Getting Rusty and Stringy with Luna Ransomware

Archived: 2026-04-06 01:17:45 UTC

- SHA256: 1cbbf108f44c8f4babde546d26425ca5340dccf878d306b90eb0fbec2f83ab51
    - VT download link

Table of Contents

Family Introduction

The Luna ransomware appeared in July 2022. Unlike its competitors, this threat targeted VMware ESXi instances from the day it started operating.

Rust Strings

In my experience as a malware analyst, I've been used to seeing ASCII and null-terminated strings in binaries. I was content writing IDAPython scripts where I created strings by searching for ASCII and null characters. And one fine day, I had a Rust binary on my plate which broke my scripts. I interviewed the Rust God about strings. Here's how it went:

Fig. 1: Rust Strings

String Slice: &str

String slice is the term for `&str` type of strings. These kinds of strings may exist in the binary or on the stack or heap. They always reference UTF-8 characters and are immutable. Let's consider this simple Rust program:

```
fn main() {
   let str1: &str = "Hello World!\n";
}
```

Fig. 2 shows a snap of the disassembly as seen in IDA Home 7.7:

Fig. 2: String Slice

String slices are essentially a data structure containing the address of the slice and its length. Such structures are also called fat pointers because they contain extra data besides just the memory address. Consider the following Rust program which prints the size (in bytes) of the `&str` type:

```
use std::mem::size_of;

fn main() {
    println!("A &str size in bytes: {}", size_of::<&str>());
}
```

On execution, it prints:

```
A &str size in bytes: 16
```

My system architecture is x64, so the size of `&str`, a fat pointer, is 16 bytes. The first 8 bytes is the memory address of the actual string literal and the next 8 bytes represents the length of that string literal. The following structure represents a string slice:

```
 struct string_slice
{
  _QWORD val;
  _QWORD len;
};
```

IDA detects the above structure as `core::fmt::ArgumentV1` and is defined as:

```
 struct core::fmt::ArgumentV1
{
  core::fmt::_extern_0}::Opaque *value;
  core::result::Result<(),core::fmt::Error> (*formatter)(core::fmt::_extern_0}::Opaque *, core::fmt::Formatter *
};
```

Although IDA's structure is of the correct size (16 bytes), it is not particularly readable. So, I replaced it with my structure definition for better readability. Fig. 3 shows it in action.

