nginx で LuaLua する

C90

lua-nginx-module でつくる
多要素認証システム

@jm6xxu

はじめに

2016 年の 5 月に、IT 分野で大事件が起こりました。これまでユーザを悩ませ続けてきたパスワードの定期変更が無意味である事がマイクロソフトによって公開され、また NIST もそれに続きました。私はこれに納得できませんでした。パスワードの定期変更は、ユーザだけでなくサービスを提供する側の落ち度でパスワードが漏洩した場合の対策だったはずでした。

問題の文書 “Microsoft Password Guidance” を入手して読んでみると衝撃的なことが書いてありました。曰く、

> ユーザにパスワードの定期変更や、長くて大文字小文字数字を含むパスワードを要求しても、クソみたいに弱いパスワードしか使わない。

曰く、

> そんなもん意味ねーから、多要素認証使おうぜ。

はい。これで納得がいきました。パスワードはオワコン、これからは多要素認証の時代だぜ！という内容でした。

そもそも認証とは、

- その人しか知らないもの (パスワードなど)
- その人しか持っていないもの（ワンタイムトークンなど）
- その人の一部（生体認証）

を用いてユーザを特定することです。多要素認証は、これらの複数を用いてユーザを特定することをいいます。

と、いうことで、今回は Web サーバである nginx で、Google Authenticator を使った多要素認証モジュールを作ってみましょう。

lua-nginx-module

lua-nginx-module は、Web サーバである nginx を、軽量プログラミング言語 Lua で拡張できるというモジュールです。以下のように Lua で記述可能なフックがいくつも用意されていますので、様々な拡張が可能です。

よく使われるのは、init、rewrite、access、content、log でしょうか。ちょっと見ないうちに、ssl 絡みや filter 絡みのフックが増えています。詳細は、openresty の lua-nginx-module の GitHub リポジトリを参照してください。

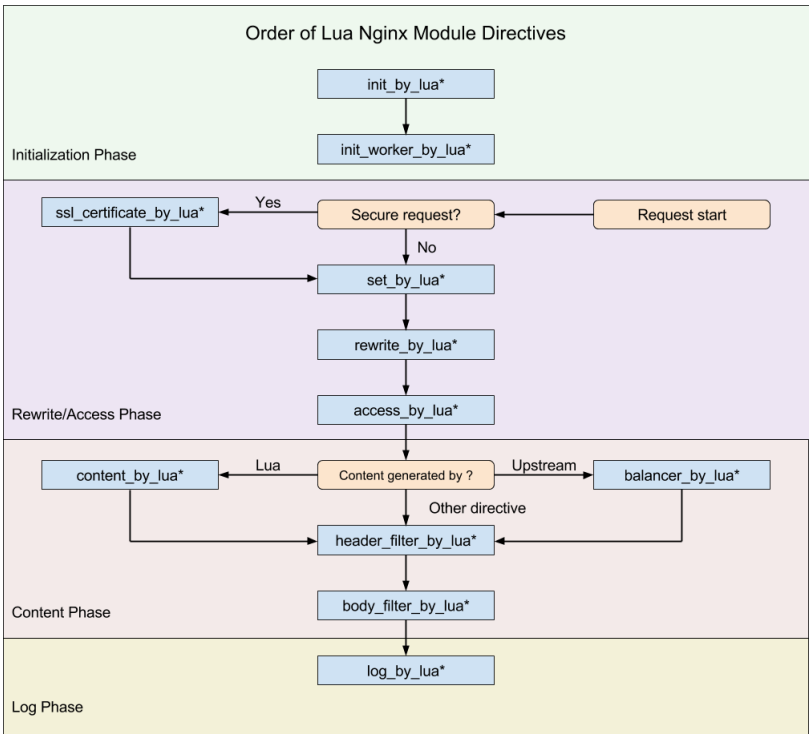

図 1　Lua で記述可能なフックと呼び出される順番

Hello World

では、さっそくやってみましょう。nginx.conf の server ブロックに以下の内容を追加します。

Listing 1　hello world

```
server{
        .... (snip)

  location /hello{
    default_type text/plain;
    content_by_lua 'ngx.say("Hello World")';
  }
```

変更したら、nginx を再起動させて、/hello にアクセスしてみましょう。自分の PC に nginx をインストールして試している人は、http://localhost/hello にブラウザでアクセスして、Hello World と表示されたら成功です。

次に GET リクエストを取得してみましょう。

Listing 2　GET パラメータの取得

```
server{
        .... (snip)

  location /hello{
    default_type text/plain;
    content_by_lua '
      local args = ngx.req.get_uri_args()
      ngx.say("Hello " .. args.name)';
    }
```

http://localhost/hello?name=chomy にアクセスして、Hello chomy と表示されるでしょう。最後は POST です。

Listing 3　POST パラメータの取得

```
server{
        .... (snip)

  location /hello{
    default_type text/plain;
    content_by_lua '
      ngx.req.read_body()
      local args = ngx.req.get_post_args()
      ngx.say("Hello " .. args.name)';
    }
```

