FLare-On 11 Challenge 9: Serpentine

December 6, 2024

Contents

1 Overview

We are given a single file, `serpentine.exe`. The executable is just a keychecker that requires the key as first argument.

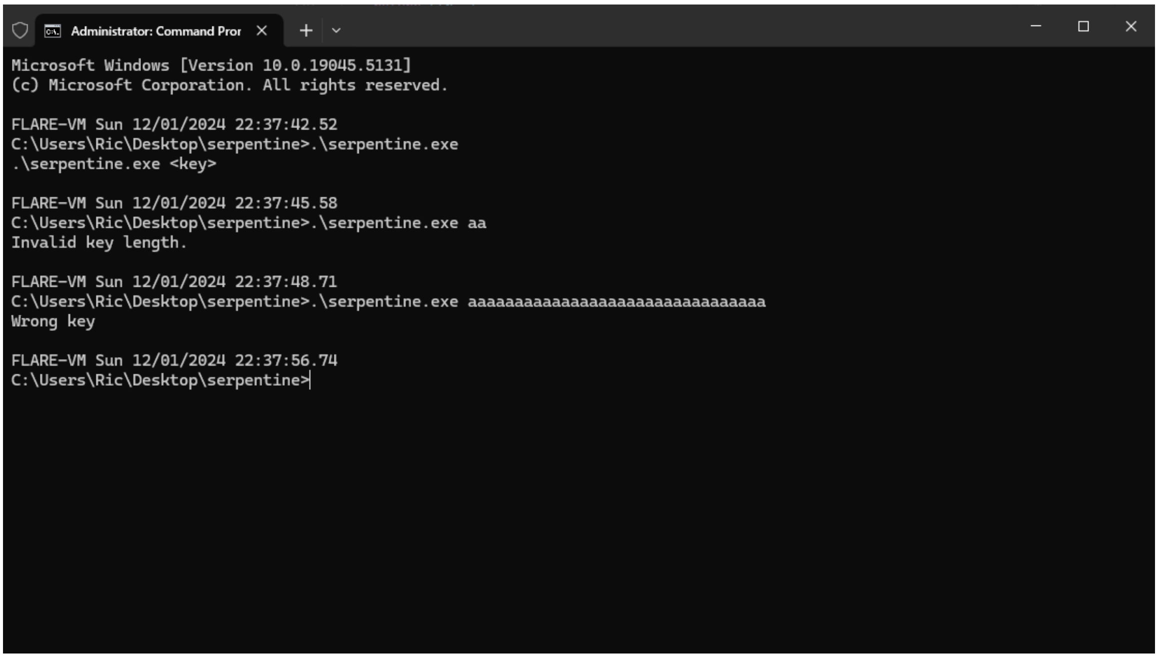


Figure 1: The serpentine.exe executable

2 Starting analysis

Opening the file with Ghidra, it seems that the program is very simple. It just checks the key length and then calls a shellcode with the key as parameter. Looking at the memory of the shellcode, we see that is not initialized at the start of the program. If we checks the references to the shellocde, we can see that it is initialized in the function `tls_callback_0`.

Decompile: main - (serpentine.exe)

```c

undefined8 main(int argc,char **argv)

{
  undefined8 uVar1;
  size_t sVar2;
  char *input;
  
  SetUnhandledExceptionFilter(FUN_00001180);
  if (argc == 2) {
    input = argv[1];
    sVar2 = strlen(input);
    if (sVar2 == 0x20) {
      memcpy(&key,input);
      (*DAT_0089b8e0)(&key);
      uVar1 = 0;
    }
    else {
      FUN_000053e4("Invalid key length.");
      uVar1 = 1;
    }
  }
  else {
    printf("%s <key>\n",*argv);
    uVar1 = 1;
  }
  return uVar1;
}

```

Figure 2: main function

Decompile: tls_callback_0 - (serpentine.exe)

