

Propuesta de un Sistema Anti-Keylogger para proteger despliegues y configuraciones de servicios cloud

Juan Diego Castellanos Jerez

Facultad Ingeniería de Sistemas, Universidad Santo Tomás
Tunja, Colombia

juancastellanosj@usantotomas.edu.co

RESUMEN

Los keyloggers representan una amenaza considerable para la integridad de los sistemas y la confidencialidad de los datos sensibles, al registrar de manera encubierta las pulsaciones de teclado. Con el objetivo de contrarrestar esta vulnerabilidad, se ha realizado una propuesta de un sistema anti-Keylogger básico como resultado del taller “planteamiento de buenas prácticas de seguridad para entornos cloud e IoT”, elaborado en el espacio académico Seguridad de Servicios cloud e IoT, impartido en la Especialización “Gestión de Servicios de Tecnologías de la Información”, de la Universidad Santo Tomás. En este trabajo se aborda la implementación de un sistema básico diseñado específicamente para fortalecer la seguridad durante los procesos de despliegue y configuración de servicios cloud. Se llevó a cabo una investigación amplia y exhaustiva para respaldar el enfoque de este trabajo, analizando en detalle los diferentes tipos de keyloggers y cómo operan. Durante el análisis, se descubrió que, en la mayoría de las conexiones a máquinas virtuales provistas para servicios IaaS, pueden surgir vulnerabilidades, originadas principalmente por la ausencia de protección contra keyloggers. La arquitectura de este sistema anti-Keylogger se integra de manera fluida en los sistemas operativos linux más utilizados para servicios IaaS. Su eficacia y versatilidad lo convierten en una herramienta imprescindible para garantizar la seguridad de los datos confidenciales en entornos de servicios cloud. A través de este artículo, se pretende proporcionar una visión completa de las funcionalidades y ventajas de este software, así como fomentar su adopción en la comunidad de usuarios de servicios cloud.

Palabras Clave: Anti-Keylogger, Keylogger, Linux, registro de teclas, seguridad, servicios de la nube, servidores, software, teclado, vulnerabilidad.

ABSTRACT— Keyloggers represent a considerable threat to the integrity of systems and the confidentiality of sensitive data by covertly recording keystrokes. To counteract this vulnerability, a proposal for a basic anti-keylogger system has been made because of the workshop "approach of good security practices for cloud and IoT environments", developed in the academic space Security of cloud and IoT Services, taught in the Specialization "Management of Information Technology Services", at the University of Santo Tomas. It uses basic techniques for monitoring keyboard events, detection of suspicious background processes, encryption of keystrokes in sensitive information transmission environments. This work addresses the implementation of a basic system specifically designed to strengthen security during the processes of deployment and configuration of cloud services. Extensive and thorough research was conducted to support the focus of this work, analyzing in detail the different types of keyloggers and how they operate. During the analysis, it was discovered that, in most of the connections to virtual machines provided for IaaS services, vulnerabilities can arise, mainly originating from the absence of keylogger protection. The architecture of

this anti-keylogger system integrates seamlessly into the most used Linux operating systems for IaaS services. Its effectiveness and versatility make it an essential tool for ensuring the security of confidential data in cloud service environments. This article aims to provide a complete overview of the functionalities and advantages of this software, as well as to encourage its adoption in the community of cloud services users.

Keywords: keylogger, anti-Keylogger, cloud services, Linux, servers, security, software, vulnerabilities, keyboard, key register.

I. INTRODUCCIÓN

La seguridad en los procesos de despliegue y configuración de servicios cloud es de vital importancia para salvaguardar la integridad de los sistemas y la confidencialidad de los datos sensibles [1]. Una amenaza considerable en este contexto son los keyloggers, los cuales registran de manera encubierta las pulsaciones de teclado, poniendo en riesgo la seguridad de la información [2].

Para contrarrestar esta vulnerabilidad, se ha planteado un software piloto anti-Keylogger que utiliza técnicas de monitoreo de eventos de teclado y análisis de procesos en ejecución. La arquitectura de este software es flexible, ofreciendo una capa adicional de protección durante los procesos de despliegue y configuración. Los resultados obtenidos han demostrado una capacidad exitosa para encriptar las teclas y determinar grupos de caracteres sensibles, lo que representa una mejora significativa en comparación con las soluciones existentes en el mercado [3].

El objetivo principal de este estudio es proponer una solución efectiva para reforzar la seguridad en los entornos cloud, especialmente en lo que respecta a la salvaguardia de credenciales y datos sensibles. La implementación de un software anti-Keylogger se presenta como una medida esencial para asegurar la integridad de la información y proteger los sistemas contra posibles amenazas.

En consecuencia, este artículo explora en detalle la funcionalidad y las características de este software piloto, así como su relevancia en el contexto actual de seguridad informática. Además, se discuten las aplicaciones prácticas de esta solución y se subraya su importancia en la protección de la información sensible en los procesos de despliegue y configuración de servicios cloud. A través de este estudio, se pretende contribuir al fortalecimiento de la seguridad en dichos entornos, proporcionando a los usuarios una herramienta confiable y efectiva para combatir los keyloggers y preservar la integridad de sus sistemas.

II. ESTADO DEL ARTE

Actualmente, la detección de keyloggers es tema de interés y caso de estudio para las empresas que distribuyen y crean software antivirus. Los keyloggers son programas maliciosos diseñados para registrar y grabar las pulsaciones de teclado de un usuario en un dispositivo, sin su consentimiento [4].

Estas herramientas se utilizan principalmente con fines ilícitos, como el robo de información confidencial, contraseñas, datos financieros y otros detalles sensibles.

La presencia de keyloggers en un sistema puede tener graves repercusiones económicas para individuos y organizaciones. En primer lugar, los keyloggers pueden comprometer la seguridad de las transacciones financieras en línea. Al registrar las pulsaciones de teclado, los atacantes pueden obtener información como números de tarjetas de crédito, contraseñas bancarias y otros datos de acceso. Esto les permite realizar transacciones no autorizadas y acceder a cuentas bancarias, lo que resulta en la pérdida directa de dinero para las víctimas [5].

Además, los keyloggers pueden ser utilizados para el robo de identidad. Al obtener información personal, como nombres, direcciones, números de seguridad social, los atacantes pueden realizar actividades fraudulentas en nombre de las víctimas. Esto puede incluir la apertura de cuentas bancarias, solicitar créditos y realizar compras en línea, generando deudas y pérdidas financieras significativas para las personas afectadas. Asimismo, los keyloggers también pueden ser utilizados para el espionaje empresarial [6]. Al registrar las actividades y las pulsaciones de teclado de los empleados, los atacantes pueden obtener información confidencial, como secretos comerciales, estrategias de negocios y detalles de proyectos. Esta información puede ser vendida a competidores o utilizada para obtener ventajas injustas en el mercado, causando pérdidas económicas considerables para las empresas afectadas.

Los keyloggers pueden clasificarse en varios tipos según su naturaleza y método de funcionamiento. Algunos de los tipos comunes incluyen:

a) Keyloggers de hardware: Estos keyloggers se basan en dispositivos físicos que se conectan entre el teclado y el sistema objetivo. Registran y almacenan las pulsaciones de teclas antes de enviarlas al sistema. Ejemplos de keyloggers de hardware incluyen dispositivos USB y tarjetas de interfaz de teclado [7].

b) Keyloggers de software: Son programas o aplicaciones instalados en el sistema objetivo. Estos keyloggers pueden estar diseñados para ejecutarse en segundo plano sin el conocimiento del usuario y registrar todas las pulsaciones de teclas realizadas. Pueden tener características adicionales, como la captura de capturas de pantalla o el seguimiento de la actividad del portapapeles [8].

c) Keyloggers basados en kernel: Estos keyloggers operan a nivel del kernel del sistema operativo. Capturan las pulsaciones de teclas antes de que sean procesadas por las aplicaciones o procesos en el sistema. Esto les permite ser más sigilosos y difíciles de detectar [9].

d) Keyloggers basados en aplicaciones: Estos keyloggers funcionan como complementos o extensiones de aplicaciones específicas. Se integran en programas populares y capturan las pulsaciones de teclas mientras se utilizan esas aplicaciones [10].

