—- Media Contacts

Email - **skypenguinlabs@gmail.com**

This book, BHGM - Black Hat Go Manual, explores the realm of security-related programs and code. The content within these pages is intended for educational purposes only and should not be used to engage in illegal activities. The author and publisher do not endorse or promote any form of malicious intent, and the techniques described herein should be utilized solely for ethical and legal purposes. All names, characters, businesses, places, events, and incidents mentioned in this book are a work of fiction or used in a fictitious manner. Any resemblance to actual persons, living or dead, or actual events is purely coincidental. The unauthorized reproduction or distribution of this copyrighted work is prohibited. Criminal copyright infringement, including infringement without monetary gain, may be investigated and is punishable by law. The reader is solely responsible for the actions and consequences resulting from the use of the information provided in this book. The author and publisher disclaim any liability arising from the misuse or inappropriate application of the knowledge shared within. Please approach the content of this book with caution and adhere to ethical guidelines and legal frameworks in all aspects of your activities. Remember that knowledge should be used for the betterment of society and the enhancement of security and tech-related communities not to damage it.

Table Of Contents

Book - Introduction and Design choice (1-5)

Golang - Basics To Weirdness (5-83)

Section - BreakPoint (83)

Helpful Programs For Design (83-111)

Digital Forensics / File Forensics (112-143)

Hyper Text Transfer Protocol (HTTP)

Networking / Socketry

Cryptography

Gopacket

Other Sections

About Black Hat Go Manual

The Black Hat Go Manual is a simple booklet/cheat sheet for the Go programming language but for hackers or aspiring security researchers. This book will give some basic to advanced code snippets within the language! It is important to note that most of the code used in this manual will be all cross-platform, however sPythonpecific commands, interfaces, and builds used will be run on a Linux system (Debian-based distro) known as Parrot OS. The author of this book (Totally Not A Haxxer Haxxer) assumes that everyone reading this book will take full responsibility for using the programs in this book for ethical usage, of course, the author does not trust everyone or anyone to do so but I am hoping you do. I AM NOT HELD LIABLE

Purpose of this booklet

The general purpose of this book is to keep things simple. Instead of going 100 pages deep or even 50 on how a giant program or library works using Go, this is rather used as a flipbook for simple programs you can create and understand with simplicity but also with sections that become more educational and deeper than others. The hope of this booklet is that it will help its readers in the field whenever they need something quick and easy that they can immediately flip to, write and execute on almost any system. Think of this book as a quick and easy

Google search engine or dictionary lookup but rather you flip pages rather than click thousands of links. This book includes programs that are extremely concise yet effective in accomplishing their purpose, providing valuable insights into how particular tasks or operations can be carried out using Go. These programs that are used should be used as a quick reference of how to get things done and understand how they can be done rather than just copying, pasting, and using.

About Go

Go is a statically-typed programming language developed by Google in 2009 to improve **"programming productivity"** within the programming realm. Go is a language that is also widely used by multiple tech companies such as Uber for cloud and server-side development and is used amongst the CyberSecurity community for Defensive and Offensive security frameworks or tools. The Go programming language is chosen due to it having the readability of languages like Python while also having the speed and performance of the C programming language to write in given the access it provides to developers.

Why Golang for security

Go is a programming language that is widely used for both offensive and defensive cyber security due to its performance and its readability. Due to the language being easy to learn along with a very very rich standard library, it allows hackers and developers to write exploits easier or defensive applications easier. Go is also used for its strong ability to work with data analysis!

Go Install

Installing the Go programming language is pretty simple on both Linux and Windows. If you are on Windows you can go to https://go.dev/doc/install and download the latest version, then execute the executable download. For Linux (Debian) you can install it with the command

```
$ sudo apt-get install golang
```

For MacOS you can go ahead to https://go.dev/dl/ and download the download file for mac OS and set it up from there.

Go Commands

The Go compiler has many useful flags that will help you when compiling your source code or working with your code. These flags can help clean your environment, test your code, build and execute your code, and clean out broken or missing dependencies.

go	[args]	Will run the go compiler with your given options and arguments
go build	go build -o [output] [input]	Will tell the go compiler to build the .go file to an executable file named what is after the -o flag
go clean	go clean -modcache	Will clean out all of the download dependencies for every project on your machine
go get	go get [flags]	Will get or download a given dependency
go install	go install [flags] [packages]	Will install and compile any packages that are described
go doc	go doc [package]	Will output general

		documentation of the package
go env	go env	Will output the current environment information such as the GOARCH which is the standard set architecture that golang will build to
go fix	go fix [dir]	Will update the given installed packages to use new APIs
gofmt	gofmt [flags] [statement]	Will take a given package and reformat the source given certain flags
go test	go test [flag] dir	Will take a directory as input and if there are tests located in that file they will be run
go vet	go vet [file or dir]	Will carry out a static analysis of your code, this will check your code for warnings that may not be picked up by the compiler during runtime
golint	golint [file or dir]	Will lint all current files within the directory or the file given as input.
go version	go version	Will output the golang compiler version
go help	go help <tool>	Will output

		documentation or help of a certain tool or command

Go command line examples

Command line usage for the flags listed before can be a bit confusing for new users, so I have laid out a list of some examples that may help you get an idea of how the flags or commands can be used.

```
$ go build -o main main.go
```

The above command will build a file named main.go and output it as an ELF (Executable and Linkable File format) named main.
Once a file is compiled in Go you can execute it like any other executable. It is important to note that Go uses standard environments that are detected during installation to compile the program. This means if you install golang on a machine with an architecture such as amd64 then the GOARCH will be amd64. This will determine the binary's build architecture. In case you ever wanted to change this you can run the command below to set the environment variables.

```
$ GOARCH=amd64 GOOS=windows go build -o main main.go
```

This will compile our main.go file to a DOS executable or Windows executable. You can view supported architectures with the command **$ go tool dist list**.

```
$ go mod init main
```

The above command will create a go.mod file which will hold the project's module name. In our case the go.mod file will have “module main” at the top of it followed by the Go version. If we want to use third-party packages or create our own it is important to create these files.

```
$ go get github.com/google/gopacket
```

The command above will attempt to take that repository on GitHub and download the module and install the package. It is important to note that before you do this you must run the command previous to the one above to ensure these dependencies can be installed properly and work within your project environment.

