# SecureSoftware v1.5

https://crackmes.one/crackme/6049c26733c5d42c3d016de3

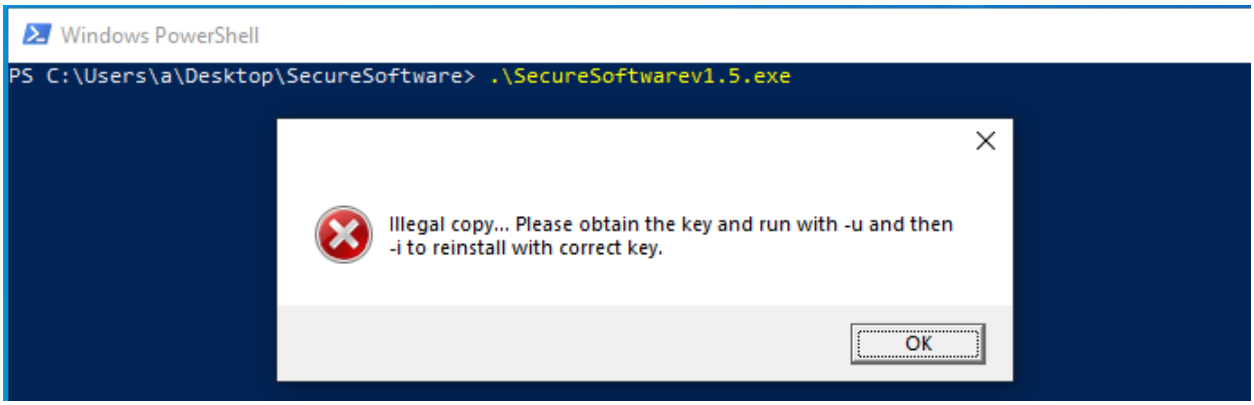
- windows 10 vm, ghidra, x64dbg

## Initial Run

Using Powershell to run the program using the command

```
.\SecureSoftwarev1.5.exe
```

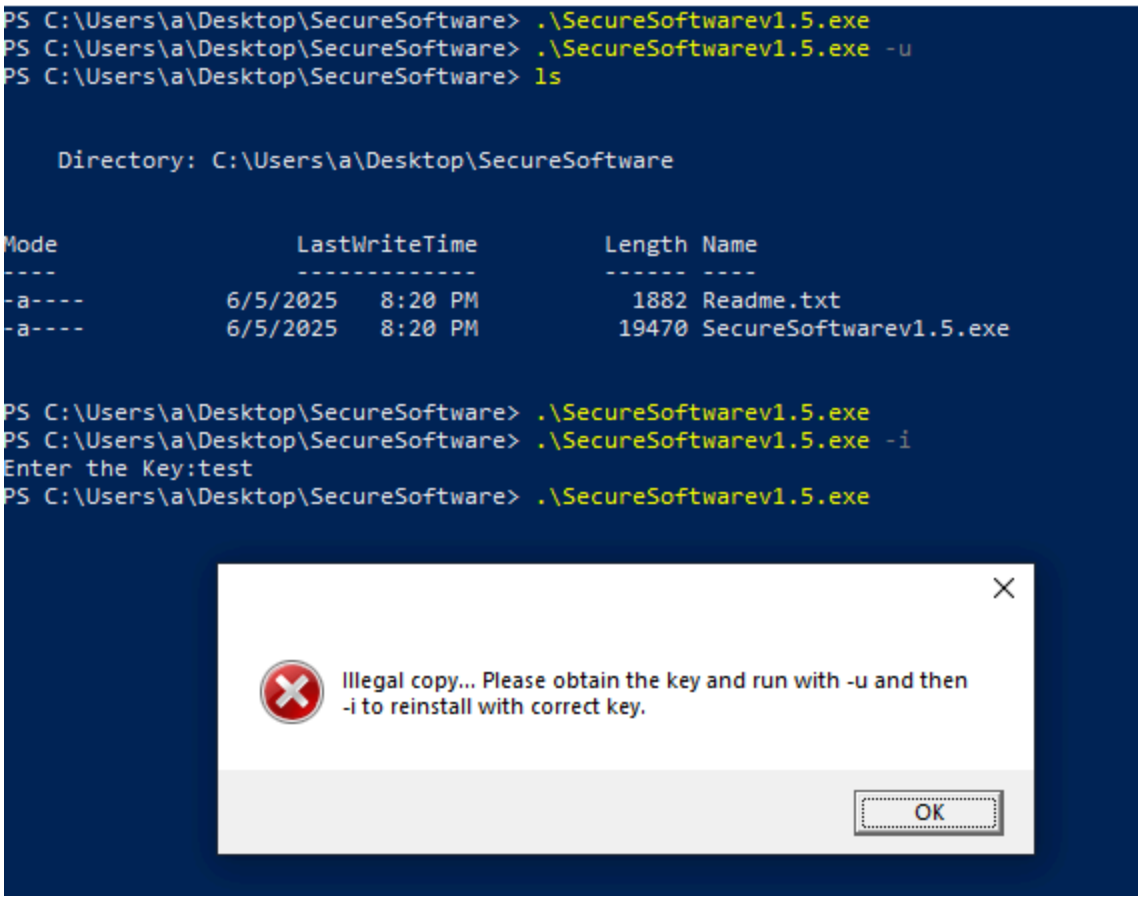
we get an error message prompting us to use additional flags

