Comparing changes

Choose two branches to see what's changed or to start a new pull request. If you need to, you can also compare across forks or learn more about diff comparisons.

base: v0.4.3 ... compare: v0.4.4

8 commits **9** files changed **1** contributor

Commits on Mar 3, 2023

Random float is now usable in const context (#50)
CasualX authored on Mar 4, 2023
Verified df094e2

Force random const eval (#51)
CasualX authored on Mar 4, 2023
Verified 04b49df

Commits on Mar 8, 2023

Improve docs and stuff (#52)
CasualX authored on Mar 9, 2023
Verified 614beed

Commits on Mar 15, 2023

Refactor simplify (#53)
CasualX authored on Mar 15, 2023
Verified 087aac4

Commits on May 30, 2023

Helper macro to obfuscate to an owned String instead of a temporary &... ...
CasualX authored on May 31, 2023
Verified ec1a20b

Commits on Oct 5, 2024

Restrict obfstr! argument type to string slices (#61)
CasualX authored on Oct 5, 2024
Verified b57034c

Support obfuscating cstr literals (#58)
CasualX authored on Oct 5, 2024 ✓ Verified 3924cd2

Release v0.4.4 (#62)
CasualX authored on Oct 5, 2024 ✓ Verified 5431a54

Showing **9 changed files** with **152 additions** and **91 deletions**. Split **Unified**

2 Cargo.toml

```
@@ -1,6 +1,6 @@
 [package]
 name = "obfstr"
-version = "0.4.3"
+version = "0.4.4"
 edition = "2021"
 license = "MIT"
```

8 examples/stringify.rs

```
 */

 use std::fmt;
-use obfstr::{obfstr, position};
+use obfstr::{obfstr, obfstring, position};

 // Let's try to obfuscate the string representation of this enum.
 pub enum Example {
         // Returns an owned String but this allocates memory.
         pub fn to_str1(&self) -> String {
                 match self {
-                        Example::Foo => String::from(obfstr!("Foo")),
-                        Example::Bar => String::from(obfstr!("Bar")),
-                        Example::Baz => String::from(obfstr!("Baz")),
+                        Example::Foo => obfstring!("Foo"),
+                        Example::Bar => obfstring!("Bar"),
+                        Example::Baz => obfstring!("Baz"),
                 }
         }

```

5 readme.md

```
This reference to a temporary value must be used in the same statement it was generated.
See the documentation for more advanced use cases.

- If you're looking for obfuscating format strings (`format!`, `println!`, etc.) I have another crate [`fmtools`](https://crates.io/crates/fmtools) with the optional
```

```
dependency `obfstr` enabled to automatically apply string obfuscation to your
formatting strings.
+ Looking for obfuscating formatting strings? See `fmtools` ([github]
(https://github.com/CasualX/fmtools), [crates.io](https://crates.io/crates/fmtools),
[docs.rs](https://docs.rs/fmtools/0.1.2/fmtools/)) with the optional dependency
`obfstr` enabled to automatically apply string obfuscation to formatting strings.

Examples
--------

The `obfstr!` macro returns the deobfuscated string as a temporary value:

```rust
- assert_eq!(obfstr::obfstr!("Hello 🌍"), "Hello 🌍");
+ use obfstr::obfstr as s;
+ assert_eq!(s!("Hello 🌍"), "Hello 🌍");
```

The `wide!` macro provides compiletime utf16 string constants:
```

