

Inside Akira Ransomware's Rust Experiment

 research.checkpoint.com/2024/inside-akira-ransomwares-rust-experiment/

December 3, 2024

Executive Summary

- Check Point Research analyzed the construction and control flow of Akira ransomware's Rust version that circulated in early 2024, which has specific features uniquely targeting ESXi server. Our analysis demonstrates how Rust idioms, boilerplate code, and compiler strategies come together to account for the complicated assembly.
- The report outlines principles to follow when analyzing ITW Rust binaries in general.
- We present an analysis of the design strategies used by the malware's authors, as indicated by the assembly and parts of the reconstructed source code.

Introduction

Earlier this year, Talos [published](#) an update on the ongoing evolution of Akira ransomware-as-a-service (RaaS) that 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.

Executables written in Rust have a reputation for being particularly challenging to reverse-engineer. While it is often possible to answer specific research questions, such as "What file extensions are being targeted?" or "What encryption method is being used?", this is usually achieved by finding ways to circumvent a comprehensive analysis at the assembly level. This is, of course, a pragmatic and cost-effective approach. However, if we set it aside for a moment, we might glean some interesting insights.

While we do answer many of the above-mentioned 'usual' research questions about a given ransomware strain in this text, this is not the focus nor the main objective here.

Our earlier research, [Rust Binary Analysis Feature by Feature](#), was conducted in a laboratory setting. We examined the language features as they appear in [Klabnik's and Nichols' "The Rust Programming Language"](#), and observed the idioms the Rust compiler uses to implement them in assembly. We controlled the source code and took care to compile in **Debug** mode so that the compiler would not complicate our lives with surprising optimizations.

In this publication, we engage that theory with the real world: an actual Rust malware that uses various standard library and third-party crates, and was compiled in **Release** mode. We note the design choices made by the malware authors and the involved Rust constructs and see how both translate into assembly – sometimes in surprising ways that lab work did not prepare us for. Our main aim is to break through that technical barrier and reach a capital-U Understanding of this Rust binary and its control flow.

Overview

We earlier noted that the binary was compiled in **Release** mode; however, there isn't a clear flag indicating this. We assume this since firstly, some compiler optimizations we discuss later definitely should not be present in a **Debug** build; and secondly, the binary debug information has no source lines, which are typically included by default in a **Debug** build, as seen below (a good way to test for these is by using the command `objdump --source <binary> | grep "let "`).

```
$ 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:
00000000000008a10 <_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$a$s$u20$core..fmt..DisplayGT
3fmt17h257e6acle0af7716E>
  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.

The ransomware's control flow, from its root to the encryption logic, is as follows:

Main -> **default_action** -> **lock** -> **lock_closure**

Broadly speaking, `Main` parses arguments, `default_action` determines program behavior and collects targeted files, and `lock` is a wrapper that launches threads that carry out the actual encryption logic that lives in `lock_closure`.

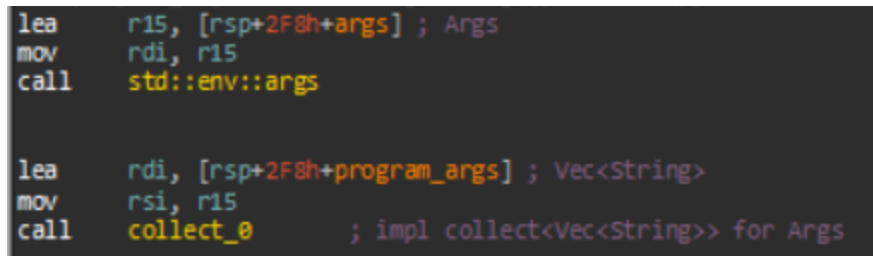
Maybe you are unpleasantly surprised by how cyber criminals are so far gone that the “default action,” in their eyes, is destroying your machine and encrypting all the files on it. However, as we will later see, there is a perfectly reasonable explanation for why this is so.

We will now sequentially analyze each of the above-mentioned functions and observe how control flows from one to the next.

Main function

Command Line Arguments

This malware has a full-fledged CLI, and as a consequence, command-line arguments play a significant role in the way it functions. The first thing the malware does is obtain them (the original source is `let program_args: Vec<String> = args().collect()`).



```
lea    r15, [rsp+2F8h+args] ; Args
mov     rdi, r15
call    std::env::args

lea     rdi, [rsp+2F8h+program_args] ; Vec<String>
mov     rsi, r15
call    collect_0          ; impl collect<Vec<String>> for Args
```

Figure 2. Compiled assembly of `args().collect()`

This brings us to our first natural question: How are program arguments stored in memory?

According to the official documentation, `std::env::args` returns `std::env::Args`, a type that implements `Iterator<Item=String>`, meaning that it implements a function `next() -> Option<String>`. However, this does not guarantee what the underlying implementation looks like. (In fact, Rust supports dynamic dispatch, allowing some objects to implement a trait even when the underlying implementation is not known at compile time — although this is not the case here, we will see this feature in action later.)

`std::env::Args` is implemented as a wrapper for another type called `ArgsOs`. This type, in turn, is implemented using `sys::args::Args`, a type returned by `args_os()`, which calls the function `sys::args::args()`. At this point, you may dutifully look for `sys::args::args` in the Rust standard library sources. You will not find it there; since this code is OS-dependent, it is part of the Platform Abstraction Layer (PAL), which swaps in the correct code at compile time depending on the targeted OS. So, the implementation is in `std::src::sys::pal::unix::args`. It first invokes an arcane function (`argc_argv`,

implementation here) that directly pulls these values via calling the C FFI. Then it manually, painstakingly converts this tuple into a `Vec<OsString>`. The complete code of `sys::args::args` is included below.

```
pub fn args() -> Args {
    let (argc, argv) = imp::argc_argv();
    let mut vec = Vec::with_capacity(argc as usize);
    for i in 0..argc {
        let ptr = unsafe { argv.offset(i).read() };
        if ptr.is_null() {
            break;
        }
        let cstr = unsafe { CStr::from_ptr(ptr) };
        vec.push(OsStringExt::from_vec(cstr.to_bytes().to_vec()));
    }
    Args { iter: vec.into_iter() }
}
```

So, if we squint at that last line and put together all the pieces of the puzzle, we can conclude that `Args` under the hood has the output type of `impl into_iter for Vec<OsString>`. However, figuring out the actual type still requires navigating murky waters. The `into_iter` method is provided by the `IntoIterator` trait (`std::iter::IntoIterator`), which has a type parameter called `IntoIter`. Meanwhile, the documentation for `std::vec::Vec` lists three different implementations of `impl into_iter for Vec`:

```
[+] impl<'a, T, A> IntoIterator for &'a Vec<T, A> 1.0.0 · source
  where
    A: Allocator,

[+] impl<'a, T, A> IntoIterator for &'a mut Vec<T, A> 1.0.0 · source
  where
    A: Allocator,

[-] impl<T, A> IntoIterator for Vec<T, A> 1.0.0 · source
  where
    A: Allocator,

[-] fn into_iter(self) -> <Vec<T, A> as IntoIterator>::IntoIter source
```

Creates a consuming iterator, that is, one that moves each value out of the vector (from start to end). The vector cannot be used after calling this.

Examples

```
let v = vec!["a".to_string(), "b".to_string()];
let mut v_iter = v.into_iter();

let first_element: Option<String> = v_iter.next();

assert_eq!(first_element, Some("a".to_string()));
assert_eq!(v_iter.next(), Some("b".to_string()));
assert_eq!(v_iter.next(), None);
```

```
[_] type Item = T source
```

The type of the elements being iterated over.

```
[_] type IntoIter = IntoIter<T, A> source
```

Which kind of iterator are we turning this into?

Figure 3. Implementations of `into_iter` for `Vec`.

One of these returns a type named `IntoIter` (`std::vec::IntoIter`), unrelated to the associated type parameter `IntoIter` of `std::iter::IntoIterator` – they *happen* to have the same name. At this point, we will mercifully note that the third implementation, highlighted in yellow above, is the one invoked by `sys::args::args` (the other two are for a *reference* to a vector). This is to say, `Args` are kept in memory as an `IntoIter<OsString>`.

We still need to determine the implications of this, concrete-byte-wise. The `vec::IntoIter` documentation can be found [here](#), and this structure is defined as follows:

```
pub struct IntoIter<T> {
    buf: NonNull<T>,
    cap: usize,
    start: *const T,
    end: *const T,
}
```

An `OsString` is implemented as a `sys::os_str::Buf`, and its precise implementation varies between operating systems. In Linux it is represented as a standard three-field `Vec<u8>` consisting of `length`, `buf`, and `capacity`. In contrast, for Windows the compiler uses a custom solution called UTF8 encoding – “a hack intended to be used internally in self-

contained systems with components that need to support potentially ill-formed UTF-16 for legacy reasons". The relevant type, `Wtf8Buf`, includes an additional field, `is_known_utf8`, bringing the total number of fields to four.

Looking at the `args` variable in memory right after `std::env::args` is called, we can confirm the structure of the `IntoIter<OsString>`:

```

.text:00007FF7D1CD180C lea     rsi, [rsp+148h+args]
.text:00007FF7D1CD1814 mov     rcx, rsi
.text:00007FF7D1CD1817 call    std_env_args ; std [rsp+148h+args]=[debug005:000000DEB1DFFC48]
.text:00007FF7D1CD181C lea     rcx, [rsp+148h+var_10] ; dq offset args_buf ; buf
.text:00007FF7D1CD1821 mov     rdx, rsi ; dq 8 ; cap
.text:00007FF7D1CD1824 call    _ZN73_$LT$std_env_ ; dq offset args_buf ; start
.text:00007FF7D1CD1829 mov     rdi, [rsp+148h+var_10] ; dq offset unk_274AD877090 ; end

```

Figure 4. `IntoIter<OsString>` structure in memory.

Below is the internal buffer of OsStrings for the arguments which we ran a test Windows program compiled from the same source: (the value 0x274ad87f101 is an artifact of the WTF8 encoding)

