Characterization of the Windows Kernel version variability for accurate Memory analysis.

Michael Cohen
scudette@google.com

DFRWS 2015 Dublin.
Memory Analysis overview

How do we analyse memory?

We need to emulate the way code looks at memory.
typedef unsigned char uchar;

enum {
    OPT1,
    OPT2
} options;

struct foobar {
    enum options flags;
    short int bar;
    uchar *foo;
}

It is generally not possible to predict the memory layout of a C struct without knowing external factors:

- Alignment
- Endianess
- Bit size (64/32 bit)
- Compiler
- Optimizations etc

Unless packed structs.
Example memory analysis technique

- Listing processes
  - Find the global kernel symbol "PsActiveProcessHead"
  - Follow the linked list _EPROCESS.ActiveProcessLinks to find all _EPROCESS structs.
  - Print _EPROCESS.ImageFileName
Data Structures

typedef unsigned char uchar;

enum {
    OPT1,
    OPT2
} options;

Debugging symbols contain the exact layout of all data structures.

Can use them to get struct offset AND kernel global constants.

struct foobar {
    enum options flags;
    short int bar;
    uchar *foo;
}

Google | Rekall
What do PDB files look like?

1. Each time the binary is built, a GUID is generated.
2. The debugging symbols are stored in a PDB file.
3. The executable is shipped.
4. PDB files can be made available publicly on a symbol server.
Historical perspective

- Older tools have profiles embedded inside the tool.
  - Profile is pre-generated from an exemplar of an OS version released.
    - e.g. Win7SP1x64
  - Profile is embedded inside the tool
  - We assume profile is applicable to all releases of this version.
  - Profile only contains structs - no use of global offsets from PDB file.
  - Global offsets are deduced by scanning.
Can we always guess kernel globals?

One-byte Modification for Breaking Memory Forensic Analysis

Takahiro Haruyama / Hiroshi Suzuki
Internet Initiative Japan Inc.
ADD -- Complicating Memory Forensics Through Memory Disarray

Jake Williams and Alissa Torres

In this presentation, we'll present ADD (attention deficit disorder), a tool that litters Windows physical memory with (configurable amounts and types of) garbage to disrupt memory forensics. Memory forensics has become so mainstream that it's catching too many malware authors during routine investigations (making Jake a sad panda). If memory forensics were much harder to perform, then attackers would retain an upper hand. ADD increases the cost of memory forensics by allocating new structures in memory that serve only to disrupt an investigation.

We'll present some basic memory forensics techniques (just to set the stage for those who aren't familiar with the concepts). We'll explain how volatility, a core memory forensics tool, actually performs its analysis. In particular, we'll show how it locates hidden processes, drivers, and modules.

Next, we'll show how running ADD on a machine under investigation completely changes the memory forensics landscape. We'll show how an investigator must weed through astounding numbers of false positives before identifying the investigation targets.

Finally, Alissa will show how all is not lost. Even though ADD may confuse junior analysts, she'll show the invariants in memory that analysts should always be able to come back to complete their forensic analysis.

Jake is the Chief Scientist at CSRgroup where he does lots of offensive and defensive research. He is also a SANS instructor and member of the DFIR author team. Occasionally, CSRgroup still lets Jake do penetration tests (where he feels like a kid in a candy store).

Alissa is a digital forensics examiner and incident response consultant for Sibertor Forensics. Also a SANS Instructor, she teaches hundreds of security professionals a year how to find evil in the form of trace artifacts and hidden processes.
Can we do better?

- Rekall chooses to rely on constants obtained from debugging symbols.
  - Pros
    - Better coverage of symbols, especially ones that are not exported.
  - Cons
    - We need to wait for Microsoft to make a pdb file available for us to use.
Let's evaluate this approach.

