C Language

The C language is probably the most popular and successful language since it was born with the Unix operating system. It is a high-level language, but it can also do the low-level work such as modifying memory content or memory mapped I/O registers in a special address. It is the most suitable language in the firmware area because the firmware needs to access the hardware-related content such as memory or memory mapped I/O register directly, and the C language is good and powerful in this area. With the C language, the developer can fully control the machine's hardware. We believe it is because of the nature of the C language – C was born to write the Unix operating system.

Now let's take a look at the C language from the security perspective. According to the Microsoft research, memory safety contributes 70% of the security vulnerabilities in C and C++. We also observed similar data in that the buffer overflow and integer overflow caused 50% of security issues in the firmware area. The reason is that the firmware is also one type of software. Firmware also has the similar memory safety issue including the memory access error such as buffer overflow, use-after-free, and uninitialized data error such as wild pointer, null pointer reference, and so on.

In Chapter 14, we discussed the best practices and guidelines for C programming. In Chapter 15, we discussed the C compiler defensive technology. In Chapter 16, we discussed the firmware kernel enhancements. On the one hand, all of the industry has provided the guidance and tools to address those flaws. On the other hand, people are also looking for a better type-safe language that can help to prevent the developer from introducing the flaws in the first place.

Rust

Rust is a new language. If we treat the C language as portable assembly, then Rust is a safe C language. The Rust language is designed to empower the developer to build reliable and efficient software. It offers

1) Performance: Rust is blazingly fast and memory efficient. It does not have a runtime or garbage collector. It can power performance-critical services and run on embedded devices. Runtime performance and runtime memory consumption are extremely important to the firmware because a typical firmware runs in a resource-constrained environment. Even for the Intel

Architecture (IA) system firmware, it needs to run code in the system management mode (SMM) which only has limited system management RAM (SMRAM), such as 1 MB or 8 MB. Similar constraints are observed by embedded devices. Rust embedded offers flexible memory management, where the dynamic memory allocation is optional. You may choose to use a global allocator and dynamic data structures or statically allocate everything. This is required because the firmware code may run on the flash device without DRAM.

2) Reliability: Rust introduces a rich type system and an ownership model that guarantees memory safety and thread safety, which enables you to eliminate many classes of bugs at compile time. This is probably the most attractive feature. Rust is not the first language to introduce the type safety concept. One big concern of the type-safe language is the performance degradation because of the runtime type check. The performance impact is hard to accept for embedded system or firmware. The advantage of Rust is that many checks have been done at compile time. As such, the final generated binary does not include such checks. With strict rules of syntax, Rust can trace the lifecycle of a data object. As such, no runtime garbage collection is required. This design not only reduces the binary size but also improves the runtime performance.

3) Productivity: Rust has a friendly compiler with useful error messages. The compiler not only shows what is wrong but also gives suggestions on how to fix the error. It teaches you how to write the Rust language. Rust has provided a unit test framework. You can write a set of unit tests just after the function implementation. Rust also considers the interoperability with other languages with the foreign function interface (FFI). Rust can generate a C language–compatible application binary interface (ABI). You can let Rust call the C language or call Rust from the C language.

Rust Security Solution

Major classes of security issues in firmware development are the memory safety issue and the arithmetic issue. In Chapter 15, we discussed two compiler defensive strategies: to eliminate the vulnerability and to break the exploit. Rust does a great job on eliminating the vulnerability by introducing a strict rule at compile time. Any violation causes the build failure. Also Rust injects the runtime boundary check for the buffer overflow to break the exploit. The generated code calls a panic handler for the runtime violation. Table 20-1 shows the Rust solution to handle the safety issues. The two classic memory safety issues – the spatial safety issue and the temporal safety issue – are resolved separately. Let's take a look at them one by one.

Table 20-1. *Rust Security Solution*

Type	Subtype	Rust Solution
Access error (spatial)	Buffer overflow (Write)	Use Offset/Index for Slice Runtime Boundary Check – [panic_handler].
	Buffer over-read	Use Offset/Index for Slice Runtime Boundary Check – [panic_handler].
Access error (temporal)	Use-after-free(dangling pointer)	Ownership – compile-time check.
	Double free	Ownership – compile-time check.
Uninitialized data	Uninitialized variable	Initialization – compile-time check.
	Wild pointer	Initialization – compile-time check.
	NULL pointer deference	Use Option<T> enum Allocation Check – [alloc_error_handler].

(*continued*)

Table 20-1. (*continued*)

Type	Subtype	Rust Solution
Arithmetic issue	Integer overflow	DEBUG: Runtime check – [panic_handler]. RELEASE: Discard overflow data. Compiler flag: -C overflow-checks=on/off. Function: checked|overflowing|saturating|wrapping_ add|sub|mul|div|rem|sh||shr|pow().

	Type cast	Must be explicit – compile-time check.
		(Dest Size == Source Size) => no-op
		(Dest Size < Source Size) => truncate
		(Dest Size > Source Size) => {
		(source is unsigned) => zero-extend
		(source is signed) => sign-extend
		}

Ownership

The gist of Rust memory safety is to isolate the aliasing and mutation. Aliasing means there can be multiple ways to access the same data. The data is sharable. Mutation means the owner has the right to update the data. The data can be changed. The danger arises from aliasing + mutation.

Let's take a look at a C program in Listing 20-1. How many issues you can find?

