Static analysis is done using Ghidra v9.1.2.
Dynamic analysis is done using x96dbg

1. Introduction File

First of all let's analyze the 'entry' point. There may be some magic in utility functions which I don't want to miss. Also there I can find the main function.
After renaming here and there the decompilation looks like this:

```
int entry(void) {
  uint uVar1;
  int _Code;
  ___security_init_cookie();
  init_glob1(1);
  uVar1 = init_heap_handle();
  if (uVar1 == 0) {
    _fast_error_exit(0x1c);
  }
  _Code = __mtinit();
  if (_Code == 0) {
    _fast_error_exit(0x10);
  }
  FUN_004036c7();
  _Code = f_util1();
  if (_Code < 0) {
    _fast_error_exit(0x1b);
  }
  ps_CmdLine = GetCommandLineA();
  p_Env = ___crtGetEnvironmentStringsA();
  _Code = __setargv();
  if (_Code < 0) {
    __amsg_exit(8);
  }
  _Code = __setenvp();
  if (_Code < 0) {
    __amsg_exit(9);
  }
  _Code = __cinit(1);
  if (_Code != 0) {
    __amsg_exit(_Code);
  }
  pp_envp2 = pp_envp1;
  _Code = main(i_argc,pp_argv,pp_envp1);
  _exit(_Code);
  __cexit();
  return _Code;
}
```

Nothing too suspicious, except Ghidra did not catch passing arguments to ‘main’ via stack so I added a function prototype for ‘main’.

Let’s switch to ‘main’.

While analyzing decompiled code I stumbled upon 3 anti-debug features:

- IsDebuggerPresent()
- (*(*(in_FS_OFFSET + 0x30) + 2) != '\0') debug flag in PEB
- GetTickCount()

Each of these feature can be patched out by replacing following byte sequences:

- [c0 **74** 08 6a 00] -> [c0 **eb** 90 6a 00]
- [**75 02** eb] -> [**eb 26** eb]

The program is a Windows command-line application, so the functions used to interact with user are WriteConsoleA(), ReadConsoleA().
For obfuscation purposes no plain text is passed to these functions. Instead they are decoded in-place at runtime and encoded back again after use.

2 slightly different algorithms are utilized for obfuscation of text.
The first one XORs each string character with steadily incrementing parameter k.
Here is a python script used to deobfuscate found strings.

```python
from malduck import unhex

def decode_string(struct):
    o = ''
    string = struct[0]
    len = struct[1]
    k = struct[2]
    kd = struct[3]
    b_str = unhex(string)
    for i in range(len):
        o = o + chr(b_str[i] ^ k)
        k = (k + kd) & 0xFF
    return o

str1 = ("75444e4f4251175f53342634693a2e3e3c3c7b2e0017141d02021756435c7f00",
0x1f, 0x25, 0x3)
str2 = ("416f4b45551930263d262b0c181559758600", 0x11, 0x16, 0x7)
str3 =
("3350734e420b475a48546a69023a2c290e024c191317f0bf80ecfcf8ecf890f0928a9cf9f3c1b3
bdbbadbfd9b3d3d3dd616f6d634d297d05170f050b06492b25332507553b4b5d551917e5fbd5a4c5
bd9f87818486b1a3adabbd8fef83c3c3cd919f9d735d324d35273f16181739", 0x6e, 0x12,
0x5)
print("str1: ", decode_string(str1))
print("str2: ", decode_string(str2))
print("str3: ", decode_string(str3))
```

```
str1:  Please enter valid password :
str2:  Wrong password!

str3:  !Good work. Little help:
char[8] = 85
char[0] + char[2] = 128
char[4] - char[7] = -50
char[6] + char[9] = 219
```

Second one uses characters from the password which should be guessed first.

Password guessing.

