CVE-2021-29939

修复前:

https://github.com/Alexhuszagh/rust-stackvector/blob/d0382d5ef903fc96bdcc08c02e36e6dd2eda11a5/src/lib.rs

```
impl<A: Array> Extend<A::Item> for StackVec<A> {
    fn extend<I: iter::IntoIterator<Item=A::Item>>(&mut self, iterable: I) {
        let mut iter = iterable.into_iter();
        let (lower_bound, upper_bound) = iter.size_hint();
        let upper_bound = upper_bound.expect("iterable must provide upper bound.");
        assert!(self.len() + upper_bound <= self.capacity());

        unsafe {
            let len = self.len();
            let ptr = self.as_mut_ptr().padd(len);
            let mut count = 0;
            while count < lower_bound {
                if let Some(out) = iter.next() {
                    ptr::write(ptr.padd(count), out);
                    count += 1;
                } else {
                    break;
                }
            }
            self.set_len(len + count);
        }

        for elem in iter {
            self.push(elem);
        }
    }
}
```

修复后:

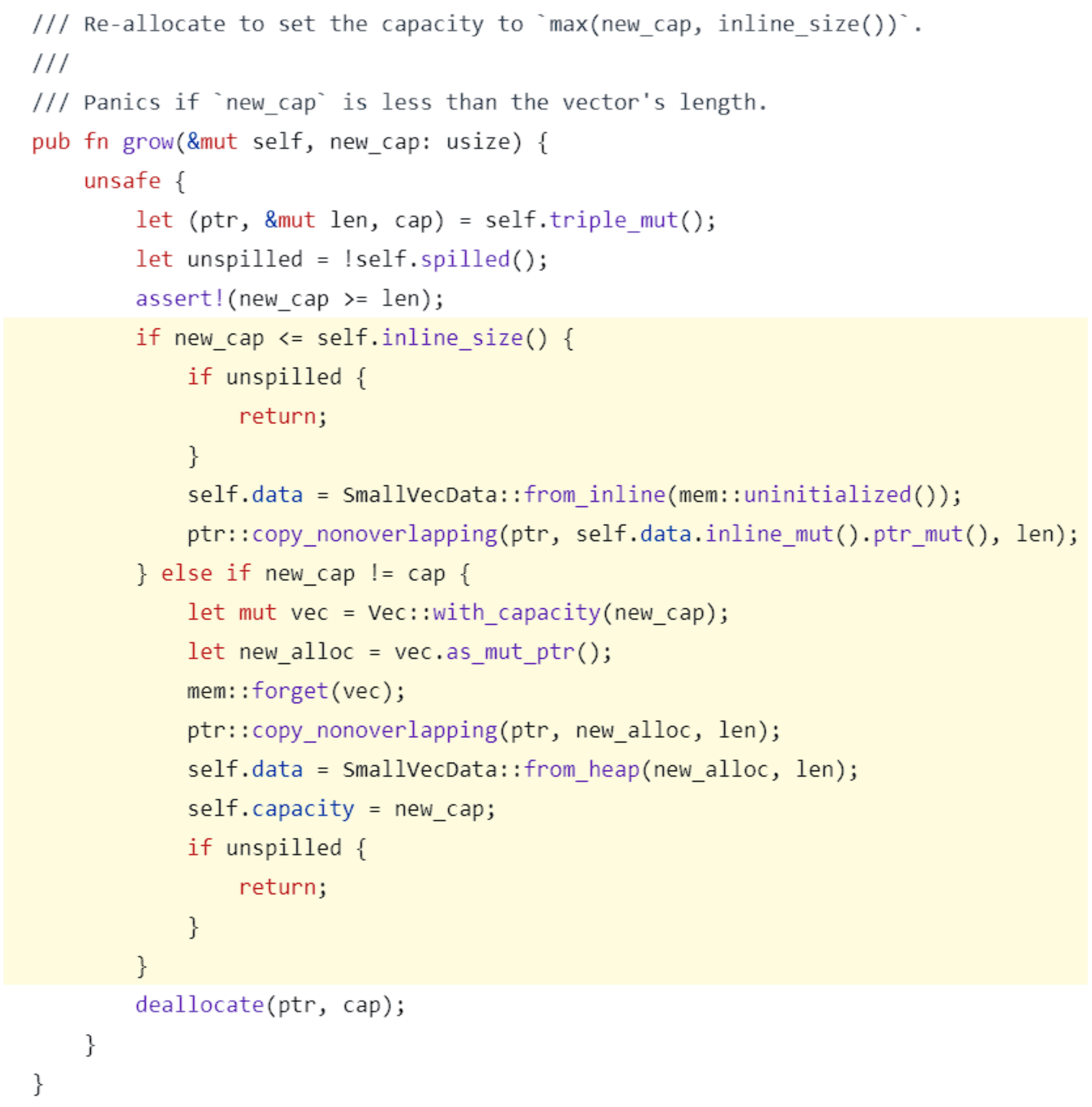
https://github.com/Alexhuszagh/rust-stackvector/blob/f45657d5a823a67bb3f5cffee65efbb401a44192/src/lib.rs

```
impl<A: Array> Extend<A::Item> for StackVec<A> {
    fn extend<I: iter::IntoIterator<Item=A::Item>>(&mut self, iterable: I) {
        // size_hint() has no safety guarantees, and TrustedLen
        // is nightly only, so we can't do any optimizations with
        // size_hint.
        for elem in iterable.into_iter() {
            self.push(elem);
        }
    }
}
```