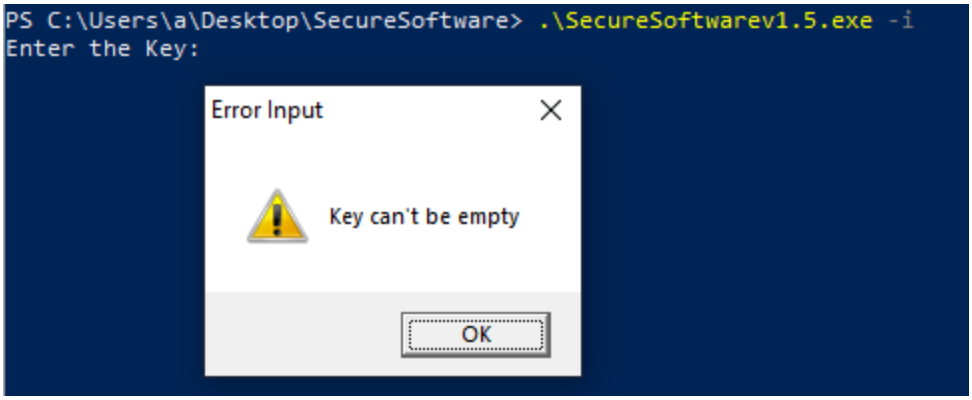



Following the prompt, I ran the following commands

```
.\SecureSoftwarev1.5.exe -u #uninitalize
.\SecureSoftwarev1.5.exe -i #initialize
```

The u flag uninitializes the program and when run prompts the user to initialize it using the i flag. Once the initialization command is ran, the program asks for a key. Afterwards the program is to be ran once more. With an incorrect key we get a prompt warning us that we are using an illegal copy and to use the correct key. Trying to enter an empty string we get a different error message as well.



## Reverse Engineering

It seems like it's time to put this program into ghidra and start reverse engineering. Since the readme states that there are anti-debugging measures in place, I would like to complete this using static analysis.