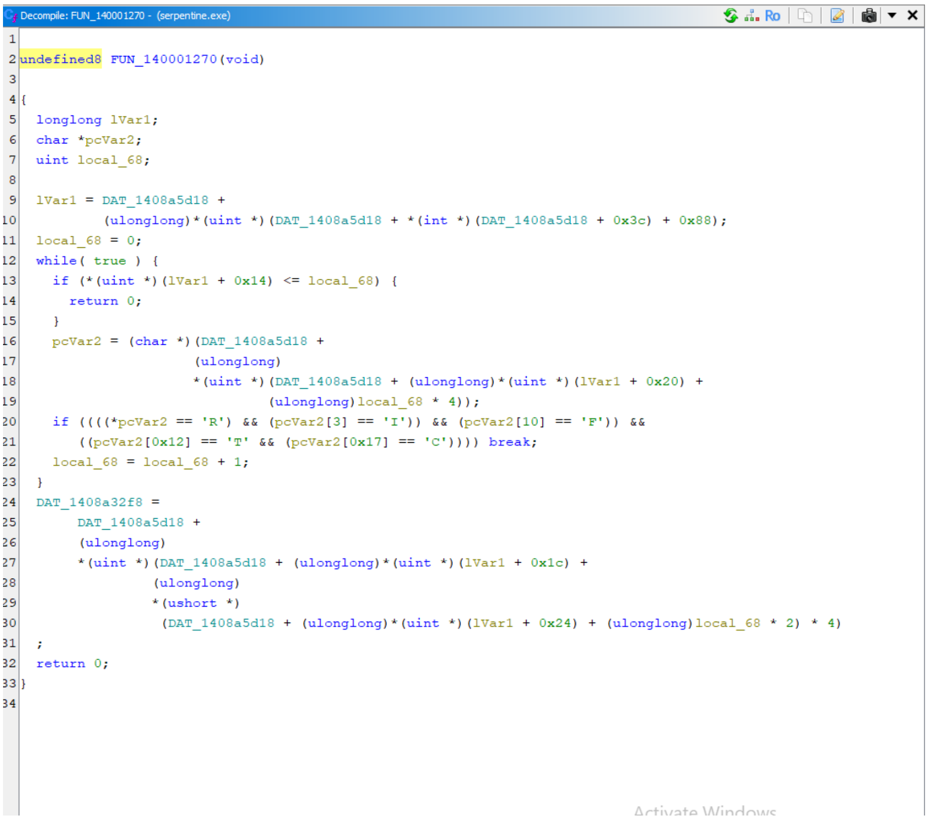
```c

void tls_callback_0(undefined8 param_1,int param_2)

{
  BOOL BVar1;
  
  if (param_2 == 1) {
    DAT_0089b8e0 = VirtualAlloc((LPVOID)0x0,0x800000,0x3000,0x40);
    if (DAT_0089b8e0 == (LPVOID)0x0) {
      FUN_000053e4("Unable to allocate memory.");
      FUN_000050e0(1);
    }
    FUN_000157d0(DAT_0089b8e0,&LAB_00097af0,0x800000);
  }
  else if (param_2 == 0) {
    BVar1 = VirtualFree(DAT_0089b8e0,0,0x8000);
    if (BVar1 == 0) {
      FUN_000053e4("Unable to free memory.");
      FUN_000050e0(1);
    }
  }
  return;
}

```

Figure 3: tls_callback_0 function

We can check the shellcode clicking on the label `LAB_00097af0` in `tls_callback_0`. The first instruction of the shellcode is an `HLT`. The `HLT` instruction is a privileged instruction and can be executed only in kernel mode. so when executed an exception is thrown.

3 Exception handling

Looking again at the main function, we can see that `SetUnhandledExceptionFilter(FUN_140001180)` is called. Unfortunately, if we put a breakpoint in any instruction of `FUN_140001180`, we find out that it is never excecuted. There must be some exception handler that gets called before the execution can reach the unhandled exception filter. Other than using the tls callbacks, another way to hide code and make it execute before the main is through `initterm`. Basically there is an array of function pointers that get called before the execution of the main. The array is handled by the function `FUN_140007ff4` and can be found at this label: `DAT_1400172a0`. We find two interesting function. Function `DAT_140001000` that calls `FUN_140001270` and function `FUN_140001030` that calls `FUN_140001430`. `FUN_140001270` resolves a function name and to store its pointer in `DAT_1408a32f8`, while `FUN_140001430` calls it.



```
Decompile: FUN_140001270 - (serpentine.exe)

undefined8 FUN_140001270(void)

{
  longlong lVar1;
  char *pcVar2;
  uint local_68;

  lVar1 = DAT_1408a5d18 +
          (ulonglong)*(uint *)(DAT_1408a5d18 + *(int *)(DAT_1408a5d18 + 0x3c) + 0x88);
  local_68 = 0;
  while( true ) {
    if (*(uint *)(lVar1 + 0x14) <= local_68) {
      return 0;
    }
    pcVar2 = (char *)(DAT_1408a5d18 +
                     (ulonglong)
                     *(uint *)(DAT_1408a5d18 + (ulonglong)*(uint *)(lVar1 + 0x20) +
                              (ulonglong)local_68 * 4));
    if ((((*pcVar2 == 'R') && (pcVar2[3] == 'I')) && (pcVar2[10] == 'F')) &&
       ((pcVar2[0x12] == 'T' && (pcVar2[0x17] == 'C')))) break;
    local_68 = local_68 + 1;
  }
  DAT_1408a32f8 =
       DAT_1408a5d18 +
       (ulonglong)
       *(uint *)(DAT_1408a5d18 + (ulonglong)*(uint *)(lVar1 + 0x1c) +
                (ulonglong)
                *(ushort *)
                 (DAT_1408a5d18 + (ulonglong)*(uint *)(lVar1 + 0x24) + (ulonglong)local_68 * 2) * 4)
  ;
  return 0;
}
```

