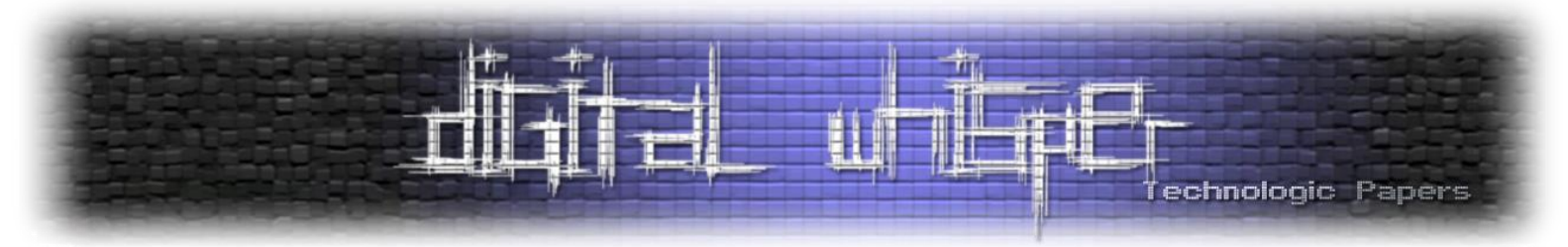

מימוש ופיצוח של PRNG ב-Golang

מאת יוראי הרצברג

הקדמה

אם תיכנסו לדוקומנטציה של פונקציית ה-random האהובה עליכם (לדוגמה random.random בפייתון או mt_rand ב-php), כנראה שתראו אזהרה מהסוג הבא:

Warning: The pseudo-random generators of this module should not be used for security purposes. For security or cryptographic uses, see the `secrets` module.

[התמונה לקוחה מתוך הדוקומנטצייה של *random.random* בפייתון]

האם האזהרה הזו חשובה, או שהסיכון הוא מינורי, ואפשר להתעלם ממנו? במאמר זה אנחנו הולכים "לפתוח את הקופסה השחורה", ולהבין מה זה בכלל PRNG (או Pseudo-Random Number Generator), איך הוא עובד מבפנים, וכיצד ניתן, בהינתן מספר מצומצם של פלטים, ובאמצעות התקפה קריפטוגרפית, לדעת מה היו כל הפלטים של ה-PRNG בעבר, ומה יהיו כל הפלטים של ה-PRNG בעתיד.

כל קונספט שיוצג במאמר ימומש ב-Golang. לא נדרש ידע קודם ב-Golang - אנחנו נשתמש רק בסט יחסית קטן של פיצ'רים מתוך Golang שדומה לשפות אחרות, ואסביר על הקוד איפה שצריך.

בתור ידע קודם למאמר, מומלץ לדעת קצת אלגברה לינארית (כפל מטריצות, מטריצות הופכיות, וכו').

אז מה זה בכלל PRNG?

כולנו מבינים, במידה מסוימת, את משמעות המושג "רנדומליות". לדוגמה: גלגול קובייה היא תופעה שנחשבת רנדומלית (כל כך רנדומלית שאם תקחו ספר כלשהו על הסתברות, תהיה על הכריכה שלו קובייה בהסתברות גבוהה ☺), מכיוון שהתוצאה לא ידועה מראש - כל תוצאה של הקוביה יוצאת בהסתברות שווה (או במילים אחרות; תוצאות הקוביה מתפלגות בצורה אחידה).

על מנת ליצור מספרים רנדומליים בעולם הדיגיטלי (לדוגמה בשביל להריץ איזושהי סימולציה מדעית, או בשביל לתת למשתמש token סודי), ישנן שתי אפשרויות:

- שימוש בחומרה: Hardware-Based Random Number Generators, או בקיצור HRNGs, משתמשים בגורמים חיצוניים מהסביבה על מנת ליצור מספרים רנדומליים. הגורמיים החיצוניים יכולים להיות פיזיים, לדוגמה הטמפרטורה בחדר, או לא פיזיים, לדוגמה מספר ה-open syscalls שהגיעו ל-kernel בדקה האחרונה
- Pseudo Random Number Generators, או בקיצור PRNGs. שומרים רשימה של מספרים, שנקראת internal state, וכאשר המשתמש מבקש מהם לייצר את המספר הבא, הם מחזירים מספר שהוא טרנספורמציה על ה-state הנוכחי שלהם, ומשנים את ה-state

חשוב להבין שהמספרים המיוצרים ע"י PRNGs עדיין מתפלגים בצורה אחידה. עם זאת, המספר הבא ש-PRNG מייצר תלוי אך ורק ב-state הנוכחי שלו. אילו נצליח למצוא את ה-state הפנימי של ה-PRNG, נוכל לדעת מה יהיו כל המספרים שהוא ייצור בעתיד, ע"י כך ש"נחקה" את הטרנספורמציה שה-PRNG מפעיל על ה-state, שכאמור אינה מושפעת מגורמים חיצוניים כגון הסביבה.

איך נראה PRNG מבפנים?

ה-PRNG שנדבר עליו היום נקרא Mersenne Twister. הוא הומצא בשנת 1997 ע"י Makoto Matsumoto ו-Takauji Nishimura, ומאז נהפך ל-PRNG הדיפולטיבי באינספור שפות תכנות: לדוגמה ה-mt בפונקציית ה-mt_rand של PHP הוא קיצור של Mersenne Twister. גם פייתון (המודול random) ו-JS (הפונקציה Math.random) משתמשים באלגוריתם זה בתור PRNG.

