Labreport 06

Julian Deinert, Tronje Krabbe

14. Juli 2016

Inhaltsverzeichnis

1. TODO

1.2

TODO

2. CAs und Webserver-Zertifikate

2.2

Um ein selbstsigniertes Zertifikat zu erzeugen, sind die folgenden Befehle notwendig:

```
openssl genrsa -out server.key 2048
openssl req -new -key server.key -out server.csr
openssl x509 -req -days 365 -in server.csr -signkey server.key -out server.crt
```

Mit dem ersten Befehl erzeugen wir einen neuen, 2048-bit langen privaten Schlüssel. Der zweite Befehl erstellt die *certificate signing request*, in der die Informationen enthalten sind, die das Zertifikat später umfassen soll. Diese werden interaktiv abgefragt. Als Common Name verwenden wir `vmsrv12.svslab`.

Im dritten Befehl verwenden wir schließlich den erstellten Private Key, um aus der CSR ein Zertifikat zu generieren, das 365 Tage gültig ist.

Um dieses Zertifikat für den Apache-Server einzusetzen, ersetzen wir in der Konfigurationsdatei die Pfade hinter `SSLCertificateFile` und `SSLCertificateKeyFile` mit den Pfaden zum Zertifikat und dem private key.

Um einen Missbrauch von Zertifikaten und damit auch ihre Wirksamkeit sicherzustellen, werden selbsterstellten Zertifikaten nicht vertraut. Vertrauenswürdige Zertifikate werden von *Certificate Authorities* ausgestellt, deren Aufgabe es ist, rechtmäßig angefragte Zertifikate auszustellen und mit Glaubwürdigkeit zu versehen sowie im Umkehrschluss unglaubwürdige Zertifikatserstellungen zu erschweren.

2.3

Das Apache-Modul *mod_ rewrite* ermöglicht Weiterleitungen einzurichten. Wir wollen alle Requests an HTTPS weiterleiten und erweitern die config daher um folgende Zeilen. [1]

```
RewriteEngine On
RewriteCond %{HTTPS} !=on
RewriteRule ^/?(.*) https://%{SERVER_NAME}/$1 [R,L]
```

Mit dem ersten Befehl wird die Bearbeitung der Rewrite-Regeln eingeschaltet. Danach wird geprüft, ob nicht bereits HTTPS verwendet wird, da damit die Weiterleitung hinfällig wird. Schließlich wird die Rewrite-Regel definiert, die dann weiterleitet. Die Optionen am Ende sind die Flags. `L` gibt an, dass die Regel die letzte ist, die ausgeführt wird. `R` spezifiziert, dass ein HTTP Redirect passieren soll.

Die Regel selbst ist ein regulärer Ausdruck, bei dem alles nach einem optionalen "`/`" matched und in die HTTPS-URL eingesetzt wird.

2.4

Über die verschlüsselte Verbindung können wir die Zugangsdaten selbstverständlich nicht mitlesen. Verwenden wir *sslstrip* als Proxy, so wird die Umleitung auf die verschlüsselte Verbindung verhindert und wir können die Zugangsdaten recht komfortabel im Log-File nachlesen.

Die Sicherheit der Weiterleitung ist damit maximal als "gut gemeinte" Maßnahme für Endanwender gedacht, taugt aber nicht, ernsthafte Angriffe abzuwehren. Hierzu sind weitere Maßnahmen nötig, wie beispielsweise HSTS.

HTTP Strict Transport Security (HSTS) ist ein in RFC 6797[2] beschriebener Standard, der einen solchen Angriff verhindern kann. Wird HSTS aktiviert, so können Seiten unter einer Domain, die HSTS setzt, nicht über eine unverschlüsselte Verbindung abgerufen weden. HSTS wird über einen Header aktiviert, der bei der Antwort mitgesendet wird und etwa so aussieht:

```
Strict-Transport-Security: max-age=15768000
```

[1] Quelle: `https://wiki.apache.org/httpd/RewriteHTTPToHTTPS`
[2] `https://tools.ietf.org/html/rfc6797`

Damit wird HSTS im Browser strikt forciert und lässt sich auch für den Endnutzer nicht wieder abschalten. Mit dem `max-age`-Parameter wird das maximale Alter in Sekunden angegeben. 15768000 Sekunden entsprechen etwa einem halben Jahr, danach wird der Browser eine unverschlüsselte Verbindung prinzipiell wieder erlauben, wenn nicht erneut ein HSTS-Header mitgesendet wird.

