Flare-On 7: Challenge 6 – codeit.exe

Challenge Author: Mike Hunhoff (@mehunhoff)

INTRODUCTION

codeit.exe is a compiled AutoIt executable for the Windows operating system. The program lets users generate Quick Response (QR) codes from text submitted to a GUI.

AutoIt is a freeware BASIC-like scripting language designed for general scripting and automating the Windows GUI. An AutoIt script can be executed directly by the AutoIt interpreter or compiled to a stand-alone executable. The current version of AutoIt has a feature rich installation package to help develop and debug AutoIt scripts. There is also a customized a customized version of the AutoIt Script Editor that includes additional coding tools for AutoIt.

Analysis tools including Exe2Aut and AutoIT Extractor can extract the source script from a compiled AutoIt executable. This write-up focuses primarily on analysis of the source script extracted from codeit.exe using Exe2Aut.

Notes on source material: codeit.exe generates QR codes using QR Code generator developed by Nayuki. This library is open sourced under the permissive MIT License and available on GitHub.

INITIAL DYNAMIC ANALYSIS

To get a basic understanding of the program we perform initial dynamic analysis. On startup the program presents us with a simple GUI that allows QR codes to be generated from text. Figure 1 shows screenshots of the program generating a QR code for the text "Hello World!".
To gain a better understanding of how codeit.exe interacts with our system we start Process Monitor and continue interacting with the program. Figure 2 shows a screenshot of the events captured by Process Monitor filtered for the Operation type CreateFile.

We see that codeit.exe repeatedly creates and deletes multiple files in its working directory. We can prevent the program from deleting these files by modifying the permissions of this folder as shown in Figure 3.
codeit.exe can no longer delete files after we modify the permissions of its working directory. This allows us access to these files for analysis but unfortunately a quick look does not reveal anything interesting.

The files created with a .bmp file extension contain the generated QR codes and multiple copies of the default image displayed in the GUI. The files created with a .dll file extension are copies of the same binary and strings found in this binary, listed in Figure 4, indicate it may simply be a supporting library used to generate QR codes.

```
JustConvertQRSymbolToBitmapPixels
justGenerateQRSymbol
qencode.dll
C:\Users\spring\source\repos\qencode\Release\qencode.pdb
```

**Figure 4: Strings found in binary generated by codeit.exe**

**INITIAL STATIC ANALYSIS**

To get a better understanding of the format and code structure of codeit.exe we perform initial static analysis. Opening codeit.exe in CFF Explorer reveals that the program is packed with UPX as shown in Figure 5.
We use the UPX Utility available in CFF Explorer to unpack the program as shown in Figure 6.

Strings found in the unpacked copy of `codeit.exe`, some of which are listed in Figure 7, indicate the program is a compiled AutoIt script.

```python
>>>AUTOIT NO CMDEXECUTE<<<
>>>AUTOIT SCRIPT<<<
```

We use the program Exe2Aut to retrieve the AutoIt script source from the original copy of `codeit.exe` as shown in Figure 8.
Figure 8: Extracting AutoIt script source with Exe2Aut

Exe2Aut unpacks two files named qr_encoder.dll and sprite.bmp. Generating hashes for these two files reveals they are the same files that we observed codeit.exe repeatedly creating and deleting during our dynamic analysis.

Figure 9 shows a snippet of the AutoIt script source. The code is obviously obfuscated.

Figure 9: Snippet of obfuscated code found in AutoIt script source

UNDERSTANDING THE OBFUSCATION

Several features of codeit.exe are obfuscated, including control flow, number constants, and string constants.

CONTROL FLOW

codeit.exe leverages the order in which AutoIt executes code to obfuscate its control flow. AutoIt resolves directives (e.g. #Include) first and then uses a top-down approach to run code found in the global scope.
We see that codeit.exe makes heavy use of code located in the global scope to make it difficult to tell which functions are called and in what order. The directive #OnAutoItStartRegister found towards the top of the script is set to areihnvapwn indicating that the function areihnvapwn is the first function called by AutoIt. After this function returns AutoIt continues running code found in the global scope until it reaches a call to the function areialbhyut shown in Figure 10.

```autoit
Func areialbhyut($filename, $filemirror)
Local $filemirror = $false
Local $filemirror = DllStructCreate("struct:short:short")
Local $filemirror = areihnvapwn($false, False)
If $filemirror <> -1 Then
    Local $ischildhood = areihnvapwn($filemirror, DllStructGetPtr($filemirrorfreedeadmagic), DllStructGetSize($filemirrorfreedeadmagic))
    If $ischildhood <> -1 Then
        $filemirror = areihnvapwn($filemirror, DllStructGetPtr($filemirrorfreedeadmagic ? 0 : DllStructGetSize($filemirrorfreedeadmagic)));  
    EndIf
EndIf
EndFunc
```

**Figure 10: Call to function areialbhyut**

The function areialbhyut is our most likely candidate for "main".