その他、ヘッダの読み書きは、

Listing 4　ヘッダの読み書き

```
1 ngx.header.content_type = 'text/plain';
2 ngx.header["X-My-Header"] = 'blah blah';
3 ngx.header['Set-Cookie'] = {'a=32; path=/', 'b=4; path=/'}
```

ngx.header 変数でアクセスできます。クライアントへのステータスコードの返却は、ngx.exit 関数で行います。

使用可能なライブラリ

nginx の中で Lua5.1 インタープリタが動いているので、OpenResty 以外のものも Lua5.1 で動くライブラリは使用可能です。インストールされているライブラリの PATH は、lua_package_path で共有ライブラリの PATH は、lua_package_cpath で指定します。

Time based One Time Password

さて、認証の話に戻りましょう。これから多要素認証システムを nginx に組み込むわけですが、パスワード以外の何を使って認証するか決めなければなりません。よくあるのは、指紋や静脈と行った生体認証ですが、これはスキャナが別途必要なので現実的ではありません。次はハードウェアワンタイムトークンなのですが、Yubikey の登場でずいぶん入手性が良くなったとはいえ、まだ敷居が高いようです。今はみんなスマートフォンを持っているので、スマートフォンをトークンにする手があります。今回は Google Authenticator や、IIJ SmartKey が使っている TOTP (Time based One Time Password) を採用しましょう。

TOTP

TOTP は RFC6238 で規定されています。ワンタイムパスワードの生成は簡単です。

1. 鍵となる数値を *secret* とする
2. パスワードの有効期限を 30 秒とし、epoch からの秒数を 30 で割って、小数点以下を切り捨てた整数値を *step* とする
3. この *secret* を鍵として、step をコンテンツとする HMAC-SHA1 を計算
4. 計算した HMAC-SHA1 の下位 4bit を $offset$ とする
5. HMAC-SHA1 の offset バイトから 4byte 取り出し、0x7FFFFFFF と AND をとる
6. 最後に必要な桁数を下位の桁から取り出す

注意してほしいのは、*secret* も *step* もビッグエンディアンで使用するということです。ここでずいぶんハマりました。

TOTP ではユーザとサービス・プロバイダで HMAC-SHA1 を計算する時に必要な鍵を共有します。この共有鍵と時刻をパラメータにした同じアルゴリズムで計算した値を比較することでユーザの認証を行います。

この方式の素晴らしい点は、ログイン時に鍵が通信経路に乗らないことです。また、ログインに使用するトークンは 30 秒毎に変化しますので、仮に中間者攻撃を受けてトークンが漏洩しても、そのトークンが使えるのは 30 秒間だけです。また、トークンから鍵を復号するには

ものすごく莫大な計算量が必要で現実的ではありません。

さて、実装時には Lua でトークンを計算するのですが、どうやら nginx に組み込まれている Lua5.1 は、ビット演算をサポートしていないようです。よって、泣く泣く C++ で関数を書いて FFI(Foreign Function Interface) を使って呼び出すことにした。トークンを計算する OpenSSL を使った C++ のコードを Listing 5 に示します。

Listing 5　トークンの計算 (C++)

```
uint32_t get_token(const unsigned char *key,const int keylen,
                  const unsigned char *step, const uint16_t digits)
{
  unsigned char hash[SHA_DIGEST_LENGTH];
  unsigned int len;

  HMAC(EVP_sha1(), key, keylen,
       step, 8, hash, &len);

  uint32_t binary = ((hash[offset]&0x7F)<< 24)
     | ((hash[offset+1]&0xFF)<< 16)
     | ((hash[offset+2]&0xFF)<<8)
     | ((hash[offset+3]&0xFF));

  return binary%(uint64_t)powl(10,digits);
}
```

Lua から呼ばれる関数は、extern ”C”宣言が必要です。C++11 の機能を使っています。ここでは使っていませんが右辺値参照がすばらしいので、もう C++99 には戻れません。g++6.1 はデフォルトで C++14 とのことですので、もう C++99 は忘れても良いでしょう。

Listing 6　呼び出し用の関数

```
typedef std::vector<uint8_t> byte_array;
typedef std::array<unsigned char, 8> step_t;

extern "C" void get_token(const char *key, char *result, uint16_t len)
{
        step_t step;
        byte_array bKey;

                {
                uint64_t t = htobe64(ticks/30);
                memcpy(step.data(), &t, step.size());
                }

        hex2byte(key, &bKey);

        uint32_t tkn = get_token(bKey.data(), 10, step.data(), len);
        stringstream s;
        s<<setfill('0')<< setw(6)<< tkn;
```

```
19          strncpy(result, s.str().c_str(), len);
20          result[len-1] = '\0';
21  }
```

