Статья Дело о инфостилере Видар - Часть 1 (Распаковка)

xss.is/threads/68543

Привет, в этом посте я буду распаковывать и анализировать инфостилер Vidar из моего выступления на **BSides Islamicabad** 2021. Образец стейджа поставляется в виде файла .xll с расширением файла надстройки Excel. Это позволяет сторонним приложениям добавлять дополнительные функции в Excel с помощью Excel-DNA, инструмента или библиотеки, которые используются для написания надстроек .NET Excel. В этом случае файл xll содержит вредоносный загрузчик dll, который дополнительно сбрасывает упакованный исполняемый файл Vidar infostealer на компьютер-жертву, исследование всей цепочки заражения выходит за рамки этого поста, однако я буду глубоко копать исполняемый файл (Packed Vidar) в части 1 статьи этого постав блоге, а финальный пайлод во второй части.

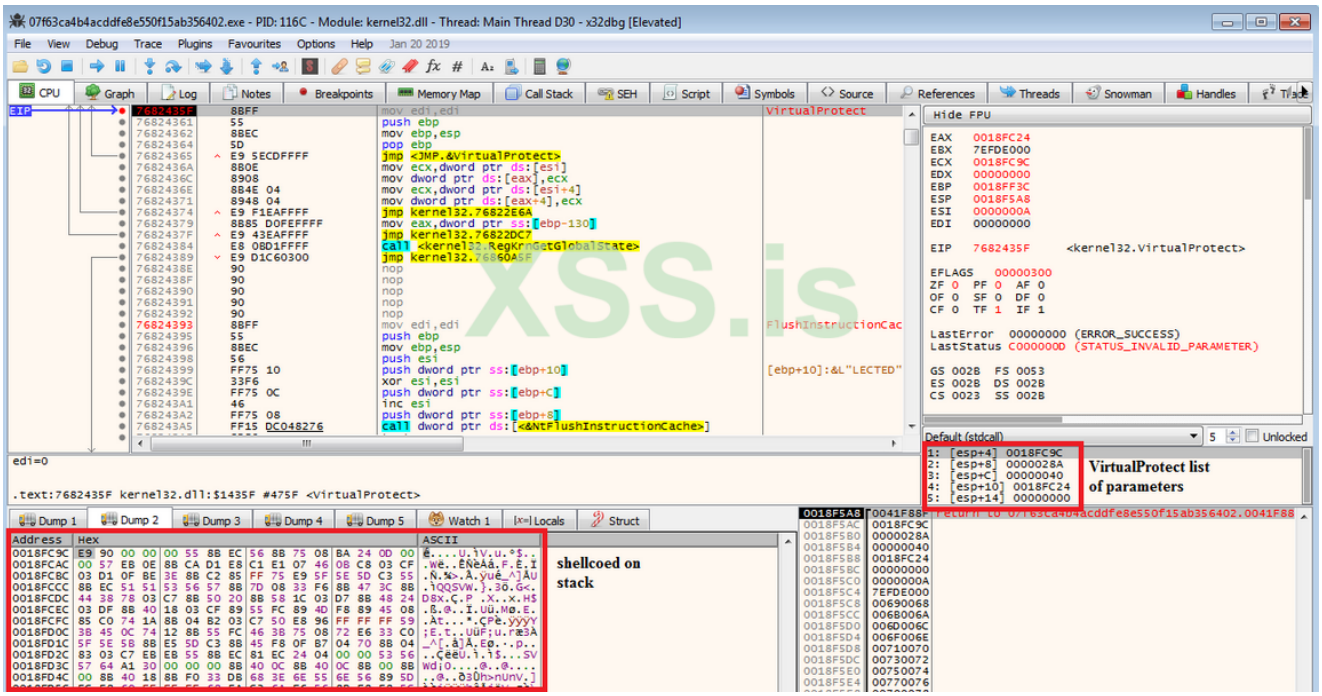
SHA256:5cd0759c1e566b6e74ef3f29a49a34a08ded2dc44408fccd41b5a9845573a34c

Технический анализ

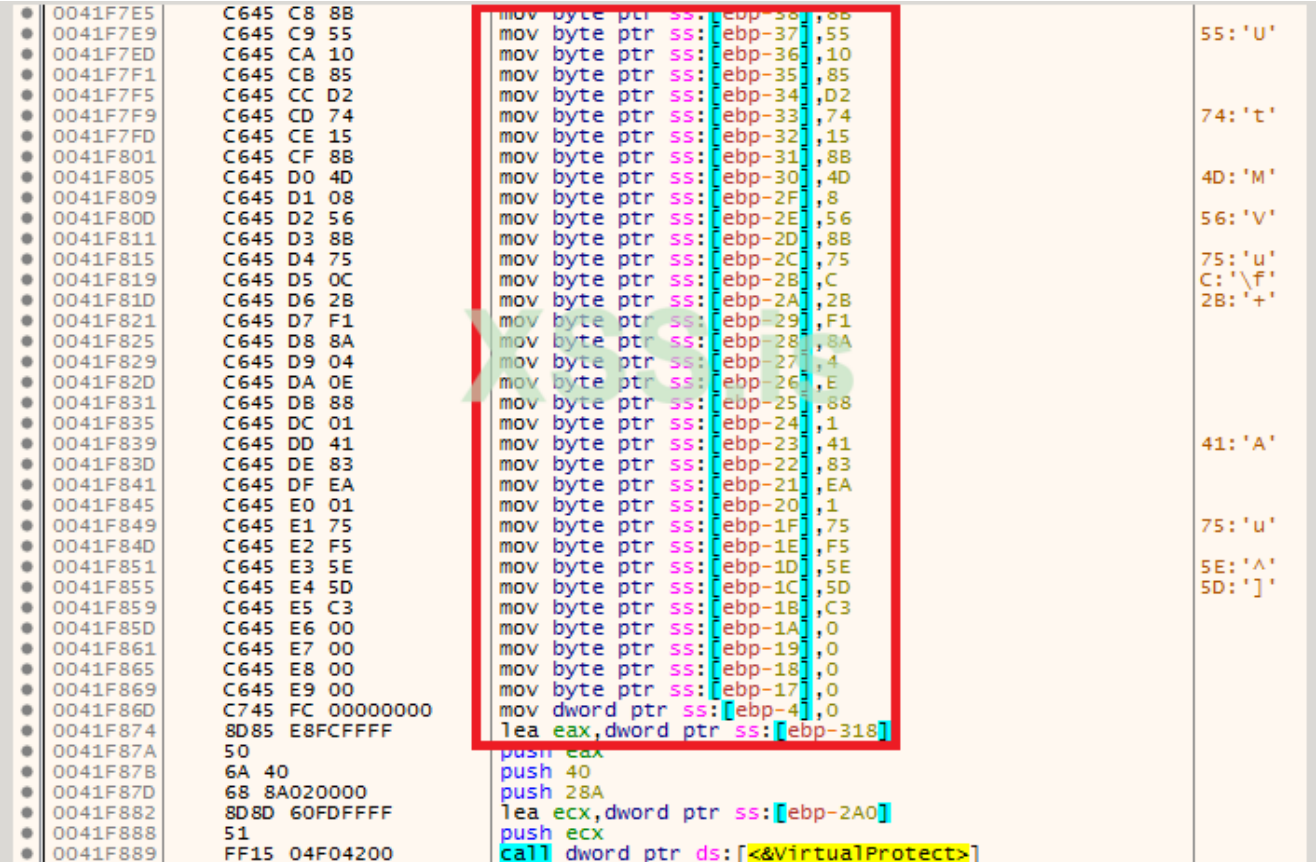
Я обычно начинаю распаковывать общие упаковщики/загрузчики вредоносного ПО, просматривая его сначала в основных инструментах статического анализа, затем открывая его в IDA и рассматривая с высоты различные разделы для переменных с возможными зашифрованными строками, ключами, импортами или другими глобальными переменными, содержащими важную информацию, проверяя, есть ли у него идентифицированные криптографические подписи, а затем начинаю его отладку. После загрузки в x64dbg я сначала ставлю точку останова на API-интерфейсы выделения памяти, такие как LocalAlloc, GlobalAlloc, VirtualAlloc и API защиты памяти: VirtualProtect, и нажимаю кнопку "Выполнить", чтобы увидеть, срабатывает ли какая-либо из точек останова. Если да, то его довольно просто распаковать и извлечь полезную нагрузку следующего этапа, в противном случае может потребоваться углубленный статический и динамический анализ. Давайте нажмем кнопку запуска, чтобы увидеть, куда это нас приведет дальше.