3. Unsichere selbstenwtickelte Verschlüsselungsalgorithmen

3.1 BaziCrypt

Die letzten 10 Bytes des Ciphertexts sind exakt der Key. Oder, genauergesagt, der Key XOR Null, was den Key unverändert lässt. Jetzt ist es sehr einfach, die Dateien zu entschlüsseln. Siehe unser Skript im Appendix. Die drei Plaintexte sind:

[n01.txt.enc] Hallo Peter. Endlich koennen wir geheim kommunizieren! Bis bald, Max

[n02.txt.enc] Hi Max! Super, Sicherheitsbewusstsein ist ja extrem wichtig! Schoene Gruesse, Peter.

[n03.txt.enc] Hi Peter, hast du einen Geheimtipp fuer ein gutes Buch fuer mich? Gruss, Max

3.2 AdvaziCrypt - Denksport

Wenn wir davon ausgehen, dass der Key vollständig (wenn auch 'verschlüsselt') in der Datei vorhanden ist, dann sind auf jeden Fall die letzten 10 Bytes des Ciphertexts die verschlüsselten Key-Bytes. Jetzt kann man auf die Anzahl der Padding-Bytes schließen, indem man darauf achtet, wann Bytes aufhören, in das Muster zu passen, und somit aufhören, Teil des vermeintlichen, verschlüsselten Keys zu sein. Man zählt also mit und schaut, wann ein Byte nicht mehr passt. Dann kennt man den Wert, mit dem man XORen muss, und kann den Key entschlüsseln. Danach kann man mit dem entschlüssselten Key den Rest der Datei entschlüsseln. Für die genaue Implementation, siehe unser Skript im Appendix.

3.3 AdvaziCrypt

Die Plaintexte lauten:

[n04.txt.enc] Hi Max, natuerlich: Kryptologie von A. Beutelspacher ist super. Gruss Peter

[n05.txt.enc] Hi Peter, worum geht es in dem Buch? Ciao, Max.

[n06.txt.enc] Hi Max, das ist ein super Buch, das viele Krypto-Themen abdeckt. Gruss Peter

4. EasyAES

Da wir Plaintext und Ciphertext einer doppelten AES-Verschlüsselung kennen, können wir zunächst alle Zwischenresultate ermitteln, die sich ergeben, wenn man den Plaintext einmal verschlüsselt, und dann mit einfach Entschlüsselungen des Ciphertexts vergleichen. Dies funktioniert einigermaßen schnell, wenn wir Lookup-Zeiten bei dem Vergleich minimieren. Wir haben den Hauptteil unseres C-Programms im Appendix angehängt. Das gesamte Projekt kann hier gefunden werden: `https://github.com/mu35li/netsec/tree/master/labreport06/EasyAES`.

5. Timing-Attack

Unsere Timing-Attack ist im Anhang zu finden. Wir würden gerne an dieser Stelle einige Erkenntnisse auflisten. Zunächst haben wir die Object-File disassemblet. Dies ist sehr einfach mit `$ objdump -d password_compare.o`, oder etwas komplizierter, aber im Ergebnis übersichtlicher mit *radare2*.

Der Assembly können wir entnehmen, dass die Funktion keinen Boolean returnt, sondern einen Int. Obwohl auf dem Aufgabenblatt nicht eindeutig dargestellt war, welchen Returnwert die Funktion hat, wurde durch die gezeigte Java-Funktion Boolean impliziert. Bei einem korrekten Passwort wird 0 zurückgegeben, andernfalls -1. Dies ist ein wichtiges Detail, denn in C gelten alle Integer *ungleich* 0 als wahr, und nur 0 als falsch. Hat man also nur den Wiedergabe-Wert der Funktion z.B. in einer `if`-Klausel stehen, wird man sich wundern. Demnach ist die Aussage über *POSIX*-Erfüllung ebenfalls diskutabel.

Weiterhin ist der Assembly zu entnehmen, dass die Länge des Passworts gar nicht überprüft wird. Dafür gibt es gar nicht genügend Loops. Dies macht uns den Angriff recht einfach; wir nehmen einfach einen langen String, und justieren an einer Stelle die Zeichen so lange, bis wir eine eindeutige Spike in der Zeit sehen, und akzeptieren diesen Buchstaben dann als richtig. Sobald die Funktion 0 widergibt, sind wir durch.