Listing 20-1.

```
==========================
char *a1 = "hello world!";
char *b1 = "hello world!";

int main()
{
  char *a2 = strdup (a1);
  char *b2 = a2;
  char *b3 = strchr (a2, 'h');
  char *b4 = strchr (a2, 'w');
```

```
  *a1 = 'k';
  *a2 = 'l';

  *(a1 + 19) = 'm';
  *(a2 + 19) = 'n';

  printf("a1=%s (%p)\n", a1, a1);
  printf("a2=%s (%p)\n", a2, a2);
  printf("b1=%s (%p)\n", b1, b1);
  printf("b2=%s (%p)\n", b2, b2);
  printf("b3=%s (%p)\n", b3, b3);
  printf("b4=%s (%p)\n", b4, b4);

  free (a1);
  free (a2);
  free (b1);
  free (b2);
  free (b3);
  free (b4);
  printf("OK\n");
  return 0;
}
==========================
```

The memory layout in this program is shown in Figure 20-1. Both a1 and b1 are in the global data section. They point to a string “hello world!” in the read-only data section. With the optimization on, the a1 and b1 point to the same location. If we turn off the optimization, the a1 and b1 point to different locations. The a2, b2, b3, and b4 are on the stack. a2, b2, and b3 point to a “hello world!” string in the heap, and b4 points to the middle of the “hello world!” Although the program updates the string pointed by a1 and a2, the string pointed by b1, b2, and b3 is also updated. It might be a side effect. Updating a1 may also cause a runtime crash because the read-only section is marked protected unless the program merges the read-only data section into the normal data section at link phase. The string update also has a buffer overflow access for m and n. Last of all, the free is only required for the data in the heap and only required once. b2, b3, and b4 point to same string as a2. As such, only free(a2) is required. See Listing 20-2 for the comment of the program.

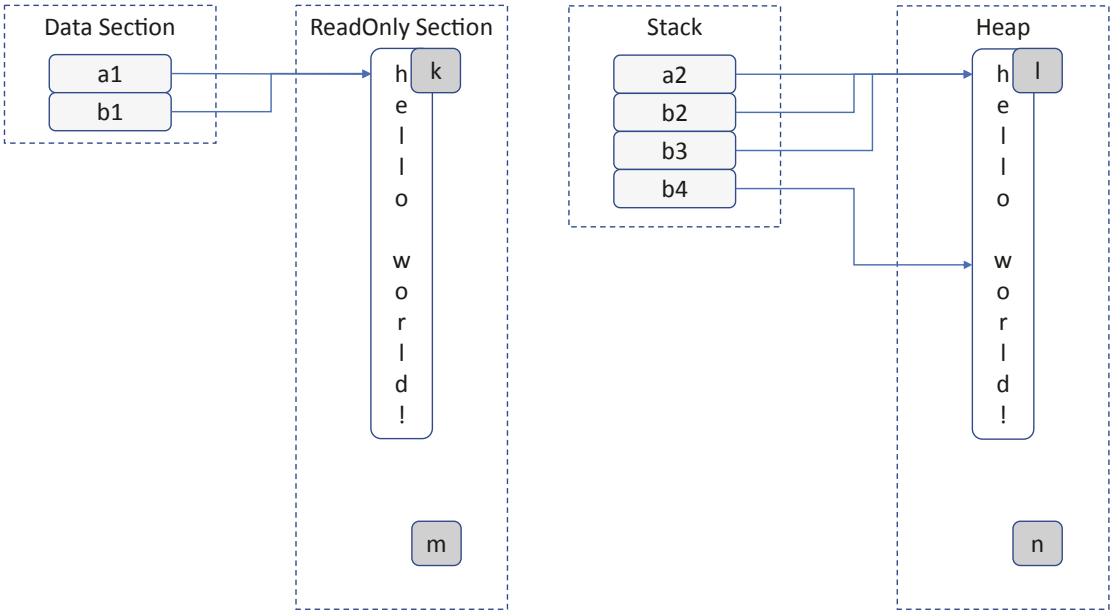

Figure 20-1. *Memory Layout of Listing 20-1*

Listing 20-2.

```
============================
char *a1 = "hello world!";
char *b1 = "hello world!";

int main()
{
  char *a2 = strdup (a1);
  char *b2 = a2;
  char *b3 = strchr (a2, 'h');
  char *b4 = strchr (a2, 'w');

  //*a1 = 'k'; // crash in normal build. Need merge .rdata to .data section.
              // b1 update if optimization on, side effect?
              // b1 not update if optimization off
  *a2 = 'l'; // cause b2, b3 update, side effect?

  //*(a1 + 19) = 'm'; // illegal, but no crash most likely
  //*(a2 + 19) = 'n'; // illegal, crash at free() most likely
```

```
  printf("a1=%s (%p)\n", a1, a1);
  printf("a2=%s (%p)\n", a2, a2);
  printf("b1=%s (%p)\n", b1, b1);
  printf("b2=%s (%p)\n", b2, b2);
  printf("b3=%s (%p)\n", b3, b3);
  printf("b4=%s (%p)\n", b4, b4);

  //free (a1); // illegal, crash
  free (a2); // legal, required otherwise memory leak
  //free (b1); // illegal, crash
  //free (b2); // maybe legal, only if a2 is not freed.
  //free (b3); // illegal, but works, if a2 is not freed
  //free (b4); // illegal, crash
  printf("OK\n");
  return 0;
}
============================
```

Listing 20-1 can be compiled successfully because the C compiler does not perform any such check. It relies on the developer to do the right thing. With Rust, you cannot write code in such a way. If the data is mutable, it cannot be shared. On the other hand, if the data is shared, it must be mutable. Rust has three basic patterns for programming: ownership, shared borrow, and mutable borrow. Let's look at them one by one.