Seeding

כפי שנאמר בחלק הקודם, PRNGs שומרים state פנימי של מספרים. ב-MT ה-state הוא בגודל 624, וכל אלמנט ב-state הוא word-sized integer (לדוגמה אם אנחנו רוצים ליצור uint32_t, כלומר שלם חיובי בגודל 32 ביט, כל אלמנט ב-state שלנו גם יהיה uint32_t). ב-Golang, אנו נייצג את ה-PRNG באמצעות ה-structure הבא:

```
// Degree of reccurence
const n int = 624
// The current state of the Mersenne Twister
type mt_state struct {
  state     [n]uint32
  state_idx uint
}
```

ה-structure מכיל את ה-state הפנימי של ה-PRNG, שהוא כאמור array בגודל 624 מספרים, ואינדקס נוכחי לתוך ה-state, שנשתמש בו מאוחר יותר.

ה-state מאותחל באמצעות אלגוריתם **seeding**, שלוקח word-size integer אחד שנקרא ה-seed (זהו ה-argument לפונקציות כמו random.seed בפייתון או mt_srand ב-PHP), וממנו מאתחל את כל המספרים ב-state לפי נוסחת הנסיגה הבאה (למי שלא מכיר, הסימון $\oplus$ מסמן XOR):

$$state[i+1] = F * (state[i] \oplus (state[i] \gg w-2))) + i$$

האלמנט הראשון ב-state מוגדר להיות ה-seed עצמו. המשתנה F הוא קבוע, שמוגדר כ-1812433253 במימושים 32-ביט של MT. המשתנה w מציין את ה-word size, שבמקרה שלנו הוא 32. ב-Golang, אנו נממש את פונקציית האתחול כך:

```
// Initialize the state of the MT according to some seed
func (mt *mt_state) init_state(seed uint32) {
  //  First element of the initial state is the seed
  mt.state[0] = seed

  // All of the next elements are initialized
  // with a reccurence
  for i := 1; i < n; i++ {
    prev := uint(mt.state[i-1])

    mt.state[i] = uint32(f*(prev^(prev>>(w-2)))) + uint32(i)
  }

  mt.state_idx = 0
}
```

הפונקציה לוקחת מצביע ל-mt_state, ואז מאתחלת אותו באמצעות נוסחת הנסיגה לפי ה-seed שהיא מקבלת. האינדקס מאותחל ל-0, כי עוד לא השתמשנו ב-PRNG על מנת ליצור מספרים. נכתוב פונקציית wrapper קטנה ל-init_state שגם תחזיר לנו structure חדש בהינתן ה-seed בלבד (הפונקצייה init_state רק מקבלת מצביע ל-mt_state קיים):

```
// Initialize a new MT
func new_mt(seed uint32) *mt_state {
    var state [n]uint32
    mt := mt_state{state: state, state_idx: 0}
    mt.init_state(seed)

    return &mt
}
```

ייצור מספרים

ייצור המספר הבא מה-PRNG הוא תהליך המורכב משני שלבים:

1. שינוי האלמנט הישן ביותר ב-state (שהאינדקס שלו מאוחסן ב-state_idx של ה-struct שייצרנו), באמצעות נוסחת נסיגה
2. הפעלת טרנספורמציית ערפול (Tempering Transformation) שנקרא לה T לערך החדש שאנחנו שמים ב-state, והחזרת הפלט של טרנספורמציה זו

עובדה שתעזור לנו מאוחר יותר, כאשר נפצח את ה-PRNG, היא שטרנספורמציית הערפול הזו היא הפיכה; כלומר, קיימת טרנספורמציה T^{-1}, כך שלכל קלט x, מתקיים $T^{-1}(T(x)) = x$.

שינוי האיבר הישן ביותר

על מנת לשנות את האיבר הישן ביותר ב-state, אנו נשתמש בנוסחת הנסיגה הבאה:

$$state[i] = state[(i+m)\%n] \oplus (state[i]^u | state[(i+1)\%n]^l)A$$

הנוסחה נראית קצת מפחידה, אז נסתכל על כל חלק שלה בנפרד :)

- הקבוע m מוגדר כ-397, לכן $state[(i+m)\%n]$ הוא האלמנט שנמצא 397 אלמנטים (מודולו n) אחרי האינדקס של האיבר שרוצים לשנות.
- בחלק הימני של הנוסחה, $state[i]^u$ הוא מספר בן 32 סיביות ש-r) $32-r$ הוא קבוע כלשהו) הביטים העליונים שלו זהים לאלו של הערך הנוכחי של $state[i]$, ו-r הביטים התחתונים שלו הם 0
- $state[(i+1)\%n]^l$ הוא מספר בן 32 סיביות ש-r הביטים התחתונים שלו זהים לאלו של $state[(i+1)\%n]$, ושאר הביטים שלו הם 0
- מכיוון שמבצעים עליהם Bitwise OR, נקבל מספר חדש ש-$32-r$ הביטים העליונים שלו הם $32-r$ הביטים העליונים של הערך הנוכחי של $state[i]$, ו-r הביטים התחתונים שלו הם r הביטים התחתונים של $state[(i+1)\%n]$.
- ב-MT, הקבוע r מוגדר כ-31.
- המספר שמקבלים אחרי ה-Bitwise OR, שהוא $(state[i]^u | state[(i+1)\%n]^l)$, מוכפל לאחר מכן משמאל במטריצה A, שמוגדרת כך:

$$A = \begin{pmatrix} & 1 & & & \\ & & 1 & & \\ & & & \ddots & \\ & & & & 1 \\ a_{w-1} & a_{w-2} & \cdots & \cdots & a_0 \end{pmatrix},$$

התמונה לקוחה מהמאמר "*Mersenne Twister: A 623-dimensionally equidistributed pseudorandom number generator*" של Makoto Matsumoto ו-Takauji Nishimura, שהוא המאמר בו *MT* הוצג לראשונה

כל האיברים הריקים במטריצה A הם אפסים. כאשר מכפילים מספר x משמאל במטריצה זו (המספר מומר לווקטור עם 32 מימדים, שהם הביטים של המספר), אם ה-LSB של המספר (כלומר הביט הימני ביותר) הוא 1, הביטים של a בשורה האחרונה של המטריצה נוספים לתוצאה.

