Creating Loaders & Dumpers - Crackers Guide to Program Flow Control

26/June/04 - Robert Yates

http://www.reverse-engineering.info/files/lad_files.rar

Introduction

So you want to unpack a program, aspack? asprotect? even safedisc?
to accomplish such a task a degree of knowledge is needed in many
many different areas, over the years i have been writting small
tutorials on these areas before i ever write a comprehensive tutorial
on a single subject.

ok so one thing that ive noticed is that 'newbies' have no sense of
how they're going to carry out all the tasks needed to repair a exe,
most plan to just dump an exe image and try and fix bits as they go
along, but a much more structured way can be taken and this is to
create a 'dumper' which effectivily launchs your target exe and
halts it in certains places so you can read some memory areas and save
data, and eventually end up at OEP when you can dump the sections to
disk and make your necessary changes.

Ok this is common knowledge to 80% of crackers excluding the
ones that message me ;-) so i doubt many people will be reading
this paper, with that lets begin one of my rare essays hehe.

How the demostration will happen

For this example im going to take a UPX packed notepad and show you how
to code a program to stop it at the point where the imports are being
resolved, then im going to output the data to screen as they get resolved
just as an example, at this point really if you were unpacking the exe you
would grab the data and produce a fresh import table. after outputing the
import data im going to then let the progam continue to OEP, halt it there
and show a msgbox.

Examing the target

Ok before i explain the process of controlling the program flow lets look
at our target and find what we have to do, i've protected my notepad with
upx and took 5mins to study how it works, i'll now briefly explain.

UPX entry point looks like this,

```
UPX1:01011651                 mov     esi, offset dword_100D000
UPX1:01011656                 lea     edi, [esi-0C000h]
UPX1:0101165C                 push    edi
UPX1:0101165D                 or      ebp, 0FFFFFFFFh
UPX1:01011660                 jmp     short loc_1011672
```

now if you scroll down 4 pages in ida you can clearly see the OEP

```
UPX1:0101179E loc_101179E:                            ; CODE XREF: start+110j
UPX1:0101179E                 popa                              ; resore registers
UPX1:0101179F                 jmp     near ptr dword_1006420
UPX1:0101179F start           endp
```

so 0101179F is our final destination.

the import loader code looks like this

```
UPX1:0101175C GET_DLLNAME_AND_THUNK:                  ; CODE XREF: start+12Ej
UPX1:0101175C                 mov     eax, [edi]      ; NO
UPX1:0101175E                 or      eax, eax
UPX1:01011760                 jz      short loc_101179E
UPX1:01011762                 mov     ebx, [edi+4]
UPX1:01011765                 lea     eax, [eax+esi+11CF4h]
UPX1:0101176C                 add     ebx, esi
UPX1:0101176E                 push    eax             ; DLL NAME
UPX1:0101176F                 add     edi, 8
UPX1:01011772                 call    dword ptr [esi+11DA8h] ; LOADLIBRARY
UPX1:01011778                 xchg    eax, ebp
UPX1:01011779
UPX1:01011779 BUILD_THUNK:                            ; CODE XREF: start+146j
UPX1:01011779                 mov     al, [edi]
UPX1:0101177B                 inc     edi
UPX1:0101177C                 or      al, al
UPX1:0101177E                 jz      short GET_DLLNAME_AND_THUNK ; NO
UPX1:01011780                 mov     ecx, edi
UPX1:01011782                 push    edi             ; PTR ASCII NAME
UPX1:01011783                 dec     eax
UPX1:01011784                 repne scasb
UPX1:01011786                 push    ebp
UPX1:01011787                 call    dword ptr [esi+11DACh] ; GETPROCADDRESS
UPX1:0101178D                 or      eax, eax        ; ADDRESS
UPX1:0101178F                 jz      short loc_1011798
UPX1:01011791                 mov     [ebx], eax      ; WRITE TO THUNK
UPX1:01011793                 add     ebx, 4
UPX1:01011796                 jmp     short BUILD_THUNK
```