First, Figure 20-2 shows the concept of ownership. In Listing 20-3, we initialize a string s1, assign s1 to s2, and then assign s1 to s3. It is legal in the C language, but illegal in Rust. The reason is that when s1 is assigned to s2, the ownership of the string "hello world!" is moved from s1 to s2. s1 is no longer valid. As such, when the code wants to assign s1 to s3, the compiler generates an error in Listing 20-4. What if we want to use both s2 and s1? We need to use borrow, also known as reference.

Listing 20-3.

```
============================
fn test1() {
    // ownership
    let s1 = String::from ("hello world!");
    let mut s2 = s1;
```

```
    let mut s3 = s1; // error because the ownership is moved to s2
    s2.make_ascii_lowercase();

    println!("s1={}", s1);
    println!("s2={}", s2);
    println!("s3={}", s3);
}
============================
```

Listing 20-4.

```
============================
error[E0382]: use of moved value: `s1`
 --> src\main.rs:6:14
  |
4 |     let s1 = String::from ("hello world!");
  |         -- move occurs because `s1` has type `std::string::String`,
                which does not implement the `Copy` trait
5 |     let mut s2 = s1;
  |                  -- value moved here
6 |     let mut s3 = s1; // error because the ownership is moved to s2
  |                  ^^ value used here after move
============================
```

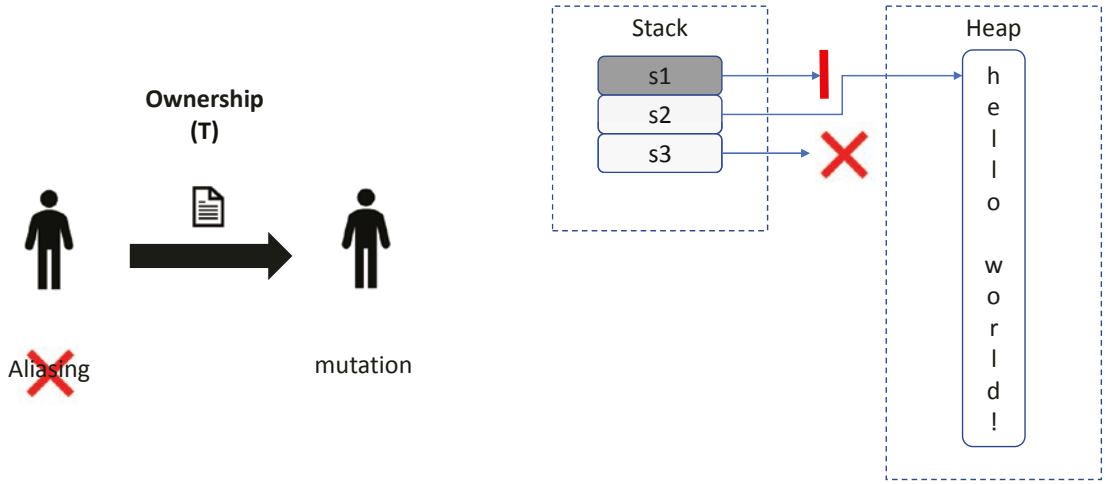

Figure 20-2. *Ownership in Rust*

Second, Figure 20-3 shows the concept of shared borrow. In Listing 20-5, we initialize s1 and assign s1 as a reference to s2 and s3 without any problem. However, if we want to update the string referenced by s2, the compiler generates error in Listing 20-6, because both s2 and s3 are immutable borrow. What if we want to update s2? We need to use mutable borrow.

Listing 20-5.

```
============================
fn test2() {
    // immutable borrow
    let s1 = String::from ("hello world!");
    let s2 = &s1;
    let s3 = &s1;
    s2.make_ascii_lowercase(); // error because s2 is immutable borrow.

    println!("s1={}", s1);
    println!("s2={}", s2);
    println!("s3={}", s3);
}
============================
```

Listing 20-6.

```
============================
error[E0596]: cannot borrow `*s2` as mutable, as it is behind a `&` reference
  --> src\main.rs:18:5
   |
17 |     let s2 = &s1;
   |              --- help: consider changing this to be a mutable
                       reference: `&mut s1`
17 |     let s3 = &s1;
19 |     s2.make_ascii_lowercase(); // error because s2 is immutable
                                    borrow.
   |     ^^ `s2` is a `&` reference, so the data it refers to cannot be
         borrowed as mutable
============================
```