**NUMBER CONSTANTS**

Number constants are obfuscated by their definition as global variables with random names as shown in Figure 11, rather than being referenced directly. We undo the obfuscation by rewriting references to those variables with their actual value.
Figure 11: Assigning number constants to global variables

We can remove the obfuscation of number constants using Python and regular expressions. Figure 12 shows one possible example of this solution.

```python
# find and replace global variables with number constant e.g. $flerqqjbmh = Number(" 1 ") -> 1 = Number(" 1 ")
for match in re.finditer(r"(\$fl\[0-9a-z\]{8})\s\=\sNumber\("\s(\[0-9\]+)\s\")\", source):
  source = source.replace(match.groups()[0], match.groups()[1])
```

Figure 12: Example code to remove obfuscation of number constants

Figure 13 shows the updated contents of the function arepfknwypw after removing the obfuscation of number constants.

```autoit
Func arepfknwypw()
Local $flqghbym = -1
Local $flqgyozgq = DIILStructCreate(archidarkq(Fos[129])
DIILStructSetData($flqgyozgq, 1, DIILStructGetSize($flqgyozgq))
Local $flqghdvgv = DIICall archidarkq(Fos[25]) archidarkq(Fos[26]) archidarkq(Fos[129]) archidarkq(Fos[139]) $flqgyozgq
If $flqghdvgv = 0 Then
  If DIILStructGetData($flqgyozgq, 2) = 0 Then
    $flqghbym = 0
  EndIf
EndIf
Return $flqghbym
EndFunc
```

Figure 13: Removing obfuscation of number constants from function arepfknwypw

### STRING CONSTANTS

String constants are hex-encoded and accessed through the global array $os. The array $os is initialized by the function areihnvapwn which is the first function called by AutoIt as a result of the #OnAutoItStartRegister directive.

The function areihnvapwn initializes the string variable $dlit to a single, large string containing all hex-encoded string constants separated by the string value 4FD5$. The string variable $dlit$ is then split into...
individual hex-encoded strings using the string value `4FD5$` as a delimiter and assigned to the array variable `$os`. After the function `areihnvapwn` executes string constants can be accessed by an index into the array `$os` and decoded by the function `arehdidxrgk`.

We can remove the obfuscation of string constants using Python and regular expressions. Figure 14 shows one possible example of this solution.

```python
dlit = []  
# collect contents of $dlit from source script
for match in re.finditer(r"\$dlit\s\&(\?)*\=\s"\([0-9a-zA-Z\$]{2,100}\)\", source):
    dlit.append(match.groups()[1])

string_array = []
# combine $dlit into single string, split on delimiter 4FD5$, and decode
for encoded in "".join(dlit).split("4FD5"):
    string_array.append(binascii.unhexlify(encoded).decode("utf-8"))

# find and replace $os array accesses with original string constant e.g. arehdidxrgk($os[25]) -> "kernel32.dll"
for match in re.finditer(r"(arehdidxrgk\($os\[(\[0-9]+)\])")", source):
    # (index – 1) as AutoIt arrays index starting at 1
    source = source.replace(match.groups()[0], "\"%s\"" % string_array[int(match.groups()[1], 10) - 1])
```