Извлечение шелл-кода

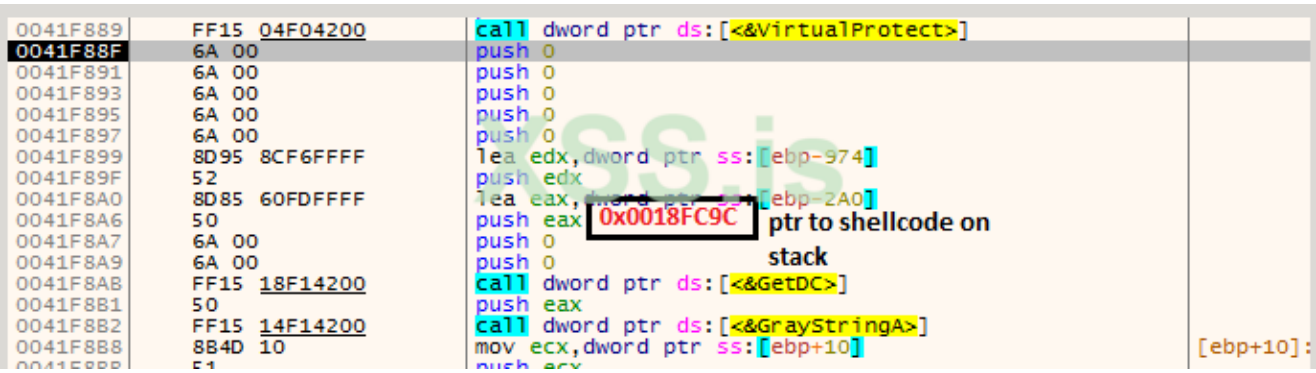
Итак, первая точка останова в этом случае — это **VirtualProtect**, вызываемый в области памяти **стека** размером **0x28A**, чтобы предоставить ему защиту **E**xecute **R**ead **W**rite (0x40), как ни странно, верно!



Первые несколько опкодов **E9, 55, 8B** в выгруженных данных в стеке соответствуют инструкциям **jmp**, **push** и **mov** соответственно, поэтому можно предположить, что это шелл-код, помещаемый в стек, а затем предоставляющий защиту от выполнения для последующего его выполнения. Если я нажму выполнить до кнопку возврата на VirtualProtect и проследить обратно в дизассемблер, я вижу шелл-код, хранящийся в виде **строк стека** прямо перед вызовом VirtualProtect, и список аргументов помещается, как показано на рисунке ниже.



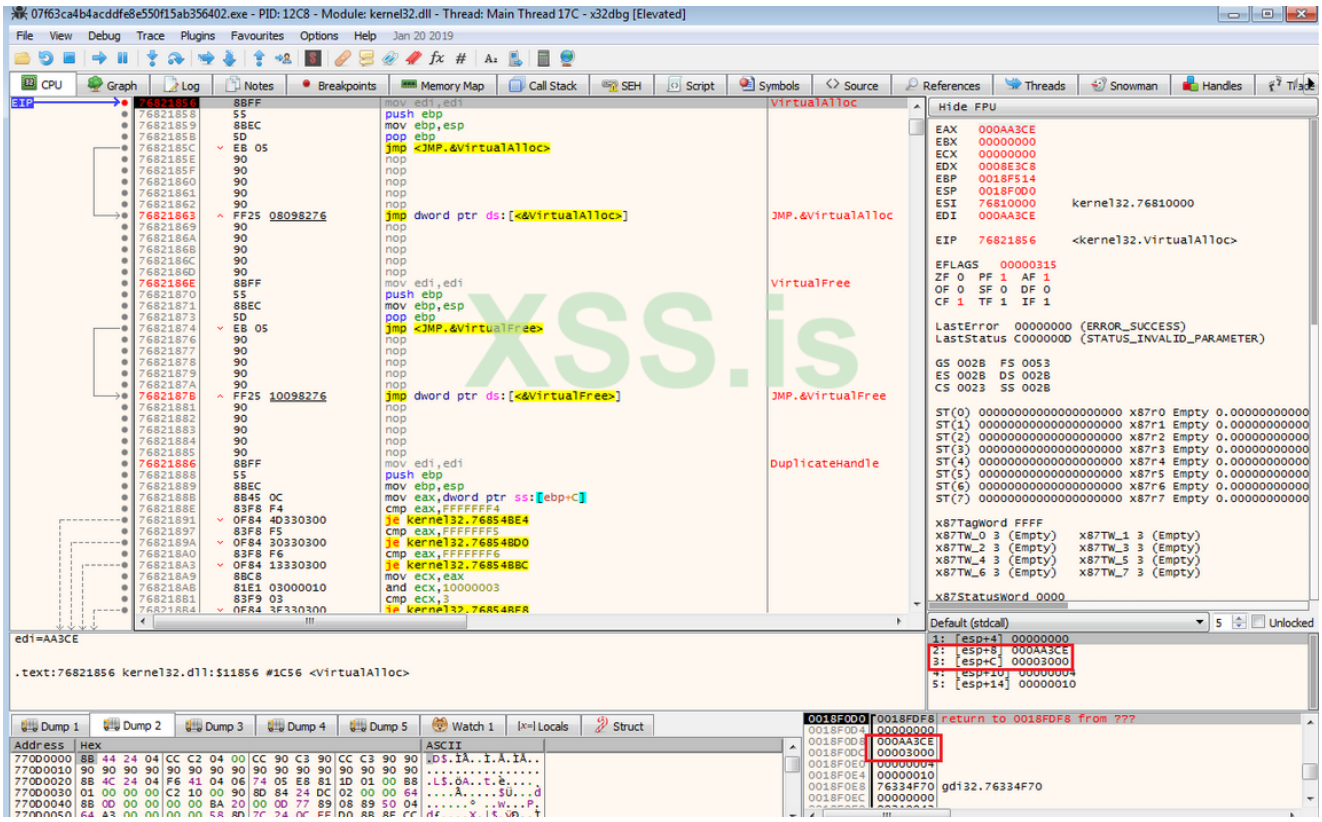
Следующие несколько операторов готовятся к выполнению шелл-кода в стеке, получая дескриптор объекта контекста устройства (DC) и передавая этот дескриптор в GrayStringA для выполнения шелл-кода из стека (значение ptr в eax взято из рисунка 1)