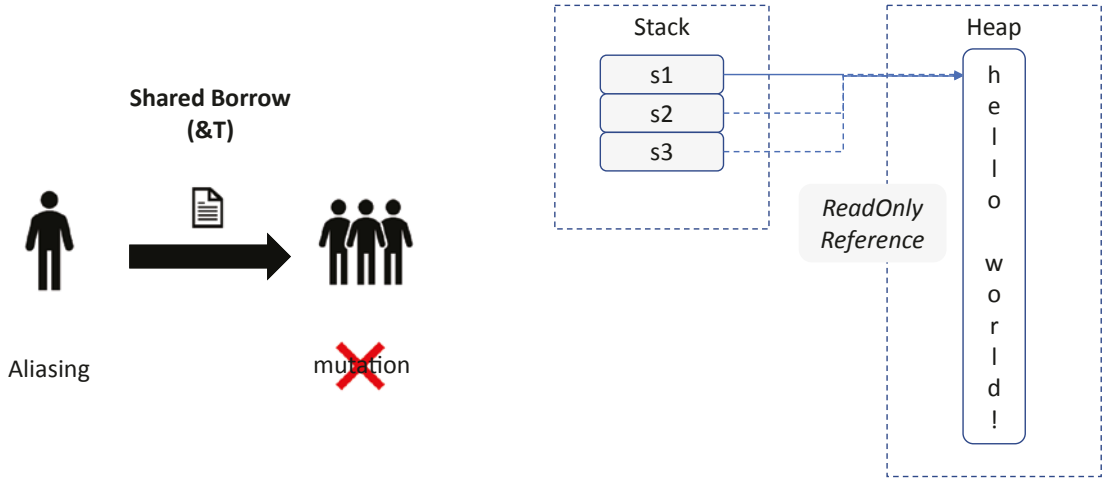

Figure 20-3. *Shared Borrow in Rust*

Third, Figure 20-4 shows the concept of mutable borrow. In Listing 20-7, we initialize s1 and assign s1 as a mutable reference to s2 and s3 because we want to update the s2 later. However, when we assign s1 as a mutable reference to s3, the compiler generates an error in Listing 20-8 because s3 is the second mutable borrow, which is illegal. At most, one mutable borrow is allowed in Rust.

Listing 20-7.

```
==========================
fn test3() {
    // mutable borrow
    let mut s1 = String::from ("hello world!");
    let s2 = &mut s1;
    let s3 = &mut s1; // error because this is second mutable borrow.
    s2.make_ascii_lowercase();

    println!("s1={}", s1);
    println!("s2={}", s2);
    println!("s3={}", s3);
}
==========================
```

Listing 20-8.

```
============================
error[E0499]: cannot borrow `s1` as mutable more than once at a time
  --> src\main.rs:29:14
   |
29 |     let s2 = &mut s1;
   |              ------- first mutable borrow occurs here
30 |     let s3 = &mut s1; // error because this is second mutable borrow.
   |              ^^^^^^^ second mutable borrow occurs here
31 |     s2.make_ascii_lowercase();
   |     -- first borrow later used here
============================
```

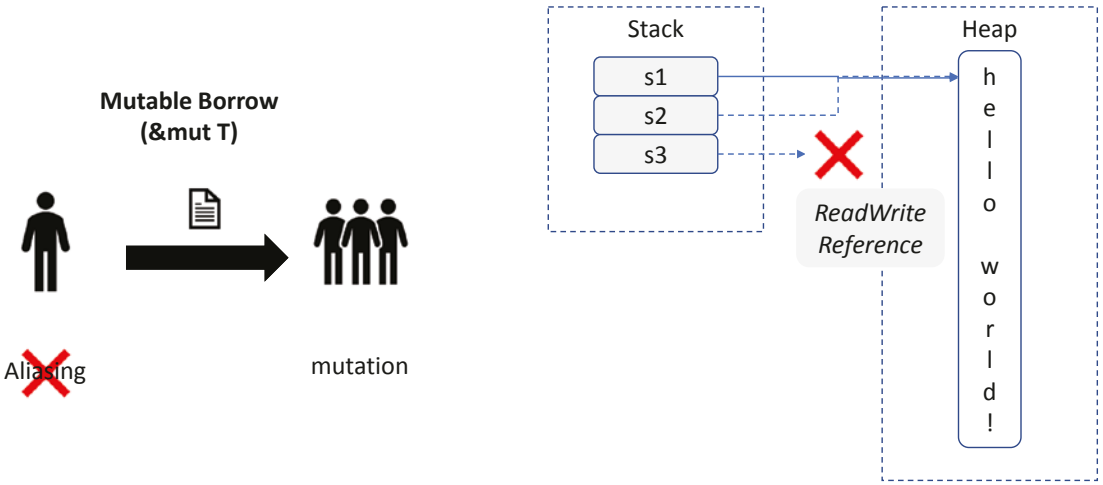

Figure 20-4. *Mutable Borrow in Rust*

You may also find that no free() is called in those functions because free() is not required in the code. free() is injected by the compiler. Listing 20-9 shows the compiler-generated binary for the entry and exit of the test1() function in Listing 20-3. With the strict rule the developer must follow, the Rust compiler can manage the object's lifecycle at build time. As such, the compiler injects __rust_alloc and __rust_dealloc in the code instead of using runtime garbage collection. Many optimizations can also be achieved at build time, and this is one reason that Rust can run fast.

Listing 20-9.

```
============================
  0000000140001140: 55                push        rbp
  0000000140001141: 56                push        rsi
  0000000140001142: 48 81 EC A8 00 00 sub         rsp,0A8h
                    00
  0000000140001149: 48 8D AC 24 80 00 lea         rbp,[rsp+80h]
                    00 00
  0000000140001151: 48 C7 45 20 FE FF mov         qword ptr [rbp+20h],
                                                  0FFFFFFFFFFFFFFFEh
                    FF FF
  0000000140001159: BE 0C 00 00 00    mov         esi,0Ch
  000000014000115E: B9 0C 00 00 00    mov         ecx,0Ch
  0000000140001163: BA 01 00 00 00    mov         edx,1
  0000000140001168: E8 93 08 00 00    call        __rust_alloc
  000000014000116D: 48 85 C0          test        rax,rax
  0000000140001170: 0F 84 D9 00 00 00 je          000000014000124F
  0000000140001176: 48 89 45 F0       mov         qword ptr [rbp-10h],rax
...
  00000001400011C4: 0F 10 45 F0       movups      xmm0,xmmword ptr [rbp-10h]
  00000001400011C8: 0F 29 45 A0       movaps      xmmword ptr [rbp-60h],xmm0
  00000001400011CC: 0F 28 45 A0       movaps      xmm0,xmmword ptr [rbp-60h]
  00000001400011D0: 0F 29 45 D0       movaps      xmmword ptr [rbp-30h],xmm0
...
  0000000140001235: 48 8B 4D D0       mov         rcx,qword ptr [rbp-30h]
  0000000140001239: 41 B8 01 00 00 00 mov         r8d,1
  000000014000123F: E8 CC 07 00 00    call        __rust_dealloc
  0000000140001244: 90                nop
  0000000140001245: 48 81 C4 A8 00 00 add         rsp,0A8h
                    00
  000000014000124C: 5E                pop         rsi
  000000014000124D: 5D                pop         rbp
  000000014000124E: C3                ret
============================
```

