

### Файлы задачи:

runner.py (MD5: 5f258c079c4accd789a6ea26f4c58ffe) - запуск задачи

Shuffle.рус (MD5: 196e859c79a612185d12b09930e2a82b) - проверка ключа, расшифровка флага

### Инструменты для решения:

1. Любой hex-редактор. В данном случае очень хорошо подходит 010editor (<https://www.sweetscape.com/>), так как есть соответствующий парсер для рус файла Python версии 2.7.

2. Декомпилятор для байткода питона (с возможностью выводить инструкции байткода - тот же pycdas (<https://github.com/zrax/pycdas>))

3. Python 2.7

### Описание:

Основным файлом является Shuffle.рус, где реализована проверка ключа и непосредственно расшифровка флага. Байткод программы собран таким образом, чтоб сразу не дать возможность преобразовать в исходный код, поэтому задача сводится к анализу инструкций байткода и восстановлению правильной структуры программы для декомпиляции.

### Решение:

При запуске скрипта runner.py, который в свою очередь запускает наш основной файл Shuffle.рус, запрашивается ключ (в 16-ричном формате) для расшифровки флага:

```
#####
#####
#####
## 8888888b. 8888888b. .d8888b. .d8888b. .d8888b. .d8888b. ##
## 888 R88b 888 "R88b d88G R88b d88G R88b d88G R88b d88G R88b ##
## 888 888 888 888 888 888 888 888 888 888 .d88G ##
## 888 d88G 888 888 888 .d88G 888 888 .d88G 8888" ##
## 88888888G" 888 888 888 888888 .od888G" 888 888 .od888G" "R8b. ##
## 888 T88b 888 888 888 888 d88G" 888 888 d88G" 888 888 ##
## 888 T88b 888 .d88G R88b d88G 888" R88b d88G 888" R88b d88G ##
## 888 T88b 88888888G" "R88888G88 888888888 "R8888G" 8888888888 "R8888G" ##
#####
##### Created by @R4d1X #####
#####
### Need key to decrypt flag:
Decryption key (in hex) = aaaaaaaaaaaaaa
Traceback (most recent call last):
  File "runner.py", line 9, in <module>
    exec code
  File "<to_deep>", line 8, in <module>
  File "<ctf_protected>", line 45, in <module>
  File "0", line 4, in decrypt2
  File "0", line 6, in error
NameError: [-] Wrong key or cipher len
#####
```

Смотрим запускающий скрипт `runner.py`:

```
# pain ahead
import marshal

f = open("Shuffle.pyc")
f.seek(8)
code = marshal.load(f)
f.close()

exec code
```

В данном случае никакой полезной информации для решения данный скрипт не дает.

Анализируем файл `Shuffle.py`. С помощью стандартной утилиты `file` можно понять, что перед нами скомпилированный питоновский код версии 2.7:

```
> file Shuffle.pyc
Shuffle.pyc: python 2.7 byte-compiled
```

Попытки декомпиляции приводят к неудаче из-за наличия в байткоде неверных инструкций (opcode):

```
> pycdc Shuffle.pyc
# Source Generated with Decompyle++
# File: Shuffle.pyc (Python 2.7)

Unsupported opcode: <255>
# WARNING: Decompyle incomplete
>
```

Открываем файл в hex-редакторе (в данном случае использовался [hiew](#)). Первый взгляд на raw-данные дает понять о том, что перед нами либо зашифрованные, либо сжатые данные (так как имеют достаточно беспорядочную структуру; чаще всего, если полезные данные ничем не обработаны, то можно отследить некоторую зависимость – в части это касается байткода или ассемблерного кода):

