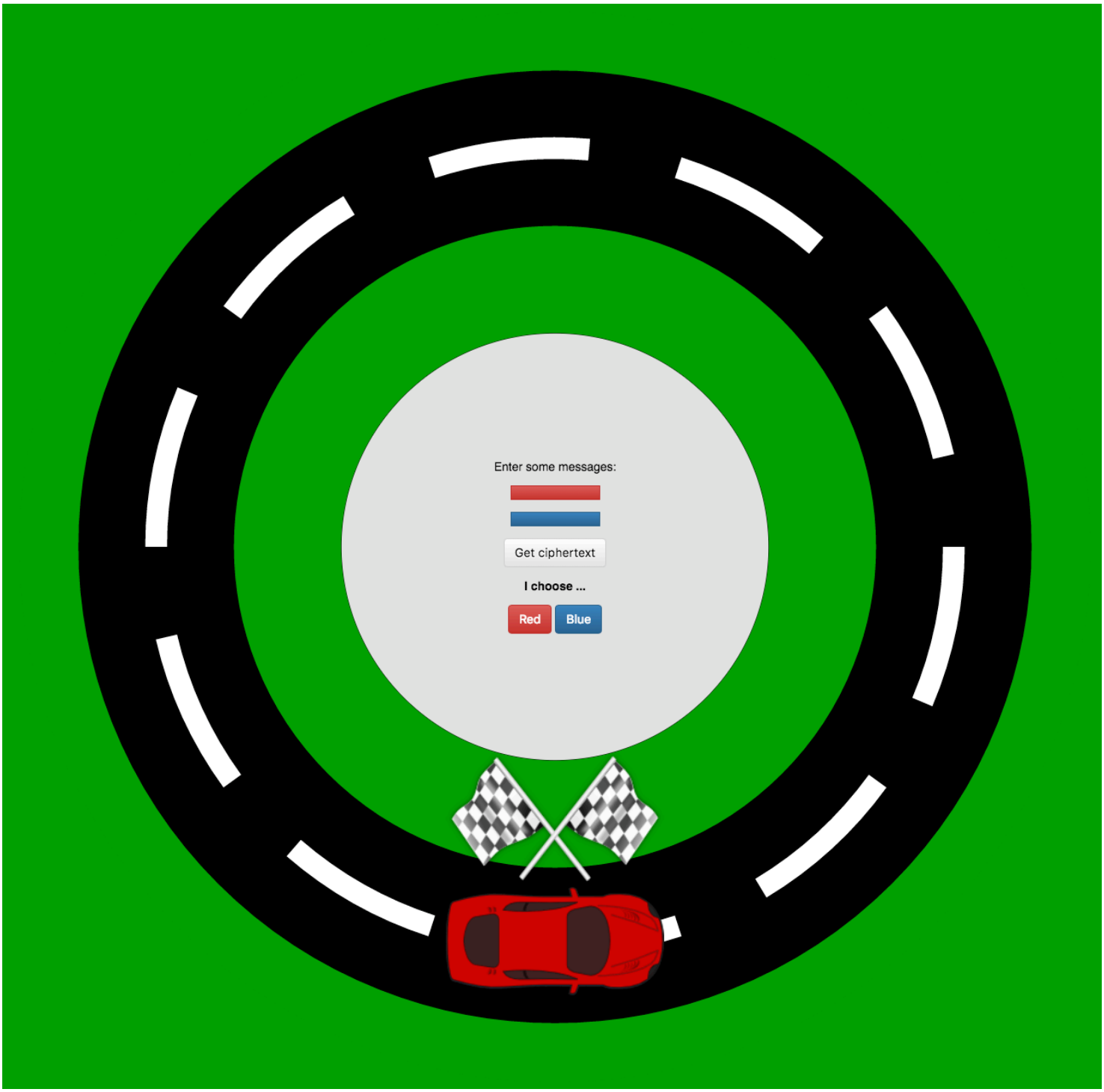
HackOver 2016 - 9soc writeup - KernelSanders

semsecrace - crypto 15

Win the race on the data autobahn! Since autonomous driving is the future of traffic 2.0 you just have a red and a blue button to drive your cyber car. No steering wheel, no gas pedal, no ancient clutch

Challenge website:



On clicking either the "Red" or "Blue" button, your car either changes to that color and moves 1/40th the way round the circle, or bursts into flames and displays "You took the wrong color."