84 ■■■■□ src/bytes.rs

```
/// The `obfstr!` macro returns the deobfuscated string as a temporary `&str`
value and must be consumed in the same statement it was used:
///
/// ```
- /// use obfstr::obfstr;
+ /// use obfstr::obfstr as s;
///
/// const HELLO_WORLD: &str = "Hello 🌍";
- /// assert_eq!(obfstr!(HELLO_WORLD), HELLO_WORLD);
+ /// assert_eq!(s!(HELLO_WORLD), HELLO_WORLD);
/// ```
///
/// Different syntax forms are supported to reuse the obfuscated strings in outer
scopes:
///
/// ```
- /// use obfstr::obfstr;
+ /// use obfstr::obfstr as s;
///
/// // Obfuscate a bunch of strings
- /// obfstr! {
+ /// s! {
///     let s = "Hello world";
///     let another = "another";
/// }
/// // Assign to an uninit variable in outer scope
/// let (true_string, false_string);
/// let string = if true {
-///     obfstr!(true_string = "true")
+///     s!(true_string = "true")
```

```
40  40  /// }
41  41  /// else {
42      - ///     obfstr!(false_string = "false")
    42  + ///     s!(false_string = "false")
43  43  /// };
44  44  /// assert_eq!(string, "true");
45  45  ///
46  46  /// // Return an obfuscated string from a function
47  47  /// fn helper(buf: &mut [u8]) -> &str {
48      - ///     obfstr!(buf <- "hello")
    48  + ///     s!(buf <- "hello")
49  49  /// }
50  50  /// let mut buf = [0u8; 16];
51  51  /// assert_eq!(helper(&mut buf), "hello");
52  52  /// ```
53  53  #[macro_export]
54  54  macro_rules! obfstr {
55  55          ($(let $name:ident = $s:expr;)*) => {$(
56      -                   $crate::obfbytes! { let $name = $s.as_bytes(); }
    56  +                   $crate::obfbytes! { let $name = ::core::convert::identity::<&str>($s).as_bytes(); }
57  57                      let $name = $crate::unsafe_as_str($name);
58  58          )*};
59  59          ($name:ident = $s:expr) => {
60      -                   $crate::unsafe_as_str($crate::obfbytes!($name = $s.as_bytes()))
    60  +                   $crate::unsafe_as_str($crate::obfbytes!($name = ::core::convert::identity::<&str>($s).as_bytes()))
61  61          };
62  62          ($buf:ident <- $s:expr) => {
63      -                   $crate::unsafe_as_str($crate::obfbytes!($buf <- $s.as_bytes()))
    63  +                   $crate::unsafe_as_str($crate::obfbytes!($buf <- ::core::convert::identity::<&str>($s).as_bytes()))
64  64          };
65  65          ($s:expr) => {
66      -                   $crate::unsafe_as_str($crate::obfbytes!($s.as_bytes()))
    66  +                   $crate::unsafe_as_str($crate::obfbytes!(::core::convert::identity::<&str>($s).as_bytes()))
    67  +           };
    68  + }
    69  +
    70  + /// Compiletime cstr constant obfuscation.
    71  + ///
    72  + /// See [`obfstr!`] for more information.
    73  + ///
    74  + /// ```
    75  + /// use std::ffi::CStr;
    76  + /// use obfstr::obfcstr as cstr;
    77  + ///
    78  + /// const HELLO_WORLD: &'static CStr = c"Hello CStr";
    79  + /// assert_eq!(cstr!(HELLO_WORLD).to_str().unwrap(), "Hello CStr");
    80  + /// ```
    81  + #[macro_export]
```

```
+ macro_rules! obfcstr {
+         ($(let $name:ident = $s:expr;)*) => {$(
+                 $crate::obfbytes! { let $name =
  ::core::ffi::CStr::to_bytes_with_nul($s); }
+                 let $name = $crate::unsafe_as_cstr($name);
+         )*};
+         ($name:ident = $s:expr) => {