最後にコンパイルして、共有ライブラリを作成します。こんな感じになるでしょう。

Listing 7　共有ライブラリの作成

```
1  $ g++ -fPIC -shared --std=c++11 `pkg-config --cflags --libs openssl` -o
       libtotp.so totp.cc
```

これで、トークンを計算する libtotp.so ができました。

実装

TOTP の計算方法も、Lua を nginx で扱う方法もわかったので、実装しましょう。ユーザの登録機能は仕様範囲外とします。

ユーザーデータベース

ユーザー名と鍵が PostgreSQL の users テーブルに格納されているとします。鍵長は 10bit で、鍵は 16 進 20 桁の ASCII で格納しましょう。

Listing 8　SQL Schema

```
CREATE TABLE users (
        name text not null,
        secret varchar(20)
);
CREATE INDEX TO idxname ON users (name);
```

データベーススキーマを Listing 8 に示します。ここでは、Key-Value Store っぽく使ってみたかったので、あえて正規化をしていません。まぁする必要もないですが。

Postgres から secret を取得するために、ドライバをロードしておきましょう。nginx.conf に init_by_lua を追記記述しましょう。init_by_lua は、nginx の設定ファイルをロードする時に実行され、Lua インタープリタの初期化に使います。PostgreSQL のアクセスには、LuaSQL を使用しました。

Listing 9　Lua インタープリタの初期化

```
init_by_lua '
        driver = require "luasql.postgres"
        env = assert(driver.postgres())
        ffi = require("ffi")
        ffi.cdef[[void get_token(const char *key, char *result, uint32_t len)
            ]]
        otp = ffi.load("/usr/local/lib/libtotp.so")
';
server {
           ....
```

また、4 行目以降は Listing 5 のコードを含むトークンを計算するためのライブラリをロードしています。

Listing 10 は、PostgreSQL から secret を取得する関数です。SQL Injection を防ぐため、Prepared statement を使用しています。

Listing 10　データベースから secret を取得する関数

```
1  function get_secret(name)
2    local con=assert(env:connect("dbname=otp user=user"))
3    con:execute("PREPARE secret (text) as SELECT secret from users where name=
         $1;")
4    local cur=con:execute(string.format([[EXECUTE secret('%s');]],name))
5    local row = cur:fetch({},"a")
6    local secret
7    if not row then
8      secret = nil
9    else
10     secret = row.secret
11   end
12   cur:close()
13   con:close()
14
15   return secret
16 end
```

C ライブラリを用いたトークンの計算

Listing 11 は、FFI を使って C++ で書いたライブラリを呼び出すところです。2 行目で、結果を格納する char 型のバッファを確保して Listing 9 の 6 行目でロードしたライブラリの get_token 関数に渡しています。そして、char[64] 型を Lua string に変換するために ffi.string 関数を使用しました。

Listing 11　Token を計算

```
1 function get_token(secret)
2   local token = ffi.new("char[64]")
3   otp.get_token(secret, token, 64)
4   return ffi.string(token)
5 end
```

認証

Listing 12 には、認証部分のコードを示します。POST メソッドで、ユーザ名とトークンが入力されると DB にアクセスして secret を取得し、token を計算しています。計算した token

が入力された token と一致したら、ngx.exit 関数を用いて、ステータスコード 200 を、ユーザーデータベースにユーザー名が登録されていなかったり、トークンが一致しなかったら、401 Unauthorized を返却します。

Listing 12　SQL Schema

```
ngx.req.read_body()
local args, err = ngx.req.get_post_args()
if not args then
  ngx.exit(ngx.HTTP_UNAUTHORIZED)
end

local user = args.user
local secret = get_secret(user)
if not secret then
  ngx.exit(ngx.HTTP_UNAUTHORIZED)
end

if(args.token ~= get_token(secret)) then
  ngx.exit(ngx.HTTP_UNAUTHORIZED)
end

ngx.exit(ngx.HTTP_OK)
```

QR コードの作成

Google Authenticator は、QR コードで secret を設定するのが一般的です。secret を生成し、QR コードを生成してみましょう。まず乱数源から 10bit 取り出して、BASE32 エンコードします。Listing 13 は、/dev/urandom から 10bit の乱数を取り出して BASE32 エンコードしています。5 行目の文字列が生成された secret です。

Listing 13　secret 生成の例

```
$ dd if=/dev/urandom bs=10 count=1|base32
1+0 レコード入力
1+0 レコード出力
10 bytes copied, 0.000120597 s, 82.9 kB/s
NDS4MMILSEND5DZZ
```