- Basic assumption in Volatility:
  - Struct layout does not change between major and minor versions.
    - An exemplar from a particular version will apply to all kernels from that version.
  - Kernel global symbols vary too much between major and minor versions to hard code
    - Therefore we need to scan for them.
Updated repo index, removed obsolete html files.

<table>
<thead>
<tr>
<th>File Path</th>
<th>Message</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>00625D7D36754CBEBA4533BA9A0F3FE22.gz</td>
<td>Initial push of profiles into the profile repository.</td>
<td>3 months ago</td>
</tr>
<tr>
<td>0100FCDAFD4049B8B06005EC97705A1F2.gz</td>
<td>Initial push of profiles into the profile repository.</td>
<td>3 months ago</td>
</tr>
<tr>
<td>01DDCBDB82AE46BEAFCD0C6A409E3B1D31...</td>
<td>Initial push of profiles into the profile repository.</td>
<td>3 months ago</td>
</tr>
<tr>
<td>01DF28C698D84DEBB1A74254C3AF800E2.gz</td>
<td>Initial push of profiles into the profile repository.</td>
<td>3 months ago</td>
</tr>
<tr>
<td>03185083233249D9BB747EA777B80C982.gz</td>
<td>Initial push of profiles into the profile repository.</td>
<td>3 months ago</td>
</tr>
<tr>
<td>04FB9A156FF44ECCA6EBCAE9617D8DB73.gz</td>
<td>Initial push of profiles into the profile repository.</td>
<td>3 months ago</td>
</tr>
<tr>
<td>05A6F49C5DD848FF983459421A79F1232.gz</td>
<td>Initial push of profiles into the profile repository.</td>
<td>3 months ago</td>
</tr>
<tr>
<td>06472CCD0ECF43B58D676891C6745DAC2.gz</td>
<td>Initial push of profiles into the profile repository.</td>
<td>3 months ago</td>
</tr>
<tr>
<td>0887873AD7FC4115AC9258B9871F81341.gz</td>
<td>Initial push of profiles into the profile repository.</td>
<td>3 months ago</td>
</tr>
<tr>
<td>08F4D00C3B54A4B34B2D3AE8402F927802.gz</td>
<td>Initial push of profiles into the profile repository.</td>
<td>3 months ago</td>
</tr>
</tbody>
</table>
Result

- Assumption is mostly validated for the kernel
  - Struct offsets do not vary per version.
  - Kernel constant very wildly.
- So the Volatility approach should work in most cases!
Issue 174: [profile offsets per build/revision number] was: Process Owner Info not found for TCP_ENDPOINT scan output on Vista (netscan)
8 people starred this issue and may be notified of changes.

Reported by welcome....@gmail.com, Jan 3, 2012

What steps will reproduce the problem?
1. Running netscan plugin on Vista SP2 dump
2. 
3.

What is the expected output? What do you see instead?
PID, Process name is not found TCPEndPoint scan output. However that information is present in Listener, UDP scan output.

<table>
<thead>
<tr>
<th>x0cf7c8de8 TCPv4</th>
<th>0.0.0.0:61618</th>
<th>0.0.0.0:0</th>
<th>LISTENING</th>
<th>2016</th>
<th>spoolsv.exe</th>
</tr>
</thead>
<tbody>
<tr>
<td>x01cedd88 TCPv4</td>
<td>0.0.0.0:57883</td>
<td>0.0.0.0:443</td>
<td>CLOSED</td>
<td>------</td>
<td>---------------</td>
</tr>
<tr>
<td>x014e85e08 TCPv4</td>
<td>10.130.179.251:64286</td>
<td>10.177.226.18:1533</td>
<td>ESTABLISHED</td>
<td>------</td>
<td>---------------</td>
</tr>
</tbody>
</table>

What version of the product are you using? On what operating system?
Using volatility 2 standalone python precompiled version on vista sp2.

Please provide any additional information below.
Here's a little more info.