אחרת, אם ה-LSB הוא 0, הביטים של a לא נוספים לתוצאה ונקבל את x >> 1 (מומלץ לוודא זאת!). ב-MT, הקבוע a מוגדר כ-x9908B0DF0. השורה התחתונה במטריצה A מכילה את הביטים של מספר זה. לסיכום, כאשר מכפילים מספר x ב-A משמאל, מקבלים:

$$x \cdot A = (x >> 1) \oplus (LSB\ of\ x\ is\ 0\ ?\ 0 :\ a)$$

להלן המימוש של חלק זה בקוד:

```
// Bitmask for taking the upper w-r=1 bits
const upper_mask uint = 0x80000000
// Bitmask for taking the lower r bits (r = 31)
const lower_mask uint = 0x7FFFFFFF
// Offset used in the reccurence
const m uint = 397
// The last row of the matrix A
const a uint = 0x9908B0DF

func (mt *mt_state) gen_next() uint32 {
    // The current state index
    i := int(mt.state_idx)
    // Compute the concatenation between the upper 1 bit of state[i] and lower 31 bits of state[i + 1]
    y := (mt.state[i] & uint32(upper_mask)) | (mt.state[(i+1)%n] & uint32(lower_mask))
    // Compute (part of) the matrix product
    y_lsb := y & 1
    var mat_prod uint
    // If the LSB of y is 0, the last row of A, which is the bits of a, won't be added to the result
    if y_lsb == 0 {
        mat_prod = 0
    } else {
        // Otherwise, we multiply a 1 with every bit of a, which yields a itself
        mat_prod = a
    }
    // Mutate the state by adding the i+m % n th number in the state to the matrix product
    mt.state[i] = mt.state[(i+int(m))%n] ^ (y >> 1) ^ uint32(mat_prod)
    ...
}
```

Tempering Transformation

כעת, לאחר שהאיבר הישן ביותר ב-state שונה, האלגוריתם מפעיל את ה-Tempering Transformation T על הערך החדש שרשמנו (זה יהיה הפלט של האלגוריתם). ה-Tempering Transformation מוגדרת בארבעה שלבים (שכל אחד מהם הפיך):

1. $y = state[i] \oplus (state[i] >> u)$
2. $y = y \oplus ((y << s)\&b)$
3. $y = y \oplus ((y << t)\&c)$
4. $out = y \oplus (y >> l)$

המשתנים u, s, b, t, c, l כולם קבועים: u, s, t, l (ה-shift amounts) מוגדרים כ-11,7,15,18 בהתאמה. b ו-c (ה-bitmasks) מוגדרים כ-0x9D2C5680 ו-0xEFC60000 בהתאמה.

להלן הקוד להפעלת ה-Tempering Transformation על הערך שחישבנו מקודם:

```
// Constants for the tempering transformation T
const u uint = 11
const s uint = 7
const b uint = 0x9D2C5680
const t uint = 15
const c uint = 0xEFC60000
const l uint = 18

// Generate a random number from the MT and mutate the oldest number in the state
func (mt *mt_state) gen_next() uint32 {
    ...

    // Compute the tempering transformation T of the state value we just mutated
    out := mt.state[i]
    out ^= out >> uint32(u)
    out ^= (out << s) & uint32(b)
    out ^= (out << t) & uint32(c)
    out ^= out >> uint32(l)
    // Update the state index
    mt.state_idx = uint((i + 1) % n)

    return out
}
```

בדיקת שפיות

וזה הכול! נבצע בדיקת שפיות קטנה ע"י השוואת הפלט של ה-PRNG שכתבנו עם הפלט של mt_rand ב-PHP, שכאמור משתמש באותו אלגוריתם:

```
// The Go code for using the PRNG
func main() {
  my_mt = new_mt(1337)

  fmt.Printf("PRNG Output: %d\n", my_mt.gen_next())
}
```

```
<?php
// The PHP code
mt_srand(1337);

$mt_out = mt_rand(0, getrandmax());
echo "PRNG Output: $mt_out";
?>
```

מעולה! שתי התוכנות מוציאות את הפלט:

```
PRNG Output: 1125387415
```

פיצוח ה-PRNG

כפי שנאמר בהצגת הרעיון של PRNG, אם נצליח למצוא את ה-state הפנימי של ה-PRNG, נוכל לדעת מה יהיו כל המספרים שה-PRNG ייצר בעתיד, בכך שנחקה את פעולת ה-PRNG (כלומר נריץ את gen_next) על ה-state שקיבלנו. אך איך נצליח להשיג את ה-state הפנימי?

זוכרים שאמרנו שה-Tempering Transformation היא הפיכה? אם נצליח להפוך אותה, כלומר למצוא טרנספורמציה הופכית T^{-1}, כך שלכל מספר x מתקיים $T^{-1}(T(x))$, נוכל, בהינתן פלט של ה-PRNG, שהוא הרי מהצורה $T(state[i])$, להפעיל את T^{-1}, ולקבל את $T^{-1}(T(state[i])) = state[i]$. בואו ניגש למלאכה ונהפוך את T!