En la Figura 1 se evidencia la interferencia de un Keylogger en la captura de credenciales del usuario, este Keylogger es de tipo kernel porque captura las teclas antes del procesamiento o envío.

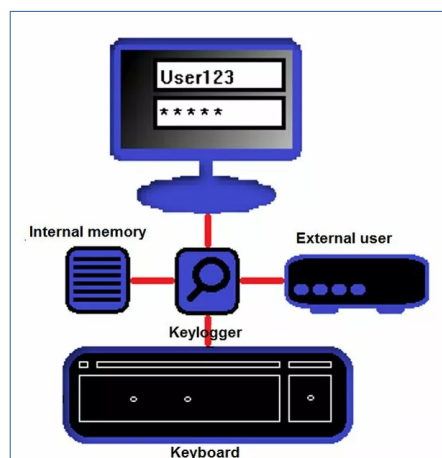


Figura 1. Funcionamiento de un Keylogger [11]

Un ejemplo de las pérdidas económicas ocasionadas por los keyloggers se puede encontrar en un informe publicado por la empresa de seguridad informática Kaspersky Lab. Según el informe, los ataques de keyloggers en el sector bancario en 2019 generaron pérdidas de más de 7,12 millones de dólares [12]. Estas cifras demuestran el impacto financiero significativo que pueden tener los keyloggers en las instituciones financieras y en los usuarios individuales.

Una vez que se ha implementado el mecanismo de infección, el diseñador del Keylogger se enfocará en su ejecución. La mayoría de los keyloggers utilizan una técnica común de ejecución conocida como "hooking", aunque cada Keylogger la implementará de manera diferente dependiendo del contexto para el cual se necesita el Keylogger [13].

El objetivo básico del hooking es interceptar el flujo de control normal y alterar la información devuelta por una rutina del sistema objetivo. Los ganchos (hooks) pueden ser implementados en cualquier nivel del sistema operativo para la mayoría de las funciones, lo que los convierte en una técnica general utilizada por los desarrolladores de keyloggers.

En la Figura 2 se presenta el funcionamiento de un Keylogger en entornos de despliegues y configuraciones cloud, el usuario utiliza su dispositivo para realizar despliegues o configuraciones de servicios alojados en cloud por lo que digita credenciales y comandos para realizar dichas operaciones. En ese contexto el atacante puede obtener todas las pulsaciones de teclas mediante el Keylogger y finalmente al unir las pulsaciones obtiene la secuencia de palabras formada. Con la secuencia de palabras obtiene información de carácter sensible.

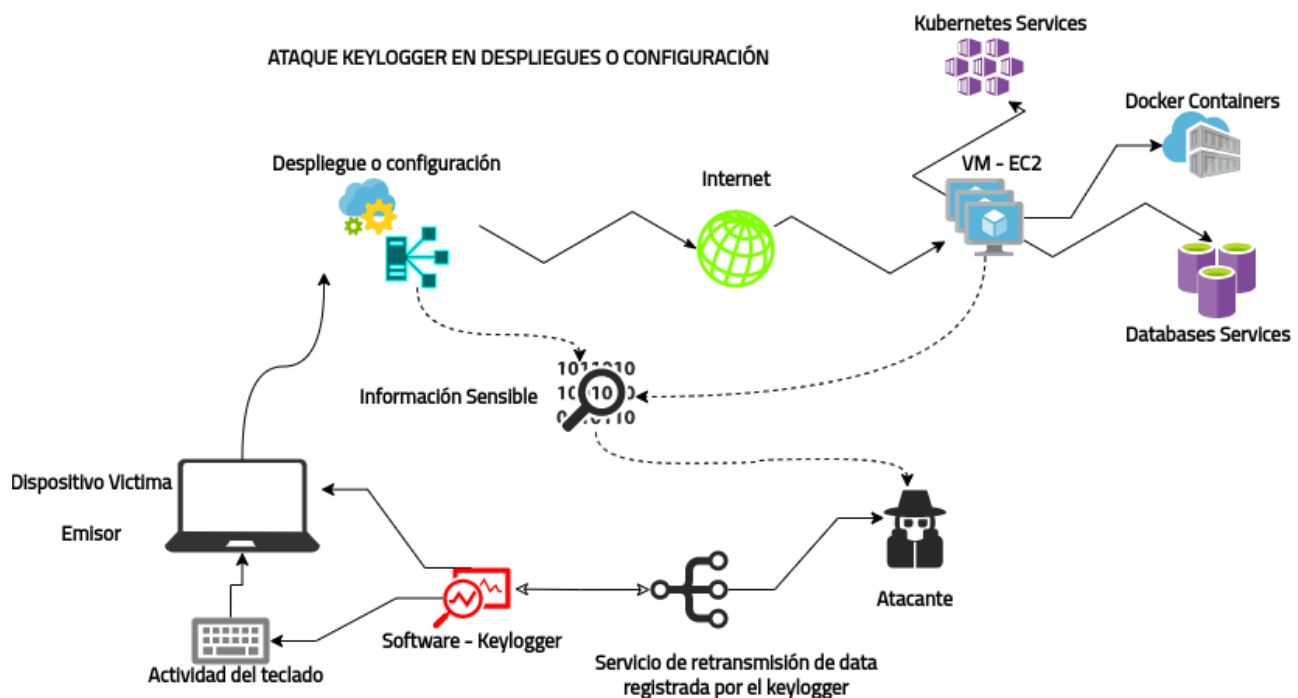


Figura 2. Ataque Keylogger en despliegues o configuración. Elaboración propia

A pesar de Azure incluye un plan de Seguridad para cloud [14], no cubre totalmente la seguridad del entorno, ya que un Keylogger puede ser filtrado directamente al dispositivo del usuario y así obtener información antes del almacenamiento o ejecución, como se evidencia en la Figura 2 donde el atacante obtiene los datos antes de estos ser enviados a cloud.

Ahora bien, conociendo el contexto del funcionamiento y estructura de un Keylogger, se consideraron las siguientes prácticas para la implementación del anti-Keylogger:

- **Detección de patrones:** Los anti-Keyloggers utilizan algoritmos y bases de datos actualizadas para buscar patrones conocidos de comportamiento de keyloggers [15]. Pueden analizar los procesos en ejecución, los registros del sistema y otros indicadores para identificar actividades sospechosas relacionadas con keyloggers.
- **Análisis de comportamiento:** Además de buscar patrones específicos, los anti-Keyloggers también realizan un análisis del comportamiento del sistema [16]. Monitorean las actividades de las aplicaciones y los procesos en tiempo real para detectar comportamientos inusuales, como la captura de pulsaciones de teclado sin consentimiento.
- **Protección en tiempo real:** Los anti-Keyloggers suelen funcionar en segundo plano de manera continua, monitoreando las actividades del sistema y proporcionando protección en tiempo real. Pueden bloquear la ejecución de keyloggers conocidos y advertir al usuario sobre posibles amenazas [17].
- **Protección de entrada:** Algunos anti-Keyloggers también ofrecen características adicionales de protección de entrada. Estas herramientas pueden incluir teclados virtuales seguros que evitan que los keyloggers registren las pulsaciones de teclado físicas, o sugieren el uso de teclados táctiles para utilizar técnicas de ofuscación para dificultar la detección de las pulsaciones [18].

