Pharisaeus Added delphi writeup		Latest commit 628ff23 on 16 Feb
..		
README.md	Added delphi writeup	3 months ago
index.php	added vhash writeup	3 months ago
vhash	added vhash writeup	3 months ago

README.md

Vhash (crypto 450p)

###ENG PL

In the task we get access to a webpage where we can login as guest/guest and as a result we get a cookie containing:

```
SOME_LONG_HEX_HASH|username=guest&date=2017-02-13T23:45:45+0000&secret_length=8&
```

We also get to see the source code of the webpage and a binary which is used to generate the hash value.

From analysis of the index page we can see that the hash value is generated with

```
function create_hmac($data) {
  return do_hash(SECRET . $data);
}
```

So there is some secret glued in front of the cookie data and then `vhash` binary is execute on this input. When the website is authenticating us with the cookie it takes the cookie data, calculates the hash again and compares it agains what is in the cookie. The idea is that we can't modify the cookie value (eg, by setting username to `admin`) because the hash would have to be changed as well. And we can't calculate the proper hash without knowing the secret bytes added by the server.

We notice that the server creates a map out of cookie parameters by parsing them in-order. As a result if a certain property is defined more than once, only the last value will get stored:

```
$pairs = explode('&', $cookie);
$args = array();
foreach($pairs as $pair) {
  if(!strpos($pair, '='))
    continue;

  list($name, $value) = explode('=', $pair, 2);
  $args[$name] = $value;
}
$username = $args['username'];
```

So if we could append to the cookie `&username=administrator&` we would get past the security check!

Once we reverse engineer vhash binary and analyse it, we can see that it is actually prone to `Hash Length Extension` attack. The general schema of the algorithm is:

1. Set initial state
2. Split input into blocks

3. For each block:
   - Take input block
   - Use `hash_update` to perform computations over the current state and input block
   - Save the results as new current state
4. Finalize the algorithm by using `hash_update` with a certain fixed block dependent only on length of already hashed input
5. Print current state as final hash

We know that the cookie we get from the page is in fact hashed value of `secret+cookie_payload+final_hash_block`. If we now set this hash as `current state` of the hashing algorithm and perform `hash_update` with some additional payload we will actually get a proper hash value for `secret+cookie_payload+final_hash_block+additional_payload`! And since we know exactly how long this payload is, we can also add here the new `final_hash_block2`, as a result getting a proper hash value of `secret+cookie_payload+final_hash_block+additional_payload+final_hash_block2`. All of this without actually knowing the `secret` bytes at all, just the `state` of the algorithm.

Now let's think what server will do if we pass as cookie payload string of `cookie_payload+final_hash_block+additional_payload`. The server will add secret bytes in front getting `secret+cookie_payload+final_hash_block+additional_payload` and then will hash it adding the final block at the end, which will result in `secret+cookie_payload+final_hash_block+additional_payload+final_hash_block2`. And this is exactly the same thing as we mentioned before! So by passing such input we can actually generate proper hash value without knowing the secret.