it starts off by reading a block of data stored in EDI, e.g.

```
UPX1:01010000                dd 0FCh                  ; DLL NAME POINTER
UPX1:01010004                dd 80h                   ; THUNK START
UPX1:01010008                db    1
UPX1:01010009 aLocalunlock   db 'LocalUnlock',0
UPX1:01010015                db    1
UPX1:01010016 aGlobalunlock  db 'GlobalUnlock',0
UPX1:01010023                db    1
UPX1:01010024 aGloballock    db 'GlobalLock',0
UPX1:0101002F                db    1
UPX1:01010030 aGetlasterror  db 'GetLastError',0
```

and as you can see by my comments the structure is pointer to name, thunk location and then a list of functions for that dll

the dll pointer is fixed up and read and UPX loads the library here,

```
UPX1:0101176E                push    eax              ; DLL NAME
UPX1:0101176F                add     edi, 8
UPX1:01011772                call    dword ptr [esi+11DA8h] ; LOADLIBRARY
```

it then reads the 80h and adds the section base to it and puts it in EBX this will be the thunk for the current dll and where all the resolved address for the api list will be placed, if you understand import tables then you know this is the point you should replace the api address with a pointer to the name.

anyway, so then further down it reads the api name then performs getprocaddress

```
UPX1:01011782                push    edi              ; PTR ASCII NAME
UPX1:01011783                dec     eax
UPX1:01011784                repne scasb
UPX1:01011786                push    ebp              ; current dll base
UPX1:01011787                call    dword ptr [esi+11DACh] ; GETPROCADDRESS
UPX1:0101178D                or      eax, eax         ; ADDRESS
UPX1:0101178F                jz      short loc_1011798
UPX1:01011791                mov     [ebx], eax       ; WRITE TO THUNK
```

ok so for fun we will stop the program at 01011782, output the current function then continue to 0101178D and output the api address :-)

Objectives
----

ok here is our mission plan

* Start Executable
* place a stop point at oep - 0101179E
* stop at 0101176C print the dll name
* stop at 01011780 print the ascii name
* stop at 01011796 print the api address
* loop these stop points until we get to oep

Im going to be showing my examples in ASM using the compiler TASM, i will also try and include C++ source codes in the final src for you new generation coders ;-)

Theory

In order to control the program the idea is we start the application in a suspended mode then we write into the programs memory the bytes 0EBh 0FEh where we want to stop, these 2 bytes are the opcodes for JMP -2 and since the instruction is 2 bytes long this causes a constant loop and the instruction keeps executing it self, so we insert these where
we want to stop then resume the program, if we wait a few milliseconds the program will become trapped in this loop, we can check what address is currently being executed using an API, so once we detect we've stopped at our target address we can then take action.

The APIs you need to know are the following

CreateProcess - Load an external executable.
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dllproc/base/createprocess.asp

ResumeThread / SuspendThread - Used to stop and start the process thread in its current state
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dllproc/base/resumethread.asp
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dllproc/base/suspendthread.asp

WriteProcessMemory / ReadProcessMemory - Used to insert our JMP -2 and read process memory
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/debug/base/writeprocessmemory.asp
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/debug/base/readprocessmemory.asp

GetThreadContext / SetThreadContext - Used to get the Register values from the running process.
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/debug/base/getthreadcontext.asp
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/debug/base/setthreadcontext.asp

Pratical
----

* place a stop point at oep - 0101179E
* stop at 0101176C print the dll name
* stop at 01011780 print the ascii name
* stop at 01011796 print the api address

So we have 4 stop points, it important to plan when placing these, in a proper cracking process you might perhapes inject a dll into the process(see my hooking import table tut) and then patch in jumps at the hook points so the target jumps into your dll and performs some operations and jumps back.