Figure 14: Example code to remove obfuscation of string constants

Figure 15 shows the updated contents of the function `arepfnkwypw` after removing the obfuscation of string constants.

```autoit
Func arepfnkwypw()
    Local $flagbdvyvy = -1
    Local $fltpvccvq = DllStructCreate("struct\(\棋牌游戏/\棋牌游戏/\棋牌游戏/\棋牌游戏/byte\(128\);\endstruct")
    DllStructSetData($fltpvccvq, 1, DllStructGetSize($fltpvccvq))
    Local $flagbdvyy = DllCall("kernel32.dll", "int", "GetVersionExA", "struct", $fltpvccvq)
    If $flagbdvyy[0] <> 0 Then
        If DllStructGetData($fltpvccvq, 2) = 6 Then
            If DllStructGetData($fltpvccvq, 3) = 1 Then
                $flgbtjbm = 0
            EndIf
        EndIf
    EndIf
    Return $flgbtjbm
EndFunc
```

Figure 15: Removing obfuscation of number and string constants from function `arepfnkwypw`

**UNDERSTANDING THE CODE**

After removing the obfuscation of number and string constants and identifying the function `areialbhuyt` as our most likely candidate for "main" we begin analyzing the code.

We notice that `codeit.exe` makes frequent use of the following AutoIt functions:

- `DllStructCreate`
**DllStructGetData**  
**DllStructSetData**  
**DllStructGetSize**  
**DllCall**  
**FileInstall**

These functions can be used in AutoIt scripts to call functions exported by a DLL. This is often used to call library code that performs CPU intensive tasks. codeit.exe performs a majority of its tasks using the Autotlt function DllCall to execute Windows library functions.

Taking a closer look at the function areialbhuyt we find the code executed when a user interacts with the Can haz code? button shown in Figure 16.

```autoit
Local $flnpapeken = $IC.IO.read($filename)
If $flnpapeken THEN
    Local $flnpapeken = DllStructGetData("struct:EscrowRequest[1600]:encoded",
        $flnpapeken, $.Zoom, $flnpapeken, $out, $flnpapeken)
If $out THEN
    Local $flnpapeken = DllStructSetData($flnpapeken, $out, $Zoom, $flnpapeken)
$flnpapeken = DllStructGetSize($flnpapeken)
    DllStructSetData($flnpapeken, $Zoom, $flnpapeken)
EndIf
If $flnpapeken THEN
    Local $flnpapeken = DllStructGetVar("$flnpapeken, $Zoom, $flnpapeken")
EndIf
EndIf
```

Figure 16: Code executed when user interacts with Can haz code? button

The calls at lines 352 and 356 execute the functions justGenerateQRSymbol and justConvertQRSymbolToBitmapPixels exported by a DLL named qr_encoder.dll. qr_encoder.dll is installed by the call at line 350 and deleted by the call at line 362. Each time a user interacts with the Can haz code? button codeit.exe creates and deletes the file qr_encoder.dll, which matches what we observed during our dynamic analysis. We ignore qr_encoder.dll and the calls at lines 352 and 356 for now based on our suspicion that the DLL simply contains code used to generate QR codes.

We see that the structure variable $flnpapeken is created at line 351 and passed to the function areyzotafnf at line 352. We also see that the structure variable $flnpapeken is passed to the function areyzotafnf at line 354 before it is passed to the function justConvertQRSymbolToBitmapPixels at line 356. Based on the names of the two exported functions we assume that the call at line 352 fills the structure variable $flnpapeken with QR symbol data which is then converted to bitmap pixels by the call at line 356. But why the call to the function areyzotafnf in between?

Taking a closer look at the function areyzotafnf we see a number of interesting calls to Windows library functions including CryptAcquireContextA, CryptoHashData, CryptoImportKey, and CryptoDecrypt. These functions indicate that the function areyzotafnf may decrypt data containing the challenge flag.

Figure 17 shows the code found in the function areyzotafnf.
We first determine the encryption algorithm used by taking a closer look at the calls to the Windows function CryptAcquireContextA at line 267 and the Windows function CryptImportKey at line 269. We see that the provider type specified for CryptAcquireContextA is 24 or PROV_RSA_AES. Based on the documentation for PROV_RSA_AES we know the encryption algorithm is one of RC2, RC4, or AES.

The second argument to the function CryptImportKey is a byte array containing a BLOBHEADER structure followed by the algorithm-specific encryption key. We see that the second argument to the function CryptImportKey is data stored at index five of the structure variable $fluelrpeax. Tracing uses of the structure variable $fluelrpeax reveals that index five is set to the binary variable $flmtvyzrsy at line 265. The binary variable $flmtvyzrsy stores the BLOBHEADER structure and is initialized at line 260 to the 12-byte value 080200001066000020000000 followed by data stored at index four of the structure variable $flnttmjfea.