When the command is executed if the package or module already exists the go tool will not output a message saying it was added, however, if it is not installed it will tell you that the go tool added it to the mod file as a dependency. You might see a file pop up when you do this called a go.sum file, this file contains the cryptographic checksums of the dependencies of a Go project. It is also used to ensure that the dependencies of a project have not been tampered with and are the same as when they were originally installed.

```
$ go doc fmt
```

The above command will tell the go tool to output the documentation on the fmt package. In go you will come across a standard package in the library of the language called fmt, this is the standard way of saying “format”. The format package in go allows you to format strings and data types. This command will `bytes.contains`output the given documentation of the tool including methods in the library.

There is more documentation below inside of the image but it would be too much to show and explain here. The doc tool in go will always output something like this, it is literally command-line documentation similar to man pages but for a language.

```
$ go help build
```


The command above upon execution will output a block of text that will describe how you can use the command `go build`.

```
$ go install golang.org/x/tools/gopls@latest
```

Using go install on systems such as Windows is preferred due to go getting deprecated. Using the command above is how you can replace go get. Go install uses the @ symbol after the entire URL or package name to specify the version number which in our case is the latest number.

Go, A hello world program

Go has many unique ways of working with things, however in order to understand some keen secrets to the language we must first go over how the language looks/operates. Below you will find a table that contains code to the most simple Go program which outputs the phrase “Hello world”.

```
package main

func main() {
    println(“Hello World”)
}
```

First, we define our package name, packages in Go are always at the top level of the program as Go must know what your module is. We will dig deeper into this later but for now, all you need to know is that whatever package was declared when using the command `go mod init` if you even made one must be the package name of your .go file. We use the import keyword to import a package called FMT which stands for `format` to call stronger functions. Finally, we end with declaring a main function, the file you run in our case main.go MUST ALWAYS HAVE A

`main` FUNCTION as if there is no main call the program can not execute. Then we make a call to fmt.Println to output our message. When we run this program using ***`go run main.go`*** we will see that the compiler for go successfully ran the code and will output hello world.

Go Keywords

Below you will find a list of common keywords used within the go programming language.

Keyword	Description
for	Starts a for loop
range	Ranges over an array or value, this statement is used with the keyword for
var	Declare a global or local variable followed by the name followed by the data type
const	Declare a constant keyword or variable
import	Declare imports or files. It might be important to note that all imports MUST be at the top level of the program
func	Define a function
type	Define an alias to a data type or a structure
if	Standard conditional
else	Standard conditional alternative
else if	Standard conditional alternative
switch	Standard switch case statement

case	Along with switch, under a switch block is used as a consequence for a conditional that returns true or false
default	If none of the case statements are triggered default executes the default value
map	Defines a map which looks something like this map[type key]type result
goto	Implementation of goto
defer	Defer

Go Variables

Variables are pretty easy to understand, you can use multiple lists and formats to state variables. There are two major keywords for setting variables which are var and const. Const defines a constant variable which holds a value that can not be changed while var is the opposite. Var sets a fluid variable or a variable that can change. Below you will find example use cases of this.

```
package main


var (
    Name  string
    Name2 = "hello"
)
const (
    PI     = 3.14159265
    MaxAge = 18
)
const keyword = 10
```


```
func main() {}
```

You do not always have to use var() and const() as you can use the keywords whenever and however as long as it is followed by a type assigned to a variable or value assignment to a variable. For context

- **Value-based assignment:** A value-based assignment to a variable is what it sounds like which looks something like a = 10, as you are assigning 10 to the variable similar to when we use Name2 = “hello” in the code above.
- **Type-based assignment:** A type-based assignment is when you declare a variable name followed by the suggested type of the variable which in our var Name instance is a string. So var Name string means we are assigning the data type string to Name and that every or any future value assigned to Name MUST be of type string.

Standard variable assignment can also be done in lists like the following below

```
package main

func main() {
   var1, var2, var3 := "name1", "name2", "name3"
   const Arith1, Arith2, Arith3 = 1, 2, 3
   var n1, n2, n3 = Arith1, Arith2, Arith3
}
```

This is pretty easy to understand, it is a list of variables and values. Some people get confused about this type of list assignment and how it works so let me explain. When we declare var followed by multiple types of values and variables we are assigning the first variable in that list the first value that is declared after the `=` symbol and so on from there. So in our case var1 = name1 and var 2 = name2. Make sense?

