WHITEPAPER

Secure Julia Coding Best Practices

Version 1.1 (July 2023)

Introduction

This document describes a set of best practices for programming securely in Julia. The operations described here do not automatically make the code unsafe, but such operations should be used after giving due consideration to their benefits.

Julia is a high-level, type-safe and memory-safe language, and so automatically avoids many of the potential security flaws of lower-level languages like C/C++. For example, all array bounds, accesses to uninitialized references, and type conversions are checked at run time, so buffer overflow and similar attacks are by default not possible for native Julia code.

However, there are also security vulnerabilities that do not stem from programming-language-level problems. That class of problems can be addressed by following the guidelines in this document.

Vulnerability Reporting

If you believe you have discovered a security vulnerability in Julia, please email the Julia Security team at security@julialang.org.

Unsafe Operations

Like other high-level languages that provide strong safety guarantees by default, Julia nevertheless has a small set of operations that bypass normal checks. These operations are clearly marked with the prefix unsafe_. By using an "unsafe" operation, the programmer asserts that they know the operation is valid even though the language cannot automatically ensure it. For high reliability these constructs should be avoided or carefully inspected during code review. They are:

- `unsafe_load`
- `unsafe_store!`
- `unsafe_read`
- `unsafe_write`
- `unsafe_string`
- `unsafe_wrap`
- `unsafe_convert`
- `unsafe_copyto!`
- `unsafe_pointer_to_objref`

- `ccall`
- `@ccall`
- `@inbounds`

Some of these constructs will be discussed further in sections below.

Non-public Operations

The Julia standard library has an intended public API indicated by marking symbols with the `export` keyword. However, it is possible to use non-public names via explicit qualification, e.g. `Base.foobar`. This practice is not necessarily unsafe, but should be avoided since non-public operations may have unexpected invariants and behaviors, and are subject to changes in future releases of the language.

Note that qualified names are commonly used in method definitions to clarify that a function is being extended, e.g. `function Base.getindex(...) … end`. Such uses do not fall under this concern.

Accessing Uninitialized Data

For certain newly-allocated data structures, such as numeric arrays, the Julia compiler and runtime do not check whether data is accessed before it has been initialized. Therefore such data structures can “leak” information from one part of a program to another. Uninitialized structures should be avoided in favor of functions like zeros and fill that create data with well-defined contents. If code does allocate uninitialized memory, it should ensure that this memory is fully initialized before being returned from the function in which it is allocated.

Example:

```
function make_array(n::Int)
    A = Vector{Int}(undef, n)
    # function body
    return A
end
```

This function allocates an integer array with undefined initial contents (note the language forces you to request this explicitly). A code reviewer should ensure that the function body assigns every element of the array.

One can similarly create structs with undefined fields, and if used this way, one should ensure all fields are initialized:

```
struct Foo
      x::Int
      Foo() = new()
end

julia> Foo().x
139736495677280
```

External Processes

The Julia standard library contains a run function and other facilities for running external processes. Any program that does this is only as safe as the external process it runs. If this cannot be avoided, then best practices for using these features are

1. Only run fixed, known executables, and do not derive the path of the executable to run from user input or other outside sources.
2. Make sure the executables used have also passed required audit procedures.
3. Make sure to handle process failure (non-zero exit code).
4. If possible, run external processes in a sandbox or “jail” environment with access only to what they need in terms of files, file handles, and ports.

When run in a sandbox or jail, external processes can actually improve security since the subprocess is isolated from the rest of the system by the kernel.

eval()

Julia contains an `eval` function that executes program expressions constructed at run time. This is not in itself unsafe, but because the code it will run is not textually evident in the surrounding program, it can be difficult to determine what it will do. For example, a Julia program could construct and `eval` an expression that performs an unsafe operation without the operation being clearly evident to a code reviewer or analysis tool.

In general, programs should try to avoid using `eval` in ways that are influenced by user input because there are many subtle ways this can lead to arbitrary code execution. If user input must influence eval, the input should only be used to select from a known list of possible behaviors. Approaches using pattern matching to try to validate expressions should be viewed with extreme suspicion because they tend to be brittle and/or exploitable.

It is common for Julia programs to invoke `eval` or `@eval` at the top level, in order to generate global definitions programmatically. Such uses are generally safe.

ccall and @ccall

Calling C (and Fortran) libraries from Julia is very easy: the `ccall` syntax (and the more convenient `@ccall` macro) allow calling C libraries without any need for glue files or boilerplate. They do require caution, however: the programmer tells Julia what the signature of each library function is and if this is not done correctly, it can be the cause of crashes and thus security vulnerabilities. An exploit is just a crash that an attacker has arranged to fail in a worse way than it would have randomly.

Safe use of `ccall` depends on both automated and manual measures.

What Julia Does (Automated)

- Julia provides aliases for C types like `Cint`, `Clong`, `Cchar`, `Csize_t`, etc. It makes sure that these match what the C ABI on the machine that code is running on expects them to be.
- The Clang.jl package automates the process of turning C header files into valid `ccall` invocations.
- Pkg+BinaryBuilder.jl allows precise versioning of binary dependencies.
- Julia objects passed directly to `ccall` are protected from garbage collection (GC) for the duration of the call.