Finally, let's see one more example to show how Rust catches an issue in Listing 20-10. The purpose of str_find_char() is to locate the first occurrence of a character in a string, similar to strchr() in the C language. The s.clear() is followed by the means to truncate this string. Because the result of str_find_char() is a reference to the original string s, this truncation impacts the result. The next println cannot print the expected content. Fortunately, the Rust compiler does a great job to catch this issue and show the error message in Listing 20-11. It notices that there is an immutable reference for s, so the s cannot have a mutable borrow to truncate the content.

Listing 20-10.

```
==========================
fn str_find_char(s: &String, c: char) -> Option<&str> {
    let bytes = s.as_bytes();
    let len = s.len();

    for (i, &item) in bytes.iter().enumerate() {
        if item == c as u8 {
            return Some(&s[i..len]);
        }
    }

    None
}

fn test4() {
    let mut s = String::from("hello world");

    let result = str_find_char(&s, 'w');

    s.clear(); // error!

    match result  {
      Some(word) => println!("found: {}", word),
      None => println!("not found"),
    }
}
==========================
```

Listing 20-11.

```
============================
error[E0502]: cannot borrow `s` as mutable because it is also borrowed as
immutable
  --> src\main.rs:71:5
   |
69 |     let result = str_find_char(&s, 'w');
   |                                -- immutable borrow occurs here
70 |
71 |     s.clear(); // error!
   |     ^^^^^^^^^ mutable borrow occurs here
72 |
73 |     match result  {
   |           ------ immutable borrow later used here
============================
```

Option<T> Type

In Listing 20-10, there is a data type – Option<&str>. What is that for?

The C language has NULL pointer concept. Dereference of a NULL pointer causes an exception at runtime. Tony Hoare treated the NULL pointer as a "billion-dollar mistake." It was invented in 1965 just because it was easy to implement. Eventually it has led to innumerable errors, vulnerabilities, and system crashes.

Rust uses Option<T> type to resolve the NULL pointer. The definition for Option<T> type is in Listing 20-12.

Listing 20-12.

```
============================
pub enum Option<T> {
    /// No value
    None,
    /// Some value T
    Some(T)
}
============================
```

Listing 20-13 is the C program that might have a problem. If a developer forgets to check the NULL pointer in the check_optional() function, a NULL reference might happen in the C version. This will never happen in Rust. Listing 20-14 shows the Rust version. If a data might be NULL, the Option<T> must be used. In order to get data, the program must use Some(T) to get the data from the Option<T> parameter and include None pattern to handle the no-value case. The risk of NULL deference is eliminated.

Listing 20-13.

```
============================
void print_ptr_data (int *optional) {
    if (optional == NULL) { // It might be missing.
        printf ("NULL pointer\n");
    } else {
        printf ("value is %d\n", *optional);
    }
}

void test() {
    int *optional = NULL;
    print_ptr_data (optional);

    optional = malloc (sizeof(int));
    *optional = 5;
    print_ptr_data (optional);
}
============================
```

Listing 20-14.

```
============================
fn print_ptr_data (optional: Option<Box<i32>>) {
    match optional {
        Some(p) => println!("value is {}", p),
        None => println!("Value is None"),
    }
}
```

```
fn test5() {
    let optional = None;
    print_ptr_data (optional);

    let optional = Some(Box::new(5));
    print_ptr_data (optional);
}
==========================
```

Boundary Check

In the C language, a buffer overflow is one of the most critical issues. In general, Rust recommends to use the iterator for the buffer access. For example, Listing 20-10 shows that the str_find_char() function uses bytes.iter().enumerate(). However, there are still chances that the developer needs to write the index of the buffer array and may make a mistake. For example, in Listing 20-15, the str_find_char() returns Some(&s[i..len+1]), while it should be Some(&s[i..len]). This skips the build-time check, but it can be caught by the runtime check in Listing 20-16.

Listing 20-15.

```
==========================
fn str_find_char(s: &String, c: char) -> Option<&str> {
    let bytes = s.as_bytes();
    let len = s.len();

    for (i, &item) in bytes.iter().enumerate() {
        if item == c as u8 {
            return Some(&s[i..len+1]); // bug
        }
    }

    None
}
==========================
```

Listing 20-16.

```
============================
thread 'main' panicked at 'byte index 12 is out of bounds of `hello
world`', src\libcore\str\mod.rs:2017:9
note: Run with `RUST_BACKTRACE=1` environment variable to display a
backtrace.
============================
```

The Rust compiler inserts code to do boundary check for the buffer access at runtime. If there is a violation, the inserted checker will call a predefined function panic_handler(). The default panic_handler() is provided in the standard library. The firmware usually only links the Rust corelib and needs to define its own panic_handler(), such as Listing 20-17.

