call hierarchy side-by-side, creating the false visual impression that they all belong to the same function. This is a situation that may theoretically have occurred before under the regime of other programming languages, but until this analysis, we've seen very few examples of anything remotely like it happening in practice. It's a perfect storm of surface-level illegibility, caused by no packer, no obfuscation, no virtualization; just Rust's rich standard library and ecosystem of third-party crates, combined with the temperament of its compiler.

## DECIPHERING THE PARALLELISM MACHINERY (OR: HOW TO DEEP DIVE INTO THE DOCS)

Among the mysteries remaining latent in the binary at that point, one of the biggest was the transfer of control from `lock` to `lock_closure`. It involved no direct call instruction, and the inclusion of 'closure' in the latter again hinted at some sort of default setup or boilerplate. Based on the ransomware's support for parallelism, it became pretty clear that the transfer of control was done by spawning threads. But this wasn't your grandparents' `CreateThread` call; it was wrapped in layers of Rust machinery that we had to decipher if we wanted to claim we'd understood the binary properly.

Unfortunately, to decipher these layers, we must perform an action popularly known to be an absolutely last resort: read the docs [9] (and, in several cases, the Rust lang source directly).

Our starting point is the call from within the `lock` function to `std::sys::unix::thread::Thread::new`. This call appears here:

```
loc_D4067:
push    58h ; 'X'
pop     rdi             ; size
push    8
pop     rsi             ; align
call    alloc::alloc::exchange_malloc
mov     [rax], r14
mov     [rax+8], rbx
mov     [rax+10h], r13
movups  xmm0, [rsp+2D8h+now]
movups  xmmword ptr [rax+18h], xmm0
mov     rcx, qword ptr [rsp+2D8h+var_278]
mov     [rax+28h], rcx
mov     rcx, [rsp+2D8h+var_2B0]
mov     [rax+30h], rcx
mov     rcx, [rsp+2D8h+var_148]
mov     [rax+38h], rcx
mov     rcx, [rsp+2D8h+var_2B8]
mov     [rax+40h], rcx
mov     rcx, [rsp+2D8h+var_2A8]
mov     [rax+48h], rcx
mov     rcx, [rsp+2D8h+var_150]
mov     [rax+50h], rcx
mov     rdi, r12
mov     rsi, r15
mov     rdx, rax
lea     rcx, locker_thread_vtable
call    std::sys::unix::thread::Thread::new
cmp     [rsp+2D8h+var_1D8], 0
mov     r15, [rsp+2D8h+var_1D0]
jz      short loc_D4109
```

*Figure 10: Thread::new call.*

The implementation of this function is found in `thread.rs`, a part of the Platform Abstraction Layer (PAL) which allows the Rust compiler to splice code specific to the target platform (e.g. *Windows*, *Linux*) into the compiled binary. It has the prototype:

```
pub unsafe fn new(stack: usize, p: Box<dyn FnOnce()>) -> io::Result<Thread>
```

This implies that under the hood, the creation of the new thread is handled using a dynamic dispatch object with a type known at runtime – this is a `Box<dyn Trait>`, which you might also know as a 'struct with a vtable pointer', and in this case the trait is `FnOnce`, meaning the struct in question needs to be able to call a function. This is nice to know, but it doesn't answer the question of *what* underlying type is being used here, and how it is represented in memory byte-wise.

To answer that, we need to look at the implementation of `thread::spawn`, which is what the user-authored source calls originally (`yarnish.rs` line 13); the direct call to `Thread::new` that we started from is an artifact of inlining. `Thread::spawn` proceeds by instantiating an object called a `Builder` – its implementation can be found in the `std::thread` module:

```
1  pub struct Builder {
2      name: Option<String>,
3      stack_size: Option<usize>,
4  }
```

This object has a method called `spawn`, which has the following prototype:

```
1  pub fn spawn<F, T>(self, f: F) -> io::Result<JoinHandle<T>>
2      where
3          F: FnOnce() -> T,
4          F: Send + 'static,
5          T: Send + 'static,
```

The argument `f` implements `FnOnce`; that is what the thread will execute. This argument is wrapped in some padding to create a new function (`main`, the 'shim') that performs some book-keeping surrounding the launch of the `f` logic, constructing a closure (named `main` in the source). This closure is the 'mystery object' that implements `FnOnce` and gets passed to `Thread::new` (line 15).

```
1   let main = move || {
2               if let Some(name) = their_thread.cname() {
3                   imp::Thread::set_name(name);
4               }
5
6               crate::io::set_output_capture(output_capture);
7
8               let f = f.into_inner();
9               set_current(their_thread);
10              let try_result = panic::catch_unwind(panic::AssertUnwindSafe(|| {
11                  crate::sys::backtrace::__rust_begin_short_backtrace(f)
12              }));
13  ...
14  Ok(JoinInner {
15              native: unsafe { imp::Thread::new(stack_size, main)? },
16              thread: my_thread,
17              packet: my_packet,
18          })
```

Again, the entire call chain that leads to `Thread::new` is spliced directly into the original user-authored `lock` function (`thread::spawn` -> `Builder::spawn` -> `Builder::spawn_unchecked` -> `Builder::spawn_unchecked_`), resulting in the 'several layers of inlining side-by-side' phenomenon we'd seen earlier.