טרנספורמציות↔מטריצות

אחד מהכלים השימושיים ביותר באלגברה לינארית, הוא השקילות בין משפחה מאוד רחבה של טרנספורמציות (הנקראות "טרנספורמציות לינאריות"), לבין מטריצות. לשם הבנת שקילות זו, נתבונן בטרנספורמציה $f(x,y,z) = (x,2y,z)$, הלוקחת, כקלט, וקטור תלת-מימדי (x,y,z), ומוציאה, כפלט, וקטור תלת-מימדי, המוגדר ע"י $(x,2y,z)$. על מנת להבין כיצד ניתן לייצג טרנספורמציה זו כמטריצה, נתבונן במטריצה הבאה A:

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

כאשר מכפילים וקטור כלשהו $\vec{v} = (x,y,z)$ במטריצה זו משמאל, כלומר מחשבים את $\vec{v}A$, נקבל את וקטור $\vec{u} = (x,2y,z)$, שהוא בדיוק הפלט של הטרנספורמציה שלנו f!

כעת, כשיש לנו מטריצה המייצגת את הטרנספורמציה שלנו, נוכל להשתמש בכלים הקשורים למטריצות. לדוגמה, על מנת למצוא את הטרנספורמציה f^{-1} ההופכית לטרנספורמציה f, נוכל למצוא את המטריצה A^{-1} שהופכית למטריצה A, ואז למצוא את הטרנספורמציה שמתאימה ל-A^{-1}. לדוגמה, המטריצה ההופכית של A בדוגמה שלנו היא:

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 0.5 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

הטרנספורמציה המתאימה למטריצה זו הינה $g(x,y,z) = x + 0.5y + z$; זוהי ההופכית של f!

שיטת העבודה

השיטה שראינו בחלק הקודם למצוא את הטרנספורמציה ההופכית של טרנספורמציה כלשהי עובדת גם על טרנספורמציות מסובכות יותר: לדוגמה ה-Tempering Transformation

יש לשים לב להבדל: מכיוון שאנחנו פועלים בעולם הדיגיטלי, בו קיימים רק אפסים ואחדים, אנחנו לא עובדים עם טרנספורמציות ממשיות (כלומר כאלו שלוקחות כקלט מספרים ממשיים), אלא עם טרנספורמציות שפועלות רק מעל המספרים 0 ו-1 (בשפה מתמטית נאמר שאנחנו עובדים מעל השדה הסופי $\mathbb{Z}_2$). במקום פעולות החיבור והחיסור, יש לנו רק פעולה אחת: XOR. מומלץ לוודא שאכן מעל $\mathbb{Z}_2$, פעולות החיבור והחיסור (מודולו 2) שקולות, ושתיהן מוגדרות כפעולת ה-XOR (כלומר לכל $x, y \in \mathbb{Z}_2$, מתקיים $x + y = x - y = x \oplus y$, כאשר הסימן ∈ מסמל שייכות לקבוצה).

זכרו שהטרנספורמציה T מוגדרת בארבעה צעדים:

```
// Compute the tempering transformation T of the state value we just mutated
out := mt.state[i]
out ^= out >> uint32(u)
out ^= (out << s) & uint32(b)
out ^= (out << t) & uint32(c)
out ^= out >> uint32(l)
```

במקום להפוך את הטרנספורמציה המסובכת T בבת אחת, נהפוך כל צעד שלה (כלומר כל שורה בקוד הנ"ל), ולאחר מכן נפעיל את השלבים ההופכיים בסדר הפוך (כלומר קודם השלב ההופכי לרביעי, אח"כ ההופכי לשלישי, וכן הלאה). לסיכום, זו תהיה שיטת העבודה שלנו:

1. לכל צעד בטרנספורמציה T, נמצא מטריצה המייצגת אותו
2. נמצא את המטריצה ההופכית לכל מטריצה כזו (לדוגמה עם NumPy)
3. נמיר את המטריצות ההופכיות חזרה לטרנספורמציות - אלו הטרנספורמציות ההופכיות לכל שלב!
4. נפעיל את הטרנספורמציות ההופכיות בסדר הפוך

הצעד הרביעי

נתחיל מהצעד האחרון בטרנספורמציה T:

```
out ^= out >> uint32(l)
```

או, בצורה יותר מפורשת (בלי ה^=):

$$y = x \oplus (x >> l)$$

כעת, נמצא נוסחה מפורשת לכל ביט ב-y:

$$y_i = x_i \oplus (x >> l)_i$$

כאשר a_i מציין את הביט ה-i של המספר a.

לאחר פעולת ה-right shift, מתקיים ש-l הביטים הגבוהים (כלומר הימניים) של $x >> l$ הם 0. כאשר עושים XOR למספר כלשהו עם 0 תמיד מקבלים את המספר עצמו, ולכן l הביטים הגבוהים של y הם $y_i = x_i \oplus 0 = x_i$ לכל $1 \le i \le l$.

לפי הגדרת right shift, שאר הביטים הנמוכים ב-$(x >> l)$ מוגדרים כך: $(x >> l)_i = x_{i-l}$ לכל $l < i < 32$. אם כך, לכל $l < i < 32$ מתקיים $y_i = x_i \oplus (x >> l)_i = x_i \oplus x_{i-l}$. אם נאסוף את כל מה שמצאנו עד עכשיו, נקבל:

$$
\begin{aligned}
y_0 &= x_0 \\
y_1 &= x_1 \\
&\vdots \\
y_{l-1} &= x_{l-1} \\
y_l &= x_l \oplus x_1 \\
y_{l+1} &= x_{l+1} \oplus x_2 \\
&\vdots \\
y_{31} &= x_{31} \oplus x_{31-l}
\end{aligned}
$$

על מנת להפוך את טרנספורמציה זו למטריצה (שנקרא לה T_4), נשים לב לדברים הבאים:

- l הביטים הגבוהים ב-y זהים ל-l הביטים הגבוהים ב-x. לכן, נרצה ש-l העמודות הראשונות במטריצה T_4 יהיו l העמודות הראשונות במטריצת היחידה מסדר 32
- לאחר מכן, עושים XOR לכל ביט עם הביט שהוא l ביטים לפניו, אז לכל עמודה באינדקס $i > l$, נרצה לשים 1 באינדקס (i, i) במטריצה , ועוד 1 באינדקס $(i, i-l)$ (התוצאה תהיה XOR של ביט i עם ביט $i-l$)
- כל שאר המקומות יהיו אפסים

קל יותר לראות מדוע זוהי המטריצה המייצגת אם מסתכלים על מספרים קטנים יותר: לדוגמה אם ה- word size שלנו הוא 8, נייצג את הטרנספורמציה $x \to x \oplus (x >> 3)$ באמצעות המטריצה הבאה:

$$
\begin{pmatrix}
1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1
\end{pmatrix}
$$

מומלץ לוודא - ע"י בחירת מספר בן 8 ביטים והכפלתו משמאל במטריצה זו, שאכן מתקבלת התוצאה של הטרנספורמציה הנ"ל.

באופן דומה, המטריצה המייצגת את הטרנספורמציה $y = x \oplus (x >> l)$ מלאה באחדות לאורך האלכסון הראשי, ולאורך האלכסון באורך $32 - l = 32 - 18 = 14$ מעל האלכסון הראשי (כלומר האלכסון ב-offset 18 מעל האלכסון הראשי). סיימנו עם השלב הראשון!

כעת, כשיש לנו את המטריצה המייצגת את הטרנספורמציה, נמצא את ההופכית שלה עם numpy:

```
import numpy as np
from np.linalg import inv

# Main diagonal full of ones
x = np.eye(32, dtype=np.int32)
# Fill the offset diagonal with 32-18=14 ones
# The diagonal is at offset 18 from the main diagonal
y = np.diag(np.ones(14), 18)
t4 = x + y

t4_inv = inv(t4)
```

זה היה השלב השני!

כדי למצוא את הטרנספורמציה המתאימה למטריצה ההופכית T_4^{-1} (t4_inv בקוד שלנו), נדפיס את כל האלכסונים במטריצה ההופכית שאינם מלאים ב-0-ים. הסיבה היא שפעולות ה-bitwise שאנחנו מבצעים מופיעות אך ורק לאורך האלכוסנים (כמו במטריצה המקורית). להלן הקוד שמבצע זאת:

```
for i in range(-32, 32):
    curr_diag = np.diagonal(t4_inv, i)

    if curr_diag.sum() != 0:
        print("Diagonal @ offset {} from the main diagonal has nonzero sum".format(i))
```

הקוד מדפיס:

```
Diagonal @ offset 0 from the main diagonal has nonzero sum
Diagonal @ offset 18 from the main diagonal has nonzero sum
```

ובכן, האלכסון ב-offset של 0 מהאלכסון הראשי הוא פשוט האלכסון הראשי עצמו. האלכסון ב-offset של 18 מהאלכסון הראשי הוא באורך 14. שמים לב למשהו? זוהי המטריצה T_4 עצמה!

אם כך, הטרנספורמציה ההופכית ל-$x \to x \oplus (x >> l)$ היא $x \to x \oplus (x >> l)$ עצמה! במילים אחרות, בהינתן הפלט $y = x \oplus (x >> l)$ של הטרנספורמציה, נוכל למצוא את x ע"י $x = y \oplus (y >> l)$.

הצעד הראשון

אמרנו שנהפוך את הטרנספורמציות לפי הסדר, אך שימו לב שהצעד הראשון בטרנספורמציה (או הרביעי בסדר הפוך) נראה כך:

```
out ^= out >> uint32(u)
```

כתבנו בצעד הקודם קוד שהופך טרנספורמציות מהצורה הזו, אז נוכל להשתמש בו גם פה! כל שנצטרך לעשות הוא להחליף את הערך של l בערכו של $u = 11$:

```
import numpy as np
from np.linalg import inv

# Main diagonal full of ones
x = np.eye(32, dtype=np.int32)
# Fill the offset diagonal with 32-11=21 ones
# The diagonal is at offset 11 from the main diagonal
y = np.diag(np.ones(21), 11)
t1 = x + y

t1_inv = inv(t1)
```

הפעם, האלכסון ב-offset של $u = 11$ מהאלכסון הראשי מלא באחדות. אם נריץ את הקוד שבודק אילו אלכסונים לא מלאים באפסים, נקבל:

Diagonal @ offset 0 from the main diagonal has nonzero sum

Diagonal @ offset 11 from the main diagonal has nonzero sum

Diagonal @ offset 22 from the main diagonal has nonzero sum

ולכן, הטרנספורמציה ההופכית ל-$x \to x \oplus (x >> u)$ היא $x \to x \oplus (x >> 11) \oplus (x >> 22)$.:

Two down, Two to go!

הצעד השלישי

הצעד השלישי של ה-Tempering Transformation נראה כך:

```
out ^= (out << t) & uint32(c)
```

מציאת הטרנספורמציה ההופכית תהיה קצת יותר מסובכת, כי הפעם יש גם bit masking. כפי שפעלנו בשלבים הקודמים, נתחיל מלמצוא צורה מפורשת לכל ביט של התוצאה של הטרנספורמציה $y = x \oplus ((x << t)\&c)$. כל ביט של y מוגדר כך:

$$y_0 = x_0 \oplus ((x << t)\&c)_0$$
$$y_1 = x_1 \oplus ((x << t)\&c)_1$$
$$\vdots$$
$$y_{31} = x_{31} \oplus ((x << t)\&c)_{31}$$

עפ"י הגדרת left shift, ה-t ביטים אחרונים של $x << t$ הינם 0. AND עם 0 תמיד נותן 0, ולכן t הביטים האחרונים של $((x << t)\&c)$ הם 0. כפי שראינו XOR עם 0 תמיד נותן את הקלט עצמו, ולכן t הביטים האחרונים של y זהים ל-t הביטים האחרונים של x.