Das Passwort lautet: `Licht-B()6eN`. Das Password lautet ebenfalls: `Licht-B()6eN1234lalalala`; tatsächlich wird jeder String, der mit `Licht-B()6eN` *beginnt*, und beliebig weitergeht, von der Funktion akzeptiert.

Unser Code ist im Appendix zu finden. Zu beachten ist auch dass dem Compiler `-O0` übergeben wird, damit er nicht optimiert und somit die Zeiten verfälscht.

Appendix

5.1 3. Selbstentwickelte Crypto

Bazi Crack

```
#!/usr/bin/env python
def crack(filename, bsize):
    """Decrypt a bazi-encrypted file given the file's
    name and the blocksize (usually 10)
    """

    result = ''
    counter = -1
    with open(filename, "rb") as f:
        f.seek(-bsize, 2) # seek last n bytes
        key = [int(b) for b in f.read(bsize)]
        f.seek(0, 0) # seek beginning of file again
        while True:
            counter += 1
            counter = counter % bsize
            b = f.read(1)
            if b == b'':
                break
            num = int.from_bytes(b, 'little')
            result += chr(num ^ key[counter])
    return result


if __name__ == "__main__":
    import sys
    print(crack(sys.argv[1], 10))
```

Advazi Crack

```
#!/usr/bin/env python
def crack(filename, bsize):
    """Decrypt an advazi-encrypted file given the file's
    name and the blocksize (usually 10)
    """

    result = ''
    with open(filename, "rb") as f:
        f.seek(-bsize, 2) # seek last n bytes

        # initialize key as last n bytes
        key = [int(b) for b in f.read(bsize)]

        f.seek(0, 0) # seek beginning of file again
        whole_file = f.read() # read in the entire file

        # compute padding
        padding = 0
        counter = bsize - 1
        for i in range(len(whole_file) - 1, 0, -1):
            if whole_file[i] != key[counter]:
                break
            else:
                padding += 1
            counter -= 1
            counter %= 10

        # decrypt key with padding value
        for i in range(len(key)):
            key[i] ^= padding

        counter = -1
        f.seek(0, 0) # seek beginning of file again

        # decrypt plaintext
        while True:
            counter += 1
            counter %= 10
            b = f.read(1)
            if b == b'':
                break
            num = int.from_bytes(b, 'little')
            result += chr(num ^ key[counter])
    return result[:len(result) - padding]

if __name__ == "__main__":
    import sys
    print(crack(sys.argv[1], 10))
```