We rewritten the hash algorithm into C to be able to test is easily:

```c
#include <cstdint>
#include <cstdio>
#include <cstring>
#include <x86intrin.h>

struct vhash_ctx{
    uint32_t state[32];
};

inline unsigned int rol4(unsigned int value, int count) { return _rotl((unsigned int)value, count); }

void vhash_init(vhash_ctx *vctx)
{
    memcpy(vctx, "vhas", 4uLL);
    memcpy(&vctx->state[1], "h: r", 4uLL);
    memcpy(&vctx->state[2], "ock ", 4uLL);
    memcpy(&vctx->state[3], "hard", 4uLL);
    memcpy(&vctx->state[4], " 102", 4uLL);
    memcpy(&vctx->state[5], "4 bi", 4uLL);
    memcpy(&vctx->state[6], "t se", 4uLL);
    memcpy(&vctx->state[7], "curi", 4uLL);
    memcpy(&vctx->state[8], "ty. ", 4uLL);
    memcpy(&vctx->state[9], "For ", 4uLL);
    memcpy(&vctx->state[10], "thos", 4uLL);
    memcpy(&vctx->state[11], "e sp", 4uLL);
    memcpy(&vctx->state[12], "ecia", 4uLL);
    memcpy(&vctx->state[13], "l mo", 4uLL);
    memcpy(&vctx->state[14], "mome", 4uLL);
    memcpy(&vctx->state[15], "nts ", 4uLL);
    memcpy(&vctx->state[16], "when", 4uLL);
    memcpy(&vctx->state[17], " you", 4uLL);
    memcpy(&vctx->state[18], " nee", 4uLL);
    memcpy(&vctx->state[19], "d th", 4uLL);
    memcpy(&vctx->state[20], "robb", 4uLL);
    memcpy(&vctx->state[21], "ing ", 4uLL);
    memcpy(&vctx->state[22], "perf", 4uLL);
    memcpy(&vctx->state[23], "orma", 4uLL);
    memcpy(&vctx->state[24], "nce.", 4uLL);
    memcpy(&vctx->state[25], " Ask", 4uLL);
    memcpy(&vctx->state[26], " you", 4uLL);
    memcpy(&vctx->state[27], "r do", 4uLL);
    memcpy(&vctx->state[28], "ctor", 4uLL);
    memcpy(&vctx->state[29], " abo", 4uLL);
    memcpy(&vctx->state[30], "ut v", 4uLL);
    memcpy(&vctx->state[31], "hash", 4uLL);
}
```

```
void vhash_round(vhash_ctx *vctx)
{
    for (int i = 0; i <= 31; ++i )
        ++vctx->state[i];
    for (int ia = 0; ia <= 31; ++ia )
    {
        uint32_t v1 = rol4(vctx->state[ia], ia);
        vctx->state[ia] = v1;
    }
    for (int ib = 0; ib <= 31; ++ib )
        vctx->state[ib] += vctx->state[((((unsigned int)((ib + 1) >> 31) >> 27) + (uint8_t)ib + 1) & 0x1F)
                                       - ((unsigned int)((ib + 1) >> 31) >> 27)];
    for (int ic = 0; ic <= 31; ++ic )
        vctx->state[ic] ^= vctx->state[((((unsigned int)((ic + 7) >> 31) >> 27) + (uint8_t)ic + 7) & 0x1F)
                                       - ((unsigned int)((ic + 7) >> 31) >> 27)];
    uint32_t t = vctx->state[0];
    for (int id = 0; id <= 30; ++id )
        vctx->state[id] = vctx->state[id + 1];
    vctx->state[31] = t;
}

void vhash_update(vhash_ctx *vctx, uint32_t (*in)[4])
{
    for (int i = 0; i <= 31; ++i )
        vctx->state[i] += (*in)[((((unsigned int)((uint64_t)i >> 32) >> 30) + (uint8_t)i) & 3)
    - ((unsigned int)((uint64_t)i >> 32) >> 30)];
    for (int ia = 0; ia <= 511; ++ia )
        vhash_round(vctx);
    for (int ib = 0; ib <= 31; ++ib )
        vctx->state[ib] ^= (*in)[((((unsigned int)((uint64_t)ib >> 32) >> 30) + (uint8_t)ib) & 3)
    - ((unsigned int)((uint64_t)ib >> 32) >> 30)];
    for (int ic = 0; ic <= 511; ++ic )
        vhash_round(vctx);
}

void vhash_final(vhash_ctx *vctx, uint32_t len)
{
    uint32_t f[4];
    f[0] = 2147483648;
    f[1] = len;
    f[2] = 0;
    f[3] = -1;
    vhash_update(vctx, (uint32_t (*)[4])f);
}

void print_hash(vhash_ctx *vctx)
{
    for (int i = 0; i <= 7; ++i )
        printf(
                "%08x%08x%08x%08x",
                vctx->state[4 * i],
                vctx->state[4 * i + 1],
                vctx->state[4 * i + 2],
                vctx->state[4 * i + 3]);
    printf("\n");
}

void process_data(const char *data, vhash_ctx* vctx, uint32_t &size, uint32_t len) {
    int done = 0;
    size= 0;
    uint32_t block[4];
    while ( !done )
    {
        memset(block, 0, 0x10uLL);
        for (int i = 0; (unsigned int)i <= 0xF; ++i )
        {
            if ( size<len )
            {
                uint8_t inbyte = data[size];
                ++size;
                block[i / 4] |= inbyte << 8 * (3 - (char)i % 4);
            }
            else
            {
                done = 1;
                break;
            }
        }
```

```
            vhash_update(vctx, (uint32_t (*)[4])block);
            if(size==len){
                done=1;
            }
        }
    }

    void hash(const char* data, uint32_t len) {
        vhash_ctx vctx;
        vhash_init(&vctx);
        uint32_t size;
        process_data(data, &vctx, size, len);
        vhash_final(&vctx, size);
        print_hash(&vctx);
    }
```