לכן, נוכל לכתוב את הביטים של y בצורה מפורשת כך:

$$y_0 = x_0 \oplus (x_{0+t} \& c_0)$$
$$y_1 = x_1 \oplus (x_{1+t} \& c_1)$$
$$\vdots$$
$$y_{31-t} = x_{31-t} \oplus (x_{31} \& c_{31-t})$$
$$y_{31-(t+1)} = x_{31-(t+1)}$$
$$\vdots$$
$$y_{31} = x_{31}$$

כעת, נמצא את המטריצה המייצגת T_1 של הטרנספורמציה. מכיוון ש-t הביטים האחרונים של y זהים ל-t הביטים האחרונים של x , ה-t עמודות האחרונות ב-T_1 הן t העמודות האחרונות במטריצת היחידה מסדר 32. לשאר הביטים, עושים XOR עם הביט שהוא t ביטים אחריהם, ב-Bitwise AND עם הביט המתאים ב-c.

לכן, לכל $i < 32 - t$, נשים 1 באינדקס (i, i) על מנת לקבל את הביט המקורי, ונשים את הביט המתאים של c באינדקס $(i, i + t)$. אם הביט של c הוא 0, נשים 0, מה שפשוט יחזיר את x_i כי נעשה XOR עם 0. אם הביט של c הוא 1, נעשה XOR לביט i של x עם הביט ה-i+t של x. כמו קודם,קל יותר לראות זאת עם מטריצות קטנות יותר. שוב נניח שה-word size הוא 8. המטריצה שמייצגת את הטרנספורמציה $y = ((x < 3) \& 0b10101100)$ היא:

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \end{pmatrix}$$

שוב, מומלץ לוודא ע"י כפל מטריצות שמטריצה זו אכן מייצגת את הטרנספורמציה הנ"ל. ובזאת, סיימנו את השלב הראשון! נמצא את המטריצה ההופכית באמצעות numpy:

```
import numpy as np
from numpy.linalg import inv

c = 0xEFC60000
t = 15

# Get the upper 15 bits of c
c_upper_bits = bin(c)[2:17]
amt_leading_zeroes = 32 - len(bin(c)[2:])
c_upper_bits = amt_leading_zeroes * "0" + c_upper_bits

x = np.eye(32, dtype=np.int32)
# The diagonal at offset -17 from the main diagonal is filled with the upper 15
# bits of C
y = np.diag([ord(x) - ord('0') for x in list(c_upper_bits)], -17)
t3 = x + y
```

הערה קטנה לגבי הקוד: ה-slicing בהגדרת c_upper_bits הוא מכיוון שהפונקציה bin מחזירה את כל הביטים של המספר בפורמט: 0b.

אז צריך להתעלם מה-0b בהתחלה. לאחר מכן, מוסיפים leading zeros כי bin לא עושה את זה בשבילנו, ואז יוצרים את T_3 שמייצגת את הצעד השלישי, בעזרת חיבור של מטריצת היחידה, עם חיבור של מטריצה y, שמלאה באפסים, חוץ מאלכסון אחד ב-offset של $32 - t = 17$ מתחת לאלכסון הראשי בביטים הגבוהים של c.

הפעם, לא נוכל להסתפק בלהדפיס אילו אלכסונים הם לא 0, ונצטרך גם להדפיס את האלכסונים עצמם, מכיוון שמעורבות פה גם bitwise masks. לשם כך, נשתמש בקוד הבא:

```
t3_inv = inv(t3)

for i in range(-32, 32):
    curr_diag = np.diagonal(t3_inv, i)

    if curr_diag.sum() != 0:
        print("Diagonal @ offset {} is {}".format(i, curr_diag))
```

הקוד מדפיס:

```
Diagonal @ offset -17 is [-1. -1. -1. 0. -1. -1. -1. -1. -1. -1. 0. 0. 0. -1. -1.]
Diagonal @ offset 0 is [1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
```

מאיפה הגיעו המספרים השליליים האלו?! זכרו שאנחנו פועלים מעל השדה הסופי $\mathbb{Z}_2$, שכולל רק 2 איברים: 0 ו-1, אבל לא אמרנו את זה ל-numpy, אז numpy חושב שאנחנו פועלים מעל הממשיים. נתייחס לכל -1 כ-1, מכיוון שב-$\mathbb{Z}_2$ 1 הוא הנגדי של עצמו, כלומר $1 \oplus 1 = 0$, ולכן $-1 = 1$.

מכיוון שהאלכסון שנמצא ב-offset של 17 מתחת לאלכסון הראשי הוא לא 0, הטרנספורמציה ההופכית ל-T_3 היא מהצורה $y = x \oplus ((x << 15)\&p)$, כאשר p הוא קבוע כלשהו. על מנת למצוא את p, נזכור שהתוכן של האלכסון הראשי הוא הביטים הגבוהים של ה-bitmask. לכן, p הוא המספר ש-17 הביטים הגבוהים שלו הם אלו שכתובים באלכסון, ושאר הביטים שלו הם 0. נקבל את המספר הבא:

11101111110001100000000000000000

בבסיס 16: 0xEFC60000, שזה בדיוק c! לכן, ההופכית של הטרנספורמציה T_3 היא T_3 עצמה: $y = x \oplus ((x << t)\&c)$.