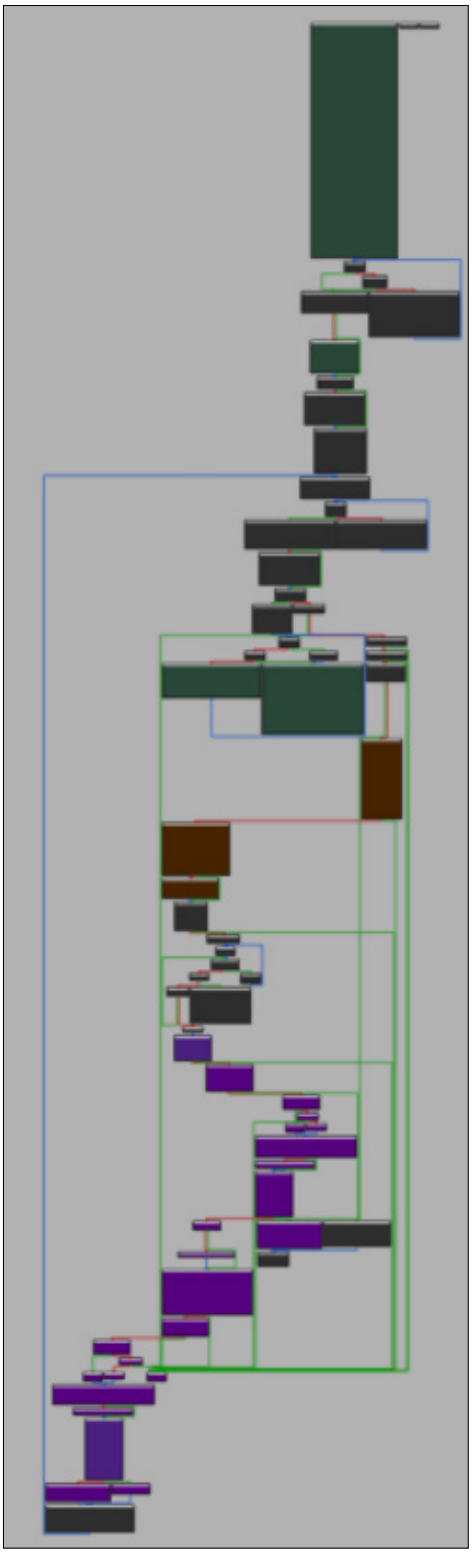

*Figure 11: Builder::spawn call chain inlined into user code.*

The vtable and the shim:

```
locker_thread_vtable dq offset builder_drop_in_place
                                        ; DATA XREF: akira_v2::lock::lock+826↑o
                     dq 58h
                     dq 8
                     dq offset locker_thread_vtable_shim
```