Figure 18 shows the definition of the BLOBHEADER structure.
typedef struct _PUBLICKEYSTRUC {
    BYTE bType;
    BYTE bVersion;
    WORD reserved;
    ALG_ID aiKeyAlg;
} BLOBHEADER, PUBLICKEYSTRUC;

Figure 18: BLOBHEADER structure definition

The 12 bytes stored in the binary variable $flmtvyzrsy map to the values shown in Figure 19.

<table>
<thead>
<tr>
<th>Description</th>
<th>Size (in bytes)</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>1</td>
<td>0x8</td>
</tr>
<tr>
<td>Version</td>
<td>1</td>
<td>0x2</td>
</tr>
<tr>
<td>Reserved</td>
<td>2</td>
<td>0x0</td>
</tr>
<tr>
<td>Algorithm ID</td>
<td>4</td>
<td>0x6601</td>
</tr>
<tr>
<td>Key length</td>
<td>4</td>
<td>0x20</td>
</tr>
</tbody>
</table>

Figure 19: Mapping values store in the binary variable $flmtvyzrsy

Based on the documentation for ALG_ID, we determine that the encryption algorithm used is AES256. We also determine that index four of the structure variable $flnttmjfea contains the 32-byte AES256 encryption key used in the call to the function CryptDecrypt at line 271.

The call at line 271 uses the AES256 encryption key to decrypt data stored at index four of the structure variable $fluelrpeax. If the decryption is successful, the decrypted data is stored in the structure variable $flodiutpuy. The structure variable $flodiutpuy contains the data that is eventually passed to the function justConvertQRSymbolToBitmapPixels at line 356 of the function areialbhuyt. Therefore, we assume that the encrypted data at index four of the structure variable $fluelrpeax stores QR code-related data containing the challenge flag.

UNDERSTANDING THE AES256 KEY GENERATION

Tracing uses of the structure variable $flnttmjfea we find the data stored at index four is derived from calls to the Windows functions CryptCreateHash, CryptHashData, and CryptGetHashParam at lines 254, 256, and 258. The second argument to the Windows function CryptCreateHash, 32780 or CALG_SHA256, tells us that the call at line 256 generates a SHA256 hash and the call at line 258 stores the resulting hash value at index four of the structure variable $flnttmjfea. The second argument to CryptHashData reveals that the SHA256 hash is generated from data stored in the structure variable $flisilaylnraw.

We see that the structure variable $flisilaylnraw is initialized at line 248 to binary data returned from the call at line 244 and the call at line 244 is a simple wrapper function that executes the Windows function GetComputerNameA. Based on these findings we determine that the AES256 key is derived from the SHA256 hash of our system's computer name.
We see the structure variable $flisilaylnraw containing our system's computer name is passed to the function aregtfdcdyni at line 249. The code found in the function aregtfdcdyni appears to use an unrecognized algorithm to transform the computer name.

**UNDERSTANDING THE TRANSFORM FUNCTION**

Based on our analysis so far, the function aregtfdcdyni is passed our system's computer name and returns a transformation of the value. The SHA256 hash of the transformed string is used as the AES256 key to decrypt what we assume is QR code-related data containing the challenge flag.

Figure 20 shows the code found in the function aregtfdcdyni.

![Figure 20: Code found in function aregtfdcdyni](image)

The call at line 216 installs a file named sprite.bmp and the calls at lines 217, 219, and 222 read its contents into memory. We see that the data read from sprite.bmp is assigned to the structure variable $flxmdchrqd at line 224. The structure variable $flxmdchrqd splits the data between two arrays: the first array at index one stores 54 bytes and the second array at index two stores the remaining bytes.

The real magic happens at lines 227 through 234. Multiple bitwise operations are executed using bytes read from index two of the structure $flxmdchrqd. Because the structure $flxmdchrqd is mapped to the data read from sprite.bmp we know that index two stores data starting at offset 54 of sprite.bmp. We look at sprite.bmp in a hex editor and immediately notice irregularities in the data starting at offset 54 as shown in Figure 21.
Figure 21: Viewing sprite.bmp in hex editor

Lines 228 through 232 can be translated to the Python code shown in Figure 22.

```python
c = ord("a")
for bidex in reversed(range(7)):
    c += ((ord(f.read(1)) & 0x1) << bidex)
```

Figure 22: Lines 228 through 232 translated to Python

These lines sum each character of the original filename with seven, 7-bit numbers calculated from the least significant bits (LSB) of seven bytes read from sprite.bmp. An example of this calculation is shown in Figure 23.

