MATRIX. Episodio 1.

Has llegado hasta aquí porque no te convencen los bandos. Sabes que los humanos son una forma de vida condenada a desaparecer, pero también te resistes a trabajar servilmente para las máquinas que controlan Matrix. Por suerte, no estás sólo. Hay alguien interesado en emancipar a quienes piensan como tú.

Estás ante un programa rebelde que fue interceptado antes de que se borrara. Haberlo interceptado nos convierte en renegados, pero la información que contenía dicho programa era demasiado valiosa como para no desobedecer las normas.

El programa contiene los códigos de acceso a uno de los servidores críticos de la ciudad. Dicho servidor es el centro neurálgico
de la infraestructura dedicada a la información, donde se almacenan todas las comunicaciones. He conseguido establecer información valiosa para nuestra causa en ese servidor, pero está encriptada. Si tomas las decisiones correctas, llegarás hasta mí.

Servidor crítico: http://34.247.69.86/matrix/episodio1/index.php

Info: La flag tiene el formato UAM{md5}

Resolución

Descargamos el fichero ***getcode.***

Analizamos

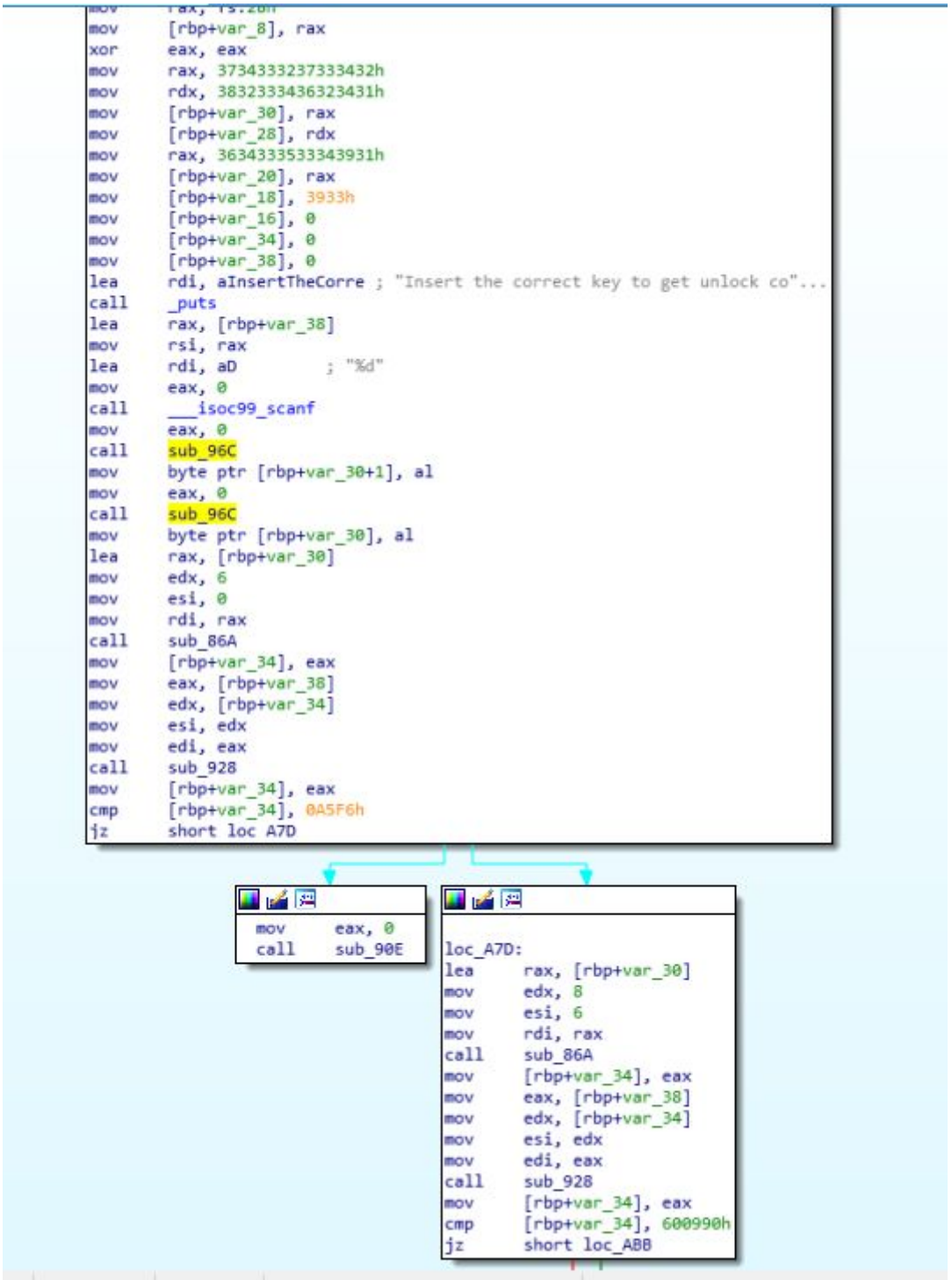
file getcode

getcode: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 3.2.0, BuildID[sha1]=fa966968762bff2e58360b56ac30754d91ebaa25, stripped

Probamos funcionamiento:
./getcode

Insert the correct key to get unlock code:
11223344
Incorrect key!

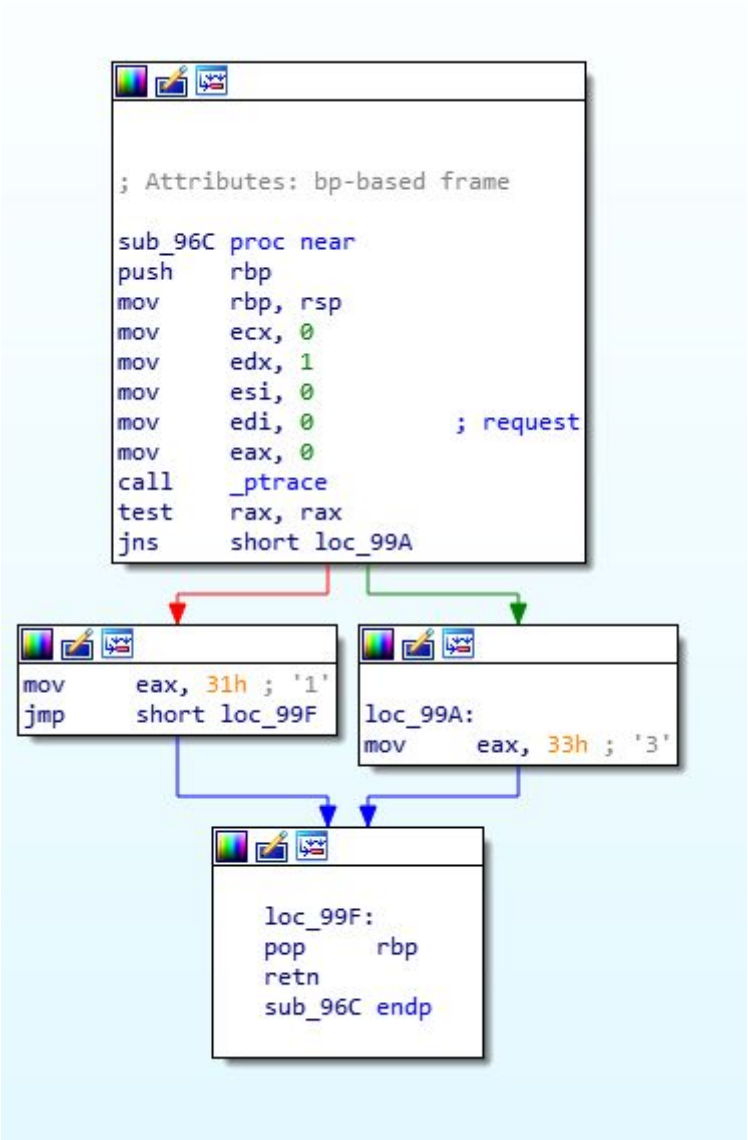
Debemos encontrar el código correcto, pasamos al análisis con Ida64



Nos fijamos en un par de cosas interesantes:

Una inicialización con una cadena de caracteres: 373433…….
Tres funciones a determinar funcionamiento, sub_96C, sub_86A, sub_928,
Y diversas comprobaciones después de la función 928...

sub_96C



Una llamada a ptrace y luego modifica el valor en base al resultado.

Parece que se trata de una pequeña técnica antidebug,

https://www.aldeid.com/wiki/Ptrace-anti-debugging

Desde un debbuger pondrá en eax un ’1’ en otro caso ‘3’.