Go operations

Go has a pretty unique and interesting way to work with variables which includes all sorts of different operations, keywords, and more. Below you will find a simple code snippet showing how you can declare a variable in a Go program.

```
func main() {
   keyword := "hello world"
   fmt.Println(keyword)
}
```

As you can see declaring a variable is pretty simple, something to also note is that if a value is assigned to a variable it can be inferred by the compiler. In our current case above the compiler will interpret this keyword to be of type string due to it being encapsulated in quotation marks. However, Go also allows you to use the `var` keyword followed by a data type to declare a variable like so.

```
var keyword string
func main() {
   keyword = "hello world"
   fmt.Println(keyword)
}
```

Var also has the ability to infer data types but mostly you will end up just using the var keyword for global variables. It might also be good to mention that like every language Go also has operators for variables which consist of unary operators, relational operators, bitwise operators, logical operators, assignment operators, and arithmetic operators. Below are definitions of the types of operators which were listed above.

Arithmetic Operators	Operators are used to perform

	mathematical operations such as division, subtraction, multiplication, division and etc.
Relational Operators	These operators are also known as boolean operators which are used to find the relation between two variables.
Logical Operators	These are operators that typically relate to boolean algebra which will help compare or add statements
Bitwise Operators	These operators are used to perform what is known as a bit-by-bit operation. This operation is used to manipulate or set individual bits in a number, and are commonly used for low-level programming tasks
Assignment Operators	These operators are used to assign a value to a variable

Below this, you will find a list and examples of individual types of operators. This list will go from the first to last in the chart above starting with arithmetic operators and ending with assignment operators.

Arithmetic operators

+	Addition
-	Subtraction
/	Division
*	Multiplication
%	Modulus

Operators are shown in the example code box below.

```
var data int


func main() {
   data = 19
   fmt.Println(data + 20) // Results in 39
   fmt.Println(data / 20) // Results in 0
   fmt.Println(data * 20) // results in 380
   fmt.Println(data - 20) // results in -1
   fmt.Println(data % 20) // results in 19
}
```

Relational Operators

==	Equal to.
!=	Does not equal or not equal to.
>	Greater than.
<	Less than.
>=	Greater than or equal to.
<=	Less than or equal to.

Below you will find a program showing how these work in a code block.

```
var data int
func main() {
   data = 19
   fmt.Println(data >= 10) // true
   fmt.Println(data <= 10) // false
   fmt.Println(data == 19) // true
```


```
    fmt.Println(data != 19) // false
    fmt.Println(data > 20)  // false
    fmt.Println(data < 20)  // true
}
```

Logical Operators

&&	Logical AND, this operator will return true if both conditions are satisfied.
||	LOGICAL OR, If one of the conditions or both of the conditions are true or satisfied.
!	LOGICAL NOT, if the condition is satisfied it will return true if else it returns false.

Below you will find an example of a program that uses these logical operators in standard conditional statements.

```
func main() {
    const age = 20
    if age != 20 || age <= 18 {
        fmt.Println("Not old enough, must be 18 or
older")
    }
    if age == 20 && age >= 18 {
        fmt.Println("Old enough!")
    }
    if !(age <= 10) {
        fmt.Println("Woah there buddy!")
    }
```


```
}
```

The code should be able to explain itself. The first conditional says if the age is not equal to 20 or the age is not greater than 18 or equal to 18 then they are not old enough. If the age is equal to 20 and the age is greater than or equal to 18 then they are old enough. Finally if the statement age <= 10 returns false then they will be WAY too young.

BitwiseOperators

&	Bitwise AND will compare each bit of the first operand to the corresponding bit of the second operand in the given set of numbers.
|	Bitwise OR will return a 1 in each bit position for which the corresponding bits of either or both operands are 1.
^	Bitwise XOR will take two numbers as an operand and then will return a 1 in each bit position for which the corresponding bits of either but not both operands are 1.
<<	left shift shifts the bits of the first operand, the second operand decides the number of places to shift.
>>	right shift shifts the bits of the first operand, and the second operand decides the number of places to shift.
&^	AND NOT is a bit clear operator.

An example program below is shown using these bitwise operators.

```
func main() {
   Operand1 := 1
   Operand2 := 0
   fmt.Printf("\n Operand1 & Operand2 = %d",
Operand1&Operand2)
   fmt.Printf("\n Operand1 | Operand2 = %d",
Operand1|Operand2)
   fmt.Printf("\n Operand1 ^ Operand2 = %d",
Operand1^Operand2)
   fmt.Printf("\n Operand1 << 1 = %d", Operand1<<1)
   fmt.Printf("\n Operand1 >> 1 = %d", Operand1>>1)
   fmt.Printf("\n Operand1 &^ Operand2 = %d\n",
Operand1&^Operand2)
   /*
   ---- OUTPUT
   Operand1 & Operand2 = 0
   Operand1 | Operand2 = 1
   Operand1 ^ Operand2 = 1
   Operand1 << 1 = 2
   Operand1 >> 1 = 0
   Operand1 &^ Operand2 = 1
   */
}
```

Assignment Operators

:=	Assignment or variable declaration. This operator will create a new variable assigned with a value on its right side, this means the variable name on the left of the operator was not previously declared using var or const keywords.

=	Will assign a value on the right of the sign for the variable on the left if the variable is already declared.
+=	Will add the data of the variable first then add the value being added to the current data.
/=	divides the current value of the variable on left by the value on the right and then assigns the result to the variable on the left.
*=	Will multiply the data of the variable and then multiply the value of the variable being assigned to the right of this operator.
%=	Will first modulo the current value of the variable on left by the value on the right and then assigns the result to the variable on the left.
-=	Will subtract the current value of the variable on the left from the value on the right and will assign the result to the variable name on the left of the operator
^=	Will first conduct a Bitwise Exclusive OR on the current value of the variable from the left by the value on the right and finally assigns the result to the variable on the left.
&=	Will first conduct a Bitwise AND on the current value of the variable from the left by the value on the right and finally assigns the result to the variable on the left.
|=	Will first conduct a Bitwise

	Inclusive OR on the current value of the variable on the left of the operator by the value on the right and then assigns the result to the variable on the left.
>>=	Will first conduct a Right shift AND statement of the current value of the variable on left by the value on the right and then will assign the result to the variable on the left.
<<=	Will first conduct a Left shift AND on the current value of the variable on left by the value on the right and then assigns the result to the variable on the left.