Ради примера можно сравнить с необработанным байткодом:

```

00000000: 63 00 00 00-00 00 00 00-00 02 00 00-00 40 00 00 c   6   @
00000010: 00 73 92 00-00 00 64 00-00 64 01 00-6C 00 00 5A sT   d   d  l  Z
00000020: 00 00 64 00-00 64 01 00-6C 01 00 5A-01 00 64 00 d   d  l  Z  d
00000030: 00 64 01 00-6C 02 00 5A-02 00 64 00-00 64 01 00 d  l  Z  d  d
00000040: 6C 03 00 5A-03 00 64 00-00 64 01 00-6C 04 00 5A l  Z  d  d  l  Z
00000050: 04 00 64 00-00 64 01 00-6C 05 00 5A-05 00 64 02  d  d  l  Z  d
00000060: 00 84 00 00-5A 06 00 64-03 00 84 00-00 5A 07 00  l  Z  d  l  Z-
00000070: 64 04 00 84-00 00 5A 08-00 64 05 00-84 00 00 5A d  l  Z  d  l  Z
00000080: 09 00 64 06-00 84 00 00-5A 0A 00 64-07 00 84 00  o  d  l  Z  d-  l
00000090: 00 5A 0B 00-64 0B 00 84-00 00 5A 0C-00 65 09 00 Z  d  l  Z  e  o
000000A0: 83 00 00 01-64 01 00 53-28 09 00 00-00 69 FF FF  r   d  S(   i
000000B0: FF FF 4E 63-00 00 00 00-00 00 00 00-01 00 00 00 Nc
000000C0: 43 00 00 00-73 04 00 00-00 64 00 00-53 28 01 00 C   s  d  S(
000000D0: 00 00 4E 28-00 00 00 00-28 00 00 00-00 28 01 00 N(   (
000000E0: 00 00 28 00-00 00 00 73-04 00 00 00-66 69 6C 65 (   s  file
000000F0: 74 09 00 00-00 70 72 65-73 73 5F 70-79 63 0A 00 to   press_pyc
00000100: 00 00 73 02-00 00 00 00-01 63 01 00-00 00 04 00 s   c
00000110: 00 00 06 00-00 00 03 00-00 00 73 49-00 00 00 64  s   c   sT   d
00000120: 01 00 7D 01-00 64 02 00-7D 02 00 78-36 00 74 00  d  d  x6 t
00000130: 00 7C 00 00-83 01 00 44-5D 28 00 7D-03 00 7C 01 |   r  D](  v  l
00000140: 00 74 01 00-7C 03 00 7C-02 00 64 03-00 16 41 83 t  |  d  -Ar
00000150: 01 00 37 7D-01 00 7C 02-00 64 04 00-37 7D 02 00 7  |  d  7
00000160: 71 19 00 57-7C 01 00 53-28 05 00 00-00 4E 74 00 q  W|  S(   Nt
00000170: 00 00 00 69-00 00 00 00-69 FF 00 00-00 69 01 00 i   i   i
00000180: 00 00 28 02-00 00 00 74-09 00 00 00-62 79 74 65 (   to   byte
00000190: 61 72 72 61-79 74 03 00-00 00 63 68-72 28 04 00 arrayt  chr(
000001A0: 00 00 74 04-00 00 00 64-61 74 61 74-03 00 00 00 t  datat
000001B0: 72 65 73 74-05 00 00 00-69 6E 64 65-78 74 03 00 rest  indext
000001C0: 00 00 6F 6E-65 28 00 00-00 00 28 00-00 00 00 73 one(   s
000001D0: 04 00 00 00-66 69 6C 65-74 07 00 00-00 65 6E 63  file-  enc

```

Отследим наличие строк в данном рус-файле. Согласно структуре рус-файла (не важно какой версии питона) [ссылка 1], все константы, переменные, строки, импортируемые библиотеки, используемые в ходе выполнения, помещаются в конце рус-файла (то есть сразу после инструкций байткода):

```
||| zlibt  
||| syst  
||| marshalt  
||| rest  
||| bytearrayt  
||| _getframet  
||| f_codet  
||| co_codet  
||| chrt  
||| loadst  
||| decompress(•  
||| <to_deep>t  
||| <module>  
||| HCCHED•
```

Видим знакомые [по крайней мере данная информация легко гуглится] библиотеки и вызова, что дает нам понять, что в программе происходит работа со сжатыми данными (библиотека `zlib`, `decompress`) и байткодом (библиотека `marshal`, `f_code`, `co_code`, `load`).

Так как непосредственная декомпиляция рус-файла не помогла, то стоит попробовать обратиться к выводу инструкций байткода (с помощью `pycdas` [идет вместе с декомпилятором `rusdc`]). Также парсинг инструкции байткода легко сделать самому, так как соответствующий инструментарий встроен в `python` в виде отдельных библиотек (например, `dis`):

```
> pycdas Shuffle.pyc > Shuffle.pycasm  
>
```

Перенаправляем вывод в файл **Shuffle.pycasm** для более удобного анализа (вывод будет достаточно большим).

Смотрим полученный листинг инструкций байткода. Выполнение кода происходит непосредственно с самого начала блока байткода (в данном случае с инструкции **SETUP\_EXCEPT**):

38		[Disassembly]	
39	1 → 0	SETUP_EXCEPT	14241 (to 14244) 2
40	3	<INVALID>	3
41	4	SETUP_LOOP	55197 (to 55204)
42	7	<INVALID>	
43	8	POP_TOP	
44	9	JUMP_ABSOLUTE	20050
45	12	<INVALID>	
46	13	<INVALID>	
47	14	JUMP_IF_TRUE_OR_POP	53273
48	17	SLICE_2	
49	18	<INVALID>	
50	19	<INVALID>	
51	20	<INVALID>	
52	21	PRINT_ITEM_TO	
53	22	DELETE_GLOBAL	36892 <INVALID>
54	25	<INVALID>	
55	26	CALL_FUNCTION_KW	25102
56	29	SETUP_EXCEPT	33436 (to 33468)
57	32	<INVALID>	
58	33	DELETE_NAME	7814 <INVALID>

1 - текущий offset инструкции (в данном случае SETUP\_EXCEPT) в блоке кода

2 - указывает на offset инструкции, куда перейдет управление программы в случае появления исключения (для удобства вычислен дизассемблером pycdas)

3 - offset до инструкции; показывает фактическое смещение вперед от текущий инструкции без учета размера самой инструкции (стандартный размер SETUP\_EXCEPT в Python 2.7 = 3 байта).