Let’s analyze this piece of decompiled code:

```
char in_buf [10];
...
ReadConsoleA(h_input,in_buf,10,&num_of_chars_read,0x0);
if ((((in_buf[7] + in_buf[6] == 0xcd) && (in_buf[8] + in_buf[5] == 0xc9)) &&
(in_buf[7] + in_buf[6] + in_buf[3] == 0x13a)) && (((in_buf[9] + in_buf[4] +
in_buf[8] + in_buf[5] == 0x16f && (in_buf[1] + in_buf[0] == 0xc2)) && (in_buf[0]
+ in_buf[1] + in_buf[2] + in_buf[3] + in_buf[4] + in_buf[5] + in_buf[6] +
in_buf[7] + in_buf[8] + in_buf[9] == 0x39b)))) { isCorrect = true;}
else {isCorrect = false;}
if (isCorrect == false) {
    decode_string(str2,0x11,0x16,7); /* Wrong password! */
}
...
```

There is a weak check performed on the incoming password to confirm its validity (password is read into ‘in_buf’ array)
I say “weak” because number of given constraints (6) unfortunately does not define single solution to this system of equations with 10 variables:

```
c[7] + c[6] = 0xcd
c[8] + c[5] = 0xc9
c[7] + c[6] + c[3] = 0x13a
c[9] + c[4] + c[8] + c[5] = 0x16f
c[1] + c[0] = 0xc2
c[0] + c[1] + c[2] + c[3] + c[4] + c[5] + c[6] +
          c[7] + c[8] + c[9] = 0x39b
```

NOTE: decoding “str3” gives us 4 additional constraints which in combination with these 6 does define a single solution

Suppose we don’t have this hint. Let’s continue.
There is a second check on a password. Here is a decompiled code.

```
uint scramble_input(char *c,uint len) {
  uint i, k;
  if (c == 0x0) {k = 0;}
  else {
    k = 0;
    for (i = 0; i < len, i++) {
      k = c[i] ^ (k >> 9 | k << 0x17);
    }
  }
  return k;
}

main() {
    ...
    uVar4 = scramble_input(in_buf,10);
    if (uVar4 == 0x1928f914) {...}
}
```

I wonder if the information from these two checks is enough to try to bruteforce or “guess” a password successfully?
I tried using Z3 SAT solver with all of the constraints (except the hint) and here is a python script for this problem.

```
from z3 import *

def convert2(m):
    o = [m[c0], m[c1], m[c2], m[c3], m[c4], m[c5], m[c6], m[c7], m[c8], m[c9]]
    os = ''
    for i in o:
        os += chr(i.as_long())
    return os

c0, c1, c2, c3, c4, c5, c6, c7, c8, c9 = BitVecs('c0 c1 c2 c3 c4 c5 c6 c7 c8
c9', 8)
k0 ,k1, k2, k3, k4, k5, k6, k7, k8, k9 = BitVecs('k0 k1 k2 k3 k4 k5 k6 k7 k8
k9', 32)
ou = BitVec('ou', 32)
s = Solver()
s.add(SignExt(24, c7) + SignExt(24, c6) == 0xcd)
s.add(SignExt(24, c8) + SignExt(24, c5) == 0xc9)
s.add(SignExt(24, c7) + SignExt(24, c6) + SignExt(24, c3) == 0x13a)
s.add(SignExt(24, c9) + SignExt(24, c4) + SignExt(24, c8) + SignExt(24, c5) ==
0x16f)
s.add(SignExt(24, c1) + SignExt(24, c0) == 0xc2)
s.add(SignExt(24, c0) + SignExt(24, c1) + SignExt(24, c2) + SignExt(24, c3)
    + SignExt(24, c4) + SignExt(24, c5) + SignExt(24, c6) + SignExt(24, c7)
    + SignExt(24, c8) + SignExt(24, c9) == 0x39b)

s.add(k0 == 0)
s.add(k1 == SignExt(24, c0) ^ (RotateRight(k0, 9) | RotateLeft(k0, 0x17)))
s.add(k2 == SignExt(24, c1) ^ (RotateRight(k1, 9) | RotateLeft(k1, 0x17)))
s.add(k3 == SignExt(24, c2) ^ (RotateRight(k2, 9) | RotateLeft(k2, 0x17)))
s.add(k4 == SignExt(24, c3) ^ (RotateRight(k3, 9) | RotateLeft(k3, 0x17)))
```

```
s.add(k5 == SignExt(24, c4) ^ (RotateRight(k4, 9) | RotateLeft(k4, 0x17)))
s.add(k6 == SignExt(24, c5) ^ (RotateRight(k5, 9) | RotateLeft(k5, 0x17)))
s.add(k7 == SignExt(24, c6) ^ (RotateRight(k6, 9) | RotateLeft(k6, 0x17)))
s.add(k8 == SignExt(24, c7) ^ (RotateRight(k7, 9) | RotateLeft(k7, 0x17)))
s.add(k9 == SignExt(24, c8) ^ (RotateRight(k8, 9) | RotateLeft(k8, 0x17)))
s.add(ou == SignExt(24, c9) ^ (RotateRight(k9, 9) | RotateLeft(k9, 0x17)))
s.add(ou == 0x1928f914)

while True:
    if(s.check()==sat):
        m = s.model()
        print(convert2(m))
        # block current solution and solve again:
        block = []
        for d in m:
            c = d()
            block.append(c != m[d])
        s.add(Or(block))
    else:
        break
```

```
Pr0m3theUs
```