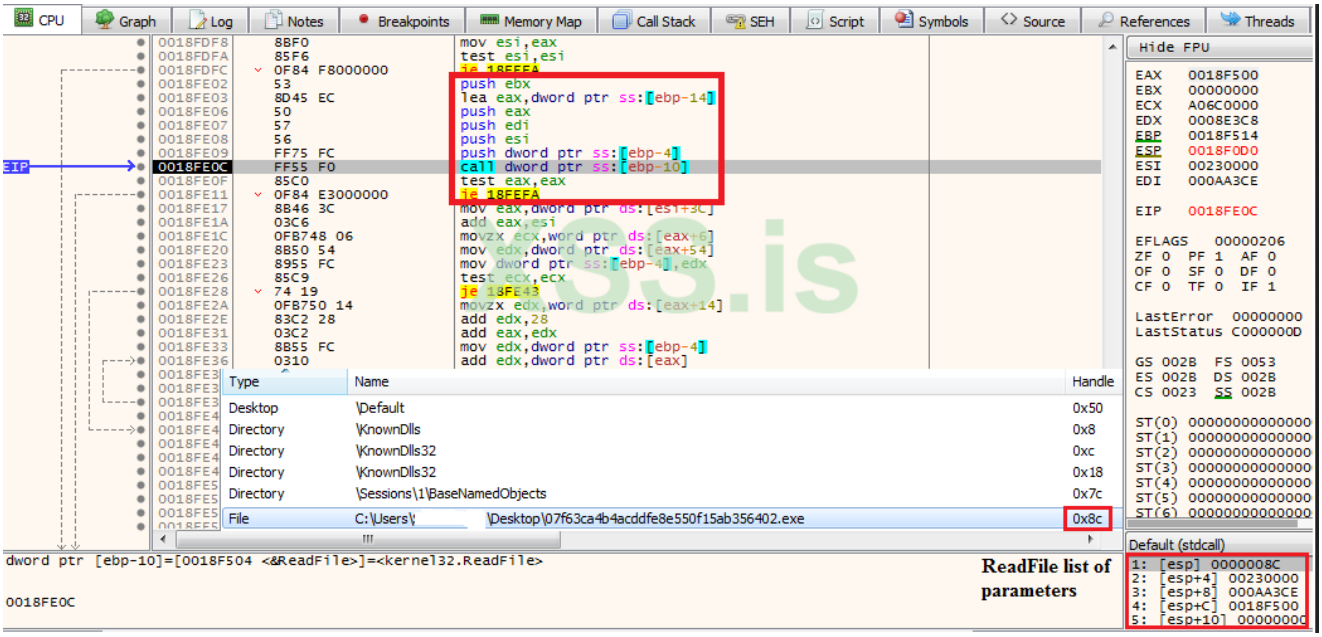
Давайте теперь приступим к изучению шеллкода.

Отладка шелл-кода для извлечения конечной полезной нагрузки

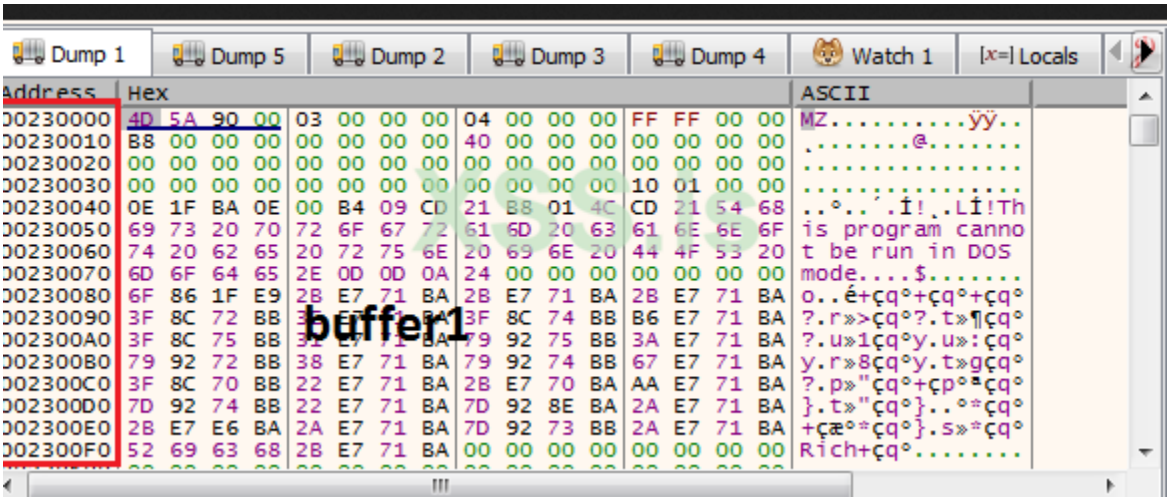
Как только **GrayStringA** выполняется, он попадает в точку останова **VirtualAlloc**, установленную в отладчике, который вызывается для резервирования/фиксации размера памяти 0xAA3CE с **MEM_COMMIT|MEM_RESERVE** (0x3000) тип выделения памяти.