Figure 4: FUN_140001270

Decompile: FUN_140001430 - (serpentine.exe)

```c
undefined8 FUN_140001430(void)

{
  if (DAT_1408a32f8 != (code *)0x0) {
    (*DAT_1408a32f8)(DAT_14089b8e0 | 3,DAT_14089b8e0,0x2e4d26,PTR_FUN_1400245b0,0,0);
  }
  return 0;
}
```

Figure 5: FUN_140001430

Using windbg we can put a breakpointer in `FUN_140001430` and find out that the function called is `RtlInstallFunctionTableCallback`.

```
ntdll!LdrpDoDebuggerBreak+0x30:
00007ffc`ee1007a0 cc              int     3
0:000> bp 1400014de
0:000> g
Breakpoint 0 hit
serpentine+0x14de:
00000001`400014de ff542430        call    qword ptr [rsp+30h] ss:00000000`067ffe50={ntdll!RtlInstallFunctionTableCallback (00007ffc`ee0998d0)}
```

Figure 6: RtlInstallFunctionTableCallback gets called

Reading the documentation of `RtlInstallFunctionTableCallback` we learn that it takes a function pointer "to the callback function that is called to retrieve the function table entries for the functions in the specified region of memory." So when an exception is thrown in the specified region of memory, the registered function is called to get the function table entry for that memory region. The function table entry has some information on how to handle the exception. Looking at the parameters of the `RtlInstallFunctionTableCallback` we can see that the memory region is the shellcode, while the registered function is `FUN_1400010b0`. In the documentation we also find out that the registered function has to be of the type: PGET_RUNTIME_FUNCTION_CALLBACK. We can look at it's definition to better analyze it in Ghidra.

```c
PRUNTIME_FUNCTION FUN_1400010b0(DWORD64 ControlPc,PVOID Context)

{
  PRUNTIME_FUNCTION p_Var1;
  uint local_28;

  p_Var1 = (PRUNTIME_FUNCTION)operator_new(0xc);
  p_Var1->BeginAddress = (int)ControlPc - (int)DAT_14089b8e0;
  p_Var1->EndAddress = p_Var1->BeginAddress + 1;
  p_Var1->UnwindData = p_Var1->EndAddress + 1 + (uint)*(byte *)(ControlPc + 1);
  local_28 = (uint)((p_Var1->UnwindData & 1) != 0);
  p_Var1->UnwindData = p_Var1->UnwindData + local_28;
  return p_Var1;
}
```

Figure 7: FUN_1400010b0

The function `FUN_1400010b0` sets the `UnwindData` field of the `PRUNTIME_FUNCTION` struct to an a value taken from an offset from the current instruction (that is the `HLT` instruction that triggers the exception). The offset is stored as a byte in the byte after the `HLT` instruction. The `UnwindData` field contains information on how to unwind the stack and the exception handler to call.

