FLARE

Flare-On 10 Challenge 3: mypassion

By Moritz Raabe (@m_r_tz)

Introduction

This challenge requires reverse engineers to recover structures, understand shellcode, deal with inlined functions and optimized code, and keep tabs on execution requirements of a program. This writeup focuses on the key components and does not describe all functionality in detail. The main analysis tools we use are IDA Pro, <u>capa</u>, <u>FLOSS</u>, and <u>CyberChef</u> – all run within <u>FLARE VM</u>.

Basic Analysis

The binary's strings don't provide much insight beyond a few Windows API names and a fun PDB path. Using a PE viewer, we see that this is a 64-bit executable with a sizable .data section. From FLOSS' output we obtain two interesting strings: capture_the_flag and RUECKWAERTSINGENIEURWESEN.

capa identifies multiple interesting capabilities as shown in Figure 1. These include obfuscation and encryption, file system activities, and process manipulation.

+	+
CAPABILITY	NAMESPACE
	+
contain obfuscated stackstrings (5 matches)	anti-analysis/obfuscation/string/stackstring
hash data with CRC32 (2 matches)	data-manipulation/checksum/crc32
encode data using XOR (3 matches)	data-manipulation/encoding/xor
create new key via CryptAcquireContext	data-manipulation/encryption
encrypt or decrypt via WinCrypt	data-manipulation/encryption
encrypt data using AES via WinAPI	data-manipulation/encryption/aes
encrypt data using RC4 PRGA	data-manipulation/encryption/rc4
hash data via WinCrypt	data-manipulation/hashing
initialize hashing via WinCrypt	data-manipulation/hashing
contains PDB path	executable/pe/pdb
contain a resource (.rsrc) section	executable/pe/section/rsrc
query environment variable	host-interaction/environment-variable
set environment variable	host-interaction/environment-variable
get common file path	host-interaction/file-system
enumerate files on Windows	<pre>host-interaction/file-system/files/list</pre>
write file on Windows (3 matches)	host-interaction/file-system/write
shutdown system	host-interaction/os
create process on Windows	host-interaction/process/create
terminate process (2 matches)	host-interaction/process/terminate
link function at runtime on Windows (2 matches)	linking/runtime-linking
parse PE header (3 matches)	load-code/pe
+	+

Figure 1: capa results for the challenge program

Before we explore the file's disassembly, we start the program and observe its run-time activities. Unfortunately, that does not provide any useful results.

Advanced Analysis

We disassemble the file in IDA Pro and quickly see why that is. Spoiler: this program has multiple stages and we'll cover them one by one here.

Stage 1

The program expects a command-line argument and performs various checks on it. We also see the program initializes various structures on the stack.

The main function moves a shellcode buffer into newly allocated memory and executes it. The protection flags for the memory allocation are read from the user input (the flag value should be 0x40 meaning PAGE_READWRITE_EXECUTE). Furthermore, the shellcode bytes are modified at offset 0x41. The modified byte, like the prior instructions, should be 0x45 instead of 0x43 as seen in the file on disk (see Figure 2).

C6 45 F0	16		mo∨	byte	ptr	<pre>[rbp-10h],</pre>	16h
C6 45 F1	17		mo∨	byte	ptr	[rbp-0Fh],	17h
C6 45 F2	3B		mo∨	byte	ptr	[rbp-0Eh],	3Bh
C6 45 F3	17		mo∨	byte	ptr	[rbp-0Dh],	17h
C6 43 F4		loc_14001EA40:	mo∨	byte	ptr	; D/ [rbx <mark>-0Ch]</mark> ,	ата X 56h

Figure 2: Shellcode before byte modification from command line argument

Analyzing the modified shellcode, we see that it decodes a string and compares it to a substring of the user-provided command line argument. The expected string is brUc3. The result of the comparison is not used directly but becomes relevant later. This applies to various data recovered along the way, so we cover them all as encountered.

The main function sets up another large structure on the stack containing data and various function pointers. This structure is passed to a function call at the end of the main routine. Figure 3 shows part of the recovered structure assignment in the decompiler view.