RUSTSEC-2019-0009 与 RUSTSEC-2019-0012

修复前:

https://github.com/servo/rust-smallvec/blob/19de50108d403efaa7cd979eac3bb97a4432fd4b/lib.rs#L651-L667



```rust
/// Re-allocate to set the capacity to `max(new_cap, inline_size())`.
///
/// Panics if `new_cap` is less than the vector's length.
pub fn grow(&mut self, new_cap: usize) {
    unsafe {
        let (ptr, &mut len, cap) = self.triple_mut();
        let unspilled = !self.spilled();
        assert!(new_cap >= len);
        if new_cap <= self.inline_size() {
            if unspilled {
                return;
            }
            self.data = SmallVecData::from_inline(mem::uninitialized());
            ptr::copy_nonoverlapping(ptr, self.data.inline_mut().ptr_mut(), len);
        } else if new_cap != cap {
            let mut vec = Vec::with_capacity(new_cap);
            let new_alloc = vec.as_mut_ptr();
            mem::forget(vec);
            ptr::copy_nonoverlapping(ptr, new_alloc, len);
            self.data = SmallVecData::from_heap(new_alloc, len);
            self.capacity = new_cap;
            if unspilled {
                return;
            }
        }
        deallocate(ptr, cap);
    }
}
```

修复后:

https://github.com/servo/rust-smallvec/blob/v0.6.10/lib.rs

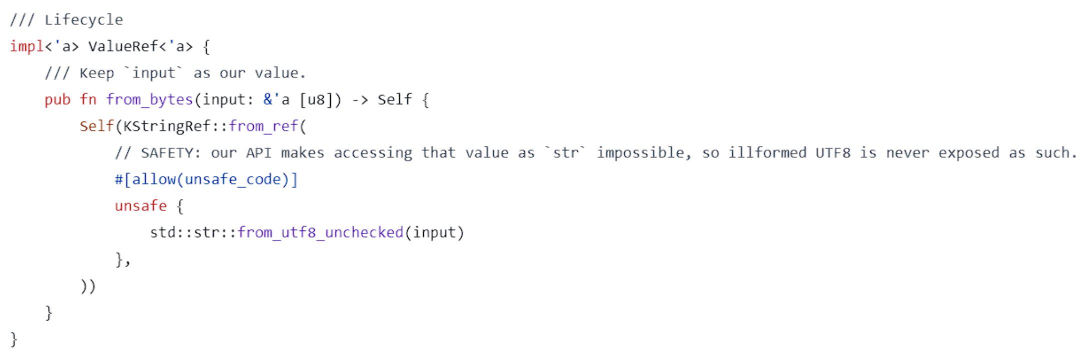
```
/// Re-allocate to set the capacity to `max(new_cap, inline_size())`.
///
/// Panics if `new_cap` is less than the vector's length.
pub fn grow(&mut self, new_cap: usize) {
    unsafe {
        let (ptr, &mut len, cap) = self.triple_mut();
        let unspilled = !self.spilled();
        assert!(new_cap >= len);
        if new_cap <= self.inline_size() {
            if unspilled {
                return;
            }
            self.data = SmallVecData::from_inline(mem::uninitialized());
            ptr::copy_nonoverlapping(ptr, self.data.inline_mut().ptr_mut(), len);
            self.capacity = len;
        } else if new_cap != cap {
            let mut vec = Vec::with_capacity(new_cap);
            let new_alloc = vec.as_mut_ptr();
            mem::forget(vec);
            ptr::copy_nonoverlapping(ptr, new_alloc, len);
            self.data = SmallVecData::from_heap(new_alloc, len);
            self.capacity = new_cap;
            if unspilled {
                return;
            }
        } else {
            return;
        }
        deallocate(ptr, cap);
    }
}
```

