Reverse Engineer's Cookbook

Toorcon Seattle 2008

Aaron Portnoy [1] Cameron Hotchkies [2]

[1] aportnoy@tippingpoint.com

[2] chotchkies@tippingpoint.com

Toorcon Seattle April 19 2008

About Us

- Work at TippingPoint's Digital Vaccine Labs
  - Responsible for vuln-discovery, patch analysis, product security
  - Keep tabs on us at http://dvlabs.tippingpoint.com
- Authors and contributors to:
  - Sulley Fuzzing Framework
  - PaiMei Reverse Engineering Framework
  - PyMSRPC Toolset
- Side projects:
  - XSO - OS X Reversers: http://0x90.org/mailman/listinfo/xso

Talk Outline

- Interacting with IDA
  - Available functions and data types
  - Resources
- Monkey Work
  - Restructuring your .idb
  - Makes next steps more meaningful
- Organizing data for analysis
  - Creating data structures you can analyze
- Using those data structures to:
  - Locate recursion
  - Traverse function or basic block paths
  - Find specific functions/instructions/libcalls
  - ...and more

Scripting in IDA

- Multiple interfaces to IDA
  - Plugins (C++)
  - IDC (C-like scripting)
  - IDAPython (python)
  - idarub (ruby, abandonware)
- We are only focusing on IDAPython
  - many IDC and the IDA SDK API functions are exposed
  - allows for python language features and libraries

Exposed IDAPython functionality

- idautils - high level stuff
  - CodeRefsTo()
  - Functions()
  - Segments()
  - ...
- idaapi - lower level stuff
  - get_func()
  - isCode()
  - ...
- idc - wrappers to IDA's IDC functions
  - AskYN()
  - DnextB()
  - SetColor()
  -

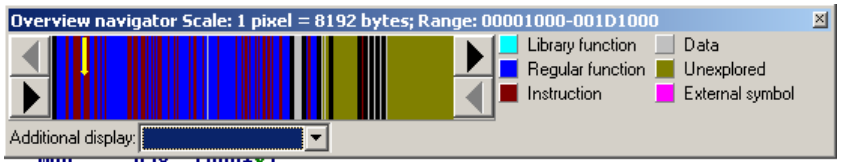
IDAPython Resources

- Hit F1 in IDA - search for IDC language
- IDAPython: http://www.d-dome.net/idapython/reference/
- IDA SDK http://www.openrce.org/reference_library/ida_sdk
- C:\PATH_TO_IDA\idc\idc.idc
- Header files from the SDK

Monkey Work

- Restructuring your database
  - IDA works, but isn't perfect
  - Misses vtables
  - Misses switch statements
  - Loses track off stack offsets
  - Misses whole functions

Functions found by IDA




```
__text:00027E09                 pop     esi
__text:00027E0A                 pop     ebp
__text:00027E0B                 retn
__text:00027E0B sub_27D62       endp
__text:00027E0B
__text:00027E0C ; ---------------------------------------------------------------------------
__text:00027E0C                 push    ebp
__text:00027E0D                 mov     ebp, esp
__text:00027E0F                 push    edi
__text:00027E10                 push    esi
__text:00027E11                 push    ebx
__text:00027E12                 sub     esp, 39Ch
__text:00027E18                 mov     eax, [ebp+8]
__text:00027E1B                 cmp     byte ptr [eax+10E73h], 0
__text:00027E22                 jz      short loc_27E3C
__text:00027E24                 mov     edx, [eax]
__text:00027E26                 mov     ecx, eax
__text:00027E28                 movzx   eax, byte ptr [eax+10E74h]
__text:00027E2F                 mov     [esp+4], eax
__text:00027E33                 mov     [esp], ecx
__text:00027E36                 call    dword ptr [edx+0FCh]
__text:00027E3C
__text:00027E3C loc_27E3C:                              ; CODE XREF: __text:00027E22↑j
__text:00027E3C                 mov     eax, [ebp+10h]
```

Simple define missed functions example

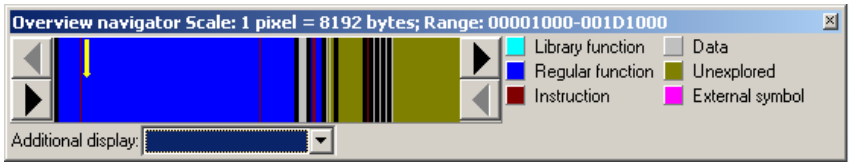
```
def rebuild_functions_from_prologues():
    seg_start = SegByName(".text")
    seg_end = SegEnd(seg_start)

    cursor = seg_start

    while cursor < seg_end:
        cursor = find_not_func(cursor, 0x1)

        # push EBP; mov EBP,ESP
        if (Byte(cursor) == 0x55 and Byte(cursor+1) == 0x89 and Byte(cursor+2)==0xE5):
            MakeFunction(cursor, BADADDR)
        else:
            cursor = FindBinary(cursor, 0x1, "55 89 E5", 16)
            if (GetFunctionName(cursor) == ""):
                MakeFunction(cursor, BADADDR)
```