次に QR コードを生成します。Linux を使っている方は、qrencode コマンドを使用するのが楽です。

Listing 14　QR コード生成の例

```
$ qrencode -o qrcode.png "otpauth://totp/Example:alice@example.com?secret=
    NDS4MMILSEND5DZZ&issuer=Example"
```

図 2　生成された QR コード

とコマンドラインで実行すると、qrcode.png というファイル名で、図 のような QR コードが生成されます。

Listing 14 で qrencode に渡した文字列の、secret がトークンの生成に使用する、BASE32 エンコードされた secret です。

次に生成した secret をデータベースに登録します。

Listing 15　データベースへの登録

```
1 $ echo NDS4MMILSEND5DZZ | base32 -d |hexdump
2 0000000 e568 31c6 910b 3e1a 398f
3 000000a
4 $ psql -c "INSERT INTO users (name,secret) VALUES('alice@example.com','68
      e5c6310b911a3e8f39');" otp
```

Listing 15 の 1 行目で生成した secret を 16 進数に変換しています。4 行目は、その変換した値を psql コマンドでデータベースに INSERT しているのですが注意して欲しいのは、登録する secret がビッグエンディアンになっています。もちろん、CPU がビッグエンディアンであるコンピュータを使用されていれば、2 行目に表示された通り入力します。

Deploy

Lua のコードを配置しましょう。作成した Lua のプログラムを nginx.conf に埋め込んでも良いのですが、行数も多くなりましたので、JIT コンパイルしたファイルを指定することにしましょう。これにより、設定ファイルの可読性が上がるだけでなく、実効速度も向上します。

JIT コンパイルは以下ように行います。

Listing 16　JIT Compile

```
1 $ luajit -bg totp.lua totp.luac
```

totp.lua をコンパイルして、totp.luac を作成します。これを設定ファイルで指定します。location ブロックの access_by_lua_file に指定します。

Listing 17　コンパイルしたファイルを設定

```
server{
          ......
  location /auth{
     default_type text/plain;
     content_by_lua_file '/usr/local/lib/totp/totp.luac';
  }
          .....
```

テスト

テストは、curl コマンドで POST リクエストを送るのが良いでしょう

Listing 18　テスト

```
$ curl -v -X POST -d "user=alice" -d"token:123456" http://localhost/auth
```

これで、ステータスコード 200 が返ってくれば認証成功、401 が返ってくれば認証失敗です。

編集後記

“nginx で LuaLua する” をお送りします。実は、今年 (2016 年) の 2 月にタイトルだけ思いついていて、認証系の何かを書こうと思っていましたが、実は 6 月までちゃんと決まっていませんでした。6 月に CISSP のセミナーに出て、Azure の AD 認証の話を聞いたのがきっかけで多要素認証の話を書くことにしました。

5 月のパスワード定期変更のニュースは衝撃でした。しかし報道では他要素認証が必須という肝心な部分が抜け落ちていた、または書いてあっても最後の方にちょこっと書いてあるくらいの扱いでしたので、このニュースを根拠に、弱いパスワードであるにもかかわらず、変更せずドヤ顔で使い続ける人が増えそうで、非常に心配です。

多要素認証といいながら、TOTP だけしか実装していません。パスワードの認証機能も入れたかったのですが、時間切れでした。サーバ側のパスワードの保存方法は、語り尽くされていますのでみなさんへの課題としておきましょう。断じて、DB に平文で保存してはいけません。ソルト、ハッシュ、ストレッチです。この辺りのことも含めて、後日加筆するかもしれません。この原稿は、GitHub で管理されていますので、下にある GitHub リポジトリを watch すると更新時にお知らせが届きます。

紙でこの同人誌を入手された方は、下の GitHub リポジトリに PDF 版も入手可能です。また電子版で入手したけれども紙も欲しいという方も面付け済みの PDF を GitHub にアップロードしていますので、短辺とじ両面印刷でコピー本を作成することができます。

最後に、この同人誌は Debian/GNU Linux、TEXLive2016、psutils、git、GNU Make、vim といった、オープンソースソフトウェアを使って作成されました。フォントは IPAex フォントを PDF に埋め込んでいます。ありがとうございました。

2016 年 8 月 13 日　Keisuke Nakao (@jm6xxu)

参考文献

- “Microsoft Password Guidance”
  https://www.microsoft.com/en-us/research/publication/password-guidance/
- “DRAFT NIST Special Publication 800-63B Digital Authentication Guideline”
  https://pages.nist.gov/800-63-3/sp800-63b.html
- “Key Uri Format”
  https://github.com/google/google-authenticator/wiki/Key-Uri-Format