FLARE

```
strcpy_s(Destination.argv, 0x100ui64, argv[1]);
Destination.field_108 = (__int64)v17;
Destination.GetTickCount = (__int64)GetTickCount;
Destination.GetProcAddress = (__int64)GetProcAddress;
Destination.LoadLibraryW = (__int64)LoadLibraryW;
Destination.GetModuleFileNameW = (__int64)GetModuleFileNameW;
Destination.CreateFileW = (__int64)CreateFileW;
Destination.ReadFile = (__int64)ReadFile;
Destination.WriteFile = (__int64)WriteFile;
Destination.Sleep = (__int64)Sleep;
Destination.ExitProcess = (__int64)ExitProcess;
Destination.VirtualAlloc = ( int64)VirtualAlloc;
Destination.free = (__int64)free;
Destination.GetStdHandle = (__int64)GetStdHandle;
Destination.strtol = (__int64)strtol;
Destination.field_388 = (__int64)sub_140001000;
Destination.field_390 = (__int64)sub_140002F00;
Destination.field_398 = (__int64)sub_140002C00;
Destination.field_3A0 = (__int64)sub_140002D20;
Destination.field_3A8 = (
                            <u>int64</u>)sub_140002DB0;
                            _int64)sub_1400018B0;
Destination.field_3B0 = (
sub_1400013E0((__int64)&Destination);
```

Figure 3: Decompilation of the recovered initial structure

For stage 1 to succeed, a working command line argument is: 0A#P_R@brUc3E

Stage 2

In the stage 2 function the program extracts a substring of the command line argument embedded within slash (/) characters. The first part of the substring is parsed as an integer. The following string is used as the name of a file that's created in the module's directory. The file's data stems from the program's .data section and also contains user provided information.

The program uses the Windows crypto API to initialize an AES key and decrypts an embedded buffer. The program executes the buffer as shellcode. Like before a pointer to the large structure is passed to the function. Figure 4 shows how to decrypt the shellcode buffer using CyberChef.

Recipe		8		Input	+ 🗅 ∋ 🛢 🎟			
AES Decrypt				D7C0A8236343B215969DF91C3E4F48D38021BDE431638D7543797C3A2A16F7877087C63D251D1653844480 220574D5671319CD2A4F5A6E4863928E60B7F01E808BAFABCCDB249B13C5628B967D88FFFF83591550FD28				
_{Key} Ødfe66dd3948566a	Key Ødfe66dd3948566a86f1d42378133c4 HE			315A48E64289DLC5CE129BC0A932F7EED2AEC0025A41312FCFE8048CB3DA22C997A52E7AD83C2B3A 91CB358E0D89E1FE8454087B5BD597BC923CB195ECB8E7B8C0AD432D372087C5C93EF2EA99469D810 c7D8378d				
IV capture_the_f…	UTF8 👻	Mode CBC		,				
Input Hex	Outpu Hex	ıt						
					_			
				Rec 352 = 1	T∎ Raw Bytes ← LF			
				Output	🖬 🗍 🖬 🗆			
				48895c24104889742418574883ec20488b8108010000488bd9c740080300 919003000041b8040000048c744243000000000488bc8488d542430488b 20000008bf8ff93880300003bc77412488b8b080100008b4908ff935803 030000488bceff9368030000488bcbff93b0030000488b5c2438488b7424	0000488b81080100008b5008ff F0ff9380030000488b4c2430ba 0000eb0c69cfe8030000ff9350 404883c4205fc3			

FLARE

Figure 4: CyberChef recipe to AES decrypt data

We load this file into IDA via File – Load File – Additional binary file... and select settings like shown in Figure 5.

<u>L</u> oading segment	0	T	(in paragraphs)			
Loading <u>o</u> ffset	0x690000	T				
<u>F</u> ile offset in bytes	0x0	T				
<u>N</u> umber of bytes	0x0	▼	(0 means maxmimum)			
 ✓ Create segments ✓ Code segment 						

Figure 5: Loading the shellcode into IDA Pro

So far, a working command line argument is: 0A#P_R@brUc3E/0file.bin/

Stage 3

To help IDA disassemble the code properly, we navigate to the newly created segment and edit it via Edit – Segments – Edit Segments as shown in Figure 6.

FLARE

💱 Change se	💱 Change segment attributes 🛛 🕹							
Segment <u>n</u> ame		shellcode_stage3						
Segment cl <u>a</u> ss		CODE	•					
<u>S</u> tart address		0x690000	-					
End address		0x6900A6	•					
Com <u>b</u> ination	Public		<u>C</u> olor DEFAULT					
Alignment	Byte	-						
🖌 Move adj	jacent seg	ments	16-bit	Execute				
Disable a	ddresses		🔵 32-bit	✓ Write				
			O 64-bit	✓ Read				
		OK Ca	ancel Help					

Figure 6: Editing the segment attributes to analyze the code properly

This stage gets the next part between slash characters from the command line argument. To continue executing the code expects a number (base 4) that must be equal to the length of the string following it. Before the shellcode calls a function configured via the large context structure it sleeps for the parsed number in seconds.

So far, a working command line argument is: 0A#P_R@brUc3E/0file.bin/1A/

Stage 4

In this function the fourth part of the command line argument is compared against characters from a substructure of the context. Starting at the second character the string must be pizza.