הצעד השני (והאחרון)

מכיוון שהצעד השני הוא מהצורה:

```
out ^= (out << s) & uint32(b)
```

כל שנצטרך לעשות הוא לערוך את הקוד שהשתמשנו בו בצעד הקודם, כך שהוא ישתמש ב-s וב-b במקום ב-t וב-c, בהתאמה:

```
import numpy as np
from numpy.linalg import inv

b = 0x9D2C5680
s = 7

# Get the upper 25 bits of b
b_upper_bits = bin(b)[2:27]
amt_leading_zeroes = 32 - len(bin(b)[2:])
b_upper_bits = amt_leading_zeroes * "0" + b_upper_bits

x = np.eye(32, dtype=np.int32)
# The diagonal at offset -7  from the main diagonal is filled with the upper 32-7=25
# bits of b
y = np.diag([ord(x) - ord('0') for x in list(b_upper_bits)], -7)
t2 = x + y
```

נהפוך את המטריצה ונדפיס את האלכסונים:

```
t2_inv = inv(t2)

for i in range(-32, 32):
    curr_diag = np.diagonal(t2_inv, i)

    if curr_diag.sum() != 0:
        print("Diagonal @ offset {} is {}".format(i, curr_diag))
```

התוצאה:

```
Diagonal @ offset -28 is [0. 0. 0. 1.]
Diagonal @ offset -21 is [ 0. 0. 0. -1. 0. -1. 0. 0. 0. 0. -1.]
Diagonal @ offset -14 is [1. 0. 0. 1. 0. 1. 0. 0. 0. 0. 1. 0. 1. 0. 0. 0. 0. 1.]
Diagonal @ offset -7 is [-1. 0. 0. -1. -1. -1. 0. -1. 0. 0. -1. 0. -1. -1. 0. 0. 0. -1. 0. -1. 0. -1. -1. 0. -1.]
Diagonal @ offset 0 is [1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
```

נמיר את ה-bitmasks לבסיס 16:

Offset -28 (<< 28): 0b00010000000000000000000000000000 => 0x10000000

Offset -21 (<< 21): 0b00010100001000000000000000000000 => 0x14200000

Offset -14 (<< 14): 0b10010100001010000100000000000000 => 0x94284000

Offset -7 (<< 7): 0b10011101001011000101011010000000 => 0x9D2C5680

ולכן, ההופכי של השלב השני T_2 הוא:

y = x ^ ((x << 28) & 0x10000000) ^ ((x << 21) & 0x14200000) ^ ((x << 14) & 0x94284000) ^ ((x << 7) & 0x9D2C5680)

Putting It All Together

סיימנו להפוך את כל הצעדים! כעת, בשביל להחזיר את ה-state בהינתן הפלט של ה-PRNG, נפעיל את השלבים ההופכיים בסדר הפוך (כלומר קודם ההופכי לשלב 4 וכן הלאה):

```
// Given some output from the PRNG, restore the corresponding element in the state
// e.g. given output 5, we can restore element 5 (or 4 if we're using zero-based indexing)
func restore_state(out uint32) uint32 {
// MT generates the output by applying an *invertible* tempering
// transformation to the state element
tempered_state := out
// Inverse of "out ^= out >> uint32(l)"
tempered_state ^= tempered_state >> uint32(l)
// Inverse of "out ^= (out << t) & uint32(c)"
tempered_state ^= (tempered_state << t) & uint32(c)
// Inverse of "out ^= (out << s) & uint32(b)"
tempered_state ^= ((tempered_state << 28) & 0x10000000) ^
((tempered_state << 21) & 0x14200000) ^
((tempered_state << 14) & 0x94284000) ^
((tempered_state << s) & uint32(b))
// Inverse of "out ^= out >> uint32(u)"
original_state := tempered_state ^ (tempered_state >> 11) ^ (tempered_state >> 22)

return original_state
}
```

בעזרת הקוד הזה, ו-624 פלטים של ה-PRNG (כי ה-state הוא בגודל 624), ניתן לדעת מה יהיו כל הפלטים העתידיים (והקודמים) של ה-PRNG! על מנת להדגים זאת, ניצור "Guessing Game" קצר שמשתמש ב-PRNG שכתבנו ומבקש מהמשתמש לנחש מספר רנדומלי מאפס עד מיליון 10 פעמים.

בנוסף, ה-Guessing Game נותן לנו leak של ה-624 פלטים הראשונים של ה-PRNG לתוך קובץ:

```
func main() {
        my_mt := mtlib.NewMt(uint32(time.Now().UnixMicro()))
        win_cnt := 0

        fmt.Println("Welcome to the guessing game!")
        fmt.Println("To win, guess a random number 10 times")
        fmt.Println("To help you, I wrote the first 624 numbers generated by the PRNG to a fil

        f, err := os.Create("./leak.txt")

        if err != nil {
                panic(err)
        }

        // Close after we return from main
        defer f.Close()

        for i := 0; i < 624; i++ {
                to_write := fmt.Sprintf("%d\n", my_mt.GenNext())

                f.WriteString(to_write)
        }

        fmt.Printf("Let's start!\n")

        for i := 0; i < 10; i++ {
                my_num := my_mt.GenNext() % 1000000
                var user_num uint32

                fmt.Printf("Guess which number I thought of: ")
                fmt.Scanf("%d\n", &user_num)

                fmt.Println("Correct!")

                if my_num != user_num {
                        fmt.Println("Incorrect. Bye!")
                        os.Exit(0)
                }

                win_cnt += 1
        }

        fmt.Println("Congratulations!")
}
```

הקוד עושה seed ל-PRNG עם ה-unix timestamp הנוכחי במיקרו-שניות, כפי שנפוץ בהרבה תוכנות שמייצרות מספרים רנדומליים. לאחר מכן, הוא משתמש ב-PRNG על מנת לייצר 624 מספרים, וכותב אותם לקובץ leak.txt.