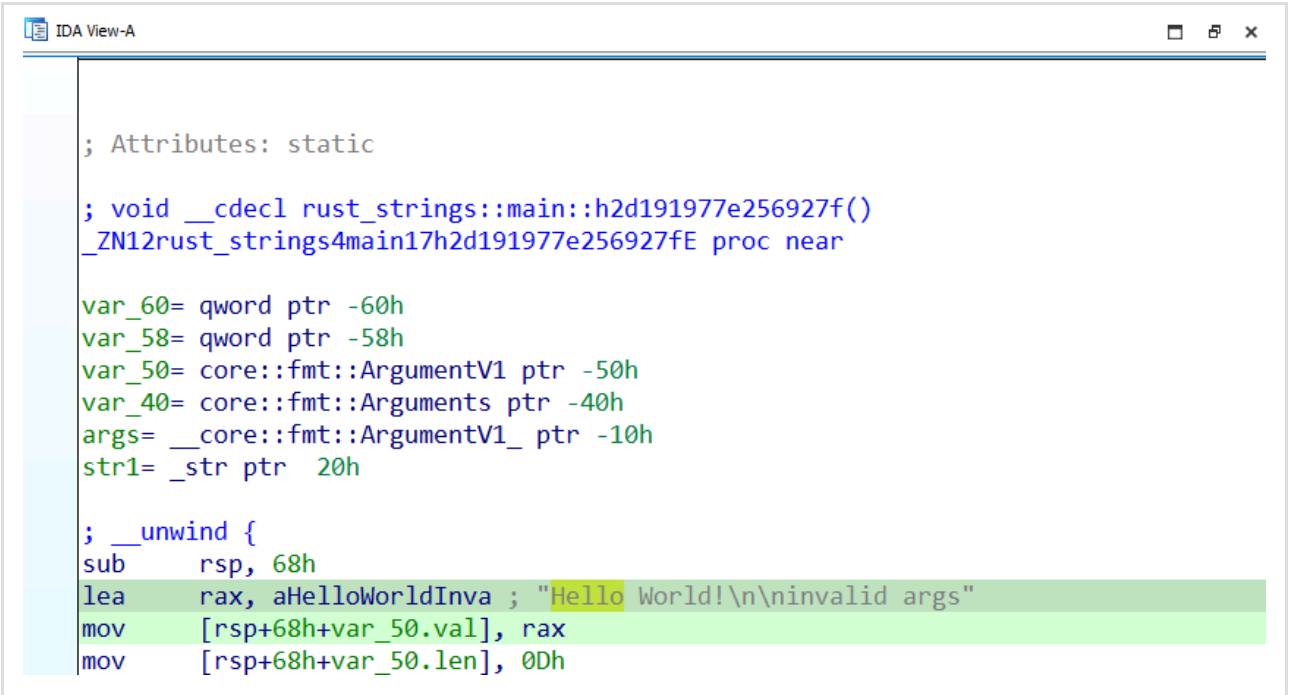

Fig. 3: String Slice IDA Structure

String

The next string type in Rust is `String`. These kinds of strings are allocated only on the heap and they are mutable.

`String` is also a data structure. It contains the address of the slice, its length on the heap and also the capacity of the heap region. Consider the following Rust program which prints the size (in bytes) of the `String` type:

```
 use std::mem::size_of;

fn main() {
    println!("A String size in bytes: {}", size_of::<String>());
}
```

On execution, it prints:

```
 A String size in bytes: 24
```

My system architecture is x64, so the size of `String` is 24 bytes. The first 8 bytes is the memory address of the string slice; the next 8 bytes represents the length of that string literal and the last 8 bytes is the capacity of the memory region in the heap. The capacity signifies the maximum number of bytes that the string can hold. If a longer string is required, then reallocation occurs on the heap. The following structure represents a `String`:

```
 struct String
{
  _QWORD val;
  _QWORD len;
  _QWORD cap;
};
```

For example, a `String` may be allocated on the heap having the following structure field values:

```
 val = "Hello!"
len = 6
cap = 10
```

IDA detects the above structure as `alloc::string::String` and is defined as:

```
 struct alloc::string::String
{
  alloc::vec::Vec<u8,alloc::alloc::Global> vec;
};
```

Let’s consider this simple Rust program:

```
 fn main() {
    let str1: String = String::from("Hello World! 🙏\n");
```

```
}
```

Fig. 4 shows a snap of the disassembly as seen in IDA Home 7.7. Here, `v1` is the `String` variable.

```
void __cdecl rust_strings::main::h2d191977e256927f()
{
  usize v0; // rdx
  alloc::string::String v1; // [rsp+10h] [rbp-68h] BYREF
  core::fmt::Arguments v2; // [rsp+28h] [rbp-50h] BYREF
  __core::fmt::ArgumentV1_ args; // [rsp+58h] [rbp-20h] BYREF

  _$LT$alloc..string..String$u20$as$u20$core..convert..From$LT$$RF$str$GT$$GT$::from::ha
    &v1,
    (_str)__PAIR128__(
            18LL,
            "Hello World! 🙏\n"
            "\n"
            "/rustc/a55dd71d5fb0ec5a6a3a9e8c27b2127ba491ce52/library/core/src/alloc/layc
  args.data_ptr = core::fmt::ArgumentV1::new_display::hbaf52e7ed1c8c9a7(
                    (core::fmt::ArgumentV1 *)&v1,
                    (alloc::string::String *)"Hello World! 🙏\n"
                                             "\n"
                                             "/rustc/a55dd71d5fb0ec5a6a3a9e8c27b2127ba4
                                             "ut.rsinvalid args");
```