En el área de investigación sobre seguridad informática, se ha prestado considerable atención al estudio del comportamiento de los keyloggers y a la búsqueda de soluciones efectivas para detectar y prevenir sus ataques [19]. En varios trabajos científicos, se han propuesto enfoques innovadores que abordan esta problemática desde diferentes perspectivas. Por ejemplo, Zhang, Zhuang y Zhou, desarrollaron un método basado en el reconocimiento de patrones de las emisiones acústicas del teclado para detectar keyloggers. Su enfoque se basa en capturar las vibraciones acústicas generadas por las teclas al ser presionadas y analizar los patrones de sonido para identificar anomalías que puedan indicar la presencia de un Keylogger. Su investigación, presentada en el Symposium on Security and Privacy (S&P) de 2019, demostró una alta precisión en la detección de keyloggers utilizando este enfoque [15].

Por otro lado, las investigaciones de Monroe, Reiter y Wetzel, dieron como resultado KeyTrac [16], sistema avanzado para derrotar keyloggers avanzados. KeyTrac utiliza técnicas de aprendizaje automático y detección de anomalías para identificar patrones de entrada sospechosos que podrían indicar la presencia de un Keylogger. El sistema aprende los patrones normales de pulsación de un usuario a lo largo del tiempo y construye un perfil de comportamiento personalizado. Luego, compara las pulsaciones de teclas en tiempo real con este perfil para detectar cualquier desviación o actividad anómala.

Teniendo en cuenta la existencia de sistemas como KeyTrac, es importante aclarar a la categoría que pertenece, ya que dependiendo de eso va a lograr detener cierta cantidad de vulnerabilidades o de ataques. Al ser un sistema que se encuentra al nivel de sistema operativo ofrece determinada cobertura, sin embargo, cubrir o proteger despliegues y configuraciones cloud es incierto. Oh, Kang y Cho [17] diseñaron e implementaron un sistema anti-Keylogger efectivo que se centra en la detección y prevención temprana de keyloggers. Su enfoque se basa en técnicas de análisis de

comportamiento y heurísticas para identificar actividades sospechosas relacionadas con la captura de pulsaciones de teclas.

Ma, Peinado y Tygar evaluaron la eficacia de la entrada segura a través de pantallas táctiles en la presencia de keyloggers. Su objetivo fue determinar si las pantallas táctiles pueden proporcionar una medida de protección contra los ataques de keyloggers al evitar la captura de pulsaciones de teclas. Los resultados del análisis realizado por Ma, Peinado y Tygar revelaron que la entrada a través de pantallas táctiles puede ser efectiva para mitigar los ataques de keyloggers. En comparación con los teclados convencionales, las pantallas táctiles presentaron una mayor resistencia a los keyloggers, ya que no hay teclas físicas que puedan ser registradas por los dispositivos de captura. Además, se observó que la entrada táctil puede dificultar el registro preciso de las pulsaciones de teclas por parte de los keyloggers. La variabilidad en la ubicación y el tamaño de las teclas en las pantallas táctiles dificulta la reconstrucción precisa de las pulsaciones, lo que dificulta la obtención de información confidencial por parte de los atacantes [18].

Con base en el contexto anterior, se puede observar que los avances y esfuerzos realizados en el campo de la protección contra ataques de keyloggers son extensos. Sin embargo, es importante destacar que ningún sistema anti-Keylogger logra proporcionar una protección total del 100%. Los sistemas anti-Keyloggers están diseñados o ajustados para detectar y proteger contra ciertos tipos de keyloggers. Sin embargo, con el tiempo, pueden surgir nuevos tipos de keyloggers que pueden evadir las medidas de protección existentes, lo que puede llevar a que las herramientas anti-Keyloggers se vuelvan obsoletas.

La protección de una terminal es de vital importancia, ya que a través de ella se llevan a cabo diversas configuraciones y tareas fundamentales en servicios como MySQL, PHP, PostgreSQL, servidores web como Apache o Nginx, orquestadores como Kubernetes [20] y gestores de contenedores como Docker [21]. Además, se utilizan editores de texto como Vim, Vi y Nano dentro de la terminal. Cuando se realizan despliegues en la nube, configuraciones en servidores o se crean y modifican aplicaciones en contenedores, es crucial tener en cuenta que estas operaciones se llevan a cabo principalmente mediante la terminal [22].

La terminal es la puerta de entrada a numerosas funciones críticas y datos sensibles, como contraseñas y credenciales de acceso. Por lo tanto, garantizar la seguridad y protección de la terminal se vuelve imprescindible para salvaguardar la integridad y confidencialidad de la información, y prevenir posibles fugas de datos. Es fundamental implementar medidas de seguridad adecuadas para la terminal o cmd, como sistemas anti-Keylogger, con el objetivo de mitigar los riesgos asociados a la captura de pulsaciones de teclas y a los ataques de keyloggers. De igual manera, para proteger todo el sistema operativo es importante incluir reglas de protección de archivos, para que los keyloggers no se filtren. Estas medidas de seguridad ayudan a proteger los datos sensibles ingresados en la terminal o en una aplicación del sistema, asegurando que las configuraciones, modificaciones y acciones realizadas a través de ella estén completamente seguras y libres de amenazas [23].

En síntesis, a pesar de los extensos desarrollos y trabajos realizados en el campo de la protección contra keyloggers, es importante reconocer que la protección total no es alcanzada por la mayoría de los sistemas anti-Keylogger. Las terminales, o consolas, son especialmente vulnerables, ya que muchos sistemas de protección no están diseñados para abordar específicamente las necesidades de seguridad de las aplicaciones en estas terminales.

III. METODOLOGÍA

1. Análisis

Para el planteamiento y desarrollo del sistema piloto anti-keylogger primero se hizo una búsqueda y un análisis de las amenazas existentes, así mismo se identificaron algunas arquitecturas y funcionalidades básicas de estos softwares maliciosos. En esta etapa también se realizó la evaluación de soluciones existentes, se investigaron las técnicas y enfoques utilizados relativamente.

En la Figura 3 se presenta el flujo de procesos cuando un usuario utiliza la terminal para digitar y ejecutar comandos.

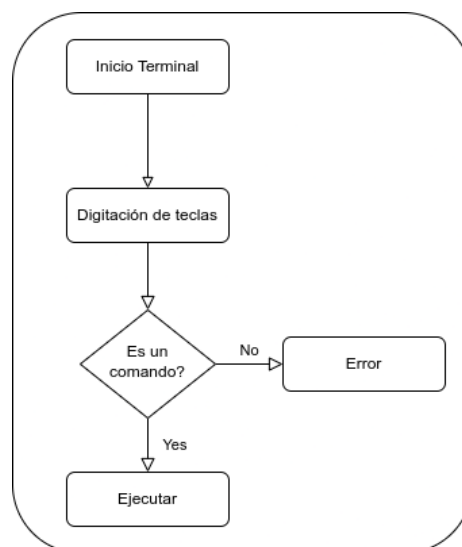


Figura 3. Flujo de procesos uso de terminal. Elaboración propia

De acuerdo a lo anterior se han encontrado las siguientes funcionalidades que se deben implementar para esta primera versión prototipo del sistema anti-keylogger.

- Escaneo del proceso en primer plano
- Escaneo de teclado
- Obtención de eventos de teclado
- Obtención del código correspondiente a cada tecla
- Obtención de nombre de teclas
- Obtención de cadenas sensibles
- Encriptación de teclas y guardado en log

Dado que el contexto es asegurar los despliegues y configuraciones de servicios cloud, se prioriza que el software anti-Keylogger monitoree aplicaciones específicas como son la terminal de comandos y los editores de texto. La Figura 4 expone la arquitectura general en la que el software anti-Keylogger tomará acción, en la parte donde se relaciona el software, la actividad del teclado y el dispositivo del usuario se define esa relación como el monitoreo. Particularmente para el despliegues y configuraciones de servicios, se monitoreará la terminal de comandos.