Tendremos que modificar este comportamiento devolviendo siempre ‘3’.

sub_86a

Esta función en ensamblador se complica un poco más, diversas llamadas relacionadas con cadenas, una función de potencia (pow….)

Pasamos a ver el comportamiento de la función con radare:

r2 -d ./getcode
[0x7f87b5f96090]> **aaa**
[0x7f87b5f96090]> **pdf @main**

```
            0x560bad5709db      48b831393433.  movabs rax, 0x3634333533343931
            0x560bad5709e5      488945e0       mov qword [local_20h], rax
            0x560bad5709e9      66c745e83339   mov word [local_18h], 0x3933
            0x560bad5709ef      c645ea00       mov byte [local_16h], 0
            0x560bad5709f3      c745cc000000.  mov dword [local_34h], 0
            0x560bad5709fa      c745c8000000.  mov dword [local_38h], 0
            0x560bad570a01      488d3d200200.  lea rdi, qword str.Insert_the_correct_key_to_get_unlock
code: ; 0x560bad570c28 ; "Insert the correct key to get unlock code: "
            0x560bad570a08      e8d3fcffff     call sym.imp.puts       ; int puts(const char *s)
            0x560bad570a0d      488d45c8       lea rax, qword [local_38h]
            0x560bad570a11      4889c6         mov rsi, rax
            0x560bad570a14      488d3d390200.  lea rdi, qword [0x560bad570c54] ; "%d"
            0x560bad570a1b      b800000000     mov eax, 0
            0x560bad570a20      e80bfdffff     call sym.imp.__isoc99_scanf
            0x560bad570a25      b800000000     mov eax, 0
            0x560bad570a2a      e83dffffff     call 0x560bad57096c
            0x560bad570a2f      8845d1         mov byte [local_2fh], al
            0x560bad570a32      b800000000     mov eax, 0
            0x560bad570a37      e830ffffff     call 0x560bad57096c
            0x560bad570a3c      8845d0         mov byte [local_30h], al
            0x560bad570a3f      488d45d0       lea rax, qword [local_30h]
            0x560bad570a43      ba06000000     mov edx, 6
            0x560bad570a48      be00000000     mov esi, 0
            0x560bad570a4d      4889c7         mov rdi, rax
            0x560bad570a50      e815feffff     call 0x560bad57086a
            0x560bad570a55      8945cc         mov dword [local_34h], eax
            0x560bad570a58      8b45c8         mov eax, dword [local_38h]
            0x560bad570a5b      8b55cc         mov edx, dword [local_34h]
            0x560bad570a5e      89d6           mov esi, edx
            0x560bad570a60      89c7           mov edi, eax
            0x560bad570a62      e8c1feffff     call 0x560bad570928
```

Ponemos punto de interrupción justo después del scanf:

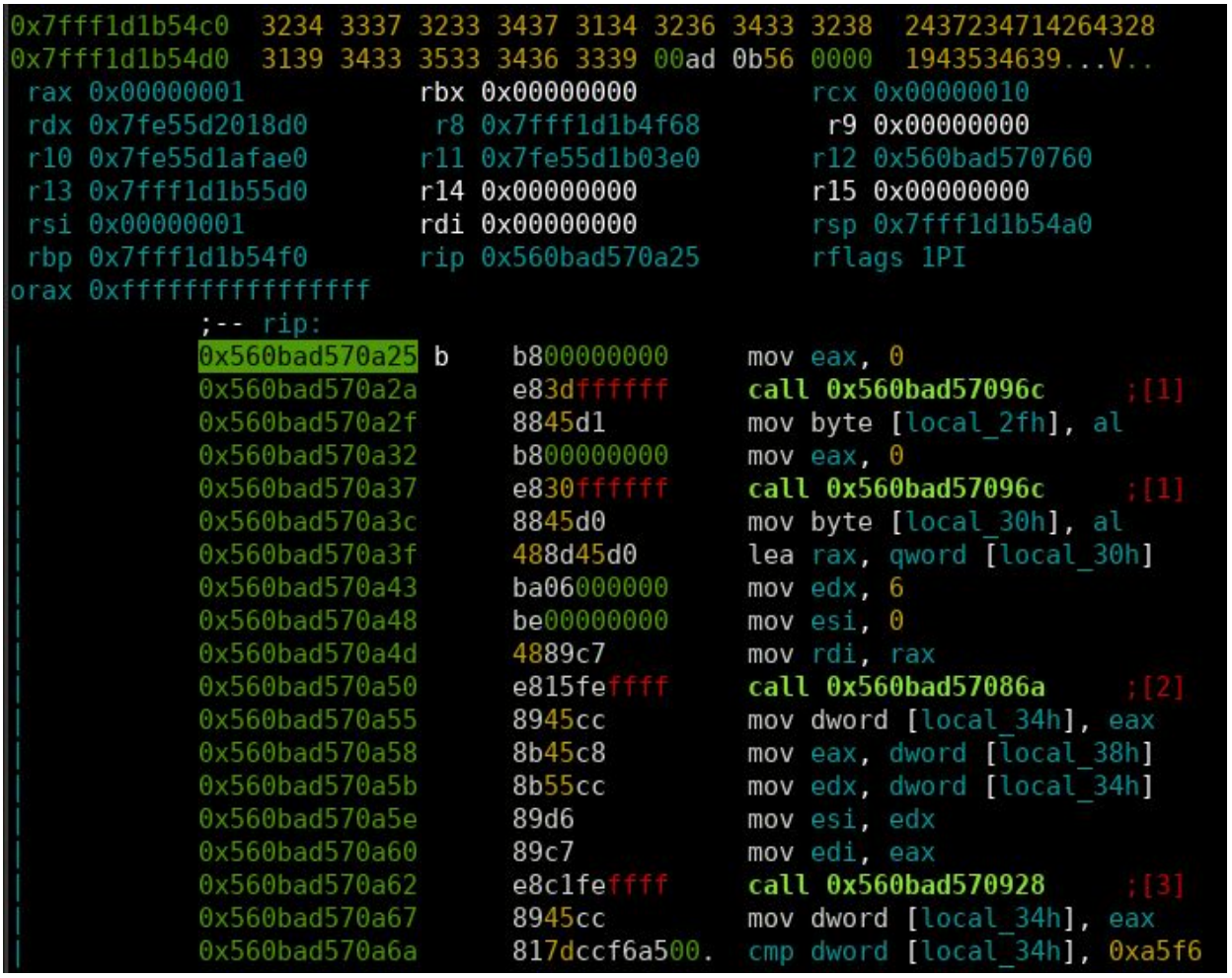
db 0x564ece408a25

dc (Ejecutamos)

child stopped with signal 28
[+] SIGNAL 28 errno=0 addr=0x00000000 code=128 ret=0

dc (Ejecutamos)
Insert the correct key to get unlock code:
11223344
hit breakpoint at: 564ece408a25

Pasamos al modo Visual **Vpp**



En la parte superior tenemos la cadena inicializada en código:
“2437234714264328 1943534639”

Tras algunos F8, vemos que la cadena se modifica por “**11**37234714264328 1943534639”

Este es el comportamiento esperado de la función sub_96C (0x564ece40896c) cuando ejecutamos desde un depurador.

Tras la llamada **call 0x564ece40886a**, obtenemos **rax 0x0001bc3b**

Y este número **0x0001bc3b** en decimal es: **113723.** (**:? 0x0001bc3b**)

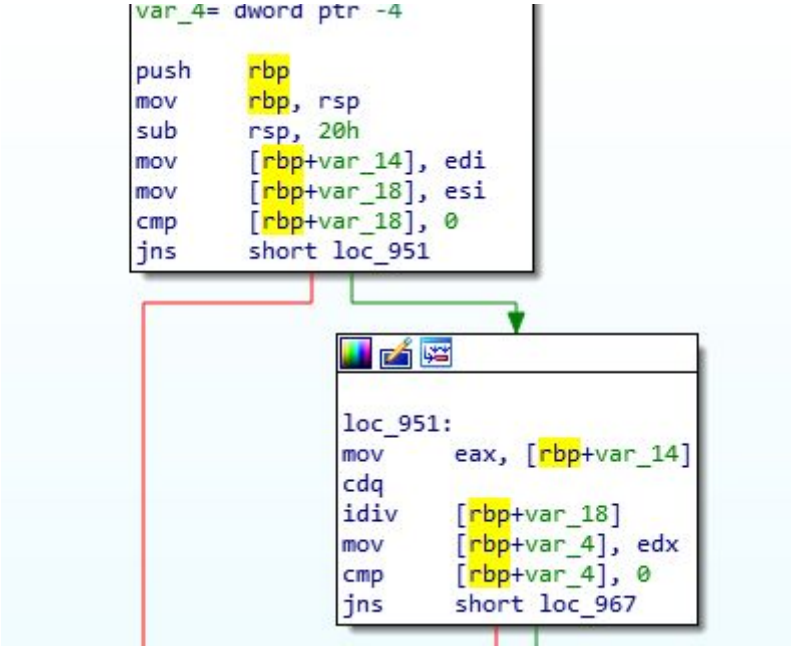
Pues la función al final no es tan complicada, parece que coge la cadena, comenzando en *esi(0)*, tantos caracteres como indica edx(6) y luego devuelve el resultado numérico en eax.

