

Forum:[Hack Forums](#)

Thread: [\[RUST\] Compile-time String Encryption](#)

Klonopin - January 12, 2025, 8:45 pm

A Rust string encryption macro that encrypts strings at compile-time and decrypts them only when accessed at runtime.

Cargo.toml

Spoiler [\(Click to View\)](#)

Code [lib]

```
proc-macro = true
```

```
[dependencies]
```

```
proc-macro2 = "1.0"
```

```
quote = "1.0"
```

```
syn = { version = "1.0", features = ["full"] }
```

```
rand = "0.8"
```

lib.rs (macro)

Spoiler [\(Click to View\)](#)

```
Code use proc_macro::TokenStream;
```

```
use quote::quote;
```

```
use rand::Rng;
```

```
use syn::{parse_macro_input, LitStr};
```

```
#[proc_macro]
```

```
pub fn encrypt(input: TokenStream) -> TokenStream {
```

```
    let input = parse_macro_input!(input as LitStr);
```

```
    let input_string = input.value();
```

```
    if input_string.is_empty() {
```

```
        return TokenStream::from(quote! {
```

```
            compile_error!("Empty string is not allowed")
```

```
        });
```

```
    }
```

```
    let mut rng = rand::thread_rng();
```

```
    let key1: Vec<u8> = (0..16).map(|_| rng.gen()).collect();
```

```
    let key2: Vec<u8> = (0..16).map(|_| rng.gen()).collect();
```

```
    let encrypted: Vec<u8> = input_string
```

```
        .bytes()
```

```
        .enumerate()
```

```
        .map(|(i, b)| {
```

```
            let k1 = key1[i % key1.len()];
```

```
            let k2 = key2[i % key2.len()];
```

```
            let b1 = b ^ k1;
```

```

        let b2 = b1.rotate_left(3);
        let b3 = b2.wrapping_add(k2);
        let b4 = b3 ^ (i as u8);
        b4.rotate_right(2)
    })
    .collect();

let key1_literals: Vec<proc_macro2::TokenStream> = key1.iter().map(|&x| quote! { #x }).collect();
let key2_literals: Vec<proc_macro2::TokenStream> = key2.iter().map(|&x| quote! { #x }).collect();
let encrypted_literals: Vec<proc_macro2::TokenStream> = encrypted.iter().map(|&x| quote! { #x
}).collect();
let encrypted_len = encrypted.len();

TokenStream::from(quote! {
    {
        const fn decrypt_byte(b: u8, k1: u8, k2: u8, pos: usize) -> u8 {
            let b1 = b.rotate_left(2);
            let b2 = b1 ^ (pos as u8);
            let b3 = b2.wrapping_sub(k2);
            let b4 = b3.rotate_right(3);
            b4 ^ k1
        }

        #[inline(never)]
        fn get_decrypted() -> String {
            const KEY1: [u8; 16] = [ #(key1_literals),* ];
            const KEY2: [u8; 16] = [ #(key2_literals),* ];
            const DATA: [u8; #encrypted_len] = [ #(encrypted_literals),* ];

            let mut result = Vec::with_capacity(#encrypted_len);
            let mut i = 0;
            while i < #encrypted_len {
                let dec = decrypt_byte(
                    DATA[i],
                    KEY1[i % 16],
                    KEY2[i % 16],
                    i
                );
                result.push(dec);
                i += 1;
            }

            unsafe { String::from_utf8_unchecked(result) }
        }

        const fn obfuscate<const N: usize>(data: [u8; N]) -> [u8; N] {
            let mut result = [0u8; N];
            let mut i = 0;
            while i < N {
                result[i] = data[i];
                i += 1;
            }
            result
        }
    }
}

```

```

    }
    get_decrypted()
}
})
}

```

Example

Code use `string_encryption::encrypt`;

```

fn main() {
    println!("Encrypted message: {}", encrypt!("This is a secret message!!1"));
}