Hoping that one of these error messages would lead to a good place to start, I was able to find the error message that the key cannot be empty.

```
if (local_90[0] == '\0') {
  MessageBoxA((HWND)0x0,"Key can\'t be empty","Error Input",0x30);
  fclose(local_24);
  FUN_0040174f();
                    /* WARNING: Subroutine does not return */
  exit(0);
}
```

Just above this code, we can see the following function:

```
FUN_00401781(PTR_s_Foufs!uif!Lfz;_00404014);
```

This happens to be encrypted using a substitution cipher ROT1, and when we decrypt it we get to see that this is indeed the prompt to enter the key


https://www.dcode.fr/rot1-cipher

Since that function was a ROT1 Encryption, I assumed there would be more variables encrypted the same way, and there were a few more.

```
00404008 44 50 40 00     addr        s_cv2pr_00405044

                    PTR_s_voefgjofe!bshvnfout///!.j!up!jot_0040400c XREF[1]:
0040400c 4c 50 40 00     addr        s_voefgjofe!bshvnfout///!.j!up!jot_0040504c

                    PTR_s_Uif!tpguxbsf!jt!opu!jojujbmj{fe/_00404010 XREF[1]:
00404010 ac 50 40 00     addr        s_Uif!tpguxbsf!jt!opu!jojujbmj{fe/_004050ac

                    s_rotl_EnterTheKey                              XREF[1]:
00404014 e9 50 40 00     addr        s_Foufs!uif!Lfz:_004050e9

                    PTR_s_Jmmfhbm!dpqz///!Qmfbtf!pcubjo!ui_00404018 XREF[1]:
00404018 f8 50 40 00     addr        s_Jmmfhbm!dpqz///!Qmfbtf!pcubjo!ui_004050f8

                    PTR_s_Tvddftt""!Uif!qsphsbn!ibt!cffo!v_0040401c XREF[1]:
0040401c 5c 51 40 00     addr        s_Tvddftt""!Uif!qsphsbn!ibt!cffo!v_0040515c

                    PTR_s_Lfz!opu!gpvoe//_00404020                  XREF[1]:
00404020 85 51 40 00     addr        s_Lfz!opu!gpvoe//_00405185

                    DAT_00404024                                    XREF[2]:
```

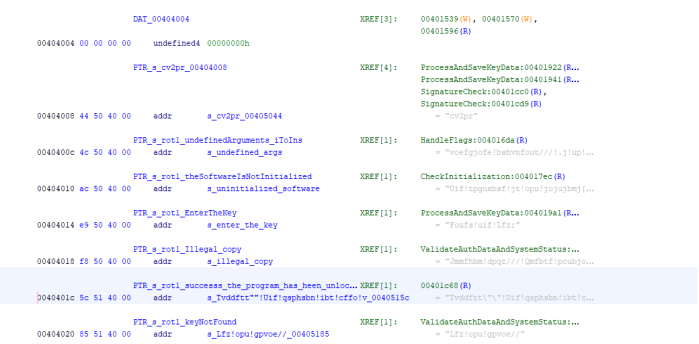
I uncovered messages for "Illegal copy," "Success," and "Key not found," as well as a significant string, `cv2pr`, located at address `0x00404008`. After deciphering the strings, I followed their trails to see where it could lead me. I then came across a piece of code that calls a function before it does a check to throw an error of it being an illegal copy.

```
else {
  FUN_00401d0c((int)local_f8);
  local_10 = 0;
  for (local_14 = 0; local_14 < (int)local_1c; local_14 = local_14 + 1) {
    if (DAT_00407618[local_14] != -1) {
      EncodeRot1(PTR_s_rot1_Illegal_copy);
      MessageBoxA((HWND)0x0,&Prompt_String,"",0x10);
      local_10 = 1;
      break;
    }
  }
```