```

dq offset args_buf ; buf
dq 8 ; cap
dq offset args_buf args_buf dq offset aCidaWindbgTmpA
dq offset unk_274 ; DATA XREF: debug005:000000DEB1DFFC48↑o
db 0 ; debug005:000000DEB1DFFC58↑o
db 0 ; "C:\\ida_windbg_tmp\\argcollect.exe"
db 0DFh dq 20h
db 0B1h dq 20h
db 0DEh dq 274AD875901h
db 0 dq offset aWeRe ; "we're"
db 0 dq 5
db 0 dq 5
db 0 dq 274AD87F101h
db 0 dq offset aNo ; "no"
db 0 dq 2
db 0 dq 2
db 0 dq 274AD87F101h
db 0 dq offset aStrangers ; "strangers"
db 0 dq 9
db 0 dq 9
db 0 dq 274AD87F101h
db 0 dq offset aTo ; "to"
db 0 dq 2
db 0 dq 2
db 0 dq offset unk_274AD87F101
db 0 dq offset aLove ; "love"
dh 0 dq 4
dq 4
dq offset unk_274AD877C01

```

Figure 5. `OsStrings` in sequence, pointed at by the `buf` pointer of the `IntoIter<OsString>`.

CLI Parser

After the command line arguments are collected, many writes to stack memory are followed by a call to a function called `default_action`. We will explore the details of this function later, but its name raised our suspicions that it might be a sample or boilerplate code of some

kind. After all, it is hard to imagine someone deliberately naming a function “default action”.

A quick web search for this function name locates the [seahorse](#) crate, which is a CLI framework. In one of its numerous “hello world” examples, we find the following code:

```
use seahorse::{App, Context, Command};
use std::env;

fn main() {
    let args: Vec<String> = env::args().collect();
    let app = App::new(env!("CARGO_PKG_NAME"))
        .description(env!("CARGO_PKG_DESCRIPTION"))
        .author(env!("CARGO_PKG_AUTHORS"))
        .version(env!("CARGO_PKG_VERSION"))
        .usage("cli [name]")
        .action(default_action)
        .command(add_command())
        .command(sub_command());

    app.run(args);
}

fn default_action(c: &Context) {
    println!("Hello, {:?}", c.args);
}
```

Now we know where the name [default_action](#) came from. The malware sources must have contained this code block or something a lot like it (the string `cli [name]` also appears as-is). It is clear that the struct being initialized with all those stack writes is a seahorse [App](#). This struct is implemented in the following way:

```
pub struct App {
    pub name: String,
    pub author: Option<String>,
    pub description: Option<String>,
    pub usage: Option<String>,
    pub version: Option<String>,
    pub commands: Option<Vec<Command>>,
    pub action: Option<Action>,
    pub action_with_result: Option<ActionWithResult>,
    pub flags: Option<Vec<Flag>>,
}
```

This allows us to decipher the initialization of the [App](#) struct:

```

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 6. Initialization of the `App` object.

The assembly then proceeds to initialize the long list of supported CLI flags (the below image accounts for the first two):

```

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 7. Repeated calls to seahorse methods manipulating the `App` object in preparation for calling `run`.

Then finally calls `run`, transferring control to `default_action`.

The Immediate Puzzle

This is the function that is launched by the `seahorse` boilerplate code we saw earlier. When we look at its disassembly, we are immediately greeted by a startling sight:

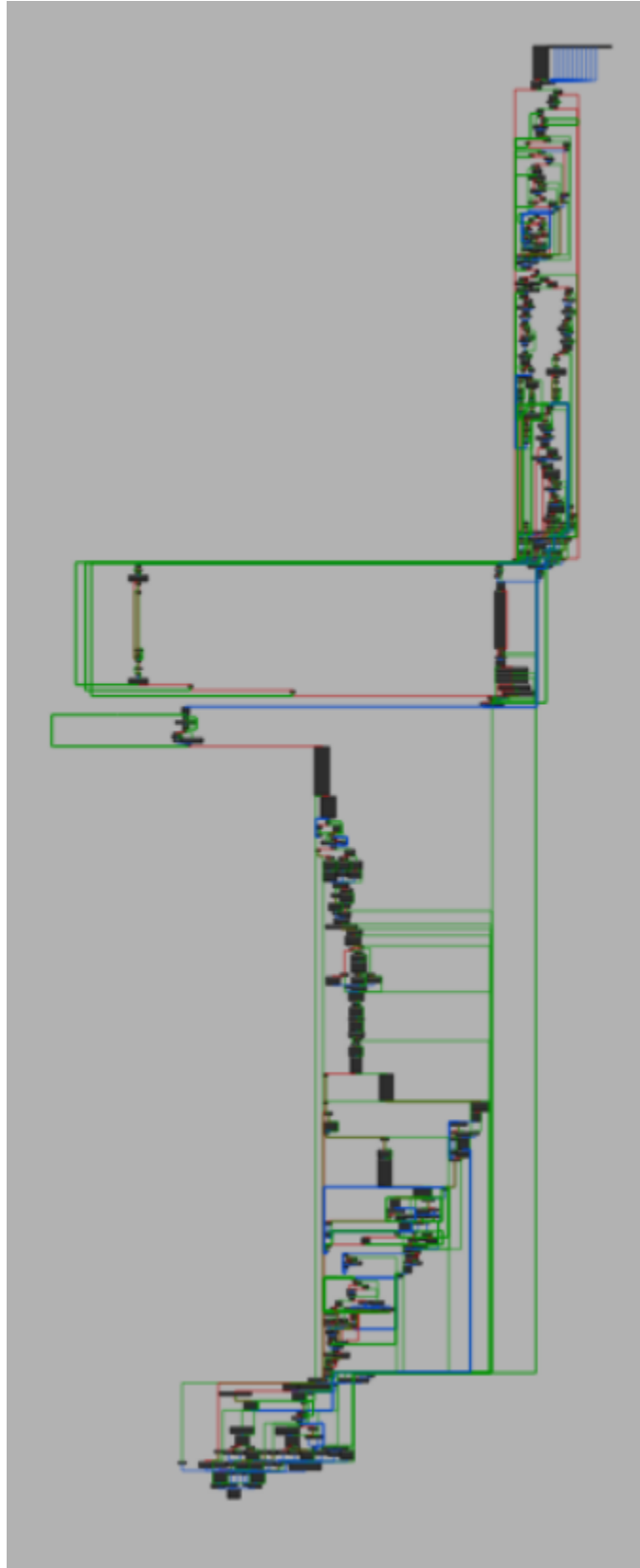


Figure 8. `default_action` overview.

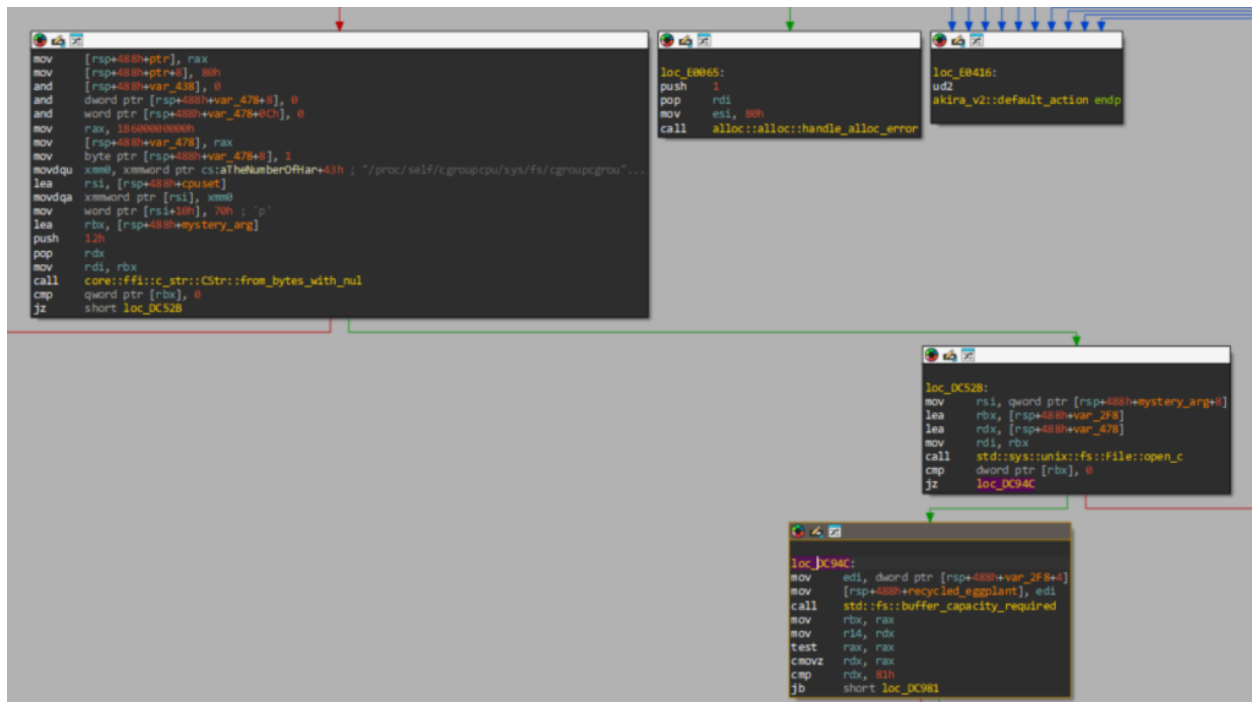


Figure 9. `default_action` first few basic blocks.

The malware authors apparently implement their desired functionality by... directly calling the C FFI function `from_bytes_with_nul` and then `std::fs::buffer_capacity_required`? This is unexpected. There are several dozen more eyebrow-raising basic blocks of this kind — the entire upper half of `default_action` (diagram above) is code like this. The *pièce de résistance* is *this* basic block, which should be the final clue that if you have been trying to parse and decompile this assembly directly in your brain, then you should probably stop.