6.0.6002.18272
Microsoft Windows Server 2008 Standard
6.0.6002 Service Pack 2 Build 6002
push    dword ptr [edi+164h]
call    ds:__imp_PsGetProcessId@4

6.0.6002.18519
Microsoft Windows Vista T Enterprise
6.0.6002 Service Pack 2 Build 6002
push    dword ptr [edi+164h]
call    ds:__imp_PsGetProcessId@4

6.0.6002.18005
Microsoft Windows Server 2008 Standard (also seen Microsoft Windows Vista T Ultimate)
6.0.6002 Service Pack 2 Build 6002
push    dword ptr [edi+160h]
call    ds:__imp_PsGetProcessId@4

As you can see, regardless of whether the OS identifies itself as "Server 2008" or "Vista" if the revision number > 18005 then the offset is 0x164. If the revision number <= 18005 then the offset is 0x160. I don't have tcpip.sys binaries for every revision, so I'm not sure if 18005 is exactly where the line is drawn.

Unfortunately until we can choose vtype offsets based on revision number (in addition to the major, minor, build, and memory model which is already possible), then there's not a good way to handle this. I'm not sure we should close this issue since currently we won't print process information for VistaSP2x86/Win2008SP2x86 whose revision numbers are > 18005. However, we might have to defer to fixing it later. In the meantime, manish, you'll have to use a version of volatility where you can change the vtype offset (and if needed build your own exe from it).
There are some problems

- Sometimes even struct layout changes within the major/minor version release cycle.
- Mr. Hale is an expert reverse engineer
  - He can figure out the correct struct layout by looking at the disassembly.
  - He knows how to change the program to account for this version.
  - We don't really know how to reproduce:
    - What function was reversed?
    - Which instructions should we look at?
We need automated reversing

● First approach:
  ○ Look at the data and surrounding context
analyze_struct "*win32k!grpWinStaList"

$0xfa800225ef60$ is inside pool allocation with tag 'Win\xe4' ($0xfa800225eed0$)

<table>
<thead>
<tr>
<th>Offset</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>Data:0x0</td>
</tr>
<tr>
<td>0x8</td>
<td>Data:$0xfa8001851a70$ Tag:Win\xe4 $0xfa8001851a70$</td>
</tr>
<tr>
<td>0x10</td>
<td>Data:$0xfa80022c4b30$ Tag:Des\xe8 $0xfa80022c4b30$</td>
</tr>
<tr>
<td>0x18</td>
<td>Data:$0xf960003af340$ Const:win32k!gTermIO</td>
</tr>
<tr>
<td>0x20</td>
<td>Data:0x0</td>
</tr>
<tr>
<td>0x28</td>
<td>Data:$0xf900c015c9f0$ Tag:Uskb $0xf900c015c9f0$</td>
</tr>
<tr>
<td>0x90</td>
<td>Data:0x0</td>
</tr>
<tr>
<td>0xa0</td>
<td>Data:$0x6f6d02580000$</td>
</tr>
<tr>
<td>0xa8</td>
<td>Data:$0x28b25eb$</td>
</tr>
<tr>
<td>0xb0</td>
<td>Data:$0xfa800225f010$ Empty Tag:moni $0xfa800225f010$</td>
</tr>
<tr>
<td>0xb8</td>
<td>Data:$0xfa8000927e70$ Tag:FxDr $0xfa8000927e70$</td>
</tr>
<tr>
<td>0xc0</td>
<td>Data:$0xf88000f0c4b0$</td>
</tr>
<tr>
<td>0xc8</td>
<td>Data:$0x303401002$</td>
</tr>
<tr>
<td>0xd0</td>
<td>Data:$0xfa8000927e70$ Tag:FxDr $0xfa8000927e70$</td>
</tr>
<tr>
<td>0xd8</td>
<td>Data:$0x10c1a$</td>
</tr>
<tr>
<td>0xe0</td>
<td>Data:$0xfa80009437b0$ _LIST_ENTRY $0xfa80009437b0$</td>
</tr>
<tr>
<td>0xe8</td>
<td>Data:$0xfa8000920460$</td>
</tr>
<tr>
<td>0xf0</td>
<td>Data:0x0</td>
</tr>
<tr>
<td>0xf8</td>
<td>Data:$0xfa8000919e70$ Tag:moni $0xfa8000919e70$</td>
</tr>
</tbody>
</table>
### Struct tagWINDOWSTATION