注意：在之后的几个版本又对 grow 函数做了一些优化，来去除 unsafe 或者使用///SAFETY 来解释 unsafe 的安全性，最新版本如下：
https://github.com/servo/rust-smallvec/blob/v2/src/lib.rs

RUSTSEC-2024-0359(典型的为了优化性能而引入的漏洞，目前程序扫描不出，因为指令数小于 5)

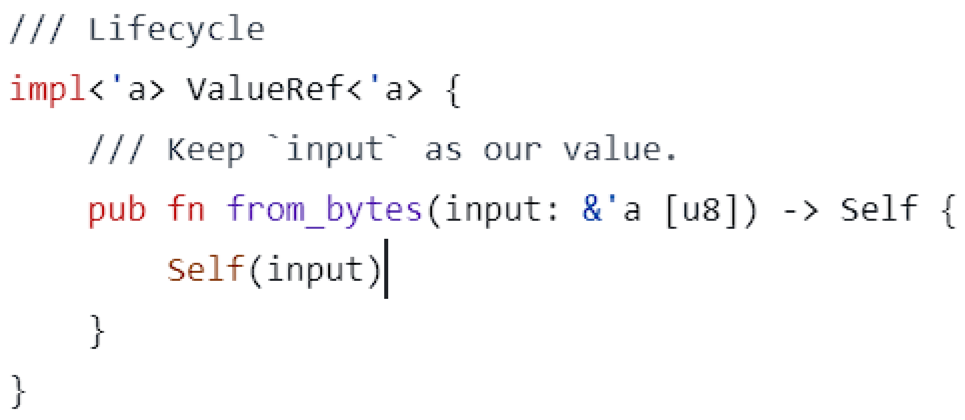
https://github.com/Byron/gitoxide/commit/7a98d8a518d771fe09614878d86d970ee1186b35

修复前:



```rust
/// Lifecycle
impl<'a> ValueRef<'a> {
    /// Keep `input` as our value.
    pub fn from_bytes(input: &'a [u8]) -> Self {
        Self(KStringRef::from_ref(
            // SAFETY: our API makes accessing that value as `str` impossible, so illformed UTF8 is never exposed as such.
            #[allow(unsafe_code)]
            unsafe {
                std::str::from_utf8_unchecked(input)
            },
        ))
    }
}
```

修复后:

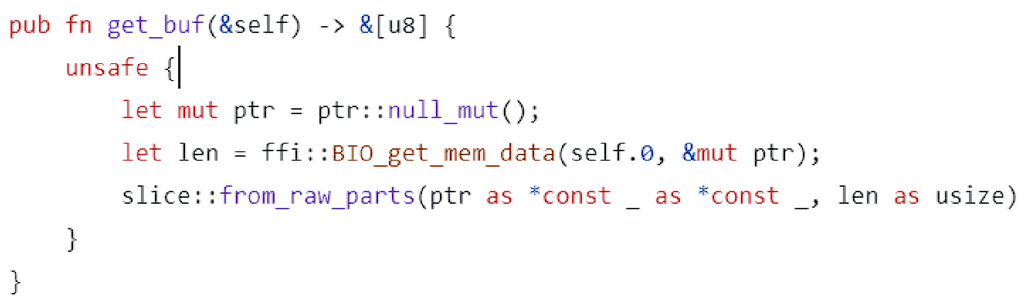


```rust
/// Lifecycle
impl<'a> ValueRef<'a> {
    /// Keep `input` as our value.
    pub fn from_bytes(input: &'a [u8]) -> Self {
        Self(input)
    }
}
```