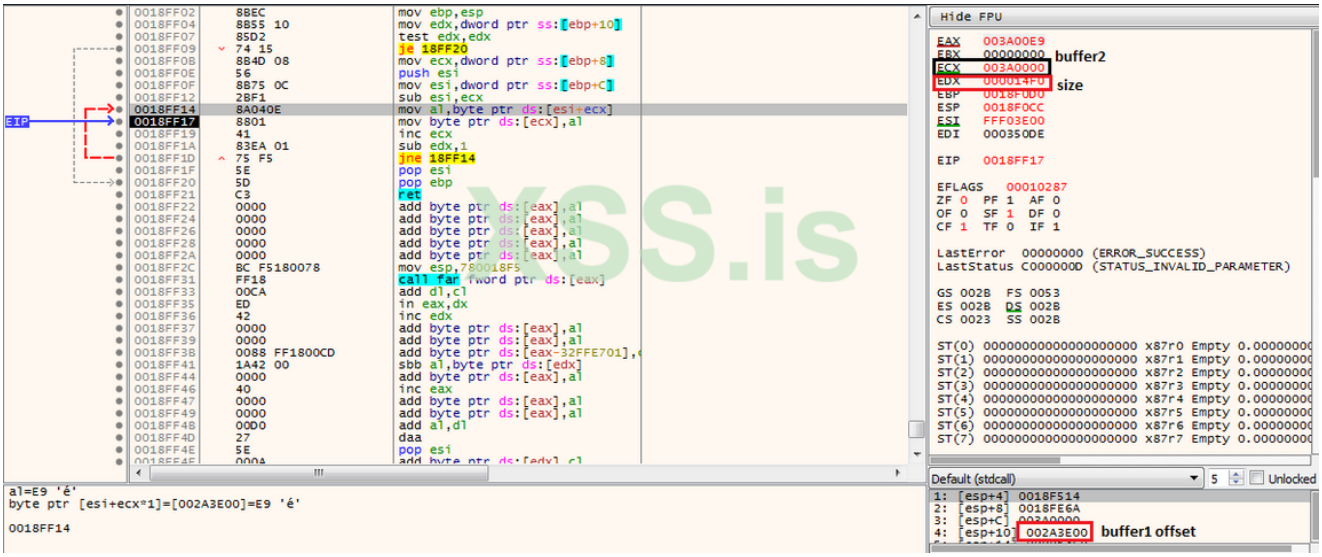
Происходит возврат управления из **VirtualAlloc** и переход еще раз из ret приводит нас к шеллкоду. Следующие несколько операторов после вызова VirtualAlloc передают указатель на вновь созданный буфер, размер буфера и дескриптор файла для текущего загруженного процесса в стеке для вызова **ReadFile**



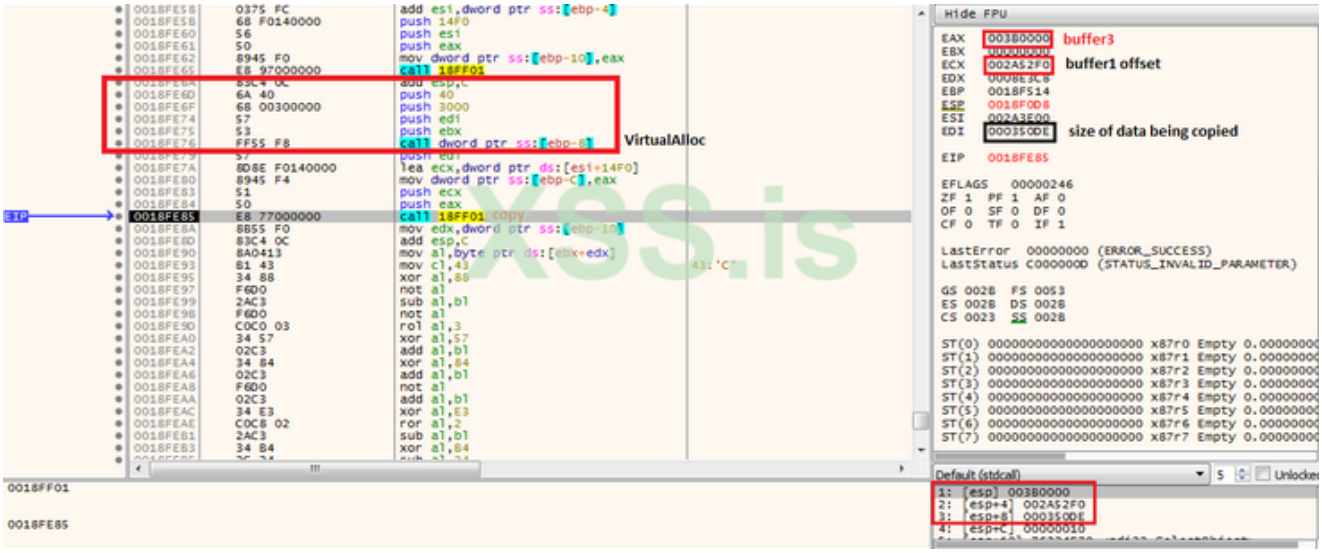
Он считывает oxAA3CE байт данных из образа родительского процесса в буфер, допустим, это **buffer1.**



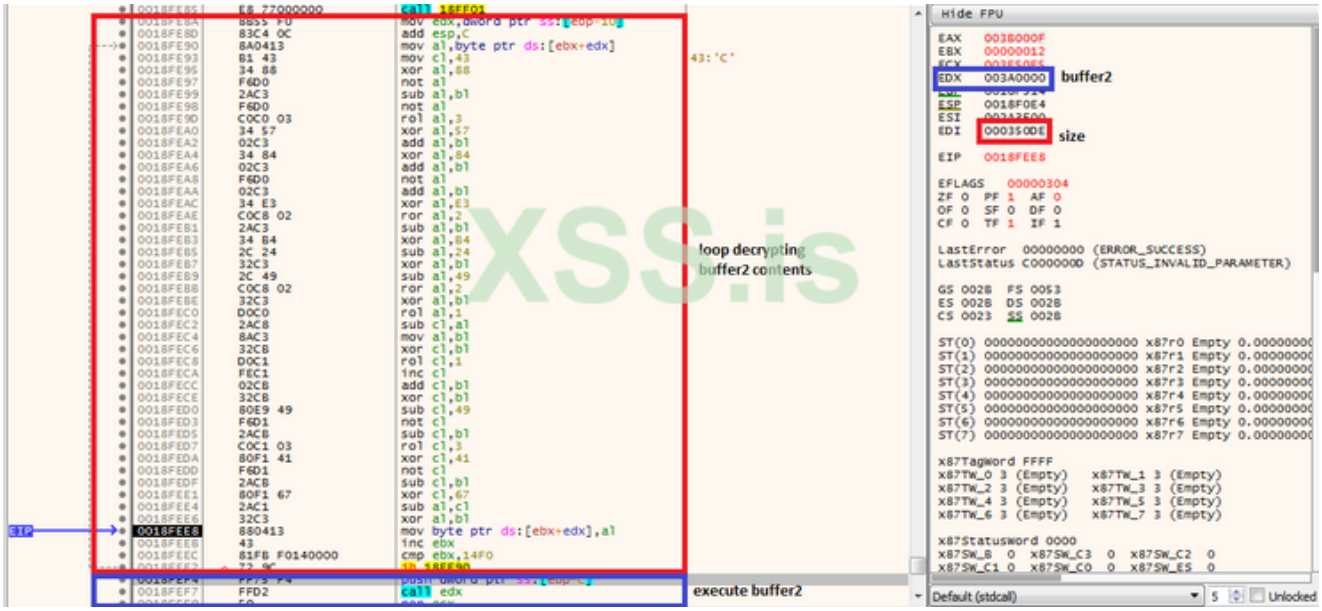
Дальнейшее выполнение снова достигает точки останова **VirtualAlloc**, на этот раз выделяя **0x14F0** байт памяти. Теперь я поставлю точку останова записи в область памяти, зарезервированную/зафиксированную вторым вызовом API VirtualAlloc, чтобы увидеть, какие и как данные сбрасываются во второй буфер, **buffer2**. Повторное нажатие кнопки "Выполнить" прервет выполнение инструкции, показанной на рисунке ниже.