<table>
<thead>
<tr>
<th>field</th>
<th>offset</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>rpwinstaNext</td>
<td>0x8</td>
<td>['Pointer', {'target': 'tagWINDOWSTATION'}]</td>
</tr>
<tr>
<td>rpdeskList</td>
<td>0x10</td>
<td>['Pointer', {'target': 'tagDESKTOP'}]</td>
</tr>
<tr>
<td>pTerm</td>
<td>0x18</td>
<td>['Pointer', {'target': 'tagTERMINAL'}]</td>
</tr>
<tr>
<td>pGlobalAtomTable</td>
<td>0x78</td>
<td>['Pointer', {'target': '_RTL_ATOM_TABLE'}]</td>
</tr>
</tbody>
</table>

### Struct tagTHREADINFO

<table>
<thead>
<tr>
<th>field</th>
<th>offset</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>pEThread</td>
<td>0x0</td>
<td>['Pointer', {'target': '_ETHREAD'}]</td>
</tr>
<tr>
<td>GdiTmpTgoList</td>
<td>0x50</td>
<td>['_LIST_ENTRY']</td>
</tr>
<tr>
<td>ppi</td>
<td>0x158</td>
<td>['Pointer', {'target': 'tagPROCESSINFO'}]</td>
</tr>
<tr>
<td>pq</td>
<td>0x160</td>
<td>['Pointer', {'target': 'tag0'}]</td>
</tr>
<tr>
<td>spklActive</td>
<td>0x168</td>
<td>['Pointer', {'target': 'tagKL'}]</td>
</tr>
<tr>
<td>rpdesk</td>
<td>0x178</td>
<td>['Pointer', {'target': 'tagDESKTOP'}]</td>
</tr>
<tr>
<td>PtiLink</td>
<td>0x260</td>
<td>['_LIST_ENTRY']</td>
</tr>
</tbody>
</table>

### Struct tagDESKTOP

<table>
<thead>
<tr>
<th>field</th>
<th>offset</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>rpdeskNext</td>
<td>0x18</td>
<td>['Pointer', {'target': 'tagDESKTOP'}]</td>
</tr>
<tr>
<td>rpwinstaParent</td>
<td>0x20</td>
<td>['Pointer', {'target': 'tagWINDOWSTATION'}]</td>
</tr>
<tr>
<td>_hsectionDesktop</td>
<td>0x78</td>
<td>['Pointer', {'target': '_SECTION_OBJECT'}]</td>
</tr>
<tr>
<td>PtiList</td>
<td>0xa8</td>
<td>['_LIST_ENTRY']</td>
</tr>
</tbody>
</table>
2: Reverse Engineering code

- We need a reproducible and robust reverse engineering method
  - Expert makes the initial reversing analysis
  - Machine parseable method of documenting the finding.
  - Repeatable analysis on similar code variants.
tagDESKTOP:
  PtList:
    - - Disassembler
    - rules:
      - MOV $var1, *grpdeskRitInput
      - TEST $var1, $var1
      - MOV $var1, [$var1+$rpwinstaParent]
      - MOV $pdesk, [$var1+$rpdeskList]
      - LEA *, [$pdesk+$out]
    start: win32k!SetGlobalCursorLevel
    target: Pointer
    max_separation: 300

pheapDesktop:
  - - Disassembler
  - rules:
    - MOV $var1, [*+$out]
    - CALL *RtlAllocateHeap
    start: win32k!DesktopAlloc
    target: Pointer

