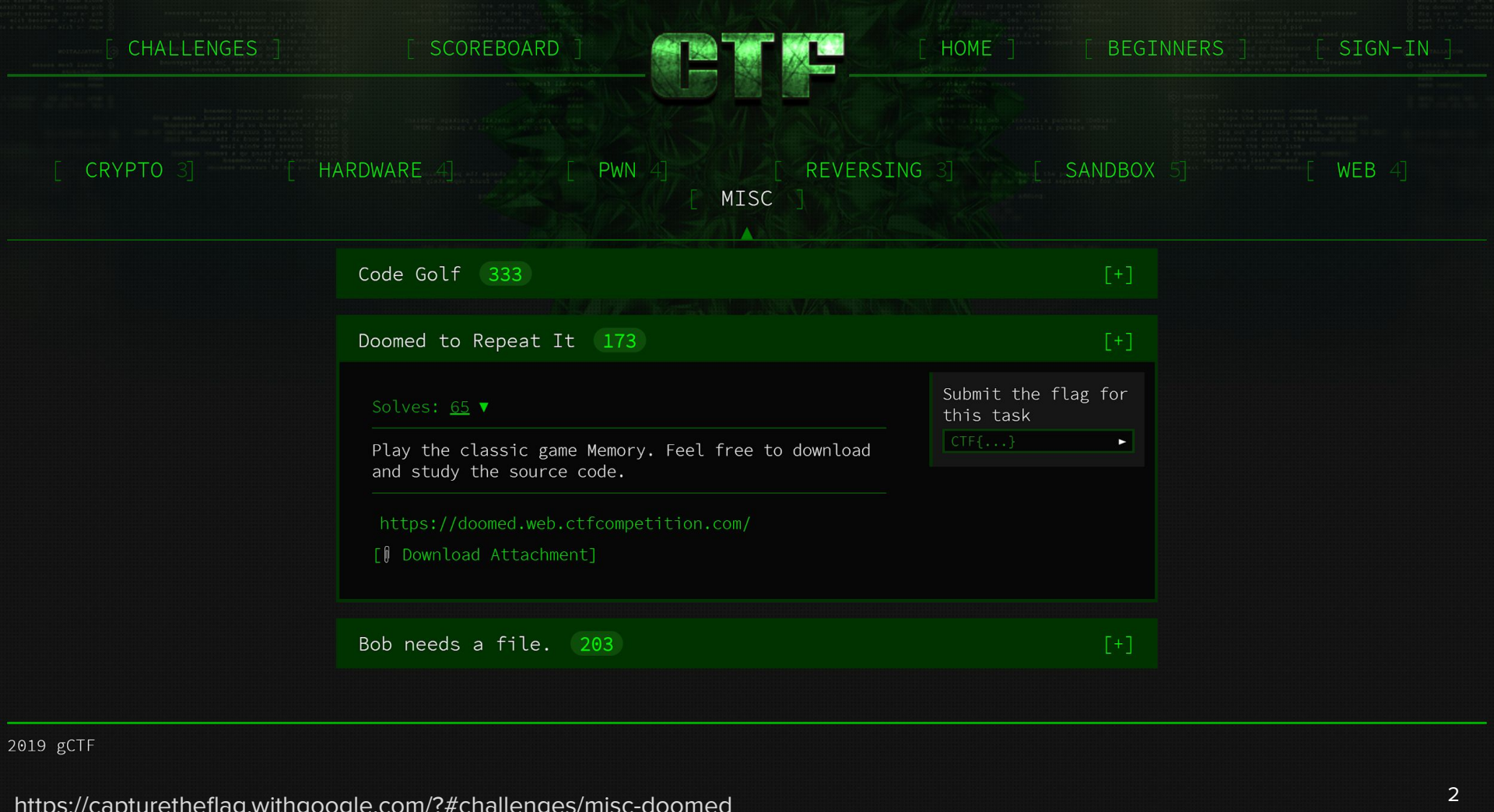
Doomed to Repeat It

Google CTF 2019 (Quals)