+                 $crate::unsafe_as_cstr($crate::obfbytes!($name =
  ::core::ffi::CStr::to_bytes_with_nul($s)))
+         };
+         ($buf:ident <- $s:expr) => {
+                 $crate::unsafe_as_cstr($crate::obfbytes!($buf <-
  ::core::ffi::CStr::to_bytes_with_nul($s)))
+         };
+         ($s:expr) => {
+                 $crate::unsafe_as_cstr($crate::obfbytes!
  (::core::ffi::CStr::to_bytes_with_nul($s)))
+         };
+ }
+
+ /// Compiletime string constant obfuscation.
+ ///
+ /// Returns an owned `String` instead of a temporary `&str`.
+ ///
+ /// See [`obfstr!`] for more information.
+ #[macro_export]
+ macro_rules! obfstring {
+         ($s:expr) => {
+                 String::from($crate::obfstr!($s))
          };
  }

          ($s:expr) => {{
                  const _OBFBYTES_STRING: &[u8] = $s;
                  const _OBFBYTES_LEN: usize = _OBFBYTES_STRING.len();
-                 const _OBFBYTES_KEYSTREAM: [u8; _OBFBYTES_LEN] =
  $crate::bytes::keystream::<_OBFBYTES_LEN>($crate::__entropy!("key", stringify!
  ($s)) as u32);
+                 const _OBFBYTES_KEYSTREAM: [u8; _OBFBYTES_LEN] =
  $crate::bytes::keystream::<_OBFBYTES_LEN>($crate::random!(u32, "key", stringify!
  ($s)));
                  static _OBFBYTES_SDATA: [u8; _OBFBYTES_LEN] =
  $crate::bytes::obfuscate::<_OBFBYTES_LEN>(_OBFBYTES_STRING,
  &_OBFBYTES_KEYSTREAM);
                  $crate::bytes::deobfuscate::<_OBFBYTES_LEN>(
-                         $crate::__xref!(
-                                 $crate::__entropy!("offset", stringify!($s)) as
  usize,
-                                 $crate::__entropy!("xref", stringify!($s)),
-                                 &_OBFBYTES_SDATA),
+                         $crate::xref::xref::<_,
+                                 {$crate::random!(u32, "offset", stringify!($s))},
```

```
141 + {$crate::random!(u64, "xref", stringify!($s))}>
142 + (&_OBFBYTES_SDATA),
103 143 &_OBFBYTES_KEYSTREAM)
104 144 }};
105 145 }
111 151 x ^= x << 13;
112 152 x ^= x >> 17;
113 153 x ^= x << 5;
114 - x
154 + return x;
115 155 }
116 156
117 157 /// Generate the key stream for array of given length.
148 188 },
149 189 _ => (),
150 190 }
151 - keys
191 + return keys;
152 192 }
153 193
154 194 /// Obfuscates the input string and given key stream.
163 203 data[i] = s[i] ^ k[i];
164 204 i += 1;
165 205 }
166 - data
206 + return data;
167 207 }
168 208
169 209 /// Deobfuscates the obfuscated input string and given key stream.
170 210 #[inline(always)]
171 211 pub fn deobfuscate<const LEN: usize>(s: &[u8; LEN], k: &[u8; LEN]) -> [u8; LEN] {
172 - let mut buffer = [0u8; LEN];
212 + let mut buf = [0u8; LEN];
173 213 let mut i = 0;
174 214 // Try to tickle the LLVM optimizer in _just_ the right way
175 215 // Use `read_volatile` to avoid constant folding a specific read and optimize the rest
176 216 // Volatile reads of any size larger than 8 bytes appears to cause a bunch of one byte reads
177 217 // Hand optimize in chunks of 8 and 4 bytes to avoid this
178 218 unsafe {
179 219 let src = s.as_ptr();
180 - let dest = buffer.as_mut_ptr();
220 + let dest = buf.as_mut_ptr();
181 221 // Process in chunks of 8 bytes on 64-bit targets
182 222 #[cfg(target_pointer_width = "64")]
183 223 while i < LEN & !7 {
219 259 _ => (),
220 260 }
221 261 }
222 - buffer
262 + return buf;
```

```
223 263   }
224 264
225 265   #[inline(always)]
```