The function then reads the file that the program created in stage 2. Recovering the fields from the stage 4 and stage 2 functions, we obtain a structure definition like shown in Figure 7.

FLARE

000 000 000	file_struc	str	ruc		sizeof=	8
000 000	Magic	dd	?			
004	XorKey	dd	?			
800	XorEncodedFilena	ame	db	?		
800						
009	field_9	db	39	dup	(?)	
030	Byte0x1F	dd	?			
034	TickCount	dd	?			
034						
038	field_38	dd	?			
03C	field_3C	db	4 c	lup(?)	
040	Systemtime	SYS	STEN	1TIM	IE ?	
040						
050	Data	db	174	135	dup(?)	

Figure 7: Recovered structure

The magic value must be 0x11333377. The XOR decoded expected filename is pr.ost. To pass these checks, we update the command line to: $0A#P_R@brUc3E/1337pr.ost/$

Before writing the structure in stage 2 the function modifies part of the data. The modification adds a byte based on the current day of the month plus a hardcoded offset 0xF1. In stage 4 the program bytewise subtracts the first character from the command line part from the buffer. The respective command line character must therefore fit the following formula: $<day> + 0xF1 = <day_character>$, so e.g., for the 20th, 20 + 0x1F = 0x33, which is the character 3.

The TickCount field is furthermore used to verify that more than 8 seconds have passed between writing the file and reading it. So, the Sleep call argument in stage 3 needs to be adjusted accordingly. The modified data is then written to newly allocated memory and a pointer to the data is stored in the large context structure.

One way to pass the checks is to update the command line to: 0A#P_R@brUc3E/1337pr.ost/20AAAAAAA/<day_character>pizza/

Stage 5

Using the debugger, we see that the function called first allocates a shellcode buffer that obtains the fifth command line argument part. Characters from the command line argument are then used to complete a shellcode template.

We write an IDAPython script or use the debugger to obtain the shellcode template from memory. One way to identify the missing bytes is to load the shellcode into IDA with placeholder values. Some of the disassembly may not make sense, but we can fix that along the way.

Google Cloud FLARE

Figure 8 shows the start of the extracted shellcode with placeholder bytes 0xAA. Inspecting the function, we see resemblance to code that manually resolves API functions like GetProcAddress. We can also deduce this from the way the program calls the shellcode and uses its return value.

48				mov	[rsp+8], rbx
48				mov	[rsp+ <mark>18h],</mark> rdi
48				mov	[rsp+10h], rdx
4C				mov	r8, rcx
48				test	rcx, rcx
74	63			jz	short loc_78007A
B8				mov	eax, 0AAAAh
66	39	01		cmp	[rcx], ax
75				jnz	short loc_78007A
48				movsxd	rax, dword ptr [rcx+3Ch]
81				cmp	dword ptr [rax+rcx], 0AA50h
88					

Figure 8: Start of shellcode with highlighted placeholder opcode bytes

For the shellcode to work, we add the expected values as shown in Figure 9.

48				mov	[rsp+8], rbx
48				mov	[rsp+ <mark>18h],</mark> rdi
48				mov	[rsp+10h], rdx
4C				mov	r8, rcx
48				test	rcx, rcx
74				jz	short loc_78007A
B8	4D			mov	eax, 'ZM'
66	39	01		cmp	[rcx], ax
75				jnz	short loc_78007A
48				movsxd	rax, dword ptr [rcx+3Ch]
81				cmp	dword ptr [rax+rcx], 'EP'
00					

Figure 9: Start of shellcode with corrected opcode bytes

Substituting all required bytes we recover six command-line argument characters. There's a second shellcode block to analyze similarly. From the context and after recovering the shellcode template, we see that the code is supposed to get the address of kernel32.dll. Replacing the missing bytes, we obtain the remaining argument part characters.

Updating the command line, we get: 0A#P_R@brUc3E/1337pr.ost/20AAAAAAA/<day_character>pizza/AMu\$E`0R.fAZe/

Stage 6

The stage 6 function translates the command-line input part using a substitution cipher. The result is expected to match the string FLOSS decoded for us: RUECKWAERTSINGENIEURWESEN. The program stores this string in the context structure and writes it to the console.



The command line now is: 0A#P_R@brUc3E/1337pr.ost/20AAAAAAAA/<day_character>pizza/AMu\$E`0R.fAZe/YPXEKCZXYI GMNOXNMXPYCXGXN

Stage 7

This function ensures the 7th command line argument part matches a string concatenated from prior stages. The expected string is ob5cUr3. The program then calls a function to manipulate the read file data that was written to memory in stage 4. The function resembles the RC4 algorithm, but both key-scheduling and pseudo-random generation algorithms execute twice. What a nice twist since people noted that FLARE On challenges often just use RC4.