Michael Reisigl
2019-12-17



https://capturetheflag.withgoogle.com/?#challenges/misc-doomed

The game

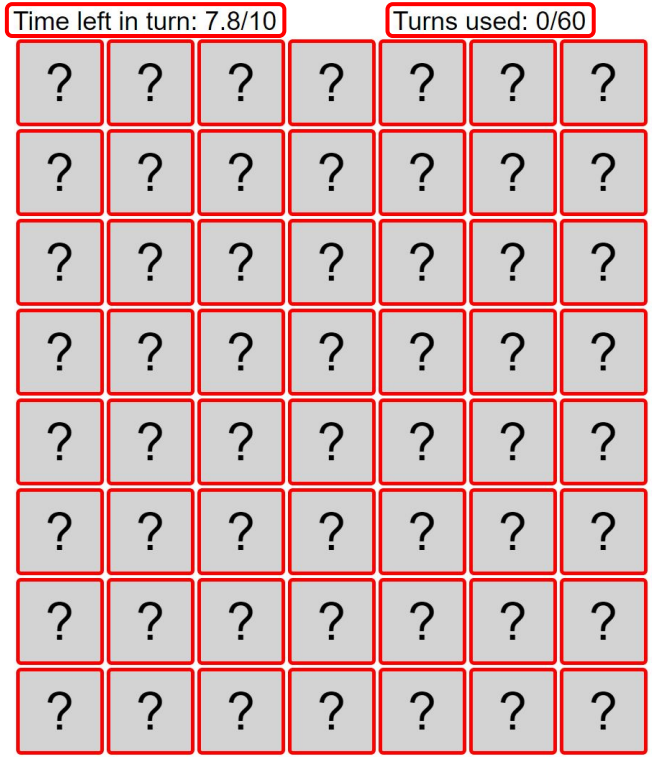
Welcome to the classic game Memory.

Rules are simple: pick an unsolved tile, then pick the matching tile. Solve all the tiles to win!

You are limited in how many guesses you can make, and in how much time per guess. Guess well!

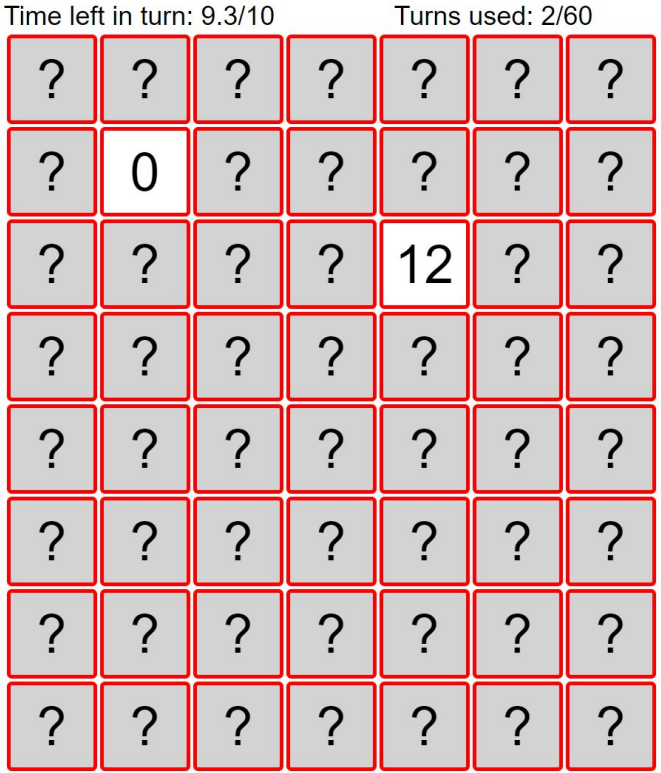
Play

The game



- 10 seconds time per guess
- 60 guesses overall

The game



- See two tiles at a time
- They stay face up, if they match
- Goal: find all the pairs?

Demo - Game

Analyze

- 10 seconds time per guess, should not be a problem
- 7 × 8 tiles = 56 tiles overall
- Just memorize all flipped tiles?
- Recap: 60 guesses available overall

➜ We can guess only 4 times wrong, gotta be lucky

Analyze traffic: json over websocket

➔ Request *info*:

```
{
    "op": "info"
}
```

➔ Response *info*:

```
{
    "width": 7,
    "board": [
        -1,
        ...
        -1,
        -1
    ],
    "maxTurns": 60,
    "maxTurnTime": 10,
    "turnsUsed": 0,
    "done": false,
    "message": "",
    "clear": []
}
```

➔ Request *guess*:

```
{
    "op": "guess",
    "body": {
        "x": 1,
        "y": 0
    }
}
```

➔ Response *guess*:

```
{
    "width": 7,
    "board": [
        -1,
        25,
        -1,
        ...
        -1,
        -1
    ],
    "maxTurns": 60,
    "maxTurnTime": 10,
    "turnsUsed": 1,
    "done": false,
    "message": "",
    "clear": []
}
```

Let’s have a look at the source code...