And now with those primitives in place we created a function for generating the fake signature hash:

```
void rigged(vhash_ctx *vectx, const char* data, uint32_t len){
    set_initial_state(vectx);
    uint32_t size;
    process_data(data, vectx, size, len);
    vhash_final(vectx, 120);
    print_hash(vectx);
}
```

What we do here is set the initial state from the cookie we have, then perform hashing step with our additional payload and then add the final block, using `120` as the size of the whole input because this is how long it should be. It is because the payload we are actually hashing has format:

```
12345678username=guest&date=2017-02-14T00:22:54+0100&secret_length=8&\0\0\0\0\0\0\0\0\0\0\0\x80\x00\x00\x0
```

assuming `12345678` as secret. `\0` bytes comes from the padding added to the last input block when it gets hashed and `\x80\x00\x00\x00\x00\x00\x00\x45\x00\x00\x00\x00\xff\xff\xff\xff` is the final block added by the server to the initial cookie (notice 0x45 as input size).

We used a small python script to generate the state initializer from cookie value:

```
from crypto_commons.generic import chunk


def dump_state(data):
    print("\n".join(["vectx->state[" + str(i) + "] = 0x" + c + ";" for i, c in enumerate(chunk(data, 8))]))


def main():
    data = 'e4d6ed47e23694e76f8a611111fd7b52a980e40a718b28256342adc740814a09fa4592e434dbb0b2b5d3405dd554e5b

    dump_state(data)

main()
```

and it generates the contents for `set_initial_state` :

```
void set_initial_state(const vhash_ctx *vectx) {
    vectx->state[0] = 0xe4d6ed47;
    vectx->state[1] = 0xe23694e7;
    vectx->state[2] = 0x6f8a6111;
    vectx->state[3] = 0x11fd7b52;
    vectx->state[4] = 0xa980e40a;
    vectx->state[5] = 0x718b2825;
    vectx->state[6] = 0x6342adc7;
    vectx->state[7] = 0x40814a09;
    vectx->state[8] = 0xfa4592e4;
    vectx->state[9] = 0x34dbb0b2;
    vectx->state[10] = 0xb5d3405d;
    vectx->state[11] = 0xd554e5b4;
    vectx->state[12] = 0x0f432cfb;
    vectx->state[13] = 0x7cbf6fe0;
    vectx->state[14] = 0x6d95ed16;
    vectx->state[15] = 0xcc1f1b21;
```

```
        vectx->state[16] = 0xf0d5f5b4;
        vectx->state[17] = 0x32b0e82c;
        vectx->state[18] = 0x9ce0693e;
        vectx->state[19] = 0xd4caca8c;
        vectx->state[20] = 0xf7a035e4;
        vectx->state[21] = 0x8a29fc35;
        vectx->state[22] = 0x8d75bb06;
        vectx->state[23] = 0x74d5e0b9;
        vectx->state[24] = 0x10252bfc;
        vectx->state[25] = 0x1712a766;
        vectx->state[26] = 0x98662a0e;
        vectx->state[27] = 0xa4441183;
        vectx->state[28] = 0xd4b3d7f1;
        vectx->state[29] = 0xade921ae;
        vectx->state[30] = 0x47f0b0c6;
        vectx->state[31] = 0x41e25813;
    }
```

Now we just need to run `rigged(&vctx,"&username=administrator&", 24);` and it will print the proper hash value.

Last step was just to send this to the game server with:

```python
import urllib
import requests


def main():
    cookies = {"auth": urllib.quote_plus(
        "39005958a36fcfabc56482ef87d46c3e1aec3c7babbc487dfc73d5c48441f24e67ffb44a2814be8e2e890e282188430b44
    result = requests.get("http://thenewandlessbrokenvhash.ctf.bsidessf.net:9292/index.php", cookies=cookie
    print(result.content)


main()
```

And it gave us `FLAG:06c211f73f4f5ba198c7fb56145b39a2` in response.

###PL version

W zadaniu dostajemy dostęp do strony gdzie możemy zalogować się jako guest/guest a jako wynik dostajemy cookie zawierące:

`SOME_LONG_HEX_HASH|username=guest&date=2017-02-13T23:45:45+0000&secret_length=8&`

Dostajemy też kod strony oraz binarke która jest użyta do generowania hasha.