Code review

Lets start with the web page. Looking at the javascript, the website is just a fancy wrapper around a POST request to `/ciphertext`. The "state" must be handled on the back end and the flag will be displayed if the

back end returns it as the `action`. The code below is where most of the magic happens on the front end.

```JavaScript
function takeColor(color) {
    var req = new XMLHttpRequest();
    req.onreadystatechange = function() {
        if (req.readyState == 4 && req.status == 200) {
            var resp = JSON.parse(req.responseText);
            if (resp["action"] != "flag") {
                changeState(resp["action"]);
            } else {
                changeState("flag");
                displayGameOver(resp["text"]);
            }
        }
    }
    req.open("POST", "/choose?driver_license=" + state["license"] + "&color=" + color);
    req.send();
}
```

The backend is more confusing. Starting at the `main` function, we can see it defines a few handles for different "pages" and then listens on a port to serve them up. Standard REST stuff.

```Go
func main() {
    var err error
    tmpl, err = template.ParseFiles("semsecrace.html")
    if err != nil {
        log.Fatal("Could not load template", err)
    }

    go cleaner()
    http.HandleFunc("/", handleIndex)
    http.HandleFunc("/race", handleRace)
    http.HandleFunc("/choose", handleChooseColor)
    http.HandleFunc("/ciphertext", handleGetCiphertext)
    http.Handle("/static/", http.StripPrefix("/static/", http.FileServer(http.Dir("./stat

    log.Fatal(http.ListenAndServe(":8202", nil))
}
```

Lets look at these one at a time to figure out what the backend is doing.

- `handleIndex` just redirects to `/race` after generating a new `state`.

```go
func handleIndex(w http.ResponseWriter, r *http.Request) {
    s := NewState()
    http.Redirect(w, r, "/race?driver_license="+s.EncodeCode(), http.StatusFound)
}
```

What is a `state` ? Its defined at the top:

```go
type State struct {
    Code       string
    Stage      uint8
    Colors      []byte
    ValidUntil time.Time
}
```

So there is some string `Code` then the `Stage` we are on, a byte array of `Colors` and what we can assume is a time that our "driver_license" expires, `ValidUntil` .

- `handleRace` calls `getState` redirects to `/` if the state is bad, and servers the template page.

```go
func handleRace(w http.ResponseWriter, r *http.Request) {
    s := getState(r.FormValue("driver_license"))
    if s == nil {
        http.Redirect(w, r, "/", http.StatusFound)
        return
    }
    err := tmpl.ExecuteTemplate(w, "semsecrace.html", struct {
        DriverLicense string
        NumStages     int
    }{s.EncodeCode(), NUM_STAGES})
    if err != nil {
        log.Fatal("Could not execute template", err)
    }
}
```

Lets look at `getState`

```go
func getState(codeBase32 string) *State {
    code, err := base32.StdEncoding.DecodeString(codeBase32)
    if err != nil {
        return nil
    }
    s := validStates[string(code)]
    if s == nil {
        return nil
    }
    if s.IsExpired() {
        return nil
    }
    s.Activity()
    return s
}
```

Line 2 takes a Base32 string and decodes it back to a bytearray, then this function returns `s` (line 14) if it can find the state in the `validStates` array. So `validStates` must be an array that holds `state` objects and is indexed by their `Code` values (aka `drivers_license`). Looking back to the top of the go file our guess is confirmed:

```go
var validStates = make(map[string]*State)
```

- `handleChooseColor` is what takes the POST from the javascript we noticed earlier and determines if you chose the correct color for that stage.

```go
func handleChooseColor(w http.ResponseWriter, r *http.Request) {
    w.Header().Set("Content-Type", "application/json")

    s := getState(r.FormValue("driver_license"))
    if s == nil {
        json.NewEncoder(w).Encode(JsonResponse{Action: "expired"})
        return
    }

    var resp JsonResponse
    color := r.FormValue("color")
    if s.GetColor() == color {
        s.Stage++
        if s.Stage >= NUM_STAGES {
            resp = JsonResponse{Action: "flag", Text: fmt.Sprintf("Winner!<br>%s", FL
            s.Burn()
        } else {
            resp = JsonResponse{Action: color}
        }
    } else {
        s.Burn()
        resp = JsonResponse{Action: "burn", Text: "You took the wrong color."}
    }
    json.NewEncoder(w).Encode(resp)
}
```

Again, this calls `getState` with our `drivers_license` / `Code` to pull our state from the `validStates` array. Then if we clicked the color that `s.GetColor()` returns (line 11) it will increment the stage number (line 13) and if this was the last stage send the front end the flag (line 15), or else return the color we just chose to the front end.