2 src/cfo.rs

```
46  46    macro_rules! obfstmt {
47  47            ($($stmt:stmt;)*) => {{
48  48                    // Initial KEY and XOR values
49      -             const _OBFSTMT_KEY: u32 = $crate::__entropy!(stringify!
          ($($stmt;)*)) as u32;
        49  +         const _OBFSTMT_KEY: u32 = $crate::random!(u32, stringify!
          ($($stmt;)*));
50  50                const _OBFSTMT_XOR: u32 = $crate::murmur3(b"XOR", _OBFSTMT_KEY);
51  51                // Count the number of statements
52  52                const _OBFSTMT_LEN: usize = <[&'static str]>::len(&[$(stringify!
          ($stmt)),*]);
```

67 src/lib.rs

```
5   5     #![cfg_attr(not(test), no_std)]
6   6
7   7     use core::str;
        8   + use core::ffi::CStr;
8   9
9   10    #[doc(hidden)]
10  11    pub mod wide;
30  31    /// The integer types generate a random value in their respective range.
31  32    /// The float types generate a random value in range of `[1.0, 2.0)`.
32  33    ///
33      - /// While the result is generated at compiletime only the integer types are
          available in const contexts.
34      - ///
35  34    /// ```
36  35    /// const RND: i32 = obfstr::random!(u8) as i32;
37  36    /// assert!(RND >= 0 && RND <= 255);
        37  + /// # const _: f32 = obfstr::random!(f32);
        38  + /// # const _: f64 = obfstr::random!(f64);
38  39    /// ```
39  40    ///
40  41    /// The behavior of the macro inside other macros can be surprising:
81  82    #[macro_export]
82  83    macro_rules! random {
83  84            ($ty:ident $(, $seeds:expr)* $(,)?) => {{
84      -             const _RANDOM_ENTROPY: u64 = $crate::entropy(concat!(file!(),
          ":", line!(), ":", column!() $(, ":", $seeds)*));
85      -             $crate::__random_cast!($ty, _RANDOM_ENTROPY)
        85  +         const _RANDOM: $ty = $crate::__random_cast!($ty,
        86  +                 $crate::entropy(concat!(file!(), ":", line!(), ":",
          column!() $(, ":", $seeds)*)));
        87  +         _RANDOM
86  88            }};
```

```
87  89    }
88  90
92  94            (u8, $seed:expr) => { $seed as u8 };
93  95            (u16, $seed:expr) => { $seed as u16 };
94  96            (u32, $seed:expr) => { $seed as u32 };
95      -         (u64, $seed:expr) => { $seed as u64 };
    97  +         (u64, $seed:expr) => { $seed };
96  98            (usize, $seed:expr) => { $seed as usize };
97  99            (i8, $seed:expr) => { $seed as i8 };
98  100           (i16, $seed:expr) => { $seed as i16 };
99  101           (i32, $seed:expr) => { $seed as i32 };
100 102           (i64, $seed:expr) => { $seed as i64 };
101 103           (isize, $seed:expr) => { $seed as isize };
102 104           (bool, $seed:expr) => { $seed as i64 >= 0 };
103     -         (f32, $seed:expr) => { f32::from_bits(0b0_01111111 <<
    (f32::MANTISSA_DIGITS - 1) | ($seed as u32 >> 9)) };
104     -         (f64, $seed:expr) => { f64::from_bits(0b0_01111111111 <<
    (f64::MANTISSA_DIGITS - 1) | ($seed >> 12)) };
    105 +
    106 +         // {f32, f64}::from_bits is unstable as const fn due to issues with NaN
    107 +         (f32, $seed:expr) => { unsafe { ::core::mem::transmute::<u32, f32>
    (0b0_01111111 << (f32::MANTISSA_DIGITS - 1) | ($seed as u32 >> 9)) } };