I then followed that function and with renaming some of the variables somewhat poorly, we can see that there is a comparison with some strings and obfuscation happening, so I believe we are going in the right path, but maybe not exactly where I wanted to be right now.

```
if (DAT_validationFlag != 0) {
  charTransformed = strcmp((char *)(param_1 + 0xc),&DAT_expectedString1);
  if ((charTransformed == 0) &&
     (charTransformed = strcmp((char *)(param_1 + 0x3e),&DAT_expectedString2),
     charTransformed == 0)) {
    DAT_validationFlag = 1;
  }
  else {
    DAT_validationFlag = 0;
  }
  expectedStrLen1 = strlen(&DAT_expectedString1);
  expectedStrLen2 = strlen(&DAT_expectedString2);
  _Str = (char *)(param_1 + 0x70);
  if ((DAT_validationFlag == 1) && (DAT_processingBlockedFlag == 0)) {
    if (DAT_reverseFlag == 5) {
      _strrev(_Str);
    }
    inputLen = strlen(_Str);
    for (i = 0; i < (int)expectedStrLen1; i = i + 1) {
      (&DAT_expectedString1)[i] = (char)(&DAT_expectedString1)[i] % '\x10';
    }
    for (j = 0; j < (int)expectedStrLen2; j = j + 1) {
      (&DAT_expectedString2)[j] = (char)(&DAT_expectedString2)[j] % '\x10';
    }
    for (k = 0; k < (int)inputLen; k = k + 1) {
      charTransformed = FUN_00402190(_Str[k]);
      if (k < (int)expectedStrLen2) {
        if (charTransformed == (char)(&DAT_obfuscatedStr2)[expectedStrLen2 - k]) {
          _Str[k] = -1;
        }
        else {
          _Str[k] = '\0';
        }
      }
      else if ((k < (int)expectedStrLen2) || ((int)(expectedStrLen2 + expectedStrLen1) <= k)) {
        _Str[k] = '\0';
      }
      else if (charTransformed ==
               (char)(&DAT_obfuscatedStr1)[expectedStrLen1 - (k - expectedStrLen2)]) {
        _Str[k] = -1;
      }
      else {
        _Str[k] = '\0';
      }
    }
  }
  return;
```

After going through the code renaming as much as I can, I returned to the encrypted and dove into the success string. This allowed me to see that there were many conditions I had to meet to get the success message.



The final success message is displayed inside my renamed ValidateSuccessProgram function. To reach the `MessageBoxA` call that shows "Success", the following conditions must be met by bypassing several conditional jumps:

- `DAT_secondaryFlag?` must be `0`
- `DAT_signatureMatchcv2pr` must be `1`
- `DAT_validationFlag` must be `1`
- `DAT_authFail` must be `0`
- `debuggerPresent` must be `2`?

These are the variable names I have given each of these flags, and they are scattered throughout the code but are set based on the outcomes of the integrity checks and the validation of `authdata.dat`.

To figure out how to set these flags correctly, I had to understand the full lifecycle of the key data. The entire validation process centers around a file named `authdata.dat` that the program creates in a `data` subfolder within the user's profile. The validation function first opens and reads this 220-byte file into memory.

```
local_18 = fopen(".\\data\\authdata.dat","rb");
if (local_18 == (FILE *)0x0) {
  EncodeRot1(PTR_s_rot1_keyNotFound);
  MessageBoxA((HWND)0x0,&Prompt_String,"",0);
                    /* WARNING: Subroutine does not return */
  exit(-1);
}
HandleAuthDataOrKey((byte *)ObfuscateStringIfExpectedMatch);
fread(local_f8,0xdc,1,local_18);
```

0xdc = 220

Going back to the obfuscation function now, it seems like it is also has big role in validation. The function shows that `authdata.dat` has a very specific structure. The function first compares sections of the file against the current user's name and computer name, which are stored at fixed offsets. Username at offset `0xc`, and computer name at offset `0x3e`

```
charTransformed = strcmp((char *)(param_1 + 0xc),&DAT_username);
if ((charTransformed == 0) &&
   (charTransformed = strcmp((char *)(param_1 + 0x3e),&DAT_computername), charTransformed == 0))
{
  DAT_validationFlag = 1;
}
else {
  DAT_validationFlag = 0;
}
```

Just a bit further, it processes a key string located at offset `0x70`.

```
_Str = (char *)(param_1 + 0x70);
```

There is also a step involving a conditional string reversal ( `_strrev` ) if a specific flag, `DAT_reverseFlag`, is set to 5.