Turns out there’s only a single solution which passes both checks.
Now with this password we can break a second obfuscation algorithm:

```
from malduck import unhex

def decode2(struct):
    o = ''
    string = struct[0]
    s_len = struct[1]
    pswd = struct[2]
    pas_len = struct[3]
    k = struct[4]
    b_str = unhex(string)
    for i in range(s_len):
        o = o + chr(b_str[i] ^ ord(pswd[i%pas_len]) + k)
    return o

str4 = ("<first_long_string_of_bytes_from_exe>", 0x100, "Pr0m3theUs", 10, 2)
str5 = ("<second_long_string_of_bytes_from_exe>", 0x100, "Pr0m3theUs", 10, 2)
print(decode2(str4))
```

```
Congratulations! You guessed the right password, but the message you see is
wrong.
Try to look for some unreferenced data, that can be decrypted the same way
as this text..
```

```
https://<next_stage_here>
```

P.S. Somewhere along the way there was this string: "Cr4ckM3"

2. The real Challenge begins

2.1. Step 1. Exe -> Dll -> Exe

Entry function of “EsetCrackme2015.exe” loads “EsetCrackme2015.dll” using LoadLibraryA() into its address space and returns.
Execution flow switches to the entry point of this DLL. Through traversing TEB structure it retrieves an offset of the “main” function (**0x00401e9f**) back in the EXE image and calls it.
(*** I don’t know why switching to DLL happens. My guess is that to finish a process Windows must go through all of the loaded dynamic libraries? And that’s how this dll intercepts execution flow? ***)

TEB traversing algorithm
(TEB structure based on https://www.aldeid.com/wiki/TEB-Thread-Environment-Block):

- TEB->Peb->LoaderData->InMemoryOrderModuleList. This list contains all of the loaded libraries of the current process

```
 p_ModuleList = *(*(*(in_FS_OFFSET + 0x30) + 0xc) + 0x14);
```

- Move through the list and calculate the hash of each BaseDllName entry to find “EsetCrackme2015.dll”

```
hash = 0x811c9dc5;
do {
  c_BaseName = *BaseDllName;
  BaseDllName = BaseDllName + 1;
  if (c_BaseName + 0x9fU < 0x1a) {
    c_BaseName += -0x20;
  }
  hash= (c_BaseName ^ var1) * 0x1000193;
  } while (*BaseDllName != L'\0');
if (hash == 0xfc706866) {
  BaseAddress = p_ModuleList->Reserved2[0];
  ...
}
```

- Search for specific markers in the Process Image to find the correct offset and then call it

```
*(BaseAddress + offset)      == 0xFA121405 &&
*(BaseAddress + offset + 4)  == 0x1FC069DE &&
*(BaseAddress + offset + 8)  == 0xC3325FA1
/* call 0x00401e9f */
(*(BaseAddress + offset + 0xc))();
```