Таким образом данная инструкция даёт нам понять, что если далее в коде произойдет ошибка, то ход выполнения программы перейдет на инструкцию с offset-ом 14244.

Как видим, следующий opcode является некорректным - <INVALID>, что соответствует байту 0xff (2) согласно байтам инструкций:

00000000:	03 F3 0D 0A-92 3B 19 64-63 00 00 00-00 00 00 00	▼eJmT;Jdc
00000010:	00 0A 00 00-00 40 00 00-00 73 3E 38-00 00 79 A1	Ъ @ s>8 y6
00000020:	37 FF 78 9D-D7 BF 01 71-52 4E FF 26-70 19 D0 20	7 x3  _@qRN &p↓
00000030:	9E EF 31 49-62 1C 90 FA-8D 0E 62 79-9C 82 E5 5B	Юя1IbLP·НЪ bybBx[
00000040:	86 1E 66 1B-03 59 1E 63-8A C5 09 07-EE F6 B9 95	ЖΔf-▼YΔcK†○·юŸ  X
00000050:	09 EB BD 89-29 42 B7 3D-47 40 1B 27-D4 E2 DB 49	Оы  й)В  =GQ←·LТ  I
00000060:	09 83 B7 BE-FD 34 AB FF-AD FA BD 9F-A1 5F F3 73	СГ  J#4Л Н·  Я6_ε5

1 - начало блока байт кода. Первая инструкция SETUP\_EXCEPT (3 байта = 0x79 0xA1 0x37)

2 - <INVALID> инструкция

Эта инструкция и вызвала ошибку при попытке декомпилировать код.

Понимаем, что эта инструкция также вызовет ошибку в ходе выполнения программы и управление перейдет к инструкции с offset-ом 14244. Смотрим какой код там расположен:

10330	14235	<INVALID>	
10331	14236	<INVALID>	
10332	14237	BINARY_ADD	
10333	14238	<INVALID>	
10334	14239	<INVALID>	
10335	14240	STORE_FAST	44899 <INVALID>
10336	14243	STORE_DEREF	100 <INVALID>
10337	14246	STOP_CODE	
10338	14247	LOAD_CONST	1: None
10339	14250	IMPORT_NAME	0: zlib
10340	14253	STORE_NAME	0: zlib
10341	14256	LOAD_CONST	0: -1
10342	14259	LOAD_CONST	1: None
10343	14262	IMPORT_NAME	1: sys
10344	14265	STORE_NAME	1: sys
10345	14268	LOAD_CONST	0: -1
10346	14271	LOAD_CONST	1: None

Видим, что декомпилятор некорректно отобразил инструкции байткода в этой области и поэтому инструкции на offset-е 14244 нет (это связано с наложением инструкций разного размера друг на друга). С помощью hex-редактора пропатчим эту область таким образом, чтоб дизассемблер отобразил корректные инструкции:

(От начала байткода в рус-файле [offset = 0x1E] отсчитываем 14244 байт [0x1E + 14244 = 0x37C2])

000036E0:	A6 07 14 25-3C E2 84 83-5F FB 34 B6-83 BA 89 AE	ж-¶%<тДГ√4  Г  Ю
000036F0:	89 0E 0B 12-81 75 5E 39-5D E9 2A 75-45 DA 8B BC	й-ηзБу^9]ш*иЕп
00003700:	CF 10 15 FD-8C DE 6A 49-28 4E F5 64-8E 73 7C F3	—§иН  jI(NX'd0s e
00003710:	8C 89 3C FC-FE 67 C1 C7-3B 67 63 62-78 73 6A 62	МЙ<и¶g-  ;gcbxsjb
00003720:	AB 97 31 D2-32 74 8A 49-70 33 EB 2D-14 3B 66 09	лЧ1π2тKIр3и-¶;f○
00003730:	14 23 52 11-0C 7F 7E 1C-51 1B 8B 65-14 37 46 C1	¶HR-¶о^LQ-¶e¶7F-+
00003740:	80 83 9F 52-50 F4 92 08-14 47 D9 84-12 33 EB C9	аГЯRPITЪ¶C-дз3и¶
00003750:	46 FF 36 91-FB A4 1E 5E-E6 EA D9 6A-32 2E 36 31	F 6C√д^цъ^j2.61
00003760:	29 ED DB 36-33 3F 32 FD-15 D4 A5 FA-D3 02 09 CF	)з  63?2и§Le-  ●-+
00003770:	06 D8 0E F5-8F DF 38 A4-73 A3 26 B6-D4 46 3F 30	†+ЫIП¶;аsг&  4F?0
00003780:	53 06 B3 B4-5B 35 D2 0E-62 C8 CF 62-74 47 BA FB	S   5πЪb  btG  √
00003790:	36 CB A1 13-C9 51 75 21-31 31 1F 4F-44 8D E0 3A	6тс!!πQu!11√0DНр:
000037A0:	E8 AA 2C 81-8E C8 8B 43-6B 43 72 01-3B 19 0B 7D	шк,60^лCkCг@;J@}
000037B0:	50 25 04 2E-08 7B 56 BF-FA C9 C0 17-2E D2 7D 63	P%L.Ъ<Uπ-π^z.π>C
000037C0:	AF 89 64 00-00 64 01 00-6C 00 00 5A-00 00 64 00	п  d d@ 1 2 d
000037D0:	00 64 01 00-6C 01 00 5A-01 00 64 00-00 64 01 00	d@ 1@ Z@ d d@
000037E0:	6C 02 00 5A-02 00 64 02-00 5A 03 00-64 03 00 5A	1@ Z@ d@ Z@ d@ Z
000037F0:	04 00 78 49-00 65 05 00-65 01 00 6A-06 00 83 00	♦ xI e♦ e@ j♦ Γ
00003800:	00 6A 07 00-6A 08 00 64-08 00 64 0A-00 21 83 01	j- j@ d@ d@ !Γ@
00003810:	00 44 5D 28-00 5A 09 00-65 04 00 65-0A 00 65 09	D]( Z@ e♦ e@ e@
00003820:	00 65 03 00-64 07 00 16-41 83 01 00-37 5A 04 00	e♥ d- -аГ@ 7Z♦
00003830:	65 03 00 64-05 00 37 5A-03 00 71 F4-37 57 65 02	e♥ d♦ 7Z♥ qI7We●
00003840:	00 6A 0B 00-65 00 00 6A-0C 00 65 04-00 83 01 00	j@ e j@ e♦ Γ@
00003850:	83 01 00 64-01 00 04 55-64 01 00 53-28 00 00 00	Γ@ d@ ♦Ud@ S(
00003860:	00 69 FF FF-FF FF 4E 69-00 00 00 00-74 00 00 00	i Ni t