RUSTSEC-2024-0357

https://github.com/sfackler/rust-openssl/pull/2266/commits/142deef717bad843fc04c5afb925bfd9e7dc4305

修复前:



```rust
pub fn get_buf(&self) -> &[u8] {
    unsafe {
        let mut ptr = ptr::null_mut();
        let len = ffi::BIO_get_mem_data(self.0, &mut ptr);
        slice::from_raw_parts(ptr as *const _ as *const _, len as usize)
    }
}
```

修复后:

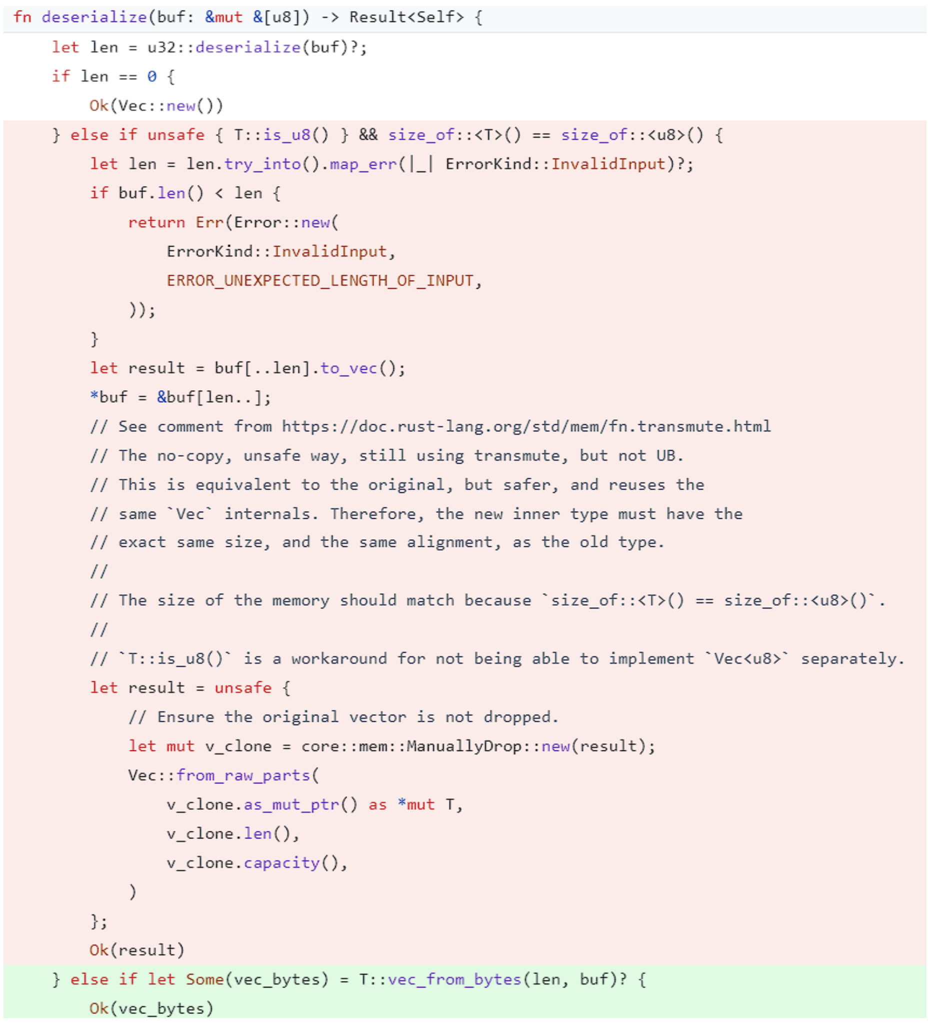
```rust
pub fn get_buf(&self) -> &[u8] {
    unsafe {
        let mut ptr = ptr::null_mut();
        let len = ffi::BIO_get_mem_data(self.0, &mut ptr);
        if len == 0 {
            &[]
        } else {
            slice::from_raw_parts(ptr as *const _ as *const _, len as usize)
        }
    }
}
```