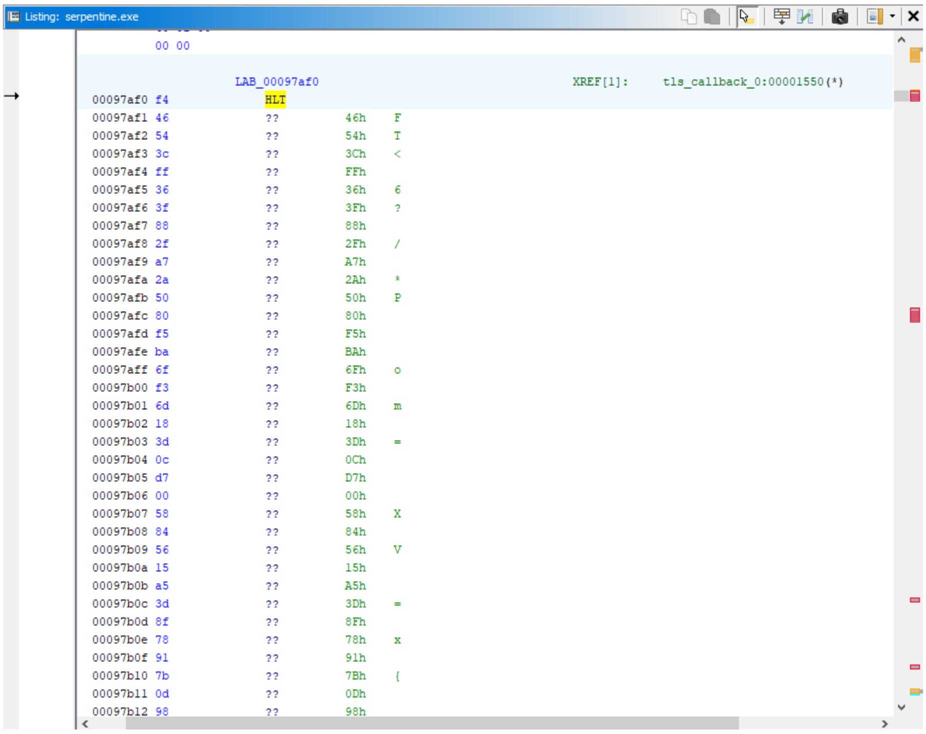


Figure 8: First bytes of the shellcode

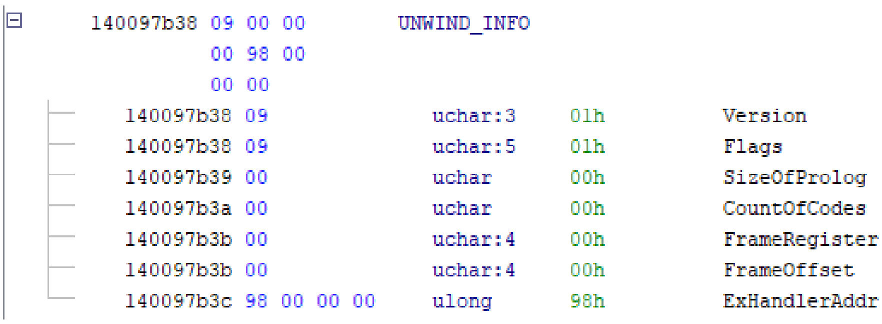


Figure 9: Unwind Info struct

4 Understanding the shellcode

Playing around with the code and the debugger i noticed that the function in charge of calling the exception handler is the function `RtlpExecuteHandlerForException`. In particular at address `RtlpExecuteHandlerForException+0xd` the handler gets called. So putting a breapkpoint there is a good way to see in the debugger what the handler does. Stepping through the instructions of the handler we can see some usual instructions but also some weird call instructions. Every call instruction seems just to decrypt a single instruction, execute it and then encrypt it again. Fortunately for us the Decrypt-Execute-Encrypt scheme is pretty simple. It is just 8 instruction to decrypt the instruction, then the decrypted instruction get executed and then other 5 instruction to encrypt it again.

```
1402e4d27 8f 05 33        POP      qword ptr [LAB_1402e4d5e+2]
          00 00 00
1402e4d2d 50              PUSH     RAX
1402e4d2e 48 c7 c0        MOV      RAX,0x0
          00 00 00 00
1402e4d35 8a 25 eb        MOV      AH,byte ptr [DAT_1402e4d26]        = 4Bh    K
          ff ff ff
1402e4d3b 67 8d 80        LEA      EAX,[EAX + 0x7f497049]
          49 70 49 7f
1402e4d42 89 05 01        MOV      dword ptr [LAB_1402e4d49],EAX
          00 00 00
1402e4d48 58              POP      RAX

                   LAB_1402e4d49                         XREF[2]:     1402e4d42(W), 1402e4d53(W)
1402e4d49 49 bb 49        MOV      R11,0x10add7f49
          7f dd 0a
          01 00 00 00
1402e4d53 c7 05 ec        MOV      dword ptr [LAB_1402e4d49],0x6767...
          ff ff ff
          dd 42 67 67
1402e4d5d 50              PUSH     RAX
                   LAB_1402e4d5e+2                       XREF[0,1]:   1402e4d27(W)
1402e4d5e 48 b8 9d        MOV      RAX,0x11f700009d
          00 00 f7
          11 00 00 00
1402e4d68 48 8d 40 05     LEA      RAX,[RAX + 0x5]=>LAB_1400000a2
1402e4d6c 48 87 04 24     XCHG     qword ptr [RSP],RAX=>LAB_1400000a2
1402e4d70 c3              RET
```