1 – нужный offset и начало (байт 0x64) корректной инструкции. А предыдущий байт 0x89 как раз обозначает ту самую некорректную в текущем контексте инструкцию STORE\_DEREF, которая занимает 3 байта (0x89 0x64 0x00) – то есть

произошло наложение инструкций. Для исправления достаточно затереть байт 0x89 на 0x09 (эквивалентно инструкции NOP).

000036E0:	A6 07 14 25-3C E2 84 83-5F FB 34 B6-83 BA 89 AE	ж•%<тПГ_√4  Г  Ио
000036F0:	89 BE BB 12-81 75 5E 39-5D E9 2A 75-45 DA 8B BC	Й-т±Бу^9]ш*UE;л²
00003700:	CF 10 15 FD-8C DE 6A 49-28 4E F5 64-8E 73 7C F3	↳\$*H  jI(Nīd0s e
00003710:	8C 89 3C FC-FE 67 C1 C7-3B 67 63 62-78 73 6A 62	ИЙ<Nmg¹  ;gcbxsjb
00003720:	A8 97 31 D2-32 74 8A 49-70 33 EB 2D-14 3B 66 09	л41т2тK1р3м-¶;f○
00003730:	14 23 52 11-0C 7F 7E 1C-51 1B 8B 65-14 37 46 C1	¶8R-00~LQ+ле¶7F¹
00003740:	80 83 9F 52-50 F4 92 08-14 47 D9 84-12 33 EB C9	аГЯRRİTЪ¶G¹д±3мт
00003750:	46 FF 36 91-FB A4 1E 5E-E6 EA D9 6A-32 2E 36 31	F 6C√д▲^цъ¹j2.61
00003760:	29 ED DB 36-33 3F 32 FD-15 D4 A5 FA-D3 02 09 CF	)э  63?2*§¹e-¹¹c¹
00003770:	06 D8 0E F5-8F DF 3B A4-73 A3 26 B6-D4 46 3F 30	▲+БİП;дsг&  ¹F?0
00003780:	53 06 B3 B4-5B 35 D2 0E-62 C8 CF 62-74 47 BA FB	S▲ ¹[5тБb¹¹btG  √
00003790:	36 CB A1 13-C9 51 75 21-31 31 1F 4F-44 8D E0 3A	6т6!!тQu?11√0DHP:
000037A0:	E8 AA 2C 01-8E C8 8B 43-6B 43 72 01-3B 19 0B 7D	шк,60¹лскCr©;↓±}»
000037B0:	50 25 D4 2E-08 7B 56 BF-FA C9 C0 17-2E D2 7D 63	P%¹.Ъ<U¹¹т¹±.т}c
000037C0:	AF 09 64 00-00 64 01 00-6C 00 00 5A-00 00 64 00	nCd d¹¹ 1 2 d
000037D0:	00 64 01 00-6C 01 00 5A-01 00 64 00-00 64 01 00	d¹¹ 1¹¹ 2¹¹ d d¹¹

Примечание:

Узнать байт инструкции, соответствующей её текстовому формату, можно из встроенной библиотеки dis:

```
Python 2.7.18 (default, Mar 8 2021, 13:02:45)
[GCC 9.3.0] on linux2
Type "help", "copyright", "credits" or "license" for
>>> import dis
>>> dis.opmap["RETURN_VALUE"]
83
>>> hex(dis.opmap["RETURN_VALUE"])
'0x53'
>>> hex(dis.opmap["NOP"])
'0x9'
>>> hex(dis.opmap["STORE_DEREF"])
'0x89'
>>> _
```

Пробуем заново дизассемблировать код и смотрим на offset 14244:



10334	14239	<INVALID>	
10335	14240	STORE_FAST	44899 <INVALID>
10336	14243	NOP	
10337	14244	LOAD_CONST	0: -1
10338	14247	LOAD_CONST	1: None
10339	14250	IMPORT_NAME	0: zlib
10340	14253	STORE_NAME	0: zlib
10341	14256	LOAD_CONST	0: -1
10342	14259	LOAD_CONST	1: None
10343	14262	IMPORT_NAME	1: sys
10344	14265	STORE_NAME	1: sys
10345	14268	LOAD_CONST	0: -1
10346	14271	LOAD_CONST	1: None
10347	14274	IMPORT_NAME	2: marshal
10348	14277	STORE_NAME	2: marshal
10349	14280	LOAD_CONST	2: 0
10350	14283	STORE_NAME	3: -
10351	14286	LOAD_CONST	3: -
10352	14289	STORE_NAME	4: res
10353	14292	SETUP_LOOP	73 (to 14368)

Как видно, теперь листинг выглядит правильно.

Проанализировав дальнейшие инструкции байткода видим, что некорректных orcode-ов нет, поэтому имеет смысл скопировать данный фрагмент байткода и собрать новый рус-файл, и декомпилировать.

Стоит отметить, что в данном случае легче будет все-таки затереть верхний кусок (начиная с самого начала, то есть инструкции SETUP\_EXCEPT) некорректного байткода байтами 0x09 (инструкция NOP). Простота обосновывается тем, что не придется изменять header-ры байткода (перед байткодом указывается размер этого кода).

00003680:	09 09 09 09-09 09 09 09-09 09 09 09-09 09 09 09	00000000000000000000000000000000
00003690:	09 09 09 09-09 09 09 09-09 09 09 09-09 09 09 09	00000000000000000000000000000000
000036A0:	09 09 09 09-09 09 09 09-09 09 09 09-09 09 09 09	00000000000000000000000000000000
000036B0:	09 09 09 09-09 09 09 09-09 09 09 09-09 09 09 09	00000000000000000000000000000000
000036C0:	09 09 09 09-09 09 09 09-09 09 09 09-09 09 09 09	00000000000000000000000000000000
000036D0:	09 09 09 09-09 09 09 09-09 09 09 09-09 09 09 09	00000000000000000000000000000000
000036E0:	09 09 09 09-09 09 09 09-09 09 09 09-09 09 09 09	00000000000000000000000000000000
000036F0:	09 09 09 09-09 09 09 09-09 09 09 09-09 09 09 09	00000000000000000000000000000000
00003700:	09 09 09 09-09 09 09 09-09 09 09 09-09 09 09 09	00000000000000000000000000000000
00003710:	09 09 09 09-09 09 09 09-09 09 09 09-09 09 09 09	00000000000000000000000000000000
00003720:	09 09 09 09-09 09 09 09-09 09 09 09-09 09 09 09	00000000000000000000000000000000
00003730:	09 09 09 09-09 09 09 09-09 09 09 09-09 09 09 09	00000000000000000000000000000000
00003740:	09 09 09 09-09 09 09 09-09 09 09 09-09 09 09 09	00000000000000000000000000000000
00003750:	09 09 09 09-09 09 09 09-09 09 09 09-09 09 09 09	00000000000000000000000000000000
00003760:	09 09 09 09-09 09 09 09-09 09 09 09-09 09 09 09	00000000000000000000000000000000
00003770:	09 09 09 09-09 09 09 09-09 09 09 09-09 09 09 09	00000000000000000000000000000000
00003780:	09 09 09 09-09 09 09 09-09 09 09 09-09 09 09 09	00000000000000000000000000000000
00003790:	09 09 09 09-09 09 09 09-09 09 09 09-09 09 09 09	00000000000000000000000000000000
000037A0:	09 09 09 09-09 09 09 09-09 09 09 09-09 09 09 09	00000000000000000000000000000000
000037B0:	09 09 09 09-09 09 09 09-09 09 09 09-09 09 09 09	00000000000000000000000000000000
000037C0:	09 09 64 00-00 64 01 00-6C 00 00 5A-00 00 64 00	00000000000000000000000000000000
000037D0:	00 64 01 00-6C 01 00 5A-01 00 64 00-00 64 01 00	00000000000000000000000000000000
000037E0:	6C 02 00 5A-02 00 64 02-00 5A 03 00-64 03 00 5A	00000000000000000000000000000000
000037F0:	04 00 78 49-00 65 05 00-65 01 00 6A-06 00 83 00	00000000000000000000000000000000
00003800:	00 6A 07 00-6A 08 00 64-08 00 64 0A-00 21 83 01	00000000000000000000000000000000
00003810:	00 44 5D 28-00 5A 09 00-65 04 00 65-0A 00 65 09	00000000000000000000000000000000
00003820:	00 65 03 00-64 07 00 16-41 83 01 00-37 5A 04 00	00000000000000000000000000000000
00003830:	65 03 00 64-05 00 37 5A-03 00 71 F4-37 57 65 02	00000000000000000000000000000000
00003840:	00 6A 0B 00-65 00 00 6A-0C 00 65 04-00 83 01 00	00000000000000000000000000000000
00003850:	83 01 00 64-01 00 04 55-64 01 00 53-28 00 00 00	00000000000000000000000000000000
00003860:	00 69 FF FF-FF FF 4E 69-00 00 00 00-74 00 00 00	00000000000000000000000000000000