```

movdqa  xmm0, xmm2
psrlw   xmm0, 1
movdqa  xmm1, cs:xmword_385F0
pand    xmm0, xmm1
psubb   xmm2, xmm0
movdqa  xmm0, cs:xmword_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:xmword_38610
pand    xmm7, xmm2
pxor    xmm0, xmm8
psadbw  xmm7, xmm8
movdqa  xmm5, xmm6
psrlw   xmm5, 1
pand    xmm5, xmm1
psubb   xmm6, xmm5
movdqa  xmm5, xmm6
pand    xmm5, xmm0
psrlw   xmm6, 2
pand    xmm6, xmm0
paddb   xmm6, xmm5
movdqa  xmm5, xmm6

```

Figure 10. `default_action` basic block featuring optimized instructions.

Actually this is just one-tenth of that basic block, which goes on for a mind-numbing 200-odd instructions.

What gives?

Pull the Thread

To provide an operator-friendly visual representation of what the malware is doing, the authors used the library `indicatif`, which is “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”. `indicatif` output is true eye candy for CLI fans:

```
mitsuhiko at herzog in ~/Development/indicatif on git:master+2? rust 1.16.0
$ cargo run --example yarnish
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
    Running `target/debug/examples/yarnish`
[1/4] 🔍 Resolving packages...
[2/4] 🚚 Fetching packages...
[3/4] 🔗 Linking dependencies...
17/1232
```

Figure 11. Demonstration of indicatif functionality, taken from the project repository's readme.

We invite you to put yourself in the malware operator's shoes for a moment, seeing this colorful box full of emojis and progress bars as it cheerfully declares “destroying backups”, “generating symmetric session key”, “encrypting VM”, “writing ransom note”. Truly, the future is here! Just like the [seahorse](#) project page, the [indicatif](#) project page contains several ‘hello world’ examples, and one of them is the source for the above eye-catching demo: [yarnish.rs](#). We will get to the full code later, but one particular feature of it is that it is multithreaded, and has the following outline:

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

We have good reason to suspect that the source in [yarnish.rs](#), or rather, code heavily inspired by it, appears verbatim in the malware source (more on this later). Sadly, this alone still does not explain why Akira's `default_action` begins with a deluge of basic blocks invoking the C FFI and other low-level functionality. To account for that, we need to consider that the malware authors introduced a small modification to the above source — one that, on their own end, must have been an afterthought. This snippet in the original boilerplate:

```
let handles: Vec<_> = (0..4u32)
    .map(|i| ...
```

Hard-codes the number of threads to 4, which, as the malware authors noted correctly, is nice for a “hello world” example but not befitting of production code. 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 added the following:

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

And modified the source code to something like:

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

We will now delve into the functions that `available_parallelism` calls, the functions *these* functions call, and so on for several layers of the call stack. The grim motivation for this will become clearer soon (and you might be able to already guess at it).

Inline-ception

The function `available_parallelism() -> Result<NonZero<usize>>` ([source](#)), according to [its documentation](#), “returns an estimate of the default amount of parallelism a program should use [...] This number often corresponds to the number of CPUs a computer has, but it may diverge in various cases”. The implementation of this function for Linux targets is below.

```
{
    quota = cgroups::quota().max(1);
    let mut set: libc::cpu_set_t = unsafe { mem::zeroed() };
    unsafe {
        if libc::sched_getaffinity(0, mem::size_of::<libc::cpu_set_t>(), &mut set) ==
0 {
            let count = libc::CPU_COUNT(&set) as usize;
            let count = count.min(quota);
            if let Some(count) = NonZeroUsize::new(count) {
                return Ok(count)
            }
        }
    }
}
match unsafe { libc::sysconf(libc::_SC_NPROCESSORS_ONLN) } {
    -1 => Err(io::Error::last_os_error()),
    0 => Err(io::const_io_error!(io::ErrorKind::NotFound, "The number of hardware
threads is not known for the target platform")),
    cpus => {
        let count = cpus as usize;
        let count = count.min(quota);
        Ok(unsafe { NonZeroUsize::new_unchecked(count) })
    }
}
```

The undocumented function `quota() -> usize` ([source](#)), according to the in-source comments, “returns cgroup CPU quota in core-equivalents, rounded down or `usize::MAX` if the quota cannot be determined or is not set.” The implementation of this function, which we’ve again redacted for brevity, goes like so:

```
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),
        };
    };
    quota
}
```

The undocumented `quota_v1` ([source](#)) and `quota_v2` ([source](#)) are sub-implementations of `quota` for different versions of the [Linux Control Groups API](#).

Meanwhile, the function `libc::CPU_COUNT` ([source](#)) is a thin wrapper for `CPU_COUNT_S` ([source](#)), which has the following implementation:

```
pub fn CPU_COUNT_S(size: usize, cpuset: &cpu_set_t) -> c_int {
    let mut s: u32 = 0;
    let size_of_mask = core::mem::size_of_val(&cpuset.bits[0]);

    for i in cpuset.bits[..(size / size_of_mask)].iter() {
        s += i.count_ones();
    }
    s as c_int
}
```

Where `count_ones()` is a wrapper for an LLVM intrinsic `popcnt64`. The assembly it will get expanded to depends on the platform and language — [here](#) is one Fortran example which contains the same tell-tale constants we observed in the Akira assembly:

```
inline constexpr int BitPopulationCount(INT x) {
    x = (x & 0x5555555555555555) + ((x >> 1) & 0x5555555555555555);
    x = (x & 0x3333333333333333) + ((x >> 2) & 0x3333333333333333);
    x = (x & 0x0f0f0f0f0f0f0f0f) + ((x >> 4) & 0x0f0f0f0f0f0f0f0f);
    x = (x & 0x001f001f001f001f) + ((x >> 8) & 0x001f001f001f001f);
    x = (x & 0x0000003f0000003f) + ((x >> 16) & 0x0000003f0000003f);
    return (x & 0x7f) + (x >> 32);
}
```

The (predictable, depressing) explanation for the indecipherable flood of basic blocks that appear in `default_action` is that ALL of them are various degrees of inlined library code.

What we mean by this is that instead of a function call to `available_parallelism`, the implementation of that function appears as-is in `default_action`. But then, instead of the call to `quota` that should appear there, the entire implementation of `quota` is spliced into the assembly — except the implementations of `quota_v1` and `quota_v2` are *also* spliced directly into the function instead of called. The same goes for `CPU_COUNT`, which gets expanded into its implementation, which is repeatedly calling `count_ones` except the loop is unrolled and each single `count_ones` is expanded into *its* implementation. What this all results in is that the entire complex, tangled upper half of `default_action` is *five level deep* inlined library code, with the different functions and levels side by side, all downstream from that one single innocuous decision by a malware author to call `available_parallelism`.

	user code
	available_parallelism
	quota
	quota_v1
	quota_v2
	CPU_COUNT

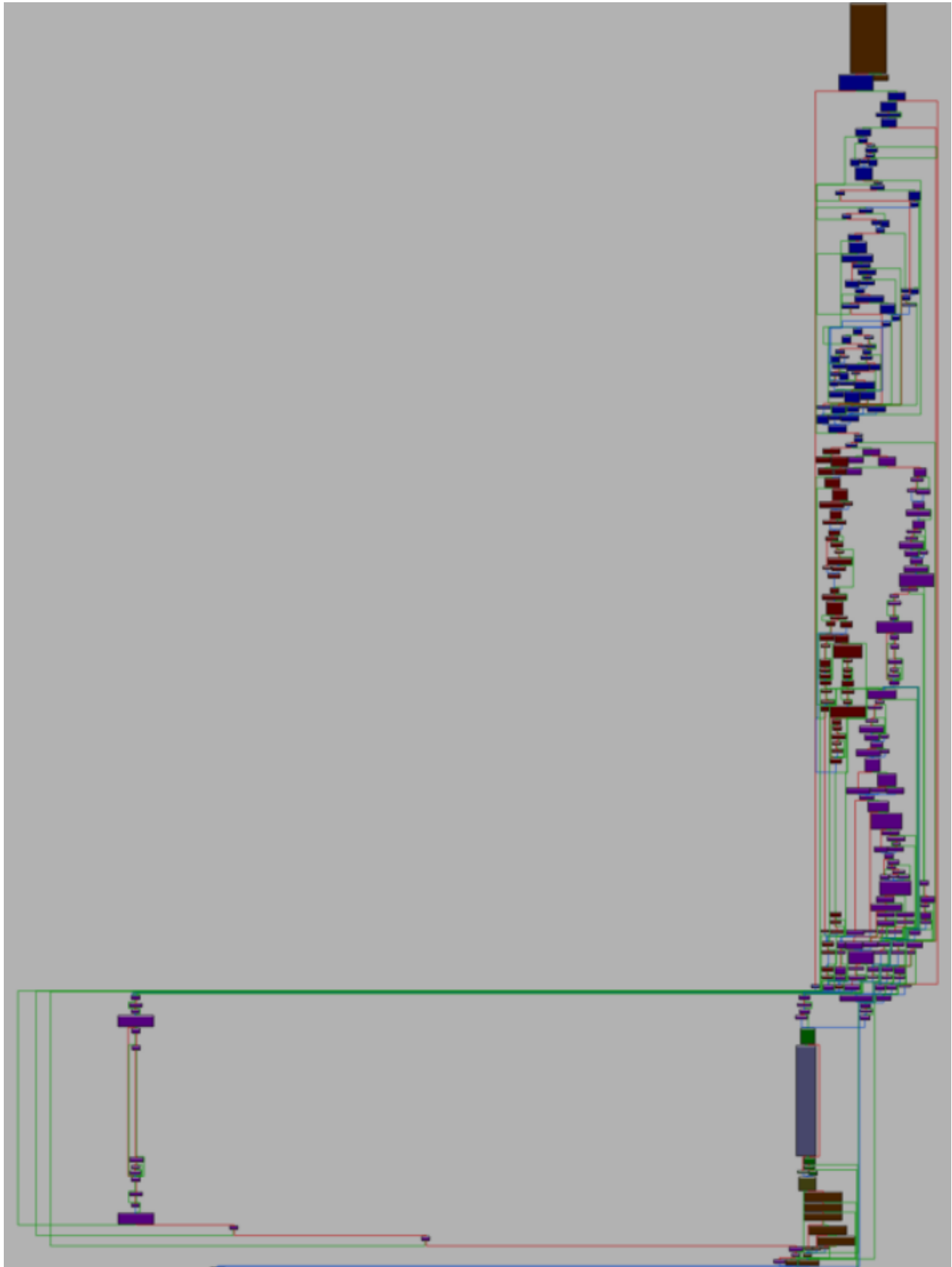


Figure 12. Breakdown of `default_action` by source of functionality.

We will see more examples of this theme later, and in fact, between the argument initialization and the call to `app.run`, the main function we discussed earlier also contains a large amount of inlined `seahorse` code.

Default Action User Code

Once all the inlined library code is accounted for, the actual manually authored user code remains. You would expect that the assembly would launch user threads here as per [yarnish.rs](#), but no. The rest of `default_action` is mainly dedicated to checking the values of various user-provided flags and adjusting malware behavior accordingly. One example is checking for the `--stopvm` flag; if this is set, the malware forks a new process with the command `vim-cmd vmsvc/getallvms | tail -n +2 | awk '{system("vim-cmd vmsvc/power.off " $1)}'`.

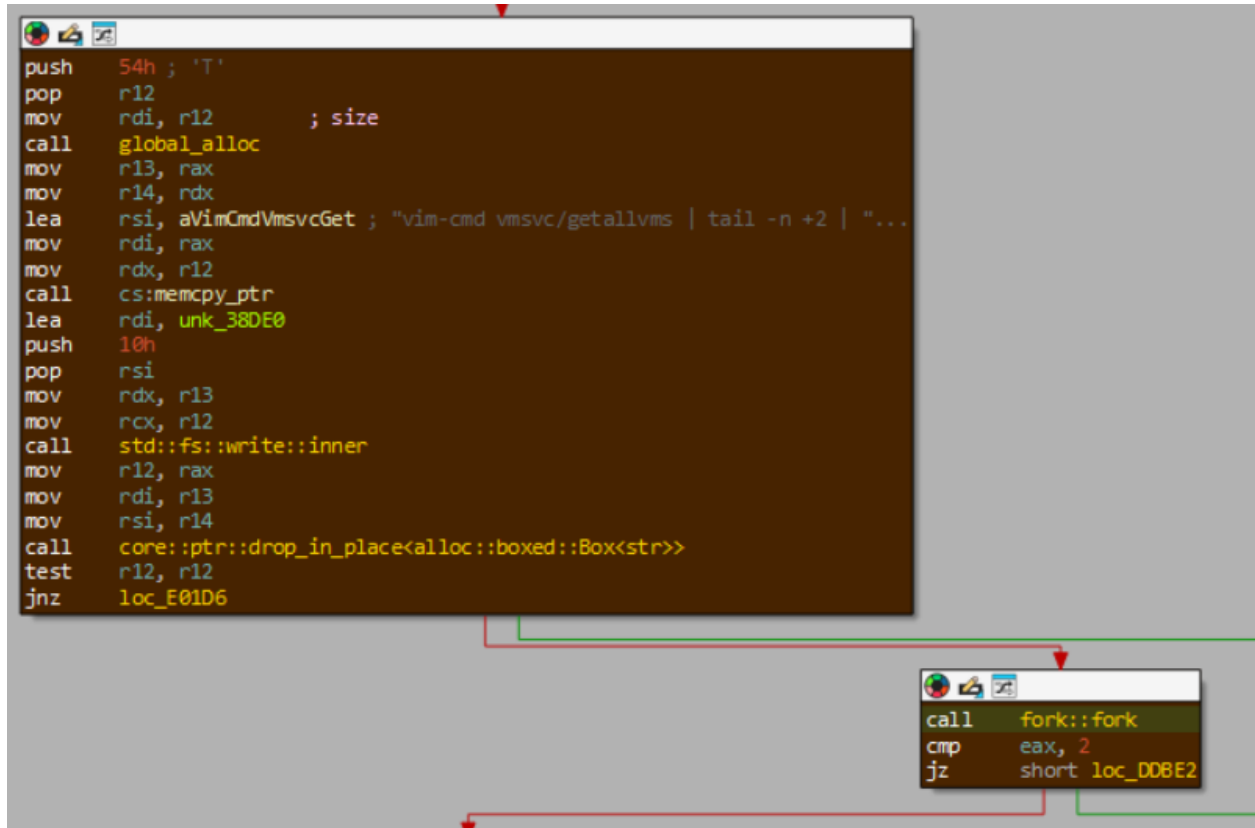


Figure 13. Authored code in `default_action` turning off all hosted VMs in the ESXi server.

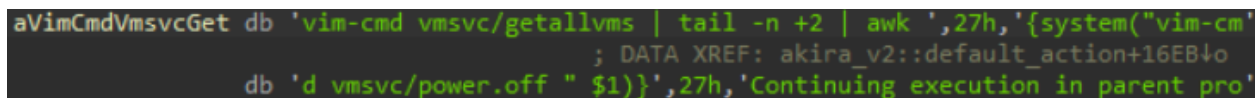


Figure 14. Shell command used to turn off all VMs.

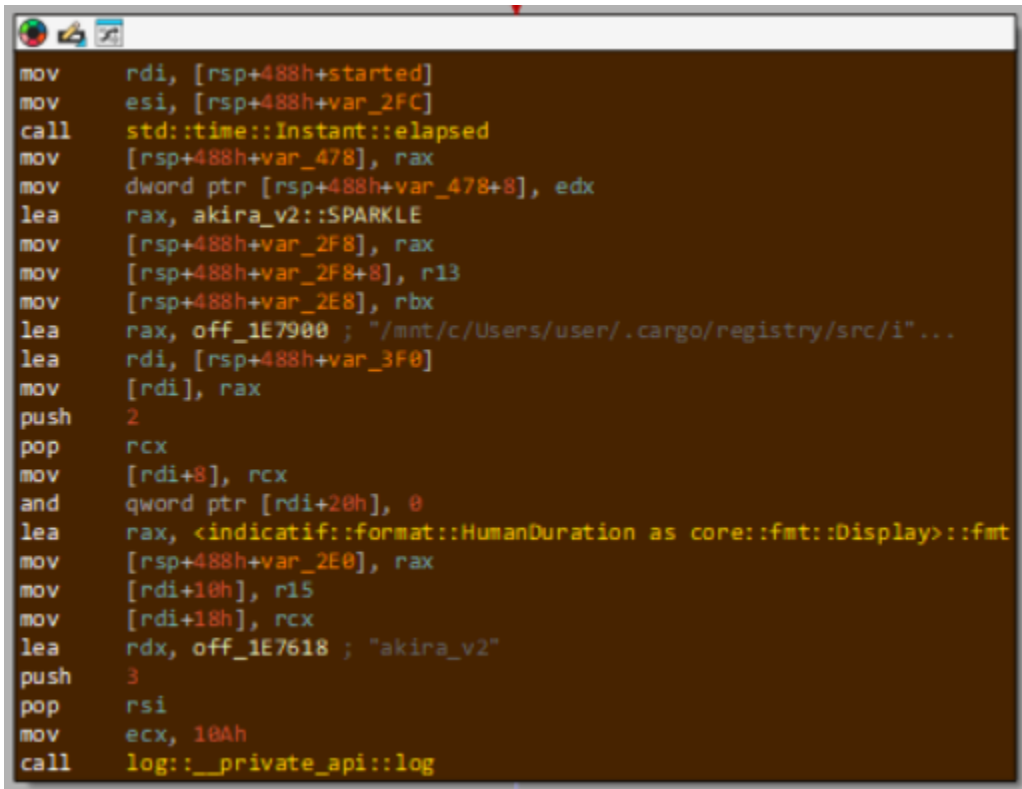
The behavior of the `--path` flag is an artifact of how this ransomware was made to target ESXi servers by default, and Linux environments in general only as an additional feature. Its official documentation is:

```
--path <string> Start path. Default value: /vmfs/volumes
```

This means that unless other behavior is specified, the ransomware will attempt to target ESXi VMs specifically. Still, we should stress that this malware has all the required functionality to act as general-purpose Linux ransomware, and this specific focus on ESXi is

merely a default.

Of course, `indicatif` is used to its full potential to inform the malware operator of every detail of the current execution status — for example, reporting the ✨ elapsed time :



```
mov     rdi, [rsp+488h+started]
mov     esi, [rsp+488h+var_2FC]
call    std::time::Instant::elapsed
mov     [rsp+488h+var_478], rax
mov     dword ptr [rsp+488h+var_478+8], edx
lea     rax, akira_v2::SPARKLE
mov     [rsp+488h+var_2F8], rax
mov     [rsp+488h+var_2F8+8], r13
mov     [rsp+488h+var_2E8], rbx
lea     rax, off_1E7900 ; "/mnt/c/Users/user/.cargo/registry/src/i"...
lea     rdi, [rsp+488h+var_3F0]
mov     [rdi], rax
push    2
pop     rcx
mov     [rdi+8], rcx
and     qword ptr [rdi+20h], 0
lea     rax, <indicatif::format::HumanDuration as core::fmt::Display>::fmt
mov     [rsp+488h+var_2E0], rax
mov     [rdi+10h], r15
mov     [rdi+18h], rcx
lea     rdx, off_1E7618 ; "akira_v2"
push    3
pop     rsi
mov     ecx, 10Ah
call    log::__private_api::log
```

Figure 15. Verbose reporting (1)

Reporting the (meticulously-calculated, as we earlier saw) 🌞 number of available threads :