sub_86a

```
rax 0x00ab4130      rbx 0x00000000      rcx 0x7fe55d2c7720
rdx 0x0001bc3b       r8 0xffffffff       r9 0x40080000
r10 0x40080000      r11 0x00000286      r12 0x560bad570760
r13 0x7fff1d1b55d0  r14 0x00000000      r15 0x00000000
rsi 0x0001bc3b      rdi 0x00ab4130      rsp 0x7fff1d1b54a0
rbp 0x7fff1d1b54f0  rip 0x560bad570a62  rflags 1PZI
orax 0xffffffffffffffff
|           0x560bad570a58      8b45c8          mov eax, dword [local_38h]
|           0x560bad570a5b      8b55cc          mov edx, dword [local_34h]
|           0x560bad570a5e      89d6            mov esi, edx
|           0x560bad570a60      89c7            mov edi, eax
|           ;-- rip:
|           0x560bad570a62      e8c1feffff      call 0x560bad570928     ;[1]
|           0x560bad570a67      8945cc          mov dword [local_34h], eax
|           0x560bad570a6a      817dccf6a500.   cmp dword [local_34h], 0xa5f6    ; [0xa5f6:4]=-1
```

Observamos los parámetros de entrada de la función:

eax = 0x00ab4130 (decimal 11223344 Los caracteres que introdujimos inicialmente)
edx = 0x0001bc3b (113723)



Podemos ver que se realiza una llamada a idiv, nos hace suponer que puede ser el resultado de la división entre los parámetros de entrada.

Tas ejecutar con F8 la llamada, obtenemos:

rax 0x0001329a (78490)

Realizamos la operación de división en:

https://www.calculator.net/hex-calculator.html?number1=ab4130&c2op=%2F&number2=1bc3b&calctype=op&x=51&y=9

Hex value:
ab4130 ÷ 1bc3b = **62 Remainder : 1329A**

Pues ya sabemos lo que realiza la función, nos devuelve el ***resto*** de dividir eax entre edx.

Y este resto, es el que se compara en el código con distintos valores.

```
cmp    [rbp+var_34], 0A5F6h        42486
cmp    [rbp+var_34], 600990h       6293904
cmp    [rbp+var_34], 0EAh          234
cmp    [rbp+var_34], 1DDC5EDh      31311341
```

Para calcular el código correcto, tendremos que obtener los divisores, los restos ya los tenemos.

“333723471426432819435346 39”

Primer divisor (0,6): **333723**
Segundo (6,8) : **47142643**
Tercero (Eh,4): **2819**
Cuarto (12h,8): **43534639**

Luego tenemos

Quinto (4,9) : **234714264** XOR “código_introducido”

Para resolver tendríamos las siguientes ecuaciones: D = dc + R

(1) X = 333726 * a + 42486
(2) X = 47142643 * b + 6293904
(3) X = 2819 *c + 234
(4) X = 43534639 * d + 31311341

Para resolverlo, de una manera rápida, podemos partir de la última ecuación, ya que el multiplicador es un número grande, por lo que d no debe ser muy alto (desbordamiento) y darle valores a d hasta encontrar algún valor que cumpla la tercera ecuación. Para ello creamos un pequeño programa en C y ejecutamos online:

```
#include <stdio.h>

int main()
{
   long a,b,c,r,x;
   int i;

   a=43534639;
   b=31311341;
   c=2819;
   for (i=0;i<10000;i++) {
      x=a*i + b;
      r=x%c;
      if (r==234) printf("i=%d n=%d\r\n",i,x);
   }
   return 0;
}
```

https://www.tutorialspoint.com/compile_c_online.php

$gcc -o main *.c

$main

i=20 n=**902004121**
i=2839 n=-927900122 (desbordamiento)
i=5658 n=1537162931
i=8477 n=-292741312

