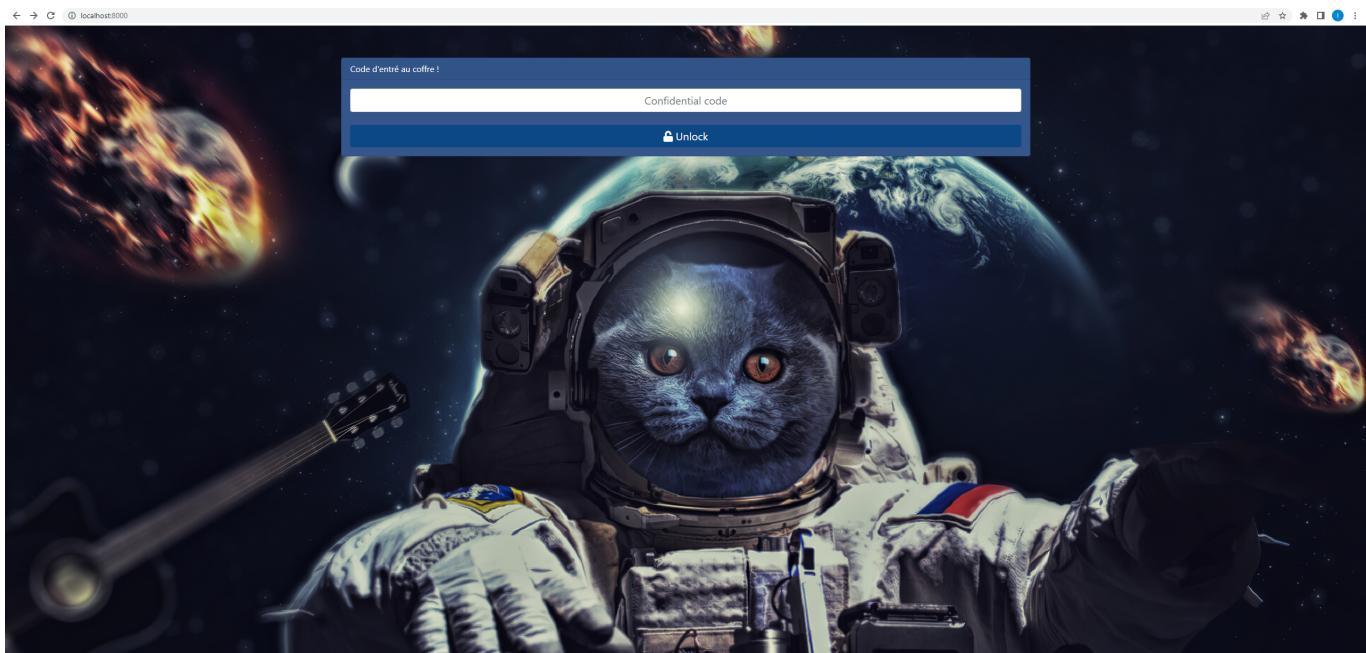


# Attack Equation Solving

Bienvenue à toi dans ma writeup jeune petit champion, si tu es ici c'est que malheureusement tu n'as pas réussi et j'en suis fort navré :)

Bon commençons ! Tout d'abord cela devrait vous intriguer mais lorsque l'on va sur le challenge de "Reverse" on a une interface web accompagné d'un binaire PE static.

Commençons par investiguer l'interface web.



En effet on peut voir que l'on peut y mettre un code afin d'accéder à un coffre fort. Hum voilà qui est fort intriguant.

Je vais jeter un coup d'oeil du côté du binaire.

```
C:\Users\bbern\OneDrive\Bureau\Attack Equation Solving>file attack_equation_solving.exe
attack_equation_solving.exe: PE32+ executable (console) x86-64 (stripped to external PDB), for MS Windows
```

J'ai par la suite commencé à désassembler le binaire afin de voir à quoi j'avais affaire !

En effet on peut remarquer assez facilement que nous avons affaire à un binaire en golang et donc statique.