```
lea    rax, akira_v2::FIRE
mov    qword ptr [rsp+488h+var_3F0], rax
lea    rax, <console::utils::Emoji as core::fmt::Display>::fmt
mov    qword ptr [rsp+488h+var_3F0+8], rax
lea    rax, [rsp+488h+cores]
mov    [rsp+488h+var_3E0], rax
lea    rax, off_1E7780 ; "/mnt/c/Users/user/.cargo/registry/src/i"...
lea    rdi, [rsp+488h+cpuset]
mov    [rdi], rax
push   2
pop    rcx
mov    [rdi+8], rcx
and    qword ptr [rdi+20h], 0
lea    rax, core::fmt::num::<impl core::fmt::Debug for usize>::fmt
mov    qword ptr [rsp+488h+var_3D8], rax
mov    [rdi+10h], rbx
mov    [rdi+18h], rcx
lea    rdx, off_1E7618 ; "akira_v2"
push   4
pop    rsi
mov    ecx, 0D7h
call   log::__private_api::log
```

Figure 16. Verbose reporting (2). This version uses the “fire” emoji.

And refusing to run when provided the ✗ wrong build-id :

```
loc_D0A89:
lea    rcx, aLockFallbackFi+0CE5h ; "idstopvmvmonlyCrypt only .vmdk, .vmem, ..."
lea    rbx, [rsp+488h+var_3F0]
push   2
pop    r8
mov    rdi, rbx
mov    rsi, r15
mov    rdx, rbp
call   seahorse::context::Context::string_flag
mov    r12, [rbx]
test   r12, r12
jz     loc_E00A1

mov    rsi, [rsp+488h+var_3E0]
lea    rdx, aLockFallbackFi+102Ah ; "GzsRFSqEhhvim-cmd vmvc/getallvms | tai"...
push   0Ah
pop    rcx
mov    rdi, r12
call   cmp ; core::cmp::impl::<impl core::cmp::PartialEq<&B> for &A>::eq::h0183f9703cf26053
test   al, al
jz     loc_E00B9
```

Figure 17. build id match check

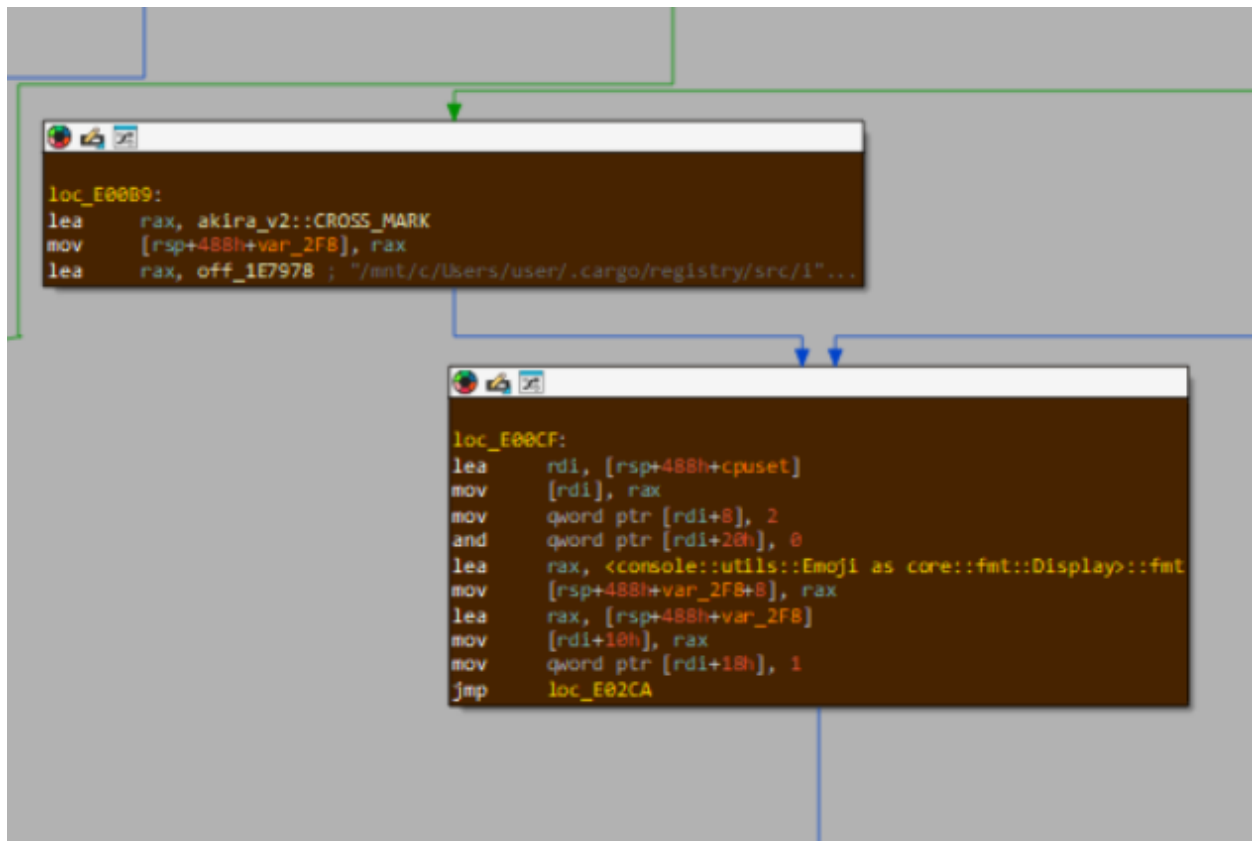


Figure 18. verbose reporting (3)

Finally, the lock function is called, and the next stage of execution begins.

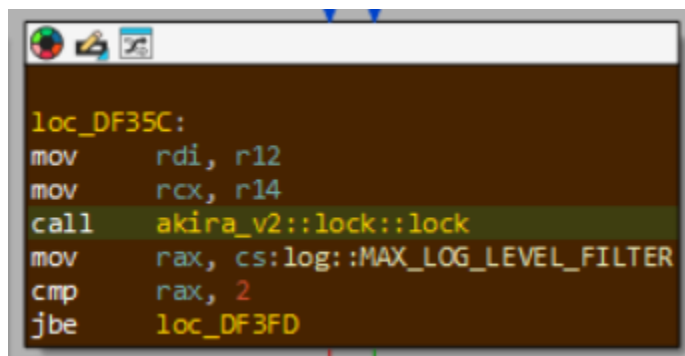


Figure 19. Call to lock function

Lock function

Thread Spawning

As we mentioned before, we have good reason to believe that `yarnish.rs` was used as a prototype during the writing of this malware — specifically for this function, `lock`, which was moved to a separate module (`lock.rs`). The code of `yarnish.rs` is reproduced below, redacted for brevity:

```

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();

```

The strings and `{prefix:.bold.dim} {spinner} {wide_msg}` appear as-is in the `lock` function, used in the same way as in the source above.

```

call    std::time::Instant::now

mov     [rsp+488h+started], rax
mov     [rsp+488h+var_2FC], edx
lea     rsi, aStartPathDefau+27Bh ; "{prefix:.bold.dim} {spinner} {wide_msg}"...
lea     rbx, [rsp+488h+cpuset]
push    27h ; ''
pop     rdx
mov     rdi, rbx
call    indicatif::style::ProgressStyle::with_template

lea     rdx, off_1E74E8 ; "akiranew/src/main.rstracedebugValid val"...
lea     r14, [rsp+488h+spinner_style]
mov     rdi, r14
mov     rsi, rbx
call    core::result::Result<T,E>::unwrap

lea     rdi, [rsp+488h+tick_chars]
mov     rsi, r14
call    indicatif::style::ProgressStyle::tick_chars

```

Figure 20. Compiled `indicatif` boilerplate

The hardcoded `tick_chars` is spliced directly into the `tick_chars` implementation by the compiler (instead of the function taking it as an argument as you'd expect from the source):

```

indicatif::style::ProgressStyle::tick_chars proc near

var_98= xmmword ptr -98h
var_80= qword ptr -80h
var_78= qword ptr -78h
var_70= xmmword ptr -70h
var_60= qword ptr -60h
var_58= qword ptr -58h
var_50= qword ptr -50h

; __unwind {
push    rbp
push    r15
push    r14
push    r13
push    r12
push    rbx
sub     rsp, 68h
mov     r12, rsi
mov     r13, rdi
lea     rax, aA1NumThreadsFr ; "... num_threads: Free threads avai"...

```

Figure 21. The “function argument” `tick_chars` appears directly in the function body instead of being passed as an argument.

Despite its name, at the assembly level `lock` does not itself perform any cryptographic operations; it delegates these to separate threads that it spawns. To get a better grip on how the call to `thread::spawn` cashes out in assembly, we go back to the lab setting for a short moment and use a toy program where we control the source:

```
use std::thread;
use rand::Rng;

fn main() {
    let _ = thread::spawn(move || {print_rand_char();}).join();
}

fn print_rand_char() {
    let s = "astitchintime";
    let chars: Vec<char> = s.chars().collect();
    let mut rng = rand::thread_rng();
    let idx = rng.gen_range(0..chars.len());
    println!("{}", chars[idx]);
}
```

Checking the generated assembly, the `main` function calls `std::thread::spawn`, except (again) a specific key argument is spliced directly into the code instead of given as an argument. This is `box_fnonce_vtable` which is directly hardcoded as an argument for `std::sys::windows::thread::Thread::new`. The basic block containing the call is highlighted in white.

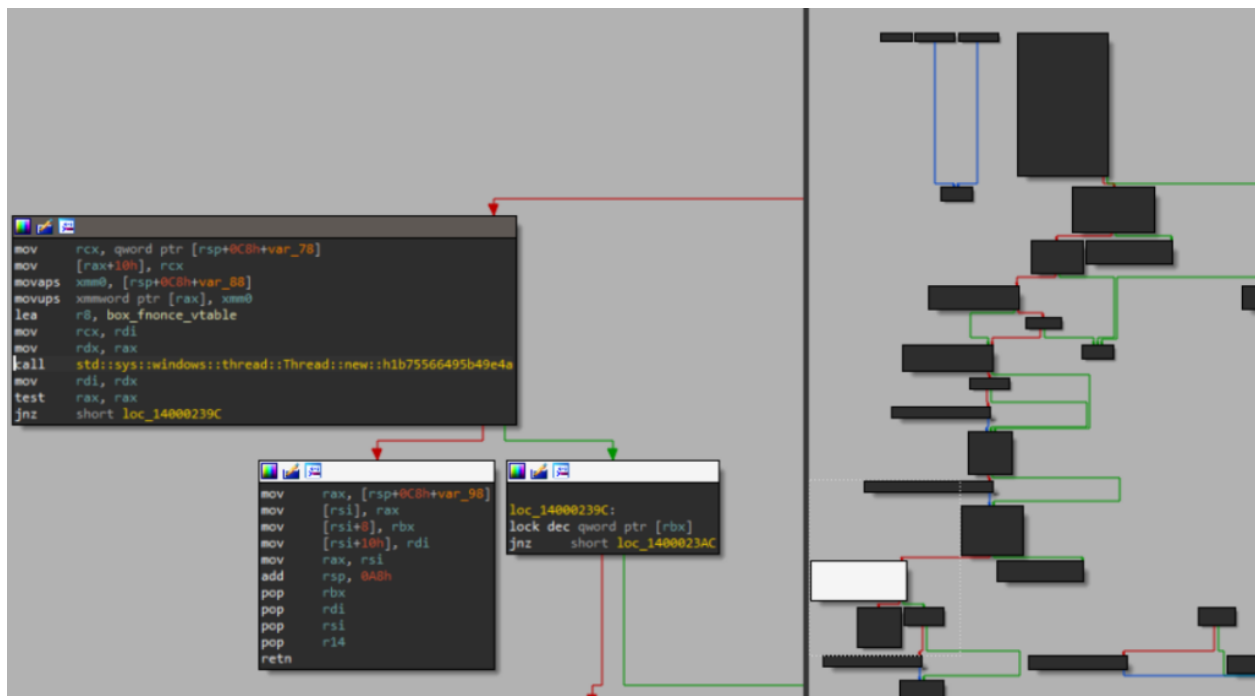


Figure 22. Call to `sys::windows::thread::Thread::new`.

What gets passed in the register `r8` is evidently a vtable containing a pointer to a function executed by the thread (we mean this is evident from looking at the assembly, not the variable name `box_fnonce_vtable`, which we named after the fact). We could have just included the screenshot and left it there, but it's more instructive to understand what object is being constructed and operated on here and why, as this pertains to all Rust binaries that use `thread::spawn`.

`Thread::new`, which does the actual thread-creating, has the prototype `new(stack: usize, p: Box<dyn FnOnce()>) -> io::Result<Thread>`. Effectively, this means 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>`, possibly also known to you as a “struct with a vtable pointer”, and in this case the trait is `FnOnce`, meaning the object can 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 `spawn`.

This function is implemented using an object called `Builder`. The implementation of that object is here:

```
pub struct Builder {
    name: Option<String>,
    stack_size: Option<usize>,
}
```

The function `impl spawn for Builder` has the prototype:

```
pub fn spawn<F, T>(self, f: F) -> io::Result<JoinHandle<T>>
    where
        F: FnOnce() -> T,
        F: Send + 'static,
        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 bookkeeping surrounding the launch of the `f` logic, and the resulting closure is the object that gets passed to `Thread::new` here:

```

let main = move || {
    if let Some(name) = their_thread.cname() {
        imp::Thread::set_name(name);
    }

    crate::io::set_output_capture(output_capture);

    let f = f.into_inner();
    set_current(their_thread);
    let try_result = panic::catch_unwind(panic::AssertUnwindSafe(|| {
        crate::sys::backtrace::__rust_begin_short_backtrace(f)
    }));
    ...
Ok(JoinInner {
    native: unsafe { imp::Thread::new(stack_size, main)? },
    thread: my_thread,
    packet: my_packet,
})

```

We can now take a proper look at the assembly implementing the multithreaded ransomware logic. The entire call chain `thread::spawn -> Builder::spawn -> Builder::spawn_unchecked -> Builder::spawn_unchecked_` is inlined (highlighted in purple) directly into the `lock` function:

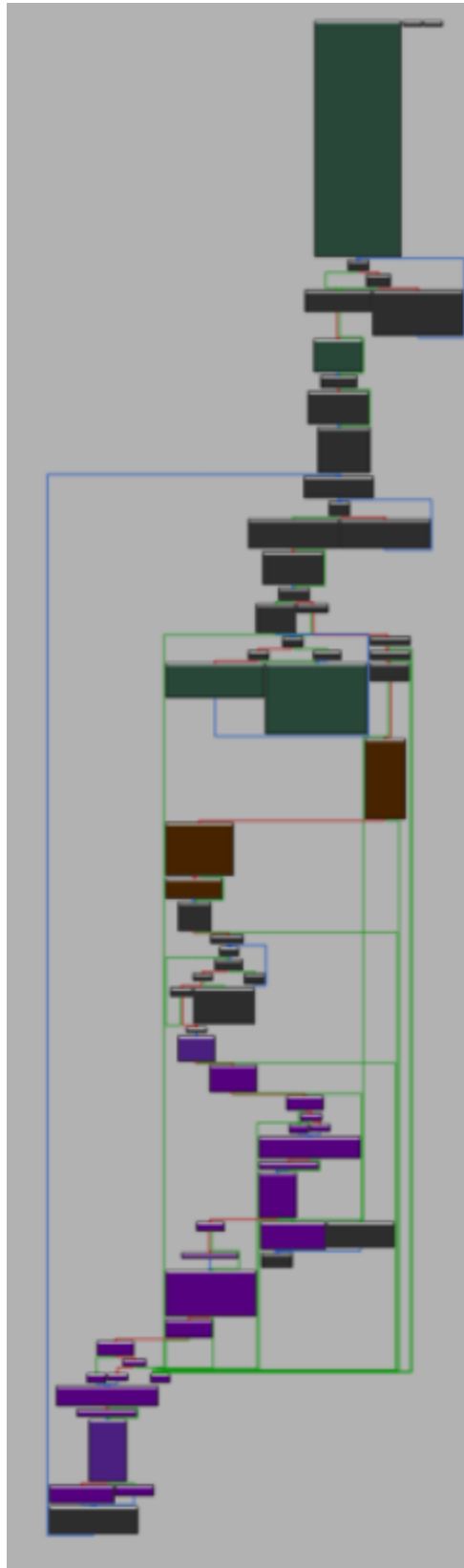
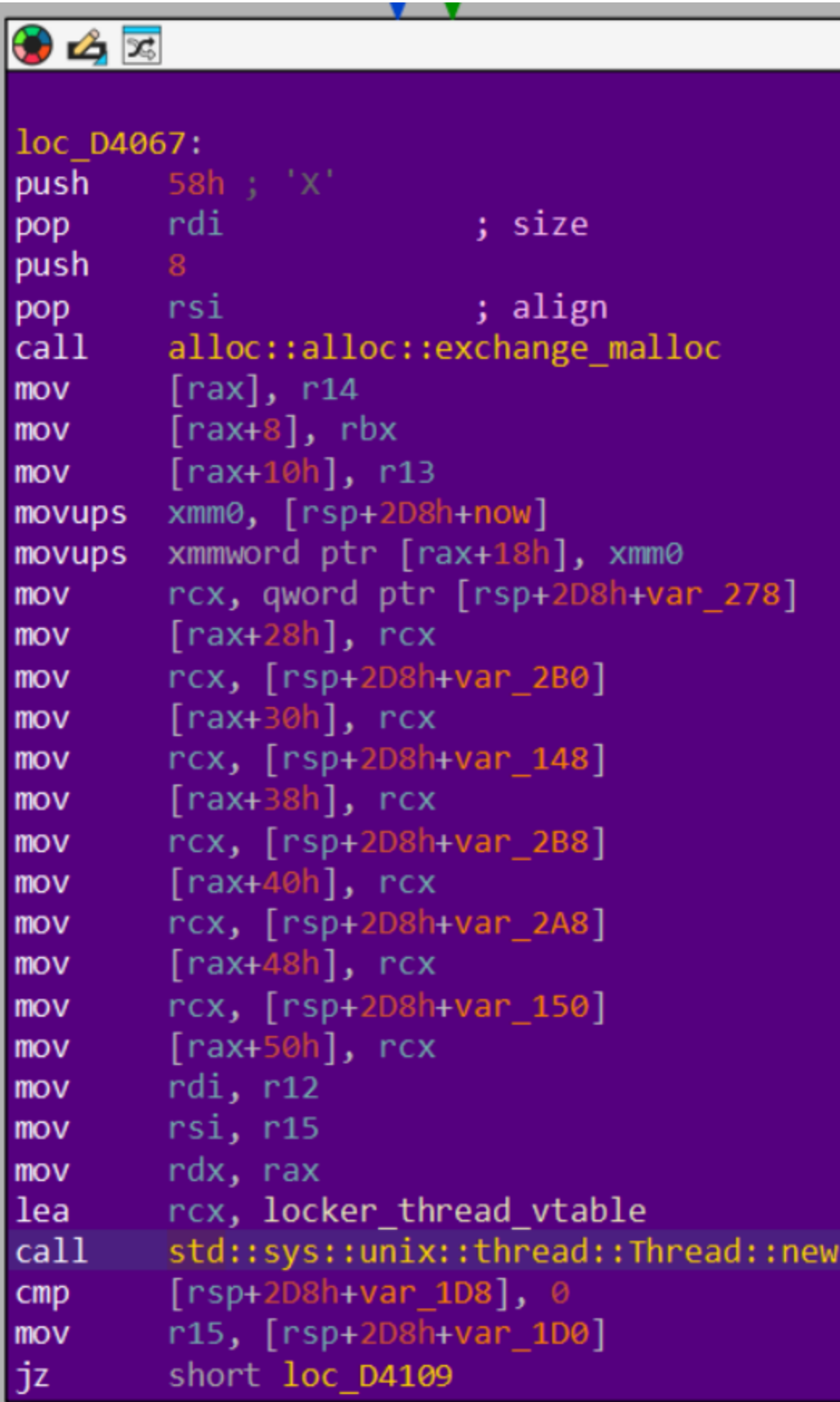


Figure 23. `Builder::spawn` call chain inlined into user code.