Z analizy strony php możemy zauważyć, że hash jest generowany przez:

```php
  function create_hmac($data) {
    return do_hash(SECRET . $data);
  }
```

Więc jest pewna sekretna wartość doklejana na początku danych cookie a następnie `vhash` jest uruchamiany na takim wejściu. Kiedy strona autentykuje nas za pomocą cookie, bierze dane, wylicza hasha i porównuje z tym co zapisane w cookie. Idea jest taka, ze nie możemy zmienić wartości w cookie (np. ustawić username na `admin`) bo hash też trzeba by odpowiednio zmienić. A nie możemy obliczyć tego nowego hasha bez znajomości sekretnych bajtów doklejanych przez serwer.

Zauważamy dodatkowo że serwer tworzy mapę z parametrów w cookie parsując je w kolejności. To oznacza, że jeśli jakiś parametr występuje więcej niż raz to tylko jego ostatnia wartość będzie zapisana:

```php
  $pairs = explode('&', $cookie);
  $args = array();
  foreach($pairs as $pair) {
    if(!strpos($pair, '='))
      continue;

    list($name, $value) = explode('=', $pair, 2);
    $args[$name] = $value;
```

```
}
$username = $args['username'];
```

Więc gdybyśmy mogli dodać do cookie `&username=administrator&` przeszlibyśmy sprawdzenie tożsamości!

Po zreversowaniu binarki vhash i wstępnej analizie doszliśmy do wniosku, że algorytm jest podatny na atak `Hash Length Extension`. Generaly przebieg algorytmu to:

1. Ustaw stan początkowy
2. Podziel dane na bloki
3. Dla każdego bloku
   - Weź blok
   - Użyj `hash_update` żeby wykonać obliczenia na aktualnym stanie oraz bloku wejściowym
   - Zapisz wyniki jako nowy stan
4. Zakończ algorytm poprzez `hash_update` z pewnym stałym blokiem zaleznym jedynie od dlugosci hashowanego inputu.
5. Wypisz aktualny stan jako końcowy hash.

Wiemy że cookie które dostajemy ze strony to w rzeczywistości wynik hashowania `secret+cookie_payload+final_hash_block`. Jeśli ustawimy ten hash jako aktualny stan algorytmu i wykonamy `hash_update` z jakimś dodatkowym payloadem to dostaniemy poprawny hash dla danych `secret+cookie_payload+final_hash_block+additional_payload`! A skoro wiem też jaką długość ma cały ten payload to możemy dodać też nowy `final_hash_block2`, w rezultacie uzyskując poprawny hash dla `secret+cookie_payload+final_hash_block+additional_payload+final_hash_block2`. I to wszystko bez znajomosci bajtów oznaczonych jako `secret`, tylko dzięki znajomości stanu algorytmu.

Zastanówmy się teraz co się stanie jeśli wyślemy do serwera jako dane `cookie_payload+final_hash_block+additional_payload`. Serwer doda do tego sekretne bajty dostając `secret+cookie_payload+final_hash_block+additional_payload` a potem zahashuje dodając na końcu finalny blok, dostając tym samym hash dla result in `secret+cookie_payload+final_hash_block+additional_payload+final_hash_block2`. A to jest dokładnie to samo o czym napisaliśmy wyżej! Więc wysyłając takie dane możemy wygenerować lokalnie poprawny hash nie znając sekretu.