*Figure 12: The Box<dyn FnOnce()> object in memory.*

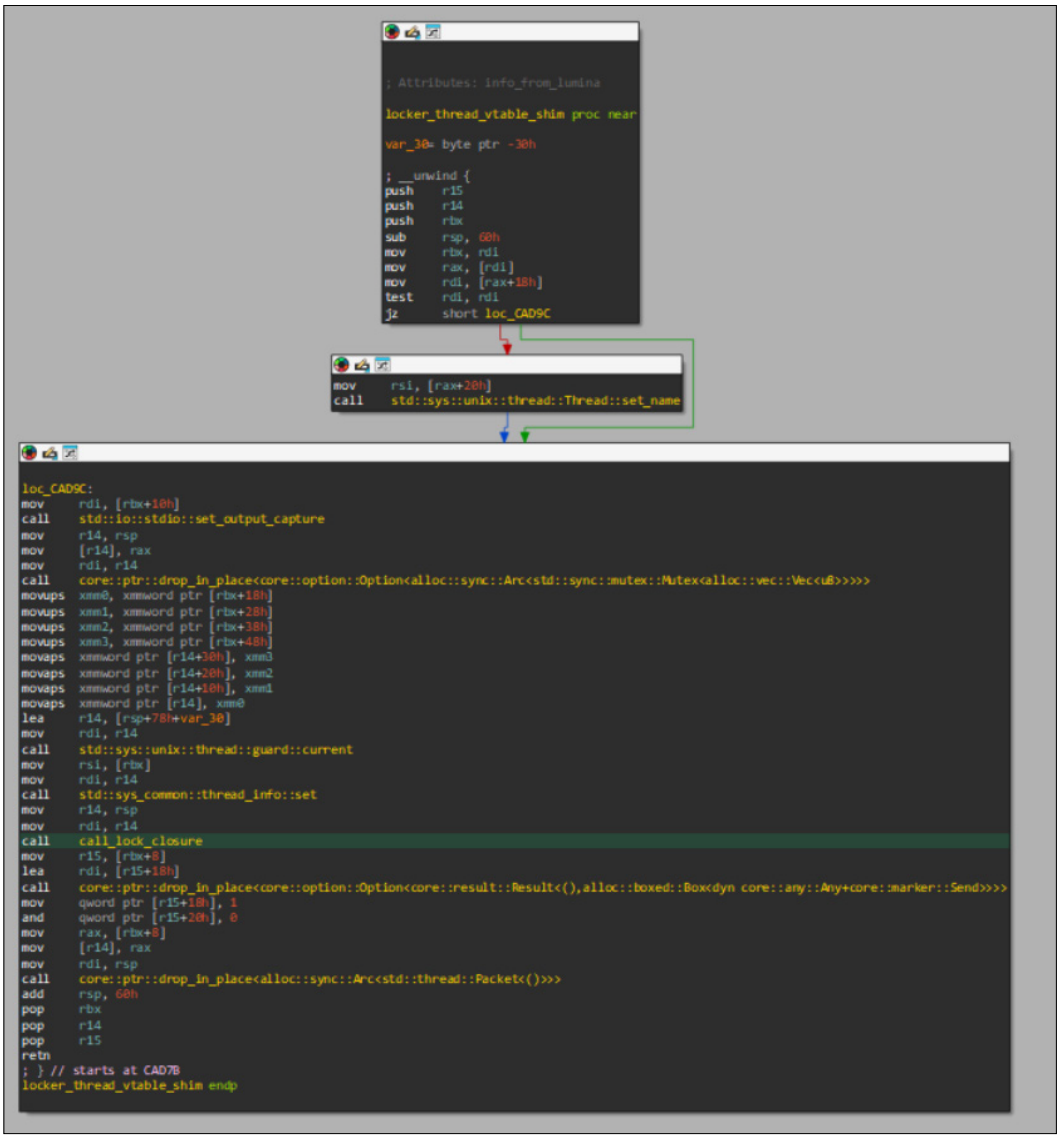


*Figure 13: Completed vtable shim, the source of which we earlier saw (let main = move ||...).*

This unconventional detour took us through the Rust sources, and its reward is that we now understand the underlying objects that are constructed and operated on when the user calls `spawn`, and the entire function call chain that results. In a more convenient universe, we could relegate all of this to a separate subject called 'Rust standard library internals', which would be of no concern to the median reverse engineer. Unfortunately, the compiler's aggressive inlining of library code has made this knowledge our concern, whether we like it or not.

## CONCLUSION

The combination of being cross-platform, having good ergonomics, and the availability of many ready-made libraries have made Rust language a surprisingly strong contender in the malware landscape despite its notoriously steep learning curve. To experiment with Rust, malware authors by necessity have left behind many of their favourite anti-analysis tricks; unfortunately for reverse engineers, the very nature of the language combined with the compiler's drive to optimize its output can often result in forbidding disassembly that seems to put pressure on the analyst to look at literally anything else. Apart from the counter-intuitive effects of generics and monomorphization (which we have covered in [1]), when analysing a Rust binary, we need to contend with aggressive inlining that, as it turns out, can easily reach five levels deep into library code.

One of the most crucial tools in our arsenal during this research was the authors' reliance on ready-made boilerplate code associated with each of the third-party libraries they used. It should go without saying that we won't always be so lucky and that the already troubled ecosystem of RE plugins and kludges now has an additional gap to bridge. The situation seems to call for automatic tooling that can isolate and identify spliced in-line code, even as it recursively contains other spliced in-line code, and this is a formidable challenge – especially in an environment where analysts didn't have it down to an exact science 100% of the time identifying pretty, encapsulated functions with well-defined calling conventions, either.

With all that said, we can say that our experience with Akira has provided an invaluable contribution to our ability to reverse engineer Rust binaries in general. In retrospect, a lot of the difficulty posed by these obstacles boiled down to their out-of-left-field nature and their shock value. Two months after this research concluded, when the next Rust binary landed on our doorstep accompanied by some urgent research questions, the process was much more streamlined than it could have been otherwise. If nothing else, we can be grateful to the Akira gang for the educational value of their handiwork.

## REFERENCES

[1] Herzog, B. Rust Binary Analysis, Feature by Feature. Check Point 6 June 2023. https://research.checkpoint.com/2023/rust-binary-analysis-feature-by-feature/.

[2] r3dhun9 / IDARustler. https://github.com/r3dhun9/IDARustler.

[3] juanandresgs / Proj-0xA11c. https://github.com/juanandresgs/Proj-0xA11c.

[4] Skochinsky, I. Rust analysis plugin tech preview. Hex-rays. 26 June 2023. https://hex-rays.com/blog/rust-analysis-plugin-tech-preview.

[5] Nutland, J.; Szeliga, M. Akira ransomware continues to evolve. Cisco Talos. 21 October 2024. https://blog.talosintelligence.com/akira-ransomware-continues-to-evolve/.

[6] Check Point Research. Inside Akira Ransomware's Rust Experiment. 3 December 2024. https://research.checkpoint.com/2024/inside-akira-ransomwares-rust-experiment/.

[7] ksk001100 / seahorse. https://github.com/ksk001100/seahorse.

[8] console-rs / indicatif. https://github.com/console-rs/indicatif.

[9] Rust Documentation. doc.rust-lang.org.