Listing 20-17.

```
============================
#[panic_handler]
fn panic_handler(_info: &core::panic::PanicInfo) -> ! {
    // Add your own debug information
    loop {}
}
============================
```

Uninitialized Data Check

Rust eliminates the uninitialized data at build time. The compiler does the static analysis to ensure the data used must be initialized in any path, including conditional assignment such as if/else statement. Similar to the C language, there might be false positive, but it is better to eliminate any risk at build time.

Arithmetic Check

In the C language, integer overflow is another big problem. In many cases, it causes a buffer overflow later. Rust makes some improvements on math operations. Take Listing 20-18 as an example. It shows five different ways to get the result from multiplication of two u32 integers.

Listing 20-18.

```
==========================
fn test6(a: u32, b:u32) {

    let c : Option<u32> = a.checked_mul(b);
    match c {
      Some(v) => println!("checked multiple: {}", v),
      None => println!("checked multiple: overflow"),
    }

    let (c, o) : (u32, bool) = a.overflowing_mul(b);
    println!("overflowing multiple: {}, overflow: {}", c, o);

    let c : u32 = a.saturating_mul(b);
    println!("saturating multiple: {}", c);

    let c : u32 = a.wrapping_mul(b);
    println!("wrapping multiple: {}", c);

    let c : u32 = a * b;
    println!("direct multiple: {}", c);
}
fn main() {
    test6(0xFFFFFFFF, 0xFFFFFFFF);
}
==========================
```

Rust provides a set of methods for the primitive:

1) checked_mul(): The result is Option<T>. If no overflow happens, the result is Some<T>. Otherwise, the result is None.
2) overflowing_mul(): The result is a tuple (T, bool). The first element is a wrapped result. The second element is a bool to show if overflow happens or not.
3) saturating_mul(): The result is T type. It is a saturated value.
4) wrapping_mul(): The result is T type. It is a wrapped value.

For the multiplication operator, the Rust compiler treats it as wrapping_mul() in the release build, but injects runtime check code in the debug build. If the violation occurs at runtime, the checker invokes the panic_handler in the debug build. The developer can also use the compiler flag "-C overflow-checks=on/off" to control the runtime overflow check on or off. See Table 20-2.

Table 20-2. *Rust Math Operation – a: u32 * b: u32*

Method	Overflow Result
c : Option<u32> = a.**checked_mul**(b)	c is None.
(c, o) : (u32, bool) = a.**overflowing_mul**(b)	c holds the wrapped value. o indicates if overflow happens.
c : u32 = a.**saturating_mul**(b)	c holds the maximum u32.
c : u32 = a.**wrapping_mul**(b)	c holds the wrapped value.
c = a * b	c holds the wrapped value in release build. Runtime overflow check fails in debug build.

With release build, the result of Listing 20-18 is shown in Listing 20-19. With debug build, the result is shown in Listing 20-20.

Listing 20-19.

```
============================
checked multiple: overflow
overflowing multiple: 1, overflow: true
saturating multiple: 4294967295
wrapping multiple: 1
direct multiple: 1
============================
```

Listing 20-20.

```
============================
checked multiple: overflow
overflowing multiple: 1, overflow: true
saturating multiple: 4294967295
wrapping multiple: 1
```

```
thread 'main' panicked at 'attempt to multiply with overflow', src\main.
rs:90:19
note: Run with `RUST_BACKTRACE=1` environment variable to display a
backtrace.
============================
```

A firmware implementation may have external input, such as a Bitmap (BMP) file, a capsule file, a portable and executable (PE) image, a file system, and so on. If the parser needs to perform some math operation, using checked version methods might be a better idea than direct multiplication. It can guarantee the overflow case is well handled.

Besides math operations, type cast might also lead to the data truncation problem. Rust does the build-time check and requests a data type cast explicitly.

All in all, Rust defines a set of very strict rules on using an object and its reference to eliminate memory safety issues. That is one reason that many developers find it is hard to pass compilation, especially for those who are familiar with the C language and satisfied with the freedom brought from the C language. With the C language, the developer can control everything. That also means the developer needs to ensure the code has no memory safety issue. Rust takes another approach. It makes it very hard to write the code to pass the compilation on the first time. The compiler keeps telling you what is forbidden and that there might be a potential problem. But once the code passes the compilation, the Rust language guarantees there is no memory safety issue.

Unsafe Code

So far, we have discussed lots of security solutions brought from Rust, and these solutions can help reduce the amount of security risks in the firmware development. However, this solution brings some limitations. For example, accessing NULL address actually is legal because the tradition Basic Input/Output System (BIOS) sets up the Interrupt Vector Table there. Sometimes, the firmware code needs to access a fixed region memory mapped I/O, such as Trusted Platform Module (TPM) at physical address 0xFED40000. In order to handle such cases, Rust introduces a keyword – unsafe. If the code is inside of an unsafe block, then the compiler does not perform any security check. This is a contract between the developer and the compiler. Unsafe means that the developer tells the Rust compiler “Please trust me.”

According to the Rust language book, we can do the following superpower unsafe actions:

- Dereference a raw pointer
- Call an unsafe function or method.
- Access or modify a mutable global static variable.
- Implement an unsafe trait.
- Access fields of union.