Function Name	Segment	Start
f go_buildid	.text	0000000000000000
f internal_cpu_Initialize	.text	0000000000000000
f internal_cpu_processOptions	.text	0000000000000000
f internal_cpu_doinit	.text	0000000000000000
f internal_cpu_cpuid_abi0	.text	0000000000000000
f internal_cpu_xgetbv_abi0	.text	0000000000000000
f type_eq_internal_cpu_option	.text	0000000000000000
f type_eq_15_internal_cpu_option	.text	0000000000000000
f runtime_internal_atomic__Uint32_Load	.text	0000000000000000
f runtime_internal_atomic__Uint32_LoadAcquire	.text	0000000000000000
f runtime_internal_atomic__Uint32_Store	.text	0000000000000000
f runtime_internal_atomic__Uint32_StoreRelease	.text	0000000000000000
f runtime_internal_atomic__Uint32_CompareAndSwap	.text	0000000000000000
f runtime_internal_atomic__Uint32_CompareAndSwapRelease	.text	0000000000000000
f runtime_internal_atomic__Uint32_Swap	.text	0000000000000000
f runtime_internal_atomic__Uint32_And	.text	0000000000000000
f runtime_internal_atomic__Uint32_Or	.text	0000000000000000
f runtime_internal_atomic__Uint32_Add	.text	0000000000000000
f runtime_internal_atomic__Uintptr_Load	.text	0000000000000000
f runtime_internal_atomic__Uintptr_LoadAcquire	.text	0000000000000000
f runtime_internal_atomic__Uintptr_Store	.text	0000000000000000
f runtime_internal_atomic__Uintptr_StoreRelease	.text	0000000000000000
f runtime_internal_atomic__Uintptr_CompareAndSwap	.text	0000000000000000
f runtime_internal_atomic__Uintptr_Swap	.text	0000000000000000
f runtime_internal_atomic__Uintptr_Add	.text	0000000000000000
f runtime_internal_atomic__noCopy_Lock	.text	0000000000000000
f runtime_internal_atomic__noCopy_Unlock	.text	0000000000000000
f runtime_internal_sys_OnesCount64	.text	0000000000000000
f internal_abi_RegArgs_Dump	.text	0000000000000000
f internal_abi_RegArgs_IntRegArgAddr	.text	0000000000000000
f internal_abi_IntArgRegBitmap_Set	.text	0000000000000000
f internal_abi_IntArgRegBitmap_Get	.text	0000000000000000
f type_eq_internal_abi_RegArgs	.text	0000000000000000
f internal_bytealg_IndexRabinKarpBytes	.text	0000000000000000
f internal_bytealg_IndexRabinKarp	.text	0000000000000000
f internal_bytealg_countGeneric	.text	0000000000000000
f internal_bytealg_countGenericString	.text	0000000000000000
f internal_bytealg_init_0	.text	0000000000000000
f internal_bytealg_Compare	.text	0000000000000000
f runtime_cmpstring	.text	0000000000000000
f internal_bytealg_Count_abi0	.text	0000000000000000
f internal_bytealg_CountString_abi0	.text	0000000000000000
f runtime_memequal	.text	0000000000000000
f runtime_memequal_varlen	.text	0000000000000000
f internal_bytealg_Index_abi0	.text	0000000000000000
f internal_bytealg_IndexString_abi0	.text	0000000000000000
f internal_bytealg_IndexByte_abi0	.text	0000000000000000
f internal_bytealg_IndexByteString_abi0	.text	0000000000000000
f internal_bytealg_countGeneric_abi0	.text	0000000000000000
f internal_bytealg_countGenericString_abi0	.text	0000000000000000
f runtime_memhash8	.text	0000000000000000
f runtime_memhash16	.text	0000000000000000
f runtime_memhash128	.text	0000000000000000
f runtime_memhash_varlen	.text	0000000000000000
f runtime_strhashFallback	.text	0000000000000000
f runtime_f32hash	.text	0000000000000000
f runtime_f64hash	.text	0000000000000000
f runtime_c64hash	.text	0000000000000000
f runtime_c128hash	.text	0000000000000000
f runtime_interhash	.text	0000000000000000

Je vais filtrer afin d'avoir ma fonction main

Function name	Segment	Start
f runtime_main_func1	.text	000000000004394
f runtime_main	.text	000000000004394
f runtime_main_func2	.text	000000000004397
f crypto_x509_domainToReverseLabels	.text	000000000005D91
f crypto_x509_matchDomainConstraint	.text	000000000005DA
f vendor_golang_org_x_net_http_httpproxy_domainMatch_match	.text	0000000000064B0
f vendor_golang_org_x_net_http_httpproxy_domainMatch_...	.text	0000000000064C1
f type_eq_vendor_golang_org_x_net_http_httpproxy_domain...	.text	0000000000064C4
f net_http_validCookieDomain	.text	0000000000064F7
f net_http_isCookieDomainName	.text	0000000000064F8
f net_http_requestBodyRemains	.text	00000000000681E
f net_http_body_bodyRemains	.text	0000000000068C1
f net_http_body_bodyRemains_func1	.text	0000000000068CE
f main_KeyToMd5	.text	000000000008794
f main_GetAESKey	.text	00000000000879E
f main_DecryptAES	.text	00000000000879C
f main_IsValidCode	.text	00000000000879E
f main_BasePath	.text	00000000000879F
f main_LoginPage	.text	00000000000879F
f main_VerifyCode	.text	0000000000087A0
f main_main	.text	0000000000087