2.2. Step 2. Setting the environment

The function itself at **0x00401e9f** is not very interesting. However it calls **0x40213b**. Let’s call this one a **thread1_main()**. Here’s a decompiled pseudo-code.

```
void thread1_main(void) {
  p_struct_Eset = &base_Esetdll;
  kernel32_base = getBase_kernel32();
  k32_base_hidden = (kernel32_base ^ 0xab12cd99) + 0x54ed3267;
  ...
  f_CreateEvent = _getFun_byhash(0xf9d2dde2);
  p_struct_Eset->h_Event = (*f_CreateEvent)(0,0,0,0);

  f_CreateThread = _getFun_byhash(0x60ac7e39);
  p_struct_Eset->h_thread2 = (*f_CreateThread)(0,0,thread2_main,0,0,0);

  p1_inEsetDll = traverse_Esetdll(0x101);
  p2_inEsetDll = traverse_Esetdll(3);
  if ((p1_inEsetDll != 0x0) && (p2_inEsetDll != 0x0)) {
    decode1((p2_InEsetDll + 3),*(p2_inEsetDll + 1),kernel32_base + 0x4d,0x20);
    decode_complex1((p1_inEsetDll + 3),*(p1_inEsetDll + 1),p_keyStart,p_keyEnd);
    PE_file_loader(s_og,s_og);
    p_NewExec = *(p1_inEsetDll + 0x203);
    while( p_struct_Eset->b_mainThreadIsFin == false)
      {
        do { v1 = (*p_NewExec)(p_struct_Eset);} while ((v1 & 0xffff) != 0);
        if (p_struct_Eset->b_mainThreadIsFin != false) break;
        ...
        f_WaitForSingleObject1 = getFun_byhash(0x71948ca4,k32_base);
        (*f_WaitForSingleObject1)(h_Event, -1);
      }
    }
    p_struct_Eset->b_mainThreadIsFin = true;
    if (h_thread2 != 0x0) {
      f_WaitForSingleObject2 = getFun_byhash(0x71948ca4,k32_base);
      (*f_WaitForSingleObject2)(h_thread2,200);
      f_TerminateThread = getFun_byhash(0x2cf3421c, k32_base);
      (*f_TerminateThread)(h_thread2,-1);
    }
  }
  return;
}
```

There is a creation of a specific structure used in various parts of this crackme. Let’s call it **struct_Eset**. It contains pointers to various Handles and functions and also some endgame data.

Table 1. Structure of struct_Eset

Offset	Length	DataType	Name
0x4	0x4	uint	EsetImageSize
0x8	0x100	WCHAR[128]	s_SomeStringFromPipe
0x108	0x1	bool	b_mainThreadIsFin
0x10b	0x1	short	LastWChar
0x10d	0x4	HANDLE	h_Event
0x111	0x4	void *	f__getFun_byhash
0x115	0x4	void *	f_decode1
0x119	0x4	int *	f_decode_complex
0x11d	0x4	byte *	s_decoded
0x121	0x4	HANDLE	h_Pipe
0x125	0x4	HANDLE	h_thread2

0x129	0x4	uint	k32_base_hidden

The main technique used to obfuscate Win32 API calls is searching for functions by hash. Here's a python script used to do the work for me in this part.

```
def getFun_byhash(hash):
    f = open("G:/crackmes/eset_crackme/kernel32_fun.txt", 'r')
    lines = f.readlines()
    for line in lines:
        res = 0x811c9dc5
        for c in line[:-1]:
            res = ((res ^ ord(c)) * 0x1000193) & 0xFFFFffff
        if(res == hash):
            break
    f.close()
    return line
```

All of the data used in this crackme is stored encoded in the DLL file. Here's a python script to help find an offset of a given marker.

```
def traverse_dll(marker):
    f = open("G:/crackmes/eset_crackme/EsetCrackme2015.dll", 'rb')
    c = f.read()
    index1 = 0
    while ((c[index1+1] << 8) | c[index1] != marker):
        index1 = index1 + ((c[index1+3] << 8) | c[index1+2]) + 6
    f.close()
    return index1
```