The call to `thread::new` 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 24. Preparation of the `Box<dyn FnOnce()>` object leading up to the call to `sys::unix::thread::Thread::new`.

The locker thread vtable:

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

Figure 25. The `Box<dyn FnOnce()>` object in memory.

The shim is below (e.g. the call to `set_output_capture` is visible here):

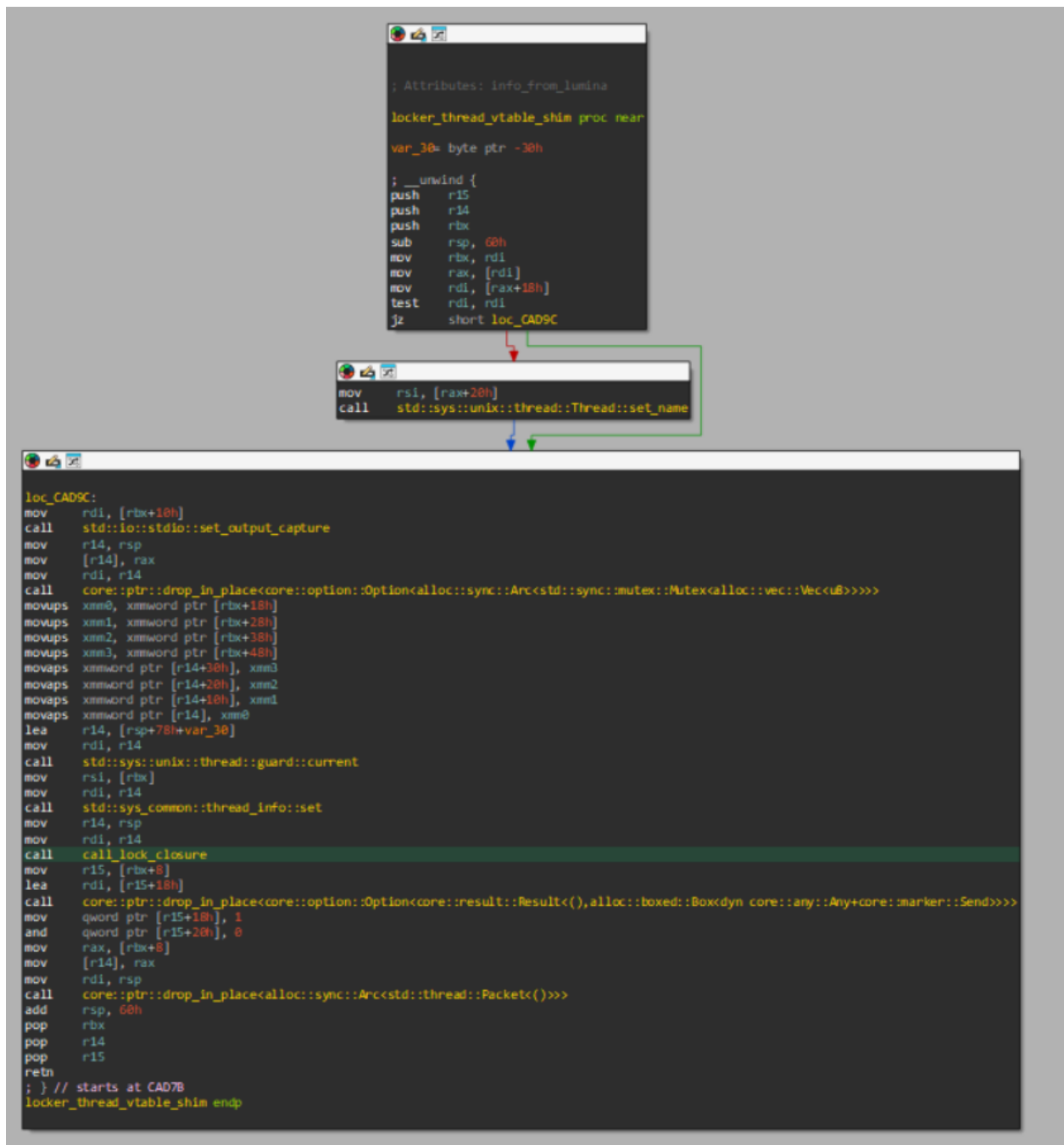


Figure 26. Completed vtable shim, the source of which we earlier saw (`let main = move || ...`)

For the sake of simplicity, we assigned the name `call_lock_closure` to the call where the shim invokes `sys::backtrace::__rust_begin_short_backtrace` with `lock_closure` as a parameter.

By default, the malware recursively targets encryption files in the directory `/vmfs/volumes`. Apart from the `--vmonly` flag, the malware offers the operator sophisticated control of exactly what files will be targeted using the `--exclude` flag, as detailed in the `help` text:

```
Exclude - skip files by "regular" extension.  
Example:  
--exclude="startfilename('.' *).(*)"  
using this regular expression will skip all files starting with startfilename and  
having any extensions.  
Multiple regular' expressions using "|" can also be processed:  
--exclude="(win10-' 3(.*)).(.*))|(win10-4(.*)).(.*))|(win10-5(.*)).(.*))"
```

Once the thread is launched, the process of encrypting its victim file begins.

“Lock Closure” & Encryption Logic

This is what the authors of this venerable ransomware placed inside `thread::spawn(move || { ... })` in their modified `yarnish.rs`. Encryption is implemented using a standard hybrid approach of an asymmetric and symmetric cipher. That is, first a per-file symmetric key is generated, and then it is encrypted using the per-binary hardcoded public key. For asymmetric encryption, the malware uses [dalek.rs](#) implementation of [curve25519](#). The embedded public key, which varies from sample to sample, is found embedded in the binary as a hexadecimal string:

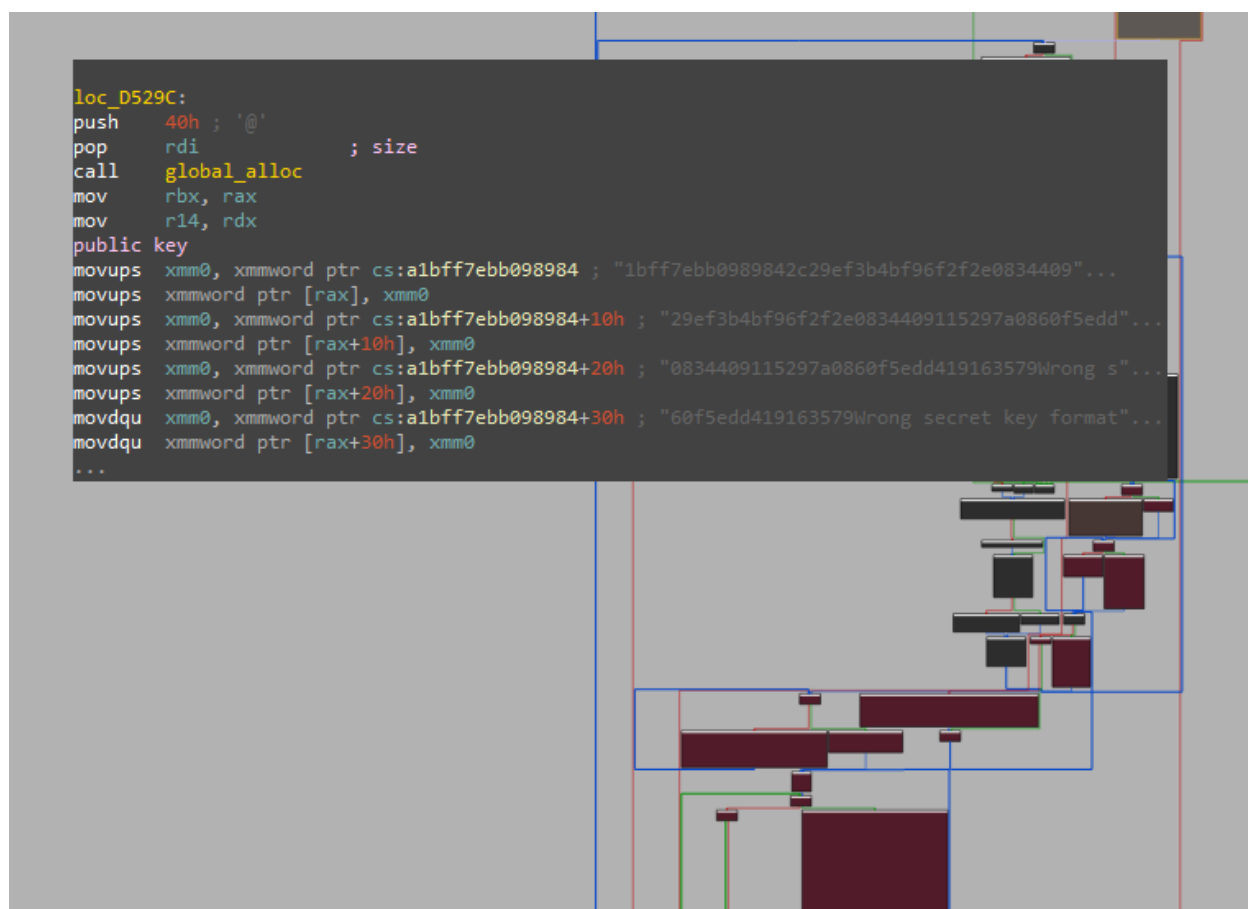


Figure 27. Hardcoded curve25519 public key in memory.