这里牵扯到与第三方库交互，所以必须要使用 unsafe 代码块，但是可以缩减 unsafe 代码块的范围，只在调用 unsafe 函数的时候才使用 unsafe 代码块，而不是使用一整个 unsafe 代码块把整个函数都包含在内。

RUSTSEC-2023-0033

修复前后对比：

https://github.com/near/borsh-rs/pull/26/commits/c173b405f19f6de1c10a6145581db6fc2a0d33ba



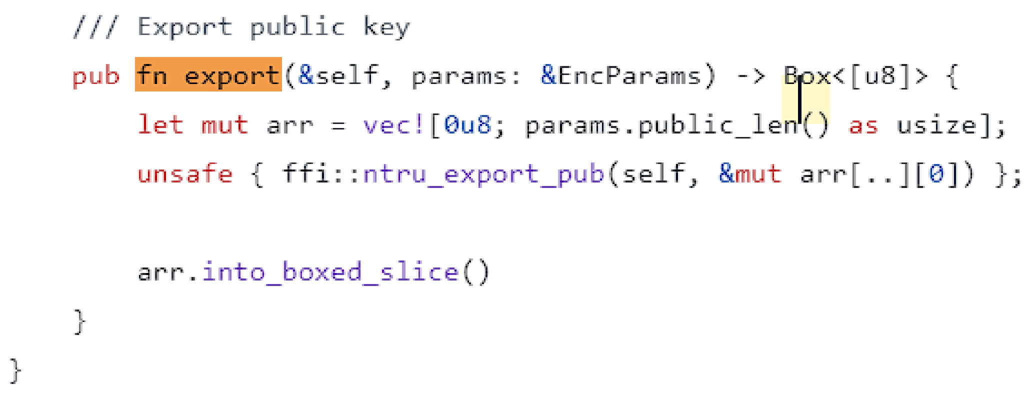
```rust
fn deserialize(buf: &mut &[u8]) -> Result<Self> {
    let len = u32::deserialize(buf)?;
    if len == 0 {
        Ok(Vec::new())
    } else if unsafe { T::is_u8() } && size_of::<T>() == size_of::<u8>() {
        let len = len.try_into().map_err(|_| ErrorKind::InvalidInput)?;
        if buf.len() < len {
            return Err(Error::new(
                ErrorKind::InvalidInput,
                ERROR_UNEXPECTED_LENGTH_OF_INPUT,
            ));
        }
        let result = buf[..len].to_vec();
        *buf = &buf[len..];
        // See comment from https://doc.rust-lang.org/std/mem/fn.transmute.html
        // The no-copy, unsafe way, still using transmute, but not UB.
        // This is equivalent to the original, but safer, and reuses the
        // same `Vec` internals. Therefore, the new inner type must have the
        // exact same size, and the same alignment, as the old type.
        //
        // The size of the memory should match because `size_of::<T>() == size_of::<u8>()`.
        //
        // `T::is_u8()` is a workaround for not being able to implement `Vec<u8>` separately.
        let result = unsafe {
            // Ensure the original vector is not dropped.
            let mut v_clone = core::mem::ManuallyDrop::new(result);
            Vec::from_raw_parts(
                v_clone.as_mut_ptr() as *mut T,
                v_clone.len(),
                v_clone.capacity(),
            )
        };
        Ok(result)
    } else if let Some(vec_bytes) = T::vec_from_bytes(len, buf)? {
        Ok(vec_bytes)
```

https://github.com/near/borsh-rs/pull/26

Ref：对我来说，这是一个危险信号，表明此包中的*所有*不安全代码都是不健全的。此 PR 重新实现了针对切片/数组/向量的相同优化行为，而无需任何不安全代码。

[＃25](#) 中的方法从文档中隐藏了特征方法但使其不健全（可以安全调用并且可以安全实现）对我来说似乎不是一个令人满意的解决方案。

RUSTSEC-2023-0032
修复前:



```
        /// Export public key
        pub fn export(&self, params: &EncParams) -> Box<[u8]> {
            let mut arr = vec![0u8; params.public_len() as usize];
            unsafe { ffi::ntru_export_pub(self, &mut arr[..][0]) };

            arr.into_boxed_slice()
        }
    }
```

https://github.com/FrinkGlobal/ntru-rs/blob/0.4.3/src/types.rs
典型的调用 FFI 导致的 rust 漏洞，这里没有写 SAFETY 注释，也没有对这个调用函数进行验证，当前的扫描器可以扫描出来。

