deroko [ARTeam], 2kAD [ARTeam]

Special Issue for SecuRom 7.30.0014 Take2 VM Analysis

Version 1.0
Last Rev.: December 2007

Into this Tutorial

Editor: Shub-Nigurrath

Forewords

After the publication of our previous tutorial on SecuROM [1] we had a lot of discussions about its title, on several forums we had comments about if it was a really complete owning or not.

On woodmann forum (www.woodmann.com, search for "*A continued discussion on "ownage"*", and "*ARTeam: Special Issue For SecuRom 7.30.0014 Complete Owning, AnonymouS, Human, deroko*") we had an interesting discussion on what can be considered owning of a program and what cannot.

The discussion has been really long and I'll not report it here, but would summarize the two main positions.

Owning is when you return the program to be virgin as it was when it was still to be packed. So cracks around which circumvent the protection without actually removing it are not real own. I would call this position the position of "Purists".

On the other hand there's the position of who tells (and I'm among them) that any method is legit to fool the protection. It the application fails its task (protecting) then it makes no sense to even exist. Protections are placed to protect and only for that reason, often even at the price of portability and efficiency of the code. If you, using any dirty method, successfully fool the application, letting it run in unforeseen contexts (e.g. a not licensed PC), you then owned the protection. Or better you might not have run the protector (not understood all its aspects) but surely you owned the protection. And a protector without a protection is nothing in my humble opinion.

"Purists" approach is much more hard because implies also understanding parts of the protection not really needed to break it. It's something one can call professional reverse engineering. If you are e developer you are anyway interested in techniques useful to avoid *any* type of reverse engineering, professional or not, which owns your application or your protection or your protector.

You can think more or less the same when you try to distinguish cracking from reverse engineering..

This said we decided to show that owning on SecuROM can be done either ways you consider it. This second issue on the protector tries to cover some aspects that was not covered in previous one, specifically the most important part of the SecuROM protector, the Virtual Machine, it's really well done: is recursive and changes from application to application ...

This time deroko again hits the ground, but it's also the first time of 2kAD. I hope you will enjoy it, but I must warn you, this one is not going to be an easy one ☺

Have phun,
Shub

Disclaimers

All code included with this tutorial is free to use and modify; we only ask that you mention where you found it. This tutorial is also free to distribute in its current unaltered form, with all the included supplements.

All the commercial programs used within this document have been used only for the purpose of demonstrating the theories and methods described. No distribution of patched applications has been done under any media or host. The applications used were most of the times already been patched, and cracked versions were available since a lot of time. ARTeam or the authors of the paper cannot be considered responsible for damages to the companies holding rights on those programs. The scope of this tutorial as well as any other ARTeam tutorial is of sharing knowledge and teaching how to patch applications, how to bypass protections and generally speaking how to improve the RCE art and generally the comprehension of what happens under the hood. We are not releasing any cracked application. We are what we know..

Verification

ARTeam.esfv can be opened in the ARTeamESFVChecker to verify all files have been released by ARTeam and are unaltered. The ARTeamESFVChecker can be obtained in the release section of the ARTeam site: http://releases.accessroot.com

Table of Contents

SecuROM : Even caveman can do it by deroko

1. Forewords

This tut was done 2 months ago, but due to lack of interest to release it we didn't release it immediately. Whenever new document is written, I always try to think what you, as a reader, will learn from it. To be honest, this dilemma was the only thing which stoped me from publishing this tutorial 2 months ago, when it was done. I still don't know what you will learn from it, but still, I decided to publish it.

So, enjoy in it ☺

2. Tools and Target

Target used this time is game protected with SecuROM 7.34 demo version as I wasn't able to get full version at the time of writing this tutorial, but everything from this tutorial should be ***mutatis mutandis*** applied to full version without any problem.

Tools that we will need, are:

- SoftICE
- IDA

3. Few words about SecuROM VM

First of all SecuROM is composed of 256 handlers, each handler is responsible for performing simple operation which exist in IA32 (eg. no instructions such as mov [vm_reg], [vm_reg]). Opcodes on other hand don't have predefined format, and execution of opcodes is dependent on execution of previous opcode(s), as if one of opcodes is not executed you will break predefined flow of a VM.

Well you'll see all of this as we go along with VM analyze, and you will see how easy is to extract all needed info to write emulator or vm decompiler. Although easy doesn't mean short and fast coding ☺

4. VM analyse

VM_Enter is code (well you may name it different) responsible for setting VM_Context, and for dispatching execution to VM handlers.