In this case we are inserting EB FE into the exe, but we are inserting them inside a loop that resolve imports. When we place the EB FE we are destroying data, so its a good idea to find a suitable place to put them, example.. over a 2 byte instruction we can emulate, let now look for good places to put our hooks.

1. OEP.

It doesnt matter where we place it since we are terminating the program when we reach it so lets choose. 0101179E 61   popa

2. DLLNAME

```
UPX1:0101176C 01 F3                           add     ebx, esi
UPX1:0101176E 50                              push    eax             ; DLL NAME
```

0101176C is a good place because we can read EAX to get the dll, and also grab ebx, esi add them and insert the result back into ebx, ok get the idea?

3. ASCII NAME

```
UPX1:01011780 89 F9                           mov     ecx, edi
UPX1:01011782 57                              push    edi             ; PTR ASCII NAME
```

same again 01011780 will do, we can emulate this mov

4. API ADDRESS

```
UPX1:01011787 FF 96 AC 1D 01+                 call    dword ptr [esi+11DACh] ;
GETPROCADDRESS
UPX1:0101178D 09 C0                           or      eax, eax        ; ADDRESS
UPX1:0101178F 74 07                           jz      short loc_1011798
UPX1:01011791 89 03                           mov     [ebx], eax      ; WRITE TO THUNK
UPX1:01011793 83 C3 04                        add     ebx, 4
UPX1:01011796 EB E1                           jmp     short BUILD_THUNK
```

the address goes into eax after getprocaddress so im going to choose

01011796 EB E1   jmp   short BUILD_THUNK   for my hook and update eip

with the address of BUILD_THUNK to simulate the jump when im done.

Coding the program

Ok i think the important bit is over, now we need to code this idea, now im no coding teacher, but perhapes for some of you coding is new, and its important to find a language your going to be happy learning and using, whilst coding the program you would normally code small sections first and test them but since its going to be hard to put this down on paper, im now going to paste my source code file in sections and explain it as much as i can, you may learn to code in a simular style to try port the idea to another language, or perhapes your an excellent coder anyway, but i never assume anything :-)

ok the source file is upx_dump.asm you should open this as i go through it, the first top bit is just the defining of some APIS and Constants, then the .data section sets up some variables we need, we will see them in use as we go along.

The first part is that we Load the notepad upx file but in suspended mode, this means the program isnt running but all of its memory is mapped.

now we patch our EB FE into all the addresses that we decided on eariler, take a look at the code below, if your new to using these apis you should look at the MSN links i provided earlier which show what all the parameters are, but it should be fairly straight forward.

```
---------------------------------------------------------------------------
Call CreateProcessA,o progname,0,0,0,0,CREATE_SUSPENDED,0,0,o tStartupInfo,o tProcessInfo

mov eax, 0101179Eh      ; OEP
call  WriteProcessMemory,[tProcessInfo],eax,o HALT_CODE,HALT_SIZE,0

mov eax, 0101176Ch      ; DLL NAME HOOK
call  WriteProcessMemory,[tProcessInfo],eax,o HALT_CODE,HALT_SIZE,0

mov eax, 01011780h      ; ASCII NAME HOOK
call  WriteProcessMemory,[tProcessInfo],eax,o HALT_CODE,HALT_SIZE,0

mov eax, 01011796h      ; API ADDRESS HOOK
call  WriteProcessMemory,[tProcessInfo],eax,o HALT_CODE,HALT_SIZE,0
--------------------------------------------------------------------------------
```

