Crab Writeup

A challenge for the TexSAW 2023 CTF

1. Initial Investigation

We first look for the `main` symbol. This is shown in Figure 1. Rust binaries are different than gcc-compiled binaries in that they contain “main wrappers”. These are wrappers around the actual main function within the rust source code. The typical arguments to `main` are passed alongside additional arguments that determine the time of runtime to use, which is the primary purpose of the “main wrapper”. This isn’t usually a concern unless you are analyzing embedded binaries.

The address of the inner main function, `crab::main`, is assigned to `var_8`, as seen in Figure 1. Let’s take a look at that next. But first, you might be wondering why most symbols have a bunch of weird strings after them. This is called symbol mangling. This is done by every modern compiler to give each function a unique signature in the binary. As for why, the reason can be better explained here, and here with more details.

```
00009820  int32_t main(int32_t argc, char** argv, char** envp)

00009820      int64_t (* rax)()
00009820      int64_t (* var_8)() = rax
0000982e      var_8 = crab::main::h8b59de86bf914414
00009846      return std::rt::lang_start_internal::h8f7e70b1a2558118(&var_8, &anon.b5d6c6c0c4032bc323c...130637891.0.llvm
```

Figure 1. Main symbol

2. Looking at Main

2.1 Command Line Arguments

When looking at `crab::main`, as seen in Figure 2, the first thing you might say is: “where did the arguments to main go”? Or “doesn’t `main` usually come with arguments like `argc`, `argv`, and `envp`”? And I’d say: “You’re right!” Instead of having the typical command line arguments passed as arguments to inner main, Rust accesses them using the `sys::env::args()` function.

There is a tremendous amount of stack manipulation going on in the disassembly. The high-level view provided by BinaryNinja and other decompilers are unable to properly show this. I will attempt to give an overview of what is happening to the command line arguments.

```
00009520  int64_t crab::main::h8b59de86bf914414()

00009520  {
00009538      int128_t var_68;
00009538      std::env::args::h3f8a8f798d8e3971(&var_68);
00009548      int128_t var_58;
00009548      int128_t var_98 = var_58;
0000954d      int128_t var_a8 = var_68;
00009559      int64_t var_40;
00009559      _$LT$alloc..vec..Vec$LT$...GT$$GT$::from_iter::hb6e297eb7d01abb4(&var_40, &var_a8);
00009564      int64_t var_30;
00009564      if (var_30 != 2)
0000955e      {
00009719          std::panicking::begin_panic::h1d80039d87d54b55("Invalid arguments!Heres the key:…"  );
00009719          /* no return */
00009719      }
00009574      int64_t* var_38;
00009574      void* const rax_4;
00009574      int64_t rcx_1;
00009574      if (var_38[5] != 8)
0000956f      {
00009723          rax_4 = &data_41063;
0000972a          rcx_1 = 0xf;
0000972a      }
```

Figure 2. crab::main prologue

First, the address of a 128-bit value is passed to sys::env::args(). The function sets that 128-bit value, which it treats as 2 64-bit values, as seen in Figure 3. The first of which is the number of command line arguments. The second appears to be a heap address of an iterator. Taking a deeper look at this iterator structure, we can see that it contains the address and length of the command-line argument that I passed to it: “22221111” and 8, respectively.

This 128-bit value returned from sys::env::args() and its contents are shifted around on the stack and passed to vec::from_iter() in the form of var_40 and var_a8. This turns the iterator structure into a Vec structure. More stack manipulation occurs behind the high-level scene. In the end, var_30 is the length of the vector and var_38[5] is the length of the first element.