Below this chart, you will find a program that uses all variables in different situations to grab or connect the given results.

```
var name string


func main() {
   name = "jef" // = operator
   print("name is  -> ", name)
   newvar := "new variables" // := operator
   op1 := 40
   op2 := 90
   print("\n", newvar)
   op1 += op2                      // add
   println("op1 += op2 = ", op1) // 130 because 40 +
90 = 130
   op1 -= op2                      // subtract
   println("op1 -= op2 = ", op1) // 140
   op1 *= op2
```


```
    println("op1 *= op2 = ", op1) // 3600
    op1 /= op2
    println("op1 /= op2 = ", op1) // 40
    op1 %= op2
    println("op1 %= op2 = ", op1) // 40
}
```

Data type and format identifier list

Golang has many different data types which all work in different forms and have their own use cases. As many other languages go has different types and categories of data types such as the following

Basic	Basic data types are data types in a programming language that consists of numbers, boolean types, and strings or characters
Reference	Reference data types may consist of type structures, functions, pointers, and slices as well as any other “sub” types like channels.
Aggregate	This type of categorization falls under arrays and structures.
Sub	Other data types or forms that are not listed

Below you will find a list of all data types that the Go programming language uses as well as their categorization and full-length description.

Type name	**Categorization**	**Description**

int	Numeric / basic	Integer
int8	Numeric / basic	8-bit signed integer
int16	Numeric/basic	16-bit signed integer
int32	Numeric/basic	32-bit signed integer
int64	Numeric/basic	64-bit signed integer
uint	Numeric / basic	Unsigned integer
uint8	Numeric / basic	Unsigned 8-bit integer
uint16	Numeric/basic	Unsigned 16-bit integer
uint32	Numeric/basic	Unsigned 32-bit integer
uint64	Numeric/basic	Unsigned 64-bit integer
uintptr	Numeric/basic	Unsigned int pointer
float32	Numeric/basic	Floating point 32-bit integer
float64	Numeric/basic	Floating point 64-bit integer
true	Boolean / basic	Boolean value true
false	Boolean/basic	Boolean value false
string	Character	String or character based value
rune	Character	Int32 representation for Unicode points
complex64	Numeric/advanced	Complex numbers that carry a float32 value as a real

		number and imaginary component
complex128	Numeric / advanced	Same as complex64 but instead of float32 it carries a float64 value

With these data types also comes something commonly known as format identifiers. Format identifiers allow you to print out variables carrying those datatypes using ***printf*** function calls. Below you will find a list of all current base format identifiers within the Go programming language. It may be handy to note that currently this book will be too small to fit every possible combination and format identifier but base ones are more commonly used than more advanced combinations of format identifiers.

Format identifier call	**Type representation**	**Description**
%v	Default data type	Will output the value in a default or standard format
%d	Decimal	Will output the variable in decimal notation
%b	Binary	Will output the variable in binary notation
%o	Octal	Will output the variable in octal formats

%x	Hexidecimal	Will output the variable in hexadecimal format (lower case)
%X	Hexidecimal	Will output the variable in hexadecimal format with upper-case letters
%f	float	Will output the variable in floating point representation
%c	s-char	Will output a variable that is a single character representation
%s	string	Will output the variable for string representation
%t	bool	Will output the variable for boolean based representation
%p	pointer	Will output the variable for pointer-based representation

If you manage to use the wrong data type with ***printf*** statements you may get a weird output. So to work a bit better and save you some googling time, I provided a list of errors that you can receive due to using the wrong data type, wrong placement, etc with the fmt.Printf statements.

- **%!d(string=hi)**: In the case of *Printf("%d", "hi")* this means that the statement is using the wrong format identifier, with the mark '!' telling you it is not the correct identifier. The data type of the variable is thrown to the output to ensure you know what data type the variable returned. To fix this we can replace "hi" which is

the argument to %d with a value that can be a decimal, or we can replace “%d” with “%s” because per output the type of the variable is a string which means we must use “%s” as a format identifier.

- **hi%!(EXTRA string=guys):** This warning tells you in the case of *Printf("hi", "guys")* that there is an extra string, the printf function call requires arguments that consist of your message or output statement and then the argument to the format identifiers. If there is not a format identifier within the first statement argument then any argument after that is considered an extra argument or redundant argument. To fix this we can replace “hi” with %s and replace “guys” with “hi guys”.

- **hi%!d(MISSING)**: In the case of *Printf("hi%d")* this means Missing an argument or secondary variable to format with %d. To fix this we would need to put an argument after the first statement which matches the format identifier.

- **%!(BADWIDTH)hi**: In the case of *Printf("%*s", 4.5, "hi")* this warning is saying that the width value specified in the format string is not an integer. In Go the width of a field in a given output string can be specified using a number following the format symbol `%`. This number to specify the width should be 4 because it is not a floating point number.

- **%!(BADPREC)**: In go’s format identifiers you have something called a precision value for floating point numbers, this tells the output to specify the number of digits that will be displayed after the decimal point. In the case of fmt.Printf("%.*s", 4.5, "hi") we get an error saying BADPREC because that is a bad precision value. This simply means that when using the format identifier we specified a data type that was not an integer but in our case rather a float value. In order to fix this we can call printf with the same arguments but replace 4.5 with 4 as 4.5 is not excepted as a precision value again due to it not being of type integer.