- INSTALL.md
- app.yaml
- main.go
- game
  - game.go
- random
  - random.go
- static
  - game.html
  - game.js
  - index.html
  - style.css

Some important parts

Source code - INSTALL.md

Memory

Memory is a fun and secure web game. Players who beat it get a little prize. ➜ The flag?

Memory is written in go because go is memory-safe, which is very important for a game like Memory with strict safety requirements.

[...]

let's see how secure it is...

Source code - main.go

[...]

```
98  func main() {
99      // flag.txt should have no newline
100     flag, err := ioutil.ReadFile("flag.txt")
101     if err != nil {
102         log.Fatalf("Couldn't read flag: %v", err)
103     }
104
105     http.HandleFunc("/_ah/health", healthCheckHandler)
106     http.Handle("/", &rootHandler{flag: string(flag)})
107     log.Print("Listening on port 8080")
108     log.Fatal(http.ListenAndServe(":8080", nil))
109 }
```

Source code - main.go

```
77  func (h *rootHandler) ServeHTTP(w http.ResponseWriter, r *http.Request) {
78      // TODO: What about DNS rebinding attacks?
79      if needsHttpsRedir(r) {
80          // A shallow copy of URL is fine because it's not being retained.
81          newUrl := *r.URL
82          newUrl.Host = r.Host
83          newUrl.Scheme = "https"
84          http.Redirect(w, r, newUrl.String(), 302)
85          return
86      }
87      if r.URL.Path == "/ws" {
88          h.handleWs(w, r)
89          return
90      }
91      staticHandler.ServeHTTP(w, r)
92  }
```

[...]

[...]

```
68  func (h *rootHandler) handleWs(w http.ResponseWriter, r *http.Request) {
69      conn, err := upgrader.Upgrade(w, r, nil)
70      if err != nil {
71          log.Printf("Couldn't upgrade: %v", err)
72          return
73      }
74      game.Run(conn, h.flag)
75  }
```

Upgrade http connection to websocket protocol

Source code - game.go

[...]

```go
// Run runs the game for a single user who is attached to conn.
// conn must be non-nil. conn will be closed when done.
func Run(conn *websocket.Conn, flag string) {
    defer conn.Close()
    board, err := newBoard()
    if err != nil {
        log.Printf("Couldn't create board: %v", err)
        return
    }
```

[...]

```go
func newBoard() (*board, error) {
    rand, err := random.New()
    if err != nil {
        return nil, fmt.Errorf("couldn't create random: %v", err)
    }
    b := &board{
        nums:    make([]int, BoardSize),
        visible: make([]bool, BoardSize),
    }
    // BoardSize is even
    for i, _ := range b.nums {
        b.nums[i] = i / 2
    }
    // https://github.com/golang/go/wiki/SliceTricks#shuffling
    for i := BoardSize - 1; i > 0; i-- {
        j := rand.UInt64n(uint64(i) + 1)
        b.nums[i], b.nums[j] = b.nums[j], b.nums[i]
    }
    return b, nil
}
```

looks interesting

Source code - game.go

[...] (function Run)

```
159             if board.nums[index] == board.nums[oldIndex] {
160                 // Correct.
161                 boardForResp = board.forResp()
162                 foundNum++
163                 if foundNum*2 == BoardSize {
164                     done = true
165                     message = fmt.Sprintf("You win! Flag: %s", flag)
166                 }
167             } else {
```

[...]

So yes, we just have to win the game to get the flag...

Demo - Source code

Source code - random.go

[...]

```
56  // New generates state for a new random stream with cryptographically secure
57  // randomness.
58  func New() (*Rand, error) {
59      osr, err := OsRand()
60      if err != nil {
61          return nil, fmt.Errorf("couldn't get OS randomness: %v", err)
62      }
63      return NewFromRawSeed(osr)
64  }
```

generate a random seed

[...]

