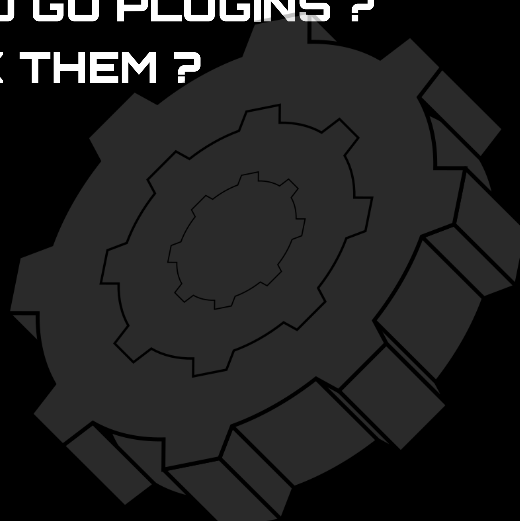
SkyPenguinLabs

Proudly Presents:

REC7 - Reverse Engineering Blackboxed Go Plugins

- WHAT'S INSIDE A GO PLUGIN ?
  - LEARN GHIDRA BASICS
- TWO TYPES OF GO PLUGINS ?
- WHAT ARE BLACKBOXED GO PLUGINS ?
  - HOW CAN WE BREAK THEM ?



Simplicity > Complexity

Author: @Totally_Not_A_Haxxer

///.//././/

SPL-REC7

TABLE OF CONTENTS

FOR THE COURSE: **Reverse Engineering Blackboxed Go Plugins**

SkyPenguinLabs_REC7

SECTION: Section 0x00 | *Course Synopsis & Details*

COURSE: Reverse Engineering Blackboxed Golang Plugins

Pages: 25

Difficulty: (o) Beginner [x] Intermediate (o) Expert

Synopsis:

This course was designed to be a really deep but still surface level introduction to what you can do with Go's plugin system, and how we can take advantage of the dynamic behavior to reverse engineer them. This course will also talk about how they're built, why they behave differently from other compiled binaries, and how to reverse engineer them to uncover hidden functionalities.

Across multiple structured sections, we'll move from the fundamentals of Go's plugin system to hands-on reversing and reimplementation techniques using tools like **Ghidra**. You'll not only understand how these plugins work under the hood but also gain the skills to pick apart black-boxed Go plugins in your own research.

Section Synopsis;

- **Section 0x03** | *Plugins, and the Go Programming Language*: This section will breakdown what plugins are, how they are used, why they get used despite bold claims by critics, and how go works with them by building our first plugin!
- **Section 0x04** | *Reverse Engineering Go Plugins*: This section bridges from section 0x03, from building our own plugins to dissecting them with Ghidra, and no other plugins within Go. Then taking the information obtained from Ghidra to interact with the plugin and observe the outputs.
- **Section 0x05** | *To Implementation & Beyond*: Building on top of the work and knowledge we have obtained from picking these plugins apart, we can now move onto what comes after - automation. We will then be discussing what you can do with observability!
- **Section 0xFF** | *Conclusion & Final Notes*: Ends the lesson, concludes with final notes, and adds anything extra that was missing along the lesson.

SkyPenguinLabs_REC7

SECTION: Section 0x01 | *Introduction & Authors Notes*

As a reverse engineer, you have probably spent some time reverse engineering programs which interact with other services, system drivers, and more. *But what about plugins? More specifically, what about black boxed plugins?*

Well, in today's world, plugins are becoming bigger and bigger by the second. Additionally, with languages like Go which **natively support plugins** within their own language, it becomes important to learn what they are, how to secure them and how to interact with them when we need to.

The first proponent of this is learning what an attacker is going to need to do first before being able to get the information necessary to exploit or mess up a plugin - reverse engineer it.

Because reverse engineering binaries compiled by the Go programming language is a little bit of a pain without debug information, or having a dynamic environment properly setup + fancy tools, we need to refocus our angles.