    108 +         (f64, $seed:expr) => { unsafe { ::core::mem::transmute::<u64, f64>
    (0b0_01111111111 << (f64::MANTISSA_DIGITS - 1) | ($seed >> 12)) } };
105 109
106 110           ($ty:ident, $seed:expr) => { compile_error!(concat!("unsupported type: ",
    stringify!($ty))) };
107 111   }
108 112
    113 + #[test]
    114 + fn test_random_f32() {
    115 +         #[track_caller]
    116 +         fn t(v: f32) {
    117 +                 assert!(v >= 1.0 && v < 2.0, "{}", v);
    118 +         }
    119 +         use random as r;
    120 +         t(r!(f32));t(r!(f32));t(r!(f32));t(r!(f32));t(r!(f32));t(r!(f32));t(r!
    (f32));t(r!(f32));
    121 +         t(r!(f32));t(r!(f32));t(r!(f32));t(r!(f32));t(r!(f32));t(r!(f32));t(r!
    (f32));t(r!(f32));
    122 +         t(r!(f32));t(r!(f32));t(r!(f32));t(r!(f32));t(r!(f32));t(r!(f32));t(r!
    (f32));t(r!(f32));
    123 +         t(r!(f32));t(r!(f32));t(r!(f32));t(r!(f32));t(r!(f32));t(r!(f32));t(r!
    (f32));t(r!(f32));
    124 + }
    125 +
    126 + #[test]
    127 + fn test_random_f64() {
    128 +         #[track_caller]
    129 +         fn t(v: f64) {
    130 +                 assert!(v >= 1.0 && v < 2.0, "{}", v);
    131 +         }
```

```diff
+        use random as r;
+        t(r!(f64));t(r!(f64));t(r!(f64));t(r!(f64));t(r!(f64));t(r!(f64));t(r!(f64));t(r!(f64));
+        t(r!(f64));t(r!(f64));t(r!(f64));t(r!(f64));t(r!(f64));t(r!(f64));t(r!(f64));t(r!(f64));
+        t(r!(f64));t(r!(f64));t(r!(f64));t(r!(f64));t(r!(f64));t(r!(f64));t(r!(f64));t(r!(f64));
+        t(r!(f64));t(r!(f64));t(r!(f64));t(r!(f64));t(r!(f64));t(r!(f64));t(r!(f64));t(r!(f64));
+ }
+
  /// Compiletime bitmixing.
  ///
  /// Takes an intermediate hash that may not be thoroughly mixed and increase its entropy to obtain both better distribution.
         splitmix(SEED ^ splitmix(hash(string) as u64))
  }

- /// Produces pseudorandom entropy from the argument literals.
- #[doc(hidden)]
- #[macro_export]
- macro_rules! __entropy {
-        ($($seeds:expr),* $(,)?) => {
-                $crate::entropy(concat!(file!(), ":", line!(), ":", column!() $(, ":", $seeds)*))
-        };
- }
-
  /// Compiletime RNG seed.
  ///
  /// This value is derived from the environment variable `OBFSTR_SEED` and has a fixed value if absent.

  #[doc(hidden)]
  #[inline(always)]
- pub fn unsafe_as_str(bytes: &[u8]) -> &str {
+ pub const fn unsafe_as_str(bytes: &[u8]) -> &str {
         // When used correctly by this crate's macros this should be safe
         #[cfg(debug_assertions)]
-        return str::from_utf8(bytes).unwrap();
+        return match str::from_utf8(bytes) { Ok(s) => s, Err(_) => panic!("invalid str") };
         #[cfg(not(debug_assertions))]
         return unsafe { str::from_utf8_unchecked(bytes) };
  }
+
+ #[doc(hidden)]
+ #[inline(always)]
+ pub const fn unsafe_as_cstr(bytes: &[u8]) -> &CStr {
+        // When used correctly by this crate's macros this should be safe
+        #[cfg(debug_assertions)]
```

```
216 +         return match CStr::from_bytes_with_nul(bytes) { Ok(cstr) => cstr, Err(_)
    => panic!("invalid cstr") };
217 +         #[cfg(not(debug_assertions))]
218 +         return unsafe { CStr::from_bytes_with_nul_unchecked(bytes) };
219 + }
```