5.2 4. EasyAES

```c
#include <stdlib.h>
#include <stdint.h>
#include <stdio.h>
#include <stdbool.h>
#include <assert.h>
#include <string.h>
#include <omp.h>
#include "mitm.h"
#include "simple_aes/simple_aes.h"
#include "util.h"

/* 7807081 possible keys */

const uint8_t plaintext[16] = {'V','e','r','s','c','h','l','u','e','s','s','e','l','u','n','g'};
const uint8_t ciphertext[16] = {
    0xbe, 0x39, 0x3d, 0x39,
    0xca, 0x4e, 0x18, 0xf4,
    0x1f, 0xa9, 0xd8, 0x8a,
    0x9d, 0x47, 0xa5, 0x74
};

/* map a key to its encryption or decryption result */
struct KeyRes {
    uint8_t * key;
    uint8_t * res;
};

/* storage for multiple KeyRes structs */
struct KeyResStore {
    KeyRes ** items;
    uint64_t size;
    uint64_t capacity;
};

int main(void) {
    /* init our key-result-store */
    KeyResStore *** store = (KeyResStore ***) calloc(256, sizeof(KeyResStore));
    CHECK_ALLOC(store);

    for (uint32_t i = 0; i < 256; i++) {
        store[i] = (KeyResStore **) calloc(256, sizeof(KeyResStore));
        CHECK_ALLOC(store[i]);
    }


    for (uint32_t i = 0; i < 256; i++) {
        for (uint32_t j = 0; j < 256; j++) {
            store[i][j] = KeyResStore_new();
        }
    }

    /* populate it with mappings of all legal keys to their encryption results */
    KeyResStore_populate(store);
    fprintf(stderr, "populated key-result-store\n");

    /* do our attack by comparing all stored encryption results
     * with all decryption results until we have a match.
     * Result will be printed to stdout */
    fprintf(stderr, "starting meet-in-the-middle attack...\n");
    meet_in_the_middle(store);
```

```
    fprintf(stderr, "done!\n");

    /* free our key-result-store; this also frees all key-result-maps stored */
    for (uint32_t i = 0; i < 256; i++) {
        for (uint32_t j = 0; j < 256; j++) {
            KeyResStore_free(store[i][j]);
        }
    }

    /* we were successful! */
    return EXIT_SUCCESS;
}

/* free a Key-Result mapping and its components */
void KeyRes_free(KeyRes * kr) {
    assert(kr != NULL);

    free(kr->key);
    free(kr->res);
    free(kr);
    DBPRINT("free KeyRes\n");
}

/* create and allocate memory for a new Key-Result mapping */
KeyRes * KeyRes_new(void) {
    KeyRes * kr = (KeyRes *) calloc(1, sizeof(KeyRes));
    CHECK_ALLOC(kr);

    kr->key = (uint8_t *) calloc(16, sizeof(uint8_t));
    CHECK_ALLOC(kr->key);

    kr->res = (uint8_t *) calloc(16, sizeof(uint8_t));
    CHECK_ALLOC(kr->res);

    DBPRINT("create new KeyRes\n");
    return kr;
}

void KeyResStore_free(KeyResStore * krs) {
    assert(krs != NULL);

    for (uint64_t i = 0; i < krs->size; i++) {
        KeyRes_free(krs->items[i]);
    }

    free(krs->items);
    free(krs);
    DBPRINT("free KeyResStore\n");
}

KeyResStore * KeyResStore_new(void) {
    KeyResStore * krs = (KeyResStore *) calloc(1, sizeof(KeyResStore));

    CHECK_ALLOC(krs);

    krs->items = (KeyRes **) calloc(100, sizeof(KeyRes **));

    CHECK_ALLOC(krs->items);

    krs->size = 0;
    krs->capacity = 100;
```

```
    DBPRINT("create new KeyResStore\n");
    return krs;
}

void KeyResStore_resize(KeyResStore * krs, uint64_t n) {
    assert(krs != NULL);

    KeyRes ** temp = (KeyRes **) realloc(krs->items, (krs->capacity + n) * sizeof(KeyRes **));

    CHECK_ALLOC(temp);

    krs->items = temp;
    krs->capacity += n;
    DBPRINT("resize KeyResStore\n");
}

void KeyResStore_add(KeyResStore * krs, KeyRes * kr) {
    assert(krs != NULL);

    /* if key-result-store is full, give it capacity for 1000 more */
    if (krs->size == krs->capacity) {
        KeyResStore_resize(krs, 1000);
    }

    krs->items[krs->size] = kr;
    krs->size++;
    DBPRINT("add KeyRes to KeyResStore\n");
}

void KeyResStore_populate(KeyResStore *** krs) {
    uint8_t byte1_pos = 0;
    uint8_t byte1_val = 0x01;
    uint64_t count = 1;

    /* add key with only null-bytes */
    uint8_t * nullpt = (uint8_t *) calloc(16, sizeof(uint8_t));
    CHECK_ALLOC(nullpt);

    KeyRes * nullkr = KeyRes_new();
    for (uint8_t i = 0; i < 16; i++) {
        nullpt[i] = plaintext[i];
    }

    aes128_encrypt(nullpt, nullkr->key);

    for (uint8_t i = 0; i < 16; i++) {
        nullkr->res[i] = nullpt[i];
    }

    KeyResStore_add(krs[nullpt[0]][nullpt[1]], nullkr);

    free(nullpt);

    /* populate with 1-non-null-byte keys */
    while (true) {
        KeyRes * kr = KeyRes_new();
        kr->key[byte1_pos] = byte1_val;

        uint8_t * pt = (uint8_t *) calloc(16, sizeof(uint8_t));
        CHECK_ALLOC(pt);

        for (uint8_t i = 0; i < 16; i++) {
```

```
        pt[i] = plaintext[i];
    }

    aes128_encrypt(pt, kr->key);

    for (uint8_t i = 0; i < 16; i++) {
        kr->res[i] = pt[i];
    }

    KeyResStore_add(krs[pt[0]][pt[1]], kr);
    count++;
    free(pt);

    if (byte1_pos == 15 && byte1_val == 0xff) {
        break;
    }

    if (byte1_val == 0xff) {
        byte1_pos++;
        byte1_val = 0x01;
    }
    else {
        byte1_val++;
    }
}

byte1_pos = 0;
byte1_val = 0x01;
uint8_t byte2_pos = 1;
uint8_t byte2_val = 0x01;

/* populate with 2-non-null-byte keys */
while (true) {
    KeyRes * kr = KeyRes_new();
    kr->key[byte1_pos] = byte1_val;
    kr->key[byte2_pos] = byte2_val;

    uint8_t * pt = (uint8_t *) calloc(16, sizeof(uint8_t));
    CHECK_ALLOC(pt);

    for (uint8_t i = 0; i < 16; i++) {
        pt[i] = plaintext[i];
    }

    aes128_encrypt(pt, kr->key);