Because plugins within Go, are native to Linux, compiled as Shared Objects, and are dynamic, we can take advantage of their structure to gain insights on their internal functions/symbols, then conversely use the native Go plugin library to interact with those internal functions by loading the plugin inside of a Golang runtime (*e.g: creating an app that loads the plugin into memory*) which allows us to figure out what the functions actually do.

This is exactly the angle we will be pivoting from and using to grab a foothold on what a plugin may actually be doing with information we give it. Additionally, this may open up the door for so many other possibilities.

I want to mention, while we are reverse engineering applications created by the Go compiler, we will _not_ be exploring any advanced, new or wild techniques for reverse engineering Go-compiled specific applications. This is primarily due to the amount we have to cover. May this serve as an introduction to reversing Go.

SkyPenguinLabs_REC7

SECTION: Section 0x02 | *Prerequisites for This Course*

This lesson will require us to have the following environment setup.

A. **Virtual Machine running** some version of Linux 6.1. We chose parrot6, running on Debian for ease of setup.

B. **Golang compiler version** go1.23.5 linux/amd64

C. **Ghidra** (*preferably version 11.4.1*)

D. **Your preferred IDE**

The difficulty of this lesson was set to intermediate, primarily because you will need to have some background experience, having understood the fundamentals of

- Computer Science
  - How shared objects work on Linux
  - What ELF files are
- Programming & Go Development
  - Basics of the Go programming language
  - Go's compiler differences: Windows vs Linux
- Reverse Engineering on the x86-64 bit architecture
- Setting up & Using Ghidra
- Theory of Reverse Engineering

This is a pretty in-depth lesson, so be prepared, and make sure you have everything set!

Since you will be downloading a binary from here, and running it, make sure you do NOT download it from anywhere else or from anyone else claiming to have the same binary. If you do, DO NOT RUN IT, UNTIL YOU VERIFY that the SHA1 hash is the exact same as the one on this page **here**.

SkyPenguinLabs_REC7

SECTION: Section 0x03 | *Plugins, and the Go Programming Language*

Plugins are a mystery to many people who do not have a large array of experience working in ecosystems which are co-dependant on other applications, or sources. Additionally, even to those who find a use for them, often can find another way around it without having to use plugins.

Wait…it doesn't help if I keep saying the word 'plugin' when you hardly have an understanding of what that actually is. So, lets break that down first.

What the hell are plugins???

For those who are new, **plug-ins** are essentially pre-compiled, independent, stateful (*in-process*) programs whose state can only be considered active or used when invoked (*called*) by another program which imports it.

Before plugins can even be loaded, they typically go through their own compilation phase. For example, making a Rust plugin, involves developing and testing the code in isolation before being able to integrate it into an existing sel built plugin system. This process often results in different files being generated, different structures of information being recorded. I say 'self built' because oftentimes, plugin systems are not usually native to use, and are incredibly annoying to build in theory and can be even more annoying in implementations when you do it yourself.

Typically, plugins are able to facilitate communication through some form of text-based or binary-based protocol, usually defined within the program's functionality that is calling or invoking the plugin - a plugin system, or plugin manager of sorts (*as I was saying above*).

Depending on who you ask, and depending on the context, plugins can mean different things. From a simple basic functionality extension on an application, to isolating logic for security and containment purposes.

Now the first important thing to get to, is what their main and primary use cases are inside of applications.

How are plugins used???

Depending on the context, plugins may be used for various reasons. The following list below describes the most popular reasons for using plugins.

- **Functionality Extensions -** Some brands or companies behind software products may often release plugins to extend the functionality of their application. For example, users that pay a pro subscription on a banking app may obtain a set of limited edition plugins that allow users on their app to trade cryptocurrencies.

- **User based modification -** Some applications are loose, and accept the security issues (*which we will get into soon*) that come with letting users develop their own plugins. This is often known as a user-based modification, or user-developed plugin. User-developed plugins are often considered problematic because it allows anybody external to the actual authorized development team to develop plugins which will be loaded inside of an applications runtime. If the plugin was developed intentionally to break an application, then it becomes problematic for anybody who downloads that plugin, as when users are allowed to develop plugins, typically other users are also allowed to download those plugins once published. This basic **steam forum** is enough to encapsulate that thought.