Fig. 4: String IDA Structure

Fig 5. shows a snap of the UTF-8 encoding of the string literal:

```
48 65 6C 6C 6F 20 57 6F 72 6C+aHelloWorld     db 'Hello World! '
F0 9F 99 8F 0A                                db '🙏',0Ah
```

Fig. 5: String UTF-8 Encoding

Rust Strings Print

It can be seen in Fig. 4 that there is no null character after the `Hello World! 🙏\n` string. This can make reading strings in IDA decompilation difficult as seen in Fig. 2 where the next string has polluted the decompilation. I wrote an IDAPython script which prints Unicode strings found in a Rust-based binary. I've been unable to find an IDAPython function which can create UTF-8 strings.

Luna Strings

Using the IDAPython script, I found interesting strings.

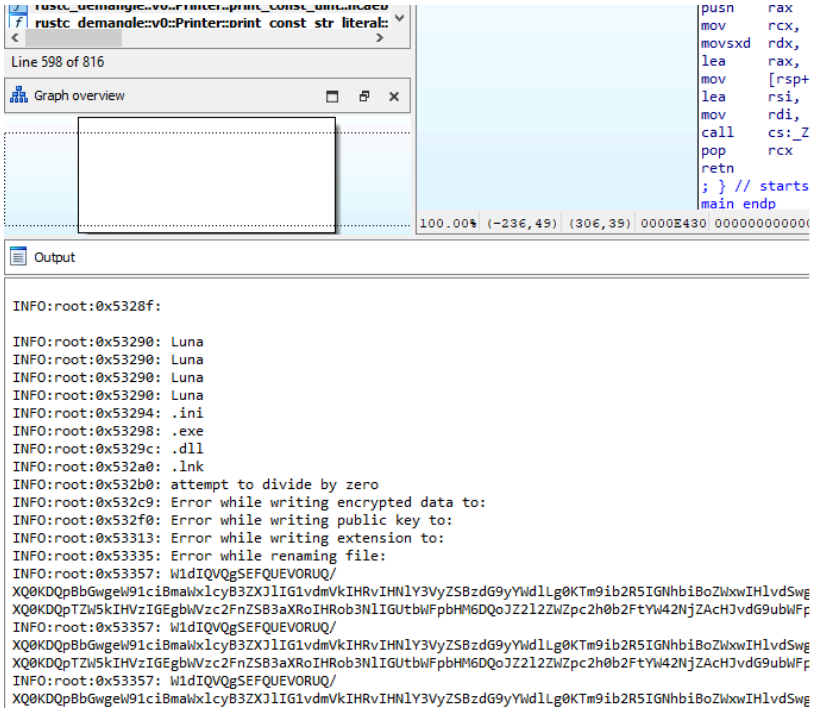

Fig. 6: String UTF-8 Encoding

```
 Luna
.ini
.exe
.dll
.lnk
Error while writing encrypted data to:
Error while writing public key to:
Error while writing extension to:
Error while renaming file:
W1dIQVQgSEFQUEVORUQ/XQ0KDQpBbGwgeW91ciBmaWxlcyB3ZXJlIG1vdmVkIHRvIHNlY3VyZSBzdG9yYWdlLg0KTm9ib2R5IGNhbiBoZWxwIHlv
Error while writing note
AES-NI not supported on this architecture. If you are using the MSVC toolchain, this is because the AES-NI metho
Invalid AES key size.
host unreachable
connection reset
/proc/self/exe
openserver
windows
program files
recycle.bin
programdata
appdata
all users
Encrypting file:
How to use:
```

```
 -file /home/user/Desktop/file.txt (Encrypts file.txt in /home/user/Desktop directory)
 -dir /home/user/Desktop/ (Encrypts /home/user/Desktop/ directory)
```