Using extra data types

As you saw above, Go has many data types, format identifiers, and prime details for using and formatting data types. These next few tables and code snippets will give you decent examples of how to use certain

data types, what other data types Go uses, and how to work with format identifiers.

- **func**: In go, there is a keyword which is also known as a data type called func, this keyword defines a function. After you define func in most situations you will need to define the name of the function like the code below.

```
func KeyFunction() {}
```

This keyword is used in many cases such as threading and other examples, however, for now, we are going to skip that as this will be the next section going deeper into how you can use func to create more simple to advanced functions

- **map:** Map is a common data type within the programming realm, in short, a map is a variable that takes an input variable, matches the input variable with a key and outputs the value assigned to the key. For example, you can say a map in computational terms is the same as looking at a physical map and comparing your input such as a mountain to a maps legend that has keys, comparing them to those keys and finding your desired output. Maps are used in many cases and are very very helpful when trying to work with multiple fields. Below you will find the syntax and usage of a map.

```
map[datatype]datatype
```

For example.

```
var Keywords = map[string]int{
   "a": 1,
   "b": 2,
}
```


```
func main() {
   fmt.Println(Keywords["a"])
}
```

The program above will spit out ‘1’ because the letter a matched and was key in the map that was assigned the variable int. The data type variable inside of ‘[ ]’ is the type of the input variable and the data type of the keys while the data type outside of the ‘[ ]’ is the output variable and the value assigned to the keys once they match. Another way to think of maps is a giant list of conditional statements without the use case of constantly using if and else statements.

- **Arrays:** Arrays in the go programming language are typically defined with `[]datatype`. An array can be any data type as long as the array holds values of only those data types. Below you will find multiple examples of arrays in different data types.

```
var Arr_Str = []string{"a", "b", "c", "d", "e", "f"}

var Arr_Int = []int{1, 2, 3, 4, 5, 6, 7}

var Arr_Uint = []uint{0x00, 0x01, 0x02, 0x03, 0x04,
0x05}

var Arr_Rune = []rune{'\n', '\r', '\t', '\x1b'}

var Arr_Float = []float32{1.4, 9999.1000000,
10.9000343}

var Arr_Complex128 = []complex128{complex(3, 4),
complex(20, 90)}
```


- **Interface**: Interface is a data type that often confuses people, however, it might be good to just see this as a data type that is

any data type. For example, creating an array of type *interface* can hold values of int, bool, float, complex, rune, string, map, func, etc.

Interface can be defined as the following

```
var name interface{}
```

An example of using the interface data type can be like so

```
var ArrayInterface = []interface{}{
   1.5,
   "he",
   1,
   true,
   false,
   func(){},
}
```

Notice how the interface array consist of values that are all separate interfaces? That is essentially what an interface is.

- **Type**: The type keyword defines a data type where you can essentially create an alias to a given or current data type which is extremely useful when working with structures or other functions. Using the type keyword works like the following

```
Type Name Datatype
```

Consider the following type alias for an interface array.

```
Type ArrayInterface []interface{}
```


Functions

In Go as defined above we already know that there is a keyword called func, but there are many different ways to use and implement this keyword from standard functions to “anonymous functions”. Below this text, you will find some decent examples using functions which include return types, arguments, type structures, and more with functions.

***Side Note**: It is worth mentioning that every single go program that has a ‘package main’ identifier at the top level of the file should always have a main function defined somewhere.*

- **Simple function usage:** Functions can be simple and wacky sometimes but this may be a good example to show exactly how a function can be defined.

```
func CallFromMain() {
   println("hello world")
}

func main() {
   CallFromMain()
}
```

Func main is where our program starts and once the program is executed we call a function titled call from main with no arguments, pointers, or return values but rather call the function to execute a brick of code which prints out hello world. Simple right?

- **Function arguments:** Function arguments are easy to get down, any argument you want to the function must be typed with the name of the argument followed by its data type like the function below.

```
func CallFromMain(msg string) {
   println(msg)
}

func main() {
   CallFromMain("hello world")
}
```

The argument of type msg is of type string which means our argument must also be a string
Function arguments can also be put in lists like the following

```
func NameAge(name string, age int) {
   println(name+" is ", age)
}

func main() {
   NameAge("bob", 10)
}
```

In the case that you do not want to manually type out string, string, and string over and over again you can put a list of variables followed by that same data type.

```
func NameList(name1, name2, name3, name4 string) {
   var list []string
   list = append(list, name1, name2, name3, name4)
   println(list)
}

func main() {
   NameList("bob", "jef", "marry", "ane")
}
```


This function takes 4 names which are a string because we made a list of variables then declared their data type. We take a list and append it of type string and output it. In the call to the function, we separate the arguments by commas.

- **Function return types:** Functions in Go have the ability to return data, lists of data, or a list of variables.

```
func NameList(name1, name2, name3, name4 string)
(name_list []string) {
   name_list = append(namelist, name1, name2, name3,
name4)
   return namelist
}
```

The function above takes the 4 names as an argument. Still, it predefines the return type as a variable titled name_list which is classified as a string array now we can already use that variable to append our values and use the return keyword to return data.

```
func NameList(name1, name2, name3, name4 string)
string {
   return name1+" "+name2+" "+name3+" "+name4
}
```

In this case the function returns a string but instead of declaring a variable for the return type it actually just shows the data type of the return value which is a single string. You can also separate and return multiple values like so.

```
func Is() (string, int, bool) {
   return "bob", 10, true
}
```


In this case the return values hold the data type string int and bool which means we also must return them in that same order. If we do not return them in the same order the Go compiler will throw an error.