Lets look at `s.GetColor()` as this is the function that controls if we get the flag or not.

```go
func (s *State) GetColor() string {
    numBit := s.Stage % 8
    colorPos := s.Stage / 8
    if (s.Colors[colorPos]>>numBit)&0x1 == 0 {
        return "red"
    } else {
        return "blue"
    }
}
```

Now this is more like a crypto challenge! Some bitwise math to determine the color at each stage. So the color for each stage is determined by if the bit at the `stage` 's `Colors` array in position `Stage / 8` shifted right by `Stage % 8` bits and `and` 'd with `0x1` is equal to `0` . This seems pretty complicated, but not really crypto yet. It turns out fully understanding how this works isn't critical to solving the challenge. At this stage we have the code so we could treat this function as a "black box" so long as we have the proper parameters. For us to determine the color for ours stage we need the `Stage` integer and the `Colors` array. `Stage` is pretty easy, we can just keep track of which stage we are on, assuming it starts at `0` . Where is a new `Stage` defined? In `New State` :

```go
func NewState() *State {
    code := make([]byte, 20)
    if _, err := io.ReadFull(rand.Reader, code); err != nil {
        log.Fatal("Oh no! Out of randomness for state code")
    }
    colors := make([]byte, NUM_STAGES/8+1)
    if _, err := io.ReadFull(rand.Reader, colors); err != nil {
        log.Fatal("Oh no! Out of randomness for colors")
    }
    s := &State{Code: string(code), Stage: 0, Colors: colors}
    s.Activity()
    validStateMutex.Lock()
    defer validStateMutex.Unlock()
    validStates[s.Code] = s
    return s
}
```

Line 10 sets the stage number for new states to `0` . Also interesting here is that `Colors` is set to `colors` in a new state, and it looks like `colors` is a random string of bytes. Now this is starting to feel like a crypto challenge. Maybe this is a weak Pseudo Random Number Generator (PRNG) problem? Lets dig into the Go docs to see how `rand.Reader` works. If there is a seed we can determine perhaps we can run our own version of this in parallel with the back end and determine which colors to

choose. The docs state that rand.Reader calls `getrandom()` and the man page for `getrandom()` states, "getrandom() relies on entropy gathered from device drivers and other sources of environmental noise." Yikes. I don't think we are going to sync our getrandom() calls with the server and attack the PRNG side of this problem. Lets keep looking.

- `handleGetCiphertext` takes two values from the front end, and based on the output of `s.GetColor()` encrypts and returns one of them.

```go
func handleGetCiphertext(w http.ResponseWriter, r *http.Request) {
    s := getState(r.FormValue("driver_license"))
    if s == nil {
        w.Header().Set("Content-Type", "text/plain")
        w.WriteHeader(http.StatusForbidden)
        fmt.Fprintln(w, "Your driver license expired. Try again.")
        return
    }

    m0 := []byte(r.FormValue("m0"))
    m1 := []byte(r.FormValue("m1"))

    if len(m0) != len(m1) {
        w.Header().Set("Content-Type", "text/plain")
        w.WriteHeader(http.StatusForbidden)
        fmt.Fprintln(w, "We are in the semantic security race. Follow the rules!")
        return
    }

    var msg []byte
    if s.GetColor() == "red" {
        msg = m0
    } else {
        msg = m1
    }
    w.Header().Set("Content-Type", "application/octet-stream")
    w.Write(encrypt(msg))
}
```

We didn't see any calls to this in the javascript, lets look at the page again. Right clicking and inspecting the "Get ciphertext" button takes us right to this HTML:

```Markup
<form method="post" action="/ciphertext">
    <input type="hidden" name="driver_license" value="{{.DriverLicense}}">
    <p>Enter some messages:</p>
    <p><input type="text" name="m0" class="m0" size="16"></p>
    <p><input type="text" name="m1" class="m1" size="16"></p>
    <button class="btn btn-white">Get ciphertext</button>
</form>
```

This is how we will determine which color to choose at each stage! The `handleCipherText` function calls `s.GetColor()` for us and returns the encrypted form of either `m0` or `m1`, which we control, based on the color we should choose. Since we control the plaintext, this could be a known-plaintext attack. Lets look at the `encrypt()` function to see if this is a feasible attack.

```Go
 func encrypt(message []byte) []byte {
    key := make([]byte, aes.BlockSize)
    if _, err := io.ReadFull(rand.Reader, key); err != nil {
        log.Fatal("Oh no! Out of randomness for key")
    }
    block, _ := aes.NewCipher(key)
    someByte := byte(aes.BlockSize - (len(message) % aes.BlockSize))
    for i := byte(0); i < someByte; i++ {
        message = append(message, someByte)
    }
    ciphertext := make([]byte, len(message))
    for i := 0; i < len(message) / aes.BlockSize; i++ {
        src := message[i*aes.BlockSize:(i+1)*aes.BlockSize]
        dst := ciphertext[i*aes.BlockSize:(i+1)*aes.BlockSize]
        block.Encrypt(dst, src)
    }
    return ciphertext
}
```

Damn. AES is not susceptible to known-plaintext attacks, and each time this function is called a new random key is created. We could try to brute force the key each stage since we know the plaintext has to be one of two strings we provide, but on a modern computer that would take roughly 149 billion years per stage (looking at the Go documentation `aes.BlockSize` is 16 bytes, which when used in `aes.NewCipher` causes it to select AES-128 as the cipher).