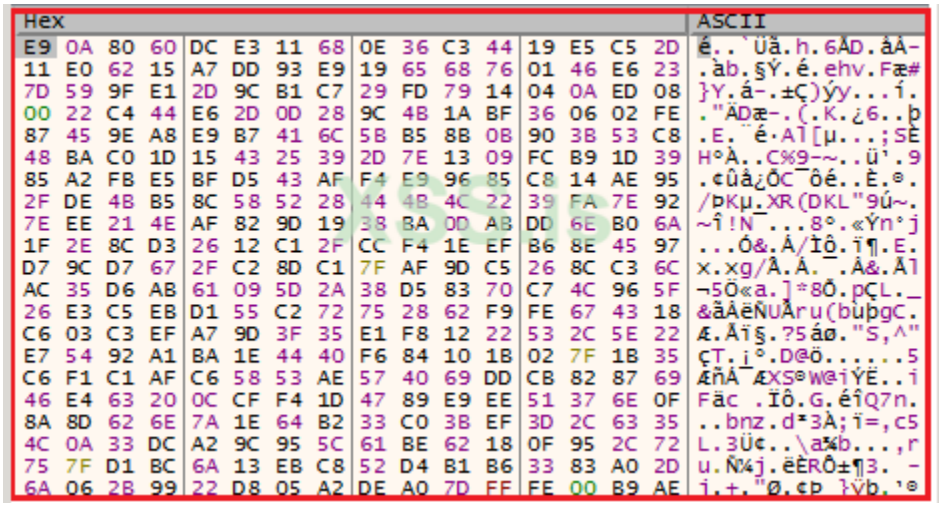
Этот цикл копирует 0x14F0 байтов данных из определенного смещения буфера1 в буфер2, следующие несколько операторов снова вызывают VirtualAlloc для выделения еще 0x350DE байтов памяти, скажем, **буфер3**, помещая возвращенный адрес буфера вместе со смещением из буфера1 в стек для копирования байтов 0x350DE данных из буфера1 в буфер3



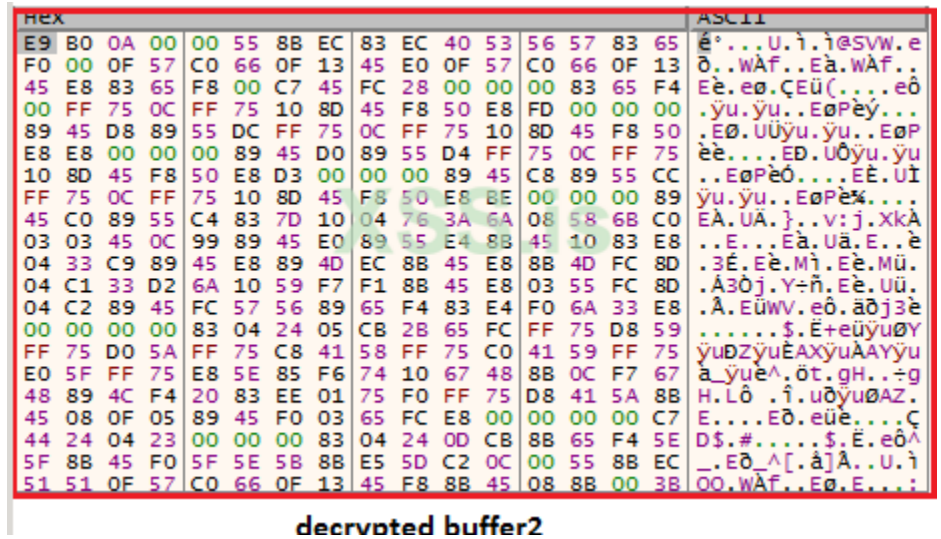
Цикл на следующем рисунке расшифровывает данные, скопированные в буфер2, следующая инструкция push помещает указатель буфера3 в стек в качестве аргумента подпрограммы, вызываемой из адреса буфера2 в edx, которая должна обрабатывать содержимое буфера3



На рисунке ниже показано расшифрованное содержимое окончательного буфера2



Hex																ASCII
E9	0A	80	60	DC	E3	11	68	0E	36	C3	44	19	E5	C5	2D	é..`Üã.h.6ÃD.åÅ-
11	E0	62	15	A7	DD	93	E9	19	65	68	76	01	46	E6	23	.àb.§Ý.é.ehv.Fæ#
7D	59	9F	E1	2D	9C	B1	C7	29	FD	79	14	04	0A	ED	08	}Y.á-.±Ç)ýy...í.
00	22	C4	44	E6	2D	0D	28	9C	4B	1A	BF	36	06	02	FE	."ÄDæ-.(.K.¿6..þ
87	45	9E	A8	E9	B7	41	6C	5B	B5	8B	0B	90	3B	53	C8	.E.¨é·Al[µ...;SÈ
48	BA	C0	1D	15	43	25	39	2D	7E	13	09	FC	B9	1D	39	Hºà..C%9-~..ü'.9
85	A2	FB	E5	BF	D5	43	AF	F4	E9	96	85	C8	14	AE	95	.¢ûå¿ÕC¯ôé..È.®.
2F	DE	4B	B5	8C	58	52	28	44	4B	4C	22	39	FA	7E	92	/ÞKµ.XR(DKL"9ú~.
7E	EE	21	4E	AF	82	9D	19	38	BA	0D	AB	DD	6E	B0	6A	~î!N¯...8º.«Ýn°j
1F	2E	8C	D3	26	12	C1	2F	CC	F4	1E	EF	B6	8E	45	97	...Ó&.Á/Ìô.ï¶.E.
D7	9C	D7	67	2F	C2	8D	C1	7F	AF	9D	C5	26	8C	C3	6C	x.xg/Â.Á.¯.Å&.Ål
AC	35	D6	AB	61	09	5D	2A	38	D5	83	70	C7	4C	96	5F	¬5Ö«a.]*8Õ.pÇL._
26	E3	C5	EB	D1	55	C2	72	75	28	62	F9	FE	67	43	18	&ãÅëÑUÂru(bùþgC.
C6	03	C3	EF	A7	9D	3F	35	E1	F8	12	22	53	2C	5E	22	Æ.Ãï§.?5áø."S,^"
E7	54	92	A1	BA	1E	44	40	F6	84	10	1B	02	7F	1B	35	çT.¡º.D@ö.....5
C6	F1	C1	AF	C6	58	53	AE	57	40	69	DD	CB	82	87	69	ÆñÁ¯ÆXS®W@iÝË..i
46	E4	63	20	0C	CF	F4	1D	47	89	E9	EE	51	37	6E	0F	Fäc .Ïô.G.éîQ7n.
8A	8D	62	6E	7A	1E	64	B2	33	C0	3B	EF	3D	2C	63	35	..bnz.d²3À;ï=,c5
4C	0A	33	DC	A2	9C	95	5C	61	BE	62	18	0F	95	2C	72	L.3Ü¢..\a¾b...,r
75	7F	D1	BC	6A	13	EB	C8	52	D4	B1	B6	33	83	A0	2D	u.Ñ¼j.ëÈRÔ±¶3. -
6A	06	2B	99	22	D8	05	A2	DE	A0	7D	FF	FE	00	B9	AE	j.+."Ø.¢Þ }ÿb.'®