Przepisaliśmy algorytm hashowania do C żeby móc wygodnie to testować:

```
#include <cstdint>
#include <cstdio>
#include <cstring>
#include <x86intrin.h>

struct vhash_ctx{
    uint32_t state[32];
};

inline unsigned int rol4(unsigned int value, int count) { return _rotl((unsigned int)value, count); }

void vhash_init(vhash_ctx *vctx)
{
    memcpy(vctx, "vhas", 4uLL);
    memcpy(&vctx->state[1], "h: r", 4uLL);
    memcpy(&vctx->state[2], "ock ", 4uLL);
    memcpy(&vctx->state[3], "hard", 4uLL);
    memcpy(&vctx->state[4], " 102", 4uLL);
    memcpy(&vctx->state[5], "4 bi", 4uLL);
    memcpy(&vctx->state[6], "t se", 4uLL);
    memcpy(&vctx->state[7], "curi", 4uLL);
    memcpy(&vctx->state[8], "ty. ", 4uLL);
    memcpy(&vctx->state[9], "For ", 4uLL);
    memcpy(&vctx->state[10], "thos", 4uLL);
    memcpy(&vctx->state[11], "e sp", 4uLL);
    memcpy(&vctx->state[12], "ecia", 4uLL);
    memcpy(&vctx->state[13], "l mo", 4uLL);
    memcpy(&vctx->state[14], "mome", 4uLL);
    memcpy(&vctx->state[15], "nts ", 4uLL);
    memcpy(&vctx->state[16], "when", 4uLL);
    memcpy(&vctx->state[17], " you", 4uLL);
    memcpy(&vctx->state[18], " nee", 4uLL);
    memcpy(&vctx->state[19], "d th", 4uLL);
    memcpy(&vctx->state[20], "robb", 4uLL);
    memcpy(&vctx->state[21], "ing ", 4uLL);
    memcpy(&vctx->state[22], "perf", 4uLL);
    memcpy(&vctx->state[23], "orma", 4uLL);
    memcpy(&vctx->state[24], "nce.", 4uLL);
    memcpy(&vctx->state[25], " Ask", 4uLL);
    memcpy(&vctx->state[26], " you", 4uLL);
    memcpy(&vctx->state[27], "r do", 4uLL);
    memcpy(&vctx->state[28], "ctor", 4uLL);
    memcpy(&vctx->state[29], " abo", 4uLL);
    memcpy(&vctx->state[30], "ut v", 4uLL);
    memcpy(&vctx->state[31], "hash", 4uLL);
}

void vhash_round(vhash_ctx *vctx)
{
    for (int i = 0; i <= 31; ++i )
        ++vctx->state[i];
    for (int ia = 0; ia <= 31; ++ia )
    {
        uint32_t v1 = rol4(vctx->state[ia], ia);
        vctx->state[ia] = v1;
    }
    for (int ib = 0; ib <= 31; ++ib )
        vctx->state[ib] += vctx->state[((((unsigned int)((ib + 1) >> 31) >> 27) + (uint8_t)ib + 1) & 0x1F)
                                       - ((unsigned int)((ib + 1) >> 31) >> 27)];
    for (int ic = 0; ic <= 31; ++ic )
        vctx->state[ic] ^= vctx->state[((((unsigned int)((ic + 7) >> 31) >> 27) + (uint8_t)ic + 7) & 0x1F)
                                       - ((unsigned int)((ic + 7) >> 31) >> 27)];
    uint32_t t = vctx->state[0];
    for (int id = 0; id <= 30; ++id )
        vctx->state[id] = vctx->state[id + 1];
    vctx->state[31] = t;
}

void vhash_update(vhash_ctx *vctx, uint32_t (*in)[4])
{
    for (int i = 0; i <= 31; ++i )
        vctx->state[i] += (*in)[((((unsigned int)((uint64_t)i >> 32) >> 30) + (uint8_t)i) & 3)
    - ((unsigned int)((uint64_t)i >> 32) >> 30)];
    for (int ia = 0; ia <= 511; ++ia )
        vhash_round(vctx);
    for (int ib = 0; ib <= 31; ++ib )
        vctx->state[ib] ^= (*in)[((((unsigned int)((uint64_t)ib >> 32) >> 30) + (uint8_t)ib) & 3)
    - ((unsigned int)((uint64_t)ib >> 32) >> 30)];
    for (int ic = 0; ic <= 511; ++ic )
        vhash_round(vctx);
}

void vhash_final(vhash_ctx *vctx, uint32_t len)
{
    uint32_t f[4];
    f[0] = 2147483648;
    f[1] = len;
    f[2] = 0;
    f[3] = -1;
    vhash_update(vctx, (uint32_t (*)[4])f);
}

void print_hash(vhash_ctx *vctx)
{
    for (int i = 0; i <= 7; ++i )
        printf(
                "%08x%08x%08x%08x",
                vctx->state[4 * i],
                vctx->state[4 * i + 1],
                vctx->state[4 * i + 2],
                vctx->state[4 * i + 3]);
    printf("\n");
}

void process_data(const char *data, vhash_ctx* vctx, uint32_t &size, uint32_t len) {
    int done = 0;
    size= 0;
    uint32_t block[4];
    while ( !done )
    {
        memset(block, 0, 0x10uLL);
```

```
            for (int i = 0; (unsigned int)i <= 0xF; ++i )
            {
                if ( size<len )
                {
                    uint8_t inbyte = data[size];
                    ++size;
                    block[i / 4] |= inbyte << 8 * (3 - (char)i % 4);
                }
                else
                {
                    done = 1;
                    break;
                }
            }
            vhash_update(vctx, (uint32_t (*)[4])block);
            if(size==len){
                done=1;
            }
        }
    }

    void hash(const char* data, uint32_t len) {
        vhash_ctx vctx;
        vhash_init(&vctx);
        uint32_t size;
        process_data(data, &vctx, size, len);
        vhash_final(&vctx, size);
        print_hash(&vctx);
    }
```