לבסוף הוא נכנס ללופ שבו הוא מבקש מהמשתמש לנחש את המספר הרנדומלי 10 פעמים.

בלי לפצח את ה-PRNG, הסיכויים שלנו לנצח הם מאוד קטנים: יש סיכוי של אחד למיליון לנחש נכון, ויש לעשות זאת 10 פעמים, מה שנותן הסתברות של אחד ל-10^{60} לנצח. למזלינו, בעזרת ה-leak, אנחנו יכולים לשבור את ה-PRNG לגמרי, ולנחש בדיוק את סדרת המספרים שהתוכנה תבקש מאיתנו לנחש. לשם כך, נשתמש בתוכנה הבאה:

```
func main() {
        var my_state [624]uint32
        state_idx := 0
        leak_path := os.Args[1]

        fmt.Println("Guessing Game Solver")
        fmt.Println("Specify the path of the leak file as a command line arg")

        f, err := os.Open(leak_path)

        check(err)

        // Initialize a scanner to read the file line-by-line
        fScanner := bufio.NewScanner(f)
        fScanner.Split(bufio.ScanLines)

        my_state[0] = 0

        // Each line in the file is a leak of a number generated by the PRNG
        for fScanner.Scan() {
                // Convert to a number
                leak, err := strconv.Atoi(fScanner.Text())
                check(err)
                // Crack the corresponding state element
                my_state[state_idx] = mtlib.RestoreState(uint32(leak))

                state_idx += 1
        }

        my_mt := mtlib.MtFromState(my_state)

        for i := 0; i < 10; i++ {
                fmt.Printf("Next number: %d\n", my_mt.GenNext()%1000000)
        }
}
```

תוכנה זו קוראת את הקובץ של ה-leak, ומשתמשת בפונקציות שכתבנו מקודם על מנת לעשות restore ל-state של ה-PRNG. לאחר מכן, היא כותבת את המספרים הבאים ב-Guessing Game למשתמש. לצערי, אין לי דרך לכלול GIF שמראה כיצד התוכנה פותרת את ה-Guessing Game, אז אני פשוט אראה screenshots מה-terminal. למי שמעוניין, העליתי ל-Youtube סרטון שמראה את זה בלייב ☺

בואו נתחיל את המשחק:

```
(.venv) ➜  game go run game.go
Welcome to the guessing game!
To win, guess a random number 1000 times
To help you, I wrote the first 624 numbers generated by the PRNG to a file named leak.txt
Let's start!
Guess which number I thought of:
```

נריץ את ה-solver שכתבנו:

```
(.venv) ➜  solver go run solver.go ../game/leak.txt
Guessing Game Solver
Specify the path of the leak file as a command line arg
Next number: 223291
Next number: 302325
Next number: 71736
Next number: 479949
Next number: 932525
Next number: 427668
Next number: 71273
Next number: 365423
Next number: 663664
Next number: 801339
```

ننסה את המספרים הללו במשחק:

```
Let's start!
Guess which number I thought of: 223291
Correct!
Guess which number I thought of: 302325
Correct!
Guess which number I thought of: 71736
Correct!
Guess which number I thought of: 479949
Correct!
Guess which number I thought of: 932525
Correct!
Guess which number I thought of: 427668
Correct!
Guess which number I thought of: 71273
Correct!
Guess which number I thought of: 365423
Correct!
Guess which number I thought of: 663664
Correct!
Guess which number I thought of: 801339
Correct!
Congratulations!
```

וניצחנו את המשחק!

סיכום

במאמר זה, ראינו מדוע שימוש של PRNG שהוא לא בטוח-קריפטוגרפית (CSPRNG) לכל דבר שקשור באבטחה הוא רעיון רע: תוקף יכול לדעת מה יהיו כל הפלטים של ה-PRNG בעבר ובעתיד עם leak יחסית קטן מה-PRNG. שאלה שיכולה לעלות לאחר הפוסט היא "האם יש דרך בטוחה להשתמש ב-MT?".

ניתן, לדוגמה, לעשות hash לפלטים של ה-PRNG. אם משתמשים בפונקציית hash טובה, יהיה בלתי אפשרי בפועל עבור תוקף למצוא את הפלט המקורי של ה-PRNG מהפלט של ה-hash function (במיוחד כשמדברים על מספרים של 64 ביט), ולכן תוקף לא יוכל לבצע את הטרנספורמציה ההופכית בקלות.

על הכותב

שמי יוראי הרצברג. אני סטודנט שנה רביעית למדמ"ח באוניברסיטה הפתוחה, ומתעניין מאוד ב-Security וב-AI. אני אוהב לשחק ב-CTF-ים, לעבוד על פרוייקטים, ולעשות Bug Bounty. יש לי בלוג שבו אני כותב פוסטים על הנושאים הנ"ל :), המאמר הזה פורסם במקור כפוסט בבלוג, אתם מוזמנים לבקר בו ולהנות ☺

אלו הפרטים שלי:

- ה-Linkedin שלי: https://www.linkedin.com/in/yoray-herzberg-b8155621b/
- ה-GitHub שלי: https://github.com/vaktibabat
- ה-Twitter שלי: https://x.com/sag0li

מקורות

Makoto Matsumoto and Takuji Nishimura. 1998. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. ACM Trans. Model. Comput. Simul. 8, 1 (Jan. 1998), 3–30 - המאמר המקורי על MT

https://vaktibabat.github.io/posts/PRNG_In_Go/ - הפוסט המקורי שכתבתי