

[Discover Packages](#) > [Standard library](#) > [plugin](#) 

plugin


package

standard library

Version: [go1.20.1](#) **Latest** | Published: Feb 14, 2023 | License: [BSD-3-Clause](#) | Imports: 4 |

Imported by: 2,633

Details

 Valid [go.mod](#) file  Redistributable license  Tagged version  Stable version [Learn more](#)

Repository

[cs.opensource.google/go/go](#)

Links

 [Report a Vulnerability](#) Documentation 

<> Documentation

Rendered for [linux/amd64](#) 

Overview

Warnings

Package plugin implements loading and symbol resolution of Go plugins.

A plugin is a Go main package with exported functions and variables that has been built with:

```
go build -buildmode=plugin
```

When a plugin is first opened, the init functions of all packages not already part of the program are called. The main function is not run. A plugin is only initialized once, and cannot be closed.

Warnings

The ability to dynamically load parts of an application during execution, perhaps based on user-defined configuration, may be a useful building block in some designs. In particular, because applications and dynamically loaded functions can share data structures directly, plugins may enable very high-performance integration of separate parts.

However, the plugin mechanism has many significant drawbacks that should be considered carefully during the design. For example:

- Plugins are currently supported only on Linux, FreeBSD, and macOS, making them unsuitable for applications intended to be portable.

- Applications that use plugins may require careful configuration to ensure that the various parts of the program be made available in the correct location in the file system (or container image). By contrast, deploying an application consisting of a single static executable is straightforward.
- Reasoning about program initialization is more difficult when some packages may not be initialized until long after the application has started running.
- Bugs in applications that load plugins could be exploited by an attacker to load dangerous or untrusted libraries.
- Runtime crashes are likely to occur unless all parts of the program (the application and all its plugins) are compiled using exactly the same version of the toolchain, the same build tags, and the same values of certain flags and environment variables.
- Similar crashing problems are likely to arise unless all common dependencies of the application and its plugins are built from exactly the same source code.
- Together, these restrictions mean that, in practice, the application and its plugins must all be built together by a single person or component of a system. In that case, it may be simpler for that person or component to generate Go source files that blank-import the desired set of plugins and then compile a static executable in the usual way.

For these reasons, many users decide that traditional interprocess communication (IPC) mechanisms such as sockets, pipes, remote procedure call (RPC), shared memory mappings, or file system operations may be more suitable despite the performance overheads.

Index

type `Plugin`

```
func Open(path string) (*Plugin, error)
```

```
func (p *Plugin) Lookup(symName string) (Symbol, error)
```

type `Symbol`

Constants

This section is empty.

Variables

This section is empty.

Functions

This section is empty.

Types

type `Plugin`

```
type Plugin struct {  
    // contains filtered or unexported fields  
}
```

Plugin is a loaded Go plugin.

func **Open**

```
func Open(path string) (*Plugin, error)
```

Open opens a Go plugin. If a path has already been opened, then the existing *Plugin is returned. It is safe for concurrent use by multiple goroutines.

func (*Plugin) **Lookup**

```
func (p *Plugin) Lookup(symName string) (Symbol, error)
```

Lookup searches for a symbol named symName in plugin p. A symbol is any exported variable or function. It reports an error if the symbol is not found. It is safe for concurrent use by multiple goroutines.

type **Symbol**

```
type Symbol any
```

A Symbol is a pointer to a variable or function.

For example, a plugin defined as

```
package main  
  
import "fmt"  
  
var V int  
  
func F() { fmt.Printf("Hello, number %d\n", V) }
```

may be loaded with the Open function and then the exported package symbols V and F can be accessed

```
p, err := plugin.Open("plugin_name.so")  
if err != nil {  
    panic(err)  
}  
v, err := p.Lookup("V")  
if err != nil {  
    panic(err)  
}  
f, err := p.Lookup("F")  
if err != nil {  
    panic(err)  
}
```

```
}  
*v.(*int) = 7  
f.(func())() // prints "Hello, number 7"
```

Source Files

[View all](#) 

[plugin.go](#)

[plugin_dlopen.go](#)

Why Go

[Use Cases](#)

[Case Studies](#)

Get Started

[Playground](#)

[Tour](#)

[Stack Overflow](#)

[Help](#)

Packages

[Standard Library](#)

[About Go Packages](#)

About

[Download](#)

[Blog](#)

[Issue Tracker](#)

[Release Notes](#)

[Brand Guidelines](#)

[Code of Conduct](#)

Connect

[Twitter](#)

[GitHub](#)

[Slack](#)

[r/golang](#)

[Meetup](#)

[Golang Weekly](#)

[Copyright](#)

[Terms of Service](#)

[Privacy Policy](#)

[Report an Issue](#)



Google