En filtrant on peut en effet voir que le chemin se raccourcis. Nous avons quelques fonctions de runtime ou de librairie externe qui sont importé dans le main. Mais ce n'est pas vraiment important. Ce qui nous intéresse nous c'est les fonctions à l'intérieur du main.

Déjà on peut observer une fonction qui selon moi renvoie un boolean si le code est valide. C'est donc pour moi ici que notre code entré en formulaire sera passé. Mais avant d'aller aussi loin analysons la fonction main.

```
void __cdecl main_main()
{
    __int64 v0; // r14
    __int64 HTMLGlob; // [rsp-40h] [rbp-60h]
    __int64 v2; // [rsp-40h] [rbp-60h]
    __int64 v3; // [rsp-38h] [rbp-58h]
    __int64 v4; // [rsp-38h] [rbp-58h]
    __int64 v5; // [rsp-30h] [rbp-50h]
    __int64 v6; // [rsp-30h] [rbp-50h]
    __int64 v7; // [rsp-28h] [rbp-48h]
    __int64 v8; // [rsp-28h] [rbp-48h]
    void *retaddr; // [rsp+20h] [rbp+0h] BYREF

    if ( (unsigned __int64)&retaddr <= *_QWORD *) (v0 + 16) )
        runtime_morestack_noctxt_abi0();
    github_com_gin_gonic_gin_Default();
    main_BasePath();
    runtime_concatstring2();
    HTMLGlob = github_com_gin_gonic_gin__Engine__LoadHTMLGlob();
    main_BasePath();
    runtime_concatstring2();
    github_com_gin_gonic_gin__RouterGroup__Static();
    main_BasePath();
    runtime_concatstring2();
    github_com_gin_gonic_gin__RouterGroup__Static();
    *_QWORD *runtime_newobject() = off_9950C0;
    github_com_gin_gonic_gin__RouterGroup__handle(HTMLGlob, v3, v5, v7);
    *_QWORD *runtime_newobject() = off_9950C8;
    github_com_gin_gonic_gin__RouterGroup__handle(v2, v4, v6, v8);
    github_com_gin_gonic_gin__Engine__Run();
}
```

En effet on voit l'utilisation d'une librairie externe qui s'appelle gin. C'est un framework web golang. J'en conclu donc que c'est le routeur de notre service web. Comme on peut le voir c'est un peu n'importe quoi le code c'est pourquoi je décide d'analyser la fonction VerifyCode.

```
void __fastcall main_VerifyCode()
{
    __int64 v0; // r14
```

```
_QWORD *v1; // rax
_QWORD *v2; // rax
_QWORD *v3; // rax
_QWORD *v4; // rax
_QWORD *v5; // rax
_QWORD *v6; // rax
__int64 v7; // [rsp-20h] [rbp-58h]
void *retaddr; // [rsp+38h] [rbp+0h] BYREF

if ( (unsigned __int64)&retaddr <= *(_QWORD *) (v0 + 16) )
    runtime_morestack_noctxt_abi0();
github_com_gin_gonic_gin___Context__GetPostForm();
strings_TrimSpace();
if ( "codecolscong" )
{
    if ( (unsigned __int8)main_IsValidCode() )
    {
        runtime_makemap_small();
        runtime_mapassign_faststr();
        *v3 = &unk_8ADD00;
        if ( runtime_writeBarrier )
            runtime_gcWriteBarrierDX();
        else
            v3[1] = &runtime_staticuint64s;
        v7 = runtime_mapassign_faststr();
        *v4 = &unk_8B3240;
        if ( !runtime_writeBarrier )
        {
            v4[1] = &off_A25890;
            goto LABEL_22;
        }
    }
    else
    {
        runtime_makemap_small();
        runtime_mapassign_faststr();
        *v5 = &unk_8ADD00;
        if ( runtime_writeBarrier )
            runtime_gcWriteBarrierDX();
        else
            v5[1] = &unk_CF33A8;
        v7 = runtime_mapassign_faststr();
        *v6 = &unk_8B3240;
        if ( !runtime_writeBarrier )
        {
            v6[1] = &off_A258A0;
            goto LABEL_22;
        }
    }
}
```

```

    }

LABEL_21:
    runtime_gcWriteBarrierCX();
    goto LABEL_22;
}

runtime_makemap_small();
runtime_mapassign_faststr();
*v1 = &unk_8ADD00;
if ( runtime_writeBarrier )
    runtime_gcWriteBarrierDX();
else
    v1[1] = &unk_CF33A8;
v7 = runtime_mapassign_faststr();
*v2 = &unk_8B3240;
if ( runtime_writeBarrier )
    goto LABEL_21;
v2[1] = &off_A25880;
LABEL_22:
    runtime_convT();
    github_com_gin_gonic_gin___Context__Render(v7);
}