При этом запомнили, что данные скорее всего сжаты/зашифрованы, поэтому придется после к ним вернуться.

Полученный пропатченный рус код пробуем декомпилировать. Видим, что декомпиляция прошла успешно и далее можем проанализировать исходный код:

```
> pycdc Shuffle_patched.pyc
# Source Generated with Decompyle++
# File: Shuffle_patched.pyc (Python 2.7)

import zlib
import sys
import marshal
_ = 0
res = ''
for __ in bytearray(sys._getframe().f_code.co_code[4:14244]):
    res += chr(__ ^ _ % 255)
    _ += 1

exec marshal.loads(zlib.decompress(res))
>
```

По итогу понимаем, что те данные, который мы затерли, действительно являются зашифрованными и сжатыми.

Если кратко, то команда `sys._getframe().f_code.co_code[4:14244]` извлекает данные из текущего блока байткода.

После расшифровки данных мы получаем еще один скомпилированный питоновский файл (в дальнейшем будет упоминаться с именем `payload`), только большего размера и без рус header-ра.

Также пробуем декомпилировать полученный файл, предварительно приделав корректный рус-header питона 2.7.

```
> cat pyc2.7header.bin payload > payload.pyc
>
```

```
> pycdc payload.pyc
# Source Generated with Decompyle++
# File: payload.pyc (Python 2.7)

Segmentation fault (core dumped)
>
```

Получаем ошибку при декомпиляции.

Оценим строки полученного файла:

```

Decryption key (in hex) = s
Congrat! Your flag = (
marshalt
chrt
loadst
complet
decrypt1t
slow_printt
raw_input
keyt
check_keyt
strt
decrypt2t
bytearrayt
fromhex
flag(•
<ctf_protected>t
<module>

```

По смыслу строк понимаем, что именно тут происходит вся полезная работа программы.

Пробуем дизассемблировать и, как в предыдущем случае, начинаем анализировать с начала блока байткода:

292		'Congrat! Your flag = '	
293		[Disassembly]	
294	0	JUMP_ABSOLUTE	10939
295	3	CALL_FUNCTION	1
296	6	BINARY_ADD	
297	7	LOAD_NAME	2: chr
298	10	LOAD_CONST	27: 195
299	13	LOAD_CONST	28: 167
300	16	JUMP_ABSOLUTE	513
301	19	<INVALID>	

Сразу происходит прыжок хода выполнения программы на инструкцию с offset-ом 10939. Смотрим:

```

5851      10938  JUMP_IF_FALSE_OR_POP      100
5852      10941  STOP_CODE
5853      10942  LOAD_CONST      1: None
5854      10945  IMPORT_NAME      0: marshal
5855      10948  STORE_NAME      0: marshal
5856      10951  LOAD_CONST      2: '\n#####
#####
#####\n## 8888888b. 8888888b. .d8888b. .d8
R88b d88G R88b d88G R88b d88G R88b ##\n## 888 888 888
888 888 .d88G 888 888 .d88G 888
"R8b. ##\n## 888 T88b 888 888 888 888 d88G"
R88b d88G 888" R88b d88G ##\n## 888 T88b 88888888G"
##\n#####
#####\n#####
5857      10954  JUMP_ABSOLUTE      9335
5858      10957  <INVALID>

```

Как видим, опять произошло наложение инструкций и offset-а 10939 нет.  
Патчим:

```

5848      10935  <INVALID>
5849      10936  NOP
5850      10937  NOP
5851      10938  NOP
5852  1 → 10939  LOAD_CONST      0: -1
5853      10942  LOAD_CONST      1: None
5854      10945  IMPORT_NAME      0: marshal
5855      10948  STORE_NAME      0: marshal
5856      10951  LOAD_CONST      2: '\n#####
#####
#####\n## 8888888b. 8888888b. .d8888b. .d8888b
R88b d88G R88b d88G R88b d88G R88b ##\n## 888 888 888
888 888 .d88G 888 888 .d88G 888 888
"R8b. ##\n## 888 T88b 888 888 888 888 d88G"
R88b d88G 888" R88b d88G ##\n## 888 T88b 88888888G"
##\n#####
#####\n#####
5857  2 → 10954  JUMP_ABSOLUTE      9335
5858      10957  <INVALID>
5859      10958  <INVALID>
5860      10959  LOAD_CONST      89: 76