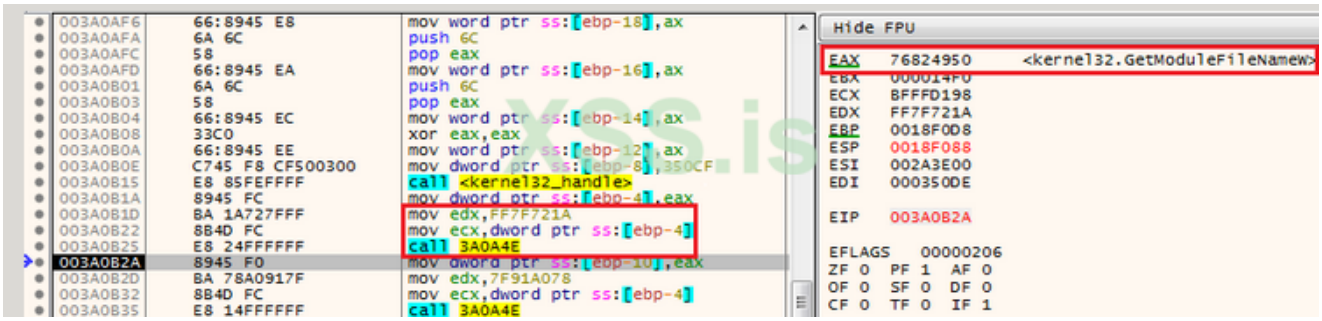


Hex																ASCII
E9	B0	0A	00	00	55	8B	EC	83	EC	40	53	56	57	83	65	é°...U.ì.ì@SVW.e
F0	00	0F	57	C0	66	0F	13	45	E0	0F	57	C0	66	0F	13	ð..WÀf..Eà.WÀf..
45	E8	83	65	F8	00	C7	45	FC	28	00	00	00	83	65	F4	Eè.eø.ÇEü(....eô
00	FF	75	0C	FF	75	10	8D	45	F8	50	E8	FD	00	00	00	.ÿu.ÿu..EøPèý...
89	45	D8	89	55	DC	FF	75	0C	FF	75	10	8D	45	F8	50	.EØ.UÜÿu.ÿu..EøP
E8	E8	00	00	00	89	45	D0	89	55	D4	FF	75	0C	FF	75	èè....EÐ.UÔÿu.ÿu
10	8D	45	F8	50	E8	D3	00	00	00	89	45	C8	89	55	CC	..EøPèÓ....EÈ.UÌ
FF	75	0C	FF	75	10	8D	45	F8	50	E8	BE	00	00	00	89	ÿu.ÿu..EøPè¾....
45	C0	89	55	C4	83	7D	10	04	76	3A	6A	08	58	6B	C0	EÀ.UÄ.}..v:j.XkÀ
03	03	45	0C	99	89	45	E0	89	55	E4	8B	45	10	83	E8	..E...Eà.Uä.E..è
04	33	C9	89	45	E8	89	4D	EC	8B	45	E8	8B	4D	FC	8D	.3É.Eè.Mì.Eè.Mü.
04	C1	33	D2	6A	10	59	F7	F1	8B	45	E8	03	55	FC	8D	.Á3Òj.Y÷ñ.Eè.Uü.
04	C2	89	45	FC	57	56	89	65	F4	83	E4	F0	6A	33	E8	.Â.EüWV.eô.äðj3è
00	00	00	00	83	04	24	05	CB	2B	65	FC	FF	75	D8	59	......$.Ë+eüÿuØY
FF	75	D0	5A	FF	75	C8	41	58	FF	75	C0	41	59	FF	75	ÿuÐZÿuÈAXÿuÀAYÿu
E0	5F	FF	75	E8	5E	85	F6	74	10	67	48	8B	0C	F7	67	à_ÿuè^.öt.gH..÷g
48	89	4C	F4	20	83	EE	01	75	F0	FF	75	D8	41	5A	8B	H.Lô .î.uðÿuØAZ.
45	08	0F	05	89	45	F0	03	65	FC	E8	00	00	00	00	C7	E....Eð.eüè....Ç
44	24	04	23	00	00	00	83	04	24	0D	CB	8B	65	F4	5E	D$.#.....$.Ë.eô^
5F	8B	45	F0	5F	5E	5B	8B	E5	5D	C2	0C	00	55	8B	EC	_.Eð_^[.å]Â..U.ì
51	51	0F	57	C0	66	0F	13	45	F8	8B	45	08	8B	00	3B	QQ.WÀf..Eø.E...;