```

Si l'on analyse un peu le code on voit en effet l'appelle de la fonction IsValidCode tout au dessus, j'en conclu donc que cette fonction est la fonction appelé par le routeur.  
 Analysons dès à présent la fonction IsValidCode

```

_BOOL8 __fastcall main_IsValidCode()
{
    __int64 v0; // rbx
    __int64 v1; // r14
    __int64 AESKey; // rbx
    __int64 v3; // rcx
    void *retaddr; // [rsp+28h] [rbp+0h] BYREF
    __int64 v6; // [rsp+38h] [rbp+10h]

    if ( (unsigned __int64)&retaddr <= *(_QWORD *) (v1 + 16) )
        runtime_morestack_noctxt_abi0();
    v6 = v0;
    AESKey = main_GetAESKey();
    runtime_stringtoslicebyte();
    main_DecryptAES();
    if ( v3 )
        return 0LL;
    return AESKey == v6 && (unsigned __int8) runtime_memequal();
}

```

En effet ce code fait déjà moins peur, on voit que cette fonction renvoie un Boolean, on peut également voir que celle-ci à pris un argument dans la variable v6. Je pense que l'on a affaire à notre code entré dans le formulaire.

Nous avons une variable AESKey qui appelle une fonction GetAESKey. Inutile de vous faire un dessin c'est de l'AES notre but actuellement va être de trouver la clé ainsi que la valeur chiffré.

Après un debugging on peut connaître facilement la valeur de retour de la fonction GetAESKey ainsi que la valeur chiffré passé à la fonction DecryptAES.

Cependant comme nous aimons nous compliquer la vie. Nous allons analyser l'algorithme qui Génère notre clé AES.

```
0087ad89      void var_18
0087ad89      while (&var_18 u<= *(arg3 + 0x10))
0087ae4f          arg2, arg1, arg4 = runtime.morestack_noctxt.abi0(arg1,
arg2)
0087ada6      int128_t var_7a = arg4
0087adac      var_7a = arg4
0087adbc      var_7a.q = 0x538d51947f538894
0087adc1      int64_t* rdx = 0x538e8f8d7f93517f
0087adcb      var_7a:8.q = 0x538e8f8d7f93517f
0087add0      int16_t var_6a = 0x7f99
0087adee      for (int64_t rax = 0; rax s< 0x12; rax = rax + 1)
0087ade0          rdx = zx.q(zx.d(*(&var_7a + rax)) - 0x20)
0087ade3          *(&var_7a + rax) = rdx.b
0087ae00      void var_28
0087ae00      uint64_t rax_2
0087ae00      int64_t rdx_2
0087ae00      int64_t rsi
0087ae00      int64_t rdi
0087ae00      int128_t zmm15
0087ae00      rax_2, rdx_2, rsi, rdi, zmm15 = runtime.slicebytetostring(arg1,
arg2, rdx, 0x12, &var_28, &var_7a, arg3)
0087ae10      void var_48
0087ae10      uint64_t rax_4
0087ae10      int64_t* rdx_3
0087ae10      void* rsi_1
0087ae10      int64_t rdi_1
0087ae10      rax_4, rdx_3, rsi_1, rdi_1 = runtime.stringtoslicebyte(rdi,
rsi, rdx_2, &var_7a, &var_48, rax_2, arg3, zmm15)
0087ae28      for (int64_t rcx_1 = 0; rax_2 s> rcx_1; rcx_1 = rcx_1 + 1)
0087ae1d          rdx_3 = zx.q(zx.d(*(&rax_4 + rcx_1)) ^ rcx_1.d)
0087ae1f          *(&rax_4 + rcx_1) = rdx_3.b
0087ae35      void var_68
```

```
0087ae35    int16_t* rax_6
0087ae35    int64_t rdx_5
0087ae35    int64_t rsi_2
0087ae35    int64_t rdi_2
0087ae35    rax_6, rdx_5, rsi_2, rdi_2 = runtime.slicebytetostring(rdi_1,
rsi_1, rdx_3, rax_2, &var_68, rax_4, arg3)
0087ae4e    return main.KeyToMd5(rdi_2, rsi_2, rdx_5, rax_6, rax_4, arg3)
```

Bon ça semble complèxe je vous l'accorde haha mais ça ne l'est pas.