- **Functionality Isolation** - In less common and more unconventional scenarios, developers may choose to isolate specific functionalities into plugins. This approach, while sometimes considered ad hoc or "hacky," leverages the stateful nature of plugins to temporarily store lightweight functions or scripts. These plugins can be dynamically loaded, executed, and then unloaded without affecting the core application. Developers may use this technique as a modular workaround to introduce or test isolated features without making permanent changes to the main codebase.

Regardless of the **way** plugins get used within the application, the way the plugins logic gets invoked is primarily dependent on the plugin loader.

I noticed I never really solidified these two terms so lets make sure we get them down pat before moving further.

- **Plugin** - pre-compiled, independent, stateful programs whose state can only be considered active or used when invoked (*called*) by another program which imports it.

- **Plugin Manager / Plugin Loader -** This is a component or set of logic inside of a program entirely dedicated to manually managing the state of the plugin, and the use of the plugin.

Lets continue onto why exactly somebody would want to invite a plugin into their codebase after some of our comments have been made.

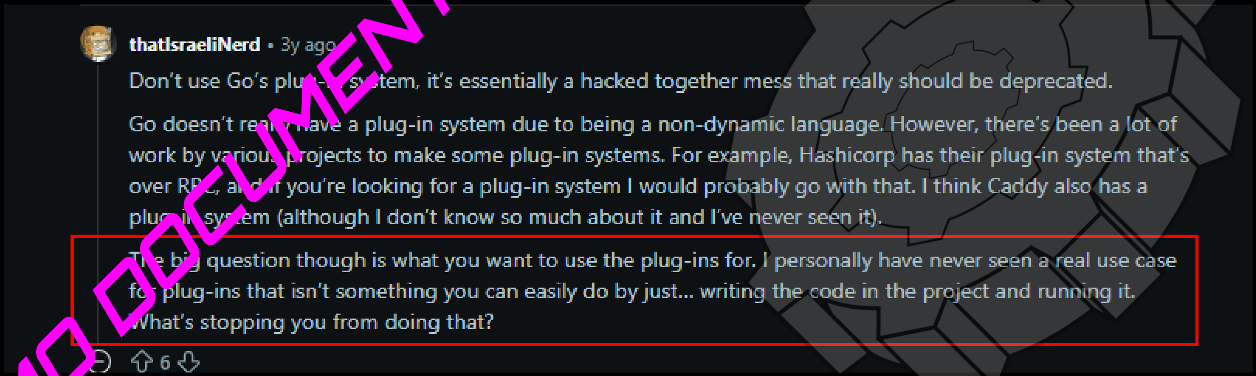
- Note: We are starting to get language specific, so the next section specifically talks about plugins in relation to the Go programming language.

Why would we use them?

For anybody that is not so new to the computer science scene, then you may already have the answer to this question. And for anybody else, as you could tell, by the section above, there are clear obvious reasons why they would be used.

But, we also come across one fancy debate inside of the dev community that seems to be pretty common.

- **"What is the difference between using a plugin and just calling a script and capturing the output through STDIN/STDOUT"?**


thatIsraeliNerd • 3y ago

Don't use Go's plug-in system, it's essentially a hacked together mess that really should be deprecated.

Go doesn't really have a plug-in system due to being a non-dynamic language. However, there's been a lot of work by various projects to make some plug-in systems. For example, Hashicorp has their plug-in system that's over RPC, and if you're looking for a plug-in system I would probably go with that. I think Caddy also has a plug-in system (although I don't know so much about it and I've never seen it).

The big question though is what you want to use the plug-ins for. I personally have never seen a real use case for plug-ins that isn't something you can easily do by just... writing the code in the project and running it. What's stopping you from doing that?

6

Well, there is a **huge** difference for many reasons. Depending on **what** you want to do for one, which we have done, but even then, there are other things where that is not feasible.

The first problem being that running a separate code project or compiling for something as such, like a single function, or isolated functionality becomes overly problematic to manage in scenarios where codebases have to grow.