- **Function's with structures:** Functions in go can also use and be called with structures like the following.

```
type User struct {
   Name string
   Age  int
}

func (U *User) GetData() (string, int) {
   return U.Name, U.Age
}

func main() {
   var u User
   println(u.GetData())
}
```

In this case there is a type structure which is assigned the function GetData() that returns the two values in the structure which is called by main. You must declare a variable name according to that structure in order to call data for it and to assign functions you have to place the structure's alias followed by a pointer to the structure name and only then place the function name.

- **Anonymous function calls / function literals:** In Go there is such thing as an anonymous function which you can call without a name or rather use with a name. Typically anonymous functions are used with threading, maps, or other data handlers.

```
func main() {
   func() {
       fmt.Println("hi")
```


```
    }()
}
```

This is a very very simple anonymous function call, when the main brick is called or started func will run under main without requiring to call it. This allows you to put the starter or thread code in a unique spot.

- **Call anonymous functions from variables**

```
func main() {
    value := func() {
        fmt.Println("hi")
    }
    value()
}
```

This code does the same as above but will only execute the anonymous function when it is called under that brick.

- **Anonymous function calls with arguments**

```
func main() {
    func(data string) {
        fmt.Println("Hello -. ", data)
    }("jef")
}
```

This code creates an anonymous function but requires one argument to start and run this function. This function will output ***'Hello -. jef'*** given the argument was *jef.*

- **Anonymous functions as arguments to functions**

```
func NewFunction(name string, age int, varname
func(name string, age int)) {
   value := varname
   value(name, age)
}


func main() {
   createanonymous := func(name string, age int) {
       fmt.Printf("%s is %d years old", name, age)
   }
   NewFunction("jef", 10, createanonymous)
}
```

This function is a good example of taking anonymous functions as function arguments to another argument. The code should be pretty understandable as it is no different than the call made above but rather as an argument.

- **Anonymous function calls in maps:** When working with maps you may want to execute a function from the map with given data so we can execute the following

```
var newmap = map[string]func(){
   "hello!": func() {
       fmt.Println("hello there!")
   },
   "bye!": func() {
       fmt.Println("have a good day!")
   },
}

func main() {
   newmap["hello"]()
}
```


The map in this code asks for an input variable of type string and will return a function. When we input hello it will run a function that outputs the phrase “hello there!” and if we use bye then it will output “have a good day!”. We call this map as a function call given the arguments

- **Returning anonymous functions**

```
func NewFunction() func(name string) string {
   newfunction := func(name string) string {
       return name
   }
   return newfunction
}


func main() {
   NewFunction()("bob")
}
```

This is a bit confusing. Still, we call a function named NewFunction which returns a function that returns a string type. We create the function under *NewFunction* titled *newfunction* and return *newfunction* as the return or output variable when NewFunction is called. When we call this function we use standard parentheses to call it then after those call the argument to that return function.

- **Init func:** The **`*init*`** func is a function keyword that will run before your main brick. This counts as an anonymous function because it can not be called but it can be declared. This function comes helpful when you need to determine operating system information, perform file system checks, integrity checks, and much more.

```
func init() {
     println(“hello world!”)
}
```


```
func main() {
   fmt.Println("generating data…")
}
```

When this program runs you will see the hello world output first before generating data.

Struct and Type

Structures in Go are defined with the type keyword followed by a name followed by the struct keyword. Structures have many use cases in go, whether simply to store small amounts of data or to marshal thousands and thousands of lines of json entries. Below you will find a list of basic structure use cases to advanced use cases like XML, JSON, etc.

- **Basic Structures with the type keyword**

```
type User struct {
   Username  string
   ID        int
   UserAlias  string
   Ammunition int64
   Health     int
}
```

A structure is declared called User, which holds 5 unique values of different data types all of which are local to just that structure and can be access like the following shown below.

```
func main() {
   var U User
   U.Ammunition = 100
   U.Health = 100
```


```
}
```

It may be nice to note that Structures already have pre-defined variables in them and you can not declare *U.Ammunition (in this instance) with :=* to start a new variable as that is an invalid use case to go. You can also declare structs like the following below

```
type User struct {
   Username   string
   ID         int
   UserAlias  string
   Ammunition int64
   Health     int
}

func main() {
   U := User{
       Health:     100,
       Ammunition: 100,
       ID:         os.Getuid(),
   }
   println(U)
}
```

This makes structures extremely flexible to use, however, they have other use cases.

- **JSON structures**

The following structure unmarshals a json file response into values that can be translated per their data type. This program takes a variable called JsonData which will load a small snippet of json data with a value called username which equals john. This value holds the data type of string and will be declared like that in the structure. Then a structure is defined with the json tag to declare it as a json variable along with its json meta tag.

```
package main


import (
    "encoding/json"
    "fmt"
    "log"
)


var JsonData = `
{"username":"John"}
`


type User struct {
    Username string `json:"username"`
}


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


func main() {
    var U User
    x := json.Unmarshal([]byte(JsonData), &U)
    CE(x)
    fmt.Println(U.Username)
}
```


When using the encoding/json package you can use tags of the following in structures to define that data structure as a *“json”* type structure.

```
`json:"VARIABLE OF JSON DATA IN JSON ARRAY"`
```

This helps when you need to unmarshal say a json response when using APIs.