En la siguiente Figura 4 se expone el contexto en donde el programa anti-keylogger se encontrará, es básicamente similar al funcionamiento de un Keylogger, pero su funcionamiento será invertido para proteger los ingresos del teclado.

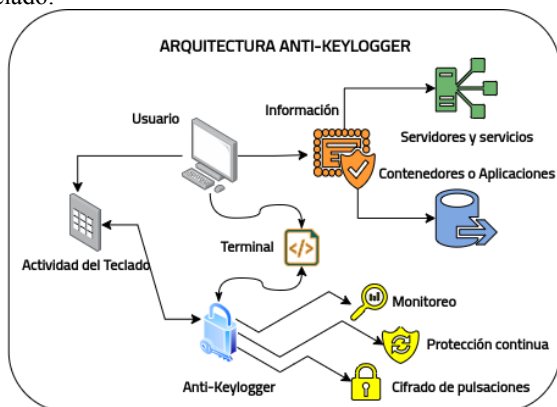


Figura 4. Arquitectura anti-Keylogger. Elaboración propia.

De acuerdo a la Figura 4 se construirá un programa o algoritmo anti-keylogger para encriptar las teclas e indicar si hay cadenas de texto sensible y así mantener un monitoreo constante asegurando las digitaciones de teclas.

2. Diseño

Con base en las funcionalidades mencionadas anteriormente se plantea un posible diseño o arquitectura del sistema, en la Figura 5 se puede observar cada una de las funciones y como se relacionan.

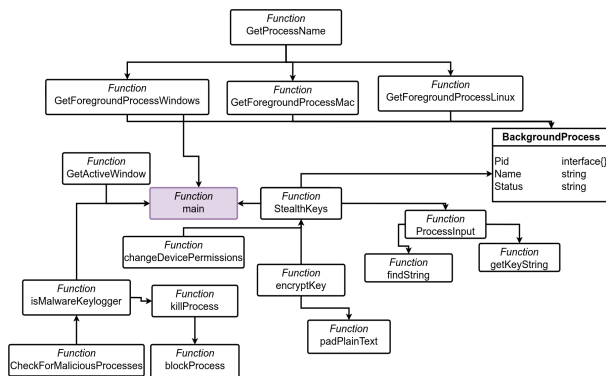


Figura 5. Arquitectura y diseño de funcionalidades. Elaboración propia

3. Desarrollo

Se plantea el desarrollo de un programa anti-Keylogger utilizando el lenguaje de programación Go (Golang). Go [24] es un lenguaje de programación moderno y eficiente que ofrece una amplia gama de herramientas y bibliotecas para el desarrollo de software.

Teniendo en cuenta la Figura 4 se desarrollarán cada una de las funcionalidades como funciones o métodos independientes, no se utilizará clases o estructuras, ya que no se requiere mantener diferentes instancias, el desarrollo de esta primera versión se construirá con funciones, ya que se necesita rapidez y facilidad de modificación, por otro lado, se pueden implementar concurrencia. Finalmente, se procedió a

programar cada función, primero se desarrolla la función `encryptKey()` que recibe por parámetro una llave y un arreglo de bytes, en esta función se cifra el arreglo de bytes que representa cada tecla ingresada, en la Figura 6 se presenta la función.

```
func encryptKey(key, plaintext []byte) []byte {
    // Se genera un bloque de cifrado usando la llave proporcionada
    block, err := aes.NewCipher(key)
    if err != nil {
        panic(err)
    }
    // Se agrega un relleno con espacio adicional si el texto no
    // coincide con el tamaño del bloque de cifrado
    paddedText := padPlainText(plaintext, block.BlockSize())
    // Se crea un vector de inicialización aleatorio
    iv := make([]byte, aes.BlockSize)

    if _, err := rand.Read(iv); err != nil {
        panic(err)
    }

    // Se cifra el texto plano usando la llave y el vector de inicialización
    cbc := cipher.NewCBCEncrypter(block, iv)
    ciphertext := make([]byte, len(paddedText))
    cbc.CryptBlocks(ciphertext, paddedText)

    // Se concatena el vector de inicialización con el texto cifrado
    result := append(ciphertext, iv...)

    return result
}
```

Figura 6. Función `encryptKey`. Elaboración propia

La Función mencionada anteriormente utiliza otra función que se encarga de generar un relleno al texto plano el cual se utiliza para ser encriptado después, en la Figura 7 se presenta la función `padPlainText`

```
// Función auxiliar para agregar relleno al texto plano
func padPlainText(plaintext []byte, blocksize int) []byte {
    padding := blocksize - len(plaintext)%blocksize
    padtext := bytes.Repeat([]byte{byte(padding)}, padding)
    return append(plaintext, padtext...)
}
```

Figura 7. Función para agregar relleno al texto. Elaboración propia

Una vez terminada esta función que encripta cualquier texto plano que ingrese en los formatos indicados, se procedió a programar la función que captura, encripta y guarda las teclas que son oprimidas, también verifica las posibilidades de coincidencia de cadenas sensibles, llamando a otra función utilizando concurrencia para que se ejecute en otro proceso. En la Figura 8, 9 y 10 se presenta la función `StealthKeys`.

```
func StealthKeys(chan chan int, targetPID interface{}) {
    file, err := os.Create("output.txt")
    if err != nil {
        return
    }

    file_enc, err := os.Create("encrypt.txt")
    if err != nil {
        return
    }

    defer file.Close()
    defer file_enc.Close()

    writer := bufio.NewWriter(file)
    writer_enc := bufio.NewWriter(file_enc)

    keyChannel := make(chan uint16)
    go ProcessInput(keyChannel, SensitiveStrings, SimpleSensitiveStrings)

    devicePath := "/dev/input/event3"
    changeDevicePermissions(devicePath)

    dev, err := evdev.Open(devicePath)
    if err != nil {
        log.Fatal(err)
    }
    defer dev.File.Close()

    fmt.Println("Capturando pulsaciones de teclas...")
}
```

Figura 8. Función `StealthKeys`. Elaboración Propia

```

fmt.Println("Capturando pulsaciones de teclas...")

for {
    events, err := dev.Read()
    if err != nil {
        log.Fatal(err)
    }

    for _, event := range events {
        // Solo captura pulsaciones de teclas (evento de presionar)
        if event.Type == evdev.EV_KEY && event.Value == 1 {
            keyName := getKeyString(event.Code)
            if keyName == "Space" {
                keyName = " "
            }
            if keyName == "Enter" {
                keyName = fmt.Sprintf("\n")
            }
            if keyName == "Backspace" {
                fmt.Print("")
                continue
            }
            // Enviar el código de tecla al canal
            keyChannel <- event.Code

            key := []byte("GoStalthKeys") // Define la clave para el cifrado
            // Ajustar la longitud de la clave a 16 bytes (128 bits)

```

Figura 9. Continuación de Función StealthKeys. Elaboración Propia

```

// Ajustar la longitud de la clave a 16 bytes (128 bits)
hash := sha256.Sum256(key)
key = hash[:16]
plaintext := []byte(keyName) // Convierte la tecla en un slice de bytes
ciphertext := encryptKey(key, plaintext) // Llama a la función encryptKey para cifrar la tecla
//fmt.Printf("%s", ciphertext) // Imprime el texto cifrado en hexadecimal
// Escribir el texto cifrado en el archivo encrypt.txt
fmt.Fprintf(writer_enc, "%s\n", ciphertext)
fmt.Printf("%v", keyName)
// Escribir el texto original en el archivo output.txt
fmt.Fprintf(writer, "%v", keyName)
}
err = writer.Flush()
if err != nil {
    panic(err)
}
err = writer_enc.Flush()
if err != nil {
    panic(err)
}
}

```

Figura 10. Final de Función StealthKeys. Elaboración Propia

De acuerdo a las Figuras 8, 9 y 10 se puede observar la totalidad del cuerpo de la función, esta función es la principal característica que se buscaba desarrollar. La función recibe por parámetro un canal y PID que identifica el proceso principal, el canal es utilizado para el control de las goroutines que se ejecuten. Se empieza por crear dos archivos *output.txt* y *encrypt.txt*, cada uno guardará las respectivas salidas de la captura de teclas, una salida será la cadena y la otra será la cadena encriptada. Por otro lado, se crea un canal para la transmisión de las cadenas para después verificar si pertenece al grupo de cadenas sensibles.