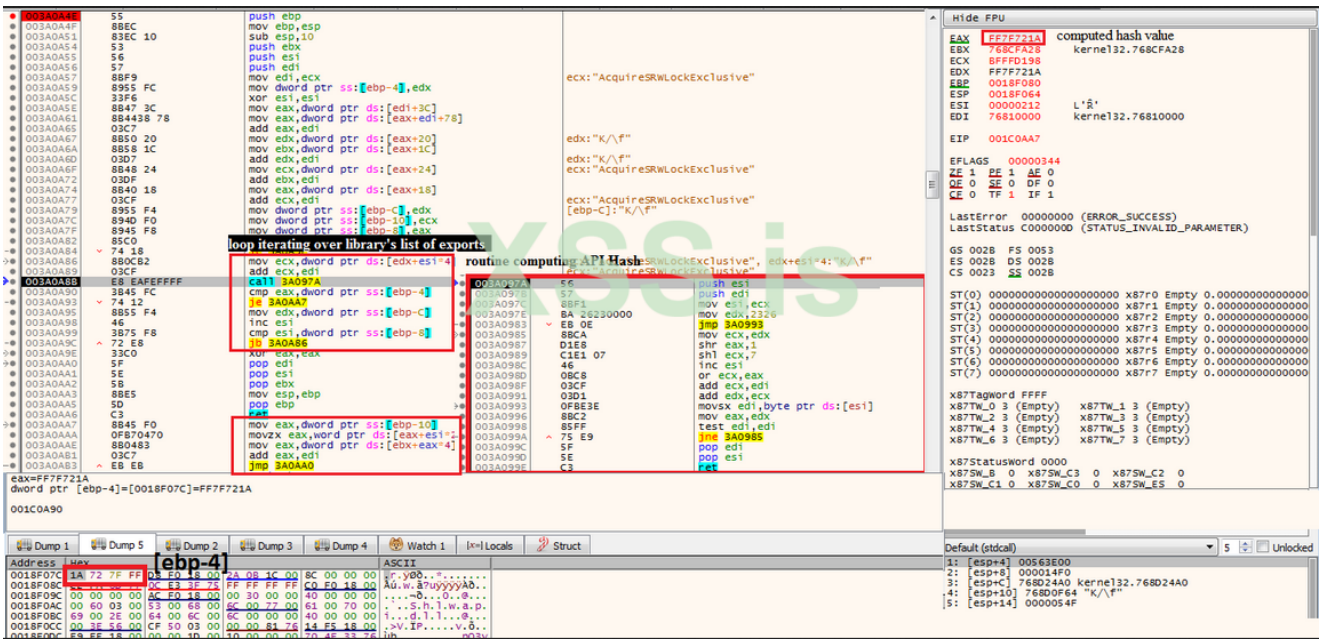
decrypted buffer2

Переход в **edx** запускает выполнение содержимого буфера 2, где, кажется, сначала помещаются строки стека для kernel32.dll, а затем извлекается дескриптор kernel32.dll путем анализа структуры PEB (Process Environment Block)

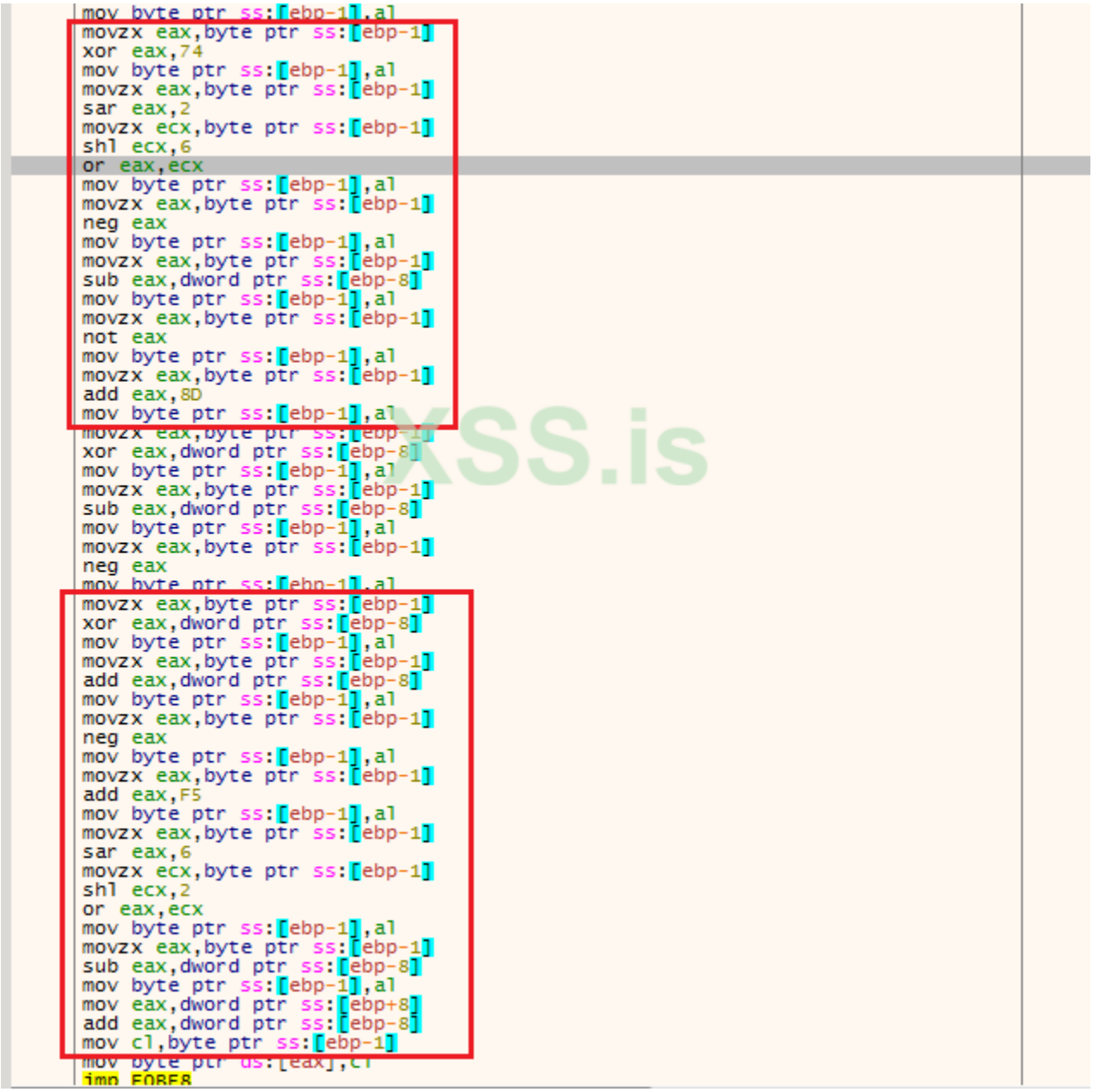
Полученный дескриптор kernel32.dll передается следующему вызову вместе с другим аргументом с постоянным значением **FF7F721A**, быстрый поиск этой константы в Google приводит к некоторым общедоступным ссылкам на песочницу, но неясно, о чем именно идет речь. Давайте углубимся в этот момент, обход этой подпрограммы **0x0A4E** приводит к разрешенному адресу API **GetModuleFileNameW** из Kernel32.dll, хранящемуся в eax, что означает, что эта подпрограмма предназначена для разрешения хешированных API.



Аналогичным образом второй вызов преобразует хеш-значение **7F91A078** в **ExitProcess API.** Подпрограмма-оболочка **0x0A4E** выполняет итерацию по экспорту библиотеки, а подпрограмма **0x097A** вычисляет хэш для входного параметра имени экспорта. Похоже, что шеллкод использует специальный алгоритм для хеширования API, вычисленное хеш-значение перенастраивается обратно в **eax**, которое сравнивается с входным хэш-значением, хранящимся в [ebp-4]. Если оба хэш-значения равны, API разрешается, а его адрес хранится в eax.