- **XML-based type structures:** These work the same exact way as json structures and have their own individual unique tags to tell the compiler that the given variable is an XML entry. Below you will see an example code brick that implements this with an anonymous function call.

```
package main


import (
    "encoding/xml"
    "fmt"
    "log"
)


type XMLResponse struct {
    XMLName  xml.Name `xml:"userdata"`
    Username string   `xml:"username"`
    ID       int64    `xml:"ID"`
    UUID     string   `xml:"UUID"`
}

var XMLData = `
    <userdata>
        <username>Ripper87</username>
        <ID>673467823674273684678</ID>
        <UUID>434324-24-234-234-4234</UUID>
    </userdata>
`

func CE(x error) {
```


```
    if x != nil {
        log.Fatal(x)
    }
}

var Resp XMLResponse

func init() {
    x:= XML.Unmarshal([]byte(XMLData), &Resp)
    CE(x)
}

func main() {
    fmt.Println(Resp.Username)
}
```

This code should be self-explanatory as it is quite simple and relational to the tags used within json, however, the difference here is in XML.name. When you want to write a call to generate XML code it is a good idea to list each branch or tree similar to JSON but instead, you call XML.name to call the tree's name.

- **Yaml structs:** YAML or Yet Another Markup Language is another structure tag you can use to decode YAML data. This works the same as the XML and JSON tags shown above. However unlike JSON and XML, you have to export an external package for YAML development, so simply run the commands from go's tools to install the following package to add YAML support.

```
gopkg.in/yaml.v3
```

Below you will find an example program that will unmarshal user data into a YAML structure using the GoPKG we installed.

```
package main

import (
    "fmt"
    "log"


    "gopkg.in/yaml.v3"
)


type UserData struct {
    Username string `yaml:"user"`
    UUID     string `yaml:"uuid"`
}


var YamlData = `
    user: User123
    uuid: aaa-aaa-aaaa-aaaa-aaaa
`


func main() {
    var U UserData
    x := yaml.Unmarshal([]byte(YamlData), &U)
    if x != nil {
        log.Fatal(x)
    }
    fmt.Println(U.UUID + " Has username " +
U.Username)
}
```

- **The type keyword:** A keyword called type, as you may know, exists in Go. This keyword can be used in two ways which include defining an alias to an existing data type such as *string, uint, uint8, uint16, uint32,* and so on or you can use it to as

shown above define data structures. Here is how we use the type to create an alias for the integer62 data type.

```
type Integer int
var A Integer = 5
func main() {
   fmt.Printf("A = %v | Type of a is = %T\n", A, A)
}
```

This program will output a text that says the value of A and that the data type of A is ***main.Integer*** given our package name is main and we used our type called Integer to represent an int value.

- **Structure Composition:** Go has a lot of various things you can use the struct keyword for and one of those includes a type called Structure Composition which is used to compose values from another structure.

```
type UserData struct {
   Username string
   Stats    UserStats
}


type UserStats struct {
   Health     int
   Ammunition int
   Inventory  UserInventory
}


type UserInventory struct {
   Weapon1 string
   Weapon2 string
   Loot    []interface{}
```


```
}


var (
   Udata UserData
)


func init() {
   Udata = UserData{
       Username: "User1",
       Stats: UserStats{
           Health:     100,
           Ammunition: 100,
           Inventory: UserInventory{
               Weapon1: "Shotgun",
               Weapon2: "Rifle",
               Loot:
append(Udata.Stats.Inventory.Loot, "SnackBar"),
           },
       },
   }
}


func main() {
   println("User data is as follows")
   println("------------------------")
   println("Username        : ", Udata.Username)
   println("User Health     : ", Udata.Stats.Health)
   println("User Ammunition : ",
Udata.Stats.Ammunition)
   println("User Weapon1    : ",
Udata.Stats.Inventory.Weapon1)
   println("User Weapon2    : ",
Udata.Stats.Inventory.Weapon2)
   println("User Inventory  : ",
Udata.Stats.Inventory.Loot[0].(string))
```


```
}
```

This is a very big program but you should get the idea, the init function does nothing different asides assigning values. However, the structures are a bit different. By connecting these structures together we can use the values and assign values and pull or modify values much easier. You may notice once we make a call to the Loot that we do a little bit of type converting. This is because the Inventory loot variable is a variable of type interface array which means it can hold any data type in the array and we want to ensure that we can physically view that. If we just call the first value in the inventory (0) without type converting it we get the following output.

```
(0x45a2a0,0x476e38)
```

So it is a good idea to convert these to a string which will yield the following output.

```
User Inventory:  SnackBar
```

Because we appended the word snack bar to the inventory of our player when the compiler ran init() before main.

- **Receivers:** In go you may find yourself requiring the use of receivers which allow a function to use a type structure without actually calling a variable.

```
package main

type UserData struct {
   Username string
   Age      int
}
```


```
func (U *UserData) HoldUserData() {
   U.Age = 19
   U.Username = "jef"
}


func main() {
   var User UserData
   User.HoldUserData()
   println(User.Username)
}
```

This is quite helpful when we do not want to place init functions or need to hold data locally or use the same var name to hold the user data. As you can see it is quite simple.

- **Embedded structures:** Embedded structures are really nice as well and similar to Structure Composition but not in the same way you think.

```
type UserData struct {
   UserInfo
   Username string
}


type UserInfo struct {
   Age int
}


func main() {
   var User UserData
   User.Age = 10
   User.Username = "Hello134!"
   println(User.Username)
```


```
}
```

Pretty simple as well and comes in handy in more advanced use cases!