It seems that many people **feel** like plugins are meant to have these massive sets of functionalities, huge files that have to be managed, dependencies and modules that get imported etc. When in reality, plugins were not, especially in Go, designed for this.

Plugins are also very wacky, and a lot of people forget that when developing plugins alone, you need to make sure you know:

- **What the plugins functionality is ->** Is it an isolated functionality, or is it an entirely new component that's supposed to plug directly into the application out of the box?

- **What type of plugin you are going to be building ->** Is it an In-process or separate processes plugin?

If you don't grasp what these are, their advantages versus disadvantages, it becomes quite easy to become blindsided by what the context of use actually is for the plugin and think that a plugin is nothing more than the equivalent of calling something like `exec.Command('run_external_script')` XD

Speaking of plugin types, let's get some down.

The two types of plugins

When developing plugins, it's important to consider that the plugin you are dealing with is going to be in some shape or form interacting with the process. But the way it interacts is often dependent on the type of plugin it is.

For the general surface introduction, there are two types of plugins that can be used to categorize interaction which are:

- **In process Plugins -** In-process plugins run within the same process as the main application. When you use in-process plugins in Go, the plugin code executes in the same memory space as your main program. This is often going to be plugins that get compiled at runtime often by a plugin loader, or interpreter (*such as a* *the lua interpreter*) that can interpret and execute the code inside of the plugin file.

- **Separate Process Plugins -** Separate-process plugins run as independent processes outside the main application. The main application communicates with these plugin processes through various methods. An example of how this gets implemented is through **RPC** via stdin/stdout:, where the main process starts the plugin process and they communicate through standard input/output using Remote Procedure Calls (RPC). Yes, this can also be **done over a network**. RPC, is a type of **IPC**. IPC is the proper way to facilitate process communication.

These two types of plugins also have their own advantages and disadvantages when being used. Since this lesson primarily focuses on in-process plugins, as the native package within the Go programming language for dynamic loading of applications, we will be starting with in-process plugins.

> 1) Advantages & Disadvantages of [In Process] Plugins

In-process plugins are a heck of a lot easier to manage. This is primarily because, when they are done correctly, you often end up with:

- **Good Performance -** Which is often due to the lack of overhead created by other IPC interfaces or frameworks.

- **Easy Deployment -** They can be seriously easy to deploy or host and if you do not spend a load of time having to deal with versioning differences (*or even build a plugin that is version specific*)

- **Simpler runtime management -** The caveat with go is that the runtime is quite beefy, but it's in exchange for the fact that the runtime does not need to be discovered, initialized or contain any specific health checks in relation to the runtime

However, on the other side you get some barriers that come with them as well such as the following:

- **Crash Impact! -** With in-process plugins, the one disadvantage (*oftentimes*), especially with Go, is that when loading them, they can crash the main process if code in the loaded plugin fails. This is because they have the same runtime, and shared memory space. Which means there is no process boundary blocking potential faults.

- **Platform Support -** Currently, specifically with Go, in-process plugins are only supported on Linux. Additionally, many platforms keep plugin support strict to specific groups of platforms because cross platform plugins often are hectic to manage.

- **No Hot Swapping -** A lot of people love plugins, especially over live production systems when you can hot swap them (for those who don't know, hot swapping is the replacement or addition of components to a computer system without stopping, shutting down, or rebooting the system). No hot swap is something a lot of people don't like because it disables live-updating, and also gets messy when you have a large amount of plugins to manage.

> 2) Advantages & Disadvantages of [Separate-Process] Plugins

Separate process plugins can be a bit of a pain to manage, but once you get the hang of them, they carry some benefit.

- **Hot Reload -** Hey look at this fancy boi! This one actually allows you to hot reload plugins. Making it greatly suitable for live environments.

- **Fault Isolation -** If a fault or panic happens within the primary plugin routine, it gets properly handled away from the main process.

- **Resource Control -** Having a separate process plugin also makes resource allocation a lot easier. While you can technically do this natively by trying to work the runtime, it becomes more of a pain with in-process plugins.