```

Compile-Time Encryption Uses proc-macro to process strings during compilation

Generates random encryption keys for each string

No plaintext strings stored in the binary

Multi-Layer Encryption Two separate 16-byte random keys

Multiple transformations per byte: XOR encryption

Bit rotation

Position-based scrambling

Wrapping addition

Performance Zero runtime encryption overhead

Minimal decryption cost

Limitations Only works with string literals

Each encryption is unique (same string encrypted differently each compile)

Requires proc-macro support

Hackers - January 13, 2025, 9:37 am

Nice, nice, a very useful method for any language. Glad to see you knocked this out in Rust.

Infinity. - January 19, 2025, 9:47 pm

Yes, but there is better way around it without having to use those dependencies

```
Code #[macro_export]
macro_rules! obfs_str {
  ($string:expr) => {{
    const KEY: u8 = 0xAA;
    const OBFUSCATED: &'static [u8] = &{
      let mut obfuscated = [0u8; $string.len()];
      let bytes = $string.as_bytes();
      let mut i = 0;
      while i < $string.len() {
        obfuscated[i] = bytes[i] ^ KEY;
        i += 1;
      }
      obfuscated
    };
    String::from_utf8(OBFUSCATED.iter().map(|b| b ^ KEY).collect()).unwrap()
  }};
}
```

and use it like you do

Code let str = obfs_str!("debugger found");

Now, I know i am using a static XOR key (0xAA) and you employ multi-step obfs, we can certainly introduce complexity in my solution without performance overhead.

but your solution is Computationally heavier due to complex obfuscation. I dont know why write like that for Performance.

However with your solution we can obfs really really long strings though

Klonopin - January 20, 2025, 7:09 am

(Jan 19, 2025 10:47 PM)Infinity. Wrote: [CITE](#) Yes, but there is better way around it without having to use those dependencies

```
Code #[macro_export]
macro_rules! obfs_str {
  ($string:expr) => {{
    const KEY: u8 = 0xAA;
    const OBFUSCATED: &'static [u8] = &{
      let mut obfuscated = [0u8; $string.len()];
      let bytes = $string.as_bytes();
      let mut i = 0;
      while i < $string.len() {
        obfuscated[i] = bytes[i] ^ KEY;
        i += 1;
      }
      obfuscated
    };
    String::from_utf8(OBFUSCATED.iter().map(|b| b ^ KEY).collect()).unwrap()
  }};
}
```

and use it like you do

Code let str = obfs_str!("debugger found");

Now, I know i am using a static XOR key (0xAA) and you employ multi-step obfs, we can certainly introduce complexity in my solution without performance overhead.

but your solution is Computationally heavier due to complex obfuscation. I dont know why write like that for Performance.

However with your solution we can obfs really really long strings though

Thanks for sharing your solution. It's clean and efficient perfect for cases where minimalism is the priority, though it focuses more on simplicity than security.

My approach prioritizes security with dynamic keys and multi-step obfuscation, making reverse engineering harder. The performance impact remains negligible unless you're decrypting many large strings at runtime.

Thanks again for HQ share!

Bastrile - February 28, 2025, 8:19 am

is this similar to <https://crates.io/crates/obfstr> ?

Chacha20 > AES > XOR ?

Tha Sneak - February 28, 2025, 6:13 pm

(Feb 28, 2025 09:19 AM)Gynamak Wrote: [\[CITE\]](#) is this similar to <https://crates.io/crates/obfstr> ?

Chacha20 > AES > XOR ?

No it's similar to you not knowing how to code even if I paid a professional programming teacher they still couldn't show you the way without a miracle.

Bastrile - March 7, 2025, 6:15 am

(Feb 28, 2025 07:13 PM)Tha Sneak Wrote: [\[CITE\]](#) No it's similar to you not knowing how to code even if I paid a professional programming teacher they still couldn't show you the way without a miracle.