What You Must Do (Manual)

- When writing `ccall` signatures, programmers should always look at the signature in the C header file and make sure the signature used in Julia matches exactly.
- Use Julia's C type aliases. For example, if an argument in C is of type int then the corresponding type in Julia is `Cint`, not `Int` — on most platforms `Int` will be the same size as `Clong` rather than `Cint`.
- If a raw pointer to memory managed by Julia's GC is passed to C via `ccall`, the owning object must be preserved using GC.@preserve around the use of `ccall`. See the documentation of this macro for more information and examples of proper usage.

Bounds Check Removal

While Julia checks the bounds of all array operations by default, it is possible to manually disable bounds checks in a program using `@inbounds`. Uses of this construct should be carefully audited during code review. For maximum safety, it should be avoided or programs should be run with the command line option `--check-bounds=yes` to enable all checks regardless of manual annotations.

To check a use of `@inbounds` for correctness, it suffices to examine all array indexing expressions (e.g. `a[i]`) within the expression it applies to, and ensure that each index will always be within the bounds of the indexed array. For example the following common use pattern is valid:

```
@inbounds for i in eachindex(A)
    A[i] = i
end
```

By inspection, the variable `i` will always be a valid index for A.

For contrast, the following use is invalid unless A is known to be a specific type (eg Vector)

```
@inbounds for i in 1:length(A)
    A[i] = i
end
```

`@inbounds` should be applied to as narrow a region of code as possible. When applied to a large block of code, it can be difficult to identify and verify all indexing expressions.

Secure Randomness

The default pseudo-random number generator in Julia, which can be accessed by calling `rand()` and `randn()`, for example, is intended for simulation purposes and not for applications requiring cryptographic security. An attacker can, by observing a series of random values, construct its internal state and predict future pseudo-random values. For security-sensitive applications like generating secret values used for authentication, the `RandomDevice()` random-number generator should be used. This produces genuinely random numbers which cannot be predicted.

Code Injection

Much like the previous discussion of the `eval()` function, when writing programs that construct any kind of code based on user input, extra caution is required and the user input must be validated or escaped. For example, a common type of attack in web applications written in all programming languages is SQL injection: a user input is spliced into an SQL query to construct a customized query based on the user's input. If raw user input is spliced into an SQL query as a string, it is easy to craft inputs that will execute arbitrary SQL commands, including destructive ones or ones that will reveal private data to an attacker. To prevent this, the user input should be passed as parameters to SQL prepared statements; a package such as SqlStrings.jl can be used to do this without a syntax burden. This protects against malicious input, but also encourages systematic marshaling of julia types into SQL types. If string interpolation must be used, all user input should be either validated to match a strict, safe pattern (e.g. only consists of decimal digits or ASCII letters), or it should be escaped to ensure that SQL treats it only as data, not as code (e.g. turn a user input into an escaped string literal).

While we have talked specifically about SQL here, this issue is not limited to SQL. The same concern occurs when executing programs via shells, for example. Julia is more secure than most programming languages in this respect because the default mechanism for running external code (see Cmd objects in the Julia manual) is carefully designed to not be susceptible to this kind of injection, but programmers may be tempted to use a shell to call external code for convenience sake. The fact that a shell must be explicitly invoked in Julia helps catch these kinds of circumstances. Using a shell like this is usually a bad idea and can typically be avoided. If an external shell must be used, be certain that any user data used to construct the shell command is carefully validated or escaped to avoid shell injection attacks.

Distributed Computing

Julia's distributed computing uses unencrypted TCP/IP sockets for communication by default and expects to be running on a fully trusted cluster. If using `Distributed` in your code through `@distributed`, `pmap`, etc., be aware that the communication channels are not encrypted. Julia opens ports for communication between processes in a distributed cluster. A pre-generated random cookie is necessary to successfully connect (Julia 0.5 onwards), which defeats arbitrary external connections. This mechanism is described in detail in https://github.com/JuliaLang/julia/pull/16292.

For additional security, these communication channels can be encrypted through the use of a custom ClusterManager which enables SSH port forwarding, or uses some other mechanism to encrypt the communication channel.

Managing Secret Data

When it's necessary to manage secret data (for example, a user's password) it's desirable to have this erased from memory immediately after finishing with the data. However, when a normal `String` or `Array` is used as a container for such data, the underlying bytes persist after the container is deallocated and in principle could be recovered by an attacker at a later time. There are also situations where a string or array may be implicitly copied, for example, if it is assigned to a location with a compatible but different type, it will be converted, thus creating a copy of the original data. Normally this is harmless and convenient, but making copies of secrets is obviously bad for security. To prevent this, Julia provides the type `Base.SecretBuffer` and a `shred!` function which should be called immediately after the data is finished with. The contents of a `SecretBuffer` must be explicitly extracted — it is never implicitly copied — and its contents will be automatically shredded upon garbage collection of the `SecretBuffer` object if the `shred!` function was never called on it, with a warning indicating that the buffer should have been explicitly shredded by the programmer.