```

После выполнения кусочка байткода, видим опять прыжок на конкретный offset. После нескольких таких попыток проследовать за ходом выполнения программы понимаем, что по факту изначальный байткод просто разбит на небольшие фрагменты, которые соединены инструкциями JUMP\_ABSOLUTE. С этого момента по факту задача сводится к написанию небольшого скрипта, который соберет данные фрагменты инструкций в один последовательный блок (убрав при этом все JUMP\_ABSOLUTE инструкции). Далее на основе полученного цельного байткода формируем новый рус-файл, который можем успешно декомпилировать (по факту в header-ах рус-кода меняем только значения размера байткода, а остальные части рус-файла – переменные, строки, константы и тп оставляем там же, где они и должны быть – в конце рус-а и никаких изменений туда вносить не нужно)

Полученный рус декомпилируем:

```

1 # Source Generated with Decompyle++
2 # File: chkf.pyc (Python 2.7)
3
4 import marshal
5 s = '\n#####\n#####'
6 _ = chr(163 ^ 192) + chr(213 ^ 213) + chr(140 ^ 140) + chr(209 ^ 209) + chr(255 ^ 255) + chr(130 ^ 130) + chr(83 ^ 83) + chr(
7 _ = marshal.loads(_)
8 exec _
9 _1 = '\xf0\xfa\x82\xe6z\x1f8^\xb4\x94\x16z\xa8\xc7\xf5\x92\xd5\xfd\xc2\xb1dM\xe1\xdb\x91\x9b\xb0\xb9 P\xc4\xb6\x10y\x88\xe6\x
10 _2 = '\xc1\xcb6Ry\x1c\xb8\xdeZz\xe8\x8b\x16~\xb3\xd6\xe7\x84\x08c\xd3\x8c\x1aq\xd9\xbc\x82\xfb\xf7\xdf\xa1\xca\xb8\xddT-\xca\
11 _3 = '\f1\xc2\xa6\x99\xfc\xd9\xbfpp\xc7\xb4\xc0\xacG(\xe1\x968gG7\x84\xf6"Kx\x16\x13gw_\xe8\x9b\xf4\x80\x11c\xe4\xcd\xb6\x8c\x
12 _4 = '\x82\x88\xf4\x90\xf3\x96\xd7\xb1Tt\x98\xfbP8\xc2\xa7C #Hf9x\x1a\x01t\xa2\xcb:V\x05qA(\x12|\xa9\xf6A(\xfa\x94\x14` \xb0\x
13 _5 = '\xc0\xca\x12\x91\xf4\x97\xf1\xc9\xe90T\x86\xe3\xd0\xb3\xd6\xa4\xc8\xb1\xe0\x90\xe2\x96\xe2\xd0\xfdV5\x89\xe0T$\xa2
14 exec compile(decrypt1(_1), chr(74 ^ 65), chr(11 ^ 110) + chr(37 ^ 93) + chr(253 ^ 152) + chr(191 ^ 220))
15 exec compile(decrypt1(_2), chr(115 ^ 40), chr(251 ^ 158) + chr(240 ^ 136) + chr(245 ^ 144) + chr(172 ^ 207))
16 exec compile(decrypt1(_3), chr(73 ^ 188), chr(57 ^ 92) + chr(101 ^ 29) + chr(152 ^ 253) + chr(112 ^ 19))
17 exec compile(decrypt1(_4), chr(227 ^ 224), chr(105 ^ 12) + chr(80 ^ 40) + chr(89 ^ 60) + chr(65 ^ 34))
18 exec compile(decrypt1(_5), chr(149 ^ 143), chr(239 ^ 138) + chr(134 ^ 254) + chr(132 ^ 225) + chr(46 ^ 77))
19 _1 = '1c\xb4\xf0 g\x93\xd0\xf4\xa0\xa5\xe33H:I\x12z\xf0\xa5\xa4\xc2yHU9xK\xc6\x99\x80\xd0\xf6\x8f\x15V\xe8\xd8\xd9\xbd\xa8\x9
20 slow_print(s)
21 print chr(253 ^ 222) + chr(21 ^ 54) + chr(55 ^ 20) + chr(6 ^ 38) + chr(2 ^ 76) + chr(114 ^ 23) + chr(88 ^ 61) + chr(194 ^ 166)
22 key = raw_input('Decryption key (in hex) = ')
23 check_key(str(key))
24 flag = decrypt2(_1, bytearray.fromhex(key))
25 print 'Congrat! Your flag = ', flag
26

```

На первый взгляд выглядит достаточно запутано, но работу облегчает то, что этот код можно выполнять в таком виде в каком он есть и очень просто избавиться от «усложняющих» конструкций