    for (uint8_t i = 0; i < 16; i++) {
        kr->res[i] = pt[i];
    }

    KeyResStore_add(krs[pt[0]][pt[1]], kr);
    count++;
    free(pt);

    if (byte1_pos == 14 && byte2_pos == 15 \
        && byte1_val == 0xff && byte2_val == 0xff) {
        break;
    }

    if (byte1_val == 0xff && byte2_val == 0xff) {
        byte1_val = 0x01;
        byte2_val = 0x01;
```

```
            byte1_pos++;

            if (byte1_pos == byte2_pos) {
                byte2_pos++;
                byte1_pos = 0;
            }

            continue;
        }

        if (byte1_val == 0xff) {
            byte1_val = 0x01;
            byte2_val++;
        }
        else {
            byte1_val++;
        }
    }

    fprintf(stderr, "nr of keys: %lu\n", count);
}

void meet_in_the_middle(KeyResStore *** krs) {
    uint8_t byte1_pos = 0;
    uint8_t byte1_val = 0x01;
    //uint64_t counter = 0;
    bool done = false;

    /* try key with only null-bytes */
    uint8_t * nullkey = (uint8_t *) calloc(16, sizeof(uint8_t));
    CHECK_ALLOC(nullkey);
    uint8_t * nullct = (uint8_t *) calloc(16, sizeof(uint8_t));
    CHECK_ALLOC(nullct);

    for (uint8_t i = 0; i < 16; i++) {
        nullct[i] = ciphertext[i];
    }

    aes128_decrypt(nullct, nullkey);

    for (uint64_t i = 0; i < krs[nullct[0]][nullct[1]]->size; i++) {
        if (txt_eq(krs[nullct[0]][nullct[1]]->items[i]->res, nullct)) {
            printf("MATCH FOUND!\n");
            printf("Key 1: ");
            for (uint8_t j = 0; j < 15; j++) {
                printf("0x%02x, ", krs[nullct[0]][nullct[1]]->items[i]->key[j]);
            }
            printf("0x%02x\n", krs[nullct[0]][nullct[1]]->items[i]->key[15]);

            printf("Key 2: ");
            for (uint8_t j = 0; j < 15; j++) {
                printf("0x%02x, ", nullkey[j]);
            }
            printf("0x%02x\n", nullkey[15]);
            free(nullct);
            free(nullkey);
            return;
        }
    }

    free(nullct);
    free(nullkey);
```

```
/* decrypt with 1-non-null-byte keys */
while (!done) {
    uint8_t * key = (uint8_t *) calloc(16, sizeof(uint8_t));
    CHECK_ALLOC(key);

    key[byte1_pos] = byte1_val;

    uint8_t * ct = (uint8_t *) calloc(16, sizeof(uint8_t));
    CHECK_ALLOC(ct);

    for (uint8_t i = 0; i < 16; i++) {
        ct[i] = ciphertext[i];
    }

    aes128_decrypt(ct, key);

    #pragma omp parallel for shared(done)
    for (uint64_t i = 0; i < krs[ct[0]][ct[1]]->size; i++) {
        /*counter++;
        if (counter % 100000 == 0) {
            fprintf(stderr, "%lu iterations completed.\n", counter);
        }*/

        if (done) continue;

        if (txt_eq(krs[ct[0]][ct[1]]->items[i]->res, ct)) {
            printf("MATCH FOUND!\n");
            printf("Key 1: ");
            for (uint8_t j = 0; j < 15; j++) {
                printf("0x%02x, ", krs[ct[0]][ct[1]]->items[i]->key[j]);
            }
            printf("0x%02x\n", krs[ct[0]][ct[1]]->items[i]->key[15]);

            printf("Key 2: ");
            for (uint8_t j = 0; j < 15; j++) {
                printf("0x%02x, ", key[j]);
            }
            printf("0x%02x\n", key[15]);

            done = true;
        }
    }

    free(key);
    free(ct);

    if (byte1_pos == 15 && byte1_val == 0xff) {
        break;
    }

    if (byte1_val == 0xff) {
        byte1_pos++;
        byte1_val = 0x01;
    }
    else {
        byte1_val++;
    }
}

fprintf(stderr, "1-non-null-byte keys done, no match yet.\n");
```

```
byte1_pos = 0;
byte1_val = 0x01;
uint8_t byte2_pos = 1;
uint8_t byte2_val = 0x01;

/* populate with 2-non-null-byte keys */
while (!done) {
    uint8_t * key = (uint8_t *) calloc(16, sizeof(uint8_t));
    CHECK_ALLOC(key);

    key[byte1_pos] = byte1_val;
    key[byte2_pos] = byte2_val;

    uint8_t * ct = (uint8_t *) calloc(16, sizeof(uint8_t));
    CHECK_ALLOC(ct);

    for (uint8_t i = 0; i < 16; i++) {
        ct[i] = ciphertext[i];
    }

    aes128_decrypt(ct, key);

    #pragma omp parallel for shared(done)
    for (uint64_t i = 0; i < krs[ct[0]][ct[1]]->size; i++) {

        if (done) continue;

        if (txt_eq(krs[ct[0]][ct[1]]->items[i]->res, ct)) {
            printf("MATCH FOUND!\n");
            printf("Key 1: ");
            for (uint8_t j = 0; j < 15; j++) {
                printf("0x%02x, ", krs[ct[0]][ct[1]]->items[i]->key[j]);
            }
            printf("0x%02x\n", krs[ct[0]][ct[1]]->items[i]->key[15]);

            printf("Key 2: ");
            for (uint8_t j = 0; j < 15; j++) {
                printf("0x%02x, ", key[j]);
            }
            printf("0x%02x\n", key[15]);

            done = true;
        }
    }

    free(key);
    free(ct);

    if (byte1_pos == 14 && byte2_pos == 15\
        && byte1_val == 0xff && byte2_val == 0xff) {
        break;
    }

    if (byte1_val == 0xff && byte2_val == 0xff) {
        byte1_val = 0x01;
        byte2_val = 0x01;
        byte1_pos++;

        if (byte1_pos == byte2_pos) {
            byte2_pos++;
            byte1_pos = 0;
        }
```

```
            continue;
        }

        if (byte1_val == 0xff) {
            byte1_val = 0x01;
            byte2_val++;
        }
        else {
            byte1_val++;
        }
    }

    if (!done) {
        printf("No match found!\n");
    }
}

bool txt_eq(uint8_t * t1, uint8_t * t2) {
    for (uint8_t i = 0; i < 16; i++) {
        if (t1[i] != t2[i]) {
            return false;
        }
    }
    return true;
}
```