RUSTSEC-2023-0017
https://github.com/tylerhawkes/maligned/pull/7/commits/c464eca8fabcb4d8bb293082b41e2c8d049a1bf0

```
pub fn align_first<T, A: Alignment>(t_capacity: usize) -> Vec<T> {
  if align_of::<A>() <= align_of::<T>() || t_capacity == 0 || size_of::<T>() == 0 {
    return Vec::<T>::with_capacity(t_capacity);
  }
  let min_bytes_to_allocate = size_of::<T>() * t_capacity;
  // Unwrap is safe because we fulfill all the invariants of `from_size_align`
  let layout = Layout::from_size_align(min_bytes_to_allocate, align_of::<A>()).unwrap();
  let ptr = unsafe { alloc::alloc::alloc(layout) };
  let type_vec = unsafe { Vec::<T>::from_raw_parts(ptr as *mut T, 0, t_capacity) };
  debug_assert_eq!(type_vec.as_ptr() as usize % align_of::<A>(), 0);
  debug_assert_eq!(type_vec.as_ptr() as usize, ptr as usize);
  type_vec
}
```

当前扫描器可以扫描出这个 bug，因为缺少 safety 注释，但是这个比较有趣，因为已经让 unsafe 代码块非常小了，依然可以导致 bug。

RUSTSEC-2022-0079
https://github.com/toku-sa-n/ramen/issues/1138

```
fn section_header_raw(&amp;self) -&gt; &amp;[ET::SectionHeader] {
    let sh_off = self.elf_header().section_header_offset() as usize;
    let sh_num = self.elf_header().section_header_entry_num() as usize;
    unsafe {
        let sh_ptr = self.content().as_ptr().add(sh_off);
        from_raw_parts(sh_ptr as *const ET::SectionHeader, sh_num)
    }
}
```

REF：如果 ELF 头是有效的，这将完全正常工作，但是恶意或格式错误的输入可能包含任意大小的节头偏移量，这意味着不安全块中的结果指针可以指向进程地址空间中的任意地址。
这可能导致不可预测的行为，在我们的模糊测试中，我们发现导致 SIGABRT(信号 6)或 SEGV(信号 11)是微不足道的。
函数要么被标记为不安全，并注明调用者只负责提供有效的输入，要么理想情况下，它应该尽职尽责

可以扫描出来，因为没有 safety 注释。

RUSTSEC-2022-0078
https://github.com/fitzgen/bumpalo/blob/3.11.0/src/collections/vec.rs
修复前：
```
#[inline]
fn into_iter(mut self) -> IntoIter<T> {
    unsafe {
        let begin = self.as_mut_ptr();
        // assume(!begin.is_null());
        let end = if mem::size_of::<T>() == 0 {
            arith_offset(begin as *const i8, self.len() as isize) as *const T
        } else {
            begin.add(self.len()) as *const T
        };
        mem::forget(self);
        IntoIter {
            phantom: PhantomData,
            ptr: begin,
            end,
        }
    }
}
```

修复后:
https://github.com/fitzgen/bumpalo/blob/3.11.1/src/collections/vec.rs

```rust
fn into_iter(mut self) -> IntoIter<'bump, T> {
    unsafe {
        let begin = self.as_mut_ptr();
        // assume(!begin.is_null());
        let end = if mem::size_of::<T>() == 0 {
            arith_offset(begin as *const i8, self.len() as isize) as *const T
        } else {
            begin.add(self.len()) as *const T
        };
        mem::forget(self);
        IntoIter {
            phantom: PhantomData,
            ptr: begin,
            end,
        }
    }
}
```

这个 bug 是因为生命周期的问题导致的 UAF 漏洞，但是当前扫描器也可以扫描出来。

--------------------------------8 月 16 日分割线--------------------------------

目前看到了 https://rustsec.org/categories/memory-corruption.html 的 RUSTSEC-2022-0012

--------------------------------8 月 17 日分割线--------------------------------