In the stage 7 function, Capa successfully recognizes the CRC32 checksum algorithm. The checksum must match a value from the context structure, which stems from the file buffer read in stage 4. The program then AES decrypts another encoded buffer using the SHA256 hash of the command line argument (ob5cUr3) as a key. Figure 10 shows the CyberChef recipe to decrypt the data and shows that the data is again 64-bit shellcode. The shellcode is copied to a newly allocated memory and executed.

Recipe		Input	+	- 🗖 🖯 i	
AES Decrypt	⊘ 11	47661E879DEC4C909BD9D1A04F393 3007FF5DF91B67EA9A3AD43880038	19E6333FDBA012D9ABF0409B7163DD8225F530 FBEA77BC0D0116D27AF211CA9B192520C992A	BD9525EA27C6	EBF6E6 91BB07
Key 4d9332e6a35090fea78dca3d0cd8e7	9 HEX 🔻	8B81576B374195422B46CFE279B7C 05F874375CE960A4503335026F7EE 9645DDF1BA3B6674B1FBD7CAB75DE	B207A0D8E65D2657B93805D249DA9609A6106 5F2A5762558699B9C57EA8BC84A40BCC37E9E 79FE3F493FE35A2C819BCCB3DC2BFC6FF132B;	;7CDD24D94FDB ;D86B74A6A88C 202C937999CB2	9E91BF DFC99A 51E353
V capture_the_f UTF8 ▼ CBC	e	AA62652D1DB049701306F1BEB25B1 F9D23DA061580945B05174530E986 8D84F2ABE65BE85A9AD9F4A34D196	0285C205275CB00C56A0B2144904070D4122FU 34AE12524E4DD35051FAD7F480476631B6F52/ 4B1AA924CB056A3479D04F3D67C4C079D1DB22	/8D580CE3B8D9 \A5CAC3BB3C9D 3544BA9522A43	9426D6 67A56A BEB5D6
Input Output Hex Hex		97EB310B9C4C4A9A51AC91BD18D12 702DB46DED11B2C3CE4CE56F36B27 C64F0045E8D9666E59373F4B0797E	21C85A10CCE6F1A49B306FFD9C3622AE579294 C7D52D8C0F3D131C3428A39B54E9A24613D60/ 1AF8320D71CF22B242233D3E6C7296F761E6A	624B2DD32B30 E7B1ED7DA251	7FD89E 89B0C1 750226
Disassemble x86	⊘ 11	1739FB561942AF89A1B5EBE742A9E 2EB34ADD423EBC74E848E8D4ED04E	7CB10EB58B2DEDD3A79AA2EFE473F8F37F13C9 9C00C0A34C07E0A43E0EA0E0CEED4EEE0B30B	F3188CC7CC4F	D301A0
Bit modeCompatibility64Full x86 ar	rchitecture	Output		80	A []
Code Segment (CS) Offset (IP) 16 Ø		48895C2418 4889742420	MOV QWORD PTR [RSP+18],RBX cr MOV QWORD PTR [RSP+20],RSI cr		
Show instruction hex		55 57 4156 488BEC	PUSH RBPcx PUSH RDIcx PUSH RSIcx MOV RBP,RSPcx		
Show instruction position		4881EC8000000 488B8108010000 488BD9 C7400808000000	SUB RSP,0000000000000000000000000000 MOV RAX,QWORD PTR [RCX+00000108]cr MOV RBX,RCXcr MOV DWORD PTR [RAX+08].00000008cr		

Figure 10: CyberChef recipe to AES decrypt another shellcode buffer

We load the additional binary file and change the respective segments like done above to end up with stage 8 and the command line:

0A#P_R@brUc3E/1337pr.ost/20AAAAAAAA/<day_character>pizza/AMu\$E`0R.fAZe/YPXEKCZXYI GMNOXNMXPYCXGXN/ob5cUr3/

Stage 8

The stage 8 shellcode compares the 8th command line argument part to characters from a structure. The argument is supposed to be fin. The final command line is:

0A#P_R@brUc3E/1337pr.ost/20AAAAAAAA/<day_character>pizza/AMu\$E`0R.fAZe/YPXEKCZXYI GMNOXNMXPYCXGXN/ob5cUr3/fin/

Running the program with the final command line opens an HTML page containing an image (see Figure 11) and a hidden message (not shown here).



Figure 11: Image embedded in the HTML file

Plus, the program writes the challenge flag to the console: b0rn_t0_5truc7_b4by@flare-on.com