<table>
<thead>
<tr>
<th>Byte Index</th>
<th>Calculated Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1 0 0 0 0 0</td>
</tr>
<tr>
<td>1</td>
<td>1 0 0 0 0</td>
</tr>
<tr>
<td>2</td>
<td>0 0 0 0 0</td>
</tr>
<tr>
<td>3</td>
<td>0 0 0 0</td>
</tr>
<tr>
<td>4</td>
<td>0 0 0</td>
</tr>
<tr>
<td>5</td>
<td>0 0</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
</tr>
</tbody>
</table>

```
1 1 0 0 0 0 1 = 97
```

Figure 23: Summing a 7-bit number extracted from sprite.bmp

The LSB of each of the seven bytes stores one bit of the 7-bit number where the LSB of the first byte stores the most significant bit (MSB) of the 7-bit number and the LSB of the last byte stores the LSB of the 7-bit number. Figure 24 shows an example of using this logic to convert the byte sequence `FFFFFEFEFEFEFFFFFFFFFFFEFF` back to its original two numbers.

```
FF FF FE FE FE FF = 1 1 0 0 0 0 1 = 97 = 'a'
FF FF FF FE FF FE FF = 1 1 1 0 1 0 1 = 117 = 'u'
```

Figure 24: Converting byte sequence `FFFFFEFEFEFEFFFFFFFFFFFEFF`
We can see from the example in Figure 24 that the calculations executed at lines 228 through 232 are used to decode characters hidden in sprite.bmp. Based on our analysis each character of the original computer name is summed with an encoded character read from sprite.bmp. The resulting sum is used for one last calculation at line 233.

Line 233 can be translated to the Python code shown in Figure 25.

```
chr((val // 2) + ((val & 0x1) << 7))
```

**Figure 25: Line 233 translated to Python**

Taking a closer look at line 233 we realize that the transformation algorithm used by the function aregtfdcyi is simple. Each character of the original computer name is summed with a character hidden in sprite.bmp. The final transformed character is the result of dividing that sum by two. Figure 26 shows two examples of this transformation.

```
('a' + 'a') / 2 = 'a'
('a' + 'e') / 2 = '?'
```

**Figure 26: Example transformations**

We can see that the final transformed character is equal to the original character if the original character and the encoded character are equal. But how does adding the result of 

```
((val & 0x1) << 7)
```

affect this calculation?

Short answer: it doesn't when we care.

Long answer: When the sum is even, meaning the two characters are the same, 

```
((val & 0x1) << 7)
```

equals zero and the final result is not changed.

From what we have learned the original computer name is not modified if each character matches the corresponding character hidden in sprite.bmp. That means the value hidden in sprite.bmp is probably the computer name that we need to correctly decrypt the QR code-related data.

We can use Python to extract the computer name hidden in sprite.bmp. Figure 27 shows one possible example of this solution.

```
encoded = []

with open("sprite.bmp", "rb") as f:
    f.seek(54, 0) # starting offset is 54
    for fidx in range(13):
        c = 0
        for bidex in reversed(range(7)):
            c += ((ord(f.read(1)) & 0x1) << bidex)
        encoded.append(chr(c))

print(f'Key: {"".join(encoded)}')
```

**Figure 27: Code to extract string hidden in sprite.bmp**
The computer name hidden in sprite.bmp is aut01tfan1999.

**SOLVING THE CHALLENGE**

Based on our analysis we determine that the QR code-related data found in the function areyzotafnf is successfully decrypted and displayed if our system's computer name is aut01tfan1999.

To confirm this, we set our system's computer name to aut01tfan1999, reboot, and re-run the program. After changing our system's computer name codeit.exe displays the same QR code regardless of the text entered as shown in Figure 28.

![Figure 28: QR code displayed by codeit.exe after changing computer name](image)

The QR code stores the challenge flag:

L00ks_L1k3_Y0u_D1dnt_Run_Aut0_Tim3_0n_Th1s_0ne!@flare-on.com