Then to decode some data there's this simple function.

```
from malduck import unhex

def decode1(struct):
    o = ''
    string, s_len, pswd, pas_len = struct
    b_str = unhex(string)
    for i in range(s_len):
        o = o + chr(b_str[i] ^ ord(pswd[i % pas_len]))
    return o

# first string extracted from dll at offset +0x1fa+0x6. size = 0x20 (32) bytes
str1 = (b'720c22102977441e220d30110e13320d23042d007a35505a2b47065308396e2b',
0x20, "!This program cannot be run in D", 0x20)
# second string extracted is at offset       +0x2e9+0x6. size = 0x19 (25) bytes
str2 = (b'0c157e1920202020200c0c2320240a222433223d200020202050', 0x19, "PIPE",
0x4)

>>SXJyZW4lMjBpc3QlMjBtZW5zY2hsaWNo
>>\\.\pipe\EsetCrackmePipe
```

And the more complex one that I didn't bother to reverse. I just dumped the decoded memory when debugging and continued static analysis this way. The big blob of data (3584 bytes) that is decoded this way is then called in a while-loop and will be described in the next part.

Here’s a pseudocode of a second thread that’s created here. In short its functionality is as follows:

- create a Named Pipe “\\.\pipe\EsetCrackmePipe”
- receive data from the pipe. This data seems like a 1-byte command + 2-byte message
- depending on a received command perform an action and send an answer.

This is part of the endgame stuff and I’ll return to it later.

```
uint pipe_server(void) {
  p3_inEsetDll = traverse_Esetdll(2);
  ...
  k = "PIPE";
  decode1(p3_inEsetDll + 3,*(p3_inEsetDll + 1),k,4);
  while (!b_mainThreadisFin) {
    f_CreateNamedPipeA = getFun_byhash(0xa215c401,k32_base);
    p_struct_Eset->h_Pipe = (*f_CreateNamedPipeA)(p3_inEsetDll + 3,
                                                  3,0,0xff,0x200,0x200,0,0);
    f_ConnectNamedPipe = getFun_byhash(0x58d5d3e6,k32_base);
    ...
    read_from_pipe(1,&in_buf1);
    read_from_pipe(2,&in_buf2);
    pipe_communicate(in_buf1,in_buf2);
    ...
    f_FlushFileBuffers = getFun_byhash(0xafd195a6,k32_base);
    (*f_FlushFileBuffers)(h_Pipe1);
    f_DisconnectNamedPipe = getFun_byhash(0x8a851d20,k32_base);
    (*f_DisconnectNamedPipe)(h_Pipe2);
    ...
  }
  return;
}
```

For now let’s return to the main thread and see what the decoded function inside the DLL does.

2.3. Step 3. Going deeper.

0x50860cdd is the address of a decoded function of interest. Here’s a decompiled pseudo-code. Let’s call it **stages()**

```
int stages(astruct *p_struct_Eset) {
  if (chk1()) { retVal = stage1_createFiles(); }
  else {
    if (chk2()) { retVal = _FreeAll(); }
    else {
      if (chk3(0xbb01) && chk3(0xbb02) && chk3(0xbb03) && chk3(0xff01)) {
        f_LoadLibraryA = (*p_struct_Eset->f__getFun_byhash)(0x53b2070f);
        h_user32 = (*f_LoadLibraryA)(s_user32.dll);
        f_GetProcAddress = (*p_struct_Eset->f__getFun_byhash)(0xf8f45725);
        p_MessageBoxA = (*f_GetProcAddress)(h_user32,s_MessageBoxA);
        (*p_MessageBoxA)(0,s_Congratulations,s_Info,0x40);
        p_struct_Eset->LastWChar = 0xffff;
        return 1;
      }
      if (p_struct_Eset->LastWChar == 1) {
        f_GetSystemDirectoryA = (*p_struct_Eset->f__getFun_byhash)(0xe129054);
        (*f_GetSystemDirectoryA)(&s_SysDir,0x104);
        ... //form a path to svchost.exe
        lunch_p1_1(&s_SysDir);
        p_struct_Eset->stage = 2;
      }
      if (p_struct_Eset->LastWChar == 0xbb01) {
        lunch_p1_2(s_drv.zip,1,0x152);
        p_struct_Eset = p_struct_Eset;
        p_struct_Eset->stage = 3;
      }
      if (p_struct_Eset->LastWChar == 0xaa10) {
        lunch_p1_2(s_PunchCardReader.exe,1,0x154);
        lunch_p1_2(s_PuncherMachine.exe,1,0x153);
        p_entit = decode_block_fromDll(4);
        ... // form a path to \\?\GLOBALROOT\Device\45736574\PunchCard.bmp
        lunch_p1_2(&s_SysDir,0,0x155);
        p_struct_Eset->stage = 4;
      }
      retVal = 0;
    }
  }
  return retVal;
}
```

Different tasks are being performed during each call. Hence 3 different check() functions.