Make and New keywords

Go has a very unique way of creating new methods and variables, one of those ways is by using keywords make and new.

- **Make:** Make is used to create new variables of type array, map, and chan which allows you to specify either the length and size of the array, the keys in a map, or the values in a channel. Make is used specifically for those types and should not be used with any other data type as they can create bugs and may even operate weirdly.

```
func main() {
   newarr := make([]string, 10, 20)
   println(newarr)
}
```

This array is created and made with the data type string array, it holds 2 extra arguments 10 and 20. 10 is the length of the array and 20 is the capacity of the array. The next brick shows how to use make to create values with maps.

```
func main() {
   GroupedAssign := make(map[string]func())
   GroupedAssign["key1"] = func() {
       println("hello world")
   }
   GroupedAssign["key2"] = func() {
       println("Hello 2")
   }
}
```


This is pretty simple as well and obvious to guess even if you have 0 experience. When we execute the variable GroupedAssign with ‘key2’ it will run an anonymous function that runs and outputs ‘*Hello 2’*. The next is an example of channel usage

```
func main() {
   channel := make(chan int)
   println(channel)
}
```

- **New:** This function can be used with other data types and is a much safer alternative to allocating memory for different types of data structures. It is important to note that there is a major difference between the zero value of a type and a properly initialized value of that type, as the zero value may not be suitable for all use cases.

```
package main

type User struct {
   Name string
   Age  int
}


func main() {
   p := new(User)
   *&p.Age = 19
   fmt.Println(*&p.Age)
}
```

It might be important to note that it may be better to just declare variables by default rather than using new ones. The new keyword should only be used for specific standards and specific use cases.

For Loops

- **Standard for loop:** For loops in go are pretty simple in their own way as they have multiple keywords such as range which are helpful when ranging over arrays. Below is a very basic sample that will output values in a string array by ranging over it.

```
package main

import "fmt"

var Arr = []string{"a", "b", "c", "d", "e", "f"}

func main() {
   for i, k := range Arr {
       fmt.Printf("index %v -> %s \n", i, k)
   }
}
```

When iterating over an array or slice in Go using the '***range'*** keyword, you can use the first variable to represent the current index of the array/slice and the second variable to represent the value at that index. In this case, k would represent the current value and i would represent the current index.

- **For loops (C-Style):** Go also has another way you can work with for loops, and that goes by declaring multiple variables and conditions like so.

```
package main

var Arr = []string{"a", "b", "c", "d", "e", "f"}
```


```
func main() {
   for i := 0; i < len(Arr); i++ {
       println(Arr[i])
   }
}
```

This is also pretty English like. For I which is equal to 0 and i is less than the length of the array increment I (++) by one then prints out the value of the array's index.

- **Infinite loops:** Infinite loops are quite simple and act like while(1)/while(true) loops

```
package main

func main() {
    for {
        println("hi")
    }
}
```


Concurrency

Concurrency in programming is one of the most important things to know about and understand, especially for a hacker since they most likely need to discover hosts or write exploit programs which require more than one process thread. Luckily Go has a fantastic solution for threading!

- **go:** The `go` keyword is a keyword that starts a go routine in go and its standard implementation is pretty simple. Below you will see a program that starts a counter as a for loop is run under a main function.

```
package main
```

```
import "time"

var c int
func Count() {
   for {
       time.Sleep(1 * time.Second)
       c++
       println(c)
   }
}


func main() {
   go Count()
   for {
       time.Sleep(1 * time.Second)
       println("At for loop")
   }
}
```

In short the go keyword starts an interactive goroutine.

- **Goroutine** ***(term)***: A goroutine is a term often used to represent a lightweight execution of a thread in Go. Goroutines are often recognized as a different form of thread because of how they work and how lightweight and usable they are for certain operations. Unlike normal threads which require a whole stack of memory to themselves, Goroutines are multiplexed into a smaller number of operating system threads which allows much more of them to run within a single process.

- **Channels**: Channels in go are a very intelligent way of communicating with threads, they can exchange data back and forth in certain contexts and also even be used to transmit and send out data. Below is a core example of how channels can be used

```
package main

import "time"

func Counter(channel chan int) {
   for k := 0; k < 10; k++ {
       channel <- k
       time.Sleep(1 * time.Second)
   }
   close(channel)
}


func main() {
   channel := make(chan int)
   go Counter(channel)
   for n := range channel {
       println("Numbers -> ", n)
   }
}
```

We define a function called Counter which takes one argument called channel followed by two keywords chan and int. When using the chan keyword you must specify the data type that the channel will hold as it can be any data type. Then we start a simple for loop and tell the channel to hold the value of k each time k is counted up using <- which will store the variable in the channel. Once done we close the channel after the function is done and start our main brick. Using make to create the channel we start a thread followed by the channel created name and then the range of the channel for each number. When you run the program every second a new number will be outputted to the screen 1-10 until 10 seconds have passed.

- **Select:** Select is used to choose from multiple channels and check the states of the channels. The select statement will also block until one of the channels is ready to be used which allows for thread management.

```
package main

import "time"


func W1(channel chan string) {
   time.Sleep(1 * time.Second)
   channel <- "w1 messaage"
}


func W2(channel chan string) {
   time.Sleep(2 * time.Second)
   channel <- "w2 message"
}


func main() {
   chan1 := make(chan string)
   chan2 := make(chan string)
   go W1(chan1)
   go W2(chan2)
   for i := 0; i < 2; i++ {
       select {
       case message1 := <-chan1:
           println(message1)
       case message2 := <-chan2:
           println(message2)
       }
   }
}
```