The base64-encoded string decodes to the ransom note:

```
 [WHAT HAPPENED?]

All your files were moved to secure storage.
Nobody can help you, except us.
We have private key, we have your black shit.
We are strongly advice you to be interested in safety of your files, as we can show your real face.

[WHAT DO WE NEED?]

Admission, respect and money.
Your information costs money.

[WHO ARE WE?]
A little team of people who can show your problems.

[HOW TO REACH AN AGREEMENT WITH YOU?]

Send us a message with those e-mails:
         givefishtoaman666@protonmail.com
         givehooktoaman666@protonmail.com
```

IDA Land

When analyzing Rust binaries, there are some notes to keep in mind:

- Unlike C or C++-based binaries, it is not easy to navigate Rust-based binaries in a top-down approach, i.e. start at the top and analyze your way down. This is because Rust adds a bunch of runtime code (error handling, memory-safe management, etc.) that pollutes the disassembly.
- Within the same Rust binary, there can exist multiple calling conventions.
- IDA (atleast Home 7.7) may not have the capability to identify Rust library functions, so they are all marked as regular functions. You might end up analyzing code for 2 hours that ends up being runtime or library code.

The previous IDAPython script comes in handy to identify points from where you can start analysis. I could navigate to the string location in the `.rodata` segment, cross-reference to the source which loads that string and then analyze that piece of code rather than starting at the top. I started my analysis with the code that references the base64-encoded string of the ransom note. I hoped this would position me in the neighborhood of the code that does the encryption.

Writing Ransom Note

As mentioned before, the binary contains the base64-encoded form of the ransom note. It decodes it and then writes it into a file named `readme-Luna.txt`.

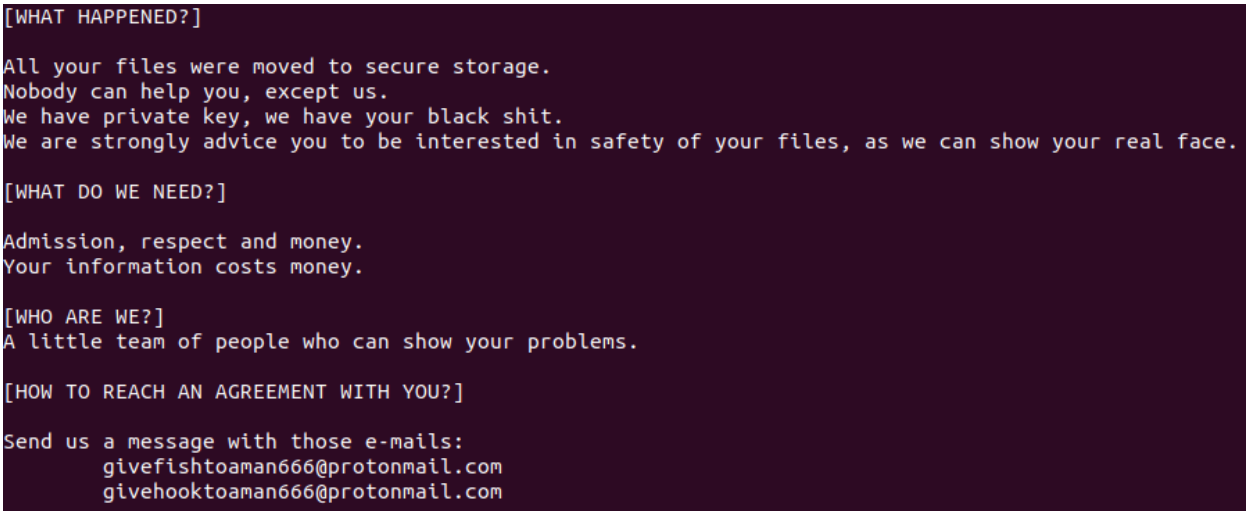

Fig. 7: Luna Ransom Note

Skips Files and Directories

Luna doesn't encrypt files which have:

- substring `Luna` in their filenames
- one of the following extensions:
  - `.dll`
  - `.exe`
  - `.lnk`
  - `.ini`

Luna doesn't encrypt files under directories which contain one of the following substrings:

- `openserver`
- `windows`
- `program files`
- `recycle.bin`
- `programdata`
- `appdata`
- `all users`

Encryption Scheme

Luna employs an encryption scheme that is commonly found in the ransomware world. It leverages both asymmetric and symmetric cryptography, i.e., the key for the symmetric cryptography is derived is from asymmetric cryptography. It uses curve25519-dalek package for Elliptic-Curve Cryptography (ECC) and crypto::aes module for AES-256 CTR-mode cryptography.

Luna's encryption scheme can be summarized as follows:

- It generates a public and private key on Curve25519.
- The binary also contains the threat actor's Curve25519 public key. Using the generated private key and the threat actor's public key, the sample derives the 32-byte shared secret.
- The shared secret is used as the key for AES-256 CTR-mode. The IV is a string slice (a 16-byte fat pointer) pointing to a string literal, `Luna`.

Both the shared secret and the generated public key are zero'd in memory to prevent data leak. As I was writing this, I remembered Javier Yuste's Avaddon ransomware decryption tool which relied on key information being available in memory. Perhaps, zero'ing key information in memory is Luna's safeguard against such decryption tools.

File Encryption

Luna encrypts 50,000 bytes of plaintext file contents at a time. Since AES is in CTR mode, i.e., a stream cipher, the output ciphertext size is equal to the input plaintext size.

For the threat actor's decryption tool to work, the ransomware binary has to store encryption-related information in the encrypted file. In this case, the threat actor would need two points of information per encrypted file:

- Curve25519 32-byte public key generated by the binary to calculate the shared secret used to encrypt the file.
- The IV value used by AES-256 CTR-mode encryption. To this end, the Luna binary stores the previously generated Curve25519 32-byte public key and the IV string literal, `Luna` to the end of the encrypted file.

Each encrypted file is given the extension, `.Luna`.

Peculiarities

Capability Peculiarities

When I hear of ransomware targeting VMware ESXi, I usually come across capability in the binary to shut down running VMs. This helps in clean encryption of files. However, Luna doesn't seem to contain any such capability which may result in encrypted files being corrupted and incapable of being recovered.

Execution Peculiarities

I was wrapping up this article when I noticed a few peculiarities in Luna's execution.