902004121 XOR **234714264 = 943589633**

Ya tenemos el código, lo comprobamos en el ejecutable:

./getcode
Insert the correct key to get unlock code:
902004121
Correct key!
Here is your unlock code: **943589633**

http://34.247.69.86/matrix/episodio1/index.php

Probamos el código y nos aparece un link de descarga:

Descarga de comunicaciones:
https://drive.google.com/open?id=1CmHuHxIPXJz5uqz6KyhYblAcwiu14jfz

Descargamos el fichero Archives.zip. Descomprimimos y obtenemos 2 ficheros:

Usuario.zip (contiene fichero flaguser.txt pero está cifrado)
sniff.pcapng

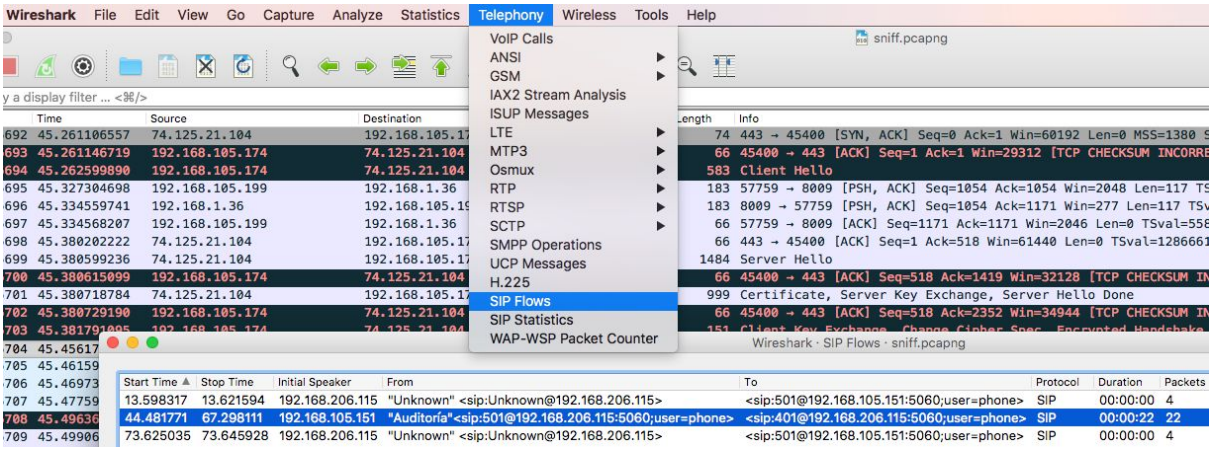
Analizamos con Wireshark. Estadísticas por protocolo

▼ Internet Protocol Version 4	99.4	16928	7.9	338568
▼ Transmission Control Protocol	51.0	8683	62.1	2666446
Secure Sockets Layer	21.6	3674	58.3	2504659
SSH Protocol	4.4	751	2.7	118071
Apache JServ Protocol v1.3	0.0	4	0.1	3088
▼ User Datagram Protocol	48.4	8237	1.5	65896
Real-Time Transport Protocol	38.0	6460	17.9	767920
Domain Name System	9.1	1544	2.8	121824
GQUIC (Google Quick UDP Internet Connections)	0.6	105	1.4	61248

Tenemos que el mayor porcentaje de paquetes se distribuye en SSL y RTP.
Damos un vistazo a RTP y vemos que se trasmite entre 192.168.206.115 - 192.168.105.151