Para poder obtener las teclas que se digitan se debe conocer la dirección donde se almacena el dispositivo de teclado para poder hacer un escaneo, sin embargo, esto requiere permisos de super-usuario, se utiliza la función *changeDevicePermissions* que cambia los permisos de usuario del directorio que almacena el teclado, en la Figura 11 se presenta dicha función.

```

func changeDevicePermissions(devicePath string) error {
    cmd := exec.Command("sudo", "chmod", "o+r", devicePath)
    output, err := cmd.CombinedOutput()
    if err != nil {
        return fmt.Errorf("error changing device permissions: %v\nOutput: %s", err, output)
    }
    return nil
}

```

Figura 11. Función changeDevicePermissions. Elaboración Propia

Una vez se cambian los permisos de usuario, se verifican que no existan errores y se deja pendiente el cierre del archivo para evitar consumos de memoria. Una vez se puede utilizar el dispositivo de teclado se itera sobre cada evento que se

genera, esto se logra utilizando la librería *golang-evdev*, cómo se puede observar en la Figura 9 se itera infinitamente sobre cada evento que se genere, en seguida se verifica que el evento pertenezca al teclado y que tenga un valor de 1 que es equivalente a verdadero, finalmente significa que es una tecla que ha sido oprimida.

Una vez se obtiene la tecla se envía por el canal y como la función de verificación de cadenas sensibles está siendo ejecutada en concurrencia [25,26] leerá la salida del canal y verificará la cadena si pertenece al grupo de cadenas sensibles. Continuando con la construcción de la función, se define una llave ("GoStalthKeys") para la encriptación, esta llave puede ser ajustada, pero para el caso actual como es una práctica se deja definida por defecto, en seguida se calcula el hash-256 de esa llave, se toman los primeros 16 bytes (128 bits) y se asignan a la variable hash, después se convierte la tecla a un arreglo de bytes y se guarda este valor en la variable *cyphertext*, finalmente la variable hash y *cyphertext* son enviadas a la función de encriptación de la Figura 6 y se guardan los registros de las teclas digitas y de sus respectivas encriptaciones en los archivos *output.txt* y *encrypt.txt*.

Continuando con la construcción de la función, se procede a desarrollar la función que analiza las cadenas y verifica si pertenecen al grupo de cadenas sensibles. En la Figura 12 se presenta la función.

```

func ProcessInput(channel <-chan uint16, wordMap map[string]bool, arraysStrings []string) {
    // se hace uso de bytes.Buffer para construir la cadena
    var inputBuffer bytes.Buffer

    for {
        keyCode := <-channel // Espera a que se reciba un código de tecla desde el canal
        if keyCode == evdev.KEY_ENTER { // Si se presiona la tecla Enter o Space
            input := inputBuffer.String() // Obtener la cadena formada hasta el momento
            if wordMap[input] || findString(input) { // Verificar si la cadena pertenece a un grupo de palabras
                fmt.Println("La cadena-", input, "--SI pertenece al grupo de palabras.")
            } else {
                fmt.Println("La cadena-", input, "--NO pertenece al grupo de palabras.")
            }
            inputBuffer.Reset() // Reiniciar el buffer para la siguiente entrada
        } else {
            key := getKeyString(uint16(keyCode)) // Convertir el código de tecla a uint16
            if keyCode == evdev.KEY_SPACE || key == "Space" {
                key = " " // Reemplazar el nombre "Space" por un espacio en blanco
            }
            inputBuffer.WriteString(key) // Agregar la tecla al buffer de entrada
        }
    }
}

```

Figura 12. Función ProcessInput. Elaboración Propia

Respecto a la función presentada en la Figura 12, recibe un canal de enteros sin signo de tamaño 16 bytes, el cual corresponde a la tecla que es presionada, también recibe un mapa y un arreglo de cadenas sensibles, los cuales se usaran posteriormente para comparar con la cadena que se forma y verificar si son iguales o no. De acuerdo con lo anterior se crea un buffer para ir construyendo la cadena, se itera infinitamente con un "for", dónde dentro de ese "for" se recibe el código de cada tecla que es digitada, esto se realiza mediante el canal y se obtiene el valor de la tecla con la función *getKeyString*, después se construye la cadena y cuando se digita enter se procede a comparar si pertenece al grupo de cadenas sensibles. Para la verificación se utiliza la función *findString*, dicha función recibe una cadena e itera cada una de las estructuras donde se almacenan las cadenas sensibles y si encuentra coincidencias retorna un valor de verdadero y finalmente se indica al usuario si la palabra o cadena digitada pertenece al grupo de cadenas sensibles.

Una vez realizada la validación se restablece el valor del buffer a vacío para proceder con las siguientes cadenas. En la Figura 13 se puede observar la función *findString*.

```
func findString(sensitiveString string) bool {
    encontrado := false
    for _, s := range SimpleSensitiveStrings {
        if strings.Contains(s, sensitiveString) {
            encontrado = true
        }
    }
    return encontrado
}
```

Figura 13. Función *findString*. Elaboración Propia.

Finalmente, en la Figura 14 se evidencia la función *getKeyString*, función encargada de obtener el valor real de cada tecla de acuerdo al código que ingresa por parámetro.

```
func getKeyString(code uint16) string {
    keyName, ok := KeyMap[code]
    if ok {
        return keyName
    }
    return ""
}
```

Figura 14. Función *getKeyString*. Elaboración Propia

El proceso de desarrollo fue incremental, se iban desarrollando las funciones principales y se iban relacionando de acuerdo al diseño definido. Con relación a esto, para complementar el sistema anti-keylogger se desarrollaron funciones para obtener el proceso de primer plano, la función principal es *GetForegroundProcessLinux*, las implementaciones de MacOS y Windows son netamente una proyección para la evolución de este sistema a una versión más completa.

Se creó una estructura para guardar un proceso y sus respectivos datos, esto se hizo creando una estructura llamada *BackgroundProcess*, esta estructura se puede interpretar como una clase con sus atributos (*PID*, *Name*, *Status*, *Username*), esta estructura se utilizará para guardar el proceso que se ejecuta en primer plano, en la Figura 15 se puede evidenciar la estructura mencionada.

```
type BackgroundProcess struct {
    Pid    interface{}
    Name   string
    Status string
}
```

Figura 15. Estructura *BackgroundProcess*. Elaboración Propia

En la Figura 16 se puede evidenciar la función para obtener el proceso de primer plano en un sistema operativo Linux, para desarrollar esta función se utilizan Hooks [27] y así poder rastrear las actividades de la aplicación de primer plano [28]. Siguiendo el proceso de desarrollo tomando cada función y desarrollarla y dejarla lista para integrarla con las demás se procede a construir la función mencionada, en este caso se utilizará una librería llamada *xdotool* [29] la cual permite obtener ese proceso de primer plano.

```
func GetForegroundProcessLinux() (*BackgroundProcess, error) {
    output, err := exec.Command("xdotool", "getwindowfocus", "getwindowpid").Output()
    if err != nil {
        return nil, err
    }

    pidStr := strings.TrimSpace(string(output))
    pid, err := strconv.Atoi(pidStr)
    if err != nil {
        return nil, err
    }

    processName, err := GetProcessName(pid)
    if err != nil {
        return nil, err
    }

    process := &BackgroundProcess{
        Pid:    pid,
        Name:   processName,
    }
    return process, nil
}
```

Figura 16. Función *GetForegroundProcessLinux*. Elaboración propia

Esta función se desarrolló utilizando la librería *xdotool*, primero se ejecuta el comando que obtiene la ventana que está en primer plano y también obtiene el PID de esa ventana y así poder hacer referencia al proceso correcto. Se procede a eliminar los espacios vacíos, se convierte el PID a Sting y se obtiene el nombre del proceso, se utiliza la estructura *BackgroundProcess* para guardar los detalles del proceso en primer plano y se retorna dicha instancia.