A teraz mając powyższe prymitywy mogliśmy napisać funkcje generującą fałszywy hash:

```
void rigged(vhash_ctx *vectx, const char* data, uint32_t len){
    set_initial_state(vectx);
    uint32_t size;
    process_data(data, vectx, size, len);
    vhash_final(vectx, 120);
    print_hash(vectx);
}
```

Ustawiamy tutaj początkowy stan wzięty z cookie ze strony, następnie wykonujemy update z naszym dodatkowym payloadem a potem dodajemy finalny blok używając `120` jako długości całych danych, bo tyle będą teraz miały. Jest tak ponieważ dane które hashuje serwer mają postać:

```
12345678username=guest&date=2017-02-14T00:22:54+0100&secret_length=8&\0\0\0\0\0\0\0\0\0\0\0\x80\x00\x00\x0
```

zakładając `12345678` jako `secret`. Bajty `\0` biorą sie z paddingu dodanego do ostatniego bloku kiedy jest on hashowany a `\x80\x00\x00\x00\x00\x00\x00\x45\x00\x00\x00\x00\xff\xff\xff\xff` to finalny blok dodany przez serwer to początkowego cookie (0x45 to rozmiar danych w tym przypadku).

Użyliśmy krótkiego kodu pythona żeby wygenerować początkowy stan z cookie:

```
from crypto_commons.generic import chunk


def dump_state(data):
    print("\n".join(["vectx->state[" + str(i) + "] = 0x" + c + ";" for i, c in enumerate(chunk(data, 8))]))


def main():
    data = 'e4d6ed47e23694e76f8a611111fd7b52a980e40a718b28256342adc740814a09fa4592e434dbb0b2b5d3405dd554e5
    dump_state(data)

main()
```

a to daje nam zawartość dla funkcji `set_initial_state` :

```
void set_initial_state(const vhash_ctx *vectx) {
    vectx->state[0] = 0xe4d6ed47;
    vectx->state[1] = 0xe23694e7;
```

```
        vectx->state[2] = 0x6f8a6111;
        vectx->state[3] = 0x11fd7b52;
        vectx->state[4] = 0xa980e40a;
        vectx->state[5] = 0x718b2825;
        vectx->state[6] = 0x6342adc7;
        vectx->state[7] = 0x40814a09;
        vectx->state[8] = 0xfa4592e4;
        vectx->state[9] = 0x34dbb0b2;
        vectx->state[10] = 0xb5d3405d;
        vectx->state[11] = 0xd554e5b4;
        vectx->state[12] = 0x0f432cfb;
        vectx->state[13] = 0x7cbf6fe0;
        vectx->state[14] = 0x6d95ed16;
        vectx->state[15] = 0xcc1f1b21;
        vectx->state[16] = 0xf0d5f5b4;
        vectx->state[17] = 0x32b0e82c;
        vectx->state[18] = 0x9ce0693e;
        vectx->state[19] = 0xd4caca8c;
        vectx->state[20] = 0xf7a035e4;
        vectx->state[21] = 0x8a29fc35;
        vectx->state[22] = 0x8d75bb06;
        vectx->state[23] = 0x74d5e0b9;
        vectx->state[24] = 0x10252bfc;
        vectx->state[25] = 0x1712a766;
        vectx->state[26] = 0x98662a0e;
        vectx->state[27] = 0xa4441183;
        vectx->state[28] = 0xd4b3d7f1;
        vectx->state[29] = 0xade921ae;
        vectx->state[30] = 0x47f0b0c6;
        vectx->state[31] = 0x41e25813;
    }
```

Teraz zostaje już tylko uruchomić `rigged(&vctx,"&username=administrator&", 24);` a to wypisze nam poprawny hash.

Ostatni krok to wysłanie nowych danych razem z hashem na serwer:

```python
import urllib
import requests


def main():
    cookies = {"auth": urllib.quote_plus(
        "39005958a36fcfabc56482ef87d46c3e1aec3c7babbc487dfc73d5c48441f24e67ffb44a2814be8e2e890e282188430b44
    result = requests.get("http://thenewandlessbrokenvhash.ctf.bsidessf.net:9292/index.php", cookies=cookie
    print(result.content)


main()
```

Co daje name `FLAG:06c211f73f4f5ba198c7fb56145b39a2` w odpowiedzi.