5704	45.456175359	192.168.206.115	192.168.105.151	SIP/SDP	960	Status: 200 OK |
5705	45.461597352	192.168.206.115	192.168.105.151	SIP/SDP	960	Status: 200 OK |
5706	45.469734168	192.168.105.151	192.168.206.115	SIP	728	Request: ACK sip:401@192.168.206.115:5060;user=phone |
5707	45.477592874	192.168.105.151	192.168.206.115	SIP	728	Request: ACK sip:401@192.168.206.115:5060;user=phone |
5708	45.496369754	3comEuro_6b:54:18	Spanning-tree-(for-bridge…	STP	64	RST. Root = 32768/0/20:fd:f1:6b:54:12 Cost = 0 Port =
5709	45.499061158	74.125.21.104	192.168.105.174	TLSv1.2	109	Change Cipher Spec, Encrypted Handshake Message
5710	45.499662702	192.168.105.174	74.125.21.104	TLSv1.2	111	Application Data
5711	45.500165325	192.168.105.174	74.125.21.104	TLSv1.2	114	Application Data
5712	45.500270202	192.168.105.174	74.125.21.104	TLSv1.2	100	Application Data
5713	45.500397923	192.168.105.174	74.125.21.104	TLSv1.2	127	Application Data
5714	45.528832857	Vmware_8a:80:8f	SuperMic_32:2c:6d	ARP	42	192.168.105.151 is at 00:0c:29:8a:80:8f
5715	45.529036769	Vmware_8a:80:8f	ThomsonT_00:60:fb	ARP	42	192.168.105.1 is at 00:0c:29:8a:80:8f (duplicate use of
5716	45.596784218	192.168.206.115	192.168.105.151	RTP	214	PT=ITU-T G.711 PCMU, SSRC=0x1E9D509A, Seq=45881, Time=2
5717	45.597706862	192.168.206.115	192.168.105.151	RTP	214	PT=ITU-T G.711 PCMU, SSRC=0x1E9D509A, Seq=45881, Time=2
5718	45.615852362	192.168.206.115	192.168.105.151	RTP	214	PT=ITU-T G.711 PCMU, SSRC=0x1E9D509A, Seq=45882, Time=2

Antes de los paquetes RTP. observamos el establecimiento de la comunicación mediante protocolo SIP. User=Phone. Comprobemos si tenemos una comunicación VOIP.



Podemos escuchar un audio de 22 segundos de Morfeo.

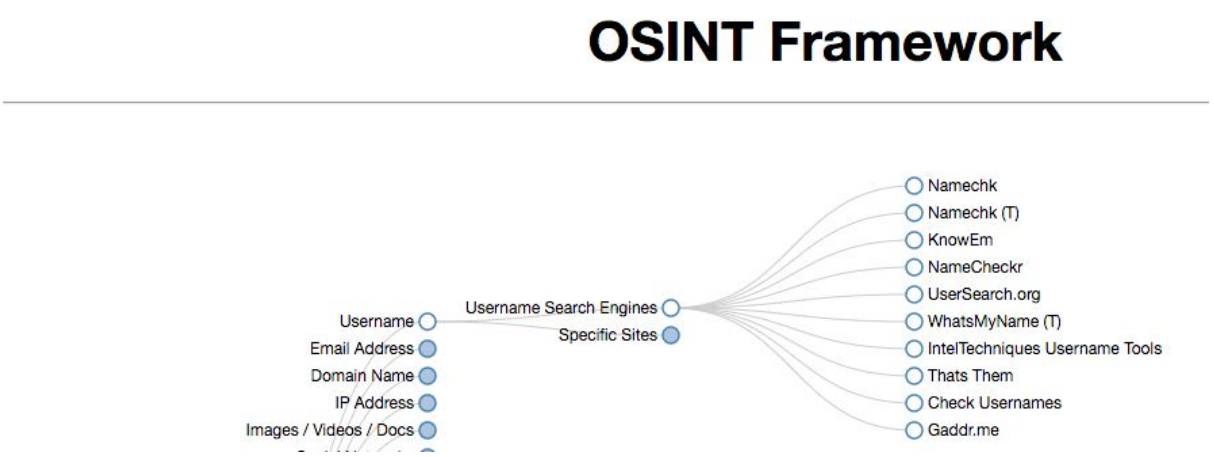
[...] si tomas la pastilla azul, fin de la historia...
[...] si tomas la roja te quedarás en el país de las maravillas...

Probamos en el zip “Usuario” la password “**pastillaroja**” y con éxito, descomprimimos flaguser.txt.

El fichero solo contiene la palabra **@agentesmith1337x**

Parece que tenemos que realizar una pequeña búsqueda OSINT

https://osintframework.com/



https://namechk.com/

Y en instagram tenemos:
https://www.instagram.com/agentesmith1337x/

UAM{bb48d678b6126102238509a886c1e299}

UAM{bb48d678b6126102238509a886c1e299}
Found : Era_inevitable_señor_Anderson

@bicacaro