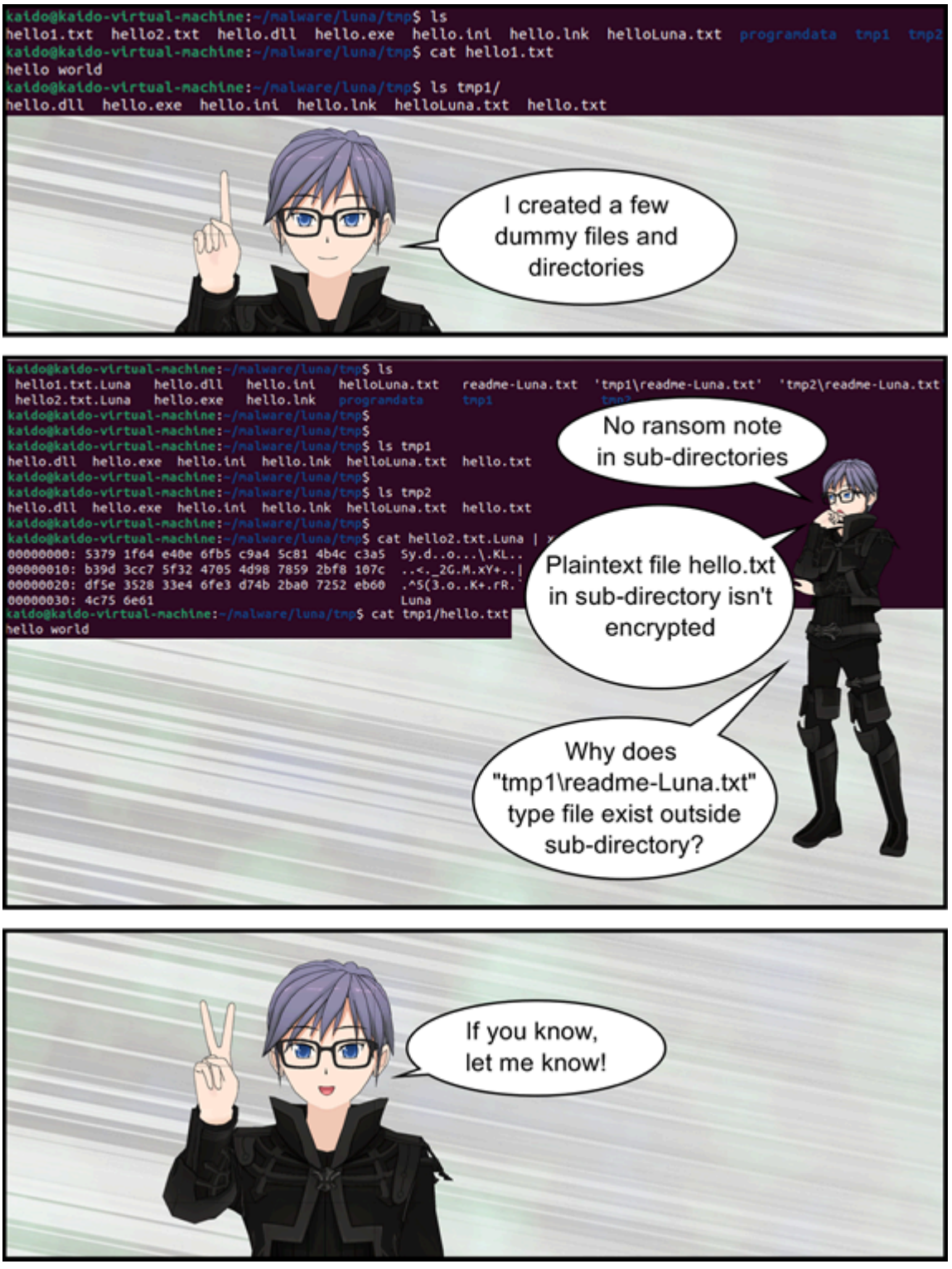

Fig. 8: Execution Peculiarities

Summary

In this article, we looked at a sample of Luna ransomware.

- It used an encryption scheme of Curve25519 (asymmetric cryptography) and AES-256 CTR-mode (symmetric cryptography).

- Unlike popular ransomware families like BlackBasta and BlackCat, Luna doesn't use intermittent encryption strategies and encrypts the entire file.
- Curve25519 public key and AES-256 IV values are stored at the end of the encrypted file.
- Like most ransomware families, certain files and directories are not encrypted.
- I also introduced you to strings in Rust. I presented an IDAPython script that prints UTF-8 strings found in the `.rodata` segment of a Rust-based binary. These results gave a start point for our analysis.
- Finally, we looked at a few peculiarities (or bugs) in Luna's code which makes for buggy directory traversal.

In summary, Luna is the typical ransomware but with bugs and no optimizations.

References

- https://doc.rust-lang.org/rust-by-example/std/str.html
- What is a fat pointer?
- https://docs.rs/c_str/latest/c_str/
- Exploring Strings in Rust
- Understanding String and &str in Rust
- Strings in Rust FINALLY EXPLAINED!
- A Deep Dive into X25519