5.3 5. Timing-Attack

timing.h

```
#pragma once

#include <stdbool.h>
#include <time.h>
#include <stdint.h>

extern int password_compare(const char * password);
size_t time_max(clock_t * arr, size_t len);
```

timing.c

```
#include <stdio.h>
#include <assert.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>
#include "timing.h"

#define LENGTH 20

const char * symbols = "abcdefghijklmnopqrstuvwxyz"
                       "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
                       "0123456789"
                       "!@#$%^&*()-_=+[]{}|/?.,<>;:'\"\\`~ ";

int main(void) {
    const size_t len = strlen(symbols);
    clock_t * times = (clock_t *) calloc(len, sizeof(clock_t));

    clock_t start, stop;
    char secret[LENGTH] = {'a'};
```

```
    int res = 0;

    for (uint32_t i = 0; i < LENGTH; i++) {
        printf("i: %u\n", i);
        for (uint64_t j = 0; j < len; j++) {
            secret[i] = symbols[j];

            start = clock();
            for (uint64_t k = 0; k < 10000000; k++) {
                res = password_compare(secret);
            }
            stop = clock();
            times[j] = stop - start;
            if (res != -1) {
                printf("compare fct returned != -1; pw might be: %s\n", secret);
            }
        }

        secret[i] = symbols[time_max(times, len)];
        printf("pw so far: %s\n", secret);
    }

    printf("timing attack says: %s\n", secret);
    printf("password_compare says: %d\n", password_compare(secret));

    free(times);

    return EXIT_SUCCESS;
}

/* return the index holding the highest element */
size_t time_max(clock_t * arr, size_t len) {
    clock_t max = 0;
    size_t res = 0;

    for (size_t i = 0; i < len; i++) {
        if (arr[i] > max) {
            max = arr[i];
            res = i;
        }
    }
    return res;
}
```