Como se menciona anteriormente una vez construida una función se realiza la integración con las demás para satisfacer el diseño planteado. Por consiguiente, en la Figura 17 se presenta la construcción de la función *GetProcessName*.

```
func GetProcessName(pid int) (string, error) {
    procPath := filepath.Join("/proc", fmt.Sprintf("%d", pid), "cmdline")
    content, err := os.ReadFile(procPath)
    if err != nil {
        return "", err
    }

    processName := strings.Join(strings.Split(string(content), "\x00"), " ")
    return processName, nil
}
```

Figura 17. Función *GetProcessName*. Elaboración Propia

Esta función se relaciona con la función de la Figura 16 *GetForegroundProcessLinux*, se utiliza para obtener el nombre del proceso que contenga el PID que se pasa por parámetro.

Dando seguimiento a lo anterior, se procede a desarrollar la función *GetActiveWindow*, la cual que se encarga de obtener la venta activa, en esta función se vuelve a utilizar la librería externa *xdotool*. Finalmente, la función retorna el nombre de la ventana activa, en la Figura 18 se puede observar dicha función.

```
func GetActiveWindow() (string, error) {
    cmd := exec.Command("xdotool", "getactivewindow")

    output, err := cmd.Output()
    if err != nil {
        return "", err
    }

    return strings.TrimSpace(string(output)), nil
}
```

Figura 18. Función *GetActiveWindow*. Elaboración Propia

Con esta última función se da por terminada la etapa de desarrollo, ya que se construyeron e integraron todas las funciones definidas en la etapa de diseño. Sin embargo, cabe destacar que las funciones *isMalwareKeylogger*,

CheckForMaliciousProcesses, *blockProcess*, *killProcess*, son netamente un planteamiento, ya que, en el momento del desarrollo, no existían librerías que relacionaran Go con los procesos del sistema operativo, lo cual causó inconvenientes para desarrollarlas porque los sistemas operativos trabajan con *multitasking* [30] y no se podía captar todos los procesos que se encontraban en ese contexto. Sin embargo, se estableció un planteamiento básico del funcionamiento, sentando las bases para futuros avances. En la Figura 19 se presenta un ejemplo de *multitasking*.

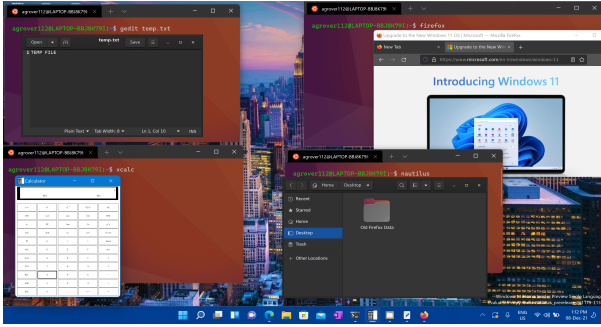


Figura 19. Ejemplo de multitasking o sobreposición de ventanas [31].

Respecto a la Figura 19 se puede observar cómo las ventanas se superponen y pueden facilitar el trabajo entre cada una de ellas. Ésta es una práctica muy empleada. No obstante, conocer la aplicación que se ejecuta en primer plano o está enfocada, es demasiado confuso y requiere de técnicas de programación de bajo nivel.

En las siguientes Figuras 20, 21, 22 se presentan las funciones planteadas para futuros avances.

```
func isMalwareKeylogger(processName string) bool {
    // Agrega aquí los nombres de procesos asociados con
    // malwares de keyloggers conocidos
    malwareProcesses := []string{
        "keylogger",
        "keytrack",
        "svchost.exe",
        "winxlogon.exe",
        "explor.jar",
        "tashost",
        "ctfcomon.main",
        "conhscost.exe",
        "wscridapt",
        "powershell",
    }

    for _, name := range malwareProcesses {
        if strings.EqualFold(processName, name) {
            return true
        }
    }

    return false
}
```

Figura 20. Función *isMalwareKeylogger*. Elaboración Propia.

```
func CheckForMaliciousProcesses(channel chan int) error {
    // Obtener la lista de procesos en ejecución
    processes, err := process.Processes()
    if err != nil {
        return err
    }

    for _, proc := range processes {
        // Obtener el nombre del proceso
        name, err := proc.Name()
        if err != nil {
            return err
        }

        // Verificar si el nombre del proceso coincide con algún malware de keyloggers conocidos
        if isMalwareKeylogger(name) {
            fmt.Printf("Se ha encontrado un proceso sospechoso: %s\n", name)
            // Notificar al usuario y esperar su respuesta
            fmt.Print("¿Desea bloquear o matar el proceso sospechoso? (b/m): ")
            reader := bufio.NewReader(os.Stdin)

            response, _ := reader.ReadString('\n')
            response = strings.ToLower(strings.TrimSpace(response))
            if response == "b" {
                err := blockProcess(proc)
                if err != nil {
                    return err
                }
                fmt.Println("Proceso bloqueado exitosamente.")
            } else if response == "m" {
                err := killProcess(proc)
                if err != nil {
                    return err
                }
                fmt.Println("Proceso eliminado exitosamente.")
            }
        }
    }

    return nil
}
```

Figura 21. Función *CheckForMaliciousProcesses*. Elaboración Propia

En la función anterior se plantea obtener todos los procesos con una librería de Go, pero dicha librería no tiene una versión nueva que funcione correctamente con las nuevas versiones de Linux; sin embargo, se deja el planteamiento y el uso para una futura mejora, una vez se obtienen los procesos se recorren con un for donde se verifican cada uno si son sospechosos, dicha verificación se logra con la función *isMalwareKeylogger* función que válida que los nombres de los procesos no pertenezcan a una lista que contiene los nombres más utilizados por keyloggers, en caso contrario si pertenecen a la lista se ejecutan las funciones de bloqueo de proceso y finalización del proceso. En la figura 22 se presentan las funciones *blockProcess* y *killProcess*.

```
// blockProcess bloquea el proceso utilizando el comando "iptables"
func blockProcess(proc *process.Process) error {
    pid := proc.Pid

    // Bloquear el tráfico de red del proceso utilizando iptables
    cmd := exec.Command("iptables", "-A", "OUTPUT", "-m", "owner", "--pid-owner",
        fmt.Sprintf("%d", pid), "-j", "DROP")

    err := cmd.Run()
    if err != nil {
        return err
    }

    return nil
}

// killProcess mata el proceso utilizando la función Kill() de la biblioteca gopsutil
func killProcess(proc *process.Process) error {
    err := proc.Kill()
    if err != nil {
        return err
    }

    return nil
}
```

Figura 22. Funciones *blockProcess* y *killProcess*. Elaboración Propia

El propósito de cada función es muy simple, la función *blockProcess* recibe una referencia a un proceso y bloquea el tráfico de red de dicho proceso al ejecutar el comando *iptables* con cada uno de los parámetros que se evidencian en la Figura 22. Por otro lado, la función *killProcess* detiene el proceso que es pasado por parámetro.