3 src/murmur3.rs

```
14  14
15  15  /// MurmurHash3 (32-bit variant) keyed hash function.
16  16  #[doc(hidden)]
17      - #[inline]
18  17  pub const fn murmur3(string: &[u8], seed: u32) -> u32 {
19  18          let mut h = seed;
20  19          const C1: u32 = 0xcc9e2d51;
49  48          fmix32(h ^ string.len() as u32)
50  49  }
51  50
52      - #[inline]
    51  + #[inline(always)]
53  52  const fn fmix32(mut h: u32) -> u32 {
54  53          h ^= h >> 16;
55  54          h = h.wrapping_mul(0x85ebca6b);
```

25 src/words.rs

```
3   3   ======================
4   4   */
5   5
    6   + use core::ptr::{read_volatile, write};
    7   +
6   8   /// Compiletime wide string constant obfuscation.
7   9   #[macro_export]
8   10  macro_rules! obfwide {
29  31          ($s:expr) => {{
30  32                  const _OBFWIDE_STRING: &[u16] = $crate::wide!($s);
31  33                  const _OBFWIDE_LEN: usize = _OBFWIDE_STRING.len();
32      -               const _OBFWIDE_KEYSTREAM: [u16; _OBFWIDE_LEN] =
    $crate::words::keystream::<_OBFWIDE_LEN>($crate::__entropy!("key", stringify!
    ($s)) as u32);
    34  +               const _OBFWIDE_KEYSTREAM: [u16; _OBFWIDE_LEN] =
    $crate::words::keystream::<_OBFWIDE_LEN>($crate::random!(u32, "key", stringify!
    ($s)));
33  35                  static _OBFWIDE_SDATA: [u16; _OBFWIDE_LEN] =
    $crate::words::obfuscate::<_OBFWIDE_LEN>(_OBFWIDE_STRING, &_OBFWIDE_KEYSTREAM);
34  36                  $crate::words::deobfuscate::<_OBFWIDE_LEN>(
35      -                       $crate::__xref!(
36      -                               $crate::__entropy!("offset", stringify!($s)) as
    usize,
37      -                               $crate::__entropy!("xref", stringify!($s)),
38      -                               &_OBFWIDE_SDATA),
    37  +                       $crate::xref::xref::<_,
```

```
38 +                                {$crate::random!(u32, "offset", stringify!($s))},
39 +                                {$crate::random!(u64, "xref", stringify!($s))}>
40 +                                (&_OBFWIDE_SDATA),
41                              &_OBFWIDE_KEYSTREAM)
42          }};
43  }
49          x ^= x << 13;
50          x ^= x >> 17;
51          x ^= x << 5;
-           x
52 +         return x;
53  }
54
55  /// Generate the key stream for array of given length.
71                  round_key = next_round(round_key);
72                  keys[i] = round_key as u16;
73          }
-           keys
74 +         return keys;
75  }
76
77  /// Obfuscates the input string and given key stream.
85                  data[i] = s[i] ^ k[i];
86                  i += 1;
87          }
-           data
88 +         return data;
89  }
90
91  /// Deobfuscates the obfuscated input string and given key stream.
92  #[inline(always)]
93  pub fn deobfuscate<const LEN: usize>(s: &[u16; LEN], k: &[u16; LEN]) -> [u16; LEN] {
-           let mut buffer = [0u16; LEN];
94 +         let mut buf = [0u16; LEN];
95          let mut i = 0;
96          // Try to tickle the LLVM optimizer in _just_ the right way
97          // Use `read_volatile` to avoid constant folding a specific read and optimize the rest
98          // Volatile reads of any size larger than 8 bytes appears to cause a bunch of one byte reads
99          // Hand optimize in chunks of 8 and 4 bytes to avoid this
100         unsafe {
-                   use ::core::ptr::{read_volatile, write};
101                 let src = s.as_ptr();
-                   let dest = buffer.as_mut_ptr();
102 +                 let dest = buf.as_mut_ptr();
103                 // Process in chunks of 8 bytes on 64-bit targets
104                 #[cfg(target_pointer_width = "64")]
105                 while i < LEN & !3 {
129                         write(dest.offset(i as isize), ct ^ k[i]);
130                 }
```

```
130  131          }
131       -        buffer
     132  +        return buf;
132  133      }
133  134
134  135      // Test correct processing of less than multiple of 8 lengths
```