Let's see one example of VM_Enter:

```
.text:10D9D436 loc_10D9D436:
.text:10D9D436                 push    eax             ; nShowCmd
.text:10D9D437                 push    esi             ; lpCmdLine
.text:10D9D438                 push    0               ; hPrevInstance
.text:10D9D43A                 push    offset __ImageBase ; hInstance
.text:10D9D43F                 call    _WinMain@16     ; WinMain(x,x,x,x)
```

…

```
.text:10912B10 ; int __stdcall WinMain(HINSTANCE hInstance,HINSTANCE hPrevInstance,LPSTR
lpCmdLine,int nShowCmd)
.text:10912B10 _WinMain@16     proc near
.text:10912B10
.text:10912B10 var_C           = dword ptr -0Ch
.text:10912B10
.text:10912B10
.text:10912B10                 jmp     ds:off_11E0515C
.text:10912B10 _WinMain@16     endp
```

And we come to code responsible for calling VM:

```
.bla:11E95FB0 win_main_ref:
.bla:11E95FB0                  push    offset vm_argument
.bla:11E95FB5                  push    401149h         ; dummy argument to simulate call
.bla:11E95FBA                  push    (offset loc_11639F6D+3)
.bla:11E95FBF                  pushf
.bla:11E95FC0                  sub     dword ptr [esp+4], 1ABB0h
.bla:11E95FC8                  popf
.bla:11E95FC9                  retn                    ; goes to 1161F3C0

.bla:11E95FCA vm_argument      dd offset vm_opcodes_0
.bla:11E95FCE                  dd 1C93h
.bla:11E95FD2                  dd 6B21h
.bla:11E95FD6                  dd 4BE4h
.bla:11E95FDA                  dd 0
.bla:11E95FDE                  db    0
.bla:11E95FDF                  db    0
```

So far so good, and we are almost there, but before we enter into VM_Enter there are certain stuff that I wanna show at this point:

1. You may see my comments and that address 401149h is dummy argument used to simulate call to VM. This is logical, as this argument is later on used as x86 EIP.
2. Code between pushfd/popfd is used to calculate offset of VM_Enter.
3. VM_Argument is important as it is used on other hand to tell VM where are Opcodes and what set of vm handlers to use.

```
.bla:11E76756 vm_opcodes_0     dd 70B2F0C5h, 0D491B739h, 0D845BB32h,
                                  13383248h, 120078DAh
.bla:11E76756                  dd 995DD350h, 1137B10h, 630F31E6h, 98B64673h,
                                  48BDCD2Fh
.bla:11E76756                  dd 0B2E7CA02h, 4C86144Dh, 3A72F00h, 6290B218h,
                                  0E287B2h
.bla:11E76756                  dd 1A175D8Dh, 0D9C28D4Dh, 87614CA1h,
                                  0DC1295ADh, 0BA815B98h
.bla:11E76756                  dd 204D9623h, 8D76ED4Dh, 67C70000h, 7571002Dh,
                                  0D63A685Ch
.bla:11E76756                  dd 7B55CA91h, 972CFEB1h, 5E7FA624h, 0CF06AC15h,
                                  13D62E2Ch
.bla:11E76756                  dd 0C39CBB56h, 13392C20h, 29DA135Ah, 5F68ADF5h,
                                  0CC40ED6Eh
.bla:11E76756                  dd 2B22794h, 0D1B64D29h, 112298E0h, 9435A453h
.bla:11E767F2                  dd 5296C311h
```

Oki now is time to show full VM_enter and to cover it line by line for better understanding:

```
valloc:018B0000 vm_enter_main_handler:
valloc:018B0000                 jmp     short __skip_space_for_rent ; esp-20
valloc:018B0000
valloc:018B0002                 db  3Ch ; <
valloc:018B0003                 db  20h
valloc:018B0004                 db  73h ; s
valloc:018B0005                 db  70h ; p
valloc:018B0006                 db  61h ; a
valloc:018B0007                 db  63h ; c
valloc:018B0008                 db  65h ; e
valloc:018B0009                 db  20h
valloc:018B000A                 db  66h ; f
valloc:018B000B                 db  6Fh ; o
valloc:018B000C                 db  72h ; r
valloc:018B000D unk_18B000D     db  20h
valloc:018B000E                 db  72h ; r
valloc:018B000F                 db  65h ; e
valloc:018B0010 unk_18B0010     db  6Eh ; n
valloc:018B0011                 db  74h ; t
valloc:018B0012                 db  20h
valloc:018B0013                 db  3Eh ; >
valloc:018B0014
valloc:018B0014
```