An important piece of the puzzle was uncovering the algorithm used to generate this key string. After the potential reversal, the validation function generates two hashes or obfuscated strings by taking the ASCII value of each character of the

current username and computer name and applying a modulo 16 operation.

```
if ((DAT_validationFlag == 1) && (DAT_processingBlockedFlag == 0)) {
  if (DAT_reverseFlag == 5) {
    _strrev(_Str);
  }
  inputLen = strlen(_Str);
  for (i = 0; i < (int)expectedStrLen1; i = i + 1) {
    (&DAT_username)[i] = (char)(&DAT_username)[i] % '\x10';
  }
  for (j = 0; j < (int)expectedStrLen2; j = j + 1) {
    (&DAT_computername)[j] = (char)(&DAT_computername)[j] % '\x10';
  }
```

After finding the hashing, I needed to see how the key I entered was being processed for comparison. I determined it was this function decoding a hex character into its integer value.

```
int __cdecl DecodeHexToChar(byte param_1)

{
  int iVar1;

  if (param_1 < 0x47) {
    iVar1 = param_1 - 0x30;
    if (9 < iVar1) {
      iVar1 = param_1 - 0x37;
    }
  }
  else {
    iVar1 = -1;
  }
  return iVar1;
}
```

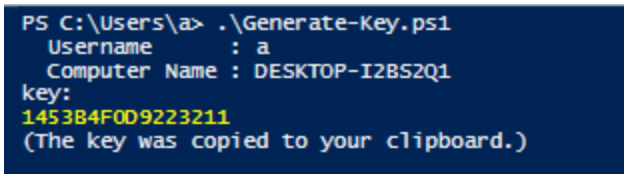
The program then compares the output of this function against the transformed username and hostname hashes. With this knowledge, to create a keygen, I

needed to perform the inverse of the `DecodeHexToChar` function and turn bytes into hex.
Using this complete algorithm, I created a PowerShell keygen to generate the correct key string.

```
<#
.SYNOPSIS
   keygen for the SecureSoftware v1.5 crackme.
#>
function Generate-Key {
   [CmdletBinding()]
   param (
      #username
      [string]$Username = $env:USERNAME,
      #computer name
      [string]$ComputerName = $env:COMPUTERNAME
   )
   Write-Host "Username      : $Username"
   Write-Host "Computer Name : $ComputerName"

   function Reverse-KeySubs {
      param([byte]$Value)

      if ($Value -gt 9) {
         #letters
         return [char]($Value + 0x37)
      } else {
         #numbers
         return [char]($Value + 0x30)
      }
   }

   $usernameHash = $Username.ToCharArray() | ForEach-Object { [byte]($_)
-band 0xf }
```

```
    $computernameHash = $ComputerName.ToCharArray() | ForEach-Object {
[byte]($_) -band 0xf }

    $stringBuilder = New-Object System.Text.StringBuilder
    foreach ($byte in $usernameHash) {
        $null = $stringBuilder.Append((Reverse-KeySubs -Value $byte))
    }
    foreach ($byte in $computernameHash) {
        $null = $stringBuilder.Append((Reverse-KeySubs -Value $byte))
    }
    $finalKey = $stringBuilder.ToString()



    Write-Host "key:"
    Write-Host -ForegroundColor Yellow $finalKey

    try {
        Set-Clipboard -Value $finalKey
        Write-Host "(The key was copied to your clipboard.)"
    } catch {
        Write-Warning "Could not copy to clipboard. Please copy the key manuall
y."
    }
}
Generate-Key
```

Time to test out the script. It output a key.



Now I would have to try to initialize the program with the key, and it seems to have been successful!