47 ■■■■□ src/xref.rs

```
19   19   #[macro_export]
20   20   macro_rules! xref {
21   21           ($e:expr) => {
22       -                $crate::__xref!($crate::__entropy!(stringify!($e), 1) as usize,
         $crate::__entropy!(stringify!($e), 2), $e)
23       -        };
24       - }
25       -
26       - #[doc(hidden)]
27       - #[macro_export]
28       - macro_rules! __xref {
29       -        ($offset:expr, $seed:expr, $e:expr) => {
30       -                $crate::xref::xref::<_, {$offset}, {$seed}>($e)
     22  +                $crate::xref::xref::<_,
     23  +                        {$crate::random!(u32, stringify!($e), "OFFSET")},
     24  +                        {$crate::random!(u64, stringify!($e), "SEED")}>($e)
31   25           };
32   26   }
33   27
34   28   #[inline(always)]
35       - const fn non_zero(rand: usize) -> usize {
     29  + const fn non_zero(rand: u32) -> u32 {
36   30           if rand == 0 { 1 } else { rand }
37   31   }
38   32
39   33   #[inline(always)]
40       - const fn obfchoice(v: usize, seed: u64) -> usize {
41       -        let rand = (seed >> 32) as i32 as usize;
     34  + const fn obfchoice(v: u32, seed: u64) -> u32 {
     35  +        let rand = (seed >> 32) as u32;
42   36           match seed & 7 {
43   37                   0 => v.wrapping_add(rand),
44   38                   1 => rand.wrapping_sub(v),
45   39                   2 => v ^ rand,
46       -                3 => v ^ v.rotate_left(non_zero(rand & 7) as u32),
     40  +                3 => v ^ v.rotate_left(non_zero(rand & 7)),
47   41                   4 => !v,
48   42                   5 => v ^ (v >> non_zero(rand & 31)),
49   43                   6 => v.wrapping_mul(non_zero(rand)),
53   47   }
54   48
55   49   #[inline(always)]
56       - const fn obfuscate<const SEED: u64>(mut v: usize) -> usize {
```

```
+ const fn obfuscate<const SEED: u64>(mut v: u32) -> usize {
          let mut seed = SEED;
          use crate::splitmix;
          seed = splitmix(seed);
          seed = splitmix(seed);
          v = obfchoice(v, seed);
          seed = splitmix(seed);
-         return obfchoice(v, seed & 0xffffffff00000000 | 3) & 0xffff
+         v = obfchoice(v, seed);
+         return (v & 0xffff) as usize
  }

  #[inline(never)]
- fn inner<const SEED: u64>(p: *const u8, offset: usize) -> *const u8 {
+ fn inner<const SEED: u64>(p: *const u8, offset: u32) -> *const u8 {
          p.wrapping_add(obfuscate::<SEED>(offset))
  }

  /// Obfuscates the xref to data reference.
  #[inline(always)]
- pub fn xref<T: ?Sized, const OFFSET: usize, const SEED: u64>(p: &'static T) ->
  &'static T {
+ pub fn xref<T: ?Sized, const OFFSET: u32, const SEED: u64>(p: &'static T) ->
  &'static T {
          unsafe {
                  let mut p: *const T = p;
                  // Launder the values through black_box to prevent LLVM from
  optimizing away the obfuscation
  #[macro_export]
  macro_rules! xref_mut {
          ($e:expr) => {
-                 $crate::__xref_mut!($crate::__entropy!(stringify!($e), 1) as
  usize, $crate::__entropy!(stringify!($e), 2), $e)
-         };
- }
-
- #[doc(hidden)]
- #[macro_export]
- macro_rules! __xref_mut {
-         ($offset:expr, $seed:expr, $e:expr) => {
-                 $crate::xref::xref_mut::<_, {$offset}, {$seed}>($e)
+                 $crate::xref::xref_mut::<_,
+                         {$crate::random!(u32, stringify!($e), "OFFSET")},
+                         {$crate::random!(u64, stringify!($e), "SEED")}>($e)
          };
  }

  #[inline(never)]
- fn inner_mut<const SEED: u64>(p: *mut u8, offset: usize) -> *mut u8 {
+ fn inner_mut<const SEED: u64>(p: *mut u8, offset: u32) -> *mut u8 {
          p.wrapping_add(obfuscate::<SEED>(offset))
  }
```

```
117 106
118 107   /// Obfuscates the xref to data reference.
119 108   #[inline(always)]
120     - pub fn xref_mut<T: ?Sized, const OFFSET: usize, const SEED: u64>(p: &'static mut
        T) -> &'static mut T {
    109 + pub fn xref_mut<T: ?Sized, const OFFSET: u32, const SEED: u64>(p: &'static mut T)
        -> &'static mut T {
121 110           unsafe {
122 111                   let mut p: *mut T = p;
123 112                   // Launder the values through black_box to prevent LLVM from
        optimizing away the obfuscation
140 129   #[test]
141 130   fn regression1() {
142 131           // Caused by `v = v ^ (v >> RNG)` when RNG is zero to always be zero
143     -         let v = obfuscate::<{(-4272872662024917058i64) as u64}>
        (4898264233338431333usize);
    132 +         let v = obfuscate::<0xC4B3B4F3D986EFBEu64>(0x3C236765u32);
144 133           assert_ne!(v, 0);
145 134   }
```