ok so now our process is loaded and we our hooks patched in.
The next stage is let the process run, then code a MAIN BODY
which will be a loop where GetThreadContext is constantly
called, GetThreadContext will retrieve all the running processes
registers, so if we are calling this in a loop we can monitor
when EIP hits one of our hooks then take action, easy eh?
ok here it is.

```
-------------------------------------------------------------------
call  ResumeThread, [tProcessInfo+4]

Call Sleep, 100h

mov     [my_context], 00010000h+1+2+4+8+10h   ; SET UP PERMISSIONS

ContextLoop:
    call    GetThreadContext, [tProcessInfo+4], o my_context
    test    eax, eax
    jz      CERR
    mov     eax, [my_context+REG_EIP]

    cmp eax, 0101179Eh          ; CHECKING EIP
    jz  OEP_REACHED

    cmp eax, 0101176Ch
    jz  DLLNAME_HOOKED

    cmp eax, 01011780h
    jz  ASCIINAME_HOOKED

    cmp eax, 01011796h
    jz  APIADDR_HOOK

jmp     ContextLoop
-------------------------------------------------------------------
```

pretty straight forward i think that is, now something to note is that i've hardcoded the addresses, perhapes sometimes it is best to subtract the imagebase then get the imagebase of the running program and add them to our values just in case of relocation, this would be essential for example if we had hooked after some loadlibrary and got the base address and were planning to place more hooks in this dll, but anyway i kept it simple.

Now we have a main body, now run through the process in your head, the first thing that will happen is we will get a hooked detected at the DLLNAME, since if you checked the upx code snipped at the start the first thing upx does is load a dll, so lets code the DLLNAME_HOOKED procedure.

```
;--------------------------------------------------------------
DLLNAME_HOOKED:

call  SuspendThread, [tProcessInfo+4]
call  GetThreadContext, [tProcessInfo+4], o my_context

mov eax, [my_context+REG_EAX]                                  ; GET THE CONTENTS OF
EAX(PTR TO ASCII DLL)
call  ReadProcessMemory,[tProcessInfo],eax,o myBuffer,30,0      ; READ DLL NAME FROM PTR

; emulate UPX1:0101176C                add     ebx, esi

mov ebx, [my_context+REG_EBX]
mov esi, [my_context+REG_ESI]
add ebx, esi
mov [my_context+REG_EBX], ebx

; skip instruction
mov eax, [my_context+REG_EIP]
add eax, 2
mov [my_context+REG_EIP], eax

; set context
call  SetThreadContext, [tProcessInfo+4], o my_context

call  ResumeThread, [tProcessInfo+4]

call dll1
db 13,10,13,10,'-> Loading DLL ',0
dll1:
call dbg_string
call dbg_string, o myBuffer
call dbg_string, o newline
jmp ContextLoop
;--------------------------------------------------------------

UPX1:0101176C 01 F3          add     ebx, esi
UPX1:0101176E 50             push    eax             ; DLL NAME
```

Ok here we stop the process with suspendthread so we stop the cpu going crazy, then we get the context so have all the current registers, now the dll name is stored in EAX, so we read this value from the context structure, now we have a pointer to the dllname in the other process, so we read from this address into a buffer.

Next we need to fix the instruction we destroyed which was ADD EBX, ESI, now if you never needed to hook this point again you could patch the instruction back but since we want to break here again we must emulate it, so i grab ebx and esi from the context struct add them and insert it back into ebx, then i also get the eip value and add 2, this is so we skip the EB FE and start at the PUSH, then i use SetThreadContext to update the process's memory, ResumeThread then sets it back on its way, i've then used my own internal functions dbg_xx to write out text and the contents of the buffer into a file called debug.txt. Now we jump back to our main context checking loop.