In Listing 20-10 and Listing 20-14, we demonstrated a string and a Box<T> type. These are Rust-defined types. They can be used to point at a data in the heap. However, these types are not available in the C language. In order to interoperate with the C language, Rust has to define a raw pointer to be compatible with the pointer defined in C. Listing 20-21 shows the raw point usage. *mut u32 means a u32 pointer and the content is mutable. Because this code wants to dereference a raw pointer, it must be included in the unsafe block. The Rust language does not provide any guarantee on the memory safety here. At runtime, it might work because the developer may want to write the IVT. Or it might crash because of a mistake.

Listing 20-21.

```
============================
fn test7() {
  unsafe {
    let p = 0 as *mut u32;
    *p = 4;
  }
}
============================
```

Calling an external function, such as a C function, is also considered as unsafe because Rust loses control for the external function. Listing 20-22 shows how Rust calls the C abs() function.

Listing 20-22.

```
============================
extern "C" {
    fn abs(a: i32) -> i32;
}

fn test8(a: i32) -> i32 {
    unsafe {
        abs (a)
    }
}
============================
```

Global static variables are considered as sharable by many functions. According to the aliasing/mutation rule, it should be immutable. However, there might be a case that we do need to modify the global static variable, such as a global count. Usually a mutable static variable is used for the interoperation with the C language. It is dangerous that Rust requires to use unsafe keyword. Listing 20-23 shows an example.

Listing 20-23.

```
============================
static mut COUNTER: u32 = 0;
fn increment_counter() {
    unsafe {
        COUNTER += 1;
    }
}
fn test9() {
    increment_counter();
    unsafe {
        println!("COUNTER: {}", COUNTER);
    }
}
============================
```

Rust supports union type, with the same reason to interoperate with the C language. Union is dangerous because the type of data is undetermined. It violates type safety and may cause problems, such as partial initialization. See Listing 20-24. It outputs garbage at runtime.

Listing 20-24.

```
============================
#[repr(C)]
union U {
  d8 : u8,
  d32: u32,
}
fn test10() {
  let dataU = U { d8 : 1 };
  unsafe {
    let data32 = dataU.d32;
    println!("get data {}", data32);
  }
}
============================
```

Current Project

Rust is still a young language. It was designed by Mozilla Graydon Hoare, and the first release is at 2014. Because of its security properties, more and more projects are adopting Rust. For example, c2rust can help you convert the C language to the unsafe Rust language. Then people can do refactoring for the unsafe version and turn it into a safe version. Mozilla uses rust for Firefox. Amazon Firecracker is a Rust-based hypervisor. Baidu released Rust-SGX-SDK for the secure enclave and Rust OP-TEE TrustZone SDK as part of MesaTEE. Facebook uses Rust to develop Libra – a decentralized financial infrastructure. Google Fuchsia uses Rust in some components. The OpenTitan hardware root-of-trust uses the Tock OS, which is written in Rust. OpenSK – a Fast Identity Online 2 (FIDO2) authenticator is written in Rust as a Tock OS application. In 2019, Microsoft announced that they would adopt Rust as a systems programming language. In 2020, Microsoft has introduced open source project Verona, a new research language for safe infrastructure programming, which is inspired by Rust.

Firmware projects are including Rust. A rust hypervisor firmware can boot the cloud hypervisor. oreboot is the downstream fork of coreboot – a coreboot without C. EDK II is also adding support to build Rust module in the full UEFI firmware.

Cryptographic algorithms are also being developed in Rust, such as RustCrypto, MesaLink, rusttls, ring, webpki, and so on. The rusttls project includes a security review and audit report by Cure53, which shows the high quality of the code. Hope they can be the replacement for openSSL or mbed TLS in the future. Currently, the ring/webpki depend upon the OS provided random number generation. In order to use the ring/webpki in the firmware, we need a firmware based random number library. For example, the efi-random crate uses the RDRAND and RDSEED instruction in the UEFI/EDK II environment.

Limitation

Rust brings great enhancements to eliminate memory safety issues in the firmware, but there are still some non–language-specific firmware security issues that need to be taken care of, such as

1) Silicon register lock: We need to use a vulnerability scan tool, such as CHIPSEC.
2) Security policy: We need to perform a policy check to ensure the firmware is configured correctly.
3) Time-of-check/time-of-use (TOC/TOU) issue: We need to carefully perform an architecture review and design review.
4) X86 system management mode (SMM) callout: We need hardware restrictions, such as the SMM_CODE_CHECK feature.

Last but not of least importance, the unsafe code in Rust is always a risk. The Baidu Rust-SGX-SDK project has summarized the rules of Rust unsafe code:

- Unsafe components should be appropriately isolated and modularized, and the size should be small (or minimized).
- Unsafe components should not weaken the safety, especially of public APIs and data structures.
- Unsafe components should be clearly identified and easily upgraded

Writing unsafe code in Rust is same as writing C code. There is no safety guarantee from Rust. Please isolate them, minimize them, and review them carefully.

Others

Other languages are also used in firmware projects. For example, Forth is a language defined in IEEE 1275-1994. It is used for open firmware projects. Some embedded devices use Java Embedded, MicroPython, or the .NET Compact Framework. Those languages require a runtime interpreter. They might be a good candidate as an application language, but they are hard to use as a system language for the firmware development. We also observe that other type safety languages such as Ada or OCaml are used for embedded system, such as the Mirage OS, but they are not widely used.