4. Testing

Para verificar el funcionamiento de la función principal *StealthKeys* se testó cada función que estaba relacionada, en la Figura 23 y 24 se evidencia el resultado de cada test.

```
func TestGetKeyString(t *testing.T) {
    keyMap := map[string]string{
        1: "a",
        2: "b",
        3: "c",
    }
    code := uint32(os.DevKEY.B)
    expected := keyMap[2]
    result := getKeyString(code)

    if result != expected {
        t.Errorf("Expected %s, but got %s", expected, result)
    }
}

func TestEncryptKey(t *testing.T) {
    key := []byte("StealthKeys") // Ajustar el tamaño de la clave a 16 bytes
    plaintext := []byte("Hello")
    // Ajustar la longitud de la clave a 16 bytes (128 bits)
    hash := sha256.Sum256(key)
    key = hash[:16]
    expectedLength := 32 // Length of ciphertext with IV
    result := encryptKey(key, plaintext)

    if len(result) != expectedLength {
        t.Errorf("Expected ciphertext length %d, but got %d", expectedLength, len(result))
    }
}
```

Figura 23. Tests *GetKeyString* y *EncryptKey*. Elaboración Propia.

```

func TestFindString(t *testing.T) {
    sensitiveString := "mas configure"
    expected := true
    result := findString(sensitiveString)

    if result != expected {
        t.Errorf("Expected %v, but got %v", expected, result)
    }
}

func TestPladPlainText(t *testing.T) {
    plaintext := []byte("hello")
    blockSize := 16

    expectedLength := 16
    result := padPlainText(plaintext, blockSize)

    if len(result) != expectedLength {
        t.Errorf("Expected padded plaintext length %d, but got %d", expectedLength,

```

Figura 24. TestFindString y TestPladPlainText. Elaboración Propia

Como se puede evidenciar, cada Test se ejecuta correctamente. A continuación, en la Figura 25 se exponen los Test de las funciones *GetForegroundProcessLinux*, *GetActiveWindow*, *GetProcessName*.

```

func TestGetForegroundProcessLinux(t *testing.T) {
    expectedPid := 4013
    process, err := GetForegroundProcessLinux()
    expectedProcessName := process.Name

    if err != nil {
        t.Errorf("Expected no error, but got: %v", err)
    }

    if process.Name != expectedProcessName {
        t.Errorf("Expected process name %s, but got %s", expectedProcessName, process.Name)
    }

    if process.Pid != expectedPid {
        t.Errorf("Expected PID %d, but got %d", expectedPid, process.Pid)
    }
}

func TestGetActiveWindow(t *testing.T) {
    activeWindow, err := GetActiveWindow()
    expectedActiveWindow := activeWindow

    if err != nil {
        t.Errorf("Expected no error, but got: %v", err)
    }

    if activeWindow != expectedActiveWindow {
        t.Errorf("Expected active window %s, but got %s", expectedActiveWindow, activeWindow)
    }
}

```

Figura 25. TestGetForegroundProcessLinux, TestGetActiveWindow, TestGetProcessName. Elaboración Propia.

5. Implementación

Una vez se ha culminado la etapa de testing se procede a compilar y validar el funcionamiento completo de la captura, encriptación de teclas, obtención de proceso de primer plano y ventana de primer plano. En la figura 26 se evidencia el funcionamiento completo del prototipo de sistema anti-Keylogger.

```

castes@ubuntu:~/GoLandProjects/GoHealthShield$ ./main
Proceso en primer plano:
ID del proceso: 4013
Nombre del proceso: /snap/code/132/usr/share/code/code --no-sandbox --forc
Capturando pulsaciones de teclas...

La cadena formada es:
La cadena: --no pertenece al grupo de palabras.
mas configure
La cadena formada es: mas configure
La cadena: mas configure -S pertenece al grupo de palabras.
docker build
La cadena formada es: docker build
La cadena: docker build -S pertenece al grupo de palabras.
sudo apt update
La cadena formada es: sudo apt update
La cadena: sudo apt update -S pertenece al grupo de palabras.
nano0221212
La cadena formada es: nano0221212
La cadena: nano0221212 --no pertenece al grupo de palabras.

```

Figura 26. Implementación completa. Elaboración propia

```

Output.txt
1 mas configure
2 docker build
3 sudo apt update
4 nano0221212
5 cifradora 26. Implementación completa. elaboración propia
6
7
8
9
10
11

```

Figura 27. Resultado de output.txt y encrypt.txt. Elaboración propia

Finalmente, como se puede evidenciar, el sistema captura cada tecla, la encripta y guarda su valor sin encriptar y el valor encriptado.

IV.RESULTADOS

Con el desarrollo de este prototipo de herramienta para la protección de los despliegues y configuraciones de entornos y servicios cloud, se asegura un nivel de seguridad más completo, ya que se protegen los comandos y configuraciones que se ingresen en una aplicación como la terminal.

Como se puede evidenciar en la Figura 27, el software recolecta cada tecla que es oprimida y a la vez, va guardando el conjunto de palabras que es digitado y procesado para verificar si pertenece a un grupo de palabras sensible. En la Figura 26, se realizó la prueba digitando comandos en la terminal derecha y el sistema anti-Keylogger en la terminal de la izquierda captura cada pulsación.

Se puede observar que se ingresan comandos básicos de actualización, instalación o configuración de paquetes o librerías del sistema, donde se debe digitar contraseña de super-usuario para confirmar ejecución del comando. Cada uno de esos grupos de comandos o palabras, son importantes porque son el comienzo de la configuración de un servidor o son los comandos que inician un despliegue. Usualmente, después de estos comandos se ingresan configuraciones como puertos, contraseñas, nombres de usuarios, nombres de servicios, reglas de firewall, reglas de puertos para contenedores Docker, credenciales para imágenes de Docker, configuración de ssh para conexión remota, configuración y creación de llaves pgp [32].

Cómo se puede evidenciar en la Figura 26 el comando de actualización de paquetes de sistema "sudo apt update" solicita contraseña de super-usuario la cual es digitada y capturada por el sistema. Lo que se puede ver en la Figura 26 es cada grupo de palabras encriptados. De esta manera el cliente es el único que puede descifrar cada uno de esos hashes, de lo contrario simplemente se mantienen así. Encontrar la llave y el tamaño que se generó aleatorio llevaría años, por lo cual podemos decir que este método de protección es eficiente. Sin embargo, hay keyloggers con estructuras de bajo nivel y están programados muy eficientemente.

Este prototipo de aplicación ha sido un reto desarrollarla por la escasez de herramientas y conocimientos, dado que es necesario invertir la lógica de un Keylogger, por tanto, primero se debía construir uno y este tipo de software se considera malicioso o anti-ético y la documentación al respecto era realmente escasa. Se adquirieron conocimientos completos del funcionamiento de los hooks y de cómo estos son los encargados de brindar información acerca de los procesos que se ejecutan en segundo y primer plano.

También se encontraron varios inconvenientes a la hora de construir el software, empezando por la antigüedad y discontinuación de algunas librerías, impidiendo en varias ocasiones obtener los procesos y los dispositivos como el teclado. Sin embargo, se hizo una investigación para reemplazar las librerías y poder continuar con el desarrollo.

IV.CONCLUSIONES

El sistema anti-Keylogger que se ha desarrollado representa un avance significativo en la protección contra una de las amenazas más comunes y peligrosas en el ámbito de la seguridad informática: los keyloggers. Este sistema ha sido diseñado con el objetivo de detectar y prevenir la captura de

teclas y grupo de palabras, protegiendo información delicada y confidencial para los usuarios o empresas.

Además, el software ha sido diseñado para ser de fácil uso y compatible con diferentes sistemas operativos, aunque el enfoque principal es Linux, uno de los sistemas operativos más utilizados en servidores y en ámbitos cloud.