rpdeskNext:
  - - Disassembler
  - rules:
    - MOV $var1, [$var2+$out]
    - TEST $var1, $var1
    - JZ *
    - MOV *CX, $var1
    - CALL *ObQueryNameInfo
    start: win32k!ParseDesktop
    target: Pointer
ERROR:root:Failed to find match for tagDESKTOP.rpdNext.
DEBUG:root:Unable to find tagDESKTOP.rpdNext via Disassemble win32k!ParseDesktop
DEBUG:root:Found match for tagDESKTOP.rpdNext
DEBUG:root:... 0xf9600001c454 MOV RCX, [RSI+0x18]
DEBUG:root:... 0xf9600001c4549 0x51 4885c9 TEST RCX, RCX
DEBUG:root:... 0xf9600001c4553 0x5b ff15c7bala0 CALL QWORD [RIP+0x1abac7] 0xfffff80002695fe0 win32k!imp_ObjDereferenceObject ->
DEBUG:root:Found match for tagDESKTOP.PtlList
DEBUG:root: 0xf960000206452 MOV RAX, [RIP+0x1a9c57] 0xfffffa80022c4b30 win32k!grpdeskRitInput
DEBUG:root: 0xf96000020645b 0xf 4885c0 TEST RAX, RAX
DEBUG:root: 0xf960000206460 0x14 4885420 MOV RAX, [RAX+0x28]
DEBUG:root: 0xf960000206464 0x18 488b510 MOV RDX, [RAX+0x18]
DEBUG:root:... 0xf960000206464a 0xe 4c8d82a8000000 LEA R8, [RDX+0x8a]
ERROR:root:Failed to find match for tagDESKTOP.rpwinstaParent.
DEBUG:root:Unable to find tagDESKTOP.rpwinstaParent via Disassemble win32k!SetGlobalCursorLevel
DEBUG:root:Found match for tagDESKTOP.pheapDesktop
DEBUG:root:... 0xf9600001924a5 0x11 488b9980000000 MOV RCX, [RCX+0x80]
DEBUG:root:... 0xf9600001924b1 0x1d ff1521e01d00 CALL QWORD [RIP+0x1de021] 0xfffff8000265e840 win32k!imp_RtlAllocateHeap -> nt!RtlAllocateHeap
[tagDESKTOP tagDESKTOP] @ 0x00000000
 0x00 rpwinstaParent <None Pointer to [x0000000000] (rpwinstaParent)>
 0x18 rpdNext <None Pointer to [x0000000000] (rpdNext)>
 0x80 pheapDesktop <None Pointer to [x0000000000] (pheapDesktop)>
 0xA8 PtlList <None Pointer to [x0000000000] (PtlList)>
Repeat analysis with undocumented structures
Scanning for constants in undocumented code is extremely difficult.
- Often requires the automated disassembly of function call sites.
- Often loses context (e.g. which session does this win32k object belong to?).
- Quite slow.

We want to be able to use known versions
- Just extract constants from the PDB files.
What does a Rekall profile look like?

- File is a JSON data structure.
- Divided into Sections:
  - $CONSTANTS
  - $FUNCTIONS
  - $METADATA
  - $STRUCTS

```json
{"$CONSTANTS": {
  "CmNtCSDVersion": 718856,
  ...
},

"$ENUMS": {
  "BUS_QUERY_ID_TYPE": {
    "0": "BusQueryDeviceID",
    "1": "BusQueryHardwareIDs",
    ...
  },

"$FUNCTIONS": {
  "ADD_MAP_REGISTERS": 606670,
  ...
},

"$METADATA": {
  "ProfileClass": "Nt",
  "arch": "I386"
  ...
},

"$STRUCTS": {
  "BATTERY_REPORTING_SCALE": [8, {
    "Capacity": [4, ["unsigned long", {}]], ...
  }