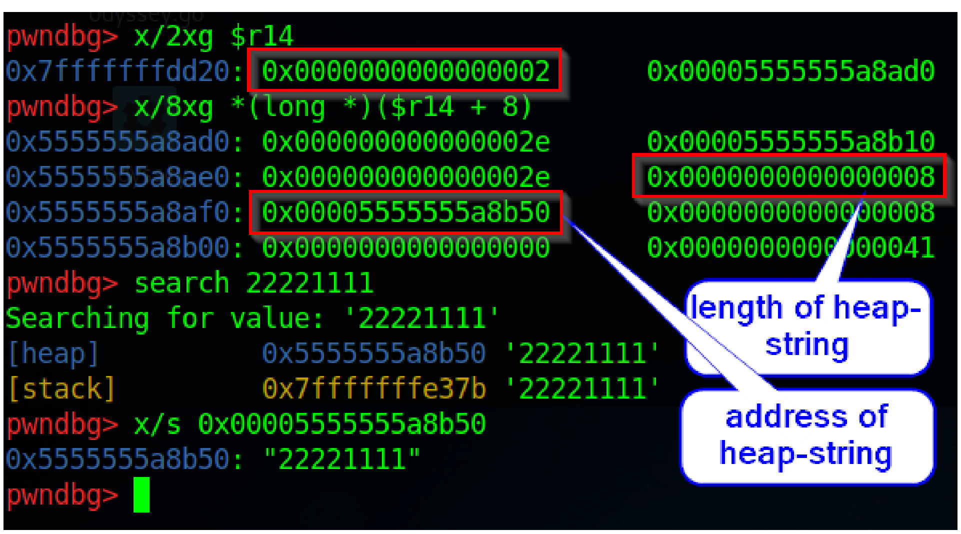

Figure 3. Iterator structure of command line arguments

It seems that 1 argument (excluding the default name of the binary as the 1st argument) must be passed, otherwise, the program panics. Also, when the length of the argument passed is not 8, the string "Invalid Length" is set to `rax_4` and the program panics with it, as seen in Figure 2. In conclusion, an argument must be passed with a length of 8.

2.2 Figuring Out the Key

After we have input a specific argument, it is passed to `Iterator::fold()` as `var_a8`. It'll be easier and more convenient to understand what is being passed to `Iterator::fold()` if we look at it using a debugger, as seen in Figure 5.

```
0000972a    }
0000957a    else
0000957a    {
0000957a        int64_t rax = var_38[4];
00009582        var_a8 = (rax + 8);
00009586        *(int64_t*)((char*)var_a8)[8] = rax;
0000958b        var_98 = 1;
00009594        *(int8_t*)((char*)var_98)[8] = 1;
000095a6        if (_$LT$core..iter..adapter...Iterator$GT$::fold::h098f16dcfca1512b(&var_a8, 0) == 6)
000095a3        {
000095c2            base64::engine::Engine::decode::h2f8378c143b0a0aa(&var_a8, &data_4107e, "dGV4c2F3e3J1c3R5X2NyYWJ9sr
000095cc            if (var_a8 != 0)
000095c7            {
00009755                var_68 = var_a8;
00009779                core::result::unwrap_failed::h456a23f68607268c("called `Result::unwrap()` on an …"  , 0x2b, &va
00009779                /* no return */
00009779            }
```

Figure 4. Iterator Fold

```
► 0x55555555d59e <crab::main+126>    call   <core::iter::adapters::step_by::StepBy<I> as core::iter::traits::i
terator::Iterator>::fold               <<core::iter::adapters::step_by::StepBy<I> as core::iter::traits::itera
tor::Iterator>::fold>
        rdi: 0x7fffffffdce0 —▸ 0x5555555a8b58 ◂— 0x0
        rsi: 0x0
        rdx: 0x2
        rcx: 0x5555555a8b58 ◂— 0x0
```

Figure 5. Iterator Fold Call Instance

Looking at the arguments being passed to `Iterator::fold()` we can see that an address of the arguments iterator is passed in `rdi`, 0 is passed in `rsi`, 2 is passed in `rdx`, and again the address of the arguments iterator is passed in `rcx`. The meaning of these variables becomes clear once you understand what `Iterator::fold()` is trying to do. That is, it takes an iterator and folds every element into an accumulator by applying an operation, eventually returning the result. So, it takes two arguments, an iterator, and an initial accumulator. Usually, the accumulator is 0, so both arguments are now accounted for.

However, what is the 2 in `rdx` doing? If we look at Figure 5 we can see that it's not just calling an ordinary iterator's fold function. Rather, it is using a StepBy iterator, which creates an iterator from another iterator wherein it iterates over the underlying iterator by "steps". We can assume that the 2 in `rdx` is used as the step in the StepBy iterator.

This would mean that the function iterates over a command line argument by steps of 2 and adds them to an accumulator. As seen in Figure 4, it then compares the result to 6, and if so base64 decodes some value. The string “22221111” we passed earlier fits just this case. How convenient. Running the crab binary with this argument gives us the flag! Of course, you could have just breezed through the challenge by finding the base64 code function and manually decoded it’s argument, but that wouldn’t challenge your knowledge of Rust binaries! After all, not every challenge will just give you the flag like this one did.