Figure 10: Decrypt-Execute-Encrypt call instruciton

I've written a WinDbg Javascript script (it takes few ours to execute but works) that creates a log file with the trace of all the executed and deobfuscated instructions. For every instruction it also prints the operands and if they are addresses, what do they point to. This trace can be very useful to understand what the shellcode is doing and will be also useful to automatically solve the challenge.

```
##### Deobfuscating exception #2 #####
mov r8,qword ptr [r9+28h] ; 00000000`067fded0, 00000000`067fdd78 => 00000000`067fd800
mov rax,qword ptr [r8+0B0h] ; 00000000`06a402a2, 00000000`067fd8b0 => 00000000`00000045
mov r10,0FFFFFFFFB93774A7h ; 00000000`067fd830, ffffffff`b93774a7
add r10,47B805E5h ; ffffffff`b93774a7, 00000000`47b805e5
push r10 ; 00000000`00ef7a8c
mul rax,qword ptr [rsp] ; 00000000`00000045, 00000000`067fd780 => 00000000`00ef7a8c
mov rbp,rax ; 00000000`067fdd00, 00000000`408c07bc
##### end of deobfuscating exception #2 #####
```

Figure 11: Trace example

We can start the debugging using the key `ABCDEFGHIJKLMNOPQRSTUVWXYZ123456` as an input. This will simplify our analysis since we can look for these ascii characters in the trace to understand where our key gets accessed. Studying the trace we can get what the code does. The code is composed by 32 parts. In each part the algorithm is this: a temp variable is initialized to a character of the key and multiplied by a constant. Then some mangling of the temp variable is done using a mapping array. After this another part of the key is taken, multiplied again for a constant value and merged with the temp variable using an operation (subtraction, addition or xor). This process is repeated for 8 times. Finally there is a last mapping of the temp value and the result it's checked to be 0. Further analyzing the code we can understand that the array mapping are implementing simple operations (again subtraction, addition and xor). So what the code is doing is just cheking the input with a system of 32 equations. Each part of the codes implements an equation and looks like this:

```
temp = key[10] * 0x0048C500
temp -= 0x8FDAA1BC
temp -= key[30] * 0x00152887
temp += 0x65F04E48
temp -= key[14] * 0x00AA4247
temp ^= 0x3D63EC69
temp ^= key[22] * 0x0038D82D
temp ^= 0x872ECA8F
temp ^= key[26] * 0x00F120AC
temp += 0x803DBDCF
temp += key[2] * 0x00254DEF
temp ^= 0xEE380DB3
temp ^= key[18] * 0x009EF3E7
temp -= 0x6DEAA90B
temp += key[6] * 0x0069C573
temp -= 0xC9AC5C5D
temp -= 0xffffffdf3ba3f0d
if temp == 0:
    // check other equation
else:
    // fail
```

Listing 1: Equation Code

5 Solution

Now that we know what the code does, the solution it's straightforward: we just have to write a z3 script that solve the systems. The problem is that manually extracting the equation from the trace generated by the windbg script is a pain. I've spent almost an hour just extracting one equation. So I've written a python script that automatically writes the solver. Since my aim was to just solve the challenge the code is a little convoluted but it is quite simple. The scripts just search for specific patterns inside the trace to understand what operation is being executed, and when the lookup tables are used it gets the correct value in the challenge code. All the details can be found in the code.

```
PS C:\Users\Ric\Desktop\FlareOn11\9-serpentine\repsitory\FlareOn11\9-Serpentine > python .\generateSolver.py
Solution 1: $$_4lway5_k3ep_mov1ng_and_m0ving
Total solutions found: 1
```

Figure 12: Solver code