Следующие несколько инструкций записывают некоторые ненужные данные в стек, после чего помещают указатель на buffer3 и общий размер содержимого buffer3 (0x350C0) в стек и выполняют подпрограмму **0x0BE9** для дешифрования - эта пользовательская схема дешифрования работает путем обработки каждого байта из буфера3 с использованием повторяющихся отрицаний, вычитания, сложения, sar, shl, not или и xor набор инструкций с жестко закодированными значениями на нескольких уровнях, промежуточный результат сохраняется в [ebp-1]



И окончательное значение перезаписывает соответствующее значение buffer3 по смещению [eax]

encrypted buffer3

buffer3 in processing

Как только содержимое буфера3 расшифровано, он продолжает разрешать другие важные API в следующей подпрограмме **0x0FB6**.

```
mov dword ptr ss:[ebp-C],eax
mov edx,FF7F721A  -> GetModuleFileNameW
mov ecx,dword ptr ss:[ebp-C]
call 230A4E
mov dword ptr ss:[ebp-78],eax
mov edx,7FE2736C  -> CreateProcessW
mov ecx,dword ptr ss:[ebp-C]
call 230A4E
mov dword ptr ss:[ebp-80],eax
mov edx,7FA1F993  -> GetThreadContext
mov ecx,dword ptr ss:[ebp-C]
call 230A4E
mov dword ptr ss:[ebp-84],eax
mov edx,7FA3EF6E  -> ReadProcessMemory
mov ecx,dword ptr ss:[ebp-C]
call 230A4E
mov dword ptr ss:[ebp-88],eax
mov edx,7FE1F1FB  -> CloseHandle
mov ecx,dword ptr ss:[ebp-C]
call 230A4E
mov dword ptr ss:[ebp-1C],eax
mov edx,FF31BF16  -> Wow64SetThreadContext
mov ecx,dword ptr ss:[ebp-C]
call 230A4E
mov dword ptr ss:[ebp-90],eax
mov edx,7FB6C905  -> GetCommandLineW
mov ecx,dword ptr ss:[ebp-C]
call 230A4E
mov dword ptr ss:[ebp-7C],eax
mov edx,7FE7F9C0  -> TerminateProcess
mov ecx,dword ptr ss:[ebp-C]
call 230A4E
mov dword ptr ss:[ebp-94],eax
```

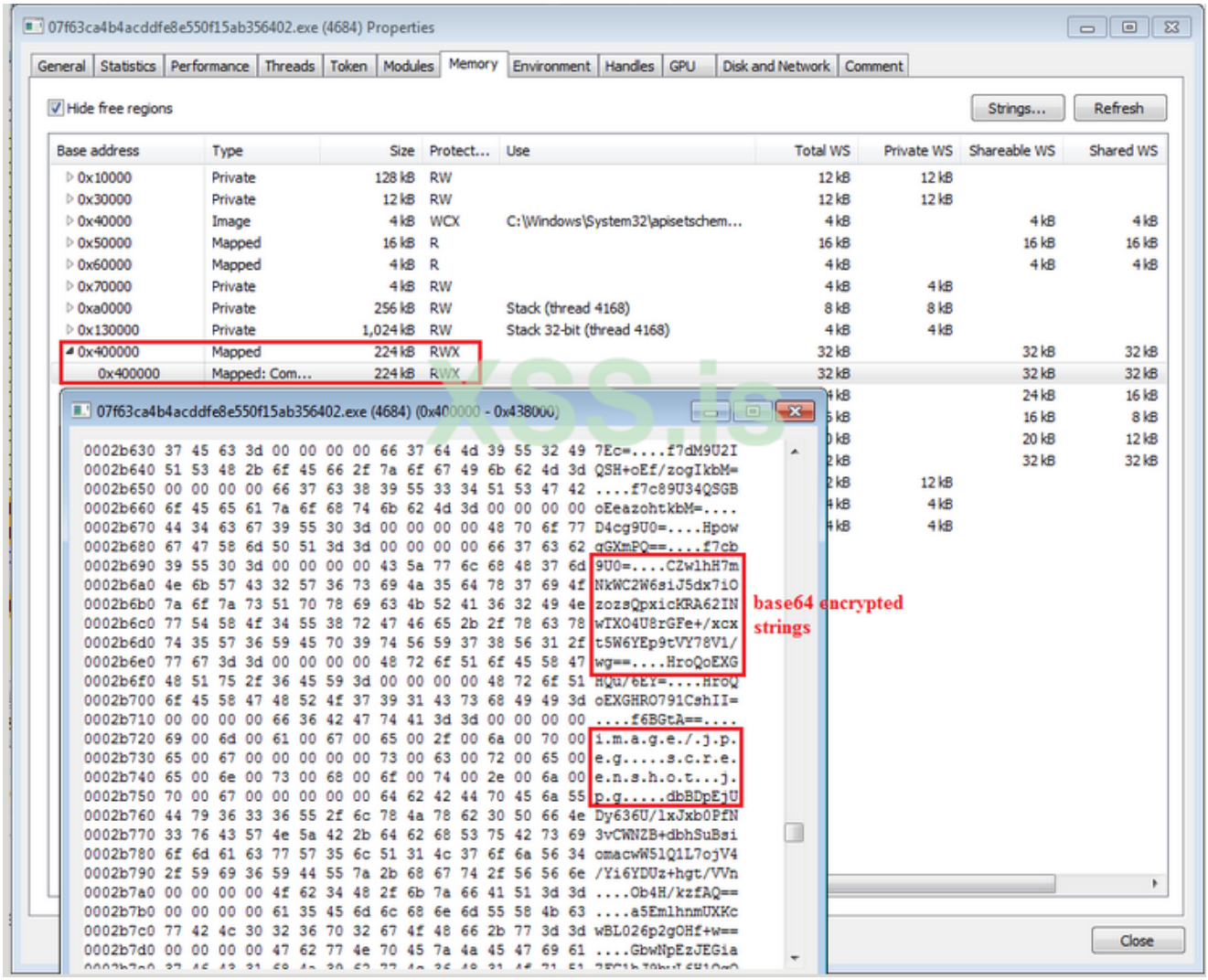
Я написал простой POC-скрипт на Python для алгоритма хеширования, реализованного с помощью расшифрованного шеллкода, который можно найти здесь. https://github.com/0x00-0x7F/RE_tips_and_tricks/blob/master/vidar_packer/api_hash_strings.py

```
In [22]: apis = ["CreateProcessW", "ReadProcessMemory", "GetCommandLineW"]

In [23]: for api in apis:
    ...:     seed = 0x2326
    ...:     for c in api:
    ...:         shr = seed >> 1
    ...:         shl = seed << 7
    ...:         bitwiseor = shr|shl
    ...:         add_char = bitwiseor + ord(c)
    ...:         new_seed = add_char+seed
    ...:         seed = new_seed
    ...:     hash = hex(seed)
    ...:     hash = hash[:-1]
    ...:     hash =  hash[-8:]
    ...:     print hash
    ...:
7fe2736c
7fa3ef6e
7fb6c905
```

После того, как все необходимые API были разрешены, он переходит к созданию нового процесса

А затем окончательная полезная нагрузка вводится во вновь созданный процесс с использованием API SetThreadContext, структура **CONTEXT** для удаленного потока настраивается с помощью ContextFlag и требуемых буферов памяти, а API **SetThreadContext** вызывается с дескриптором текущего потока и структурой CONTEXT удаленного потока для внедрения кода.



Основной процесс завершается сразу после запуска этого процесса, теперь мы можем сделать дамп этого процесса, чтобы извлечь финальную полезную нагрузку.

Вот и все, что нужно для распаковки! Скоро увидимся в следующей статье, посвященном подробному анализу инфостилера Vidar.

Переведено специально для XSS.IS
Автор перевода: yashechka
Источник: https://0x00-0x7f.github.io/A-Case-of-Vidar-Infostealer-Part-1-(-Unpacking-)/