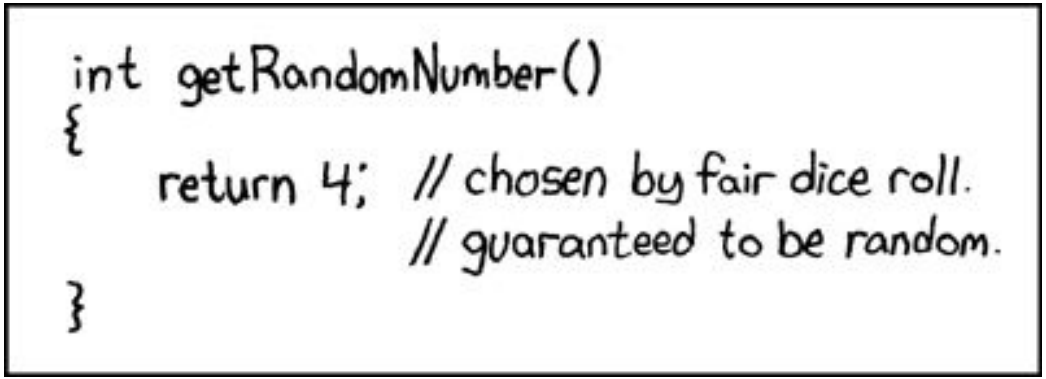
[...]

```
22  // OsRand gets some randomness from the OS.
23  func OsRand() (uint64, error) {
24      // 64 ought to be enough for anybody
25      var res uint64
26      if err := binary.Read(rand.Reader, binary.LittleEndian, &res); err != nil {
27          return 0, fmt.Errorf("couldn't read random uint64: %v", err)
28      }
29      // Mix in some of our own pre-generated randomness in case the OS runs low.
30      // See Mining Your Ps and Qs for details.
31      res *= 14496946463017271296
32      return res, nil
33  }
```

looks and sounds suspicious

[...]

Is this maybe something like...?



https://xkcd.com/221/

Analyze the (P)RNG

- Secure random number in a 64 bit unsigned integer variable
- Multiplied by: 1449694646301727 1296
  - It's prime factorisation is: $2^{47} \times 103007$
  - Binary representation: 1100100100101111100000000000000000000000000000000000000000000000
- What does that mean?
  - Suppose the following random number: 13135265590706583189
  - Binary representation: 1011011001001001110101100111011000001100011111010001101101001010 1
  - Multiplied by the *pre-generated random number*:

    1110110100101100000000000000000000000000000000000000000000000000
- The multiplication eliminates the 47 least significant bits

What do we gain?

- Instead of having 2^{64} possible seeds
- We now have 2^{64-47} possible seeds
- Those are 2^{17} = 131072 possible layouts
- Which is totally feasible

Exploit idea

- First pre-generate all the 131072 layouts
  - Reuse the go program for it
- Flip four memory tiles
- Find the correct layout in the pre-generated layouts
- *Guess* all the tiles
- Profit?

Pre-generate memory layouts

random.go

```go
// New generates state for a new random stream with cryptographically secure
// randomness.
func New(rawSeed uint64) (*Rand, error) {
    //osr, err := OsRand()
    //if err != nil {
    //  return nil, fmt.Errorf("couldn'
    //}
    return NewFromRawSeed(rawSeed)
}
```

use our custom seed

main.go

```go
var i uint64
for i = 0; i < 1<<17; i++ {
    rand, err := random.New(i << 47) // use custom seed
    if err != nil {
        log.Printf("couldn't create random: %v", err)
        return
    }
    boardLayout := make([]int, BoardSize)
    for i, _ := range boardLayout {
        boardLayout[i] = i / 2
    }
    for i := BoardSize - 1; i > 0; i-- {
        j := rand.UInt64n(uint64(i) + 1)
        boardLayout[i], boardLayout[j] = boardLayout[j], boardLayout[i]
    }
    board := strings.Trim(strings.Join(strings.Fields(fmt.Sprint(boardLayout)), " "), "[]")
    if _, err = file.WriteString(board + "\n"); err != nil {
        log.Printf("couldn't write to file: %v", err)
        return
    }
}
```

write board layouts to file

Let’s write the exploit in python

- Load the board layouts in a dictionary
  - The first four tiles are our key
- Establish a connection to the websocket
- Flip the first four tiles
- Find the valid board
- Solve the memory game
- Get the flag :)

Demo - Exploit

Challenge solved

You win! Flag: CTF{PastPerf0rmanceIsIndicativeOfFutureResults}

Impact of the security threat

- In this context: good memory player :)
- In general: predictable keys for crypto
  - RSA keys
  - Initialisation vectors
  - MAC keys
  - Nones
  - Everything which expects true randomness...
- CVE-2008-0166 - Weak Debian OpenSSL Keys

Possible countermeasures

- Don’t do RNG yourself
- Use a secure random number generator
- Don’t do modulo multiplication on a random number
  - E.g on a 64 bit integer
- In any case, don’t do it with a large power of two
  - Least significant bits are eliminated

Thank you for your attention!

Questions?

References

- Used writeup: https://ctftime.org/writeup/15815