Summary

In this chapter, we introduced the languages used in firmware development. Because we already introduced a lot on C language in previous chapters, the focus in this chapter is to introduce a new promising language – Rust – including the benefit brought from Rust and its limitation in the firmware security area. This is the last chapter of Part III. In Part IV, we will introduce security testing.

References

Book

[B-1] Peter van der Linden, *Expert C Programming: Deep C Secrets*, Prentice Hall, 1994

[B-2] Jim Blandy, Jason Orendorff, *Programming Rust: Fast, Safe Systems Development*, O'Reilly Media, 2017

[B-3] Brian L. Troutwine, *Hands-On Concurrency with Rust: Confidently build memory-safe, parallel, and efficient software in Rust*, Packt Publishing, 2018

[B-4] Steve Klabnik , Carol Nichols, *The Rust Programming Language* (Covers Rust 2018), No Starch Press, 2019, https://doc.rust-lang.org/book/

Conference, Journal, and Paper

[P-1] Nicholas Matsakis, "Rust," Mozilla Research, http://design.inf.unisi.ch/sites/default/files/seminar-niko-matsiakis-rustoverview.pdf

[P-2] Tony Hoare, "Null References: The Billion Dollar Mistake," in QCon 2009, available at www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare

[P-3] Yu Ding, Ran Duan, Long Li, Yueqiang Cheng, Yulong Zhang, Tanghui Chen, Tao Wei, Huibo Wang, "POSTER: Rust SGX SDK: Towards Memory Safety in Intel SGX Enclave," Computer and Communications Security 2017, https://github.com/apache/incubator-teaclave-sgx-sdk/blob/master/documents/ccsp17.pdf

[P-4] Florian Gilcher, Robin Randhawa, "The Rust Programming Language: Origin, History and Why it's so important for Arm today," in Linaro Connect 2020, www.youtube.com/watch?v=yE55ZpQmw9Q

[P-5] Bryan Cantrill, "The Soul of a New Machine: Rethinking the Computer," Stanford Seminar 2020, www.youtube.com/watch?v=vvZA9n3e5pc&list=PLoROMvodv4rMWw6rRoeSpkiseTHzWj6vu&index=4&t=0s

[P-6] Ryan O'Leary, "oreboot," in Open Source Firmware Conference 2019, available at https://www.youtube.com/watch?v=xJ6zI8MmcUQ

[P-7] Jiewen Yao, Vincent Zimmer, "Enabling Rust for UEFI Firmware," in UEFI webinar 2020, https://www.brighttalk.com/webcast/18206/428896, https://uefi.org/sites/default/files/resources/Enabling%20RUST%20for%20UEFI%20Firmware_8.19.2020.pdf

Specification and Guideline

[S-1] IEEE, "IEEE Standard for Boot (Initialization Configuration) Firmware: Core Requirements and Practices," IEEE 1275-1994, available at www.openbios.org/data/docs/of1275.pdf

Web

[W-1] Rust, www.rust-lang.org/

[W-2] Rust Unsafe code guideline, https://github.com/rust-lang/unsafe-code-guidelines

[W-3] c2rust, https://c2rust.com/

[W-4x] Amazon Firecracker, https://github.com/firecracker-microvm/firecracker

[W-5] Rust SGX SDK, https://github.com/apache/incubator-teaclave-sgx-sdk

[W-6] Rust OP-TEE TrustZone SDK, https://github.com/mesalock-linux/rust-optee-trustzone-sdk

[W-7] Facebook Libra, https://github.com/libra/libra

[W-8] Google fuchsia, https://fuchsia.googlesource.com/fuchsia/

[W-9] Tock OS, https://github.com/tock/tock

[W-10] OpenTitan, https://github.com/lowRISC/opentitan

[W-11] OpenSK, https://github.com/google/OpenSK

[W-12] oreboot, https://github.com/oreboot/oreboot

[W-13] EDKII rust support, https://github.com/tianocore/edk2-staging/tree/edkii-rust, https://github.com/jyao1/edk2/tree/edkii-rust/RustPkg, https://github.com/jyao1/ring/tree/uefi_support, https://github.com/jyao1/webpki/tree/uefi_support, https://github.com/jyao1/edk2/tree/edkii-rust/RustPkg/External/efi-random

[W-14] Rust hypervisor firmware, https://github.com/cloud-hypervisor/rust-hypervisor-firmware

[W-15] r-efi, UEFI Reference Specification, https://github.com/r-util/r-efi

[W-16] Rust wrapper for UEFI, https://github.com/rust-osdev/uefi-rs

[W-17] Rust Crypto, https://github.com/RustCrypto

[W-18] MesaLink, https://github.com/mesalock-linux/mesalink

[W-19] RustTls, https://github.com/ctz/rustls/

[W-20] ring – cryptographic operations, https://github.com/briansmith/ring

[W-21] webpki, https://github.com/briansmith/webpki

[W-22] Microsoft – A proactive approach to more secure code, https://msrc-blog.microsoft.com/2019/07/16/a-proactive-approach-to-more-secure-code/

[W-22] Microsoft – why rust for safe systems programming, https://msrc-blog.microsoft.com/2019/07/22/why-rust-for-safe-systems-programming/

[W-23] Open Firmware, www.openfirmware.info/Welcome_to_OpenBIOS

[W-24] mirage OS, https://mirage.io/

[W-25] Cure53, "Security Review and Audit Report RUST TLS," https://github.com/ctz/rustls/blob/master/audit/TLS-01-report.pdf

[W-26] Project Verona, www.microsoft.com/en-us/research/project/project-verona/, https://github.com/microsoft/verona