Looking at the Go documentation, `aes.NewCipher` makes no mention of what **mode** AES will operate in. Lines 7-9 are padding our message to fit the `aes.Blocksize` with the number of bytes of padding. This sounds like a perfect set up for a padding oracle attack. But padding oracle only works if we are in

CBC mode.

Digging into the `cipher.Block` documentation, it looks like the programmer has to specify `NewCBCEncrypter` to use CBC mode. Most AES crypto libraries will default to ECB mode if no mode is specified. Looking at the ECB wiki page, there is a nice example of the weakness of ECB. We can detect patterns in large samples of data well enough to determine the original. How much data can we POST to a form in Go? Back to the documentation, it looks like 10MB unless otherwise specified is the max. That should be plenty for us to tell the difference between two pieces of data. Now that we have a plan of attack, lets implement it!

Implimentation

So we need to send the back end two pieces of data and determine which one got encrypted and returned to us. Why reinvent the wheel, lets reuse the example from wikipedia and upload the Tux image for `m0` and an inverted Tux image for `m1`. Even after encryption it should be easy to tell which one was returned.

Since the website will only accept text, lets do this with `curl`:

Bash

```
curl -vvv -F "driver_license=QIVG6432WBS3GQ5ZAZ2RQKK6ZTTOMQHS" \
-F "m0=@tux.bmp" \
-F "m1=@tux2.bmp" \
http://challenges.hackover.h4q.it:8202/ciphertext
```

However, we get this as the response from the server:

```
ÿØÿàJFIFÿÛC
```

That is way too short to be our encrypted image. What is going on? Since we are sending image files to a web form, the Go backend is expecting text as Form values:

Go

```
m0 := []byte(r.FormValue("m0"))
m1 := []byte(r.FormValue("m1"))
```

Text is usually terminated with null bytes `\x00`, I wonder if we have any nulls in the tux bmp images?

```Bash
> xxd tux.bmp | head -5
0000000: 424d de9c 0b00 0000 0000 1e04 0000 2800  BM..........(.
0000010: 0000 2003 0000 b603 0000 0100 0800 0000  .. ............
0000020: 0000 c098 0b00 120b 0000 120b 0000 fa00  ................
0000030: 0000 fa00 0000 0402 0200 0406 0600 0440  ...............@
0000040: 5800 0791 b400 1493 c700 3090 bd00 20c6  X.........0... .
```

Looks like BMP header contains a null at the 6th byte. Ok so we can't upload a BMP, what if we created a BMP that had no null bytes in the data section, stripped the header, uploaded that, and then reattached the header after getting the encrypted output so it would be viewable?

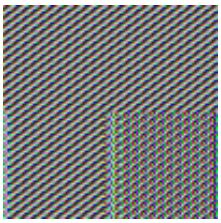
First lets make two BMPs that have no nulls, are the same size, and have recognizable patterns. Its really hard to see on this white page, but these are white squares with a not-quite-black (since black is `\x00` in a BMP) square on either the lower left or upper right.

Bottom Left: 

Top Right:

Now, we chop off the header, upload them to the challenge, save the result, slap the header back on and....

```Bash
head -c 54 r.bmp > BMP_HEADER
tail -c 30002 tl.bmp > RED
tail -c 30002 br.bmp > BLUE
curl -vvv -F "driver_license=3RCFJB3DFNXIF3UM6IOFFQVJ5DVXGD77" \
-F "m0=$(cat RED)" \
-F "m1=$(cat BLUE)" \
http://challenges.hackover.h4q.it:8202/ciphertext > OUT
cat BMP_HEADER OUT > IMG.bmp
```

Victory! 

Since this is obviously the encrypted form of "bottom right" which we defined as blue, we click blue on the challenge and our car moves 1/40th the way around the circle.

Now its just a matter of repeating this 39 times to complete the circle and get the flag. This could be scripted and automatically send our choices to `/choose`, but that would have probably taken longer than just up-arrowing in the terminal and hitting enter while watching a preview of `IMG.bmp` change and clicking the correct value.

39 POST's and clicks later:

`hackover16{CBCisSem.Sec.ButCanUProofIt?}`

This was a cool challenge that demonstrated the weakness of ECB mode ciphers which are used by default in many programing languages.