The next thing that will happen is we'll break on the ASCIINAME_HOOKED, so lets code that, you can almost copy paste the above function and make minor tweaks.

```
;--------------------------------------------------------------
ASCIINAME_HOOKED:
call  SuspendThread, [tProcessInfo+4]
call  GetThreadContext, [tProcessInfo+4], o my_context

mov eax, [my_context+REG_EDI]                                 ; GET THE CONTENTS OF EDI(PTR TO
ASCII API)
call  ReadProcessMemory,[tProcessInfo],eax,o myBuffer,200,0     ; READ DLL NAME FROM PTR

; emulate UUPX1:01011780 89 F9   mov     ecx, edi

mov edi, [my_context+REG_EDI]
mov [my_context+REG_ECX], edi

; skip instruction
mov eax, [my_context+REG_EIP]
add eax, 2
mov [my_context+REG_EIP], eax

; set context
call  SetThreadContext, [tProcessInfo+4], o my_context

call  ResumeThread, [tProcessInfo+4]

call dll2
db ' FUNC: ',0
dll2:
call dbg_string
call dbg_string, o myBuffer
jmp ContextLoop
;--------------------------------------------------------------

UPX1:01011780 89 F9           mov     ecx, edi
UPX1:01011782 57              push    edi             ; PTR ASCII NAME
```

Ok so the same as before, stop proces, get the pointer to the ascii name from edi and read it into our buffer, now i emulate the MOV ECX, EDI and update EIP

Next is the api function address hook APIADDR_HOOK

```
;---------------------------------------------------------------
APIADDR_HOOK:
call  SuspendThread, [tProcessInfo+4]
call  GetThreadContext, [tProcessInfo+4], o my_context

mov eax, [my_context+REG_EAX]                                   ; GET THE CONTENTS OF
EDI(PTR TO ASCII API)

call dll3
db 9,9,9,'ADDR: ',0
dll3:
call dbg_string
call dbg_dword,eax,0
call dbg_string, o newline

; emulate 01011796 EB E1    jmp     short BUILD_THUNK

mov eax, 01011779h
mov [my_context+REG_EIP], eax


; set context
call  SetThreadContext, [tProcessInfo+4], o my_context

call  ResumeThread, [tProcessInfo+4]
jmp ContextLoop
;---------------------------------------------------------------
```

```
UPX1:01011787 FF 96 AC 1D 01+          call    dword ptr [esi+11DACh] ; GETPROCADDRESS
UPX1:0101178D 09 C0                    or      eax, eax        ; ADDRESS
UPX1:0101178F 74 07                    jz      short loc_1011798
UPX1:01011791 89 03                    mov     [ebx], eax      ; WRITE TO THUNK
UPX1:01011793 83 C3 04                 add     ebx, 4
UPX1:01011796 EB E1                    jmp     short BUILD_THUNK
```

since the api address is EAX all i need do is get the value from the context structure, then we had our hook at 01011796, so i emulate the 'jmp   short BUILD_THUNK' by placing the address of BUILD_THUNK into EIP and continue.

Ok and last of all we need some code for OEP_REACHED

```
;-------------------------------------------------------------
OEP_REACHED:

call MessageBoxA,0,o msgOEP,o msgok, 0

call TerminateProcess, [tProcessInfo]


jmp End_Process
;-------------------------------------------------------------
```

just a simple messagebox to say hello :)
and the end of the code looks like,

```
;---------------------------------------------------------------
CERR:
        call MessageBoxA,0,o msgcontext,o msgerr, 0

End_Process:
call exitprocess, 0


end main
;---------------------------------------------------------------
```

ta da! and thats it, now since we are messing around with a program during a small loop that resolves the imports it considerably slows the app down, if you test the example it will take about 1 minute until the message box appears, click ok then view debug.txt

I've provided ASM and CPP code, and both compiled exes for you to test, the CPP one seems to run much faster, it also screen output, reading the CPP code is probably easier to understand than the ASM as you can see the program structure much better.

Now dont take this tutorial as a literal way of cracking something, it merely describes a common technique used my dumpers, you should reverse your target application and find good hook points, like after some decryption, then make use of your dumper to run through the target collecting information needed for a final unpacked target,
so have fun and watch out for CRCs ;-)

regards,
     yates.

yates@reverse-engineering.info
26/June/04

Republished - 12th November 2007 – Robert Yates

Republished - 12th November 2007 – Robert Yates