For the symmetric per-file encryption, the malware somewhat unusually uses SOSEMANUK. This stream cipher has seen some documented malicious use since at least 2022 in Pridelocker, a fork of Babuk. For this specific cipher, the malware uses the `rust-crypto` crate. As a point of interest `rust-crypto` has its implementation of encryption using curve25519, separate from that of `dalek.rs`.

The use of SOSEMANUK is (again) not unprecedented and documented in other versions of this ransomware. However, we can still clearly visualize the malware author picking this specific cipher and subconsciously snickering, “Heh heh, have fun.” Indeed, with this decision they forced the hand of a heroic soul to implement Sosamanuk from scratch in Python specifically to deal with this ransomware.

The symmetric session key is generated immediately after the public key is loaded into memory. This is done by calling `thread_rng` and obtaining a `ThreadRng` object, then calling `generate` on its internal `ReseedingCore`. The authors would have used the user-facing `Rng` trait, which offers functions like `random` and `fill` for this purpose (the underlying layers of logic eventually pass through the PAL and `query /dev/(u)random`); what we see is the inlined internal implementation down to the level of `generate` (part of the `BlockRngCore` trait).

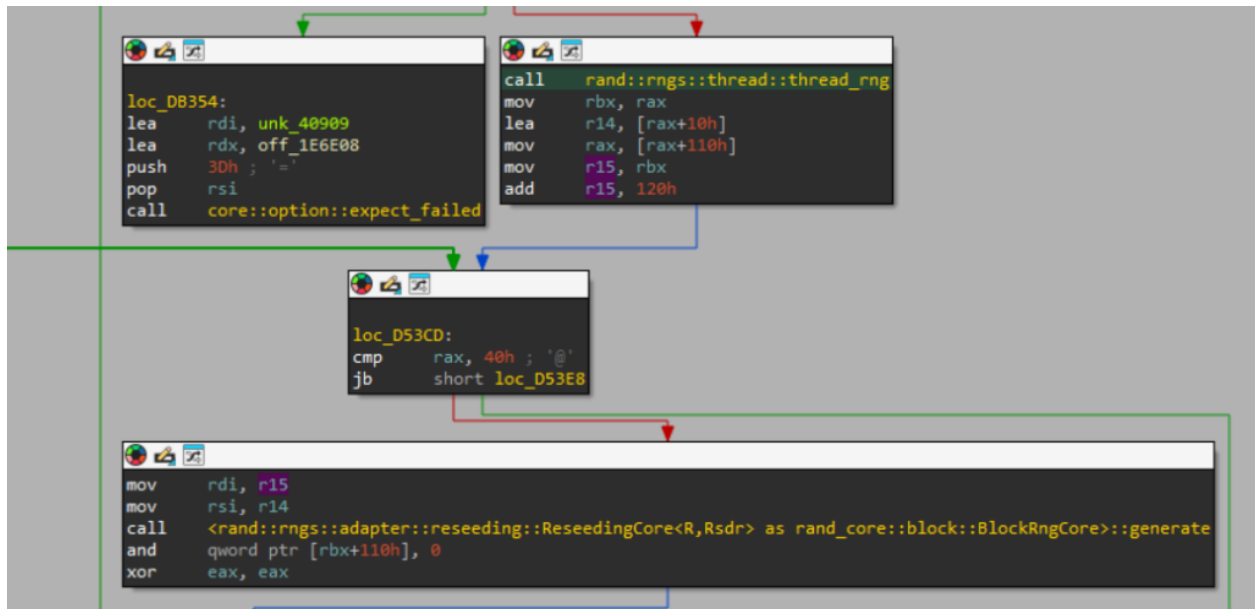


Figure 28. Call to generate downstream from idiomatic Rust used to generate a random symmetric key; it was inlined up the call chain.

The same is true for the curve25519 and the SOSEMANUK implementations, which appear as-is in the lock closure. We visualize the spliced basic blocks below.

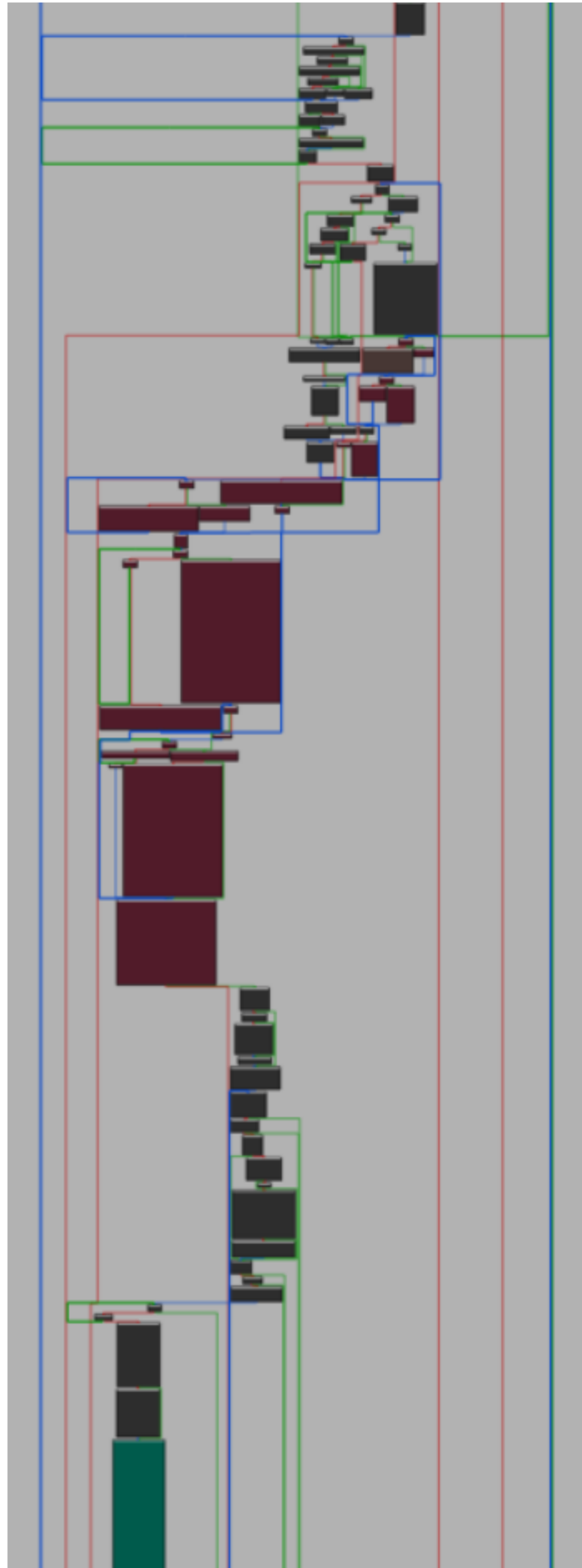




Figure 29. curve25519 code (red) followed by SOSEMANUK code (green), both inlined up the call chain into user code.

Whereas the `dalek.rs` implementation of curve25519 leaves some identifiable traces in the form of function calls, the SOSEMANUK implementation is completely in-lined, and the cipher has to be identified using the traditional method of identifying incriminating constants (for example `0x54655307`, which when converted to decimal gives the first 10 decimals of pi). The image above only has the tip of the iceberg of the inlined SOSEMANUK implementation; it is one single, gigantic basic block spanning hundreds of instructions. Once the target file is encrypted, the malware appends the extension `akiranew`. And, once all files in a target directory have been encrypted, the malware writes a ransom note in that directory to `akiranew.txt`, which begins:

```
Hi friends,  
Whatever who you are and what your title is if you're reading this  
it means the internal infrastructure of your company is fully or partially  
dead, all your backups - virtual, physical - everything that we managed to reach -  
are completely removed. Moreover, we have taken a great amount of your  
corporate data prior to encryption. [..]
```

Tragically, the person reading the above is not likely to appreciate the great effort made by the Rust compiler to bring this result to them with fearless concurrency and in a blazingly fast manner.

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 favorite 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 a [previous publication](#)), when analyzing a Rust binary, we need to contend with aggressive in-lining that, as it turns out, can easily reach five levels deep into library code.

In this publication, we leveraged some favorable conditions and reached a state where we could map the control flow of an actual ITW Rust binary end to end. One of the most crucial tools in our arsenal 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.

Still, we hope you gained something from the journey of analyzing an actual ITW Rust binary and sorting through the various hairy internals that get spliced into top level user code, where they have no business being. Once upon a time, reverse-engineering C binaries were also primordial and scary; eventually, understanding improved, tooling caught up, and the task became much less formidable than it once was. We can only reason and hope that even the occasionally painful output of the Rust compiler will meet the same fate — one day in the not extremely far future.

Protections

Check Point Threat Emulation and Harmony Endpoint provide comprehensive coverage of attack tactics, filetypes, and operating systems and protect against the types of attacks and threats described in this report.

Harmony Endpoint

- Ransomware_Linux_Akira_C
- Ransomware_Linux_Akira_D

Threat Emulation

- Ransomware.Wins.Akira.G
- Ransomware.Wins.Akira.H

IOCs

SHA256	3298d203c2acb68c474e5fdad8379181890b4403d6491c523c13730129be3f75
VM termination substring	awk '{system("vim-cmd vmshvc/power.off " \$1)}'
Encrypted file extension	.akiranew

[GO UP](#)

[BACK TO ALL POSTS](#)