Functions found by helper script




```
__text:00027E0C sub_27E0C       proc near
__text:00027E0C
__text:00027E0C var_3A8         = dword ptr -3A8h
__text:00027E0C var_3A4         = dword ptr -3A4h
__text:00027E0C var_3A0         = dword ptr -3A0h
__text:00027E0C var_39C         = dword ptr -39Ch
__text:00027E0C var_390         = dword ptr -390h
__text:00027E0C var_389         = byte ptr -389h
__text:00027E0C var_388         = dword ptr -388h
__text:00027E0C var_384         = dword ptr -384h
__text:00027E0C var_380         = dword ptr -380h
__text:00027E0C var_37C         = dword ptr -37Ch
__text:00027E0C var_350         = word ptr -350h
__text:00027E0C var_250         = dword ptr -250h
__text:00027E0C var_144         = dword ptr -144h
__text:00027E0C var_140         = dword ptr -140h
__text:00027E0C var_13C         = dword ptr -13Ch
__text:00027E0C var_38          = dword ptr -38h
__text:00027E0C var_34          = dword ptr -34h
__text:00027E0C var_30          = dword ptr -30h
__text:00027E0C var_2C          = dword ptr -2Ch
__text:00027E0C var_28          = dword ptr -28h
__text:00027E0C var_24          = dword ptr -24h
__text:00027E0C var_20          = dword ptr -20h
__text:00027E0C var_1C          = dword ptr -1Ch
__text:00027E0C arg_0           = dword ptr  8
__text:00027E0C arg_8           = dword ptr  10h
__text:00027E0C
__text:00027E0C                 push    ebp
__text:00027E0D                 mov     ebp, esp
__text:00027E0F                 push    edi
__text:00027E10                 push    esi
__text:00027E11                 push    ebx
```

Building from symbols

- For OS X, various sources to automate names:
  - Objective-C stores metadata in the _OBJC segment of MACH-O binary
  - __class section contains class data
  - method names are stored in __inst_meth, __cls_meth, etc...
  - this takes a lot of guess work out of functions
- For Windows, you can use things like:
  - arguments to OutputDebugString
  - arguments to custom logging functions
  - PDB files, if you've got them

Objective C Metadata

```
dword_29EC80    dd 0                    ; DATA XREF: __class:stru_293740↑o
                dd 108h
                dd offset a_updatesearchi, offset aV8@04, offset sub_17705 ; "v8@0:4"
                dd offset aUndomanagerfor, offset a@12@04@8_0, offset sub_D1251 ; "undoManagerForMessageTransfer:"
                dd offset aSelectmessag_0, offset aV12@04@8, offset sub_10345C ; "selectMessagesForUndo:"
                dd offset aUnhidemessages, offset aV12@04@8, offset sub_103442 ; "unhideMessagesForMessageTransfer:"
                dd offset aHidemessagesfo, offset aV12@04@8, offset sub_103428 ; "hideMessagesForMessageTransfer:"
                dd offset aTransferDidcom, offset aV16@04@8@12, offset sub_D126B ; "transfer:didCompleteWithError:"
                dd offset a_reporterror, offset aV12@04@8, offset sub_103363 ; "_reportError:"
                dd offset a_transfermessa, offset aC28@04@8@12c16, offset sub_CE2A4 ; "c28@0:4@8@12c16c20c24"
                dd offset aSynchronouslym, offset aV16@04@8@12, offset sub_103257 ; "synchronouslyMarkAsNotJunkMail:inStores"...
                dd offset aMarkmessagesas, offset aV16@04@8@12, offset sub_103192 ; "markMessagesAsNotJunkMail:stores:"
                dd offset aMarkasnotjun_0, offset aV12@04@8, offset sub_103B24 ; "v12@0:4@8"
```

```
8056 out of 8059 functions are unnamed
[+] rebuilt from prologues
236 out of 8059 functions are unnamed
Retrieving information from the database... ok
```

One step forward...

While analyzing, it is frequently common to want to know where a variable value came from.

- Backtraces are tricky
- Do you want the IDA name of an operand?
- or the actual value?

There is no one single variable backtrace script that will work every time. They should be purpose dependent.

- If you are renaming variables, consider using OpAlt vs SetMemberName

Identify arguments

```
01C mov     [ebp+var_4], esi
01C mov     [esp+18h+msgSend_selector], eax
01C mov     [esp+18h+msgSend_recipient], ebx
01C call    _objc_msgSend   ; a = [[ebp+arg_0] textStorage]
01C mov     [esp+18h+msgSend_recipient], ebx
01C mov     esi, eax
01C mov     eax, ds:off_2887F0
01C mov     [esp+18h+msgSend_selector], eax
01C call    _objc_msgSend   ; a = [[ebp+arg_0] selectedRange]
01C mov     [esp+18h+msgSend_recipient], esi
01C mov     [esp+18h+var_10], eax
01C mov     eax, ds:off_288C58
01C mov     [esp+18h+var_C], edx
01C mov     [esp+18h+msgSend_selector], eax
01C call    _objc_msgSend   ; a = [eax attachmentsInRange:]
01C mov     ebx, [ebp+var_8]
```

Graphing overview

- Creating relationships
  - Code can be represented as a graph
  - To analyze it, we need downgraph/upgraph structures
  - We do this with IDAPython...

Generating graph structures

- We need parents and children
  - Functions()
  - CodeRefsTo()
  - Also need to parse the imports (.idata)

Now that we have a graph structure

- Lets do fun stuff..
  - Find all functions matching a given regular expression
  - Locate all recursive functions
  - Find all network and file I/O
  - Find all allocations
  - Find one or all paths from node A to node B

Applications for auditing

- Finding possible bugs
  - Bad allocations
  - Unsafe libcalls
  - Sign extensions
- We use backtracing to accomplish some of this
  - Example, "Was any math applied to this function argument?"

Demo

- Going to show how to use this
  - In IDA, hit Alt+9 to run our .py
  - Provides you with a 'here' object
  - Enumerates available methods using python's introspection
    - You can then use the scriptbox to do stuff like: here.find_func(".*str.*")
- Code will be available on http://dvlabs.tippingpoint.com/blog next week

Questions?

- Ask in the provided time following our talk
- Or e-mail us, aportnoy@tippingpoint.com, chotchkies@tippingpoint.com

Total Slide Count

21