On peut voir que nous avons un tableau de byte dans la variable v6.

Et on peut également voir une boucle qui va simplement soustraire 32 à chacun de ces bytes.

Scriptons cela:

```
def GetAESKey()
    v6_0 = [0x53, 0x8d, 0x51, 0x94, 0x7f, 0x53, 0x88, 0x94].reverse

    v6_1 = [0x53, 0x8e, 0x8f, 0x8d, 0x7f, 0x93, 0x51, 0x7f].reverse

    v6_2 = [0x99, 0x7f]

    v6 = ""

    v6_0.each do |byte|
        v6 += (byte-32).chr
    end

    v6_1.each do |byte|
        v6 += (byte-32).chr
    end

    v6_2.each do |byte|
        v6 += (byte-32).chr
    end

    key = ""

    v6.bytes.map.with_index do |byte, i|
```

```

key += (byte^i).chr

end

key = Digest::MD5.hexdigest(key)

return key
end

```

En effet j'ai reproduis assez facilement le générateur de clé

Si on analyse le code assembleur il prend un tableau de byte il soustrait chaque byte de 32 en base 10

```

0087adee    for (int64_t rax = 0; rax < 0x12; rax = rax + 1)
0087ade0        rdx = zx.q(zx.d(*(&var_7a + rax)) - 0x20)
0087ade3        *(&var_7a + rax) = rdx.b

```

Ensuite il va simplement xorer la chaîne de sortie avec son propre index

```

0087ae28    for (int64_t rcx_1 = 0; rax_2 > rcx_1; rcx_1 = rcx_1 + 1)
0087ae1d        rdx_3 = zx.q(zx.d(*(rax_4 + rcx_1)) ^ rcx_1.d)
0087ae1f        *(rax_4 + rcx_1) = rdx_3.b

```

Ensuite on peut voir qu'il appelle une fonction GetMD5 et nous avons donc la fameuse clé en md5

la voici: [4c474c5780c095ed2f186e8d6d439e0e](#)

Analysons maintenant la fonction isValidCode

```

0087ae69    while (true)
0087ae69        void* rsp
0087ae69        int128_t zmm15
0087ae69        if (rsp - 0x20 > *(arg6 + 0x10))
0087ae6f            rsp = rsp - 0xa0
0087ae76            *(rsp + 0x98) = arg5
0087ae7e            arg5 = rsp + 0x98
0087ae86            *(rsp + 0xa8) = arg3
0087ae8e            *(rsp + 0xb0) = arg4
0087ae96            int64_t rax