```
$STRUCT section.

```
{ "_EPROCESS": [624, {  
  "AccountingFolded": [548, ["BitField", {  
    "end_bit": 2,  
    "start_bit": 1,  
    "target": "unsigned long"  
  }]],  
  "ActiveProcessLinks": [160, ["_LIST_ENTRY", {}]],  
  "ActiveThreads": [376, ["unsigned long", {}]],  
  "AddressCreationLock": [232, ["_EX_PUSH_LOCK", {}]],  
  "AddressSpaceInitialized": [552, ["BitField", {  
    "end_bit": 12,  
    "start_bit": 10,  
    "target": "unsigned long"  
  }]],  
  "AffinityPermanent": [548, ["BitField", {  
    "end_bit": 19,  
    "start_bit": 18,  
    "target": "unsigned long"...```
$CONSTANTS and $FUNCTIONS

"NtAlpcSendWaitReceivePort": 2207436,
"NtAlpcSetInformation": 1805611,
"NtApphelpCacheControl": 2308968,
"NtAreMappedFilesTheSame": 2367400,
"NtAssignProcessToJobObject": 1912487,
"NtBuildGUID": 411132,
"NtBuildLab": 410688,
"NtBuildLabEx": 410912, ...

- These addresses come directly from Microsoft Debugging symbols.
  - Identical to the way the kernel debugger works.
  - No need to scan, guess or otherwise deduce symbol addresses.
Rekall Profiles - JSON files

- A profile file is a data structure which represents all the information needed to parse OS specific memory.
  - Files are stored in the public profile repository:
    - [http://profiles.rekall-forensic.com](http://profiles.rekall-forensic.com)
  - Windows Profiles are identified by GUID.

Revision c39b14f8dca9: /nt/GUID

Profiles for nt kernel are stored here.
Every single kernel build has a unique GUID.
Profile Indexes

Problem statement: Set membership:
● Given a set of binaries, is this binary in the set, and which one is it?

Solution:
● Generate a sample of data points in each binary and build a decision tree.
Conceptual overview

Known Data

Binary mapped into memory image
Potential Complications

- Not all pages in binary are always readable:
  - Must discount comparison points in unreadable pages -> weakens signatures.
- How many points should we use?
- Sometimes we do not have the actual binary - we only have the binary GUID.
  - Deduce data in binary purely from symbol information:
    - Functions have known preamble.
    - String constants have debug symbols.
Symbol Name (Usually mangled)

Probably what the address contains.

Before each function - NOP slide.
Result index

{ "$INDEX": {
  "nt/GUID/00625D7D36754CBEBA4533BA9A0F3FE22": [[2038160, ["4b65726e656c5370616365"]], [3601204, ["4952505f4d4e554552595f444556494345545854"]], [253980, ["4100500050005f004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004n"], [120086, ["90"]], [2559256, ["90"]], [137962, ["90"]], [2500200, ["90"]], [2569084, ["90"]], [206055, ["90"]], [630264, ["90"]], [10\4589, ["90"]],
  "nt/GUID/0100FCDAFD4049B8B06005EC07705A1F2": [[463544, ["5a775175657279496e666f726d6174696f6e4669c65"]], [2197832, ["4952505f4d4e554552595f444556494345545854"]], [253980, ["4100500050005f004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004e004n"], [120086, ["90"]], [2559256, ["90"]], [137962, ["90"]], [2500200, ["90"]], [2569084, ["90"]], [206055, ["90"]], [630264, ["90"]], [10\4589, ["90"]],
"Profile Name": "Offset in binary."
"Expected data."}
Conclusions

- Validating assumptions about kernel versions.
  - Will our analysis work in every case?
    - Maybe not
- Develop methods for automated reverse engineering
  - Helps to document expert effort.
  - Helps to repeat on many samples.
- Towards fully automated Linux profile generation!
  - Given a binary kernel image, calculate the correct profile automatically.
Sorry, Quaid. Your whole life is just a dream.

See you at the party, Richter!