Funny, really funny, but as this code obviously has no any meaning we may skip it. Now we enter into VM_wait loop, which will wait until byte at 18B0020 isn't set to 1 again.

```
valloc:018B0014 __skip_space_for_rent:
valloc:018B0014                 pusha                   ; esp-20
valloc:018B0015                 pushf                   ; esp-24
valloc:018B0016                 call    $+5             ; esp-28
valloc:018B001B                 call    sub_18B0022
valloc:018B001B
valloc:018B0020 byte_18B0020    db 1
valloc:018B0021                 db 1
valloc:018B0022
valloc:018B0022
valloc:018B0022
valloc:018B0022 sub_18B0022     proc near
valloc:018B0022
valloc:018B0022                 pop     edx
valloc:018B0023
valloc:018B0023 __vm_wait_loop:
valloc:018B0023                 lock dec byte ptr [edx]
valloc:018B0026                 jns     short __vm_ready
valloc:018B0028
valloc:018B0028 __vm_wait:
valloc:018B0028                 cmp     byte ptr [edx], 0
valloc:018B002B                 pause
valloc:018B002D                 jle     short __vm_wait
valloc:018B002F                 jmp     short __vm_wait_loop
valloc:018B002F
valloc:018B0031 aMassesAgainstT db '-[ Masses Against the Classes <¦>>< ]-',0
```

In this code you may see that SecuROM gives you 10 possible VM_context which means that there is possibility of executing 10 threads in VM at the same time. (mov ecx, 0Ah). Each context struct will be checked for busy flag, until one of them isn't released by VM_exit handlers (I'll cover a few latter on). Note how byte at 18B0020 is used as vm interpreter busy flag.

```
valloc:018B0058
valloc:018B0058 __vm_ready:
valloc:018B0058
valloc:018B0058                 mov     eax, 0FFFFFFFCh
valloc:018B005D                 mov     ecx, 0Ah
valloc:018B0062
valloc:018B0062 __find_free_vmcontext:
valloc:018B0062                 add     eax, 4
valloc:018B0067                 xchg    ebp, ebp
valloc:018B0069                 adc     ebp, eax
valloc:018B006B                 mov     edi, ds:vm_contexts[eax]
valloc:018B0071                 cmp     [edi+srom_vm_context.busy], 0DE859E9h
valloc:018B0078                 jnz     short __vm_context_free_found
valloc:018B007A                 loop    __find_free_vmcontext
valloc:018B007C                 pause
valloc:018B007E                 jmp     short __vm_ready
valloc:018B0080
```

Now SecuROM simply walks all 10 vm_cotexes and checks busy flag in all of them, once free context is found it will zero whole context, mark it as used, and simply will set byte as 18B0020 to 1, which means that vm_interpreter is ready to handle another thread. You may see that in following code:

```
valloc:018B0080
valloc:018B0080 __vm_context_free_found:
valloc:018B0080                 cld
valloc:018B0081                 mov     ebx, edi
valloc:018B0083                 mov     ecx, 100h
valloc:018B0088                 mov     eax, 0
valloc:018B008D                 rep stosd
valloc:018B008F                 mov     [ebx+srom_vm_context.busy], 0DE859E9h
valloc:018B0096                 mov     al, [edx+1]
valloc:018B0099                 xchg    al, [edx]
```

VM_context is marked as used, and now SecuROM will processed with filling important parts of vm_context with needed data.

```
valloc:018B009B                 sub     dword ptr [esp], 1Bh ; edx = 018B0000
valloc:018B009B                                      ;           |
valloc:018B00A2                 pop     edx          ; <---------+
```

At this point SecuROM vm_interpreter is calculating it's offset in the memory which will be later used as delta to VM_Opcodes and delta to VM_Handlers.

```
valloc:018B00A3                 mov     eax, [esp+28h]
valloc:018B00A3
valloc:018B00A3
valloc:018B00A7                 mov     [ebx+srom_vm_context.argument], eax
valloc:018B00AA                 mov     eax, [eax] ; grab dword from argument
valloc:018B00AC                 sub     eax, edx
```

If you remember I showed earlier that argument has pointer to vm_opcodes, so this is the point when that dword is taken and SecuROM makes it relative to vm_enter using this simple formula:

VM_opcodes – offset vm_enter

```
valloc:018B00AE             push    eax
valloc:018B00AF             mov     eax, [esp+2Ch]  ; argument
valloc:018B00B3             cmp     dword ptr [eax+0Ch], 50h ; argument + 0C
valloc:018B00BA             jnz     short loc_18B00CA
valloc:018B00BC             xchg    ebp, ebp
valloc:018B00BE             adc     ebp, eax
```