- When stages() is called the first time it allocates 3 blocks of memory and fills them each with decoded data from DLL.

```
int stage1_createFiles(void) {
  p1_Exec = 0;
  p1_VirtAlloc = decode_block_fromDll(0x102);
  if (p1_VirtAlloc != 0x0) { p1_File = p1_VirtAlloc + 6; }
  p2_VirtAlloc = decode_block_fromDll(0x103);
  p3_VirtAlloc = decode_block_fromDll(0x104);
  p_struct_Eset->stage = 1;
  return 1;
}
```

- Next time stages() is called it should create a Dialog Box with invites to write 3 secret codes. But the procedure by which this Window is created is quite elaborate. It will be described later.
- Manually tempering with memory during debugging can give us means to acquire 3 additional files: drv.zip, PunchCardReader.exe, PuncherMachine.exe.

(*** Haven't reversed this part ***)

- Finally, if all checks are passed successfully this function displays a congratulatory window.

If I understand correctly, data at p1_VirtAlloc is supposedly a PunchCardReader. And p2_VirtAlloc and p3_VirtAlloc contain two different PunchCards.
I think we can see this as an example of a virtual machine with programs written in its own intermediate language. Where p1_VirtAlloc is a VM that performs some utility such as allocating 4kB of memory for (most likely) stack purpose, initializing a translation table (a list of functions that correspond to each "op-code" in a program) and so on.

The functions at **0x50860beb**(p1_launch_1()) and **0x50860b8f**(p1_launch_2()) each perform a VM call as follows: *(p_VM)(PunchCard, Data). Where Data is a blob of data additionally decoded from the Dll file.

2.4. Step 4. Going sideways.

Turns out p1_launch_1() is responsible for creating a DialogBox with 3 invites to write secret codes. It does so this way:

- launch svchost.exe in suspended mode with CreateProcessA()
- inject entirely new executable inside svchost.exe with VirtualAllocEx() and WriteProcessMemory()
- manipulate pointer in memory to call this new executable instead of original code when process is resumed
- ResumeThread() and thus launch an injected code

Here's the decompiled injected pseudo-code:

```
thread1_inj_main(HWND h_wnd,UINT msg,WPARAM par1,LPARAM par2) {
  if (msg == 0x110) {
    ...
    CreateThread(0x0,0,thread2_main,&DAT_00412ef0,0,&msg);
    init_og_hashes();
    return 1;
  }
  if (msg == 0x111) {
    if (par1 == 1) {
      pass_hashing(0x3e9);
      if (lstrcmpA(&og_hash1,&in_hash) == 0) {
        ... // gray out the box #1
        uVar2 = 0xbb01;
      }
    } else {
      if (par1 == 0x3eb) {
        pass_hashing(0x3ea);
        if (lstrcmpA(&og_hash2,&in_hash) == 0) {
        ... // gray out box #2
        uVar2 = 0xbb02;
        }
    } else {
      if (par1 == 0x3ed) {
        pass_hashing(0x3ec);
        if (lstrcmpA(&og_hash3,&in_hash) == 0) {
          ... // gray out box #3
          uVar2 = 0xbb03;
        }
      }
    }
  }
...
pipe_communication(2,uVar2,&msg,5);
return 0;
}
```

Functionality is pretty simple. The first time this function is called (on creating the dialog box) it initializes the environment and gets hashes of secret codes via Named Pipe.
On pressing “check” in a DialogBox it calculates a hash of an input and compares it with the stored one in memory.

Original 3 hashes ripped from “svchost.exe” during debug
"869B39E9 F2DB16F2 A771A3A3 8FF656E0 50BB1882"
"0F30181C F3A98573 60A313DB 95D5A169 BED7CC37"
"0B6A1C66 51D1EB5B D21DF592 1261697A A1593B7E"