- **It gets slow -** Sometimes, because it's IPC, there is a lot of performance overhead that often results in slower response times for time-critical applications. This is often because of a few things- the first being process isolation. Each plugin exists as its own operating system process with its own memory space, completely separate from the main application. This conversely means each plugin has its own independent lifecycle to manage. So Plugins can be started, stopped, restarted, or even upgraded without affecting the main application. But because all of this requires IPC, there will be some latency between the calls. And if things need to be synchronized, it may not be the best option.

- **Added Development Complexity -** Depending on who you are, this can be good or bad. Personally, a little complexity is okay. But be warned. Working with RPC is definitely going to take some understanding if you want to do it right, and that will start to get complex. It's nothing scary, and it may actually be more worth it for you because of the benefits you can gain out of using it.

Now that we understand the type of plugin we will be taking a look at today (*in-process*), we can now start to move onto the actual implementation of plugins which involves the security theory that surrounds them, then transfer into actually building our very first plugin with Go!

Plugins and Security...ohhhh boy.

I know I waited quite a bit to get to this section, but it's for a very good reason.

Plugins are problematic for many reasons, and as we discussed right off the bat, you are literally inviting external code into an application to be called, without actually knowing who it's from.

Let's go by scenario, and see if we can pinpoint some security issues.

- **Scenario A)** A proprietary application, specifically Desktop application for Windows and Linux has a custom built in in-process plugin system designed specifically for its own plugins. By default, it downloads, cache's and loads into memory a plugin from a local file location the app always uses. The filename of the plugin is hardcoded in the app, the plugin is not checked with binary integrity rules, and is not verified.

- **Issue A)** The first problem for security nerds becomes obvious. Familiar with [DLL sideloading]()? Yes? Well our first issue has a similar approach. If the application loads a plugin named `loginfunctionality.bin`, which contains a function `LoginFunctionality' and an attacker has `loginfunctionality.bin`, which also contains 'LoginFunctionality' with the same exact requirements, the attacker could easily replace the real plugin with their own, and see if the application loads it. The simple solution to this would be to implement some form of code signing. However, not everybody is smart enough, or cares enough to do this in design.

- **Scenario B)** Open source game allows players to write their own lua plugins for the sake of developing their own themes for the game. Only, the way the application parses the code to load the lua plugins involves taking the file from the user, and passing it to the lua interpreter to be executed, then for the program to take the results of execution and apply them to global configuration settings.

- **Issue B)** The application executes user-provided Lua code directly without proper sandboxing or input validation, creating a code injection vulnerability that allows malicious plugins to escape the intended runtime environment and potentially compromise the host system.

Generally speaking, the safety of a plugin system (like all other systems) requires that of its implementation. However, it's also important to understand that they do introduce their own forms of security loopholes.

Since security is kind of broad with plugins, and more specific to the scenario, and vulnerability at hand (*such as RCE obtained with a plugin, rather than RCE in a plugin*), additionally this being an RE article, we won't really have a lot to speak about on security. That is also why I saved this section for last- since it was the least important out of all sections in this course.

The next page will be able to give us a glimpse of what plugins in the Go programming language are like, and how implementing them becomes quite wonky with safety off the bat!

Getting Hands on!

Before we get hands on, it's important for me to make sure you have everything you need before continuing! That includes:

- A working VM running a version of Linux able to support the Go compiler (*we used debian/parrot6*)
- A working, latest install of the **Ghidra** reverse engineering framework
- **Go compiler**, version 1.21.0+ Linux
- Your choice of an IDE, we used **vscode**

Now, what exactly are we going to be developing?

Since this is most likely your first time with Go plugins, we are just going to be writing a very simple hello world plugin. The cool unique thing about this plugin though? It changes the color of 'hello world' every single time its called O_O!!!!!! SO SPECIAL!

Doing this is going to be simple.

- 1) We will have one main program, which has an entrypoint. This entrypoint will call the plugin package from Go to load and invoke a plugin symbol!

- 2) The plugin, on the other hand, is going to be where our beefy logic is at. This will contain 3 functions

  - `local_GetRandomHexValue` - Will randomly generate 6 hex values in between 0 and 255.