```

```
0087ae96          int64_t rdx_1
0087ae96          int64_t rsi
0087ae96          int64_t rdi
0087ae96          int128_t zmm15_1
0087ae96          rax, rdx_1, rsi, rdi, zmm15_1 = main.GetAESKey(arg1,
arg2, arg6, zmm15)
0087ae9b          *(rsp + 0x80) = rax
0087aea3          *(rsp + 0x50) = arg4
0087aeb0          arg4 = "59E2267F61917A7832D3608F6DDB2C61..."
0087aeb7          char* rax_1
0087aeb7          int64_t rcx_1
0087aeb7          int128_t zmm15_2
0087aeb7          rax_1, rcx_1, zmm15_2 = runtime.stringtoslicebyte(rdi,
rsi, rdx_1, 0x40, nullptr, "59E2267F61917A7832D3608F6DDB2C61...", arg6,
zmm15_1)
0087aecb          *(rsp + 0x78) = rax_1
0087aec1          *(rsp + 0x40) = rcx_1
0087aecf          arg3, arg2, arg1 = encoding/hex.Decode(rax_1,
"59E2267F61917A7832D3608F6DDB2C61...", rax_1,
"59E2267F61917A7832D3608F6DDB2C61...")
0087aed4          uint64_t rdx_2 = *(rsp + 0x40)
0087aee3          if (arg3 u<= rdx_2)
0087aee9          *(rsp + 0x48) = arg3
0087aef3          int16_t* rbx = *(rsp + 0x80)
0087af00          int64_t rax_3
0087af00          int64_t rcx_3
0087af00          int64_t rdx_3
0087af00          rax_3, rcx_3, rdx_3 =
runtime.stringtoslicebyte(arg1, arg2, rdx_2, *(rsp + 0x50), rsp + 0x58, rbx,
arg6, zmm15_2)
0087af1b          *(rsp + 0x78)
0087af20          *(rsp + 0x48)
0087af2a          int128_t* rax_5
0087af2a          int64_t rdx_4
0087af2a          int64_t rsi_2
0087af2a          void* rdi_3
0087af2a          int128_t zmm15_3
0087af2a          rax_5, rdx_4, rsi_2, rdi_3, zmm15_3 =
github.com/forgoer/openssl.AesECBDecrypt(rax_3, rbx, rdx_3, *(rsp + 0x40),
rcx_3, arg6)
0087af2f          bool cond:0 = rdi_3 == 0
0087af32          if (rdi_3 == 0)
0087af88          int64_t rcx_6 = *(rsp + 0xb0)
0087af93          if (0 == rcx_6 && runtime.memequal(rdi_3,
rsi_2, rdx_4, rcx_6, *(rsp + 0xa8), rax_5) != 0)
0087afc0          *(rsp + 0x98)
0087afcfc          return 1
0087afab          *(rsp + 0x98)
```

```

0087afba          return 0
0087af34          *(rsp + 0x88) = zmm15_3
0087af3d          if (not(cond:0))
0087af3f          rdi_3 = *(rdi_3 + 8)
0087af43          *(rsp + 0x88) = rdi_3
0087af4b          *(rsp + 0x90) = rsi_2
0087af53          os.Stdout
0087af71          fmt.Fprintln(1, 1, &go.itab.*os.File, io.Writer,
arg6)
0087af78          *(rsp + 0x98)
0087af87          return 0
0087afd3          runtime.panicSliceAcap(arg1, arg2, rdx_2, arg3)
? 0087afdf        *(rsp + 8) = arg3
0087afde          *(rsp + 0x10) = arg4
0087afe3          arg2, arg1, zmm15 = runtime.morestack_noctxt.abi0(arg1,
arg2)
0087afe8          arg3 = *(rsp + 8)
0087afed          arg4 = *(rsp + 0x10)

```

Ici on peut voir en effet qu'il tente de mettre de l'hexa en byte

```

arg3, arg2, arg1 = encoding/hex.Decode(rax_1,
"59E2267F61917A7832D3608F6DDB2C61...", rax_1,
"59E2267F61917A7832D3608F6DDB2C61...")

```

Je pense que nous avons affaire à notre fameux flag chiffré. Maintenant que nous avons les deux éléments pour déchiffrer notre data. il n'y a plus qu'a scripter ça.

```

require "openssl"

require "digest"

def GetAESKey()

v6_0 = [0x53, 0x8d, 0x51, 0x94, 0x7f, 0x53, 0x88, 0x94].reverse

v6_1 = [0x53, 0x8e, 0x8f, 0x8d, 0x7f, 0x93, 0x51, 0x7f].reverse

```

```
v6_2 = [0x99, 0x7f]
```

```
v6 = ""
```

```
v6_0.each do |byte|
```

```
  v6+= (byte-32).chr
```

```
end
```

```
v6_1.each do |byte|
```

```
  v6+= (byte-32).chr
```

```
end
```

```
v6_2.each do |byte|
```

```
  v6+= (byte-32).chr
```

```
end
```

```
key = ""
```

```
v6.bytes.map.with_index do |byte, i|
```

```
  key+= (byte^i).chr
```

```
end
```

```

key = Digest::MD5.hexdigest(key)

return key

end

def hexDecode(data)

data = data.scan(/../).map { |x| x.hex.chr }.join

return data

end

def aes256_decrypt(key, data)

decipher = OpenSSL::Cipher::AES.new(256, :ECB)

decipher.decrypt

decipher.key = key

plain = decipher.update(data) + decipher.final

return plain

end

key = GetAESKey()

puts "FLAG: #{aes256_decrypt(key,
hexDecode("59E2267F61917A7832D3608F6DDB2C6146E68ABBE82D0904CE10FE27BEBE4A54")
)}"

```

Flag: pr0\_cr4ck3r\_g000

Voilà en esperant que vous aurez apprécié le chall !!  
:))))