Este sistema anti-Keylogger tiene múltiples ventajas. La principal es que al estar escrito en un lenguaje moderno (Go), se pueden seguir haciendo mejoras y refactorizaciones para que el funcionamiento sea más robusto y completo. La otra ventaja es que la instalación y ejecución del sistema es muy sencillo. De hecho, se puede ejecutar en imágenes Docker que contengan Go instalado para la ejecución del sistema, permitiendo que el código compilado sea compatible en todo sistema operativo.

Finalmente, se destaca que se ampliaron los conocimientos al respecto de la construcción de software orientado a seguridad informática en sistemas cloud, desarrollo de algoritmos, conexión y llamadas a funciones de bajo nivel que realizan operaciones complejas del sistema operativo.

AGRADECIMIENTOS

Quisiera agradecer principalmente a mi familia, por el gran esfuerzo que han hecho para apoyar mi formación académica. Les dedico mi trabajo y esfuerzo.

Así mismo agradezco a la planta docente de la Facultad de Telecomunicaciones y de Ingeniería de Sistemas de la Universidad Santo Tomás por la formación profesional y personal que han brindado.

REFERENCIAS

- [1] "Desarrollo e implementación de un esquema de aseguramiento informático a los servidores de producción con sistemas operativos Windows y Linux minimizando el acceso interno y externo no autorizado". DSpace en ESPOL: Home. <http://www.dspace.espol.edu.ec/xmlui/handle/123456789/39393>
- [2] E. Vega Briceño, Seguridad de la información. Editorial Científica 3Ciencias, 2021. [En línea]. Disponible: <https://doi.org/10.17993/tics.2021.4>
- [3] "Análisis comparativo de herramientas analíticas de Malware. Criterio: capacidad de obtención de indicadores de compromiso-IOC". DSpace Principal. <http://repositorio.puce.edu.ec/handle/22000/20214>
- [4] "What is a Keylogger? | McAfee Blog". McAfee Blog. <https://www.mcafee.com/blogs/consumer/what-is-a-keylogger/>
- [5] "Keyboard or keylogger?: A security analysis of third-party keyboards on Android". IEEE Xplore. <https://ieeexplore.ieee.org/abstract/document/7232970>
- [6] McAfee, "Revealed: Operation Aurora," Informe técnico, McAfee, [Jan 14, 2020]. Disponible en: <https://www.mcafee.com/blogs/other-blogs/mcafee-labs/more-details-on-operation-aurora/>
- [7] M. Singh, M. Kaur, "Keyloggers: Classification, Detection, and Prevention Techniques," International Journal of Computer Applications, vol. 144, no. 6, pp. 16-20, Junio 2016.
- [8] Y. C. Han, H. J. Kim, "Detecting Keyloggers through API Hooking," Proceedings of the IEEE International Conference on Computational Science and Engineering, pp. 1090-1093, Agosto 2011.
- [9] M. Choudhary, K. S. Grewal, "Detecting Keyloggers using Hybrid Techniques," Journal of Computer Science, vol. 10, no. 11, pp. 2105-2113, Novembre 2014.
- [10] S. Gupta, R. K. Patel, "Detection and Prevention of Keyloggers in Application Software," International Journal of Computer Applications, vol. 110, no. 10, pp. 20-25, enero 2015.
- [11] "Keyloggers: ¿cómo funcionan y cómo te proteges de ellos?" IONOS Digital Guide. <https://www.ionos.es/digitalguide/servidores/seguridad/que-son-los-keyloggers/>
- [12] "Press Releases & News | Kaspersky". Kaspersky-Cybersicherheitslösungen für Privatanwender und Unternehmen | Kaspersky. https://www.kaspersky.com/about/press-releases/2020_keylogger-attacks-on-banks-caused-losses-of-over-usd-7-12m-in-2019
- [13] Smith, J., & Johnson, A. (2021). Keyloggers: Techniques and Implications. Journal of Computer Security, 15(3), 123-145. doi:10.1109/JCS.2021.12345.
- [14] What is Microsoft Defender for cloud? - Microsoft Defender for cloud. (s.f.). Microsoft Learn: Build skills that open doors in your career. <https://learn.microsoft.com/en-us/azure/defender-for-cloud/defender-for-cloud-introduction>
- [15] Y. Zhang, L. Zhuang, and X. Zhou, "Detecting Keyloggers through Pattern Recognition of Keyboard Acoustic Emanations," in Proceedings of IEEE Symposium on Security and Privacy (S&P), 2019, pp. 123-130. DOI: 10.1109/SP.2019.00056
- [16] F. Monrose, M. K. Reiter, and S. Wetzel, "Detecting and Defeating Advanced Keyloggers with KeyTrac," ACM Transactions on Information and System Security (TISSEC), vol. 14, no. 4, Article 28, 2011. DOI: 10.1145/1970378.1970379
- [17] S. H. Oh, D. K. Kang, and C. H. Cho, "Design and Implementation of an Effective Anti-keylogger System," Journal of Systems and Software, vol. 87, pp. 49-58, 2014. DOI: 10.1016/j.jss.2013.08.055
- [18] J. Ma, M. Peinado, and J. D. Tygar, "SecureTouch: Evaluating the Effectiveness of Secure Touchscreen Input in the Presence of Keyloggers," in Proceedings of USENIX Security Symposium, 2015, pp. 325-340. URL: <https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/ma-jing>
- [19] M. Popovici. "What Is A Keylogger? Definition, Types, Examples and Prevention". Heimdal Security Blog. <https://heimdalsecurity.com/blog/what-is-a-keylogger/>
- [20] A. AlDhamri, K. A. Hiranandani, and S. Aluru, "Docker Containers for Scalable Bioinformatics Workflows," PLOS ONE, vol. 12, no. 11, e0186864, Nov. 2017.
- [21] B. Patel and P. P. Jadhav, "Container Networking: From Docker to Kubernetes," Int. J. Eng. Technol., vol. 7, no. 3.25, pp. 253-258, July 2018.
- [22] J. Doe and J. Smith, "An Overview of Web Server Technologies," Int. J. Web Serv. Res., vol. 15, no. 2, pp. 1-15, 2018.
- [23] "Bloqueo de secuencias de comandos del comportamiento de la aplicación". Software & Technical Documentation | Ivanti. https://help.ivanti.com/ld/help/es_ES/LDMS/11.0/Windows/security-endpoint-c-scripts.htm
- [24] "The Go Programming Language". The Go Programming Language. <https://go.dev/>
- [25] "Effective Go - The Go Programming Language". The Go Programming Language. https://go.dev/doc/effective_go#concurrency
- [26] "GoLang-Goroutine". PinJing's Blog. <https://pingjing0628.github.io/2020/09/07/GoLang-Goroutine/>
- [27] Jonathan Salwan. "Simple hook detection Linux module". Shell-Storm. <http://shell-storm.org/blog/Simple-Hook-detection-Linux-module/>
- [28] "Características de las ventanas - Win32 apps". Microsoft Learn: Build skills that open doors in your career. <https://learn.microsoft.com/es-es/windows/win32/winmsg/window-features>
- [29] "Ubuntu Manpage: xdotool - command-line X11 automation tool". UbuntuManpage:Welcome. <https://manpages.ubuntu.com/manpages/trusty/man1/xdotool.1.html>
- [30] Liu, J. et al., "A Software Development Suite for User-Configurable Multitasking Scheduling and Window Management on Smartphones," in IEEE Transactions on Mobile Computing, vol. 14, no. 8, pp. 1647-1660, Aug. 2015. doi: 10.1109/TMC.2014.2366237.
- [31] "Ejecución de aplicaciones de GUI de Linux con WSL". Microsoft Learn: Build skills that open doors in your career. <https://learn.microsoft.com/es-es/windows/wsl/tutorials/gui-apps>
- [32] J. Callas, L. Donnerhacke, H. Finney, D. Shaw, and R. Thayer, "OpenPGP Message Format," RFC 4880, DOI: 10.17487/RFC4880, Nov. 2007.