Don't read coding stuff too much, you will confuse yourself and maybe you will forget your clever methods for "removing malware". Maybe you will forget how to use Geek Uninstaller to force remove malware registry.

I assume geek.exe out of support so you are in huge problem these days...

Infinity - March 13, 2025, 7:22 am

Btw, I found a similar package doing what the thread suggests and more!

<https://github.com/anvie/litcrypt.rs>

Klonopin - March 13, 2025, 9:49 am

(Mar 13, 2025 07:22 AM)Infinity. Wrote: [\[CITE\]](#) Btw, I found a similar package doing what the thread suggests and more!

<https://github.com/anvie/litcrypt.rs>

Yes, this is one of the projects that inspired me to create my own.

Infinity - March 13, 2025, 10:49 am

(Mar 13, 2025 09:49 AM)Klonopin Wrote: [\[CITE\]](#) Yes, this is one of the projects that inspired me to

create my own.

That's awesome mate, let me know if you need some help or are stuck at something

PELock - April 10, 2026, 2:08 am

how about real polymorphic rust string encryption

```
Code // encrypted with https://www.stringencrypt.com (v1.5.0) [Rust]
// secret = "Rust polymorphic string encryption"
let mut YITit: [u16; 35] = [ 0x6F65, 0xA401, 0x8F00, 0x8125, 0x3877, 0xBD48, 0xC551, 0x6F0B,
                             0x3F64, 0x22C8, 0xCCEC, 0x58EC, 0xB07D, 0x2CD8, 0x51E8, 0x85D5,
                             0x8A37, 0x9036, 0x0A4B, 0xA642, 0x9E3D, 0x121F, 0x69BD, 0x464F,
                             0x412D, 0xDEd5, 0x0F3B, 0x3DF8, 0x4185, 0x71C2, 0x93B7, 0xFEBA,
                             0xC388, 0x2682, 0xE3E7 ];

let mut Aiptj: u16 = 0;
for IsQOo in 0..35
{
    Aiptj = YITit[IsQOo];
    for KWRyH in 0..4
    {
        Aiptj = (Aiptj & 0xFFFF).wrapping_add(IsQOo as u16) & 0xFFFF;
        Aiptj = (Aiptj & 0xFFFF).wrapping_sub(0x6D20) & 0xFFFF;
    }
    Aiptj = (Aiptj & 0xFFFF).wrapping_add(IsQOo as u16) & 0xFFFF;
    Aiptj = (Aiptj ^ 0xFFFF) & 0xFFFF;
    Aiptj = (Aiptj & 0xFFFF).wrapping_sub(IsQOo as u16) & 0xFFFF;
    for caOPY in 0..4
    {
        for OzgHv in 0..4
        {
            for dXzWl in 0..4
            {
                Aiptj = (Aiptj & 0xFFFF).wrapping_add(0xC3CD) & 0xFFFF;
                Aiptj = (Aiptj ^ 0x96EF) & 0xFFFF;
                Aiptj = ((Aiptj & 0xFFFF).wrapping_shr((IsQOo as u32) % 16u32) | (Aiptj &
0xFFFF).wrapping_shl((16u32).wrapping_sub((IsQOo as u32) % 16u32).wrapping_rem(16u32))) &
0xFFFF;
            }
            Aiptj = (Aiptj & 0xFFFF).wrapping_add(IsQOo as u16) & 0xFFFF;
            for HWD0z in 0..2
            {
                Aiptj = ((Aiptj ^ 0xFFFF).wrapping_add(1)) & 0xFFFF;
            }
        }
        Aiptj = (Aiptj & 0xFFFF).wrapping_add(0x02CE) & 0xFFFF;
    }
    Aiptj = (Aiptj ^ (((Aiptj & 0xFFFF).wrapping_shl((14u32) & 15)) & 0xFFFF)) & 0xFFFF;
    YITit[IsQOo] = Aiptj;
}
let secret = String::from_utf16_lossy(&YITit[..]);
```