По ходу работы понимает, что зашифрованные строки представляют собой непосредственно исходные коды функций, используемых в программе.

```

3
4 import marshal
5 s = '\n#####\n#####'
6 _ = chr(163 ^ 192) + chr(213 ^ 213) + chr(140 ^ 140) + chr(209 ^ 209) + chr(255 ^ 255) + chr(130 ^ 130) + chr(83 ^ 83) + chr(17 ^ 17) + chr(81 ^ 81)
7 _ = marshal.loads(_)
8 exec _
9 _1 = '\xf0\xfa\x82\xe6z\x1f8^\xb4\x94\x16z\xa8\xc7\xf5\x92\xd5\xfd\xc2\xb1dM\xe1\xdb\x91\x9b\xb0\xb9 P\xc4\xb6\x10y\x88\xe6\x
10 _2 = '\xc1\xcb6Ry\x1c\xb8\xdeZz\xe8\x8b\x16~\xb3\xd6\xe7\x84\x08c\xd3\x8c\x1aq\xd9\xbc\x82\xfb\xf7\xdf\xa1\xca\xb8\xddT-\xca\x
11 _3 = '\f1\xc2\xa6\x99\xfc\xd9\xbfpp\xc7\xb4\xc0\xacG(\xe1\x968gG7\x84\xf6"Kx\x16\x13gw_\xe8\x9b\xf4\x80\x11c\xe4\xcd\xb6\x8c\x
12 _4 = '\x82\x88\xf4\x90\xf3\x96\xd7\xb1Tt\x98\xfbP8\xc2\xa7C #Hf9x\x1a\x01t\xa2\xcb:V\x05qA(\x12|\xa9\xf6A(\xfa\x94\x14` \xb0\x
13 _5 = '\xc0\xca\x12\x91\xf4\x97\xf1\xc9\xe90T\x86\xe3\xd0\xb3\xd6\xa4\xc8\xb1\xe0\x90\xe2\x96\xe2\xd0\xfdV5\x89\xe0T$\xa2\xca\x
14 print(decrypt1(_1))
15 print(decrypt1(_2))
16 print(decrypt1(_3))
17 print(decrypt1(_4))
18 print(decrypt1(_5))
19
20
21
22 exec compile(decrypt1(_1), chr(74 ^ 65), chr(11 ^ 110) + chr(37 ^ 93) + chr(253 ^ 152) + chr(191 ^ 220))
23 exec compile(decrypt1(_2), chr(115 ^ 40), chr(251 ^ 158) + chr(240 ^ 136) + chr(245 ^ 144) + chr(172 ^ 207))
24 exec compile(decrypt1(_3), chr(73 ^ 188), chr(57 ^ 92) + chr(101 ^ 29) + chr(152 ^ 253) + chr(112 ^ 19))
25 exec compile(decrypt1(_4), chr(227 ^ 224), chr(105 ^ 12) + chr(80 ^ 40) + chr(89 ^ 60) + chr(65 ^ 34))
26 exec compile(decrypt1(_5), chr(149 ^ 143), chr(239 ^ 138) + chr(134 ^ 254) + chr(132 ^ 225) + chr(46 ^ 77))
27 _1 = '1c\xb4\xf0 g\x93\xd0\xf4\xa0\xa5\xe33H:I\x12z\xf0\xa5\xa4\xc2yHU9xK\xc6\x99\x80\xd0\xf6\x8f\x15V\xe8\xd8\xd9\xbd\xa8\x9b\x8a\x
28 slow_print(s)
29 print chr(253 ^ 222) + chr(21 ^ 54) + chr(55 ^ 20) + chr(6 ^ 38) + chr(2 ^ 76) + chr(114 ^ 23) + chr(88 ^ 61) + chr(194 ^ 166) + chr(236 ^ 204) +
30 key = raw_input('Decryption key (in hex) = ')
31 check_key(str(key))
32 flag = decrypt2(_1, bytearray.fromhex(key))
33 print 'Congrat! Your flag = ', flag
34

```

Находим нужную функцию decrypt2 (по ходу вывода будет понятно, что она определяется в \_5).

Шифр достаточно простой и из работы функции понимаем, что нам даже необязательно понимать, как он работает, лишь стоит убрать лишнюю проверку ключа (так как очевидно ключ вычисляется из шифртекста, судя из контекста). По итогу получаем значения флага.

**\*\*\*\* Ссылки:**

1. Структура рус файла (версия 2.7) =  
<https://www.sweetscape.com/010editor/repository/files/PYC.bt>
2. Компиляция дебаг версии питона =  
[https://pythonextensionpatterns.readthedocs.io/en/latest/debugging/debug\\_python.html](https://pythonextensionpatterns.readthedocs.io/en/latest/debugging/debug_python.html)
3. Включение дебаг режима в скрипте =  
[https://fedoraproject.org/wiki/Features/DebugPythonStacks#Verify\\_LLTRACE](https://fedoraproject.org/wiki/Features/DebugPythonStacks#Verify_LLTRACE)
4. Информация об инструкциях байткода питона (версия 2.7) =  
<https://docs.python.org/2.7/library/dis.html#python-bytecode-instructions>