

1 | Clojure, in brief...

The programming language Clojure, a variant of Lisp built atop the Java Virtual Machine, is known to be a safe language to low level attacks such as buffer overflow due to the insulation of its execution space.

To preserve the functionality of being a Lisp, however, the Clojure preserves most of the powerful syntax-manipulation tools like unquote slicing and compile-time macros.

Before we could introduce the vulnerabilities that could come with the misuse of these tools, it is important to introduce these tools first.

1.1 | Unquote slicing

"Unquoting" is a facility to introduce `code-mode` elements into `data-mode` lists. Here, for instance, is a regular `data mode` list that evaluates to value `(1 2 3 4)`:

```
'(1 2 3 4)
```

Leveraging "unquoting", we could instead execute a *statement* that returns, for instance, 4 in place of the number 4 in the list instead of supplying the r-value 4 itself: therefore resulting in the same list but introducing live code during its parsing.

The syntax of unquoting is like so:

```
`(1 2 3 ~(* 2 2)) ; notice the ` instead of ' to denote a unquotable list
```

As you could see, the result is still `(1 2 3 4)`, yet the latter case leveraged `(* 2 2)`; \Rightarrow 4 to *unquote* the value 4 into the list using live code.

1.2 | Macros

Macros are a powerful way of manipulating the syntax of code. Essentially, they are compile-time unraveled directives that take arguments as `data-mode` lists and return a sliced `data-mode` expression.

Here's a basic example of the functionality of Macros. Take, for instance, the following expression adding two numbers:

```
(+ 1 2)
```

We will now write a macro whose behavior is to reverse the order of the first two expressions before evaluating, like so:

```
(defmacro rev-arg [a b c]  
  `(~b ~a ~c))
```

Now, leveraging this macro, we could write the addition statement in an now admittedly weird manner:

```
(rev-arg 1 + 2)
```

As you could see, this macro did not *evaluate* its arguments, but instead took them as `data-mode` items and swapped their order. The return value of the macro is yet another `data-mode` list representing the unwrapped expression (now with the order corrected), which is finally evaluated on runtime.

2 | Dangers of Macro Arguments

Macros do not evaluate nor — by default — make any attempts to parse the contents of its arguments. Therefore, if an argument is unquoted wholesale into the result unwrapped expression, it is functionally equivalent to calling `eval` upon the statement as it will not be parsed at *compile-time* but instead on *run-time*.

Especially dangerous an action is if the argument of macros come from user input, as — with functions — the argument would either be passed consciously as fully data or evaluated with `eval` but is potentially sliced into the source tree and evaluated automatically on run-time for macros.

Here's the simplest example of runtime evaluation caused by a macro:

```
(defmacro secreteval [a]
  `(do
    (println "one")
    ~a))
```

And now, we call this macro to demonstrate the run-time eval.

```
(secreteval (println "two"))
```

The stdout result is printed in order of `one two`, meaning that, indeed, the execution of the argument `(println "two")` not only took place but took place *after* the execution of the first part of the macro.

Compare this to the following functionally almost-similar *function*:

```
(defn nosecreteval [a]
  (do
    (println "one")
    a))
```

And calling it with the same input...

```
(nosecreteval (println "two"))
```

As you could see, the argument is evaluated and passed evaluated *before* the execution of the macro (likely compiled away by the JVM) and, should user data be passed this was as *data*, will not be evaluated at all due to its persistent as a list.

3 | Preventing Macro Argument Evaluation

The simplest mechanism by which this type of evaluation could be prevented is via the use of assert statements or even list manipulation to establish basic assumptions regarding the argument.

3.1 | Nonexample

For instance, we write the following macro create a list of two elements.

```
(defmacro bad-concat [a]
  `(list 1 ~a))
```

Calling it with some "normal" elements will act as you'd expect.

```
(bad-concat 12)
```

However, you could introduce code execution by, for instance, passing an expression that produces a side effect.

```
(bad-concat (println "Code execution!"))
```

As you could see, the statement `code execution` is *printed*, instead of all of `(println "aoenust")` being concatenated to the list to 1, as the author of the program presumably desired.

3.2 | assert based solution

One way to solve this is by asserting that the argument `a` is an `atom` — meaning it does not contain `s-expressions` that could be accidentally evaluated. This would be implemented as follows:

```
(defmacro better-concat [a]
  (assert (not (list? a)))
  `(list 1 ~a))
```

The first example returns what one would expect:

```
(better-concat 12)
```

And the statement with side-effects would, on *compile-time* throw an `AssertError`.

```
(better-concat (println "Code execution!"))
```

3.3 | More ergonomic solution

There is an even more ergonomic solution to this problem. Instead of validating input, we could coerce the input to `data-mode` using the `quote-expression`, essentially sanitizing it. Hence, we could create an even more ergonomic `concat` like so:

```
(defmacro ergo-concat [a]
  `(list 1 '~a))
```

The first example still remains the same:

```
(ergo-concat 12)
```

and the side-effect statement will perform more ergonomically:

```
(ergo-concat (println "Code execution!"))
```

As you could see, the list is actually, properly sliced in instead of triggering executing code.

4 | Legal/Ethical Concerns

The problem that exist with Clojure macro evaluation is that it is very easily (and perhaps commonly) created by even seasoned Clojurists. Acc