  - `local_HextoAscii` - Takes an input set of 4 hex values as a string of RRGGBB and converts it to ASCII escape sequences which can be used in text output.

  - `Hello` - Takes 0 arguments, called only once, lives only once, uses both local functions to generate a random color value, and prints *'hello world'* with that color to the screen.

Making this setup is going to be pretty easy. Let's run the following commands to setup our environment.

```
mkdir HelloWorld                                        # Root directory
mkdir ./HelloWorld/hello_world_plugin                   # Make our plugin directory
cd HelloWorld                                           # Change into directory
go mod init main                                        # Initiate a module
touch main.go ./hello_world_plugin/plugin.go            # Create main files
```

This now means our folder tree should look like this.

```
./HelloWorld
├── go.mod
├── hello_world_plugin
│   └── plugin.go
└── main.go
```

Now if you have everything you want to set up, let's open up our IDE and start getting to work on this app!

[Note]:

If you want to be lazy, I decided to upload this entire script to github for those who already can grasp the contents. It's published on this gist.

Since the plugin library literally only contains the better half of two functions to call, our main script is going to be super simple.

```
1) func Open(path string) (*Plugin, error)
2) func (p *Plugin) Lookup(symName string) (Symbol, error)
```

Inside of our main.go file, we are going to create a heading with the following at the top.

```go
package main

import (
	"log"
	"plugin"
	"reflect"
)

func CE(x error) {
	if x != nil {
		log.Fatal(x)
	}
}

```

We only need three libraries here

- Log - for standard logging and output of information
- Plugin - for the main plugin interaction
- Reflect - because before we interact with anything such as a function within the plugin, the reflect package becomes helpful for knowing the data type expected for the data we are pulling from the plugin.

Additionally, I added an error helper function which stops us from having to constantly write the error checking statement over and over. That's all the `CE` function does - Check Error.

With that we can define our main routine. All this will do is use plugin.Open to open up a plugin filename called ./plugin, then lookup a symbol called HelloWorld, finally taking that symbol and using the reflect package to make sure it's a function before executing it.

```go
func main() {
    ////// Open the plugin here
    p, x := plugin.Open("./plugin")
    CE(x)

    ////// @Check: Function exists?
    sym, x := p.Lookup("HelloWorld")
    CE(x)

    ////// @Check: Is it a func?
    Routine := reflect.ValueOf(sym)
    ///// Plugin symbol MUST be a function, while we can not dynamically
    ///// detect arguments right away, we do know the symbols type
    if Routine.Kind() != reflect.Func {
        log.Fatal("[-] Symbol type is not a function. Must be function")
    }

    ////// Call the symbol, once we typecast it to a func()
    sym.(func())()
}
```


As you can tell, with the image above, the control flow is pretty obvious.

Now, what do we do with the plugin?

For the plugin, navigate to the `./hello_world_plugin/plugin.go` file and start with the following imports.

```go
package main

import (
    "fmt"
    "math/rand"
    "strconv"
    "time"
)
```

There is not a lot of code we will be using, but I also figured I would do you the good honor of pre-providing you with the set of functions in those libraries we use, followed by their purpose in the code we will be writing for the plugin.

- fmt – Used for string formatting
    - fmt.Sprintf creates RGB values into a 6-character hex string `("%02X%02X%02X")`, formats the ANSI escape sequence string to insert numeric RGB values into `\033[38;2;%d;%d;%dm`
- math/rand - Provides pseudo-random number generation
    - `rand.Intn(256)` is first out of the package for generating integer values between 0 and 255 for R, G, and B channels. This is the start of our hex code generator function
    - `rand.NewSource(time.Now().UnixNano())` is used to seed the generator with the current time in nanoseconds to ensure different results on each run
- strconv - Handles string-to-integer conversions for RGB parsing
    - `strconv.ParseUint` is used to convert each two-character hex segment (*RR, GG, BB*) into numeric values. Parsing with base 16 to make sure the hex strings are correctly interpreted as integers for the ANSI sequence
- time - Provides a timestamp system for us to use
    - time.Now().UnixNano() supplies a unique seed to rand.NewSource to avoid repeating the same RGB color pattern across executions.